マイクロプロセッサと並列処理
期待と現実の狭間で考えること

スケーラブルシステムズ株式会社

説明概要

- はじめに
- 並列処理技術の動向
  - Peta-Scaleコンピューティングと Commodity コンピューティング
  - Commodityコンピューティングでの課題
- マイクロプロセッサと並列処理
  - 並列処理での重要なポイント
  - マイクロプロセッサの進化
- スケーラブルCommodityコンピューティング
  - クラスタOpenMP
  - ハイブリッド並列処理
  - SMP仮想化
- まとめとして

IDF2009 INTEL DEVELOPER FORUM

一部、配布資料にIDF2009の資料を追加致しました。

ユビキタス並列処理プログラミング

- 現在の状況
  - 全てのプロセッサはマルチコアプロセッサ
  - コンピュータは様々な並列処理により性能向上を図っている
  - コンパイラやコンピュータ自身が並列処理の適用を行う場合もあるが、本質的な並列プログラミングはユーザが行う必要がある
- 課題
  - 並列プログラミングの専門家が並列化プログラミングを行い、より高速実行を目指すことは容易
  - より広範囲なユーザが並列処理を行い、その効果を実現出来ることが課題

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:

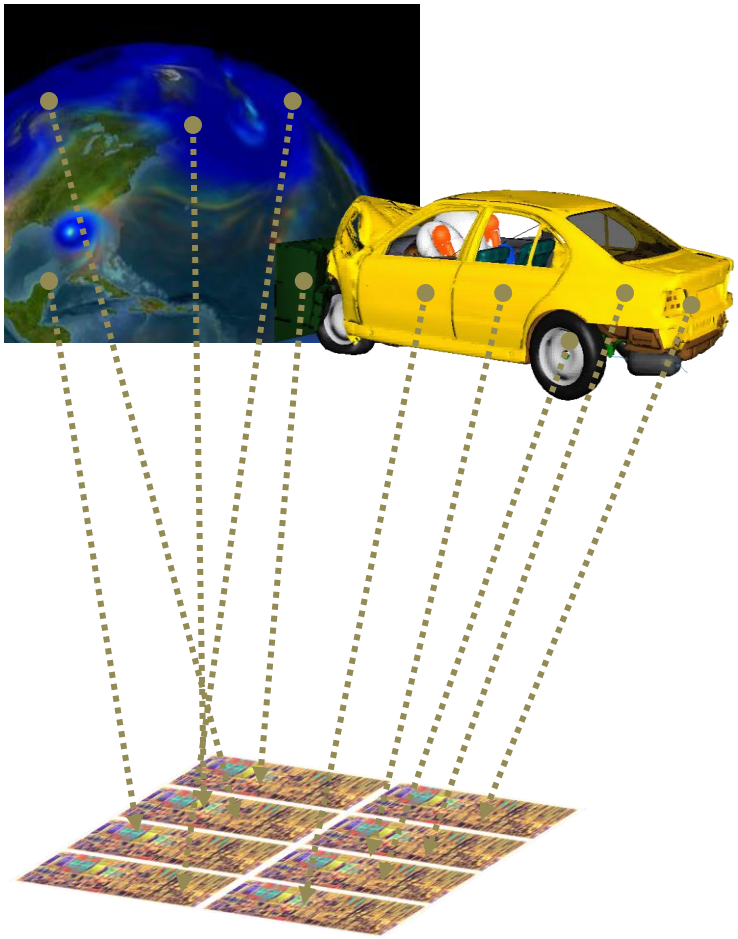
1. Think parallel. Approach all problems looking for the parallelism. Understand where parallelism is, and organize your thinking to express it. Decide on the best parallel approach before other design or implementation decisions. Learn to "Think Parallel."
2. Program using abstraction. Focus on writing 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.

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

1. “並列化”について学ぶ
2. 並列化手法の正しい選択
3. スレッドについて考えるのではなく、タスクでのプログラムを考える
4. 並列実行をオフに出来るようにプログラムをデザインする
5. ロック（同期）処理などは可能な限り行わない
6. 良い並列支援ツールを使う
7. メモリアロケーションに注意する
8. ワークロードに合わせてスケーリング出来るデザインとする

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

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
・・・・・・

システムとユーザの尺度

システムの尺度		ユーザの尺度
Flop/s	⇔	計算終了までの時間
メモリサイズ(GB)	⇔	モデルのサイズと計算結果
プロセッサ数	⇔	ワークロードでの同時実行ジョブ数
データ長	⇔	計算精度
システム構成(クラスタ)	⇔	導入コストと運用コスト
スケーラビリティ	⇔	評価対象での尺度

- ユーザの尺度での性能（Performance）は、時間当たりにどれだけの仕事を処理出来るか（仕事量 / 時間）
- Flopsでの評価は実際には意味がない。また、問題の規模(small, medium, large) という評価も難しい。
- “スケーラビリティ”は、対象を明確に規定する必要がある

Peta-Scaleコンピューティングと
Commodityコンピューティング

並列処理技術の動向

コンピューティングのギャップ

Pleiades Supercomputer
Photo Credit: NASA Ames Research Center

‘Peta-Scale’
コンピューティング
独自のアプリケーション開発
複雑なシステム構成
新しいAPIの提案

ギャップの広がり

コンパクトでより多くのプロセッサコアを搭載したサーバ製品

2Uサイズ
32コア搭載サーバ

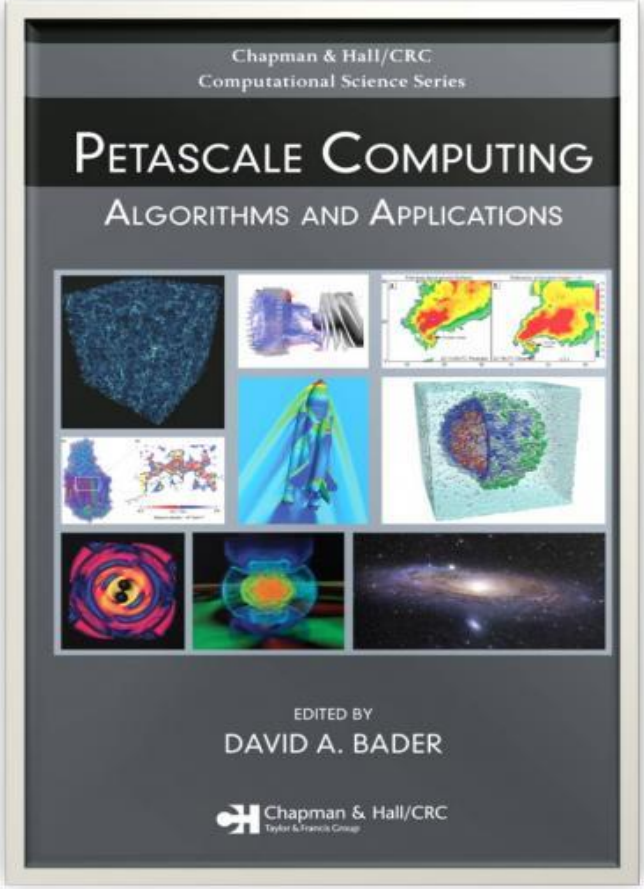
1Uサイズ
CPU+GPU
Hybridサーバ

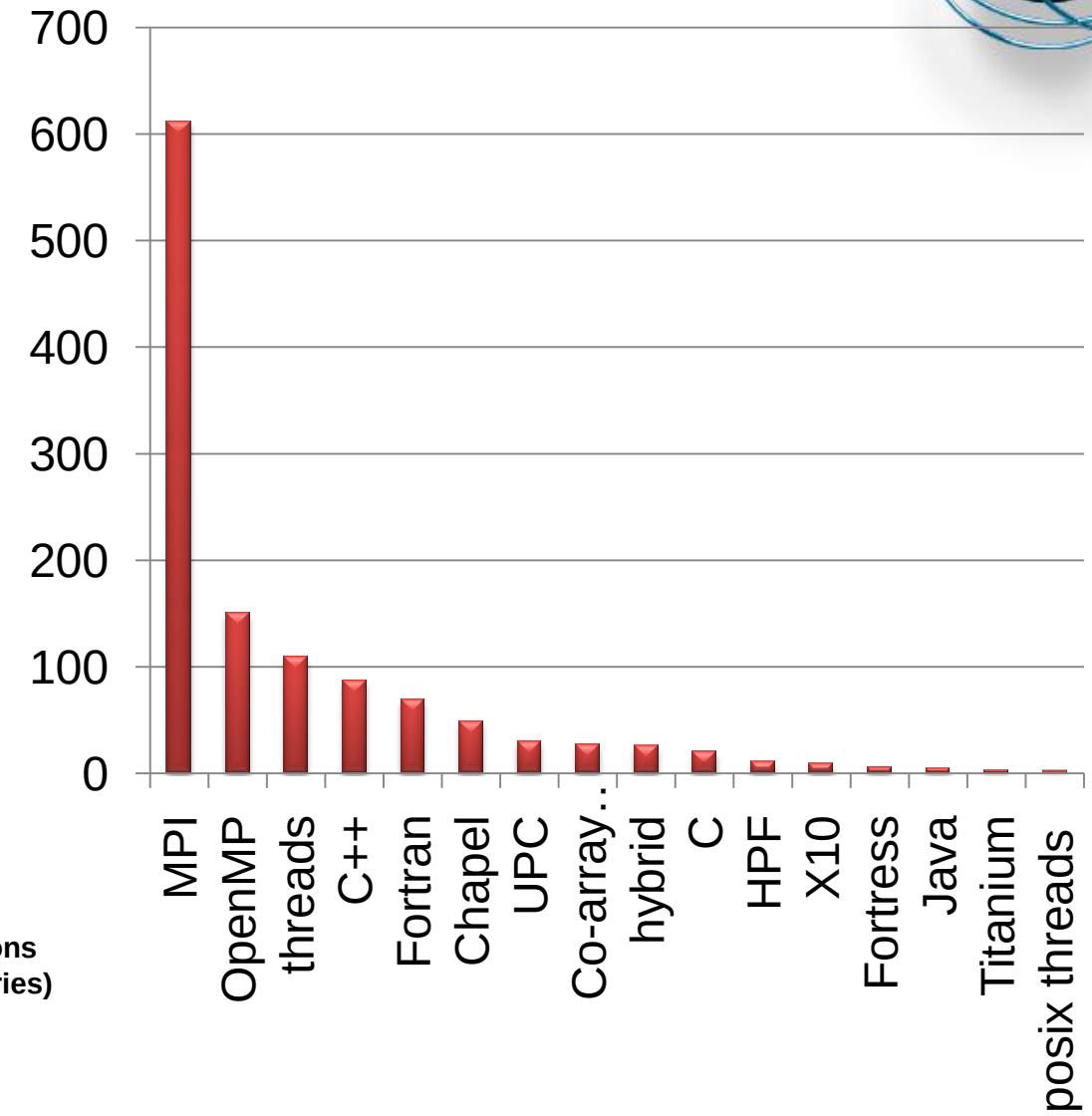
スケーラブル
‘Commodity’
コンピューティング

商用HW/SW
オープンソース
商用アプリケーション
マルチスレッド

マルチコア、マルチプロセッサ、クラスタ
システムの利用の拡大と広範囲なユーザ環境

“First Petascale Book”検索





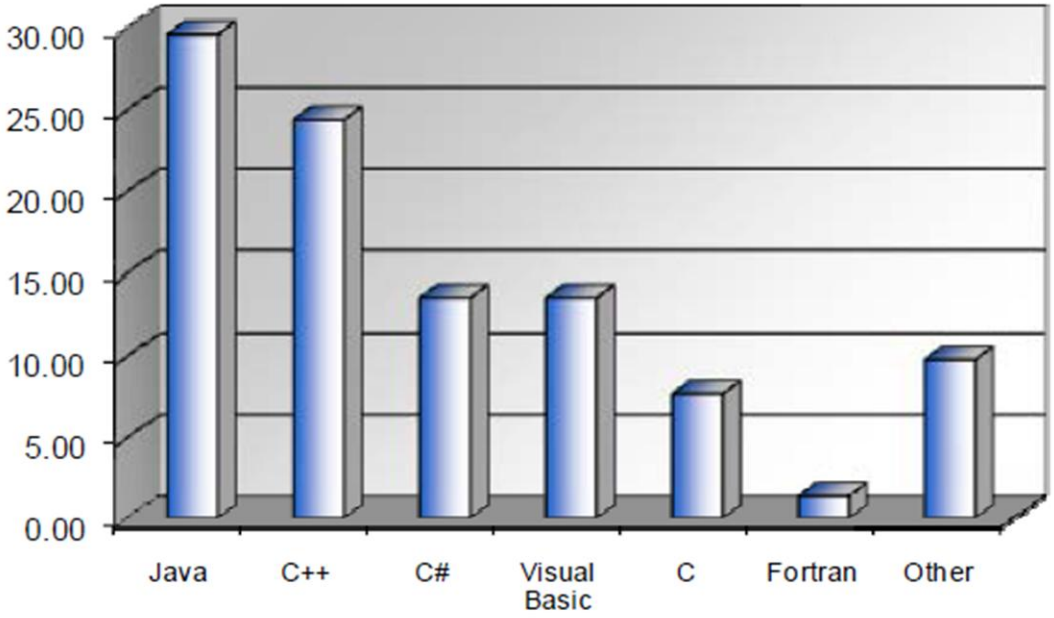
Petascale Computing: Algorithms and Applications (Chapman & Hall/Crc Computational Science Series)

Scalable OpenMP Programming
Dieter an Mey RWTH Aachen Universityより抜粋

マルチスレッドアプリケーション

Evans Data Corporation
EDC

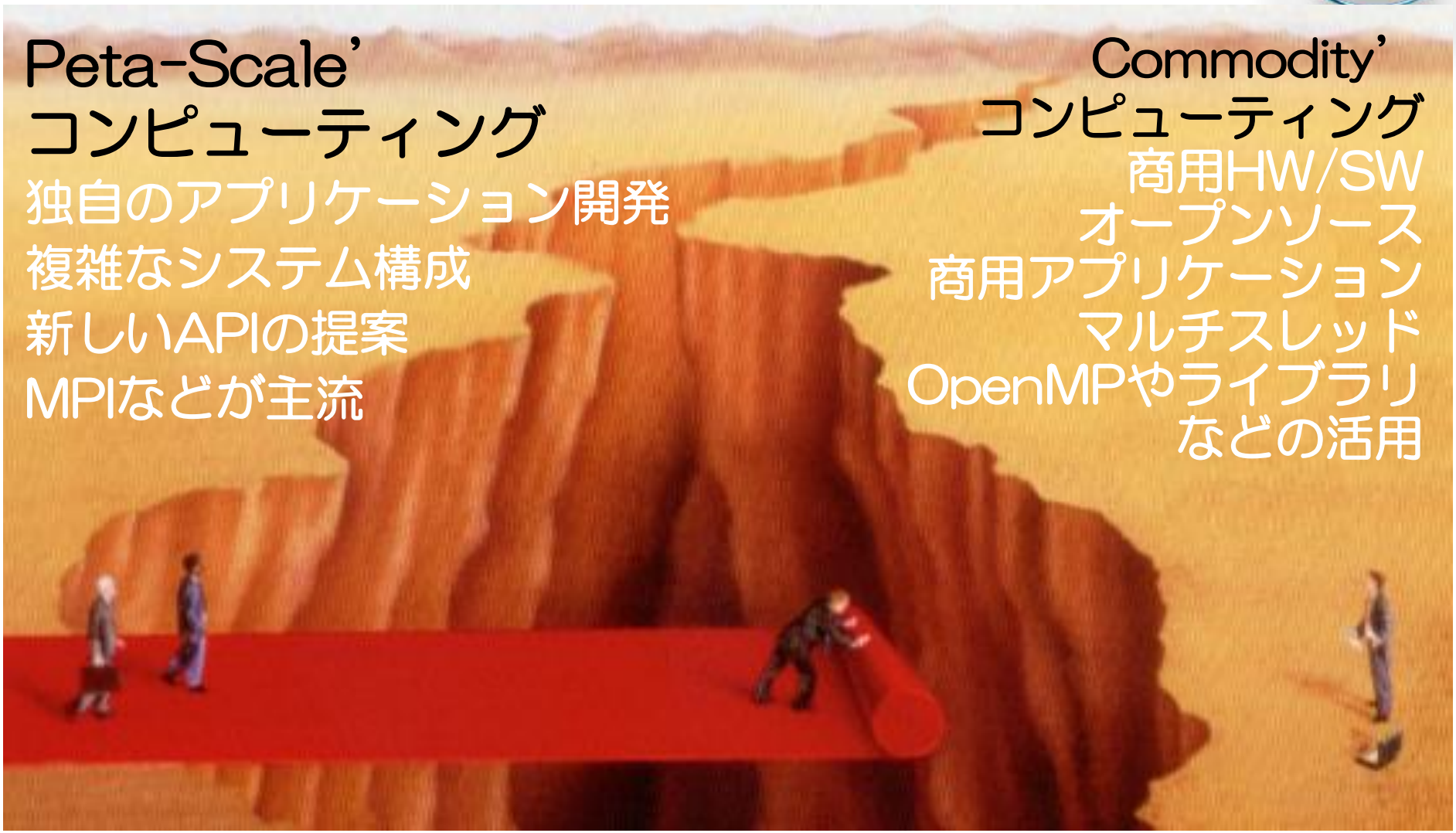
Languages Used for Creating Multi-Threaded Applications



EMEA Development Survey © 2007 Evans Data Corp.

