BAB II
LANDASAN TEORI

Pada dasarnya komputasi paralel digunakan untuk menyelesaikan suatu permasalahan besar, dengan memecah-mecah permasalahan tersebut menjadi bagian-bagian dari permasalahan yang lebih kecil (sub-masalah). Kemudian sub-masalah tersebut di selesaikan oleh kumpulan-kumpulan dari prosesor (*multi-processors*) yang nantinya terlibat dalam pengeksekusian masalah tersebut. Dimana setiap bagian dari sub-masalah di selesaikan oleh satu prosesor (*single-processor*). Sehingga kita dapat mengambil kesimpulan jika sebuah masalah yang diselesaikan oleh satu prosesor membutuhkan berapa banyak sub-masalah dan berapa lama waktu yang dibutuhkan oleh prosesor tersebut. Kemudian dilakukan perbandingan dengan masalah yang sama, jika masalah tersebut diselesaikan oleh banyak prosesor.

Tujuan utama komputasi paralel adalah untuk mempersingkat waktu eksekusi program yang menggunakan komputasi serial. Beberapa alasan lain yang menjadikan suatu program menggunakan komputasi paralel antara lain:

1. Untuk komputasi yang sangat kompleks, terkadang sumber daya (*resource*) yang ada sekarang belum cukup mampu untuk mendukung penyelesaian terhadap permasalahan secara cepat.
2. Adanya keterbatasan memori pada mesin untuk komputasi serial.
3. Adanya sumber daya non-lokal yang dapat digunakan melalui jaringan lokal atau *internet*.
4. Penghematan biaya pengadaan perangkat keras, dengan menggunakan beberapa mesin yang murah sebagai alternatif penggunaan satu mesin yang bagus tapi mahal, walaupun menggunakan P-Processor (*Multicore*).

Penggunaan komputasi paralel sebagai solusi untuk mempersingkat waktu yang dibutuhkan, namun untuk eksekusi program mempunyai beberapa hambatan. Hambatan-hambatan tersebut antara lain adalah:

1. *Hukum Amdahl*, yaitu percepatan waktu eksekusi program dengan menggunakan komputasi paralel tidak akan pernah mencapai kesempurnaan karena selalu ada bagian program yang harus dieksekusi secara serial.

2. Hambatan yang diakibatkan karena beban jaringan, dalam eksekusi program secara paralel, prosesor yang berada di mesin yang berbeda memerlukan pertukaran data melalui jaringan. Untuk program yang dibagi menjadi *task-task* membutuhkan sinkronisasi, *network latency* (keterlambatan jaringan) menjadi masalah utama. Permasalahan ini muncul karena ketika suatu *task* membutuhkan data dari *task* yang lain, bagian ini dikirimkan melalui jaringan dimana kecepatan transfer data kurang dari kecepatan prosesor yang mengeksekusi instruksi *task* tersebut. Hal ini menyebabkan *task* tersebut harus menunggu sampai data tiba terlebih dahulu, sebelum mengeksekusi instruksi selanjutnya.
3. Hambatan yang terkait dengan beban waktu untuk inisialisasi *task*, terminasi *task*, dan sinkronisasi.

2.1 Arsitektur Komputer Paralel

Dalam sebuah artikel yang direferensikan oleh *Flynn*[4], Dalam mendesain sebuah komputer di karakteristikkan oleh perjalanan (alur) dari instruksi-instruksi yang akan diselesaikan oleh suatu arsitektur komputer. Taksonomi ini diklasifikasikan yang disesuaikan melalui perjalanan dari gabungan instruksi dan data. Taksonomi ini menghasilkan empat kemungkinan kombinasi dari pengoperasian instruksi. Yang terlihat pada tabel dibawah ini:

Tabel 2.1 Arsitektur Flyns Taksonomi

SISD (*Single Instruction, Single Data*)	SIMD (*Single Instruction, Multiple Data*)
MISD (*Multiple Instruction, Single Data*)	SIMD (*Multiple Instruction, Multiple Data*)

Keterangan:

1. **SISD (*Single Instruction, Single Data*)**, Pada arsitektur ini adalah arsitektur yang mewakili komputer serial, dimana hanya ada satu prosesor dan satu aliran masukan data (memori), sehingga hanya ada satu *task* yang dapat dieksekusi pada suatu waktu. Arsitektur *Von-Neumann* termasuk dalam jenis ini.
2. **SIMD (*Single Instruction, Multiple Data*)**. Pada arsitektur ini, eksekusi sebuah instruksi akan dilakukan bersamaan oleh beberapa prosesor, dimana sebuah prosesor dapat menggunakan data yang berbeda dengan prsesor lain.

3. **MISD (*Multiple Instruction, Single Data*)**, Pada arsitektur ini, berbagai instruksi akan di eksekusi secara bersamaan oleh beberapa prosesor dengan menggunakan data yang sama.
4. **MIMD (*Multiple Instruction, Multiple Data*)**, Pada arsitektur ini, berbagai instruksi dapat dieksekusi oleh beberapa prosesor dimana masing-masing prosesor dapat menggunakan data yang berbeda.

2.1.1 Paradigma Komputasi Paralel

Secara umum, paradigma komputasi paralel dapat dibagi menjadi dua jenis, yaitu *paralelisme implisit* dan *paralelisme eksplisit*.

Paralelisme implisit merupakan suatu pendekatan dimana penulis program (*programmer*) tidak perlu memperhatikan masalah pembagian *task* ke beberapa prosesor dan memori beserta sinkronisasinya. Pembagian *task* dan sinkronisasi ditangani sepenuhnya oleh sistem dibawah program (sistem operasi atau mesin virtual) atau *compiler*.

Berbeda dengan paradigma *paralelisme eksplisit* harus memperhatikan dan menangani masalah pembagian *task* dan komunikasinya. Komunikasi antar dua atau lebih *task* pada umumnya dilakukan dengan menggunakan s*hared-memory* atau *message-passing*.

1. ***Shared memory***

   Pada model komunikasi ini, *task-task* menggunakan memori (*address space*) yang sama, dimana masing-masing *task* dapat menulis atau membaca secara *asinkron*.

2. ***Message Passing***

   Pada model komunikasi ini, setiap *task* mempunyai memori (*address space*) lokal masing-masing, dan paradigma yang digunakan adalah *parallelisme eksplisit*. Contoh sumber daya komputasi yang cocok menggunakan pemodelan ini adalah beberapa *workstation* yang dijadikan *cluster*.

