**UNIVERSITAS INDONESIA**

# PR 1 PEMROGRAMAN MPI

## LAPORAN TUGAS PEMROGRAMAN PARALEL


**KELOMPOK III**
**Muhammad Fathurachman 1506706276**
**Otniel Yosi Viktorisa 1506706295**
**Yohanes Gultom 1506706345**



**FAKULTAS ILMU KOMPUTER**
**PROGRAM STUDI MAGISTER ILMU KOMPUTER**
**DEPOK**
**APRIL 2016**

# DAFTAR ISI

# DAFTAR GAMBAR

# BAB 1
# LINGKUNGAN PERCOBAAN

## 1.1 Lingkungan *Cluster*

Dalam eksperimen yang dilakukan, kelompok kami menggunakan empat buah lingkungan yaitu *cluster* Rocks University of California San Diego (UCSD), *cluster* Fasilkom Universitas Indonesia (UI) dan *cluster* Rocks pada *laptop*.

### 1.1.1 Cluster Rocks University of California San Diego (UCSD)

*Cluster* Rocks[1] milik University of California San Diego (UCSD) ini dapat diakses pada alamat *nbcr-233.ucsd.edu* menggunakan protokol SSH dari komputer yang telah didaftarkan *public key* nya. Berdasarkan informasi dari aplikasi *monitoring* Ganglia [2], cluster ini terdiri 10 *nodes* dengan total 80 prosesor.

Pada *cluster* ini sudah terpasang pustaka komputasi paralel OpenMPI[3] dan MPICH[4] serta paket dinamika molekular AMBER. Program paralel MPI dan eksperimen AMBER dijalankan mekanisme antrian *batch-jobs* untuk menjamin ketersediaan sumberdaya komputasi (*computing nodes*) saat program dieksekusi. Pengaturan eksekusi program paralel ini ditangani oleh *Sun Grid Engine*[5] yang juga tersedia dalam paket Rocks.

### 1.1.2 Cluster Fasilkom Universitas Indonesia (UI)

*Cluster* milik Fakultas Ilmu Komputer (Fasilkom) UI ini berada di jaringan lokal yang tidak bisa diakses langsung dari luar. *Cluster* ini berbasis Linux dan terdiri dari empat buah *nodes* dengan total 32 prosesor.

Pada *cluster* ini sudah tersedia pustaka OpenMPI untuk menjalankan program paralel. Program dijalankan langsung tanpa mekanisme *batch-jobs* seperti pada *cluster* UCSD dengan menjalankan program dari direktori khusus (supaya dapat direplikasi ke seluruh *nodes* pada *cluster* ).

[1] www.rocksclusters.org
[2] http://nbcr-233.ucsd.edu/ganglia
[3] https://www.open-mpi.org/
[4] https://www.mpich.org/
[5] http://www.rocksclusters.org/roll-documentation/sge/5.4/

### 1.1.3 Cluster Rocks pada Laptop

Rocks merupakan distribusi Linux CentOS yang dikustomisasi untuk membangun *cluster high performance computing (HPC)* yang bersifat *opensource*. Rocks menyediakan berbagai macam paket (*rolls*) yang digunakan dalam berbagai *task* HPC.

Dalam eksperimen ini Rocks 6.2 Sidewinder dipakai untuk membangun *cluster* dengan dua buah *virtual machine (VM)* VirtualBox[6] pada *laptop* dengan sistem operasi Ubuntu (Prosesor Intel Core i7-3610QM dengan memori DDR3 8 GB serta penyimpanan HDD 1 TB). Konfigurasi *nodes* pada *cluster* ini adalah sbb:

1. *Front-end Node* (1 CPU, 1GB RAM, 30GB HDD)
   *Node* ini memiliki GUI (*Graphical User Interface*) berperan sebagai antarmuka pengguna dan manajer dari *compute node*. Program HPC idealnya tidak menggunakan sumberdaya dari *node* ini karena digunakan untuk menjalankan berbagai program antarmuka dan manajemen *cluster* .

2. *Compute Node*: 4 CPU, 1GB RAM, 30GB HDD
   *Node* ini tidak memiliki GUI dan berperan khusus sebagai sumberdaya komputasi program HPC.

## 1.2 Lingkungan Pengembangan

Program paralel yang digunakan di dalam eksperimen ini dibuat menggunakan bahasa C yang di-*compile* menggunakan pusaka OpenMPI pada sistem operasi Linux Ubuntu dan Mint.

Kode program-program dan laporan eksperimen ini kami simpan menggunakan layanan GitHub di alamat *https://github.com/yohanesgultom/parallel-programming-assignment*. Hal ini kami lakukan untuk mempermudah kolaborasi dalam pembuatan program dan laporan.

[6]https://www.virtualbox.org/

# BAB 2
# PERKALIAN MATRIKS & VEKTOR

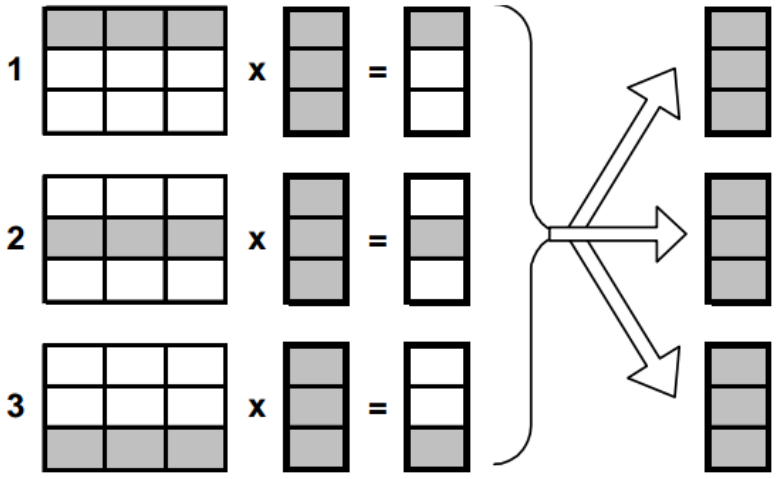
## 2.1 Pendahuluan

Topik eksperimen pertama adalah perkalian matriks-vektor dan perkalian matriks bujur sangkar dengan beberbagai algoritma paralel.

### 2.1.1 Perkalian Matriks-Vektor

#### 2.1.1.1 Row-Wise Decomposition

Algoritma paralel perkalian matriks-vektor yang paling sederhana, yaitu memecah proses perkalian berdasarkan baris matriks (*row-wise*). Setiap prosesor akan bertanggung jawab untuk mengalikan sebuah baris matriks dan vektor pada satu waktu. Jika jumlah prosesor ($np$) lebih sedikit dari jumlah baris matriks ($r$) maka setiap prosesor bertugas mengalikan $n = \frac{r}{np}$ secara sekuensial.

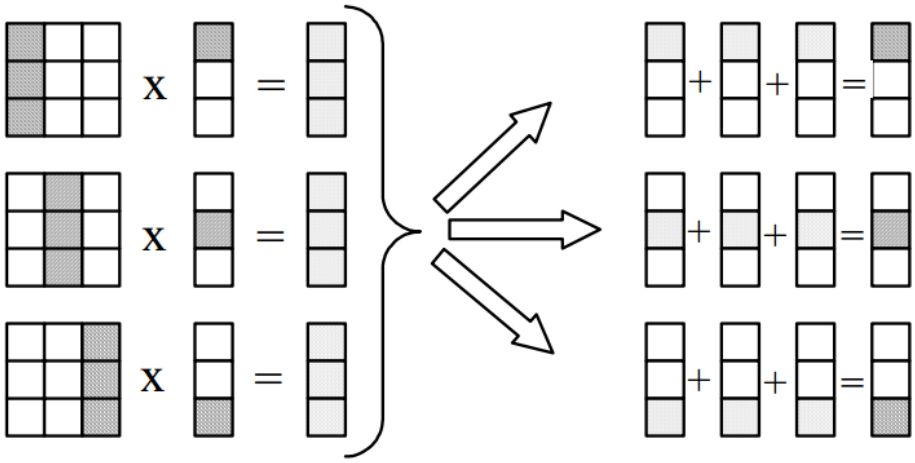



**Gambar 2.1:** Perkalian matriks-vektor Row-Wise Decomposition

#### 2.1.1.2 Column-wise Decomposition

Algoritma perkalian matriks-vektor ini merupakan alternatif dari *row-wise decomposition* di mana pemecahan proses perkalian dilakukan berdasarkan kolom matriks. Setiap proses akan mengalikan sebuah kolom matriks dan sebuah elemen

vektor pada satu waktu. Mirip dengan algoritma *row-wise decomposition*, jika jumlah prosesor ($np$) lebih sedikit dari jumlah kolom matriks ($c$) maka setiap prosesor bertugas mengalikan $n = \frac{c}{np}$ secara sekuensial.



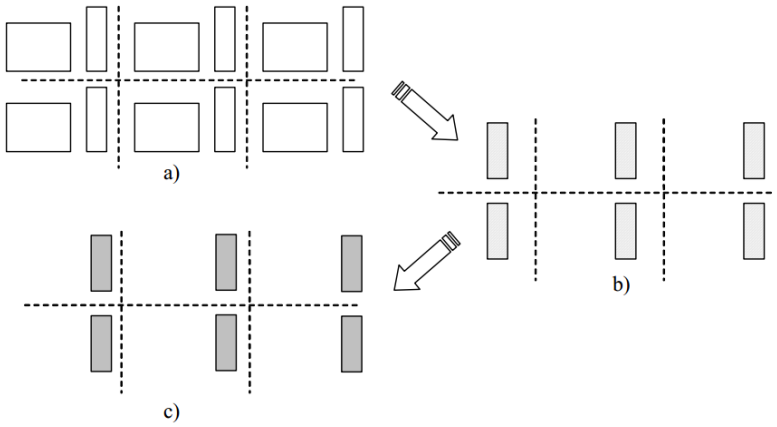
**Gambar 2.2:** Perkalian matriks-vektor Column-Wise Decomposition

#### 2.1.1.3 Checkerboard Decomposition

Algoritma perkalian matriks-vektor *checkerboard decomposition* ini membagi matriks menjadi submatriks dengan ukuran yang sama dan mengalikannya dengan subvektor yang sesuai. Hasil perkalian tersebut kemudian akan dijumlahkan dan dipetakan ke vektor hasil.