Intel Software Product Conference:Parallel Programming AdoptionMarket Situation and Outlook
presentations from Multicore Days 2008, 11-12 September / James Reinders, Intel

プログラミングのギャップ

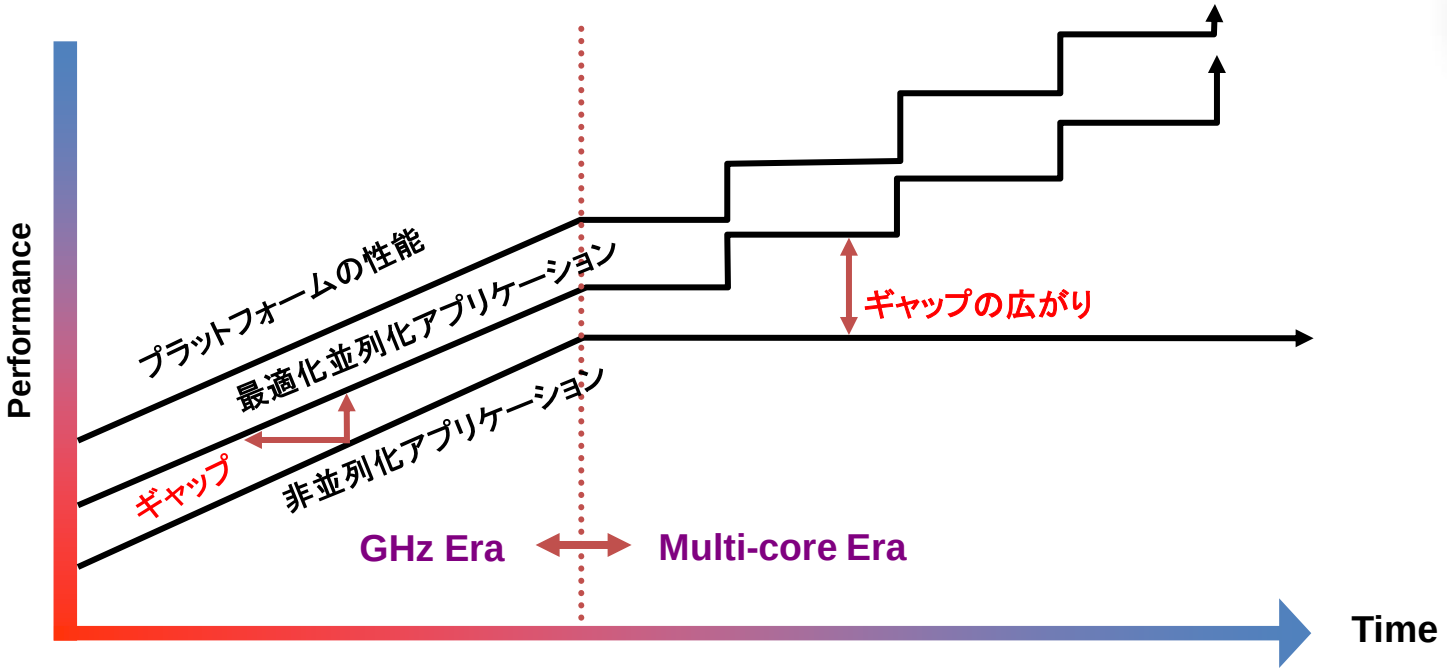


プログラミングのギャップ

マイクロプロセッサの進化と並列処理への貢献

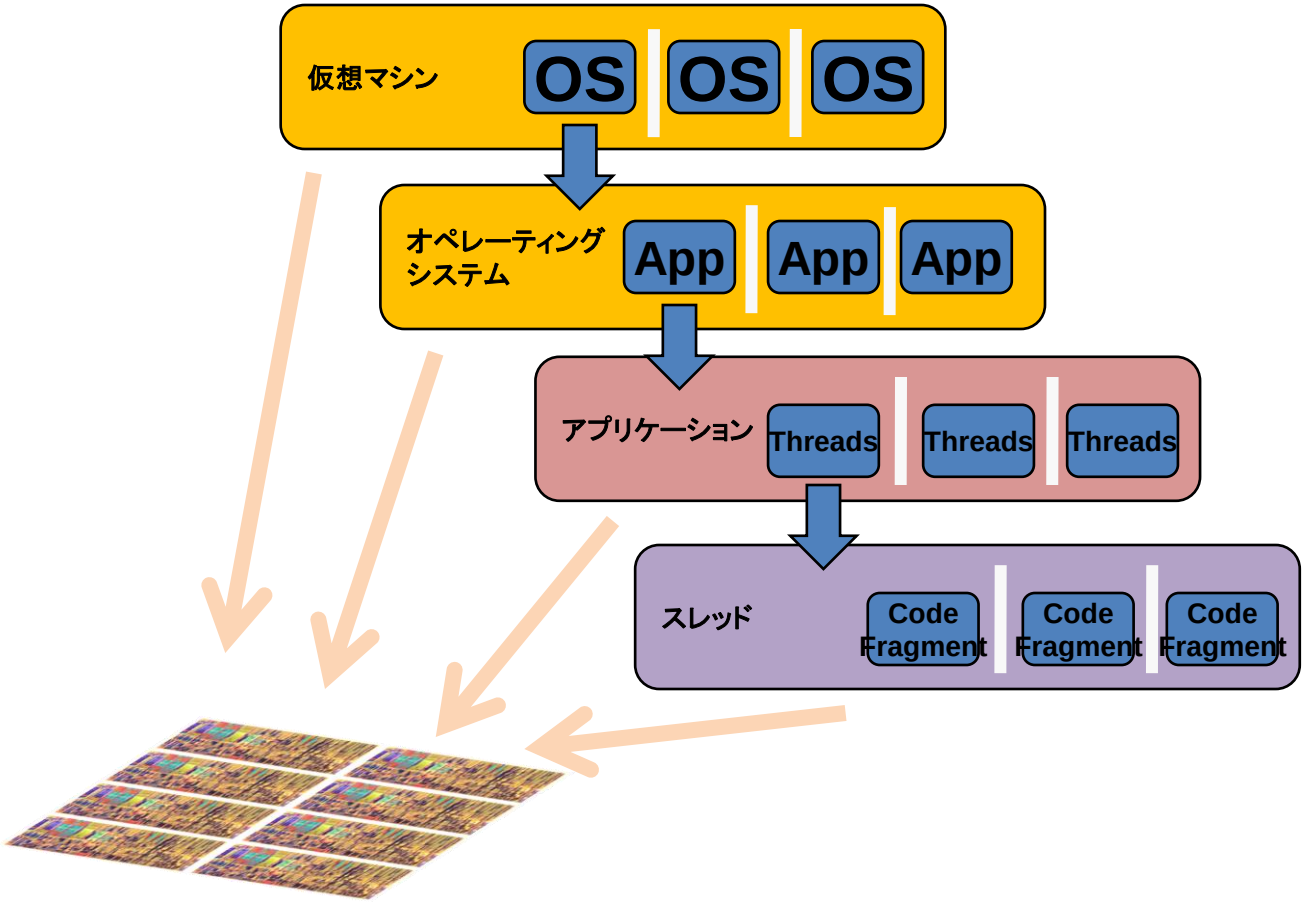
マイクロプロセッサと並列処理

アプリケーションの性能向上

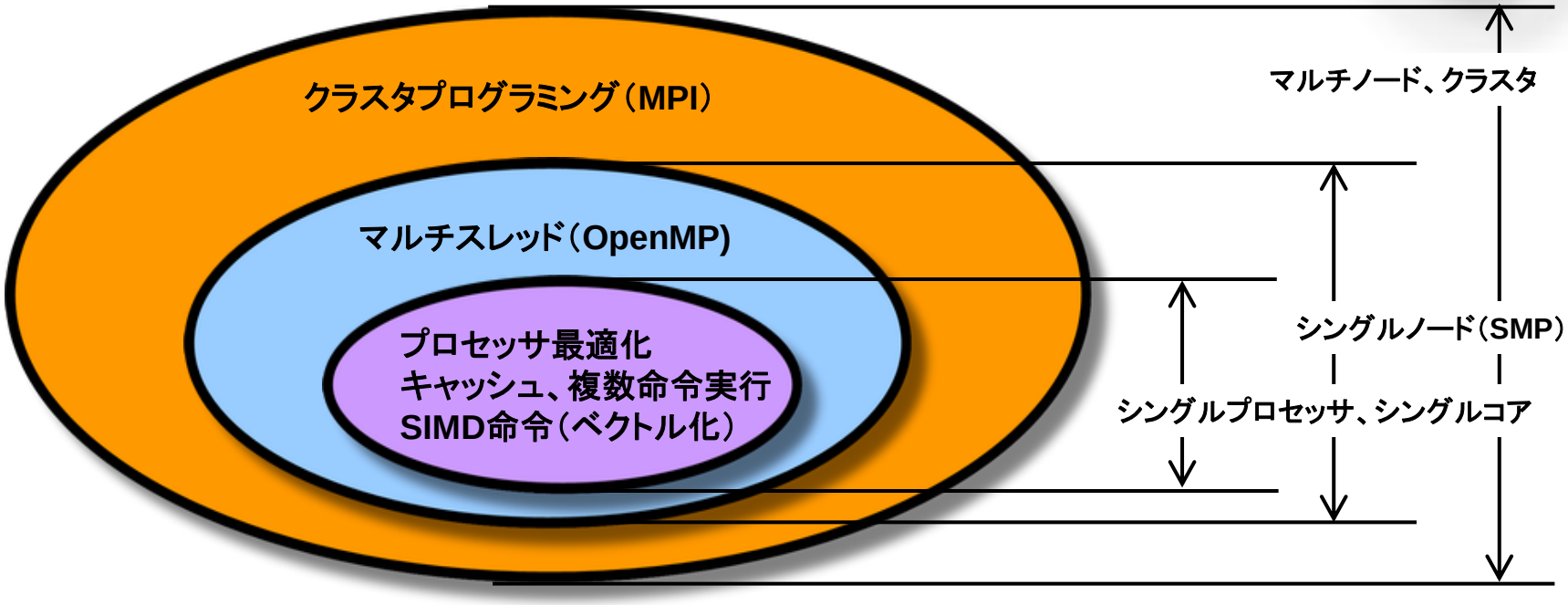


- 並列処理は誰でも利用可能（利用のメリット）
  - より少ないコスト（低価格なシステム）でより効率の良い処理が可能（対費用効果）
  - より短時間でシュミレーションを完了（開発サイクルの短縮によるコスト削減）

