



# 並列アルゴリズム

---

2005年後期 火曜 2限

青柳 睦

[Aoyagi@cc.kyushu-u.ac.jp](mailto:Aoyagi@cc.kyushu-u.ac.jp)

<http://server-500.cc.kyushu-u.ac.jp/>

10月18(火)

4. 数値計算における各種の並列化
5. MPI の基礎

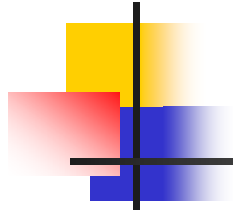


# 講義の概要

---

並列計算機や計算機クラスターなどの  
分散環境における並列処理の概論

- MPIおよびOpenMPによる並列計算
- 理工学分野の並列計算アルゴリズム



# 成績評価

---

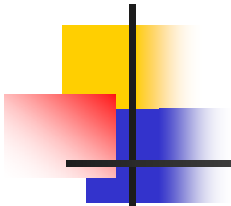
出席点5割, レポート5割

aoyagi@cc.kyushu-u.ac.jpへメール

Subject: 並列アルゴリズム

学籍番号, 氏名, 専攻,

座席番号 (A-1, A-2, ... C-3など)



## 3. 並列処理の課題(続き)

---

### 計算負荷のアンバランス

- 同期待ち時間の増大

### 通信のオーバーヘッド

- プロセッサ間通信の「頻度」と計算粒度

### 並列アルゴリズムによるオーバーヘッド

- 並列化の為にデータをコピーする
- 並列化の為に制御文を挿入



# MPIプログラムの例

```
/* greetings.c -- greetings program */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int    my_rank;    /* rank of process */
    int    p;         /* number of processes */
    int    source;    /* rank of sender */
    int    dest;      /* rank of receiver */
    int    tag = 0;   /* tag for messages */
    char    message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

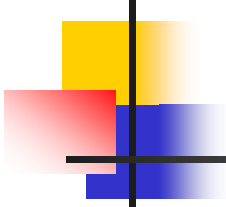
    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR,
            dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```



## 4. 数値計算における各種の並列化

### 数値計算コアとなるSolverによる分類

数値積分, 連立一次方程式, 固有値問題, ..

概して計算粒度が小さなアルゴリズムが多い

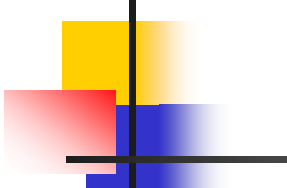
### シミュレーション手法による分類

差分法, 有限要素法, 粒子多体問題, ..

自然を模倣したアルゴリズム等..

シミュレーテッドアニーリング, 遺伝的アルゴリズム,  
ニューラルネットワーク, ..

計算粒度の大きなアルゴリズムが多い



## 4.1 並列性による分類

### 自明な並列

最初にデータと仕事をばらまいて、最後に結果を回収・整理統合するタイプ。もともと粒度が大きな並列化が実現しやすい。モンテカルロ法, パラメータサーベイ, 探索問題, 等。ただし, データのばらまき・回収に通信量が多すぎたり, データが1プロセッサのメモリに入りきらない場合には, 別の並列性を考慮する必要がある。

### タスク並列


内容の異なる複数のタスク(処理のまとめり)からなり, それらを一定の順序制約のもとに処理するタイプ。探索問題, 分割統治法, 疎行列計算など。中粒度から大粒度。

### データ並列

沢山の(別の)データのそれぞれに対して(同一の)処理を行うタイプ。密行列・ベクトルなど線形演算, 物理シミュレーション, 画像処理など。「個々のデータに対する処理に, (近傍か遠方か又は部分データか全体データか)どこのデータが必要となるか」が効率の良い並列アルゴリズム生成のカギとなる。中粒度～小粒度。

### パイプライン型

データの列があり, データ要素に対して複数の処理を逐次的に施すタイプ。いわゆる流れ作業。多倍長演算や動的計画法など。



## 4.2 分散メモリにおける並列プログラミング

### SPMD(Single Program Multiple Data )

並列プログラミングの一つのスタイルで、複数の計算機上に個々別々のデータが乗っているが、それを処理するプログラムはすべて同一のものを使う。しかし、プログラムが同一だからといって処理内容が同一であるとは限らない。条件分岐を用いて、プロセッサ毎に全く異なる処理を行うプログラムを SPMD の枠内で記述することもできる(…というか、通常そうする)。

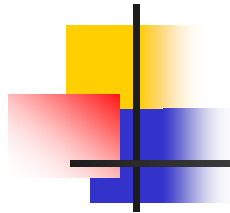
### Owner Computes Rule

分散メモリにおける並列プログラミングの3つのKEYポイントは、データの分散、処理(タスク)の分散、そして通信方法と通信量(頻度も重要)である。3つを同時に考えながらプログラミングを行うことは難しい。まずはデータの分散を考え、処理(タスク)の分散をデータの分散に従わせることをOwner Computes Ruleと呼ぶ。通信は最適になるとは限らないが、

### 同期通信

同期通信では、データを持っているプロセスは送信 (send) というルーチン呼び、それを受け取るプロセスは受信 (recv) というルーチンと呼ぶ。このタイプの通信では、データのやり取りと同期を兼ねており、受信ルーチン呼び出すまでは相手のメッセージは自分の領域に入っていないことが保証される点でプログラムの構造は比較的簡単な場合が多いが、通信は各時刻で必ず一方通行 (half duplex) となる。





