

MPIによるプログラム 例の紹介, 及び チューニング演習

社会人向けスパコン実践セミナー

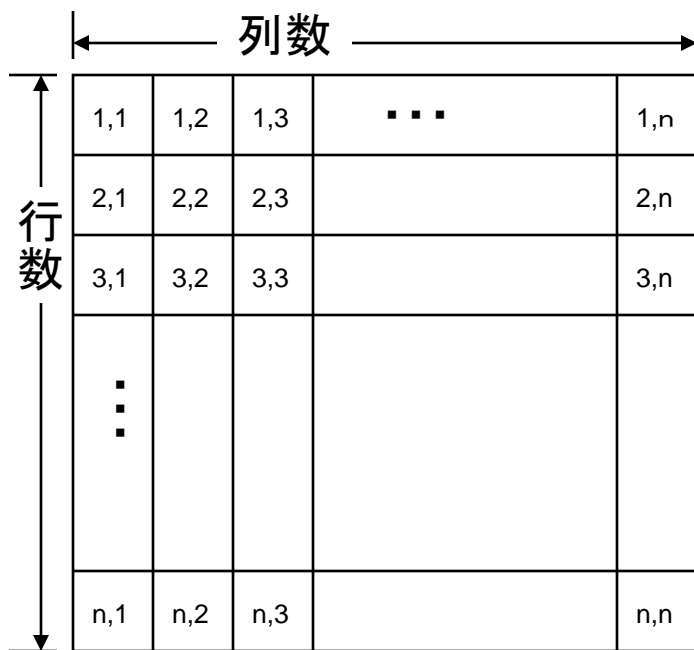
2009年6月17日(水曜)

シミュレーションプログラム

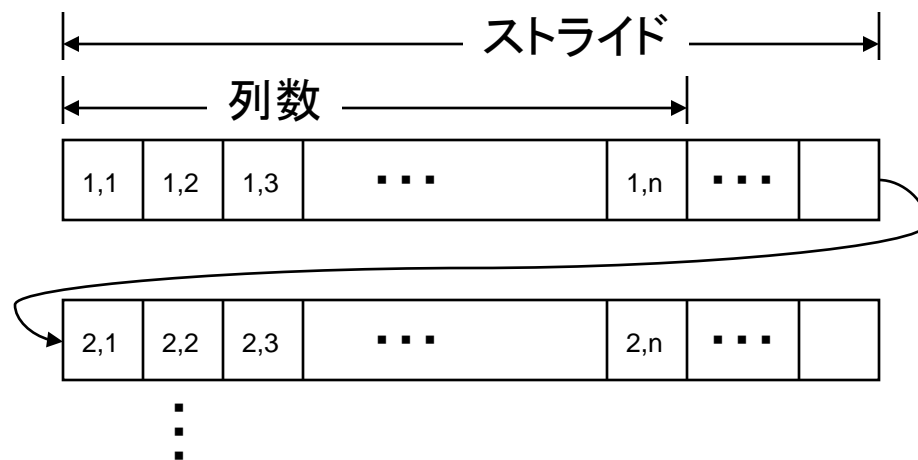
- 世の中で広く行われている複雑なシミュレーションも、基本的な数値計算を組み合わせて作られている。
- 計算機や計算プログラムの総合的な性能を比較するために、実際のシミュレーションプログラムを使う場合があるが、性能の向上方法を研究したり、プログラミングの演習の題材とする場合には、その要素となる個々の計算を対象とするほうがよい。
- シミュレーションプログラムの要素となる基本計算
 - 線形代数(行列の四則演算、固有値、固有ベクトル計算、連立一次方程式)
 - フーリエ変換
 - 積分計算
 - 並べ替え
 - etc.
- ここでは、このうち、線形代数計算の中で比較的単純な、行列積の問題を考えることとする。

行列積の計算(1)

- 線形代数演算は、さまざまな数値計算の中で現れ、時には、数万次元×数万次元の行列計算を必要とする場合もある。
- ここでは、その中で比較的簡単な正方行列の積計算を例として、その高速化と並列化を実行する。



i 行 j 列の要素を行列の (i,j) 要素と書くこともある。



計算機のメモリー上では、1次元的に並べて配置することが多い(この並べ方は、Cで二次元配列を利用する場合に良く使われる。Fortranの二次元配列では、行と列の並び方が反対となる)。

行列積の計算(2)

$\mathbf{A}, \mathbf{B}, \mathbf{C} : n \times n$ 正方行列

行列の積 $\mathbf{AB} = \mathbf{C}$ の計算

成分表示

$$\begin{aligned} \mathbf{A} &= (a_{ij}) \\ \mathbf{B} &= (b_{ij}) \\ \mathbf{C} &= (c_{ij}) \end{aligned} \quad \mathbf{AB} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$
$$= \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \mathbf{C}$$

行列積の計算(3)

行ベクトル、列ベクトルによる分割表示

\mathbf{a}_i : \mathbf{A} の第 i 行目の行ベクトル

$$\mathbf{a}_i = (a_{i1} \quad a_{i2} \quad \cdots \quad a_{in}) \quad \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \hline a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \vdots \\ \mathbf{a}_n \end{pmatrix}$$