アプリケーション実行階層



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

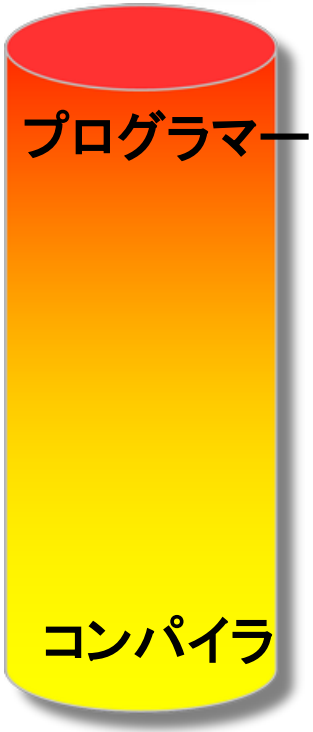
プログラミング階層

```
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

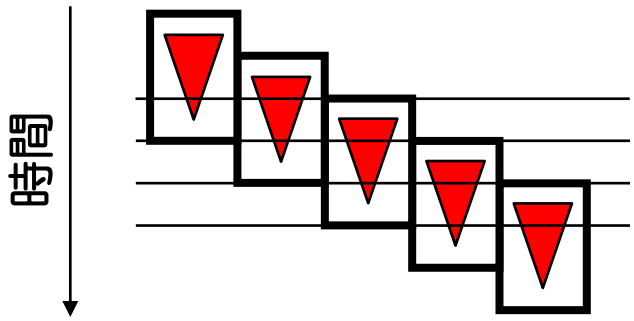
ノード内でのマルチスレッド並列化
OpenMPやスレッドプログラミング

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

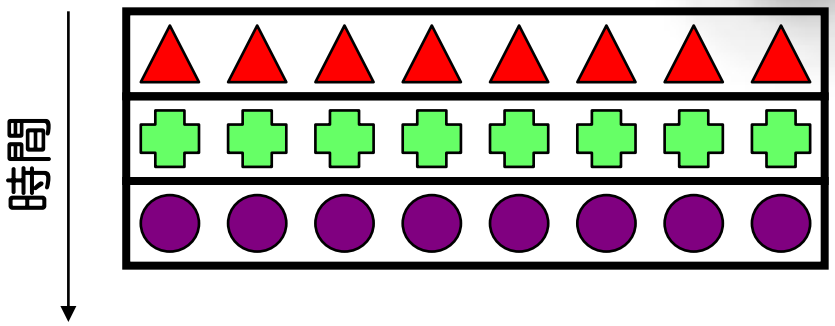


最適化と並列化の適用作業

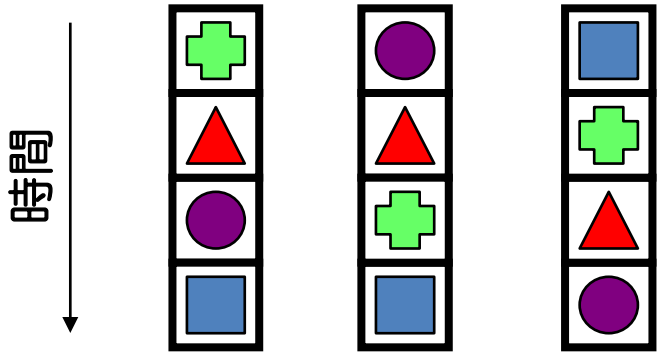
並列性（Parallelism）の利用



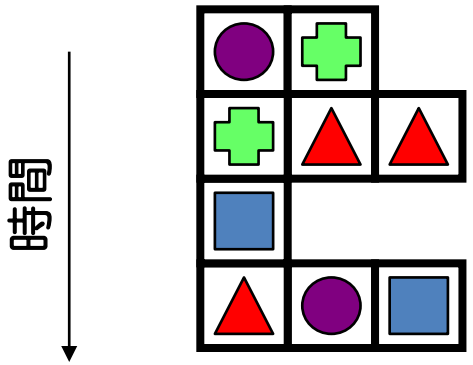
パイプライン処理



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）

ループのベクトル化処理

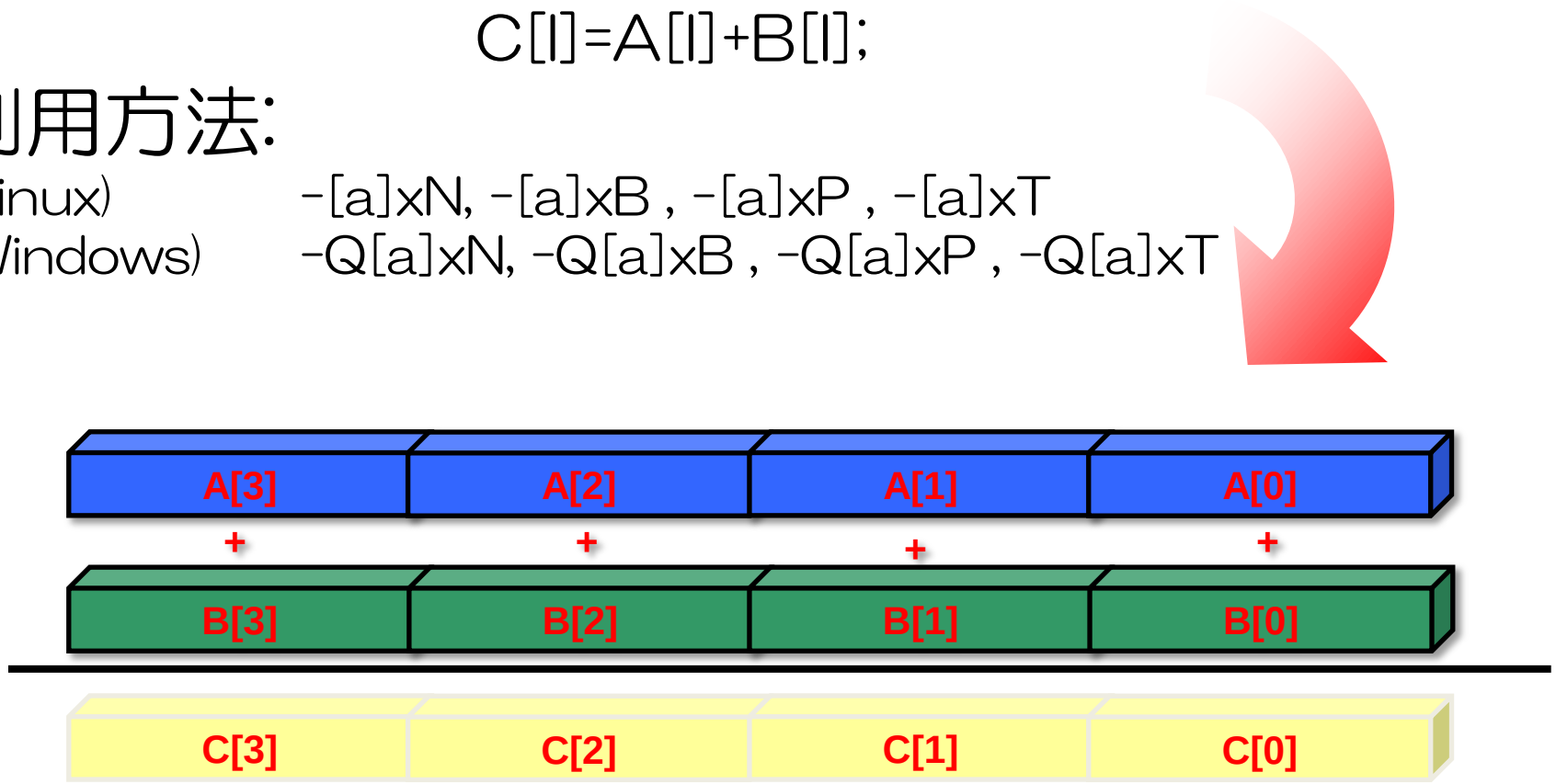
プログラム例:

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

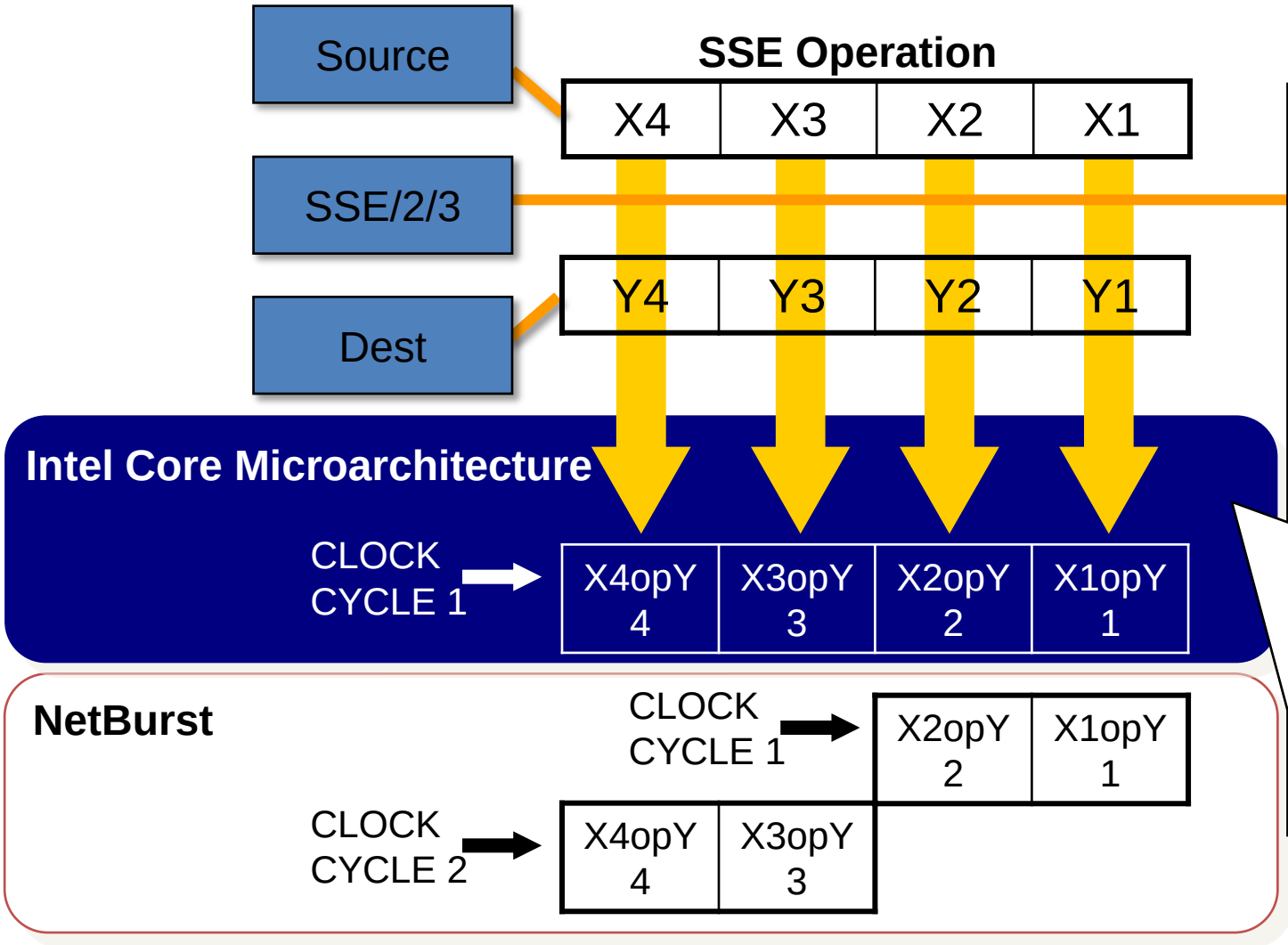
利用方法:

(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

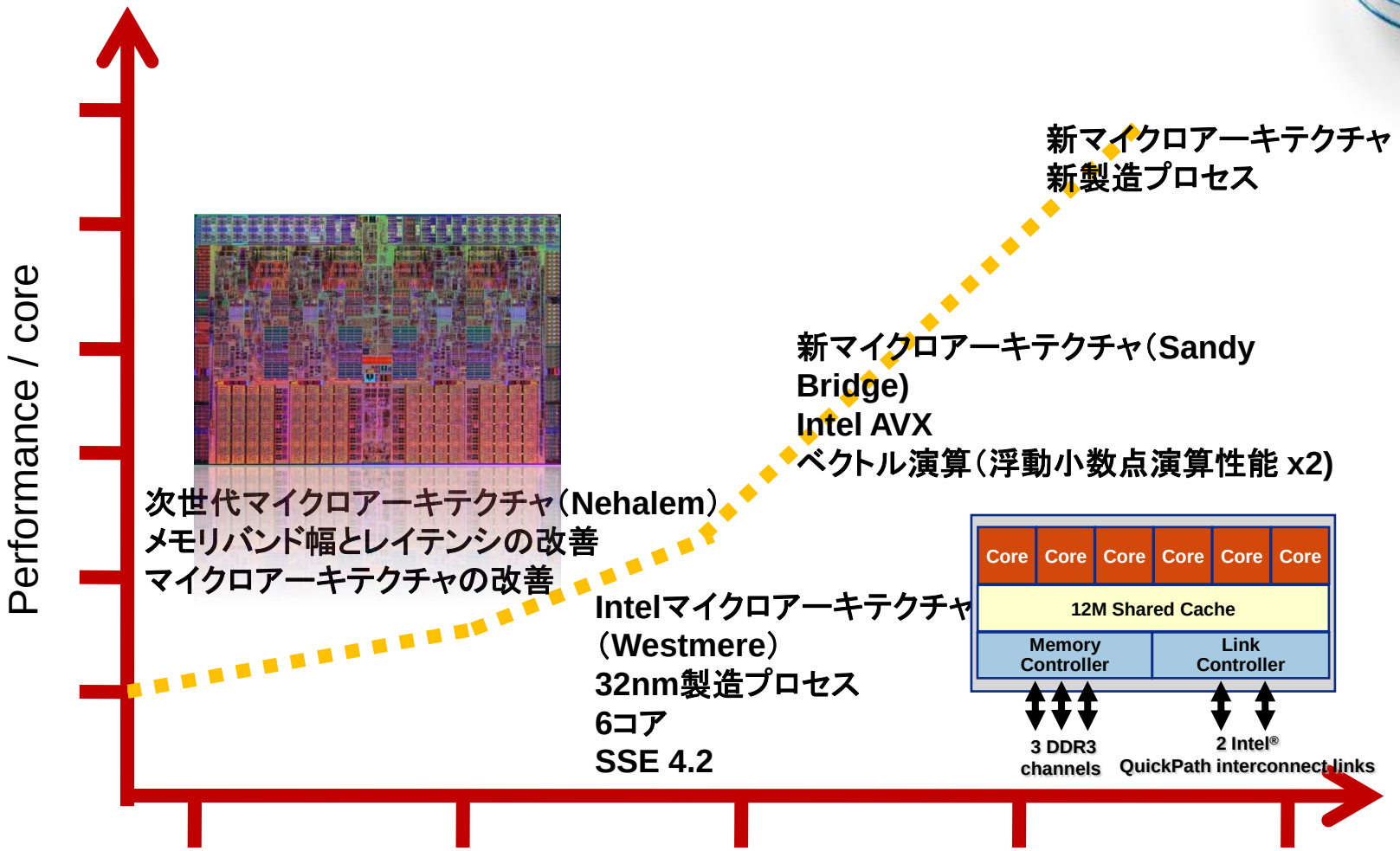


各MMX/SSE演算は、128ビットの演算をシングルサイクルで実行可能となる。MMX/SSE演算器は2セットあり、同時実行が可能となる。従って、単精度では、8浮動小数点演算、倍精度では、4浮動小数点演算を1クロックで実行することが出来る。

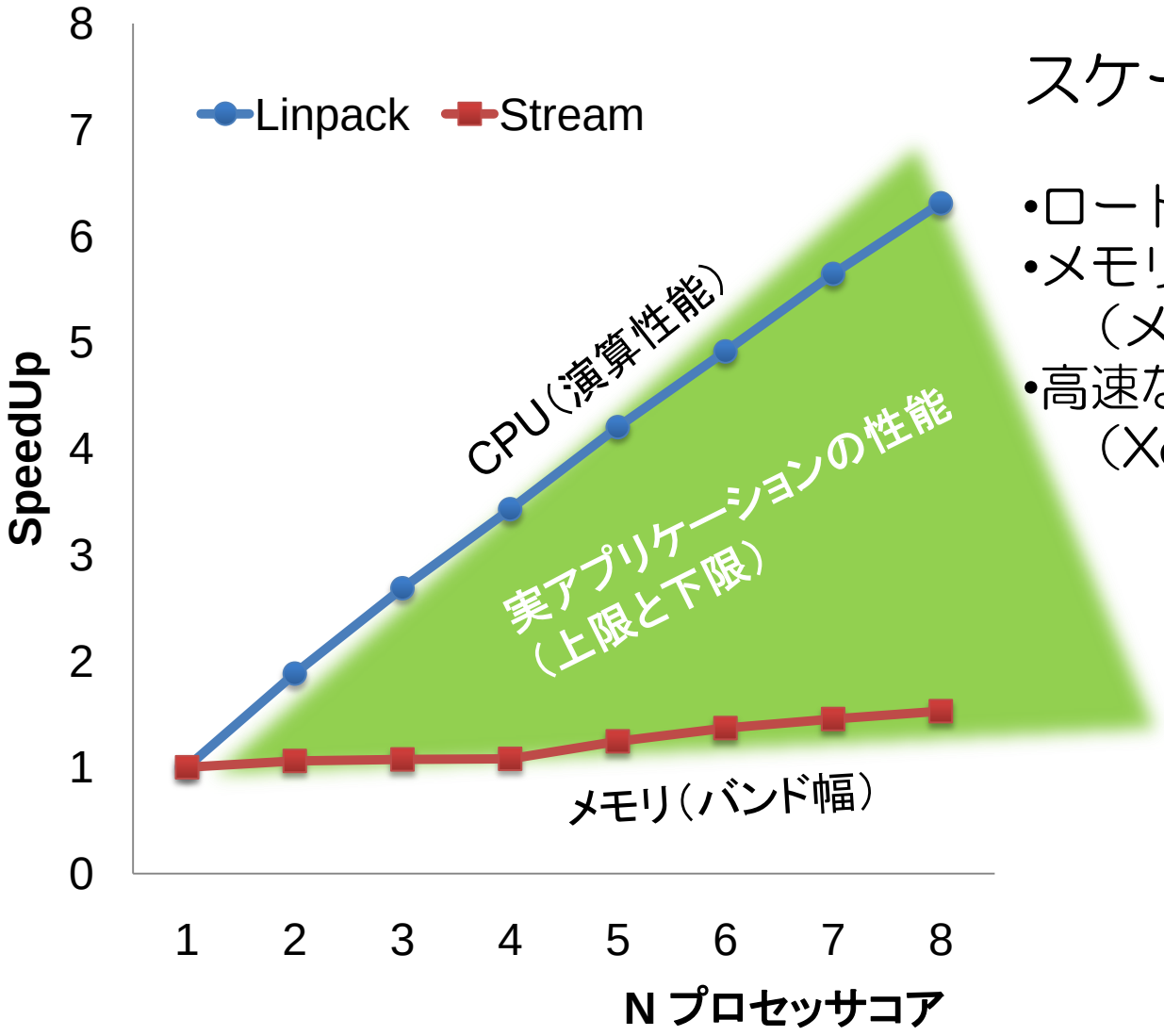
並列処理での重要なポイント

- 並列化
  - 最適な並列化アルゴリズムの選択
  - プロセッサ数や問題の規模に対応可能な並列化実装
  - コア数やプロセッサ数に依存しないこと
- メモリの局所性
  - メモリ階層を効率的に利用
  - 頻繁にアクセスするデータをよりコアの近傍に配置

Intel マイクロアーキテクチャ



並列処理スケーラビリティ

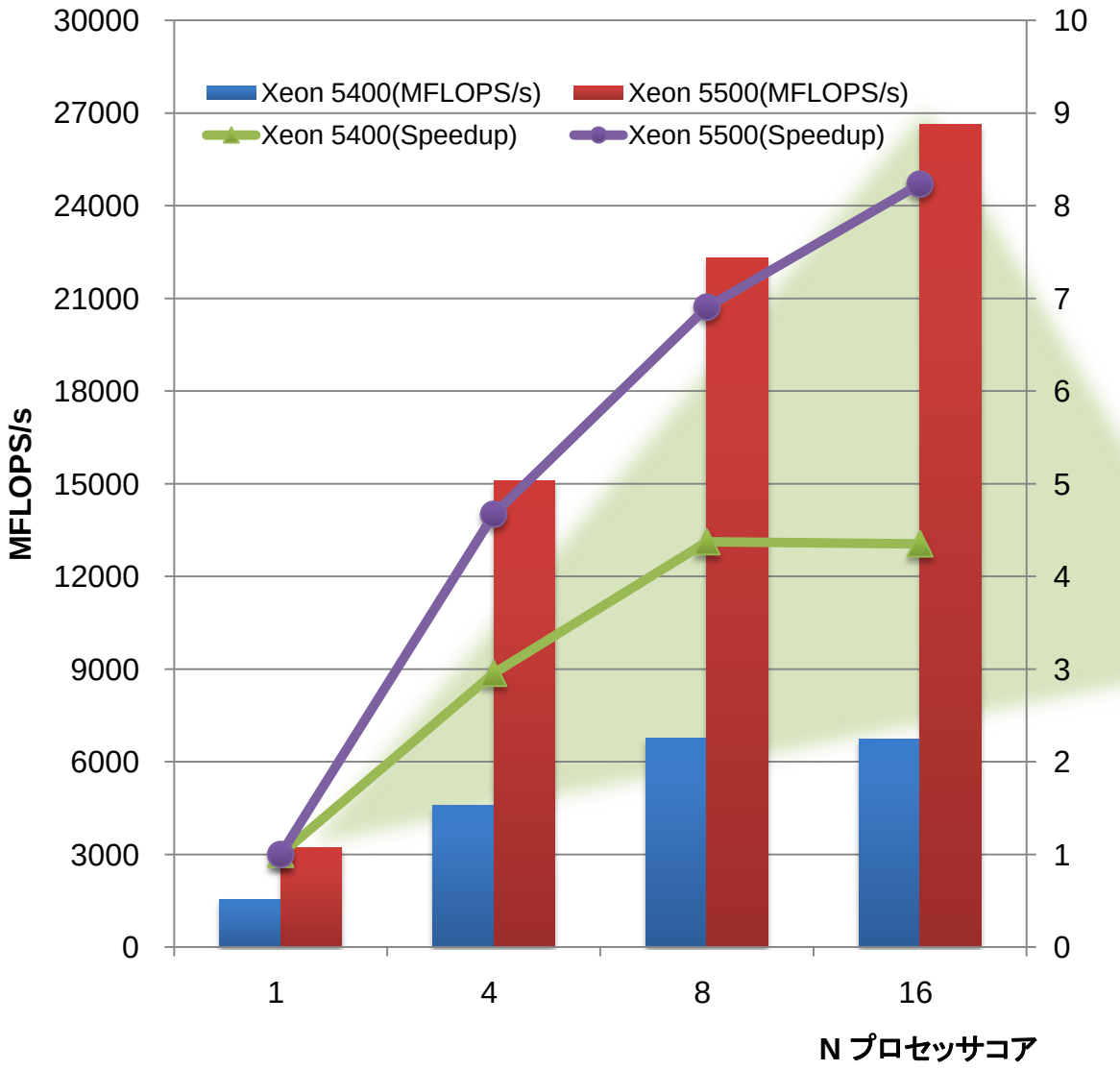


スケーラビリティの向上

- ロードバランスの改善
- メモリ階層の効率的な活用
  （メモリアクセスの低減）
- 高速なメモリシステムの採用
  （Xeon 5500）

テスト事例

並列処理スケーラビリティ



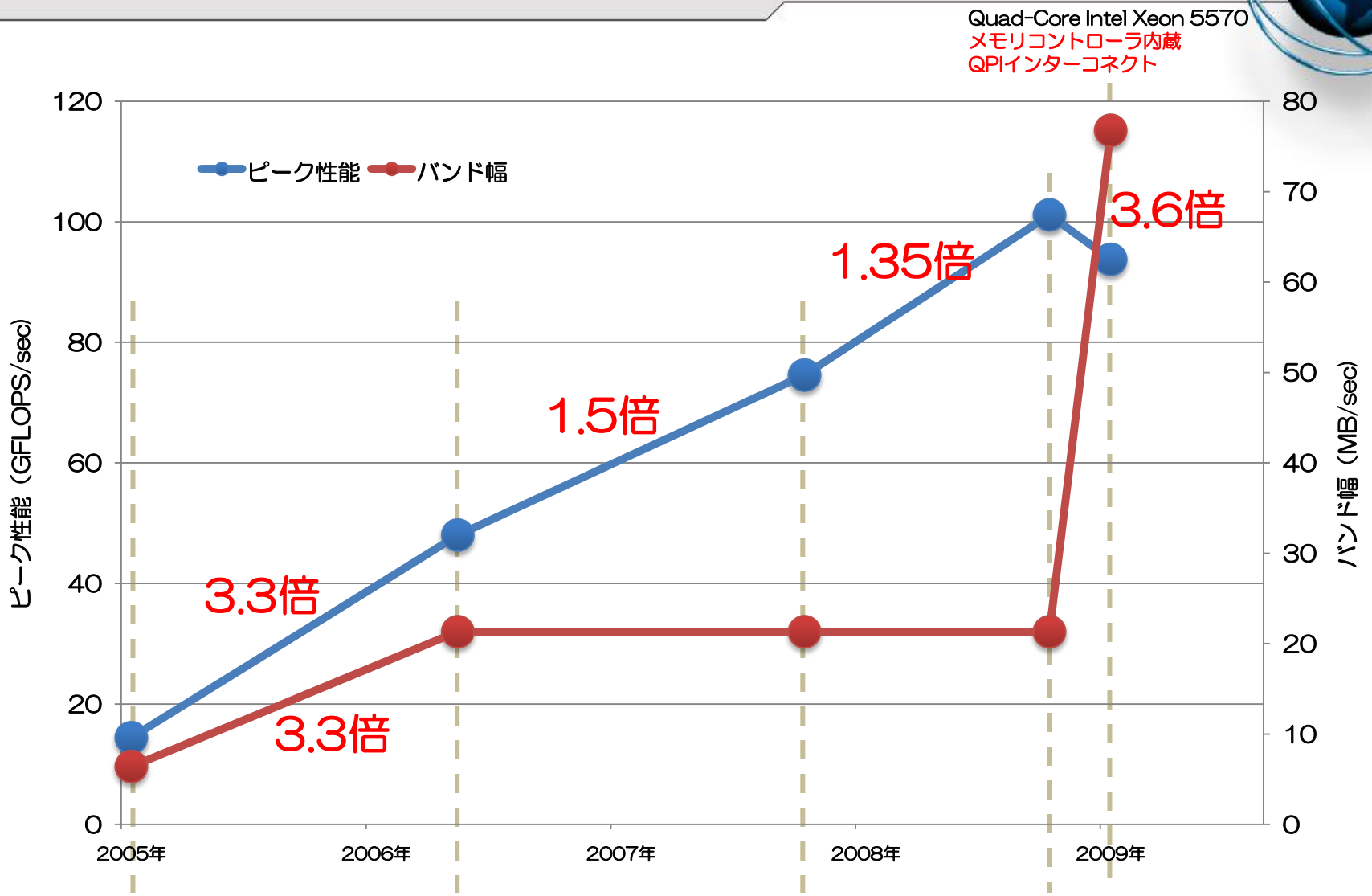
システムアーキテクチャの進化

シングル(逐次処理)性能の向上
スケーラビリティの向上

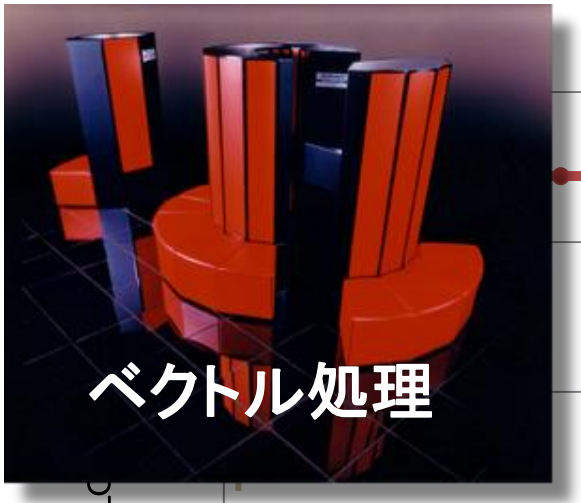
効率的な並列処理には
プラットフォームが重要

Himeno Benchmark
http://w3cic.riken.go.jp/HPC/HimenoBMT/index.html

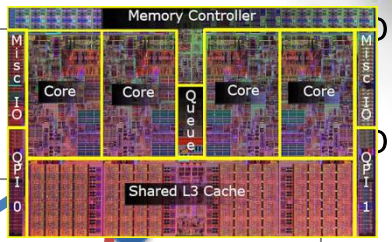
