並列プログラミングの問題と解決

インテル® Cilk™ Plus によるソリューション

インテル株式会社
ソフトウェア&サービス統括部

April 22’ 2011

内容

- ソフトウェア開発のトレンド
  - ソフトウェア開発における 6 のトレンド
  - ソフトウェア全体に与える影響の比較
  - 並行プログラミングにおける 3 の原理
- 並列プログラミングにおける問題と
  インテル® Cilk™ Plus によるソリューション
  - 並列指向
  - 並列プログラミングにおける問題の分離
  - 決定性の問題

内容

- ソフトウェア開発のトレンド
  - ソフトウェア開発における 6 のトレンド
  - ソフトウェア全体に与える影響の比較
  - 並行プログラミングにおける 3 の原理

- 並列プログラミングにおける問題と
  インテル® Cilk™ Plus によるソリューション
  - 並列指向
  - 並列プログラミングにおける問題の分離
  - 決定性の問題

ソフトウェア開発における 6 つのトレンド

それぞれ 1958 年から 1973 年の間に誕生し、1990 年代/2000 年代に台頭し始めて、5 年強をかけて成熟したツール/言語/フレームワーク/ランタイム体系が構築された

GUI　オブジェクト GC　ジェネリック　Net　並列化

大きな相違点: 並行化時代の到来は察知できていた

1990 1992 1994 1996 1998 2000 2002 2004 2006 2008 2010 2012

ソフトウェア全体に与える影響の比較

	GUI	オブジェクト	GC	ジェネリック	Web	並行化
アプリケーション・プログラミング・モデル	●●●	●●●	●	●	●	●●●
ライブラリーとフレームワーク	●●●	●●●		●●	●●●	●●●
言語とコンパイラー	●●	●●●	●	●●		●●
ランタイムと OS	●●		●●		●	●●●
ツール (設計、測定、テスト)	●●●	●	●		●●	●●

影響の範囲と重要性
● = 多少。1 つの主要な製品リリース
●● = 重要。1 つ以上の製品リリース
●●● = 新しい考え方が必須。複数の主要リリース

並行プログラミングにおける 3 つの原理

	I. 非同期エージェントによる隔離	II. 並行収集によるスケーラビリティー	III. 安全な共有リソースによる一貫性
タグライン	ブロッキングの回避	フリーランチの復活	共有状態問題の回避
要約	個別にタスクを実行し、メッセージにより通信することで応答性を維持	複数のコアを使用し、グループ単位で処理を実行することでより素早く結果を取得。データ/アルゴリズムの構造を並列化	共有リソース（特に、共有メモリーの可変オブジェクト）へのアクセスを同期させることで競合を回避
例	GUI、Web サービス、バックグラウンド印刷/コンパイル	ツリー、クイックソード、コンパイル	メモリーにある可変共有オブジェクト、データベース・テーブル
主要な目的	応答性	スループット、スケーラビリティー	競合なし、デッドロックなし
条件	隔離、個別	低オーバーヘッド、副作用なし	結合可能、直列化可能
現在の抽象概念	スレッド、メッセージキュー	スレッドプール、OpenMP*	明示的ロック、ロックフリー・ライブラリー、トランザクション
可能な抽象概念	アクティブ・オブジェクト、Future	Chore、Future、Parallel STL、PLINQ、Cilk Plus	トランザクション・メモリー、改善されたロック、ArBB

内容

- ソフトウェア開発のトレンド
  - ソフトウェア開発における 6 のトレンド
  - ソフトウェア全体に与える影響の比較
  - 並行プログラミングにおける 3 の原理

- 並列プログラミングにおける問題と
  インテル® Cilk™ Plus によるソリューション
  - 並列指向
  - 並列プログラミングにおける問題の分離
  - 決定性の問題

並列指向

並列指向とは、そしてアルゴリズムにおける並列性とは?
多くの技術者はそれを知らないまま過ごしてきた

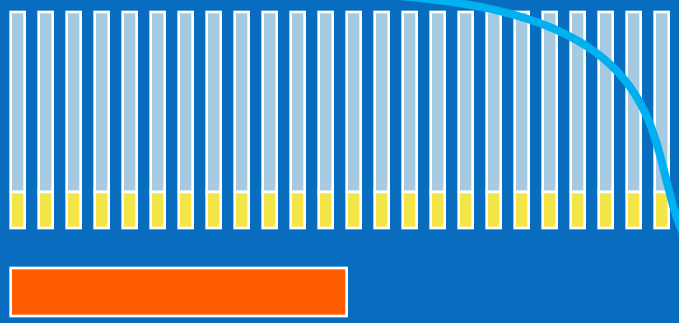
アプリケーションのアルゴリズムに並列性を実装するため、並列アルゴリズムのパターンを知らなければならない

高度な並列指向の例:
問題(処理)を細かく分割して並列に処理し、結果を結合する「分割統治」に基づくアルゴリズム。 多くの場合「分割統治」を再帰呼び出しで実装される

どのように並列アプリケーションを作成するか：

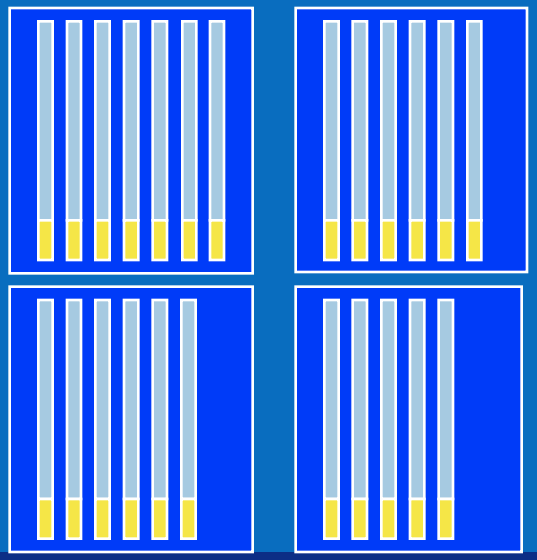
もとのシリアル処理

タスクへ分解する



タスクは共有データと
ローカルデータを持つ

実行ユニットへ
グループ化する



並列プログラム
環境をコード化

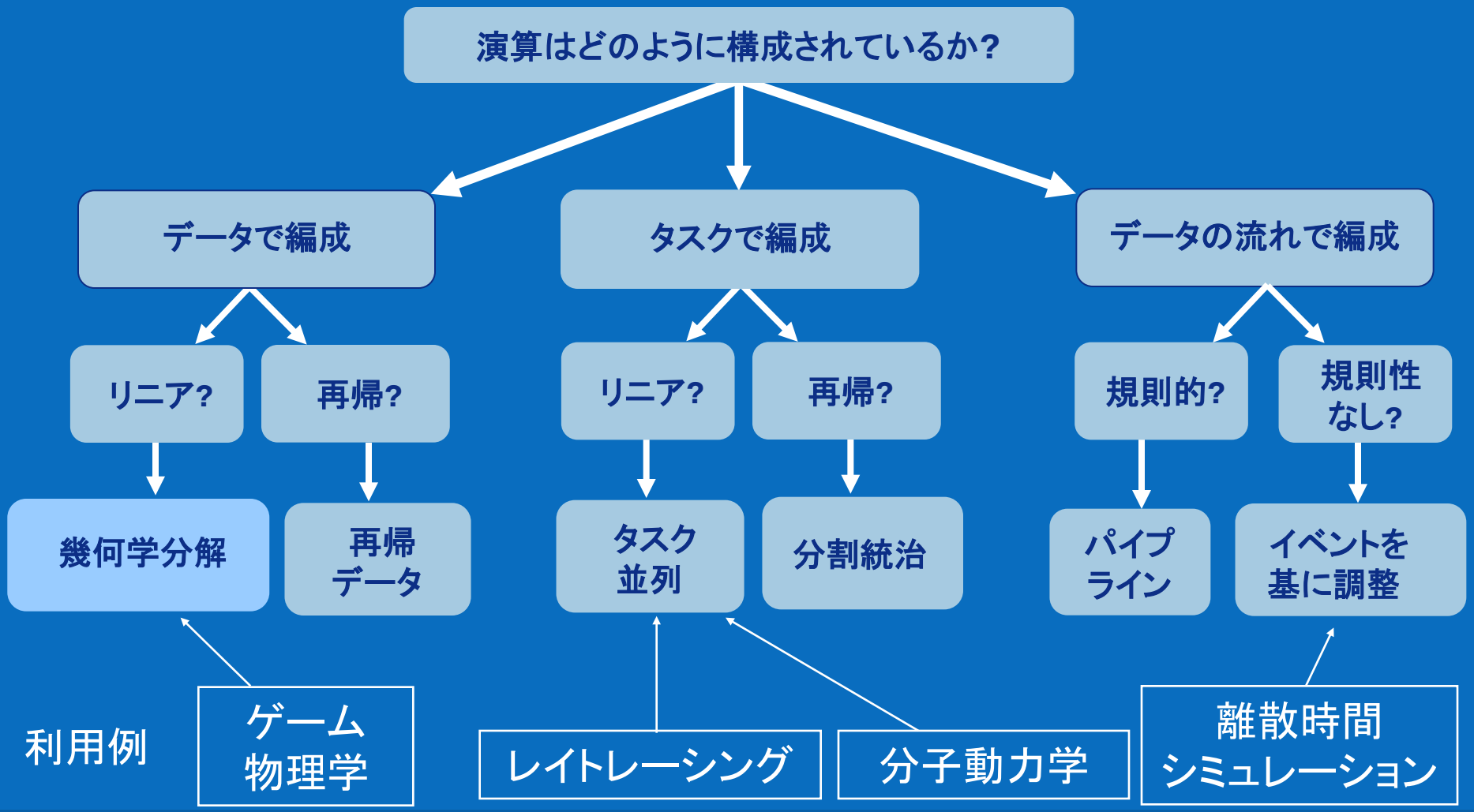
```
Program SPMD_Emb_Par ()
Program SPMD_Emb_Par ()
Program SPMD_Emb_Par ()
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
      tmp = func(I, Data);
      Res.accumulate( tmp);
  }
}
```

