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とCluster 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。

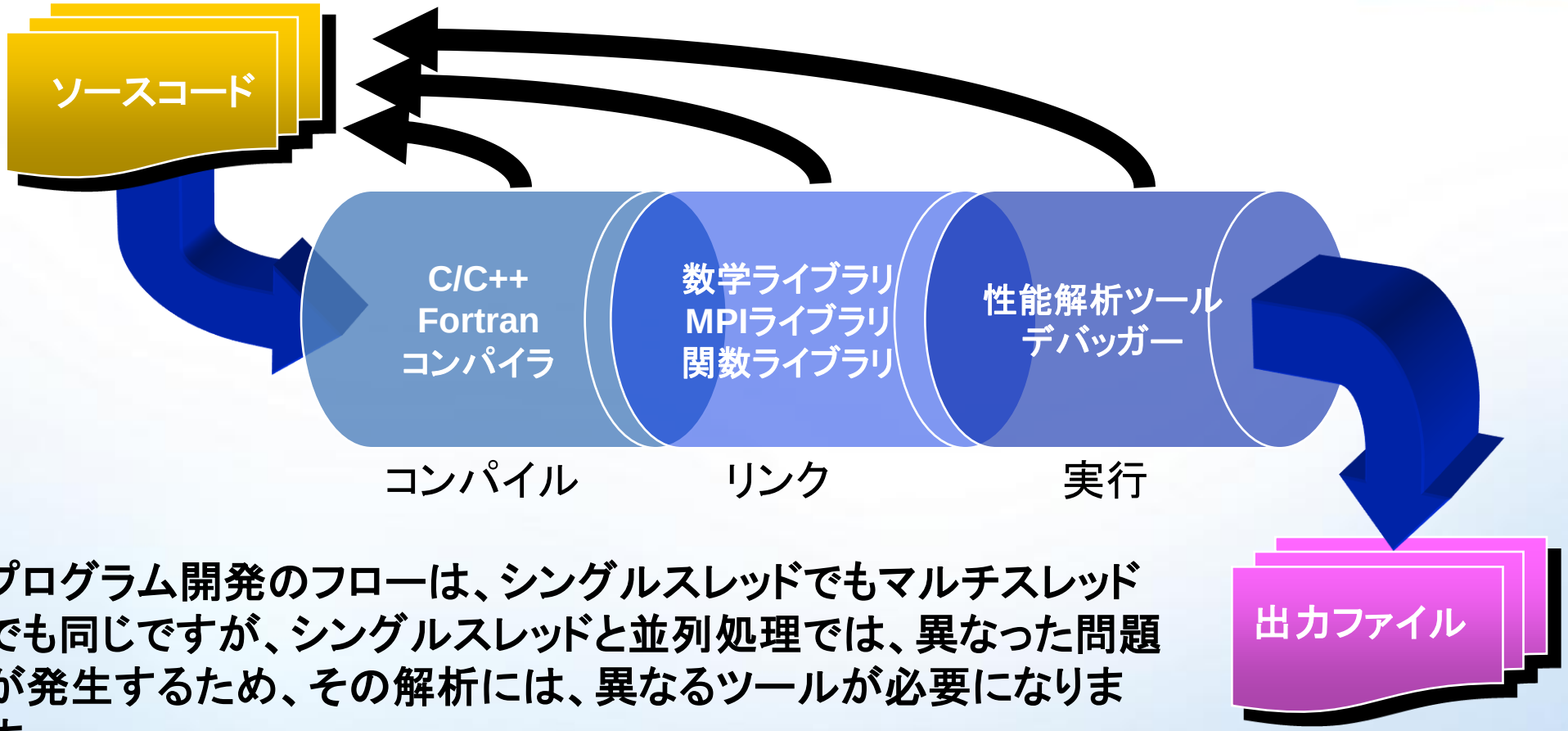
アプリケーションのマルチスレッド化の利点

- **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

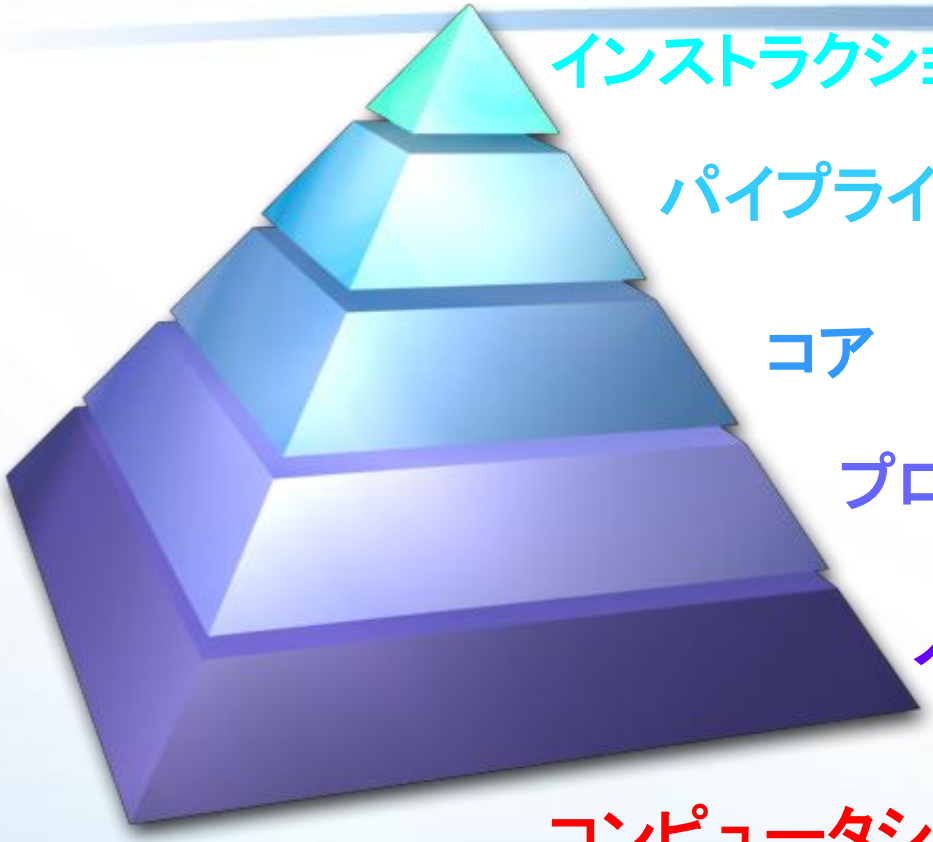
- **Hyper-Threading (HT)　テクノロジー**
  - CPUリソースの有効活用とプロセッサの性能向上のためのハードウエア技術
- **Daul-Core、Multi-Core**
  - 複数のプロセッサコアを一つのプロセッサ上に実装することで、プロセッサの性能向上を図るハードウエア技術
- **Multi-threading (マルチスレッド化）**
  - 複数のプロセッサ（コア）を同時に利用することで、処理性能の向上を図るソフトウエア技術
  - オペレーティングシステムが行っている複数のタスク、プロセスのマルチプロセッサでの多重並列処理をアプリケーションレベルで実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

開発環境



プログラム開発のフローは、シングルスレッドでもマルチスレッドでも同じですが、シングルスレッドと並列処理では、異なった問題が発生するため、その解析には、異なるツールが必要になります。

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

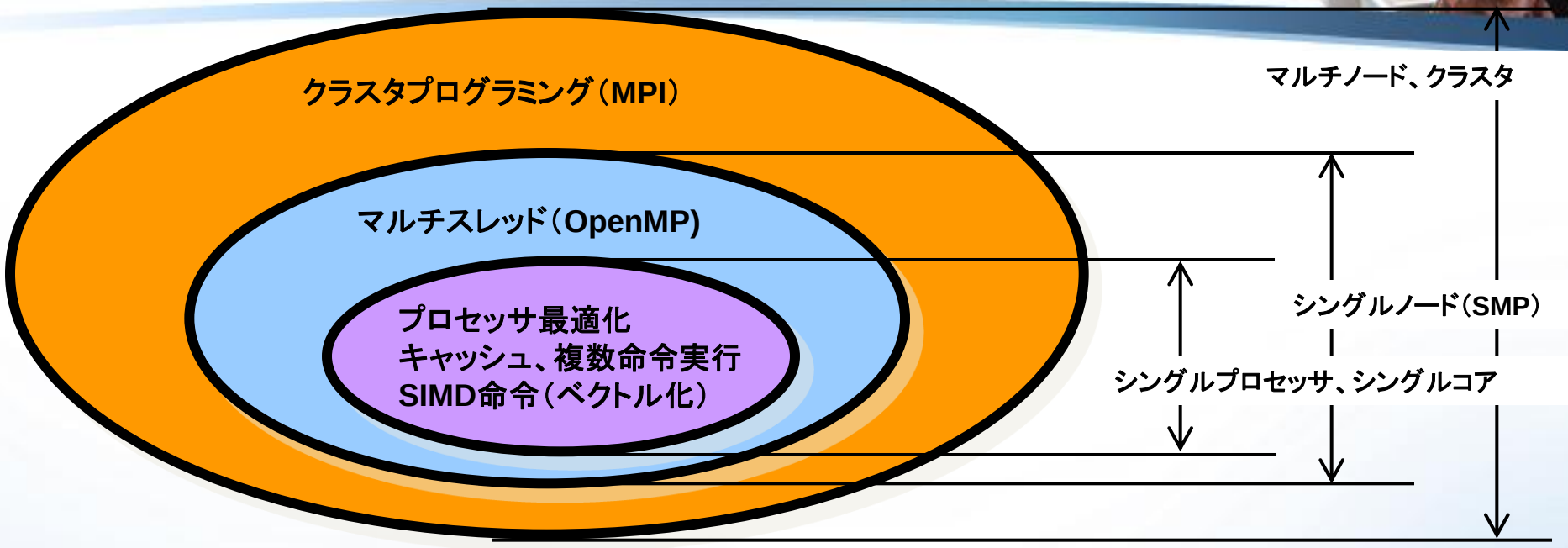
MPIなどによる複数のノード間での並列処理

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

プログラミング階層

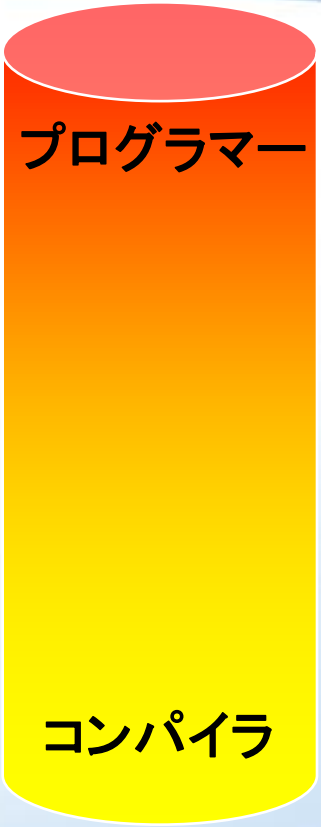
```
do izone = 1, nzone
   ……
   do j = 1 , jmax
      ……
      do i = 1, imax
        ……
        ……..

       end do
```

ノード内、ノード間並列化
MPIやCluster OpenMPなどの利用

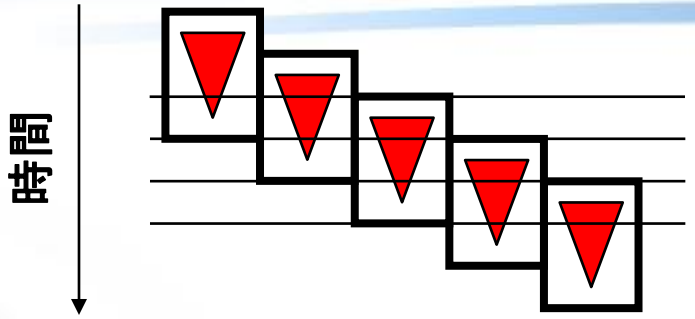
ノード内でのマルチスレッド並列化
OpenMPやスレッドプログラミング

プロセッサリソースの並列利用
ベクトル化
スーパースカラ実行
パイプライン処理
キャッシュ最適化など

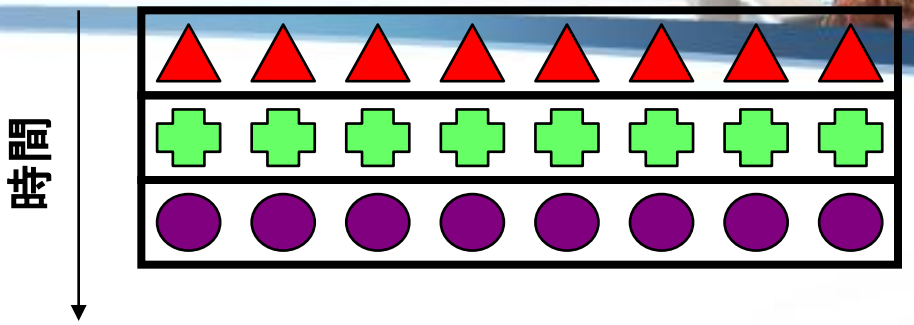


最適化と並列化の適用作業

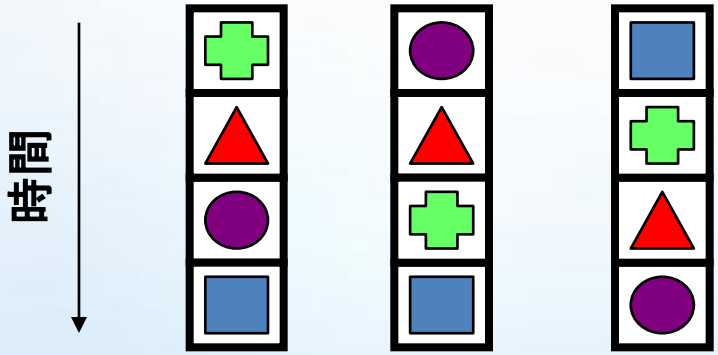
並列性（Parallelism）の利用



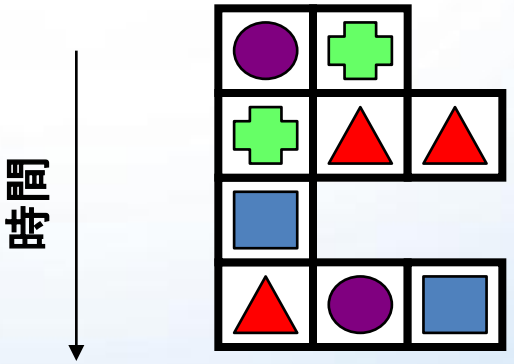
パイプライン処理



データレベル並列処理 (DLP)



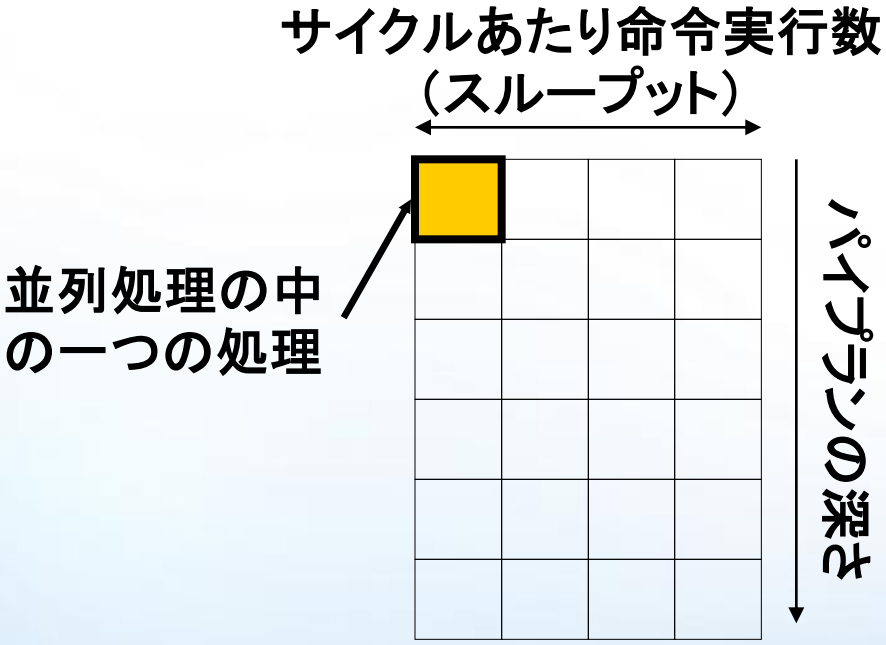
スレッドレベル並列処理 (TLP)



命令レベル並列処理 (ILP)

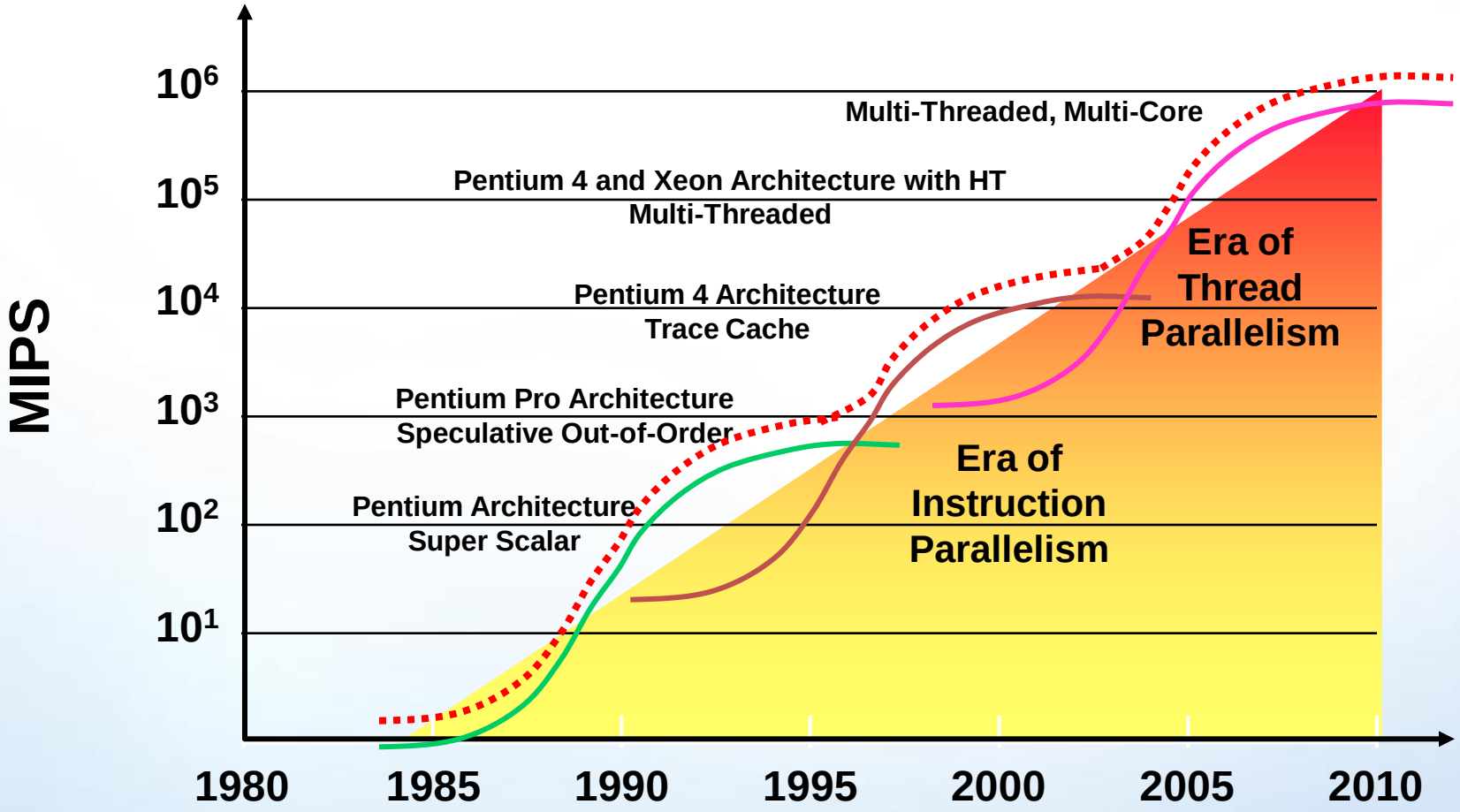
Parallelism = Throughput * Latency
（並列度=スループット*レイテンシ）

複数の命令の同時実行を可能とするスーパースカラ命令では、実行時の同時並列処理の問題と共に、プログラムのアルゴリズムでの命令実行の並列実行を可能とするその依存性解析が重要となり、可能な限り命令実行スロットを埋めるために、動的に命令のリオーダーなども必要です。



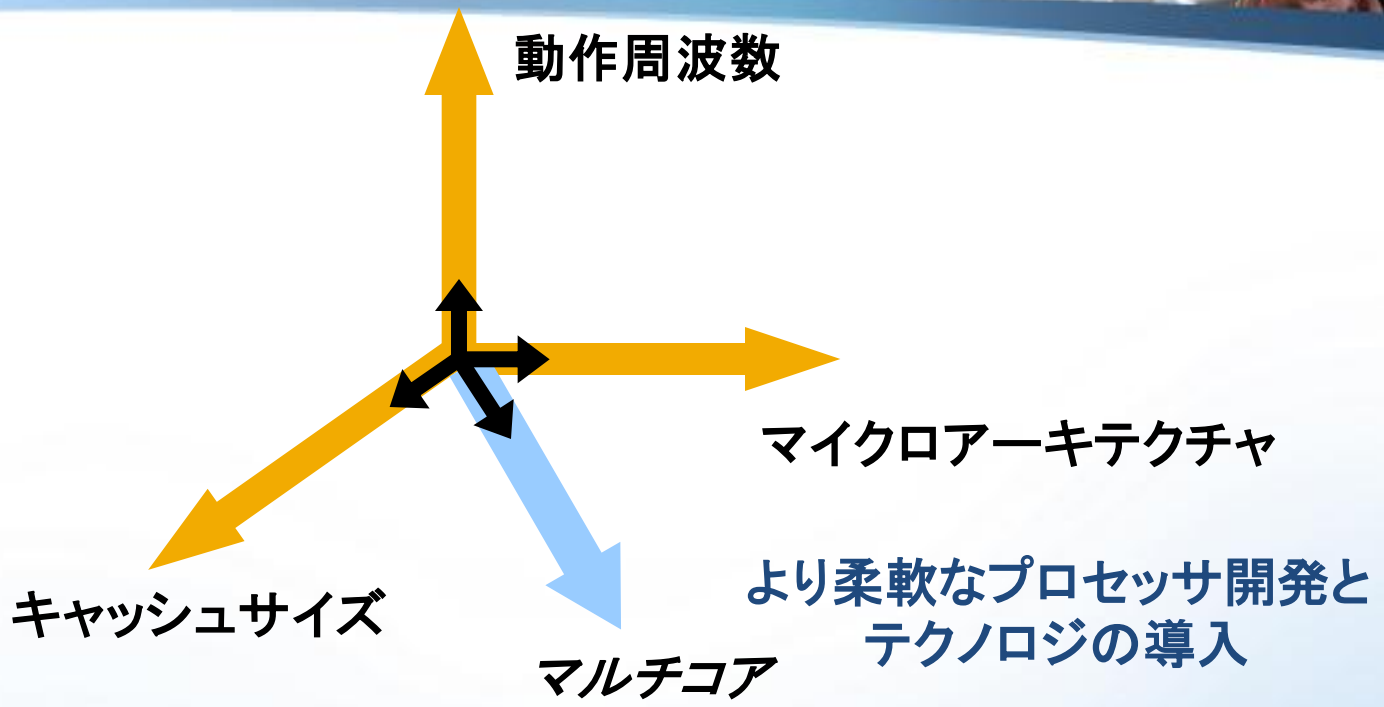
動作周波数の増大は、半導体デバイスのスイッチング遅延の改善のための半導体の微細化とより深いパイプライン化によって、命令実行をより多くのクロックサイクルに分散させ、各サイクルでの処理を減らすことで実現されています。

マイクロアーキテクチャのSカーブ



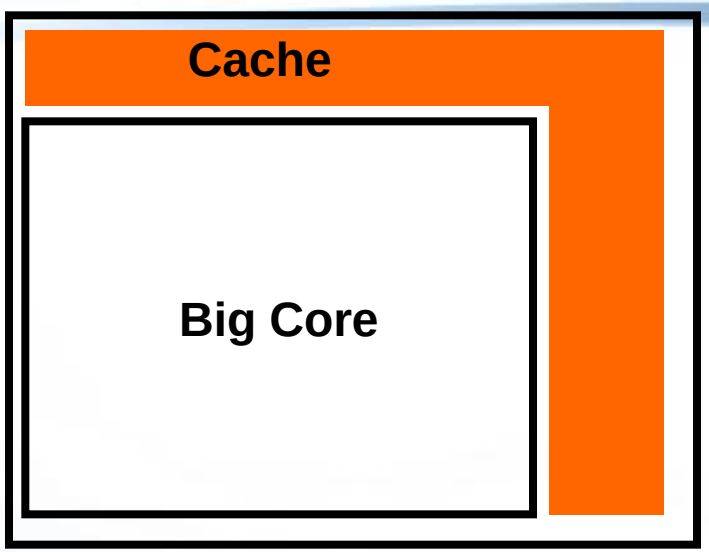
Johan De Gelas, Quest for More Processing Power,
AnandTech, Feb. 8, 2005.
http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2343

