

Teknis dan Strategi Pembangunan Algoritma Paralel

Ayi Purbasari

pbasari@bdg.centrin.net.id

Lisensi Dokumen:

Copyright © 2003-2006 IlmuKomputer.Com

*Seluruh dokumen di **IlmuKomputer.Com** dapat digunakan, dimodifikasi dan disebarkan secara bebas untuk tujuan bukan komersial (nonprofit), dengan syarat tidak menghapus atau merubah atribut penulis dan pernyataan copyright yang disertakan dalam setiap dokumen. Tidak diperbolehkan melakukan penulisan ulang, kecuali mendapatkan ijin terlebih dahulu dari **IlmuKomputer.Com**.*

Sekalipun didukung oleh teknologi prosesor yang berkembang sangat pesat, komputer sekuensial tetap akan mengalami keterbatasan dalam hal kecepatan pemrosesannya. Hal ini menyebabkan lahirnya konsep keparalelan (parallelism) untuk menangani masalah dan aplikasi yang membutuhkan kecepatan pemrosesan yang sangat tinggi, seperti misalnya prakiraan cuaca, simulasi pada reaksi kimia, perhitungan aerodinamika dan lain-lain.

Konsep keparalelan itu sendiri dapat ditinjau dari aspek design mesin paralel, perkembangan bahasa pemrograman paralel atau dari aspek pembangunan dan analisis algoritma paralel. Algoritma paralel itu sendiri lebih banyak difokuskan kepada algoritma untuk menyelesaikan masalah numerik, karena masalah numerik merupakan salah satu masalah yang memerlukan kecepatan komputasi yang sangat tinggi.

Untuk dapat mengadaptasi suatu algoritma sekuensial ke dalam algoritma paralel, terlebih dahulu harus dipelajari mengenai konsep pemrosesan paralel dan bagaimana proses-proses dapat berlangsung secara paralel. Konsep pemrosesan paralel secara tidak langsung melibatkan studi mengenai arsitektur komputer paralel. Karena itu tulisan ini akan membahas mengenai konsep paralel dan arsitektur komputer paralelnya, terutama arsitektur multikomputer. Dimulai dengan pengelompokan Flynn, Arsitektur Komputer Paralel, Konsep Paralelisme, Pembangunan Algoritma Paralel, Konsep Proses dan Proses Komunikasi

PENGELOMPOKAN FLYNN

Berdasarkan jumlah aliran instruksi dan aliran datanya, Michael J. Flynn pada tahun 1966 mengelompokkan komputer digital menjadi empat golongan besar [Hwa85]. Aliran instruksi (*instruction stream*) adalah urutan instruksi yang dieksekusi oleh sistem komputer, sedangkan aliran data (*data stream*) adalah urutan data yang diolah termasuk data masukan, bagian dari data, maupun data sementara yang dipanggil atau digunakan oleh aliran instruksi.

Keempat kelompok komputer tersebut adalah :

1. Komputer SISD (Single Instruction stream-Single Data stream)

Pada komputer jenis ini semua instruksi dikerjakan terurut satu demi satu, tetapi juga dimungkinkan adanya *overlapping* dalam eksekusi setiap bagian instruksi (*pipelining*). Pada umumnya komputer SISD berupa komputer yang terdiri atas satu buah pemroses (*single*

processor). Namun komputer SISD juga mungkin memiliki lebih dari satu unit fungsional (modul memori, unit pemroses, dan lain-lain), selama seluruh unit fungsional tersebut berada dalam kendali sebuah unit pengendali. Skema arsitektur global komputer SISD dapat dilihat pada gambar .1 (a).

2. Komputer SIMD (Single Instruction stream-Multiple Data stream)

Pada komputer SIMD terdapat lebih dari satu elemen pemrosesan yang dikendalikan oleh sebuah unit pengendali yang sama. Seluruh elemen pemrosesan menerima dan menjalankan instruksi yang sama yang dikirimkan unit pengendali, namun melakukan operasi terhadap himpunan data yang berbeda yang berasal dari aliran data yang berbeda pula. Skema arsitektur global komputer SIMD dapat dilihat pada gambar .1 (b).

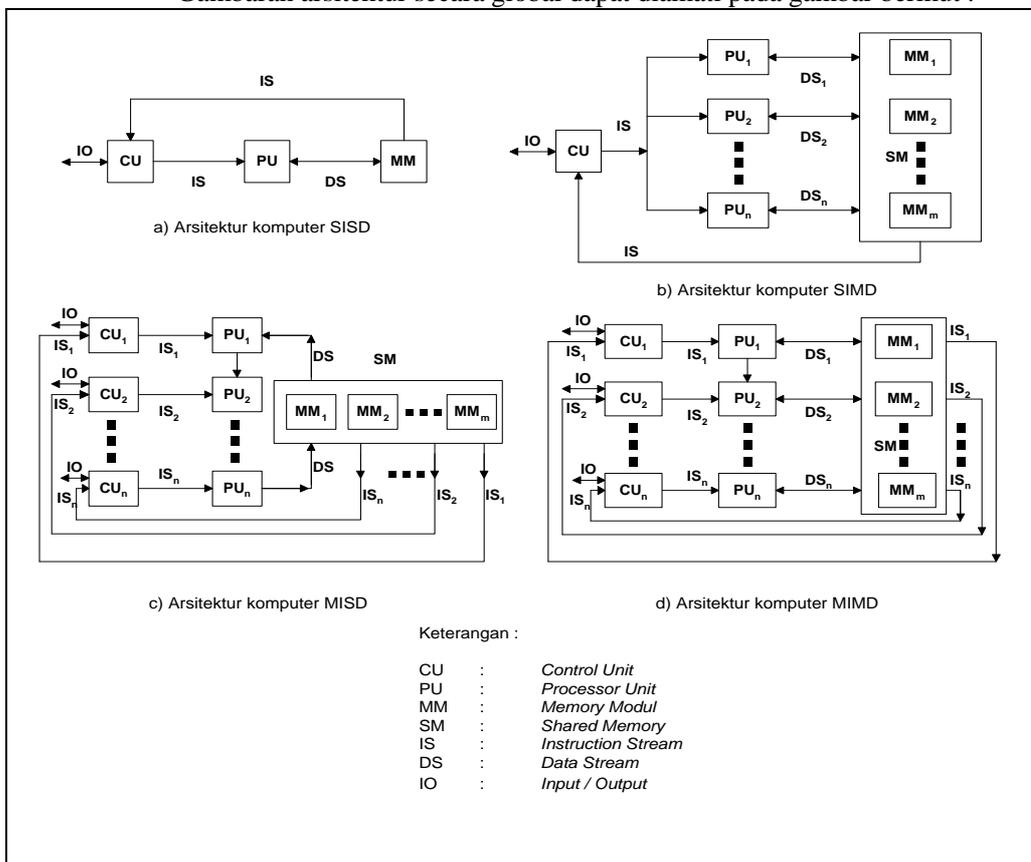
3. Komputer MISD (Multiple Instruction stream-Single Data stream)

Komputer jenis ini memiliki n unit pemroses yang masing-masing menerima dan mengoperasikan instruksi yang berbeda terhadap aliran data yang sama, dikarenakan setiap unit pemroses memiliki unit pengendali yang berbeda. Keluaran dari satu pemroses menjadi masukan bagi pemroses berikutnya. Belum ada perwujudan nyata dari komputer jenis ini kecuali dalam bentuk prototipe untuk penelitian. Skema arsitektur global komputer MISD dapat dilihat pada gambar .1 (c).

4. Komputer MIMD (Multiple Instruction stream-Multiple Data stream)

Pada sistem komputer MIMD murni terdapat interaksi di antara n pemroses. Hal ini disebabkan seluruh aliran dari dan ke memori berasal dari *space* data yang sama bagi semua pemroses. Komputer MIMD bersifat *tightly coupled* jika tingkat interaksi antara pemroses tinggi dan disebut *loosely coupled* jika tingkat interaksi antara pemroses rendah.

Gambaran arsitektur secara global dapat diamati pada gambar berikut :



Gambar 1 Pengelompokan Arsitektur Komputer Menurut Flynn

ARSITEKTUR KOMPUTER PARALEL

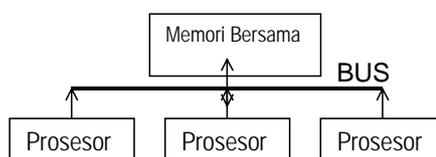
Komputer sekuensial - berdasarkan klasifikasi Flynn adalah kelompok komputer SISD - hanya mempunyai satu unit pengendali untuk menentukan instruksi yang akan dieksekusi. Pada setiap satuan waktu hanya satu instruksi yang dapat dieksekusi, dimana kecepatan akses ke memori dan kecepatan piranti masukan dan keluaran dapat memperlambat proses komputasi. Beberapa metoda dibangun untuk menghindari masalah tersebut, seperti penggunaan *cache memory*. Namun komputer sekuensial ini tetap mengalami keterbatasan jika menangani masalah yang memerlukan kecepatan tinggi. Hal-hal tersebut di atas pada akhirnya melatarbelakangi lahirnya sistem komputer paralel.

Berdasarkan klasifikasi Flynn, komputer paralel termasuk kelompok SIMD atau MIMD. Komputer paralel mempunyai lebih dari satu unit pemroses dalam sebuah komputer yang sama. Hal yang membuat suatu komputer dengan banyak prosesor disebut sebagai komputer paralel adalah bahwa seluruh prosesor tersebut dapat beroperasi secara simultan. Jika tiap-tiap prosesor dapat mengerjakan satu juta operasi tiap detik, maka sepuluh prosesor dapat mengerjakan sepuluh juta operasi tiap detik, seratus prosesor akan dapat mengerjakan seratus juta operasi tiap detiknya [Les93].