Prekondisi dari algoritma ini adalah jumlah elemen matriks ($n$) harus bisa dibagi rata ke sejumlah prosesor ($np$) atau dengan kata lain memenuhi persamaan 2.1. Hasil pembagian ini ($x$) akan menjadi ukuran submatriks (dan subvektor) yang dikerjakan di tiap proses secara paralel.

$$x = \sqrt{\frac{n}{np}}, \text{ where } x \in \mathbb{Z} \tag{2.1}$$

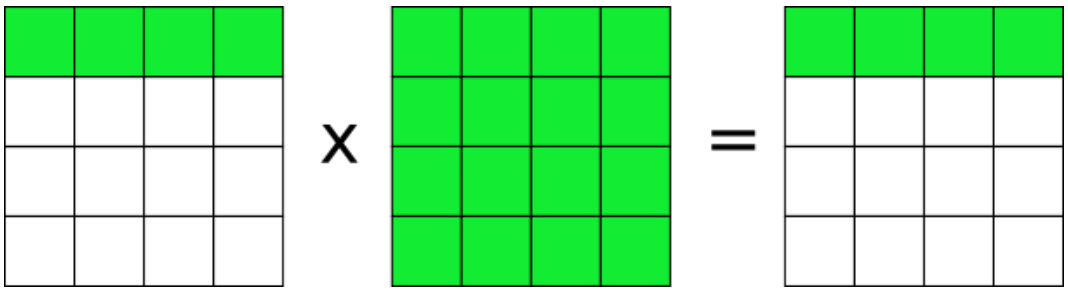



**Gambar 2.3:** Perkalian matriks-vektor Checkerboard Decomposition

### 2.1.2 Perkalian Matriks Bujursangkar

#### 2.1.2.1 Row-Wise Decomposition

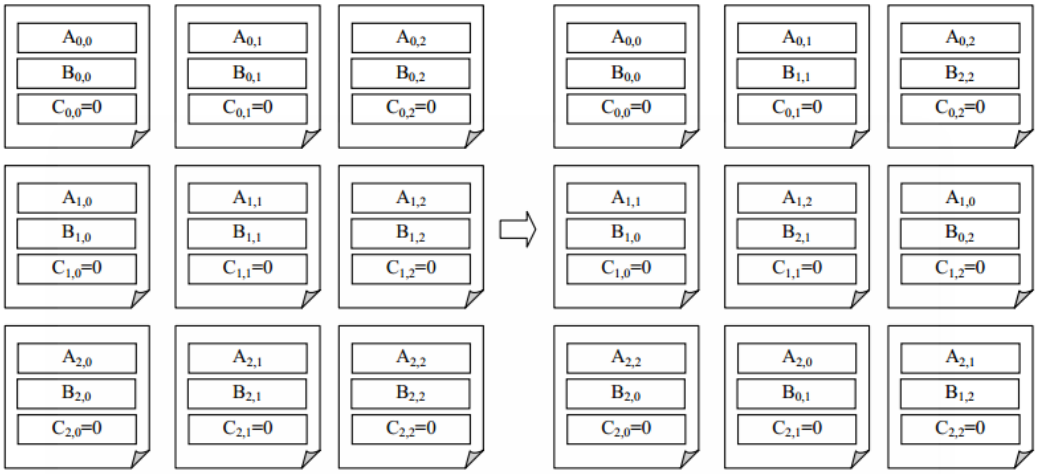
Mirip dengan algoritma perkalian matriks-vektor *row-wise decomposition*, algortima ini juga membagi pekerjaan berdasarkan baris dari matriks pertama seperti yang diilustrasikan pada gambar 2.4. Setiap prosesor akan mengalikan sebuah baris dari matriks pertama $A$ dengan seluruh kolom dari matriks kedua $B$. Hasil dari seluruh prosesor kemudian dikonkatenasi menjadi matriks baru $C$.



**Gambar 2.4:** Perkalian matriks bujursangkar Row-Wise Decomposition

#### 2.1.2.2 Cannon

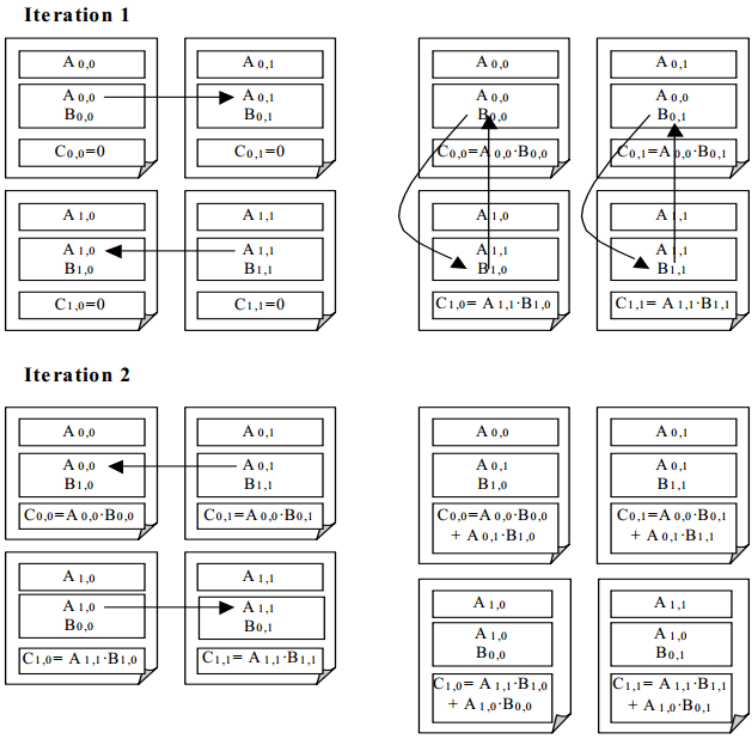
Algoritma Cannon menggunakan dekomposisi seperti algortima matriks-vektor *checkerboard decomposition* di mana matriks $A$ dan $B$ dibagi menjadi menjadi submatriks bujursangkar. Perbedaan pada algoritma Cannon adalah prekondisi di mana jumlah proses ($np$) harus merupakan bujursangkar sempurna (*perfect square*). Tujuan utama dari algoritma ini adalah untuk meningkatkan efisiensi penggunaan memori pada proses paralel.




**Gambar 2.5:** Perkalian matriks bujursangkar Cannon

### 2.1.2.3 Fox

Algoritma Fox memiliki kemiripan dengan algoritma Cannon dalam hal dekomposisi matriks menjadi submatriks bujursangkar dan pemetaan prosesor yang harus dapat membentuk bujursangkar sempurna (*perfect square*). Yang membedakan algoritma Fox dan Cannon adalah skema distribusi awal submatriks ke prosesor

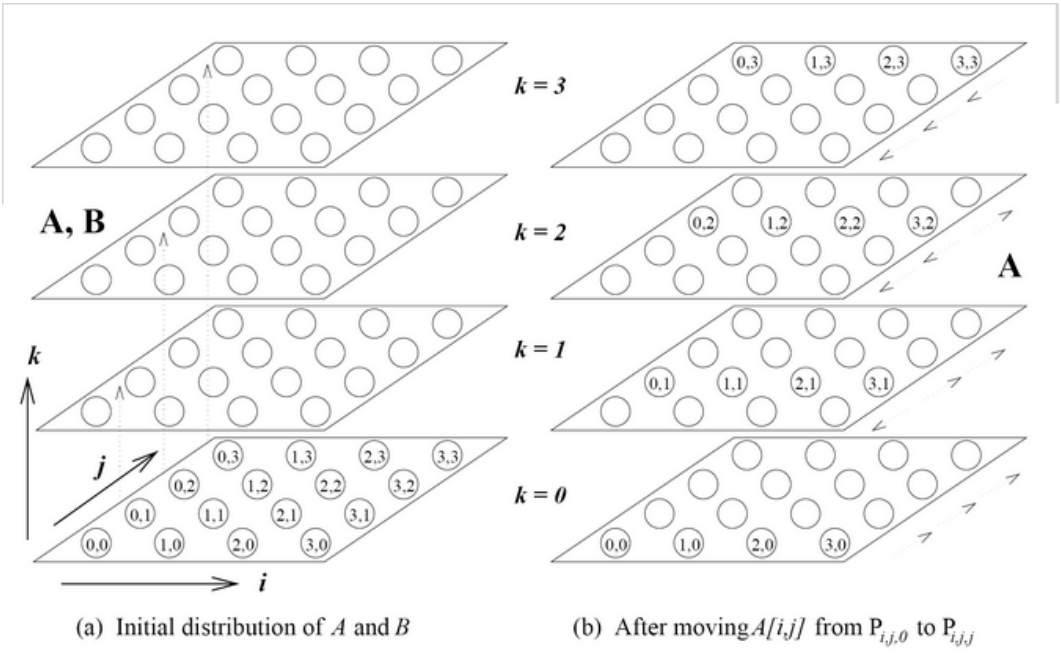



**Gambar 2.6:** Perkalian matriks bujursangkar Fox

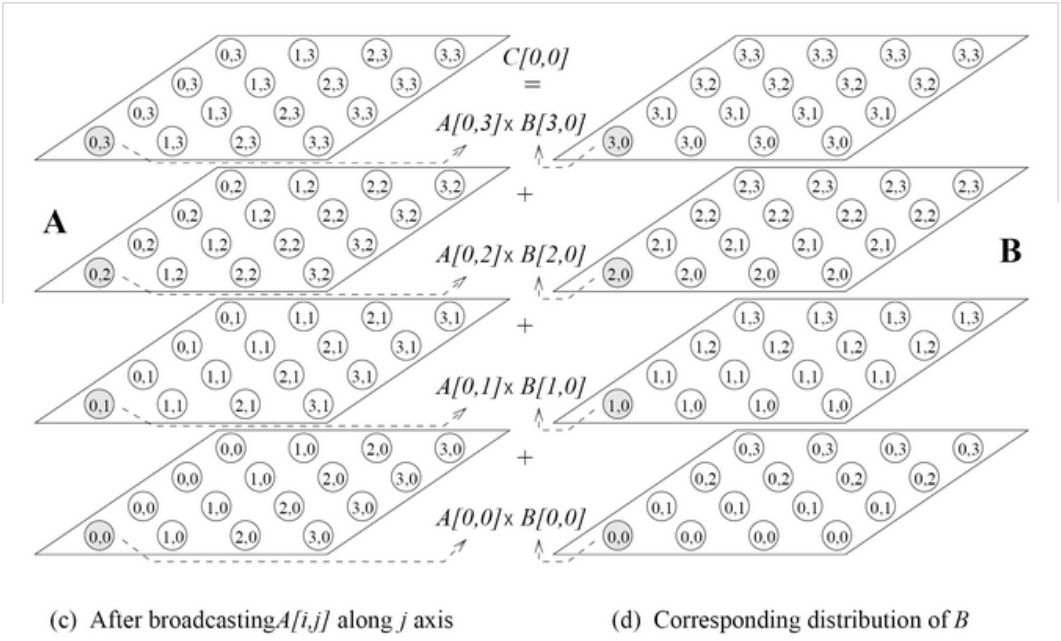
#### 2.1.2.4 DNS

Algoritma yang diberi nama berdasarkan nama pembuatnya (Dekel, Nassimi and Aahni) ini, diajukan dalam rangka meningkatkan lagi efisiensi penggunaan memori pada perkalian matriks bujursangkar secara paralel. Karakteristik algoritma ini adalah:

- Berdasarkan partisi *intermediate data*
- Melakukan perkalian skalar $n^3$ sehingga membutuhkan proses sebanyak $n \times n \times n$
- Membutuhkan waktu komputasi $O(\log n)$ dengan menggunakan $O(\frac{n^3}{\log n})$




