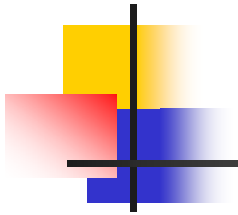


並列アルゴリズム

2005年後期　火曜　2限

青柳　睦

Aoyagi@cc.kyushu-u.ac.jp

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

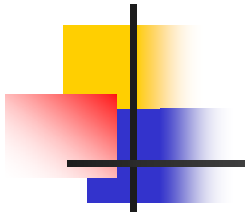
10月11日（火）

1. 序　並列計算機の現状

2. 計算方式およびアーキテクチュアの分類(途中から)

3. 並列計算の目的と課題

4. 数値計算における各種の並列化

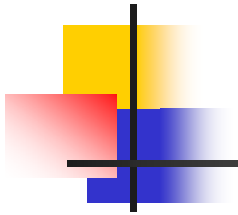


講義の概要

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)

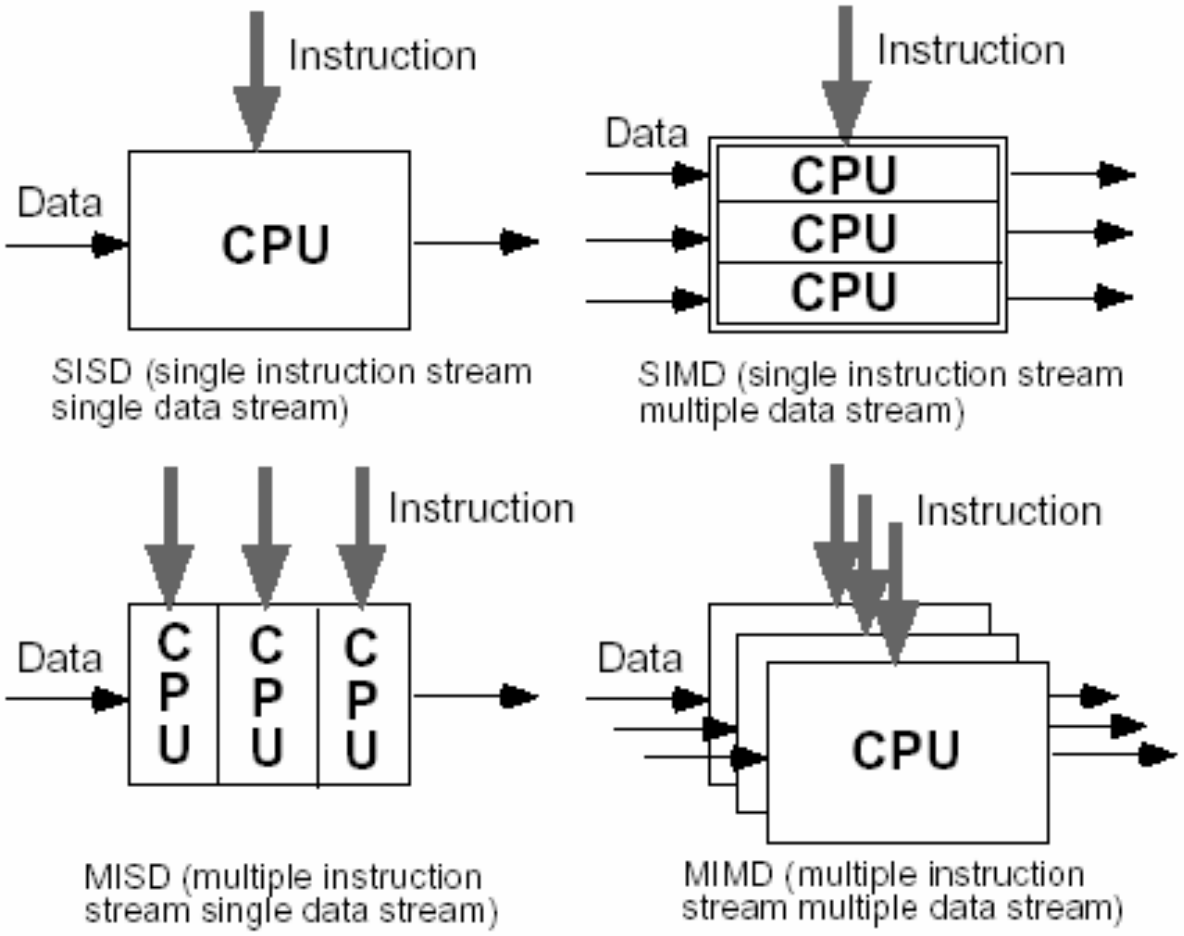
2. 計算方式およびアーキテクチュアの分類

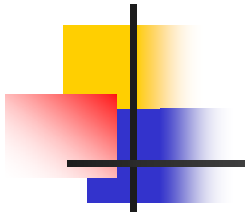
(by M. J. Flynn)

