並列プログラミング入門

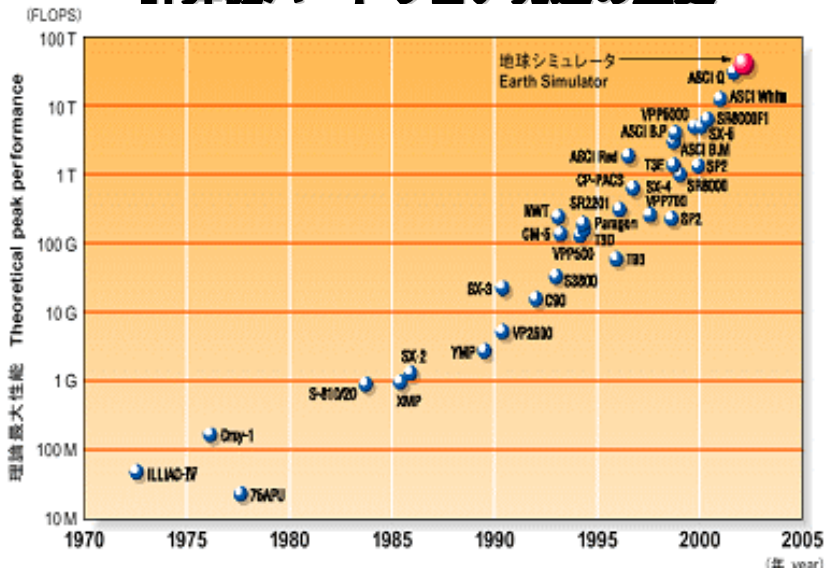
筑波大学　計算科学研究センター
担当　佐藤

もくじ

- 基本的な知識
- 並列プログラミング環境
  - OpenMP
  - MPI
- OpenとMPIでのプログラムの比較
- まとめ

計算機ハードウェア発達の歴史

http://www.es.jamstec.go.jp/



http://www.top500.org/

高速化とは

- コンピュータの高速化
  - デバイス
  - 計算機アーキテクチャ
- 計算機アーキテクチャの高速化の本質は、いろいろな処理を同時にやること
  - CPUの中
  - チップの中
  - チップ間
  - コンピュータ間