Pada dasarnya aktivitas sebuah prosesor pada komputer paralel adalah sama dengan aktivitas sebuah prosesor pada komputer sekuensial. Tiap prosesor membaca (*read*) data dari memori, memprosesnya dan menuliskannya (*write*) kembali ke memori. Aktivitas komputasi ini dikerjakan oleh seluruh prosesor secara paralel.

HUBUNGAN ANTAR PROSESOR

Sistem komputer paralel dengan banyak prosesor yang bekerja secara simultan memerlukan kemampuan untuk membagi data dan berkomunikasi antar prosesor. Berdasarkan kedua kebutuhan tersebut, terdapat dua arsitektur komputer paralel, yaitu memori bersama dan *message passing* [Les93]. Pada arsitektur memori bersama, biasanya disebut multiprosesor (*multiprocessor*), setiap prosesor dapat mengakses memori global dan menggunakan isi data dan struktur data yang disimpan dalam memori bersama (*shared memory*). Prosesor berkomunikasi dengan prosesor lain dengan menulis pesan ke memori global dimana prosesor kedua dapat membaca pesan tersebut pada lokasi memori yang sama. Skema arsitektur memori bersama dapat dilihat pada gambar .2.



Gambar 2 Shared Memory Multiprocessor

Semua prosesor dapat melakukan komputasi secara paralel dan masing-masing dapat mengakses memori melalui bus. Bus bertanggung jawab mengatur permintaan pemakaian memori yang berlangsung secara simultan oleh beberapa prosesor. Bus juga bertanggung jawab untuk meyakinkan bahwa semua prosesor dilayani secara adil dengan waktu tunda (*delay*) akses yang minimum.

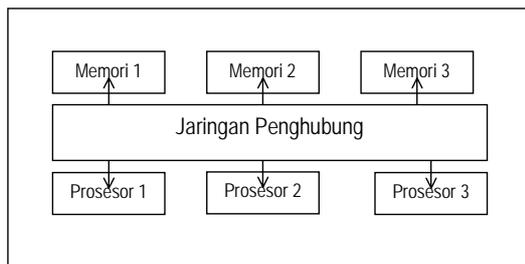
Salah satu kesulitan utama dari arsitektur multiprosesor dengan memori bersama adalah kemungkinan adanya tabrakan memori (*memory contention*). Peristiwa ini terjadi ketika beberapa prosesor mencoba untuk mengakses memori bersama dalam periode waktu yang sangat singkat, sehingga memori tidak akan dapat menampung semua permintaan secara simultan. Akibatnya beberapa prosesor akan harus menunggu sampai prosesor lainnya dilayani. Kemungkinan terjadinya tabrakan memori ini berbanding lurus dengan bertambahnya jumlah prosesor.

Beberapa teknik telah dikembangkan untuk mereduksi tabrakan memori dan membuat sistem menjadi lebih efisien. Salah satu teknik adalah dengan adanya *cache memory* lokal pada masing-masing prosesor. *Cache memory* ini digunakan untuk menyimpan salinan isi memori yang paling

baru dipergunakan. Hal ini menimbulkan masalah baru, yaitu *cache coherent problem*, dimana akan banyak salinan isi memori pada *cache* yang berbeda yang menimbulkan adanya kemungkinan isi *cache* yang *outdate* setelah isi memori bersama diperbaharui (*update*).

Teknik lain untuk mereduksi tabrakan memori adalah dengan membagi memori bersama tersebut menjadi beberapa bagian modul yang dapat diakses secara paralel oleh prosesor yang berbeda. Data disebar ke beberapa modul memori untuk menghindari kemungkinan permintaan yang simultan ke modul memori yang sama oleh beberapa prosesor. Setiap n prosesor dapat mengakses m modul memori melalui jaringan penghubung prosesor - memori (*processor - memory connection network*).

Skema arsitektur memori bersama dengan jaringan penghubung prosesor - memori dapat dilihat pada gambar .3.

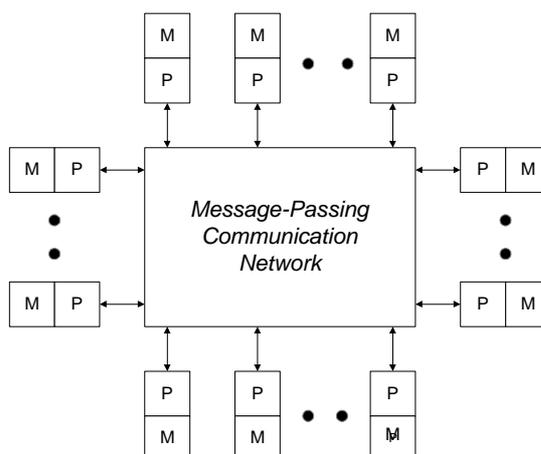


Gambar 3 Memori Bersama dengan Modul-modul

Jaringan penghubung ini menyebabkan beberapa prosesor dapat mengakses modul-modul memori yang berbeda secara simultan. Biaya dan kinerja tipe multiprosesor ini tergantung pada rancangan internal jaringan penghubung. Beberapa contoh jaringan penghubung ini adalah jaringan *butterfly*, *shuffle-exchange*, *cross-bar* dan *omega*.

Pendekatan lain untuk mereduksi tabrakan memori adalah dengan mengeliminasi seluruh memori bersama dan menyediakan memori lokal untuk setiap prosesor. Jenis komputer paralel dengan memori yang tersebar ini disebut multikomputer [Les93].

Setiap pasangan memori - prosesor ini berlaku seperti halnya komputer sekuensial. Prosesor-prosesor dapat membaca (*read*) dan menulis (*write*) data secara bebas dengan menggunakan lokal memorinya masing-masing. Sebuah prosesor tidak dapat mengakses lokal memori prosesor lain dengan secara langsung, tetapi prosesor tersebut dapat mengirim atau menerima data dari prosesor lain dengan menggunakan jaringan komunikasi *message passing*. Sehingga data dapat disebar dan ditukar sesuai dengan kebutuhan. Skema arsitektur *Message-Passing Multicomputer* dapat dilihat pada gambar .4.



Keterangan :
 P : Prosesor
 M : Memori

Gambar 4 Message-passing Multicomputer

ARSITEKTUR MULTIKOMPUTER

Dalam multikomputer, setiap prosesor mempunyai modul memori untuk menyimpan dan mengambil data selama komputasi. Masing-masing prosesor mempunyai satu atau lebih hubungan langsung ke prosesor lain untuk transmisi data. Jika prosesor tidak mempunyai koneksi langsung ke prosesor lain, komunikasi dapat dilangsungkan melalui prosesor antara (*intermediate processor*) untuk mengirim data. Transmisi data antar prosesor membutuhkan sejumlah waktu yang selanjutnya disebut waktu tunda komunikasi. Jika selama eksekusi program sering terjadi komunikasi antar prosesor, maka jumlah waktu tunda komunikasi akan menambah waktu eksekusi program.

Pada dasarnya waktu komunikasi sebuah message pada multicomputer, terdiri dari tiga komponen, yaitu waktu transmisi (*transmission time*), waktu proses (*processing time*) dan waktu tunggu (*waiting time*). Waktu transmisi adalah waktu yang dibutuhkan untuk transmisi secara fisik sejumlah bit *message* melalui saluran komunikasi. Waktu proses adalah waktu yang diperlukan untuk memproses suatu komputasi. Sedangkan waktu tunggu adalah waktu yang diperlukan untuk menunda pengiriman pesan. Peristiwa penundaan ini disebabkan karena prosesor tersebut sedang menerima *message* dari prosesor lainnya, atau karena prosesor tersebut sedang sibuk. Peristiwa penundaan ini disebut kongesti (*congestion*).

Penundaan *message* dalam perjalanannya melalui prosesor-prosesor akan tergantung pada rata-rata waktu tunda komunikasi yang dibutuhkan untuk transmisi setiap saluran komunikasi dan juga jumlah total saluran yang dilalui antara prosesor sumber dan prosesor tujuan. Jumlah saluran yang dilalui akan tergantung pada struktur keseluruhan jaringan komunikasi, yang disebut topologi dari multikomputer. Salah satu parameter yang penting dari topologi multikomputer ini adalah jumlah saluran yang terhubung pada setiap antar muka prosesor, yang disebut keterhubungan (*connectivity*) topologi. Keterhubungan ini menjadi faktor yang penting untuk menentukan beban jaringan. Parameter penting lainnya adalah *diameter* dari topologi, yaitu jumlah maksimum saluran yang dibutuhkan untuk mengirim suatu *message* pada jarak terjauh prosesor. Diameter menjadi faktor penting kinerja jaringan.

Topologi jaringan dapat bersifat statis ataupun dinamis. Jaringan dinamis diimplementasikan dalam *switched channel* yang konfigurasi akan berubah-ubah, sesuai dengan kebutuhan komunikasi program yang dieksekusi. Jaringan dinamis, misalnya *Busses*, *Crossbar Switces* dan *Multistages Networks*, biasanya dipergunakan dalam multiprosesor.

Dengan mempergunakan saluran berarah yang tetap, jaringan statis dibentuk dari hubungan langsung dari titik ke titik. Hubungan ini tidak akan berubah selama program dieksekusi. Tipe jaringan ini sesuai untuk pembangunan komputer dimana pola komunikasi dapat diperkirakan atau diimplementasikan dengan hubungan statis. Terdapat banyak topologi jaringan komunikasi atau multikomputer. Topologi tersebut dapat dikategorikan menjadi satu dimensi, dua dimensi ataupun tiga dimensi. Misalnya untuk topologi satu dimensi ialah topologi *Line (Linear Array)*, *Ring*, *Star* dan *Tree*. Untuk dua dimensi misalnya topologi *Mesh*, *Torus*, *Illiac Mesh* dan *Systolic Array*. Untuk contoh topologi tiga dimensi adalah topologi *Mesh 3* dimensi dan *Hypercube*.