2.1.2 Model Pemrograman

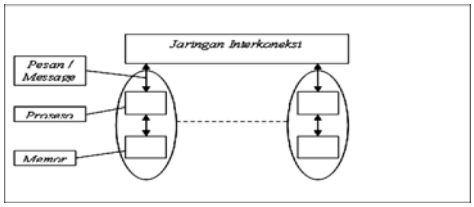
Untuk mempermudah pengembangan program di lingkungan komputasi paralel, dikembangkan model pemrograman yang menjadi cara untuk menggambarkan struktur algoritma paralel, sesuai dengan pilihan metode pemetaan proses dan dekomposisi data.

Pada tugas akhir ini model pemrograman yang digunakan yaitu model Data-Paralel . *Task* dipetakan secara statik pada masing-masing proses dan masing-masing proses tersebut melakukan operasi yang sama terhadap data yang berbeda secara bersama-sama (*concurent*). Karena setiap *task* melakukan operasi yang sama, maka metode dekomposisi yang digunakan adalah dekomposisi data (*Data Decomposition*).

2.2 Pertukaran Pesan (*Message Passing*)

Message-passing dilakukan oleh prosesor-prosesor yang terlibat dalam komputasi paralel untuk melakukan pertukaran data dan sinkronisasi antar prosesor.

Pertukaran data pada *message-passing* dibuat dengan menghubungkan sekumpulan komputer melalui sebuah interkoneksi jaringan. Setiap komputer memiliki sebuah prosesor dan memori lokal, seperti ditunjukkan pada gambar 2.1, dan komunikasi antar komputer dilakukan melalui interkoneksi jaringan. Pesan tersebut berisikan data yang diperlukan dalam komputasi.



Gambar 2.1 Model Komputer *Message-Passing*

2.2.1 MPI (*Message Passing Interface*)

Message-Passing Interface atau MPI adalah sebuah standar untuk *interface* pada pemrograman paralel. Ada lima karakter dasar pada model pemrograman *message-passing*, yang dapat diimplementasikan pada MPI[1], sebagai berikut:

1. Sebuah komputasi terdiri dari sekumpulan proses-proses besar (biasanya dalam jumlah yang ditentukan), dimana masing-masing memiliki identifikasi yang unik (bilangan bulat 0 sampai P-1 (Jumlah Prosesor-1)).
2. Proses-proses saling berinteraksi dengan saling bertukar pesan-pesan tertentu, dengan melibatkan diri pada operasi komunikasi kolektif ataupun menunggu pesan-pesan yang dikirimkan.
3. Modularitas didukung melalui kelompok jaringan *(communicator)*, yang membuat sub-program dapat mengkapsulasi operasi-operasi komunikasi dan dapat di kombinasikan dengan komposisi sekuensial maupun paralel.

4. Algoritma yang membuat pekerjaan-pekerjaan dinamis atau beberapa pekerjaan sekaligus pada sebuah prosesor, memerlukan perbaikan sebelum dapat diimplementasikan dengan MPI.
5. Sifat deterministik (beban yang sama) tidak terjamin, tetapi dapat diperoleh dengan pemrograman yang cermat.

2.2.2 Komunikasi Pada *Message-Passing*

Pada sebuah proses komunikasi, ada dua aktifitas yang dilakukan, yaitu pengiriman dan penerimaan, masing-masing proses memiliki identifikasi yang unik. Identifikasi ini digunakan seperti penggunaan alamat dalam pengiriman surat. Sebuah pesan yang sederhana pada model pemrograman *message-passing* memiliki alamat penerima (identifikasi proses tujuan) serta alamat pengirim (identifikasi proses asal).

Komunikasi sederhana dalam MPI yang hanya melibatkan dua buah proses. Komunikasi yang melibatkan lebih dari dua proses disebut komunikasi kolektif. Komunikasi kolektif ini melibatkan semua proses yang berada pada suatu kelompok jaringan *(communicator)* yang sama.

Operasi pada komunikasi kolektif disertai dengan proses sinkronisasi. Ini artinya setiap kali proses-proses melakukan operasi komunikasi kolektif, proses-proses tersebut tidak dapat melakukan pekerjaan selanjutnya sebelum semua proses selesai melakukan operasi tersebut. Jika sebuah proses belum selesai maka proses yang lain akan menunggu proses tersebut hingga selesai.

2.2.3 Fungsi-Fungsi Dasar MPI

Sebelum dapat menggunakan fungsi-fungsi MPI, MPI harus dinisialisi dengan memanggil fungsi:

MPI_Init (int *argc_ptr, char **argv_ptr[])

Keterangan:

1. MPI_Init merupakan variabel untuk melakukan inisialisasi awal dari proses MPI.
2. Variabel argc_ptr dan argv_ptr adalah alamat dari parameter yang diberikan untuk fungsi *main* (utama), yang merupakan argumen dari program.

Setelah program selesai menggunakan MPI, harus dilakukan finalisasi. Finalisasi dilakukan untuk menyelesaikan pekerjaan-pekerjaan MPI yang belum selesai seperti

melakukan dealokasi memori yang digunakan oleh MPI. Finalisasi pada MPI dilakukan dengan memanggil fungsi:

MPI_Finalize();

Setelah melakukan inisialisasi, hal yang perlu kita lakukan selanjutnya adalah mendapatkan informasi yang kita perlukan untuk keperluan eksekusi paralel dari program yang dirancang. Diantaranya adalah identifikasi unik dari proses, yang disebut *rank*, jumlah proses yang ada dan lain-lain. *Rank* dari suatu proses dan jumlah proses didapatkan dengan memanggil fungsi:

MPI_Comm_rank(MPI_Comm komunikasi, int *Rank);

MPI_Comm_size(MPI_Comm komunikasi, int *JumlahProsesor)

Keterangan:

1. MPI_Comm_rank mendifinisikan sebuah fungsi yang ditujukan untuk menentukan identifikasi unik dari proses-proses yang ada.
2. MPI_Comm_size mendifinisikan sebuah fungsi yang ditujukan untuk menentukan berapa banyak jumlah pemroses yang terlibat dalam kelompok jaringan *(communicator)* ini.
3. MPI_Comm mendifinisikan sebuah kelompok jaringan *(communicator)* yang mencakup seluruh proses yang ada. Umumnya menggunakan MPI_COMM_WORLD jika tidak berniat membuat kelompok jaringan *(communicator)* sendiri.
4. int *Rank dan *JumlahProsesor merupakan varibel penentu dalam mengidentifikasikan pemroses-pemroses yang terlibat dan penentu berapa banyak jumlah pemroses yang akan digunakan.

