Faktor Exacta 7(1): 1-11, 2014
ISSN: 1979-276X

ARSITEKTUR PROGRAM PARALEL BERBASIS MESSAGE-PASSING INTERFACE

YUSTINA SRI SUHARINI
yustina.ss@gmail.com
085779277771
Teknik Informatika Institut Teknologi Indonesia

Abstrak. Komputer bekerja dengan cara mengeksekusi baris-baris kode (program) yang ada di memori utama. Jumlah baris dan kompleksitas kode (program) sangat mempengaruhi banyak sedikitnya sumber daya sistem komputer yang diperlukan untuk mengeksekusi kode (program) yang bersangkutan. Terdapat banyak kasus pada mana kode (program) membutuhkan sumber daya yang sangat besar sehingga diperlukan waktu yang sangat lama untuk menyelesaikan eksekusi. Penelitian ini merupakan sebuah tahap awal untuk menyelidiki apakah arsitektur paralel bisa menjawab tantangan tersebut. Pada tahap awal ini, dibangun sebuah kode (program) dengan arsitektur paralel, dengan menggunakan Message-Passing Interface. Metodologi yang digunakan adalah metodologi pembangunan perangkat lunak dengan model proses prototyping, dilanjutkan dengan analisis hasil eksekusi dan penarikan kesimpulan. Pertama-tama, prototipe dibuat untuk dijalankan di sebuah komputer pribadi. Kemudian prototipe dikembangkan untuk bisa dijalankan pada lima komputer pribadi secara bersamaan. Eksperimen dilakukan menggunakan perangkat keras berupa lima unit komputer bersistem operasi Linux Fedora 19 yang tergabung dalam sebuah sistem jaringan. Penelitian dilakukan dengan bantuan biaya dari Direktorat Jenderal Pendidikan Tinggi melalui skema Hibah Dosen Pemula pendanaan tahun 2013.

Kata Kunci: program paralel, Message-Passing Interface

Abstract. Computer executes process in main memory which is scheduled by operating system. Number of resources needed to execute depends on the size and complexity of the process. In many cases, process needs huge number of resources to run. On the other hand, system has only limited number of resources, so it takes very long time to run. It is possible that for the kind of process to finish execution in years, the thing we want to avoid. This is the beginning of our studying about the possibility of parallel architecture to address the problem. For the first moment we built a parallel program prototype to run on a single personal computer. For the next moment we extended the prototype to run on five personal computers. The prototype is built by using Message-Passing Interface, a standard communication for parallel architecture program. This research is funded by the Directorate General of Higher Education of Indonesia (DIKTI) by PDP scheme 2013.

Keywords: parallel program, Message-Passing Interface

PENDAHULUAN

Komputer merupakan alat yang diciptakan untuk mempermudah pekerjaan manusia, misalnya pekerjaan-pekerjaan yang membutuhkan kecepatan tinggi, rutinitas tinggi, ataupun pekerjaan dengan ukuran data yang relatif besar. Setiap beban pekerjaan yang dilakukan oleh komputer membutuhkan sejumlah sumber daya (*resources*), yang besarnya tergantung pada beban komputasi yang ditangani. Sumber daya yang dibutuhkan bisa berupa waktu eksekusi prosesor (*CPU time*), ruang pada memori utama (*memory space*), koneksi untuk transfer data, atau sumber daya lain yang sesuai. Terdapat

banyak kasus pada mana kalkulasi membutuhkan sumber daya yang relatif besar dibanding kasus-kasus lainnya, misalnya kalkulasi matriks dengan order tinggi dan karakteristik yang sangat spesifik.

Ketika sebuah pekerjaan dilakukan oleh komputer, sistem operasi mengelola seluruh sumber daya yang dibutuhkan untuk menjalankan pekerjaan tersebut. Pengelolaan sumber daya yang dilakukan oleh sistem operasi meliputi pengelolaan proses, memori, input output, dan file yang terlibat atau yang dibutuhkan. Setelah sistem operasi menyiapkan sumber daya yang dibutuhkan, prosesor akan dijadwalkan untuk mengeksekusi proses dengan cara membaca baris demi baris kode yang telah diletakkan di ruang tertentu dalam memori utama komputer. Satu baris kode membutuhkan waktu sejumlah tertentu siklus clock. Semakin banyak jumlah baris kode yang harus dibaca, atau semakin kompleks alur logika sebuah program tentunya semakin lama pula waktu yang diperlukan oleh prosesor untuk mengeksekusinya. Dengan kata lain, jumlah baris kode dan tingkat kompleksitas program mempengaruhi jumlah *CPU time* (clock cycles) yang diperlukan untuk eksekusi.

Persoalan muncul ketika sebuah kode mempunyai jumlah baris yang relatif sangat banyak, atau sangat kompleks misalnya karena mempunyai banyak *nested loop* (pengulangan di dalam pengulangan). Sebuah kode yang seperti itu akan membutuhkan CPU time yang juga sangat banyak, sehingga diperlukan waktu berbulan-bulan bahkan bertahun-tahun untuk menyelesaikannya. Tabel 1 merupakan sebuah contoh kebutuhan CPU time yang sangat tinggi. Kode dibuat oleh penulis untuk mencari bitstring A dan bitstring B yang keduanya mempunyai karakteristik tertentu, sebagai tahap awal kalkulasi matriks skew-Hadamard yang terdiri atas matrik-matrik A, B, C, D dengan susunan pola dan keteraturan yang sudah ditentukan.