プロセッサ‘性能’向上比率



プロセッサ‘性能’向上比率

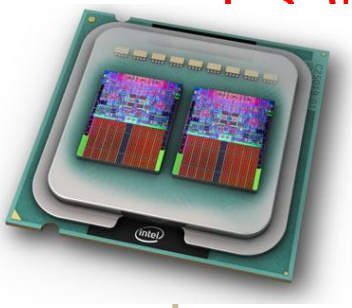


Quad-Core Intel Xeon 5570
メモリコントローラ内蔵
QPIインターコネクト



バンド幅

1.35倍



1.5倍

3.3倍

3.3倍

ピーク性能（GFLO

60
40
60
50
0

2006年
2007年
2008年
2009年

64-bit Intel Xeon 3.6GHz 2M

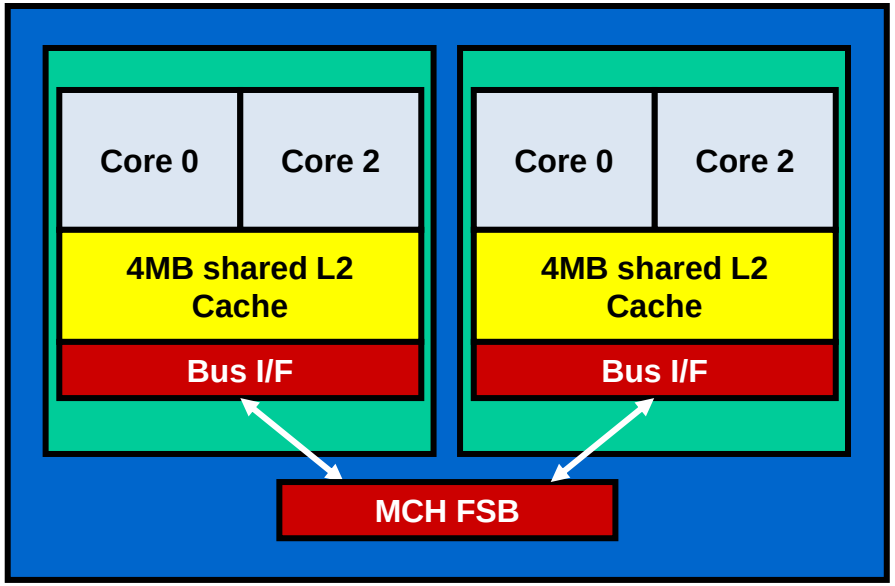
Dual-Core Intel Xeon 5160
デュアルコア
4浮動小数点演算/クロック
デュアルFSBバス

Quad-Core Intel Xeon 5355
クァッドコア

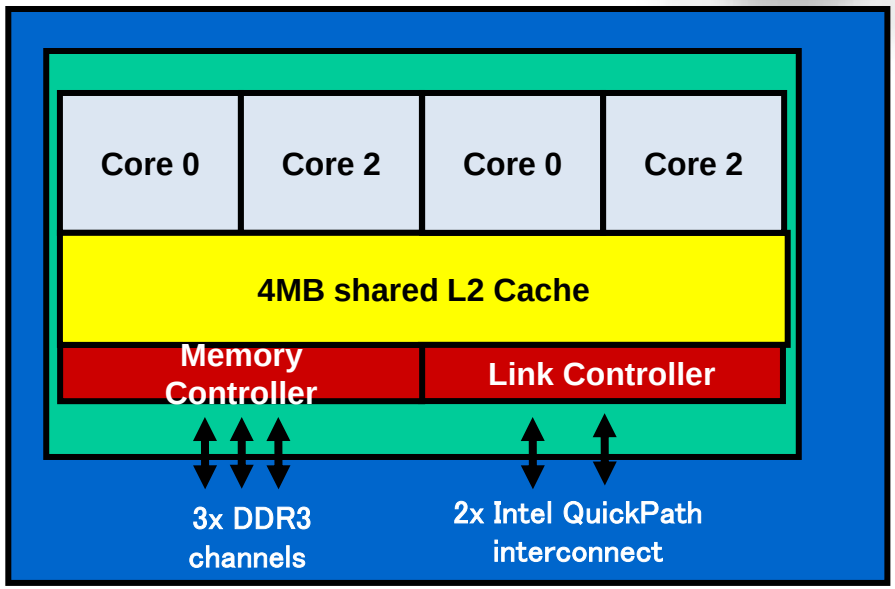
Quad-Core Intel Xeon460
45nm製造プロセス
動作クロック

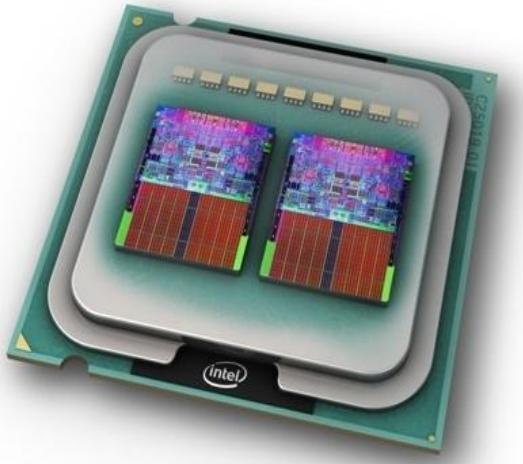
メモリ性能とスケーラビリティ

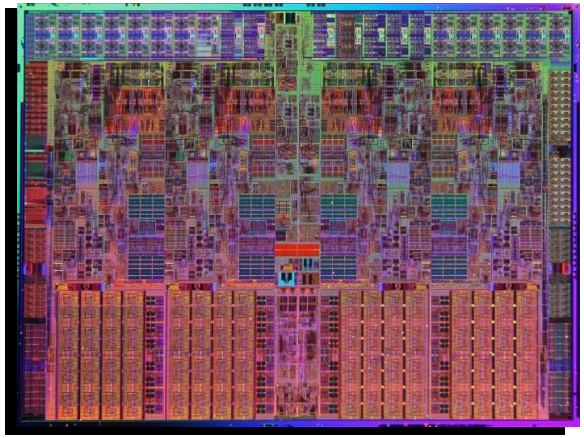
Core 2 Extreme QX6700



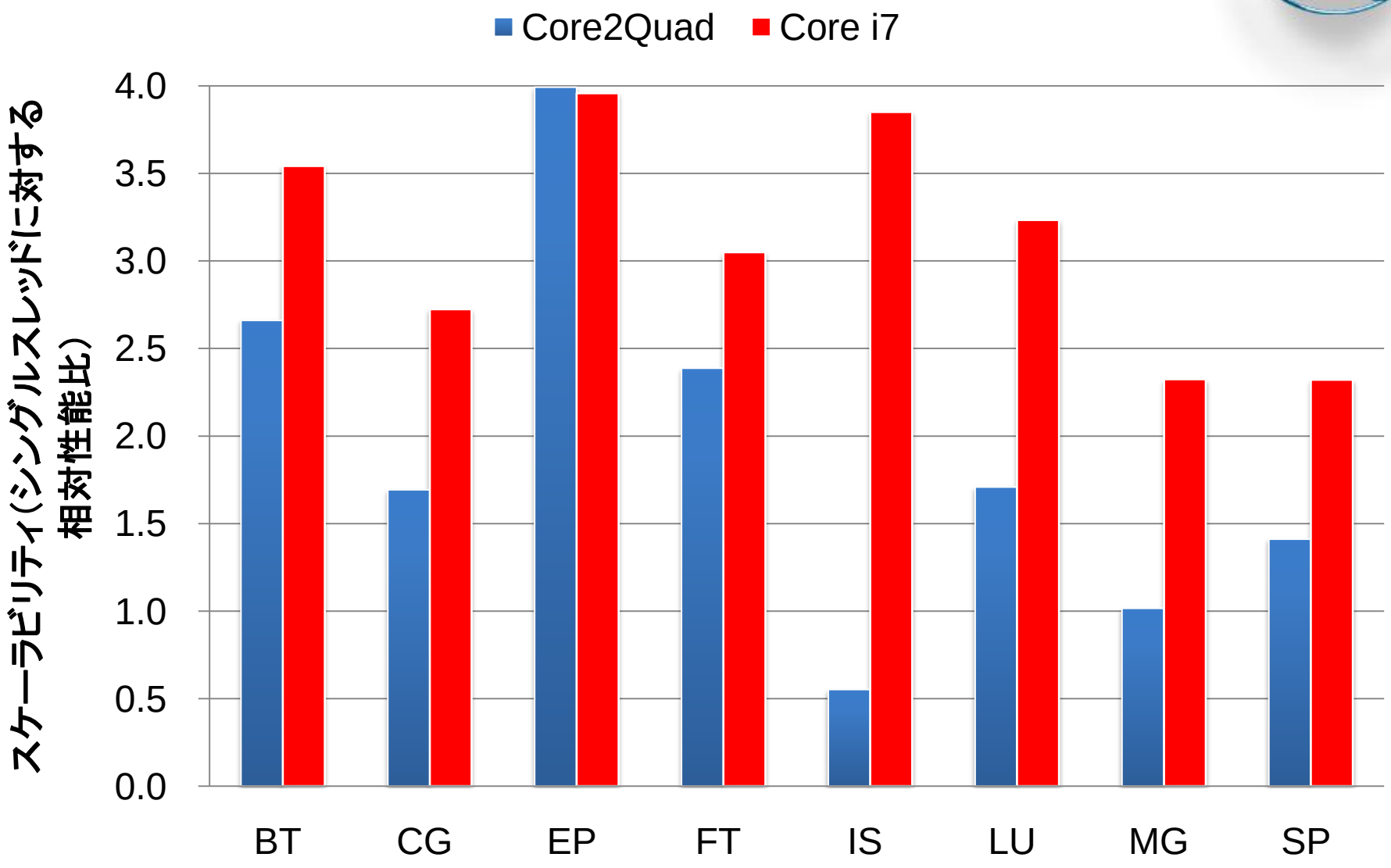
Nehalem



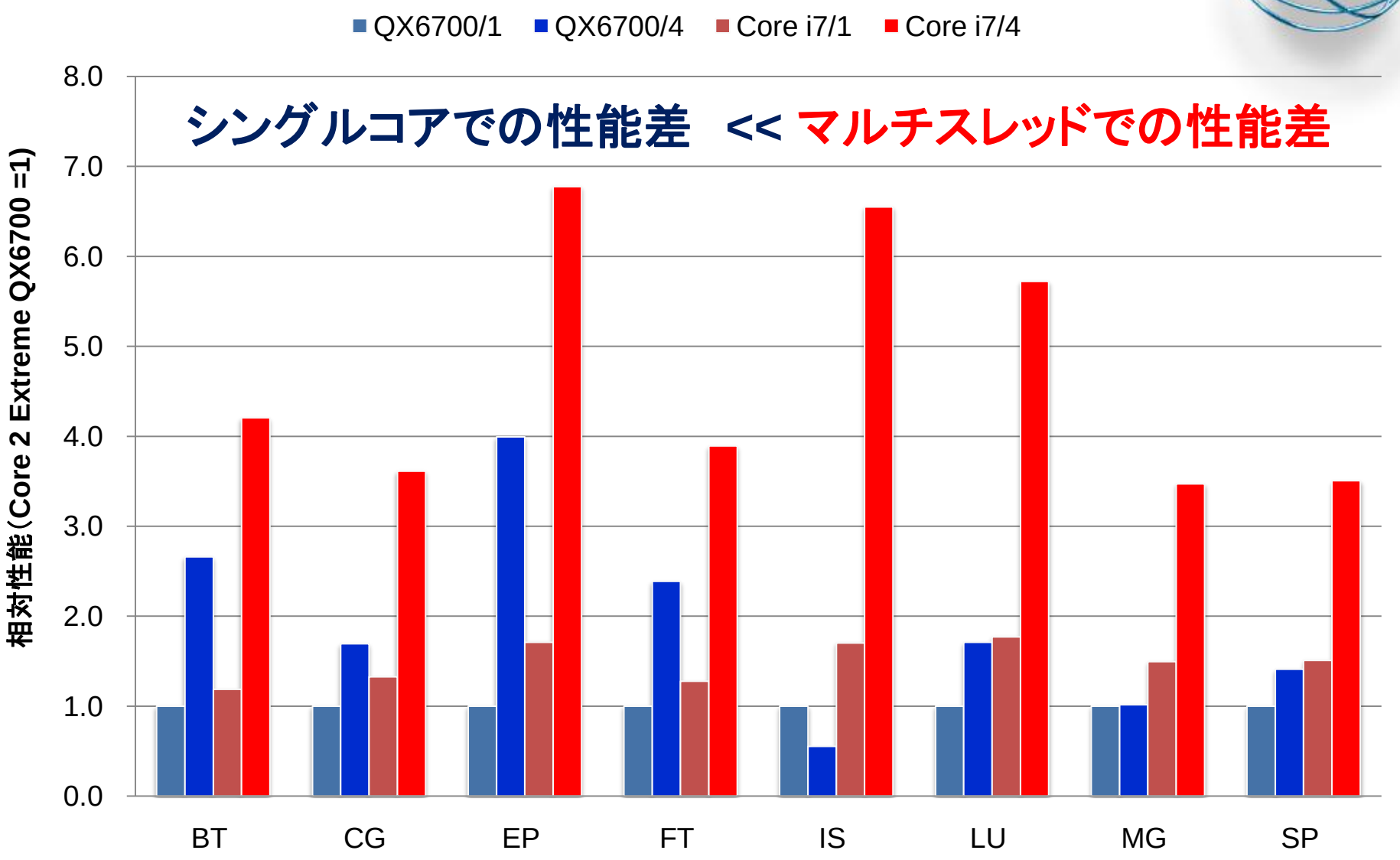




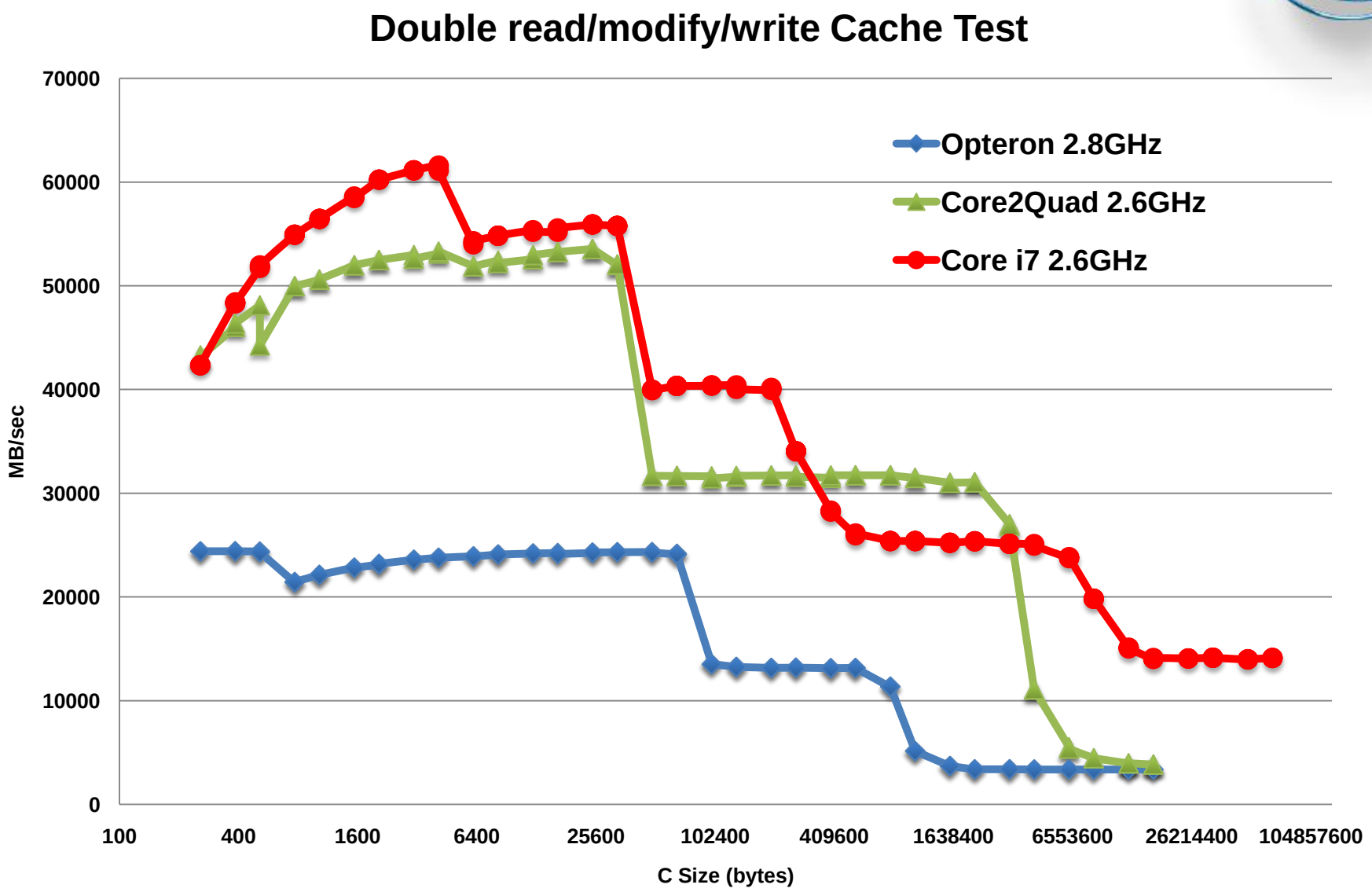
NPB OpenMP - スケーラビリティ



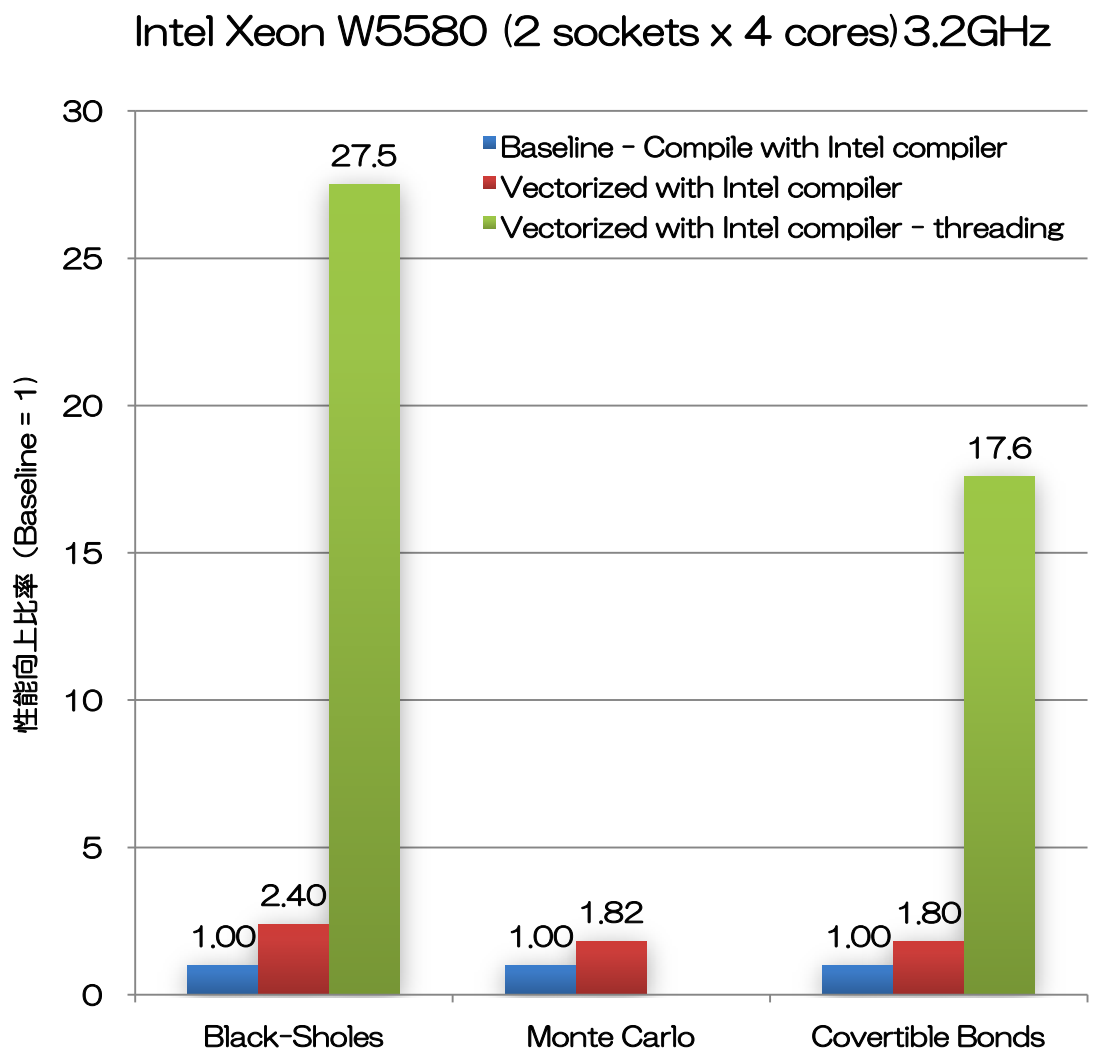
NPB OpenMP -相対性能



メモリ階層ベンチマーク



ベクトル化・マルチスレッド



IDF2009
INTEL DEVELOPER FORUM

Vectorized /
ベクトル化
- コンパイラ指示行の追加
- コードのアンローリング
- ループの分割

Threading/
- 自動並列化
- OpenMP指示行の追加

コンパイラの自動ベクトル化と
自動並列化にユーザの最適化
作業でより高い性能を実現

"Money Tree - Optimizing FSI Benchmarkswith Intel® Software Tools for Multicore & Manycore"
Intel Developer Forum

スケーラブルCommodityコンピューティング

並列処理の課題と挑戦

“Many core” CPU

- 2012(?)に想定される計算ノード 1)
  - Node：960 GFLOPS/CPU
    - Many core CPU, 48 cores, 2.5GHz,共有キャッシュ
    - シンプルな実行コア(in-orderでSMT機能付き)
    - メモリバンド幅を最大限に活用するアーキテクチャ
    - SIMDベクトルユニット.... 8 FLOP / cycle / core
- 複数の計算ノードがNUMA構成で接続
- 複数の計算サーバ、ブレードでクラスタ構成

➡ 効率の良い並列化が求められる

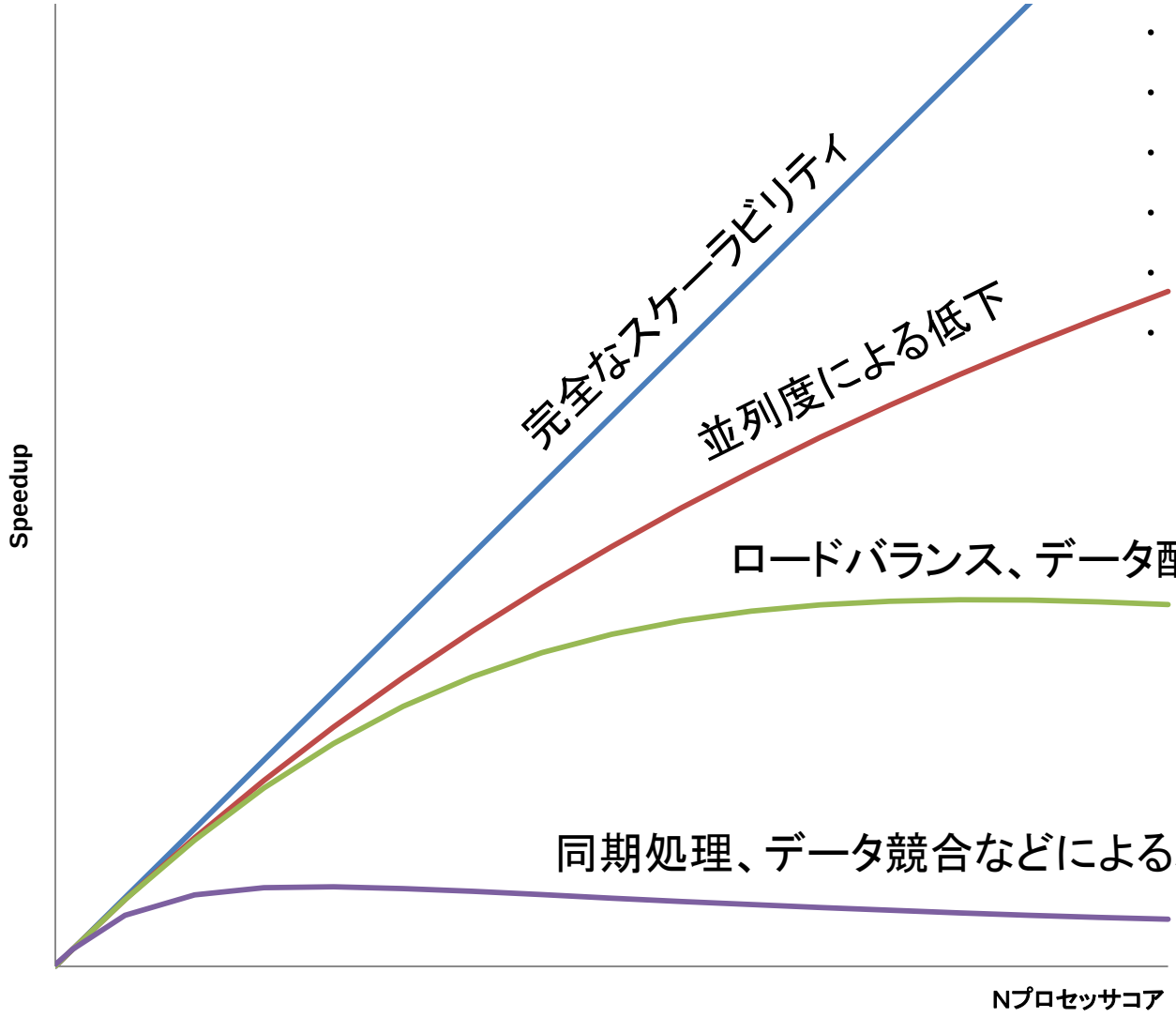
1) 仮定として想定したプロセッサに基づく推察

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。