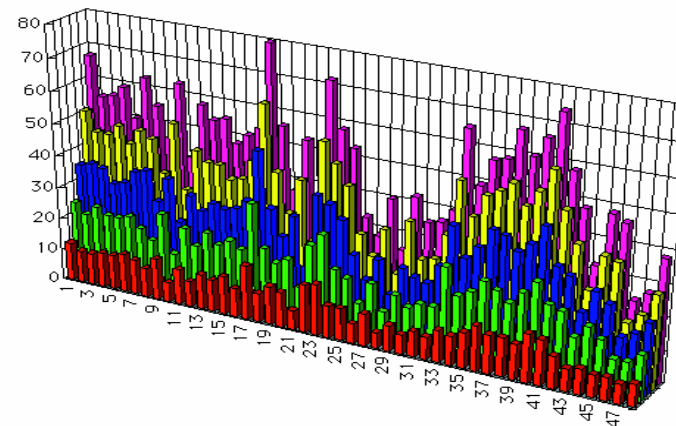
## 非同期通信

非同期通信では、受け手が受信ルーチン呼び出ししていなくても送り手はデータを送信することができるので、送ってしまったら次の処理を開始し、演算と通信が同時に行われる様にアルゴリズムを工夫できる場合には「通信遅延の隠蔽」ができる。

非同期通信では(うまくゆけば)双方向(full duplex)の通信が可能である。また動的負荷分散を考慮したMaster-Workerモデルなど、柔軟なプログラミングが可能だが、多用するとプログラムの構造が複雑になる。一般に、非同期通信は同期通信よりも多くのバッファメモリを必要とする。

(例) 自作の電子構造計算プログラムにおける2電子積分ルーチンの負荷分散

Master-Workerタイプのアルゴリズムを用いており、各計算ノード(横軸)でIteration (奥行に表示)毎、「積分計算の処理量」(縦軸)が異なる。





## 4.3 並列プログラミング環境

### MPI (Message Passing Interface)

分散メモリプログラミングのためのライブラリルーチンのインタフェースを定義.

MPI Forum(<http://www.mpi-forum.org/>)により策定 (MPI-2.0, MPI-1.2, ..)

実装としては, MPICH(<http://www-unix.mcs.anl.gov/mpi/mpich/>), LAMなどオープンソース&フリーなものから, ベンダー提供のライブラリなど各種ある.

事実上の世界標準. 共有メモリマシンでも利用できる.

### OpenMP

共有メモリプログラミングのための指示文のセット

逐次プログラムをベースとし, Pragma指示行という形で並列処理の仕方を定義.

アルゴリズムが特定のパターンにマッチすれば簡単に並列化ができる.

### HPF (High Performance Fortran)

逐次版 fortranプログラムに指示行を加えて, データと処理の分散を指示.

必要な通信はコンパイラにより自動的に生成される. 分散メモリ計算機上で, 共有メモリのなプログラミングを行うことを意図. 最近は少し元気がない?



## 5.1 MPIの序 (history)

MPI (Message Passing Interface) は、メッセージ通信のプログラムを記述するために広く使われる「標準」を目指して作られた、メッセージ通信のAPI仕様である。MPIの標準化への取り組みはSupercomputing'92会議において、後にMPIフォーラムとして知られることになる委員会が結成され、メッセージ・パッシングの標準が作成された。これには主としてアメリカ、ヨーロッパの40の組織から約60人の人間が関わっており、産官学の研究者、主要な並列計算機ベンダのほとんどが参加した。そしてSupercomputing'93会議において草案MPI標準が示され、1994年にMPI-1がリリースされた。MPIの成功を受けて、MPIフォーラムはオリジナルのMPI標準文書の改定と拡張を検討し始めた。このMPI-2フォーラムにおける最初の成果物として、1995年6月にMPI1.1がリリースされた。1997年7月には、MPI1.1に対する追加訂正と説明がなされたMPI1.2と、MPI-1の機能の拡張を行ったMPI-2がリリースされた。MPI-2の仕様は基本的にMPI-1の改定ではなく、新たな機能の追加であるために、MPI-1で書かれたプログラムもMPI-2をサポートするプラットフォームで実行できる。

以上 (<http://www.mpi-forum.org/>)とMPI-1, MPI-2の日本語訳(MPI-Jプロジェクト)より抜粋。



# MPIの実装 MPICHとLAM

MPIはインターフェースの規定であり、実装パッケージそのものではない。MPICHは、アメリカのアルゴンヌ国立研究所(Argonne National Laboratory) が模範実装として開発し、無償でソースコードを配布したライブラリである。移植しやすさを重視した作りになっているため盛んに移植が行われ、LAM 同様、Linuxマシンは勿論、世界中のほとんどのベンダの並列マシン上で利用することができる。特に、MPICH ではUNIX 系に限らず Windows 系へのサポートも行われている。さらに、SMP, Myrinet などのハード面にも対応している上、バッチシステムDQS, グリッドツールキットGlobus といった様々なツールを使用できることも大きな特徴の一つである。また、MPICH 1.2.0 では、MPI-1.2 の全ての機能をカバーしており、MPI-2 に関しても幾つかの機能についてはサポートしている。MPICH 1.2.0 におけるMPI-2 への対応等に関する詳細な情報は、<http://www-unix.mcs.anl.gov/mpi/mpich/>に載せられている。

LAM(Local Area Multicomputer) は、ノートルダム大学の科学コンピュータ研究室(Laboratory for Scientific Computing, University of Notre Dame) が作成したフリーのMPI ライブラリである。LAMは、標準的なMPI API だけでなく幾つかのデバッキングとモニタリングツールをユーザに提供している。MPI-1 を完全にサポートしているだけでなくMPI-2 の標準的な幾つかの要素についても機能を提供している。最新バージョン7.0.x ではMPI-2 における1 方向通信、動的プロセスの管理に関する機能がカバーされている。また、LAM は世界中のほとんどのUNIX 系ベンダの並列マシン上で利用することができる。ただし、2003年秋時点でWindows に関してはサポートされていない。LAM は、並列ジョブ実行・デバッグ統合環境であるXMPI との相性が良くXMPI を使用したいのであれば便利である。LAM に関する詳細な情報は、<http://www.mpi.nd.edu/lam/>にある。



# MPIの紹介と並列アルゴリズム

## MPIプログラムの例

