CCS HPC seminar 2013

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超入門
  - OpenMP
- Openプログラミングの概要
- Advanced Topics
  - SMPクラスタ、Hybrid Programming
  - OpenMP 3.0　(task)
  - OpenMP 4.0
- 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

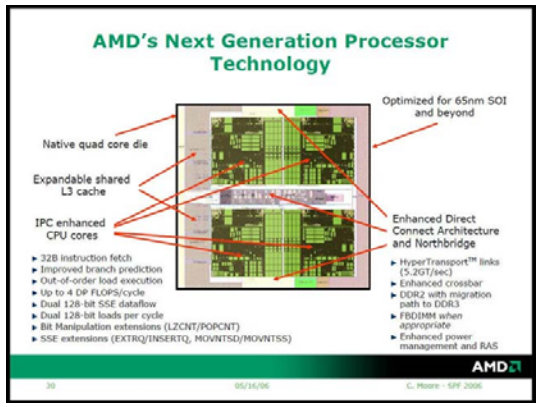
マルチ・コア

共有メモリ
並列コンピュータ

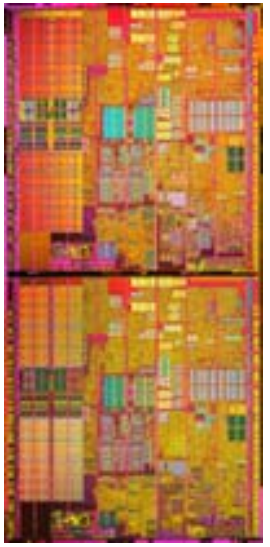
分散メモリ並列コンピュータ
グリッド

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(2001) ⇒ マルチコア
  - プロセスは28nm⇒10nm , 将来的には7nm
    - トランジスタ数は増える！

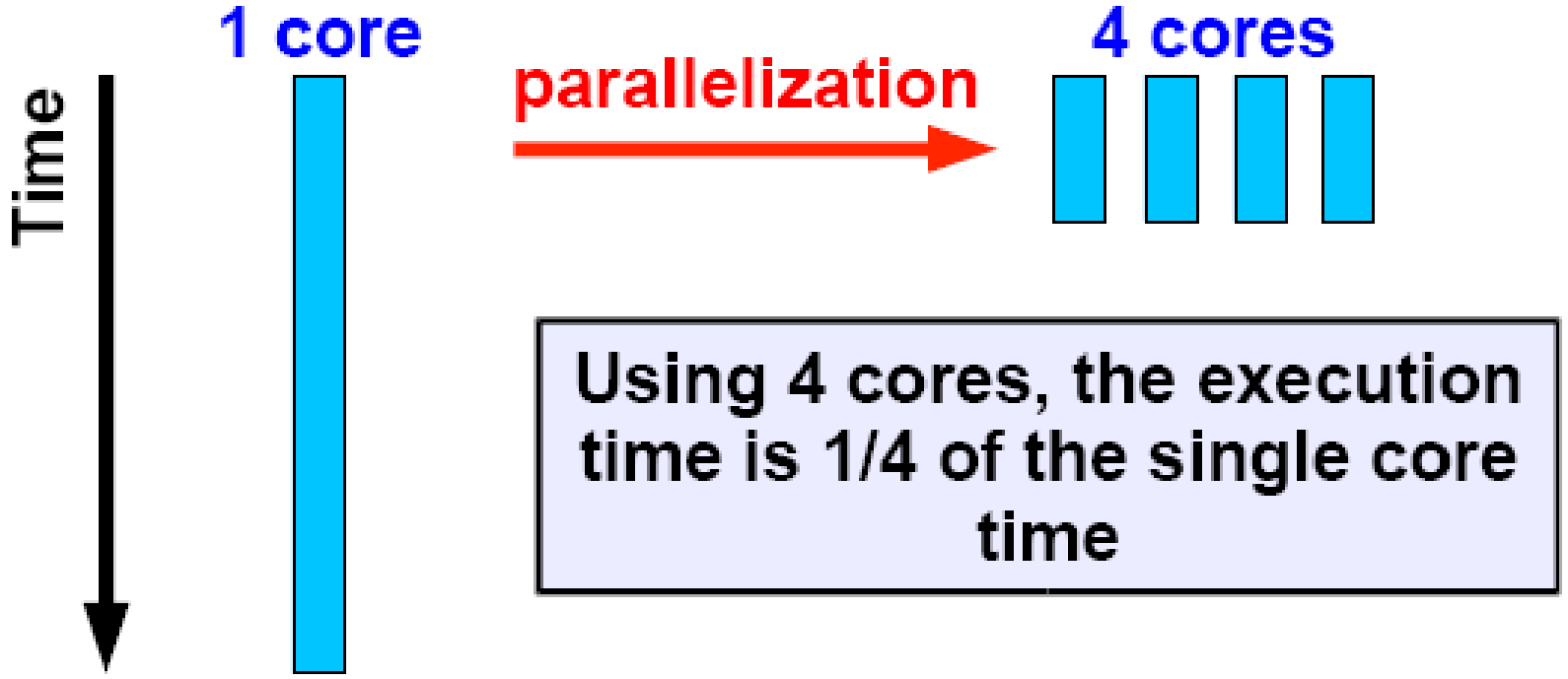


- アーキテクチャの改良
  - スーパーパイプライン、スーパースカラ、VLIW…
  - キャッシュの多段化、マイクロプロセッサでもL3キャッシュ
  - マルチスレッド化、Intel Hyperthreading
    - 複数のプログラムを同時に処理
  - マルチコア：1つのチップに複数のCPU

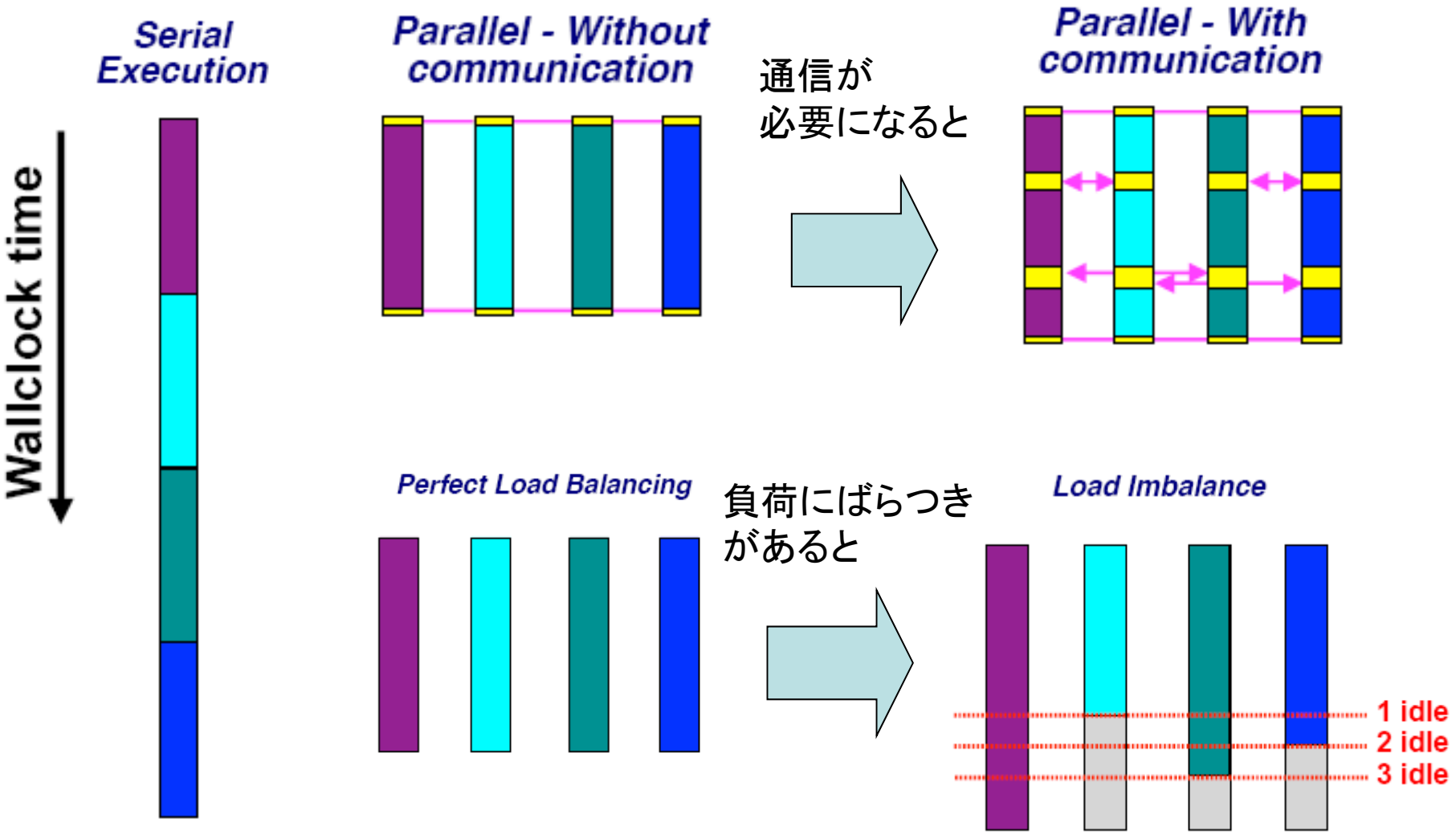


インテル® Pentium® プロセッサ エクストリーム・エディションのダイ

なぜ、並列化するのか？
4つのコアがあれば、4倍！



並列化のオーバヘッド


Serial Execution
Parallel - Without communication
通信が必要になると
Parallel - With communication
Wallclock time
Perfect Load Balancing
負荷にばらつきがあると
Load Imbalance
1 idle
2 idle
3 idle