実行のユニット化 + 依存性を解決するための新たな共有データ

ソースコードに実装

インテル® ソフトウェア開発製品

主要な並列アルゴリズム

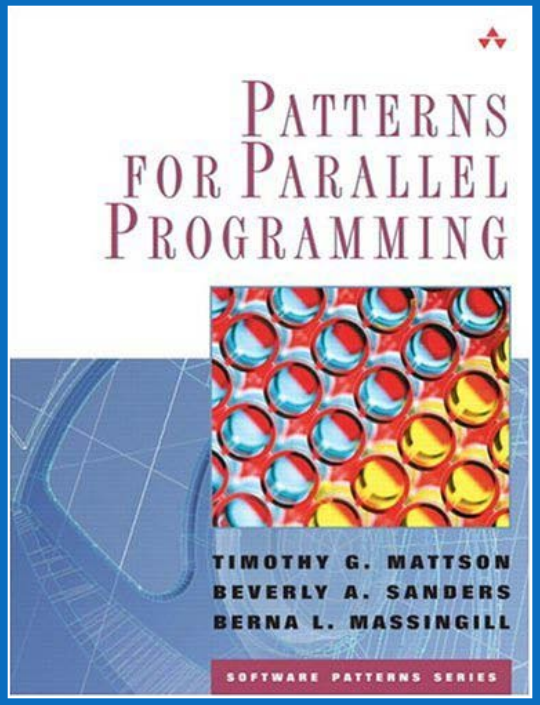


参考文献

並列アルゴリズム設計におけるパターン言語について解説

MPI、OpenMP そして Java による例題を提示

プログラマーがどのように並列プログラミングについて考えるのか、著者の仮説を述べている



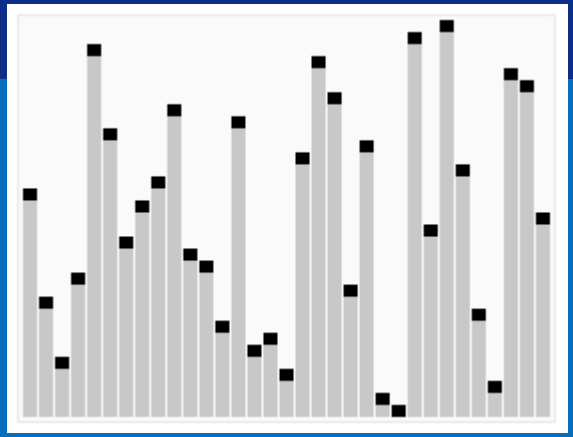
***Patterns for Parallel Programming*, Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, Addison-Wesley, 2005, ISBN 0321228111**

並列指向の備わったアルゴリズムの例

```
void qsort(int * begin, int * end) {
  if (begin != end) {
    --end;                                          // 最後にピボットを除く
    int * middle = std::partition(begin, end,       // パーティションテンプレートで
       std::bind2nd(std::less<int>(),*end));        // 分割する

    std::swap(*end,*middle);                        // ピボットを中間へ移動
    qsort(begin, middle);
    qsort(++middle, ++end);                         // ピボットを除く
    }
}
```

2 つの呼び出しは、異なるデータ領域を処理
つまり、同時に実行することができる

QuickSort のアルゴリズムに並列性が備わっていることを、誰も教えてくれなかった



高度な並列指向をサポートする OpenMP* とインテル® Cilk™ Plus

```
void qsort(int * begin, int * end) {
  if (begin != end) {
    --end;                                          // 最後にピボットを除く
    int * middle = std::partition(begin, end,       // パーティションテンプレートで
       std::bind2nd(std::less<int>(),*end));        // 分割する

    std::swap(*end,*middle);                        // ピボットを中間へ移動
    qsort(begin, middle);
    qsort(++middle, ++end);                         // ピボットを除く
    }
}
```

OpenMP*　Cilk™ Plus

このアルゴリズムを簡単に、効率よく並列化するには？
OpenMP* とインテル® Cilk™ Plus を比較してみる

01Sample_1¥sample1.cpp

高度な並列指向をサポートする OpenMP* では

```
void qsort(int * begin, int * end) {
   if (begin != end) {
         --end;                                          // 最後にピボットを除く
   int * middle = std::partition(begin, end,  // パーティションテンプレートで
      std::bind2nd(std::less<int>(),*end));  // 分割する
#pragma omp parallel
  {
#pragma omp single
    {
#pragma omp task
     qsort(begin, middle);
#pragma omp task
     qsort(++middle, ++end);
#pragma omp taskwait
    } // end single
  } // end parallel
   }  // end if
}
```

