(財)計算科学振興財団、大学院GP「大学連合による計算科学の最先端人材育成」

第2回　社会人向けスパコン実践セミナー　資料

ベンチマークアプリで性能を検証

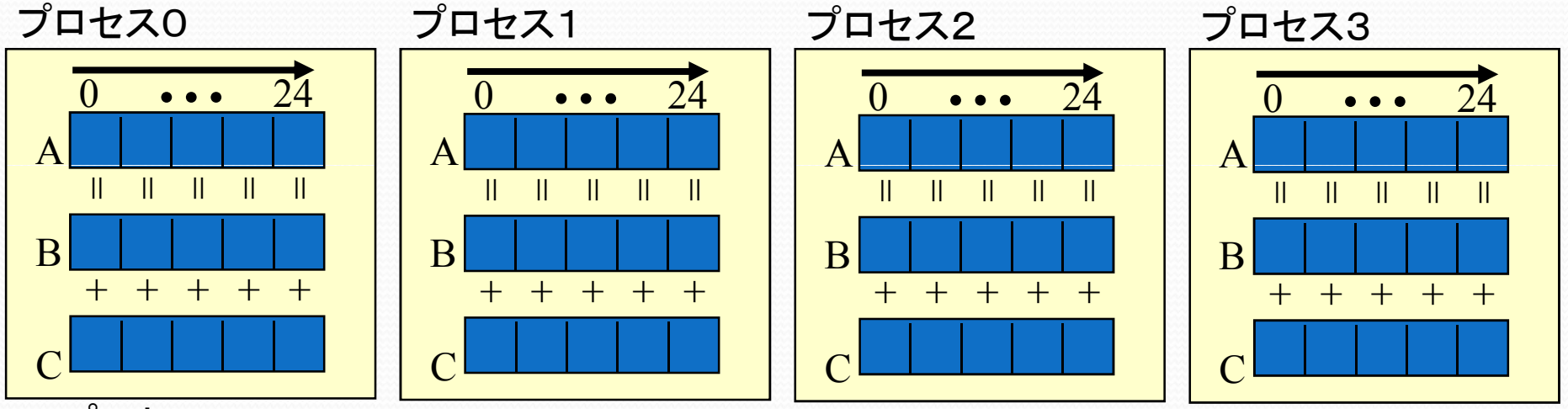
2009年6月17日　9:00～12:15

九州大学情報基盤研究開発センター

南里　豪志

MPIによる並列処理

- それぞれ独自のメモリを使いながら仕事を分担する並列処理
- 他のプロセスの計算結果を参照するには通信が必要



プロセス0

```
double A[25], B[25], C[25];
  ...
for (i = 0; i < 25; i++)
    A[i] = B[i] + C[i];
```

プロセス3

```
double A[25], B[25], C[25];
  ...
for (i = 0; i < 25; i++)
    A[i] = B[i] + C[i];
```

並列処理に対する期待と現実

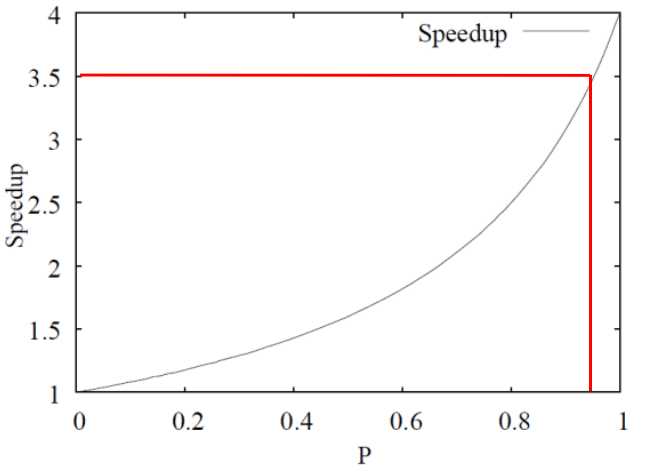
- 計算機利用者:
  「CPUを8台使うんだから8倍… とまでは言わなくても、
  7倍程度は速くなってほしいよね」

- 計算機センター職員、計算機メーカーSE
  「CPU台数の半分くらい速くなれば上等だよね」
  - アムダールの法則
  - 負荷のバランス
  - 通信のコスト