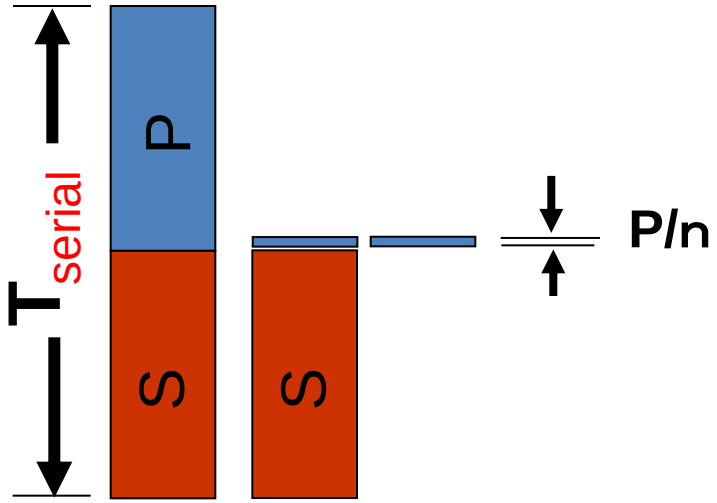
並列プログラミングで留意点



- 十分な計算量
- 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

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）



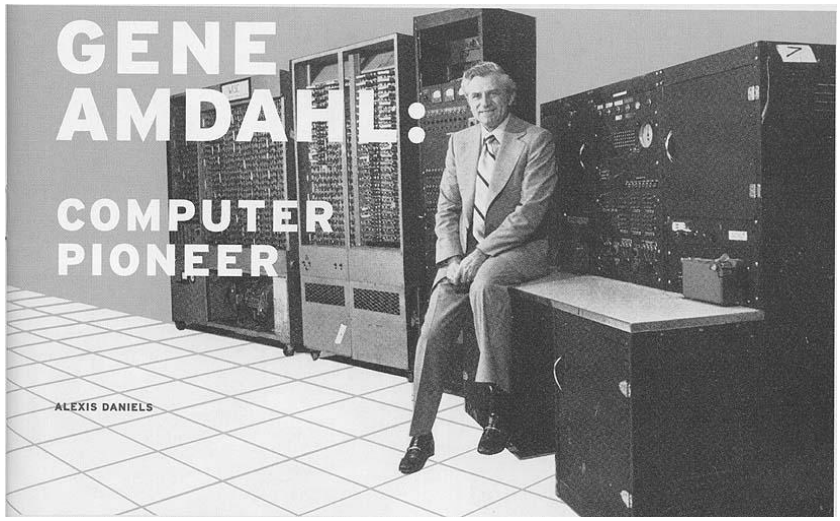
$$T_{parallel} = (S + P/n)\, T_{serial} + O$$

n = number of processors

$$Speedup = T_{serial} / T_{parallel} = 1 / (S + P/n)$$

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
Speedup = 1.0 / (0.5 + 0) = 2.0



グスタフソンの法則

●アムダールの法則
作業負荷や問題の規模が一定であることを仮定

●グスタフソンの法則（ Gustafson-Barsis‘ law）
並列処理では問題の規模や作業負荷がプロセッサコア数に比例して大きくなり、その負荷増加は逐次処理部分に影響しないことを仮定

「アムダールの法則の限界から並列処理を救い出すことが可能？」

適用出来る問題と利用環境に大きな制限がある

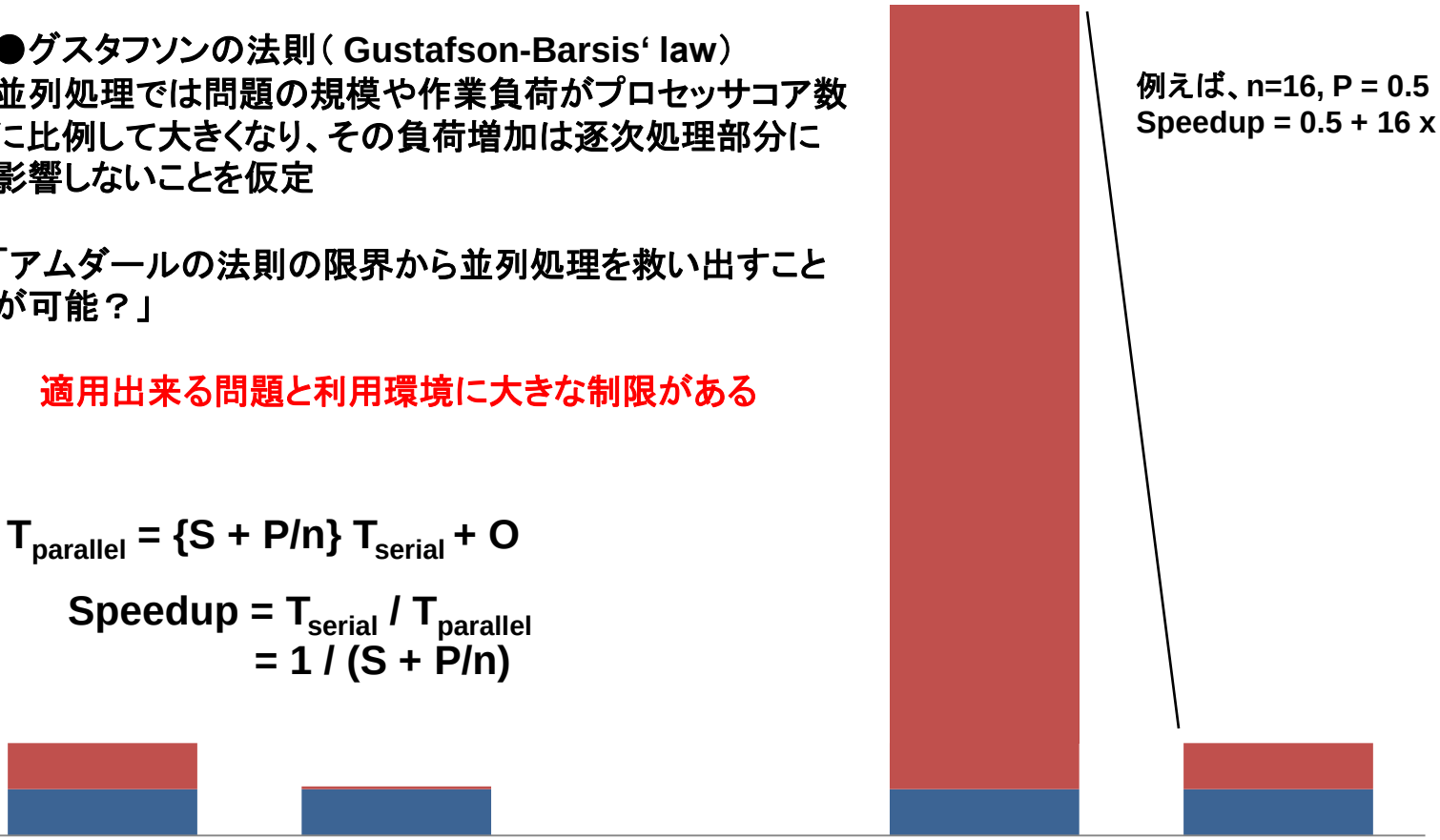
$$T_{parallel} = \{S + P/n\} T_{serial} + O$$

$$Speedup = T_{serial} / T_{parallel} = 1 / (S + P/n)$$

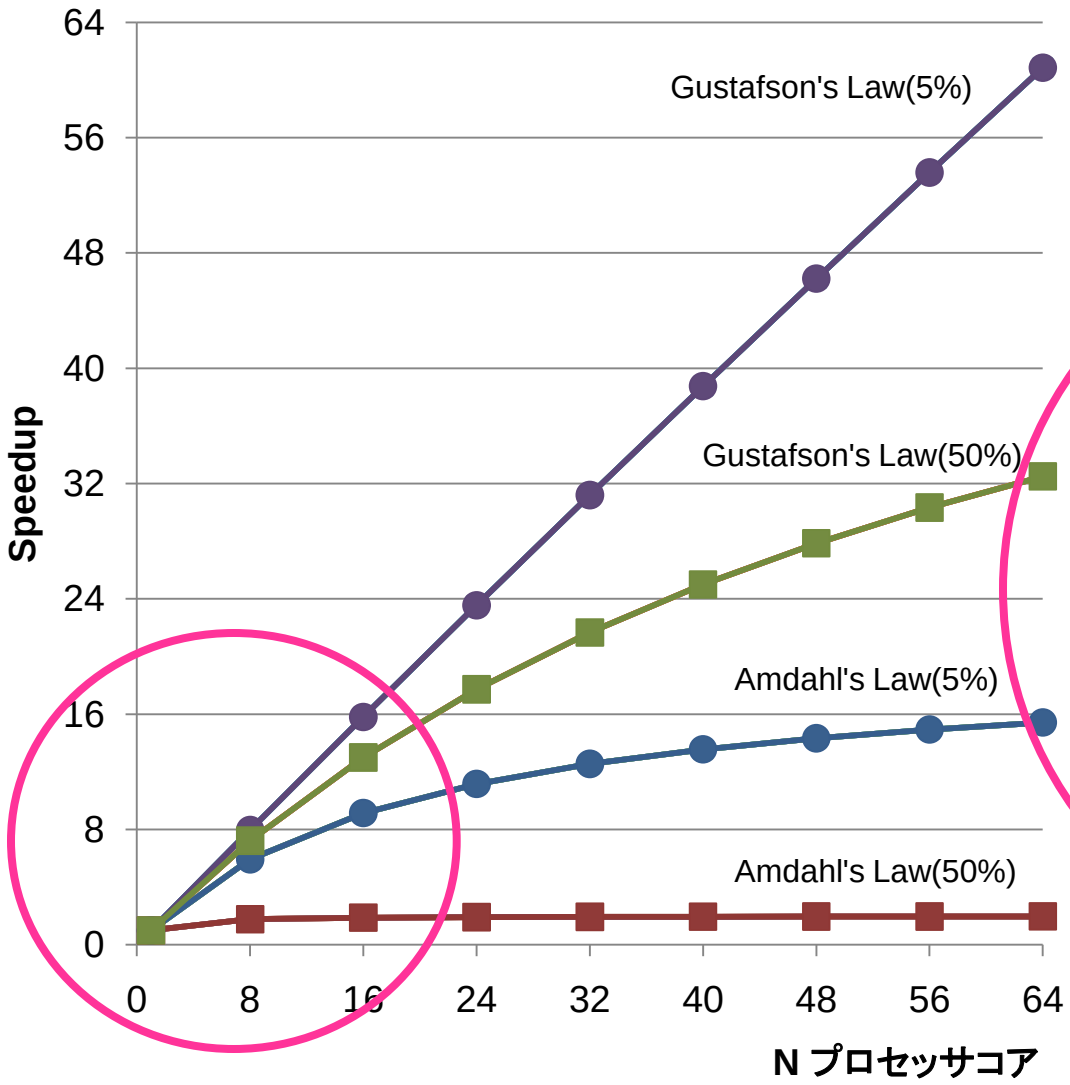
$$T_{serial} = (S + n \times P) T_{parallel}$$

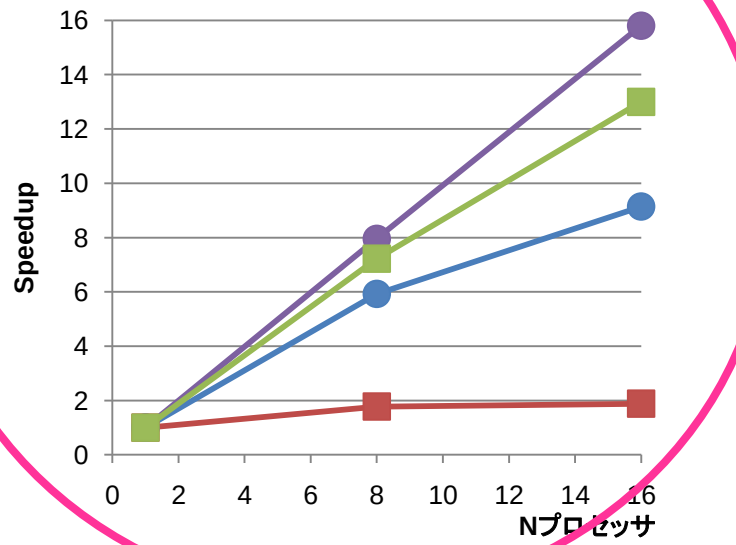
$$Speedup = T_{serial} / T_{parallel} = (S + n \times P)$$

例えば、n=16, P = 0.5 の場合
Speedup = 0.5 + 16 x 0.5 = 8.5

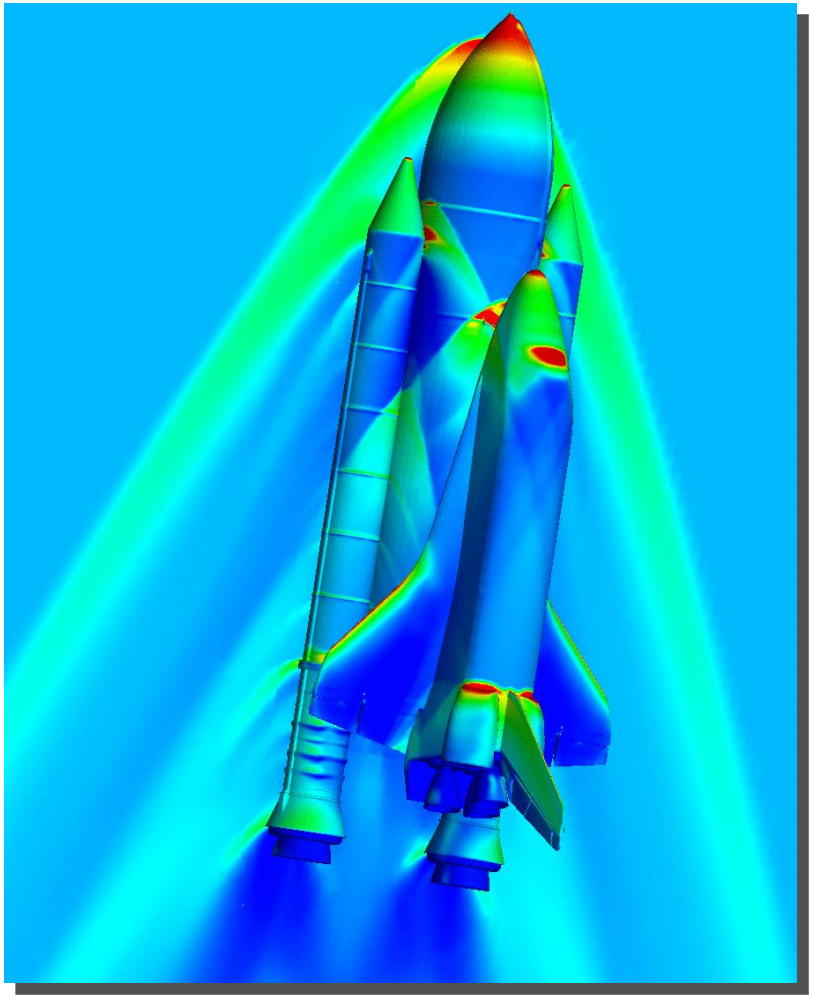


スケーラビリティ

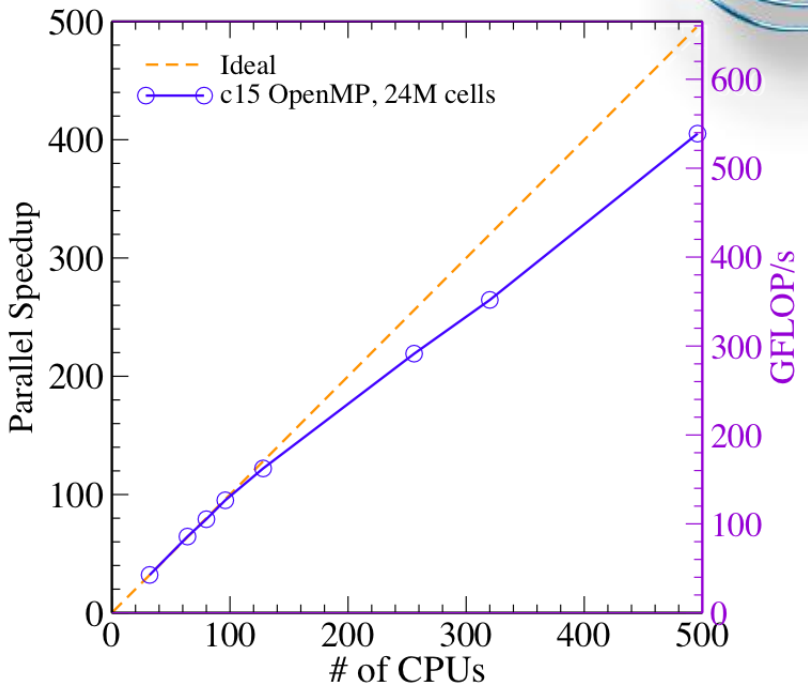

64
56
48
40
32
24
16
8
0
Speedup
Gustafson's Law(5%)
Gustafson's Law(50%)
Amdahl's Law(5%)
Amdahl's Law(50%)
0
8
16
24
32
40
48
56
64
N プロセッサコア


16
14
12
10
8
6
4
2
0
Speedup
0
2
4
6
8
10
12
14
16
Nプロセッサ

NASAによる流体解析コード



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



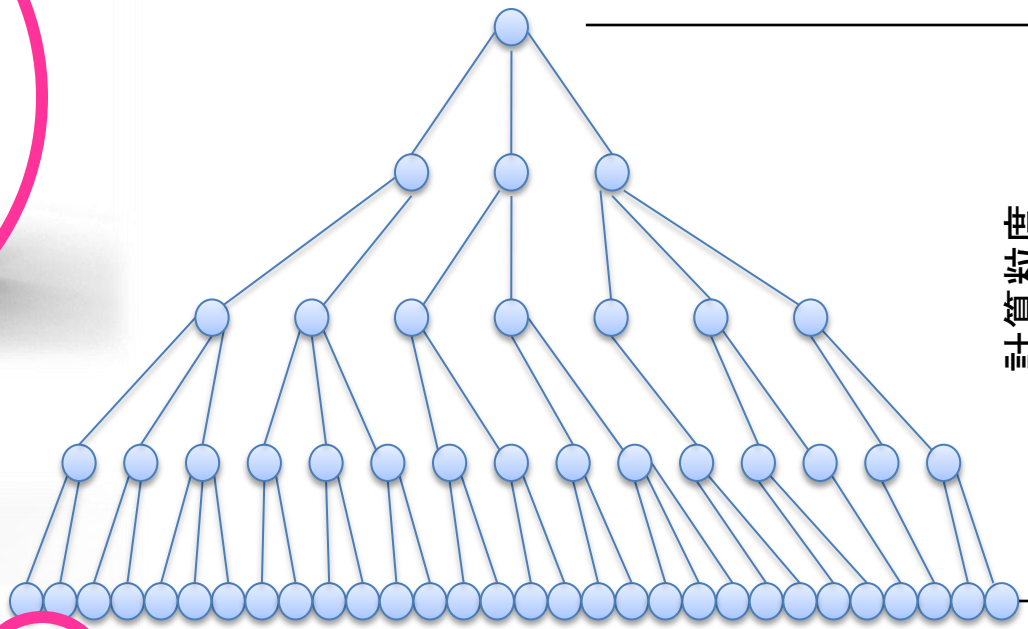
並列性能

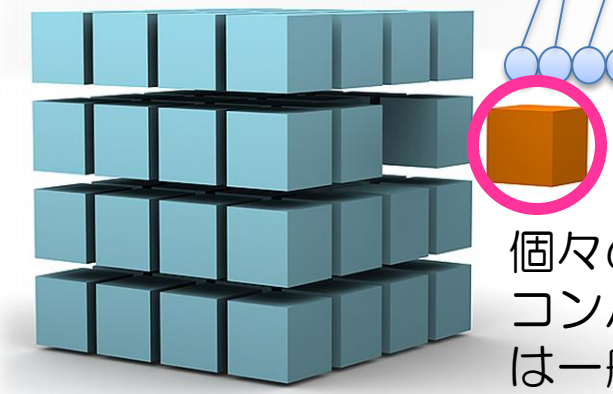
- 496プロセッサで405倍の性能向上が可能
  - 540 GFLOP/s
  - CPUあたりの性能：1.33 GFLOP/s