OpenMP では並列領域の定義の仕方によって大きく性能が異なる

01Sample_1¥sample1omp.cpp
01Sample_1¥sample1omp2.cpp

Software

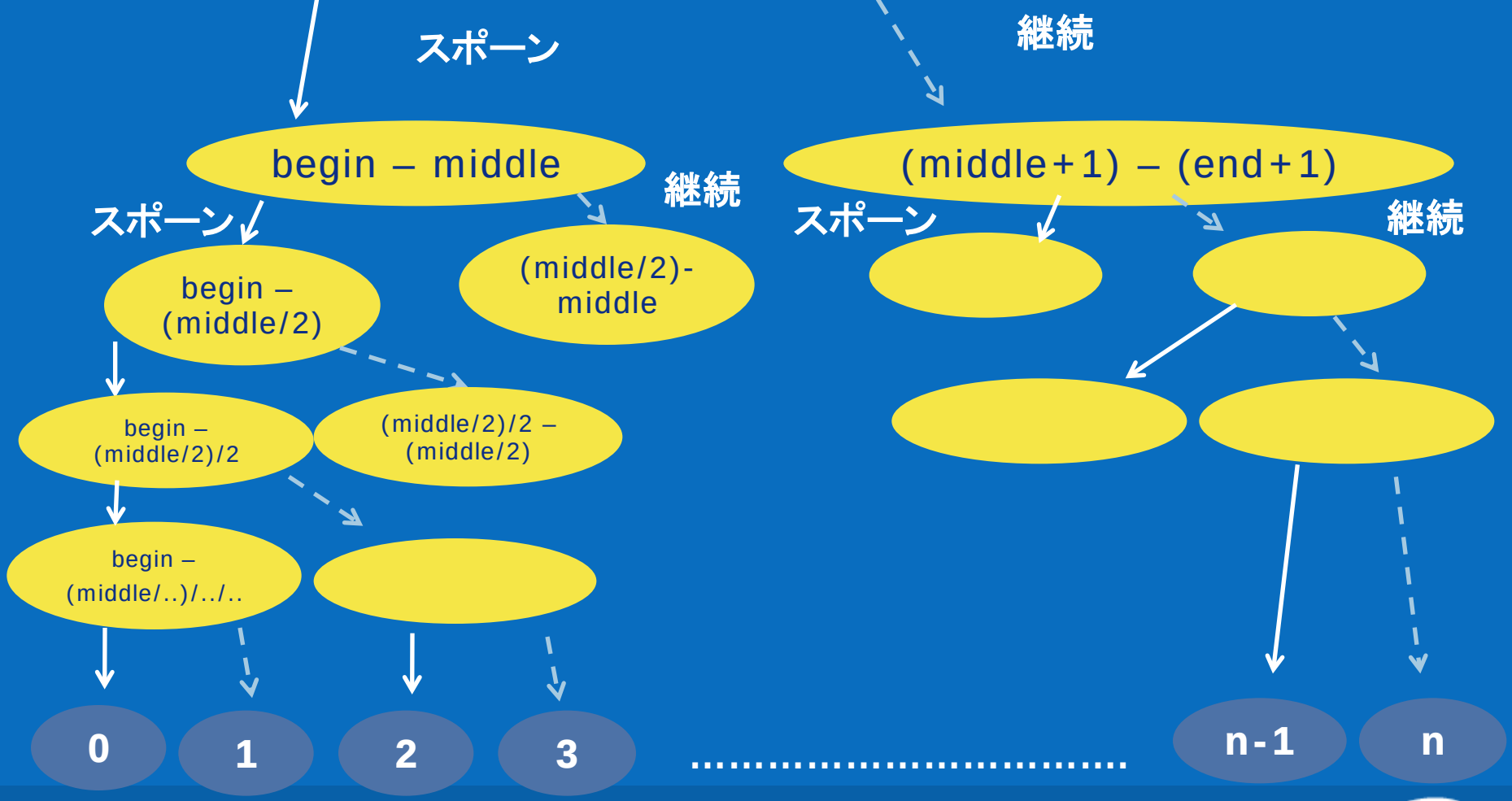
高度な並列指向をサポートする インテル® Cilk™ Plus

```
void qsort(int * begin, int * end) {
  if (begin != end) {
    --end;                                          // 最後にピボットを除く
    int * middle = std::partition(begin, end,       // パーティションテンプレートで
       std::bind2nd(std::less<int>(),*end));        // 分割する

    std::swap(*end,*middle);                        // ピボットを中間へ移動
    cilk_spawn qsort(begin, middle);
    qsort(++middle, ++end);                         // ピボットを除く
  }
}
```

このアルゴリズムを簡単に、効率よく並列化するには？
OpenMP* と比較してみる

01Sample_1¥sample1cilk.cpp

```
cilk_spawn qsort(begin, middle);
qsort(++middle, ++end);
```



インテル® ソフトウェア開発製品

高度な並列指向の例
レガシー処理を分割統治ベースのアルゴリズムに変更

```
for(int i = 0; i < ii; ++i)
    for (int k = 0; k < kk; ++k)
        for(int j = 0; j < jj; ++j)
            C[i * jj + j] += A[i * kk + k] * B[k * jj + j];
```

この例では、3 階層のループの入れ子の最外と最内のループが依存性なく並列化できる

for ループをデータ分割するのがベストな方法か？

02Matmul¥sample4[a,b].cpp

内容

- ソフトウェア開発のトレンド
  - ソフトウェア開発における 6 のトレンド
  - ソフトウェア全体に与える影響の比較
  - 並行プログラミングにおける 3 の原理

- 並列プログラミングにおける問題と
  インテル® Cilk™ Plus によるソリューション
  - 並列指向
  - 並列プログラミングにおける問題の分離
  - 決定性の問題

並列プログラミングにおける問題の分離

並列プログラミングを容易にするため、様々な抽象化技術が開発されてきた:

ネイティブ・スレッド

OpenMP*

インテル® Cilk™ Plus

アプリケーションを並列実行する上で、本来アプリケーション自身に関わる以外のことを開発者に求めてはいけない

並列プログラミングにおける問題の分離の例

```
static long num_steps = 100000;
double step;

void main ()  {
        int i;
        double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;

        for (i=1;i<= num_steps; i++){
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Win32 API によるネイティブ・スレッドの実装

```
#include <windows.h>
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg){
   int i, start;
   step = 1.0/(double) num_steps;

   for (i=start;i<= num_steps;
   EnterCriticalSection(&hUpdateMutex);
   global_sum += sum;
   LeaveCriticalSection(&hUpdateMutex);
}
```

マルチ・スレッドのサポートを設定

各スレッドに分割する作業を定義し、それらを関数に切り分ける

演算結果を1つのスレッドが安全に生成できるように同期

```
void main (){
   double pi; int i;
   DWORD
   int thre

   for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

   InitializeCriticalSection(&hUpdateMutex);

   for (i=0; i<NUM_THREADS; i++){
            thread_handles[i] = CreateThread(0, 0,
            (LPTHREAD_START_ROUTINE) Pi,
                     &threadArg[i], 0, &threadID);
   }

   W
                                                  E);

   pi = global_sum * step;

   printf(" pi is %f ¥n",pi);
}
```

スレッドを生成する準備と生成

すべてのスレッドの完了を待つ

答えを表示

OpenMP* による PI のプログラム:
リダクションによる並列処理

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{       int i;   double x, pi, sum = 0.0;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=1;i<= num_steps; i++){
                x = (i-0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

これで十分ですか？

並列プログラミングにおける問題の分離

素数を求めるサンプルコード:

```
void FindPrimes(int start, int end){

#pragma omp parallel for reduction(+:gPrimes))
   for( int i = start; i <= end; i+=2 ){
      if( TestForPrime(i) )
                  gPrimes++;
   }
}
```

このループはどのように並列処理されるか?

- どのように負荷は分散されるか?
- スレッドはどのコアで実行されるか?

OpenMP ではプロセッサーコアへのスレッドの割り当ては OS 依存

並列プログラミングにおける問題の分離を解決する インテル® Cilk™ Plus と OpenMP*

```
void FindPrimes(int start, int end){
#pragma omp parallel for reduction(+:gPrimes))
    for( int i = start; i <= end; i+=2 ){
        if( TestForPrime(i) )
                    gPrimes++;
    }
}
```

```
void FindPrimes(int start, int end){
    cilk::reducer_opadd<int> Primes;
    Primes.set_value(gPrimes);
    cilk_for( int i = start; i <= end; i+=2 ){
        if( TestForPrime(i) )
                    gPrimes++;
    }
    gPrimes = Primes.get_value();
}
```

この2つの違いは?

並列プログラミングにおける問題の分離を解決する OpenMP* の実装

```
void FindPrimes(int start, int end){
#pragma omp parallel for reduction(
   for( int i = start; i <= end; i+=2
      if( TestForPrime(i) )
                    gPrimes++;
   }
}          end =10000000
```

```
bool TestForPrime(int val){
   int limit, factor = 3;

   limit = (long)(sqrtf((float)val)+0.5f);
   while( (factor <= limit) && (val % factor))
      factor ++;

   return (factor > limit);
}
```

4 コア･システムの場合

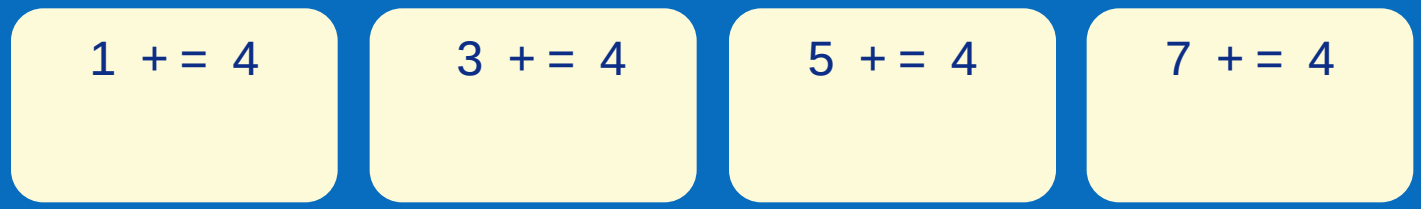
1 – 2500000	2500001 – 5000000	5000001 – 7500000	7500001 – 10000000

03Sample_2¥sample2b.cpp

並列プログラミングにおける問題の分離を解決する OpenMP* での解決 ....

