OpenMP
プログラミング・ワークショップ

平成13年3月22日

日本SGI株式会社
製品技術本部
スケーラブルシステムテクノロジーセンター
芦澤　芳夫

OpenMPの概要

共有メモリ型並列化APIの必要性

- 標準化による利点
  - 利用者
    - 異機種間の移行が容易
  - ソフトウェアベンダ
    - 移植性、保守性、品質向上
- APIの標準化が遅れた理由
  - 各ベンダが独自のAPIを提案
    - 同様の機能を各社各様の指示行で実現
  - X3H5/PCFの失敗
    - 各社の意向の相違、仕様自身の機能不足
  - MPIという移植性の高いAPIの存在

OpenMPの利点

- 移植性
  - 共有メモリ型プログラミングの標準API
- 幅広い並列構造の記述
  - コンパイラ指示行によるループレベルの並列化
  - 関数呼び出しを含む高いレベルの並列化
  - スレッド生成によるMPIと同様の並列構造の記述
- 共有メモリ型プログラミングの活用
  - 段階的な並列化が可能
  - 既存資産への適用が容易

OpenMPの特徴

- 移植性の高いAPI
  - Fortran77(1.1), C/C++(1.0)に関する仕様
  - Fortran90対応、機能追加 (2.0)
- 機能
  - コンパイラ指示行による並列化構造の記述
  - 細粒度(ループレベル)
  - 粗粒度(関数呼び出しを含む)
  - スレッド、プロセスの制御

OpenMPの仕様

- OpenMPの仕様のリリース
  - Fortran　version 1.0　　1997年10月
  - C/C++　version 1.0　　1998年10月
  - Fortran　version 1.1　　1999年11月
  - Fortran　version 2.0　　2000年11月
- 仕様、公開情報
  - http://www.OpenMP.org/
  - 仕様書、解釈
  - チュートリアル、プログラミング例
  - ワークショップ

ARBメンバー

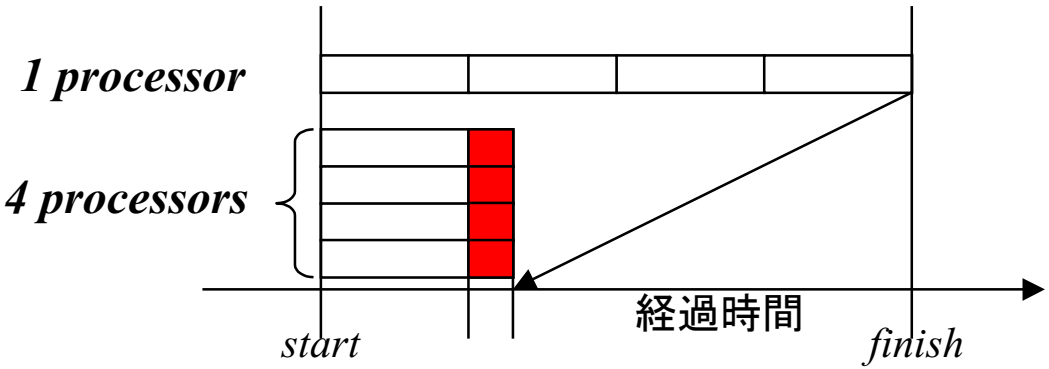
- Architecture Review Board メンバー
  - 米国エネルギー省 ASCIプログラム
  - Compaq
  - 富士通　2000年10月　米国外のベンダとして初
  - HP
  - IBM
  - Intel
  - KAI　2000年4月　Intelの子会社
  - SGI
  - Sun

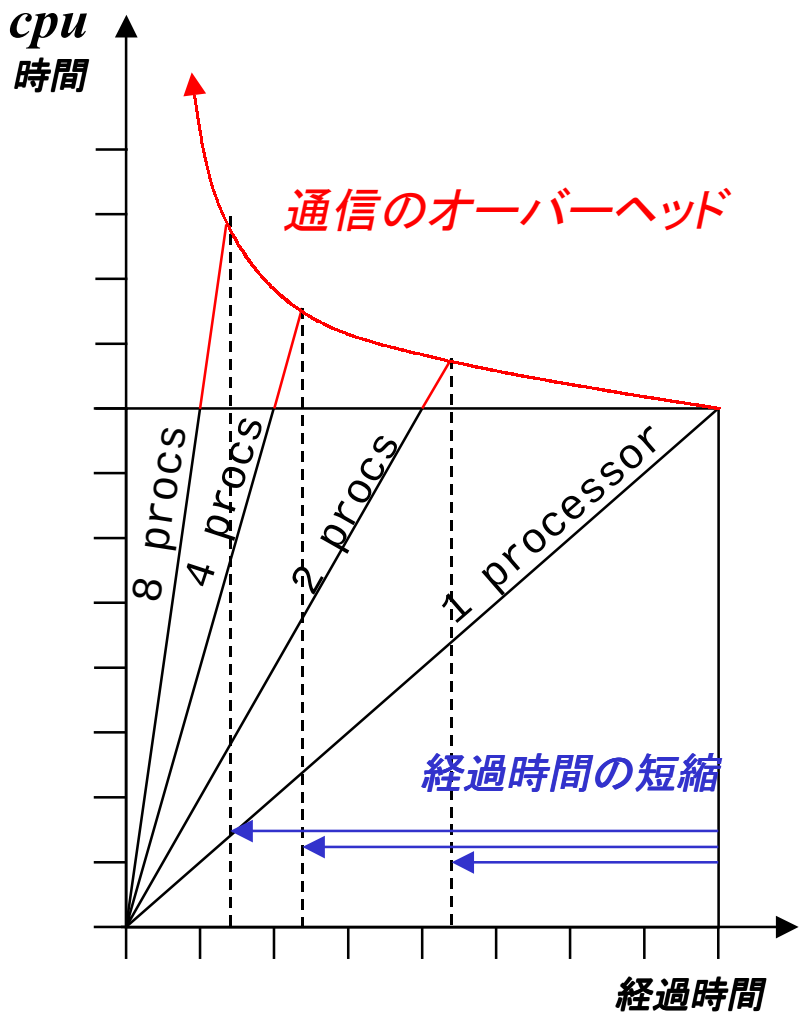
OpenMPの構成要素

- 並列化指示行(!$omp, #pragma)
  - 並列化構造の記述
  - ワークシェアリング
  - データのスコープ
  - 同期
- 実行時ライブラリ
  - 実行制御ライブラリ(スレッド数、ダイナミックスレッドなど)
  - 基本的な関数(スレッド数、スレッド番号の取得など)
- 環境変数
  - 最大スレッド数
  - ダイナミックスレッドなど

