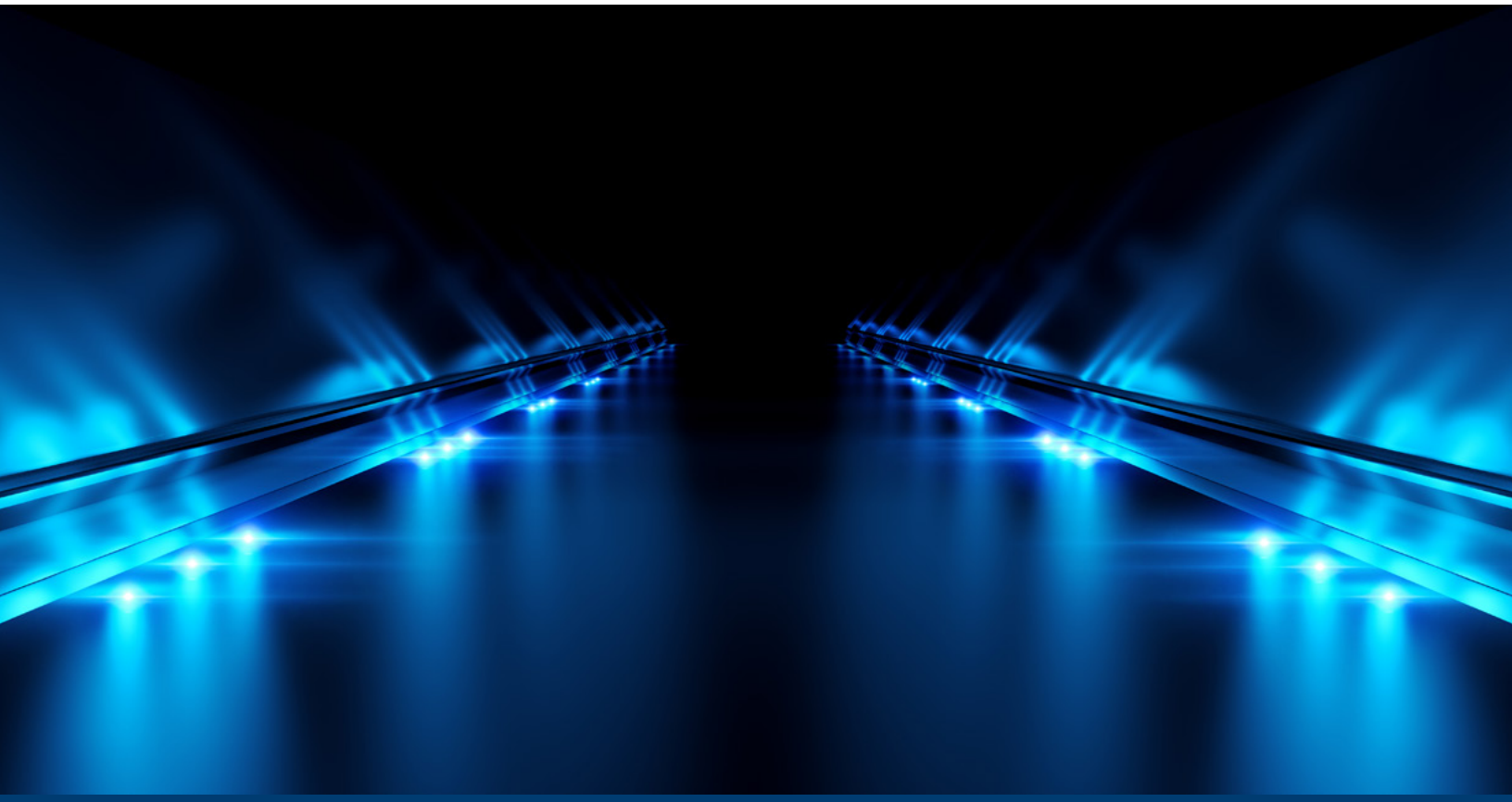

R を使用した HPC: 基本

Drew Schmidt テネシー大学ノックスビル校 大学院研究助手

R はハイパフォーマンス・コンピューティング (HPC) に適していないと言い切る人もいれば、その可能性は否定しないものの HPC での R の使用に懐疑的な人もいます。