- 短時間でのシュミレーションを可能とし、問題への緊急的な対応を可能となります。

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

シングルAPIでの並列処理

	OpenMP	MPI
ノード内	○	○
ノード間	???	○

MPI
OpenMP

Vertical Scaling

16P+ / 32P+
8P / 16P
4P / 8P
2P / 4P

1ノード　2ノード　4ノード　8ノード　・　nノード

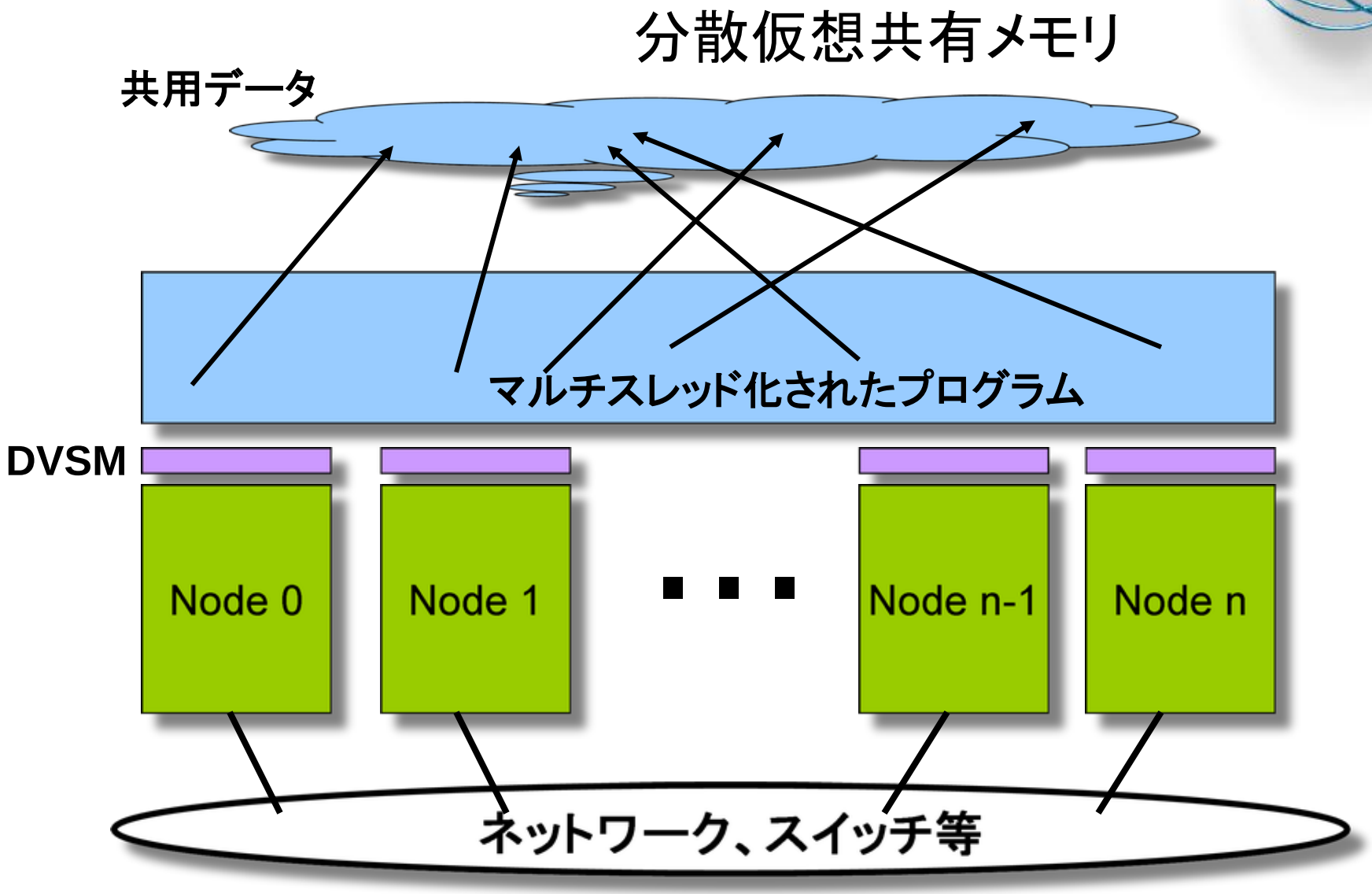
Horizontal Scaling

MPI
OpenMP????

OpenMPの価値

- シュミレーションでのソフトウエア開発
  - 大規模なシュミレーションを行うアプリケーションは、C++、C、Fortranで記述されている
  - ソフトウエア開発と利用は数十年単位で継続して行われる
- クラスタやスーパーコンピュータ
  - MPIを利用した大規模並列処理が一般的
  - MPI利用の限界と課題
    - 全てのアプリケーションに適用出来る訳ではない
    - スケーラビリティは様々な制限を受ける
    - “many cores” への対応がMPIタイプのAPIでは非常に難しい
  - OpenMPはこのようなMPIに対する他の選択肢の提供と同時にMPIを補完する役割を担う（MPI + OpenMP）

インテルクラスタOpenMP



一般的OpenMPの課題

- OpenMP版のコンパイル時の問題
  - OpenMP構文に基づく並列化によって、マイクロプロセッサ向け最適化が阻害される
- 実行時ライブラリでのオーバヘッド
  - 頻繁なライブラリ呼び出しの悪影響
- アルゴリズムの変更のオーバヘッド
  - プログラム並列化のためのコードの冗長化やコードの追加
- 同期処理
  - Fork-Join モデルによる過大な同期処理
  - 負荷分散
- メモリ階層の有効活用
  - キャッシュ、ローカルメモリ、リモートメモリの参照頻度

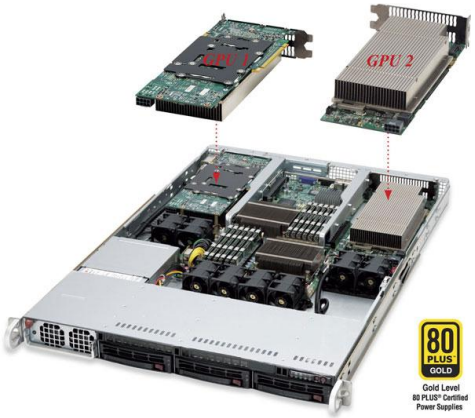
OpenMPの課題

- ハードウエアの動向
  - 今後は複数ソケットの製品はすべてNUMAアーキテクチャ
- OpenMP 3.0リリース
  - NUMA対応の拡張無し
  - アフィニティ問題
    - データの配置及び利用とスレッド実行の管理

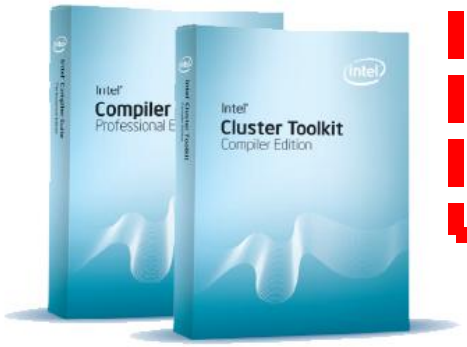
OpenMP 3.0: The World is still flat, no support for cc-NUMA (yet)!
http://terboven.wordpress.com/category/openmp/

ハイブリッド:現代のトレンド？

・ハイブリッドカー
  - 内燃機関動力（ガソリンエンジンやディーゼルエンジン）と蓄電池



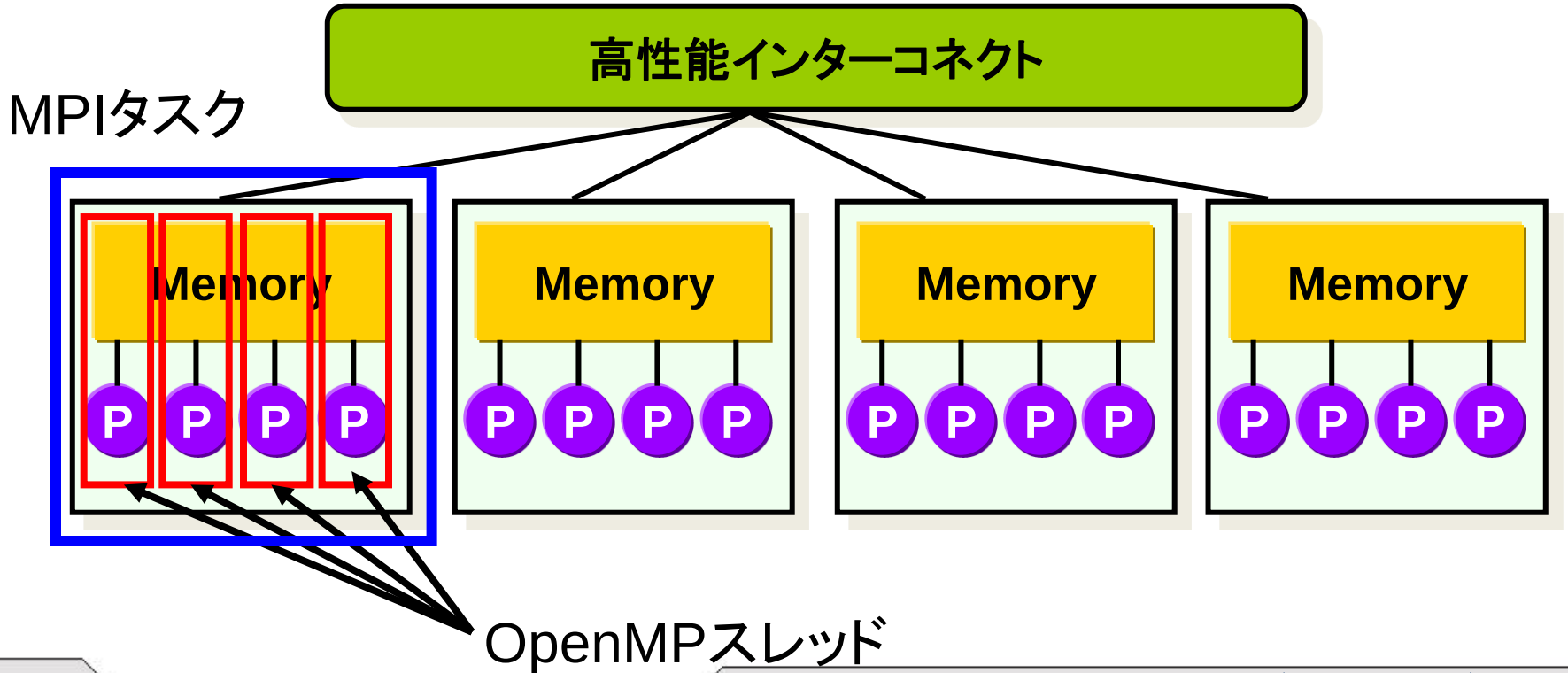
・ハイブリッドコンピューティング
  - GPU+CPUによるハイパフォーマンスコンピューティング



・ハイブリッド並列プログラミング
  - スレッドプログラング+MPI

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

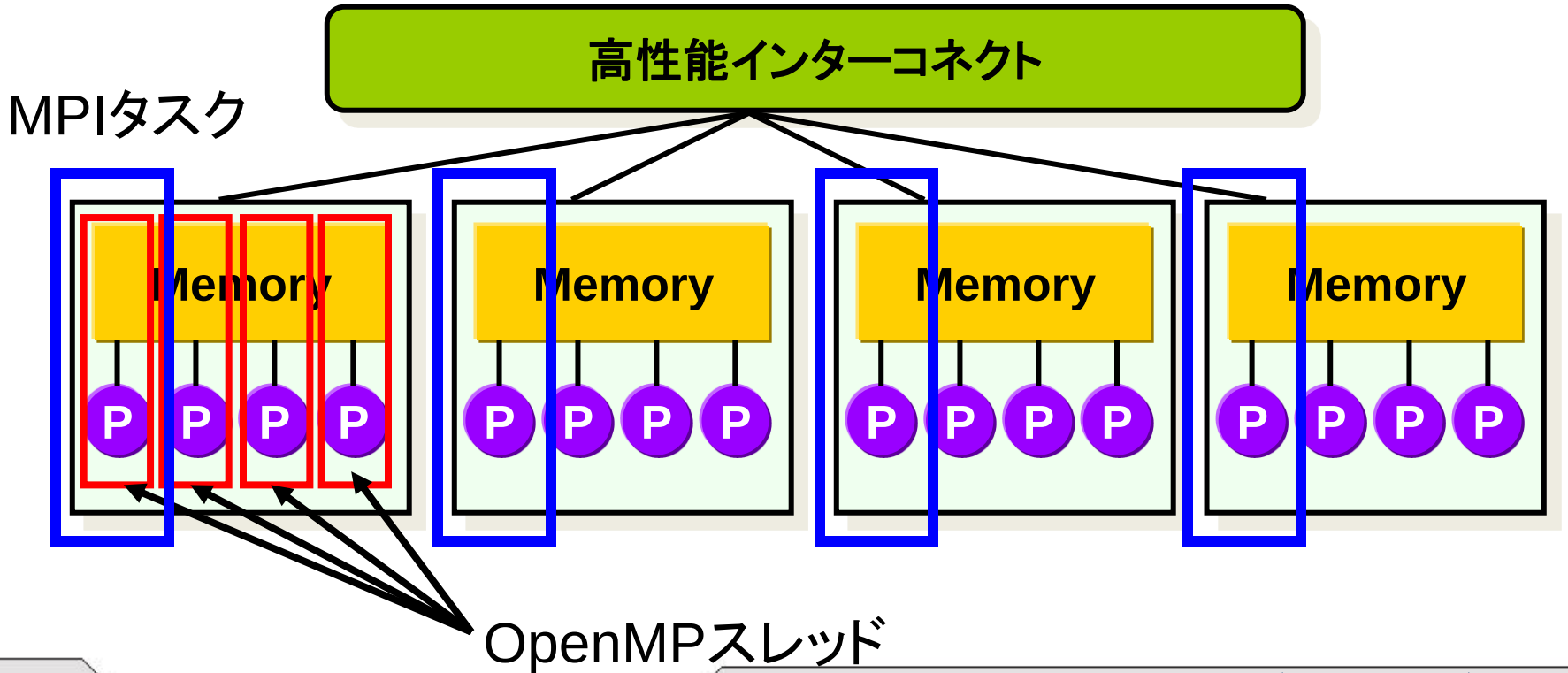
```
include ‘mpif.h’
program hybsimp