プログラムの並列化

- プログラムの経過時間の短縮

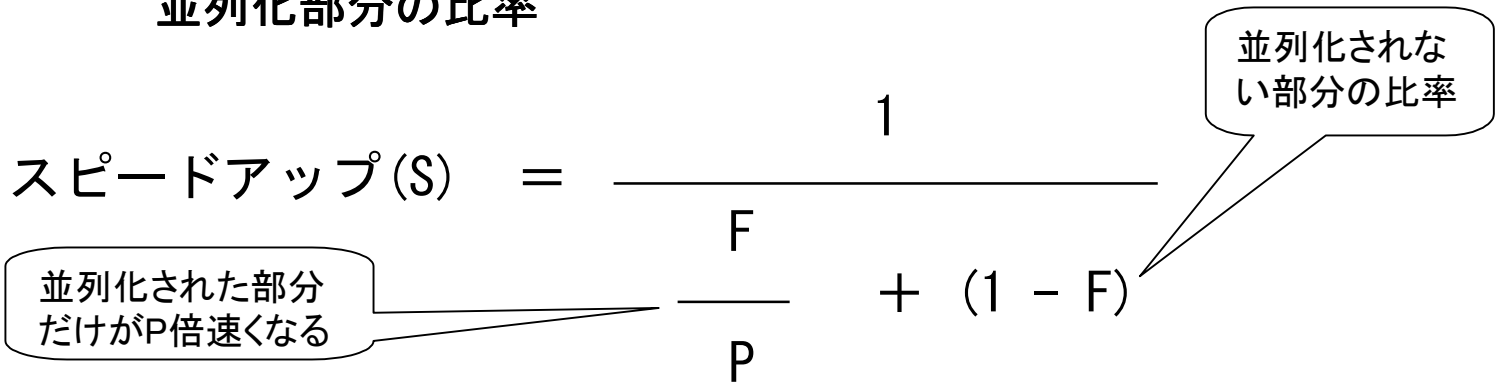




- オーバーヘッド:
  - 通信
  - 同期
  - アルゴリズムの変更に伴う処理の増加
  - コードのなかの並列化できない部分
  - 全体のcpu 時間は増加

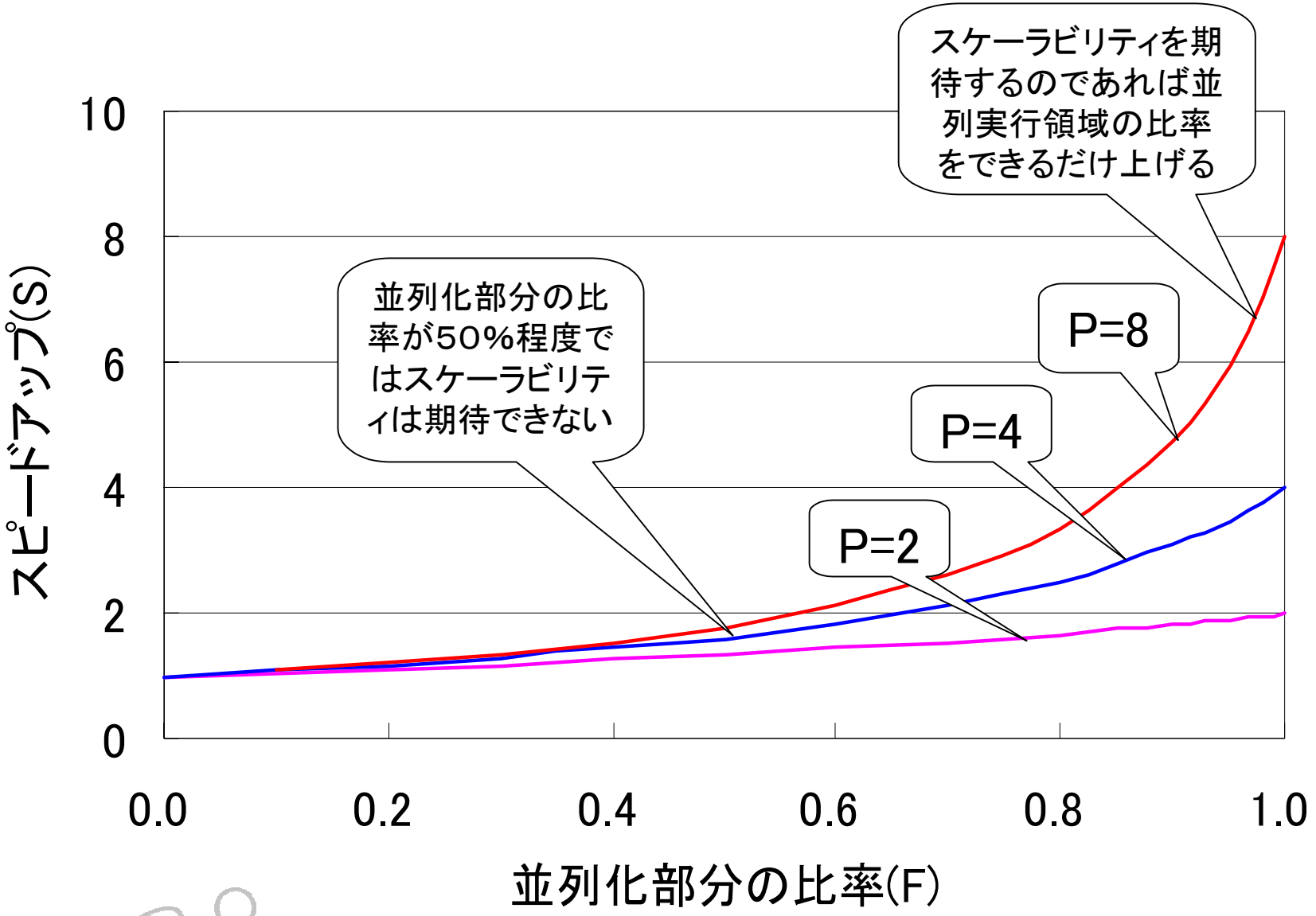
アムダールの法則

- 並列化されない部分の比率(1−F)によって並列実行によるスピードアップ(S)の上限が決まる
  - S　　スピードアップ
  - P　　プロセッサ数
  - F　　並列化部分の比率