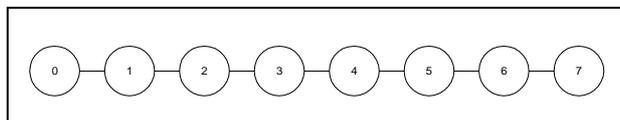
Topologi Linear Array

Linear Array adalah topologi yang paling sederhana. Dalam topologi ini, antar muka komunikasi dihubungkan dalam sebuah garis lurus seperti pada gambar .5. Dalam gambar tersebut, diasumsikan bahwa tiap lingkaran bernomor terdiri dari sebuah prosesor, memori dan antar muka komunikasi. Setiap garis diantara sepasang lingkaran merepresentasikan saluran komunikasi langsung dalam jaringan. Struktur internal yang rinci dari setiap komponen jaringan tidak ditampilkan dalam gambar.

Sewaktu dua prosesor dihubungkan secara langsung dalam topologi ini, kedua prosesor tersebut disebut bertetangga (*adjacent*). Untuk membandingkan karakteristik kinerja relatif dari topologi yang berbeda, diasumsikan waktu tunda komunikasi dasar untuk prosesor yang bertetangga adalah sama untuk setiap topologi. Kinerja relatif hanya tergantung pada *distance* (jarak) antar prosesor. Jarak antara sepasang prosesor pada topologi ini didefinisikan sebagai jumlah saluran komunikasi yang harus dilalui suatu message pada lintasan langsung antar prosesor. *Diameter* topologi didefinisikan sebagai jarak terbesar antara prosesor dalam jaringan. Pada topologi *Linear Array* dengan 8 prosesor, diameternya adalah 7. Secara umum, suatu topologi *Linear Array* dengan n

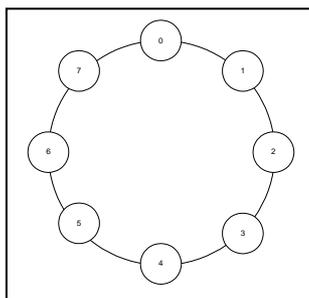
prosesor akan mempunyai keterhubungan (*connectivity*) 2 dan diameter $n-1$. *Distance* diantara 2 prosesor i dan j selalu $\{i-j\}$.

Struktur *Linear Array* yang tidak simetris menyebabkan ketidakefisienan komunikasi ketika n bertambah besar. Untuk n yang kecil, misalnya 2, adalah sangat ekonomis untuk mengimplementasikan *Linear Array*. *Linear Array* tidak sama dengan bus yang membagi waktunya melalui *switching* untuk beberapa prosesor.



Gambar 5 Topologi *Linear Array*

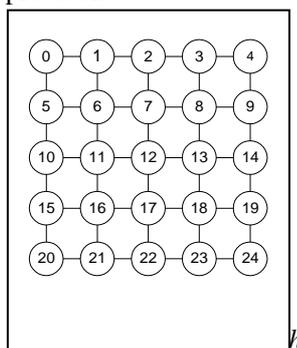
Tanpa menambah beban, kinerja topologi *Linear Array* dapat dengan mudah ditingkatkan dengan menambah sebuah saluran penghubung dari prosesor pertama ke prosesor terakhir. Topologi ini disebut topologi *Ring*, seperti pada gambar .6. Penambahan sambungan ini mereduksi jarak rata-rata antar prosesor dengan faktor 2. Suatu topologi *Ring* dengan n prosesor akan mempunyai diameter $n/2$. Jadi sebuah *message* paling banyak harus melalui $\frac{1}{2}$ jarak *Ring* tersebut. Berdasarkan gambar, pesan dari prosesor 0 ke prosesor 7 harus melalui seluruh saluran penghubung dalam seluruh jaringan. Gambar topologi *Ring* dapat dilihat di bawah ini :



Gambar 6 Topologi Ring

Topologi *Mesh*

Dengan menambah jumlah saluran komunikasi yang terhubung ke setiap prosesor, dimungkinkan untuk mereduksi diameter jaringan dan rata-rata waktu tunda komunikasi. Salah satunya adalah dengan arsitektur *Mesh* dua dimensi, seperti pada gambar. Topologi ini terdiri dari array dua dimensi. Prosesor pada baris i dan kolom j dihubungkan ke empat prosesor tetangga terdekatnya, yaitu prosesor sebelah kiri, kanan, atas dan bawah, dengan lokasi $(i-1,j)$, $(i+1,j)$, $(i,j-1)$, $(i,j+1)$. Semua hubungan antara prosesor bertetangga secara kolom adalah horisontal dan antara prosesor bertetangga secara baris adalah vertikal. Tidak ada hubungan diagonal. Prosesor batas, yaitu prosesor terluar, hanya mempunyai dua atau tiga tetangga terdekat. Gambar di bawah ini adalah contoh topologi *Mesh* dengan 25 prosesor.



Skema penomoran prosesor pada gambar, prosesor dinomori secara berurutan berdasarkan baris, dengan prosesor 0 pada pojok kiri atas. Skema penomoran ini disebut *row-major order*. Perjalanan *message* dari suatu prosesor ke prosesor lain harus melalui lintasan horisontal atau vertikal prosesor antara. Misalnya *message* yang berjalan dari prosesor 6 ke prosesor 13, pertama kali akan berjalan dari prosesor 6 ke prosesor 11, kemudian ke prosesor 12 dan akhirnya ke prosesor 13. Totalnya adalah 3 langkah. Pesan tersebut dapat juga mengambil lintasan alternatif, 6 - 7 - 8 - 13 atau 6 - 7 - 12 - 13. Namun jumlah langkahnya tetap 3. Setiap pasangan prosesor akan mempunyai minimum panjang lintasan (*path-length*) diantara prosesor-prosesor tersebut, diukur dari jumlah jarak baris dan kolom. Untuk setiap topologi $m \times m$ Mesh, diameter jaringan adalah panjang lintasan antara prosesor pada sudut yang berlainan pada Mesh tersebut dan panjang lintasan tersebut selalu $2(m-1)$.

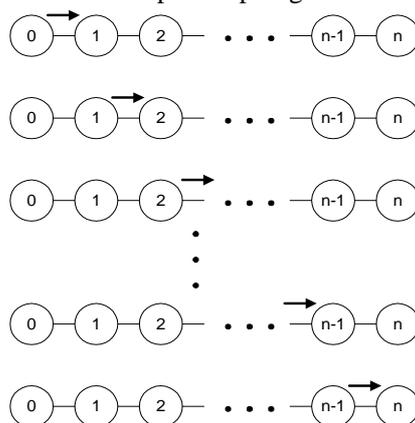
Dalam topologi Mesh, setiap baris dan kolom adalah sama dengan dengan topologi Linear. Seperti pada topologi Linear yang ditingkatkan menjadi topologi Ring, kinerja topologi Mesh dapat ditingkatkan dengan menambah hubungan untuk setiap baris dan kolom. Hal ini akan mengubah setiap baris dan kolom menjadi Ring. Setiap prosesor pada batas kiri dihubungkan ke prosesor-prosesor lain ke batas kanan. Topologi ini disebut topologi Torus. Keterhubungan (*connectivity*) pada topologi Torus selalu 4, selama setiap prosesor sekarang mempunyai tepat 4 saluran penghubung. Maksimum jarak dalam Torus adalah antara sudut dan prosesor tengah. Jadi sebuah $m \times m$ Torus akan mempunyai diameter m .

BROADCASTING DAN AGREGASI

Operasi *Broadcast* dan Agregasi merupakan operasi penting yang sering muncul dalam algoritma paralel. *Broadcast* didefinisikan sebagai suatu operasi penyebaran salinan item data dari prosesor 0 ke seluruh prosesor lain dalam suatu topologi multikomputer. Data mengalir dari prosesor 0 ke prosesor-prosesor tetangga, dimana data tersebut kemudian disalin. Selanjutnya, data dialirkan lagi ke prosesor-prosesor tetangga berikutnya dan seterusnya sampai seluruh prosesor menerima salinan data tersebut [Les93].

Pada gambar di bawah, nilai data pertama kali dikirim dari prosesor 0 ke prosesor 1. Prosesor 1 menyimpan salinan data dan kemudian mengirim salinan data ke prosesor 2 dan seterusnya sampai akhirnya data mencapai prosesor n , yaitu prosesor terujung pada topologi. Diasumsikan waktu tunda komunikasi pada prosesor yang bertetangga adalah T unit waktu. Diasumsikan juga bahwa waktu yang dibutuhkan untuk menyalin data dapat diabaikan jika dibandingkan dengan waktu tunda komunikasi. Maka total waktu eksekusi untuk operasi *broadcast* dengan $n+1$ prosesor adalah nT .

Berikut ini adalah ilustrasi operasi *broadcast* pada topologi Linear Array.