しかし、世界は変化しています。データ解析は、新しく注目を集めている分野です。データサイエンスに求められるものが、具体的な利点であっても、あるいは利益であっても、R は HPC 分野において優れた能力を発揮します。HPC は、データ解析アプリケーションとそのユーザーに対応するため変化しています。そのような状況において、圧倒的な人気を誇る R が注目されるのは当然と言えます。すでに多くの開発者が HPC 環境での R の利用に取り組んでいます。

この記事は、R に馴染みはないが、興味がある読者を想定しています。皆さんの動機は、アプリケーションのニーズを満たすため、皆が R を使用しているから、あるいは顧客からの R ソリューションに対する要求の高まりなど、さまざまでしょう。どのような動機であっても、皆さんがこの記事を手に取ってくださったことを嬉しく思います。ここでは、皆さんが HPC に精通しており、インテル ® コンパイラーを使用しているという前提で説明します。

これらの前提を基に、まず、R の基本について述べ、できるだけ中立的な立場から R の優れた点をいくつか紹介します。数千ノードの分散コンピューティングで R を使用して、テラバイトのスピードを達成する話のほうが面白いかもしれません。それも可能ですが、R は HPC 分野においてあまりよく知られていないため、高度なトピックをカバーする前に基本的なトピックをカバーしたいと思います。

最初に、R を選択し、パフォーマンスに関心がある開発者であれば、知っておくべき簡単な歴史と基本事項を説明します。その後、コンパイル済みコードを R に統合する方法を示し、並列コンピューティングについて述べたいと思います。この記事は、概要を示すことを目的としているため、それぞれのトピックについてここでは詳しく取り上げません。この記事が、実際に R を試してみるきっかけになればと願っています。

R の歴史

1976 年、当時の Bell Labs の John Chambers が取り組み始めた S が R のルーツです。S は、Fortran コードの対話型インターフェイスとして設計されました。Fortran も長い歴史がありますが、R も同じです。R は、1990 年代に無料版の S として Ross Ihaka と Robert Gentleman によってリリースされました。厳密には、R は S 言語の方言です。実際、1980 年代に作成された多くの S コードが R でも動作します。

R は奇妙な言語です。以下は、そのことを示すコード例です。

```
typeof(1)
## [1] “double”

typeof(2)
## [1] “double”

# 1:2 は数字 “1 2” のベクトル
typeof(1:2)
## [1] “integer”
```

上記のコードをよく観察すると、`1:2` はインデックスであると仮定できます。しかし、ほかにも奇妙なところがあります。一般に、`:` は予測が付きにくいものです。`"1":"2"` は、整数 1 と 2 のベクトルを返します。つまり、文字から整数へ ASCII 変換を行っているだけなのでしょう。例外として、`"A":"B"` はエラーになります。また、`:` は常に整数を扱うとは限りません。例えば、`1.5:2.5` とすることができます。

R には非常に活発なコミュニティーがあり、CRAN (Comprehensive R Archive Network) 上では 10,000 を超える、数値計算から最先端の統計およびデータサイエンス手法、インタラクティブなデータ解析 Web サイトの構築 (**Shiny** (英語)) まで、さまざまなパッケージが配信されています。R のパッケージ・インフラストラクチャーは素晴らしく、どれも問題なく動作します。ここまで、「当然のことだ」と思って読み進めてこられた Python* 開発者の皆さんも、どのようにしたら、プログラミングの知識がない統計学者達が問題なく動作するパッケージ・フレームワークを作成できるのか不思議ではありませんか？

R は人気があると前述しましたが、2016 年に発表された **IEEE Spectrum*** (英語) のプログラミング言語のランキングにおいて、R は C# と JavaScript* を抜いて第 5 位でした。プログラミング言語のランキングに R がランクインしたことは非常に興味深いです。R の支持者であっても、R がひどいプログラミング言語であることを公言しています。R は特殊ではあるものの、データ解析に非常に優れているため、それを上回る価値があると評価されたのかもしれません。

言い換えれば、R は映画「パイレーツ・オブ・カリビアン」のジャック・スパロウに似ているかもしれません。これまでに聞いたことのないような最悪の言語かもしれませんが、それについて聞いたことはあるでしょう。

