Analisis dan Perancangan Bahasa Pemrograman Paralel Beserta Pembuatan Prototipe Kompilatornya

Tjatur Kandaga
Jurusan Teknik Informatika
STMIK AMIKBANDUNG
Jl. Jakarta no. 28 Bandung

Abstract
Since the days of computer system invention, the requirement for computing power has been continuously increased. The most feasible solution for increasing computing power requirement is to build parallel computer system. Data and processes can be done in parallel, therefore a parallel computer system can provide bigger computing power than the fastest processor available at that times. Building blocks to establish a parallel processing system besides its hardware architecture are parallel algorithms, parallel programming languages and its compilers, and support from operating systems. Development in parallel processing has been supported by rapid advancement in hardware technology, change of paradigm in parallel programming models, features developed in parallel programming languages, and better techniques for compiler development.
Important criteria for parallel programming languages are features, efficiency, simplicity, expressiveness, locality, uniformity, and modularity. The designed parallel programming language named Parallel Programming Language version 0.1 (PPL v0.1). Current implementation of PPL v0.1 language compiler including lexical analyzer (scanner) and syntax analyzer (parser) stages. The lexical analyzer and syntax analyzer were implemented using flex and accent program generator, which is available on Linux/Unix operating system.

Keywords: parallel processing, language design, parallel programming language, compiler.

1. Pendahuluan

Kompilator adalah sebuah program yang digunakan untuk mengubah program sumber menjadi program tujuan. Pada umumnya program sumber merupakan program dalam bahasa tingkat tinggi dan program tujuannya adalah bahasa mesin atau bahasa assembler. Kompilator adalah salah satu program yang termasuk dalam golongan program sistem.

Kompilator merupakan program atau alat bantu yang sangat dibutuhkan oleh para pembangun program aplikasi. Dengan tersedianya kompilator untuk suatu bahasa tingkat tinggi yang berjalan pada prosesor tertentu, maka pembuatan program aplikasi menjadi jauh lebih mudah karena tidak perlu memprogram dalam bahasa mesin atau assembler.

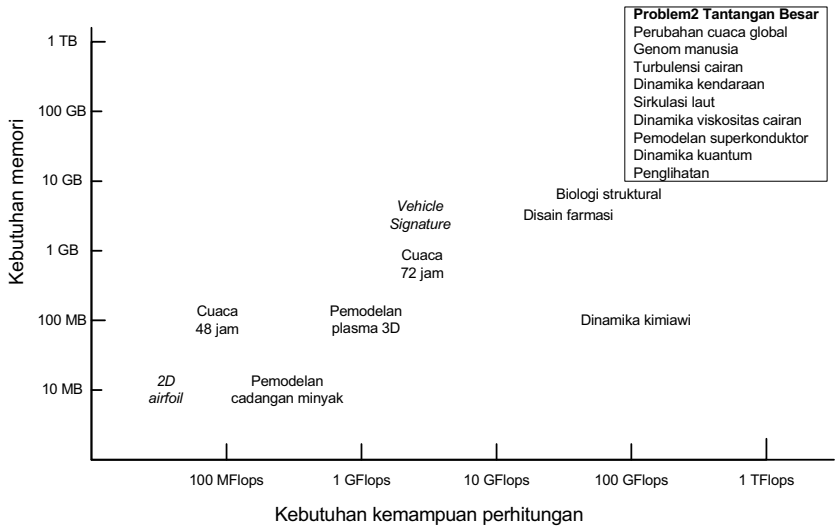
Kompilator merupakan program yang cukup besar dan kompleks, sehingga untuk membuat kompilator atau bahkan untuk mengerti cara kerja kompilator secara detail merupakan pekerjaan yang tidak mudah. Sekalipun demikian kompilator maupun interpreter dalam bentuk-bentuk yang lebih sederhana seringkali diperlukan di dalam program aplikasi. Beberapa hal yang biasa dilakukan program aplikasi dimana prinsip-prinsip dasar kompilator dapat dipergunakan diantaranya : pengecekan kesalahan input, pemilahan data input, transformasi format data, pengolahan perintah/bahasa *script*, dll. Dengan memahami cara kerja sebuah kompilator, para pembuat program akan mendapatkan dasar untuk mempertimbangkan pemilihan kode program. Pertimbangan tersebut diantaranya :

apakah kode tersebut akan mudah atau sulit diproses oleh kompilator, apakah kode tersebut dapat menghasilkan kode tujuan yang efisien, dan sebagainya.

2. Kebutuhan Pemrosesan Paralel

Kemajuan dalam kemampuan perangkat keras akan membuka kemungkinan untuk penyelesaian persoalan-persoalan yang selama ini belum dapat dipecahkan. Setiap aplikasi di bidang sains dan teknologi memiliki batas minimal kebutuhan kapasitas komputasi. Untuk memperoleh kapasitas komputasi yang lebih besar dari kemampuan prosesor tercepat yang tersedia maka caranya adalah dengan mengunakan lebih dari satu prosesor secara paralel.

United States High Performance Computing and Communications program mengidentifikasi aplikasi-aplikasi di bidang sains dan teknik yang merupakan *Grand Challenge* (tantangan besar) yang membutuhkan memori dan kemampuan perhitungan yang sangat besar dalam orde *teraflops (floating point operations per second),* yaitu 10^{12} operasi bilangan real perdetik [Cul99] [Bak96]. Gambar 1 menunjukkan kebutuhan pemrosesan aplikasi-aplikasi yang kompleks [Cul99].



Gambar 1: Kebutuhan aplikasi Grand Challenge

Supaya kemampuan komputer paralel dapat dimanfaatkan sepenuhnya maka diperlukan program yang secara khusus ditulis untuk komputer paralel. Dalam pembuatan program khusus tersebut digunakan bahasa pemrograman paralel beserta kompilatornya.

Komputer paralel juga dibutuhkan dan dapat digunakan dalam dunia bisnis, hiburan, medis, militer, dan bidang-bidang lainnya. Pada dasarnya komputer paralel dibutuhkan di setiap bidang dimana diperlukan kapasitas komputasi yang lebih tinggi dari yang dapat disediakan oleh komputer dengan prosesor tunggal. Di dalam dunia bisnis, komputer paralel dibutuhkan terutama karena semakin besarnya volume data yang harus diolah, dan pergeseran peran komputer yang awalnya hanya digunakan untuk transaksi operasional menjadi tulang punggung dalam sistem informasi *enterprise* secara keseluruhan. Kemampuan komputer dalam bidang bisnis dihitung berdasarkan *throughput*-nya dalam satuan jumlah transaksi permenit (tpm).

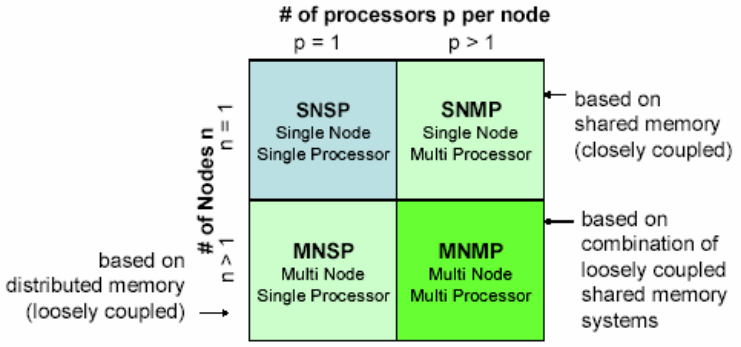
3. Klasifikasi dan Arsitektur Komputer Paralel

Komputer paralel adalah kumpulan dari elemen pemrosesan yang saling berkomunikasi dan berkooperasi untuk menyelesaikan persoalan yang besar secara cepat [Cul99] [Fos95]. Paralelisme terdiri dari beberapa tingkatan yang dibedakan dari tingkat abstraksinya. Pada tingkat yang lebih rendah paralelisme lebih bersifat *fine-grain* dan pada tingkat tinggi bersifat *coarse-grain*. Pemrosesan paralel yang dilakukan pada tiap tingkatan adalah sebagai berikut :

1. Tingkat program : program-program dieksekusi oleh sistem operasi secara paralel, bisa murni paralel atau secara *time sharing*.
2. Tingkat prosedur : beberapa bagian dari program yang sama dieksekusi secara paralel dalam bentuk proses atau *thread*.
3. Tingkat ekspresi : sebuah ekspresi aritmatik dibagi-bagi menjadi sub-ekspresinya kemudian bagian-bagian tersebut dieksekusi secara paralel.
4. Tingkat bit : pemrosesan paralel yang terdapat pada semua komputer von Neumann, dimana 8 atau 16 bit ALU memproses bit-bit data atau instruksi secara paralel.

Dalam penelitian ini pemrosesan paralel yang akan dibahas adalah pemrosesan paralel pada tingkat prosedur.

Pada tahun 1966 Flynn mengklasifikasikan komputer menjadi empat kategori, yaitu SISD, SIMD, MISD, dan MIMD. Tetapi *Electronic Intelligence* membagi kelas-kelas komputer paralel berdasarkan jumlah *node* dan jumlah prosessor per-*node*, sehingga lebih menggambarkan arsitektur fisiknya, seperti terlihat pada Gambar 2 berikut [EI02a].