Tabel 1. Waktu Eksekusi (1 Core CPU time) Pencarian Bitstring A dan B

Bitstring A	Bitstring B	CPU Time	Keterangan
1 bitstring	1 bitstring	3.5 detik	
1 bitstring	31824 bitstring	111384 detik	≅ 30.94 jam
131072 bitstring	31824 bitstring	4055367 jam	≅ 168973.653 hari ≅ 462.941516 tahun

Tabel 1 merupakan sebuah catatan tentang kebutuhan waktu eksekusi prosesor (*CPU time*) untuk pencarian kelompok bitstring A dan B. Waktu yang diperlukan untuk mengeksekusi kode agar ditemukan sebanyak setengah anggota himpunan (setengah belahan simetris) bitstring A adalah 4055367 jam atau 168973,653 hari yang jika dibagi 365 didapatkan waktu eksekusi 462,941516 tahun. Itu apabila kode dieksekusi oleh satu core CPU tunggal yang berdiri sendiri.

Perumusan Masalah

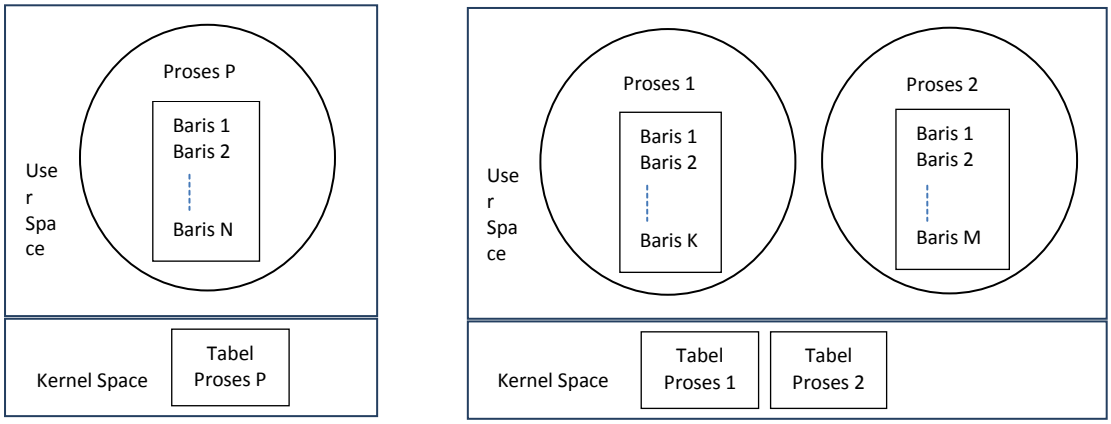
Persoalan yang dicoba atasi pada penelitian ini adalah bagaimana mempercepat waktu eksekusi (*CPU time*) yang terlalu lama. Dengan demikian pertanyaan yang hendak dicoba jawab melalui penelitian ini adalah : bisakah teknologi komputer yang berkembang saat ini membantu percepatan waktu kalkulasi untuk program yang kompleks seperti pencarian matriks ?

Tujuan

Penelitian bertujuan untuk menyelidiki apakah arsitektur paralel bisa meningkatkan kinerja eksekusi program yang kompleks, misalnya mempercepat pencarian matriks.

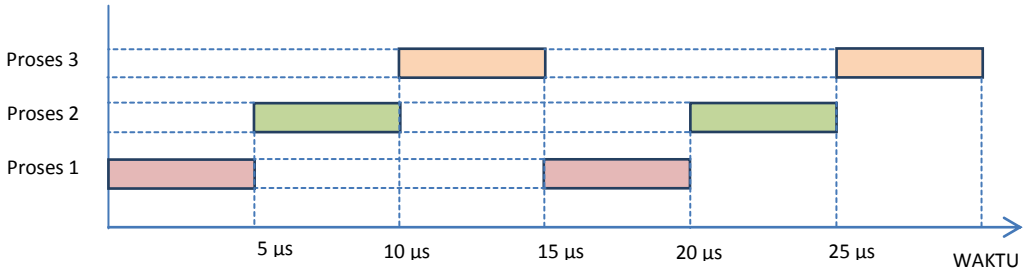
TINJAUAN PUSTAKA

Proses didefinisikan sebagai program yang sedang aktif, dalam arti sedang berada di memori utama, dalam keadaan siap dieksekusi atau sedang dieksekusi oleh prosesor. Sebuah proses tunggal yang dieksekusi oleh prosesor dengan cara serial dapat dilihat pada Gambar 1. Proses dieksekusi baris demi baris dari awal sampai akhir dengan sumber daya yang dikelola oleh sistem operasi. Apabila jumlah proses yang sedang aktif dalam satu saat ada lebih dari satu, maka sistem operasi menjadwalkan prosesor agar mengeksekusi proses-proses itu berdasarkan algoritma penjadwalan yang diberlakukan.



Gambar 1. Memori utama dengan satu proses dan dua proses.