2-1.命令とデータの流れに着目した計算方式の分類1

- SISD (Single Instruction stream Single Data stream)
  - 通常の逐次計算機
- SIMD (Single Instruction stream Multiple Data stream)
  - 命令は一つだが，複数のデータに対して同じ処理を行う
  - 単純で，低コストで，高い並列生を持つ計算機を作れる
  - 計算機が物理的に大きくなると同期が大変
  - 画像処理のように同じ処理を同時に行う場合に便利
- MISD (Multiple Instruction stream Single Data stream)
  - 例がない
- MIMD (Multiple Instruction stream Multiple Data stream)
  - 各々独立した計算機が，独立したデータに対して処理
  - 比較的大きな計算機を作れる
  - 自由度が高い

2-1.命令とデータの流れに着目した計算方式の分類2

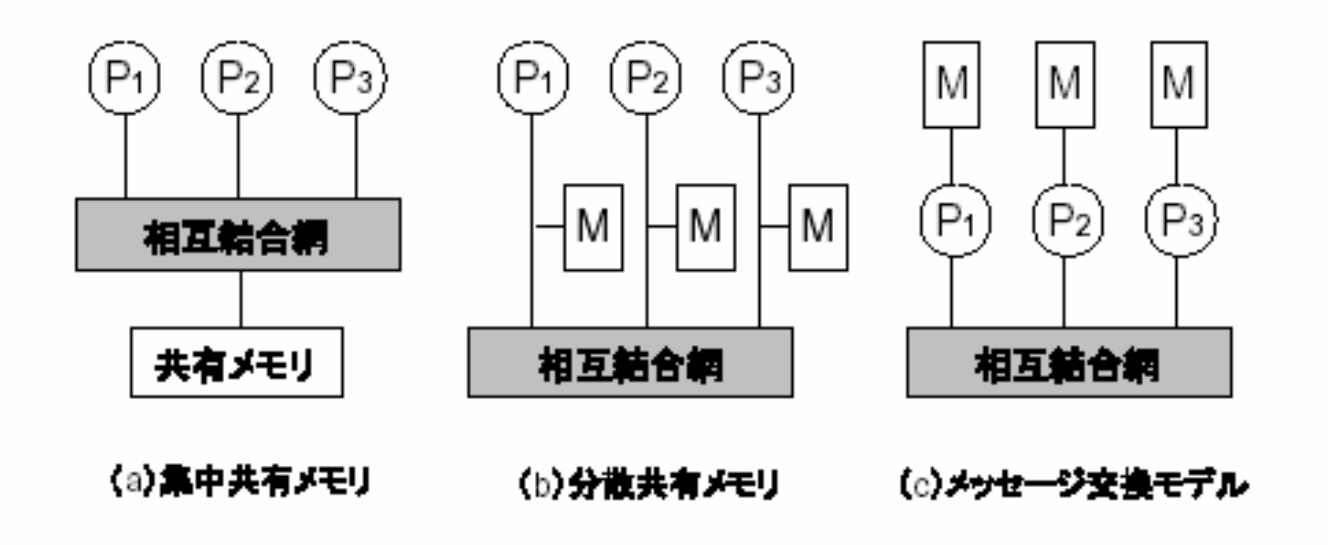




2-2.メモリシステムに着目した分類1

- UMA (Uniformed Memory Access model)
  - 全プロセッサがアドレス空間を共有
  - 全プロセッサからメモリへのアクセス時間が一様
  - 共有メモリモデル
- NUMA(Non-Uniformed Memory Access model)
  - アドレス空間は共有
  - プロセッサから見たメモリアクセス時間は一様では無い
  - 分散共有メモリモデル
- NORA (NO Remote memory Access model)
  - アドレス空間を共有しない
  - 独立したメモリのみを持つ
  - 分散メモリモデル

