応用数理工学特論
第 2 回授業ノート

計算理工学専攻 張研究室
鈴木 綾華，早戸拓也

2008 年 4 月 24 日

目次

1 序論

スーパーコンピュータの性能は図?? のように変化してきた．この図から，1990 年代を機に性能の向上率が上がっていることわかる．並列計算機の登場である．並列計算機とは，複数のプロセッサを並列につないだもので，プロセッサ数を増やすことでピーク性能を無制限に向上させることができる．また汎用のプロセッサを使用することで，設計コストの大幅な削減も可能である．これにより，プロセッサの動作周波数向上の飽和と，専用スーパーコンピュータの設計コスト増加の問題が軽減された．

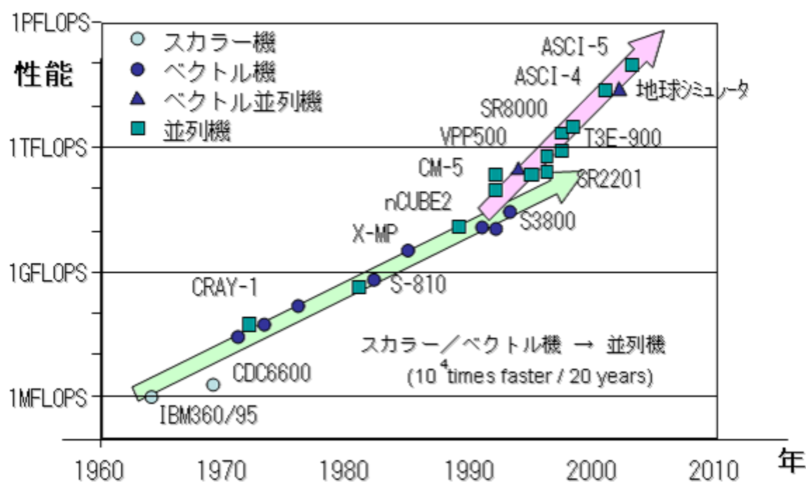


図 1　スーパーコンピュータの性能動向

また，並列計算機を効率的に活用するには複数のプロセッサに均等に演算を分担させ，待ち時間を減らすための良いアルゴリズムを構築することが重要になる．

さらに，並列計算機には下記のような種類

- 共有メモリ型並列計算機
- 分散メモリ型並列計算機
- SMP クラスタ

があるが，良い並列化アルゴリズムは計算環境によって異なるので，種類に合わせたプログラミングが必要である．以下の章では，これらの並列計算機の構造や特徴について詳しく述べる．

2 共有メモリ型並列計算機

共有メモリ型並列計算機は図?? のように複数のプロセッサ (PU) がバスを通してメモリを共有し，PU はそれぞれキャッシュを持っている．メモリ空間が単一であるためプログラミングが容易ではあるが，一方弊害として PU の数が多すぎると，アクセス競合により性能が低下する．そのため使用の際は一般的に 4〜8 台ほどの並列が多い．プログラミング言語は OpenMP(6 節にて説明) を用いる．

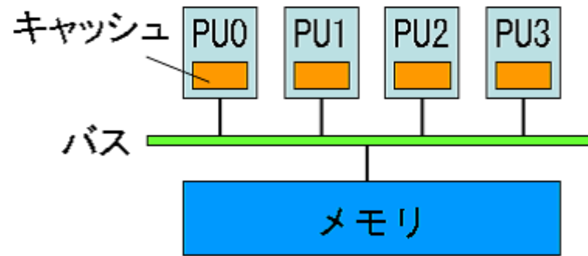


図 2　共有メモリ型並列計算機

3　分散メモリ型並列計算機

共有メモリ型並列計算機は図?? のように各々がメモリを持つ複数の PU をネットワークで接続したものであり，各 PU はそれぞれにキャッシュを持っている．各々がメモリを持つのでアクセス競合は起こらず，数千〜数万 PU 規模の並列化が可能である．ただし PU 間のデータ通信が必要なため，データ分散，通信を意識したプログラミングが必要になる．

