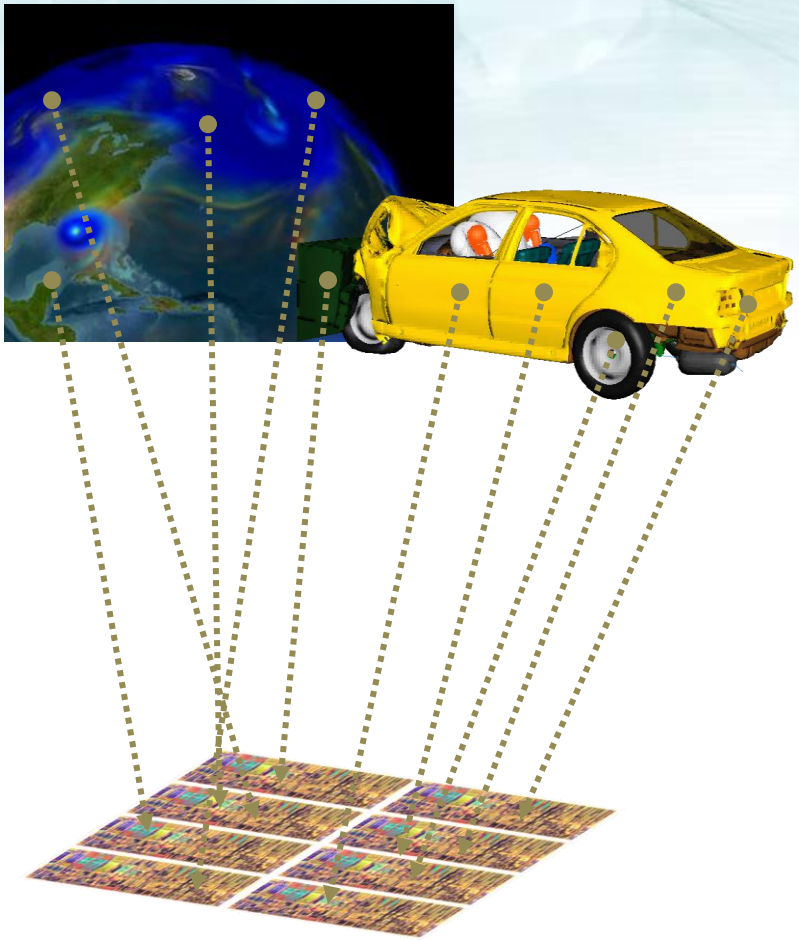
# マルチスレッドプログラミング

スケーラブルシステムズ株式会社

# なぜ、並列処理は容易でないのか？



不規則なオペレーション
複雑なデータ構造
アルゴリズム上の問題

↓

マルチコア上での並列処理の難しさ

↓

継続的なプロセッサコア数の増加
ベクトル処理の強化
メモリシステムの強化
キャッシュシステムの改善
……

# なぜ、並列処理は容易でないのか？

Dr. Dobb's

## Rules for Parallel Programming for Multicore

James offers eight key rules for multicore programming based on parallel programming experiences of his own and others.

By James Reinders, Dr. Dobb's Journal
9 05, 2007
URL:http://www.ddj.com/hpc-high-performance-computing/201804248

*James is part of Intel's Software Development Products team, and author of Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. He can be reached at james.r.reinders@intel.com.*

---

Programming for multicore processors poses new challenges. Here are eight rules for multicore programming to help you be successful:

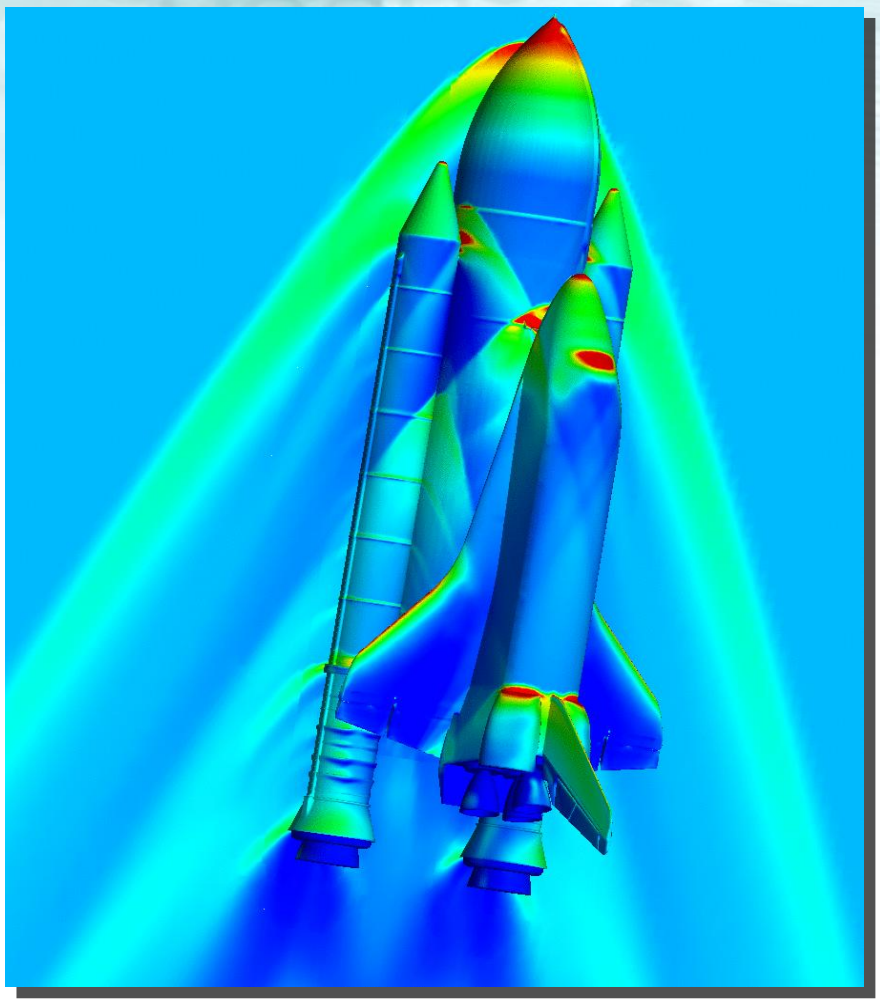
1. Think parallel. Approach all problems looking for the parallelism. Understand where parallelism is, and organize your thinking to express it. Decide on the best parallel approach before other design or implementation decisions. Learn to "Think Parallel."
2. Program using abstraction. Focus on writing code to express parallelism, but avoid writing code to manage threads or processor cores. Libraries, OpenMP, and Intel Threading Building Blocks are all examples of using abstractions. Do not use raw native threads (pthreads, Windows threads, Boost threads, and the like). Threads and MPI are the assembly languages for parallelism. They offer maximum flexibility, but require too much time to write, debug, and maintain. Your programming should be at a high-enough level that your code is about your problem, not about thread or core management.
3. Program in tasks (chores), not threads (cores). Leave the mapping of tasks to threads or processor cores as a distinctly separate operation in your program, preferably an abstraction you are using that handles thread/core management for you. Create an abundance of tasks in your program, or a task that can be spread across processor cores automatically (such as an OpenMP loop). By creating tasks, you are free to create as many as you can without worrying about oversubscription.
4. Design with the option to turn concurrency off. To make debugging simpler, create programs that can run without concurrency. This way, when debugging, you can run programs first with—then without—concurrency, and see if both runs fail or not. Debugging common issues is simpler when the program is not running concurrently because it is more familiar and better supported by today's tools. Knowing that something fails only when run concurrently hints at the type of bug you are tracking down. If you ignore this rule and can't force your program to run in only one thread, you'll spend too much time debugging. Since you want to have the capability to run in a single thread specifically for debugging, it doesn't need to be efficient. You just need to avoid creating parallel programs that require concurrency to work correctly, such as many producer-consumer models. MPI programs often violate this rule, which is part of the reason MPI programs can be problematic to implement and debug.
5. Avoid using locks. Simply say "no" to locks. Locks slow programs, reduce their scalability, and are the source of bugs in parallel programs. Make implicit synchronization the solution for your program. When you still need explicit synchronization, use atomic operations. Use locks only as a last resort. Work hard to design the need for locks completely out of your program.
6. Use tools and libraries designed to help with concurrency. Don't "tough it out" with old tools. Be critical of tool support with regards to how it presents and interacts with parallelism. Most tools are not yet ready for parallelism. Look for threadsafe libraries—ideally ones that are designed to utilize parallelism themselves.
7. Use scalable memory allocators. Threaded programs need to use scalable memory allocators. Period. There are a number of solutions and I'd guess that all of them are better than *malloc()*. Using scalable memory allocators speeds up applications by eliminating global bottlenecks, reusing memory within threads to better utilize caches, and partitioning properly to avoid cache line sharing.
8. Design to scale through increased workloads. The amount of work your program needs to handle increases over time. Plan for that. Designed with scaling in mind, your program will handle more work as the number of processor cores increase. Every year, we ask our computers to do more and more. Your designs should favor using increases in parallelism to give you advantages in handling bigger workloads in the future.

1. Learn to “Think Parallel.”
2. Program using abstraction.
3. Program in tasks(chores), not threads(cores).
4. Design with the option to turn concurrency off.
5. Avoid using locks. Avoid when you can, use when you must.
6. Get tools to help.
7. Use scalable memory allocators.
8. Design to scale through increased workloads.

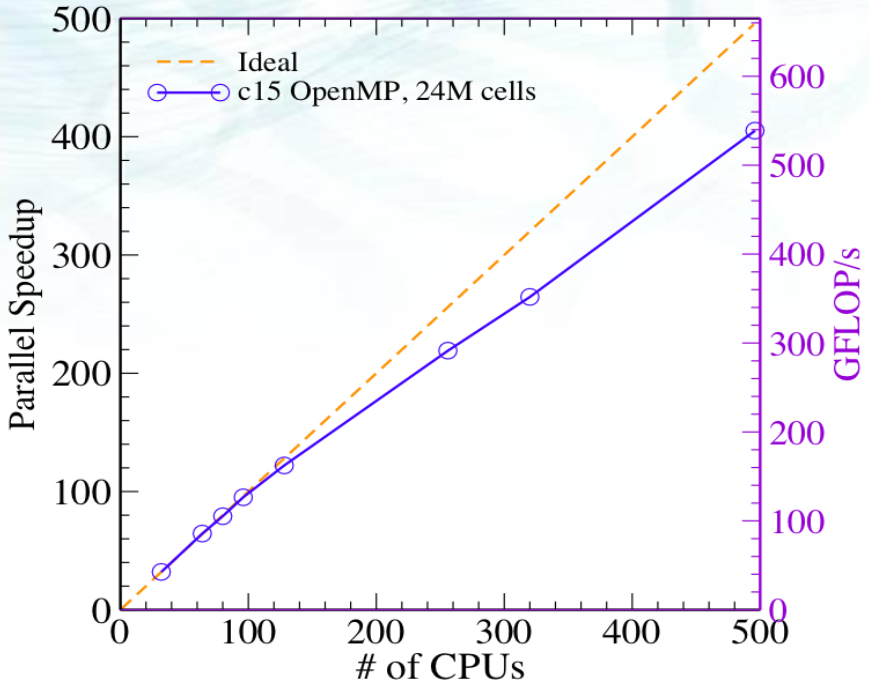
マルチコアでのプログラミングでのルール：これら8つのルール全てを理解して、プログラミングに取り組む必要がある

http://www.ddj.com/hpc-high-performance-computing/201804248

# NASAによる流体解析コードの性能評価



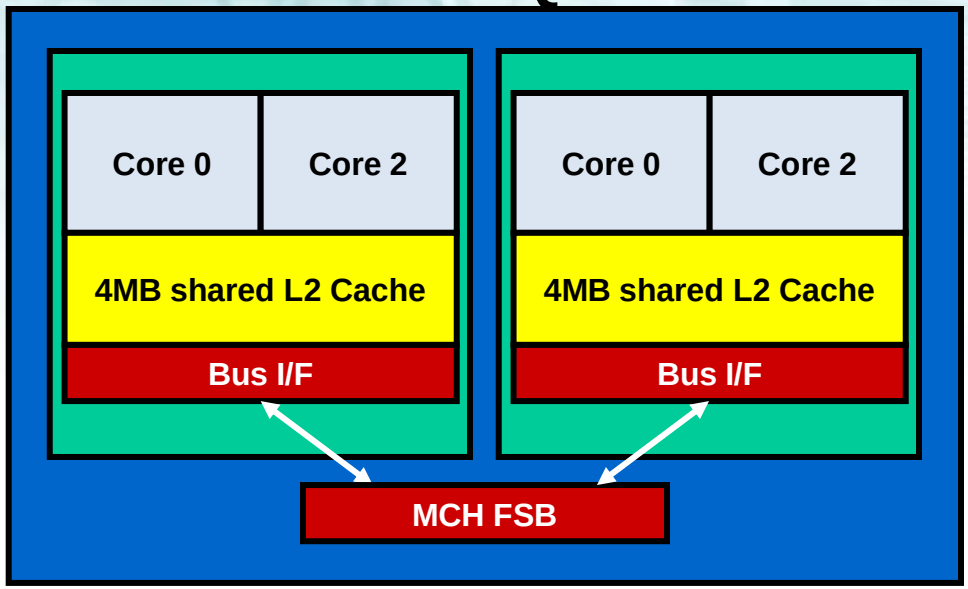
Virtual Flight on High-Performance Architectures
M. J. Aftosmis, S. M. Murman, M. Nemec, NASA Ames
SC2004,Pittsburgh, PA, Nov. 6-12, 2004
**Graphics courtesy of NASA Ames**



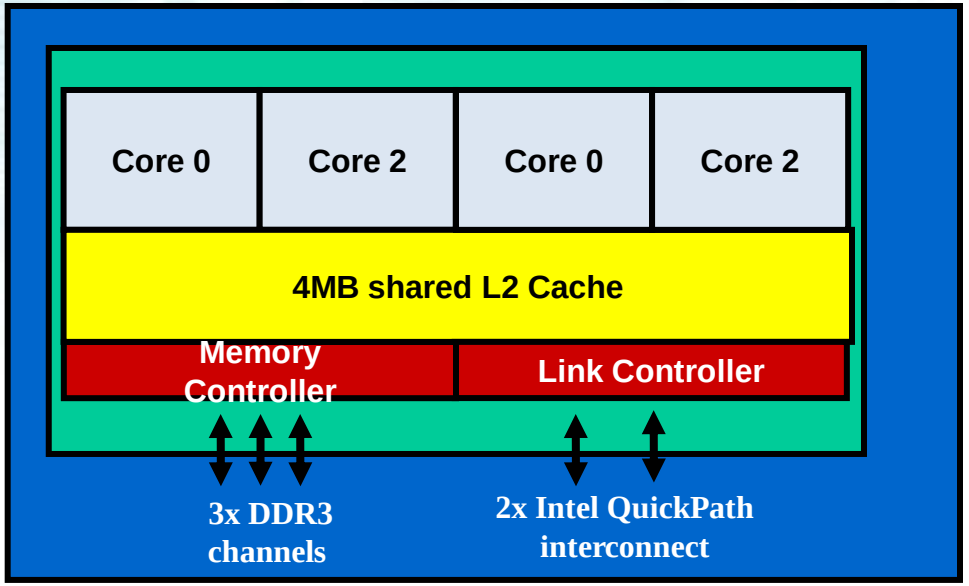

並列性能

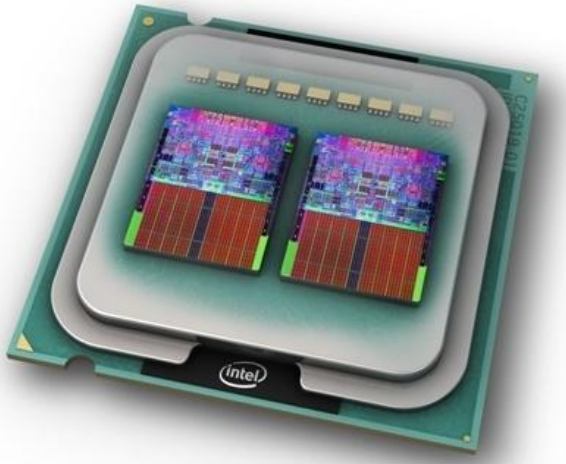
- 496プロセッサで405倍の性能向上が可能
  - 540 GFLOP/s
  - CPUあたりの性能：1.33 GFLOP/s
- 短時間でのシュミレーションを可能とし、問題への緊急的な対応を可能となります。