\mathbf{a}^j : \mathbf{A} の第 j 列目の列ベクトル

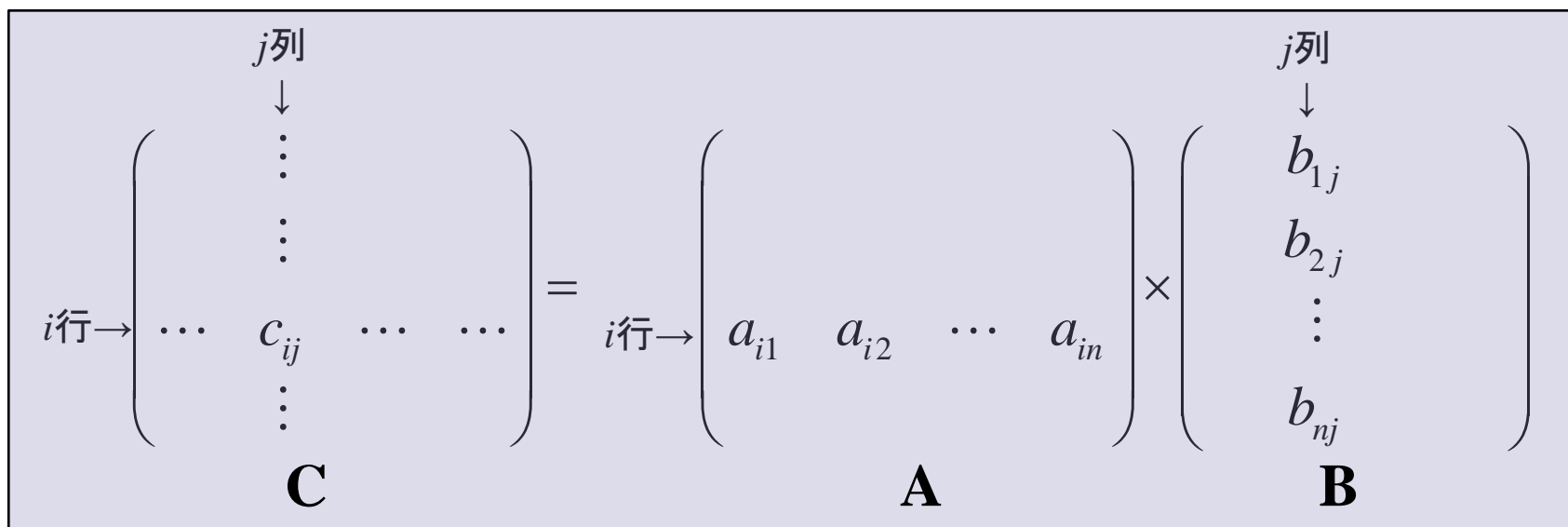
$$\mathbf{a}^j = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{nj} \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = (\mathbf{a}^1 \quad \mathbf{a}^2 \quad \cdots \quad \mathbf{a}^n)$$

行列積の計算(4)

行列 \mathbf{C} の i, j 成分 c_{ij}

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} = \underbrace{a_{i1}b_{1j} + \cdots + a_{in}b_{nj}}_{n\text{回の掛け算と足し算}} = \mathbf{a}_i \mathbf{b}^j$$

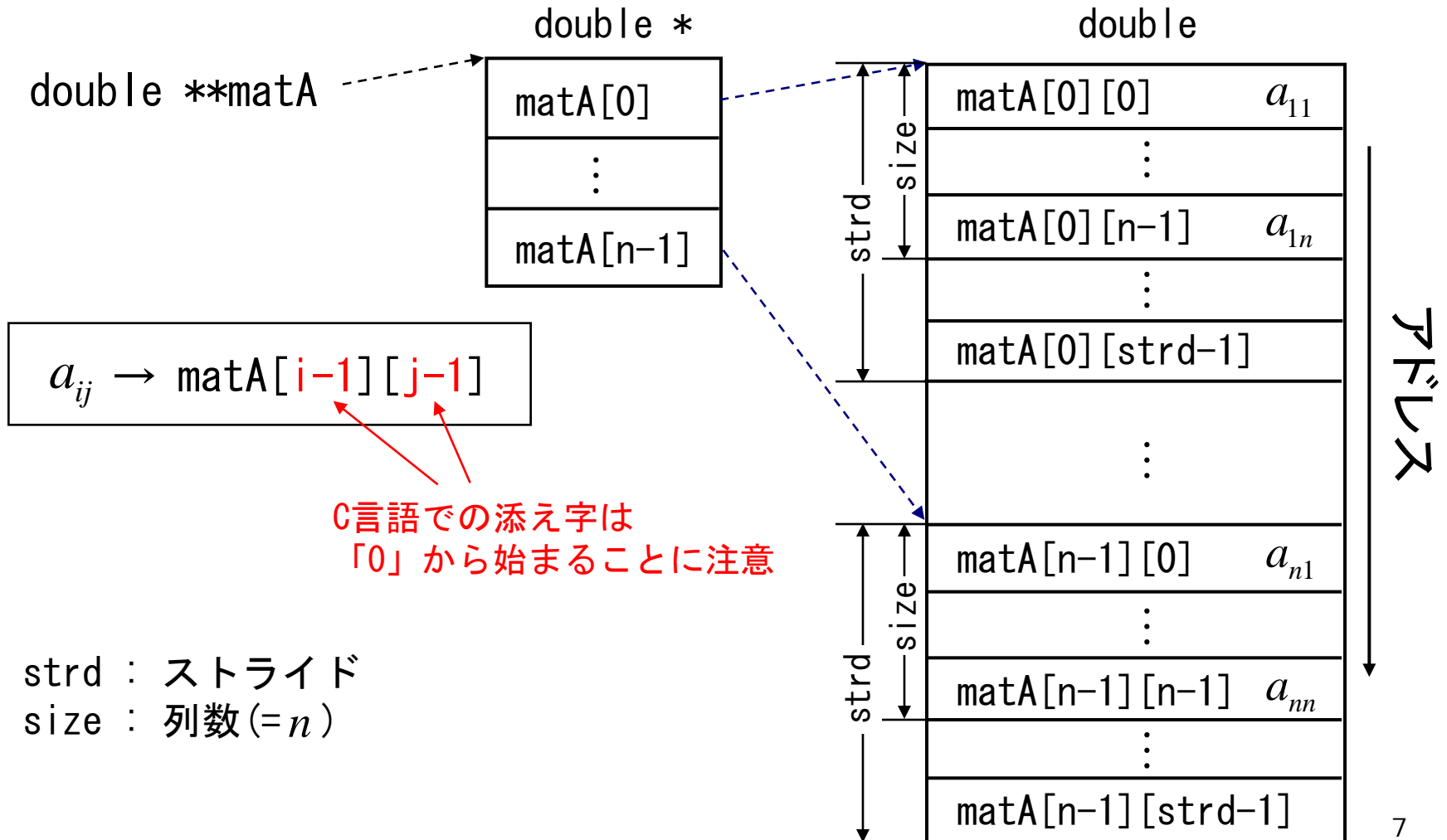
↑
行ベクトル \mathbf{a}_i と列ベクトル \mathbf{b}^j の内積