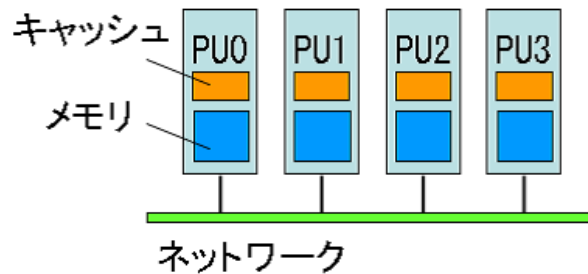


図 3　分散メモリ型並列計算機

4　SMP クラスタ

SMP クラスタは図?? のように，複数の共有メモリ型並列計算機 (SMP) をネットワークで接続したものである．各ノードの性能を高くできるため，比較的少ないノード数で高性能を達成することができる．プログラミングは，ノード内部の計算では共有メモリ型並列計算機として，ノードをまたがる計算では分散メモリ型並列計算機として行う．そのため，プログラミング言語は MPI と OpenMP とを組み合わせて使用する．

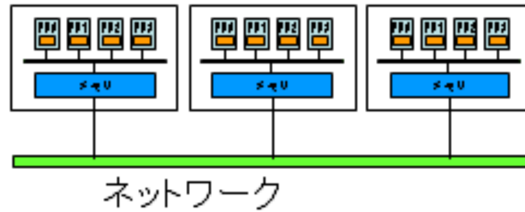


図 4　SMP クラスタ

5　並立化効率

並列計算の実行時間は

$$\text{並列実行時間} = \text{演算時間} + \text{通信時間} + \text{待ち時間}$$

となっている．この式から，単純に演算時間だけを考えれば良いのではないことがわかる．式の中で，通信時間は PU 間のデータ通信にかかる時間，待ち時間は複数の PU 間で演算時間が異なり，データ通信のために別の PU が演算を中止している時の演算処理待ち時間である．各プロセッサに同程度の演算負荷をかけることが待ち時間を少なくする良いプログラムである．

p 台のプロセッサを使用した場合の並列化効率は

$$\text{並列化効率} = \frac{1\ \text{プロセッサでの実行時間}}{p\ \text{プロセッサでの実行時間} \times p}$$

となる.、並列化効率が 1 に近いほど，良い並列化アルゴリズムであると言えるが，実際には通信時間や待ち時間のため，1 より小さい値となる．

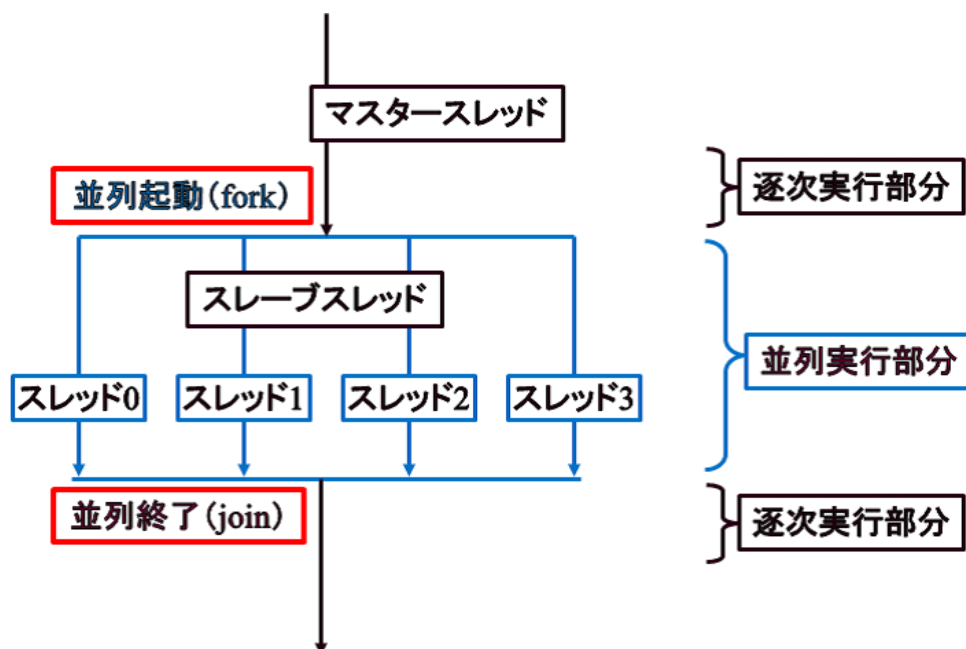
6 OpenMP

OpenMP とは，共有メモリ型並列計算機上での並列プログラミングのためのプログラミング環境である．ベース言語となる FORTRAN・C・C++ にディレクティブ（指示文）を加えることで，並列プログラミングに拡張することができる．