**Gambar 2.7:** Perkalian matriks bujursangkar DNS iterasi 1




**Gambar 2.8:** Perkalian matriks bujursangkar DNS iterasi 2

## 2.2 Eksperimen

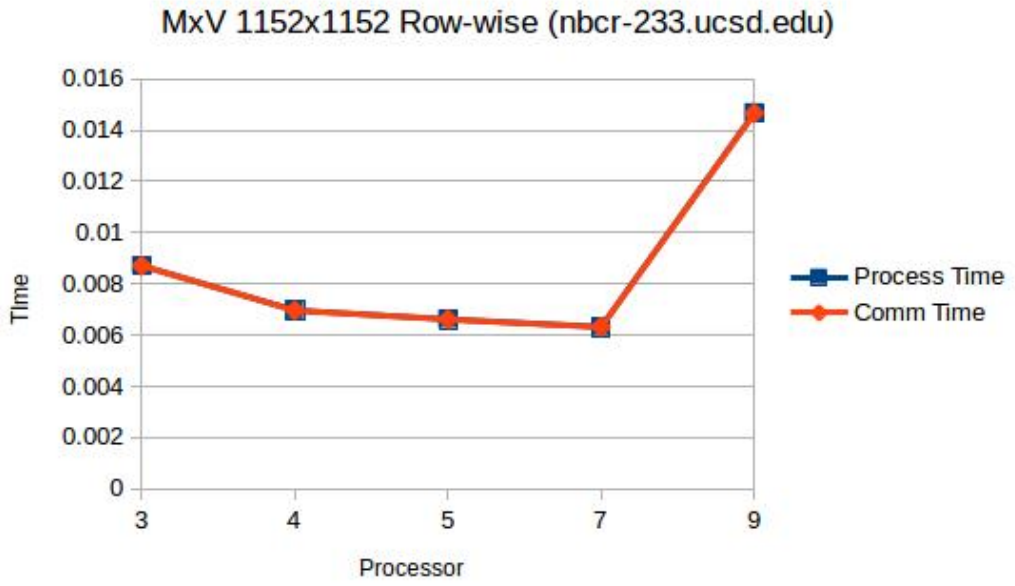
### 2.2.1 Perkalian Matriks-Vektor

#### 2.2.1.1 Row-Wise Decomposition

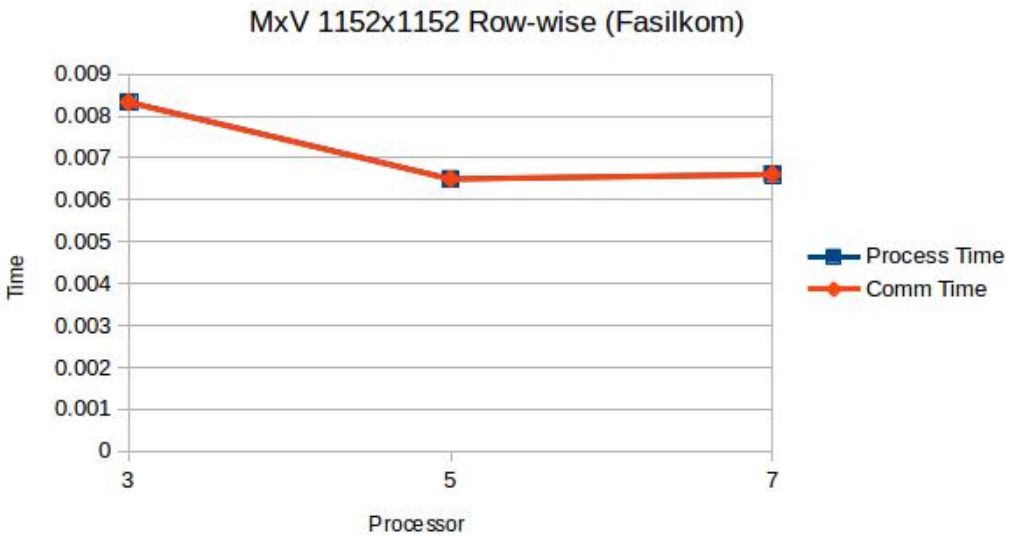
Deskripsi program yang digunakan:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem1/mv_rowwise.c`
- Satu prosesor berlaku sebagai *manager* dan sisanya berperan sebagai *worker*
- Tugas *manager* adalah menginisialisasi matriks dan vektor, mendistribusikannya secara *row-wise decomposition* menggunakan `MPI_Send` dan `MPI_Bcast` dan mengumpulkan hasil dari tiap *worker* dengan `MPI_Recv`
- Waktu eksekusi dan komunikasi dihitung (dalam detik) menggunakan `MPI_Wtime`

Eksperimen dilakukan di *cluster* UCSD dan Fasilkom dengan variasi jumlah prosesor di mana waktu yang diukur adalah nilai rata-rata dari lima kali eksekusi.




**Gambar 2.9:** Grafik hasil eksperimen Row-Wise Decomposition cluster UCSD




**Gambar 2.10:** Grafik hasil eksperimen Row-Wise Decomposition cluster Fasilkom UI

Pada grafik gambar 2.9, terlihat bahwa pada *cluster* UCSD terjadi penurunan waktu eksekusi/komunikasi (*speed-up* ) ketika digunakan 3, 4, 5, 7 prosesor (2, 3, 4, 6 *worker* ). Tapi ketika digunakan 9 prosesor (8 *worker* ), waktu yang dibutuhkan malah meningkat (tidak terjadi *speed-up* ). Kami tidak mencoba lebih dari 9 prosesor karena sepertinya hasilnya akan sama (tidak ada *speed-up* ).

Sedangkan pada *cluster* Fasilkom UI, kami hanya mencoba 3, 5, 7 prosesor karena masalah *permission* yang mengakibatkan hanya satu *node* (8 prosesor) yang

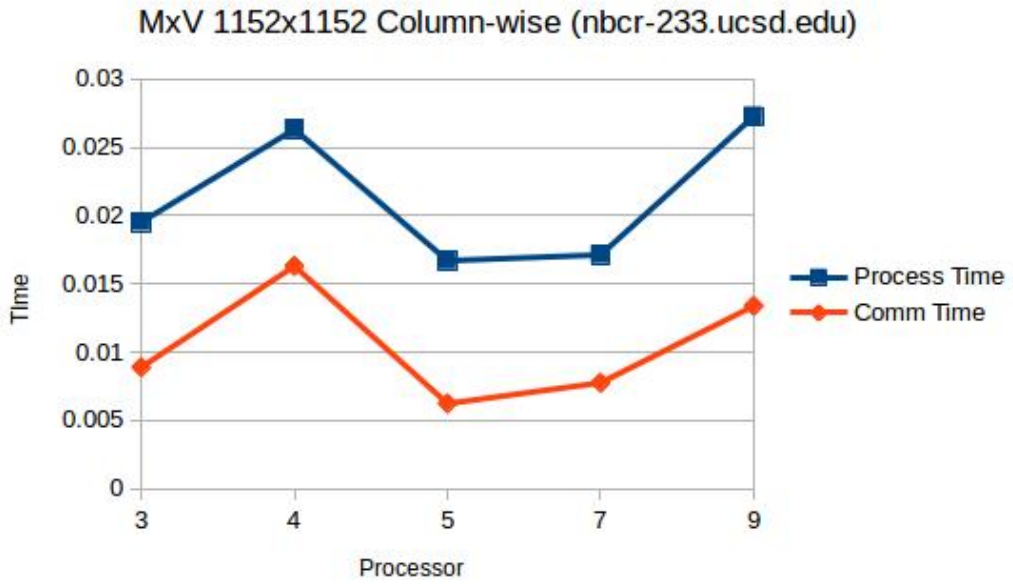
bisa digunakan. Seperti yang terlihat pada grafik gambar 2.10, *speed-up* hanya terjadi pada prosesor 3-5. Sedangkan dari 5-7 prosesor tidak terjadi *speed-up* (waktu ekskusi hampir sama).

#### 2.2.1.2 Column-wise Decomposition

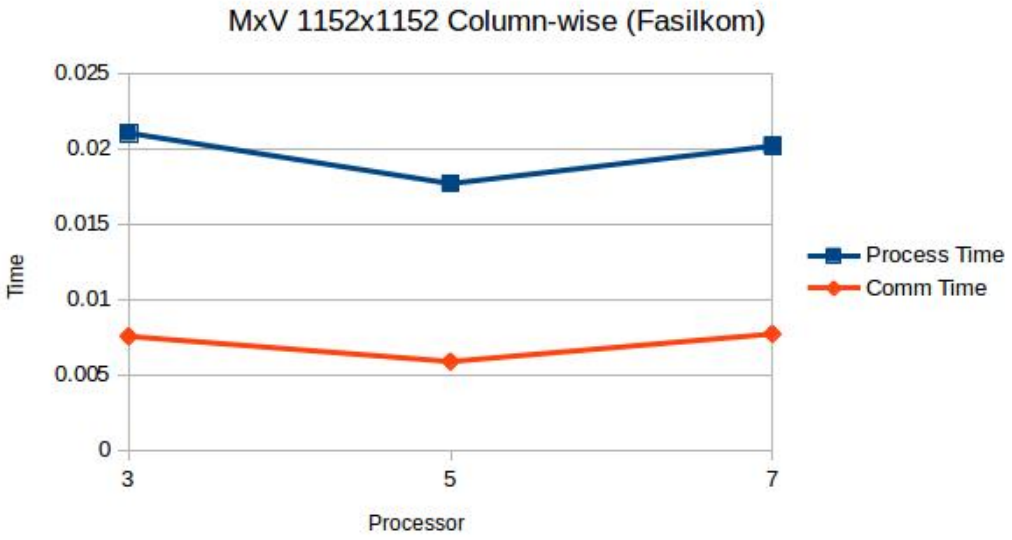
Deskripsi program yang digunakan:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem1/mv_columnwise.c`
- Satu prosesor berlaku sebagai *manager* dan sisanya berperan sebagai *worker*
- Tugas *manager* adalah menginisialisasi matriks dan vektor, mendistribusikannya secara *column-wise decomposition* menggunakan `MPI_Send` dan `MPI_Bcast` dan mengumpulkan hasil dari tiap *worker* dengan `MPI_Recv`
- Waktu eksekusi dan komunikasi dihitung (dalam detik) menggunakan `MPI_Wtime`

Eksperimen dilakukan di *cluster* UCSD dan Fasilkom dengan variasi jumlah prosesor di mana waktu yang diukur adalah nilai rata-rata dari lima kali eksekusi.




**Gambar 2.11:** Grafik hasil eksperimen Column-Wise Decomposition cluster UCSD




**Gambar 2.12:** Grafik hasil eksperimen Column-Wise Decomposition cluster Fasilkom UI

Pada grafik gambar 2.11, terlihat bahwa pada *cluster* UCSD terjadi penurunan waktu eksekusi/komunikasi (*speed-up* ) ketika digunakan 4-5 prosesor. Tapi ketika digunakan 3-4, 5-9 prosesor, waktu yang dibutuhkan malah meningkat (tidak terjadi *speed-up* ).

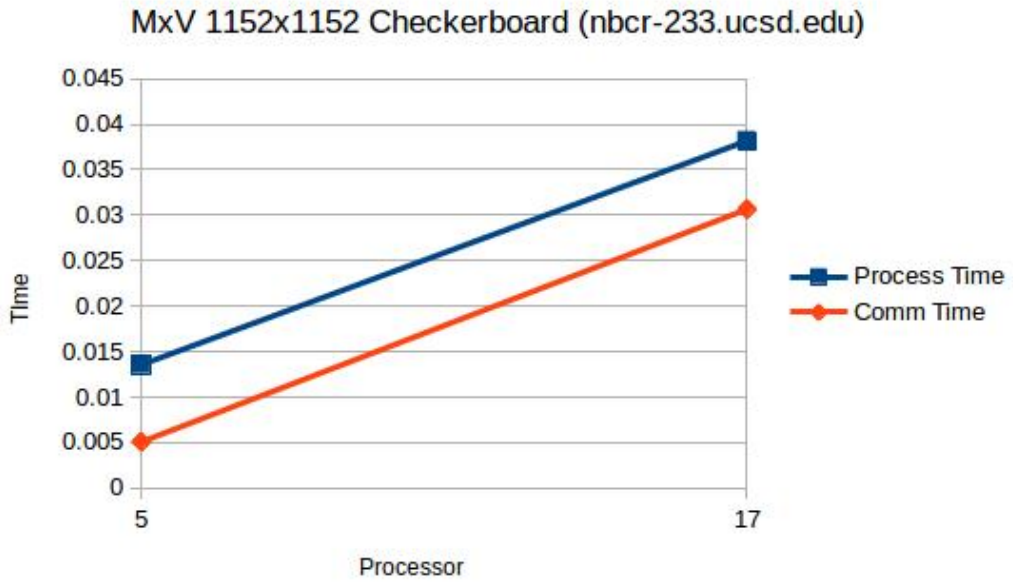
Sedangkan pada *cluster* Fasilkom UI, kami hanya mencoba 3, 5, 7 prosesor karena masalah *permission* yang mengakibatkan hanya satu *node* (8 prosesor) yang bisa digunakan. Seperti yang terlihat pada grafik gambar 2.12, *speed-up* hanya terjadi pada prosesor 3-5. Sedangkan dari 5-7 prosesor tidak terjadi *speed-up* (waktu ekskusi meningkat).