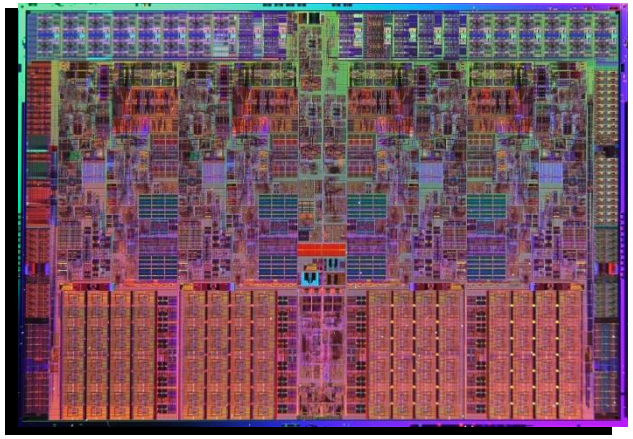
## Core 2 Extreme QX6700



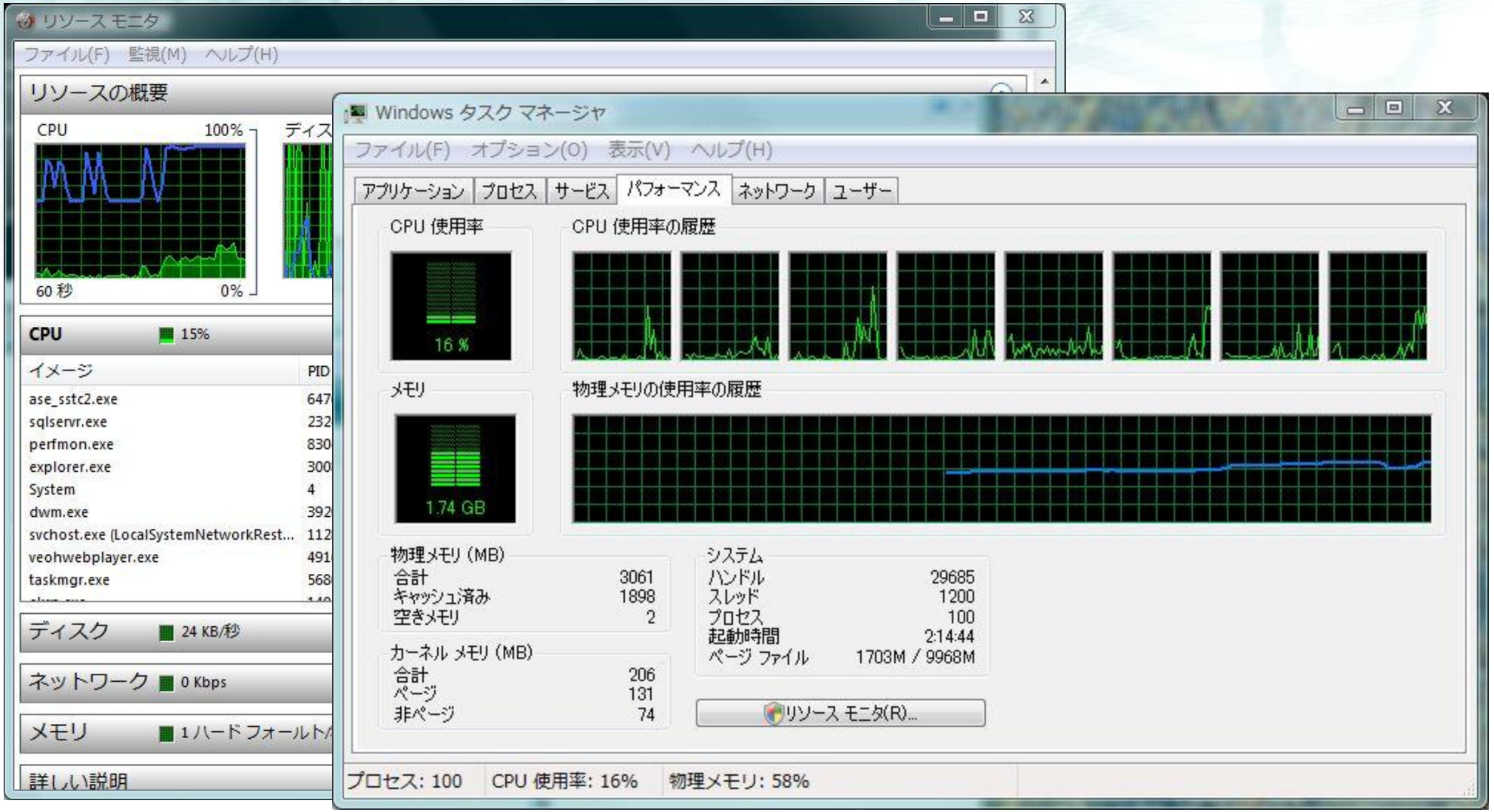

## Nehalem



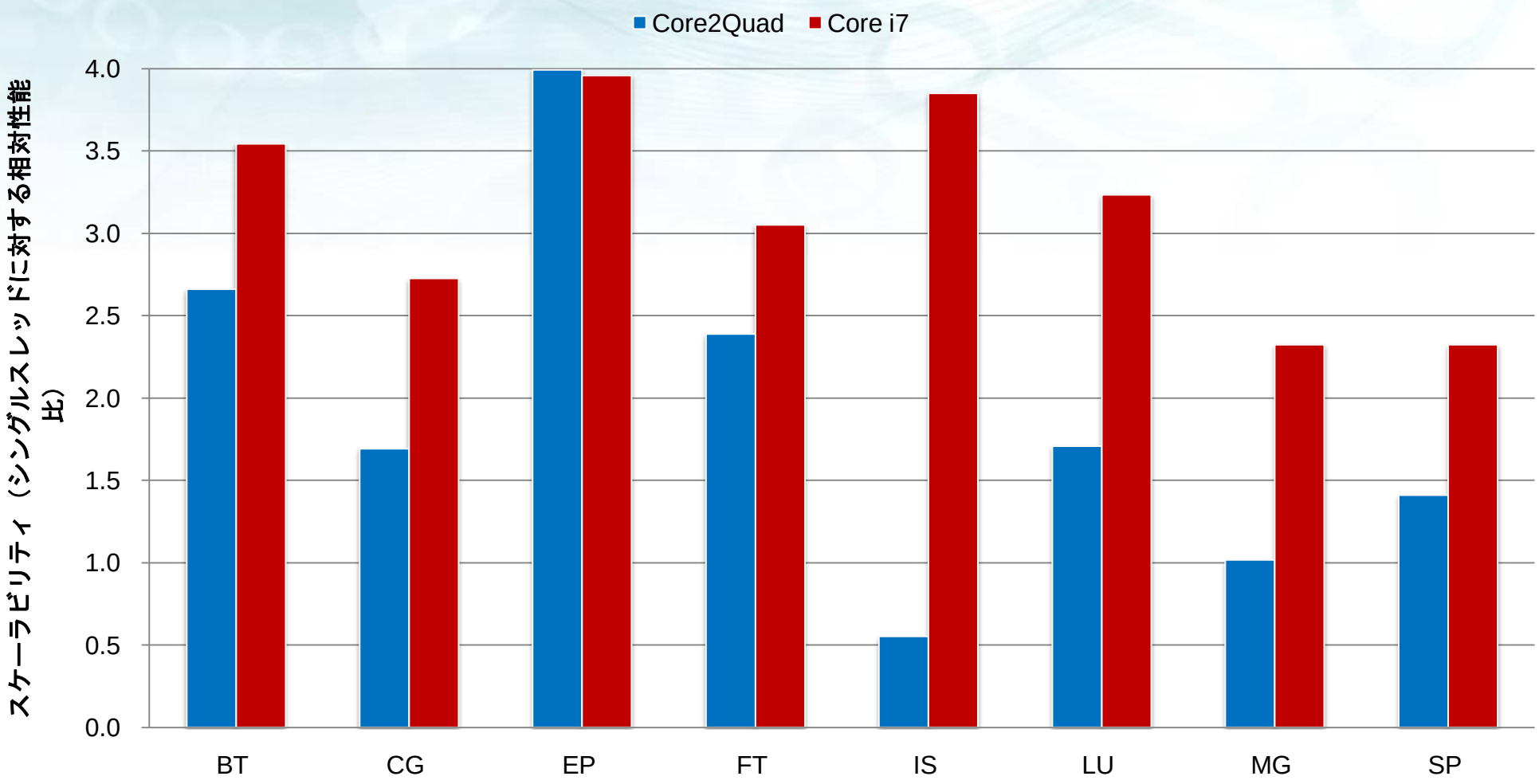



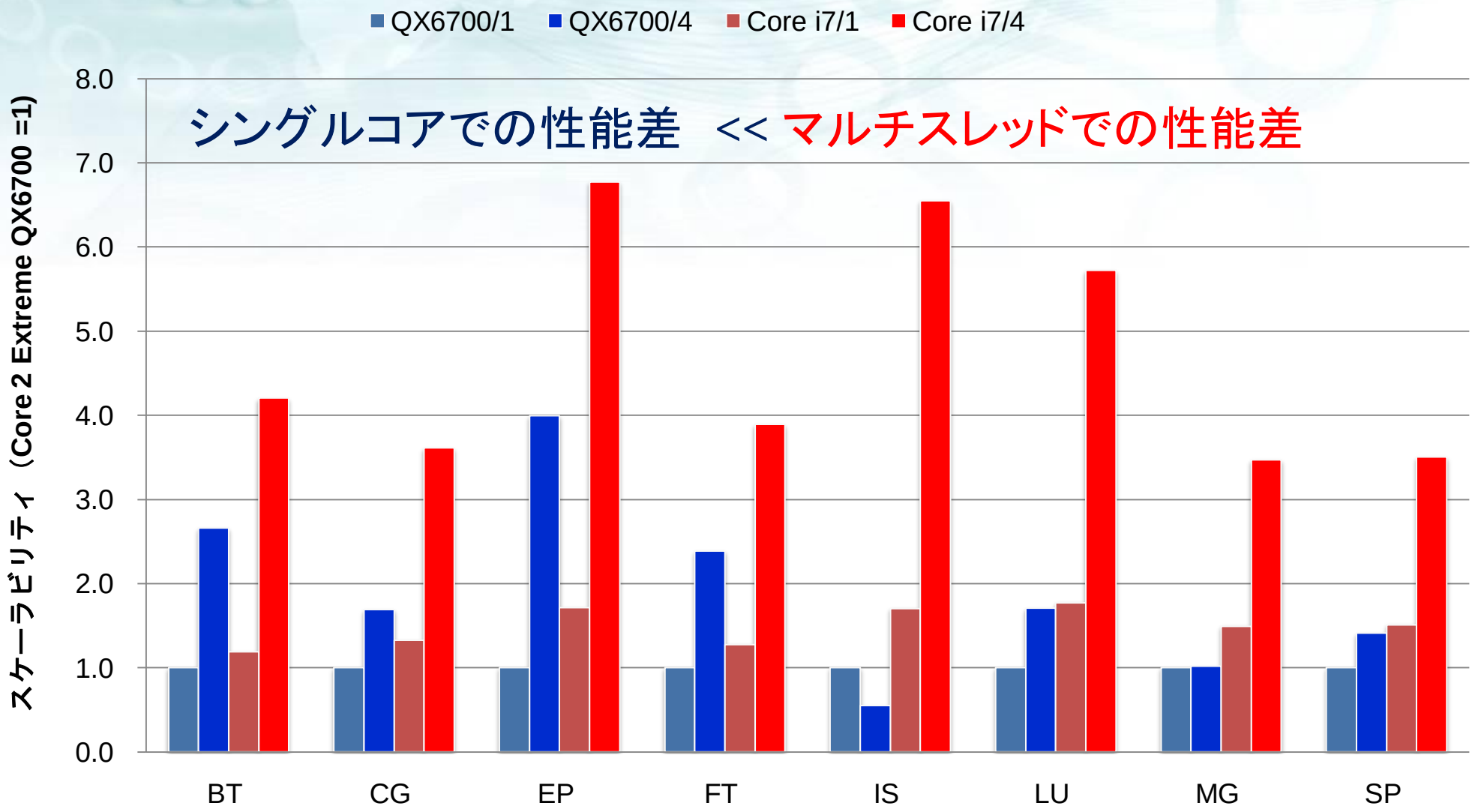
# Core i7 – 4コア、8スレッド(HT)



# OpenMPスケーラビリティ



# OpenMPスケーラビリティ

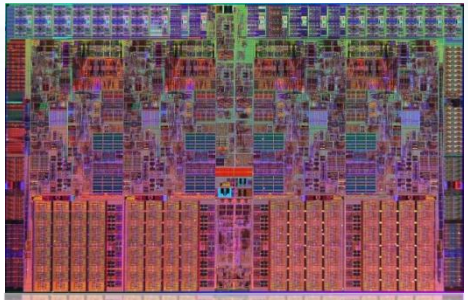


# Intel マイクロアーキテクチャ

Performance / core

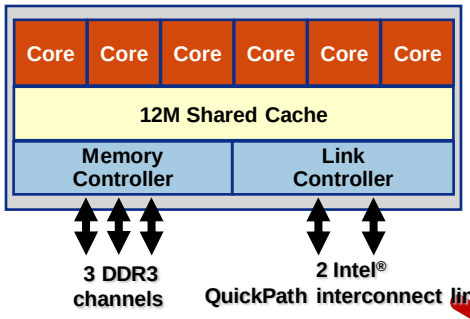
新マイクロアーキテクチャ
新製造プロセス

新マイクロアーキテクチャ（Sandy Bridge)
Intel AVX
ベクトル演算（浮動小数点演算性能 x2)



次世代マイクロアーキテクチャ（Nehalem）
メモリバンド幅とレイテンシの改善
マイクロアーキテクチャの改善

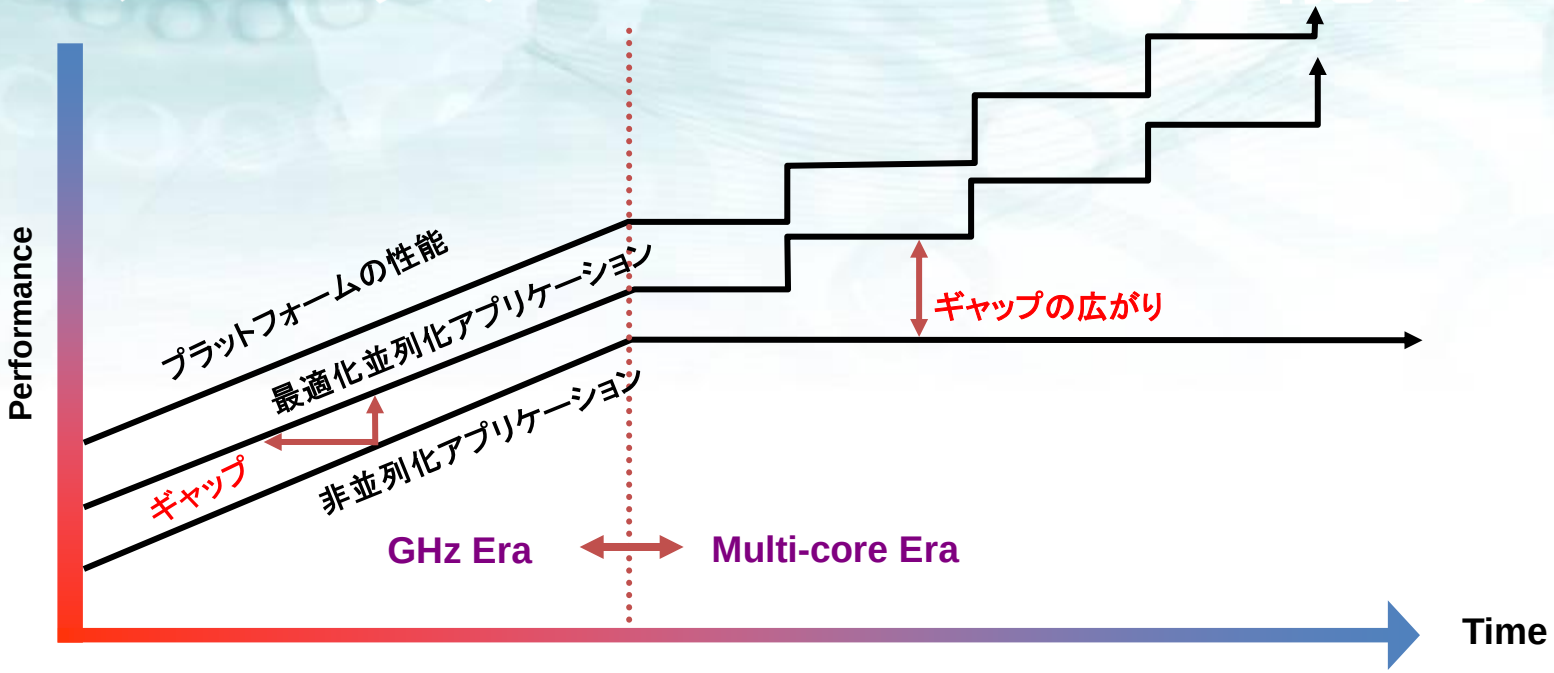
Intelマイクロアーキテクチャ
（Westmere）
32nm製造プロセス
6コア
SSE 4.2




2008　2009　2010　＞2010

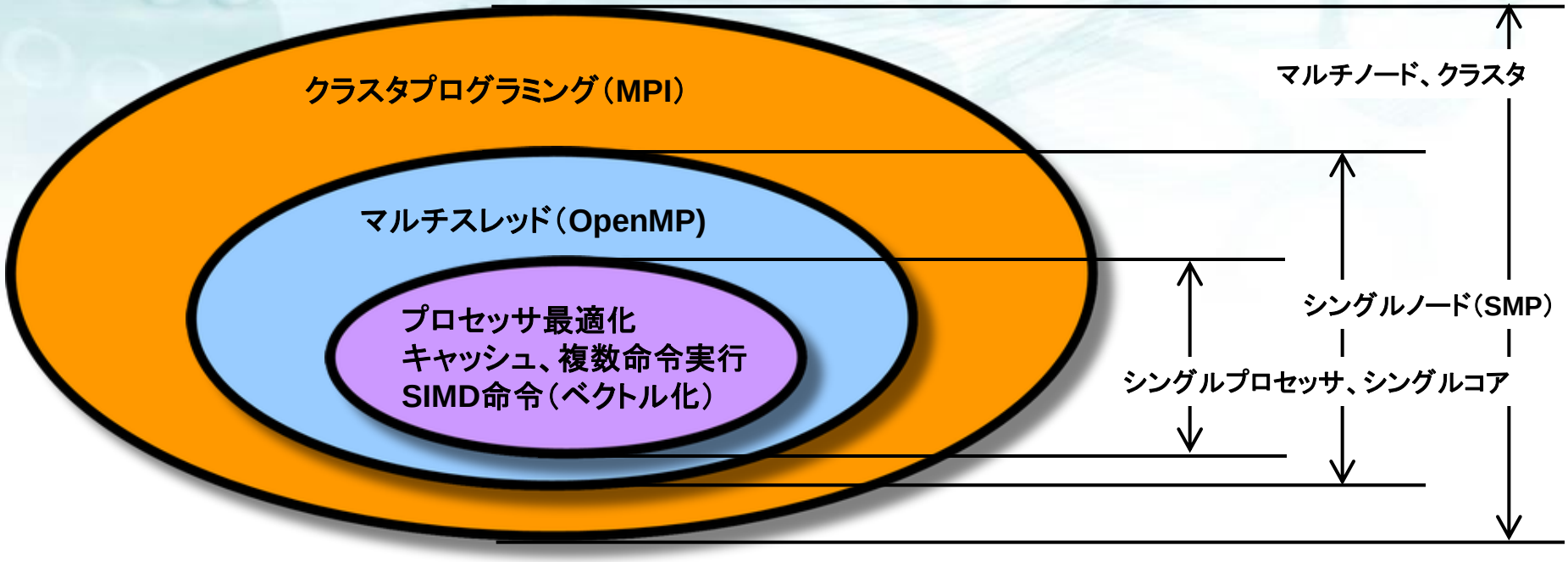
Intel AVX (Advanced Vector Extensions)

# アプリケーションの性能向上




- 並列処理は誰でも利用可能（利用のメリット）
  - より少ないコスト（低価格なシステム）でより効率の良い処理が可能（対費用効果）
  - より短時間でシュミレーションを完了（開発サイクルの短縮によるコスト削減）

# 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

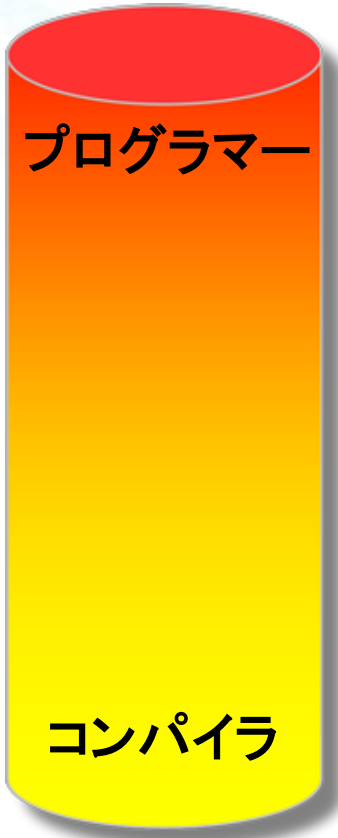
# プログラミング階層

```
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

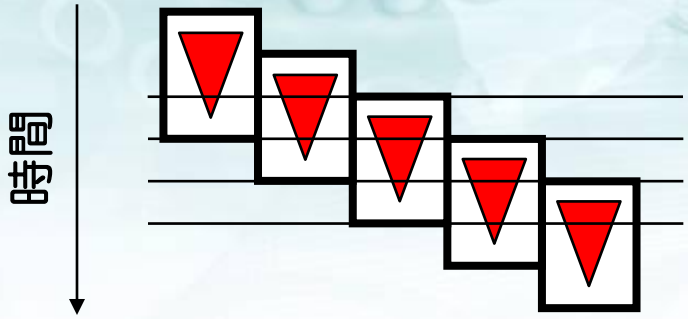
ノード内でのマルチスレッド並列化
OpenMPやスレッドプログラミング

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

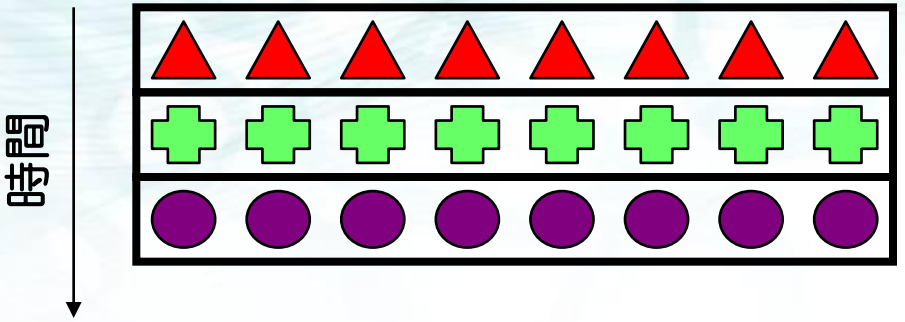



## 最適化と並列化の適用作業

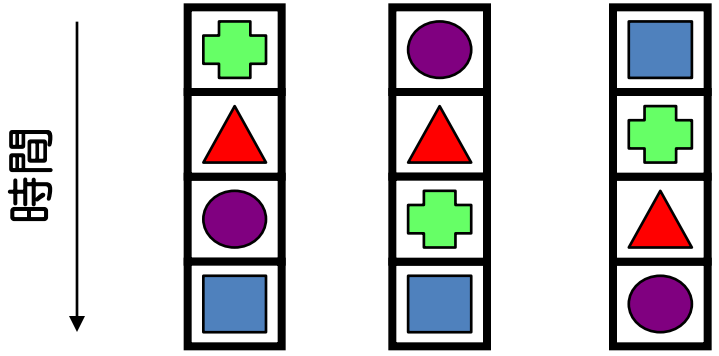
# 並列性（Parallelism）の利用



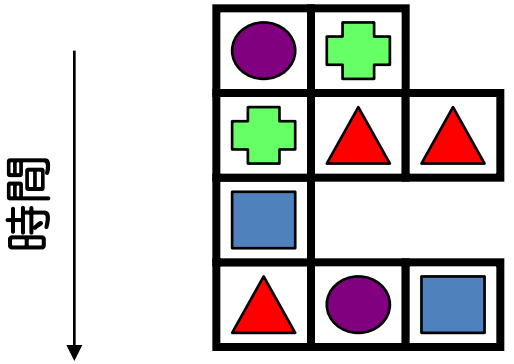

パイプライン処理




データレベル並列処理 (DLP)



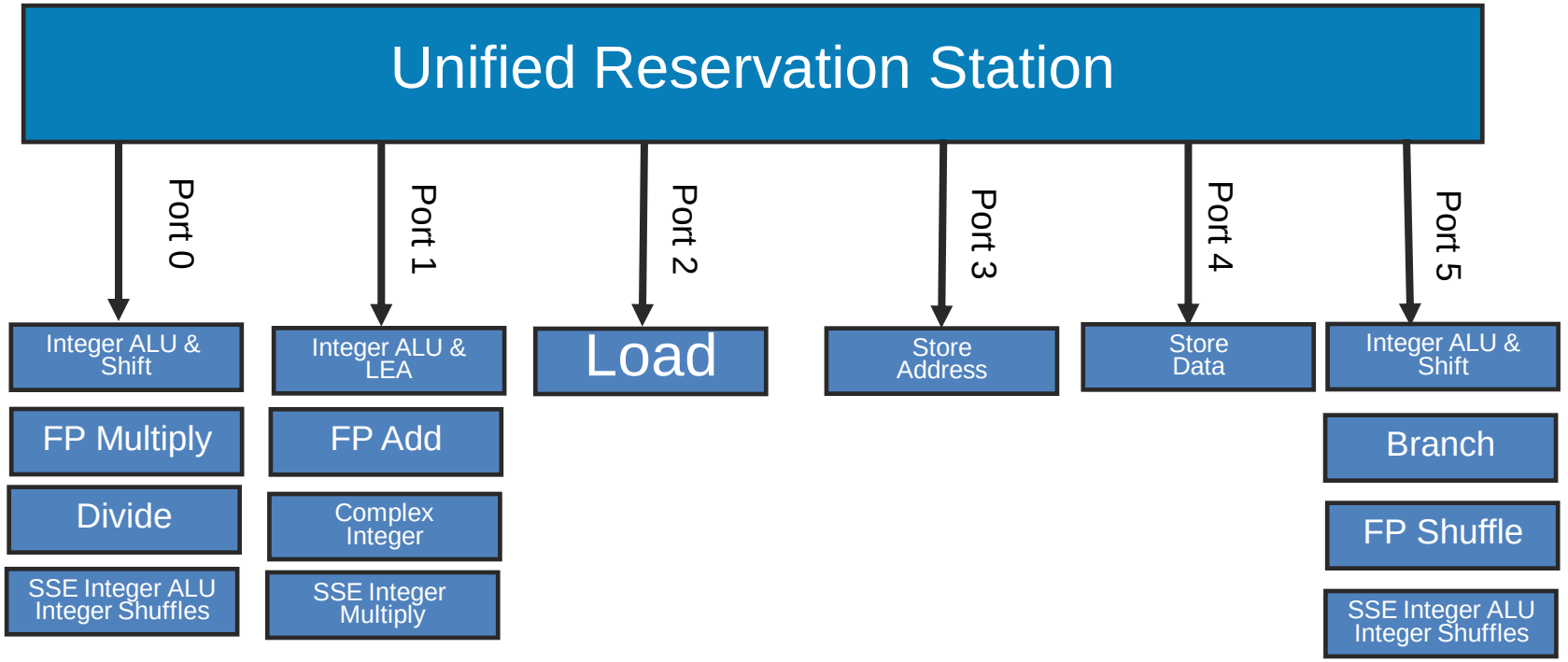

スレッドレベル並列処理 (TLP)




命令レベル並列処理 (ILP)

# 命令実行ユニット
## Coreマイクロアーキテクチャ

- 実行ユニットでのオペレーションのスケジューリング
- 全実行ユニットに対するシングルスケジューラ
- 1サイクルで6命令の実行が可能
- 3つのメモリオペレーション（1 Load/1 Store Address/1 Store Data)
- 3つの‘計算’オペレーション



# ループ内での命令実行の並列度の向上

## ループアンロール