n^2 個の成分を計算するための演算回数 $2n^3$

行列データのアクセス(2次元配列)

配列データのメモリ上の配置



行列データのアクセス(1次元配列)

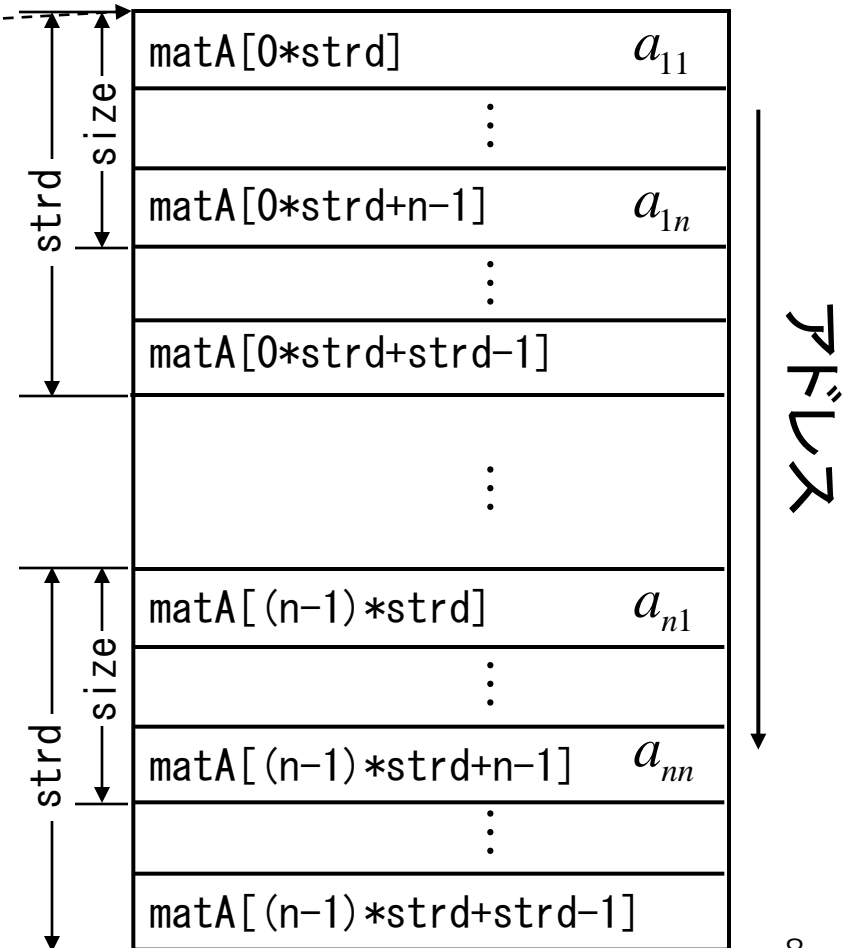
配列データのメモリ上の配置

double

double *matA

$$a_{ij} \rightarrow \text{matA}[(i-1)*\text{strd}+j-1]$$

strd : ストライド
size : 列数 (= n)



行列積のプログラム (2次元配列)

以下のように置き換えて考えると

$$a_{ik} \rightarrow \text{matA}[i-1][k-1]$$

$$b_{kj} \rightarrow \text{matB}[k-1][j-1] \quad (i, j, k=1, \dots, \text{size})$$

$$c_{ij} \rightarrow \text{matC}[i-1][j-1]$$

```
for (i=0; i<size; i++)  
  for (j=0; j<size; j++)  
    matC[i][j]=0.0;
```

← $\mathbf{C} = \mathbf{0}$

*i, j, kのループは
0から始まっていることに注意*

```
for (i=0; i<size; i++)  
  for (j=0; j<size; j++)  
    for (k=0; k<size; k++)  
      matC[i][j] +=  
        matA[i][k]*matB[k][j];
```

← $c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$

行列積のプログラム (1次元配列)

1次元配列でのアクセス

```
matA[i][k] → matA[i*strd+k]
matB[k][j] → matB[k*strd+j]
matC[i][j] → matC[i*strd+j]
```

前頁のプログラムは

```
for (i=0; i<size; i++)
    for (j=0; j<size; j++)
        matC[i*strd+j]=0.0;
```

```
for (i=0; i<size; i++)
    for (j=0; j<size; j++)
        for (k=0; k<size; k++)
            matC[i*strd+j] += matA[i*strd+k]*matB[k*strd+j];
```

単体CPUで動くプログラム(1)