アムダールの法則

- 並列化による性能向上率の理論的な限界
  ＝1/((1−P)+P/N)
  - P：プログラム中の並列化対象部分が全処理時間に占める割合
  - N：プロセス数（スレッド数）

- 例えば N=4 の場合、
  3.5倍以上高速化するためには
  プログラムの処理時間の95%以上が
  並列化できなければならない.
  - さらに、並列化することによって新たに
    発生するコストがある。
    - スレッド（プロセス）間の待ち時間や、
      プロセス間データ転送に要する通信時間等。



プロセス数4（N=4）の場合の性能向上率

本実習の目的

- いくつかのプログラムで並列化の効果を検証する
- さらに、並列化による性能向上を阻害している要因を追及するための解析手法を試してみる
  - 負荷のバランス
  - 通信コスト

並列化の効果を示す指標

- **速度向上率＝**
  (1プロセスによる実行時間) / (複数プロセスによる実行時間)
  - 並列化によって何倍速くなったか

- **並列化効率＝**
  速度向上率 / 使用CPU台数
  - 並列化した意味がどの程度あったか

元も子もない話

- **並列化は必要ですか？**

- **例）並列化効率 0.5**
  - ８台のCPUを使って４倍の性能向上

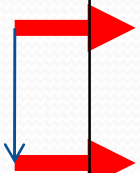
- **（もし、実行したいプログラムがたくさんあるのであれば）**
  ８台のCPUでそれぞれ別のプログラムを実行
  ⇒　全体で8倍の性能向上　＝　並列化効率 1.0
  - 例えば、同じプログラムに対して入力データが多数ある場合に有効

- **状況（プログラムの性質、仕事の内容）によって選択**
  - 並列化するか否か
  - 並列化する場合、１回の実行についてCPUは何台使うか？

MPIプログラムの時間計測 MPI_Wtime

- 現在時間(秒)を実数で返す関数
- 利用例

計測対象

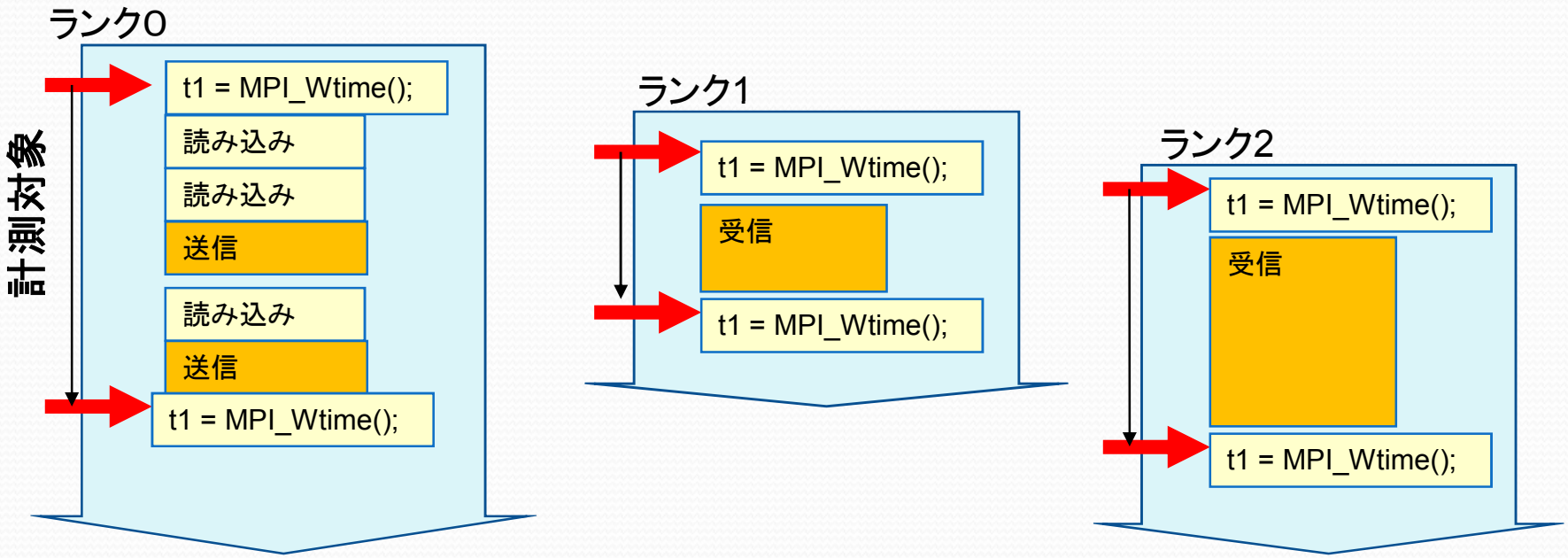


```
 ...
double t1, t2;
 ...
t1 = MPI_Wtime();
 処理
t2 = MPI_Wtime();

printf("Elapsed time: %e sec.¥n", t2 – t1);
```

