Today's TBB

スレッディング・ビルディング・ブロックを使用した
C++ 並列プログラミング

—

第 2 版

—

Michael J. Voss
James R. Reinders

iSUS 編集部 訳

Apress
open

Today's TBB

スレッディング・ビルディング・ブロックを使用した C++ 並列プログラミング

第 2 版

Michael J. Voss
James R. Reinders

iSUS 編集部 **訳**

Apress open

Today’s TBB: スレッディング・ビルディング・ブロックを使用した C++ 並列プログラミング 第 2 版

Michael J. Voss
Austin, TX, USA

James R. Reinders
Portland, OR, USA

ISBN-13 (pbk): 979-8-8688-1269-9
ISBN-13 (電子版): 979-8-8688-1270-5
https://doi.org/10.1007/979-8-8688-1270-5

Apress Media LLC マネージング・ディレクター: Welmoed Spahr
編集者: Susan McDermott
開発編集者: Laura Berendson
プロジェクト・マネージャー: Jessica Vakili

Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004 により世界中の書籍業界に配布されます。電話 1-800-SPRINGER、ファックス (201) 348-4505、e-mail: orders-ny@springer-sbm.com または www.springeronline.com。Apress Media, LLC はカリフォルニア州の LLC であり、唯一のメンバー (所有者) は Springer Science + Business Media Finance Inc (SSBM Finance Inc) です。SSBM Finance Inc はデラウェア州の法人です。

翻訳に関する情報については、booktranslations@springernature.com まで電子メールでお問い合わせください。再版、ペーパーバック、またはオーディオの権利については、bookpermissions@springernature.com まで電子メールでお問い合わせください。

Apress のタイトルは、学術的、企業的、または販促的な用途で大量に購入できます。ほとんどのタイトルには電子書籍版とライセンスも用意されています。詳細については、当社の印刷物および電子書籍の大量販売ウェブページ (http://www.apress.com/bulk-sales) を参照してください。

本書を廃棄する場合は、紙をリサイクルしてください。

私たちは本書を、尊敬する同僚である Rafael Asenjo の思い出に捧げます。彼の知恵、優れた指導力、そして素晴らしいユーモアのセンスが深く惜しまれます。

目次

著者紹介

Michael J. Voss は、インテルのミドルウェア・アーキテクチャー担当シニア・プリンシパル・エンジニアです。彼は、2006 年の 1.0 リリース以前から TBB 開発チームの主要メンバーでした。Michael は、並列プログラミングに関する 40 以上の論文や記事を共同執筆しており、さまざまな分野の顧客と頻繁にコンサルティングを行い、顧客がアプリケーションで並列処理を効果的に活用できるよう支援しています。彼は ISO C++ 委員会（WG21）のメンバーであり、ライブラリーと並行性に関する議論に参加しています。2006 年にインテルに入社する前、Michael はトロント大学のエドワード S. ロジャースシニア電気/コンピューター工学部の准教授を務めていました。彼はパデュー大学の電気/コンピュータ工学部で博士号を取得しました。

James R. Reinders は、インテルで 40 年以上にわたって輝かしいキャリアを築きました。並列コンピューティングに関する豊富な経験を持つ James は、並列プログラミングに関する 12 の技術書籍の執筆、共著、編集に携わっており、その他多数の書籍にも寄稿しています。
ミシガン大学の工学プログラムを卒業した彼は、システムの最適化と教育に深い情熱を持っています。James は、TOP500 リストで 1 位を獲得した世界最速のコンピューター 2 台の開発に貢献する栄誉に恵まれました。また、他の多くのスーパーコンピューターやソフトウェア開発ツールにも貢献しています。
本書が完成し、ジェームズはインテルを退職し、現在はオレゴンでゆったりとした生活を楽しんでいます。

謝辞

Michael は、妻の Natalie、そして子供たちの Nick、Luke、Alexandra（そして彼女の夫 Royce）の揺るぎないサポートに心から感謝しています。

James は、この研究の実現に多大なる貢献をしてくれた妻の Susan Meredith に深い感謝の意を表しています。

また、長年にわたり TBB に貢献してくださったすべての方々、特にライブラリーの充実に多大な努力を払ってくださったインテルの多くの開発者の方々に感謝します。私たちのレビュアーは、TBB ユーザーと主要な開発者の優れたグループであり、この作業を大幅に強化する貴重なフィードバックを提供しました。多くの方々からいただいた貴重なアドバイスと洞察に感謝いたします: Ruslan Arutyunyan、Raja Bala、Piotr Balcer、Karolina Bober、Konstantin Boyarinov、Jake Chuang、Chunyang Dai、Lukasz Dorau、Mikhail Dvorskiy、Alexandra Epanchinzeva、Andrey Fedorov、Aleksei Fedotov、Elvis G. Fefey、Adam Fidel、J. Daniel Garcia、Wenju He、Dan Hoeflinger、Ilya Isaev、Shweta Jha、Seung-Woo Kim、Kyle Knoepfel、Alexandr Konovalov、Sergey Kopienko、Alexey Kukanov、Pavel Kumbrasev、Mark Lubin、Olga Malysheva、Matthew Michel、Dmitri Mokhov、Rob Mueller-Albrecht、Sarath Nandu R.、Eric L. Palmer、Arun Parkugan、Marc F. Paterno、Lukasz Plewa、Pablo Reble、Ekaterina Semenova、Deepanshi Sharma、Pawel Skowron、Timmie Smith、Tobias Weinzierl、Anuya Welling、Alex M. Wells、および Marek Wyszumirski。ご協力いただいた皆様に心より感謝申し上げますとともに、記載漏れがありましたらお詫び申し上げます。

並列パターンと、並列コードを記述するプログラマーとして最も効果的な方法を説明する上での並列パターンの役割に関して、励ましとアドバイスをくれた Tim Mattson に感謝します。

長年にわたる揺るぎないサポートと時折の優しい助言をいただいた Sanjiv Shah、Joe Curley、Cristina Belidica、Herb Hinstorff に、心から感謝します。

最後に、契約、編集、制作を通じてこの本を導いてくれた Apress チーム全員の献身的な努力に感謝したいと思います。ありがとう、皆さん。

Mike Voss と James Reinders

序文
はじめに

この序文では、1 章でスレッディング・ビルディング・ブロック（TBB）の解説を始めるにあたり、念頭に置くべき重要な並列プログラミングの概念を確認します。

Think Parallel（並列に考える）: 本書は、並列プログラミングの初心者と熟練した専門家の両方に価値があるものになるよう考慮されています。C プログラミングにのみ慣れている場合でも、最新の C++ に精通している場合でも、この本は理解しやすく、役立ちます。

本書では、内容を過度に単純化することなく、多様な読者層に対応するため、この序文を執筆し、共通の立場で学習できるよう配慮しました。読者の皆さんは、次の章に進む前に、この章を一読されることを強くお勧めします。

ここをブックマーク

TBB は長年開発が続けられてきたため、TBB に関する以前の 2 冊の書籍を含め、大量の古い資料がオンライン上に存在します。古いリソースの大部分は正確ですが、微妙な違いが勘違いの原因となる場合があります。特に、進化する C++ 標準に準拠するため変更が反映されている場合、その傾向が強くなります。2006 年に TBB がデビューした頃は C++03 が標準でしたが、この言語は大幅に進化しており、アップデートされた資料を参照することが重要です。

TBB と並列プログラミングに精通したら、詳細なインターフェイス情報と高度な制御に関する最適なオンラインリソースは https://tinyurl.com/tbbdoc と https://tinyurl.com/tbbspec にあります。これほど包括的かつ最新のオンラインリソースは他にありません。本書の学習を補うため、ブックマークすることをお勧めします。

TBB を指導する際には、[X . Y]（例: [algorithms.parallel_for]）という表記を使用して、関連するオンライン仕様が記載されている仕様ページ（https://tinyurl.com/tbbspec）を参照します。タイトル X（例: アルゴリズム）は、oneTBB インターフェイスまたは oneTBB 補助インターフェイスのいずれかにあり、そこから Y（例: parallel_for）のリンクが存在するページに移動します。

さらに、本書全体の図に使用されているソースコードにアクセスするには、https://tinyurl.com/tbbBOOKexamples をブックマークしてください。コードは本書に記載されているとおりに提供されており、将来サンプルが改訂された場合に更新を見つけるリソースも提供されています。すべてのコードはダウンロード可能です。

TBB と oneTBB とは？

TBB は、C++ で並列プログラムを作成する強力なソリューションであり、この言語の並列プログラミングに対する最もポピュラーで包括的なサポートを提供します。TBB が初めて導入されたとき、タスク・スチール・スケジューラー、並列処理を考慮したメモリー管理、高度な並行コンテナーなどのコア機能に加えて、ポータブルアトミックなど当時の C++ にはなかった機能も提供されました。TBB は、最初のドキュメントから、これらの機能を C++ 標準に組み込むことを目標としています。

2019 年、TBB は、進化した C++ 標準に組み込まれた機能のサポートを廃止することで、提供内容を合理化する大胆な決断を下しました。この取り組みは、TBB の核となる強みであるスケジューリング、メモリー管理、スケーラブルな並列プログラミングのためのコンテナーに重点を置いて、「今日の C++」を補完するように TBB を適応させたものと考えることができます。oneTBB 名称変更されましたが、最新の C++ 基盤に依存していることを除けば、オリジナルの TBB と基本的な違いはありません。名称が変更されても、インターフェイス名と名前空間は `tbb` のままで、`oneapi::tbb` が `tbb` のエイリアスとして定義されているため、一般には引き続き TBB と呼ばれます。
どちらを使用しても問題ありません。同様に、ヘッダーファイルのパスは変更されませんが、必要に応じて `oneapi/` をパスに追加できるリダイレクト・ヘッダー・セットもあります。
TBB のドキュメントと仕様には、多くの場合、新しいトップレベルの oneapi 名前空間内にインターフェイスが示されていますが、oneTBB 仕様のセクション [`configuration.namespaces`] には、「`oneapi::tbb` 名前空間は、`tbb` 名前空間のエイリアスと見なすことができます」と記載されています。本書では、`tbb` 名前空間を直接使用し、インクルード・ファイルのパスでは `tbb` 名前空間のみを使用します。

oneAPI は、UXL（Unified Acceleration）Foundation によって管理されるコミュニティーです。UXL の恩恵を受ける oneTBB を含む複数の oneAPI プロジェクトがあります。

TBB は C++ で並列プログラムを作成する最も効果的な方法であり、一部の計算をアクセラレーターにオフロードする場合でもホスト CPU の能力を最大限に活用できると考えています。私たちの目標は、TBB を活用して生産性を向上できるよう、皆さんを支援することです。

本書の構成と序文

本書は次の節で構成されています:

- はじめに: この節では、本書の残りの部分を理解するのに必要な背景と基礎について説明します。これには、TBB 並列プログラミング・モデルの動機、並列プログラミングの紹介、局所性とキャッシュの概要、ベクトル化（単一命令複数データ、SIMD）の紹介、および TBB でサポートまたは利用される C++ 機能（C の機能を超えるもの）の説明が含まれます。

- 1 章 – 9 章: これらの章では、TBB を包括的に紹介し、効果的な並列プログラミングを実行するのに必要な知識を提供します。

- 10 章 – 11 章: これらの章では、TBB を使用して並列アプリケーションを作成する実用的なアドバイス、ヒント、コツを紹介します。10 章では、デフォルトの TBB 設定で得られるパフォーマンス（構成の可能性の柱）について説明し、TBB アリーナと他のプログラミング・モデルとの相互運用性について説明します。11 章では、デフォルトの TBB 設定を超えたパフォーマンス・チューニングについて詳しく説明します。

- 12 章: この章では、オリジナルの TBB（2006 年～2019 年）から現在の TBB（oneTBB、2020 年～現在）への移行に焦点を当てます。これは、古いアプリケーションを更新する貴重なリファレンスであり、今日の TBB への移行と並列プログラミングに最新の C++ を活用するアドバイスも含まれています。

- 付録 A: TBB の最初の書籍以来の伝統であるこの付録では、TBB の背後にあるインスピレーションを歴史的な観点から考察します。今日の TBB が、以前の並列プログラミングにおける先駆的な研究からどのように恩恵を受けているかを示します。

Think Parallel !（並列に考える）

並列プログラミングを初めて学ぶ人にとって、この序文は、本書の残りの部分をより有用でわかりやすく、自己完結的なものにする強固な基礎を提供します。本書は、C++ プログラミングの基本的な知識があることを前提としていますが、TBB が依存しサポートする主要な C++ 要素についても紹介しています。並列プログラミングに対するアプローチは実用的であり、並列プログラムを最も効果的にする戦略を重視しています。経験豊富な並列プログラマーにとって、この序文が並列コンピューター・ハードウェアを最大限に活用する必須の用語と概念の簡単な復習となることを願っています。

並列プログラミングの生産性と効率性を高めるには、次の 3 つの重要な原則に従います:

（1） スレッドではなくタスクを使用してプログラムする。

（2） 並列プログラミング・モデルは乱雑である必要はない。

（3） スケーラビリティー、パフォーマンス、パフォーマンスの移植性を実現するには、移植可能でオーバーヘッドの少ない並列プログラミング・モデルが必要である。

TBB はこれら 3 つの条件をすべて満たしており、非常に効果的で人気があります。

この序文を読んだ後には、分解、スケーリング、正確性、抽象化、パターンの観点から「並列に考える」とはどういう意味かを明確に説明できるようになるはずです。また、並列プログラミングにおける局所性の重要性についても理解できるようになります。さらに、TBB が推進する並列プログラミングにおける革新的なアプローチである、タスクベースのプログラミングとスレッドベースのプログラミングの背後にある原理も理解できます。最後に、TBB を効果的に使用するのに必要な、基本的な C/C++ の知識を超えた C++ プログラミング要素についても理解できるようになります。

効果的な並列プログラミングには、注目するものを明確に分離する必要があります。プログラマーはタスクによって並列性を明示することに重点を置く必要があり、プログラミング・モデルの実装（TBB）はそれらのタスクをハードウェア・スレッドにマッピングします。

アクセラレーターへのオフロードの役割

TBB は、ホスト CPU で利用可能なすべての並列性を管理できる最も効果的な方法を提供します。最新のシステムでは、GPU などのアクセラレーターによる追加の並列計算機能を利用できます。これらのオフロード手法は、Parallel STL（PSTL）、CUDA、SYCL、HIP、OpenCL などのさまざまな C++ 拡張機能を通じて実現されます。
ただし、オフロードを成功させるには、TBB による堅牢な並列処理を活用した、適切に設計された並列ホスト・アプリケーション内が最適です。

並列プログラミングでパフォーマンスを最大化するには、CPU とアクセラレーターの両方でシステム全体を最大限に活用することが不可欠です。アクセラレーターは規則性の高い並列タスクの処理に優れていますが、CPU は不規則な並列処理を独自に効率良く処理できます。これらの違いは、利用可能なツールを徹底的に理解し、特定のニーズに基づいてそれらを最適に使用することの重要性を強調しています。

スレッディング・ビルディング・ブロック（TBB）の背後にある動機

TBB は 2006 年に登場しました。これはインテルの並列プログラミングの専門家による成果であり、彼らの多くは OpenMP を含む並列プログラミング・モデルで数十年の経験を持っていました。
TBB チームの多くのメンバーは、これまで何年もかけて OpenMP 実装の開発とサポートに携わり、OpenMP の大きな成功に貢献してきました。付録 A では、TBB の歴史と、タスク・スチール・スケジューラーの画期的な概念を含む、TBB の中核となる概念について詳しく説明します。

TBB は、初期のマルチコア・プロセッサーと同じ頃に誕生し、C++ プログラマーにとって最も人気のある並列プログラミング・モデルとして急速に普及しました。TBB は豊富な機能を組み込みながら進化し、初心者と専門家のどちらにも並列プログラミングの選択肢となりました。TBB はオープンソース・プロジェクトとして、世界中からフィードバックと貢献を得ています。

TBB は、プログラマーがためらうことなく並列処理の可能性を明確にでき、基盤となるプログラミング・モデル（TBB）が実行時にそれらをハードウェアにマッピングする、という革新的なアイデアを推進しています。

TBB の重要性と価値を十分に理解するには、次の 3 つの主要原則を理解することが不可欠です: (1) スレッドではなくタスクを使用してプログラムする。(2) 並列プログラミング・モデルは複雑である必要はない。(3) スケーラビリティー、パフォーマンス、パフォーマンスの移植性を実現するには、TBB のような移植可能でオーバーヘッドの少ない並列プログラミング・モデルが必要である。

これらの原則は、効果的で構造化された並列プログラミングを習得する上で非常に重要であるため、それぞれを詳しく見ていきます。これらの概念は、並列プログラミングを理解する上での基礎となるまで、長い間過小評価されてきたと言ってもよいでしょう。

スレッドではなくタスクを使用したプログラム

並列プログラミングは、常に*スレッド*ではなく*タスク*の観点から概念化する必要があります。Edward Lee 氏は、このトピックについて信頼性のある包括的な調査を行い、2006 年に「**並行プログラミングが主流になるには、プログラミング・モデルとしてのスレッドを廃止する必要がある**」と述べています。

並列プログラミングをスレッドで表現する場合、残念ながら、アプリケーションを、実行するマシンで利用可能な特定数の並列実行スレッドに割り当てるという困難な作業に取り組むことになります。逆に、タスクを中心に並列プログラミングを構築する場合、並列処理の可能性を特定することに重点を置きます。これにより、スレッディング・ビルディング・ブロック（TBB）ランタイムなどのランタイム環境では、アプリケーションのロジックを複雑にすることなく、利用可能なハードウェアにタスクを効率良く割り当てることができます。

*スレッド*は、一定時間ハードウェア・スレッド上で実行される個別の実行ストリームを表し、後続のタイムスライスで別のハードウェア・スレッドに移行する可能性があります。このスレッド中心のアプローチでは、実行スレッドとハードウェア・スレッド（プロセッサー・コアなど）の間の誤ったマッピングが頻繁に発生します。ただし、ハードウェア・スレッドはマシンごとに異なる物理リソースであり、実装が微妙に異なります。スレッドの定義と説明については、後ほど「スレッドとは何か」の節で詳しく説明します。

対照的に、*タスク*は並列処理の*可能性*を表現します。必要に応じてタスクを細分化する機能を利用して、必要なときに利用可能なスレッドを埋めることができます。

これらの定義を念頭に置くと、スレッドをベースに記述されたプログラムは、各アルゴリズムを特定のハードウェアおよびソフトウェア・システムにマッピングする必要があります。これは注意をそらすだけでなく、並列プログラミングをより困難にし、効率を低下させ、移植性を著しく低下させるさまざまな問題を引き起こします。

プログラムがタスクをベースに記述されている場合、TBB などのランタイムは、実行時にそれらのタスクを利用可能なハードウェアに動的にマッピングできます。このアプローチにより、システム内に備わる実際のハードウェア・スレッド数を心配する必要がなくなります。

さらに重要なことは、階層化された並列処理を効率良く有効にする唯一の実用的な方法であるということです。これを考慮して、本書全体を通じてネストされた並列処理の重要性を取り上げています。

構成の可能性: 並列プログラミングは必ずしも複雑である必要はない

TBB は並列プログラミングの*構成の可能性*を提供し、それがすべてを変えます。構成の可能性とは、TBB の機能を制限なく組み合わせることができることを意味します。最も注目すべきは、階層化機能が含まれることです。したがって、`parallel_for` ループ内に `parallel_for` をネストするのは理にかなっています。また、`parallel_for` が関数を呼び出し、その関数の中に `parallel_for` が含まれていても問題ありません。

構成可能なネストされた並列処理をサポートすると、並列処理の可能性が増え、よりスケーラブルなアプリケーションが実現できるため、非常に望ましいことがわかるでしょう。例えば、OpenMP は、ネストに関しては構成可能ではありません。ネストの各レベルによって、大きなオーバーヘッドが発生し、リソースの消費が枯渇し、プログラムの終了につながる可能性があるためです。 ライブラリー・ルーチンに並列コードが含まれている可能性があることを考慮すると、これは大きな問題です。そのため、並列処理をすでに実装しているときにライブラリーを呼び出すと、構成不可能な手法では問題が発生する可能性があります。TBB は構成可能であるため、そのような問題は発生しません。TBB は、並列処理（タスク）を明示する可能性を提供し、実行時にそれらをハードウェア（スレッド）にマップする方法を決定することで、この問題を部分的に解決できます。

これは、スレッド（必須の並列処理）ではなくタスク（潜在的な並列処理）の観点からコーディングすることの大きな利点です。`parallel_for` が必須であるとみなされた場合、ネストによってスレッドが急増し、制御されていないとプログラムが簡単に（そして頻繁に）クラッシュするリソースの問題を引き起こす可能性があります。`parallel_for` が利用可能な強制的でない並列性を明示する場合、ランタイムはその情報を自由に使用して、マシンの能力に最も効果的な方法で利用できます。

私たちはプログラミング言語に構成の可能性を期待するようになりましたが、ほとんどの並列プログラミング・モデルはそれを維持することができていません（幸いなことに、TBB は構成の可能性を維持しています）。`if` 文および `while` 文について考えてみます。C 言語と C++ 言語では、`if` 文と `while` 文を必要に応じて自由に組み合わせたりネストしたりできます。もし、`if` 文内から呼び出される関数に `while` 文を含めることが禁止されていたらどうでしょう？

そのような制限の提案は愚かなことのように思われるでしょう。TBB は、並列構造を制限なく、また問題を引き起こすことなく自由に混在およびネストできるようにすることで、この種の構成の可能性を並列プログラミングにもたらします。

スケーリング、パフォーマンス、そしてパフォーマンスの移植性の追求

TBB を使用してプログラミングする最も重要な利点は、パフォーマンスの移植が可能なアプリケーションを作成できることです。*パフォーマンスの移植性*とは、プログラムがさまざまなマシン（異なるハードウェア、異なるオペレーティング・システム、またはその両方）間で同様の「ピーク・パフォーマンスのパーセンテージ」を維持できる特性であると定義します。コードを変更することなく、さまざまなマシンでピーク・パフォーマンスの高いパーセンテージを実現したいと考えています。

また、64 コアマシンでは、クアッドコア・マシンと比較して 16 倍のパフォーマンス向上が期待できます。さまざまな理由から、理想的なスピードアップが得られることはほとんどありません（絶対にないとは言えませんが、合計キャッシュサイズの増加により、理想を上回るスピードアップが得られることもあります。この状態をスーパーリニア・スピードアップと呼びます）。

スピードアップとは？

スピードアップは、順次実行する時間を並列実行する時間で割ったものとして定義されます。プログラムが通常は 3 秒で実行されるが、クアッドコア・プロセッサーでは 1 秒しかかからない場合、3 倍のスピードアップが達成されたと言えます。場合によっては、処理コアの数で割ったスピードアップを効率と言うこともあります。3 倍のスピードアップは、並列処理の効率では 75% になります。

クアッドコア・マシンから 64 コアマシンに移行してパフォーマンスを 16 倍向上させるという理想的な目標は、*線形スケーリング*または*完全なスケーリング*と呼ばれます。

これを達成するには、コア数が増えてもすべてのコアをビジー状態に保つ必要があります。これには、かなりの並列処理（利用可能な並列性）が必要となります。この “利用可能な並列性” の概念については、序文の後半でアムダールの法則とその意味について説明するときに詳しく紹介します。

ここでは、TBB がハイパフォーマンス・プログラミングをサポートし、パフォーマンスの移植性を大幅に向上させる、ということを知っておくことが重要です。TBB のオーバーヘッドは非常に低いため、問題なくスケーリングが可能であることから、ハイパフォーマンスが実現します。パフォーマンスの移植性により、新しいマシンがより多くの機能を提供すると、アプリケーションは利用可能な並列処理を自動的に活用できるようになります。

ここでは、わずかなオーバーヘッドのみで、最も効果良く並列性を明示して、それを利用できる動的タスク・スケジューリングの世界を想定しています。この仮定には 1 つの欠点があります: 動的な調整を行わずに、ハードウェアに完全に一致するようにアプリケーションをプログラムできる場合、パフォーマンスが数パーセント向上する可能性があります。従来のハイパフォーマンス・コンピューティング（HPC）プログラミングは、世界最大級のコンピューターで集中的に計算するプログラミングする手法として知られており、高度に並列化された科学計算は長い間この特性を備えてきました。静的スケジューリングで OpenMP を活用し、パフォーマンスの向上が見られた HPC 開発者は、TBB の動的な性質によりパフォーマンスがわずかに低下すると感じるかもしれません。しかし、静的スケジュールの利点は、さまざまな理由によってなくなりつつあります（日々稀になってきています）。HPC プログラミングを含むすべてのプログラミングは、ネストされた動的並列処理のサポートが必要になるほど複雑になっています。これらの効果は、マルチフィジックス・モデルの成長、AI（人工知能）の導入、ML（マシンラーニング）手法など、HPC プログラミングのあらゆる面で見られます。複雑性が増す主な要因の 1 つは、ハードウェアの多様性が増し、単一のマシン内で異種コンピューティング機能が実現されていることです。TBB は、フローグラフ機能など、これらの複雑性に対処する強力なオプションを提供します。フローグラフ機能については、4 章と 5 章で詳しく説明します。

効果的な並列プログラミングには、注目するものを明確に分離する必要があります。プログラマーはタスクによって並列性を明確にすることに重点を置く必要があり、プログラミング・モデルの実装（TBB）はそれらのタスクをハードウェア・スレッドにマッピングします。

スケーリングとは?

最終的な目標は、並列プログラミングにより、コンピューター・システムで利用可能な並列計算の利点を活用してスピードアップを実現することです。スケーリングとは、アプリケーションが利用可能な並列処理が増えたらそれらを活用できる特性を指します。問題を 2 つの並列計算に単純に分割するプログラムを記述するのは容易です。そのようなプログラムは、デュアルコア・システムでスピードアップが見られます（1 コアから 2 コアに移行すると適切にスケーリングされます）。ただし、4 コア、8 コア、またはそれ以上の処理コアで継続してスケールすることはありません。スピードアップとスケーラビリティーはどちらも並列プログラミングの重要な目標であり、それらを十分に理解することが極めて重要です。後述する「アプリケーションにどれだけの並列処理があるか」という節では、これらの重要な概念をしっかりと理解できます。

並列プログラミングの紹介

並列プログラミングの用語と主な概念を詳しく説明する前に、誇張して主張しておきましょう。並列プログラミングは順次プログラミングよりも直感的です。並列処理は私たちの日常生活の基本であり、タスクを段階的に実行することはめったに経験したり期待したりしない贅沢です。 並列処理は私たちにとって馴染み深いものであり、プログラミングへのアプローチにも自然に受け入れられるはずです。

並列処理は身近にある

我々は日常生活の中でも、並列処理を考えていることがあります。いくつか例をあげてみましょう:

長い行列: 長い行列に直面したとき、もっと短くて速い行列を複数用意したり、係員を増やしたてもっと効率良く接客して欲しいと思ったことがあるでしょう。例えば、スーパーのレジの列、駅の切符売り場の列、コーヒーを買う列などです。

反復的なタスク: 多くの人が同時に作業する大きなタスクに直面したとき、きっともっと手伝ってくれる人がいればいいのにと思ったことがあるでしょう。例えば、家財道具を家から家へ移動させたり、大量の郵便物を送るために封筒に手紙を詰めたり、研究室の複数の新しいコンピューターに同じソフトウェアをインストールしたりすることが挙げられます。「人手が多ければ作業が楽になる」ということわざは、コンピューターにも同様に当てはまります。

並列処理を深く理解し、使い始めると、並列思考が身につきます。まずはプロジェクトの並列処理について考え、それからコーディングについて考えるようになります。

有名なコンピューター・アーキテクトであるイェール・パットは次のように述べています:

従来の知識の問題とは、「並列思考は難しい」という思い込みです。

おそらく（すべての）思考は難しいのです！

並列思考が自然だと信じてもらうにはどうすればいいでしょう？

（全く同感です）

並行性と並列性

「並行」と「並列」という用語には関連性がありますが、微妙に異なるため、区別しておきましょう。「並行」とは「同じ期間に起こる」という意味ですが、「並列」はより具体的に「(少なくとも一部の時間は) 同時に起こる」ことを意味します。並行 (同時) 実行は、1 人の人がマルチタスクを実行するときの試みに似ており、並列実行は複数の人が協力して作業することに似ています。図 P-1 は、並行性と並列性の概念を示しています。

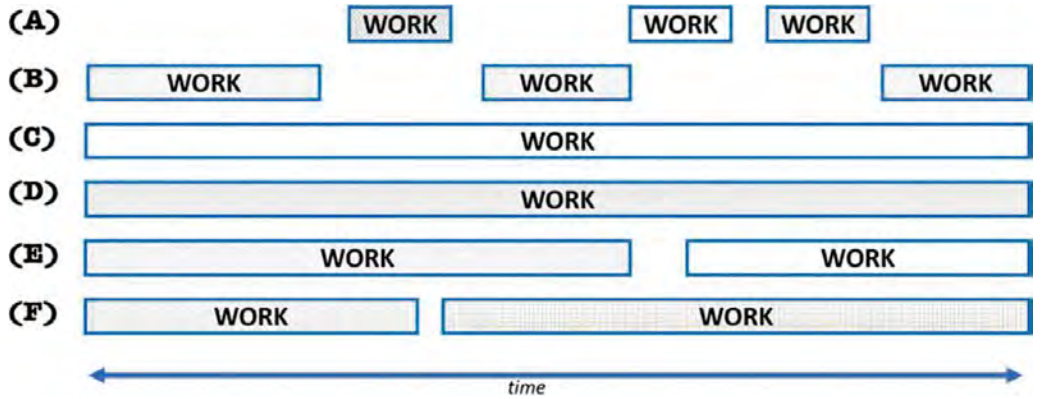


図 P-1. 並列実行 vs. 並行実行: タスク A と B は互いに並行して同時に実行されますが、互いに並列ではありません。その他のすべての組み合わせは並行かつ並列です

効率良い並列プログラムを作成する場合、私たちの目標は、並行性以上のものを実現することです。一般的に、並行性は、かなりの量のアクティビティーが真に並列であることが期待されないことを意味します。つまり、2 人のワーカーが、それぞれ単独で理論的に達成できる以上の成果を達成するわけではありません (図 P-1 のタスク A と B を参照)。並行作業はすぐには完了しないため、タスクの待ち時間 (タスクが開始するまでの遅延) は改善されません。

逆に、「並列」という用語を使用すると、レイテンシーとスループット (特定の時間内に実行されるワーク量) の改善が期待されます。これについては、並列処理の限界を説明し、アムダールの法則の重要な概念を詳しく調べ、さらに詳しく検討していきます。

並列処理を妨げるもの

並列プログラミングの敵を念頭に置くと、私たちが特定のプログラミング手法を支持するのを理解するのに役立ちます。並列プログラミングを妨げる要因には以下が含まれます:

ロック: 並列プログラミングでは、ロックまたは排他制御オブジェクト（ミューテックス）を使用して、スレッドにリソースへの排他的アクセスを提供し、他のスレッドが同じリソースに同時にアクセスすることをブロックします。ロックは、（純粋なカオスを許すのではなく）並列タスクが協調して共有データを更新することを保証する最も一般的な方法です。ロックはプログラムの一部をシリアル化してスケーリングを制限するため、好ましくありません。「私たちはロックが嫌いだ」という感情が、本書の全体にわたって感じ取っていただけるでしょう。私たちは、いつ適切に同期すべきかを忘れずに、この言葉を皆さんにも浸透させていきたいと考えています。しかし注意が必要です。必要な場合のロックは実は好きです。それはこのロックがなければ災難に見舞われるからです。ロックに対するこの好き嫌いの関係を理解する必要があります。

共有された可変状態: 「可変（Mutable）」とは、「変更可能（can be changed）」という意味です。共有された可変状態は、複数のスレッド間でデータを共有するたびに発生し、共有中にデータを変更できます。このような共有は、同期が必要であり、適切に使用されていてもスケーリングを低下させたり、同期（ロックなど）が誤って使用されると正当性の問題（競合状態またはデッドロック）を引き起こします。現実的には、アプリケーションを記述するときには、共有された変更可能な状態が必要です。共有された可変状態の扱いについて考えることは、ロックに対する好き嫌いの根底を理解するより簡単かもしれません。最終的には、共有された変更可能な状態と排他制御（ロックを含む）を「管理」して、希望どおりに動作させることになります。ロック – 私たちはロックとともに生きられないし、ロックなしでは生きてはいけません。

“並列思考”ではない: 巧妙な処置やパッチの適用は、スケーラブルなアルゴリズムに対する不適切な戦略を補うものではありません。実装する前に、並列処理が利用可能な場所と、それをどのように活用できるか考える必要があります。アプリケーションを作成した後に並列処理を追加しようとすると、危険が伴います。既存のコードの中には、並列処理に移行可能なものがあるかもしれませんが、ほとんどのコードはアルゴリズムを大幅しなければ恩恵は受けられませ

ん。

アルゴリズムが勝ることを忘れる: これは単に「並列に考えなさい」と言っているだけかもしれません。アルゴリズムの選択は、アプリケーションのスケーラビリティーに大きな影響を与えます。アルゴリズムの選択によって、タスクをどのように分割するか、データ構造にどのようにアクセスし、結果を結合するかが決まります。最適なアルゴリズムは、実際には最適なソリューションの基礎として機能するものです。最適なソリューションは、最適なアルゴリズムと並列データ構造、およびデータに対する計算をスケジュールする方法を組み合わせたものです。私たちプログラマーにとって、より優れたアルゴリズムの探求と発見には終わりがありません。並列プログラマーとして、私たちはアルゴリズムの定義に*スケーラビリティー*を加える必要があります。適切なスケーリングするには、計算の複雑さに注意を払わなければなりません。特に、選択したアルゴリズムを比較する場合、特定のアルゴリズムにおけるデータセットのサイズに応じて時間とメモリー要件がどのように変化するか理解する必要があります。

並列処理の用語

並列プログラミングの用語は、他の並列プログラマーと会話するためにも学ぶ必要があるものです。いずれの概念も難しいものではありませんが、習得することが大切です。並列プログラマーは、他のプログラマーと同様に、基礎的なことは説明できるほど単純であるにもかかわらず、自身の技術に対する深い直感を身につけるのに何年も費やします。

ここでは、ワークを並列タスクに分解すること、スケーリングの用語、正当性に関する考慮事項、およびキャッシュの効果による局所性の重要性について説明します。

では、アプリケーションについて考えるとき、並列処理の可能性をどのように見つけるのでしょうか？

最上位のレベルにおける並列処理は、並列に操作するデータの形式、または並列に実行するタスクの形式で存在します。そしてそれらは相互に排他的ではありません。ある意味では、並列処理において重要なことはすべてデータ並列処理にあります。しかしながら、両方考えておくと便利な場合があるので、両方を紹介します。
スケーリングとアムダールの法則について議論すると、データの並列性を求める私たちの傾向がより理解しやすくなります。

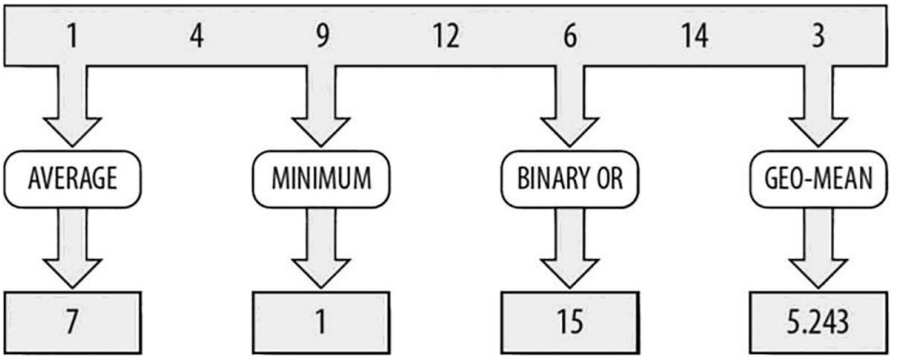


図 P-2. タスク並列処理

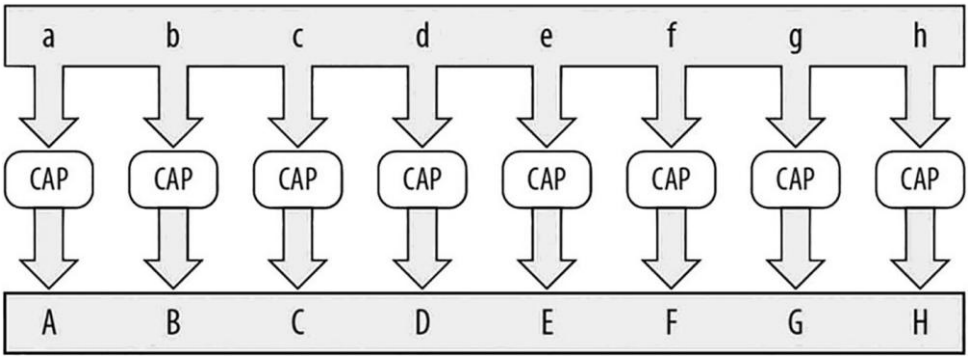


図 P-3. データ並列処理

用語: タスク並列処理

*タスク並列処理*とは、異なる独立したタスクを指します。図 P-2 は、独立した値を計算する同じデータセットにそれぞれ適用できるいくつかの数学演算の例を示しています。 ここでは、データセットの平均値、最小値、2 進論理和、幾何平均が計算されます。タスク並列処理という観点から並行して実行するワークを見つけることは、想定される独立した操作の数によって制限されます。

この序文の前半では、スレッドの代わりにタスクを使用することを推奨してきました。ここで、データ並列性とタスク並列性について説明しますが、タスク並列性とデータ並列性を比較するときに、「タスク」という単語を別の説明でも再度使用するため、少し混乱するかもしれません。どちらのタイプの並列処理でも、スレッドではなくタスクの観点からプログラムします。これは並列プログラマーが使用する用語です。

用語: データ並列処理

データ並列処理（図 P-3）は容易に想像できます。大量のデータを取得し、データの各部分に同じ変換を適用します。図 P-3 では、データセット中の文字はそれぞれ対応する大文字に変換しています。この単純な例では、データセットと要素ごとに適用する操作が指定されているので、要素ごとに同じタスクを並列に適用できることを示しています。スーパーコンピューター向けのコードを記述するプログラマーは、この種の課題が大好きで、並列処理することは容易であると考えます。この種の課題は、*驚異的並列*と呼ばれます。

アドバイス: 大量のデータを並列処理する場合、困惑するのではなく楽しんでください。*並列処理は楽しい*と考えることです。

並列で実行するワークを見つける労力を比較すると、データの並列処理に重点を置いたアプローチは、処理するために取得できるデータの量によって制限されます。タスクの並列処理のみに基づくアプローチは、プログラムで表現するさまざまなタスクの種類によって制限されます。どちらの方法も有効かつ重要ですが、真にスケーラブルな並列プログラムを実現するには、処理するデータ内で並列性を見つけることが最も重要です。スケーラビリティーとは、データが十分に供給される場合、ハードウェア（プロセッサー・コアの増加など）を追加するとアプリケーションのパフォーマンスが向上することを意味します。ビッグデータの時代では、ビッグデータと並列プログラミングは互いに相性が良いことが分かります。データサイズの増加は、追加ワークの確実な発生源となるようです。これについては、序文の後半でアムダールの法則を説明するときに再度取り上げます。

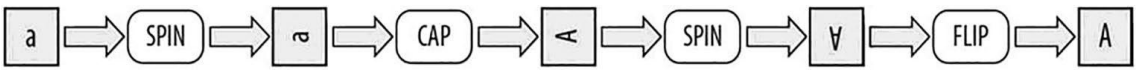


図 P-4. パイプライン

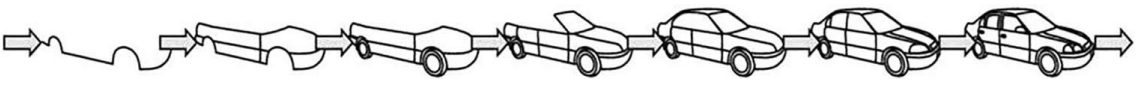


***図 P-5.** 各ポジションが、異なる組み立てステージにある異なる自動車であると考えてください。これは、データが流れるパイプラインです*

用語: パイプライン

タスク並列処理はデータ並列処理よりも見つけにくいですが、タスク並列処理の特定種類である*パイプライン処理*は注目に値します。この種のアルゴリズムでは、多くの独立したタスクにデータのストリームを適用する必要があります。 各項目は、図 P-4 の文字 A で示されているように、各ステージで処理されます。図 P-5 では、異なる項目を異なるステージで同時に処理するため、パイプラインを使用しデータのストリームをより高速に処理する様子を示しています。これらの例では、結果を取得する時間を短縮できない可能性があります（入力から出力までの時間として測定される*レイテンシー*と呼ばれます）が、単位時間あたりの完了（出力）で測定されるため、*スループット*は高くなります。パイプラインを使用すると、順次（シリアル）処理と比較して並列処理によって*スループット*を向上できます。パイプラインはさらに洗練され、データを新しいルートで処理したり、特定の項目をスキップしたりできます。TBB は、単純なパイプライン（2 章）と複雑なパイプライン（4 章）をサポートしています。 もちろん、パイプラインの各ステップでは、データまたはタスクの並列処理も導入できます。TBB の構成の可能性により、これはシームレスにサポートされます。

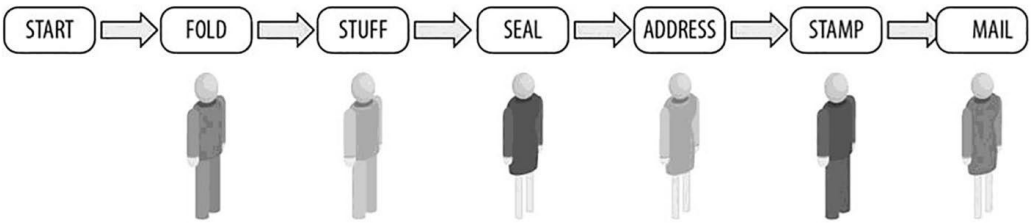


図 P-6. パイプライン – 各人の作業は異なる

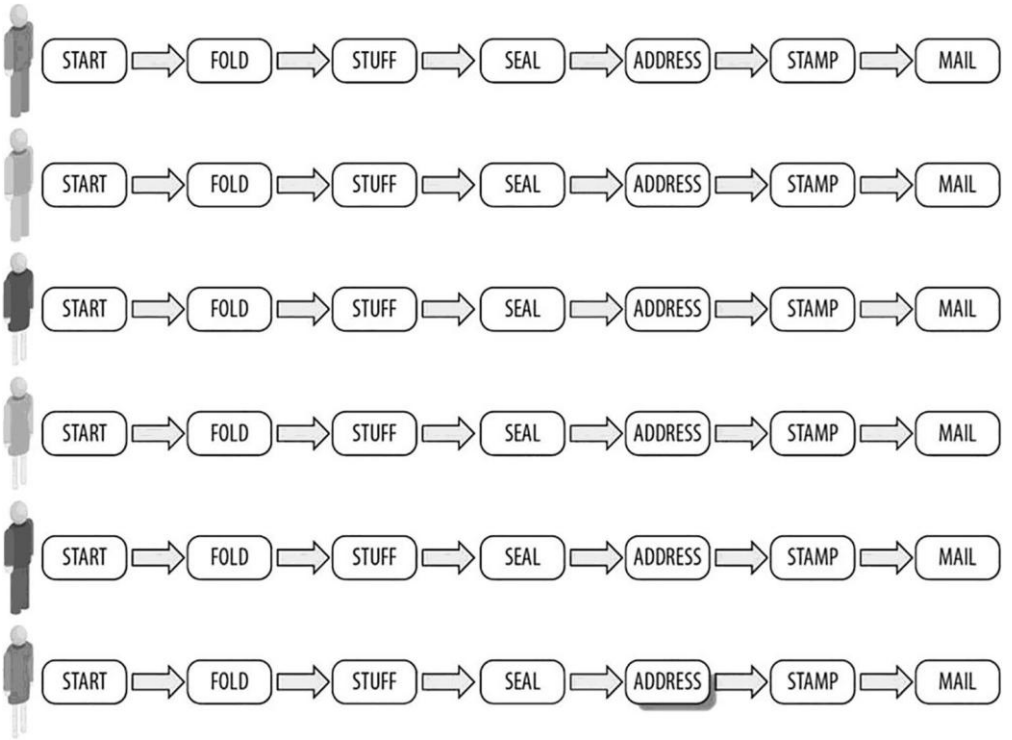


図 P-7. データ並列処理 – 各人が同じ作業をする

ミックスされた並列処理の活用例

手紙を折りたたみ、封筒に入れて、封をし、宛名書きをして、切手を貼り、そして発送するタスクを考えます。この一連の封筒詰めタスクに 6 人のグループを編成する場合、各人はそれぞれの作業を担当して、パイプライン方式で 1 つのタスクを実行します（図 P-6）。これは、必要な資材を分割し、すべての資材を一括して各作業員に渡すデータ並列処理とは対照的です（図 P-7）。各人は割り当てられた資材を使用して、すべてのステップを自身のタスクとして実行します。

全員が互いに離れた場所で離れて作業しなければならない場合、図 P-7 は明らかに正しい選択です。タスク間のやり取りがほとんど起こらない（作業員は封筒を集めるためだけに集まり、その後は各自の場所で、発送を含む各作業を行う）ため、*疎粒度*の並列処理と呼ばれます。図 P-6 で示されるもう一つの選択肢は、頻繁にやり取りされる（すべての封筒が作業のさまざまなステップですべての作業員に渡される）ため、*細粒度*の並列処理として知られています。

たまには非常に近い状況になるかもしれませんが、この極端な状況が当てはまることは現実にはほとんどありません。この例では、宛名書きは 3 人分の作業量ですが、最初の 2 つのステップと最後の 2 つのステップは、2 つのステップを合わせて 1 人分の作業量しかありません。図 P-8 は、各ステップで行う作業量をサイズで示しています。図 P-6 に示すように、各ステップに 1 人だけを割り当てた場合、この作業パイプラインの一部の人々は「作業がない」状態になり、アイドル状態になると考えられます。隠れた「不完全雇用」と言えるかもしれません。パイプラインで適切なバランスを実現するソリューション（図 P-9）は、実際にはデータとタスクの並列処理のハイブリッドです。

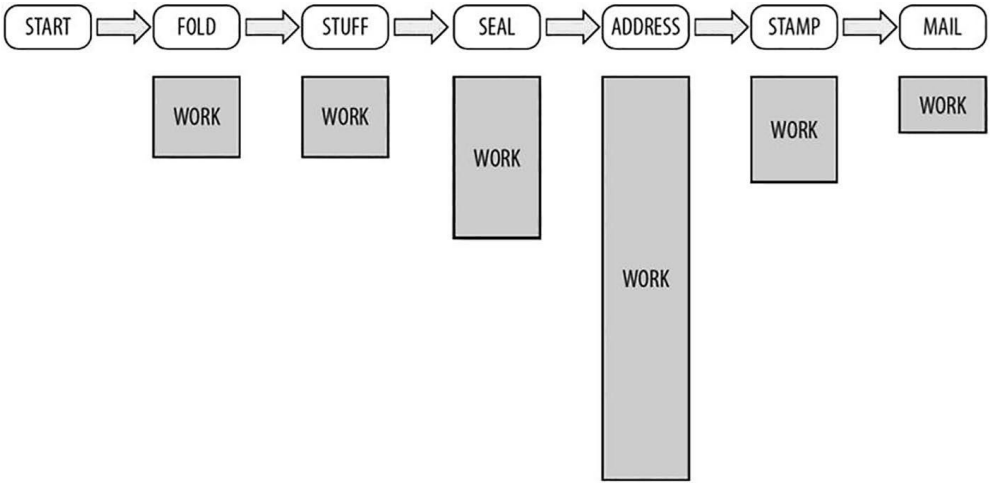


図 P-8. 異なるタスクは組み合わせるか人数に合わせて分割する

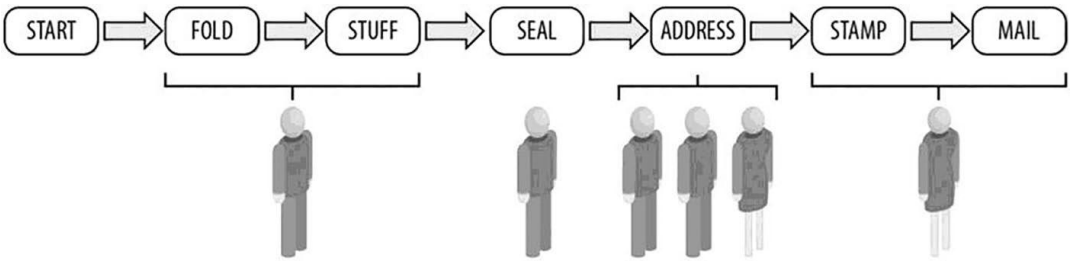


図 P-9. タスクの作業員が異なるため、宛名書きタスクにより多くの人を割り当てる

並列処理の実現

封筒を準備および発送する作業に関わる人々を調整することは、次の 2 つの概念的な手順で表現できます。

1. タスクに人を割り当てる（そして作業量が平均化するようにタスクを調整する）。

2. 6 つのタスクをそれぞれ 1 人で開始して、2 人以上が一緒に作業できるようにタスクを分割する。

6 つのタスクは、手紙を折りたたむ、封筒に入れる、封をする、宛名を書く、切手を貼る、手紙を発送する、です。そして、作業を行う人（リソース）は 6 人です。このような場合、まさに TBB が最適に働きます。わかっている範囲でタスクとデータを定義して、作業を行うことができるリソースと一致するようにデータを分割または組み合わせます。

並列プログラムを記述する最初のステップは、どこを並列処理できるかを考えることです。多くの文献では、まるで他に選択肢がないかのようにタスクの並列処理とデータの並列処理を紹介しています。TBB は、2 つの並列処理を自由に組み合わせることができます。

私たちは、カオスを非難するどころか、協調性のない多くのタスクが互いに確認（同期）することなく作業をこなすカオスが大好きなのです。この、「疎結合」並列プログラミングは素晴らしいものです! ロックよりも同期が嫌悪されるのは、同期によってタスクが他のタスクを待機するようになるからです。タスクは動作するために存在します。ただ待つためではありません。

運が良ければ、プログラムには利用できるデータ並列性が豊富に備わっていることになります。この作業を単純化するため、TBB ではタスクとその分割方法を指示する必要があります。完全にデータ並列なタスクの場合、TBB では、すべてのデータを処理する 1 つのタスクを定義します。次に、そのタスクは利用可能なハードウェアの並列処理機能（プロセッサーやコア）を利用するため自動的に分割されます。

暗黙の同期（コーディングで直接記述される明示的な同期とは対照的）により、ロックを使用することなく同期できることがあります。私たちがロックを嫌っているという事実を考慮すると、暗黙の同期は好ましいことです。では、「暗黙の」同期とはどういうことでしょうか? これは、同期が行われることを意味しますが、同期を明示的にコード化したわけではありません。最初、これは「不正」のように見えるはずです。同期が発生したということは、誰かがそれを要求する必要があったということです。ある意味、私たちはこれらの暗黙の同期が慎重に計画され、実装されることを期待しています。TBB の標準的なメソッドを多く使用し、独自のロックコードを明示的に記述する必要がなくなると、一般的に、状況は改善します。

TBB にワークを管理させることで、ワークを分割し、必要に応じて同期の役割を任せます。ライブラリーによって行われる同期（暗黙の同期と呼びます）により、明示的な同期のコードが不要になることがあります（8 章を参照）。

まずはそこから始め、絶対に必要または有益な場合にのみ、明示的な同期（8 章）を行うことを強く推奨します。経験則から言うと、同期が必要に思えても、必ずしも必要ではないということです。十分に注意してください。皆さんも私たちと同じなら、時々警告を無視して失敗することもあるでしょう。

分割についてはこれまで数十年調査が行われ、いくつかのパターンが明らかになっています。そのような効果的な設計パターンと TBB インターフェイスの概念を 2 章で関連付けます（図 2-1 を参照）。

効果的な並列プログラミングとは、すべてのタスクを常にビジー状態に維持し、有用なワークを終えることです。アイドル時間を検出して排除することが、スケーリングによって大幅なスピードアップを達成する鍵となります。

用語: スケーリングとスピードアップ

プログラムのスケーラビリティーは、より多くの計算能力を追加したときにプログラムがどれだけスピードアップするかという単位です。スピードアップは、並列処理なしでプログラムを実行するのにかかった時間と、並行処理ありでプログラムを実行するのにかかった時間の比率です。4 倍のスピードアップは、並列プログラムがシリアルプログラムの 4 分の 1 の時間で実行されることを示します。
シングルコア・マシンで実行に 100 秒かかり、クアッドコア・マシンでは 25 秒で実行できるシリアルプログラムを考えます。

通常は、2 つのプロセッサー・コア上で実行されるプログラムは、1 つのプロセッサー・コア上で実行するプログラムよりも高速であると期待されます。同様に、4 つのプロセッサー・コア上での実行は 2 つのコアよりも高速になると期待されます。

どのようなプログラムにも、並列処理を実装すると効果が減少するポイント（収穫逓減）が存在します。過度に計算リソースを使用すると、パフォーマンスが安定するどころか、低下することも珍しくありません。問題の細分化を停止する粒度は、*グレインサイズ（grain size）*として表現できます。TBB はグレインサイズの概念を使用して、データの分割を適切なレベルに制限し、パフォーマンスが低下する問題を回避します。 グレインサイズは通常、TBB 内の自動パーティショナーによって、初期推測と実行状況に応じた動的なヒューリスティックの組み合わせにより自動的に決定されます。ただし、必要に応じてグレインサイズの設定を明示的に操作することもできます。明示的な設定を使用しても、TBB の自動パーティショナーよりパフォーマンスが向上することはほとんどないため、本書ではこれを推奨しません。これはある程度マシンに固有である傾向があるため、グレインサイズを明示的に設定すると、パフォーマンスの移植性が低下することもあります。

並列思考が身につけば、スケールするように処理を構築することが当たり前になるでしょう。

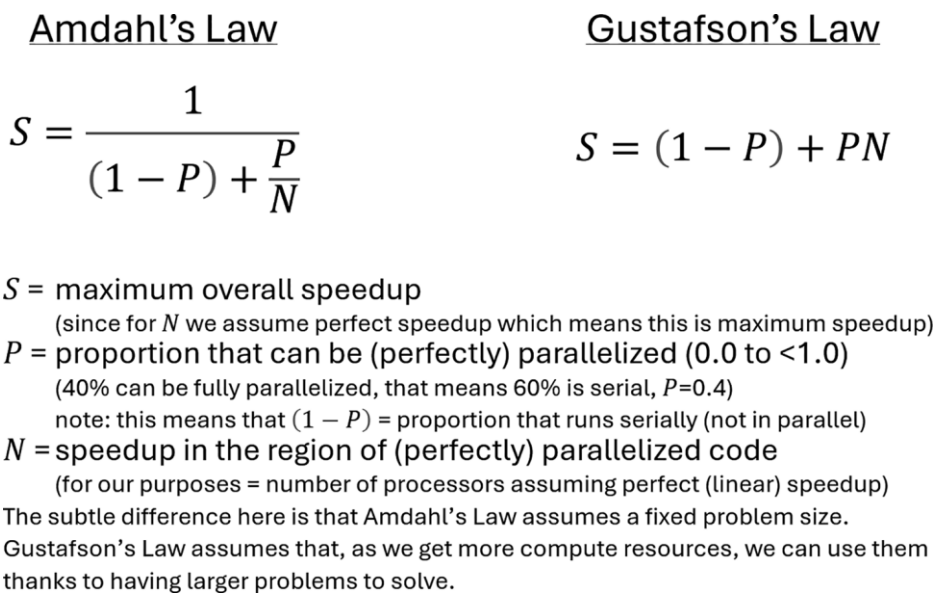


図 P-10. 「どれだけの並列性」に関するアムダールの法則とグスタフソンの見解

アプリケーションにどれだけの並列処理があるか？

アプリケーションにどれだけの並列処理があるかという話題は、これまでかなり議論されてきましたが、その答えは「状況に依存する」です。この問題に対する 2 つの正しい見解の式を、同じ 2 つの入力を使用する形式で示した図 P-10 を示します。この意味と視点の違いは、以下の説明から明らかになります。

それは確かに、解く問題のサイズや並列処理を活用する適切なアルゴリズム（そしてデータ構造）を見つけ出す能力に依存します。マルチコア・プロセッサーが登場するまでは、この議論の多くは大規模な並列コンピューターの分野で効率的なプログラムを記述しているか確認することに集中していました。マルチコア・プロセッサーの登場とともに、サイズの定義、効率への要求、そしてコンピューターの設備費用はすべて変化しました。一歩下がって我々が現在立っている位置を見つめ直す必要があります。世界は変わったのです。

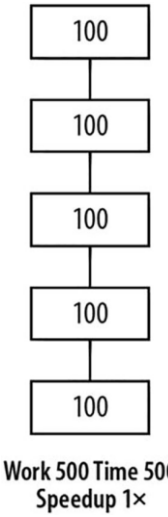


図 P-11. 並列処理のないオリジナルのプログラム

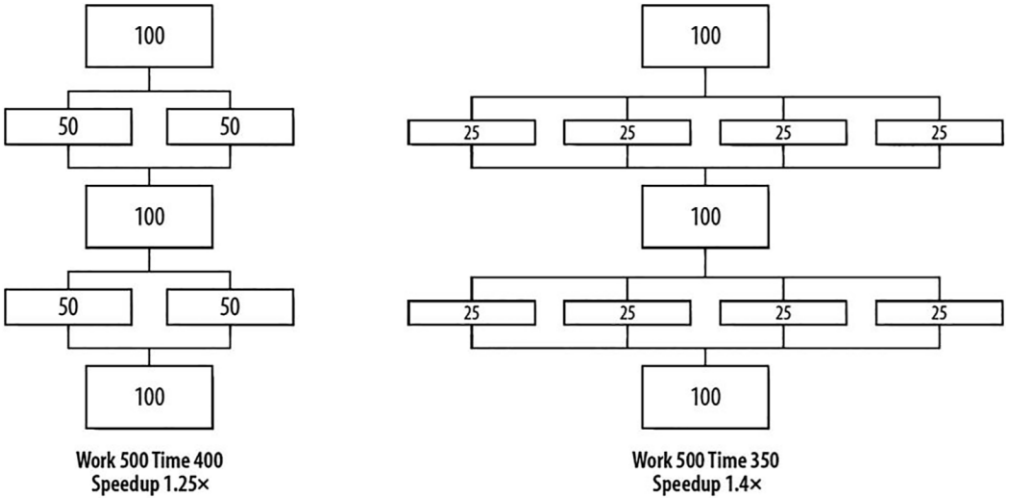


図 P-12. 並列処理の追加の進捗

アムダールの法則

著名なコンピューター設計者であるジーン・アムダール氏は、システムの一部のみを改良したときにコンピューター・システムで期待できる最大向上率に関する発表を行いました。1967 年に行った彼の発表は、アムダールの法則として知られています。

この法則によれば、プログラムのすべてを 2 倍に高速化すると、プログラムは 2 倍の速さで動作することが期待できます。しかし、プログラムの 5 分の 2 のパフォーマンスのみを 2 倍に向上させた場合、システム全体では 1.25 倍しか向上しません。

アムダールの法則は容易に図で示すことができます。図 P-11 で示されるように、500 秒で動作する 5 つの等しい領域を含むプログラムを想像してください。図 P-12 で示されるように、2 つの領域を 2 倍と 4 倍にできれば、500 秒がそれぞれ 400 秒（1.25 倍のスピードアップ）と 350 秒（1.4 倍のスピードアップ）に減ります。しかし、並列処理でスピードアップしない領域の制限は残ったままです。どれだけ多くのプロセッサー・コアが利用可能でも、シリアル領域に突破できない 300 秒の壁が残ります（図 P-13）。そのため、スピードアップは最大 1.7 倍しか得られません。実行時間の 5 分の 2 のみに並列プログラミングが制限されている場合、パフォーマンスを 1.7 倍以上向上させることはできません。

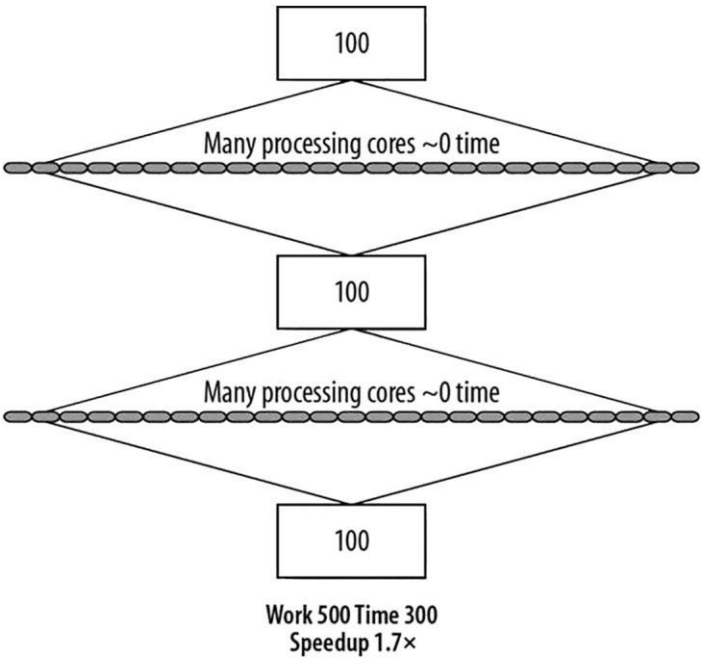


図 P-13. アムダールの法則による限界

並列プログラマーは、多数のプロセッサーを使用して期待できる最大向上率を予測するのに、長い間アムダールの法則を使用してきました。この解釈は、どんなに多くのプロセッサーがあっても、コンピューター・プログラムは並列で実行しない（シリアル）領域の合計より速くなることはないことを最終的に示しています。

多くの人が並列コンピューターの先行きの暗さを予言するのにアムダールの法則を使用してきましたが、その予想を覆す別の法則が発表されました。

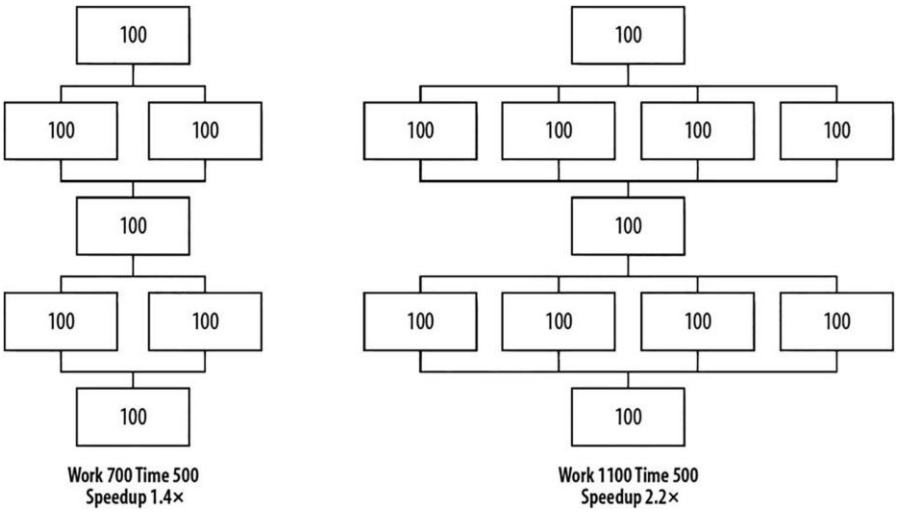


図 P-14. 能力に応じて作業量をスケール

グスタフソン氏によるアムダールの法則の再評価

アムダールの法則では、プログラムを固定のものとして、コンピューターに変更を加えました。しかし、これまで何度も、コンピューターが新しい能力を得ると、それらの機能を活用するためにアプリケーションも変化してきました。最近のアプリケーションのほとんどは、20 年前のコンピューターではまず動作しません。また 10 年以上前のコンピューターでは快適に動作するとは限りません。この法則はゲームや AI のように日々進歩するアプリケーションだけに限定されません。オフィス・アプリケーション、ウェブブラウザー、画像処理ソフトウェア、ビデオ編集ソフトウェアにも当てはまります。

アムダールの法則が登場してから 20 年以上後に、ジョン・グスタフソン氏は、サンディア国立研究所在籍中に異なるアプローチによりアムダールの法則を再評価しました。グスタフソン氏は、時間の経過とともにワークロードが増加する場合、並列処理がより有用であると指摘しました。つまり、コンピューターが強力になるにつれて、変化しないワークロードに注目するのではなく、より多くのワークをコンピューターに要求するようになったのです。多くの処理で、演算する処理のサイズは大きくなり、処理の並列部分で必要な作業は並列化できない領域（シリアル領域）よりも急速に増加しています。
そのため、処理のサイズが大きくなるとともに、シリアル領域が減少し、アムダールの法則によれば、スケーラビリティーが改善します。図 P-11 のようなアプリケーションから始めることもできますが、利用可能な並列処理に合わせて問題が拡大すると、図 P-14 に示すような進歩が見られる可能性が高くなります。

シリアル部分の実行にこれまでと同じ時間がかかっても、全体の割合から見るとその比率は徐々に少なくなります。このアルゴリズムは最終的に、図 P-15 で示されている結論に達します。パフォーマンスはプロセッサーの数と同じ比率で向上します。これは線形スケーリングまたは *N* オーダーのスケーリングと呼ばれ、*O(N)* と表されます。

ここで説明した例でも、プログラムの効率はまだシリアル部分によって制限されています。この例でプロセッサーの使用効率は、プロセッサー数が多い場合は約 40% です。スーパーコンピューターでは、これは大いなる無駄です。マルチコア・プロセッサーを搭載したシステムでは、アプリケーションが使用しない処理能力を活用するため、コンピューター上で他のワークを並行に実行することが期待できます。この新しい世界は非常に複雑です。「コップの半分は空っぽ」と考えてアムダールの法則を用いた場合でも、「コップの半分は一杯」と考えるグスタフソンの見解を用いた場合でも、シリアルコードを最小化することは有効です。

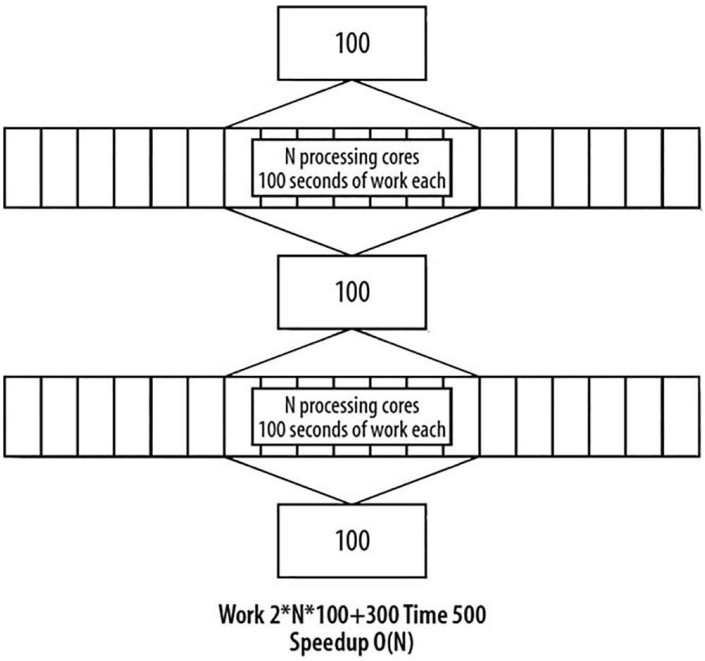


図 P-15. グスタフソンによるスケーリングの見解

アムダールの法則とグスタフソンの見解はどちらも正しく、矛盾するものではありません。それらは同じ事象を異なる視点から見て強調しています。アムダールの法則は、既存のプログラムを同じワークロードでより速く実行したいだけであれば、シリアルコードによって大幅に制限されることになると警告しています。より大きなワークロードに取り組むことを考慮すると、希望が持てるようになると、グスタフソン氏は指摘しています。

プログラムがより複雑になり、より大規模な問題を解くようになっていることは歴史的に見て明らかであり、並列プログラミングが成果を上げるときです。

並列処理の価値は、世界は変わっていないと考えるよりも、前に進んでいると考えるほうが簡単に証明できます。

アプリケーションの規模を拡張せずにわずかな変更で並列アルゴリズムを実装して実行速度を上げるのは、より大きな処理をアプリケーションに導入し実行速度を上げるよりも困難です。並列処理の価値は、現在のマシンですでに動作しているアプリケーションをスピードアップすることを強要されていない場合により簡単に証明できます。

一部の人たちは、処理のサイズが増加する傾向のあるスケーリングを*弱いスケーリング*と定義しています。前述した*驚異的並列*という用語が、他の種類のスケーリング、つまり*強いスケーリング*にも一般に用いられているのは皮肉です。本来のスケーリングは、問題のサイズが利用可能な並列処理に合わせてスケールされたときにのみ発生するので、それは単に*スケーリング*と呼ぶべきです。しかし、問題サイズが拡大せずにスケールする場合は「*驚異的並列*」または「*強いスケーリング*」と呼び、データサイズが拡大するスケーリングは「*弱いスケーリング*」と呼ぶのが一般的です。驚異的並列処理と同様に、驚異的スケーリングがある場合でも、私たちはそれを喜んで受け入れるでしょう。スケーリングは、一般にいわゆる*弱いスケーリング*であると予想されますが、どのようなスケーリングでも良いことはわかっており、そのような場合には単純にアルゴリズムが*スケール*すると言えます。

アプリケーションのスケーラビリティーは、並列で処理されるワークを増やして、シリアルで行われるワークを最小化することで得られます。アムダールはシリアル領域を減らすように勧めていますが、グスタフソンはより大きな問題を考慮するように勧めています。

彼らの言葉の真の意味は ？

アムダール氏とグスタフソン氏が実際に述べた言葉を以下に引用します。

> *. . . the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude*
> *(ほぼ同じ大きさのシーケンシャル処理の速度が改善されないのであれば、並列処理の速度を改善しようとする労力は無駄になる)。*
>
> —アムダール、1967

> *speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size（スピードアップは処理サイズを固定するのではなく、プロセッサーの数に処理をスケールして測定すべきである）。*
>
> —グスタフソン、1988

シリアルと並列アルゴリズム

プログラミングの真理の 1 つは、最良のシリアル・アルゴリズムが最良の並列アルゴリズムであることはめったになく、その逆もまためったにない、ということです。

これは、シングルコアのプロセッサーで適切に動作し、デュアルコア・プロセッサーやクアッドコア・プロセッサーのシステムでも適切に動作するプログラムを記述することは、優れたシリアルプログラム（シングルコア用）や優れた並列プログラム（メニーコア用）を記述するよりも難しいことを意味します。

スーパーコンピューターのプログラマーは、問題のサイズに応じて必要なワークが急速に増加することを経験から知っています。ワークがシリアルのオーバーヘッド（通信や同期など）よりも速く増加する場合、問題のサイズを増やすだけでスケーリングが十分ではないプログラムを修正できます。100 プロセッサーを越えるとスケールしないプログラムが、処理サイズを倍にするだけで 300 以上のプロセッサーでもうまくスケールするようになることは珍しいことではありません。

将来に備えるには、過去は捨て今後は並列プログラムを記述することです。これが最も単純で最良のアドバイスです。一方でシングルスレッドのパフォーマンスが向上するコードを記述しながら、並列処理のパフォーマンスを最大化するコードを記述することほど困難なことはありません。

スレッドとは？

スレッドについて知識がある方は、この後の「同時実行状態における安全性」節に進んでください。TBB の目標はスレッド管理を抽象化することですが、スレッドの概念を知っておくことは重要です。現実的には、今後もスレッドプログラムを構築することになるでしょう。そのためにも、この基本的な実装の意味を理解する必要があります。

現在のオペレーティング・システムはほぼすべて、プリエンプティブなスケジューラーを使用するマルチタスク・オペレーティング・システムです。マルチタスクとは、2 つ以上のプログラムを同時にアクティブにできることです。今では誰もが、電子メールプログラムとウェブ・ブラウザー・プログラムを同時に実行できるのは当然であると考えています。しかし、そう遠くない昔はそうではありませんでした。

プリエンプティブなスケジューラーによるマルチタスクでは、各プログラムがプロセッサー・コアを利用できる時間をオペレーティング・システムが制限しています。つまり、1 つのプロセッサー・コアのみで作業を実行していても、オペレーティング・システムが、電子メールプログラムとウェブブラウザーを同時に実行しているように見せかけています。

一般に、プロセス（プログラム）はそれぞれ、他のプロセスとは独立して動作します。特に、プログラム変数が格納されるメモリーは、他のプロセスで使用されるメモリーとは完全に分離されています。電子メールプログラムがウェブ・ブラウザー・プログラム内の変数を直接アクセスすることはできません。例えば、電子メールのリンクからウェブページを開くように、電子メールプログラムとウェブブラウザーが通信できる場合、その処理はメモリーアクセスよりも時間がかかる何らかの通信（おそらくプロセス間通信）によって行われます。

プログラムを互いに分離することには価値と意味があり、今日のコンピューティングでは主力となっています。プログラム内では、単一のプログラム内に複数の実行スレッドを存在させることができます。オペレーティング・システムは、プログラムを *プロセス*と呼び、実行スレッドを（オペレーティング・システムの）*スレッド*と呼びます。

そのため、現代のオペレーティング・システムはすべて、プロセスを複数の実行スレッドに分割することをサポートしています。スレッドはプロセスのように独立して動作します。スレッドは、明示的に同期しない限り、他にどんなスレッドが動作しているか、またプログラムのどこに含まれているのかを判断できません。スレッドとプロセスの重要な違いは、プロセス内のスレッドがそのプロセスのデータをすべて共有するということです。 したがって、単純なメモリーアクセスで別のスレッドの変数を設定できます。私たちはこれを「共有された可変状態」（共有された変更可能なメモリー位置）と呼び、本書では共有によって生じる問題を排除します。データの共有を管理することは、並列プログラミングの「敵」として挙げた、多岐にわたる問題の 1 つです。本書では、この課題と解決策を繰り返し取り上げていきます。

プロセス間で変更可能な状態を共有することは一般的です。それは、メモリー空間に明示的にマップされたメモリーであることも、データベースなどの外部ストア内のデータである場合もあります。典型的な例としては、航空会社の予約システムが挙げられます。このシステムは、それぞれ異なる顧客の予約を独立して処理できますが、最終的にはフライトと空席のデータベースを共有しています。したがって、単一のプロセスについて説明した概念の多くは、より複雑な状況でも簡単に発生する可能性があることを知っておく必要があります。並列に考えることを学ぶことは、TBB 以外にもメリットがあります。ただし、ほとんどの場合 TBB は複数のスレッドを持つ単一のプロセス内で使用されます。

スレッドはそれぞれ、独自の*命令ポインター*（実行しているプログラム内の場所を指すレジスター）とスタック（サブルーチンの戻りアドレスとサブルーチンのローカル変数を保持するメモリーの領域）を持ちますが、それ以外では、スレッドは同一プロセス内のスレッドとメモリーを共有します。各スレッドのスタックメモリーが他のスレッドからアクセス可能であっても、適切にプログラムされていれば、スレッドが互いのスタックに干渉することはありません。

各スレッドは同じプロセス内で他のスレッドと同じメモリーにアクセスするため、プロセス内のスレッドをそれぞれ独立して実行しても、メモリーを共有することで作業を素早く完了できるという利点があります。オペレーティング・システムは、複数のスレッドをメモリー領域に対して同じ権限を持つ複数のプロセスとして認識します。前述したように、「共有された可変状態」は、良くも悪くもあります。

スレッドのプログラミング

プロセスは通常、単一のスレッドで実行を開始し、より多くのスレッドを実行することが許されています。スレッドを使用すると、ユーザー・インターフェイス（UI）とメインプログラムのように、1 つのプログラムを複数のタスクに論理的に分解できます。スレッドは、マルチコア・プロセッサー用に並列処理をプログラムするときにも役立ちます。

スレッドを使用してプログラミングを始めると、多くの疑問が生じます。利用可能なプロセッサー・コアをビジー状態に維持するには、タスクをどのように分割して割り当てればいいのでしょうか？新しいタスクに対して毎回スレッドを作成すべきでしょうか？それとも、スレッドプールを使用して管理すべきでしょうか？スレッドの数はコアの数に依存すべきでしょうか？また、タスクが不足しているスレッドで何をすべきでしょうか？

これらはマルチタスクの実装における重要な質問ですが、これらの質問に答えることは必須ではありません。これらの質問は、プログラムの目標を示すことからはかけ離れたものです。かつて、アセンブリー言語のプログラマーは、メモリー・アライメント、メモリーレイアウト、スタックポインター、レジスター割り当てについて心配しなければなりませんでした。Fortran、C、C++、Java、および Python などの言語は、重要な細部を抽象化して、その処理をコンパイラーとライブラリーで解決するようになりました。同様に、私たちは、スレッド管理を抽象化することで、プログラマーが容易に並列処理を表現できるようにしました。

TBB を使用すると、プログラマーはより高い抽象レベルで並列処理を表現できます。適切に使用すると、TBB を使用したコードは暗黙的に並列化されます。

TBB の重要な概念は、プロセッサーよりも多くのタスクへプログラムを分割するということです。実用的な範囲で並列処理を指定し、実際にどれだけ並列処理が利用されるかは TBB に任せてください。

同時実行状態における安全性

同時実行（並行）性により問題が発生する可能性があるようなコードは、スレッドセーフではないと言われます。たとえ TBB が提供する抽象化を使用していても、スレッドの安全性の概念は不可欠です。スレッドセーフなコードとは、複数のスレッドが同じコードを使用する場合でも、期待どおりに機能することを保証する方法で記述されたコードです。コードがスレッドセーフでなくなる要因には、更新中に共有データへのアクセスを制御する同期の欠如（破損につながる可能性があります）や、不適切な同期（デッドロックにつながる可能性があります。これについては後述します）などがあります。

関数呼び出し間でお互いに持続的な状態を維持する場合、スレッドセーフであることを保証するよう注意深く記述する必要があります。同時に使用される可能性のある関数でのみこれを行う必要があります。一般に、同時に使用される可能性のある関数は、関数は同時使用が問題にならないように、相互に影響しないように記述すべきです。単一変数の設定やファイルの作成など、グローバルな作用が本当に必要な場合は、排他制御を呼び出しにより、副作用のあるコードを一度に 1 つのスレッドのみが実行できるよう注意する必要があります。

並行性や並列性を使用するコードでは、スレッドセーフなライブラリーを使用するのは必須です。使用するライブラリーがすべて、スレッドセーフであることを確認してください。C++ ライブラリーには、C から継承したいくつかの関数がありますが、それらの関数は呼び出し間で内部ステートを保持するためにスレッドでの同時利用には問題があります。特に `asctime`、`ctime`、`gmtime`、`localtime`、`rand`、`strtok` には注意してください。これらの関数を使用する場合、スレッドセーフなバージョンが利用可能であるかを必ずドキュメントを確認してください。C++ 標準テンプレート・ライブラリー（STL）コンテナークラスは、一般に、状態を維持するこれらの C 関数よりも安全です。すべての STL コンテナー関数は、異なるコンテナーで操作する場合、異なるスレッドから安全に呼び出すことができますが、同じコンテナーで異なるスレッドから呼び出すのは必ずしも安全ではありません。https://en.cppreference.com/w/cpp/container の「Thread Safety（スレッドの安全性）」の節に要約さているルールは簡単ではありません。3 章で説明する TBB の同時実行コンテナーは、同時アクセスを効率的かつ簡単にサポートするように設計されています。8 章では、TBB の同期機能をスレッドセーフでないコンテナーと組み合わせて使用して、スレッドセーフにする方法について説明します。

排他制御とロック

プログラム内で同じリソースへの同時アクセスが発生するか考慮する必要があります。最も関係すると思われるリソースはメモリーに保持されているデータですが、すべての種類のファイルと I/O についても考える必要があります。

共有データの更新において、正確な順序付けが必要な場合、何らかの形式の同期が必要になります。最良の手法は、同期が頻繁に行われないように問題を分解することです。これは、独立して動作するようにタスクを分割することで達成できます。発生する唯一の同期は、最後にタスクがすべて終了するのを待機することです。このような、「すべてのタスクが完了したとき」の同期は、一般にバリア同期と呼ばれます。バリアは、すべての並列処理を一時的に停止（シリアル化）させるため、粗粒度の並列処理に適しています。バリアを頻繁に使用すると、アムダールの法則に従ってスケーラビリティーが急速に失われます。

細粒度の並列処理では、データ構造の周囲で同期を使用して、書き込み中に他のユーザーによる読み取りと書き込みの両方を制限する必要があります。カウンターを 10 ずつ増やすなど、以前のメモリー値に基づいてメモリーを更新する場合、初期値の読み取りを開始した時点から新しく計算された値の書き込みが完了するまで、他の読み取りと書き込みを制限します。図 P-16 に、これを行う簡単な方法を示します。読み取りのみの場合であっても、複数の関連データを読み取るときは、データ構造の同期を使用して、読み取り中の書き込みを制限します。これらの制限は他のタスクにも適用され、*排他制御*と呼ばれます。*排他制御*の目的は、一連の操作を *アトミック*（不可分）に見せることです。

TBB は*排他制御*向けの移植可能なメカニズムを実装します。基本的に 2 つのアプローチがあります。非常に単純で一般的な操作（増分など）に対するアトミック（不可分）操作と、より長いコードシーケンスに対するロック/ロック解除メカニズムです。これらはすべて 8 章で説明します。

Thread A	Thread B	Value of X
LOCK (X)	*(wait)*	44
Read X (44)	*(wait)*	44
add 10	*(wait)*	44
Write X (54)	*(wait)*	54
UNLOCK (X)	*(wait)*	54
	LOCK (X)	54
	Read X (54)	54
	subtract 12	54
	Write X (42)	42
	UNLOCK (X)	**42**

図 P-16. 排他制御時に発生する可能性のあるシリアル化

`X = 44` で始まる 2 つのスレッドを持つプログラムを考えます。スレッド A は、`x = x + 10` を実行します。スレッド B は、`X = X - 12` を実行します。ロックを使用した場合（図 P-16）、スレッド A またはスレッド B どちらか一方だけがそのステートメントを実行でき、常に `X = 42` で終了します。両方のスレッドがロックを同時に取得しようとすると、1 つはブロックされ、ロックが取得できるようになるまで待たなければなりません。図 P-16 は、スレッド A とスレッド B がロックを同時に要求して、スレッド A が最初に取得しスレッド B がロックを取得できなかった場合、スレッド B がどれだけの時間を待機するかを示しています。

図 P-16 で使用するロックの代わりに、システムが *アトミック*（不可分）であることを保証する小さな操作セットを使用することもできます。ここでロックについて解説するのは、ロックはあらゆるコードシーケンスをアトミックに見せることができる一般的なメカニズムであるためです。このようなシーケンスは、アムダールの法則によるとスケーリングを低下させるため、常に、可能な限り短くする必要があります。特定のアトミック操作（インクリメントなど）が利用可能な場合、それが最も高速な方法であり、スケーリングの低下が最も少ないため、それを使用します。

Thread A	Thread B	Value of X
Read X (44)		44
add 10	Read X (44)	44
Write X (54)	subtract 12	54
	Write X (32)	**32**

RACE – A first, B second

Thread A	Thread B	Value of X
	Read X (44)	44
	subtract 12	44
	Write X (32)	32
Read X (32)		32
add 10		32
Write X (42)		**42**

DESIRED

Thread A	Thread B	Value of X
	Read X (44)	44
Read X (44)	subtract 12	44
add 10	Write X (32)	32
Write X (54)		**54**

RACE – B first, A second

図 P-17. 排他制御がないと競合状態によって問題が発生する可能性があります。ここでの簡単な修正は、読み取り-書き込み操作を、適切なアトミック操作（アトミック・インクリメントまたはアトミック・デクリメント）に置き換えることです。

私たちはロックを嫌っていますが、ロックがなければ事態はさらに悪化することを認めなければなりません。ロックのない場合を考えてみましょう。*競合状態*が存在し、少なくとも 2 つの結果が生成されると考えられます: x=32 または x=54（図 P-17）。非常に重要な概念である*競合状態*については、すぐに定義します。各ステートメントが x を読み取り、計算して x に書き込むため、誤った加算結果が生じる可能性があります。ロックがないと、スレッドが x の値を読み取るのが、他のスレッドが値を書き込む前か後かは保証されません。

正当性

並列思考を学ぶときの最も重要な課題は、並列性に関連する正当性を理解することです。並行性は、同時にアクティブな複数のスレッドを制御することを意味します。オペレーティング・システムは、さまざまな方法でそれらのスレッドをスケジュールします。プログラムを実行するごとに、操作の正確な順序は潜在的に異なります。プログラマーとしての課題は、どのような順番で並列プログラムが実行されても正しい結果が得られることを確実にすることです。TBB のような高レベルの抽象化は非常に役立ちますが、プログラムが結果を並列に計算する場合、潜在的な変化や、ロックを間違って使用した場合のプログラムのバグなど、自力で取り組まなければならない問題もあります。

並列で行われた計算は、しばしば元のシーケンシャル・プログラムと異なる結果になることがあります。演算の丸めによるエラーはプログラムを並列で実行することでプログラマーが経験する最も一般的な問題です。浮動小数点値を使用する場合、計算を並列実行するように変更すると、浮動小数点値の精度が制限されるため、結果の数値が変化することを予想する必要があります。

例えば、(A+B+C+D) を ((A+B)+(C+D)) として計算する場合、A+B と C+D を並列で計算すると、最後の合計が (((A+B)+C)+D) のように計算したときと若干異なることがあります。並列計算の結果は、計算の順番に依存して実行するたびに変わることさえあります。このような非決定的な動作は、多くの場合、実行時の柔軟性を減らすことによって制御できます。 本書では、そのようなオプション、特に決定論的なリダクション操作のオプション (2 章) について説明します。非決定性によりデバッグとテストが困難になる可能性があるため、決定論的な動作を強制することが望ましい場合もあります。状況によっては、同期が強制されるため、パフォーマンスが低下する場合があります。

いくつかの種類のプログラムの失敗は、タスクの実行順序が変化する並列プログラムでのみ発生します。これらの障害は*デッドロック*や*競合状態*として知られています。同時実行プログラムでは、独立して動作するタスクが多数存在し、実行パスも多数存在する可能性があるため、決定論も課題となります。

TBB はプログラミングを簡素化することで障害の発生率を減らしますが、それでも障害が発生する可能性はあります。マルチスレッド・プログラムは、同じプログラムで同じ入力データを使用しても毎回呼び出されるたびに異なる実行パスになりえるため、競合状態により非決定的になる可能性があります。このような状況が発生すると、障害が一貫して再現されるわけではなく、デバッガーを使用することで障害の状況が簡単に代わる可能性があるため、デバッグは特に困難になります。

不要な非決定性の原因を追跡して排除することは容易ではありません。インテル Advisor のような専門のツールは役に立ちますが、最初のステップは、これらの問題を理解して回避することです。

また、順次コードから並列コードに移行すると、結果の不安定性という、非決定性の兆候でもある別の問題も生じる可能性があります。結果の不安定性とは、ワークが実行される微妙な順序の変化によって、異なる結果が生成されることを意味します。一部のアルゴリズムは不安定になりますが、複数の正しい順序があると考えられるアルゴリズムにおいて、操作の順序を変更するよう設計されている場合は不安定になりません。

次に、並列プログラミングにおける 3 つの主なエラーとそれぞれの解決策について説明します。

デッドロック

少なくとも 2 つのタスクが互いを待機し、他のタスクが処理されるまでそれぞれが処理を再開できない場合、デッドロックが発生します。デッドロックは、コードが複数のロックの取得を要求する場合に簡単に発生します。タスク A がロック R とロック X を必要とする場合、タスク A はロック R を取得してからロック X を取得しようとする可能性があります。一方、タスク B が同じ 2 つのロックを必要とし、最初にロック X を取得した場合、タスク A はロック R を保持しながらロック X を要求し、タスク B はロック X のみを保持しながらロック R を待機するという状況に陥りやすくなります。その結果生じる行き詰まりは、一方のタスクが保持しているロックを解放した場合にのみ解決できます。どちらもロックを解放しない場合、デッドロックが発生し、タスクは永久に動きがとれません。

デッドロックの解決策

ロックを回避するために、可能な限り暗黙の同期を使用することをお勧めします。一般に、ロックを(特に同時に複数のロックを)使用しないようにしてください。ロックを取得してからロックを使用する関数やサブルーチンを呼び出すと、多くの場合、多重ロック問題の原因となります。 共有リソースへのアクセスが不可欠な場合があるため、最も一般的な解決策は、特定の順序 (例えば、常に A、B の順) でロックを取得する方法と、ロックを取得できない場合は常にすべてのロックを解放してからやり直す方法です。

競合状態

複数のタスクが適切な同期を行わないで同じメモリー位置を読み書きすると競合状態が発生します。「競合」が発生するコードは、正しく終了してエラーなしで完了することもあれば、問題を引き起こして終了することもあります。
図 P-17 は、競合状態によって 3 つの異なる結果が返される簡単な例を示しています。

競合状態はデッドロックよりも壊滅的ではありませんが、明らかな障害が発生しないでデータが破損する (間違った値が読み書きされる) ことがあるため、より有害です。また、複数のスレッドが状態の一部 (複数のデータ要素) のみの更新に成功する可能性があるため、予期しない (望ましくない) 状態になることがあります。

競合状態の解決策

8 章で説明されている同期メカニズムは、プログラムの正当性を保証するため、共有データを秩序ある方法で管理するのに役立ちます。間違いの元となるため、ロックに基づく低レベルのメソッドを使用しないでください。 明示的なロックは最後の手段としてのみ使用してください。可能な限り、アルゴリズム・テンプレートとタスク・スケジューラーによって暗黙に定義される同期を使用してください。例えば、独自の共有変数を作成する代わりに `parallel_reduce`（2 章で説明）テンプレートを使用します。`parallel_reduce` の `join`（結合）操作は、結合する領域の処理が完了するまで実行されないことが保証されます。

結果の不安定性（決定論的な結果の欠如）

並列プログラムでは通常、多くの同時タスクが、異なる呼び出し間で、また特にプロセッサー数が異なるシステム上でわずかな変化を伴って動作するため、毎回異なる答えを計算します。これについては、正当性に関する説明で触れました。

結果が不安定な場合の解決策

TBB は、実行時の柔軟性と引き換えに、より決定論的な動作を保証する方法を提供します。これによりパフォーマンスが低下する可能性がありますが、決定論によってもたらされる利点はしばしばそれに値します。決定論については 2 章で説明します。

抽象化

プログラムを記述するとき、適切なレベルの抽象化を選択することが重要です。現在ではアセンブリー言語を使用しているプログラマーはほとんどいません。C や C++のようなプログラミング言語は低レベルの処理を行わないように抽象化されています。古いプログラミング手法を復活させたい開発者はいないでしょう。

並列処理でも同じです。低レベルのコードを記述すると問題が生じる可能性があります。ネイティブ・スレッド・プログラミングでは、エラーが発生しやすいスレッドを管理することが要求されます。

TBB を使用したプログラミングでは、スレッドを管理は抽象化されます。そのため、より容易に作成そして維持できる、より洗練されたコードを記述できます。実際に、TBB を使ったコードはコードの移植性とパフォーマンスの移植性が高いことがわかっています。ただし、このアプローチでは、どの作業を分割できるのか、また、データをどのように分割できるのかという観点からアルゴリズムを考える必要があります。

パターン

経験豊富な並列プログラマーは、一般的な問題には既知の解決策があることを知っています。これはあらゆるプログラミングに共通しており、その結果、スタック、キュー、リンクリストなどの概念があります。並列プログラミングでは、マップ、レデュース、パイプラインなどの概念が導入されます。

これらを*パターン*と呼び、特定のニーズに合わせて適応できます。共通のパターンを学ぶことは、先人たちから学ぶ最適な方法です。TBB は主要なパターンに対するソリューションを実装しているため、TBB を学習するだけでそれらを暗黙のうちに学習できます。2 章では、一般的なパターン名と TBB アルゴリズムを関連付けた表を示しています（図 2-1 を参照）。

局所性とキャッシュの逆襲

効果的な並列プログラミングには、*局所性*の重要性を認識することが必要です。それには、ハードウェア、特にメモリー*キャッシュ*について簡単な説明が必要かもしれません。「キャッシュ」とは、一種のハードウェア・バッファーであり、最近アクセスまたは変更したデータを、大容量のメインメモリーからではなく、より近い場所に保持することで、アクセス速度を高速化するものです。キャッシュの目的は処理を高速化することであるため、プログラムでキャッシュを有効活用すれば、アプリケーションは高速に動作する可能性があります。

現代のコンピューター設計は、一般に複数レベルのキャッシュ・ハードウェアで構成されており、各レベルがキャッシュであるため、「cache」ではなく「caches」と表記されます。日本語訳では、特に問題がない限り両方を「キャッシュ」と表記しています。本書の目的のためには、キャッシュを単一のデータのコレクションとして考えれば十分です。

*キャッシュ*を深く理解する必要はありませんが、キャッシュを高レベルで理解することで、局所性、*可変状態*の共有に関連する問題、そして特に厄介な*フォルス・シェアリング*と呼ばれる現象を理解するのに役立ちます。

キャッシュの影響である局所性、共有、およびフォルス・シェアリングには注意を払うべきです。これらを理解するには、キャッシュとキャッシュラインを理解する必要があります。これらは、すべての現代のコンピューター設計の基本を成しています。

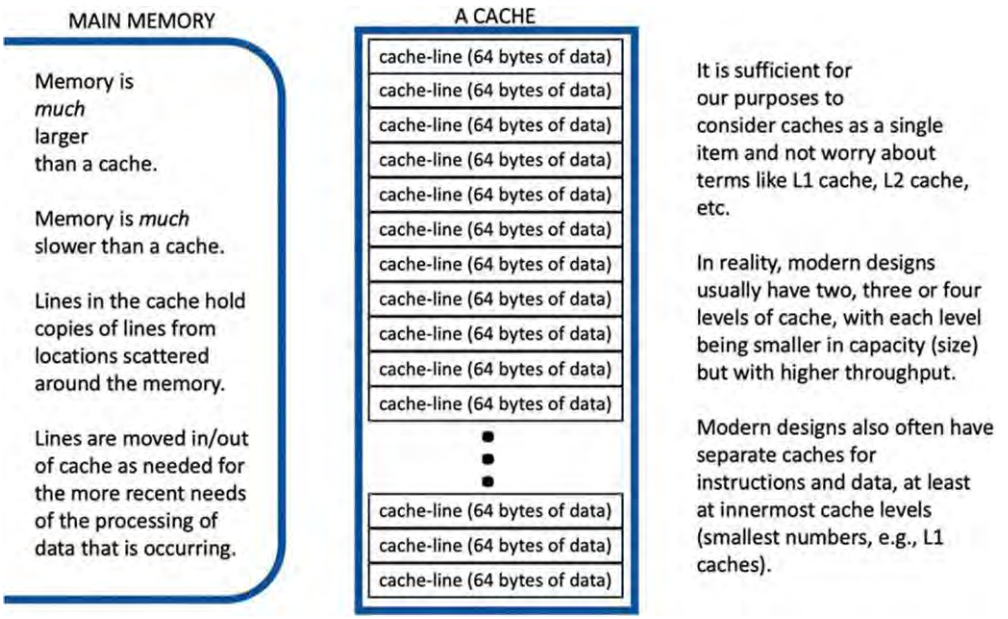


図 P-18. メインメモリーとキャッシュ

ハードウェアの動機

一般的に、特定のシステムに特化すればするほど、移植性とパフォーマンスの移植性が失われるため、ハードウェア実装の詳細についてはできる限り無視したいと考えていますが、キャッシュは例外です（図 P-18）。

メモリー*キャッシュ*は、すべての最新のコンピューター設計に搭載されており、ほとんどのシステムには複数レベルのキャッシュがあります。常にそうではありませんが、当初、コンピューターは必要なときにのみメモリーからデータと命令を取得し、結果をすぐにメモリーに書き込みました（現代よりもシンプルでした）。

プロセッサーの速度は、メインメモリーよりもはるかに速くなりました。すべてのメモリーをプロセッサーと同じ速度にすることは、ほとんどのコンピューターでは非常に高価になり、現実的ではありません。代わりに、プロセッサーの設計者は、プロセッサーに近い速度で動作する*キャッシュ*と呼ばれる少量のメモリーを実装しました。メインメモリーは、キャッシュより速度は遅い代わりに安価です。ハードウェアは、必要に応じて情報をキャッシュに出し入れする方法を認識しており、これにより、メモリーとプロセッサー・コアの間でデータがやり取りされる場所が増えます。キャッシュは、メモリー速度とプロセッサー速度の不一致を解消する上で非常に重要です。

すべてのコンピューターは、最終的にメインメモリーに格納されるべきデータの**一時コピー**としてのみキャッシュを利用しています。したがって、メモリー・サブシステムの機能は、各プロセッサー・コアが入力するデータをコアの近くのキャッシュへ移動して、コアが出力したデータをメインメモリーへ移動することです。データがメモリーからキャッシュへ移動する際、新しく要求されたデータを格納するスペースを空けるため、キャッシュから一部のデータを退避する必要が生じる場合があります。プロセッサー設計者は、再使用されないと予想されるデータを退避します。利用頻度の少ないデータは、近い将来にも必要となる可能性が低いと考えられます。そうすることで、貴重なスペースであるキャッシュに、再利用される可能性が最も高いデータを保存します。

プログラムがデータにアクセスする場合、データがキャッシュに存在する間にデータの処理を終わらせるのが最良です。継続的にデータを使用するとデータはキャッシュで保持されますが、未使用の期間が続くとデータはキャッシュから排出されます。排出されたデータを再び使用すると、メモリーからデータを取得することになるため、コストの高い（遅い）アクセスが必要になります。さらに、プロセッサー上で新しいスレッドが実行されるたびに、特定のスレッドに必要なデータを保持するスペースを確保するため、キャッシュからデータが退避される可能性があります。

参照の局所性

最初にメモリーからデータを取得するのはコストがかかりますが、取得後に一定期間データを使用するとはるかに安価になります。これは、キャッシュが情報を保持する仕組みが、私たちの短期記憶と似ているためです。短期記憶では、その日に経験したことは簡単に思い出せますが、数ヶ月後には思い出すのが難しくなります。

よく引用される簡単な行列乗算の例 C=AxB（サイズ n x n の行列 a、b、c）を図 P-19 に示します。

```
for(i=0;i<n;i++)(i=0;i<n;i++)
      for(j=0;j<n;j++)
         for (k=0;k<n;k++)
             c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

図 P-19. 参照の局所性が低い行列乗算

C と C++ では配列は行優先で格納されます。つまり、連続する配列要素は最後の配列次元にあります。これは、c[i][2] と c[i][3] はメモリー内で隣接しているのに対し、c[2][j] と c[3][j] は離れている（例では n 要素離れている）ことを意味します。

図 P-20 に示すように、j と k のループ順序を入れ替えると、参照の局所性が大幅に向上するため、劇的にスピードアップします。これにより、通常、数学的な結果は変わることはありませんが、メモリーキャッシュが効果的に利用されるため、効率が向上します。この例で n の値は、行列の合計サイズがキャッシュサイズを上回るよう十分に大きくする必要があります。小さいと、どちらの順序でもキャッシュ内に収まるため、順序はそれほど重要ではなくなります。n の値が 10,000 の場合、各行列には 1 億個の要素が含まれます。倍精度浮動小数点値の場合、3 つの行列は合計 2.4 GB のメモリーを占有します。これにより、本書出版時点では、すべてのマシンでキャッシュ効果を観察できるでしょう。ほとんどのコンピューターはインデックスの順序を入れ替えることで十分な恩恵を受けると考えられますが、n が小さく、すべてのデータがキャッシュに収まる場合、全く効果は見られないでしょう。

```
for(i=0;i<n;i++)
    for(k=0;k<n;k++)
        for (j=0;j<n;j++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

図 P-20. 参照の局所性を改善した行列乗算

キャッシュライン、アライメント、共有、排他制御、およびフォルス・シェアリング

キャッシュは行（ライン）ごとに管理されます。プロセッサーは、キャッシュライン単位でメインメモリーとキャッシュ間でデータを転送します。これにより、データのアライメント、データの共有、およびフォルス・シェアリングという 3 つの懸念事項が発生します。これらについて説明します。

キャッシュライン長は、ある程度任意ですが、現在最も一般的なのは 512 ビット（64 バイト）であり、8 個の倍精度浮動小数点数、または 16 個の 32 ビット整数を格納できます。

一貫性/分散キャッシュ

最も内側かつ最速のキャッシュは、計算ユニット、具体的にはプロセッサー・コアの近くに配置されています。並列システムでは、最も内側のキャッシュが分散され、複数の小さなキャッシュがそれぞれのプロセッサー・コアの近くにデータを保持します。この分散により、すべてのデータのコピーは一貫性を保つ必要があり、キャッシュ一貫性と呼ばれる課題が発生します。

これについては、これ以上詳しく説明しません。キャッシュの一貫性がどのように維持されるか、また、そのような設計に伴うさまざまなトレードオフについて、より深く理解するには、コンピューター・アーキテクチャーの講義や書籍が役立つでしょう。(訳者注) 一部の記事を紹介しておきます (https://www.isus.jp/products/vtune/pu38-02-detecting-and-mitigating-false-sharing/、および https://www.isus.jp/products/c-compilers/data-and-code-in-order-optimization-and-memory-part-1/)。

メカニズムが機能すると想定していますが、特にキャッシュ間でデータを転送する場合、一貫性を維持するにはコストがかかることを知っておくべきです。さらに、複数のプロセッサー間で同じデータにアクセスすると、特にそのデータが更新される場合は、コストがかかります。

これらの問題によるパフォーマンスの低下を軽減するには、次の 2 つのことを行う必要があります: (1) プログラムを分解して、タスク間の共有データへの依存関係を最小限に抑え、(2) 偽の依存関係 (一般にフォルス・シェアリングと呼ばれる) を排除します。最初のアプローチは、本書全体、そして並列プログラミングに関するあらゆる文献で繰り返し登場するテーマであり、開発者にとって継続的な追求事項です。2 番目のアプローチについては、「フォルス・シェアリング」の節で説明します。

アライメント

任意のデータ項目 (`int`、`long`、`double`、`short` など) が同一キャッシュライン内に収まっている方がはるかに適切です。図 P-18 または図 P-21 を参照して、単一のデータ項目 (例えば、double) が 2 つのキャッシュラインにまたがっている場合を考えてみましょう。その場合、2 つのキャッシュラインにアクセス (読み取りまたは書き込み) する必要があります。通常、これには 2 倍の時間がかかります。単一のデータ項目をキャッシュラインにまたがらないように配置することは、パフォーマンスにとって重要です。一部のハードウェアには、アライメントされていないデータ (多くの場合、不整列データと呼ばれます) に対するペナルティーを軽減する機能が備わっています。このようなサポートは常に利用できるとは限らないため、データを自然な境界に沿って配置することを推奨します。配列は、非常に小さい場合を除いて、通常、キャッシュラインにまたがります。配列が複数のキャッシュラインにまたがる場合でも、1 つの要素が 2 つのキャッシュラインにまたがらないように、配列を単一の配列要素のアライメント・サイズに合わせることをお勧めします。同じアドバイスは構造体にも当てはまります。小さな構造体をキャッシュラインに収まるように配置することには、いくつかの利点があります。

アライメントの欠点は、スペースが無駄になることです。データをアライメントするたびに、一部のメモリーがスキップされる可能性があります。一般に、メモリーは安価であるため、これは単に無視できるでしょう。アライメントが頻繁に発生する場合、それを回避したり、メモリーを節約するために再配置することが、稀に重要になることもあります。そのため、アライメントの欠点についても触れる必要がありました。一般的に、アライメントはパフォーマンスの観点からは重要であるため、適用する必要があります。 コンパイラーは、配列や構造体を含む変数を要素サイズに合わせて自動的にアライメントします。メモリー割り当て (`malloc` など)

を使用する場合、明示的にアラインメントする必要がありますが、その方法については 7 章で説明します。

本書でアライメントについて説明する本当の理由は、共有とフォルス・シェアリングの弊害について議論するためです。

共有

メモリーから不変（変更不可能）なデータのコピーを共有するのは簡単です。不変データを共有しても、並列処理に特別な問題は発生しません。

データ並列処理を行うときに大きな課題を引き起こすのは、変更可能なデータです。前の節で、*共有された可変状態*は並列処理の敵だと言いました。一般的に、タスク間では変更可能なデータの共有は最小限に抑える必要があります。共有が少ないほど、デバッグするポイントが少なくなり、間違いも少なくなります。現実には、データを共有することで、並列タスクが同じ問題に取り組んでスケーリングを達成できるため、データを正しく共有する方法について議論する必要があります。

共有された変更可能な状態によって、次の 2 つの課題が生じます: (1) 順序付け、および (2) フォルス・シェアリング。1 つ目は並列プログラミング固有のものであり、ハードウェアによって発生するものではありません。前述の*排他制御*の説明では、正当性に関する懸念を図 P-17 で示しました。これは、すべての並列プログラマーが理解すべき重要なトピックです。

フォルス・シェアリング

データはキャッシュライン内で移動されるため、完全に独立した複数の変数やメモリー割り当ては、すべてまたは一部が同じキャッシュライン内に存在する可能性があります（図 P-21）。このキャッシュラインの共有によってプログラムの実行が失敗することはありません。しかし、パフォーマンスが大幅に低下する可能性があります。複数のタスクで使用される可変データのキャッシュライン共有の問題を完全に説明するには、多くのページが必要になります。簡単に説明すると、あるタスクがキャッシュライン内のいずれかのデータを更新すると、他のタスクからの同一キャッシュラインへのアクセスが遅くなる可能性があるということです。

フォルス・シェアリングによってマシンの速度が低下する詳しい理由はさておき、適切に作成された並列プログラムではフォルス・シェアリングを回避する対策が講じられています。あるマシン構成が他のマシン構成よりも影響が少ない場合でも、パフォーマンスの移植性を確保するには、常にフォルス・シェアリングを回避する対策を講じる必要があります。

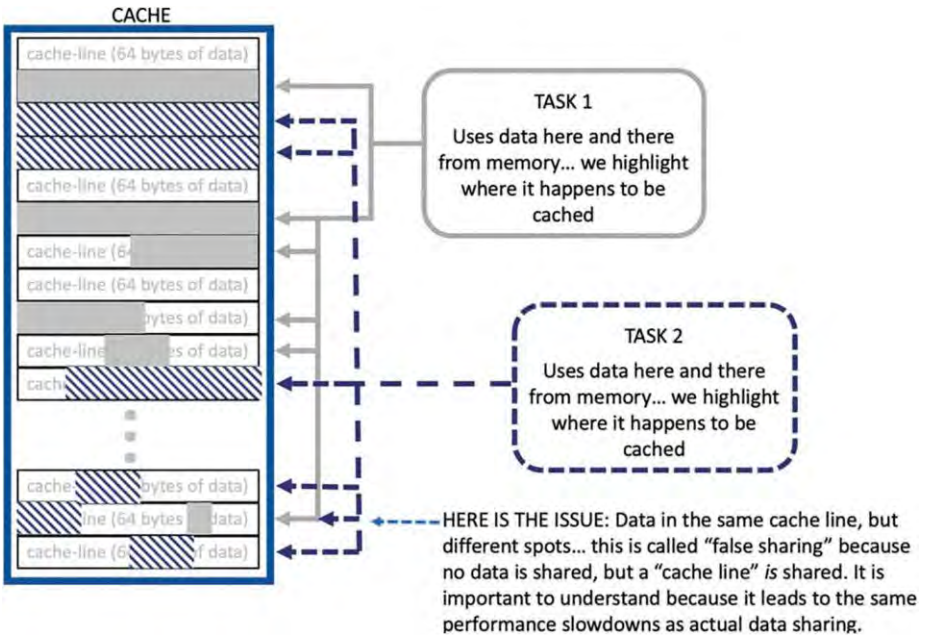


図 P-21. フォルス・シェアリングは、2 つの異なるタスクのデータが同じキャッシュラインに配置される場合に発生します。

なぜ**フォルス・シェアリング**がパフォーマンスのペナルティーとなるのか説明するため、2 つのスレッドが互いに近くのメモリーにアクセスするときに発生するキャッシュとオペレーティング・システムのオーバーヘッドを示します。ここでは、キャッシュラインは 64 バイトで、2 つのスレッドがキャッシュを共有するプロセッサー上で実行され、プログラムではスレッドがスレッド ID に基づいてアクセスおよび更新する配列を定義していると仮定します:

```
int my_private_counter[MAX_THREADS];
```

my_private_counter 内の連続する 2 つのエントリーは、同じキャッシュラインにある可能性があります。したがって、このサンプルプログラムでは、異なるスレッドで使用されるデータが同じキャッシュラインに配置されることによって引き起こされるフォルス・シェアリングにより、オーバーヘッドが発生する可能性があります。コア 0 とコア 1 でそれぞれ実行されている 2 つのスレッド 0 と 1 と、次の一連のイベントについて考えます:

スレッド 0 は my_private_counter[0] をインクリメントします。これは、コア 0 のプライベート・キャッシュの値を読み取り、カウンターをインクリメントして結果を書き戻します。正確には、コア 0 はこのカウンターを含むライン全体（例えば 64 バイト）をメモリーからキャッシュに読み込み、カウンターを新しい値（通常はキャッシュラインのみ）に更新します。

次に、スレッド 1 が my_private_counter[1] をインクリメントすると、フォルス・シェアリングによるオーバーヘッドが発生します。my_private_counter の位置 0 と 1 が同じキャッシュラインに配置される可能性が非常に高くなり、コア 1 のスレッド 1 がカウンターを読み取ろうとすると、キャッシュ・コヒーレンス・プロトコルが機能します。このプロトコルはキャッシュライン・レベルで動作し、スレッド 1 がスレッド 0 によって書き込まれた値を読み取っ

ている（スレッド 0 とスレッド 1 が実際に 1 つのカウンターを共有しているように）と想定します。したがって、コア 1 はコア 0 からキャッシュラインを読み取る必要があります（最も低速な手段は、コア 0 のキャッシュラインをメモリーにフラッシュし、そこからコア 1 が読み取ることです）。これはすでに高コストですが、スレッド 1 がこのカウンターをインクリメントし、コア 0 のキャッシュラインのコピーを無効化すると、さらに状況が悪化します。

ここで、スレッド 0 が `my_private_counter[0]` を再度インクリメントすると、そのカウンターは無効化されているため、コア 0 のローカルキャッシュ内に見つかりません。更新された最新のコア 1 のラインにアクセスするには、さらにコストが必要です。再度、スレッド 0 がこのカウンターをインクリメントすると、コア 1 のコピーは無効になります。この動作が続くと、スレッド 0 とスレッド 1 がそれぞれのカウンターにアクセスする速度は、各カウンターが異なるキャッシュラインに配置されている場合より大幅に低下します。

この問題は「フォルス・シェアリング」と呼ばれます。これは、実際に各スレッドがプライベート（共有されていない）カウンターを持っているにもかかわらず、キャッシュ・コヒーレンス・プロトコルがキャッシュライン・レベルで動作し、両方のカウンターが同じキャッシュラインを「共有」しているため、ハードウェアの観点からは 2 つのカウンターが共有されているように見えるためです。

ここで、おそらく簡単な解決策は、キャッシュ・コヒーレンス・プロトコルのハードウェア実装を修正して、ラインレベルではなくワードレベルで動作するようにすることだと考える方もいるでしょう。しかし、このハードウェアによる代替は法外なコストがかかるため、ハードウェア・ベンダーはソフトウェアで問題を解決するよう求めています。（訳者注）おまけにワードレベルでキャッシュ・コヒーレンス・プロトコルを実装するのは現実的ではありません。高コストなキャッシュ・コヒーレンス・メカニズムの起動が必要ないことをハードウェアに明確に伝えるため、各プライベート・カウンターを異なるキャッシュラインに配置するようにします。

フォルス・シェアリングによって、莫大なオーバーヘッドが簡単に発生することが分かります。この例では、要素を分散させて異なるキャッシュラインに配置するのが適切な解決策です。 これにはいくつかの方法がありますが、その 1 つは、フォルス・シェアリングのリスクを持つすべてのデータを一緒に割り当てるのではなく、特定のスレッドに必要なデータに対して TBB のキャッシュ・アラインメント・アロケーター（7 章で説明）を使用することです。

アライメントによってフォルス・シェアリングを回避

フォルス・シェアリング（図 P-21）を回避するには、並列に更新される可能性のある異なる変更可能なデータが同じキャッシュラインに配置されないようにする必要があります。これは、アライメントとパディングを組み合わせて行います。

配列や構造体などのデータ構造をキャッシュライン境界に合わせると、データ構造の開始位置が他の何かによって使用されているキャッシュラインに配置されるのを防ぐことができます。通常、これはキャッシュラインにアライメントされた `malloc` バージョンを使用することを意味します。7 章では、キャッシュ・アラインメント・メモリー・アロケーター（`tbb::cache_aligned_allocator` など）を含むメモリー割り当てについて説明しています。静的変数とローカル変数をコンパイラー・ディレクティブ（プラグマ）で明示的にアライメントすることもできますが、その必要性ははるかに低くなります。
メモリー割り当てによるフォルス・シェアリングのパフォーマンス低下は、多くの場合、ある程度非決定的であり、アプリケーションの一部の実行には影響しますが、他の実行には影響しない可能性があることにも注意する必要があります。これには本当に混乱します。なぜなら、アプリケーションの実行で毎回問題が発生するとは限らないため、非決定的な問題のデバッグは非常に困難であるためです。

TBB はキャッシュを考慮

TBB は、タスクとデータの不要な移動を避けるよう、キャッシュを考慮して設計されています。TBB がタスクを別のプロセッサーに移動する場合、現在のプロセッサー・キャッシュにデータが存在する可能性が最も低いタスクを選択して移動します。これらの考慮事項は、ワーク（タスク）スチール・スケジューラーに組み込まれているため、2 章で説明するアルゴリズム・テンプレートの一部となっています。

キャッシュの取り扱いについては多くの章で取り上げていますが、注目すべき章をいくつか挙げると次のようになります:

7 章では、フォルス・シェアリングを回避するアライメントやパディングなど、キャッシュに役立つメモリー・アロケーターに関連する重要な考慮事項について説明します。

8 章では、プライベート化とリダクションについて説明します。

11 章では、データの局所性を取り上げ、局所性を向上させるオプションのチューニングについて詳しく説明します。

最高のパフォーマンスを得るには、プログラムの構造を検討する際にデータの局所性を考える必要があります。アプリケーションを設計して、集中した時間内に単一のデータセットを使用できる場合は、データ領域を散発的に使用することを避ける必要があります。

興味深いことに、並列クイックソートはキャッシュ効率が最大並列処理を上回る例です。並列マージソートは、並列クイックソートよりも並列性が高くなります。しかし、並列マージソートはインプレース（データを直接入れ換える）のソートではないため、キャッシュの容量が並列クイックソートの 2 倍必要になります。そのため、実際にはクイックソートの方が高速に実行されるのが一般的です。

プログラムの構造を検討する際に、データの局所性を考慮してください。アプリケーションを設計して、集中した時間内に単一のデータセットを使用できる場合、データ領域を散発的にすることを避ける必要があります。特にプログラムの上位レベルでデータ分解を行うと、この問題が自然に発生します。

TBB はタイムスライスのコストを考慮

タイムスライスにより、物理スレッドよりも多くの論理スレッドを存在させることが可能になります。各論理スレッドは、物理スレッドによって「タイムスライス」という短い期間だけ処理されます。この期間は、オペレーティング・システムによって定義されており、スレッドはプリエンプトされるまで動作できます。多くのスレッドがそうであるように、もし、スレッドがタイムスライスよりも長く実行された場合、次の順番が回ってくるまで物理スレッドを明け渡します。

最も明らかなコストは、論理スレッド間のコンテキスト・スイッチにかかる時間です。コンテキスト・スイッチでは、プロセッサーで実行中の論理スレッドのすべてのレジスターを退避し、次に実行する論理スレッドの情報を含むレジスターをロードする必要があります。

それよりもコストがかかるのは、キャッシュの冷却と呼ばれる現象です。プロセッサーは、最近アクセスされたデータをキャッシュに保持します。キャッシュはメインメモリーに比べると、非常に高速ですが、容量は小さいものです。プロセッサーは、キャッシュを使い果たすと、キャッシュからデータを排出し、メインメモリーに戻します。これには、通常はキャッシュの中で最も古いデータが選ばれます（実際のセット・アソシエーティブ・キャッシュ（連想方式キャッシュ）ではもう少し複雑です）。

論理スレッドがタイムスライスを取得し、データを初めて参照すると、数百サイクルを消費しデータはメモリーからキャッシュに読み込まれます。データが頻繁に参照され、退避されない場合、後続の参照はキャッシュ中のデータを利用できるため、数サイクルを要するだけです。そのようなデータは、*キャッシュ*上で*ホット*であると呼ばれます。

タイムスライスでは、スレッド A がそのタイムスライスを終了し、後続のスレッド B が同じ物理スレッドで実行されると、両方のスレッドでそのデータが必要でない限り、B は A が利用したキャッシュでホットだったデータを排出する傾向があります。スレッド A が次のタイムスライスを取得すると、それぞれのキャッシュミスに数百サイクルを費やして、排出によって退避されたデータを再びロードします。さらに、スレッド A の次のタイムスライスは、異なるキャッシュを持つ異なる物理スレッド上にある可能性もあります。

もう 1 つのコストは*ロックのプリエンプション*です。これは、スレッドがロックを取得し、そのタイムスライスが、ロックを解放する前に時間切れになったときに発生する現象です。スレッドがロックを保持する時間が短かったとしても、少なくともタイムスライスで次の順番が来るまでの間、ロックを保持することになります。ロック上で待機している別のスレッドは、無駄にビジーウェイトになったり、あるいは、残りのタイムスライスを失うことになります。これは、前のプリエンプトされたスレッドの実行が再開されるのを “数珠つなぎ” に待機するため、コンボイと呼ばれます。

ベクトル化（SIMD）の概要

並列プログラミングとは、最終的にはハードウェアの並列計算機能を活用することです。本書では、複数のプロセッサー・コアを活用する並列処理に焦点を当てています。このようなハードウェア並列処理は、スレッドレベル（または抽象的にはタスクレベル）の並列処理によって利用され、TBB がそれを解決します。

重要なハードウェア並列処理にはもう一つ、ベクトル命令と呼ばれる別のクラスがあります。ベクトル命令は、通常の命令よりも多くの計算を（並列に）実行できる命令です。例えば、通常の加算命令は 2 つの数値を受け取って加算し、1 つの結果を返します。8 ウェイのベクトル加算では、8 組の入力を処理し、それぞれを個別に加算して 8 つの出力を生成できます。通常の加算命令の `C=A+B` の代わりに、単一のベクトル加算命令では `C0=A0+B0`、`C1=A1+B1`、`C2=A2+B2` … および `C7=A7+B7` が得られます。これにより、パフォーマンスが 8 倍向上します。ベクトル命令では、一度に 8 つの同じ操作を実行する必要がありますが、これは数値演算を主とするコードではよくあることです。

単一の命令で複数の操作を実行するこの機能は、SIMD（Single Instruction Multiple Data）と呼ばれ、コンピューター科学者によって Flynn の分類法として知られているコンピューター・アーキテクチャーの 4 つの分類の 1 つにあたります。

ベクトル化は SIMD 並列性を活用する技術であり、コンパイラーがハードウェア命令を選択するため、コンパイラーに依存する技術です。

本書ではベクトル並列性については無視して、「これは別の主題であり、勉強すべき内容も異なる」と言うこともできますが、そうはしません。優れた並列プログラムでは通常、タスク並列性（TBB を使用）と SIMD 並列性（ベクトル化コンパイラーを使用）の両方が使用されます。

皆さんが使用するコンパイラーのベクトル化の機能について学習し、使用することをお勧めします。`#pragma SIMD` は、最近のコンパイラーで人気の機能の 1 つです。また、Parallel STL のオプション（oneAPI DPC++ ライブラリー（oneDPL）など）を理解することも役立ちます。

これには、それが効果的な並列プログラミングにとって一般に最善のソリューションではない理由（アムダールの法則では、並列処理は STL 呼び出しよりもプログラムの上位レベルに配置することが推奨されます）も含まれます。

ベクトル化は重要ですが、TBB を使用すると、ほとんどのアプリケーションでスピードアップが実現できます（どちらかを選択する場合）。これは、システムでは通常、SIMD レーン（ベクトル命令の幅）よりもコアによる並列性の恩恵が高く、さらにタスクは SIMD 命令で使用できる限られた数の操作よりもはるかに汎用的であるためです。

マルチタスクとベクトル化に関するアドバイス
良いアドバイス: まず TBB を使用してコードをマルチタスク化し、次にベクトル化します。 最良のアドバイス: 両方を行います。

プログラムにベクトル化の恩恵を得られる計算が含まれる場合、通常、両方を行います。AVX ベクトル命令を備えた 32 コアのプロセッサー考えてみましょう。マルチタスク・プログラムでは、TBB を使用することで、理論上の最大 32 倍のパフォーマンスを実現できる可能性があります。ベクトル化されたプログラムでは、倍精度の数学コードをベクトル化することで、理論上の最大 4 倍のパフォーマンスを実現できる可能性があります。これらを合わせると理論上のパフォーマンスの最大向上は 256 倍になります。この乗法効果のため、数値演算を主とするプログラムの多くの開発者は、常に両方を行います。

C++ の機能紹介（TBB に必要な）

並列プログラミングの目的は、コア数の多いマシンでアプリケーションのパフォーマンスを向上させることであるため、C と C++ はどちらも効率を重視した抽象化の組み合わせを提供します。TBB は C++ をベースにしますが、C プログラマーにも使いやすい方法で作られています。

どの分野にも独自の用語がありますが、C++ も例外ではありません。本書の末尾には、C++、並列プログラミング、TBB などの用語を理解するのに役立つ用語集を掲載しています。ここでは、C++ プログラマーにとって基本的な用語として、ラムダ関数、ジェネリック・プログラミング、コンテナー、テンプレート、標準テンプレート・ライブラリー（STL）、オーバーロード、レンジ、イテレーターなどをいくつか確認します。

ラムダ関数

C++11 標準にラムダ関数が組み込まれ、コードを匿名関数としてインライン展開できるようになったため、TBB を使用したコードを読み取るのが簡単になりました。

ラムダ式のサポートは C++11 で導入されました。これらは、囲みのあるスコープから変数をキャプチャーできる匿名関数オブジェクト（名前付き変数に割り当てることも可能）を作成するのに使用されます。C++ ラムダ式の基本構文は、

```
[capture-list] (params) ->ret{body}
```

です。ここで、

- `capture-list` はカンマで区切られたキャプチャー・リストです。キャプチャー・リストに変数名をリストすることで、変数を値でキャプチャーします。変数を参照によってキャプチャーするには、`&v` のように、先頭にアンパサンド（`&`）を付けます。これにより、現在のオブジェクトを参照によってキャプチャーできます。デフォルトもあります: `[=]` は、ボディーで使用されるすべての自動変数を値でキャプチャーし、現在のオブジェクトを参照でキャプチャーするために使用され、`[&]` は、ボディーで使用されるすべての自動変数と現在のオブジェクトを参照でキャプチャーするのに使用されます。`[]` は何もキャプチャーしません。

- `params` は、名前付き関数と同様に、関数のパラメーター・リストです。

- `ret` は戻り型です。`->ret` が指定されていない場合、`return` 文から推測されます。

- `body` は関数ボディーです。

次の例は、変数 `i` を値で取得し、変数 `j` を参照で取得する C++ ラムダ式を示しています。また、パラメーター `k0` と、参照によって受け取る別のパラメーター `l0` もあります:

```
int  main(int  argc, char  *argv[]) {
  int  i = 1, j = 10, k = 100, l = 1000;
  auto  lambdaExpression = [i, &j] (int  k0, int& l0) -> int {
    j = 2  * j;
   k0 = 2  * k0;
   l0 = 2  * l0;
    return  i + j + k0 + l0;
  };

  printValues(i, j, k, l);
  std::cout << "First call returned " << lambdaExpression(k, l) << std::endl;
  printValues(i, j, k, l);
  std::cout << "Second call returned " << lambdaExpression(k, l) << std::endl;
  printValues(i, j, k, l);
  return  0;
}
```

例を実行すると、次の出力が生成されます:

```
i == 1
j == 10
k == 100
l == 1000
First call returned 2221
i == 1
j == 20
k == 100
l == 2000
Second call returned 4241
i == 1
j == 40
k == 100
l == 4000
```

ラムダ式は関数オブジェクトのインスタンスと考えることができますが、クラス定義はコンパイラーによって生成されます。例えば、前の例のラムダ式は、クラスのインスタンスに類似しています:

```
int  main(int  argc, char  *argv[]) {
  int  i = 1, j = 10, k = 100, l = 1000;
  Functor f{i,j};

  PrintValues(i, j, k, l);
  std::cout << "First call returned "  << f(k, l) << std::endl;
  PrintValues(i, j, k, l);
  std::cout << "Second call returned "  << f(k, l) << std::endl;
  PrintValues(i, j, k, l);
  return  0;
}
```

C++ ラムダ式を使用する場合はどこでも、前述のような関数オブジェクトのインスタンスで置き換えることができます。実際、TBB ライブラリーは C++11 標準よりも古く、当初すべてのインターフェイスでユーザー定義クラスのオブジェクトのインスタンスを渡す必要がありました。C++ ラムダ式は、TBB アルゴリズムを使用するたびにクラスを定義する余分な手順を排除することで、TBB の使用を簡素化します。

ジェネリック・プログラミング

ジェネリック・プログラミングとは、あらゆるデータ型に対して汎用的に動作するアルゴリズムを記述することです。これらは「後で指定する」パラメーターを使用する、と考えることができます。C++ は、コンパイル時の最適化を優先し、型に基づいて実行時に選択する必要がないように、ジェネリック・プログラミングを実装します。これにより、最新の C++ コンパイラーを高度に調整して、ジェネリック・プログラミング、テンプレートと STL の多用によって生じる抽象化のペナルティーを最小限にすることが可能になりました。ジェネリック・プログラミングの簡単な例として、項目にアクセスし、2 つの項目をスワップし、2 つの項目を比較する方法を提供することで、任意の項目のリストを並べ替えるソートアルゴリズムが挙げられます。このようなアルゴリズムを組み立てたら、各データ型に対してスワップと比較を実行する方法を定義してインスタンス化することで、整数、浮動小数点数、複素数、文字列のリストをソートすることができます。

ジェネリック・プログラミングの簡単な例は、複素数のサポートを考慮することから始まります。複素数を構成する 2 つの要素は、`float` 型、`double` 型、または `long double` 型になります。3 種類の複素数を宣言し、さまざまな型を操作する 3 セットの関数を用意するのではなく、ジェネリック・プログラミングにより汎用複素数データ型の概念を作成できます。実際の変数宣言には、複素数変数に必要な要素の型を指定する次のいずれかの宣言を使用します:

```
complex<float> my_single_precision_complex;
complex<double> my_double_precision_complex;
complex<long  double> my_quad_precision_complex;
```

これらは、適切なヘッダーファイルをインクルードすることで、C++ の C++ 標準テンプレート・ライブラリー（STL - 定義は近日公開予定）でサポートされます。

コンテナー

「**コンテナー**」とは、データ項目のコレクションを整理および管理する「構造体」を表す C++ 用語です。C++ コンテナーは、オブジェクト指向の機能（「構造体」を操作できるコードを分離）とジェネリック・プログラミングの特性（コンテナーは抽象的であり、さまざまなデータ型を操作できる）の両方を備えています。TBB が提供するコンテナーについては、3 章で説明します。TBB を使用する上でコンテナーを理解することは優先事項ではありません。コンテナーは主に、すでにコンテナーを理解して使用している C++ ユーザー向けに TBB によってサポートされます。

テンプレート

テンプレートは、効率良い汎用プログラミング方式で関数またはクラス（コンテナーなど）を作成するパターンです。つまり、テンプレートの仕様は型に対して柔軟ですが、実際にコンパイルされたインスタンスは特定のものであるため、この柔軟性によるオーバーヘッドは発生しません。効果的なテンプレート・ライブラリーを作成するのは複雑な作業ですが、使用するのはそれほど難しくありません。TBB はテンプレート・ライブラリーなのです。

TBB やその他のテンプレート・ライブラリーを使用するには、それらを関数呼び出しのコレクションとして扱うことができます。テンプレートは C 言語の一部ではないので、C++ コンパイラーでコンパイルする必要があります。最新の C++ コンパイラーは、テンプレートの多用によって生じる抽象化のペナルティーを最小限に抑えるように調整されています。

STL

C++ 標準テンプレート・ライブラリー（STL）は、C++ プログラミング言語向けのソフトウェア・ライブラリーであり、C++ プログラミング標準の一部です。すべての C++ コンパイラーは STL をサポートする必要があります。

「**ジェネリック・プログラミング**」の概念は、テンプレートに大きく依存する STL に深く根付いています。ソートなどの STL アルゴリズムは、データ型（コンテナー）には依存しません。STL は、コンテナーや連想配列など複雑なデータ型をサポートします。これらのデータ型は、コピーや割り当てなどの基本的な操作をサポートしていれば、任意のビルトイン型またはユーザー定義型に基づくことができます。

実行ポリシー

C++17 標準では、`seq`、`par`、`par_unseq`、`unseq` などの実行ポリシーを受け入れるため、ほとんどのアルゴリズムにオーバーロードが導入されました。これらのオーバーロードをサポートする C++ STL の実装は、C++17 の一部のみをサポートしているにもかかわらず、Parallel STL（PSTL）ライブラリーと呼ばれることがあります。C++ 標準はホスト上での実行用に設計されていますが、標準ライブラリー実装では拡張機能として追加の実行ポリシーを定義できます。ライブラリーがアプリケーションにリンクされているときに、計算を特定のアクセラレーターにオフロードする PSTL 実装もいくつか登場しています。最近では、一部の PSTL 実装に管理サポートが追加され、プログラムが特定のデバイスにワークを送信したり、少なくとも実装によって計算を最も適切に実行できる場所に選択的に配置できるようになりました。しばらくの間、PSTL 実装は TBB に同梱されていましたが、この取り組みは、現在では oneDPL（oneAPI DPC++ ライブラリーの略）と呼ばれる独自のプロジェクトに引き継がれており、TBB プロジェクトではなくなりました。oneDPL は、ホスト（CPU）上のスケジュールを管理するため実装で TBB を使用しています。

オーバーロード

オペレーターのオーバーロードは、ビルトイン型（例: `int`）が受け入れるコンテキストで新しいデータ型（例: `complex`）を使用できるようにするオブジェクト指向の概念です。これは、関数や = または + などのオペレーターへの引数として使用できます。C++ テンプレートは、さまざまなパラメーターや戻り値の組み合わせを持つ関数名にオーバーロードを拡張する一般化されたオーバーロードを提供します。テンプレートを使用した汎用プログラミングとオーバーロードを使用したオブジェクト指向プログラミングの目標は、*ポリモーフィズム（多相性）*、つまりデータのタイプに基づいてデータを異なる方法で処理しながら、コード処理を再利用する機能です。TBB はこれをうまく行うため、TBB テンプレート・ライブラリーのユーザーとしてそれを活用できます。

レンジ（範囲）とイテレーター（反復子）

C++ の専門家によると、レンジはアルゴリズムとイテレーターの強力な一般化および拡張です。C++ レンジ・ライブラリーは、積極的なレンジ・アルゴリズムと遅延レンジアダプターの両方を提供します。アダプターをパイプで接続して、ビューが反復処理されるときに遅延実行される強力なアクションを作成できます。

ユーザーにとって、イテレーターやレンジの背後にある概念はほとんど同じです。つまり、セットを示すショートカット（およびそれを走査する方法に関するヒント）です。0 から 999999 までの数字を表す場合、この間隔を数学的に `[0,999999]` または `[0,1000000)` と表記できます。数学表記法では、角括弧 [ ] は包含的であり、丸括弧 ( ) は包含的でないことに注意してください。TBB 構文を使用して、`blocked_range<size_t>(0,1000000)` と記述します。

レンジは、強制的な並列処理ではなく「可能な並列処理」を指定したい範囲を示します。100 万回反復するように計画された“`parallel for`”を考えてみます。直ちに 100 万のスレッドを作成してワークを並列に実行することも、`[0,1000000)` のレンジで 1 つのスレッドを作成することもできます。利用可能な並列性を満たすため必要に応じて細分化できることがレンジの好まれる理由です。

TBB はイテレーターとレンジをサポートし使用することで、C++ のレンジの概念を拡張して、分割（再帰レンジ）を含む独自の TBB レンジの概念を導入します。本書では、イテレーター、C++ のレンジ、TBB のレンジについて随所で説明しています。2 章以降には、このような例が多数あります。使い方の例も紹介しているので、簡単に再現できると思います。何を、どこで、どのように使用するか簡単に示します。 C++ におけるイテレーターとレンジの深い意味を理解しても、TBB をうまく使用できるわけではありません。現時点での簡単な説明は、イテレーターはレンジよりも抽象度が低いため、少なくとも、2 つのパラメーター（`something.begin()` と `something.end()`）を持つイテレーターを使用するコードが多くなりますが、伝えたいことは**「レンジ（`begin` から `end`）を使用してください」**ということだけです。再帰レンジ（TBB）を使用すると、ワークを分割する際に（レンジを分割する必要がある）、さらに進んで並列にワークを実行できるようになります。

まとめ

私たちは、分解、スケーリング、正当性、抽象化、パターンという視点から、「並列に考える」方法を探ってきました。また、すべての並列プログラミングの取り組みにおいて重要な懸念事項として局所性を上げました。さらに、TBB でサポートされるように、スレッドの代わりにタスクに注目することが並列プログラミングにおける革新的な進歩を表す方法であることを説明しました。 そして、TBB を効果的に活用するため、基本的な C の知識を超えた C++ プログラミングの重要な要素についても説明しました。

これらの重要な概念を念頭に置くことで、皆さんは「並列に考える」ことができるようになります。プログラマーの皆さんは、プログラミングに役立つ並列処理に関する直感を身に付けるでしょう。

本書を読み進めて、TBB を使用した並列プログラミングをさらに探求し学習していく上で、この序文と用語集は非常に貴重なリソースとなります。

関連情報

C++ 標準: https://isocpp.org/std/the-standard

オンライン・リファレンス形式の C++ 標準 – 強くお勧めします: https://cppreference.com

“The Problem with Threads” by Edward Lee, 2006. *IEEE Computer Magazine*, May 2006, https://tinyurl.com/ddr7thu3, or U. C. Berkeley Technical Report: www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

本書で使用されているすべてのコード例は、https://tinyurl.com/tbbBOOKexamples で入手できます。

1 章 “Hello, oneTBB!” のはじめ

この章では、oneAPI スレッディング・ビルディング・ブロック（oneTBB）ライブラリーの背景を探り、その主要コンポーネントの概要を示し、ライブラリーの入手方法を説明して、一連の簡単な例を紹介します。

今日の TBB は、標準 C++ の並列処理を最大限に活用する最良の方法の 1 つです。TBB プロジェクトは、C++ 標準への取り組み、タスク（スレッドではなく）によるプログラミングの利点の重視、優れた実装のタスク・スチール・スケジューラーの提供により、数十年にわたってその地位を維持してきました。それらの特徴については「序文」で詳しく説明しました。この章では、TBB 自体の解説に移ります。

スレッディング・ビルディング・ブロックがユニークであり続けるのは、いくつかの重要な決定事項に基づいているためです:

- 既存のコンパイラーで一般的な C++ プログラムをサポートします。
- 緩和されたシーケンシャル実行。
- 再帰的な並列処理と汎用アルゴリズムを使用します。
- タスクスチールを使用します。

TBB プロジェクトが 2006 年に開始されて以来、C++ の状況は進化しており、並列処理はこれまで以上に重要になっています。今日の TBB は、最新の C++ 標準と競合することなく、それに基づいて構築されており、シームレスな統合と強化された機能を保証します。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_1

```
1.   int main(int argc, char **argv)
2.   {
3.   auto values = std::vector<unsigned>(HOWMANY);
4.
5.   bbpHexPi bbp;
6.
7.   tbb::parallel_for(tbb::blocked_range<int>(0,values.size()),
8.                [&](tbb::blocked_range<int> r) {
9.                    for (int i=r.begin(); i<r.end(); ++i) {
10.                       values[i] = bbp.EightHexPiDigits(i*8);
11.                   }
12.               });
13.
14.  for (unsigned  eightdigits : values)
15.        printf("%.8x", eightdigits);
16.
17.  printf("\n");
18.
19.  return 0;
20.}
```

図 1-1. Pi（16 進数）の数値を並列に設定します。サンプルコード intro/intro_pi.cpp

Hello, oneTBB: π はいかが?