Setiap proses dalam program MPI perlu berinteraksi dengan saling bertukar pesan. Pertukaran pesan paling sederhana pada program MPI dilakukan antara dua proses secara langsung. Salah satu proses mengirimkan pesan, sementara proses yang lainnya akan menerimanya. Pengriman pesan pada MPI dilakukan dengan menggunakan fungsi:

MPI_Send(void *pesan, int jumlah_pesan, MPI_Datatype tipe_data, int tujuan, int tag, MPI_Comm komunikasi);

MPI_Recv(void *pesan, int jumlah_pesan, MPI_Datatype tipe_data, int asal, int tag, MPI_Comm komunikasi, MPI_Status *status);

Keterangan:

1. MPI_Send dan MPI_Recv mendefinisikan fungsi untuk melakukan proses pertukaran pesan kirim dan terima.
2. Variabel pesan merupakan *pointer buffer* (data) yang akan dikirimkan atau diterima.
3. Variabel tipe_data merupakan struktur data dari variabel pesan, yang bisa bertipe MPI_CHAR, MPI_INIT, MPI_FLOAT, MPI_DOUBEL, dan lain-lain.
4. Varibel jumlah_pesan menentukan berapa besar data dari pesan yang akan dikirimkan atau data yang akan diterima.
5. Variabel asal dan tujuan merupakan alamat kemana pesan tersebut akan dikirim atau darimana asal pesan tersebut datang.
6. Variabel tag sendiri ditujukan untuk membedakan antara pesan-pesan mana yang akan dikrimkan dan mana yang akan diterima.
7. Variabel status digunakan apabila terjadi kesalahan pada saat penerimaan pesan.

2.2.4 Sinkronisasi Proses Komunikasi Kolektif

Kadang kala kita perlu melakukan sinkronisasi pada bagian-bagian tertentu dari proses-proses paralel yang kita gunakan. Sinkronisasi antar proses-proses dalam suatu kelompok jaringan *(communicator)* tanpa melibatkan operasi apapun pada MPI dilakukan dengan menggunakan fungsi:

MPI_Barrier(MPI_Comm komunikasi)

Fungsi ini menyebutkan setiap proses pada kelompok jaringan *(communicator)*, akan berhenti hingga semua proses telah memaggil fungsi tersebut. *Barrier* biasanya digunakan untuk memisahkan dua fase komputasi dan komunikasi, supaya pesan-pesan antara kedua fase tersebut tidak tercampur.

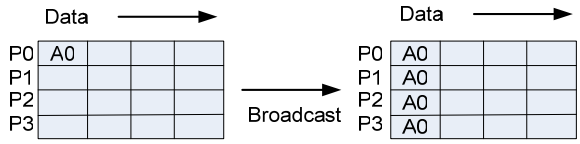
2.2.5 Broadcast

Broadcast adalah sebuah operasi komunikasi kolektif dimana sebuah proses dalam suatu kelompok jaringan *(communicator)* mengirimkan data yang sama kepada semua proses yang lain pada kelompok jaringan *(communicator)* tersebut, dapat dilihat pada gambar 2.2. Fungsi MPI yang digunakan untuk melakukan *broadcast* adalah:

MPI_Bcast (void *pesan, int jumlah_pesan, MPI_Datatype tipe_data, int root, MPI_Comm komunikasi)

Keterangan:

1. MPI_Bcast mendifinisikan fungsi untuk melakukan *broadcast* data sebesar jumlah_pesan pada semua pemroses pada kelompok jaringan *(communicator)* tersebut.
2. Variabel int root merupakan penentu siapa yang akan melakukan operasi *broacast* itu sendiri.



Gambar 2.2 Operasi *Broadcast*

2.2.6 *Scatter* dan *Gather*

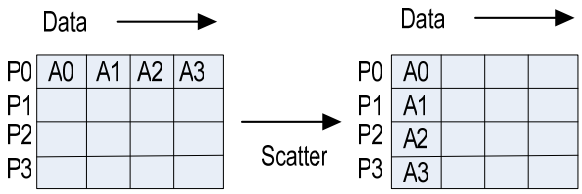
Salah satu pendekatan pada pemrograman paralel adalah paralel data. Pada pendekatan ini, proses-proses yang ada membagi data dan melakukan pekerjaan tertentu untuk datanya masing-masing sebagai bagian dari pemetaan data.

MPI telah menyediakan operasi yang efisien untuk melakukan pemetaan data ini, sehingga kita tidak perlu melakukannya sendiri. Fungsi yang dapat digunakan untuk mendistribusikan data secara merata kepada proses-proses yang ada adalah:

MPI_Scatter (void *buffer_kirim, int jumlah_data_kirim, MPI_Datatype tipedata_pesankirim, void *buffer_terima, int jumlahdata_terima, MPI_Datatype tipedata_pesanterima, int root, MPI_Comm komunikasi)

Keterangan:

1. MPI_Scatter adalah mendefinisikan sebuah fungsi untuk memecah data menjadi sejumlah segmen. masing-masing pemroses mendapat besar data yang sama. Dapat dilihat pada gambar 2.3.
2. Variabel buffer_kirim merupakan alamat dari data yang akan dikirimkan pada proses yang mengirim. Variabel buffer_terima merupakan alamat tempat data tersebut akan disimpan pada proses-proses yang menerima.
3. Variabel jumlah_data_kirim dan jumlah_data_terima, maupun tipedata_kirim biasanya bernilai sama.

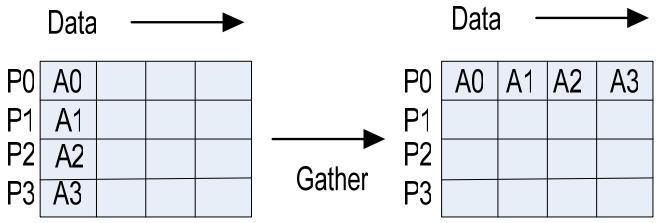


Gambar 2.3 Operasit *Scatter*

Data biasanya akan diolah setelah dibagikan ke proses-proses yang ada. Selanjutnya data yang telah diolah tersebut dikumpulkan kembali. MPI menyediakan operasi *gather* yang merupakan kebalikan dari operasi *scatter*, untuk melakukan hal tersebut. Fungsi yang digunakan yaitu:

int MPI_Gather (void *alamat_kirim, int jumlahdata_kirim, MPI_Datatype tipedata_kirim, void *alamat_terima, int jumladata_terima, MPI_Datatype tipedata_terima, int root, MPI_Comm komunikasi)

Parameter yang digunakan oleh fungsi ini sama dengan *MPI_Scatter*, tetapi dengan arah pengiriman yang terbalik. Pada operasi *gather*, root akan menerima data, sementara prorses yang lainnya akan mengirimkan data. Proses pengumpulan data dengan operasi *gather* dapat dilihat pada gambar 2.4



Gambar 2.4 Operasi *Gather*

2.2.7 Topologi Virtual

Sejauh ini, kita menggunakan *rank* dalam mengidentifikasi sebuah proses. *Rank* dapat dikatakan sebagai alamat dari suatu proses dalam sebuah kelompok jaringan *(communicator)*. Skema dari pengalamatan proses-proses dalam kelompok jaringan *(communicator)* disebut topologi.