```
/* greetings.c -- greetings program */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int    my_rank;    /* rank of process */
    int    p;         /* number of processes */
    int    source;    /* rank of sender */
    int    dest;      /* rank of receiver */
    int    tag = 0;   /* tag for messages */
    char   message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

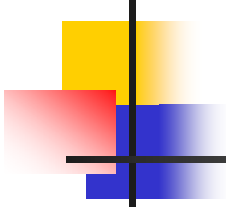
    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR,
            dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```



## コミュニケータ, rank, 利用可能プロセッサ数

**コミュニケータ** : お互いに通信を行うプロセスの集合である。ほとんどのMPIルーチンは引数としてコミュニケータを取る。変数MPI\_COMM\_WORLD は、あるアプリケーションを一緒に実行している全プロセスからなるグループを表しており、これは最初から用意されている。また新しいコミュニケータを作成することも可能である。

**rank** : コミュニケータ内の全てのプロセスは、プロセスが初期化されたときにシステムによって示されたID をもっている。これは0 から始まる連続した正数が割り当てられる。プログラマはこれを用いて、処理の分岐、あるいはメッセージの送信元や受信先を指定することができる。 `MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);`

**利用可能プロセッサ数** : コミュニケータ内で利用できるプロセッサ数(p)は、`MPI_Comm_size(MPI_COMM_WORLD, &p);` により取得できる。



## (補足) ジョブ, プロセス, スレッド

---

### ジョブ(Job)

コンピュータが処理する仕事の単位 .