私たちはみんなコードを見ることが大好きなので、いろいろ説明する前に、TBB を使用した簡単な例から始めましょう。

図 1-1 では、並列処理を使用して配列に π（pi）の数値を設定します。この並列処理を適切にスケールするには、TBB が提供する高度なタスクスチール機能が必要です。

この章では、プログラマーとしての私たちの仕事は TBB に並列性を明示することであると繰り返し強調します。7 行目の `tbb::parallel_for` は、ワークを多数のタスクで表現することで、これを実現しています。8 行目から 11 行目では、タスクがループとして指定されています。このループは、TBB ランタイムが、完了すべきワークの全体範囲の中から、そのタスクに割り当てた要素（範囲）をすべて埋めるように動作します。1 つの例で TBB のユニークな特徴をすべて紹介できませんが、TBB は不規則な並列処理に優れているため、この例では他の並列処理方式よりもはるかにスケールします。実際には、不規則な並列性は一見しただけでは分からないかもしれません。これを説明するためこの単純な例を作成しましたが、不規則な並列処理が一般的であることを忘れないでください。不規則な並列処理におけるタスクスチールによる優位性が、TBB が広く受け入れられている主な要因です。

図 1-1 のプログラムでは、この無理数の位置 n から始まる 8 桁の円周率（π）を返す関数 `EightHexPiDigits(n)` が利用できることを前提としています。この実装では、先頭の “3” は位置 0（`n=0`）にあるので、値 *`n>0`* は小数点の右の n 番目の位置（1 番目、2 番目など）を表します。16 進数の円周率は、`3.243F6A8885A308D313198A2E03707344` で始まるため、`EightHexPiDigits(0)` は `0x3243F6A8` を、`EightHexPiDigit(8)` は `0x885A308D` を返します。

興味深いことに、`EightHexPiDigits(n)` を、適度なメモリー消費および線形時間で完了するように記述することができます。つまり、円周率の n 番目の桁を計算する場合、それより前の桁をすべて計算する必要はありません。一部の数字は他よりも速く計算されるため、並列処理は不規則であることに注意してください。
最も小さなタスクは、1 回の `EightHexPiDigits(n)` 呼び出しで、1 つのベクトル要素に 8 桁の値を入力します。数十個のコアを持つマルチコアシステムで何千ものベクトル要素を入力する場合、各プロセッサー・コア対し正確に `elements ÷ cores` 個のタスクを割り当てたくなるかもしれませんが、一部の要素の計算が遅くなるため、それは誤りです。TBB を使用してコードを記述すると、ランタイムは、キャッシュの再利用とスケーリングの両方を考慮して負荷分散を行う、非常に効率良いタスク・スチール・スケジューラーを利用できます。

TBB のランタイムは、適切なスレッド数を自動的に決定するので、通常はそのままにしておきます。ただし、割り当てを制限する方法はいくつかあります（1 つはローカル制限を設定し、もう 1 つはプログラム全体/グローバル制限を設定します）。TBB を 1 スレッド、2 スレッドなどに制限してパフォーマンスをテストすることもできます。ハードウェアに 224 個の論理スレッドがあるシステムで、1 ～ 448 個のスレッドでこれを検証しました。結果を図 1-2 に示します。これは、プログラムの実際のスケーリングを調査する方法です。このプログラムはマシンの限界まで適切にスケールしていることが分かります。グラフ（図 1-2）には、達成できるスケーリングを制限するハードウェアの要因が記載されています。TBB がハードウェアへ適切にマッピングし、各マシンを手動でマッチングさせる必要もなく、利用可能な並列処理を明らかにするようにプログラムするだけで済むのは幸いなことです。グラフを見ると、スレッド数が 56 を超えると複数ソケットが制限され、非均一メモリーアクセス（NUMA）効果（今は理解する必要はありません）によりスケーリングが低下し、スレッド数が 112 を超えると、コア全体を使用するほどスケーラブルではないハイパースレッドの使用が制限され、スレッド数が 224 を超えるとマシンサイズの上限に達し、実際のハードウェア並列処理の限界に達していることが分かります。224 スレッド以降は、TBB ランタイムが、メリットのないオーバーヘッドによってパフォーマンスを低下させることなく、計算をマシンの能力に自動的に一致させるためパフォーマンスは安定します。これは、TBB が効果的であることを示す一例にすぎませんが、TBB が私たちにとって最善のことをしてくれる信頼性を示しているため、非常に重要です。

この章の GitHub リポジトリーには完全なコードが含まれています。これには、図 1-2 のスケーリング・グラフを作成するためだけに、`tbb::global_control` クラスを使用して `tbb::global_control::max_allowed_parallelism` を設定するのも含まれます。

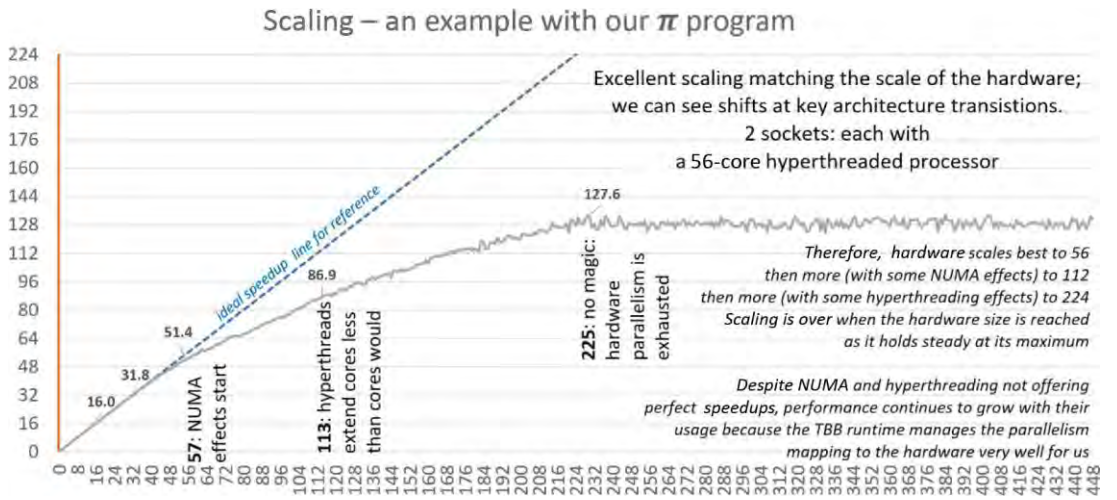


図 1-2. 56 コアのハイパースレッド・プロセッサーのペアで実行 – いくつかの変曲点に注目

これくらいで、私たちの興味をそそるには十分でしょう。これは、これから起こる多くのことのヒントに過ぎません。この章の残りを学ぶため、TBB について知っておくべき重要な情報を詳しく見ていきましょう。これには、すべてのコード例 (`EightHexPiDigits(n)` の実装を含む) が GitHub リポジトリーで入手できることも含まれます。

図 1-2 に使用されたシステムは、インテル Core i5-13500 プロセッサーを 2 つ搭載したデュアルソケット、64GB のメインメモリーを備えたシステムです。各プロセッサー (ソケット) には 14 個のコアがあり、各コアはハイパースレッドが有効にされており (プロセッサーごとに 28 個のハードウェア・スレッド)、プロセッサーは 2 つあるため合計 56 個のハードウェア・スレッドを実行できます。このコードをさまざまなマシンで実行すると、物理コア数、論理コア数、またはソケット数を使い果たしたときのハードウェアの制限を反映する同様の曲線が得られることが予想されます。これはプログラマーとって素晴らしいニュースです: TBB は実行時にアルゴリズムをハードウェアにマッピングするように設計されているため、実行時に並列処理数を決定する必要がなく、潜在的な並列処理の特定に集中できます。

時代を超えて: TBB と oneTBB - TBB に変わりない

「スレッディング・ビルディング・ブロック」プロジェクトは、2006 年に TBB という名称で始まり、2020 年後半に oneAPI スレッディング・ビルディング・ブロック（oneTBB）に名称が変更されました。ライブラリー内の機能では、常に“tbb”という名称が使用されており、現在も継続されています。oneTBB の背景にある技術的な変更は、“最新の C++”を採用することによって動機付けられました。これにより、オリジナルの TBB と比較して、oneTBB の多くの機能が簡素化されました。これは、TBB が登場した頃は C++ が移植可能な並列処理をサポートする前であったため、オリジナルの TBB にはこれを補う複雑性がありました。最新の C++（C++11 以降）には、ロックやアトミック操作など移植可能な並列処理のサポートが含まれています。TBB が最新の C++ 仕様を完全に網羅するように簡素化されたため、名称が oneTBB に変更されました。なぜ TBB v2.0 と名付けなかったのでしょうか？ それは、TBB の名称変更が、“oneAPI”と呼ばれるオープン・アクセラレーターのサポートを求める大きな動きに巻き込まれたためです。これにより、TBB v2.0 ではなく oneTBB という名前が推奨されました。ビリー・ジョエルが「It’s Still Rock and Roll to Me（ロックンロールが最高さ）♫」と歌ったことになぞらえて、私たちは「It’s Still TBB to Me（TBB が最高さ）」と言えるでしょう。

以降 oneTBB と TBB は同義語として考えます。記述する際には、実装全体やライブラリー名を指す場合は“oneTBB”を使用します。それ以外の場合は、単純にスレッディング・ビルディング・ブロック（TBB）とします。すべてのプログラム・インターフェイスは、これまでどおり名前空間 `tbb::` にあります。oneTBB で変わったのは、2006 年のオリジナル TBB プロジェクトでは利用できなかった最新の C++ のインターフェイスに依存するようになったことです。

スレッディング・ビルディング・ブロックが役立つ場所

並列性があらゆる場所に存在する世界では、TBB がどこに位置付けられるかを問うのは理にかなっています。

TBB は、アクセラレーターの制限に合わせることなく、並列性にアプローチしやすいプログラミング・モデルを提供します。これにより、CPU が提供する最大限の並列処理が可能になります。これは、2 つのことから重要です: 第 1 に、システムの全パワーを使用する必要があり、これは CPU の並列処理を適切に使用することから始まります。第 2 に、新しい技術を自由に探求できます（おそらく、自由な探求により、最初に発見された技術向けに設計されたドメイン固有のハードウェアに制約されません）。

「すべてはアムダールの法則による」という点で明らかなことから、TBB の重要性を示す次の 4 つの主な項目が挙げられます:

- TBB は最新仕様の C++ とともに、最新のコンピューターの CPU に備わる並列処理の可能性を最大限に引き出します。
  注: アクセラレーターは現代のコンピューターの並列処理に不可欠ですが、TBB ではその分野のプログラミングを他のモデル（CUDA、SYCL、OpenCL など）に任せます。後述しますが、アクセラレーターを適切に使用するには、システムの潜在能力を最大限に引き出すため、適切に構成された並列 CPU プログラムが重要になります。
- TBB は、ほぼ無制限の柔軟性を可能にする高度な機能を備えています。シンプルで使いやすい機能を維持したまま、最も要件の厳しい新しい並列アルゴリズムも実装できます。
- TBB は並列処理を教えたり学んだりするのに非常に役立ちます。
- TBB は、この分野の専門家や新規ユーザーにも並列処理に最適なツールです。

アムダールの法則は TBB が重要である理由を示す

TBB は、並列コンピューティングにおいて最も重要な汎用タスク・ライブラリーです。高度にスケーラブルなアプリケーションを実現する高品質なオープンソース実装を提供します。スケーリングを妨げるのはシリアル化（並列化されていないもの）です。これはアムダールの法則の本質です。

アクセラレーターを使用するしないにかかわりなく、コンピューター・システムのパフォーマンスを最大化するため理解し、対処する必要がある最も重要なことは、アムダールの法則です。本書の「序文」では、この概念について復習を提供しました。これは、この概念についておさらいし、パフォーマンス向上につながる具体的な行動へとどう結びつけるかについて、補足的な解説を加えるためです。

近年、計算集約型のコード（AI、視覚効果、科学、エンジニアリング、可視化）ではアクセラレーターが活用されています。当然ですが、これらのコードに関する議論は、コアとなるアルゴリズムのパフォーマンスを最大化することに焦点が当てられています。しかし、アムダールの法則は、TBB が他の方法よりも優れた形で独自にうまく対処できる問題にも目を向けさせることになります。

この章では、TBB の概要を示し、アプリケーションでそのメリットを最大化する構成の戦略について説明します。

2 つの真実 – どちらも TBB を使用するように言っています

アプリケーションで並列処理を効果的に活用する取り組みの中で、次の 2 つの真実が明らかになりました:

- タスクレベルのプログラミング: アプリケーション開発では、“タスク” レベルに重点を置く必要があります。プログラマーは並列処理の可能性を特定することに集中することで、TBB などの抽象化にアクセスできるようになります。これらの可能性を基盤となるハードウェアに効率的にマッピングするのは、この抽象化の役割です。このコンセプトには豊富な情報が含まれており、さらに詳しく知りたい方のために、序文に「スレッドではなくタスクを使用したプログラム」という節を設けています。

- カスタム・ソリューションの使用を避ける: スレッドプールなど独自のソリューションを作成することは、一般的にお勧めできません。見た目以上に複雑なだけでなく、これらの課題に対処する TBB のようなソリューションがすでに存在しています。さらに、今後ハードウェアが進化しても、TBB のような共通ライブラリーを利用することで、アプリケーションがライブラリーの API によって提供される抽象化に準拠している限り、アプリケーション自体を変更することなく適応できます。これもまた、検討に値する重要な意味を持っています。

スレッディング・ビルディング・ブロック（TBB）ライブラリー

スレッディング・ビルディング・ブロック（TBB）ライブラリーは、次の 2 つの役割を果たす C++ ライブラリーです:

（1）C++ 標準が十分に拡張されていない（場合によっては拡張されるべきではない）並列処理のサポートや、すべてのコンパイラーが新機能を完全にサポートしていない並列処理のサポートにおいてその空白を埋める、

（2）C++ 言語標準に含まれる可能性のある範囲を超える、並列処理の高レベルの抽象化を提供します。TBB には、図 1-3 に示すように、さまざまな機能が含まれています。

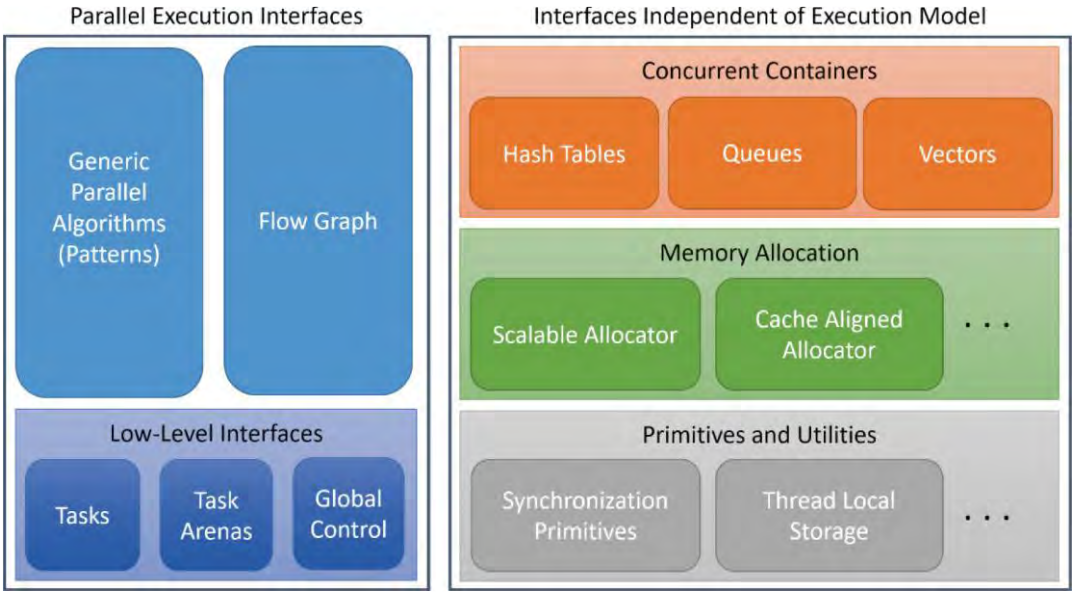


図 1-3. TBB ライブラリーの機能

これらの機能は、並列計算を表現するインターフェイスと、実行モデルに依存しないインターフェイスの 2 つのグループに分類できます。

並列実行インターフェイス

TBB を使用して並列プログラムを作成する場合、高レベル・インターフェイスを使用するか、タスクグループを使用してタスクで直接、アプリケーション内の並列性を表現します。タスクについてはこの章の後半で詳しく説明しますが、ここでは、TBB タスクは小さな計算とその関連データを定義する軽量オブジェクトと考えることができます。TBB 開発者として、私たちは、直接的に、あるいはパッケージ化された TBB アルゴリズムを介して間接的に、タスクを使用してアプリケーションを表現し、ライブラリーがこれらのタスクをプラットフォームのハードウェア・リソースにスケジューリングします。

TBB は多くの並列パターンに対して高レベルのインターフェイスを提供しますが、それでもそれらがいずれも問題に一致しない場合があります。その場合、タスクグループなどの他の TBB 機能を使用して、独自のアルゴリズムを直接構築できます。

TBB 並列実行インターフェイスの真の能力は、それらを混在させる、つまり構成の可能性にあります。トップレベルにフローグラフを持ち、そのノードがネストされた汎用並列アルゴリズムを使用するようなアプリケーションを作成できます。さらに、そのネストされたアルゴリズムの内部に、追加の汎用並列アルゴリズムをさらにネストさせることも可能です。

TBB を構成可能にする重要な特性の 1 つは、*緩和された順次セマンティクス*をサポートしていることです。緩和された順次セマンティクスとは、TBB タスクによって表現する並列性は、実際にはライブラリーへのヒントに過ぎず、タスクが実際に互いに並列実行される保証がないことを意味します。これにより、TBB ライブラリーはパフォーマンス向上の必要に応じてタスクをスケジュールできる柔軟性が大幅に高まります。この柔軟性から、ライブラリーは、システムのコア数が 1 個、8 個、または 80 個であっても、スケーラブルなパフォーマンスを提供できます。また、ライブラリーはプラットフォーム上の動的な負荷に適応できるようになります。

実行モデルに依存しないインターフェイス

並列実行インターフェイスとは異なり、図 1-3 の 2 番目に大きな機能グループは、実行モデルおよび TBB タスクから完全に独立しています。これらの機能は、TBB タスクを使用するアプリケーションと同様に、`std::jthread`、`pthreads`、`WinThreads` などのネイティブ・スレッドを採用するアプリケーションに役立ちます。

TBB は、ハッシュテーブル、キュー、ベクトルなど一般的なデータ構造へのスレッド対応のインターフェイスを提供するコンカレント・コンテナーをサポートします。TBB スケーラブル・メモリー・アロケーターやキャッシュ・アライン・アロケーターなどのメモリー割り当て機能もあります。また、同期プリミティブやスレッド・ローカル・ストレージ（TLS）など低レベルの機能もあります。

開発者は、TBB からアプリケーションに役立つ機能を選択できます。例えば、スケーラブルなメモリー・アロケーターのみを使用し、他は使用しないこともできます。もしくは、コンカレント・コンテナーといくつかの汎用並列アルゴリズムを使用することもできます。もちろん、3 つの高レベル実行インターフェイスをすべて組み合わせて、TBB のスケーラブル・メモリー・アロケーターとコンカレント・コンテナー、およびライブラリー内の機能を利用するアプリケーションをビルドすることもできます。

スレッディング・ビルディング・ブロック（TBB）ライブラリーの入手

本書の執筆時点では、ライブラリーは https://github.com/uxlfoundation/oneTBB から、oneAPI ツールキットの一部として、またはスタンドアロン・バージョンとして入手できます: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html。

TBB を入手する最も適切なルートを選択し、各サイトに記載されているパッケージのインストール手順に従うことは、読者に委ねます。

例のコピーを入手

この章で使用されているすべてのコード例は、https://tinyurl.com/tbbBOOKexamples で入手できます。リポジトリーには、図番号と特定の例（ディレクトリーとファイル名）の関係を示すデコーダーがメイン・ディレクトリーに公開されています。

最初の “Hello, TBB!” の記述例

図 1-4 は、`tbb::parallel_invoke` を使用して 2 つの関数を評価する小さな例を示しています。1 つは “Hello” を出力し、もう 1 つは “TBB!” を並列に出力します。この例は単純なものであり、並列化のメリットはありませんが、TBB を使用する環境が適切に設定されていることを確認するのに使用できます。図 1-4 では、TBB 関数とクラスにアクセスするため、`tbb.h` ヘッダーをインクルードしています。これらはすべて `tbb` 名前空間にあります。`parallel_invoke` の呼び出しは、渡された 2 つの関数が互いに独立しており、異なるコアまたはスレッドで任意の順序で並列実行しても安全であることを TBB ライブラリーにアサートします。この制約の下では、結果の出力には `Hello` または `TBB!` のどちらも最初に表示される可能性があります。

```
#include <iostream>
#include <tbb/tbb.h>

int main() {
    tbb::parallel_invoke(
        []() { std::cout << " Hello " << std::endl; },
        []() { std::cout << " TBB! " << std::endl; }
    );
    return 0;
}
```

図 1-4. “Hello, TBB!” の例。サンプルコード: `intro/intro_helloTBB.cpp`

図 1-4 では、*C++ ラムダ式*を使用して関数を指定しています。ラムダ式は、TBB などのライブラリーを使用して、タスクとして実行するユーザーコードを指定する場合、特に便利です。C++ ラムダ式を説明するため、序文では、「C++ の機能紹介（TBB に必要な）」節の「ラムダ関数」でこの重要なモダン C++ 機能について概説しています。

TBB.H を使用するかどうか

1 つのヘッダーファイル **tbb.h** に、TBB のすべての定義が含まれます。これは、特に TBB を学習するときに便利です。必要以上に定義するのは好ましくないと考える人が多いため、TBB では特定の機能ごとにヘッダーファイルを個別に提供しています。すべてが名前空間 `tbb::`（別名の名前空間 `oneapi::tbb::`）で定義されているため、多くの開発者は `tbb.h` だけを使用しています。多くの例では `tbb.h` を使用しますが、一部の例では選択的に特定のヘッダーファイルのみを使用します。これは、両方の特徴を紹介するためであり、アプリケーションで何が最適かは、自身で判断してください。

Today's TBB（現在の TBB）

TBB は 2006 年以来、重要な存在となっていますが、C++ の世界では多くの変化があり、それが oneTBB への移行に反映されています。

2006 年当時、C++ には並列プログラミングをサポートする言語機能がなく、標準テンプレート・ライブラリー（STL）を含む多くのライブラリーがスレッドセーフではなかったため、並列プログラムで簡単に使用できませんでした。オリジナルの TBB バージョンでは、C++ 言語のサポート不足に対処する必要がありました。

TBB を最初に導入したとき、私たちは並列処理に必要な基本的なサポートはすべて C++ 言語自体に含まれていることが望ましい、と考えました。これにより、TBB は堅牢な基盤を利用して、より高レベルの並列処理の抽象化を構築できるようになります。

オリジナルの TBB と C++ が 2006 年にうまく統合されたのと同じように、今日の C++ と TBB もうまく統合されています。

2006 年に TBB が登場したとき、マルチコア・プロセッサーは新しいものでした。それはまさに理想のマリアージュであり、TBB はすぐに多くの人に受け入れられました。TBB は、スレッドではなくタスクの観点からのプログラミングを可能にし、高いスケーラビリティーと構成の容易性を備え、C++ と連携することで、マルチコアシステムで並列処理を調整する新しい標準を確立するのに役立ちました。

微妙ですが重要な点として、TBB では、プログラマーがタスクまたはタスクを生成する高レベルのアルゴリズムに基づいて記述し、ランタイムがタスクをスレッドにマッピングするモデルを重視しています。これは並列プログラミングにおいて非常に重要であることが証明されています。以前は重要とされていたスレッドを直接プログラミングすることに重点が置かれる以前の問題点は、バークレー大学の Edward A. Lee 教授による古典的な論文 “The Problem with Threads” に詳しく記載されています。

図 1-5 は、oneTBB とオリジナルの TBB における現代化の詳細を示しています。数年経た現在でも、TBB は経験に基づいてオリジナルの TBB から進化し、最新の C++ に適合しているため、依然として高い妥当性を保っています。

2006 年	今日
マルチコアは稀: マルチコア・プロセッサーは新しいものであり、プロセッサー市場のほんの一部でした。	**シングルコアは稀:**マルチコア・プロセッサーは一般的なものとなりました。「メニーコア」という用語は、「マルチコア」が 4 コアか 8 コアで成長が止まるかどうかが議論されていたときに一時的に登場しました。現在、マルチコア・プロセッサーのコア数は急増しており、場合によっては「メニーコア」ですら想像もしなかったレベルにまで達することもあります。
C++ 標準はシングルスレッドのみ: C++ では、言語や標準ライブラリーでスレッドがサポートされていませんでした。	**最新の C++ 標準のマルチスレッド・サポート:** C++ には、移植可能なアトミックとロックを含むスレッドのサポートが標準化されています。
スレッドセーフは非標準: ほとんどのライブラリーはスレッドセーフではなく、スレッドセーフという概念自体がほとんどのプログラマーには関心事ではありませんでした。	**スレッドセーフは標準:** ライブラリーは、ほとんどスレッドセーフです。そうでない場合、マルチスレッド環境で使用するソリューションが存在します。
並列プログラミングはニッチ: 並列プログラミングは、ほとんどのプログラマーがあまり知らないニッチな専門分野でした。並列プログラミングはスレッドレベルで行われることが多く、その問題については前述の「スレッドの問題」で詳しく説明されています。	**並列プログラミングはメインストリーム:** 並列プログラミングは、さまざまな形で主流になっています。
クロックレートの上昇により加速:アプリケーションは年々加速し、MHz の波に乗って数 GHz まで達しましたが、それは終わりました。 この危機は、画期的な論文「無料ランチは終わった:ソフトウェアにおける並行処理への根本的な転換 」Herb Sutter 著、で告知されていました。	**並列処理により加速:** 今日のアプリケーションの加速は、マルチコア並列処理だけでなく、ドメイン固有のプロセッサー（GPU、TPU など）にも依存しています。この傾向と将来における重要性については、John L. Hennessy と David A. Paterson による重要な論文 “A New Golden Age for Computer Architecture” で詳しく説明されています。私たちの「加速されたコンピューティング」の時代は、異種並列処理に基づいています。
TBB はスレッドとタスクの移植性を解決する必要がありました: TBB は、そのようなサポートを提供していなかった C++ 標準をベースにタスクとスレッドのサポートを提供しました。	**TBB は移植性のあるタスクに重点を置いています:** TBB は、ポータブルなスレッドのサポートと、ラムダを提供する最新の C++ をベースにしたタスクサポートを提供します。

図 1-5. 当時（2006 年）と現在：TBB はこれまで以上に便利に

TBB と C++ の並列処理サポートは進化を継続中

C++ 言語委員会は、言語とそれに付随する標準テンプレート・ライブラリー（STL）にスレッド機能を追加することに取り組んでいます。図 1-6 は、並列処理と並行処理に対応する新しい C++ の機能と、今後計画されている機能を示しています。

ISO C++ 標準	この標準で導入された機能の一部
C++11/14	標準化されたメモリーモデル、`std::unique_ptr`、`std::shared_ptr`、`std::async`、`std::future`、`std::thread`、`std::atomic`、`std::mutex`, `std::lock_guard`、`std::unique_lock`、`std::conditional_variable`
C++17	実行ポリシーの実行（`std::seq`, `std::par`, `std::par_unseq`, `std::unseq`）、実行ポリシーの実行を受け取るアルゴリズムのオーバーロード
C++20/23	コルーチン（`co_await`、`co_yield`、`co_return`）、`std::jthread`、`std::stop_token`、`std::atomic_ref`、`std::counting_semaphore`、`std::binary_semaphore`、`std::latch`、`std::barrier`、`std::atomic_ref`
C++26（予定）	実行ライブラリー（`sender`、`receiver`、`scheduler`）、`std::simd`

図 1-6. C++ 標準の機能と提案されている機能

C++11

図 1-6 に示すように、C++11 標準は、スレッド化のため標準化されたメモリーモデル、`std::async`、`std::future`、`std::thread` などを含む低レベルの基本的な構成要素を導入することで、大きく前進しました。また、アトミック変数、排他オブジェクト、条件変数も導入されました。これらの拡張では、プログラマーが高レベルの抽象化を導入するため大量のコーディングを行う必要がありますが、以降は基本的な並列処理を C++ で直接表現できるようになります。C++11 標準ではスレッド化の機能は明らかに改善されましたが、移植可能で効率良い並列コードを簡単に記述できる高レベルの機能はまだ提供されていません。また、タスクや、ワークスチール・タスク・スケジューラーもありません。

C++17

C++17 標準では、低レベルの構成要素よりも抽象化のレベルを上げる機能が導入され、低レベルの詳細を気にすることなく並列処理を表現しやすくなりました。この章の後半で説明するように、まだいくつかの重大な制限があり、表現力やパフォーマンスが十分ではありません。C++ 標準にはまだやるべきことがたくさんあります。

C++17 で追加された機能の中で最も関連性の高いのは、標準テンプレート・ライブラリー（STL）アルゴリズムで使用できる*実行ポリシー*です。このポリシーにより、アルゴリズムを安全に並列化、ベクトル化、または並列化とベクトル化できるか、あるいは元の順序付けされたセマンティクスを保持する必要があるかどうかを選択できます。これらのポリシーをサポートする STL 実装を Parallel STL（PSTL）と呼ぶこともあります。PSTL のオープンソース実装は、DPL プロジェクトの 1 つです（https://tinyurl.com/uxlonedpl を参照）。この章の後半では PSTL を使用して、最新の C++ で使用される TBB の能力を説明します。

今日の Parallel STL の現実

C++17 標準では Parallel STL が定義されていますが、それでもまだ標準ライブラリー実装によるサポートが必要です。PSTL のオープンソース実装である oneDPL プロジェクト（https://tinyurl.com/uxlonedpl を参照）は、CPU の優れた並列処理のサポートにより、TBB を扱う多くの人が利用しています。C++17 の初期には、新しい API への移行の利便性のため、一部のインテルのディストリビューションで TBB に PSTL の実装がタグ付けされていました（ただし、現在はバンドルされていません）。その実装は oneDPL プロジェクト（https://tinyurl.com/uxlonedpl を参照）に進化し、TBB を介して CPU 並列処理の優れたサポートを現在でも提供しており、TBB の利用者にとっては便利です。アクセラレーターのソフトウェア・エコシステムには、独自のソリューションが存在することがあります（例として、oneDPL は PSTL アルゴリズムの SYCL ベースの実装も提供しています）。特定のベンダーのライブラリーを使用する場合、すべてをそのベンダーのアクセラレーターにオフロードするのではなく、ワークを最適なリソースにインテリジェントに割り当てるには、注意が必要になることがあります。実装が成熟するにつれて、この種のチューニングは進歩します。ここでは、C++17 実装の問題の微妙な違いを詳細に検討するつもりはありません。ここで強調しておきたいのは、特定のコンパイラーとライブラリーの組み合わせが PSTL に対してどのように影響するか理解するには、別途学ぶ必要があるということです。

C++20/C++23

C++20 および C++23 標準では、C++ 言語にコルーチンの基本的なサポートが導入されました。コルーチンは、一時停止して後で再開できる関数です（TBB で利用可能な同様の概念である再開可能タスクについては、6 章で紹介します）。C++ 標準テンプレート・ライブラリーには、コルーチンに対する重要なライブラリー・ サポートがまだなく、その実装は現在も進行中です。

`std::thread` に関する懸念に対処するため、C++20 で `std::jthread` が追加されました。これは、破棄時に自動的にジョインされ、キャンセルをサポートします（`std::stop_token` 経由）。この標準では、アトミックの改善や、非常に一般的な同期プリミティブであるラッチとバリアも導入されました。

C++26

本書の執筆時点では、C++26 標準では、データ並列型とそれらの型に対する操作が移植可能であることを表現する `std::simd` ライブラリーの追加が予定されています。また、標準実行ポリシーをサポートする新しい基礎的な線形代数アルゴリズムも備えています。新しい実行ライブラリーでは、送信者、受信者、スケジューラーに加えて、汎用実行リソースでの非同期実行を管理するフレームワークを定義する多数のフリー関数が導入されています。

ご覧のとおり、C++ は並列処理と並行処理のサポートにおいて、その水準を高め続けています。。しかし、C++ の並列処理サポートがこれだけ増強されても、TBB は標準 C++ を基盤とする貴重な機能であることに変わりはありません。

完全な例

この節では、図 1-3 に示す両方の高レベル実行インターフェイスを使用して、並列実行の恩恵を受けることができる大きな例を示します。アルゴリズムと機能の詳細をすべて説明する代わりに、この例を使用して TBB で表現できるさまざまな並列処理レイヤーを確認します。これは、数段落で説明できるほど単純でありながら、考えられるすべての並列性の階層を示すほど複雑です。ここで作成する最終的なマルチレベル並列バージョンは、最適な TBB アプリケーションの作成に関するガイドではなく、構文の例として見るべきです。以降の章では、この節で使用するすべての機能について詳しく説明し、現実的なアプリケーションで優れたパフォーマンスを得るためのガイダンスを提供します。

シリアル実装から始める

まず、図 1-7 に示すシリアル実装から始めましょう。この例では、画像のベクトル内の各画像にガンマ補正と色合いを適用し、それぞれの結果をファイルに書き込みます。強調表示された関数 `intro_gamma` には、各イメージに対して `applyGamma`、`applyTint`、および `writeImage` 関数を実行してベクトル要素を処理する `for` ループが含まれています。

これらの各関数のシリアル実装も図 1-7 にあります。画像表現の定義といくつかのヘルパー関数は intro_examples.h に含まれています。このヘッダーファイルは、例のすべてのソースコードとともに、https://tinyurl.com/tbbBOOKexamples で入手できます。

```
#include <iostream>
#include <vector>
#include <tbb/tbb.h>
#include "intro_examples.h"

using ImagePtr = std::shared_ptr<ch01::Image>;

ImagePtr applyGamma(ImagePtr image_ptr, double gamma);
ImagePtr applyTint(ImagePtr image_ptr, const double *tints);
void writeImage(ImagePtr image_ptr);

void myfuncG(const std::vector<ImagePtr>& image_vector) {
  const double tint_array[] = {0.75, 0, 0};
  for (ImagePtr img : image_vector) {
    img = applyGamma(img, 1.4);
    img = applyTint(img, tint_array);
    writeImage(img);
  }
}

ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel& p = in_rows[i][j];
      double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
      double res = pow(v, gamma);
      if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
      out_rows[i][j] = ch01::Image::Pixel(res, res, res);
    }
  }
  return output_image_ptr;
}
```

図 1-7. 画像ベクトルにガンマ補正と色合いを適用するシリアル実装の例。サンプルコード: intro/intro_gamma.cpp

```
ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
      ch01::IMAGE_WIDTH,
      ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  for ( int i = 0; i < height; ++i ) {
    for ( int j = 0; j < width; ++j ) {
      const ch01::Image::Pixel& p = in_rows[i][j];
      std::uint8_t b = (double)p.bgra[0] +
                  (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
       std::uint8_t g = (double)p.bgra[1] +
                  (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
     std::uint8_t r = (double)p.bgra[2] +
                 (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
     out_rows[i][j] =
       ch01::Image::Pixel(
         (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
         (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
         (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
       );
    }
  }
  return output_image_ptr;
}

void writeImage(ImagePtr image_ptr) {
  image_ptr->write( (image_ptr->name() + ".bmp").c_str());
}

int main(int argc, char* argv[]) {
  std::vector<ImagePtr> image_vector;

  for ( int i = 2000; i < 20000000; i *= 10 )
    image_vector.push_back(ch01::makeFractalImage(i));

  tbb::tick_count t0 = tbb::tick_count::now();
  myfuncG(image_vector);
  std::cout << "Time : " << (tbb::tick_count::now()-t0).seconds()
                  << " seconds" << std::endl;
  return 0;
}
```

図 1-7.（続き）

applyGamma 関数と applyTint 関数はどちらも、外側の for ループで画像の行を走査し、内側の for ループで各行の要素を走査します。新しいピクセル値が計算され、出力画像に割り当てられます。applyGamma 関数はガンマ補正を適用します。applyTint 関数は、画像に青い色合いを適用します。これらの関数は、メモリー管理を簡素化するため std::shared_ptr オブジェクトを受け取って返します。std::shared_ptr に馴染みのない読者は、「スマートポインターに関する注意事項」を参照してください。図 1-8 は、サンプルコードで入力された画像出力を示しています。

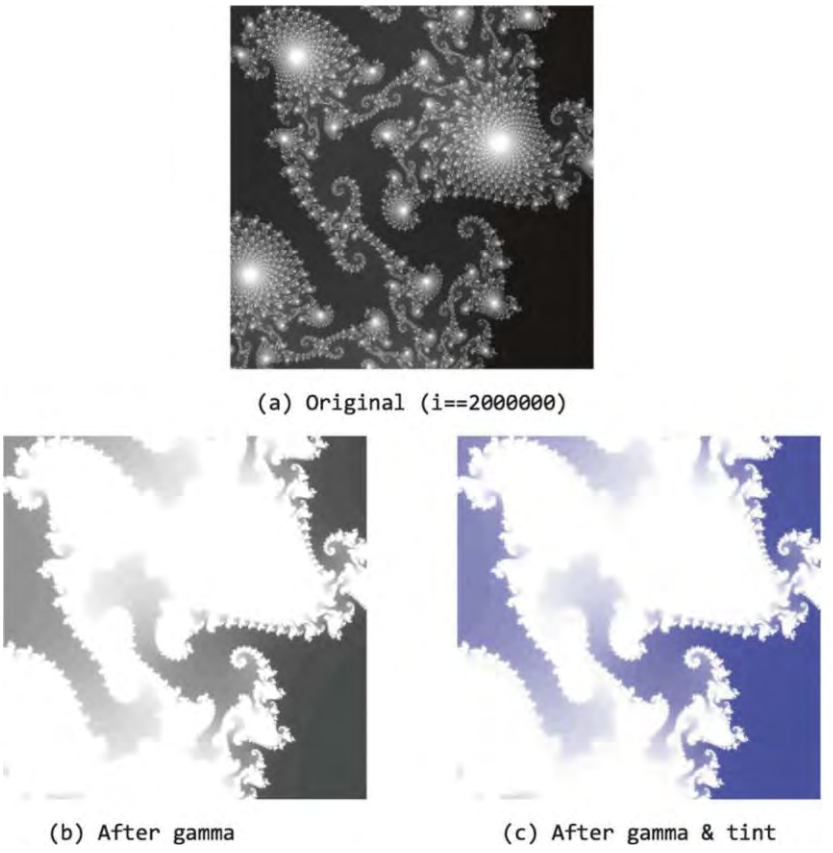


図 1-8. 図 1-7 の例の出力: (a) 元の生成画像、(b) ガンマ補正後の画像、(c) ガンマ補正と色付け後の画像

スマートポインターに関する注意事項

C/C++ でのプログラミングにおいて、最も難しいものの 1 つは、動的メモリー管理です。`new/delete` または `malloc/free` を使用する場合、メモリーリークや二重解放を避けるため、それらが一致していることを確認する必要があります。`unique_ptr`、`shared_ptr`、`weak_ptr` などのスマートポインターは、自動的に例外安全性に対応したメモリー管理を提供するため C++11 で導入されました。例えば、`make_shared` を使用してオブジェクトを割り当てると、オブジェクトへのスマートポインターを受け取ります。この共有ポインターを他の共有ポインターに割り当てると、C++ ライブラリーが参照カウントを処理します。スマートポインターを介したオブジェクトへの未処理の参照がない場合、オブジェクトは自動的に解放されます。図 1-7 を含むこの章のほとんどの例では、生のポインターの代わりにスマートポインターを使用します。スマートポインターを使用すると、`free` の挿入や削除が必要なすべてのポイントを確認する必要がなくなります。スマートポインターが適切な処理を行うことを信頼できます。

フローグラフを使用したメッセージ駆動型レイヤーの追加

トップダウンのアプローチを使用すると、図 1-7 の外側のループを、フィルターのセットを介して画像をストリーミングする TBB フローグラフに置き換えることができます（図 1-9 を参照）。この例は、最も作為的な選択であることは認めます。この場合、外側の並列ループを簡単に使用することも、`gamma` と `tint` ループのネストを結合することもできます。しかし、デモの目的で、TBB を使用してメッセージ駆動型の並列処理を表現する方法を示すため、これを個別のノードのグラフとして表現することを選択します。4 章では、TBB フローグラフ・インターフェイスについてさらに詳しく学び、この高レベルのメッセージ駆動型実行インターフェイスを使用する自然なアプリケーションについて説明します。

図 1-9. 4 つのノードを持つデータ・フローグラフ: (1) 画像を取得または生成するノード、(2) ガンマ補正を適用するノード、(3) 色合いを適用するノード、および (4) 結果画像を書き出すノード

図 1-9 のデータ・フローグラフを使用すると、パイプラインのさまざまなステージを異なる画像に適用して実行できます。例えば、最初の画像 `img0` が `gamma` ノードで完了すると、その結果は `tint` ノードに渡され、新しい画像 `img1` が `gamma` ノードに入ります。同様に、この次のステップが完了すると、`gamma` ノードと `tint` ノードの両方を通過した `img0` が書き込みノードに送信されます。一方、`img1` は `tint` ノードに送信され、新しい画像 `img2` が `gamma` ノードで処理を開始します。各ステップで、フィルターの実行は互いに独立しているため、これらの計算は異なるコアまたはスレッドに分散できます。図 1-10 は、TBB フローグラフとして表現されたループを示しています。

```
void myfuncFG(const std::vector<ImagePtr>& image_vector) {
  const double tint_array[] = {0.75, 0, 0};

  tbb::flow::graph g;
  int i = 0;
  tbb::flow::input_node<ImagePtr> src( g, [&]( tbb::flow_control &fc ) ->
ImagePtr
  {
    if ( i < image_vector.size() )
    {
      return image_vector[i++];
    }
    else
    {
      fc.stop(); return nullptr;
    }
  });

  tbb::flow::function_node<ImagePtr, ImagePtr> gamma(g,
    tbb::flow::unlimited,
    [] (ImagePtr img) -> ImagePtr {
      return applyGamma(img, 1.4);
    }
  );

  tbb::flow::function_node<ImagePtr, ImagePtr> tint(g,
    tbb::flow::unlimited,
    [tint_array] (ImagePtr img) -> ImagePtr {
      return applyTint(img, tint_array);
    }
  );

  tbb::flow::function_node<ImagePtr> write(g,
    tbb::flow::unlimited,
    [] (ImagePtr img) {
      writeImage(img);
    }
  );

  tbb::flow::make_edge(src, gamma);
  tbb::flow::make_edge(gamma, tint);
  tbb::flow::make_edge(tint, write);
  src.activate();
  g.wait_for_all();
}
```

図 1-10. 外側の `for` ループの代わりに TBB フローグラフを使用。サンプルコード: `intro/intro_flowgraph.cpp`

4 章で説明するように、TBB フローグラフをビルドして実行するにはいくつかの手順があります。最初に、グラフ・オブジェクト g が構築されます。次に、データ・フローグラフ内の計算ノードを構築します。残りのグラフに画像をストリーミングするノードは、src という名前の input_node です。計算は、gamma、tint、write という名前の function_node オブジェクトで実行されます。input_node は入力がなく、送信するデータがなくなるまでデータを送信し続けるノードと考えることができます。そして、function_node は、入力を受け取って出力を生成する関数のラッパーと考えることができます。ノードが作成されたら、エッジを使用してそれらを接続します。エッジは、ノード間の依存関係または通信チャネルを表します。図 1-10 の例では、src ノードから gamma ノードに初期画像を送信したいので、src ノードから gamma ノードにエッジを作成します。次に、gamma ノードから tint ノードへのエッジを作成します。同様に、tint ノードから write ノードへのエッジを作成します。グラフ構造の構築が完了したら、src.activate() を呼び出して input_node を開始し、g.wait_for_all() を呼び出してグラフが完了するまで待機します。

図 1-10 のアプリケーションが実行されると、src ノードで生成された各イメージは、前述のようにノードのパイプラインを通過します。画像が gamma ノードに送信されると、TBB ライブラリーは gamma ノードの本体を画像に適用するタスクとして作成しスケジュールします。処理が完了すると、出力は tint ノードに送られます。同様に、TBB は gamma ノードの出力に対して tint ノードの本体を実行するタスクを作成してスケジュールします。最後に、処理が完了すると、tint ノードの出力が write ノードに送信されます。ここでも、ノード本体を実行するタスクが作成され、スケジュールされます。この例では、画像をファイルに書き込みます。src ノードの実行が終了して true を返すたびに、新しいタスクが生成され、src ノードの本体が再度実行されます。src ノードが新しい画像の生成を停止し、すでに生成したすべての画像が書き込みノードで処理を完了した後にのみ、wait_for_all 呼び出しから返ります。

parallel_for を使用した Fork-Join レイヤーの追加

さて、applyGamma 関数と applyTint 関数の実装に注目してみましょう。図 1-11 では、シリアル実装の外側の i ループを tbb::parallel_for 呼び出しに置き換えています。parallel_for 汎用並列アルゴリズムは、異なる行を並列に実行します。parallel_for は、プラットフォーム上の複数のプロセッサー・コアに分散できるタスクを作成します。

```
ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, gamma](int i) {
      for ( int j = 0; j < width; ++j ) {
        const ch01::Image::Pixel& p = in_rows[i][j];
        double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
        double res = pow(v, gamma);
        if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
        out_rows[i][j] = ch01::Image::Pixel(res, res, res);
      } }
    );
    return output_image_ptr;
  }

  ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
    auto output_image_ptr =
      std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
        ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, tints](int i) {
      for ( int j = 0; j < width; ++j ) {
        const ch01::Image::Pixel& p = in_rows[i][j];
        std::uint8_t b = (double)p.bgra[0] +
                    (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
        std::uint8_t g = (double)p.bgra[1] +
                    (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
        std::uint8_t r = (double)p.bgra[2] +
                    (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
        out_rows[i][j] =
          ch01::Image::Pixel(
            (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
            (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
            (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
          );
      } }
  );
  return output_image_ptr;
}
```

図 1-11. parallel_for を追加して、ガンマ補正と色合いを行全体で並列に実行。サンプルコード: intro/intro_parallel_for.cpp

標準 C++17 の実行ポリシーを使用したベクトル化

内部ループ j を STL 関数 transform の呼び出しに置き換えることで、2 つの計算カーネルをさらに最適化できます。変換アルゴリズムは、入力レンジ内の各要素に関数を適用し、結果を出力レンジに格納します。変換の引数は (1) 実行ポリシー、(2 と 3) 要素の入力レンジ、(4) 出力レンジの先頭、および (5) 入力レンジ内の各要素に適用され、結果が出力要素に格納されるラムダ式です。

図 1-12 では、C++17 の std::unseq 実行ポリシーを使用して、コンパイラーに変換関数の SIMD バージョンを使用するように指示しています。

```
#include <algorithm>
#include <execution>
#include <iostream>
#include <vector>
#include <tbb/tbb.h>
#include "intro_examples.h"

using ImagePtr = std::shared_ptr<ch01::Image>;
void writeImage(ImagePtr image_ptr);

ImagePtr applyGamma(ImagePtr image_ptr, double gamma) {
  auto output_image_ptr =
    std::make_shared<ch01::Image>(image_ptr->name() + "_gamma",
      ch01::IMAGE_WIDTH, ch01::IMAGE_HEIGHT);
  auto in_rows = image_ptr->rows();
  auto out_rows = output_image_ptr->rows();
  const int height = in_rows.size();
  const int width = in_rows[1] - in_rows[0];

  tbb::parallel_for( 0, height,
    [&in_rows, &out_rows, width, gamma](int i) {
      auto in_row = in_rows[i];
      auto out_row = out_rows[i];
      std::transform(std::execution::unseq, in_row, in_row+width,
        out_row, [gamma](const ch01::Image::Pixel& p) {
          double v = 0.3*p.bgra[2] + 0.59*p.bgra[1] + 0.11*p.bgra[0];
          double res = pow(v, gamma);
          if(res > ch01::MAX_BGR_VALUE) res = ch01::MAX_BGR_VALUE;
          return ch01::Image::Pixel(res, res, res);
        });
      }
    );
    return output_image_ptr;
  }

  ImagePtr applyTint(ImagePtr image_ptr, const double *tints) {
    auto output_image_ptr =
       std::make_shared<ch01::Image>(image_ptr->name() + "_tinted",
     ch01::IMAGE_WIDTH,
         ch01::IMAGE_HEIGHT);
    auto in_rows = image_ptr->rows();
    auto out_rows = output_image_ptr->rows();
    const int height = in_rows.size();
    const int width = in_rows[1] - in_rows[0];

    tbb::parallel_for( 0, height,
      [&in_rows, &out_rows, width, tints](int i) {
        auto in_row = in_rows[i];
        auto out_row = out_rows[i];
        std::transform(std::execution::unseq, in_row, in_row+width,
          out_row, [tints](const ch01::Image::Pixel& p) {
            std::uint8_t b = (double)p.bgra[0] +
                        (ch01::MAX_BGR_VALUE-p.bgra[0])*tints[0];
            std::uint8_t g = (double)p.bgra[1] +
                        (ch01::MAX_BGR_VALUE-p.bgra[1])*tints[1];
            std::uint8_t r = (double)p.bgra[2] +
```

```
                    (ch01::MAX_BGR_VALUE-p.bgra[2])*tints[2];
            return ch01::Image::Pixel(
              (b > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : b,
              (g > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : g,
              (r > ch01::MAX_BGR_VALUE) ? ch01::MAX_BGR_VALUE : r
          );
        });
      }
    );
    return output_image_ptr;
  }
```

図 1-12. `std::transform` を使用して内部ループに SIMD 並列処理を追加。サンプルコード: `intro/intro_parallel_for_transform.cpp`

図 1-12 では、各 `Image::Pixel` オブジェクトに、ピクセルの青、緑、赤、アルファー値を表す 4 つの 1 バイト要素の配列が含まれています。`unseq` 実行ポリシーを使用すると、ベクトル化されたループが要素の行全体に関数を適用するために使用されます。このレベルの並列化では、コードが実行される CPU コアのベクトルユニットを利用しますが、計算を異なるコアに分散することはありません。

注: 実行ポリシーを STL アルゴリズムに渡しても並列実行が保証されるわけではありません。ライブラリーが、要求された実行ポリシーよりも制限された実行ポリシーを選択することは認められています。したがって、実行ポリシー、特にコンパイラーの実装に依存する実行ポリシーを使用する場合は影響を確認することが重要です！

図 1-7 から 1-12 で作成した例は少し作為的ですが、TBB ライブラリーの並列実行インターフェイスの幅広さと強力さを示しています。

まとめ

この章では、TBB のようなライブラリーが最初に導入されたときと同じように、今日でも非常に重要であることを示しました。並列実行インターフェイスや実行インターフェイスに依存しない機能など、ライブラリーの主な機能について簡単に説明しました。そして、簡単な例を記述、コンパイル、実行することで、TBB 開発環境が正しく設定されていることを確認する方法を紹介しました。この章の最後では、3 つのレベルの並列処理を使用する完全な例をビルドし、TBB と標準 C++ 並列処理との組み合わせの可能性を示しました。

次の章では、並列プログラミングの主要なサポートについて説明します。

2 章 アルゴリズム

効率良い並列プログラムの作成は、アルゴリズムの実装に大きく依存します。多くのタスクパターンは実際に効果があることが証明されており、TBB アルゴリズムはこれらの実証済みのパスに導きます。

TBB は、主要なタスクパターンをサポートする 8 つの並列アルゴリズムを提供します。C++ 標準テンプレート・ライブラリー（STL）とその 100 を超えるアルゴリズム、および 80 を超えるさまざまな並列アルゴリズムを含む Parallel STL（PSTL）ライブラリーに精通している開発者は、わずか 8 つのアルゴリズムしかないことを意外に思うかもしれません。TBB の 8 つのアルゴリズムは、汎用的かつ必須の並列アルゴリズムです。これらは、一般に*並列パターン*と呼ばれる実装です。この必要かつ十分なパターンのセットに焦点を絞ることで、TBB ライブラリーは、幅広いアプリケーション・ドメインにわたって追加のアルゴリズムを実装するのに使用できる、高度に調整された汎用アルゴリズムのセットを提供できます。

8 つを鍵として選ぶ

8 つの TBB アルゴリズムは、7 つの主要アルゴリズム構成要素と、1 つの便利なアルゴリズム (ソート) から構成されています。TBB アルゴリズムにソートが含まれていることは、有用ではありますが、他のアルゴリズムとは異なる存在です。TBB のアルゴリズムのリストは、タスクスチールを考案した MIT の研究プロジェクト「`Cilk`」に影響を受けています。`Cilk` は当初、`cilk_spawn` と `cilk_sync` という 2 つの構成要素に限定されていたため、プログラマーにとって実際には困難なことがありました。`Cilk` は後に `cilk_for` を追加しましたが、それでもユーザーにとって理想的ではありませんでした。10 年程前、Microsoft は 8 つすべてではなく少数の TBB アルゴリズムで十分であると考え、TBB アルゴリズムのサブセットを直接サポートしました。ユーザーからのフィードバックでは、TBB がサポートする 8 つが圧倒的に好評でした。TBB が、コミュニティーからのフィードバックを基にフローグラフのサポートとタスクグループのサポートを追加し、並列パターンのサポートを充実させたことは注目に値します（これらについては 4 章と 5 章で説明します）。8 つのアルゴリズムは完璧ではないかもしれませんが、TBB に残っている 8 つのアルゴリズムは、時間の経過とともに、非常に幅広いユーザーのニーズを満たすのに効果的であることが証明されています。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_2

TBB アルゴリズムは、幅広いアプリケーション・ドメインで使用される並列実行パターンを表現する汎用関数です。通常、これはコレクション内の要素の検索や行列の乗算など、特定のドメインまたは数学の問題を解決するのではなく、特定の問題のソリューションを効率的に表現するために使用されます。実際、oneAPI データ並列ライブラリー（C++ Parallel STL の実装）を含む、oneAPI エコシステム内の多くのドメイン固有ライブラリーおよびパフォーマンス・ライブラリーは、これらの 8 つの TBB アルゴリズムを基に幅広いアルゴリズムを表現しています。

図 2-1 は、TBB が提供する一連の関数と簡単な説明、および実装に使用される主なソフトウェアの設計パターンを示しています。3 列目のデザインパターンを知らなくても心配ありません。この節の後半で簡単に説明します。図 2-1 は、アルゴリズムのスケーラビリティーによって大まかに順序付けられています。表の中でアルゴリズムが上位にあるほど、タスクの順序付けに対する制約が少なくなります。

並列アルゴリズム	説明	主なデザインパターン	説明がある節 （詳細はどこにあるか）
既知のセットから独立したタスクを作成			
`parallel_for`	値のレンジの並列反復を実行します。	map	独立したタスクパターン （105 ページから）
`parallel_invoke`	複数の関数を並列に評価します。	主に fork-join パターンを実装するために使用されます。ネストと組み合わせると、分割統治法、分岐限定法、その他のツリーベースのパターンを実装できます。	独立したタスクパターン （105 ページから）
拡張セットから独立したタスクを作成			
`parallel_for_each`	レンジに動的に値を追加するオプションを備えた、`std::for_each` の並列実装。	work-pile	独立したタスクパターン （105 ページから）
連想操作のタスクグラフ作成			
`parallel_reduce`	値のレンジ全体のリダクションを計算します。	reduction	単一の値を計算するタスクパターン （123 ページから）
`parallel_deterministic_reduce`	決定論的な結合性を持つ値のレンジにわたるリダクションを計算します。	再現可能な結果による reduction	単一の値を計算するタスクパターン （123 ページから）
`parallel_scan`	値のレンジ全体の並列プレフィクスを計算します。	`scan/prefix`	単一の値を計算するタスクパターン （123 ページから）
関数のパイプライン実行を実装するタスクグラフの作成			
`parallel_pipeline`	フィルターのパイプラインを実行。	`pipeline`	パイプラインのパターン （136 ページから）
ソート			
`parallel_sort`	並列クイックソートを実行します。。	**N/A** ソートは非常に一般的なアルゴリズムですが、基本的な並列パターンとは見なされません。	`parallel_sort` （147 ページから）

図 2-1. スレッディング・ビルディング・ブロック・ライブラリーの汎用アルゴリズムは、タスク間の依存関係の制約が最も少ないものから多いものの順に並べられています

この章では、図 2-1 にリストされている 8 つの TBB アルゴリズムすべて詳しく説明します。図 2-1 の最初の 7 つのアルゴリズムは、特定のニーズを満たす独自アルゴリズムを構築する構成要素として機能します。`parallel_sort` は便利ですが、ソートという特定の問題を解決するため、アルゴリズムからは少し外れたものと考えます。TBB が初めて導入されたとき、ソートはユーザーによって主要なアルゴリズムとして認識され、それ以来維持されてきました。

並列パターンを TBB にマッピングすることへのコメント

図 2-1 や図 2-2 に示されるソフトウェア設計パターンは、プログラミングにおいて長い歴史を持っています。オブジェクト指向プログラミングにおけるパターンは、俗に GoF（Gang of Four）と呼ばれる（Erich Gamma 氏、Richard Helm 氏、Ralph Johnson 氏、John Vlissides 氏）の共著『オブジェクト指向における再利用のためのデザインパターン』（ソフトバンク・クリエイティブ発行）で有名になりました。多くの人々が、オブジェクト指向プログラミングの世界に秩序をもたらしたこの書籍を賞賛しています。この書籍にはコミュニティーの英知が集められ、パターンというシンプルな名前にまとめられているため、人々がパターンについて議論できるようになりました。

デザインパターン	説明
ネスト	別のパターン内でパターンを実行。TBB はすべてネスト機能をサポートします。
フォーク・ジョイン	制御フローを 2 つ以上の並列パスに分割し、それらを 1 つのパスに再結合します。すべての TBB アルゴリズムは内部的にフォーク・ジョイン・パターンです。各アルゴリズムをベースに作成される並列処理は、呼び出し中に開始され、呼び出しが完了する前に再結合されるため、TBB アルゴリズムについてはローカルかつ段階的に考えることができます。すべての TBB アルゴリズムに共通するこの特性により、`parallel_invoke` アルゴリズム、`task_group`（6 章で説明）、およびフローグラフ（4 章で説明）は、アプリケーションでフォーク・ジョイン・パターンを簡単に表現できる API です。
マップ	ワークを均一な独立したタスクに分割します。これは、最小限の順序付けと同期を必要とするため、使用するのに最適なパターンです。TBB の `parallel_for` を使用してマップを実装できます。
ワークパイル	ワークを均一な独立したタスクに分割し、他の項目を処理している間に新しいワーク項目を動的に追加できます。TBB の `parallel_for_each` を使用してワークパイルを実装できます。
分割統治	ワークを徐々に小さなワーク単位に再帰的に分割します。このパターンでは通常、各レイヤーに次の 3 つのステップが含まれます: (1) サブ問題に分割し、(2) 各サブ問題を解決し、(3) サブ問題の解決方法を組み合わせる。サブ問題は、ベースケースに到達するまで同じ方法で再帰的に解決されます。`parallel_invoke` アルゴリズムと `task_group` は、分割統治パターンを実装するのに通常使用される TBB の機能です。
分岐限定	ワークを徐々に小さなワーク単位に再帰的に分割しますが、すべてのサブ問題を無駄に実行する可能性を減らすため、不要な項目を削除またはキャンセルする場合があります。これは検索によく使用されます。キャンセルと組み合わせた `parallel_invoke` アルゴリズムと `task_group` は、分岐限定パターンを実装するのに通常使用される TBB の機能です。
リダクション	ワークを独立したタスクに分割し、それぞれの部分的な結果を計算し、1 つの結果にまとめます。分割統治の特殊な例。TBB は、リダクションを解決するため `parallel_reduce` と `parallel_deterministic_reduce` という 2 つのアルゴリズムを提供します。
スキャン/プリフィクス	ワークを独立したタスクに分割して、部分的な結果を計算し、それらを結合しますが、個々のワーク項目ごとに中間結果も計算されます。`y[i]=y[i-1] op f(i)` の並列実装。TBB の `parallel_scan` はスキャン/プレフィクスを実装します。
パイプライン	生産と消費操作の固定された線形チェーンを通じてアイテムを渡します。TBB の `parallel_pipeline` はパイプラインの実装に使用されます。
イベントベースの調整	生産と消費、およびデータフローのパターンを表現するパターン。フローグラフ（4 章で説明）は、イベントベースの調整に使用されます。

図 2-2. 並列プログラミングに関連する重要な設計パターンと TBB アルゴリズムとの関係に関する用語

Mattson、Sanders、Massingill による「*Patterns for Parallel Programming*」（Addison-Wesley）も同様に、並列プログラミングのコミュニティーからの知識を集約しています。専門家は共通な手法を用い、独自の言葉で技術を述べます。これらのパターンを念頭に置くことで、オブジェクト指向プログラマーが有名な GoF（Gang of Four）本を通じて学んできたように、プログラマーは並列プログラミングを素早く習得できます。

「*Patterns for Parallel Programming*」は、この書籍よりページ数も多く、非常に密度の高い内容ですが、著者の Tim Mattson 氏の協力により、パターンが TBB とどのように関連しているかまとめることができました。

Mattson 氏らは、並列プログラムを開発するにあたり、プログラマーは 4 つの設計空間を検討する必要があると述べています。以下でそれらについて説明します。

並行性の発見

この設計空間では、問題領域内で利用可能な同時実行性（並行性）を特定し、それをアルゴリズムの設計で利用できるようにします。TBB は、ハードウェア・スレッドにタスクをマップする方法を気にすることなく、できるだけ多くのタスクを見つけることに集中できるようにすることで、プログラマーの作業を簡素化します。また、タスクが大きいと判断された場合に、タスクを半分に分割する最適な方法も提供します。TBB はこの情報を使用して、自動的に大きなタスクを繰り返し分割し、プロセッサー・コア間でワークを均等に分散できるようにします。タスクが豊富であると、アルゴリズムのスケーラビリティーが向上します。

アルゴリズム構造

この設計空間は、並列アルゴリズムを編成する高レベルの方針を具体化します。ワークフローをどのように構成するか考えなければなりません。図 2-2 には、ニーズに最適なパターンを選択するのに参照できる重要なパターンが示されています。これらの「機能するパターン」は、McCool、Robison、Reinders（Elsevier）による『*Structured Parallel Programming*（構造化並列プログラミング）』で説明されており、パターンについてさらに深く知りたい人に役立つ書籍です。TBB を効果的に使用するためには必要はありません。

サポートされる構造

このステップには、アルゴリズムの設計方針を実際のコードとして実装する詳細が含まれます。並列プログラムがどのように構成されるか、共有データ（特に変更可能なデータ）を管理するためどのような手法が使用されるか考えます。これらの考慮事項は重要であり、並列プログラミングの過程全体に影響します。TBB は適切なレベルの抽象化を促すように設計されているので、TBB を適切に使用することでこの設計空間が満たされます（これは、この本で伝えたいことです）。

実装のメカニズム

この設計空間にはスレッド管理と同期が含まれます。TBB は、プログラマーがより高い設計レベルでタスクについて考えることができるよう、スレッド管理をすべて処理します。TBB を使用する場合、ほとんどのプログラマーは明示的な同期を回避するようにコーディングし、デバッグを行います。この章で説明する TBB アルゴリズムとフローグラフ API は、明示的な同期を最小限にするために使用されます。

パターン言語を使用すると、優れた並列プログラミング環境の構築に役立ち、TBB を最大限に活用して並列ソフトウェアを記述できるようになります。並列パターンを TBB アルゴリズムにマッピングすることを議論するのは、パターンに一致するスケーラブルなアルゴリズムを見つけることが目的です。スケーラブルなアルゴリズムは、追加のコアとハードウェア・リソースが利用可能になると、それらを効果的に利用できます。 歴史的に、スケーリングは、*強いスケーリング*と*弱いスケーリング*の 2 つに分類されてきました。

コアが増加するにつれて、固定サイズの問題を解決する時間が短くなる場合、アルゴリズムは強いスケーリングを示します。例えば、強いスケーリングを示すアルゴリズムで 2 つのコアを利用可能な場合、同じデータセットの処理がシーケンシャル・アルゴリズムの 2 倍の速さで完了し、100 個のコアが利用可能であれば、同じ処理は 100 倍の速さで完了します。

プロセッサーを増やしても、*プロセッサーあたり*のデータ・セット・サイズが固定されている問題を解決するのにかかる時間が同じである場合、アルゴリズムは弱いスケーリングを示します。弱いスケーリングを示すアルゴリズムでは、2 つのプロセッサーを利用して一定期間内にシリアルバージョンの 2 倍のデータを処理できる可能性があり、100 個のプロセッサーを使用すると、同じ期間内にシリアルバージョンの 100 倍のデータを処理できる可能性があります。

*強いスケーリング*と*弱いスケーリング*のどちらかが本質的に優れているというわけではなく、すべてはユースケースによって決まります。問題のサイズが固定されており、それをより速く解決したい場合は、強いスケーリングが必要です。問題のより正確な（またはより良い）結果を得るため、処理するデータの量を増やしたい場合は、弱いスケーリングが必要です。

独立したタスクパターン

最も効果的な並列プログラムを実現するには、計算を可能な限り独立させる方法を常に考えなければなりません。したがって、最初に完全に独立したタスクのアルゴリズムについて説明します。可能であれば、プログラミング問題をアルゴリズムにマッピングしてみるのが賢明です。ここでは、並列プログラミングの効果を最大化するために必要な、独立性を最大化する方法で依存関係を処理するアルゴリズムを紹介します。

独立したタスクは複雑な同期を必要としないため、タスク・スケジューラーにとって理想的なものであり、完全な自由を提供します。この節では、独立したタスクを生成する 3 つのアルゴリズムについて説明します。並列タスクとして実行される関数のセットの実行を指示するには、`parallel_invoke` を使用します。ループ反復からタスクを作成するには `parallel_for` を使用しますが、レンジは再帰的に分割可能である必要があります。つまり、レンジを効率良く繰り返し 2 つの部分に分割できます。最後に、`parallel_for_each` を使用して、コンテナーをトラバースするループ、または開始イテレーターから終了イテレーターまでトラバースするループからタスクを生成します。スケーラビリティーを高めるため、`parallel_for_each` では、既知のタスクを実行しながら新しいタスクを追加することもできます。これらのパターンに共通の特徴は、一度作成されると、アルゴリズムによって生成されたタスクは相互に順序付けされず、任意の順序で、TBB の任意のワーカースレッドによって実行できることです。

図 2-1 と 2-2 では、これらのアルゴリズムを使用して設計パターンを実装できることを示しました。`parallel_invoke`、`parallel_for`、`parallel_for_each` の違いは、開発者が独立したタスクをどのように記述するかであり、この違いはスケーラビリティーとマッピングできるワークロードの種類に大きく影響します。

parallel_invoke: 関数呼び出しから独立したタスク

TBB 関数 `parallel_invoke` は、TBB アルゴリズムの中で最も理解しやすく、ユーザー定義の関数呼び出しを並列タスクとしてスケジュールし、それらのタスクが完了するまで呼び出しスレッドをブロックします。4 つの関数を渡し、4 つの並列タスクを実行して、完了するまで待機します。これはフォーク・ジョイン・パターンの典型的な例です。

parallel_invoke は、図 2-3 に示すように、2 つ以上のユーザー定義の関数を並列に実行する関数テンプレートです。

```
// ヘッダーで定義:  <tbb/parallel_invoke.h>
namespace  tbb  {
template<typename...Functions>
    void  parallel_invoke(Functions&&... fs);
}  // namespace tbb
```

図 2-3. [algorithms. parallel_invoke] で説明されている parallel_invoke アルゴリズム

例えば、2 つのベクトル v1 と v2 がある場合、各ベクトルに対して serialQuicksort を連続して呼び出すことで、これら 2 つのベクトルをソートできます:

```
serialQuicksort(v1.begin(), v1.end());
serialQuicksort(v2.begin(), v2.end());
```

これらを順番に実行すると、呼び出しを実行する合計時間は、最初の serialQuicksort の実行にかかる時間と 2 番目の serialQuicksort の実行にかかる時間の合計に等しくなります。

図 2-3 のキャプションにある [ALGORITHMS.PARALLEL_INVOKE] の意味は？

序文で述べたように、関連するオンライン仕様のページ（https://tinyurl.com/tbbspec）を参照するため、[X.Y]（例: [algorithms.parallel_invoke]）という表記を使用します。タイトル X（例: algorithms）は、oneTBB インターフェイスまたは oneTBB 補助インターフェイスのいずれかにあり、そこから Y（例: parallel_invoke）のリンクが存在するページに移動します。

図 2-4 に示すように、oneTBB の parallel_invoke アルゴリズムを使用して、これら 2 つの呼び出しを並列に実行できます。

```
#include  <vector>
#include  <tbb/tbb.h>

struct  DataItem  { int  id; double  value; };
using  QSVector  = std::vector<DataItem>;

template<typename  Iterator> void  serialQuicksort(Iterator  b,
Iterator  e);

void  example(QSVector&  v1, QSVector&  v2) {
  tbb::parallel_invoke(
    [&]() { serialQuicksort(v1.begin(), v1.end()); },
    [&]() { serialQuicksort(v2.begin(), v2.end()); }
  );
}
```

図 2-4. parallel_invoke を使用して、2 つの serialQuicksort 呼び出しを並列に実行します。サンプルコード algorithms/parallel_invoke_two_quicksorts.cpp

図 2-4 の oneTBB の parallel_invoke の呼び出しにより、異なるワーカースレッドで並列実行できる 2 つのタスクが作成され、これらの関数の実行が時間的にオーバーラップします。serialQuicksort の 2 回の呼び出しがそれぞれ同じ時間実行され、システム上の CPU が他の処理でビジー状態でない場合、この並列実装は、単一のスレッドで関数を連続して呼び出す時間のおよそ半分で完了できます。

しかし、parallel_invoke を使って 2 つのソートを並列に実行するというこの単純な例は、強いスケーリングも弱いスケーリングも示しません。この例では、独立したソートを 2 つだけ作成するため、プロセッサーを 2 つだけ使用します。利用可能なプロセッサーが 100 個ある場合、そのうち 98 個は何も処理しないためアイドル状態になります。つまり、強いスケーリングは不可能でであり、問題の規模を大きくしても役に立ちません。小さなソートを 2 つ実行するか、大きなソートを 2 つ実行するかにかかわりなく、パフォーマンスは最大で 2 倍向上します。したがって、これが皆さんのアプリケーションである場合、より多くのコアを備えた新しいコンピューターを導入しても期待する効果は得られません。

parallel_for: 既知のループ反復セットから独立したタスク

parallel_for アルゴリズムは、for ループの反復を独立したタスクとして実行する関数テンプレートです。図 2-5 に示すように、parallel_for で使用できる関数シグネチャーは多数あります。すべてのシグネチャーに共通するのは、TBB ライブラリーがレンジを再帰的に細分化することでタスクを効率的に生成できることです。

```
namespace tbb {

//! 整数の範囲にわたるステップなしの並列反復
template<typename Index, typename Func>
void parallel_for(Index  first, Index  last, const Func& f, partitioner,
                  task_group_context& context);
template<typename Index, typename Func>
void parallel_for(Index  first, Index  last, const Func& f);
/* … ここに示されていない他の 2 つの同様のシグネチャー … */

//! 整数の範囲にわたるステップありの並列反復
template<typename Index, typename Func>
void parallel_for(Index  first, Index  last, Index  step, const Func& f,
                  partitioner, task_group_context& context);
/* … ここに示されていない他の 3 つの同様のシグネチャー … */

//! TBB 範囲での並列反復
template<typename Range, typename Body>
void parallel_for(const Range& range, const Body& body, partitioner,
                  task_group_context& context);
/* … ここに示されていない他の 3 つの同様のシグネチャー … */

} // namespace tbb
```

図 2-5. [algorithms. parallel_for] で説明されている parallel_for の関数シグネチャー

多くの並列アプリケーションは一連の並列ループまたはネストされた並列ループとして表現されるため、parallel_for アルゴリズムは TBB で最も広く使用される機能です。parallel_for アルゴリズムはループ反復から独立したタスクを生成するため、反復回数が多いと潜在的なタスクも多数存在します。そのため、フォークジョイン並列処理の単一レイヤーを表現する場合、parallel_for は通常、parallel_invoke よりもスケーラブルです。

例えば、次のループは N 回の反復が行われ、反復間でデータの依存関係がないことが分かります:

```
for (int i = 0; i < N; ++i) {
   a[i] = f(a[i]);
}
```

図 2-5 で強調表示されているシグネチャーを使用して、parallel_for でこのループを並列化できます:

```
tbb::parallel_for(0, N, [&](int i) {
   a[i] = f(a[i]));
});
```

parallel_for を使用することで、ループ反復を任意の順序で並列実行しても安全であると宣言していることを理解することが重要です。TBB ライブラリーは、parallel_for の反復を並列に実行した場合に（実際には汎用アルゴリズムによって並列に作成された任意のタスク）、アルゴリズムのシリアル実行と同じ結果が生成されるか正当性はチェックしません。並列アルゴリズムを選択した場合にこれを確認するのは、開発者の仕事です。最終的には、並列アルゴリズムを使用する際に、読み取りおよび書き込みアクセスパターンの潜在的変更によって結果の有効性が変わらないようにする必要があります。また、並列コード内からスレッドセーフなライブラリーと関数のみを呼び出していることを確認する必要があります。

例えば、次のループは、各反復が前の反復の結果に依存するため、parallel_for として実行するのは安全ではありません。このループの実行順序を変更すると、配列 a の要素に格納される最終的な値が変更されます:

```
for (int i = 0; i < N; ++i) {
   a[i] = a[i-1] + 1;
}
```

配列 a={1,0,0,0,...,0} の場合を考えてみてください。このループを順番に実行すると、{1,2,3,4,...,N} が保持されます。しかし、ループが順序どおりに実行されなかった場合、結果は異なります。安全に並列実行できるループを探すときの心構えとしては、ループ反復をすべて一度に実行した場合、ランダムな順序で実行した場合、または逆の順序で実行した場合、結果が同じであるかを確認することです。この場合、a={1,0,0,0,...,0} でループ反復が逆の順序で実行されると、ループが完了したときに a は {1,2,1,1,...,1} を保持します。このループでは実行順序が重要です。

図 2-6 は、N x K および K x M 行列の c = ab を計算する行列乗算ループネストの最適化されていないシリアル実装を示しています。**先に進む前に、ほとんどの開発者は行列乗算コードを独自に記述してはならないことを警告しておかなければなりません。** ここでは、このカーネルをデモ目的で使用しています。実際のアプリケーションで行列乗算を使用する必要があり、最適化の専門家でないならば、マス・カーネル・ライブラリー（MKL）、BLIS、ATLAS などの Basic Linear Algebra Subprograms（BLAS）を実装する数学ライブラリーの高度に最適化された実装を使用する方がほとんどの場合で良い結果が得られます。

しかし、その注意点を念頭に置くと、行列乗算は小さなカーネルであり、私たちがよく知っている基本的な演算を実行するため、ここでは良い例と言えます。また、興味深いメモリー・アクセス・パターンも含まれていますが、これについては 11 章で再度取り上げます。これらの留意事項を説明した上で、図 2-6 に進みます。

```
template<typename  InMat1, typename  InMat2, typename  OutMat>
void  simpleSerialMatrixProduct(int  M, int  N, int  K, const  InMat1& a,
                        const  InMat2& b, OutMat& c) {
  for  (int  i0 = 0; i0 < M; ++i0) {
    for  (int  i1 = 0; i1 < N; ++i1) {
      auto& c0 = c[i0*N+i1];
      for  (int  i2 = 0; i2 < K; ++i2) {
        c0 += a[i0*K+i2] * b[i2*N + i1];
      }
    }
  };
}
```

図 2-6. 単純なシリアル行列積の実装。サンプルコード algorithms/parallel_for_unoptimized_mxm.cpp

図 2-7 に示すように、parallel_for を使用すると、図 2-6 の行列乗算を簡単に並列バージョンにできます。この実装では、外側の i0 ループを並列化します。外側の i0 ループの反復は、内側の i1 ループと i2 ループを実行します。多くの場合、オーバーヘッドを抑えるため、可能な限り外側のループを並列化すると良いでしょう。

```
template<typename  InMat1, typename  InMat2, typename  OutMat>
void  simpleParallelMatrixProduct(int  M, int  N, int  K, const  InMat1& a,
                          const  InMat2& b, OutMat& c) {
  tbb::parallel_for( 0, M, [&](int  i0) {
    for  (int  i1 = 0; i1 < N; ++i1) {
      double& c0 = c[i0*N+i1];
      for  (int  i2 = 0; i2 < K; ++i2) {
        c0 += a[i0*K+i2] * b[i2*N + i1];
      }
    }
  });
}
```

図 2-7. 単一の parallel_for を使用する単純な並列行列積の実装。サンプルコード algorithms/parallel_for_unoptimized_mxm.cpp

図 2-7 のコードは、行列乗算の基本的な並列バージョンを簡単に実現したものです。これは正しい並列実装ですが、配列を走査するためパフォーマンスが大幅に低下します。後ほど、parallel_for の他の関数シグネチャーによって性能を向上させる高度な機能について説明します。

拡張セットから独立したタスクを作成する parallel_for_each

parallel_invoke と parallel_for の両方を使用すると、並列タスクを効率良く識別できます。parallel_invoke ではタスクが引数になります。parallel_for の場合、タスクは複数のスレッドを使って再帰的にレンジを分割することで生成できます。parallel_for_each アルゴリズムを使用すると、図 2-2 の *ワークパイル*を表現できます。ワークパイルは、独立したタスクを作成し、開始時にそのすべて効率的に列挙できない場合に使用されます。

while ループの並列バージョンを作成する簡単な例を示します:

```
while (auto i = get_image()) {
    f(i);
}
```

このループは、画像がなくなるまで画像を読み取り続けます。各画像が読み込まれた後、関数 f によって処理されます。画像の数が不明でレンジを指定できないため、parallel_for は使用できません。

より微妙なケースは、ランダム・アクセス・イテレーターを提供しないコンテナーがある場合です:

```
std::list<image_type> my_images = get_image_list();
for (auto &i : my_list) {
    f(i);
}
```

std::list は要素へのランダムアクセスをサポートしないため、レンジ my_images.begin() と my_images.end() の区切りを取得することはできますが、リストを順番に走査しないとポイント間の要素にアクセスできません。したがって、TBB ライブラリーは、コンテナーを走査しないとチャンクの開始点と終了点を指定できないため、複数のスレッドにタスクとして渡す反復チャンクを迅速に作成することができません。

このようなループを管理するため、TBB ライブラリーでは parallel_for_each が提供されます。TBB の parallel_for_each は、処理する項目がなくなるまで、ワーク項目に Body を適用します。一部のワーク項目はループの開始時に事前に提供でき、その他のワーク項目は他の項目を処理中に Body の実行によって追加できます。

parallel_for_each 関数シグネチャーには、図 2-8 に示すように、最初と最後のイテレーターを受け入れるバリアントと、C++ のレンジを受け入れるバリアントの 2 つのバリエーションがあります。

```
namespace  tbb {

//! range にわたって並列反復を実行し、
//  オプションでさらにワークを追加。
template<typename  InputIterator, typename  Body>
void  parallel_for_each(InputIterator  first, InputIterator  last,
                        Body  body);
template<typename  InputIterator, typename  Body>
void  parallel_for_each(InputIterator  first, InputIterator  last,
                        Body  body, task_group_context& context);
template<typename  Container, typename  Body>
void  parallel_for_each(Container& c, Body  body);
template<typename  Container, typename  Body>
void  parallel_for_each(Container& c, Body  body,
                        task_group_context& context);
template<typename  Container, typename  Body>
void  parallel_for_each(const  Container& c, Body  body);
template<typename  Container, typename  Body>
void  parallel_for_each(const  Container& c, Body  body,
                        task_group_context& context);

}  // namespace tbb
```

図 2-8. [algorithms.parallel_for_each] で説明されている parallel_for_each の関数シグネチャー

簡単な例として、std::pair<int, bool> 要素の std::list から始めましょう。各要素にはランダムな整数値と false が含まれています。各要素について、int 値が素数であるかを計算し、素数の場合は true をブール値に格納します。コンテナーにデータを入力し、数値が素数であるか判定する関数が与えられていると仮定します。シリアル実装を図 2-9 に示します。

```
#include  <list>
#include  <utility>
#include  <tbb/tbb.h>

using  PrimesValue  = std::pair<int, bool>;
using  PrimesList  = std::list<PrimesValue>;
bool  isPrime(int  n);

//
// リスト内の各要素をチェックし、素数の場合は true を割り当てる単純なシリアル実装
//
void  serialPrimesList(PrimesList&  values) {
  for  (PrimesList::reference v : values) {
     if  (isPrime(v.first)) v.second  = true;
   }
}
```

図 2-9. std::list を使用した素数のシリアル実装の例。サンプルコード algorithms/parallel_for_each_primes.cpp

```
void  parallelPrimesList(PrimesList& values) {
 tbb::parallel_for_each(values,
   [](PrimesList::reference  v)
     {
     if  (isPrime(v.first))
        v.second = true;
     }
  );
}
```

図 2-10. TBB の parallel_for_each を使用して std::list を処理する素数の並列実装の例。サンプルコード algorithms/parallel_for_each_primes.cpp

図 2-10 に示すように、TBB の parallel_for_each を使用して、このループの並列実装を作成できます。

TBB の parallel_for_each アルゴリズムは、各要素にボディーを適用するタスクを作成しながら、コンテナーを安全に順次走査します。コンテナーは順番に走査される必要があるため、parallel_for_each は parallel_for ほどスケーラブルではない可能性がありますが、ボディーが比較的大きい限り（例えば、実行時間が 1 マイクロ秒以上）、要素に対するボディーの並列実行と比較して、走査のオーバーヘッドはごくわずかです。

ランダムアクセスを提供しないコンテナーの処理に加えて、parallel_for_each を使用すると、ボディーの実行からワーク項目を追加することもできます。ボディーが並列に実行され、新しい項目が追加されると、これらの項目も並列に生成されるため、parallel_for_each の順次タスク生成の制限を回避できます。

図 2-11 は、値が素数であるか計算するシリアル実装を示していますが、値はリストではなくツリーに格納されます。

```
void  serialPrimesTree(PrimesTreeElement::Ptr  e) {
   if  (e) {
     if  (isPrime(e->v.first))
       e->v.second  = true;
     if  (e->left) serialPrimesTree(e->left);
     if  (e->right) serialPrimesTree(e->right);
   }
}
```

図 2-11. 要素のバイナリーツリーを使用した素数のシリアル実装の例。サンプルコード: algorithms/parallel_for_each_primes.cpp

図 2-12 に示すように、parallel_for_each を使用して、このツリーバージョンの並列実装を作成します。この実装でワーク項目を提供するさまざまな方法で提供できることを示すため、値の 1 つのツリーを保持するコンテナーを使用します。parallel_for_each は 1 つのワーク項目のみで開始されますが、各ボディー実行で 2 つの項目が追加されます。1 つは左のサブツリーを処理し、もう 1 つは右のサブツリーを処理します。tbb::feeder<T>::add メソッドを使用して反復空間に新しいワーク項目を追加します。クラス tbb::feeder<T> は TBB ライブラリーによって定義されており、このクラスのインスタンスがボディーの 2 番目の引数として渡されます。

ボディーがツリーのレベルを下るにつれて、利用可能なワーク項目の数は指数的に増加します。図 2-12 では、現在の要素が素数であるか確認する前でも、フィーダーを通じて新しい項目を追加して、他のタスクができるだけ早く生成されるようにしています。

```
void  parallelPrimesTree(PrimesTreeElement::Ptr  root) {
    PrimesTreeElement::Ptr tree_array[] = {root};
    tbb::parallel_for_each(tree_array,
     [](PrimesTreeElement::Ptr  e,
       tbb::feeder<PrimesTreeElement::Ptr>& f) {
         if  (e) {
           if  (e->left) f.add(e->left);
           if  (e->right) f.add(e->right);
           if  (isPrime(e->v.first))
             e->v.second = true;
         }
       }
  );
}
```

図 2-12. 要素のバイナリーツリー、parallel_for_each、およびフィーダーを使用して新しい要素をワークパイルに追加する、素数の並列実装の例です。サンプルコード: algorithms/parallel_for_each_primes.cpp

ここで検討した 2 つの parallel_for_each の使用法は、それぞれ異なる理由でスケールする可能性があることに注意する必要があります。図 2-10 のフィーダーなしの最初の実装では、各ボディー実行にリストを順番に走査するオーバーヘッドを軽減する十分なワークがある場合に良好なパフォーマンスを示すことができます。2 番目の実装では、図 2-12 のフィーダーを使用して、1 つのワーク項目で開始しますが、ボディーが実行され、新しい項目が追加されるにつれて、使用可能なワーク項目の数が急速に増加します。

本書で何度か取り上げるさらに複雑な例は、前方置換です。前方置換は、方程式 Ax = b のセットを解く簡単な方法です。ここで、A は n x n の下三角行列です。図 2-6 の行列乗算の例と同じ警告がここでも当てはまります。並列ソルバー用に高度に調整されたライブラリーがあるので、この分野の専門家でない限り、それらを使用するべきであり、独自のコードを記述するのは推奨されません。しかし、行列の乗算と同様に、これは並列性を示す良い例です。

行列として見ると、方程式は次のようになります:

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

そして、これは 1 行ずつ解くことができます:

$$x_1 = b_1 / a_{11}$$

$$x_2 = \left(b_2 - a_{21}x_1\right) / a_{22}$$

$$x_3 = \left(b_3 - a_{31}x_1 - a_{32}x_2\right) / a_{33}$$

...

$$x_m = \left(b_n - a_{n1}x_1 - a_{n2}x_2 - \ldots - a_{nn-1}x_{n-1}\right) / a_{nn}$$

このアルゴリズムを直接実装したシリアルコードを図 2-13 に示します。シリアルコードでは、b は各行の合計を格納するため破壊的に更新されます。

```
void  serialFwdSub(std::vector<double>& x,
                   const  std::vector<double>& a,
                   std::vector<double>& b) {
  const  int  N = x.size();
  for  (int  i = 0; i < N; ++i) {
     for  (int  j = 0; j < i; ++j) {
       b[i] -= a[j + i*N] * x[j];
     }
     x[i] = b[i] / a[i + i*N];
  }
}
```

図 2-13. 前方置換を直接実装するシリアルコード。この実装は、アルゴリズムを明確にするために書かれたものであり、最高のパフォーマンスを目的としたものではありません。サンプルコード: algorithms/parallel_for_each_fwd_substition.cpp

図 2-14(a) は、図 2-13 のループ i と j のネストボディーの反復間の依存関係を示しています。内側の j ループの各反復（図の行で表示）では、b[i] へのリダクションが実行され、i ループの以前の反復で書き込まれた x のすべての要素に依存します。内部の j ループを並列化するため parallel_reduce を使用できますが、i ループの初期の反復では、これを効果的に行うにはワークが十分ではない可能性があります。図 2-14(a) の点線は、反復空間を対角線上に調べることで、このループネスト内の並列性を見つける別の方法があることを示しています。この並列処理は、図 2-12 で新しいツリー要素を検出して追加したように、依存関係が満たされた場合にのみ反復処理を追加する parallel_for_each を使用して利用できます。

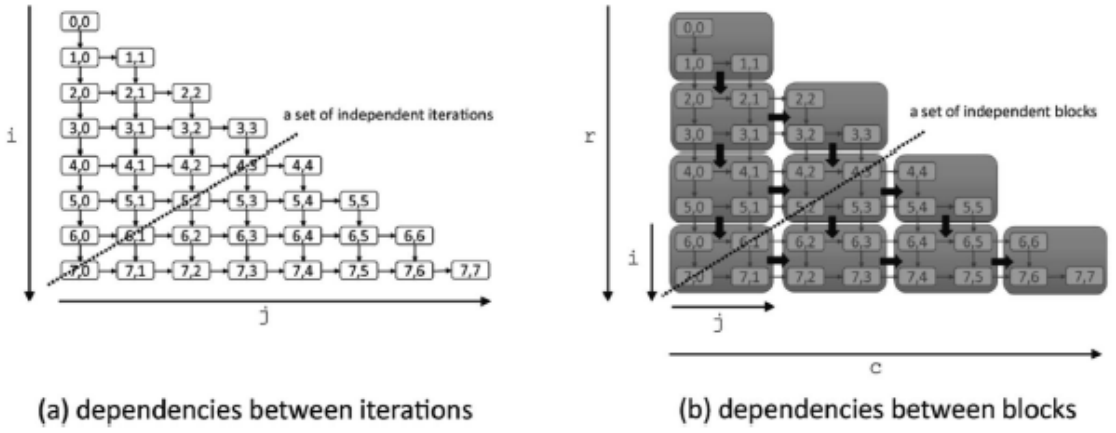


図 2-14. 小さな 8 × 8 行列の前方置換における依存関係。(a) には反復間の依存関係が表示されます。(b) では、スケジュールのオーバーヘッドを削減するため反復がブロックにグループ化されています。(a) と (b) の両方において、各ブロックは、安全に実行する前に、上と左の隣接ブロックが完了するまで待機する必要があります。

各反復の並列性を個別に表現すると、各タスクは少数の浮動小数点演算のみになるため、スケジュールのオーバーヘッドを相殺するには小さすぎるタスクが作成されます。代わりに、図 2-14(b) に示すように、ループネストを変更して反復ブロックを作成できます。依存パターンは同じままですが、この大きな反復ブロックをタスクとしてスケジュールできるようになります。シリアルコードのブロックバージョンを図 2-15 に示します。

```
const  int  block_size = 512;

void  serialFwdSub(std::vector<double>& x,
                   const  std::vector<double>& a,
                   std::vector<double>& b) {
  const  int  N = x.size();
  for  (int  i = 0; i < N; ++i) {
    for  (int  j = 0; j < i; ++j) {
      b[i] -= a[j + i*N] * x[j];
    }
    x[i] = b[i] / a[i + i*N];
  }
}

static  inline  void
computeBlock(int  N, int  r, int  c,
             std::vector<double>& x,
             const  std::vector<double>& a,
             std::vector<double>& b);

void  serialFwdSubTiled(std::vector<double>& x,
              const  std::vector<double>& a,
              std::vector<double>& b) {
  const  int  N = x.size();
  const  int  num_blocks = N / block_size;

  for  ( int  r = 0; r < num_blocks; ++r ) {
    for  ( int  c = 0; c <= r; ++c ) {
      computeBlock(N, r, c, x, a, b);
    }
  }
}
```

図 2-15. 前方置換のブロックされたシリアル実装のブロックバージョン。サンプルコード: `algorithms/parallel_for_each_fwd_substition.cpp`

`parallel_for_each` を使用した並列実装を図 2-16 に示します。ここでは、コンテナー全体ではなく、開始と終了イテレーターを指定できる `parallel_for_each` へのインターフェイスを使用します。

図 2-12 の素数ツリーの例とは異なり、すべての隣接ブロックを単純にフィーダーに送信する代わりに、カウンターの配列 `ref_count` を初期化し、各ブロックが実行を開始する前に完了しなければならないブロックの数を保持します。アトミック変数については 8 章で詳しく説明します。ここでは、これらを並列に安全に変更できる変数とみなすことができます。特に、デクリメントはスレッドセーフな方法で行われます。

左上の要素には依存関係がなく、最初の列と対角線に沿ったブロックには 1 つの依存関係があり、その他すべてには 2 つの依存関係があるようにカウンターを初期化します。これらのカウントは、図 2-14 に示すように、各ブロックの先行する処理の数と一致します。

```
void  parallelFwdSub(std::vector<double>& x, const
                     std::vector<double>& a,
                     std::vector<double>& b) {
  const  int  N = x.size();
  const  int  num_blocks = N / block_size;

  // 参照カウントを作成
  std::vector<std::atomic<char>> ref_count(num_blocks*num_blocks);
  ref_count[0] = 0;
  for  (int  r = 1; r < num_blocks; ++r) {
    ref_count[r*num_blocks] = 1;
    for  (int  c = 1; c < r; ++c) {
      ref_count[r*num_blocks + c] = 2;
    }
    ref_count[r*num_blocks + r] = 1;
  }

  using  BlockIndex  = std::pair<size_t, size_t>;
  BlockIndex top_left(0,0);

  tbb::parallel_for_each( &top_left, &top_left+1,
    [&](const  BlockIndex& bi, tbb::feeder<BlockIndex>& f) {
     auto  [r, c] = bi;
      computeBlock(N, r, c, x, a, b);
      // 準備ができたらサクセサーを右に追加
      if  (c + 1  <= r && --ref_count[r*num_blocks + c + 1] == 0) {
       f.add(BlockIndex(r, c + 1));
      }
      // 準備ができたらサクセサーを下に追加
      if  (r + 1  < (size_t)num_blocks &&
          --ref_count[(r+1)*num_blocks + c] == 0) {
          f.add(BlockIndex(r+1, c));
      }
    }
  );
}
```

図 2-16. parallel_for_each を使用した前方置換の実装。サンプルコード: algorithms/parallel_for_each_fwd_substition.cpp

図 2-16 の parallel_for_each の呼び出しでは、最初に左上のブロック [&top_left、&top_left+1) のみを提供します。&top_left+1 は top_left 変数の 1 つ先を指し、終了イテレーターとして機能します。ただし、各ボディーの実行では、下部にある if 文によって、処理されたブロックに依存するブロックのアトミックカウンターがデクリメントされます。カウンターがゼロに達すると、そのブロックの依存関係は満たされ、フィーダーに渡されます。

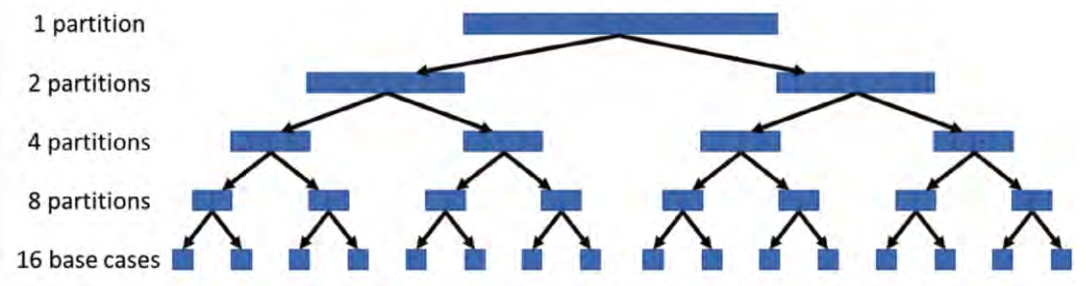
前述の素数の例と同様に、この例は parallel_for_each を使用するアプリケーションの特徴を示しています。並列性が制約されるのは、コンテナーへのシーケンシャルなアクセスや、動的にワーク項目を見つけてアルゴリズムに供給する必要があるためです。

再帰的、ツリーベースのタスクパターン

先ほど、parallel_invoke を使用して、完全に独立したタスクのセットを生成しました。しかし、図 2-4 のように、クイックソートへの 2 つの独立した呼び出しを生成すると、コアが 100 個あっても 2 倍しか改善されないことに気付きました。これは、parallel_invoke の欠点として、独立したタスクとなる各関数が明示的な引数として提供されるため、このアルゴリズムへの単一の呼び出しのスケーリングが制限されることが上げられます。100 個のコアをビジー状態に保つには、100 個の関数をリストする必要がありますが、これはあまり現実的ではありません。対照的に、parallel_for または parallel_for_each への 1 回の呼び出しで、多くのタスクが生成される場合があります。

幸いなことに、TBB はネストされた並列処理を効率的に行うため、parallel_invoke への呼び出しは、他の parallel_invoke への呼び出しの中にネストできます。実際、効率的な入れ子は一般的に TBB の特徴であり、いずれの TBB アルゴリズムも他の TBB アルゴリズムの中に効率良くネストできます。フォーク・ジョイン・パターンをネストと組み合わせることで、以前は非常に制限があるように思われた parallel_invoke を使用して、分割統治や分岐限定などスケーラビリティーの高い設計パターンを実装できます。parallel_for と parallel_for_each を使用してください。

クイックソート自体は、図 2-17(a) に示すように、分割統治アルゴリズムです。クイックソートは、ピボット値の周囲に配列を再帰的にシャッフルし、ピボット値以下の値を配列の左側のパーティションに配置し、ピボット値より大きい値を配列の右側のパーティションに配置することで機能します。再帰が基本ケースであるサイズ 1 の配列に達すると、配列全体がソートされたことになります。図 2-17(b) の下部にある再帰呼び出しは、図 2-4 で並行して行った呼び出しと同じであることに注意してください。これらは、より大きな再帰アルゴリズムの一部にすぎません。



（a）分割統治クイックソートは反復空間を再帰的に小さな部分に分割します。

```
void serialQuicksort(Iterator b, Iterator e) {
   if (b >= e) return;

   // シャッフルを行う
   double pivot_value = b->value;
   Iterator i = b, j = e-1;
   while (i != j) {
     while (i != j && pivot_value < j->value) --j;
     while (i != j && i->value <= pivot_value) ++i;
     std::iter_swap(i, j);
   }
   std::iter_swap(b, i);

   // 再帰呼び出し
   serialQuicksort(b, i);
   serialQuicksort(i+1, e);
}
```

（b）再帰クイックソートのシリアル実装。

図 2-17. クイックソートのシリアル実装。サンプルコード: `algorithms/parallel_invoke_two_quicksorts.cpp`

図 2-18 は、`serialQuicksort` への 2 つの再帰呼び出しを、再帰呼び出しでタスクを作成する `parallel_invoke` に置き換えた、クイックソートの並列実装を示しています。これら 2 つのタスクは並列タスクです。ただし、重要なことですが、`parallel_invoke` の各呼び出しはフォーク・ジョイン・パターンであるため、いくつかの同期ポイントがあります。`parallel_invoke` を呼び出すと独立したタスクが生成されますが、それらのタスクが完了するまで関数呼び出しは戻りません。幸いなことに、この動作はクイックソートのような分割統治アルゴリズムを表現するのに必要なものです。確かに、すべてのソートワークが完了する前に並列クイックソートが返されることは望ましくありません。同様に、図 2-17(a) に示すパーティションの 1 つをソートしている関数は、その子パーティションがすべてソートされるまで戻ることは望ましくありません。

図 2-18 の parallel_invoke に加えて、カットオフ値も導入します。オリジナルのシリアル・クイックソートでは、単一要素になるまで配列を再帰的に分割します。TBB タスクの生成とスケジュール設定はコストが発生するため、極端に小さなタスクは好ましくありません。並列実装のオーバーヘッドを抑えるため、要素数が 100 未満になるまで parallel_invoke を再帰的に呼び出し、その後は serialQuicksort を直接呼び出します。適切なタスクサイズについては、11 章で説明します。

```
template<typename  Iterator>
void  parallelQuickSort(Iterator  b, Iterator  e) {
 const  int  cutoff = 100;

  if  (e -  b < cutoff) {
    serialQuicksort(b, e);
  } else  {
    // シャッフルを行う
    double  pivot_value = b->value;
    Iterator i = b, j = e -  1;
    while  (i != j) {
      while  (i != j && pivot_value < j->value) --j;
      while  (i != j && i->value <= pivot_value) ++i;
      std::iter_swap(i, j);
    }
    std::iter_swap(b, i);

    // 再帰呼び出し
    tbb::parallel_invoke(
      [=]() { parallelQuickSort(b, i); },
      [=]() { parallelQuickSort(i + 1, e); }
    );
  }
}
```

図 2-18. parallel_invoke を使用したクイックソートの並列実装。サンプルコード: algorithms/parallel_invoke_recursive_quicksort.cpp

クイックソートの並列実装には大きな制限があることに気付くかもしれません。シャッフルは完全にシリアル実行されます。上位レベルでは、並列ワークを開始する前に、単一のスレッドで $O(n)$ 操作が実行されることを意味します。これにより、スピードアップが制限されます。

単一の値を計算するタスクパターン

図 2-2 に示したように、アプリケーションでよく見られる並列パターンはリダクションです。これは、マップパターンと組み合わせることが多いため、「リデュースパターン」または「マップリデュース」とも呼ばれます。リダクションでは、値のコレクションから単一の値が計算されます。アプリケーションの例として、合計、最小値、最大値の計算などがあります。アプリケーションではあまり一般的ではありませんが、スキャン（プレフィクスと呼ばれることもあります）は重要なパターンです。スキャンはリダクションに似ていますが、コレクション内の各要素（プレフィクス）の中間結果も計算します。

リダクションとスキャンはどちらも、値のコレクションから単一の最終値（スキャンの場合は中間値）を計算します。計算を並列タスクに分割するため、リダクションとスキャンを実装する TBB アルゴリズムは*結合性*に依存します。演算が結合的である場合、要素がどのようにグループ化されていても、同じ数学的結果が生成されます。例えば、整数または実数を使用する場合、`a + b + c` は `(a + b) + c` または `a + (b + c)` として計算できます。数学的には、グループ化は重要でありません。

直感的には、これは、`r = a + b + c + d` を 3 つのタスク（`t1 = a + b`、`t2 = c + d`、そして 3 番目のタスク）に分割するのと同じように、結合演算をタスクとなる独立したグループに分割できることを意味します。3 番目のタスクは、最初の 2 つのタスクが完了するまで待機し、部分的な結果を結合します（`r = t1 + t2`）。要素が 4 つ以上ある場合、要素をグループ化する方法と、保存して結合する中間値の数に関するオプションが増えます。すべてのタスクを独立して実行することはできません。中間結果を組み合わせるタスクは、中間結果が利用可能になった後にのみ実行される必要があります。しかし、それでも、並行して実行できる独立したタスクを作成することはできます。

結合性と浮動小数点演算

先に進む前に、コンピューター・システムの結合性に依存する場合に生じる複雑性について触れておく必要があります。コンピューター上で実数を正確な精度で表現するのは、必ずしも現実的でありません。代わりに、`float`、`double`、`long double` などの浮動小数点型が近似値として使用されます。これらの近似の結果、実数の演算に適用される数学的特性は浮動小数点型には適用されません。例えば、実数では加算は結合法則と可換法則に従いますが、浮動小数点数ではそのどちらでもありません。

例えば、それぞれが 1.0 に等しい N 個の実数値の合計を計算すると、結果は N になると予想されます。

```
float  r = 0.0;
for  (uint64_t  i = 0; i < N; ++i) {
   r += 1.0;
}
std: :cout << "in-order sum == "  << r << std  :: endl;
```

しかし、float 表現では有効桁数が限られているため、すべての整数値を float 値として正確に表現できるわけではありません。例えば、このループを N == 10e6（1000 万）で実行すると、出力は 10000000 になりますが、N == 20e6（2000 万）で実行すると、出力は 16777216 になります。標準の float 表現には 24 ビットの仮数部（有効桁数）があり、16777217 では 25 ビットが必要なので、変数 r は 16777217 を表現できません。16777216 に 1.0 を加えると、結果は 16777216 に切り捨てられますが、その後 1.0 を加えるたびに 16777216 に切り捨てられます。各ステップでの 16777216 の結果は 16777217 の近似値になります。こうした丸め誤差が蓄積されると、最終結果は非常に悪化します。

この合計を 2 つのループに分割し、部分的な結果を組み合わせると、どちらの場合も正しい答えが得られます（合計は 1,000 万まで正常に実行できたことを思い出してください）:

```
float  tmp1 = 0.0, tmp2 = 0.0;
for  (uint64_t  i = 0; i < N/2; ++i)
   tmp1 += 1.0;
for  (uint64_t  i = N/2; i < N; ++i)
   tmp2 += 1.0;
float  r = tmp1 + tmp2;
std  :: cout << "associative sum == "  << r << std  :: endl;
```

なぜでしょう？ r より大きな数値を表現できますが、必ずしも正確には表せません。tmp1 と tmp2 の値は同じ大きさであるため、加算により使用可能な有効桁数に影響し、2,000 万に近い結果が得られます。この例は、結合性によって浮動小数点数を使用した計算結果がどのように変化するかを示す極端な例です。

要点は、parallel_reduce と parallel_scan は結合性に依存して、部分的な結果を並列に計算して結合するということです。したがって、浮動小数点数を使用する場合、シリアル実装と比較すると異なる結果が得られる可能性があります。そして実際、参加しているスレッドの数に応じて、これらのアルゴリズムの実装では、実行ごとに異なる数の部分結果を作成されることがあります。

これらのアルゴリズムを決して使用すべきではないと結論付ける前に、浮動小数点数を使用する実装では一般に近似値になることを念頭に置く必要があります。シリアル実装でも近似値になります。同じ入力に対して異なる結果が得られたとしても、必ずしも結果の 1 つが誤っているということではありません。これは、2 つの異なる実行で丸め誤差が異なって蓄積されたことを意味します。これらの違いがアプリケーションにとって重要であるかを判断するのは、開発者次第です。

では、アルゴリズム自体に戻りましょう。

parallel_reduce

parallel_reduce は、TBB の関数テンプレートであり、結合性に依存して並列タスクを使ってリダクションを実行します。

```
namespace  tbb {

//! ラムダ対応シグネチャー
template<typename  Range, typename  Value,
         typename  Func, typename  Reduction>
Value  parallel_reduce(const  Range& range, const  Value& identity,
                       const  Func& real_body, const  Reduction& reduction,
                       partitioner, task_group_context& context);
template<typename  Range, typename  Value,
Value  parallel_reduce(const  Range& range, const  Value& identity,
                       const  Func& real_body, const  Reduction& reduction);
/* … ここに示されていない他の 2 つの同様のシグネチャー … */

//! クラス対応シグネチャー
template<typename  Range, typename  Body>
void  parallel_reduce(const  Range& range, Body& body,
                partitioner, task_group_context& context);
template<typename  Range, typename  Body>
void  parallel_reduce(const  Range& range, Body& body);
/* … ここに示されていない他の 2 つの同様のシグネチャー … */

}  // namespace tbb
```

図 2-19. oneTBB 仕様の [algorithms. parallel_reduce] セクションで説明されている parallel_reduce アルゴリズム。[algorithms.parallel_reduce] に記載されている仕様

TBB の parallel_reduce（図 2-19 を参照）には、parallel_for と同様に、Range（range）と Body（real_body）が必要です。ただし、アイデンティティー値（identity）とリダクション・ボディー（reduction）も提供する必要があります。

parallel_reduce の並列処理を実現するには、TBB ライブラリーはレンジをチャンク（データの塊）に分割し、各チャンクに real_body を適用するタスクを作成します。11 章では、パーティショナーで作成されるチャンクサイズを制御する方法について説明しますが、

現時点では、TBB がオーバーヘッドを最小限に抑えて負荷を分散するため、適切なサイズのチャンクを作成すると想定できます。real_body を実行する各タスクは、アイデンティティーで初期化された値 init から開始し、そのチャンクの部分的な結果を計算して返します。TBB ライブラリーは、リダクション関数を呼び出してこれらの部分的な結果を結合し、ループ全体に対して単一の結果を作成します。

アイデンティティー引数は、並列化される操作を使用して結合されたときに他の値を変更しない値です。例えば、加法の恒等式は“0”（x + 0 = x であるため）であり、乗法の恒等式は“1”（x * 1 = x であるため）です。 したがって、合計を並列化する場合は“0”を提供しますが、乗算を並列化する場合は“1”を提供します。リダクション関数は 2 つの部分的な結果を受け取り、それらを結合します。

図 2-20 は、レンジが 4 つのチャンクに分割される場合に、real_body およびリダクション関数を適用して 16 要素の配列から最大値を計算する方法を示しています。この例では、real_body によって配列の要素に適用される結合演算は max() であり、単位元は -∞ です（max(x,- ∞)=x であるため）。C++ では、演算として std::max を使用し、-∞ の近似値として std::numeric_limits<int>::min() を使用できます。

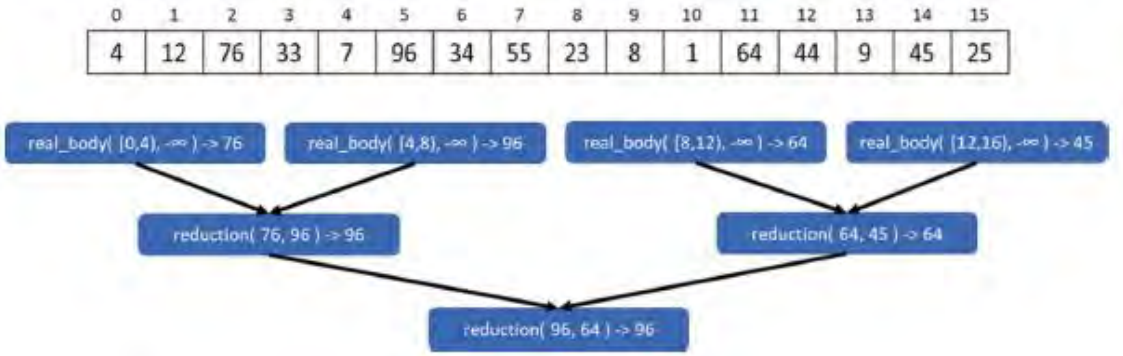


図 2-20. 最大値を計算するため real_body 関数と reduction 関数がどのように呼び出されるか

```
#include  <limits>
#include  <tbb/tbb.h>

int  simpleParallelMax(const  std::vector<int>&  v) {
 int  max_value = tbb::parallel_reduce(
    /* レンジ = */  tbb::blocked_range<int>(0, v.size()),
    /* アイデンティティー = */  std::numeric_limits<int>::min(),
    /* 関数 = */
    [&](const  tbb::blocked_range<int>&  r, int  init) -> int  {
       for  (int  i = r.begin(); i != r.end(); ++i) {
         init = std::max(init, v[i]);
       }
       return  init;
    },
    /* リダクション = */
    [](int  x, int  y) -> int  {
     return  std::max(x,y);
    }
  );
  return  max_value;
}
```

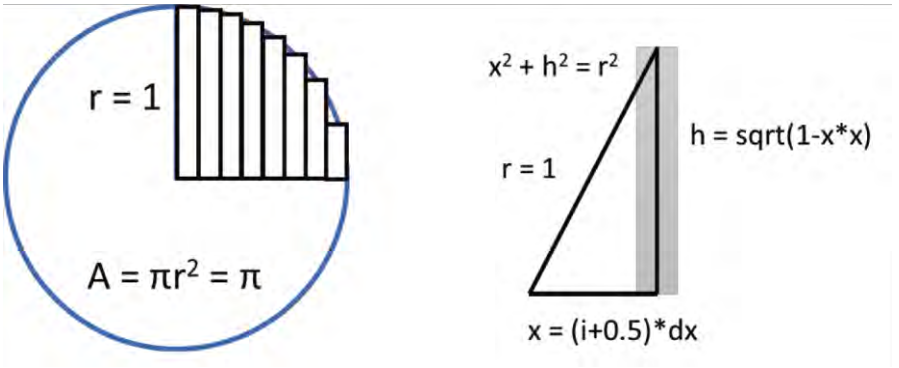
図 2-21. parallel_reduce を使用して最大値を計算します。サンプルコード: algorithms/parallel_reduce_max.cpp

図 2-21 に示すように、parallel_reduce を使用して単純な最大値ループを表現できます。

図 2-21 では、parallel_for のようにレンジの開始と終了を指定するだけでなく、レンジに blocked_range オブジェクトを使用しています。parallel_for アルゴリズムは、parallel_reduce では使用できない簡略化された構文を提供します。parallel_reduce では、レンジ・オブジェクトを直接渡しますが、幸いなことに、TBB ライブラリーによって提供される定義済みレンジの 1 つ（blocked_range、blocked_range2d、blocked_range3d など）を使用できます。他のレンジ・オブジェクトについては、11 章で詳しく説明します。

図 2-21 で使用されている blocked_range は、1D 反復空間を表します。これを構築するには、開始値と終了値を指定します。ボディーでは、begin() 関数と end() 関数を使用して、ボディーの実行に割り当てられた値のチャンクの開始値と終了値を取得し、そのサブレンジを反復処理します。図 2-7 では、レンジ内の個々の値が parallel_for ボディーに送信されるため、レンジを反復処理する i ループは必要ありません。図 2-21 では、ボディーは反復のチャンクを表す blocked_range オブジェクトを受け取るため、割り当てられたチャンク全体を反復する i ループがまだ存在します。オーバーヘッドを減らしたい場合、parallel_for でもこのレンジベースの構文を選択できます。これについては、11 章で再度取り上げます。

もう少し複雑な例を見てみましょう。図 2-22 は、数値積分によって π を計算する方法を示しています。各長方形の高さはピタゴラスの定理を使用して計算されます。単位円の 1 つの象限の面積がループ内で計算され、4 倍されて円の総面積が算出されます。これは π に等しくなります。



(a)

```
double serialPI(int num_intervals) {
  double dx = 1.0 / num_intervals;
  double sum = 0.0;
  for (int i = 0; i < num_intervals; ++i) {
    double x = (i+0.5)*dx;
    double h = std::sqrt(1-x*x);
    sum += h*dx;
  }
  double pi = 4 * sum;
  return pi;
}
```

(b)

図 2-22. 長方形近似を用いたシリアルの π 計算。サンプルコード: `algorithms/parallel_reduce_pi.cpp`

図 2-22(b) のコードは、すべての長方形の面積の合計を計算するリダクション演算です。TBB の `parallel_reduce` を使用するには、レンジ、`real_body`、アイデンティティー、リダクションを識別する必要があります。この例では、レンジは `[0, num_intervals)` であり、`real_body` は図 2-22(b) の `i` ループのようになります。合計を実行しているため、アイデンティティー値は `0.0` です。

そして、部分的な結果を結合するリダクション・ボディーは、2 つの値の合計を返します。TBB の parallel_reduce を使用した並列実装を図 2-23 に示します。

```
#include  <cmath>
#include  <tbb/tbb.h>
//
// TBB parallel_reduce による数値積分を用いた
// 円周率の推定
//
double  parallelPI(int  num_intervals) {
   double  dx = 1.0  / num_intervals;
   double  sum = tbb::parallel_reduce(
     /* レンジ = */  tbb::blocked_range<int>(0, num_intervals),
     /* アイデンティティー = */0.0,
     /* 関数 */
     [=](const  tbb::blocked_range<int>&  r, double  init) -> double  {
      for  (int  i = r.begin(); i != r.end(); ++i) {
         double  x = (i + 0.5)*dx;
         double  h = std::sqrt(1  -  x*x);
         init += h*dx;
       }
       return  init;
     },
     /* リダクション */
     [](double  x, double  y) -> double  {
      return  x + y;
     }
   );
   double  pi = 4  * sum;
   return  pi;
}
```

図 2-23. tbb::parallel_reduce を使用した pi の実装。サンプルコード: algorithms/parallel_reduce_pi.cpp

parallel_for と同様に、parallel_reduce ではパフォーマンスを調整し、丸め誤差を管理する高度な機能とオプションがあります（「結合性と浮動小数点演算」を参照）。

parallel_deterministic_reduce

parallel_deterministic_reduce は、TBB の関数テンプレートであり、parallel_reduce と同様に結合性に依存して並列タスクでリダクションを実行し、同じマシンで実行された場合、同じ入力データでの各実行で同じ結果が得られることを保証します。これによりパフォーマンスがわずかに低下する場合がありますが、実際には無視できる程度です。この関数は、決定論的な結果が必要な場合にのみ、parallel_reduce の代わりに使用されます。多くの場合、これはテスト用に使用されます。プログラムをデバッグする際、特に役立ちます!