手を掛けずに向上

アメリカのコメディアン W. C. Fields はかつて、「私がこれまでに会った中で一番の怠け者は、パンケーキが勝手にひっくり返るようにポップコーンを入れていたよ」と言っていました。この怠け者は、素晴らしいエンジニアになれたでしょう。何だかんだ言っても、誰かがやってくれるのであれば、わざわざ自分でする必要はないのですから。

R では、このように手を掛けずにパフォーマンスを向上するいくつかの方法があります。最初に、これは驚くべきことではありませんが、R を適切なコンパイラーでコンパイルすると、パフォーマンスが少なからず向上します。R は C、Fortran、および R で記述されているため、インテル製ハードウェアでインテル® コンパイラー (icc や ifort) を使用すると良いでしょう。インテル® デベロッパー・ゾーンには、インテル® コンパイラーで R を使用する方法に関して**役立つ記事** (英語) があります。

これは、R で優れたパフォーマンスを引き出すための強力な最初のステップと言えます。ベースの R を構成している R コードについてはどうでしょうか？R 2.13.0 以降にはバイトコード・コンパイラーが含まれており、それを使用してバージョン 2.14.0 以降の内部コードをコンパイルできます。ユーザーコードに関しては、以前はバイトコード・コンパイラーを使用する方法を自力で見つける必要がありましたが、2017 年 4 月にリリースされた R 3.4.0 から JIT が含まれるようになり、それまでのバイトコード・コンパイラーの使用に関する推奨事項の多くは意味のないものになりました。

バイトコード・コンパイラーは、一般的なコンパイラーからは程遠く、例えば、コードに不要な処理が含まれていても、その処理はそのままバイト形式で実行されます。また、R から C へ変換してくれるわけではありません。ループを多用するコード (ただし、暗黙のループは除く) で最適に動作しますが、パフォーマンスにはあまり効果がありません。ループ本体のパフォーマンスが 10% 向上したケースもありますが、パフォーマンスが 0.01% しか変わらなかったケースもあります。しかし、自分では何もしなくてもいいのですから、少しでも向上できれば良いでしょう。

これらの改善は R コードのランタイムにとってプラスとなりますが、驚くような成果は得られないでしょう。新しいマシンでは、適切な LAPACK ライブラリーと BLAS ライブラリーを選択することで、パフォーマンスを大幅に向上することができます。これらは、行列演算向けの業界標準の数値演算ライブラリーで、R では、低レベルの線形代数と統計処理のほとんどにこれらのライブラリーを使用します。皮肉なことに、統計における最も重要な操作とも言える線形回帰では、LAPACK ではなく、LINPACK を大幅に改良したバージョンを使用します。私が言っているのは、スーパーコンピューターで実行するベンチマークのことではなく、1970 年代の LAPACK の前身のことです。その理由は少し複雑ですが、ちゃんとした理由があります。高度にチューニングされた LAPACK は、線形回帰に役立ちませんが、レベル 1 BLAS の利点を得ることはできます。

BLOG HIGHLIGHTS

ネットワーク・ソフトウェア開発者 – 将来に向けて取り組んでいますか?

ELIZABETH WARNER (INTEL) ›

インテル® Software Innovator Program は、先見の明のあるプロジェクトとそれを開発するスキルがある革新的な独立系開発者を支援します。イノベーターは、専門知識と最先端のテクノロジーを利用した革新的なアイデアを通して、主要分野のデベロッパー・コミュニティーに刺激を与えることができる創意工夫、実験、進歩的な考え方を示します。インテル® Software Innovator Program は、いくつかの技術分野に注目しており、それらがこのプログラムのブランチを構成しています。この記事では、その中からネットワーキングについて取り上げます。

インテル® Networking Developer Zone

SDN (Software Defined Networking) と NFV (Network Function Virtualization) では、パケット処理関数に柔軟性が求められ、パケット処理関数ではデータプレーンのソフトウェア実装が求められます。**Networking Program** (英語) は、x86 ベースの大容量サーバー上で SDN と NFV への対応に取り組む開発者を支援します。インテル® アーキテクチャーは、動的にアップグレード、保守、スケーリングできる、SDN/NFV 向けの標準の再利用可能な共有プラットフォームを提供します。