#### 2.2.1.3 Checkerboard Decomposition

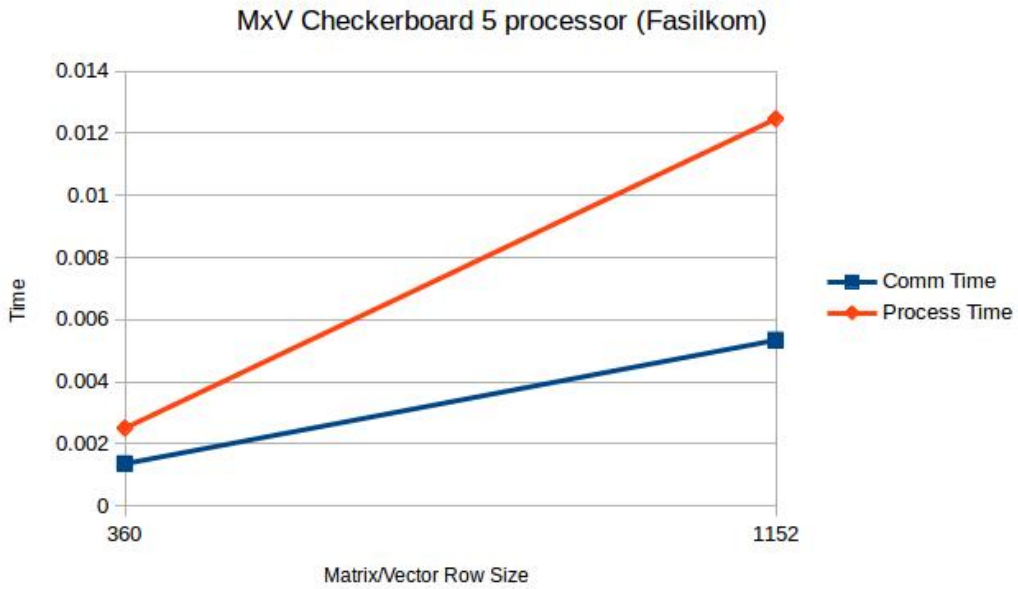
Deskripsi program yang digunakan:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem1/mv_checkerboard.c`
- Satu prosesor berlaku sebagai *manager* dan sisanya berperan sebagai *worker*
- Tugas *manager* adalah menginisialisasi matriks dan vektor, mendistribusikannya secara *checkerboard decomposition* menggunakan `MPI_Send` dan `MPI_Bcast` dan mengumpulkan hasil dari tiap *worker* dengan `MPI_Recv`
- Waktu eksekusi dan komunikasi dihitung (dalam detik) menggunakan `MPI_Wtime`

Eksperimen dilakukan di *cluster* UCSD dan Fasilkom dengan variasi jumlah prosesor di mana waktu yang diukur adalah nilai rata-rata dari lima kali eksekusi. Sedangkan pada *cluster* Rocks hanya dilakukan perbandingan algoritma.




**Gambar 2.13:** Grafik hasil eksperimen Checkerboard cluster UCSD

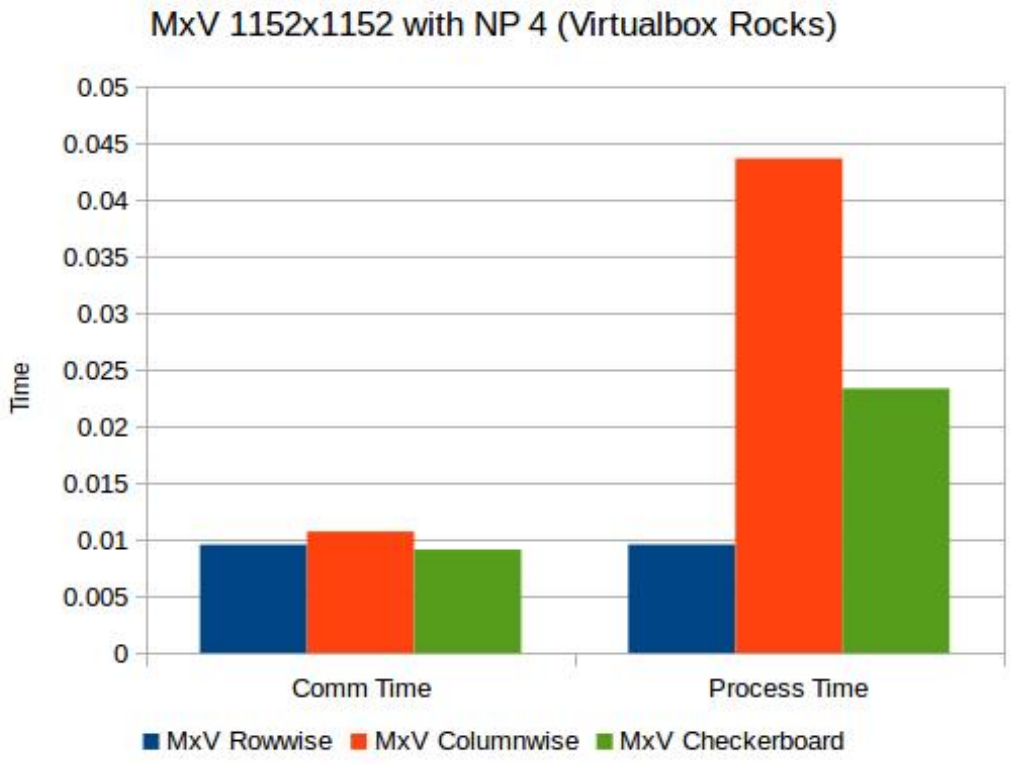



**Gambar 2.14:** Grafik hasil eksperimen Checkerboard cluster Fasilkom UI

Pada grafik gambar 2.13, terlihat bahwa waktu yang dibutuhkan malah meningkat (tidak terjadi *speed-up* ) ketika kami meningkatkan jumlah prosesor dari

5-7. Kami tidak mencoba prosesor yang lebih banyak karena melihat pola yang sama (tidak ada *speed-up* ).

Sedangkan pada *cluster* Fasilkom UI, kami hanya mencoba 5 prosesor karena masalah *permission* yang mengakibatkan hanya satu *node* (8 prosesor) yang bisa digunakan. Seperti yang terlihat pada grafik gambar 2.14, kami hanya mencoba melakukan eksekusi dengan ukuran matriks/vektor yang berbeda (360x360 dan 1152x1152).




**Gambar 2.15:** Grafik hasil eksperimen di cluster Rocks

Pada *cluster* Rocks kami mengamati perbandingan kinerja dari tiga algoritma yang sudah di bahas. Sesuai dengan grafik pada gambar 2.15, hasil tercepat diperoleh dari algoritma *row-wise decomposition*. Kami tidak melakukan eksperimen dengan jumlah prosesor yang berbeda pada *cluster* Rocks karena keterbatasan jumlah prosesor pada *laptop* yang kami gunakan.

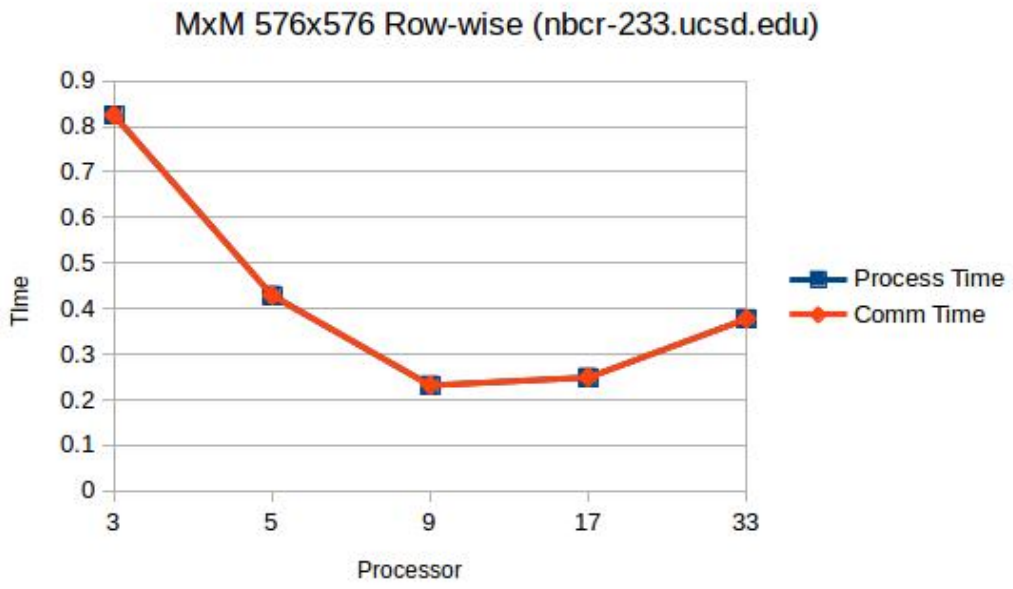
## 2.2.2 Perkalian Matriks Bujursangkar

### 2.2.2.1 Row-Wise Decomposition

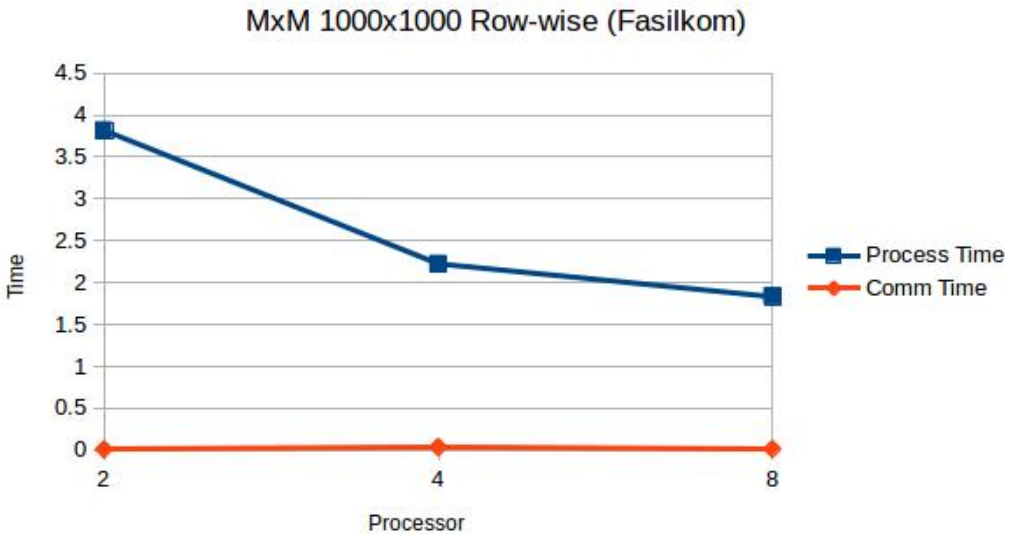
Deskripsi program yang digunakan:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem1/mm_rowwise.c`
- Satu prosesor berlaku sebagai *manager* dan sisanya berperan sebagai *worker*
- Tugas *manager* adalah menginisialisasi dua buah matriks bujursangkar, mendistribusikannya secara *row-wise decomposition* menggunakan `MPI_Send` dan `MPI_Bcast` dan mengumpulkan hasil dari tiap *worker* dengan `MPI_Recv`
- Waktu eksekusi dan komunikasi dihitung (dalam detik) menggunakan `MPI_Wtime`

Eksperimen dilakukan di *cluster* UCSD dan Fasilkom dengan variasi jumlah prosesor di mana waktu yang diukur adalah nilai rata-rata dari lima kali eksekusi.