パイプライン、スーパスカラ

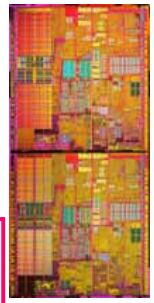
マルチ･コア

共有メモリ並列コンピュータ

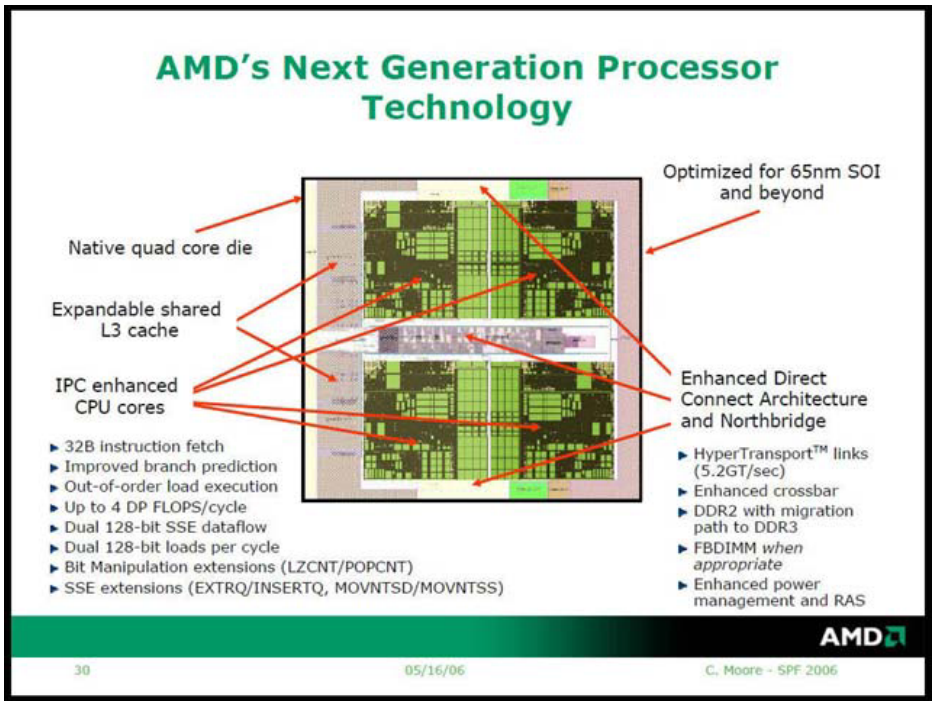
分散メモリ並列コンピュータ グリッド

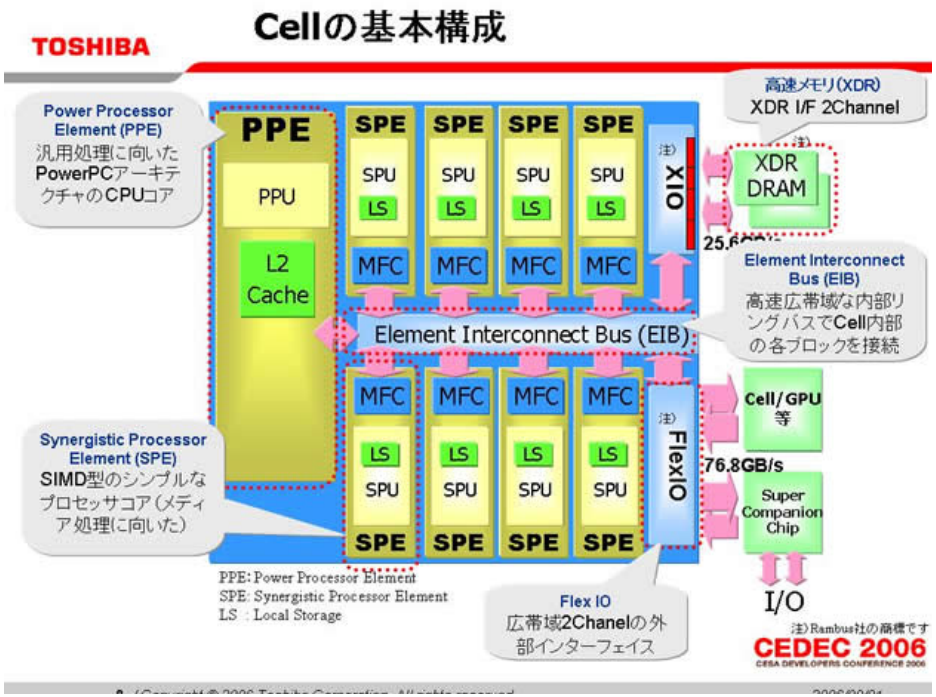
プロセッサ研究開発の動向

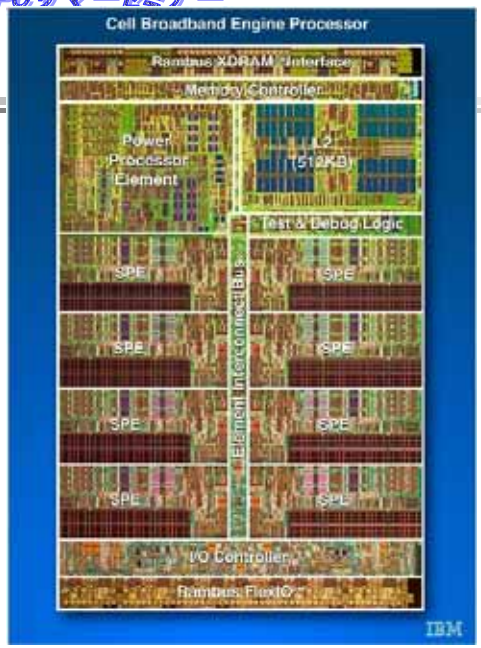
- クロックの高速化、製造プロセスの微細化
  - いまでは3GHz, 数年のうちに10GHzか！？
    - インテルの戦略の転換　⇒　マルチコア
  - プロセスは90nm　⇒65nm , 将来的には45nm
    - トランジスタ数は増える！
- アーキテクチャの改良
  - スーパーパイプライン、スーパースカラ、VLIW…
  - キャッシュの多段化、マイクロプロセッサでもL3キャッシュ
  - マルチスレッド化、Intel Hyperthreading、複数のプログラムを同時に処理
  - マルチコア：１つのチップに複数のCPU



インテル® Pentium® プロセッサ エクストリーム･エディションのダイ







Cellのダイ

2億3400万個のトランジスタを実装するが、90nmプロセスで製造されることから、ダイ･サイズは 221mm2と意外と小さい。細長い部分がSPUである。

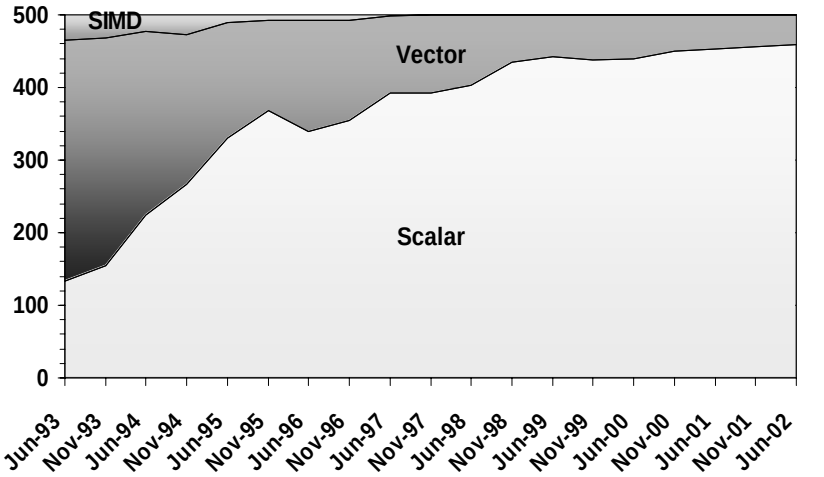
計算機ハードウェアの歴史

- 1.5年に2倍の割合で処理速度が増加している（Moore's Law）
- 1983年：1 GFLOPS，1996年：1 TFLOPS，2002年：36 GFLOPS
  - MFLOPS: Millions of FLoating Point OPerationS.（1秒間に10^6回の浮動小数点処理）
  - GFLOPS：10^9回，TFLOPS：10^{12}回，PFLOPS：10^{15}回
  - 2010年頃にはPFLOPS（Peta FLOPS）マシンが登場すると言われている。
- 「地球シミュレータ」はピーク性能 40 TFLOPS
- 2005年にはIBM BlueGene/L（367 TFLOPS）が完成
- スーパーコンピュータは並列処理の時代へ
  - 単一プロセッサでは限界！
  - スーパーコンピュータは並列処理により早くなっている