6.1 OpenMP の実行モデル

OpenMP の実行モデルは，Fork-join モデルが用いられている．Fork-Join モデルとは，並列化を指定しない部分は逐次的に実行され，指示文で指定された部分のみを複数のスレッドで実行するようなモデルである．このとき，各スレッドは同じメモリ空間にアクセスする．この OpenMP で記述されたプログラムは，最初にマスタースレッドが起動され，逐次的にプログラムを実行する．プログラムが並列構文に到達すると，スレーブスレッドと呼ばれるスレッドを生成 (スレッドを Fork する) し，分割されたプログラムのタスク を並列に処理する．スレーブスレッドの処理は，プログラム中での並列実行領域の終了を示す構文に到達すると終了する．この終了時には，全スレッドが各自の処理を終了するまで先に終了したスレッドも待つことになり， 同期処理を必要とする．全スレッドが完了した時点 (スレッドを join する) で，プログラムの実行処理は再びマスタースレッドだけが行うことになる．

図 5　Fork-Join モデル



6.2 OpenMP の構成要素

OpenMP を用いたプログラムは，以下のような構成要素から成る．

- **元のプログラム** 通常の FORTRAN または C/C++ で書かれたプログラム.
- **指示文** 並列化すべき場所 (Parallel Region)，並列化方法を指定.
- **ライブラリ関数** 並列処理において様々な働きを行う関数.
  (例)
  -omp_get_thread_num : 関数を実行するスレッドのスレッド番号を返す.
  -omp_get_num_procs　: 関数の呼び出し時点で使用できるプロセッサ数を返す.
  -omp_get_wtime　　　: あるポイントからの経過時間の秒数値を返す.
- **環境変数** プログラムの並列実行に関する様々な環境を指定する変数.
  (例)
  -OMP_NUM_THREADS : 並列実行部分で使うスレッド数を指定する.

6.3 OpenMP を用いたプログラム例

環境変数 OMP_NUM_THREADS を 2 に設定する.

6.3.1 並列実行を行うスレッド番号を出力するプログラム

下記のプログラムをコンパイル・実行する.

```
program hello
integer omp_get_thread_num    ⇐ スレッド番号を取得するライブラリ関数
write(6,*) ‘ program start. ’
!$omp parallel    ⇐ 指示文： 並列実行部分の開始
write(6,*) ‘ My thread number = ’ , omp_get_thread_num()
!$omp end parallel    ⇐ 指示文：並列実行部分の終了
write(6,*) ‘ program end. ’
stop
end
```

実行結果

My thread number = 0

My thread number = 1

6.3.2 ベクトルの加算 $\boldsymbol{z} = a\boldsymbol{x} + \boldsymbol{y}$

下記のプログラムをコンパイル・実行する.

```
program axpy
integer i
double precision a, x(100), y(100), z(100)
(a, x(100), y(100) の値を設定)
!$omp parallel do    ⇐ 指示文： 直後の do ループの並列化指示
```

```
do i = 1, 100
   z(i) = a*x(i) + y(i)     ⇐ ベクトルの加算 z = ax + y
end do
(z(100) の値を表示)
stop
end
```

実行結果
-1≤ i ≤50 がスレッド 0，51≤ i ≤100 がスレッド 1 で計算される．

6.4 共有変数とプライベート変数

OpenMP を用いたプログラムでは，全ての変数，配列に対してそれぞれ，共有変数 (shared) かプライベート変数 (private) かのどちらかの属性が与えられる．各属性について，以下に簡単に説明する．

共有変数

共有変数とは，どのスレッドからも書き込み，もしくは読み出し可能である変数，配列のことである．OpenMP のプログラミングモデルでは，基本的にすべての変数は共有変数となる．共有変数を指定する場合，並列化指示文の後に shared 節を追加すればよい．しかし，何も指示をしない変数については，基本的に共有変数となるため，通常は指定する必要がない．