Terdapat beberapa macam algoritma penjadwalan yang mungkin bagi sistem operasi. Salah satu algoritma penjadwalan yang dilakukan berdasarkan prinsip keadilan (*fair share*) untuk beberapa proses yang mempunyai prioritas yang sama dapat dilihat pada Gambar 2. Pada gambar ini prosesor dijadwalkan untuk mengeksekusi proses-proses secara bergantian dengan alokasi waktu yang sama untuk setiap kesempatan.

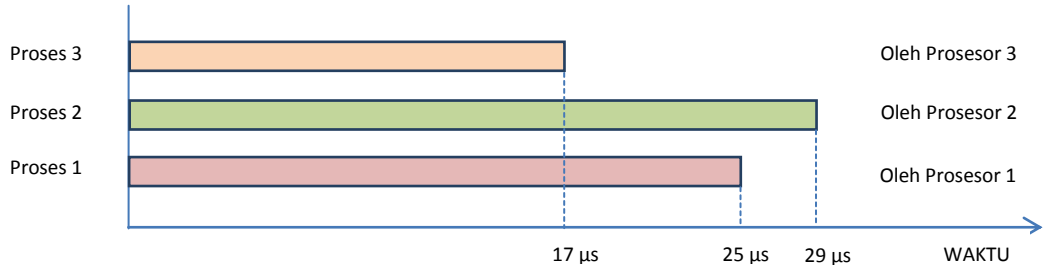


Gambar 2. Eksekusi prosesor untuk proses-proses berprioritas sama.

Apapun jenis penjadwalan prosesor yang dilakukan oleh sistem operasi, keseluruhan baris program yang ada dalam tiap proses akan dieksekusi dari awal sampai akhir. Itu berarti dua hal. Pertama, jumlah baris program yang lebih banyak akan membutuhkan waktu eksekusi yang lebih lama pula. Kedua, jumlah total baris program untuk semua proses yang sedang aktif akan dieksekusi, tidak peduli perlu waktu berapa lama. Masalah muncul apabila waktu yang diperlukan terlalu lama, berbulan-bulan bahkan bertahun-tahun, kadang menyebabkan hasil eksekusi sudah tidak relevan lagi seiring bertambahnya waktu.

Arsitektur program paralel adalah arsitektur yang memungkinkan sistem operasi untuk menjadwalkan proses-proses pada lebih dari satu prosesor yang bereksekusi secara bersamaan. Dengan melibatkan banyak prosesor yang bekerja secara paralel pada saat yang bersamaan, maka diharapkan kinerja sistem menjadi jauh lebih baik, sehingga waktu yang diperlukan untuk mengeksekusi proses keseluruhan dapat dipersingkat.

Eksekusi pada arsitektur paralel secara umum dapat dilihat pada Gambar 3. Seperti terlihat pada gambar, masing-masing prosesor mempunyai tugas untuk mengeksekusi bagian-bagian yang berbeda satu sama lain sehingga ada semacam pembagian tanggung jawab yang berbeda-beda yang semuanya merupakan bagian lebih kecil dari atau sama dengan tanggung jawab sistem keseluruhan. Eksekusi proses 1 dilakukan oleh prosesor 1, demikian juga eksekusi proses 2 dilakukan oleh prosesor 2, dan eksekusi proses 3 dilakukan oleh prosesor 3. Kinerja sistem secara keseluruhan didukung oleh seluruh prosesor yang ada dalam sistem.



Gambar 3. Eksekusi 3 proses oleh 3 prosesor secara paralel.

Hipotesis

Berdasarkan logika di atas, diambil hipotesis bahwa arsitektur paralel memungkinkan peningkatan kinerja sistem. Artinya, dengan program paralel seharusnya kecepatan kalkulasi meningkat dibandingkan dengan program yang dieksekusi oleh prosesor tunggal. Kembali pada persoalan utama tentang waktu eksekusi pencarian setengah jumlah bitstring A dan B yang sangat lama yaitu 4055367 jam atau 168973,653 hari (atau 462,941516 tahun) CPU time, maka diharapkan arsitektur paralel akan mempercepat waktu eksekusi tersebut. Dapat dikatakan secara sederhana bahwa untuk proses yang tingkat kompleksitasnya rendah, misalnya proses dengan nilai *cyclomatic complexcity* 1, kebutuhan jumlah sumber daya berbanding lurus dengan jumlah baris program. Oleh karena itu jika dalam sistem terdapat 240 core prosesor maka waktu yang diperlukan menjadi 1.92892298 tahun atau dibulatkan menjadi 2 tahun. Jika dalam sistem terdapat 1000 core prosesor maka waktu eksekusi menjadi 168.973653 hari atau 24.1390933 minggu. Demikian pula jika dipunyai 2000 core maka waktu eksekusi diperkirakan 84.4868267 hari.