```
N=1025
M=5
DO I=1,N
  DO J=1,M
      A(J,I) = B(J,I) + C(J,I) * D
  ENDDO
ENDDO
```

```
K = IMOD (N,4)
N=1025
M=5
DO I = 1,N
  DO J=1,M-K,4
    A(J,I) = B(J,I) + C(J,I) * D
    A(J+1,I) = B(J+1,I) + C(J+1,I) * D
    A(J+2,I) = B(J+2,I) + C(J+2,I) * D
    A(J+3,I) = B(J+3,I) + C(J+3,I) * D
  ENDDO
ENDDO
```

コンパイラによる並列実行のサポート

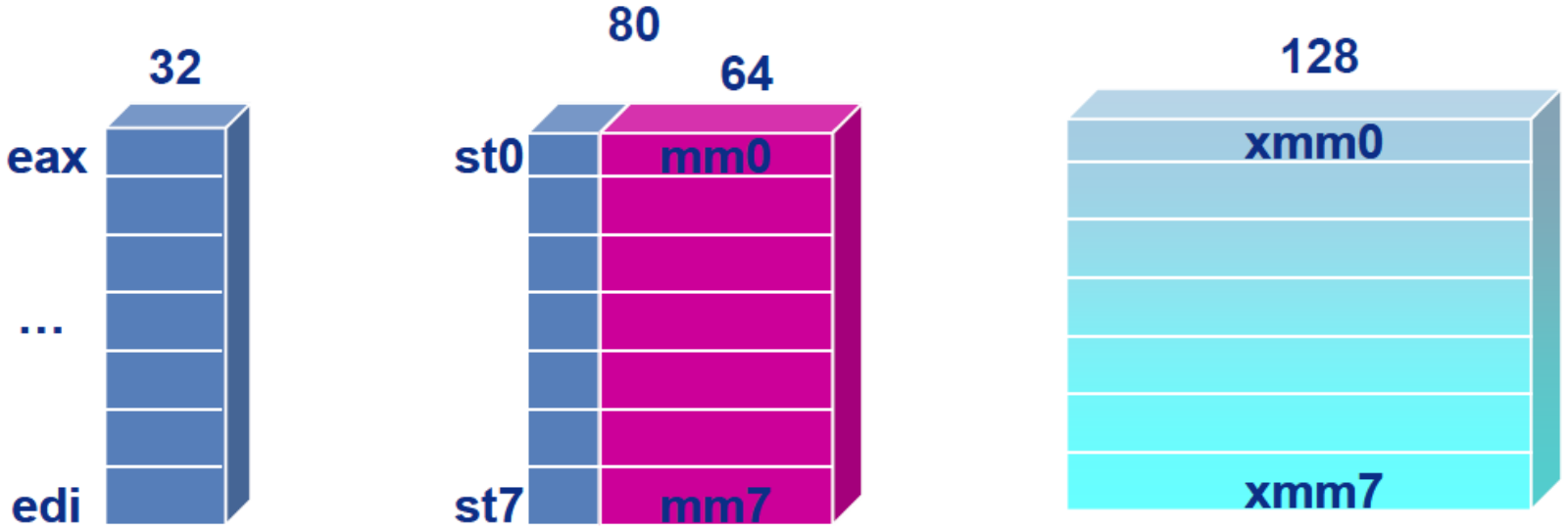
```
DO I = N
  DO J=1,M-K+1, M
    A(J,I) = B(J,I) + C(J,I) * D
  ENDDO
ENDDO
```

# 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

# MMX/SSEレジスタ構成

整数レジスタ　浮動小数点レジスタ　SSEレジスタ



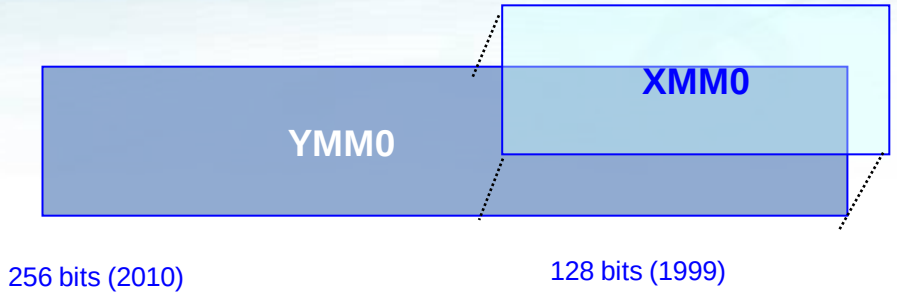

32bit レジスタ(14)
スカラデータ&アドレス

80/64bit レジスタ(8)
データのみ

128bit レジスタ(16)
4つの単精度浮動小数点データ
2つの倍精度浮動小数点データ
128ビット整数データ

# Intel AVX
## Advanced Vector Extensions

- Intel® AVX は16 XMMレジスタを256ビットに拡張




- Intel AVX
  - 従来のSSEを256ビットに拡張
  - 256-bits全てを利用しての演算（4つの倍精度浮動小数点演算など）
  - 下位 128-bits (既存のSSE命令との互換性）

# ループのベクトル化処理

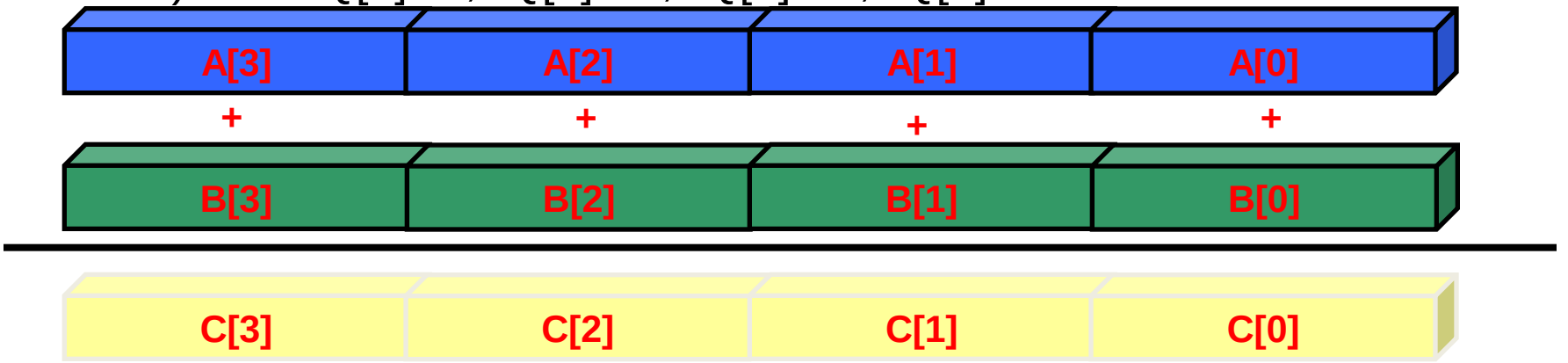
- プログラム例:

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

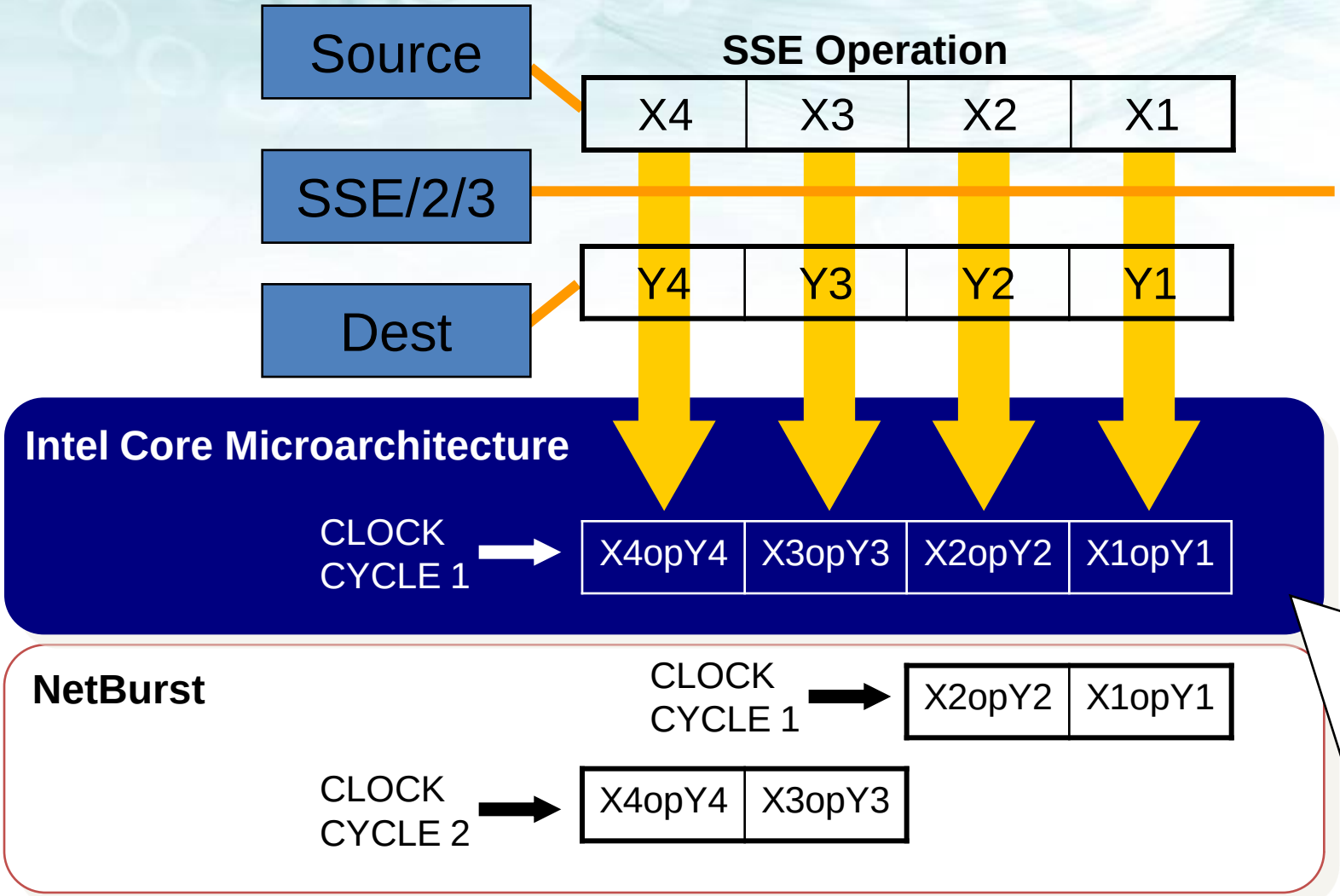
- 利用方法:

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

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

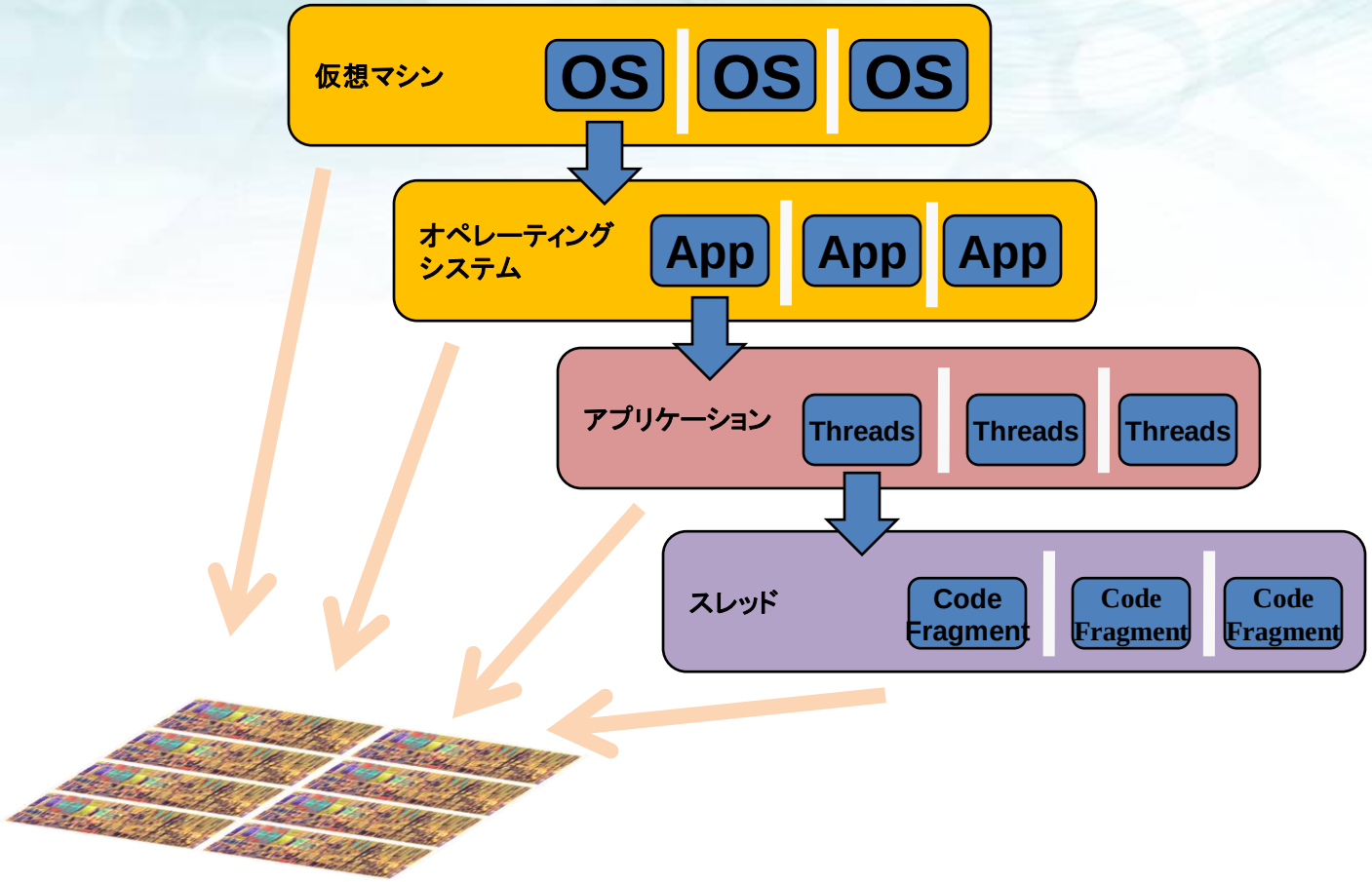


# インテルプロセッサでのSIMD処理



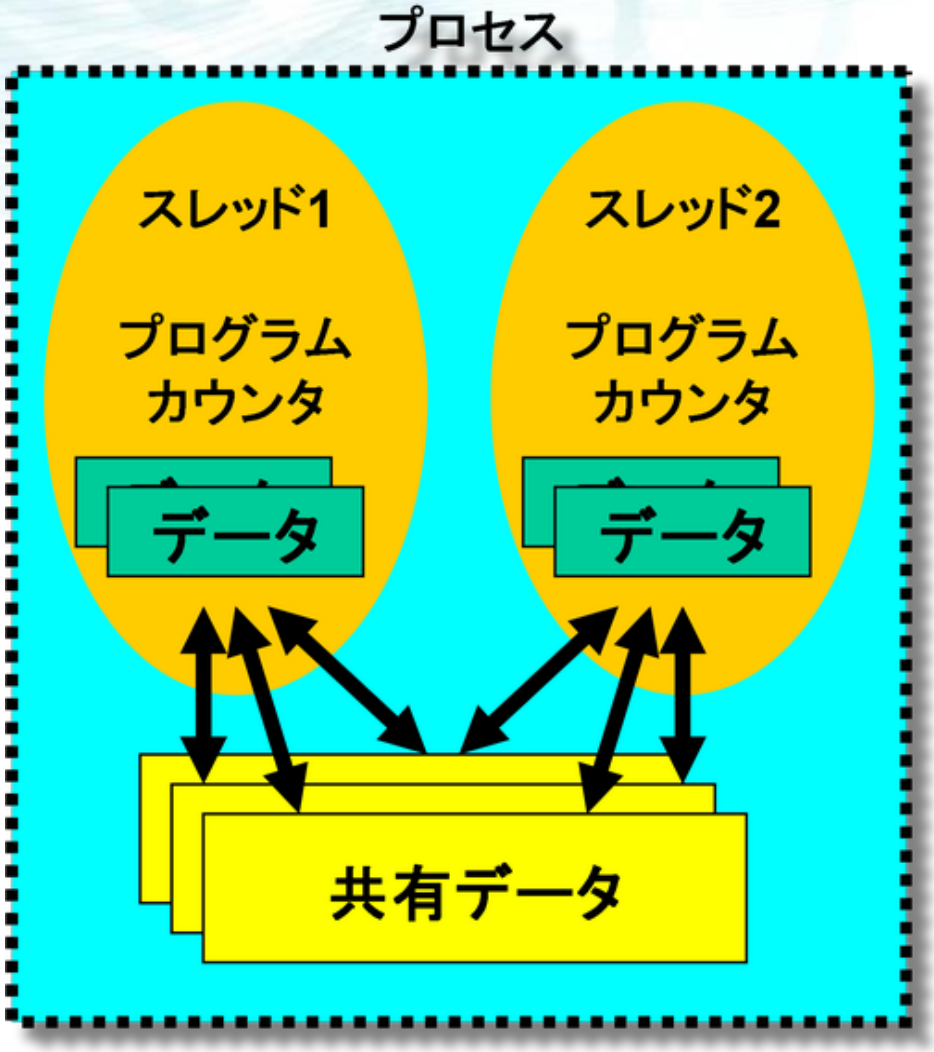

各MMX/SSE演算は、128ビットの演算をシングルサイクルで実行可能となる。MMX/SSE演算器は2セットあり、同時実行が可能となる。従って、単精度では、8浮動小数点演算、倍精度では、4浮動小数点演算を1クロックで実行することが出来る。

# アプリケーション実行階層

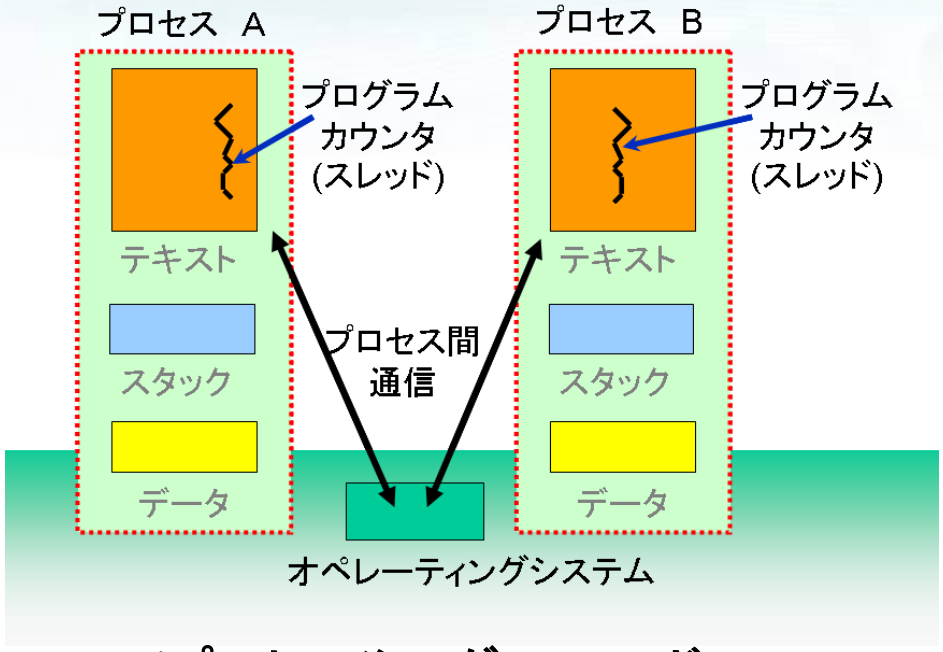


# プロセスとスレッドについて

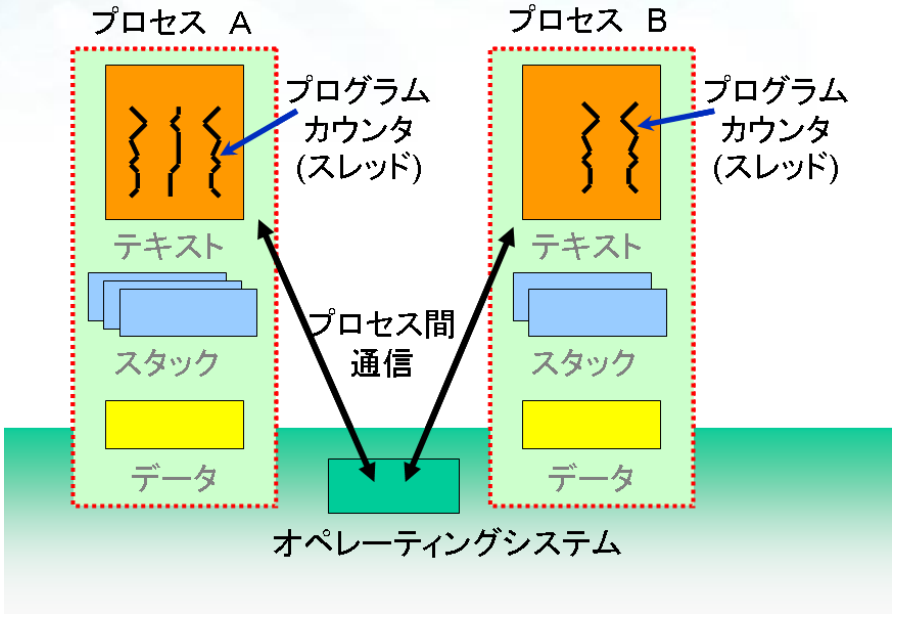
- プロセスは、独自のリソースを持って個々に実行
  - 個々に独立したアドレス空間
  - 実行命令、データ、ローカル変数スタック、ステート情報など
  - 個々のレジスタセット
- スレッドは、一つのプロセス内で実行
  - アドレス空間を共有
  - レジスタセットも共有
  - 個々のスレッドの独自データ用のスタック領域を持つ



# 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

# 共有メモリプログラミング

- デスクトップとノートＰＣ
  - 2又は4コア・・・今後もコア数は増える？
- サーバ及びクラスタノード
  - 2、4、8、16コア
  - 32コア（8プロセッサ、4コア）

- 6コア/プロセッサの製品化

# OpenMP:プログラムモデル

- OpenMPは、コンパイラ指示行（directives/pragmas）による並列化
  - C 及び C++での指示行：

    #pragma omp construct [clause [clause]…]
  - Fortranでの指示行:

    C$OMP construct [clause [clause]…]

    !$OMP construct [clause [clause]…]

    *$OMP construct [clause [clause]…]
- インクルードファイル

  #include “omp.h”

# OpenMPプログラム

- OpenMPは、通常はループに対して、その並列実行を指示：
  - 計算負荷の高いループを見つけて
  - そのループを複数スレッドで同時に処理する

**Sequential Program**

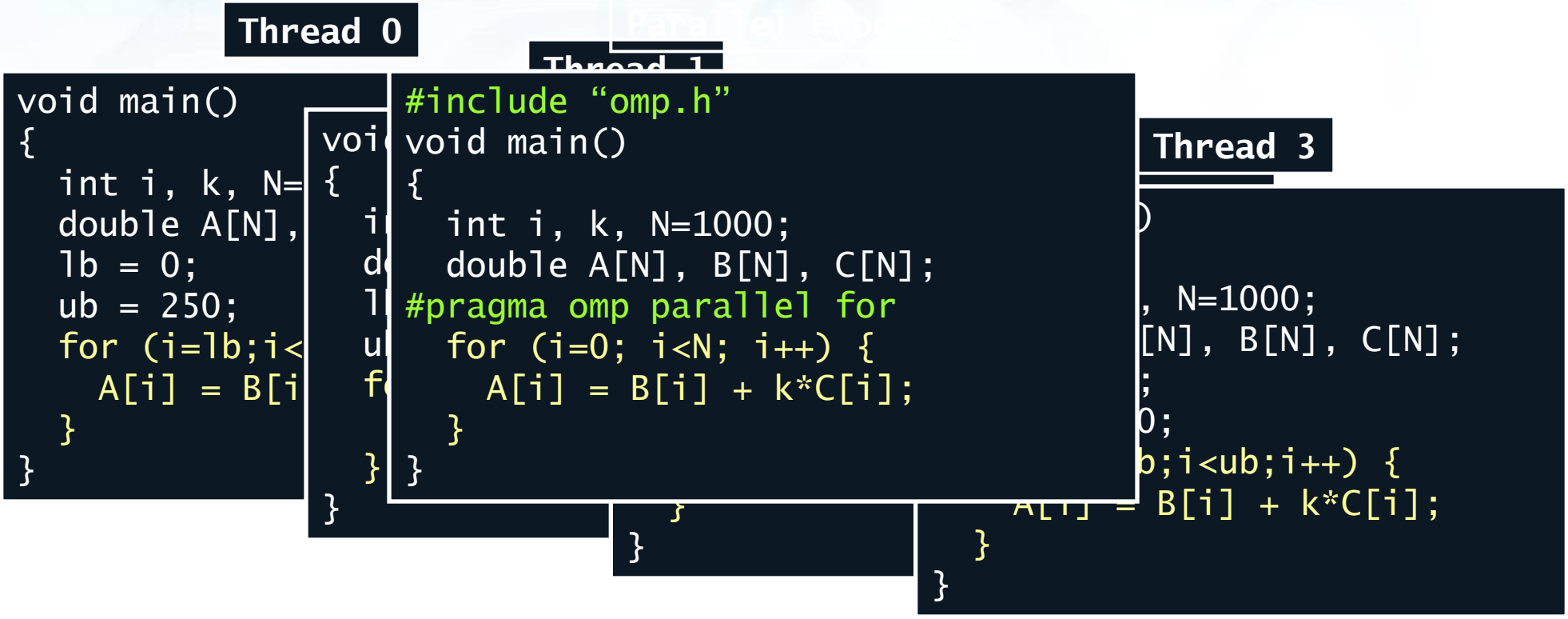
```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

