計算科学演習 I

実践編 2

陰山

計算科学専攻

2015.07.16

事務連絡

課題受領確認

リソースグループ

- 今日 13:00 から 17:00 の間だけ諸君だけで large キューを占有
- π-computer の演習用アカウント（ID）は当分の間有効だが、今年度末までのどこかのタイミングで予告なく削除される。
- 研究等で継続利用したい場合は、指導教員と相談の上、利用申請を行う。

今日の準備

サンプルコード

cp -r /tmp/150716 自分のディレクトリ

先週の課題の解答

さきほどコピーした heat3.f90 を見よ。

時間計測

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 defaul
!!    3) pjsub heat5.sh
```

演習

heat5.f90 を次のコマンドでコンパイルし、実行せよ。

—

```
mpifrtpx heat5.f90
pjsub heat5.sh
```

—

どこで時間がかかっているか調べよ。
次のページに答えを載せる。

答え

stopwatch 出力の copy un to u ラベルの部分が遅い。
ソースコードでは以下の部分

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

ちなみに …

コンパイルで -O3 オプションをつければ早くなる。
mpifrtpx -O3 heat5.f90

演習

heat5.f90 で最も時間のかかる

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

の部分を 2 重 do loop に展開し、それを heat5b.f90 とせよ。

—

heat5b.f90 を次のコマンドでコンパイル＆実行せよ。
mpifrtpx heat5b.f90
pjsub heat5.sh

—

実行時間はどうか変わったか？

実行例

```
##################################################
job start at Tue Jul 15 21:07:29 JST 2015
##################################################
 # 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 15 21:07:39 JST 2015
```

演習

heat5b.f90 で時間がかかるのは stopwatch 出力の `jacobi` と `copy un to u` ラベルの二カ所（どちらも 2 重 do loop）である。この部分を OpenMP でスレッド並列化し、

heat6.f90

とせよ。（MPI と OpenMP のハイブリッド並列化！）

ハイブリッド並列化用のジョブスクリプト（heat6.sh）は用意した。(OpenMP スレッド数を 1 2 4 8 16 と変えて自動投入する。)

コンパイル＆実行方法：

```
mpifrtpx -Kopenmp heat6.f90
pjsub heat6.sh
```

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
.
```

今日の課題：スケーリングの確認

1. heat6.sh のジョブキューの指定を small から large に変更せよ。（今日の 17 時までは large キュー占有。）
2. 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（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）を、gnuplot で描け。

【評価基準】P の値が大きく、並列化スケーリングが線形に近いものほど高い評価とする。
（ヒント：並列化スケーリングが悪い時には、格子点数 NGRID を増やす。）

提出方法

【注意】large キューが占有して使えるのは今日の 17 時まで。
以下の内容を書いてメールで提出せよ。

(a) 氏名・学籍番号
(b) 使用した NGRID の値, ノード数 N, ノードあたりのスレッド数 M（=OMP_NUM_THREADS）の値
(c) gnuplot で描いたグラフのキャプチャ図

提出先： kage@port.kobe-u.ac.jp
ファイルフォーマット： pdf （表紙なし。1 ページ。圧縮しない。）
メールタイトル：計算科学演習 レポート
締めきり：7 月 22 日 23:59
うけとったら数日以内に返信する。