新たな次元でのプロセッサ開発

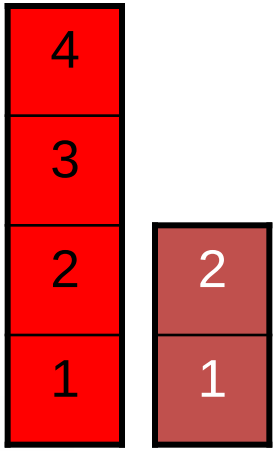


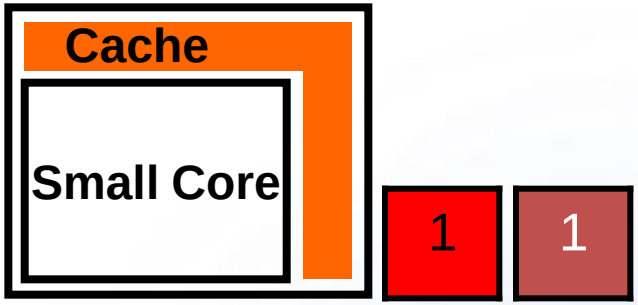
- プロセッサの性能向上のための選択肢が広がる
- 価格性能比の向上を違った次元で提供可能
- 技術的な利点と‘マーケティング’の要求

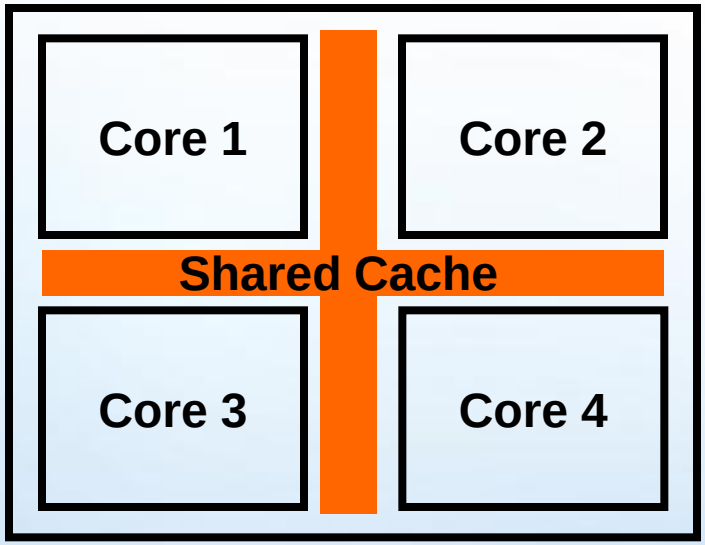
マルチコア：‘性能/消費電力’を改善

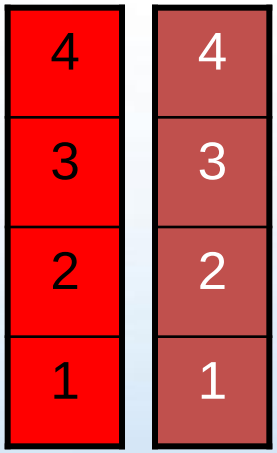


消費電力 / 性能



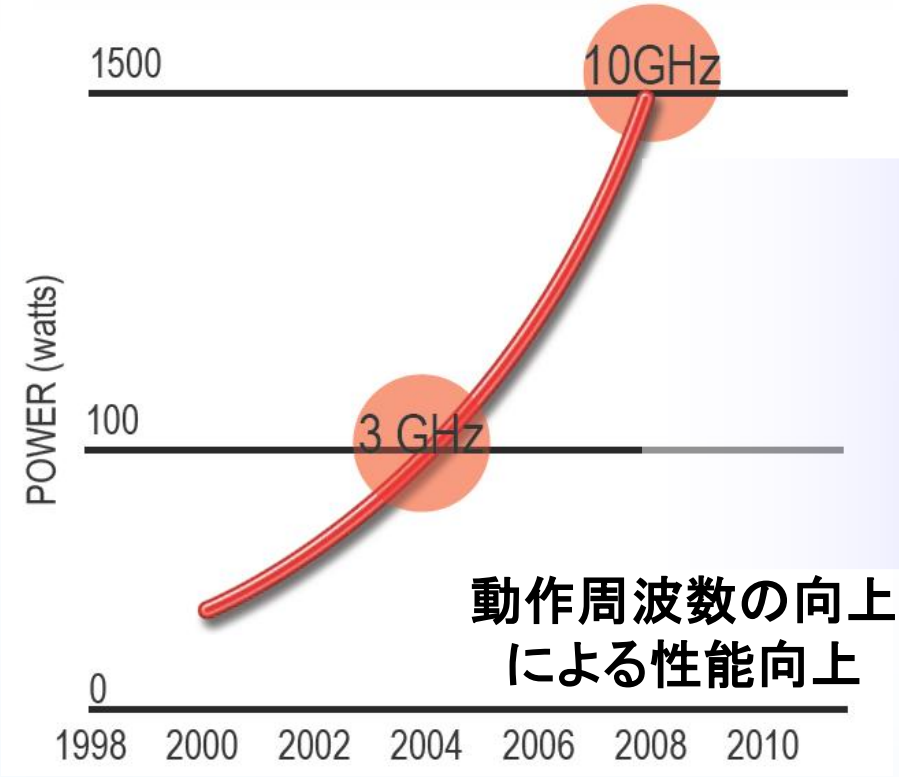






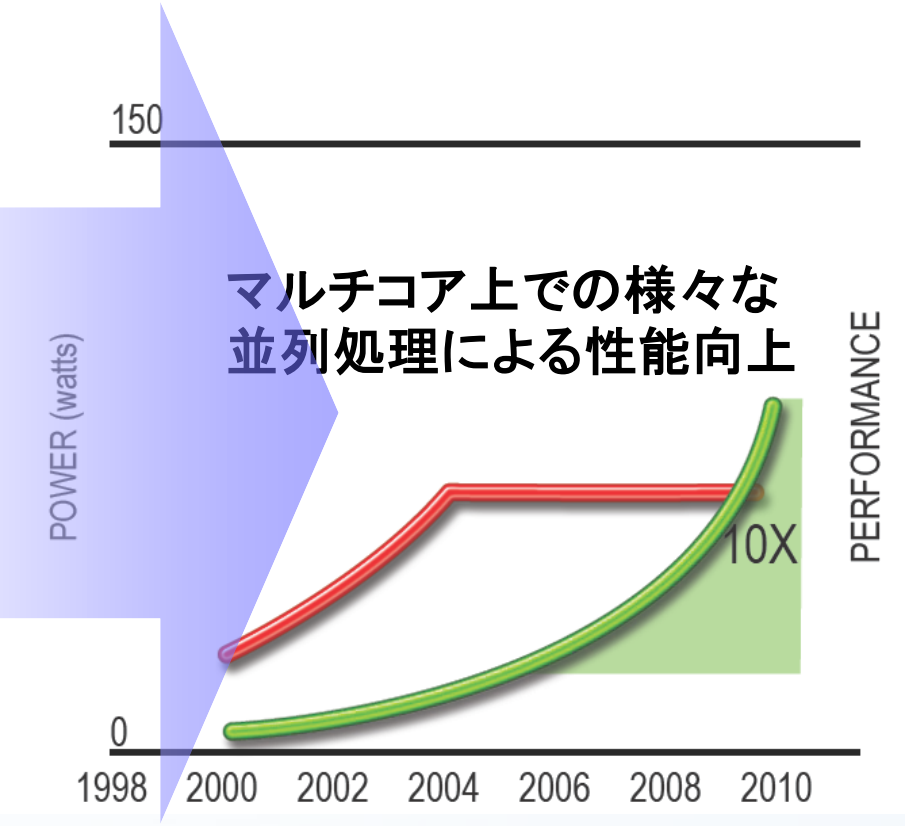
Power ~ コアサイズ
PERFORMANCE ~ √コアサイズ

マイクロプロセッサの開発方針の変遷

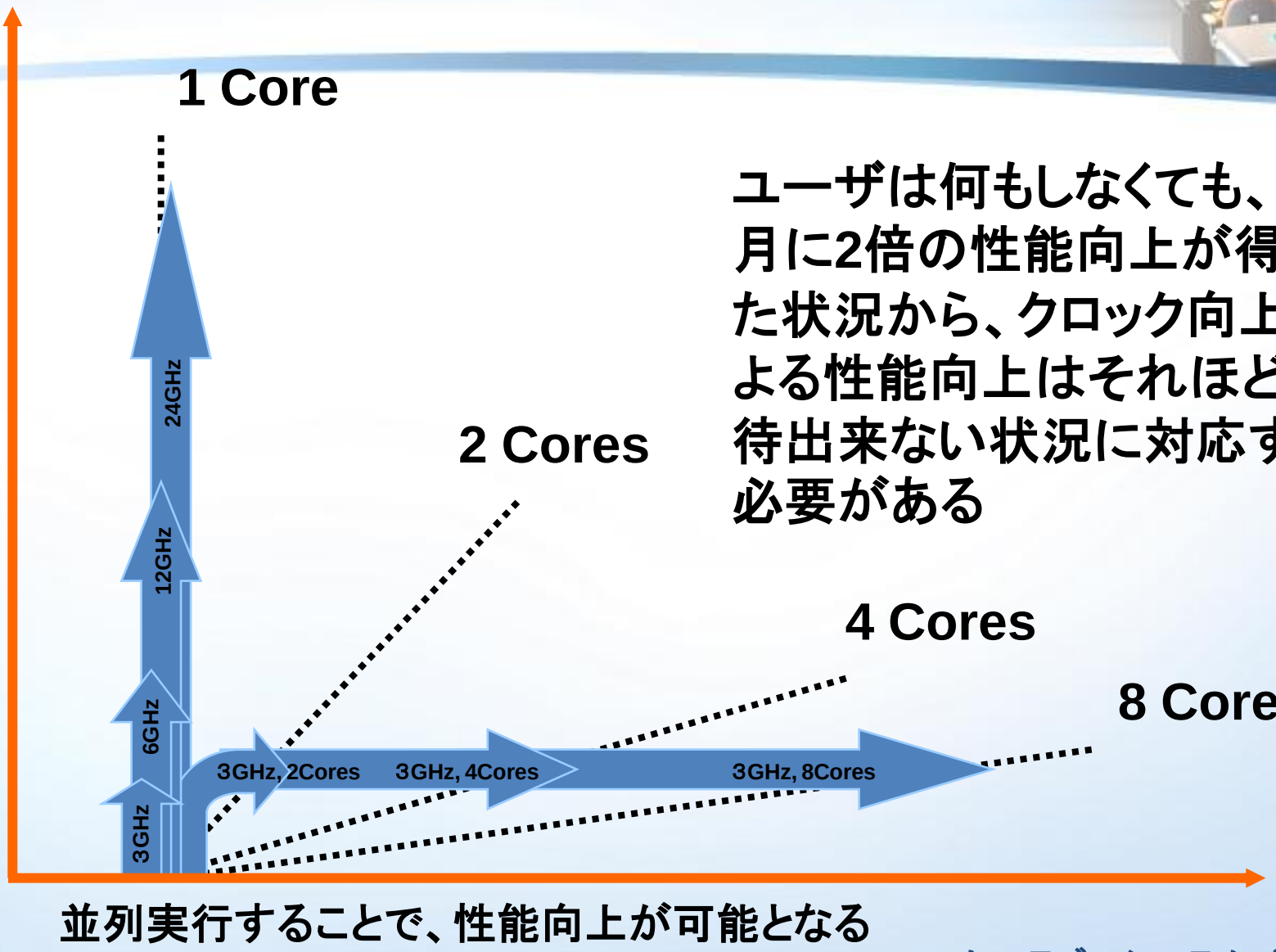


動作周波数の向上
による性能向上

マルチコア上での様々な
並列処理による性能向上

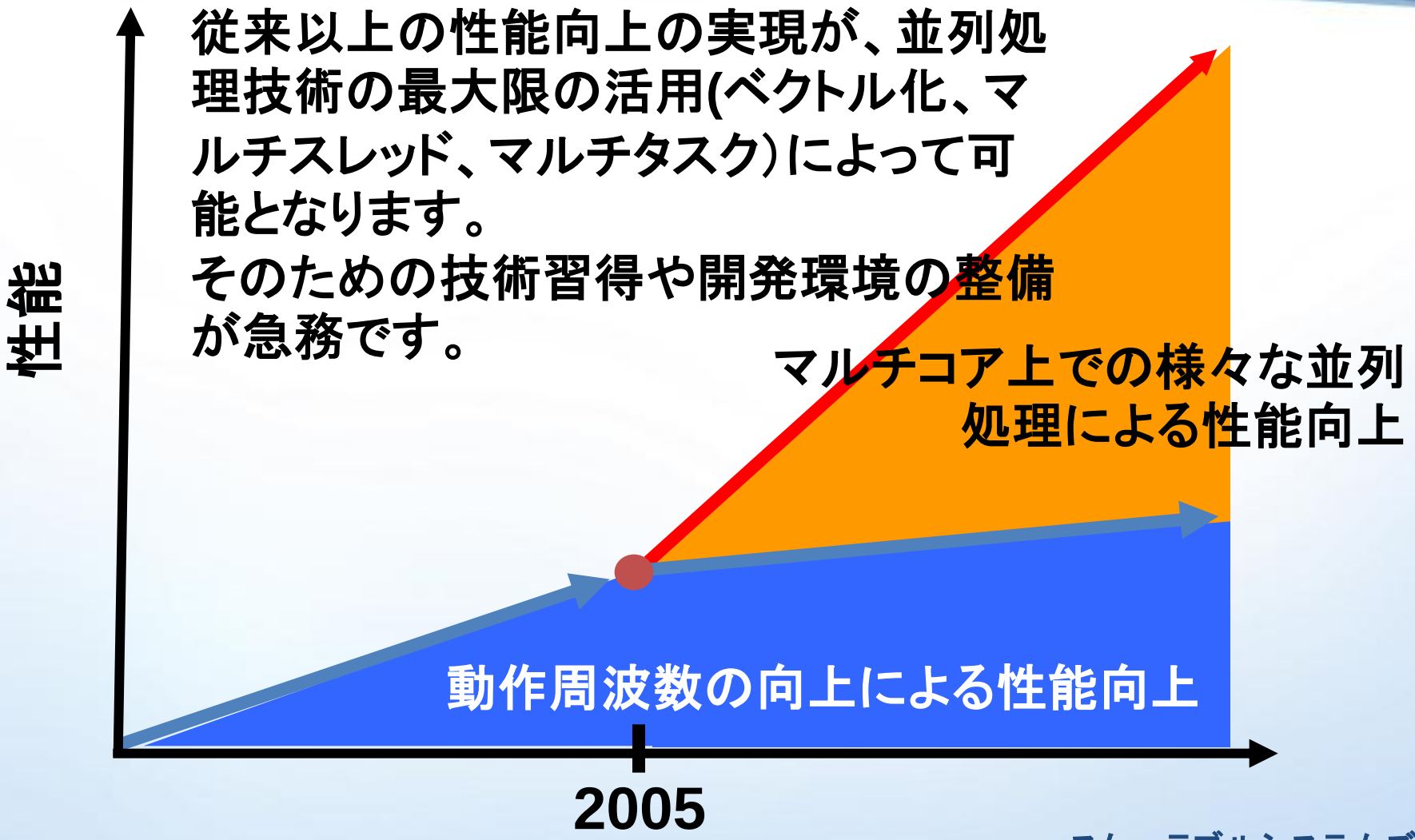


並列処理の重要性



ユーザは何もしなくても、18ヶ月に2倍の性能向上が得られた状況から、クロック向上による性能向上はそれほど期待出来ない状況に対応する必要がある

ムーアの法則
動作周波数からマルチコアへ



ループのベクトル化処理

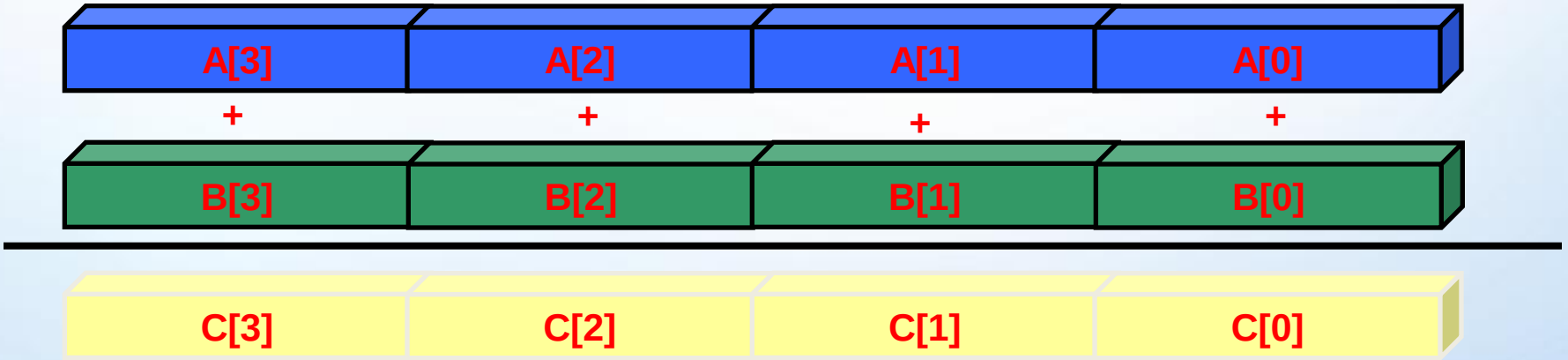
- **プログラム例:**

```
for (I=0;I<=MAX;I++)
    C[I]=A[I]+B[I];
```

- **利用方法:**

(Linux) -[a]xN, -[a]xB, -[a]xP

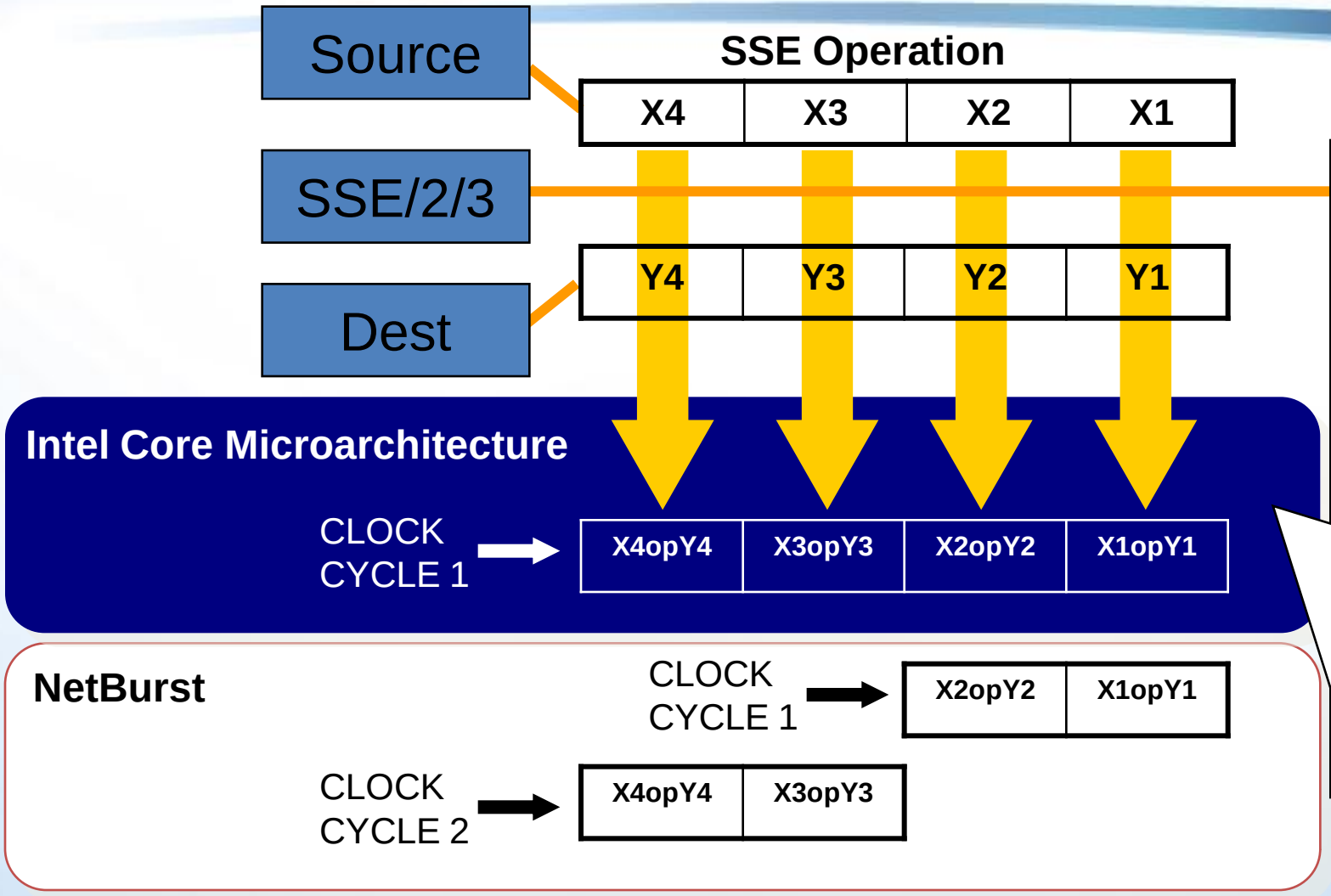
(Windows) -Q[a]xN, -Q[a]xB, -Q[a]xP



x86プロセッサでのSIMD演算

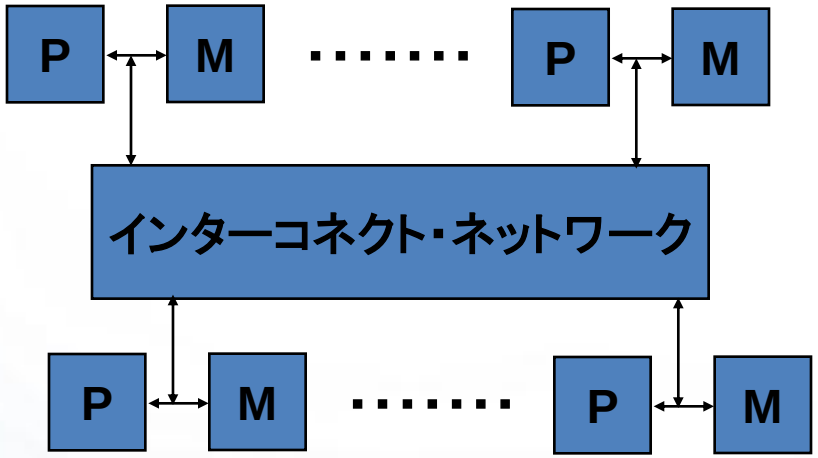
- **コンパイラは、プログラムを解析し、SIMD演算のためのベクトル化を行う**
  - 単なるパターン認識ではなく、プログラムフローを解析してベクトル化を適用
  - ベクトル化は、現在のx86プロセッサでの高速実行において、非常に重要な技術となっている
- **現在のx86プロセッサは、全てSIMD演算をサポート**
  - データ型変換と飽和データ型変換
  - 飽和算術演算（Saturation arithmetic）
  - クリッピング（Clipping）
  - 平均（AVG）及び絶対値（ABS）の計算

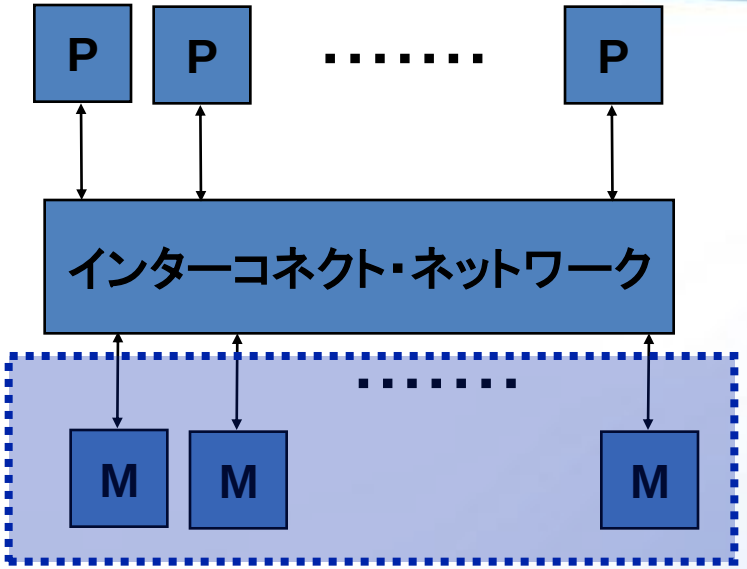
インテルプロセッサでのSIMD処理



各MMX/SSE演算は、128ビットの演算をシングルサイクルで実行可能となる。MMX/SSE演算器は2セットあり、同時実行が可能となる。従って、単精度では、8浮動小数点演算、倍精度では、4浮動小数点演算を1クロックで実行することが出来る。

並列コンピュータシステム





分散メモリシステム

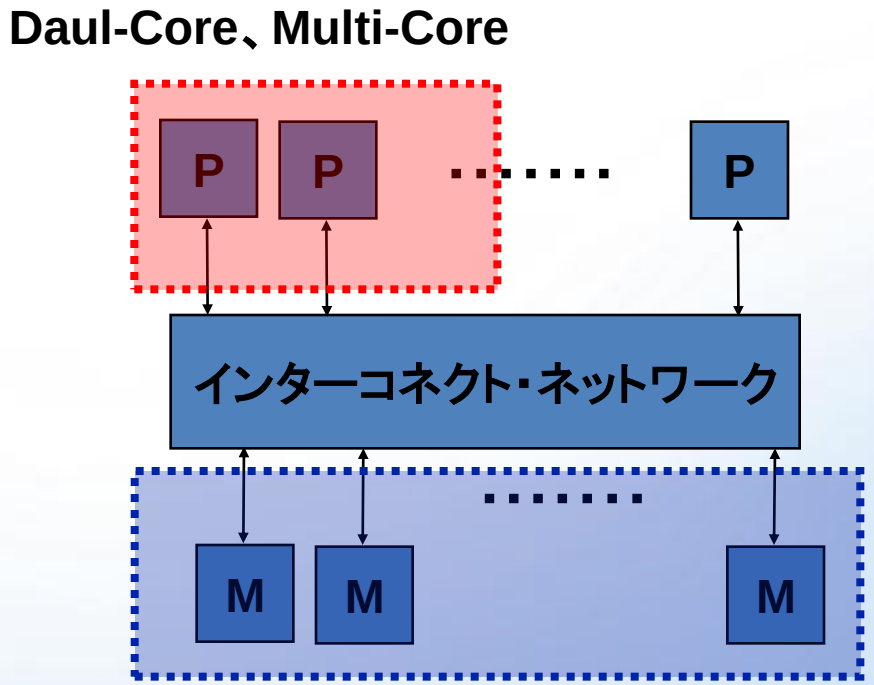
- マルチプロセス
- ローカルメモリ
- メッセージ通信（メッセージ・パッシング）によるデータ共有

