実践編[1]

陰山 聡

神戸大学 システム情報学研究科 計算科学専攻

2013.07.18

[1] 計算科学演習 I（2013 年前期） 3 号館 演習室

ジョブキュー

ハイブリッド並列化

時間計測

スレッド並列化

補遺 A：Flat MPI 並列化

補遺 B：1 次元領域分割と 2 次元領域分割

補遺 C: 2 次元並列化

ジョブキュー

ジョブキュー

これまでジョブスクリプトでは、

```
#PJM -L "rscgrp=small"
```

としてきた。これはジョブキューの種別の指定。

π-computer のジョブキュー：

- `small`: 12 ノードまで
- `large`: 84 ノードまで　　$\Rightarrow$ 最大 $84 \times 16 = 1344$ 並列
- `school`: 24 ノードまで

今日、この講義中（だけ）は `small` $\Rightarrow$ `school` に。

講義後は、`small` または `large` で[2]。

---

[2]特に講義後、レポート課題のためには large で。

ハイブリッド並列化

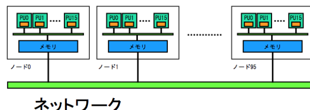
MPI + OpenMP = ハイブリッド並列化

- 一つのノード（プロセッサ）に一つの MPI プロセスを走らせる。
- 各 MPI プロセスで OpenMP によるスレッド並列をする。
- 計算時間の最もかかる do-loop 部分（いまの場合はヤコビ法の部分）だけを本演習の「 OpenMP を使った並列計算」で学んだ方法でコアの数[3]だけ fork させて計算すればよい。

[3] π-computer の場合は 1 プロセッサあたり 16 コア。

本演習の第３回「並列計算とは」の講義資料から引用

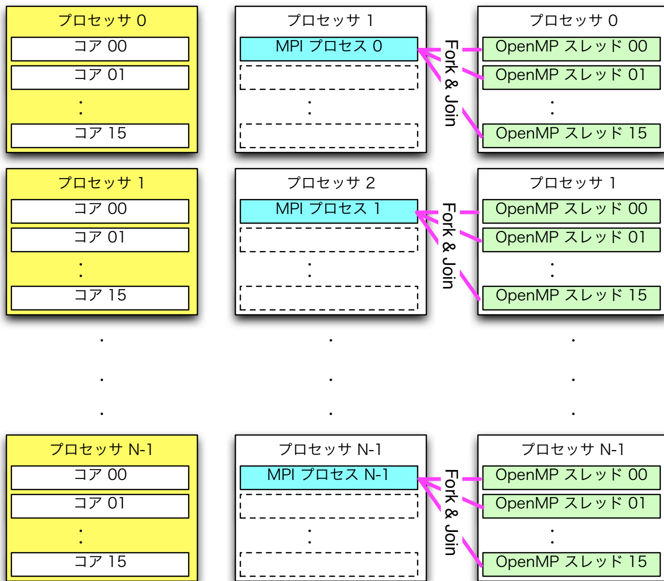
- 富士通 FX10(「京」の上位互換機)
  - 16コア／プロセッサ
  - 1プロセッサ／ノード
  - メモリ32GB ／ノード
  - 全96ノード



富士通FX10(96ノード)

- 本セミナーでの利用法
  - 各ノードを使い，OpenMP の練習
  - 複数ノードを使い，MPI の練習
    - この場合，ノード内も含めて MPI で並列化

ハイブリッド並列化



時間計測

計算時間を測る方法

- CPU 時間
  - Fortran95 ⇒ cpu_time()
- 実時間 (wall clock time)
  - MPI ⇒ MPI_WTIME()
  - OpenMP ⇒ omp_get_wtime()
  - Fortran90 ⇒ system_clock()

heat5.f90

これまで使ってきた heat4...f90 に、system_clock() 関数を使った時間計測モジュール stopwatch_m を組み込んだ。

```
!
! heat5.f90
!   + module stopwatch, to monitor time.
!   + many calls to stopwatch__stt and ..__stp.
!   - data output calls for profile 1d and 2d (commented out.)
!!   usage (on pi-computer)
!!    1) mkdir ../data  (unless there is already.)
!!    2) mpifrtpx -O3 heat5.f90  (copy un to u is slow in default.)
!!    3) pjsub heat5.sh
```