授業評価アンケートへの協力依頼

全員：https://questant.jp/q/CWHNM4WN

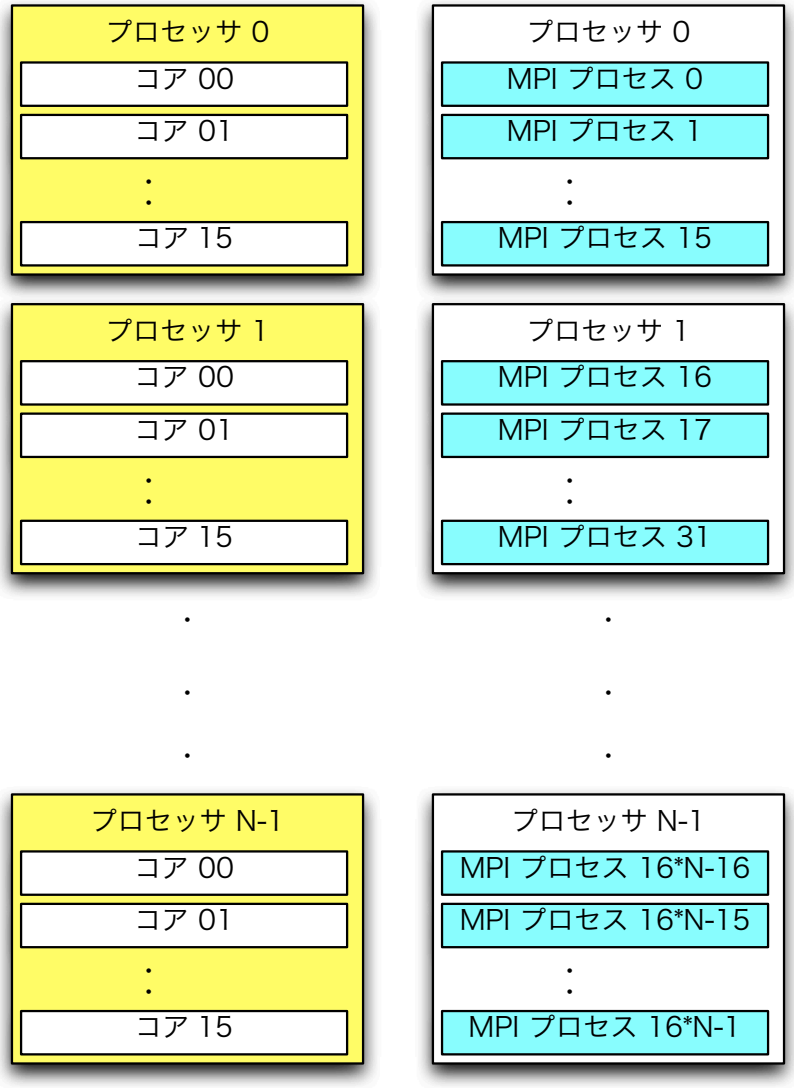
うりぼーネット（修士）： http://goo.gl/jxykms

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。（OpenMP を使わない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

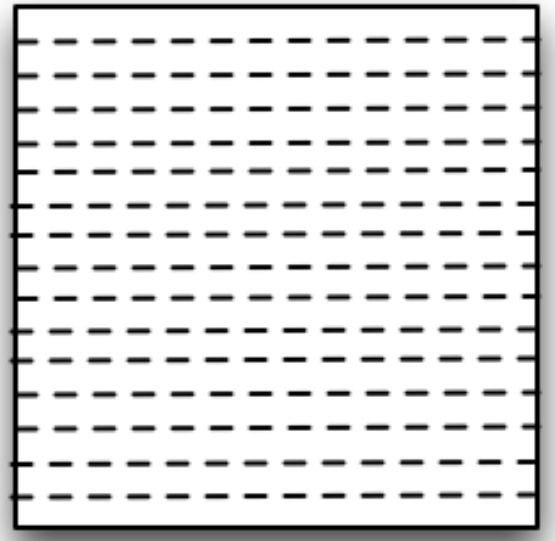

プロセッサ 0
コア 00
コア 01
コア 15
プロセッサ 0
MPI プロセス 0
MPI プロセス 1
MPI プロセス 15
プロセッサ 1
コア 00
コア 01
コア 15
プロセッサ 1
MPI プロセス 16
MPI プロセス 17
MPI プロセス 31
プロセッサ N-1
コア 00
コア 01
コア 15
プロセッサ N-1
MPI プロセス 16*N-16
MPI プロセス 16*N-15
MPI プロセス 16*N-1