### プロセス(Process)

アドレス空間を排他的に利用する計算処理の単位.

プロセス実行中の資源や情報は個別に管理, 生成や切り替えに時間がかかる.  
複数プロセスを並列計算に使うためには, プロセス間通信が必要.

### スレッド(Thread)

プロセスをさらに細分化した並行処理単位. 実行に必要な資源や情報の多くをスレッド間で共有できるため, スレッド固有の必要資源を少なくし, 操作負荷を軽減することができる.

## 5.1 MPI send と recv

`int MPI_Send( /* 機能：送信バッファのデータを特定の受信先に送信する。 */`

<code>void*</code>	<code>message</code>	<code>/* 送信バッファの開始アドレス(IN) */,</code>
<code>int</code>	<code>count</code>	<code>/* データの要素数(IN) */,</code>
<code>MPI_Datatype</code>	<code>datatype</code>	<code>/* データタイプ(IN) */,</code>
<code>int</code>	<code>dest</code>	<code>/* 送信先(IN) */,</code>
<code>int</code>	<code>tag</code>	<code>/* メッセージ・タグ(IN) */,</code>
<code>MPI_Comm</code>	<code>comm</code>	<code>/* コミュニケータ(IN) */ )</code>

`int MPI_Recv( /* 機能：要求されたデータを受信バッファから取り出す。  
またそれが可能になるまで待つ。 */`

<code>void*</code>	<code>message</code>	<code>/* 受信バッファの開始アドレス(OUT) */,</code>
<code>int</code>	<code>count</code>	<code>/* データの要素数(IN) */,</code>
<code>MPI_Datatype</code>	<code>datatype</code>	<code>/* データタイプ(IN) */,</code>
<code>int</code>	<code>source</code>	<code>/* 送信元(IN) */,</code>
<code>int</code>	<code>tag</code>	<code>/* メッセージ・タグ(IN) */,</code>
<code>MPI_Comm</code>	<code>comm</code>	<code>/* コミュニケータ(IN) */,</code>
<code>MPI_Status*</code>	<code>status</code>	<code>/* ステータス(OUT) */ )</code>



## 定義済みMPIデータ型

MPIデータ型	Cデータ型
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

定義済みMPIデータ型:ポータビリティを高めるために, MPI によって前もって定義されたデータ型である. 例えば, int型であればMPI\_INT というハンドルを用いる. BYTE とPACKED以外はC言語の型と対応している. プログラマが, 新たなデータ型を定義することも可能である.

## 5.2 プログラム例 積分の台形公式

端点と刻幅(h)

$$a = x_0, b = x_n, h = (b - a) / n$$

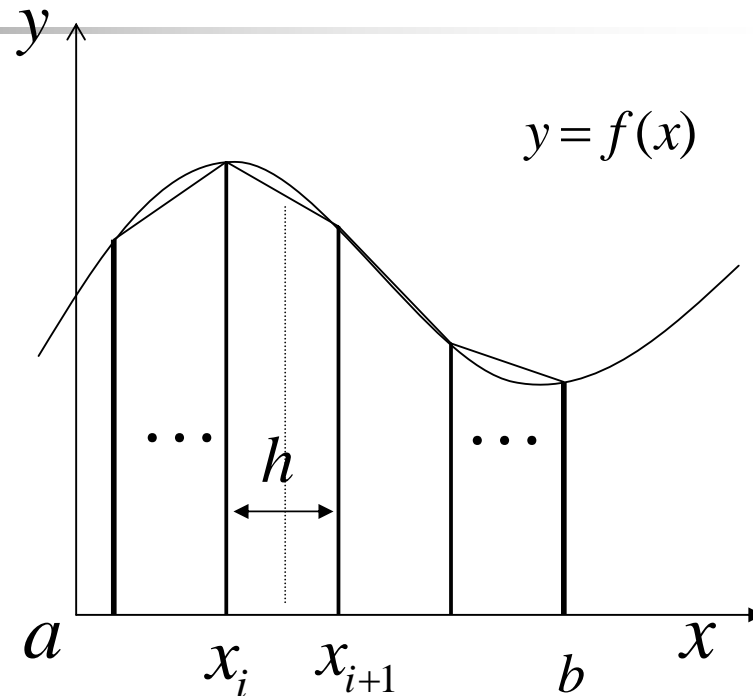
$$x_i = a + ih \quad (i = 0, 1, 2, \dots, n)$$