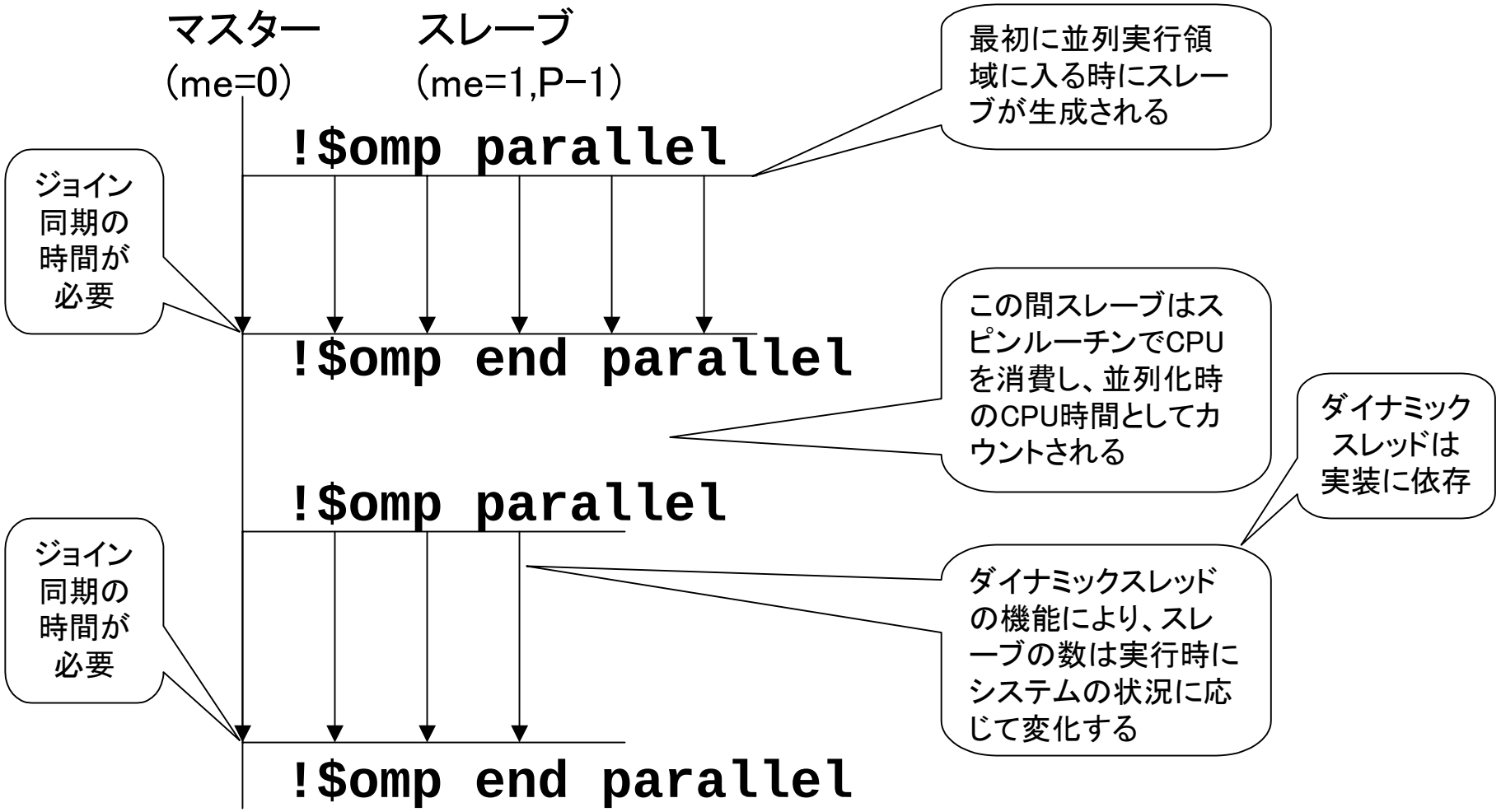
- プログラミングモデル(MPI/OpenMPなど)に依存しない
- 考慮していないこと
  - 通信同期のオーバーヘッド
  - 並列化の粒度(細粒度、粗粒度)

アムダールの法則


スケーラビリティを期待するのであれば並列実行領域の比率をできるだけ上げる
並列化部分の比率が50%程度ではスケーラビリティは期待できない
P=8
P=4
P=2
スピードアップ(S)
10
8
6
4
2
0
0.0
0.2
0.4
0.6
0.8
1.0
並列化部分の比率(F)

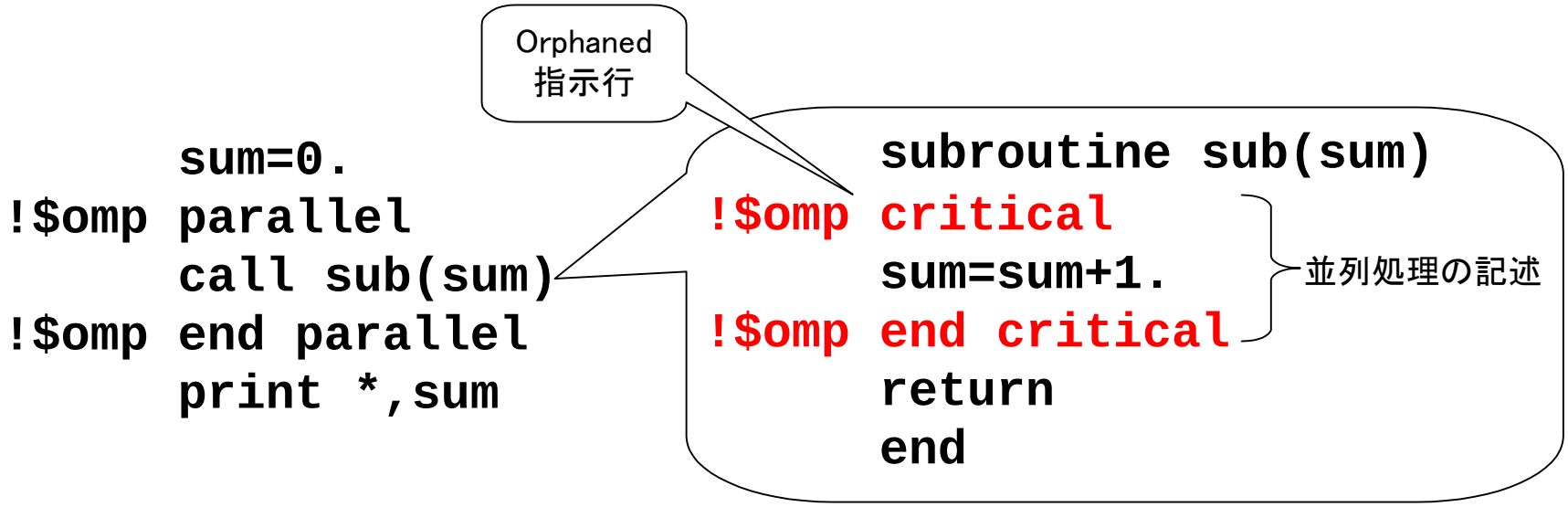
OpenMPの実行モデル(SGI)

- 並列実行のfork-joinモデル



関数レベルの並列構造の記述 Orphaned指示行

- 粗粒度の並列処理構造を実現
  - 並列実行領域から呼び出される関数内で使用されるOpenMP指示行の総称
  - 異なったコンパイルユニット間での同期処理が可能
  - X3H5/PCFでは考慮されていなかった構造



ダイナミックスレッド

- 機能
  - 並列化プログラムを実際の運用環境で効率的に実行するための機能
  - CPU数以上のプロセスの同時実行を回避
    - システムのスループットの向上
    - システム利用効率の向上
- OpenMPの実行時ライブラリ
  - 各並列実行領域の開始時点で、スレッド数を調整し、その並列実行領域が終了するまでは同じ値が保証される
  - SGIの実装
    - デフォルトでダイナミックスレッドが有効

OpenMPプログラミング

自動並列化とOpenMP

- コンパイラがコードをうまく並列化してくれれば. . .
  ユーザはOpenMPを知らなくてもOK

  しかし、. . .