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

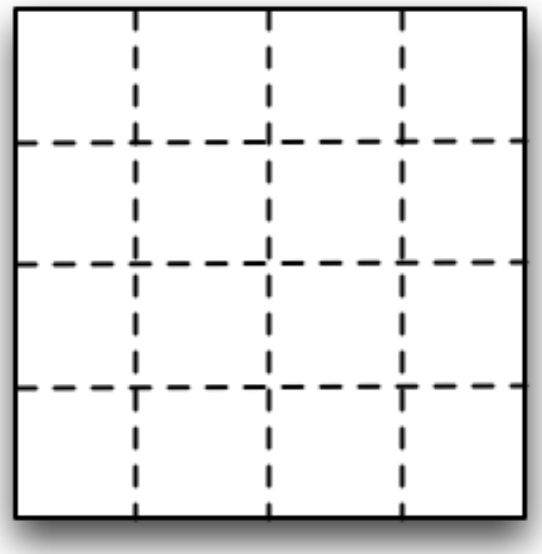
1次元並列

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）

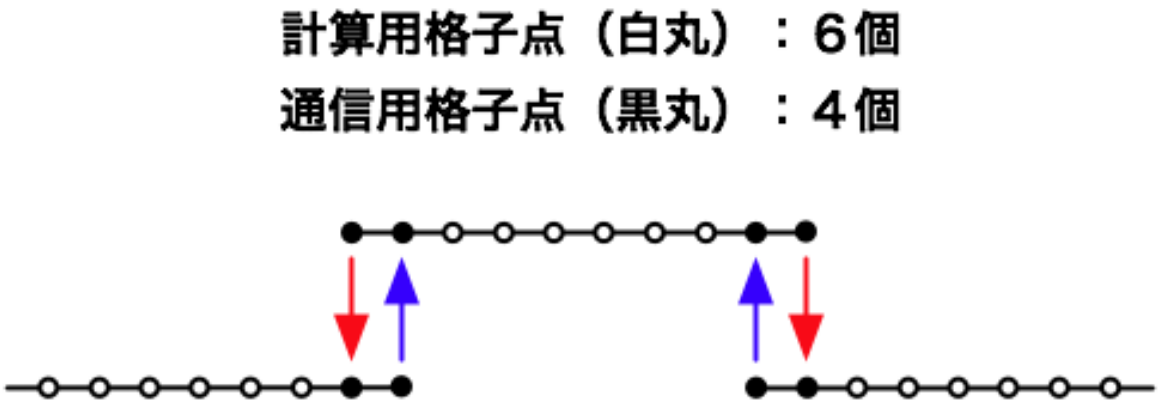
1次元分割と2次元分割のちがい

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。

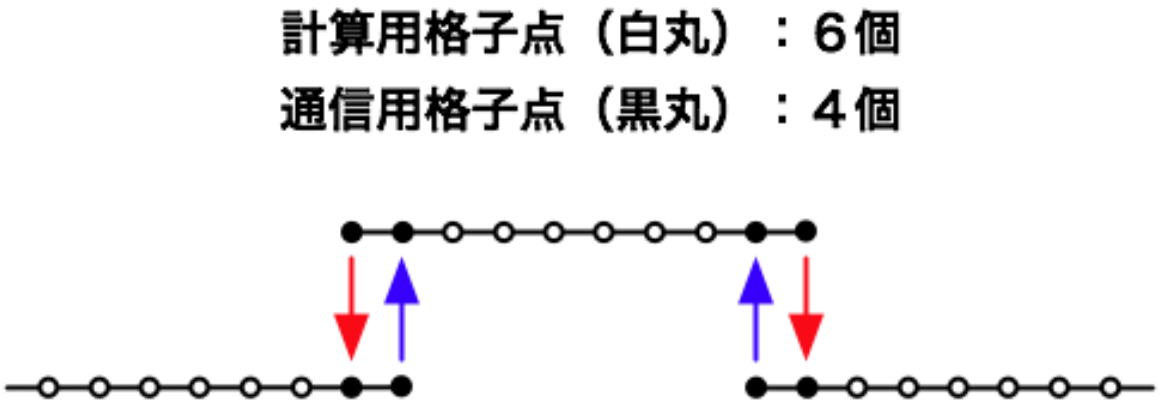