- 二次元配列表現の行列積計算をCで書く。
- (i, j) 要素の参照方法 `matA[i][j]` に注意。

```
/* matMul2d.c: Matrix Multiplication in 2d representation of matrices */  
  
/*  
 * Calculate C = A * B  
 * A, B, C: 2d-array representation of matrices (size x strd).  
 * Thus, the (i,j) element is accessed by A[i][j], etc.  
 */  
void matMul(double **matA, double **matB, double **matC, int size, int strd)  
{  
    int i, j, k;  
  
    /* Initialize */  
    for (i = 0; i < size; i++)  
        for (j = 0; j < size; j++)  
            matC[i][j] = 0.0;  
  
    /* Matrix Multiplication */  
    for (i = 0; i < size; i++)  
        for (j = 0; j < size; j++)  
            for (k = 0; k < size; k++)  
                matC[i][j] += matA[i][k] * matB[k][j];  
}
```

関数の引き数として渡す時の
二次元配列の作り方は、
`matAlloc.c` の `matAlloc()` の方法を参照

単体CPUで動くプログラム(2)

- 一次元配列表現の行列積計算をCで書く。
- (i, j) 要素の参照方法 `matA[i*strd+j]` に注意。

```
/* matMul.c: Matrix Multiplication */  
  
/*  
 * Calculate C = A * B  
 * A, B, C: 1d-array representation of matrices (size x strd)  
 * Thus, the (i,j) element is accessed by A[i * strd + j], etc.  
 */  
void matMul(double *matA, double *matB, double *matC, int size, int strd)  
{  
    int i, j, k;  
  
    /* Initialize */  
    for (i = 0; i < size; i++)  
        for (j = 0; j < size; j++)  
            matC[i * strd + j] = 0.0;  
  
    /* Matrix Multiplication */  
    for (i = 0; i < size; i++)  
        for (j = 0; j < size; j++)  
            for (k = 0; k < size; k++)  
                matC[i * strd + j] += matA[i * strd + k] * matB[k * strd + j];  
}
```

例題プログラムによる性能評価(1)

- プログラム名: tm (一次元表現)、tm2d (二次元表現)
 - さまざまなサイズの行列積計算で、数値演算の実効性能を測定する。
- プログラムの構造
 - tm.c, tm2d: 初期設定、時間計測、結果検証
 - matMul.c, matMul2d.c: 行列計算(高速化の対象)
 - second.c: 経過時間(or CPU時間)計測関数
 - matAlloc.c: 2次元配列のメモリ確保、開放の関数
- コンパイル方法
 - `cp /tmp/matrix.tar.gz .`
 - `tar xvfz matrix.tar.gz`
 - `cd matrix/serial`
 - `make`
 - 詳しくは Makefile を参照

例題プログラムによる性能評価(2)

■ 実行方法、結果の取得方法

- 基本的にバッチ処理で実行する
- `qsub run-s.sh`
- 実行が終わると、`run-s.sh.o????` (????はジョブの番号) というファイルに結果が出力される

■ 行列サイズの種類

- `tm.c`

```
int main(int argc, char **argv)
{
    int size[] = { 16, 32, 64, 128, 256, 512, 1024, 2048 };
    int i, j, nitera, n, Nsize = 1, sz, it, strd;
    double *matA, *matB, *matC;
```

Nsizeは行列積を計算する行列サイズの種類、Nsize=8まで増やせるが、時間がかかるので、最初はNsize=1か2で試して、その後増やした方がよい

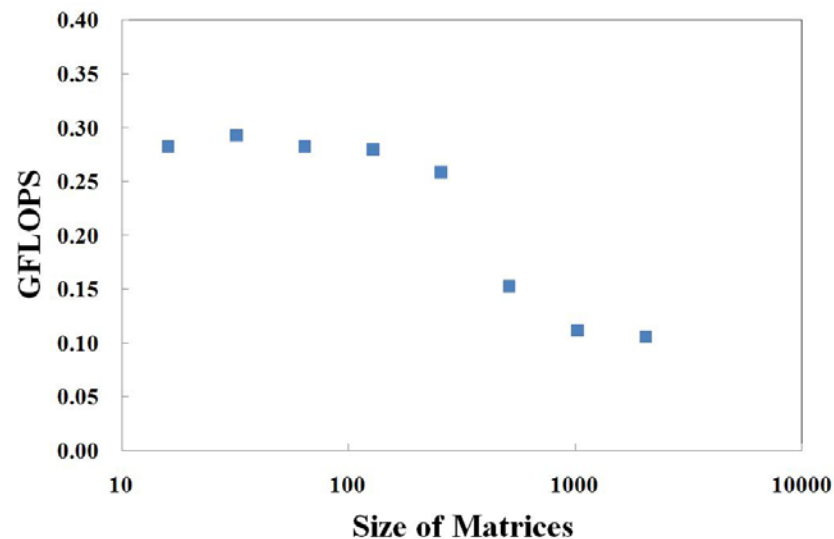
例題プログラムによる性能評価(3)

■ 実行結果の例、および、各項目の意味

- 行列のサイズ、計測した時間、行列積の計算回数、実効性能(概数)、数値誤差

CPU time is measured.
The time resolution is 0.0009998480 (sec).
Minimum duration for a measurement is 10 (sec).

size	time(sec)	n Calc	G-FLOPS	ERROR
16	15.18869	524288	0.283	2.04241e-15
32	14.65277	65536	0.293	4.16353e-15
64	15.16170	8192	0.283	1.30492e-14
128	15.35267	1024	0.280	2.73106e-14
256	16.55948	128	0.259	8.79056e-14
512	14.06586	8	0.153	2.56351e-13
1024	19.22008	1	0.112	5.84971e-13
2048	162.19834	1	0.106	1.39013e-12