この記事の続きはこちら (英語) でご覧になれます。›

R には " リファレンス " BLAS と呼ばれる、非常に遅いライブラリーが含まれています。上記の例でも、R と適切な BLAS および LAPACK 実装を使用することで、大幅なパフォーマンスの向上を期待できます。幸運にも、**インテル® マス・カーネル・ライブラリー (インテル® MKL)** は、非常に優れた実装を提供します。Microsoft* は、インテル ® コンパイラーでコンパイルした R とインテル® MKL を同梱した R のディストリビューション、**Microsoft* R Open** (英語) を無料で提供しています。インテル® MKL の優れたパフォーマンスを実証する**ベンチマーク** (英語) のコレクションもあります。また、インテルからは、R とインテル® MKL をリンクするための**ドキュメント** (英語) が提供されています。

最新のマシンをお持ちの場合、MIC アクセラレーターでもこれを利用できます。この初期の研究は、**Texas Advanced Computing Center (TACC)** (英語) において行われ、特にインテル® MKL の自動オフロードがテストされました。問題なく動作するというのは、控えめな表現であって、インテル® Xeon Phi™ プロセッサーをお持ちの R ユーザーは、ぜひこれを試してみるべきです。どこから手を付けていいのか分からない方のために、インテルから**ガイド** (英語) が提供されています。

コンパイル済みコードの利用

最近の R の傾向の 1 つとして、R パッケージでの C++ の使用が増加しています。これは、主に **Rcpp** (英語) パッケージを作成した Dirk Eddelbuettel と Romain Francois の功績によるものです。Rcpp を利用することで、R 解析パイプラインに C++ コードを非常に簡単に組込むことができます。Rcpp は、Python* は C++ でプログラムするには複雑すぎると思っていた統計学者達の考えを一変させました。皆さんと同じように、私もこれには驚きました。しかし、どのような方法であったにしても、我々はその恩恵を受けられるようになりました。これにより、CRAN パッケージはメモリー消費を抑えつつ、さらに高速に動作することができます。また、インテル® コンパイラーを使用している場合は、最も大きな利点が得られるでしょう。前述のとおり、最適なコンパイラーは、高速なコードをもたらすからです。

C を使用している開発者向けに、R には素晴らしい C API があります。実際、これをベースに (多大な労力をつぎ込んで) Rcpp は構築されました。C API は、C++ リンカーを使用せずに C ラッパーを作成したいと考えている Fortran 開発者にとっても便利です。API に関する説明は、**Writing R Extensions** (英語) にあります。このマニュアルは、R を使用する開発者にとって必須と言えます。しかし、問題が見つかった場合は、R のヘッダーファイルを確認することになるかもしれません。

Rcpp を使用するには、次のように、ほかの R パッケージと同様の方法でインストールします。

```
install.packages("Rcpp")
```

このパッケージを評価するため、皆さんもよくご存じの π を求める " 数値版 Hello World" (モンテカルロ積分) について考えてみます。R では、次のように記述することができます。

```
mcpi <- function(n)
{
  r <- 0L

  for (i in 1:n){
    u <- runif(1)
    v <- runif(1)

    if (u*u + v*v <= 1)
      r <- r + 1L
  }

  return(4*r/n)
}
```

これは、R にあまり馴染みのない方の書き方です。R に精通した方がこのように記述することはあまりないでしょう。上記のコードは C に似ています。コードを高速にするため R を使用する場合、C に似ているほど、R ではうまく動作しませんが、C/C++ への変換は簡単に行うことができます。逆も同様で、R に似ていないほうが、変換は困難です。より自然な R コードは、次のようにベクトル化されています。

```
mcpi_vec <- function(n)
{
  x <- matrix(runif(n * 2), ncol=2)
  r <- sum(rowSums(x^2) <= 1)

  return(4*r/n)
}
```

ほかの高水準言語と同様に、**ベクトル化**により実行パフォーマンスは向上しますが、RAM の使用量も増えます。いったん R から離れ、C++ でコードを記述してみましょう。

```
#include <Rcpp.h>

// [[Rcpp::export]]
double mcpi_rcpp(const int n)
{
  int r = 0;

  for (int i=0; i<n; i++){
    double u = R::runif(0, 1);
    double v = R::runif(0, 1);

    if (u*u + v*v <= 1)
      r++;
  }

  return (double) 4.*r/n;
}
```