**Gambar 2.16:** Grafik hasil eksperimen Row-Wise Decomposition cluster UCSD




**Gambar 2.17:** Grafik hasil eksperimen Row-Wise Decomposition cluster Fasilkom UI

Pada grafik gambar 2.16, terlihat bahwa pada *cluster* UCSD terjadi penurunan waktu eksekusi/komunikasi (*speed-up* ) ketika digunakan 3-9 prosesor (2-7 *worker* ). Tapi ketika digunakan 17 prosesor (16 *worker* ), waktu yang dibutuhkan malah meningkat (tidak terjadi *speed-up* )

Sedangkan pada *cluster* Fasilkom UI, kami hanya mencoba 2, 4, 8 prosesor karena masalah *permission* yang mengakibatkan hanya satu *node* (8 prosesor) yang bisa digunakan. Seperti yang terlihat pada grafik gambar 2.17, *speed-up* terjadi di seluruh eksperimen (pada prosesor 2-8).

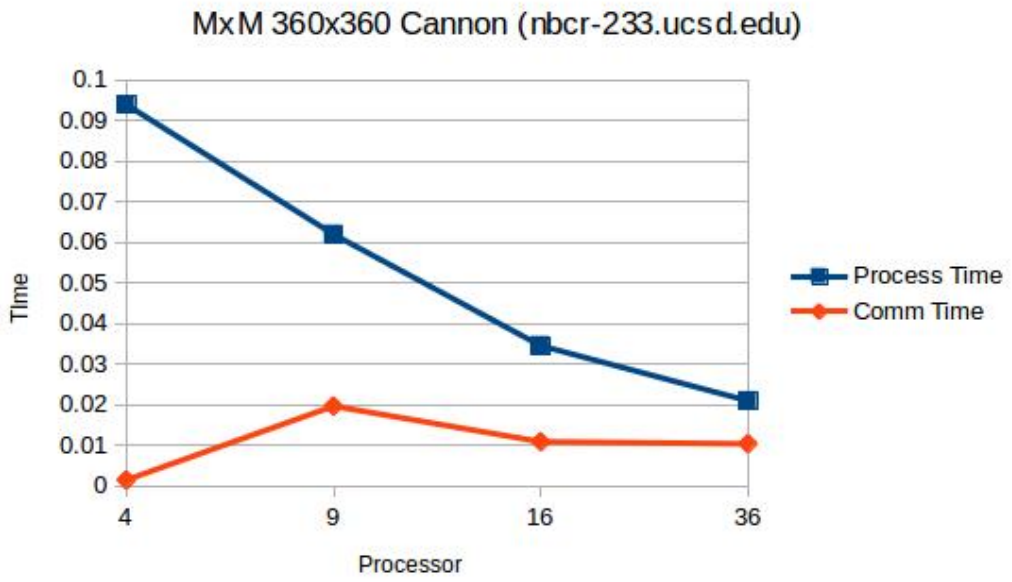
#### 2.2.2.2 Cannon

Deskripsi program yang digunakan:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem1/mm_cannon.c`
- Satu prosesor berlaku sebagai *manager* tapi sekaligus *worker* sedangkan sisanya hanya berperan sebagai *worker*
- Tugas *manager* adalah menginisialisasi dua buah matriks bujursangkar, mempersiapkan topologi proses kartesian dengan `MPI_Cart_create`, mendistribusikan pekerjaan dengan `MPI_Sendrecv_replace` dan `MPI_Cart_shift` untuk menentukan destinasi proses

- Waktu eksekusi dan komunikasi dihitung (dalam detik) menggunakan `MPI_Wtime`

Percobaan hanya dilakukan di *cluster* UCSD karena variasi jumlah prosesor harus bersifat *perfect square*, yaitu 4, 9, 16 dan 36.




**Gambar 2.18:** Grafik hasil eksperimen algoritma Cannon cluster UCSD

Seperti yang terlihat pada grafik gambar 3.3, terjadi *speed-up* pada penambahan prosesor 4-36. Selain itu juga terlihat bahwa waktu komunikasi memiliki proporsi yang lebih sedikit dibandingkan dengan proporsi pada algoritma *row-wise decomposition*.

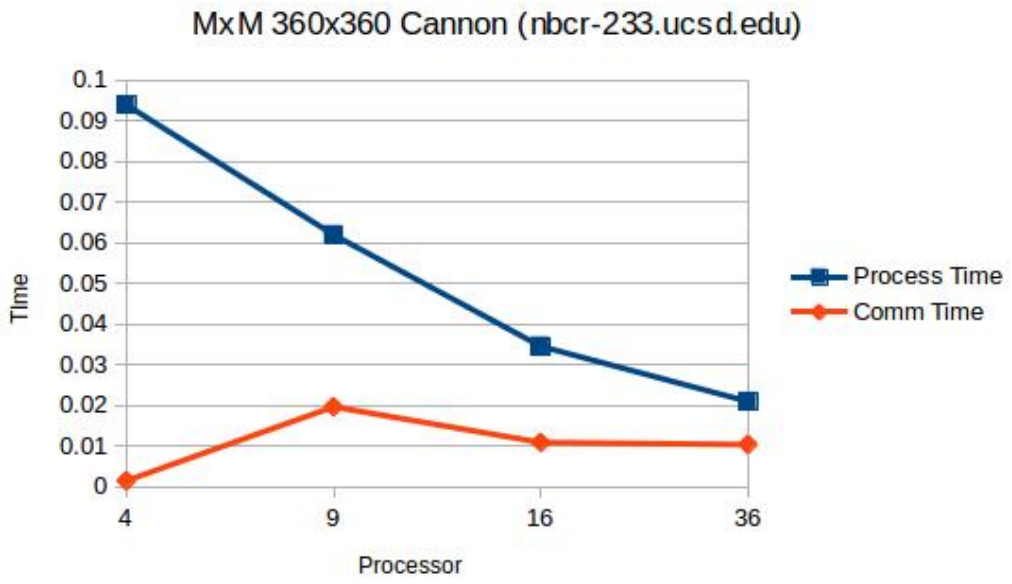
#### 2.2.2.3 Fox

Deskripsi program yang digunakan:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem1/mm_fox.c`
- Satu prosesor berlaku sebagai *manager* tapi sekaligus *worker* sedangkan sisanya hanya berperan sebagai *worker*
- Tugas *manager* adalah menginisialisasi dua buah matriks bujursangkar, mempersiapkan topologi proses kartesian dengan `MPI_Cart_create`, mendistribusikan pekerjaan dengan `MPI_Sendrecv_replace` dan `MPI_Cart_shift` untuk menentukan destinasi proses

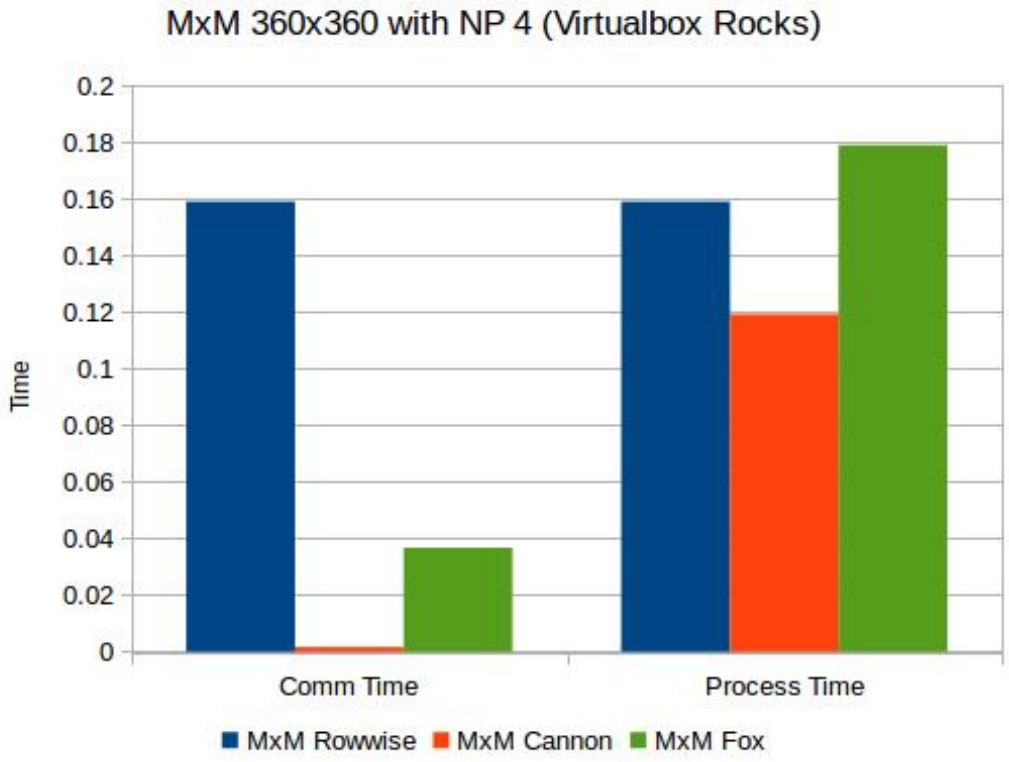
- Waktu eksekusi dan komunikasi dihitung (dalam detik) menggunakan `MPI_Wtime`

Percobaan hanya dilakukan di *cluster* UCSD karena variasi jumlah prosesor harus bersifat *perfect square*, yaitu 4, 9, 16 dan 36.




**Gambar 2.19:** Grafik hasil eksperimen algoritma Fox cluster UCSD

Seperti yang terlihat pada grafik gambar 2.19, terjadi *speed-up* pada penambahan prosesor 4-36. Selain itu juga terlihat bahwa waktu komunikasi memiliki proporsi yang lebih sedikit dibandingkan dengan proporsi pada algoritma *row-wise decomposition*.




**Gambar 2.20:** Grafik perbandingan algoritma pada cluster Rocks

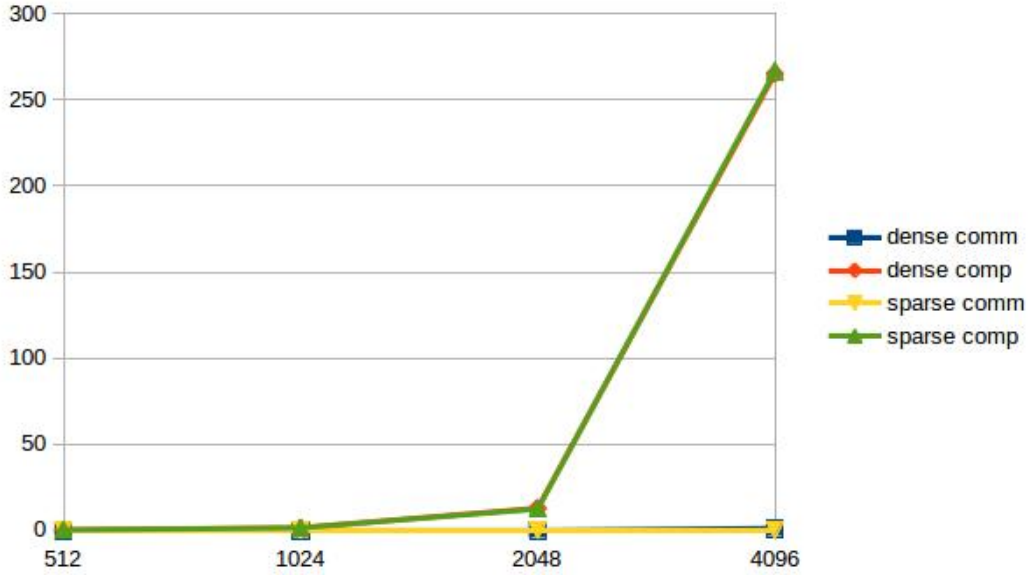
Pada *cluster* Rocks, kami mencoba membandingkan waktu proses antara algoritma *row-wise decomposition*, Cannon dan Fox dengan 4 prosesor. Hasil percobaan menunjukkan waktu proses dan komunikasi terendah dihasilkan oleh algoritma Cannon.