上の表は、一次元表現での計測プログラム(tm)をFujitsu PRIMERGY
単体CPUで実行した結果。右のグラフは、このうちGFLOPSの数値をプロットしたもの。

性能向上のためのヒント

- コンパイラにもう少しがんばってもらうには？
 - コンパイルオプションの調整による高速化
 - man などにより、情報取得
- 行列積の計算部分は、三重ループになっている
 - ループの順序を変えると、性能に違いが出るか？
- 無駄な計算を省くには？
 - 一次元表現の場合は、プログラム中でインデックス計算 ($i * \text{strd} + j$) を行っているが、...
- 行列の要素の並べ方を変えると？
 - 例えば `matB` の転置行列を作ると、要素のアクセス順序がどう変わるか？
- BLAS、LAPACKなどの数値演算ライブラリを利用する。
 - 時には、データの格納方法などを変更する必要が生ずる。

例題プログラムの行列データ(1)

例題のプログラムでは

$$\mathbf{A} = (a_{ij}) = \left(\sqrt{\frac{2}{n}} \sin \left[\frac{(i-j)\pi}{n} \right] \right)$$

$$\mathbf{B} = (b_{ij}) = \left(\sqrt{\frac{2}{n}} \cos \left[\frac{(i-j)\pi}{n} \right] \right)$$

$$\begin{aligned} c_{ij} &= \sum_{k=1}^n a_{ik} b_{kj} \\ &= \frac{2}{n} \sum_{k=1}^n \sin \left[\frac{(i-k)\pi}{n} \right] \cos \left[\frac{(k-j)\pi}{n} \right] \end{aligned}$$

例題プログラムの行列データ(2)

三角関数の和の公式

$$\sin \alpha \cos \beta = \frac{1}{2} \sin (\alpha + \beta) + \frac{1}{2} \sin (\alpha - \beta)$$

を使うと

$$\begin{aligned} & \sin \left[\frac{(i-k)\pi}{n} \right] \cos \left[\frac{(k-j)\pi}{n} \right] \\ &= \frac{1}{2} \sin \left[\frac{(i-j)\pi}{n} \right] + \frac{1}{2} \sin \left[\frac{(i+j-2k)\pi}{n} \right] \end{aligned}$$

↑
第二項は k についての和でキャンセル

例題プログラムの行列データ(3)

解析的な答え \tilde{c}_{ij} は

$$\tilde{c}_{ij} = \sin \left[\frac{(i-j)\pi}{n} \right]$$

c_{ij} と \tilde{c}_{ij} の差の自乗の和の平方根

$$R = \sqrt{\sum_{i,j=1}^n |c_{ij} - \tilde{c}_{ij}|^2}$$

数値計算の結果 c_{ij} のチェックとして
プログラム内で R の値を表示



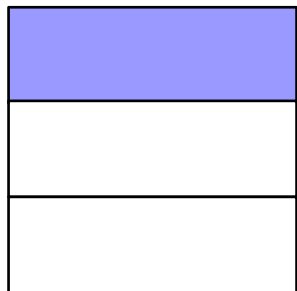
MPIを用いた行列積の並列化

MPIによる並列化(1)－行列積の分割－

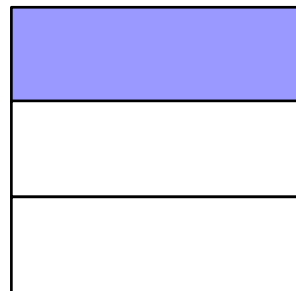
C

A

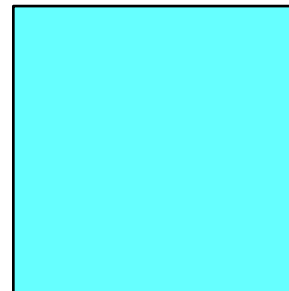
B



=



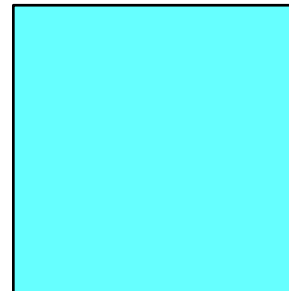
×



=



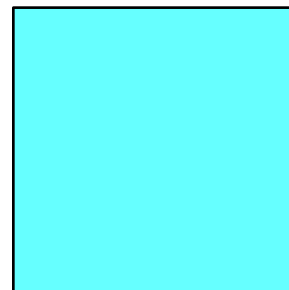
×



=



×



$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

行方向に分割すると、Bは全部必要だが、Aは一部しか必要ない

MPIによる並列化(2)

nprocs個の並列プロセスが
それぞれ rows = $n/nprocs$ 行ずつ計算

$r = \text{rows}$
 $p = \text{nprocs}$

$$\begin{array}{c}
 \text{rank } 0 \\
 \downarrow \text{rows} \\
 \left(\begin{array}{ccc} a_{11} & \cdots & a_{1n} \\ \vdots & \vdots & \vdots \\ a_{r1} & \cdots & a_{rn} \end{array} \right) \\
 \hline
 \vdots \\
 \hline
 \text{rank } p-1 \\
 \downarrow \text{rows} \\
 \left(\begin{array}{ccc} a_{r*(p-1)+11} & \cdots & a_{r*(p-1)+1n} \\ \vdots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{array} \right)
 \end{array}
 \begin{array}{c}
 \mathbf{B} \\
 \left(\begin{array}{ccc} b_{11} & \cdots & b_{1n} \\ \vdots & \vdots & \vdots \\ b_{r1} & \cdots & b_{rn} \end{array} \right)
 \end{array}
 =
 \begin{array}{c}
 \mathbf{C} \\
 \left(\begin{array}{ccc} c_{11} & \cdots & c_{1n} \\ \vdots & \vdots & \vdots \\ c_{r1} & \cdots & c_{rn} \end{array} \right) \\
 \hline
 \vdots \\
 \hline
 \left(\begin{array}{ccc} c_{r*(p-1)+11} & \cdots & c_{r*(p-1)+1n} \\ \vdots & \vdots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{array} \right)
 \end{array}$$