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

- **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

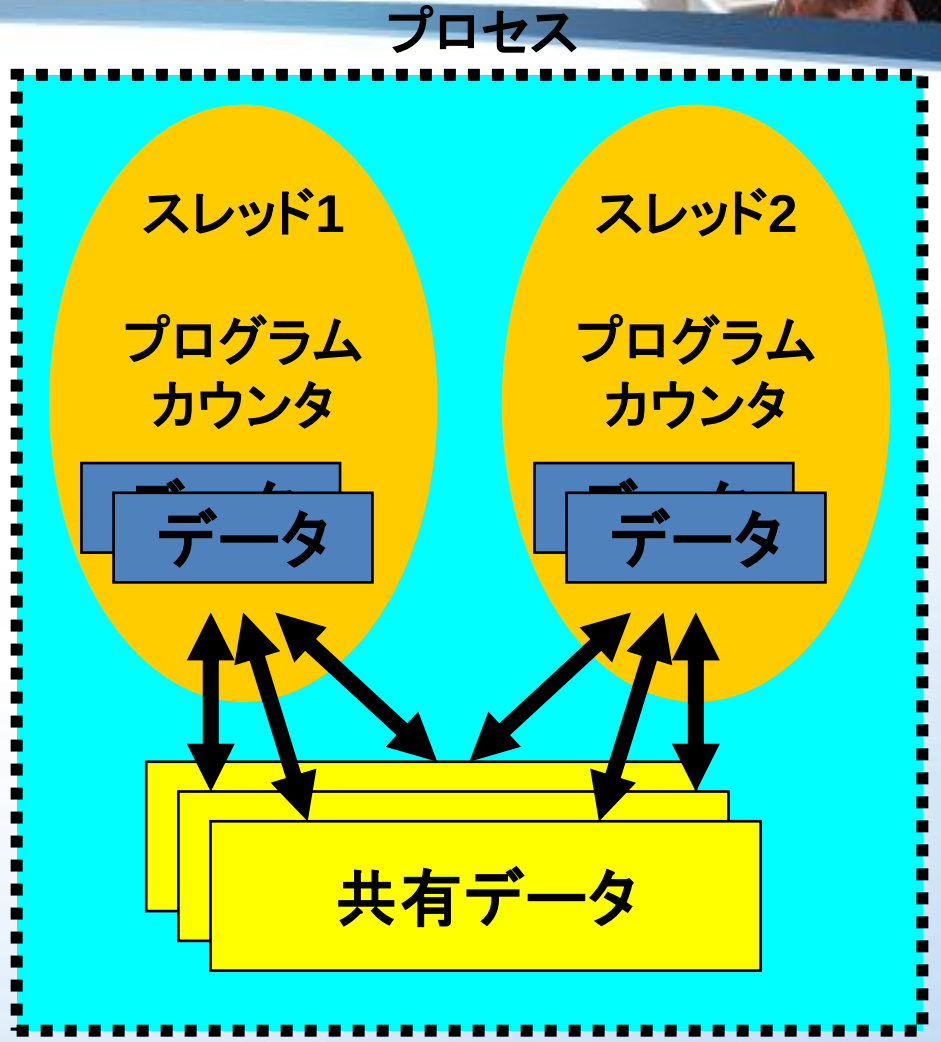
- **マルチタスク**
  - 複数のタスクを同時に処理する
  - データベースやWEBなどのシステムなどでの並列処理
    - 一度に複数のユーザからの大量のデータ処理の要求
  - プロセス単位でのOSによる並列処理
    - 各タスクは複数のプロセスやスレッドを利用して処理を行う
    - プログラム自身の並列化は必ずしも必要ない
- 並列計算
  - **特定の問題に対して、その計算処理を複数のプロセッサを利用して高速に処理する並列処理**
  - **対象となる問題を複数のコア、プロセッサを同時に利用して短時間で解く**
  - **並列プログラミングAPIを利用して、複数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

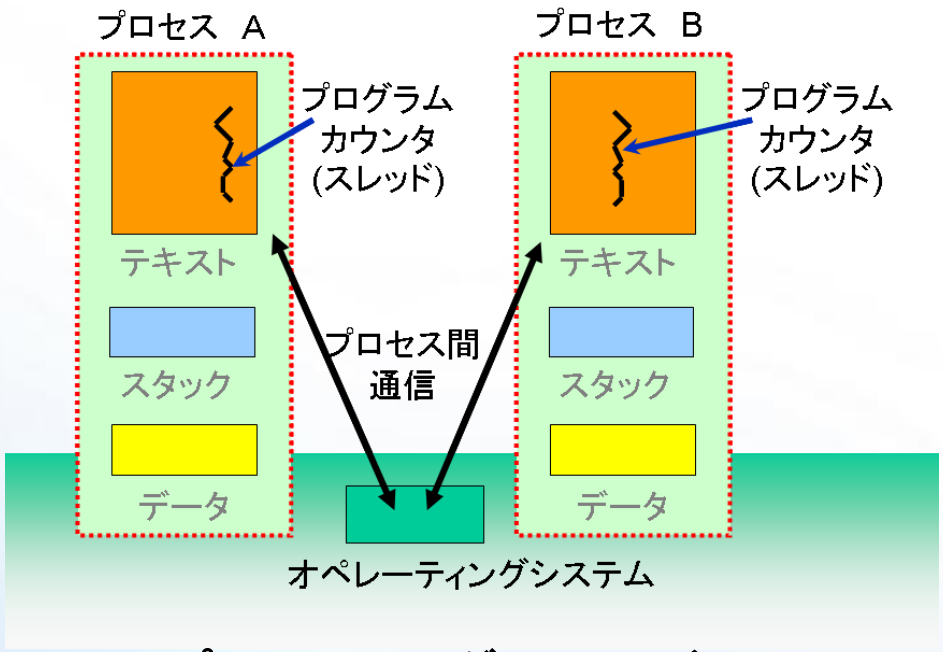
- **並列処理**
  - 「同時に複数の処理（タスク）をOSが処理すること」
  - OSによる処理単位がプロセスとスレッドということになります。
- **プロセス**
  - OSは、要求されたタスクに対して複数のプロセスを起動してその処理を行います。
  - 複数のプロセスを利用して行う並列処理がマルチプロセスとなります。
- **スレッド**
  - これらのプロセス内で生成されて実際の処理を行うのがスレッドとなります。
  - プロセス内で複数のスレッドを生成して、並列処理を行うことをマルチスレッドと呼びます

プロセスとスレッドについて

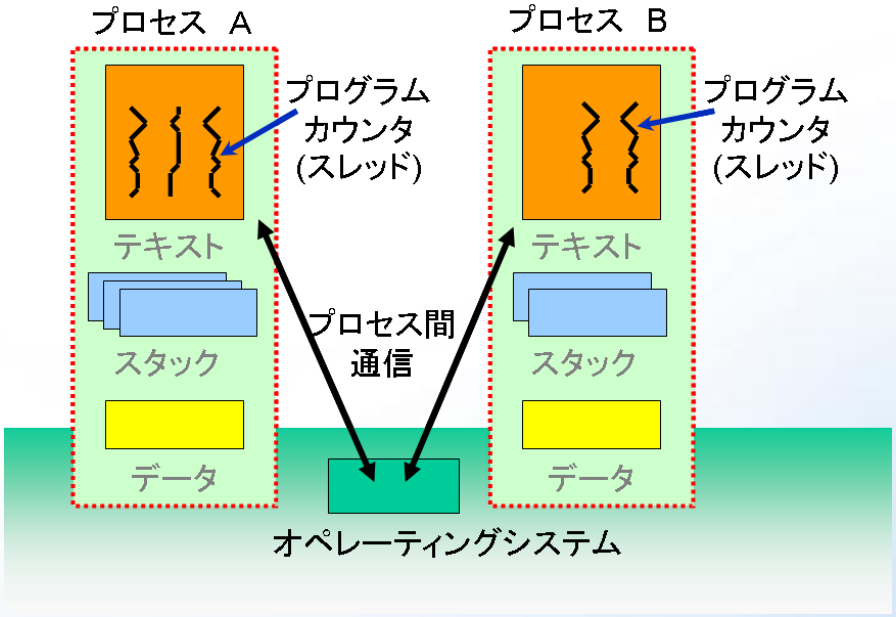
- **プロセスは、独自のリソースを持って個々に実行**
  - 個々に独立したアドレス空間
  - 実行命令、データ、ローカル変数スタック、ステート情報など
  - 個々のレジスタセット
- **スレッドは、一つのプロセス内で実行**
  - アドレス空間を共有
  - レジスタセットも共有
  - 個々のスレッドの独自データ用のスタック領域を持つ



マルチプロセスとマルチスレッド

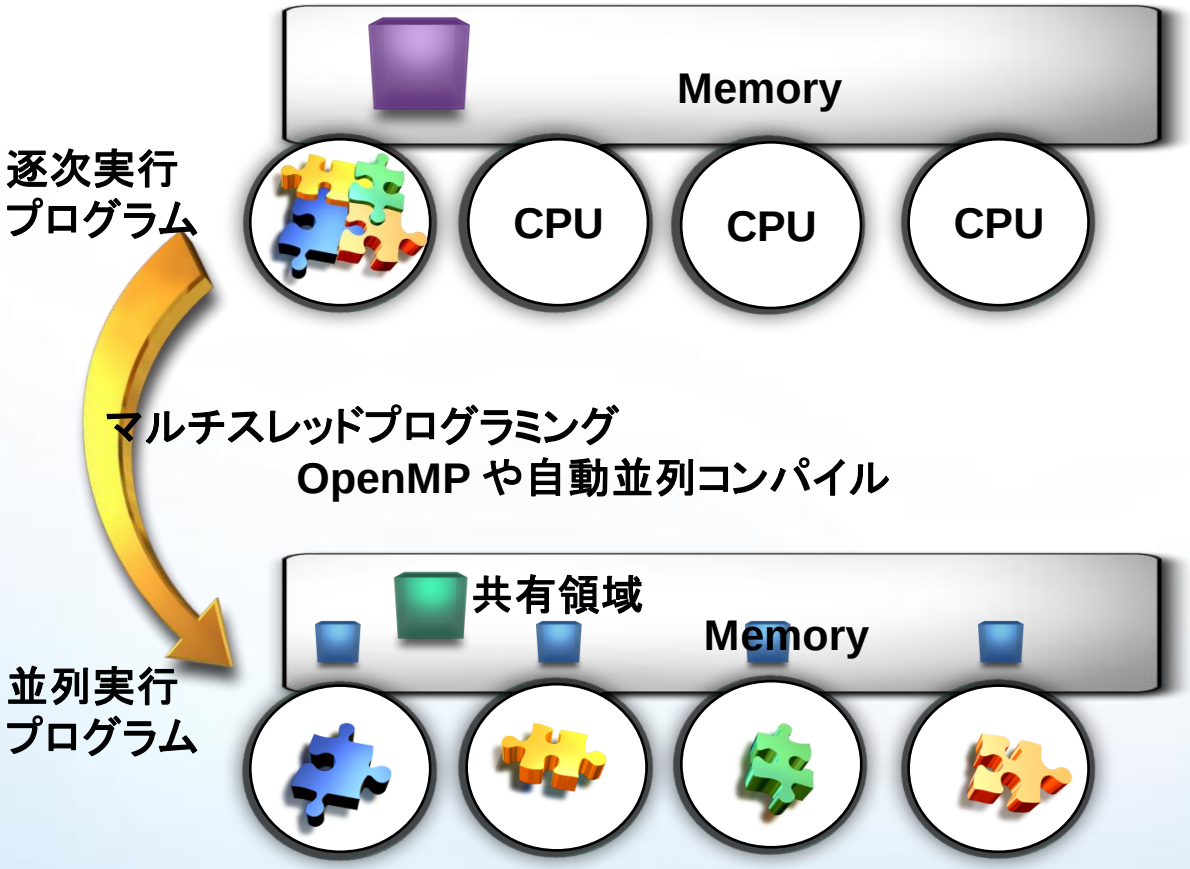


マルチプロセス/シングルスレッド



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)
```

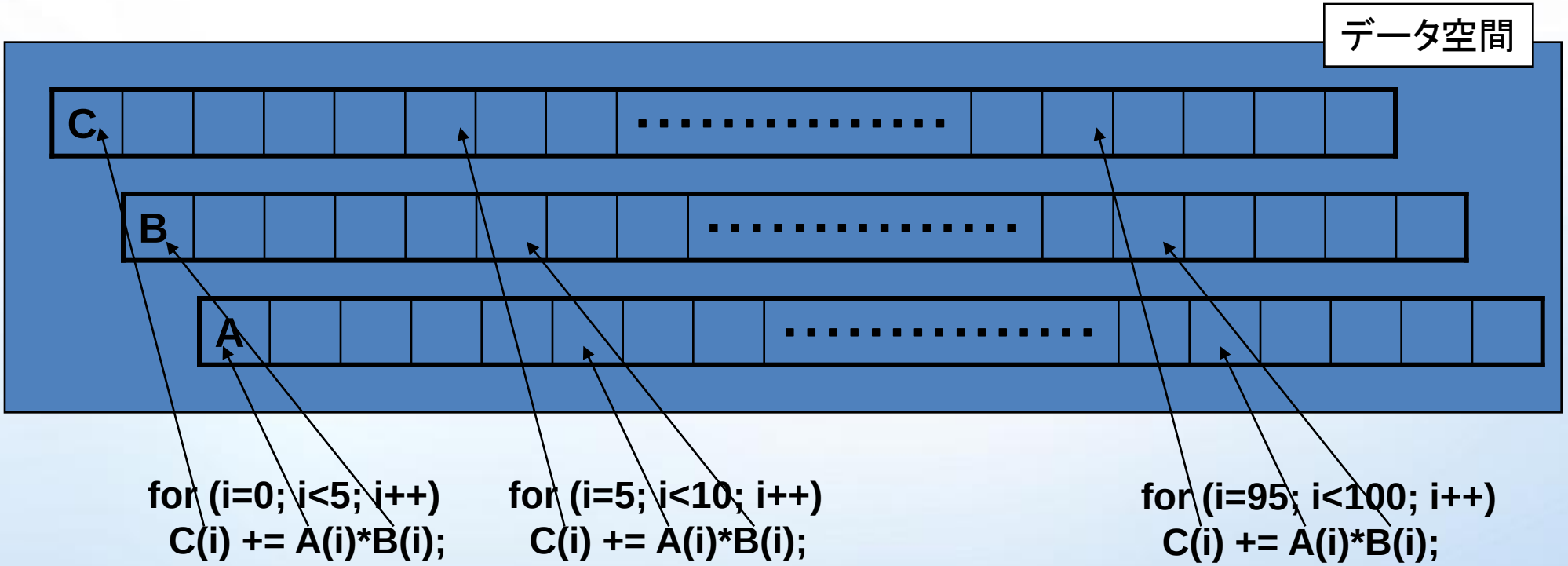
データ並列処理:
独立したループ反復を分割し、並列に実行する

```
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

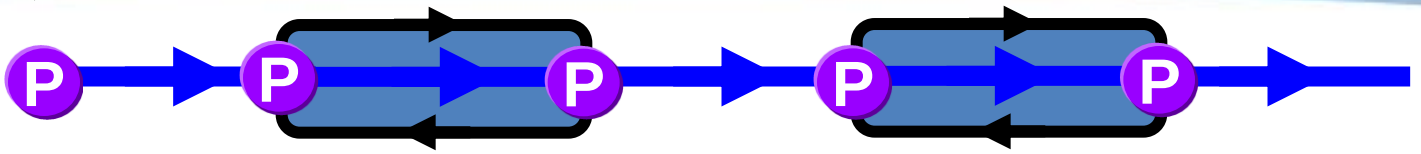
- 計算負荷の大きなループやプログラムのセクションを複数のスレッドで同時に処理
- 複数のスレッドを複数のプロセッサコア上で、効率良く処理する

```
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

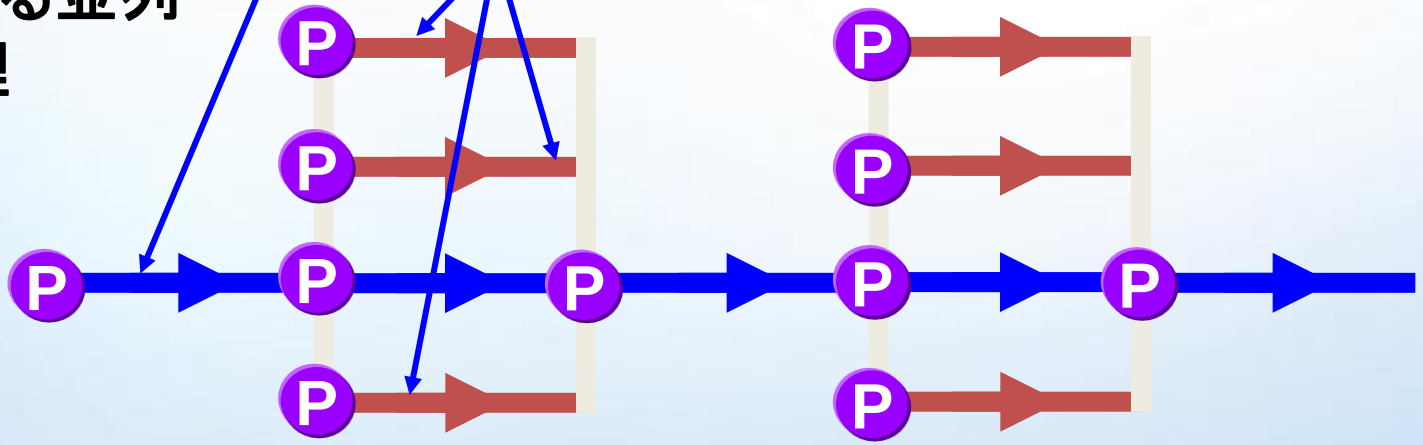


プログラムのループなどの反復計算を複数のスレッドに分割し、並列処理を行う

“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

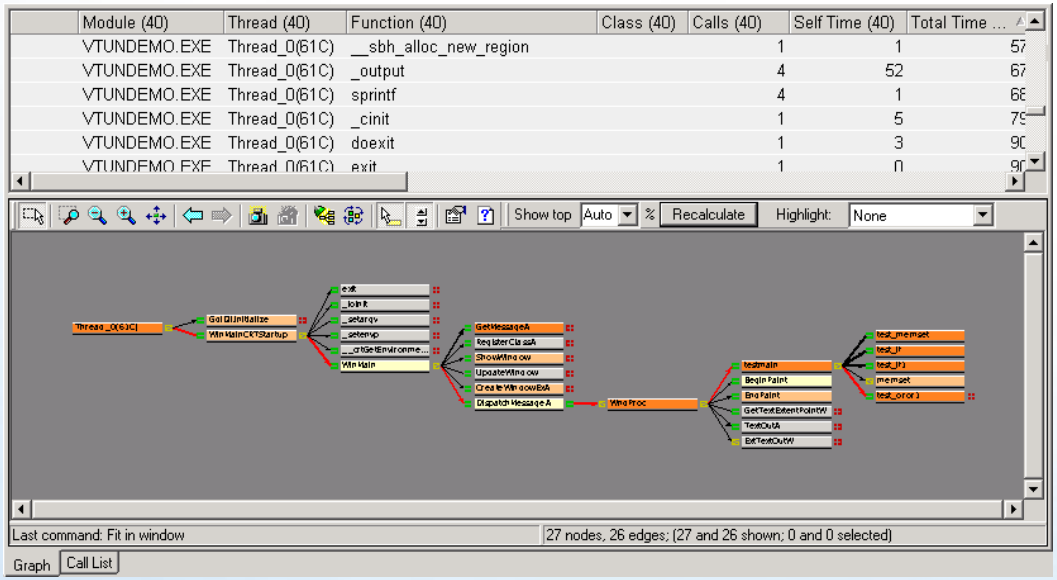
```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
      int i; double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
      for (i=0;i<= num_steps;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