計算機の進歩

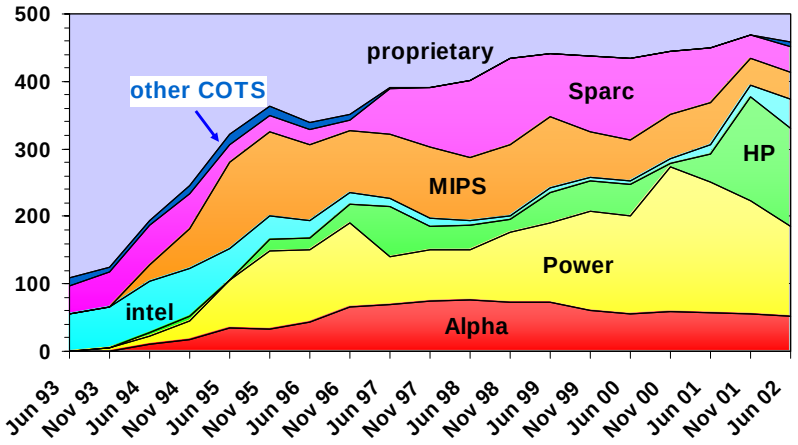
- 個々のプロセッサ
  - ベクトルプロセッサ　⇒　10年前はこのタイプが多かった
    - 一つのプロセッサで行列演算を効率的にできる
  - スカラープロセッサ：Pentium, Power，Alpha，Itanium
- 並列計算機のアーキテクチャの種類
  - 分散メモリ型並列計算機
  - SMP（Symmetrical Multi Processor）：共有メモリ型並列計算機
  - SMPクラスタ型並列計算機（「Constellation」とも言う）
- 並列計算機の別の分類
  - PCクラスタ
  - 専用並列計算機
  - 専用機・・・GRAPE等：天体のシミュレーション

Processor Type



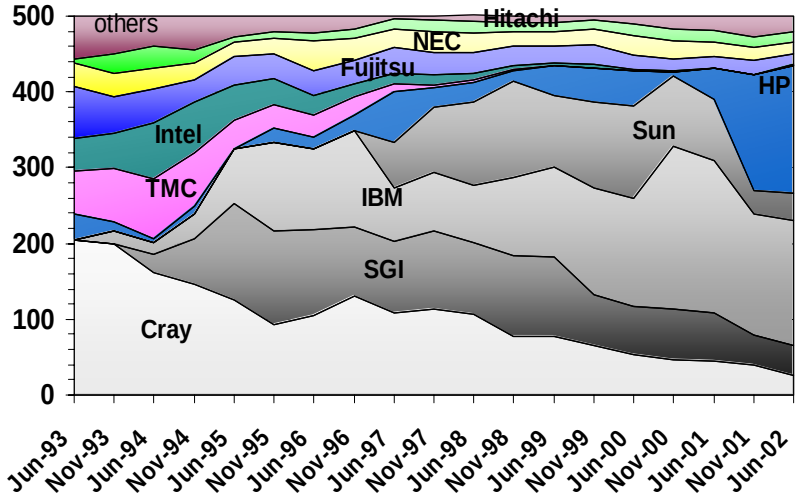
http://www.top500.org/

Chip Technology



http://www.top500.org/

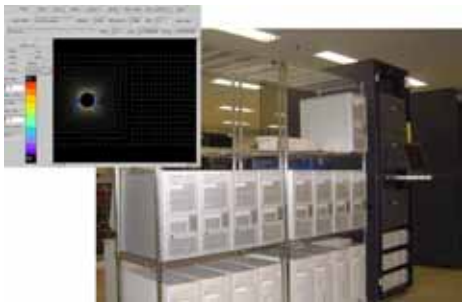
Manufacturers



http://www.top500.org/

クラスタコンピューティング

- PCやネットワークなど、汎用（コモデティ）の部品を組み合わせて、並列システムを構成する技術
  - PCが急激に進歩した！
  - ネットワークも早くなっている！
  - 安価に、個人レベルでも並列システムを作れる！



PACS-CS

Parallel Array Computer System for Computational Sciences

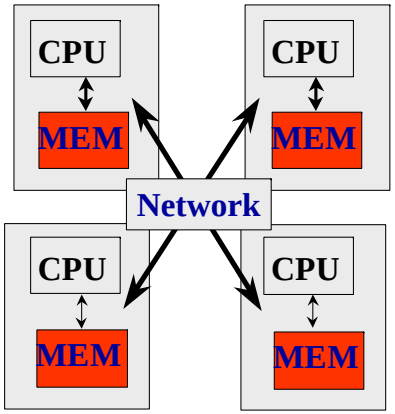
- 計算科学計算センターで７月から稼動
- コモデティ（ＰＣ、イーサネット）」で構成

- 国内で開発されたスパコンとしては、２位

Peak 14.34TF
Linpack 10.35TF

34位(2006/6)

分散メモリ型計算機

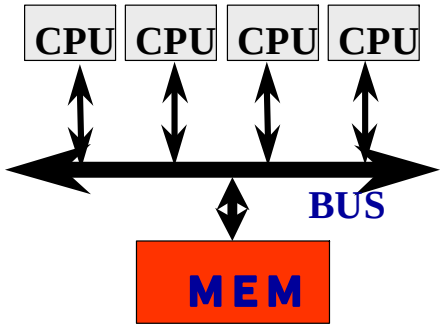


◆CPUとメモリという一つの計算機システムが、ネットワークで結合されているシステム

◆それぞれの計算機で実行されているプログラムはネットワークを通じて、データ(メッセージ)を交換し、動作する

◆超並列（MPP：Massively Parallel Processing)コンピュータ
◆クラスタ計算機

共有メモリ型計算機



◆複数のCPUが一つのメモリにアクセスするシステム。

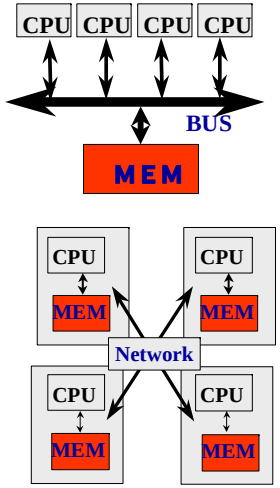
◆それぞれのCPUで実行されているプログラム(スレッド)は、メモリ上のデータにお互いにアクセスすることで、データを交換し、動作する。