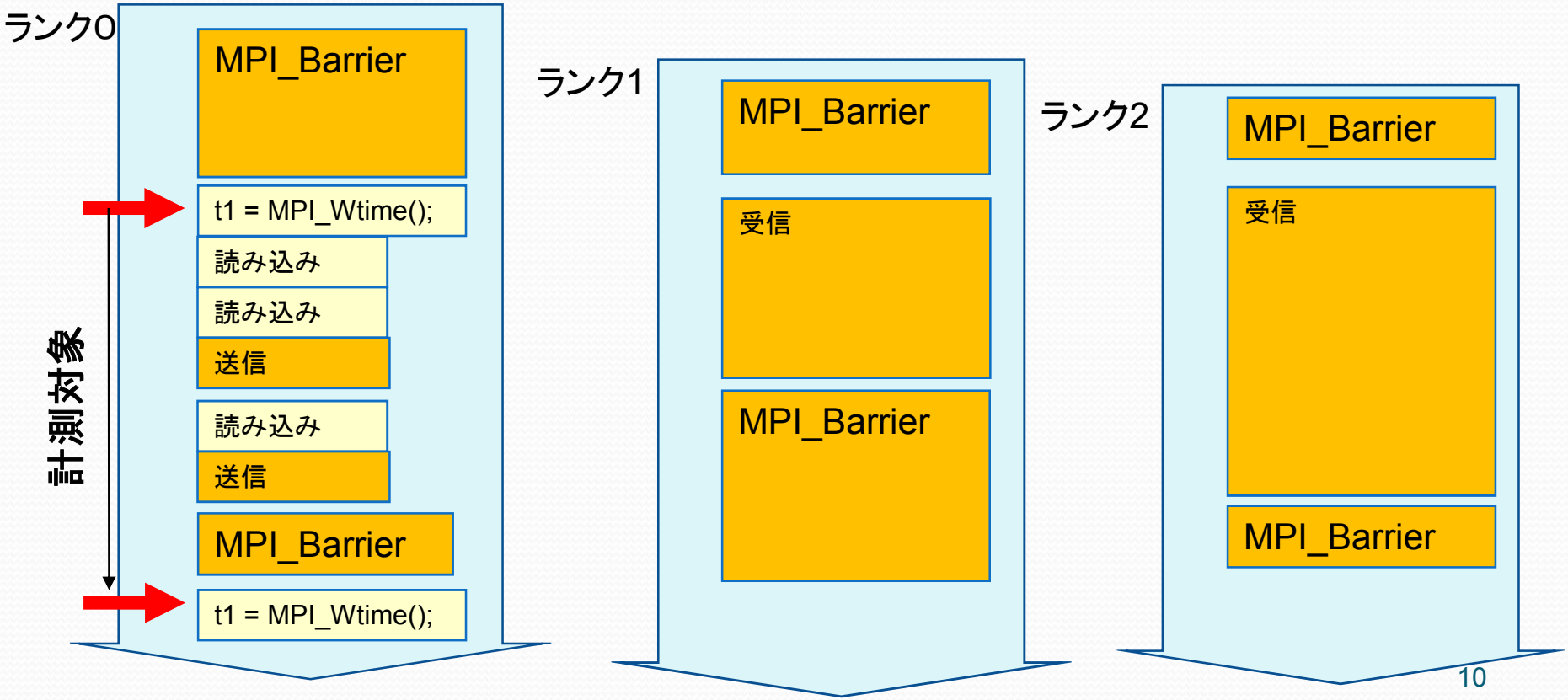
並列プログラムにおける時間計測の問題

- プロセス毎に違う時間を測定：
  どの時間が本当の所要時間か？



集団通信 MPI_Barrierを使った解決策

- **時間計測前にMPI_Barrierで同期**
  - ランク0の計測だけでおおまかな傾向をつかめる
  - 計測結果に MPI_Barrierのコストが含まれる