```
void FindPrimes(int start, int end){
#pragma omp parallel for reduction(+:gPrimes)) schedule(static, 1)
   for( int i = start; i <= end; i+=2 ){
      if( TestForPrime(i) )
                  gPrimes++;
   }
}          end =10000000
```

4 コア･システムの場合



03Sample_2¥sample2b.cpp

並列プログラミングにおける問題の分離を解決する インテル® Cilk™ Plus の実装と解決

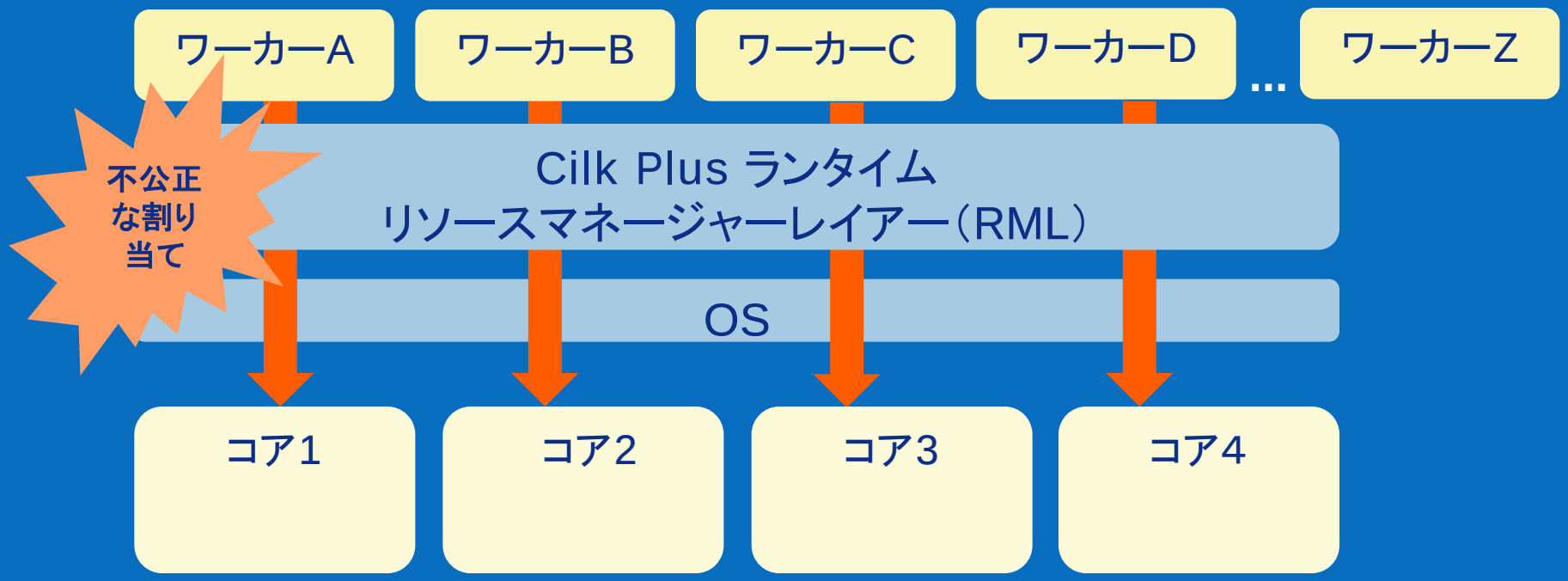
```
void FindPrimes(int start, int end){
    cilk::reducer_opadd<int> Primes;
    Primes.set_value(gPrimes);
    cilk_for( int i = start; i <= end; i+=2 ){
        if( TestForPrime(i) )
                    gPrimes++;
    }
    gPrimes = Primes.get_value();
}
```

Cilk Plus ランタイムが最小粒度を判断し、粒度に達するまで分割を繰り返す
ランタイムは分割されたタスクをワーカーへ割り当てる
ワーカーはランタイムによってプロセッサーコアに配置される

03Sample_2¥sample2c.cpp

インテル® Cilk™ Plus ランタイム・システム

インテルコンパイラーは並列化技術が効果的にプロセッサーのリソース（キャッシュやメモリー帯域）を利用できるよう、リソースマネージャーレイアー（RML）を実装している



内容

- ソフトウェア開発のトレンド
  - ソフトウェア開発における 6 のトレンド
  - ソフトウェア全体に与える影響の比較
  - 並行プログラミングにおける 3 の原理

- 並列プログラミングにおける問題と
  インテル® Cilk™ Plus によるソリューション
  - 並列指向
  - 並列プログラミングにおける問題の分離
  - 決定性の問題

決定性の問題

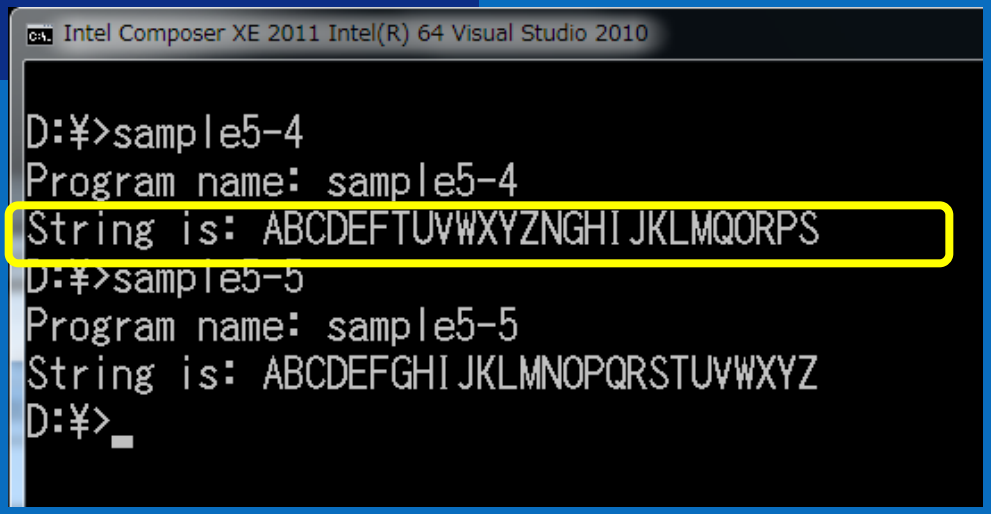
並列に実行されることで結果が変わらないか?

```
int result += input[i];

int result *= input[i];

std::list<char> result = 'A' + i;
```

その処理は可換可能か?

決定性の問題の例

```
std::list<char> result;
#pragma omp parallel for
for (std::size_t i = 'A'; i < 'Z'+1; ++i){
        lock();
        result.push_back((char)i);
        unlock();
}
```



このコードを実行すると、
シリアルプログラムと結果が異なる

決定性の問題を解決する
インテル® Cilk™ Plus

Mutex などのロックは決定性を損ねる