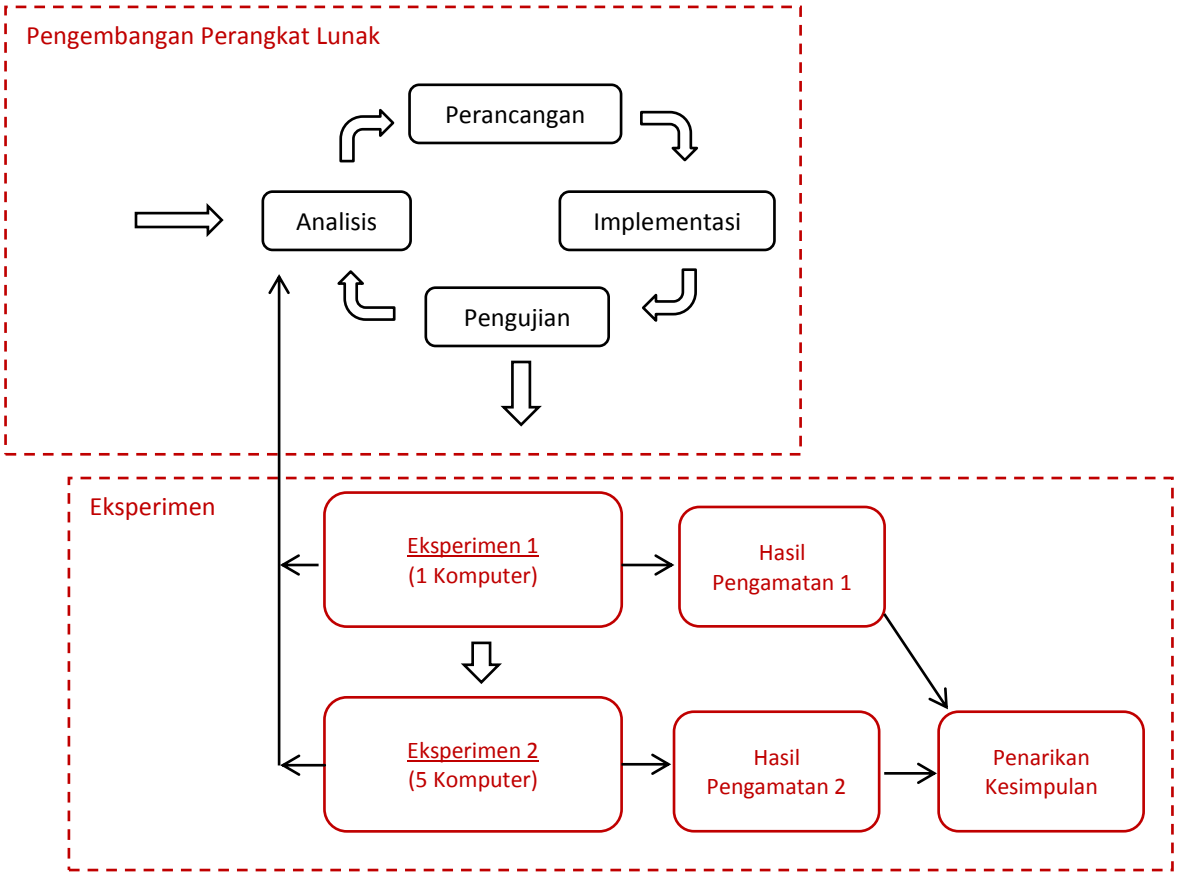
METODE

Metode penelitian terdiri atas dua bagian. Pertama, metode pengembangan perangkat lunak menggunakan model proses prototyping. Kedua, metode eksperimen yaitu metode pengamatan terhadap hasil eksekusi program.

Pengembangan Perangkat Lunak

Metode yang digunakan dalam tahap pertama adalah pengembangan perangkat lunak dengan model proses *prototyping*. Tahapan dalam model proses prototyping adalah sebagai berikut.

(1) Analisis kebutuhan perangkat lunak, yaitu tahap pemahaman persoalan, ruang lingkup, batasan, penentuan daftar fitur perangkat lunak. Luaran tahap ini berupa gambaran menyeluruh atau spesifikasi lengkap terhadap perangkat lunak yang akan dirancang.
(2) Perancangan perangkat lunak, yaitu tahap pendekomposisian perangkat lunak menjadi modul-modul dan sub-sub modul. Dilakukan perancangan algoritma dan struktur data untuk setiap modul dan sub modul. Luaran berupa daftar seluruh modul dan sub modul, algoritma dan struktur data untuk setiap modul dan sub modul tersebut.
(3) Implementasi hasil rancangan ke dalam program, yaitu tahap penulisan kode program untuk setiap sub modul yang kemudian diintegrasikan ke dalam modul yang sesuai. Program ditulis dalam Bahasa C.
(4) Pengujian, yaitu tahap menguji semua sub modul yang telah dituliskan dalam kode program. Selain pengujian terhadap sub-sub modul, dilakukan juga pengujian integrasi terhadap kumpulan sub modul.
(5) Pengamatan hasil untuk putaran berikutnya, yaitu tahap pengamatan apakah perangkat lunak dapat berfungsi sesuai spesifikasi yang ditetapkan. Jika belum sesuai, maka dilakukan perbaikan sebagai masukan atau *feedback* untuk siklus pengembangan perangkat lunak berikutnya.



Gambar 4. Metode pengembangan perangkat lunak dan metode eksperimen.