- 実用コードではコンパイラだけではうまく並列化されない場合がある
  自動並列化とOpenMP指示行との併用も可能

例えば、

```
do  i = 1, n
      A(IB(i)) = ... ...
enddo
```

全てのイタレーションで IB(i) の値が異なっていれば並列化可能
- コンパイラは確認できない
- ユーザが確認できれば並列化指示行を指定

ループを並列化する指示行(1)

指示行に続くループを各スレッドに分散して並列処理

- Fortran
  - 大文字、小文字の区別なし
  - 1行に収まらない場合は、!$omp& を指定して次行に続ける(固定形式)

```
!$OMP PARALLEL DO PRIVATE(変数p1、変数p2、...) SHARED(変数s1、変数s2、...)
      DO I=i1,i2,i3
            block
      ENDDO
```

並列実行領域

同じ意味

```
!$OMP PARALLEL DO
!$OMP& PRIVATE(変数p1、変数p2、...) SHARED(変数s1、変数s2、...)
      DO I=i1,i2,i3
            block
      ENDDO
```

並列実行領域

ループを並列化する指示行(2)

指示行に続くループを各スレッドに分散して並列処理

- C/C++
  - 大文字と小文字の区別があり
  - 1行に収まらない場合は、＼(バックスラッシュ)で改行をエスケープ

```
#pragma omp parallel for private(変数p1、...) shared(変数s1、...)]
    for (i=0;i<n;i++) {
      ...
    }
```

並列実行領域

同じ意味

```
#pragma omp parallel for ＼
     private(変数p1、...) shared(変数s1、...)]
     for (i=0;i<n;i++) {
       ...
     }
```

並列実行領域

private変数とshared変数

並列実行領域内の変数の属性を定義:

- private
  - スレッド毎にプライベート
  - 新たに記憶領域が確保される
  - 並列実行領域内だけでアクセス可能
  - 初期値は未定義

- shared
  - 共有変数
  - すべてのスレッドから同じアドレスで参照可能

Doループ並列化の記述

インデックス変数はデフォルトでprivate省略可能

```
!$omp parallel do private(i, tmp) shared(max, A, B, C, n)
      do i = 1, n
         tmp = i + max
         C(i) = A(i) * B(i) + tmp
      enddo
      print *, 'C(n) = ', C(n)
      ... ...
```

デフォルトはsharedで省略可能

縮約(リダクション)演算の記述

演算の種類(+,−,*,MAX,MINなど)によりローカル変数に初期値が与えられる

```
      sum=0.
!$omp parallel do private(i) shared(A,n)
!$omp&           reduction(+:sum)
      do i = 1, n
         sum = sum + A(i)
      enddo
      print *, 'sum = ', sum
      ... ...
```

並列実行ループ

各スレッドがローカルsumを計算し、最後に加え合わせる

クリティカルセクションの記述

```
      sum=0.
!$omp parallel private(i,localsum)
!$omp&          shared(A,n,sum)
      localsum=0.
!$omp do
      do i = 1, n
         localsum = localsum + A(i)
      enddo
!$omp critical
      sum = sum + localsum
!$omp end critical
      end parallel
      print *, 'sum = ', sum
      ... ...
```

各スレッドのlocalsumを初期化

並列実行ループ

各スレッドが自分の担当分についてlocalsumを計算

並列実行領域

各スレッドが排他的に実行することでsumを計算

（注）!$omp parallel doは直後のループだけを並列処理する簡略な記述

SGI Originでのコンパイルと実行

SGI Originでのコンパイル方法

- コンパイラはOpenMP 1.1に準拠
- コンパイル方法
  - 自動並列化(自動＋指示行を有効にする)

```
% f90 -apo  prog.f
% cc -apo prog.c
```

  - 指示行だけを有効にする

```
% f90 -mp prog.f
% cc -mp prog.c
```

- 分割コンパイル

```
% f90 -c -mp prog1.f
% f90 -c -apo prog2.f
% f90 -mp prog1.o prog2.o
```

並列化制御のライブラリをリンクするため-mpが必要

コンパイル時の-mpと混同しないこと

OpenMPプログラムの実行

- 環境変数に実行プロセッサ数を指定

```
% setenv OMP_NUM_THREADS 4
% a.out
```

- 実行プロセッサ数のデフォルト
  8プロセッサ

プログラム実行時の注意点

- 必要以上にCPU数を使用しない
  - オーバヘッドによるCPU時間の増加
- 使用するプロセッサ数
  - 必ずしも指定した値で実行されるとは限らない
  - デフォルトでダイナミックスレッドは有効（SGIの実装）
    - ダイナミックスレッドを無効にする

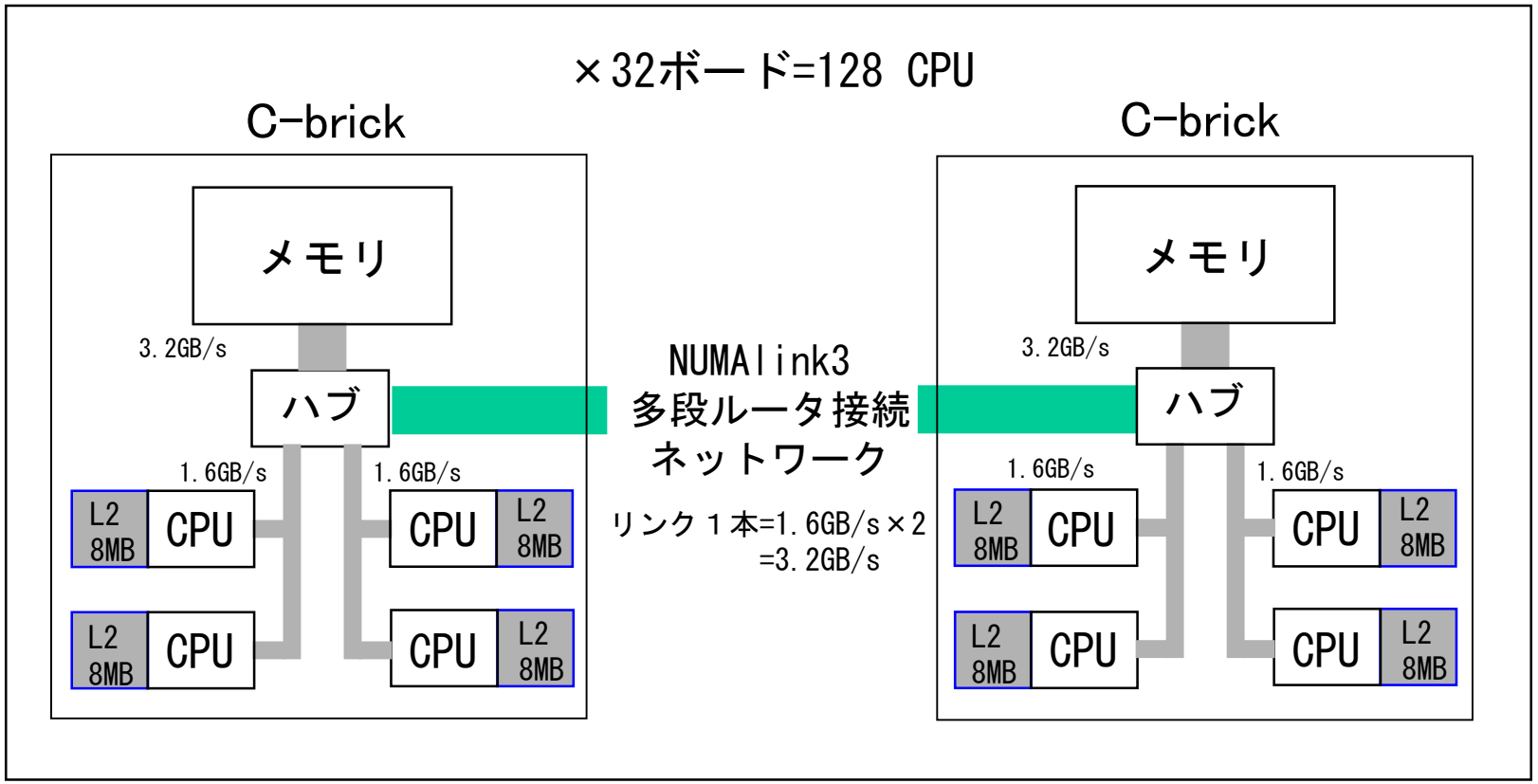
      ```
      % setenv OMP_DYNAMIC FALSE
      ```
    - threadprivateの内容を保持する場合
    - 並列実行領域間で使用するCPU数を固定する場合