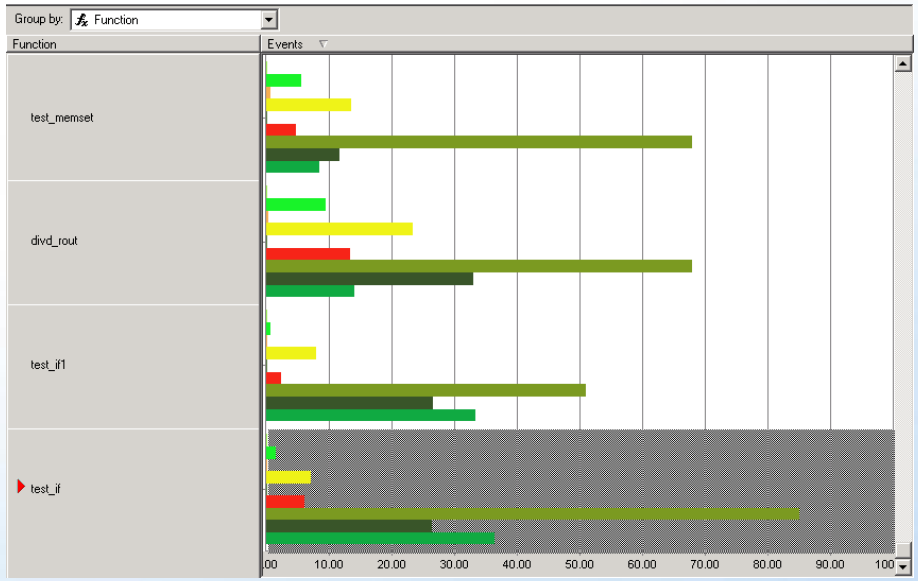
```
#include <omp.h>
static long num_steps = 1000000; double step;
static double sum = 0.0;
#pragma intel omp sharable(sum)
#pragma intel omp sharable(num_steps)
#pragma intel omp sharable(step)
#define NUM_THREADS 4
void main ()
{
        int i; double x, pi;
        step = 1.0/(double) num_steps;
        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

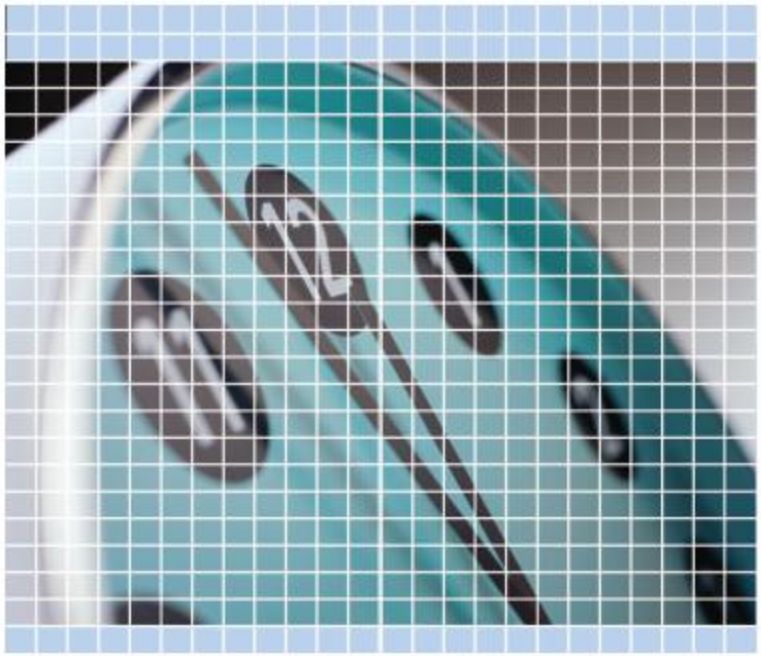


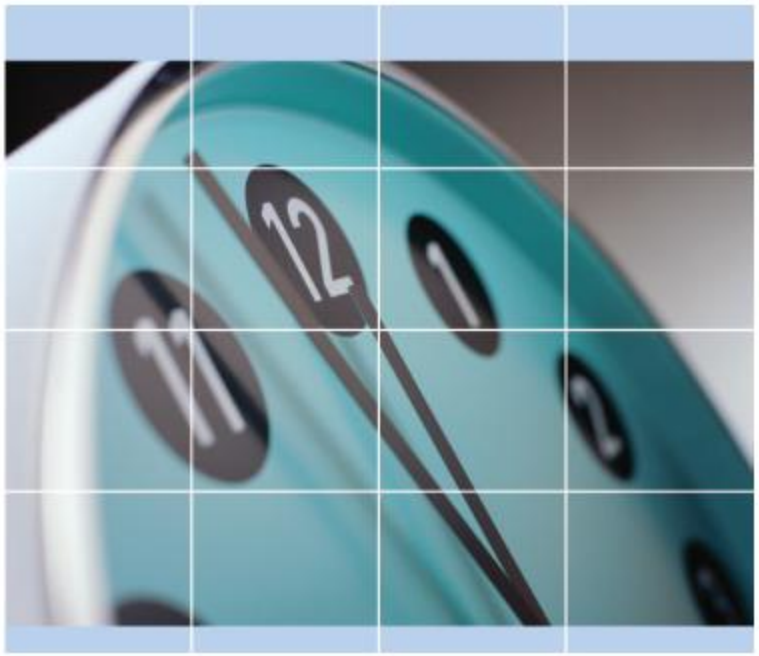
Intel Vtune Hotspot Graph

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



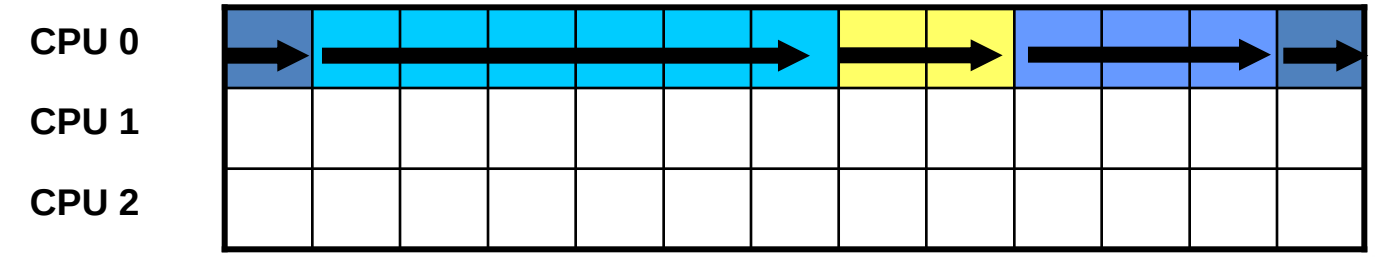


計算粒度を模式的に示した図：領域を例えばスレッド数に分割し、それぞれの領域を個々のスレッドが計算するような場合は疎粒度での領域分割となります。一方、領域を細かく分割し、各スレッドが複数の領域を順次処理するような場合は細粒度での領域分割となります。

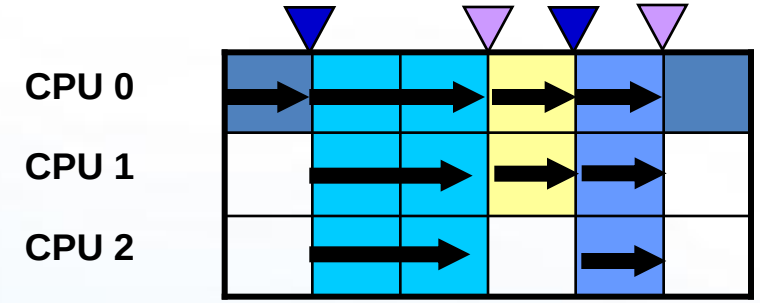
インテル® コンパイラ
OpenMP* 入門
デュアルコア/マルチコア対応アプリケーション開発①より

計算粒度とワークロードの分散

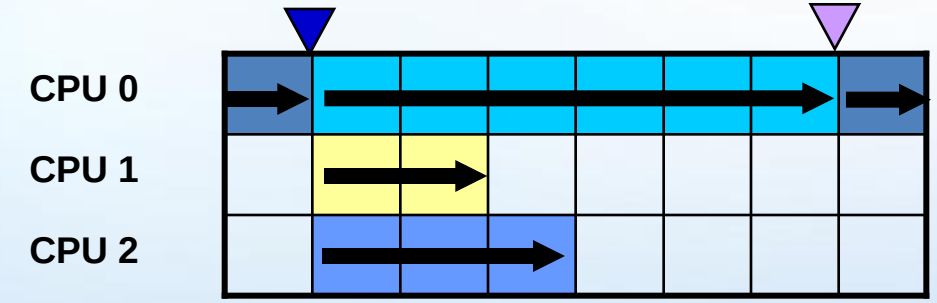
逐次処理



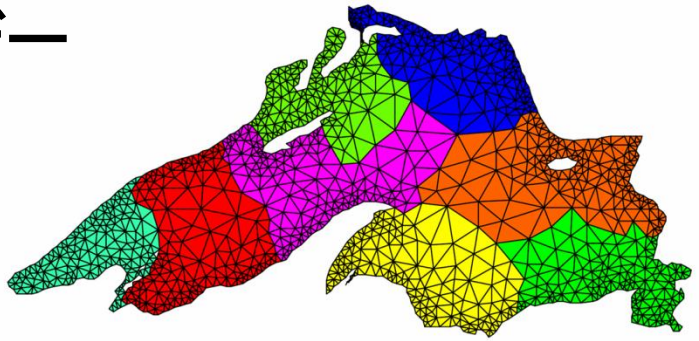
並列処理：細粒度での並列処理



並列処理：疎粒度での並列処理

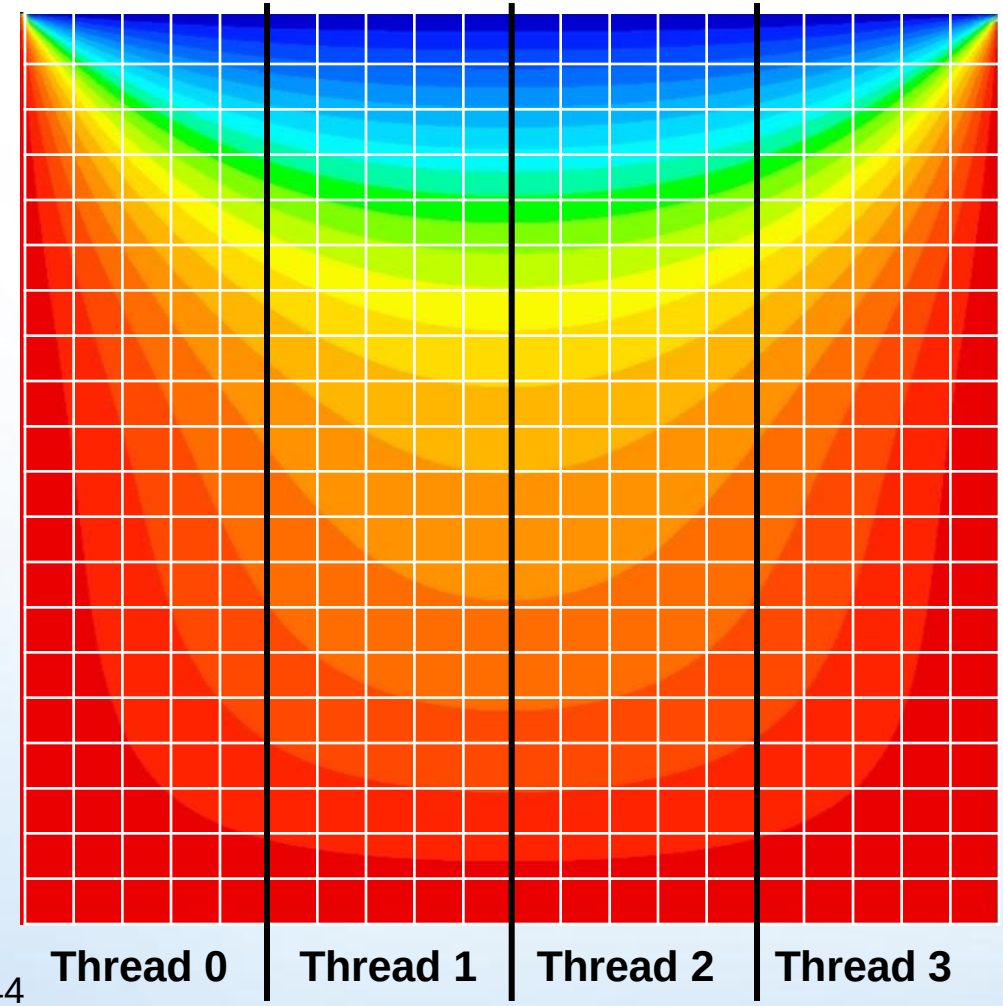


パーティショニング



- **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

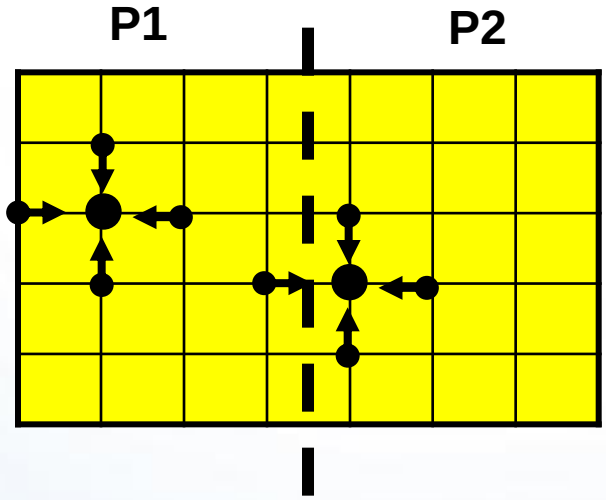
熱伝導方程式 –ポアソン方程式



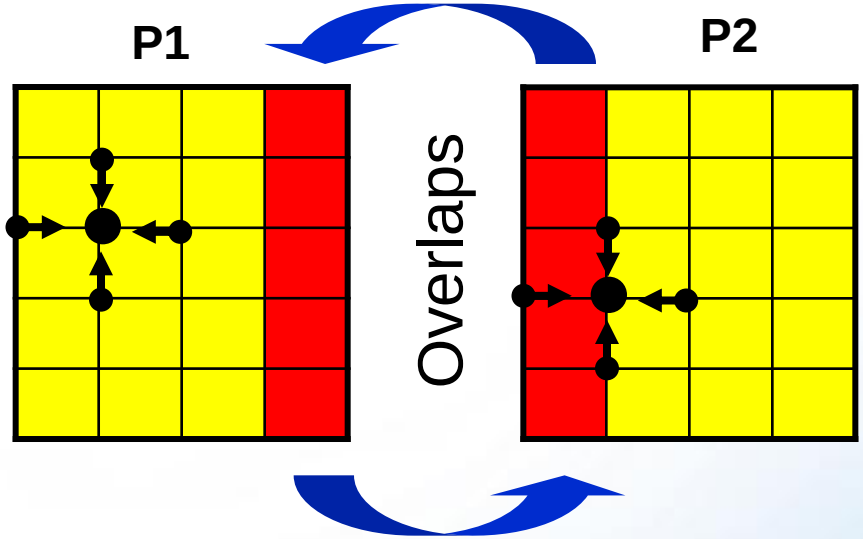

```
float w[*][*], u[*][*];

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



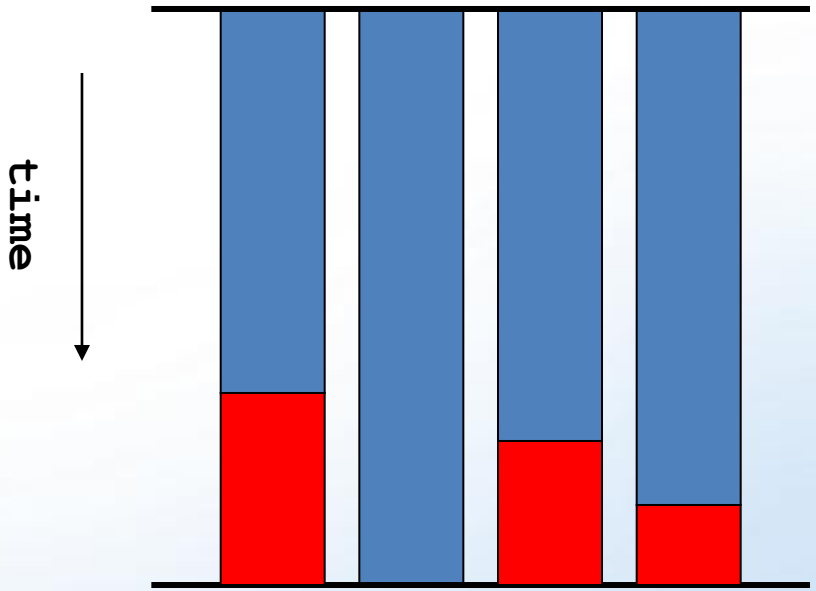
共有メモリでの並列処理



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

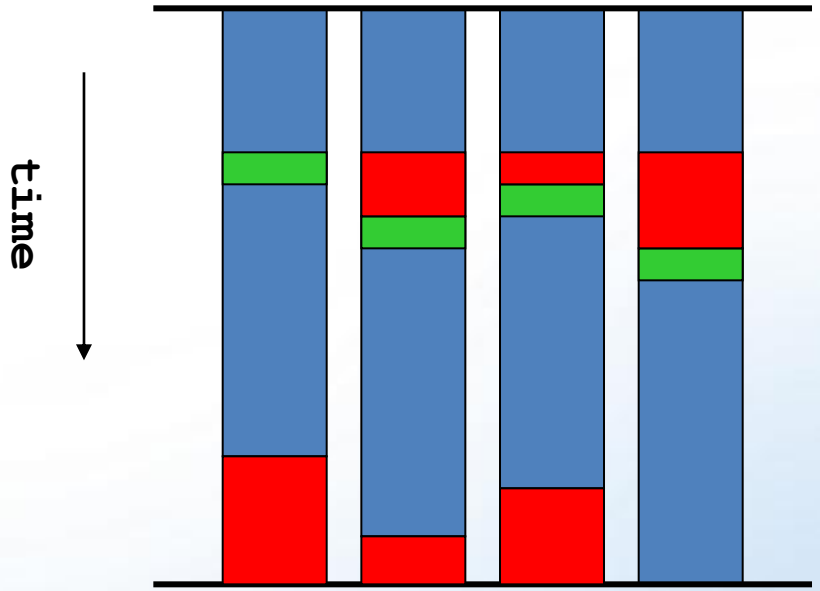
- **共有データを持たず、各プロセスが、独自のメモリ領域を持つ**
- **従って、同期 = 通信となる**
  - MPIにおいては、同期通信を行った場合、データ転送の終了まで、その実行を待つことになる
- **データへのアクセス制御**
  - あるプロセスが、他のノード上のa[i]のデータを必要とした場合、そのデータを転送し、その転送が終了するまで、計算を進めることはできない

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**

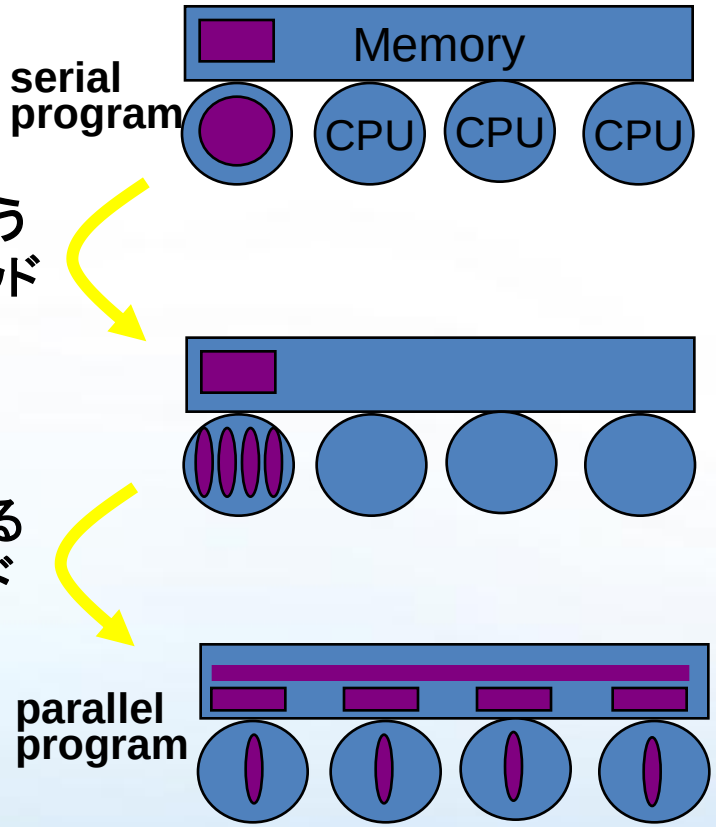


並列化の阻害要因

- **“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
```

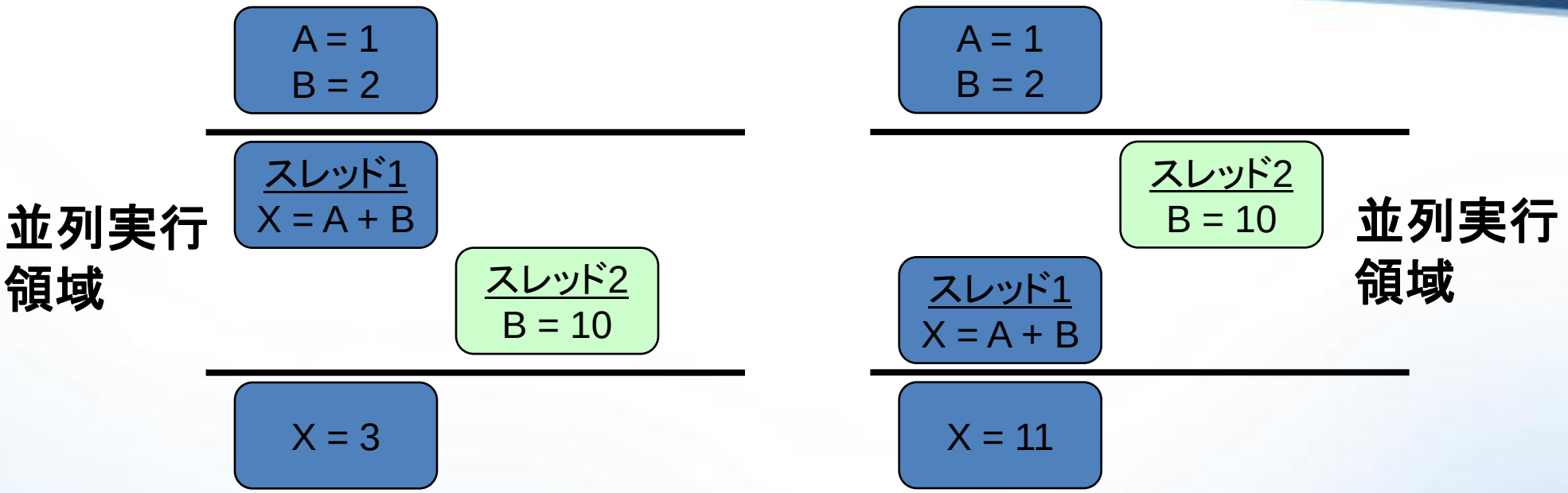
2.

```
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の特徴

- **コンパイラのサポート**
  - コンパイラ･オプションでの適用、非適用の選択が可能（Windows：/Qopenmp スイッチ、Linux:-openmp スイッチ_）
  - スレッドの生成や各スレッドの同期コントロールといった制御を気にする必要がない
  - OpenMP_での並列化を適用していて、計算などが不正になった場合、簡単にその部分だけを逐次実行に切り替えることも可能（プログラムのデバッグが容易）
- **明示的な並列化の指示**
  - コンパイラに対して、並列化のためのヒントを与えるのでなく、明示的に並列化を指示→間違った指示行を指定しても、コンパイラはその指示に従って、並列化を行う

マルチスレッドプログラミングの基本

- 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

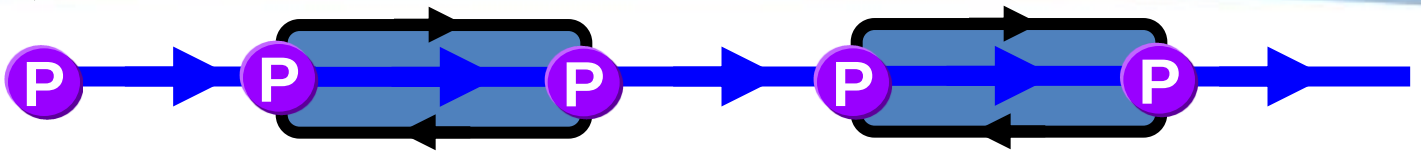
```
void main()
{
    double Res[1000];
// 計算負荷の大きな計算ループに対して、
// マルチスレッドでの並列処理を適用します
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

OpenMP の適用

```
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

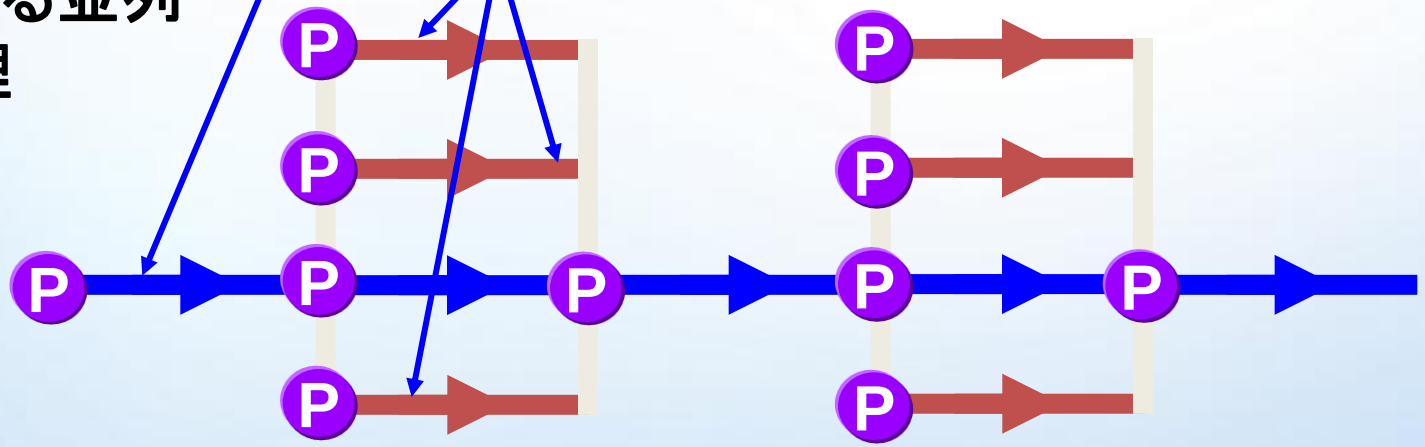


プログラムのループなどの反復計算を複数のスレッドに分割し、並列処理を行う

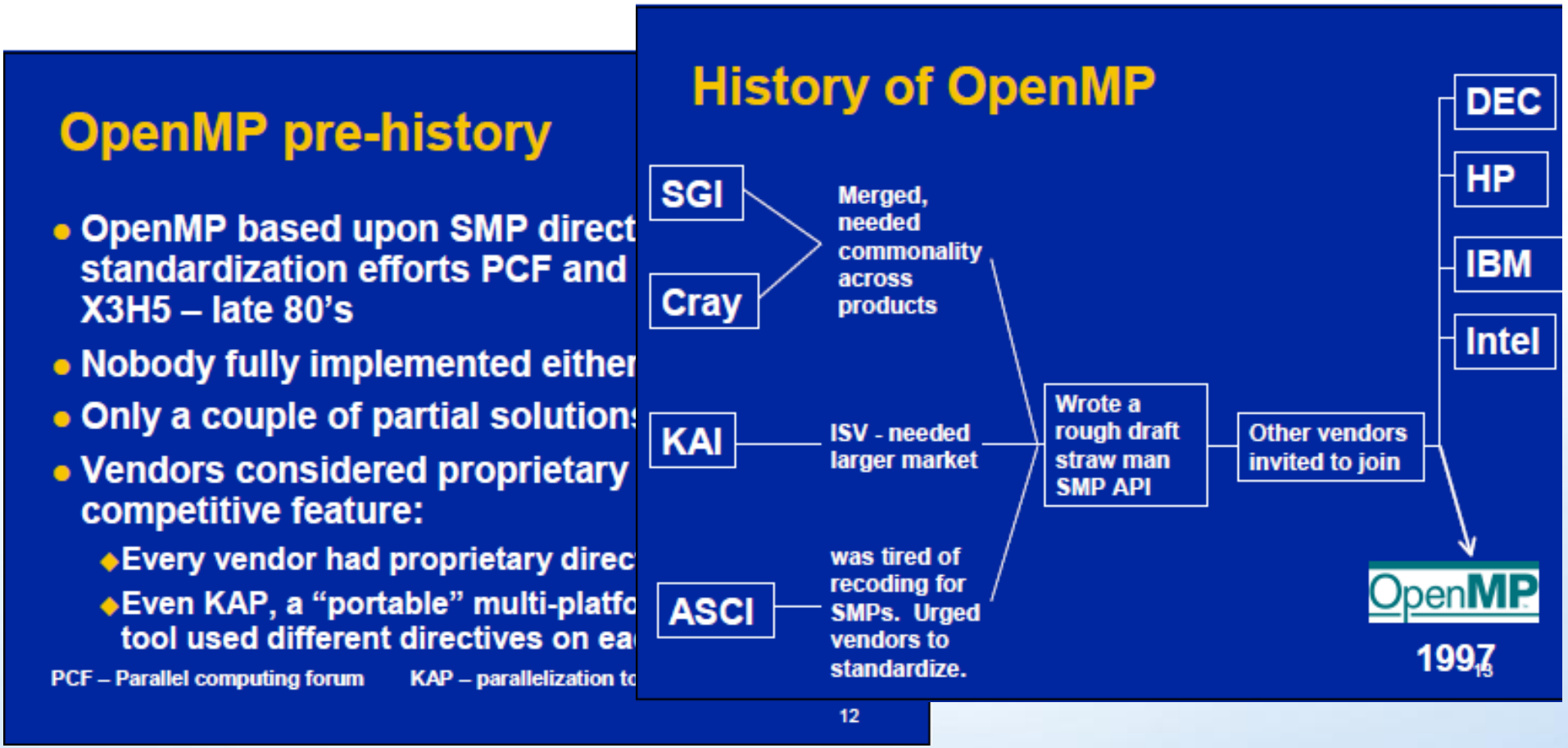
“マスタースレッド”

“ワーカースレッド”