Gambar 2: Klasifikasi komputer menurut EI

Pada klasifikasi tersebut selain jumlah *node* dan jumlah prosessor per-node, juga menggambarkan granularitas *node*, yaitu *distributed memory* (*loosely coupled*) dan *shared memory* (*closely coupled*).

Bentuk umum dari masing-masing kelas tersebut [EI02a] :

5. *Single Node Single Processor* (SNSP) pemrosesan paralel diperoleh dengan cara *time-sharing*.
6. *Single Node Multiple Processors* (SNMP) adalah komputer *shared memory* (*closely coupled*) dimana komputer terdiri dari sebuah *node* yang memiliki banyak prosessor yang mengakses memori global yang sama. Dalam kelas ini termasuk *Symmetric shared memory Multiprocessing Systems* (SMP), array/grid dan vektor komputer.
7. *Multiple Node Single Processor* (MNSP) adalah komputer *distributed memory/distributed system* (*loosely coupled*) dimana banyak node dengan

prosesor tunggal dihubungkan oleh jaringan membentuk *cluster* atau *network of workstation* (NOW).

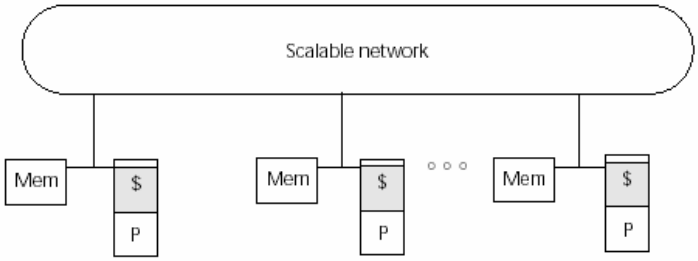
8. *Multiple Node Multiple Processor* (MNMP) terbentuk dari beberapa komputer SNMP yang terhubung melalui jaringan. MNMP adalah *loosely coupled cluster* dari *closely coupled nodes.* Contohnya adalah *cluster* dari SMP.

Masih ada beberapa kelas lagi yang tidak terlalu umum, diantaranya *Dataflow, Systolic Arrays, Wavefront Arrays, dan Very Long Instruction Word* (VLIW) [Brä93] [Cul99].

Model pemrograman paralel, terutama pada awal perkembangan komputer paralel sangat terkait dengan arsitektur perangkat kerasnya. Berikut ini adalah model pemrograman dan contoh bahasa pemrograman paralel atau kepustakaan paralel yang biasa digunakan, yang tersedia pada arsitektur perangkat keras tertentu [Bak96] [Cul99] [EI02a] :

a. SNSP : model pemrograman *multitasking* yang dijalankan secara *time sharing* pada satu prosesor dengan cara penjadwalan tertentu. Tersedia kepustakaan diantaranya pthreads.
b. SNMP : model pemrograman dan bahasa yang biasa digunakan :
   i) *Shared memory*, contoh : ABC++, Cilk, OpenMP.
   ii) Data paralel, contoh : Linda, *High Performance Fortran* (HPF), C*, NESL.
c. MNSP : model pemrograman dan bahasa yang biasa digunakan :
   i) *Message passing*, dengan kepustakaan yang tersedia : *Message Passing Interface* (MPI), *Paralel Virtual Machine* (PVM), P4, CSP/Occam, SR.
   iii) *Virtual Shared Memory* / *Distributed Shared Memory.*
d. MNMP : kombinasi antara SNMP dengan MNSP, sehingga dapat digunakan model pemrograman shared memory pada tingkat node, virtual shared memory untuk keseluruhan komputer, data paralel dan message passing.

Dalam perkembangannya arsitektur komputer parallel mengarah pada arsitektur komputer yang disebut NUMA, yaitu *NonUniform Memory Access* seperti pada gambar 3 [Cul99].



Gambar 3: NonUniform Memory Access (NUMA)

Arsitektur komputer NUMA memiliki beberapa kelebihan, diantaranya sebagai berikut :

e. Penempatan memori secara lokal akan mempercepat akses memori terutama jika akses ke *shared memory* dapat diminimalkan, karena memori lokal dapat diakses dalam orde nanodetik, sedangkan *shared memory*, memori global, atau memori pada *node* lain dalam orde mikrodetik [Bak96].

f. Penggunaan memori *cache* dapat mempercepat akses, karena *cache* dapat menyimpan data yang sering diakses baik dari memori lokal maupun dari *node* lain.
g. Penambahan prosesor hanya akan mengurangi *bandwidth* jaringan global, tetapi tidak berpengaruh terhadap akses lokal.
h. Pengontrol I/O (tidak terlihat pada gambar) dapat ditempatkan pada setiap *node* atau ditempatkan pada lokasi tersendiri.
i. *Shared memory* dapat disimulasikan dengan *virtual shared memory*, dimana pengontrol memori akan memeriksa setiap akses memori. *Shared memory* dibagi-bagi menjadi halaman-halaman (*pages*) dan mengambil sebagian tempat dari memori lokal masing-masing *node*. Untuk akses terhadap *shared memory* yang tidak terdapat pada memori lokal, akan terjadi *page fault* yang ditangani oleh sistem operasi, atau diterjemahkan menjadi permintaan data dari *node* lain dengan cara *message passsing*.
j. Arsitektur NUMA dengan satu prosesor pada setiap *node* sama dengan arsitektur MNSP yang digunakan dalam model pemrograman *message passsing*, baik berupa komputer *message passing* khusus ataupun *Network Of Workstations* (NOW).

   Arsitektur NUMA di atas dapat melayani model pemrograman *shared memory* maupun *message passing*. Pada arsitektur NUMA, setiap *node* dapat memiliki lebih dari satu prosesor, dan memiliki perangkat untuk mengontrol komunikasi dengan jaringan.

   Secara umum kebutuhan pemrosesan paralel yang berhubungan dengan perangkat keras adalah sebagai berikut ini.

9. Pemrosesan yang cepat, dan kapasitas penyimpanan yang besar.
10. Skalabilitas, sehingga memudahkan penambahan elemen pemrosesan, yaitu prosesor, memori, perangkat I/O, perangkat jaringan, ataupun sebuah node lengkap.
11. Keterandalan, dimana sistem harus dapat diandalkan untuk melakukan pemrosesan tanpa berhenti dalam jangka waktu yang panjang.
12. *Error recovery*, kemampuan untuk memulihkan diri dari kegagalan pemrosesan, yaitu dapat kembali ke posisi sebelum kesalahan terjadi.
13. *Fault tolerant*, jika terjadi kegagalan pada satu bagian, bagian lain masih dapat berfungsi dengan baik.
14. *Hot plugable*, dapat melakukan penambahan, pengurangan atau penggantian peralatan tanpa harus menonaktifkan keseluruhan sistem.
15. Kompatibilitas, penggunaan komponen pemrosesan yang dapat digantikan dengan produk lain yang sejenis.

4. Bahasa Pemrograman Paralel

Kebutuhan pemrosesan paralel yang berhubungan dengan perangkat lunak, khususnya bahasa pemrograman paralel pada dasarnya sama dengan bahasa pemrograman sekuensial dengan beberapa kebutuhan tambahan. Cara pemrogram memandang persoalan, melakukan analisa, menyusun dan menerapkan pemecahan masalah yaitu cara pemrogram melakukan pekerjaannya akan mempengaruhi desain bahasa pemrograman, demikian juga sebaliknya desain bahasa pemrograman akan mempengaruhi cara berfikir dan pola penyelesaian masalah yang dilakukan oleh pemrogram. Hal tersebut menunjukkan pentingnya rancangan bahasa pemrograman yang baik.

Konstruksi sintaks dan semantik dari sebuah bahasa pemrograman memiliki banyak aspek. Kriteria tersebut seringkali memiliki keterkaitan satu dengan yang lainnya baik secara positif (saling menguatkan), atau secara negatif (saling melemahkan), dan terkadang didalamnya sudah tercakup beberapa kriteria lainnya. Kriteria kebutuhan bahasa pemrograman paralel yang disusun mulai dari aspek yang paling penting adalah sebagai berikut :

16. Fitur (*features*) : Fitur/fasilitas yang tersedia untuk mendukung pemrosesan program secara paralel. Sebuah bahasa pemrograman paralel pasti memiliki satu atau lebih fitur yang dapat digunakan untuk melakukan pemrosesan paralel. Fitur merupakan kriteria kebutuhan yang paling penting, karena fitur tersebutlah yang memungkinkan dilakukannya pemrosesan paralel. Melalui kriteria ini dapat ditinjau beberapa hal :
    i) Fitur apa yang disediakan.
    ii) Apakah fitur tersebut merupakan hasil dari gagasan yang cemerlang, atau bahkan merupakan sebuah terobosan baru dalam paradigma pemrograman paralel.
    iii) Apakah fitur tersebut memang sangat dibutuhkan, atau dapat menjawab kebutuhan pemrosesan paralel secara umum.
    iv) Bagaimana fitur tersebut berkaitan dengan aspek komunikasi, pemrosesan paralel dan/atau sinkronisasi dalam pemrograman paralel.