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

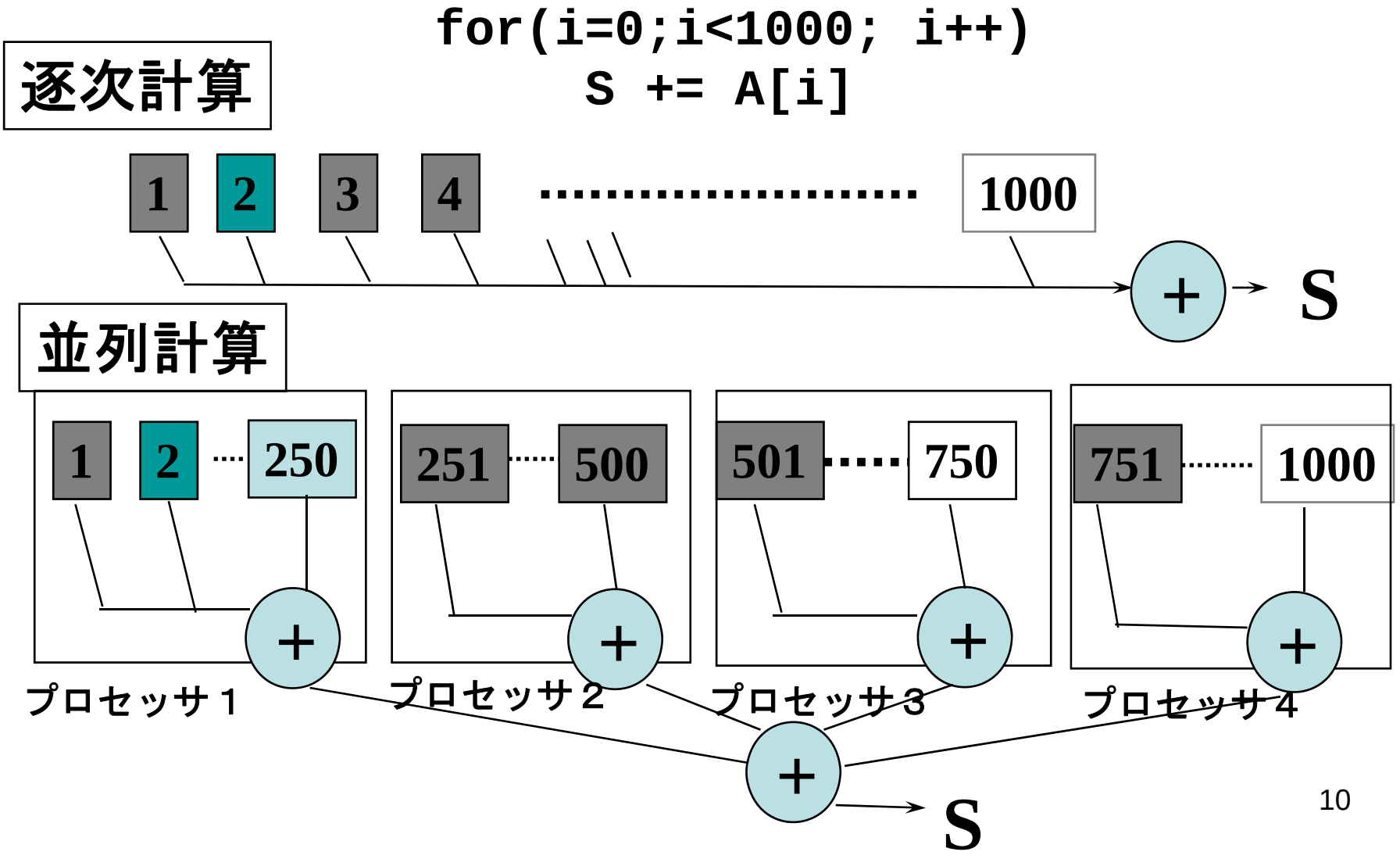
- *There are numerous parallel programming models*
- *The ones most well-known are:*
  - *Distributed Memory*
    - ✔ *Sockets (standardized, low level)*
    - ✔ *PVM - Parallel Virtual Machine (obsolete)*
    - ✔ *MPI - Message Passing Interface (de-facto std)*
  - *Shared Memory*
    - ✔ *Posix Threads (standardized, low level)*
    - ✔ *OpenMP (de-facto standard)*
    - ✔ *Automatic Parallelization (compiler does it for you)*

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

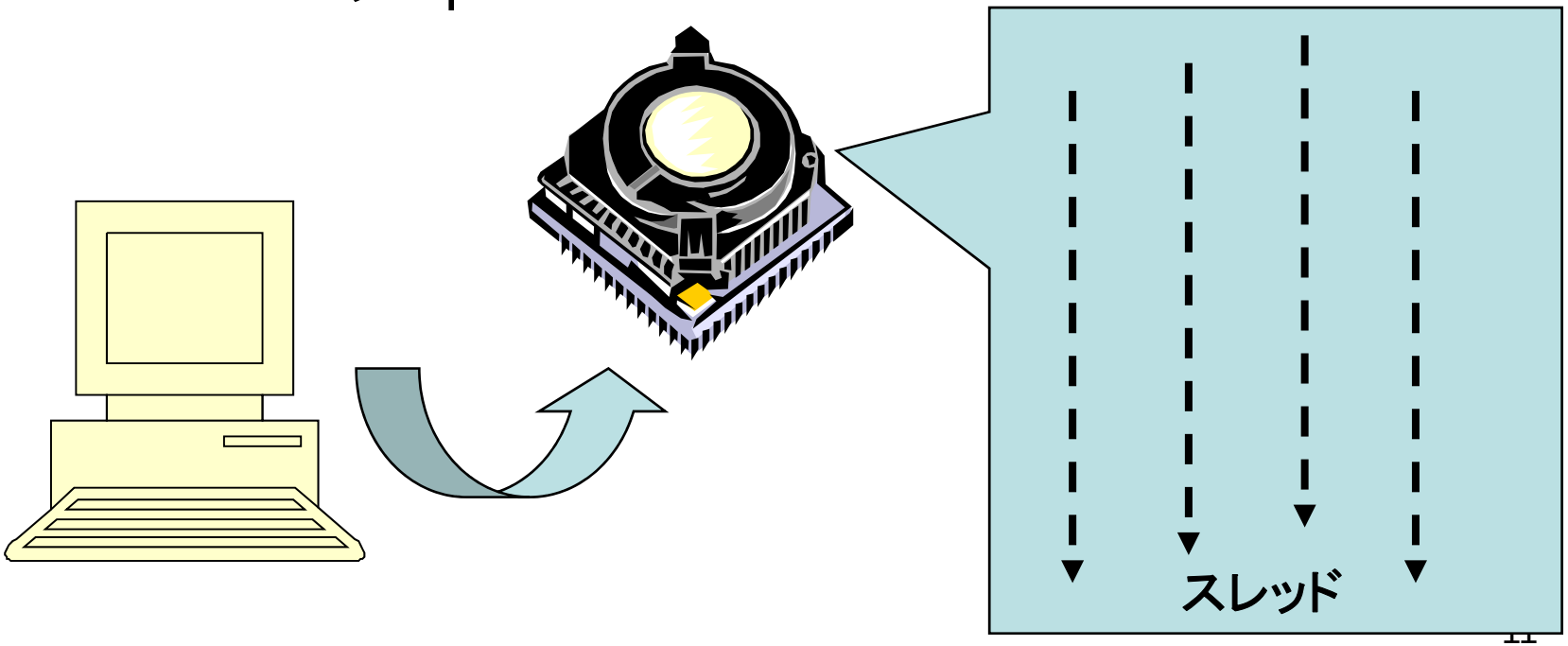
- 共有メモリ (shared memory)
  - 共通にアクセスできるメモリを解して、データのやり取り
  - 共有メモリシステム（DSMシステムon分散メモリ）
  - プログラミングしやすい（逐次プログラムから）
  - システムがデータの移動を行ってくれる
  - プロセッサ数に対してスケーラブルではないことが多い。