積分をn個の台形面積の和で近似

$$\int_a^b f(x) dx \approx \sum_{i=0}^{n-1} \frac{1}{2} h [f(x_i) + f(x_{i+1})]$$

$$= \sum_{i=0}^{n-1} h f\left(x_i + \frac{h}{2}\right) \text{ または素直に,}$$

$$= \left[ \frac{f(x_0)}{2} + \frac{f(x_n)}{2} + f(x_1) + f(x_2) + \dots + f(x_{n-1}) \right] h$$



(図) 非負関数の定積分と台形公式



【例題】

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

を用いて,  $\pi$  を数値計算で求める

【証明】

$$\int_0^1 \frac{4}{1+x^2} dx \quad \text{変数変換 } x = \tan \theta \text{ により,}$$

$$dx = \frac{d\theta}{\cos^2 \theta}, [0, 1] \rightarrow [0, \pi/4], \text{ だから}$$

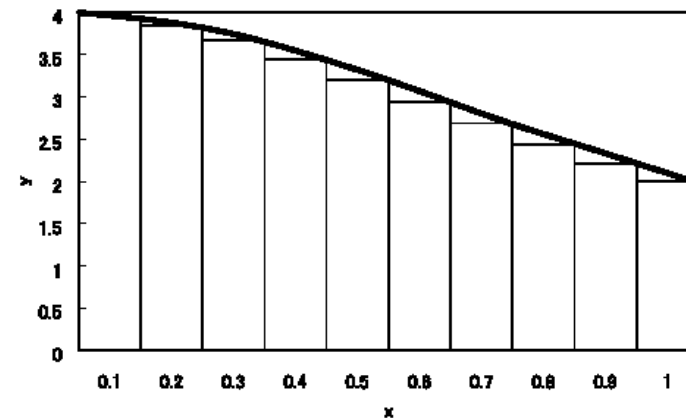
$$\int_0^{\pi/4} \frac{4}{1+\tan^2 \theta} \frac{d\theta}{\cos^2 \theta} = 4 \int_0^{\pi/4} d\theta = \pi$$

## 台形公式を用いた $\pi$ の計算 逐次プログラム(例)

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int i,loop;
    double width,x,pai=0.0;
    loop = atoi(argv[1]);
    width = 1.0 / loop;

    for(i=0; i<loop; i++){
        x = (i + 0.5) * width;
        pai += 4.0 / (1.0 + x * x);
    }
    pai = pai * width;
    printf("PAI = %f¥n",pai);
    return 0;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



## 5.3 MPIを用いた並列化 台形公式を用いた $\pi$ の計算

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>

double PI25DT =
3.141592653589793238462643;

int main( int argc, char *argv[])
{
    int i, loop;
    int numprocs, my_rank;
    double pai, width, sum, x;
    int source, dest , tag = 0;
    int local_loop;

    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,
        &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,
        &my_rank);
```

```
    if (my_rank == 0) {
        loop=atoi(argv[1]);
        for (dest = 1; dest < numprocs; dest++) {
            MPI_Send(&loop, 1, MPI_INT, dest, tag,
                MPI_COMM_WORLD);
        }
    } else {
        source=0;
        MPI_Recv(&loop, 1, MPI_INT, source, tag,
            MPI_COMM_WORLD, &status);
    }

    width = 1.0 / loop;
    local_loop = loop / numprocs;

    sum = 0.0;
    for (i = my_rank*local_loop;
        i < (my_rank+1)*local_loop; i++) {
        x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x*x);
    }
```



## MPIを用いた並列化 台形公式を用いた $\pi$ の計算

```
tag=1;
if (my_rank == 0) {
    pai=sum;
    for (source = 1; source < numprocs; source++) {
        MPI_Recv(&sum, 1, MPI_DOUBLE, source,
            tag, MPI_COMM_WORLD, &status);
        pai=pai+sum;
    }
} else {
    dest=0;
    MPI_Send(&sum, 1, MPI_DOUBLE, dest, tag,
        MPI_COMM_WORLD);
}

pai = pai * width;

if (my_rank == 0) {
    printf("PAI =%.16f¥n",pai);
    printf("Error = %.16f¥n", fabs(pai - PI25DT));
}