演習

heat5.f90 をコンパイル・実行せよ[4]。

—

注意： このコンパイラでは-O3 オプションを付けないと以下の部分の処理[5]が遅い。

```
u(1:NGRID,jj%stt:jj%end)=un(1:NGRID,jj%stt:jj%end)
```

[4]実行方法は コードの冒頭、usage を参照。
[5]次のページの stopwatch 出力の `copy un to u` ラベルの部分。

実行例

```
##################################################
job start at Tue Jul 16 21:07:29 JST 2013
##################################################
 # myrank= 3  jj%stt & jj%end =  751 1001
 # myrank= 0  jj%stt & jj%end =  1 250
 # myrank= 2  jj%stt & jj%end =  501 750
 # myrank= 1  jj%stt & jj%end =  251 500
 //============<stop watch>===============\\
        profile 1d:    0.000 sec
         main loop:    8.334 sec
      mpi sendrecv:    0.409 sec
            jacobi:    4.103 sec
      copy un to u:    3.799 sec
  ---------------------------------------
             Total:    8.386 sec
 \\============<stop watch>===============//
##################################################
job end at Tue Jul 16 21:07:39 JST 2013
```

スレッド並列化

OpenMP によるスレッド並列化

heat6.f90

```
!
! heat6.f90
!   + OpenMP (now this is a hybrid parallel code, with MPI.)
!   - array calc of u(:,:)=un(:,:). see below.
!   + double do-loops of u(i,j)=un(i,j), for OpenMP.
!   usage (on pi-computer)
!    1) mkdir ../data  (unless there is already.)
!    2) mpifrtpx -Kopenmp heat6.f90
!    3) pjsub heat6.sh
```

OpenMP化した部分

（1）program main の冒頭

```
!$ use omp_lib
```

（2）

```
!$omp parallel do
do j = jj%stt , jj%end
  do i = 1 , NGRID
    un(i,j)=(u(i-1,j)+u(i+1,j)+u(i,j-1)+u(i,j+1))*0.25_DP+heat_
  end do
end do
!$omp end parallel do
```

OpenMP 化した部分（続き）

（3）

```
!    u(1:NGRID,jj%stt:jj%end)=un(1:NGRID,jj%stt:jj%end)
     !$omp parallel do
     do j = jj%stt , jj%end
       do i = 1 , NGRID
         u(i,j)=un(i,j)
       end do
     end do
     !$omp end parallel do
```

演習

（１）コンパイル mpifrtpx -Kopenmp heat6.f90

（２）ジョブスクリプト heat6.sh のジョブキュー指定を school とする

（３）pjsub heat6.sh ジョブ投入

（４）結果をみる

ジョブスクリプト：heat6.sh（中心部分）

```
#!/bin/bash
#PJM -N "heat6"
#PJM -L "rscgrp=small"
#PJM -L "node=4"
#PJM -L "elapse=02:00"
#PJM -j
export FLIB_CNTL_BARRIER_ERR=FALSE
.
.
for opn in 1 2 4 8 16
do
export OMP_NUM_THREADS=$opn
echo "# omp_num_threads = " $opn
mpiexec -n 4 ./a.out
done
.
.
```

レポート課題の前に ···（周囲の人と相談しないこと。）

$$x^x - 2x = 0$$

の $x \geq 0$ の範囲での実数解（近似値）を求めよ[6]。

$x = 2$ は明らかに解。これ以外にもう一つの解 β がある。β の値は次のうちどれに最も近いか？

a: 0.293 b: 0.346 c: 0.432

[6] ヒント：gnuplot で $y = x^x$ と $y = 2x$ の二つのグラフを描けばよい。x の範囲を指定するのは、set xrange [xmin:xmax]