マルチスレッドによる並列処理

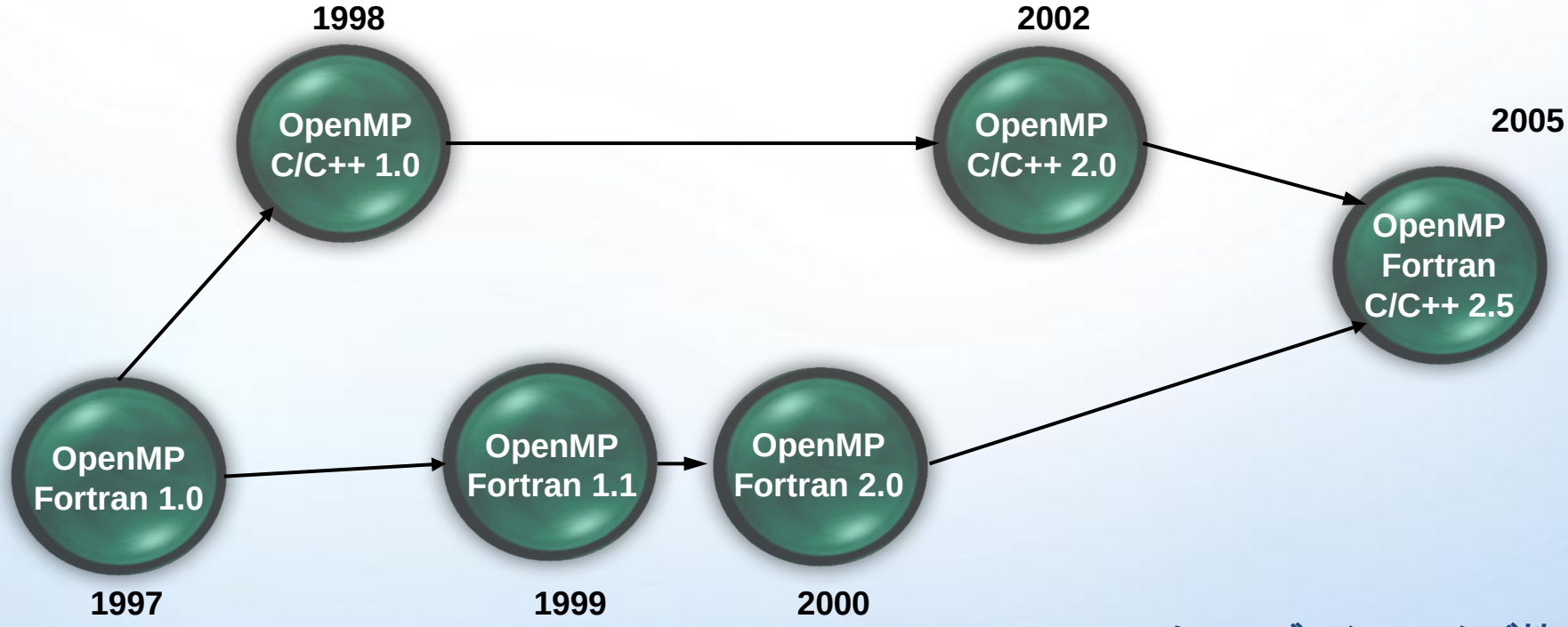


OpenMP以前

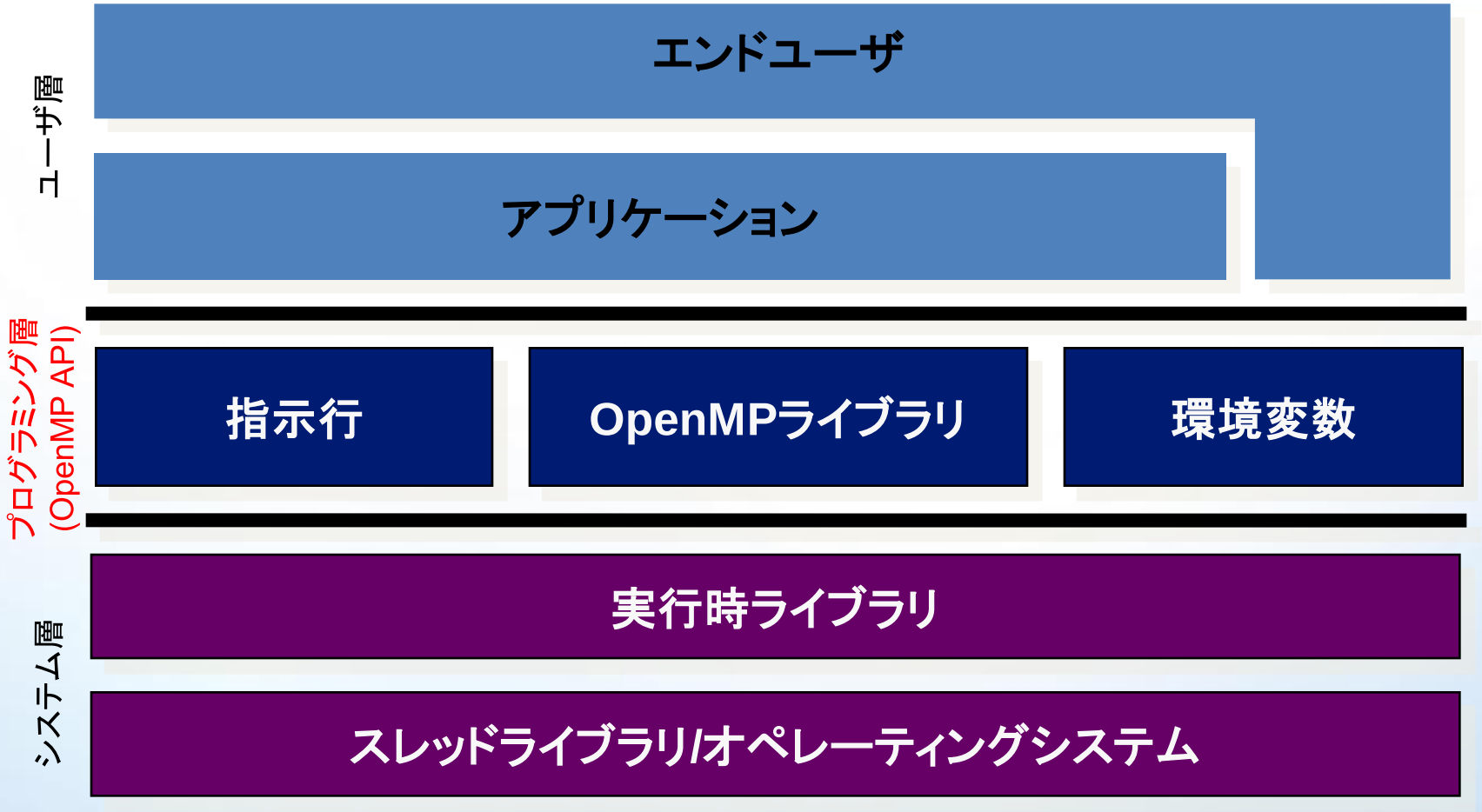

OpenMP pre-history
• OpenMP based upon SMP direct
standardization efforts PCF and
X3H5 – late 80's
• Nobody fully implemented either
• Only a couple of partial solution
• Vendors considered proprietary
competitive feature:
◆Every vendor had proprietary direc
◆Even KAP, a "portable" multi-platfo
tool used different directives on ea
PCF – Parallel computing forum
KAP – parallelization to
12
History of OpenMP
SGI
Cray
Merged, needed commonality across products
KAI
ISV - needed larger market
ASCI
was tired of recoding for SMPs. Urged vendors to standardize.
Wrote a rough draft straw man SMP API
Other vendors invited to join
DEC
HP
IBM
Intel
OpenMP
1997
13

OpenMP APIのリリースの歴史

- **OpenMPの詳細な仕様などは、OpenMPのホームページ www.openmp.org で入手することが可能です。最新のOpenMPのリリースは、2005年5月に発表された、OpenMP 2.5であり、この仕様でC/C++とFortranの規格が統合されました。**



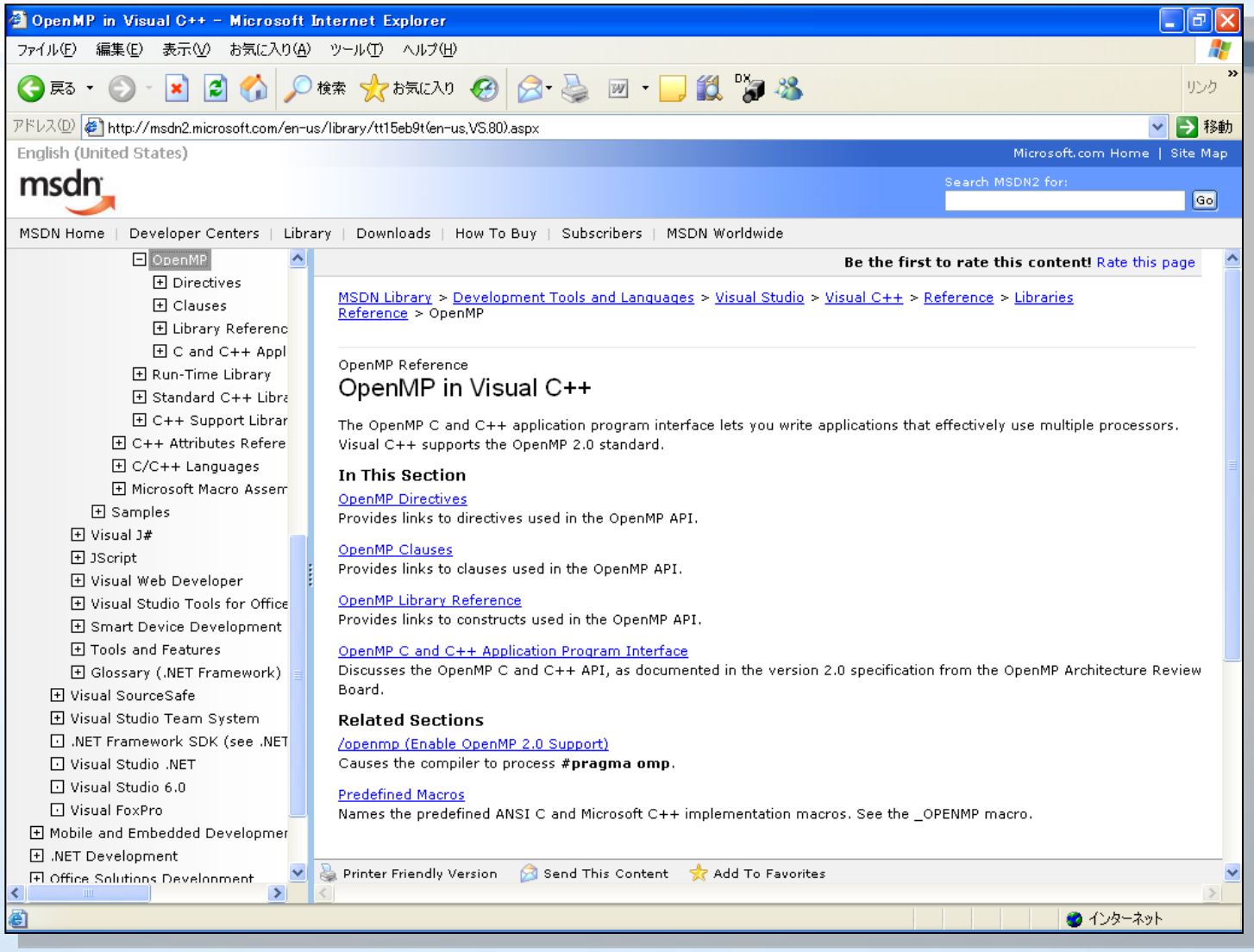
OpenMP APIの構造



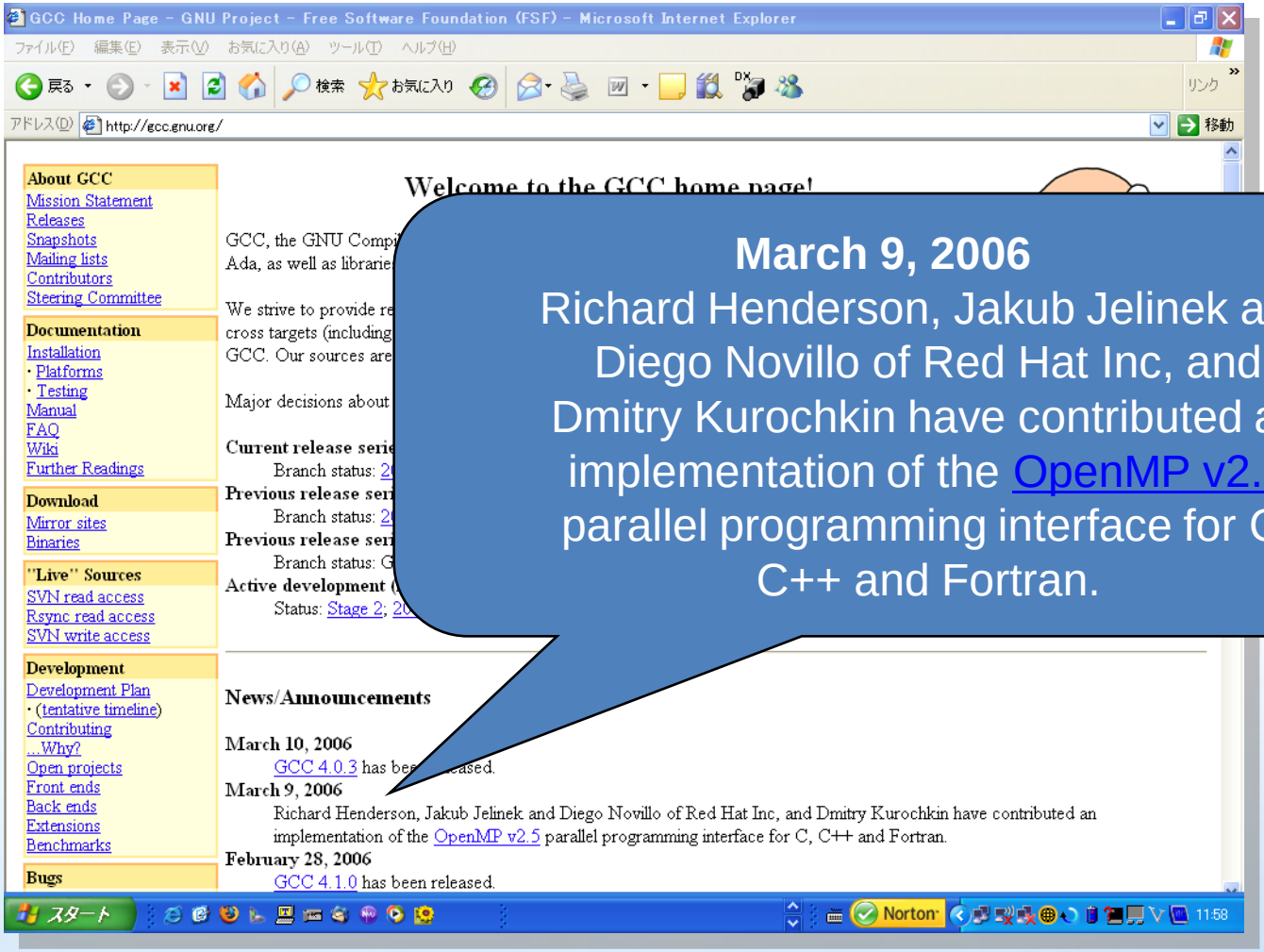
OpenMPの特徴

- **プログラムの段階的な並列化が可能**
  - コードの設計時からOpenMPを利用した並列処理を実装することも可能
  - 既に開発されたプログラムを、OpenMPを利用して、段階的に並列化することも可能
- **自動並列化との併用**
  - 自動並列化とOpenMPを併用することも可能であり、プログラムの一部だけをOpenMPで並列化し、他の部分を自動並列化することも可能
- **WindowsでもLinuxでも同じAPIが利用可能**
  - ソースの互換性
- **疎粒度での並列化の適用**
  - 自動では並列化が難しい関数やサブルーチンの呼び出しを含むタスクでの並列化（疎粒度での並列化）も可能
  - 粒度の大きな並列化では、よりオーバーヘッドの小さな並列化処理が可能

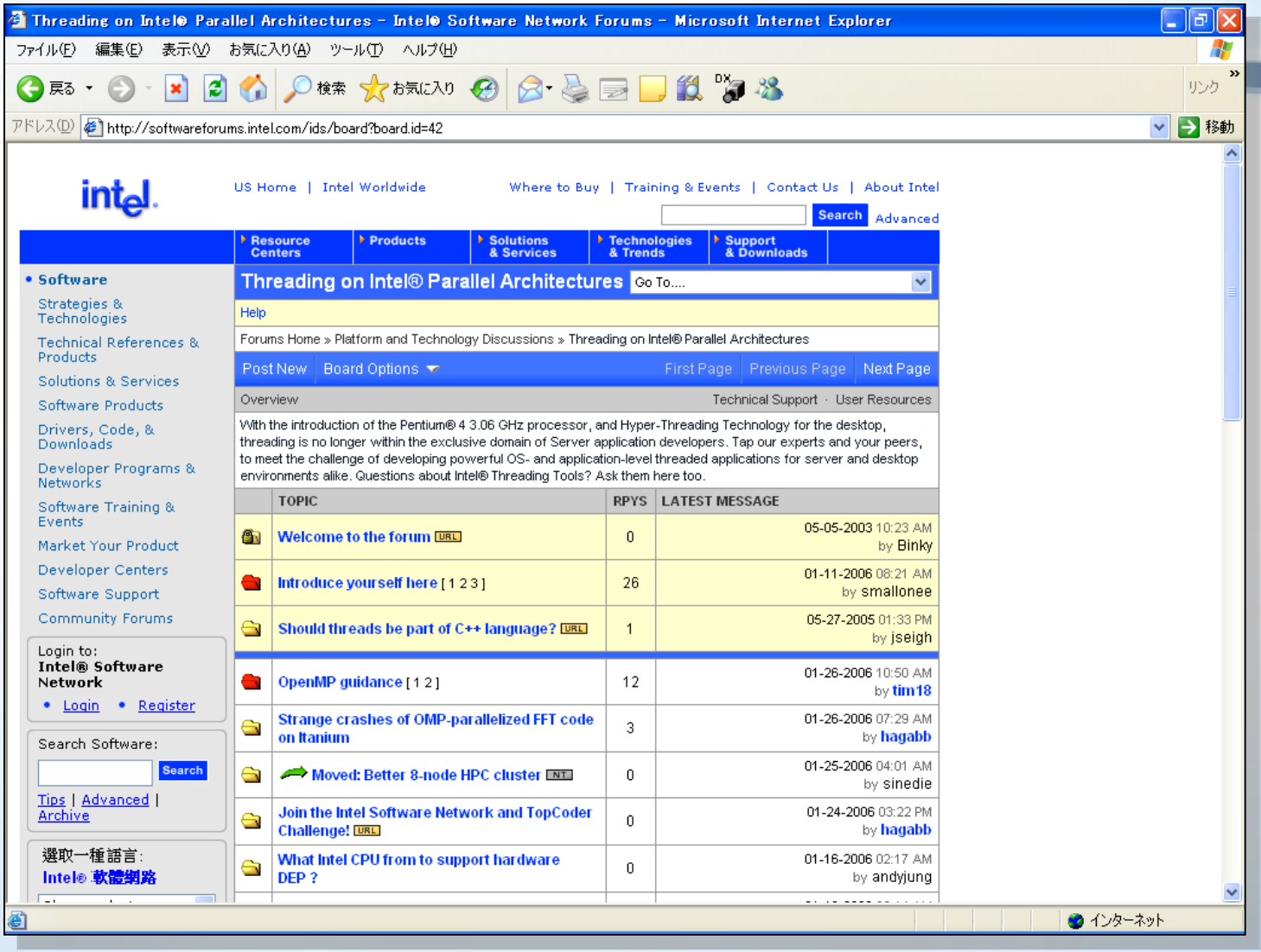
OpenMP in Visual C++



GNU Compiler Collection



Intel® Software Network Forums



OpenMPスケーラビリティ

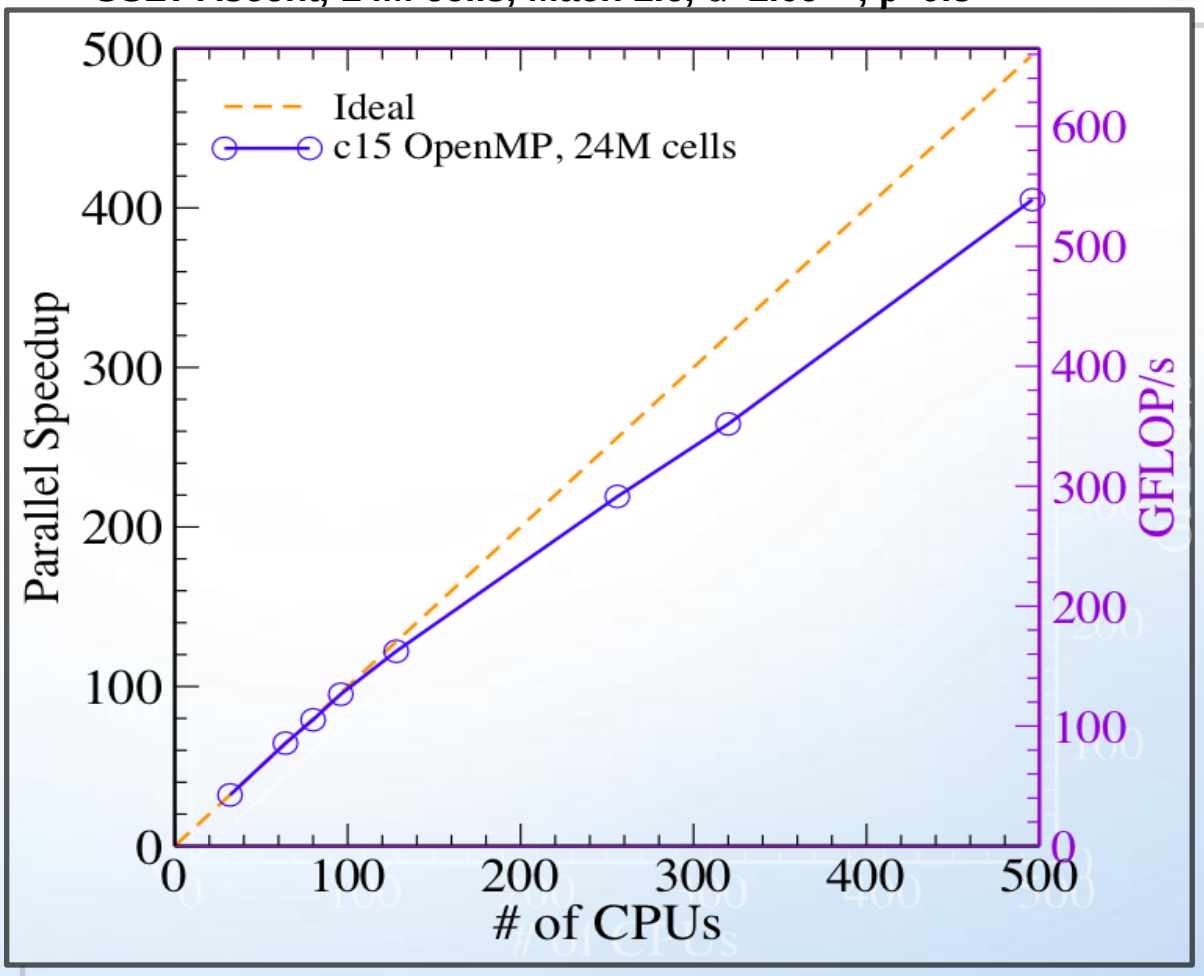
NASA/CART3D

540 GFLOP/s

CPUあたりの性能：1.33 GFLOP/s

並列性能：496プロセッサで405倍

SSLV Ascent, 24M cells, Mach 2.6, α=2.09°, β=0.8°



MPIとOpenMPのAPIとしての比較

	MPI （メッセージパッシング）	OpenMP
利点	• 分散メモリシステムと共有メモリシステムの双方で利用可能 • ノードサイズを超えての並列処理が可能 • データ配置の制御が容易	• 並列化が容易 • 低いレイテンシと高いバンド幅 • 通信制御が不要 • 粒度に依存しない並列化が可能（細粒度と疎粒度の双方が可能） • 動的な負荷分散（ロードバランス）が可能
問題点	• プログラム開発が容易でなく、また、デバッグが困難 • 高いレイテンシと低いバンド幅 • 疎粒度でのプログラミングが必要（ループレベルでの並列化は難しい） • 負荷分散（ロードバランス）が難しい	• 共有メモリシステムだけ • ノードサイズがスケーラビリティの限界 • データ配置が問題になる可能性がある • スレッドの細かな制御が困難

MPIとOpenMPのAPIとしての比較

	MPI （メッセージパッシング）	OpenMP
並列化	• 疎粒度での並列化 • 一般には、SPMD型のプログラミング • データ並列でもタスク並列でも利用可能	• 疎粒度での並列化も可能 • 一般には、ループレベルでの並列化を行うが、SPMD型のプログラミングも可能 • データ並列でもタスク並列でも利用可能 • OpenMPの基本は、スレッドのワークシェアであるが、個々のスレッドへのデータのアサインも可能
	• 複数のプロセスから構成される • ‘Shared Nothing’ プロセス • 陽的なメッセージ交換 • 同期処理はメッセージ交換時に実行される	• 複数スレッドから構成される • スレッドスタック以外は、全て共有される • 陽的な同期処理が必要 • 共有データへのアクセス