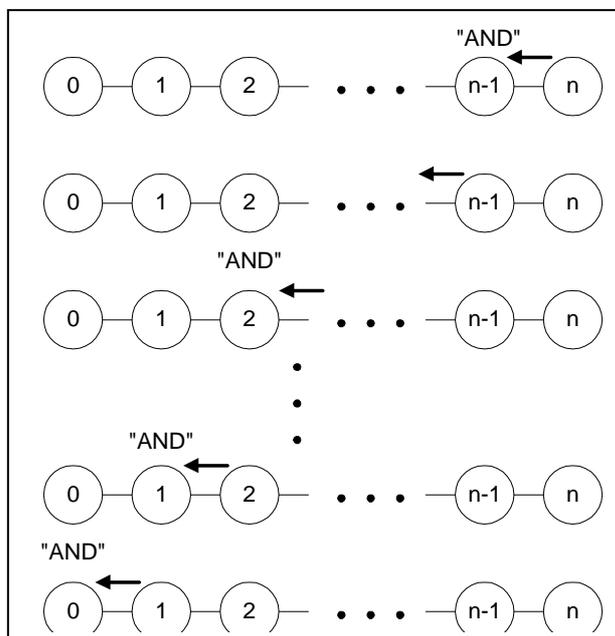


Gambar 8 Broadcasting pada topologi Linear Array

Agregasi adalah pengumpulan item-item data dari setiap prosesor dan menggabungkan item-item data ini menjadi sebuah item data tunggal. Secara berturut-turut, sepasang item data dari sepasang prosesor digabungkan menjadi sebuah item data tunggal [Les93]. Diasumsikan bahwa hasil

operasi penggabungan ini bergantung pada urutan operasi. Contoh operasi penggabungan adalah operasi *AND*, *OR*, *max*, *min*. Pada dasarnya agregasi ini mirip dengan *broadcasting* dan hanya dibedakan oleh aliran data yang berlainan arah.

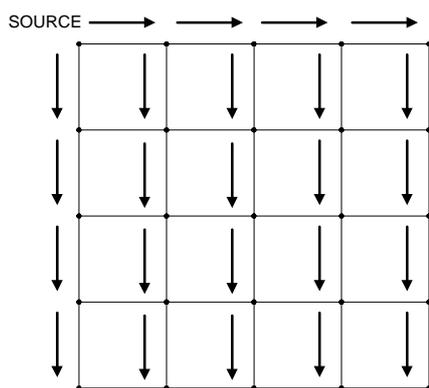
Operasi Agregasi pada topologi *Linear Array* digambarkan di bawah ini.



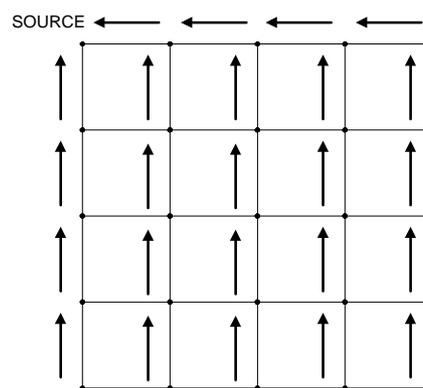
Gambar 9 Agregasi pada topologi *Linear Array*

Setiap prosesor mempunyai nilai data awal. Misalnya data tersebut adalah nilai data Boolean dan operasi penggabungannya adalah logika *AND*. Pada tahap pertama, prosesor n akan mengirim nilai Booleannya ke prosesor $n-1$. Kemudian prosesor $n-1$ akan menggabungkan nilai Boolean tersebut dengan nilai Booleannya sendiri dengan operasi logika *AND*. Nilai Boolean hasil penggabungan ini kemudian dikirim ke prosesor tetangga sebelah kiri. Komunikasi ini dan operasi logika *AND* dilanjutkan pada setiap prosesor secara berturutan sampai hasil akhirnya dicapai prosesor 0. Jika diasumsikan bahwa waktu eksekusi operasi logika *AND* dapat diabaikan dibandingkan dengan waktu komunikasi, maka total waktu eksekusi untuk operasi agregasi ini sama persis dengan waktu eksekusi operasi *broadcast*, yaitu nT .

Pada topologi *Mesh*, *broadcast* dilakukan dengan mengirim nilai data melalui baris teratas dari prosesor-prosesor, dengan demikian data mencapai setiap kolom teratas. Kemudian data mengalir ke bawah pada setiap kolom. Hal ini diilustrasikan pada gambar .10. Panah hitam menunjukkan menggambarkan arah aliran data. Total waktu yang dibutuhkan untuk *broadcast* adalah sebanding dengan panjang lintasan dari sumber prosesor 0 pada sudut kiri teratas *Mesh* ke prosesor sudut kanan terbawah. Dalam $m \times m$ *Mesh*, panjang lintasan ini adalah $2(m-1)$, yang juga merupakan diameter dari jaringan. Dapat dilihat juga pada topologi *Linear Array*, bahwa waktu *broadcast* juga sama dengan diameter topologi *Linear Array*. Hal ini menunjukkan bahwa waktu eksekusi *broadcast* dan agregasi adalah diameter topologinya dikalikan waktu tunda komunikasi. Berdasarkan definisi diameter, hal ini akan menentukan waktu minimum yang dibutuhkan untuk *broadcast* dan agregasi..



Gambar .10 (a) Broadcasting pada Mesh

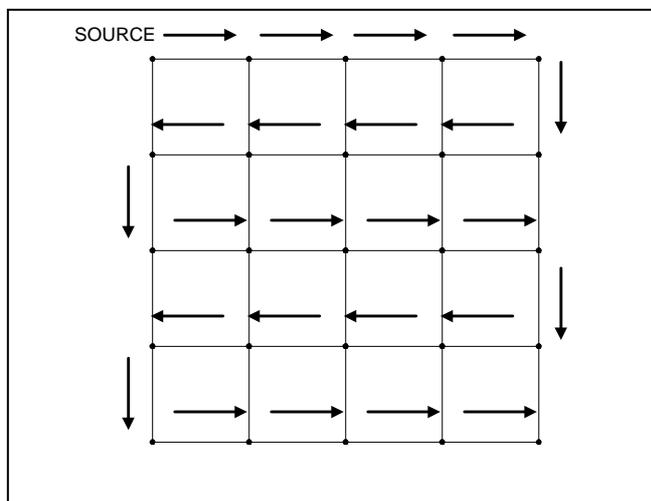


Gambar .10 (b) Agregasi pada Meah

Gambar 10 *Broadcasting* dan *Agregasi* pada topologi *Mesh*

TOPOLOGICAL EMBEDDING

Sewaktu dieksekusi - khususnya untuk sistem multikomputer - setiap algoritma paralel akan mempunyai struktur komunikasi. Jika struktur komunikasi logik yang dipergunakan dalam algoritma sesuai dengan struktur komunikasi fisik topologi multikomputer, maka akan diperoleh kinerja yang baik [Les93]. Pada dasarnya suatu topologi X dapat di-*embedded* menjadi topologi Y, jika terdapat pemetaan yang spesifik dari prosesor X ke prosesor Y. Konsep *embedding* dalam pemrograman multikomputer ini disebut *topological embedding*. Topologi yang akan di-*embedded* harus mempunyai jumlah prosesor yang sama dengan topologi tujuannya. Topologi yang mempunyai keterhubungan yang lebih rendah dapat di-*embedded* ke dalam topologi yang mempunyai keterhubungan yang lebih tinggi [Les93]. Contohnya adalah topologi *line* dapat di-*embedded* ke dalam topologi *mesh*, seperti gambar di bawah ini.



Gambar 11 *Line* di-*embedded* ke dalam *Mesh*

KONSEP PARALELISME

Konsep pemrosesan konkuren lahir sekitar tiga puluh tahun yang lalu untuk memenuhi kebutuhan akan sistem komputer yang memiliki performansi tinggi dengan menggunakan teknologi perangkat keras yang tersedia pada saat itu [KOG81]. Paralelisme (*parallelism*) lahir dari pendekatan yang biasa dipergunakan oleh para perancang sistem untuk menerapkan konsep pemrosesan konkuren. Teknik ini meningkatkan kecepatan proses dengan cara memperbanyak jumlah modul perangkat keras yang dapat beroperasi secara simultan disertai dengan membentuk beberapa proses yang bekerja secara simultan pada modul-modul perangkat keras tersebut. Secara

formal, pemrosesan paralel adalah sebuah bentuk efisien pemrosesan informasi yang penekanan pada eksploitasi dari konkurensi kejadian-kejadian dalam proses komputasi.

Pemrosesan paralel dapat terjadi pada beberapa tingkatan (level) proses. Tingkatan tertinggi pemrosesan paralel terjadi pada proses di antara banyak *job* (pekerjaan) atau pada program yang menggunakan *multiprogramming*, *time sharing*, dan *multiprocessing*. *Multiprogramming* berarti mengoptimalkan operasi-operasi di CPU (*Central Processing Unit*) dan di I/O (*Input/Output device*) diantara beberapa program sekaligus, dan *time sharing* berarti menyediakan pembagian selang waktu yang tetap atau berubah-ubah untuk banyak program. Pemrosesan paralel dapat juga terjadi pada proses di antara prosedur-prosedur atau perintah-perintah (segmen program) pada sebuah program.

PEMROGRAMAN PARALEL

Program komputer untuk komputer sekuensial harus menyediakan sederetan operasi untuk dikerjakan oleh prosesor tunggal. Program komputer untuk komputer paralel harus menyediakan sederetan operasi untuk beberapa prosesor untuk dikerjakan secara paralel, termasuk operasi untuk mengatur dan mengintegrasikan prosesor-prosesor yang terpisah tersebut mengerjakan suatu komputasi yang koheren. Kebutuhan akan pembuatan dan pengaturan berbagai aktivitas komputasi paralel menambah dimensi baru proses dari pemrograman komputer. Algoritma untuk problem yang spesifik harus diformulasikan sedemikian rupa, agar menghasilkan aliran operasi paralel yang kemudian akan dieksekusi di prosesor yang berbeda. Karena itu, meskipun arsitektur multiprosesor dan multikomputer mempunyai potensi yang tinggi untuk meningkatkan kemampuan komputasi, potensi ini akan tercapai melalui pengertian yang baik mengenai bahasa pemrograman paralel dan perancangan algoritma paralel.