分岐

a と思う人はすぐにアンケートに入力。

b と思う人は Emacs で M-x animate として自分の名前（1st name）を入れよ。「それ」が終わってからすぐに M-x zone と入力。そして、もう一度 M-x zone と入力。その後、アンケートに入力。

c と思う人は Emacs で M-x dunnet を入力。これは（かなり難しい）テキストアドベンチャーである[7]。制限時間は 3 分。ゲームを終了したか、あるいは 3 分経過したらきっぱりと諦めてアンケートに入力。

[7] ヘルプは help と入力。難しいので、最初だけ入力テキストの答えを教えると：get shovel, look shovel, e, e, dig, look, get cpu, ...

レポート課題

heat6.f90 を使い、

- 1 ノード $M \quad (\leq 16)$ スレッドのハイブリッド並列で、
- $N \quad (\leq 84)$ ノードを使い、
- スレッド総数 $P(= M \times N)$ v.s. 計算速度 S のグラフ[8]を、

gnuplot で描け[9]。

[8] S は stopwatch module の出力の "Total" で表示される秒数の逆数と定義する。
[9] 並列化のスケールが悪い時には、格子点数 NGRID を増やす。

レポートには以下を含めること

(a) 氏名・学籍番号
(b) 使用した NGRID, ノード数 N, ノードあたりのスレッド数 M（=OMP_NUM_THREADS）の値、そして（もしあれば）コードを変更した箇所の説明
(c) gnuplot で描いたグラフ
(d) そのグラフについての簡単な考察
(e) この「計算科学演習 I」全体を対する感想と今後改善して欲しいところなど[10]

提出先（gmail）： kageyama.lecture@gmail.com
ファイルフォーマット： pdf
メールタイトル：130718 "学籍番号" "名字"
例：130718 120x227x Yamada
締めきり：2013 年 7 月 25 日 24 時

[10] この項目 (e) は成績評価とは無関係。

補遺 A：Flat MPI 並列化

Flat MPI 並列化

- これまでは 1 ノード（1 プロセッサ、16 コア）に一つだけ MPI プロセスを動かしていた。
- （OpenMP を使ったハイブリッド並列をしない場合）他の 15 個のコアは遊んでいた。
- 1 ノード（1 プロセッサ）に 16 個の MPI プロセスを走らせることも可能。
- 例えば 4 ノード使う場合には合計 $4 \times 16 = 64$ MPI プロセスで並列化。
- このような並列化を Flat MPI という[11]。

