マルチスレッドプログラミング 入門

OpenMP、Cluster OpenMPによる並列プログラミング

スケーラブルシステムズ株式会社

内容

- **はじめに**
  なぜ、マルチスレッドプログラミング？
- **並列処理について**
- **マルチスレッドプログラミングの概要**
- **並列処理での留意点**
- **OpenMPによるマルチスレッドプログラミングのご紹介**
- **まとめとして**
  参考資料のご紹介

なぜ、マルチスレッドプログラミング？

HWの進化

- マイクロプロセッサのマルチコア化が進み、モバイル、デスクトップ、サーバの全ての分野で複数のプロセッサコアが利用出来ます。

並列処理

- 計算処理に際してのユーザの要求に対応し、そのようなマルチコアの利点を活用するために、複数のスレッドによる並列処理が求められています。

マルチスレッド

- 複数スレッドの並列処理のためのプログラミングがマルチスレッドプログラミングです。マルチスレッドプログラミングによって、アプリケーションの性能向上や機能強化を図ることが可能となります。

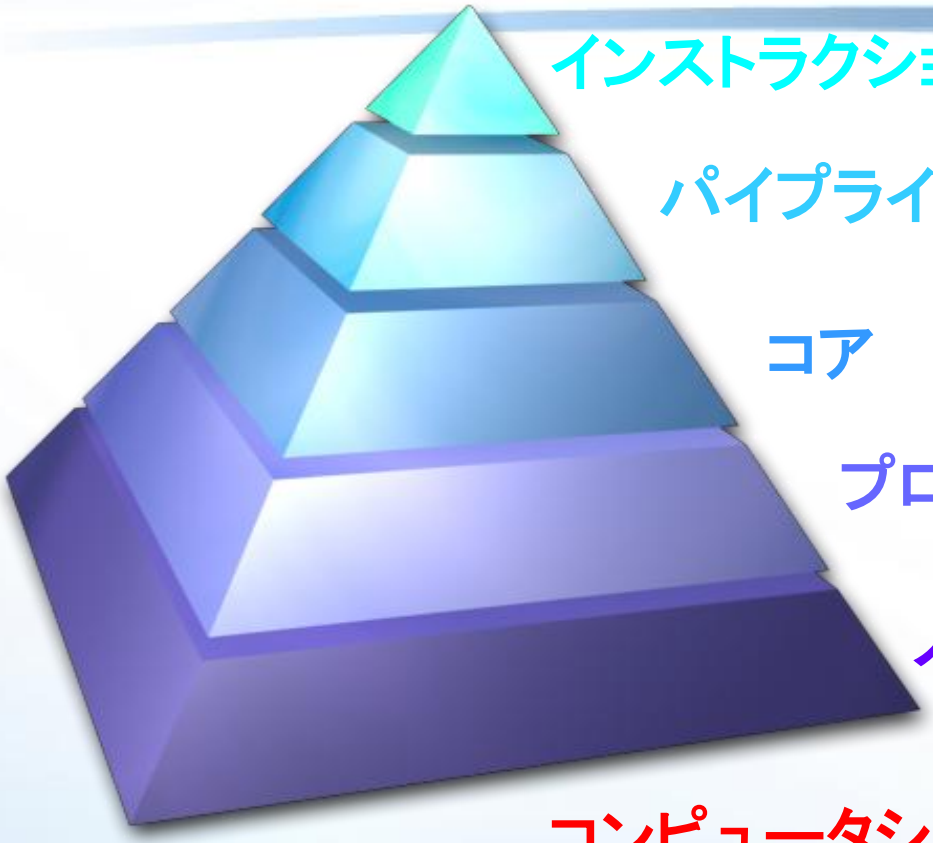
アプリケーションのマルチスレッド化の利点

- **レスポンスの改善と生産性の向上**
  - アプリケーション利用時のレスポンスの向上を、複数のタスクを並列に実行し、処理を行うことで実現可能
- **アプリケーションの実行性能の向上**
  - 多くの計算シュミレーションやWEBサービスなどは、‘並列性’を持つ
  - 計算処理を複数のプロセッサ、プロセッサコアに分散し、処理することでより短い時間で処理を終了させることが可能となる
- **コンピュータリソースの節約と有効利用**
  - より多くのコンピュータリソースをコンパクトに実装可能となり、設置面積と消費電力の効率化が可能

ハードウエアとソフトウエア

- **Hyper-Threading (HT)　テクノロジー**
  - CPUリソースの有効活用とプロセッサの性能向上のためのハードウエア技術
- **Daul-Core、Multi-Core**
  - 複数のプロセッサコアを一つのプロセッサ上に実装することで、プロセッサの性能向上を図るハードウエア技術
- **Multi-threading (マルチスレッド化）**
  - 複数のプロセッサ（コア）を同時に利用することで、処理性能の向上を図るソフトウエア技術
  - オペレーティングシステムが行っている複数のタスク、プロセスのマルチプロセッサでの多重並列処理をアプリケーションレベルで実現する技術

コンピュータでの並列処理階層



インストラクション

パイプライン

コンパイラによる命列実行レベルでの並列処理（ILP）クロックあたり複数の命令実行

コア

プロセッサ

ユーザ及びコンパイラによるスレッドレベルでの並列処理（TLP）

ノード

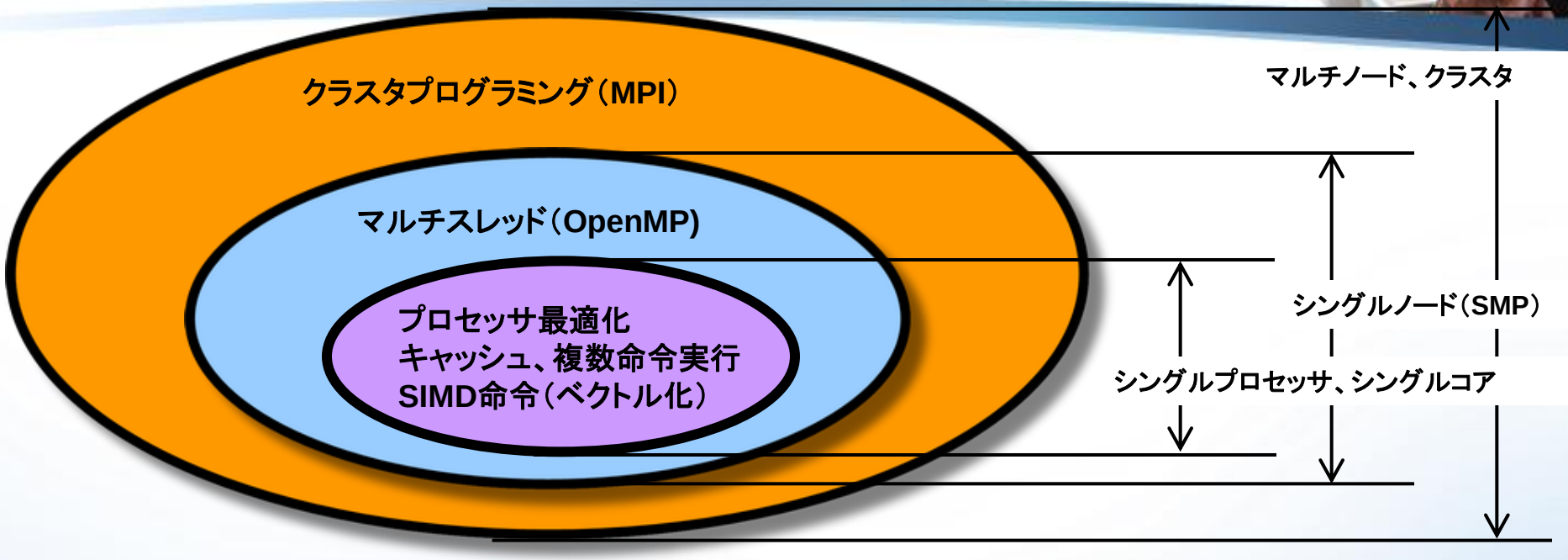
MPIなどによる複数のノード間での並列処理

コンピュータシステム

マルチタスク処理(OSの並列処理)複数のプロセス、スレッド、ジョブの同時実行

これらの全ての並列処理を効率よくスケジューリングすることで、高い性能を実現することが可能

プログラミング階層



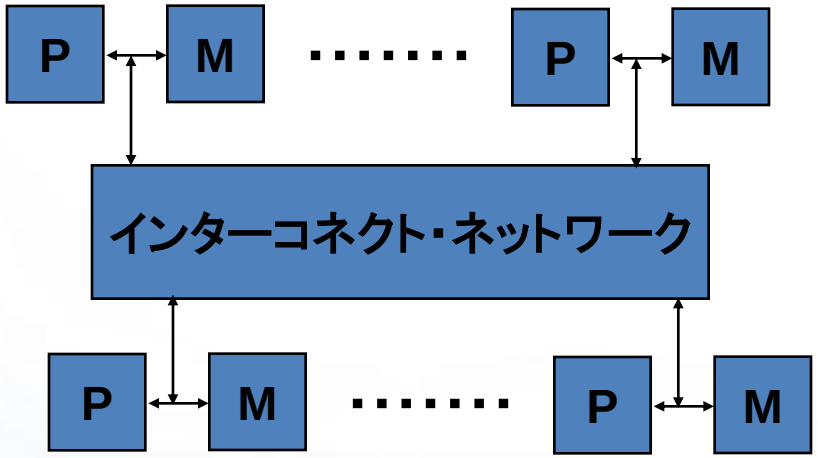

```
do izone = 1, nzone
    ..........
    do j = 1 , jmax
        ..........
        do i = 1, imax
          ..........
```

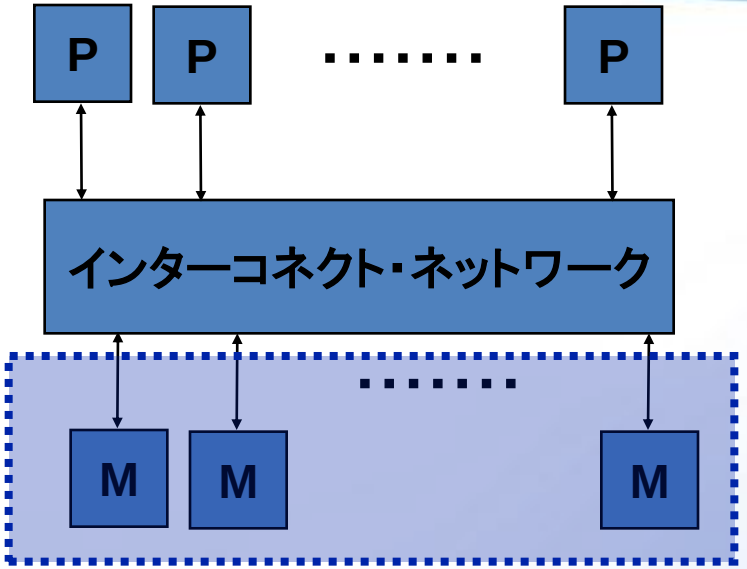
ノード内、ノード間並列化

ノード内でのマルチスレッド並列化

プロセッサリソースの並列利用

並列コンピュータシステム





分散メモリシステム

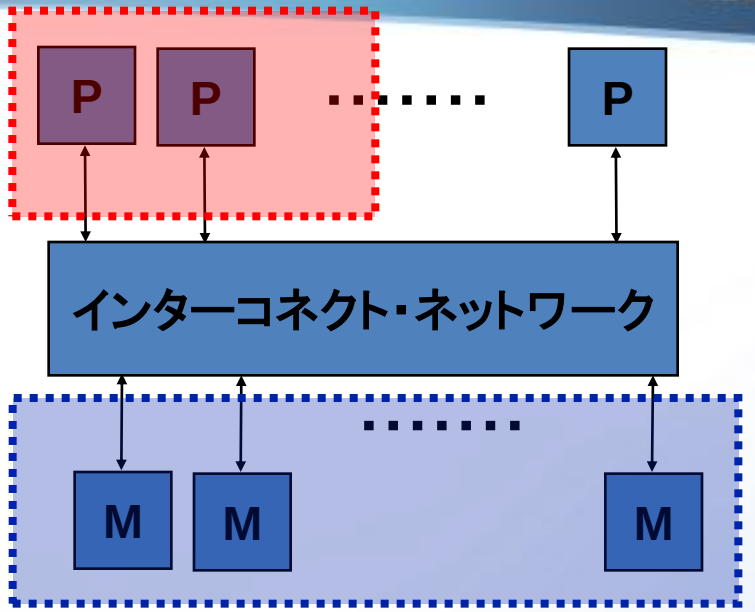
- マルチプロセス
- ローカルメモリ
- メッセージ通信（メッセージ・パッシング）によるデータ共有