各プロセスは行列積計算のデータとして

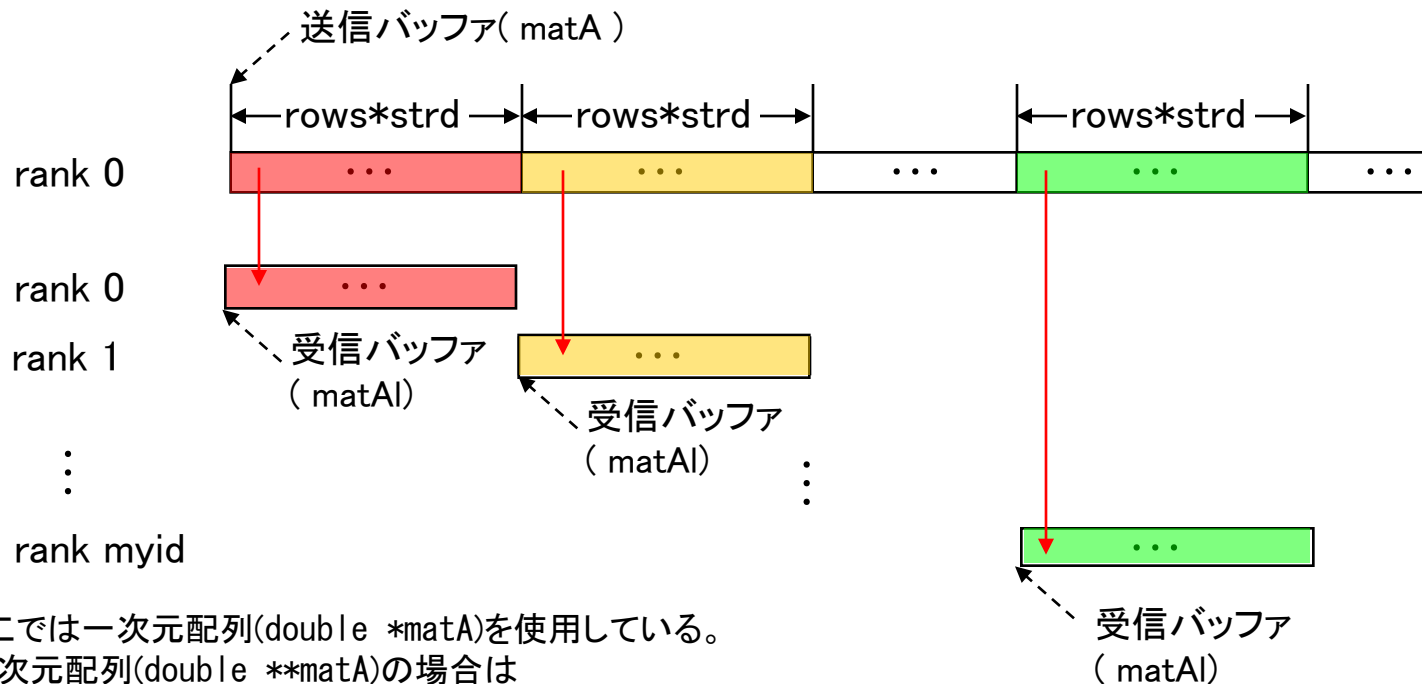
Aは一部(計算を担当する部分の行)、**B**は全体が必要

MPIによる並列化(3)

並列版例題プログラムの計算手順

1. **A**、**B** の行列要素はrank 0のプロセスで計算
2. rank 0のプロセスから各プロセスへ
 - A**の必要な行を送信(MPI_Scatter)
 - B**の全ての行を送信(MPI_Bcast)
3. 各プロセスで担当部分の行列積を計算
4. 各プロセスからrank 0のプロセスへ計算結果を集める(MPI_Gather)

行列Aの分配(MPI_Scatter)