より細かい解析に向けて

- **MPI_Reduceで統計的な数値を取得**
  - 全プロセスの平均時間
  - 全プロセスの中の最大時間
  - 全プロセスの中の最小時間

```
double t1, t2, t, total;

t1 = MPI_Wtime();
 処理
t2 = MPI_Wtime();
t = t2 - t1;
MPI_Reduce(&t, &total, 1, MPI_DOUBLE, MPI_SUM,
             0, MPI_COMM_WORLD);
if (myrank == 0)
  printf("Ave. elapsed: %e sec.¥n", total/procs);
```

最大(Max)、平均(Ave)、最小(Min)の関係

- プロセス毎の負荷（仕事量）のばらつき検証に利用

	Max – Ave 大	Max – Ave 小
Ave – Min 大	NG 全体でばらつき	ほぼOK 少数のプロセスで 負荷が少ない
Ave – Min 小	NG 少数のプロセスで負荷が 多く足を引っ張っている	OK 負荷は均等

しかし、計算に要している時間と通信に要している時間を区別できない

さらに細かく

- 通信時間の計測

```
double t1, t2, t3, t4 comm=0;
t3 = MPI_Wtime();
for (i = 0; i < N; i++){
  計算
  t1 = MPI_Wtime();
  通信
  t2 = MPI_Wtime(); comm += t2 – t1;
  計算
  t1 = MPI_Wtime();
  通信
  t2 = MPI_Wtime(); comm += t2 – t1;
}
t4 = MPI_Wtime();
```

計算時間の算出

- 計算時間　＝　所要時間　－　通信時間
  - もしくは、直接計算時間を計測し、積算しても良い

各プロセスの計算時間を
MPI_Reduceで集計（平均、最大、最小）して評価する

- 計算時間のばらつき　＝　負荷の不均衡の度合い

- 注意:　通信時間には、負荷の不均衡によって生じた待ち時間が含まれるので、単純な評価は難しい
  ⇒　まず負荷を均等にした上で、通信時間を評価する

ハイブリッド並列化の効果

- **MPIと OpenMPによるハイブリッド並列化:**
  - 計算ノードに複数のCPUコアが搭載されている場合、
    「計算ノード毎に MPIのプロセスを一つ起動」
    「各プロセスをさらに複数の OpenMPスレッドに分ける」
    という階層的な並列化が有効な場合がある

- **期待される効果:**
  - プロセスの数が減る　＝　通信回数が減る
  - ノード内でのデータの相互参照を通信ではなくメモリの共有によって実現するので、さらに通信回数が減る
- **ただし、プログラムによっては性能低下**
  - メモリアクセスの衝突、スレッドの生成コスト

実習

- プログラムを展開
- どれか１～２種類のプログラムについて
  - コンパイル
  - テスト実行
  - 所要時間計測（ランク０）
  - 所要時間解析（平均、最大、最小）
  - 通信時間解析
- 時間に余裕があったら、計測結果を Excel で表にまとめる
  - さらに、グラフ化する

実習

- プログラムの展開と確認
  - scalar.scitec.kobe-u.ac.jp にログインして

```
$ tar xvf /tmp/codes.tar
$ ls
$ cd codes
$ ls
```

- その後、各プログラムのディレクトリに移動
  - 例) sieve1 があるディレクトリへ移動

```
$ cd sieve1
```

ベンチマークプログラムの種類

- **sieve1**
  - 「エラトステネスの篩」アルゴリズムによる素数の数の導出
- **floyd**
  - 「フロイド」のアルゴリズムによる最短パスの導出
- **mv1**
  - 行列ベクトル積（行列を行方向にブロックサイクリック分割）
- **mv2**
  - 行列ベクトル積（行列を列方向にブロック分割。ベクトルもブロック分割）

全て以下のサイトより
http://fac-staff.seattleu.edu/quinnm/web/education/ParallelProgramming/mpi/
（ Michael J. Quinn, "Parallel Programming in C with MPI and OpenMP")