```
cilk::reducer_list_append<char> result;

cilk_for (std::size_t i = 'A'; i < 'Z'+1; ++i){
        result.push_back((char)i);
}
```

reducer_list_append で定義された result には、並列操作における決定性が保証される

04Sample_3¥sample3[a-c].cpp

まとめ

並列指向

並列プログラミングにおける問題の分離

決定性の問題

バックアップ

インテル® Cilk™ Plus 概要

インテル® Cilk™ Plus 概要

インテル® Cilk™ Plus は、共有メモリー・マルチコア・システム向けの C/C++ ベース言語拡張

- Cilk™ Plus は、3 のキーワードを定義する
  - cilk_spawn は、関数呼び出しを並列実行することを許可する
  - cilk_sync は、スポーンされたすべての子関数が終了するまで指定された位置で待機することを指示する
  - cilk_for は、ループ反復を並列に実行することを許可する

- Cilk™ Plus のハイパーオブジェクトは、結合可能な操作を行う場合に一般的なリダクションやユーザー定義リダクションを可能にする
- Cilk™ Plus は、ベクトル化を容易にするため並列配列表記（array notation ）を提供する
- Cilk™ Plus は、要素関数によるベクトル化を提供する
- Cilk™ Plus のタスク・スケジューラーは、自身の作業デックが空の場合、他のワーカーの作業をスチールすることを許す
- Cilk™ Plus では、#pragma simd を利用してユーザーの強制によるループのベクトル化が可能

フィボナッチ数列の例

- フィボナッチ数列は、<0,1,1,2,3,5,8,13,21,34,...> のように、自身の2つ前の数を足した値になるような数列
- 再起的な関係:

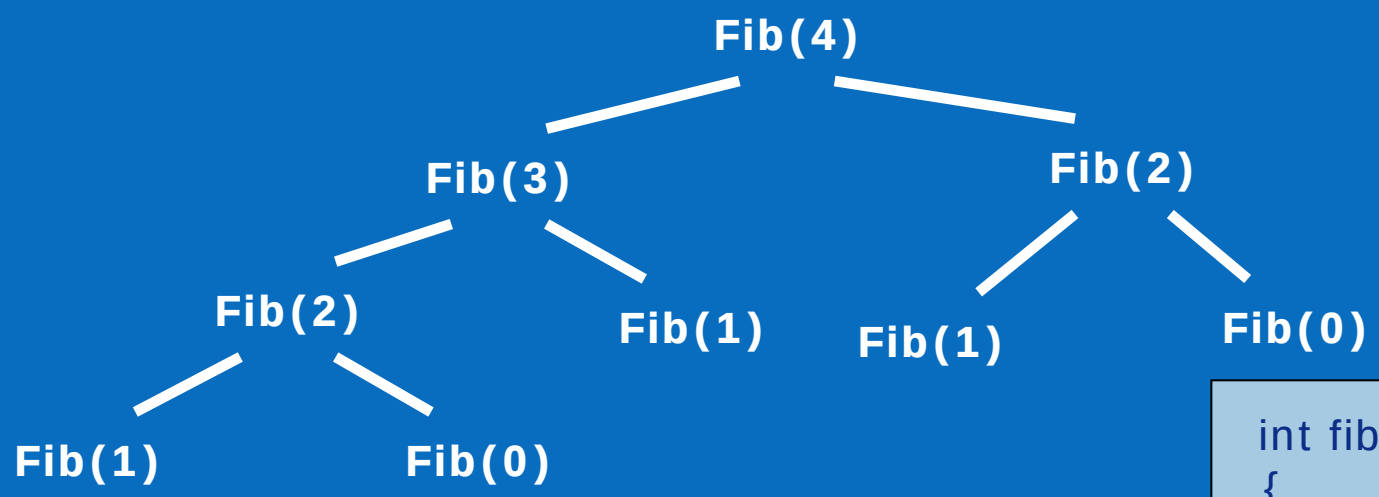
  F0 = 0,

  F1 = 1,

  Fn = Fn-1 + Fn-2 for n > 1
- 数列は、Leonardo di Pisa (1170-1250 AD.) にちなんで命名された。フィボナッチ filius Bonaccii の構造 -「Bonnaccio の息子」として知られている。それ以前にインドの数学者によって発見されていたが、フィボナッチの 1202 の書籍の *Liber Abaci* は、西洋の数学者に数列を紹介した

注意:この再帰プログラムは、n 番目のフィボナッチ数列を計算する最適な方法ではないが、これは良い教訓となる

フィボナッチの実行




```
int fib(int n)
{
  if (n<2) return n;
  else {
    int x = fib(n-1);
    int y = fib(n-2);
    return x+y;
  }
}
```

並列化のための考察

fib(n-1) と fib(n-2) の計算は、干渉なしに同時に実行することができる

シリアル版のフィボナッチ

```
int fib(int n)
{
  if (n<2) return n;
  else {
    int x = fib(n-1);
    int y = fib(n-2);
    return x+y;
  }
}
```

ネスト並列版のフィボナッチ

```
int fib(int n)
{
  if (n<2) return n;
  else {
    int x = cilk_spawn fib(n-
    int y = fib(n-2);
     cilk_sync;
    return x+y;
  }
}
```

スポーンされる子関数は、呼び出した親と並列に実行される

この同期ポイント以前にスポーンされたすべての子関数が終了するまで、このポイントで待機する

Cilk™ Plus キーワードは、並列実行の許可を与えるもので、並列実行を強制するものではない

Cilk_for によるマトリクス演算の例

ここから

```
...
for(unsigned int i = 0; i < n; ++i) {

  int itn = i * n;
  for (unsigned int k = 0; k < n; ++k) {
    for (unsigned int j = 0; j < n; ++j) {
      int ktn = k * n;
      A[itn + j] += B[itn + k] * C[ktn + j];
    }
  }
}
```

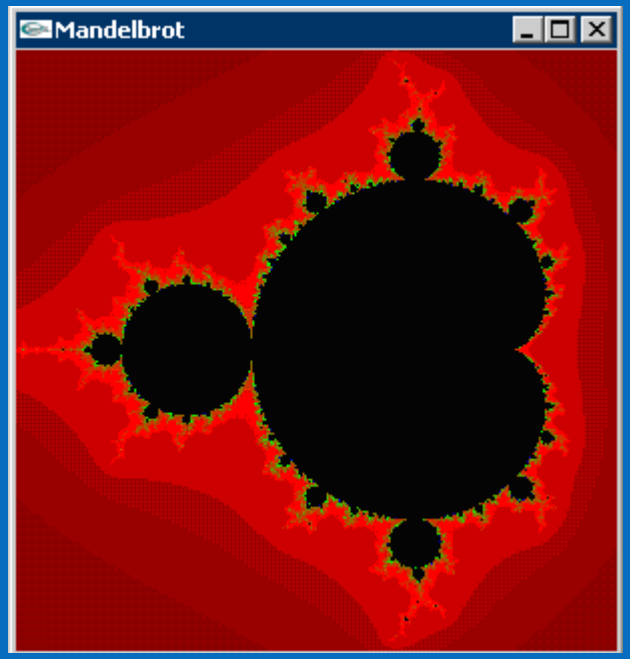
Cilk_for によるマトリクス演算の例

```
#include <cilk/cilk.h>
...
cilk_for (unsigned int i = 0; i < n; ++i) {
  // このプログラムで利用される唯一の Cilk™ Plus キーワード
  int itn = i * n;
  for (unsigned int k = 0; k < n; ++k) {
    for (unsigned int j = 0; j < n; ++j) {
      int ktn = k * n;
      A[itn + j] += B[itn + k] * C[ktn + j];
    }
  }
}
```

ここへ

実験 – マンデルブロー集合の並列版

目的: 並列版のマンデルブローを作成する。ソースに Cilk™ Plus の cilk_for キーワードを追加し、並列にマンデルブロー集合の計算を行う

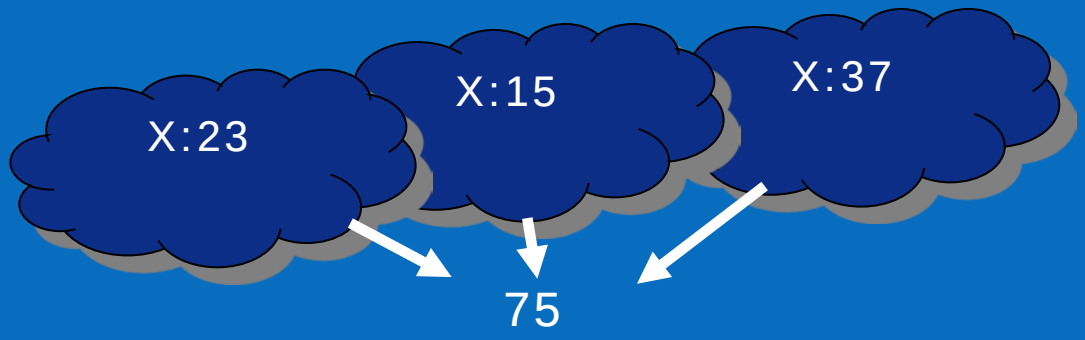


レデューサー・ハイパーオブジェクト

レデューサー・ハイパーオブジェクト

- 変数 x は、加算、乗算、論理AND、リストの連結のような結合操作が可能なレデューサーとして定義できる
- ストランドは、x を通常の非局所変数のようにアップデートできるが、実際は x 異なったビューの収集として保持される
- Cilk™ Plus ランタイム・システムは、ビューを操作し必要とされる時にそれらを結合する
- x の 1 のビューだけが残っていれば、保持される値は安定しており、引用できる

総和
レデューサー
の例



Cilk™ Plus レデューサーの例

Parallel Studio のサンプルにある、Nqueens の場合