MPI sebenarnya menyediakan topologi lain untuk pengalamatan proses. Topologi tersebut adalah topologi kartesian. Topologi kartesian dapat dimanipulasi dengan fungsi-fungsi sebagai berikut:

MPI_Cart_Create(MPI_Comm komunikator_lam, int besar_dimensi, int dimensi[], int wrap_arround[], int reorder, MPI_Comm *kartesian_komunikator)

MPI_Cart_coords(MPI_Comm komunikator, int rank, int max_dimensi, int kordinat[])

MPI_Cart_rank(MPI_Comm komunikator, int kordinat[], int *rank)

MPI_Cart_sub(MPI_Comm komunikator_kartesian, int kordinat[], MPI_Comm *komunikator_baru)

Keterangan:

1. MPI_Cart_Create Digunakan untuk membuat sebuah kelompok jaringan *(communicator)* baru yaitu kartesian-komunikator dari kelompok jaringan *(communicator)* sebelumnya yaitu komunikator_lam.
2. Topologi tersebut berdimensi besar_dimensi, dimana untuk masing-masing dimensi panjangnya ditentukan oleh variabel int dimensi.
3. Variabel wrap_arround digunakan untuk menentukan apakah untuk dimensi tertentu *sirkular* atau *linier*. Jika kita akan menentukan bahwa dimensi 0 adalah *sirkular* maka wrap_arround[0] bernilai 1 dan bernilai 0 jika sebaliknya.
4. Variabel reorder digunakan untuk menentukan apakan MPI boleh melakukan pengaturan ulang untuk efisiensi pada proses-proses yang ada. Variabel ini bernilai 1 jika MPI boleh melakukan pengaturan dan bernilai 0 jika sebaliknya.
5. MPI_Cart_coords digunakan untuk mendapatkan koordinat kartesian dari proses dengan *rank* tertentu. Koordinat dengan dimensi yang kita inginkan akan disimpan dalam kordinat.
6. MPI_Cart_rank digunakan mendapatkan *rank* dari suatu proses dengan koordinat tertentu.
7. MPI_Cart_sub digunakan untuk melakukan partisi dari topologi kartersian dengan dimensi tertentu menjadi topologi dengan dimensi yang lebih rendah. Misalnya mengambil dimensi 0 saja dari topologi kartesian berdimensi dua, maka variabel titik_kordinat[0] harus bernilai 1 dan titik_kordinat[1] harus bernilai 0.

2.3 Cluster

Cluster adalah kumpulan komputer yang terhubung melalui interkoneksi jaringan. Setiap node pada *cluster* bisa sebagai sebuah *server*, komputer-personal atau komputer yang memiliki dua atau lebih prosesor yang terintegrasi sebagai bagian dari sistem multiprosesor. Setiap node adalah sebuah komputer yang memiliki sistem *input/output (I/O)* dan sistem operasi sendiri.

Ketika semua node pada sebuah *cluster* memiliki arsitektur yang sama dan beroperasi pada sistem operasi yang sama maka *cluster* ini disebut sebagai *cluster* homogen (*homogeneous cluster*), Begitu juga sebaliknya jika *cluster* dengan sistem operasi maupun arsitektur yang berbeda disebut *cluster* heterogen (*heterogeneous cluster*).

2.3.1 SSI (*Single System Image*)

SSI (*Single System Image*) adalah sistem *cluster* yang menggabungkan beberapa node (PC dan sebagainya) menjadi satu kesatuan sehingga dari pandangan luar (*end-user*) terlihat sebagai suatu kesatuan komputasi. "Satu-Kesatuan" yang dimaksud disini adalah satuan unit CPU (*Control Processing Unit*), *Memory* dan Media Penyimpanan (*Storage*). Ada dua cara untuk membentuk satu kesatuan ini:

1. *Middleware*

   Secara umum ini meliputi semua komponen (*library*) yang menjadi perantara antara program yang berjalan di *user-mode* (mode pengguna) dengan *kernel*. Contohnya sistem Batch Scheduler seperti Condor atau Maui Scheduler atau Enfuzion. Library PVM (*Parallel Virtual Machine*) dan MPI (*Message Passing Interface*) seperti LAM-MPI, MPICH.
2. Modifikasi *Kernel* (*Kernel-Extension*)

   Secara umum, meliputi semua tambahan kernel sistem operasi yang membuat sistem operasi tersebut menjadi sistem *cluster*, termasuk yang bertipe SSI, IBM Sysplex, Compaq Tru Cluster, dan Open-Mosix.

2.3.2 Interkoneksi Jaringan

Seperti yang telah dijelaskan pada sub-bab sebelumnya, komputasi *cluster* dengan arsitektur *message-passing*, melakukan pertukaran data (komunikasi) antar prosesor melalui jaringan baik itu jaringan lokal maupun internet. Dalam tugas akhir ini tidak mebahas spesifikasi mengenai interkoneksi jaringan secara keseluruhan.

Adapun topologi jaringan yang biasa menjadi penentu dari proses alur dari pertukaran data antar komputer diatur melalui program yang dirancang, pada tugas akhir ini topologi yang digunakan adalah topologi *linier*, *ring*, dan *mesh*. Artinya metode pengiriman dan penerimaan data dibentuk melalui proses yang didukung oleh *library-MPI* yang bisa membentuk topologi sesuai keinginan perancang program.

Walaupun secara keseluruhan harusnya ada kesinambungan antara topologi fisik dengan topologi yang dirancang berdasarkan program. Sedangkan pada tugas akhir ini lebih difokuskan dengan memanfaatkan *library-library* yang didukung oleh MPI termasuk memanfaat *topology cartesian* yang disediakan oleh MPI.

2.4 Waktu Eksekusi Program Paralel

Waktu eksekusi program paralel adalah lamanya waktu sejak dimulai sampai berakhirnya ekeskusi program paralel. Secara praktis, waktu ini dimulai sejak input untuk memulai program dimasukkan oleh user sampai dengan konfirmasi bahwa program telah selesai muncul ke layar.

Dalam eksekusi serial, waktu total eksekusi program hanyalah waktu eksekusi komputasi program tersebut. Sedangkan dalam eksekusi paralel, waktu total eksekusi adalah waktu komputasi program tersebut ditambah dengan waktu untuk melakukan komunikasi antar proses yang bisa berupa pertukaran data maupun sinkronisasi,

Jika waktu keseluruhan eksekusi program paralel dilambangkan dengan $t_{paralel}$ maka persamaan untuk waktu eksekusi program paralel adalah:

$$t_{paralel} = t_{komputasi} + t_{komunikasi} \quad (1)$$

Keterangan:

$t_{kompoutasi}$ adalah waktu komputasi

$t_{komunikasi}$ adalah waktu komunikasi

2.4.1 Performansi Komputasi Teoritis

Waktu komputasi dapat diperkirakan dengan cara yang sama untuk algoritma sekuensial. Jika satu proses dieksekusi bersamaan, hanya diperlukan jumlah komputasi pada proses yang paling kompleks. Analisa terhadap waktu komputasi dilakukan dengan asumsi semua prosesor sama dan beroperasi pada kecepatan yang sama.