◆大規模サーバ

並列処理の利点

- 計算能力が増える。
  - １つのＣＰＵよりも多数のＣＰＵ。
- メモリの読み出し能力（バンド幅）が増える。
  - それぞれのＣＰＵがここのメモリを読み出すことができる。
- ディスク等、入出力のバンド幅が増える。
  - それぞれのＣＰＵが並列にディスクを読み出すことができる。
- キャッシュメモリが効果的に利用できる。
  - 単一のプロセッサではキャッシュに載らないデータでも、処理単位が小さくなることによって、キャッシュを効果的に使うことができる。
- 低コスト
  - マイクロプロセッサをつかえば。

→ クラスタ技術



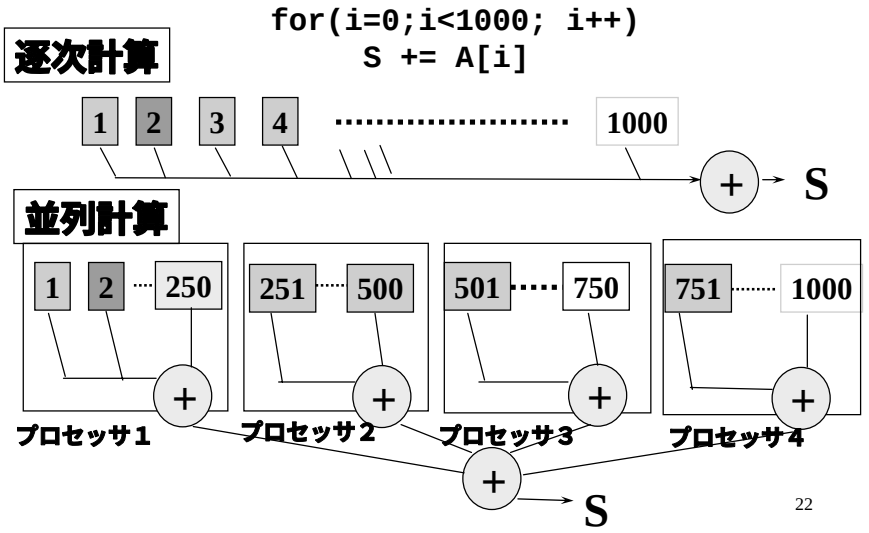
並列プログラミング

- メッセージ通信 (Message Passing)
  - 分散メモリシステム（共有メモリでも、可）
  - プログラミングが面倒、難しい
  - プログラマがデータの移動を制御
  - プロセッサ数に対してスケーラブル
- 共有メモリ (shared memory)
  - 共有メモリシステム（DSMシステムon分散メモリ）
  - プログラミングしやすい（逐次プログラムから）
  - システムがデータの移動を行ってくれる
  - プロセッサ数に対してスケーラブルではないことが多い。

並列プログラミング

- メッセージ通信プログラミング
  - MPI,PVM
- 共有メモリプログラミング
  - マルチスレッドプログラミング
    - pthread, solaris thread, NT thread
  - OpenMP
    - 指示文によるannotation
    - thread制御など共有メモリ向け
  - HPF
    - 指示文によるannotation,
    - distributionなど分散メモリ向け
- 自動並列化
  - 逐次プログラムをコンパイラで並列化
    - コンパイラによる解析には制限がある。指示文によるhint
- Fancy parallel programming languages

並列処理の簡単な例



POSIXスレッドによるプログラミング

- スレッドの生成

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
       pthread_join();
```

- ループの担当部分の分割
- 足し合わせの同期

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```


OpenMPによるプログラミング

これだけで、OK!

```
#pragma omp parallel for reduction(+:s)
   for(i=0; i<1000;i++) s+= a[i];
```

OpenMPとは

- 共有メモリマルチプロセッサの並列プログラミングのためのプログラミングモデル
  - ベース言語(Fortran/C/C++)をdirective（指示文）で並列プログラミングできるように拡張
- 米国コンパイラ関係のISVを中心に仕様を決定
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - 現在、OpenMP 3.0が策定中
- URL
  - `http://www.openmp.org/`

背景

- 共有メモリマルチプロセッサシステムの普及
  - SGI Cray Origin
    - ASCI Blue Mountain System
  - SUN Enterprise
  - PC-based SMPシステム
  - そして、いまや　マルチコア！
- 共有メモリマルチプロセッサシステムの並列化指示文の共通化の必要性
  - 各社で並列化指示文が異なり、移植性がない。
    - SGI Power Fortran/C
    - SUN Impact
    - KAI/KAP
- OpenMPの指示文は並列実行モデルへのＡＰＩを提供
  - 従来の指示文は並列化コンパイラのためのヒントを与えるもの

科学技術計算とOpenMP

- 科学技術計算が主なターゲット（これまで）
  - 並列性が高い
  - コードの5%が95%の実行時間を占める(?)
    - 5%を簡単に並列化する
- 共有メモリマルチプロセッサシステムがターゲット
  - small-scale(～16プロセッサ）からmedium-scale (～64プロセッサ）を対象
  - 従来はマルチスレッドプログラミング
    - pthreadはOS-oriented, general-purpose
- 共有メモリモデルは逐次からの移行が簡単
  - 簡単に、少しずつ並列化ができる。
    - （でも、デバックはむずかしいかも）

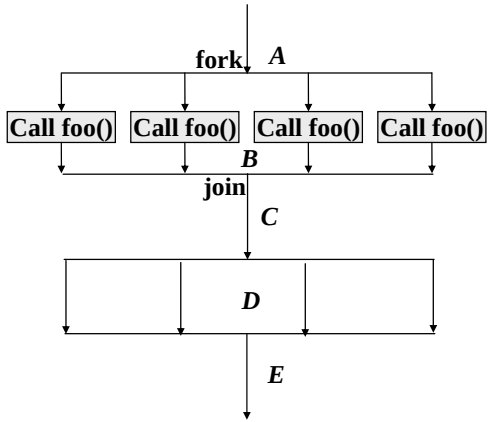
OpenMPのAPI

- 新しい言語ではない！
  - コンパイラ指示文（directives/pragma）、ライブラリ、環境変数によりベース言語を拡張
  - ベース言語：Fortran77, f90, C, C++
    - Fortran： !$OMPから始まる指示行
    - C: #pragma omp のpragma指示行
- 自動並列化ではない！
  - 並列実行･同期をプログラマが明示
- 指示文を無視することにより、逐次で実行可
  - incrementalに並列化
  - プログラム開発、デバックの面から実用的
  - 逐次版と並列版を同じソースで管理ができる

OpenMPの実行モデル

- 逐次実行から始まる
- Fork-joinモデル
- parallel region
  - 関数呼び出しも重複実行