Kinerja Program Paralel

Parameter yang sangat penting untuk mengukur kinerja suatu program paralel adalah waktu eksekusi dan *speedup*. Waktu eksekusi dapat diartikan sebagai waktu berlangsungnya (*running*) program paralel pada arsitektur komputer paralel yang dituju. Waktu eksekusi sekuensial didefinisikan sebagai waktu *running* algoritma yang sama yang dieksekusi oleh satu prosesor. *Speedup* dari suatu program paralel adalah waktu eksekusi sekuensial dibagi dengan waktu eksekusi paralel. *Speedup* dapat didefinisikan sebagai berikut :

$$S(n) = \frac{T(1)}{T(n)}$$

dimana : $S(n) = \text{speedup}$,

$T(1)$ = waktu eksekusi operasi pada sistem satu prosesor,

$T(n)$ = waktu eksekusi pada sistem n prosesor.

Dari definisi *speedup* tersebut, lahir nilai *efisiensi*, $E(n)$, untuk sistem dengan n prosesor. Efisiensi didefinisikan sebagai berikut :

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

Dapat dilihat bahwa efisiensi merupakan indikator atas tingkat kinerja *speedup* yang dicapai dibandingkan dengan nilai maksimum yang dapat dicapai. Dari persamaan-persamaan di atas dapat dilihat bahwa $1 \leq S(n) \leq n$, dan $1/n \leq E(n) \leq 1$. Efisiensi terendah terjadi saat seluruh kode program dijalankan secara sekuensial pada satu prosesor, dan efisiensi maksimum dicapai saat seluruh kode program dijalankan pada n prosesor.

Ukuran yang penting lain untuk kinerja prosesor adalah utilitas prosesor. Untuk setiap prosesor, utilitas didefinisikan sebagai prosentase waktu dimana prosesor benar-benar berlangsung selama eksekusi program. Dengan kata lain, utilitas didefinisikan sebagai rata-rata utilitas keseluruhan prosesor.

Hukum Amdahl's

Salah satu hal yang dapat mengurangi *speedup* dari program paralel adalah bagian kode yang dieksekusi secara sekuensial. Diasumsikan terdapat operasi sejumlah n , masing-masing membutuhkan satu unit waktu untuk eksekusi. Diasumsikan juga terdapat fraksi f dari operasi tersebut yang harus dibentuk secara sekuensial (f antara 0 -1). Maka, sejumlah fn operasi harus dilaksanakan secara sekuensial dan sisanya, $(1-f)n$, operasi dapat diparalelkan. Jika program tersebut dikerjakan pada komputer paralel dengan p prosesor, maka waktu total eksekusi minimum adalah $fn+(1-f)n/p$. Sedangkan waktu total eksekusi program pada komputer sekuensial adalah n . Dengan membagi waktu sekuensial dengan waktu paralel minimum, didapatkan *speedup* maksimum yang akan diperoleh komputer paralel dengan p prosesor sebagai berikut :

$$\text{Maksimum speedup} = \frac{1}{f + \frac{1-f}{p}}$$

Rumus di atas adalah rumus untuk Hukum Amdahl's. Jika p mendekati tak terbatas, maka nilai rumus di atas akan mendekati $1/f$. Hal ini menunjukkan bahwa bagian kode sekuensial dalam program paralel dapat membatasi *speedup* maksimum yang dapat diperoleh [Les93]. Dengan hukum Amdahl's dapat diketahui bahwa untuk mempergunakan mesin paralel secara efektif, bagian kode sekuensial harus dapat dijaga sekecil mungkin. Sebagai contoh, untuk memperoleh *speedup* 100, kode sekuensial harus lebih kecil dari 1 persen dari total kode.

Hukum Amdahl's di atas menjadi argumen untuk menentang kemampuan dari pemrosesan paralel [Qui87]. Terdapat beberapa program yang tidak akan efisien jika dieksekusi secara paralel. Walaupun demikian, berdasarkan penelitian, terdapat banyak problem komputasi yang dapat memperoleh *speedup* yang baik, mencapai 100 bahkan lebih.

PEMBANGUNAN ALGORITMA PARALEL

Dalam beberapa kasus, algoritma sekuensial dengan mudah dapat diadaptasi ke dalam lingkungan paralel. Namun dalam kebanyakan kasus, problem komputasi harus dianalisa ulang dan menghasilkan algoritma paralel yang baru. Terdapat beberapa penelitian mengenai perancangan algoritma paralel untuk problem-problem praktis seperti pengurutan, pemrosesan graf, solusi untuk persamaan linier, solusi untuk persamaan diferensial, dan untuk simulasi. Teknik pembangunan algoritma paralel dapat dibedakan sebagai berikut :

Paralelisme Data

Teknik paralelisme data merupakan teknik yang paling banyak digunakan dalam program paralel. Teknik ini lahir dari penelitian bahwa aplikasi utama komputasi paralel adalah dalam bidang sains dan engineer, yang umumnya melibatkan array multi-dimensi yang sangat besar. Dalam program sekuensial biasa, array ini dimanipulasi dengan mempergunakan perulangan bersarang untuk mendapatkan hasil. Kebanyakan program paralel dibentuk dengan mengatur ulang algoritma sekuensial agar perulangan bersarang tersebut dapat dilaksanakan secara paralel. Paralelisme data menunjukkan bahwa basis data dipergunakan sebagai dasar untuk membentuk aktifitas paralel, dimana bagian yang berbeda dari basis data akan diproses secara paralel. Dengan kata lain paralelisme dalam program ini dibentuk dari penerapan operasi-operasi yang sama ke bagian array data yang berbeda. Prinsip paralelisme data ini berlaku untuk pemrograman multiprosesor dan multikomputer.

Partisi Data

Merupakan teknik khusus dari Paralelisme Data, dimana data disebar ke dalam memori-memori lokal multikomputer. Sebuah proses paralel kemudian ditugaskan untuk mengoperasikan masing-masing bagian data. Proses tersebut harus terdapat dalam lokal memori yang sama dengan bagian data, karena itu proses dapat mengakses data tersebut secara lokal. Untuk memperoleh kinerja yang baik, setiap proses harus memperhatikan variabel-variabel dan data-data lokalnya masing-masing. Jika suatu proses membutuhkan akses data yang terdapat dalam *remote* memori, maka hal ini dapat dilakukan melalui jaringan *message passing* yang menghubungkan prosesor-prosesor. Karena

komunikasi antar prosesor ini menyebabkan terjadinya waktu tunda, maka *message passing* ini sebaiknya dilakukan dalam frekuensi yang relatif kecil.

Dapat disimpulkan bahwa tujuan dari partisi data adalah untuk mereduksi waktu tunda yang diakibatkan komunikasi *message passing* antar prosesor. Algoritma paralel mengatur agar setiap proses dapat melakukan komputasi dengan lokal data masing-masing.

Algoritma Relaksasi

Pada algoritma ini, setiap proses tidak membutuhkan sinkronisasi dan komunikasi antar proses. Meskipun prosesor mengakses data yang sama, setiap prosesor dapat melakukan komputasi sendiri tanpa tergantung pada data antara yang dihasilkan oleh proses lain. Contoh algoritma relaksasi adalah algoritma perkalian matrik, pengurutan dengan menggunakan metode *ranksort* dan lain sebagainya.

Paralelisme Sinkron

Aplikasi praktis dari komputasi paralel adalah untuk problem yang melibatkan array multi-dimensi yang sangat besar. Problem tersebut mempunyai peluang yang baik untuk paralelisme data karena elemen yang berbeda dalam array dapat diproses secara paralel. Teknik komputasi numerik pada array ini biasanya iteratif, dan setiap iterasi akan mempengaruhi iterasi berikutnya untuk menuju solusi akhir. Misalnya saja untuk solusi persamaan numerik pada sistem yang besar. Umumnya, setiap iterasi mempergunakan data yang dihasilkan oleh iterasi sebelumnya dan program akhirnya menuju suatu solusi dengan akurasi yang dibutuhkan.

Algoritma iterasi ini mempunyai peluang paralelisme pada setiap iterasinya. Beberapa proses paralel dapat dibentuk untuk bekerja pada array bagian yang berbeda. Namun setelah setiap iterasi, proses-proses harus disinkronkan, karena hasil yang diproduksi oleh satu proses dipergunakan oleh proses-proses lain pada iterasi berikutnya. Teknik pemrograman paralel seperti ini disebut paralelisme sinkron. Contohnya adalah perhitungan numerik pada Metode Eliminasi Gauss.

Pada paralelisme sinkron ini, struktur umum programnya biasanya terdiri dari perulangan **FORALL** yang akan membentuk sejumlah besar proses paralel untuk dioperasikan pada bagian array data yang berbeda. Setiap proses diulang dan disinkronkan pada akhir iterasi. Tujuan dari sinkronisasi ini adalah untuk meyakinkan bahwa seluruh proses telah menyelesaikan iterasi yang sedang berlangsung, sebelum terdapat suatu proses yang memulai iterasi berikutnya.

Komputasi Pipeline

Pada komputasi pipeline, data dialirkan melalui seluruh struktur proses, dimana masing-masing proses membentuk tahap-tahap tertentu dari keseluruhan komputasi. Algoritma ini dapat berjalan dengan baik pada multikomputer, karena adanya aliran data dan tidak banyak memerlukan akses ke data bersama. Contoh komputasi pipeline adalah teknik penyulihan mundur yang merupakan bagian dari metoda Eliminasi.

Dalam merancang suatu algoritma paralel kita harus mempertimbangkan pula hal-hal yang dapat mengurangi kinerja program paralel. Hal-hal tersebut adalah :

1. Memory Contention

Eksekusi prosesor tertunda ketika prosesor menunggu hak akses ke sel memori yang sedang dipergunakan oleh prosesor lain. Problem ini muncul pada arsitektur multiprosesor dengan memori bersama.

2. Excessive Sequential Code

Pada beberapa algoritma paralel, akan terdapat kode sekuensial murni dimana tipe tertentu dari operasi pusat dibentuk, seperti misalnya pada proses inialisasi. Dalam beberapa algoritma, kode sekuensial ini dapat mengurangi keseluruhan *speedup* yang dapat dicapai.