Waktu komputasi dalam eksekusi paralel akan di normalisasi dalam ukuran satuan eksekusi operasi aritmatika yang tergantung pada sistem yang dipakai. Satuan

tersebut menyatakan waktu yang diperlukan untuk eksekusi satu operasi aritmatika (+, -, ÷, ×) atau waktu operasi. Jika suatu komputasi memerlukan sejumlah k_1 langkah maka:

$$t_{komputasi} = k_1 \text{ x } t_{operasi} \quad (2)$$

Keterangan:

k_1 adalah berapa banyak jumlah dari operasi aritmatika dalam satu waktu.

$t_{operasi}$ (waktu operasi) adalah waktu yang dibutuhkan sebuah prosesor untuk melakukan operasi aritmatika (+, -, ÷, ×).

Dalam suatu lingkungan pemrograman paralel, komputer yang dipergunakan dapat memiliki kemampuan komputasi yang berbeda. Kemampuan komputasi tiap komputer akan dinyatakan dalam suatu satuan *Floating Point Instruction per Second (FLOPS)*.

Satuan *FLOPS* dipakai karena sebagian besar program paralel melakukan komputasi numerik dan komputasi numerik tersebut adalah perhitungan terhadap angka bertipe *floating point*. Dari *FLOPS* tiap-tiap komputer ditentukan satuan waktu untuk eksekusi *FLOPS* adalah waktu operasi bagi masing-masing komputer.

2.4.2 Performansi Komunikasi Teoritis

Dengan mengetahui *bandwith* suatu jaringan komputer dapat ditentukan besarnya waktu yang diperlukan untuk pengiriman satu satuan data. Sedangkan, secara teoritis, waktu pengiriman suatu data dengan panjang b satuan data, dapat dinyatakan dalam persamaan:

$$t_{transmisi} = t_{startup} + b \text{ x } B \quad (3)$$

Keterangan:

$t_{startup}$ adalah waktu *startup* atau seringkali disebut dengan *message latency* atau sering disingkat dengan *latency*.

Latency adalah waktu yang diperlukan untuk pengiriman pesan tanpa data atau waktu yang diperlukan untuk mempersiapkan data yang dikirimkan dan mempersiapkan data yang diterima.

B adalah istilah *Bandwith*, dengan satuan Mbps *(Megabit Perseconds)*, *Bandwith* adalah sama dengan $1/t_w$, dimana t_w merupakan waktu maksimal yang dibutuhkan sebuah perangkat jaringan (*Ethernet Card*) dalam melakukan proses pengiriman satu satuan data dalam satu waktu (bps).

Performansi perhitungan waktu dalam komunikasi MPI[3] dapat dilihat pada tabel dibawah ini:

Tabel 2.2 Persamaan Perhitungan Waktu Operasi MPI

Operasi MPI	Persamaan Perhitungan Waktu Operasi
MPI_Barrier	$t_{Barrier} = t_{startup} \text{ x } \log_2 P$
MPI_Scatter dan MPI_Gather	$t_{scatter} = (\log_2 P \text{ x } t_{startup} + \frac{N}{t_w} \text{x } (P-1))$
MPI_Sendrecv	$t_{sendrecv} = \left(t_{startup} + \frac{N}{t_w} \right)$
MPI_Broadcast	$t_{Broadcast} = \log_2 P (t_{startup} + \frac{N}{t_w})$

Keterangan:

1. Nilai $\log_2$ didapat berdasarkan metode yang digunakan LAM-MPI, untuk operasi komunikasi kolektif menggunakan metode algoritma-*tree,*topologi *mesh*, dimana $\log_2$ mencirikan berapa banyak langkah yang dibutuhkan untuk mengirimkan data dalam susunan topologi jaringan yang digunakan.
2. N adalah besar data yang akan dikirimkan (*Bytes*)
3. P adalah jumlah prosesor. Untuk semua fungsi diatas persamaan yang digunakan adalah untuk *O* (P), dimana *O* merupakan notasi dari sebuah kompleksitas.

2.4.3 Percepatan (Speedup)

Percepatan (*Speedup*) adalah perbandingan antara kecepatan eksekusi suatu program serial (1-Prosesor) dengan program paralel menggunakan jumlah prosesor (P-Prosesor).

$$Sp(P) = \frac{T_{serial}}{T_{paralel}} \quad (4)$$

Keterangan:

Sp(P) adalah peningkatan kecepatan dengan menggunakan jumlah prosesor (P-Prosesor)

T_{serial} adalah waktu eksekusi 1-Prosesor

$T_{paralel}$ adalah waktu eksekusi paralel.

Percepepatan maksimum dari komputasi paralel adalah sebesar jumlah prosesor (P), yang bisa dicapai jika komputasi dapat dibagi menjadi proses-proses dengan durasi yang sama dengan tiap proses dipetakan ke satu prosesor dan tidak ada *overhead*.

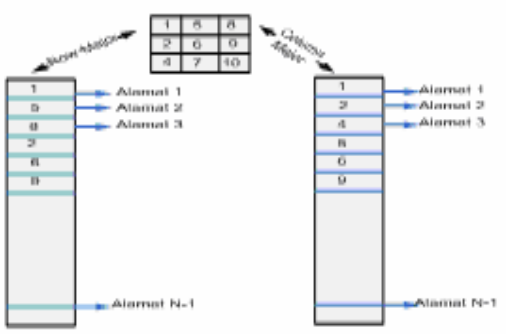
2.4.4 Overhead

Ada beberapa faktor yang menyebabkan terjadinya *overhead* pada komputasi paralel dan membatasi percepatan dari komputasi paralel sebagai berikut:

1. Periode ketika semua prosesor tidak melakukan komputasi
2. Komputasi tambahan pada komputasi paralel yang tidak ada pada komputasi serial, misalnya perhitungan nilai variabel yang diperlukan untuk membagi *task* ke komputer-komputer yang tersedia
3. Waktu komunikasi untuk pengiriman pesan.