```
… A …
#pragma omp parallel
{
    foo(); /* ..B… */
}
… C ….
#pragma omp parallel
{
… D …
}
… E …
```



Parallel Region

- 複数のスレッド(team)によって、並列実行される部分
  - Parallel構文で指定
  - 同じParallel regionを実行するスレッドをteamと呼ぶ
  - region内をteam内のスレッドで重複実行
    - 関数呼び出しも重複実行

Fortran:

```
!$OMP PARALLEL
    …
    … parallel region
    …
!$OMP END PARALLEL
```

C:

```
#pragma omp parallel
{
    …
    … Parallel region…
    …
}
```


Work sharing構文

- Team内のスレッドで分担して実行する部分を指定
  - parallel region内で用いる
  - for 構文
    - イタレーションを分担して実行
    - データ並列
  - sections構文
    - 各セクションを分担して実行
    - タスク並列
  - single構文
    - 一つのスレッドのみが実行
  - parallel 構文と組み合わせた記法
    - parallel for 構文
    - parallel sections構文

For構文

- Forループ（DOループ）のイタレーションを並列実行
- 指示文の直後のforループはcanonical shapeでなくてはならない

```
#pragma omp for [clause…]
 for(var=lb; var logical-op ub; incr-expr)
   body
```

  - *var*は整数型のループ変数（強制的にprivate）
  - *incr-expr*
    - ++*var*, *var*++, --*var*, *var*--, *var*+=*incr*, *var*-=*incr*
  - *logical-op*
    - <、<=、>、>=
  - ループの外の飛び出しはなし、breakもなし
  - *clause*で並列ループのスケジューリング、データ属性を指定

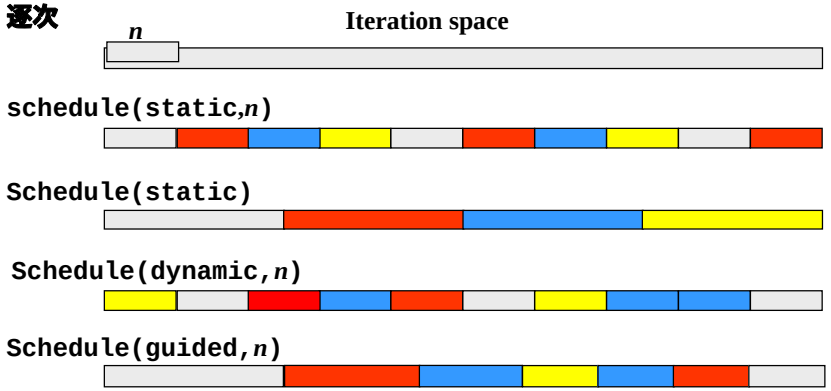
例

疎行列ベクトル積ルーチン

```
Matvec(double a[],int row_start,int col_idx[],
 double x[],double y[],int n)
{
   int i,j,start,end; double t;
#pragma omp parallel for private(j,t,start,end)
   for(i=0; i<n;i++){
      start=row_start[i];
      end=row_start[i+1];
      t = 0.0;
      for(j=start;j<end;j++)
         t += a[j]*x[col_idx[j]];
      y[i]=t;
  }
}
```


並列ループのスケジューリング

- プロセッサ数４の場合



Data scope属性指定

- parallel構文、work sharing構文で指示節で指定
- shared(*var_list*)
  - 構文内で指定された変数がスレッド間で共有される
- private(*var_list*)
  - 構文内で指定された変数がprivate
- firstprivate(*var_list*)
  - privateと同様であるが、直前の値で初期化される
- lastprivate(*var_list*)
  - privateと同様であるが、構文が終了時に逐次実行された場合の最後の値を反映する
- reduction(*op*:*var_list*)
  - reductionアクセスをすることを指定、スカラ変数のみ
  - 実行中はprivate、構文終了後に反映

Barrier 指示文

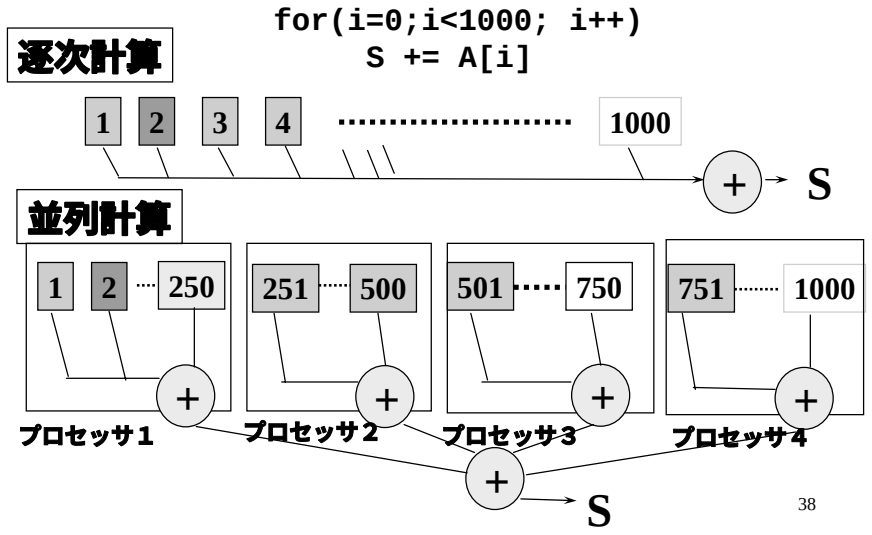
- バリア同期を行う
  - チーム内のスレッドが同期点に達するまで、待つ
  - それまでのメモリ書き込みもflushする
  - 並列リージョンの終わり、work sharing構文でnowait指示節が指定されない限り、暗黙的にバリア同期が行われる。