Rcpp::export を除いて、C++ コードになりました。**Rcpp::export** は、すべての定型コードを生成します。R にはスカラーがなく (冒頭で奇妙な言語と言ったとおりでしょう)、長さ 1 のベクトルになります。内部で Rcpp は処理を行い、ラッパーで長さ 1 の倍精度ベクトルを作成します。一般に、整数、倍精度、およびそれらのベクトルを使用できます。複雑な問題には、複雑な処理が伴いますが、これで十分と言えるでしょう。

さまざまな定型コードをコンパイル、リンク、ロード、および生成するには、**sourceCpp()** を呼び出すだけで、R でその関数をすぐに利用できます。

```
Rcpp::sourceCpp(file="mcpi.cpp")
mcpi_rcpp(10000)
## [1] 3.1456
```

注意深い方は、上記のコードで '10000' が倍精度かどうか疑問に思われるかもしれません。そのとおりです。**10000L** と指定すると、通常の 32 ビット整数になります。Rcpp は、自動的に型変換を行い、その方法は R コードと同じです。これについては賛否両論あるでしょうが、Rcpp と R ではこのように扱われることを知っておいたほうが良いでしょう。

ここで紹介した 3 つのコード例のパフォーマンスは、rbenchmark または microbenchmark パッケージを使用して簡単に比較できます。ここでは、rbenchmark を使用しました。

```
library(rbenchmark)

n <- 100000
cols <- c("test", "replications", "elapsed", "relative")
benchmark(mcpi(n), mcpi_vec(n), mcpi_rcpp(n), columns=cols)
##           test replications elapsed relative
## 1      mcpi(n)          100  49.901  214.167
## 2  mcpi_vec(n)          100   1.307    5.609
## 3 mcpi_rcpp(n)          100   0.233    1.000
```

上記のように、良い結果が得られました。OpenMP* や**インテル® スレッディング・ビルディング・ブロック (インテル® TBB)** などを利用することもできます。次に、並列処理について考えてみましょう。

並列プログラミング

R 2.14.0 以降には parallel パッケージが同梱されています。ソケットを使用する API と OS の fork を使用する API により、単純なタスクレベルの並列処理を行うことができます。2 つのインターフェイスが存在する理由はいくつかあります。その 1 つは、各 API が別々の古い multicore パッケージと snow パッケージから派生したためです。しかし、主な理由は、(**`fork`** がない) Windows* も含め、すべてのプラットフォームを R コアでサポートするためです。Windows* 以外のプラットフォームでは、**`mclapply()`** 関数が **`lapply()`** のマルチコアバージョンです。**`lapply()`** は関数を適用し、リストを返すため、このような名前が付けられました。この関数では、R の機能的なプログラミングの特徴が活かされています。

```
lapply(my_data, my_function)
parallel::mclapply(my_data, my_function)
```

データは非常に大きく複雑なオブジェクトのインデックスまたは複雑なリストとして表現され、関数が処理できる入力値である限り、問題なく動作します。

正式にサポートされている 2 つ目のインターフェイスは、やや複雑で、通常 Windows* プログラマーのみが使用します。これは、R ユーザーの間にわずかな議論を生じさせています。私の個人的な意見ですが、R ユーザーは、ほかの言語のプログラマーほど、複数の選択肢を好まず、適切な 1 つの方法だけで十分だと考えているように思われます。そのため、2 つのインターフェイスの統合に向けて、複数のプロジェクトが立ち上げられました。例えば、古く実績のある foreach パッケージや Bioconductor プロジェクトからの新しい BiocParallel などです。

2 つのインターフェイスが存在すると問題があるのか疑問に思われる方もいらっしゃるでしょう。実際、R にはこのほかにも並列処理を可能にする多数のパッケージが存在します。それらについては、**HPC Task View** (英語) で確認できます。