計算と通信

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。

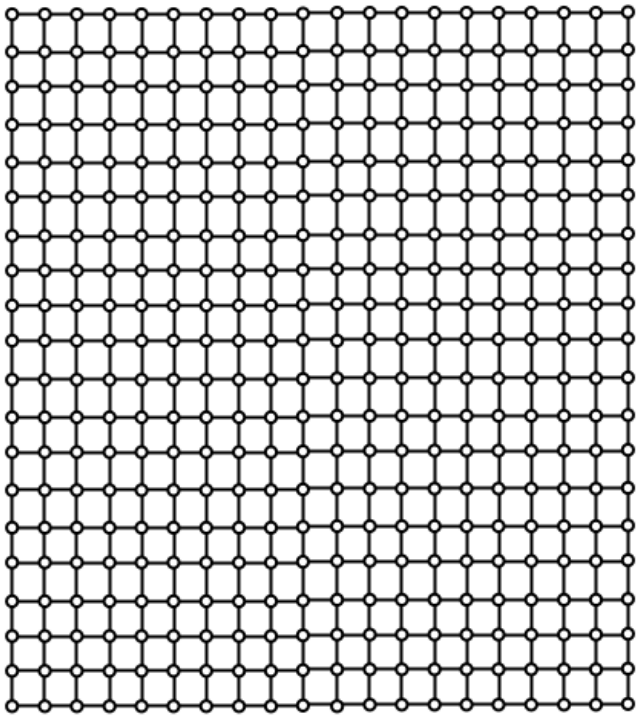
（一番外側から２番目の格子は計算も通信も行う。）



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。

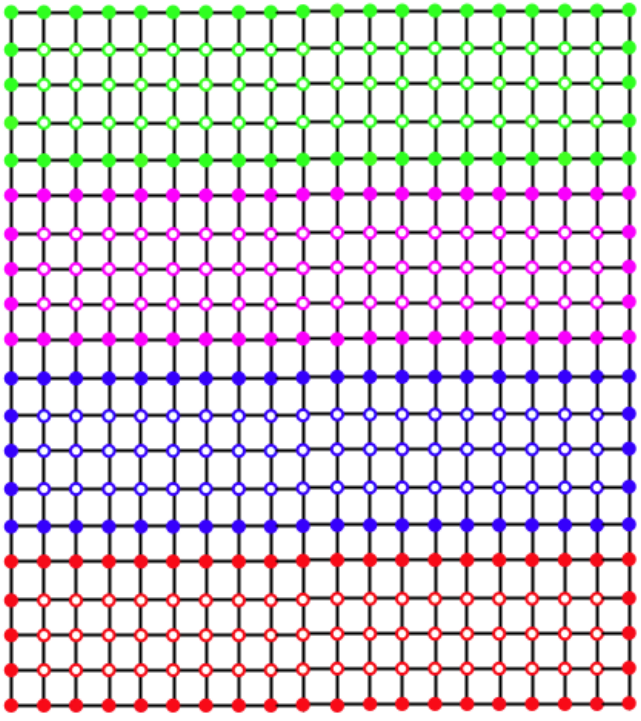
2 次元領域分割

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