```
#pragma omp barrier
```

MPIによるプログラミング

- MPI (Message Passing Interface)
- 現在、分散メモリシステムにおける標準的なプログラミングライブラリ
  - 100ノード以上では必須
  - 面倒だが、性能は出る
    - アセンブラでプログラミングと同じ
- メッセージをやり取りして通信を行う
  - Send/Receive
- 集団通信もある
  - Reduce/Bcast
  - Gather/Scatter

並列処理の簡単な例



MPIでプログラミングしてみると

```
#include "mpi.h"
#include <stdio.h>
#define MY_TAG 100
double A[1000/N_PE];
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s¥n", myid, processor_name);

    ....
```


MPIでプログラミングしてみると

```
    sum = 0.0;
    for (i = 0; i < 1000/N_PE; i++){
       sum+ = A[i];
    }

    if(myid == 0){
       for(i = 1; i < numprocs; i++){
          MPI_Recv(&t,1,MPI_DOUBLE,i,MY_TAG,MPI_COMM_WORLD,&status)
            sum += t;
        }
    } else
          MPI_Send(&t,1,MPI_DOUBLE,0,MY_TAG,MPI_COMM_WORLD);
    /* MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_\
     MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

コンパイル・実行の仕方

- MPICH（クラスタ）などの場合
- コンパイル

```
% mpicc  … test.c …
```

  - MPI用のコンパイルコマンドがある
  - もちろん、手動でリンクもできる

- 実行

```
% Mpirun –np <n_proc>  a.out …
```

  - 設定されているプロセッサ群で、a.outが実行される
  - Mpichの場合はmachine fileで設定
  - デーモンをあらかじめ立ち上げておかなくてはならない場合がある。

解説

- まず、宣言されたデータは各プロセッサに重複して取られている。
  - なので、ひとつのプロセッサではプロセッサ数N_PEで割った分だけでいい
- 各プロセッサでは、mainからプログラムが実行される
- SPMD (single program/multiple data)
  - 大体、同じようなところを違うデータ（つまり、実行されているノードにあるデータ）に対して実行するようなプログラムのこと
- 初期化
  - MPI_Init
  - プロセッサ数を取る

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
```

  - 現在実行されているプロセッサ番号（rankという）
    - 0から始まる

```
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```


解説（続き）

- コミュニケータ
  - 通信のcontextと保持する仕組み
  - MPI_COMM_WORLDだけつかえば、当分の間十分
- 計算・通信
  - 各プロセッサで部分和を計算して、集計
  - Send/Recvを組み合わせて実現
  - これでもＯＫ

```
MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE,
        MPI_SUM, 0, MPI_COMM_WORLD);
```

- 最後にexitの前で、前プロセッサで！

```
MPI_Finalize();
```


メッセージ通信

- Send/Receive

```
MPI_Send(
  void         *send_data_buffer, // 送信データが格納されているメモリのアドレス
  int          count,             // 送信データの個数
  MPI_Datatype data_type,         // 送信データの型(*1)
  int          destination,       // 送信先プロセスのランク
  int          tag,               // 送信データの識別を行うタグ
  MPI_Comm     communicator       // 送受信を行うグループ.
);
```

```
MPI_Recv(
  void         *recv_data_buffer, // 受信データが格納されるメモリのアドレス
  int          count,             // 受信データの個数
  MPI_Datatype data_type,         // 受信データの型(*1)
  int          source,            // 送信元プロセスのランク
  int          tag,               // 受信データの識別を行うためのタグ.
  MPI_Comm     communicator,      // 送受信を行うグループ.
  MPI_Status   *status            // 受信に関する情報を格納する変数のアドレス
);
```

メッセージ通信

- メッセージはデータアドレスとサイズ
  - 型がある　MPI_INT MPI_DOUBLE …
  - Binaryの場合は、MPI_BINARYで、サイズにbyte数を指定
- Source/destinationは、プロセッサ番号（rank)とタグを指定
  - 同じタグを持っているSendとRecvがマッチ
  - どのようなタグでもRecvしたい場合はMPI_ANYを指定
- コミュニケータはMPI_COMM_WORLD
- Statusは適当に
- 注意
  - これは、同期通信
  - つまり、recvが完了しないと、sendは完了しない。

注:正確には、sendはバッファにあるデータを送りだした時点で終了する。しかし、recvされないと送りだしができないことがあるので、「相手がrecvしないとsendが終了しない」として理解したほうが安全。