プライベート変数

プライベート変数とは，あるスレッドからのみ書き込み，もしくは読み出し可能である変数，配列のことである．ループインデックス変数については，異なったスレッドで共有すると，正しい制御ができないため，このような変数が必要となる．プライベート変数を指定する場合，並列化指示文の後に private 節を追加すればよい．しかし，並列化されたループのインデックス変数のように，明らかにプライベート変数でなければならない変数については，特に指示をしなくてもプライベート変数となる (ただし，多重ループの場合は並列化対象ループのインデックス変数のみプライベート変数となる)．

6.5 変数の属性指定のプログラム例

6.5.1 2 重ループの並列化 (行列ベクトル積)

下記のプログラムをコンパイルし，実行する．

```
program gemv
integer i, j
double precision a(100,100), x(100), y(100)
(a(100,100)，x(100) の値を設定)
!$omp parallel do private(j)     ⇐ j をプライベート変数に指定
do i = 1, 100
   y(i) = 0.0d0
```

```
    do j = 1, 100
      y(i) = y(i) + a(i,j) * x(j)
    end do
  end do
  (y(100) の値を表示)
  stop
  end
```

上記のプログラムでは，a, x, y は自動的に共有変数となる．また，i は明らかにプライベート変数でなければならない変数とみなされ，自動的にプライベート変数となる．しかし，j は並列化対象ループのインデックス変数ではないために，自動的にはプライベート変数とはならない．そのため，j がプライベート変数であることを指定する必要がある．

6.6 リダクション変数

下記の内積を求めるプログラムについて考える．

```
program dot
integer i
double precision x(100), y(100), c
(x(100)，y(100) の値を設定)
c = 0.0d0
do i = 1, 100
  c = c + x(i)*y(i)
end do
(c の値を表示)
stop
end
```

ここで，並列化をおこなう際の，内積の結果を表す変数 c について考える．c を共有変数とした場合，共有変数に各スレッドに同時にアクセスするため，総和が正しく計算できない．この問題は，c に同時にアクセスするスレッドを 1 つに制限する排他制御を行うことで回避できるが，同時に処理することができないため，並列処理による計算時間の短縮は期待できない．また，c をプライベート変数とした場合，並列処理をおこなう事が出来るが，各スレッドで違う値を持つため，マスタースレッドにプログラムの実行が移る時に，c の値はどのスレッドが持っていたものか解らなくなってしまう．このため，並列実行部分終了後に，c の値が消えてしまうことから，内積を計算できない．このような問題を解決するために，リダクション変数を使用する．

リダクション変数

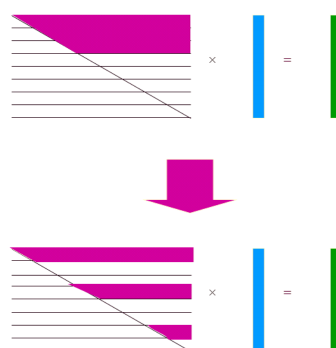
リダクション変数とは，並列実行部分ではプライベート変数で，並列終了時に各スレッドの値が合計されるような変数である．並列ループが開始され，各スレッドがループを実行している間はプライベート変数として扱われる．そのため，各スレッドはそれぞれのリダクション変数に対して独自の値を持つことになる．その後，全スレッドが並列ループの処理を終了すると，各スレッドのリダクション変数の値は１つにまとめられ，共有変数のようにマスタスレッドの対応する変数に格納される．このリダクション変数を１つにまとめる際に，和，積，最大値等の計算処理が行われる．リダクション変数を指定する場合，指示文に reduction 節を追加する．上述のベクトルの内積を例にして，リダクション変数を用いると，以下のようなプログラムになる．

```
program dot
integer i
double precision x(100), y(100), c
(x(100), y(100) の値を設定)
c = 0.0d0
!$omp parallel do reduction(+:c)    ⇐ c をリダクション変数に指定（総和型）
do i = 1, 100
  c = c + x(i)*y(i)
end do    ⇐ ここで c の値が合計される
(c の値を表示)
stop
end
```

6.7　ループへのスレッド割り当ての指定

OpenMP では，スレッド割り当てを特に指定しない場合，インデックス変数の値の範囲をスレッド数分に分割し，各部分をスレッドに割り当てる．しかし，このように割り当ててしまうと，上三角行列とベクトルの積の計算などでは，各スレッドで負荷が異なってしまい，並列計算の効率化が低下してしまう．このような場合は，一定の長さ L のブロックを周期的にスレッドに割り，当てるブロックサイクリック割り当てを採用すると，負荷分散を改善できる場合がある．