#### 2.2.2.4 DNS

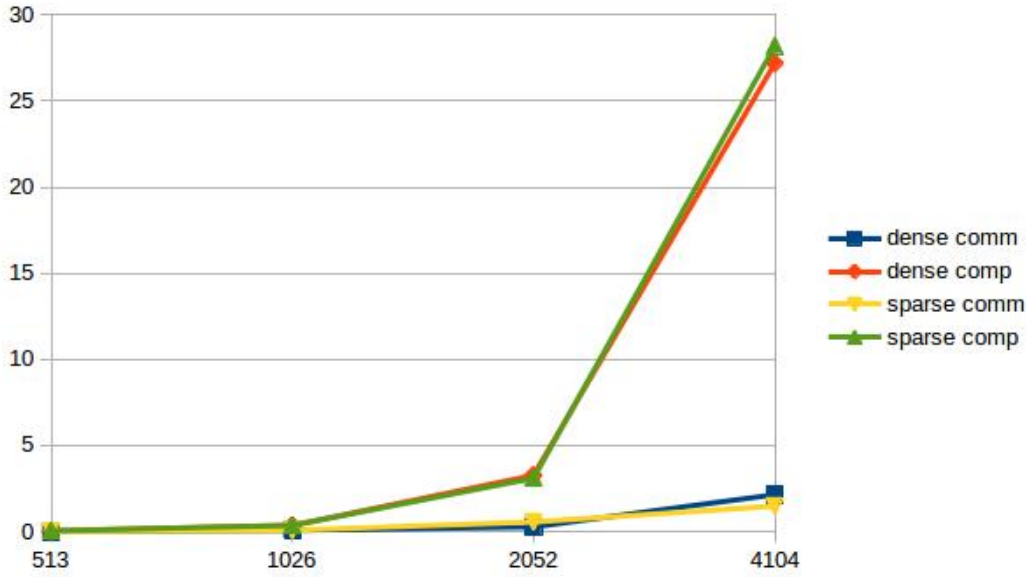
Deskripsi program yang digunakan:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/dns/dns.c`
- Jumlah prosesor ($np$) harus memenuhi $x = \sqrt[3]{np}$ di mana $x \in \mathbb{Z}$

Eksperimen dilakukan dengan prosesor 8, 27 di *cluster* UCSD dan 8 prosesor di *cluster* Fasilkom UI,

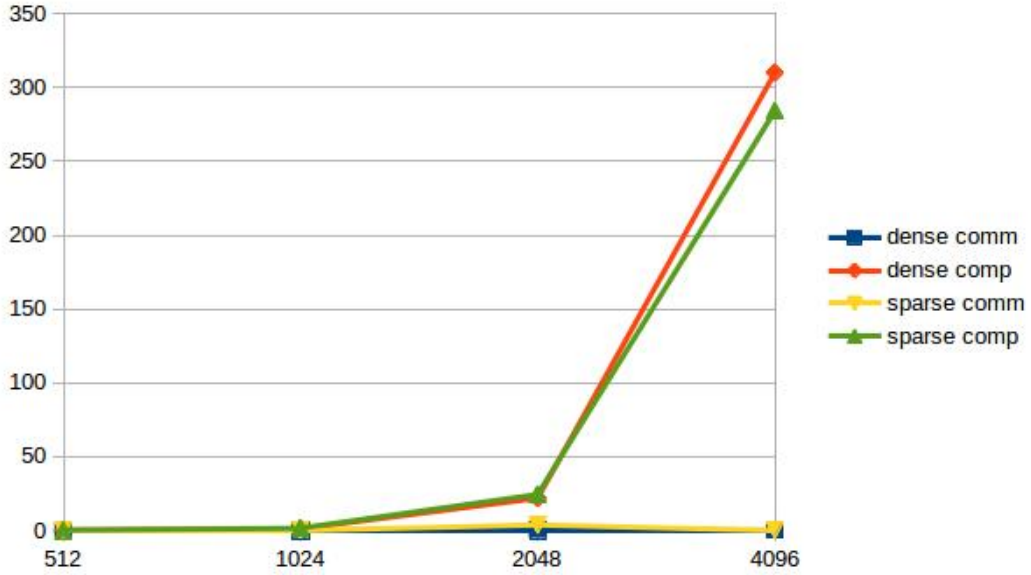



**Gambar 2.21:** Grafik hasil eksperimen algoritma DNS degan 8 prosesor cluster UCSD




**Gambar 2.22:** Grafik hasil eksperimen algoritma DNS degan 27 prosesor cluster UCSD

Hasil percobaan pada *cluster* UCSD yang ditunjukkan oleh grafik pada gambar 2.21 dan 2.22 menunjukkan bahwa proporsi waktu komunikasi hanya sedikit meningkat ketika ukuran matriks diperbesar. Dengan membandingkan dua grafik tersebut kita juga melihat adanya *speed-up* , khususnya pada ukuran matriks di atas 2048.




**Gambar 2.23:** Grafik hasil eksperimen algoritma DNS degan 8 prosesor cluster Fasilkom UI

## 2.3 Kesimpulan

Berdasarkan seluruh eksperimen pada topik ini, kami mengambil beberapa kesimpulan:

1. Algoritma paralel dapat mengurangi waktu komputasi total (terjadi *speed-up* ) khususnya pada algoritma yang proporsi waktu komunikasinya sedikit (Cannon, Fox, DNS)

2. *Speed-up* akan semakin terlihat jika jumlah data semakin besar

3. Pembagian bobot pekerjaan (*load balancing*) mempengaruhi kinerja algoritma paralel karena itu perlu disesuaikan perbandingan ukuran data dan jumlah prosesor yang digunakan

4. Jika jumlah prosesor terus ditingkatkan, algoritma paralel akan mencapai titik jenuh (tidak terjadi *speed-up* lagi)

5. Terdapat algoritma paralel didesain untuk kasus spesifik (matriks bujursangkar, matriks *dense/sparse*, jumlah prosesor *perfect square*)

# BAB 3
# PROCESS TOPOLOGIES & DYNAMIC PROCESS GENERATION

## 3.1 Pendahuluan

### 3.1.1 Process Topologies

Topologi proses (*Process Topologies*) adalah alternatif representasi dan akses kumpulan proses paralel (*process group*). Tujuan dari topologi proses adalah mempermudah pembuat program dari algoritma yang menggunakan topologi seperti algoritma Cannon, Fox (*kartesian*) dan DNS (*cube*)

Pada pustaka MPI, topologi proses diimplementasikan secara generik menggunakan topologi kartesian (*cartesian topology*) karena topologi ini sudah cukup untuk membentuk topologi lain seperti *line*, *ring*, *mesh*, *hypercube*.

Fungsi-fungsi dasar yang disediakan MPI untuk topologi kartesian adalah:

1. `MPI_Cart_create`: membuat topologi kartesian
2. `MPI_Cart_rank`: mendapatkan *rank* prosesor di topologi berdasarkan koordinat
3. `MPI_Cart_coords`: mendapatkan koordinat di topologi berdasarkan rank
4. `MPI_Cart_shift`: mendapatkan *rank* tetangga (*neighbors*) dari sebuah prosesor

### 3.1.2 Dynamic Process Generation

Pembuatan proses secara dinamis (*Dynamic Process Generation*) adalah fitur yang baru dikenalkan pada MPI v2. Fitur ini memungkinkan pembuatan proses (*process spawning*) secara dinamis (tidak perlu ditentukan di awal dengan opsi `-n` atau `-np`)

Fungsi dasar MPI untuk pembuatan proses secara dinamis

1. `MPI_Comm_spawn`: membuat proses secara dinamis sekaligus membuat domain komunikasi baru dengan proses tsb
2. `MPI_Comm_get_parent`: mendapatkan domain komunikasi dengan *parent* (pada *child process*)

## 3.2 Eksperimen

### 3.2.1 Process Topologies

Deskripsi program:

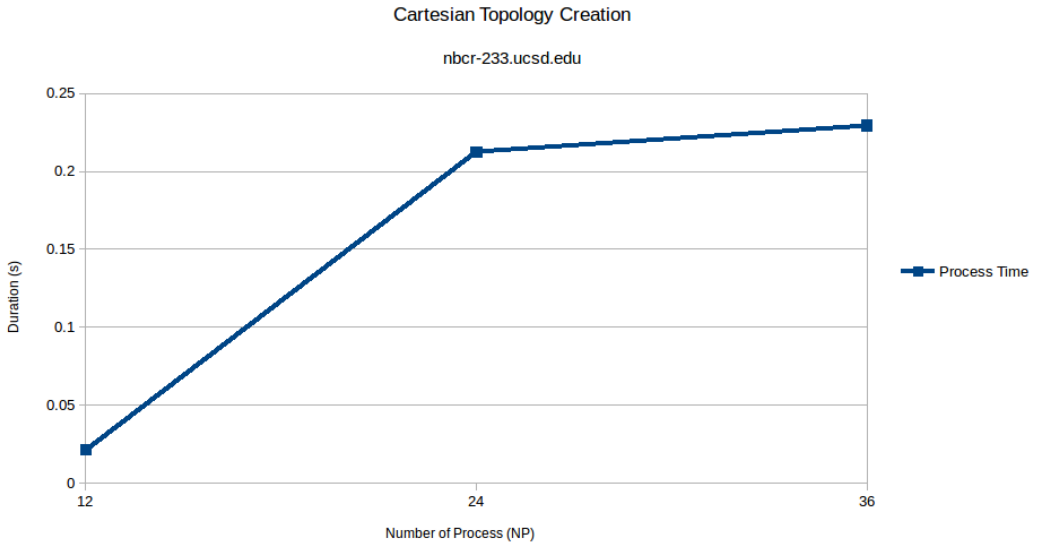
- Kode sumber `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem2/demo.c`
- Program ini membuat topologi cartesian 2 dimensi dengan ukuran baris ($a$) dan kolom ($b$) yang optimal untuk jumlah proses ($np$) adalah $np = axb$ dengan fungsi `MPI_Cart_create`, mencetak rank prosesor (*print*) topologi berdasarkan koordinat dengan `MPI_Cart_rank`, mengakses prosesor berdasarkan koordinat dengan `MPI_Cart_coords` dan mendapatkan tetangga (*neighbors*) prosesor dengan `MPI_Cart_shift`

Dengan program ini kami ingin mengamati perilaku fungsi-fungsi dasar topologi proses MPI dan mengamati waktu yang dibutuhkan untuk membuat topologi kartesian dengan variasi jumlah proses.

```
yohanes@Asus-K401LB:~/Workspace/parallel-programming-assignment/problem2$ mpirun -n 32 demo.o
6 x 5
MPI_Cart_rank:
(0,0)->0        (1,0)->5        (2,0)->10       (3,0)->15       (4,0)->20       (5,0)->25
(0,1)->1        (1,1)->6        (2,1)->11       (3,1)->16       (4,1)->21       (5,1)->26
(0,2)->2        (1,2)->7        (2,2)->12       (3,2)->17       (4,2)->22       (5,2)->27
(0,3)->3        (1,3)->8        (2,3)->13       (3,3)->18       (4,3)->23       (5,3)->28
(0,4)->4        (1,4)->9        (2,4)->14       (3,4)->19       (4,4)->24       (5,4)->29

P:9 neighbors (MPI_Cart_shift) are r: 14 d:-2 1:4 u:8
P:5 coordinates (MPI_Cart_coords) are (1,0)
32      0.103323
yohanes@Asus-K401LB:~/Workspace/parallel-programming-assignment/problem2$
```