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

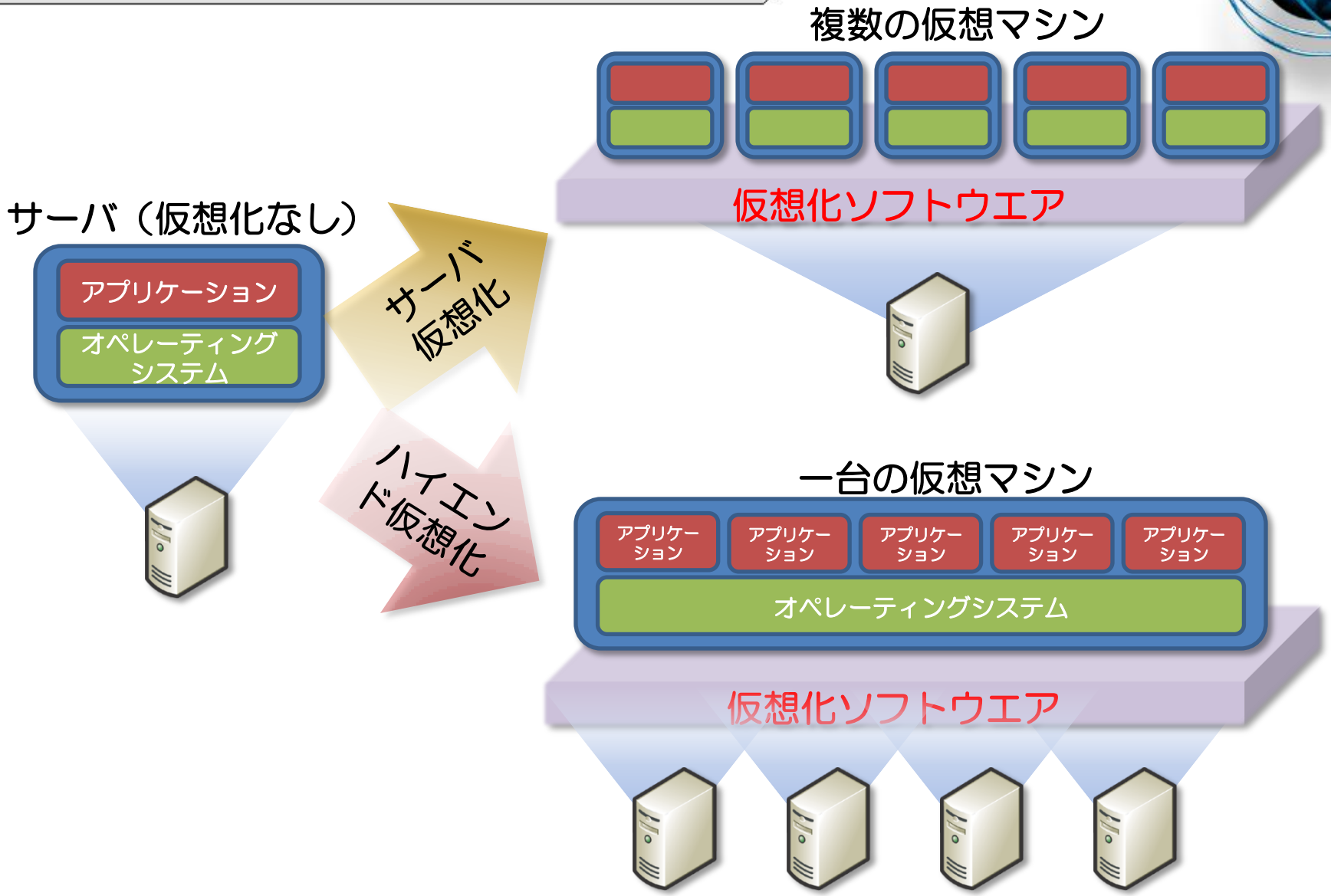
Fortran

```
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++

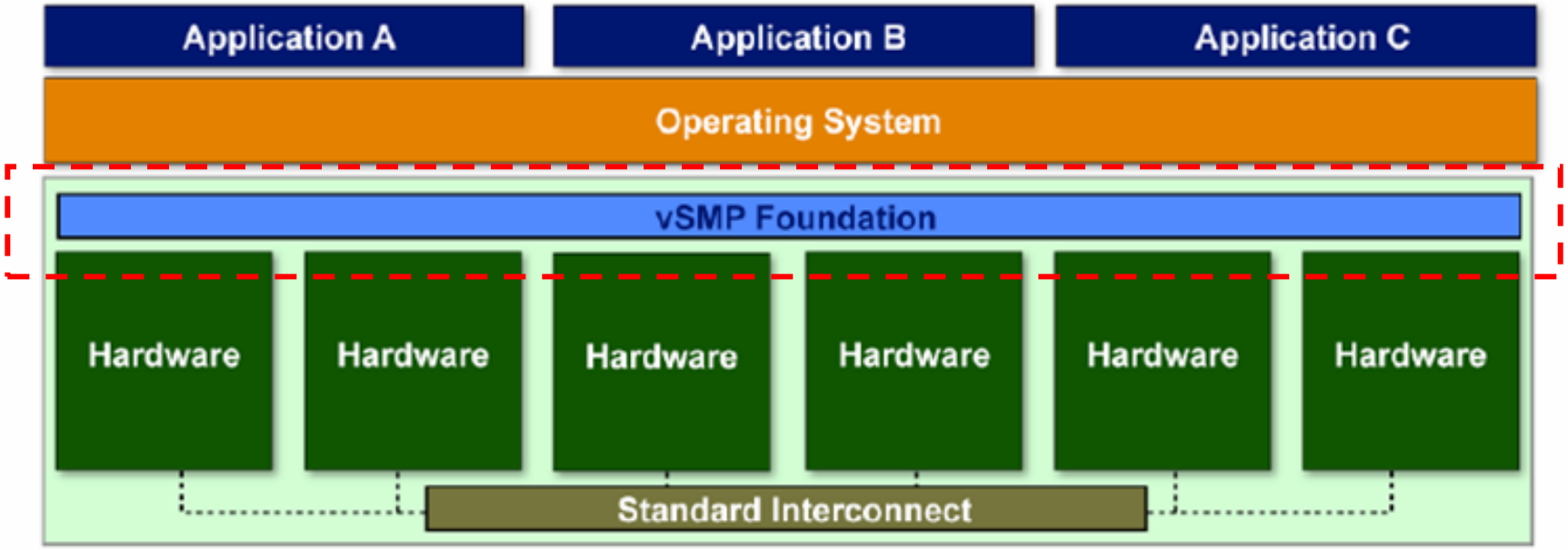
```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;