共有メモリシステム

- シングルプロセスでのマルチスレッド処理
- 共有メモリとリソース
- 明示的なスレッド、OpenMP

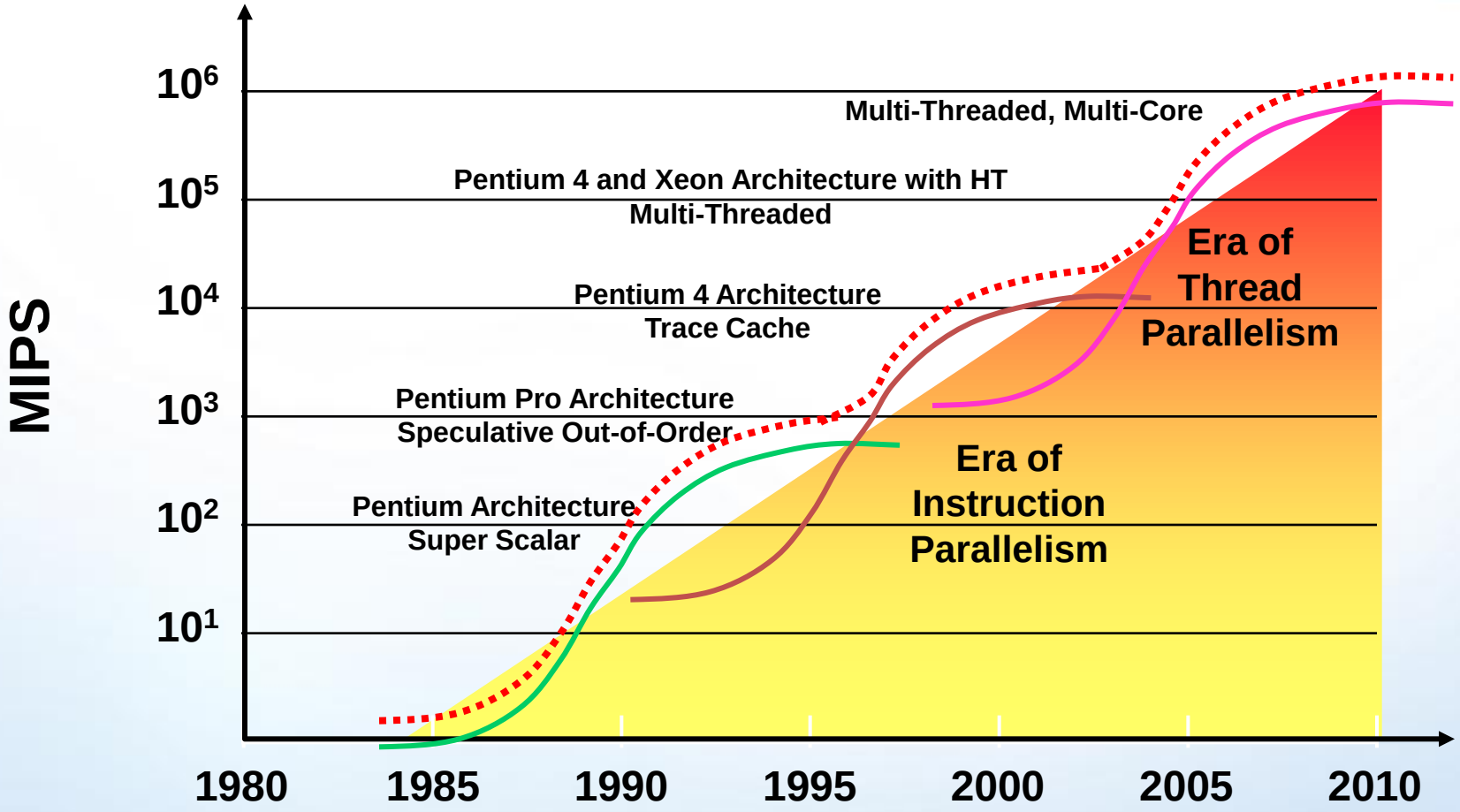
並列コンピュータシステム

Daul-Core、Multi-Core



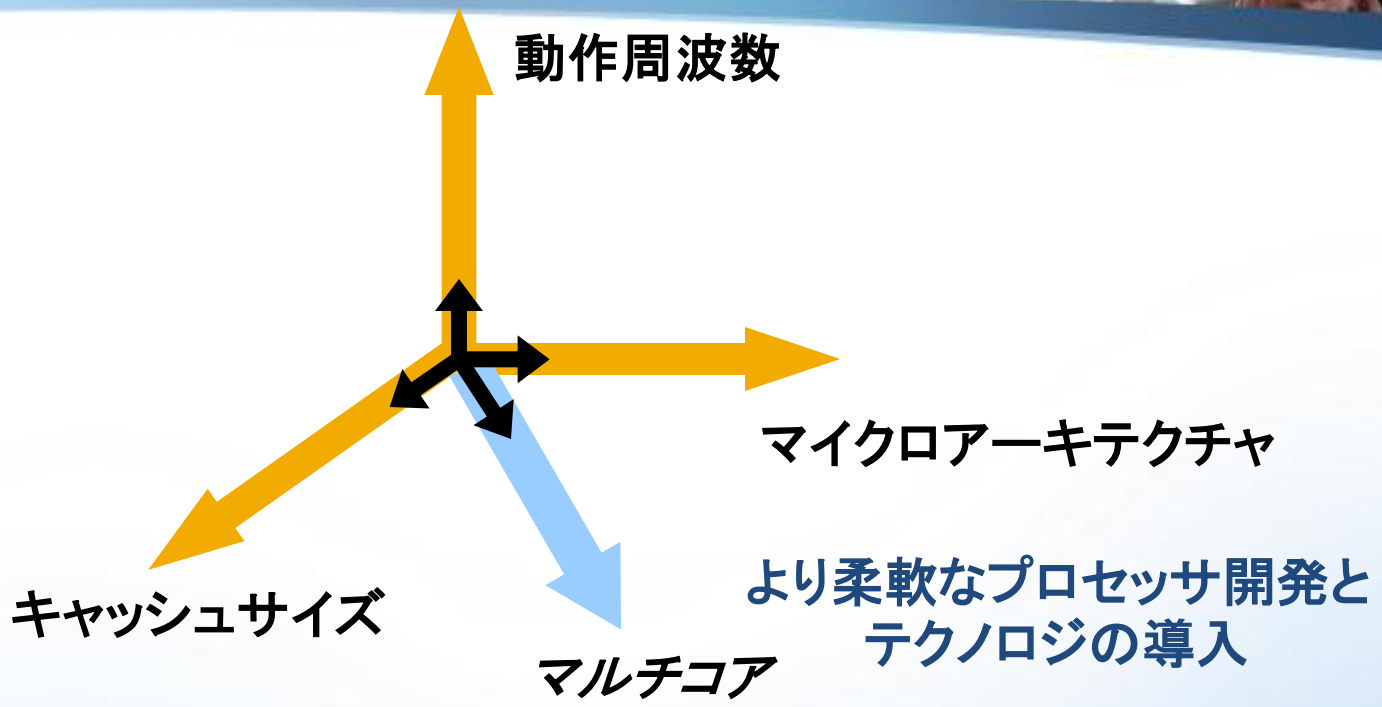
- **複数スレッドのコントロール**
  - 一つ以上のスレッドを並列に実行
  - 各タスクの分割と各部分のスレッドでの実行
  - スレッドの同期制御や共有リソースへのアクセス制御
- **共有メモリシステム**
  - シングルプロセスでのマルチスレッド処理
  - 共有メモリとリソース
  - 明示的なスレッド、OpenMP

マイクロアーキテクチャのSカーブ



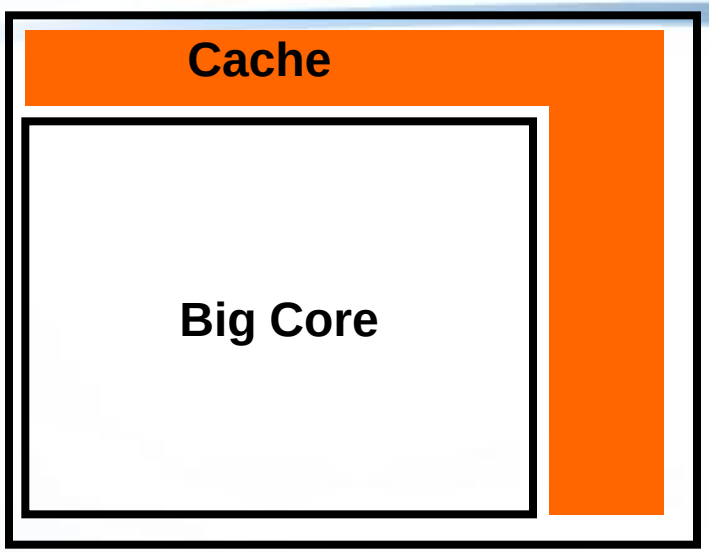
Johan De Gelas, Quest for More Processing Power, AnandTech, Feb. 8, 2005.
http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2343

新たな次元でのプロセッサ開発

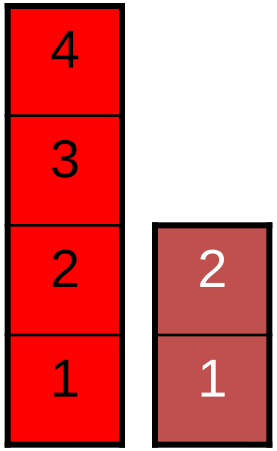


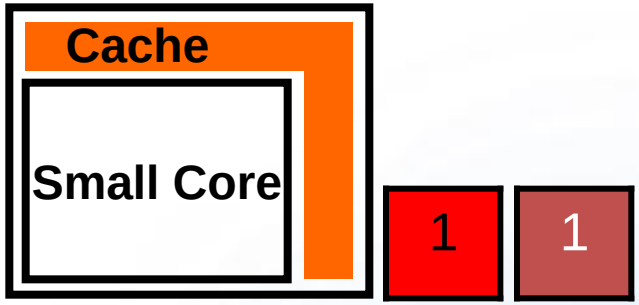
- プロセッサの性能向上のための選択肢が広がる
- 価格性能比の向上を違った次元で提供可能
- 技術的な利点と‘マーケティング’の要求