MPI_Finalize();
return 0;
}
```



## 5.4 並列化の効果

---

- 何故並列化するか？
  - 高速に実行したい！
    - メモリをたくさん使いたいという理由で並列化する場合もある。
- 本当に速くなったか？
  - 計測してみないとわからない。 ⇒ 6. 性能評価



## 5.5 MPIプログラムの実行

---

MPIプログラムの翻訳方法、実行方法は計算機によって異なる。

MPICH の場合:

MPIプログラム(C言語) の翻訳コマンド `mpicc`

例)

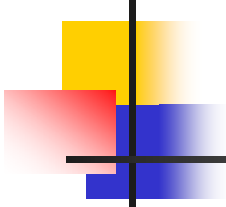
```
% mpicc test.c -o test
```

MPIプログラムの実行コマンド `mpirun`

例)

```
% mpirun -np 8 ./test
```





## 並列プログラムの実行時間

---

プログラムの評価に用いる時間は二通り

- CPU使用時間: CPUが働いた時間.
- 経過時間: 計算機の動作にかかわらず, 消費した時間.

計算が主体のプログラムでは,

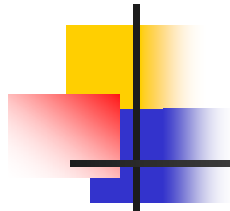
**CPU使用時間  $\simeq$  経過時間**

だが, CPU以外の装置(ディスク, ネットワーク等)を使用している時間が長いプログラムでは, その間CPUは待機するので

**CPU使用時間  $<$  経過時間**

となる.

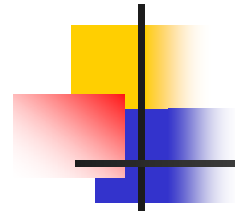
並列プログラムは通信時間による影響も評価の対象となるので, 通常は経過時間を用いる.



## 実行環境 PCクラスタ

プロセッサ	Intel Itanium2 900MHz
メモリ	512MB
OS	RedHat Linux Advanced Server
コンパイラ	Intel C compiler 7.1
ノード数	16
ネットワーク	Myrinet2000 PCI64B (GM-1.6.4)
ライブラリ	MPICH-GM 1.2.5.10






## MPIプログラムの実行時間（台数公式プログラム）

Loop = 1000000000

使用プロセッサ数	経過時間(秒)
1	11.41
2	5.94
4	3.26
8	2.20
16	1.95

Loop = 1000

使用プロセッサ数	経過時間(秒)
1	0.28
2	0.37
4	0.47
8	0.78
16	1.30



## 並列化すればよいというものではない

---

並列プログラムの作成は非並列プログラムより難しい。

- 問題解決のアルゴリズム以外に少なくとも処理の分割方法を考慮する必要がある。
- MPIの場合, さらにデータの分割方法や通信のタイミングも重要である。

苦勞に見合うだけの結果が得られるかどうかを事前に検討する。



## どんなプログラムでも並列化可能というわけではない

並列化とは、複数の処理を同時に進行させることであるので、実行の順序が非並列の場合と異なる。そのため、実行順序によって値が変わる処理は並列化できない。

並列化できないループの例：

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

再帰参照



## 次週講義の振替のお知らせ

次週10/25(火)の並列アルゴリズム講義

「グリッド講演会」

日時: 平成17年10月25日(火) 9:00~12:00

場所: 九州大学 国際ホール

<http://www.nii.ac.jp/gridforum/>

講演演題:

9:00~ 9:45 『グリッド技術入門』

独立行政法人 産業技術総合研究所グリッド研究センター長 関口 智嗣

9:45~10:30 『グリッドの研究開発状況』

国立大学法人 東京工業大学 学術国際情報センター 教授 松岡 聡

(休憩)

10:40~11:10 『グリッドの新たな可能性』

独立行政法人 産業技術総合研究所 グリッド研究センター長 関口 智嗣

11:10~11:40 『情報基盤センターの取り組み』

国立大学法人 九州大学 情報基盤センター

11:40~12:00 『大学間連携のための全国共同電子認証基盤(UPKI)構築事業』

国立大学法人 九州大学 情報基盤センター