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];
```

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 4.0

- 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
  - 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。
- 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

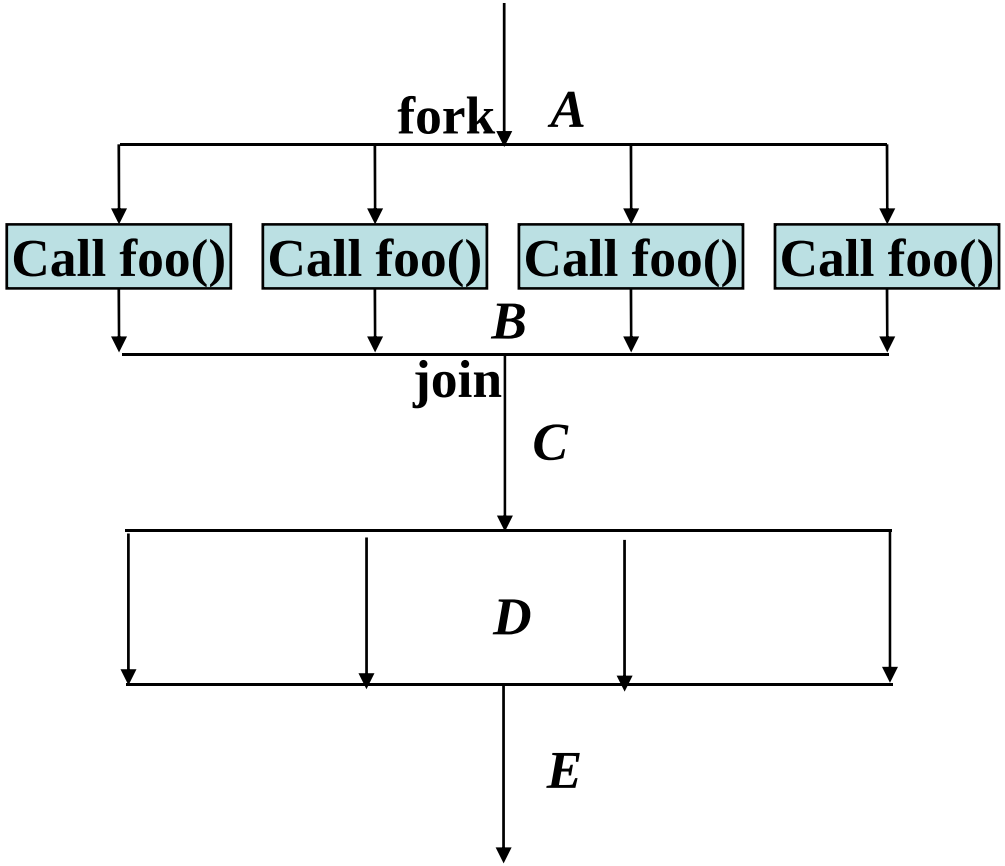
OpenMPのAPI

- 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…
    …
}
```

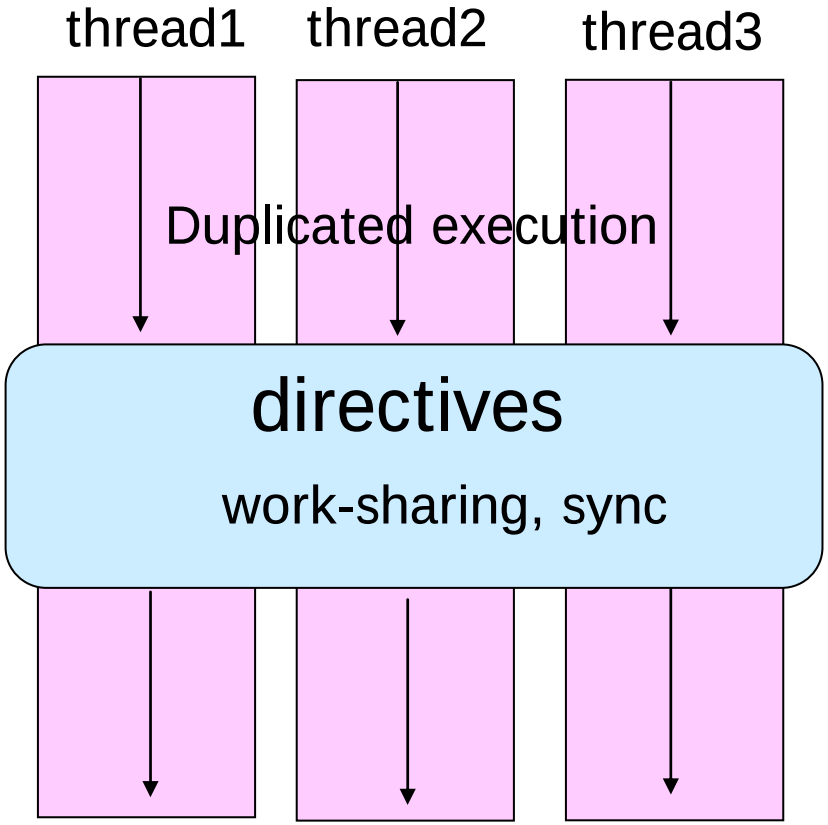
簡単なデモ

- プロセッサの確認 /proc/cpuinfo
- gcc –fopenmp, gccは、4.2からサポート, gfortran
- 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

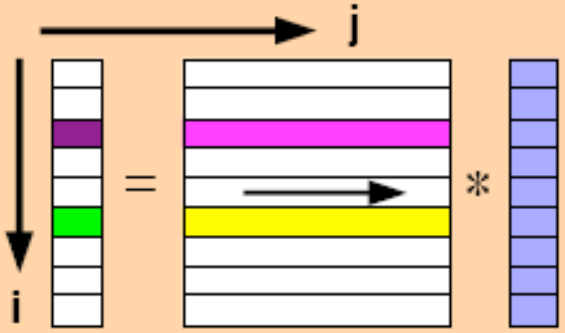
- Forループ（DOループ）のイタレーションを並列実行
- 指示文の直後のforループは*canonical shape*でなくてはならない

```
#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：行列ベクトル積

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

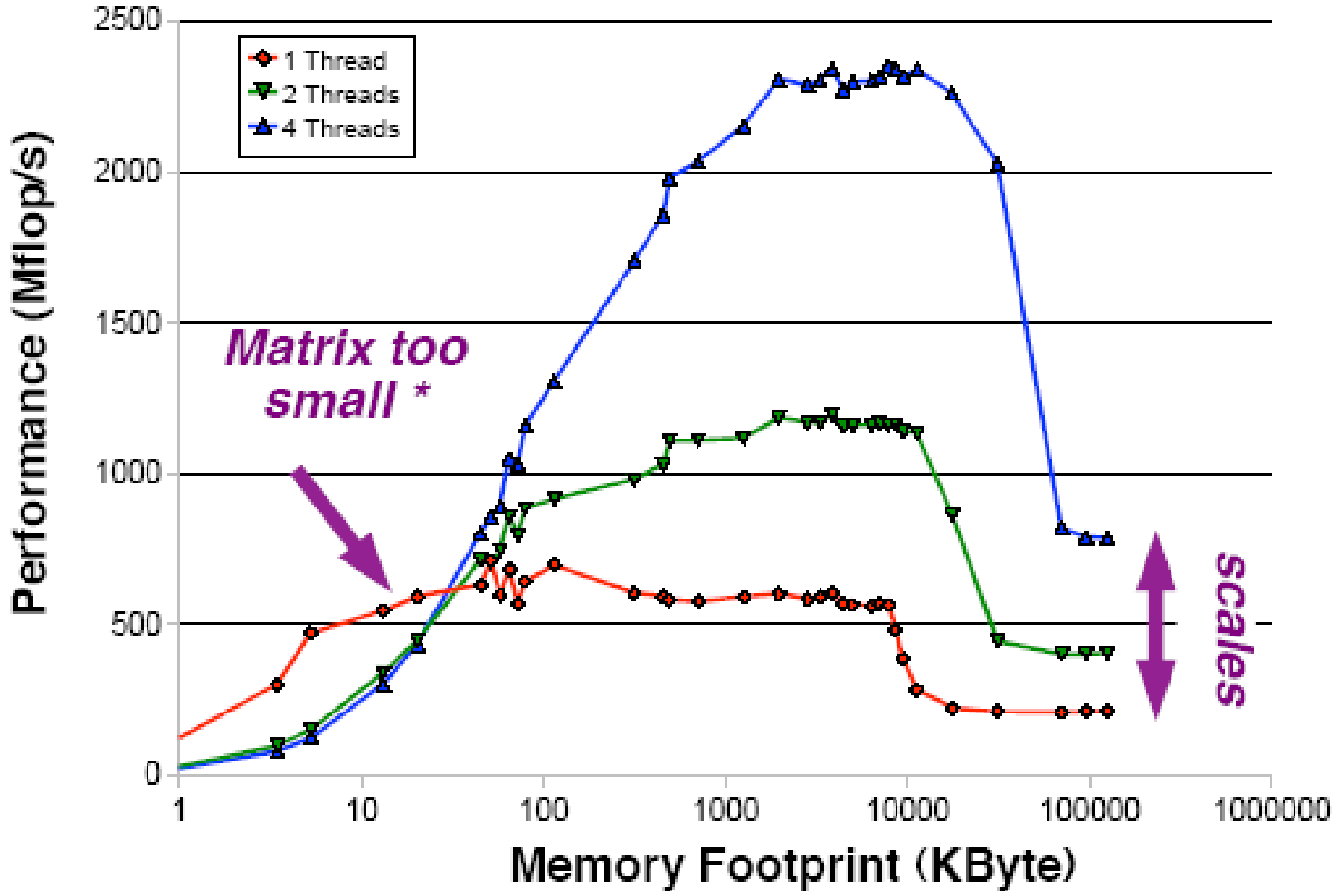
```
for (i=0,1,2,3,4)
 i = 0
   sum = Σ b[i=0][j]*c[j]
   a[0] = sum
 i = 1
   sum = Σ b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

```
for (i=5,6,7,8,9)
 i = 5
   sum = Σ b[i=5][j]*c[j]
   a[5] = sum
 i = 6
   sum = Σ b[i=6][j]*c[j]
   a[6] = sum
```