データの共有と保護

	メッセージ・パッシング	スレッド
データの共有	• メッセージを送受信 • ブロードキャスト • スキャッタ、ギャザ	• 共有メモリ領域に値を格納
データの保護	• 別のプロセスからメモリを読み取ることができない	• スレッドローカル格納領域 • スレッドスタックと関数からのスコープ • OpenMP* スコープ • ミューテックス（Mutex）
データの競合		• 複数のスレッドが共有データにアクセス • 実行順は仮定されているが保証されていない • 診断が困難

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

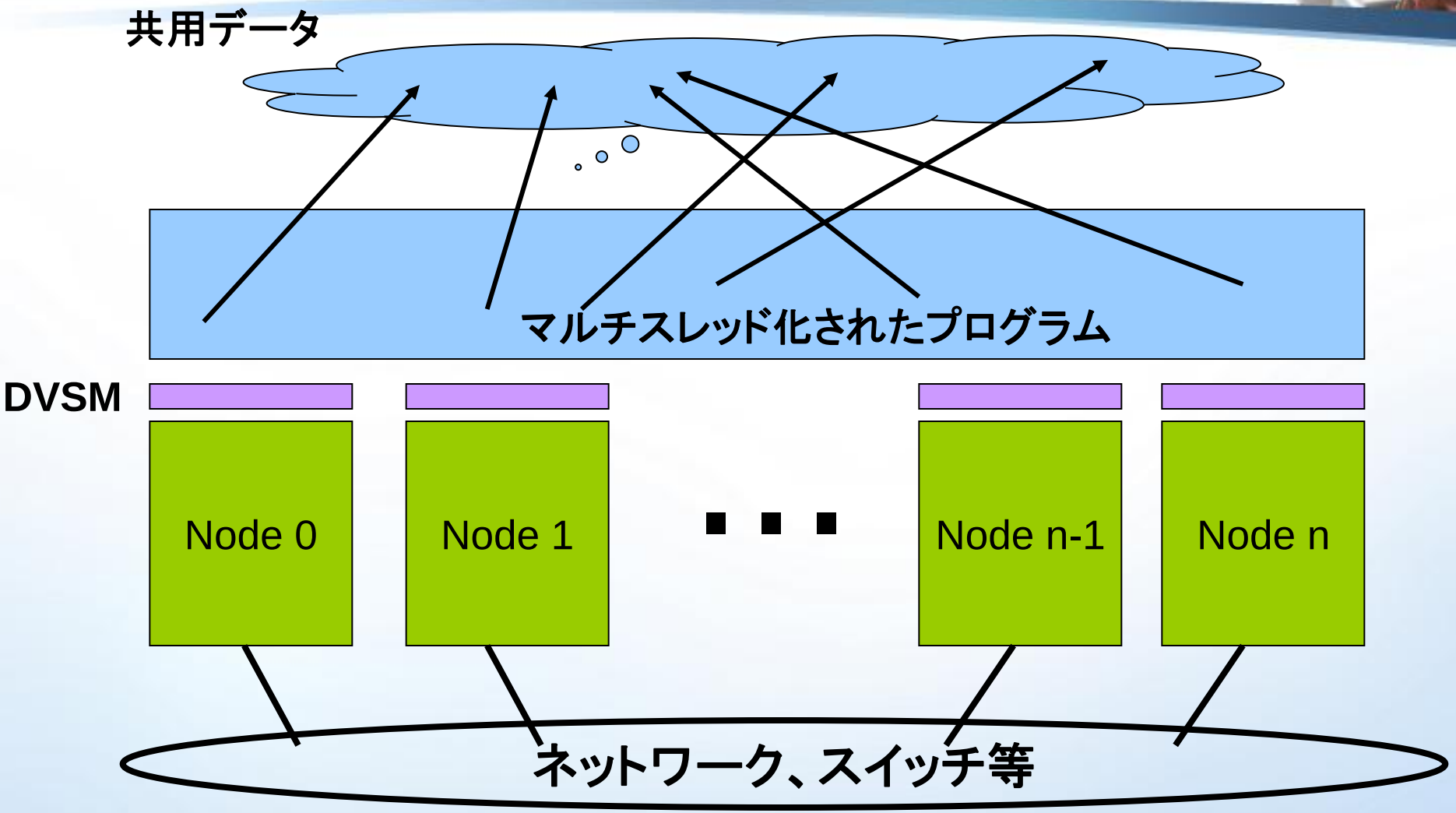
コンパイルとメッセージ

環境変数の設定

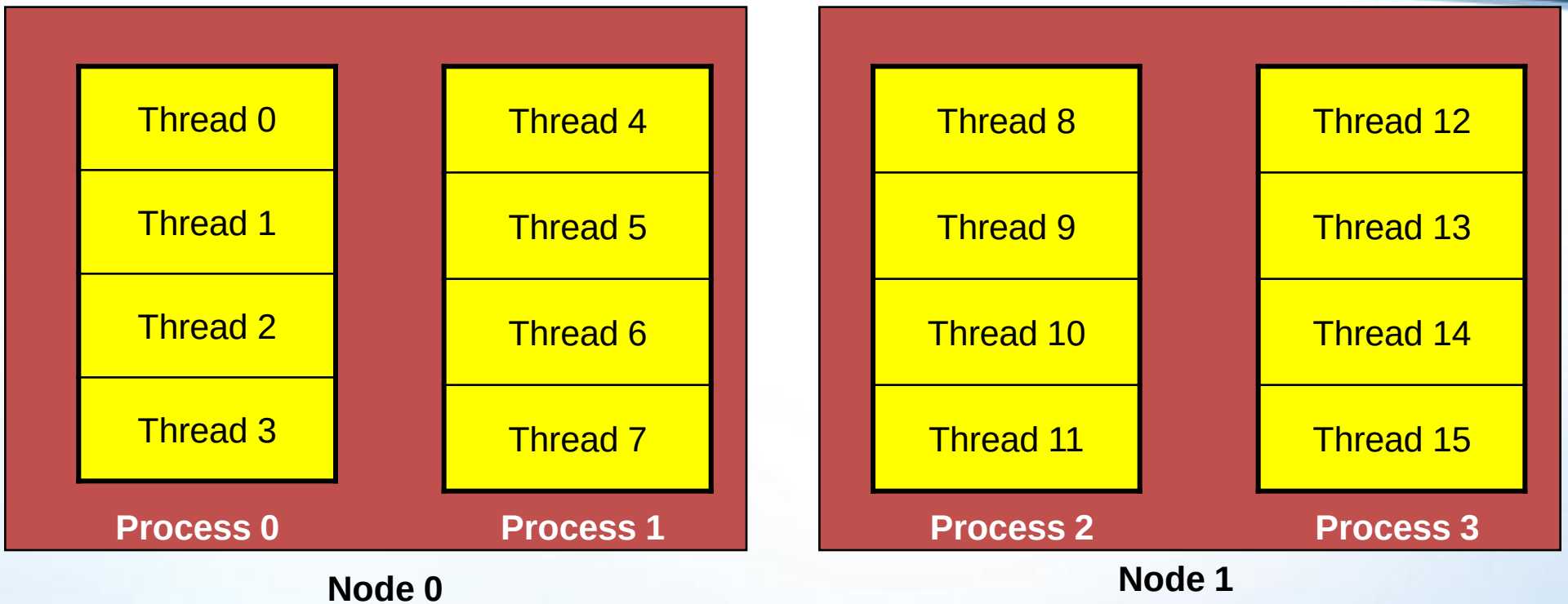
クラスタOpenMP

- **特徴**
  - OpenMPプログラミングモデルをクラスタ環境に拡張
  - Run-Time ライブラリによるクラスタ上でのOpenMPプログラミングのサポート
- **並列化API**
  - OpenMPのメモリ階層モデルを拡張
  - OpenMP APIを利用し、各ノードへのOpenMPプログラムの分散を支援
- **並列化効率**
  - MPIやOpenMPほどのプログラムの汎用性には欠ける
  - 利用可能なプログラムは限定的
  - 実行性能は、プログラムの実装に強く依存する

分散仮想共有メモリ(DVSM)



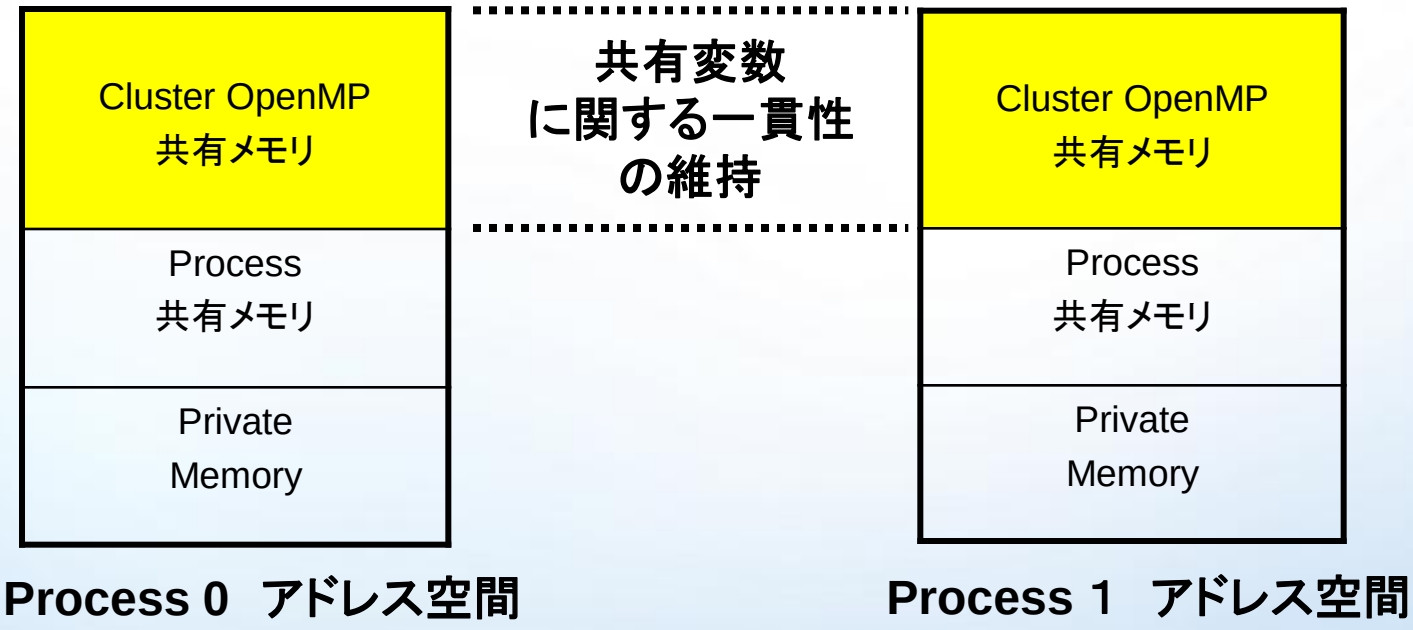
Cluster OpenMP スレッドとプロセス



Node – クラスタを構成する各計算機システム
プロセス – Linuxのプロセス
スレッド – OpenMPのスレッド（プロセス中のスレッド）

Cluster OpenMPメモリモデル

- **プロセス間でOpenMPスレッドがアクセスする変数は、'sharable' 変数として指示**
  - 通常のOpenMPの共有データの宣言では、プロセス間でのデータ共有は出来ない（プロセス内でのデータの共有を宣言）



’sharable’ 変数の宣言

- **OpenMPの指示句で、’shared’と指示される変数や共有される変数は、この ‘sharable’の宣言が必要（ただし、ファイルポインタなどのシステムが使用する変数は除く）**
- **‘sharable’の宣言が必要な変数に関する情報は、コンパイラ時のメッセージとして、確認も可能**

  例）–clomp-sharable-propagation オプションの指定

  ifort –cluster-openmp –clomp-sharable-propagation –ipo file.f file2.f

  fortcom: Warning: Sharable directive should be inserted by user as

  ‘!dir$ omp sharable(n)’ in file file.f, line 23, column 16
- **データの‘sharable’ 宣言は、コンパイラ指示行で適用**

  #pragma intel omp sharable(var)　　// C,C++

  !dir$ omp sharable(var)　　! Fortran

コンパイラによるデータ共有に関する解析

プログラム中のデータに関して、’sharable’ 指定が必要なデータに関する情報を出力し、コンパイラ指示行の内容に関してもその情報を提供

```
% ifort -cluster-openmp -clomp-sharable-propagation -ipo jacobi.f
fortcom: Warning: Global variable '/idat/' not made sharable since it is not allocated space in any compilation unit
in file jacobi.f, line 86, column 26

fortcom: Warning: Sharable directive should be inserted by user as '!dir$ omp sharable(dx)'
in file jacobi.f, line 72, column 39

fortcom: Warning: Sharable directive should be inserted by user as '!dir$ omp sharable(dy)'
in file jacobi.f, line 72, column 42

fortcom: Warning: Argument #6 must be declared as sharable
in file jacobi.f, line 86, column 26

fortcom: Warning: Global variable '/fdat/' not made sharable since it is not allocated space in any compilation unit
in file jacobi.f, line 81, column 20

fortcom: Warning: Argument #7 must be declared as sharable
in file jacobi.f, line 81, column 20

fortcom: Warning: Argument #8 must be declared as sharable
in file jacobi.f, line 81, column 20

jacobi.f(177) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
jacobi.f(186) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
jacobi.f(175) : (col. 6) remark: OpenMP DEFINED REGION WAS PARALLELIZED.
jacobi.f(112) : (col. 6) remark: OpenMP DEFINED LOOP WAS PARALLELIZED.
```

’sharable’ 変数の宣言

オリジナルコード	コンパイラオプション	コンパイラオプションと同等の ’sharable’ 変数の宣言
common /blk/ a(100)	-clomp-sharable-commons	**common /blk/ a(100)** **!dir$ omp sharable (/blk/)**
real a(100) **save a**	-clomp-sharable-localsaves	**real a(100)** **save a** **!dir$ omp sharable (a)**
module m **real a(100)**	-clomp-sharable-modvars	**module m** **real a(100)** **!dir$ omp sharable (a)**

簡単なクラスタOpenMPプログラム例

```
#include <omp.h>

static int x;
#pragma intel omp sharable(x)

int main()
{
        x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x++;
    }
printf("%d should equal %d¥n", omp_get_max_threads(), x);
}
```

sharableディレクティブでコンパイラに変数xはDVSM上に置かなければならないことを指示する

クラスタOpenMPプログラム例

```
      common/storage/ x(2*nk), q(0:nq-1), qq(0:nq-1)
      data              dum /1.d0, 1.d0, 1.d0/
!dir$ omp sharable(/storage/)
!dir$ omp sharable(sx,sy)

!$omp parallel default(shared)
!$omp& private(k,kk,t1,t2,t3,t4,i,ik,x,x1,x2,l,qq)
      do 115 i = 0, nq - 1
         qq(i) = 0.d0
 115  continue

!$omp do reduction(+:sx,sy)
      do 150 k = 1, np
         kk = k_offset + k
.....
         do 140 i = 1, nk
    .....
                sx    = sx + t3
                sy    = sy + t4
             endif
 140     continue
 150  continue
!$omp end do nowait
```

sharableディレクティブでコンパイラに変数xはDVSM上に置かなければならないことを指示する

プログラムのコンパイルと実行

コンパイル時に、Cluster OpenMPでの並列処理を指定

```
$ icc –cluster-openmp test.c
$ cat kmp_cluster.ini
--hostlist=rufus,dufus --processes=2 --process_threads=4
$ a.out
8 should equal 8
```

設定ファイル kmp_cluster.ini に利用する2つのノードを指定し、各ノード上で利用するプロセス数と各プロセスあたりのスレッド数を指定する。この場合には、合わせて、8スレッドでの並列処理となる。

Cluster OpenMP プログラムの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

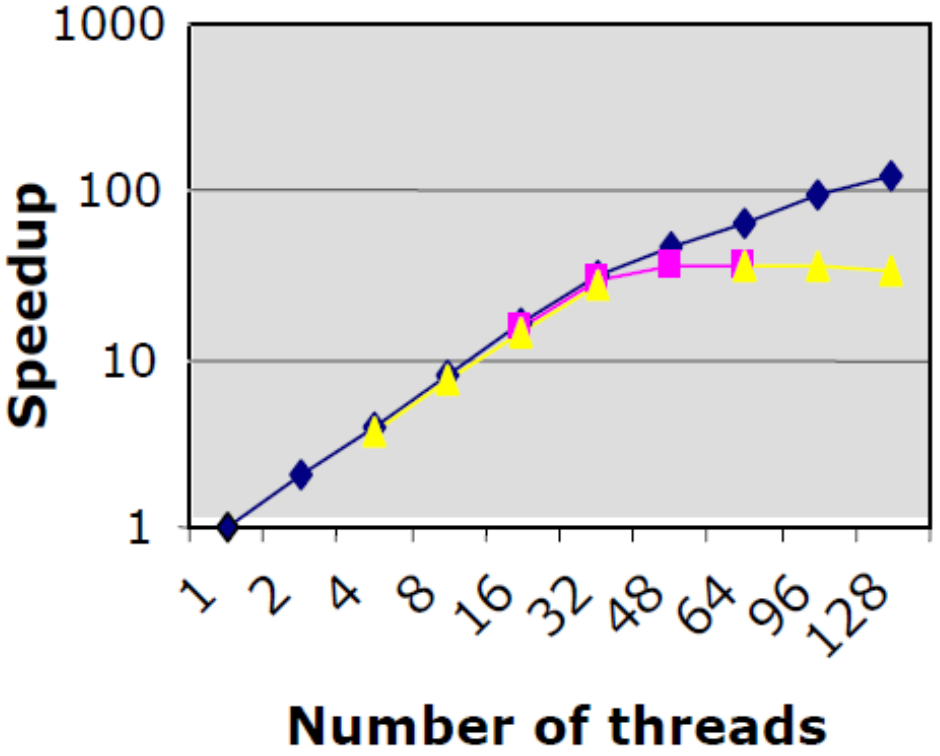
コンパイルとメッセージ

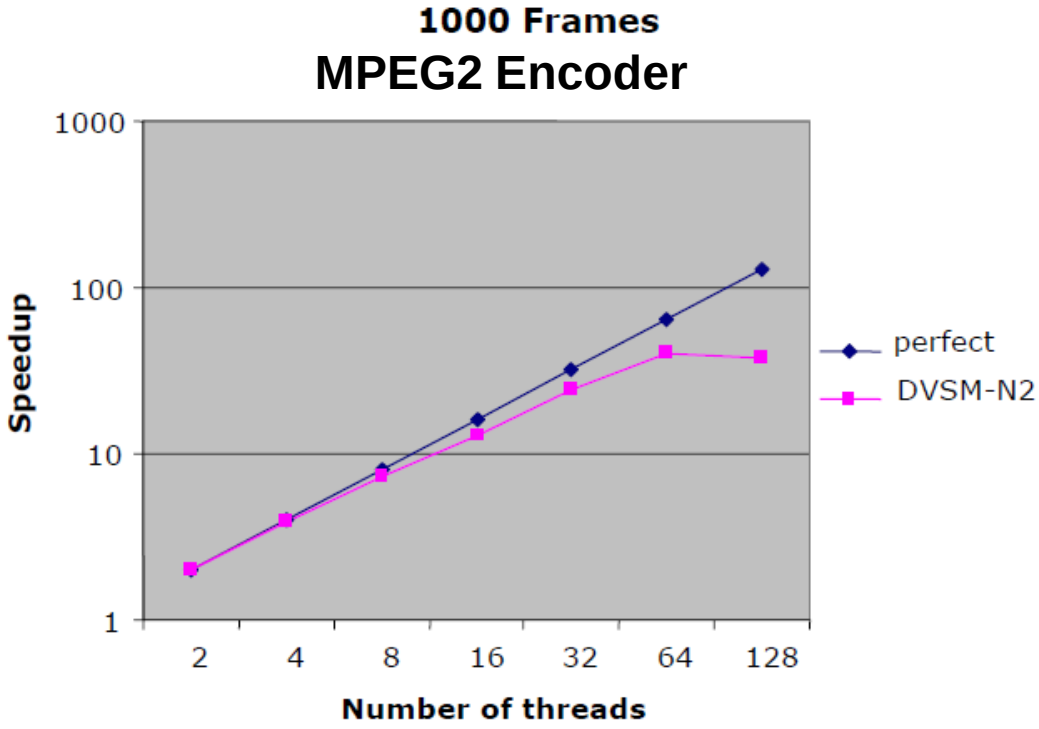
並列実行処理環境の設定

```
$ 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
```

性能(ベンチマーク)データ

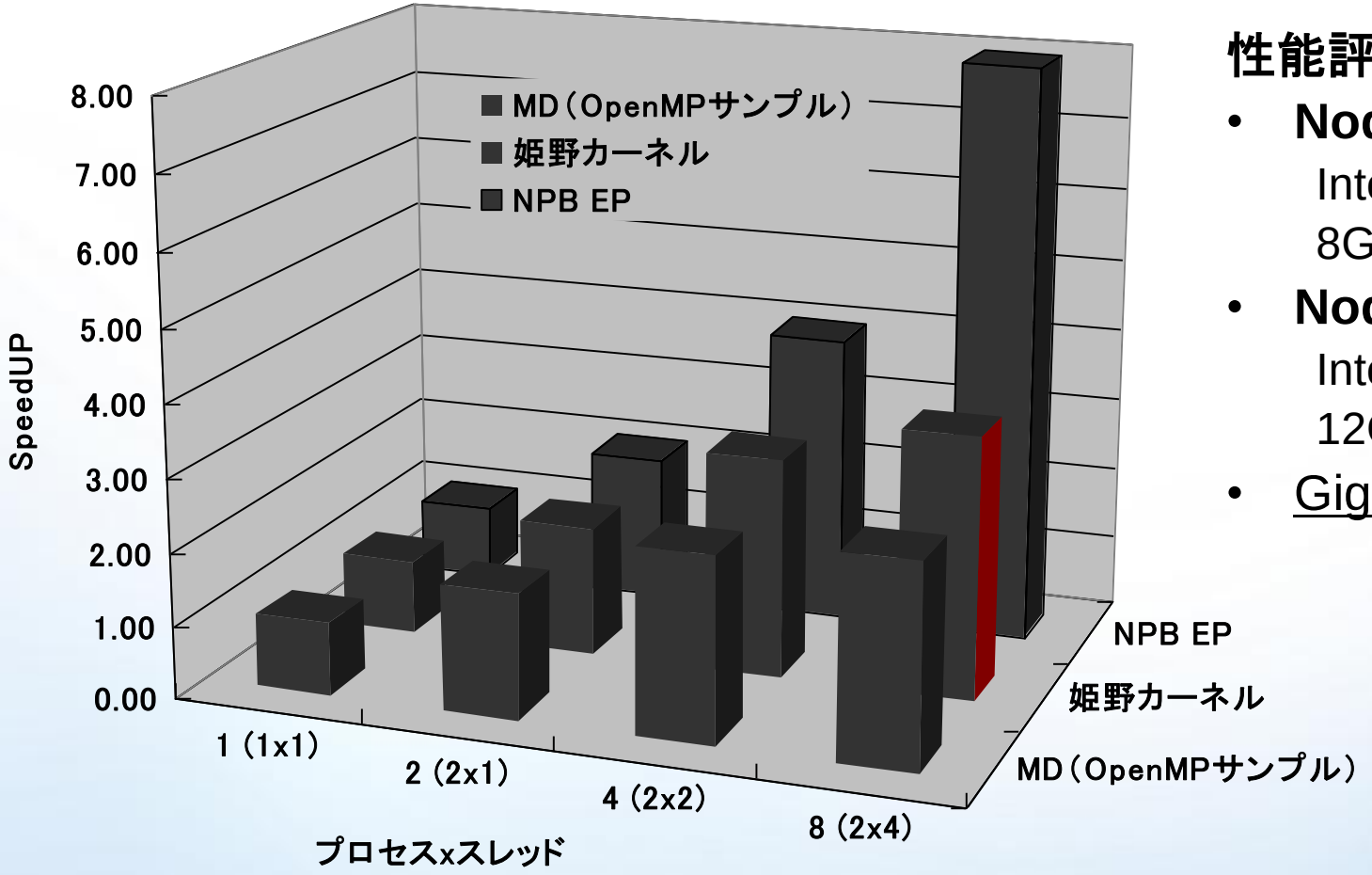
- 幾つかのベンチマークを実施した結果がインテルから報告されています。
- Data MiningやRenderingではある程度のスケーラビリティが示されています





インテル社のCluster OpenMP関連の資料より抜粋
Parallel Parallel Programming for Programming for Hybrid Hybrid Architectures Archite
Tom Lehmann
Technical Director Technical Director
HPC Programs Office HPC Programs Office
January 23, 2006