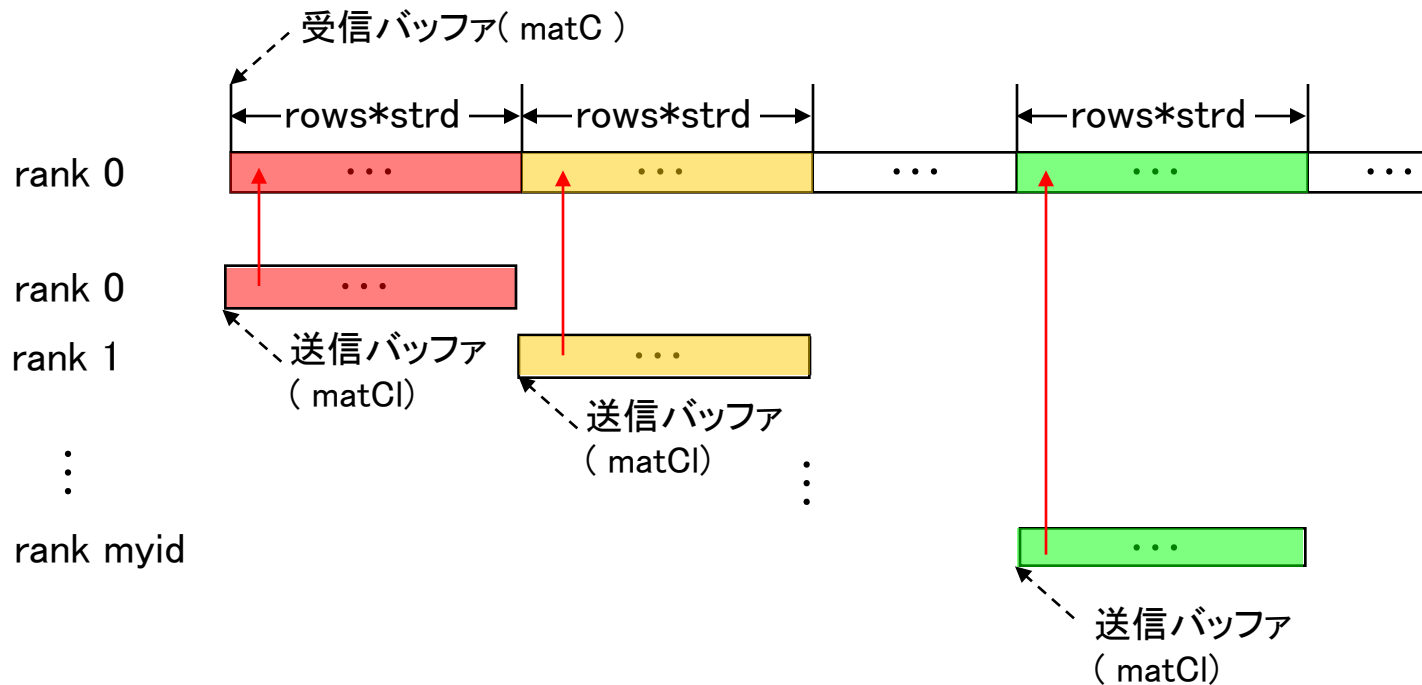


ここでは一次元配列(double *matA)を使用している。
二次元配列(double **matA)の場合は
&matA[0][0]に置き換える

```
MPI_Scatter(matA,  
            rows * strd,  
            MPI_DOUBLE,  
            matAI,  
            rows * strd,  
            MPI_DOUBLE,  
            0,  
            MPI_COMM_WORLD);
```

送信バッファのアドレス
各プロセスに送信するデータの個数
送信データのタイプ(doubleのデータ)
受信バッファのアドレス
各プロセスが受信するデータの個数
受信データのタイプ(送信と同じ)
送信元プロセスのランク
全並列プロセスを含むコミュニケータ

行列Cの収集(MPI_Gather)



```

MPI_Gather (matC,
           rows * strd,
           MPI_DOUBLE,
           matC,
           rows * strd,
           MPI_DOUBLE,
           0,
           MPI_COMM_WORLD);
    
```

送信バッファのアドレス
 各プロセスが送信するデータの個数
 送信データのタイプ(doubleのデータ)
 受信バッファのアドレス
 各プロセスから受信するデータの個数
 受信データのタイプ(送信と同じ)
 受信プロセスのランク
 全並列プロセスを含むコミュニケーター

MPI版の行列積プログラム(1)

■ 二次元配列表現のプログラム: matMul2d-mpi.c

```
/* matMul2d.c: Matrix Multiplication in 2d representation of matrices */  
  
#include <mpi.h>  
  
/*  
 * Calculate C = A * B  
 *   A, B, C: 2d-array representation of matrices (size x strd).  
 *   Thus, the (i,j) element is accessed by A[i][j], etc.  
 */  
void matMul(double **matA, double **matB, double **matC, double **matA1,  
            double **matC1, int size, int strd, int myid, int nprocs)  
{  
    int i, j, k;  
    int rows;  
  
    rows = size / nprocs;  
    MPI_Scatter(& matA[0][0], rows * strd, MPI_DOUBLE,  
              & matA1[0][0], rows * strd, MPI_DOUBLE,  
              0, MPI_COMM_WORLD);  
    MPI_Bcast(& matB[0][0], size * strd, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
    for (i = 0; i < rows; i++)  
        for (j = 0; j < size; j++)  
            matC1[i][j] = 0.0;  
    for (i = 0; i < rows; i++)  
        for (j = 0; j < size; j++)  
            for (k = 0; k < size; k++)  
                matC1[i][j] += matA1[i][k] * matB[k][j];  
    MPI_Gather(& matC1[0][0], rows * strd, MPI_DOUBLE,  
             & matC[0][0], rows * strd, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
}
```

ただし、このプログラムでは、行列の先頭から、一次元的にデータが並んでいることが仮定されている。

詳しくは、matAlloc.c 中の matAlloc() を参照。

MPI版の行列積プログラム(2)