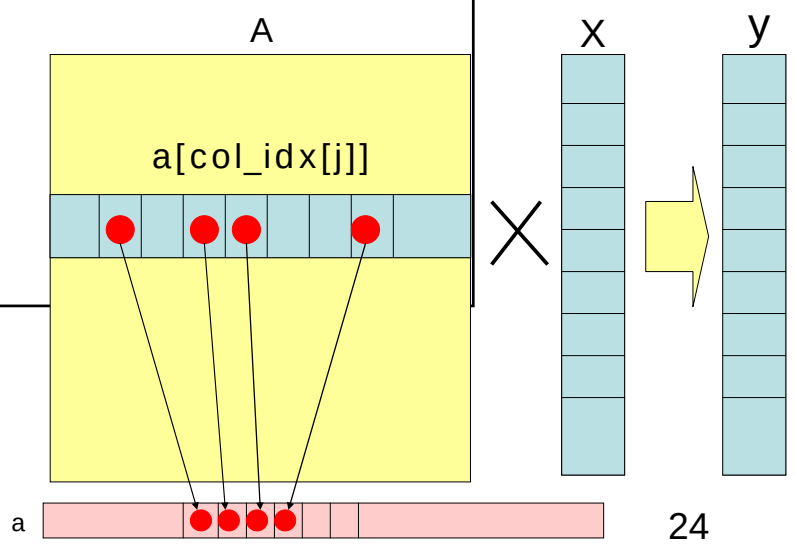
... etc ...

だいたい、性能はこうなる



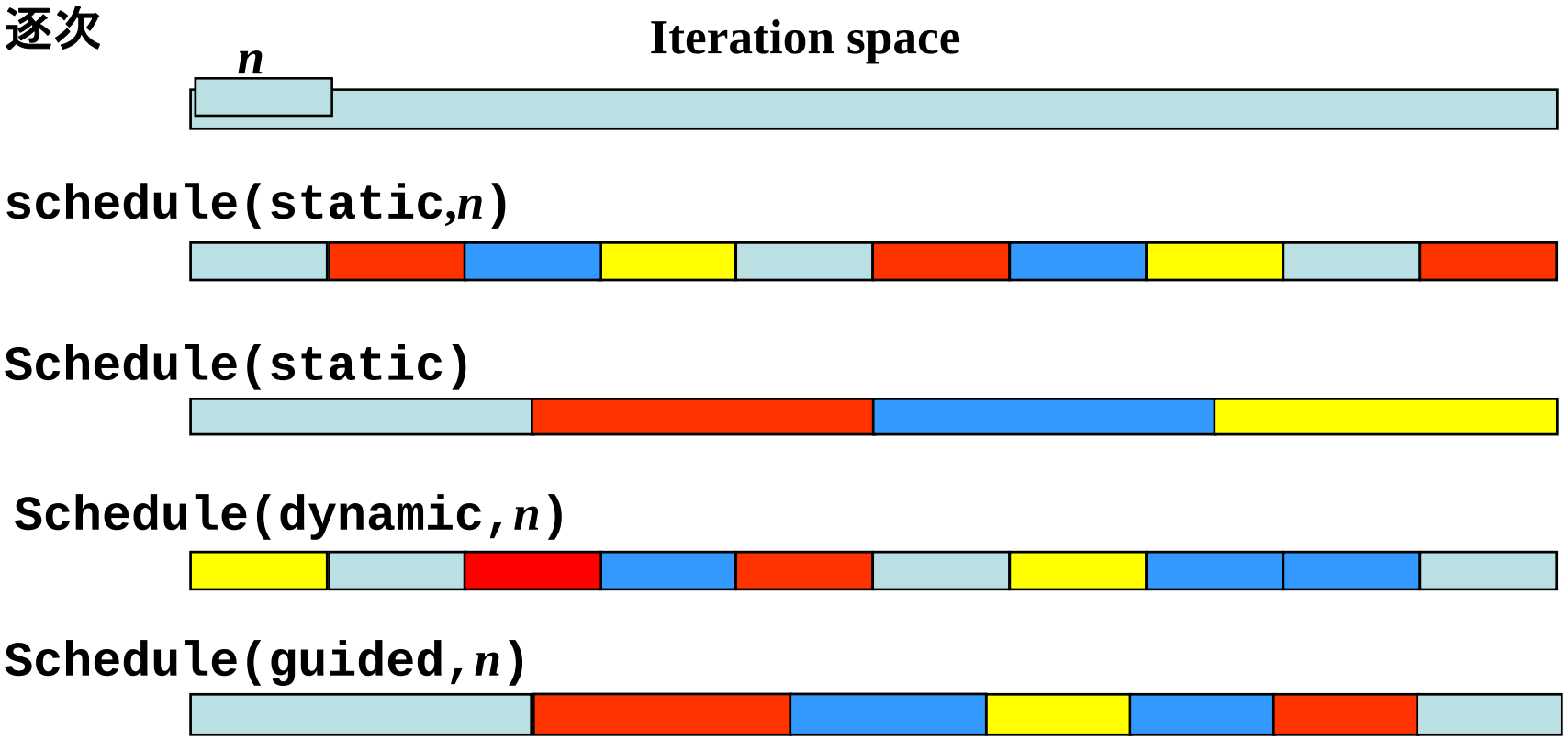
例：疎行列ベクトル積ルーチン

```
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

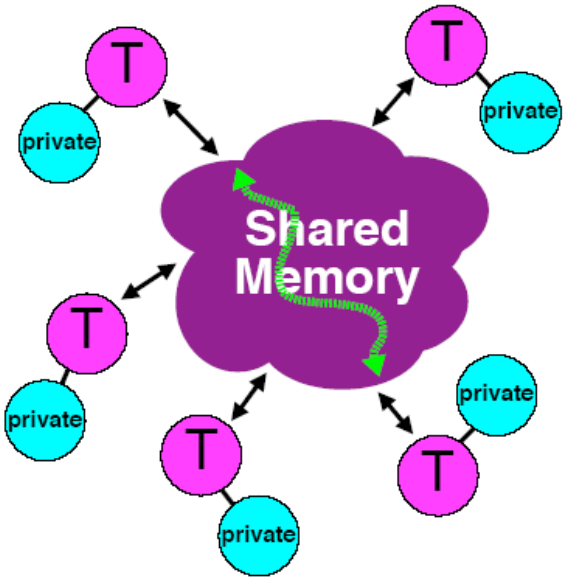


どのようなときに使い分けをするのかを考えてみましょう。

Data scope属性指定

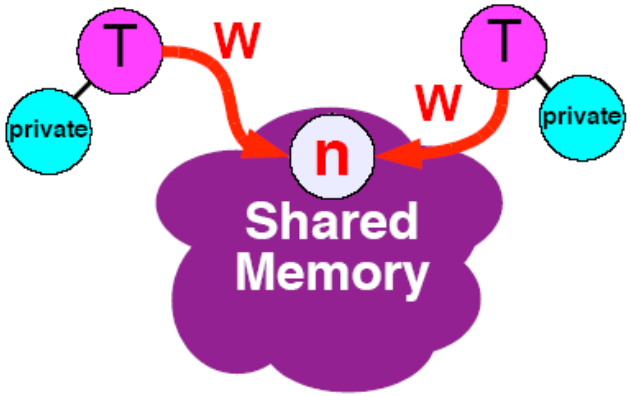
- `parallel`構文、work sharing構文で指示節で指定
- `shared(var_list)`
  - 構文内で指定された変数がスレッド間で共有される
- `private(var_list)`
  - 構文内で指定された変数がprivate
- `firstprivate(var_list)`
  - privateと同様であるが、直前の値で初期化される
- `lastprivate(var_list)`
  - privateと同様であるが、構文が終了時に逐次実行された場合の最後の値を反映する
- `reduction(op:var_list)`
  - reduction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

Data Race

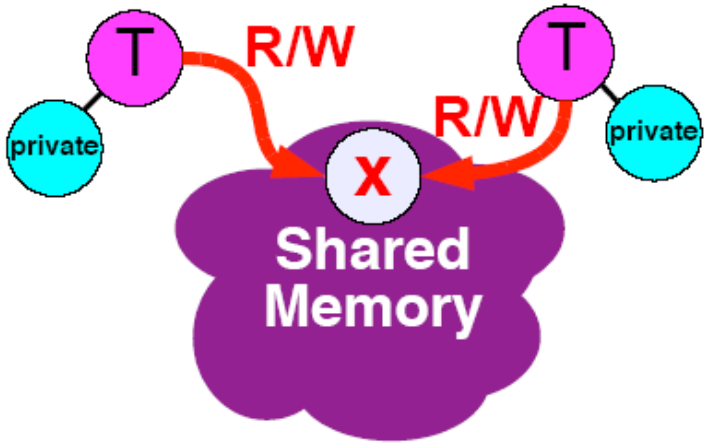


OpenMP
は共有メモリ
モデル

