OpenMP

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

- **背景**
- **並列プログラミング環境 (超入門編)**
  - OpenMP
  - MPI
- Open**プログラミングの概要**
- Advanced Topics
  - SMP**クラスタ**、Hybrid Programming
  - OpenMP 3.0 (task)
- **まとめ**

計算の高速化とは

- コンピュータの高速化
  - デバイス
  - 計算機アーキテクチャ

パイプライン、
スーパスカラ

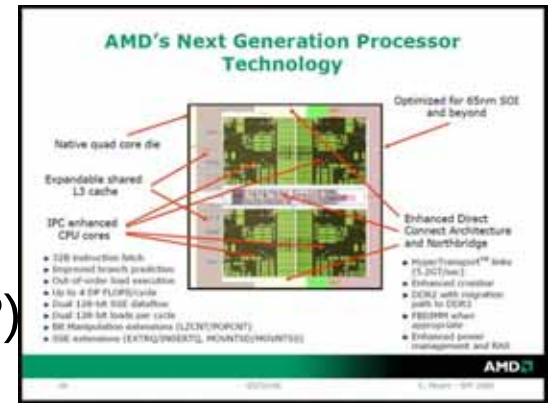
- 計算機アーキテクチャの高速化の本質は、いろいろな処理を同時にやること
  - CPUの中
  - チップの中
  - チップ間
  - コンピュータ間

マルチ・コア

共有メモリ
並列コンピュータ

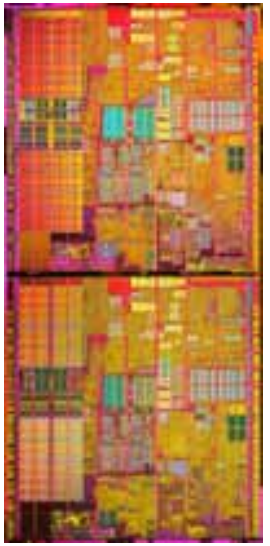
分散メモリ並列コンピュータ
グリッド

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**65nm⇒45nm , **将来的には**32nm(20?)
    - **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

並列プログラミングの必要性

- 並列処理が必要なコンピュータの普及
  - クラスタ
    - 誰でも、クラスタが作れる
  - マルチ・コア
    - 1つのチップに複数のCPUが！
  - サーバ
    - いまではほとんどがマルチプロセッサ

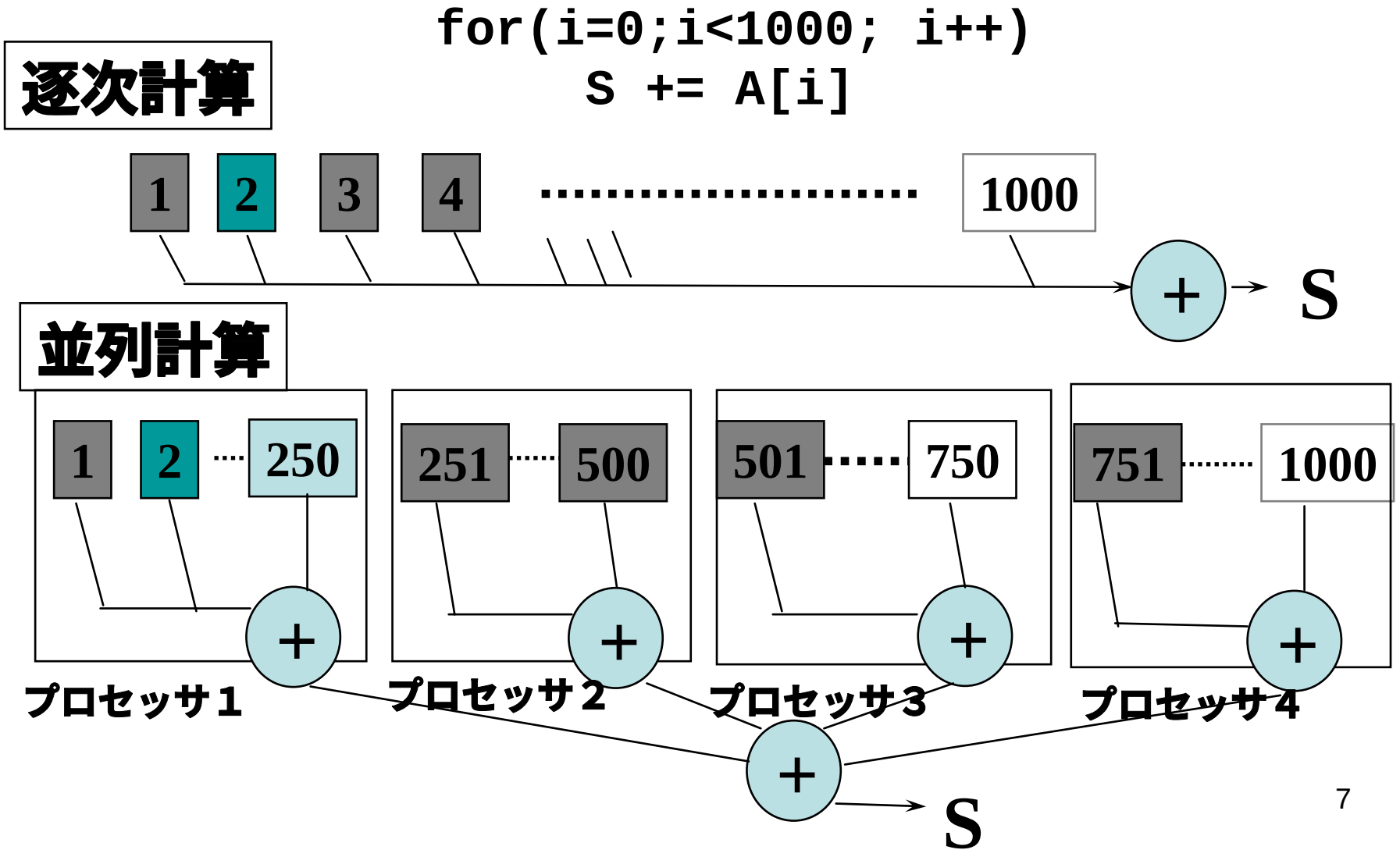
- これらを使いこなすためにはどうすればいいのか？

並列プログラミング・モデル

- **メッセージ通信** (Message Passing)
  - メッセージのやり取りでやり取りをして、プログラムする
  - 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

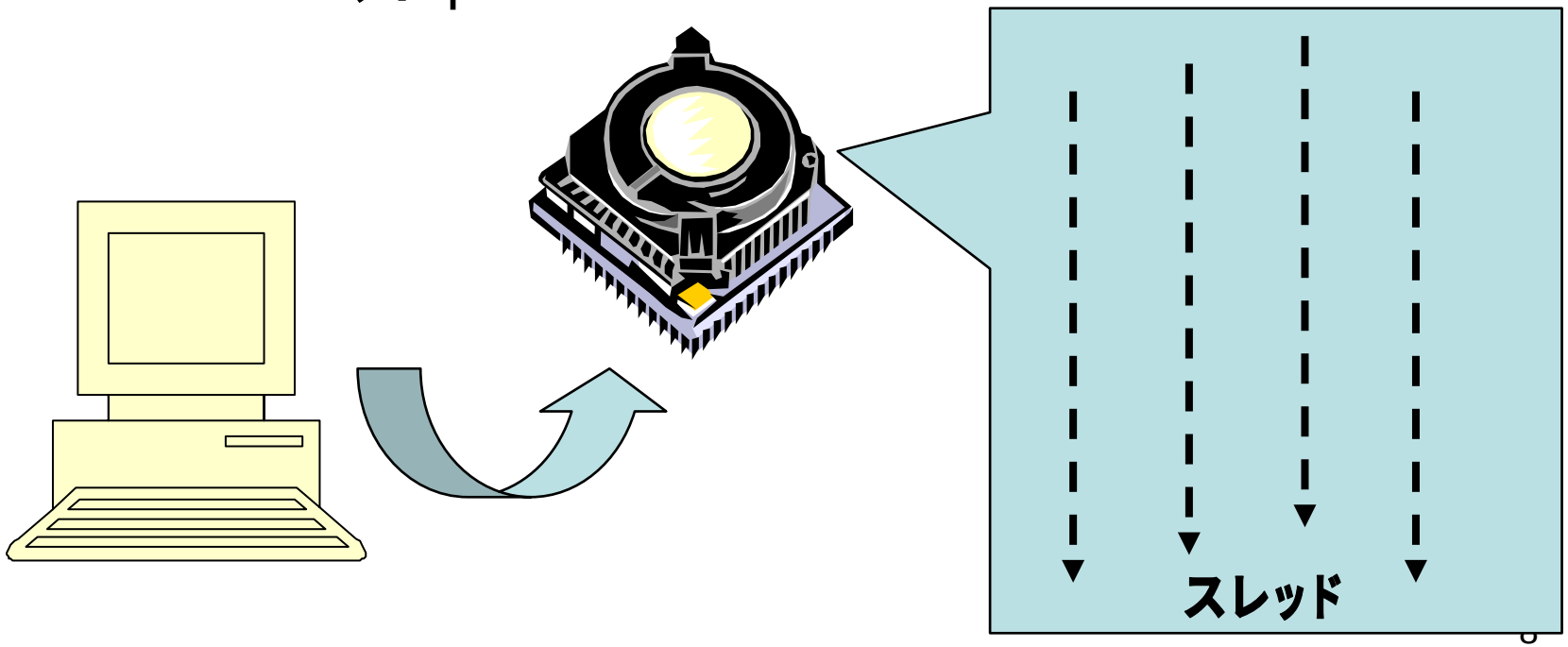
- **共有メモリ** (shared memory)
  - 共通にアクセスできるメモリを解して、データのやり取り
  - 共有メモリシステム (DSMシステムon分散メモリ)
  - プログラミングしやすい (逐次プログラムから)
  - システムがデータの移動を行ってくれる
  - プロセッサ数に対してスケーラブルではないことが多い。