多くの取り組みと関心が共有メモリー並列処理に集中しています。R コミュニティーは、比較的最近までパフォーマンスに対する関心が低かったのですが、その主な理由は、R ではまだデータサイエンスの統計的側面が重視されているためだと思われます。統計分野のほとんどのビッグデータの問題では、そのようなことはありません。確かに、データをダウンサンプリングして、従来の統計手法を使用するだけで、多くのことに対応することができます。並列処理を可能にするパッケージは、すべての問題に適用できるわけではありませんが、その需要はあります。それどころか、これらのツールは正当に評価されていないと言えます。

しかし、スーパーコンピューターで利用されているものはないため、これ以上の議論は無意味でしょう。次に、MPI についてお話ししましょう。Rmpi パッケージの歴史は 2002 年まで遡ります。最近では、**Programming with Big Data in R (pbdR)** (英語) プロジェクトが、スーパーコンピューティング環境で R を使用して大規模計算を行うパッケージを開発しています。実は、私もそのプロジェクトのメンバーです。そのため、私の意見は偏っているかもしれませんが、いくつかの興味深い例を皆さんにお見せしたいと思います。

pbdR プロジェクトでは、いくつかのパッケージを管理し、HPC を最大限に利用して R でデータ解析とプロファイルを行うため取り組んでいます。ここでは、MPI プログラミングに絞って述べたいと思います。Rmpi と pbdMPI パッケージを比較してみましょう。1 番大きな違いは、Rmpi はインタラクティブに使用できますが、pbdMPI はできません。これは、pbdMPI が 1 つのプログラムで複数のデータを扱うこと (SPMD: Single Program Multiple Data) を想定して設計されているのに対し、Rmpi はマネージャー / ワーカー形式で動作するためです。MPI を使用してクラスター上でバッチジョブを送信する場合、SPMD で記述することがほとんどです。それが最も直感的な方法で、名前も付けられています。HPC から R に来られた方には、pbdMPI は馴染みやすいでしょう。

プログラミング形式に加えて、2 つのパッケージの API には大きな違いがあります。Rmpi では、データ型を指定する必要があります。次に例を示します。

```
library(Rmpi)
mpi.allreduce(x, type=1) # int
mpi.allreduce(x, type=2) # double
```

この記事の最初に説明した型の例を思い出してください。pbdMPI では、R のオブジェクト指向の機能を活用して、型やその他の下位レベルの詳細を自動的に処理します。

```
library(pbdMPI)
allreduce(x)
```

単純な例ですが、うまく示されています。HPC コミュニティーの取り組みは素晴らしいものですが、HPC ツールはほとんどの人には使用することが難しく、もっと使いやすくあるべきです。

より現実的な例として、並列版の "Hello World" について考えてみます。前述のとおり、pbdMPI はバッチモードで使用する必要があります。インタラクティブな処理に慣れている R ユーザーにとって、バッチ処理は馴染みがないものですが、リソース・マネージャーやジョブ・スケジューラーなどの HPC 機能とうまく連携できます。以下の "Hello World" を実行してみましょう。

```
library(pbdMPI)

comm.print(paste("Hello from rank", comm.rank(), "of", comm.size()), all.
rank=TRUE)

finalize()
```

mpirun (または同等のコマンド) を呼び出すだけです。

```
mpirun -np 2 Rscript hello_world.r
```

以下が出力されます。

```
[1] “Hello from rank 0 of 2”
[1] “Hello from rank 1 of 2”
```

まとめ

統計分野でよく知られている、George Box の「すべてのモデルには間違いがあるが、役に立つものもある」という言葉があります。それに見習えば、「すべてのプログラミング言語には問題があるが、役に立つものもある」と言えます。R は、その究極の言語でしょう。(S から数えて) 40 年間も支持され続け、最近ではプログラミング言語ランキングの第 5 位にランクインした実績から、R にはそれだけの価値があるはずです。R は遅いことで知られていますが、それを解決する方法はあります。まずは、適切なコンパイラーを使用することです。また、適切な BLAS と LAPACK は、多くのデータサイエンス処理のパフォーマンスを向上します。さらに、R 解析パイプラインにコンパイル済みカーネルを組込むことで、パフォーマンスを大幅に向上できます。それでも期待するパフォーマンスが得られない場合は、コア数を増やしてみてください。