**Gambar 3.1:** Contoh eksekusi program demo proses topologi




**Gambar 3.2:** Eksperimen waktu pembuatan topologi kartesian di cluster UCSD

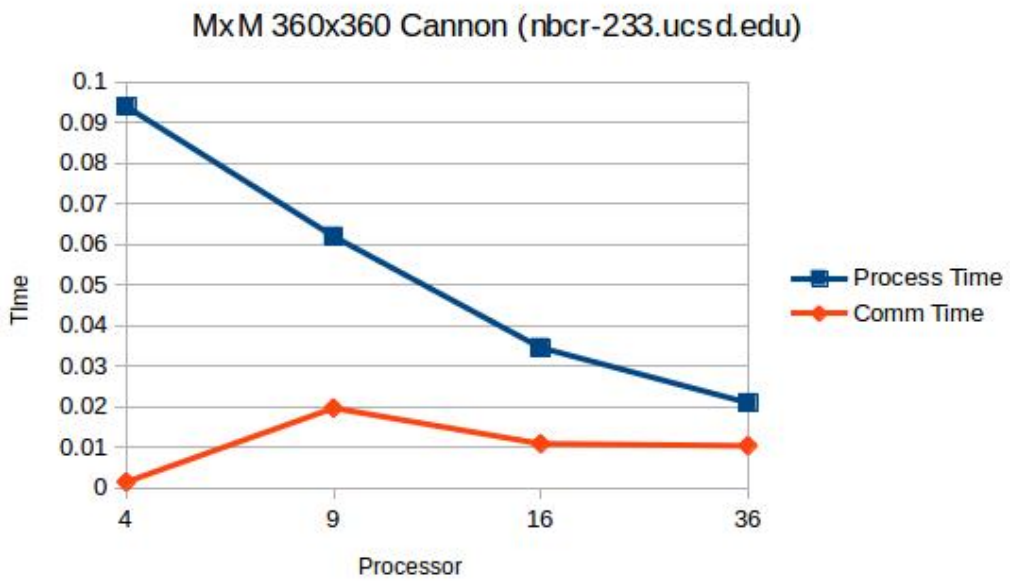
Pada percobaan pertama yang hasilnya ditunjukkan pada gambar 3.1, kami mengamati bahwa fungsi `MPI_Cart_shift` akan mengembalikan nilai negatif jika kita mencoba mendapatkan tetangga bawah dari proses terbawah atau tetangga teratas. Sedangkan jika kita mencoba mendapatkan tetangga kiri dari proses terkiri atau tetangga kanan dari proses terkanan, `MPI_Cart_shift` akan mengembalikan *rank* secara *cyclic*.

Pada percobaan berikutnya, grafik pada gambar 3.2 menunjukkan bahwa peningkatan waktu semakin sedikit ketika topologi yang dibuat semakin besar. Dengan kata lain implementasi topologi kartesian pada MPI didesain untuk lebih optimal untuk ukuran topologi yang besar.

Program kedua yang kami gunakan adalah perkalian matriks bujursangkar dengan algoritma Cannon yang memanfaatkan topologi kartesian:

- Sumber kode: `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem2/mm_cannon.c`
- Satu prosesor berlaku sebagai *manager* tapi sekaligus *worker* sedangkan sisanya hanya berperan sebagai *worker*
- Tugas *manager* adalah menginisialisasi dua buah matriks bujursangkar, mempersiapkan topologi proses kartesian dengan `MPI_Cart_create`, mendistribusikan pekerjaan dengan `MPI_Sendrecv_replace` dan `MPI_Cart_shift` untuk menentukan destinasi proses

- Waktu eksekusi dan komunikasi dihitung (dalam detik) menggunakan `MPI_Wtime`




**Gambar 3.3:** Grafik hasil eksperimen algoritma Cannon cluster UCSD

Percobaan ini sudah dilakukan juga pada topik sebelumnya dan hasilnya menunjukkan bahwa penggunaan topologi kartesian dapat mendukung algoritma perkalian matriks bujursangkar Cannon dalam meningkatkan efisiensi komunikasi dan memperoleh *speed-up* .

### 3.2.2 Dynamic Process Generation

Deskrpisi program:

- Terdapat 2 buah suprogram yaitu *manager* dan *worker* yang bertugas melakukan perkalian matriks-vektor dengan (*row-wise decomposition*). Sumber kode:

  1. `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem2/manager.c`
  2. `https://github.com/yohanesgultom/parallel-programming-assignment/blob/master/problem2/worker.c`

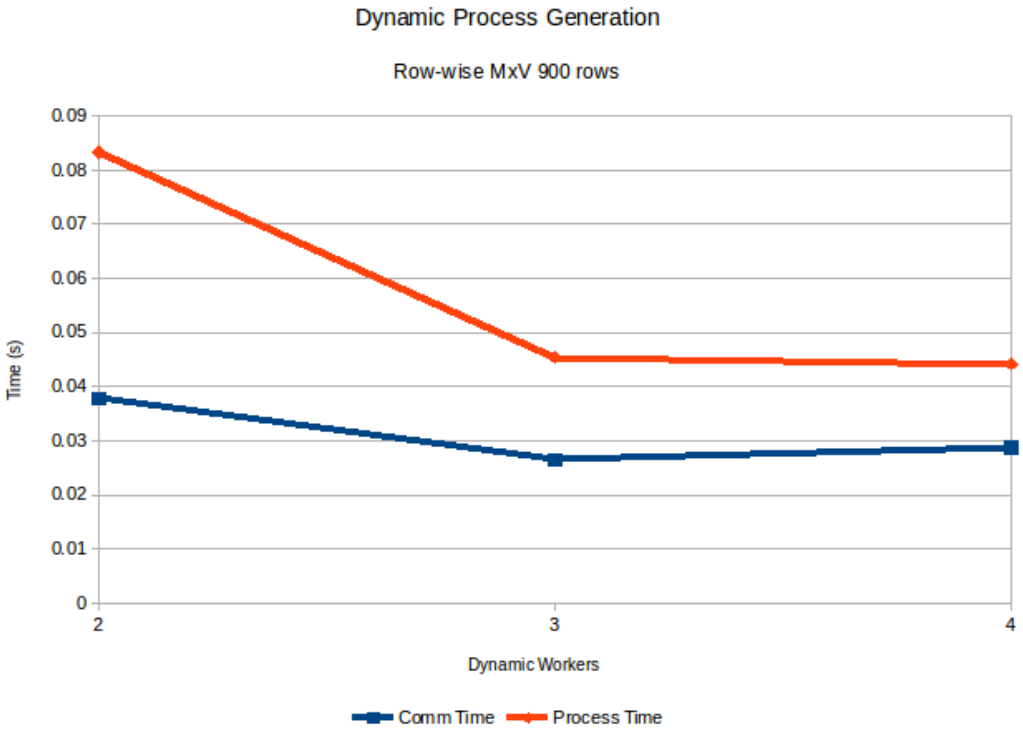
- Saat dijalankan, *manager* membuat *worker* sebanyak argumen yang diberikan dengan `MPI_Comm_spawn`. *Worker* berkomunikasi dengan manager dengan domain komunikasi yang didapat dari `MPI_Comm_get_parent`. *Manager* mengirimkan baris dan vektor kepada worker dengan `MPI_Send` dan `MPI_Bcast` untuk dikalikan secara *row-wise*

```
yohanes@Asus-K401LB:~/Workspace/parallel-programming-assignment/problem2$ mpirun manager 2 4
worker 0 is up!
worker 1 is up!
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000

2.000000
2.000000
2.000000
2.000000

8.000000
8.000000
8.000000
8.000000
2       1.030830
yohanes@Asus-K401LB:~/Workspace/parallel-programming-assignment/problem2$
```

**Gambar 3.4:** Contoh eksekusi program dynamic process generation




**Gambar 3.5:** Eksperimen waktu dynamic process generation di PC multicore

Pada gambar 3.4, kami berhasil melakukan perkalian matriks-vektor dengan *row-wise decomposition* dengan pembuatan proses dinamis pada PC multicore.

Sedangkan seperti yang ditunjukkan pada grafik gambar 3.5, perkalian matriks-vektor ini juga mengalami *speed-up* dari jumlah prosesor 2-4.

Selain itu, kami menemukan keterbatasan dari pembuatan proses dinamis MPI:

- Tidak bisa dilakukan dengan *job batch/queue* (seperti di *cluster* UCSD)
  - Jika prosesor pada *job* dialokasikan sebanyak jumlah *worker* maka *manager* akan dibuat sebanyak jumlah *worker*
  - Jika prosesor pada *job* hanya dialokasikan 1, *worker* gagal mendapat alokasi
- Tidak bisa dilakukan pada konfigurasi cluster tertentu (*cluster* Fasilkom UI)
- *Manager* bisa melakukan `MPI_Bcast` tetapi tidak bisa melakukan `MPI_Send` (sepertinya terkait dengan masalah *ranking* pada *worker process group*)

## 3.3 Kesimpulan

Kesimpulan yang kami peroleh dari eksperimen ini adalah:

- Fitur *Process Topologies* mempermudah implementasi algoritma yang membutuhkan topologi spesifik seperti algoritma Cannon
- *Dynamic Process Generation* berguna ketika sifat *job* dinamis (ditentukan pada saat *runtime*) dan/atau dibutuhkan *failover* (penanganan proses yang gagal)
- *Dynamic Process Generation* memiliki beberapa keterbatasan terkait spesifikasi *cluster* yang digunakan. Perlu dipastikan kecocokan dengan *cluster* yang akan digunakan pada saat mempertimbangkan penggunaan fitur MPI ini

# BAB 4
# CONJUGATE GRADIENT

## 4.1 Pendahuluan

Conjugate gradient method digunakan untuk menyelesaikan persamaan linear $Ax = b$ di mana matriks koefisiennya bersifat simetris dan definit positif. Matriks $A$ $n$ x $n$ dikatakan simetris jika $a_{ij} = a_{ji}$ untuk $i, j = 1, ..., n$. Matriks $A$ dikatakan definit positif jika untuk setiap vektor $x$ bukan nol, perkalian skalar $x \cdot Ax$ menghasilkan nilai lebih besar dari nol. Algoritma conjugate gradient method ditunjukkan pada Gambar 4.1. $r_k$ merupakan sisa atau selisih antara $b$ dengan $Ax_k$, sedangkan $p_k$ merupakan *search direction*.

$k = 0$; $x_0 = 0$; $r_0 = b$
**while** $(\|r_k\|^2 > \text{tolerance})$ **and** $(k < \text{max\_iter})$
    $k++$
    **if** $k = 1$
        $p_1 = r_0$
    **else**
        $\beta_k = \frac{r_{k-1} \cdot r_{k-1}}{r_{k-2} \cdot r_{k-2}}$  // minimize $\|p_k - r_{k-1}\|$
        $p_k = r_{k-1} + \beta_k p_{k-1}$
    **endif**
    $s_k = Ap_k$
    $\alpha_k = \frac{r_{k-1} \cdot r_{k-1}}{p_k \cdot s_k}$  // minimize $q(x_{k-1} + \alpha p_k)$
    $x_k = x_{k-1} + \alpha_k p_k$
    $r_k = r_{k-1} - \alpha_k s_k$
**endwhile**
$x = x_k$

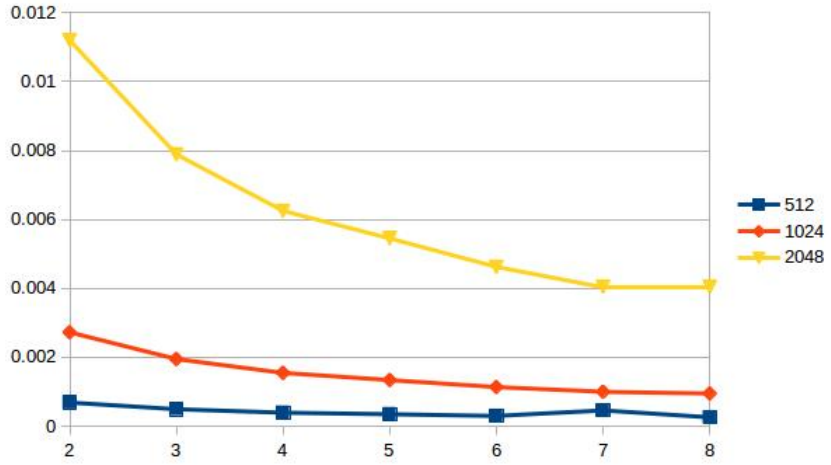
**Gambar 4.1:** Algoritma Conjugate Gradient Method.