SGI Originシステムの特徴

- ccNUMAアーキテクチャの採用
  - Cache Coherent Non-Uniform Memory Access
- 特徴
  - メモリは物理的にシステム全体に分散
  - システム全体で論理的な共有メモリを実現
  - ハードウェアでキャッシュのコヒーレンシーを保証
  - ローカルメモリへのアクセスは、リモートメモリへのアクセスよりも高速
  - それぞれのノード内のローカルなメモリアクセスには相互干渉がない

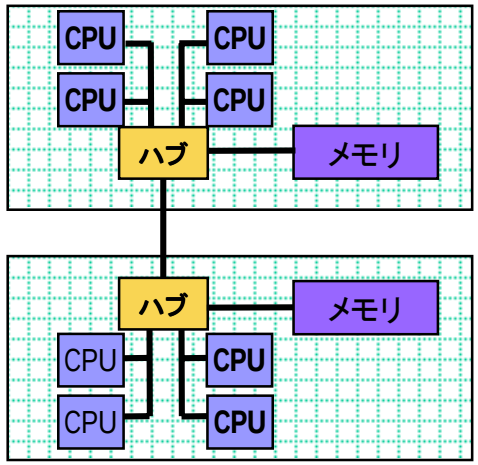
SGI Origin3800 アーキテクチャ

- システム全体として共有メモリシステム
- 4CPUでローカルメモリを共有



データ配置方法

- データ(ページ単位)は初期化されて初めてメモリに配置される(ファーストタッチ)
- もし、初期化が逐次実行領域であれば該当するデータはすべて特定ボードのメモリに配置される
- 並列実行領域では全プロセッサが特定ボードのメモリをアクセスすることになり、プログラムのスケーラビリティが低下する



実際の演算部分と同じ配置になるよう初期化部分も並列化する

```
              real*8  A(n), B(n), C(n), D(n)
!$omp parallel do private(i)
              do  i=1, n
                A(i) = 0.
                B(i) = i/2
                C(i) = i/3
                D(i) = i/7
              enddo
!$omp parallel do private(i)
              do  i=1, n
              A(i) = B(i) + C(i) + D(i)
              enddo
```

SGI Originでの性能解析

SpeedShopによる解析

- プログラムのボトルネックの解析
- 関数、サブルーチン単位での解析
  - 行レベルでも解析可能
- 特別なコンパイルオプションは不要
- 再コンパイルなしで解析が可能

```
man speedshop
```

```
% ssrun -fpcsampx 実行モジュール名
% prof -h データファイル名１ [データファイル名2]…
```

実行モジュール名+.fpcsampx.[mp]PID

SpeedShop: 解析例

SpeedShop解析結果

```
Summary of statistical PC sampling data (fpcsampx)--
        2844: Total samples
       2.844: Accumulated time (secs.)
         1.0: Time per sample (msecs.)
           4: Sample bin width (bytes)
-----------------------------------------------------------------------
line list, in descending order by time
-----------------------------------------------------------------------
    secs      %     cum.%  samples   function (dso: file, line)

   2.600   91.4%   91.4%     2600   __mpdo_MAIN__2 (a.out: main.f, 14)
   0.079    2.8%   94.2%       79   __mpdo_MAIN__2 (a.out: main.f, 15)
   0.053    1.9%   96.1%       53   __mp_wait_for completion (libmp.so:...
   0.042    1.5%   97.5%       42   __mpdo_MAIN__1 (a.out: main.f, 8)
        . . .  （略）. . .
```

mainの中の2番目の並列実行領域の意味

main.fの14行目に対応

ソースコード

```
1.      program main
2.      parameter(n=1000)
3.      real a(n,n),b(n,n),c(n,n)
4.        do j=1,n
5.        do i=1,n
6.            a(i,j)=1.
7.            b(i,j)=1.
8.            c(i,j)=0.
9.        enddo
10.       enddo
11.      do k=1,n
12.       do j=1,n
13.       do i=1,n
14.         c(j,k)=c(j,k)+a(k,i)*b(i,j)
15.       enddo
16.       enddo
17.       enddo
18.     print *,c(1,1)
19.     end
```

perfex コマンド

- プログラム実行時の命令レベルの統計情報を収集
  - 全体のMflops値、キャッシュ利用情報など
- 実行終了時に標準エラー出力に出力
- 再コンパイルなしで解析が可能

```
man perfex
```

```
% perfex -a -x -y 実行モジュール名
```