```
#pragma omp parallel shared(n)
{n = omp_get_thread_num();}
```




```
#pragma omp parallel shared(x)
{x = x + 1;}
```



Data Race（データレース）＝
複数のスレッドが同じ共有変
数を同時に書き換える

並列化できないループ

```
for (i=0; i<8; i++)
   a[i] = a[i] + b[i];
```

Every iteration in this loop is independent of the other iterations

Thread 1	Thread 2
a[0]=a[0]+b[0]	a[4]=a[4]+b[4]
a[1]=a[1]+b[1]	a[5]=a[5]+b[5]
a[2]=a[2]+b[2]	a[6]=a[6]+b[6]
a[3]=a[3]+b[3]	a[7]=a[7]+b[7]

Time

```
for (i=0; i<8; i++)
   a[i] = a[i+1] + b[i];
```

The result is not deterministic when run in parallel !

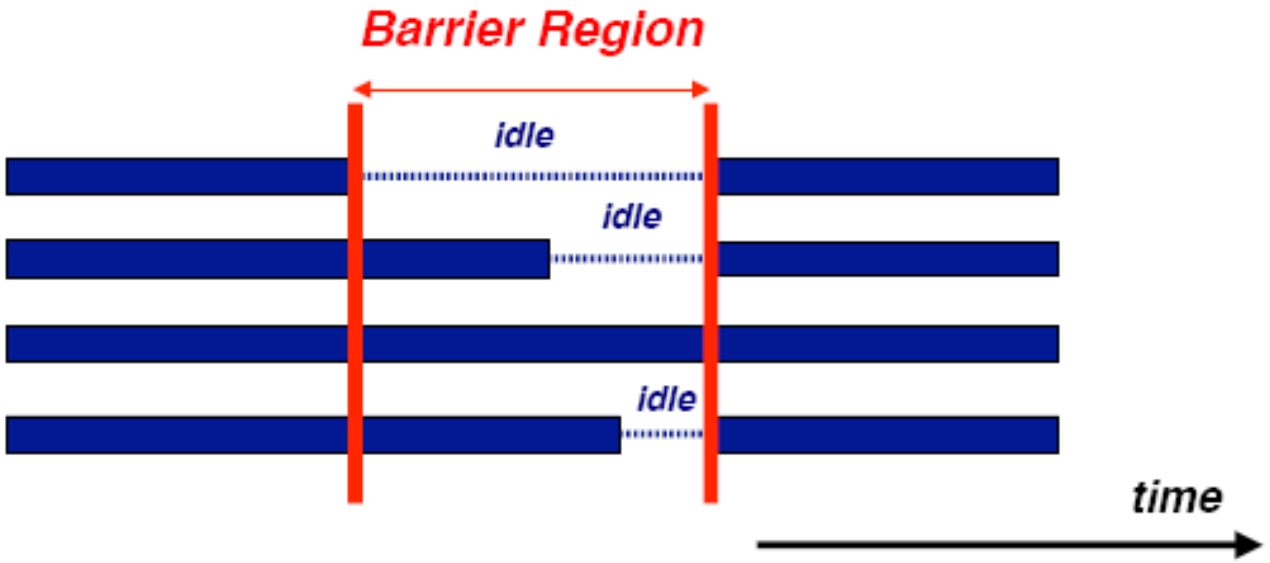
Thread 1	Thread 2
a[0]=a[1]+b[0]	a[4]=a[5]+b[4]
a[1]=a[2]+b[1]	a[5]=a[6]+b[5]
a[2]=a[3]+b[2]	a[6]=a[7]+b[6]
a[3]=a[4]+b[3]	a[7]=a[8]+b[7]

Time

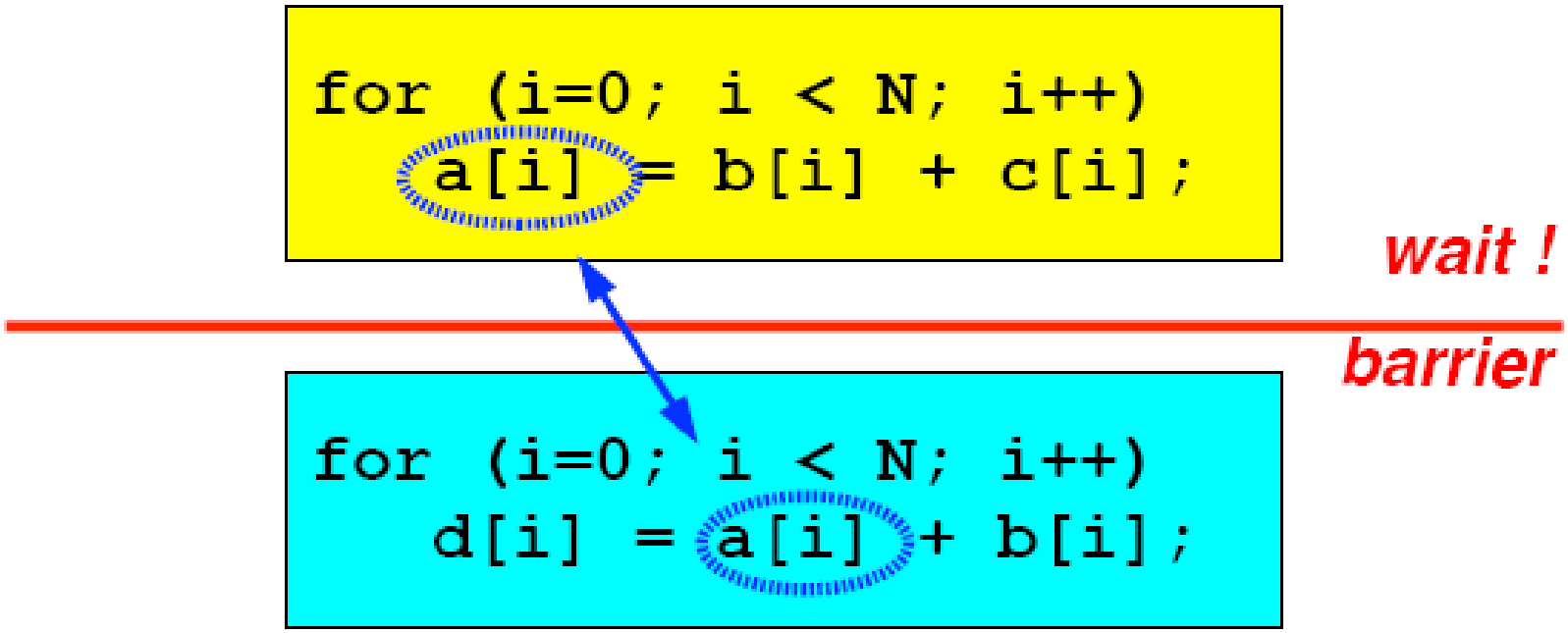
Barrier 指示文

- 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
```



バリアはこういう時大事



通常のfor構文は、implicitにバリアがとられているために、
特別にバリアをいれる必要はない

nowaitの使い方

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)

その他、重要な指示文

- single構文： １つのスレッドだけで実行する部分を指定
- master構文：マスタ・スレッドだけで実行する部分を指定
- section構文：別々のプログラム実行をスレッドをスレッドに割り当てる
- critical構文：排他領域（同時に実行できない部分）を指定
- flush構文
- threadprivate構文

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

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

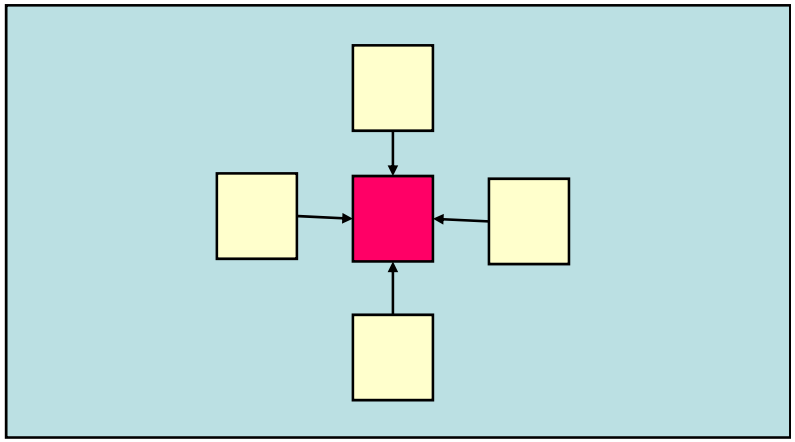
- OpenMP版
  - ループを並列化するだけ, 1行のみ
- MPI版(cpi-mpi.c)
  - 入力された変数nの値をBcast
  - 最後にreduction
  - 計算は、プロセッサごとに飛び飛びにやっている