決定論的なスケジューラーの制限

前述したように、浮動小数点数の実装は近似値です。つまり、結合性や可換性などの特性に依存する場合、並列処理によって異なる結果が生じる可能性があるということです。生成されるさまざまな結果は、必ずしも誤っているわけではなく、単に異なるだけです。TBB は、同じマシンで実行したときに、同じ入力データの各実行で同じ結果が得られるよう、`parallel_deterministic_reduce` アルゴリズムを提供します。

`parallel_deterministic_reduce` は、`simple_partitioner` または `static_partitioner` のみを受け入れます。これは、両方のパーティショナー・タイプでサブレンジ数が決定論的であるためです。パーティショナーのタイプについては、11 章で詳しく説明します。また、`parallel_deterministic_reduce` は、実行に動的に参加するスレッドの数や、タスクがスレッドにどのようにマップされるかに関係なく、特定のマシン上で常に同じ分割操作と結合操作のセットを実行しますが、`parallel_reduce` アルゴリズムではそうならない場合があります。その結果、`parallel_deterministic_reduce` は同じマシンで実行すると常に同じ結果を返しますが、それにはある程度の柔軟性が犠牲になります。

`parallel_deterministic_reduce` ではすべての分割と結合を実行する必要があるため、オーバーヘッドが発生しますが、このオーバーヘッドは通常は小さいものです。それより大きな制限は、`auto_partitioner` や `affinity_partitioner` のようにチャンクサイズを自動的に検出するパーティショナーを使用できないことです。

parallel_scan: 中間値によるリダクション

アプリケーションではあまり一般的ではありませんが、それでもスキャン（プレフィクスと呼ばれることもあります）は重要なパターンです。スキャンはリダクションに似ていますが、値のコレクションから単一の値を計算するだけでなく、レンジ（*プレフィクス*）内の各要素の中間結果も計算します。例としては、値 `x0`、`x1`、`...` `xN` の累計が挙げられます。結果には、累積合計、`y0`、`y1`、`...` `yN`、および最終合計 `yN` の各値が含まれます。

一見すると、このシリーズの値は、並列処理するには独立性がないように見えます:

```
y0 = x0
y1 = x0+ x1
…
yN = x0+ x1 + … + xN
```

ベクトル v から累計を計算するシリアルループは、図 2-24 のようになります。

```
int  serialImpl(const  std::vector<int> &v, std::vector<int> &rsum) {
  int  N = v.size();
  rsum[0] = v[0];
  for  (int  i = 1; i < N; ++i) {
      rsum[i] = rsum[i-1] + v[i];
  }
  int  final_sum = rsum[N-1];
  return  final_sum;
}
```

図 2-24. ベクトル v から累計を計算するシリアルループ。サンプルコード: algorithms/serial_running_sum.cpp

これは、スキャンはシリアル・アルゴリズムのように見えます。各プレフィクスは、以前のすべての反復で計算された結果に依存します。驚くかもしれませんが、この一見シリアルなアルゴリズムには、効率良い並列実装が存在します。TBB の parallel_scan アルゴリズムは、効率良い並列スキャンを実装します。そのインターフェイスでは、図 2-25 に示すように、レンジ、アイデンティティー値、スキャンボディー、結合ボディーを提供する必要があります。

```
template<typename  Range, typename  Value, typename  Scan,
         typename  Combine>
Value  parallel_scan(const  Range& range, const  Value& identity,
                     const  Scan& scan, const  Combine& combine);
```

図 2-25. oneTBB 仕様の [algorithms. parallel_scan] セクションで説明されている parallel_scan アルゴリズム。[algorithms.parallel_scan] に記載されている仕様

レンジ、アイデンティティー値、および結合ボディーは、parallel_reduce のレンジ、アイデンティティー値、およびリダクション・ボディーに類似しています。また、他のループ・アルゴリズムと同様に、レンジは TBB ライブラリーによってチャンクに分割され、チャンクにボディー（スキャン）を適用する TBB タスクが作成されます。parallel_scan インターフェイスの詳しい説明は [algorithms.parallel_scan] に記載されています。

parallel_scan で異なるのは、スキャンボディーが同じ反復チャンクで複数回実行される可能性があることです。最初は*事前スキャン*モードで実行され、その後、*最終スキャン*モードで実行されます。

*最終スキャン*モードでは、そのサブレンジの直前の反復の正確なプレフィクスの結果がボディーに渡されます。この値を使用して、ボディーはサブレンジ内の各反復のプレフィクスを計算して保存し、サブレンジ内の最後の要素の正確なプレフィクスを返します。

ただし、スキャンボディーが*事前スキャン*モードで実行されると、指定されたレンジの前にある要素の最終値ではない開始プレフィックス値を受け取ります。parallel_reduce と同様に、parallel_scan も結合性に依存します。事前スキャンモードでは、開始プレフィクス値は、その前にあるサブレンジを表す場合がありますが、その前にある完全なレンジを表すとは限りません。この値を使用して、サブレンジ内の最後の要素の（最終ではない）プレフィクスを返します。返される値は、開始プレフィクスとそのサブレンジを組み合わせた部分的な結果を表します。これらの*事前スキャン*モードと*最終スキャン*モードを使用すると、スキャン・アルゴリズムで並列処理を活用できます。

どのような仕組みか ？

累積合計の例を振り返って、3 つのチャンク A、B、C で計算することを考えてみましょう。順次実装では、A、B、C のすべてのプレフィクスを計算します（3 つのステップが順番に実行されます）。図 2-26 に示すように、並列スキャンを使用するとさらに効果的です。

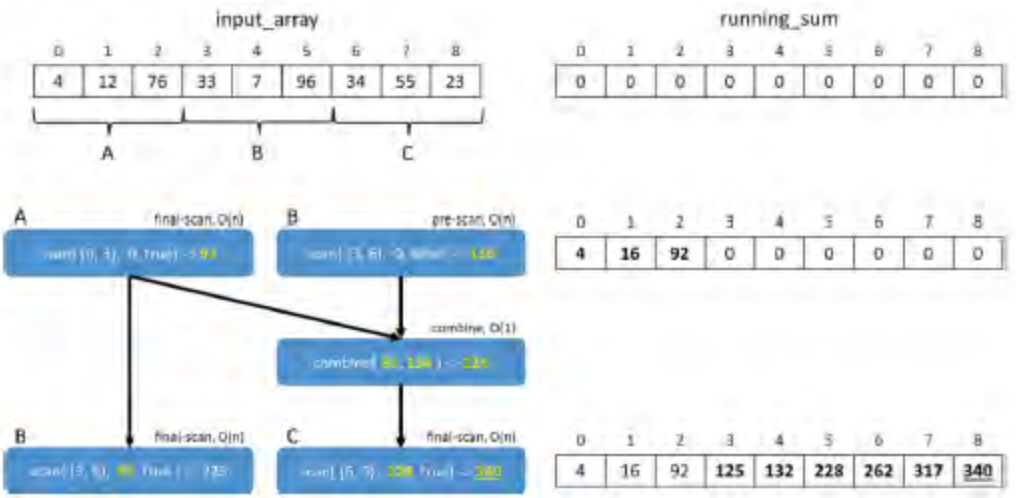


図 2-26. 合計を計算するため並列スキャンを実行

まず、A のスキャンを最終スキャンモードで計算します。これは、A が最初の値セットであり、初期値としてアイデンティティーが渡された場合、そのプレフィクス値が正確になるためです。A を起動すると同時に、B を事前スキャンモードで起動します。これら 2 つのスキャンが完了すると、B と C の両方の正確な開始プレフィクスを計算できるようになります。B には A の最終結果（92）を提供し、C には A の最終スキャン結果と B の事前スキャン結果の組み合わせ（92+136 = 228）を提供します。

結合操作には一定の時間がかかりますが、スキャン操作よりもはるかにコストは低くなります。3 つの大きなステップを順番に実行する順次実装とは異なり、並列実装では、A の最終スキャンと B の事前スキャンを並列で実行し、次に一定時間の結合ステップを実行し、最後に B と C の最終スキャンを並列で計算します。少なくとも 2 つのコアがあり、N が十分に大きい場合、3 つのチャンクを使用する並列プレフィクス合計は、順次実装の約 3 分の 2 の時間で計算できます。もちろん、`parallel_prefix` は、さらに多くのコアを活用するため 3 つ以上のチャンクで実行することもできます。

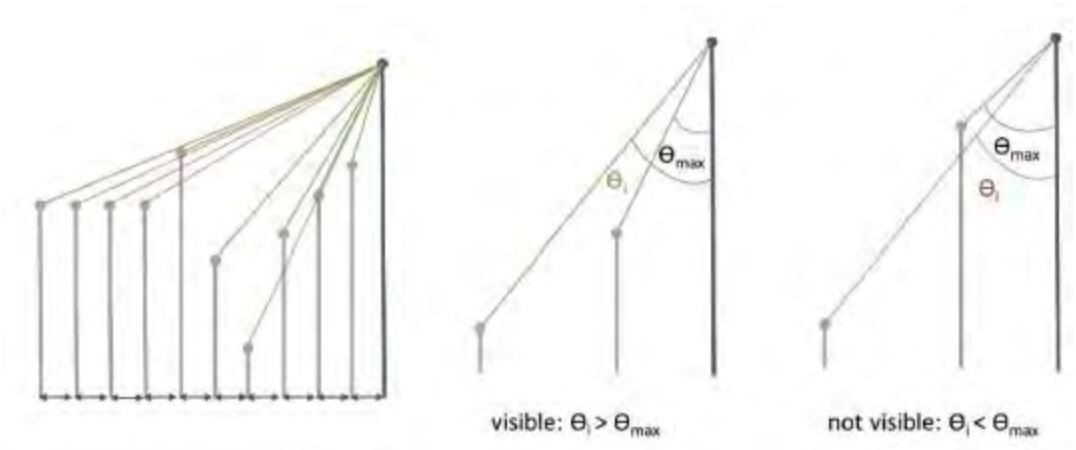
図 2-27 は、TBB の `parallel_scan` を使用した単純な部分和の例の実装を示しています。レンジは区間 `[1, N)`、アイデンティティー値は `0` であり、結合関数は 2 つの引数の合計を返します。スキャンボディーは、受け取った初期合計に加算された、そのサブレンジ内のすべての値の部分合計を返します。ただし、`is_final_scan` 引数が `true` の場合にのみ、プレフィクスの結果が `running_sum` 配列に割り当てられます。

```
int  simpleParallelRunningSum(const  std::vector<int>&  v,
                              std::vector<int>&  rsum) {
  int  N = v.size();
  rsum[0] = v[0];
  int  final_sum = tbb::parallel_scan(
    /* range = */  tbb::blocked_range<int>(1, N),
    /* identity = */  (int)0,
    /* scan ボディー */
    [&v, &rsum](const  tbb::blocked_range<int>&  r,
                int  sum, bool  is_final_scan) -> int  {
      for  (int  i = r.begin(); i < r.end(); ++i) {
        sum += v[i];
        if  (is_final_scan)
          rsum[i] = sum;
      }
      return  sum;
    },
     /* combine ボディー
                    */
    [](int  x, int  y) {
      return  x + y;
    }
  );
  return  final_sum;
}
```

図 2-27. parallel_scan を使用した合計実行の実装。サンプルコード: algorithms/parallel_scan_running_sum.cpp

少し複雑なスキャンの例: 視線

図 2-28 は、『Vector Models for Data-Parallel Computing』(Guy E. Blelloch、The MIT Press) で説明されているものと同様の視線問題のシリアル実装を示しています。視点の高さと、視点から一定間隔にあるポイントの高さが与えられると、視線コードは視点から見えるポイントを決定します。図 2-28(a) に示すように、あるポイントと視点 `altitude[0]` の間にあるいずれかのポイントの角度 θ の方が大きい場合、そのポイントは見えません。シリアル実装では、スキャンを実行して、特定のポイントと視点の間のすべてのポイントの最大 θ 値を計算します。指定されたポイントの θ 値がこの最大角度より大きい場合、そのポイントは可視ポイントになります。それ以外は、可視ポイントになりません。



(a)視点から見えるポイントを計算。

```
void  serialLineOfSight(const  std::vector<double>& altitude,
                        std::vector<bool>& is_visible, double  dx) {
  const  int  N = altitude.size();

  double  max_angle = std::atan2(dx, altitude[0] -  altitude[1]);
  double  my_angle = 0.0;

  for  (int  i = 2; i < N; ++i ) {
    my_angle = std::atan2(i * dx, altitude[0] -  altitude[i]);
    if  (my_angle >= max_angle) {
      max_angle = my_angle; } else  {
      is_visible[i] = false;
    }
  }
}
```

(b)シリアル実装。

図 2-28. 視線の例。サンプルコード: algorithms/parallel_scan_line_of_sight.cpp

図 2-29 は、TBB parallel_scan を使用する視線の例の並列実装を示しています。アルゴリズムが完了すると、is_visible 配列に各ポイントの可視性（true または false）が含まれます。このコードでは、ポイントの可視性を判断するため、各ポイントの最大角度を計算する必要がありますが、最終的な出力は各ポイントの最大角度ではなく、各ポイントの可視性であることが重要です。max_angle は必要ですが最終結果ではないため、事前スキャンモードと最終スキャンモードの両方で計算されますが、最終スキャンの実行中は各ポイントに対して is_visible 値のみが保存されます。

```
void parallelLineOfSight(const std::vector<double>& altitude,
std::vector<bool>& is_visible, double dx) {
  const int N = altitude.size();
  double max_angle = std::atan2(dx, altitude[0] - altitude[1]);

  double final_max_angle = tbb::parallel_scan(
    /* range = */ tbb::blocked_range<int>(1, N),
    /* identity */ 0.0,
    /* scan body */
    [&altitude, &is_visible, dx](const tbb::blocked_range<int>& r,
                                 double max_angle,
                                 bool is_final_scan) -> double {
     for (int i = r.begin(); i != r.end(); ++i) {
        double my_angle = atan2(i*dx, altitude[0] - altitude[i]);
        if (my_angle >= max_angle)
          max_angle = my_angle;
        if (is_final_scan && my_angle < max_angle)
          is_visible[i] = false;
      }
      return max_angle;
    },
    [](double a, double b) -> double {
     return std::max(a,b);
    }
  );
}
```

図 2-29. parallel_scan を使用した視線問題の実装。サンプルコード: algorithms/parallel_scan_line_of_sight.cpp

パイプラインのパターン

パイプラインは、通過する*項目*を変換する*フィルター*の線形シーケンスです。パイプラインは、ビデオフレームやオーディオフレーム、財務データなど、アプリケーションに流れ込むデータを処理するために使用されます。4 章では、ファンインフィルターとファンアウト・フィルターを含む複雑なグラフを構築できるフローグラフ・インターフェイスについて説明します。この節では、フィルターの線形シーケンスのみを扱う parallel_pipeline について説明します。

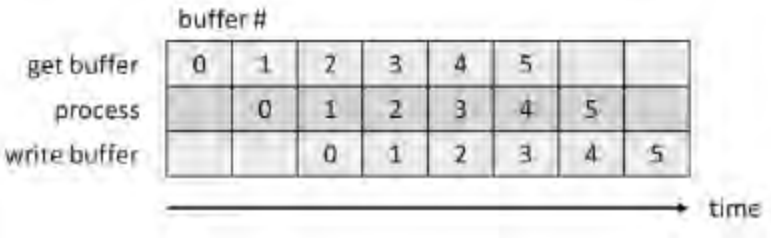
図 2-30 は、文字配列を読み取り、すべての小文字を大文字に、すべての大文字を小文字に変換して、結果を順番に出力ファイルに書き込むループの例を示しています。

```
#include  <algorithm>
#include  <cctype>
#include  <fstream>
#include  <iostream>
#include  <memory>
#include  <string>
#include  <tbb/tbb.h>

using  CaseStringPtr  = std::shared_ptr<std::string>;
CaseStringPtr  getCaseString(std::ofstream&  f);
void  writeCaseString(std::ofstream&  f, CaseStringPtr  s);

void  serialChangeCase(std::ofstream&  caseBeforeFile,
                       std::ofstream&  caseAfterFile) {
  while  (CaseStringPtr s_ptr = getCaseString(caseBeforeFile))
    { std::transform(s_ptr->begin(), s_ptr->end(), s_ptr->begin(),
      [](char  c) -> char  {
       if  (std::islower(c))
          return  std::toupper(c);
        else  if  (std::isupper(c))
          return  std::tolower(c);
        else
          return  c;
      }
    );
    writeCaseString(caseAfterFile, s_ptr);
  }
}
```

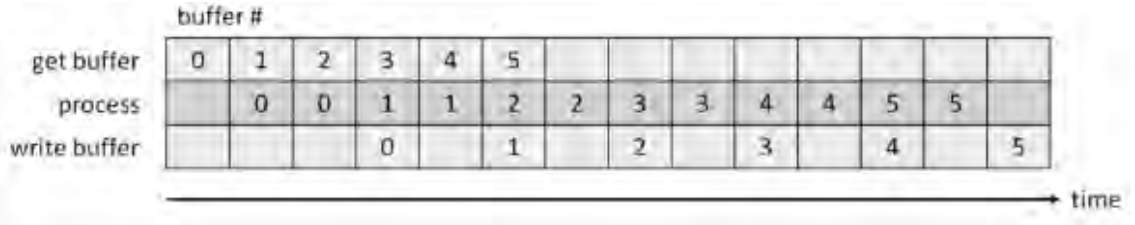
図 2-30. シリアル文字変更の例。サンプルコード: algorithms/parallel_pipeline_case.cpp

操作は各バッファーに対して順番に実行する必要がありますが、異なるバッファーに適用された異なるフィルターの実行をオーバーラップできます。図 2-31(a) は、この例をパイプラインとして示しており、“write buffer” が bufferi に対して動作し、並行して “process” フィルターが bufferi+1 に対して動作し、“get buffer” フィルターが bufferi+2 を読み取ります。

（a）異なるフィルターによって並列に処理される 3 つのバッファー。



（b）すべてのフィルターが一度に 1 つの項目に対して動作できる定常状態の動作。



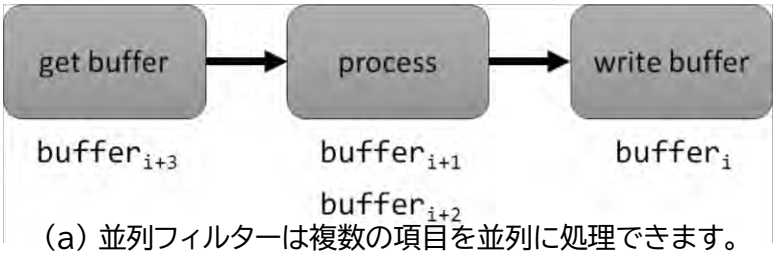
（c）プロセスに他のフィルターの 2 倍の時間がかかる定常状態の動作。

図 2-31. パイプラインを使用した文字変更の例

図 2-31(b) に示すように、定常状態では各フィルターがビジー状態にあり、その実行が重複しています。ただし、図 2-31(c) に示すように、アンバランス・フィルターでは速度向上効果が低下します。シリアルフィルターのパイプラインのパフォーマンスは、最も低速なシリアルステージによって制限されます。

TBB ライブラリーは、シリアルフィルターと並列フィルターの両方をサポートします。並列フィルターを異なる項目に並列に適用して、フィルターのスループットを向上できます。図 2-32(a) は、2 つの項目に対して並列に実行される中間/プロセスフィルターを使用した「文字変更」の例を示しています。図 2-32(b) は、中央のフィルターが特定の項目を処理するのに他のフィルターの 2 倍の時間がかかる場合、このフィルターに 2 つのスレッドを割り当てることで、他のフィルターのスループットに一致できることを示しています。

図 2-33 は、TBB 並列パイプラインに関連付けられている型と関数を示しています。



（a）並列フィルターは複数の項目を並列に処理できます。

（b）並列フィルターにさらに多くのスレッドを使用してスループットを向上できます。

図 2-32. 並列フィルターを備えたパイプラインを使用した文字変更の例。並列フィルターの 2 つのコピーを使用することで、パイプラインはスループットを最大化します

```
namespace tbb {
  enum class filter_mode {
    parallel = /* 実装定義 */,
    serial_in_order = /* 実装定義 */,
    serial_out_of_order = /* 実装定義 */
  };

  //! 型セーフなパイプライン・フィルターのチェーンを表すクラス
  template<typename Input_Type, typename OutputType>
  class filter;

  //! parallel_pipeline に参加するフィルターを作成
  template<typename Body> filter<filter_input<Body>, filter_output<Body>>
  make_filter( filter_mode mode, const Body& body );

  //! 左右のフィルターの構成
  template<typename T, typename V, typename U> filter<T, U>
  operator&( const filter<T,V>& left, const filter<V,U>& right );

  //! ユーザー指定のコンテキストを持つフィルターのチェーン上の並列パイプライン
  inline void
  parallel_pipeline(size_t max_number_of_live_tokens,
                    const filter<void, void>& filter_chain,
                    task_group_context& context);

  //! フィルターのチェーンにわたる並列パイプライン
  inline void
  parallel_pipeline(size_t max_number_of_live_tokens,
                    const filter<void, void>& filter_chain);

  //! フィルターのシーケンスにわたる並列パイプライン
  template<typename F1, typename F2, typename ...FiltersContext>
  void parallel_pipeline(size_t max_number_of_live_tokens,
                         const F1& filter1,
                         const F2& filter2,
                         FiltersContext&& ... filters);
} // namespace tbb
```

図 2-33. oneTBB 仕様の [algorithms. parallel_pipeline] セクションで説明されている parallel_pipeline アルゴリズム。[algorithms.parallel_pipeline] に記載されている仕様

図 2-33 で強調表示されている parallel_pipeline 関数の最初の引数は max_number_of_live_tokens です。この引数は、特定の時点でパイプラインを通過できる項目の最大数を設定するのに使用されます。この値はリソースの消費を制限するために必要です。例えば、単純な 3 つのフィルターのパイプラインを考えてみます。中間フィルターがシリアルフィルターで、新しいバッファーを取得するフィルターよりも 1,000 倍の時間がかかるとしたらどうなるでしょうか。最初のフィルターは、2 番目のフィルターの前にキューに投入するためだけに 1,000 個のバッファーを割り当てる可能性があり、大量のメモリーが無駄になります。

中間フィルターがシリアルフィルターで、新しいバッファーを取得するフィルターよりも 1,000 倍の時間がかかるとしたらどうなるでしょうか。最初のフィルターは、2 番目のフィルターの前にキューに投入するためだけに 1,000 個のバッファーを割り当てる可能性があり、大量のメモリーが無駄になります。

`parallel_pipeline` の 2 番目の引数は `filter_chain` で、これは `make_filter` 関数で作成されたフィルターを `operator&` 関数を呼び出して連結することで作られる一連のフィルターです。これら 2 つの関数は図 2-33 でも強調表示されています。

`make_filter` テンプレート関数は、`filter_mode` と `Body` の 2 つの引数を受け取ります。作成されたフィルターの入力/出力タイプは、`Body` の入力/出力タイプから設定されます。`filter_mode` 引数は、`serial_in_order`、`serial_out_of_order`、または `parallel` です。図 2-34 は、TBB の `parallel_pipeline` を使用した大小文字変更例の実装を示しています。

```
void  parallelChangeCase(std::ofstream& caseBeforeFile,
                         std::ofstream& caseAfterFile) {
  int  num_tokens = tbb::info::default_concurrency();
  tbb::parallel_pipeline(
    /* トークン s */  num_tokens,
    /* 取得フィルター */
    tbb::make_filter<void, CaseStringPtr>(
      /* フィルターノード */  tbb::filter_mode::serial_in_order,
      /* フィルターボディー */
      [&](tbb::flow_control& fc) -> CaseStringPtr  {
        CaseStringPtr s_ptr = getCaseString(caseBeforeFile);
        if  (!s_ptr)
          fc.stop();
        return  s_ptr;
    }) &  // 連結演算
    /* 変更ケースフィルターを作成 */
    tbb::make_filter<CaseStringPtr, CaseStringPtr>(
    /* フィルターノード */  tbb::filter_mode::parallel,
    /* filter body */
      [](CaseStringPtr  s_ptr) -> CaseStringPtr  {
       std::transform(s_ptr->begin(), s_ptr->end(), s_ptr->begin(),
           [](char  c) -> char  {
             if  (std::islower(c))
              return  std::toupper(c);
            else  if
             (std::isupper(c))
              return  std::tolower(c);
             else
               return  c;
           });
        return  s_ptr;
      }) &  // 連結演算
    /* 書き込みフィルターを作成 */
    tbb::make_filter<CaseStringPtr, void>(
      /* フィルターノード */  tbb::filter_mode::serial_in_order,
      /* フィルターボディー */
      [&](CaseStringPtr  s_ptr) -> void  {
        writeCaseString(caseAfterFile, s_ptr);
      })
  );
}
```

図 2-34. TBB の parallel_pipeline、2 つの serial_in_order フィルター、および中央の parallel フィルターを使用して実装された大小文字変更の例。サンプルコード: algorithms/parallel_pipeline_case.cpp

最初のフィルターは、tbb::flow_control 型の特別な引数を受け取ることが分かります。この引数は、パイプラインの最初のフィルターが新しいアイテムを生成しないことを通知するために使用します。例えば、図 2-34 の最初のフィルターでは、getCaseString() が返すポインターが NULL のときに、fc.stop() を呼び出しています。

この実装では、最初のフィルターと最後のフィルターは serial_in_order モードで作成されます。これは、両方のフィルターが一度に 1 つの項目に対してのみ実行され、最後のフィルターが最初のフィルターによって生成された順序と同じ順序で項目を実行することを指定します。serial_out_of_order フィルターでは、任意の順序でアイテムを実行できます。中間フィルターには、モードとして parallel が渡され、異なる項目に対して並行して実行することができます。

複雑なパイプラインの例を図 2-35 に示します。while ループがフレーム番号を読み込み、フレームごとに左右の画像を読み込み、左の画像に赤、右の画像に青の着色を加えます。次に、結果の 2 つの画像を 1 つのレッドシアンの 3D 立体画像に結合します。

（a）2 つの画像を組み合わせてレッドシアンの 3D 立体画像を作成します。オリジナルの写真は Elena Adams が撮影しました。

```
class
PNGImage  {
public:
  uint64_t  frameNumber = -1;
  unsigned  int  width = 0, height = 0;
  std::shared_ptr<std::vector<unsigned  char>> buffer;
  static  const  int  numChannels = 4;
  static  const  int  redOffset = 0;
  static  const  int  greenOffset = 1;
  static  const  int  blueOffset = 2;

  PNGImage() {}
  PNGImage(uint64_t  frame_number, const  std::string& file_name);
  PNGImage(const  PNGImage& p);
  virtual  ~PNGImage() {} void  write() const;
};

int  getNextFrameNumber();
PNGImage  getLeftImage(uint64_t  frameNumber);
PNGImage  getRightImage(uint64_t  frameNumber);
void  increasePNGChannel(PNGImage& image, int  channel_offset, int
increase);
void  mergePNGImages(PNGImage& right, const  PNGImage& left);

void  serial3DStereo() {
  while  (uint64_t  frameNumber = getNextFrameNumber()) {
    auto  left = getLeftImage(frameNumber);
    auto  right = getRightImage(frameNumber);
    increasePNGChannel(left, PNGImage::redOffset, 10);
    increasePNGChannel(right, PNGImage::blueOffset, 10);
    mergePNGImages(right, left);
    right.write();
  }
}
```

（b）3D 効果を適用するシリアル実装。

図 2-35. 3D 効果を適用するシリアル処理の例。サンプルコード: algorithms/3Dstereo_serial_no_pipeline.cpp

単純な大小文字変更のサンプルと同様に、ここでも一連のフィルターを通過する一連の入力があります。重要な関数を特定し、それらをパイプライン・フィルターに変換します: `getNextFrameNumber`、`getLeftImage`、`getRightImage`、`increasePNGChannel`（左の画像へ）、`increasePNGChannel`（右の画像へ）、`mergePNGImages`、`right.write()`。図 2-36 はパイプラインとして描かれた例です。`increasePNGChannel` フィルターは、最初に左側の画像に適用され、次に右側の画像に適用されます。

図 2-36. パイプラインとしての 3D 立体視サンプル・アプリケーション

TBB の `parallel_pipeline` を使用した並列実装を図 2-37 に示します。

```
void  parallel3DStereo() {
  using  Image  = PNGImage;
  using  ImagePair  = std::pair<PNGImage, PNGImage>;
  tbb::parallel_pipeline(
    /* トークン */  8,
    /* 左の画像フィルターを作成 */
    tbb::make_filter<void, Image>(
      /* フィルタータイプ */
      tbb::filter_mode::serial_in_order,
      [&](tbb::flow_control& fc) -> Image  {
        if  (uint64_t  frame_number = getNextFrameNumber()) {
          return  getLeftImage(frame_number);
        } else  {
          fc.stop();
          return  Image{};
        }
      }) &
    tbb::make_filter<Image, ImagePair>(
      /* filter タイプ */  tbb::filter_mode::serial_in_order,
      [&](Image  left) -> ImagePair  {
        return  ImagePair(left, getRightImage(left.frameNumber));
      }) &
    tbb::make_filter<ImagePair, ImagePair>(
      /* filter タイプ */  tbb::filter_mode::parallel,
      [&](ImagePair  p) -> ImagePair  {
        increasePNGChannel(p.first, Image::redOffset,
        10); return  p;
      }) &
    tbb::make_filter<ImagePair, ImagePair>(
      /* filter タイプ */  tbb::filter_mode::parallel,
      [&](ImagePair  p) -> ImagePair  {
        increasePNGChannel(p.second, Image::blueOffset,
        10); return  p;
      }) &
    tbb::make_filter<ImagePair, Image>(
      /* filter type */  tbb::filter_mode::parallel,
      [&](ImagePair  p) -> Image  {
        mergePNGImages(p.second,
        p.first); return  p.second;
      }) &
    tbb::make_filter<Image, void>(
      /* filter タイプ */  tbb::filter_mode::parallel,
      [&](Image  img) {
        img.write();
      })
  );
}
```

図 2-37. parallel_pipeline を使用した 3D 立体サンプル。サンプルコード: algorithms/3Dstereo_parallel_pipeline.cpp

TBB の parallel_pipeline 関数は、パイプライン・フィルターの線形化を行います。最初のステージからの入力がパイプラインを通過すると、フィルターが次々に適用されます。これは、parallel_pipeline の制限により、このサンプルに課せられた必要のない制約です。左画像処理と右画像処理は mergeImageBuffers フィルターまでは独立していますが、parallel_pipeline のインターフェイスであるため、フィルターを線形化する必要があります。それでも、画像を読み込むフィルターだけがシリアルフィルターであるため、実行時間が後続の並列ステージによって支配される場合でも、この実装はスケーラブルになります。

4 章では、フィルターの非線形実行の恩恵を受けるアプリケーションを直接的に表現できる TBB フローグラフを紹介します。

parallel_sort

前の節では、実装に役立つ主要なパターンの観点から、8 つの TBB アルゴリズムのうち 7 つについて説明しました。最後のアルゴリズムであるソートは便利ですが、単なるアルゴリズムであり、構成要素ではないため、意味が異なります。

parallel_sort は、シーケンスまたはコンテナーを可能であれば並列にソートします。ソートは不安定で再現性がありません。等しいキーの相対的な順序は保持されず、同じシーケンスを再度ソートした場合でも同じ結果になる保証はありません。イテレーターとシーケンスの要件は、std::sort と同じです。

parallel_sort には 3 つの呼び出しタイプがあります:

1. parallel_sort(begin,end,comp) を呼び出すと、相対的な順序を決定するため、引数 comp を使用してシーケンス [begin, end) をソートします。comp(x,y) が true を返す場合、x はソートされたシーケンスで y の前に現れます。

2. parallel_sort(begin, end) 呼び出しは、parallel_sort (begin,end,std::less<T>) と同等です。

3. parallel_sort(c[,comp]) 呼び出しは、parallel_ sort(std::begin(c),std::end(c)[,comp]) と同様です。

図 2-38 は parallel_sort を使用する簡単な例を示しています。

```
#include  <cstdio>
#include  <tbb/tbb.h>
#include  <array>

#define  PV(X) printf(" %02d", X);
#define  PN(Y) printf( "\nHello, Sorted "  #Y  ":\t");
#define  P(N) PN(N); std::for_each(N.begin(),N.end(),[](int x) { PV(x); });
#define  V(Z) myvect.push_back(Z);

int  main( int  argc, char  *argv[] ) {
  int  myvalues[]              = { 3, 9, 4, 5, 1, 7, 6, 8, 10, 2  };
  std::array<int, 10> myarray  = { 19, 13, 14, 11, 15, 20, 17, 16, 12, 18  };
  std::array<int, 10> disarray = { 23, 29, 27, 25, 30, 21, 26, 24, 28, 22  };
  tbb::concurrent_vector<int> myvect;
  V(40); V(31); V(37); V(33); V(34); V(32); V(34); V(35); V(38); V(36);

  tbb::parallel_sort( myvalues,myvalues+10  );
  tbb::parallel_sort( myarray.begin(), myarray.end() );
  tbb::parallel_sort( disarray );
  tbb::parallel_sort( myvect );

  PN(myvalues); for(int  i=0;i<10;i++) PV(myvalues[i]);
  P(myarray);
  P(disarray);
  P(myvect);
  printf("\n\n");
  return  0;
}
```

この出力は決定論的です。

```
Hello, Sorted myvalues:  01 02 03 04 05 06 07 08 09 10

Hello, Sorted myarray:   11 12 13 14 15 16 17 18 19 20

Hello, Sorted disarray:  21 22 23 24 25 26 27 28 29 30

Hello, Sorted myvect:    31 32 33 34 34 35 36 37 38 40
```

図 2-38. parallel_sort の例。サンプルコード: algorithms/parallel_sort.cpp

その他のアルゴリズム、パターン、機能

この章では、すべての TBB アルゴリズムについて説明しましたが、図 2-2 で紹介した並列パターンのすべてをカバーしているわけではありません。

イベントベースの調整並列パターンは、TBB のフローグラフ API によって適切にカバーされています。フローグラフは非常に大きなトピックであるため、4 章でフローグラフを紹介し、5 章で活用方法を説明します。

TBB の task_group は、タスクを直接的に作成および管理する、一般的で、幅広く適用可能な方法です。図 2-2 の並列パターンの多くは、task_group をベースに何らかの方法で表現できます。task_group については、6 章で詳しく説明します。

まとめ

この章では、TBB ライブラリーによって提供される一般的な並列アルゴリズムの概要を示し、それらを使用して並列パターンを実装する方法について説明しました。これらのパッケージ化されたアルゴリズムは、十分にテストされ調整された実装を提供し、アプリケーションに段階的に適用してパフォーマンスを向上できます。

この章で示したコードは、これらのアルゴリズムの使用方法を示す例です。以降の章では、これらのアルゴリズムを構成可能な方法で組み合わせ、局所性の最適化、オーバーヘッドの最小化、優先順位の追加に使用できるライブラリー機能を使用してアプリケーションを調整することにより、TBB を最大限に活用します。

次の章（3 章）では、タスクの独立性に加えてデータの独立性も重要であるため、並行コンテナーについて検討します。次に、4 章から 6 章でフローグラフとタスクグループについて説明して、並列パターンの説明を終了します。

3 章　並行性のためのデータ構造

並行タスク間のデータ共有が適切に管理されないと、正当性とパフォーマンスに重大な問題が生じる可能性があります。TBB は、長年にわたって信頼できるツールとして受け入れられ、十分にテストされたオープンソース・ソリューションを提供します。TBB コンテナーは、単独で使用することも、他の TBB コンポーネントと連携して使用することもできます。

並列に読み取り操作を実行する場合、並行コンテナーは必要でないことに注意してください。特別なサポートは、並列コードがコンテナーを変更する場合にのみ必要です。新しいキーが導入されない、頻度の低い変更であれば、標準 C++ ライブラリーで十分です。ただし、開発者は C++ リファレンス（https://en.cppreference.com/w/cpp/container）に記載されているスレッドのセーフティー・ガイドラインをよく理解しておく必要があります。

新しい要素の追加など、コレクションが動的に拡大するシナリオでは、この章で説明する並列対応コンテナーは不可欠です。このような場合に TBB コンテナーを採用する理由は、パフォーマンスの向上です。

C++ 標準テンプレート・ライブラリー（STL）コンテナーでは、コンテナーの構造を変更する同時更新はスレッドセーフではないため、同時実行性が制限されます。適切な同期なしで同時に変更を行うと、コンテナーの破損につながります。STL コンテナーは、安全な同時アクセスを保証するためミューテックスでラップして保護できますが、このアプローチでは、一度に実行されるスレッドが 1 つに制限され、アムダールの法則で説明されように並列スピードアップが制限されます。対照的に、TBB の高度な並行コンテナーは、並行環境で主要なデータ構造を管理する優れたソリューションを提供します。

本章では、基本的なデータ構造の概要に続いて、TBB プロジェクトで利用できる高度な並行性を備えたコンテナーを紹介します。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_3

アルゴリズムを賢く選択しましょう: 同時実行コンテナーは万能薬ではない

並列データアクセスは、明確な並列化戦略に基づいている場合に最適ですが、その中でもアルゴリズムの選択は重要です。並行コンテナーで提供される制御されたアクセスにはコストがかかり、また、常に可能であるとは限りません。TBB は、そのようなサポートが実際にうまく機能する場合に、同時実行コンテナーを提供します（キュー、ハッシュテーブル、ベクトル）。

TBB は、「リスト」や「ツリー」などのコンテナーの並行処理をサポートしません。そのようなコンテナーでは、細粒度の共有が適切にスケールされないためです。並列処理を実現するには、アルゴリズムやデータ構造の選択を見直します。

同時実行コンテナーは、並列プログラムで同時実行サポートが適切に機能するコンテナーのスレッドセーフ・バージョンを提供します。これらは、粗粒度のロックを備えたシリアルコンテナーよりも高いパフォーマンスの代替手段を提供します。TBB コンテナーは、内部的に細粒度のロックまたはロックなしの実装を利用します。

主要なデータ構造: 基本

ハッシュテーブル、マップ、セット、キュー、ベクトルに精通している場合は、この節をスキップして、「同時実行コンテナー」の節から読み続けることもできます。TBB がどのように並列プログラミングをサポートしているか説明する前に、基本的なことを復習するため、主要なデータ構造について簡単に紹介します。

ベクトル

ベクトルは、固定インデックスを使用せずに要素を格納する動的なシーケンスコンテナーです。固定サイズを維持し、同じ型の要素のコレクションをインデックス形式で格納する配列とは異なり、ベクトルは要素の追加や削除に応じて自動的にサイズを調整できます。この柔軟性により、ベクトルは容量を増やすことができ、さまざまなデータ量に対応できる一方、配列は一度初期化されると静的なままであり、適応性が制限されます。

同時実行コンテナー、順序付きおよび順序なし

同時実行コンテナーはコレクションとして考えられ、より簡単に「セット」と呼ばれることがあります。ただし、技術用語では、マップ、セット、ハッシュテーブルなど特定の用語を使用して、

さまざまな種類のコレクションを記述します。

C++ では、同時実行コンテナーは連想配列を実装するために設計されたクラス・テンプレートです。各コンテナーは、要素を整理するキーに依存しています。マップコンテナーとセットコンテナーの各キーは一意ですが、マルチマップ・コンテナーとマルチセット・コンテナーでは同じキーの複数のインスタンスが許されます。セットコンテナーとマルチセット・コンテナーの場合、「Glass of Juice（グラス 1 杯のジュース）」、「Loaf of Bread（パン 1 斤）」、「Puppy in the Window（窓際の子犬）」などのキーの存在のみを確認できます。対照的に、マップコンテナーとマルチマップ・コンテナーにはキーと値のペアが格納され、`Info["Glass of Juice"].cost`、`Info["Loaf of Bread"].cost`、`Info["Puppy in the Window"].cost` などの値を照会できます。

これらのコンテナーは、整数、浮動小数点数、カスタムクラスなど、さまざまなデータ型を保持できる多目的テンプレートです。さらに、順序なしと順序ありの両方のバージョンが提供されます。各タイプのコンテナーには、その要素に特有の制約があります。

同時実行コンテナーには、次の 3 種類があります:

1. 順序付き vs. 順序なし
2. マップ vs. セット: *キーと値*があるか？ または、*キー*のみか ？
3. 複数の値: 同じ *キー*を持つ 2 つの項目を同じコレクションに挿入できるか？

順序付きと順序なし

順序なし同時実行コンテナーでは、要素は順序付けされません。順序付き同時実行コンテナーでは、要素は事前定義された順序（昇順など）で格納されます。

順序なしのコレクションに対する操作は、特に大規模なデータの場合、操作の平均時間が *O(log n)* ではなく *O(1)* であるため、より高速になる可能性があります。したがって、アルゴリズムですべてのメンバーを特定の順序で走査（C++ の用語では*反復処理*）する機能が必要な場合にのみ、順序付きバージョンを選択する必要があります。

順序なしコンテナーでは、すべてのメンバーを反復処理するときに保証されるのは、コンテナーの各メンバーを 1 回だけ参照することだけです。順序なしコンテナーでメンバーが遭遇する順序は保証されず、実行ごと、マシンごと、などによって異なる場合があります。

マップとセット

ここで「マップ」と呼ぶものは、実際には値が付加された単なる「セット」です。マップは辞書と呼ばれることもあります。果物のバスケット（リンゴ、オレンジ、バナナ、洋ナシ、レモン）を想像してください。果物が入った *セット*を見れば、バスケットの中に特定の種類の果物が入っているかが分かります。シンプルに「*はい*」か「*いいえ*」で、果物の種類をバスケッ

トに追加したり、削除したりできます。*マップ*はこれに値を追加しますが、多くの場合、情報を含むデータ構造になります。コレクション（フルーツバスケット）内の果物の種類の *マップ*を使用すると、数、価格、その他の情報を保持することを選択できます。単純な「*はい*」や「*いいえ*」の質問の代わりに、`Info[Apple].cost` や `Info[Banana].ripeness` について問い合わせることができます。値が複数のフィールドを持つ構造である場合、コスト、熟成度、色など複数の項目を照会できます。

複数の値

マップ内に存在するアイテムと同じ *キー*を使用してマップ/セットに何かを挿入することは、通常の「マップ」または「セット」コンテナーでは許されません（一意性を保証するため）が、「マルチマップ」および「マルチセット」バージョンでは許可されます。

「複数」バージョンでは重複が許可されますが、キー `Apple` はマップ/セット内で一意ではなくなるため、`Info[Apple].cost` などを検索できなくなります。このような場合、記録された各コストを調べるには、キー `Apple` を持つすべての要素を反復処理する必要があります。

ハッシュと比較

これまで述べたすべての順序なしバージョン（連想配列、マップ/セット、単一/複数）は、一般的に*ハッシュ*関数を使って実装されます。*ハッシュ*関数を理解するには、その目的を理解するのが最善です。連想配列 `LibraryCardNumber[顧客名]` を考えてみましょう。配列 `LibraryCardNumber` は、インデックスとして指定された名前（文字列）を持つ顧客のライブラリー・カードの番号を返します。この連想配列を実装する方法の 1 つは、要素のリンクリストを使用することです。残念ながら、要素を検索するには、リストを 1 つずつ検索して一致する要素を探す必要があります。これにはリスト全体の走査が必要になることがありますが、共有リスト構造へのアクセス競合が発生するため、並列プログラムでは非常に非効率です。並列処理を行っていなくても、項目を挿入する際に同じキーを持つ項目が他にないことを確認するには、リスト全体を検索する必要があります。リストに数千、数百万の利用者が登録されている場合、これには膨大な時間が必要です。ツリーなどさらに特殊なデータ構造では、これらの問題の一部は修正できますが、すべてを修正できるわけではありません。代わりに、データを配置する膨大な配列を想像してください。この配列は、従来の `array[integer]` メソッドでアクセスされます。これは非常に高速です。この場合、必要なのは、連想配列のインデックス（*顧客名*）を取得して、整数に変換する魔法の*ハッシュ*関数だけです。

同時実行コンテナー

TBB は、すべての C++ スレッド・アプリケーションで有用な、高度な並行性を備えたコンテ

ナークラスを提供します。TBB 並行コンテナークラスは、TBB を含むあらゆるスレッド化で利用できます。

C++ STL は当初、並行性を考慮して設計されていませんでした。そのため、STL コンテナーを同時に更新しようとすると、構造を変更する操作でコンテナーが破損する可能性があります（一部の操作は安全です）。もちろん、STL コンテナーを粗粒度のミューテックスでラップして、コンテナー上で一度に 1 つのスレッドのみが操作できるようにすることで、同時アクセスの安全性を保証できます。ただし、このアプローチでは並行性が排除されるため、パフォーマンスが重要なコードに適用すると、並列処理のスピードアップが制限されます。これは重要なことです。コンテナーを TBB 並行コンテナーに変換する場合、必要性に基づいて行う必要があります。並列に使用されるデータ構造は、アプリケーションのスケーリングを可能にするため同時実行性を考慮して設計する必要があります。並列にアクセスされないデータ構造は変更する必要はありません。ただし、将来のある時点で同時に使用される可能性があることを前提として新しいコードを作成することを強く推奨します。

TBB の並行コンテナーは、STL によって提供されるコンテナーと同様の機能を提供しますが、スレッドセーフな方法でそれを実行します。例えば、`tbb::concurrent_vector` は `std::vector` クラスに似ていますが、ベクトルを安全に並列に拡張できます。前述したように、並列に読み取るだけであれば、並行コンテナーは必要ありません。コンテナーを変更する並列コードがある場合にのみ、特別なサポートが必要になります。

TBB は、対応する STL コンテナーを互換性のある方法で置き換えることを目的としたいくつかのコンテナークラスを提供し、複数のスレッドが同じコンテナー上の特定のメソッドを同時に呼び出すことを可能にします。

重要　並行コンテナーは、それによって可能になる追加の同時実行によるスピードアップが、遅い順次パフォーマンスを上回る場合にのみに使用してください。

TBB の並行コンテナーには若干のコストがかかることに注意してください。一般的な STL コンテナーよりもオーバーヘッドが大きいため、STL コンテナーでの操作よりも時間がかかります。同時アクセスの可能性がある場合は、並行コンテナーを使用する必要があります。ただし、同時アクセスが不可能な場合は、STL コンテナーの使用を推奨します。並行コンテナーによって達成される同時実行によるスピードアップが、低速な順次パフォーマンスを上回る場合は、並行コンテナーを使用する必要があります。

コンテナーのインターフェイスは、同時実行性をサポートするのに変更が必要な場合を除き STL と同じです。一歩進んで、一部のインターフェイスがスレッドセーフではない典型的な例を考えてみます。これは理解すべき重要なポイントです。この例（図 3-1）は、STL の `test-for-empty` を使用したコードシーケンスに頼って、テストが空でない場合はポップする代わりに、キューに新しい `pop-if-not-empty` 機能（`try_pop` と呼ばれる）が必要になります。この STL コードの危険性は、別のスレッドが実行中で、元のスレッドのテストの後でポップの前にコンテナーを空にする可能性があり、その結果、ポップによって未定義の動作が発生する競合状態となる可能性があることです。つまり、STL コードはスレッドセ

ーフではありません。テストとポップ間のキューの変更を防ぐため、シーケンス全体をロックすることもできますが、このようなロックはアプリケーションの並列領域で使用するとパフォーマンスの低下を招くことが知られています。 この例（図 3-1）を理解すると、並列処理を適切にサポートするのに何が必要であるかが明らかになるでしょう。

std:: コードは、スレッドセーフではありません

```
#include <iostream>
#include <queue>
#include <tbb/parallel_invoke.h>

int main() {
  int sum0(0), sum1(0);
  std::priority_queue<int> myPQ;
  for(int i=0; i<10001; i+=1) {
    myPQ.push(i);
  }

  tbb::parallel_invoke(
    [&]() {
      while( !myPQ.empty() ) {
        sum0 += myPQ.top();
        myPQ.pop();
      }
    },
    [&]() {
      while( !myPQ.empty() ) {
        sum1 += myPQ.top();
        myPQ.pop();
      }
    });

// 正しい場合（正しくはありませんが）、
// 常に "total: 50005000" と表示されます
  std::cout << "total: "
            << sum0+sum1 << '\n';
  return 0;
}
//サンプルの出力 :
// total: 128379594
// total: 124432912
// total: 107697942
// total: 50005000
// total: 115224669
// total: 146790135
// total: 130683763
// total: 126960607
```

tbb:: コードは、スレッドセーフです

```
#include <iostream>
#include <tbb/concurrent_priority_queue.h>
#include <tbb/parallel_invoke.h>
#include <tbb/parallel_for.h>

int main() {
  int sum0(0), sum1(0);
  tbb::concurrent_priority_queue<int> myPQ;
  tbb::parallel_for(0,10001,1,
        [&](size_t i){ myPQ.push(i); } );

  tbb::parallel_invoke(
    [&]() {
      int item = 0;
      while( myPQ.try_pop(item) )
        sum0 += item;
    },
    [&]() {
      int item = 0;
      while( myPQ.try_pop(item) )
        sum1 += item;
    });

 // 正しいので、
 // 常に "total: 50005000" と表示されます
  std::cout << "total: "
            << sum0+sum1 << '\n';
 return 0;
}


 // 常に出力 :
 // total: 50005000
```

図 3-1.　STL と TBB の優先キューコードを並べて比較すると、top と pop の代わりに try_pop を使用する理由が分かります。STL 優先キューを並列で使用するのは安全ではないため、予期しない結果が生じます。サンプルコード: containers/pop_danger.cpp and pop_danger_fixed.cpp

STL と同様に、TBB のコンテナーはアロケーター引数に基づいてテンプレート化されます。各コンテナーは、それぞれのアロケーターを使用して、ユーザーが使用可能な項目にメモリーを割り当てます。TBB のデフォルト・アロケーターは、TBB に付属するスケーラブル・メモリー・アロケーターです（7 章で説明）。指定されたアロケーターに関係なく、コンテナーの実装では厳密に内部構造に対して異なるアロケーターを使用することがあります。

TBB は次の並行コンテナーを提供します:

- 同時実行コンテナー（順序付きバージョンと順序なしバージョン）: マップ、マルチマップ、セット、マルチセット
- ハッシュマップ（これは順序なしの同時実行コンテナークラスです。C++11 STL 以前の概念です）
- キュー: 通常、制限付き、優先
- ベクトル

TBB コンテナーのアロケーター引数はなぜ TBB にデフォルト設定されるか？

アロケーター引数はすべての TBB コンテナーでサポートされており、デフォルトでは TBB スケーラブル・メモリー・アロケーターに設定されます（7 章を参照）。

コンテナーは、デフォルトで `tbb::cache_aligned_allocator` と `tbb:tbb_allocator` を組み合わせて使用します。この章ではデフォルトについて説明しますが、デフォルトを理解する最も信頼できるリソースは、oneTBB 仕様と TBB ヘッダーファイルです。TBB はスケーラブル・アロケーターを検出するとそれを動的にロードするため、TBB スケーラブル・アロケーター・ライブラリー（7 章を参照）にリンクする必要はありません。ライブラリーが存在しない場合、コンテナーは暗黙的に `malloc` にフォールバックします。もちろん、このフォールバックはパフォーマンスの向上には効果がありません。

クラス名と C++11 に関する注意事項	並行走査と挿入	キーに値が関連付けられる	同時消去をサポート	ビルトインロック（解除が必要）	可視的ロックなし（ロックフリー・インターフェイス）	同じものを 挿入可能	[] と at アクセサー	順序付き要素（若干の追加オーバーヘッド）
`concurrent_hash_map` *C++11 以前のもの*	✓	✓	✓	✓	×	×	×	×
`concurrent_map` *C++11 の `_map` に類似*	✓	✓	×	×	✓	×	✓	✓
`concurrent_multimap` *C++11 の `_multimap` に類似*	✓	✓	×	×	✓	✓	×	✓
`concurrent_set` *C++11 の `_set` に類似*	✓	×	×	×	✓	×	×	✓
`concurrent_multiset` *C++11 の `_multiset` に類似*	✓	×	×	×	✓	✓	×	✓
`concurrent_unordered_map` *C++11 の `unordered_map` に類似*	✓	✓	×	×	✓	×	✓	×
`concurrent_unordered_multimap` *C++11 の `unordered_multimap` に類似*	✓	✓	×	×	✓	✓	×	×
`concurrent_unordered_set` *C++11 の `unordered_set` に類似*	✓	×	×	×	✓	×	×	×
`concurrent_unordered_multiset` *C++11 の `unordered_multiset` に類似*	✓	×	×	×	✓	✓	×	×

図 3-2.　同時順序付き同時実行コンテナーと非順序付き同時実行コンテナーの比較

並行順序なし同時実行コンテナー

順序なし同時実行コンテナーは、ハッシュテーブルのバリアントを実装するクラス・テンプレートのグループです。図 3-2 に、これらのコンテナーとそれらの主な機能の違いを示します。`concurrent_unordered` 順序なし同時実行コンテナーはテンプレートであるため、整数やカスタムクラスなど任意の要素を格納できます。TBB は、並行実行の優れたパフォーマンスを発揮できる同時実行コンテナーの実装を提供します。図 3-2 は、同時実行コンテナーの 9 つの TBB 実装を比較するリファレンスです。

C++11 以降、ハッシュテーブルの実装が STL に追加され、以前の実装との混乱や衝突を防ぐため、クラスには `unordered_map` という名前が選ばれました。

unordered_map という名前は、クラスへのインターフェイスとその要素の順序付けられていない性質を暗示しているため、より説明的であると言えます。C++11 では、unordered_map に加えて、unordered_set、unordered_multimap、unordered_multiset も追加されました。

並行処理で安全でない void STL インターフェイス

TBB は安全な並行性を提供しますが、TBB が安全でないとラベル付けする STL インターフェイスは避ける必要があります。

tbb::concurrent_hash_map と呼ばれるオリジナルの TBB ハッシュテーブルは、C++11 以前のものです。これは非常に有用であり、標準化に合わせて変更する必要はありませんでした。現在の TBB には、C++11 の機能追加を反映する map と set など 8 つのバリアント（図 3-2 を参照）のサポートが含まれるようになりました。同時アクセスをサポートするため、必要に応じてインターフェイスが拡張または調整されます。並列処理に適さないインターフェイスを避けることは、効率良い並列プログラミングを「促進する」取り組みの一部です。並列スケーリングを改善する注目すべき 2 つの調整があります:

1. 消去メソッド（concurrent_hash_map 以外）はスレッドセーフではありません。C++ 標準関数の消去メソッドには、同時消去が concurrent_hash_map でのみサポートされているため、同時実行が安全ではないことを示す unsafe_ プレフィクスが付けられます。これは、同時消去をサポートしていないため、concurrent_hash_map には適用されません。
2. バケットメソッドはスレッドセーフではありません。挿入操作には、同時実行が安全でないことを示す unsafe_ プレフィクスが付いています。これらは STL との互換性のためにサポートされていますが、可能な限り使用することを避けてください。使用する場合、挿入と同時に使用されないように保護する必要があります。TBB の設計者はこのような関数を避けたため、これらのインターフェイスは concurrent_hash_map には適用されません。

安全でないということは本当に安全でないこと

“unsafe” ラベルの付いたメソッドを使用するのは非常に危険であり、その弊害を理解した上で、デバッグコード内でのみ一時的に使用すべきです。

並列プログラムでの消去メソッドの実行は安全ではなく、消去自体をロックしても問題は解決しません。消去を行う必要がある場合、アプリケーションのどの部分も、消去によって影響を受ける可能性のあるコンテナーのビューを保持していないことを確認するのは開発者の責任です。消去を利用するアルゴリズムは避けるのが最善ですが、避けられない場合は、同時ビューが適切に管理される必要があります。

反復メソッドは、並列プログラムでは走査がアトミックではないため安全ではありません。つまり、走査中に同時にアイテムが追加された場合、走査は必ずしも今まで存在したビューを参照するとは限りません（サイドバーを参照）。

存在しなかった要素のリストを見る

走査（反復処理）の開始時に、a-C-e を含むセットについて考えてみます。a から C に移動した後、別の同時タスクが B を追加し、セットが a-B-C-e になったとします。さらに別の同時タスクが D を追加しセットが a-B-C-D-e になったとします。最初の走査は C から D へ、そして e へ続く可能性があります。この場合、セットの内容を a-C-D-e として参照したことになります。しかし奇妙なことに、a-C-D-e という状態は、実際には一度も存在していません。実際には a-C-e、次に a-B-C-e、最後に a-B-C-D-e という状態をたどりました。リストの変更と同時に走査を実行する場合、そのビューが適切であることを確認する必要があります。セット/マップの内容が独立している限り、これは比較的容易です。ただし、セット/マップのメンバー相互に依存性がある場合、面倒なことになる可能性があります。

このような同時実行性の影響を無視することはできません。走査は注意を払って適切に使用してください。

これらの構造を反復処理するとトラブルを招きます

図 3-4 では、最後のハッシュテーブルを反復処理してダンプする部分に、意図的に並行性に対して安全でないコードをいくつか忍び込ませてい。テーブルを走査している間に挿入や削除が同時に行われると、問題が発生する可能性があります。「これはデバッグ用のコードなので、気にする必要はありません」と言うこともできます。しかし、経験から、このようなコードがデバッグ用ではない本番コードに紛れ込んでしまうのは非常に簡単だということが分かっています。注意してください！

TBB の設計者は、デバッグ目的で、concurrent_hash_map にイテレーターを使用できるようにしましたが、他のメンバーからの戻り値としてイテレーターを使用することは意図的に避けました。残念ながら、STL は、私たちを誘惑します。concurrent_unordered_* コンテナーは、concurrent_hash_map とは異なります。API は、同時実行コンテナーの

C++ 標準に従います（オリジナルの TBB の concurrent_hash_map は、C++ による unordered コンテナーの標準化よりも古いことに注意してください）。データを追加または検索する操作はイテレーターを返すため、反復処理を実行します。並列プログラムでは、マップ/セット上の他の操作と同時に実行される危険があります。

誘惑に負けてしまうと、データの整合性を保護するのは完全にプログラマーに委ねられ、コンテナーの API は役立ちません。C++ 標準コンテナーは柔軟性をもたらしますが、concurrent_hash_map が提供するビルトインの保護が欠如していると言えます。追加または検索操作から返されたイテレーターを、検索した項目の参照以外の目的で使用しないという誘惑に負けなければ、STL インターフェイスを並行して使用するのは容易です。誘惑に負けてしまったら、アプリケーションの同時更新について考慮すべきことがたくさんあります（少し動機付けが必要な場合は、前の説明「存在しなかった要素のリストを参照する」を参照してください）。もちろん、更新が行われず、検索のみが行われる場合、イテレーターを使用しても並列プログラミングの問題は発生しません。

ハッシュマップ

ハッシュマップ（一般にハッシュテーブルとも呼ばれます）は、ハッシュ関数を使用してキーを値にマッピングするデータ構造です。ハッシュ関数はキーからインデックスを計算し、そのインデックスを使用して、キーに関連付けられた値が格納されている「バケット」にアクセスします。

適切なハッシュ関数の選択が特に重要です。完璧なハッシュ関数は各キーを一意のバケットに割り当てるため、異なるキーの衝突は発生しません。ただし、実際のハッシュ関数は完璧ではなく、複数のキーに対して同じインデックスが生成されることがあります。これらの衝突にはハッシュテーブルの実装による調整が必要であり、これによりオーバーヘッドが発生します。ハッシュ関数は、入力をバケット全体に均等に分散するようにハッシュすることで衝突を最小限に抑えるように設計する必要があります。

ハッシュマップの利点は、平均して検索と挿入に *O(1)* 時間を提供できることです。TBB ハッシュマップの利点は、正当性とパフォーマンスの両方において同時使用をサポートしていることです。

これは、適切なハッシュ関数（使用されるキーの衝突をあまり引き起こさない）が使用されることを前提としています。ハッシュ関数が不完全である場合、またはハッシュテーブルの次元が適切でない場合は、理論上の最悪ケースである O(n) となります。

実際の使用ケースにおいて、ハッシュマップは、検索ツリーを含む他のテーブル検索データ構造よりも効率的である場合が多くなります。そのため、ハッシュマップは、連想配列、データベース・インデックス、キャッシュ、セットなど、さまざまな目的のデータ

構造に選ばれます。

concurrent_hash_map

TBB は、複数のスレッドが find、insert、erase メソッドを介して同時に値にアクセスできるようにキーを値にマッピングする、concurrent_hash_map を提供します。後述するように、tbb::concurrent_hash_map は並列処理用に設計されているため、この章の後半で説明する STL map/set インターフェイスとは異なり、スレッドセーフです。

キーは順序付けされません。concurrent_hash_map には、キーごとに最大 1 つの要素があります。キーにはマップ以外で処理を行っているほかの要素がある可能性があります。HashCompare 型は、キーをハッシュする方法と、キーが等しいか比較する方法を指定します。ほとんどのハッシュテーブルでは、2 つのキーが等しい場合、同じハッシュコードにハッシュされる必要があります。そのため、HashCompare では、比較とハッシュの概念を別々に扱うのではなく、単一のオブジェクトに結び付けています。このもう 1 つの結果として、ハッシュテーブルが空でない間はキーのハッシュコードを変更してはなりません。

concurrent_hash_map は、std::pair<const Key,T> 型の要素のコンテナーとして動作します。通常、コンテナー要素にアクセスする場合、更新または読み取りを行います。concurrent_hash_map テンプレート・クラスは、スマートポインターとして動作する accessor と const_accessor を使用して、それぞれ 2 つの目的をサポートします。accessor は、update (write) アクセスを表します。要素を指す限り、accessor が完了するまで、テーブルのキーを参照する操作はすべてブロックされます。const_accessor は、読み取り専用アクセスであることを除いて同じです。複数のアクセサーで同時に同じ要素を指すことができます。この機能は、要素が頻繁に読み取りされ、ほとんど更新されない状況で並列性を大幅に向上させます。

図 3-3 と 3-4 に、concurrent_hash_map コンテナーを使用した簡単なコード例を示し、図 3-5 に出力例を示します。要素アクセスの有効期間を短縮することで、この例のパフォーマンスを向上できます。find メソッドおよび insert メソッドは、accessor または const_accessor を引数として受け取ります。これは、concurrent_hash_map に更新と読み取り専用アクセスのどちらを要求するかを知らせます。いったんメソッドが返されると、accessor または const_accessor が破棄されるまでアクセスは続きます。要素へのアクセスはほかのスレッドをブロックするため、accessor または const_accessor のライフタイムが短くなるように、最内ブロックで宣言を行ってください。アクセスをブロックの終了よりも早く解放するには、release メソッドを使用します。図 3-6 のサンプルは、図 3-3 のループ本体を変更したもので、破棄時に依存してスレッドのライフタイムを終了する代わりに、release を使用しています。remove(key) メソッドも同時に操作できます。このメソッドは書き込みアクセスを暗黙的に要求します。そのため、キーを削除する前にその key に対するほかの既存のアクセスが完了するのを待機します

ビルトインロックと不可視なロック

コンテナー concurrent_hash_map と concurrent_unordered_* は、アクセスされた要素のロックに関していくつかの違いがあります。したがって、競合状態では動作が大きく異なる可能性があります。concurrent_hash_map のアクセサーは本質的にロックです。accessor は排他ロックであり、const_accessor は共有ロックです。ロックベースの同期はコンテナーの使用モデルにビルトインさせており、コンテナーの整合性だけでなく、データ整合性も保護します。図 3-3 のコードでは、テーブルへの挿入に accessor を使用します。

```
#include <tbb/concurrent_hash_map.h>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#include <string>

// ユーザー定義型を扱うハッシュと比較操作を定義する構造体
struct MyHashCompare {
  static size_t hash( const std::string& x ) {
    size_t h = 0;
    for( const char* s = x.c_str(); *s; ++s )
      h = (h*17)^*s;
    return h;
  }
  //! 文字列が等しい場合は true
  static bool equal( const std::string& x, const std::string& y) {
    return x==y;
  }
};

// int に文字列をマップする並列ハッシュテーブル
typedef tbb::concurrent_hash_map<std::string,int,MyHashCompare> StringTable;

// 文字列の出現回数をカウントする関数オブジェクト
struct Tally {
  StringTable& table;
  Tally( StringTable& table_) : table(table_) {}
  void operator()( const tbb::blocked_range<std::string*> range) const
    {
    for(std::string* p=range.begin(); p!=range.end(); ++p) {
      StringTable::accessor a; table.insert(a, *p);
      a->second += 1;
    }
  }
};
```

図 3-3. ハッシュテーブルの例 (パート 1/2)。サンプルコード: containers/concurrent_hash_maps.cpp

```
const  size_t  N = 10;

std::string Data[N] = { "Hello", "World", "TBB", "Hello", "So  Long",
      "Thanks for all the fish", "So Long", "Three", "Three",
      "Three"  };

int  main() {
  // 空のテーブルを構築
  StringTable table;

  // 発生回数をテーブルに格納
  tbb::parallel_for( tbb::blocked_range<std::string*>( Data, Data+N, 1000  ),
        Tally(table) );


  // 単純なウォークを使用して出現箇所を表示
  // (注意:concurrent_hash_map は const_iterator を提供していません)
  // このコードに問題はありますか ???
  // "これらの構造を反復処理すると問題が発生する" を参照してください
  // (数ページ後にあります)
  for( StringTable::iterator i=table.begin(); i!=table.end(); ++i )
    printf("%s  %d\n",i->first.c_str(),i->second);

  return  0;
}
```

図 3-4. ハッシュテーブルの例 (パート 2/2)。サンプルコード: containers/concurrent_hash_maps.cpp

```
Three 3
So Long 2
Hello 2
TBB 1
World 1
Thanks for all the fish 1
```

図 3-5. 図 3-3 と 3-4 のサンプルプログラムの出力

```
        for( std::string* p=range.begin(); p!=range.end(); ++p ) {
          StringTable::accessor a;
          table.insert( a, *p );
          a->second += 1;
          a.release();
        }
```

図 3-6. スケーリングの向上を期待して、アクセサーのライフタイムを短縮する図 3-3 への修正。サンプルコード: containers/concurrent_hash_maps.cpp

ハッシュマップのパフォーマンスに関するヒント

- 常にハッシュテーブルの初期サイズを指定します。デフォルトが 1 の場合、スケールは悪化します。適切なサイズは間違いなく数百から始まります。小さいサイズが適切であると思われる場合、小さなテーブルでロックを使用すると、キャッシュの局所性により速度面で利点が得られます。
- ハッシュ関数をチェックして、ハッシュ値の下位ビットに適度な疑似乱数性があることを確認してください。特に、ポインターをキーとして使用してはなりません。一般に、ポインターはオブジェクトのアライメントにより、下位ビットに一定数のゼロビットを持つためです。このような場合、ポインターをそれが指す型のサイズで除算し、常にゼロのビットをシフトアウトして変化するビットを優先することを強く推奨します。素数を掛けて、下位ビットの一部をシフトアウトすることは、検討に値する戦略です。ほかのハッシュテーブルと同様に、等価なキーのハッシュコードは同じでなければなりません。理想的なハッシュ関数は、キーをハッシュコード空間に均等に分散します。最適なハッシュ関数の調整はアプリケーションによって異なりますが、TBB が提供するデフォルトを使用すると、うまく機能する傾向があります。
- アクセサーを使用せずに済む場合は使用せず、アクセサーが必要な場合はそのライフタイムを可能な限り短くします（図 3-6 の例を参照）。これらは実質的に細粒度のロックであり、存続する間は他のスレッドを禁止するため、スケーリングが制限される可能性があります。
- TBB メモリー・アロケーターを使用します（7 章を参照）。`scalable_allocator` の使用を強制し、`malloc` へのフォールバックを許可しない場合は、コンテナーのテンプレート引数として `scalable_allocator` を使用します。これにより、開発中にパフォーマンスをテストする際の健全性チェックも適切に行われます。

map と set の同時サポート

標準 C++ STL では、set、map、multiset、multimap、unordered_set、unordered_map、unordered_multiset、unordered_multimap が定義されています。これらのコンテナーは、要素に課された制約によってのみ異なります。TBB はこれらのインターフェイスの同時サポートを提供します。図 3-2 は、map および set の STL と互換性のある 8 つのインターフェイスの STL 名に、concurrent_ が付加されていることを示しています。

要素の反復処理を同時に実行するのは安全ではないことに十分注意してください。それが安全でないのは、オリジナルの TBB がこれらのインターフェイスの順序なしバージョンのみをサポートしていたためです。順序付けられたバージョンは、利用者の要求に基づいて後から追加され、同時走査（反復処理）の危険性に関する TBB 開発チームからの警告が加えられています。

図 3-7 は、concurrent_map と concurrent_unordered_map の簡単な例を示しており、その後に、安全でない走査を使用するコンテンツのデバッグダンプが示されています（ただし、これはデバッグ専用です）。順序付けられたマップがアルファベット順であることに注意してください。これにより、ギリシャ語順に挿入するときに値として割り当てた数字が並べ替えられます。また、順序なしマップの走査順序は、挿入した順序と一致しないことにも注意してください。順序なしマップでは順序は重要ではありません。

```
#include  <iostream>
#include  <vector>
#include  <tbb/concurrent_map.h>
#include  <tbb/concurrent_unordered_map.h>

using  namespace  std;

int  main()
  vector<string> names {"alpha", "beta", "gamma", "delta",
                        "epsilon", "zeta", "eta", "theta", "iota",
                        "kappa", "lambda", "mu", "nu", "xi",
                        "omicron","pi", "rho", "sigma", "tau",
                        "upsilon", "phi", "chi", "psi", "omega"};

  tbb::concurrent_map<string,int> greekOrdered;
  tbb::concurrent_unordered_map<string,int> greekToMe;

  for(int  i=0; i<names.size(); i++) {
    greekOrdered[names[i]] = i;
    greekToMe[names[i]] = i;
  }

  for(auto  i=greekOrdered.begin(); i!=greekOrdered.end(); i++) {
    cout << i->first << "("  << i->second << ") ";
  }
  cout << "\n\n";
  for(auto  i=greekToMe.begin(); i!=greekToMe.end(); i++) {
    cout << i->first << "("  << i->second << ") ";
  }
  cout << "\n";
  return  0;
}

Output:
alpha(0)  beta(1)  chi(21)  delta(3)  epsilon(4)  eta(6)  gamma(2)  iota(8)
kappa(9) lambda(10) mu(11) nu(12) omega(23) omicron(14) phi(20) pi(15)
psi(22)  rho(16)  sigma(17)  tau(18)  theta(7)  upsilon(19)  xi(13)  zeta(5)

rho(16)  omega(23)  omicron(14)  tau(18)  pi(15)  delta(3)  mu(11)
eta(6)  kappa(9)  alpha(0)  nu(12)  xi(13)  epsilon(4)  theta(7)  beta(1)
psi(22)  sigma(17)  lambda(10)  gamma(2)  phi(20)  iota(8)  upsilon(19)
zeta(5)  chi(21)
```

図 3-7. concurrent_map と concurrent_unordered_map を使用する簡単な例。サンプルコード: containers/concurrent_maps.cpp

同時キュー: 通常のキュー、制限付きキュー、および優先度付き

キューは、push（追加）と pop（削除）操作によって項目が追加または削除される便利なデータ構造です。無制限キュー・インターフェイスは、キューが空で、キューから値が pop されなかったかどうか通知する try_pop を提供します。これにより、スレッドセーフではない操作で空かテストしてブロッキング・ポップを回避する独自のロジックを記述する必要がなくなります（図 3-1 を参照）。複数のスレッド間でキューを共有することは、スレッドからスレッドにワーク項目を渡す効果的な方法です。「ワーク」を保持するキューには、将来の処理を要求するワーク項目を追加したり、処理を実行するタスクが項目を取り出すことができます。

通常、キューは先入れ先出し（FIFO）方式で動作します。空のキューから始めて、push(10) を実行し、次に push(25) を実行すると、最初の pop 操作で 10 が返され、2 番目の pop 操作で 25 が返されます。これは、後入先出となるスタックの動作とは大きく異なります。しかし、ここではスタックの話をしていません。

図 3-8 に簡単な例を示します。これは、pop 操作が、push 操作でキューに追加されたのと同じ順序で値を返すことを明確に示しています。

```
#include <tbb/concurrent_queue.h>
#include <tbb/concurrent_priority_queue.h>
#include <iostream>

int myarray[10] = { 16, 64, 32, 512, 1, 2, 512, 8, 4, 128 };

void pval(int test, int val) {
  if (test) {
    std::cout << " " << val;
  } else {
    std::cout << " ***";
  }

void simpleQ() {
  tbb::concurrent_queue<int> queue;
  int val = 0;
  for( int i=0; i<10; ++i )
    queue.push(myarray[i]);
  std::cout << "Simple Q pops are";
  for( int i=0; i<10; ++i )
    pval( queue.try_pop(val), val );
  std::cout << std::endl;
}

出力:
Simple  Q pops are 16 64 32 512 1 2 512 8 4 128
```

図 3-8.　シンプル（FIFO）キューの使用例。サンプルコード: containers/concurrent_queues.cpp

キューには、境界と優先順位の 2 つがあります。境界はキューのサイズを制限します。つまり、キューがいっぱいの場合、push できない場合があります。これを管理するため、制限付きキュー・インターフェイスは、キューに追加できるようになるまで push を待機させる方法や、push できる場合は push を実行するか、キューがいっぱいであることを知らせる「push を試行する」操作を実行する方法を提供します。キューはデフォルトでは無制限です。制限付きキューが必要な場合は、concurrent_bounded_queue を使用し、set_capacity メソッドを呼び出してキューのサイズを設定します。図 3-9 に、push された最初の 6 つの項目だけがキューに入る、制限付きキューの簡単な使用法を示します。try_push にテストを追加して処理ををを行うことができます。この場合、pop 操作でキューが空であることが検出されると、プログラムは *** を出力します。

```
void  boundedQ() {
  tbb::concurrent_bounded_queue<int>
  queue;
  int  val = 0;

  queue.set_capacity(6);

  for( int  i=0; i<10; ++i )
    queue.try_push(myarray[i]);

  std::cout << "Bounded Q     pops are";

  for( int  i=0; i<10; ++i )
    pval( queue.try_pop(val), val );
  std::cout << std::endl;
}

出力:

Simple Q pops are 16 64 32 512 1 2 512 8 4 128
Bounded Q pops are 16 64 32 512 1 2 *** *** *** ***
```

図 3-9.　このルーチンはプログラムを拡張し、制限されたキューの使用状況を表示します。サンプルコード: containers/concurrent_queues.cpp

優先順位は、キュー内の項目を効果的に並べ替えることで、先入先出ポリシーに 2 番目の工夫を加えます。コード内で指定しない場合、デフォルトの優先順位は std::less<T> です。つまり、ポップ操作はキュー内の最も値の大きい項目を返します。

図 3-10 は、優先順位の使用例を 2 つ示しています。1 つはデフォルトで std::less<int> を指定し、もう 1 つは std::greater<int> を明示的に指定しています。

```
void  prioQ() {
  tbb::concurrent_priority_queue<int> queue;
  int  val = 0;
  for( int  i=0; i<10; ++i )
    queue.push(myarray[i]);
  std::cout << "Prio  Q  pops are";

  for( int  i=0; i<10; ++i )
    pval( queue.try_pop(val), val );
  std::cout << std::endl;
}

void  prioQgt() {
  tbb::concurrent_priority_queue<int,std::greater<int>> queue;
  int  val = 0;
 for( int  i=0; i<10; ++i )
   queue.push(myarray[i]);
   std::cout << "Prio    Qgt pops are";

 for( int  i=0; i<10; ++i )
   pval( queue.try_pop(val), val );
   std::cout << std::endl;
}

出力:

Simple  Q   pops are 16 64 32 512 1 2 512 8 4 128
Bounded Q pops are 16 64 32 512 1 2 *** *** *** ***
Prio Q pops are 512 512 128 64 32 16 8 4 2 1
Prio Qgt pops are 1 2 4 8 16 32 64 128 512 512
```

図 3-10. これらのルーチンは、優先キューを示すためプログラムを拡張します。サンプルコード containers/concurrent_queues.cpp

前の 3 つの図の例で示すように、キューにこれらの 3 つのバリエーションを実装するため、TBB は、3 つのコンテナークラス oncurrent_queue、concurrent_bounded_queue、concurrent_priority_queue を提供しています。すべての同時実行キューでは、複数のスレッドが項目の push と pop を同時に行うことができます。インターフェイスは、STL std::queue または std::priority_queue に似ていますが、キューの同時変更を安全に行うために異なる点があります。

キューの基本メソッドは、push と try_pop です。push メソッドは std::queue と同じように動作します。フロントメソッドやバックメソッドはキュー内の項目への参照を返すため、同時実行環境では安全ではないことから、これらのメソッドはサポートされていないことに注意してください。並列プログラムでは、キューの先頭または最後尾が別のスレッドによって並列に変更される可能性があるため、先頭または最後尾の使用は無意味です。

同様に、pop と空であるかのテストは、無制限のキューではサポートされていません。代わりに、項目が利用可能な場合はそのア項目をポップして true ステータスを返すようにメソッド try_pop が定義されています。そうでない場合、メソッド try_pop は項目を返さず、false ステータスを返します。スレッドセーフなコーディングを促進するため、test-for-empty メソッドと pop メソッドが 1 つのメソッドに結合されています。制限付きキューの場合、潜在的にブロックする push メソッドに加えて、非ブロックである try_push メ

ソッドがあります。これらは、キューのサイズを照会する size メソッドを回避するのに役立ちます。一般に、size メソッドは、特にシーケンシャル・プログラムから引き継がれた場合には、使用を避ける必要があります。並列プログラムではキューのサイズが同時に変更される可能性があるため、size メソッドを使用する場合は慎重に検討してください。例えば、キューが空で、保留中の pop メソッドがある場合、TBB は size メソッドに対して負の値を返すことがあります。サイズが 0 以下の場合、empty メソッドは true になります。

境界サイズ

concurrent_queue および concurrent_priority_queue の容量は無制限ですが、ターゲットマシンのメモリー量に制限されます。concurrent_bounded_queue は境界の制御を提供します。重要な機能は、キューに空きができるまでプッシュメソッドがブロックされることです。制限付きキューは、キューが無制限に拡大するのを許す代わりに、消費速度に合わせて供給速度を遅くするのに役立ちます。

concurrent_bounded_queue は、pop メソッドを提供する唯一の concurrent_queue_* コンテナーです。pop メソッドは、項目が利用可能になるまでブロックします。push メソッドは、concurrent_bounded_queue でのみブロッキングできるため、このコンテナータイプでは try_push と呼ばれる非ブロッキング・メソッドも提供されます。

メモリーのオーバーフローやコアのオーバーコミットを回避するため、境界を使用してレートを一致させる方法は、limiter_node を使用してフローグラフ（4 章を参照）でも使用できます。

優先度付き

優先度付きキューは、キューに投入された個々の項目の優先度に基づいて、キュー内の順序を維持します。前述したように、通常のキューは先入れ先出し（FIFO）ポリシーを持ちますが、優先度付きキューは項目をソートします。std::less<T> のデフォルト順序を変更するため、独自の比較関数を提供できます。例えば、std::greater<T> を使用すると、最小の要素が pop メソッドで次に取得される要素になります。図 3-10 のサンプルコードでこれを行っています。

スレッドセーフの維持: Top、Size、Empty、Front、および Back を忘れてください

top メソッドは存在しないことに注意することが重要であり、size メソッドと empty メソッドの使用は避けるべきです。同時（並行）使用とは、他のスレッドの push/pop メソッドにより、3 つの値がすべて変更される可能性があることを意味します。また、clear メソッドと swap メソッドはサポートされていますが、スレッドセーフではありません。TBB では、

返される要素が同時 pop によって無効化される可能性があるため、`std::priority_queue` を `tbb::concurrent_priority_queue` に変換するときに、`top` を使用してコードを書き直す必要があります。戻り値は同時実行によって危険になることはないため、TBB は `size`、`empty`、`swap` の `std::priority_queue` メソッドをサポートします。ただし、どちらかの機能に依存している場合は、並行処理用にコードを書き直す必要性を示唆するため、並行アプリケーションでどちらの機能を使用するか慎重に検討することをお勧めします。

イテレーター

デバッグ目的のみで、3 つの同時キューはすべて、限定的なイテレーターのサポート（`iterator` および `const_iterator` 型）を提供します。このサポートは、デバッグ中にキューを検査できるようにするだけのものです。`iterator` および `const_iterator` 型はどちらも前方イテレーターの通常の STL 規則に従います。反復順は、以前プッシュされたものから最近プッシュされたものの順になります。キューを変更すると、それを参照するすべてのイテレーターが無効になります。イテレーターは比較的遅いため、デバッグにのみ使用してください。使用例を図 3 - 11 に示します。

```
#include  <tbb/concurrent_queue.h>
#include  <iostream>

int  main() {
  tbb::concurrent_queue<int> queue;
  for( int  i=0; i<10; ++i )
    queue.push(i);
  for( tbb::concurrent_queue<int>::const_iterator
       i(queue.unsafe_begin());
       i!=queue.unsafe_end();
       ++i )
    std::cout << *i << "  ";
  std::cout << std::endl;
  return  0;
}

出力:
0 1 2 3 4 5 6 7 8 9
```

図 3-11. 同時実行キューを反復処理するサンプル・デバッグ・コード – これらのメソッドがデバッグ専用の非スレッドセーフであることを強調するため、`begin` と `end` に `unsafe_` プレフィクスが付加されていることに注意してください。サンプルコード: `containers/debugging_queue.cpp`

同時実行キューを使用する理由: A-B-A 問題

この章の冒頭で述べたように、並列処理の専門家によって書かれたコンテナーを「ただ使うだけ」で済むことには、大きな価値があります。スケーラブルな実装をアプリケーションごとに一から作り直したいと思う人はいないでしょう。その動機付けとして、ここでは A-B-A 問

題という、並列処理がうまくいかない典型的なコンピューター・サイエンスの例を挙げて説明します。一見すると、並行キューは自力で簡単に記述できるように思えるかもしれません。しかし、実際にはそうではありません。TBB の `concurrent_queue` や、十分に調査され、適切に実装された他の並行キューを使用することをお勧めします。この経験は謙虚な気持ちにさせるかもしれませんが、並行キューが安易に作れるものではないと信じていたのは、私たちだけではありません。A-B-A 問題（別項を参照）が意図に反する場合、8 章で紹介される更新イディオム（`compare_exchange_strong`）は適切ではありません。リンクされたデータ構造に対する非ブロック・アルゴリズムの設計に問題があるケースが大半です。

TBB の設計者は、同時実行キューのソリューションの中に A-B-A 問題に対する解決策をすでに組み込んでいます。私たちはそれを利用するだけで済みます。もちろん、オープンソース・コードであるため、興味があればコードを調べて解決策を確認することもできます。ソースコードを調べると、アリーナ管理（11 章）でも A-B-A 問題に対処する必要があることが分かります。もちろん、これらを知らなくても TBB を使用することはできます。ここで強調したいのは、並行データ構造の解決は見た目ほど簡単ではないということです。それが、ここで TBB でサポートされている並行データ構造の使用にこだわる理由です。

A-B-A 問題

A-B-A 問題を理解することは、独自のアルゴリズムを設計する際に並行性の影響を徹底して考えるように自身を訓練する重要な方法です。TBB は、同時実行キューや他の TBB 構造を実装する際に A-B-A 問題を回避しますが、「並列で考える」必要があることを思い出させてくれます。

A-B-A 問題は、スレッドが位置をチェックして値が A であることを確認し、値が A であった場合にのみ更新を続行するときに発生します。最初のタスクが検知しない方法で他のタスクが同じ位置を変更した場合、問題になるのかという疑問が生じます:

1. タスクが `globalx` から値 A を読み取ります。
2. 他のタスクが `globalx` を A から B に変更し、その後 A に戻します。
3. 最初のタスクは `compare_and_swap` を実行し、A を読み取りますが、途中で B に代わっていたことを検出できません。

最初のタスクが、「そのアドレスの値は最初に読み取ってから変わっていない」という誤った前提で処理を進めてしまうと、オブジェクトを破壊したり、誤った結果を得る可能性があります。

リンクリストの例を考えてみましょう。リンクリスト `W(1)→X(9)→Y(7)→Z(4)` を

想定します。ここで、文字はノードの位置、数字はノード内の値です。あるタスクがリストを走査して、デキューするノード X を検索するとします。タスクは、次のポインター X.next (Y) をフェッチし、それを W.next に格納することを目的としています。ただし、スワップが完了する前に、タスクはしばらく中断されます。

この停止中に、他のタスクは稼動しています。他のタスクは X をデキューして、次に同じメモリーを再使用して、ある時点で Y をデキューして Q を追加するとともに、ノード X の新しいバージョンをキューします。ここで、リストは W(1)→X(2)→Q(3)→Z(4) です。

オリジナルのタスクが最終的に起動すると、W.next がまだ X を指していることが分かるので、W.next を Y にスワップアウトし、リンクリストが完全に混乱してしまいます。

アトミック操作は、アルゴリズムで十分な保護を組み込んでいる場合は最適な方法です。A-B-A 問題で 1 日が台無しになる可能性がある場合は、より複雑な解決策を見つける必要があります。tbb::concurrent_queue に、この問題を正しく解決するのに必要な複雑さを備えています。

キューを使用しない場合: アルゴリズムを考える

キューは、生産タスクと消費タスクを仲介するため並列プログラムで広く使用されています。明示的なキューを使用する前に、代わりに parallel_for_each または parallel_pipeline の使用を検討する必要があります（2 章を参照）。これらの操作は、多くの場合、次の理由によりキューより効率的です。

キューは順序を維持する必要があるため、本質的にボトルネックとなります。

キューが空の場合、値をポップしているスレッドは値がプッシュされるまでストールします。

キューは受動的なデータ構造です。スレッドが値をプッシュした場合、値をポップするまで時間がかかり、値（およびその参照）はキャッシュにない「コールド」状態になります。また、さらに悪い状況では、別のスレッドが値をポップし、値（およびその参照）を別のプロセッサーに移動しなければなりません。

対照的に、parallel_for_each および parallel_pipeline はこれらのボトルネックを回避します。これらのスレッド化は暗黙的であるため、値が利用可能になるまでほかのワークを行うようにワーカースレッドを最適化します。また、キャッシュ上の項目を*ホット*な状態で維持しようとします。例えば、新しいワーク項目が parallel_for_each に追加されると、アイドル状態の別のスレッドがその項目をスチールできるまで、その項目を追加したスレッドのローカルに保持されます。これにより、項目はホットスレッドによってより頻繁に処

理され、データ取得の遅延が削減されます。

同時実行ベクトル

TBB は、concurrent_vector と呼ばれるクラスを提供します。concurrent_vector<T> は、T の動的に拡張可能な配列です。他のスレッドが要素を操作しているときや、それ自体を拡張しているときでも、concurrent_vector を安全に拡張できます。安全な同時拡張のため、concurrent_vector には、動的配列の一般的な使用をサポートする 3 つのメソッド（push_back、grow_by、grow_to_at_least）があります。

図 3-12 は、concurrent_vector の簡単な使用方法を示しており、図 3-13 は、ベクトルの内容のリストで、並列スレッドによる同時実行の効果を示しています。同じプログラムからの出力は、数値順に並べれば同一であることが分かります。

std::vector の代わりに tbb::concurrent_vector を使用する場合

concurrent_vector<T> の重要な点は、ベクトルを同時に拡張できることと、要素がメモリー内で移動しないことが保証されることです。

concurrent_vector は std::vector よりもオーバーヘッドが大きくなります。したがって、他のアクセスを処理する際に動的にサイズを変更する必要がある場合、または要素を移動してはならない場合のみ、concurrent_vector を使用してください。

```
#include  <iostream>
#include  <tbb/concurrent_vector.h>
#include  <tbb/parallel_for.h>
void  oneway() {
// 整数を含むベクトルを作成
     tbb::concurrent_vector<int> v = {3, 14, 15,
     92};
     // ベクトルに整数をシリアルに加算
    for( int  i = 100; i < 1000; ++i ) {
      v.push_back(i*100+11);
      v.push_back(i*100+22);
      v.push_back(i*100+33);
      v.push_back(i*100+44);
    }
     // ベクトルの値を反復してプリント（デバッグのみ）
    for(int  n : v) {
    std::cout << n << std::endl;

    }
}
void  allways() {
// 整数を含むベクトルを作成
tbb::concurrent_vector<int> v = {3, 14, 15, 92};
     // ベクトルに整数を並列に加算
     tbb::parallel_for( 100, 999, [&](int  i){
```

```
            v.push_back(i*100+11
            );
            v.push_back(i*100+22
            );
            v.push_back(i*100+33
            );
            v.push_back(i*100+44
            );
          });
        // ベクトルの値を反復してプリント（デバッグのみ）
        for(int  n : v) {
          std::cout << n << std::endl;
        }
    }
```

図 3-12.　並行ベクトルの小さな例。サンプルコード: containers/concurrent_vectors.cpp

```
3                       3
14                      14
15                      15
92                      92
10011                   10011
. . .                   . . .
84911                   72611
84922                   91211
84933                   87111
84944                   72622
85011                   91222
85022                   87122
85033                   72633
85044                   91233
. . .                   . . .
99933                   99833
99944                   99844
```

図 3-13.　左側は for（並列ではない）使用時に生成された出力であり、右側は parallel_for（ベクトルへの同時プッシュ）使用時の出力を示しています。

要素が移動しない

concurrent_vector では、その成長に伴って要素が移動することはありません。これは、シングルスレッドのコードでも、STL std::vector より有利です。コンテナーは連続した配列のシリーズを割り当てます。最初の確保、拡張、または割り当て操作によって、配列のサイズが決まります。初期サイズとして少数要素を使用すると、キャッシュライン間で断片化が発生し、要素のアクセス時間が長くなる可能性があります。shrink_to_fit() は、いくつかの小さな配列を 1 つの連続した配列にマージし、アクセス時間を改善できる可能性があります。

concurrent_vectors の並行成長

並行成長は理想的な例外安全性と基本的に非互換ですが、concurrent_vector は実用レベルの例外安全性を提供します。要素型には、例外をスローしないデストラクターが必要です。また、コンストラクターが例外をスローできる場合、デストラクターは非仮想であり、ゼロで埋められたメモリー上で正しく動作する必要があります。

push_back(x) メソッドは x をベクトルに安全に追加します。grow_by(n) メソッドは、T() で初期化された n 個の連続する要素を安全に追加します。どちらのメソッドも、最初に追加された要素を指すイテレーターを返します。各要素は T() で初期化されます。次のルーチンは、文字列を共有ベクトルに安全に追加します:

```
void  Append( tbb::concurrent_vector<char>& vector, const  char* string  ) {
  size_t  n = strlen(string)+1;
  std::copy( string, string+n, vector.grow_by(n) );
}
```

grow_to_at_least(n) は、ベクトルがまだその大きに達していない場合、ベクトルを少なくともサイズ n まで拡大します。growth メソッドの同時呼び出しは、必ずしも要素がベクトルに追加された順序で返されるわけではありません。

size() はベクトル内の要素数を返します。これには、push_back、grow_by、または grow_to_at_least メソッドによって並行して構築中の要素が含まれる場合があります。前の例では、concurrent_vector 内の要素が連続したアドレスにない可能性があるため、strcpy とポインターではなく、std::copy とイテレーターを使用しています。また、イテレーターが end() 値を超えない限り、concurrent_vector が拡張する間も、安全にイテレーターを使用できます。ただし、イテレーターは並行に構築中の要素を参照する場合があります。そのため、構築とアクセスを同期する必要があります。

concurrent_vector の操作は、拡張に関しては安全に並行処理できますが、ベクトルのクリアや破棄に関しては安全ではありません。parallel_vector で他の操作が進行中の場合、clear() を呼び出さないでください。

まとめ

この章では、TBB でサポートされている 3 つの主要なデータ構造（ハッシュ/マップ/セット、キュー、ベクトル）について説明しました。これらの TBB のサポートは、スレッドセーフ（同時実行可能）と、スケーラブルな実装を提供します。また、並列プログラムでは問題の原因になりやすい、避けるべきことのアドバイスを提供しました。これには、マップ/セットから返されるイテレーターを、検索した 1 つの項目以外に用いることも含まれます。独自の concurrent_queue を書く代わりに TBB を使用する動機付けとして、また並列プログラムがデータを共有するときに必要な考え方の優れた例として、A-B-A 問題を検討しました。

他の章と同様に、図に示されているコードはすべてダウンロード可能です。

コンテナーの並列使用はサポートされてはいますが、効果的な並列プログラミングにはあらゆる種類の同期を最小限に抑えたアルゴリズムを検討することが、効果的な並列プログラミングに重要であるという概念は、強調してもしすぎることはありません。「キューを使

用しない場合: アルゴリズムを考える」の節で述べたように、parallel_for_each、parallel_pipeline、parallel_reduce などを使用してデータ構造の共有を回避できれば、プログラムのスケーラビリティーが向上する可能性があります。最も効果的な並列プログラミングには、この点を徹底的に考えることが重要であるため、本書ではこの点について何度も言及しています。

4 章　フローグラフ: 基本

フローグラフ・インターフェイス [flow_graph] を使用すると、2 章の図 2-2 で説明したイベントベースの協調並列パターンをモデル化する相互接続された計算のグラフとしてプログラムを表現できます。多くの場合、これらのアプリケーションは、一連のフィルターまたはステージを通じてデータをストリーミングします。これらを *データ・フローグラフ*と呼びます。グラフは、明示的にデータを渡すことなく操作間の事前発生（happens-before）関係を表現できるため、並列ループやパイプラインでは表現が困難な依存関係構造を表現できます。コレスキー分解など一部の線形代数計算では、小さな操作の依存関係を追跡することで重い同期ポイントを回避する効率良い並列実装があります。 このような事前発生関係を表すグラフを *依存関係グラフ*と呼びます。

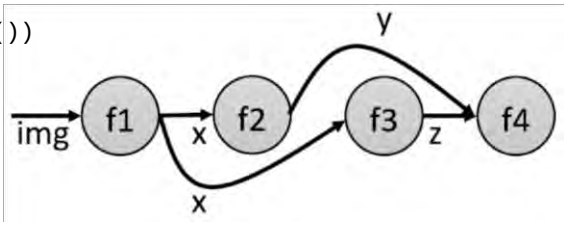
フローグラフは本質的に非同期です。メッセージは非ブロッキング関数によってノードに渡され、メッセージの到着に応じて TBB タスクが作成されます。利用可能な TBB ワーカースレッドが作成されたタスクの実行に参加します。最も一般的な使用例では、グラフにメッセージを送信する前に完全なグラフ構造を作成します。また、実行中にグラフ構造を動的に変更することもできます（詳細については、5 章を参照してください）。

並列性を表現するためにグラフを使用する理由

グラフはアプリケーションを表現するのに便利な方法です。ホワイトボードに計算間の関係をスケッチする場合、それらの関係を捉えるためグラフを使用することがあります。アプリケーションを計算グラフとして表現し、設計上の関係を C++ コードに変換します。これらの関係がコードで明示されたことで、グラフは実行時に並列計算のスケジューリングに使用できる情報を明らかにします。図 4-1(a) にコード例を示します。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_4

```
while  (auto  img = getImage())
{
  auto  x = f1(img);
  auto  y = f2(x);
  auto  z = f3(x);
  f4(y,z);
}
```



（a）ソースコード　　（b）グラフ・ダイアグラム

図 4-1. グラフとして表現された計算のシリアルシーケンス

図 4-1(a) の `while` ループの各反復では、画像が読み取られ、一連のフィルター `f1`、`f2`、`f3`、`f4` に渡されます。フィルター間のデータの流れは、図 4-1(b) のようになります。この図では、各関数から返されたデータを渡す変数は、値を生成するノードからその値を消費する後続のノードへのエッジに置き換えられています。例えば、`f1` によって返された値は図 4-1(a) の変数 `x` に格納され、この変数は `f2` と `f3` に渡されます。図 4-1(b) では、変数 `x` は 2 つのエッジに置き換えられています。1 つは `f1` から `f2` に流れ、もう 1 つは `f1` から `f3` に流れます。図 4-1(b) はまだ TBB フローグラフを表していません。分かりやすく TBB フローグラフで入力をペアにする場合（例えば、`y` と `z` を `f4` への入力としてペアにする）、ジョインノードを使用してそれらを結合する必要があります。

ここでは、図 4-1(b) のグラフがこれらの関数間で共有されるすべてのデータをキャプチャーしていると仮定します。そのため、私たち（そして TBB のようなライブラリー）は、図 4-2 に示すように、何を並列実行するのが正当であるか多くのことを推測できます。エッジはノードの部分的な順序付けを必須にします。

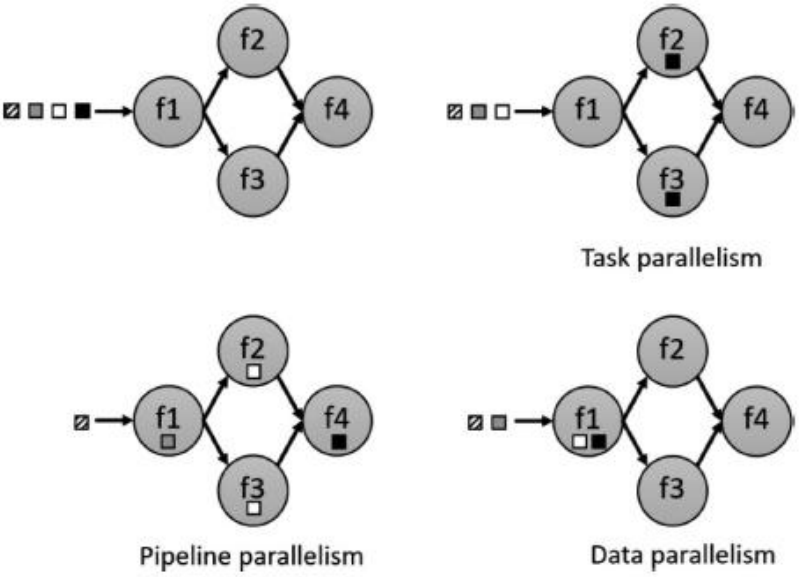


図 4-2.　図 4-1 のグラフから推測できる並列処理の種類。同じ塗りつぶしパターンを持つボックスは同じ入力メッセージに関連付けられており、必ずしも共有オブジェクトであるとは限りません。

例えば、タスク並列グラフの黒い実線は、図 4-1 のグラフに入った最初の `img` に `f1` を適用して生成された `x` 値のコピーを表しています

図 4-1(b) のグラフを介して 4 つの画像をストリーミングする場合、図 4-2 に示すように、TBB ライブラリーによって利用可能な並列処理の種類はいくつかあります。ノード `f2` と `f3` の間にはエッジがないため、並列に実行できます。同じデータに対して 2 つの異なる関数を並列に実行するのは、タスク並列処理の一例です。関数が共有グローバル状態を更新しない場合は、パイプラインの並列処理を利用して、グラフ内の異なる画像処理をオーバーラップすることもできます。最後に、関数が*スレッドセーフ*である場合、つまり、異なるデータに対して各関数をそれ自体と並列に実行しても安全な場合は、同じノードで 2 つの異なるイメージの処理をオーバーラップして、*データの並列性*を活用できます。

TBB フローグラフ・インターフェイスを使用すると、さまざまな種類の並列処理を活用するのに必要な情報がライブラリーに提供されるため、プラットフォームのハードウェアに最も効果的な方法で計算をマッピングできます。

TBB フローグラフ・インターフェイスの基本

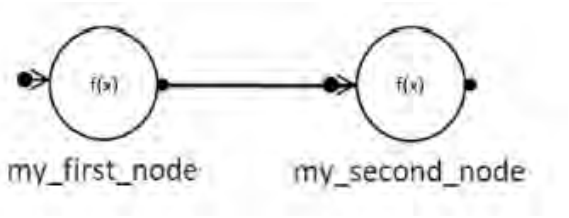
AI モデル、ニューラル・ネットワーク、または画像処理パイプラインを構築する機能を提供するドメイン固有のグラフのようなフレームワークとは異なり、TBB フローグラフは汎用のジェネリック API です。これらはアプリケーションの構築に使用できますが、ドメイン固有の機能やアルゴリズム自体は提供されません。接続された計算グラフを構築できますが、それらの計算を実装する必要があります。多くの場合、フローグラフ・ノードからドメイン固有の関数を呼び出すだけで実現できます。

TBB フローグラフ・クラスと関数は `flow_graph.h` で定義され、`tbb::flow` 名前空間に含まれます。包括的な `tbb.h` では `flow_graph.h` もインクルードさているため、他に何もインクルードする必要はありません。

フローグラフを使用するには、*グラフ・オブジェクト*を作成し、操作を実行する*ノード*を作成し、これらのノード間のメッセージチャンネルまたは依存関係を表現する*エッジ*を作成します。フローグラフ API は非同期かつ即時的（`eager`）であり、グラフの実行中にエッジの追加と削除が可能です。ただし、簡素化のため、この節では、グラフ内でワークを実行する前にグラフを完全に定義する最も単純なケースについて説明します。このシナリオでは、5 つのステップでグラフを使用します:

（1）グラフ・オブジェクトを作成します。
（2）ノードを作成します。
（3）ノードの入力ポートと出力ポートの間にエッジを追加します。
（4）メッセージをグラフに直接、または `input_node` オブジェクトをアクティブにして送信します。
（5）生成された結果を使用できるようにグラフが完了するのを待機します。

図 4-3 は、この 5 つのステップを実行する小規模な例を示しています。この節では、これらの各ステップについて詳しく説明します。この章の最初の例では、出力を表示するのに `std::cout` を使用していることに注意してください。これは、ノードが実際には副作用がないわけではないことを意味します。並列処理をさらに増やすと、出力が混乱する可能性があることが分かります。ただし、図 4-3 では、グラフを流れるメッセージは 1 つだけであり、ノードの実行は重複しないため、`std::cout` オブジェクトをノード間で共有しても問題ありません。



単純な 2 つのノードグラフの図

```
void  graphTwoNodes() {
  // ステップ 1: グラフを構築
  tbb::flow::graph g;

  // ステップ 2: ノードを作成
  tbb::flow::function_node<int, std::string> my_first_node{g,
    tbb::flow::unlimited,
    []( const  int&  in  ) {
      std::cout << "first node received: "  << in << std::endl;
      return  std::to_string(in);
    }
  };

  tbb::flow::function_node<std::string> my_second_node{g,
    tbb::flow::unlimited,
    []( const  std::string&  in  ) {
      std::cout << "second node received: "  << in << std::endl;
    }
  };

  // ステップ 3: エッジを追加
  tbb::flow::make_edge(my_first_node, my_second_node);

  //
  my_first_node.try_put(10);

  // ステップ 5: グラフの完了を待機
  g.wait_for_all();
}
```

図 4-3. 単純な 2 つのノードのフローグラフ。この例では、ノードは、入力が `int` などの基本型であっても、`const` 参照として入力を受け取ります。実際のコードでは、フローグラフ・メッセージが基本型になることはほとんどありません。サンプル以外のコードで `int` メッセージを使用する場合、代わりに値で受信します。どちらのオプションも有効です。ただし、大きなメッセージの場合は、ほとんどの場合、コピーを避ける方が適切であるため、ここでは `const` 参照を使用して、予想されるパターンを強調します。サンプルコード `graph/graph_two_nodes.cpp`

ステップ 1: グラフ・オブジェクトを作成

フローグラフのインターフェイスにおいて、グラフ・オブジェクトは、グラフの実行に関連するすべてのタスクが完了するのを待機したり、グラフ内の全ノードの状態をリセットしたり、グラフ内の全ノードの実行をキャンセルするなど、グラフ全体にわたる操作を呼び出すために使用されます。グラフを構築する場合、各ノードは 1 つのグラフに属し、エッジは同じグラフのノード間でのみ作成されます。異なるグラフ内のノード間にエッジを作成すると、未定義の動作となります。グラフ・オブジェクトは、ノードが追加される前に作成される必要があり、また、グラフに代わって実行されるすべてのタスクとそれに関連付けられたすべてのノードよりも長く存続する必要があります。コピー・コンストラクターと代入演算子が削除されているため、グラフ・オブジェクトをコピーすることはできません。図 4-4 は、グラフクラスの主要なパブリック関数を示しています。

```
namespace tbb {
namespace flow {

  class graph : no_copy {
  public:

    //! 分離された task_group_context を持つグラフを構築 (9 章を参照)
    graph();

    //! use_this_context をコンテキストとしてグラフを構築
    explicit graph(task_group_context&
    use_this_context);

    //! グラフを破棄。
    ~graph();

    //! グラフがアイドル状態になり、
    //! release_wait 呼び出しの回数が
    //! reserve_wait 呼び出しの数と等しくなるまで待機。
    void wait_for_all();

    // スレッドアンセーフ状態をリセット。
    void reset(reset_flags f = rf_reset_protocol);

    //! 関連する task_group_context の実行をキャンセルし、
    void cancel();

    //! グラフ実行のステータスを返します
    bool is_cancelled();
    bool exception_thrown();

  }; // class graph

} }
```

図 4-4. [flow_graph.graph] で記述されているクラスグラフの主要なメンバー関数

ステップ 2: ノードの構築

TBB フローグラフ API は、豊富なノードタイプ（図 4-5）を定義しており、これは機能ノードタイプ、結合ノードタイプ、バッファリング・ノード・タイプ、および制御フロー・ノード・タイプの 4 つのグループに分類されます。図 4-5 で使用されている記号は、図のノードタイプを明確にするため本書全体で使用しています。

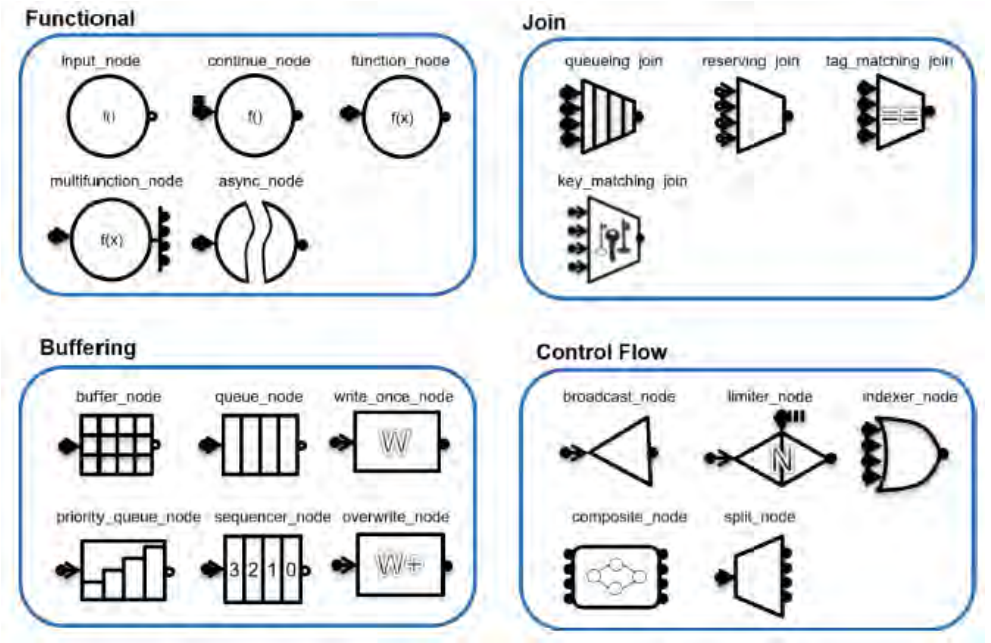


図 4-5.　フローグラフ API のノードの種類

機能ノード

機能ノードはフローグラフ API の中核です。これらのノードはユーザー定義のコードを実行します。すべての機能ノード・コンストラクターは、少なくともグラフ・オブジェクト引数と、ユーザーが提供する関数オブジェクトを受け取ります。機能ノードは、どのような受信メッセージに応答するか（応答する場合）、ユーザー関数オブジェクト（ボディー）に期待されるシグネチャー、およびユーザーが提供する計算出力が他のノードに伝播される方法が異なります。

表 4-1 では、サポートされる入力と出力ポートの数、到着メッセージの処理ポリシー、入力メッセージまたは出力メッセージをバッファリングするか、ノードレベルの優先順位をサポートするか、ノードの同時実行性を制限できるかなど、各ノードタイプのプロパティーについて説明します。

ノードレベルの優先順位、ポリシー、同時実行制限については、5 章でさらに詳しく説明します。

表 4-1. フローグラフ API の機能ノードタイプ。ポートはエッジの接続ポイントとして機能します。1 つの出力ポートを複数の後続ポートに接続でき、入力ポートを複数の先行ポートに接続できます。`multifunction_node` は複数の出力ポートを持つことができ、それぞれが異なるタイプのメッセージを送信し、各ポートは 0 個以上の後続ポートに接続できます

ノード	入力ポート	出力ポート	ポリシー	バッファリング	優先サポート	同時実行数の制限
`input_node`	0	1	N/A	出力ポート	いいえ	いいえ、シリアル
`continue_node`	1	1	デフォルト、軽量	なし	はい	いいえ、無制限
`function_node`	1	1	デフォルト、軽量、キューイング、拒否	ポリシーが拒否されていない場合の入力ポート	はい	はい
`multifunction_node`	1	0…N	デフォルト、軽量、キューイング、拒否	ポリシーが拒否されていない場合の入力ポート	はい	はい
`async_node`	1	1	デフォルト、軽量、キューイング、拒否	ポリシーが拒否されていない場合の入力ポート	はい	はい

メッセージに適用するコードを指定するには、機能ノード内の body 引数を使用します。表 4-2 では、これらのボディー・オブジェクトの必須シグネチャーと、このシグネチャーに対して oneTBB 仕様で使用される名前付き要件について説明します。

表 4-2. ユーザーが提供したボディーに必要なシグネチャー

ノード	名前付き要件	ボディー・シグネチャー
input_node	InputNodeBody	Output operator()(flow_control& fc)
continue_node	ContinueNodeBody	Output operator()(const continue_msg& v)
function_node	FunctionNodeBody	Output operator()(const Input& v)
multifunction_node	MultifunctionNodeBody	void operator()(const Input& v, OutputPortsType& p)
async_node	AsyncNodeBody	void operator()(const Input& v, GatewayType& gateway

図 4-3 では、my_first_node が int 値を受け取り、その値を出力し、std::string に変換して、変換された値を返すように定義しました。図 4-3 の例では、オブジェクトの構築時にすべてのテンプレート引数を指定していますが、TBB には、より優れたクラス・テンプレート引数推論（CTAD）をサポートする推論ガイドがあります。CTAD および推論ガイドの詳細については、https://en.cppreference.com/w/cpp/language/class_template_argument_deduction を参照してください。C++17 以降を使用する場合、多くの場合、フローグラフ・クラスのテンプレート引数は推測できるため、明示的に指定する必要はありません。例えば、図 4-6 に示すように、テンプレート引数や戻り値の型を直接指定せずに、my_first_node を構築することもできます。

```
tbb::flow::function_node  my_first_node{g,
  tbb::flow::unlimited,
  []( const  int& in  ) {
    std::cout << "first node received: "  << in << std::endl;
    return  std::to_string(in);
  }
};
```

図 4-6. 引数または戻り値の型を直接指定せずに構築された my_first_node。サンプルコード: graph/graph_two_nodes_deduced.cpp

function_node の重要なメンバー関数を図 4-7 に示します。これには、コンストラクターと関数 try_put が含まれます。try_put 関数を呼び出すことで、フローグラフ内の任意のノードにメッセージを直接渡すことができますが、通常、メッセージはエッジを介して暗黙的に渡されます。通常、try_put を明示的に呼び出すのは、グラフへの入力として機能するノードにメッセージを送信する場合のみです。

例えば、図 4-3 では、my_first_node に対して try_put を呼び出しましたが、my_second_node では呼び出していません。my_second_node は、my_first_node からの出力エッジを介して入力を受け取ります。

```
namespace tbb {
namespace flow {

  template < typename Input,
  typename Output = continue_msg,
             typename Policy = /*実装定義*/ >
  class function_node : public graph_node,
                        public receiver<Input>,
                        public sender<Output> {
  public:
    //! Constructor
    template<typename Body>
    function_node( graph& g, size_t concurrency, Body body,
                   Policy /*未指定*/ = Policy(),
                    node_priority_t priority = no_priority );

    template<typename Body>
    function_node( graph& g, size_t concurrency, Body body,
                   node_priority_t priority =
                   no_priority );

    ~function_node();

    //! コピー・コンストラクター
    function_node( const function_node& src );

    //! ノードにメッセージを明示的に渡す bool
    try_put( const Input& v );
  };

}
}
```

図 4-7. [flow_graph.function_node] で説明されているクラス function_node の主要メンバー関数

function_node は単一の入力値から単一の出力値を生成しますが、表 4-2 に示すように、フローグラフで使用できる他の種類の機能ノードがあります。

ジョインノード

join_node は複数の入力ポートからの入力を結合して出力 std::tuple を作成します。join_node の主要メンバーを図 4-8 に示します。join_node には、表 4-3 で説明されているように、queueing、reserving、key_matching、tag_matching の 4 つのジョインポリシーのいずれかを設定できます。

```
namespace tbb {
namespace flow {
  using tag_value = /*実装定義*/;
  // JoinNodePolicies:
  struct reserving;
  struct queueing;
  template<typename K,
           class KHash=tbb_hash_compare<K> > struct key_matching;
  using tag_matching = key_matching<tag_value>;

  template<typename OutputTuple,
           class JoinPolicy = /*実装定義*/>
  class join_node {
   public:
    using input_ports_type = /*実装定義*/;

    explicit join_node( graph& g );
    join_node( const join_node& src );

    input_ports_type& input_ports();

    bool try_get( OutputTuple& v );

    template<typename OutputTuple, typename K,
             class KHash=tbb_hash_compare<K> >
    class join_node< OutputTuple, key_matching<K, KHash> > {
    public:
      using input_ports_type = /*実装定義*/;

      explicit join_node( graph& g );
      join_node( const join_node& src );

      template<typename ...TagBodies>
      join_node( graph& g, TagBodies ...);

      input_ports_type& input_ports();

      bool try_get( OutputTuple& v );
    };
} }
```

図 4-8. [flow_graph.join_node] で説明されているジョインノードの主な型と関数

表 4-3. [flow_graph.join_node_policies] で説明されている join_node で使用可能なジョインポリシー

ジョインポリシー	バッファリング	説明
queueing	各入力ポートの FiFO キュー	受信メッセージをポートごとのキューに保存し、先入先出方式を使用してメッセージをタプルに結合します。
key_matching	入力ポート間のハッシュテーブル	受信メッセージをポートごとのマップに保存し、一致するキーに基づいてメッセージを結合します。
tag_matching	入力ポート間のハッシュテーブル	受信メッセージをポートごとのマップに保存し、一致する tag_value タグ（符号なし整数）に基づいてメッセージを結合します。これは、key_matching の特殊化です。
reserving	メッセージをバッファリングせず、入力の（可能な）可用性を記録	各入力ポートのメッセージの可用性の既知の状態を保存します。 すべてのポートが利用可能としてマークされている場合、各ポートに対してメッセージを予約します。 失敗すると、そのポートのマークを解除し、以前に取得したすべての予約を解放します。成功した場合、これらのメッセージを含むタプルをすべての後続ノードにブロードキャストします。少なくとも 1 つの後続ノードがタプルを受け入れると、予約は消費され、それ以外の場合は解放されます。

j に続く join_node には、3 つの入力ポートと 1 つの出力ポートがあります。入力ポート 0 は int 型のメッセージを受け入れます。入力ポート 1 は、std::string 型のメッセージを受け入れます。入力ポート 2 は、double 型のメッセージを受け入れます。受信したメッセージを結合し、std::tuple<int, std::string, double> 型のメッセージをブロードキャストする単一の出力ポートがあります:

```
tbb::flow::join_node<std::tuple<int, std::string, double>,
                     tbb::flow::queueing> j(g);
```

queueing、key_matching、および tag_matching ポリシーの場合、join_node は各入力ポートに到着した受信メッセージをバッファリングします。これらのポリシーでは、受信メッセージを拒否することはありません。

特別なケースとして、予約 join_node は受信メッセージを全くバッファリングしません。代わりに、先行するバッファーの状態を追跡します。各入力ポートに利用可能なメッセージがあることを検出すると、各入力ポートの項目を予約します。 予約が保持されている間は他のノードは項目を消費できなくなります。join_node が各入力ポートの要素の予約を正常に取得できた場合にのみ、これらのメッセージを消費します。それ以外は、すべての予約を解放し、メッセージを先行するバッファーに残します。予約中の join_node がすべての入力を予約できなかった場合、1 つ以上の先行ノードの状態の変化を検出すると、後で再試行します。予約ポリシーは、5 章の「トークンによる繰り返し制御」で詳しく説明されているよう

に、メッセージの生成を制限するのに使用されます。

図 4-9 と 4-10 には、double と std::string からタプルを作成する join_node を含むグラフの例を示しています。

```
#include  <iostream>
#include  <tbb/tbb.h>

void  graphJoin() {
  // ステップ 1: グラフを構築
  tbb::flow::graph g;

  // ステップ 2: ノードを作成
  tbb::flow::function_node my_node{g,
   tbb::flow::unlimited,
   []( const  int&  in  ) {
      std::cout << "received: "  << in <<
      std::endl; return  std::to_string(in);
    }
  };

  tbb::flow::function_node my_other_node{g,
    tbb::flow::unlimited,
    [](const  int&  in) {
      std::cout << "other received: "  << in << std::endl;
      return  double(in);
    }
  };

  tbb::flow::join_node<std::tuple<std::string, double>>
  my_join_node{g};

  tbb::flow::function_node my_final_node{g,
    tbb::flow::unlimited,
    [](const  std::tuple<std::string, double>&  in) {
     std::cout << "final: "  << std::get<0>(in)
               << " and "  << std::get<1>(in) <<
      std::endl; return  0;
    }
  };

  // ステップ 3: エッジを追加
  make_edge(my_node, tbb::flow::input_port<0>(my_join_node));
  make_edge(my_other_node, tbb::flow::input_port<1>(my_join_node));
  make_edge(my_join_node, my_final_node);

  // ステップ 4: メッセージを送信
  my_node.try_put(1);
  my_other_node.try_put(2)
  ;
  // ステップ 5: グラフの完了を待機
  g.wait_for_all();
}
```

図 4-9. 図 4-10 に示すグラフの実装。機能ノードの入力と出力の型を推測するため、CTAD に依存していることに注意してください。サンプルコード: graph/graph_with_join.cpp

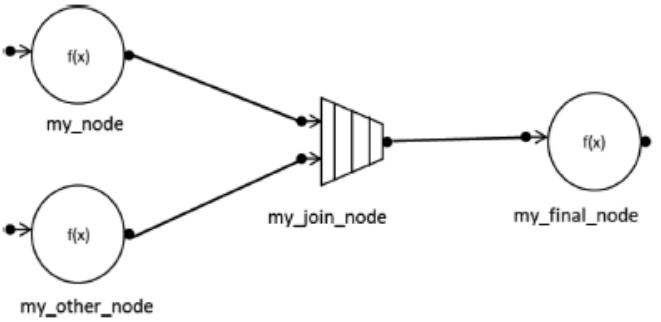


図 4-10. join_node を持つ 4 つのノードグラフ

バッファリング・ノード

バッファリング・ノードは、限定的ではありますが重要な状況で使用されます。5 章では、リソース使用量の制限、タスクのオーバーヘッドの管理、繰り返しの制御、グラフ構造の動的な変更など、さらに高度なトピックについて説明します。バッファリング・ノードはこのトピックで重要な役割を果たします。これらの重要なノードについては、この章で説明します。

制御フローノード

最後のノードのカテゴリーは制御フローノードです。broadcast_node は、単純に受信メッセージをすべての後続ノードに転送します。この種のノードは、グラフまたはグラフ・セクションへの単一のエントリーポイントを作成する場合に役立ちます。limiter_node は、グラフ内のポイントを通過できるメッセージ数を制御するために使用され、通常はメモリー使用量を制限するのに使用されます。indexer_node は、グラフ内の分離パスを結合するために使用され、メッセージが到着したポート番号がタグ付けられたメッセージを、一度に 1 つずつ後継ノードに転送します。split_node は、入力タプルを構成要素に分割し、各要素を異なる出力ポートを通じて転送します。最後に、composite_node はサブグラフを単一のノードとしてカプセル化するのに使用され、より大きなグラフに簡単に統合できるようになります。

バッファリング・ノードと同様に、制御フローノードは、5 章で説明されている重要なパターンと最適化によく使用されます。

ステップ 3: エッジを追加

図 4-3 と 4-9 で見たように、グラフ・オブジェクトとノードを構築した後、make_edge を呼び出してメッセージチャンネルまたは依存関係を設定します:

```
make_edge(predecessor_node, successor_node);
```

ノードに複数の入力または出力ポートがある場合、ノード自体ではなく特定のポートで make_edge を呼び出す必要があります。これは、input_port および output_port 関数テンプレートを使用して実行できます:

```
make_edge(tbb::flow::output_port<0>(predecessor_node),
          tbb::flow::input_port<1>(successor_node));
```

図 4-3 では、単純な 2 ノードグラフで my_first_node と my_second_node の間にエッジを作成しました。図 4-9 は、4 つのノードを持つ、少し複雑なフローグラフを示しています。

図 4-9 の最初の 2 つのノードは、キューの join_node、my_join_node によってタプルに結合される結果を生成します。join_node の入力ポートにエッジを作成する場合、ポート番号を指定する必要があります:

```
make_edge(my_node, tbb::flow::input_port<0>(my_join_node));
make_edge(my_other_node, tbb::flow::input_port<1>(my_join_node));
```

join_node の出力である std::tuple<std::string, double> は、my_final_node に送信されます。ポートが 1 つしかない場合、ポート番号を指定する必要はありません:

```
make_edge(my_join_node, my_final_node);
```

ステップ 4: グラフにメッセージを送信

TBB フローグラフを作成して使用する 4 番目の手順は、グラフにメッセージを送信してグラフの実行を開始することです。メッセージがグラフに入る方法は 2 つあります。(1) ノードへの明示的な try_put を介して、または (2) input_node の出力としてです。図 4-3 と 4-9 の両方で、ノード上で try_put を呼び出して、グラフへのメッセージの入力を開始します。