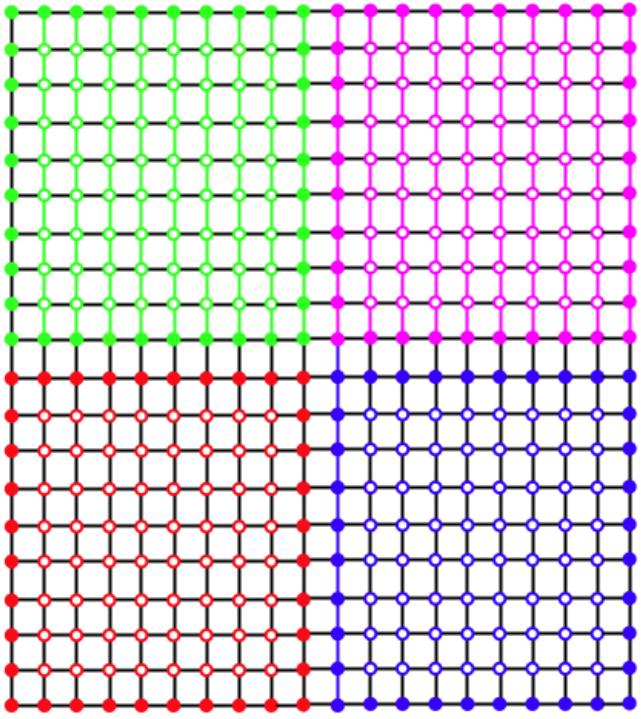
4 つの MPI プロセスで並列化。1 次元領域分割。



赤のプロセスの通信担当格子点の数：46

2 次元領域分割

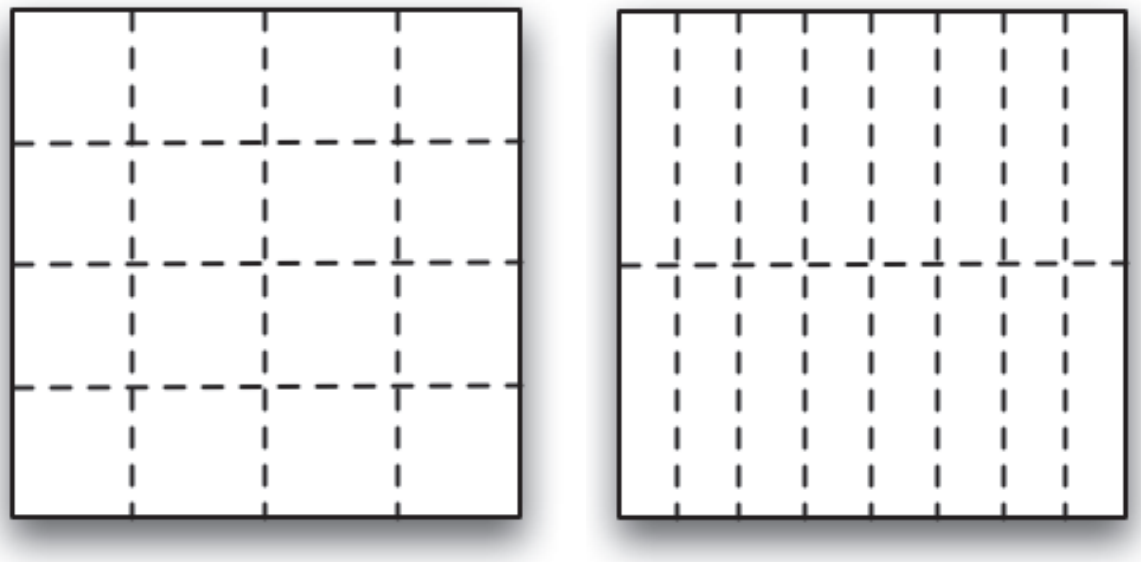
2 次元領域分割の場合、同じく 4 つの MPI プロセスで並列化。



赤のプロセスの通信担当格子点の数：38

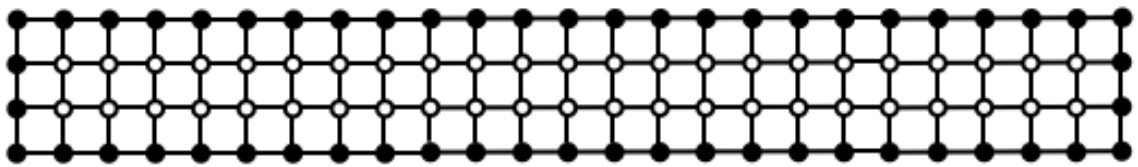
2次元領域分割の方法

どちらがよいか？



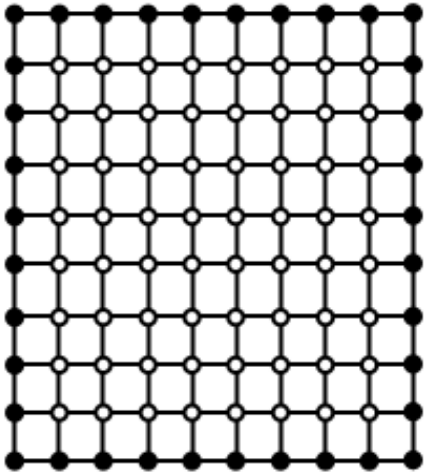