Process Creation Time

Pembentukan proses paralel akan membutuhkan waktu eksekusi. Jika proses yang dibentuk relative berjalan dalam waktu yang relatif lebih singkat dibandingkan dengan waktu pembentukan proses itu sendiri, maka overhead pembuatan akan lebih besar dibandingkan dengan waktu eksekusi paralel algoritma.

Communication Delay

Overhead ini muncul hanya pada multikomputer. Hal ini disebabkan karena interaksi antar prosesor melalui jaringan komunikasi. Dalam beberapa kasus, komunikasi antar dua prosesor mungkin melibatkan beberapa prosesor antara dalam jaringan komunikasi. Jumlah waktu tunda komunikasi dapat menurunkan kinerja beberapa algoritma paralel.

Synchronization Delay

Ketika proses paralel disinkronkan, berarti bahwa suatu proses akan harus menunggu proses lainnya. Dalam beberapa program paralel, jumlah waktu tunda ini dapat menyebabkan *bottleneck* dan mengurangi *speedup* keseluruhan.

Load Imbalance

Dalam beberapa program paralel, tugas komputasi dibangun secara dinamis dan tidak dapat diperkirakan sebelumnya. Karena itu harus selalu ditugaskan ke prosesor-prosesor sejalan dengan pembangunan tugas tersebut. Hal ini dapat menyebabkan suatu prosesor tidak bekerja (*idle*), sementara prosesor lainnya tidak dapat mengerjakan task yang ditugaskannya.

KONSEP PROSES

Untuk membangun suatu algoritma program paralel, pengertian mengenai konsep proses merupakan alat bantu yang sangat berguna. Proses sendiri diartikan sebagai sederatan operasi yang dapat dibentuk oleh sebuah prosesor tunggal [Les93]. Proses dapat dipergunakan sebagai blok pembangunan dasar dari pemrograman paralel, setiap prosesor mengeksekusi proses tertentu pada saat tertentu. Secara informal, proses dapat berupa sub rutin atau prosedur yang dieksekusi oleh prosesor tertentu. Ketersediaan sejumlah besar prosesor secara fisik berarti bahwa sejumlah besar proses dapat dieksekusi secara paralel oleh perangkat keras komputer. Dengan asumsi bahwa setiap aktifitas dari setiap proses mempunyai kontribusi terhadap keseluruhan komputasi, maka eksekusi komputasi tersebut akan lebih cepat dibandingkan jika dilakukan dengan prosesor tunggal. Agar konsep proses berguna dalam pembuatan program, konsep proses ini menjadi feature tambahan dalam bahasa pemrograman paralel.

Untuk menggabungkan konsep proses ke dalam bahasa pemrograman paralel, harus terdapat beberapa mekanisme dalam bahasa tersebut untuk mendefinisikan dan membentuk proses baru. Harus terdapat pula beberapa feature dalam bahasa tersebut untuk membagi (*sharing*) data antar prosesor paralel, dengan demikian setiap proses dapat berinteraksi satu sama lainnya untuk menyelesaikan keseluruhan komputasi. Setiap proses dapat mengeksekusi dengan kecepatan yang berbeda pada prosesor yang berbeda.

Status Proses

Pada dasarnya terdapat lima kemungkinan status untuk proses yang sedang aktif, yaitu :

- *ready*, proses dikatakan dalam status *ready* jika proses tersebut memungkinkan untuk dieksekusi tetapi pada saat tersebut tidak dieksekusi oleh prosesor yang telah ditentukan. Pada saat pertama kali proses dibentuk, proses tersebut mempunyai status *ready*.
- *running*, tiap satuan waktu hanya terdapat satu proses yang dapat berjalan pada satu prosesor.
- *blocked*, suatu proses yang dalam status *ready* harus menunggu event dari proses lainnya, misalnya menunggu data hasil proses lain, maka proses tersebut berubah status menjadi *blocked*. Jika semua proses dalam status *blocked*, maka kondisi ini disebut *deadlock*. Hal ini menyebabkan eksekusi program tidak dapat diteruskan.
- *delayed*, status *delayed* pada dasarnya hampir sama dengan status *blocked*, kecuali bahwa event yang ditunggu sebenarnya telah muncul. Karena waktu tunda komunikasi dalam arsitektur komputer, maka event tersebut belum tiba pada proses yang sedang *blocked* tersebut. Status *delayed* hanya akan muncul pada arsitektur multikomputer
- *spinning*, status proses *spinning* hanya akan muncul pada arsitektur multiprosesor Status ini muncul pada saat proses tersebut harus menunda eksekusinya karena diharuskan menunggu proses lain selesai dieksekusi. Teknik menunda proses tersebut dengan menggunakan operasi *lock*, menunggu sampai proses lain melakukan operasi *unlock*. Teknik ini disebut *busy-waiting*.

Pada dasarnya status ini sama dengan *blocked*, perbedaannya terletak pada teknik penundaan proses yang dipergunakan.

Pembentukan Proses

Aktivitas komputasi dimulai ketika suatu proses ditugaskan ke suatu prosesor dalam komputer paralel. Proses dapat digambarkan sebagai suatu kumpulan kode program yang ditugaskan ke suatu prosesor tertentu untuk dieksekusi.

Dalam program paralel, eksekusi program dimulai sama seperti halnya program sekuensial biasa. Program paralel memerlukan suatu statement untuk pembentukan proses. Statement tersebut menyebabkan program membentuk proses baru untuk ditugaskan pada prosesor lain, yang kemudian mengeksekusi proses tersebut. Gambaran umum aktivitas paralel adalah sebagai berikut : proses yang sedang berjalan pada prosesor mengeksekusi statemen pembentukan proses, proses yang dibentuk disebut *child process* (proses anak), sedangkan pembentuk proses disebut *parent process* (proses induk).

Program di bawah ini merupakan program sekuensial untuk mendapatkan akar dari elemen-elemen array [Les93]. Pada contoh program tersebut terdapat satu perulangan FOR. Perulangan ini menunjukkan potensi paralelisme program. Modifikasi yang dilakukan adalah dengan mengubah statement FOR dengan statement FORALL untuk membentuk proses anak. Statemen FORALL ini merupakan statement yang terdapat pada sistem Multi-Pascal.

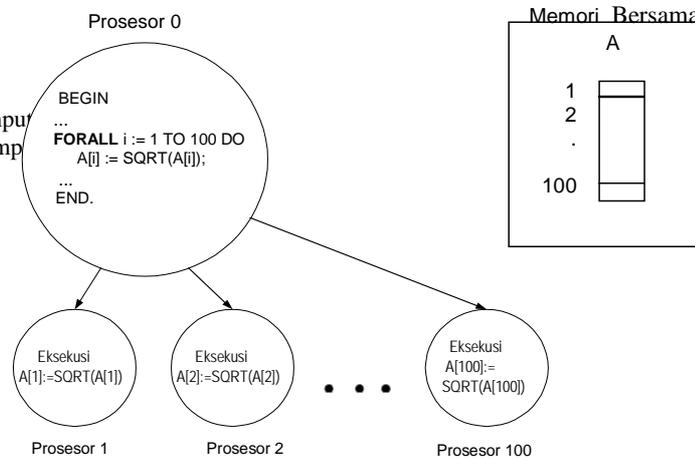
```
PROGRAM SQRT;  
VAR A: ARRAY [1..100] OF REAL;  
    I : INTEGER;  
BEGIN  
    ...  
    FOR i := 1 TO 100 DO  
        A[i] := SQRT([i]);  
    END.
```

Hasil modifikasi statement FOR menjadi FORALL dapat dilihat pada program di bawah ini :

```
PROGRAM SQRTParalel;  
VAR A: ARRAY [1..100] OF REAL;  
    I : INTEGER;  
BEGIN  
    ...  
    FORALL i := 1 TO 100 DO  
        A[i] := SQRT([i]);  
    END.
```

Pada awal program utama, proses induk mengeksekusi statemen FORALL, dalam keadaan demikian status induk proses adalah *running*. Statemen FORALL pada program di atas membentuk 100 salinan statemen $A[i]:=SQRT([i])$ dan setiap satu proses paralel mempunyai variabel indeks i . Ke- 100 proses anak tersebut dieksekusi pada prosesor yang berbeda dan berlangsung secara paralel.

Sewaktu proses induk menunda eksekusinya hingga seluruh proses anak selesai dieksekusi, proses induk tersebut dalam status *blocked*. Sedangkan seluruh proses anak yang sedang dieksekusi mempunyai status *running*. Setelah seluruh Proses anak selesai dieksekusi, proses induk berstatus *running* hingga terminasi. Pembentukan proses ini digambarkan sebagai berikut [Les93] :



Gambar 12 Pembentukan Proses

Pada contoh di atas, array A disimpan ke dalam memori bersama yang mudah diakses oleh prosesor-prosesor. Setiap prosesor akan bekerja dengan bagian elemen array, seluruhnya berlangsung secara paralel.

Hal yang penting pada contoh di atas adalah mengenai variabel indeks i . Pada perulangan FOR sekuensial, indeks i adalah variabel integer tunggal yang bernilai 1 pada iterasi pertama, kemudian bernilai 2 pada iterasi kedua dan seterusnya. Pada saat iterasi dimulai, indeks variabel i secara otomatis berubah menjadi nilai berikutnya dan dapat dipergunakan dalam perulangan (*loop*). Hal ini merupakan karakter standar perulangan FOR dalam Perangkat Lunak Pascal.