図 6　上三角行列へのブロックサイクリック割り当ての適用



6.7.1　ブロックサイクリック割り当てのプログラム例

下記に上三角行列とベクトルの積を並列計算するプログラムを記す．

```
program gemv
integer i, j
double precision a(100,100), x(100), y(100)
(a(100,100), x(100) の値を設定)
!$omp parallel do private(j) schedule(static,10)    ⇐ ブロック幅 10 のブロックサイクリック割り当て
```

```
do i = 1, 100
   y(i) = 0.0d0
   do j = i, 100
      y(i) = a(i,j) * x(j)    ⇐ 上三角行列とベクトルの積
   end do
end do
（y(100) の値を表示）
stop
end
```

上記のプログラム例では，コンパイル時に割り当てを決める静的 (static) 割り当てが行われる．また，実行中に適宜割り当てが行われる動的 (dynamic) 割り当てや，環境変数による割り当て方法の指定も可能である．

6.8 セクション型の並列化

今までは主に do ループの並列化ついて述べてきたが，各スレッドに別々の仕事を並列に行わせたい場合は，セクション並列化を指定すればよい．

```
program main
（逐次処理の部分）
!$omp parallel sections    ⇐ セクション並列化の開始
!$omp section
（ここにスレッド 0 の処理内容を記述）
!$omp section
（ここにスレッド 1 の処理内容を記述）
!$OMP end parallel sections    ⇐ セクション並列化の終了
（逐次処理の部分）
stop
end
```

7 高性能化の技法

7.1 高速化の考え方

共有メモリ型並列計算機において並列化を効率よく行うためには 3 つの重要な点がある．

7.1.1 PU 間の負荷分散均等化

各 PU の処理量が均等になるよう処理を分割することで，演算を終えた PU の待ち時間を減らすことができる [図??].

7.1.2　PU 間同期の削減

すべての PU が処理を終えると，同期という作業を行い，各プロセッサの結果を参照するが，これには一般的に数百サイクルの時間が必要となる [図??]．つまり，並列粒度 (PU 間での同期なしに行える処理の大きさ) の増大による同期回数の削減により高速化が期待できる．

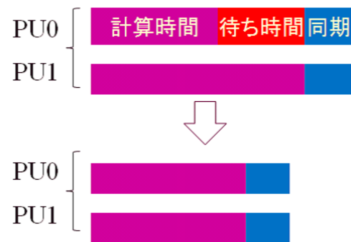


図 7　処理量均等化

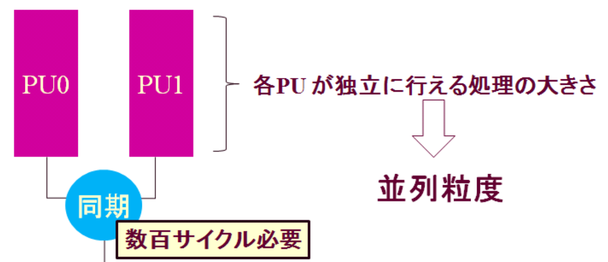


図 8　並列粒度と同期

7.1.3　キャッシュメモリの有効利用

演算器と主メモリの速度差は，年々増大しており，さらに，CPU 数が多いとバスのオーバーヘッドが大きくなってしまう．これは，演算順序の変更等により，キャッシュ中のデータの再利用性を向上させることで，主メモリのデータ転送速度の遅さをカバーできる．