[11] OpenMP を使わないで済むのでプログラムはその分簡単になるが、計算速度は一般にはハイブリッド並列化に劣るので推奨しない。

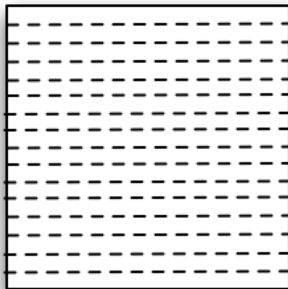
FLAT MPI 並列化

補遺Ｂ：１次元領域分割と２次元領域分割

補遺 C: 2 次元並列化

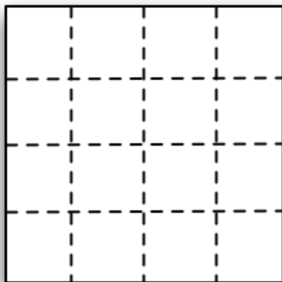
1 次元並列

引き続き、正方形領域の熱伝導問題（平衡温度分布）を考える。これまでの並列化： 1 次元領域分割による並列化: 16 並列



2 次元並列

これも 16 並列。どちらが速いか？1 次元領域分割と 2 次元領域分割どちらを採用すべきか？



そもそも 1 次元分割ができない場合

- 格子数 NGRID 61
- 並列プロセス数 100
  - 1 次元分割不可能
  - 2 次元分割なら可能（総格子点数 3721）

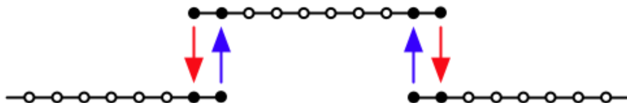
1 次元分割と 2 次元分割のちがい

格子点数 NGRID が十分大きければ 1 次元分割と 2 次元分割は同じか？

- プロセスあたりの計算量は同じ
- 通信量が違う

計算と通信

1 次元空間を格子点で離散化した上で、MPI でプロセス間通信を行う場合を考える。

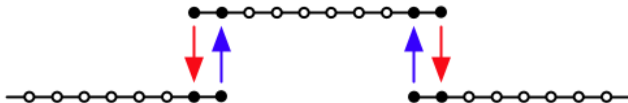


計算と通信

計算格子には 2 種類ある。

1. その上で計算だけを行う格子。
2. MPI 通信のデータを送ったり、受けたりする格子である。

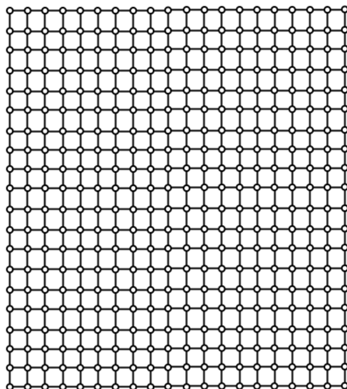
(一番外側から 2 番目の格子は計算も通信も行う。)