perfex -a -x -y: 出力例

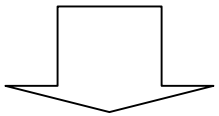
```
                Costs for pid    4331257 (a.out)        Typical              Minimum              Maximum
   Event Counter Name                                 Counter Value    Time (sec)    Time (sec)    Time (sec)
================================================================================================================
 0 Cycles.....................................  127370944     0.318427     0.318427     0.318427
16 Executed prefetch instructions.............   11718624     0.000000     0.000000     0.000000
21 Graduated floating point instructions......  116991216     0.292478     0.146239    15.208858
18 Graduated loads............................   69565280     0.173913     0.173913     0.173913
 2 Decoded loads..............................   59090752     0.147727     0.147727     0.147727
 4 Miss handling table occupancy..............   40460768     0.101152     0.101152     0.101152
25 Primary data cache misses..................    2473200     0.052555     0.013417     0.052555
 6 Resolved conditional branches..............   18776368     0.046941     0.046941     0.046941
26 Secondary data cache misses................      60752     0.015171     0.009573     0.015171
 3 Decoded stores.............................    2840912     0.007102     0.007102     0.007102
19 Graduated stores...........................    2837952     0.007095     0.007095     0.007095
22 Quadwords written back from primary data cache...  590944     0.005880     0.004639     0.005880
24 Mispredicted branches......................     129136     0.002350     0.001937     0.002844
.. .. .. .. .. (略) .. .. .. .. ..

Statistics
================================================================================
Graduated instructions/cycle.....................................    1.757898
Graduated floating point instructions/cycle......................    0.918508
Graduated loads & stores/cycle...................................    0.568444
Graduated loads & stores/floating point instruction..............    0.618878
Mispredicted branches/Resolved conditional branches..............    0.006878
Graduated loads /Decoded loads ( and prefetches )................    0.982430
Graduated stores/Decoded stores..................................    0.998958
Data mispredict/Data scache hits.................................    0.041684
Instruction mispredict/Instruction scache hits...................    0.000000
L1 Cache Line Reuse..............................................   28.275122
L2 Cache Line Reuse..............................................   39.709771
L1 Data Cache Hit Rate...........................................    0.965841
L2 Data Cache Hit Rate...........................................    0.975436
Time accessing memory/Total time.................................    0.784272
Time not making progress (probably waiting on memory) / Total time.....  1.000000
L1--L2 bandwidth used (MB/s, average per process)................  278.234584
Memory bandwidth used (MB/s, average per process)................   25.829338
MFLOPS (average per process).....................................  367.403153
Cache misses in flight per cycle (average).......................    0.317661
Prefetch cache miss rate.........................................    0.048238
```

領域分割法への適用

ループレベルの並列化の限界

- すべてのループの並列化は困難
  - 逐次実行領域の存在
- 並列化の粒度
  - 一般的にループレベルの並列化の粒度は小さい
    - ループ終了時の同期（no waitで非同期）
    - 並列実行領域内部の同期（クリティカルセクション）
    - キャッシュとメモリの整合性



- 領域分割法による記述
    - 同期ポイントの減少による並列化のオーバーヘッドの減少
    - メッセージパッシングプログラミング
    - データのローカリティの向上
    - スケーラビリティの向上

プログラミングモデルの比較

- MPI
  - メッセージ通信ライブラリ
  - 必要な機能が標準関数として用意される
  - 関数の勉強が大変（MPI-1.2, 125種類）
  - 移植性が高い
- OpenMP
  - 指示行、環境変数の体系
  - プログラムの記述やプロトタイプ作成が容易
  - 段階的な並列化が可能
  - MPIのコレクティブルーチン相当の機能がない

MPIとOpenMPでの記述例

MPI

```
% cat mpi.f
      program main
      include 'mpif.h'
      call MPI_Init(ierr)
      call MPI_Comm_Rank
     &    (MPI_COMM_WORLD,me,ierr)
      call MPI_Comm_Size
     &    (MPI_COMM_WORLD,np,ierr)
      print *,'me = ',me,' np = ',np
      call MPI_Finalize(ierr)
      stop
      end
% f90 mpi.f -lmpi
% mpirun -np 4 ./a.out
 me =  0  np =  4
 me =  1  np =  4
 me =  3  np =  4
 me =  2  np =  4
%
```

デフォルト値の設定
−mp,−apoがない時
はコメントの扱い

OpenMP

```
% cat openmp.f
      program main
!$omp parallel private(me,np)
      me=0
!$    me=omp_get_thread_num()
      np=1
!$    np=omp_get_num_threads()
      print *,'me = ',me,' np = ',np
!$omp end parallel
      stop
      end
% f90 -mp openmp.f
% setenv OMP_NUM_THREADS 4
% ./a.out
 me =  1  np =  4
 me =  2  np =  4
 me =  3  np =  4
 me =  0  np =  4
%
```

並列実行領域
の中で呼ぶ

並列実行領域
の中で呼ぶ

スレッド間でデータを交換する

- 並列実行領域内で呼ばれた関数がローカルに確保したメモリはプライベート
  - シェアード変数を介して交換する方法
    - #Nがローカル変数の内容をシェアード変数にコピー
    - 同期
    - #Mがそのシェアード変数を参照し、ローカル変数にコピー
      - » ダブルコピー
  - スレッド間で直接データを交換する方法
    - グローバルポインタを使用して直接参照することが可能
      - » コピー不要
    - SGI拡張MP_SHMEM通信ライブラリ
      - » シングルコピー

グローバルポインタによる参照例

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define  MAX_THREADS  16
#define  N    10
double *r[MAX_THREADS];
int hello(void)
{
        int     i,me,np,you;
        double  *p;
        me=omp_get_thread_num();
        np=omp_get_num_threads();
        you=(me+1)%np;
        p=malloc(sizeof(double)*N);
        r[me]=p;
        for ( i=0 ; i<N ; i++)
                p[i]=me;
#pragma omp barrier
        p=r[you];
        printf("me = %d you = %d p[0]= %f¥n "
           ,me,you,p[0]);
        return (EXIT_SUCCESS);
}
```

グローバルポインタ

初期化の完了を待つ

ローカルなポインタをグローバルポインタにコピー

```
int main(void)
{
#pragma omp parallel
{
        hello();
}
        exit (EXIT_SUCCESS);
}

% cc -mp test.c
% ./a.out
me = 1 you = 2 p[0]= 2.000000
me = 2 you = 3 p[0]= 3.000000
me = 0 you = 1 p[0]= 1.000000
me = 3 you = 0 p[0]= 0.000000
%
```

MP_SHMEM通信ライブラリの例

```
% cat mp_shmem.f
        program main
        integer    source,target,me,pe
        common/com/target
!$omp threadprivate(/com/)

!$omp parallel
!$omp&  private(source,np,target,me,pe)
        me=omp_get_thread_num()
        source=me
        np=omp_get_num_threads()
        pe = mod(source+1,np)
        call MP_SHMEM_PUT32
     &  (target, source, 1, pe)
!$omp barrier
        print '(a5,4i4)','put: ',
     &  me,pe,source,target
!$omp end parallel
        end
%
```

リモート参照する変数をthreadprivateで宣言する

MP_SHMEMは一方向通信ライブラリなのでデータ転送の完了を待つ