簡単なカーネルでの性能



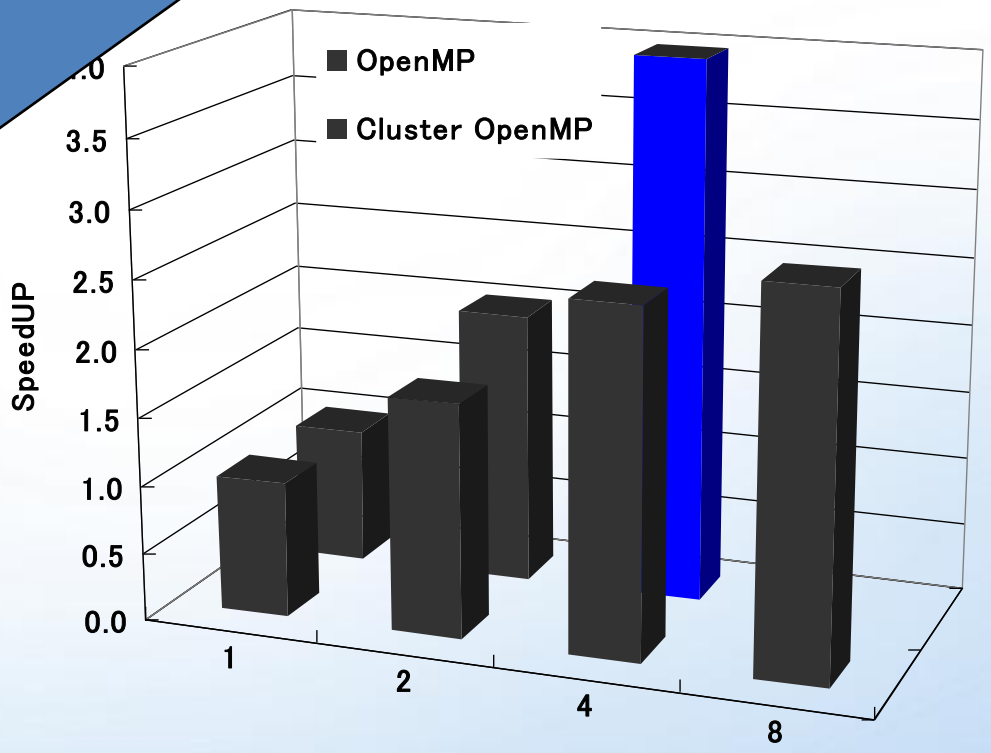
性能評価システム

- **Node1**
  Intel Xeon 5100 2.67GHz
  8GB memory
- **Node2**
  Intel Xeon 5100 2.67GHz
  12GB memory
- Gigabit Ethernet**での接続**

Cluster OpenMPでの注意点

Cluster OpenMPは、ソフトウエア上は、‘NUMA’の構成となるため、配置されるメモリの場所とそのアクセスによって、性能が大きく変わります。

```
!$omp  parallel
!$omp& default(shared)
!$omp& private(i,j,k,rij,d,pot_i,kin_i)
      pot_i = 0.0
      kin_i = 0.0
!$omp  do
      do i=1,np
! compute potential energy and forces
      f(1:nd,i) = 0.0
      do j=1,np
      if (i .ne. j) then
      call dist(nd,pos(1,i),pos(1,j),rij,d)
.......
! compute kinetic energy
      kin_i = kin_i + dotr8(nd,vel(1,i),vel(1,i))
      enddo
!$omp  end do
!$omp  critical
      kin = kin + kin_i
      pot = pot + pot_i
!$omp  end critical
!$omp  end parallel
```



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。

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® コンパイラー自動並列化ガイド　デュアルコア/ マルチコア対応アプリケーション開発 4 [PDF 形式 797 KB]**
- **インテル® Fortran コンパイラー: OpenMP* 活用ガイド　デュアルコア/ マルチコア対応アプリケーション開発 3 [PDF 形式 1,543 KB]**
- **インテル® C/C++コンパイラー: OpenMP* 活用ガイド　デュアルコア/ マルチコア対応アプリケーション開発 2 [PDF 形式 1,391 KB]**
- **インテル® コンパイラー: 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 [PDF 形式 1,577 KB]**

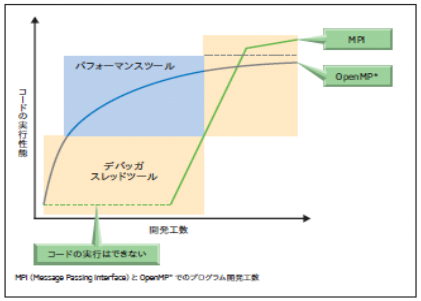
インテルコンパイラ OpenMP入門

インテル® コンパイラー
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

7. まとめ

「並列処理」と聞くと、何かプログラム上で特別なことを行っているような印象を受けるかもしれません。実際のところ、現在のシングルプロセッサー、シングルコアのマイクロプロセッサーでも、プロセッサー・コア内部で多くの並列処理が行われており、最新の高速マイクロプロセッサーでは、プロセッサー内に並列処理のための非常に多くのリソースを実装しています。これらのリソースを同時に利用することで、プログラムの高速実行を可能としています。インテル® Itanium® 2 プロセッサーなどはそのもっとも進んだ例です。コンパイラーは高度にプログラムを解析し、命令レベルでの並列化 (ILP) を既に高度なレベルで実現しています。



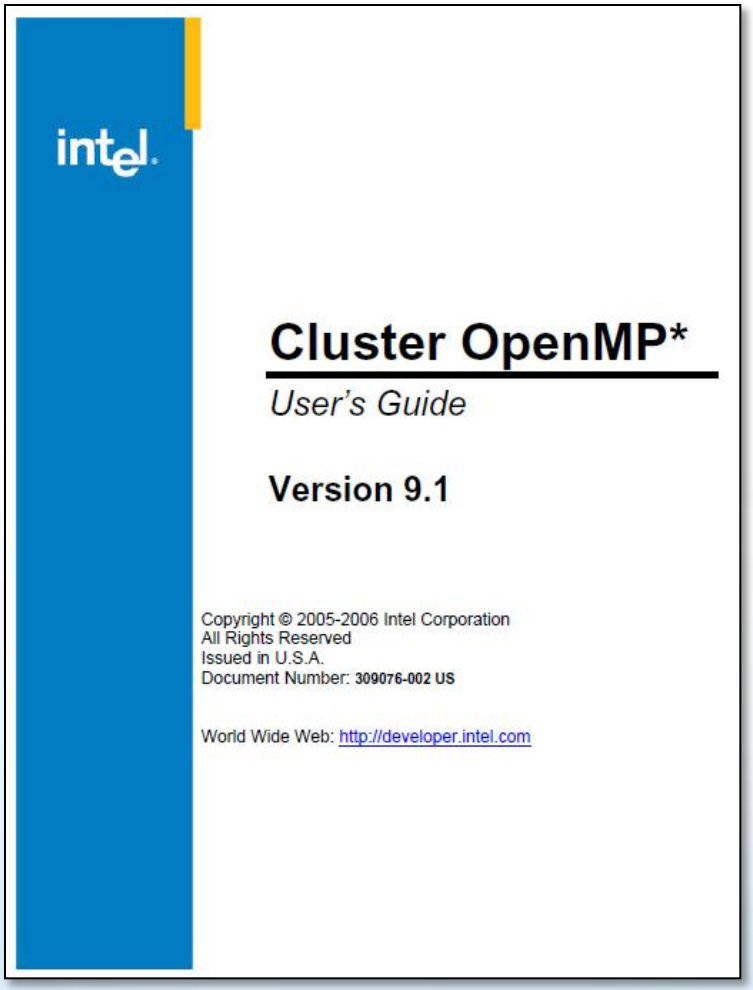
MPI (Message Passing Interface) と OpenMP* でのプログラム開発工数

パフォーマンスに対する高度な要求に答える形で、プロセッサーは高速化の一途をたどってきました。しかし、現在プロセッサーとメモリーの性能格差が広がるにつれて、様々なアプリケーションにおいて、メモリー・レイテンシーがパフォーマンス面でのボトルネックになっています。またプロセッサーの消費電力と発熱量も大きな問題です。このような状況を打破するためにも、デュアルコアやマルチコアといった最新のプロセッサーの実装技術が求められています。自動並列化、OpenMP* でのプログラム処理の並列化は、このような時代の要求に応えるものです。コンパイラーの更なる進化によって、今後、これらの機能はさらに強化されていきます。

様々なレベルでの並列処理において、それらの並列処理技術を緊密に結合することで、アプリケーションの性能を大幅に向上させることが可能です。

執筆者プロフィール
スケーラブルシステムズ株式会社
代表取締役　　戸室　隆彦
約 20 年間にわたる HPC およびハイエンドコンピューティングの経験と、日本 SGI における CTO（チーフテクノロジー・オフィサー）としての幅広い経験を基に、2005 年 6 月、HPC およびハイエンドコンピューティングに関する技術コンサルテーションを提供する「スケーラブルシステムズ株式会社」を設立。URL：http://www.scte.co.jp/

Cluster OpenMPに関する資料・ドキュメント


intel

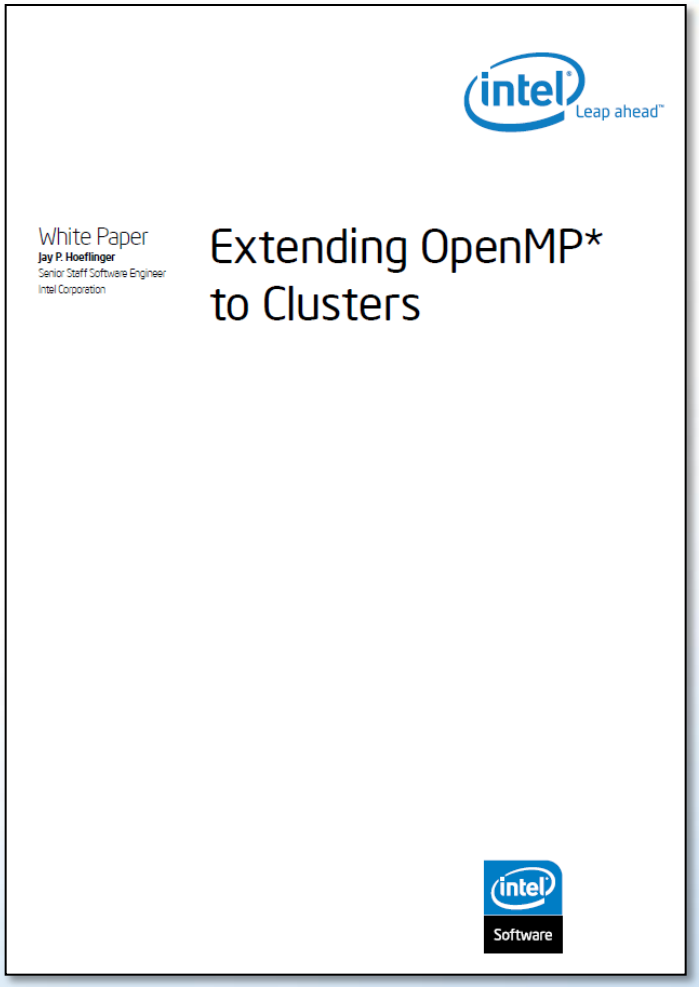
Cluster OpenMP*

User's Guide

Version 9.1

Issued in U.S.A.
Document Number: **309076-002 US**

World Wide Web: http://developer.intel.com


intel Leap ahead™

White Paper

Jay P. Hoeflinger
Senior Staff Software Engineer
Intel Corporation

Extending OpenMP* to Clusters

intel Software

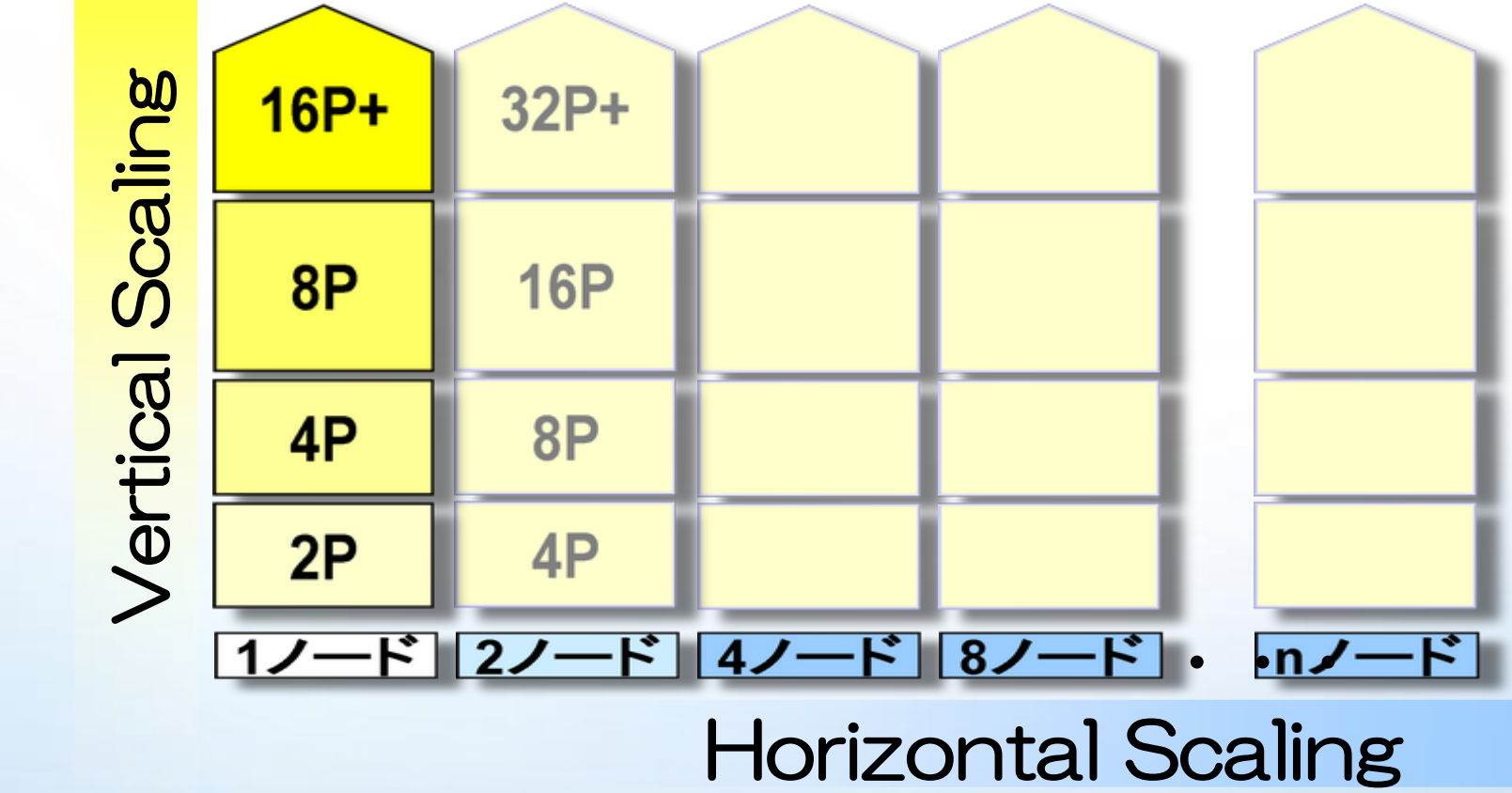
インテルコンパイラ関連資料(英文)

- 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

シングルAPIでの並列処理

MPI
OpenMP

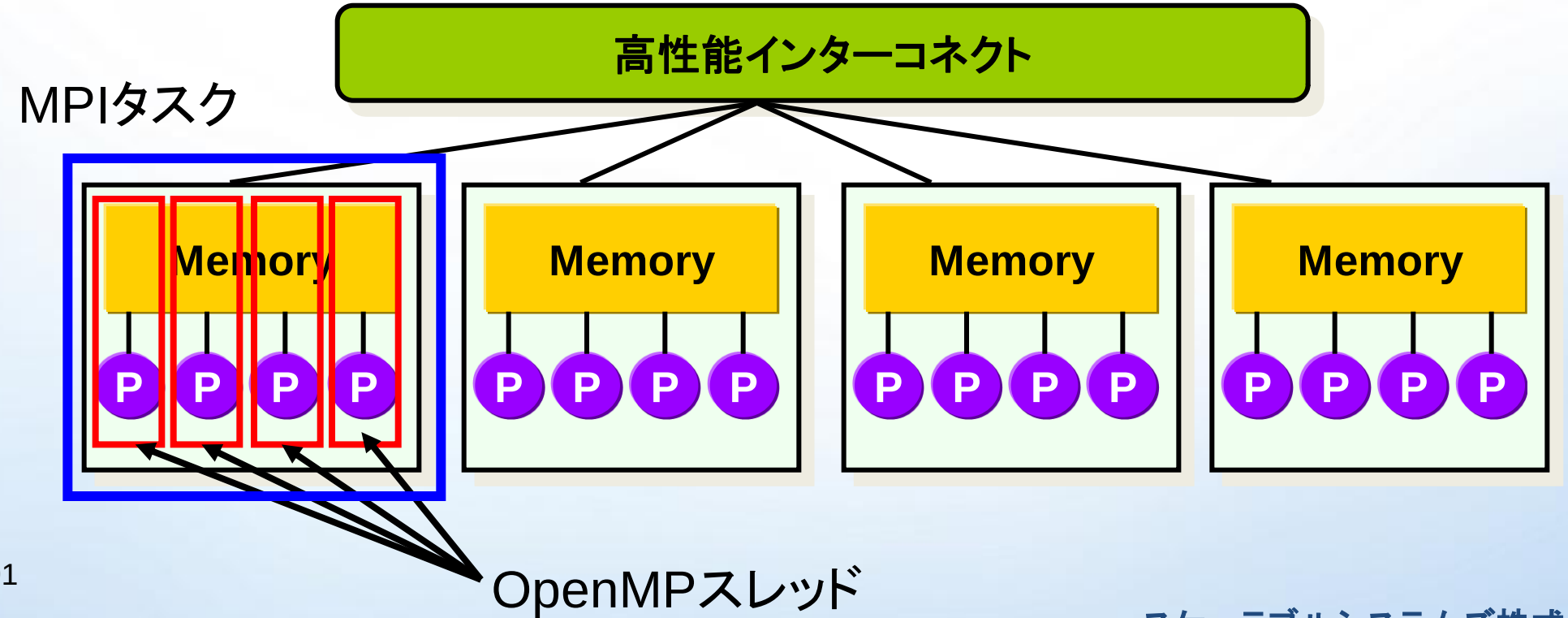
	OpenMP	MPI
ノード内	○	○
ノード間	???	○



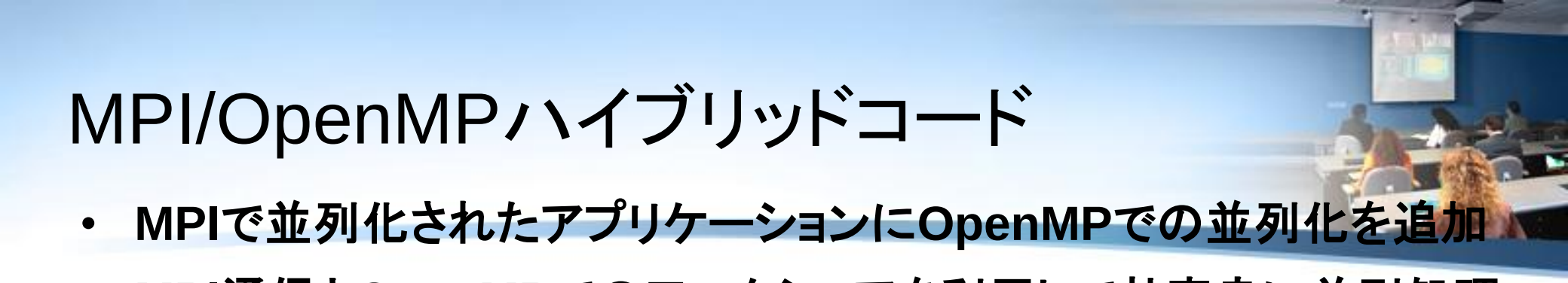
MPI
OpenMP????

MPI/OpenMPハイブリッドモデル

- **MPIでは領域分割などの疎粒度での並列処理を行う**
- **OpenMPは、各MPIタスク内で、ループの並列化などのより細粒度での並列化を担う**
- **計算は、タスク-スレッドの階層構造を持つ**



MPI/OpenMPハイブリッドコード



- **MPIで並列化されたアプリケーションにOpenMPでの並列化を追加**
- **MPI通信とOpenMPでのワークシェアを利用して効率良い並列処理の実現**

Fortran

```
include ‘mpif.h’
program hybsimp


call MPI_Init(ierr)
call MPI_Comm_rank (...,irank,ierr)
call MPI_Comm_size (...,isize,ierr)
! Setup shared mem, comp. & Comm
!$OMP parallel do
      do i=1,n
        <work>
      enddo
! compute & communicate
call MPI_Finalize(ierr)
end
```

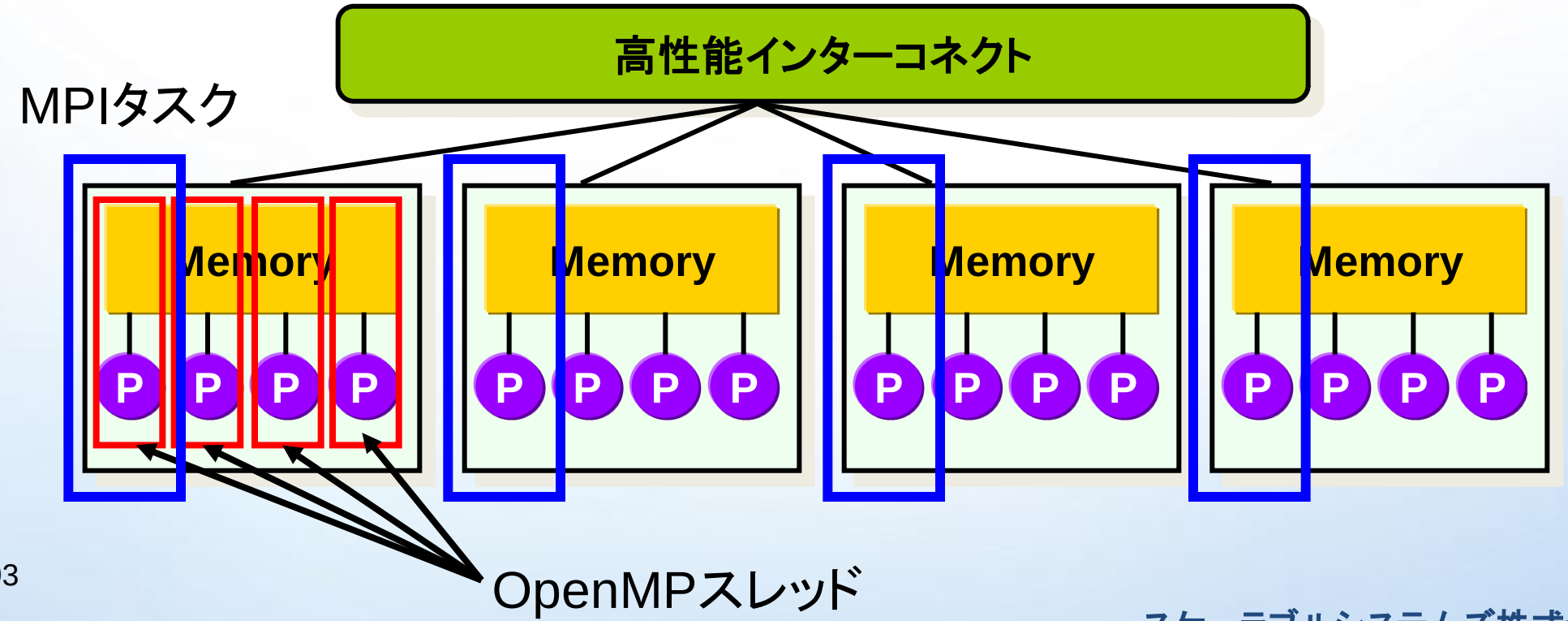
C/C++

```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;


ierr= MPI_Init(&argc,&argv[]);
ierr= MPI_Comm_rank (...,&rank);
ierr= MPI_Comm_size (...,&size);
//Setup shared mem, compute & Comm
#pragma omp parallel for
   for(i=0; i<n; i++){
      <work>
   }
// compute & communicate
ierr= MPI_Finalize();
```

OpenMP/MPIハイブリッドモデル

- MPIは実績のある高性能な通信ライブラリ
- 計算と通信を非同期に実行することも可能
- 通信はマスタースレッド、シングルスレッド、全スレッドで実行することが可能



OpenMP/MPIハイブリッドコード

- **OpenMPのプログラムにMPI通信を追加**
- **既存のOpenMPプログラムの拡張やスレッドプログラムの新規開発時のオプションとして選択**
- **MPIは非常に高速また最適化されたデータ通信ライブラリ**

Fortran

```
include ‘mpif.h’
program hybmas


!$OMP parallel

  !$OMP barrier
  !$OMP master
  call MPI_<Whatever>(…,ierr)
  !$OMP end master
  !$OMP barrier


!$OMP end parallel
end
```

C/C++

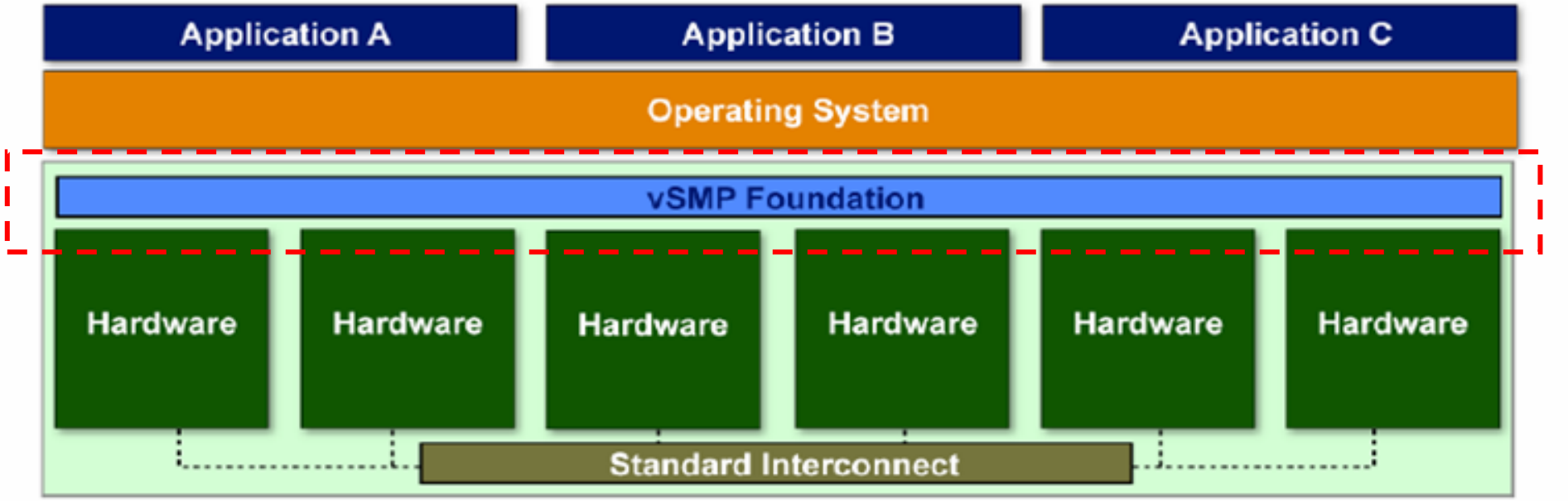
```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;


#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp master
  {
  ierr=MPI_<Whatever>(…)
  }
  #pragma omp barrier

}
```