例えば、図 4-9 では、try_put を直接呼び出して my_node にメッセージを送信します:

```
my_node.try_put(1);
```

これにより、TBB ライブラリーは int メッセージ 1 に対して my_node のボディーを実行するタスクを生成し、「received: 1」などの出力が出力されます。内部ノードを含め、どのノードの任意の入力ポートで try_put を呼び出すことはできますが、グラフのルートノードでのみ try_put を呼び出すのが一般的です。

グラフにメッセージを送信する 2 番目の方法は、input_node を使用することです。input_node はデフォルトでは非アクティブな状態で構築されます。つまり、すぐにはメッセージの送信を開始しません。これにより、最も一般的な使用シナリオでグラフが完全に構築される前にメッセージが送信されるのを防ぎます。グラフの構築が完了した後にメッセージを送信するには、すべての input_node オブジェクトで activate() 関数を呼び出す必要があります。

図 4-11 は、シリアルループの代わりに input_node を使用してグラフにメッセージを送信する方法を示しています。図 4-11(a) では、ループがノード my_node に対して try_put を繰り返し呼び出し、メッセージを送信しています。図 4-11(b) では、同じ目的で input_node が使用されています。

表 4-2 で説明されているように、input_node ボディーは、**InputNodeBody** の名前付き要件を満たす必要があります。input_node はボディーを適用して次の項目を生成します。新しい要素を生成できない場合、ボディーは fc.stop() を呼び出して入力が完了したことを知らせます。fc.stop() が呼び出されると、有効な出力値が返され、すぐに破棄されます。fc.stop() が呼び出された後、グラフまたはノードがリセットされない限り、ボディーは再び呼び出されません。図 4-11(b) では、input_node ボディーは送信したメッセージの数を追跡する内部状態（カウント）を維持し、事前定義された制限に達すると fc.stop() を呼び出します。

```
void tryPutLoop() {
  const int limit = 3;
  tbb::flow::graph g;
  tbb::flow::function_node my_node{g, tbb::flow::unlimited,
    [](const int& i) {
      std::printf("my_node: %d\n", i);
      return 0;
    }
  };
  for (int count = 0; count < limit; ++count) {
    my_node.try_put(count);
  }
  g.wait_for_all();
}
```

（a）ループ内で `try_put` 呼び出しを使用する。

```
void inputNodeLoop() {
  tbb::flow::graph g;

  tbb::flow::input_node my_input{g,
    [](tbb::flow_control& fc) {
      const int limit = 3;
      static int count = 0;
      if (count >= limit)
        fc.stop();
      return
      count++;
    }
  };
  tbb::flow::function_node my_node{g,
    tbb::flow::unlimited,
    [](const int& i)
      { std::printf("my_node: %d\n", i);
      return 0;
    }
  };

  tbb::flow::make_edge(my_input, my_node);

  my_input.activate();

  g.wait_for_all();
}
```

（b）`input_node` を使用してメッセージを送信します。

図 4-11.（a）ループを使用して `int` 値 0、1、2 をノード `my_node` に送信し、（b）`input_node` は `int` 値 0、1、2 をノード `my_node` に送信します。サンプルコード: `graph/graph_loops.cpp`

ループの代わりに input_node を使用する利点は、グラフ内の他のノードに応答できることです。5 章では、input_node を予約済みの join_node または limiter_node と組み合わせて使用して、グラフ領域に入力できるメッセージ数を制御する方法のヒントについて説明します。単純なループを使用すると、意図せずグラフに入力が集中し、ノードが対応できずに多くのメッセージをバッファリングせざるを得なくなる可能性があります。

この章のすべての例では、メッセージを送信する前にグラフ構造を完全に定義します。TBB フローグラフは、即時的な実行モデルを使用するため、グラフ表現を実行可能形式に変換する最終処理ステップはありません。TBB フローグラフは常に実行可能であり、メッセージを受信する準備ができています。ただし、すべてのノードを構築し、すべてのエッジを意図どおりに接続する前にグラフにメッセージを送信すると、その時点で表現されているグラフの実行が開始されます。TBB フローグラフのこのプロパティーは、グラフを動的に拡大および変更するのに使用できるため強力ですが、慎重に実行しないと予期しない結果につながる可能性があるため、問題になることもあります。5 章の後半では、グラフを安全かつ動的に拡張する方法について説明します。

ステップ 5: グラフの実行が完了するまで待機する

try_put または input_node を使用してグラフにメッセージを送信したら、グラフ・オブジェクトで wait_for_all() を呼び出してグラフの実行が完了するまで待機できます。これらの呼び出しは、図 4-3、4-9、および 4-11 で確認できます。wait_for_all を呼び出さずにノードまたはグラフを破棄すると、未定義の動作が発生します。

図 4-3 のグラフを作成して実行すると、次のような出力が表示されます。

```
first node received: 10
second node received: 10
```

図 4-9 のグラフを作成して実行すると、次のような出力が表示されます。

```
other received: received: 21
final: 1 and 2
```

図 4-9 の出力は少し乱れているように見えますが、実際乱れています。最初の 2 つの機能ノードは並列に実行され、両方とも std::cout にストリーミングされます。出力では、2 つの出力が混在して表示されていますが、これは、この章の前半でグラフベースの並列処理について説明した際の仮定に反したためです。つまり、ノードには副作用がないわけではないのです。これら 2 つのノードは並列に実行され、どちらもグローバル std::cout オブジェクトの状態に影響を与えます。この例では、この出力はグラフ上のメッセージの進行状況を示すためだけに印刷されるため、問題ありません。しかし、これは知っておくべき重要なポイントです。

図 4-9 の最後の function_node は、先行する機能ノードの両方の値が join_node によって結合され、それに渡された場合にのみ実行されます。したがって、この最終ノードはそれ自体で実行され、予想される最終出力を std::cout: "final: 1 and 2" にストリーミングします。

データ・フローグラフのより複雑な例

2 章では、左画像と右画像のペアに赤とシアンの 3D 立体効果を適用する例を紹介しました。2 章では、TBB の parallel_pipeline を使用して例を並列化しましたが、その際にパイプライン・ステージを線形化することで、並列性を十分に活用しきれていないことを認めました。出力例を図 2-35 に示します。それではフローグラフを使ってみましょう。

立体 3D サンプルを実装する TBB フローグラフの構築を段階的に進めます。作成するフローグラフの構造を図 4-12 に示します。この図は図 2-36 とは異なって見えます。これは、ノードが TBB フローグラフ・ノード・オブジェクトを表し、エッジが TBB フローグラフ・エッジを表しているためです。

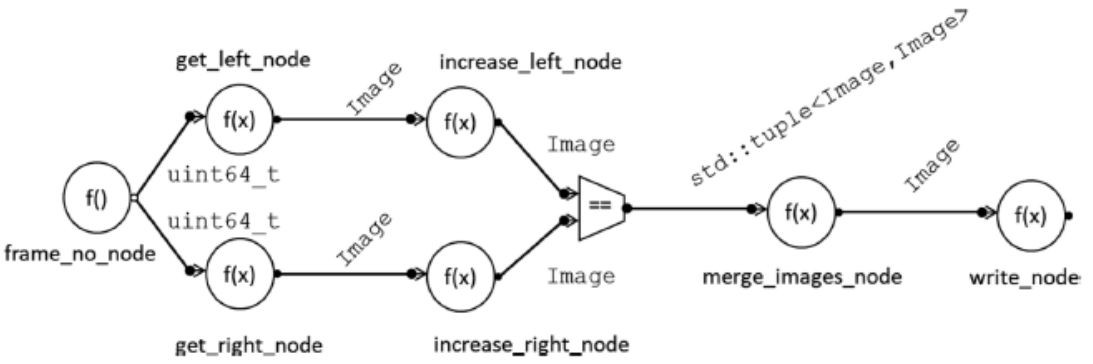


図 4-12.　図 2-36 の呼び出しを表すグラフ。サンプルコード graph/graph_stereoscopic_3d.cpp

図 4-13 は、TBB フローグラフ・インターフェイスを使用して実装された立体 3D の例を示しています。5 つの基本的な手順がボックスに示されています。まず、グラフ・オブジェクトを作成します。次に、input_node、いくつかの function_node インスタンス、および join_node を含む 8 つのノードを作成します。ノード作成の詳細を図 4-14 に示します。次に、make_edge を呼び出してノードを接続します。エッジを作成した後、input_node をアクティブ化します。最後に、グラフが完成するのを待機します。

前述したように、getLeftImage 関数と getRightImage 関数は独立して実行されますが、ファイルまたはカメラから画像を順番に読み取るため、順番に実行されます。図 4-13 のコードでは、これらのノードの同時実行の制約を flow::serial に設定することで、この制約をランタイム・ライブラリーに通知します。

対照的に、increase_left_node オブジェクトと increase_right_node オブジェクトは、flow::unlimited の同時実行の制約に従って構築されます。ランタイム・ライブラリーは、受信メッセージが到着するたびに、ノードのボディーを実行するタスクを生成します。

図 4-12 では、merge_images_node 関数には右左の画像の両方が必要であることが分かります。オリジナルのシリアルコードでは、while ループは一度に 1 つのフレームのみを処理するため、画像が同じフレームから取得されることが保証されていました。ただし、フローグラフ・バージョンでは、複数のフレームがフローグラフを通じてパイプライン化される可能性があるため、同時に進行中になる可能性があります。merge_images_node に一致する左右の画像のペアを提供するには、tag_matching ポリシーを使用して join_images_node を作成します。コンストラクターの呼び出しには、2 つの入力ポートの受信メッセージからタグ値を取得する 2 つのラムダ式が含まれています。merge_images_node はタプルを受け入れ、単一の結合された画像を生成します。

図 4-14 の最後に作成されたノードは write_node です。これは、Image オブジェクトを受け取り、write を呼び出して各受信バッファーを出力ファイルに保存する flow::unlimited function_node です。

```
// ステップ 1: グラフ・オブジェクトを構築
tbb::flow::graph g;
```

```
// ステップ 2: ノードを作成
// 図 4-14 を参照
```

```
// ステップ 3: エッジを追加
tbb::flow::make_edge(frame_no_node, get_left_node);
tbb::flow::make_edge(frame_no_node, get_right_node);
tbb::flow::make_edge(get_left_node, increase_left_node);
tbb::flow::make_edge(get_right_node, increase_right_node);
tbb::flow::make_edge(increase_left_node,
                     tbb::flow::input_port<0>(join_images_node));
tbb::flow::make_edge(increase_right_node,
                     tbb::flow::input_port<1>(join_images_node));
tbb::flow::make_edge(join_images_node, merge_images_node);
tbb::flow::make_edge(merge_images_node, write_node);
```

```
// ステップ 4: メッセージをグラフに送信
frame_no_node.activate();
```

```
// ステップ 5: グラフの完了を待機
g.wait_for_all();
```

図 4-13. TBB フローグラフとしての立体 3D の例。フローグラフを使用するため 5 つの手順が強調表示されます。サンプルコード: graph/graph_stereoscopic_3d.cpp

```
// ステップ 2: ノードを作成
tbb::flow::input_node<uint64_t> frame_no_node{g,
  []( tbb::flow_control  &fc  ) -> uint64_t  {
    uint64_t  frame_number = getNextFrameNumber();
    if  (frame_number)
      return  frame_number;
    else  {
      fc.stop();
      return  frame_number;
} } };
tbb::flow::function_node<uint64_t, Image> get_left_node{g,
  /* 並行性 */  tbb::flow::serial,
  [] (uint64_t  frame_number) -> Image  {
    return  getLeftImage(frame_number);
} };
tbb::flow::function_node<uint64_t, Image> get_right_node{g,
  /* 並行性 */  tbb::flow::serial,
  [] (uint64_t  frame_number) -> Image  {
    return  getRightImage(frame_number);
} };
tbb::flow::function_node<Image, Image> increase_left_node{g,
  /* 並行性 */  tbb::flow::unlimited,
  [] (Image  left) -> Image  {
      increasePNGChannel(left, Image::redOffset, 10);
      return  left;
} };
tbb::flow::function_node<Image, Image> increase_right_node{g,
  /* 並行性 */  tbb::flow::unlimited,
  [] (Image  right) -> Image  {
      increasePNGChannel(right, Image::blueOffset, 10);
      return  right;
} };
tbb::flow::join_node<std::tuple<Image, Image>, tbb::flow::tag_matching >
  join_images_node(g, [] (Image  left) { return  left.frameNumber; },
                      [] (Image  right) { return  right.frameNumber; } );
tbb::flow::function_node<std::tuple<Image, Image>, Image>
merge_images_node{g,
  /* 並行性 */  tbb::flow::unlimited,
  [] (std::tuple<Image, Image> t) -> Image  {
    auto& l = std::get<0>(t);
    auto& r = std::get<1>(t);
    mergePNGImages(r, l);
    return  r;
} };
tbb::flow::function_node<Image> write_node{g,
  /* 並行性 */  tbb::flow::unlimited,
  [] (Image  img) {
    img.write();
} };
```

図 4-14. 立体 3D の例の「ノードの作成」手順(図 4-13 を参照)。サンプルコード: graph/graph_stereoscopic_3d.cpp

依存関係グラフの実装

依存関係グラフを使用する手順は、データ・フローグラフと同じです。グラフ・オブジェクトを作成し、ノードを作成し、エッジを追加して、メッセージをグラフに入力します。主な違いは、機能ノードが `continue_node` オブジェクトであること、グラフが非循環である必要があること、およびグラフにメッセージを送るたびにグラフの実行完了を待機する必要があることです。依存関係グラフの利点は、メッセージを `std::tuple` オブジェクトに結合するのではなく、カウンターをインクリメントおよびデクリメントするメッセージを渡すため、オーバーヘッドが低いことです。`continue_node` オブジェクトは、受信したメッセージ数を単純にカウントするため、依存関係グラフで使用されます。データ・フローグラフで `continue_node` を使用するのは合法ですが、役立つ用途はまれです。

この節では、グラフが開始される前にグラフが完全に構築されます。5 章では、`write_once` ノードを使用して依存関係グラフの同時構築と実行を行う例について説明します。

依存関係グラフの例を作成しましょう。この例では、TBB の `parallel_for_each` を使用して、2 章で実装したのと同じ前方置換の例を実装します。2 章では、シリアルおよび `parallel_for_each` ベースの例の詳しい説明を参照できます。

依存関係グラフを使用すると、依存関係を直接表現し、TBB ライブラリーがグラフ内で利用可能な並列処理を検出して活用できるようになります。2 章の `parallel_for_each` バージョンのように、カウントを維持したり、明示的に完了を追跡する必要がなく、不要な同期ポイントも必要ありません。

図 4-15 は、この例の依存関係グラフバージョンを示しています。`std::vector` を使用して、`continue_node` オブジェクトへの共有ポインターのセットを保持します。各ノードは反復のブロックを表します。グラフを作成するには、次の一般的なパターンに従います:

（1）グラフ・オブジェクトを作成します。
（2）ノードを作成します。
（3）エッジを追加します。
（4）グラフにメッセージを送信します。
（5）グラフが完成するのを待ちます。

ただし、図 4-15 に示すように、ループネストを使用してグラフ構造を作成します。`createNode` 関数は各ブロックに対して新しい `continue_node` オブジェクトを作成し、`addEdges` 関数はノードを、その完了を待機する隣接ノードに接続します。

```
using  Node  = tbb::flow::continue_node<tbb::flow::continue_msg>;
using  NodePtr  = std::shared_ptr<Node>;

void  graphFwdSub(std::vector<double>&  x,
                  const  std::vector<double>&  a,
                  std::vector<double>&  b) {
  const  int  N = x.size();
  const  int  block_size = 1024;
  const  int  num_blocks = N / block_size;

  std::vector<NodePtr> nodes(num_blocks*num_blocks);
  tbb::flow::graph g;
  for  (int  r = num_blocks -  1; r >= 0; --r) {
    for  (int  c = r; c >= 0; --c) {
      nodes[r*num_blocks + c] = createNode(g, r, c, block_size, x, a, b);
      addEdges(nodes, r, c, block_size, num_blocks);
    }
  }
  nodes[0]->try_put(tbb::flow::continue_msg{});
  g.wait_for_all();
}
```

図 4-15. 前方置換の例の依存関係グラフ実装。フローグラフを使用するため 5 つの手順が表示されます。サンプルコード: graph/graph_fwd_substitution.cpp

図 4-16 に、**createNode** の実装を示します。図 4-17 に、**addEdges** の実装を示します。

```
NodePtr  createNode(tbb::flow::graph&  g,
                    int  r, int  c, int
                    block_size,
                    std::vector<double>&  x,
                    const  std::vector<double>&
                    a, std::vector<double>&  b) {
  const  int  N = x.size();
  return  std::make_shared<Node>(g,
    [r, c, block_size, N, &x, &a, &b] (const  tbb::flow::continue_msg&  msg)
      {
      int  i_start = r*block_size, i_end = i_start + block_size;
      int  j_start = c*block_size, j_max = j_start + block_size -  1;
      for  (int  i = i_start; i < i_end; ++i) {
        int  j_end = (i <= j_max) ? i : j_max + 1;
        for  (int  j = j_start; j < j_end; ++j) {
          b[i] -= a[j + i*N] * x[j];
        }
        if  (j_end == i) {
          x[i] = b[i] / a[i + i*N];
        }
      }
      return  msg;
    }
  );
}
```

図 4-16. createNode の実装。サンプルコード: graph/graph_fwd_substitution.cpp

createNode で作成された continue_node オブジェクトは、2 章に示した前方置換のブロックバージョンから内部の 2 つのループをカプセル化するラムダ式を使用します。依存関係グラフのエッジ間でデータは渡されないため、各ノードに必要なデータは、ラムダ式によってキャプチャーされたポインターを使用して共有メモリー経由でアクセスされます。図 4-16 では、ノードは整数 r、c、N、block_size とベクトル x、a、b への参照を値によってキャプチャーします。

図 4-17 では、関数 addEdges は、make_edge を呼び出して各ノードをその右および下の隣接ノードに接続します。これは、実行する前に新しいノードが完了するまで待機する必要があるためです。図 4-15 のループネストが完了すると、2 章で示したものと同様の依存関係グラフが構築されます。

```
void  addEdges(std::vector<NodePtr>&  nodes,
               int  r, int  c, int  block_size, int
  num_blocks) { NodePtr np = nodes[r*num_blocks + c];
  if  (c + 1  < num_blocks && r != c)
    tbb::flow::make_edge(*np, *nodes[r*num_blocks + c + 1]);
  if  (r + 1  < num_blocks)
    tbb::flow::make_edge(*np, *nodes[(r + 1)*num_blocks + c]);
}
```

図 4-17. addEdges の実装。サンプルコード: graph/graph_fwd_substitution.cpp

図 4-15 に示すように、完全なグラフが構築されると、左上ノードに単一 continue_msg を送信してグラフを開始します。先行ノードを持たない continue_node は、メッセージを受信するたびに実行されます。左上のノードにメッセージを送信すると、依存関係グラフが開始されます。ここでも、グラフの実行完了を待機するには g.wait_for_all() を使用します。

まとめ

この章では、2 章で紹介したアルゴリズムに基づいて、アプリケーションで並列パターンを実装する基本ツールとしてフローグラフを確認しました。フローグラフを使用すると、プログラムを相互接続された計算のグラフとして表現し、イベントベースの協調並列パターンを効果的にモデル化できます。これにより、フィルターまたはステージを介したデータフローの表現が可能になり、明示的なデータ転送を必要とせずに操作間の依存関係を表現することもできます。

フローグラフが本質的に非同期であり、非ブロック化メッセージパッシングを利用して TBB タスクをトリガーし、利用可能なワーカースレッドによって実行される仕組みを確認しました。一般的なアプローチでは、メッセージの受け渡しを開始する前に完全なグラフを構築しますが、実行中にグラフの構造を動的に変更することもできます。動的変更については 5 章で説明します。

並列性を表現するためにグラフを使用する利点を確認し、グラフが計算間の関係を自然で明確に表す方法を提供することを確認しました。この明確性により、TBB はタスク、パイプライン、データ並列処理などのさまざまなタイプの並列処理を活用して、タスクを効率良く並列にスケジュールできます。

グラフ・オブジェクトの作成、さまざまな種類のノード（機能、ジョイン、バッファリング、制御フロー）の構築、ノード間のメッセージフローの管理など、フローグラフの使用に必要な手順の例を示しました。この章では、これらの概念を例と図で説明することで、フローグラフの基礎を理解し、並行プログラミングにおけるより高度なアプリケーションを構築できるようにしました。

5 章では、使用するリソース量の制限、タスクのオーバーヘッド管理、ノードの優先順位の使用、グラフの構造の動的な変更、アクセラレーターとの相互運用など、フローグラフの使用に関する高度な側面を詳しく説明します。

5 章 フローグラフ: アプリケーションの表現

前章で学んだように、フローグラフ API は非常に汎用性が高く、ループを組み込んだグラフを構築したり、`multifunction_node` などのノードを利用して各受信メッセージに対して複数のメッセージを生成したり、さまざまな方法で並列処理を導入することができます。このレベルの柔軟性により、非常に効率良い実装から効率の悪い実装まで、同じアプリケーションをさまざまな形式で表現できるようになります。それぞれのアプローチは正しい結果を生成できますが、複雑性とオーバーヘッドの点で大きく異なる場合があります。この章では、実行効率を最適化するフローグラフ・アプリケーションの開発に役立つ戦略について説明します。

最適なメッセージタイプを選択

メッセージは、グラフ内のさまざまなポイント、例えば入力ポートや出力ポートにバッファリングされることがあります (表 4-1 を参照)。メッセージはバッファリングされると、バッファーにコピーされます (バッファー以外にも受信者がいる可能性があるため、移動されません)。メッセージのコピーにかかる時間とメモリーのコストは、メッセージの種類によって異なります。可能であれば、コピー関連のコストが低くなるように、小さなオブジェクトまたはポインターを渡します。

したがって、メッセージの種類は慎重に選択しなければなりません。大きなオブジェクトを渡す場合は、値ではなくポインター経由で渡す (または `shared_ptr` などのスマートポインターを使用してより安全に渡す) 方が適切です。ただし、ポインターを渡すときは、グラフ内の並列処理による潜在的な同時アクセスに注意し、必要に応じてそれらのアクセスを保護する必要があります。大きなオブジェクトを読み取り専用部分と読み取り/書き込み部分に分割し、読み取り専用部分にはポインター経由でアクセスし、読み取り/書き込み部分には値で渡すこともできます。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_5

読み取り専用データと読み取り/書き込みデータを分割すると、同時アクセスごとに読み取り/書き込み部分の保護されたコピーを保持し、読み取り専用部分を共有してメモリー容量を最小限に抑えることができます。

ストリーミング・メッセージのリソース使用量の制限

メッセージをフローグラフにストリーミングする場合、無制限の数のタスクが生成されたり、グラフ内に無制限の数のメッセージがバッファリングされないように、リソース使用を制限する必要がある場合があります。表 5-1 は、この節で詳しく説明するアプローチの違いを示しています。図 5-1 では、機能ノードの同時実行制限を使用して、`input_node` によるメッセージの生成を制限する方法を説明します。図 5-2 では、`limiter_node` がサブグラフに到達するメッセージの数を制御する方法を示しています。最後に、図 5-4 では、トークンが並列処理を制限しながら、メッセージ割り当ての総数を制限するキャッシュのように機能するのを示しています。

表 5-1. この節で説明するタスクとメモリーの増加を制御するアプローチ

アプローチ	説明
同時実行制限 (図 5-1)	ノードごとの同時実行制限により、入力ノードの新しいメッセージの生成が制限されます。
リミッターノード (図 5-2)	`limiter_node` は、グラフ内のポイントを通過するメッセージの数を制限します。サブグラフの最後のノードは、新しいメッセージが入る可能性があるときに、`limiter_node` にメッセージを送信します。
トークンパッシング (図 5-4)	予約 `join_node` は、グラフ内のポイントを通過するメッセージの数を制限します。`buffer_node` に格納される事前に割り当てられたトークンはメッセージ生成を制限し、キャッシュとしても機能します。

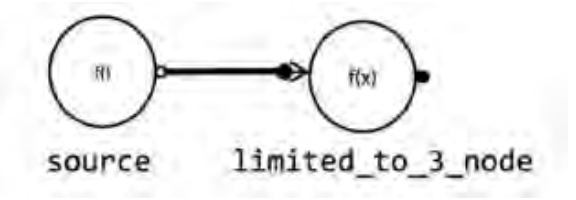
ノードごとの同時実行制限によるタスクとメモリー増加の制御

function_node、multifunction_node、および　async_node コンストラクターは、それぞれのノードのメッセージを処理するため同時に実行できるタスクの数を制限する concurrency_limit　引数を受け取ります。この値を 1 または flow::serial に設定して、スレッドセーフでないコードを含むノードを保護したり、flow::unlimited に設定して、各メッセージが到着するたびにタスクが生成されるようにし、ノードによって生成される潜在的な並列処理を最大化できます。

concurrency_limit を 1 から flow::unlimited までの値（定数 4 など）に設定すると、タスク数が制限され、ある程度の並列処理は許可されますが、無制限の並列処理は許可されません。これにより、タスクとメモリーの増加を制限できます。並列処理を制限する理由がない場合、最大限の並列処理が可能になり、ブックキーピングのオーバーヘッドが最小限に抑えられるため、flow::unlimited の使用を推奨します。

ただし、concurrency_limit だけでは十分ではありません。デフォルトでは、function_node、multifunction_node、または　async_node の入力ポートには無制限のキューがあります。このデフォルトモードでは、同時実行制限を超えると、タスクをすぐに生成することができず、代わりに受信メッセージがこのキューに格納され、タスクが最終的に終了すると、このキューから FIFO 順に別のメッセージがキャプチャーされ、新しいタスクが生成されます。つまり、ノードがメッセージを処理する速度よりも速くメッセージが到着すると、このキューは無制限に大きくなる可能性があります。ノードが追いつくまでメッセージのバースト的な到着に対処するためにのみバッファリングが必要な場合は、問題はありません。しかし、到着率が継続的に処理率を上回ると、メモリー不足になる可能性があります。

図 5-1 では、拒否ポリシーと同時実行制限 3 を持つ function_node limited_to_3_node を構築しています。拒否ポリシーを使用すると、ノード内の暗黙的なバッファリングがオフになります。このグラフでは一度に最大 4 つのメッセージが表示されます。function_node limited_to_3_node によって並列処理されるメッセージは最大 3 つあり、input_node ソースによって生成またはバッファリングされるメッセージは 1 つになります。




```
tbb::flow::graph g;

int  id = 0; tbb::flow::input_node
input{g, [&id](tbb::flow_control& fc) -> MsgPtr  {
   int  next_id = id++;
   if  (!is_msg(next_id)) { fc.stop();
      return  nullptr;
   } else  {
      return  make_msg(id);
   }
  }};
tbb::flow::function_node<MsgPtr, int, tbb::flow::rejecting>
limited_to_3_node{g, 3, [](const  MsgPtr& m) {
    doWork("L3", m);
    return  0;
  }
};
tbb::flow::make_edge(input, limited_to_3_node);

input.activate();
g.wait_for_all();
```

図 5-1. 同時実行性が制限されたノードに接続された input_node を使用して、タスクの作成とメモリーの使用を制限します。サンプルコード: graph/graph_limiting_messages.cpp

このグラフが合計 12 個のメッセージを生成する場合、一度に最大 4 個のメッセージと 3 個のワークタスクがアクティブになります。ただし、ノードの同時実行の制限は単一のノードにのみ影響し、先行ノードにバックプレッシャーをかける目的で使用できますが、グラフ全体の同時実行を制限したり、サブグラフ内の同時実行を制限するには十分ではありません。

limiter_node によるタスクとメモリーの増加を制御

グラフ全体の同時実行を制限したり、サブグラフ内の同時実行を制限するには、`limiter_node [flow_graph.limiter_node]` を使用することもできます。`limiter_node` は、通過するメッセージ数をカウントして制限します。しきい値引数を受け取るコンストラクターと、カウントを減分するために使用される埋め込みレシーバーへの参照を返すデクリメンター関数があります。図 5-3 では、図 5-2 に示されているグラフを実装します。しきい値 3 の `limiter_node` を構築し、サブグラフの最後のノードの出力をそのデクリメンターに接続します。しきい値に達すると、最後のノードからメッセージを受信したときにのみ、新しいメッセージがリミッターを通過することが許可されます。

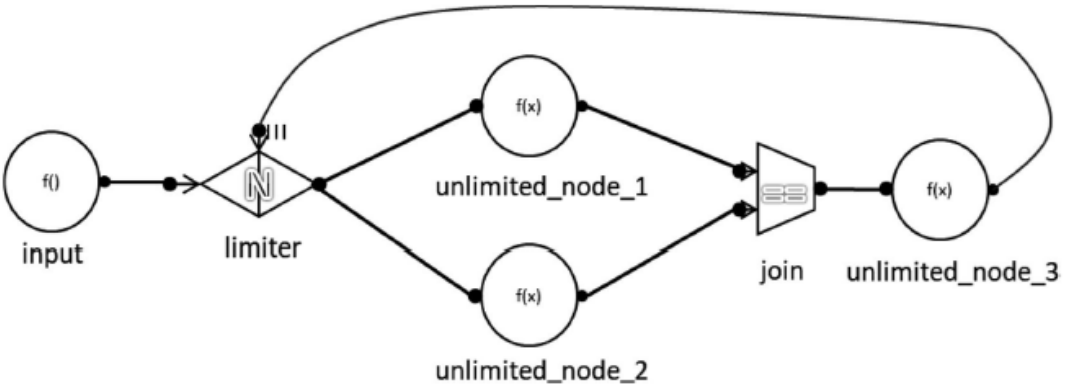


図 5-2. `limiter_node` を使用したグラフの図。`input_node` はリミッターの制限によって制約されますが、通過するメッセージは後続の両方のノードに送信されます。

前の例と同様に、送信中のメッセージ数は制限されていますが、入力によって 12 個のメッセージが生成されると、最大で 4 個のメッセージと *6 つのタスク*が同時に実行されます。このアプローチと図 5-1 のアプローチの主な違いは、`limiter_node` の後にサブグラフ全体を配置できることです。その結果、主に次のような 2 つの帰結が得られます。まず、グラフ内のこの単一のポインターを介して、サブグラフ全体で使用されるリソースを制限できます。しかし、2 番目に、`limiter_node` を通過するメッセージとグラフによって生成されるタスク数の間には、単純な 1 対 1 の対応はありません。例えば、リミッターによって 3 つのメッセージのみが通過を許可されている場合でも、最大 6 つのタスクを同時に実行できます。3 つのメッセージのそれぞれは、`unlimited_node_1` から 1 つ、`unlimited_node_2` から 1 つ、合計 2 つの同時実行タスクを作成できます。

```
tbb::flow::graph g;

int  id = 0;
tbb::flow::input_node
input{g, [&id](tbb::flow_control& fc) ->
    MsgPtr  { int  next_id = id++;
    if  (!is_msg(next_id)) {
      fc.stop();
      return  nullptr;
     } else  {
      return  make_msg(id);
    }
  }};
tbb::flow::limiter_node<MsgPtr> limiter{g, 3};
tbb::flow::function_node
   unlimited_node_1{g,tbb::flow::unlimited,
       [] (const  MsgPtr& m) {doWork("U1", m);return  m;}};
tbb::flow::function_node
   unlimited_node_2{g,tbb::flow::unlimited,
       [] (const  MsgPtr& m) {doWork("U2", m);return  m;}};
tbb::flow::join_node<std::tuple<MsgPtr, MsgPtr>,
   tbb::flow::key_matching<int>>
   join{ g, [](const  MsgPtr& p) { return  p->get_id(); },
            [](const  MsgPtr& p) { return  p->get_id(); }};
tbb::flow::function_node
unlimited_node_3{g, tbb::flow::unlimited,
[] (const  std::tuple<MsgPtr, MsgPtr>& m) {
    doWork("U3", std::get<0>(m));return  tbb::flow::continue_msg{};}};

tbb::flow::make_edge(input, limiter);
tbb::flow::make_edge(limiter, unlimited_node_1);
tbb::flow::make_edge(limiter, unlimited_node_2);
tbb::flow::make_edge(unlimited_node_1, tbb::flow::input_port<0>(join));
tbb::flow::make_edge(unlimited_node_2, tbb::flow::input_port<1>(join));
tbb::flow::make_edge(join, unlimited_node_3);
tbb::flow::make_edge(unlimited_node_3, limiter.decrementer());

input.activate();
g.wait_for_all();
```

図 5-3. `limiter_node` を使用して、タスクの作成とメモリーの使用を制限します。ボックス内のコードは、`limiter_node` とそれに接続されたエッジの作成を示しています。サンプルコード: `graph/graph_limiting_messages.cpp`

トークンによる繰り返しの制御

グラフ全体またはサブグラフの増加を管理するもう 1 つの方法は、図 5-4 に示すようにトークン・パッシング・スキームを使用することです。ここでは、トークンはノードごとの同時実行制限または `limiter_node` しきい値の代わりになります。前章の表 4-3 で説明したように、予約ポリシーで構築された `join_node` は、受信メッセージをバッファリングせず、各受信ポートでメッセージを予約できる場合にのみメッセージを消費します。図 5-4 では、この性質により、`token_buffer` のトークンとペアにできる場合にのみソースからのメッセージを消費することで、入力から生成されるメッセージの数を制限します。

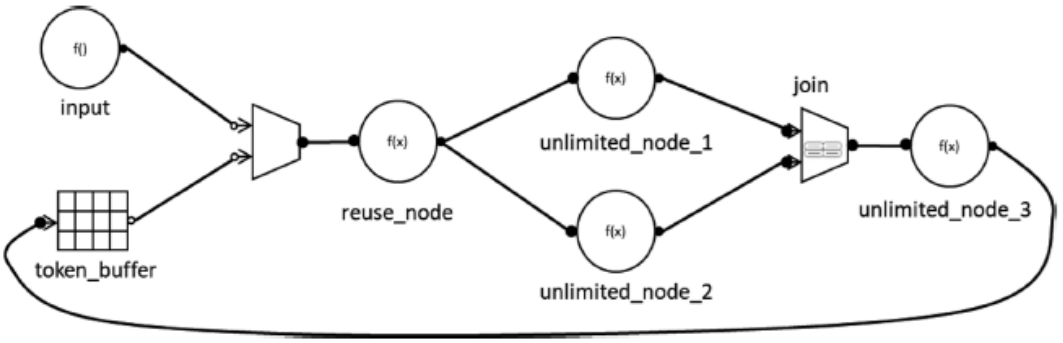


図 5-4.　予約 `join_node` とトークンを使用して、タスクの作成とメモリーの使用を制限します。サンプルコード: `graph/graph_limiting_messages.cpp`

トークンは、プリミティブ型、空のオブジェクト、事前に割り当てられたバッファーなど、任意の型にできます。したがって、`token_buffer` を未使用アイテムのキャッシュのように使用できます。ここでは、割り当てコストが高い大きなオブジェクトが格納され、受信メッセージと一致すると再利用されます。図 5-5 では、`token_buffer` に事前に割り当てられ、`for` ループで明示的にグラフに配置された 3 つのメッセージが格納されています。`limiter_node` の例と同様に、このグラフは最大 6 つのタスクを同時に実行できますが、これらのオブジェクトは再利用され、ポインターによってアクセスされるため、アプリケーションのライフタイム全体にわたって割り当てられるメッセージは 3 つだけです。

```
using  token_t  = MsgPtr;
tbb::flow::graph g;
int  id = 0;
tbb::flow::input_node
input{g, [&id](tbb::flow_control& fc) {
  int  next_id = id++;
    if  (!is_msg(next_id)) {fc.stop();return  -1;} else  {return  next_id;}
  }};
tbb::flow::buffer_node<MsgPtr> token_buffer{g};
tbb::flow::join_node<std::tuple<int, token_t>,
                     tbb::flow::reserving> token_join{g};
tbb::flow::function_node<std::tuple<int,  token_t>,
  MsgPtr, tbb::flow::lightweight>
    reuse_node{g, tbb::flow::unlimited,
    [] (const    std::tuple<int, token_t>& m) -> MsgPtr  {
      return  recycle_token_as_msg(std::get<1>(m), std::get<0>(m));}};
tbb::flow::function_node
    unlimited_node_1{g, tbb::flow::unlimited,
    [] (const  MsgPtr& m) {doWork("U1", m);return  m;}};
tbb::flow::function_node
    unlimited_node_2{g,tbb::flow::unlimited,
    [] (const  MsgPtr& m) {doWork("U2", m);return  m;}};
tbb::flow::join_node<std::tuple<MsgPtr, MsgPtr>, tbb::flow::tag_matching>
    join{ g, [](const  MsgPtr& p) { return  p->get_id(); },
             [](const  MsgPtr& p) { return  p->get_id(); }};
tbb::flow::function_node
    unlimited_node_3{g, tbb::flow::unlimited,
    [] (const  std::tuple<MsgPtr, MsgPtr>& t) {
      auto  m = std::get<0>(t);doWork("U3", m);return  m;}};
tbb::flow::make_edge(input, tbb::flow::input_port<0>(token_join));
tbb::flow::make_edge(token_buffer, tbb::flow::input_port<1>(token_join));
tbb::flow::make_edge(token_join, reuse_node);
tbb::flow::make_edge(reuse_node, unlimited_node_1);
tbb::flow::make_edge(reuse_node, unlimited_node_2);
tbb::flow::make_edge(unlimited_node_1, tbb::flow::input_port<0>(join));
tbb::flow::make_edge(unlimited_node_2, tbb::flow::input_port<1>(join));
tbb::flow::make_edge(join, unlimited_node_3);
tbb::flow::make_edge(unlimited_node_3, token_buffer);

for  (int  i = 0; i < 3; ++i)
    token_buffer.try_put(make_msg(-1));
input.activate();
g.wait_for_all();
```

図 5-5. トークンを使用して、タスクの作成とメモリーの使用を制限します。この実装では、未使用の大きなメッセージのキャッシュとして `token_buffer` を使用します。枠で囲まれたコードは、トークンバッファー、予約 `join_node`、およびそれらに接続されたエッジの作成を示しています。サンプルコード: `graph/graph_limiting_messages.cpp`

タスク・オーバーヘッドの管理

通常、機能ノードは、TBB タスクを生成して、ユーザー提供のボディーを実行します。タスクの作成とスケジュール設定はコストが掛かるため、タスクのワーク量について慎重に検討しなければなりません。ノードのワーク量が少なすぎると、それをタスクとしてパッケージ化して実行するコストが、他のタスクとの同時実行によって得られるメリットを上回る可能性があります。スケジュールのコストはシステムによって異なるため、タスク・スケジュールのコストを償却するのに十分なワーク量について厳格なルールはありません。しかし、一般的な目安として、タスクは 1 マイクロ秒より大きくする必要があります。

アプリケーションの自然な表現によってノードが小さすぎる場合はどうなるでしょうか？ この問題に対処する方法は 2 つあります:

（1）人間が読むとコードが少し理解しにくくなるとしても、手動でノードを結合するか、

（2）`tbb::flow::lightweight` ポリシーを使用して、メッセージが到着したときにボディーを別々にスケジュールされたタスクとしてパッケージ化するのではなく、呼び出し元のスレッドによって直ちに実行するように TBB ライブラリーに指示します。

`tbb::flow::lightweight` ポリシーは TBB スケジューラーへのヒントとして機能するため、TBB は引き続きタスクを作成するのを選択する場合があります。さらに、軽量ポリシーを有効にするには、ノードボディーの関数呼び出し `operator()` は `noexcept` である必要があります。

図 5-6 は、関数 `small_nodes` で、加算ノード、乗算ノード、立方ノードを持つ 3 ノードグラフを実装する例を示しています。各ノードはほとんどワークを実行しません。関数 `small_nodes_combined` は、ノードを 1 つのノードに結合する例を示します。コードの意図は分かりにくくなりますが、オーバーヘッドを大幅に削減できます。関数 `small_nodes_lightweight` では、乗算ノードと立方ノードが軽量ポリシーで構築されます。このグラフでは、追加ノードへの各 `try_put` が新しいタスクを開始するため、ある程度の並列処理が可能です。ただし、乗算ノードと立方ノードは、先行する追加で作成されたタスク内から直接実行されるため、追加タスクは作成およびスケジュールされません。

どのアプローチが最適であるかは、ユースケースによって異なります。ノードを結合する場合、コンパイラーは結合されたコード全体を把握できるようになり、適切に最適化できるため、さらに利点が得られます。軽量ポリシーを使用すると、コードの可読性はそのままで、オーバーヘッドが削減されます。

軽量ポリシーは同時実行制限やその他のポリシーと組み合わせ可能であることに注意してください。例えば、次に示すように、同時実行制限が 3 で、その制限に達するとメッセージを拒否するノードを作成できます:

```
function_node< int, int, rejecting_lightweight >
my_node( g, 3, [](const  int  &v) noexcept { f(v); } );
```

また、軽量（lightweight）は単なるヒントであることにも注意してください。TBB は、ボディーを実行するタスクのスケジュールを選択できます。

小さなノードは定義上小さいため、頻繁に実行されない限り、その影響は大きなノードによって実現される並列処理によって隠匿されてしまうことがあります。したがって、図 5-6 に示す手法は便利ですが、小さなノードを繰り返し実行することによってアプリケーション時間に大きな影響が出ない限り、通常は必要ありません。

```
void small_nodes() {
  tbb::flow::graph g;
  tbb::flow::function_node add( g, tbb::flow::unlimited,
                      [](const int &v) { return v+1; } );
  tbb::flow::function_node multiply( g, tbb::flow::unlimited,
                          [](const int &v) { return v*2; } );
  tbb::flow::function_node cube( g, tbb::flow::unlimited,
                      [](const int &v) { return v*v*v; } );
  tbb::flow::make_edge(add, multiply);
  tbb::flow::make_edge(multiply, cube);
  for(int i = 1; i <= N; ++i)
     add.try_put(i);
  g.wait_for_all();
}

void small_nodes_combined() {
  tbb::flow::graph g; tbb::flow::function_node< int, int >
    combined_node( g, tbb::flow::unlimited,
      [](const int &v) {auto v2 = (v+1)*2;return v2*v2*v2;});
  for(int i = 1; i <= N; ++i)
    combined_node.try_put(i); g.wait_for_all();
}

void small_nodes_lightweight() {
  tbb::flow::graph g;
  tbb::flow::function_node< int, int >
    add( g, tbb::flow::unlimited,[](const int &v) { return v+1; } );
  tbb::flow::function_node< int, int, tbb::flow::lightweight >
    multiply( g, tbb::flow::unlimited,
              [](const int &v) noexcept { return v*2; } );
  tbb::flow::function_node< int, int, tbb::flow::lightweight >
    cube( g, tbb::flow::unlimited,
             [](const int &v) noexcept { return v*v*v; } );
  tbb::flow::make_edge(add, multiply);
  tbb::flow::make_edge(multiply, cube);
  for(int i = 1; i <= N; ++i)
     add.try_put(i);
  g.wait_for_all();
}
```

図 5-6. ノードを結合するか、軽量ポリシーを使用して、タスク・スケジュールのオーバーヘッドを削減します。サンプルコード: `graph/graph_small_nodes.cpp`

多数のメッセージの代わりにネストされた並列処理を使用

2 章で説明したように、TBB は一般的な並列パターンに最適化されたアルゴリズムをいくつか提供します。ループを含むグラフを表現し、multifunction_node などのノードを使用して各呼び出しから多くのメッセージを出力できますが、ネストされた並列処理の有用性に注意する必要があります。簡単な例を図 5-7 に示します。

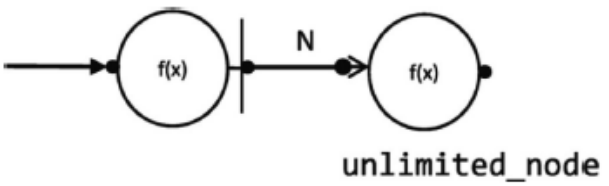


図 5-7. 受信したメッセージごとに多数のメッセージを送信する multifunction_node。このパターンは、ネストされた parallel_for ループとして表現した方が良いかもしれません。

図 5-7 では、multifunction_node が受信するメッセージごとに、無制限の同時実行性で後続の function_node に流れる多数の出力メッセージを生成しています。このグラフは並列ループのように動作し、multifunction_node が制御ループとして機能し、function_node がボディーとして機能します。しかし、これは並列ループを実装するには非効率な方法です。このパターンにも有効な用途があるかもしれませんが、高度に最適化された並列ループ・アルゴリズムを使用する方が効率良い可能性があります。このグラフ全体は、ネストされた parallel_for を含む単一ノードに縮小される可能性があります。もちろん、この置き換えが可能であるか望ましいかは、アプリケーションによって異なります。

ノードの優先度はスケジューリングを改善

通常、TBB スケジューラーによる選択を制御すべきではありません。ただし、フローグラフ内の一部のノードが明らかに他のノードよりも優先度が高い場合を除きます。最も一般的なケースは、ノードがアクセラレーターや GPU などの非同期エージェントと対話する場合です。アクセラレーターにワークを送信するノードの実行が遅れると、アクセラレーターを十分に活用できなくなる可能性があります。

図 5-8 の例は、ノードの 1 つが他のノードよりも重要であるフローグラフを示しています。そのノードは、「私が一番だ!」と叫んでいます。しかし、このグラフを 2 つのスレッドのみで実行すると、TBB ライブラリーはどのノードを実行するかを選択し、3 つのノード間で公平になるように努める必要があります。

```
tbb::flow::graph g;

tbb::flow::broadcast_node<int> b{g};

tbb::flow::function_node
  n1{g, tbb::flow::unlimited,
  [](const  int& t) { std::printf("Hi from n1\n"); return  0; }};

tbb::flow::function_node
  n2{g, tbb::flow::unlimited,
  [](const  int& t) { std::printf("Hi from n2\n"); return  0; }};

tbb::flow::function_node
  n3{g, tbb::flow::unlimited,
   [](const  int& t) { std::printf("n3: Me first!\n"); return  0; }};

tbb::flow::make_edge(b, n1);
tbb::flow::make_edge(b, n2);
tbb::flow::make_edge(b, n3);

for  (int  i = 0; i < 5; ++i) b.try_put(10);
 g.wait_for_all();
```

図 5-8. 他のノードよりも時間のかかるノードがあるフローグラフ。サンプルコード: graph/graph_node_priorities.cpp

図 5-8 を 2 つのスレッドのみで実行した例では、出力は次の順序で表示されました:

```
Hi from n1
Hi from n2
n3: Me first!
Hi from n1
Hi from n2
n3: Me first!
Hi from n1
Hi from n2
n3: Me first!
n3: Me first!
Hi from n1
Hi from n2
Hi from n2
Hi from n1
n3: Me first!
```

しかし実際に、ノード n3 を他の 2 つのノードより優先させたい場合は、図 5-9 に示すように、ノード優先度 [flow_graph.node_priorities] を書き直して追加できます。

```
tbb::flow::function_node
  n3{g, tbb::flow::unlimited,
    [](const  int& t) { std::printf("n3: Me first!\n"); return  0; },
    tbb::flow::node_priority_t(1)};
```

図 5-9. ノード優先度を使用して、実行を n3 に偏らせます。サンプルコード: graph/graph_node_priorities.cpp

TBB ノードの優先度は符号なし整数値で、値が大きいほど優先度が高くなります。明示的に優先度が指定されていないノードのデフォルト優先度は 0 になります。図 5-9 に示すように n3 を変更し、再度 2 つのスレッドのみを使用して実行すると、TBB ライブラリーは n3 を優先します。

```
n3: Me first!
n3: Me first!
n3: Me first!
n3: Me first!
n3: Me first!
Hi from n2
Hi from n1
Hi from n1
Hi from n2
Hi from n1
Hi from n2
Hi from n1
Hi from n2
Hi from n1
Hi from n2
```

グラフ構築と実行を重複する

TBB フローグラフ API は、即時的な実行モデルを使用します。機能ノードがメッセージを受信するたびに、同時実行の制限とポリシーが許す限り、受信メッセージのボディーを実行するタスクを生成します。この即時的な設計のため、input_node オブジェクトをアクティブにする前、または明示的なメッセージを送信する前に、グラフ全体を完全に定義するのが一般的な方法です。ただし、それは必須ではありません。グラフの実行をその構築と重複することはできますが、そうする場合、部分的に定義されたグラフ内でメッセージがどのように流れるか注意しなければなりません。

これを説明するため、図 5-10 に示すような依存関係グラフを構築しますが、グラフの構築と実行を重複させます。このグラフでは、グラフの構築時にメッセージが失われないように、write_once_ ノード・オブジェクト [flow_graph.write_once_node] を使用してメッセージをバッファリングします。write_once_node は単一の値をバッファリングしてブロードキャスト値が書き込まれた後に追加された後続も含め、すべての後続にそれを渡します。図 5-10 では、フューチャーのように write_once_node オブジェクトを使用しています。

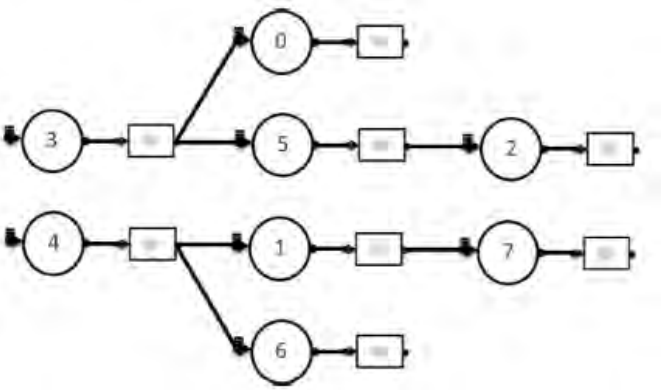


図 5-10. write_once_node オブジェクトを使用してフューチャーのように使用することで、一部のノードの実行中に安全に構築できる依存関係グラフ

図 5-11 は、この依存関係グラフを構築して実行するコードを示しています。メイン関数の for ループは構成ベクトルを反復処理します。このベクトルには、図 5-10 に示すノード依存関係が保存されます。各要素が処理されるたびに、ID に対応する continue_node が作成されます。コードの枠で囲まれた部分に示すように、新しいノードに先行ノードが存在する場合、その先行ノードの「フューチャー」write_once_node から新しいノードの入力へのエッジが作成されます。ノードに先行ノードがない場合、ノードを開始するのに try_put が呼び出されます。

このサンプルを実行すると、ノードが ID 0 から ID 7 の順に作成され、先行ノードのないノードが検出されるとすぐに開始される場合でも、図 5-10 に示すノードの部分的な順序が守られていることが分かります。実行すると、ノードは 3、0、4、1、5、2、6、7 の順序で実行されることがわかりました。

この節では依存関係グラフを示しますが、同じ考え方がデータ・フローグラフにも当てはまります。グラフは動的に構築できますが、グラフ内の（最終的な）受信ノードが構築され、エッジが確立される前に、グラフ内のあるポイントに到着する可能性のあるメッセージをバッファリングすることに注意する必要があります。

```
struct config_t { int id; int predecessor; };
std::vector<config_t> configuration =
  {{0,3}, {1,4}, {2,5}, {3,-1}, {4,-1}, {5,3}, {6,4}, {7,1}};
 int num_nodes = configuration.size();

int main() {
  tbb::flow::graph g;

  // ワークノードへのポインターのベクトルを作成
  using work_node_t = tbb::flow::continue_node<tbb::flow::continue_msg>;
  std::vector<std::unique_ptr<work_node_t>> work_nodes(num_nodes);

  // "promises" の完全なベクトルを作成
  using future_node_t = tbb::flow::write_once_node<tbb::flow::continue_msg>;
  std::vector<future_node_t> future_nodes(num_nodes, future_node_t{g});

  // グラフをビルド (先行ノードのない開始ノード)
  for(int i = 0; i < num_nodes; ++i) {
    const config_t& c = configuration[i];

    // 要素のワークノードを作成
    work_nodes[c.id].reset(
        new work_node_t{g,
                        [c](const tbb::flow::continue_msg& m) {
                          std::printf("Executing %d\n", c.id);
                          return m;
                        }});
    // 新しいノードをフューチャーに接続
    tbb::flow::make_edge(*work_nodes[c.id], future_nodes[c.id]);

    // ノードを開始するか、先行ノードの promise にリンク
    if (c.predecessor != -1) {
       std::printf("new %d  with %d  -> %d\n", c.id, c.predecessor, c.id);
       tbb::flow::make_edge(future_nodes[c.predecessor], *work_nodes[c.id]);
    } else {
       std::printf("starting %d  from main\n", c.id);
       work_nodes[c.id]->try_put(tbb::flow::continue_msg{});
    }
    std::printf("**** wait a bit in main\n");
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    std::printf("**** done waiting in main\n");
  }
  g.wait_for_all();
  return 0;
}
```

図 5-11. グラフ構築と実行の重複。サンプルコード: graph/ graph_execute_with_building.cpp

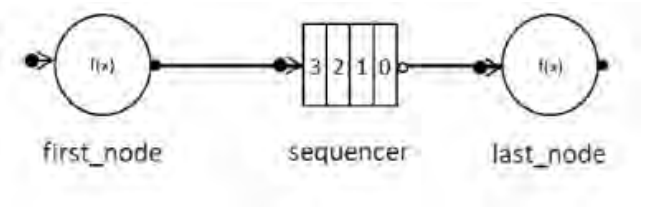
必要に応じて並列処理後に順序を再確立

フローグラフは単純な tbb::parallel_pipeline ほど構造化されていないため、グラフ内のポイントでメッセージ順序の確立が必要となる場合があります。データ・フローグラフで順序を確立する一般的な方法は 3 つあります。キーマッチング join_node を使用する、sequencer_node を使用する、または multifunction_node を使用する方法です。

例えば、4 章では、立体 3D フローグラフの並列処理により、左右の画像が mergeImageBuffersNode に順序どおりに到着していませんでした。この例では、タグマッチング join_node を使用して、正しい 2 つの画像が mergeImageBuffersNode への入力としてペアになっていることを確認しました。さまざまなジョインノードのポリシーについては、表 4-3 で説明します。

順序を確立する別の方法は、sequencer_node を使用することです。sequencer_node は、ユーザーが提供するボディー・オブジェクトを使用して受信メッセージからシーケンス番号を取得し、メッセージをシーケンス順に出力するバッファーです。

図 5-12 には、first_node、sequencer、last_node の 3 つのノードのグラフがあります。最終シリアル出力ノード last_node の前に、sequencer_node を使用してメッセージの入力順序を再確立します。function_node first_node は無制限であるため、タスクは順序どおりに終了せず、完了すると出力を送信できます。sequencer_node は、各メッセージが最初に作成されたときに割り当てられたシーケンス番号を使用して、入力順序を再確立します。




```
void  orderWithSequencer() {
  const  int  N = 10;
  tbb::flow::graph g;
  tbb::flow::function_node
  first_node{g, tbb::flow::unlimited,
  [] (const  MessagePtr& m) {
    m->my_string += "  with sequencer";
    return  m;
  }};
  tbb::flow::sequencer_node
  sequencer(g, [] (const  MessagePtr& m) {
    return  m->my_seq_no;
  });
  tbb::flow::function_node<MessagePtr, int, tbb::flow::rejecting>
  last_node{g, tbb::flow::serial, [] (MessagePtr  m) {
      std::cout << m->my_string << std::endl;
      return  0;
  }};
  tbb::flow::make_edge(first_node, sequencer);
  tbb::flow::make_edge(sequencer, last_node);

  for  (int  i = 0; i < N; ++i)
    first_node.try_put(std::make_shared<Message>(i));
  g.wait_for_all();
}
```

図 5-12. sequencer_node は、メッセージが my_seq_no メンバー変数によって指定された順序でプリントするのに使用されます。サンプルコード: graph/graph_reestablish_order.cpp

シーケンサー・ノードを使用せず、N=10 で同様の例を実行すると、メッセージが last_node に向かう途中で互いに通過するため、出力が前後します:

```
9 no sequencer
8 no sequencer
7 no sequencer
0 no sequencer
1 no sequencer
2 no sequencer
6 no sequencer
5 no sequencer
4 no sequencer
3 no sequencer
```

図 5-12 に示すコードを実行すると、次の出力が表示されます:

```
0 with sequencer
1 with sequencer
2 with sequencer
3 with sequencer
4 with sequencer
5 with sequencer
6 with sequencer
7 with sequencer
8 with sequencer
9 with sequencer
```

ご覧のとおり、sequencer_node はメッセージの順序を再確立できますが、シーケンス番号を割り当て、受信メッセージからその番号を取得できる sequencer_node にボディーを提供する必要があります。

順序を確立する最後のアプローチは、シリアル multifunction_node を使用する方法です。multifunction_node は、特定の入力メッセージに対して、任意の出力ポートに 0 個以上のメッセージを出力できます。受信メッセージごとにメッセージを出力する必要がないため、受信メッセージをバッファリングして、ユーザー定義の順序の制約が満たされるまで保持できます。

例えば、図 5-13 は、シーケンサー順のメッセージが到着するまで受信メッセージをバッファリングすることにより、multifunction_node を使用して sequencer_node を実装する方法を示しています。この例では、最大 N 個のメッセージがノードシーケンサーに送信され、シーケンス番号は 0 から始まり N-1 まで連続すると想定しています。ベクトル v は、空の shared_ptr オブジェクトとして初期化された N 個の要素で作成されます。メッセージがシーケンサーに到着すると、そのメッセージは v の対応する要素に割り当てられます。次に、最後に送信されたシーケンス番号から始めて、有効なメッセージを持つ v の各要素が送信され、シーケンス番号が増加します。一部の受信メッセージには出力メッセージが送信されませんが、その他の受信メッセージについては 1 つ以上のメッセージが送信される場合があります。

```
using MessagePtr = std::shared_ptr<Message>;
using  MFNSequencer  =
  tbb::flow::multifunction_node<MessagePtr, std::tuple<MessagePtr>>;
using  MFNPorts  = typename  MFNSequencer::output_ports_type;

int  seq_i = 0;
std::vector<MessagePtr> v{(const  unsigned)N, MessagePtr{}};

 MFNSequencer sequencer{g, tbb::flow::serial,
[N, &seq_i, &v](MessagePtr  m, MFNPorts& p) {
  v[m->my_seq_no] = m;
  while  (seq_i < N && v[seq_i].use_count()) {
    std::get<0>(p).try_put(v[seq_i++]);
  }
 }};
```

図 5-13. multifunction_node を使用してバッファリングし、順序を再確立します。サンプルコード: graph/graph_reestablish_order.cpp

図 5-13 は、multifunction_node を使用してメッセージをシーケンス順に並べ替える方法を示していますが、一般に、ユーザー定義のメッセージの順序付けやバンドルも使用できます。

同じ入力に関連するメッセージを結合する必要があり、結合されたメッセージを処理する順序は重要でない場合は、キーまたはタグが一致する join_node が適切な選択です。メッセージがグラフに入った（または通過した）順序を再確立する必要がある場合は、sequencer_node を使用します。最後に、さらに複雑なものが必要な場合は、multifunction_node を使用して独自のロジックを作成できます。

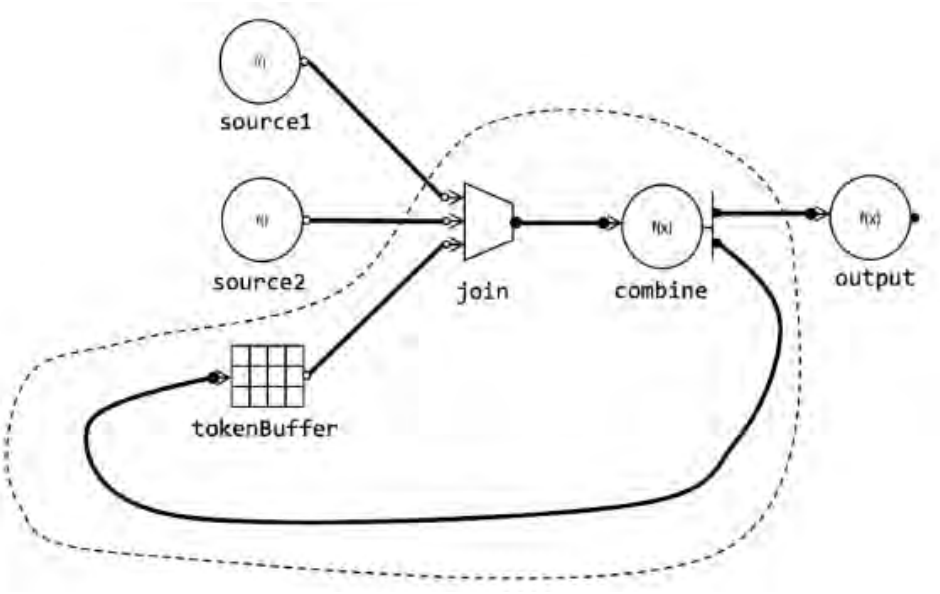
簡単に再利用できるようにノードをグループ化

繰り返し作成する共通のパターンがある場合や、1 つの大きなフラットグラフに詳細が多すぎる場合は、ノードのグループをカプセル化すると便利です。

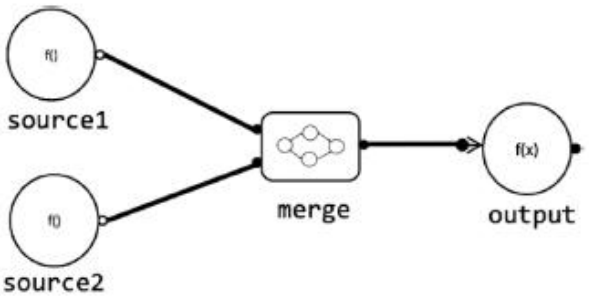
これには、tbb::flow::composite_node を使用できます。compound_node は、他のノードのコレクションをカプセル化して、ファーストクラスのグラフノードのように使用できるようにします。インターフェイスは次のとおりです:

```
template<typename...InputTypes, typename...OutputTypes>
class  composite_node  <std::tuple<InputTypes...>,
std::tuple<OutputTypes...>> :
public  graph_node  {
  public:
    typedef  std::tuple< receiver<InputTypes>&...> input_ports_type;
    typedef  std::tuple< sender<OutputTypes>&...> output_ports_type;

    composite_node  ( graph &g );
    virtual  ~composite_node();

    void  set_external_ports(input_ports_type&& input_ports_tuple,
                             output_ports_type&& output_ports_tuple);
    input_ports_type& input_ports();
    output_ports_type& output_ports();
};
```

この章で説明した他のノードタイプとは異なり、その機能を利用するには、`tbb::flow::composite_node [flow_graph.composite_node]` から継承する新しいクラスを作成する必要があります。例えば、図 5-14(a) のフローグラフを考えてみましょう。このグラフは、source1 と source2 からの 2 つの入力を組み合わせ、トークン・パッシング・スキームを使用してメモリー消費を抑えます。



（a）他の場所で再利用できる可能性のあるパターンを含むグラフ。



（b）グラフは、`composite_node` を使用するように書き換えられました。

図 5-14. 再利用可能なパターンを `composite_node` としてカプセル化

このトークン・パッシング・パターンがアプリケーション、または開発チームのメンバーによって頻繁に使用される場合、図 5-14(b) に示すように、それを独自のノードタイプにカプセル化すると効果的です。また、詳細を隠匿することで、アプリケーションの高レベルのビューを分かりやすく整理します。図 5-15 は、図 5-14(a) の破線部分を実装するノードを単一のマージノードに置き換えたフローグラフ実装を示しています。

```
void  usingMergeNode() {
  tbb::flow::graph g;

  tbb::flow::input_node<BigObjectPtr> source1{g,
  [&] (tbb::flow_control& fc) {
    static  int  in1_count = 0;
    BigObjectPtr p;
    if  (in1_count < A_LARGE_NUMBER)
      p = std::make_shared<BigObject>(in1_count++);
    else
      fc.stop();
    return  p;
  }};

 tbb::flow::input_node<BigObjectPtr> source2{g,
 [&] (tbb::flow_control& fc) {
    static  int  in2_count = 0;
    BigObjectPtr p;
    if  (in2_count < A_LARGE_NUMBER)
      p = std::make_shared<BigObject>(in2_count++);
    else
      fc.stop();
    return  p; }};

  MergeNode  merge{g};

  tbb::flow::function_node<BigObjectPtr> output{g,
    tbb::flow::serial,
    [] (BigObjectPtr  b) {
      std::cout << "Received id == "  << b->getId()
                << "  in final node"  << std::endl;
    }};

  tbb::flow::make_edge(source1, tbb::flow::input_port<0>(merge));
  tbb::flow::make_edge(source2, tbb::flow::input_port<1>(merge));
  tbb::flow::make_edge(merge, output);

  reset_counters();
  source1.activate();
  source2.activate();
  g.wait_for_all();
}
```

図 5-15. tbb::flow::composite_node から継承するクラス MergeNode を使用するフローグラフを作成します。サンプルコード: graph/graph_composite_node.cpp

図 5-15 では、他のフローグラフ・ノードと同様にマージ・ノード・オブジェクトを使用して、その入力ポートと出力ポートにエッジを作成します。図 5-16 は、tbb::flow::composite_node を使用して MergeNode クラスを実装する方法を示しています。図 5-16 では、MergeNode は CompositeType から継承します。これは、以下のエイリアスです。

```
tbb::flow::composite_node<std::tuple<BigObjectPtr, BigObjectPtr>,
                          std::tuple<BigObjectPtr>>;
```

2 つのテンプレート引数は、MergeNode に BigObjectPtr メッセージを受信する 2 つの入力ポートと、BigObjectPtr メッセージを送信する 1 つの出力ポートがあることを示します。

MergeNode クラスには、カプセル化される各ノード（tokenBuffer、join、および combine ノード）のメンバー変数があります。これらのメンバー変数は、MergeNode コンストラクターのメンバー初期化子リストで初期化されます。コンストラクターのボディーでは、tbb::flow::make_edge を呼び出してすべての内部エッジを設定します。

set_external_ports の呼び出しは、メンバーノードのポートを MergeNode の外部ポートに割り当てるために使用されます。この場合、join の最初の 2 つの入力ポートは MergeNode の入力になり、combine の出力は MergeNode の出力になります。最後に、ノードはトークン・パッシング・スキームを実装しているため、tokenBuffer にトークンが入ります。

tbb::flow::composite_node から継承する新しい型を作成するのは大変に思えるかもしれませんが、このインターフェイスを使用すると、特にフローグラフが大きく複雑になるにつれて、読みやすく再利用しやすいコードを作成できます。

```
using BigObjectPtr = std::shared_ptr<BigObject>;
using CompositeType =
  tbb::flow::composite_node<std::tuple<BigObjectPtr, BigObjectPtr>,
                            std::tuple<BigObjectPtr>>;
class MergeNode : public CompositeType {
 using token_t = int;
 tbb::flow::buffer_node<token_t> tokenBuffer;
 tbb::flow::join_node<std::tuple<BigObjectPtr, BigObjectPtr, token_t>,
                      tbb::flow::reserving> join;
 using MFNode =
   tbb::flow::multifunction_node<std::tuple<BigObjectPtr,
     BigObjectPtr, token_t>,std::tuple<BigObjectPtr, token_t>>;
 MFNode combine;

public:
    MergeNode(tbb::flow::graph& g) :
    // ノードを作成
    CompositeType{g},
    tokenBuffer{g},
    join{g},
    combine{g, tbb::flow::unlimited,
      [] (const MFNode::input_type& in, MFNode::output_ports_type& p) {
          BigObjectPtr b0 = std::get<0>(in);
          BigObjectPtr b1 = std::get<1>(in);
          token_t t = std::get<2>(in);
          spinWaitForAtLeast(0.0001);
          b0->mergeIds(b0->getId(), b1->getId());
          std::get<0>(p).try_put(b0);
          std::get<1>(p).try_put(t);
      }}
  {
    // エッジを作成
    tbb::flow::make_edge(tokenBuffer, tbb::flow::input_port<2>(join));
    tbb::flow::make_edge(join, combine);
    tbb::flow::make_edge(tbb::flow::output_port<1>(combine), tokenBuffer);

    // 入力ポートと出力ポートを設定
    CompositeType::set_external_ports(
      CompositeType::input_ports_type(
        tbb::flow::input_port<0>(join),
        tbb::flow::input_port<1>(join)
      ),
      CompositeType::output_ports_type(
        tbb::flow::output_port<0>(combine)
      ));

    // トークンバッファーにデータを入力
    for (token_t i = 0; i < 3; ++i)
      tokenBuffer.try_put(i);
  }};
```

図 5-16. MergeNode を実装します。サンプルコード: graph/graph_composite_node.cpp

ワーカースレッドをブロックする代わりに非同期を使用

フローグラフ API 自体は非同期です。try_put への呼び出しは（軽量ポリシーのノードに対して行われた場合を除いて）、メッセージをグラフに挿入した直後に戻ります。スレッドはブロックせず、メッセージがグラフで処理されるのを待機します。ワークが完了するまでグラフでスレッドをブロックしたい場合、グラフ・オブジェクトで wait_for_all を呼び出す必要があります。

フローグラフとのやり取りは主に非同期です。それでも、さらに非同期性を導入する必要がある場合は async_node [flow_graph.async_node] を使用します。これは図 5-17 の例で説明できます。この例の目的は、input_node、in_node から非同期ノード、offload_node での処理を経てメッセージを送信し、結果を out_node で表示することです。ただし、offload_node のユーザー提供のボディー内でメッセージを直接処理するのではなく、ボディーは SYCL キューを介してワークを SYCL ランタイム（GPU、FPGA、または CPU セット）によって管理されるデバイスにオフロードします。デバイスが完了するとすぐに、別の依存 SYCL タスクを使用して、ゲートウェイ・オブジェクトを介してフローグラフに結果を送り返します。TBB ワーカースレッドはブロックされず、ゲートウェイを介して値が返されるのをアクティブに待機しません。

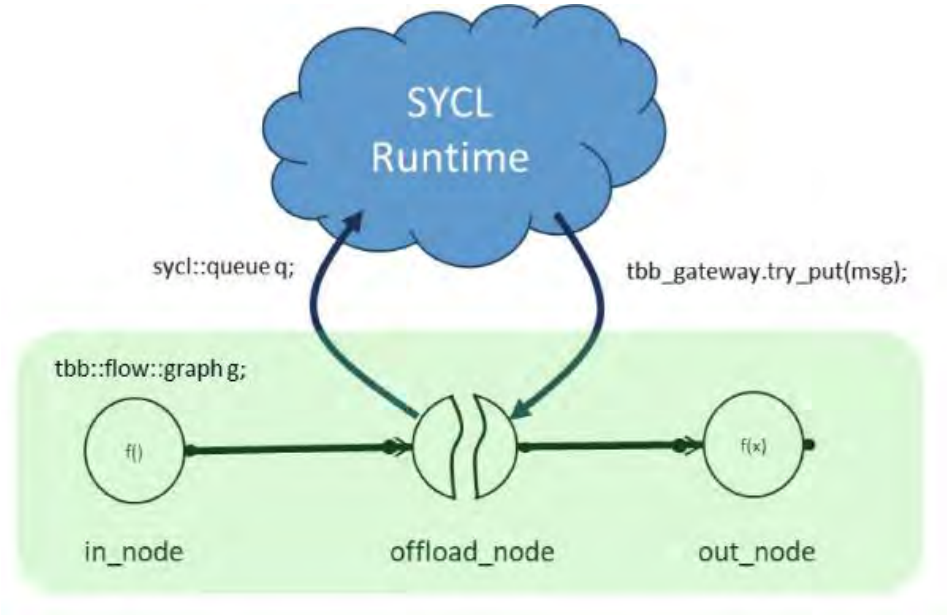


図 5-17. SYCL キューにフィル操作をオフロードする簡単な例

この節では、デモ目的でのみ SYCL を使用します。async_node は、SYCL、CUDA、OpenCL、またはネイティブ CPU スレッドプールにオフロードされるかどうかに関係なく、あらゆる非同期を隠匿するように機能します。

図 5-18 は、図 5-17 を表すグラフの実装を示しています。図 5-18 では、`tbb::flow::graph` と `sycl::queue` の両方が構築されています。フローグラフは、TBB ワーカースレッドに割り当てられたワークを管理します。SYCL キューは、デバイスにオフロードされたワークを管理します。async_node は、TBB フローグラフ・ノードと SYCL キュー間のインターフェイスを管理します。本書は TBB に関する書籍です。SYCL についてさらに詳しく知りたい場合は、「*Data Parallel C++*」というタイトルの書籍が優れたリソースになります。この説明を知っておく必要があるのは、SYCL キューはワークをアクセラレーターにオフロードするのに使用され、ワークが完了した後に実行される依存タスクを SYCL で作成できることです。

```
const  int  N = 1000;
int  total_messages = 3;
```

```
using  msg_t  = std::pair<int, int  *>;
using  offload_node_t  = tbb::flow::async_node<msg_t, msg_t>;
using  gateway_t  = offload_node_t::gateway_type;
```

```
sycl::queue sycl_q;
tbb::flow::graph g;
```

```
tbb::flow::input_node<msg_t>
in_node{g,
    [&](tbb::flow_control& fc)
      { int  *a = nullptr;
      if  (total_messages < 1)
         fc.stop();
      else
         a = sycl::malloc_shared<int>(N, sycl_q);
      return  msg_t{total_messages--, a};
    }};
```

```
offload_node_t  offload_node{ /* 図 5-19 を参照  */  };
```

```
tbb::flow::function_node<msg_t>
out_node{g, tbb::flow::serial,
     [=](const  msg_t& msg) {
       int  id = msg.first;
       int* a = msg.second;
       if  (a == nullptr
           || std::any_of(a, a+N, [id](int  i) { return  i != id; })) {
           std::cout << "ERROR: unexpected msg in out_node\n";
       } else  {
           std::cout << "Received well-formed msg for id "  << id << "\n";
       }
     }};
```

```
tbb::flow::make_edge(in_node, offload_node);
tbb::flow::make_edge(offload_node, out_node);
in_node.activate();
g.wait_for_all();
```

図 5-18. シンプルな SYCL オフロードの例を実装する async_node を含むフローグラフ。サンプルコード: graph/graph_async_sycl.cpp

図 5-19 は、offload_node 構築の詳細を示しています。まず、他の機能ノードタイプと同様に、コンストラクターはグラフ・オブジェクトと同時実行制限の両方を受け取ります。しかし、図 5-19 に示すように、そのボディーは受信メッセージだけでなく、ゲートウェイへの参照 [req.gateway_type] も受け取ります。

```
offload_node_t
offload_node{g, tbb::flow::unlimited,
             [&sycl_q](const  msg_t& msg, gateway_t& tbb_gateway) {

                // グラフに非同期の目的を通知
                tbb_gateway.reserve_wait();

                // 非同期ワークを送信 int
                id = msg.first;
                int* a = msg.second;
                auto  sycl_event = sycl_q.fill(a, id, N);

                // グラフにワークが完了したことを通知
                sycl_q.submit([&](sycl::handler& sycl_h) {
                  sycl_h.depends_on(sycl_event);  // wait for  sycl_event
                  sycl_h.host_task([&tbb_gateway, msg]() {
                    // 結果を送信
                    tbb_gateway.try_put(msg);
                    // グラフにこのエージェントを待たないように指示
                    tbb_gateway.release_wait();
                  });
                });
             }};
```

図 5-19. offload_node のユーザー提供のボディーの詳細。サンプルコード: graph/graph_async_sycl.cpp

async_node ゲートウェイ・タイプ gateway_t は、フローグラフと async_node オブジェクトの両方と対話する方法を表し、次の 3 つのメンバー関数を提供します:

```
bool  gateway_t::try_put(const  Output  &v);
void  gateway_t::reserve_wait();
void  gateway_t::release_wait();
```

図 5-19 では、tbb_gateway.reserve_wait() の呼び出しにより、非同期ワークが開始されていることがグラフに通知されます。reserve_wait 呼び出しの数が release_wait 呼び出しの数を超えると、グラフ内でアクティブに動作している TBB タスクがない場合でも、グラフのワークは完了したとは見なされません。例えば、図 5-18 の最後の行にある g.wait_for_all() の呼び出しは、グラフの完了を待機してブロックし続けます。

図 5-19 で SYCL キューを通じてオフロードされるワークは単純です。これは sycl_q.fill(a, id, N) の呼び出しで、単に a のすべての要素を id に設定するだけです。繰り返しますが、本書では SYCL の動作については詳しく説明しませんので、ここでの説明をそのまま受け入れてください。fill の呼び出しにより、ワークがデバイスに送信されます。

ただし、TBB ワールドに結果を送り返し、非同期ワークが完了したことをフローグラフに通知するには、tbb_gateway を使用する必要があります。これは、図 5-19 で、fill 操作の完了に依存する SYCL host_task を追加することによって実現されます。SYCL host_task は、SYCL ランタイムによってホスト上で実行され、有効な C++ コードを含めることができます。図 5-19 では、host_task のボディーが try_put を呼び出し、その後ゲートウェイで release_wait を呼び出しています。

図 5-19 には多くのコードがありますが、実際の fill 操作の実行と依存するホストタスクのボディーは非同期に実行されることに注意してください。TBB ワーカーは完了を待機してブロックされません。代わりに、async_node ボディーは、SYCL キューを介してタスクの実行を SYCL ランタイムにオフロードし、SYCL ホストタスクによって実行されたときに tbb_gateway.try_put 呼び出しで生成されたタスクに応答して、TBB ワーカースレッドでのみワークが再開されます。また、tbb_gateway は参照渡しされ、この参照は async_node のライフタイム中有効であるため、ノードが破棄された後は使用してはならないことにも注意してください。

この例では SYCL を使用しましたが、前述したように、このコンテキストでは TBB と SYCL の関係に特別なことは何もありません。CUDA や HIP を同様に使用することもできます。

reserve_wait 呼び出しを最適化する可能性

reserve_wait() の呼び出し回数が release_wait() の呼び出し回数より多い間は、graph::wait_for_all() の呼び出しは終了しません。ただし、async_node が受信する入力メッセージごとに reserve_wait() を呼び出す必要はありません。graph::wait_for_all() が早期に戻るのを防ぐため reserve_wait() を呼び出す必要がありますが、必要に応じて reserve_wait() と release_wait() を呼び出す回数を（慎重に）最適化できる可能性があります。

まとめ

フローグラフ API は、依存関係とデータ・フローグラフを構築する柔軟で強力なインターフェイスを提供します。この章では、フローグラフの高レベル実行インターフェイスを利用してアプリケーションのパフォーマンスを向上させるさまざまなアプローチを検討しました。フローグラフ・インターフェイスは TBB タスク上に構築されるため、構成の可能性と最適化の機能を継承します。この章で紹介したヒントが、フローグラフ向けの強力な API セットの可能性を最大限に引き出すのに役立つことを願っています。

6 章 タスクとタスクグループ

私たちが、TBB で最も気に入っている点は、その「多重解像度」の性質です。並列プログラミング・モデルのコンテキストにおける多重解像度とは、高レベルのアルゴリズムや低レベルのスレッド化またはタスク化 API の使用など、さまざまな抽象化レベルから選択してアルゴリズムを記述できることを意味します。TBB には、関数を調整するフローグラフ（4 章を参照）と、アルゴリズムがこれらの特定のパターンに適合するとすぐに使用できる `parallel_for` や `pipeline`（2 章を参照）などの高レベルのテンプレートが用意されています。

しかし、アルゴリズムがそれほど単純でない場合はどうなるでしょうか？
もしくは、利用可能な高レベルの抽象化によって並列ハードウェアのパフォーマンスが最大限に引き出されていない場合はどうなるでしょうか？
私たちは、プログラミング・モデルの高レベル機能に縛られたままでいるべきでしょうか？

もちろん、それは違います。ハードウェアの利用、独自のテンプレートをゼロから構築する方法、プログラミング・モデルの低レベルでより調整可能な特性を使用して実装を徹底的に最適化する方法が必要です。TBB にはこの機能が存在します。この章では、TBB の最も強力な低レベル機能の 1 つである `task_group` プログラミング・インターフェイスに焦点を当てます。本書を通して述べてきたように、タスクは TBB の中核であり、`parallel_for` や `pipeline` などの高レベルのテンプレートを構築するために使用される構成要素です。しかし、さらに深いところまで踏み込んで、`task_group` を使用してアルゴリズムを直接コーディングしたり、タスク上で将来使用するために独自の高レベルなテンプレートを構築したり、次の章で説明するように、タスクの実行方法を調整して実装を完全に最適化することを妨げるものは何もありません。本質的に、この章と以降の章を読むことで、これが判明します。深く掘り下げることを楽しんでください！

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_6

サンプルプログラムの実行: フィボナッチ数列

タスクベースの TBB 実装は、特にツリー構造の分解に従って問題を再帰的に小さなサブ問題に分割できるアルゴリズムに適しています。このような問題はたくさんあります。分割統治法と分岐限定法の並列パターン（2 章）は、このようなアルゴリズムのクラスの例です。問題が十分に大きい場合、ハードウェアを最大限に活用して負荷の不均衡を回避するのに十分なタスクに分割することが容易であるため、通常は並列アーキテクチャーで適切にスケールできます。

タスクのサイズとタスクの数が十分に大きいとは、どういうことでしょうか？ これはアプリケーションによって多少異なりますが、オーバーヘッドを償却したい場合は、タスクの実行時間を 1 マイクロ秒より長くすべきであるという経験則を忘れないでください。また、TBB のスケジューラーにハードウェアを活用して負荷分散する柔軟性を持たせたい場合、その柔軟性を提供するのに十分なタスク（少なくともハードウェア・スレッド数の倍数）を作成する必要があります。

この章では、ツリーのようなアプローチで実装できる最も単純な問題を選択しました。この問題はフィボナッチ数列として知られており、`0` と `1` で始まり、数列内のすべての数字が前の 2 つの数字の合計になる整数数列を計算するというものです:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
```

数学的には、シーケンスの `n` 番目の数値 `Fn` は、初期値 `F0=0` および `F1=1` として、

```
Fn = Fn-1 + Fn-2
```

として再帰的に計算できます。`Fn` を計算するアルゴリズムはいくつかありますが、TBB タスクの動作を説明するため、効率的ではありませんが、図 6-1 に示すアルゴリズムを選びました。

```
long fib(long n) {
    if(n<2)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

図 6-1. `Fn` の計算の再帰実装。サンプルコード: `tasks/parallel_invoke_fib.cpp`

フィボナッチ数の計算は、帰納を説明する際によく紹介されるコンピューター・サイエンスの例ですが、また単純なアルゴリズムが非効率であることを顕著に示している例でもあります。より効率良い方法は、

$$F_n = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

を計算して、左上の要素を取得する方法です。行列の累乗は、2 乗を繰り返すことにより、すばやく累乗を行うことができます。しかし、この節では先に進み、学習目的で古典的な再帰の例を使用します。

図 6-1 に示すコードは、`Fn = Fn-1 + Fn-2` を計算する再帰方程式に明らかに類似しています。理解するのは簡単かもしれませんが、`fib(4)` を呼び出す再帰呼び出しツリーを示す図 6-2 でさらに詳しく説明します。

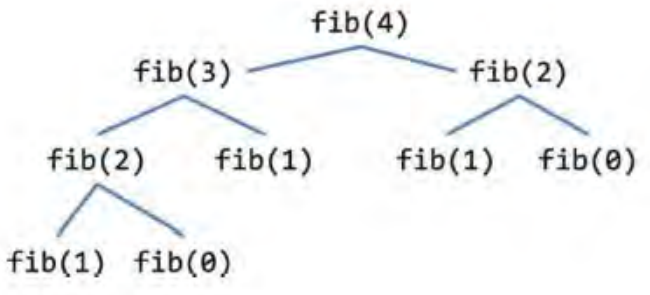


図 6-2. fib(4) の再帰呼び出しツリー

図 6-1 のシリアルコードの先頭にある `if(n<2)` の行は、無限再帰を避けるため再帰コードで常に必要な、いわゆるベースケースに対応しています。これで、スタックを破壊せずに済みます。

ここでは、より単純なバージョンからより精巧で最適化されたバージョンまで、さまざまなタスクベースのアプローチを使用して、この最初のシリアル実装を並列化します。これらの例から学んだ教訓は、他のツリー型または再帰型アルゴリズムにも適用でき、ここで示す最適化は、同様の状況で並列アーキテクチャーを最大限に引き出すために使用できます。

高レベルのアプローチ: parallel_invoke

2 章では、並列タスクを生成する際に、ニーズに適した高レベルのクラスである parallel_invoke について説明しました。このクラスを利用することで、図 6-3 に示すフィボナッチ・アルゴリズムの最初の並列実装を作成します。

```
long parallel_fib(long n) {
  if(n<2) {
      return n;
  }
  else {
      long x, y;
      tbb::parallel_invoke([&]{x=parallel_fib(n-1);},
                           [&]{y=parallel_fib(n-2);});
      return x+y;
  }
}
```

図 6-3. parallel_invoke 使用したフィボナッチの並列実装。サンプルコード: tasks/parallel_invoke_fib.cpp

parallel_invoke 関数は、parallel_fib(n-1) と parallel_fib(n-2) を再帰的に呼び出し、2 つのラムダ関数で参照によって取得されたスタック変数 x と y に結果を返します。これら 2 つのタスクが完了すると、呼び出し元のタスクは x+y の合計を返すだけです。実装の再帰的な性質により、n<2 のときに基本ケースに達するまで並列タスクが呼び出され続けます。つまり、TBB は、それぞれ 1 と 0 を返すだけの parallel_fib(1) と parallel_fib(0) を計算するタスクも作成します。本書を通して述べてきたように、十分な数のタスクを作成するアーキテクチャーに十分な並列性を持たせたいのですが、同時に、タスク作成のオーバーヘッドが償却されるように、タスクの粒度は最低限（1 マイクロ秒以上）である必要があります。このトレードオフは通常、図 6-4 に示すように、「カットオフ」パラメーターを使用してアルゴリズムに実装されます。

```
long parallel_fib_cutoff(log n) {
  if(n<cutoff) {
     return fib(n);
  }
  else {
     long x, y;
     tbb::parallel_invoke([&]{x=parallel_fib_cutoff(n-1);},
                          [&]{y=parallel_fib_cutoff(n-2);});
      return x+y;
  }
}
```

図 6-4. parallel_invoke 実装のカットオフバージョン。サンプルコード: tasks/parallel_invoke_fib.cpp

考え方は、n が十分に大きくない場合（n<cutoff）にタスクの作成を停止し、代わりにシリアル・アルゴリズムを呼び出すように基本ケースを変更することです。適切な cutoff 値を計算するにはある程度の実験が必要です。そのため、cutoff を入力パラメーターにして、適切な値を容易に検索できるようにコードを記述することをお勧めします。例えば、私たちのテスト環境では、fib(30) は約 1 ミリ秒しかかからず、これは分割を抑制するのに十分に細かいタスクです。このカットオフによりタスクのサイズが大きくなりますが、fib(40) のテストケースでハードウェアをビジー状態に保つのに十分なタスクが生成されます。私たちの実験に基づくと、cutoff=30 を設定するのが合理的であり、その結果、図 6-5 に示すように、n=29 および n=28 を受け取るタスクではコードのシリアルバージョンが呼び出されます。

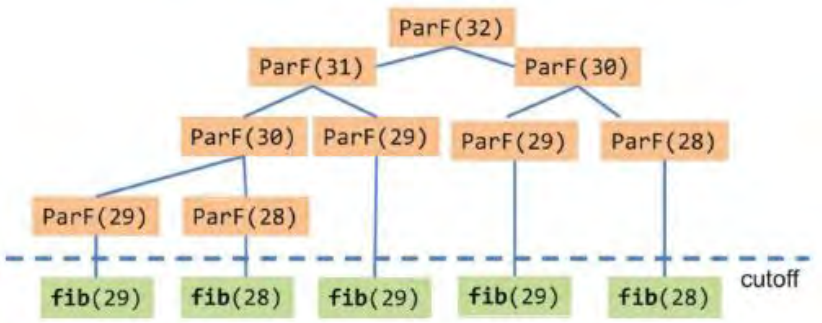


図 6-5. parallel_fib(32) を呼び出した後の呼び出しツリー – 図ではスペースを節約するため ParF(32) を表示 – fib() はシリアルに実装された基本ケースです

図 6-5 を見て、3 つの異なるタスクで fib(29) を計算し、さらに 2 つのタスクで fib(28) を計算するのは無意味と思われるでしょう。免責事項として、これは最適な実装ではなく、学習上の目的に役立つ、一般に使用される再帰的な例であることを述べました。

明らかな最適化は、すでに計算されたフィボナッチ数が再度計算されないように再帰を実装し、最適な *O(n)* の複雑さを達成することですが、それはここでの目標ではありません。また、図 6-4 を見た後で、2 章ですでに説明した `parallel_invoke` をもう一度取り上げるのを疑問に思うかもしれませんが、心配ありません。これは、`task_group` を使用してタスクを直接使用する手順を始めるために導入する小さな例にすぎません。

低レベルのアプローチ: task_group

図 6-6 は、クラス `task_group` のメンバー関数を示しています。`task_group` は、oneTBB でタスクを作成および管理する方法です。以前のバージョンの TBB の低レベルの古いタスク API である `tbb::task` に精通している場合は、12 章で比較と移行の例を見つけることができます。

```
namespace tbb {

    class task_group {
    public:
        task_group();
        task_group(task_group_context& context);

        ~task_group();

        template<typename Func> void run(Func&& f);

        template<typename Func> task_handle defer(Func&& f);

        void run(task_handle&& h);

        template<typename Func>
        task_group_status run_and_wait(const Func& f);

        task_group_status run_and_wait(task_handle&& h);

        task_group_status wait();
        void cancel();
    };

    bool is_current_task_group_canceling();

} // namespace tbb
```

図 6-6. [scheduler.task_group] で説明されている task_group のクラス定義。Func 型は ISO C++ 標準の [function.objects] セクションに記載されている関数オブジェクトの要件を満たす必要があります

task_group に依存するフィボナッチ・コードの再実装を図 6-7 に示します。一見すると、これは単に、parallel_invoke を使用した図 6-4 のコードをより冗長に実装する方法にすぎません。ただし、parallel_invoke の代替方法とは異なり、タスクのグループ g へのハンドルが用意されていることを強調しておきます。後ほど説明するように、これにより、タスクのキャンセルなどが可能になります。また、メンバー関数 g.run() や g.wait() を明示的に呼び出すことで、新しいタスクを生成し、2 つの異なるプログラム ポイントで計算が完了するまで待機します。run() と wait() を分離することで、呼び出し元のスレッドは、いくつかのタスクを生成してからブロック呼び出し wait() で、それらのタスクを待つ間に計算を行うことができます。

```
long  parallel_fib(long  n) {
  if(n<cutoff) {
    return  fib(n);
  }
  else  {
    long  x, y; tbb::task_group g;
     g.run([&]{x=parallel_fib(n-1);});  // タスクをスポーン
     g.run([&]{y=parallel_fib(n-2);});  // 別のタスクをスポーン
     g.wait();  // 両方のタスクの完了を待機
     return  x+y;
  }
}
```

図 6-7. task_group に基づく並列フィボナッチ。サンプルコード: tasks/task_group_fib.cpp

さらに、task_group には、便利なメンバー関数も用意されています。

void run_and_wait(const Func& f) は {run(f); wait();} と同等ですが、f が現在のスレッドで実行されることを保証します。この複合関数を選択すると、実装で TBB スケジューラーをバイパスできるようになります。最初に run(f) を呼び出すと、基本的に、ワーカースレッドのローカルキューにエンキューされるタスクが生成されます。wait() を呼び出すときに、その間に他の誰もタスクをスチールしていない場合は、エンキューされたばかりのタスクをデキューするスケジューラーを呼び出します。

run_and_wait の目的は 2 つあります。1 つ目は、エンキュー、スケジュール、デキューの手順によるオーバーヘッドを回避できること、2 つ目は、タスクがキュー内にある間に発生するスチールを回避できることです。

void cancel() は、task_group 内のすべてのタスクをキャンセルします。計算は「キャンセル」ボタンも含まれているユーザー・インターフェイス（UI）からトリガーされたかもしれません。ユーザーがこのボタンを押すと、計算を停止することができます。9 章では、キャンセルと例外処理についてさらに詳しく説明します。

task_group 内のすべてのタスクが完了するかキャンセルされるまで、task_group_status wait() はブロックします。task_group の最終ステータスを返します。戻り値は、完了（グループ内のすべてのタスクが終了）、またはキャンセル（task_group がキャンセル要求を受信した）になります。

図 6-7 の並列実装では、parallel_fib を呼び出すたびに新しい task_group が作成されるため、9 章で説明するように、他のブランチに影響することなく 1 つのブランチをキャンセルできます。

単一の task_group を使用する場合、単一のタスクまたはスレッドからすべてのタスクを順番に送信しないように注意する必要があります。例えば、図 6-8 のようなコードを記述したくなるかもしれません。

```
#include <tbb/tbb.h>

void f();

void serializedCallsToRun(tbb::task_group& g, int depth) {
    int n = 1<<depth;
    for (int i = 1; i < n; ++i) {
        g.run([]() { f(); });
    }
}

void serialCalls(int depth) {
  tbb::task_group g;
  serializedCallsToRun(g, depth);
  g.wait();
}
```

図 6-8. task_group::run へのシリアル化された呼び出しは、単一のスレッドからすべてのタスクをスポーンするためボトルネックを発生させます。サンプルコード: tasks/task_group_poor_scaling.cpp

図 6-8 に示すように、n 個のタスクが同じスレッドから次々に生成されます。他のワーカースレッドは、g.run() を実行しているワーカースレッドによって作成されたすべてのタスクを強制的にスチールすることになります。これは、特に f() が細粒度のタスクであり、ワーカースレッド数が多い場合、パフォーマンスを確実に低下させます。推奨される代替案は、タスクの再帰的な展開が実行される図 6-7 の方法です。このアプローチでは、ワーカースレッドは計算の開始時にスチールし、理想的には、log2(p) ステップ（p はスレッド数）で、すべての p 個のスレッドが自身のタスクで作業し、それらのタスクはローカルキューにさらに多くのタスクをエンキューします。例えば、p=4 の場合、最初のスレッド A は 2 つのタスクを生成し、1 つのタスクでワークを開始し、スレッド B がもう 1 つのタスクをスチールします。ここで、スレッド A と B はそれぞれ 2 つのタスク（合計 4 つ）を生成し、そのうちの 2 つでワークを開始しますが、残りの 2 つはスレッド C と D によってスチールされます。それ以降は、4 つのスレッドすべてがワークを実行し、ローカルキューにさらに多くのタスクをエンキューし、ローカルタスクが不足した場合にのみ再度スチールを行います。ローカルキューからワークを実行する場合、同期のオーバーヘッドは低くなり、キャッシュの局所性は高くなる傾向があります。このタイプのタスク分散の利点については、10 章の図 10-11 で詳しく説明します。

図 6-8 のコードを変更して、図 6-9 に示すように、タスクをツリーとして再帰的に生成できます。私たちのテストシステムでは、図 6-8 と 6-9 の両方で f() の呼び出しの合計数が同じであるにもかかわらず、図 6-9 のツリーベースのタスク作成では、シリアルケースと同じ数のタスクを 30% の時間で実行できました。

```
#include <tbb/tbb.h>

void f();

void treeCallsToRun(tbb::task_group& g, int depth) {
  if (depth>1) {
    g.run([&g, depth]() {
      treeCallsToRun(g, depth-1);
    });
    g.run([&, depth]() {
      treeCallsToRun(g, depth-1);
    });
  }
  f();
}

void treeCalls(int depth) {
  tbb::task_group g;
  treeCallsToRun(g, depth);
  g.wait();
}
```

図 6-9. task_group を使用して、異なるワーカースレッド内からタスクを再帰的に生成する再帰アルゴリズム。サンプルコード: tasks/task_group_poor_scaling.cpp

task_group 関数はスレッドセーフです

図 6-9 のコードは、task_group のメンバー関数がスレッドセーフであるため機能します。したがって、異なるスレッドが同じ task_group オブジェクトに対して run と wait を並行して安全に呼び出すことができます。それでも、run と wait の呼び出しが重複する可能性がある場合、異なるスレッド（またはタスク）から同じ task_group にアクセスする場合は注意が必要です。関数 wait は、task_group 内で実行中のタスクがなくなるとリターンします。あるスレッドが run を呼び出しているのとほぼ同時に別のスレッドが wait を呼び出している場合、run が wait の呼び出しの前か後かを判断する競合が発生する可能性があります。wait の前に run が発生した場合、そのタスクが完了するまで wait はリターンしません。

最初に wait が発生し、wait が呼び出されたときにタスク数がゼロである場合、まだ実行されていないタスクを待たずにリターンします。

図 6-9 には危険な競合はありません。タスクは他のタスク内から再帰的に追加されるため、これらのタスクの 1 つから g.run が呼び出された場合、task_group g で実行されているタスク数が 0 になることはありません。

延期されたタスク

task_group の利点の 1 つは、タスクの実行（task_group::run）と、コード内でタスクの完了を待機するポイント（task_group::wait）を分離できることだと説明しました。遅延タスクは、コード分離の別のポイントを提供します。延期されたタスクは tbb::task_handle で表され、（task_group::defer）を呼び出すことで作成されます。tbb::task_handle を使用すると、タスクをすぐにスポーンせずに定義するための別のポイントを分離できます。図 6-6 では、タスクを生成する task_group 内の関数 task_group::run と task_group::run_and_wait に、関数オブジェクトの代わりに tbb::task_handle を受け取るオーバーロードがあることが分かります。

tbb::task_handle は、ユーザーボディーへの単なるポインターではありません。TBB がタスクのスケジュールと実行に使用する構造体へのハンドルでもあります。

図 6-10 は、図 6-7 のフィボナッチの例を拡張したものですが、遅延タスクを使用しています。この例では、タスクの定義に別の関数を作成する必要はないように見えますが、タスクを作成するのに複雑なロジックがある場合は、そのロジックを別の関数にカプセル化すると便利なことがあります。task_group::defer と tbb::task_handle を使用すると、コードの一部でタスクを定義し、後で別の部分でタスクを実行できます。

```
tbb::task_handle  make_task(tbb::task_group& g, long& r, long n);

long parallel_fib(long n) {
  if(n<cutoff) {
    return fib(n);
  }
  else {
    long x, y;
    tbb::task_group g;
    tbb::task_handle h1 = make_task(g, x, n-1);
    tbb::task_handle h2 = make_task(g, y, n-2);
    g.run(std::move(h1));
    g.run(std::move(h2));
    g.wait();     // 両方のタスクの完了を待機
    return x+y;
  }
}

tbb::task_handle make_task(tbb::task_group& g, long& r, long n) {
  return g.defer([&r,n]{r=parallel_fib(n);});
}
```

図 6-10. すべてのタスクを再帰的に生成し、遅延タスクを使用してタスクの定義とタスクの実行を分離する task_group を基にする並列フィボナッチ。サンプルコード: tasks/task_group_fib_defer.cpp

以前のバージョンの TBB の最も低いレベルのタスク API による最適化に精通している場合は、継続受け渡し、タスクのリサイクル、およびスケジューラーのバイパスを使用する既存のコードの移行方針として遅延タスクを使用できます。これらの技術の移行の詳細については、12 章を参照してください。

依存関係のあるタスク

一般的に、フローグラフは現在、TBB で依存関係を表現する高レベルの方法です。それでも、必要に応じて手動で参照カウントを実装し、準備が整ったら tbb::task_group を使用してタスクを実行できます。2 章では、実行準備ができたらさらにワークを追加するため、tbb::parallel_for_each で明示的な参照カウントを使用する方法について説明しました。また、2 章では、tbb::parallel_for_each と参照カウントを使用した前方置換の例を実装しました。図 6-11 と 6-12 に示すように、tbb::task_group を使用して実行されるタスクに対しても同様のことを行うことができます。

```
const int block_size = 512;
using BlockIndex = std::pair<size_t, size_t>;

void parallelFwdSubTaskGroup(std::vector<double>& x,
                             const std::vector<double>& a,
                             std::vector<double>& b) {
  const int N = x.size();
  const int num_blocks = N / block_size;

  // 参照カウントを作成
  std::vector<std::atomic<char>> ref_count(num_blocks*num_blocks);
  ref_count[0] = 0;
  for (int r = 1; r < num_blocks; ++r) {
    ref_count[r*num_blocks] = 1;
    for (int c = 1; c < r; ++c) {
      ref_count[r*num_blocks + c] = 2;
    }
    ref_count[r*num_blocks + r] = 1;
  }

  BlockIndex top_left(0,0);

  tbb::task_group tg; tg.run([&]() {
    fwdSubTGBody(tg, N, num_blocks, top_left, x, a, b, ref_count);
  });
  tg.wait();
}
```

図 6-11. タスクを実行する前に参照カウントを使用して依存関係を追跡する task_group に基づく前方置換。サンプルコード: tasks/task_ group_with_dependencies.cpp。前方置換問題については 2 章で詳しく説明されており、依存関係は図 2-14 に示されています。

```
void fwdSubTGBody(tbb::task_group&  tg,
                  int N, int num_blocks,
                  const std::pair<size_t, size_t> bi,
                  std::vector<double>& x,
                  const std::vector<double>& a,
                  std::vector<double>& b,
                  std::vector<std::atomic<char>>& ref_count) {
  auto [r, c] = bi;
  computeBlock(N, r, c, x, a, b);
  // 準備ができたらサクセサーを右に追加
  if (c + 1 <= r && --ref_count[r*num_blocks + c + 1] == 0) {
    tg.run([&, N, num_blocks, r, c]() {
      fwdSubTGBody(tg, N, num_blocks,
                   BlockIndex(r, c+1), x, a, b, ref_count);
    });
  }
  // 準備ができたらサクセサーを下に追加
  if  (r + 1  < (size_t)num_blocks
       && --ref_count[(r+1)*num_blocks + c] == 0) {
    tg.run([&, N, num_blocks, r, c]() {
      fwdSubTGBody(tg, N, num_blocks,
                   BlockIndex(r+1, c), x, a, b, ref_count);
    });
  }
}
```

図 6-12. task_group に基づく前方置換サンプルに使用されるタスクのボディー。サンプルコード: tasks/task_group_with_dependencies.cpp。前方置換問題については 2 章で詳しく説明されており、computeBlock のコードは図 2-15 に示されています

図 6-11 では、std::atomic 変数を使用して、タスクの実行準備状況を追跡しています。2 章で詳しく説明されているように、この例のタスクは、その左側と上にあるブロックが完了すると実行準備完了です。図 6-12 は、これらの依存関係を追跡し、依存関係が満たされると各ブロックを処理するタスクを実行する fwdSubTGBody 関数の実装を示しています。

タスクグラフのコントロールを追加しますか？

TBB の古いバージョンでは、最も低いレベルのタスク・アプリケーションで参照カウントが特別にサポートされており、これを使用して低レベルのタスクグラフを作成できました。また、古いタスク・アプリケーションは、低レベルの最適化を実装するのにも使用できました。

この書籍の発行時点では、このサポートは（まだ?）現在のバージョンの TBB では利用できません。TBB コミュニティーの一部の人々は、タスクの依存関係を作成および管理する直接的なアプローチを再導入する API に関心を示しており、その方向で作業が進行中です。進捗状況の詳細については、oneTBB コミュニティーの議論に注目してください。それまでは、12 章にある、これらの古い低レベルのタスクベースの実装と最適化の多くを移行する方法についてガイダンスを提供します。

タスクの一時停止と再開

一般に、TBB タスクは非プリエンプティブに実行されます。TBB スケジューラーは、ワーカースレッドで現在実行中のタスクを停止せず、代わりに別のタスクの実行を開始します。つまり、タスクが何かを待機してブロックされている場合（例えば、I/O 操作の終了やアクセラレーターへのオフロードの完了など）、タスクは待機中に TBB ワーカースレッドの 1 つを占有します。

TBB は計算の並列処理を目的として設計されているため、非プリエンプティブな実行はほとんどの場合に意味を持ちますが、常にそうであるとは限りません。

意味をなさない場合、開発者は TBB の*再開可能*なタスクサポートによってタスクを一時停止し、後で再開してワーカースレッドを解放できます。これは協調スレッド形式であり、さまざまなユーザー関数がタスクを一時停止し、続行に必要な結果が準備できた時点でタスクを再開することができます。API の 2 つの関数は、`tbb::task::suspend` と `tbb::task::resume` です:

```
using tbb::task::suspend_point =  /* 実装定義 */;
template < typename Func  > void tbb::task::suspend( Func );
void tbb::task::resume( tbb::task::suspend_point );
```

`tbb::task::suspend` 関数はタスク内から呼び出されます。タスクは、汎用アルゴリズム、フローグラフ、または `task_group` によって作成できます。`tbb::task::suspend` は現在のコンテキストをキャプチャーし、`tbb::task::suspend_point` を引数としてユーザー提供の関数を呼び出して、現在のタスクの実行を中断します。例えば、`tbb::task::suspend` は `parallel_for` のボディー内から呼び出されることがあります:

```
tbb::parallel_for(0, N, [&](int) {
   tbb::task::suspend(
       [&] (tbb::task::suspend_point  tag) {
          async_activity.submit(tag);
       });
       // 再開されるとここで実行を開始:
       next_thing_to_do_after_resumed();
});
```

ユーザー提供の関数は完了まで実行されますが、`tbb::task::resume(tag)` が呼び出されるまで、プログラムフローは `suspend` の呼び出しから戻りません。これにより、中断されたタスクを実行していたスレッドが解放され、他の TBB タスクを実行できるようになります。

中断されたタスクは、一致するタグで `tbb::task::resume(tag)` を呼び出すことで、同じスレッドまたは他のスレッドによって再開できます:

```
tbb::task::resume(tag);
```

再開されると、タスクは、中断された呼び出しが返された場所から再び実行を開始するようスケジュールされます。タスクが中断ポイントから再開された場合、中断を呼び出した同じスレッドが中断ポイント後に継続される保証はないことに注意する必要があります。TBB 実装では、トップレベルの TBB 呼び出し（メインスレッドから行われた `tbb::parallel_for`、`tbb::flow::graph::wait_for_all`、`tbb::task_ arena::execute` 呼び出しなど）の場合、同じスレッドがコードを実行することを保証します。しかし、ネストされたケースではそのような保証はありません。

5 章では、フローグラフで `tbb::flow::async_node` を使用して、アクセラレーターでワークが完了するのを待機する間にワーカースレッドがブロックされるのを防ぎました。図 6-13 は、再開可能なタスクを使用した修正バージョンの例を示しています。

```
#include <iostream>
#include <utility>
#include <sycl/sycl.hpp>
#include <tbb/tbb.h>

int main() {
   const int N = 1000;
   int num_iterations = 3;

   try {
      sycl::queue sycl_q;

      tbb::parallel_for(0, num_iterations,
         [&](int id) {
         int *a = sycl::malloc_shared<int>(N, sycl_q);
         tbb::task::suspend([=,&sycl_q](tbb::task::suspend_point tag) {
            auto sycl_event = sycl_q.fill(a, id, N);
            sycl_q.submit([=](sycl::handler& sycl_h) {
               sycl_h.depends_on(sycl_event); // sycl_event の完了後に実行
               sycl_h.host_task([tag]() {
                  tbb::task::resume(tag);
               });
            });
         });
         if (std::any_of(a, a+N, [id](int i) { return i != id; })) {
            std::printf("ERROR: unexpected fill result\n");
         } else {
            std::printf("Well-formed fill for id %d\n", id);
         }
      }
      );
   } catch (const sycl::exception&) {
      std::cout << "No CPU SYCL device on this platform\n";
   }
   return 0;
}
```

図 6-13. tbb::task::suspend は、SYCL を使用してワークがデバイスにオフロードされている間に、parallel_for で作成されたタスクがブロックされるのを防ぐために使用されます。SYCL ホストタスク（SYCL ランタイムによって実行される）は、ワークが完了すると tbb::task::resume を呼び出してタスクを再開します。サンプルコード: tasks/resumable_tasks.cpp

図 6-13 では、parallel_for の各反復で配列が割り当てられ、その配列が SYCL キューを使用して SYCL デバイスにオフロードされ値が格納されます。5 章と同様に、この例はまだ有用なワークを実行せず、デモのみを目的としています。

また、5 章と同様に、SYCL コード自体については説明しません。SYCL についてさらに詳しく知りたい場合は、James Reinders が執筆に協力した『*Data Parallel C++*』という書籍をお勧めします。

図 6-13 の `tbb::task::suspend` に渡される関数は、カーネルを送信して `fill` 操作を実行し、その完了に依存する別の SYCL タスクをアタッチします。この依存タスクは、デバイスでのワークが完了すると、`resume` を呼び出して `parallel_for` タスクを再開します。

まとめ

この章では、高レベルな TBB アルゴリズムの外部でタスクを実行する方法として `task_group` を紹介しました。再帰アルゴリズムで使用してタスクツリーを生成したり、参照カウントを使用してタスクグラフを作成できることを示しました。古い TBB の低レベルタスク API に精通している開発者は、12 章で `task_group` に移行する方法に関する追加情報を参照できます。

また、非同期ワークを待機しているワーカースレッドを解放できる協調スレッドの機能である再開可能なタスクについても触れました。再開可能なタスクは、高レベルの TBB アルゴリズム、`task_group`、またはフローグラフによって生成されたタスクで機能します。5 章では、フローグラフに固有の同様の機能である `async_node` について説明しました。

7 章 メモリー割り当て

この章では、並列プログラムの*重要*な部分である、スケーラブルなメモリー割り当てについて説明します。これには、new や、malloc、calloc などの明示的な呼び出しが含まれます。スケーラブルなメモリー割り当ては、スレッディング・ビルディング・ブロック（TBB）の他の部分を使用するかどうかにかかわりなく使用できます。TBB は、直接使用するインターフェイスに加え、動的なメモリー割り当てに C/C++ 関数を自動的に置き換えるプロキシー・ライブラリーも提供しています。これは、コードを変更せずにパフォーマンスを向上させる簡単で効果的な方法です。これは重要なことであり、C++ の経験の程度に関係なく機能します。具体的には、推奨されるスマートポインター（std::unique_ptr、std::shared_ptr、または std::weak_ptr）を使用するか、現在では推奨されない生のポインターを使用するかにかかわりなく機能します。スケーラブルなメモリー・アロケーターを使用することで得られるパフォーマンスの利点は、スケーリングを制限したり、偽の共有（フォルス・シェアリング）のリスクを引き起こす問題に直接対処するため、非常に大きなものです。TBB は、広く使用された最初のスケーラブル・メモリー・アロケーターの 1 つです。これは、並列プログラムにおけるメモリー割り当ての重要性を強調するため、TBB に付属していたことが大きな理由です。これは現在でも非常に人気がある、最もスケーラブルなメモリー・アロケーターの 1 つです。

最新の C++ プログラミング（スマートポインター優先）と並列思考を組み合わせると、std::allocate_shared を使用して明示的に、または std::make_shared を使用して暗黙的に、TBB のスケーラブル・メモリー・アロケーターを使用することが推奨されます。

現代の C++ メモリー割り当て

パフォーマンスは並列プログラミングにおいて特に重要ですが、*正当性*は *すべての*アプリケーションにとって重要なトピックです。メモリーの割り当て/割り当て解除の問題は、アプリケーションのバグの大きな原因であり、これにより C++ 標準に多くの機能が追加されることで、現代の C++ プログラミングが変化してきました。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_7

最新の C++ プログラミングでは、C++11 でのスマートポインターの導入（make_shared、allocate_shared など）により、管理されたメモリー割り当てが*推奨され*、malloc や new の多用は*推奨されていません*。本書では、最初の章から、例中で std::make_shared を使用してきました。C++17 で std::aligned_alloc が追加され、偽の共有を回避するキャッシュ・アラインメントが利用できますが、スケーラブルなメモリー割り当てには対応していません。将来の C++ に向けて多くの追加機能が開発中ですが、スケーラビリティーは明示的にサポートされていません。

TBB は、並列プログラマーにとって重要な要素である*スケーラブルなメモリー割り当て*を提供し続けます。TBB は、C++ と C 標準のすべてのバージョンに完全に適合する方法でこれを行います。TBB のサポートの核心は、*スレッドによるメモリープーリング*であると言えます。このプーリングにより、キャッシュ間の不必要なデータの移動によって引き起こされるパフォーマンスの低下を回避できます。TBB は、キャッシュ・アライメントと組み合わせたスケーラブルなメモリー割り当ても提供しており、単純に std::aligned_alloc を使用するよりもスケーラブルな属性を提供します。キャッシュ・アライメントは、考えなく使用するとメモリー使用量が大幅に増加する可能性があるため、デフォルトの動作とはされていません。

この章で説明するように、スケーラブルなメモリー割り当てはパフォーマンス面で非常に重要になります。std::make_shared ではアロケーターを指定できませんが、std::allocate_shared ではアロケーターを指定できます。

ここでは、アプリケーションでどのような C++ メモリー割り当て方法を選択した場合でも使用すべきスケーラブルなメモリー・アロケーターに焦点を当てています。並列思考を取り入れた最新の C++ プログラミングでは、TBB スケーラブル・メモリー・アロケーターで std::allocate_shared を明示的に使用したり、TBB スケーラブル・メモリー・アロケーターを使用するためデフォルトで暗黙的に std::make_shared を使用する new をオーバーライドして TBB で使用することが推奨されます。std::make_shared は、クラスのコンテンツとブックキーピング用の追加スペース（具体的には、スマートポインターにするため追加されるアトミック）の両方を管理するため、実際にはより大きなメモリーブロックを割り当てることで特定クラスの new オペレーターの影響を受けないことに注意してください。そのため、デフォルトの new をオーバーライドする（TBB アロケーターを使用する）だけで、std::make_shared に影響を与えることができます。

使用方法	概要	インターフェイス一覧
C/C++ プロキシー	最も一般的な使用法。標準的なメモリー割り当て方法の自動置き換え。コードの変更は不要です。	図 7-4 に、プロキシー・ライブラリーで置き換えられる関数の一覧が示します。
C 関数	C 標準関数（例: `malloc`）。	図 7-10 の関数一覧。
C++ クラス	C++ 標準インターフェイス（`std:allocator`）。	図 7-12 のクラス一覧。
パフォーマンス最適化の調整	ラージページの使用など、特定のニーズに合わせて（あらゆる使用方法で）パフォーマンスを微調整する方法。究極のパフォーマンスを実現する最適化に役立ちます。	図 7-14 にリストされている機能インターフェイスと環境変数。

図 7-1. TBB スケーラブル・メモリー・アロケーターの使用方法

スケーラブルなメモリー割り当て: 何を

この章では、図 7-1 に示す 4 つのカテゴリーで TBB のスケーラブルなメモリー機能について説明します。4 つのカテゴリーの機能は自由に組み合わせることができます。ここでは、すべての機能を説明するためカテゴリーに分類しています。C/C++ プロキシー・ライブラリーは、スケーラブルなメモリー・アロケーターを使用する最も一般的な方法です。

スケーラブル・メモリー・アロケーターはスレッディング・ビルディング・ブロックの他の機能とは明確に分かれているため、同時使用で利用するメモリー・アロケーターが並列アルゴリズムとコンテナー・テンプレートの選択に依存しないよう独立しています。

スケーラブルなメモリー割り当て: なぜ

本書全体でワークを並列に実行することでアプリケーションのパフォーマンスを向上させる方法を説明していますが、スレッド非対応のメモリー割り当てと解除によって、努力が無駄になってしまう可能性があります。並列プログラムで慎重なメモリー割り当てが重要であるのは、アロケーターの競合とキャッシュ効果の 2 つです。

スレッド化されていないアロケーターが使用された場合、各スレッドが単一のグローバルヒープからそれぞれのメモリー割り当てと解除を行う際にグローバルロックで競合が発生します。そのため、マルチスレッド・プログラムで深刻なボトルネックが発生します。
このようなメモリー・アロケーターを使用するプログラムはスケーラブルではありません。その競合のため、メモリー割り当てを集中的に行うプログラムはプロセッサー・コア数の増加とともに速度が低下することがあります。スケーラブルなメモリー・アロケーターは、さらに洗練されたデータ構造を使用して競合を大幅に軽減することでこの問題を解決します。

もう 1 つの問題であるキャッシュ効果は、メモリーアクセスにはデータをキャッシュするハードウェアのメカニズムがあることで発生します。したがって、プログラムでデータを使用すると、データをキャッシュする場所に影響します。スレッド B にメモリーを割り当て、アロケーターがスレッド A によって最近解放されたメモリーを提供する場合、意図せずにキャッシュからキャッシュにデータがコピーされる可能性が高く、アプリケーションのパフォーマンスが意図せず低下する可能性があります。さらに、個別のスレッドのメモリー割り当てが近すぎると、キャッシュラインが共有される可能性があります。共有は、*真の共有*（同じオブジェクトを共有する）または*偽の共有*（オブジェクトは共有されないが、オブジェクトが同じキャッシュラインに含まれる）として説明できます。どちらのタイプの共有もパフォーマンスに特に大きな悪影響を及ぼす可能性がありますが、共有が意図されていないため回避できる*偽の共有（フォルス・シェアリング）*は特に重要です。クラス `cache_aligned_allocator<T>` を使用して、常にキャッシュラインから割り当てを開始し、必要に応じて随時再調整されるスレッドごとのヒープを維持することで、フォルス・シェアリングを回避できます。これは、前述の競合問題の解決にも役立ちます。

スケーラブルなメモリー・アロケーターを使用すると、パフォーマンスが 20～30% 向上するメリットが得られます。また、極端なケースでは、スケーラブルなメモリー・アロケーターと再リンクするだけでプログラムのパフォーマンスが 4 倍になるというケースもあります。

パディングによってフォルス・シェアリングを回避

データ構造の内部でフォルス・シェアリングによる問題が発生する場合、パディングが必要です。8 章では、ヒストグラムの例を使用します。ヒストグラムのバケットとバケットのロックはどちらも、複数のタスクが単一のキャッシュラインでデータを更新できるようにメモリー内に密にパックされたデータ構造です。

データ構造におけるパディングの考え方は、複数のタスクによって更新される隣接する要素を共有しないように、要素間に十分な間隔を空けることです。

フォルス・シェアリングに対して、最初に取るべき対策は、図 7-2 に示すように、共有ヒストグラム（図 8-20 を参照）を宣言するときに、`std::allocator` や `malloc` ではなく、`tbb::cache_aligned_allocator` または `std::aligned_alloc` を使用することです。

```
std::vector<atom_bin, tbb::cache_aligned_allocator<atom_bin>>
   hist_p(num_bins);
```

図 7-2. アトミックの単純なヒストグラム・ベクトル。サンプルコード: synchronization/histogram_07_3rd_safe_parallel_cache_aligned.cpp

ただし、これはヒストグラム・ベクトルの先頭を揃え、hist_p[0] をキャッシュラインの先頭に配置するだけです。これは、hist_p[0]、hist_p[1]、. . .、hist_p[15] が同じキャッシュラインに格納されることを意味します。ここで、スレッドが hist_p[0] をインクリメントし、別のスレッドが hist_p[15] をインクリメントすると、フォルス・シェアリングが起こります。この問題を解決するには、ヒストグラムの各位置、各ビンが完全なキャッシュラインを占有するようにします。これは、図 7-3 に示すパディングによって実現できます。

```
struct  atom_bin  {
  alignas(128) std::atomic<int>  count;
};
std::vector<atom_bin, tbb::cache_aligned_allocator<atom_bin>>
  hist_p(num_bins);

std::for_each(image.begin(), image.end(),
              [&](uint8_t  i){hist[i]++;});
```

図 7-3. アトミックのヒストグラム・ベクトル内のパディングを使用して、フォルス・シェアリングを排除します。サンプルコード: synchronization/histogram_07_3rd_safe_parallel_cache_ aligned.cpp

図 7-3 から分かるように、ビンの配列 hist_p は構造体のベクトルになっており、各構造体にはアトミック変数が含まれ、アラインメントは 128 に設定され、インテル・プロセッサーのキャッシュラインのフォルス・シェアリングの安全な実装が保証されています（現時点では）。

フォルス・シェアリングのないデータ構造は、元のデータ構造より 16 倍多くのスペースを占有します。これは、コンピューター・プログラミングで頻繁に発生する空間と時間のトレードオフのもう 1 つの例です。つまり、メモリーの占有量は増えますが、コードは高速になります。その他の例としては、より小さなコードとループのアンロール、関数の呼び出しと関数のインライン化、圧縮データの処理と非圧縮データの処理などがあります。

これにより、alignas() メソッドのおかげで、hist_p の各ビンが完全にキャッシュラインを占有することが保証されます。あと一つだけ！私たちは移植可能なコードを書くのが大好きです。異なるアーキテクチャーまたは将来のアーキテクチャーで、キャッシュライン・サイズが異なる場合はどうなるでしょうか。問題ありません。C++17 標準には私たちが求めている解決策があります:

```
struct  atom_bin  {
   alignas(std::hardware_destructive_interference_size)
      std::atomic<int> count;
};
std::vector<atom_bin, tbb::cache_aligned_allocator<atom_bin>>
  hist_p(num_bins);
```

*フォルス・シェアリング*の問題が解決されたと仮定すると、真の共有の問題はどうでしょうか？

2 つの異なるスレッドが最終的に同じビンをインクリメントし、それが 1 つのキャッシュから別のキャッシュにピンポンされることになります。これを解決するにはもっと良いアイデアが必要です! この問題への対処方法については、8 章のプライベート化とリダクションで説明します。

スケーラブルなメモリー割り当ての代替

最近では、スケーラブルなメモリー割り当てをサポートするのは、TBB だけではありません。この節では最も人気のあるオプションを紹介します。並列プログラミングに TBB を使用する場合、TBB が提供するものであろうと他のものであろうと、スケーラブルなメモリー・アロケーターを使用することが不可欠です。TBB を使用して作成されたプログラムは、任意のメモリー・アロケーター・ソリューションを利用できます。

TBB は、他の並列プログラミング手法と並んでスケーラブルなメモリー割り当てを促進する最初の並列プログラミング手法でした。これは、TBB の作成者が、あらゆる並列プログラムでメモリー割り当てを考慮することの重要性を理解していたためです。
TBB のメモリー・アロケーターは現在でも非常に人気があり、間違いなく利用可能な最もスケーラブルなメモリー・アロケーターの 1 つです。

TBB のスケーラブル・メモリー・アロケーターは、スレッディング・ビルディング・ブロック（TBB）の他の機能を使用するかどうかに関係なく使用できます。同様に、TBB は任意のスケーラブルなメモリー・アロケーターでも動作できます。

TBB のスケーラブル・メモリー・アロケーターの最も一般的な代替は、`jemalloc`、`tcmalloc`、`llalloc` です。TBB のスケーラブルなメモリー・アロケーターと同様に、スケーラブルな同時実行サポートを提供しながら断片化の回避に重点を置く `malloc` の代替があります（メモリー割り当ての断片化は、空きメモリーが小さな非連続ブロックの集合となり、割り当て要求を満たすのが困難になることで発生する、メモリーの非効率的な使い方です）。これら 3 つはすべて、自由なライセンス（BSD または Apache）でオープンソースとして利用できます。

自身のアプリケーションで `tbbmalloc` を他と比較した結果、自身のアプリケーションでは `tbbmalloc` の方が優れていることが判明したと言う人もいます。これはかなり一般的なことです。ただし、TBB を広範囲に使用する場合でも、`jemalloc`、`tcmalloc` または `llalloc` を選択する人もいます。これもまた機能します。選択するのは皆さんです。

`jemalloc` は FreeBSD `libc` のアロケーターです。最近では、ヒープ・プロファイリングや広範な監視/チューニング・フックなどの開発者サポート機能が追加されました。
`jemalloc` は Facebook で使用されています。

tcmalloc は、tcmalloc といくつかのパフォーマンス解析ツールを含む Google の gperftools の一部です。tcmalloc は Google で使用されています。

Lockless Inc. の llalloc は、オープンソースのロックレス・メモリー・アロケーターとして無料で入手できます。また、クローズド・ソース・ソフトウェアで使用するため購入することもできます。

個々のアプリケーションの動作、特にメモリー割り当てと解放のパターンにより、これらのオプションからすべてに適合する単一のコンポーネントを選択することは不可能です。TBBmalloc、jemalloc、tcmalloc、llalloc のいずれを選択しても、スケーラブルでない種類のものであれば、デフォルトの malloc 関数や new オペレーターよりはるかに優れていると確信しています（これは、時間の経過とともに問題ではなくなってきています。現在では多くの malloc がスレッドコード向けに最適化されていますが、TBB が初めて登場したときには、いずれも最適化されていませんでした）。

コンパイルの注意事項

インテル・コンパイラーまたは gcc を使用してプログラムをコンパイルする場合、次のフラグを指定するのが最適です:

- -fno-builtin-malloc（Windows: /Qfno-builtin-malloc）
- -fno-builtin-calloc（Windows: /Qfno-builtin-calloc）
- -fno-builtin-realloc（Windows: /Qfno-builtin-realloc）
- -fno-builtin-free（Windows: /Qfno-builtin-free）