OpenMPとMPIのプログラム例：Cpi

- 積分して、円周率を求めるプログラム
- MPICHのテストプログラム

$$\pi = \int_0^1 \frac{4}{1+t^2}\,dt$$

- OpenMP版（cpi-seq.c)
  - ループを並列化するだけ, 1行のみ
- MPI版(cpi-mpi.c)
  - 入力された変数nの値をBcast
  - 最後にreduction
  - 計算は、プロセッサごとに飛び飛びにやっている

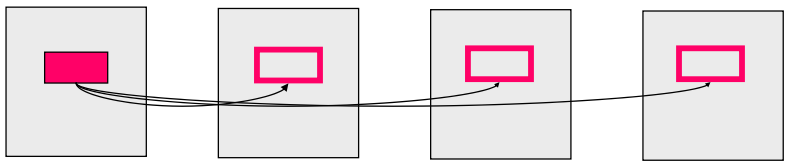

```
…
 MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

 h   = 1.0 / (double) n;
 sum = 0.0;
 for (i = myid + 1; i <= n; i += numprocs){
  x = h * ((double)i - 0.5);
  sum += f(x);
 }
 mypi = h * sum;

 MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
              MPI_SUM, 0, MPI_COMM_WORLD);
```


集団通信: ブロードキャスト

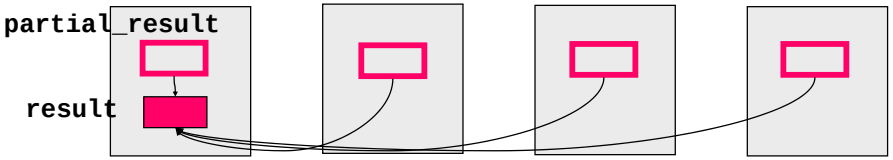
```
MPI_Bcast(
  void         *data_buffer,    // ブロードキャスト用送受信バッファのアドレ
  int          count,           // ブロードキャストデータの個数
  MPI_Datatype data_type,       // ブロードキャストデータの型(*1)
  int          source,          // ブロードキャスト元プロセスのランク
  MPI_Comm     communicator     // 送受信を行うグループ
);
```



全プロセッサで実行されなくてはならない

集団通信: リダクション

```
MPI_Reduce(
  void         *partial_result, // 各ノードの処理結果が格納されているアドレス
  void         *result,         // 集計結果を格納するアドレス
  int          count,           // データの個数
  MPI_Datatype data_type,       // データの型(*1)
  MPI_Op       operator,        // リデュースオペレーションの指定(*2)
  int          destination,     // 集計結果を得るプロセス
  MPI_Comm     communicator     // 送受信を行うグループ
);
```

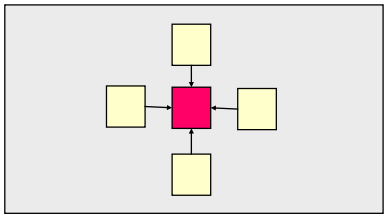


全プロセッサで実行されなくてはならない

Resultを全プロセッサで受け取る場合は、MPI_AllReduce

OpenMPとMPIのプログラム例：laplace

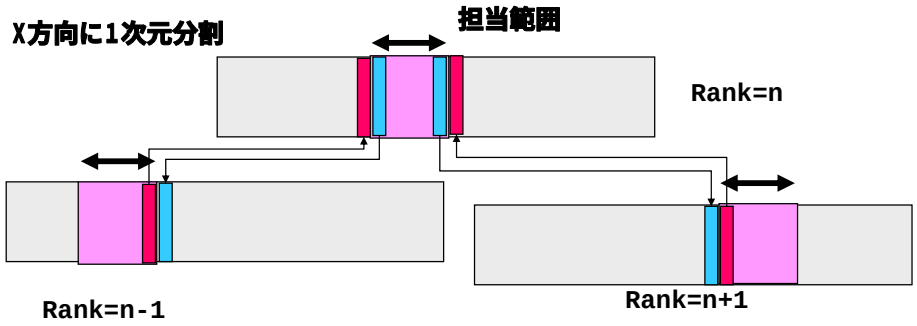
- Laplace方程式の陽的解法
  - 上下左右の4点の平均で、updateしていくプログラム
  - Oldとnewを用意して直前の値をコピー
  - 典型的な領域分割
  - 最後に残差をとる



- OpenMP版 lap.c
  - 3つのループを外側で並列化
    - OpenMPは１次元のみ
  - Parallel指示文とfor指示文を離してつかってみた
- MPI版
  - 結構たいへん

隣接通信

- 隣の部分を繰り返しごとに通信しなくてはならない



- 非同期通信を使う方法
- 同期通信を使う方法
- Sendrecvを使う方法

非同期通信

- Send/recvを実行して、後で終了をチェックする通信方法
  - 通常のsend/recv（同期通信）では、オペレーションが終了するまで、終わらない

```
int MPI_Isend( void *buf, int count, MPI_Datatype datatype,
   int dest, int tag, MPI_Comm comm, MPI_Request *request )
```

```
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
 int source, int tag, MPI_Comm comm, MPI_Request *request )
```

```
int MPI_Wait ( MPI_Request  *request,
               MPI_Status   *status)
```

```
if(myid != 0)  /* recv from down */
    MPI_Irecv(&uu[x_start-1][0],YSIZE,MPI_DOUBLE,myid-1,TA(
            MPI_COMM_WORLD,&req1);
if(myid != (numprocs -1)) /* recv from up */
    MPI_Irecv(&uu[x_end][0],YSIZE,MPI_DOUBLE,myid+1,TAG_2,
            MPI_COMM_WORLD,&req2);

if(myid != 0) /* send to down */
    MPI_Send(&u[x_start][0],YSIZE,MPI_DOUBLE,myid-1,TAG_2,
           MPI_COMM_WORLD);
if(myid != (numprocs-1)) /* send to up */
    MPI_Send(&u[x_end-1][0],YSIZE,MPI_DOUBLE,myid+1,TAG_1,
           MPI_COMM_WORLD);

if(myid != 0) MPI_Wait(&req1,&status1);
if(myid != (numprocs -1)) MPI_Wait(&req2,&status2);
```