**Parallel Program**

```
#include “omp.h”
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

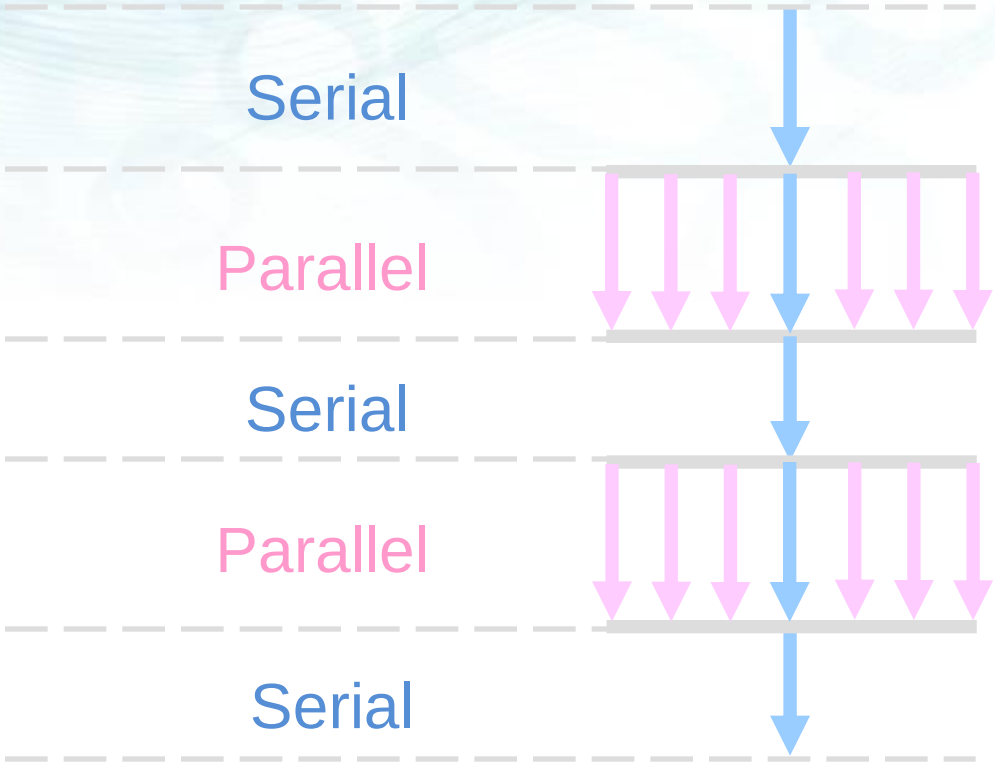
# OpenMPプログラムモデル

## Single Program Multiple Data (SPMD)



# OpenMP Fork-and-Join モデル

```
printf(“program begin¥n”);
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] – r[j];

printf(“program done¥n”);
```



# 並列プログラミングで留意点

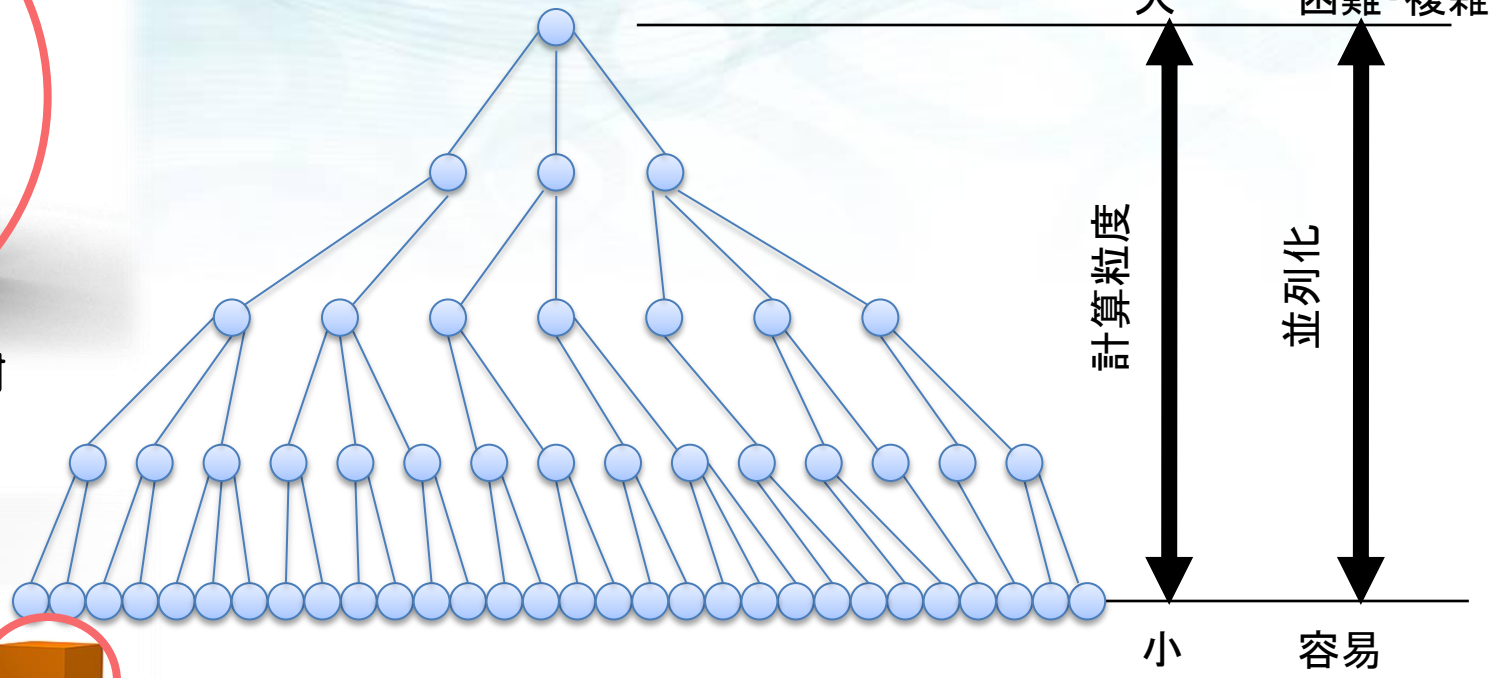
- 十分な計算量(Amdahl's Law)
- 計算粒度
- データの局所性
- ロードバランス
- 分散と同期処理
- 並列処理階層での最適化

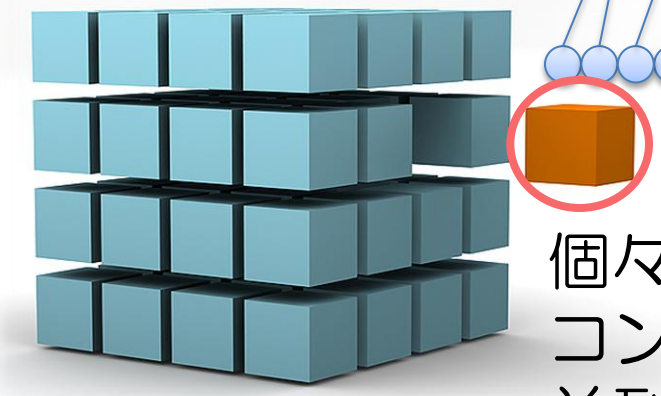
逐次処理(シングルスレッド)アプリケーションと比較しても検討課題が多いことが、並列処理をより困難にしています。

# よりハイレベルでの並列化

全体処理の把握とその並列化の検討

より上位(領域、範囲、対象）での並列化



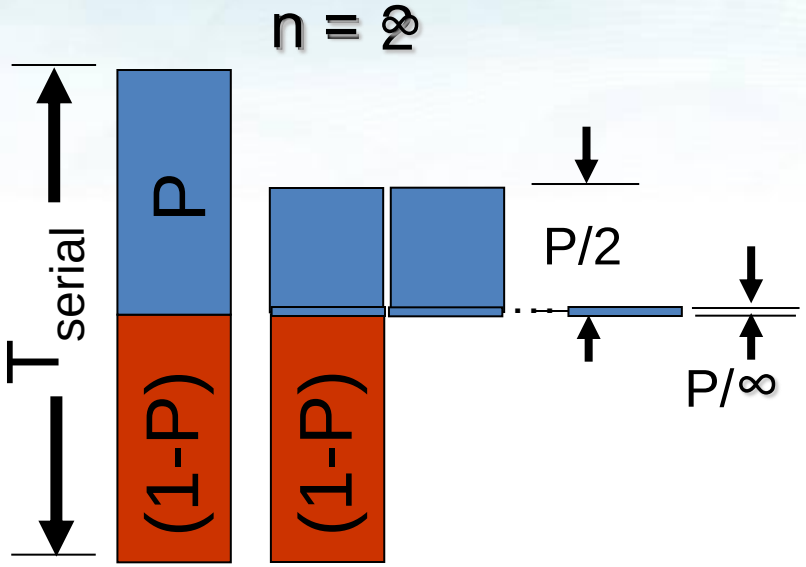



処理の末端での並列化

個々の処理の並列化の検討
コンパイラによる並列化（ベクトル化や自動並列化）は一般にはこのレベルでの並列化

# Amdahl's Law（アムダールの法則）

- 並列処理での性能向上の上限値（スケーリング）




$$T_{parallel} = \{(1-P) + P/n\}\, T_{serial}$$

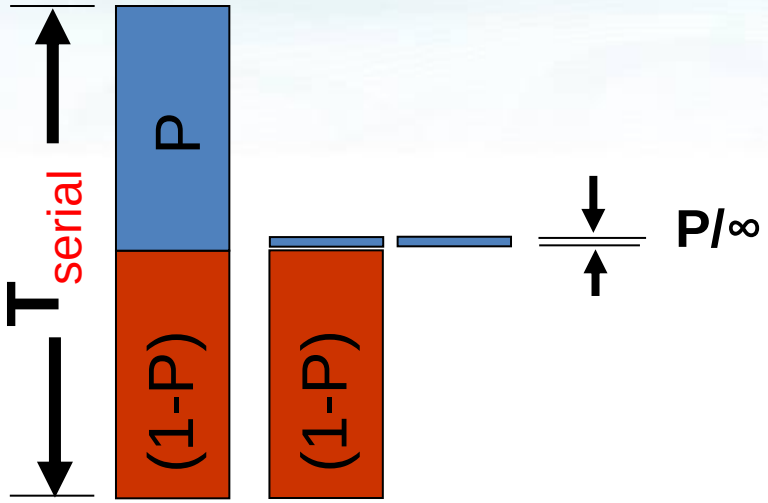
n = number of processors

$$Speedup = T_{serial} / T_{parallel}$$

プログラムの逐次処理部分（非並列処理）部分の排除が必要

# Amdahl's Law（アムダールの法則）

- 並列処理での性能向上の上限値（スケーリング）




$$T_{parallel} = \{(1-P) + P/n\} T_{serial} + O$$

n = number of processors

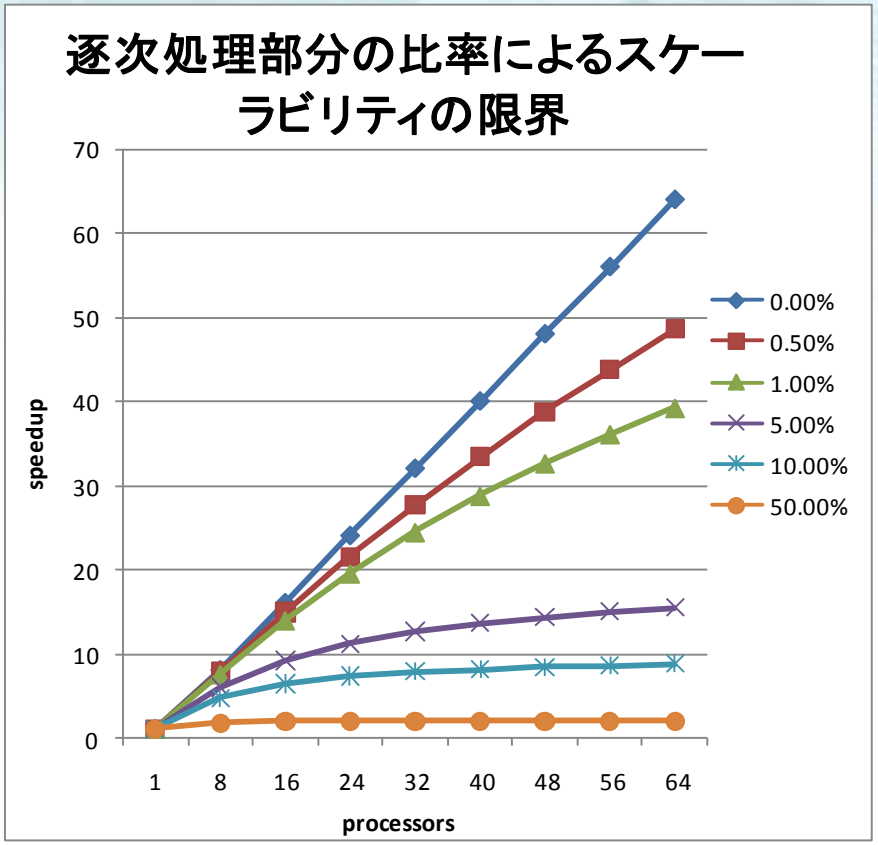
$$Scaling = T_{serial} / T_{parallel}$$

プログラムの逐次処理部分（非並列処理）部分の排除が必要

例えば、n=∞, P = 0.5 の場合

Scaling = 1.0 / (0.5 + 0) = 2.0

# Amdahl's Law（アムダールの法則）




- 実行時間 = 逐次処理 + 並列処理

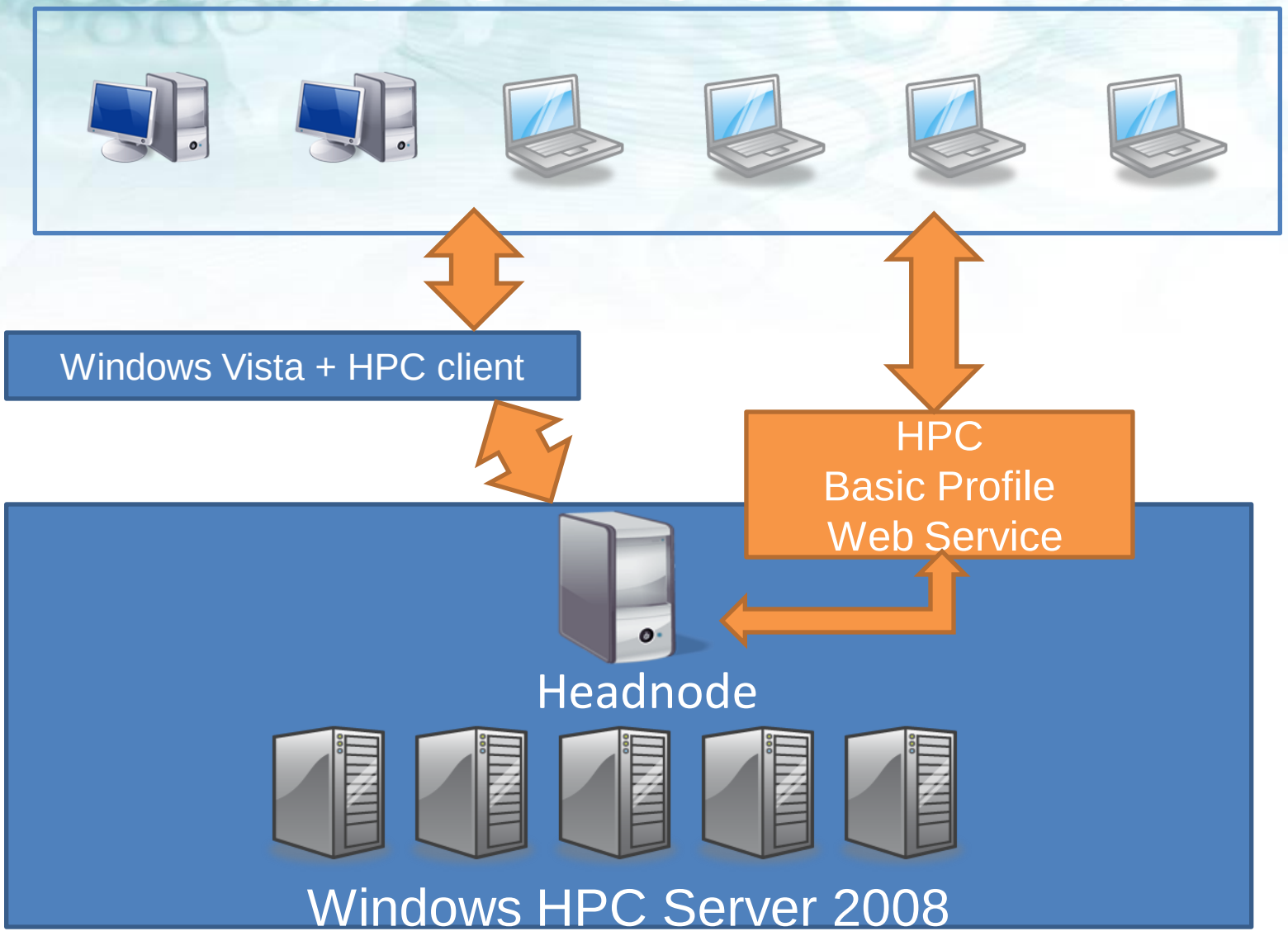
理論的な性能向上の限界

- 実行時間 = 逐次処理 + 並列処理時/P

- 64プロセッサで50倍の性能向上を得るには、逐次処理部分を0.5% 以下にする必要がある

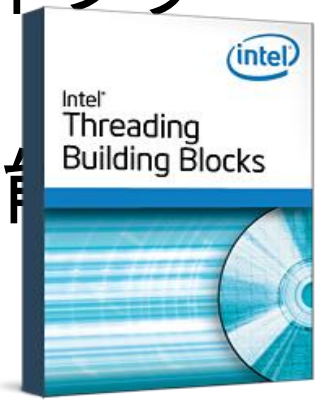
I/O処理は遂次処理の典型であり、I/O処理自身を並列に処理することが高いスケーラビリティの実現のためには必須である

# Windows HPC Server 2008



# Intel® Threading Building Blocks

- ヘッダーとランタイム・ライブラリーからなる
- タスクベースの並列化を表現する
  - スケーリングを持ち、将来における保守作業を軽減
  - 複数のスレッディング向けの置き換えライブラリー
- オープンソース版のヘッダーを入手可能
  - http://www.threadingbuildingblocks.org/



# Intel® Threading Building Blocks (C++ template library)

Generic Parallel Algorithms
parallel_for
parallel_reduce
pipeline
parallel_sort
parallel_while
parallel_scan

Concurrent Containers
concurrent_hash_map
concurrent_queue
concurrent_vector

Task Scheduler

Low-Level Synchronization Primitives
atomic
mutex
spin_mutex
queuing_mutex
spin_rw_mutex
queuing_rw_mutex

Memory Allocation
cache_aligned_allocator
scalable_allocator

Timing
tick_count

# サンプルプログラム