並列処理の簡単な例


for(i=0;i<1000; i++)
S += A[i]
逐次計算
1
2
3
4
1000
+
S
並列計算
1
2
250
251
500
501
750
751
1000
+
プロセッサ１
プロセッサ２
プロセッサ３
プロセッサ４
+
S

マルチスレッドプログラミング

- スレッド
  - 一連のプログラムの実行を抽象化したもの
  - 仮想的なプロセッサとしてもちいてもよい
  - プロセスとの違い
  - POSIXスレッド pthread



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();
```

スレッド＝
プログラム実行の流れ

- 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;
}
```

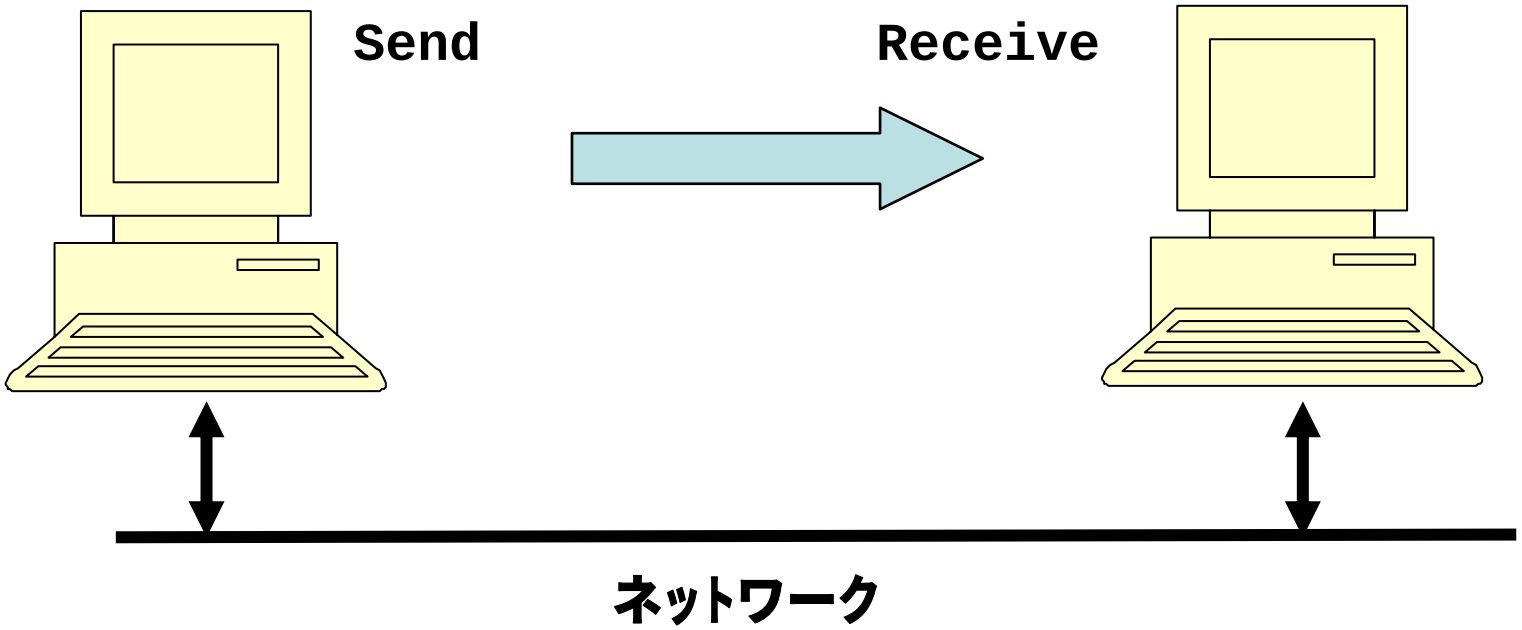
OpenMPによるプログラミング

これだけで、OK!

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

メッセージ通信プログラミング

- sendとreceiveでデータ交換をする
  - MPI (Message Passing Interface)
  - PVM (Parallel Virtual Machine)



メッセージ通信プログラミング

- 1000個のデータの加算の例

```
int a[250]; /* それぞれ、250個づづデータを持つ */

main(){     /* それぞれのプロセッサで実行される */
   int i,s,ss;
   s=0;
   for(i=0; i<250;i++) s+= a[i]; /*各プロセッサで計算*/
   if(myid == 0){   /* プロセッサ0の場合 */
      for(proc=1;proc<4; proc++){
         recv(&ss,proc);  /*各プロセッサからデータを受け取る*/
         s+=ss;   /*集計する*/
      }
   } else {  /* 0以外のプロセッサの場合 */
      send(s,0);  /* プロセッサ0にデータを送る */
   }
}
```

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

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;
}
```

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
- URL
  - `http://www.openmp.org/`

OpenMPの背景

- **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･プロセッサが主流に！**

- **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**

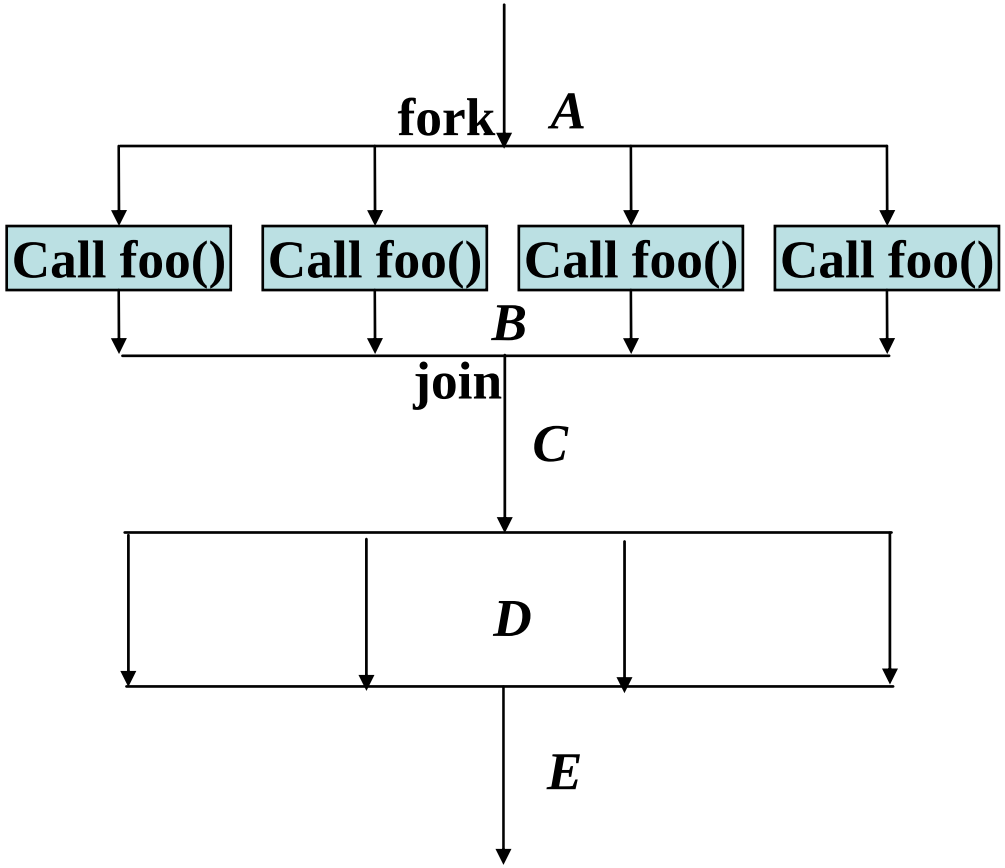
- **自動並列化ではない！**
  - **並列実行・同期をプログラマが明示**

- **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…
    …
}
```