これは、コンパイラーが独自のビルトイン関数を使用していると仮定して最適化を行う可能性があるためです。他のメモリー・アロケーターを使用する場合、これらの仮定は当てはまらない可能性があります。これらのフラグを使用しなくても問題は発生しない可能性がありますが、安全のために使用しています。使用するコンパイラーのドキュメントを確認してください。

最も人気のある使用法（C/C++ プロキシー・ライブラリー）: どのように

プロキシー・ライブラリーを使用すると、動的メモリー・インターフェイス置換テクニックを使用して、new/delete および malloc、calloc、realloc、free などのルーチンをグローバルに置き換えることができます。動的メモリー割り当てのため malloc やその他の C/C++ 関数を自動的に置き換える方法は、TBB のスケーラブル・メモリー・アロケーター機能を使用する最も一般的な方法です。また、非常に効果的でもあります。

tbbmalloc_proxy ライブラリーを使用して、malloc、calloc、realloc、free など（完全なリストは図 7-4 を参照）、と new/delete を置き換えできます。この方法の使用は簡単で、ほとんどのプログラムで十分です。各オペレーティング・システムで使用されるメカニズムの詳細は若干異なりますが、実質的な効果はいずれも同じです。ライブラリー名は図 7-5 に示されます。

	Linux	macOS	Windows
グローバル C++ 演算子 new と delete を置き換え	はい	はい	はい
標準 C ライブラリー関数: malloc、calloc、realloc、free	はい	はい	はい
標準 C ライブラリー関数 (C11 で追加): aligned_alloc	はい		
標準 POSIX 関数: posix_memalign	はい	はい	
プラットフォームに応じて他の機能も置き換えられます (以下に、公開時点での最新のリストを示します。追加/変更がある場合は、TBB 開発者ガイドに記載されます)。			
GNU C ライブラリー (glibc) 固有の関数: malloc_usable_size、__libc_malloc、__libc_calloc、__libc_memalign、__libc_free、__libc_realloc、__libc_pvalloc、__libc_valloc	はい		
Microsoft C ランタイム・ライブラリー関数: Msize、_aligned_malloc、_aligned_realloc、_aligned_free、_aligned_msize			はい
valloc	はい	はい	
malloc_size		はい	
Memalign、pvalloc、mallopt	はい		

図 7-4. プロキシーによって置き換えられるルーチンのリスト

	リリース・バージョン	デバッグ・バージョン
Linux	libtbbmalloc_proxy.so.2	libtbbmalloc_proxy_debug.so.2
macOS	libtbbmalloc_proxy.dylib	libtbbmalloc_proxy_debug.dylib
Windows	tbbmalloc_proxy.dll	tbbmalloc_proxy_debug.dll

図 7-5. プロキシー・ライブラリーの名前

Linux: malloc/new プロキシー・ライブラリーの使い方

Linux では、LD_PRELOAD 環境変数を使用してプログラムロード時にプロキシー・ライブラリーをロードするか、メインの実行ファイルをプロキシー・ライブラリー（-ltbbmalloc_proxy）にリンクして置換できます。
Linux プログラムローダーは、プログラムロード時にプロキシー・ライブラリーとスケーラブル・メモリー・アロケーター・ライブラリーを検索可能でなければなりません。検索可能にするには、ライブラリーを含むディレクトリーを LD_LIBRARY_PATH 環境変数に追加するか、/etc/ld.so.conf に追加します。動的メモリー置換には次の 2 つの制限があります: (1) glibc のメモリー割り当てフック（__malloc_hook など）はサポートされておらず、(2) Mono（Microsoft .NET Framework のオープンソース実装）はサポートされていません。

Windows: malloc/new プロキシー・ライブラリーの使い方

Windows では、実行可能ファイルを変更する必要があります。ソースコードに #include を追加してプロキシー・ライブラリーを強制的にロードするか、図 7-6 に示すように特定のリンカーオプションを使用することができます。
Windows プログラムローダーは、プログラムロード時にプロキシー・ライブラリーとスケーラブル・メモリー・アロケーター・ライブラリーを検索可能でなければなりません。検索可能にするには、ライブラリーを含むディレクトリーを PATH 環境変数に追加します。

tbbmalloc_proxy.h をインクルードします（これにより、アプリケーションの起動時に読み込まれます）。

```
#include <tbb/tbbmalloc_proxy.h>
```

または、バイナリー（アプリケーションの起動時に読み込まれる）のリンカーオプションに次のパラメーターを追加します。これらは、アプリケーションの起動時に読み込まれる EXE ファイルまたは DLL に対して指定できます:

win32 向け (3 つの下線):

```
tbbmalloc_proxy.lib /INCLUDE:"___ TBB_malloc_proxy"
```

win64 向け (2 つの下線):

```
tbbmalloc_proxy.lib /INCLUDE:"__TBB_malloc_proxy"
```

図 7-6. Windows でプロキシー・ライブラリーを使用する方法（注: win32 と win64 では下線の数が異なります）

Windows 開発者には、デバッグを支援する次の 2 つの追加機能があります:

1. TBB_MALLOC_DISABLE_REPLACEMENT 環境変数を 1 に設定すると、設定されている間はプログラム呼び出しの置換が無効になります。この場合、プログラムは標準の動的メモリー割り当て関数を使用します。oneTBB のメモリー割り当てライブラリーは、使用が無効になっている場合でも、プログラムの起動に必要であることに注意してください。
2. TBB_malloc_replacement_log 関数は、動的メモリー置換が行われたかどうかを示すログを作成します。これは、Windows での置換の性質（他のオペレーティング・システムとは大きく異なる）により、Windows に特有のものです。図 7-7 に例を示します。

```
#include "tbb/tbbmalloc_proxy.h"
#include <stdio.h>

int main () {
  char **func_replacement_log;
  int func_replacement_status =
        TBB_malloc_replacement_log (&func_replacement_log);

  if (func_replacement_status != 0) {
    printf ("tbbmalloc_proxy cannot replace memory allocation routines\n");
    for (char  ** log_string = func_replacement_log;
         *log_string != 0;
         log_string++)
      printf("%s\n",*log_string);
    }
  return  0;
}
```

出力例:

```
tbbmalloc_proxy cannot replace memory allocation routines
Success: free (ucrtbase.dll), byte pattern: <C74424100000000008B4424>
Fail: _msize (ucrtbase.dll), byte pattern:
<E90B000000CCCCCCCCCCCC>
```

図 7-7. Windows 置換ログ・ユーティリティー関数 TBB_malloc_replacement_log。サンプルコード: memalloc/windows_proxy.cpp

プロキシー・ライブラリーの使用テスト

プログラムがより高速な割り当てを活用しているか確認する簡単な二重チェックとして、マルチコアマシンで図 7-8 のテストプログラムを使用できます。図 7-9 では、このテストを実行する方法と、Ubuntu Linux を実行するクアッドコアの仮想マシンで確認されたタイミングの違いを示しています。Windows 上で直接、Visual Studio の「パフォーマンス・プロファイラー」を使用したところ、スケーラブル・メモリー・アロケーターなしでは 94 ミリ秒、スケーラブル・メモリー・アロケーターあり（`tbb_mem.cpp` に `#include <tbb/tbbmalloc_proxy.h>` を追加）では 50 ミリ秒でした。タイミングを計測する唯一の目的は、正しく構成されていることを確認するためです。これらすべての実行は、このテストによって、スケーラブルなメモリー・アロケーターの挿入が機能していること（新規/削除の場合）と、パフォーマンスが大幅に向上していることを検証できることを示しています。`malloc()` と `free()` を使用するという些細な変更でも、同等の結果が得られます。本書に関連付けられているサンプルプログラムのダウンロードには、`tbb_malloc.cpp` として含まれています。

サンプルプログラムはスタックスペースを大量に使用するため、「`ulimit -s unlimited`」（Linux/macOS）または「`/STACK:10000000`」（Visual Studio:➤ [構成のプロパティ] ➤ [Linker] ➤ [System] ➤ [Stack Reserve Size)] で即時クラッシュを回避します。

```
int  main() {
   double  *a[N];

   tbb::parallel_for(0, N-1, [&](int  i) { a[i] = new  double; });
   tbb::parallel_for(0, N-1, [&](int  i) { delete  a[i];       });

   return  0;
}
```

図 7-8. new/delete の速度をテストする小さなプログラム。サンプルコード: memalloc/tbb_mem.cpp

```
デフォルトでは、環境変数によってプロキシーの使用が制御されます：
g++ -o tbb_mem tbb_mem.cpp -ltbb

time ./tbb_mem
real
0m0.090s user
0m0.301s sys
0m0.099s

export LD_PRELOAD=$TBBROOT/lib/libtbbmalloc_proxy.so
time ./tbb_mem
real   0m0.020s
user   0m0.188s
sys    0m0.039s

unset LD_PRELOAD
time ./tbb_mem
real    0m0.091s
user    0m0.272s
sys     0m0.102s

または… 代わりに （常にプロキシーを使用）：
g++ -o tbb_mem tbb_mem.cpp -ltbb -ltbbmalloc_proxy
time ./tbb_mem
real    0m0.025s
user    0m0.190s
sys     0m0.029s
```

図 7-9. 図 7-8 のコード実行とタイミング – サンプル出力

ファミリー 1	`void *scalable_malloc (size_t size)`	`malloc` に類似
	`void scalable_free (void *ptr)`	`free` に類似
	`void *scalable_realloc (void *ptr, size_t size)`	`realloc` に類似
	`void *scalable_calloc (size_t nobj, size_t size)`	スケーラブルな `malloc` を補完する `calloc` に類似
	`int scalable_posix_memalign (void **memptr, size_t alignment, size_t size)`	`posix_memalign` に類似
ファミリー 2	`void *scalable_aligned_malloc (size_t size, size_t alignment)`	`posix_memalign` に類似
	`void *scalable_aligned_realloc (void *ptr, size_t size, size_t alignment)`	`realloc` に類似の補完スケーラブル `malloc`
	`void scalable_aligned_free (void *ptr)`	以前の `scalable_aligned_malloc` または `scalable_aligned_remalloc` の `free` に類似
ファミリー 3	`size_t scalable_msize (void *ptr)`	`msize/malloc_size/malloc_usable_size` に類似。`scalable_x` によって以前に割り当てられたメモリーブロックの使用可能サイズを返します。`ptr` がそのようなブロックを指していない場合は `0` を返します。
	`int scalable_allocation_mode (int param, intptr_t value)`	TBB アロケーター固有の割り当てモードを設定します。この章の終わりにある「パフォーマンス・チューニング:制御ノブ」という節で説明します。
	`int scalable_allocation_command (int cmd, void *param)`	TBB アロケーター固有のコマンドを呼び出します。この章の終わりにある「パフォーマンス・チューニング:制御ノブ」という節で説明します。

図 7-10. TBB スケーラブル・メモリー・アロケーターが提供する関数

C 関数: C 用のスケーラブルなメモリー・アロケーター

図 7-10 にリストされている関数は、スケーラブルなメモリー・アロケーターへの C レベルのインターフェイスを提供します。
TBB プログラミングでは C++ が使用されるため、これらのインターフェイスは TBB ユーザー向けではなく、C コードで使用するために用意されています。

`scalable_x` ルーチンはそれぞれ、x ライブラリー関数に似た動作を行います。ルーチンは、図 7-11 に示す 2 つのファミリーを形成します。あるファミリーの `scalable_x` 関数で割り当てられたストレージ領域は、C 標準ライブラリー関数ではなく、同じファミリーの `scalable_x` 関数で解放またはサイズ変更を行う必要があります。同様に、C 標準ライブラリー関数で割り当てられた記憶領域は、`scalable_x` 関数で解放またはサイズ変更してはなりません。

これらの関数は、特定の `#include <tbb/scalable_allocator.h>` によって定義されます。

ファミリー	割り当てルーチン	解放ルーチン	類似ライブラリー

1	`scalable_malloc`
	`scalable_calloc`
	`scalable_realloc`
		`scalable_free`	C 標準ライブラリー

	`scalable_posix_memalign`		POISIX
2	`scalable_aligned_malloc` `scalable_aligned_realloc`	`scalable_aligned_free`	Microsoft C ランタイム

図 7-11. ファミリーによる割り当て-解放関数の結合

`tbb::cache_aligned_allocator<T>`	スケーラブルなメモリー割り当て。キャッシュラインの開始位置に合わせてアラインメントされます。フォルス・シェアリングの回避に役立ちますが、アラインメントによってメモリー消費量が増加し、コストも増大する可能性があります。
`tbb::scalable_allocator<T>`	スケーラブルなメモリー割り当て。`TBBmalloc` ライブラリーが利用できない場合、直接呼び出すと失敗します。
`tbb::tbb_allocator<T>`	このクラスは、利用可能な場合は `tbb::scalable_allocator` を選択し、`TBBmalloc` ライブラリーが利用できない場合は標準の `malloc` にフォールバックします。この呼び出しは、`TBBmalloc` ライブラリーが利用できない場合でも機能します。

図 7-12. TBB スケーラブル・メモリー・アロケーターが提供するクラス

C++ クラス: C++ 用のスケーラブルなメモリー・アロケーター

プロキシー・ライブラリーはスケーラブルなメモリー割り当てを採用する包括的なソリューションを提供しますが、それらはすべて、直接使用することを選択できる特定の機能に基づいています。TBB は、割り当て用の C++ クラスを 3 つの方法で提供します:

（1） C++ STL `std::allocator<T>` に必要なシグネチャーを持つアロケーター、

（2） STL コンテナー用のメモリープールのサポート、および

（3） アライメントされた配列用の特定のアロケーター。

std::allocator<T> シグネチャーを持つアロケーター

図 7-12 にリストされているクラスは、スケーラブルなメモリー・アロケーターへの C++ レベルのインターフェイスを提供します。TBB には、C++ 標準の `std::allocator<T>` と同じシグネチャーをサポートする 3 つのテンプレート・クラス（`tbb_allocator`、`cache_aligned_allocator`、および `scalable_allocator`）があります。

これには、C++11 以前の標準 <T> と <void> のサポートが含まれますが、これは C++17 で非推奨になりました。つまり、これらは、ベクトルなど STL クラス・テンプレートで使用される割り当てルーチンとして渡すことができます。これら 4 つのクラスはすべて、C++ のすべての「アロケーター要件」を満たす概念をモデル化していますが、ISO C++ コンテナーで使用するため標準で要求される保証も備えています。

scalable_allocator

scalable_allocator クラス・テンプレートは、プロセッサー数にスケールするようにメモリーの割り当てと解放を行います。std::allocator の代わりに scalable_allocator を使用すると、プログラムのパフォーマンスが改善されます。scalable_allocator で割り当てたメモリーは、std::allocator ではなく scalable_allocator で解放してください。

scalable_allocator アロケーター・クラス・テンプレートでは、TBBmalloc ライブラリーが使用可能でなければなりません。ライブラリーが利用できない場合、scalable_allocator の呼び出しは失敗します。
対照的に、メモリー・アロケーター・ライブラリーが利用できない場合、他のアロケーター（tbb_allocator または cache_aligned_allocator）は malloc および free にフォールバックします。

このクラスは #include <tbb/scalable_allocator.h> で定義されており、（通常は）すべてを包含する tbb/tbb.h には含まれません。

tbb_allocator

tbb_allocator クラス・テンプレートは、TBBmalloc ライブラリーが使用可能である場合は、それを介してメモリーを割り当て、解放します。そうでない場合は、malloc と free を使用します。cache_aligned_allocator と zero_allocator は tbb_allocator を使用するため、malloc で同じフォールバックを提供しますが、scalable_allocator は提供しないため、TBBmalloc ライブラリーが使用できない場合は失敗します。このクラスは、#include <tbb/tbb_allocator.h> で定義されています。

cache_aligned_allocator

cache_aligned_allocator クラス・テンプレートは、スケーラビリティーとフォルス・シェアリングに対する保護の両方を提供します。割り当てが異なるキャッシュラインで行われていることを確認することでフォルス・シェアリングに対応します。

そのため、フォルス・シェアリングが問題になりそうな場合のみ、cache_aligned_allocator<T> を使用してください（図 7-2 を参照）。

cache_aligned_allocator の機能は、たとえ小さなオブジェクトであっても、キャッシュライン・サイズの倍数を割り当てるため、ある程度のスペースが必要になります。パディングは通常 128 バイトです。そのため、cache_aligned_allocator で多くの小さなオブジェクトを割り当てると、メモリーの使用量が増加します。

tbb_allocator と cache_aligned_allocator の両方を試して、特定のアプリケーションで得られるパフォーマンスを測定することをお勧めします。

2 つのオブジェクト間におけるフォルス・シェアリングの回避は、両方のオブジェクトが cache_aligned_allocator で割り当てられている場合にのみ保証されることに注意してください。例えば、1 つのオブジェクトが cache_aligned_allocator<T> によって割り当てられ、他のオブジェクトが別の方法で割り当てられている場合、cache_aligned_allocator<T> はキャッシュライン境界で割り当てを開始しますが、必ずしもキャッシュラインの末尾に割り当てるわけではないため、フォルス・シェアリングに対する保証はありません。配列または構造体が割り当てられる場合、割り当ての先頭部分のみがアライメントされるため、個々の配列または構造体の要素が他の要素と一緒にキャッシュライン上に配置される可能性があります。 この例と、要素を個々のキャッシュラインに強制的に配置するパディングを図 7-3 に示します。

このクラスは、#include <tbb/cache_aligned_allocator.h> で定義されています。

new と delete の選択的な置き換え

カスタム new/delete オペレーターを開発する理由は、エラーチェック、デバッグ、最適化、使用状況統計の収集など、いくつか挙げられます。

new/delete は、個々のオブジェクトとオブジェクトの配列に対してさまざまなバリエーションで実行されると考えることができます。さらに、C++11 では、それぞれについて、スローするバージョン、スローしないバージョン、配置バージョンが定義されています。グローバルセット（::operator new、::operator new[]、::operator delete、::operator delete[]）またはクラス固有のセット（クラス X の場合、X::operator new、X::operator new[]、X::operator delete、X::operator delete[]）のいずれかです。最後に、C++17 では、すべてのバージョンの new にオプションの配置パラメーターが追加されました。

すべての new/delete オペレーターをグローバルに置き換え、カスタムニーズがない場合は、プロキシー・ライブラリーを使用します。これには、malloc/free および関連する C 関数を置き換えるという利点もあります。

カスタムニーズの場合、グローバル・オペレーターではなく、クラス固有のオペレーターをオーバーロードするのが最も一般的です。この節では、特定のニーズに合わせてカスタマイズできるグローバルな new/delete オペレーターを置き換える方法を例として示します。スローするバージョンとしないバージョンを示しましたが、配置バージョンは実際にはメモリーを割り当てないため、オーバーライドしませんでした。
また、アライメント（C++17）パラメーターを含むバージョンも実装していません。同じ概念

を使用して、個々のクラスの new/delete オペレーターを置き換えることもできます。その場合、配置バージョンとアライメント機能を実装することを選択できます。プロキシー・ライブラリーが使用されている場合、それらすべては TBB によって管理されます。

図 7-13 は、new と delete の置き換え方法を示しています。new と delete のすべてのバージョンを一度に置き換える必要があります。つまり、new の 4 つのバージョンと delete の 4 つのバージョンが置き換えられます。もちろん、スケーラブルなメモリー・ライブラリーとのリンクが必要です。

この例では、スレッドの安全性に問題があるため、新しいハンドラーを無視することを選択しています。基本的なシグネチャーの改良版には、割り当てが失敗した場合に例外をスローせず NULL を返すことを意味する追加パラメーター const std::nothrow_t& が含まれます。これらの例外をスローしないオペレーターは、C ランタイム・ライブラリーで使用できます。

```
#include <tbb/scalable_allocator.h>

// scalable_malloc がメモリー割り当てに必要なすべてを行うと想定しているため、
// 再試行ループはありません。
// そのため、これを繰り返し呼び出しても状況は全く改善されません

void* operator new (size_t  size, const  std::nothrow_t&)
{
  if (size  == 0) size  = 1;
  if (void* ptr  = scalable_malloc
    (size)) return  ptr;
  return  NULL;
}

void* operator new[] (size_t  size, const  std::nothrow_t&)
{
  return operator new  (size, std::nothrow);
}
void  perator delete  (void* ptr, const  std::nothrow_t&)
{
  if (ptr  != 0) scalable_free  (ptr);
}

void operator delete[]  (void* ptr, const  std::nothrow_t&)
{
  operator delete  (ptr, std::nothrow);
}
```

図 7-13. new オペレーターと delete オペレーターの置き換えのデモ。サンプルコード: memalloc/replace_new_n_delete.cpp

パフォーマンス・チューニング: 制御ノブ

TBB は、OS からの割り当て、ヒュージページのサポート、内部バッファーのフラッシュに関する特別な制御を提供します。これらはそれぞれ、パフォーマンスを微調整するために提供されています。

ヒュージページ（Windows では *ラージページ*）は、非常に大量のメモリーを使用するプログラムのパフォーマンスを向上させるために使用されます。ヒュージページを使用するには、プロセッサーとそれをサポートするオペレーティング・システムが必要であり、アプリケーションがヒュージページを活用できるよう何らかの操作を行う必要があります。幸いなことに、ほとんどのシステムはこの要件を満たしており、TBB にはヒュージページのサポートが組み込まれています。

ヒュージページとは?

ほとんどの場合、プロセッサーは一般にページ単位で一度に 4K バイトのメモリーを割り当てます。仮想メモリーシステムはページテーブルを介して、アドレスを実際のメモリー位置にマップします。あまり詳細に触れずに言えば、アプリケーションが使用するメモリーのページ数が増えるほど、必要なページ記述子の数も増え、多くのページ記述子が使用されると、さまざまな理由でパフォーマンスの問題が発生します。この問題を解決するため、最新のプロセッサーは 4K よりもはるかに大きいページサイズ（2MB など）をサポートしています。2GB のメモリーを使用するプログラムの場合、4K ページで 2GB のメモリーに変換するには 524,288 個のページ記述子が必要です。2MB の記述子を使用すると 1,024 個のページ記述子で済み、1GB の記述子では 2 ページ記述子のみが必要です。もちろん、完璧なものはなく、ラージページは場合によってはパフォーマンスを低下させる可能性があります。マルチソケット・システムでは、ヒュージページを有効にすると、OS が大きなページ全体（例: 2MB）をコピーする必要があるため、特定のテクニック（例: 自動 NUMA、メモリー階層化）を使用する場合、ソケット（NUMA ターゲット）間のメモリー移行にかかるコストが高くなる可能性があります。いつものように、特定のアプリケーションの動作に基づいて実際の結果を確認するためにチューニングする場合、ある程度の注意が必要です。

TBB のヒュージページのサポート

TBB のメモリー割り当てでヒュージページを使用するには、`scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES,1)` を呼び出すか、`TBB_MALLOC_USE_HUGE_PAGES` 環境変数を `1` に設定して、明示的に有効にする必要があります。環境変数は、標準の `malloc` ルーチンを `tbbmalloc_proxy` ライブラリーに置き換えるときに便利です。

これらは、TBB のスケーラブル・メモリー・アロケーターのすべての使用法（プロキシー・ライブラリー、C 関数、C++ クラスなど）に関係なくアルゴリズムを微調整する方法を提供します。これらの関数は、環境変数の設定よりも優先されます。これらは決してカジュアルな用途を目的としたものではありません。「コントロールを望む利用者」向けに用意されており、特定のニーズに合わせてパフォーマンスを最適化する優れた方法を提供します。これらの機能を使用する場合、ターゲット環境でアプリケーションへのパフォーマンスの影響を慎重に評価することをお勧めします。

どちらの方法でも、システム/カーネルがヒュージページを割り当てるように構成されていることを前提としています。TBB のメモリー・アロケーターは、事前割り当て済みページと透過的ヒュージページもサポートしており、これらは Linux カーネルによって自動的に割り当てられます。ヒュージページは万能薬ではありません。適切に使用されていない場合、パフォーマンスに悪影響を与える可能性があります。

図 7-14 にリストされている関数は、`#include <tbb/tbb_allocator.h>` で定義されています。

`int scalable_allocation_mode (int mode, intptr_t value)`
`mode=`
`TBBMALLOC_USE_HUGE_PAGES` または
`TBBMALLOC_SET_SOFT_HEAP_LIMIT`
	TBB アロケーター固有の割り当てモードを設定します。

環境変数: `TBB_MALLOC_USE_HUGE_PAGES`	値が「1」の場合、オペレーティング・システムでサポートされている場合、アロケーターによるヒュージページの使用が有効になります。
`int scalable_allocation_command (int cmd, void *reserved)` `reserved` はゼロにする必要があります	TBB アロケーター固有のコマンドを呼び出します。

図 7-14. TBB スケーラブル・メモリー・アロケーターの動作を改良する方法

scalable_allocation_mode(int mode, intptr_t value)

`scalable_allocation_mode` 関数は、スケーラブル・メモリー・アロケーターの動作を調整するために使用されます。次の 2 つの段落で説明する引数は、TBB アロケーターの動作を制御します。この関数は、操作に成功すると `TBBMALLOC_OK` を返しますが、`mode` が次のいずれでもない、または値が指定されたモードで有効でない場合は `TBBMALLOC_INVALID_PARAM` を返します。記述された条件が適用される場合に、`TBBMALLOC_NO_EFFECT` の戻り値が返る可能性があります（各関数の説明を参照）。

TBBMALLOC_USE_HUGE_PAGES

```
scalable_allocation_mode(TBBMALLOC_USE_HUGE_PAGES,1)
```

この関数がオペレーティング・システムでサポートされている場合、アロケーターのヒュージページを有効にします。2 番目のパラメーターに 0 を指定すると無効になります。TBB_MALLOC_USE_HUGE_PAGES 環境変数を 1 に設定すると、scalable_allocation_mode を呼び出してこのモードを有効にするのと同じ効果があります。scalable_allocation_mode で設定されたモードは環境変数よりも優先されます。プラットフォームでヒュージページがサポートされていない場合、この関数は TBBMALLOC_NO_EFFECT を返します。

TBBMALLOC_SET_SOFT_HEAP_LIMIT

```
scalable_allocation_mode(TBBMALLOC_SET_SOFT_HEAP_LIMIT, size)
```

この関数は、アロケーターがオペレーティング・システムから取得するメモリー量に、size バイトのしきい値を設定します。しきい値を超えると、アロケーターは内部バッファーからメモリーを解放します。ただし、TBB スケーラブル・メモリー・アロケーターが必要に応じてメモリーを要求するのを妨げるものではありません。

int scalable_allocation_command(int cmd, void *param)

scalable_allocation_command 関数は、最初の引数で指定された動作を実行するようスケーラブル・メモリー・アロケーターに指示するコマンドとして使用されます。2 番目の引数は予約済みでありゼロ (0) でなければなりません。この関数は、操作が成功した場合は TBBMALLOC_OK を返し、reserved がゼロでない場合、または cmd が定義されたコマンド (TBBMALLOC_CLEAN_ALL_BUFFERS または TBBMALLOC_CLEAN_THREAD_BUFFERS) でない場合、TBBMALLOC_INVALID_PARAM を返します。次に説明するように、TBBMALLOC_NO_EFFECT が返される可能性があります。

TBBMALLOC_CLEAN_ALL_BUFFERS

```
scalable_allocation_command(TBBMALLOC_CLEAN_ALL_BUFFERS, 0)
```

この関数は、アロケーター内部のメモリーバッファーを消去するため、メモリー・フットプリントを減らす可能性があります。これにより、後続のメモリー割り当て要求にかかる時間が増えます。このコマンドは頻繁に使用されることを想定していません。パフォーマンスに与える影響を慎重に評価することを推奨します。バッファーが解放されなかった場合、関数は

TBBMALLOC_NO_EFFECT を返します。

TBBMALLOC_CLEAN_THREAD_BUFFERS

```
scalable_allocation_command(TBBMALLOC_CLEAN_THREAD_BUFFERS, 0)
```

この関数は、呼び出しスレッドの内部メモリーバッファーのみをクリーンアップします。これにより、後続のメモリー割り当て要求にかかる時間が長くなる可能性があります。パフォーマンスへの影響を慎重に評価することをお勧めします。バッファーが解放されなかった場合、関数は TBBMALLOC_NO_EFFECT を返します。

まとめ

スケーラブルなメモリー・アロケーターは、あらゆる並列プログラムにとって不可欠な要素です。パフォーマンス上の利点は驚くほど大きくなります。スケーラブルなメモリー・アロケーターがないと、割り当ての競合、フォルス・シェアリング、その他の無駄なキャッシュ間転送により、深刻なパフォーマンスの問題が発生することがあります。TBB スケーラブルなメモリー割り当て（TBBmalloc）機能には、new や malloc などの明示的な呼び出しが含まれており、これらは直接使用することも、TBB のプロキシー・ライブラリー機能を使用して自動的に置き換えることもできます。TBB のスケーラブルなメモリー割り当ては、TBB の他の部分を使用するかどうかに関係なく使用できます。また、TBB の残りの部分は、どのメモリー・アロケーター（TBBmalloc、tcmalloc、jemalloc、llalloc、malloc など）が呼び出されていても使用できます。

8 章 同期

同期について理解しておくべき 3 つの重要な点は次のとおりです:

1. 必要な場合にのみ使用してください。そうしないと、一部のアルゴリズムが確実に動作しなくなります。
2. 並列処理によるパフォーマンスの可能性を最大限に高めるため、可能な限り同期を回避する方法を見つけます。
3. アルゴリズムの選択は、どの程度の同期が必要であるか、ということに大きく関係します。

この章では、相互排他を実現する同期メカニズムと代替手段について説明します。これにより、必要に応じてそれらを使用できるようになります。

効果的な並列アルゴリズムは、同期戦略と実装に影響します。この章では、さまざまな同期オプションを確認します。これらのオプションはすべてには、使うべきタイミングと場所があります。簡単なヒストグラムの例を見ると、素晴らしいオプションもあれば、ひどいオプションもあることが分かります（スピードアップを望んでいると仮定した場合）。

同様に重要なことは、同期の必要性を最小限に抑えるためアルゴリズムを再考するプロセスを説明することです。この例では、ミューテックスを利用する単純なアプローチを使用する単純なコードから始めて、アトミック操作を活用するように進化させ、さらにプライベート化とリダクションのテクニックを利用してスレッド間の同期を削減します。後者では、競合の激しい相互排他オーバーヘッドを回避する方法として、スレッド・ローカル・ストレージ（TLS）を活用する方法を示します。この章では、「ロック」、「共有可変状態」、「相互排他」、「スレッドの安全性」、「データ競合」など同期に関連する用語の概念について、ある程度理解していることを前提としています。そうでない場合は、本書の序文と用語集でそれらを簡単に紹介していますのでお読みください。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_8

TBB ではアトミック操作が提供されなくなりました（レガシーコードの更新方法については、12 章で詳しく説明します）。これは、最新の C++ が、ポータブルなアトミック操作をサポートしているためです。この章では `std::atomic` について説明します。TBB は、さまざまなスタイルのスケーラブルなミューテックスを提供しています。これは、最新の C++ の標準機能は、本格的な並列プログラミングを間違いなく改善できるからです。この章では、`tbb::` と `std::` の両方のミューテックスについて説明します。それでも、可能であれば、なぜそれらの使用を避けるべきか説明するよう努めます。

同期をどのように行うか（`std::`、`tbb::` など）にかかわりなく、この章の重要なポイントは、アルゴリズムを慎重に再考すると、パフォーマンスが大幅に向上する、よりクリーンな実装が実現できる場合が多くある、ということです。この教訓を深く理解できるまで、この章を繰り返し読むことを強く勧めます。

実行例: 画像のヒストグラム

さまざまな種類の相互排他（mutex）オブジェクトまたはアトミックを実装したり、ほとんどの同期操作を完全に回避できる簡単な例から始めましょう。これらすべての可能な実装について、それぞれの長所と短所を説明し、それらを使用したミューテックス、ロック、アトミック変数、およびスレッド・ローカル・ストレージの使用方法を説明します。

ヒストグラムにはさまざまな種類がありますが、特に画像やビデオデバイス、画像処理ツールでは、画像ヒストグラムが最も広く使用されています。例えば、ほとんどすべての写真編集アプリケーションでは、図 8-1 に示すように、写真のヒストグラムを表示するパレットを容易に見つけることができます。

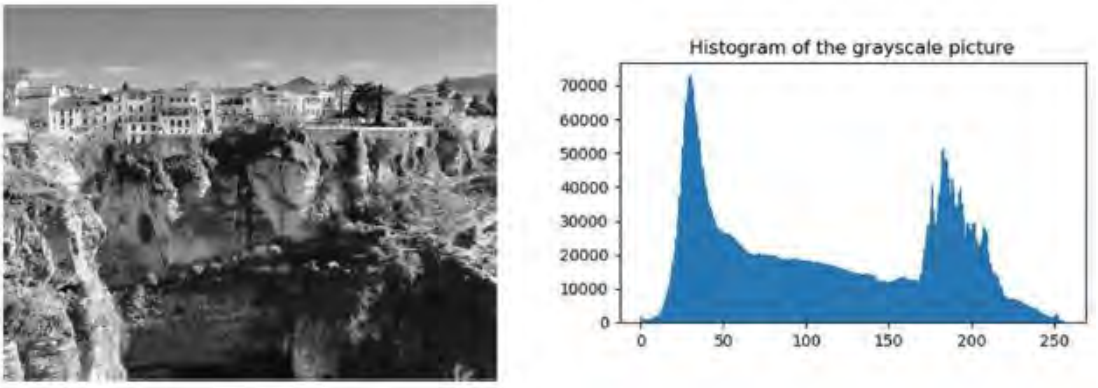


図 8-1. グレースケール画像（ロンダ、マラガ）とそれに対応する画像ヒストグラム

分かりやすくするためグレースケール画像を想定します。この場合、ヒストグラムは、各輝度値（x 軸）を持つピクセル数（y 軸）を表します。画像のピクセルがバイトとして表現される場合、256 のトーンまたは輝度値のみが可能で、0 が最も暗いトーン、255 が最も明るいトーンになります。図 8-1 では、画像内で最も頻繁に出現するトーンは暗いトーンであることが分かります。5M ピクセルのうち、70,000 個以上がトーン 30 であり、`x = 30` 付近のスパイクで確認できます。写真家や画像専門家は、ピクセルのトーン分布を素早く確認し、写真の黒く塗りつぶされた領域や飽和した領域に画像情報が隠れていないか識別する補助として、ヒストグラムを活用しています。

図 8-2 では、ピクセルが 0 から 7 までの 8 つの異なるトーンしか持たない 4 × 4 画像のヒストグラム計算を示しています。2 次元画像は通常、行優先順序に従って 16 個のピクセルを格納する 1 次元ベクトルとして表されます。異なるトーンは 8 つしかないため、ヒストグラムには 0 から 7 までのインデックスを持つ 8 つの要素のみが必要です。ヒストグラムのベクトル要素は「ビン」と呼ばれることもあり、ここで「分類」して各トーンのピクセルをカウントします。図 8-2 は、特定の画像に対応するヒストグラム `hist` を示しています。ビン番号 1 に格納されている「4」は、画像内のトーン 1 の 4 つのピクセルをカウントした結果です。したがって、画像を走査しながらビンの値を更新する基本操作は、`hist[<tone>]++` です。

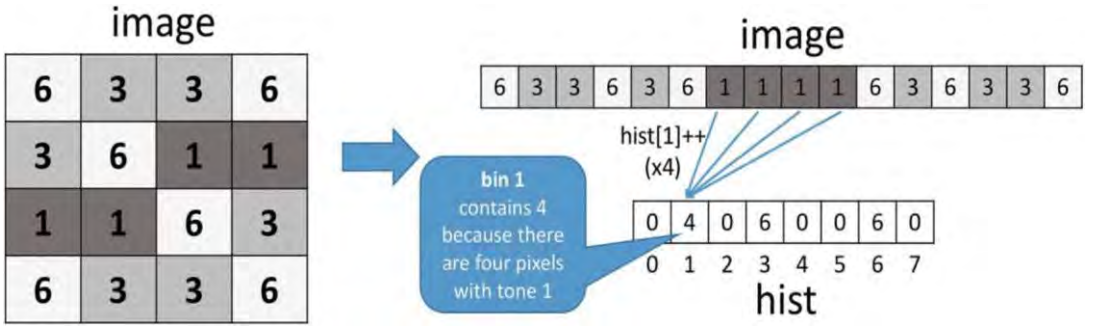


図 8-2. 16 ピクセルの画像からヒストグラム `hist` を計算する（画像の各値はピクセルのトーンに対応）

アルゴリズムの観点から見ると、ヒストグラムは、すべての可能なトーンレベルを考慮するのに十分な要素を持つ整数の配列として表されます。画像がバイト配列であると仮定すると、256 のトーンが存在することになります。したがって、ヒストグラムには 256 個の要素またはビンが必要です。この画像のヒストグラムを計算する順次コードを図 8-3 に示します。

```
int  main() {
  constexpr  long  int  n = 1000000;
  constexpr  int  num_bins = 256;

  // 乱数ジェネレーターの初期化
  std::random_device         // 乱数デバイスシード
  seed; std::mt19937         // mersenne_twister_engine
  mte{seed()};
  std::uniform_int_distribution<> uniform{0,num_bins};
  // 画像を初期化
  std::vector<uint8_t> image;  // 空のベクトル
  image.reserve(n);            // 事前割り当てられた画像ベクトル
  std::generate_n(std::back_inserter(image), n,
                  [&] { return  uniform(mte); }
                );

  // ヒストグラムを初期化
  std::vector<int> hist(num_bins);

  // シリアル実行
  tbb::tick_count t0 = tbb::tick_count::now();
  std::for_each(image.begin(), image.end(),
      [&hist] (uint8_t  i) { hist[i]++; }
  );
  tbb::tick_count t1 = tbb::tick_count::now();
  double  t_serial = (t1 -  t0).seconds();

  std::cout << "Serial time: "  << t_serial << std::endl;
  return  0;
}
```

図 8-3. 画像ヒストグラム計算のシーケンシャル実装を含むコードリスト。関連する記述はボックス内で強調表示されます。サンプルコード: synchronization/histogram_01_sequential.cpp

図 8-3 のコードリストをすでに理解している場合、いくつかの手順をスキップして、この節の最後から読み始めるとよいでしょう。このコードは、サイズ n（メガピクセル画像の場合は 100 万）のベクトル画像を宣言し、乱数ジェネレーターを初期化した後、uint8_t 型の [0,255] レンジの乱数を画像ベクトルに設定します。これには、mersenne_twister_engine、mte を使用し、[0、num_bins) のレンジで均一に分布した乱数を生成し、それを画像ベクトルに挿入します。次に、num_bins 位置（デフォルトではゼロに初期化されます）を使用して hist ベクトルが構築されます。後で image(n) を構築する代わりに、空のベクトル画像を宣言して n 個の整数を予約していることに注意してください。こうすることで、最初にベクトルを走査してゼロで初期化してから、乱数を挿入する必要がなくなります。

実際のヒストグラム計算は、より伝統的なアプローチである、

```
for (int i = 0; i < N; ++i) hist[image[i]]++;
```

のように C/C++ で記述することもできました。これは、ヒストグラム・ベクトルの各ビンに含まれる、あらゆる色調値のピクセル数をカウントします。ただし、図 8-3 の例では、STL for_each アルゴリズムを用いた C++ プログラマーにはより自然である C++ の代替手段を示します。STL for_each アプローチを使用すると、画像ベクトルの各要素（uint8_t 型の色調）がラムダ式に渡され、色調に関連付けられたビンがインクリメントされます。便宜上、ヒストグラムの計算に必要な秒数を求めるため tbb::tick_count クラスを使用します。メンバー関数 now と seconds については、ここではこれ以上説明しません。

安全でない並列実装

ヒストグラム計算を並列化する最初の単純な試みは、図 8-4 に示すように tbb::parallel_for を使用することです。

```
    // 並列実行
    std::vector<int> hist_p(num_bins);
    t0 = tbb::tick_count::now();
    tbb::parallel_for(tbb::blocked_range<size_t>{0, image.size()},
              [&](const  tbb::blocked_range<size_t>& r)
               {
                 for  (size_t  i = r.begin(); i < r.end(); ++i)
                   hist_p[image[i]]++;
               });
    t1 = tbb::tick_count::now();
    double  t_parallel = (t1 -  t0).seconds();

   std::cout << "Serial: "    << t_serial   << ", ";
   std::cout << "Parallel: "  << t_parallel << ", ";
   std::cout << "Speed-up: "  << t_serial/t_parallel << std::endl;

   if  (hist != hist_p)
       std::cerr << "Parallel computation failed!!"  << std::endl;
    return  0;
```

図 8-4. 画像ヒストグラム計算の安全ではない並列実装を含むコードリスト。サンプルコード: synchronization/histogram_02_unsafe_parallel.cpp

図 8-3 のシーケンシャル実装から得られたヒストグラムと並列実行の結果を比較するため、新しいヒストグラム・ベクトル `hist_p` を宣言します。次に、無鉄砲で挑戦的なアイデアは、すべてのピクセルを並列に走査することです...が、なぜいけないのでしょうか？ 独立したピクセルではないからですか？ そのため、2 章で説明した `parallel_for` テンプレートを利用して、異なるスレッドが反復空間の異なるチャンクを走査し、画像の異なるチャンクを読み取ります。しかし、これは機能しません。図 8-4 の最後のベクトル `hist` と `hist_p`（C++ では `hist!=hist_p` は正しい動作です）を比較すると、ロジックの問題により 2 つのベクトルが異なることが分かります（もちろん、ロジックのエラーにもかかわらずランダムに同じになる可能性もありますが、その可能性は低く、実行時には発生していませんでした）:

```
./histogram_02_unsafe_parallel
Serial: 0.253051, Parallel: 9.39047, Speed-up: 0.0269477
Parallel computation failed!!
```

本書の他の例と同様に、サンプル間の相対的な変化を確認できるよう、本書の作成時に確認したタイミングをいくつか共有します。すべてのサンプルコードは、任意のシステムで試行できます。本書に示すプログラミング手法によって、一般的に起こり得る（そして起こるべき）相対的な変化の感覚を理解してもらえれば良いと考えています。皆さんは、自身のシステム構成で表示されるタイミングを常に確認してください。

並列実装では、異なるスレッドが同じ共有ビンを同時にインクリメントする可能性があるため、問題となります。言い換えれば、そのコードはスレッドセーフではありません。より正式には、安全ではない（unsafe）並列コードは「未定義の動作」を示しており、これもコードが正しくないことを意味します。図 8-5 では、コア 0 と 1 で実行されている 2 つのスレッド A と B があり、それぞれがピクセルの半分を処理していると仮定し、問題を提示しています。スレッド A に割り当てられた画像チャンクには輝度 1 のピクセルがあるため、`hist_p[1]++` が実行されます。スレッド B も同じ輝度のピクセルを読み取り、`hist_p[1]++` を実行します。両方のインクリメントが時間的に一致し、一方がコア 0 で実行され、もう一方がコア 1 で実行される場合、インクリメントを見逃す可能性が高くなります。

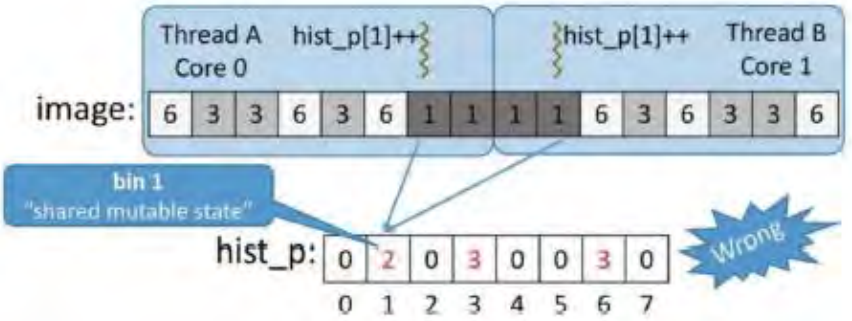


図 8-5. 共有ヒストグラム・ベクトルの安全でない並列更新

これは、インクリメント操作がアトミックではない（または分割不可である）ために発生します。代わりに、通常は 3 つの操作で構成されます。変数をメモリーからレジスターにロードし、レジスターをインクリメントし、レジスターをメモリーに書き戻します。[1] より正式な用語を使用すると、この種の操作は**読み取り - 変更 - 書き込み**、または **RMW** 操作と呼ばれます。

共有変数への同時書き込みは、共有可変状態と呼ばれます。図 8-6 に、C++ の `hist_p[1]++` に対応するマシン命令のシーケンスを示します。

`hist_p[1]++ ->`	**コア 0 の R1**	**コア 1 の R1**	**時間またはサイクル**
`load R1, @(hist_p+1)`	1	1	X
`add R1, R1, #1`	2	2	X+1
`store R1, @(hist_p+1)`	2	2	X+2

図 8-6. 共有変数または共有可変状態の安全でない更新

これら 2 つのインクリメントを実行する時点で、輝度 1 のピクセルがすでに 1 つ見つかっていると、`hist_p[1]` には値 1 が含まれます。この値は、両方のスレッドによって読み取られ、プライベート・レジスターに保存される可能性があり、このビンには 3 つではなく 2 つが書き込まれることになります。これは、これまでに検出された輝度 1 のピクセルの正しい数です。このケースについて考えることは、キャッシュとキャッシュの一貫性を考慮していないため、単純化されすぎていますが、データ競合の問題を説明するのに役立ちます。詳しい例は序文に記載されています（図 16 および 17 を参照）。

[1] フォン・ノイマン・アーキテクチャーの本質により、計算ロジックはデータストレージから分離されているため、データは計算可能な場所に移動し、計算されてから、最終的に再びストレージに戻される必要があります。

この一連の厄介な問題が起こる可能性は低く、あるいは、たとえ起こったとしても、アルゴリズムの並列バージョンを実行したときに、わずかに異なる結果が許容されると考えるかもしれません。報酬はより速い実行でしょうか？ 必ずしもそうではありません。前のページで見たように、安全でない並列実装は、シーケンシャル実装（16 コア・プロセッサーで実行され、n が 10 億ピクセルの場合）より約 10 倍遅くなります。原因は、「序文」で紹介したキャッシュ一貫性プロトコルです（「序文」の「局所性とキャッシュの逆襲」の節を参照）。シリアル実行では、ヒストグラム・ベクトルは、コードを実行しているコアの L1 キャッシュに完全に収まる可能性があります。ピクセルが 100 万個あるため、ヒストグラム・ベクトルには 100 万のインクリメントが存在し、そのほとんどはキャッシュ速度で提供されます。

注：ほとんどの x86 64 ビット・プロセッサーでは、キャッシュラインが 16 個の整数（64 バイト）を保持できます。256 個の整数を持つヒストグラム・ベクトルは、ベクトルが適切にアライメントされていれば、16 個のキャッシュラインのみを必要とします。したがって、16 回のキャッシュミス（プリフェッチが実行されるとさらに少ない回数）の後、すべてのヒストグラム・ビンがキャッシュされ、シリアル実装では各ビンに約 3 サイクル（非常に高速）でアクセスできます（L1 キャッシュが十分に大きく、ヒストグラム・キャッシュラインが他のデータによって排出されないことを前提としています）。

一方、並列実装では、すべてのスレッドがコアごとのプライベート・キャッシュにビンをキャッシュしようとしますが、1 つのスレッドが 1 つのコアの 1 つのビンに書き込むと、キャッシュ・コヒーレンス・プロトコルによって、他のすべてのコアの対応するキャッシュラインに収まる 16 個のビンが無効化されます。この無効化により、無効化されたキャッシュラインへの後続のアクセスには、L1 アクセス時間よりも桁違いに長い時間がかかります。このピンポン相互無効化の最終的な効果は、並列実装のスレッドがキャッシュされていないビンをインクリメントするのに対し、シリアル実装の単一スレッドはほとんど常にキャッシュされたビンをインクリメントすることです。 繰り返しますが、1 メガピクセルの画像にはヒストグラム・ベクトルの 100 万のインクリメントが必要であるため、できるだけ高速なインクリメント実装を作成する必要があります。このヒストグラム計算の並列実装では、偽の共有（例えば、スレッド A が `hist_p[0]` をインクリメントし、スレッド B が `hist_p[15]` をインクリメントする場合、両方のビンが同じキャッシュラインに配置されるため）と真の共有（スレッド A と B の両方が `hist_p[i]` をインクリメントする場合）の両方が見られます。 偽の共有と真の共有については、後の節で説明します。

最初の安全な並列実装: 粗粒度のロック

まず、共有データ構造への並列アクセスの問題を解決します。別のスレッドがすでに同じ変数への書き込み処理を行っているときに、他のスレッドが共有変数を読み書きできないようにするメカニズムが必要です。簡単に言えば、試着室は 1 人でも入室でき、服の着心地を確認してから、退出し、次の人が順番に入れるようにしたいのです。図 8-7 は、試着室のドアが閉じられ、他者が排除されることを示しています。並列プログラミングでは、試着室のドアはミューテックスと呼ばれます。人が試着室に入ると、ドアを閉めて施錠することでミューテックスのロックを取得して保持し、去るときにはドアを開けたままにしておくことでロックを解除します。正確には、ミューテックスは、保護されたコード領域の実行時に相互排他性を提供するために使用されるオブジェクトです。相互排他によって保護する必要があるコード領域は、通常「クリティカル・セクション」と呼ばれます。試着室の例は、図 8-7(c) に示すように、リソース（試着室）が複数の人によって同時に要求されている状態である競合の概念も示しています。 試着室には一度に 1 人しか入れないため、試着室の利用は「シリアル化」されています。同様に、ミューテックスによって保護されているコード領域は、プログラムのパフォーマンスを低下させる可能性があります。第一に、ミューテックス・オブジェクトの管理によって生じるオーバーヘッドが原因であり、第二に、そしてより重要な点として、ミューテックスによって競合とシリアル化が引き起こされる可能性があるためです。同期を可能な限り減らしたい主な理由は、競合とシリアル化を回避するためであり、これによって並列プログラムのスケーリングが制限されます。

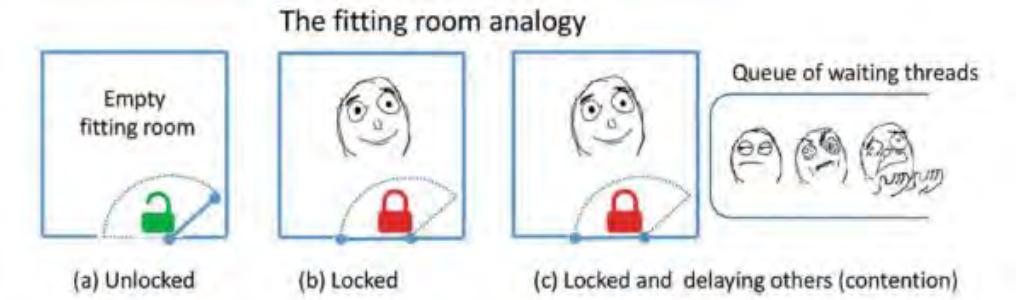


図 8-7. 試着室のドアを閉めると他の人が入れなくなる

この節では、TBB ミューテックス・クラスと同期に関連するメカニズムに注目します。TBB は C++11 より古いものですが、C++11（ミューテックス・クラスのサポートが標準化されていました）では、TBB ライブラリーほどカスタマイズ可能ではありませんでした。

TBB における最も単純なミューテックスは、tbb/spin_mutex.h または包括的な tbb.h ヘッダーファイルをインクルードした後に使用できる spin_mutex です。この新しい機能を使用すると、図 8-8 に示すように、画像ヒストグラム計算の安全な並列バージョンを実装できます。

```
#include <tbb/spin_mutex.h>

// 並列実行
using my_mutex_t=tbb::spin_mutex;
my_mutex_t my_mutex;

parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const tbb::blocked_range<size_t>& r)
             {
               my_mutex_t::scoped_lock my_lock{my_mutex};
               for (size_t i = r.begin(); i < r.end(); ++i)
                 hist_p[image[i]]++;
             });
```

図 8-8. 粗粒度のロックを使用する画像ヒストグラム計算の最初の安全な並列実装のコード。サンプルコード: synchronization/histogram_03_1st_safe_parallel.cpp

my_mutex のロックを取得したオブジェクト my_lock は、オブジェクト・スコープを離れるときに呼び出されるオブジェクト・デストラクター内で自動的にロックを解除（または解放）します。他の待機中のスレッドにできるだけ早く順番が回るように、保護する領域を中括弧 {} で囲んで、ロックのスコープを可能な限り短くすることを推奨します。

注: 図 8-8 のコードでロック・オブジェクトに名前を付け忘れた場合、

```
// my_lock{my_mutex}
my_mutex_t::scoped_lock {my_mutex};
```

コードは警告なしにコンパイルされますが、scoped_lock のスコープはセミコロンで終了します。オブジェクト名（my_lock）がない場合、scoped_lock クラスの匿名/名前のないオブジェクトが構築され、名前付きオブジェクトは定義より長く存続しないため、その有効期間はセミコロンで終了します。これは有用ではなく、クリティカル・セクションは相互排他によって保護されません。

図 8-8 のコードをより明示的に記述する（*ただし推奨されません*）代替方法を図 8-9 に示します。

```
parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const  tbb::blocked_range<size_t>& r)
              {
                my_mutex_t::scoped_lock my_lock;
                my_lock.acquire(my_mutex);
                for  (size_t  i = r.begin(); i < r.end(); ++i)
                   hist_p[image[i]]++;
                my_lock.release();
              });
```

図 8-9. ミューテックスのロックを取得する非推奨の代替手段（図 8-8 のように実行するのが最善）。サンプルコード synchronization/histogram_03_1st_safe_parallel.cpp

C++ の専門家は、図 8-8 の「リソース取得は初期化」（RAII）として知られている代替案を好みます。これは、ロックがオブジェクトの有効期間に結び付けられているため、ロックの解除を覚えておく必要がないためです。さらに重要なことは、RAII バージョンを使用すると、例外が発生した場合にロックが解放されるロック・オブジェクトのデストラクターも呼び出されるため、例外によってロックが取得されたままになるのを防ぐことができることです。図 8-9 のバージョンでは、`my_lock.release()` メンバー関数が呼び出される前に例外がスローされた場合でも、デストラクターが呼び出され、そこでロックが解放されます。ロックがスコープを離れても、それ以前に `release()` メンバー関数によって解放されていると、デストラクターは何も行いません。

図 8-8 のコードを見直すと、「並列コードを粗粒度のロックでシリアル化していませんか？」と思うかもしれません。はい、その通りです。図 8-10 から分かるように、イメージのチャンクを処理する各スレッドは、最初にミューテックスのロックを取得しようとしますが、成功するのは 1 つだけで、残りのスレッドはロックが解放されるのを待ち続けることになります。ロックを保持しているスレッドがロックを解除するまで、ほかのスレッドは保護されたコードを実行できません。したがって、`parallel_for` はシリアルに実行されることになります。良いニュースとしては、ヒストグラム・ビンの同時インクリメントがなくなり、結果が最終的に正しくなったことです。

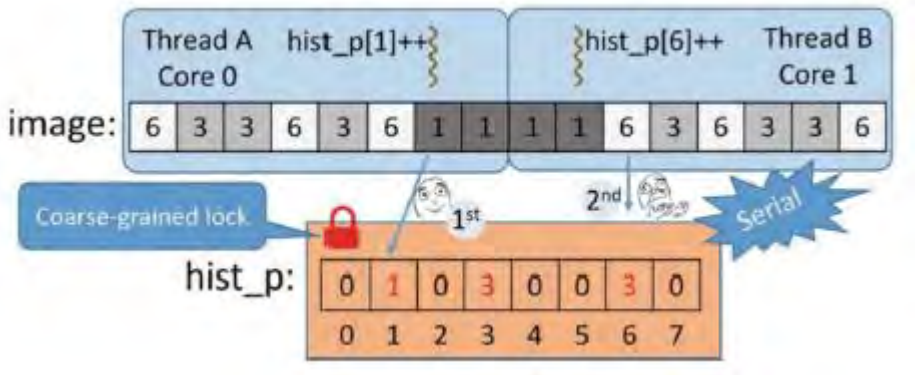


図 8-10. スレッド A はビン番号 1 をインクリメントするのに粗粒度のロックを保持し、スレッド B はヒストグラム・ベクトル全体がロックされているため待機します

新しいバージョンをコンパイルして実行すると、並列実行はシーケンシャル実行よりも遅くなりました:

```
./histogram_03_1st_safe_parallel
Serial: 0.274083, Parallel: 0.488502, Speed-up: 0.561068
```

このアプローチは、粗粒度のデータ構造（この場合は実際にはデータ構造全体、つまりヒストグラム・ベクトル）を保護するため、粗粒度ロックと呼ばれます。ヒストグラム・ベクトルを複数のセクションに分割し、各セクションを個別のロックで保護することができます。これにより、同時実行レベルは上がります（異なるセクションにアクセスする異なるスレッドが並行して処理を進めることができます）が、コードが複雑になり各ミューテックス・オブジェクトに必要なメモリーも増加します。

注意すべきことがあります。図 8-11 は並列プログラミングの初心者が陥る、よくある間違いを示しています。

```
  // my_mutex_t my_mutex; ここではありません! ボディーに移動!
  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
    [&](const  tbb::blocked_range<size_t>& r)
    {
      my_mutex_t  my_mutex;
      my_mutex_t::scoped_lock my_lock{my_mutex};
      for  (size_t  i = r.begin(); i < r.end(); ++i)
        hist_p[image[i]]++;
    });
```

図 8-11. 並列プログラミングの初心者が犯しがちな間違い

このコードはエラーや警告なしでコンパイルされますが、何が問題なのでしょうか？ 試着室の例に戻ると、私たちの目的は、複数の人が同時に試着室に入るのを避けることでした。図 8-11 では、並列セクション内で my_mutex が定義されており、タスクごとにミューテックス・オブジェクトが存在し、それぞれが独自のミューテックスをロックするため、クリティカル・セクションへの同時アクセスは妨げられません。図 8-12 から分かるように、初心者向けのコードでは、基本的に同じ試着室に入る各人用の別々のドアが用意されています。これは私たちが望んでいることではありません! 解決策は、すべての人が同じドアを通って試着室に入るように、my_mutex を 1 回宣言することです（図 8-8 で行ったように）。

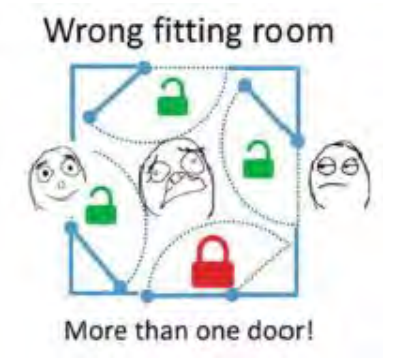


図 8-12. 複数のドアがある試着室

細粒度のロックの代替手段に取り組む前に、知っておくべき価値のある 2 つの側面について説明しましょう。まず、図 8-8 の「並列化してからシリアル化」するコードの実行時間は、シリアル実装の時間よりも長くなります。これは、「並列化してからシリアル化」するオーバーヘッドによるものですが、キャッシュの利用効率が低いことも原因です。もちろん、偽の共有も真の共有もありません。シリアル化された実装では「共有」が全く行われないためです。それともあるのでしょうか？ シリアル実装では、キャッシュされたヒストグラム・ベクトルにアクセスするスレッドは 1 つだけです。粗粒度の実装では、1 つのスレッドが画像のチャンクを処理する際に、そのスレッドが実行されているコアのキャッシュにヒストグラムが読み込まれます。キュー内の次のスレッドが最終的に独自のチャンクを処理できるようになると、ヒストグラムを別のキャッシュに取り込む必要があります（スレッドが別のコアで実行されている場合）。スレッドは依然としてヒストグラム・ベクトルを共有しており、提案された実装ではシリアル実装よりもキャッシュミスが多く発生する可能性が高くなります。

2 番目に言及したいことは、図 8-13 に示すミューテックスの種類の 1 つを選択して、ミューテックスの動作を構成できる点です。したがって、

```
using  my_mutex_t  = <mutex_flavor>
```

を使用し、その後は my_mutex_t を使用することをお勧めします。これにより、単一のプログラム行でミューテックスの種類を簡単に変更し、どの種類が最適であるか評価できます。図 8-13 に示すように、別のヘッダーファイルもインクルードするか、すべてを含んだ tbb.h を使用することもあります。

Mutex flavor	Scalable	Fair	Recursive	Long Wait	Size
tbb::mutex <tbb/mutex.h>	✓	✗	✗	blocks	1 byte
tbb::null_mutex <tbb/null_mutex.h>	✓	✓	✓	never	empty
tbb::rw_mutex <tbb/rw_mutex.h>	✓	✗	✗	blocks	1 word
tbb::null_rw_mutex <tbb/null_rw_mutex.h>	✓	✓	✓	never	empty
tbb::queuing_mutex <tbb/queuing_mutex.h>	✓	✓	✗	yields	1 word
tbb::queuing_rw_mutex <tbb/queuing_rw_mutex.h>	✓	✓	✗	yields	1 word
tbb::speculative_spin_mutex <tbb/spin_mutex.h>	HW depend. (yes if H/W support, no if not)	✗	✗	yields	2 cache lines
tbb::speculative_spin_rw_mutex <tbb/spin_rw_mutex.h>	HW depend.	✗	✗	yields	3 cache lines
tbb::spin_mutex <tbb/spin_mutex.h>	✗	✗	✗	yields	1 byte
tbb::spin_rw_mutex – if using try_lock() <tbb/spin_rw_mutex.h>	✗	✗	✗	yields	1 word
tbb::spin_rw_mutex – if using lock() <tbb/spin_rw_mutex.h>	✓	✗	✗	blocks	1 word
std::mutex <mutex>	✗	✗	✗	blocks	≥ 3 words
std::recursive_mutex <mutex>	✗	✗	✓	blocks	≥ 3 words
std::recursive_timed_mutex <mutex>	✗	✗	✓	blocks	≥ 3 words
std::shared_mutex <shared_mutex>	✗	✗	✗	blocks	≥ 3 words
std::shared_timed_mutex <shared_mutex>	✗	✗	✗	blocks	≥ 3 words
std::timed_mutex <mutex>	✗	✗	✗	blocks	≥ 3 words

図 8-13. さまざまなミューテックスの種類とそのプロパティー: `tbb::` から 7 つ、`std::` から 6 つ。`std::` の場合、いくつかの実装は示されるよりも良い結果となる可能性がありますが、C++標準では保証されていません

ミューテックスの種類

さまざまなミューテックスの種類を理解するには、まずそれらを分類するプロパティーについて説明する必要があります。

1. **スケーラブル**なミューテックスは、順番を待つ間にコアサイクルやメモリー帯域幅を過度に消費しません。目的は、待機中のスレッドが、待機していない他のスレッドが必要なハードウェア・リソースの消費を回避することです。

2. **公平な**ミューテックスは、スレッドが順番を FIFO ポリシーを使用して管理します。ミューテックスの「公平」という用語は、最も長く待機したスレッドにロックを優先的に与えることを意味します。ロックが不公平な場合、スレッドが実行されなくなる可能性があります（他のスレッドがロックを要求している限り、ロックが与えられない）。

3. **再帰**ミューテックスは、すでにロックを保持しているスレッドが別のミューテックスのロックを取得することを許可します。ミューテックスを避けるためコードを再考するのは素晴らしいことですが、再帰ミューテックスを避けるためコードを再考するのはほぼ必須です。では、なぜ標準 C++ ライブラリーがそれらを提供しているのでしょうか？ それは、再帰ミューテックスが避けられない特殊なケースが存在する可能性があるためです。困難であったり、効率的な解決策を探す時間がない場合にも役立つかもしれません。
   注: オリジナルの TBB には、標準 C++ ライブラリーと同じ理由、つまり利便性のために再帰ミューテックスがありました。コード内に再帰ミューテックスを残す稀なケースでは、標準 C++ ライブラリーが使用できるようになったため、最新の C++ に更新したときに TBB から削除されました。

図 8-13 の表には、ミューテックス・オブジェクトのサイズと、ミューテックスのロックを取得するのに長時間待機するスレッドの動作も示されています。最後の点に関して、スレッドが順番を待っているときは、ビジー待機、ブロック、または譲渡することができます。ブロックされたスレッドはブロック状態に変更され、必要なリソースはスリープ状態を維持するメモリーのみになります。スレッドが最終的にロックを取得できると、スレッドは起動して準備完了状態に戻り、準備完了状態のスレッドは次の順番を待ちます。OS スケジューラーは、準備完了状態のキューで待機している準備完了スレッドにタイムスライスを割り当てます。ロック取得の順番を待っている間に譲渡するスレッドは、準備完了状態のままになります。

スレッドが準備完了状態キューの先頭に到達すると、実行のためディスパッチされますが、ミューテックスがまだ他のスレッドによってロックされている場合は、再びタイムスライスを放棄し（他に何もすることがない）、準備完了状態キューに戻ります。

注

この過程では、次の 2 つのキューが関係することに注意してください:

(i) オペレーティング・システムのスケジューラーによって管理される準備完了状態キュー。準備完了スレッドは、必ずしも FIFO 順ではなく、アイドルコアにディスパッチされ実行スレッドになるのを待機しています。

(ii) オペレーティング・システムまたはユーザー空間のミューテックス・ライブラリーによって管理されるミューテックス・キュー。スレッドは、キューイング・ミューテックスのロックを取得する順番を待機しています。

コアがオーバーサブスクライブされていない場合（このコアで実行されているスレッドが 1 つだけの場合）、ミューテックスがまだロックされているため譲渡したスレッドは、準備完了状態のキューにある唯一のスレッドとなり、すぐにディスパッチされます。この場合、譲渡メカニズムは実質的にビジー待機と同等になります。

ミューテックスの実装を特徴付けるさまざまなプロパティーを説明したので、次は `tbb::` と `std::` が提供する特定のミューテックスの種類について詳しく見ていきましょう。

ミューテックスは、最新の C++ と TBB によって提供されます。TBB バージョンの利点は、図 8-13 の表から明らかです。長時間待機するとミューテックスはブロックされ、ミューテックスが解放されたときに応答時間が長くなる可能性があります。

一方、`tbb::spin_mutex` はスレッドをブロックしません。ミューテックスのロック保持を待機しながら、ユーザー空間でビジー待機状態になります。待機中のスレッドは、ループの取得を何回か試行した後に譲渡しますが、コアがオーバーサブスクライブでない場合、このスレッドはコアのサイクルと電力を浪費し続けます。一方、ミューテックスが解放されると、それを取得する応答時間は最短になります（起動して実行のためディスパッチを待つ必要はありません）。このミューテックスは公平でないため、スレッドがどれだけ長く順番を待っていたとしても、ミューテックスのロックが解除されていることを最初に発見したスレッドが追い越してロックを取得できます。この場合、フリーフォーオール（ルールがない状態）が起こり、極端な状況では、弱いスレッドが飢餓状態に陥り、ロックを取得できなくなる可能性があります。それでも、競合が少ない状況では最も高速になる可能性があるため、これが推奨されるミューテックスの種類です。

`tbb::queueing_mutex` は、`spin_mutex` のスケーラブルで公平なバージョンです。ユーザー空間でビジー待機しながらスピンし続けます。ただし、そのミューテックスを待機しているスレッドは FIFO 順にロックを取得するため、飢餓状態にはなりません。

`tbb::speculative_spin_mutex` は、一部のプロセッサーでサポートされるハードウェア・トランザクショナル・メモリー（HTM）上に構築されます。HTM のないシステムでは、単純に `spin_mutex` として動作します。HTM の哲学は楽観的です。HTM では、共有メモリーの競合が発生しないことを前提とし、すべてのスレッドが同時にクリティカル・セクションに入ることができます。しかし、競合したらどうなるでしょうか？ この場合、ハードウェアは競合を検出し、競合しているスレッドの 1 つの実行をロールバックし、クリティカル・セクションの実行を再試行します。図 8-8 に示す粗粒度の実装では、次の行を追加できます。

```
using my_mutex_t = tbb::speculative_spin_mutex;
```

すると、イメージを走査する `parallel_for` が再び並列化されます。現在、すべてのスレッドがクリティカル・セクションに入ることができます（画像の特定チャンクのヒストグラム・ビンを更新するため）。ただし、ビンの 1 つを更新する際に競合が発生した場合のみ、競合するスレッドの 1 つが実行を再試行します。これを効率良く動作させるには、保護されたクリティカル・セクションが十分に小さく、競合や再試行がほとんど発生しない必要がありますが、図 8-8 のコードではそれが当てはまりません。

`tbb::spin_rw_mutex`、`tbb::queueing_rw_mutex`、および `tbb::speculative_spin_rw_ mutex` は、これまで説明した種類のリーダー/ライター・ミューテックスに対応します。この実装により、複数のリーダーが共有変数を同時に読み取ることができます。ロック・オブジェクト・コンストラクターには 2 番目の引数（ブール値）があり、クリティカル・セクション内で読み取りのみ（書き込みなし）を行う場合は `false` に設定します:

```
using  my_mutex_t=tbb::spin_mutex;
rwmutex_t  my_mutex;
{
   rwmutex_t::scoped_lock my_lock{my_mutex,  /*is_writer =*/false};
   // リーダーロックが取得されるため、
   // 複数の同時読み取りが許可されます
}
```

何らかの理由でリーダーロックをライターロックに昇格する必要がある場合、TBB は次のような `upgrade_to_writer()` メンバー関数を提供します。

```
bool  success=my_lock.upgrade_to_writer();
```

これは、`my_lock` がロックを解放せずにライターロックに正常にアップグレードされると `true` を返し、それ以外は `false` を返します。

最後に、何もしないダミー・オブジェクトである `null_mutex` と `null_rw_mutex` があります。それは一体何を意味しているのでしょうか。実際のミューテックスが必要であるか

不明な関数テンプレートにミューテックス・オブジェクトを渡す場合、このミューテックスは便利です。関数が実際にミューテックスを必要としない場合は、ダミーを渡すだけです。

スケーラビリティーが最も重要である場合

*スケーラブル*という用語は、多数のロックの競合がある場合にパフォーマンスが良好になる実装に使用されます。TBB ミューテックスのこの特性は、一部の x86 実装で利用可能なスケーラブルなハードウェア・メカニズムに依存する `speculative_spin_mutex` を除くすべてのミューテックスの実装（アルゴリズム）で実現されます。

競合が軽量でない、またはクリティカル・セクション（ロックが保持される期間）がそれほど短くない場合は、スケーラブルなロックが必要です。
つまり、次の 2 つの条件が満たされない限り、スケーラブルでないロックを避ける必要があります:
(1) ロックの競合が軽量である。
(2) クリティカル・セクションが非常に高速である。
スケーラブルでないロックを選択し、これら 2 つの条件が満たされないと、アプリケーションのスケーラビリティーが大幅に低下する可能性があります。

ただし、スケーラブルなミューテックスを使用しても、アプリケーション自体が魔法のようにスケールされるわけではないことを忘れてはなりません。ロックが大量に存在する場合、コード内にシリアル化のボトルネックが発生します。スケーラブルなミューテックス自体は、スケーラブルでない代替手段と比較して、そのような状況でも適切に機能し、競合が発生した場合のオーバーヘッドが可能な限り少なくなります。しかし、アプリケーションのアルゴリズムが有用な並列性をもたらすとは限りません。前述したように、`tbb::spin_mutex` はアクティブにスピンしますが、すぐにロックを取得できない場合は譲渡します。スレッドを譲渡すると、他のスレッドが処理を続行する機会が与えられ、競合が減少しますが、スレッドは必要なロックを取得できず、順番を待つ状態は変わりません。スケーラブルでないミューテックスを使用すると、この状況はさらに悪化するだけです。

2 番目の安全な並列実装: 細粒度のロック

ミューテックスの種類について理解できたので、図 8-8 の粗粒度ロックの代替実装について考えてみましょう。代替の 1 つは、ヒストグラムの各ビンに対してミューテックスを宣言し、データ構造全体をロックするのではなく、実際にインクリメントしているメモリー位置のみを保護することです。

それには、図 8-14 に示すようなミューテックスのベクトル fine_m が必要です。

```
  using  my_mutex_t=tbb::spin_mutex;
  std::vector<my_mutex_t> fine_m(num_bins);
  std::vector<int> hist_p(num_bins);
  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
               [&](const  tbb::blocked_range<size_t>& r)
               {
                 for  (size_t  i = r.begin(); i < r.end(); ++i){
                   int  tone=image[i];
                   my_mutex_t::scoped_lock my_lock{fine_m[tone]};
                   hist_p[tone]++;
                 }
               });
```

図 8-14. 細粒度のロックを使用する画像ヒストグラム計算の 2 番目の安全な並列実装のコード。サンプルコード: synchronization/histogram_04_2nd_safe_parallel.cpp

parallel_for で使用されているラムダは、スレッドがビン hist_p[tone] をインクリメントする必要がある場合、fine_m[tone] のロックを取得し、他のスレッドが同じビンにアクセスするのを防ぎます。「他のビンは更新できますが、この特定のビンは更新できません。」これは図 8-15 に示されており、スレッド A と B はヒストグラム・ベクトルの異なるビンを並列に更新しています。

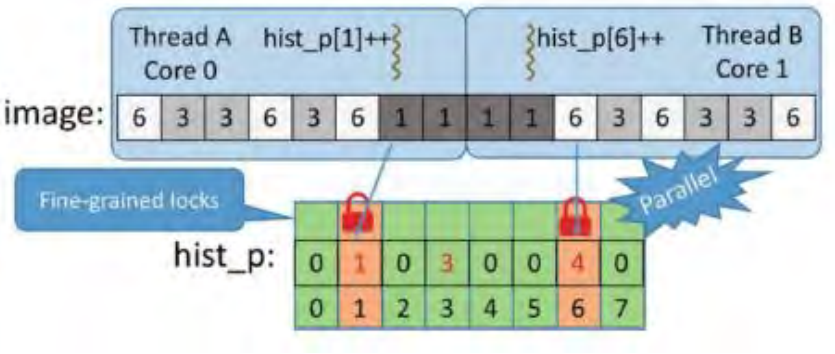


図 8-15. 細粒度のロックにより、より多くの並列処理を活用できます。しかし、後で分かりますが、これはこれまでで最もパフォーマンスの悪い例です。議論し学ぶべきことはまだあります

ただし、パフォーマンスの観点から見ると、この代替案は最適なものではありません（実際には、これまでのところ最も低速な代替案です）:

```
./histogram_04_2nd_safe_parallel
Serial: 0.266333, Parallel: 108.981, Speed-up: 0.00244386
```

ここでは、ヒストグラム配列だけでなく、同じ長さのミューテックス・オブジェクトの配列も必要になります。これは、より大きなメモリー要件と、キャッシュされるデータの増加、および偽の共有と真の共有の影響を受けるデータの増加を意味します。失敗です。

コンボイとデッドロック

ロック固有のオーバーヘッドに加えて、ロックはコンボイとデッドロックという 2 つの問題の原因となります。まず「コンボイ」について説明しましょう。この名前は、すべてのスレッドが最初のスレッドよりも遅い速度で次々に移動するイメージから来ています。この状況を分かりやすく説明するには、図 8-16 のような例が必要です。スレッド 1、2、3、4 が同じコアで同一のコードを実行していて、スピン・ミューテックス A で保護されるクリティカル・セクションがあるとします。これらのスレッドが異なるタイミングでロックを保持する場合、競合なく正常に実行されます（状況 1）。しかし、スレッド 1 がロックを解放する前にタイムスライスを使い果たし、A が準備完了状態キューの最後に送られる可能性があります（状況 2）。

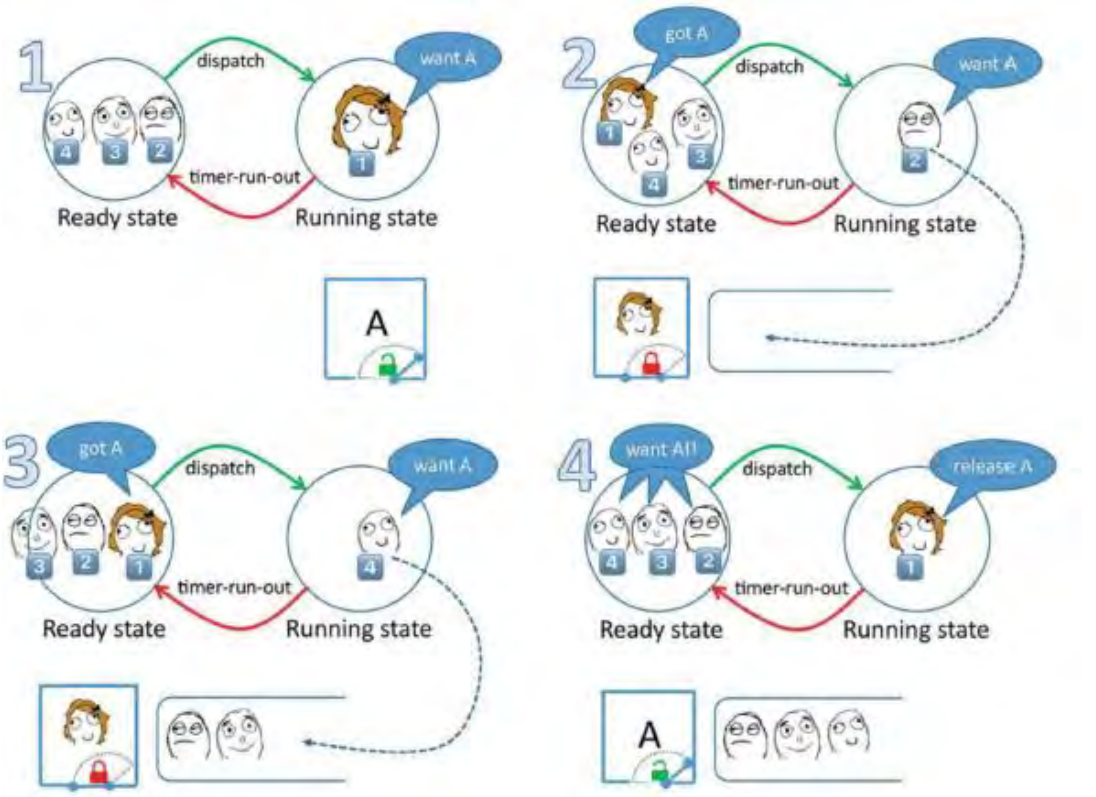
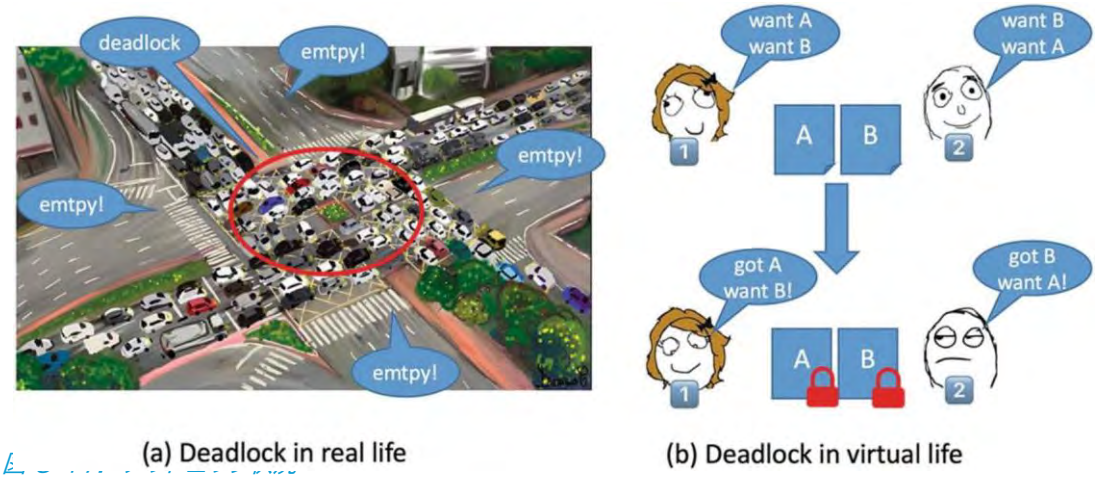


図 8-16. オーバーサブスクリプションのコンボイ(1 つのコアで 4 つのスレッドが実行され、それらすべてが同じミューテックス A を必要とする)

スレッド 2、3、4 はそれぞれタイムスライスを取得しますが、スレッド 1 がまだロックの所有者であるため、取得できません (状況 3)。つまり、2、3、4 は譲渡したり、スピンできるようになりましたが、いずれにせよ、1 速ギアで大型トラックの後ろに留まっています。1 が再度ディスパッチされると、ロック A が解除されます (状況 4)。現在、2、3、4 はロックをめぐって競合する態勢が整っていますが、成功するのは 1 人だけで、残りは再び待機状態となります。この状況は、特にスレッド 2、3、4 が保護されたクリティカル・セクションを実行するため 1 回以上タイムスライスを必要とする場合に、繰り返し発生します。さらに、スレッド 2、3、4 が意図せず調整され、同じコード領域が同時に実行されると、ミューテックス競合の可能性が高くなります。コアがオーバーサブスクリプション状態である場合 (この例では 4 つのスレッドが 1 つのコアで実行するために競合しています)、コンボイは特に深刻になることに注意してください。これも、オーバーサブスクリプションを回避する推奨を補強するものです。

ロックによって発生するもう 1 つの問題は、「デッドロック」です。図 8-17(a) は、利用可能なリソース (どの車も通行していない車線) があるにもかかわらず、誰も前進できない最悪の状況を示しています。これは現実の世界では行き詰まりですが、このイメージを払拭して (できれば!)、並列プログラミングの仮想世界に戻ってください。

ロックを保持し、セット内の他のスレッドがすでに保持しているロックの取得を待機している N 個のスレッドのセットがある場合、N 個のスレッドはデッドロックになります。図 8-17(b) に、スレッドが 2 つだけの例を示します。スレッド 1 は、ミューテックス A のロックを保持しており、ミューテックス B のロックの取得を待機していますが、スレッド 2 もまたミューテックス B のロックを保持しており、ミューテックス A のロックの取得を待機しています。明らかに、どのスレッドも処理を進めることはできず、永遠に致命的な状況に陥る運命にあります。スレッドがすでにミューテックスを保持している場合に別のミューテックスの取得を要求しないようにすることで、この不幸な状況を回避できます。または、少なくとも、すべてのスレッドが常に同じ順序でロックを取得することで回避できます。



(a) Deadlock in real life　　(b) Deadlock in virtual life

すでにロックを保持しているスレッドが、別のロックを取得する関数を呼び出すと、意図せずデッドロックが発生する可能性があります。関数が何を行うか不明である場合、ロックを保持している間は、その関数を呼び出さないようにします（通常は、ロックを保持したまま、他者が作成したコードを呼び出さないようにアドバイスされます）。あるいは、後続の関数呼び出しチェーンがデッドロックを引き起こさないことを確認する必要があります。また、可能な限りロックを避けるべきです。

C++11 以降、関数 `std::lock` を使用して、複数のロック可能オブジェクト（https://en.cppreference.com/w/cpp/named_req/Lockable）をロックするときにデッドロックを回避できます。`std::lock` は、デッドロック回避アルゴリズムを適用し、`lock`、`try_lock`、`unlock` の一連の呼び出しを使用して、渡されたオブジェクトのセットがデッドロックなしでロックされるようにします。

std::lock の詳細については、
https://en.cppreference.com/w/cpp/thread/lock を参照してください。この関数は標準のミューテックス型だけでなく、ロック可能な名前付き要件をモデル化する任意の型でも機能します。tbb::mutex、tbb::spin_mutex、tbb::rw_mutex、および tbb::spin_rw_mutex は、ISO C++ 標準の [thread.mutex.requirements] セクションの要件を満たしており、*ロック可能な*オブジェクトです。そのため、std::lock や、ロック可能なオブジェクトまたは C++ ミューテックスで動作する他の C++ 関数およびクラスで使用できます。

コンボイとデッドロックは、実際にはヒストグラムの実装には影響しませんが、ロックは解決するよりも多くの問題を引き起こすことが多く、高い並列パフォーマンスを得る最善の代替手段ではないことを確認するのに役立つはずです。競合の可能性が低く、クリティカル・セクションの実行時間が最小である場合にのみ、ロックは許容できる選択肢です。この場合、基本的な spin_lock または speculative_spin_lock を使用すると、ある程度のスピードアップが得られます。しかし、それ以外は、ロックベースのアルゴリズムのスケーラビリティーは損なわれるため、最善のアドバイスは、既成概念にとらわれずに、ミューテックスを全く必要としない新しい実装を考案することです。しかし、複数のミューテックス・オブジェクトに依存せず細粒度の同期を実現し、オーバーヘッドや潜在的な問題を回避することはできるでしょうか？

3 番目の安全な並列実装: アトミック

幸いなことに、多くの場合、ミューテックスとロックを排除できる、より安価なメカニズムが存在します。アトミック変数を使用してアトミック操作を実行できます。図 8-6 に示されているように、インクリメント操作はアトミックではありませんが、3 つの小さな操作（ロード、インクリメント、および保存）に分割できます。アトミック変数を宣言して次の操作を行うことができます:

```
#include  <atomic>
std::atomic<int> counter;
counter++;
```

注: アトミックについては、C++ std::atomic に依存できます（https://en.cppreference.com/w/cpp/atomic を参照）。オリジナルの TBB では、C++ でアトミック操作が定義されていなかったため、アトミック操作を提供する必要がありました。現在の TBB は有益な高性能ミューテックスを提供していますが、アトミックに関しては標準の C++ に依存しています。すべてを TBB に依存していた古いプログラムを現在の TBB と C++ に移行する詳細とアドバイスについては、12 章を参照してください。

ここで示したケースでは、アトミック変数のインクリメントはアトミック操作です。つまり、カウンターの値にアクセスする他のスレッドは、インクリメントが 1 ステップで実行されたように操作を「認識」します（3 つの小さな操作ではなく、1 つの操作として）。つまり、他の「目ざとい」スレッドは、操作が完了したかどうかは監視しますが、インクリメントの過程は監視しません。

アトミック操作は、コンボイやデッドロック [2] の影響を受けず、相互排除の代替手段よりも高速です。ただし、すべての操作がアトミックに実行できるわけではなく、また、実行できる操作がすべてのデータ型に適用できるわけではありません。`std::atomic` テンプレートは、*`TriviallyCopiable`* であり、*`CopyConstructible`* と *`CopyAssignable`* の両方で、任意の型に対してインスタンス化できます（これらの名前付き要件を正確に確認するには、cppreference の `std::atomic` を参照してください）。幸いなことに、`std::atomic<T>` は、`T` が整数、列挙、またはポインターデータ型である場合を含め、最も一般的に使用される型の多くでアトミック操作をサポートしています。`std::atomic<T>` のような型の変数 `x` でサポートされる最も重要なアトミック操作を図 8-18 に示します。

[2] アトミック操作はネストできないため、デッドロックを引き起こすことはありません。

=x x.load()	read the value of x
x=	write the value of x, and return it
x.store(y)	do x=y *(no result – is void)*
x.fetch_add(y) x++	do x+=y and return the old value of x do x+=1 and return the old value of x
x+=y x.fetch_sub(y)	do x+=y and return the old value of x do x−=y and return the old value of x
x-- x-=y	do x−=1 and return the old value of x do x−=y and return the old value of x
x.fetch_and(y) x&=y	do x&=y and return the old value of x
x.fetch_or(y) x|=y	do x|=y and return the old value of x
x.fetch_xor(y) x^=y	do x^=y and return the old value of x
x.compare_exchange_weak(z,y) x.compare_exchange_strong(z,y)	if x equals z, then do x=y. Result (bool) indicates if swap occurred. Strong version is preferred outside loops, and weak is used in loops.

図 8-18. アトミック変数の基本的な操作。さらに、C++ではいくつかの操作のパラメーターとして `std::memory_order` が提供されています

アトミックに基づくもう 1 つの便利な慣用句は、図 2-16（2 章）に示されている例ですでに使用されているものです。アトミック整数 `refCount` を “y” に初期化し、複数のスレッドでこのコードを実行すると、

```
if (--refCount == 0) { .../* ボディー */  ...};
```

は、前の行を実行している y 番目のスレッドのみが「ボディー」に入ることになります。

これらの基本操作のうち、`compare_exchange_strong`（比較交換）は、すべてのアトミック読み取り - 変更 - 書き込み（RMW）操作のベースと考えることができます。これは、すべてのアトミック RMW 操作を比較交換操作上に実装できるためです。表には、比較交換の 2 つのバリエーション、`compare_exchange_strong` と `compare_exchange_weak` がリストされています。違いは、`compare_exchange_weak` は、「見せかけの失敗」が許可されていることです。つまり、x が実際に z に等しい場合でも、`false` を返して交換しない可能性があります。`weak` バリエーションが存在するのは、一部のシステムではこの実装の方が効率的である可能性があるためです。`weak` バリエーションを使用する場合、見せかけの失敗を考慮してコードを記述する必要があります。

アトミックと事前発生

TBB の構成要素は、並列実行する必要がないことを許容しながら、並列実行が可能であることを示唆するために存在していることを知ることは重要です。これを緩和されたシーケンシャル・セマンティクスと呼びます。 これは、同期の考え方に影響します。TBB アルゴリズムによって生成された 2 つのタスクが、必ず同時に実行されると想定してはなりません。TBB は、すべてのタスクを実行するため、単一のスレッドを使用する可能性もあります。例えば、アトミック変数を繰り返しチェックして、1 つの TBB タスクが別の TBB タスクの結果を待つことは避けるべきです。このパターンは、タスクを実行する複数のスレッドが割り当てられている場合は機能する可能性がありますが、1 つのスレッドのみがすべてのタスクを実行する場合はデッドロックが発生します。通常、アトミックは TBB タスク間の同期操作として使用すべきではありません。

アトミック性に関しては、単一のアトミック操作はアトミックですが、連続する 2 つのアトミック操作は組み合わせてアトミックではなく、個別にのみアトミックであることを覚えておく必要があります。これはよくある誤解であり、一連のアトミック操作が全体としてアトミックに動作すると誤って想定することはよくあります。

しかし、すべてが失われたわけではありません。C++ では、特定の順序を保証するため、一部のアトミック操作に `std::memory_order` パラメーターが用意されています。これらのパラメーターは、アトミックではないメモリーアクセスを含むメモリーアクセスがアトミック操作の周囲でどのように順番付けされるか決定します。C++ 標準ではメモリー順序について詳細に規定されており、cppreference（https://en.cppreference.com/w/cpp/atomic/memory_order）にこのトピックの詳細な説明が記載されています。 したがって、ここではその話題に触れるだけに止めます。

`std::atomic` 操作のデフォルトは `std::memory_order_seq_cst` です。この順序付けは、`std::memory_order_seq_cst` としてマークされたアトミックストアの前に、スレッドで実行されるすべてのメモリー操作が、同じ変数からアトミック `std::memory_order_seq_cst` ロードを実行する別のスレッドに可視であることを意味します。さらに、`std::memory_order_seq_cst` としてタグ付けされたすべてのアトミック操作の変更順序は、すべてのスレッドで合計 1 つです。これは、すべてのスレッドがアトミック変数の変更を同じ順序で認識することを意味します。これは適切で安全なデフォルトです。

しかし、柔軟性が向上し、潜在的にパフォーマンスが向上する可能性もあるため、別の順序も存在します。一例として、解放（`release`）-取得（`acquire`）の順序付けが挙げられます。最初のスレッドが、`memory_order_release` を使用してアトミック変数に値をストアし、2 番目のスレッドが、`memory_order_acquire` を使用して同じアトミック変数からその値をロードする場合、最初のスレッドでのストアは、2 番目のスレッドのロードと同期されます。アトミックストアの前に最初のスレッドで書き込まれたメモリーは、アトミックロードが完了すると 2 番目のスレッドで認識されることが保証されます。

これは std::memory_order_seq_cst に類似していますが、どれを取得し、どれを解放するかを覚えておく必要がある点が異なります。ただし、すべてのアトミック操作にまたがる全体的な変更が作成されるわけではないため、スレッドは異なる順序で異なるアトミック変数の更新を認識する場合があります。繰り返しますが、cppreference では発生する可能性のある複雑な問題について詳しく説明しているので、興味のある読者はそちらを参照してください。

TBB は、特に*緩和されたシーケンシャル・セマンティクス*・モデル、つまりオプションの並列処理を通じて、スケーラブルな手法を使用するようにガイドすることで、高パフォーマンスに重点を置いています。ある TBB タスクによって実行されたメモリー操作が、特定の順序で別の TBB タスクで認識されるようにしたい場合、アトミックとミューテックスを使用します。柔軟性を求める場合、デフォルトの std::memory_order_seq_cst よりも緩やかな順序付けを検討できます。

小さなクリティカル・セクションを考慮する際の注意点

小さなクリティカル・セクションを保護する必要があり、ロックを回避する必要があることを認識している場合は、比較交換操作の詳細を少し見てみましょう。コードでは、共有整数変数 v を 3 でアトミック乗算する必要があるとします。ロックフリーのソリューションを目指していますが、乗算はアトミック操作としてサポートされないことは分かっているので、ここで比較交換が登場します。まず、v をアトミック変数として宣言します。

```
std::atomic<uint32_t> v;
```

これで、v.compare_exchange_weak(old_v, new_v) を呼び出すことができ、これはアトミックに

```
ov=v; if  (ov == old_v) v=new_v; return  ov;
```

つまり、v が old_v と等しい場合にのみ、v を新しい値で更新できます。いずれにせよ、ov (“==” 比較で使用される共有 v) を返します。ここで、「3 倍」のアトミック乗算を実装するには、比較交換ループをコード化することです。

```
void  fetch_and_triple(std::atomic<uint32_t>& v)
{
  uint32_t  old_v;
  do  {
    old_v=v;  //スナップショットを取得
  } while  (!v.compare_exchange_weak(old_v, old_v * 3));
}
```

新しい `fetch_and_triple` は、同じ共有アトミック変数を渡して呼び出されても、スレッドセーフです（複数のスレッドから同時に安全に呼び出すことができます）。この関数は基本的に、最初に共有変数のスナップショットを取得する `do-while` ループです（これは、後で他のスレッドが変更できたか比較する鍵となります）。次に、アトミックに、他のスレッドが `v` を変更していない場合は `v (v==old_v))`、それを更新して `((v=old_v*3))`、`v` を返します。この場合、`v == old_v`（この場合も、他のスレッドが `v` を変更していない）のため、`do-while` ループを終了し、共有 `v` が正常に更新され関数から戻ります。

ただし、スナップショットを取得した後に別のスレッドが `v` を更新する可能性があります。この場合、`v!=old_v` となり、(i) `v` を更新せず、(ii) 次回は順番が来ることを期待して `do-while` ループに留まります（スナップショットを取得してから `v` の更新に成功するまでの間に、他のスレッドが `v` に触れる可能性がない場合）。図 8-19 は、`v` が常にスレッド 1 またはスレッド 2 によって更新される様子を示しています。スレッドの 1 つが 1 回以上再試行する場合（スレッド 2 が最初は 27 を書き込もうとしていたのに、最終的に 81 を書き込む場合など）がありますが、適切に設計されたシナリオではこれは大きな問題ではありません。

この戦略の 2 つの留意点は、(i) スケーラビリティーが低いことと、(ii)「A-B-A 問題」が発生する可能性があることです（古典的な A-B-A 問題の背景については、3 章で説明しています）。最初の問題に関して、同じアトミックで競合する `p` 個のスレッドを考えてみましょう。`p-1` 回目の再試行で成功するのは 1 つだけで、次に別のスレッドが `p-2` 回目の再試行で成功し、次に `p-3` 回目の再試行で成功するように、結果として 2 次的なワークが発生します。この問題は、競合を減らすのに連続再試行の比率を乗法的に削減する「指数バックオフ」戦略で改善できます。一方、A-B-A 問題は、中間時間（スナップショットを取得してから `v` の更新に成功するまでの間）に別のスレッドが `v` を値 `A` から値 `B` に変更し、再び値 `A` に戻したときに発生します。比較交換ループは介入スレッドに気付かずに成功することがあり、これが問題になる可能性があります。開発において比較交換ループを使用する必要がある場合は、この問題とその結果を再確認してください。

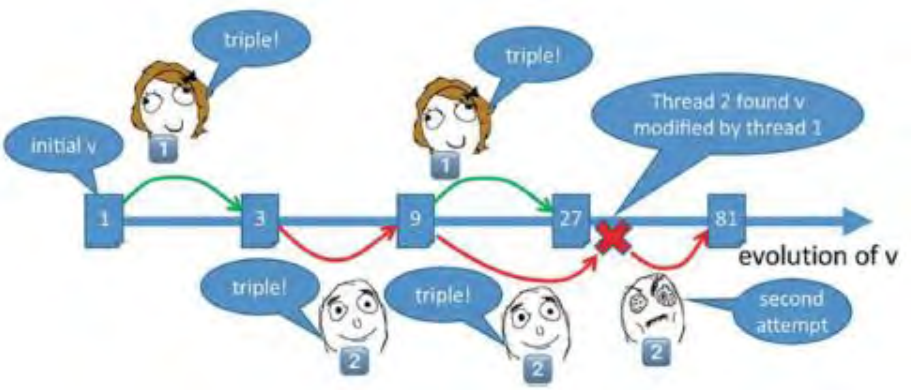


図 8-19. 2 つのスレッドが比較交換ループの上に実装された fetch_and_triple アトミック関数を同時に呼び出します

さて、実行中の例に戻りましょう。ヒストグラム計算の再実装は、図 8-20 に示すように、アトミックを使用することで表現できるようになりました。

```
#include <atomic>

std::vector<std::atomic<int>> hist_p(num_bins);
parallel_for(tbb::blocked_range<size_t>{0, image.size()},
             [&](const tbb::blocked_range<size_t>& r)
             {
               for (size_t i = r.begin(); i < r.end(); ++i)
                  hist_p[image[i]]++;
             });
```

図 8-20. アトミック変数を使用する画像ヒストグラム計算の 3 番目の安全な並列実装のコード。サンプルコード: synchronization/histogram_06_3rd_safe_parallel.cpp

この実装では、ミューテックス・オブジェクトとロックを削除し、各ビンが `std::atomic<int>`（デフォルトでは 0 に初期化）になるようにベクトルを宣言します。次に、ラムダでビンを並列に、かつ安全にインクリメントできます。最終的な結果として、細粒度ロックと同様にヒストグラム・ベクトルの並列インクリメントが可能になりますが、ミューテックス管理とミューテックス・ストレージのコストは低くなります。

しかし、パフォーマンスの面では、この実装はまだ低速です:

```
./histogram_06_3rd_safe_parallel
Serial: 0.282609, Parallel: 10.4025, Speed-up: 0.0271674
```

アトミック・インクリメントのオーバーヘッドに加えて、偽の共有と真の共有の問題にはまだ対処していません。7 章では、整列アロケーターとパディングを活用して、フォルス・シェアリング（偽の共有）に対処しています。フォルス・シェアリングは並列パフォーマンスを妨げる原因として頻繁に発生します。これを回避する推奨手法については 7 章で説明しているので、参照してください。

フォルス・シェアリングの問題が解決されたと仮定すると、真の共有の問題はどうでしょうか？ 2 つの異なるスレッドが最終的に同じビンをインクリメントし、それが 1 つのキャッシュから別のキャッシュにピンポンされることになります。これを解決するにはもっと良いアイデアが必要です!

より優れた並列実装: プライベート化とリダクション

ヒストグラムのリダクションによって生じる真の問題は、すべてのスレッドがインクリメントする 256 個のビンを保持する単一の共有ベクトルが存在することです。これまで、粗粒度、細粒度、アトミックベースのものなど、機能的に同等の実装をいくつか見てきましたが、パフォーマンスやエネルギーなどの機能面以外のメトリックも考慮すると、どれも完全に満足できるものではありません。

何かを共有することを避ける一般的な解決策は、それをプライベート化することです。並列プログラミングもこの点で違いはありません。各スレッドがヒストグラムのプライベート・コピーを保持すると、各スレッドはそのコピーを使用して正常に動作し、スレッドが実行されているコアのプライベート・キャッシュにキャッシュするため、すべてのビンがキャッシュ速度でインクリメントできます（理想的な場合）。ヒストグラム・ベクトルが共有されなくなったため、偽の共有も真の共有もなくなりました。

しかし、各スレッドは画像の一部のピクセルのみを参照しているため、各スレッドはヒストグラムの部分的なビューを取得することになります。これは問題ありません。ここで、この実装のリダクション部分が機能します。ヒストグラムのプライベート化された部分バージョンを計算した後の最後のステップは、すべてのスレッドのプライベートの結果をリダクションして、完全なヒストグラム・ベクトルを取得することです。 一部のスレッドは、ローカル/プライベート計算が完了していない他のスレッドを待機する必要があるため、ここにはまだ同期が残っていますが、一般的なケースでは、この解決策は、これまで説明した他の実装よりはるかに低コストです。図 8-21 は、ヒストグラムの例におけるプライベート化とリダクションの手法を示しています。

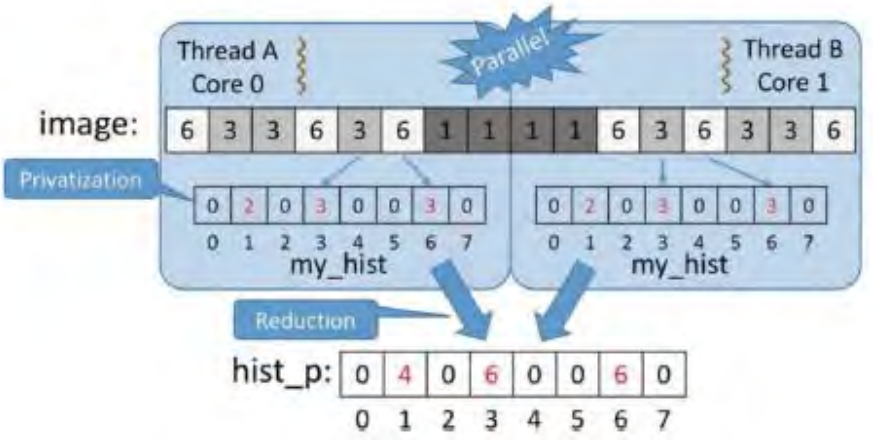


図 8-21. 各スレッドはローカル・ヒストグラム my_hist を計算し、これは後で 2 番目のステップで縮小されます

TBB は、プライベート化とリダクション操作を実行するいくつかの代替手段を提供します。その中には、スレッド・ローカル・ストレージ（TLS）に基づくものや、リダクション・テンプレートに基づく使いやすいものがあります。では、ヒストグラム計算の TLS バージョンを見てみましょう。

スレッド・ローカル・ストレージ（TLS）

ここでスレッド・ローカル・ストレージとは、スレッドごとにプライベート化されたデータのコピーを持つことを意味します。TLS を使用すると、スレッド間で共有される変更可能な状態へのアクセスを削減できるだけでなく、各プライベート・コピーを（場合によっては部分的に）スレッドが実行されているコアのローカルキャッシュに保存できるため、局所性も高まります。もちろん、コピーはスペースを占有するため、過度に使用すべきではありません。

TBB の重要な側面は、特定の時点で使用されているスレッドの数が不明であることです。64 コアのシステムで実行していて、64 回の反復に `parallel_for` を使用したとしても、64 個のスレッドがアクティブになるとは想定できません。これは、コードを構成可能にする上で重要な要素であり、並列プログラム内で呼び出された場合でも、並列で実行されるライブラリーを呼び出しても機能することを意味します。したがって、64 回の反復を持つ `parallel_for` の例では、データのスレッドローカルのコピーがいくつ必要か分かりません。TBB のスレッド・ローカル・ストレージのテンプレート・クラスは、コピー数を気にすることなく、TBB に適切な数のコピーを割り当て、操作し、結合する抽象的な方法を提供するためにあります。これにより、スケーラブルで構成可能、かつ移植可能なアプリケーションを作成できます。

TBB は、スレッド・ローカル・ストレージ用の 2 つのテンプレート・クラスを提供します。どちらもスレッドごとにローカル要素へのアクセスを提供し、要求に応じて要素を（遅延し

て）作成します。それぞれ、使用モデルが異なります:

`enumerable_thread_specific` クラスは、スレッドごとに 1 つの要素を含む STL コンテナーのように動作するスレッド・ローカル・ストレージを提供します。コンテナーは、通常の STL 反復の表現方法を使用して要素を反復します。どのスレッドも、他のスレッドのローカルデータを参照しながら、すべてのローカルコピーを反復処理できます。

`combinable` クラスは、後で 1 つの結果になる、スレッドごとのサブ計算を保持するスレッド・ローカル・ストレージを提供します。各スレッドは、ローカルデータのみ、または `enumerable_thread_specific::combine` を呼び出した後は結合されたデータのみを参照できます。

enumerable_thread_specific（ETS）

まず、`enumerable_thread_specific` クラスを使用して、並列ヒストグラム計算をどのように実装できるか考えてみましょう。図 8-22 は、入力画像の異なるチャンクを並列処理し、各スレッドでヒストグラム・ベクトルのプライベート・コピーを書き込むのに必要なコードを示しています。

```
#include <tbb/enumerable_thread_specific.h>
  // 並列実行
  using vector_t = std::vector<int>;
  using priv_h_t = tbb::enumerable_thread_specific<vector_t>;
  priv_h_t priv_h{num_bins};
  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
               [&](const tbb::blocked_range<size_t>& r)
               {
                 priv_h_t::reference my_hist = priv_h.local();
                 for (size_t i = r.begin(); i < r.end(); ++i)
                   my_hist[image[i]]++;
               });
  //プライベートヒストグラムの順次リダクション
  vector_t hist_p(num_bins);
  for(auto i=priv_h.begin(); i!=priv_h.end(); ++i){
     for (int j=0; j<num_bins; ++j)
        hist_p[j]+=(*i)[j];
  }
```

図 8-22. `enumerable_thread_specific` クラスを使用してプライベート・コピー上で並列ヒストグラム計算を実行します。サンプルコード: `synchronization/histogram_08_4th_safe_parallel_private.cpp`

まず、`std::vector<int>` 型の `enumerable_thread_specific` オブジェクト `priv_h` を宣言します。コンストラクターは、ベクトルのサイズが `num_bins` の整数であることを示します。次に、`parallel_for` 内で、不確定な数のスレッドが反復空間のチャンクを処理し、チャンクごとに `parallel_for` のボディー（この例ではラムダ）が実行されます。特定のチャンクを処理するスレッドは、次のように動作する `my_hist = priv_h.local()` を呼び

出します。このスレッドが local() メンバー関数を初めて呼び出すと、このスレッドに対して新しいプライベート・ベクトルが作成されます。逆に、初めてではない場合は、ベクトルはすでに作成されているので、それを再利用します。どちらの場合も、プライベート・ベクトルへの参照が返され、my_hist に割り当てられます。これは、指定されたチャンクのヒストグラム・カウントを更新するため parallel_for 内で使用されます。これにより、異なるチャンクを処理するスレッドは、最初のチャンクのプライベート・ヒストグラムを作成し、それを後続のチャンクで再利用できます。かなり改善できました。

parallel_for の最後には、すべての部分的な結果を累積して最終的なヒストグラム hist_p を計算するために結合する必要がある、未確定数のプライベート・ヒストグラムが残ります。しかし、プライベート・ヒストグラムの数が不明であるならば、このリダクションをどのように実行できるでしょう？ 幸いなことに、enumerable_thread_specific は、T 型の要素に対してスレッド・ローカル・ストレージを提供するだけでなく、最初から最後まで STL コンテナーのように反復処理することもできます。これは図 8-22 の最後で実行され、変数 i はさまざまなプライベート・ヒストグラムを順番に走査し、ネストされたループ j は hist_p にすべてのビンカウントを累積します。

優れた C++ プログラミング・スキルを披露したい場合、priv_h が別の STL コンテナーであるという事実を利用して、図 8-23 に示すようにリダクションを記述することもできます。

```
for  (const auto& i:priv_h) {  // i はすべてのプライベート・ベクトルを走査
  std::transform(hist_p.begin(),     // ソース 1 開始
                 hist_p.end(),       // ソース 1 終了
                 i.begin(),hist_p.   // ソース 2 開始
                 begin(),            // デスティネーション開始
                 std::plus<int>()  );// バイナリー操作
}
```

図 8-23. リダクションを実装するさらにスタイリッシュな方法。サンプルコード: synchronization/histogram_09_5th_safe_parallel_private.cpp

リダクション操作は頻繁に実行されるため、enumerable_thread_specific では、リダクションを実装する 2 つの追加メンバー関数（combine_each() と combin()）も提供されています。図 8-24 では、図 8-23 と完全に同等のコードでメンバー関数 combin_each を使用する方法を示しています。

```
priv_h.combine_each([&](const vector_t&  a)
    {  // 各 priv ヒストグラム a
     std::transform(hist_p.begin(),   // ソース 1 開始
                    hist_p.end(),     // ソース 1 終了
                    a.begin(),        // ソース 2 開始
                    hist_p.begin(),   // デスティネーション開始
                    std::plus<int>() );//  バイナリー操作
    });
```

図 8-24. リダクションを実装するには、combine_each() を使用します。サンプルコード synchronization/histogram_10_6th_safe_parallel_private.cpp

図 8-24 に示すように、関数 f はラムダとして提供され、STL 変換アルゴリズムがプライベート・ヒストグラムを hist_p に蓄積するのを監視します。一般に、メンバー関数 combin_each は、enumerable_thread_specific オブジェクト内の各要素に対して単項関数を呼び出します。シグネチャー void(T) または void(const T&) を持つこの結合関数は、通常、プライベート・コピーをグローバル変数に縮小します。

代替メンバー関数 combin() は、型 T の値を返します。図 8-25 では、T(T,T) シグネチャーを使用したリダクションの実装を示しています。このシグネチャーは、プライベート・ベクトルの各ペアに対して、ベクトル a へのベクトル加算を行い、それをさらにリダクションできるように返します。combine() メンバー関数は、ヒストグラムのすべてのローカルコピーを訪れ、最終的な hist_p へのポインターを返します。

```
vector_t  hist_p = priv_h.combine([](vector_t  a,
                                   vector_t  b) -> vector_t
        {// 各 priv ヒストグラム
    std::transform(a.begin(),      // ソース 1 開始
                   a.end(),        // ソース 1 終了
                   b.begin(),      // ソース 2 開始
                   a.begin(),      // デスティネーション開始
                   std::plus<int>() );// バイナリー操作
     return  a;
  });
```

図 8-25. 同じリダクションを実装するには、combine() を使用します。サンプルコード: synchronization/histogram_11_7th_safe_parallel_combine.cpp

並列パフォーマンスはどうでしょうか？

```
./histogram_08_4th_safe_parallel_private
Serial: 0.318113, Parallel: 0.0355086, Speed-up: 8.95876
```

ついに勝利！そして、さらに勝利が続きます！

この単純なヒストグラムでは、並列処理によるパフォーマンスの向上がついに確認できました。重要なことは、並列処理を実現する効果的な方法が必要だったことです。特に重要なことは、どんなテクニックでもうまくいくはずだと思い込まず、自身に合ったものを見つける必要があるということです。この章の残りの部分では、改善されるにつれて、さらに良いことを学べるでしょう。将来、他のアルゴリズムやアプリケーションに効果的に適用できるように、同期のオプションを段階を追って検討しています。

図 8-23、8-24、および 8-25 に示す 3 つの同等のリダクションは順番に実行されるため、削減するプライベート・コピーの数が多い場合（64 スレッドがヒストグラムを計算している場合など）、またはリダクション操作の計算量が多い場合（例えば、プライベート・ヒストグラムに 1,024 個のビンがある場合）、パフォーマンス向上の余地はまだあります。この問題にも対処しますが、その前に、スレッド・ローカル・ストレージを実装する 2 番目の代替手段について説明する必要があります。

組み合わせ可能

`combinable<T>` オブジェクトは、並列計算中にスレッドローカル値を保持するため、各スレッドに `T` 型の独自のローカル・インスタンスを提供します。前述の ETS クラスとは異なり、結合可能なオブジェクトは、図 8-22 と 8-23 の `priv_h` のように反復処理することはできません。ただし、この結合可能なクラスは、ローカル・データ・ストレージのリダクションを実装する唯一の目的で TBB で提供されているため、`combine_each()` および `combin()` メンバー関数が利用可能です。

図 8-26 では、並列ヒストグラム計算をもう一度再実装し、今度は結合可能クラスを用いています。

```
#include <tbb/combinable.h>

// 並列実行
using vector_t = std::vector<int>;
tbb::combinable<vector_t> priv_h{[num_bins](){
                                  return vector_t(num_bins);}};

parallel_for(tbb::blocked_range<size_t>{0, image.size()},
          [&](const tbb::blocked_range<size_t>& r)
           {
             vector_t& my_hist = priv_h.local();
             for (size_t i = r.begin(); i < r.end(); ++i)
               my_hist[image[i]]++;
           });

// プライベート・ヒストグラムの順次リダクション
vector_t hist_p(num_bins);
priv_h.combine_each([&](const vector_t& a)
  {  // 各プライベート・ヒストグラム a
    std::transform(hist_p.begin(), // ソース 1 開始
                   hist_p.end(),   // ソース 1 終了
                   a.begin(),      // ソース 2 開始
                   hist_p.begin(), // デスティネーション開始
                   std::plus<int>() );//  バイナリー操作
  });
```

図 8-26. 結合可能なオブジェクトを使用してヒストグラム計算を再実装します。サンプルコード: synchronization/histogram_12_8th_safe_parallel_combine.cpp

この場合、priv_h は組み合わせ可能なオブジェクトであり、コンストラクターは、priv_h.local() が呼び出されるたびに起動される関数を含むラムダを提供します。この場合、ラムダは num_bins 個の整数の初期ベクトルを作成するだけです。スレッドごとのプライベート・ヒストグラムを更新する parallel_for は、my_hist が単なる整数のベクトルへの参照であることを除けば、ETS の代替手段の図 8-22 の実装と非常によく似ています。前述のように、図 8-22 のようにプライベート・ヒストグラムを手動で反復処理することはできませんが、それを補うため、メンバー関数 combin_each() および combin() は、図 8-24 および 8-25 の ETS クラスの同等のメンバー関数と同じように動作します。このリダクションは引き続きシーケンシャルに実行されるため、削減するオブジェクト数が少ない場合や 2 つのオブジェクトを削減するのにかかる時間が短い場合にのみに適していることに注意してください。

最も簡単な並列実装: リダクション・テンプレート

2 章で説明したように、TBB には、parallel_reduce を簡単に実装する高レベルの並列アルゴリズムがすでに備わっています。それでは、プライベート・ヒストグラムの並列リダクションを実装する場合、この parallel_reduce テンプレートを活用してみてはどうでしょう。図 8-27 では、このテンプレートを使用して効率的な並列ヒストグラム計算のコードを示します。

```
#include <tbb/parallel_reduce.h>
  // 並列実行
  using vector_t = std::vector<int>;
  using image_iterator = std::vector<uint8_t>::iterator;
  vector_t hist_p = parallel_reduce (
    /*range*/
    tbb::blocked_range<image_iterator>{image.begin(), image.end()},
    /*identity*/
    vector_t(num_bins),

    // 最初の ラムダ: プライベート・ヒストグラムの並列計算
    [](const tbb::blocked_range<image_iterator>& r, vector_t v) {
           std::for_each(r.begin(), r.end(),
               [&v](uint8_t i) {v[i]++;});
           return v;
       },
    // 2 番目のラムダ: プライベートヒストグラムの並列リダクション
    [](vector_t a, const vector_t& b) -> vector_t {
       std::transform(a.begin(          // ソース 1 開始
                     ),
                     a.end(),           // ソース 1 終了
                     b.begin(),         // ソース 2 開始
                     a.begin(),         // デスティネーション開始
                     std::plus<int>() );// バイナリー操作
       return a;
    });
```

図 8-27. プライベート化とリダクションを使用する画像ヒストグラム計算のさらに優れた並列実装のコードリスト。サンプルコード: synchronization/histogram_13_9th_safe_parallel_reduction.cpp

parallel_reduce の最初の引数は、自動的にチャンクに分割され、スレッドに割り当てられる反復のレンジです。実際に内部で何が起こっているか単純化しすぎると、スレッドは、リダクション操作の ID 値で初期化されたプライベート・ヒストグラムを取得します。この場合、このヒストグラムは 0 で初期化されたビンのベクトルです。

最初のラムダは、画像のチャンクの一部のみにアクセスして得られる部分ヒストグラムのプライベートなローカルな計算を処理します。最後に、2 番目のラムダはリダクション操作を実装します。この場合は次のように表現できます:

```
[num_bins](vector_t a, const vector_t & b) -> vector_t {
    for(int i=0; i<num_bins; ++i) a[i] += b[i];
    return a;
});
```

これはまさに `std::transform` STL アルゴリズムが行っていることです。実行時間は、ETS と結合可能を使用した場合とほぼ同じです:

```
./histogram_13_9th_safe_parallel_reduction
Serial: 0.269894, Parallel: 0.0324757, Speed-up: 8.31066
```

ここまで説明してきたヒストグラムの実装の意味を明確にするため、図 8-28 にテストシステムで得られた高速化の結果をまとめています。正確なマシンやタイミングはここでは重要ではなく、私たちが遭遇する可能性のあるさまざまな局面についての視点を提供することが目的です。

実装:	安全でない	粗粒度	細粒度	アトミック	TLS	リダクション
スピードアップ	0.027	0.561	0.002	0.027	8.959	8.310

図 8-28. さまざまなヒストグラム実装のスピードアップの例

3 つの異なる動作状況があることが分かります。安全でない、細粒度のロック、およびアトミック・ソリューションは、4 コアではシーケンシャルよりもかなり遅くなります（ここでの「かなり遅い」とは、1 桁以上遅いことを意味します）。前述したように、ロックや偽の共有/真の共有による頻繁な同期は実際の問題であり、ヒストグラム・ビンが 1 つのキャッシュから別のキャッシュへ移動するとスピードアップは期待外れなものになります。ヒストグラム・ベクトルとミューテックス・ベクトルの両方に偽の共有と真の共有があるため、細粒度のソリューションは最悪です。 同じ種類の粗粒度のソリューションはシーケンシャルなソリューションよりもわずかに劣ります。これは単に「並列化してからシリアル化する」バージョンであり、粗粒度のロックによってスレッドがクリティカル・セクションに 1 つずつ入らなければならないことを覚えておいてください。

粗粒度バージョンのわずかなパフォーマンス低下は、実際には並列化とミューテックス管理のオーバーヘッドですが、これで偽の共有や真の共有はなくなりました。最後に、プライベート化 + リダクション・ソリューション（TLS および `parallel_reduce`）が先頭に立っています。これはかなりうまくスケールし、線形以上の性能向上を見せています。ただし、ツリー状のリダクションの影響で `parallel_reduction` は少し遅く、この問題では効果を発揮していません。コア数が少なく、リダクション（256 個の `int` ベクトルへの加算）に必要な時間はごくわずかです。この小さな問題では、TLS クラスで実装されたシーケンシャル・リダクションで十分です。

オプションの要約

ヒストグラム計算のような単純なアルゴリズムを実装するために提案したさまざまな代替案をすべてバックアップするため、各代替案の長所と短所をまとめ、詳しく説明しましょう。図 8-29 は、8 つのスレッドを使用して 800 個の数値をさらに簡単にベクトル加算するいくつかのオプションを示しています。対応するシーケンシャル・コードは次のようになります:

```
sum = 0;
for (int i = 0; i < N; ++i) sum += vec[i];
```

この章の役付けでは（この章のユーモアに興味がある方は、「ローレルとハーディ」をご覧ください）、私たちは「間違えた人、粗野な人、上品な人、全力の人、地元の人、そして賢い人」を選びました:

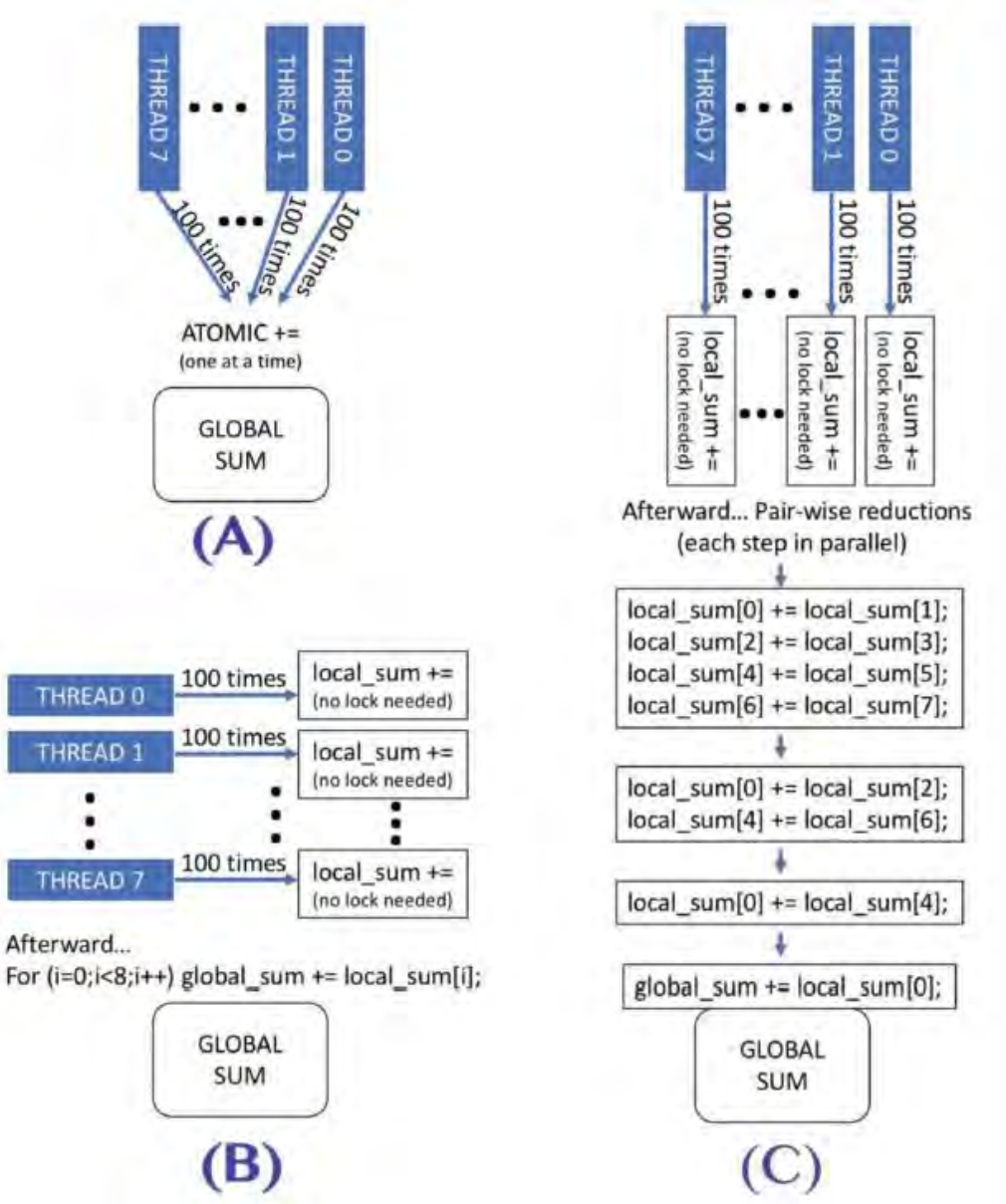


図 8-29. 8 つのスレッドで 800 個の数値を合計する競合を回避します: (A) `atomic`、アトミック操作でグローバル合計を保護する。(B) `local`、`enumerable_thread_specific` を使用する。(C) `wise`、`parallel_reduce` を使用する

- Mistaken (間違えた人): これ以上の考慮、熟考、後悔をすることなく、8 つのスレッドでグローバルカウンター sum_g を並行してインクリメントできます。おそらく、sum_g は不正確になり、キャッシュ・コヒーレンス・プロトコルもパフォーマンスを低下させます。十分に注意してください。

```
long  long  sum_g = 0;
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const  tbb::blocked_range<size_t>& r)
  {
    for  (int  i=r.begin(); i<r.end(); ++i) sum_g+=vec[i];
  });
```

- Hardy (粗野な人): 粗粒度のロックを使用すると正しい結果が得られますが、ミューテックスが HTM を実装していない限り (投機的な場合のように)、コードもシリアル化します。これはクリティカル・セクションを保護する最も簡単な代替手段ですが、最も効率的な方法ではありません。ベクトル和の例では、各ベクトルチャンクの累積を保護することで、粗粒度のロックを示し、粗粒度のクリティカル・セクションを取得します。

```
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const  tbb::blocked_range<size_t>& r){
    my_mutex_t::scoped_lock mylock{my_mutex};
    for  (int  i=r.begin(); i<r.end(); ++i) sum_g+=vec[i];
  });
```

LAUREL と HARDY

誰もが理解できるわけではないユーモラスな言及を削除するのではなく、私たちは教育的な小ネタを提供することにしました。粗粒度のロックと細粒度のロックについて、ユーモアを交えて議論してみましょう。