通信は時間がかかるので、通信を行う格子点は少ないほうが望ましい。

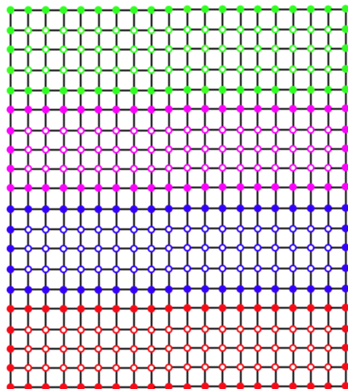
2 次元領域分割

正方形領域を 400 個の格子点で離散化した場合



2 次元領域分割

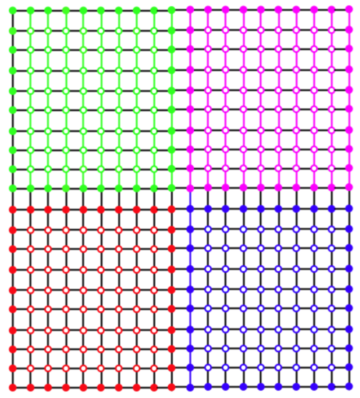
4 つの MPI プロセスで並列化。1 次元領域分割。



赤のプロセスの通信担当格子点の数：46

2 次元領域分割

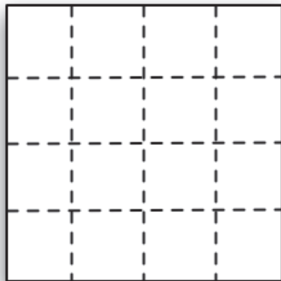
2 次元領域分割の場合、同じく 4 つの MPI プロセスで並列化。

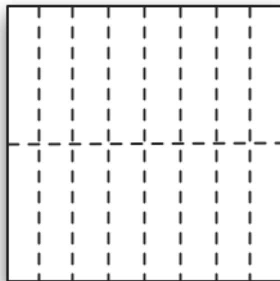


赤のプロセスの通信担当格子点の数：38

2 次元領域分割の方法

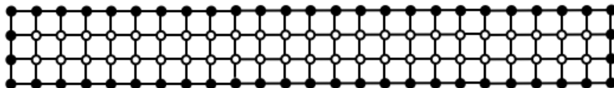
どちらがよいか？





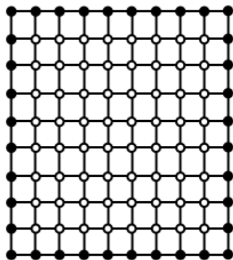
2 次元領域分割の方法

計算格子（白丸）：46 個
通信格子（黒丸）：54 個
合計：100 個



2 次元領域分割の方法

計算格子（白丸）：64 個
通信格子（黒丸）：36 個
合計：100 個

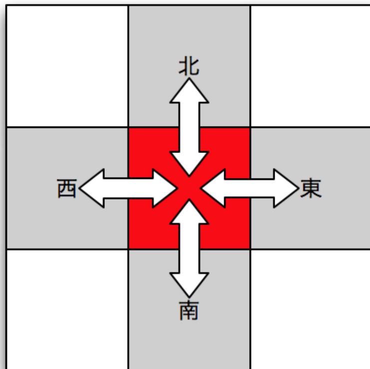


面積の等しい長方形の中で、４辺の長さの合計が最も小さいものは正方形。

2 次元領域分割による並列コード

領域分割による並列化を行うときに注意すべき点の一つは、 MPI プロセスの配置方法。

隣同士の通信がもっとも通信速度的に「近い」位置にプロセスを配置することが望ましい。



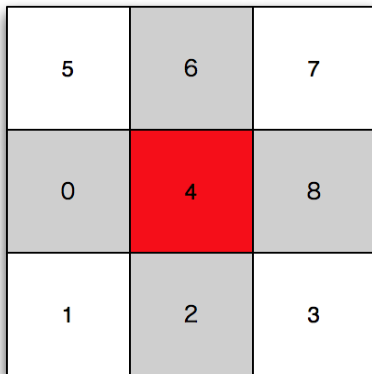
2 次元領域分割による並列コード

4 番のプロセスはランク番号 1,3,5,7 のプロセスと頻繁に通信する

2	5	8
1	4	7
0	3	6

2 次元領域分割による並列コード

もしも使用している並列計算機のネットワークの構成上、 4 番のプロセスはむしろランク番号 0,2,6,8 のプロセスと通信した方が速い場合には、以下のようにプロセスを配置する方が望ましい。



MPI_CART_CREATE

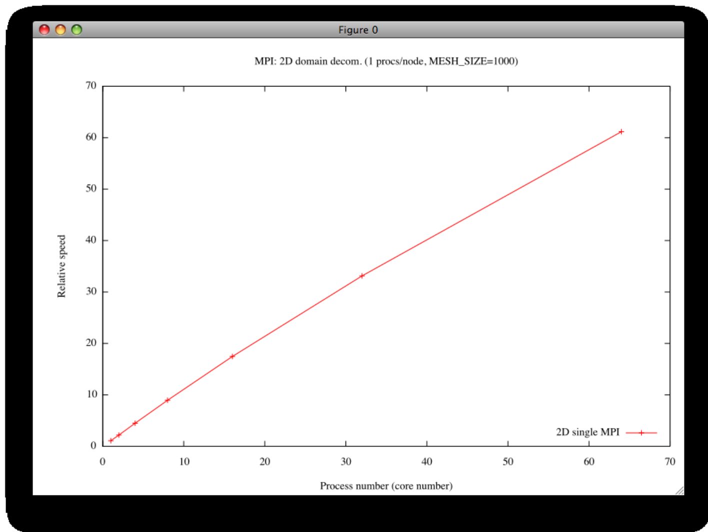
MPI 関数の一つ MPI_CART_CREATE を使うと（使用する計算機がネットワークの通信性能の関する情報を提供している場合には）通信効率の点で最適な配置でプロセスを自動的に分配してくれる[12]。

[12] しかし京コンピュータの場合はユーザが明示的にランク番号を割り当てることを想定しているようである。

速度のスケーリング

【参考】π-computer とは別のシステムにおける測定結果

【参考】MPI プロセス数と計算速度の関係（例）



【参考】1次元領域分割と2次元領域分割の比較（例）