    Fitur pemrosesan paralel merupakan bagian dari konstruksi bahasa, sehingga dapat dievaluasi juga menggunakan kriteria-kriteria lainnya.
17. Efisiensi : sintaks dan semantik bahasa harus dirancang sedemikian rupa sehingga memungkinkan kompilator untuk menghasilkan kode yang efisien. Meskipun kriteria ini sangat berkaitan dengan aspek implementasi kompilator dari bahasa pemrograman, tetapi perancang bahasa harus berusaha menghindari rancangan sintaks yang akan menyulitkan kompilator dalam pemrosesan dan pembangkitan kode dari bahasa tersebut. Efisiensi berkaitan dengan waktu yaitu kecepatan pemrosesan, ruang yaitu penggunaan tempat penyimpanan baik primer maupun sekunder, dan penggunaan sumber daya komputer lainnya, dan yang tidak boleh diabaikan adalah efisiensi waktu, tenaga, pikiran dan biaya dari pembuat program. Kecepatan pemrosesan menyangkut dua hal yaitu kecepatan proses kompilasi program, dan kecepatan eksekusi program yang dihasilkan. Kemampuan kecepatan pemrosesan yang tinggi merupakan alasan utama penggunaan komputer paralel, sehingga kriteria efisiensi menjadi sangat penting untuk diperhatikan.

    Efisiensi waktu merupakan gambaran dari kinerja sebuah program. Sebuah program dituntut untuk memiliki kinerja/unjuk kerja yang maksimal, karena jika kinerja sebuah program kurang baik maka tujuan dari dilakukannya pemrosesan paralel menjadi tidak tercapai. Dalam hal ini yang diperhatikan adalah kinerja dari program yang dibuat dengan bahasa pemrograman paralel, terlepas dari perangkat keras yang digunakan.
18. *Simplicity* : diharapkan hanya menambahkan sedikit konsep dasar paralel terhadap bahasa pemrograman sekuensial, tetapi dapat memenuhi kebutuhan algoritma-algoritma pemrosesan paralel. *Simplicity* (kesederhanaan) merupakan hal yang sangat penting, karena bahasa yang cukup sederhana akan mudah diingat sehingga mempermudah pemrogram dalam membangun dan menerapkan strategi pemecahan masalah tanpa harus melihat buku manual dari bahasa pemrograman yang digunakan. *Simplicity* tidak boleh dicapai dengan melakukan generalisasi yang berlebihan, karena akan menghasilkan bahasa

yang sangat sulit untuk diiplementasikan dengan baik. Sebaliknya *simplicity* seharusnya dicapai dengan ide/konsep yang sederhana tetapi dapat menjawab kebutuhan dan dapat diimplementasikan dengan efisien.

Sebuah bahasa pemrograman yang memiliki banyak sekali konsep dasar paralel (tidak sederhana), dapat menjadi petunjuk desain bahasa yang kurang baik, yang mungkin disebabkan oleh beberapa hal, diantaranya sebagai berikut :

- Bahasa tidak memiliki sebuah fitur yang cukup berdaya guna (*powerful*).
- Pembuat bahasa gagal dalam mengidentifikasi kebutuhan utama dari pengguna bahasa pemrograman paralel.
- Pembuat bahasa tidak berhasil melihat inti dari kompleksitas yang terdapat pada pemrosesan paralel.
- Pembuat bahasa berusaha untuk membuat bahasa yang dapat menjawab kebutuhan semua jenis pemrosesan paralel yang berbeda-beda, sehingga mengorbankan kesederhanaan dan efisiensi.
- Bahasa pemrograman paralel dibuat menggunakan basis bahasa pemrograman sekuensial yang sudah terlalu kompleks.
- Bahasa dibuat oleh sebuah kelompok yang cukup besar, yang hanya memprioritaskan kepentingannya masing-masing.

Kesederhanaan sangat berkaitan dengan efisiensi dan *expressiveness*, dimana yang diharapkan adalah bahasa pemrograman yang efisien dan sederhana tetapi tetap cukup ekspresif. Kriteria kesederhanaan merupakan kriteria yang lebih penting dari *expressiveness*, karena *expressiveness* dapat dicapai salah satunya dengan cara menyediakan banyak konstruksi sintaks untuk jenis-jenis algoritma paralel yang berbeda-beda sehingga mengorbankan kesederhanaan dan efisiensi.

19. *Expressiveness* : bahasa pemrograman paralel harus dapat mengimplementasi-kan algoritma paralel dengan jelas dan mudah. *Expressiveness* sangat berkaitan erat dengan *clarity* (kejelasan), efektivitas, dan abstraksi.

    Bahasa pemrograman tingkat tinggi lebih abstrak dari bahasa tingkat rendah, sehingga lebih ekspresif dalam penerapan algoritma. Pada kasus tertentu abstraksi juga dapat dianggap mengurangi *expressiveness*, misalnya dalam pembuatan program untuk mengontrol peralatan yang harus melakukan akses terhadap perangkat keras secara langsung. Dalam kasus ini penggunaan bahasa tingkat tinggi menjadi kurang ekspresif karena tidak menyediakan fasilitas yang cukup untuk melakukan akses perangkat keras, atau fasilitas tersebut telah dibungkus sehingga menjadi tidak ekspresif. Berdasarkan kedua kasus di atas, maka tingkat abstraksi yang dapat menghasilkan *expressiveness* yang maksimal adalah yang sesuai dengan tingkat abstraksi dari pemikiran pemrogramnya. Sehingga sebuah bahasa pemrograman adalah ekspresif bagi seorang pemrogram jika pemrogram tersebut dapat menerapkan secara langsung konsep-konsep yang dipergunakannya untuk memikirkan pemecahan masalahnya.

    Jika faktor tingkat abstraksi dan subjektivitas pemrogram diabaikan, masih dapat diidentifikasi adanya perbedaan antara bahasa pemrograman yang dapat menggambarkan pola/strategi pemecahan masalah yang diambil pemrogram dengan jelas, dan bahasa yang memaksa pemrogram untuk menterjemahkan pola pemecahan masalahnya kedalam bentuk lain yang menyebabkannya menjadi tidak jelas (tidak ekspresif). *Expresiveness* ini penting karena bahasa yang ekspresif akan memudahkan pemrogram untuk menerapkan algoritma dan

pola berfikirnya kedalam program, dan program menjadi lebih sederhana dan mudah dibaca.

20. Lokalitas : dalam pemrograman paralel, kemampuan mengenali dan menerapkan lokalitas akan sangat mempengaruhi kinerja dan menyederhanakan interaksi antar proses. Dari sisi bahasa pemrograman, lokalitas dapat didukung dengan menyediakan sintaks bahasa yang memungkinkan pengelompokan dan penempatan data dan proses. Dalam model arsitektur komputer NUMA dan *distributed memory*, proses yang dilakukan secara lokal akan lebih cepat dan efisien. Lokalitas berkaitan dengan modularitas, meskipun demikian modularitas tidak dapat menjamin untuk terjadinya lokalitas yang baik, sehingga lokalitas dianggap lebih penting dari modularitas.
21. *Uniformity* : konsep dasar paralel yang disediakan dapat digunakan secara konsisten pada konteks yang berbeda-beda dalam program tanpa harus merubah bentuknya. Penerapan konsep yang sama secara berbeda-beda pada konteks yang berbeda akan sangat menyulitkan pemrogram, karena harus selalu mengingat konteks yang sedang dihadapinya, dan mengingat perbedaan-perbedaan sintaks dan/atau semantik yang ada. Sebaliknya sebuah konsep yang dapat diterapkan secara seragam dalam konteks yang berbeda-beda selama penerapan konsep tersebut secara intuitif memiliki arti yang jelas akan membuat pemrogram merasa nyaman dan bebas berimprovisasi.
22. Modularitas : mekanisme untuk membagi program menjadi modul-modul yang memiliki saling ketergantungan sesedikit mungkin akan sangat membantu dalam membentuk unit-unit pemrosesan paralel. Bahasa pemrograman dengan modularitas yang baik akan dapat meminimalkan komunikasi yang diperlukan. Modularitas dapat diterapkan pada tingkatan-tingkatan yang berbeda, misalnya pada tingkatan subprogram/subrutin, kelas dan objek, struktur data, file atau direktori, dan lain-lain. Sebagian besar bahasa pemrograman terstruktur telah memiliki mekanisme tertentu untuk melakukan modularitas, sehingga kriteria ini dianggap sudah merupakan kriteria kebutuhan yang umum.