```
static void SerialUpdateVelocity() {
    for( int i=1; i<UniverseHeight-1; ++i )
#pragma ivdep
        for( int j=1; j<UniverseWidth-1; ++j )
            V[i][j] += (S[i][j] - S[i][j-1] + T[i][j] - T[i-
   1][j])*M[i];
}
```

# サンプルプログラム（並列版）

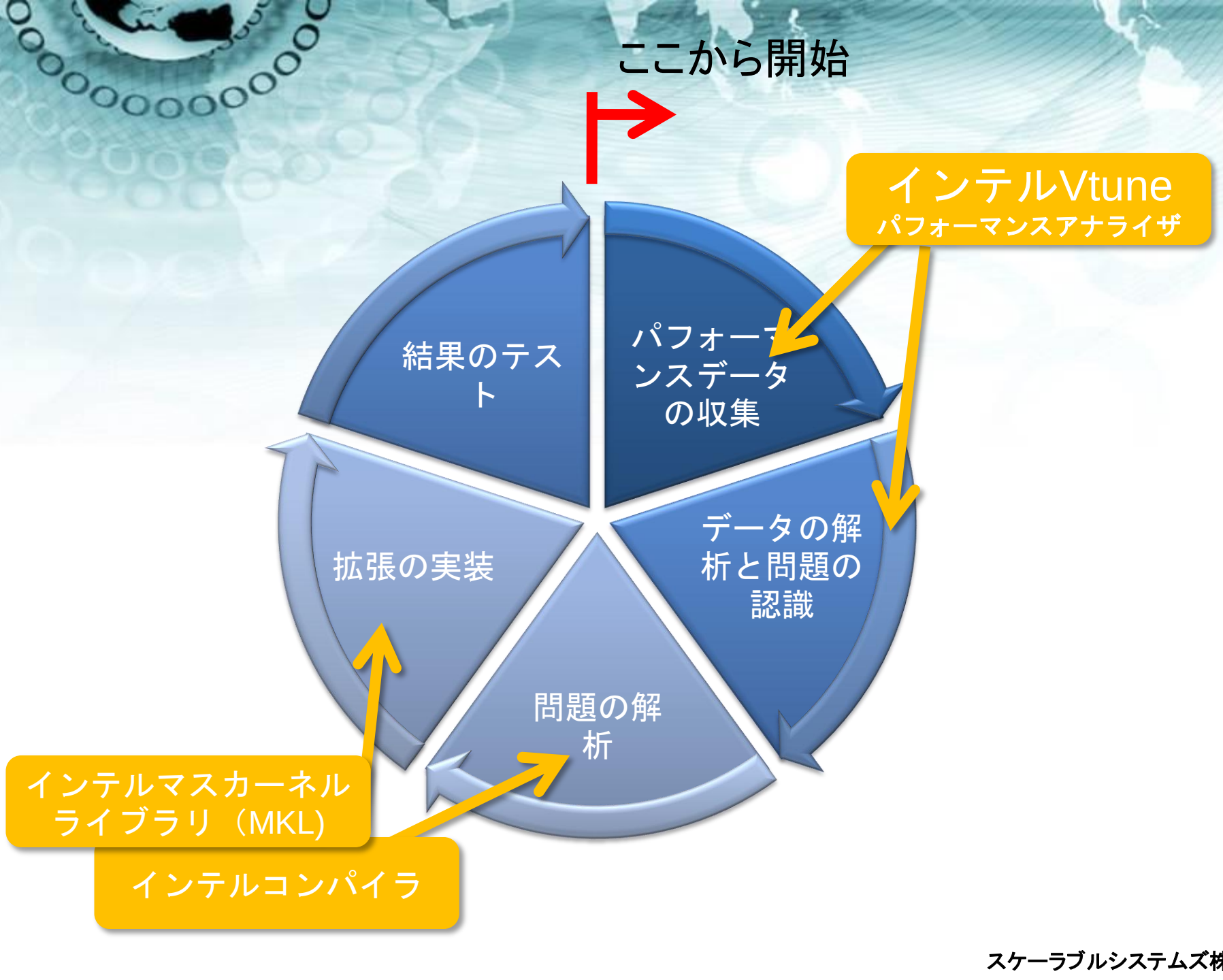
blue = original code
red = provided by TBB
black = boilerplate for library

```
struct UpdateVelocityBody {
    void operator()( const blocked_range<int>& range ) const {
        int end = range.end();
        for( int i= range.begin(); i<end; ++i ) {
#pragma ivdep
            for( int j=1; j<UniverseWidth-1; ++j )
                V[i][j] += (S[i][j] - S[i][j-1] + T[i][j] - T[i-1][j])*M[i];}
    }
};