```
% f90 -mp mp_shmem.f
% setenv OMP_NUM_THREADS 4
% setenv OMP_DYNAMIC FALSE
% ./a.out
put:    0   1   0   3
put:    1   2   1   0
put:    2   3   2   1
put:    3   0   3   2
%
```

スレッド数の制御にOpenMPの実行時ライブラリを使用するので、リンク時に-mpのフラグが必要

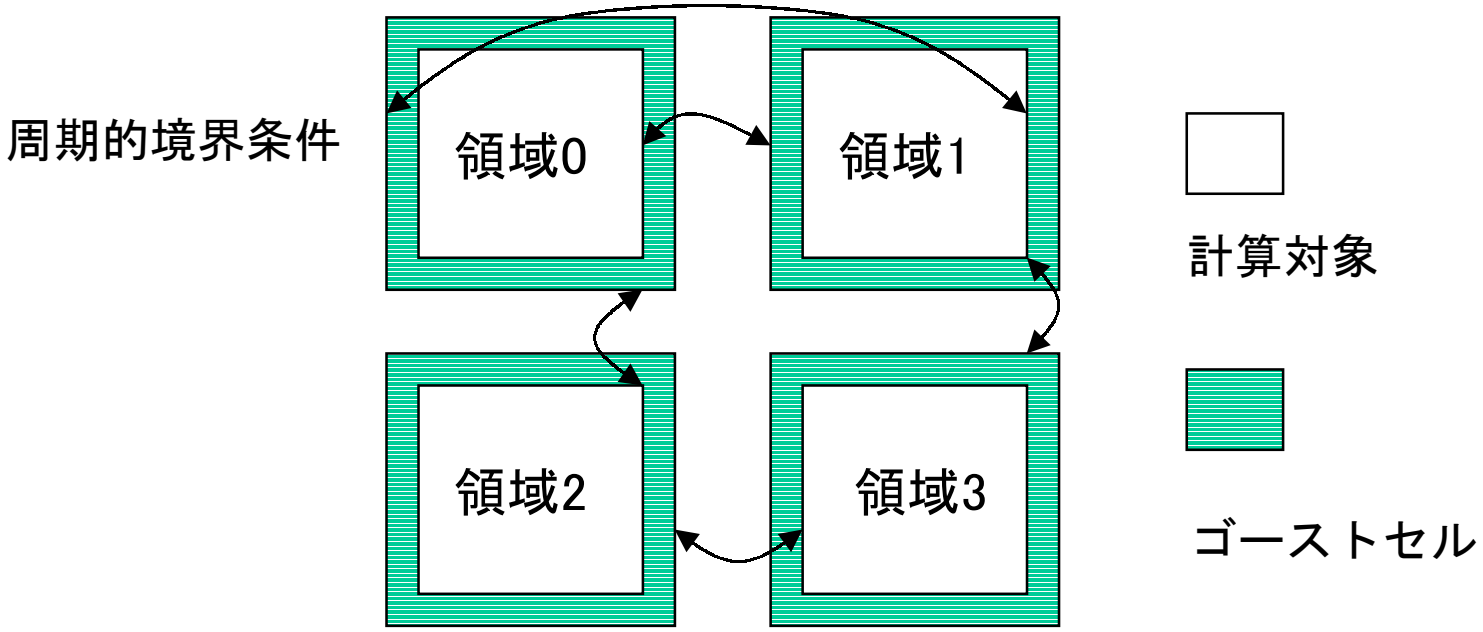
threadprivateで宣言された変数の内容を別の並列実行領域で参照する場合、ダイナミックスレッドの機能を無効にして、その間にスレッド数が変化しないことを保証する
この例のように並列実行領域が1つしかない場合は意味がありません

メッセージパッシングとの比較

- プログラミング上の利点
  - 領域分割のアルゴリズムは、MPIなどと同様であるが、プログラミングは容易
    - グローバル参照可能なデータを全スレッドで共有
    - メッセージ交換を不要にすることも可能
      - » ゴーストセルの取り扱いがあるので実際には難しい
- 性能面での利点
  - データ転送性能
    - レイテンシ
    - バンド幅
  - メッセージ交換時間の削減

領域分割法おけるゴーストセル

- ゴーストセルの導入
  - 差分法では隣接する格子点の値の重み付き平均を計算する



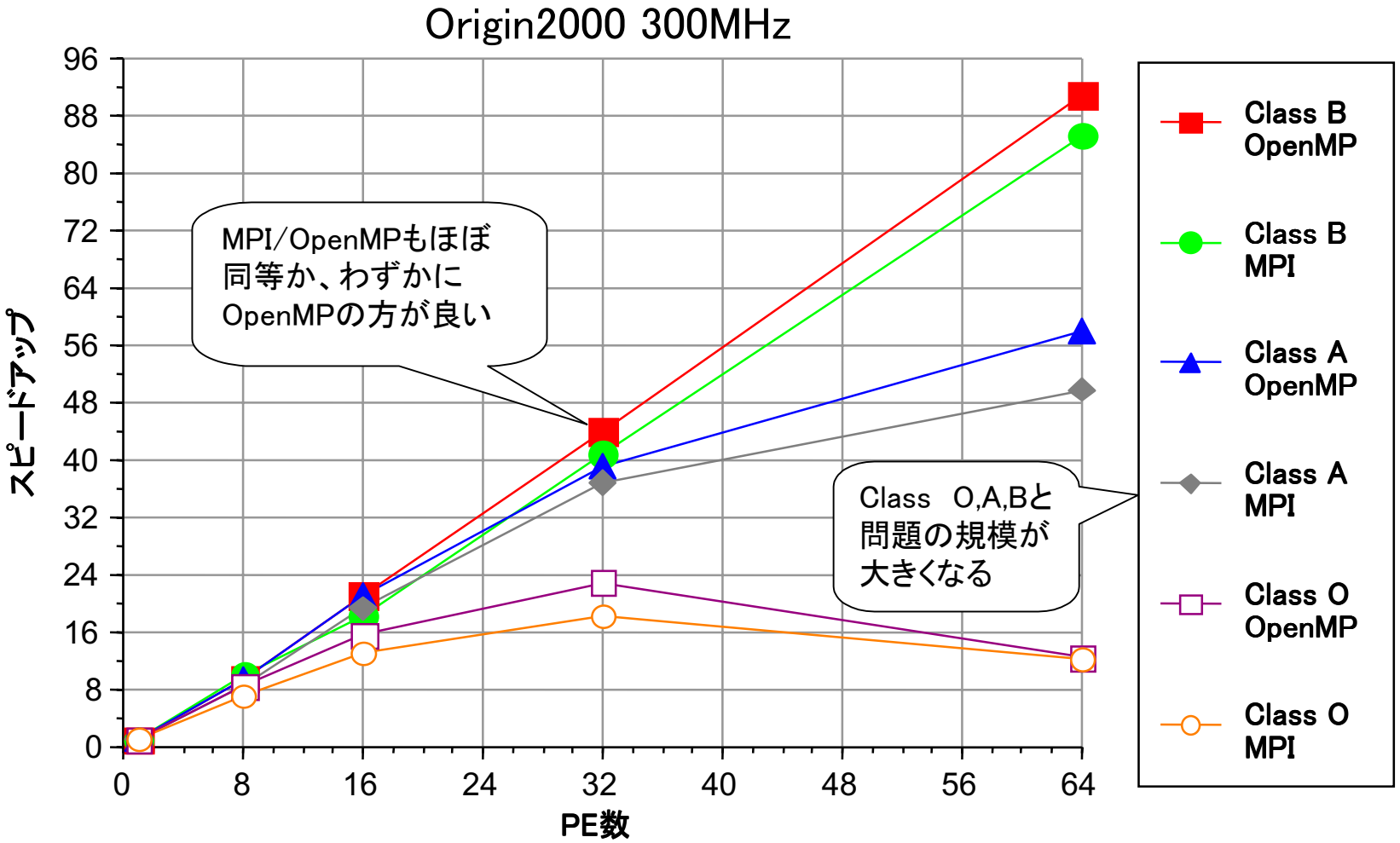
  - 計算対象よりも一回り大きなグリッドをとり、隣接する領域と境界条件を交換しながら計算を行う

MPIとOpenMPのスケーラビリティ

- NAS Parallel Benchmark（NPB）2.2　APPLU
  - MPIに基づくソースコードが提供されるベンチマーク
  - 問題の規模（グリッド数）に応じた3ケース（Class O,A,B）
  - 流体擬似アプリケーション
- 共有メモリ型プログラミングへの変更
  - MPI版と同じアルゴリズムで並列化
  - 並列化の粒度はMPI版と同じ
  - ソースは25%程度減少
- 通信性能の差がプログラムの性能差

スケーラビリティの比較

- NAS Parallel Benchmark 2.2 APPLU



OpenMPのスケーラビリティの限界

OpenMPの仕様と実装

- OpenMPはスケーラビリティを考慮した仕様
  - Orphaned指示行
  - ダイナミックスレッド
  - 並列実行領域の入れ子
- OpenMPの実装は各ベンダ任せ
  - 機能を実装するか否か
  - デフォルト値の設定
  - ハード・ソフトに依存した最適化
- 性能評価

スカラの縮約(リダクション)演算

- スカラの縮約演算の時間を評価する
  - MPI_Allreduceの縮約(MPI_SUM)演算
  - OpenMPのクリティカルセクションで縮約演算を記述

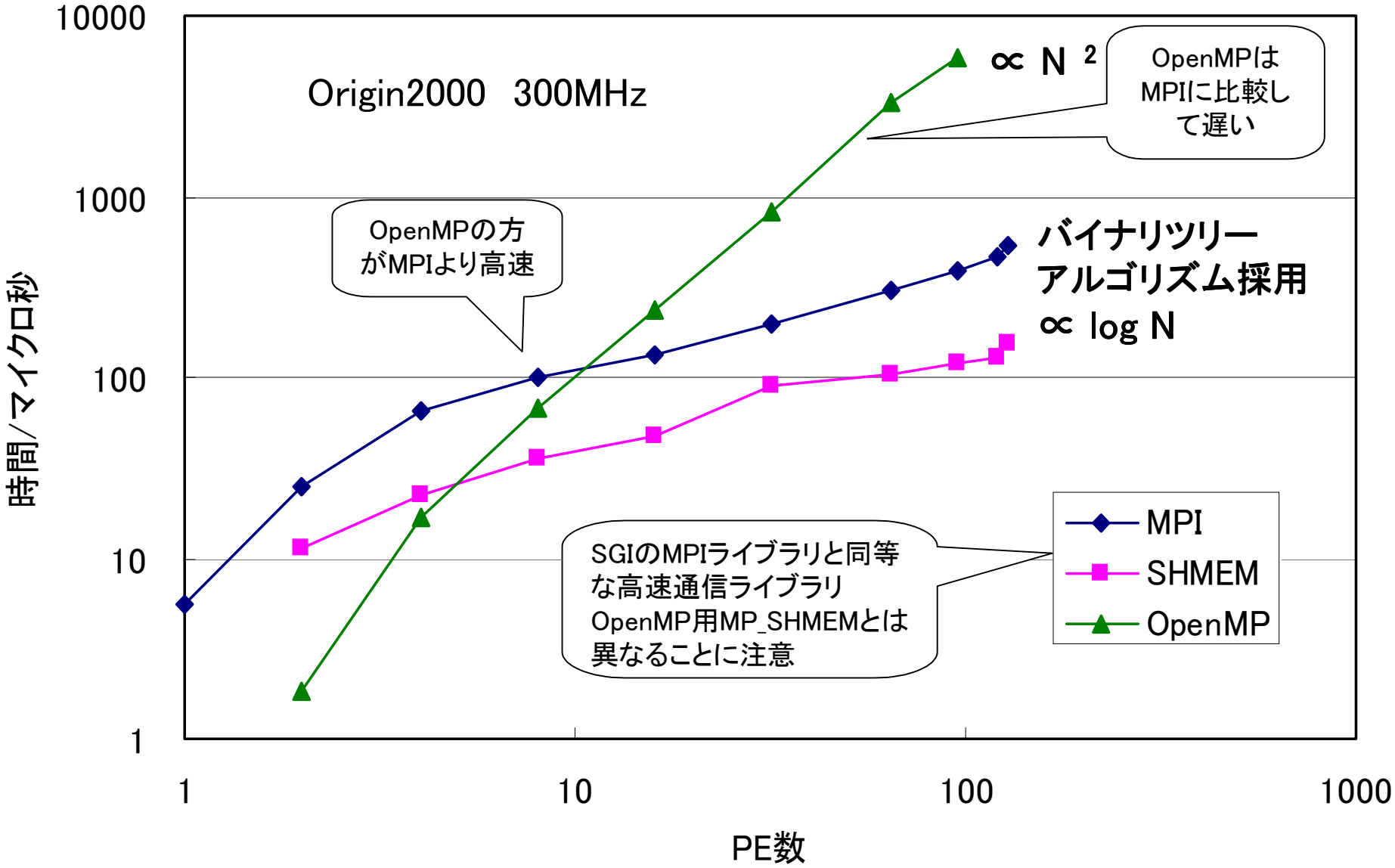
MPI