```
cilk::reducer_opadd<unsigned int>
    total;
    ...
    total.set_value(g_nSolutions);
    cilk_for (int i=0; i<g_nsize; i++) {
                ...
        nqSetQueen(pNQ, 0, i);
                ...
    }
    g_nSolutions = total.get_value();
```

総和を行うレデューサーオブジェクトを total として宣言する

g_nSolutions 変数の値をレデューサー total の初期値として設定する

各 cilk_for セクションの終端には、暗黙の同期がある

各ストランドが持つローカルな total の値を結合して、g_nSolutions に読み出す

Cilk™ Plus 配列表記

インテル® Cilk™ Plus は、プログラマーがアルゴリズムのループ中で配列要素をインデックスで操作する代わりに、配列を簡単に操作する言語構造を提供する

コンパイラーは、並列を操作するため SIMD 命令を利用したベクトル・コードを生成する

- 例: 配列表記なしの操作

```
for (int i = 0; i < length; ++i) {
  res[i] = elemental_func(a[i],b[i]);
}
```

- 例: 配列表記を利用すると、ループを排除できる

```
res[:] = elemental_func(a[:] , b[:]);
```

括弧中の数字の意味は?

- r[0:N] = src1[0:N] + src2[0:N];
- [0:N] は、要素 0 から始まり、N 個の要素を操作することを意味する
- [0:N:2]は、0 から N 個の要素を、2 ずつスキップする

Cilk™ Plus 要素関数

要素関数は、スカラー引数や配列要素で並列に呼び出すことができる正規関数

コンパイラーにショートベクトル・バージョンの関数を生成することを指示するには、
__declspec (vector) アノテーションを利用する

- 関数宣言に vector 宣言子を追加する
- __declspec(vector) float elemental_func(float, float);

例: マルチコアとベクトル化に対応する

```
cilk_for (i = 0; i < length; ++i) {
        res[i] = elemental_func(a[i],b[i]);
}
```

例: 配列表記を利用してベクトル化する

```
res[:] = elemental_func(a[:] , b[:]);
```

戻る

OpenMP* の概要

OpenMP* とは?

- 移植性の高い、共有メモリー型スレッド API
  - Fortran, C, and C++
  - 複数のコンパイラーベンダーが、Linux と Windows をサポート
- 標準化されたタスクとループ・レベルの並列性
- 粗い粒度の並列性をサポート
- 1 のソースコードで、シリアル版と並列版に対応
- 20 年以上にわたる、コンパイラーによるスレッド化の経験

http://www.openmp.org

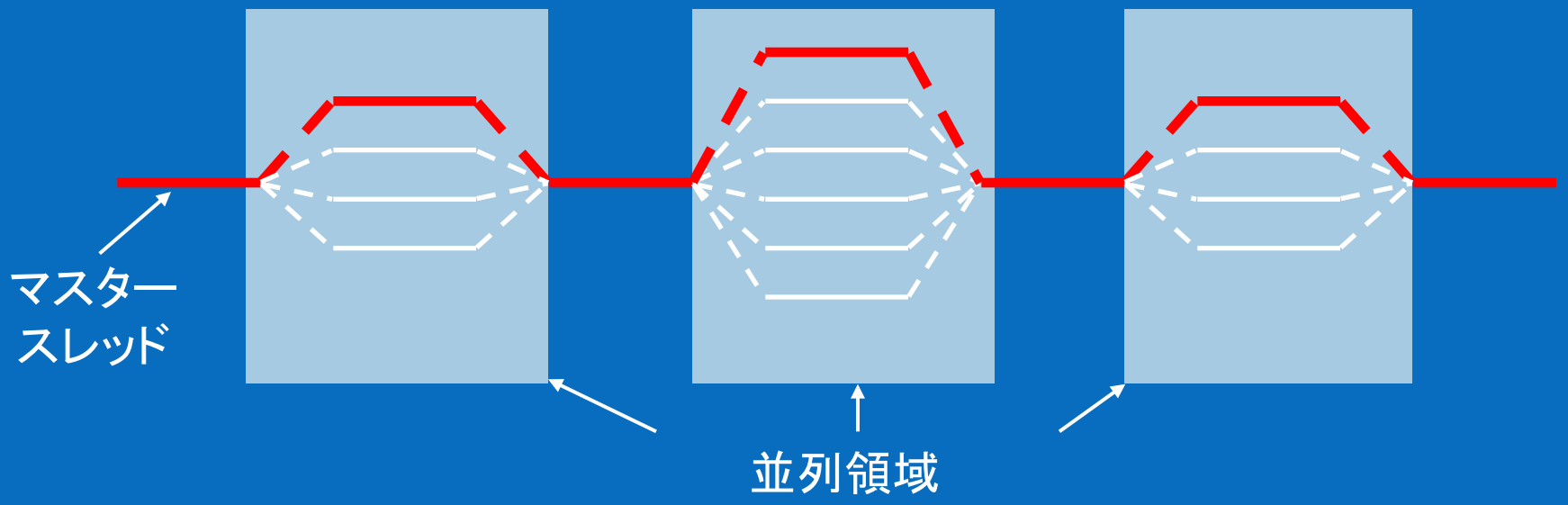
現在の仕様は、OpenMP 3.0（C/C++ と Fortran の仕様を統合）

OpenMP 3.1 仕様を策定中

OpenMP* プログラミング・モデル

Fork-Join 型並列:

- マスタースレッド が、指示された場所でスレッドのチームを構成
- 並列性はインクリメンタルに追加される: シリアル・プログラムが、並列プログラムに発展する



開始するにあたっていくつかの構文の詳細

OpenMP* における大部分の構造は、
コンパイラー指示文かプラグマ

C と C++ では、プラグマで始まる:

#pragma omp 指示句 *[指示節 [指示節]...]*

ヘッダーファイル

#include “omp.h”

OpenMP* 並列領域と構造化ブロック（C/C++）

ほとんどの OpenMP 構文は、構造化ブロックで適用される

- 構造化ブロック: 入口が 1 で、出口が 1 の実行ブロック
- C/C++ で許される分岐の例外は、exit() 文のみ

構造化ブロック

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

  if (conv (res[id]) goto more;
}
printf (“All done¥n”);
```