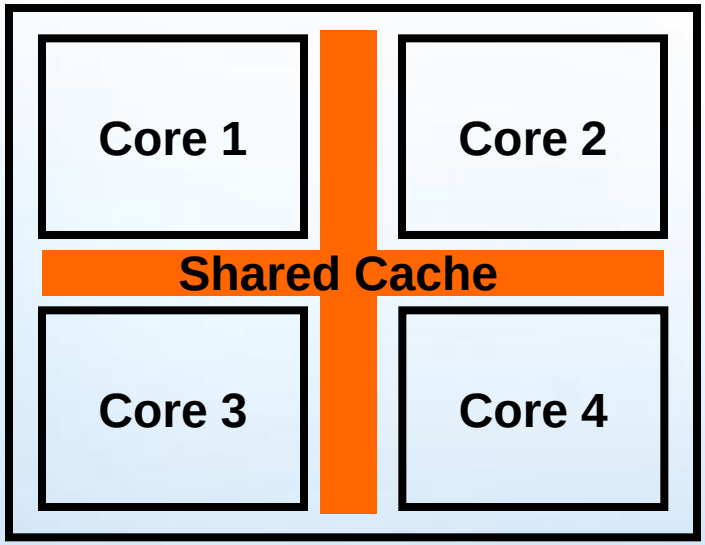
マルチコア：‘性能/消費電力’を改善

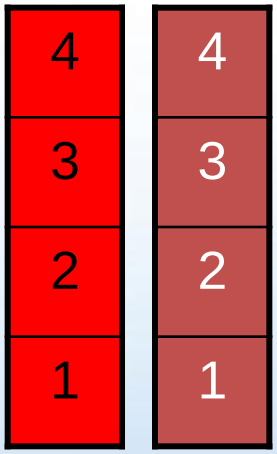


消費電力 / 性能



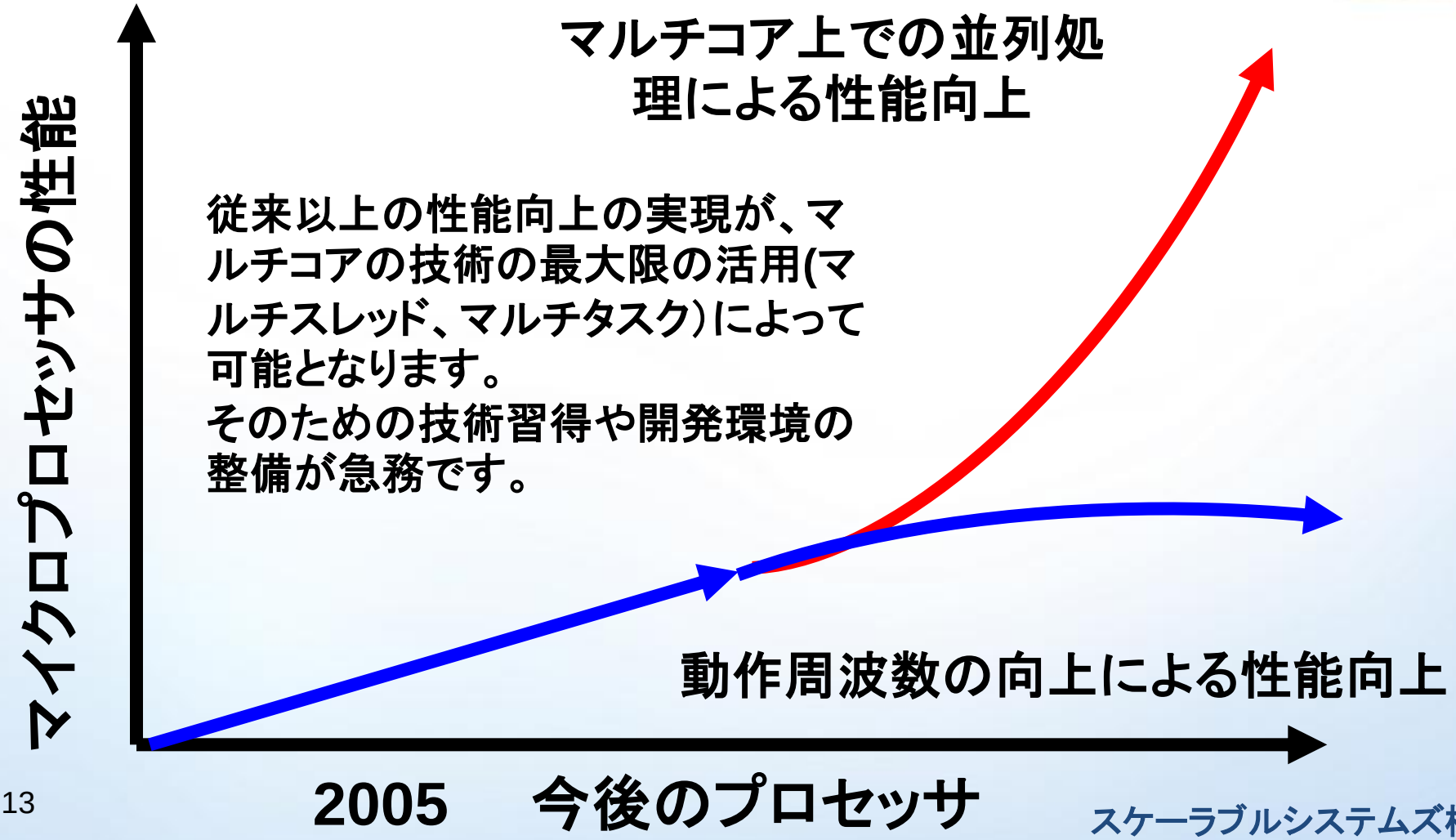






Power ~ コアサイズ
PERFORMANCE ~ √コアサイズ

マイクロプロセッサの性能向上
動作周波数からマルチコアへ



マルチスレッドプログラミングに際して

予習

- スレッドコンセプトの理解
- 並列処理のためのソフトウエア製品の理解
- マルチスレッドプログラミングのAPIの学習

実践

- プログラミング構造の理解
- プログラム実行時のプロファイルの把握（ホットスポット）
- プログラム内の並列性の検討

マルチタスクと並列計算

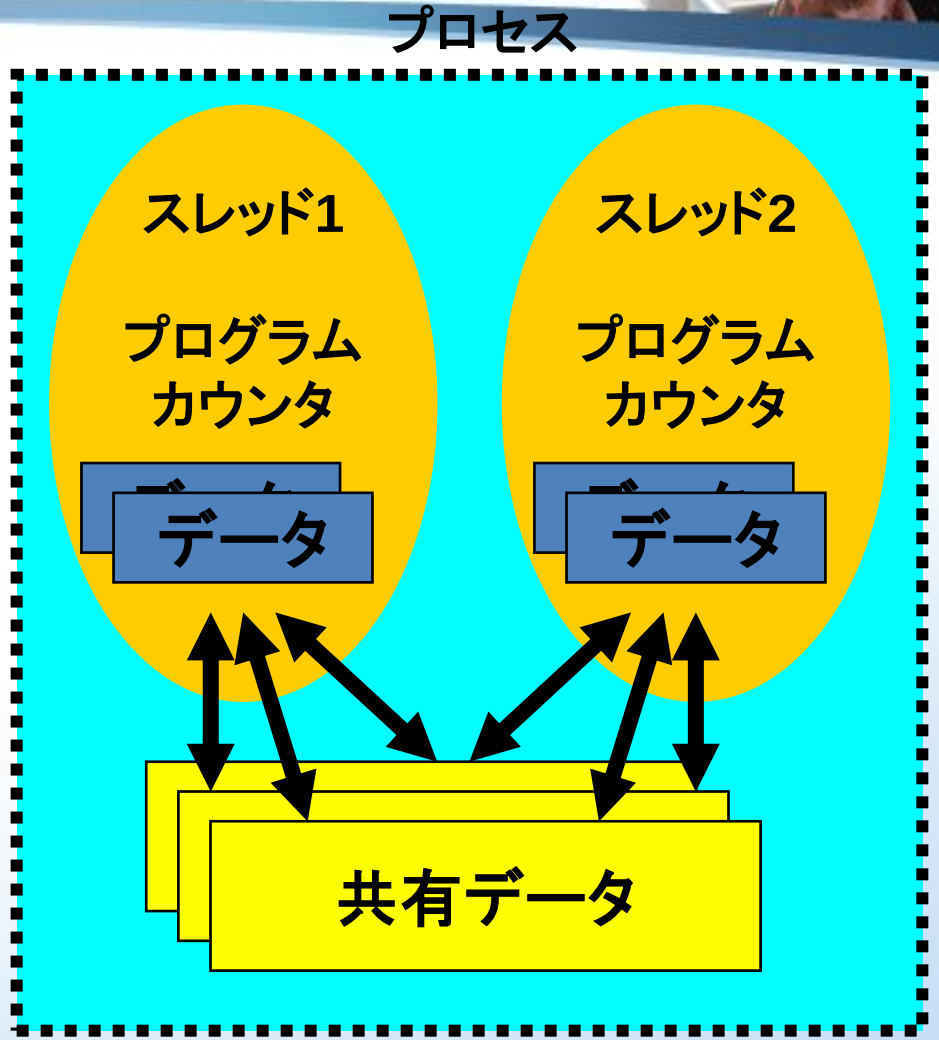
- **マルチタスク**
  - 複数のタスクを同時に処理する
  - データベースやWEBなどのシステムなどでの並列処理
    - 一度に複数のユーザからの大量のデータ処理の要求
  - プロセス単位でのOSによる並列処理
    - 各タスクは複数のプロセスやスレッドを利用して処理を行う
    - プログラム自身の並列化は必ずしも必要ない
- 並列計算
  - **特定の問題に対して、その計算処理を複数のプロセッサを利用して高速に処理する並列処理**
  - **対象となる問題を複数のコア、プロセッサを同時に利用して短時間で解く**
  - **並列プログラミングAPIを利用して、複数のプロセスとスレッドを利用するアプリケーションの開発が必要**

プロセスとスレッド

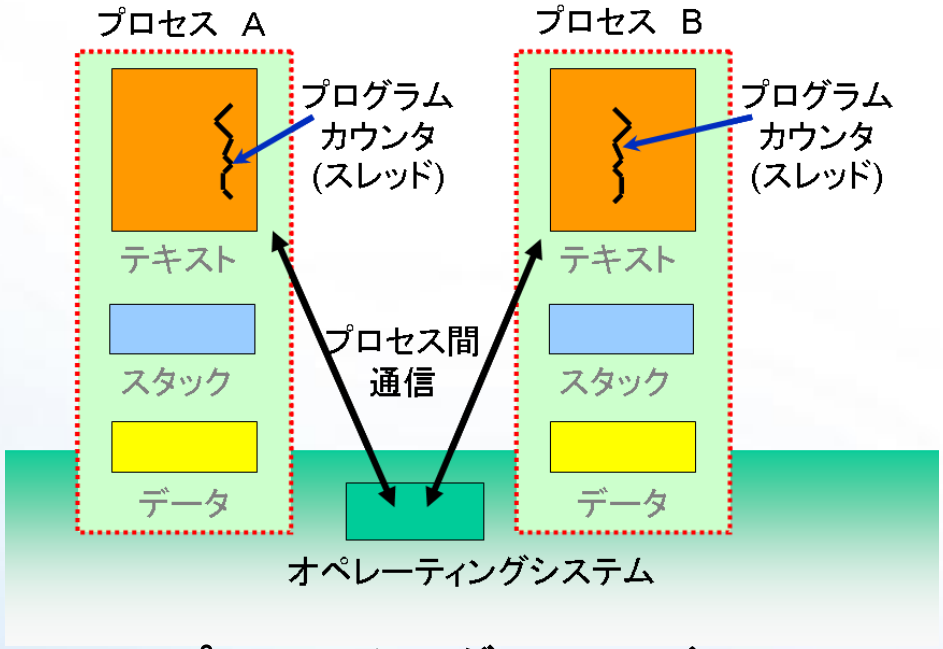
- **並列処理**
  - 「同時に複数の処理（タスク）をOSが処理すること」
  - OSによる処理単位がプロセスとスレッドということになります。
- **プロセス**
  - OSは、要求されたタスクに対して複数のプロセスを起動してその処理を行います。
  - 複数のプロセスを利用して行う並列処理がマルチプロセスとなります。
- **スレッド**
  - これらのプロセス内で生成されて実際の処理を行うのがスレッドとなります。
  - プロセス内で複数のスレッドを生成して、並列処理を行うことをマルチスレッドと呼びます

プロセスとスレッドについて

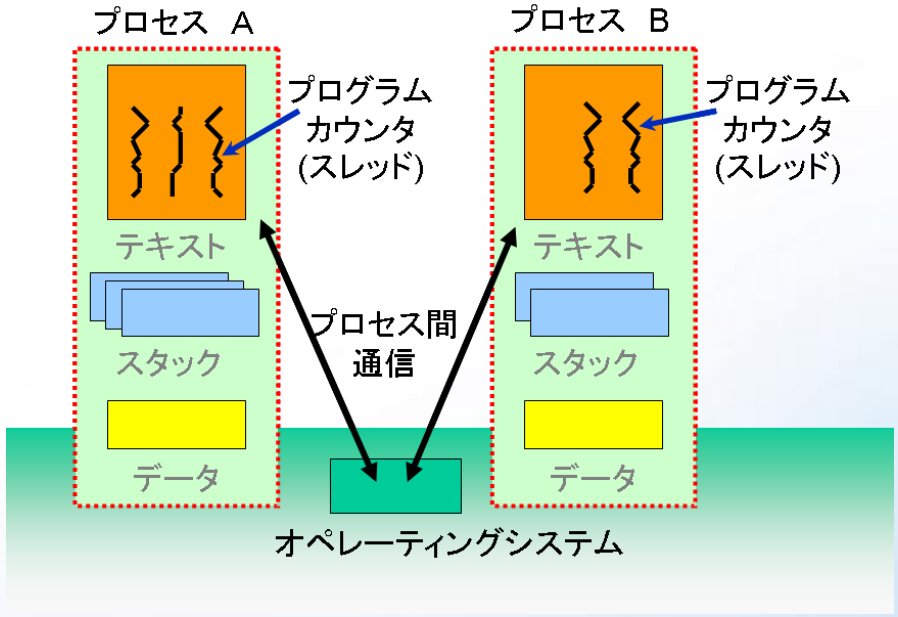
- **プロセスは、独自のリソースを持って個々に実行**
  - 個々に独立したアドレス空間
  - 実行命令、データ、ローカル変数スタック、ステート情報など
  - 個々のレジスタセット
- **スレッドは、一つのプロセス内で実行**
  - アドレス空間を共有
  - レジスタセットも共有
  - 個々のスレッドの独自データ用のスタック領域を持つ