ファイルの種類

- **～.c (sieve1.c, floyd.c等)**
  - 各アプリケーションのオリジナルのソースコード
    - 若干、オリジナルから変更しています
- **～-omp.c (sieve1-omp.c等)**
  - 各アプリケーションを OpenMPで並列化したコード
- **～-t.c (sieve1-t.c, floyd-t.c, siveve1-omp-t.c 等)**
  - 各アプリケーションで計算時間を計測して表示するソースコード
- **～-数字.sh**
  (sieve1-1.sh, sieve1-2x2.sh, floyd-4.sh 等)
  - 各アプリケーションのオリジナル版を実行するためのスクリプト
  - ～-数字x2.shは、OpenMPとMPIによるハイブリッドでの実行
- **～-t-数字.sh (sieve1-t-4.sh, floyd-t-8x2.sh等)**
  - 各アプリケーションの計算時間を計測して表示するプログラムを実行するためのスクリプト

コンパイル方法(MPIのみ)

- **sieve1**

```
$ mpicc -O3 sieve1.c -o sieve1
```

- **floyd, mv1, mv2**
  - MyMPI.c と結合
  - 例）floyd

```
$ mpicc-O3 floyd.c MyMPI.c -o floyd
```

コンパイル方法(MPI＋OpenMP）

- ファイル名に **omp** が付いているプログラムは、
  オプションに **-mp** を追加してコンパイルし、
  翻訳後のファイル名を ～**-omp** とする

- **sieve1**

```
$ mpicc -mp -O3 sieve1-omp.c -o sieve1-omp
```

- **floyd, mv1, mv2**
  - MyMPI.c と結合

```
$ mpicc -mp -O3 floyd.c MyMPI.c -o floyd-omp
```

実行方法

- 以下のように、ジョブとして投入
  - 例）sieve1 の4プロセス（MPIのみ）

```
$ qsub sieve1-4.sh
```

  - 例）floyd の8プロセスx2スレッド（MPI＋OpenMP）

```
$ qsub floyd-8x2.sh
```

- 混んでいる場合は、順番待ちが発生

結果の確認

- ジョブの出力ファイルをless で確認
  - 例）sieve1-4.sh を投入した結果（ジョブ番号が 12345番だった場合）

```
$ less sieve1-4.sh.o12345
```

- less の操作:
  スペースキーもしくは f : 1ページ下へ
  b : 1ページ上へ
  j : 1行下へ
  k : 1行上へ
  q : less の終了

各アプリケーションの出力

- 所要時間の出力
  - sieve1

```
[0] SIEVE (4)  31.865740
```

  - floyd

```
[0] Floyd, matrix size 1000, 2 processes:   1.18 seconds
```

  - mv1

```
[0] MV1) N = 1024, Processes = 4, Time =     0.001507 sec,Mflop = 1391.61
```

  - mv2

```
[0] MV3) N = 1024, Processes = 4, Time =     0.001431 sec,Mflop = 1465.52
```

計算時間の表示

- 計算時間計測用のプログラムをコンパイル
  - -t がついたソースファイル

- 例）sieve1

```
$ mpicc -O3 sieve1-t.c -o sieve1-t
```

```
$ mpicc -mp -O3 sieve1-omp-t.c -o sieve1-omp-t
```

- 例）floyd

```
$ mpicc -O3 floyd-t.c MyMPI.c -o floyd-t
```

```
$ mpicc -mp -O3 floyd-omp-t.c MyMPI.c -o floyd-omp-t
```

計算時間を計測するプログラムの実行

- **-t のついた .shファイルを投入**

- **例）sieve1**

```
$ qsub sieve1-t-4.sh
```

```
$ qsub sieve1-t-8x2.sh
```

計算時間時間(平均、最大、最小)の確認

- ジョブの出力ファイル (sieve1-t-4.sh.o番号　等)に、
  所要時間（平均計算時間，最小計算時間，最大計算時間）
  の形で表示

```
[0] SIEVE (4)  31.865740 (3.12717 4.88995, 0.03348)
```

おわりに:
チューニングに向けた検討

- **計算負荷のばらつきを回避できないか**
  - プロセスへの計算の割り当て方を調整する
- **計算自体を高速化できないか**
  - コンパイル時の最適化オプションは適切か
  - 数値計算ライブラリ(行列積、連立一次方程式求解等)が利用できないか
  - ループのチューニング(キャッシュサイズに合わせた分割)
  - 並列化に伴って、計算部分の効率が低くなっていないか
    - 可能であれば、並列化前の逐次プログラムの性能と比較する
- **通信を減らすことができないか**

おつかれさまでした

- 本講義資料に関する質問は以下まで:

九州大学情報基盤研究開発センター　南里　豪志
Email　nanri@cc.kyushu-u.ac.jp