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

}
```

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
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
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



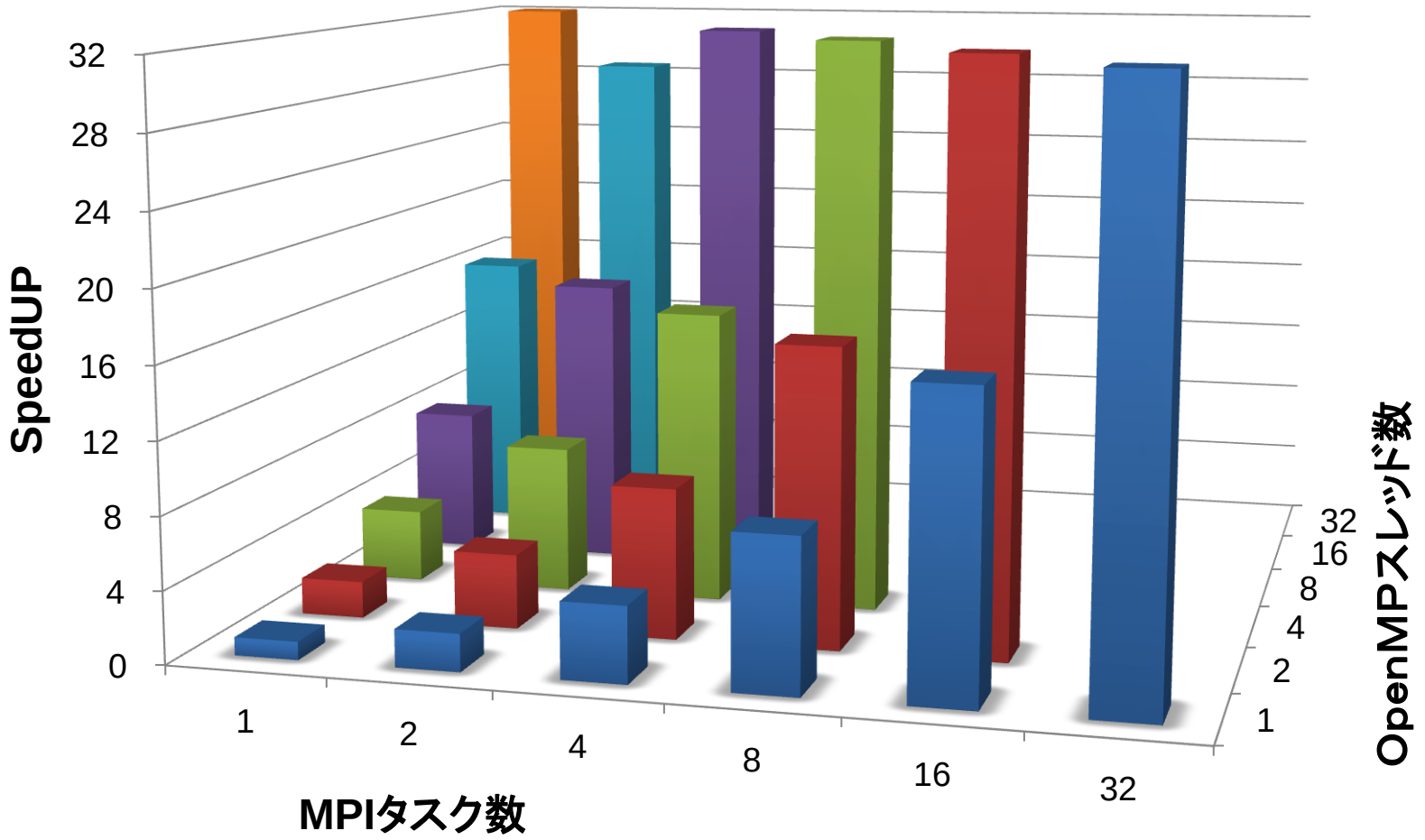
Hardwareは一般のx86チップセットと標準インターコネクトでシステムの構築が可能

vSMP Foundation でのシステムのSMP拡張を実現

OpenMP/MPI/ハイブリッド

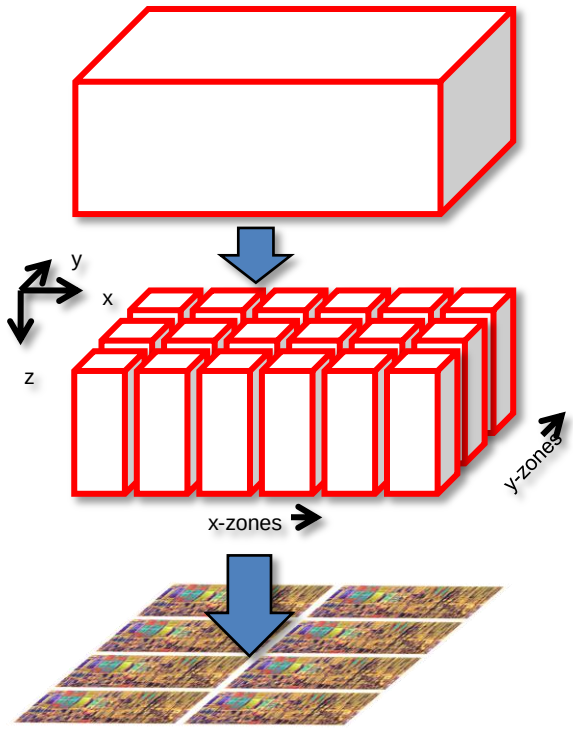
Hybrid OpenMP MPI Benchmarkproject (″homb″)

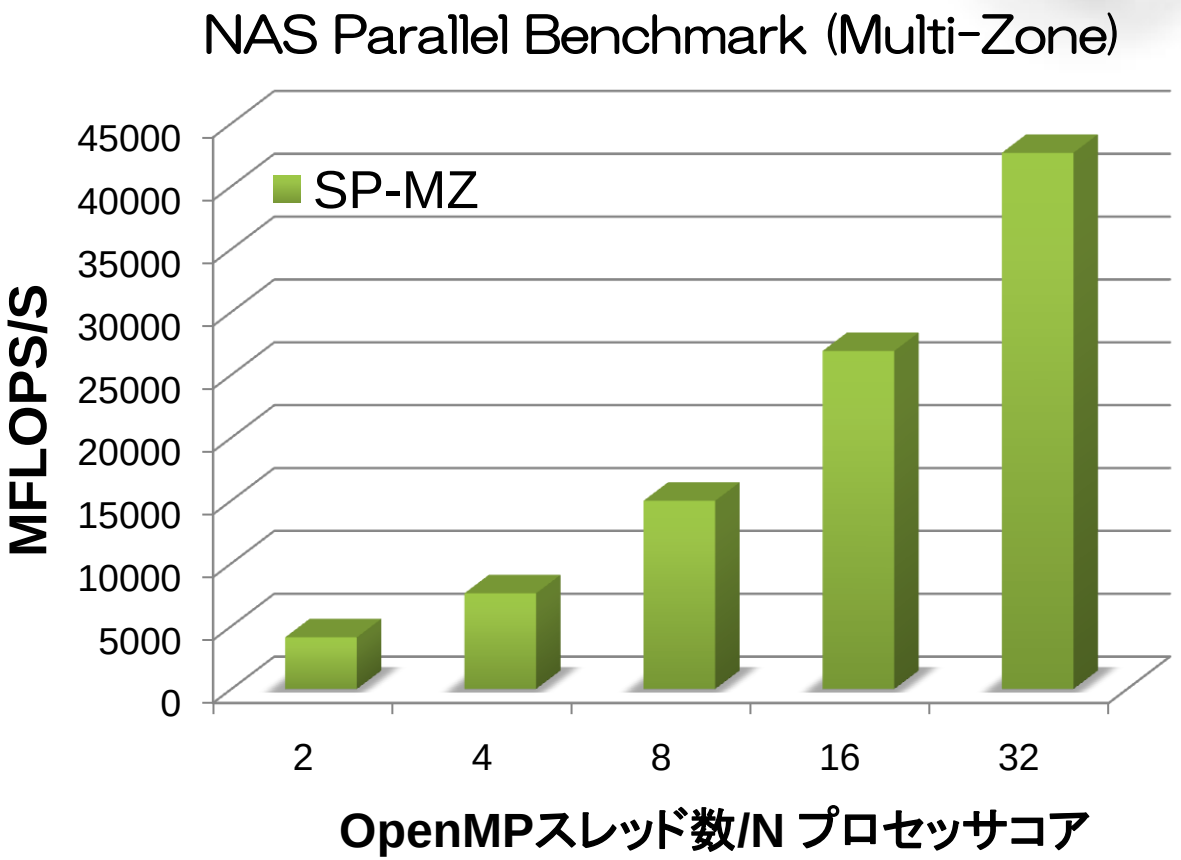
This is the Hybrid OpenMP MPI Benchmarkproject (″homb″) This project was registered on SourceForge.net on May 16, 2009, and is described by the project team as follows: HOMB is a simple benchmark based on a parallel iterative Laplace solver aimed at comparing the performance of MPI, OpenMP, and hybrid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



Xeon 5550 (2.66GHz) vSMP Foundation

OpenMPベンチマーク

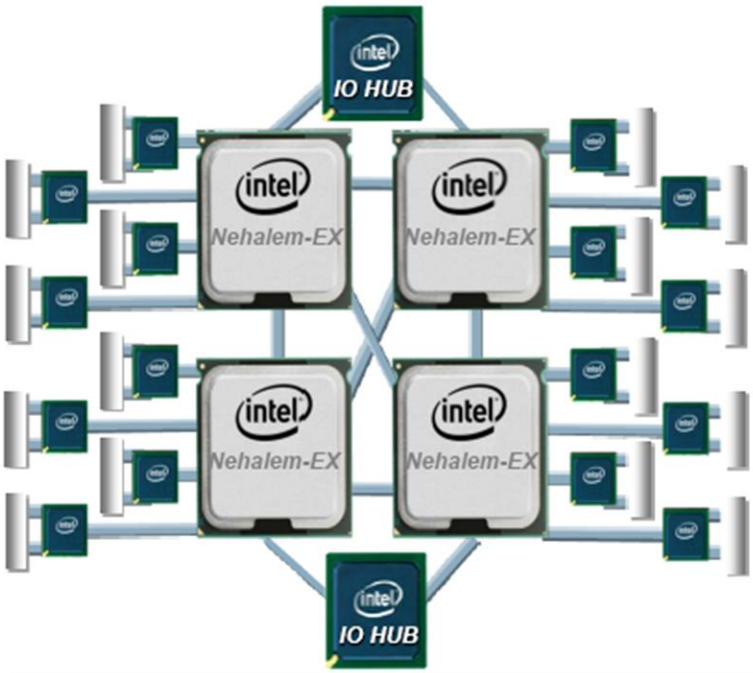




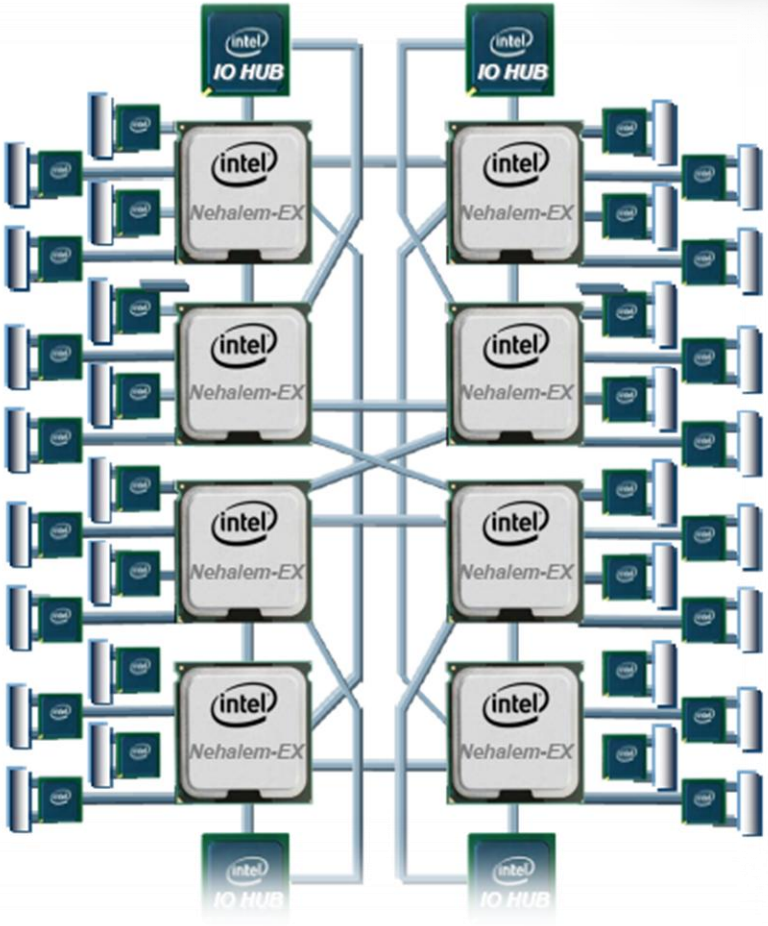
著名な公開ベンチマークツールである NAS Parallel Benchmark (NPB) の一つであるNPB-MZ (NPB Multi-Zone)はより粒度の大きな並列化の提供を行っています。NPB-MZでは、ハイブリッド型の並列処理やネストしたOpenMPのテストが可能です。ここでの結果は、OpenMPだけでの並列処理の性能を評価しています。
Xeon 5550 (2.66GHz) vSMP Foundation

Nehalem-EX トポロジ

IDF2009
INTEL DEVELOPER FORUM



4プロセッサトポロジ
32プロセッサコア

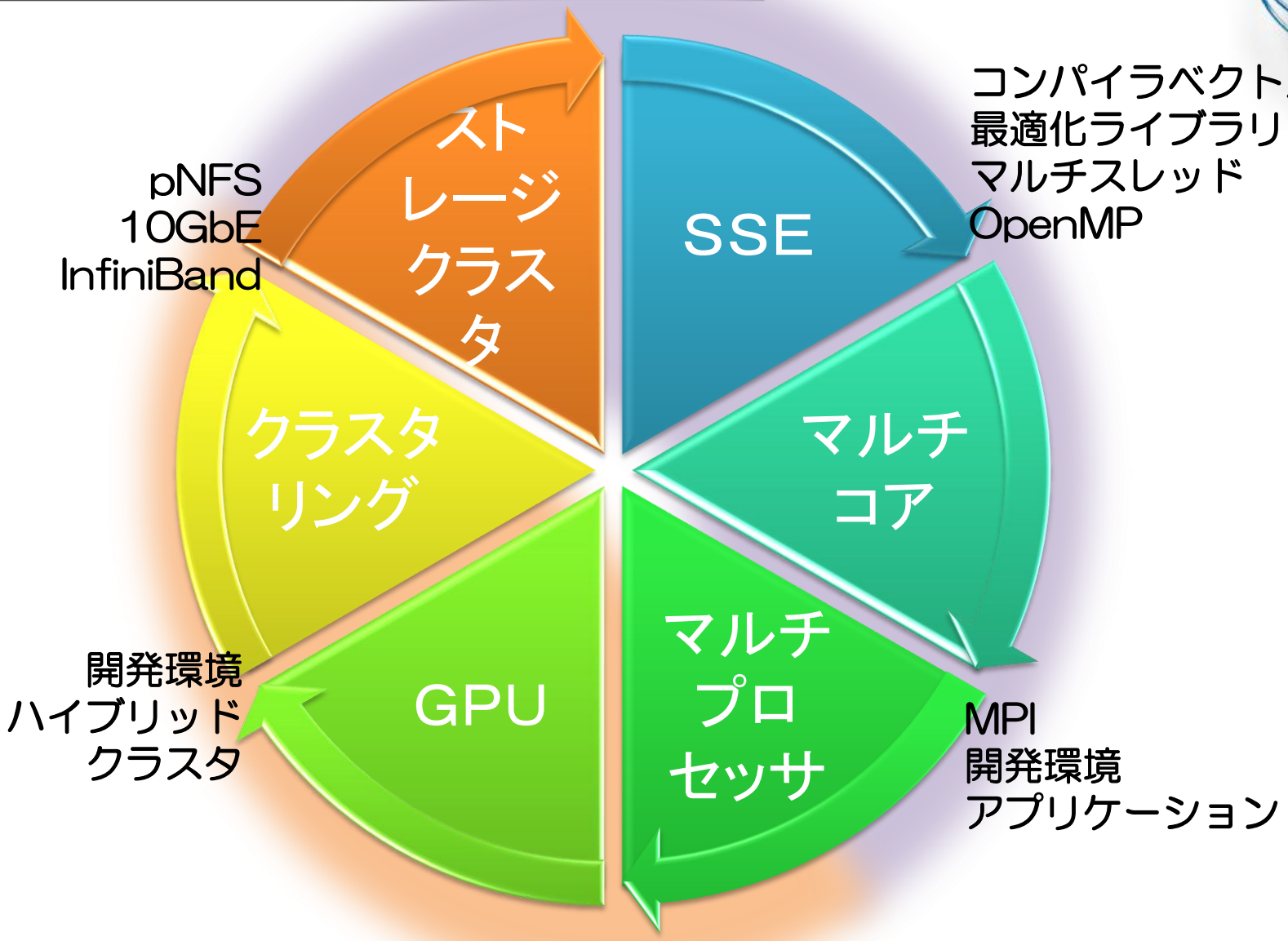


8プロセッサトポロジ
64プロセッサコア

ユビキタス並列処理プログラミング

まとめとして

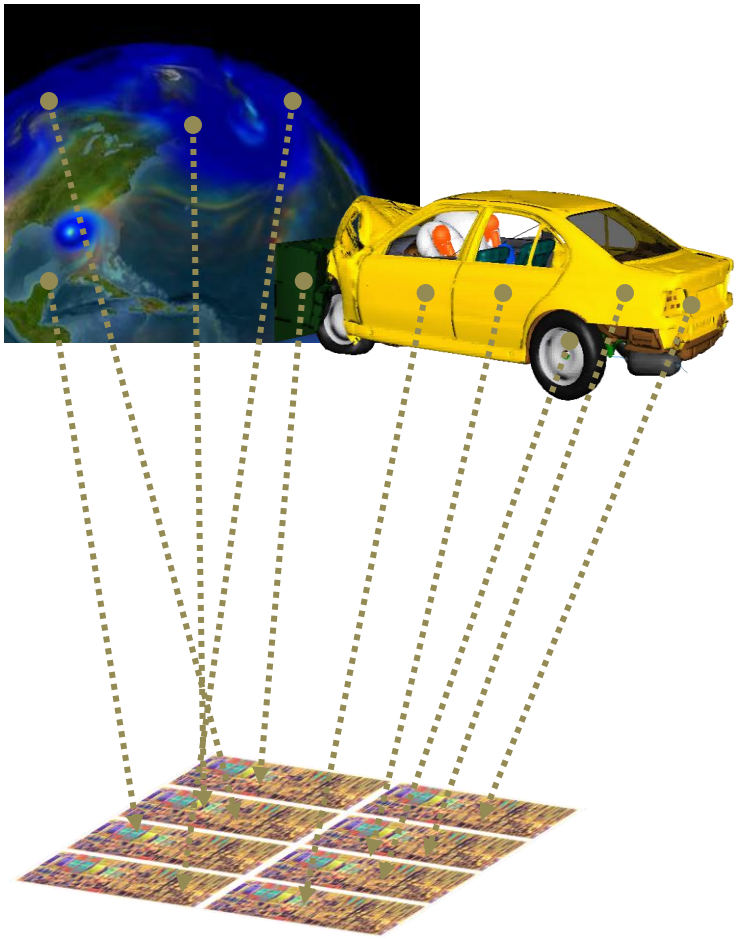
並列処理はITの根幹


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（並列処理）

- 並列処理技術の課題
  - Peta-Scaleコンピューティングと Commodity コンピューティングでのギャップの克服
- マイクロプロセッサと並列処理
  - マイクロプロセッサでの並列実行ユニットの増加
  - ベクトル演算（x8）、プロセッサコア（+6以上）
  - プロセッサ内部での並列処理の重要性
- スケーラブルCommodityコンピューティング
  - 今後の技術的な課題と幾つかの提案・試行
    - クラスタOpenMP
    - ハイブリッド並列処理
    - SMP仮想化

まとめとして（マルチスレッド）



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
・・・・・・

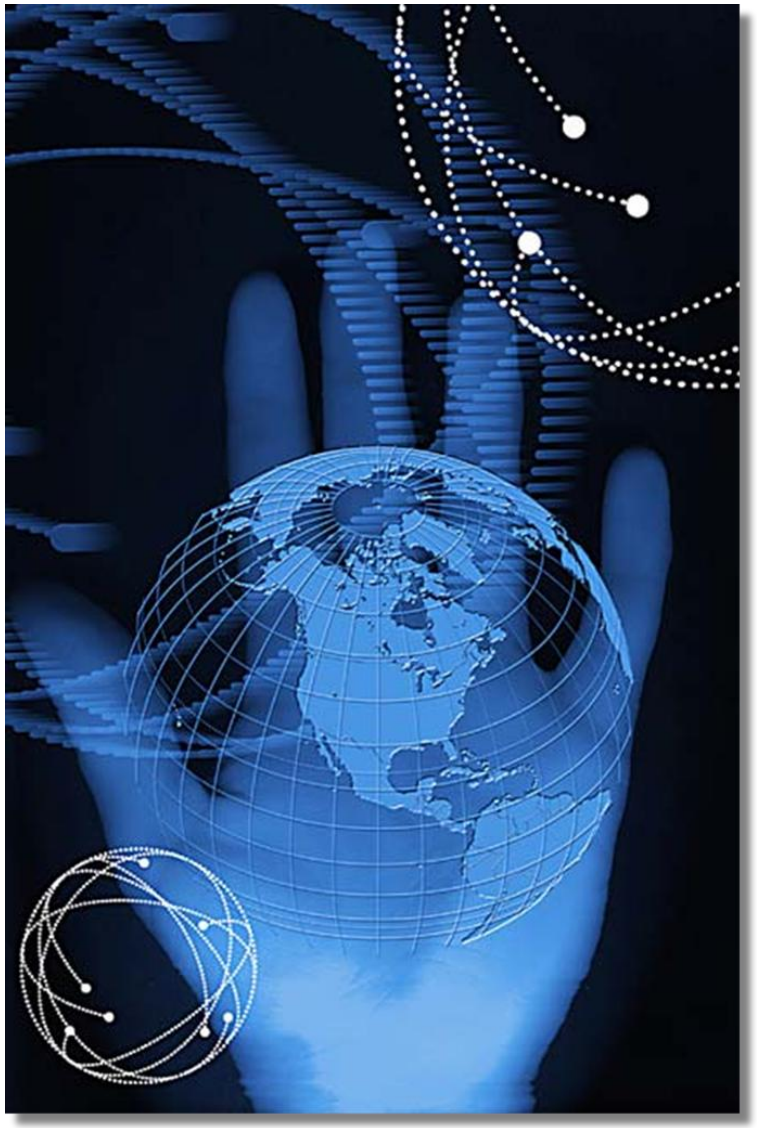
↓

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

この資料について



ここに掲載した資料は、弊社の調査と見解に基くものであり、資料の中で示されている製品やサービスを提供している各社の公式な見解でも、また、マーケティング戦略に基くものではありません。 あくまで、弊社としての意見だということにご注意ください。
本資料は情報提供のみを目的として作成されたものであり、商品の勧誘を目的としたものではありません。また、本資料は弊社が信頼できると判断した各種データに基づき作成されておりますが、その正確性、確実性を保証するものではありません。本資料に記載された内容は予告なしに変更されることもあります。
これらの資料の無断での引用、転載を禁じます。
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。なお、本文中では、特に® 、TMマークは明記しておりません。

2009年9月