マルチプロセスとマルチスレッド

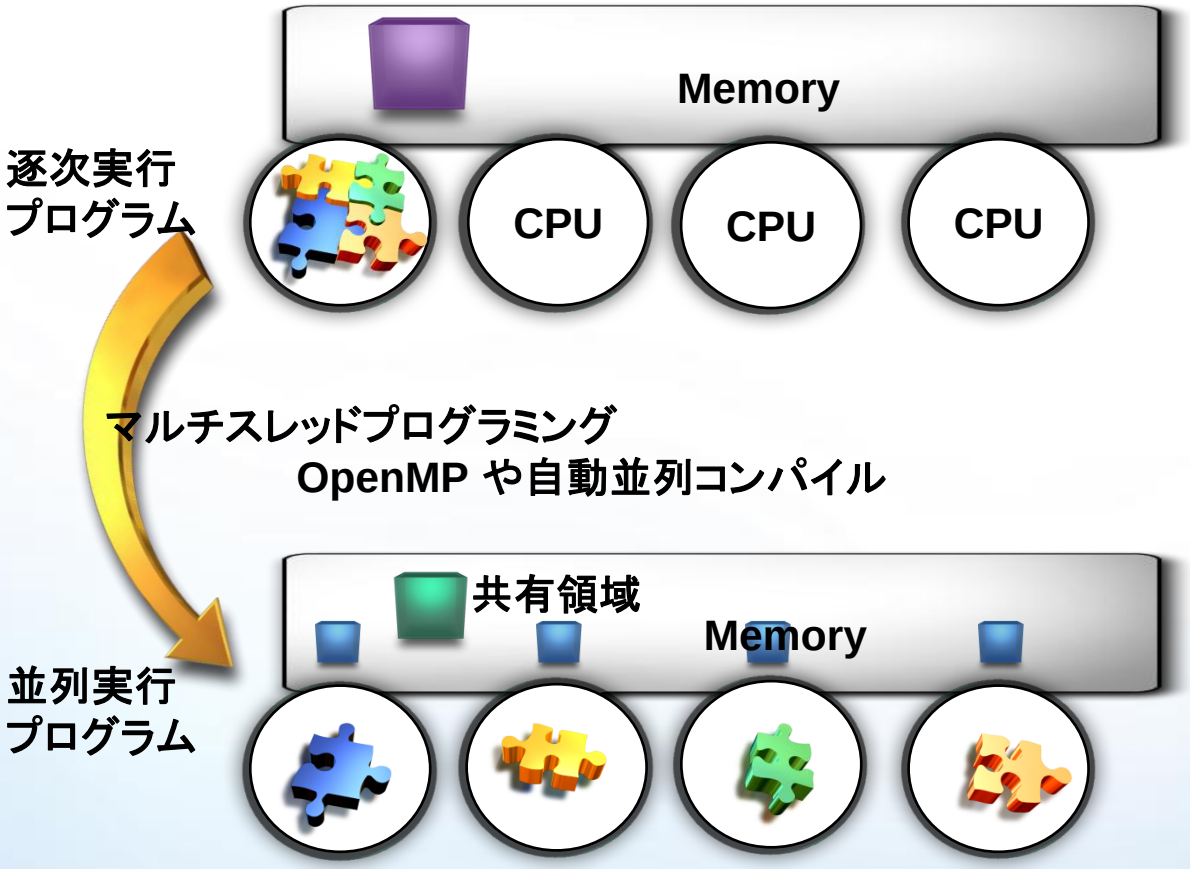


マルチプロセス/シングルスレッド



マルチプロセス/マルチスレッド

マルチスレッドプログラミング



並列計算

- プログラム中には、多くの並列処理可能な処理が存在しているが、通常はそれらの処理を逐次的に処理している
- これらの並列処理可能なコードセグメントに対して、複数のプロセッサ(コア)による同時・並列処理を行う

タスク並列処理:
独立したサブプログラムの並列に呼び出す

```
call fluxx(fv,fx)
call fluxy(fv,fy)
call fluxz(fv,fz)
```

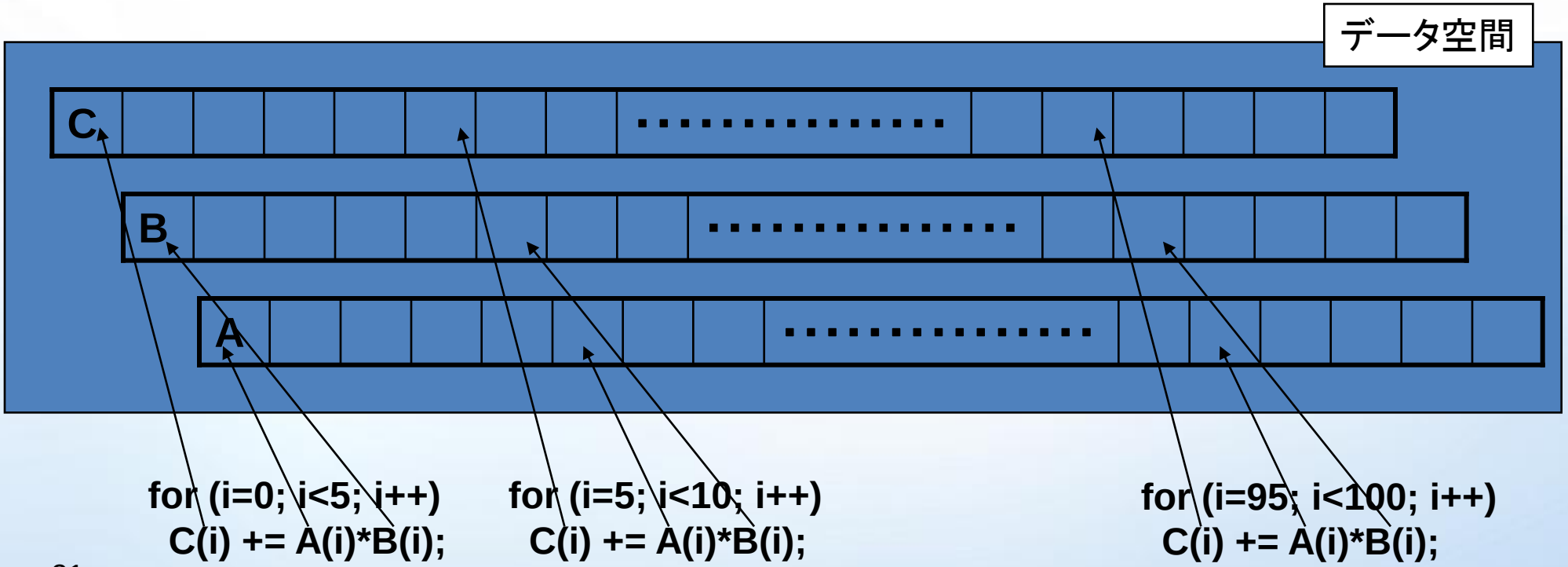
データ並列処理:
独立したループ反復を分割し、並列に実行する

```
for (y=0; y<nLines; y++)
   genLine(model,im[y]);
```

共有メモリデータ並列処理

- 並列処理の一つの方式
- データ空間を共有して並列化を行う

```
for (i=0; i<100; i++)
  C(i) += A(i)*B(i);
```



マルチスレッドプログラミングの基本

OpenMPでのマルチスレッドプログラミング例

- 計算負荷の大きなループやプログラムのセクションを複数のスレッドで同時に処理
- 複数のスレッドを複数のプロセッサコア上で、効率良く処理する

```
void main()
{
    double Res[1000];
// 計算負荷の大きな計算ループに対して、
// マルチスレッドでの並列処理を適用します
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

```
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

逐次処理 .vs. マルチスレッド並列処理

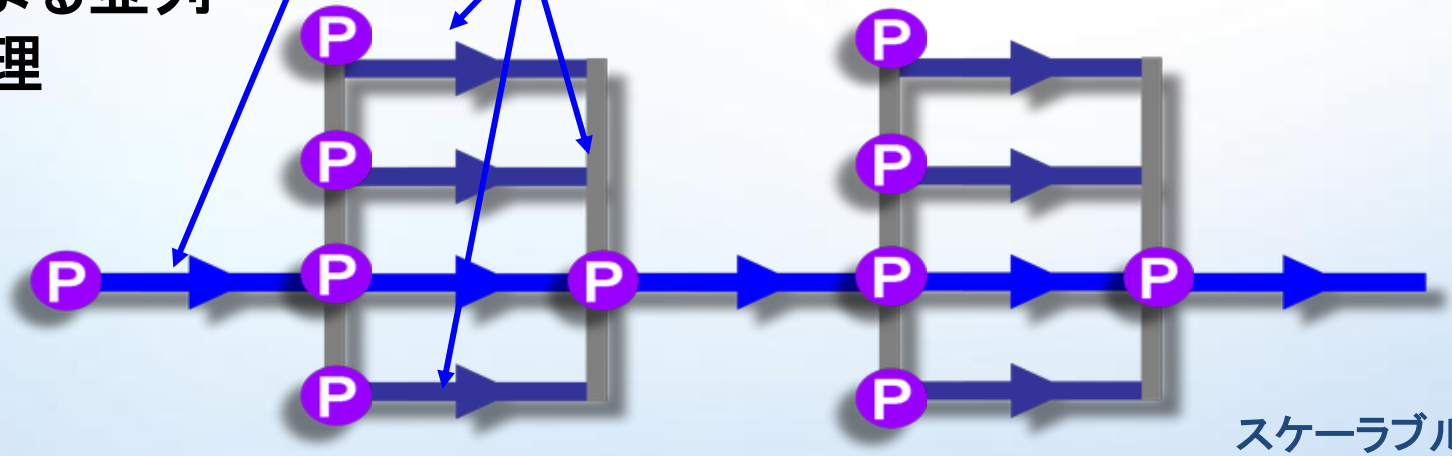
逐次処理

プログラムのループなどの反復計算を複数のスレッドに分割し、並列処理を行う

“マスタースレッド”

“ワーカースレッド”

マルチスレッドによる並列処理



並列化プログラミングAPIの比較

	MPI	スレッド	*OpenMP*
可搬性	✓		✓
スケーラブル	✓	✓	✓
パフォーマンス指向	✓		✓
並列データのサポート	✓	✓	✓
インクリメンタル並列処理			✓
高レベル			✓
直列コードの保持			✓
正当性の確認			✓
分散メモリ	✓		*ClusterOpenMP*

Win32 APIによるπの計算

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
  int i, start;
  double x, sum = 0.0;
  start = *(int *) arg;
  step = 1.0/(double) num_steps;
  for (i=start;i<= num_steps; i=i+NUM_THREADS){
       x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
          thread_handles[i] = CreateThread(0, 0,

           (LPTHREAD_START_ROUTINE) Pi,
                    &threadArg[i], 0, &threadID);
}
  WaitForMultipleObjects(NUM_THREADS,
           thread_handles,TRUE,INFINITE);

  pi = global_sum * step;
  printf(" pi is %f ¥n",pi);
}
```

MPIによるπの計算

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD) ;
}
```

OpenMPによるπの計算

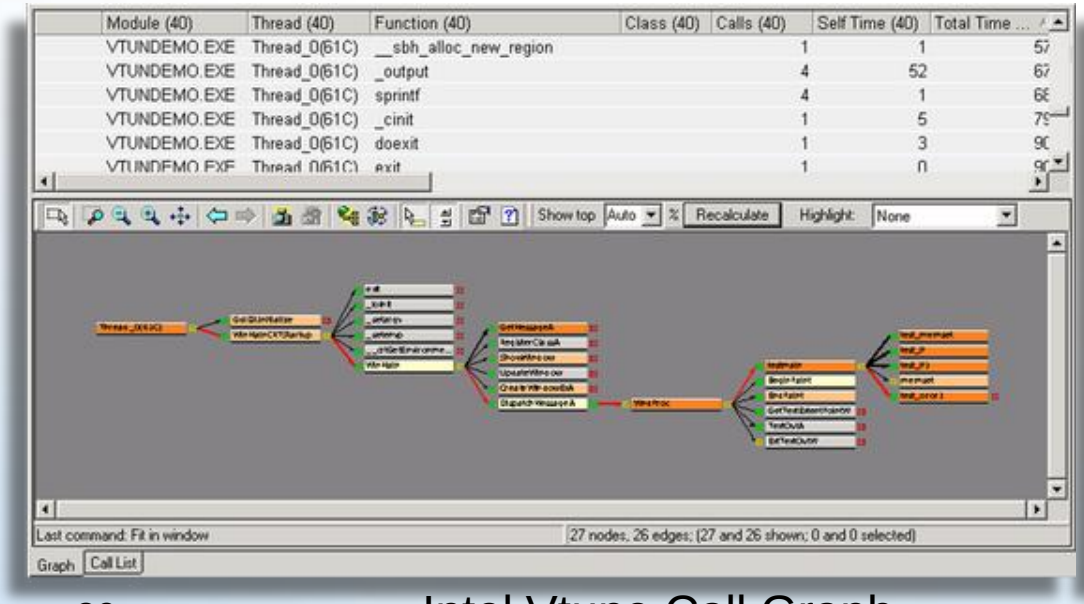
```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
      int i; double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
      for (i=0;i<= num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
      }
     pi = step * sum;
}
```