Beberapa kriteria di luar kriteria di atas yang lebih banyak berhubungan dengan aspek implementasi dari bahasa pemrograman paralel yaitu pembangunan kompilator untuk bahasa tersebut diantaranya seperti di bawah ini :

1. Skalabilitas : program dapat dijalankan pada komputer dengan kemampuan pemrosesan yang rendah sampai komputer berkemampuan tinggi. Skalabilitas sangat berguna pada saat pengembangan program, dimana program dapat dikembangkan dan diuji coba pada komputer berkemampuan rendah terlebih dahulu, kemudian baru dijalankan pada komputer berkemampuan tinggi untuk melakukan proses sebenarnya. Hal ini dapat menghemat biaya operasional komputer berkemampuan tinggi.
2. Portabilitas : program yang dibuat dalam bahasa pemrograman paralel tersebut sebaiknya dapat dijalankan pada *platform* yang berbeda-beda. Portabilitas akan membantu dalam proses migrasi dari satu *platform* ke *platform* lainnya, dan memungkinkan program dijalankan pada sistem komputer yang terdiri dari komputer dengan *platform* yang berbeda-beda. Portabilitas memberikan kebebasan pemilihan *platform* dan membuka peluang bagi bahasa pemrograman dan kompilatornya untuk digunakan oleh lebih banyak kalangan.
3. *Safety* (keamanan) : kemampuan untuk dapat mendeteksi kesalahan semantik, sebaiknya pada saat kompilasi, dan kemampuan untuk mendeteksi dan

memberitahukan kesalahan pada waktu program dijalankan. Kemampuan pendeteksian kesalahan tentunya akan membantu pemrogram dalam memperbaiki kesalahan-kesalahan yang terjadi. *Safety* hampir selalu bertolak belakang dengan efisiensi, karena *safety* hanya dapat dicapai dengan menyertakan perangkat tambahan yang dapat menurunkan efisiensi dan kinerja dari program. Pendeteksian kesalahan pada saat kompilasi akan memperlambat proses kompilasi dan menambah ukuran kode dari kompilator. Pendeteksian kesalahan pada saat program dijalankan akan memperlambat eksekusi program dan menambah ukuran kode program.

Sebuah bahasa pemrograman didefinisikan dengan cara menspesifikasikan sintaks (struktur), dan semantik (arti) dari bahasa tersebut. Spesifikasi sebuah bahasa pemrograman biasanya dinyatakan dalam bentuk Backus Naur *Form* (BNF), *Extended* BNF, atau Diagram Sintaks.

Bahasa pemrograman paralel dapat dikelompokan berdasarkan model pemrogramannya. Suatu bahasa dapat memiliki fitur yang mendukung lebih dari satu model pemrograman, sehingga dapat terdaftar pada beberapa kelompok. Berikut ini adalah beberapa bahasa pemrograman paralel yang dikelompokkan berdasarkan model pemrogramannya, beserta fitur pemrosesan parallelnya.

4. Shared memory / distributed (virtual) shared memory
   - Ada : protected type
   - Cilk : *fork/join*
   - CxC : komputer paralel virtual, *connected memory*
   - HPC++ : *thread*
   - Java : *synchronized method*
   - SR : *fork/join* dan *semaphore*
   - Titanium : *zone based memory management*
5. Message passing / distributed memory
   - Ada : rendezvous
   - Concurrent C : *rendezvous*
   - CSP/Occam : *synchronous message passing*
   - CxC : topologi, array *controller*
   - Emerald : objek terdistribusi
   - Fortran M : *asynchronous message passing*
   - HPC++ : objek terdistribusi, *remote method*
   - Java : *network* dan *remote method invocation*
   - MPI : *message passing*
   - SR : *message passing, RPC, rendezvous*
   - Titanium : *broadcast, exchange, scan, reduce*
6. Data parallel
   - C* : *data layout,* eksekusi paralel
   - HPF : pemetaan data, statemen array, parallel *loop*
   - NESL : *nested data parallelism*
   - ZPL : *data region* dan *direction,* operasi array
7. Fungsional
   - NESL : paralelisme rekursif
   - Sisal : paralelisme rekursif dan iterasi
8. Logic
   - Concurrent Prolog : evaluasi *AND* dan *OR* paralel

- PARLOG : evaluasi *AND* dan *OR* paralel

Berdasarkan hasil tinjauan bahasa-bahasa pemrograman paralel, diperoleh hasil analisis yang dikelompokkan berdasarkan kriteria kebutuhan yang telah ditetapkan. Hasil analisis hanya menyertakan bahasa-bahasa yang memiliki aspek untuk memenuhi suatu kriteria, dan mengevaluasi kelebihan atau kekurangan dari kriteria tersebut. Sehingga bahasa-bahasa yang tidak memiliki aspek yang cukup berarti untuk suatu kriteria kebutuhan tidak disertakan dalam tabel.

9. Fitur

Tabel 1 *Fitur dari bahasa pemrograman paralel*

Bahasa	**+/-**	**Keterangan**
Ada	-	Mekanisme sinkronisasi dalam perintah select tidak memiliki penjagaan output.
Ada	-	Penerimaan *entry call* selalu FIFO tidak bisa menggunakan kondisional.
Ada	-	Tidak dapat memberikan prioritas terhadap alternatif pilihan.
Cilk	+	Fungsi inlet menyediakan cara reduksi (*reduction*) yang sederhana tetapi sangat fleksibel.
Cilk	+	Memiliki fungsi pengukur kinerja (*profiler*) *built-in* pada *runtime system*.
CxC	+/-	Harus mendefinisikan protokol *virtual*.
CxC	-	Menyediakan fungsi *scan* dan *reduction* yang sangat banyak sehingga terkesan menambah kompleksitas bahasa.
Emerald	-	Hanya merupakan bahasa pemrograman berbasis objek, bukan bahasa pemrograman berorientasi objek (tidak ada kelas dan *inheritance*).
Java	-	Kelas *ThreadGroup* tidak menunjukkan kegunaan yang signifikan.
Titanium	-	Menyediakan fungsi *scan* dan *reduction* yang sangat banyak sehingga terkesan menambah kompleksitas bahasa.
Titanium	+	Manajemen memori *zone based* merupakan kompromi yang baik antara *automatic garbage collection* dg manajemen memori secara manual.
HPF	+	Mekanisme pengulangan paralel dengan perintah FORALL dan *compiler directive* INDEPENDENT.

Bahasa Ada sudah memiliki fitur-fitur yang cukup baik untuk menangani komunikasi dan sinkronisasi, tetapi fitur-fitur tersebut dirasa tanggung karena fungsinya tidak dilengkapi dengan kemampuan penetapan prioritas, penjagaan *output*, dan pemilihan *entry call*.

Bahasa Cilk memiliki pengukur kinerja yang terdapat pada *runtime system*-nya, dan fitur ini tidak terdapat pada semua bahasa lainnya. Cilk juga memiliki spesifikasi/*modifier* fungsi inlet yang sangat sederhana dan dapat menggantikan semua fungsi-fungsi *scan* dan reduksi seperti yang terdapat pada bahasa CxC dan Titanium. Keharusan mendefinisikan protokol *virtual* pada bahasa CxC merupakan suatu kelemahan untuk pembuatan program yang sederhana, dan hanya dijalankan pada satu jenis komputer. Tetapi juga merupakan keunggulan, dimana protokol *virtual* dapat digunakan untuk membuat program mengikuti model pemrograman *task parallel* atau *data parallel*.

Bahasa Java memiliki fitur yang tidak memiliki kegunaan yang jelas, tetapi fitur tersebut tidak segera dibuang pada versi yang lebih baru.

Manajemen memori *zone based* pada bahasa Titanium merupakan jalan tengah antara *automatic garbage collection* dengan manajemen memori secara manual, juga dapat digunakan untuk mengatur lokalitas data.

Mekanisme pengulangan paralel pada HPF, dan *compiler directive* yang disediakan mudah untuk diterapkan pada berbagai persoalan.

10. Efisiensi

4. Efisiensi dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
Cilk	+	Kode sangat efisien, *overhead* yang minimal terhadap ANSI C.
Cilk	+	Penjadwalan dengan *nanoscheduler, micro-scheduler* dan *work stealing*, serta pendistribusian beban kerja otomatis akan meningkatkan kinerja /efisiensi waktu.
Cilk	+	Memenangkan berbagai kontes pemrograman.
CxC	+	Penggunaan *virtual machine* untuk masing-masing platform sangat memungkinkan untuk dioptimalkan sehingga tetap efisien pada setiap *platform*.
Java	+	Penggunaan *virtual machine* untuk masing-masing platform sangat memungkinkan untuk dioptimalkan sehingga tetap efisien pada setiap *platform*.
Titanium	+	Kelas *immutable* menghasilkan kelas/objek yang lebih efisien.

Efisiensi kode dan penjadwalan pada bahasa Cilk sangat baik, penambahan kode terhadap bahasa C sebagai bahasa dasar sangat sedikit. Penggunaan *virtual machine* pada CxC dan Java membuka peluang untuk efisiensi.

11. Simplicity

5. *Simplicity* dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
Cilk	+	Hanya menambahkan sedikit kata kunci dan fungsi terhadap bahasa C.
Cilk	+	Programmer hanya menstrukturkan program tidak perlu memusingkan protokol dan penjadwalan.
HPF	+	Hanya menambahkan kemampuan *data parallel* terhadap Fortran 90.
HPF	+	Secara umum model *data parallel* memiliki sintaks dan pengaturan yang lebih sederhana dari model *task parallel*.
MPI	+	Menambahkan kepustakaan *message passing* terhadap bahasa pemrograman sekuensial.

12.

Bahasa Cilk memiliki pengaturan protokol dan penjadwalan secara otomatis, tetapi tetap efisien. HPF memiliki sintaks dan pengaturan yang sederhana karena menggunakan model pemrograman *data parallel*.

Meskipun kepustakaan MPI memiliki fungsi yang sangat banyak, tetapi sebagian besar program hanya membutuhkan enam buah fungsi dasar dari MPI.