2-2. メモリシステムに着目した分類2

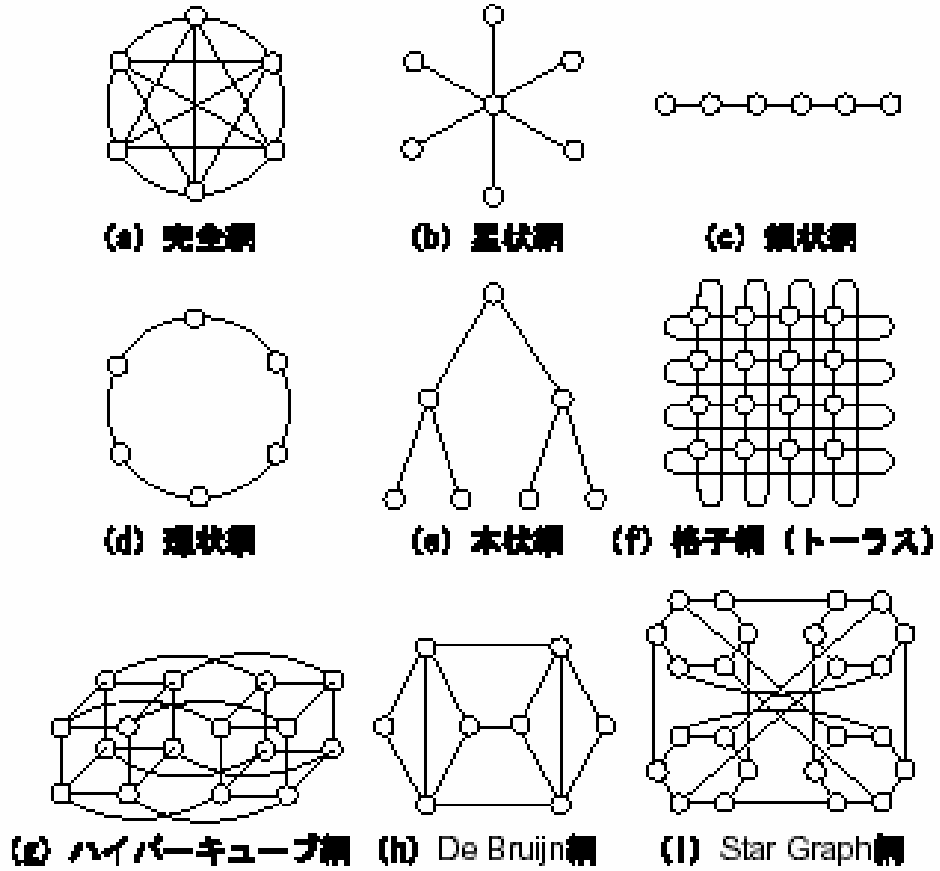


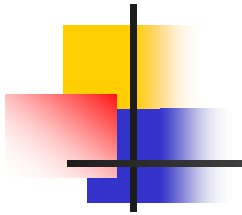
対称型マルチプロセッサ（SMP）

論理的にメモリを共有

論理的にせよメモリを共有しない

2-3.相互結合網に着目した分類1

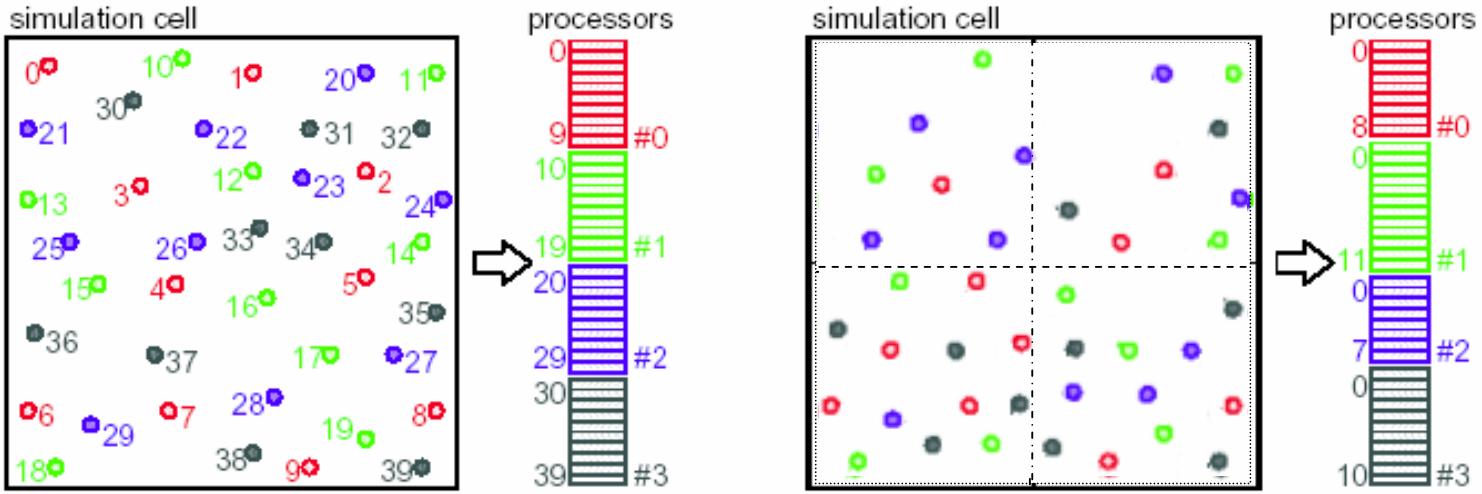




3. 並列計算の目的と課題

- 目的
  - 計算時間の短縮（大規模高性能計算）
  - 大メモリ容量が必要（大容量計算）
  - 並列アルゴリズムの研究
- 計算科学・シミュレーションの例
  - ナノスケール・分子シミュレーション
  - 大気・海洋シミュレーション（前述：地球シミュレータ　参照）
- 並列アルゴリズムの研究例
  - 粒子シミュレーションにおける「分割」

粒子シミュレーションにおける「分割」



粒子分割法

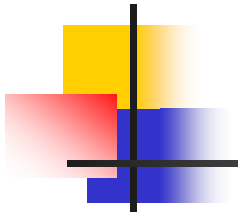
粒子数は保存
↓
計算負荷は一定
しかし
粒子の位置は変動
↓
通信負荷は不均衡
になりやすい

粒子間相互作用(Force)の計算部

```
DOi = 1, N
    DOj = 1, i
        Call calc_force(i, j)
    ENDDO
ENDDO
```

領域分割法

領域内の粒子については
通信負荷が無い（軽い）
↓
しかし
領域内粒子の数は変動
↓
計算負荷は不均衡
になりやすい



3. 並列処理の課題

- 並列化不可能な部分が有る
  - アムダール則 （3-1）
- 計算負荷のアンバランス
  - 計算粒度の問題 （3-2）
- 並列化によるオーバーヘッド
  - 通信のオーバーヘッド （3-3）
  - 並列アルゴリズムそのもの