Cluster OpenMPによるπの計算

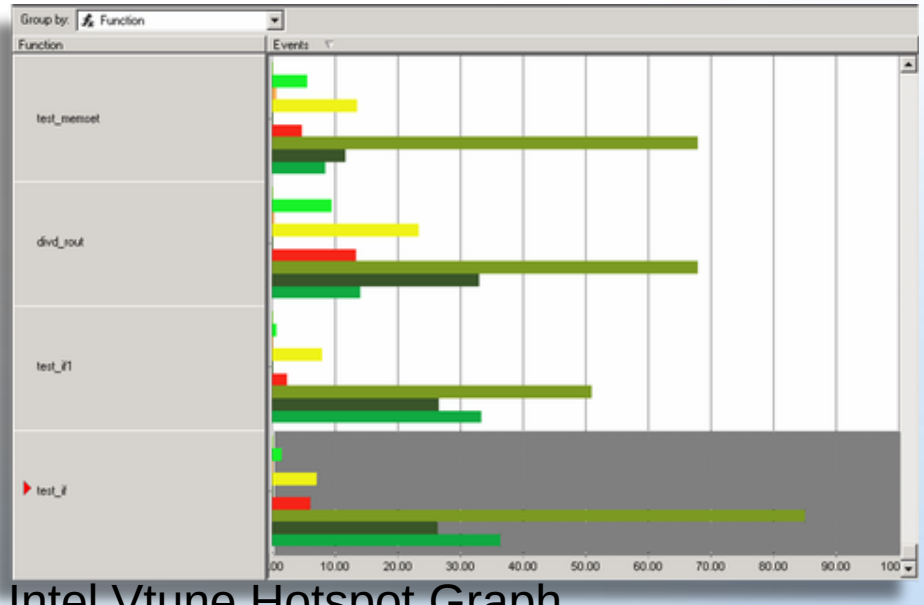
```
#include <omp.h>
static long num_steps = 1000000; double step;
static double sum = 0.0;
#pragma intel omp sharable(sum)
#pragma intel omp sharable(num_steps)
#pragma intel omp sharable(step)
#define NUM_THREADS 4
void main ()
{
        int i; double x, pi;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=0;i<= num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

マルチスレッドの適用候補?

- **ホットスポットでの反復ループ**
  【適用条件】各ループの反復はお互いに独立して計算可能であること
- **ホットスポットでの実行処理タスク**
  【適用条件】タスクが相互に依存することなく実行可能であること



Intel Vtune Call Graph
& Critical Path



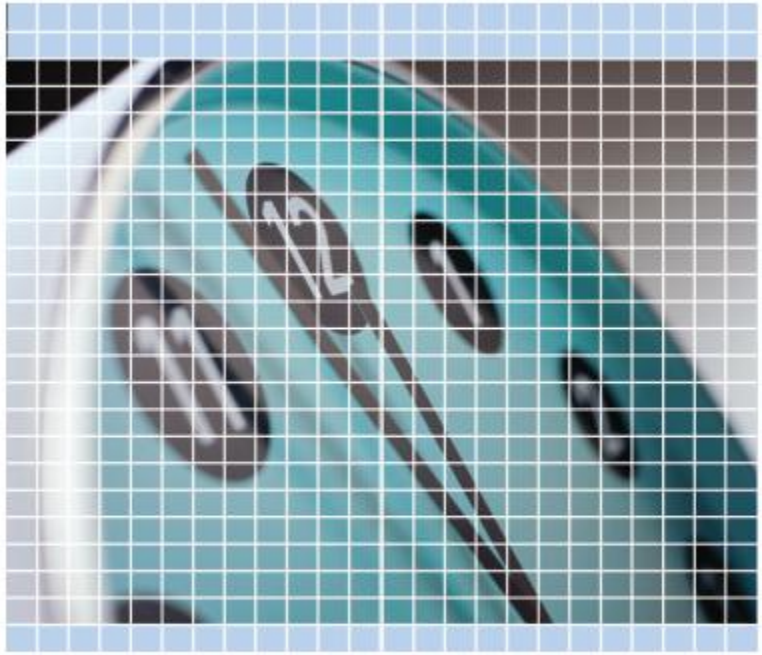
Intel Vtune Hotspot Graph

マルチスレッドプログラミングのステップ

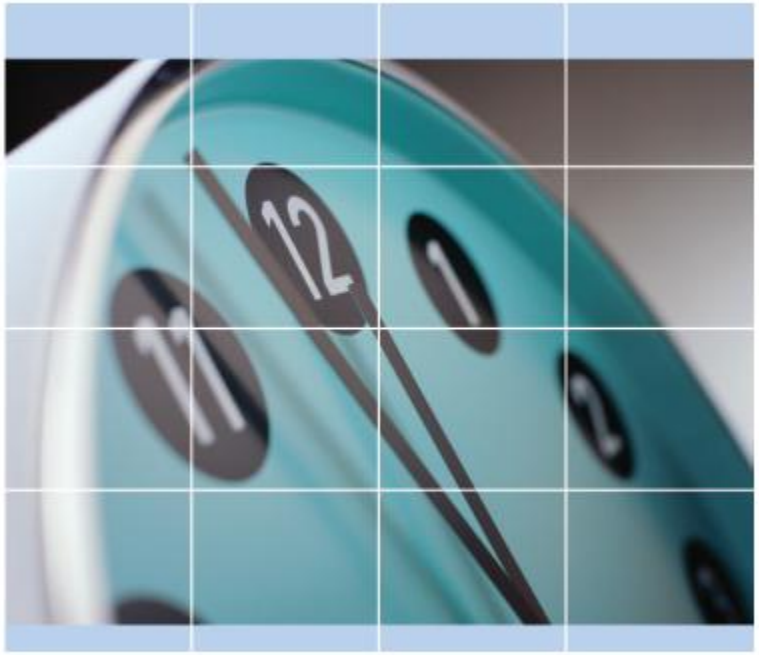
1. **パフォーマンスツールを使いプログラムの動作の詳細な解析を行う。**
   - ホットスポットを見つけることが並列処理では必須
2. **ホットスポットに対してマルチスレッド実行の適用などを検討**
   - データの依存関係などのために並列化出来ない部分などについては、依存関係の解消のために行うプログラムの変更を行う
   - この時、他のハイレベルの最適化手法（ソフトウエアパイプラインやベクトル化）などに影響を与えるときがあるので、この並列化による他のハイレベルの最適化の阻害は避ける必要がある。
   - 並列化の適用時と非適用時の性能を比較検討する必要がある
3. **計算コアの部分について、もし可能であれば既にマルチスレッド向けに高度に最適化されているインテル MKL (Math Kernel Library) などを積極的に利用する**

計算粒度を模式的に示した図

細粒度での領域分割



疎粒度での領域分割

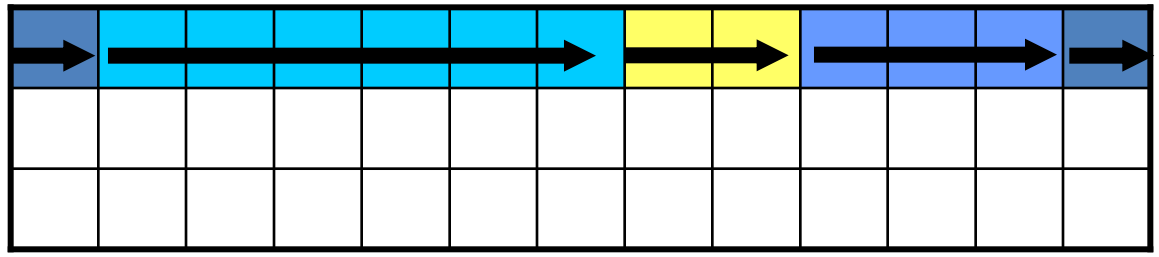


計算粒度を模式的に示した図：領域を例えばスレッド数に分割し、それぞれの領域を個々のスレッドが計算するような場合は疎粒度での領域分割となります。一方、領域を細かく分割し、各スレッドが複数の領域を順次処理するような場合は細粒度での領域分割となります。

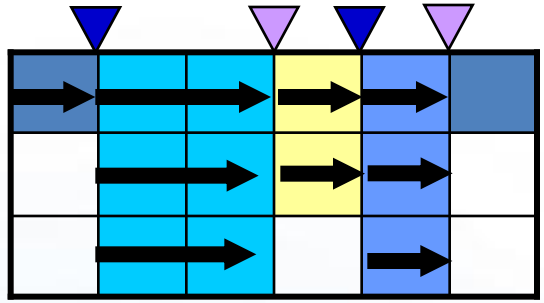
インテル® コンパイラ
OpenMP* 入門
デュアルコア/マルチコア対応アプリケーション開発①より

計算粒度とワークロードの分散

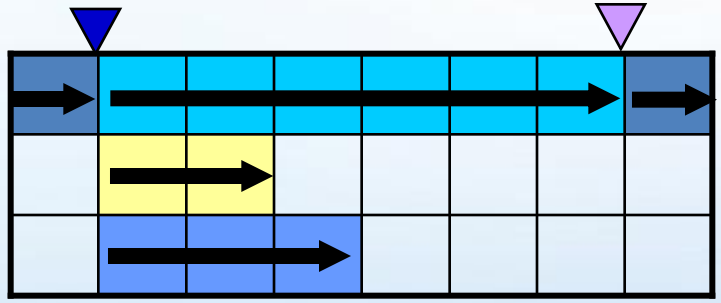
逐次処理



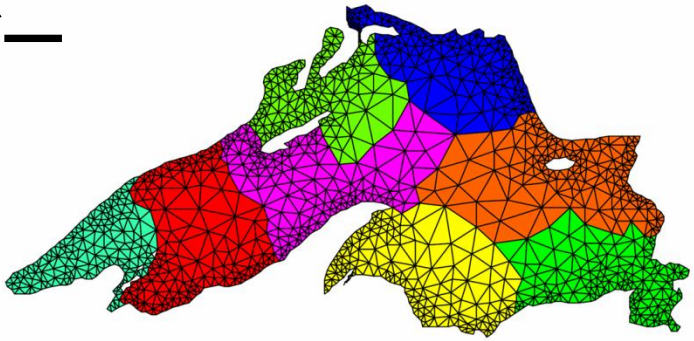
並列処理:細粒度での並列処理



並列処理:疎粒度での並列処理

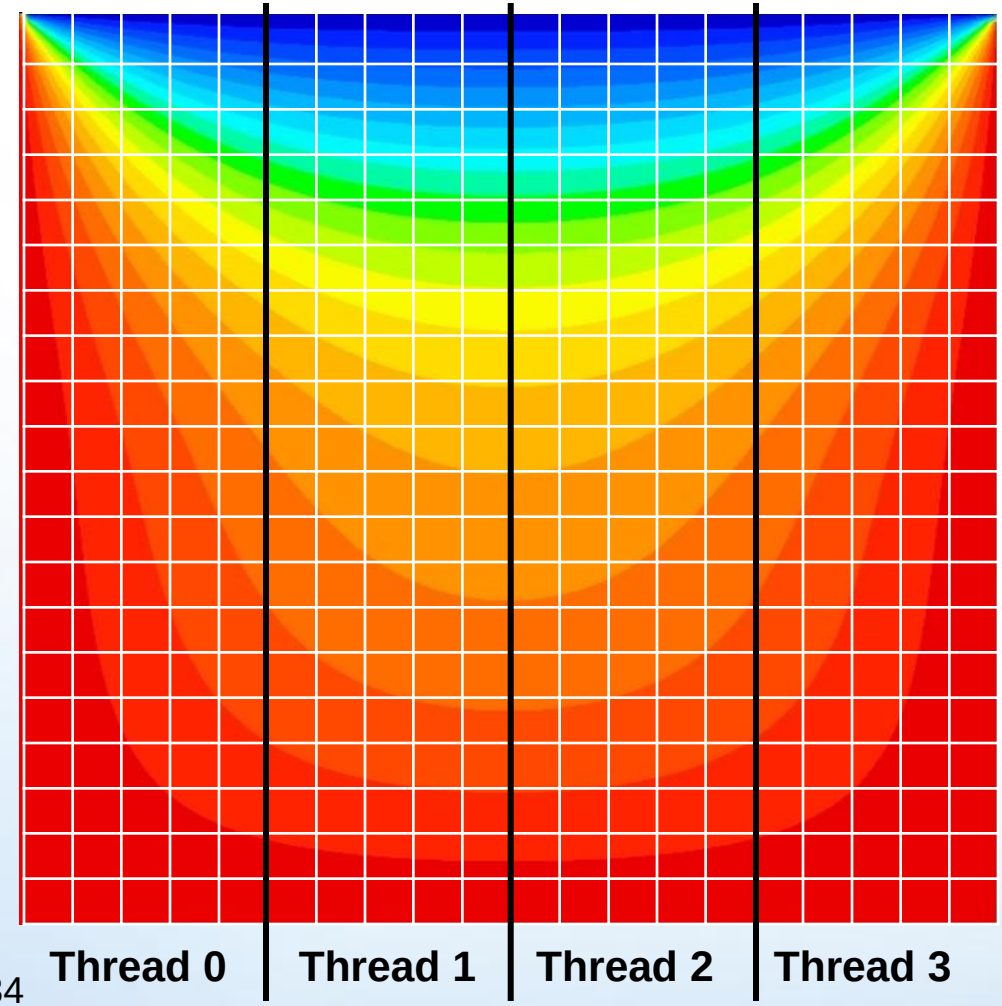


パーティショニング



- **計算処理とデータをより小さな処理単位とデータに分割**
- **領域分割（Domain decomposition）**
  - データの細分化
  - 細分化したデータに対する処理の相互関係に関しての検討が必要
- **機能分割（Functional decomposition）**
  - 計算処理の細分化
  - 分割された計算処理でのデータの取り扱いに関しての検討が必要

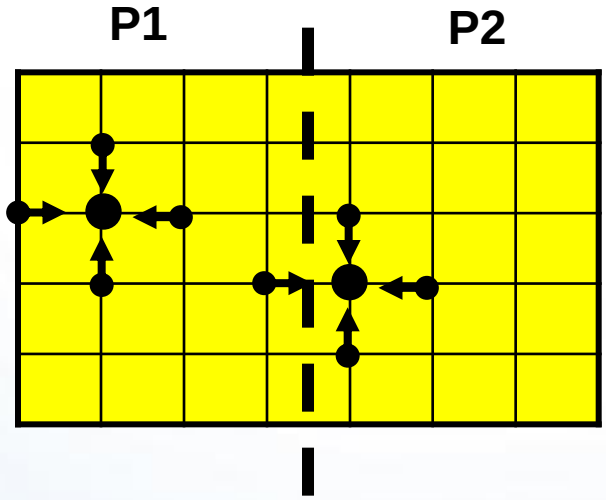
熱伝導方程式 –ポアソン方程式



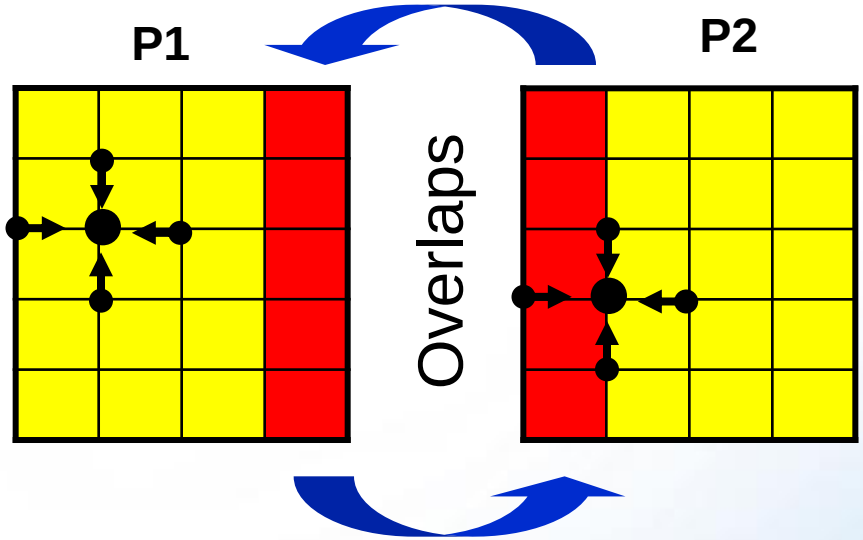