2.4.5 Alokasi Data Pada Memori

Setiap bahasa pemrograman memiliki kemampuan sendiri-sendiri dalam melakukan alokasi (penyimpan data) pada memori, seperti bahasa-C memiliki kemampuan dalam alokasi datanya secara *Row-Major* (urutan baris), Sedangkan bahasa-Fotran melakukannya secara *Column-Major* (urutan kolom). Hal ini penting untuk diperhatikan terlebih dahulu sebelum melakukan pemrograman pada MPI, karena MPI sendiri juga memilki ketentuan dalam melakukan proses pembagian datanya, khususnya dalam hal ini proses komunikasi kolektif yang banyak digunakan pada penelitian tugas akhir ini.



Gambar 2.5 Model Alokasi Memori Multidimensi

Dari gambar 2.5 terlihat perbedaan pengalokasian data *array* 2-Dimensi (baris-kolom). Memilih kondisi dari alokasi ini sangat mempengaruhi dari kecepatan kemampuan sebuah prosesor dalam melakukan tugasnya, semakin cepat data yang

diterima oleh sebuah prosesor dari memori maka semakin cepat juga proses komputasi dilakukan oleh prosesor.

2.5 Teori Perkalian Matriks

Perkalian antara Matriks A[M][N] yang terdiri dari M baris dan N kolom dengan Matriks (B[N][L]) yang terdiri dari N baris dan L kolom akan menghasilkan Matriks C[M][L] dengan M baris dan L kolom. Setiap elemen Matriks C dihitung berdasarkan persamaan berikut:

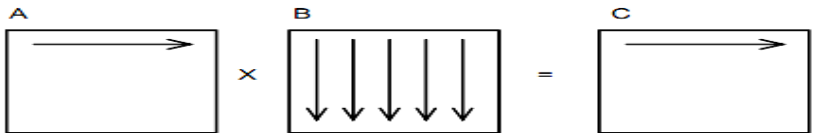
$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \text{ x } B_{kj} \tag{5}$$

Keterangan :

Untuk setiap elemen dari Matriks C adalah hasil dari perkalian bagian luar dari baris pada Matriks A dengan kolom bagian terdalam dari Matriks B, dimana nilai i lebih kecil dari atau sama dengan M dan nilai j lebih kecil dari atau sama dengan L.

Jika pada kasus tugas akhir ini menggunakan matriks persegi dengan ukuran Matriks (N x N), dimana N adalah banyaknya baris dan kolom pada setiap matriks adalah sama, maka kompleksitas matriks sekuensial sebesar $O(N^3)$.

2.5.1 Algoritma Perkalian Matriks Sekuensial



Gambar 2.6 Proses Perkalian Matriks Sekuensial

Dari gambar 2.6 dapat dijelaskan, setiap elemen Matriks C adalah hasil perkalian skalar dari baris Matriks A dan kolom-kolom Matriks B, maka diperlukan operasi komputasi sebanyak (**N x N)(2N-1)**. Dengan mensubstitusikan persamaan (2) maka untuk waktu komputasi sebuah komputer pada perkalian matriks adalah :

$$\begin{aligned} t_{komputasi} &= k_1 \text{ x } t_{operasi} \\ &= \left[(N \text{ x } N)(2N-1) \right] \text{ x } t_{operasi} \end{aligned} \tag{6}$$

2.5.2 Algoritma *Broadcast* Matriks-B

Sebelum menjelaskan lebih jauh tentang algoritma *Broadcast* Matriks-B, beberapa hal yang harus diperhatikan terlebih dahulu, nilai N harus merupakan kelipatan dari jumlah prosesor, dengan tujuan agar pembagian banyak baris atau kolom pada matriks untuk semua prosesor adalah jumlahnya sama.

Algoritma ini merupakan algoritma paling sederhana pada perkalian matriks paralel, dimana hanya Matriks A yang di *scatter* sedangkan Matriks B akan di *Broadcast* ke semua pemroses (prosesor), artinya untuk semua prosesor akan memilki data yang sama dari Matriks B tersebut. Adapun langkah-langkah pada algoritma *Broadcast* Matriks-B sebagai berikut:

1. *Scatter* Matriks A

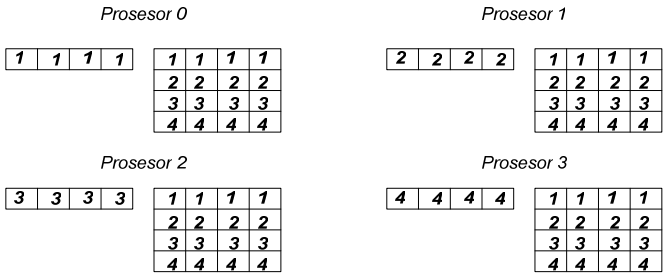
   Proses *scatter* adalah proses pembagian data yang ditentukan berdasarkan variabel HasilBagi, dimana HasilBagi adalah nilai N Mod P. Dimana N adalah banyak baris Matriks A sedangkan P adalah jumlah prosesor, sedangkan operasi Mod ditujukan agar pembagian data seimbang (sisa bagi harus 0). Jika pada contoh ini nilai N adalah empat maka variabel HasilBagi akan bernilai 1. Dapat dilihat pada gambar 2.7, proses setelah dilakukannya *scatter* pada Matriks A dan *broadcast* Matriks B.

2. Komputasi Lokal

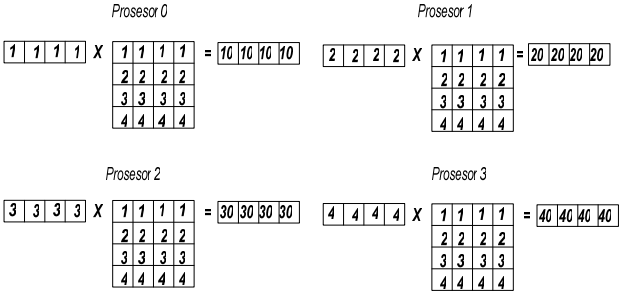
   Jika pada perkalian matriks sekuensial total komputasi yang dilakukan oleh sebuah prosesor pada adalah (N x N) x (2N-1), maka pada algoritma paralel *Broadcast* Matriks-B ini masing-masing komputer akan melakukan komputasi menjadi (N/P x N) x (2N-1). Dari algoritma ini terlihat bahwa beban komputasi akan menjadi berkurang pada baris dari Matriks A. setiap prosesor akan memiliki sub-blok baris Matriks A sebesar (N/P x N).

3. Kumpulkan Hasil dan dikirim ke Prosesor-0

   Proses pengumpulan data akan dilakukan oleh masing-masing prosesor ke (i+1), yang menyimpan masing-masing baris Matriks C ke prosesor 0, untuk lebih jelasnya dapat dilihat pada gambar 2.8.