Eksperimen

Eksperimen dilakukan setelah perangkat lunak selesai dibuat. Eksperimen terdiri atas dua tahap utama

(1) Eksperimen tahap pertama dilakukan dengan mengeksekusi program pada sebuah komputer pribadi, dengan empat program client dan satu program server.
(2) Eksperimen tahap kedua dilakukan menggunakan lima komputer yang terhubung. Empat program *client* dan satu program *server* masing-masing dijalankan pada komputer yang berbeda.

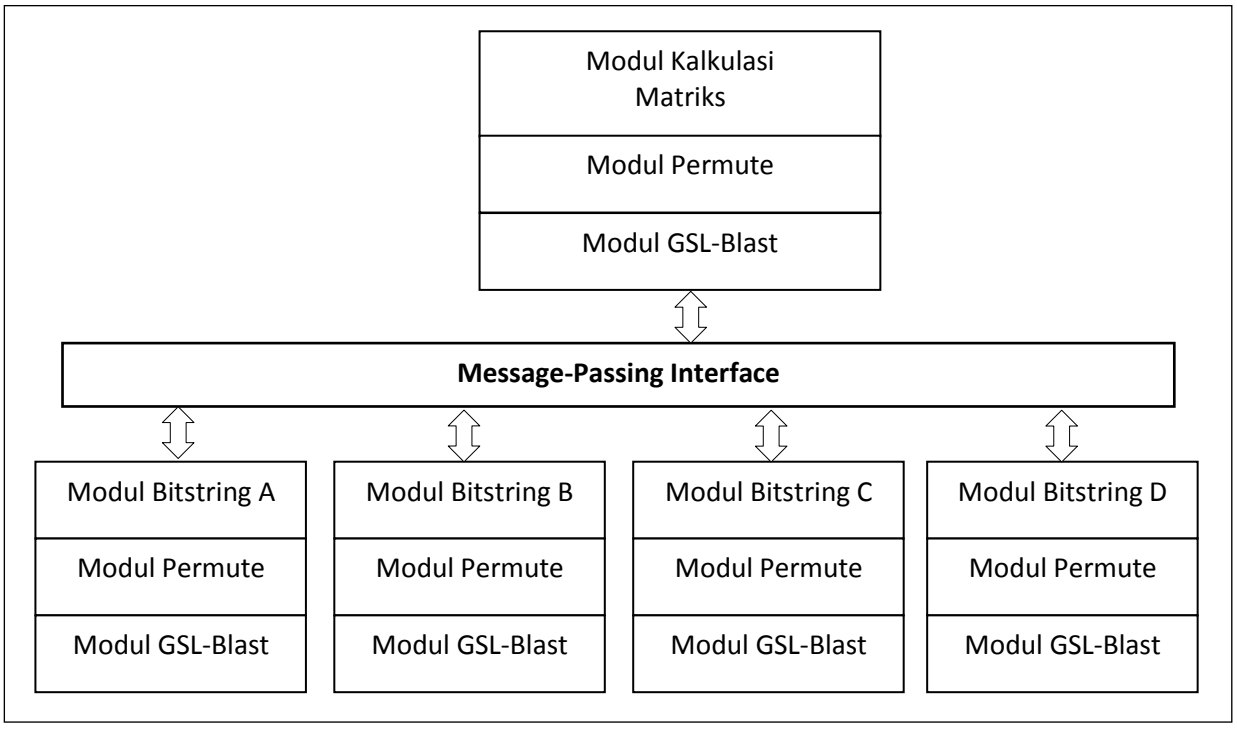
Metode penelitian yang telah diuraikan dalam dua tahap di atas dapat dilihat pada Gambar 4.

HASIL DAN PEMBAHASAN

Penelitian menghasilkan prototipe program paralel dan perbandingan antara waktu yang dibutuhkan untuk mengeksekusi program dengan komputer tunggal dan waktu yang dibutuhkan untuk mengeksekusinya di atas lima komputer yang terhubung dalam jaringan komputer.

Arsitektur Prototipe Program

Arsitektur prototipe program paralel yang dibuat dapat dilihat pada Gambar 5.



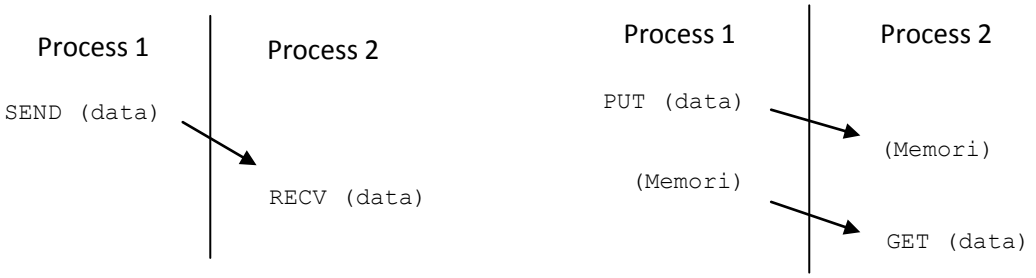
Gambar 5. Arsitektur prototipe program paralel yang dibuat untuk kalkulasi matriks.