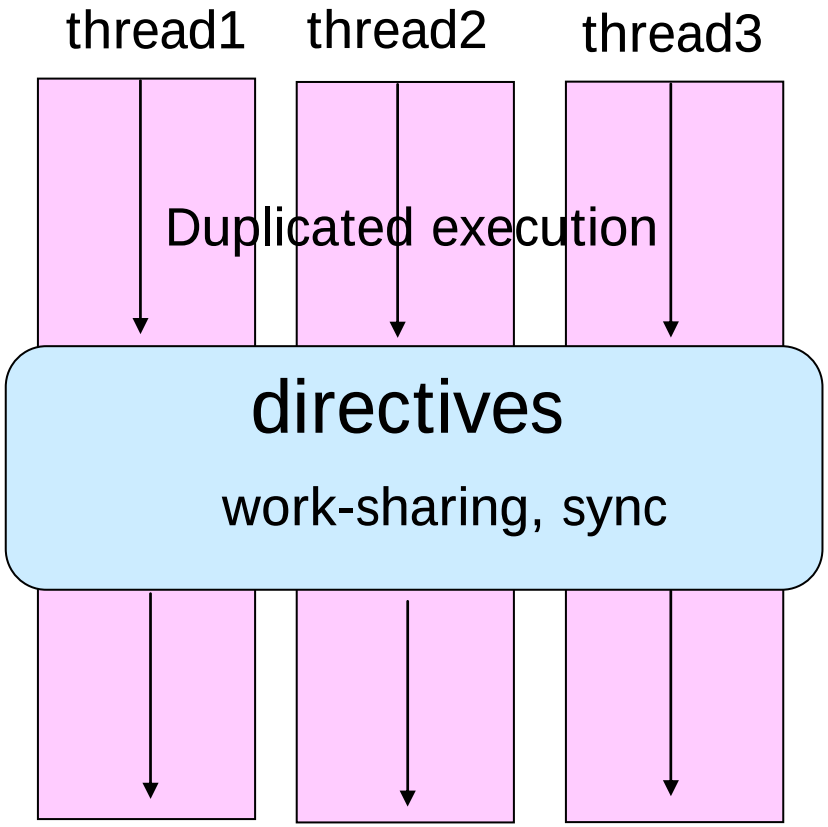
簡単なデモ

- **プロセッサの確認** /proc/cpuinfo
- gcc –fopenmp, gcc**は、**4.2**からサポート**, g95?
- **簡単なプログラム**
- **プロセッサ数は環境変数**OMP_NUM_THREADS**で制御**

```
#include <omp.h>
#include <stdio.h>

main()
{
    printf("omp-test ... n_thread=%d¥n",omp_get_max_threads());
#pragma omp parallel
    {
            printf("thread (%d/%d)...¥n",
                omp_get_thread_num(),omp_get_num_threads());
    }
    printf("end...¥n");
}
```

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

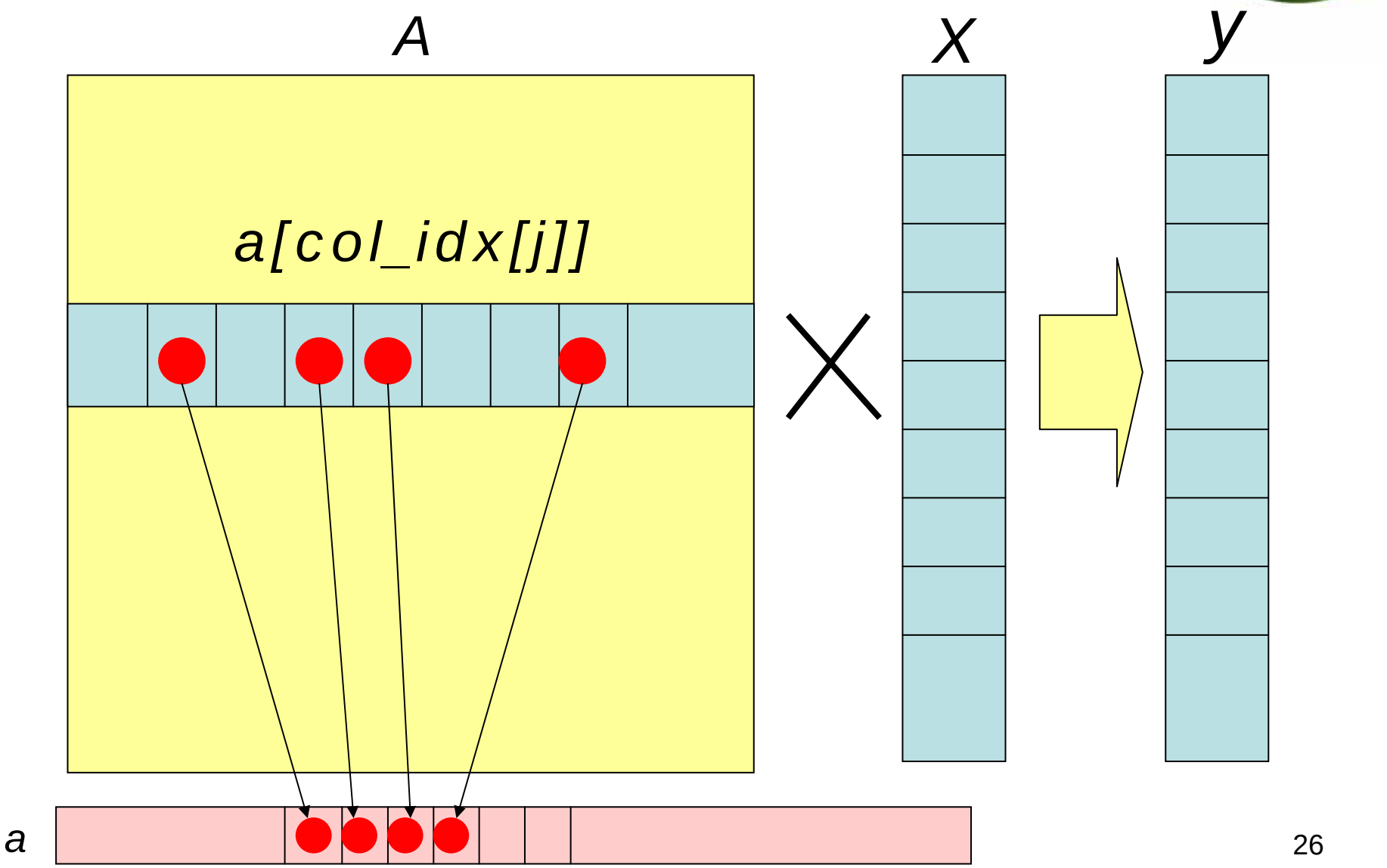
```
#pragma omp for [clause...]
 for(var=lb; var logical-op ub; incr-expr)
    body
```

  - *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

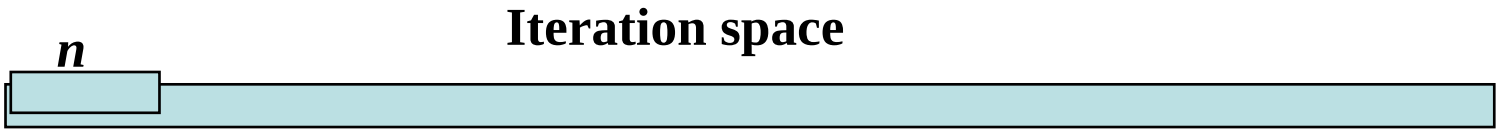
```
Matvec(double a[],int row_start,int col_idx[],
 double x[],double y[],int n)
{
   int i,j,start,end; double t;
#pragma omp parallel for private(j,t,start,end)
   for(i=0; i<n;i++){
      start=row_start[i];
      end=row_start[i+1];
      t = 0.0;
      for(j=start;j<end;j++)
         t += a[j]*x[col_idx[j]];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


A
x
y
a[col_idx[j]]
a

並列ループのスケジューリング

- プロセッサ数4の場合

逐次



schedule(static,*n*)

Schedule(static)

Schedule(dynamic,*n*)

Schedule(guided,*n*)

どのようなときに使い分けをするのかを考えてみましょう。

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
```

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
  - **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**

OpenMP版

```
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a )
{
    return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
    int n, i;
    double PI25DT = 3.14159265358979323846264643;
    double pi, h, sum, x;

    scanf(“%d",&n);
    h   = 1.0 / (double) n;
    sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
    for (i = 1; i <= n; i++){
       x = h * ((double)i - 0.5);
       sum += f(x);
    }
    pi = h * sum;
    printf("pi is approximately %.16f, Error is %.16f¥n",
           pi, fabs(pi - PI25DT));
    return 0;
}
```