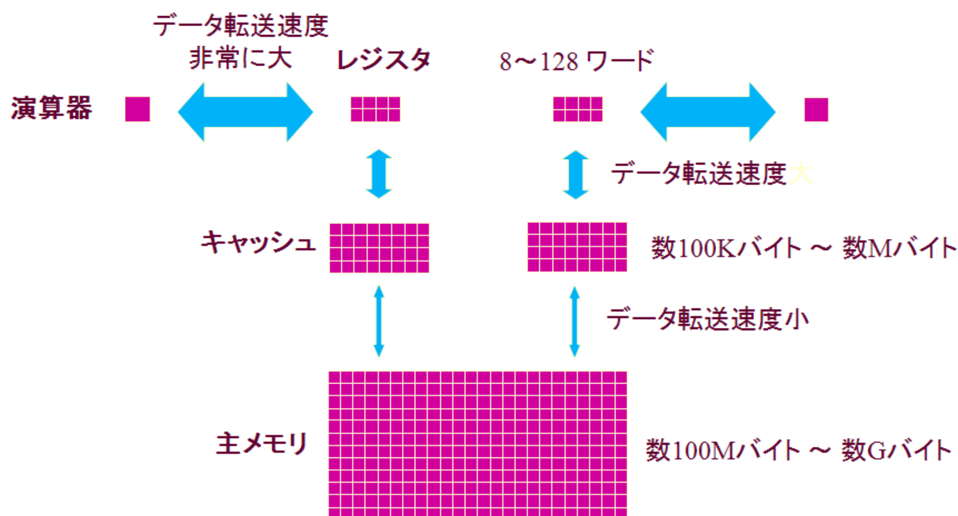


図 9　共有メモリ型並列計算機におけるメモリ階層の例

7.2　BLAS の利用による高性能化

BLAS(Basic Linear Algebra Subprograms) とは，個々のマシン向けにチューニングした基本行列演算のライブラリである．下記が主な現在利用可能な最適化 BLAS である．

- **プロセッサメーカーの提供する** BLAS
  -Intel MKL，AMD ACML，IBM ESSL など
  -メーカーによっては共有メモリ向け並列化版もあり

- ATLAS（Automatically Tuned Linear Algebra Subprograms）
  -対象プロセッサに合わせて自動的に性能を最適化する BLAS
  -http://math-atlas.sourceforge.net/ より入手可能
  -Intel MKL，AMD ACML，IBM ESSL など
  -共有メモリ向け並列化版もあり

- GOTO BLAS
  -テキサス大学オースチン校の後藤和茂氏による BLAS
  -http://www.tacc.utexas.edu/resources/software/ より入手可能
  -この 3 種の BLAS の中では，多くの場合最高速
  -共有メモリ向け並列化版もあり

7.2.1 BLAS におけるデータ再利用性と並列粒度

BLAS の種類には，BLAS1，BLAS2，BLAS3 とあり，下記にそれぞれの特徴を記す．

- BLAS1 ベクトルとベクトルの演算
  -内積　　: $c := \vec{x}^t \vec{y}$
  -AXPY 演算 : $\vec{y} := a\vec{x} + \vec{y}$　など
  -演算量 : $O(N)$　データ量 :$O(N)$
  -データ再利用性: なし
  -並列粒度 : $O(N/p)$　(N: ベクトルの次元，p: プロセッサ台数)

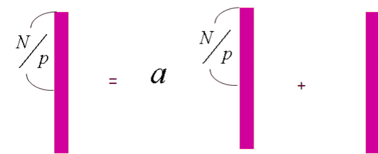


図 10　ベクトルとベクトルの和

- BLAS2 行列とベクトルの演算
  -行列ベクトル積　　: $\vec{y} := A\vec{x}$
  -行列の rank-1 更新 : $A := A + \vec{x}\vec{y}^t$
  -演算量 : $O(N^2)$　データ量 :$O(N^2)$
  -データ再利用性: ベクトルデータのみ再利用性あり
  -並列粒度 : $O(N^2/p)$

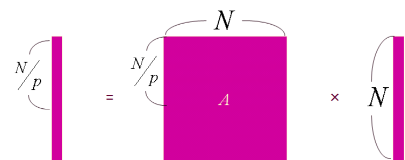


図 11　行列ベクトル積

- BLAS3 行列と行列の演算
  -行列乗算　　: $C := AB$　など
  -演算量 : $O(N^3)$　データ量 :$O(N^2)$
  -データ再利用性: $O(N)$ 回のデータ再利用が可能
  -並列粒度 : $O(N^3/p)$

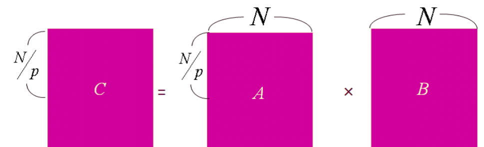


図 12　行列と行列の積

BLAS1 を使うよりもデータの再利用性が高い BLAS2，BLAS3 を使用したほうがキャッシュメモリを有効に利用できるので，並列化の効率向上につながる．よって，プログラムを作成する時，BLAS3 を出来るだけ使用できるように改良する必要がある．