13. Expressiveness

6. *Expressiveness* dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
HPF	+	Operasi terhadap array/matriks yang sangat ekspresif.
Titanium	+	Array multidimensi yang ekspresif karena didefinisikan berdasarkan domain yang telah dibentuk terlebih dahulu.

14.

Bahasa HPF sesuai tujuan pembuatannya untuk komputasi *scientific* memiliki operasi array/matriks yang sangat ekspresif. Titanium memiliki fasilitas untuk mendefinisikan domain data.

15. Lokalitas

7. Lokalitas dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
Emerald	+	Migrasi objek/proses meningkatkan lokalitas.
HPF	+	*Compiler directive* untuk distribusi atau pemetaan data dapat mengoptimalkan lokalitas data.

16.

Migrasi objek/proses secara dinamis pada Emerald dapat meningkatkan lokalitas, termasuk jika terjadi perubahan pola eksekusi dari program. HPF menggunakan *compiler directive* untuk mengoptimalkan lokalitas data, sehingga pengaturan data dilakukan pada saat proses kompilasi program.

17. Uniformity

8. *Uniformity* dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
Emerald	+	Memiliki sintaks yang sama dalam pemanggilan operasi lokal maupun *remote.*

Pemanggilan operasi tidak berubah untuk lokal maupun *remote*.

18. Modularitas

9. Modularitas dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
Java	+	Fasilitas pengelompokan kelas-kelas menjadi paket (*package*).

Hampir semua bahasa pemrograman paralel telah memiliki fasilitas untuk membagi program menjadi modul-modul, tetapi Java memiliki fasilitas yang tidak terdapat pada bahasa lain.

Beberapa kriteria yang lebih banyak berhubungan dengan aspek implementasi kompilator dari bahasa pemrograman paralel.

19. Skalabilitas

10. Skalabilitas dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
Cilk	+	Skalabilitas baik karena bisa dijalankan pada komputer dengan 1 s/d n prosesor.
CxC	+	Skalabilitas baik karena bisa dijalankan pada komputer dengan 1 s/d n prosesor.
Java	+	Skalabilitas baik karena bisa dijalankan pada komputer dengan 1 s/d n prosesor.
Java	-	Terdapat batas maksimal jumlah *thread*, dimana jika jumlah *thread* melewati batas tertentu maka proses menjadi sangat lambat. Batas maksimal ini sulit untuk diketahui dan berbeda-beda tergantung dari sistem operasi dan JVM yang digunakan.

Sebagian besar bahasa pemrograman paralel telah memiliki skalabilitas yang cukup baik.

20. Portabilitas

11. Portabilitas dari bahasa pemrograman paralel

Bahasa	**+/-**	**Keterangan**
Cilk	+	Kode sumber portabel karena menumpang pada gcc, GNU make, dan POSIX threads.
Cilk	+	Kompilator yang sederhana : cilk = cilk2c + gcc.
Cilk	-	Hanya jalan di komputer *shared memory* & *distributed/virtual shared*

		memory.
Cilk	+	Sudah jalan di Linux/386, Linux/Alpha, Linux/IA-64, Solaris/SPARC, Irix/MIPS, OSF/Alpha.
CxC	+	*Executable* kode portabel karena menggunakan *virtual machine / runtime system* yang terpisah.
CxC	+	Berjalan di komputer *shared* maupun *distributed memory*.
CxC	-	Harus membuat *virtual machine* untuk semua *platform* yang akan didukung.
Java	+	*Executable* kode portabel karena menggunakan *virtual machine / runtime system* yang terpisah.
Java	+	Berjalan di komputer *shared* maupun *distributed memory*.
Java	-	Harus membuat *virtual machine* untuk semua *platform* yang akan didukung.
MPI	+	Tersedia pada banyak *platform* yang berlainan.
Titanium	+	Kode sumber portabel karena menumpang pada bhs C.
Titanium	+	Sudah jalan di beberapa superkomputer (CrayT3D, IBM SP, SGI Origin 2000).

Portabilitas dari bahasa pemrograman paralel diatas diperoleh dengan beberapa cara, yaitu menggunakan bahasa C sebagai bahasa antara, menggunakan kepustakaan paralel yang telah tersedia pada berbagai *platform*, atau menggunakan *runtime system/virtual machine*.

21. Safety

Kemampuan pendeteksian kesalahan biasanya telah disertakan pada setiap kompilator. Tingkat kemampuan pendeteksian kesalahan untuk sebuah bahasa dapat berbeda-beda antara implementasi kompilator yang berbeda. Pendeteksian kesalahan untuk elemen di luar sintaks suatu bahasa biasanya sangat terbatas, misalnya pendeteksian kesalahan untuk kepustakaan MPI yang digunakan dalam sebuah program.

Beberapa kriteria dan aspek lain yang diperoleh ketika menganalisa bahasa-bahasa pemrograman paralel adalah sebagai berikut :

12. Kriteria lainnya dari bahasa pemrograman paralel

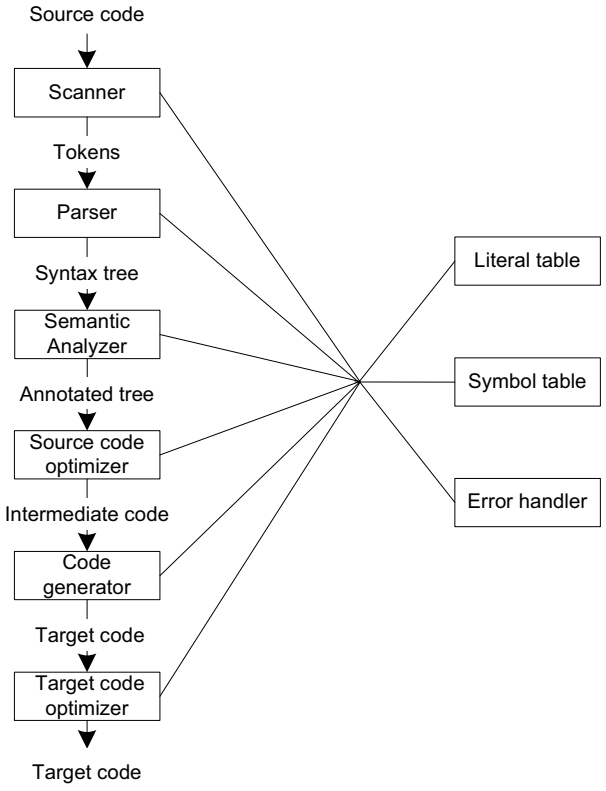
Bahasa	**+/-**	**Keterangan**
Cilk	+	Lisensi GNU/GPL dapat menarik banyak dukungan dan penggunaan yang luas.
Cilk	+	Sebagian besar strategi yang diterapkan adalah hasil penelitian & tesis PhD (*dag consistency memory model, cactus stack*, pengukuran *critical path,* penjadwalan *nanoscheduler*, *mikroscheduler* dan *work stealing).*
Cilk	+	*Runtime system* yang dapat diprediksi akan membantu mengurangi sifat nondeterministik pada pemrograman paralel.
CxC	-	Bahasa *proprietary* milik EI.
CxC	-	Cukup mudah untuk membuat program yang benar, tetapi untuk membuat program yang efisien adalah jauh lebih sulit, karena pemrogram dituntut untuk dapat mengatur topologi komputer *parallel virtual* dengan baik sehingga eksekusi program menjadi efisien.
CxC	+	Sudah tersedia GUI untuk development.
HPF	+	Didesain oleh tim yang cukup besar yang tergabung dalam HPFF.
HPF	+	Banyak digunakan oleh komunitas *High Performance Computing* (HPC).
HPF	+	Banyak menyediakan fungsi-fungsi matematis, yang diantaranya

		dapat diterapkan pada sebagian atau keseluruhan dari array/matriks.
HPF	-	Cukup mudah untuk membuat program yang benar, tetapi untuk membuat program yang efisien adalah jauh lebih sulit, karena pemrogram dituntut untuk dapat mengatur pemetaan data dengan baik sehingga eksekusi program menjadi efisien.
Java	+	Sudah tersedia GUI untuk development.
MPI	+	Didesain oleh banyak kalangan.
MPI	-	Implementasi-implementasi yang tersedia tidak seragam dan sebagian besar bukan implementasi yang lengkap.
MPI	+	Merupakan standar *de facto* untuk pemrograman *message passing*.
Titanium	+	Merupakan bagian dari penelitian, tesis master dan PhD yang berkelanjutan.

Sebagian besar bahasa pemrograman paralel dibuat di universitas dan pusat penelitian, sehingga merupakan bagian dari penelitian yang berkelanjutan. Pengembangan bahasa dengan menggunakan lisensi *Open Source* GNU/GPL memungkinkan lebih banyak pihak berpartisipasi dalam pengembangan dan penggunaan bahasa.