ScaleMP vSMPアーキテクチャ

アプリケーションについては、他のx86システムと100%のバイナリ互換を実現
OSは通常のLinuxディストリビューションが利用可能



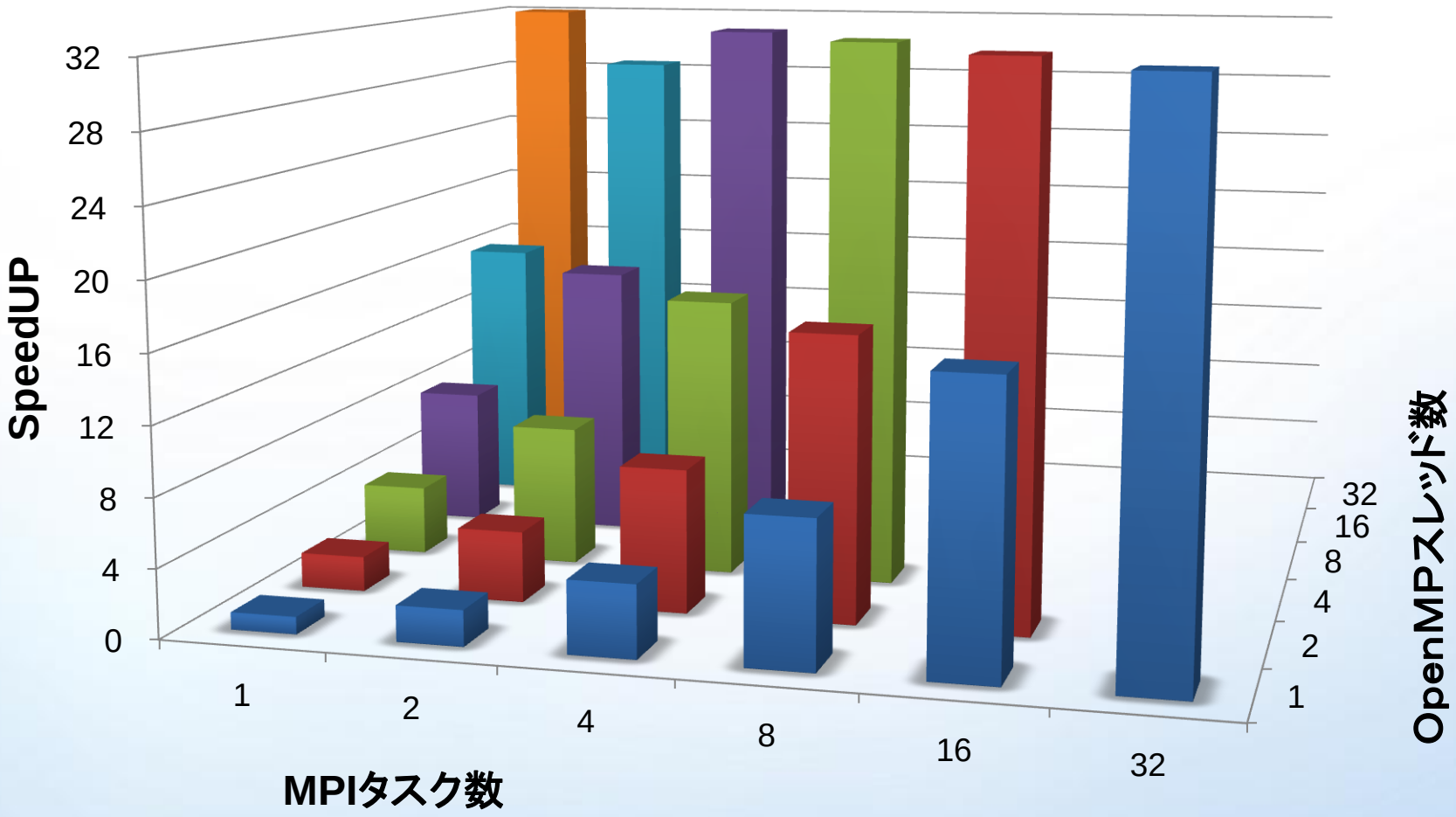
Hardwareは一般のx86チップセットと標準インターコネクトでシステムの構築が可能

vSMP Foundation でのシステムのSMP拡張を実現

OpenMP/MPI/ハイブリッド

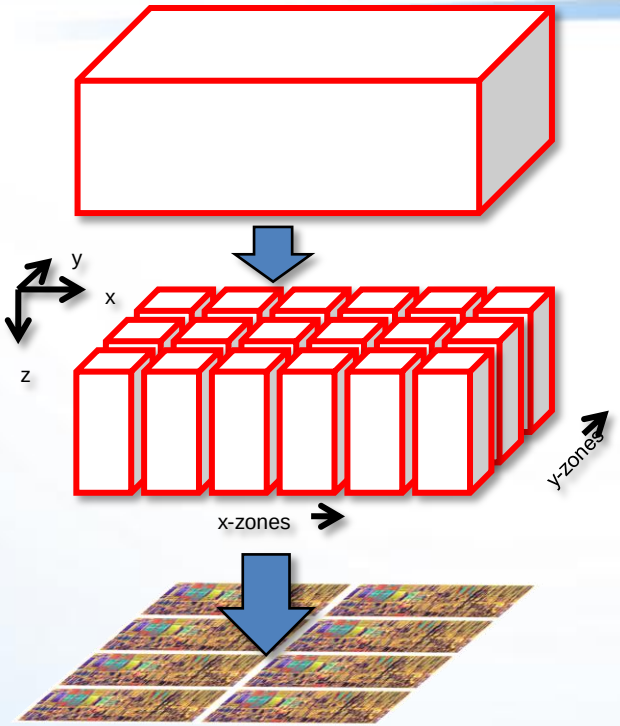
Hybrid OpenMP MPI Benchmarkproject (″homb″)

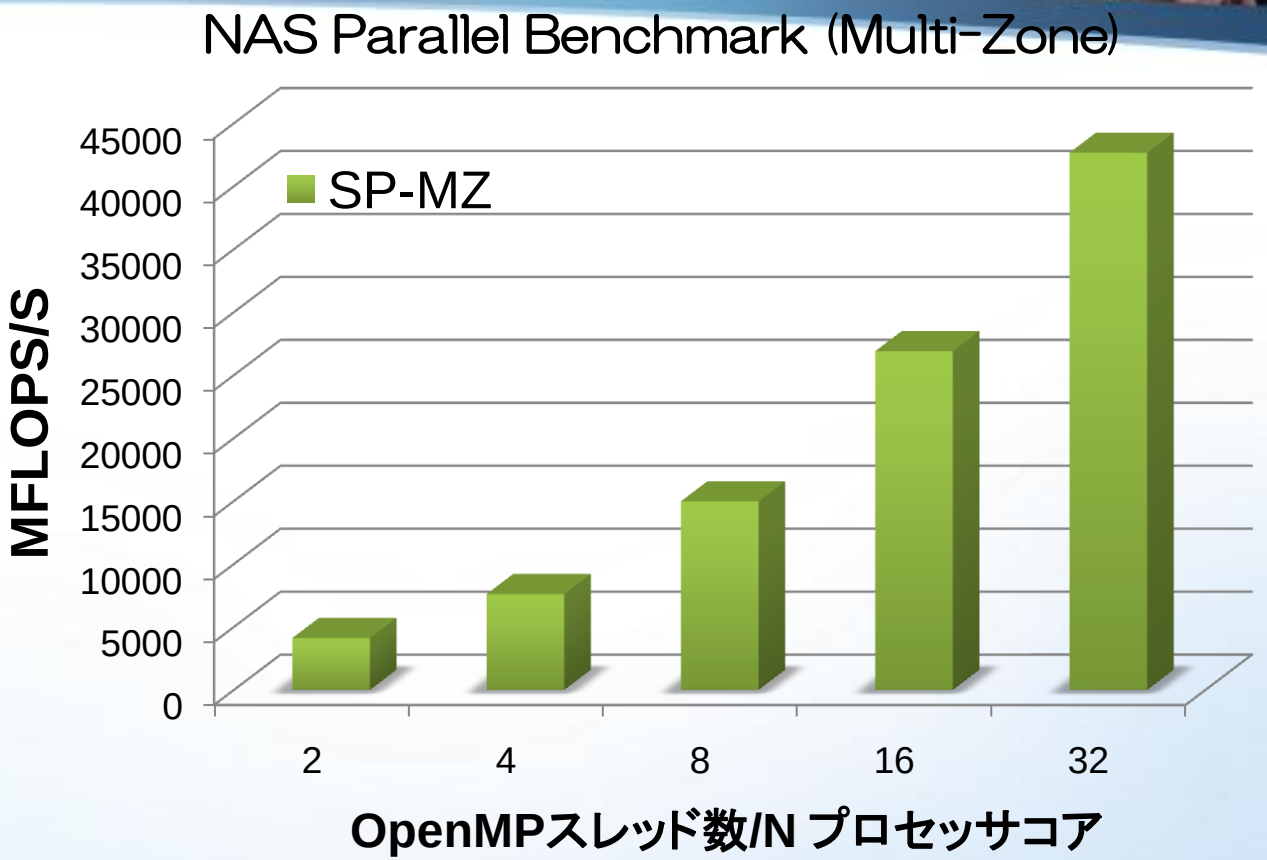
This is the Hybrid OpenMP MPI Benchmarkproject (″homb″)
This project was registered on SourceForge.net on May 16, 2009, and is described by the project team as follows:
HOMB is a simple benchmark based on a parallel iterative Laplace solver aimed at comparing the performance of MPI, OpenMP, and hybrid codes on SMP and multi-core based machines.



スケーラブルシステムズ株式会社

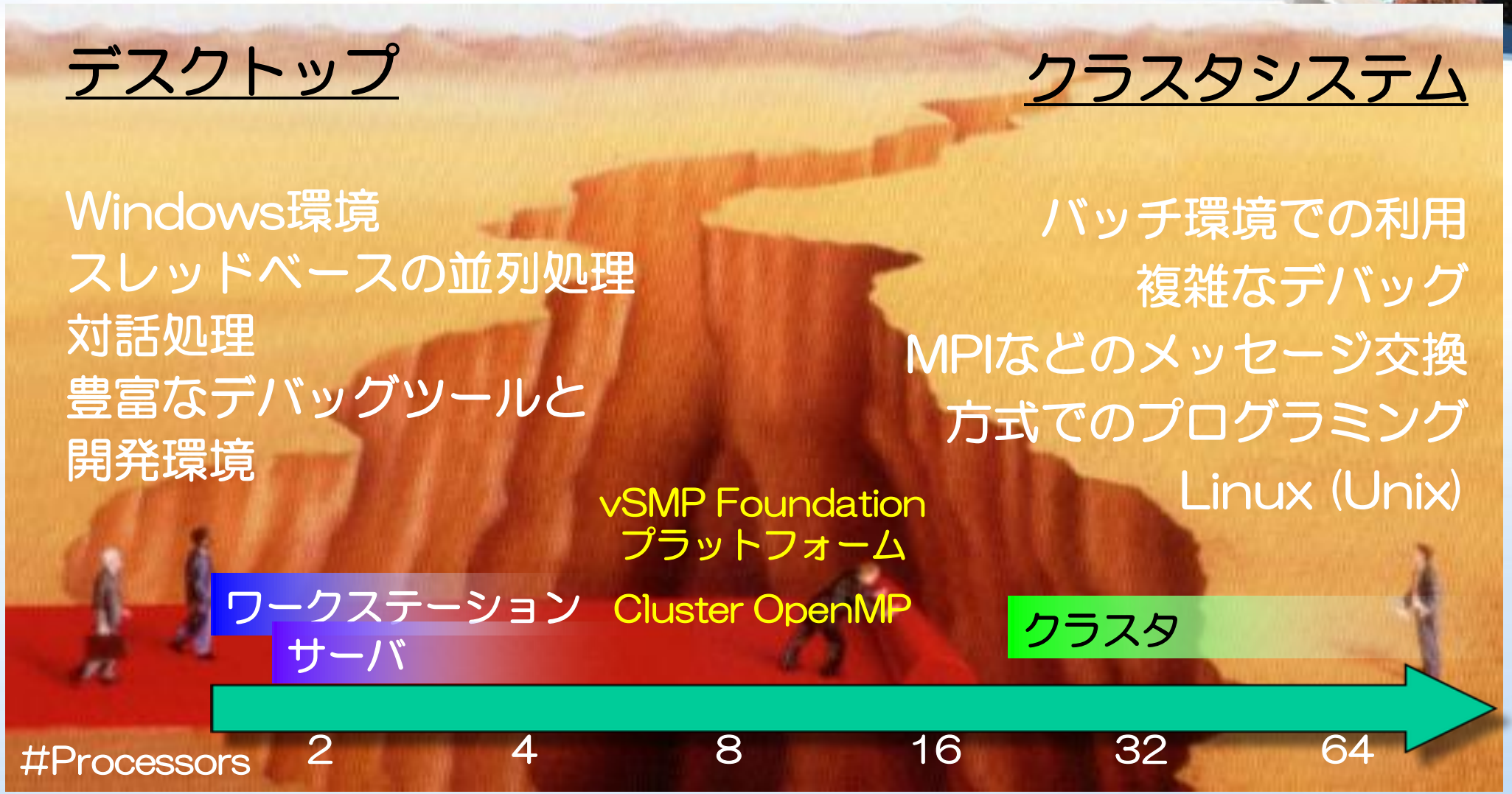
OpenMPベンチマーク

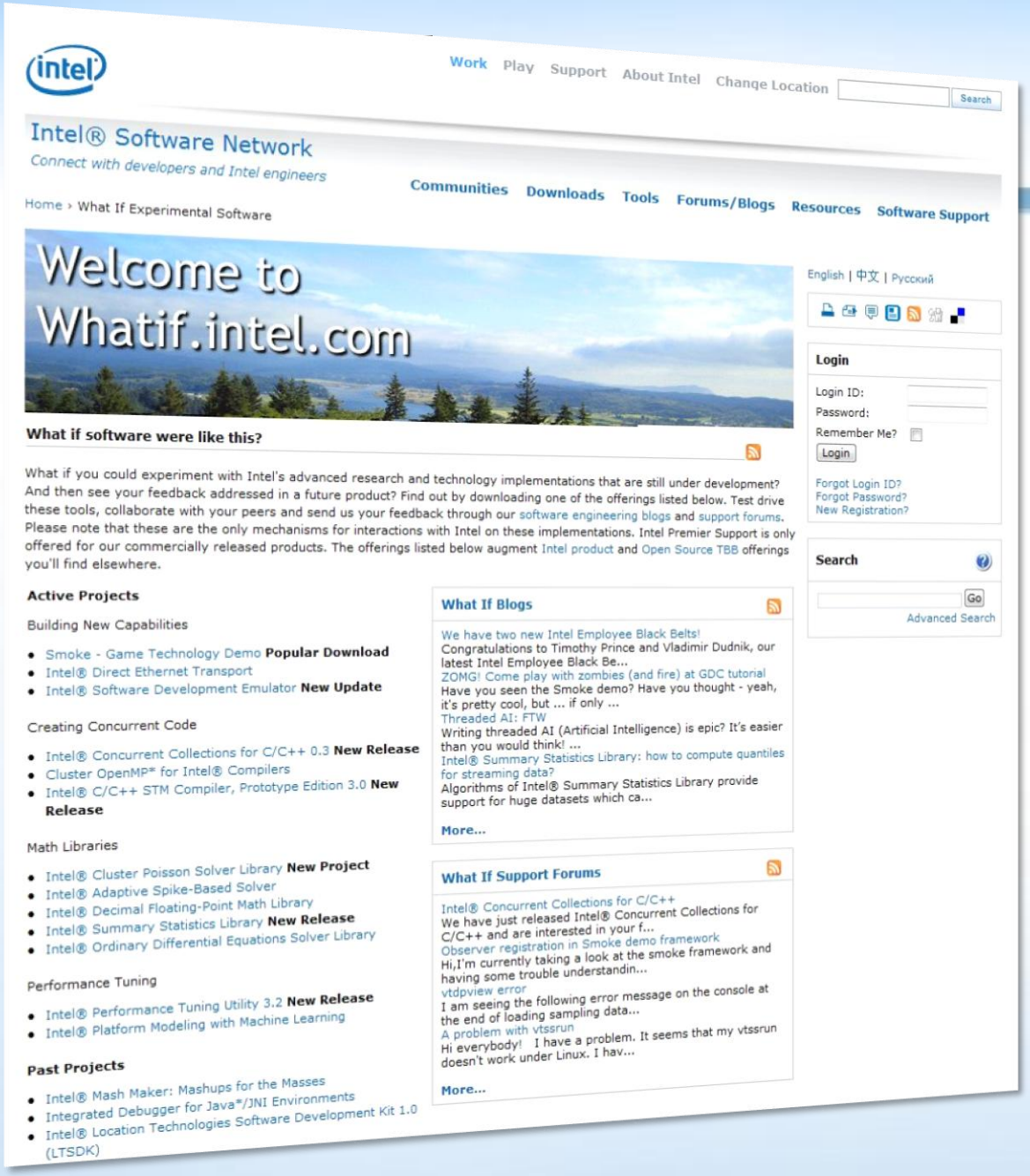




著名な公開ベンチマークツールであるNAS Parallel Benchmark (NPB) の一つであるNPB-MZ (NPB Multi-Zone)はより粒度の大きな並列化の提供を行っています。NPB-MZでは、ハイブリッド型の並列処理やネストしたOpenMPのテストが可能です。ここでの結果は、OpenMPだけでの並列処理の性能を評価しています。
Xeon 5550 (2.66GHz) vSMP Foundation

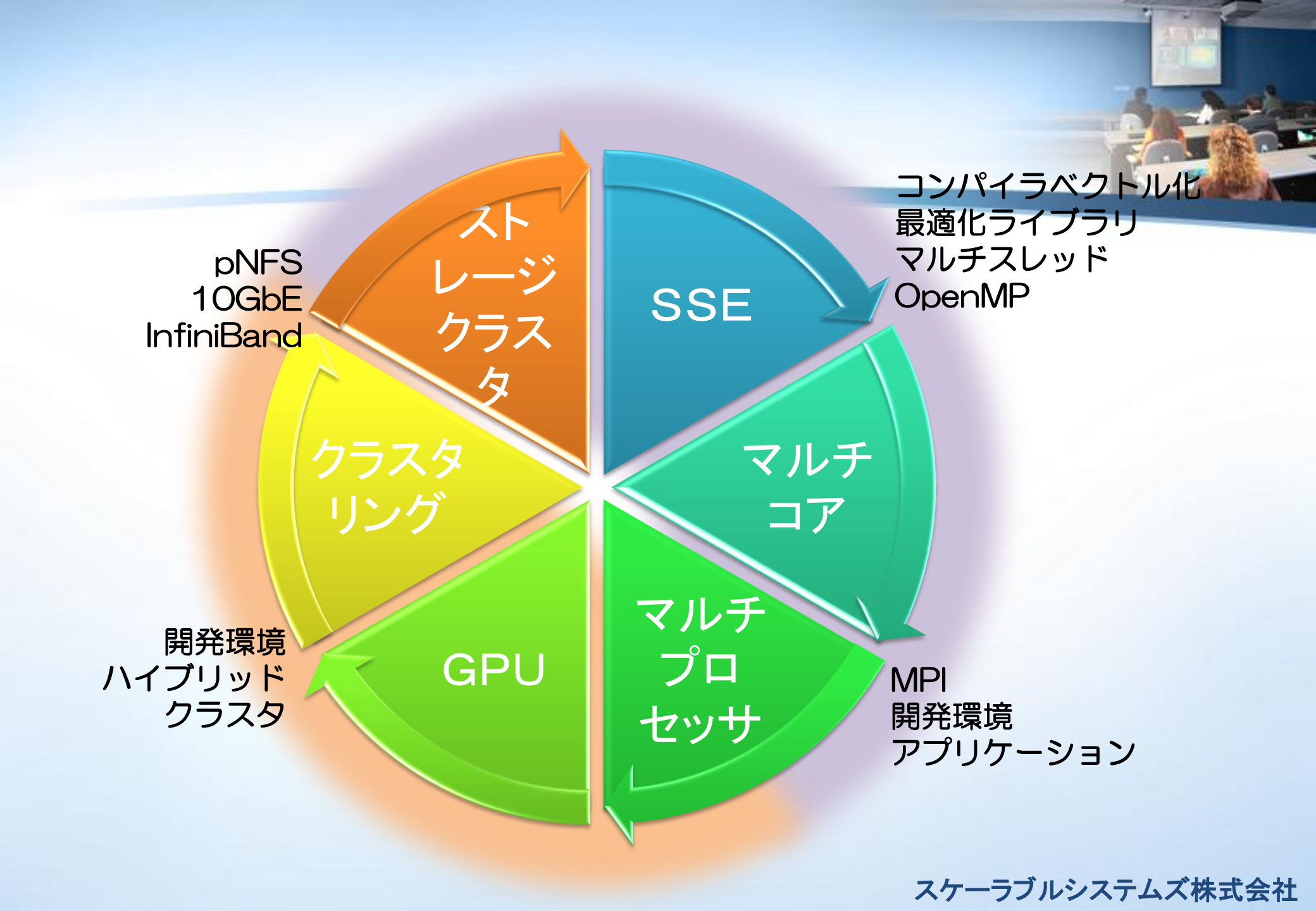
ソフトウエアのギャップの解決



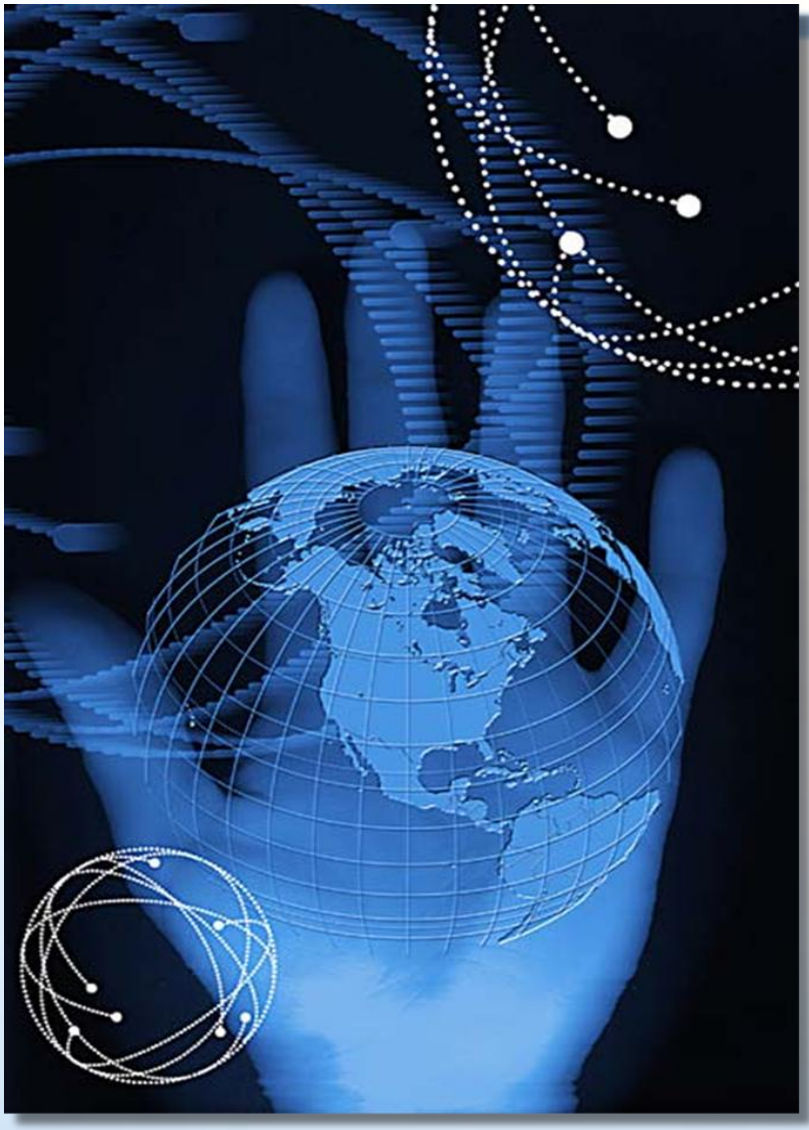


- *New Parallel Languages*
- *New Threading tools*
- *Thread Management & Abstraction layers*
- *Transactional memory*
- *Auto-threading compilers*
- *Auto-threading hardware*

スケーラブルシステムズ株式会社


ストレージクラスタ
SSE
マルチコア
マルチプロセッサ
GPU
クラスタリング
pNFS
10GbE
InfiniBand
コンパイラベクトル化
最適化ライブラリ
マルチスレッド
OpenMP
MPI
開発環境
アプリケーション
開発環境
ハイブリッド
クラスタ

お見積りのご依頼・お問い合わせ



お問い合わせ
0120-090715
携帯電話・PHSからは（有料）
03-5875-4718
9:00-18:00 (土日・祝日を除く)

WEBでのお問い合わせ
www.sstc.co.jp/contact