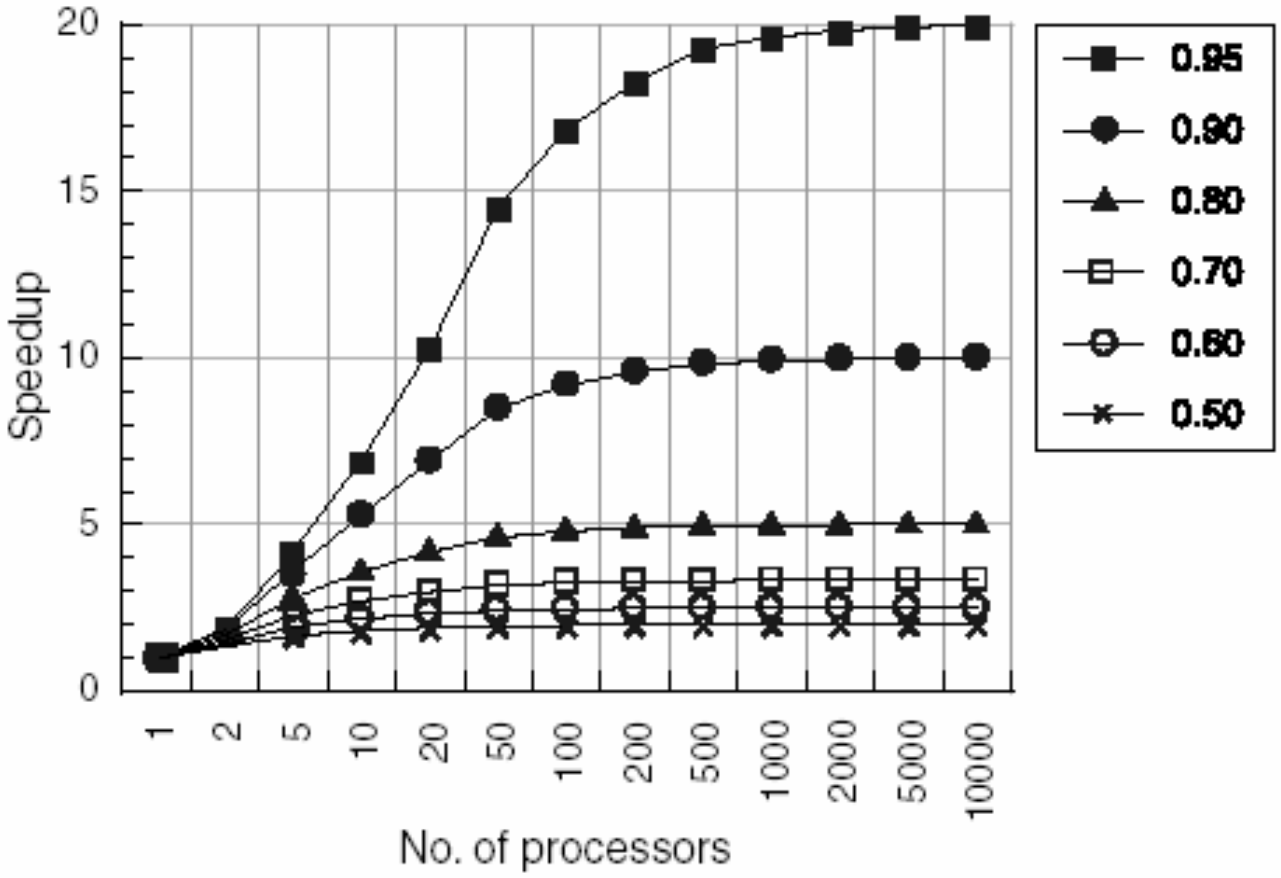
3-1.アムダール(Amdahl)の法則1

- 高速(並列)化前　　高速化後
  - 速度: $V \rightarrow \quad V'$
  - 時間: $T \rightarrow \quad T'$
- 今全体の $(1-s)$ を n 倍高速化できたとする