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;
}
```

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;
    }
}
```

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

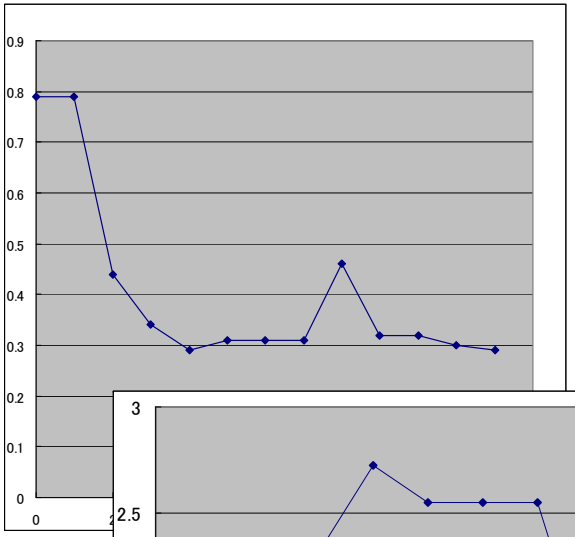
- Webで探してみてください。
- ぜひ、自分でやって、みてください。

Laplaceの性能

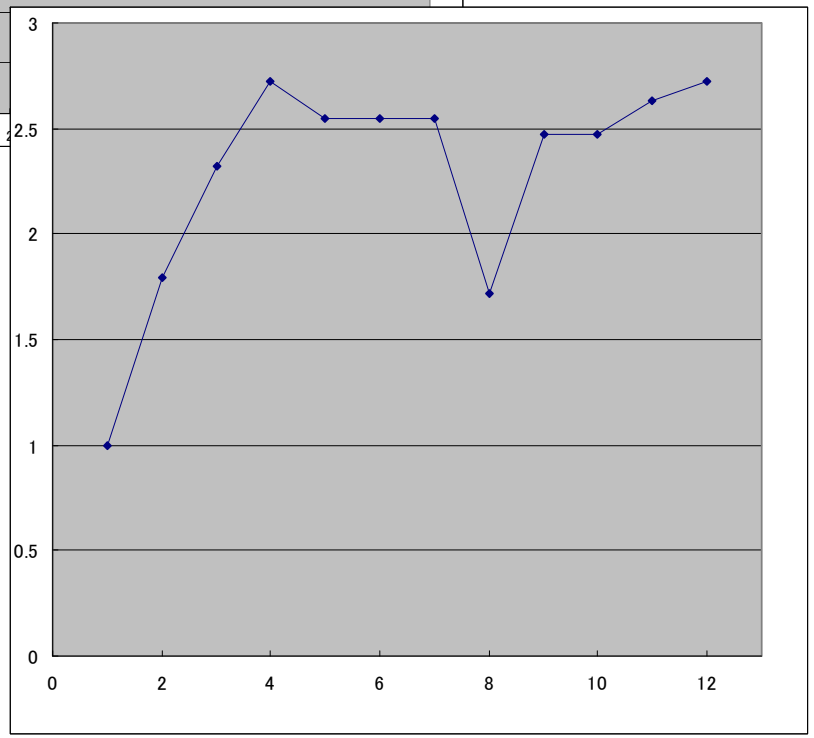
Core i7 920 @ 2.67GHz, 2 socket

XSIZE=YSIZE=1000

実行時間

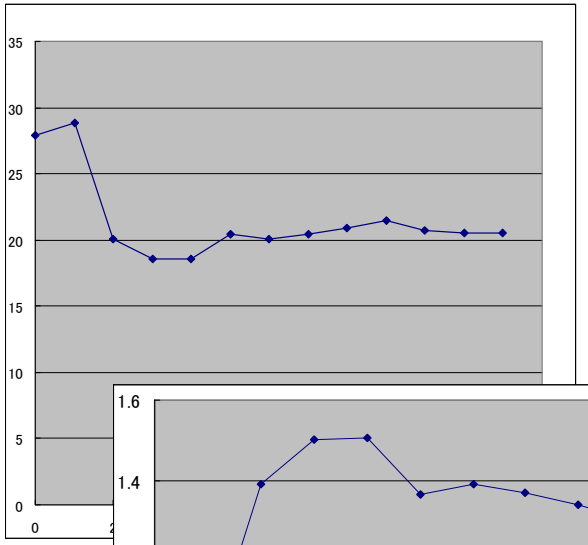


対逐次性能比

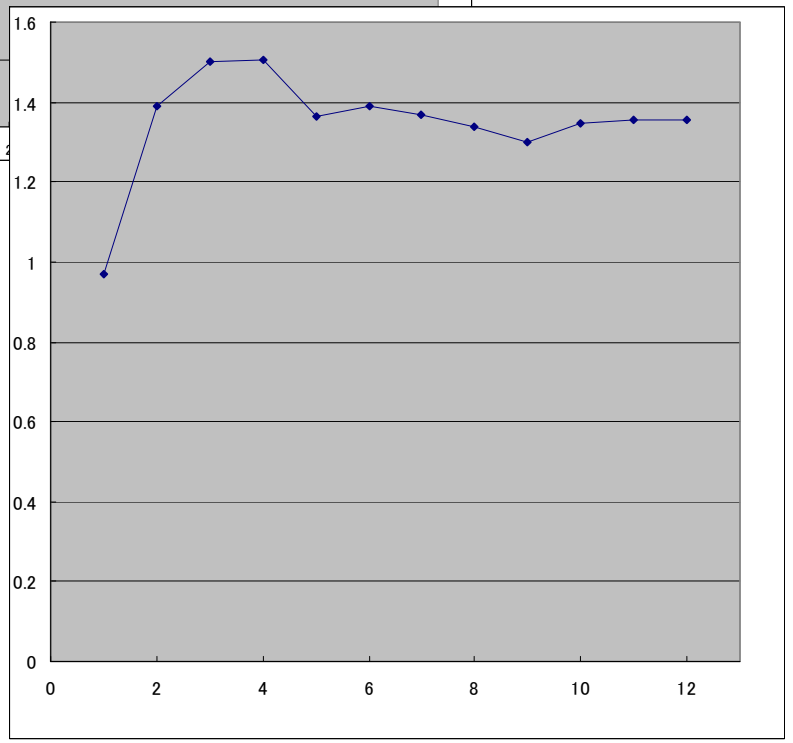


XSIZE=YSIZE=8000

実行時間



対逐次性能比

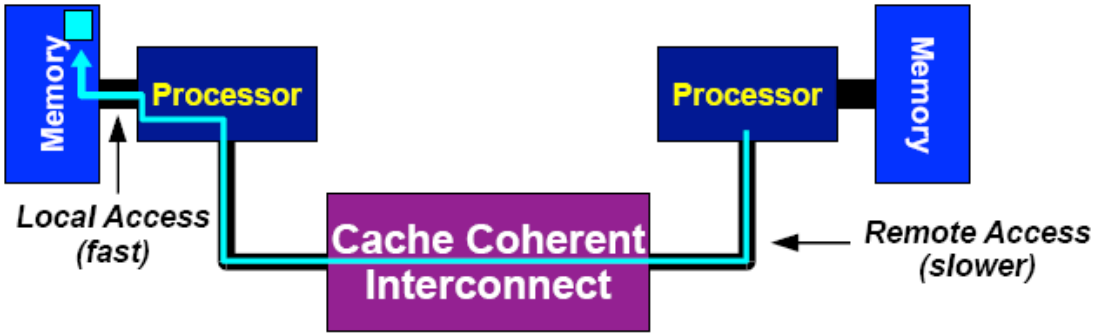


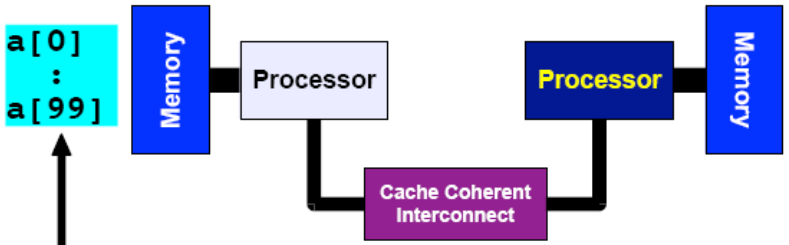
The Myth

"OpenMP Does Not Scale"

- OpenMPはスケールしない？
  - *The transparency of OpenMP is a mixed blessing*
    - *Makes things pretty easy*
    - *May mask performance bottlenecks*
  - *In the ideal world, an OpenMP application just performs well*
  - *Unfortunately, this is not the case*
  - *Two of the more obscure effects that can negatively impact performance are cc-NUMA behavior and False Sharing*
  - *Neither of these are restricted to OpenMP, but they are important enough to cover in some detail here*