2次元領域分割の方法

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

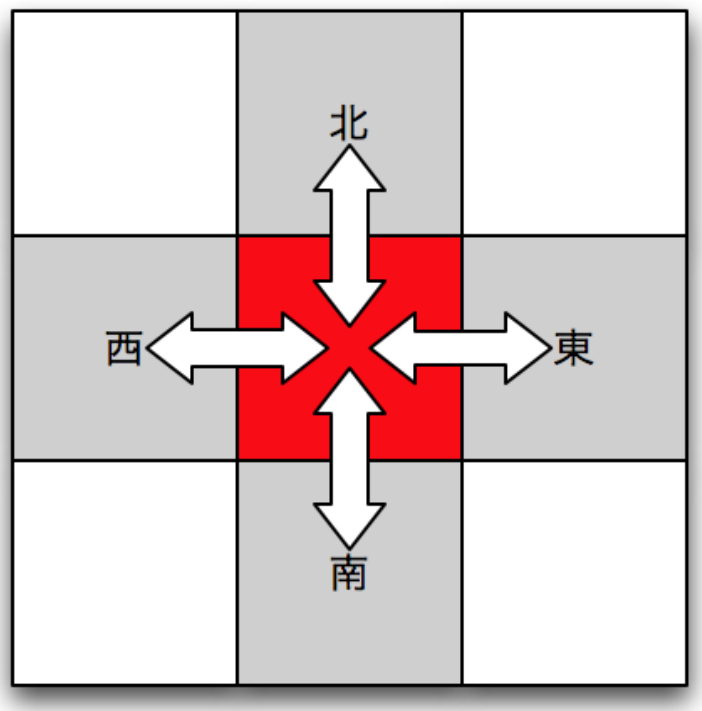
計算格子（白丸）：64 個
通信格子（黒丸）：36 個
合計：100 個



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。

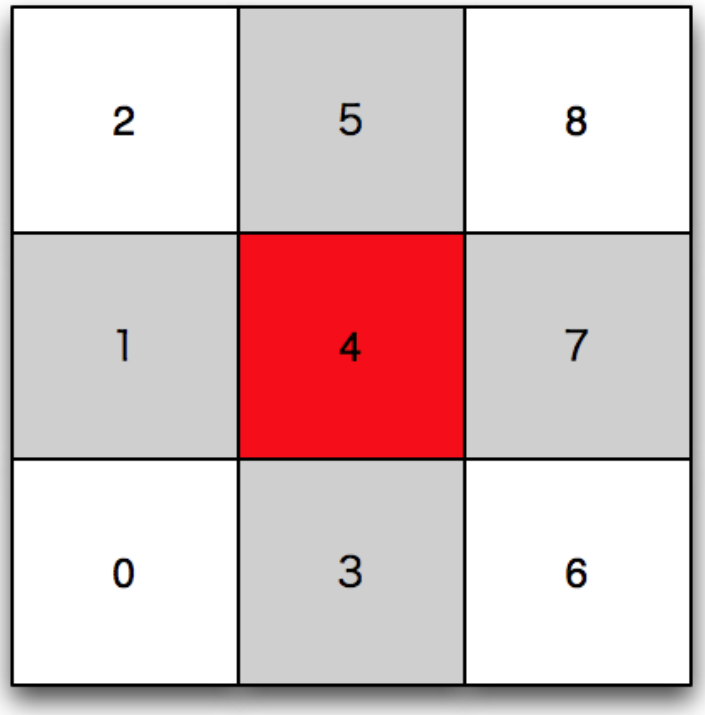
2次元領域分割による並列コード

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。