非構造化ブロック

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
printf (“All done¥n”);
```

OpenMP* ワークシェア

ワークシェアは、スレッドに処理を分散する OpenMP で利用される一般的な技術

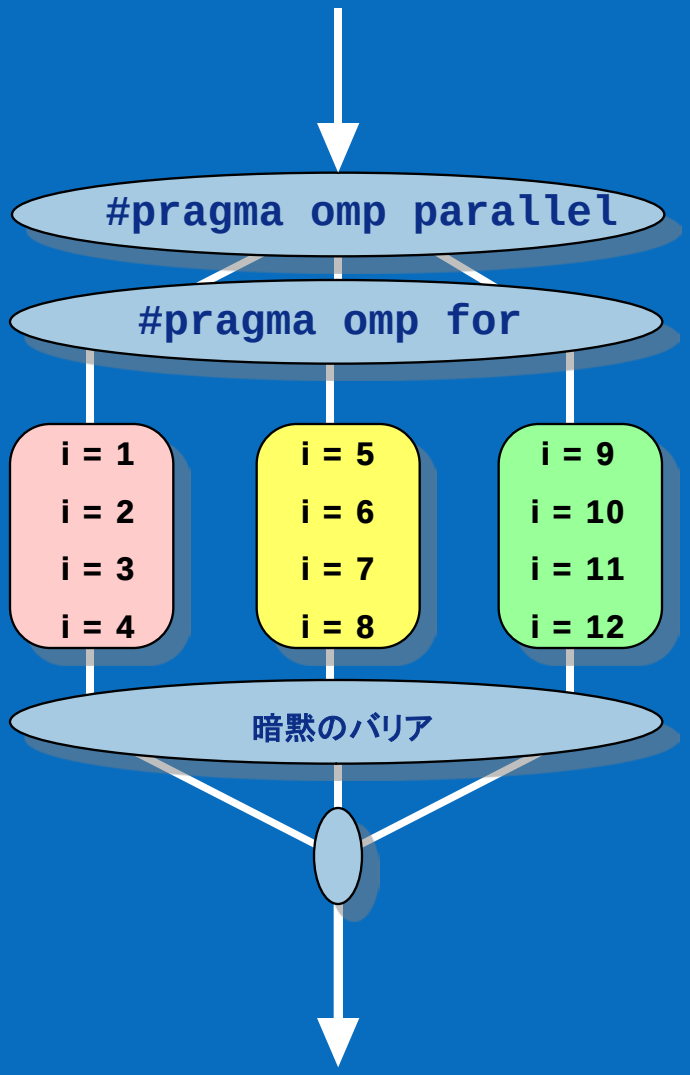
OpenMP における 3 のワークシェアの例:

- omp for 指示句
- omp section 指示句
- omp task 指示句
- omp single 指示句

作業を自動的にスレッドに分割する

omp for 指示句

```
// N=12 と想定
#pragma omp parallel
#pragma omp for
   for(i = 1, i < N+1, i++)
      c[i] = a[i] + b[i];
```



スレッドには独立したループが割り当てられる

スレッドはワークシェア領域の終わりで待機する

指示句の組み合わせ

下記の 2 の記述は等価

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
```

Private 指示句

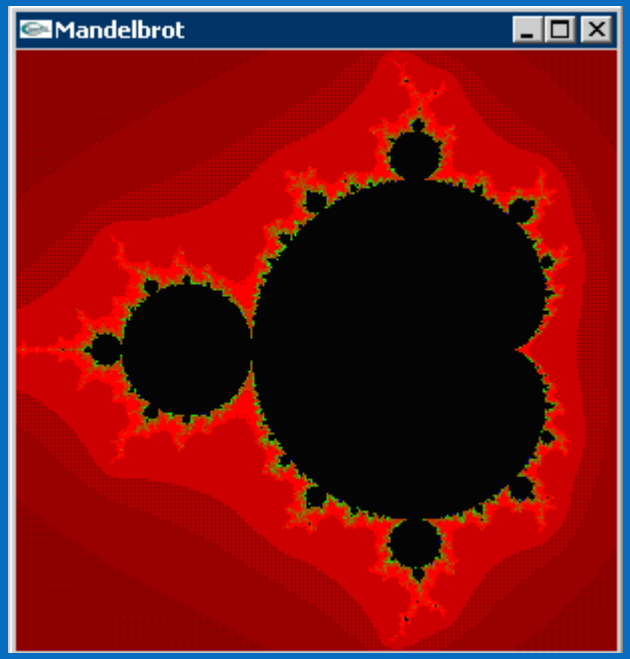
各タスクに変数を複製する

- 変数は初期化されない; C++ オブジェクトはデフォルトのコンストラクタ
- 並列領域外の変数はすべて未定義

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
       x = a[i]; y = b[i];
       c[i] = x + y;
     }
}
```

実験 – マンデルブロー集合の並列版

目的: 並列版のマンデルブローを作成する。ソースに OpenMP ワークシェア指示句を追加し、並列にマンデルブロー集合の計算を行う



スケジュール指示節

スケジュール指示節 は、ループ反復がどのようにスレッドに割り当てられるかに影響する

schedule(static [,chunk])

- “chunk” で指定される回数のループ反復ブロックをスレッドに割り当て
- ラウンドロビン形式の分散
- 低いオーバーヘッドのため、ロードインバランスになり易い

schedule(dynamic[,chunk])

- スレッドが “chunk” サイズの反復を要求
- スレッドが反復を処理し終えると、次を要求
- 高いオーバーヘッド、ロードインバランスを軽減

schedule(guided[,chunk])

- 開始時は大きなブロックをダイナミックにスケジュール
- 終わりに近づくとブロックサイズを小さくする; “chunk” より大きい

スケジュール節の例

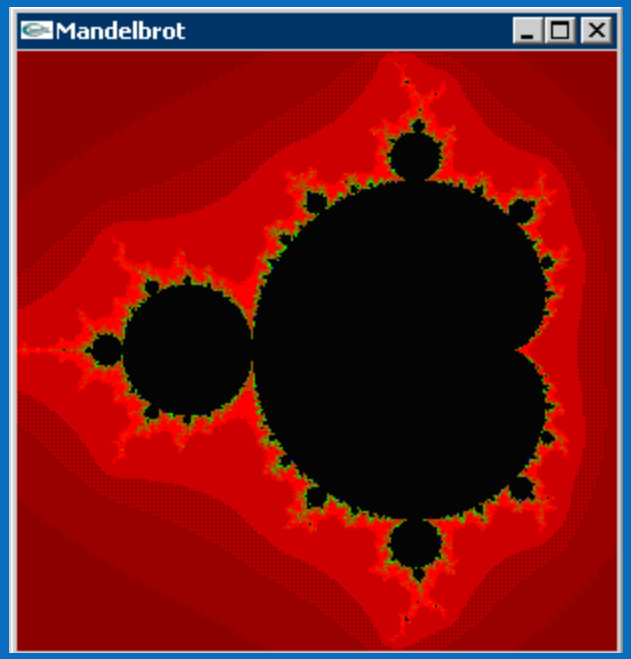
```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
        if ( TestForPrime(i)) gPrimesFound++;
    }
```

反復は 8 回のチャンクに分割される

start = 3 なら、最初のチャンクは、
i={3,5,7,9,11,13,15,17}　　// 8回の反復

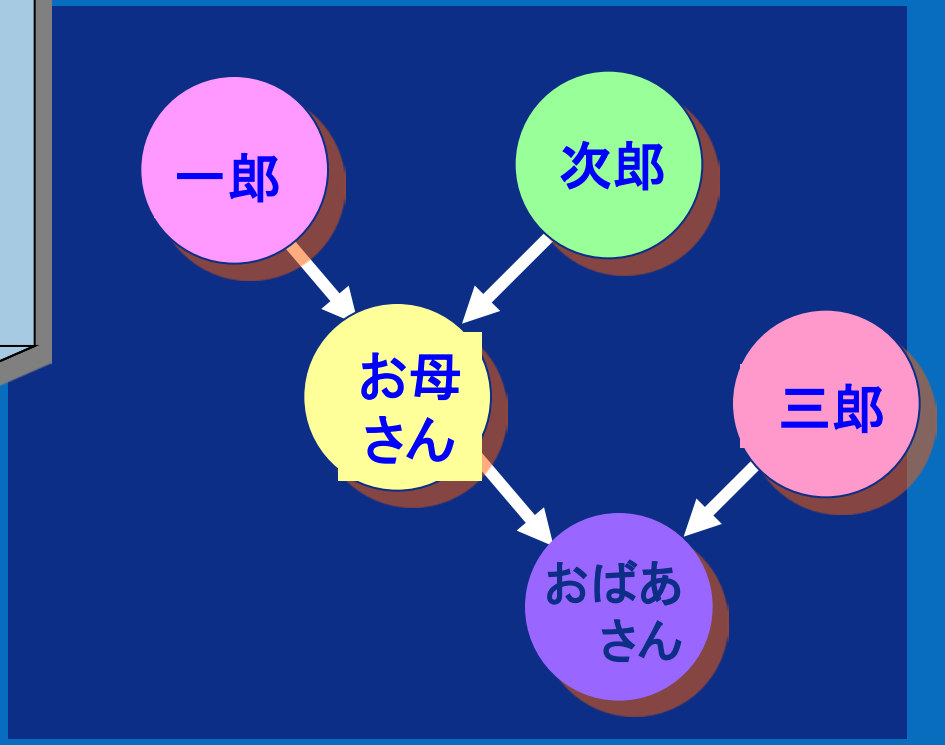
実験 –マンデルブロー集合のスケジューリング

目的: 並列版のマンデルブロー集合を作成し、OpenMP のスケジュール指示節でダイナミックにスケジュールする



タスク分解