- 速度向上比 E_{ff}

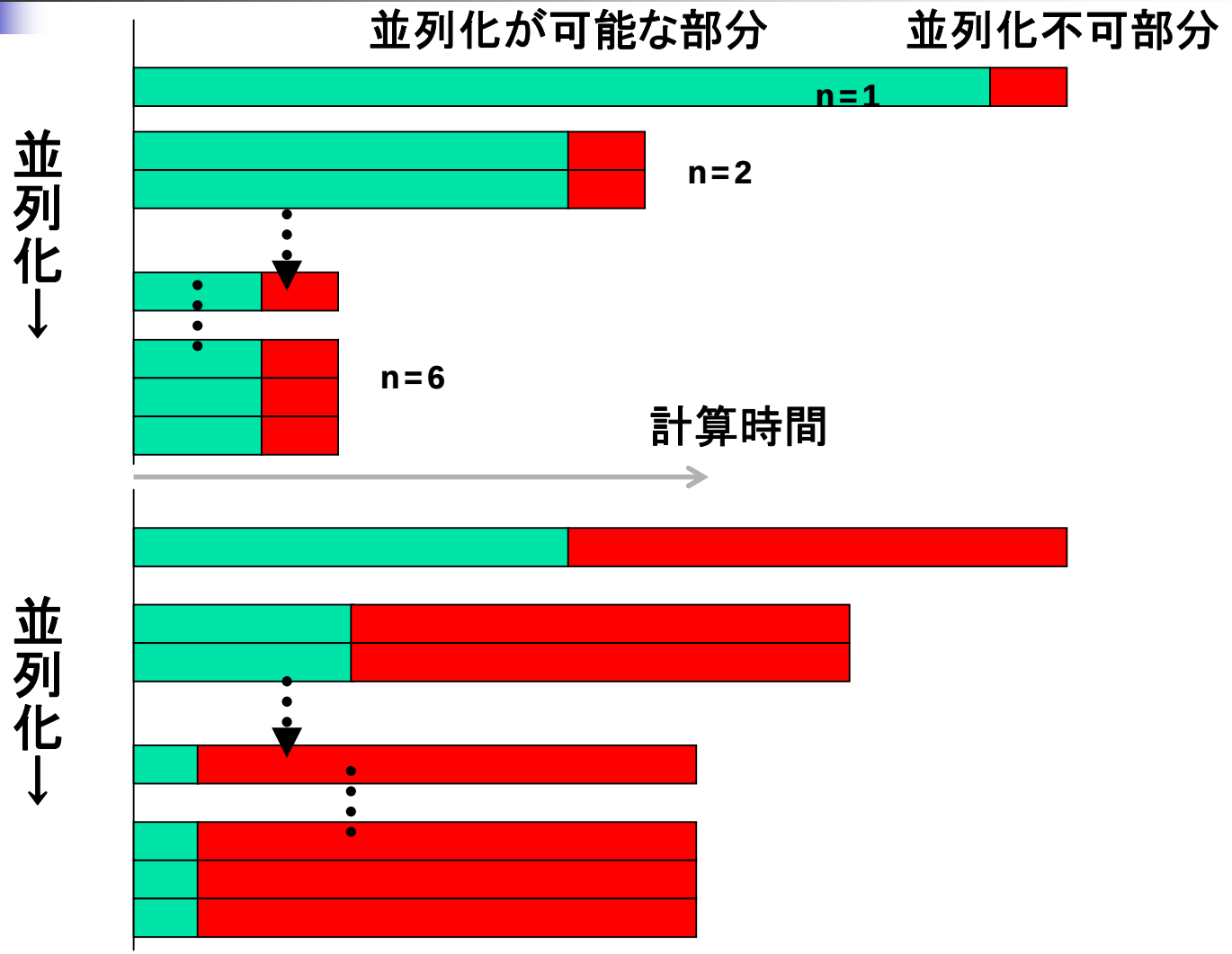
$$E_{ff} = \frac{V'}{V} = \frac{T}{T'} = \frac{T}{T(s + \frac{1-s}{n})} = \frac{1}{s + \frac{1-s}{n}}$$