void ParallelUpdateVelocity() {
    parallel_for( blocked_range<int>( 1, UniverseHeight-1, GrainSize ),
                UpdateVelocityBody() );
}
```

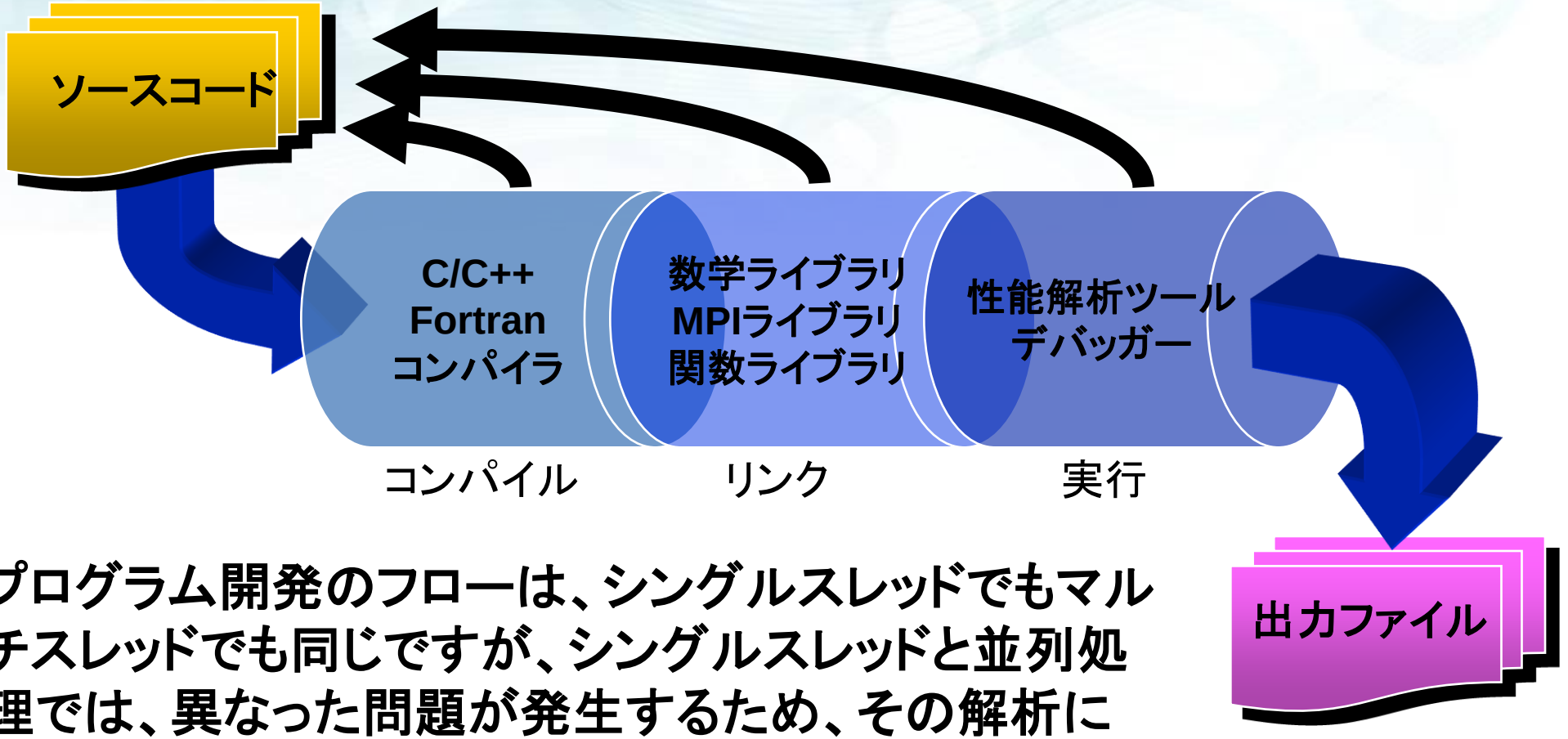
Task

Establishes grain size

Pattern


ここから開始
インテルVtune
パフォーマンスアナライザ
結果のテスト
パフォーマンスデータの収集
データの解析と問題の認識
問題の解析
拡張の実装
インテルマスカーネルライブラリ（MKL)
インテルコンパイラ

# 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。

# パフォーマンスカウンター

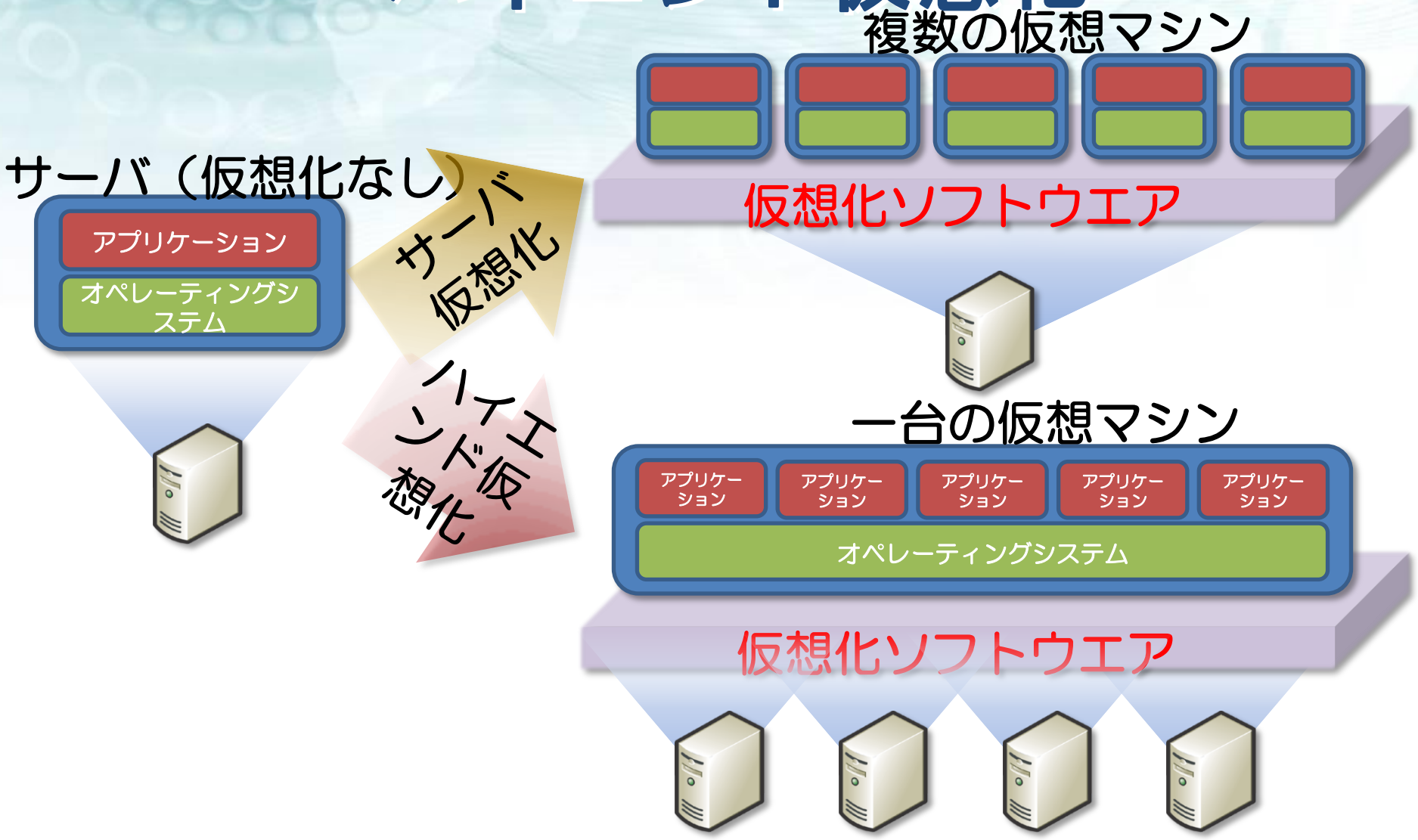
- Vtuneでは、デフォルトで2つのカウンターの結果を出力
  - CPU_CLK_UNHALTED.CORE カウントされたクロック数
  - INST_RETIRED.ANY 実行が完了した命令数
- 解析結果についての留意点
  - CPU_CLK_UNHALTED.CORE/ INST_RETIRED.ANYの数値は、１つの命令がその実行を完了するのに要した平均のクロック数になります。
  - インテルプロセッサ（Coreマイクロアーキテクチャの場合）では、複数の命令実行ポート（5つ）を持つため、この平均クロック数が1以下であれば、内部の実行ユニットが効率的に動作しています。一般には、この数値が1.5を超える場合には、実行時の問題を検討する必要があります。
  - 性能の数値化は、最適化・並列化の指針として重要です。

# ソフトウエアのギャップの解決

# プログラミングの生産性の向上

## スケーラブルSMPシステム

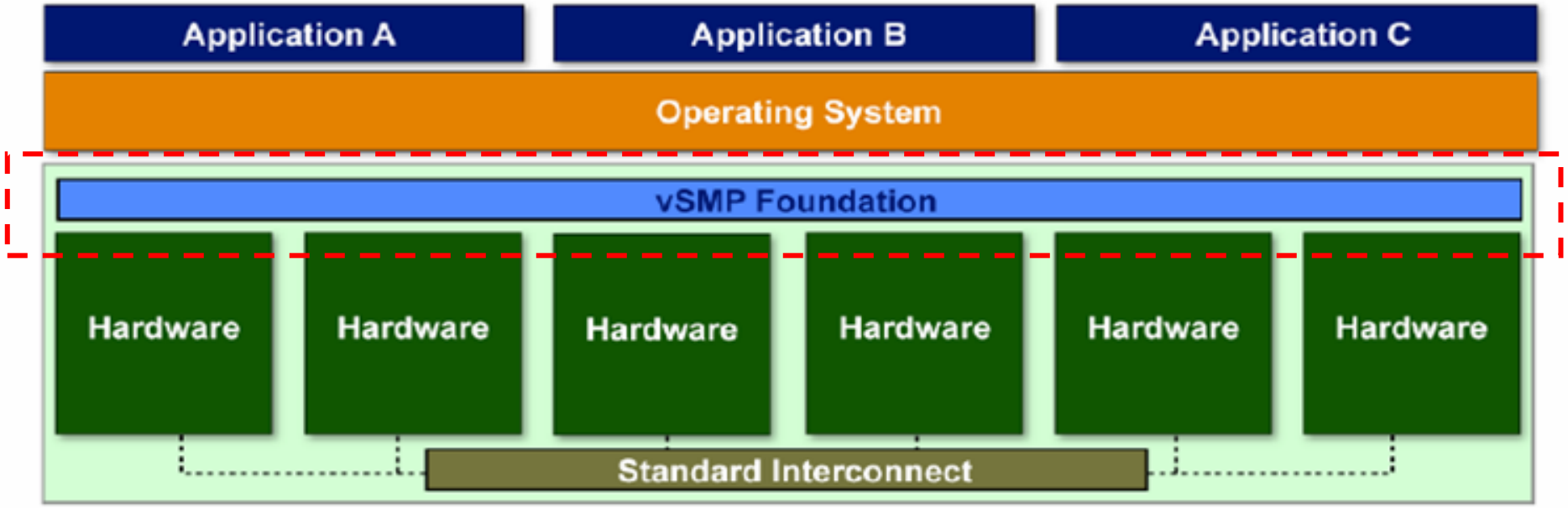
# ハイエンド仮想化


サーバ（仮想化なし）
アプリケーション
オペレーティングシステム
サーバ仮想化
ハイエンド仮想化
複数の仮想マシン
仮想化ソフトウエア
一台の仮想マシン
アプリケーション
アプリケーション
アプリケーション
アプリケーション
アプリケーション
オペレーティングシステム
仮想化ソフトウエア

# 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



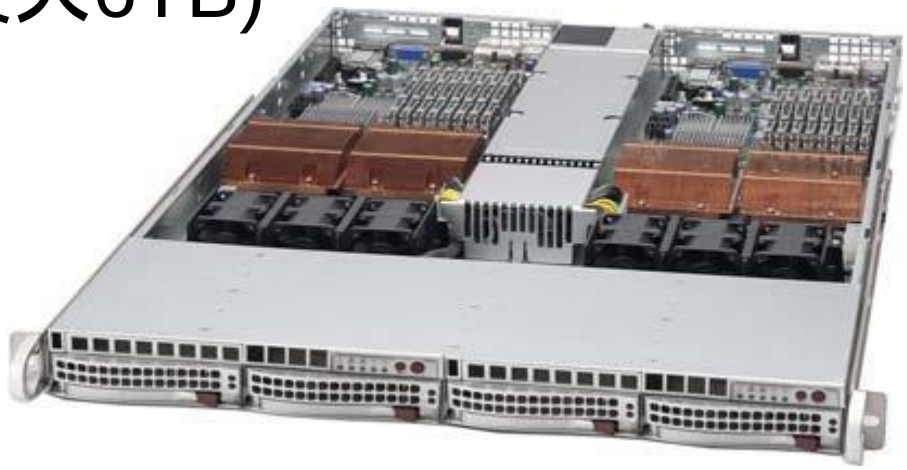

Hardwareは一般のx86チップセットと標準インターコネクトでシステムの構築が可能

vSMP Foundation でのシステムのSMP拡張を実現

# VXSMP 1440 システム

VXSMP 1440システムとは？

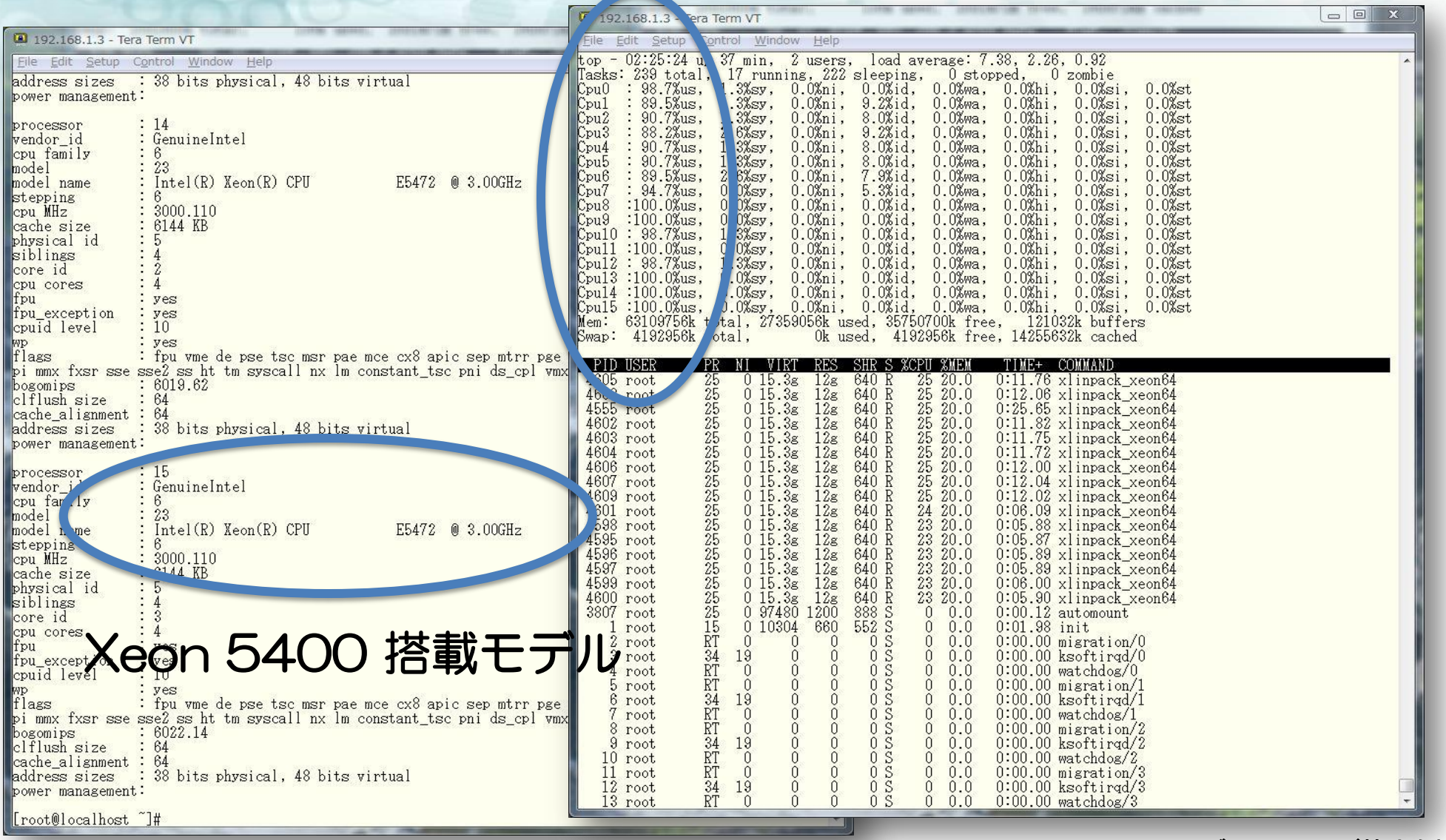
- VXPRO R1440ベースのvSMPシステム
- 1Uの筐体に4台のXeon 55xx/54xx/52xx を搭載し、最大16コアSMPを実現
- 96GBの共有メモリ空間（最大）
- 4台のHDDをOSが共有（最大6TB)



写真はXeon 5400 搭載モデル

# VXSMP1400

16プロセッサコア
＝2ノードx2ソケットx4コア




Xeon 5400 搭載モデル

スケーラブルシステムズ株式会社

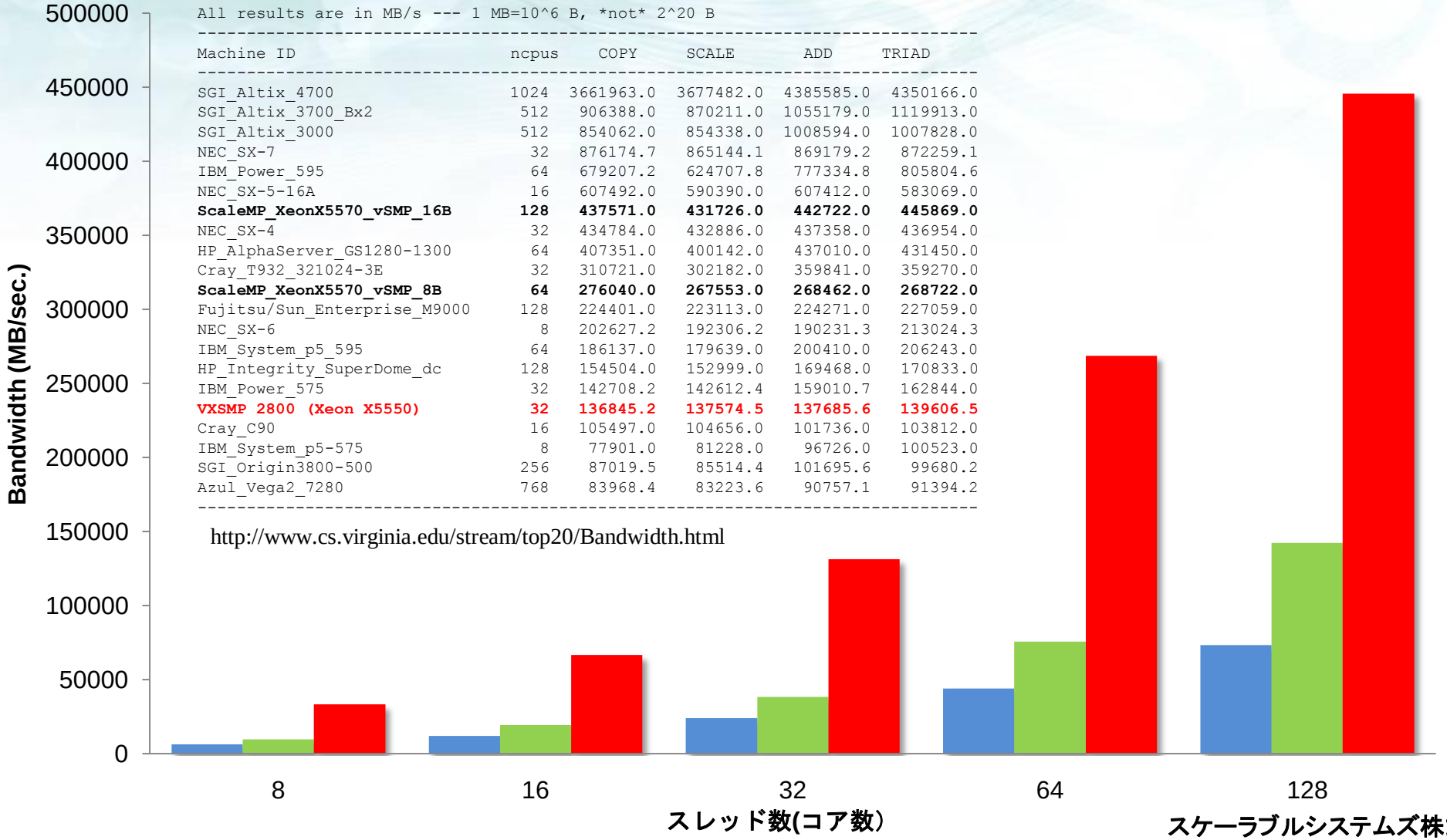
# Stream (OMP)ベンチマーク

■1333MHz FSB(128cores/16 boards) ■1600MHz FSB(128cores/16 boards) ■6.4GT/s QPI(128cores/16 boards)

All results are in MB/s --- 1 MB=10^6 B, *not* 2^20 B

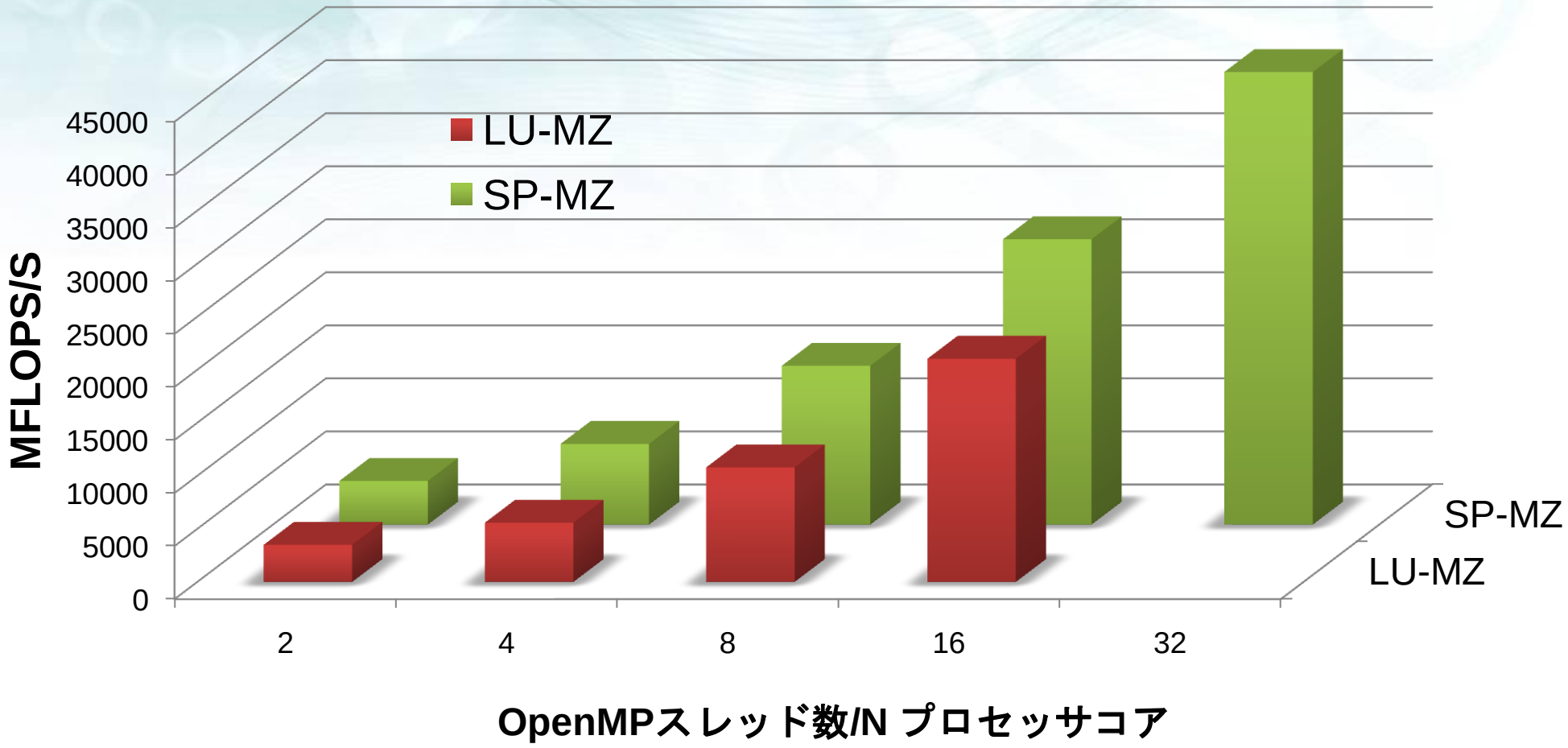
| Machine ID | ncpus | COPY | SCALE | ADD | TRIAD |
|---|---|---|---|---|---|
| SGI_Altix_4700 | 1024 | 3661963.0 | 3677482.0 | 4385585.0 | 4350166.0 |
| SGI_Altix_3700_Bx2 | 512 | 906388.0 | 870211.0 | 1055179.0 | 1119913.0 |
| SGI_Altix_3000 | 512 | 854062.0 | 854338.0 | 1008594.0 | 1007828.0 |
| NEC_SX-7 | 32 | 876174.7 | 865144.1 | 869179.2 | 872259.1 |
| IBM_Power_595 | 64 | 679207.2 | 624707.8 | 777334.8 | 805804.6 |
| NEC_SX-5-16A | 16 | 607492.0 | 590390.0 | 607412.0 | 583069.0 |
| **ScaleMP_XeonX5570_vSMP_16B** | **128** | **437571.0** | **431726.0** | **442722.0** | **445869.0** |
| NEC_SX-4 | 32 | 434784.0 | 432886.0 | 437358.0 | 436954.0 |
| HP_AlphaServer_GS1280-1300 | 64 | 407351.0 | 400142.0 | 437010.0 | 431450.0 |
| Cray_T932_321024-3E | 32 | 310721.0 | 302182.0 | 359841.0 | 359270.0 |
| **ScaleMP_XeonX5570_vSMP_8B** | **64** | **276040.0** | **267553.0** | **268462.0** | **268722.0** |
| Fujitsu/Sun_Enterprise_M9000 | 128 | 224401.0 | 223113.0 | 224271.0 | 227059.0 |
| NEC_SX-6 | 8 | 202627.2 | 192306.2 | 190231.3 | 213024.3 |
| IBM_System_p5_595 | 64 | 186137.0 | 179639.0 | 200410.0 | 206243.0 |
| HP_Integrity_SuperDome_dc | 128 | 154504.0 | 152999.0 | 169468.0 | 170833.0 |
| IBM_Power_575 | 32 | 142708.2 | 142612.4 | 159010.7 | 162844.0 |
| **VXSMP 2800 (Xeon X5550)** | **32** | **136845.2** | **137574.5** | **137685.6** | **139606.5** |
| Cray_C90 | 16 | 105497.0 | 104656.0 | 101736.0 | 103812.0 |
| IBM_System_p5-575 | 8 | 77901.0 | 81228.0 | 96726.0 | 100523.0 |
| SGI_Origin3800-500 | 256 | 87019.5 | 85514.4 | 101695.6 | 99680.2 |
| Azul_Vega2_7280 | 768 | 83968.4 | 83223.6 | 90757.1 | 91394.2 |

http://www.cs.virginia.edu/stream/top20/Bandwidth.html



# OpenMPベンチマーク

## NAS Parallel Benchmark (Multi-Zone)




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。

スケーラブルシステムズ株式会社

# OpenMP－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
```