5. Perancangan Kompilator

Tahap-tahap dari sebuah kompilator adalah sebagai berikut [Lou97] :



Gambar 4: Tahap-tahap sebuah kompilator

Tahap pengoptimasian kode sumber dan tujuan tidak mutlak harus ada. Tahap-tahap pemrosesan kompilator tersebut adalah sebagai berikut :

k. *Scanner* (Penganalisa Leksikal)

Scanner bertugas membaca program sumber yang biasanya dalam bentuk urutan karakter-karakter, kemudian mengubah urutan karakter-karakter tersebut menjadi *token-token,* yaitu unit-unit leksikal yang mempunyai arti. Selain mengenali *token*, biasanya *scanner* melakukan tugas lain seperti memasukkan *identifier* (pengenal) kedalam tabel simbol, dan memasukkan literal kedalam tabel literal.

l. *Parser* (Penganalisa Sintaks)
Parser menerima *token-token* dari *scanner* dan melakukan analisa sintaks yang menentukan elemen-elemen struktural dari program dan hubungan diantaranya. Hasil analisa sintaks biasanya dinyatakan dalam bentuk *parse tree* atau pohon sintaks.

m. *Semantic Analyzer* (Penganalisa Semantik)
Semantik dari program merupakan "arti" dari program tersebut, yang dapat dibagi menjadi semantik statik yaitu perilaku program yang dapat ditentukan dari teks programnya tanpa dieksekusi, dan semantik dinamis yang hanya dapat diketahui ketika program dijalankan.
Penganalisa semantik bertugas menganalisa semantik statik, yang diantaranya meliputi pemeriksaan tipe data, dan pemeriksaan deklarasi-deklarasi. Informasi yang dihasilkan dari analisa semantik biasa disebut sebagai atribut, yang digabungkan kedalam tabel simbol atau *abstract syntax tree* sebagai anotasi atau dekorasi.

n. *Source Code Optimizer* (Pengoptimasi Kode Sumber)
Kompilator biasanya menyertakan satu atau lebih pengoptimal kode. Posisi paling awal yang memungkinkan langkah ini dilakukan adalah setelah penganalisa semantik, dimana terdapat kemungkinan untuk pengoptimalan kode berdasarkan kode sumber. Pengoptimalan dapat dilakukan terhadap pohon sintaks beranotasi, atau yang lebih mudah yaitu terhadap bentuk pohon sintaks beranotasi yang sudah disederhanakan menjadi kode tiga alamat (*three address code*), atau *P-code* yang banyak digunakan oleh kompilator bahasa Pascal.
Hasil dari tahap ini berupa kode antara (*intermediate code*), atau secara lebih umum disebut representasi antara (*intermediate representation / IR*).

o. *Code Generator* (Pembangkit kode)
Generator kode menerima kode antara kemudian memprosesnya sehingga menghasilkan kode target, biasanya dalam bentuk bahasa mesin atau bahasa assembly. Dalam tahap ini keunikan dari masing-masing komputer tujuan harus diperhitungkan, baik dalam pemilihan kode bahasa mesin / assembly maupun dalam cara merepresentasikan data.

p. *Target Code Optimizer* (Pengoptimasi Kode Tujuan)
Tahap ini berusaha melakukan pengoptimalan program berdasarkan kode target yang sudah dihasilkan. Beberapa pengoptimalan yang dapat dilakukan diantaranya adalah memilih atau mengganti mode pengalamatan untuk meningkatkan unjuk kerja, mengganti instruksi yang lambat dengan instruksi yang dapat dieksekusi lebih cepat, dan membuang operasi yang tidak diperlukan.

6. Program Bantu Pembangunan Kompilator

Setelah spesifikasi sebuah bahasa pemrograman terbentuk, untuk membuat kompilatornya dapat digunakan program bantu berupa generator

kompilator. Keuntungan penggunaan program generator kompilator dibandingkan dengan memprogram sendiri diantaranya [Gru00] :

1. Input untuk program generator memiliki tingkat abstraksi yang lebih tinggi daripada program buatan sendiri. Programmer tidak perlu membuat spesifikasi yang terlalu banyak, karena detail kompilator akan ditangani oleh generator, sehingga memperkecil kemungkinan kesalahan.
2. Penggunaan program generator meningkatkan fleksibilitas dan kemudahan modifikasi. Jika terjadi perubahan sintaks, hanya perlu memodifikasi spesifikasi dan menjalankan kembali generator, sedangkan bila program dibuat sendiri mungkin harus melakukan banyak perubahan.
3. Kode-kode tambahan seperti pengecekan kesalahan masukan seringkali sudah disertakan secara langsung pada kode yang dihasilkan, sehingga menambah keandalannya tanpa harus bersusah payah membuatnya sendiri.

Deskripsi formal sebuah bahasa seringkali dapat digunakan untuk menghasilkan lebih dari satu tipe program. Misalnya sebuah aturan tata bahasa selain digunakan untuk menghasikan sebuah *parser*, dapat juga digunakan untuk menghasilkan sebuah teks editor yang dapat mengenali konstruksi sintaks bahasa tersebut.

Kode yang dihasilkan program generator mungkin kurang efisien atau sebaliknya mungkin juga lebih efisien, tetapi dengan menggunakan program generator waktu dan usaha yang diperlukan akan jauh lebih sedikit.

Program generator kompilator yang tersedia saat ini sebagian besar hanya mencakup satu atau beberapa tahap pemrosesan kompilator, dan sebagian besar hanya tersedia pada sistem operasi Unix / Linux. Program generator yang tersedia diantaranya : lex, yacc (*yet another compiler compiler*), flex, bison, reflex, coco, accent, *ANother Tool for Language Recognition* (antlr), JavaCC, Memphis, *Gentle compiler construction system* (gccs).

7. Implementasi Bahasa Pemrograman Paralel

Rancangan bahasa PPL v0.1 menggunakan bahasa pemrograman C sebagai bahasa dasar dengan alasan supaya cukup sederhana, sehingga dapat berkonsentrasi pada aspek paralel yang akan dirancang. Konsep model *nested data parallel* diambil dari bahasa V dan NESL.

Hal yang paling mendasar dari rancangan yang diajukan adalah adanya struktur data *vector* dan model pemrosesan *vector comprehension* yang dapat melakukan pemrosesan pada elemen-elemen vektor secara independen, sehingga dapat dieksekusi secara paralel. Vektor merupakan struktur data yang mirip dengan array dan dideklarasikan dengan menggunakan notasi [*]. Dalam contoh berikut, v merupakan sebuah vektor dengan elemen yang bertipe integer, ptr_ke_vector merupakan pointer terhadap vektor dengan elemen bertipe integer, matriks merupakan vektor dari vektor dengan elemen bertipe integer, dan func merupakan fungsi yang menghasilkan *return value* berupa vektor dari bilangan integer.

- Deklarasi vektor

```
int v[*], (*ptr_ke_vektor)[*];
int matriks[*][*];
int func() [*] {...}
```

Vektor berisi data dengan tipe yang sama dengan panjang/ banyak elemen yang dapat berbeda-beda. Model ekspresi *vector comprehension* diimplementasikan dalam bentuk konstruksi *apply to each*.

- Konstruksi *apply to each*

```
xs      = [1, 2, 3];
pangkat = [ x*x : x in xs ];
less    = [x : x in xs : x < pivot];
y       = xs[1];
z       = $xs;
```

Pada contoh di atas, xs merupakan sebuah vektor dengan tiga elemen, pangkat akan merupakan vektor dengan elemen-elemennya berisi hasil dari perhitungan x*x untuk setiap x pada vektor xs, dan less akan berupa vektor dengan elemen-elemen bernilai x untuk setiap x pada xs dimana x lebih kecil dari nilai pada variabel pivot. Variabel y digunakan untuk mengakses elemen pada xs pada posisi indeks satu, dimana indeks vektor dimulai dari 0. Panjang dari suatu vektor dapat diperoleh dengan notasi $ seperti pada contoh di atas, dimana variabel z akan berisi panjang dari vektor xs.

Dua buah vektor dapat digabungkan dengan notasi <>. Suatu *range* nilai dapat didefinisikan dengan menggunakan notasi [awal::akhir::step], dimana akan dihasilkan vektor yang memiliki nilai mulai dari awal, awal+step, dan seterusnya sampai nilai terakhir yang masih lebih kecil dari akhir.

- Penggabungan dan *range* vektor

```
gab     = vek1 <> [3, 5, 7];
urut    = [1 :: 10 :: 2]
```

Juga terdapat *pure function* yang merupakan fungsi yang tidak memiliki efek-samping (*side effect*) secara global.

Implementasi dari *Scanner* / penganalisa leksikal dilakukan dengan mempergunakan program generator flex yang terdapat pada sistem operasi Linux/Unix. Program flex menerima file masukan (pplv01.lex) yang berisi ekspresi regular untuk semua *token* yang harus dikenali oleh penganalisa leksikal, dan menghasilkan file program penganalisa leksikal untuk bahasa PPL v0.1 dalam bahasa C (lex.yy.c).

File masukan untuk flex, yaitu pplv01.lex berisi definisi-definisi ekspresi regular dari *token* yang terdapat pada bahasa PPL. Pada kode di bawah ini *token-token* yang merupakan tambahan terhadap bahasa C dicetak dengan huruf tebal.