また、映画の初期の有名なコメディアンである Laurel＆Hardy を楽しんでいたとします。Hardy は粗野な人物、Laurel は上品な人物を演じました。Laurel は明らかにイギリス人で、Hardy は筋金入りのアメリカ人であるという事実を考えると、どんなに笑いが広がるか想像できます。
うまくいけば、Hardy（粗野な人）と Laurel（上品な人）を、Mistaken（間違えた人）、Nuclear（全力の人）、Local（地元の人）、Wise（賢い人）と一緒に共演することが、同期プログラミングの物語として論理的に思えた理由が分かるでしょう。さらに余力があれば、C++ プログラミングに忙しくないときに、これらの作品を簡単に楽しむことができることに注目してください。

- Laurel（上品な人）: 細粒度ロックは実装がより面倒であり、通常、データ構造の細粒度セクションを保護するさまざまなミューテックスを格納するためにより多くのメモリーが必要になります。ただし、良いことは、スレッド間の同時実行性が向上することです。さまざまなミューテックスを評価して、製品コードに最適なものを選択する必要があります。ベクトルの合計の場合、各部分を個別に保護できる分割可能なデータ構造がありません。
  より軽量なクリティカル・セクションを持つ細粒度の実装（次の例）を考えてみましょう（この場合は粗粒度の実装と同じくらいシリアルですが、スレッドはより細かい粒度でロック競合します）。

```
parallel_for(tbb::::blocked_range<size_t>{0, N},
  [&](const  tbb::blocked_range<size_t>& r){
      for  (int  i=r.begin(); i<r.end(); ++i){
        my_mutex_t::scoped_lock mylock{my_mutex};
        sum_g+=vec[i];
      }
  });
```

- Nuclear（全力の人）: 場合によっては、アトミック変数が役に立つことがあります。例えば、共有される変更可能な状態を整数型で保存でき、操作が単純な場合などです。これは細粒度ロック方式よりもコストが低く、同時実行レベルも同等です。ベクトル和の例（図 8-29(A) 参照）は次のようになります。これは、前の 2 つの方式と同様にシーケンシャルであり、グローバル変数の競合は細粒度ロック方式の場合と同様に高くなります。

```
std::atomic<long  long> sum_a{0};
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const  tbb::blocked_range<size_t>& r)
  {
    for  (int  i=r.begin(); i<r.end(); ++i) sum_a+=vec[i];
  });
```

- Local（地元の人）: 共有された変更可能な状態のローカルコピーをプライベート化して実装することで、常に問題を解決できるとは限りません。しかし、そのような場合は、`enumerate_thread_specific`（ETS）と `combinable` クラスを使用して、スレッド・ローカル・ストレージ（TLS）を実装できます。共同ワークを実行するスレッド数が不明な場合でも機能し、便利なリダクション・メソッドが提供されます。これらのクラスは、さまざまなシナリオで使用できる柔軟性を提供し、単一の反復空間でのリダクションでは十分ではないニーズに適合できます。ベクトルの合計を計算するため、図 8-29(B) に示すように、プライベート部分和 `priv_s` を後で順番に累積する代替案を以下に示します。

```
using  priv_s_t  = tbb::enumerable_thread_specific<long  long>;
priv_s_t  priv_s{0};
parallel_for(tbb::blocked_range<size_t>{0, N},
  [&](const  tbb::blocked_range<size_t>& r)
  { priv_s_t::reference my_s = priv_s.local();
    for  (int  i=r.begin(); i<r.end(); ++i) my_s+=vec[i];
  });
  long  long  sum_p = 0;
  for  (auto& i:priv_s) {sum_p+=i;}
```

- Wise（賢い人）: 計算がリダクション・パターンに適合する場合、TBB のスレッド・ローカル・ストレージ機能を使用してプライベート化とリダクションを手動でコーディングせず、`parallel_reduction` テンプレートを使用することを強く推奨します。次のコードは前のコードよりも複雑に見えるかもしれませんが、賢明なソフトウェア設計者は、この一般的なリダクション操作を完全に最適化する巧妙なトリックを考案しました。例えば、この場合、リダクション操作は、図 8-29(C) に示すように、複雑度が *O(n)* ではなく *O(n/p + log n)* のツリーのようなアプローチに従います。注: 並列操作の場合、並列リダクションの複雑性は *O(log n)* に減少することは珍しくありませんが、これは `n/p` が比較的小さくなる場合にのみ当てはまります。ここで、p は並列ワークのスレッドの数です。車輪の再発明をするのではなく、ライブラリーが提供するものを活用してください。それが、多数のコアとコストのかかるリダクション操作に最適なスケーリング方法です。

```
sum_p = parallel_reduce(tbb::blocked_range<size_t>{0, N}, 0,
[&](const tbb::blocked_range<size_t>& r, const long long&  mysum)
  {
    long  long  res = mysum;
    for  (int  i=r.begin(); i<r.end(); ++i)
    res+=vec[i]; return  res;
  },
[&](const  long  long& a, const  long  long& b)
  {
    return  a+b;
  });
```

ヒストグラム計算と同様に、図 8-30 に示すように、スレッドを 8 に設定したインテル Core Ultra プロセッサー ・ベースのシステムで、サイズ 109 のベクトル加算のさまざまな実装のパフォーマンスを評価します。これで計算はさらに細粒度になり（変数を増分するだけ）、109 回のロック/ロック解除操作またはアトミック・インクリメントの相対的な影響は大きくなります。これは、アトミック（Nuclear）実装と細粒度（Laurel）実装の高速化（より適切には減速!）に表れています。粗粒度（Hardy）の実装では、ヒストグラムの場合よりもわずかに大きな影響があります。安全でない（Mistaken）実装は、シーケンシャルよりも 5.75 倍高速になりました。真の勝者は両方とも、シーケンシャル・コードよりも 6.3 倍高速な TLS（Local）および `parallel_reduction`（Wise）実装です。

さまざまな構成でコードを試してみると、ハードウェアによっては、リダクション・コード（Wise）が TLS（Local）よりもはるかに高速になる場合もありますが、ほぼ同等になる場合もあります。これらは、Laurel と Hardy の悪い例よりも常に桁違いに高速であり、スレッドセーフでもあります。

実装:	Mistaken	Hardy	Laurel	Nuclear	Local	Wise
スピードアップ	5.7536	0.8140	0.0008	0.0095	6.2983	6.3107

図 8-30. N=109 のベクトル加算の異なる実装による高速化の例

なぜこれらすべての異なる選択肢を検討して、最後の選択肢を推奨するのか不思議に思うかもしれません。`parallel_reduce` ソリューションが最善であるのに、なぜ直接それを採用しなかったのでしょうか？ 残念ながら、並列処理は難しく、すべての並列化の問題が単純なリダクションで解決できるわけではありません。この章では、同期メカニズムが本当に必要な場合にそれを活用するデバイスを提供するだけでなく、可能な場合にはアルゴリズムとデータ構造を再考することの利点も示しました。

まとめ

TBB ライブラリーは、共有データに安全にアクセスするため、スレッドを同期できる相互排他をサポートしています。このライブラリーは、同期を回避するのに役立つスレッド・ローカル・ストレージ（TLS）クラス（ETS および `combinable` など）とアルゴリズム（`parallel_reduction` など）も提供します。この章では、画像ヒストグラムの計算を並列化する手順を見てきました。これらの機能はすべて、アトミック変数など最新の C++ 機能に依存し、相互作用します。
これらを組み合わせて使用することで最大の効果が得られる方法を分かりやすく説明するため、1 つのトピックとして扱いました。

最善の戦略は、同期の必要性が最も少ない（競合が最も少ない）問題を解決するアルゴリズムを見つけることであることを忘れないでください。

この章の実行例では、誤った実装から始まり、粗粒度ロック、細粒度ロック、アトミック、ロックを全く使用しない代替実装など、さまざまな同期の代替手段を反復するさまざまな並列実装を確認しました。途中、注目すべき点に触れ、ミューテックスを特徴付けるプロパティー、標準 C++ ライブラリーと TBB ライブラリーの拡張機能で利用可能なミューテックスの種類、アルゴリズムの実装がミューテックスに依存する場合に発生する一般的な問題などを紹介しました。

アルゴリズムを慎重に再考することで、よりクリーンな実装が可能になり、パフォーマンスが大幅に向上することが多いということを、忘れてはなりません。そうでない場合は、その章をもう一度読むべきでしょうか？ これは非常に重要な教訓です。

関連情報

ここでは、本章に関連した推奨文献を紹介します:

C++ Concurrency in Action, 第 2 版、Anthony Williams、Manning 2018 年に出版。

A Primer on Memory Consistency and Cache Coherence、Daniel J. Sorin、Mark D. Hill および David A. Wood、Morgan & Claypool Publishers、2011。

謝辞

図 8-1 のマラガ州ロンダの写真は、Rafael Asenjo によって撮影され、許可を得て使用しています。

図内に表示されているミームは、「365psd.com “33http://365psd.comVector meme faces”」から許可を得て使用しています。

図 8-17 の交通渋滞は、マラガ大学の博士課程の学生だった Denisa-Adreea Constantinescu が描いたもので、許可を得て使用しています。

9 章 キャンセルと例外処理

前の章では、並列処理を実現するため多数のタスクを作成する方法について説明しました。この章では、タスク管理の重要な 2 つの側面、つまりタスクのキャンセルとタスク内で発生する可能性のある例外処理に焦点を当てます。

C++ は、復旧力のあるアプリケーションの構築に不可欠な、堅牢な例外処理を提供します。TBB を使用すると、TBB を使用して作成したタスクで発生する例外を効果的にアクセスして管理できるようになります。

アプリケーションの使用中に、完了する前に後でキャンセルしたいタスクを開始する場合があります。これは、1 つのタスクの問題によって他のタスクの結果を生成する必要がなくなる例外により発生します。例えば、フローグラフを処理しているときに、1 つのステージが失敗して計算全体が無効になった場合は、残りのタスクをキャンセルするかもしれません。さらに、解決策がすでに見つかっている場合は、タスクをキャンセルして、他のタスクを不要にできます。例えば、複数の検索タスクを開始し、いずれかのタスクが一致を報告したらすぐに検索を終了したい場合があります。

TBB の例外とキャンセルのサポートは、いくつかの要因から生じる複雑性に対処します:

- 複数のスレッドによって実行されるタスク内で例外がスローされる可能性があります。
- 例外を引き起こすワークを速やかに終了できるようにするには、タスクのキャンセルを実装する必要があります。
- 処理全体を通じて構成の可能性を維持する必要があります。
- 例外が発生しない場合、例外管理がパフォーマンスに影響してはなりません。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_9

TBB 内での例外の実装は、タスクのキャンセルのサポートを含む、必要な要件をすべて満たしています。例外をスローすると、例外を生成した並列アルゴリズムの実行停止が必要となる場合があるため、タスクのキャンセルは不可欠です。
例えば、parallel_for アルゴリズムで範囲外またはゼロ除算例外が発生した場合、ライブラリーは parallel_for 全体のキャンセルが必要となる場合があります。
これには、TBB が並列反復空間のチャンクを処理するすべてのタスクを終了し、例外ハンドラーに遷移することが必要です。TBB のタスクキャンセルは、無関係な並列タスクに影響をすることなく、問題のある parallel_for に関係するタスクのキャンセルをシームレスに処理します。

タスクのキャンセルは、例外処理だけでなく、それ自体でも価値があります。この章では、まず、キャンセルを活用して並列アルゴリズムを高速化する方法を説明します。TBB のアルゴリズムはキャンセルを簡単に処理しますが、経験のある開発者はタスクのキャンセルを完全に制御し、TBB での実装を理解することを望むことがあります。この章では、そのような要望に対応することを目指します。

この章の残りの部分では、例外処理に焦点を当てています。例外処理は、複雑になることなく「そのまま機能します」。シーケンシャル・コードと同様に、使い慣れた try-catch 構造を使用するだけで、標準の C++ 例外と追加の TBB 固有の例外をキャプチャーできます。
また、TBB のカスタム例外を作成する方法を説明し、TBB 例外処理とキャンセルの相互作用について検討し、基本を超えた詳しい内容を説明します。

例外処理に懐疑的で、「エラーコード」を使用するアプローチを好む場合でも、読み続けて、信頼性の高いフォールトトレラントな並列アプリケーション開発における TBB 例外処理の利点を知ってください。

共同作業のキャンセル方法

状況によっては、タスクをキャンセルする必要があります。これは、外部要因（ユーザーが GUI ボタンを使用して操作をキャンセルするなど）または内部要因（アイテムが見つかりそれ以上の検索が不要になるなど）が原因となります。これらのシナリオはシーケンシャル・コードでは一般的ですが、並列アプリケーションでも発生します。例えば、計算負荷の高いグローバル最適化アルゴリズムの中には、分岐限定法の並列パターンを使用するものがあります。このパターンでは、検索空間はツリーとして編成されており、解決策が他の場所にある可能性が高い場合は、特定のブランチを通過するタスクをキャンセルすることが望ましいことがあります。

若干不自然ですが、次の例を使ってキャンセルをどのように活用できるか見てみましょう。例では、整数のベクトルデータ内の単一の -2 の位置を見つけます。この例は、data[500]=-2 と設定しているため、出力が事前に判明している（つまり、-2 がどこに格納されているかが分かっている）という不自然な例です。実装では、図 9-1 に示すように parallel_for アルゴリズムを使用します。

```
std::vector<int> data(n);
data[500] = -2;
intindex = -1;
autot1 = tbb::tick_count::now();
tbb::parallel_for(tbb::blocked_range<int>{0, n},
   [&](consttbb::blocked_range<int>& r){
      for(inti=r.begin(); i!=r.end(); ++i){
         if(data[i] == -2) {
            index = i;
            // 次の行をコメントアウトすると、実行時間が長くなる可能性があります
            tbb::task::current_context()->cancel_group_execution();
            break;
         }
      }
});
autot2 = tbb::tick_count::now();
std::cout << "Index "  << index;
std::cout << "found in "<< (t2-t1).seconds() << "seconds!\n";
```

図 9-1.　-2 が格納されているインデックスを検出。サンプルコード: cancellation/cancel_group_execution1.cpp

目標は、1 つのタスクが data[500]==-2 であることを検出したら、parallel_for 内のすべての同時タスクをキャンセルすることです。task::current_context()->cancel_group_execution() はどのように機能しますか？ task::current_context() は、最も内側のタスク・グループ・コンテキスト（TGC）への参照を返します。

タスク・グループ・コンテキスト（TGC）は、キャンセルできるタスクのグループを表します。TGC は tbb::task_group_context オブジェクトで表されます。名前からは分かりにくいかもしれませんが、tbb::task_group_context は、6 章で紹介した tbb::task_group とは異なります。6 章で紹介した tbb::task_group はタスクの実行と待機に使用され、tbb::task_group_context は実行中にタスク・スケジューラーによって使用される一連のプロパティーを表します。この章では、混乱を避け明確に区別するため TGC を使用します。

名前が示すとおり、task::current_context()->cancel_group_execution() は呼び出し元のタスクだけでなく、同じグループ内の *すべての*タスクをキャンセルします。

この例では、グループは parallel_for アルゴリズムに参加するすべてのタスクで構成されます。このグループをキャンセルすると、すべてのタスクが停止し、並列検索が中断されます。data[500]==-2 を見つけたタスクが兄弟たちに「見つけたよ！探すのをやめて！」と伝えることを想像してみてください。通常、各 TBB のアルゴリズムは独自の TGC を作成し、このグループ内のどのタスクもアルゴリズム全体をキャンセルできます。

サイズ n=1,000,000,000 のベクトルの場合、このループは 0.0004 秒かかり、出力は次のように表示されます:

```
Index 500 found in 0.000368532 seconds!
```

ただし、task::current_context()->cancel_group_execution() を省略すると、この例のテストに使用したシステムでは実行時間が数百倍（0.1 秒）長くなります。
基本的な TBB アルゴリズムのキャンセルに必要なのはこれだけです。タスクをキャンセルする説得力のある理由（この例では 200 倍以上のスピードアップを実現）があれば、タスクのキャンセル動作を詳しく調べ、どのタスクをキャンセルするか制御する戦略を検討できます。

高度なタスクキャンセル

オプションで、TGC オブジェクトを parallel_for やフローグラフなどの高レベル・アルゴリズムに渡すことができます。例えば、図 9-1 のコードに対する別のアプローチを図 9-2 に示します。

```
tbb::task_group_context tg;
...
tbb::parallel_for(tbb::blocked_range<int>{0, n},
   [&](const  tbb::blocked_range<int>& r){
      for(int  i=r.begin(); i!=r.end(); ++i){
         if(data[i] == -2) {
            index = i;
            // 次の行をコメントアウトすると、実行時間が長くなる可能性があります
            tg.cancel_group_execution();
            break;
         }
      }
}, tg); // parallel_for への新しいパラメーター
```

図 9-2. 図 9-1 の代替実装。サンプルコード: cancellation/cancel_group_execution2.cpp

このコードでは、TGC tg が作成され、parallel_for の最後の引数として渡され、tg.cancel_group_execution()（ここでは task_group_context クラスのメンバー関数）の呼び出しに使用されることが分かります。

図 9-1 と図 9-2 のコードは完全に同等であることに注意してください。parallel_for の最後の引数であるオプションの TGC パラメーター tg は、より複雑な開発への扉を開きます。例えば、並列スレッドで起動する parallel_pipeline に同じ TGC 変数 tg を渡すとします。これで、parallel_for または parallel_pipeline のいずれかで連携するすべてのタスクが tg.cancel_group_execution() を呼び出して、両方の並列アルゴリズムをキャンセルできるようになります。

タスクが TGC 全体のキャンセルをトリガーすると、キューで待機している生成されたタスクは実行されずに終了しますが、スケジューラーは非プリエンプティブであるため、すでに実行中のタスクは TBB スケジューラーではキャンセルされません。したがって、スケジューラーは、タスク本体に制御を渡す前に、タスクの TGC のキャンセルフラグをチェックし、タスクを実行するか、TGC 全体をキャンセルするか決定します。ただし、タスクがすでに制御権を持っている場合、タスクがスケジューラーに制御権を戻すまで、制御権は保持されます。

次の質問: 新しいタスクはどの TGC に割り当てられますか？ この割り当てを完全に制御するツールはありますが、デフォルトの動作も理解しておきましょう。まず、タスクを TGC に手動で割り当てる方法を説明します。

TGC の明示的な割り当て

これまで見てきたように、TGC オブジェクトを作成し、それを高レベルの並列アルゴリズム（parallel_for など）と低レベルのタスク API（tbb::task_group）に渡すことができます。同じ task_group::run() メンバー関数で起動されたすべてのタスクは同じ TGC に属するため、グループ内のどのタスクからでもセット全体をキャンセルできます。

例えば、図 9-3 のコードでは、データベクトル内の「隠匿された」特定の値のインデックスを見つけるため、並列検索を書き換えています。ここでは、task_group 機能を使用して、手動で実装された分割統治アプローチを使用します（parallel_for が内部的に行う処理に似ています）。クラス task_group は、基礎となる TGC をキャンセルする cancel 関数を提供します。図 9-3 では、g.cancel() を呼び出しており、g は task_group ですが、その呼び出しにより、根底となる TGC がキャンセルされることに注意してください。

```
int grainsize = 100;
std::vector<int> data;
int myindex=-1;
tbb::task_group g;

void SerialSearch(long begin, long end) {
   for(int i=begin; i<end; ++i){
      if(data[i]==-2){
         myindex=i;
         g.cancel();
         break;
} } }

void ParallelSearch(long begin, long end) {
   // 初回実行後にコメントを解除
   //if(tbb::is_current_task_group_canceling()) return;
   if((end-begin) < grainsize) { //カットオフに相当
      return SerialSearch(begin, end);
   } else {
      long mid=begin+(end-begin)/2;
      g.run([&]{ParallelSearch(begin, mid);}); // タスクをスポーン
      g.run([&]{ParallelSearch(mid, end);});    // 別のタスクをスポーン
   }
}

int main(int argc, char** argv)
{
   int n = 100000000;
   data.resize(n);
   data[n/2] = -2;

   auto t0 = tbb::tick_count::now();
   SerialSearch(0,n);
   auto t1 = tbb::tick_count::now();
   ParallelSearch(0,n);
   g.wait(); // スポーンされたタスクを待機
   auto t2 = tbb::tick_count::now();
   double t_s = (t1 - t0).seconds();
   double t_p = (t2 - t1).seconds();

   std::cout << "SerialSearch: " << myindex << " Time: " << t_s << std::endl;
   std::cout << "ParallelSearch: " << myindex << " Time: " << t_p
             << " Speedup: "     << t_s/t_p << std::endl;
   return 0;
}
```

図 9-3. task_group クラスを使用した並列検索の手動実装。サンプルコード: cancellation/cancel_group_execution3.cpp

便宜上、ベクトルデータ、結果の myindex、および task_group g はグローバル変数としています。このコードは、特定の粒度に達するまで検索空間を再帰的に分割します。この並列分割には関数 ParallelSearch(begin,end) が使用されます。粒度が十分に小さくなると（例では 100 回目の反復）、SequentialSearch(begin,end) が呼び出されます。ターゲット値 -2 が SequentialSearch でチェックされたレンジ内で見つかった場合、生成されたすべてのタスクは g.cancel() を使用してキャンセルされます。4 コアのラップトップで、N が 1000 万の場合、アルゴリズムの出力は次のようになります:

```
SerialSearch:   50000000 Time: 0.0201569
ParallelSearch: 50000000 Time: 0.00018978 Speedup: 106.212
```

5000000 は、ここで見つけた -2 値のインデックスです。

スピードアップを考慮すると、並列バージョンがシーケンシャル・コードより 119 倍高速に実行されるのは驚くべきことです。これは、並列実装ではシーケンシャル実装よりもワーク量が少なくなることが多いためであると考えられます。タスクがキーを見つけると、ベクトルデータをそれ以上走査する必要がなくなります。ここでの実行では、キーはベクトルの中央、N/2 にあります。シーケンシャル・バージョンではこのポイントに到達する必要がありますが、並列バージョンでは 0、N/4、N/2、N·3/4 などの異なる位置を同時に検索します。達成されたスピードアップに感銘を受けた場合、さらに改善の可能性があります。cancel() は実行中のタスクを終了できないことに注意してください。ただし、実行中のタスクは、TGC 内の別のタスクが実行をキャンセルしたかどうか確認できます。task_group クラスでこれを実装するには、実行中のタスク内の ParallelSearch() 関数の先頭に、

```
if(tbb::is_current_task_group_canceling()) return;
```

というチェックを追加するだけです（コードのコメントを解除します）。この一見マイナーな修正により、実行時間は次のようになります:

```
SerialSearch:   50000000 Time: 0.0217766
ParallelSearch: 50000000 Time: 0.000181158 Speedup: 120.208
```

キャンセルは間違いなく実装する価値があります。正確なアルゴリズムとデータに応じて、その効果ははるかに劇的となる可能性があります。キャンセルチェックのコストは無視できないため、使いすぎないように注意してください。いつものように、新しいテクニックを試すときはパフォーマンスをテストして影響を確認し、パフォーマンス結果を特定のニーズに合わせて調整します。

TGC のデフォルト割り当て

TGC を明示的に指定しないとどうなりますか？ デフォルトの動作は次のルールに従います:

- トップレベル（ルート）スレッドは、アルゴリズムを使用するときに暗黙的に独自の TGC を作成し、「**isolated**」としてタグが付けられます。このスレッドで実行される最初のタスクはその TGC に属し、後続の子タスクは同じ親の TGC を継承します。
- これらのタスクの 1 つが、オプション引数として TGC を明示的に渡さずに並列アルゴリズムを呼び出すと（例: `parallel_for`、`parallel_reduce`、`parallel_pipeline`、`task_group`、フローグラフなど）、ネストされたアルゴリズムで連携する新しいタスクに対して、「**bound**」というラベルの付いた新しい TGC が暗黙的に作成されます。したがって、この TGC は、分離された親 TGC にバインドされた子 TGC です。
- 並列アルゴリズムのタスクがネストされた並列アルゴリズムを呼び出す場合、この新しいアルゴリズムに対して新しいバインドされた子 TGC が作成され、親は呼び出しタスクの TGC になります。

仮想的な TBB コードによって自動的に構築された TGC ツリーのフォレストの例を図 9-4 に示します。

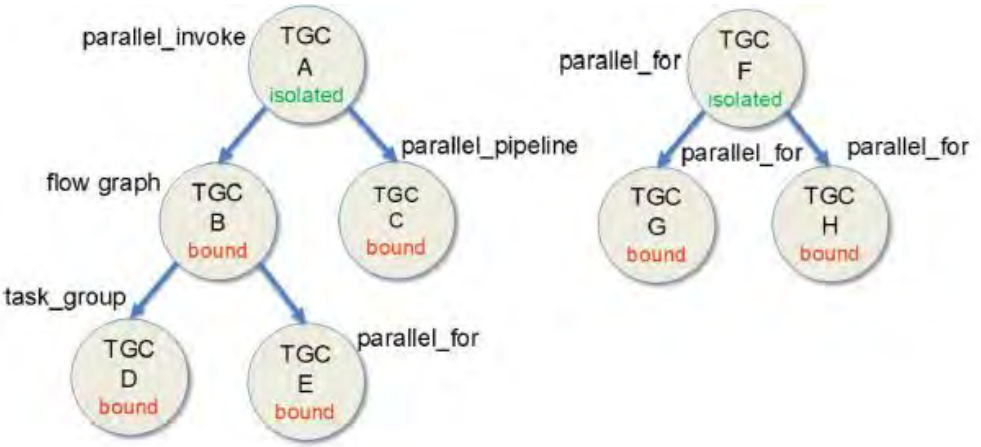


図 9-4. 仮想的な TBB コードを実行すると自動的に作成される TGC ツリーのフォレスト

この仮想的な TBB コードでは、ユーザーは複数の TBB アルゴリズムをネストしたいと考えていますが、TGC については何も知らないため、オプションの明示的な TGC オブジェクトを渡さずにアルゴリズムを呼び出しています。マスタースレッドでは、`parallel_invoke` が呼び出され、これによりスケジューラーが自動的に初期化され、1 つのアリーナと最初の分離された TGC `A` が作成されます。次に、`parallel_invoke` 内で、フローグラフと `parallel_pipeline` の 2 つの TBB アルゴリズムが作成されます。これらのアルゴリズムごとに、新しい TGC（この場合は `B` と `C`）が自動的に作成され、`A` にバインドされます。フローグラフ・ノードの 1 つの内部に `task_group` が作成され、別のフローグラフ・ノードに `parallel_for` がインスタンス化されます。その結果、新しく作成された 2 つの TGC（`D` と `E`）が `B` にバインドされます。これによって、分離されたルートを持ち、他のすべての TGC がバインドされている（つまり、親がある）TGC フォレストの最初のツリーが形成されます。2 番目のツリーは、2 つの並列レンジのみを持つ `parallel_for` を作成する別のマスタースレッドで構築され、各レンジに対してネストされた `parallel_for` が呼び出されます。ここでも、ツリーのルートは独立した TGC `F` であり、他の TGC `G` と `H` はバインドされています。ユーザーは、いくつかのアルゴリズムを他のアルゴリズムの中にネストして、TBB コードを単純に記述していることに注意してください。TBB は TGC のフォレストを自動的に構築します。複数のタスクが各 TGC を共有することに注意してください。

さあ、タスクがキャンセルされるとどうなるでしょうか？ 答えは、簡単です。ルールとしては、このタスクを含む TGC 全体がキャンセルされますが、キャンセルは下位にも伝播されます。例えば、フローグラフのタスク（TGC `B`）をキャンセルすると、図 9-5 に示すように、`task_group (TGC D)` と `parallel_for (TGC E)` もキャンセルされます。これは理にかなっています。フローグラフと、そこから作成されたすべてをキャンセルしているからです。この例は多少不自然で、このレベルのアルゴリズムのネストを備えた実際のアプリケーションを見つけるのは難しいかもしれません。ただし、これは、TBB の構成の可能性をサポートするため、さまざまな TGC がどのように自動的にリンクされるか効果的に示しています。

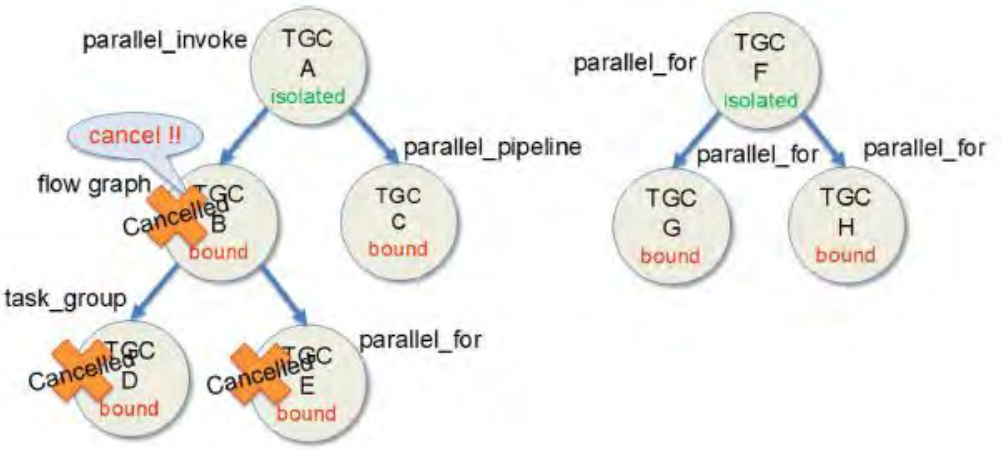


図 9-5. キャンセルは、TGC B に属するタスクから呼び出されます。

parallel_for (TGC E) をアクティブにしたままフローグラフと task_group をキャンセルする場合、分離された TGC オブジェクトを手動で作成し、それを parallel_for の最後の引数に渡すことでこれを実現できます。これは、図 9-6 のようなコードで実行できます。ここでは、フローグラフの function_node g がこの機能を利用しています。

```
tbb::flow::function_node<float,float> node{g,…,[&](float a){
   tbb::task_group_context TGC_E(tbb::task_group_context::isolated);
   // ネストした parallel_for
   tbb::parallel_for(0, N, 1,[&](…){ /*loop body*/ }, TGC_E);
   return a;
}};
```

図 9-6. ネストされたアルゴリズムを TGC のツリーから切り離す代替手段。サンプルコード: cancellation/cancel_group_execution4.cpp

分離された TGC オブジェクト TGC_E がスタック上に作成され、parallel_for の最後の引数として渡されます。ここで、図 9-7 に示すように、フローグラフのタスクが TGC B をキャンセルしても、キャンセルは TGC D まで下方向に伝播しますが、TGC E はツリーから切り離されて作成されているため、TGC E には到達しません。

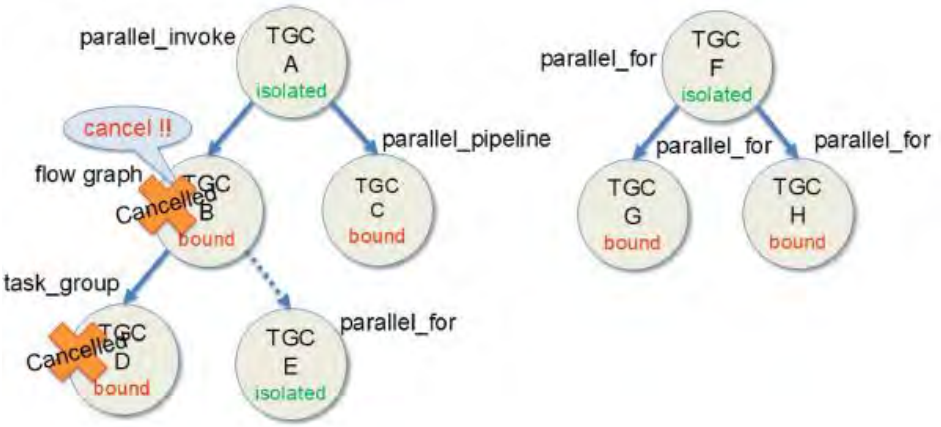


図 9-7. TGC E は分離されておりキャンセルされることはありません

正確には、分離された TGC E は分離された TGC であり、TGC のフォレスト内の別のツリーのルートになることができ、より深いネストされたアルゴリズム向けに作成された新しい TGC の親になることができます。次の節でこの例を見ていきます。

要約すると、TGC オブジェクトを明示的に渡さずに TBB アルゴリズムをネストすると、キャンセルの際にデフォルトの TGC フォレストは期待どおりの動作となります。ただし、必要な数の TGC オブジェクトを作成し、それを目的のアルゴリズムに渡すことで、動作を自由に制御できます。例えば、単一の TGC A を作成し、それを仮想 TBB の例の最初のスレッドで呼び出す並列アルゴリズムに渡すことができます。この場合、図 9-8 に示すように、すべてのアルゴリズムで連携するすべてのタスクがその TGC A に属します。ここでフローグラフのタスクがキャンセルされると、ネストされた `task_group` および `parallel_for` アルゴリズムだけでなく、TGC A を共有しているすべてのアルゴリズムもキャンセルされます。

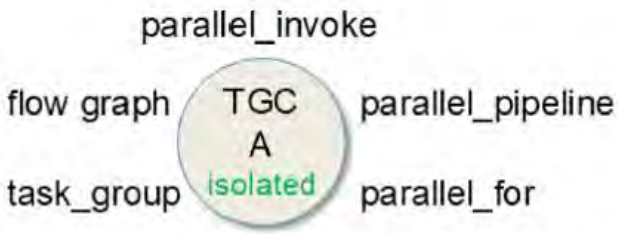


図 9-8. 仮想 TBB コードを修正して、単一の TGC A をすべての並列アルゴリズムに渡す

TBB における例外処理

C++ 例外

C++ 例外の基礎を説明するために、復習用の例を示します:

```
int  main(){
   try{
      try{
         throw  5;
      }
      catch  (const int& n){
         cout << "Re-throwing value: "  << n << endl;
         throw;
      }
   }
   catch(int& e){
      cout  << "Value caught: "  << e << endl;
   }
   catch  (...){
      cout << "Exception occurred\n";
   }
}
```

以下は、このサンプルコードの実行後の出力です。

```
Re-throwing value: 5
Value caught: 5
```

ご覧のとおり、最初の `try` ブロックにはネストされた `try-catch` が含まれています。これは、値 `5` の整数として例外をスローします。`catch` ブロックは型と一致するため、このコードは例外ハンドラーになります。ここでは、受け取った値のみを出力し、例外を上方向に再スローします。外部レベルには 2 つの `catch` ブロックがありますが、引数の型がスローされた値の型と一致するため、最初のブロックが実行されます。外部レベルの 2 番目の `catch` には省略記号（…）が付けられるため、例外の型が前の `catch` 関数のチェーンで考慮されていない型の場合、これが実際のハンドラーになります。例えば、`5` ではなく `5.0` をスローすると、出力メッセージは「`Exception occurred.`」になります。
サンプルコード `exception/cpp_exceptions.cpp`

キャンセルが TBB 例外管理をサポートする重要なメカニズムであることを理解できたので、次に、例外について考え、キャンセルが発生した場合の例外の動作について考えてみます。本書では std::exception_ptr については詳しく説明しません。これは、例外ハンドラーを記述するときに、TBB でも C++ と同様に役立つからです。ここでの目標は、図 9-9 に示すように、例外を実行する堅牢なコードの開発を習得することです。

```
int main(){
   std::vector<int> data(1000);
   try{
      tbb::parallel_for(0, 2000, [&] (int i) { data.at(i)++;  });
   }
   catch(const  std::out_of_range& ex) {
      std::cout << "Out_of_range: " << ex.what() << std::endl;
   }
   return  0;
}
```

図 9-9. TBB 例外処理の基本的な例。サンプルコード: exception/exception_catch1.cpp

まだ完璧ではないかもしれませんが、最初の例としては十分です。問題は、ベクトルデータには要素が 1000 個しかないのに、parallel_for アルゴリズムが位置 2000-1 まで移動することを要求していることです。さらに悪いことに、データは data[i] を使用してアクセスされるのではなく、data.at(i) を使用してアクセスされます。これは前者とは逆に、境界チェックを追加し、規則に従わない場合 std::out_of_range オブジェクトをスローします。したがって、図 9-9 のコードをコンパイルして実行すると、次のようになります。

```
Out_of_range: vector::_M_range_check: __n (which is 1500) >= this->size()
(which is 1000)
```

データ要素を並列にインクリメントするため、複数のタスクが生成されます。そのうちのいくつかは、999 を超える位置をインクリメントしようとします。範囲外の要素に最初にアクセスするタスク（例: data.at(1003)++）は明らかにキャンセルする必要があります。次に、std::vector::at() メンバー関数は、存在しない 1003 の位置をインクリメントする代わりに、std::out_of_range をスローします。例外オブジェクトはタスクによってキャッチされないため、上方に再スローされ、TBB スケジューラーに到達します。次に、スケジューラーが例外をキャッチすると、対応する TGC のすべての同時タスクをキャンセルします（TGC 全体がキャンセルされる方法はすでに分かっています）。さらに、例外オブジェクトへのポインターが TGC データ構造に格納されます。

すべての TGC タスクがキャンセルされると、TGC はファイナライズされ、TGC の実行を開始したスレッドで例外が再スローされます。この例では、これは `parallel_for` を呼び出したスレッドです。しかし、`parallel_for` は、`out_of_range` オブジェクトを受け取る `catch` 関数を含む `try` ブロック内にあります。つまり、`catch` 関数は例外ハンドラーになり、最終的に例外メッセージを出力します。`ex.what()` メンバー関数は、例外に関する詳細な情報を含む文字列を返す役割を担います。

実装の詳細に関する注意。コンパイラーは、TBB 並列アルゴリズムのスレッドの性質を認識しません。つまり、このようなアルゴリズムを `try` ブロックで囲むと、呼び出しスレッド（マスタースレッド）のみが保護され、ワーカースレッドも例外をスローする可能性があるタスクを実行することになります。これを解決するため、スケジューラーには `try-catch` ブロックがすでに含まれており、ワーカースレッドがタスクから発生する例外をインターセプトできます。

`catch()` 関数の引数は `const` 参照で渡すことを推奨します。そうすることで、基本クラスをキャプチャーする単一の `catch` 関数で、すべての派生型のオブジェクトをキャプチャーできます。

例えば、図 9-9 では、`catch(const std::out_of_range& ex)` ではなく `catch(const std::exception& ex)` と記述することもできました。これは、`std::out_of_range` が `std::logic_failure` から派生し、`std::logic_failure` が基本クラス `std::exception` から派生しており、参照によるキャプチャーですべての関連クラスがキャプチャーされるためです。

例外処理に関する注意事項

TBB は C++ の例外を完全にサポートしているため、残る課題は例外処理（またはその欠如）が最終的にプログラムにどのように影響するか検討することです。単一の例外は、タスク依存関係チェーン内で上方向に伝播します。最終的には、どのレベルでスローされたかに関係なく、並列アルゴリズム全体が適切に中断されます（シリアル・アルゴリズムと同様）。必要なレベルで例外をキャッチするか、独立した TGC で必要なネストされたアルゴリズムを構成することで、キャンセルの連鎖を防止できます。選択権は皆さんにあります。スレッドが並列実行されるときに発生する可能性のある多くの問題の原因（並列処理に関連する）を考慮する必要があることを除けば、TBB を使用しないプログラムの C++ 例外処理と変わりません。

まとめ

この章では、TBB の並列アルゴリズムをキャンセルし、実行時のエラーに対して例外処理を使用するのが簡単なプロセスであることを説明しました。どちらの機能も、デフォルト設定でそのままシームレスに動作します。また、TBB の重要な側面であるタスク・グループ・コンテキスト（TGC）についても説明しました。これは、TBB でキャンセルと例外処理を実装するために重要であり、手動で利用するとこれらの機能をより細かく制御できます。

まず、キャンセルの手順について説明し、タスクが TGC 全体をキャンセルする方法を説明しました。次に、TGC をタスクに手動で割り当てる方法と、開発者がこの割り当てを明示的に定義しない場合に適用されるルールを確認しました。デフォルトのルールでも、期待される動作が保証されます。並列アルゴリズムがキャンセルされると、ネストされたすべての並列アルゴリズムもキャンセルされます。

次に、例外処理について説明しました。TBB の例外はシーケンシャル・コードの例外と同様に動作しますが、並列処理の影響を考慮すると作業は複雑になります。1 つのタスクの 1 つのスレッドによってスローされた例外は、ライフタイムによっては別のスレッドでキャッチされる場合があります。最新の C++ 機能のおかげで、スレッド間で例外の正確なコピーを転送できます。TBB と C++ を組み合わせることで、並列アプリケーションで発生する可能性のあるあらゆる状況に対処するため、必要なすべての機能が提供されます。

関連情報

ここでは、本章に関連した推奨文献を紹介します:

Deb Haldar、Top 15 C++ Exception handling mistakes and how to avoid them（C++ 例外処理の間違いトップ 15 とその回避方法）。
www.acodersjourney.com/2016/08/top-15-c-exception-handling-mistakes-avoid/

10 章 パフォーマンス: 構成の可能性の柱

構成の可能性の種類

TBB ライブラリーは、構成可能なパフォーマンスを提供するように設計されています。TBB が構成可能な並列ライブラリーであることに言及する場合、開発者は TBB を使用するコードを好きな場所で自由に組み合わせることができることを意味します。TBB は、シリアルに行うことも、ネストすることも、並行して行うこともできます。TBB コードをこれらいずれかの方法で組み合わせた場合でも、プログラムは正常に動作し、パフォーマンスは良好になります。

並列プログラミング・モデルには、複雑なアプリケーションでは管理が難しい制限があることは、あまり知られていないかもしれません。もし、呼び出す関数内で間接的にも、「`if`」文内で「`while`」文を使用できなかったらどうなるか考えてみてください。TBB 以前にも、OpenMP など一部の並列プログラミング・モデルに同様に厳しい制限がありました。新しい SYCL 標準でさえ、完全な構成の可能性を欠いています（最も明白な例は、SYCL は単一ソースモデルですが、すべての SYCL コマンドが SYCL カーネル内で使用できるわけではないことです）。同様に、CUDA ではカーネルから実行できるコードの種類が厳しく制限されています。

構成不可能な並列プログラミング・モデルで最もいら立たしい点は、過度な並列処理を要求したり、間違った場所や間違ったタイミングで並列処理を導入する可能性があることです。これは最悪なことであり、TBB はこれを避けています。私たちの経験では、非構成可能モデルを熟知していないユーザーは並列処理を過度に使用しがちです。その結果、メモリー使用量の増加によりプログラムがクラッシュしたり、過剰な同期オーバーヘッドにより動作が極端に遅くなったりします。これらの問題に対する懸念から、経験豊富なプログラマーが並列処理を低く見積もり、負荷の不均衡やスケーリングの低下を引き起こす可能性があります。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_10

構成可能なプログラミング・モデルを使用すると、この難しいバランス調整の心配がなくなります。

TBB は構成の可能性を備えているため、単純なアプリケーションでも複雑なアプリケーションでも使用でき、高い信頼性があります。構成の可能性とは、並列処理を恐れることなく利用できるため、よりスケーラブルなプログラムを作成できる設計哲学です。

残念ながら、構成の可能性は、プログラミング・モデルの単純な「はい」や「いいえ」のような特性ではありません。OpenMP にはネストされた並列処理に対する構成の可能性の問題があることが知られていますが、OpenMP を構成不可能なプログラミング・モデルとして分類するのは誤りです。アプリケーションが OpenMP 構造を 1 つ呼び出してから別の OpenMP 構造を連続して呼び出す場合、このシリアル構成は正常に機能します。

同様に、TBB はあらゆる状況で他のすべての並列プログラミング・モデルとうまく連携し、完全に構成可能なプログラミング・モデルであると言っても過言ではありません。構成の可能性は、より正確には、2 つのプログラミング・モデル、またはそれ自身を組み合わせた単一のプログラミング・モデルが特定の方法で構成された場合に、どの程度適切に実行されるか示す尺度であると考えられます。

例えば、モデル A とモデル B という 2 つの並列プログラミング・モデルを考えてみましょう。TA を、モデル A を使用して外側レベルの並列処理を表現するカーネルのスループットとして定義し、TB を、モデル B（モデル A は使用しない）を使用して内側レベルの並列処理を表現する同じカーネルのスループットとして定義します。プログラミング・モデルが構成可能である場合、外側の並列処理と内側の並列処理の両方を使用したカーネルのスループットは `TAB >= max(TA, TB)` になると予想されます。TAB が `max(TA, TB)` よりどれだけ多いかは、モデルが互いにどの程度効率的に合成されるかと、コア数、メモリーのサイズなど、対象プラットフォームの物理的特性によって決まります。しかし、外側の並列処理と内側の並列処理を組み合わせて並列処理を加えても、一定期間内に完了できるワーク量が減少することに、ほとんどの開発者は驚くでしょう。残念ながら、構成不可能なモデルでは、これはよくあるケースです。

図 10-1 は、ソフトウェア構造の組み合わせに使用できる 3 つの一般的な構成の種類（ネストされた実行、同時実行、およびシリアル実行）を示しています。TBB は、図 10-1 に示す 3 つの方法のいずれかで TBB 並列アルゴリズムを他の TBB 並列アルゴリズムと合成すると、結果のコードのパフォーマンスが良好になり、スループット `TTBB1+TBB2 >= max(TTBB1, TTBB2)` となるため、構成可能なスレッド・ライブラリーであると言えます。TBB が共有アービトレーターを介して調停する別の並列モデルと組み合わされている場合も、同様に `TTBB+TOther >= max(TTBB, TOther)` となります。TBB のアービトレーターについては、この章の後半で説明します。

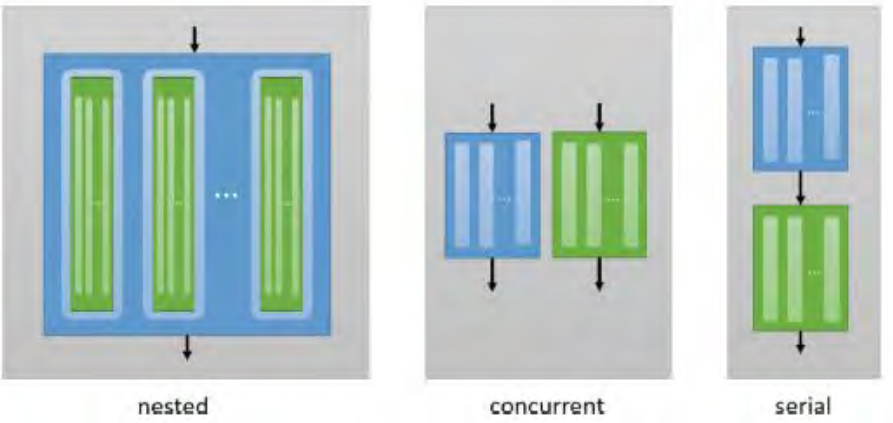


図 10-1. ソフトウェア構造を構成する方法

ただし、TBB が共有アービトレーターを介して TBB と調停しない別の並列モデルと組み合わされる場合、他のモデルが貪欲に動作する可能性があるため、$T_{TBB+Other} >= \max(T_{TBB}, T_{Other})$ は保証されません。それでも、TBB ライブラリーは良きコンポーネントであるように設計されており、TBB ライブラリーと連携しないライブラリーとでもうまく連携できる特性を備えています。この特性については、この章全体でさらに詳しく説明します。

ネストされた構成

ネストされた構成では、マシンは 1 つの並列アルゴリズムを別の並列アルゴリズム内で実行します。ネストされた構成の目的は、並列処理をさらに追加することであり、図 10-2 に示すように、並列で実行できるワーク量を指数関数的に増加できます。ネストされた並列処理を効果的に処理することが、TBB 設計における主な目標でした。

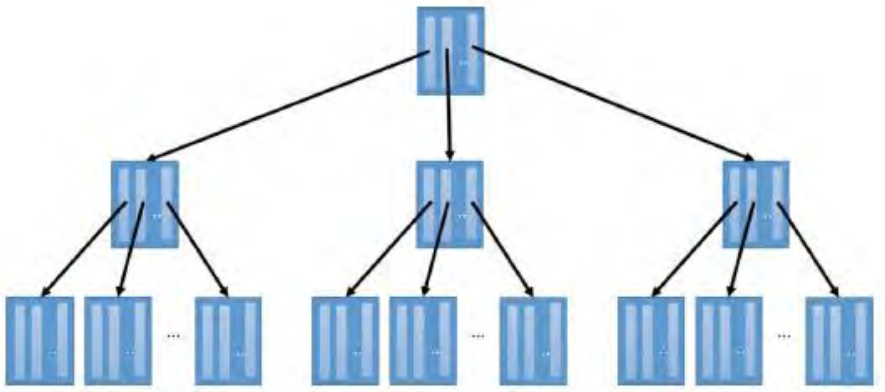


図 10-2.　ネストされた並列処理により、利用可能な並列タスクの数（または、非構成可能なライブラリーを使用する場合はスレッド）が指数関数的に増加する可能性があります

実際、TBB ライブラリーによって提供されるアルゴリズムは、多くの場合、スケーラブルな並列処理を作成するためネストされた構成に依存しています。例えば、2 章では、TBB の `parallel_invoke` のネストされた呼び出しを使用して、スケーラブルな並列バージョンのクイックソートを作成する方法について説明しました。TBB は、ネストされた並列処理を効果的に実行できるように設計されています。

TBB とは対照的に、他の並列モデルでは、ネストされた並列処理が存在するとパフォーマンスが著しく低下する場合があります。具体的な例としては OpenMP API があります。OpenMP は、共有メモリー並列処理に広く採用されているプログラミング・モデルであり、単一レベルの並列処理では非常に効果的です。ただし、その基本である並列ループ構造では、強制的な並列処理が定義上不可欠であるため、ネストされた並列処理には非常に不適切なモデルであると言えます。複数レベルの並列処理を持つアプリケーションでは、それぞれ OpenMP 並列構造によって追加のスレッドチームが作成されます。各スレッドはスタック領域を割り当て、OS のスレッド・スケジューラーによってスケジュールされる必要があります。スレッド数が非常に多い場合、アプリケーションのメモリーが不足する可能性があります。論理コア数よりもスレッド数が多い場合、スレッドはコアを共有する必要があり、ハードウェア・リソースのオーバーサブスクリプションにより追加スレッドによるメリットはほとんどなく、オーバーヘッドのみが課される傾向があります。

OpenMP を使用したネストされた並列処理の最も実用的な選択肢は、通常、ネストされた領域で並列処理を完全にオフにすることです。実際、OpenMP API は、ネストされた並列処理をオンまたはオフにする環境変数 `OMP_NESTED` を提供しています。TBB はシーケンシャル・セマンティクスを緩和し、スレッドではなくタスクを使用して並列処理を表現するため、利用可能なハードウェア・リソースに柔軟に並列処理を適応させることができます。TBB では、ネストされた並列処理を安全にオンにしておくことができます。TBB で並列処理のタイプを広範囲にオフにするメカニズムは必要ありません。

実際、前述したように、ネストされた並列処理は多くの TBB アルゴリズムの主要な設計の原則です。

この章の後半では、スレッドプールやワークスチール・タスク・スケジューラーなど、ネストされた並列処理を非常に効率良く実行する TBB の機能について説明します。

並行構成

図 10-3 に示すように、並行構成とは、並列アルゴリズムの実行が時間的に重なり合うことです。並行構成は、意図的に並列性を追加するのに使用できますが、関連のない 2 つのアプリケーション（または同じプログラム内の構成要素）が同じシステム上で同時に実行されると、偶然発生することもあります。並行実行と並列実行は必ずしも同じではありません。図 10-3 に示すように、並行実行とは 2 つの構成要素が同じ期間に実行されることであり、並列実行とは 2 つの構成要素が同時に実行されることです。つまり、並列実行は並行実行の一種ですが、並行実行は必ずしも並列実行とは限りません。
並行実行を効果的に並列実行に変換すると、並行構成によってパフォーマンスが向上します。

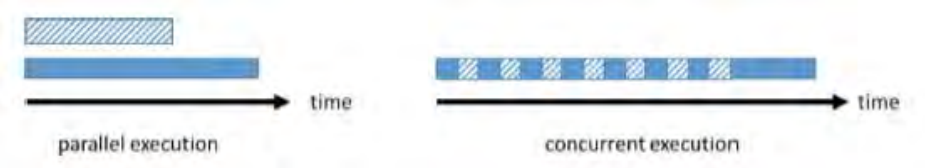


図 10-3. 並列実行と並行実行

注: C++ 標準では、並行進行保証、並列進行保証、弱並列進行保証などの用語を含む進行保証を定義しているため、混乱が生じる場合があることに注意してください。C++ 標準では、`sequenced_policy`、`unsequenced_policy`、`parallel_policy`、`parallel_unsequenced_policy` を含む実行ポリシーも定義されています。特に明記しない限り、「並列」および「並行」という用語を使用する場合、それらはこの節で定義されているものを意味します。C++ 標準の特定の用語を参照する場合は、それらの用語を明示的に参照します。

図 10-4 の 2 つのループの並行構成とは、2 つの異なるプロセスであっても、同じプロセス内の 2 つの異なるスレッドであっても、ループ 1 の並列実装がループ 2 の並列実装と並行に実行されることです。

```
// ループ 1                          // ループ 2
for  (int  i = 0; i < N; ++i)       for  (int  i = 0; i < M; ++i)
{                                   {
    b[i] = f(a[i]);                     d[i] = g(c[i]);
}                                   }
```

図 10-4. 同時に実行できる 2 つのループ

複数の構成要素を同時に実行する場合、アービトレーター（TBB などのランタイム・ライブラリー、オペレーティング・システム、または複数のシステムの組み合わせ）が、異なる構成要素にシステムリソースを割り当てます。2 つの構成要素が同時に同じリソースにアクセスする場合、これらのリソースへのアクセスをインターリーブする必要があります。

並行構成のパフォーマンスが良好であるということは、他のすべての構成要素がそれと並行して実行できるため、ウォールクロック実行時間が最も長い構成要素の実行時間と同じくらい短いことを意味します（図 10-3 の並列実行のように）。また、実行をインターリーブする必要がある場合（図 10-3 の並行実行のように）、ウォールクロック実行時間がすべての構成要素の実行時間の合計よりも長くないことが、良好なパフォーマンスであることを意味することがあります。しかし、理想的なシステムなど存在せず、破壊的干渉と建設的干渉の両方の発生により、どちらのケースにも完全に対応するパフォーマンスを得ることは困難です。

この章の後半で説明する TBB のスレッドプールとそのワークスチール・タスク・スケジューラーは、並行構成にも役立ち、調停のオーバーヘッドを削減し、多くの場合、リソースの使用を最適化するタスク分散を行います。TBB のデフォルトの動作に満足できない場合、必要に応じて 11 章で説明されている機能を使用して、リソース共有による影響を軽減できます。

シリアル構成

2 つの要素を構成する最後の方法は、それらを時間的に重複させずに次々に連続して実行することです。これはパフォーマンスに影響を与えない些細な構成のように思えるかもしれませんが、（残念ながら）そうではありません。シリアル構成を使用する場合、通常、2 つの構成要素間には干渉がないため、良好なパフォーマンスが得られると予想されます。

例えば、図 10-5 のループを考えると、シリアル構成ではループ 3 を実行してからループ 4 を実行します。シリアルに実行された各並列構成の完了にかかる時間は、同じ構成を単独で実行する時間と変わらないことが予想されます。並列プログラミング・モデル A を使用して並列処理を追加した後、ループ 3 のみの実行にかかる時間が $t_{3,A}$ で、並列プログラミング・モデル B を使用してループ 4 のみを実行するのにかかる時間が $t_{4,B}$ である場合、構造を連続して実行する合計時間は、各構造の時間の合計、$t_{3,3,A} + t_{4,B}$ 以下になると予想されます。

```
// ループ 3
for  (int  i = 0; i < N; ++i) {
    b[i] = f(a[i]);
}

// ループ 4
for  (int  i = 0; i < N; ++i) {
    c[i] = f(b[i]);
}
```

図 10-5. 2 つのループが順番に実行される

シリアル構成では、アプリケーションは 1 つの並列構造から次の並列構造に移行する必要があります。図 10-6 は、同じまたは異なる並列プログラミング・モデルを使用した場合の構成要素間の理想的な移行と理想的でない移行を示しています。どちらの理想的なケースでもオーバーヘッドはなく、1 つの構造から次の構造にすぐに移動します。実際には、構造を並列に実行した後にリソースをクリーンアップする時間がかかる場合が多く、次の構造を実行する前にリソースを準備するのにも時間がかかります。

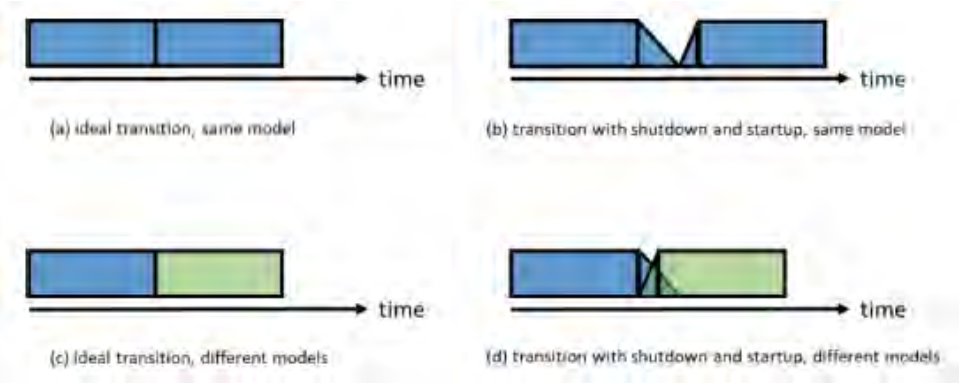


図 10-6. 異なる構造間での実行の移行

同じモデルを使用すると、図 10-6(b) に示すように、ランタイム・ライブラリーが並列ランタイムをシャットダウンする処理を実行しても、すぐに再起動しなければならない場合があります。図 10-6(d) では、構成要素に 2 つの異なるモデルが使用されていると、モデルが互いを認識しない可能性があり、最初の構成要素のシャットダウンと次の構成要素の起動、さらには実行が重複して、パフォーマンスが低下する可能性があることが分かります。どちらの場合も最適化可能であり、TBB はこれらの移行を考慮して設計されています。

また、あらゆる構成と同様に、2 つの構成要素間でリソースを共有すると、パフォーマンスに影響する可能性があります。ネストされた構成や同時実行構成とは異なり、構成要素はリソースを同時にまたはインターリーブして共有しませんが、それでも 1 つの構成要素が終了した後のリソースの終了状態が、次の構成要素のパフォーマンスに影響する可能性があります。例えば、図 10-5 では、ループ 3 で配列 b に書き込み、ループ 4 で配列 b から読み取っていることが分かります。ループ 3 とループ 4 の同じ反復処理を同じコアに割り当てることで、データの局所性が向上し、キャッシュミスが減少する可能性があります。 対照的に、同じ反復を異なるコアに割り当てると、不要なキャッシュミスが発生する可能性があります。互いを認識せず、反復がどこに配置されるか不明である 2 つのランタイムを使用すると、互換性のない割り当てが発生し、パフォーマンスが低下する可能性があります。

TBB が構成可能なライブラリーである理由

TBB ライブラリーは、設計上、構成可能なライブラリーです。最初に導入されたとき、フラットなモノリシックなアプリケーション開発者だけでなく、すべての開発者を対象とした並列プログラミング・ライブラリーとして、構成可能な課題に取り組む必要があるという認識がありました。TBB が使用されるアプリケーションは多くの場合モジュール化されており、それ自体に並列処理が含まれている可能性のあるサードパーティーのライブラリーを使用します。これら他の並列アルゴリズムは、意図する、しないに関わらず、TBB ライブラリーを使用するアルゴリズムと組み合わせられる場合があります。さらに、アプリケーションは通常、共有サーバーや個人用ラップトップなど、複数のプロセスが同時に実行されるマルチプログラム環境で実行されます。すべての開発者にとって効果的な並列プログラミング・ライブラリーとなるには、TBB は構成の可能性を正しく実現する必要があります。それは達成されています。

TBB の機能を使用してスケーラブルな並列アプリケーションを作成するのに TBB の設計を詳細まで理解する必要はありませんが、興味のある読者向けにこの節でいくつか詳しく説明します。また、次の章で説明するパフォーマンス機能を使用する場合、TBB のデフォルトの動作を制限または変更するたびに、構成の可能性のトレードオフを理解する必要があります。

TBB が正しい動作をすると信じて十分満足しており、その方法やデフォルトの変更にはあまり興味がない場合は、この章の残りの部分を飛ばしてもかまいません。そうでない場合は、TBB が構成の可能性において、なぜそれほど効果的であるのか、詳しく読んでみてください。

TBB スレッドプールとタスクアリーナ

TBB ライブラリーの構成可能性に関連する機能は、*グローバル・スレッド・プールと アリーナ*、およびスレッドをアリーナに割り当てる方法を決定する *アービトレーター*です。図 10-7 は、単一のメインスレッドを持つアプリケーションで、グローバル・スレッド・プールと単一のデフォルトアリーナがどのように相互作用するかを示しています。分かりやすくするため、ターゲットシステムには P = 4 個の論理コアがあると仮定します。

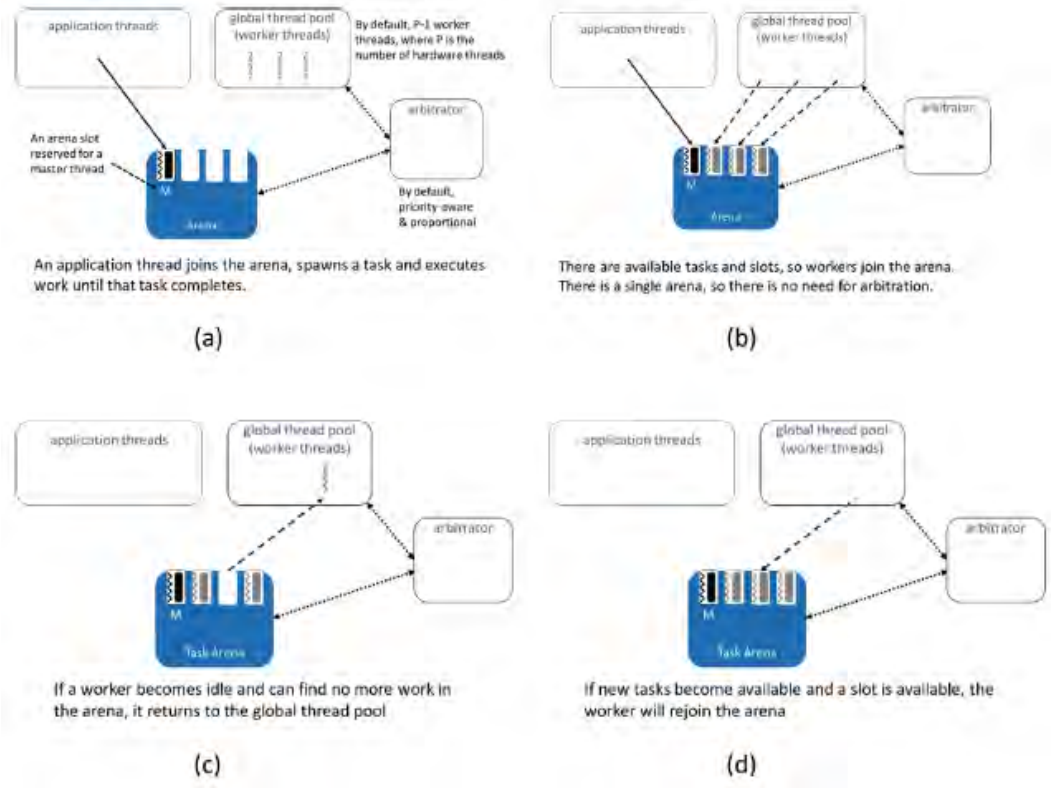


図 10-7.　多くのアプリケーションでは、メインスレッドが 1 つと、デフォルトのアリーナが 1 つあります。ワーカースレッドは、ワークの実行に参加するため TBB ライブラリーによって作成されます。

図 10-7(a) は、アプリケーションに 1 つのアプリケーション・スレッド（メインスレッド）と、P-1 個のスレッドで初期化されたワーカーのグローバル・スレッド・プールがあることを示しています。最初、グローバル・スレッド・プール内の各スレッドは、並列ワークに参加するのを待つ間、スリープ状態になります。図 10-7(a) は、単一のデフォルトのタスクアリーナが作成されていることを示しています。TBB を使用するアプリケーション・スレッドには、そのワークを他のアプリケーション・スレッドのワークから分離する独自のタスクアリーナが与えられます。図 10-7 では、アプリケーション・スレッドが 1 つしかないため、タスクアリーナも 1 つしかありません。アプリケーション・スレッドが TBB の並列アルゴリズムを実行すると、アルゴリズムが完了するまでそのタスクアリーナに関連付けられたディスパッチャーが実行されます。ディスパッチャーは、波線と実線のボックスを含むボックスで表されます。アルゴリズムが完了するのを待機する間に、マスタースレッドはアリーナに生成されたタスクの実行に参加できます。メインスレッドは、マスタースレッド用に予約されたスロットを埋め、ディスパッチャーとペアであることが示されています。

マスタースレッドがアリーナに参加して最初にタスクを生成すると、アービトレーターによってアリーナにスレッドが割り当てられ、図 10-7(b) に示すように、グローバル・スレッド・プールでスリープしているワーカースレッドが起動してアリーナに移行します。この例では、競合するアリーナがないため、このアリーナの空きスロットに割り当てが行われます。スレッドがアリーナに参加すると、そのスロットの 1 つを埋めることで、ディスパッチャーは、アリーナ内の他のスレッドによって生成されたタスクの実行に参加できるだけでなく、アリーナに接続されている他のスレッドのディスパッチャーで表示およびスチールできるタスクを生成することもできます。図 10-7 では、グローバル・スレッド・プールが P-1 個のスレッドを作成し、デフォルトのタスクアリーナに P-1 個のスレッド用のスロットが十分にあるため、タスクアリーナのスロットを埋める十分なスレッドが存在します。通常、メインスレッドと P-1 個のワーカースレッドが、オーバーサブスクライブすることなくマシンのコアを完全に占有するため、これが必要なスレッド数になります。アリーナが完全に占有されると、タスクの生成によって、グローバル・スレッド・プールで待機しているスレッドは起動されなくなります。

図 10-7(c) は、ワーカースレッドがアイドル状態になり、現在のアリーナで実行するワークがなくなると、アリーナの割り当てが減らされ、ワーカースレッドがグローバル・スレッド・プールに戻ることを示しています。その時点で、ワーカーは、ワーカーを必要とする別のタスクアリーナ（利用可能な場合）に参加できますが、図 10-7 ではタスクアリーナが 1 つしかないため、スレッドはスリープ状態になります。後にタスクが利用可能になると、アリーナの割り当てが調整され、グローバル・スレッド・プールに戻ったスレッドが再び起動してアリーナに参加し、図 10-7(d) に示すように追加ワークを支援します。

図 10-7 のシナリオは、1 つのメインスレッドがあり、追加のアプリケーション・スレッドがなく、デフォルトを変更するため TBB の高度な機能が使用されていないアプリケーションの一般的なケースを表しています。11 章では、さまざまな優先順位とスロット数を持つ複数のタスクアリーナを含む、複雑な例を作成する高度な TBB 機能について説明します。タスクアリーナのスロット数がワーカースレッド数より多い場合、アービトレーターが各アリーナに割り当てるスレッドの数を決定します。本書の執筆時点では、TBB はデフォルトで、優先順位を考慮した比例調停ポリシーを使用しています。本書では、TBB の調停選択について説明する際に、このポリシーが有効であると想定しています。

図 10-8 は、この調停ポリシーの主な特徴を示す 2 つの簡単な事例です。図 10-8(a) には、優先順位が同じ 2 つの競合アリーナがあり、ワーカーのスロットは合計 6 つですが、使用できるワーカースレッドは 3 つしかありません。アービトレーターはワーカーを両者の間で比例配分し、それぞれ 3 つのワーカーの半分を受け取ることになります。図 10-8(b) では、2 つのアリーナの優先順位が異なります。アービトレーターは、優先順位の低いアリーナにワーカーを提供する前に、優先順位の高いアリーナを完全に埋めます。

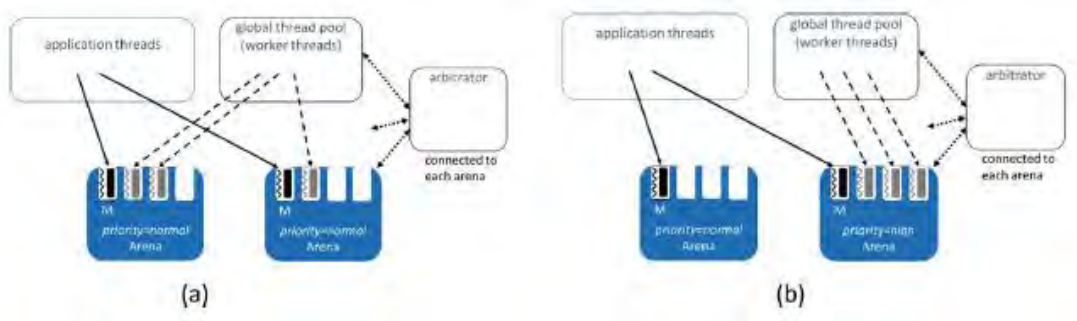


図 10-8. 競合するアリーナは、アービトレーターによって決定された割り当てに基づいてワーカースレッドを受け取ります。デフォルトの優先度を考慮した比例アービトレーターを使用すると、(a) 同じ優先度を持つアリーナはワーカースロットに比例したスレッドを受け取り、(b) 優先度の低いアリーナにスレッドが割り当てられる前に、優先度の高いアリーナが完全に埋められます。

11 章では、図 10-8 と 10-9 に示すような、複雑な例を作成できる高度な TBB 機能について説明しています。図 10-9 には、多数のアプリケーション・スレッドと複数のアリーナがあり、アリーナに関する興味深い点がいくつか示されています。デフォルトでは、図 10-7 と 10-8 に示すように、アプリケーション・スレッド用に 1 つのスロットが予約されています。ただし、図 10-9 の右側の 2 つのタスクアリーナに示すように、アプリケーション・スレッド用に複数のスロットを予約するタスクアリーナ、またはアプリケーション・スレッド用のスロットを全く予約しないタスクアリーナを作成することもできます。

アプリケーション・スレッドは任意のスロットを埋めることができますが、グローバル・スレッド・プールからアリーナに移行するワーカースレッドは、アプリケーション・スレッド用に予約されたスロットを埋めることはできません。アービトレーターは、各アリーナのワーカースレッドに使用できるスロット数を認識し、それに応じて配置を決定します。

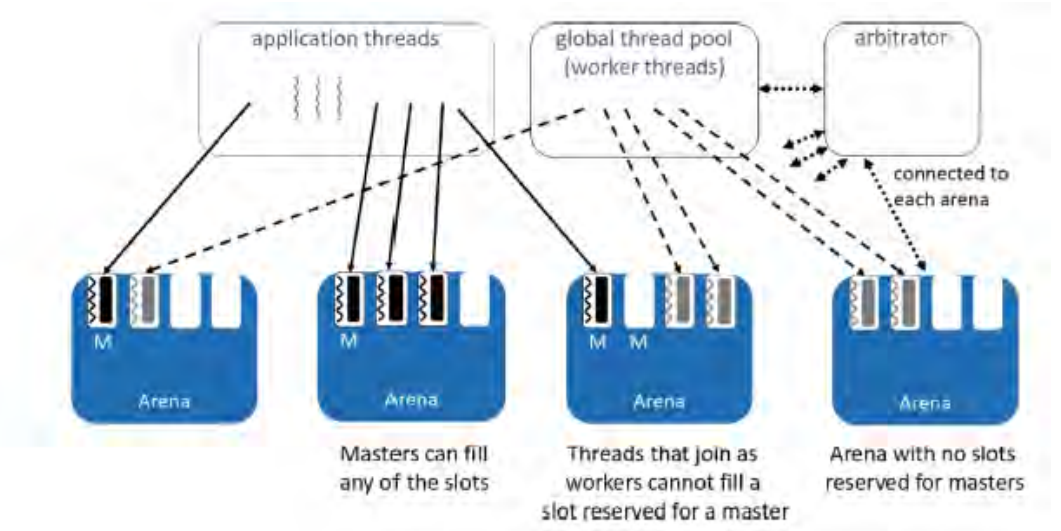


図 10-9.　アリーナには、マスタースレッド用に予約されたスロットが複数あることもあれば、スロットが全くない場合もあります

ただし、アプリケーションが複雑であっても、TBB ライブラリーがワーカースレッドを割り当てて要求する単一のグローバル・スレッド・プールが常に存在します。11 章では、初期化時に、または必要に応じて動的に、グローバル・スレッド・プールに割り当てられるスレッド数を変更する機能について説明します。しかし、この限られたワーカースレッドのセットは、プラットフォーム・コアの意図しないオーバーサブスクリプションを防ぐため、TBB が構成可能である理由の 1 つです。

各アプリケーション・スレッドには、独自の暗黙的なアリーナも割り当てられます。スレッドは別のタスクアリーナにあるスレッドからタスクをスチールできないため、デフォルトで異なるアプリケーション・スレッドによって実行されるワークが適切に分離されます。

TBB の設計上、TBB タスクを使用するアプリケーションとアルゴリズムは、ネスト、同時、またはシリアルで実行されても適切に構成されます。ネストされている場合、すべてのレベルで生成された TBB タスクは、TBB ライブラリーによってアリーナに割り当てられたワーカースレッドのみを使用して同じアリーナ内で実行されるため、スレッド数の爆発的な増加が防止されます。異なるマスタースレッドによって同時に実行される場合、ワーカースレッドはアリーナ間で分割されます。シリアルに実行される場合、ワーカースレッドは構造全体で再利用されます。

TBB タスク・ディスパッチャー: ワークスチールなど

スレッディング・ビルディング・ブロックのスケジュール戦略は、多くの場合、ワークスチールと呼ばれます。そしてこれはほぼ真実です。ワークスチールは、タスクが動的に生成され、マルチプログラム・システムで実行される動的な環境やアプリケーションで適切に機能するように設計された戦略です。タスクがワークスチールによって分散されると、スレッドは、受動的にワークが割り当てられるのではなく、アイドル状態になると積極的に新しいタスクを探します。この従量制によるワークの分散アプローチは、ワークの一部を他のアイドル状態のスレッドに分散するためだけに、スレッドに有効なワークの実行を強制停止させる必要がないため、効率的です。 ワークスチールは、オーバーヘッドをアイドルスレッドに移動しますが、アイドルスレッドには他にやるべきことがありません。ワークスチール・スケジューラーは、タスクが最初に生成されたときに事前にワーカースレッドにタスクを割り当てる *ワークシェア*・スケジューラーとは対照的です。タスクが動的に生成され、一部のハードウェア・スレッドが他のスレッドよりも負荷が重かったり、能力が低い動的環境では（例えば、E-core と P-core）ワークスチール・スケジューラーの反応が多くなるため、負荷分散が向上しパフォーマンスが向上します。

TBB アプリケーションでは、スレッドは、特定のタスクアリーナに接続されたタスク・ディスパッチャーを実行することで、TBB タスクの実行に参加できます。図 10-10 は、各アリーナで維持される重要なデータ構造の一部と、アリーナに参加するスレッドによって個別に維持されるデータ構造の一部を示しています。

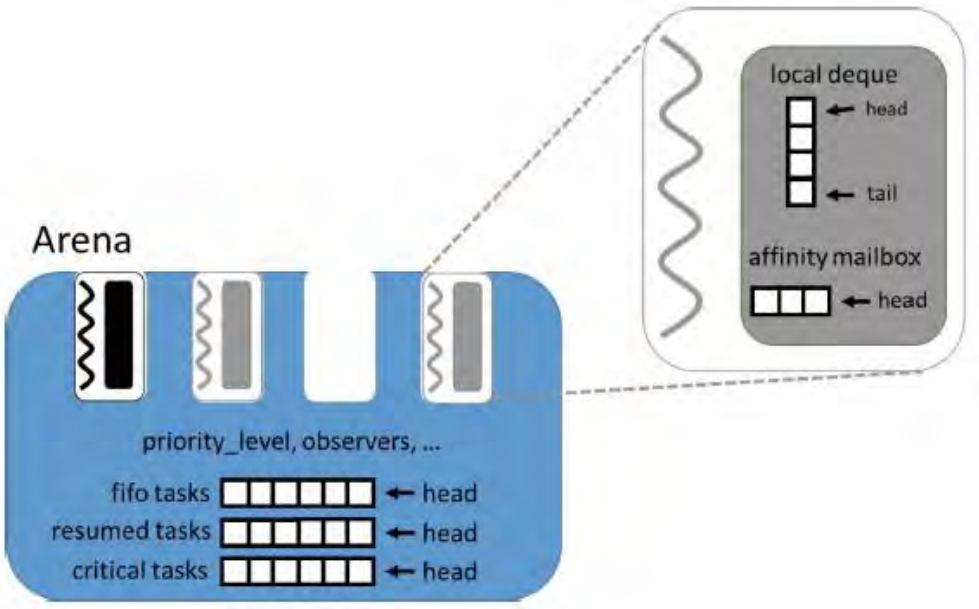


図 10-10. アリーナ内のキューとスレッドごとのデータ構造。図では単純なキューとして示されていますが、TBB ではスケーラブルなキューが実装されることに注意してください。

現時点では、タスクアリーナの共有キューとスレッドごとのデータ内のアフィニティー・メールボックスを無視し、タスク・ディスパッチャー内のローカルデック [1] のみに注目します。これは、TBB でワーク・スチール・スケジュールの戦略実装に使用されるローカルデックです。他のデータ構造は、ワークスチールの拡張機能を実装するために使用されますが、これについては後で説明します。

2 章では、TBB ライブラリーに含まれる汎用並列アルゴリズムによって実装されるさまざまな種類のループについて説明しました。それらの多くは、ループの反復空間を再帰的に分割可能な値のセットであるレンジの概念に依存しています。これらのアルゴリズムは、ループのレンジを再帰的に分割し、*分割タスク*を使用してサブ範囲を作成し、ループボディーとペアにして*ボディータスク*として実行するのに適したサイズに達するまで処理します。図 10-11 は、ループパターンを実装するタスクの分布例を示しています。最上位レベルのタスク `t0` の完全なレンジの分割を表します。これは、ループボディーが各指定されたサブレンジに適用されるリーフに再帰的に分割されます。図 10-11 に示す分散では、各スレッドは連続した 1 つのスレッドにわたってボディータスクを実行します。

[1] デック（deque）は両端キュー（データ構造）を意味します。キューから項目を削除するアクションであるデキュー（dequeue）と混同しないでください。

隣接する反復処理では近くのデータにアクセスすることが多いため、この分散では局所性が最適化される傾向があります。また、スレッドは分離されたタスクツリー内でタスクを実行するため、スレッドが動作する最初のサブレンジを取得すると、他のスレッドとあまり相互作用なくそのツリー上で実行できます。

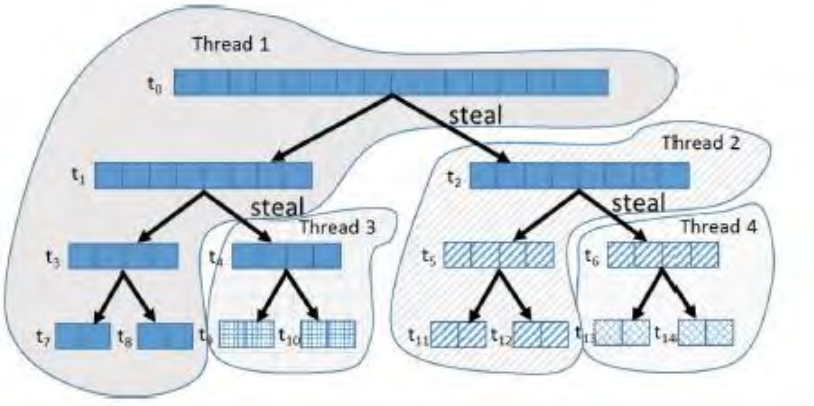


図 10-11. ループパターンを実装するタスクの分散

TBB ループ・アルゴリズムは、*キャッシュ非依存アルゴリズム*の例です。皮肉なことに、キャッシュ非依存アルゴリズムは、CPU データキャッシュの利用を高度に最適化するように設計されています。キャッシュやキャッシュライン・サイズを意識することなくこれを実行します。TBB ループ・アルゴリズムと同様に、これらのアルゴリズムは通常、データセットをサイズにかかわりなく最終的にデータキャッシュに収まる小さなサイズに再帰的に分割する分割統治アプローチを使用して実装されます。キャッシュ無視するアルゴリズムについては、11 章で詳しく説明します。

TBB ライブラリーのタスク・ディスパッチャーは、ローカルデックを使用して、キャッシュ非依存アルゴリズムで動作するように最適化されたスケジュール戦略を実装し、図 10-11 のような分散を行います。この戦略は、深さ優先ワーク、幅優先スチールポリシーと呼ばれることもあります。スレッドが新しいタスクを生成すると（つまり、そのタスクをタスクアリーナで実行できるようにすると）、そのタスクはタスク ・ディスパッチャーのローカルデックの先頭に配置されます。 その後、現在処理中のタスクが終了し、新しいタスクの実行が必要になると、図 10-12 に示すように、最後に生成したタスクを取得して、ローカルデックの先頭からワークを取得します。ただし、タスク・ディスパッチャーのローカルデックに利用可能なタスクがない場合、タスクアリーナ内の別のワーカースレッドをランダムに選択することで、ローカル以外のワークを探します。 ディスパッチャーが選択されたスレッドからタスクをスチールしようとしているので、選択されたスレッドを犠牲者（`victim`）と呼びます。犠牲者のローカルデックが空でない場合、ディスパッチャーは、図 10-12 に示すように、犠牲スレッドのローカルデックの末端からタスクを取得し、そのスレッドによって最近生成されたタスクを取得します。

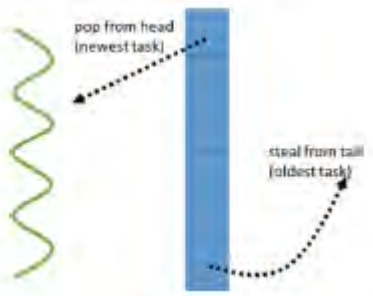


図 10-12.　タスク・ディスパッチャーがローカルデックの先頭からローカルタスクを取得し、犠牲スレッドのデックの末端からタスクをスチールするのに使用するポリシー

図 10-13 は、TBB ライブラリーが、図 10-11 のような配置を行う方法を簡略化した例を示しています。TBB アルゴリズムの実装は高度に最適化されているため、タスクを生成することなく一部のタスクを再帰的に分割したり、スケジューラー・バイパス（この章の後半で説明）などの手法を使用する場合があります。図 10-13 の例では、各分割タスクとボディータスクがタスクアリーナに生成されることを前提としています。これは、最適化された TBB アルゴリズムには当てはまりませんが、説明のためこの仮定を行っています。

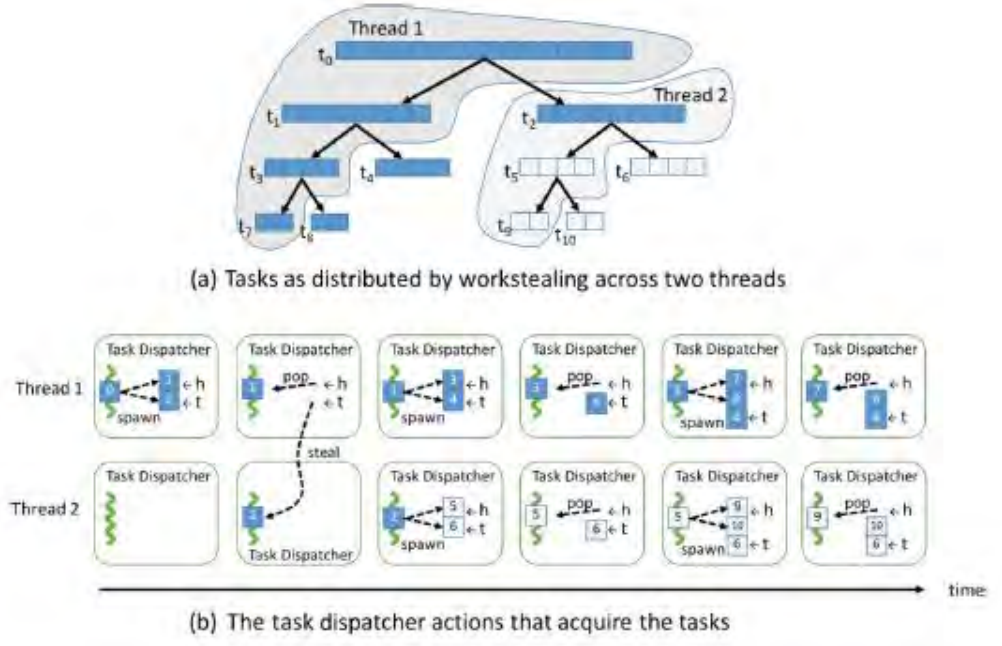


図 10-13.　タスクが 2 つのスレッド間でどのように分散されるか、および 2 つのタスク・ディスパッチャーがタスクを取得するために実行したアクションのスナップショット。注: TBB ループパターンの実装では、スケジューラー・バイパスや、一部のスポーンを削除する最適化などが使用されます。それでも、スチールと実行の順序はこの図のようになるでしょう。

図 10-13 では、スレッド 1 がルートタスクから開始され、最初にレンジを 2 つの大きな領域に分割します。次に、タスクツリーの一方を深さ優先で進み、リーフに到達するまでタスクを分割し、そこでボディーを最終サブレンジに適用します。最初にアイドル状態のスレッド 2 は、スレッド 1 のローカルデックの末端からデータをスチールし、スレッド 1 が元のレンジから作成した 2 番目の大きな領域を自身に供給します。図 10-13(a) は、タスク t_4 と t_6 がまだどのスレッドでも実行されていない時点のスナップショットです。さらに 2 つのワーカースレッドが使用可能である場合、図 10-11 のような分布が得られることが想像できます。図 10-13(b) のタイムラインの最後では、スレッド 1 と 2 のローカルデックにはまだタスクが残っています。次のタスクをポップすると、完了したタスクに隣接するリーフを取得します。

図 10-11 と 10-13 の分布は、単なる 1 つの可能性であることを忘れてはなりません。反復ごとのワークが均一で、どのコアもオーバーサブスクライブされていない場合は、図のような均等な配分が得られる可能性が高くなります。ただし、ワークスチールとは、スレッドの 1 つが過負荷のコアで実行されていると、そのスレッドがスチールする頻度が低くなり、結果として取得するワークが少なくなることを意味します。その後、他のスレッドがその不足を補います。反復をコアに静的かつ均等に分割するだけのプログラミング・モデルでは、このような状況に適応できません。

しかし、前述したように、TBB タスク・ディスパッチャーは単なるワークスチール・スケジューラーではありません。図 10-14 と 10-15 は、TBB タスク・ディスパッチ・ループ全体の簡略化された疑似コードを表現しています。

```
try_to_get_next_task:
  t_next = 重要なタスクを取得
    else   スレッドのローカルデデックの最後からポップ
  if  t_next は無効
    do
      do
        t_next = 重要なタスクを取得
          else   スレッドのアフィニティー・メールボックスから取得
          else   アリーナの再開されたタスクから取得
          else   アリーナの FIFO タスクから取得
          else   同じアリーナ内のランダムな被害スレッドからタスクをスチール
      while  t_next は無効であり、スレッドはこの領域に留まる
    while  スレッドはこのアリーナ内に留まる
  end if

  return  t_next
```

図 10-14. ディスパッチャーがアリーナ内のスレッドの次のタスクを見つける方法を示す疑似コード。スレッドが「このアリーナに留まる」のは、ワーカースレッドであり、アリーナの割り当てが削減されておらず、新しいタスクの検索に要した時間が比較的短い場合です。ワークを探すのに費やされる適切な時間は実装依存であり、TBB ユーザーが制御することはできません。スレッドがマスターであり、そのワークがまだ完了していない場合は「アリーナに留まる」必要があります。

図 10-14 は、スレッドがアイドル状態のときに、次に実行するタスクを見つけるロジックを示しています。図 10-14 からは、クリティカル・タスクが優先されることが分かります。現在、開発者がクリティカル・タスクを直接送信する公開 API はありませんが、一部の TBB 機能の実装では、タスクのスケジュールを設定しワークを優先順位付けする必要がある場合にクリティカル・タスクが使用されます。クリティカル・タスクがある場合、すべての方法に優先して次のタスクを探します。アプリケーション開発者がクリティカル・タスクを直接送信していなくても、特定の機能の実装でそれらの使用方法が変わる可能性があるため、本書では詳しくは説明しません。

次に、図 10-14 では、ローカルデックがチェックされます。ローカルデックには、このスレッドによって生成されたタスクが保持されます。これらのタスクは、次のタスクを探しているスレッドによってスポーンされたため、時間および空間の局所性を示す可能性があります。

ローカルデックに何もない場合、スレッドは繰り返しワークを探します。内側のループでは、まずクリティカル・タスクをチェックし、次に共有メールボックスをチェックし（アフィニティー・メールボックスの詳細については、11 章を参照）、次に再開されたタスクをチェックし（再開可能なタスクについては、6 章を参照）、次に FIFO でキューに登録されたタスクをチェックし（キューに登録されたタスクについては、11 章を参照）、最後に別のスレッドのローカルデックからタスクをスチールしようとします。

この内部ループは、タスクが見つかるか、スレッドがアリーナを離れると判断するまで、これらのステップを繰り返します。ワーカーは、アリーナの割り当てが減少した場合、またはしばらくの間ワークを探そうとしたが、有効な次のワークを見つけることができなかった場合、アリーナを離れることを決定します。スレッドがワークを見つけるために行う正確な時間または試行回数は、ライブラリーの実装次第です。図 10-14 の外側のループは、最初のタスクに関連するワークがすべて完了するまで、マスタースレッドがアリーナから離れるのを防ぐだけです。

図 10-14 の関数を呼び出すタスク・ディスパッチャー全体の簡略化された説明を図 10-15 に示します。

```
ディスパッチャー:
  if  is master
    t_next = starting task
  else
    t_next = try_to_get_next_task()
  end if

  while  t_next is valid

    do  // 最も内側の実行およびバイパスループ
       t = t_next
       if  t のタスク・グループ・コンテキストがキャンセルされました
          t_next = t->cancel()
       else
          t_next = t->execute()
       end if
       t_crit = try to get critical task
       if  t_crit is valid
          spawn t_next
          t_next = t_crit
       end if
    while  t_next is a valid task  // バイパスループの終わり

    if  マスターはここでワークを完了
      break
    else  ワーカーであれば、アリーナの割り当てが変わったので完了
      break
    end if

    t_next = try_to_get_next_task()

  end while

  // 有効な t_next が見つかりませんでした

  if  ワーカースレッド
     return  自身をグローバル・スレッド・プールに追加
  else  マスタースレッド
     return
  end if
```

図 10-15. TBB タスク・ディスパッチ・ループの近似擬似コード

マスタースレッドは実行するタスクを含むアリーナに参加するため、ディスパッチャーに入った時点で最初のタスクを持っています。アリーナに参加するワーカーは、実行する最初のタスクを見つけるため `try_to_get_next_task()` を呼び出します。図 10-15 に示すように、スレッドは次に外側のループに入り、さらにワークを実行できる限り、また実行する必要がある限り、スレッドをアリーナ内に保持します。次に、「最も内側の実行およびバイパスループ」に入ります。そこで、スレッドは、`task_group_context` の状態に応じて、識別した `t_next` を実行するかキャンセルします。キャンセル については 9 章で詳しく説明します。

`t->execute()` の呼び出しは、次のタスクを返す場合があります。これは、*スケジューラー・バイパス*と呼ばれる手法です。これはディスパッチャーの選択ロジックを短絡するため、タスクが次のタスクを供給する最も速い方法です。ただし、タスクが次のタスクを返す場合でも、スレッドは最初に、代わりに実行すべき重要なタスクがあるか確認します。重要なタスクがあれば、`execute` 呼び出しから返されたタスクをスポーンし、代わりに重要なタスクを実行します。前述のように、重要なタスクは TBB 機能の実装で使用され、アリーナに直接送信するものではありません。スレッドは、重要なタスクがなくなるか、実行の呼び出しが次のタスクを返さなくなるまで、最も内側のループを繰り返します。最も内側のループを離れるときに、それがマスタースレッドであり、そのワークが完了した場合、またはそれがワーカースレッドであり、アリーナの割り当てが削減された場合、スレッドはアリーナを離れます。

スレッドがアリーナを離れない場合は、`try_to_get_next_task()` を再度呼び出して、実行するワークを探します。`try_to_get_next_task` が有効な次のタスクを見つけられずに戻った場合、外側の `while` ループは終了し、スレッドはアリーナを離れます。

図 10-14 と 10-15 で説明したディスパッチ・ループは、図 10-13 で説明した単純なモデルとはいくつかの点で異なります。まず、TBB 実装では、他のすべての種類のワークよりも優先される重要なタスクを挿入できます。2 番目に、ワーカースレッドがアリーナを離れる条件があります。これは、ワーカースレッドを、優先度の高い、またはより必要性の高い別のアリーナに移行できるようにするため、または、単にそのアリーナで実行するワークがなくなったためです。3 番目に、アフィニティー化されたタスク (11 章で説明)、キューに入れられたタスク (11 章で説明)、再開可能なタスク (6 章で説明) など、ワークスチールによって分散されない他の種類のタスクがあります。本書を通して紹介されている他の種類のタスクについても説明します。

TBB と他のモデル、フレームワーク、ライブラリーとの相互運用性

TBB を共通の CPU スレッドレイヤーとして使用することで実現される相互運用性

これまでは、TBB がそれ自体でうまく構成できる理由について説明しました。開発者は、TBB アルゴリズムと TBB を使用するライブラリーを組み合わせて、TBB を利用して適切な処理を行うことで、良好なパフォーマンスを維持できます。この自己構成性の影響は、TBB が他のライブラリーの実装でスレッドレイヤーとして使用される場合、さらに高まります。

例えば、oneDPL、oneMKL、oneDNN、oneDAL などの多くの oneAPI 数学ライブラリーと AI ライブラリーは、CPU 上でワークを起動するときにデフォルトで oneTBB を使用するか、CPU バックエンドとして oneTBB を使用するオプションがあります。同様に、インテルの SYCL 実装の SYCL CPU デバイスは、CPU 用の SYCL 並列構造を実装するのに oneTBB を使用します。インテル OpenVINO 推論フレームワークは、CPU 上のワークをスケジュールするため oneTBB を使用します。oneAPI およびインテル・ライブラリー以外にも、サードパーティー製のライブラリーやフレームワークの中には、同様に TBB をデフォルト、または少なくともオプションのバックエンドとして選択しているものもあります。例えば、VFX リファレンス・プラットフォームでは、共通のスレッドレイヤーとして TBB を採用しています。Nvidia の Thrust ライブラリーには TBB バックエンドがあります。

そのため、これらのライブラリーが TBB を使用するように構成されている場合は、それらすべては互いにうまく組み合わされることが分かっているので、明示的な TBB コードと組み合わせて使用できます。

この構成の可能性への要請こそが、oneAPI ツールキットと Unified Acceleration（UXL）Foundation の両方が、oneTBB を共通のバックエンドとして推進することになった理由です。他のフレームワーク、ライブラリー、取り組みでも、同様の理由で同じ選択が行われています。

TBB と他の CPU スレッドモデル間との相互運用性

この章で説明する TBB の構成の可能性プロパティーは、TBB コードを CPU スレッドの他のモデルと組み合わせる場合にも役立ちます。図 10-1 で紹介した構成の種類をもう一度考えてみましょう。

TBB が別のモデルのスレッド内でネストされている場合、TBB を使用するとオーバーサブスクリプションが発生しますが、そのオーバーサブスクリプションは制限されます。TBB の `parallel_for` ループを 1,000 個のネイティブ C++ スレッド内で同時に実行すると、グローバル・スレッド・プール内のスレッド数が限られているため、TBB ライブラリーは限られた数のワーカースレッドのみを追加します。つまり、これらの 1,000 個のスレッドが 8 個のコアを持つプラットフォームで実行される場合、`1,000 + 1,000 × 7 = 8,000` 個のワーカースレッドではなく、`1,000 + 7 = 1,007` 個のスレッドのみになります。TBB はオーバーサブスクリプションを増加させますが、その程度は限定的です。

同様に、TBB 並列アルゴリズムが別の TBB 以外の並列アルゴリズムと同時に実行される場合も、TBB はグローバル・スレッド・プール内のスレッド数を最大限までしか追加しません。例えば、8 個のコアを持つプラットフォーム上で 10 個の TBB 並列ループが 10 個の OpenMP 並列ループと同時に実行される場合、TBB はこの場合も 7 個のワーカースレッドのみを追加します。そのため、10 個の OpenMP ループは `10 × 8 = 80` 個のスレッドを使用する可能性があります（OpenMP には必須の並列処理チームがあります）。ただし、TBB はその上に 7 個のみを追加し、合計 `80 + 7 = 87` 個のスレッドが生成されます。繰り返しになりますが、TBB によりオーバーサブスクリプションは発生しますが、その程度は限定的です。

最後に、TBB が別の TBB 以外の並列アルゴリズムと連続的に構成される場合、TBB はスレッドをすぐにシャットダウンしてリソースの競合を制限します。図 10-14 で説明されているように、TBB のワーカースレッドは、アリーナを離れ、グローバル・スレッド・プールに戻ってスリープ状態に入る前に、短期間だけワークを探します。TBB ワーカースレッドは、ワークを探して無限にスピンするわけではありません。

TBB が他のライブラリーを認識していない場合、これらすべての制限が発生します。しかし、図 10-9 では、各 TBB アリーナの割り当てを決定するアービトレーターがあったことを思い出すかもしれません。デフォルトでは、アービトレーターのポリシーは、TBB アリーナのみを管理する、優先度を考慮した比例ポリシーです。ただし、アービトレーターは TBB 以外のランタイムを認識できます。oneAPI ツールキットには（本書の執筆時点では）実験的なスレッド・コンポーザビリティー・マネージャー（TCM）があり、これを oneAPI 並列ランタイム全体のアービトレーターとして使用することで、アービトレーターが oneTBB、OpenMP、OpenCL、および SYCL で使用される CPU スレッドを認識できるようになります。将来的には、このアービトレーターは公開機能となり、追加のモデルが調停に参加できるようになる可能性があります。

TBB スレッド化とベクトル化の相互運用性

CPU スレッド以外にも、ベクトル化など、他の形式の並列処理があります。図 10-16 は、`std::execution` の標準実行ポリシーとしてサポートされている、多くの最新の CPU でサポートされる並列処理の種類の類似性を示しています。

図 10-16 では、高速道路が CPU の並列処理の例えとして使用されており、車はワークを表し、車内の人はワークに適用された実行リソースを表し、高速道路の車線はハードウェア・スレッドを表しています。

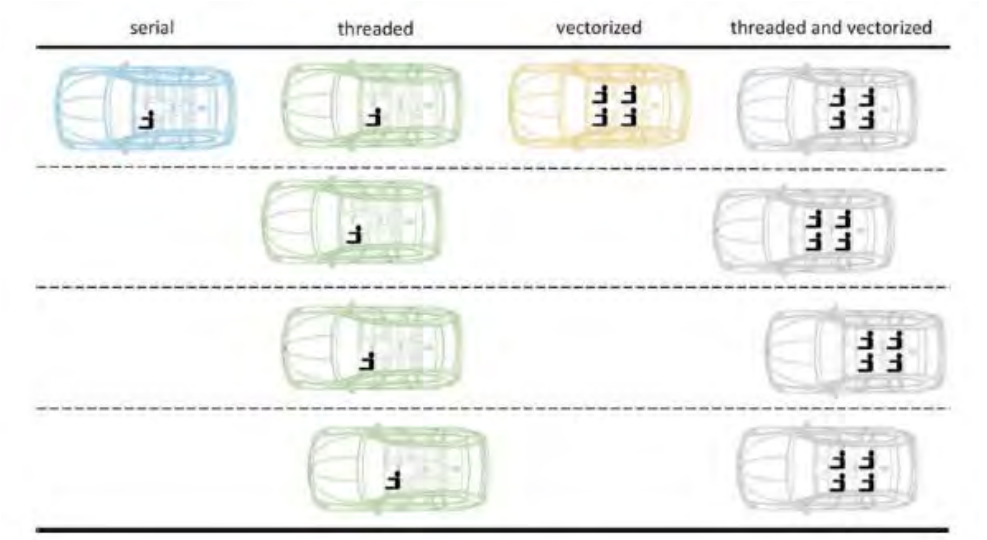


図 10-16.　CPU におけるさまざまな種類の並列処理。図の左から右の順に、C++ 標準テンプレート・ライブラリー（STL）の実行ポリシーに対応しています: `std::execution::seq`、`std::execution::par`、`std::execution::unseq`、`std::execution::par_unseq`

図 10-16 では、シリアル実行では 1 つのコアのみが使用され、並列実行ではそのコア上の SIMD/ベクトルユニットは使用されません。スレッド実行では複数のコア（つまり高速道路の車線）が使用されますが、SIMD 実行の利点は活用されません。
スレッド化およびベクトル化された実行では、すべての並列リソース、つまりコアと SIMD ユニットが使用されます。

TBB ライブラリーは、スレッドにより並列処理を追加します。したがって、図 10-16 の例を使用すると、ドライバーのいる車がすべての車線に配置されます。しかし、どうすればすべての座席に追加の人を座らせることができるでしょう？ 幸いなことに、TBB はベクトル化とうまく連携するため、TBB アルゴリズムでベクトル化されたループまたはベクトル化可能なループを使用して、すべての SIMD ユニットを有効にすることができます。
図 10-17 は、さまざまな並列実行の例と、TBB の `parallel_for` 内でそれらを構成する方法を示しています。図 10-17 の `std::for_each` に `execute::seq` が渡される場合、通常のシリアルループとしてベクトル化なしで動作します。`std::execution::unseq` がある呼び出しではベクトル化が追加される可能性があります。実行ポリシーは常に、そのタイプの並列処理を適用するのが正当であるか実装に伝えるヒントにすぎません。

関数 pfor_unseq_f では、TBB 並列アルゴリズムの本体内にベクトル化を追加するのは、unseq ポリシーを使用して blocked_range を反復処理するのと同じくらい簡単であることが分かります。

```
void  serial_f(int  N, v_type* v)
    { std::for_each(std::execution::seq, v, v + N,
        [](v_type& e) { e = f(e); });
}

void  pfor_f(int  N, v_type* v)
    { tbb::parallel_for(tbb::blocked_range<v_type*>(v, v + N),
        [=](const  tbb::blocked_range<v_type*>& r)
            { std::for_each(std::execution::seq, r.begin(), r.end(),
                [](v_type& e) { e = f(e); });
        }, tbb::static_partitioner());
}

void  unseq_f(int  N, v_type* v)
    { std::for_each(std::execution::unseq, v, v + N,
        [](v_type& v) { v = f(v); });
}

void  pfor_unseq_f(int  N, v_type* v)
    { tbb::parallel_for(tbb::blocked_range<v_type*>(v, v + N),
        [=](const  tbb::blocked_range<v_type*>& r) {
            std::for_each(std::execution::unseq, r.begin(), r.end(),
                [](v_type& e) { e = f(e); });
        }, tbb::static_partitioner());
}
```

図 10-17. CPU におけるさまざまな種類の並列処理

TBB スレッドとアクセラレーターへのオフロード間の相互運用性

ベクトル化と同様に、TBB はアクセラレーターにオフロードを直接導入するのではなく、オフロードとシームレスに連携するように設計されています。これをサポートする主なメカニズムは、再開可能なタスク（GPU への SYCL オフロードと対話する例は 6 章にあります）とフローグラフ API の async_node（SYCL で使用されるこれらの機能の例は 5 章と 6 章に示されます）の 2 つです。どちらの機能も、非同期ワークの完了を待機する間に、TBB ワーカースレッドがブロックされるのを防ぐ非同期の導入を可能にします。これらの機能がないと、GPU にワークを送信する TBB タスクは、完了するまで待機が必要となることがあります。

図 10-16 の例を拡張すると、これは、ワークが CPU 以外のデバイスにオフロードされるたびに高速道路の車線の真ん中に車を止めるようなものです。ワークは他のデバイスで加速される可能性がありますが、CPU（高速道路の車線）は完了するまでブロックされます。

TBB スレッドと将来の C++ 機能との相互運用性

将来を予測することは困難ですが、`std::execution` の追加機能など、TBB によって提供される機能と相互作用したり、重複する可能性のある機能が C++26 に組み込まれる予定です（`https://en.cppreference.com/w/cpp/experimental/execution` を参照）。

C++26 向けに提案された実行ライブラリーは、汎用実行リソースで非同期実行を管理するフレームワークを提供します。このフレームワークでは、提案では「実行リソースにワークをスケジュールする方針を表す軽量なハンドル」と説明されているスケジューラーが導入されています。TBB は、CPU スレッドのプールにワークを割り当てるスケジューラーとして提供できる既存のライブラリーの例です。

プロセスベースの並列処理との相互運用性

TBB ライブラリーは、OS 呼び出しや `numactl` などのツールによって設定されたプロセス・アフィニティー・マスクを尊重します。MPI（メッセージ・パッシング・インターフェイス）ライブラリー、またはプロセスを使用して並列処理を実装する Python ライブラリーとともに使用すると、TBB ライブラリーは設定に従ってマシンを解釈します。例えば、TBB がグローバル・スレッド・プールのデフォルトスレッド数を計算する場合、作成されるワーカーの数は、ベースとなる物理マシンではなく、設定に基づいて決定されます。

TBB の構成の可能性と相互運用性のまとめ

図 10-18 は、TBB をそれ自体、他の並列プログラミング手法、および予想される将来の C++ 機能と併用した場合の構成の可能性と相互運用性の概要を示しています。

構成の種類	説明
TBB をシリアルに、同時に、またはネストに組み合わせた場合。	TBB の設計では、グローバル・スレッド・プール、アリーナ、および過度なサブスクリプションを防ぎ、TBB コードを安心して組み合わせることができるアービトレーターを使用します。
TBB とその他の CPU スレッドモデル	設計上、TBB は、他のモデル内にネストされたり、TBB 以外のスレッド内で同時に使用される場合でも、オーバーサブスクリプションを制限する良きコンポーネントです。
TBB とその他の CPU スレッドモデル（ジョイント・アービトレーター付き）	TBB の設計により、スレッドモデル間の共通の調停が可能になります。oneAPI ツールキットには現在、実験的な調停ツールであるスレッド・コンポーザビリティー・マネージャー（TCM）があり、これを TBB、OpenMP、OpenCL、SYCL で使用して CPU のオーバーサブスクリプションを制限できます。
TBB および TBB ベースのライブラリーとフレームワーク	TBB を基盤エンジンとして使用する並列ライブラリーとフレームワークは、相互に、また明示的な TBB コードと組み合わせても問題なく動作します。例としては、oneDPL、oneDNN、oneMKL などの oneAPI / UXL ライブラリーや、OpenVINO、VFX プラットフォームのライブラリーがあります。
TBB と SIMD / ベクトル化	SIMD / ベクトル化を TBB コード内にクリーンにネストして、パフォーマンスをさらに向上できます。
TBB とアクセラレーターへのオフロード	再開可能なタスクや `async_node` などの TBB 機能により、アクセラレーターへの効率的な非同期オフロードが可能です。
TBB と将来の C++ 機能	TBB の開発チームは C++ 標準の方向性を十分に認識しており、新しい機能で TBB がうまく機能するように取り組んでいます。TBB は、デフォルトの C++ 並列スケジューラーを TBB に置き換える方法を含め、今後の C++26 実行機能に適合したスケジューラーを提供することが期待されています。
TBB とプロセスベースの並列処理	TBB ライブラリーは、`numactl` などのツールによって設定されたアフィニティー・マスクの OS 設定に従ってマシントポロジーを表示します。したがって、TBB はそれらの設定に合わせてデフォルトの同時実行性を制限します。

図 10-18. TBB ライブラリーの構成可能性と相互運用性の概要

まとめ

この章では、TBB 内の構成可能性の概念を詳しく説明し、ネスト実行、同時実行、シリアル実行に分類しました。私たちは、スケーラビリティーを強化し、開発者が利用の過不足を心配することなく並列処理を公開できるように設計哲学として、TBB の構成の可能性を確立しました。

TBB のネストされた並列処理へのアプローチと OpenMP のアプローチを比較し、TBB の緩和されたシーケンシャル・セマンティクスとタスクベースのモデルによって、ネストされた並列処理を無効にすることなく効率的な実行が可能になることを示しました。この区別は C++ プログラミングでは重要ですが、単純なループ指向の C および Fortran コードではそれほど重要ではありません。

構成の可能性はバイナリー属性ではなく、プログラミング・モデルの相互作用に応じて変わるものであることを明確にしました。この章では、ネストされた並行構成の例を通じてこれを説明し、TBB がネストされた並列処理を効果的に管理してスループットを最適化する方法を示しました。並列アルゴリズムが実行時にオーバーラップしてパフォーマンスをさらに向上できる並行構成についても説明しました。

TBB は構成可能性をサポートするモデルを提供します。それを理解し、失わないようにするのは私たち次第です。

この章では、構成の可能性の概念を説明しただけでなく、常に構成可能なコードを自身で作成したいという強い願望を定着できたら幸いです。

11 章　パフォーマンス・チューニング

本書の前半では、アルゴリズム、フローグラフ、タスクグループなど、TBB で並列処理を表現するさまざまな方法について説明しました。これらの章では、コア数やメモリーサイズに関係なく、TBB がさまざまなシステムにわたって強力なパフォーマンスを実現できることを説明しました。10 章では、TBB のグローバル・スレッド・プール、アービトレーター、アリーナ、タスク・スケジューラーの組み合わせが、TBB の構成可能性にどのように貢献し、一般的な状況に効果的に適応できることを説明しました。

ただし、並列プログラミングは本質的にパフォーマンスに関するものであり、パフォーマンスを考慮しなければ、並列処理を追加する複雑性を気にする必要はないでしょう。TBB ランタイムのパフォーマンスを向上させるヒントや追加の制約を提供して、パフォーマンスを向上させたい場合があります。これらのパフォーマンス制御の可用性は、TBB と C++ 並列アルゴリズムなど他の高レベル・アルゴリズム・ライブラリーとの主な違いの 1 つです。ただし、パフォーマンス・チューニングの欠点は、追加のヒントや制約を提供すると、特定のプラットフォーム向けにチューニングすることでライブラリーの選択が制限されるため、アプリケーションの構成性と移植性が低下する可能性があることです。
したがって、この章で説明した機能を使用する場合は、かなり注意する必要があります。

この章では、チューニングの 2 つの大まかなカテゴリーについて説明します:

（1）　ライブラリーが使用するスレッド数とハードウェア・リソースを制限します。

（2）　TBB アルゴリズムの粒度とアフィニティーを最適化します。

まず、制約について、`global_control`、`task_arena`、ハードウェア対応スケジュール、`task_scheduler_observer` の 4 つのトピックで説明します。次に、粒度と局所性、決定論、パイプラインの調整を扱う 3 つのセクションでアルゴリズムの最適化について説明します。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_11

TBB が使用するリソースを制御

パフォーマンスに影響を与える重要な制御の 1 つは、タスクの実行にどのような種類の計算リソースがどれだけ使用されるか制御することです。

TBB は、デフォルト設定の決定にいくつかの仮定を行います。デフォルトでは次のことが想定されています:

1. メモリー階層やコアの機能に関係なく、アプリケーションで使用できるすべての論理 CPU コアを使用したいと考えています。
2. TBB が作成するすべてのワーカースレッドにデフォルトのスタックサイズを使用します。
3. プラットフォームの論理コアをオーバーサブスクライブするのではなく、論理コアごとに単一のソフトウェア・スレッドでタスクを処理するようにします。
4. すべてのタスクの優先度は同じです。

これにより、TBB ライブラリーとアルゴリズムで使用されるデフォルトで、通常は良好なパフォーマンスが得られます。したがって、一般的なガイドラインとしては、どうしても必要である場合を除き、これらのデフォルトを変更しないことが最善です。

図 11-1 はこれらのデフォルトをオーバーライドできる機能をまとめたものです。アプリケーション開発者としては、上に挙げた仮定の一部がユースケースには当てはまらないことを知っているかもしれません。10 章で説明したように、TBB ライブラリーは、グローバル・スレッド・プール、調停ポリシー、デフォルトの暗黙的なタスク アリーナ、およびワークスチール・タスク・スケジューラー間の相互作用により、構成可能なパフォーマンスを提供します。この章で説明した機能を使用する場合、バランスのとれた相互作用に干渉することになるので、注意が必要です。

これらの注意事項を念頭に置いて、この節の残りの部分では、図 11-1 の機能についてさらに詳しく説明します。これにより、デフォルトよりも優れたパフォーマンスを実現したい場合、これらの機能を使用できます。

機能	説明
`tbb::global_control`	アプリケーション・レベルで使用することを目的としたこのオブジェクトのインスタンスは、許容される最大並列性、作成されたワーカースレッドのスタックサイズ、および終了動作を制御できます。
`tbb::task_arena`	`task_arena` のインスタンスは、新しい TBB アリーナを作成したり、既存のアリーナへの軽量ハンドルとして機能します。アリーナは、異なる数のスロット、マスタースレッド用に予約された異なる数のスロットで作成できます。また、スレッドを NUMA ノードまたは E-core や P-core などの特定のプロセッサー・タイプに固定して作成することもできます。
`tbb::info`	`tbb::info` 名前空間は、NUMA ノードの数、使用可能なコアの種類、プラットフォームまたは特定の NUMA ノード上のデフォルトの同時実行性を照会する関数を提供します。これらの照会結果は、明示的に初期化された `task_arena` インスタンスを制限するのに使用できます。
`tbb::task_scheduler_observer`	`task_scheduler_observer` のインスタンスは、スレッドが特定の TBB アリーナに参加したり、離脱するときに、ユーザー定義のコードを実行します。例えば、ユーザー定義コードを使用すると、スレッドの配置を正確に制御できます。

図 11-1. TBB が使用する実行リソースを制御するのに使用される TBB の機能

図 11-2 は 10 章の図を改めて紹介するものです。図 11-2 は、グローバル・スレッド・プール内のスレッドのセットが限られていることを示します。これらのスレッドは、アービトレーターのポリシーと、アリーナ内のスロット数とタイプに基づいて、TBB アリーナに割り当てられます。

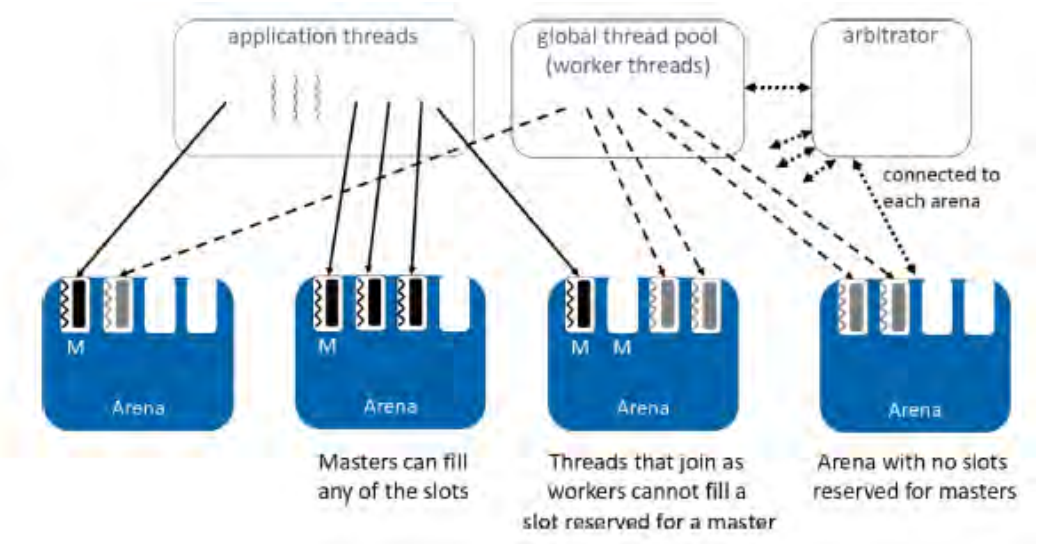


図 11-2. 複数の TBB アリーナを使用する複雑な例

図 11-1 にリストされている機能を使用すると、グローバル・スレッド・プールで使用できる最大スレッド数、アリーナ内のスロット数、マスタースレッド用に予約されているスロット数、およびアリーナの優先順位を変更できるほか、アリーナがスレッドをプラットフォーム上の論理コアに割り当てる方法に関連するハードウェア対応の設定を行うこともできます。

global_control を使用してアプリケーション・レベルのプロパティーを設定

TBB に設定できるアプリケーション・レベルのプロパティーは 3 つあります。1 つ目は、タスクをアクティブに実行できる最大スレッド数です。これは、グローバル・スレッド・プールのサイズを変更するものと考えることができます。2 番目は、ワーカースレッドのスタックサイズです。3 番目は、エラー処理に関するものです。

これらのアプリケーション・レベルのプロパティーを変更するには、`global_control` オブジェクトを作成します。`global_control` クラスのクラス定義を図 11-3 に示します。`global_control` には、`max_allowed_parallelism`、`thread_stack_size`、`terminate_on_exception` の 3 つのパラメーターが定義されています。

各 `global_control` オブジェクトは 1 つのパラメーターのみを変更できるため、複数のパラメーターを設定するには複数のインスタンスを作成する必要があります。`global_control` オブジェクトの影響は、オブジェクトのライフタイムが終了すると終了します。同じパラメーターを設定する `global_control` オブジェクトが複数存在する場合、競合する要求はパラメーター固有の*選択規則*によって解決されます。

```
namespace tbb {
    class global_control {
    public:
        enum parameter {
            max_allowed_parallelism,
            thread_stack_size,
            terminate_on_exception
        };

        global_control(parameter p, size_t value);
        ~global_control();

        static size_t active_value(parameter param);
    };
} // namespace tbb
```

図 11-3. [scheduler. global_control] で説明されている global_control クラスのインターフェイス

global_control インスタンス（図 11-3）はプロセス全体に影響を及ぼします。これは原始的なツールなので、慎重にアプリケーションのトップレベルからのみ使用します。例えば、大部分の開発者は、サードパーティーのライブラリーがグローバル・スレッド・プール内のスレッド数を設定し、アプリケーションの他のすべての部分に影響することを想定していません。それは大変なことです。

max_allowed_parallelism を使用する

グローバル max_allowed_parallelism パラメーターは、同時にアクティブにタスクを実行できるワーカースレッド数の*上限*を設定します。TBB ではマススレッドが 1 つあると想定されるため、実際のワーカー数は max_allowed_parallelism - 1 に制限され、マススレッドが参加する余地が残されます。

max_allowed_parallelism によって設定される上限は、並列実行のあらゆる面に影響を与える可能性があります。この上限を下げると、利用可能なハードウェアを最大限に活用できなくなる可能性があります。また、上限を上げると、TBB ライブラリーが同じ論理コアに複数のソフトウェア・スレッドを割り当てるため、ハードウェア・リソースをオーバーサブスクライブする可能性があります。図 11-2 に示すアービトレーターは、比例的で優先度を考慮したポリシーに基づいてスレッドをアリーナに分配します。そのため、競合するアリーナがある場合、max_allowed_parallelism は各アリーナに配置できるスレッドに影響します。

このグローバル・パラメーターを変更する理由は？ これは上限であり、ライブラリーが使用するスレッド数が多くならないようにこの値を減らしたり、ライブラリーが論理コアごとに複数のスレッドを使用できるようにこの値を増やす必要があるかもしれません。スレッドごとに大量のメモリーを割り当てがあることが分かっている場合、`max_allowed_parallelism` を減らすことを選択する場合があります。 おそらく、コア数の多いシステムではメモリーが不足します。あるいは、システム上のすべての論理コアを利用するには、計算スケールが不足することが分かっているかもしれません。値を制限することで、オーバーヘッドが増加するだけであるとわかっているスレッド管理のオーバーヘッドを回避できます。しかし、繰り返しますが、これはグローバル設定であることを忘れないでください。そのため、コードの他の部分への影響を完全に理解していない場合は、不用意にグローバル値を変更すべきではありません。

同様に、`max_allowed_parallelism` を増やすことが有効なケースもあるかもしれません。例えば、タスクが何らかの I/O を実行し、オーバーサブスクリプションを容認すると、結果の待機に費やされたシステム時間が隠匿される可能性があります。

図 11-4 は、`global_control` オブジェクトを作成し、`parallel_for` を実行する簡単な例を示しています。この例では、`max_allowed_parallelism` の値は `p` であり、引数として渡されます。プラットフォーム上のデフォルトの同時実行性は `default_P` に保存されます。

```
#include <tbb/tbb.h>

const int default_P = tbb::info::default_concurrency();
void noteParticipation(); /* 参加するベクトルの記録情報 */
void dumpParticipation(int p); /* 参加するベクトルを表示 */

void doWork(double seconds) {
  noteParticipation();
  tbb::tick_count t0 = tbb::tick_count::now();
  while ((tbb::tick_count::now() - t0).seconds() < seconds);
}

void arenaGlobalControlImplicitArena(int p) {
  tbb::global_control gc(tbb::global_control::max_allowed_parallelism, p);
  tbb::parallel_for(0,
                    10*tbb::info::default_concurrency(),
                    [](int) { doWork(0.01); });
}

int main() {
  arenaGlobalControlImplicitArena(default_P);
  dumpParticipation(default_P);
  arenaGlobalControlImplicitArena(default_P/2);
  dumpParticipation(default_P/2);
  arenaGlobalControlImplicitArena(2*default_P);
  dumpParticipation(2*default_P);
  return 0;
}
```

図 11-4. 関数 arenaGlobalControlImplicitArena では、global_control オブジェクトを使用して max_allowed_parallelism を設定し、次に tbb::parallel_for が呼び出されます。noteParticipation 関数はスレッドローカルの合計をインクリメントします。この合計は後で dumpParticipation によって表示される出力ベクトルを作成するのに使用されます。サンプルコード: performance_tuning/global_control_and_implicit_arena.cpp

この例では、暗黙のアリーナのタスク実行に参加する各スレッドが、parallel_for ループを何回繰り返したか示すベクトルが表示されます。メイン関数では、p = default_P、p = default_P/2、および p = 2*default_P で arenaGlobalControlImplicitArena が呼び出されます。ベクトルを保存および表示するコードを図 11-5 に示します。

```
#include  <atomic>
#include  <iostream>
#include  <vector>

std::atomic<int> next_tid;
tbb::enumerable_thread_specific<int> my_tid(-1);
std::vector<std::atomic<int>> tid_participation(2*default_P);

void  noteParticipation() {
  auto& t  = my_tid.local();
  if  (t  == -1) {
    t  = next_tid++;
  }
  ++tid_participation[t];
}

void  clearParticipation() {
  next_tid  =  0;
  my_tid.clear();
  for  (auto& p  : tid_participation)
    p  =  0;
}

void  dumpParticipation(int  p) {
  int  sum  = tid_participation[0];
  std::cout  <<  "["  <<  tid_participation[0];
  for  (int  i  = 1; i  < std::min(p, default_P); ++i) {
    sum  += tid_participation[i];
    std::cout  <<  ", "  <<  tid_participation[i];
  }
  for  (int  i  = p; i  < default_P; ++i)
    std::cout  <<  ", -";
  std::cout  <<  "]\n"
             <<  "sum == "  <<  sum     <<  "\n"
             <<  "expected sum "  << 10*default_P  << "\n\n";
  clearParticipation();
}
```

図 11-5. 図 11-4 のパーティシペーション・ベクトルを作成して表示する関数。サンプルコード: performance_tuning/global_control_and_implicit_arena.cpp

図 11-4 の例を 8 論理コアを持つシステムで実行し、`arenaGlobalControlImplicitArena` を `p=8` で呼び出すと、スレッドの割り当ては図 11-6 のようになります。