Pada perulangan FOR paralel, indeks variabel i - seperti pada statemen FOR - dideklarasikan sebagai variabel INTEGER di awal program. Setelah itu, variabel i akan menjadi variabel lokal tunggal yang disimpan di memori bersama. Jika iterasi akan dieksekusi secara paralel, maka satu indeks variabel tunggal i tidak akan mencukupi. Semua nilai i dari 1 sampai 100 harus diperoleh secara simultan. Hal ini ditangani oleh perangkat lunak secara otomatis, dengan jaminan setiap prosesor mempunyai salinan lokal indeks i sendiri. Jadi prosesor 1 mempunyai variabel lokal i dengan nilai 1, prosesor 2 mempunyai nilai $i = 2$ dan seterusnya.

Granularitas Proses

Hal penting yang perlu dipertimbangkan dalam pembentukan proses adalah granularitas proses, yaitu waktu berlangsungnya atau waktu eksekusi setiap proses. Dari contoh di atas, ketika statemen FORALL dieksekusi di prosesor 0, maka aktifitas komputasi yang terjadi di prosesor 0 adalah sebagai berikut :

Buat Proses Anak dan diberikan ke prosesor 1
 Buat Proses Anak dan diberikan ke prosesor 2
 .
 .
 .
 Buat Proses Anak dan diberikan ke prosesor 100

Jika waktu pembentukan proses anak adalah 10 unit waktu, maka statemen FORALL akan membutuhkan $100 \cdot 10 = 1000$ unit waktu untuk membentuk seluruh proses anak. Jika statemen penugasan $A[i] := \text{SQRT}(A[i])$ membutuhkan waktu eksekusi 10 unit waktu, maka keseluruhan waktu yang statement FORALL adalah $1000 + 10 = 1010$ unit waktu. (Proses Induk yang berjalan pada prosesor 0 membutuhkan 1000 unit waktu untuk membuat 100 Proses Anak, dan keseluruhan proses anak ini akan berjalan secara paralel dalam waktu 10 unit waktu).

Jika menggunakan FOR secara sekuensial, maka seluruh perulangan akan dilaksanakan di prosesor 0. Misalkan statemen penugasan $A[i] := \text{SQRT}(A[i])$ membutuhkan waktu eksekusi 10 unit waktu, maka waktu keseluruhan yang dibutuhkan oleh statemen FOR adalah $100 \cdot 10 = 1000$ unit waktu. Dibandingkan dengan menggunakan statemen FORALL yang membutuhkan waktu sebesar 1010 unit waktu, maka eksekusi secara sekuensial menghasilkan waktu yang lebih baik. Pada contoh di atas, terjadi kegagalan *speedup*.

Jika waktu untuk pembentukan proses anak adalah tetap 10 unit waktu dan setiap proses anak mempunyai granularitas yang besar, misalnya 10000 unit waktu, maka total waktu eksekusi secara paralel adalah 11000. 1000 unit waktu untuk pembentukan 100 proses anak dan 10000 unit waktu untuk eksekusi proses. Dibandingkan dengan eksekusi yang dilakukan secara sekuensial yang akan membutuhkan $100 \cdot 10000 = 1000000$ unit waktu, maka *speedup* yang diperoleh adalah $1000000 / 11000 = 91$.

Untuk meningkatkan efisiensi, granularitas dapat ditingkatkan dengan cara mengelompokkan beberapa nilai indeks ke dalam proses yang sama. Pada contoh algoritma di atas, misalkan dibentuk 10 proses dengan masing-masing 10 nilai indeks variabel i . Contohnya seperti di bawah ini :

```
PROGRAM SQRTParallelGroup;  
VAR  A: ARRAY [1..100] OF REAL;  
     I : INTEGER;  
  
BEGIN  
    FORALL I := 1 TO 10 GROUPING 10 DO  
        A[I] := SQRT([I]);  
    END.
```

Pada contoh di atas, prosesor 0 hanya membentuk 10 proses. Masing-masing proses secara sekuensial melakukan pengulangan 10 kali. Proses 1 melakukan iterasi dengan nilai indeks 1 sampai 10, proses 2 melakukan iterasi nilai indeks 11 sampai 20 dan seterusnya. Jika ukuran kelompok tidak genap dibagi dengan nilai indeks, maka proses anak terakhir akan mempunyai nilai indeks yang lebih kecil daripada nilai indeks ukuran .

Seperti halnya statemen FOR, statemen FORALL dapat dipergunakan dalam perulangan bersarang. Contohnya adalah program penambahan matrik berikut ini :

```
PROGRAM TambahMatrik;  
VAR  i, j : INTEGER;  
     A,B,C : ARRAY [1..20,1..30] OF REAL;  
  
BEGIN  
    FORALL i := 1 to 20 DO  
        FORALL j:= 1 to 30 DO  
            C[i,j] := A[i,j] + B[i,j];  
        END.  
    END.
```

Contoh program di atas menunjukkan 600 proses telah dibentuk. Pembentukan ini terdiri dari dua tahap. Tahap pertama - pada statemen FORALL pertama dengan nilai indeks i dari 1 sampai 20 -, menghasilkan 20 proses anak, masing-masing untuk nilai indeks i . Setiap proses anak ini terdiri dari statemen FORALL kedua. Ketika setiap proses anak generasi pertama ini dieksekusi pada prosesor yang telah ditugaskan, ke-20 proses tersebut masing-masing akan membentuk 30 proses anak, sesuai dengan nilai indeks j . Jadi total proses anak yang dibentuk adalah 600 proses. Masing-masing proses dibentuk untuk setiap elemen dari Matrik.

Pembuatan proses secara bertingkat ini akan mereduksi keseluruhan waktu pembentukan proses. Hal ini disebabkan generasi pertama dari proses anak akan dieksekusi secara paralel pada prosesor yang berbeda. Dibandingkan dengan hanya satu prosesor membentuk seluruh proses, maka akan terdapat 20 prosesor yang membentuk proses anak baru secara paralel. Misalkan terdapat dua perulangan FORALL bersarang dengan perulangan terluar mempunyai nilai indeks n dan perulangan terdalam mempunyai nilai indeks m . Jika waktu untuk membentuk satu proses adalah C , maka total waktu untuk membentuk semua nm proses adalah $C(n+m)$. Jika seluruh proses dibentuk oleh satu proses induk, maka waktu total pembentukan adalah Cnm .

Secara umum aktifitas yang terjadi pada saat statement FORALL tersebut dieksekusi adalah sebagai berikut :

```
Buat sejumlah n proses anak (dengan statemen FORALL),  
Tunggu sampai seluruh n proses anak tersebut selesai dieksekusi,  
Lanjutkan eksekusi setelah FORALL
```

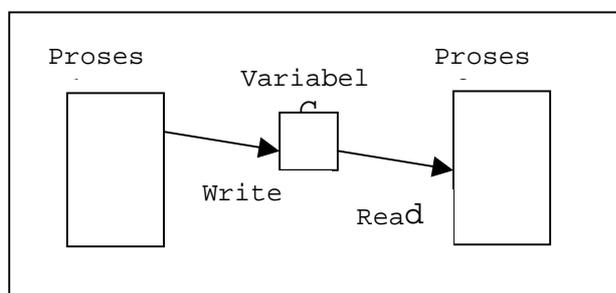
Dapat dilihat bahwa ketika proses anak sedang dieksekusi, maka proses induk akan menunda eksekusinya – berstatus *blocked* - sampai seluruh proses anak selesai dieksekusi. Untuk beberapa

algoritma tertentu, akan lebih efisien jika seluruh proses anak *running* secara paralel dengan proses induk. Konsekuensinya, meskipun proses induk telah mencapai terminasi program sewaktu proses anak sedang berjalan, proses induk tidak diijinkan untuk berhenti sampai seluruh proses anaknya selesai dieksekusi. Dengan kata lain diperlukan featuring untuk menahan proses induk mencapai terminasi program utama agar seluruh proses anak dapat dieksekusi dengan benar. Featuring ini terdapat dalam beberapa beberapa perangkat lunak paralel.

PROSES KOMUNIKASI

Pada algoritma relaksasi, setiap proses paralel berjalan secara independen tanpa interaksi dengan proses-proses lain. Dalam algoritma relaksasi paralel, tidak ada proses yang menulis nilai yang kemudian digunakan oleh proses lain. Algoritma ini sederhana dan menghasilkan *speedup* yang baik sewaktu diterapkan pada berbagai arsitektur komputer paralel. Untuk algoritma yang tidak termasuk algoritma relaksasi, diperlukan mekanisme tertentu dalam menangani proses komunikasi yang terjadi. Misalkan terdapat 2 proses pada multiprosesor, P1 dan P2. Dalam eksekusinya P1 menghasilkan suatu nilai dan menulis nilai tersebut ke variabel bersama C yang kemudian dipergunakan oleh P2 untuk komputasi berikutnya, seperti pada gambar .13.

Selama P1 dan P2 merupakan proses paralel, tidak ada jaminan bahwa P1 akan menulis nilai ke variabel C sebelum P2 membacanya. Jika P1 dibiarkan menyelesaikan eksekusinya dan kemudian P2 memulai eksekusi, berarti paralelisme proses tidak diperoleh. Tipe komunikasi proses ini sering muncul dalam berbagai program paralel, satu proses melakukan komputasi terhadap beberapa nilai yang akan dipergunakan oleh proses paralel lainnya. Untuk menangani masalah ini, diperlukan suatu kanal (*channel*) yang mempunyai properti 'kosong'. Ketika suatu proses membaca kanal yang kosong, maka eksekusi proses secara otomatis akan tertunda sampai terdapat proses lain yang menulis suatu nilai ke dalam kanal tersebut. Untuk contoh di atas, dengan membuat variabel C menjadi variabel kanal yang diinisiasi 'kosong', proses P2 akan menunggu pembacaan variabel C sebelum proses P1 selesai menuliskan suatu nilai ke dalam variabel C. Dengan demikian komunikasi proses dijamin berlangsung dengan benar.