```
float w[*][*], u[*][*];

Initialize u;
while (!stable) {
    copy w => u;  //swap pointers
    for i = …
        for j = …
            Compute w;
    for i = …
        for j = …
            Sum up differences;
    stable = (avg diff < tolerance);
}
```

領域分割



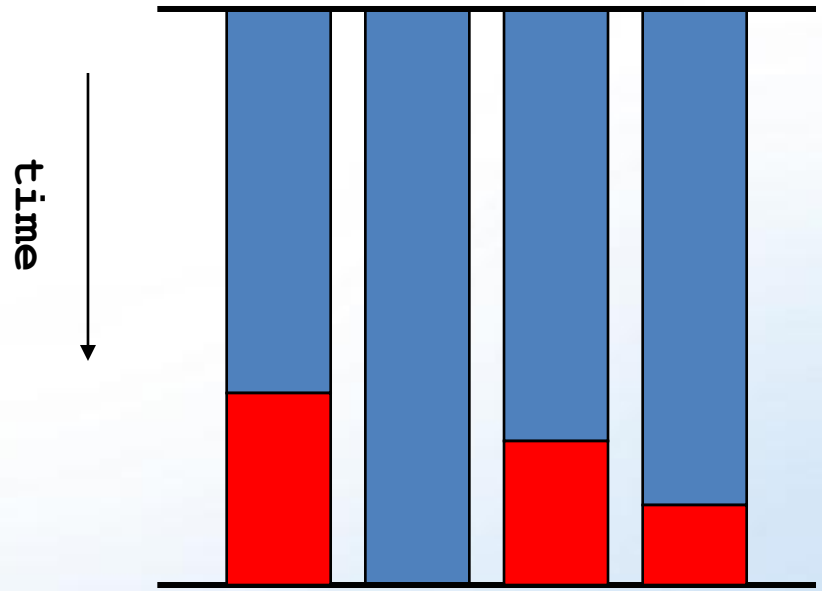
共有メモリでの並列処理



分散メモリでの並列処理

ロードバランス

- **ロードバランス：各タスクのワークロードの分担比率の問題**
  - 並列計算の速度向上は、もっとも時間のかかるタスクの処理時間に依存する
  - ロードバランスが悪い場合には、スレッドの待ち時間が大きくなるという問題が発生
  - マルチスレッドでの並列処理では、各スレッドが等分なワークロードを処理することが理想
- **対策**
  - 並列タスクの処理量は、可能なかぎり各スレッドに当分に分散することが必要
  - ワークロードの陽的な分散、スケジューリングオプションなど



通信

- 並列プログラムにあって、逐次プログラムにないものが通信
- （共有メモリでの並列プログラミングでは、通信を意識してプログラムを書くことは実際はないので、一概に並列プログラムとは言えないが….）
- 通信では、通信の際には誰に送るのか，誰から受け取るのかを特定することが必要
- 通信パターンの決定
  - 一対一通信　（ローカル）
  - 集団通信　（グローバル）
- メッセージの順序関係には依存関係がある
  - 同期と非同期

同期処理

分散メモリシステム

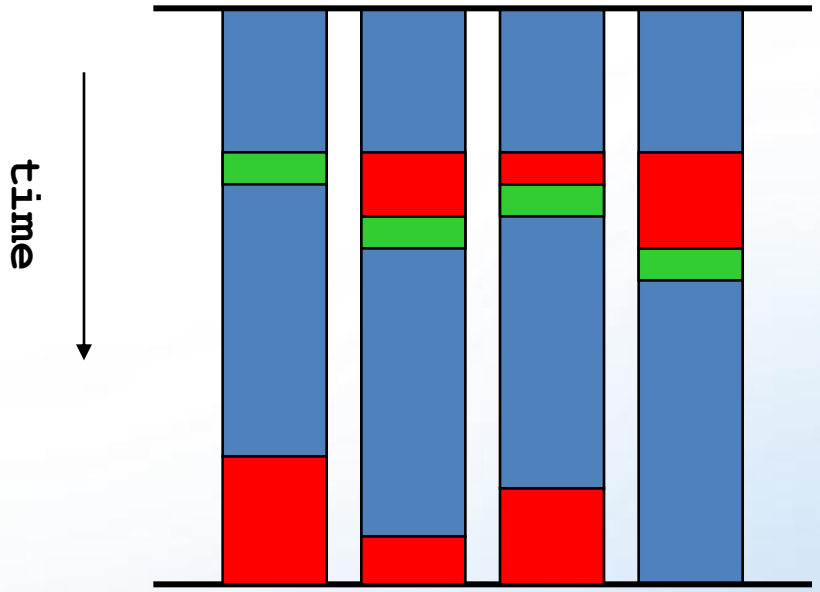
- **共有データを持たず、各プロセスが、独自のメモリ領域を持つ**
- **従って、同期 = 通信となる**
  - MPIにおいては、同期通信を行った場合、データ転送の終了まで、その実行を待つことになる
- **データへのアクセス制御**
  - あるプロセスが、他のノード上のa[i]のデータを必要とした場合、そのデータを転送し、その転送が終了するまで、計算を進めることはできない

共有メモリシステム

- **同期処理は非常に重要**
- **データへのアクセス制御**
  - バリア同期
  - クリティカル・セクション
  - 共有メモリAPIでは、メモリ上のa[i] は、いつでもアクセス可能であるが、そのデータの更新時期やアクセスのための同期処理はユーザの責任となる

スレッド実行時の同期処理

- **並列処理においては、複数のスレッドが同時に処理を行うため、スレッド間での同期処理が必要**
  - 全てのワークシェアリングの終了時に同期処理を行う
  - プログラムによっては、不要な同期処理がプログラム中に加えられる可能性がある
- **クリティカルセクションのような並列実行領域内での排他制御も、スレッド数が多くなると性能に大きな影響を与えることになる**

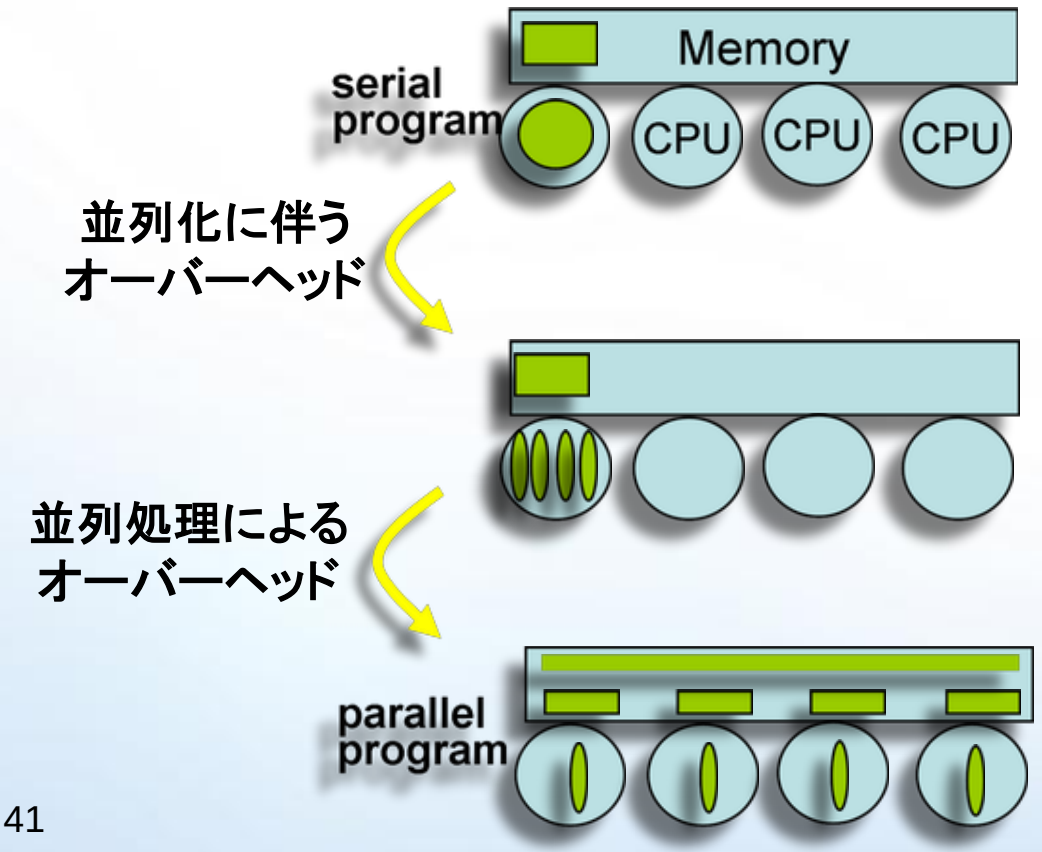


並列化の阻害要因

- **“ステート”を伴うサブプログラム**
  - 擬似乱数生成
  - ファイル I/O ルーチン
- **依存関係があるループ**
  - ある反復で書き込まれ、別の反復で読み取られる変数
  - ループキャリー：値をある反復から次の反復に運ぶ
  - 帰納変数：ループごとにインクリメントされる
  - リダクション： 配列を単一データに変換する
  - 循環：次の反復に情報を伝える

並列化における性能劣化の原因

- 並列化によって、逆に性能が劣化した場合の原因について



オーバヘッドの原因:

- 並列実行のスタートアップ
- 短いループ長
- 並列化のためにコンパイラが追加するコード
- 最内側ループの並列化
- 並列ループに対する最適化の阻害

- 不均等な負荷
- 同期処理
- メモリアクセス(ストライド)
- 共有アドレスへの同時アクセス
- 偽キャッシュ共有など

並列ループの選択

- **並列化の適用は、可能なかぎり粒度を大きくすること**
  - ループでの並列化の場合には、最外側のループ
  - より大きなループカウントのループ
  - 関数やサブルーチンの呼び出しを含むループでも、その呼び出しを含んで並列化できるかどうかの検討が必要
- **データの局所性の維持**
  - 可能な限り、全ての共有データに対する各スレッドの処理を固定する
  - 複数のループを並列化し、それらのループで同一の共有データにアクセスするのであれば、並列化の適用を同じループインデックスに対して行う

並列化の阻害要因とその対策

- **OpenMPによるマルチスレッド化をfor/DOループに適用して並列化する場合、ループ構造によって並列処理の適用ができない場合がある**
  - ループの反復回数がループの実行を開始する時点で明らかになっている必要がある
  - 現在のOpenMPの規格では、whileループなどの並列化はできない
  - 並列処理の適用には十分な計算負荷が必要
  - 並列処理では、for/DOループの実行は相互に独立である必要がある

1.