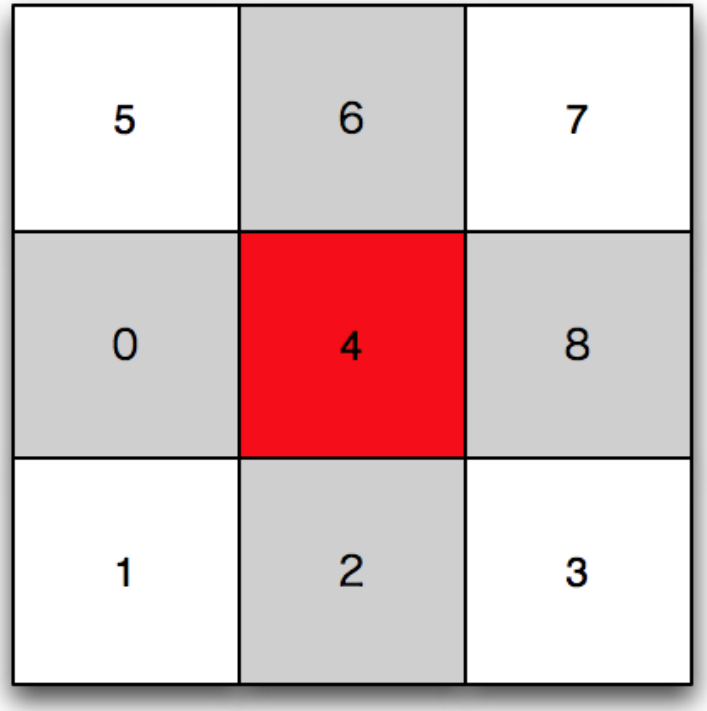
2次元領域分割による並列コード

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。



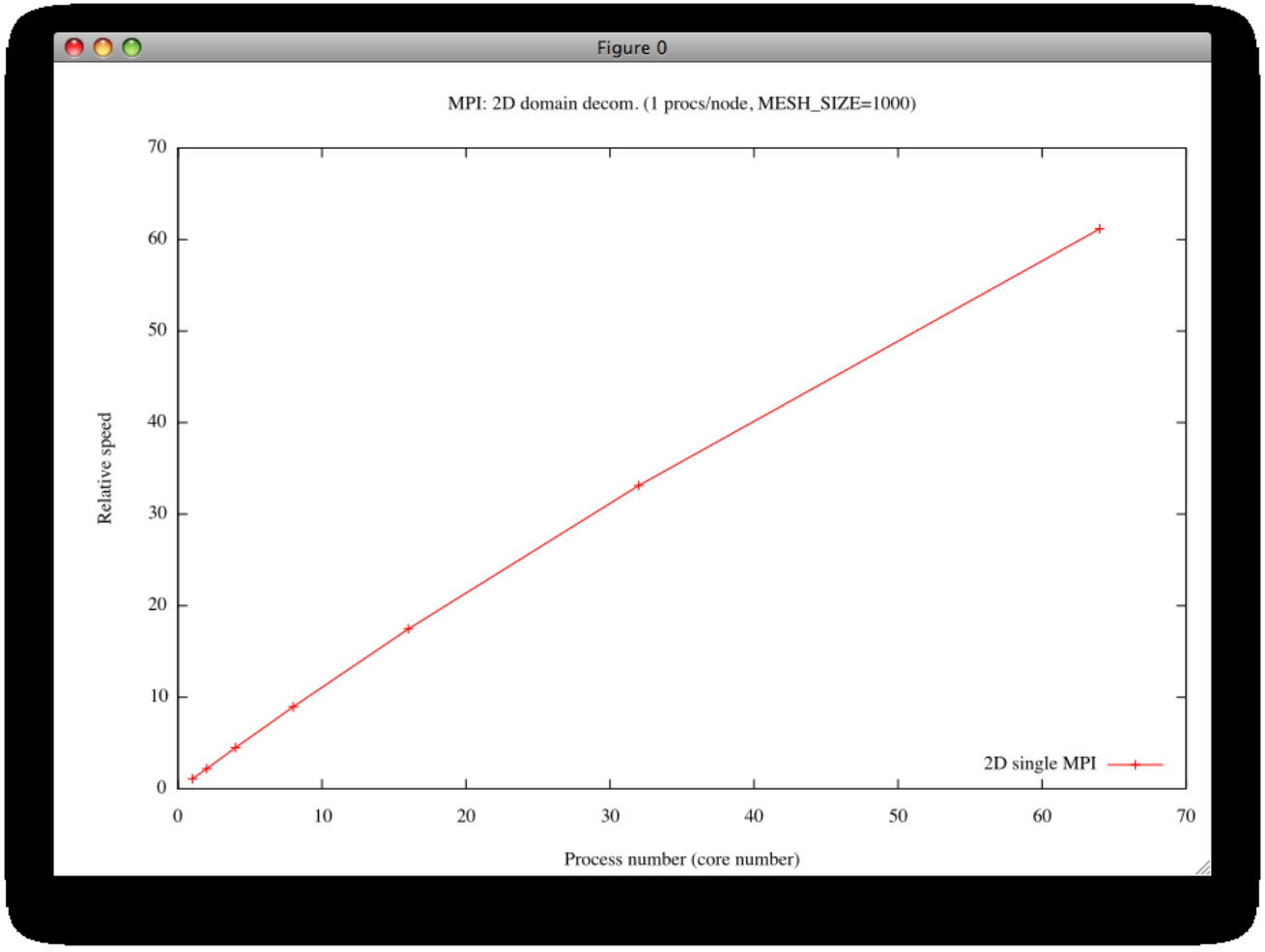
MPI_CART_CREATE

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 [1]。

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）