Gambar 13 Komunikasi diantara Proses-proses Paralel

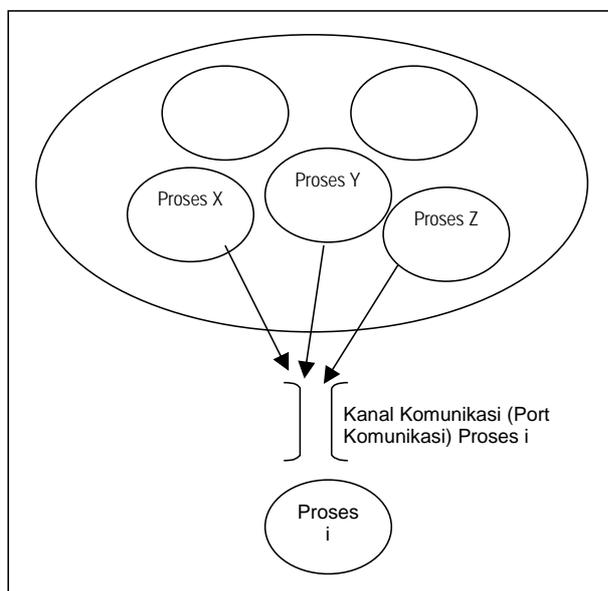
Variabel kanal ini akan sangat berguna untuk komputasi pipeline, dimana data hasil komputasi proses dikirimkan ke proses berikutnya melalui variabel kanal.

Terdapat perbedaan yang sangat mendasar pada pemodelan perangkat lunak konseptual untuk multiprosesor dan multikomputer. Hal ini dikarenakan adanya perbedaan dalam hal variabel bersama, yang dapat diakses oleh seluruh prosesor. Pada multikomputer, yang tidak mengenal variabel bersama, tiap prosesor mempunyai memori lokal sendiri untuk mengakses variabel lokalnya sendiri. Prosesor tersebut terhubung secara fisik melalui jaringan, yang dapat mengirim blok data dari suatu proses ke proses lain.

Secara fisik, prosesor dapat berkomunikasi dengan prosesor lain dengan mengirimkan message melalui sambungan saluran penghubung (*connection link*). Jika proses merupakan abstraksi perangkat lunak dari prosesor fisik, maka abstraksi perangkat lunak untuk saluran komunikasi dapat berupa kanal komunikasi antara proses-proses. Dengan kata lain, proses dapat berkomunikasi dengan proses lain dengan mengirimkan *message* melalui variabel kanal.

Untuk menyesuaikan variabel kanal dalam program multikomputer, fungsi dan semantiknya harus diubah. Pada multiprosesor, variabel kanal serupa dengan variabel bersama yang dapat

diakses oleh semua prosesor. Variabel ini tidak seperti saluran komunikasi dari suatu multikomputer, yang secara fisik menghubungkan dua prosesor yang spesifik. Untuk membuat variabel kanal dapat mencerminkan sifat saluran komunikasi, ujung penerima (*receiving end*) dari tiap kanal haruslah dihubungkan ke suatu proses yang spesifik seperti gambar .14.



Gambar 14 Kanal Komunikasi

Karakteristik penting pada komunikasi adalah adanya jaminan bahwa pesan yang dikirim oleh suatu sumber tertentu ke tujuan yang sama, akan datang sesuai dengan urutan pengirimannya. Variabel kanal yang ditugaskan pada proses spesifik dalam multikomputer akan memiliki sifat yang sedikit berbeda dengan variabel kanal biasa pada multiprosesor. Untuk itu membedakannya, variabel kanal tersebut disebut port komunikasi. Pembentukan port komunikasi ini dapat dilaksanakan bersamaan dengan pembentukan proses anak. Port komunikasi hanya dapat dibaca oleh satu proses yang spesifik.

Pada operasi penulisan ke port terdapat waktu tunda sebelum message sampai ke kanal tujuan. Besar waktu tunda tergantung dari topologi arsitektur multikomputer. Waktu tunda yang terjadi pada penulisan port memiliki implikasi penting yang mempengaruhi kinerja program. Secara umum, aktifitas proses induk pada program multikomputer dapat digambarkan sebagai berikut :

interaksi dengan user untuk mendapatkan data awal, baik dari disk maupun dari piranti masukan, membentuk proses-proses anak yang akan membangun komputasi paralel, mengumpulkan hasil komputasi proses anak dan melaporkannya ke user atau menuliskan ke disk.

Dari aktifitas di atas, terlihat perlu adanya komunikasi antara proses induk dengan proses anak. Komunikasi proses induk yang berjalan dua arah ini, yaitu kirim (*send*) dan terima (*receive*) pesan, dilakukan dengan menggunakan port komunikasi. Pengiriman hasil komputasi dari proses anak ke proses induk dapat juga dilakukan melalui parameter-parameter proses anak. Parameter yang dipergunakan dapat berupa parameter value dan parameter VAR. Parameter value digunakan untuk memberikan data awal ke proses anak, sedangkan parameter VAR digunakan untuk mengumpulkan data hasil komputasi proses anak. Parameter VAR di sini merupakan *remote var* yang mempunyai dua karakteristik penting, yaitu bahwa parameter ini hanya dapat ditulis dan tidak dapat dibaca. Proses penulisan dilakukan di prosesor anak, sedangkan yang ditulis sebenarnya adalah alamat parameter yang ada di *remote* prosesor.

Untuk topologi multikomputer yang mempertimbangkan waktu komunikasi antar prosesor, lokasi relatif dari prosesor akan sangat mempengaruhi waktu tunda komunikasi tersebut. Prosesor yang

jauh letaknya akan mempunyai waktu tunda komunikasi lebih lama dibandingkan dengan prosesor yang dekat. Untuk itu perlu dipertimbangkan pengalokasian prosesor untuk proses tertentu secara eksplisit oleh pemrogram.

PENUTUP

Pemrosesan paralel diartikan sebagai proses yang dilakukan terhadap data di dalam beberapa pemroses. Dalam hal ini beberapa instruksi dapat langsung mengakses data di dalam pemroses tersebut secara paralel dan menjaga setiap pemroses tetap bekerja. Langkah pertama dalam membangun pemrograman paralel adalah eksplorasi algoritma sekuensial untuk digali potensi paralelisme dalam algoritma tersebut. Untuk mendapatkan algoritma paralel, pertimbangkan juga aspek arsitektur apakah menggunakan arsitektur *multiprocessor* dan *multicomputer*. Arsitektur *multicomputer* melibatkan teknik yang lebih kompleks dibandingkan pada *multiprocessor*. Algoritma paralel dapat diimplementasikan pada beberapa topologi arsitektur *multicomputer*, yaitu *linear array* dilengkapi dengan topologi *ring*, *mesh connected* dilengkapi dengan *torus* dan *fullyconnected*. Dari topologi-topologi tersebut, dilakukan perbandingan kinerja keparalelan, berupa kompleksitas waktu, percepatan pemrosesan (*speedup*), efisiensi dan utilitas masing-masing prosesor.

Implementasi algoritma paralel dapat dilakukan pada mesin simulator, seperti Multi-Pascal [Les93]. Simulator Multi-Pascal sebagai perangkat lunak implementasi algoritma paralel dapat dikatakan memadai sebagai alat bantu untuk mempelajari paralelisme yang ditinjau dari aspek algoritmanya. Simulator ini dengan mudah dapat mengimplementasikan semua algoritma yang sudah dibangun, meski terdapat berbagai keterbatasan dalam beberapa algoritma tertentu. *Debugging tools* yang dimiliki oleh Multi-Pascal dapat dimanfaatkan untuk mempelajari bagaimana proses paralel itu sendiri dapat berlangsung. Selain itu *debugging tools* ini berhasil digunakan untuk mendapatkan kinerja program paralel yang diambil berupa waktu eksekusi sekuensial, waktu eksekusi paralel, speed-up, granularitas proses anak dan waktu pembentukan proses anak.

Selain dengan simulator, algoritma paralel yang sudah didapat, dapat diimplementasikan di perangkat lunak paralel lainnya, misalnya PVM (*Parallel Virtual Machine*) yang mempunyai keunggulan *multi-platform* atau MPI (*Message Passing Interface*), yang sama-sama berarsitektur *message passing*, dimana platform yang dipergunakan mencerminkan arsitektur komputer paralel yang sesungguhnya, sehingga perlu dipertimbangkan masalah beban jaringan platform, message passing dan waktu komunikasinya dan lain-lain.

Masih banyak hal-hal yang dapat digali dalam mempelajari paralelisme dalam aspek algoritma. Studi paralelisme merupakan studi yang sangat menarik dan dapat dikaitkan dengan sub bidang keahlian lainnya seperti jaringan syaraf tiruan, pengolahan citra dan lain-lain. Penulis berencana akan menyusun tulisan mengenai bahasan tersendiri mengenai Simulator Multi-Pascal dan studi kasus adaptasi algoritma paralel, mulai dari penggalian potensi paralelisme, adaptasi menjadi algoritma paralel, strategi implementasi dan analisis paralelisme itu sendiri.

DAFTAR PUSTAKA

1. [Ber89] Bertsekas, D.P. and Tsitsklis, J. N., *Parallel and Distributed Computation : Numerical Methods*, NJ : Prentice-Hall, 1989
2. [Fre92] Freeman, T.L and Phillips, C., *Parallel Numerical Algorithms*, Prentice-Hall, 1992
3. [Hwa93] Hwang, Kai., *Advanced Computer Architecture : Parallelism, Scalability, Programmability*, McGraw-Hill, 1993
4. [Les93] Lester, Bruce P., *The Art of Parallel Programming*, NJ : Prentice-Hall, 1993
5. [Mat92] Mathews, John. H., *Numerical Methods for Mathematics, Science, and Engineering*, Second Edition, NJ : Prentice-Hall, 1992
6. [Qui87] Quinn, Michael J, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987