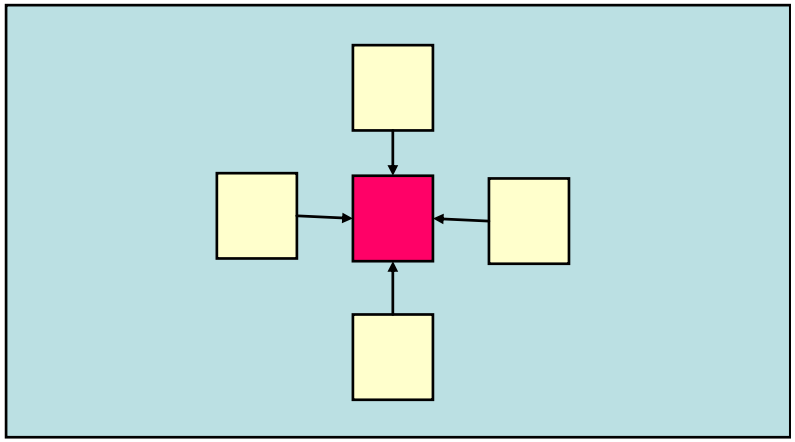
MPI版

```
/* cpi mpi version */
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f( double );
double f( double a )
{
    return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    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);

   if(mypid == 0) scanf("%d",&n);
    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);

    if (myid == 0){
            printf("pi is approximately %.16f, Error is %.16f¥n",
                   pi, fabs(pi - PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```

```
…
 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);
```

OpenMP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

```
/*
 * Laplace equation with explict method
 */
#include <stdio.h>
#include <math.h>

/* square region */
#define XSIZE 1000
#define YSIZE 1000
#define PI 3.1415927
#define NITER 100

double u[XSIZE+2][YSIZE+2],uu[XSIZE+2][YSIZE+2];

double time1,time2;
double second();

void initialize();
void lap_solve();

main()
{
    initialize();

    time1 = second();
    lap_solve();
    time2 = second();

    printf("time=%g¥n",time2-time1);
    exit(0);
}
```

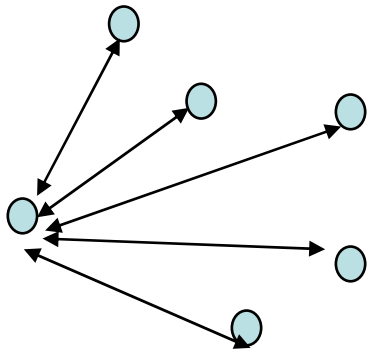
```
void lap_solve()
{
    int x,y,k;
    double sum;

#pragma omp parallel private(k,x,y)
{
    for(k = 0; k < NITER; k++){
        /* old <- new */
#pragma omp for
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            uu[x][y] = u[x][y];
        /* update */
#pragma omp for
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
    }
 }

/* check sum */
    sum = 0.0;
#pragma omp parallel for private(y) reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
        for(y = 1; y <= YSIZE; y++)
          sum += (uu[x][y]-u[x][y]);
    printf("sum = %g¥n",sum);
}
```

```
void initialize()
{
    int x,y;

    /* initalize */
    for(x = 1; x <= XSIZE; x++)
      for(y = 1; y <= YSIZE; y++)
        u[x][y] = sin((double)(x-1)/XSIZE*PI) + cos((double)(y-1)/YSIZE*PI);

    for(x = 0; x < (XSIZE+2); x++){
        u[x][0] = 0.0;
        u[x][YSIZE+1] = 0.0;
        uu[x][0] = 0.0;
        uu[x][YSIZE+1] = 0.0;
    }

    for(y = 0; y < (YSIZE+2); y++){
        u[0][y] = 0.0;
        u[XSIZE+1][y] = 0.0;
        uu[0][y] = 0.0;
        uu[XSIZE+1][y] = 0.0;
    }
}
```

粒子計算の例

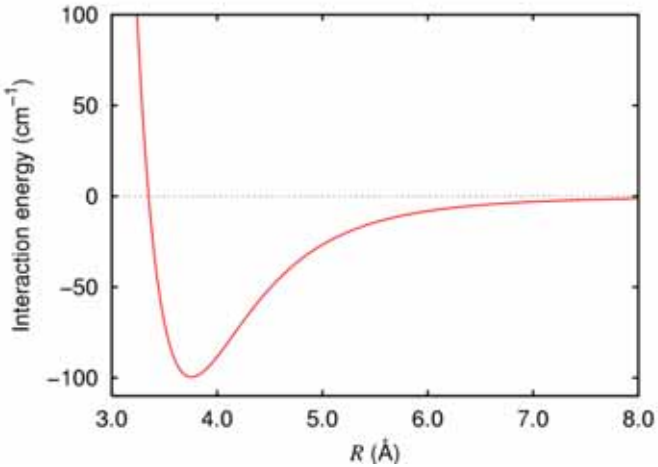
- 個々の粒子の相互作用を計算する
- 宇宙：重力
  - $F_{ij} = G\, m_i\, m_j / r^2$
- 分子シミュレーション：Van der Waals forces
  - 分子動力学
    - 2体（或いはそれ以上）の原子間ポテンシャルの下に、古典力学におけるニュートン方程式を解いて、系の静的、動的安定構造や、動的過程（ダイナミクス）を解析する手法。

$$F_i = \Sigma F_{ij}$$



$$F = m\ a$$
$$V = v + a\Delta t$$
$$p = p + v\Delta t$$

各ステップで
運動方程式
を解く



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

typedef struct my_particle {
    double m;
    double x,y,z;
    double vx,vy,vz;
    double ax,ay,az;
} particle;

double DT;
int n_steps;
int n_particles;
particle *particles;

void do_step();
```

```
int main(int argc, char ** argv)
{
    int problem_no,it,i;
    double *a,*ap;
    particle *p;

    … 初期化…
    … データのセットアップ…

#pragma omp parallel private(it)
{
    for(it = 0; it < n_steps; it++){
        do_step();
    }
}

    … 後処理…

}
```

```
void do_step()
{
    int i,j;
    double a2 = 256.0;
    double b2 = 1024.0;
    double ax,ay,az,dx,dy,dz,X,f;
    particle *p,*q;

#pragma omp for
    for(i = 0; i < n_particles; i++) {
        p = &particles[i];
        ax = 0.0; ay = 0.0; az = 0.0;
        for(j = 0; j < n_particles; j++){
            if(i == j) continue;
            q = &particles[j];
            dx = p->x - q->x;
            dy = p->y - q->y;
            dz = p->z - q->z;
            f = ここで、forceの計算…
             ax += f * dx;
             ay += f * dy;
             az += f * dz;
            }
        }
        p->ax = ax;
        p->ay = ay;
        p->az = az;
    }
```

```
#pragma omp for
    for(i = 0; i < n_particles; i++){
        p = &particles[i];
        p->x += p->vx * DT;
        p->y += p->vy * DT;
        p->z += p->vz * DT;
        p->vx += p->ax * DT;
        p->vy += p->ay * DT;
        p->vz += p->az * DT;
    }
}
```

では、性能は？

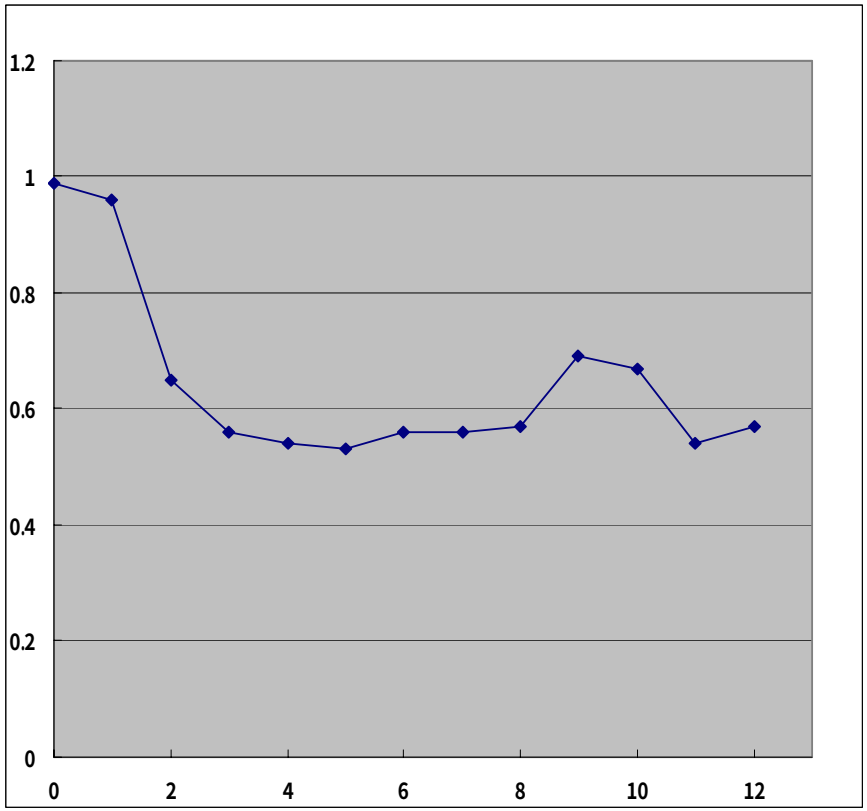
- プラットフォーム、問題規模による
- 特に、問題規模は重要
  - 並列化のオーバーヘッドと並列化のgainとのトレードオフ