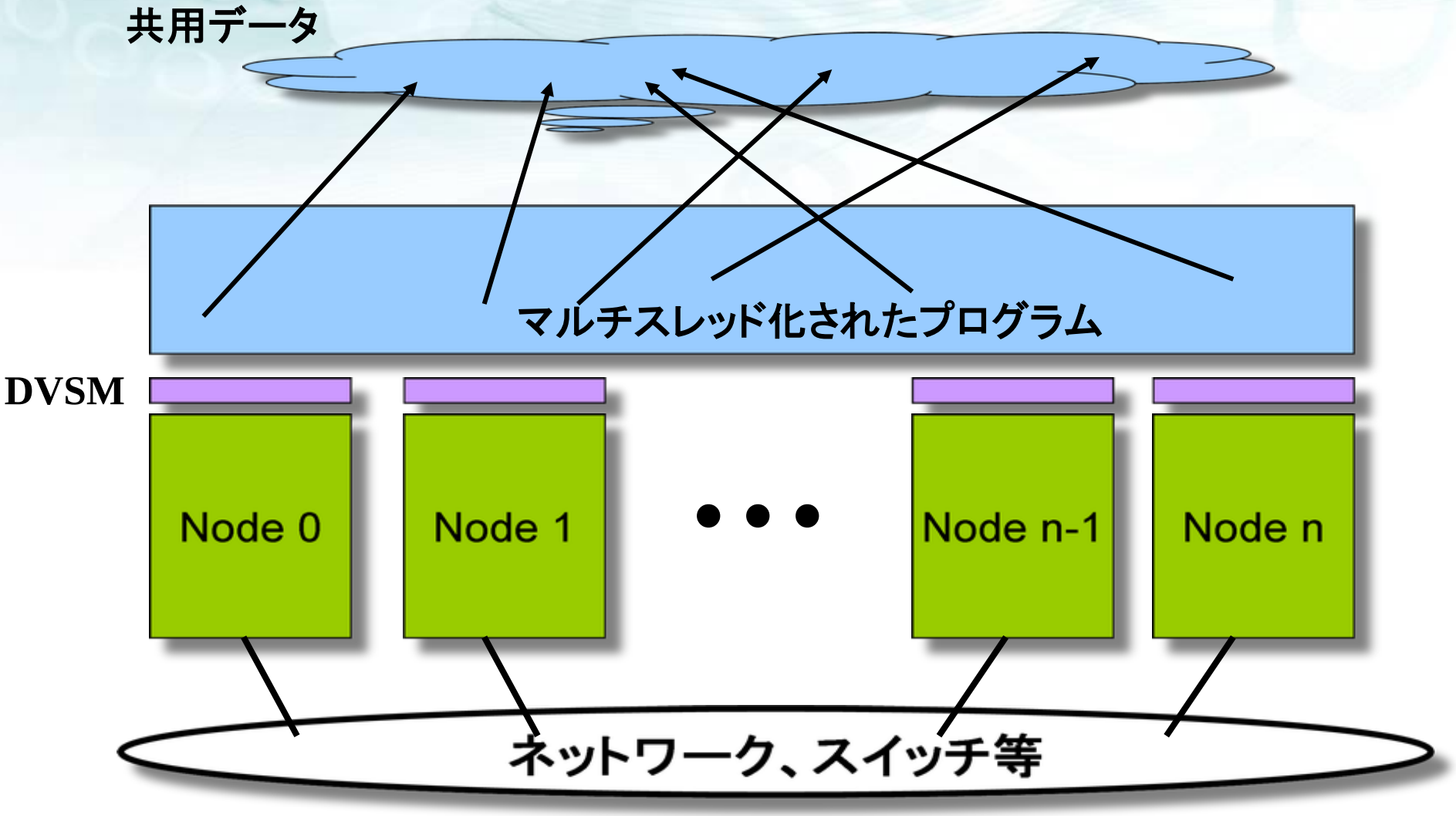
OpenMP指示行

OpenMP実行時関数

コンパイルとメッセージ

環境変数の設定

# インテルクラスタOpenMP



# Cluster OpenMP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
```

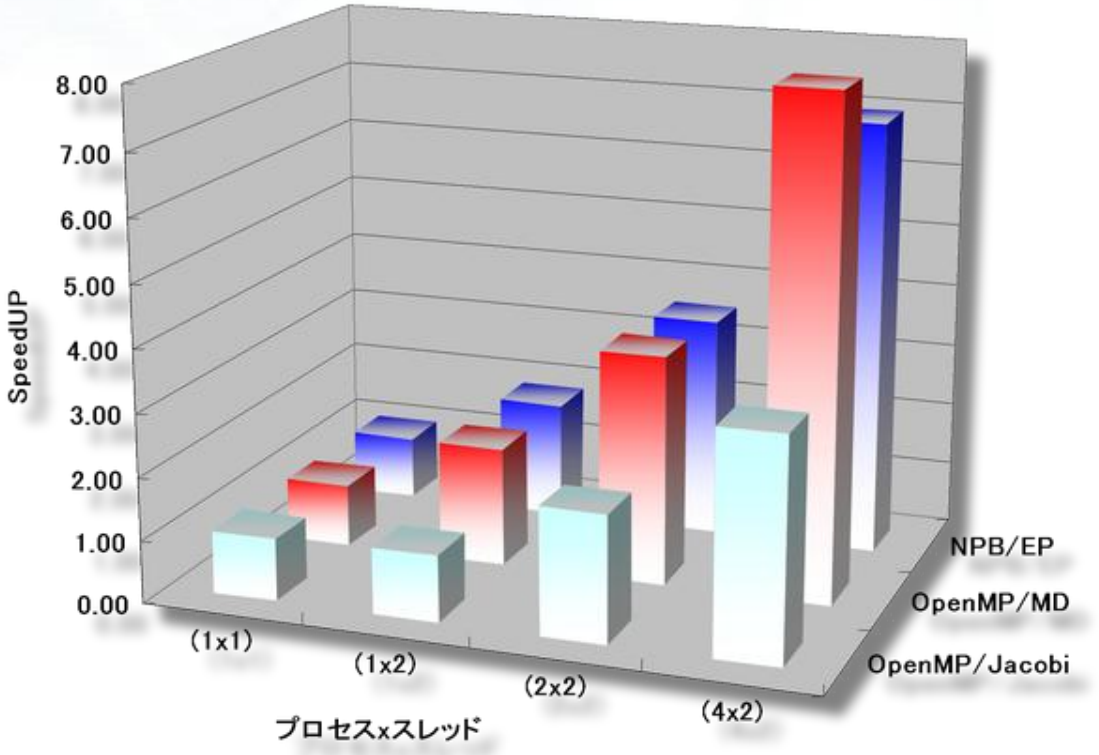
# Cluster OpenMPスケーラビリティ

ベンチマークシステム

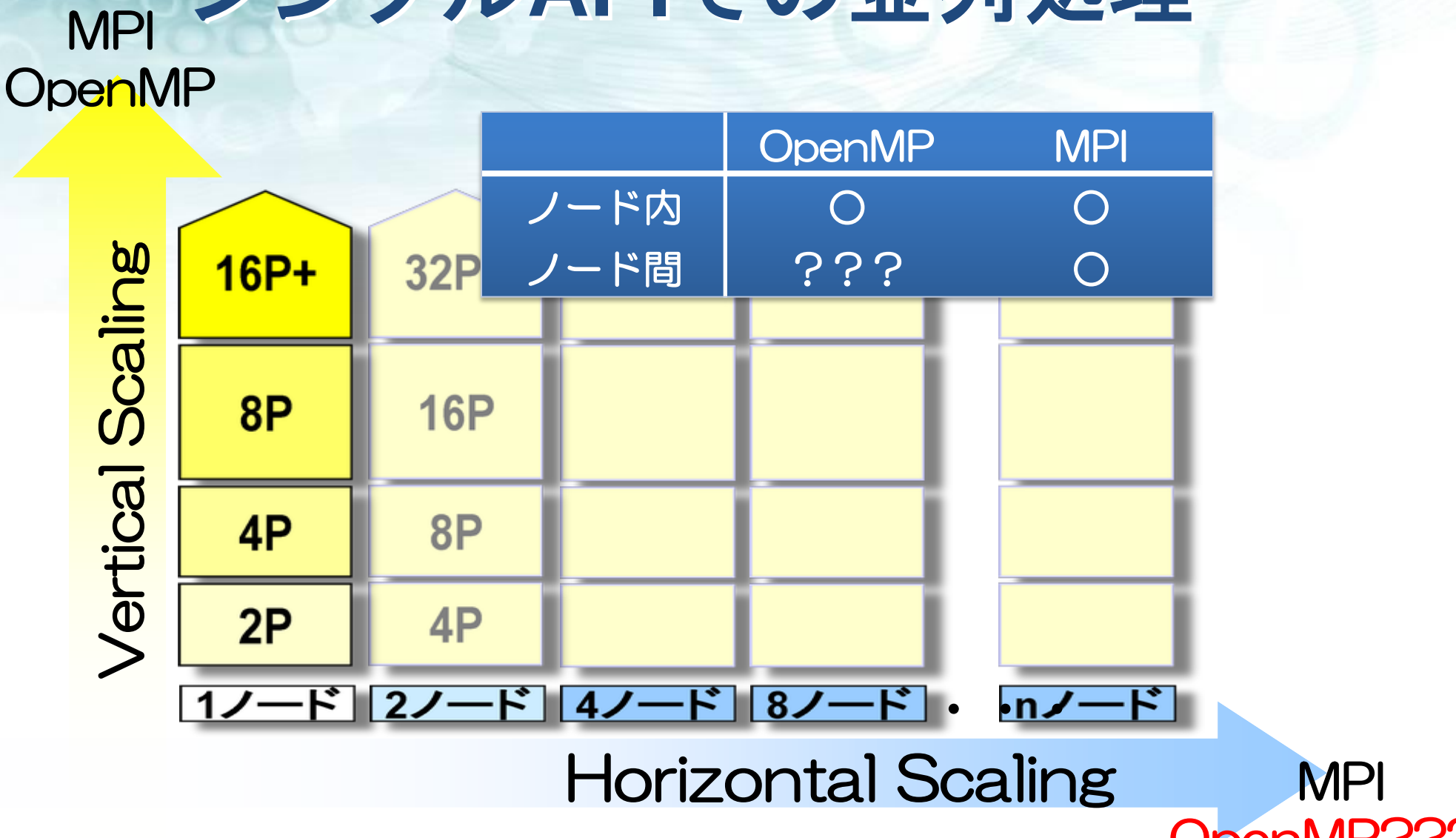
- NEXXUS 4820-PT
- 2.66GHz/1066MHz FSB/16GB Memory/InfiniBand

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）
- OpenMPサンプル(Jacobi法サンプル、5000x5000）



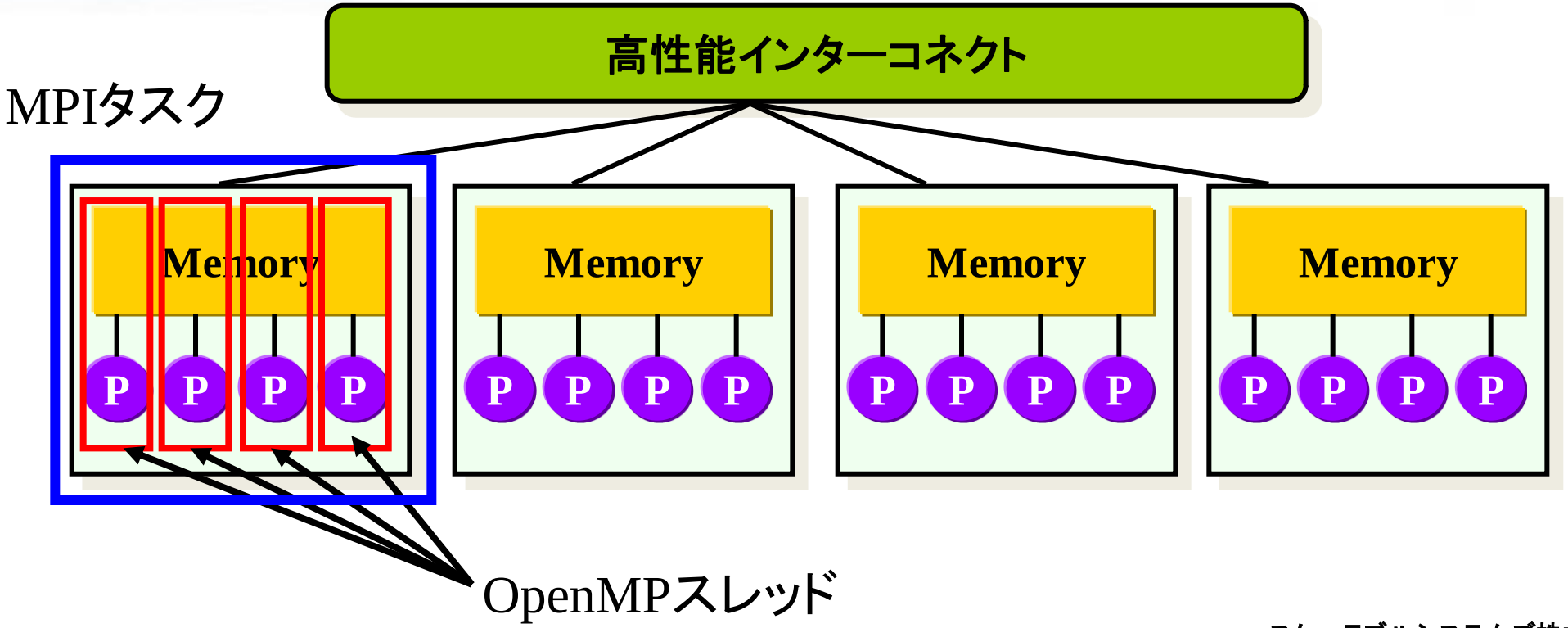
# シングルAPIでの並列処理




| | OpenMP | MPI |
|---|---|---|
| ノード内 | ○ | ○ |
| ノード間 | ??? | ○ |

# 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

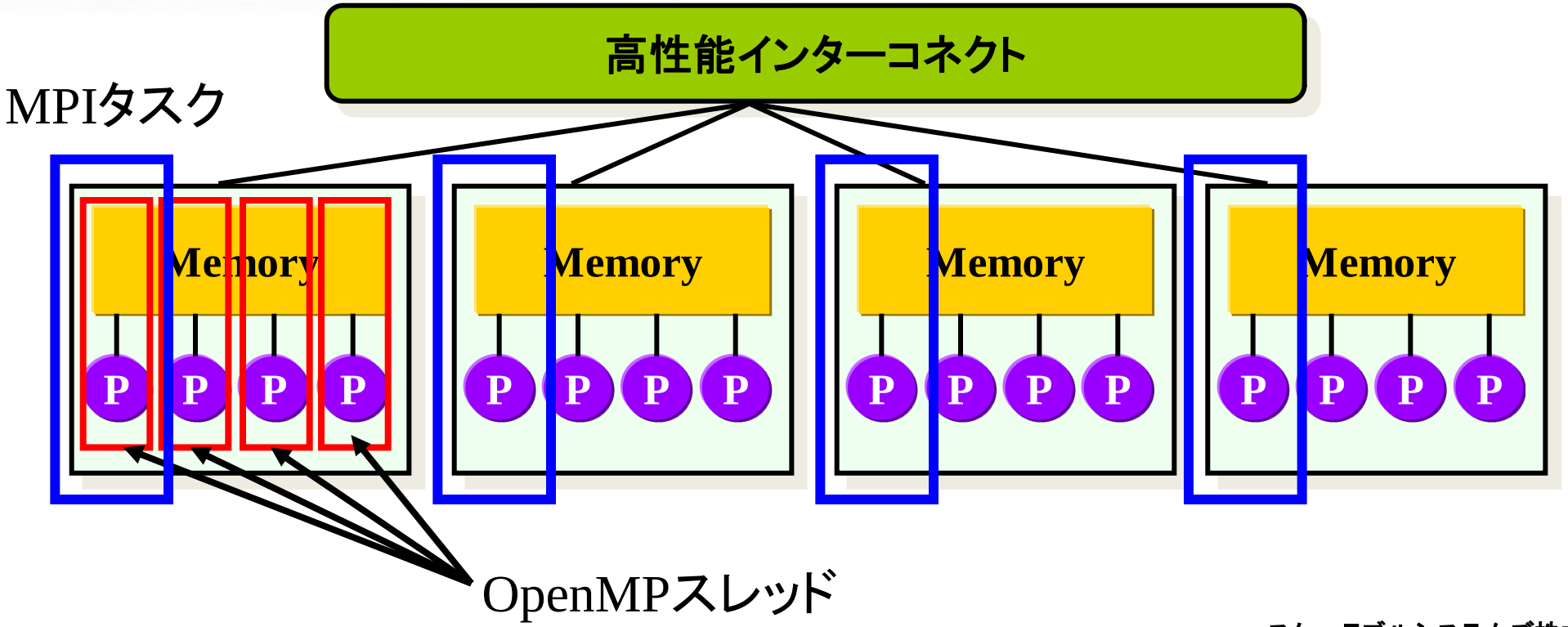
```
include ‘mpif.h’
program hybsimp


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
```

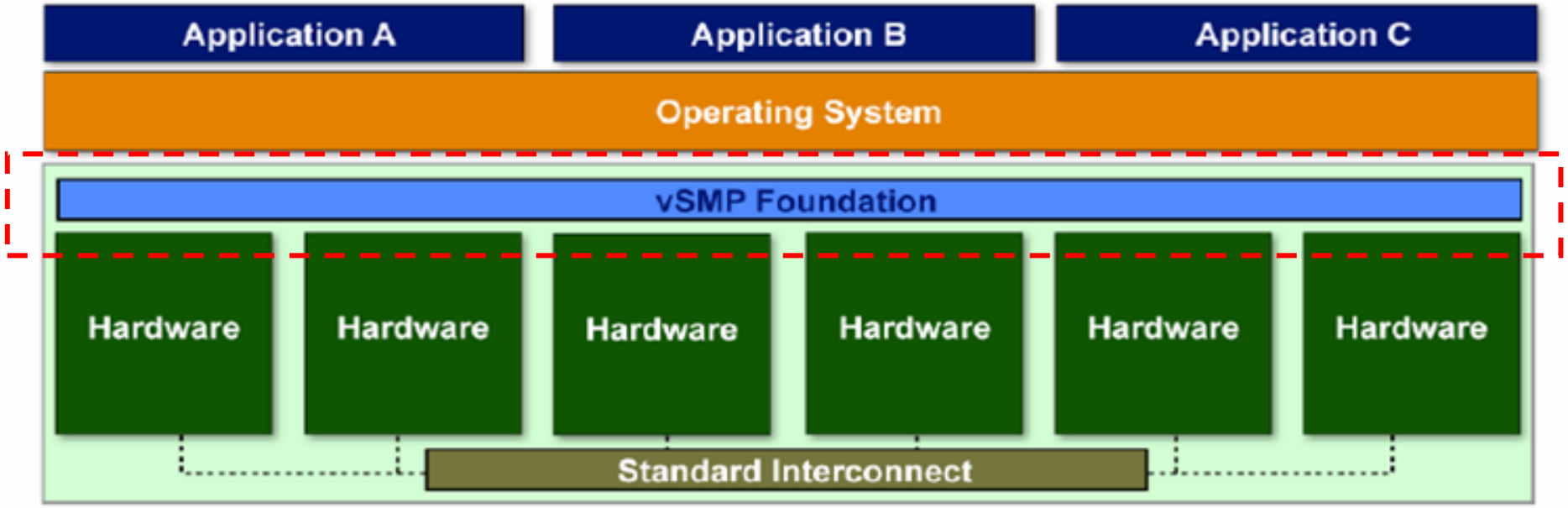
C/C++