- Sebagian file masukan untuk program flex

```
%{
/*---------- Lexical analyzer untuk PPLv0.1 ------------*/
#include "yygrammar.h"
%}
/* ----------- definitions section ------------*/
let     [_a-zA-Z]
alnum       [_a-zA-Z0-9]
h           [0-9a-fA-F]
o           [0-7]
d           [0-9]
suffix  [UuLl]
white       [\r\t ]
sep     [(){},;.]
ident       {let}{alnum}*

/* token rules section */
```

```
%%
"static"        { return STATIC; }
"pure"          { return PURE; }
"void"      { return VOID; }
"char"      { return CHAR; }
"int"           { return INT; }
"float"     { return FLOAT; }
"[*]"           { return VECTOR; }
"..."           { return ELIPSIS; }
"case"      { return CASE; }
"if"            { return IF; }
"else"      { return ELSE; }
"switch"        { return SWITCH; }
"while"         { return WHILE; }
"do"            { return DO; }
"for"           { return FOR; }
"goto"      { return GOTO; }
"continue"  { return CONTINUE; }
"break"         { return BREAK; }
"return"        { return RETURN; }
"in"            { return IN; }
"<|"            { return REPLACE; }
"|>"            { return EXTRACT; }
"&&"            { return L_AND; }
"=="            { return L_EQ; }
"<>"            { return CONCAT; }
"*="            { return MUL_AS; }
"::"            { return RANGE; }
"define"        { return DEFINE; }
"include"   { return INCLUDE; }
{sep}           { return yytext[0]; }
\n              { yypos++;  /* no baris source */ }
\"(\\.|[^\"])*\"        { return STRING; }
\"(\\.|[^\"])*[\r\n]        { yyerror("String tidak ditutup }
                                {  dengan \".\n"); return STRING; }
([0-9])+{suffix}?           { return CONST_INT; }
\'.\'                       { return CONST_CHAR; }
({d}+|{d}+\.{d}*|{d}*\.{d}+)([eE][-+]?{d}+)?[fF]?
                                { return CONST_FLOAT; }
{ident}         { return IDENTIFIER; }
{let}*      { return CONST_ENUM; }
{white}+        { /* white space */ }
"/*"([^*/])*"*/"        { /* komentar */}
.               { return yytext[0]; /* teks lainnya */}
```

Program flex memiliki parameter –d (*debug*), yang jika digunakan akan menampilkan *token* yang berhasil dibaca beserta nomor baris dari ekspresi regular pada file masukan yang digunakan untuk mengenali *token* tersebut. Contoh hasil keluaran penganalisa leksikal dengan *debug* dari program sumber kode 7.

- Hasil penganalisa leksikal dengan *debug*

```
--(end of buffer or a NUL)
--accepting rule at line 26 ("void")
--accepting rule at line 100 (" ")
--accepting rule at line 98 ("main")
```

```
--accepting rule at line 91 ("(")
--accepting rule at line 91 (")")
--accepting rule at line 92 ("
")
--accepting rule at line 91 ("{")
--accepting rule at line 92 ("
")
--accepting rule at line 92 ("
")
--accepting rule at line 91 ("}")
--accepting rule at line 92 ("
")
--(end of buffer or a NUL)
--EOF (start condition 0)
```

Implementasi *parser* / penganalisa sintaks dilakukan dengan menggunakan program *generator* accent yang dapat dijalankan pada sistem operasi Linux/Unix. Program accent menerima file masukan (pplv01.acc) yang berisi tata bahasa (*grammar*) dalam notasi EBNF (*Extended Backus Naur Form*), dan menghasilkan program *parser* untuk bahasa PPL v0.1 dalam bahasa C (yygrammar.h dan yygrammar.c). File yygrammar.h berisi semua definisi *token*, dan digunakan oleh program *scanner* lex.yy.c.

Untuk setiap aturan dari notasi EBNF dapat ditambahkan aksi semantik (*semantic actions*), yaitu aksi yang akan dilakukan jika aturan tersebut dipenuhi oleh program *input* yang sedang diproses. Aksi semantik tersebut berupa kode dalam bahasa C, dan akan disisipkan pada program *parser* yang dihasilkan.

Accent dapat menerima aturan sintaks yang memiliki rekursif kiri, sehingga tidak diperlukan perubahan terhadap aturan sintaks dalam notasi EBNF yang memiliki rekursif kiri.

Runtime system accent dapat mendeteksi jika terdapat aturan sintaks yang ambigu, dan memberikan saran untuk memperbaikinya. Ambiguitas diantara alternatif-alternatif aturan sintaks dapat diselesaikan dengan memberikan anotasi prioritas %prio n yang berbeda pada aturan-aturan yang ambigu. Nilai n yang lebih tinggi menggambarkan prioritas yang lebih tinggi. Ambiguitas di dalam alternatif-alternatif aturan sintaks dapat diselesaikan dengan memberikan anotasi %long atau %short didepan sebuah nonterminal. Anotasi %long digunakan jika nonterminal diharapkan untuk menghasilkan string yang lebih panjang, dan sebaliknya anotasi %short jika nonterminal diharapkan menghasilkan string yang lebih pendek. Fasilitas untuk penyelesaian ambiguitas dengan menggunakan anotasi pada aturan sintaks, menyebabkan tidak diperlukannya perubahan bentuk aturan sintaks dengan menggunakan teknik substitusi dan pemfaktoran kiri (*left factoring*), seperti yang biasa dilakukan untuk menghilangkan ambiguitas pada tata bahasa LL(1) dan LALR(1).

Bentuk aturan sintaks pada bahasa PPL yang memiliki rekursif kiri dan ambiguitas diantara alternatif yang telah diperbaiki adalah sebagai berikut.

- Aturan sintaks dengan rekursif kiri dan ambiguitas

```
statement_list:
  statement
| statement_list statement
;
```

```
selection_statement:
  IF '(' expression ')' statement                  %prio 1
| IF '(' expression ')' statement ELSE statement   %prio 2
| SWITCH '(' expression ')' statement
;
```

Kemampuan untuk memproses aturan tata bahasa dengan rekursif kiri dan ambiguitas merupakan keunggulan accent, yang diperoleh dengan cara menerapkan dua teknik *parsing*, yaitu *predictive parsing* seperti yang biasa dilakukan pada tata bahasa LL(1), dan *exhaustive parsing* yang dapat digunakan pada semua tata bahasa. *Predictive parsing* memeriksa sebuah *input token* berikutnya (*lookahead token*) untuk memilih sebuah alternatif yang unik. *Exhaustive parsing* digunakan yang akan menelusuri semua alternatif yang ada secara paralel. *Exhaustive parsing* digunakan untuk meperoleh generalitas, dan *predictive parsing* digunakan untuk meningkatkan efisiensi.

Bahasa PPL versi 0.1 menambahkan beberapa aturan sintaks terhadap bahasa C, yang berkaitan dengan vektor dan pemrosesan paralel.

Berikut ini adalah sebuah program singkat dalam bahasa PPL v0.1 dan keluaran yang dihasilkan penganalisa sintaks dengan menggunakan *preprocesor directive* TRACE dan PRINTTREE. Hasil dari *preprocesor directive* TRACE hanya disertakan sebagian kecil, karena untuk program yang singkat seperti program di bawah saja sudah menghasilkan *output* lebih dari seribu baris.

- Program dalam bahasa PPL v0.1

```
void main()
{

}
```

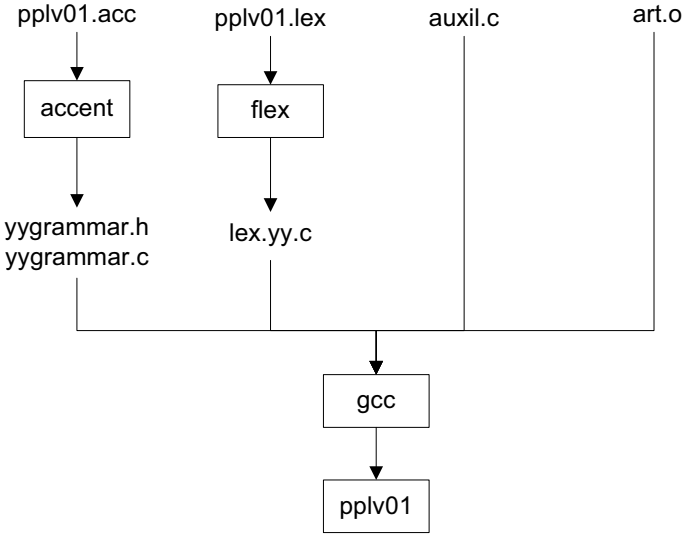
- Output menggunakan *preprocesor directive* TRACE dan PRINTTREE

```
/* ============================> trace ========== */
initial itemlist:
 1: YYSTART : * translation_unit '
closure
predictor for item 1:
 2: translation_unit : * external_declaration (back:1 sub:0 left:0)
 3: translation_unit : * translation_unit external_declaration (back:1 sub:0 left:0)
    . . .
next closure iteration
predictor for item 1:
 14: translation_unit : * external_declaration (back:1 sub:0 left:0)
 [already inserted]
    . . .
89: YYSTART : translation_unit '
closure
completer for item 89:
end of itemlist
 90: [ separator-item ]
END OF RECOGNIZER

/* ============================> printtree ========== */
translation_unit alternative at line 16, col 22 of grammar {
```

```
external_declaration alternative at line 22, col 21 of grammar {
 function_definition alternative at line 28, col 24 of grammar {
  declaration_specifiers alternative at line 47, col 16 of grammar {
   type_specifier alternative at line 62, col 6 of grammar {
    VOID
   }
  }
  declarator alternative at line 145, col 19 of grammar {
   direct_declarator alternative at line 155, col 19 of grammar {
    direct_declarator alternative at line 149, col 12 of grammar {
     IDENTIFIER
    }
    '('
    ')'
   }
  }
  compound_statement alternative at line 256, col 3 of grammar {
   '{'
   '}'
  }
 }
}
}
```

Hubungan semua file dan program yang digunakan untuk menghasilkan program penganalisa leksikal dan semantik pplv01 dapat digambarkan sebagai berikut.