- Webで探してみてください。
- 自分でやって、みてください。

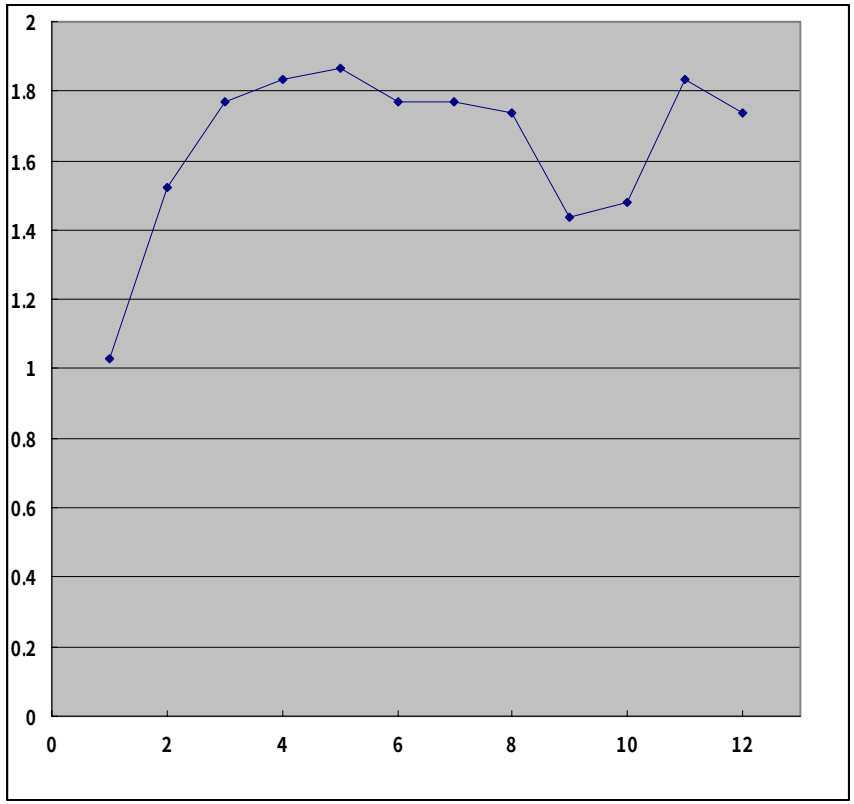
Laplaceの性能

XSIZE=YSIZE=1000

実行時間



対逐次性能比

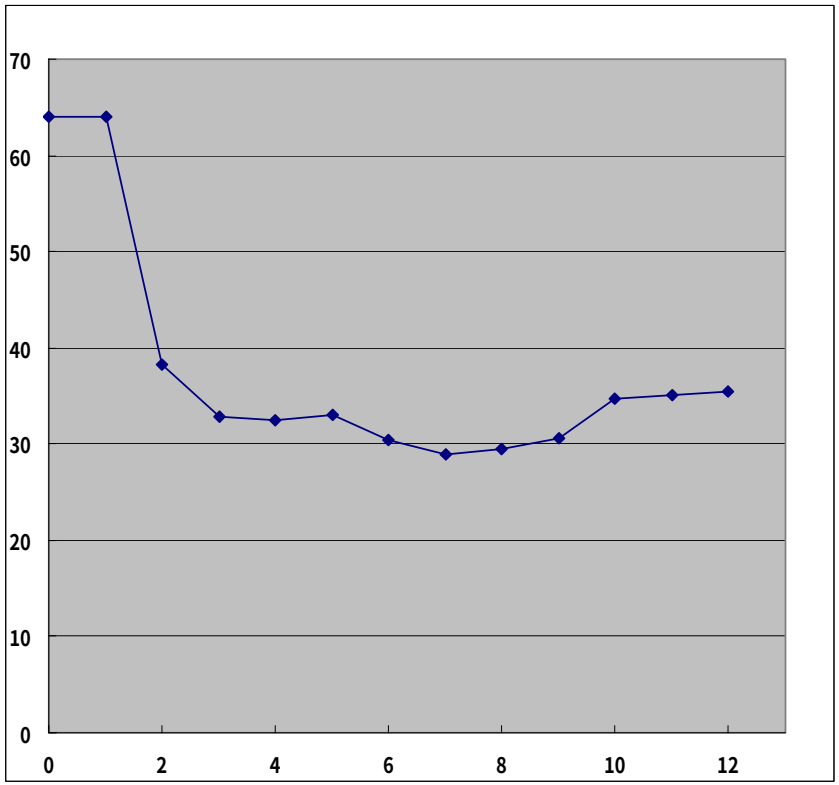


AMD Opteron quad , 2 socket

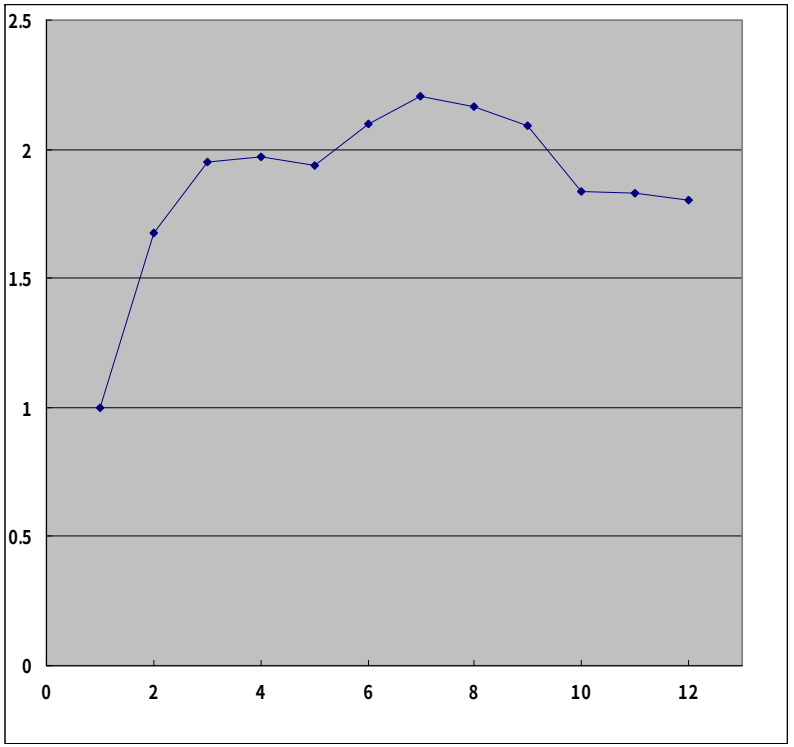
Laplaceの性能

XSIZE=YSIZE=8000

実行時間



対逐次性能比

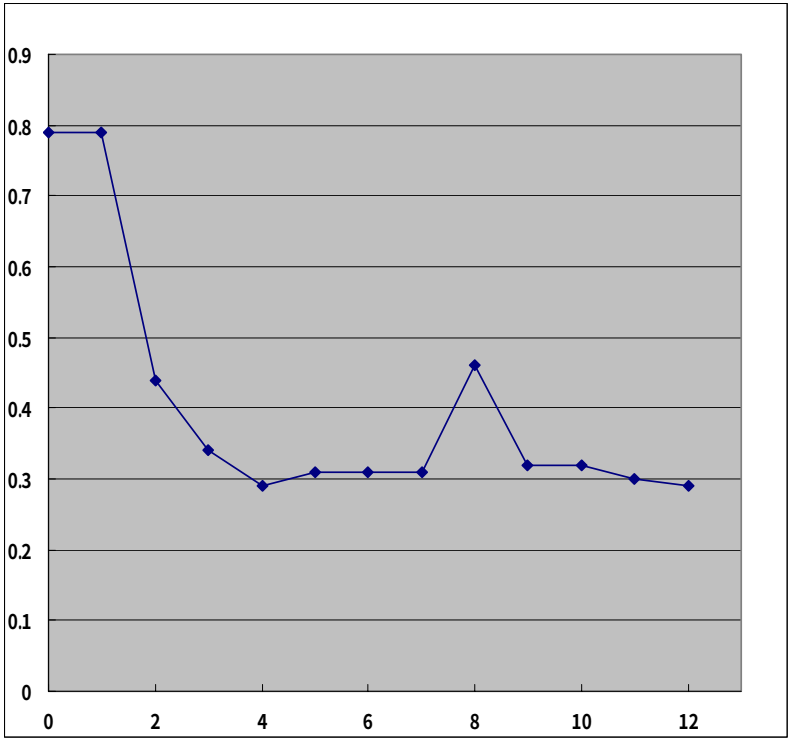


AMD Opteron quad , 2 socket

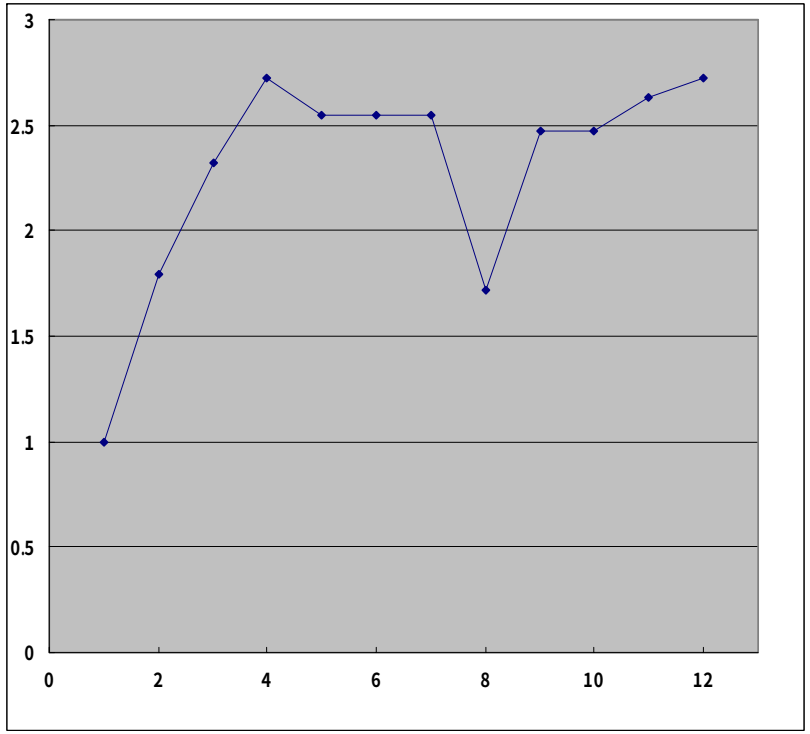
Laplaceの性能

XSIZE=YSIZE=1000

実行時間



対逐次性能比

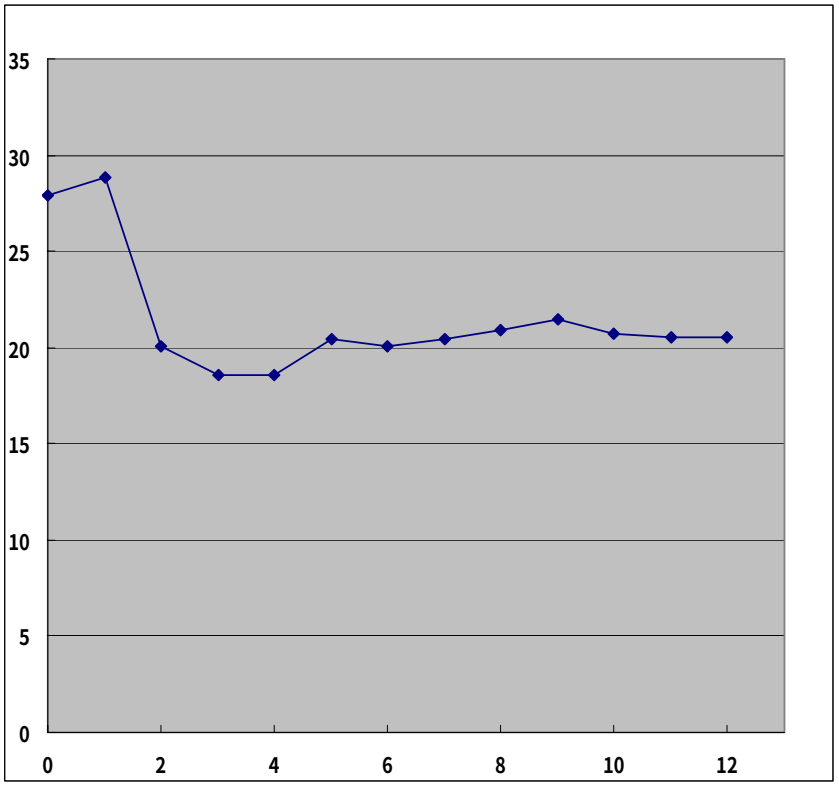


Core i7 920 @ 2.67GHz, 2 socket

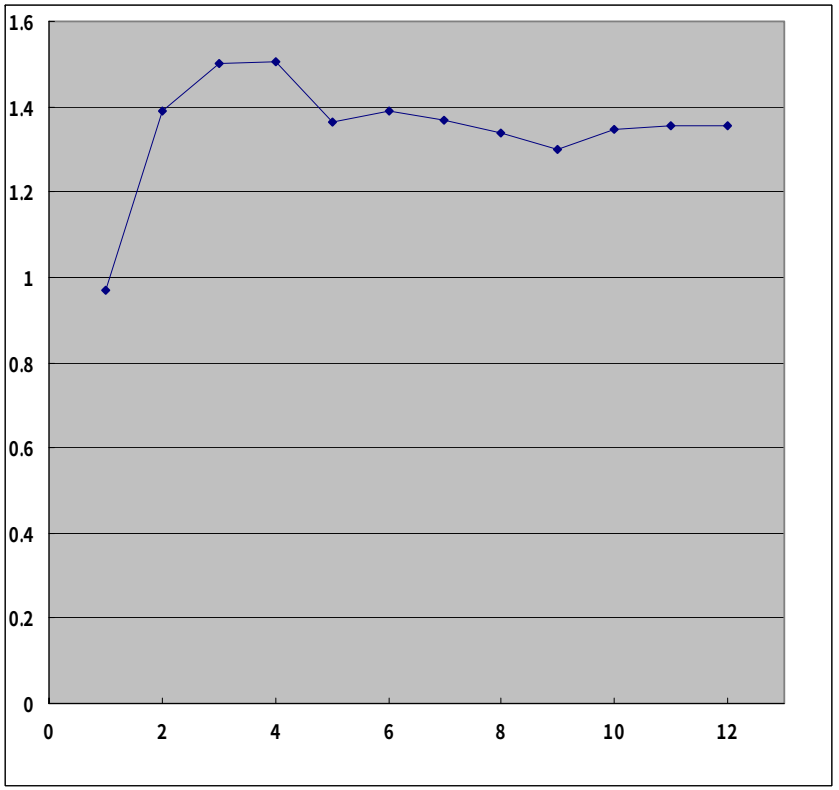
Laplaceの性能

XSIZE=YSIZE=8000

実行時間



対逐次性能比



Core i7 920 @ 2.67GHz, 2 socket

Advanced topics

- OpenMP 3.0
  - 2007**年に**approve**された**
- MPI/OpenMP Hybrid Programming
  - SMP**クラスタでのプログラミング**

OpenMP3.0で追加された点

Openmp.orgに富士通訳の日本語バージョンの仕様書がある

- タスクの概念が追加された
  - Parallel 構文とTask構文で生成されるスレッドの実体
  - task構文
  - taskwait構文
- メモリモデルの明確化
  - Flushの扱い
- ネストされた場合の定義の明確化
  - Collapse指示節
- スレッドのスタックサイズの指定
- C＋＋でのprivate変数に対するconstructor, destructorの扱い

Flushの例

誤った例

a = b = 0

スレッド1	スレッド2
b = 1	a = 1
flush(b)	*flush*(a)
flush(a)	*flush*(b)
if (a == 0) then	if (b == 0) then
critical section	*critical section*
end if	end if

正しい例

a = b = 0

スレッド1	スレッド2
b = 1	a = 1
flush(a,b)	*flush*(a,b)
if (a == 0) then	if (b == 0) then
critical section	*critical section*
end if	end if

Task構文の例

外側にparallel **構文が必要**

```
struct node {
        struct node *left;
        struct node *right;
};

void postorder_traverse( struct node *p ) {
        if (p->left)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->left);
        if (p->right)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->right);
        #pragma omp taskwait
        process(p);
}
```

Stephen Olivier, Jan Prins, Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs, presented in IWOMP 2009

Example Tree: Unbalanced Tree Search (UTS)

- Sample tree of 7079 nodes with depth of 142 nodes
- Experiments in the paper use a much larger tree
  - Size > 4M nodes
  - Depth > 1500 nodes
  - Imbalance such that > 99% of nodes are in a single subtree below the root node