端(0とnumprocs-1)のプロセッサについては特別な処理が必要なことを注意

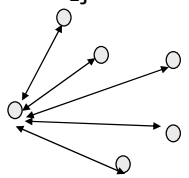
改善すべき点

- 配列の一部しか使っていないので、使うところだけにする
  - 配列のindexの計算が面倒になる
  - 大規模計算では本質的な点
- １次元分割だけだが、２次元分割したほうが効率がよい
  - 通信量が減る
  - 多くのプロセッサが使える

粒子計算の例

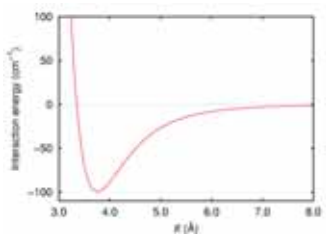
- 個々の粒子の相互作用を計算する
- 宇宙：重力
  - $F_{ij} = G\, m_i\, m_j / r^2$
- 分子シミュレーション：Van der Waals forces
  - 分子動力学
    - ２体（或いはそれ以上）の原子間ポテンシャルの下に、古典力学におけるニュートン方程式を解いて、系の静的、動的安定構造や、動的過程（ダイナミクス）を解析する手法。

$F_i = \Sigma F_{ij}$



F = m a
V = v+aΔt
p = p+vΔt

各ステップで運動方程式を解く



PSC 2001から

n 個の質点 $P_0, \cdots, P_{n-1}$ の質量および，初期速度，位置が与えられる．速度，位置は 3 次元のベクトルで，各成分は倍精度型浮動小数点数として与えられる．質量は倍精度型浮動小数点数として与えられるが，実は整数である．

P_i の位置が (x_i, y_i, z_i) であるとき，P_i の加速度 $\vec{\alpha}_i$ は以下の式で定まる．

$$\vec{\alpha}_i = \sum_{j=0}^{n-1} \vec{\alpha}_{i,j} \tag{1}$$

$$\vec{\alpha}_{i,j} = m_j f(|\vec{r}_{i,j}|^2)\vec{r}_{i,j} \tag{2}$$

ただし，

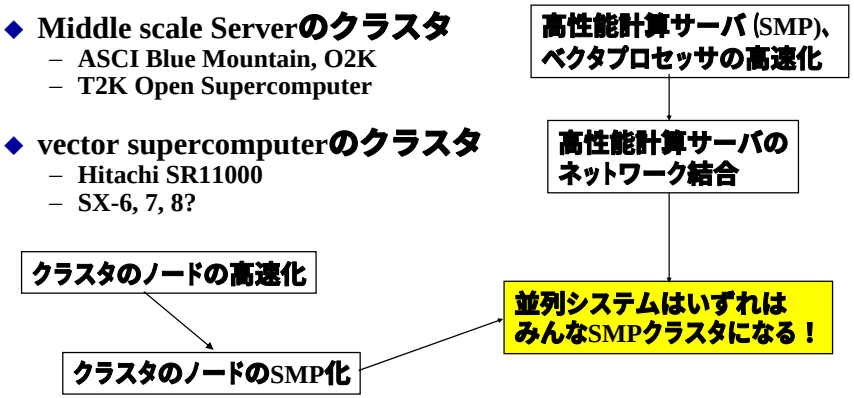
$$f(X) = \begin{cases} (X-256)(X-1024) & (X \le 1024\text{のとき}) \\ 0 & (X > 1024\text{のとき}) \end{cases}$$

$$\vec{r}_{i,j} = (\mathrm{round}(x_i - x_j), \mathrm{round}(y_i - y_j), \mathrm{round}(z_i - z_j))$$

である．つまり，$\vec{r}_{i,j}$ はベクトル $\vec{p}_i - \vec{p}_j$ の各成分を，整数に丸めた (四捨五入した) ものである．

SMPクラスタ

- PC-based SMPクラスタ
  - マルチコア
- Middle scale Serverのクラスタ
  - ASCI Blue Mountain, O2K
  - T2K Open Supercomputer
- vector supercomputerのクラスタ
  - Hitachi SR11000
  - SX-6, 7, 8?



MPIとOpenMPの混在プログラミング

- 分散メモリは、MPIで、中のSMPはOpenMPで
- MPI+OpenMP
  - はじめに、MPIのプログラムを作る
  - 並列にできるループを並列実行指示文を入れる
    - 並列部分はSMP上で並列に実行される。
- OpenMP+MPI
  - OpenMPによるマルチスレッドプログラム
  - single構文・master構文・critical構文内で、メッセージ通信を行う。
    - thread-SafeなMPIが必要
    - いくつかの点で、動作の定義が不明な点がある
      - マルチスレッド環境でのMPI
      - OpenMPのthreadprivate変数の定義？
- SMP内でデータを共用することができるときに効果がある。
  - かならずしもそうならないことがある（メモリバス容量の問題？）

おわりに

- これからの高速化には、並列化は必須
- １６プロセッサぐらいでよければ、OpenMP
- それ以上になれば、MPIが必須
  - だだし、プログラミングのコストと実行時間のトレードオフか
  - 長期的には、MPIに変わるプログラミング言語が待たれる
- 科学技術計算の並列化はそれほど難しくない
  - 内在する並列性がある
  - 大体のパターンが決まっている
  - 並列プログラムの「デザインパターン」

性能も…