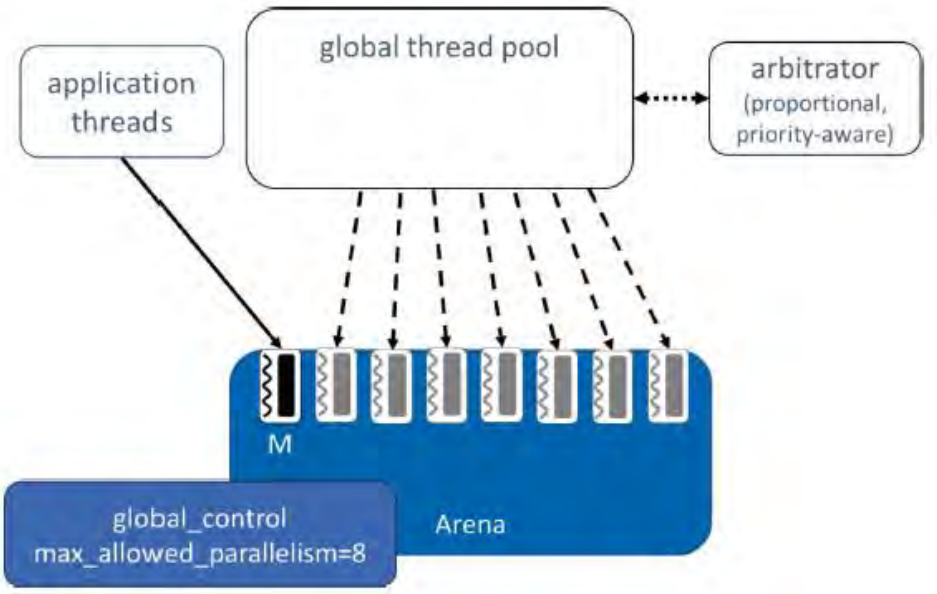


図 11-6. サンプルコード: performance_tuning/global_control_and_implicit_arena.cpp を実行する際に、8 論理コアを持つシステムで max_allowed_parallelism=8 とした場合のスレッド割り当ての例

図 11-4 の例を `p = default_P` で実行すると、記録される出力には、8 つのスレッドのそれぞれが 80 回の反復のうち 10 回を実行することが示されます（ただし、ワークスチールでは正確な分散は保証されません）:

```
[10, 10, 10, 10, 10, 10, 10, 10]
sum == 80
expected sum 80
```

`max_allowed_parallelism` を設定する際に重要なことは、これがアリーナ内のスロット数に直接影響しないということです。暗黙に作成されたすべてのアリーナには、マスタースレッド用に予約された 1 つのスロットと、ワーカーに使用可能な `P-1` のスロットが残ります。したがって、`arenaGlobalControlImplicitArena` が `p = default_P/2` で呼び出されると、スレッドの割り当ては図 11-7 のようになります。

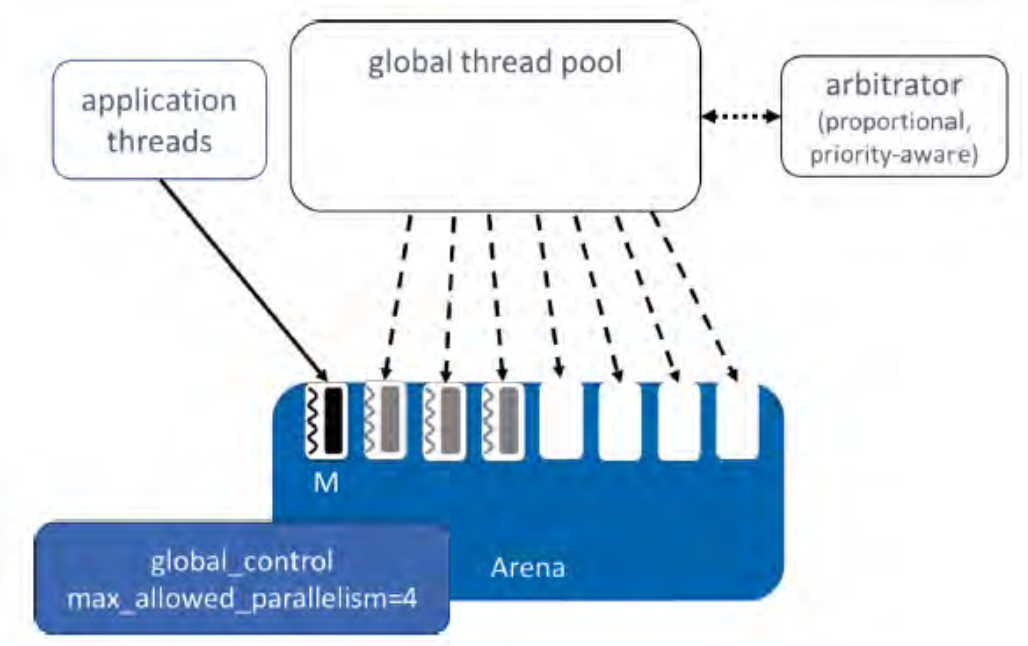


図 11-7. サンプルコード: performance_tuning/global_control_and_implicit_arena.cpp を実行する際に、8 論理コアを持つシステムで max_allowed_parallelism=4 とした場合のスレッド割り当ての例

実行すると、拡張された例の出力は次のようになります。

```
[20, 20, 20, 20, -, -, -, -]
sum == 80
expected sum 80
```

max_allowed_parallelism をスロット数より小さい値に設定すると、デフォルトのアリーナのスロットを埋めるのに十分なスレッドがなくなりますが、スロットは依然として存在します。それは大丈夫です。最後に、p = 2*default_P で arenaGlobalControlImplicitArena を呼び出しますが、暗黙のアリーナをまだ使用している場合、図 11-8 のような割り当てになります。

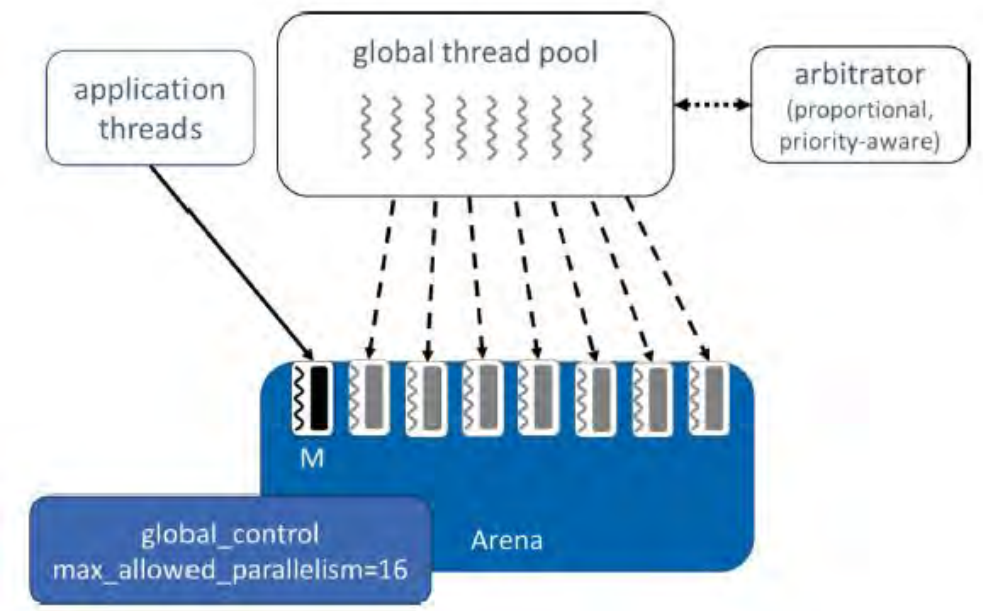


図 11-8. サンプルコード: performance_tuning/global_control_and_implicit_arena.cpp を実行する際に、8 論理コアを持つシステムで max_allowed_parallelism=16 とした場合のスレッド割り当ての例

図 11-8 から予想されるとおり、出力には、max_allowed_parallelism 値によってアリーナに分散できるスレッド数が増え、暗黙のアリーナには default_P=8 スロットしかないため、アリーナに参加できるスレッドは依然として 8 つだけであることが示されています。TBB がアリーナに供給するスレッドを選択できる場合、そのアリーナを最近離れたスレッドが TBB によってそのアリーナに戻ることを選択される可能性が高くなることに注意してください。p = 16 の出力は次のようになります。

```
[10, 10, 10, 10, 10, 10, 10, 10]
sum == 80
expected sum 80
```

しかし、心配ありません。望むだけ多くの、または少ないスロットを保持できる明示的なアリーナを作成する方法については、すぐに説明します。その前に、global_control が複数ある例を考えてみます。

図 11-9 では、arenaGlobalControlImplicitArena が 2 つの異なる C++ 標準スレッドから並列に呼び出されています。各スレッドで、max_allowed_parallelism を設定する競合する要求があります。

waitUntil 関数は、どちらかのスレッドが tbb::parallel_for を開始する前に、両方の tbb::global_control オブジェクトが作成され、両方のスレッドが終了する前に、どちらのオブジェクトも破棄されないことを保証します。したがって、重複する 2 つの設定の効果を確認できます。

max_allowed_parallelism コントロールは、アクティブなスレッド数に上限を設定するために使用され、これらの上限を安全に組み合わせる唯一の方法は、最小値を使用することです。もちろん、global_control インスタンスが破棄されると、TBB ライブラリーは最小値を再計算し（実際に再計算します）、可能であれば分散用にグローバルプールにスレッドを追加します。global_control パラメーターの現在のアクティブな値は、静的メンバー関数 global_control::active_value(parameter) を呼び出して取得できます。

```
#include  <thread>
#include  <tbb/tbb.h>

const  int  default_P  = tbb::info::default_concurrency();

void  waitUntil(int  N);
void  noteParticipation(int  offset);
void  dumpParticipation()
void  doWork(int  offset, double  seconds) {
  noteParticipation(offset);
  tbb::tick_count t0 = tbb::tick_count::now();
  while  ((tbb::tick_count::now() -  t0).seconds() < seconds);
}

counter_t  counter1  = 0, counter2  = 0;

void  arenaGlobalControlImplicitArena(int  p, int  offset) {
  tbb::global_control  gc(tbb::global_control::max_allowed_parallelism, p);

  waitUntil(2, counter1);
  tbb::parallel_for(0,
                    10*default_P,
                    [=](int) {
                      doWork(offset, 0.01);
                    });
  waitUntil(2, counter2);
}

void  runTwoThreads(int  p0, int  p1) {
  std::thread  t0([=]() {  arenaGlobalControlImplicitArena(p0, 1); });
  std::thread  t1([=]() {  arenaGlobalControlImplicitArena(p1, 10000); });
  t0.join();
  t1.join();
}

int  main() {
  runTwoThreads(default_P/2, default_P);
  dumpParticipation();
  return  0;
}
```

図 11-9. 競合する 2 つの global_control オブジェクトが max_allowed_parallelism を設定します。waitUntil 関数と noteParticipation 関数の実装を図 11-10 に示します。サンプルコード: performance_tuning/global_control_and_implicit_conflict.cpp

```
void noteParticipation(int offset) {
  auto& t = my_tid.local();
  if (t == -1) {
    t = next_tid++;
  }
  tid_participation[t] += offset;
}

void waitUntil(int N, counter_t& c) {
  ++c;
  while (c != N);
}
```

図 11-10. オフセットでインクリメントする更新された noteParticipation 関数と、単純な waitUntil 実装。C++20 では std::barrier と std::latch が導入され、C++20 をサポートする標準ライブラリーを使用すると waitUntil の代わりに使用できるようになりました。サンプルコード: performance_tuning/global_control_and_implicit_conflict.cpp

拡張例の出力をより分かりやすくするため、図 11-9 では、2 つの標準スレッドのそれぞれにプロファイル・カウンターを別々にインクリメントします（t0 の場合、スレッドが実行する反復ごとに 1 ずつ増加し、t1 呼び出しの場合、カウンターが 10,000 ずつ増加します）。

そこで、8 論理コアを持つシステムで、p0 = default_P/2、p1 = default_P で runTwoThreads を呼び出すと、次のような出力が表示されます。

```
[340000, 27, 60027, 70026, 330000, 0, 0, 0, 0]
sum == 800080
expected sum == 800080
```

これらの数字を分解してみましょう。まず、両方のループが 80 回の反復を実行すると、スレッドごとのカウンターの合計は 10 × 8 × 1 + 10 × 8 × 10,000 = 800,080 になると予想されますが、その通りです。次に、合計 5 つの異なるスレッドが何らかのワークを実行していることが分かります。競合する要求のルールは最小値を使用することだと述べました。そして、min(8/2, 8) – 1 = 3 のワーカースレッドと 2 つのマスタースレッドの、合計 5 つのスレッドであることが分かります。

新しいスレッドのスタックサイズを変更

他の 2 つの global_control パラメーターは、thread_stack_size と terminate_on_exception です。

名前が示すように、thread_stack_size は TBB ライブラリーによって作成されるワーカースレッドのスタックサイズに影響します。thread_stack_size の重要なポイントを以下に示します。

1. このパラメーターを設定しても、アプリケーション・スレッドはすでに作成されているため影響はありません。TBB によって作成されたワーカースレッドのみが影響を受けます。
2. 値は、ワーカースレッドが作成される前に設定する必要があります。これは通常、タスクを実行する TBB が初めて使用される前です。

thread_stack_size の global_control 設定値が複数あり、競合する場合の選択ルールは、最大値を使用することです。この場合、最大値は、スレッドが最大の要求に対応できるだけのスタックスペースを持つようにするために使用されます。また、max_allowed_parallelism と同様に、値は global_control オブジェクトのライフタイムに基づいて動的に更新されます。最も大きな値を設定したオブジェクトが破棄された場合、2 番目に大きい値が新しいアクティブ値になります。新しいワーカースレッドが作成されるたびに、現在のアクティブな値が使用されます。

例外処理の変更

最後のパラメーターは terminate_on_exception です。このパラメーターはブール値で、値は 0 または 1 であり、選択ルールは論理和です（すべての値が 0 の場合のみ false になります）。パラメーターを 1 に設定すると、例外をスローまたは再スローする条件で終了します。

task_arena を使用してリソースを制限し優先順位を設定

図 11-2 に示すように、スレッドは TBB タスクの実行に参加する前にアリーナに移動する必要があります。さらに、アリーナのスロット数を設定することで、リソースの使用を制御します。

クラス task_arena の定義を図 11-11 に示します。

```
namespace tbb {
    class task_arena {
    public:
        static const int automatic = /* 未指定 */;
        static const int not_initialized = /* 未指定 */;

        enum class priority : { /* 図 11-14 参照 */ };
        struct constraints { /* 図 11-18 参照 */ };

        task_arena(int max_concurrency = automatic,
                   unsigned reserved_for_masters = 1,
                   priority a_priority = priority::normal);
        task_arena(constraints a_constraints,
                   unsigned reserved_for_masters = 1,
                   priority a_priority = priority::normal);
        task_arena(const task_arena &s);
        explicit task_arena(tbb::attach);
        ~task_arena();

        void initialize();
        void initialize(int max_concurrency,
                        unsigned reserved_for_masters = 1,
                        priority a_priority = priority::normal);
        void initialize(constraints a_constraints,
                        unsigned reserved_for_masters = 1,
                        priority a_priority = priority::normal);
        void initialize(tbb::attach);

        void terminate();
        bool is_active() const;
        int max_concurrency() const;

        template<typename F> auto execute(F&& f) -> decltype(f());
        template<typename F> void enqueue(F&& f);

        void enqueue(task_handle&& h);
    };
} // namespace tbb
```

図 11-11. [scheduler.task_arena] で説明されている tbb::task_arena のクラス定義

図 11-2 に示すアリーナは、TBB ライブラリーによって管理される内部オブジェクトであり、スレッドがワークに参加する場所を表しています。アリーナへのハンドルとして機能する tbb::task_arena オブジェクトを作成することで、新しいアリーナを明示的に作成したり、既存のアリーナにアクセスすることができます。10 章で説明したように、アプリケーション スレッドが、明示的に作成された task_arena にタスクを生成またはキューに追加せずに並列ワークを開始すると、そのスレッドには暗黙的なアリーナが使用され、そのスレッドにまだアリーナが存在しない場合は新しいアリーナが作成されます。

前の節で説明したように、暗黙的なアリーナには常に P 個のスロットがあり、そのうち 1 スロットはマスタースレッド用に予約されているため、最大 P-1 個のワーカースレッドとマスタースレッドで占有できます。

デフォルト以外のスロット数を持つ明示的アリーナの作成

図 11-11 には、いくつかのコンストラクターが表示されています。ここでは、優先順位と制約の議論は止めておきましょう。これらについては後で説明します。

4 つのスロットを持つアリーナを作成したい場合は、アリーナを構築して 4 を渡すだけです:

```
tbb::task_areana a(4);
```

この場合、合計 4 つのスロットを持つ新しく作成されたアリーナがあり、それらのスロットの 1 つがマスタースレッド用に予約されます。2 番目の引数（reserved_for_masters）に 0 を指定すると、すべてのスロットをワーカーが利用できるようになります。

```
tbb::task_areana a(4, 0);
```

これで、合計 4 つのスロットが存在し、マスター用に予約されているスロットはなくなります。これは、アプリケーション・スレッドが参加できないという意味ではありません。これは、十分な数のワーカースレッドが利用可能である場合、すべてのスロットをワーカーで埋めることができることを意味します。しかし、ワーカースレッドが足りないか、アプリケーション・スレッドが最初にアリーナに参加しようとする可能性があります。この場合には、アプリケーション・スレッドは空きスロットを埋めることができます。

アリーナを構築するには、次の 2 つの特別な方法もあります:

```
tbb::task_arena a(tbb::attach);
tbb::task_arena b(a);
```

tbb::attach を渡すと、呼び出しスレッドが使用する内部アリーナに接続された task_arena が構築されます。呼び出しスレッドは、暗黙的なアリーナを持つマスタースレッド、明示的なアリーナに参加しているマスタースレッド、またはワーカースレッドである可能性があります。呼び出しスレッドがマスタースレッドで内部アリーナがまだ存在しない場合、このコンストラクターはデフォルトのパラメーターを使用して task_arena を作成します。

task_arena コピー・コンストラクターは、提供されたアリーナから設定をコピーしますが、同じ内部アリーナへのハンドルを表すものではありません。同じ設定で新しいものを作成します。

明示的な task_arena へワークを送信

task_arena には、ワークを送信する 2 つのインターフェイス（実行とエンキュー）があります。どちらも関数オブジェクトを引数として受け取り、アリーナ内で関数オブジェクトを実行するタスクを作成します。

スレッドが task_arena::execute を呼び出すと、アリーナに参加してタスクを実行します。アリーナに参加できない場合（占有するスロットがないため）、タスクをアリーナのキューに入れて、タスクが完了するまで待機します。スレッドがすぐにスロットを見つけられない場合は、後でスロットが利用可能になったときに参加する可能性があります。ただし、いずれの場合でも、実行からは送信されたワークが完了したときにのみ戻ります。

対照的に、スレッドが task_arena::enqueue を呼び出すと、アリーナに参加せずにタスクをアリーナのキューに登録し、その後、タスクが完了する前にすぐに戻ります。

図 11-4 と 11-5 の例を再度確認してみましょう。ただし、今回は明示的なタスクアリーナを使用して、ワークに参加できるスレッド数を厳密に制御します。図 11-12 は、global_control オブジェクトのインスタンスが 1 つと、2*default_P スロットで作成された task_arena がある例を示しています。p=default_P/2 または p=default_P で実行すると、スレッド数はスロット数以下になるため、利用可能なすべてのスレッドが参加できます。

```
#include <tbb/tbb.h>

const int default_P = tbb::info::default_concurrency();
void doWork(double seconds);

void arenaGlobalControlExplicitArena(int p) {
  tbb::global_control gc(tbb::global_control::max_allowed_parallelism, p);

  tbb::task_arena a{2*tbb::info::default_concurrency()};

  a.execute([]() {
     tbb::parallel_for(0,
                       10*tbb::info::default_concurrency(),
                       [](int) { doWork(0.01); });
  });
}

int main() {
  arenaGlobalControlExplicitArena(default_P);
  dumpParticipation(default_P);
  arenaGlobalControlExplicitArena(default_P/2);
  dumpParticipation(default_P/2);
  arenaGlobalControlExplicitArena(2*default_P);
  dumpParticipation(2*default_P);
  return 0;
}
```

図 11-12. global_control と tbb::task_arena を使用する例。サンプルコード: performance_tuning/performance_tuning/global_control_and_explicit_arena.cpp

図 11-12 で興味深いのは、p=2*default_P の場合です。これを 8 つのハードウェア・スレッドを持つシステムで実行すると、次のようになります。

```
[7, 9, 9, 6, 5, 5, 9, 5, 3, 4, 3, 3, 4, 4, 3, 1]
sum == 80
expected sum 80
```

明示的なアリーナ a にスロットが追加され、すべてのスレッドが参加できることが分かります。スレッド数が多くなり、スレッドが明示的なアリーナに移動する必要性が高まると、ワークがスレッド間で均等に分散されないことも分かります。この不均等な分布は、これが小さな例であることも起因しています。同じアリーナが繰り返し使用される通常の計算集約型のワークロードでは、スレッドがアリーナに留まり、新しいワークが到着するとすぐに実行できる状態である可能性が高くなります。

2 つの global_control インスタンス間で競合が発生した、図 11-9 の例をもう一度見てみましょう。図 11-13 に更新された例を示します。

```
void  arenaGlobalControlExplicitArena(int  p, int  offset) {
  tbb::global_control gc(tbb::global_control::max_allowed_parallelism, p);

  // GC の有効期間を強制的に重複させるため waitUntil を使用
  waitUntil(2, counter1);
  tbb::task_arena a{2*tbb::info::default_concurrency()};

  a.execute([=]() {
     tbb::parallel_for(0,
                       10*tbb::info::default_concurrency(),
                       [=](int) { doWork(offset, 0.01); });
  });

  // 両方の GC が完了するまで、どちらかの GC が破壊されるのを防ぐ
  waitUntil(2, counter2);
}

void  runTwoThreads(int  p0, int  p1) {
  std::thread  t0([=]() { arenaGlobalControlExplicitArena(p0, 1); });
  std::thread  t1([=]() { arenaGlobalControlExplicitArena(p1, 10000); });
  t0.join();
  t1.join();
}

int  main() {
  runTwoThreads(default_P/2, 2*default_P);
  dumpParticipation();
  return  0;
}
```

図 11-13. 競合する 2 つの global_control オブジェクトは、多くのスロットを持つ明示的なアリーナが作成される前に、max_allowed_parallelism を設定します。サンプルコード: performance_tuning/global_control_and_implicit_conflict.cpp

8 つのハードウェア・スレッドを持つシステムで、p0=default_P/2、p1=2*default_P として図 11-13 を実行すると、次のようになります。

```
[27, 340000, 330000, 60027, 70026, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sum == 800080
expected sum == 800080
```

ここでも、2 つのアリーナに多数の空きスロットがあるにもかかわらず、参加しているスレッドは 5 つだけであることが分かります。p1 = 16 の場合、実行に参加したスレッドが 4 つだけであるのは残念に思えるかもしれませんが、max_allowed_parallelism を制限するという矛盾した要求があることを覚えておく必要があります。

この結果は、グローバル値の変更は、細心の注意を払って、グローバルなアプリケーションの観点から行う必要があることを思い出させるものです。

task_arena の優先度の設定

ここまで、図 11-11 に示す task_arena コンストラクターの優先度引数は無視してきました。図 11-14 は、設定可能な優先度の値（低、通常、高）を示しています。

```
enum class priority : /* 未指定タイプ */ {
       low =    /* 未指定 */,
       normal = /* 未指定 */,
       high =   /* 未指定 */
};
```

図 11-14. ［scheduler.task_arena］で説明されている tbb::task_arena クラスの優先度定義

10 章で説明したように、アービトレーターはデフォルトで、優先度を考慮した比例割り当てポリシーを使用します。図 11-14 に示す値のいずれかを指定することでアリーナの優先順位を設定できます。高優先度アリーナからのワーカースレッドの需要は、標準または低優先度アリーナにワーカースレッドが割り当てられる前に完全に満たされ、同様に、通常優先度アリーナからの需要は、低優先度アリーナにワーカースレッドが割り当てられる前に完全に満たされます。「ワーカースレッドの需要」は、空いているワーカースロット数とアリーナ内のワークの可用性の両方の関数であることに注意してください。他のスレッドからスチールしたり、共有キューから取得できるタスクがない場合、アリーナには、空いているスロットがあっても、ワーカースレッドの需要がないとみなされます。TBB は、ワーカースレッドがアリーナに到着しても何もすることがない場合、ワーカースレッドをアリーナに供給しません。

図 11-15 は、2 つの異なる標準 C++ スレッド内で、2 つの異なる task_arena インスタンスを作成する例を示しています。この例を実行する関数を図 11-16 に示します。各アリーナはデフォルトのスロット数を使用し、1 つのスロットはマスター用に予約されていますが、1 つは優先度 0 を使用し、もう 1 つは優先度 1 を使用します。各アリーナは異なる C++ スレッドから使用されるため、execute が呼び出されたときに、各アリーナにはマスタースレッドとしてアリーナに参加するスレッド（C++ スレッド）が少なくとも 1 つあることに注意してください。

```
#include  <thread>
#include  <tbb/tbb.h>

void  printArrival(tbb::task_arena::priority  priority);

using  counter_t  = std::atomic<int>;
counter_t  counter  = 0;
void  waitUntil(int  N, counter_t&  c) {
  ++c;
  while  (c  != N);
}

void  explicitArenaWithPriority(tbb::task_arena::priority  priority) {
  tbb::task_arena a{tbb::info::default_concurrency(),
                    /* マスター専用 */ 1, priority};
  a.execute([=]() {
  tbb::parallel_for(0,
                     2*tbb::info::default_concurrency(),
                     [=](int) { printArrival(priority); });
  });
}

void  runTwoThreads(tbb::task_arena::priority  priority0,
                    tbb::task_arena::priority  priority1) {
  std::thread t0([=]() {
     waitUntil(2, counter);
     explicitArenaWithPriority(priority0);
  });
  std::thread t1([=]() {
    waitUntil(2, counter);
    explicitArenaWithPriority(priority1);
  });
  t0.join();
  t1.join();
}
```

図 11-15. 優先順位が異なる 2 つのアリーナが作成されます。`waitUntil` 関数は、スレッドが互いに競合する可能性を高めるバリアを実装します。`main` 関数と `printArrival` 関数を図 11-16 に示します。サンプルコード: `performance_tuning/priorities_and_conflict.cpp`

```
#include  <cstdio>

int  main() {
  counter  = 0;
  std::printf("\n\nrunTwoThreads with low (.) and high (|)\n");
  runTwoThreads(tbb::task_arena::priority::low,
                tbb::task_arena::priority::high);

  counter  = 0;
  std::printf("\n\nrunTwoThreads with low (.) and normal (:)\n");
  runTwoThreads(tbb::task_arena::priority::low,
                tbb::task_arena::priority::normal);

  counter  = 0;
  std::printf("\n\nrunTwoThreads with normal (:) and high (|)\n");
  runTwoThreads(tbb::task_arena::priority::normal,
                tbb::task_arena::priority::high);
  std::printf("\n");
  return  0;
}

void  printArrival(tbb::task_arena::priority  priority) {
  switch  (priority) {
   case  tbb::task_arena::priority::low:
      std::printf(".");
      break;
   case  tbb::task_arena::priority::normal:
      std::printf(":");
      break;
   case  tbb::task_arena::priority::high:
      std::printf("|");
      break;
   default:
      break;
  }
  std::fflush(stdout);
  tbb::tick_count t0 = tbb::tick_count::now();
  while  ((tbb::tick_count::now() -  t0).seconds() < 0.01);
}
```

図 11-16. 図 11-15 で使用される main 関数と printArrival 関数。サンプルコード: performance_tuning/priorities_and_conflict.cpp

図 11-16 の main 関数は、3 つの異なる構成で runTwoThreads を呼び出します。t0 は低優先度で t1 は高優先度、t0 は低優先度で t1 は通常優先度、t0 は通常優先度で t1 は高優先度の組み合わせです。printArrival 関数は、各タスクが parallel_for の反復を実行するときにメッセージを出力します。

呼び出しスレッドが高優先度アリーナにある場合は「|」を出力し、通常優先度アリーナにある場合は「:」を出力し、低優先度アリーナにある場合は「.」を出力します。文字が大きくて密度が高いほど、優先度が高くなります。出力からは、優先度の高いワーク項目が最初に発生することが分かりますが、各アリーナには execute を呼び出したスレッド（C++ スレッド/マスター）が少なくとも 1 つあるため、優先度の低いアリーナもまだ進行します。

この例から得られた出力は次のようになります。

```
runTwoThreads with low (.) and high (|)
|.|||||||.||||||||.............

runTwoThreads with low (.) and normal (:)
:.:::::::.:::::::::...:::::::::

runTwoThreads with normal (:) and high (|)
|||:|||||||:||||||:::::::::::::
```

優先度の高いアリーナが優先的に実行され、ワーカースレッドが最初に受信されました。優先度の高いアリーナの使用可能なスロット数がワーカースレッド数よりも少ない場合、残りのスレッドは優先度の低いアリーナの需要を満たすために使用されます。優先順位の設定では、優先順位の低いアリーナが除外されることはありませんが、優先順位が最も高いアリーナの要求が最初に満たされます。

制約を使用してハードウェア対応のタスクアリーナを作成

明示的なアリーナのスロット数を設定することに加えて、特定の NUMA ノードでワークを実行するようにアリーナを設定したり、P-core のみ、E-core のみなど、特定の種類のコアのみを使用するようにアリーナを制限することもできます。これらの設定はヒントであり、TBB が提案を受け入れる保証はありませんが、通常は可能な場合は受け入れます。TBB はこれらの制約を実装するため OS サポートに依存しており、OS によってサポートの種類が異なり、保証を提供するものもあれば、しないものもあります。

制約パラメーターを受け取る task_arena コンストラクター（または対応する初期化関数）を使用して、TBB に設定を通知します（図 11-11 を参照）。制約パラメーターを渡すと、アリーナをハードウェア指向に設定できます。これらには、numa_id、core_type、max_threads_per_core が含まれます。

tbb::info 名前空間と tbb::task_arena::constraints

info 名前空間には、プラットフォームに適切な値を選択するのに役立つフリー関数があります。これらの関数を図 11-17 に示します。

```
namespace  tbb {
    using  numa_node_id  = /*実装定義*/;
    using  core_type_id  = /*実装定義*/;

     namespace  info {
         std::vector<numa_node_id> numa_nodes();
         std::vector<core_type_id> core_types();

          int  default_concurrency(task_arena::constraints c);
          int  default_concurrency(numa_node_id  id  =
                                   task_arena::automatic);
     }
}  // namespace tbb
```

図 11-17.［info_namespace］で説明されている tbb/info.h の info 名前空間内のフリー関数

TBB ライブラリーは、プラットフォームのトポロジーを決定するため、使用可能な場合は hwloc を使用します。hwloc は、システムのトポロジーに関する情報を照会するポータブルな方法を提供するソフトウェア・パッケージであり、データの配置やスレッド・アフィニティーなどの NUMA 制御を適用できます。hwloc は、本書の執筆時点では TBB の一部として配布されていませんでしたが、Windows、Linux、macOS で広く利用可能であり、TBB はシステムで検出された最新バージョンの hwloc を使用して、図 11-17 の機能をサポートします。

図 11-18 は、コンストラクター、セッター関数、メンバー変数など、task_arena クラスにネストされた制約タイプの詳細を示しています。

```
struct constraints {
    constraints(numa_node_id numa_node_       = task_arena::automatic,
                int          max_concurrency_  =
                task_arena::automatic);
    constraints& set_numa_id(numa_node_id  id);
    constraints& set_max_concurrency(int
    maximal_concurrency);
    constraints& set_core_type(core_type_id  id);
    constraints& set_max_threads_per_core(int  threads_number);

    numa_node_id numa_id = task_arena::automatic;
    int  max_concurrency = task_arena::automatic;
    core_type_id core_type =
    task_arena::automatic;
    int  max_threads_per_core = task_arena::automatic;
};
```

図 11-18. [scheduler.task_arena] で説明されている tbb/info.h にある制約の定義

アリーナを単一の NUMA ノードに制限

パフォーマンスを重視する上級プログラマーは、局所性の活用が最も重要であることを知っています。局所性といえば、すぐに思い浮かぶのはキャッシュの局所性ですが、多くの場合、大規模な共有メモリー・アーキテクチャー上で実行される高負荷アプリケーションでは、非均一メモリーアクセス（NUMA）の局所性も考慮する必要があります。ご存知のとおり、NUMA はメモリーが異なるバンクに編成されており、一部のコアは「遠い」バンクよりも「近い」バンクに高速にアクセスできるという概念を伝えます。正式には、*NUMA ノード*はコア、キャッシュ、およびローカルメモリーのグループ化であり、すべてのコアがローカル共有キャッシュとメモリーへの同じアクセス時間を共有します。ある NUMA ノードから別の NUMA ノードへのアクセス時間は大幅に長くなる可能性があります。プログラムのデータ構造がさまざまな NUMA ノードにどのように割り当てられるか、これらのデータ構造を処理するスレッドがどこで実行されているか（データに近いか、それとも遠いか）など、いくつかの疑問が生じます。

NUMA システムのパフォーマンスのチューニングは、次の 4 つの作業に要約されます:

1. プラットフォーム・トポロジーの検出
2. メモリーアクセスに関連するコストを理解する
3. データの保存場所の制御（データの配置）
4. ワークの実行場所を制御（スレッドまたはタスクのアフィニティー）

TBB ライブラリーは（1）と（4）の作業を大幅に簡素化します。`info` 名前空間の関数 `numa_nodes` は、`numa_node_id` 要素のベクトルを返します。これらの `ID` は、プラットフォームの NUMA ノードに対応します。制約オブジェクトに `numa_node_id` を設定して渡すことで、ベースとなるアリーナに入るすべてのスレッドをマスクする `task_arena` を作成し、それらのスレッドをその NUMA ノードでのみスケジュールできるようにすることができま

す。

TBB は、TBB の責任範囲外のプラットフォーム固有の微妙な知識が必要になる可能性があるため、アクティビティー（2）を支援しません。TBB はアクティビティー（3）を直接支援するわけではありませんが、ファーストタッチをサポートするプラットフォーム（Linux など）ではその機能を使用してループを実行し、後にデータの処理に使用される NUMA ノードにデータを配置できます。

図 11-19 は `tbb::info` と `tbb::task_arena` を使用してアクティビティー（1）と（4）を実行する例を示しています。この例では、`tbb::info` を使用して、プラットフォーム上の NUMA ノード数を検出します。次に、`task_arena` オブジェクト用と `task_group` オブジェクト用の 2 つのベクトルが作成されます。ループでは、それぞれが異なる NUMA ノードに制限されるように、`task_arena` オブジェクトが初期化されます。これらのアリーナにはマスター用に予約されたスロットがないことに注意してください（明示的に `0` を渡します）。これを行うのは、アプリケーション・スレッドは 1 つしかありませんが、アリーナは複数ある可能性があるためです。TBB はデフォルトで `P-1` 個のワーカースレッドを作成するため、1 つを除くすべてのアリーナを完全に埋めるのに十分なワーカースレッドが存在します。例えば、それぞれ 8 個のコアを持つ NUMA ノードが 4 つある場合、TBB は `4 × 8 – 1 = 31` 個のワーカースレッドを作成します。この例では、アプリケーション・スレッドはマスタースレッドとしてアリーナの 1 つに参加します。

アリーナが初期化された後、ループを使用して、1 つを除くすべてのアリーナで `parallel_for` ループを実行するタスクをキューに登録します。`task_group::defer` を呼び出してタスクを作成します。この関数は 6 章で紹介されています。`task_group::defer` を使用するとタスクが作成され、`task_group` 内の参照カウントがインクリメントされます。`task_group` を待機する呼び出しは、このタスクが実行されるまで戻りません。`tbb::task_arena::enqueue` はタスクをアリーナに送信しますが、呼び出しスレッドはアリーナに参加してもワークには参加しません。`task_group::defer` と `tbb::task_arena::enqueue` を組み合わせて使用することで、メインのアプリケーション・ スレッドは、アリーナに参加せずに、各アリーナと `task_group` にワークを安全に送信できるようになります。

そのループの後、アプリケーション・スレッドは残りのアリーナで `execute` を呼び出します。これにより、タスクがアリーナに送信され呼び出しスレッドがアリーナに参加するか、すべてのスロットが一杯の場合、送信されたタスクが完了するまでブロックされます。アプリケーション・スレッドは、`parallel_for` が完了するまでそのアリーナに留まります。

最後に、他のアリーナ（NUMA ノード）のすべてのワークを待機するため、アプリケーション・スレッドは、最終ループに入って execute を使用して残りの各アリーナに送信します。これらのアリーナが完全に設定されている可能性が高いですが、その場合でも、呼び出しスレッドは task_group[i].wait() タスクが完了するまでブロックされます。

図 11-19 の例は、NUMA システムで tbb::info と tbb::task_group サポートを使用する 1 つの方法にすぎません。TBB は、プラットフォームのプロパティーを照会し、アリーナを制限するのに使用できる基本的な機能を提供します。これらの機能をどのように使用するかは、私たち次第です。

```
#include  <vector>
#include  <tbb/tbb.h>

int  N  = 1000;
double  w  = 0.01;
double  f(double  v);

void  constrain_for_numa_nodes() {
  std::vector<tbb::numa_node_id> numa_nodes  = tbb::info::numa_nodes();
  std::vector<tbb::task_arena> arenas(numa_nodes.size());
  std::vector<tbb::task_group> task_groups(numa_nodes.size());

  // 各アリーナを初期化し、それぞれ異なる NUMA ノードに制限
  for  (int  i  = 0; i  < numa_nodes.size(); i++)
    arenas[i].initialize(tbb::task_arena::constraints(numa_nodes[i]), 0);

  // 最初のアリーナ以外のすべてのアリーナに作業をキューに入れて task_group を使用して
  // 作業を追跡。defer を使用すると、task_group の参照カウントがすぐに増加
  for  (int  i  = 1; i  < numa_nodes.size(); i++)
    arenas[i].enqueue(
      task_groups[i].defer([] {
        tbb::parallel_for(0, N, [](int  j) { f(w); });
      })
    );

  // 残りのアリーナでワークを直接完了まで実行
  arenas[0].execute([] {
    tbb::parallel_for(0, N, [](int  j) { f(w); });
  });

  // 他のアリーナに参加して、task_groups を待機
  for  (int  i  = 1; i  < numa_nodes.size(); i++)
    arenas[i].execute([&task_groups, i] { task_groups[i].wait(); });
}
```

図 11-19. info::numa_nodes を使用して、NUMA ノードごとに 1 つの task_group と 1 つの task_arena を作成します。サンプルコード: performance_tuning/priorities_and_conflict.cpp

アリーナを特定のコアタイプに制限

インテルのハイブリッド・コアと Arm の big.LITTLE テクノロジーの導入により、単一の CPU に効率と処理能力が異なる多様なコアセットを搭載できるようになりました。例えば、2021 年に導入された第 12 世代インテル Core プロセッサー（開発コード名 Alder Lake）は、パフォーマンス・コア（P-core）とエフィシェント・コア（E-core）の両方を含む CPU です。ほとんどの場合、オペレーティング・システムはスレッドを適切なコアタイプに動的に移動できるため、介入の必要はありません。実際、インテル・スレッド・ディレクターなどの複雑なソフトウェアやハードウェア・システムが導入されており、オペレーティング・システムと連携してワットあたりのパフォーマンスを最大化します。しかし、私たちがもっとよく知っていると思うなら（おそらくそうではないでしょうが）、TBB の機能を使って特定のコアタイプでワークを実行するアリーナを作成し、注意深く協調設計されたインテリジェントなシステムを回避できます。

`task_arena` 制約とともに使用される `info` 名前空間の関数により、介入が可能になります。`tbb::info` 名前空間の関数 `core_types`（図 11-20）は、`core_type_id` 要素のベクトルを返します。このベクトルは、システムで使用可能なコアタイプのうち、最もパフォーマンスが低いものから最もパフォーマンスが高いものの順にソートされます。したがって、前述の E-core と P-core を備えた Alder Lake プラットフォームでは、ベクトルには 2 つの要素が含まれます。最初の `ID` は E-core を表し、2 番目の `ID` は P-core を表します。制約オブジェクトに `core_type_id` を設定して渡すことで、ベースとなるアリーナに入るすべてのスレッドをそのコアタイプにバインドする `task_arena` を作成できます。

```
void  constrain_for_core_type() {
    std::vector<tbb::core_type_id> core_types  = tbb::info::core_types();
    tbb::task_arena arena(
      tbb::task_arena::constraints{}.set_core_type(core_types.back())
    );

    arena.execute([] {
        tbb::parallel_for(0, N, [](int) { f(w); });
    });
}
```

図 11-20. `info::core_types` を使用して、システム上で最もパフォーマンスの高いコアタイプを使用する `task_arena` を作成します。サンプルコード: `performance_tuning/constraints.cpp`

コアごとの最大スレッド数を設定

一部のプロセッサーでは、物理コアよりも論理コア数が多くなる場合があります。このテクノロジーは、ハイパースレッディング（HT）または同時マルチスレッディング（SMT）と呼ばれます。これをサポートするプロセッサーは、同じ物理コア上にある論理コア間で一部のリソースを共有します。デフォルトでは、TBB は論理コアごとに 1 つのスレッドを使用することを想定しており、その想定に基づいてアリーナ内にアリーナスロットを作成し、グローバル・スレッド・プール内にワーカースレッドを作成します。場合によっては、パフォーマンス測定に基づいて、論理コア間でリソースを共有するオーバーヘッドを回避するため、物理コアごとに使用される論理コア数を制限する必要があります。これを行うには、制約オブジェクトで `max_threads_per_core` 値を設定し、それを渡して `task_arena` を作成し、各物理コアで実行できるスレッド数を制限します。制限が望ましい場合の一般的な選択肢は、図 11-21 に示すように値を 1 に設定することです。

図 11-21 では、まず `tbb::info` を使用して `core_type_id` のベクトルを取得します。次に、最もパフォーマンスの高いコアタイプを使用する制約を作成しますが、`c.set_max_threads_per_core(1)` を使用して物理コアあたりのスレッド数を 1 に制限し、そのコアタイプで HT または SMT が存在する場合はオフにします。

```
void  constrain_for_no_hyperthreading() {
    tbb::task_arena::constraints c;
    std::vector<tbb::core_type_id> core_types  = tbb::info::core_types();
    c.set_core_type(core_types.back());
    c.set_max_threads_per_core(1);
    tbb::task_arena no_ht_arena(c);

    no_ht_arena.execute( [] {
        tbb::parallel_for(0, N, [](int) { f(w); });
    });
}
```

図 11-21. コアあたりで使用する最大スレッド数を 1 に設定します。サンプルコード: performance_tuning/constraints.cpp

コア上で実行されるスレッドの数を制限する、より容易な構成の方法は、図 11-22 に示すように `tbb::task_arena` の最大同時実行性を設定することです。この例では、図 11-21 と同じ制約オブジェクトを作成しますが、それを `task_arena` に制約するのに使用せず、関数 `tbb::info::default_concurrency` に渡します。この関数に制約を渡すと、その制約によるデフォルトの同時実行性が返されます。

2 つの P-core（各 P-core は 2 つのハードウェア・スレッド（HT）をサポート）と、8 つの E-core（各コアは 1 つのハードウェア・スレッドのみをサポート）を備えたプラットフォームを考えてみましょう。このプラットフォームには、合計 `2 × 2 + 8 = 12` 個のハードウェア・スレッドがあります。ただし、図 11-22 の `tbb::task_arena::constraints c` が指定されている場合、`tbb::info::default_concurrency(c)` は 12 ではなく 2 を返します。P-core が 2 つあり、P-core ごとに 1 つのスレッドのみを使用する場合、デフォルトの同時実行性は 2 になります。

```
void  limit_concurrency_for_no_hyperthreading() {
    tbb::task_arena::constraints c;
    std::vector<tbb::core_type_id> core_types  = tbb::info::core_types();
    c.set_core_type(core_types.back());
    c.set_max_threads_per_core(1);
    int  no_ht_concurrency  = tbb::info::default_concurrency(c);
    tbb::task_arena arena( no_ht_concurrency );

    arena.execute( [] {
        tbb::parallel_for(0, N, [](int) { f(w); });
    });
}
```

図 11-22. ハイパースレッディングの使用を明示的に妨げずに、コアごとに 1 つのスレッドを持つのに十分なスロットだけが存在するよう、アリーナ内のスレッド数を減らします。サンプルコード: `performance_tuning/constraints.cpp`

task_scheduler_observer を使用して独自のコードを実行

制約を使用するだけではスレッドを希望どおりに構成できない場合、もう 1 つのフォールバック・オプションである `task_scheduler_observer` があります。`task_scheduler_observer` は、スレッドがアリーナ内のタスクの処理を開始および停止するタイミングを監視します。この機能を使用するには、`task_scheduler_observer` から独自のクラスを派生し、`on_scheduler_entry` または `on_scheduler_exit` をオーバーライドして独自のコードを導入します。作成時に監視は無効になっていますが、監視を有効にするには、`observe()` を呼び出す必要があります。図 11-23 は `task_scheduler_observer` のクラス宣言を示しています。

```
namespace  tbb {

   class  task_scheduler_observer  {
   public:
       task_scheduler_observer();
       explicit  task_scheduler_observer( task_arena& a  );
       virtual  ~task_scheduler_observer();

       void  observe( bool  state=true  );
       bool  is_observing() const;

       virtual  void  on_scheduler_entry( bool  is_worker  ) {}
       virtual  void  on_scheduler_exit( bool  is_worker  } {}
   };

}  // namespace tbb
```

図 11-23. ［scheduler.task_scheduler_observer］で説明されている task_scheduler_observer クラス

図 11-24 は、tbb::task_scheduler_observer を継承する PinningObserver クラスの例を示しています。このクラスは、task_arena a へのスレッドの出入り時に、OS 固有の関数を呼び出して各スレッドのスレッド・アフィニティー・マスクを直接設定するために使用できます。この図では、スレッドがアリーナに出入りした際に出力する関数を提供していますが、OS 固有のピンニング呼び出しは含まれていません。

```
#include <iostream>
#include <sstream>
#include <thread>
#include <tbb/tbb.h>

// これらは OS 固有の型や呼び出しを配置する場所のプレースホルダー
using affinity_mask_t = std::string;
void set_thread_affinity( int tid, const affinity_mask_t& mask ) {
  std::ostringstream buffer;
  buffer << std::this_thread::get_id()
         << " -> (" << tid
         << ", " << mask << ")\n";
  std::cout << buffer.str();
}
void restore_thread_affinity() {
  std::ostringstream buffer;
  buffer <<std::this_thread::get_id()
         << " -> (restored)\n";
  std::cout << buffer.str();
}

// オブザーバー・クラス
class PinningObserver : public tbb::task_scheduler_observer {
public:
    // アリーナ内のスレッドに使用される HW アフィニティー・マスク
    affinity_mask_t m_mask;
    PinningObserver( tbb::task_arena &a, const affinity_mask_t& mask )
        : tbb::task_scheduler_observer(a), m_mask(mask) {
        observe(true); // オブザーバーをアクティブにする
    }
    void on_scheduler_entry( bool worker ) override {
        set_thread_affinity(
            tbb::this_task_arena::current_thread_index(), m_mask);
    }
    void on_scheduler_exit( bool worker ) override {
        restore_thread_affinity();
    }
};
```

図 11-24. スレッドが領域に出入りするときにスレッド・アフィニティーを設定および復元するユーザー提供の関数を呼び出す `PinningObserver`。この例では、OS 固有のコードを配置できるプレースホルダー関数 `set_thread_affinity` と `restore_thread_affinity` を提供します。関数 `current_thread_index` は、スレッドがアリーナ内で占めるスロットの番号を返します。サンプルコード: `performance_tuning/task_scheduler_observer.cpp`

図 11-25 では、`PinningObserver` を使用して、スレッドが 2 つのタスクアリーナに出入りする様子を観察します。各アリーナは、`tbb::info::default_concurrency()/2` スロットを持つように初期化されます。

```
void  observeTwoArenas() {
  int  P  = tbb::info::default_concurrency();

  // 2 つのアリーナ、それぞれ半分の HW スレッド
  tbb::task_arena a0(P/2);
  tbb::task_arena a1(P/2);

  PinningObserver  obs0(a0, "mask_zero");
  PinningObserver  obs1(a1, "mask_one");

  // 連続ループを実行
  std::cout  <<  "Execute a0 loop\n";
  a0.execute([] {
    tbb::parallel_for(0, N, [](int  j) { f(w); });
  });
  std::cout  <<  "Execute a1 loop\n";
  a1.execute([] {
    tbb::parallel_for(0, N, [](int  j) { f(w); });
  });

  // 同時ループを実行
  std::cout  <<  "Execute a0 and a1 concurrently\n";
  std::thread  t0([&]() {
    waitUntil(2, counter);
    a0.execute([] {
      tbb::parallel_for(0, N, [](int  j) { f(w); });
    });
  });
  std::thread  t1([&]() {
     waitUntil(2, counter);
     a1.execute([] {
      tbb::parallel_for(0, N, [](int  j) { f(w); });
    });
  });
  t0.join();
  t1.join();
}
```

図 11-25. PinningObserver を使用して、2 つのアリーナからのスレッドの入口と出口を観察します。サンプルコード: performance_tuning/task_scheduler_observer.cpp

この例を 8 つのハードウェア・スレッドを備えたプラットフォームで実行すると、次の出力が表示されました:

```
Execute a0 loop
139781604495104 -> (0, mask_zero)
139781564798720 -> (1, mask_zero)
139781568997120 -> (2, mask_zero)
139781560600320 -> (3, mask_zero)
139781560600320 -> (restored)
139781564798720 -> (restored)
139781568997120 -> (restored)
139781604495104 -> (restored)
Execute a1 loop
139781604495104 -> (0, mask_one)
139781564798720 -> (1, mask_one)
139781568997120 -> (2, mask_one)
139781560600320 -> (3, mask_one)
139781604495104 -> (restored)
Execute a0 and a1 concurrently
139781560600320 -> (restored)
139781568997120 -> (restored)
139781564798720 -> (restored)
139781556401920 -> (0, mask_zero)
139781564798720 -> (1, mask_zero)
139781568997120 -> (2, mask_zero)
139781560600320 -> (3, mask_zero)
139781548009216 -> (0, mask_one)
139781535418112 -> (3, mask_one)
139781539616512 -> (2, mask_one)
139781531219712 -> (1, mask_one)
139781556401920 -> (restored)
139781564798720 -> (restored)
139781560600320 -> (restored)
139781568997120 -> (restored)
139781548009216 -> (restored)
```

各行の左側には、参加する各スレッドの一意のスレッド ID が表示されます。スレッドがアリーナに入る場合、右側には、スレッドがアリーナ内で占有しているスロットと、この例で実際の OS 固有のピンニングコードがある場合に適用されるマスクを示すペア（アリーナスロット、マスク）が表示されます。スレッドがアリーナを離れると、右側は（**restored**）になります。

この出力には、注目すべき興味深い点がいくつかあります。まず、ループが連続して実行されると、スレッドがアリーナ間を移動することが分かります。例えば、スレッド `139781564798720` は `a0` に参加し、その後 `a1` に移動します。ループが同時に実行されると、2 つのアリーナ全体でより多くのワーカーが参加していることが分かります。

粒度と局所性に関する機能

これまでこの章では、`global_control` または明示的な `task_arena` オブジェクトを使用してリソース割り当てを制御することに焦点を当ててきました。

2 章では、TBB ライブラリーが提供する一般的な並列アルゴリズムについて説明し、その使い方を示す例をいくつか示しました。その際、アルゴリズムのデフォルトの動作は多くの場合十分に優れているものの、必要に応じてパフォーマンスを調整する方法があることに気付きました。この節では、いくつかの TBB アルゴリズムを再検討してその主張を裏付け、デフォルトの動作を変更するために使用できる重要な機能について説明します。

ここでの議論の中心となる懸念事項は 3 つあります。1 つ目は**粒度**、つまりタスクが実行するワークの量です。TBB ライブラリーはタスクのスケジュール設定に効率的ですが、特にタスクが非常に小さいか非常に大きい場合、タスクのサイズがパフォーマンスに大きく影響する可能性があるため、アルゴリズムが作成するタスクサイズを考慮する必要があります。2 番目の問題は**データの局所性**です。「序文」で詳しく説明したように、アプリケーションがキャッシュとメモリーをどのように使用するかによって、アプリケーションのパフォーマンスが決まります。3 番目の問題は、**利用可能な並列性**です。TBB を使用する目標は、もちろん並列処理を導入することですが、粒度と局所性を考慮せずに盲目的に並列処理を実行することはできません。アプリケーションのパフォーマンスを調整することは、多くの場合、これら 3 つの懸念事項の間のトレードオフのバランスを取ることです。

TBB アルゴリズムと Parallel STL などの他のインターフェイスとの主な違いの 1 つは、TBB アルゴリズムが、これら 3 つの懸念事項に関連する動作に影響するフックと機能を提供していることです。TBB アルゴリズムは、制御できない単なるブラックボックスではありません。

この節では、タスクの粒度について説明し、タスクサイズはどの程度の大きさであれば十分であるか経験則を示します。次に、単純なループ・アルゴリズムと、レンジとパーティショナーを使用してタスクの粒度とデータの局所性を制御する方法に焦点を当てます。また、パフォーマンスをチューニングする際の決定論と、それが柔軟性に与える影響についても簡単に説明します。この章の最後に、TBB パイプライン・アルゴリズムに注目し、その機能が粒度、データの局所性、最大並列性にどのように影響するか説明します。

使用したテストマシン

この章では、パフォーマンスに関連する機能の説明に役立つパフォーマンス測定値をいくつか収集します。これらは厳密に収集されたベンチマークの結果ではありませんが、主要なパフォーマンス上の懸念事項や特定の TBB 機能の影響を示すために使用されます。

テストマシンは 2 台使用します。

ここで、Linux サーバーシステムと呼ぶ最初のマシンは、Ubuntu 22.04.4 LTS を実行する 2 つのインテル Xeon Platinum 8580 プロセッサー（開発コード名 Emerald Rapids）を搭載したシステムです。このシステムには 2 つのソケットがあり、各ソケットに 2 つの NUMA ノードがあります。各ソケットには 60 個の物理コアがあり、各コアは 2 つの論理コアをサポートします（インテル・ハイパースレッディング・テクノロジを使用）。したがって、システム全体では 2 × 60 × 2 = 240 個の論理コアがあります。ほとんどすべての実験では、例で使用されるスレッドの数を減らして、説明を簡単にし、より小さなデータセットのサイズを使用できるようにしています。2 つのプロセッサーのそれぞれに 300MB のキャッシュがあり、システムの合計メモリーは 528MB です。

2 つ目のシステムをラップトップ・システムと呼びます。これは、Windows 11 を「最高のパフォーマンス」モードで実行する、インテル Core Ultra 7 165u プロセッサー（開発コード名 Meteor Lake）を搭載したシステムです。このプロセッサーには、2 つのパフォーマンス・コア（P-core）と 8 つのエフィシェント・コア（E-core）があります。2 つの P-core はそれぞれ、2 つの論理コア（インテル・ハイパースレッディング・テクノロジーを使用）をサポートします。したがって、システム全体では、2 × 2 + 8 = 12 個の論理コアがあります。このシステムには、12MB のインテル・スマート・キャッシュと 32GB のメモリーがあります。

タスクの粒度: どれくらいの大きさが十分か?

TBB ライブラリーがスレッド間で負荷を分散する際に最大限の柔軟性を持つようにするには、アルゴリズムによって実行されるワークを多くの部分に分割する必要があります。同時に、ワークスチールとタスク・スケジュールのオーバーヘッドを最小限に抑えるため、できるだけ大きなタスクを作成したいと考えています。これらは互いに相反するため、アルゴリズムの最高のパフォーマンスはその中間のどこかで得られます。

問題を複雑にしているのは、正確な最適なタスクサイズがプラットフォームやアプリケー

ションによって異なるため、通常適用される正確なガイドラインが存在しないことです。それでも、ガイドラインとして使用できる大まかな数値を知っておくことは有用です。これらの注意事項を念頭に置いて、次のような経験則を提案します。

タスクサイズに関する経験則

ワークスチールのオーバーヘッドを効果的に隠匿するには、TBB タスクの平均が 1 マイクロ秒を超える必要があります。

すべてのタスクが 1 マイクロ秒を超える必要はありません。実際、それが不可能な場合がよくあります。例えば、分割統治アルゴリズムでは、小さなタスクを使用してワークを分割し、次にリーフでより大きなタスクを使用することがあります。TBB の `parallel_for` アルゴリズムはこのように動作します。TBB タスクは、レンジを分割し、最後のサブレンジにボディーを適用するために使用されます。分割タスクは通常、非常に小さなワークしか行いませんが、ループボディーのタスクははるかに大きなワークを行います。この場合、すべてのタスクを 1 マイクロ秒以上にすることはできませんが、タスクサイズの平均を 1 マイクロ秒より大きくすることを目指すことができます。

`parallel_invoke` などのアルゴリズムを使用したり、TBB タスクを直接使用する場合、タスクサイズを完全に制御できます。例えば、2 章では、`parallel_invoke` を使用してクイックソートの並列バージョンを実装し、配列のサイズ（およびタスク実行時間）がカットオフのしきい値を下回ると、再帰並列実装をシリアル実装に切り替えました:

```
if  (end -  begin < cutoff) {
    serialQuicksort(begin, end);
}
```

`parallel_for`、`parallel_reduce`、`parallel_scan` などの単純なループ・アルゴリズムを使用する場合、レンジ引数とパーティショナー引数によって必要な制御が提供されます。これらについては次の節でさらに詳しく説明します。

ループレンジとパーティショナーの選択

2 章で紹介したように、レンジは再帰的に分割可能な値のセット（通常はループの反復空間）を表します。レンジは、`parallel_for`、`parallel_reduce`、`parallel_deterministic_reduce`、`parallel_scan` などの単純なループ・アルゴリズムで使用します。TBB アルゴリズムはレンジを分割し、TBB タスクを使用してアルゴリズムのボディー・オブジェクトをこれらのサブレンジに適用します。パーティショナーと組み合わせることで、レンジは反復スペースを表現し、それをタスクに分割してワーカースレッドに割り当てる方法を制御するシンプルかつ強力な方法を提供します。このパーティション化は、タスクの粒度とデータの局所性を調整するために使用できます。

クラスがレンジになるには、図 11-26 に示すように、TBB 仕様 [`req.range`] の名

前付き要件レンジの要件セットに一致する必要があります。レンジはコピーでき、*分割コンストラクター*を使用して分割でき、オプションで*比例分割コンストラクター*を提供することもできます。また、空であるか、または分割可能かどうかを確認するメソッドも提供する必要があります。

擬似署名	セマンティクス
`R::R(const R&)`	コピー・コンストラクター。
`R::~R()`	デストラクター。
`bool R::empty() const`	範囲が空の場合は `true` です。
`bool R::is_divisible() const`	範囲を 2 つのサブ範囲に分割できる場合は
`R::R(R &r, split)`	基本分割コンストラクター。`r` を 2 つのサブ範囲に分割します。
`R::R(R &r, proportional_split proportion)`	**オプション。**比例分割コンストラクター。`r` を `proportion` に応じて 2 つのサブ範囲に分割します。

図 11-26. 要件 [req.range] という名前のレンジ

独自のレンジタイプを定義することもできますが、TBB ライブラリーでは、図 11-27 に示すようなブロックされたレンジが提供されており、ほとんどの状況に対応します。例えば、次のネストされたループの反復空間は、

```
blocked_range2d<int, int> r(i_begin, i_end, j_begin, j_end)
```

を使用して表現できます:

```
for  (int  i = i_begin; i < i_end, ++i )
  for  (int  j = j_begin; j < j_end; ++j )
   /* ループ本体 */
```

レンジタイプ	コンストラクター引数	説明

`blocked_range`	`Value begin、`
	`Value end、`
	`size_type grainsize`
		1 次元のレンジをモデル化。

`blocked_range2d`	`RowValue row_begin、`
	`RowValue row_end、`
	`[size_type row_grainsize]、`
	`ColValue col_begin、`
	`ColValue col_end、`
	`[size_type col_grainsize]`
		2 次元のレンジをモデル化。 繰り返し分割すると、サブレンジは行と列の粒度のアスペクト比に近づきます。

`blocked_range3d`	`PageValue page_begin、`
	`PageValue page_end、`
	`[size_type page_grainsize]、`
	`RowValue row_begin,`
	`RowValue row_end、[size_type`
	`row_grainsize]、ColValue`
	`col_begin、ColValue col_end、`
	`[size_type col_grainsize]`
		3 次元のレンジをモデル化。繰り返し分割すると、サブレンジはページ、行、および列の粒度サイズのアスペクト比に近づきます。

`blocked_nd_range`	`const dim_range_type& dim0,`
	`/*, ...同じタイプの N 個のパラメーター */`
	`const Value (&dim_size)[N],`
	`size_type grainsize = 1`
		N 次元のレンジをモデル化。この表の他のブロック・レンジ・タイプとは異なり、すべての次元は同じ値タイプに基づいています。 `dim_range_type` は `blocked_range<Value>` です。コンストラクターは 2 つあります。最初のコンストラクターは、次元ごとに 1 つずつ、正確に N 個の `dim_range_type` オブジェクトを受け取ります。2 番目のコンストラクターは、各要素が対応する次元サイズである N 値型要素の配列を取り、レンジが 0 から始まると想定します。2 番目のコンストラクターも、すべての次元に対して単一の共通粒度を受け取ります。

図 11-27. TBB によって提供されるさまざまな定義済みレンジタイプは、`[algorithms.blocked_range]`、`[algorithms.blocked_range2d]`、`[algorithms.blocked_range3d]`、`[algorithms.blocked_nd_range]` で説明されています

パーティショナーの概要

TBB アルゴリズムは、レンジに加えて、アルゴリズムがレンジを分割する方法を指定するパーティショナーもサポートします。さまざまなパーティショナー・タイプを図 11-28 に示しま

す。

パーティショナー	説明	粒度 g の blocked_range で使用する場合
simple_partitioner	チャンクサイズは粒度によって決まります。	g/2 ≤ chunksize ≤ g
auto_partitioner	粒度が下限として機能する自動チャンクサイズ。	g/2 ≤ chunksize
affinity_partitioner	粒度が下限として機能する自動チャンクサイズ。キャッシュ・アフィニティーと反復の初期の均一分散。	
static_partitioner	決定論的なチャンクサイズ。キャッシュ・アフィニティーと、それ以上の細分化を行わない反復の均一な分散。粒度によって P チャンクの作成が妨げられない限り、均一な分布が作成されます。	max(g/3, N/P) ≤ chunksize 説明: N は問題サイズ P はタスクの実行に参加できるスレッドの数です。

図 11-28. TBB が提供するパーティショナー・タイプは、[algorithms. simple_partitioner]、[algorithms.auto_partitioner]、[algorithms.affinity_partitioner]、[algorithms.static_partitioner] で説明されています

simple_partitioner は、is_divisible メソッドが false を返すまでレンジを再帰的に分割するのに使用されます。ブロックされたレンジタイプの場合、レンジサイズがその粒度以下になるまで分割されることを意味します。粒度を高度に調整した場合（次の節で説明）、最終的なサブレンジが指定された粒度を尊重することを保証する simple_partitioner を使用します。

auto_partitioner は、負荷を分散するためレンジを分割する動的アルゴリズムを使用しますが、必ずしも is_divisible が許容するほど細かくレンジを分割するわけではありません。ブロック化されたレンジクラスで使用する場合、粒度は最終的なチャンクのサイズの下限を提供しますが、auto_partitioner がより大きな粒度を決定できるため、粒度の重要性は大幅に低下します。したがって、粒度 1 を使用し、auto_partitioner に最適な粒度を決定させるのが一般的です。oneTBB では、parallel_for、parallel_reduce、parallel_scan で使用されるデフォルトのパーティショナー・タイプは auto_partitioner です。

static_partitioner は、レンジを可能な限り均一にワーカースレッドに分散し、それ以上の負荷分散はありません。ワークの分散とスレッドへのマッピングは決定論的であり、反復回数、粒度、およびスレッド数によってのみ決まります。

static_partitioner は動的な決定を行わないため、パーティショナーの中で最もオーバーヘッドが低くなります。static_partitioner を使用すると、同じループの実行全体でスケジュール・パターンが繰り返されるため、キャッシュの動作が改善されることもあります。ただし、static_partitioner は負荷分散を厳しく制限するため、慎重に使用してください。「static_partitioner を使用」の節では、static_partitioner の長所と短所について説明します。

affinity_partitioner は、auto_partitioner と static_partitioner の長所を組み合わせ、同じデータセットに対してループが再実行されるときに同じパーティショナー・オブジェクトが再利用されると、キャッシュ・アフィニティーが向上します。affinity_partitioner は、static_partitioner と同様に、最初は均一な分散を行いますが、追加の負荷分散を可能にします。また、どのスレッドがどのレンジのチャンクを実行したか履歴も保持し、後続の実行時にこの実行パターンを再現しようとします。データセットがプロセッサーのキャッシュに完全に収まる場合、スケジュール・パターンを繰り返すことでパフォーマンスが大幅に向上する可能性があります。

タスクの粒度を管理する粒度サイズの選択（または選択しない）

この節の冒頭で、タスクの粒度が重要であることを説明しました。ブロック化されたレンジ型を使用する場合、常に粒度を細かく調整する必要がありますか？ 必ずしもそうとは言えません。ブロック化された範囲を使用するときに適切な粒度を選択することは、重要なこともあれば、ほとんど無関係である場合もあります。すべては、使用されているパーティショナーによって決まります。

simple_partitioner を使用する場合、粒度はボディーに渡されるレンジサイズを決定する唯一の要素になります。simple_partitioner を使用すると、is_divisible が false を返すまでレンジが再帰的に分割されます。対照的に、他のすべてのパーティショナーには、レンジの分割を停止するタイミングを決定する独自の内部アルゴリズムがあります。下限としてのみ割り切れるその他のパーティショナーの場合、通常は粒度 1 を選択すれば十分です。

さまざまなパーティショナーに対する粒度の影響を示すため、単純な parallel_for マイクロベンチマーク（図 11-29）を使用して、ループの反復回数（N）、粒度（gs）、ループ反復あたりの実行時間（tpi）、およびパーティショナー（p）を変更します。

```
template< typename  P  >
static  inline  double  executePfor(int  num_trials, int  N,
                                    int  gs, P &p, double  tpi) {
    tbb::tick_count  t0;
    for  (int  t = -1; t < num_trials; ++t) {
    if  (!t) t0  = tbb::tick_count::now();
        tbb::parallel_for  (
            tbb::blocked_range<int>{0, N, gs},
            [tpi](const  tbb::blocked_range<int> &r) {
                for  (int  i = r.begin(); i < r.end(); ++i){
                    spinWaitForAtLeast(tpi);
                }
            },
            p // パーティショナー
        );
    }
    tbb::tick_count  t1  = tbb::tick_count::now();
    return  (t1 -  t0).seconds()/num_trials;
}
```

図 11-29. パーティショナー（p）、粒度（gs）、反復あたりの時間（tpi）を使用して、N 反復で parallel_for を実行する時間を測定する関数。サンプルコード: performance_tuning/parallel_for_partitioners_timed.cpp

図 11-30 は、TBB で使用可能な各パーティショナー・タイプとさまざまな粒度を使用して、N = 2^{18} で実行された、図 11-29 のプログラムの結果を示しています。

tbb::task_arena の制約を使用して、計算をコアごとに 1 つのスレッド、合計 8 つのスレッドを持つ単一の NUMA ノードに制限しました。

非常に小さい 100ns の time_per_iteration の場合、粒度が 16 以上のときに simple_partitioner が他のパーティショナーの最大パフォーマンスに近づくことが分かります。反復あたりの時間が長くなると、スケジュールのオーバーヘッドを相殺するのに必要な反復回数が少なくなるため、simple_partitioner は最大パフォーマンスに早く近づきます。

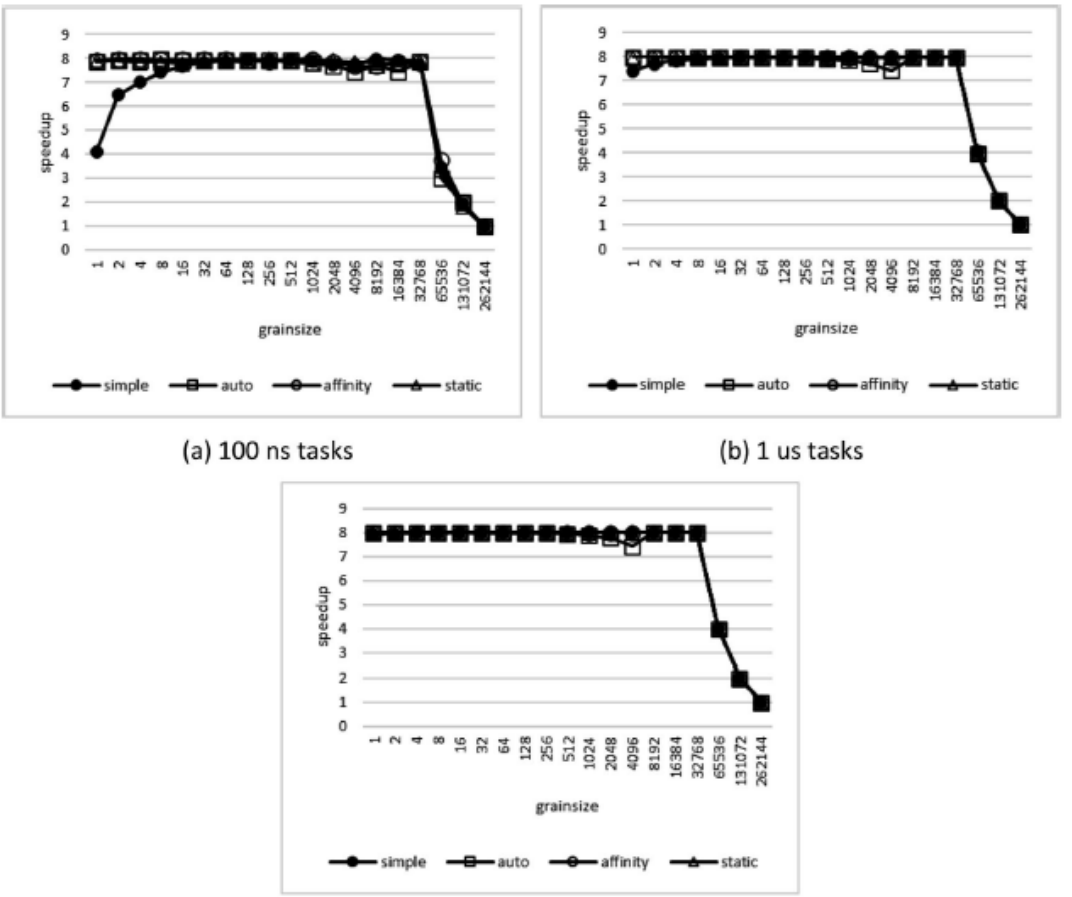


図 11-30. シリアル実行と比較して、さまざまなパーティショナー・タイプと粒度の増加によるスピードアップ。テスト対象のループの合計反復回数は 218 = 262,144 です。これらの結果は、この章の前半で説明した Linux サーバーシステムで収集しました。tbb::task_arena 制約を使用して、計算をコアごとに 1 つのスレッドと最大 8 つのハードウェア・スレッドの同時実行を持つ単一の NUMA ノードに制限しました

図 11-30 に示されている simple_partitioner 以外のすべてのパーティショナー・タイプでは、粒度が 1 から 4,096 で最大のパフォーマンスが得られます。アプリケーションは 8 つのスレッドを使用するように構成されているため、各スレッドに少なくとも 1 つのチャンクを提供するには、218/8 = 32,768 以下の粒度が必要です。その結果、粒度が 32,768 を超えると、すべてのパーティショナーのパフォーマンスが低下し始めます。また、粒度が 4,096 の場合、すべてにおいて auto_partitioner と affinity_partitioner のパフォーマンスが低下することにも注意してください。

これは、大きな粒度を選択すると、これらのアルゴリズムで利用できる選択肢が制限され、自動パーティショニングを行う能力が妨げられるためです。

この実験により、単純な分割には粒度が非常に重要であることが確認されました。`simple_partitioner` を使用してタスクサイズを手動で選択することもできますが、その場合はより正確に選択する必要があります。

2 つ目のポイントは、ボディーサイズが `1us`（`100ns × 16 = 1.6us`）に近づくと、線形の上限に近いスピードアップを伴う効率的な実行が見られるという点です。この結果は、本書の前半で紹介した経験則を裏付けるものです。そもそも、このような経験と実験が私たちの経験則の根拠となっているので、これは驚くべきことではありません。

レンジ、パーティショナー、およびデータキャッシュのパフォーマンス

レンジとパーティショナーは、キャッシュ非依存アルゴリズムを有効にするか、キャッシュ・アフィニティーを有効にすることで、データキャッシュのパフォーマンスを向上できます。キャッシュ非依存アルゴリズムは、データセットが大きすぎてデータキャッシュに収まらない場合に役立ちますが、分割統治法を使用して解決する場合は、アルゴリズム内でデータの再利用を利用できます。対照的に、キャッシュ・アフィニティーは、データセットがキャッシュに完全に収まる場合に役立ちます。キャッシュ・アフィニティーは、レンジの同じ部分を同じプロセッサーに繰り返しスケジュールするのに使用されます。これにより、キャッシュに収まるデータに同じキャッシュから再度アクセスできるようになります。

キャッシュ非依存アルゴリズム

キャッシュ非依存アルゴリズムは、ハードウェアのキャッシュ・パラメーターの知識に依存せずに、データキャッシュを適切に（または最適に）使用するアルゴリズムです。概念はループのタイリングやループのブロッキングに似ていますが、正確なタイルサイズやブロックサイズは必要ありません。キャッシュ非依存アルゴリズムでは、多くの場合、問題が再帰的に小さなサブ問題に分割されます。ある時点で、これらの小さなサブ問題がマシンのキャッシュに収まり始めます。再帰的な分割は、可能な限り最小のサイズまで継続される場合もありますが、効率性のためカットオフポイントが存在する場合もあります。ただし、カットオフポイントはキャッシュサイズとは関係がなく、通常は、妥当なキャッシュサイズをはるかに下回るサイズのデータにアクセスするパターンが作成されます。

キャッシュ非依存アルゴリズムはキャッシュのパフォーマンスに全く関心がないため、遭遇するあらゆるキャッシュに合わせて最適化する「*キャッシュ・アグノスティック*」や、キャッシュのレベルが無限にあると想定する「*キャッシュパラノイド*」など、さまざまな名前が提案されています。しかし、文献では「cache-oblivious」という名前が使用されており、それが定着しています。

ここでは、キャッシュを無視した実装の恩恵を受けるアルゴリズムの例として、行列転置を使用します。キャッシュ非依存アルゴリズムではない行列転置のシリアル実装を図 11-31 に示します。

```
void  serialTranspose(int  N, double  *a, double  *b) {
  for  (int  i  = 0; i  < N; ++i) {
    for  (int  j  = 0; j  < N; ++j) {
      b[j*N+i] = a[i*N+j];
    }
  }
}
```

図 11-31.　行列転置のシリアル実装。サンプルコード: performance_tuning/parallel_for_transpose_partitioners.cpp

簡単にするために、マシンのキャッシュラインに 4 つの要素が収まると仮定しましょう。図 11-32 は、N × N の行列 a の最初の 2 行の転置中にアクセスされるキャッシュラインを示しています。キャッシュが十分に大きい場合、a の最初の行の転置中に b でアクセスされるすべてのキャッシュラインを保持できるため、a の 2 番目の行の転置中にこれらを再ロードする必要はありません。ただし、キャッシュが十分に大きくない場合、これらのキャッシュラインを再ロードする必要があり、その結果、行列 b にアクセスするたびにキャッシュミスが発生します。図では 16 × 16 の配列を示していますが、これが非常に大きかったらどうなるか想像してみてください。

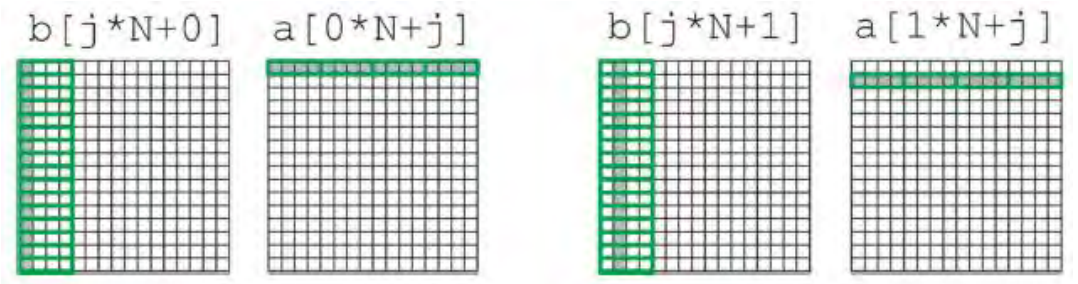


図 11-32.　行列 a の最初の 2 行を転置する際にアクセスされるキャッシュライン。簡単にするため、各キャッシュラインに 4 つの項目を表示しています

このアルゴリズムのキャッシュ非依存の実装により、同じキャッシュラインまたはデータ項目の再利用でアクセスされるデータ量が削減されます。図 11-33 に示すように、行列 a の他のブロックに移る前に、行列 a の小さなブロックのみを転置することに焦点を当てると、行列全体の大きさに関係なく、キャッシュラインの再利用によるパフォーマンスの向上のためキャッシュに保持する必要がある、b の要素を保持するキャッシュラインの数を減らすことができます。

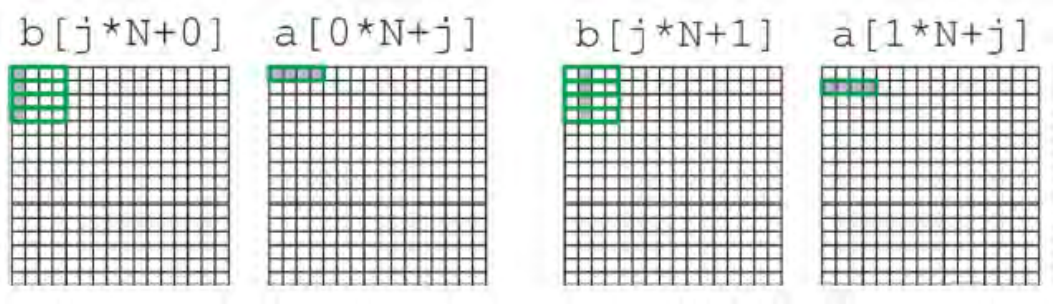


図 11-33. 一度にブロックを転置すると、再利用のメリットを得るために保持するキャッシュラインの数が減ります

キャッシュ非依存の行列転置のシリアル実装を図 11-34 に示します。これは、問題を i 次元と j 次元に沿って再帰的に細分化し、レンジがしきい値を下回ったときにシリアル for ループを使用します。

```
void  obliviousTranspose(int  N, int  ib, int  ie, int  jb, int  je,
                         double  *a, double  *b, int  gs) {
  int  ilen = ie-ib;
  int  jlen = je-jb;
  if  (ilen > gs || jlen > gs) {
     if  ( ilen > jlen ) {
        int  imid = (ib+ie)/2;
        obliviousTranspose(N, ib, imid, jb, je, a, b, gs);
        obliviousTranspose(N, imid, ie, jb, je, a, b, gs);
     } else  {
       int  jmid = (jb+je)/2;
        obliviousTranspose(N, ib, ie, jb, jmid, a, b, gs);
        obliviousTranspose(N, ib, ie, jmid, je, a, b, gs);
     }
  } else  {
    for  (int  i = ib; i < ie; ++i) {
      for  (int  j = jb; j < je; ++j) {
        b[j*N+i] = a[i*N+j];
      }
    }
  }
}
```

図 11-34. キャッシュ非依存の行列転置のシリアル実装。サンプルコード: performance_tuning/parallel_for_transpose_partitioners.cpp

実装では i 方向と j 方向の分割が交互に実行されるため、行列 a は図 11-35 に示す走査パターンを使用して転置され、最初にブロック 1 が完了し、次にブロック 2、ブロック 3 というように続きます。gs が 4 でキャッシュライン・サイズが 4 の場合、図 11-33 に示した各ブロック内での再利用が実現します。しかし、キャッシュラインが 4 項目ではなく 8 項目の場合（実際のシステムではその可能性がはるかに高くなります）、最小のブロック内だけでなく、ブロック間でも再利用が可能になります。例えば、データキャッシュがブロック 1 と 2 の間にロードされたすべてのキャッシュラインを保持できる場合、ブロック 3 と 4 を転置するときにこれらのキャッシュラインが再利用されます。

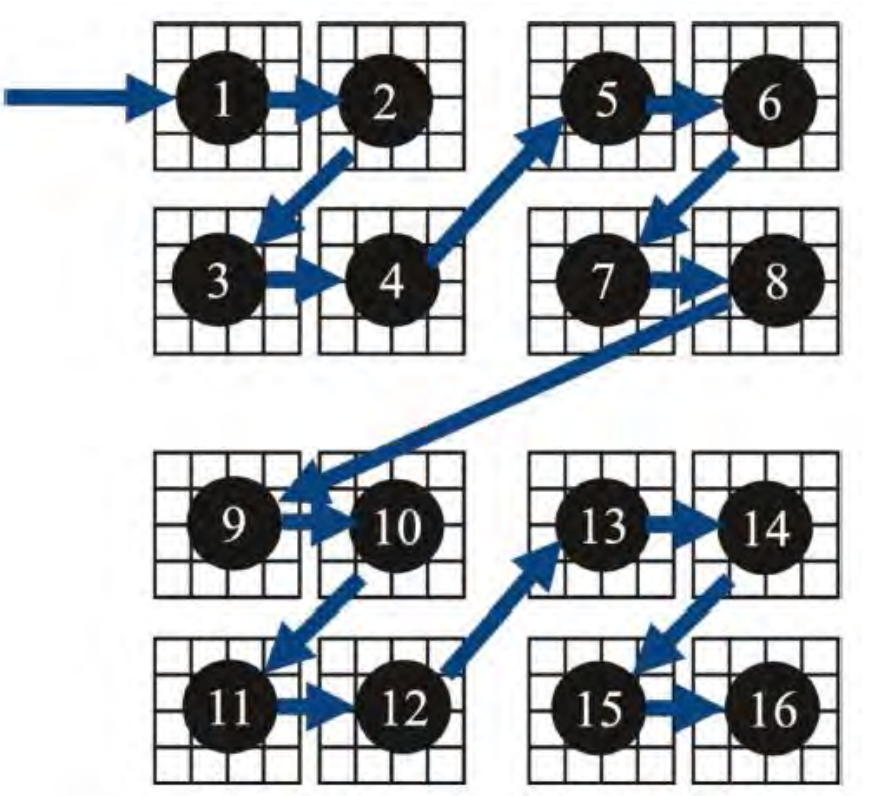


図 11-35. 他のブロックに移動する前に、サブブロックの転置を計算する走査パターン

これがキャッシュ非依存アルゴリズムの真の威力です。メモリー階層の各レベルのサイズを正確に知る必要はありません。サブ問題が小さくなるにつれて、メモリー階層のより小さな部分に収まるようになり、各レベルでの再利用性が向上します。

TBB ループ・アルゴリズムと TBB スケジューラーは、特にキャッシュ非依存アルゴリズムをサポートするように設計されています。したがって、図 11-36 に示すように、`parallel_for`、`blocked_range2d`、および `simple_partitioner` を使用して、キャッシュ非依存の並列行列転置実装を迅速に実装できます。反復空間を 2 次元ブロックに分割する場合、`blocked_range2d` を使用します。また、ブロックがキャッシュサイズよりも小さいサイズに分割されている場合にのみ再利用のメリットが得られるため、`simple_partitioner` を使用します。他のパーティショナー・タイプは負荷分散を最適化するため、負荷分散に十分であればより大きなレンジサイズを選択することがあります。

```
template<typename  P>
double  pforTranspose2d(int  N, double  *a, double  *b, int  gs) {
   tbb::tick_count t0 = tbb::tick_count::now();
   tbb::parallel_for( tbb::blocked_range2d<int,int>{
                         0, N, static_cast<size_t>(gs),
                         0, N, static_cast<size_t>(gs)},
     [N, a, b](const  tbb::blocked_range2d<int,int>&  r) {
       for  (int  i = r.rows().begin(); i < r.rows().end(); ++i) {
         for  (int  j = r.cols().begin(); j < r.cols().end(); ++j) {
           b[j*N+i] = a[i*N+j];
         }
       }
     }, P()
   );
   tbb::tick_count t1 = tbb::tick_count::now();
   return  (t1-t0).seconds();
}
```

図 11-36. blocked_range2d、粒度（gs）、およびパーティショナーをテンプレート引数 P として使用する、キャッシュ非依存の行列転置の並列実装。サンプルコード: performance_tuning/parallel_for_transpose_partitioners.cpp

図 11-37 は、TBB の parallel_for がレンジを再帰的に細分化する方法によって、キャッシュ非依存の実装に必要なブロックが作成されることを示しています。TBB スケジューラーの深さ優先ワークと幅優先スチール動作は、ブロックが図 11-35 に示す順序と同様の順序で実行されることも意味します。

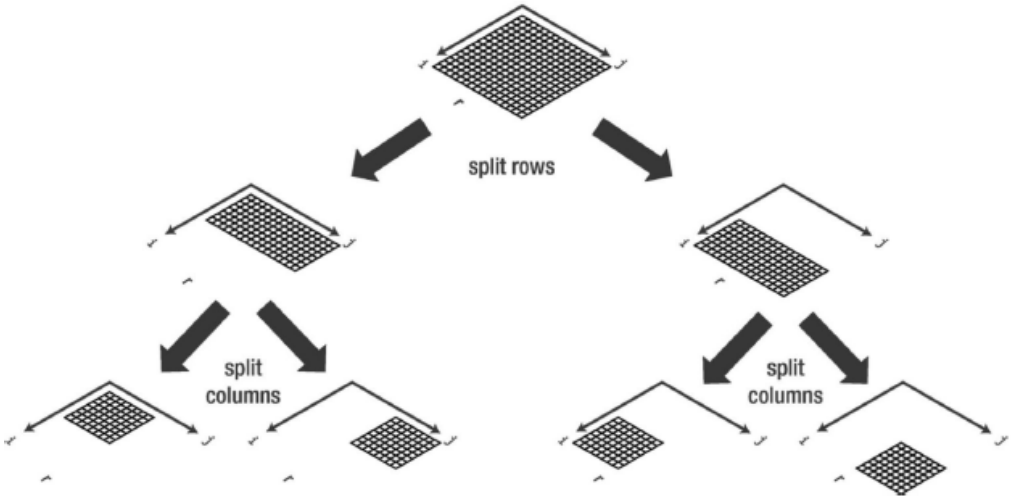
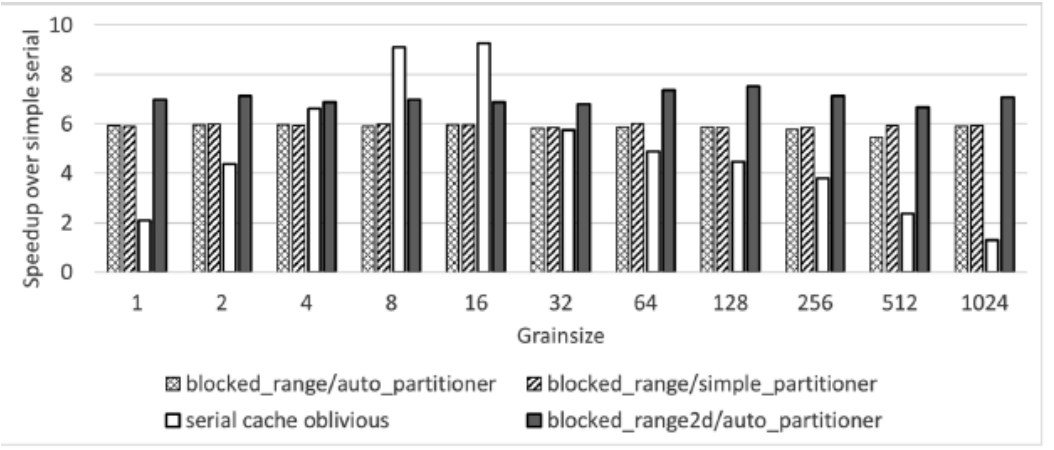


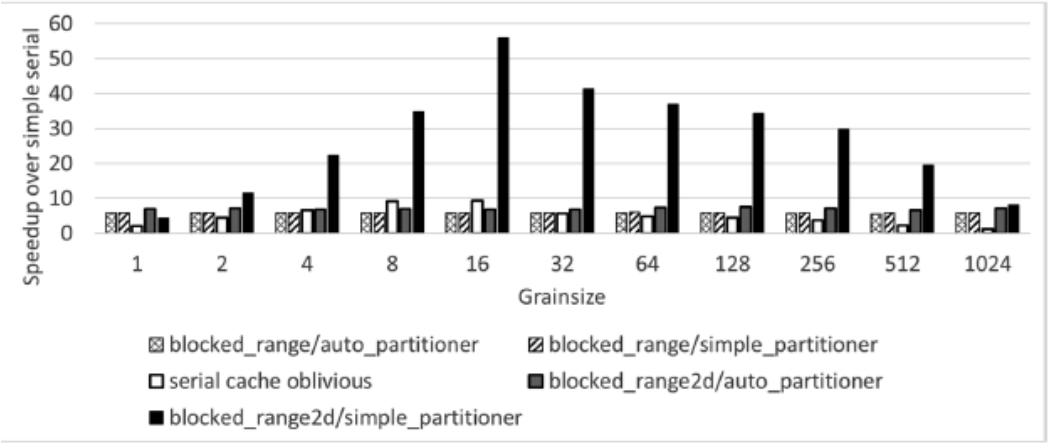
図 11-37. `blocked_range2d` の再帰的細分化は、キャッシュ非依存の並列実装に必要なブロックに一致する分割を提供します

図 11-38 は、図 11-34 のシリアルなキャッシュ非依存の実装のパフォーマンス、`1D blocked_range` を使用した実装のパフォーマンス、および図 11-36 の実装と同様の `blocked_range2d` 実装のパフォーマンスを示しています。図 11-36 に示すように、粒度とパーティショナーを簡単に変更できるように並列バージョンを実装しました。

図 11-38 では、図 11-31 の単純なシリアル実装と比較して、32768 × 32768 の行列を使用した Linux サーバーシステムでの実装のスピードアップを示しています。



(a) The serial cache-oblivious algorithm outperforms most parallel versions



(b) But `blocked_range2d` with simple partitioner shows better performance

図 11-38.　さまざまな粒度とパーティショナーを使用した `N = 32,768` のテストマシンでのスピードアップ。これらの結果は、この章の前半で説明した Linux サーバーシステムで収集しました。`tbb::task_arena` 制約を使用して、計算をコアごとに 1 つのスレッドと最大 8 つのハードウェア・スレッドの同時実行を持つ単一の NUMA ノードに制限しました

行列の転置は、データの読み取りと書き込みの速度によって制限されます。計算は全く行われません。図 11-38 から、粒度に関係なく、ほとんどの並列実装では 8 スレッドで 8 のスピードアップには到達しないことが分かります。スピードアップはメモリー帯域幅によって制限されます。

私たちのシリアルのキャッシュ非依存アルゴリズムは、メモリーアクセスを並べ替え、キャッシュミスの数を減らします。シンプルなバージョンや大部分の並列実装よりも大幅に優れたパフォーマンスを発揮します。並列実装で blocked_range2d を使用すると、同様に 2D の分割が得られます。しかし、図 11-38 に見られるように、simple_partitioner を使用した場合にのみ、完全にキャッシュ非依存のアルゴリズムのように動作します。実際、blocked_range2d と simple_partitioner を使用したキャッシュを無視した並列アルゴリズムは、メモリー階層への負荷を大幅に軽減し、8 スレッドのみを使用した場合でも、シリアル実装に比べて 50 倍以上のスピードアップを実現します。

すべての問題にキャッシュ非依存の解決策があるわけではありませんが、多くの一般的な問題には存在します。キャッシュ非依存の解決策が可能で、かつ価値があるかどうかを調べるため、問題を調査する価値は十分にあります。もしそうであれば、ブロック化されたレンジ型と simple_partitioner により、TBB アルゴリズムを使用して実装することが非常に簡単になります。

キャッシュ・アフィニティー

キャッシュ非依存アルゴリズムは、データの局所性があってもキャッシュに適合しない問題を、キャッシュに適合する小さな問題に分割することで、キャッシュのパフォーマンスを向上させます。対照的に、キャッシュ・アフィニティーは、すでにキャッシュに収まっているデータにまたがるレンジの繰り返し実行に対処します。データはキャッシュに収まるため、後続の実行で同じサブレンジが同じプロセッサーに割り当てられていると、キャッシュされたデータに素早くアクセスできます。TBB ループ・アルゴリズムのキャッシュ・アフィニティーを有効にするには、affinity_partitioner または static_partitioner のいずれかを使用できます。図 11-39 は、1 次元配列の各要素に値を加算する単純なマイクロベンチマークを示しています。この関数はパーティショナーへの参照を受け取ります。これは、affinity_partitioner オブジェクトに履歴を記録するためパーティショナーを参照として受け取る必要があるためです。

```
template <typename Partitioner>
void parForAdd(double v, int N, double *a, Partitioner& p) {
   tbb::parallel_for( tbb::blocked_range<int>(0, N, 1),
    [v, a](const tbb::blocked_range<int>& r) {
       for (int i = r.begin(); i < r.end(); ++i) {
         a[i] += v;
       }
    }, p
  );
}
```

図 11-39. TBB parallel_for を使用して 1D 配列のすべての要素に値を加算する関数。サンプルコード: performance_tuning/parallel_for_addition_partitioners.cpp

キャッシュ・アフィニティーの影響を確認するには、この関数を繰り返し実行し、N に同じ値と同じ配列 a を渡します。auto_partitioner を使用する場合、スレッドへのサブレンジのスケジュールは呼び出しごとに異なります。配列 a がプロセッサーのキャッシュに完全に収まる場合でも、後続の実行では a の同じ領域が同じプロセッサーに収まらない可能性があります。

```
for (int i = 0; i < M; ++i) {
   parForAdd(v[i], N, a,
   tbb::auto_partitioner{});
}
```

ただし、affinity_partitioner を使用すると、TBB ライブラリーはタスクのスケジュールを記録し、実行ごとにそれを再作成しようとします。履歴はパーティショナーに記録されるため、後続の実行で同じパーティショナー・オブジェクトを渡す必要があり、auto_partitioner のように単純に一時オブジェクトを作成することはできません。

```
tbb::affinity_partitioner aff_p;
for (int i = 0; i < M; ++i) {
  parForAdd(v[i], N, a, aff_p);
}
```

最後に、static_partitioner を使用してキャッシュ・アフィニティーを作成することもできます。static_partitioner を使用する場合、スケジュールは決定論的であるため、実行ごとに同じパーティショナー・オブジェクトを渡す必要はありません。

```
for (int i = 0; i < M; ++i) {
    parForAdd(v[i], N, a,
    tbb::static_partitioner{});
}
```

このマイクロベンチマークは、N = 1,000,000、M = 10,000 で 8 つのスレッドのみを使用するように構成されたテストマシンで実行されました。2 つの配列のサイズは 1,000,000 × 8 = 8MB です。テストマシンには、コアごとに 1 つずつ、2 MB の L2 データキャッシュがあります。affinity_partitioner を使用すると、auto_partitioner よりも 1.5 倍速くテストが完了しました。static_partitioner を使用すると、auto_partitioner よりも 3 倍速くテストが完了しました。

データは L2 キャッシュサイズ（8 × 2MB = 16MB）に収まるため、同じスケジュールを再生すると実行時間に大きな影響を与えます。次の節では、static_partitioner が auto_partitioner よりも優れたパフォーマンスを示した理由と、それがそれほど驚くべきことではない理由について説明します。N を 10,000,000 要素に増やすと、配列 a が大きくなりすぎてテストシステムのキャッシュに収まらなくなるため、実行時間に大きな違いはなくなります。この場合、キャッシュの局所性を活用するためタイリング/ブロッキングを実装するアルゴリズムを再考する必要があります。

static_partitioner を使用

static_partitioner は、オーバーヘッドが最も低いパーティショナーであり、アリーナ内のスレッド全体にブロックされたレンジを均一に分散させます。パーティショニングは決定論的であるため、ループまたは一連のループが同じレンジで繰り返し実行されるキャッシュ動作も改善されます。 前の節では、マイクロベンチマークでは、affinity_partitioner よりも大幅に優れたパフォーマンスを示すことを確認しました。ただし、アリーナ内の各スレッドに 1 つずつ提供するのに十分なチャンクを作成するため、ワークスチールによって負荷を動的に分散する余地はありません。実際には、static_partitioner は TBB ライブラリーのワークスチールのスケジュール・アプローチを無効にします。

しかし、TBB に static_partitioner が含まれているのには十分な理由があります。コア数が増えると、特にアプリケーションのシリアル部分から並列部分に移行する、ランダム・ワークスチールのコストが高くなります。マスタースレッドがアリーナに初めて新しい処理を生成すると、すべてのワーカースレッドが起動し、群れのように処理すべき処理を探し始めます。さらに悪いことに、ワーカースレッドはどこを探せばいいのか分からず、マスタースレッドのデックだけでなく、互いのローカルデックもランダムに検索し始めます。最終的に、あるワーカースレッドがマスター内のワークを見つけてそれを細分化し、別のワーカースレッドが最終的にこの細分化された部分を見つけてそれを細分化する、という動作が行われます。そしてしばらくすると、事態は落ち着き、すべてのワーカーは何かワークを見つけ、自分のローカルデックで動作するようになります。

しかし、ワークロードのバランスが取れていて、システムがオーバーサブスクライブされておらず、すべてのコアが同等の性能であることが判明している場合、ワーカー間で均一に分散するためだけに、このようなワークスチールのオーバーヘッドが本当に必要なのでしょうか？ static_partitioner を使用する場合はそうではありません。
このようなケースのために設計されています。レンジを均一に分散するタスクをワーカースレッドにプッシュするので、ワーカースレッドがタスクをスチールする必要がありません。適用される場合、static_partitioner はループを分割する最も効率的な方法です。

しかし、static_partitioner にあまり依存しすぎないでください。ワークロードが均一でない場合、またはいずれかのコアが追加スレッドでオーバーサブスクライブされている場合、static_partitioner を使用するとパフォーマンスが低下する可能性があります。例えば、図 11-40 は、粒度がパフォーマンスに与える影響を調べる、図 11-29 で使用したものと同じマイクロベンチマーク構成を示しています。しかし、他のデスクトップ・アプリを実行しながらラップトップ・システムで実行すると何が起こるかを示します。

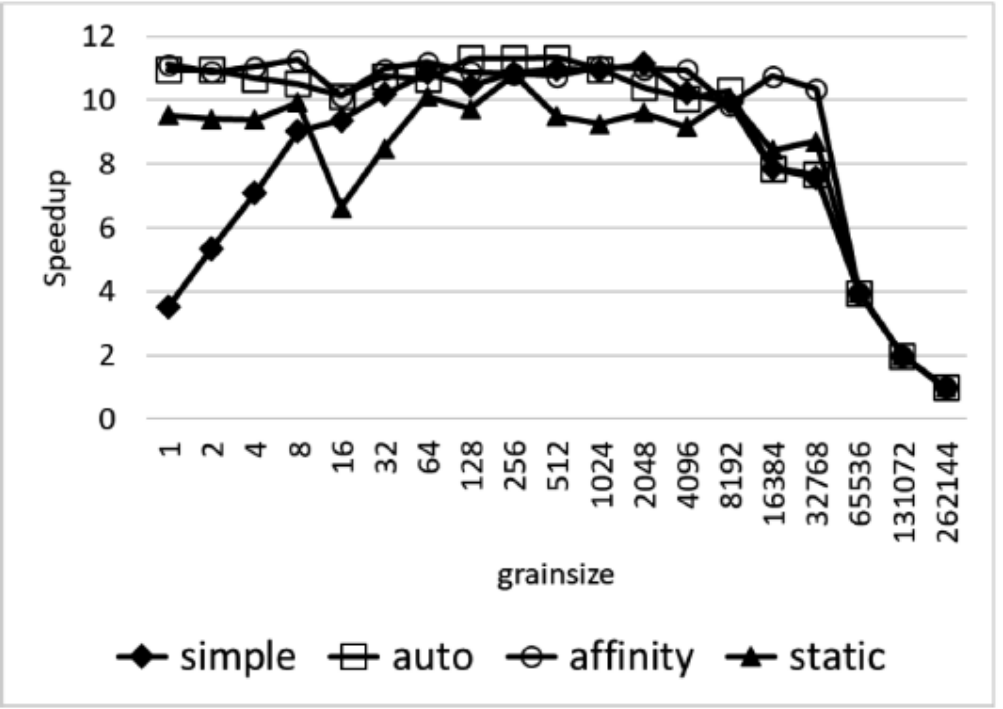


図 11-40.　さまざまなパーティショナー・タイプのシリアル実装に比べてスピードアップされ、追加のスレッドがバックグラウンドでスピンループを実行するときの粒度が増加します。反復あたりの時間は 100ns に設定されています

図 11-40 では、simple_partitioner は粒度が大きくなるにつれて通常の改善を示しています。また、負荷を柔軟に分散できるように粒度を自動選択するパーティショナー（auto_partitioner と affinity_partitioner）は、システムへの余分な負荷に耐性があり、12 論理コアシステムで 12 倍近くのスピードアップを実現します。static_partitioner は同じように機能しません。すべてのコアが同等の能力を持ち、同等にワークに参加できると想定し、それらの間でワークを均一に分散します。私たちのラップトップ・システムでは、他のアプリケーションも実行されており、E-core と P-core も混在しています。

このシナリオでは、コアの能力は同等ではなく、負荷のため同等に利用できない可能性があります。その結果、static_partitioner のスピードアップは他のパーティショナーよりも低く、安定性も低くなります。

static_partitioner が影響を受けるのはシステムだけではありません。図 11-41 は、反復ごとにワークが増加するループを示しています。static_partitioner を使用すると、反復回数が最も少ないスレッドのワーク量は、反復回数が最も多い不運なスレッドのワーク量よりもはるかに少なくなります。

```
void  doWork(double  usec) {
   double  sec  = usec*1e-06;
   tbb::tick_count t0 = tbb::tick_count::now();
   while  ((tbb::tick_count::now() -  t0).seconds() <= sec);
}

template  <typename  Partitioner>
void  buildingWork(int  N, Partitioner&  p) {
   tbb::parallel_for( tbb::blocked_range<int>(0, N, 1),
     [](const  tbb::blocked_range<int>&  r) {
       for  (int  i = r.begin(); i < r.end(); ++i) {
          doWork(i);
       }
     }, p
   );
}
```

図 11-41. 反復ごとにワークが増加するループ。サンプルコード: performance_tuning/partitioners_imbalanced_loops.cpp

図 11-41 のループを、Linux サーバーシステムで、N = 1,000 の各パーティショナー・タイプを使用して 10 回実行しました。いつものように、task_arena 制約を使用してアプリケーションを 8 スレッドに制限しました。結果は予想どおりです。この例では static_partitioner のパフォーマンスが低いことを示しています:

```
auto_partitioner = 0.629797 seconds
affinity_partitioner = 0.629912
secondsstatic_partitioner = 1.17187 seconds
```

auto_partitioner と affinity_partitioner はスレッド間で負荷を再分散しますが、static_partitioner は初期の均一ですが不公平な分散のままです。

したがって、static_partitioner は、ハイパフォーマンス・コンピューティング（HPC）アプリケーションでのみ有効です。これらのアプリケーションは、多数のコアを持つシステム上で実行され、多くの場合、一度に 1 つのアプリケーションが起動されるバッチモードで実行されます。ワークロードに動的な負荷分散が必要ない場合、static_partitioner は他のパーティショナーよりも優れたパフォーマンスを発揮します。バランスの取れたワークロードとシングルユーザーのバッチモード・システムは例外であり、一般的ではありません。

決定論的なスケジューラーの制限

2 章では、結合性と浮動小数点型について説明しました。浮動小数点数の実装はすべて近似値であるため、結合性や可換性などの特性に依存する場合、並列処理によって異なる結果が生じる可能性に注意しました。これらの結果は必ずしも誤りではなく、単に異なるだけです。それでも、リダクションの場合、TBB は、同じマシンで実行したときに、同じ入力データの各実行で同じ結果が得られるよう、parallel_deterministic_reduce アルゴリズムを提供します。

想像のとおり、parallel_deterministic_reduce は、両方のパーティショナー・タイプでサブレンジ数が決定論的であるため、simple_partitioner または static_partitioner のみを受け入れます。また、parallel_deterministic_reduce は、実行に動的に参加するスレッドの数や、タスクがスレッドにどのように割り当てられるかに関係なく、特定のマシン上で常に同じ分割操作と結合操作のセットを実行しますが、parallel_reduce アルゴリズムではそうならないことがあります。その結果、parallel_deterministic_reduce は同じマシンで実行すると常に同じ結果を返しますが、それにはある程度の柔軟性が犠牲になります。

図 11-42 は、parallel_reduce と parallel_deterministic_reduce を使用して実装した、2 章の pi 計算例のスピードアップを示しています。この例では、ラップトップ・システムで測定値を収集しました。tbb::task_arena 制約を使用してアプリケーションを構成し、8 つの E-core のみを使用してこのアプリケーションを実行しました。他のアプリケーションをすべて終了しなかったため、いくつかのデスクトップ・アプリケーションがアクティブなままでした。これを行ったのは、静的パーティショニングの限界を再度強調するためです。

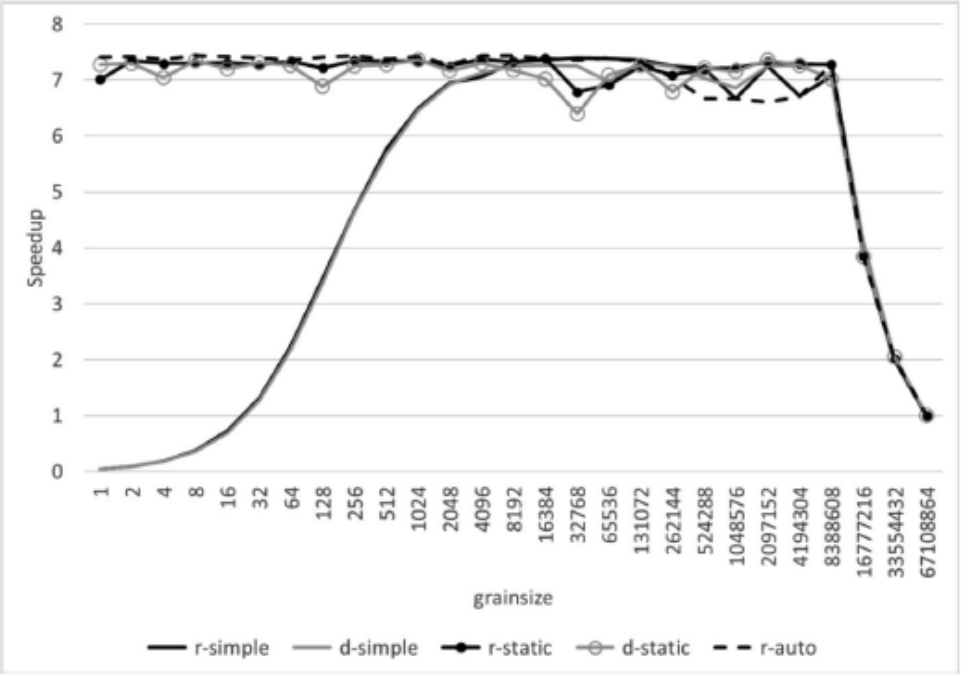


図 11-42. 2 章の pi の例を、auto_partitioner (r-auto)、simple_partitioner (r-simple)、static_partitioner (r-static) を使用した parallel_reduce と、simple_partitioner (d-simple) と static_partitioner (d-static) を使用した parallel_deterministic_reduce を使用してスピードアップします

最大のスピードアップは、reduce の決定論的バージョンと非決定論的バージョンの両方で同様です。ただし、auto_partitioner は parallel_reduce で優れたパフォーマンスを発揮しますが、parallel_deterministic_reduce ではこのオプションは使用できません。parallel_deterministic_reduce ではすべての分割と結合を実行する必要があるため、オーバーヘッドが発生しますが、このオーバーヘッドは通常は小さいものです。それより大きな制限は、チャンクサイズを自動的に検出するパーティショナーを使用できないことです。

他のアプリケーションで少し負荷をかけたシステムでは、大きな粒度によって柔軟性が制限されるポイントに達するまで、r-auto（auto_partitioner を使用した parallel_reduce）がピーク時のスピードアップに近い状態を維持することが分かります。予想どおり、十分な粒度を使用すると、r-simple（simple_partitioner を使用した parallel_reduce）と d-simple（simple_partitioner を使用した parallel_deterministic_reduce）の両方で良好なパフォーマンスを発揮します。この 2 つの差はごくわずかであり、決定論を追加してもパフォーマンスに大きな影響はありません。r-static（static_partitioner を使用した parallel_reduce）と d-static（static_partitioner を使用した parallel_deterministic_reduce）は、どちらも粒度が小さい場合はパフォーマンスが低下しませんが、粒度が大きくなるにつれてピーク時のスピードアップを維持するのが難しくなることに注意してください。

これらのパーティショナーは、すべてのスレッドが平等に参加できると想定して均一な分散を行いますが、負荷の低いシステムでは、実行されるコアが他のアプリケーションに関連するワークに関係している可能性があるため、平等に参加できません。

この例から得られる教訓は 2 つあります。まず、`parallel_deterministic_reduce` を使用しても、通常は大きなオーバーヘッドは課されません。2 番目に、`parallel_deterministic_reduce` を使用するには、決定論的パーティショナーを使用する必要があります。つまり、`simple_partitioner` で使用して十分に大きな粒度を選択するか、`static_partitioner` を使用してワークスチールの利点の一部を失うかです。

TBB パイプラインのチューニング: フィルター、モード、トークン数

ループのアルゴリズムと同様に、TBB パイプラインのパフォーマンスは、粒度、局所性、および利用可能な並列性に影響されます。ループのアルゴリズムとは異なり、TBB パイプラインはレンジとパーティショナーをサポートしません。代わりに、パイプラインの調整に使用されるコントロールには、フィルターの数、フィルター実行モード、およびパイプラインの実行時にパイプラインに渡されるトークンの数が含まれます。

TBB のパイプライン・フィルターはタスクとして生成され、TBB ライブラリーによってスケジュールされるため、ループ・アルゴリズムによって作成されるサブレンジと同様に、フィルターボディーがオーバーヘッドを軽減するのに十分な長さで実行される必要がありますが、十分な並列処理も必要です。 ここでは、ワークをフィルターに分割する方法によって、バランスを取っています。最も低速なシリアルステージがボトルネックとなるため、フィルターの実行時間も適切にバランスが取れている必要があります。

2 章で説明したように、パイプライン・フィルターも実行モード（`serial_in_order`、`serial_out_of_order`、または `parallel`）を使用して作成されます。`serial_in_order` モードを使用する場合、フィルターは一度に 1 つの項目を処理でき、最初のフィルターが生成した順序で項目を処理する必要があります。`serial_out_of_order` フィルターでは、任意の順序で項目を実行できます。並列フィルターは、異なる項目を並列に実行できます。これらのモードがパフォーマンスをどのように制限するか、このセクションの後半で説明します。

実行時には、TBB パイプラインに `max_number_of_live_tokens` 引数を指定する必要があります。これにより、特定の時点でパイプラインを通過できる項目数を制限します。

図 11-43 は、これらのコントロールを調査するのに使用するマイクロベンチマークの構造を示しています。図の両方のパイプラインには 8 つのフィルターが表示されていますが、実験ではこの数を変更します。一番上のパイプラインには同じ実行 `mode` を使用するフィルターがあり、すべて同じ `spin_time` を持っているため、バランスの取れたパイプラインを表しています。下のパイプラインには、`imbalance * spin_time` の間スピンするフィルターが 1 つあります。このインバランス係数を変更して、インバランスがスピードアップに与える影響を確認します。

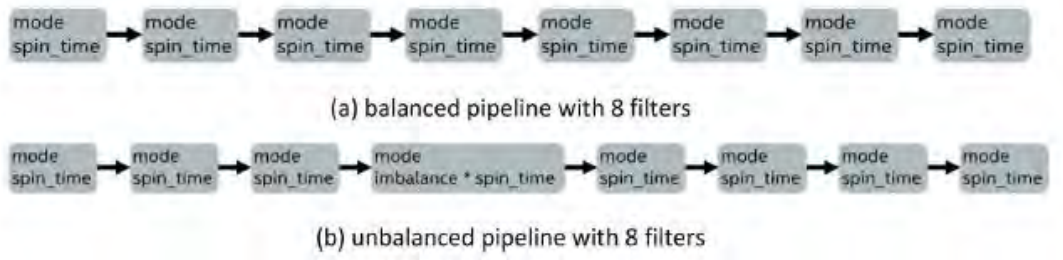


図 11-43. バランスの取れたパイプライン・マイクロベンチマークとインバランスなパイプライン・マイクロベンチマーク

バランスの取れたパイプラインを理解する

まず、タスクサイズに関する経験則がパイプラインにどの程度当てはまるかを考えてみましょう。1 マイクロ秒のフィルターボディーはオーバーヘッドを軽減するのに十分ですか？ 図 11-44 は、1 つのトークンのみを使用して `8,000` 個の項目を入力したバランスの取れたパイプライン・マイクロベンチマークのスピードアップを示しています。各種フィルターの実行時間の結果が表示されます。トークンは 1 つしかないため、一度にパイプラインを通過できるのは 1 つの項目だけです。結果は、パイプラインのシリアル実行になります（フィルター実行モードが並列に設定されている場合でも）。

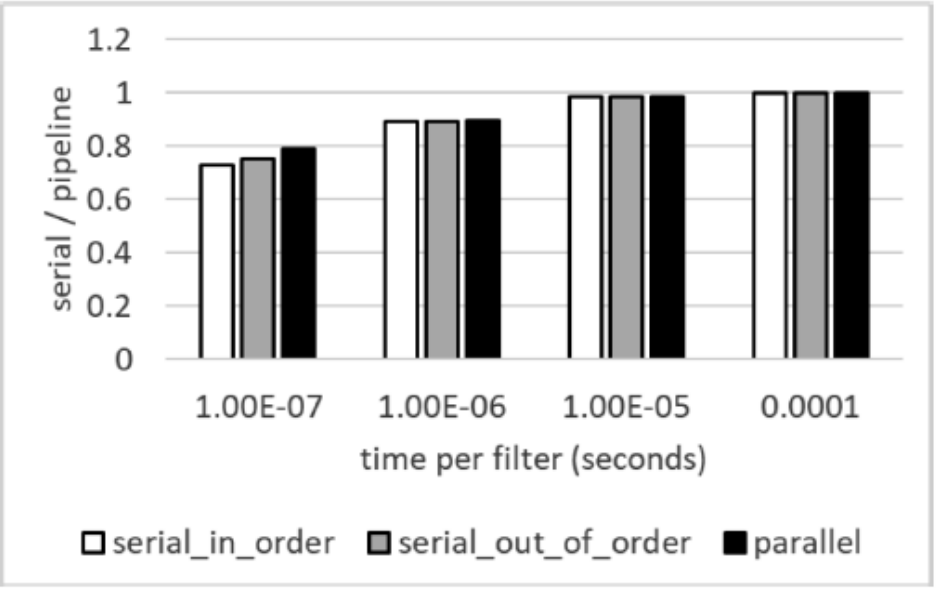


図 11-44. Linux サーバーシステムで 8 つのフィルター、単一のトークン、8,000 個の項目を含むバランスの取れたパイプラインを実行する際に、さまざまなフィルター実行モードで発生するオーバーヘッド。サンプルコード: `performance_tuning/parallel_pipeline_timed.cpp`

for ループで適切な数のスピンを実行する真のシリアル実行と比較すると、ワークを TBB パイプラインとして管理する影響が分かります。図 11-44 では、`spin_time` が 1 マイクロ秒に近づくとオーバーヘッドがかなり低くなり、シリアル実行の実行時間に近づくことが分かります。私たちの経験則は TBB パイプラインにも当てはまるようです。

それでは、フィルター数がパフォーマンスにどのように影響するかを見てみましょう。シリアル・パイプラインでは、並列性は異なるフィルターをオーバーラップすることによってのみ生じます。並列フィルターを備えたパイプラインでは、異なる項目に対して並列フィルターを同時に実行することでも並列性が実現されます。ターゲット・プラットフォームは 8 つのスレッドをサポートしているため、並列実行では最大 8 倍のスピードアップが期待できます。

図 11-45 は、トークン数を 8 に設定したバランス型パイプライン・マイクロベンチマークのスピードアップを示しています。どちらのシリアルモードでも、フィルター数に応じてスピードアップします。これは覚えておくべき重要なことです。シリアル・パイプラインのスピードアップは、TBB ループ・アルゴリズムのようにデータセットのサイズに比例してスケールするわけでないからです。しかし、すべての並列フィルターを含むバランス型パイプラインでは、フィルターが 1 つしかなくても 8 倍のスピードアップが実現できます。これは、8,000 個の入力項目を単一のフィルターで並列処理できるためです。ボトルネックとなるシリアルフィルターはありません。

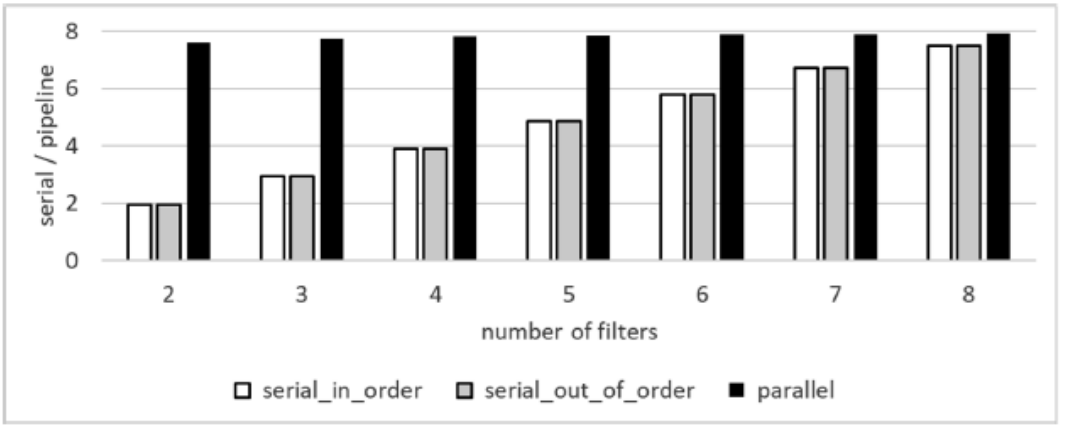


図 11-45. 8 つのトークン、8,000 個の項目、およびフィルター数が増加するバランスの取れたパイプラインを実行するときに、さまざまなフィルター実行モードで達成されるスピードアップ。フィルターは 100 マイクロ秒スピンします。Linux サーバーシステムで収集されました。サンプルコード: `performance_tuning/parallel_pipeline_timed.cpp`

図 11-46 では、8 つのフィルターを使用しながらトークンの数を変えたバランス型パイプラインのスピードアップが分かります。ここで使用するプラットフォームには 8 つのスレッドがあるため、トークンが 8 個未満の場合、すべてのスレッドをビジー状態に保つ項目が不足します。パイプラインに少なくとも 8 つの項目があれば、すべてのスレッドが参加できるようになります。トークン数を 8 より増やしても、パフォーマンスにはほとんど影響がありません。

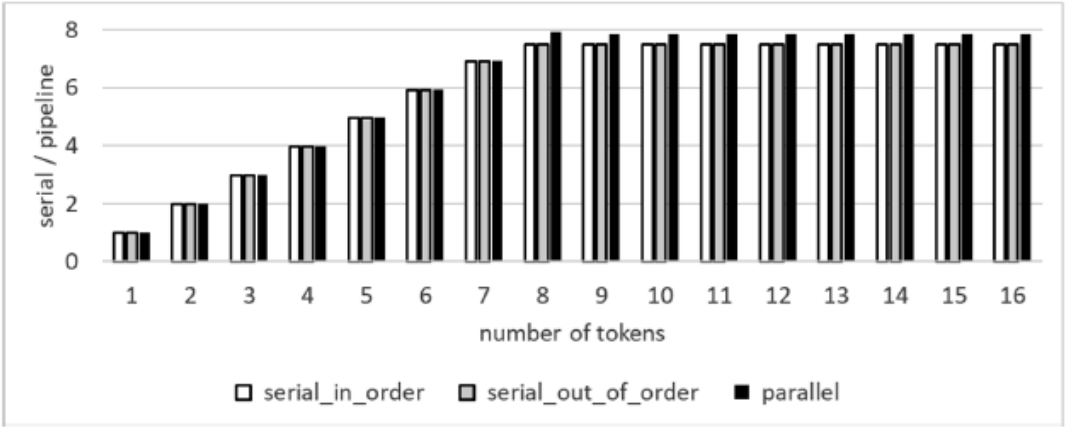


図 11-46. 8 つのフィルター、8,000 個の項目、およびトークン数が増加するバランスの取れたパイプラインを実行するときに、さまざまなフィルター実行モードで達成されるスピードアップ。フィルターは 100 マイクロ秒スピンします。Linux サーバーシステムで収集されました。サンプルコード: `performance_tuning/parallel_pipeline_timed.cpp`

インバランスなパイプラインを理解する

ここで、図 11-43 のインバランスなパイプラインのパフォーマンスを見てみましょう。このマイクロベンチマークでは、`spin_time * imbalance` 秒間スピンするフィルターを除き、すべてのフィルターが `spin_time` 秒間スピンします。したがって、8 つのフィルターを備えたインバランスなパイプラインを通過する `N` 個の項目を処理するのに必要な時間は次のようになります。

$$T_1 = N * \left(7 * spin\_time + spin\_time * imbalance\right)$$

安定状態では、シリアル・パイプラインは最も遅いシリアルステージによって制限されます。安定状態では、パイプラインの増加または減少にかかる時間は無視されるため、各フィルターは最大速度で実行されます。インバランスなフィルターがシリアルモードで実行されていても、すべてのフィルターの実行をオーバーラップさせるのに十分なスレッドがある場合、同じパイプラインの安定状態時間を次のようにモデル化できます。

$$T_\infty = N * \max\left(spin\_time, spin\_time * imbalance\right)$$

図 11-47 は、さまざまなインバランス係数を使用してテスト・プラットフォームで実行した、インバランスなパイプラインの結果を示しています。また、次のように計算される理論上の最大スピードアップ T_1/T_∞ も示します。

$$Speedup_{\max} = \frac{7 * \text{spin\_time} + \text{spin\_time} * \text{imbalance}}{\max\left(\text{spin\_time},\ \text{spin\_time} * \text{imbalance}\right)}$$

予想通り、図 11-47 はシリアル・パイプラインが最も低速なフィルターによって制限されることを示し、測定結果は T_1/T_∞ 計算で予測される結果に近いものになっています。

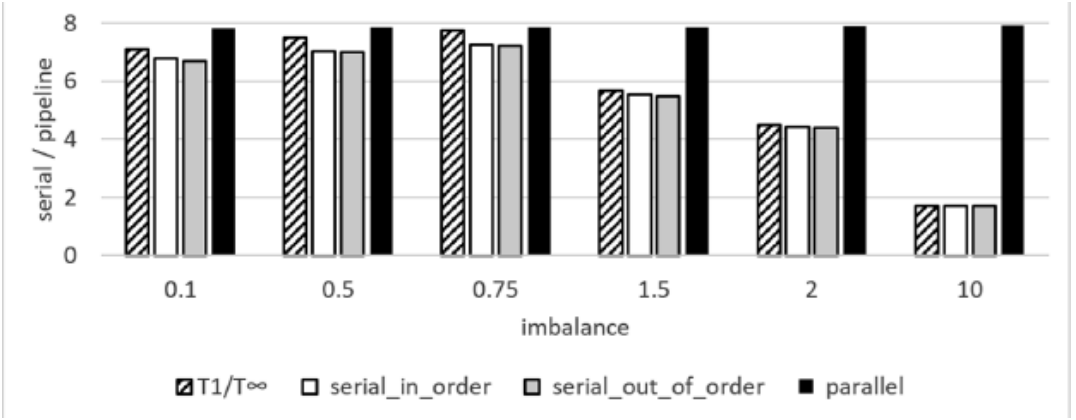


図 11-47. 8 つのフィルター、8,000 個の項目、および異なるインバランス係数を持つインバランスなパイプラインを実行するときに、さまざまなフィルター実行モードで達成されるスピードアップ。フィルターのうち 7 つは 100 マイクロ秒スピンし、残りのフィルターは `imbalance` × 100 マイクロ秒スピンします

対照的に、フィルターに `parallel` モードを使用する `parallel_pipeline` は、TBB スケジューラーが最も遅いフィルターの実行と同じフィルターの他の呼び出しとオーバーラップできるため、最も低速なステージによって制限されないことが分かります。トークン数を 8 つ以上に増やす効果を疑問に思うかもしれませんが、今回はそうではありません。テストシステムには 8 スレッドしかないため、最も遅いフィルターのインスタンスを最大 8 個オーバーラップできます。一時的な負荷インバランスは、スレッド数よりも多くのトークンを使用して平滑化できる場合がありますが、インバランスが定数要素となるマイクロベンチマークは、最も低速なステージとスレッド数によって制限されており、トークン数をいくら増やしても変わりません。