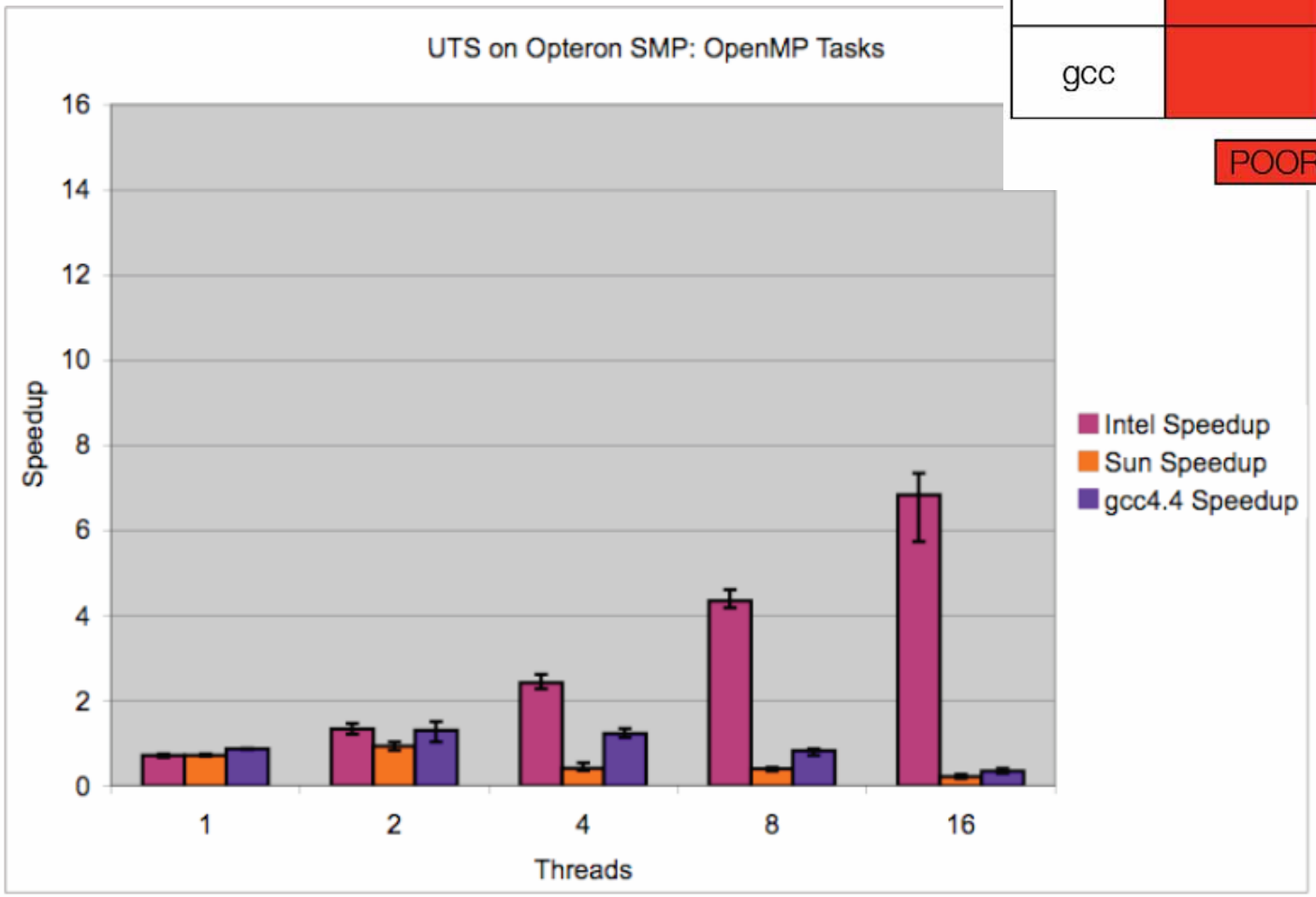
```
void Generate_and_Traverse(Node parentNode, int childNumber)
{
   Node currentNode = generateID(parentNode, childNumber);
   nodeCount++;    // threadprivate, combined at termination
   int numChildren = m with prob q, 0 with prob 1-q
   for (i = 0; i < numChildren; i++) {
      #pragma omp task firstprivate(i)
         Generate_and_Traverse(currentNode, i);
   }
   #pragma omp taskwait
}
```

Stephen Olivier, Jan Prins, Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs, presented in IWOMP 2009

Summary of Findings

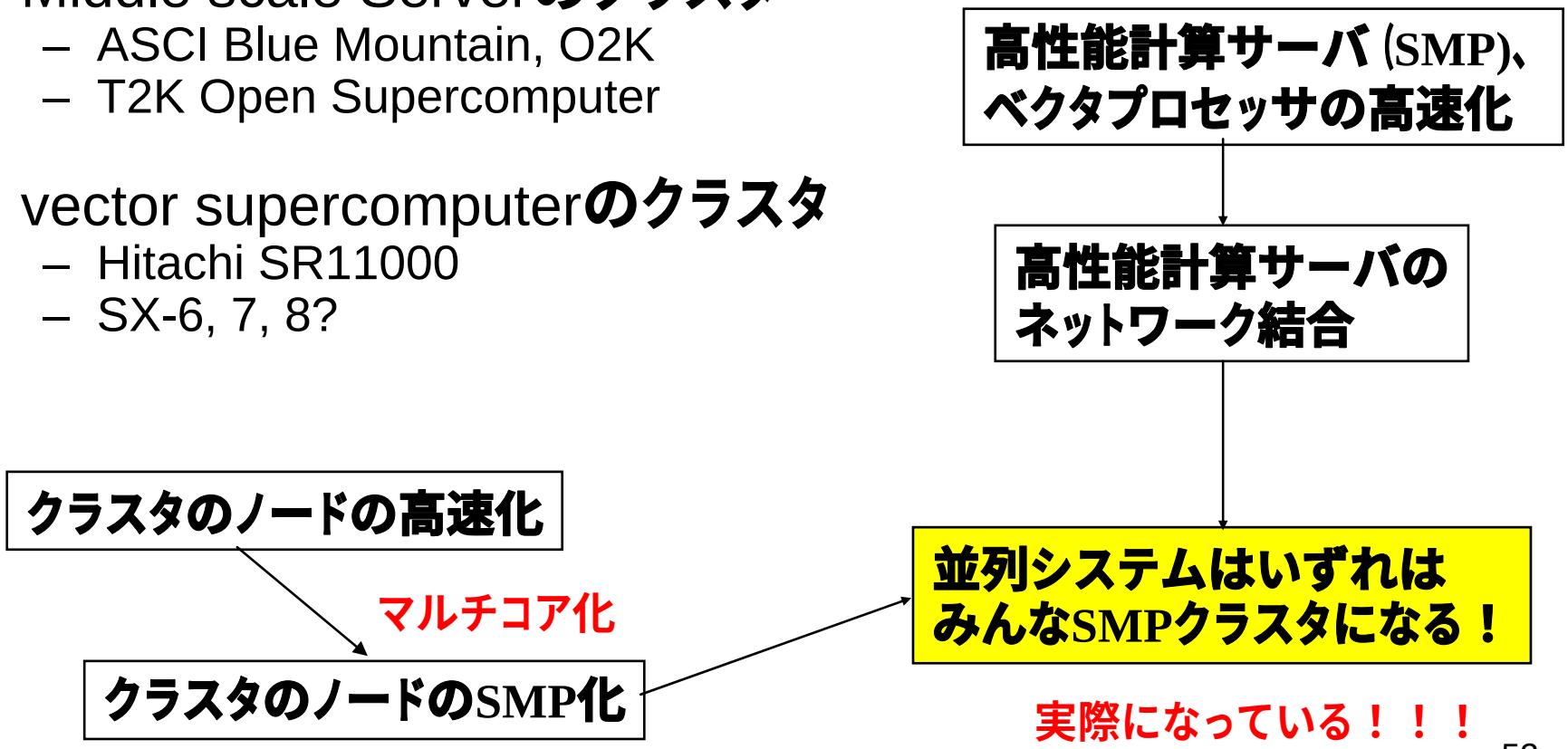
Compiler and Run Time	Scalability on UTS Problem	Achieved Load Balance	Attempted Load Balance	Overhead Costs
Intel				
Sun				
gcc				

POOR FAIR GOOD



SMPクラスタ・マルチコアクラスタ

- 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Hybrid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？）