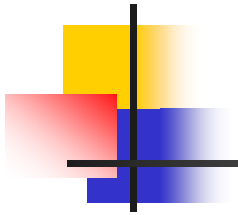
- 例えば 全体の90%を並列化できたとして
  $n=10$ で・・E_{ff}=約5．2倍

3-1.アムダール(Amdahl)の法則2



アムダール則





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

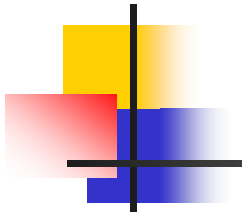
並列アルゴリズムによるオーバーヘッド

- 並列化の為にデータをコピーする
- 並列化の為の制御文を挿入

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．　もっとも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（処理のまとまり）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

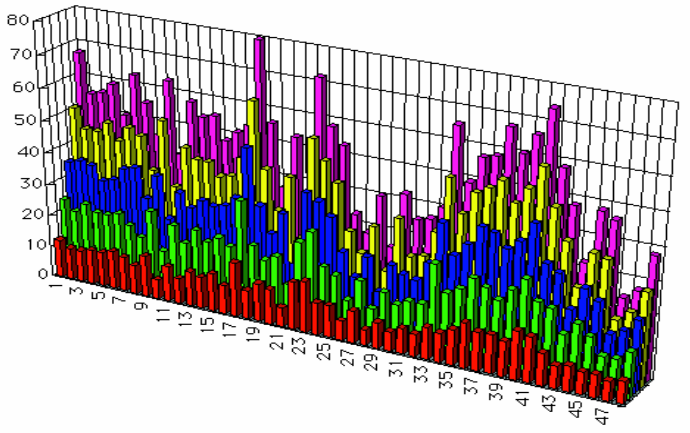
同期通信では，データを持っているプロセスは 送信 (send) というルーチンを呼び，それを受け取るプロセスは受信 (recv) というルーチンを呼ぶ．このタイプの通信では，データのやり取りと同期を兼ねており，受信ルーチンを呼び出すまでは相手のメッセージは自分の領域に入ってこないことが保証される点でプログラムの構造は比較的簡単な場合が多いが，通信は各時刻で必ず一方通行 (half duplex) となる．

非同期通信

非同期通信では、受け手が受信ルーチンを呼び出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 Forumh(ttp://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,
共有メモリ的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. 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);
```

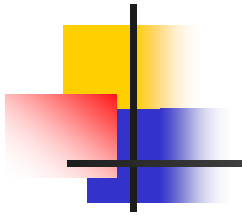
```
if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen+1 so that '¥0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR,
            dest, tag, MPI_COMM_WORLD);
    } 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.