■ 一次元配列表現のプログラム: matMul-mpi.c

```
/* matMul_mpi.c: Matrix Multiplication written with MPI routines */

#include <mpi.h>

/*
 * Calculate C = A * B
 *   A, B, C: 1d-array representation of matrices (size x strd)
 *   Thus, the (i,j) element is accessed by A[i * strd + j], etc.
 */
void matMul(double *matA, double *matB, double *matC, double *matA1,
            double *matC1, int size, int strd, int myid, int nprocs)
{
    int i, j, k;
    int rows;

    rows = size / nprocs;
    MPI_Scatter(matA, rows * strd, MPI_DOUBLE,
               matA1, rows * strd, MPI_DOUBLE,
               0, MPI_COMM_WORLD);
    MPI_Bcast(matB, size * strd, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for (i = 0; i < rows; i++)
        for (j = 0; j < size; j++)
            matC1[i * strd + j] = 0.0;
    for (i = 0; i < rows; i++)
        for (j = 0; j < size; j++)
            for (k = 0; k < size; k++)
                matC1[i * strd + j] += matA1[i * strd + k] * matB[k * strd + j];
    MPI_Gather(matC1, rows * strd, MPI_DOUBLE,
              matC, rows * strd, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

ここでは、集団通信だけを使って、並列化してみる。

行列Aはブロック分割してscatter、行列Bは全体をbroadcast、そして、結果はブロック分割された内容をgatherする。

例題プログラムによる性能評価(1)

- プログラム名: tm-mpi、tm2d-mpi
 - さまざまなサイズの行列積計算で、MPIによる並列化性能を測定
- プログラムの構造
 - tm-mpi.c, tm2d-mpi.c: 初期設定、時間計測、結果検証
 - matMul-mpi.c, matMul2d-mpi: 行列計算(高速化の対象)
- コンパイル方法(例)
 - `cp /tmp/matrix.tar.gz .`
 - `tar xvfz matrix.tar.gz`
 - `cd matrix/mpi`
 - `make`
 - 詳しくは `Makefile` を参照

例題プログラムによる性能評価(2)

■ 実行方法、結果の取得方法

□ バッチ処理で実行する

□ 1プロセス

■ `cp /tmp/run-01.sh .`

■ `qsub run-01.sh`

□ 2プロセス

■ `cp /tmp/run-02.sh .`

■ `qsub run-02.sh`

(途中略)

□ 64プロセス

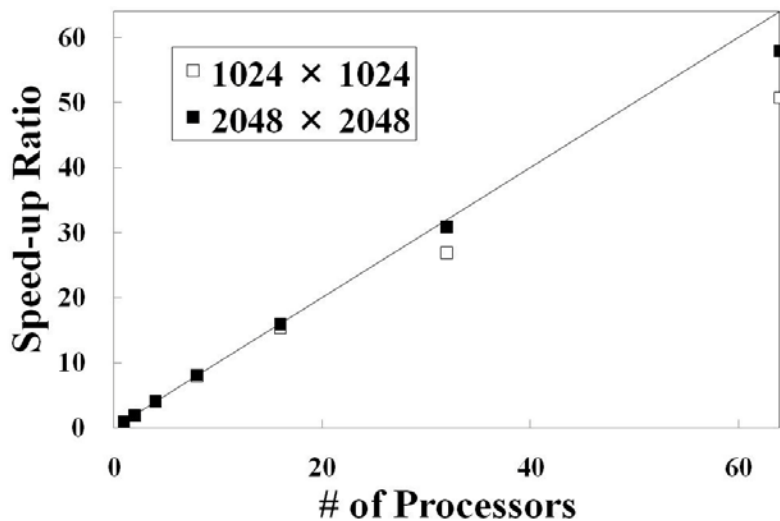
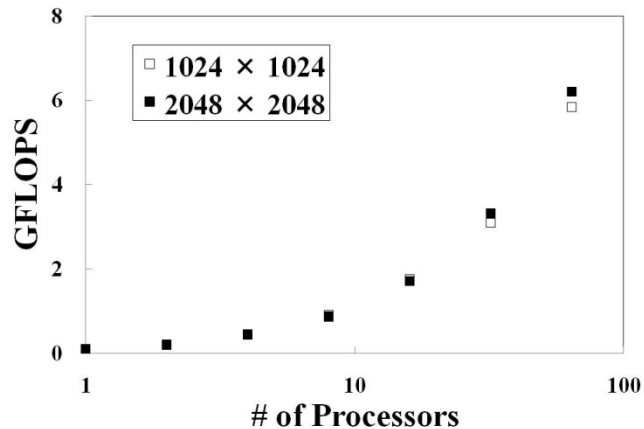
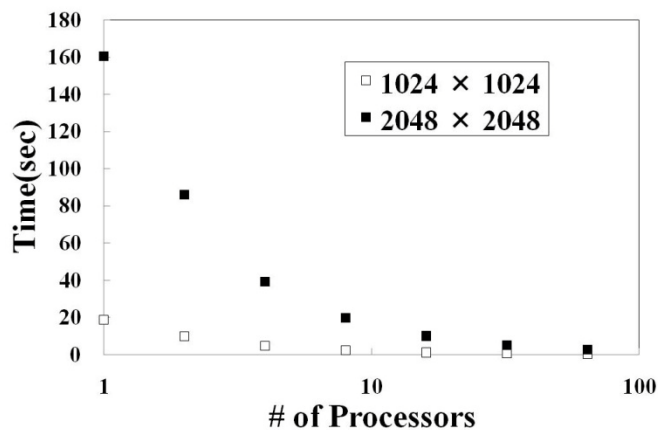
■ `cp /tmp/run-64.sh .`

■ `qsub run-64.sh`

□ 実行が終わると、`run-xx.sh.o????(????はジョブの番号)`というファイルに結果が出力される

例題プログラムによる性能評価(3)

- 実行結果の意味は単体CPU版のものと同じ
- 並列化の効果をグラフで見ると...



上の図は、一次元表現での計測プログラム (tm_mpi) を Fujitsu Primergy で実行した結果。

並列性能向上のためのヒント

- 単体CPUでの高速化手法のうち、すべてが並列プログラムでも効果的とは限らないが、試してみる価値は十分にある。
- 並列性能向上のための基本は、通信量の削減と負荷分散。しかし、行列積の場合、この部分でさらに効果を上げるのは難しい。
- 各CPUで実行される問題サイズが、ちょうどキャッシュを利用できる大きさの場合、単純に並列数倍 (linear speed-up) よりも性能が向上する場合がある。
- 分割方法とアルゴリズムを変えてreduce演算を利用することで、高速化ができる場合がある。