Gambar 2.7 Proses *Scatter* Matriks A dan *Broadcast* Matriks B



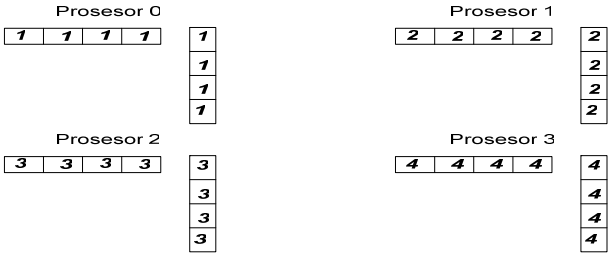
Gambar 2.8 Komputasi Lokal Masing-masing Prosesor

2.5.3 Algoritma *Row-Coloumn Striped Block*

Pada algoritma *Row-Column Block Striped Block* ini akan melakukan pembagian baris dari Matriks A dan kolom dari Matriks B, jika pada algoritma matriks sekuensial, ukuran Matriks A[N x N] dan Matriks B [N x N] akan dibagi menjadi sub-blok A [$\frac{N}{P}$ x N] dan sub-blok B[N x $\frac{N}{P}$] .Adapun langkah-langkah dari algoritma *Row-Column Striped Block* sebagai berikut:

1. Proses *Scatter* Matriks A dan Scatter Matriks B

   Pada proses *scatter* Matriks B, hampir sama dengan proses *scatter* Matriks A seperti algoritma *Broadcast* Matriks-B hal yang membedakannya adalah *scatter* pada Matriks B dilakukan pada kolom dari Matriks B seperti pada gambar 2.9.



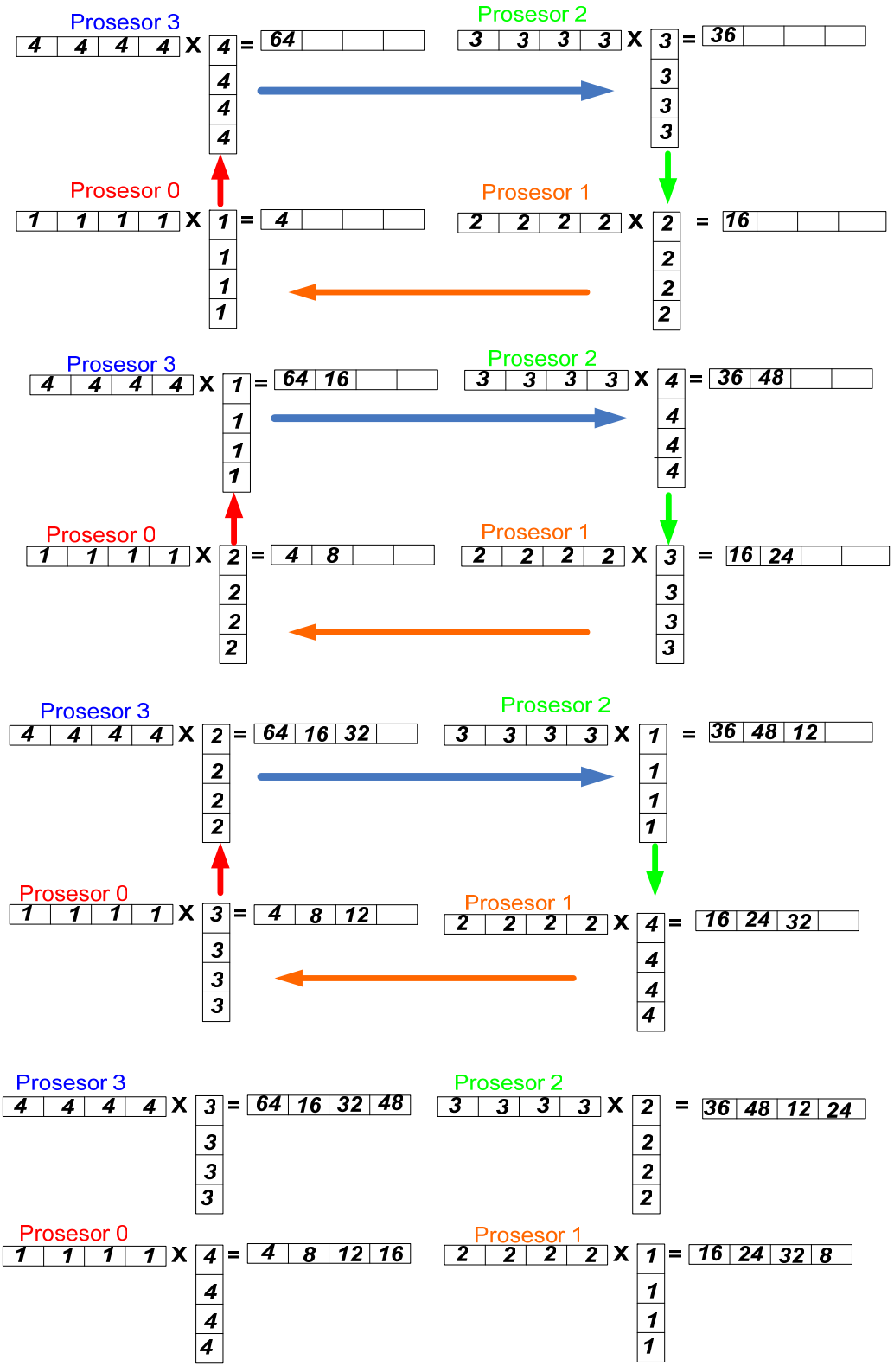
Gambar 2.9 Proses *Scatter* Baris Matriks A dan *Scatter* Kolom Matriks B

2. Komputasi Lokal dan Kirim Sub-Matriks B ke prosesor ke i+1

   Pada proses ini, komputasi dilakukan perulangan sebanyak jumlah prosesor dan melakukan pertukaran data dari setiap sub-kolom Matriks B sebanyak jumlah prosesor dikurang satu (P-1). selama perulangan jumlah prosesor P, $0 \leq loop \leq P$, setiap prosesor akan melakukan komputasi sub-baris i Matriks-A dengan sub-kolom Matriks-B, Kemudian masing-masing prosesor akan menyimpan sub-baris Matriks-C dan sub-baris Matriks A, kemudian mengirimkan setiap sub-kolom dari Matriks B ke prosesor di sebelah atas nya, dan menukar data sub-kolom dari Matriks B tersebut dengan sub-kolom dari Matriks B yang terima dari prosesor yang sebelah bawahnya. Metode ini dikenal dengan metode *successor-ring* (pertukaran data membentuk lingkaran).Untuk lebih jelasnya dapat dilihat pada gambar 2.10.

3. Kumpulkan Hasil dan Kirim ke-Prosesor 0

   Proses pengumpulan hasil komputasi dari masing-masing prosesor (P+1) ke Prosesor 0 adalah sama seperti yang dilakukan pada algoritma sebelumnya.