Pada implementasi paralel CGM, matriks $A$ akan didistribusikan menggunakan fungsi `MPI_scatter` kepada setiap proses dan masing-masing proses mendapat sebanyak $n$ x $n$ / $p$ data. Algoritma tersebut dijalankan di setiap proses untuk setiap bagian distribusi matriks $A$.

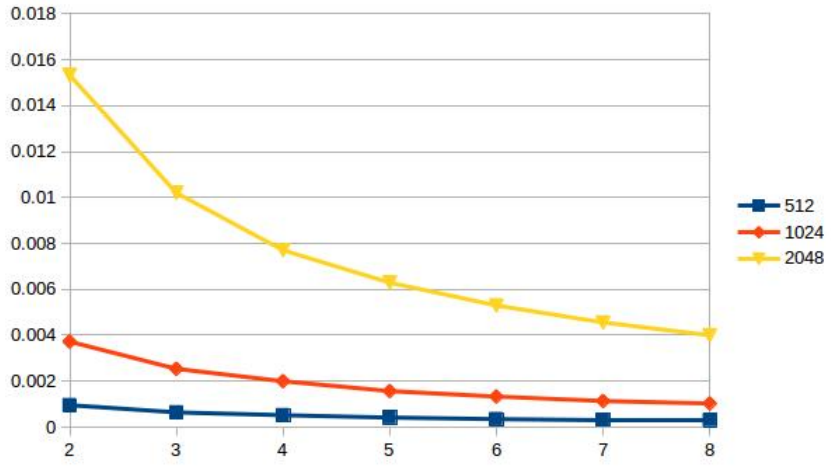
## 4.2 Eksperimen

Implementasi algoritma paralel CGM menggunakan kode sumber yang dibangun oleh Joseph Tanigawa yang dipublikasikan melalui Github. Eksperimen dilakukan

di lingkungan cluster Fasilkom dan nbcr-233.ucsd.edu. Percobaan dilakukan dengan variasi jumlah prosesor dan besar data. Banyaknya prosesor yang digunakan mulai dari 2 sampai dengan 8 sedangkan besar data yang digunakan adalah 512, 1024, dan 2048. Gambar 4.2 dan Gambar 4.3 berturut-turut menunjukkan hasil unjuk kerja implementasi algoritma paralel CGM di cluster Fasilkom dan nbcr-233.ucsd.edu. Pada grafik tersebut menunjukkan *execution time* per iterasi karena setiap eksperimen menghasilkan jumlah iterasi yang berbeda-beda. Pada eksperimen tersebut kami atur jumlah maksimum iterasi yang diijinkan adalah 1000 dengan toleransi ($r$) sebesar 1e-06.




**Gambar 4.2:** Hasil eksperimen paralel CG pada cluster Fasilkom.




**Gambar 4.3:** Hasil eksperimen paralel CG pada cluster nbcr-233.ucsd.edu.

Dari grafik tersebut menunjukkan bahwa paralelisasi algoritma CGM menghasilkan *speed up*. Meski terdapat penyimpangan saat melakukan eksperimen di cluster Fasilkom namun secara keseluruhan terdapat peningkatan performa.

# BAB 5
# MOLECULAR DYNAMICS: AMBER

## 5.1 Pendahuluan

AMBER (*Assisted Model Building with Energy Refinement*) adalah paket program untuk menjalankan simulasi dinamika molekular (*molecular dynamics*) yang dikembangkan oleh *University of California*, San Fransico. Alamat *website* resmi dari AMBER adalah http://ambermd.org.

AMBER digunakan untuk berbagai eksperimen dinamika molekular seperti simulasi pergerakan fisik dari atom dan molekular, pemodelan protein dan eksperimen yang terkait dengan perancangan obat (*drug discovery*).

AMBER didistribusikan dalam dua bagian:

1. AMBER (berbayar, versi terakhir 14)
2. AMBERTool (*opensource* GPL, versi terakhir 15)

AMBER memiliki dua mode instalasi, yaitu berbasis CPU (dengan OpenMPI) dan berbasis GPU (dengan CUDA). Petunjuk instalasi dapat dilihat di http://jswails.wikidot.com/installing-amber14-and-ambertools14.

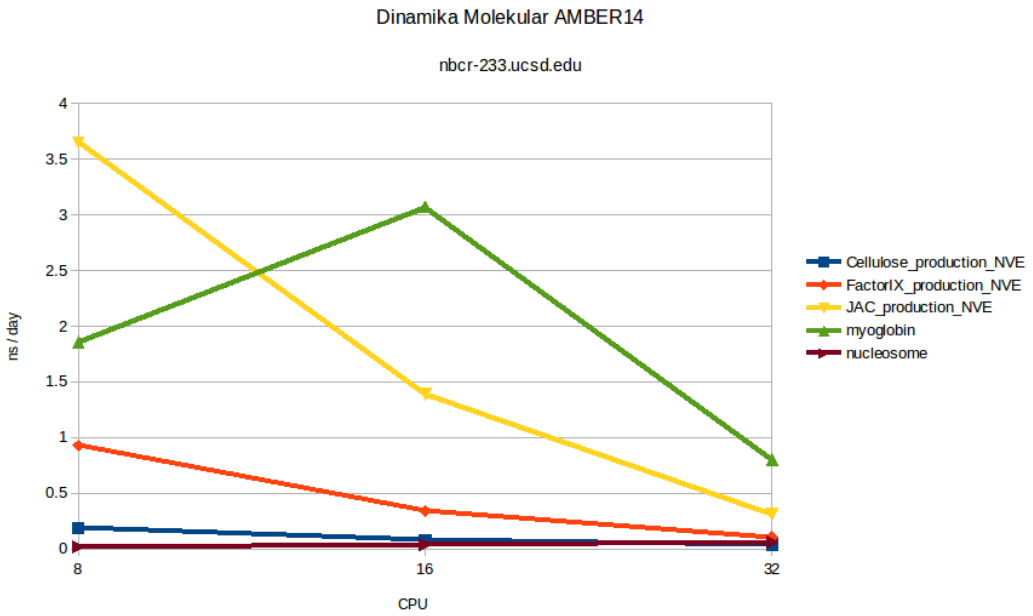
## 5.2 Eksperimen

Percobaan dilakukan dengan menjalankan 6 buah eksperimen yang disediakan pada AMBER GPU *Benchmark Suite* yang diperoleh dari *website* resmi AMBER http://ambermd.org/Amber14_Benchmark_Suite.tar.bz2. Enam buah eksperimen yang kami jalankan adalah:

1. `TRPCAGE Production` (304 atoms, 250.000 nsteps)
2. `Myoglobin Production` (2,492 atoms, 25.000 nsteps)
3. `JAC Production NVE` (23,558 atoms, 25.000 nsteps)
4. `Nucleosome Production` (25,095 atoms, 600 nsteps)
5. `Factor IX Production NVE` (90,906 atoms, 10.000 nsteps)

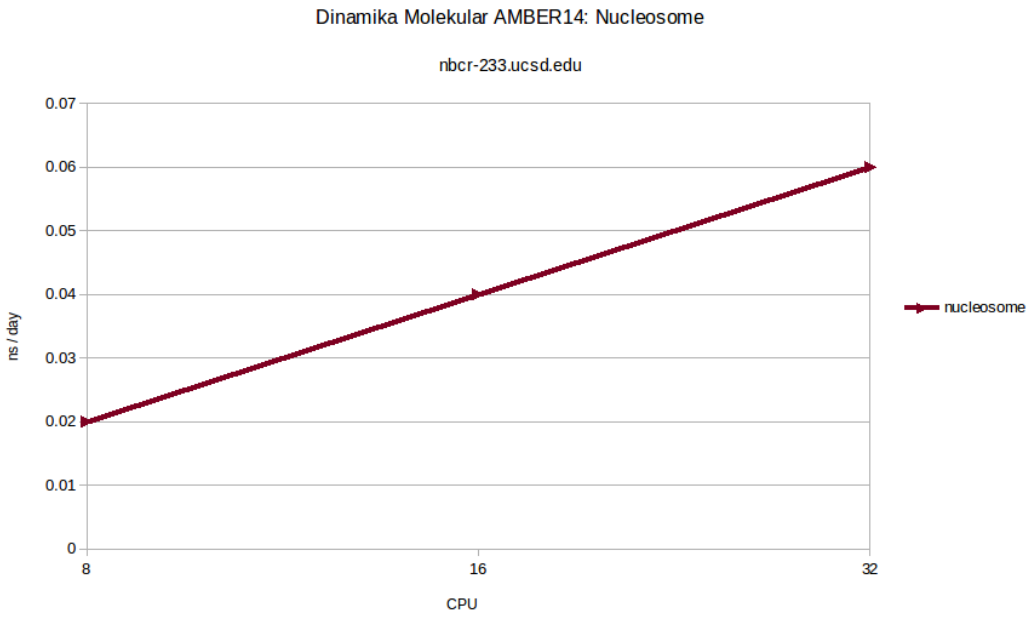
6. `Cellulose Production NVE` (408,609 atoms, 5.000 nsteps)

Pada eksperimen ini kami mengamati hubungan antara jumlah atom dan nsteps dengan kecepatan proses (`ns/day`). Jika kecepatan proses meningkat seiring bertambahnya prosesor (CPU) yang digunakan berarti terjadi *speed-up* . Sebaliknya, jika tidak terjadi pertambahan maka tidak terjadi *speed-up* .




**Gambar 5.1:** Eksperimen AMBER di cluster UCSD

Hasil yang kami peroleh ditujukkan pada grafik gambar 5.1, yaitu *speed-up* hanya terjadi pada `Cellulose_production_NVE` (CPU 8-16) dan `Nucleosome`. Peningkatan kecepatan proses dapat lebih jelas lagi diamati pada gambar 5.2 di mana grafik `Nucleosome` dipisahkan tersendiri.




**Gambar 5.2:** Eksperimen AMBER Nucleosome di cluster UCSD

## 5.3 Kesimpulan

Berdasarkan eksperimen ini, kami menarik beberapa kesimpulan:

- AMBER dapat berjalan secara paralel di atas MPI
- AMBER dapat melakukan prediksi kecepatan proses (dan durasi) sebelum percobaan selesai (dengan opsi `mdinfo`)
- Kecepatan proses (`ns/day`) bergantung pada jumlah atom dan nsteps
- Seperti halnya pada topik sebelumnya, *speed-up* terlihat jika proses dinamika molekuler membutuhkan banyak resource (jumlah atom besar dan nsteps kecil)

# BAB 6
# KONTRIBUSI

Kontribusi tiap anggota kelompok pada tugas ini adalah sebagai berikut:

**Otniel Yosi Viktorisa**:

1. Topik 1: eksperiman MPI dengan algoritma DNS (perkalian matriks bujursangkar)
2. Topik 3: eksperimen *Conjugate Gradient Method*

**Yohanes Gultom**:

1. Instalasi dan konfigurasi *cluster* Rocks dengan Virtualbox pada laptop
2. Topik 1: eksperiman MPI dengan algoritma *row-wise*, *column-wise*, *checkerboard decomposition* (perkalian matriks-vektor), *row-wise decomposition*, Fox dan Cannon (perkalian matriks bujursangkar)
3. Topik 2: eksperimen pembuatan topologi kartesian dengan MPI, algoritma Cannon dengan topologi kartesian, perkalian matriks dengan proses dinamis
4. Topik 4: eksperimen AMBER dengan enam jenis *benchmarking set* dari *website* resmi AMBER