```
a = 一郎();
b = 次郎();
s = お母さん(a, b);
c = 次郎();
printf (“%6.2f¥n”,
    おばあさん(s,c));
```



一郎、次郎、そして三郎は同時に演算できる

タスク

不規則な処理の並列化を可能にする

- 限りないループ
- 再帰アルゴリズム
- 生産と消費アルゴリズム

タスクとは?

タスクは作業の独立した単位

スレッドには、各タスクの作業が割り当てられる

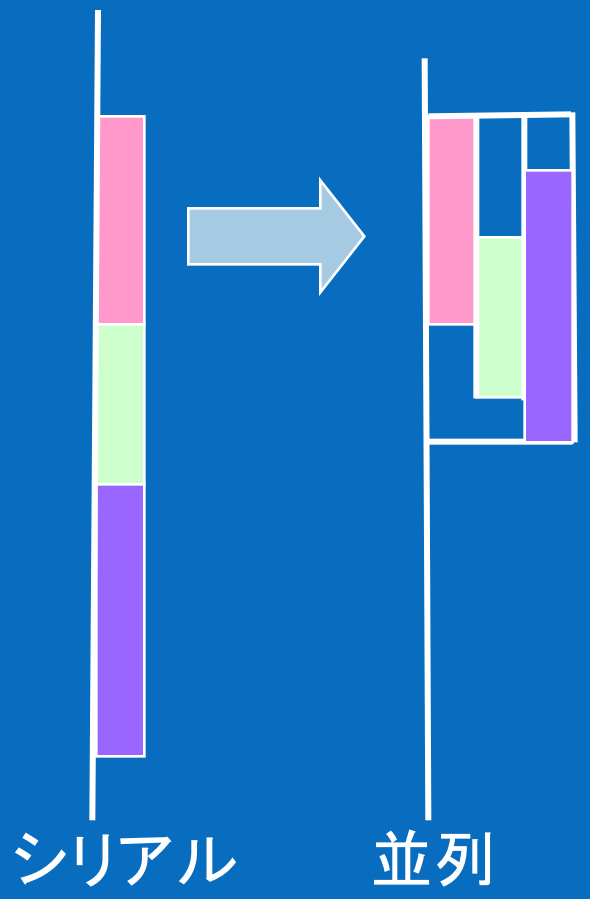
タスクは据え置かれる

タスクは即座に実行、もしくは延期される

ランタイム・システムは、以下のいずれかを行うか決定する

タスクは以下で構成される:

- 実行するコード
- data 環境
- 内部制御変数(ICV)



簡単なタスクの例

```
#pragma omp parallel
// 8 スレッドと過程
{
  #pragma omp single private(p)
  {
    ...
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

8 スレッドのプールが
ここで作成される

1 つのスレッドのみが
while ループを実行

while ループ実行される
たびに、ここで processwork()
がタスクとして生成される

タスク構造 – 明示的なタスク

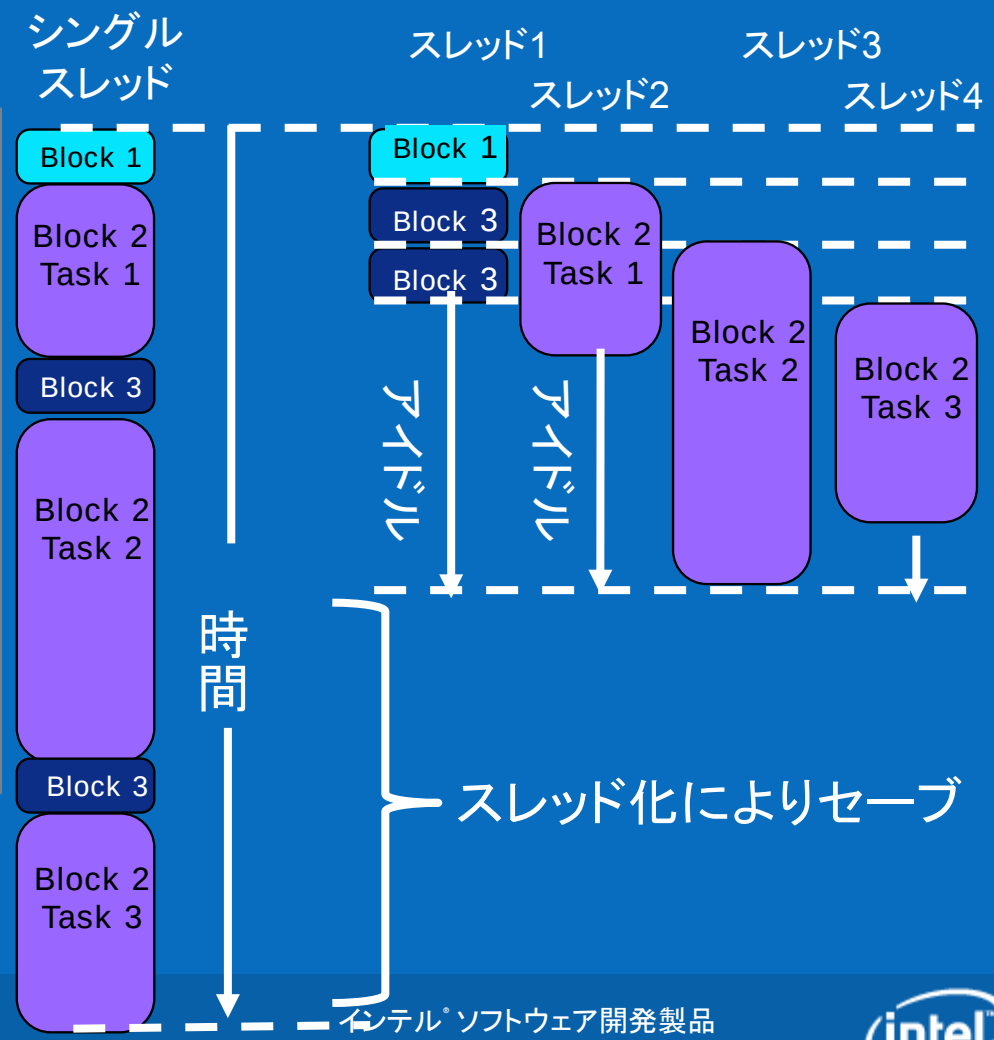
- omp parallel 指示句でスレッドのチームが生成される
- While ループを実行するため single 指示句を指定する。これをスレッド “L” と呼ぶことにする
- スレッド “L” は、while ループ内でタスクを生成し、次のポインターを割り当てる
- スレッド “L” は、新たにスレッドに割り当てるタスクを生成するため omp task 指示句を実行する
- 各タスクは、それぞれのスレッドで実行される
- Single 領域の終わりには暗黙のバリアがあり、すべてのタスクはバリアで実行を完了する

```
#pragma omp parallel
{
  #pragma omp single
  { // ブロック 1
    node * p = head;
    while (p) { // ブロック 2
    #pragma omp task private(p)
      process(p);
    p = p->next; // ブロック 3
    }
  }
}
```

なぜタスクが有効なのか?

不規則なパターンと再帰関数呼び出しを並列化できる可能性がある

```
#pragma omp parallel
{
  #pragma omp single
  { // ブロック1
    node * p = head;
    while (p) { // ブロック2
    #pragma omp task
      process(p);
    p = p->next; // ブロック3
    }
  }
}
```



タスクの完了が保障されるのはいつか?

- タスクの完了は次の条件で保障される:
- スレッドもしくはタスクのバリア
  - `#pragma omp barrier` が指定される位置
  - `#pragma omp taskwait` が指定される位置

タスク完了の例

```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

foo 関数がタスクとして生成される – 各スレッド分のタスクを生成

すべての foo タスクは、ここで完了する

bar タスクがここで生成される

bar タスクはこの位置で完了することが保障される

戻る

intel®

Leap ahead™