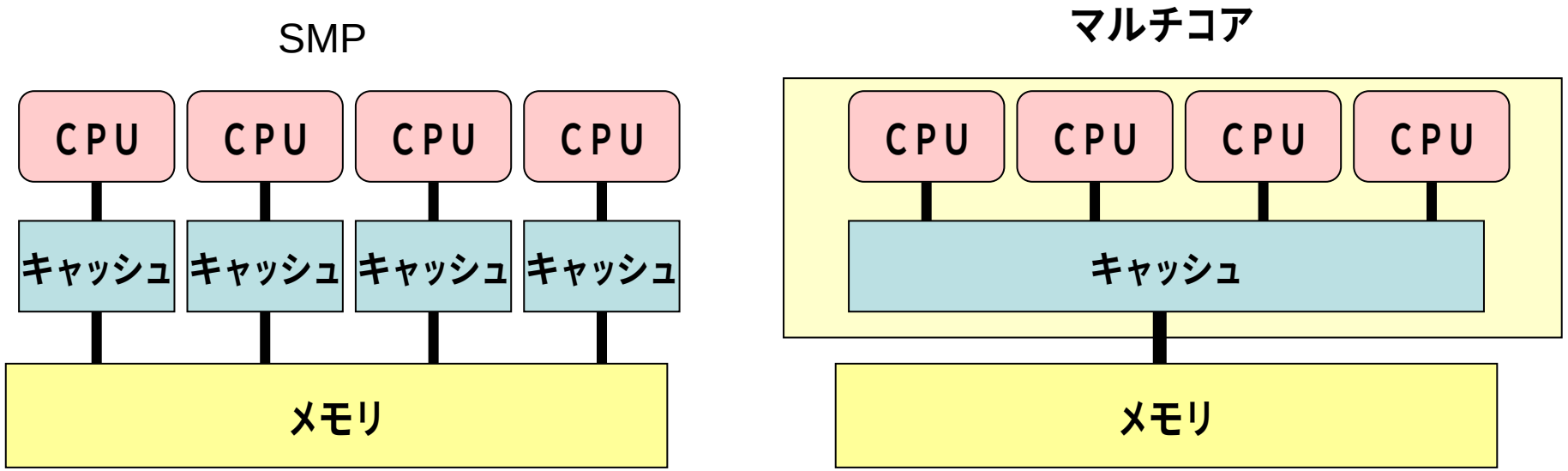
Thread-safety of MPI

- MPI_THREAD_SINGLE
  - A process has only one thread of execution.
- MPI_THREAD_FUNNELED
  - A process may be multithreaded, but only the thread that initialized MPI can make MPI calls.
- MPI_THREAD_SERIALIZED
  - A process may be multithreaded, but only one thread at a time can make MPI calls.
- MPI_THREAD_MULTIPLE
  - A process may be multithreaded and multiple threads can call MPI functions simultaneously.
- MPI_Init_thread **で指定、サポートされていない可能性もある**

SMPとマルチコア

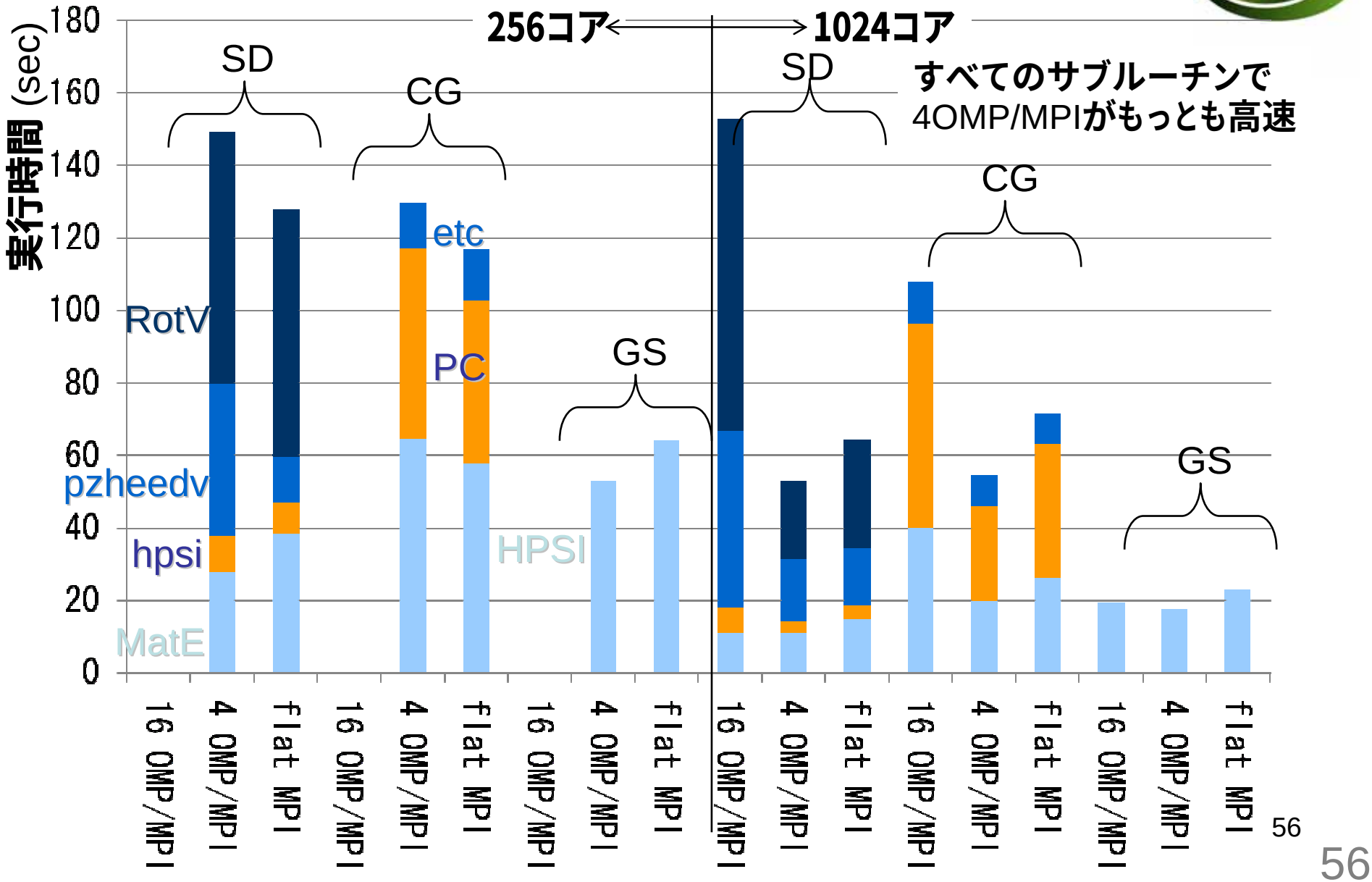
- 必ずしも、Hybridプログラムは速くなかった
  - “flat-MPI” (SMPの中でもMPI) が早い
  - 利点
    - データが共有できる⇒メモリを節約
    - 違うレベルの並列性を引き出す

- しかし、マルチコアクラスタではHybridが必要なケースが出てくる
  - キャッシュが共有される



RS-DFT on T2K の例

辻美和子、佐藤三久、大規模 SMP クラスタにおける OpenMP/MPI ハイブリッド NPB，RSDFT の評価、情報処理学会研究会報告2009-HPC-119、pp. 163-168, 2009.



56

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
- マルチコアプロセッサでは、必須
- １６プロセッサ以上になれば、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…

課題

- ナップサック問題を解く並列プログラムをOpenMPを用いて作成しなさい。
  - ナップサック問題とは、いくつかの荷物を袋に最大の値段になるように袋に詰める組み合わせを求める問題
  - N個の荷物があり、個々の荷物の重さをw_i、値段をp_iとする。袋(knapsack)には最大Wの重さまでいれることができる。このとき、袋にいれることができる荷物の組み合わせを求め、そのときの値段を求めなさい。
  - 求めるのは、最大の値段だけでよい。(組み合わせは求めなくてもよい)
  - 注意：Task構文は使わないこと
  - ヒント：幅探索にする。

例

```
#define MAX_N 100
int N; /*データの個数*/
int Cap; /*ナップサックの容量*/
int W[MAX_N]; /* 重さ */
int P[MAX_N]; /* 価値 */

int main()
{
   int opt;
   read_data_file(“test.dat”);
   opt = knap_search(0,0,Cap);
   printf(“opt=%d¥n”,opt);
   exit(0);
}
```

```
read_data_file(file)
     char *file;
{
    FILE *fp;
    int i;

    fp = fopen(file,"r");
    fscanf(fp,"%d",&N);
    fscanf(fp,"%d",&Cap);
    for(i = 0; i < N; i++)
      fscanf(fp,"%d",&W[i]);
    for(i = 0; i < N; i++)
      fscanf(fp,"%d",&P[i]);
    fclose(fp);
}
```

逐次再帰版

```
int knap_search(int i,int cp, int M)
{
  int Opt;
  int l,r;

  if (i < N && M > 0){
    if(M >= W[i]){
      l = knap_seach(i+1,cp+P[i],M-W[i]);
      r = knap_serach(i+1,cp,M);
      if(l > r) Opt = l;
      else Opt = r;
    } else
      Opt = knap_search(i+1,cp,M);
  } else Opt = cp;
  return(Opt);
}
```