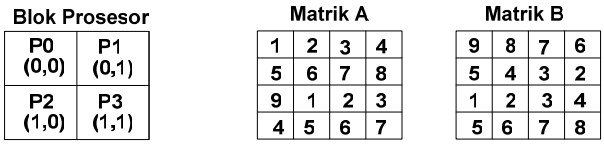
Gambar 2.10 Proses Pertukaran Sub-Blok B

dan Perkalian Matriks Masing-masing Prosesor

2.5.4 Algoritma *Checkboard-Block*

Dengan asumsi yang sama seperti dua algoritma sebelumnya, Jika pada dua algoritma sebelumnya dekomposisi (pemecahan) nilai N dilakukan secara horizontal (Matriks A) dan vertikal (Matriks B). Pada algoritma *checkboard-block* ini dekomposisi menjadi lebih kecil menjadi sub-blok Matriks persegi.

Pada algoritma ini jumlah prosesor (P) juga ikut didekomposisikan menjadi sub-blok persegi. Untuk mendapatkan sub-blok prosesor persegi tersebut, maka diasumsikan jumlah prosesor (P) harus merupakan bilangan yang habis diakarkan (*perfect square*). Sehingga terjadi pembentukan susunan-susunan sub-blok $\sqrt{P}$ baris dan kolom. Untuk lebih jelasnya dapat dilihat pada gambar 2.11.



Gambar 2.11 Inisialisasi Data Matriks Algoritma *Checkboard-Block*

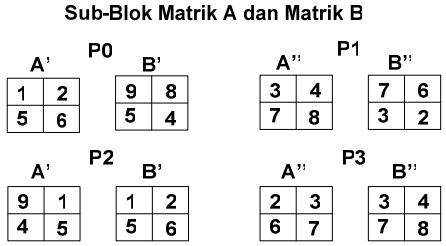
Sebagai contoh, Matriks (N x N) berukuran 4 x 4 dan akan dilakukan komputasi paralel menggunakan 4 prosesor, maka tiap prosesor akan mendapaatkan sub-blok matriks berukuran sebesar 2 x 2 $\left(\bar{n} \times \bar{n}\right)$ dimana $\bar{n}$ adalah:

$$\left(\bar{n}\right) = \frac{N}{\sqrt{P}} \tag{7}$$

Keterangan:

N adalah banyaknya dari baris dan kolom matriks

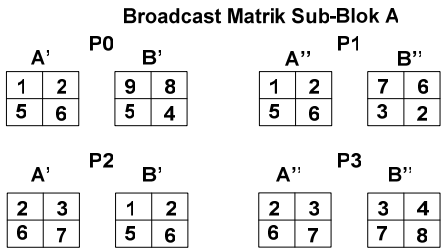
$\sqrt{P}$ adalah besar *grid* (blok-prosesor).



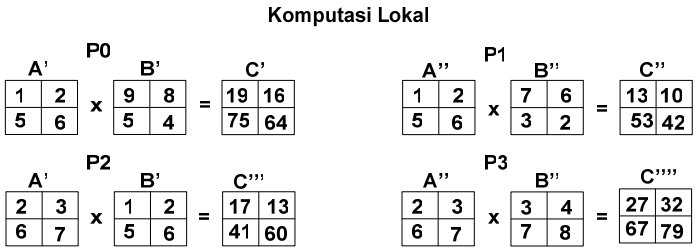
Gambar 2.12 *Scatter* Matriks A dan Matriks B Algoritma *Checkboard-Block*

Ada dua jenis algoritma yang mengerjakan perkalian matriks dengan menggunakan algoritma *checkboard-block*, yaitu *Algoritma Fox dan Algoritma Cannon*, Dalam tugas akhir ini hanya mengerjakan *Algoritma Fox*.

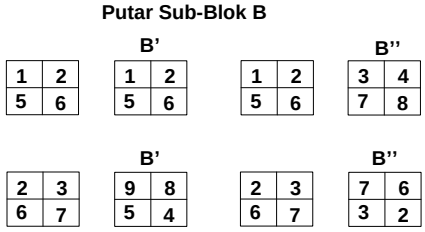
Sedangkan perbedaan keduanya adalah pada proses langkah yang dilakukan pada setiap perulangan yang dilakukan dengan jumlah perulangan sebanyak $\sqrt{P}$, pada algoritma Fox, setiap perulangan melakukan langkah broadcast, perkalian dan pertukaran data (*roll*). Sedangkan algoritma *cannon* melakukan pertukaran data (*roll*) terlebih dahulu.



Gambar 2.13 Proses *Broadcast* Sub-Blok A ke Prosesor yang Sejajar

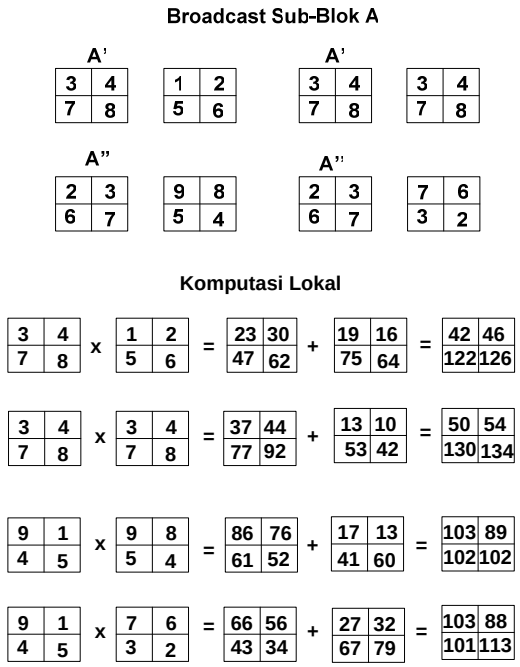


Gambar 2.14 Proses komputasi Matriks pada masing-masing Prosesor dan menyimpan hasilnya pada Sub-blok $C^{(i)}$.



Gambar 2.15 Pertukaran Sub-Blok B dengan Prosesor yang memiliki Nilai Kordinat Kolom yang sama

Untuk setiap perulangan algoritma ini melakukan *broadcast* sub-blok Matriks A ke semua prosesor yang sebaris, dapat dilihat pada gambar 2.13, kemudian masing-masing prosesor melakukan komputasi pada data sub-blok Matriks yang dimilikinya. Setelah melakukan komputasi seperti pada gambar 2.14, dilakukan pertukaran data sub-blok Matriks B ke semua prosesor yang sekolom seperti pada gambar 2.15. Dari gambar 2.16, dapat dilihat secara keseluruhan hasil dari langkah-langkah algoritma *checkboard-block.*



Gambar 2.16 Proses Algoritma Fox pada perulangan ke i+1