```
do i=2,n
   a(i)=2*a(i-1)
end do
```

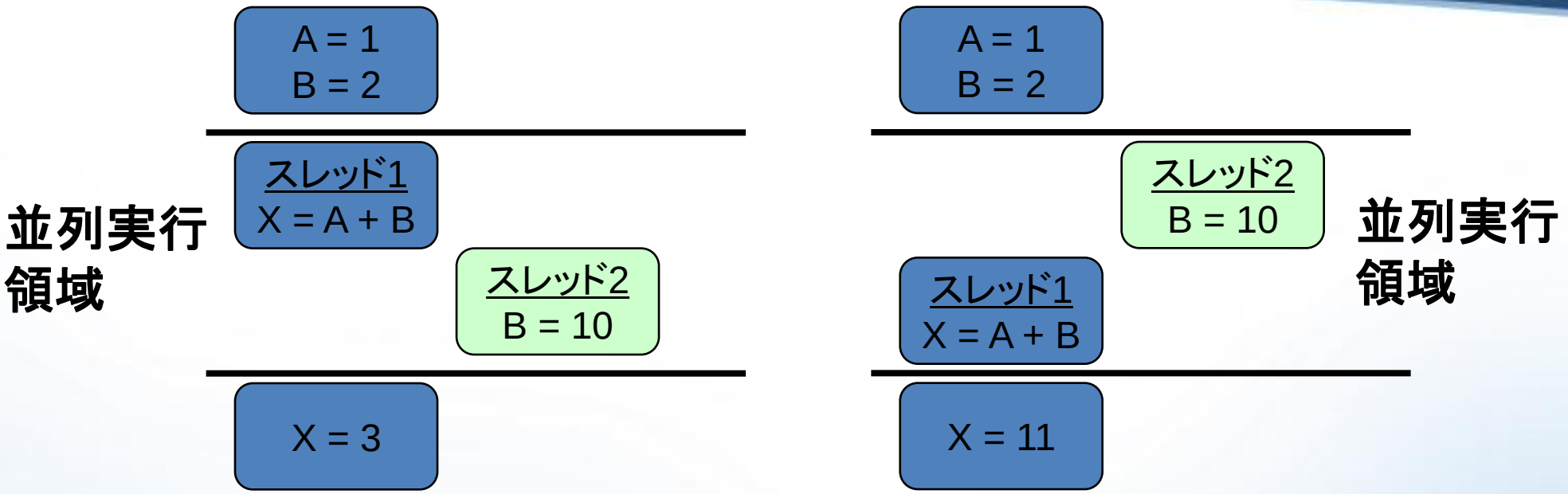
2.

```
ix = base
do i=1,n
  a(ix) = a(ix)*b(i)
  ix  = ix + stride
end do
```

3.

```
do i=1,n
   b(i)= (a(i)-a(i-1))*0.5
end do
```

Race Condition (競合状態）



- **共有リソースであるデータへのアクセス順序によって、計算結果が変わることがあります。このような状態をRace Condition (競合状態）と呼びます。マルチスレッドでのプログラミングでは、最も注意する必要のある問題の一つです。**

OpenMP マルチスレッド並列プログラミング

- **OpenMP は、マルチスレッド並列プログラミングのための API (Application Programming Interface)**
  - OpenMP API は、1997年に発表され、その後継続的に、バージョンアップされている業界標準規格
  - 多くののハードウェアおよびソフトウェア・ベンダーが参加する非営利会社（Open MP Architecture Review Board）によって管理されており、Linux、UNIX そして 、Windows システムで利用可能
- **OpenMP は、C/C++ や Fortran と言ったコンパイラ言語ではない**
  - コンパイラに対する並列処理の機能拡張を規定したもの
  - OpenMP を利用するには、インテルコンパイラ　バージョン 9.0 シリーズのような OpenMP をサポートするコンパイラが必要

OpenMP プログラムのコンパイルと実行例

```
$ cat -n pi.c
     1  #include <omp.h>                                  // OpenMP実行時関数呼び出し
     2  #include <stdio.h>                                // のためのヘッダファイルの指定
     3  #include <time.h>
     4  static int num_steps = 1000000000;
     5  double step;
     6  int main ()
     7  {
     8          int i, nthreads;
     9          double start_time, stop_time;
    10          double x, pi, sum = 0.0;
    11          step = 1.0/(double) num_steps;            // OpenMPサンプルプログラム:
    12          #pragma omp parallel private(x)           // 並列実行領域の設定
    13          {       nthreads = omp_get_num_threads(); // 実行時関数によるスレッド数の取得
    14                  #pragma omp for reduction(+:sum)  // "for" ワークシェア構文
    15                  for (i=0;i< num_steps; i++){      // privateとreduction指示句
    16                          x = (i+0.5)*step;         // の指定
    17                          sum = sum + 4.0/(1.0+x*x);
    18                  }
    19          }
    20          pi = step * sum;
    21          printf("%5d Threads : The value of PI is %10.7f¥n",nthreads,pi);
    22  }
$ icc -O -openmp pi.c
pi.c(14) : (col. 3) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
pi.c(12) : (col. 2) remark: OpenMP DEFINED REGION WAS PARALLELIZED
$ setenv OMP_NUM_THREADS 2
$ a.out
    2 Threads : The value of PI is  3.1415927
```

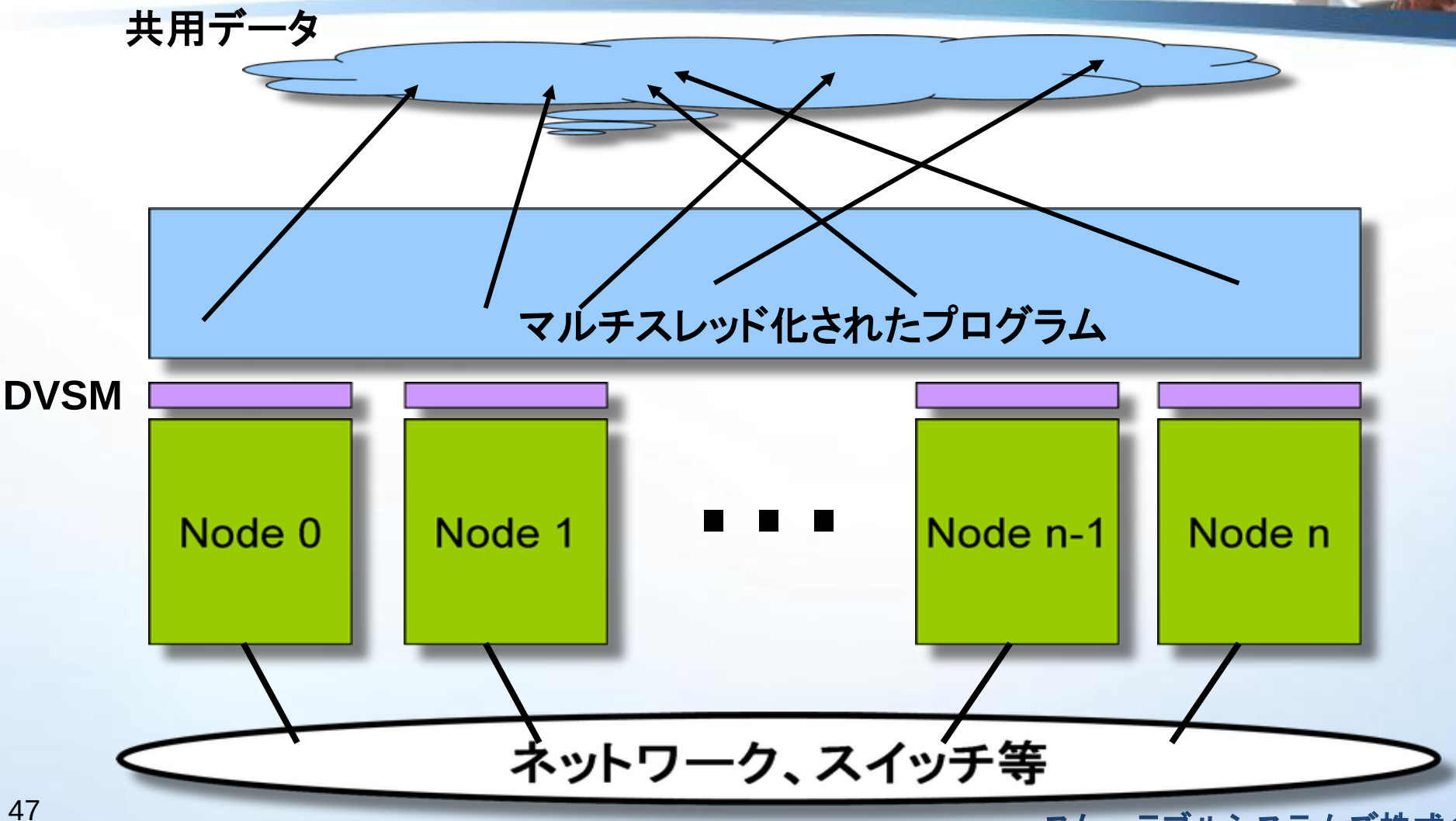
OpenMP指示行

OpenMP実行時関数

コンパイルとメッセージ

環境変数の設定

分散仮想共有メモリ(DVSM)
インテル　クラスタOpenMP



Cluster OpenMP プログラム コンパイルと実行例

クラスタ間共有データの定義

OpenMP実行時関数

コンパイルとメッセージ

並列実行処理環境の設定