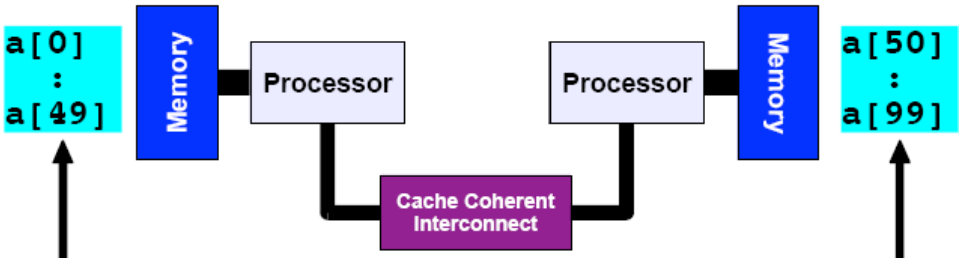
CC-NUMAとfirst touch






```
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
All array elements are in the memory of the processor executing this thread

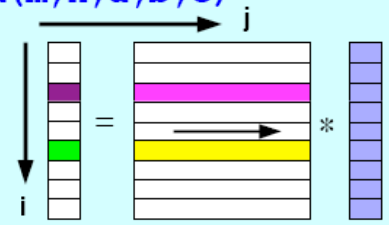


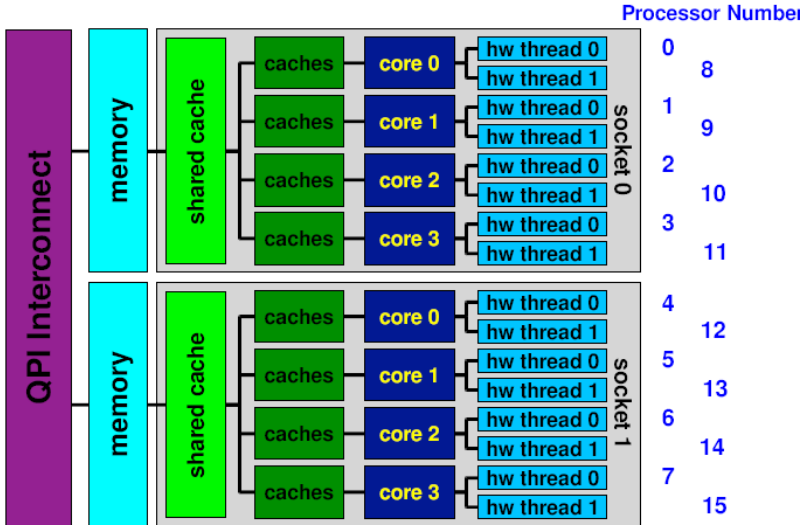

```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
Both memories each have "their half" of the array

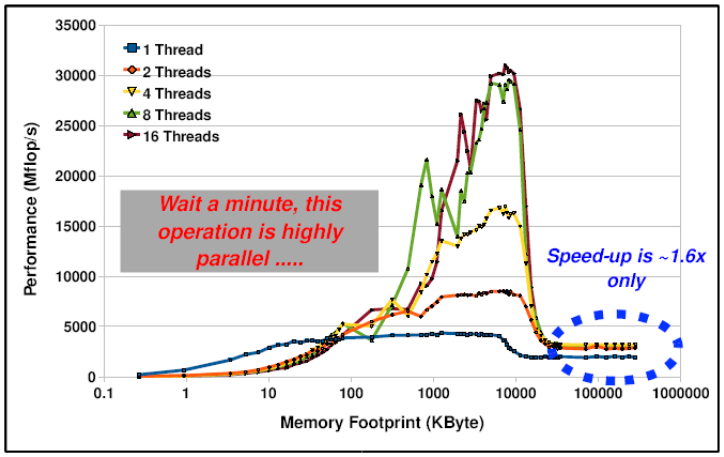
First touchをすると

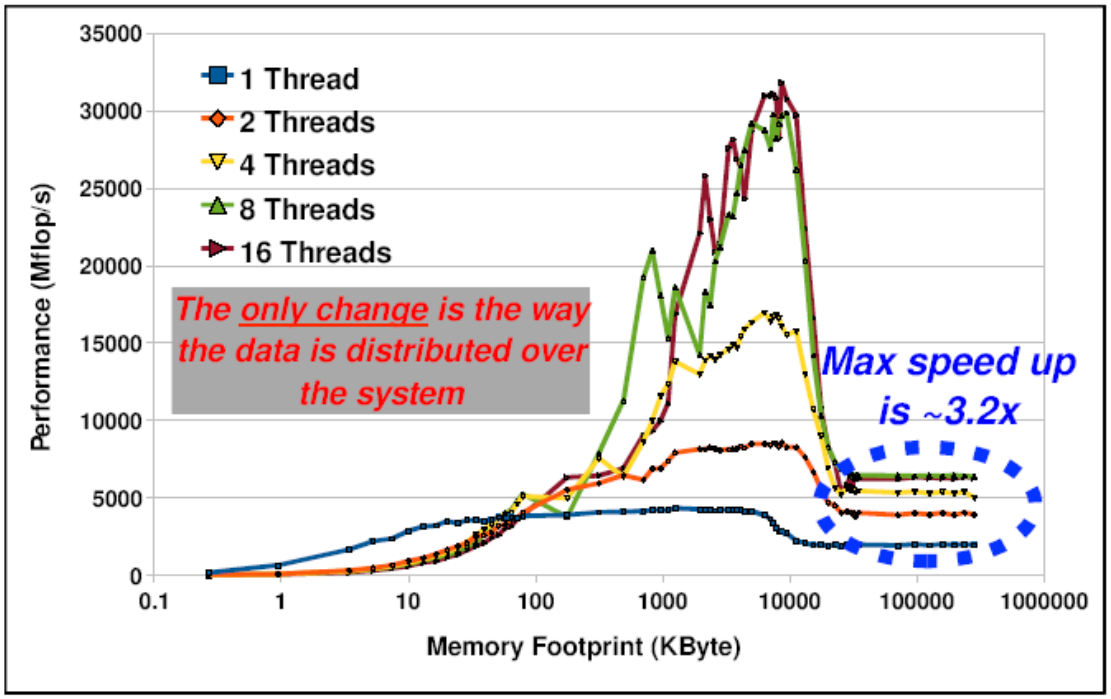
```
#pragma omp parallel for default(none) \
        private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i][j]*c[j];
 }
```





2 socket Nehalem





Advanced topics

- MPI/OpenMP Hybrid Programming
  - SMPクラスタでのプログラミング
- OpenMP 3.0
  - 2007年にapproveされた
  - Task
- OpenMP 4.0
  - 2013年にリリース
  - GPUなどのACCも対応

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

Task構文の例

```
long comp_fib_numbers(int n){

   // Basic algorithm: f(n) = f(n-1) + f(n-2)

   long fnm1, fnm2, fn;

   if ( n == 0 || n == 1 ) return(n);

#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}

#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}

#pragma omp taskwait
   fn   = fnm1 + fnm2;

   return(fn);
}
```

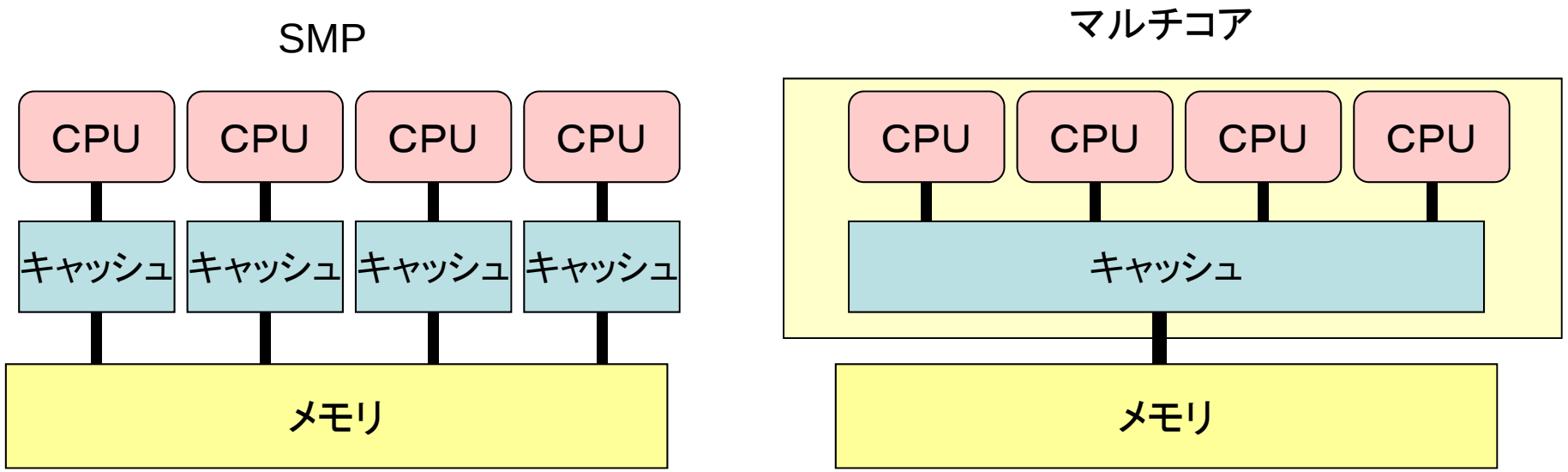
外側にparallel 構文が必要

```
long comp_fib_numbers(int n){

   // Basic algorithm: f(n) = f(n-1) + f(n-2)

   long fnm1, fnm2, fn;

   if ( n == 0 || n == 1 ) return(n);
   if ( n < 20 ) return(comp_fib_numbers(n-1) +
                        comp_fib_numbers(n-2));

#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}

#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}

#pragma omp taskwait
   fn   = fnm1 + fnm2;

   return(fn);
}
```