ただし、トークン数が不十分であるためワークスチール TBB スケジューラーの自動負荷分散機能が妨げられるアルゴリズムもあります。これは、ステージのバランスが悪く、連続ステージによってパイプがストールしている場合に発生します。

パイプラインとデータの局所性とスレッドのアフィニティー

TBB ループ・アルゴリズムでは、ブロック・レンジ・タイプ、`affinity_partitioner`、`static_partitioner` を使用してキャッシュのパフォーマンスを調整しました。TBB の `parallel_pipeline` 関数には同様のオプションはありません。しかし、すべてが失われたわけではありません! TBB の `parallel_pipeline` にビルトインされた実行順序は、特別な操作を必要とせずに時間的なデータの局所性を強化するように設計されています。

TBB マスターまたはワーカースレッドが TBB フィルターの実行を完了すると、実行モードの制約によってフィルターを実行できない場合を除き、パイプライン内の次のフィルターを実行します。例えば、フィルター f_0 が項目 i を生成し、その出力が次のフィルター f_1 に渡される場合、f_0 を実行したスレッドが f_1 の実行に進みます。ただし、次のフィルターが serial_out_of_order フィルターで、現在何か他を処理中であるか、serial_in_order フィルターで、項目 i が次の項目ではない場合を除きます。この場合、項目は次のフィルターにバッファリングされ、スレッドは他の処理を探します。それ以外は、スレッドは局所性を最大化するため、生成したばかりのデータを次のフィルターで実行して項目を処理します。

内部的にフィルター f_0 内の 1 つの項目の処理は、スレッド/コアで実行されるタスクとして実装されています。フィルターが完了すると、タスクはリサイクルされて次のフィルター f_1 を実行します。本質的に、終了するタスク f_0 はスケジューラーをバイパスして新しい f_1 タスクになります。つまり、f_0 を実行した同じスレッド/コアが f_1 も実行します。データの局所性とパフォーマンスの点では、これは通常の単純なパイプライン実装よりもはるかに優れています。つまり、フィルター f_0（1 つまたは複数のスレッドで処理される）が項目をフィルター f_1 のキューに追加します（ここでも f_1 は 1 つまたは複数のスレッドで処理されます）。この単純な実装では、あるコアのフィルター f_0 によって処理される項目が、別のコアのフィルター f_1 によって処理される可能性が高いため、局所性が損なわれます。TBB では、f_0 と f_1 が前述の条件を満たしている場合、これは発生しません。その結果、TBB パイプラインは、パイプラインの先頭にさらに項目を投入する前に、すでに処理中の項目を終了する傾向があります。この動作は、データの局所性を高めるだけでなく、シリアルフィルターに必要なキューのサイズを減らすことでメモリー使用量を削減します。

まとめ

この章では、TBB を使用して表現された並列処理のパフォーマンスを調整する 2 つの方法について説明しました:

（1） ライブラリーが使用するスレッド数とハードウェア・リソースを制限する。

（2） TBB アルゴリズムの粒度とアフィニティーを最適化する。

リソースを制限するため、global_control、明示的な task_arena オブジェクト、および task_scheduler_observer オブジェクトを導入しました。global_control インスタンスは、同時に実行できるワーカースレッド数の上限などのグローバル パラメーターを設定するために使用されます。

明示的な task_arena オブジェクトを使用すると、特定のワークに使用できるスレッド数を制御し、アリーナに参加するスレッドをハードウェア・プラットフォームに割り当てる方法のヒントを提供できます。最後に、task_scheduler_observer オブジェクトを使用すると、スレッドが並列実行に参加するたびに独自のコードを実行できます。

次に、TBB アルゴリズムのパフォーマンス・フックに注目しました。これらのフックを使用すると、粒度、アフィニティー、同時実行性を制御できます。

12 章 TBB から oneTBB への移行

現在の TBB（oneTBB）は最新の C++、特に C++11 以降に依存しているため、C++11 以前では必要であったインターフェイスが廃止されています。さらに、10 年以上にわたるユーザー体験に基づいて、スケジューラーへのインターフェイスが整理され、使いやすくなりました。

oneTBB は TBB とほぼソース互換性がありますが、一部のインターフェイスは廃止されました。この章では、これらの変更点と、*古い TBB* から新しい *oneTBB* への移行が必要な場合の対処方法について説明します。また、コードを移行するガイドを提供するとともに、TBB を長年使用してきたユーザーのために、変更の理由を説明したいと考えています。バグやセキュリティーの修正だけでなく、新機能や改善も oneTBB に追加されているため、移行は重要です。したがって、TBB を最大限に活用し続けるには、oneTBB に移行する必要があります。

長年使われてきた tbb:: – oneapi:: へコードを移行する必要はありません

「私にとっては依然として TBB です」という私たちの主張と同じように、`oneapi` 名前空間を使用するかどうかはオプションです。TBB コードに `oneapi` 名前空間やヘッダーパスを指定する変更を加えないことをお勧めします。`tbb` 名前空間とヘッダーパス内の `tbb` を使用することは、これまでも機能しており、今後も機能し続けます。`tbb::` を `oneapi::tbb::` にしたり、`<tbb/...>` を `<oneapi/tbb/...>` に変更する必要はありません。

これがどのように動作するのか知りたい開発者向けに、`include/tbb` 内のファイルには `include/oneapi/tbb` と一致するヘッダーが含まれています。例えば、著作権情報以外に、ファイル `include/tbb/parallel_for.h` には次の 1 行が含まれています。

```
#include  "../oneapi/tbb/parallel_for.h"
```


M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5_12

すべてのパブリッククラスと関数は tbb 名前空間で定義されていますが、その後 oneapi::tbb 名前空間に*挿入*されます。include/oneapi/tbb の各ヘッダーには、次のようなコードが含まれています。

```
namespace oneapi {
    namespace tbb = ::tbb;
}
```

これがどのように機能するか完全に理解することは重要ではありませんが、tbb 名前空間内のすべてが oneapi::tbb 名前空間に存在し（その逆も同様）、include/tbb 内のすべてのヘッダーが include/oneapi/tbb ディレクトリー内のヘッダーにリダイレクトされることを覚えておいてください。

oneapi 名を追加しなくても問題なく動作するコードを変更することはお勧めしません。oneapi::tbb:: は tbb:: に簡単に接続する便利な機能です。<oneapi/tbb/...> は <tbb/...> と同等です。

何が変わったか

TBB から oneTBB への変更の動機は、次の 2 つのカテゴリーに分けられます:
(1) 最新の C++ との整合性。
(2) インターフェイスの改善。
最後に、古い TBB と新しい TBB（oneTBB）を併用した場合のランタイム互換性に関する注意事項をいくつか示します（これらはともに動作しますが、新しいライブラリーに移行するのは利点があります）。

変更: 現代の C++ のアライメント

C++ 標準委員会は、2011 年以来 3 年ごとに新しい標準仕様をリリースしています。最初のアップデートである C++11 は、現代の C++ の始まりとして知られています。oneAPI とその C++ ライブラリーは、時間経過とともに必要な C++ 標準のバージョンが増えることが予想されますが、現在は C++11 のみが必要です。C++11 では、右辺値参照、ラムダ式、auto、アトミック変数型、unique_ptr、shared_ptr など、多くの重要な機能が導入されました。

アプリケーションをさらに新しい C++ 標準でコンパイルすると、推論補助（C++17 で導入）などの oneTBB の追加機能が利用できるようになります。推論補助を使用すると、明示的なテンプレート引数を削除できることがあります。C++11 コンパイラー（またはフラグ）を使用してコンパイルすると、新しい C++ 標準の機能に依存するサポートは利用できなくなります。

tbb::atomic の削除

TBB から oneTBB へ移行する際の重要な変更点の 1 つは、TBB アトミック・インターフェイスが廃止され、代わりに最新の C++ で追加された新しい標準アトミック・インターフェイスが採用されたことです。TBB の開発者は、C++ 標準のアトミック・インターフェイス実装のパフォーマンスが TBB アトミックに匹敵することを確認したため、それらの手動による実装を保守する必要はありませんでした。同様に、TBB ライブラリーの実装も標準インターフェイスを内部で使用するように変更され、TBB ライブラリー自体の移植性が向上しました。

TBB プログラムを TBB アトミックから最新の C++ アトミックに変更するのは、単純な置き換え作業です。これには、名前空間 `tbb` を `std` に変更し、`<atomic>` ヘッダーを使用することも含まれます。多くの名前は全く同じですが、一部は異なります。例えば、`compare_and_swap` の呼び出しは、`compare_exchange_weak` または `compare_exchange_strong` に置き換える必要があります。`tbb::atomic` はコピー・コンストラクター（新しい独立したコピーを作成する）を提供しますが、`std::atomic` は提供しません。

アトミックを含む同期は並列プログラミングにおいて非常に重要なトピックであるため、本書では 1 つの章を割り当てています。8 章では、同期がアルゴリズムの効率良いスケーリングをサポートするのを確認する高レベルのトピック（つまり、「同期が必要以上に邪魔にならないようにする」）を正当に評価するため、TBB と最新の C++ の同期サポートについて説明します。

8 章の実行例では、誤った実装から始まり、粗粒度ロック、細粒度ロック、アトミック、ロックを全く使用しない代替実装など、さまざまな同期の代替手段を反復するさまざまな並列実装を確認します。途中、注目すべき点に触れ、ミューテックスを特徴付けるプロパティー、標準 C++ ライブラリーと TBB ライブラリーの拡張機能で利用可能なミューテックス・フレーバーの種類、アルゴリズムの実装がミューテックスに依存する場合に発生する一般的な問題などを紹介します。

図 12-1 は、8 章の例を古い TBB アトミックを使用して表現した場合と、標準アトミックを使用して表現したものです。このように、この具体的なケースでの違いはごく僅かです。

```
#include  <tbb/tbb.h>
  // 並列実行
  std::vector<tbb::atomic<int>> hist_p(num_bins);
  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
           [&](const  tbb::blocked_range<size_t>& r)
           {
              for (size_t  i = r.begin(); i < r.end(); ++i)
                 hist_p[image[i]]++;
           });
```

(a) 古い TBB のアトミックを使用

```
#include  <atomic>
  // 並列実行
  std::vector<std::atomic<int>> hist_p(num_bins);
  parallel_for(tbb::blocked_range<size_t>{0, image.size()},
           [&](const  tbb::blocked_range<size_t>& r)
           {
              for (size_t  i = r.begin(); i < r.end(); ++i) hist_p[image[i]]++;
           });
```

(b) 標準アトミックを使用

図 12-1. 8 章の例では、tbb::atomic から std::atomic への移行の前後を示しています。サンプルコード: migration/migrate_atomics

tbb::tbb_exception、tbb::captured_exception、tbb::movable_exception

前述のように、oneTBB には、`std::exception_ptr` を使用して例外をキャプチャーする機能を含む C++11 のサポートが必要です。C++11 を利用できるようになる前は、TBB は例外をキャプチャー、要約、再スローする仕組みを独自に提供していました。例外については、9 章で詳しく説明しています。

その他の変更

C++ 標準の進化に関連するその他の変更は、アルゴリズム本体を提供するラムダ式のサポートなど、TBB の使いやすさを高めるものや、適切な場所での右辺値参照や `std::move` の使用など、TBB コードの効率化を図るものなどがあります。ただし、これらの変更は新しく記述されたコードにのみ影響し、既存のコードを修正する必要はありません。

変更: 冗長または問題のあるインターフェイスの削除

TBB の 10 年の経験から、一部のインターフェイスは単に不要であるか、エラーが発生しやすいことが分かりました。

task_scheduler_init 経由でアクセスされるコントロール

TBB がリリースされた当初、明示的なタスクアリーナはサポートされていませんでした（明示的なアリーナについては 11 章で説明されています）。本来、すべてのスレッドで共有されるアリーナは 1 つしかありませんでした。この間、グローバル・コントロールと特定の領域に影響するコントロールを区別する必要はありませんでした。`task_scheduler_init` クラスは、スレッドプール内のスレッドの最大数、単一アリーナ内のスロット数、およびスレッド・スタック・サイズを設定する単一のインターフェイスでした。ライフタイムは、TBB スケジューラー全体の初期化と終了も制御していました。

しかし、ライブラリーが進化し、複数の暗黙的および明示的なタスクアリーナのサポートが導入されるにつれて、ライブラリー全体のプロパティーに影響するコントロールと、特定のアリーナに影響するコントロールに分けることが有用になりました。
`task_scheduler_init` クラスは最新バージョンから除外されたのは、単純にこのクラスが、現在ではクリーンに分離されたコントロールによって対処されており、あまりにも多くの個別の懸念事項を単純に統合したためです。

`task_scheduler_init` クラスは、oneTBB の最初のリリース以前に非推奨となり、最新のライブラリーでは削除されました。デフォルトのスレッド数を照会する方法として、`tbb::task_scheduler_init::default_num_threads()` に代わって `tbb::info::default_concurrency()` 関数が導入されました。11 章で説明されている `tbb::global_control` クラスは、最大同時実行数、スレッド・スタック・サイズ、エラー処理などのグローバル・パラメーターを設定する oneTBB のアプローチです。同様に 11 章で説明した `task_arena` クラスは、スレッドがワークに参加できるスロット数を制御できる新しい方法です。

`task_arena` クラスは、`task_scheduler_init` でサポートされていたレンジを大幅に超えて進化を続け、現在は 11 章で説明するように、優先度レベルとハードウェア対応の制約をサポートしています。`task_scheduler_init` をこれらの新しいインターフェイスに置き換える簡単な例を図 12-2 に示します。

```
const int N = tbb::task_scheduler_init::default_num_threads();

void setThreadsAndSlots() {
   tbb::task_scheduler_init init(N);

   tbb::parallel_for(0,
                     10*N,
                     [](int) { doWork(0.01); });
}
```

(a) 古いインターフェイス

```
const int N = tbb::info::default_concurrency();

void setThreadsAndSlots() {
   tbb::global_control gc(tbb::global_control::max_allowed_parallelism, N);

   tbb::task_arena a{N};

   a.execute([]() {
      tbb::parallel_for(0,
                        10*N,
                        [](int) { doWork(0.01); });
   });
}
```

(b) 新しいインターフェイス

図 12-2. 古い tbb::task_scheduler_init インターフェイスから新しい tbb::info、tbb::global_control、および tbb::task_arena インターフェイスに移行します。サンプルコード: migration/migrate_task_scheduler_init.cpp

図 12-2 は冗長かもしれませんが、グローバル・スレッド・プールで使用可能なスレッド数が、アリーナ内で使用可能なスロット数と対照的であることを混同する余地はほとんどありません。どのアリーナが使用されているかは、正確に把握できます。これはアリーナを制限する場合に重要です。11 章では、`tbb::global_control` と `tbb::task_arena` を組み合わせて、許容される最大同時実行性を正確に制御する方法を示します。

最後に、TBB スケジューラーのライフタイムは、`task_scheduler_handle` クラスと `free` 関数でによって制御されるようになりました:

```
void finalize(task_scheduler_handle &handle)
```

図 12-3 の表は、これらの融合された懸念事項が oneTBB でどのように明確に分離されているかをまとめたものです。

task_scheduler_init によって制御される懸念事項	代替
デフォルトのスレッド数を照会	`int tbb::info::default_concurrency();`
max_concurrency やスレッド・スタック・サイズなどのグローバル・パラメーター。	`tbb::global_control`
特定のワークのためにワーカースレッドとマスタースレッドが使用できるスロット数。	`tbb::task_arena`
TBB スケジューラー、グローバル・スレッド・プール、データ構造のライフタイム。	`tbb::task_scheduler_handle` `void tbb::finalize(task_scheduler_handle &handle);`

図 12-3. task_scheduler_init によって処理される懸念事項が oneTBB でより明確に分離方法のまとめ

parallel_do の削除

ライブラリーから削除されたもう 1 つのインターフェイスは parallel_do です。これは混乱を招くからではなく、冗長であるために行われました。TBB の進化の過程で、parallel_do と parallel_for_each という 2 つの類似したアルゴリズムがライブラリーに組み込まれていた時期がありました。どちらも、開始から終了イテレーターまで走査するか、コンテナー内の要素を反復処理することによってタスクを作成しました。2 つのインターフェイスの違いは、parallel_do には、並列処理する項目を追加供給するため、ボディーが使用するオプションのフィーダー引数が含まれている点です。TBB が近代化されるにつれて、これら 2 つのアルゴリズムは 1 つに統合され、parallel_for_each になりました。2 章で説明されているこのアルゴリズムの更新バージョンでは、ボディーへのオプションのフィーダー引数がサポートされています。これにより、非常に類似した 2 つのアルゴリズムを維持する理由がなくなりました。

parallel_do から parallel_for_each への移行パスは、図 12-4 に示すように単純な置き換えです。

```
tbb::parallel_do( &top_left, &top_left+1,
    [&](const BlockIndex&    tbb::parallel_do_feeder<BlockIndex>) {
   auto [r, c] = bi;
    computeBlock(N, r, c, x, a, b);
    // 準備ができたらサクセサーを右に追加
    if (c + 1 <= r && --ref_count[r*num_blocks + c + 1] == 0) {
      f.add(BlockIndex(r, c + 1));
    }
    // 準備ができたらサクセサーを下に追加
    if (r + 1 < (size_t)num_blocks &&
        --ref_count[(r+1)*num_blocks + c] == 0) {
      f.add(BlockIndex(r+1, c));
    }
  }
);
```

(a) 古いインターフェイス

```
tbb::parallel_for_each( &top_left, &top_left+1,
    [&](const BlockIndex&    tbb::feeder<BlockIndex>&) {
   auto [r, c] = bi;
    computeBlock(N, r, c, x, a, b);
    // 準備ができたらサクセサーを右に追加
    if (c + 1 <= r && --ref_count[r*num_blocks + c + 1] == 0) {
      f.add(BlockIndex(r, c + 1));
    }
    // 準備ができたらサクセサーを下に追加
    if (r + 1 < (size_t)num_blocks &&
        --ref_count[(r+1)*num_blocks + c] == 0) {
      f.add(BlockIndex(r+1, c));
    }
  }
);
```

(b) 新しいインターフェイス

図 12-4. 古い tbb::parallel_do インターフェイスから新しい tbb::parallel_for_each インターフェイスに移行します。サンプルコード: migration/migrate_parallel_do.cpp

pipeline クラスの削除

ライブラリーのもう一つの整理として、tbb::pipeline クラスが削除されました。TBB 開発者は、2 章で説明されているタイプセーフな tbb::parallel_pipeline 関数がはるかにユーザーフレンドリーであり、TBB ユーザーが必要とするユースケースには十分であることを学びました。 tbb::pipeline クラスは、適切な代替手段ではありますが、tbb::parallel_pipeline 関数ではカバーされていない、スレッド・バウンド・フィルターという 1 つのユースケースをカバーしていました。

スレッド・バウンド・フィルターは、TBB ワーカースレッドでは全く処理されませんでした。代わりに、これらのフィルターは、TBB で制御されていないスレッドからフィルターの `process_item` または `try_process_item` 関数を呼び出すことによって処理されます。この機能は、TBB の以前のバージョンで導入されたとき、特定のスレッド（特定のリソースへのアクセスが許可された唯一のスレッド）から関数を呼び出す必要がある通信、またはオフロード・ライブラリーの実装で利用するように設計されていました。時が経つにつれて、こうした制限は次第に少なくなってきました。そして、現在、この機能は不要になりました。

古いクラス `tbb::pipeline` から新しいタイプセーフな `tbb::parallel_pipeline` 関数に移行するパスは、単純な検索と置換では済みませんが、それでも、ほとんどのユースケースのコンセプトは簡単にマッピングできます。`parallel`、`serial_out_of_order`、`serial_in_order` フィルターがあります。インフライトの項目の合計数を制限する `max_number_of_live_tokens` 引数もまだあります。また、パイプラインを通過する項目にフィルターを適用する線形順序を表現するためのインターフェイスも存在します。しかし、インターフェイスの構文は変更されました。2 章と 11 章では、現在の TBB のインターフェイスについて詳しく説明します。

タスク指向の優先順位を削除してアリーナの優先順位を優先する

11 章では、優先度を考慮した `task_arena` を作成するインターフェイスについて説明します。oneTBB 以前のバージョンでは、キューに投入されたタスクごとにタスク単位で優先順位を設定するためのインターフェイス、またはタスクのグループに対して `task_group_context` を通じて優先順位を設定するためのインターフェイスがありました。TBB が進化するにつれて、明示的な `task_arena` オブジェクトは、異なる方法で制約されるべきワークを分離するための抽象化となりました。11 章では、`task_arena` にハードウェア対応の制約を追加する方法と、`task_arena` ごとに優先順位を設定する方法について説明しました。`task_arena` には、`execute` メソッドと `enqueue` メソッドの両方があり、関連するタスクをグループ化するのに最適です。

古いコードがキューに登録されたタスクを使用しているか、`task_group_context` を使用しているかにかかわりなく、優先順位の移行パスは同じです。どちらの場合も、必要な優先順位で明示的な `task_arena` を作成し、その特定のアリーナでワークを実行できます。`task_arena` は、単一のタスクがキューに入れられた用途を置き換える `enqueue` 関数を提供します。また、アルゴリズムを `task_arena::execute` の呼び出し内にネストして、`task_group_context` が使用されていたケースを移行できます。

最初に、いわゆる「*ファイア・アンド・フォーゲット（撃ち放し能力）*」タスクについて見てみましょう。古いバージョンの TBB では、これらは最低レベルのタスク・インターフェイスを使用してキューに投入されたタスクでした。図 12-5(a) では、`tbb::task` を継承し、仮想関数 `execute` をオーバーライドする新しいクラスが定義されています。次に、メインループで、このタイプのタスクが、最下位レベルのタスク API を使用してキューに投入されます。関数の最後の引数は、`tbb::priority_low`、`tbb::priority_normal`、`tbb::priority_high` などの優先度です。

```
auto  P = tbb::task_scheduler_init::default_num_threads();

class  MyTask  : public  tbb::task  {
public:
  MyTask(const  char  *m, int  i) : msg(m), messageId(i) { }
  tbb::task  *execute() override  {
    taskFunction(msg, messageId);
    return  NULL;
  }
private:
  std::string msg;
  int  messageId;
};

void  enqueueSeveralTasks(int  num_iterations) {
  for  (int  i = 0; i < num_iterations; ++i) {
    tbb::task::enqueue(*new( tbb::task::allocate_root() )
                       MyTask( "L", i ), tbb::priority_low);
    tbb::task::enqueue(*new( tbb::task::allocate_root() )
                       MyTask( "N", i ), tbb::priority_normal);
    tbb::task::enqueue(*new( tbb::task::allocate_root() )
                       MyTask( "H", i ), tbb::priority_high);
  }
  doWork(1.0);
}
```

(a) 古いインターフェイス

```
auto  P = tbb::info::default_concurrency();

void  enqueueSeveralTasks(int  num_iterations) {
  tbb::task_arena low_arena{P, 0, tbb::task_arena::priority::low};
  tbb::task_arena normal_arena{P, 0, tbb::task_arena::priority::normal};
  tbb::task_arena high_arena{P, 0, tbb::task_arena::priority::high};

  for  (int  i = 0; i < num_iterations; ++i) {
    low_arena.enqueue([i]() { taskFunction("L", i); });
    normal_arena.enqueue([i]() { taskFunction("N", i); });
    high_arena.enqueue([i]() { taskFunction("H", i); });
  }
  doWork(1.0);
}
```

(b) 新しいインターフェイス

図 12-5. 単一のタスクをキューに登録する古い enqueue インターフェイスから新しい tbb::task_arena インターフェイスに移行します。サンプルコード: migration/migrate_priorities.cpp

図 12-5(b) は、制約により優先順位を設定した task_arena オブジェクトを使用してこのパターンを oneTBB に移行できることを示しています。各タスクは異なるタスクアリーナにキューイングされます。

TBB の以前のバージョンでは、task_group_context の一部として優先順位もサポートされていました。task_group_context オブジェクトに優先順位を設定すると、そのオブジェクトはそのアルゴリズムに関連するすべてのワークの優先順位を設定できます。

一貫して、oneTBB は、優先順位などの制約を設定する場合、代わりに task_arena オブジェクトを使用します。図 12-7 は、図 12-6 の例を oneTBB でどのように表現するか示しています。

```
void  runParallelForWithHighPriority() {
  std::thread t0([]() {
    waitUntil(2);
    tbb::task_group_context tgc;
    tgc.set_priority(tbb::priority_normal);
    tbb::parallel_for(0, 10,
       [](int  i) {
         std::printf("N");
         std::this_thread::sleep_for(std::chrono::milliseconds(10));
       }, tgc);
  });
  std::thread t1([]() {
  waitUntil(2);
  tbb::task_group_context tgc;
  tgc.set_priority(tbb::priority_high);
  tbb::parallel_for(0, 10,
    [](int  i) {
      std::printf("H");
      std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }, tgc);
  });
  t0.join();
  t1.join();
  std::printf("\n");
}
```

図 12-6. task_group_context を使用して優先順位を設定する古いインターフェイスの例。サンプルコード: migration/migrate_priorities.cpp

```
void runParallelForWithHighPriority() {
  std::thread t0([]() {
    waitUntil(2);
    tbb::task_arena normal_arena{P, 0, tbb::task_arena::priority::normal};
    normal_arena.execute([]() {
      tbb::parallel_for(0, 10, [](int i) {
        std::printf("N");
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
      });
    });
  });
  std::thread t1([]() {
    waitUntil(2);
    tbb::task_arena high_arena{P, 0, tbb::task_arena::priority::high};
    high_arena.execute([]() {
      tbb::parallel_for(0, 10, [](int i) { std::printf("H"); });
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    });
  });
  t0.join();
  t1.join();
  std::printf("\n");
}
```

図 12-7. task_arena を使用して oneTBB で優先順位を設定する。サンプルコード: migration/migrate_priorities.cpp

最下位レベルのタスク/スケジューラー API の削除

oneTBB における最も大きな変更点は、パブリック・インターフェイスとしての最低レベルのタスク API の削除です。

TBB は、アプリケーションにおける*スレッドの構成可能性*とスケーラビリティーを最大限に高めることを目的としていることを念頭に置くことが重要です。複数のコード（一部は自身で書いたもの、一部は自身で書いていないもの）からアプリケーションを組み立てた後でも、アプリケーションが正しく効率的に動作することを期待します。オリジナルのインターフェイスには、不要で、混乱を招き、過度に制約する低レベルのノブが残されていました。この組み合わせは最適でないコードにつながり、それが修正されない限りスケーリングを低下させます。

フローグラフ、task_group、タスクアリーナなどの TBB の新しい機能は、アプリケーションに適切なインターフェイスを提供するように進化しました。アプリケーションを新しいインターフェイスに移行し、元のインターフェイスを廃止することは有益であることが証明されています。この章は、コードを移行する詳細な点を理解するのに役立ちます。

興味があれば、新しい低レベルのスケジューラー・インターフェイスがあり、これは純粋に内部的なアルゴリズムの実装に使われます。これらのインターフェイスは、TBB の実装者とアプリケーション作成者の両方に役立つものではありません。そのため、専門家以外にはあまり適していませんが、専門的な TBB 実装者向けにより高度な制御と柔軟性が提供されています。これらのインターフェイスは、TBB の内部実装の問題のみを解決することを目的としています。それを維持することが重要であることが証明されました。同様に、アプリケーション・レベルのインターフェイスは、その目的を適切に果たすことに集中できるようになりました。

現在の TBB は、スケーラブルで構成可能なコードの作成を最大限に高めるのに役立ちますが、今後も改良と追加が続けられます。さらに、TBB 開発者は、6 章で説明されているように、`task_group` の改良を続け、さらに多くのユースケースの表現を簡素化し、古い低レベルのタスク・インターフェイスのレベルを継続的に向上させています。

低レベルタスク API からの移行

スレッディング・ビルディング・ブロック（TBB）の低レベルのタスク API は複雑でエラーが発生しやすいと考えられていたため、これが oneAPI スレッディング・ビルディング・ブロック（oneTBB）から削除された主な理由でした。このガイドは、低レベルのタスク API が使用されるユースケースにおいて、TBB から oneTBB（現在の TBB）へ移行するのに役立ちます。

タスク・ブロッキング

ほとんどのユースケースでは、個別の独立したタスクの生成は、`tbb::task_group` または `tbb::parallel_invoke` のどちらかに置き換えることができます。

例えば、図 12-8(a) では、古い TBB インターフェイスを使用してタスクの作成とスケジュールを行っています。`RootTask`、`ChildTask1`、および `ChildTask2` は、tbb::task から継承し、そのインターフェイスを実装するユーザー提供の関数です。`ChildTask1` タスクと `ChildTask2` タスクを生成すると、それらのタスクが（おそらく）相互に並行して実行されるようになります。`RootTask` は、その両方が完了した後に実行されます。

```
class MyTask : public tbb::task {
  double my_time;
public:
  MyTask(double t=0.0) : my_time(t) {}
  virtual tbb::task* execute() {
    doWork(my_time);
    return nullptr;
  }
};

void taskBlocking() {
  MyTask& root = *new(tbb::task::allocate_root()) MyTask{};

  MyTask& child1 = *new(root.allocate_child()) MyTask{0.1};
  MyTask& child2 = *new(root.allocate_child()) MyTask{0.2};

  root.set_ref_count(3);

  tbb::task::spawn(child1);
  tbb::task::spawn(child2);
  root.wait_for_all();
}
```

(a) 古いインターフェイス

```
void taskBlocking() {
  tbb::task_group g;
  g.run([]() { doWork(0.1); });
  g.run([]() { doWork(0.2); });
  g.wait();
}
```

(b) 新しいインターフェイス

図 12-8. 単純なブロッキング・タスクのために、古いタスク・インターフェイスから新しい tbb::task_group インターフェイスに移行します。サンプルコード: migration/migrate_task_blocking.cpp

この最も単純な例でも、古いタスク API がいかに低レベルであったかが分かります。RootTask、ChildTask1、ChildTask2 など、tbb::task から継承したクラスは、仮想関数 execute をオーバーライドします。次に、ユーザーは、tbb::task::allocate_root() や root.allocate_child() などの特殊関数からスペースを取得し、インプレース new を使用してオブジェクトを割り当てる必要があります。allocate_child() などの関数を使用すると、特別な方法でメモリーが初期化され、親子関係が生成されます。

それでも、root.set_ref_count(3) の呼び出しで示されているように、参照カウントは明示的に行う必要があります。“なぜ 3 なのか？” と疑問に思うかもしれません。それは、開発者は、wait_for_all 呼び出し（暗黙的に参照カウントを減分）の前にルートタスクが途中で終了しないように参照カウントを追加する必要があるためです。エラーが発生しやすいようには思われません。

対照的に、図 12-8(b) では、コードが tbb::task_group に書き直されています。コードがより簡潔になりました。また、ラムダ関数も有効になり、tbb::task* tbb::task::execute() 仮想メソッドをオーバーライドする tbb::task インターフェイスを実装する必要がなくなります。この新しいアプローチでは、C++ 標準に従った方法で関数を操作します。

この例は非常に単純ですが、図 12-9 に示すように、古い TBB と新しい TBB（parallel_invoke）の両方でサポートされている、さらに単純なソリューションもあります。

```
void  parallelInvoke() {
  tbb::parallel_invoke([]() { doWork(0.1); },
                []() { doWork(0.2); } );
}
```

図 12-9. parallel_invoke を使用して 2 つのタスクを起動して待機します。サンプルコード: migration/migrate_task_blocking.cpp

タスクからさらにワークを追加

TBB の古いバージョンでは、ワークの量が事前に分からず、並列アルゴリズムの実行中にワークを追加する必要がある場合に、tbb::parallel_do アルゴリズムが使用されます。この章の前半では、tbb::parallel_do アルゴリズムを tbb::parallel_for_each アルゴリズムに移行する方法について説明しました。ただし、古いアプリケーションの中には、図 12-10、12-11、および 12-12 に示すように、最下位レベルのタスク API を直接使用してワークを動的に追加するものもあります。

```
const  int  block_size  = 512;
using  BlockIndex  = std::pair<size_t, size_t>;

void  parallelFwdSub(std::vector<double>&  x,
                const  std::vector<double>&  a,
                std::vector<double>&  b) {
  const  int  N  = x.size();
  const  int  num_blocks  = N  / block_size;

  // 参照カウントを作成
  std::vector<std::atomic<char>> ref_count(num_blocks*num_blocks);
  ref_count[0]  =  0;
  for  (int  r  = 1; r  < num_blocks; ++r) {
    ref_count[r*num_blocks]  =  1;
    for  (int  c  = 1; c  < r; ++c) {
      ref_count[r*num_blocks  + c]  =  2;
    }
    ref_count[r*num_blocks  + r]  =  1;
  }

  BlockIndex  top_left(0,0);

  RootTask& root = *new(tbb::task::allocate_root()) RootTask{};
  root.set_ref_count(2);
  FwdSubTask& top_left_task  =
    *new(root.allocate_child()) FwdSubTask(root, N, num_blocks, top_left, x, a,
b, ref_count);
  tbb::task::spawn(top_left_task);
  root.wait_for_all();
}
```

図 12-10. 古いタスク・インターフェイスを使用して前方置換の例を表現します。サンプルコード: migration/migrate_tasks_adding_work.cpp

図 12-10 は、1 つの子タスクを持つルートタスクの作成を示しています。低レベルのタスク API から移行するため、その詳細は省略しますが、高レベルでは、ルートタスクは、図 12-11 で再帰的にスポーンされるタスクの完了を待機する上位レベルのハンドルとして使用されます。

図 12-11 は、`tbb::task` から継承する `FwdSubTask` クラスを示しています。このクラスには、図 12-12 に示されている `execute` 関数から使用されるプライベート・メンバー変数があります。古いタスク API では、開発者は `tbb::task` から継承し、`execute` メソッドをオーバーライドするクラスを実装する必要がありました。

```
using  RootTask  = tbb::empty_task;

class  FwdSubTask  : public  tbb::task  {
public:
  FwdSubTask(RootTask&  root,  int  N, int  num_blocks,
             const  std::pair<size_t, size_t>&  bi, std::vector<double>&  x,
             const  std::vector<double>&  a, std::vector<double>&  b,
             std::vector<std::atomic<char>>&  ref_count) :
             my_root(root), my_N(N), my_num_blocks(num_blocks), my_index(bi),
             my_x(x), my_a(a), my_b(b), my_ref_count(ref_count) {}

  tbb::task*  execute() override  {  /*   図 12-12 を参照 */ }

private:
  RootTask& my_root;
  BlockIndex  my_index;
  const  int  my_N, my_num_blocks;
  std::vector<double>& my_x;
  const  std::vector<double>& my_a;
  std::vector<double>& my_b;
  std::vector<std::atomic<char>>& my_ref_count;
};
```

図 12-11. 前方置換の例を表現するため古いタスク・インターフェイスを実装するクラス。サンプルコード: migration/migrate_tasks_adding_work.cpp

図 12-12 の execute 関数の定義では、ルートタスクを親として、左または右の子、あるいはその両方が追加されることがあります。`allocate_additional_child_of(root)` でインプレース new を使用すると、ルートが途中で終了しないように必要な参照カウントが処理されます。新しい子タスクによって、さらにルートに子タスクが追加されることもあります。最終的に、リーフは新しい子を追加せずに `FwdSubTask::execute` を呼び出します。

```
tbb::task*  execute() override  {
  auto  [r, c] = my_index;
  computeBlock(my_N, r, c, my_x, my_a, my_b);
  // 準備ができたらサクセサーを右に追加
  if  (c  + 1  <= r  && --my_ref_count[r*my_num_blocks  + c  + 1]  == 0) {
    FwdSubTask& child  =
      *new(allocate_additional_child_of(my_root))
         FwdSubTask(my_root, my_N, my_num_blocks, {r, c+1},
                  my_x, my_a, my_b, my_ref_count);
    tbb::task::spawn(child);
  }
  // 準備ができたらサクセサーを下に追加
  if  (r  + 1  < (size_t)my_num_blocks
     && --my_ref_count[(r+1)*my_num_blocks  + c]  == 0) {
    FwdSubTask& child  =
      *new(allocate_additional_child_of(my_root))
          FwdSubTask(my_root, my_N, my_num_blocks, {r+1, c},
                   my_x, my_a, my_b, my_ref_count);
    tbb::task::spawn(child);
  }
  return  nullptr;
}
```

図 12-12. 古いタスク・インターフェイスで前方置換の例を表現できるようにする execute の実装。サンプルコード: migration/migrate_tasks_adding_work.cpp

この例を移行する方法の 1 つは、図 12-4 に示すように、tbb::parallel_do からの移行と同じように、tbb::parallel_for_each を使用することです。よりタスク指向のアプローチを使用したい場合は、図 12-13 と 12-14 に示すように task_group を使用できます。

```
const  int  block_size  = 512;
using  BlockIndex  = std::pair<size_t, size_t>;

void  parallelFwdSubTaskGroup(std::vector<double>&  x,
                        const  std::vector<double>&  a,
                        std::vector<double>&  b) {
  const  int  N = x.size();
  const  int  num_blocks = N / block_size;

  // 参照カウントを作成
  std::vector<std::atomic<char>> ref_count(num_blocks*num_blocks);
  ref_count[0] = 0;
  for  (int  r = 1; r < num_blocks; ++r) {
    ref_count[r*num_blocks] = 1;
    for  (int  c = 1; c < r; ++c) {
      ref_count[r*num_blocks + c] = 2;
    }
    ref_count[r*num_blocks + r] = 1;
  }

  BlockIndex top_left(0,0);
  tbb::task_group tg;
  tg.run([&]() {
    fwdSubTGBody(tg, N, num_blocks, top_left, x, a, b, ref_count);
  });
  tg.wait();
}
```

図 12-13. 図 12-10 の parallelFwdSub 関数は、tbb::task の代わりに task_group を使用するように書き換えられました。fwdSubTGBody 関数の実装を図 12-14 に示します。サンプルコード: migration/migrate_tasks_adding_work.cpp

図 12-14 では、task_group オブジェクト tg は、図 12-10 のルートタスクと同様に動作します。再帰的に追加されたすべての子タスクは、図 12-10 ですべての子タスクがルートタスクの子として追加されるように、同じ task_group tg 内で実行されます。

```
void  fwdSubTGBody(tbb::task_group&  tg,
                int  N, int  num_blocks, const  std::pair<size_t, size_t>
bi,
                std::vector<double>&  x, const  std::vector<double>&  a,
                std::vector<double>&  b,
                std::vector<std::atomic<char>>&  ref_count) {
  auto  [r, c] = bi;
  computeBlock(N, r, c, x, a, b);
  // 準備ができたらサクセサーを右に追加
  if  (c + 1  <= r && --ref_count[r*num_blocks + c + 1] == 0) {
    tg.run([&, N, num_blocks, r, c]() {
      fwdSubTGBody(tg, N, num_blocks, BlockIndex(r, c+1), x, a, b,
ref_count);
    });
  }
  // 準備ができたらサクセサーを下に追加
  if  (r + 1  < (size_t)num_blocks && --ref_count[(r+1)*num_blocks + c]
== 0) {
    tg.run([&, N, num_blocks, r, c]() {
      fwdSubTGBody(tg, N, num_blocks, BlockIndex(r+1, c), x, a, b,
ref_count);
    });
  }
}
```

図 12-14. 図 12-12 の execute 関数のようなロジックを実装する fwdSubTGBody 関数。サンプルコード: migration/migrate_tasks_adding_work.cpp

タスクの再利用

TBB の以前のバージョンでは、タスク・オブジェクトを**再利用**して再実行することができました。これにより、タスク割り当てのオーバーヘッドが削減されました。新しいタスクを割り当てる代わりに、現在のタスク・オブジェクトのメンバー変数が更新され、タスクは新しいタスクであるかのようにスケジュールされます。古い API では再利用のバリエーションがいくつかありましたが、これもまた、この古い API が低レベルでエラーが発生しやすい性質を示しているかもしれません。古い関数には、`recycle_as_continuation`、`recycle_as_safe_continuation`、`recycle_as_child_of`、`recycle_to_reexecute` がありました。これらは、どのタスクがタスクの親になるか、および `recycle_as_continuation` と `recycle_as_safe_continuation` では、早期実行を防ぐためどのように参照がカウントされるかという点で異なります。繰り返しますが、削除された API についてはここでは詳しく説明しません。

図 12-15 は、再利用されるときに `tbb::task` を使用する前方置換の例を示しています。右側のサクセサー (`c+1`) が準備できている場合、`recycle_as_c1` は `true` に設定されます。両方のサクセサーの準備ができている場合は、下のサクセサー (`r+1`) が新しいタスクとして生成され、現在のタスクが右側のサクセサー `my_index = {r+1, c}` になるよ

うに更新されます。

サクセサーのうち 1 つだけが準備できている場合、新しいタスクは生成されず、現在のタスクは、my_index = {r+1, c} または my_index = {r, c+1} のいずれかの準備できているタスクとして再利用されます。現在のタスクが更新されると、recycle_to_reexecute() が呼び出され、TBB スケジューラーにこのタスクが戻った後に再度スケジュールするように指示します。

```
tbb::task* execute() override {
  auto [r, c] = my_index;
  computeBlock(my_N, r, c, my_x, my_a, my_b);
  // 準備ができたらサクセサーを右に追加
  bool recycle_as_c1 = false;
  if (c + 1 <= r && --my_ref_count[r*my_num_blocks + c + 1] == 0) {
    recycle_as_c1 = true;
  }
  // 準備ができたらサクセサーを下に追加
  if (r + 1 < (size_t)my_num_blocks
      && --my_ref_count[(r+1)*my_num_blocks + c] == 0) {
    if (recycle_as_c1) {
      FwdSubTask& child = *new(allocate_additional_child_of(my_root))
                        FwdSubTask(my_root, my_N, my_num_blocks, {r+1, c},
                                   my_x, my_a, my_b, my_ref_count);
      tbb::task::spawn(child);
    } else {
      my_index = {r+1, c};
      recycle_to_reexecute();
    }
  }
  if (recycle_as_c1) {
    my_index = {r, c+1};
    recycle_to_reexecute();
  }
  return nullptr;
}
```

図 12-15. 前方置換の例のタスクを再利用する execute 関数（修正済み）。サンプルコード: migration/migrate_recycling.cpp

oneTBB の現在のバージョンでは、operator() 関数の実行中に関数オブジェクトのメンバーを変更し、*this を tbb::task_group::run() に渡すことで同様の結果を得ることができます。図 12-16 は、図 12-13 に示した関数を少し変更したバージョンです。このバージョンでは、ラムダ式の代わりに関数オブジェクトが task_group::run に渡されます。

```
void parallelFwdSub(std::vector<double>& x,
                    const std::vector<double>& a,
                    std::vector<double>& b) {
  const int N = x.size();
  const int num_blocks = N / block_size;

  // 参照カウントを作成
  std::vector<std::atomic<char>> ref_count(num_blocks*num_blocks);
  ref_count[0] = 0;
  for (int r = 1; r < num_blocks; ++r) {
    ref_count[r*num_blocks] = 1;
    for (int c = 1; c < r; ++c) {
      ref_count[r*num_blocks + c] = 2;
    }
    ref_count[r*num_blocks + r] = 1;
  }

  BlockIndex top_left(0,0);

  tbb::task_group tg;
  tg.run(FwdSubFunctor{tg, N, num_blocks, top_left, x, a, b, ref_count});
  tg.wait();
}
```

図 12-16. 図 12-13 で使用されるラムダ式の代わりに、関数オブジェクトを使用する task_group ベースの前方置換バージョン。サンプルコード: migration/migrate_recycling.cpp

図 12-17 と 12-18 は、FwdSubFunctor クラスの定義を示しています。図 12-15 の recycle_to_reexecute の呼び出しの代わりに、tg.run(*this) の呼び出しがあります。テストシステムで実行し、古い API または現在の API のいずれかを使用して前方置換の再利用を使用すると、実行時間が約 2% 短縮されました。したがって、小さなタスクを多数作成する場合、関数オブジェクトの再利用により、TBB の以前のバージョンでタスクの再利用によってパフォーマンスが向上したように、パフォーマンスがわずかに向上する可能性があります。

```
class FwdSubFunctor {
public:
  FwdSubFunctor(tbb::task_group& tg,
                int N, int num_blocks,
                const std::pair<size_t, size_t>& bi,
                std::vector<double>& x,
                const std::vector<double>& a,
                std::vector<double>& b,
                std::vector<std::atomic<char>>& ref_count) :
                my_tg(tg), my_index(new BlockIndex{bi}),
                my_N(N), my_num_blocks(num_blocks),
                my_x(x), my_a(a), my_b(b), my_ref_count(ref_count) {}

  void operator()() const { /* 図 12-18 参照 */ }

private:
  tbb::task_group& my_tg;
  const std::shared_ptr<BlockIndex> my_index;
  const int my_N, my_num_blocks;
  std::vector<double>& my_x;
  const std::vector<double>& my_a;
  std::vector<double>& my_b;
  std::vector<std::atomic<char>>& my_ref_count;
};
```

図 12-17. クラス FwdSubFunctor。operator()() の定義を図 12-18 に示します。サンプルコード: migration/migrate_recycling.cpp

```
void  operator()() const  {
  auto  [r, c] = *my_index;
  computeBlock(my_N, r, c, my_x, my_a, my_b);
  // 準備ができたらサクセサーを右に追加
  bool  recycle_as_c1 = false;
  if  (c + 1  <= r && --my_ref_count[r*my_num_blocks + c + 1] == 0) {
    recycle_as_c1 = true;
  }
  // 準備ができたらサクセサーを下に追加
  if  (r + 1  < (size_t)my_num_blocks
      && --my_ref_count[(r+1)*my_num_blocks + c] == 0) {
    if  (recycle_as_c1) {
      my_tg.run(FwdSubFunctor{my_tg, my_N, my_num_blocks,
                              BlockIndex(r+1, c), my_x, my_a, my_b,
                              my_ref_count});
    } else  {
      *my_index = BlockIndex{r+1,c};
      my_tg.run(*this);
    }
  }
  if  (recycle_as_c1) {
    *my_index = BlockIndex{r,c+1};
    my_tg.run(*this);
  }
}
```

図 12-18. 前方置換の task_group ベースのバージョンにおける関数定義。サンプルコード: migration/migrate_recycling.cpp

スケジューラーのバイパス

古い TBB タスク API で使用されるもう 1 つの手法は、*スケジューラーのバイパス*です。`task::execute()` メソッドはポインターを返します。この章の前半の例では、図 12-11 に示すように、常に `nullptr` を返しました。`nullptr` を返しても効果はありません。ただし、ポインターが `null` でない場合、スケジューラーは返されたタスクを現在のスレッドで実行する次のタスクとして使用します。タスクの再利用と同様に、この手法はオーバーヘッドを削減する可能性があります。タスクの再利用によって割り当てのオーバーヘッドが削減されるのに対し、スケジューラーのバイパスでは、TBB スケジューラーで次のタスクを見つける過程をバイパスすることで、スケジュールのオーバーヘッドが減る可能性があります。図 12-19 は、スケジューラーのバイパスを使用して前方置換のスケジュールのオーバーヘッドを削減する例を示しています。

```
tbb::task*  execute() override  {
  auto  [r, c] = my_index;
  computeBlock(my_N, r, c, my_x, my_a, my_b);
  tbb::task* bypass_task = nullptr;
  // 準備ができたらサクセサーを右に追加
  if  (c  + 1  <= r  && --my_ref_count[r*my_num_blocks  + c  + 1]  == 0)
{
    bypass_task = new(allocate_additional_child_of(my_root))
                FwdSubTask(my_root, my_N, my_num_blocks, {r, c+1},
                          my_x, my_a, my_b, my_ref_count);
  }
  // 準備ができたらサクセサーを下に追加
  if  (r  + 1  < (size_t)my_num_blocks
     && --my_ref_count[(r+1)*my_num_blocks  + c]  == 0) {
    tbb::task* child_task =
      new(allocate_additional_child_of(my_root))
        FwdSubTask(my_root, my_N, my_num_blocks, {r+1, c},
                  my_x, my_a, my_b, my_ref_count);
    if  (bypass_task == nullptr)
      bypass_task = child_task;
    else
      tbb::task::spawn(*child_task);
  }
  return  bypass_task;
}
```

図 12-19. 前方置換の例の古い低レベルタスク API を使用してタスクをバイパスする execute 関数（修正済み。サンプルコード: migration/migrate_bypass_tasks.cpp

この技法に類似したものが、現在 TBB のプレビュー機能として存在します。プレビュー機能はまだ製品化されていないため、この機能は変更されるか、完全に削除される可能性があります。新しいアプローチは遅延タスクに依存しており、これについては 6 章で詳しく説明します。`task_group::run` を呼び出す代わりに、`task_group::defer` を呼び出して `tbb::task_handle` を作成します。`task_group::run` 呼び出しで開始されたタスク内で実行される関数から `tbb::task_handle` が返されると、返された `task_handle` はその関数を実行したスレッドで直ちに実行されます。`defer` と `run` は同じ `task_group` オブジェクトで呼び出す必要があります。

このプレビュー機能を有効にするには、図 12-20 の最初の 2 行に示すように、TBB ヘッダーをインクルードする前に機能固有マクロを定義します。

```
#define TBB_PREVIEW_TASK_GROUP_EXTENSIONS 1
#include <tbb/tbb.h>
```

図 12-20 は、スケジューラーのバイパスと task_group を使用して更新された task_group バージョンのコードを示しています。

```
#define TBB_PREVIEW_TASK_GROUP_EXTENSIONS 1
#include <tbb/tbb.h>

tbb::task_handle  fwdSubTGBody(tbb::task_group&  tg,
                       int  N, int  num_blocks,
                       const  std::pair<size_t, size_t> bi,
                       std::vector<double>&  x,
                       const  std::vector<double>&  a,
                       std::vector<double>&  b,
std::vector<std::atomic<char>>&  ref_count) {
  auto  [r, c] = bi;
  computeBlock(N, r, c, x, a, b);

  tbb::task_handle deferred_task;

  // 準備ができたらサクセサーを右に追加
  if (c + 1  <= r && --ref_count[r*num_blocks + c + 1] == 0) {
    deferred_task = tg.defer([&, N, num_blocks, r, c]() {
                  return  fwdSubTGBody(tg, N, num_blocks, BlockIndex(r, c+1),
x, a, b, ref_count);
                });
  }
  // 準備ができたらサクセサーを下に追加
  if (r + 1  < (size_t)num_blocks && --ref_count[(r+1)*num_blocks + c] ==
0) {
    if (deferred_task)
      tg.run([&, N, num_blocks, r, c]() {
        return  fwdSubTGBody(tg, N, num_blocks,
                       BlockIndex(r+1, c), x, a, b, ref_count);
      });
    else
      deferred_task = tg.defer([&, N, num_blocks, r, c]() {
                  return  fwdSubTGBody(tg, N, num_blocks,
                        BlockIndex(r+1, c), x, a, b, ref_count);
                });
  }
  return  deferred_task;
}
```

図 12-20. 遅延タスクを使用した前方置換の例のタスクをバイパスする execute 関数（修正済み。サンプルコード: migration/migrate_bypass_tasks.cpp

タスクの再利用と同様に、バイパスを使用すると、どの API が使用されていてもパフォーマンスがわずかに向上します。私たちのテストシステムでは、ゲインは約 2% でした。

タスクの継続

古い低レベルの TBB タスク API で利用できたもう 1 つの機能は、継続です。継続とは、別のタスクが完了した後にのみ実行されるようにスケジュールされたタスクです。oneTBB でこのような依存関係を表現する方法は、フローグラフ API を使用するか、この章の前方置換の例で行ったように手動で参照をカウントすることです。

タスクの依存関係を管理

TBB コミュニティーの一部の人々は、タスクの依存関係を作成および管理する直接的なアプローチを再導入する API に関心を示しており、その方向で作業が進行中です。進捗状況の詳細については、oneTBB コミュニティーの議論に注目してください。

oneTBB と TBB を併用する

oneTBB に移行すると、最新の機能とバグ修正にアクセスできるようになります。したがって、移行を強くお勧めします。ただし、同じアプリケーション内の異なるコンポーネントから oneTBB と TBB ランタイム・ライブラリーの両方を使用することはできます。同じアプリケーションで両方のライブラリーを使用できる理由を説明するには、実装の詳細に触れる必要があるため、ここでは詳しく説明しません。しかし、より高いレベルでは、oneTBB と TBB のオープンソース実装は、共有ライブラリーからエクスポートされるシンボルに一意の名前が付けられるよう、名前空間を慎重に使用しています。

これらのシンボルは、両方のライブラリーにリンクするときに競合することはありません。TBB ヘッダーでコンパイルされたアプリケーションのコンポーネントは、TBB 共有ライブラリーによってエクスポートされたシンボルを使用し、oneTBB ヘッダーでコンパイルされたアプリケーションのコンポーネントは、oneTBB 共有ライブラリーによってエクスポートされたシンボルを使用します。

ただし、10 章で詳しく説明しているように、TBB を使用する主な利点の 1 つは、構成の可能性です。TBB と oneTBB を組み合わせると、2 つの異なるグローバル・スレッド・プールが生成されます。これらのプールと関連付けられたタスクアリーナは互いを認識しないため、TBB と oneTBB の両方を併用すると、オーバーサブスクリプションが発生する可能性があります。 それでも、TBB は他のランタイムと混在しても問題が起こらないように設計されているため、オーバーサブスクリプションは制限されます。各ライブラリーは、デフォルトで `P-1` 個のワーカースレッドを作成します。ここで、`P` はハードウェア・スレッドの数です。したがって、`P` 個のコアを持つシステムでは、結合されたライブラリーによって `2x(P-1)` 個のワーカースレッドが作成されます。

まとめ

この章では、オリジナルの TBB から現在の TBB（oneTBB）に移行するための重要な考慮事項と手順について説明しました。最新の C++ 標準、特に C++11 以降との整合性を強調し、標準のアトミック・インターフェイスの採用や `task_scheduler_init` などの古いクラスの削除など、重要な変更点を説明しました。これらの更新は、使いやすさを向上させるだけでなく、開発者が現在の TBB と最新の C++ で利用可能な最新の機能とセキュリティーの改善を活用できるようにします。

私たちは皆、最新の C++ とともに進化し、並列コンピューティングを最大限に活用することに重点を置いています。

付録 A 歴史とインスピレーション

この付録では、TBB の歴史について、まず、TBB から今日の「oneTBB」への移行について触れて、次に TBB の最初の 10 年間を振り返り、最後に TBB の先駆けと、きっかけとなったテクノロジーについて考察します。これら 3 つの節は、2025 年、2016 年、2006 年からの見解として考えることができます。それぞれの節は重要な歴史に関連しており、興味のある方は TBB をより深く理解することができます。これらを楽しんでいただき、TBB に関する理解が深まることを願っています。

現在の TBB への変革

10 年以上にわたって広く採用され、成功を収めた後も TBB の中核部分は価値を保ち続けましたが、並列処理の重要な基礎をサポートするため C++ が近代化された結果、プロジェクトには不要になったレガシー機能もいくつかあります。

2019 年頃、TBB プロジェクトは、TBB を C++11 の機能に依存するように変更し、プログラミングの基盤として C++11 が使用されたため、不要になった TBB の重要なサポートを削除することを決定しました。多くのインテルのプロジェクトがさまざまな理由（多くは oneAPI のサポート）でプロジェクトを変革し、「one」を採用したことから、このプロジェクトは「oneTBB」と名付けられました。TBB で「one」が使用されたのは、C++ の新しい基本的な並列処理サポートを利用する変革のためです。

私たちはすぐに書籍を改訂する作業に着手しました。TBB のエンジニアの多くはロシアにいたため、2022 年のでき事により私たちの取り組みは遅延しました。プロジェクトは 2024 年までに再開され、本書を完成させることができました。

C++ の進化は続いていますが、C++ の変更が TBB にそれほど大きな影響を与えるとは考えていません。これは単に、C++11 以降では並列処理の基礎が備わっているため、TBB などのライブラリーは、ハードウェアへの低レベル・インターフェイス（元の TBB のアトミックやロックなど）を提供して移植する必要なく、並列処理の基礎を活用できるためです。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5

現在の TBB は、2006 年の開始当初と同じ理由で価値を持ち続けていますが、その基盤にはより強力な C++ が組み込まれています。これは素晴らしい組み合わせです。

この付録の残りでは、TBB の起源に関する初期の詳細と、TBB が登り始めた巨人たちの肩に対する功績についてさらに詳しく説明します。

「孵化から飛翔へ」の 10 年

この付録の冒頭部分は、James Reinders が TBB の 10 周年（2016 年半ば）に書いた記事を基に作成されています。TBB のストーリーは、偉大な技術的作業に挑む過去と未来の多くの重要な決断と呼応しており、そのストーリーは教育的でもあります。以下は、James の独り言を少し興味深い逸話を交えてアレンジしたものですが、TBB を使用するだけであれば、これらを知る必要は全くありません。

#1 TBB のインテル内部での革命

TBB は、オープンソースを採用したインテル初の商業的に成功したソフトウェア製品であり、継続的なリーダーシップにより、TBB は最近 Apache ライセンスに移行しました。

私たちは最初から TBB をオープンソース化したいと思っていましたが、2006 年にリリースした時点ではまだ準備ができていませんでした。オープンソース・プロジェクトは、私たちの小さなチームにとっても、インテルにとっても新しいものでした。
私たちはまず、強力な TBB を作成し、それを 2006 年半ばに製品としてリリースすることに注力しました。リリース後、私たちはビルドシステムの改訂、コードのクリーンアップ（コメント付加）、そしてソースコードを理解して貢献したいと考えている他のユーザーを招待するのに役立つさまざまなことに重点を移しました。そして 2007 年半ばにオープンソース化するという目標を掲げました。しかし新たな問題が発生しました。TBB は顧客の間ですぐに人気となりました。私たちはオープンソース化への意欲を顧客に公開していたため、TBB に対する顧客の関心はさらに高まりました。私たちの成功はすぐにインテル社内で話題となり、経営陣の一部から「なぜこれほど成功した製品のソースコードを無料で提供するのか」という疑問が投げかけられました。James Reinders は、チームからの事実と数字を武器に、オープンにすべき理由を大胆に数多く示しました。しかし、それは間違いで、James は 2006 年が終わるまでに必要な許可を得ることができませんでした。彼は傷を癒し、私たちは最終的に、ただ一つのことを証明すればいいだけだと気づきました: TBB は、オープンソース化した場合、そうでない場合よりも、はるかに広く採用されるでしょう。結局のところ、開発者はプログラミング・モデルを採用するときに、コードの将来そのものを賭けることになります。おそらく、オープン化は、他のほとんどのソフトウェアよりもプログラミング・モデルにとっては重要です。私たちはこの点を理解していましたが、これが本当に重要なことであり、TBB をオープンソース化する必要があることを理解するために必要なことのすべてであることを、経営陣に明確に伝えることができませんでした。この視点を武器に、James は私たちの提案を承認しなければならな

かった上級副社長を驚かせました。彼は、オープンソース化の有無による TBB の導入予測の比較を示す簡単なグラフが描かれた紙 1 枚だけ持って現れ、彼女を驚かせました。私たちは、この重要なプログラミング・モデルをオープンソースで提供しなければ、TBB は 5 年以内に消滅し、置き換えられるだろうと予測しました。私たちはオープンソース化することの大きな成功を予測していました（実際には成功をはるかに過小評価していたのですが）。上級副社長の Renee James は彼の 2 分間のプレゼンを聞き、彼を見て尋ねました。
「なぜ最初にこれを言わなかったの？もちろん、そうすべきです。」
彼は、それが 2 か月前に発表した、元々は長すぎる 20 枚のスライドから成るプレゼンテーションの、まさに 7 枚目のスライドであることを指摘できたはずです。 代わりに、彼は「ありがとう」と言うことに止めました。残りは歴史です。私たちは、その時点で最も人気のあるオープンソース・ライセンスを選択しました:
クラスパス例外付きの GPL v2（C++ テンプレート・ライブラリーにとっては重要）。10 年後、TBB を Apache ライセンスに移行しました。ユーザーや貢献者のコミュニティーからは、これが現代の TBB を使用するのに適切なライセンスであるという多くのフィードバックが寄せられています。

#2 TBB の最初の並列処理革命

TBB によって提供された並列処理の最初の革命は、完全な構成の可能性を備えた完全な C++ サポートを提供しながら、タスクスチールの抽象化を採用したことでした。

OpenMP は非常に重要ですが、構成可能ではありません。これは、長期に渡って影響を及ぼす重大な過ちであり、OpenMP は非常に重要ですが、互換性を重視しているため、変更することはできません。James は、1997 年以降、OpenMP をまとめ上げ、レビュー、推進に協力した他のすべての人々とともに、OpenMP の過ちに加担していたとよく言っています。ハードウェアの並列処理が増えるにつれて、ネストされた並列処理の重要性が増すことを、私たちは皆見落としていました。1997 年当時は、それは全く懸念事項ではありませんでした。

構成可能であることは、TBB の最も素晴らしい機能です。オーバーサブスクリプションやネストされた並列処理などについて心配する必要がないことの重要性は、いくら強調しても過ぎることはありません。TBB は、アプリケーションの構成の可能性を要求する特定の開発者コミュニティーに徐々に革命をもたらしています。
長い間 OpenMP をベースにしてきたインテル・マス・カーネル・ライブラリー（MKL）は、まさにこの理由から、TBB 上に構築されたバージョンを提供しています。さらに新しいオープンソースのインテル・データ・アナリティクス・ライブラリーでは、常に TBB と TBB を利用した MKL が使用されます。実際、TBB は Python の一部のバージョンでも使用されています。

もちろん、TBB の中核となるタスク・スチール・スケジューラーこそが真の魔法です。HPC の顧客は、専用コアでアプリケーションを実行する際に最大限のパフォーマンスを引き出すことに懸念を抱いていますが、TBB は、実行するコアを共有する場合に並列処理をうまく機能させるにはどうすればよいかという、HPC ユーザーが決して心配しない問題に取り組んでいます。8 つのコアで実行されており、アプリケーションの実行中に 1 つのコアでウイルス

チェッカーが実行されていると考えてみてください。スーパーコンピューターではそのようなことは決して起こりませんが、ワークステーションやラップトップでは頻繁に起こります。TBB タスク・スチール・スケジューラーの動的な性質がなければ、このようなプログラムはウイルスチェッカーが動作した時間だけ実行を遅延されるだけです。なぜなら、実質的にアプリケーション内のすべてのスレッドが遅延されるからです。8 つのコアで TBB を使用する場合、1 つのコアで継続時間 TIME の中断が発生すると、アプリケーションが TIME/8 だけ遅延される可能性があります。この現実世界の柔軟性は非常に重要です。

最後に、TBB は C++ に並列処理を完全に取り入れた C++ テンプレート・ライブラリーです。TBB の C++ への取り組みは、C++ 標準の変更を促すきっかけとなりました。おそらく私たちの最大の関心ごとは、TBB がいつの日かスケジューラーとそれを使用するアルゴリズムだけになることです。TBB には、STL の一部を並列化したり、真に移植可能なロックやアトミックを作成したり、メモリー割り当ての欠陥に対処したり、C++ に並列性をもたらすその他の機能など、最終的には標準言語の一部になる可能性があり、またそうなるべきものが沢山あります。もしかしたら TBB の方が多いかもしれません。時間が明らかにしてくれるでしょう。

#3 TBB の並列処理の 2 番目の革命

TBB によってもたらされた並列処理の 2 番目の革命は、バルク同期プログラミングよりも優れた代替手段を提供することでした。

TBB のタスク・スチール・スケジューラーは称賛に値しますが、アプリケーションで最も頻繁に使用されるアルゴリズムは、実行時に多くの同期が行われるように構成されています。これは、並列プログラミングが長年にわたってどのように成功してきたかという点における、時代の兆候です。しかし、並列処理の量が増加するにつれて、これはスケーリングの追求において大きな障害となってきました。 より良いアプローチは、データの流れを表現し、最小限のレベルの同期を要求することです。TBB に加えて、TBB フローグラフは並列プログラミングにおけるこの重要な新しい革命のリーダー的存在です。この種の考え方は、あらゆる並列プログラミング・モデルが将来を適切にサポートするために必要です。

野生のカナリア

オリジナルの TBB 書籍（2007 年）は、オライリーが表紙に選んだ動物、野生のカナリアを描いたオライリー・ナッツシェル・ブックでした。私たちは（著作権を侵害しない形で野生のカナリアを素材にした）T シャツ、ウェブサイト、そして 2 冊目の本（2019 年）の制作を続けました。Today's TBB（2025 年）では、野生のカナリアの写真を取り入れてこの伝統を継承しています。

#4 TBB の鳥

全く別の話ですが、私たちが書籍の表紙に採用した鳥について質問を受けることがあります。もちろん、オリジナルの TBB 本（2007 年）は、常に動物が描かれた象徴的なデザインの O'Reilly Nutshell から出版されました。オライリーは著者である James に、動物を選ぶつもりだと明言しました（これは謎めいたプロセスでした）。彼は私たちにとって意味のある動物に関するいくつかの考えを伝えてくれました。オライリーは表紙に、私たちが考えたこともなかった美しい鳥であるカナリアが描かれた絵を選びました。すぐにインテルの周りの私たちは「鳥を受け入れよう」になりました。このリフレーミングと、T シャツ、ステッカー、ウェブサイトで使用した著作権を侵害しない人気の「チャープ（chirp）」鳥については、Belinda Adkisson に感謝します。陽気な小鳥は、TBB のマスコットです。私たちは「鳥を受け入れ」、本書の表紙には実際の写真が載っています。

Embrace the Bird!

オリジナルの TBB 本の奥付には次のように書かれています:

本書のカバーの動物は、野生のカナリア(学名: Serinus canaria、アトリ科の小さな鳥)です。西ヨーロッパ沖のマデイラ諸島、アゾレス諸島および鳥の名前であるカナリア諸島に生息していることから、島カナリアや大西洋カナリアとしても知られています。カナリアという名前の由来は、ローマ帝国時代に大プリニウスがカナリア諸島へ遠征した際、島に大型の犬が多くいたことから、彼の博物誌でラテン語の canaria(犬の変化形)と呼んだことに始まります。カナリアは果樹園、農地、雑木林に生息していて、潅木や樹木に巣を作ります。

野生のカナリアは、飼育されているカナリアより色黒でわずかに大きいことを除けば、外観はほぼ同じです。胸は黄緑色で、背中は茶色で縞があります。多くの種と同様に、雄は雌より色が鮮やかです。また、雄のほうが優しい鳴き声をしています。スペイン人によって 15 世紀に島が征服された後、人々はカナリアを飼育し、品種改良を始めました。16 世紀には、ヨーロッパの至る所でペットとして重宝されました(Samuel Pepys 氏は、1661 年の日記でカナリアについて記述しています)。その後、500 年間の品種改良により、現在の一般的な山吹色のカナリアを含む、多くの種類のカナリアが生まれてきました。カナリアの寿命は約 10 年で、飼育する際に特別な注意を払う必要はありません。また、すべての鳥の中で最も鳴き声の美しい鳥と評されており、現在でもペットとして広く親しまれています。

1980 年代後半、カナリアは炭鉱夫の警報システムとして使用され、各炭坑に 2 羽のカナリアが用意されました。米国鉱山局によると、カナリアはネズミより毒ガスに敏感に反応するので、危険を察知するためによく使用されていました。鉱山のカナリアは、平時は一日中さえずっていますが、一酸化炭素の濃度が上昇すると、そのさえずりを止め様子が変わることから、炭鉱夫に炭坑から外に出るよう警報が出されたのです。

James は、TBB チームの多大な協力を得て、2007 年春に最初の TBB 書籍(*Intel Threading Building Blocks*)を執筆しました。James は、Michael と Rafa 教授と合流し、2017 年に最初に TBB に関するチュートリアルを共同で作成し、その後、より多くの教育的資料を含む 2 冊目の本(Pro TBB)を共同で完成させました。Michael と James は、特に、TBB プロジェクトが最新の C++ 標準に合わせて合理化を進め、本書(*Today's TBB*)を作成したことにより、その作業は簡素化されました。私たちは本書を、尊敬する親友で同僚でもある Rafael Asenjo の思い出に捧げます。

この節は、James がインテルの *Parallel Universe* マガジンの TBB 10 周年記念特別版(日本語未翻訳)に寄稿した記事から抜粋したものです。Parallel Universe マガジンの各号は、https://software.intel.com/parallel-universe-magazine から、日本語翻訳版は https://www.xlsoft.com/jp/products/intel/tech/documents.html#tab2 から無料でオンラインから入手できます。TBB の歴史については、

同じ号に掲載されたオリジナルの設計者による記事「インテル・スレッディング・ビルディング・ブロックの起源と進化」で詳しく知ることができます。これには、初期の TBB 1.0 設計に関する 2 つの「後悔」も含まれています。長年にわたり、TBB に関する多くの興味深い記事が *Parallel Universe* マガジンに掲載されてきました。

TBB 10 周年 – インテル Parallel Universe マガジン特集号

TBB へのインスピレーション

この付録の最後は、TBB が公開されてまだ 1 年だったときに、James Reinders が最初の TBB 書籍（2007 年）を執筆する際に作成した歴史的メモを基に作成されています。この書籍は、TBB の先駆けとなり、TBB に影響を与えたものを紹介しています。

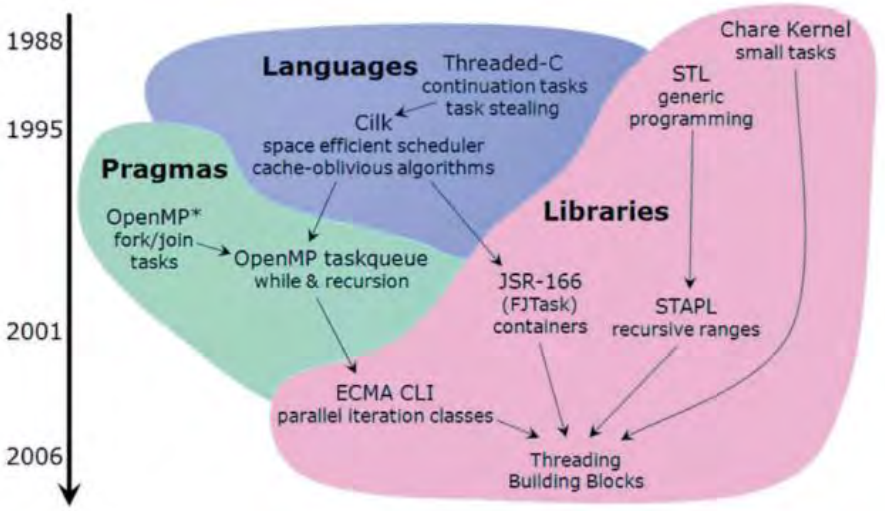


図 A-1. スレッディング・ビルディング・ブロックの設計に影響したテクノロジー

スレッディング・ビルディング・ブロックは、多くのルーツを持っています。図 A-1 では、過去 10 年間の重要な影響を説明しています。その影響は、着想として得られたものであり、McRT-Malloc を除いて、実際にソース・コード・レベルでの参照は行っていません。1988 年以前については、他の文献を参照してください。

1988、Chare Kernel、イリノイ大学アーバナ・シャンペーン校

1988 年当時は、それは単純な C ライブラリーでした。重要な概念は、chare（チャール）と呼ばれる小さなワーク単位にプログラムを分割することでした。スケジューラーは、プロセッサー上に chare を（空間と時間の両方で）効率良く割り当てました。スレッドを直接プログラムする代わりにスレッドにタスクをマップすることは重要な概念です。Chare Kernel はその後、分散型メモリーマシンに対応するためマーシャリング機能が追加され、Charm++ となりました。

1993、C++ 標準テンプレート・ライブラリー（STL）、ヒューレット・パッカード社

STL は、1993 年 11 月に ANSI/ISO C++ 標準化委員会に提出され、HP は 1994 年に自由に利用可能にし、その後、C++ 標準として採用されました。Arch Robison 氏は次のように語っています。「私は、以前 Stepanov 氏の汎用プログラミングに関する素晴らしい話を聞きました。そこで彼は、実際に汎用的な最大公約数アルゴリズムを記述する方法を説明しました［論文はそのときの話に似ていますが、より数学的に強調されています］。汎用プログラミングが優れているのは、パラメーター型ではなく、コンセプトを使用するプログラミングであることです。」STL と汎用プログラミングにおける Stepanov 氏による多くの作業は、本章の後に紹介しています。

注: Alexander Stepanov 氏は、オリジナル書籍の序文を親切に執筆してくれ、その中で、TBB の設計におけるジェネリック・ログラミングの採用を称賛しました。

1999、Java 仕様リクエスト #166（JSR-166）、Doug Lea 氏

これまで実際に標準化されてはいませんが、Lea 氏が最初にこの仕様を紹介したのは 1999 年です。FJTask は、ストック Java ライブラリーに Cilk スタイルの並列処理を追加する試みでした。JSR-166 で提案されましたが、標準には至りませんでした。

2001、Standard Template Adaptive Parallel Library（STAPL）、テキサス A&M 大学

STAPL は、再帰的な並列レンジ（pRanges）の概念と並列の汎用的なアルゴリズムを並列コンテナーとバインドするため、イテレーターの代わりにその範囲を使用する概念を導入しました。STL には、再帰的レンジは含まれていません。STAPL は、ハイパフォーマンス・コンピューティング（HPC）の典型的な分散型メモリー・アーキテクチャーを採用しているため、スレッディング・ビルディング・ブロックより複雑です。
さらに、STAPL は、並列タスクグラフを任意の順序で実行する仕様をサポートしています。この機能により、複数のスケジュール・ポリシーを使用して実行時間を最適化できます。

2004、ECMA CLI Parallel Profile、インテル

この .NET 仮想マシン用の ECMA スペックには、Arch Robison 氏によって設計された、並列反復のクラスが含まれています。

2006、McRT-Malloc、インテル研究所

スケーラブル・トランザクショナル・メモリー・アロケーターである、McRT は、スレッディング・ビルディング・ブロックで提供されているスケーラブル・メモリー・アロケーターの基礎となっています。2006 年に Hudson、Saha、Adl-Tabatabai、Hertzberg が発表した論文（http://doi.acm.org/10.1145/1133956.1133967）の 3 章と 3.1 節では、TBB のスケーラブルなメモリー・アロケーターの基礎について説明されており、これは小さい（8K 未満）オブジェクトに最適です。スケーラブルなメモリー・アロケーターは、それ以来、大幅に強化されてきました。

影響を受けた言語

1994、Threaded-C、マサチューセッツ工科大学

Parallel Continuation Machine（PCM）は、Threaded-C のランタイムサポートでした。これは、Thinking Machines Corporation のコネクション・マシン・モデル CM-5 スーパーコンピューター上で継続渡しスタイルのスレッドを提供する、C ベースのパッケージで、計算のロードバランスと局所性を改善するため一般的なスケジュール・ポリシーとしてワークスチールを使用しました。この言語は、マックギル大学およびデラウェア大学の EARTH 用の Threaded-C とは異なるものです。混同しないように注意してください。PCM については、オリジナルの Cilk マニュアルの 2 ページで簡単に説明されています。

1995、Cilk、マサチューセッツ工科大学

Cilk の最初の実装は、PCM/Threaded-C の直接の後継でした。Cilk は継続タスクのプログラミングの難しさを修正し、キャッシュ非依存アルゴリズムを使用して、キャッシュのサイズが不明であってもキャッシュへのタスク割り当てを調整する方法を生み出しました。Cilk は、非常に効率的な fork/join 並列処理をサポートする C の拡張です。そのスペース効率については、http://supertech.csail.mit.edu/papers/ cilkjpdc96.pdf で説明されています。FFTW（www.fftw.org）は、キャッシュの恩恵を受けるアルゴリズムの例です。

影響を受けたプラグマ

1997、OpenMP、主なコンピューター・ハードウェアおよびソフトベンダーのコンソーシアム

OpenMP は、マルチプラットフォームで、C と Fortran における共有メモリー並列プログラミングをサポートしており、コンパイラー・ディレクティブ、ライブラリー・ルーチン、環境変数の標準的なセットを提供します。OpenMP 以前は、多くのベンダーが同様の、しかし移植性に欠けた独自のコンパイラー・ディレクティブを提供していました。OpenMP は、fork/join の概念を具体化します。www.openmp.org を参照。

1998、OpenMP タスクキュー、Kuck & Associates (KAI)

ループを超えた OpenMP の拡張提案。この提案の改良版が採用され、OpenMP 3.0 として OpenMP に追加されたのは 2008 年のことです。

まとめ

TBB は、事前の研究を活用して情報に基づいた設計選択を行う理想的なタイミングで開発されました。並列ツールの専門家チームが、マルチコア並列処理の新たなニーズに対応するため、基盤となる堅牢なライブラリーを作成しました。マルチコア・プロセッサーの登場により TBB への関心が高まり、その実装が改良され、オープンソース化によって広範な採用が促進されました。C++ が並列処理を採用する際に、TBB の経験が重要な役割を果たしました。

並列処理は、今日のあらゆるコンピューティングの一部となっています。コンピューターにおける並列処理は、もはや新しいニュースではありません。また、C++ の並列処理のサポートも新しいニュースではありません。大量の特殊な並列処理をコンピューターに組み込むためアクセラレーターを使用するのもまた、新しいニュースではありません。あらゆる並列システムの中心には、システム全体の並列性を調整する堅牢で信頼性の高い制御が必要です。

今日の TBB は C++ プログラマーのそのニーズを解決します。TBB で実現できる強固な基盤の上に、pragma SIMD、OpenMP、CUDA、SYCL、OpenCL などのツールを使用して、ベクトル化やその他のアクセラレーションのより具体的なニーズにも対応できます。これらすべてにおいて、TBB は、アムダールの法則の悪影響を最小限に抑えるため最大限の並列処理を達成できるように支援します。

さらに詳しい文献

Acar, U., G. Blelloch, and R. Blumofe (2000).“The Data Locality of Work Stealing.” *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1‐12.

Amdahl, G. M. (1967, April).“Validity of the single-processor approach to achieving large scale computing capabilities.”*AFIP Conference Proceedings*, 30.Reston, VA: AFIPS Press, 483‐485.

An, P., A. Jula, et al.(2003).“STAPL: An Adaptive, Generic Parallel C++ Library.” Workshop on Language and Compilers for Parallel Computing, 2001.Lecture Notes in Computer Science 2624, 193‐208.

Austern, M. H., R. A. Towle, and A. A. Stepanov (1996).“Range partition adaptors: a mechanism for parallelizing STL.”*ACM SIGAPP Applied Computing Review*, 4, 1, 5‐6.

Blumofe, R. D., and D. Papadopoulos (1998).“Hood: A User-Level Threads Library for Multiprogrammed Multiprocessors.”

Blumofe, R. D., C. F. Joerg, et al.(1996).“Cilk: An Efficient Multithreaded Runtime System.”*Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 207‐216.

Boehm, H. (2006, June).“An Atomic Operations Library for C++.”C++ standards committee document N2047.

Butenhof, D. R. (1997).*Programming with POSIX Threads*.Reading, MA: Addison Wesley.

Flynn, M. J. (1972, September).“Some Computer Organizations and Their Effectiveness.”*IEEE Transactions on Computers*, C-21, 9, 948-960.

Garcia, R., J. Järvi, et al.(2003, October).“A Comparative Study of Language Support for Generic Programming.”*Proceedings of the 2003 ACM SIGPLAN Conference on Object- Oriented Programming, Systems, Languages, and Applications*.

Gustafson, J. L. (1988).“Reevaluating Amdahl’s Law.”*Communications of the ACM*, 31(5), 532‐533.

Halbherr, M., Y. Zhou, and C. F. Joerg (1994, March).“MIMD-Style Parallel Programming Based on Continuation-Passing Threads.”Computation Structures Group Memo 355.

Halbherr, M., Y. Zhou, and C. F. Joerg (1994, September).“MIMD-style parallel programming with continuation-passing threads.”*Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy.

Hansen, B. (1973).“Concurrent Programming Concepts.”*ACM Computing Surveys*, 5, 4.

Hoare, C. A. R. (1974).“Monitors: An Operating System Structuring Concept.”*Communications of the ACM*, 17, 10, 549‑557.

Hudson, R. L., B. Saha, et al.(2006, June).“McRT-Malloc: a scalable transactional memory allocator.”*Proceedings of the 2006 International Symposium on Memory Management*.New York: ACM Press, 74‑83.

Intel Threading Building Blocks 1.0 for Windows, Linux, and Mac OS‑Intel Software Network (1996).

“A Formal Specification of Intel Itanium Processor Family Memory Ordering”(2002, October).

ISO/IEC 14882:1998(E) International Standard (1998).Programming languages‑C++.ISO/IEC, 1998.

ISO/IEC 9899:1999 International Standard (1999).Programming languages‑C. ISO/IEC, 1999.

Järvi, J., and B. Stroustrup (2004, September).“Decltype and auto (revision 4).”C++ standards committee document N1705=04-0145.

Kapur, D., D. R. Musser, and A.A.Stepanov (1981).“Operators and Algebraic Structures.”*Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, 59‑63.

MacDonald, S., D. Szafron, and J. Schaeffer (2004).“Rethinking the Pipeline as Object-Oriented States with Transformations.”Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments.

Mahmoud, Q. H. (2005, March).“Concurrent Programming with J2SE 5.0.”Sun Developer Network.

Massingill, B. L., T. G. Mattson, and B. A. Sanders (2005).“Reengineering for Parallelism: An Entry Point for PLPP (Pattern Language for Parallel Programming) for Legacy Applications.”*Proceedings of the Twelfth Pattern Languages of Programs Workshop*.

Mattson, T. G., B. A. Sanders, and B. L. Massingill (2004).*Patterns for Parallel Programming*.Reading, MA: Addison Wesley.

McDowell, C. E., and D. P. Helmbold (1989).“Debugging Concurrent Programs.”*Communications of the ACM*, 21, 2.

Meyers, S. (1998).*Effective C++*, Second Edition.Reading, MA: Addison Wesley.Musser, D. R., and A. A. Stepanov (1994).“Algorithm-Oriented Generic

Libraries.”
Software‐Practice and Experience, 24(7), 623‐642.

Musser, D. R., G. J. Derge, and A. Saini, with foreword by Alexander Stepanov (2001).*STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Second Edition.Boston, MA: Addison Wesley.

Narlikar, G., and G. Blelloch (1999).“Space-Efficient Scheduling of Nested Parallelism.”*ACM Transactions on Programming Languages and Systems*, 21, 1, 138‐173.

OpenMP C and C++ Application Program Interface, Version 2.5 (2005, May).Ottosen, T. (2006, September).“Range Library Core.”C++ standards committee
document N2068.

Plauger, P. J., M. Lee, et al.(2000).*C++ Standard Template Library*.Prentice Hall.Rauchwerger, L., F. Arzu, and K. Ouchi (1998, May).“Standard Templates Adaptive
Parallel Library.”*Proceedings of the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA.Also Lecture Notes in Computer Science, 1511, Springer-Verlag, 1998, 402‐410.

Robison, A. D. (2006).“A Proposal to Add Parallel Iteration to the Standard Library.
“Robison, A. (2003, April).“Memory Consistency & .NET.”Dr. Dobb’s Journal.Samko, V. (2006, February).“A proposal to add lambda functions to the C++
standard.”C++ standards committee document N1958=06-028.

Schmidt, D. C., and I. Pyarali (1998).Strategies for Implementing POSIX Condition Variables on Win32.Department of Computer Science, Washington University, St. Louis, MO.

Schmidt, D. C., M. Stal, et al.(2000).Patterns for Concurrent and Networked Objects.Pattern-Oriented Architecture, 2.

Shah, S., G. Haab, et al.(1999).“Flexible Control Structures for Parallelism in OpenMP.”*Proceedings of the First European Workshop on OpenMP*.

Siek, J., D. Gregor, et al.(2005).“Concepts for C++0x.”

Stepanov, A. A., and M. Lee (1995).“The Standard Template Library.”HP Laboratories Technical Report 95-11(R.1).

Stepanov, A. A. (1999).“Greatest Common Measure: The Last 2500 Years.”

Stroustrup, B. (1994).*The Design and Evolution of C++*, also known as D&E.Reading,

MA: Addison Wesley.

Stroustrup, B. (2000).*The C++ Programming Language*, Special Edition.Reading, MA: Addison Wesley.

Stroustrup, B., and G. Dos Reis (2005, April).“A Concept Design (rev.1).” Technical Report N1782=05-0042, ISO/IEC SC22/JTC1/WG21.

Stroustrup, B., and G. Dos Reis (2005, October).“Specifying C++ concepts.” Technical Report N1886=05-0146, ISO/IEC SC22/JTC1/WG21.

Su, E., X. Tian, et al.(2002, September).“Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures.”Fourth European Workshop on OpenMP, Rome.

Sutter, H. (2005, January).“The Concurrency Revolution.”Dr. Dobb’s Journal.Sutter, H. (2005, March).“The Free Lunch Is Over: A Fundamental Turn Towards Concurrency in Software.”Dr. Dobb’s Journal.

Voss, M. (2006, December).“Enable Safe, Scalable Parallelism with Intel Threading Building Blocks’ Concurrent Containers.”

Voss, M. (2006, October).“Demystify Scalable Parallelism with Intel Threading Building Blocks’ Generic Parallel Algorithms.”

Willcock, J., J. Järvi, et al.(2006).“Lambda Expressions and Closures for C++.” N1968-06-0038.

用語集

抽象化: TBB における抽象化は、ワークをプログラマーに適したワークとランタイムに任せることで、最適なワークの分離に役立ちます。このような抽象化の目的は、コードを書き直すことなく、各種マルチコアおよびメニーコア・システム、さらにはヘテロジニアス・プラットフォーム上でスケーラブルな高いパフォーマンスを実現することです。このような注意深い役割分担により、プログラマーは並列処理の可能性を公開し、ランタイムはそれらをハードウェアにマッピングする責任を負います。抽象化に基づいて記述されたコードでは、キャッシュサイズ、コア数、さらには処理ユニット間のパフォーマンスの一貫性に関するパラメーター化は不要になります。

アクセラレーターは、汎用 CPU だけでは不可能な特定の種類の計算をより効率良く実行するため、特別に作成された特殊なハードウェアです。大規模な並列処理は、特定の計算を大幅に高速化する能力の最も一般的な源です。

アフィニティー: 特定のソフトウェア・スレッドを特定のハードウェア・スレッドに関連付ける方式の仕様であり、通常はより高いまたは予測可能なパフォーマンスを得ることを目的とします。親和性とも呼ばれます。

アフィニティーの仕様には、競合を抑制するため通信距離を最大限に広げたり（スキャッター）、最小にするため密にパックする（コンパクト）概念が導入されています。OpenMP は、抽象的な手法から完全な手動による制御まで、さまざまなレベルでアフィニティー制御をサポートしています。Fortran 2008 ではアフィニティーを制御できませんが、インテルは OpenMP の制御を “do concurrent” に適用できるようにしました。スレッディング・ビルディング・ブロック（TBB）は、ループ間のアフィニティーを抽象化するバイアス機能を提供します。

アムダールの法則: スピードアップはワークを並列化できないシリアル部分によって制限されます。プログラムの 3 分の 2 が並列実行でき、残りの 3 分の 1 がシリアル実行される場合、並列化によるプログラムのスピードアップの上限は 3 倍であり、同じワークを使用する限りそれを超えることはありません。問題サイズをスケールして、プログラムの並列領域により多くの要求が配置される場合、アムダールの法則の上限はそれほど悪くないように見えます。**グスタフソンの法則**を参照。

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5

アービトレーターは、重複する計算リソースを要求したときに、ソフトウェア・コンポーネント間の競合を解決します。TBB は、優先度を考慮した比例アービトレーターを使用して、アリーナからのワーカースレッドに対する要求を満たします。oneAPI ツールキットは、CPU 計算リソースへのアクセス要求を許可し、TBB で使用できるアービトレーターであるスレッド・コンポーザビリティー・マネージャー（TCM）と、OpenMP、OpenCL、SYCL の一部の実装を提供します。

アトミック操作は、他のスレッドからの干渉を受けることなく、単独で発生したかのように見えることが保証された操作です。例えば、プロセッサーが提供するメモリーのインクリメント操作があります。この操作は、メモリーから値を読み込み、レジスターをインクリメントして、それをメモリーに書き戻します。アトミック・インクリメントでは、読み込みと書き込みの間にそのメモリー位置に他の命令操作が行われることがなく、同じメモリー値が保持されることが保証されます。

バリア: 計算がいくつかのフェーズで構成される場合、いずれのスレッドも次のフェーズへ移行する前に、すべてのスレッドがそのワークを完了していることを保証することが求められます。バリアはこれを保証する同期形式であり、最後のスレッドが到達するまで、バリアの位置ですべてのスレッドは待機します。すべてのスレッドがバリアに到達すると、処理を継続します。バリアは、アトミック操作を使用して実装されます。例えば、すべてのスレッドが共有変数をインクリメントして、変数の値がスレッド数と同一でない場合、バリアで同期する必要があります。そして、最後にバリアに到達したスレッドは、バリアをゼロにリセットしてブロックされているすべてのスレッドを解放できます。

ブロックは次の 2 つの意味で使用されます: (1) いくつかの同期イベントを待機する間、スレッドが処理を続行できない状態。(2) メモリーの領域 2 番目の意味では、ループを適切な粒度の並列タスクに分割する意味でも使用されます。

キャッシュは、メモリーシステムの一部であり、データを一時的に高速なメモリーに保存することで、将来そのデータが必要な時により高速にアクセスすることを可能にします。キャッシュにデータが無いと、上位階層のメモリーシステムからデータをフェッチするまでの遅延が生じます。キャッシュは通常自動的に管理され、プログラムの時間と空間の局所性を高めるように設計されています。

現代のコンピューター・システムは、複数レベルのキャッシュを搭載します。

キャッシュ・フレンドリーとは、問題サイズが増加するに応じてパフォーマンスも向上するが、帯域幅の上限に達するとパフォーマンスも平坦になるアプリケーションの特性です。

キャッシュラインは、キャッシュ内にデータが取得および保持される単位です。空間の局所性を生かすため、通常はワードよりも大きくなります。一般的な傾向として、キャッシュライン・サイズは増加傾向にあります。

これは通常、少なくとも 2 つの倍精度浮動小数点数を保持できる大きさですが、現在の設計では 8 つを超える数を保持できる可能性は考えられません。大きなキャッシュラインは、メインメモリーからの効率良い転送を可能にしますが、フォルス・シェアリングなどパフォーマンスの低下を招く問題を悪化させる可能性があります。

キャッシュ非依存アルゴリズムとは、異なるレベルの異なるサイズのキャッシュなど、複数のマシンメモリー構成で修正なしで適切に実行されるアルゴリズムです。このようなアルゴリズムは、コンパクトなメモリーを再利用するように注意深く設計されているため、このようなアルゴリズムはキャッシュ非依存と呼ぶ方が理にかなっていると思われます。*「意識しない」*という用語は、そのようなアルゴリズムがキャッシュサイズや相対速度などのメモリー・サブシステムのパラメーターを認識しないことを指します。これは、特定のキャッシュ・ハードウェア向けに慎重にブロック・アルゴリズムを開発した以前の取り組みとは対照的です。

キャッシュ非対応とは、キャッシュメモリーを効果的に利用せず、非効率的なデータ・アクセス・パターンにつながるアプリケーションの特性です。これにより、レイテンシーが増加し、パフォーマンスが低下する可能性があります。問題として、ランダム・メモリー・アクセス・パターンの広域な使用、非常に大きなデータ構造、頻繁なコンテキスト切り替え、高いメモリー帯域幅の要求、その他非効率的なアルゴリズムなどが挙げられます。キャッシュ非対応のアプリケーションは、特にメモリーアクセス速度が重要なシステムでパフォーマンスを大幅に低下させる可能性があります。そのようなアプリケーションを最適化するには、データの局所性向上、ランダムアクセスの最小化、キャッシュ階層を効果的に活用するアルゴリズムの設計が必要です。

クラスターは、高速相互接続を介して通信を行う分散メモリーを備えたコンピューターのセットです。個々のコンピューターは、**ノード**と呼ばれることもあります。TBB はクラスター内のノードレベルで使用されますが、複数のノードが TBB でプログラムされて接続されるのが一般的です（通常は MPI を使用）。

通信: ソフトウェア・タスクやスレッド間のデータ交換、または同期を示します。並列プログラミングにおいて、通信コストがスケーリングを妨げる主な要因であることを理解することは重要です。

構成の可能性（Composability）: 障害や不当な衝突（衝突が無いのが理想的）を引き起こすことなく、2 つのコンポーネントを相互に使用するようにすること。構成の可能性が制限される場合、テスト時ではなくビルド時に完全に診断できる場合が最適です。構成の可能性の問題は、構成不可能なシステムにおいて実行時にしか露見しないことです。システム障害、プログラミング・モデル、もしくはソフトウェア・コンポーネントを参照できます。

並行性（同時実行/コンカーレント）とは、物事が論理的に同時に起こることを意味します。2 つのタスクが同時期に同じポイントで論理的にアクティブである場合、平行であると考えることができます。**並列（パラレル）**とは対照的です。

コア: マルチコア・プロセッサー上の独立したサブプロセッサー。コアは、同一プロセッサーの他のコアから（少なくとも 1 つ）分離および分岐した制御フローをサポートする必要があります。

CUDA は、Nvidia 社が開発した、Nvidia アクセラレーターを対象とした独自の並列計算プラットフォームです。

データ並列処理は、タスクよりもデータ主導の並列処理を試みるアプローチです。しかし、現実の並列アルゴリズム開発を成功に導く戦略は、データの並列性を見つけることに注目します。これは、データ分割は（異なるデータユニット向けのタスクを生成）スケールし、機能分割は（異なる機能向けのヘテロジニアス・タスクを生成）スケールしないためです。
アムダールの法則とグスタフソンの法則を参照。

デッドロックは、プログラミングのエラーです。少なくとも 2 つのタスクがお互いに待機し、互いのタスクが処理されるまでそれぞれが処理を再開できない場合、デッドロックが発生します。コードが複数のミューテックスを要求する場合、容易に発生する可能性があります。例えば、各タスクは他のタスクが要求するミューテックスを保持することができます。

決定論的とは、予測通りに動作するアルゴリズムである決定論性を備えるアルゴリズムを指します。決定論性を持つアルゴリズムは、特定の入力が与えられると常に同じ出力を生成します。数学演算の精度と、並列化などの最適化による操作の順番の並べ替えにより、“同じ” であるという定義が重要になることがあります。 元のプログラムと最終的な並行プログラムの数学演算の順番により計算の答えが異なる場合、それは “丸め” 誤差に起因することがあります。同時実行性は**非決定論的**なアルゴリズムにつながる唯一の要因ではありませんが、実際に多くの場合同時実行性が原因となります。シーケンシャルなセマンティクスを備えたプログラミング・モデルを使用し、適切なアクセス制御でデータ競合を排除することは、一般に “丸め” 誤差以外の同時実行性の影響を排除します。

分散メモリーは、物理的に異なるコンピューターに配置されたメモリーです。リモート・コンピューター上のメモリーにアクセスするには、メッセージパッシングなどの間接的なインターフェイスが必要ですが、ローカルメモリーには直接アクセスが可能です。分散メモリーは一般にクラスターによってサポートされ、クラスターはコンピューターの集合体であると考えられています。システムに装着されるコプロセッサーのメモリーもホストから直接アクセスすることができないため、機能的に分散メモリーの一種であると考えることができます。

DSP（デジタル信号プロセッサー）は、フィルター、FFT、アナログからデジタルへの変換など、無線通信に関連するデジタル信号処理タスク専用に設計された計算デバイスです。CPU と並んで DSP の計算能力により、ヘテロジニアス・プラットフォームの初期の例がいくつか生まれ、DSP を制御し対話するさまざまなプログラミング言語の拡張が生まれました。OpenCL は、DSP の計算能力を活用するプログラミング・モデルです。**ヘテロジニアス・プラットフォーム**を参照。

emacs は、James によると、世界で最も優れたテキストエディターであり、オープンソース化されています。vi エディターよりも優れています。“emacs” は、James が使用するコンピューター・システムに最初にインストールするパッケージです。

驚異的並列とは、タスク間の同期がわずかもしくは全く無い、独立した多数のタスクに分解可能なアルゴリズムのことです。

ExaFLOP/s = 1 秒あたり 10^18 回の倍精度（64 ビット）浮動小数点演算。

ExaFLOP = 10^18 回の倍精度（64 ビット）浮動小数点演算。

ExaOP/s = 1 秒あたり 10^18 回の数学演算。

ExaOP = 10^18 回の数学演算。

フォルス・シェアリング: 2 つの異なるコアで動作する 2 つのタスクがメモリーの異なる位置に書き込みを行う際に、それらのメモリー位置が同じキャッシュラインに配置される場合、ハードウェアはキャッシュの一貫性を維持しようとします。その結果、タスクは実際にはデータを共有しないにもかかわらず、予期しないプロセッサー・コア間の通信が発生しパフォーマンスが低下します。

Far メモリー: NUMA システムでは、近くのメモリーよりもアクセス時間が長いメモリー。メモリーのいずれの領域が近いか遠いかは、コードが実行されるプロセスによって異なります。11 章では、このメモリーを（ローカルメモリーとは対照的に）非ローカルメモリーとも呼んでいます。

浮動小数点数は、数値に利用可能なビット（仮数部）と固定位置の小数点を左右にシフトする（指数部）ことによって、精度を低下させる代わりに数値の表現範囲を広めることを特徴とするコンピューターの数値方式です。対照的に、固定小数点表現には指数がないため、すべてのビットを数値 (仮数) として使用できます。

浮動小数点操作には、浮動小数点数に対して実行される加算、乗算、減算などがあります。

FLOP/s = 1 秒あたりの倍精度（64 ビット）浮動小数点操作。

FLOPs = 倍精度（64 ビット）浮動小数点操作。

フォワード・スケーリングとは、単純に新しいコンパイラーで再コンパイルしたり、新しいライブラリーと再リンクすることで、将来のハードウェアで向上する並列性を活用できるように、すでにスレッド化やベクトル化されているプログラムやアルゴリズムをスケーラブルにする概念です。並列プログラムを記述する場合、適切な抽象化により並列性を表現することは、将来に向けたスケーリングを可能にするうえで重要です。

FPGA(Field Programmable Array) は、デバイスがプログラムされるまで互いに接続されていない多数のゲート (多くの場合、DSP、浮動小数点ユニット、ネットワーク・コントローラなどの高レベルの構成) を統合したデバイスです。プログラミングはもともと、システムの起動時に一度だけ実行されることを目的としたチップ全体のプロセスでしたが、最新の FPGA は部分的な再構成をサポートしており、新しいプログラムが動的にロードされることが多くなっています。従来、FPGA は、設計内の多数の個別のチップを 1 つの FPGA に統合する手段と考えられており、通常ボードのスペース、電力、および全体的なコストを節約します。そのため、FPGA は、ボードまたはチップレベルで回路を設計するのに使用されるツールと同様のツール、つまり高水準記述言語 (VHDL や Verilog など) を使用してプログラムされます。最近では、FPGA を計算エンジンとして活用するため、OpenCL プログラミング・モデルが使用されています。

将来性のある (future-proofed): プログラム自身に大きな変更を加えることなく、将来のコンピューター・アーキテクチャーの変化に対応できるように記述されたコンピューター・プログラムのことです。一般により抽象化されたプログラミング方式ほど、プログラムの**将来性を高め**ます。何らかの方法でコンピューター・アーキテクチャーに限定される低レベルのプログラミング方式は、変更なしでは将来に備えることはできません。しかし、抽象化による**将来性のある**アプローチには、効率とのトレードオフがあります。

GigaFLOP/s = 1 秒あたり 10^9 回の倍精度 (64 ビット) 浮動小数点操作。

GigaFLOP = 10^9 回の倍精度 (64 ビット) 浮動小数点操作。

GPU (グラフィック・プロセッシング・ユニット) は、照明、変換、切り抜き、レンダリングなどのグラフィックスに関連する計算を改革するために設計された計算デバイスです。GPU の計算機能は、もともと汎用計算デバイス (CPU) とディスプレイの間にある「グラフィカル・パイプライン」で使用するためにのみ設計されました。結果をディスプレイに送信せずに計算を行うプログラミング・サポートの出現と、それに続く GPU 設計の拡張により、さらに一般化された計算機能が多くの GPU に関連付けられるようになりました。OpenCL と CUDA は、GPU の計算能力を活用するために使用される 2 つの一般的なプログラミング・モデルです。**ヘテロジニアス・プラットフォーム**を参照。

粒度とは、粗粒度の並列処理や細粒度の並列処理、あるいは粒度サイズのように、新しいタスクに移行したり、同期したりする前に「どれだけのワーク」が完了するかという概念を指します。プログラムは、粒度が可能な限り大きい

（スレッドが独立して実行できる）が、すべての計算リソースを完全にビジー状態に維持できるほど小さい（負荷分散）場合に、最もよくスケールします。これら 2 つの要因は互いに若干矛盾して作用するため、粒度を考慮する必要が生じます。TBB はパーティショニングを自動化しますが、プログラマーがアルゴリズムに関する知識を基に最高のパフォーマンスに調整できない完璧な世界は決して存在しません。

グスタフソン・バルシスの法則は、問題サイズが増加するに従って、計算のシリアル部分がワークの合計に占める割合が減少するという要因が**アムダールの法則**と異なります。

ハードウェア・スレッドは、独立した制御フローを備えたスレッドのハードウェア実装です。複数のハードウェア・スレッドは、複数のコアまたは、プロセッサー・コアのハイパースレッディングなどの方法によりレイテンシーを隠匿するために、単一コア上で平行もしくは同時に実行することで実装されます。後者の場合（ハイパースレッディングまたは同時マルチスレッディング（SMT））、物理コアに複数の論理コア（またはハードウェア・スレッド）が備わっていると言われます。

ヘテロジニアス・プラットフォームは、CPU のみの同種の集合ではなく、計算デバイスの混合で構成されます。ヘテロジニアス・コンピューティングは通常、GPU、DSP、FPGA などの接続されたデバイスを介して特定の加速を提供するのに使用されます。**OpenCL** も参照してください。

ハイパフォーマンス・コンピューティング（HPC）とは、ある時点で利用可能な最高パフォーマンスのコンピューティングを指し、今日では一般的に少なくとも 1 petaFLOP/秒の計算能力を意味します。HPC という用語は、スーパーコンピューティングの同義語として使用されることがありますが、スーパーコンピューティングはよりパフォーマンスが高いシステムを示します。HPC は様々な産業で利用されていますが、一般的に科学と工学における最も困難な問題の解決に役立つものとされています。多くの場合、人工知能（AI）やマシンラーニング（ML）の技術を使用する高性能データ分析ワークロードは、大規模なインスタンス化では HPC ワークロードとして最適であり、長年使用されている（従来の）HPC ワークロードと組み合わされることがあります。

ハイパースレッディングとは、2 つ以上のソフトウェア・スレッドから命令を同時にフェッチすることで、アウトオブオーダー・コア内の機能ユニットの利用率を高めるため、シングル・プロセッサー・コア上でマルチスレド処理を行うことを指します。ハイパースレッディングを利用すると、複数のハードウェア・スレッドが 1 つのコア上でリソースを共有して実行され、並列性もしくは並行性による利点が得られます。一般にそれぞれのハイパースレッドは、ハイパースレッド間の切り替えを軽量にするため、少なくとも専用のレジスターファイルとプログラムカウンターを持っています。 ハイパースレッディングはインテル社に関連しています。**同時マルチスレッディング**を参照。

レイテンシーとは、タスクが完了するまでに要する時間、つまりタスクが開始され終了するまでの時間です。レイテンシーには時間単位があります。スケールは、ナノ秒から数日までのいずれかです。低レイテンシーが一般に良いとされます。

レイテンシーの隠蔽は、コア上で実行される他のタスクがメモリーやディスク転送などの長いレイテンシーの操作の完了を待機する間に、処理ユニットに計算をスケジュールします。それぞれのタスクが完了するまでの時間は同じであるため、レイテンシーは避けることはできませんが、特定の時間内でより多くのタスクを完了することは、リソースを効率良く共有してスループットを向上できます。

負荷分散は、不均一なサイズのタスクを処理しながら、タスクをリソースに割り当てることです。負荷分散の目標は、オーバーヘッドによる無駄を最小限に抑えながら、すべての計算デバイスをビジー状態に保つことです。

負荷の不均衡（インバランス）とは、計算リソースへの不均等なワークの分散状態を指します。負荷の不均衡により、一部の計算リソースが完全に利用されないため、リソースの使用が非効率になり、アプリケーションの実行時間が長くなる可能性があります。

局所性とは、メモリー位置を有効に利用するため、離れた位置よりも隣接した位置に配置すること。これによりキャッシュライン、メモリーページなどが最大限に再利用されます。高い参照の局所性を維持することは、スケーリングにおいて重要です。

ロックは**相互排他**を実装するメカニズムです。排他実行領域に入る前に、スレッドは領域のロックの取得を試みる必要があります。すでに他のスレッドによってロックが取得済みである場合、後続のスレッドは操作を延期するかスピンすることで**ブロック**される必要があります。ロックが開放されると、後続のスレッドがロックを取得できるようになります。ロックは、それ自身が基本的な相互排他の一種でありハードウェアによって実装される**アトミック操作**を使用して実装できます。

ループ伝搬依存: 同じデータ要素（例えば、配列 element[3]）があるループ反復で書き込まれ、異なるループ反復で読み込まれる場合、ループ伝搬依存が存在します。ループ伝搬依存が存在しない場合に、そのループはベクトル化や並列化を適用できます。ループ伝搬依存が存在する場合、その方向（前の反復か以降の反復か、これは前方および後方依存と呼ばれます）と距離（読み込みと書き込みを分割する反復数）を考慮する必要があります。

メニーコア・プロセッサーとは、非常に多くのコアを持つ**マルチコア・**プロセッサーであり、実際にそのコア数は列挙されません。単に“多くの”で表されます。この用語は、一般に 32 個以上のコアを備えるプロセッサー向けに使用されていましたが、厳密には定義されていません。

メガヘルツ時代は、トランジスターの設計においてほぼ 2 年でトランジスター数が倍増するにしたがって、プロセッサーのクロックも倍増された歴史的な期間です。プロセッサーの急激なクロックスピードの上昇は、2004 年の 4GHz（4000MHz）あたりまで続いていました。以降プロセッサーの設計は、より多くのコアを追加する**マルチコアの時代**へと移行しています。

ムーアの法則とは、半導体の歴史において、集積回路のトランジスター数の密度は 2 年ごとに倍増するという観点から見た法則です。

メッセージ・パッシング・インターフェイス（MPI）は、様々な種類の並列コンピューター上でデータ交換を行うために設計された業界標準のメッセージ交換システムです。

マルチコアは複数のサブプロセッサーを持つプロセッサーであり、各サブプロセッサー（コアと呼ばれる）は少なくとも 1 つのハードウェア・スレッドをサポートします。

マルチコアの時代は、プロセッサーの設計方針がクロックの急激な上昇から、コアを追加する方向へ切り替わってからの時代です。この時代は、2005 年ごろから始まりました。

ノード（クラスター内）とは、共有メモリー・コンピューター、多くの場合単一ボードに複数のプロセッサーを搭載し、他のノードへ**クラスター**やスーパーコンピューターとして接続されます。

非決定論的（Non deterministic）: 決定論性を欠くアルゴリズムは、実行ごとに結果が異なります。同時実行性は非決定論的なアルゴリズムにつながる唯一の要因ではありませんが、実際に多くの場合、同時実行性が原因となります。詳細については、決定論性の定義をご覧ください。

不均一メモリーアクセス（NUMA）: 共有メモリー・アーキテクチャーにおいて、メモリー設計の特性を分類するために使用されます。NUMA = メモリー・アクセス・レイテンシーは、メモリーによって異なります。UMA = メモリー・アクセス・レイテンシーは、すべてのメモリーに対して同じです。**UMA** と比較してください。11 章を参照。

オフロード: 計算の一部分を FPGA、GPU または他のアクセラレーターなどの装着されたデバイスへ配置します。

OpenCL（Open Computing Language）は、異機種プラットフォーム間で実行されるプログラムを作成するためのフレームワークです。OpenCL は、オフロードと接続されたデバイスを制御するホスト API と、接続されたアクセラレーター（GPU、DSP、FPGA など）で実行されるコードを表現する C/C++ 拡張機能を提供し、接続されたデバイスが存在しないか実行時に利用できない場合に CPU にフォールバックして使用する機能を備えています。

oneAPI は、CPU、GPU、AI アクセラレーター、FPGA などのデバイスの異種プログラミング向けのオープン・スタンダードです。

oneDPL は、oneAPI プロジェクトの一部である PSTL 実装です。

oneTBB: 長年続いている TBB プロジェクトの新しい正式名称です。oneAPI の取り組みにグループ化された他のプロジェクトの命名に合わせて名前が変更されました。このプロジェクトは、Linux Foundation の一部である UXL（Unified Acceleration）Foundation の後援によるコミュニティー管理の恩恵を受けています。

OpenMP は、ほとんどのプロセサー・アーキテクチャーとオペレーティング・システム上の C、C++、および Fortran においてマルチプラットフォーム、共有メモリー、マルチスレッド・プログラミングをサポートする API です。コンパイラー・ディレクティブ、ライブラリー・ルーチン、およびランタイムを制御できる環境変数で構成されます。

OpenMP は、非営利団体である OpenMP アーキテクチャー・レビュー・ボードと賛助メンバーである主要なコンピューター・ハードウェアとソフトウェア・ベンダーによって管理されています（http://openmp.org）。

オーバーサブスクリプション: 複数のソフトウェア・スレッドを同じハードウェア・スレッドにスケジュールする必要がある場合、スレッドは同じハードウェア・スレッドを同時に占有することはできないため、オペレーティング・システムによって管理される異なるタイムスライス中にスレッドを占有します。オーバーサブスクリプションは、いくつかのシナリオではパフォーマンスを向上させることができますが、多くの計算集約型の並列アプリケーションでは、パフォーマンスは向上せずに OS スケジュールのオーバーヘッドが追加されます。

並列とは、実際に同時に起こることを意味します。2 つのタスクが同時期に同じポイントで実際にワークを実行する場合、並列であると考えることができます。同時（並行）実行と並列を区別する場合、重要なのはワークを同時に実行できるかどうかです。
単一のプロセッサー・コアしか利用できないため同時での実行が不可能である場合、数十年間はシングルコア・プロセッサーを多重化して、オペレーティング・システムがマルチタスクを同時実行することができました。

並列処理とは、一度に複数の事を行うことです。並列処理のタイプを分類する試みは多数あります。

並列化とは、同時にアクティビティーを行うこと可能にするコードへ変換する行為です。プログラムの並列化では、少なくともプログラムの一部を並列に実行します。

PetaFLOP/s = 1 秒あたり 10^15 回の倍精度（64 ビット）浮動小数点操作。

PetaFLOP = 10^15 回の倍精度（64 ビット）浮動小数点操作。

PSTL（並列標準テンプレート・ライブラリー）は、並列機能を備えた STL の実装を指します。

競合状態は、並列プログラムにおける非決定論的な動作であり、通常はプログラミングのエラーです。並行タスクが適切な同期なしに同じメモリー位置で複数の操作を実行し、そのうちの 1 つが書き込みの場合、競合状態が発生します。競合が発生したコードは、正しく動作することもあれば、動作に失敗することもあります。

再帰とは、関数のインスタンスが実行されているスレッドでアクティブである間に、関数を再度呼び出す動作です。最も簡単で一般的な方法は、関数が直接自分自身を呼び出しますが、再帰は複数の関数間でも適用できます。 再帰は、スタックなど動的に割り当てられたメモリー上に、部分的に完了した関数の状態を保持することによってサポートされます。高次の関数がサポートされる場合、複雑なメモリー割り当てスキームが必要になるかもしれません。再帰の回数に拘束される場合、過度なメモリーの利用を防ぐことが重要です。

スケーラビリティーとは、並列に使用できるハードウェアの増加に応じてパフォーマンスがどれだけ向上するかを表す指標です。

スケーラブル: 並列ハードウェアのリソースが追加された場合に、パフォーマンスが向上するアプリケーションは**スケーラブル**です。**強いスケーリング**という用語は、スケーリングを実現するために利用可能な計算能力が増えるため、問題のサイズを変更する必要がない場合のスケーリングを指します。**弱いスケーリング**とは、追加の計算が利用可能な場合に問題のサイズが拡大された場合にのみ発生するスケーリングを指します。**スケーラビリティー**を参照。

シリアルとは並行でも並列でもないことを意味します。

シリアル化は、潜在的に並列性を持つアルゴリズムのタスクが、リソース不足のため特定のシリアルな順番で実行されることです。並列化の反対です。

共有メモリー: 並列ワークの 2 つのユニットが同じ位置のデータをアクセスできること。通常、これは同期によって安全に行われます。並列ワーク、プロセス、スレッド、タスクのユニットは、物理メモリーシステムが許容する場合データを共有できます。しかし、プロセスはデフォルトではメモリーを共有できず、特殊なオペレーティング・システムの機能を使用する必要があります。

SIMD: 単一命令複数データは、1 つの命令で複数のデータ要素（配列の要素など）を同じ操作で処理する機能を指します。SIMD は、1966 年に最初に提案された、フリンの分類法として知られる分類システムで広く使用されたコンピューター・システムのアーキテクチャーです。

同時マルチスレッドとは、単一のプロセッサー・コア上でのマルチスレッドを指します。**ハイパースレッディング**を参照。

ソフトウェア・スレッドは、仮想ハードウェア・スレッドです。言い換えると、ハードウェア・スレッドに 1 対 1 でマッピングすることを意図するソフトウェアの単一の実行フローです。オペレーティング・システムは通常、必要に応じてハードウェア・スレッドにソフトウェア・スレッドをマップして、実際に存在するハードウェア・スレッドよりも多くのソフトウェア・スレッドの利用を可能にします。ハードウェア・スレッドよりもソフトウェア・スレッドが多い状態を**オーバーサブスクリプション**と呼びます。

空間の局所性:（メモリーアドレスの）距離を測定する際に近くにあること。時間の局所性と比較。空間の局所性は、プログラムが 1 つのデータ要素を使用した際に、近くのデータ（多くの場合、次のデータ要素）がすぐに使用される可能性が高いことを示すプログラムの動作を指します。優れた空間の局所性でデータを使用するアルゴリズムは、キャッシュラインとプリフェッチ・ハードウェア（両者とも現代のコンピューターで一般的なコンポーネント）の利点を活用できます。これらはどちらも現代のコンピューターで一般的なコンポーネントです。

スピードアップとは、単一の処理ユニットが問題を解決するレイテンシーと、複数の処理ユニットが並列に同じ問題を解決するレイテンシーの比率です。

SPMD: 単一プログラム複数データは、制限のある SIMD アーキテクチャーとは対照的に、同じプログラムで複数のデータ要素（配列の要素など）を処理する能力を示します。SPMD は多くの場合、分散メモリー・コンピューター・アーキテクチャー上でのメッセージ・パッシング・プログラミングについて言及します。SPMD は、1966 年に最初に提案された、フリンの分類法として知られる分類システムで広く使用された MIMD コンピューター・アーキテクチャーのサブカテゴリーです。

STL（標準テンプレート・ライブラリー）は、C++ 標準の一部です。

ストラングルド・スケーリングとは、高い頻度の競合やオーバーヘッドにより、並列コードのパフォーマンスが並列化されていない（シリアル）コードよりも低くなるプログラミング上のエラーを指します。

対称型マルチプロセッサー（SMP）は、共有メモリーと単一のオペレーティング・システムを実行するマルチプロセッサー・システムです。

SYCL は、Khronos グループによって管理される、異種コンピューティング向けのオープンなクロスプラットフォーム プログラミングの抽象化です。

同期: タスクやスレッドが、目的とする実行順序になるように調整すること。一般に、競合状態を排除するために使用されます。

タスク: 独自の制御フローを備えた潜在的な並列処理の軽量ユニット。通常、OS 管理のスレッドではなく、ユーザーレベルで実装されます。スレッドとは異なり、タスクは通常単一コア上でシリアル化され、完了するまで実行されます。スレッドは、タスクを並列に実行するメカニズムであり、タスクは、単に並列実行の*可能性*を示すワークのユニットです。タスクは、それ自身では並列実行のメカニズムとはなりません。

タスク並列処理: データよりもタスク指向の並列関係を試みること。

TBB: スレッディング・ビルディング・ブロック（TBB）を参照。

TCM: スレッド・コンポーザビリティー・マネージャー（TCM）を参照。

時間の局所性とは、時間の観点から測定する際に近くにあることを意味します。。空間の局所性と比較。時間の局所性は、データは比較的早く再利用されるプログラムの振る舞いを指します。優れた時間の局所性によりデータを使用するアルゴリズムは、データキャッシュ（現代のコンピューターで一般的）の利点を活用できます。時間と空間の局所性の両方を活用するのは珍しいことではありません。コンピューター・システムでは一般に、両方が達成されたときに最高のパフォーマンスを得られる可能性が高く、そのためアルゴリズムの設計で考慮されます。

スレッドは*ソフトウェア・スレッド*または*ハードウェア・スレッド*を指す場合があります。一般に、“ソフトウェア・スレッド” は、独立した制御フローと並列ワークのソフトウェアのユニットであり、“ハードウェア・スレッド” は、単一の制御フローを実行することができるハードウェア・ユニットです（特に、単一のプログラムカウンターを維持するハードウェア・ユニット）。

スレッド・コンポーザビリティー・マネージャー（TCM）は、CPU 上で実行するときに TBB、OpenMP、OpenCL、および SYCL 間のコンポーザビリティー（構成の容易性）を向上させる、oneAPI ツールキットに含まれる共有アービトレーターです。

スレッド並列処理とは、タスクごとに個別の制御フローを使用して、ハードウェアによる並列処理を実装するメカニズムです。

スレッド・ローカル・ストレージとは、少なくとも同時計算中は単一のスレッドからのみアクセスすることを目的として意図的に割り当てられたデータのことです。その目標は、アルゴリズムの最も集中した計算の瞬間に同期の必要性を回避することです。スレッド・ローカル・ストレージの典型的な例は、大規模な配列内のすべての数値を加算するときに部分和を作成する場合です。これは、ローカル/プライベートであるという性質上、合計するためにグローバル同期を必要としないローカル部分和（プライベート化された変数とも呼ばれます）に最初にサブ領域を並列に加算することによって行われます。

スレッディング・ビルディング・ブロック（TBB）は、C++ における並列プログラミング向けの最も人気のある抽象化ソリューションです。TBB はインテルによって開発されたオープンソース・プロジェクトであり、多くのオペレーティング・システムとベンダーのプロセッサーに移植されています。TBB は現在、UXL Foundation の管理下にあります。

スループットは、実行される一連のタスクが与えられた場合に、それらのタスクが完了する速度として定義されます。スループットは計算速度を測定し、単位時間あたりのタスク数を示します。**帯域幅**と**レイテンシー**を参照。

TFLOP/s = 1 秒あたり 10^12 回の倍精度（64 ビット）浮動小数点操作。

TFLOPs = 10^12 回の倍精度（64 ビット）浮動小数点操作。

タイリング（タイル化）とは、ループを適切な粒度の並列タスクのセットに分割することを指します。一般的に、タイル化は問題全体をステップに分けてシーケンシャル実行するのではなく、問題の小さな部分領域を複数のステップで実行することで構成されます。
タイル化の目的は、キャッシュ内のデータの再利用を高めることです。プログラム全体がキャッシュに収まりきらない場合、タイル化は劇的なパフォーマンス向上をもたらします。本書では、“ブロック化”の代わりに“タイル化”を、“ブロック”の代わりに“タイル”という用語を使用しています。タイル化とタイルは、近年一般的な用語として使用されるようになりました。

TLB は、トランスレーション・ルックアサイド・バッファーの略です。TLB は、仮想アドレスから物理ページアドレスへの変換情報を保持する特殊なキャッシュです。TLB 内のエントリー数は、どれくらいのメモリーページを同時に効率良くアクセスできるかを決定します。TLB に登録されていないページをアクセスすると、TLB ミスが発生します。TLB ミスは通常、ページテーブルを参照して TLB を更新するため、オペレーティング・システムに割り込みをかけます。

トリップカウントは、特定のループが実行される回数（トリップ）であり、反復回数とも呼ばれます。

Unified Acceleration（UXL）Foundation は、オープン・スタンダード・アクセラレーター・ソフトウェアのエコシステム開発を推進するプロジェクトです。TBB は UXL Foundation によって管理されています。

均一メモリーアクセス（UMA）: 共有メモリー・アーキテクチャーにおいて、メモリー設計の特性を分類するために使用されます。UMA = メモリー・アクセス・レイテンシーは、すべてのメモリーに対して同じです。NUMA = メモリー・アクセス・レイテンシーは、メモリーによって異なります。**NUMA** と比較してください。11 章を参照。

UXL: Unified Acceleration（UXL）Foundation を参照。

ベクトル操作は、SIMD 方式で複数のデータ要素に同時に作用する低レベルの操作で

す。

ベクトル並列性とは、複数のデータ要素に同じ制御フローを使用して、ハードウェアによる並列性を実装するメカニズムです。

ベクトル化は、ベクトル・ハードウェアを使用して同時に計算を行うこと可能にするコードへ変換する行為です。MMX、SSE、AVX、AVX2、AVX-512 命令などのマルチプロセッサー命令はベクトル・ハードウェアを利用しますが、CPU 外部のベクトル・ハードウェアは、ベクトル化の対象となる他の形式で提供される場合があります。コードのベクトル化により、命令ごとに処理するデータ量が増えるため、パフォーマンスが向上する傾向があります。**ベクトル化**を参照。

ベクトル化するとは、SIMD 命令 (MMX、SSE、AVX、AVX2 および AVX-512 など) などのベクトル・ハードウェアを活用して、プログラムのスカラー実装をベクトル化された実装に変換することを指します。ベクトル化は並列化の特殊な形式です。

vi は、Bill Joy によって開発された、ほとんどの UNIX および BSD システムに同梱されているテキストベースのエディターです。James によると、まだ emacs を発見していない人たちの間でのみ人気があります。もちろん、オープンソースです。emacs や Atom と比べると劣ります。

仮想メモリーは、実メモリーの物理アドレスからソフトウェアによって使用されるアドレスを分離します。仮想アドレスから物理アドレスの変換は、オペレーティング・システムにより初期化および制御されるハードウェアによって行われます。

ZettaFLOP/s = 1 秒あたり 10^21 回の倍精度 (64 ビット) 浮動小数点操作。

ZettaFLOP = 10^21 回の倍精度 (64 ビット) 浮動小数点操作。

ZettaOP/s = 1 秒あたり 10^21 回の数学操作。

ZettaOP = 10^21 回の数学操作。

索引

A

B

C

M. J. Voss and J. R. Reinders, *Today's TBB*, https://doi.org/10.1007/979-8-8688-1270-5

D

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V

W、X、Y、Z

訳者あとがき

インテル社のソフトウェア開発製品は、コンパイラーをはじめ、ライブラリー、解析ツールなど多岐にわたりますが、インテル社のマニュアル以外に解説書が書籍化されているものはそれほど多くはありません。その中で、インテル・スレッディング・ビルディング・ブロック（TBB）は、その進化とともに書籍が出版されている唯一のソフトウェア開発製品です。TBB の歴史と背景については、本書の「付録 A: 歴史とインスピレーション」で詳しく解説されています。

2007年にオリジナルのインテル・スレッディング・ビルディング・ブロックの書籍が米国でオライリーから出版されしばらくした頃、当時米国の同じ部署であった James Reinders から書籍を紹介され、日本国内での翻訳出版を検討しました。その頃私は、インテル・コンパイラーとインテル VTune アナライザーの日本語版開発に携わっており、忙しかったのですが、TBB の将来性に期待し、2008年2月にオライリージャパンから出版しました。手に取ってご覧になった方もおられるでしょう。あまり良いできではなかったのですが …

本書の付録にもありますが、James は 2019 年に 2 冊目の TBB 関連書籍 Pro TBB を執筆しましたが、この書籍は翻訳しませんでした。そして今年、本書 Today’s TBB が出版されました。本書は oneAPI が普及してからの書籍であり、TBB から oneTBB に移行する手引書としての必要性から翻訳することにしました。本書でも説明がありますが、従来の TBB から oneTBB へに移行には、それほど作業は必要ありませんが、安全で確実な移行のため TBB の利用者は本書を役立てていただければと思います。

本書の出版に前後して James は、インテル社をリタイアし楽しい生活を送っているようです。これまで、James は TBB の書籍以外にも Xeon Phi プロセッサー関連の書籍など数冊執筆していますが、そのほとんどを日本語版として皆さんに紹介していました。長年にわたり、テクノロジーのインスピレーションを与えてくれた James には、この場を借りて感謝します。

最後に、本書のプルーフリードに協力いただいた皆様に感謝いたします。

2025年 8 月
iSUS 編集部　すがわら