Program kalkulasi terdiri atas lima modul utama, yaitu modul pencarian bitstring A, modul pencarian bitstring B, modul pencarian bitstring C, modul pencarian bitstring D, dan modul kalkulasi matriks yang bertugas mencari matriks. Matriks yang dicari tersusun atas A, B, C, D dengan karakteristik susunan yang sangat spesifik. Keempat modul pencarian bitstring A, B, C, D, berperan sebagai program *client* yang melaporkan

hasil pencarian ke modul kalkulasi matriks, yang berperan sebagai modul *server*. Modul *server* menerima hasil pencarian dari keempat modul *client* lalu melakukan kalkulasi berdasarkan masukan yang diberikan. Di samping kelima modul tersebut, dikembangkan modul *Message-Passing Interface* (MPI) yang bertugas mengelola transfer data seluruh modul kalkulasi. MPI merupakan spesifikasi sekaligus library yang dikembangkan untuk mempermudah pembuatan program paralel. Berikut ini penjelasan tentang implementasi MPI dalam arsitektur program paralel yang dibuat.

Komputasi paralel dapat diartikan sebagai komputasi yang dilakukan pada lebih dari satu proses yang terpisah. Interaksi antar proses terjadi saat ada pertukaran informasi dari satu proses ke proses lain dan sebaliknya. Jenis-jenis komputasi paralel ditentukan oleh jenis program dan data yang terlibat dalam komputasi. Sebagai contoh, SIMD adalah komputasi dengan satu macam operasi dan banyak macam data, yang berarti masing-masing proses menangani data yang berbeda satu sama lain. Sedangkan MIMD adalah komputasi dengan banyak operasi dan banyak macam data. Biasanya MIMD dapat dibentuk dari SIMD. Pada dasarnya *Message-Passing Interface* (MPI) ditujukan untuk SIMD dan MIMD. [2]

Komunikasi antar proses dapat dilakukan dengan dua macam cara yaitu cara kooperatif dan cara sepihak. Pertukaran data secara kooperatif melibatkan partisipasi kedua pihak yaitu pihak pengirim dan penerima pesan. Karakteristik penting dari cara kooperatif adalah bahwa setiap perubahan yang terjadi pada memori penerima dibuat dengan persetujuan penerima. Pengirim tidak bisa memaksakan perubahan di sisi penerima. Dengan demikian data harus didefinisikan secara eksplisit untuk dikirim (oleh pihak pengirim) dan diterima (oleh pihak penerima). Sedangkan transfer data sepihak hanya melibatkan partisipasi satu pihak. [5] Gambar 6 mendeskripsikan perbedaan antara transfer data yang dilakukan secara kooperatif dan transfer data yang dilakukan sepihak.



Gambar 6. Interaksi kooperatif dan interaksi sepihak.

Pengiriman dan penerimaan pesan menggunakan *Message-Passing Interface* (MPI) dilakukan menggunakan perintah-perintah berikut. [2] [3]

(1) Perintah pengiriman pesan / (*blocking*) *send* :

```
MPI_Send (start, count, datatype, dest, tag, comm)
```

(2) Perintah penerima pesan / *receive* :

```
MPI_Recv ( start, count, datatype, source, tag, comm,
status)
```

(3) Sumber, tag, jumlah pesan yang diterima dapat di-retrieve dari status.

```
MPI_Status status;
MPI_Recv( ..., &status );
... status.MPI_TAG;
... status.MPI_SOURCE;
```

(4) Operasi broadcast dan reduction :

```
MPI_Bcast (start, count, datatype, root, comm)
MPI_Reduce (start, result, count, datatype, operation, root,
comm)
```

(5) Perintah untuk mendapatkan informasi tentang sebuah pesan :

```
MPI_Get_count( &status, datatype, &count );
```

MPI merupakan spesifikasi dan sekaligus library yang bisa dipanggil dan dideklarasikan sesuai kebutuhan. Sebelum bisa dipanggil dan dideklarasi, spesifikasi MPI perlu di-*invoke* atau di-*include*, yang jika dikode dalam Bahasa C maka kerangka programnya kira-kira seperti ini.

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char **argv;
{
MPI_Init( &argc, &argv );
/*  program client/server ditulis di sini  */
MPI_Finalize();
return 0;
}
```

Jika diperlukan empat program client dan satu program server sehingga total diperlukan lima modul program, maka kelima-limanya juga harus melibatkan pemanggilan MPI sehingga transfer data antar modul bisa dilakukan. Kode (program) untuk kelima modul perlu dikompilasi (*compiled*) menggunakan kompilator MPI.

```
$ mpicc –o modulA modulA.c
```

MPI memungkinkan banyak proses dieksekusi secara paralel pada waktu yang sama. Untuk mengetahui berapa jumlah proses yang sedang dieksekusi, dapat digunakan perintah

```
MPI_Comm_size
```

Sedangkan untuk mengetahui identitas proses yang sedang diamati (seperti perintah “ `who am I` “ dalam *console* Unix atau Linux) digunakan perintah

```
MPI_Comm_rank
```

Berikut ini contoh program sederhana yang melakukan panggilan terhadap kedua perintah. [2]

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char **argv;
{
int rank, size;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf( "Hello world! I'm %d of %d\n",
rank, size );
MPI_Finalize();
return 0;
}
```

Modul tersebut juga harus dikompilasi menggunakan kompilator yang sesuai agar bisa dieksekusi dalam arsitektur paralel

```
$ mpicc –o hello hello.c
```

Apabila program cukup besar maka perlu dibuat kompilasi otomatis menggunakan standard Makefile untuk MPI.

```
$ mpireconfig Makefile
```

Cara tersebut memungkinkan program besar dari berbagai kompilator seolah disatukan dan memungkinkan untuk bekerja sama dalam sebuah jaringan komputer atau dalam sistem komputer paralel. Untuk bisa menjalankan `mpireconfig`, perintah tersebut harus sudah ada dalam PATH, jika digunakan sistem operasi Unix atau Linux.