```
$ cat -n cpi.c
     1  #include <omp.h>                          // OpenMP実行時関数呼び出し
     2  #include <stdio.h>                        //  のためのヘッダファイルの指定
     3  #include <time.h>
     4  static int num_steps = 1000000;
     5  double step;
     6  #pragma intel omp sharable(num_steps)
     7  #pragma intel omp sharable(step)
     8  int main ()
     9  {
    10  int i, nthreads;
    11  double start_time, stop_time;
    12  double x, pi, sum = 0.0;
    13  #pragma intel omp sharable(sum)
    14  step = 1.0/(double) num_steps;            // OpenMPサンプルプログラム:
    15  #pragma omp parallel private(x)           // 並列実行領域の設定
    16    {
    17      nthreads = omp_get_num_threads();     // 実行時関数によるスレッド数の取得
    18  #pragma omp for reduction(+:sum)          // "for" ワークシェア構文
    19      for (i=0;i< num_steps; i++){          // privateとreduction指示句
    20        x = (i+0.5)*step; // の指定
    21        sum = sum + 4.0/(1.0+x*x);
    22        }
    23    }
    24      pi = step * sum;
    25      printf("%5d Threads : The value of PI is %10.7f¥n",nthreads,pi);
    26  }
    27
$ icc -cluster-openmp -O -xT cpi.c
cpi.c(18) : (col. 1) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
cpi.c(15) : (col. 1) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
$ cat kmp_cluster.ini
--hostlist=node0,node1 --processes=2 --process_threads=2 --no_heartbeat --startup_timeout=500
$ ./a.out
4 Threads : The value of PI is  3.1415927
```

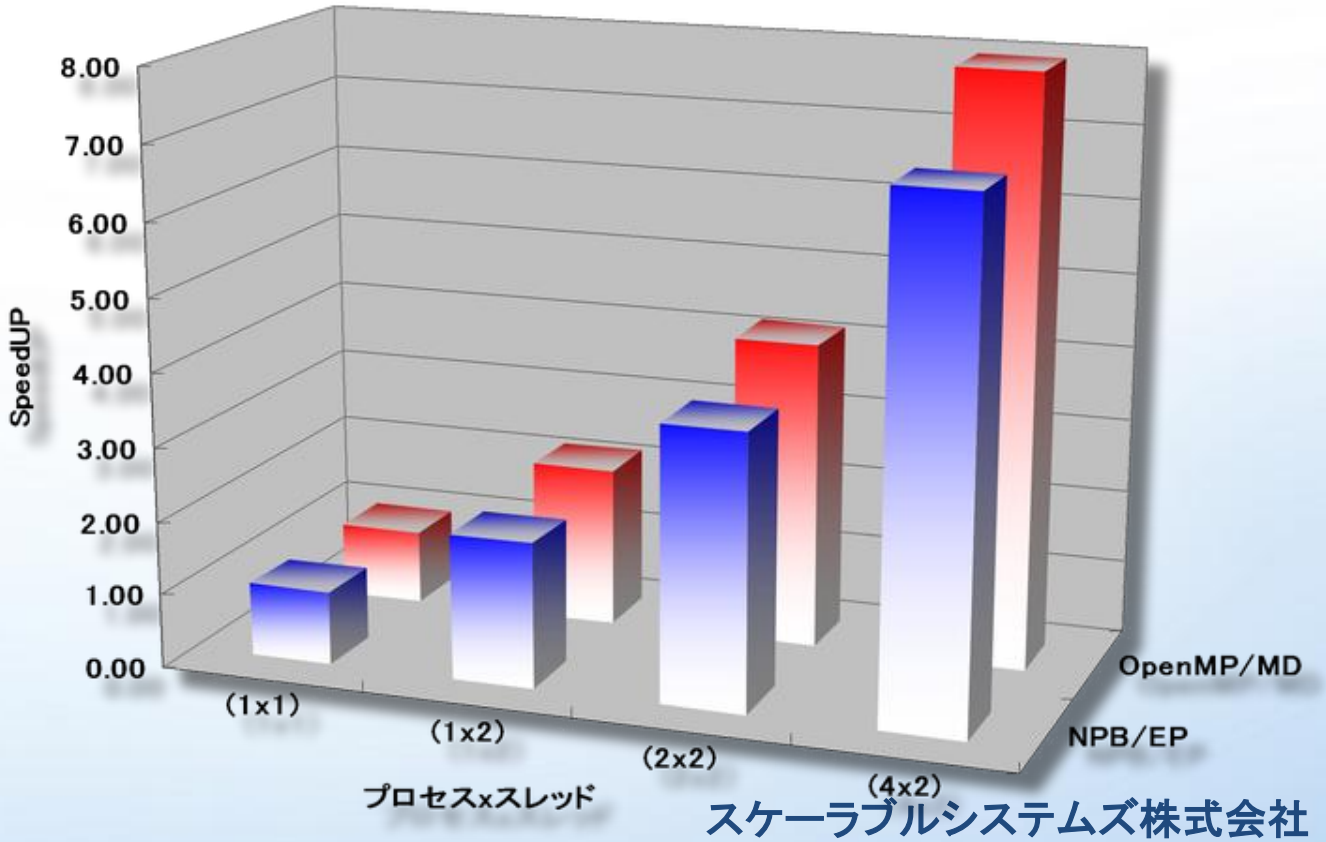
Cluster OpenMP プログラム スケーラビリティサンプル

ベンチマークシステム

- **NEXXUS 4820-PT**
- **2.66GHz/1066MHz FSB/16GB Memory/InfiniBand**

プログラムサンプル

- **NAS Parallel Benchmark / EP ベンチマーク**
- **OpenMPサンプルプログラム（分子動力学サンプル、nparts=10000で実行）**



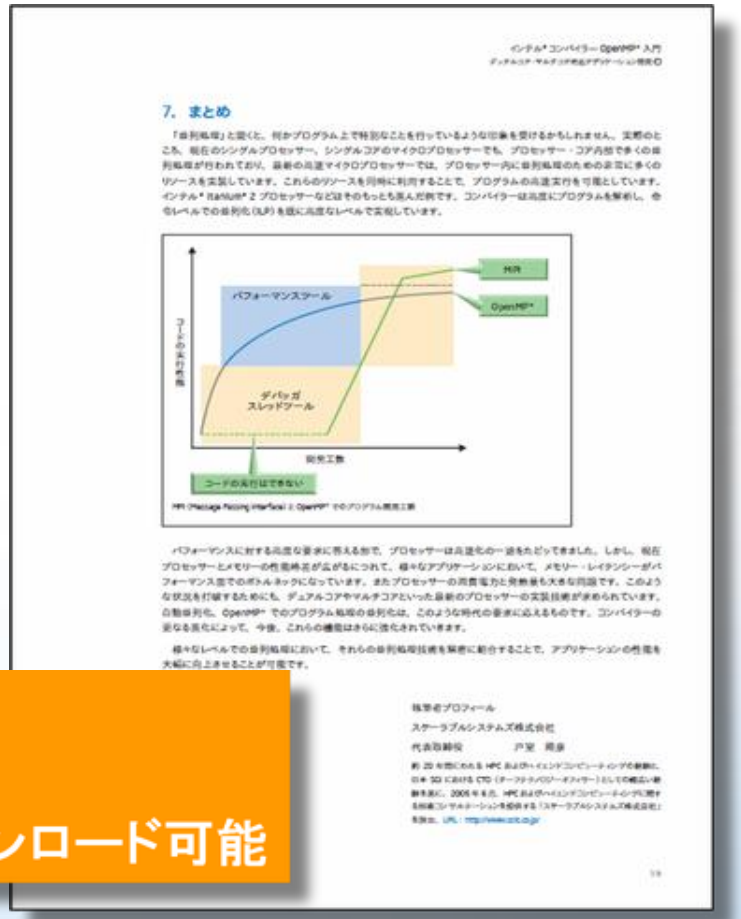
スケーラブルシステムズ株式会社

OpenMPプログラミング入門

http://www.sstc.co.jp/OpenMP

OpenMPによるマルチスレッドプログラミングに関してのトレーニング資料やドキュメントを掲載したホームページです。

インテルコンパイラ
OpenMP入門



インテルコンパイラ
OpenMP入門
インテルHPCリソースセンターからダウンロード可能

スケーラブルシステムズ株式会社

OpenMP日本語ドキュメント

OpenMPに関する日本語ドキュメント
（インテル社ホームページに掲載）

- インテル® C/C++コンパイラー: OpenMP* 活用ガイド **[日本語: PDF 形式 2,051 KB]**
- インテル® コンパイラー: OpenMP* 入門 **[日本語: PDF 形式 1,577 KB]**

インテル HPC リソース・センター
http://www.intel.co.jp/jp/business/japan/feature/HPC/

インテルコンパイラ関連資料(英文)

- Intel Software Network
  - http://www.intel.com/cd/ids/developer/asmo-na/eng/index.htm
- Intel Developer Center – Threading
  - http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/index.htm
- Intel Multi-Core Processing
  - http://www.intel.com/cd/ids/developer/asmo-na/eng/strategy/multicore/index.htm
- Intel Developer Solutions Catalog
  - http://www.intel.com/cd/ids/developer/asmo-na/eng/catalog/index.htm

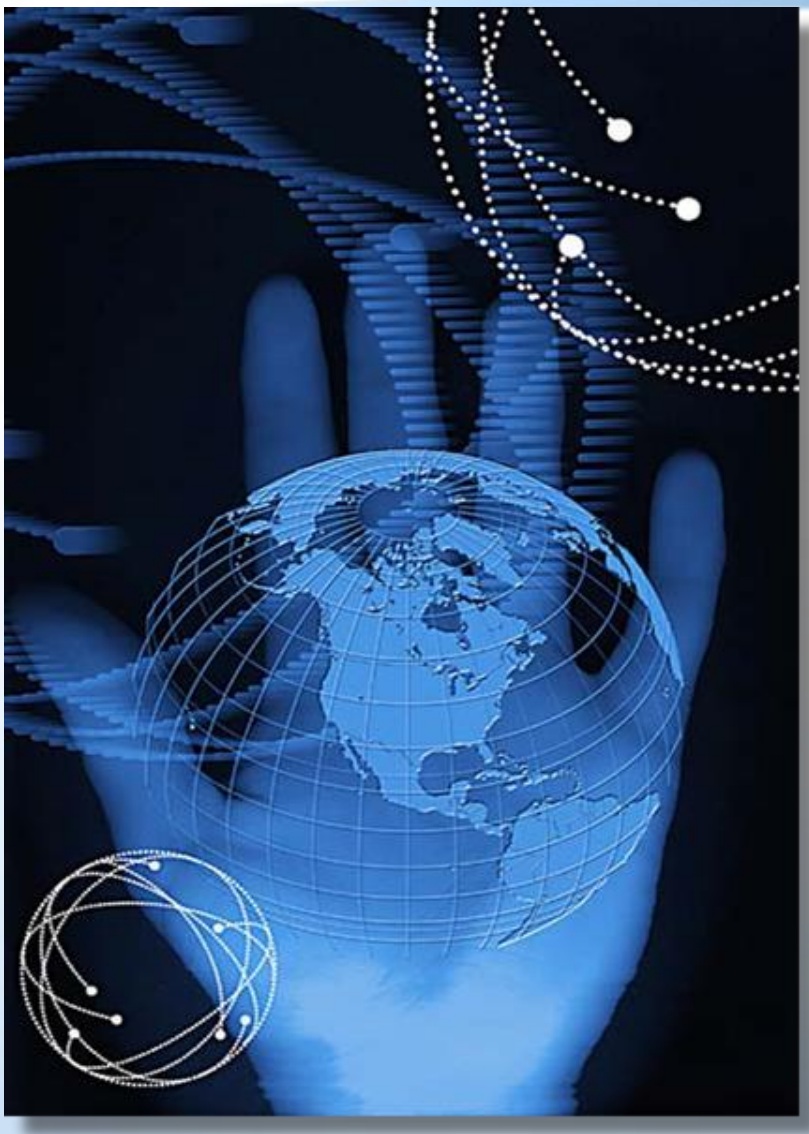
さらに詳しい情報は……

- **弊社のコンサルテーションに関するご提案資料もダウンロード可能です。（非公開WEBページ）別途、弊社に内容等については、お尋ねください。**

お問い合わせ先：

〒102-0083
東京都千代田区麹町3-5-2
BUREX麹町 8F
電話：03-5875-4718
FAX:03-3237-7612
E-mail：biz@sstc.co.jp
http://www.sstc.co.jp

この資料について



社名、製品名などは、一般に各社の商標または登録商標です。無断での引用、転載を禁じます。

なお、本文中では、特に® 、TMマークは明記しておりません。

2006年12月

In general, the name of the company and the product name, etc. are the trademarks or, registered trademarks of each company.

Copyright Scalable Systems Co., Ltd. , 2006. Unauthorized use is strictly forbidden.

2006.12

スケーラブルシステムズ株式会社