```
#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



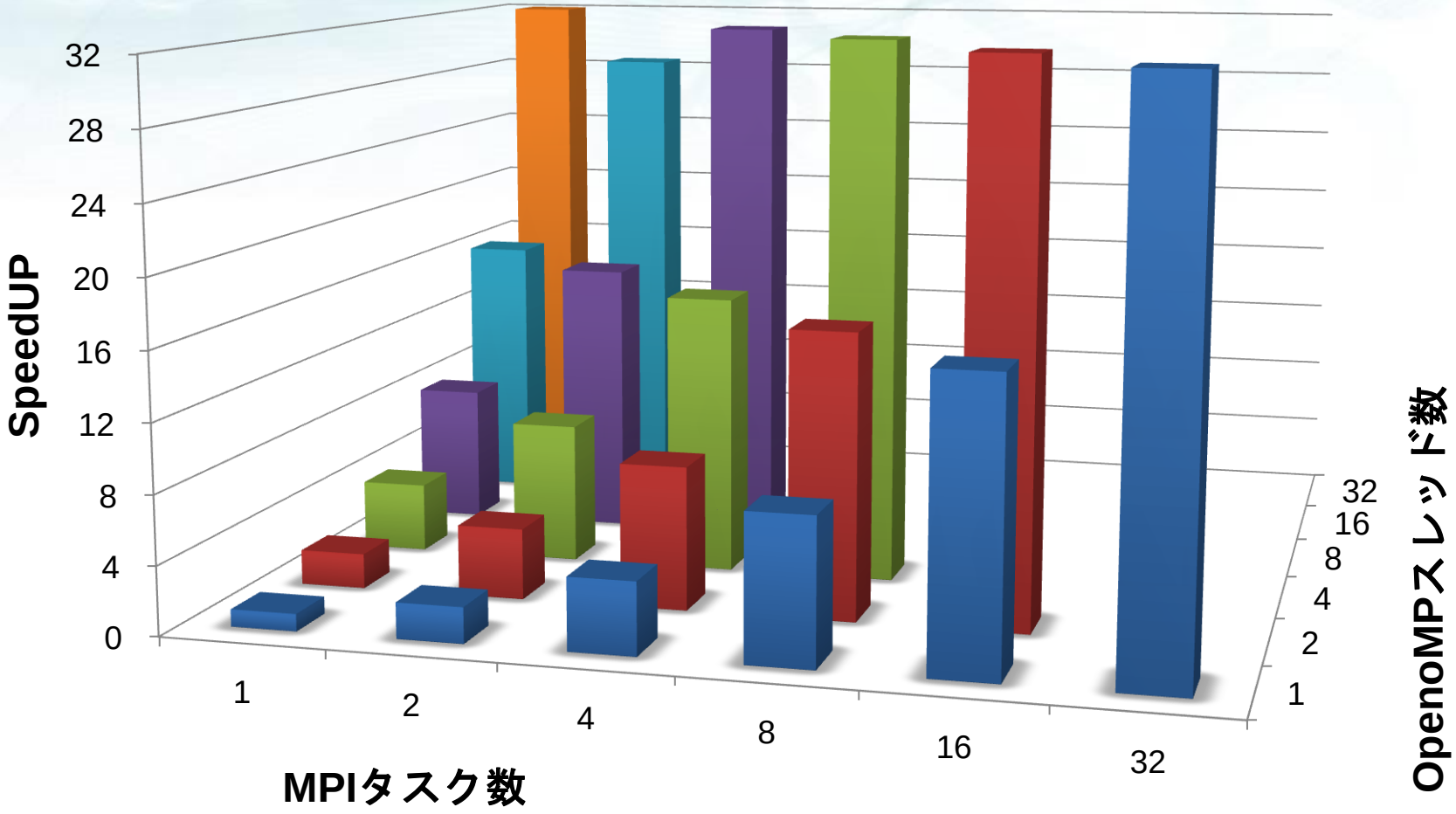

Hardwareは一般のx86チップセットと標準インターコネクトでシステムの構築が可能

vSMP Foundation でのシステムのSMP拡張を実現

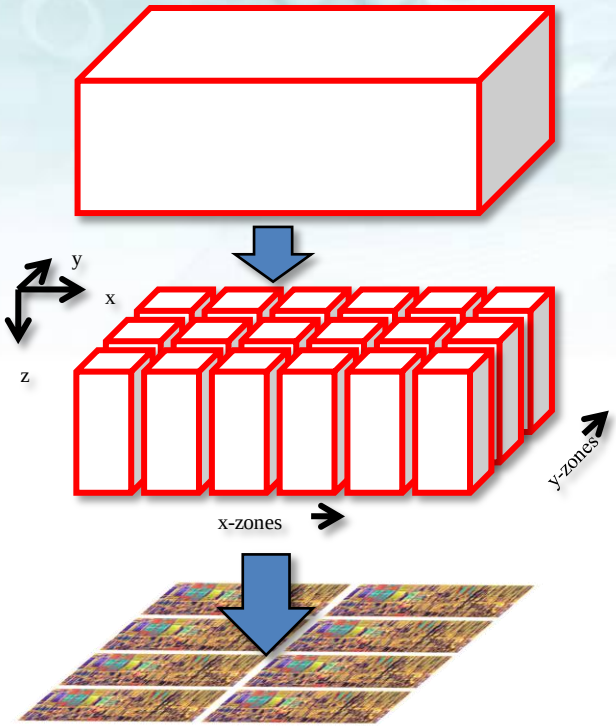
# OpenMP/MPI/ハイブリッド

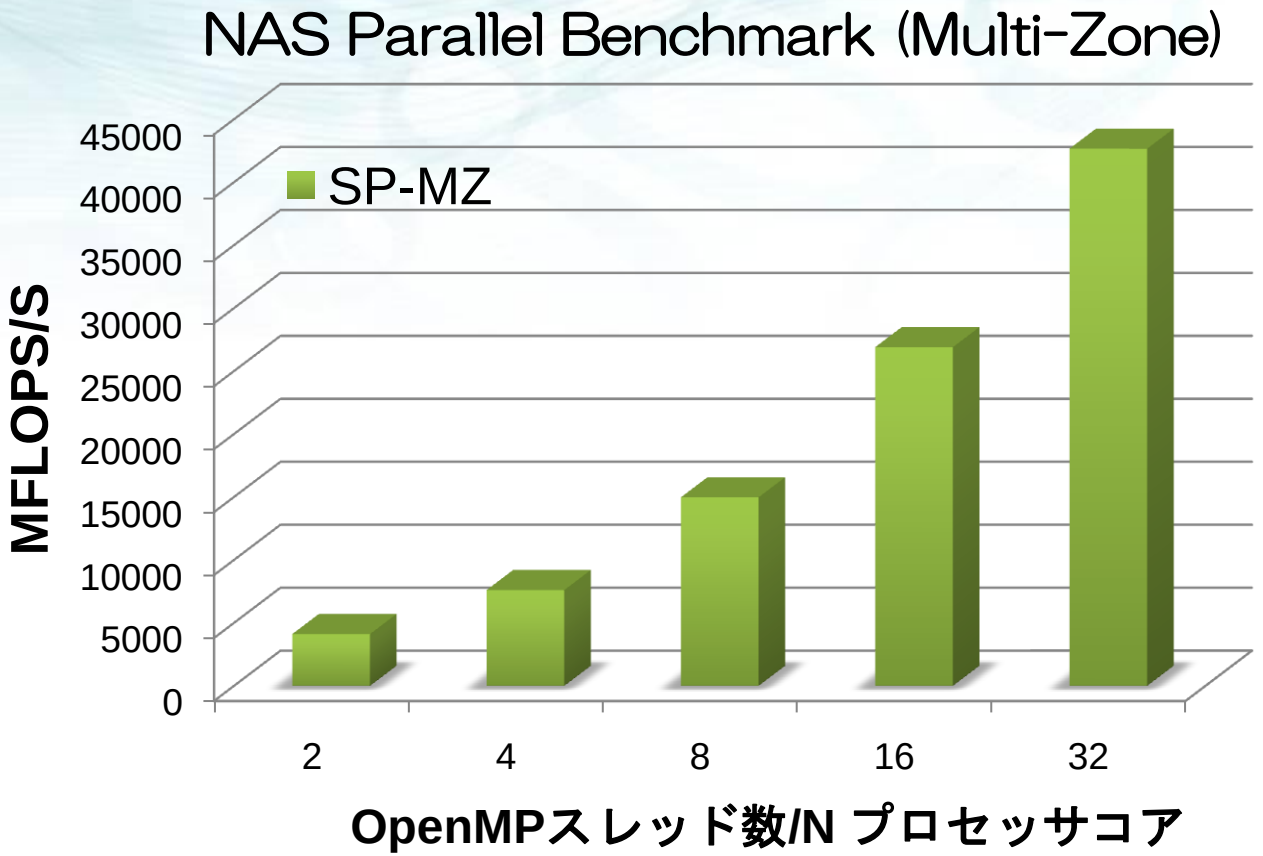
## Hybrid OpenMP MPI Benchmarkproject ("homb")

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.



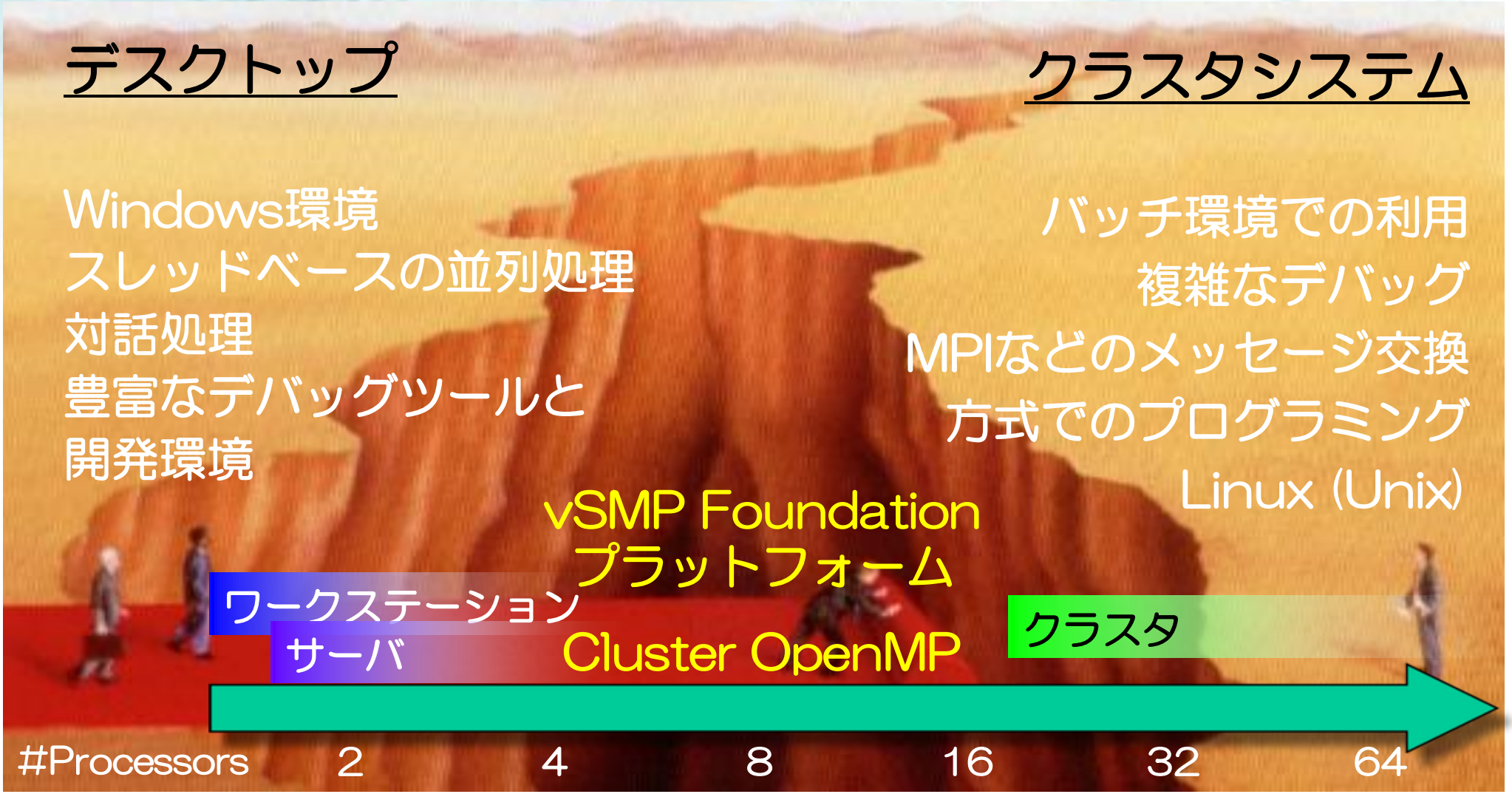
# OpenMPベンチマーク



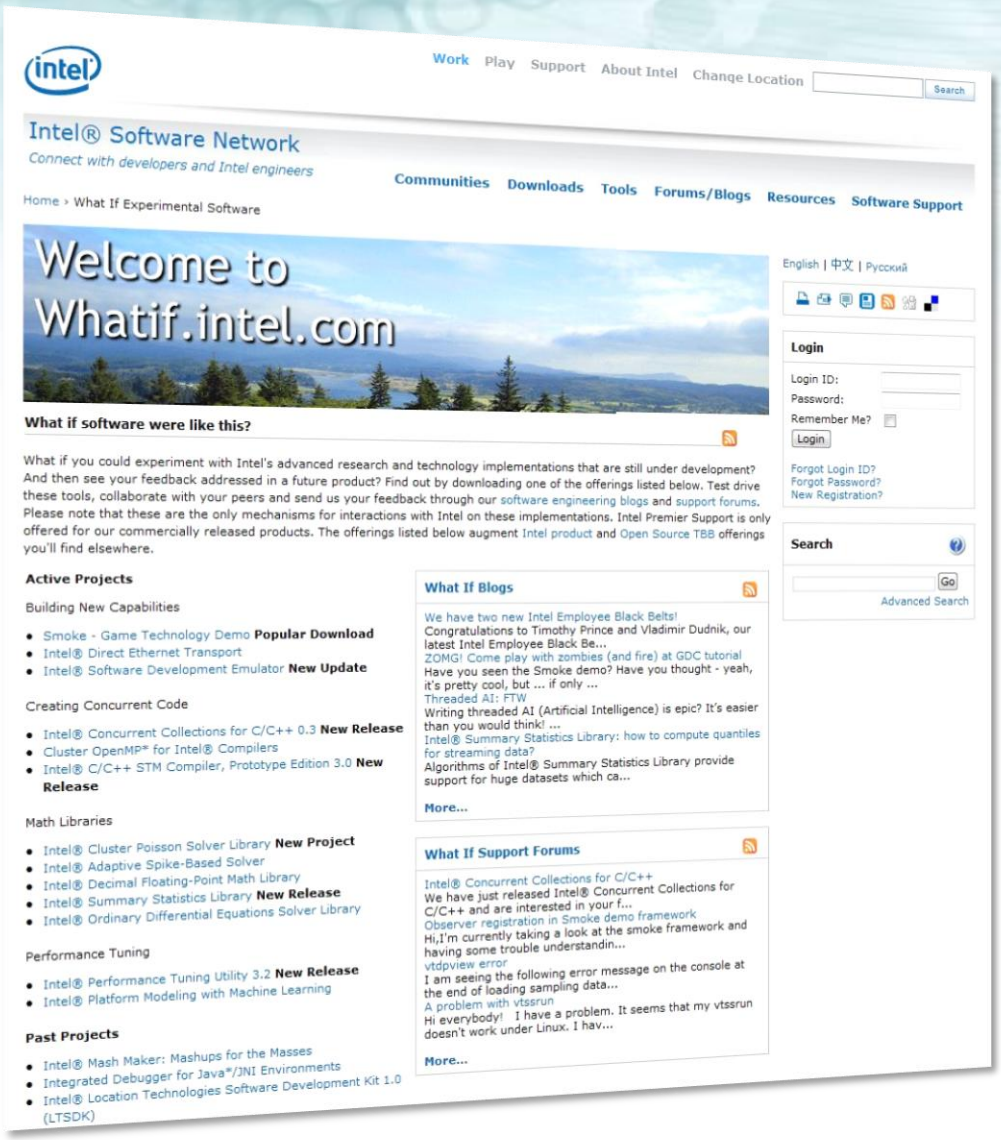




著名な公開ベンチマークツールであるNAS Parallel Benchmark (NPB) の一つであるNPB-MZ (NPB Multi-Zone)はより粒度の大きな並列化の提供を行っています。NPB-MZでは、ハイブリッド型の並列処理やネストしたOpenMPのテストが可能です。ここでの結果は、OpenMPだけでの並列処理の性能を評価しています。
Xeon 5550 (2.66GHz) vSMP Foundation

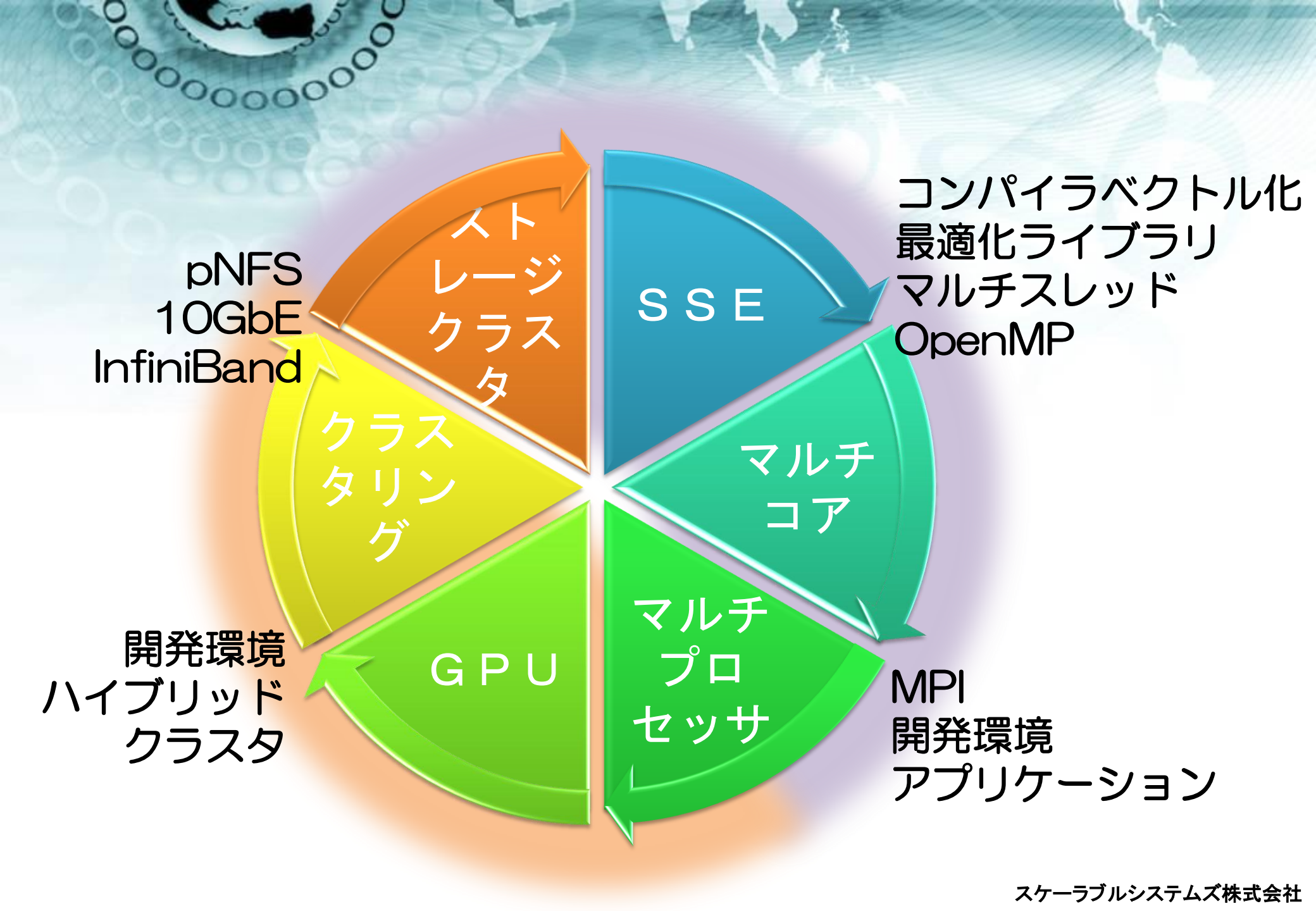
# ソフトウエアのギャップの解決



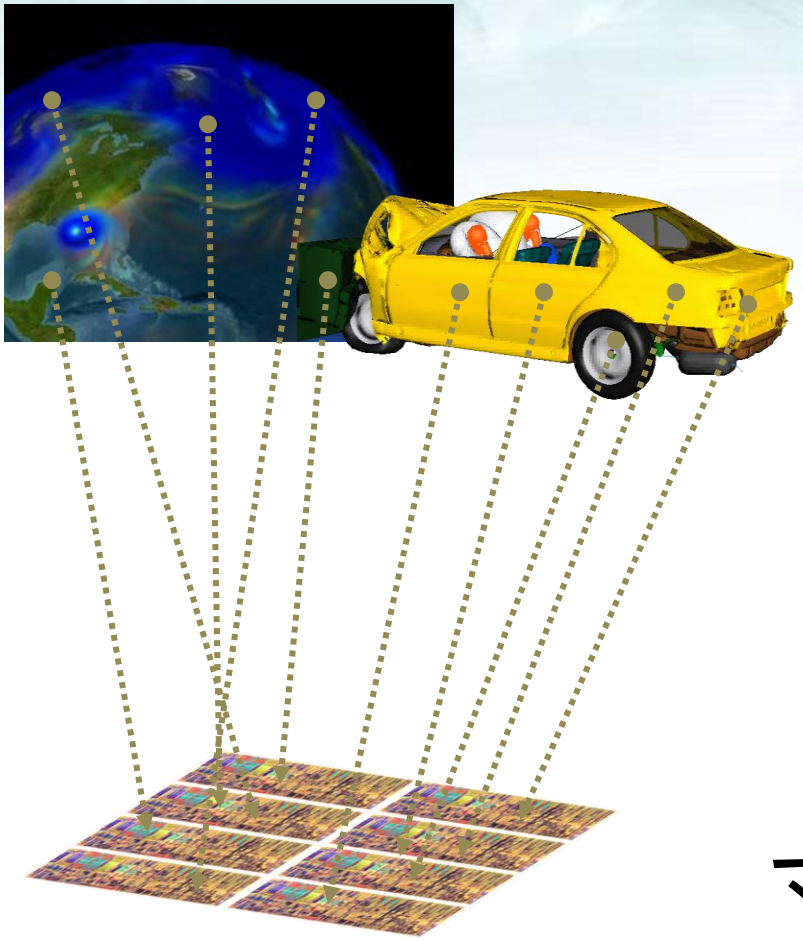
# Whatif.intel.com




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


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

# まとめとして



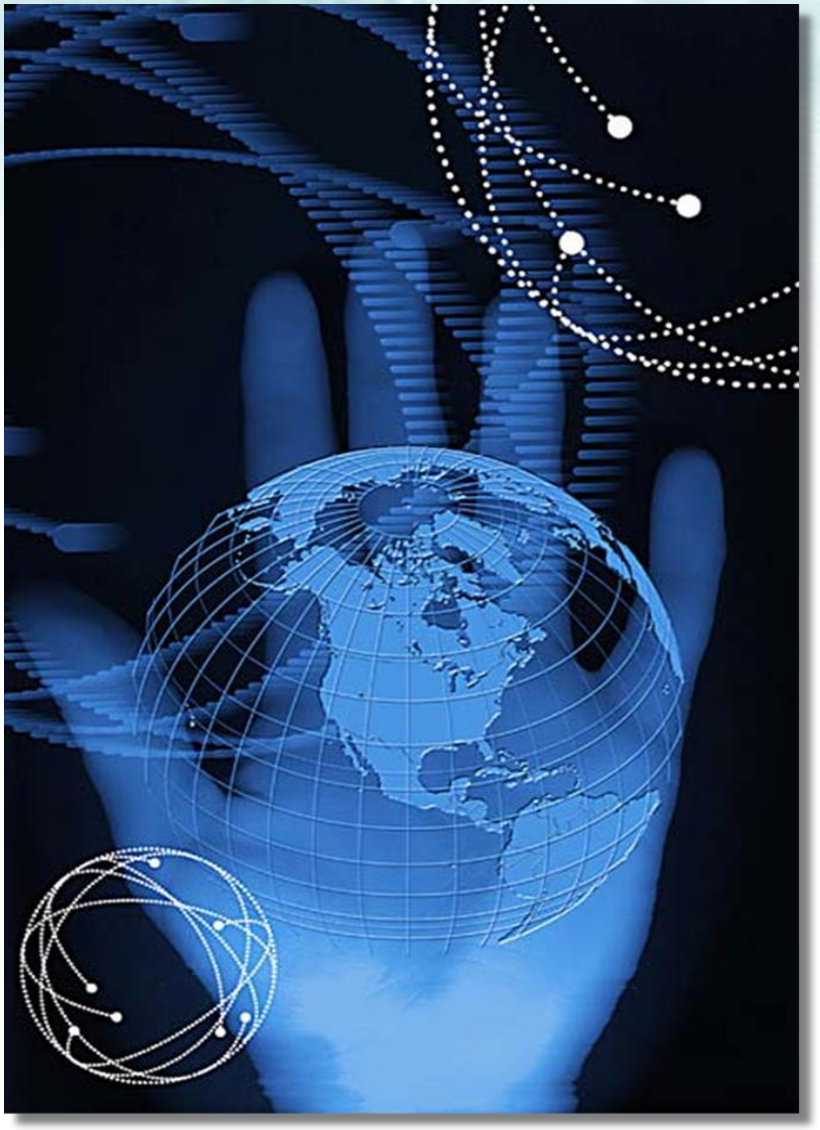
継続的なプロセッサコア数の増加
ベクトル処理の強化
メモリシステムの強化
キャッシュシステムの改善
……

↓

マルチコア上での並列処理
低価格（低い導入コスト）での
スケーラブルなプラットフォーム

↓

マルチスレッドプログラムの可能性



## 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**



**10/16/2009**

**スケーラブルシステムズ株式会社**