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）が早い場合がある
  - 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
    - 大規模になった時に、MPIプロセス数が問題になる

- 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
  - MPIプロセス数が少なくなる。

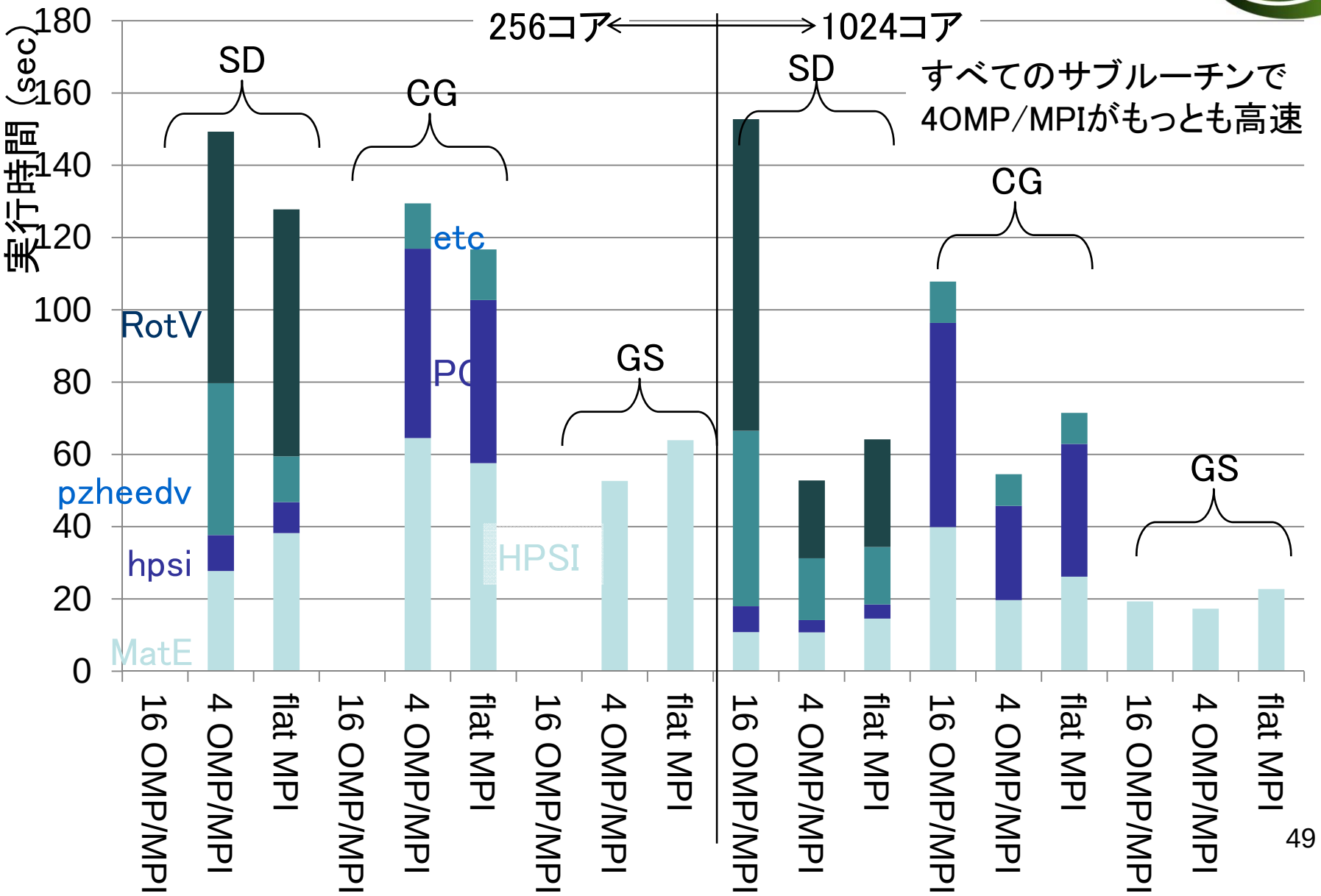


MPIとOpenMPのHybridプログラミング

- 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？）

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.



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

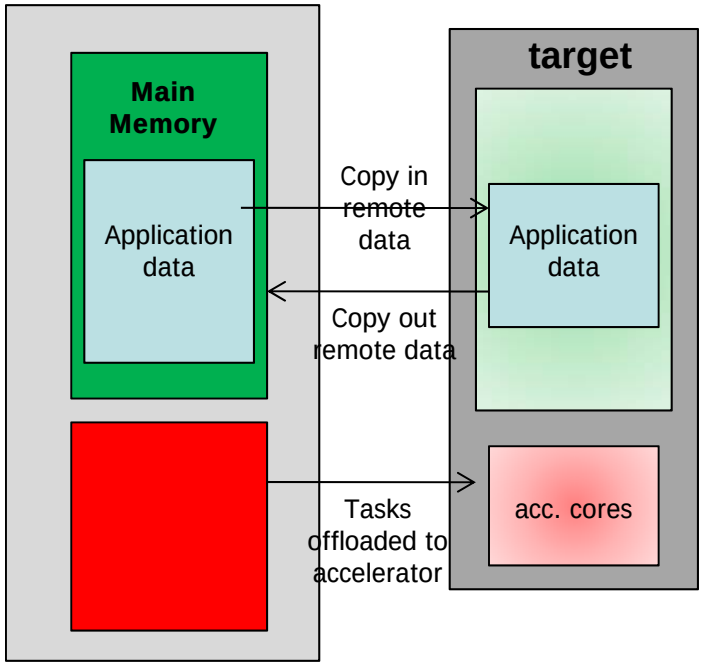
OpenMP 4.0

- Released July 2013
  - http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
  - A document of examples is expected to release soon
- Changes from 3.1 to 4.0 (Appendix E.1):
  - *Accelerator: 2.9*
  - *SIMD extensions: 2.8*
  - *Places and thread affinity: 2.5.2, 4.5*
  - *Taskgroup and dependent tasks: 2.12.5, 2.11*
  - *Error handling: 2.13*
  - *User-defined reductions: 2.15*
  - *Sequentially consistent atomics: 2.12.6*
  - *Fortran 2003 support*

slide by Yonghong@UH

Accelerator (2.9): offloading

- Execution Model: Offload data and code to accelerator
- *target* construct creates tasks to be executed by devices
- Aims to work with wide variety of accs
  - GPGPUs, MIC, DSP, FPGA, etc
  - A target could be even a remote node, intentionally




```
#pragma omp target
{
      /* it is like a new task
        * executed on a remote device */
{
```


slide by Yonghong@UH

Accelerator: explicit data mapping

- Relatively small number of truly shared memory accelerators so far
- Require the user to explicitly *map* data to and from the device memory
- Use array region

```
long a = 0x858;
long b = 0;
int anArray[100]

#pragma omp target data map(to:a) ¥¥
   map(tofrom:b,anArray[0:64])
{
    /* a, b and anArray are mapped
      * to the device */

   /* work here */
}
/* b and anArray are mapped
  * back to the host */
```


slide by Yonghong@UH

Accelerator: hierarchical parallelism

- Organize massive number of threads
  - teams of threads, e.g. map to CUDA grid/block
- Distribute loops over teams

```
#pragma omp target

#pragma omp teams num_teams(2)
            num_threads(8)
{
    //-- creates a “league” of teams
   //-- only local barriers permitted
#pragma omp distribute
for (int i=0; i<N; i++) {

}

}
```

Only target directive makes it as accelerator region

slide by Yonghong@UH

target and map examples

```
void vec_mult(int N)
{
   int i;
   float p[N], v1[N], v2[N];
   init(v1, v2, N);
   #pragma omp target map(to: v1, v2) map(from: p)
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

```
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

slide by Yonghong@UH

target date example

```
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data map(from: p[0:N])
   {
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
      init_again(v1, v2, N);
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = p[i] + (v1[i] * v2[i]);
   }
   output(p, N);
}
```

Note mapping inheritance

slide by Yonghong@UH

teams and distribute loop example

```
float dotprod_teams(float B[], float C[], int N, int num_blocks,
  int block_threads)
{
    float sum = 0;
    int i, i0;
    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(num_blocks) thread_limit(block_threads)
      reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += num_blocks)
       #pragma omp parallel for reduction(+:sum)
       for (i=i0; i< min(i0+num_blocks,N); i++)
           sum += B[i] * C[i];
    return sum;
}
```

Double-nested loops are mapped to the two levels of thread hierarchy (league and team)

slide by Yonghong@UH

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
- OpenMP 4.0 でaccも対応
- 16プロセッサ以上になれば、MPIが必須
  - ただし、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);
}
```