Gambar 6: Pembangunan program pplv01

Untuk membangun program pplv01 yang dapat melakukan analisa sintaks dan analisa leksikal diperlukan program yang berfungsi sebagai *driver*. Program tersebut (auxil.c) berisi rutin main yang akan memanggil fungsi utama dari program *parser* yaitu yyparse(). Program *scanner* akan dipanggil oleh program *parser* ketika diperlukan (*ad-hoc scanner*) dengan memanggil fungsi utamanya yaitu yylex(). Program *parser* yang dibangun dengan accent membutuhkan modul *runtime system* yaitu art.o yang harus digabungkan pada proses *linking*.

Untuk melakukan pencarian kesalahan dan penelusuran pohon sintaks yang dihasilkan, kode sumber untuk file objek art.o yaitu art.c yang disertakan dalam program accent memiliki beberapa definisi *preprocesor directive* yang dapat diubah sesuai kebutuhan, yang terpenting diantaranya TRACE, DETECTAMBIGUITY dan PRINTTREE. Dengan memberikan nilai 1 (*true*) pada TRACE maka accent akan menampilkan penelusuran rule yang dilakukan dalam rangka mencocokkan token yang diperoleh dari penganalisa leksikal. *Preprocesor directive* PRINTTREE 1 (bernilai *true*) akan menghasilkan pohon sintaks dari program yang dianalisa. Pohon sintaks hanya dihasilkan jika proses analisa sintaks berhasil dengan baik. DETECTAMBIGUITY 1 berfungsi untuk mendeteksi jika terdapat aturan sintaks yang ambigu, dan STATISTICS 1 akan menghasilkan statistik dari program yang dianalisa.
Pengujian analisa leksikal dan analisa sintaks terhadap prototipe kompilator bahasa pplv01 dilakukan dengan cara *black box*. Hasil pengujian cukup memuaskan, dimana pplv01 telah dapat mendeteksi kesalahan leksikal maupun sintaks dari program sumber, dan telah dapat menghasilkan pohon sintaks untuk program yang benar secara leksikal dan sintaks.

8. Kesimpulan dan Saran

Penelitian ini membahas mengenai pemrosesan paralel yang meliputi arsitektur perangkat keras untuk pemrosesan paralel, analisis bahasa-bahasa pemrograman paralel yang ada, perancangan bahasa pemrograman paralel dengan model pemrograman *nested data parallel*, dan perancangan kompilator untuk bahasa pemrograman paralel tersebut.
Beberapa kesimpulan yang dapat diambil adalah sebagai berikut :

a. Bahasa pemrograman paralel PPL v0.1, masih merupakan bahasa pemrograman paralel yang sederhana, tetapi memiliki dasar-dasar paralel yang cukup lengkap untuk dikembangkan lebih lanjut. Model *nested data parallel* pada bahasa PPL v0.1 memiliki fitur paralel untuk mendefinisikan vektor, *range* vektor, dan ekspresi *vector comprehension*.
b. Teknik-teknik dan fitur-fitur yang digunakan oleh bahasa-bahasa pemrograman paralel semakin berkembang, sehingga menghasilkan bahasa-bahasa pemrograman paralel yang semakin baik.
c. Bahasa PPL v0.1 sudah dapat memenuhi kriteria fitur, efisiensi, *simplicity*, dan *uniformity* dengan baik. Sedangkan kriteria *expresiveness*, lokalitas, dan modularitas dapat dipenuhi secara terbatas.

Beberapa saran untuk penelitian lebih lanjut bahasa pemrograman paralel adalah sebagai berikut.

a. Bahasa PPL v0.1 memiliki model pemrograman *nested data parallel*. Bahasa tersebut dapat dikembangkan lagi menjadi model pemrograman *object parallel*, bahkan tidak tertutup kemungkinan untuk membuat *parallelizing compiler*-nya. Model pemrograman *object parallel* akan dapat menggabungkan model *task parallel* dengan *data parallel*.
b. Supaya bahasa pemrograman paralel semakin banyak digunakan dan berkembang, salah satu cara adalah dengan mengaplikasikan pemrosesan paralel pada bidang-bidang lain diluar pusat-pusat riset dan laboratorium penelitian yang selama ini telah menggunakan pemrosesan paralel, misalnya pada bidang bisnis, pemerintahan dan sosial.

Beberapa saran untuk penelitian lebih lanjut kompilator bahasa pemrograman paralel adalah sebagai berikut.

a. Program analisa leksikal dan analisa sintaks yang dihasilkan masih harus dilanjutkan pengembangannya, sehingga menghasilkan kompilator yang lengkap.
b. Penanganan kesalahan pada program penganalisa sintaks yang dihasilkan oleh pembangkit program accent perlu dikembangkan supaya dapat menghasilkan pesan kesalahan yang lebih spesifik, dan memiliki fitur *error recovery*. Pengembangan dapat dilakukan terhadap program analisa sintaks yang dihasilkan oleh accent, atau mengembangkan program accent yang didistribusikan berikut kode sumbernya dengan lisensi GNU *General Public License*.
c. *Runtime system* accent perlu dikembangkan supaya dapat menjalankan pemrosesan paralel. Strategi pengembangan *runtime system* accent dapat dilakukan dengan menggunakan kepustakaan paralel yang tersedia seperti MPI, pthread atau Globus, menangani sendiri pemrosesan paralel dengan mempergunakan POSIX *thread* pada sistem operasi Linux, atau mengubah implementasi *runtime system* accent menggunakan bahasa yang menyediakan fitur pemrosesan paralel, misalnya *thread* pada bahasa Java dan C++.

Daftar Pustaka

Baker, Lou, Smith, Bradley J. (1996). *Parallel Programming*. McGraw-Hill Inc.

Bräunl, Thomas. (1993). *Parallel Programming: An Introduction.* Prentice-Hall International Inc.

Chakravarty, Manuel M.T., Schröer, Friedrich W., Simons, Martin. (2003). *V Reference Manual.* GMD FIRST.
Available: http://www.first.fraunhofer.de/v/manual.ps. Accessed: 15/04/2003.

Culler, David E., Singh, Jaswinder Pal, Gupta, Anoop. (1999). *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann.

Eckel, Bruce. (2003). *Thinking in Java, 3rd ed.* Prentice Hall.
Available: http://www.mindview.net/books/tij/TIJ-3rd-edition4.0.zip. Accessed: 23/05/2003.

_____. (2002). *The CxC Parallel Programming Guide.* Fort Collins, CO. Engineered Intelligence.
Available: http://www.engineeredintelligence.com/download/manuals /CxCPrgGud.pdf. Accessed: 02/01/2003.

Finkel, Raphael A. (1996). *Advanced Programming Language Design.* Addison-Wesley Publishing Company.

Foster, Ian. (1995). *Designing and Building Parallel Programs.* Addison Wesley
Available: http://www-unix.mcs.anl.gov/ dbpp/text/book.html. Accessed: 17/01/2003.

Grune, Dick, Bal, Henri E., Jacobs, Ceriel J.H., and Langendoen, Koen G. (2000). *Modern Compiler Design.* John Wiley & Sons Ltd., 2000.

Holub, Allen I. (2002). *Programming Java Threads in The Real World*, Java World, 2002, http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html, 01/07/2002.

Kernighan, Brian W., Ritchie, Denis M. (1998). *The C Programming Language, 2nd ed.* Prentice Hall.

Louden, Kenneth C. (1997). *Compiler Construction: Principles and Practice*. PWS Publishing Company, a division of International Thomson Publishing Inc.

Schröer, Friedrich W. (1997). *The GENTLE Compiler Construction System*. R. Oldenbourg Verlag, Munich and Vienna
Available: www.first.fraunhofer.de/gentle /BOOK.ps .gz. Accessed: 26/11/2002.

_____. (2000). *Cilk 5.3.1. Reference Manual,* Supercomputing Technologies Group. MIT Laboratory for Computer Science
Available: http://supertech.lcs.mit.edu/cilk/manual-5.3.1.pdf. Accessed: 26/11/2002.

_____. (1999). *Titanium Tutorial.* Computer Science Division, University of California at Berkeley and Lawrence Berkeley National Laboratory
Available: http://www.cs. berkeley.edu/Research/Projects/titanium/doc/tutorial.ps. Accessed: 22/10/2002.

Yelick, Kathy, et. al. (1998). *Titanium: A High-Performance Java Dialec.* Computer Science Division, University of California at Berkeley and Lawrence Berkeley National Laboratory.
Available: http://www.cs.berkeley.edu/Research/Projects /titanium/papers/hpj98.ps. Accessed: 22/10/2002.