Perbandingan Waktu Eksekusi

Selain prototipe program, hasil penelitian yang lain adalah waktu eksekusi prototipe program pertama dan prototipe program kedua. Waktu eksekusi prototipe program pertama dapat dilihat pada Tabel 2. Pada tabel tersebut disajikan jumlah elemen bitstring yang dicari, yaitu bilangan ganjil positif mulai dari 3, 5, 7, 9, 11, 13, sampai dengan 15, serta kebutuhan waktu eksekusi masing-masing pencarian.

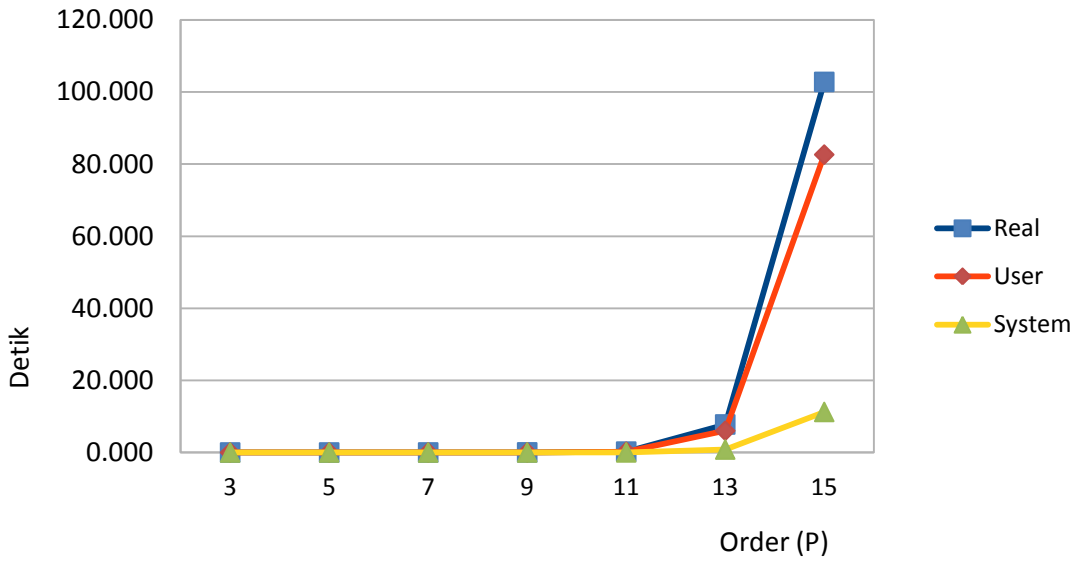
Waktu *real* adalah waktu yang dibutuhkan proses untuk dieksekusi oleh CPU dari awal sampai akhir. Waktu real merupakan waktu total yang terdiri atas waktu *user* dan waktu *system*. Angka-angka yang terdapat pada tabel merupakan angka terdekat yang berhasil dibaca dari *time stamp* yang dicatat oleh sistem operasi. Ketidaksesuaian jumlah waktu *user* dan waktu *system* terhadap waktu *real* terjadi karena pendekatan tersebut.

Tabel 2. Waktu Eksekusi Prototipe Pertama

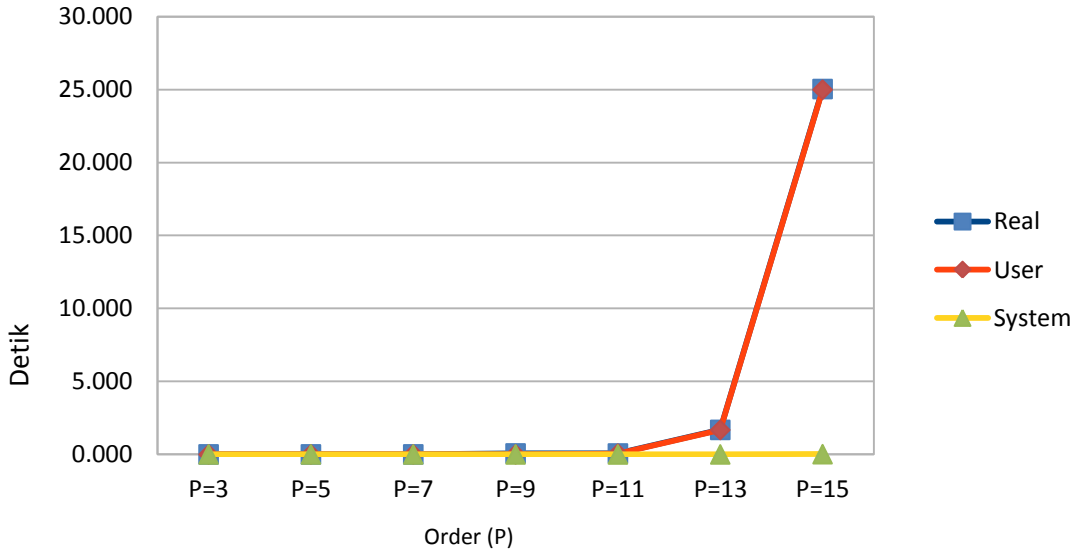
	Waktu Real	Waktu User	Waktu System
Order Bitstring	(detik)	(detik)	(detik)
P=3	0.002	0.001	0.001
P=5	0.003	0.001	0.002
P=7	0.008	0.005	0.002
P=9	0.029	0.020	0.008
P=11	0.243	0.213	0.030
P=13	7.777	6.022	0.827
P=15	102.795	82.634	11.252