```
  real(KIND=8):: sum,gsum
  ...
  sum=1.d0
  call MPI_Allreduce(sum,gsum,
 &     1,MPI_DOUBLE_PRECISION,
 &     MPI_SUM,MPI_COMM_WORLD,ierr)
```

OpenMP

```
      real(KIND=8):: gsum
      gsum=0.d0
!$omp parallel
!$omp critical
      gsum=gsum+1.d0
!$omp end critical
!$omp end parallel
```

スカラの縮約演算の処理時間



OpenMP

- プログラミング
  - 簡潔な記述で並列処理構造の記述が可能
    - ループレベル
    - 関数やコードブロックレベル
    - 同じ実行モジュールに対して環境変数による制御
    - 使用するスレッド数に制約がない
  - MPIのような領域分割法にも適応可能
  - スケーラビリティの限界
- システム運用
  - ダイナミックスレッドによりシステムの利用効率が向上

SGI Origin マニュアルページ

- プログラミング
  - OpenMP関連

```
% man pe_environ
% man omp_threads
```

  - MP_SHMEMライブラリ

```
% man mp
```

  - メッセージパッシングライブラリ MPI/SHMEM

```
% man mpi
% man shmem
```

- プロファイルツール

```
% man speedshop
% man perfex
```

2000年11月 OpenMP Fortran 2.0

- OpenMP 1.1に対する機能追加
  - Fotran90 配列構文に対するworkshare/end workshare
  - data文に現れる変数にsaveの属性が必要(Fortran95)
  - Fortran90 モジュール内のprivate変数の記述
  - 経過時間計測関数 omp_get _wtime/omp_get_wtick
  - threadprivateが共通ブロック以外の変数にも適用可能
  - copyinが共通ブロック以外の変数にも適用可能
  - copyprivateの追加
  - reductionに配列の指定も可能
  - 使用するCPU数の指定 num_threads節
  - など