Tabel 3. Waktu Eksekusi Prototipe Kedua

	Waktu Real	Waktu User	Waktu System
Order Bitstring	(detik)	(detik)	(detik)
P=3	0.003	0.000	0.002
P=5	0.003	0.001	0.001
P=7	0.003	0.002	0.002
P=9	0.069	0.004	0.002
P=11	0.065	0.054	0.000
P=13	1.683	1.678	0.003
P=15	25.045	25.000	0.019



Gambar 7. Grafik hubungan antara order bitstring dan waktu eksekusi prototipe pertama.



Gambar 8. Grafik hubungan antara order bitstring dan waktu eksekusi prototipe kedua.

Waktu yang diperlukan untuk mengeksekusi kode (program) dalam pencarian matriks menggunakan prototipe program pertama, lebih lama 2,46 kali dibandingkan dengan waktu yang diperlukan untuk mengeksekusi prototipe kedua. Hal itu dapat dijelaskan sebagai berikut.

(1) Pada prototipe program pertama, sumber daya komputer terbatas hanya satu unit komputer. Sementara itu prototipe program kedua dieksekusi oleh lima unit komputer yang terhubung pada jaringan komputer yang tentunya mempunyai jumlah sumber daya yang jauh lebih besar dibanding dengan prototipe pertama.

(2) Ada semacam *overhead* yang harus dibayarkan untuk menghubungkan lima unit komputer dalam sistem jaringan, yang menyebabkan waktu eksekusi tidak berbanding lurus dengan ketersediaan sumber daya komputer. [1] [4]
(3) Jika koneksi dianggap sebagai I/O waiting, maka rumus utility CPU : $U = 1 - p^n$ dengan p adalah I/O waiting, dan n merupakan banyaknya proses yang terlibat dalam sistem, dapat diberlakukan.
(4) Kompleksitas program tergolong tinggi, dapat diamati dari ukuran masukan yang berbeda mengakibatkan perbedaan sangat signifikan dalam hal lamanya waktu eksekusi.

PENUTUP

Kesimpulan

Penelitian menghasilkan beberapa kesimpulan:

(1) *Message-Passing Interface* (MPI) dapat digunakan dengan baik untuk mengelola transfer data di antara proses-proses yang terlibat dalam sistem.
(2) Penggunaan arsitektur paralel sangat efektif untuk program yang mempunyai kompleksitas rendah namun tidak efektif untuk program dengan kompleksitas tinggi, dalam arti bahwa arsitektur paralel tidak bisa menurunkan tingkat kompleksitas suatu program.
(3) Komunikasi antar proses di dalam jaringan komputer membutuhkan *effort* atau *overhead* yang menyebabkan penggunaan sumber daya dalam arsitektur paralel tidak berbanding lurus dengan program yang sama namun dengan arsitektur serial.

Saran

Program yang sangat kompleks sebaiknya disusun ulang untuk menurunkan tingkat kompleksitas. Setelah itu baru dibuat arsitektur paralel menggunakan *Message-Passing Interface*.

DAFTAR PUSTAKA

Balaji, P., Chan, A., Gropp, W., Thakur, R. & Lusk, E. 2008. **Non-data-communication Overheads in MPI: Analysis on Blue Gene/P,** EuroPVM/MPI 2008, LNCS 5205, pp. 13–22

Gropp, William. 2010. **Tutorial on MPI: The Message-Passing Interface**, Mathematics and Computer Science Division, Argonne National Laboratory

Hoefler, Torsten & Snir, Marc. 2011. **Writing Parallel Libraries with MPI – Common Practice, Issues, and Extensions,** EuroMPI 2011, LNCS 6960, pp. 345–355

Kumar, Sameer, Dozsa, Gabor, Berg, Jeremy, Cernohous, Bob, Miller, Douglas, Ratterman, Joseph, Smith, Brian, Heidelberger, Philip. 2008. **Architecture of the Component Collective Messaging Interface,** EuroPVM/MPI 2008, LNCS 5205, pp. 23–32

Potluri, Sreeram, Wang, Hao, Dhanraj, Vijay, Sur, Sayantan & Panda Dhabaleswar K. 2011. **Optimizing MPI One Sided Communication on Multi-core InfiniBand Clusters Using Shared Memory Backed Windows,** EuroMPI 2011, LNCS 6960, pp. 99–109