KAJIAN PERANGKATBANTU KOMPUTASI *PARALLEL* PADA JARINGAN *PC*

Heru Suhartanto

Fakultas Ilmu Komputer, Universitas Indonesia, Depok 16424, Indonesia

E-mail: heru@cs.ui.ac.id

Abstrak

Banyak model fenomena alam, aplikasi *engineering*, dan industri membutuhkan Sumber Daya Komputasi (SDK) yang tinggi untuk memroses data sehingga menghasilkan informasi yang dibutuhkan. Teknologi komputasi tingkat tinggi pun diperkenalkan banyak peneliti dengan diciptakannya *Supercomputer* beserta *Operating System* dan perangkatbantu (*tools*) pengembangnya seperti kompilator dan pustaka (*library*). Namun, mahalnya investasi SDK ini baik dalam pengadaan maupun pemeliharaannya memberatkan banyak pihak, sehingga diperlukan alternatif SDK yang tetap berkinerja tinggi tetapi murah. Untuk mengatasi keterbatasan tersebut, para peneliti telah membuat konsep alternatif, yakni konsep komputasi *parallel* pada jaringan komputer yang sudah ada. Banyak perangkatbantu diciptakan guna mengembangkan aplikasi dalam sistem SDK yang memanfaatkan mesin atau komputer dalam suatu jaringan, dimana masing-masing komputer ini berperan sebagai pemroses layaknya pemroses dalam sistem *super computer*. Tulisan ini akan mengkaji beberapa perangkatbantu yang cukup dominan di kalangan pemakai, yakni *Parallel Virtual Machine* (PVM), *Message Passing Interface* (MPI), *Java Remote Method Invocation* (RMI), serta *Java Common Object Request Broker Architecture* (CORBA) dan menyajikan eksperimen untuk mengetahui perangkatbantu mana yang paling cocok sehingga dapat pembantu calon *user* dalam memilihnya. Percobaan dilakukan pada SDK berbasis jaringan komputer pribadi (*Personal Computer*) dan menghasilkan percepatan yang cukup berarti. Dari keempat perangkatbantu tersebut masing-masing teridentifikasi cocok untuk pengembangan pada kondisi tertentu.

Abstract

A Study on Parallel Computation Tools on Networked PCs. Many models for natural phenomena, engineering applications and industries need powerfull computing resources to solve their problems. High Performance Computing resources were introduced by many researchers. This comes in the form of Supercomputers and with operating systems and tools for development such as parallel compiler and its library. However, these resources are expensive for the investation and maintenance, hence people need some alternatives. Many people then introduced parallel distributed computing by using available computing resource such as PCs. Each of these PCs is treated as a processors, hence the cluster of the PC behaves as Multiprocessors Computer. Many tools are developed for such purposes. This paper studies the peformance of the currently popular tools such as Parallel Virta\ual Machine (PVM), Message Passing Interface (MPI), Java Remote Method Invocation (RMI) and Java Common Object Request Broker Architecture (CORBA). Some experiments were conducted on a cluster of PCs, the results show significant speed up. Each of those tools are identified suitable for a certain implementation and programming purposes.

Keywords: cluster computing, parallel, tools

1. Pendahuluan

Dalam masyarakat teknologi informasi, terdapat beberapa aplikasi yang membutuhkan proses cepat namun terkendala dengan kondisi perangkat keras (*hardware*). Aplikasi ini biasanya didominasi oleh aplikasi dalam bidang rekayasa (*engineering*) atau suatu aplikasi yang memakai suatu metode namun metode ini tidak bisa dipakai dengan hanya mesin atau komputer berprosesor tunggal. Persoalan yang coba diatasi oleh aplikasi tersebut antara lain permasalahan konsentrasi zat polusi *(pollutant)* dan pemecahan suatu model matematis yang menggunakan metode yang perlu dukungan teknologi komputasi parallel. Persoalan konsentrasi zat polusi itu digambarkan dalam Sistem Persamaan *Differential Ordiner* (PDO) berikut:

$$\frac{\partial c_s}{\partial t} = -\left(\frac{\partial uc_s}{\partial x} + \frac{\partial vc_s}{\partial y} + \frac{\partial wc_s}{\partial z}\right) + \frac{\partial(K_x \partial c_s / \partial x)}{\partial x} + \frac{\partial(K_y \partial c_s / \partial y)}{\partial y} + \frac{\partial(K_z \partial c_s / \partial z)}{\partial z} + E_s(\theta,t) - (k_{1s} + k_{2s})c_s(\theta,t) + R_s(c_1,...,c_q) \quad (1)$$

Dalam hal ini s = 1, …, q, $c_s(\theta,t)$ adalah konsentrasi dari polutan ke s pada ruang θ, kemudian $u(\theta,t), v(\theta,t) \text{ dan } w(\theta,t)$ adalah kecepatan angin sepanjang sumbu x, y dan z, $E_s(\theta,t)$ menyatakan emisi pada titik ruang θ dan waktu t untuk polutan ke s, k_{1s} dan k_{2s} berturut-turut adalah koefisien deposisi kering dan basah, K(θ,t) menyatakan koefision difusi sepanjang tiga sumbu koordinat, dan R_s menyatakan reaksi kimia yang terkait dengan komponen ke *s* [1,2].

Labotorium polusi udara Denmark mempelajari model ini dengan q=29 dan jika metode garis dipakai maka akan terbentuk empat sistem nilai awal PDO. Sistem tersebut kemudian harus dipecahkan secara siklus pada setiap langkah integrasi. Sebagai contoh, jika grid sumbu x, y dan z yang dipakai berukuran 32 x 32 x 9, maka masing-masing PDO akan mengandung 267,264 persamaan yang harus dicari solusinya pada setiap langkah integrasi. Oleh karena model itu harus dicari solusinya sepanjang garis skala waktu untuk mempelajari variasi dalam satuan bulan dan musim, maka ini hanya dapat ditangani oleh lingkungan komputasi yang dapat menangani jumlah tingi operasi bilangan dalam satuan detik (*floating operations per seconds - flops*). Salah satu aplikasi yang telah dikembangkan misalnya adalah aplikasi pemecahan persoalan yang direpresentasikan oleh model matematika PDO [1]. Pengembangan aplikasi tersebut relatif mudah karena ia memakai perangkatbantu dan *compiler* khusus yang berjalan di atas sistem SDK super. Namun, investasi dan perawatan SDK ini sangat mahal sehingga memacu orang untuk mencari alternatif.

Karena itulah muncul ide pengembangan sistem komputasi *Cluster* dengan memanfaatkan SDK yang sudah ada menjadi suatu sumber daya komputasi yang tinggi. Perkembangan ini mulai dari pengaitan mesin-mesin yang ada dari berbagai jenis sampai pengaitan sekumpulan (*cluster*) PC menjadi SDK berkinerja tinggi. Masing-masing mesin yang berpartisipasi dapat dianggap sebagai suatu prosesor, yaitu masing-masing prosesor dapat melakukan proses dengan memakai data yang berbeda, sehingga secara keseluruhan komputasi yang dilakukan di sistem ini disebut sebagai *komputasi tersebar*. Pengembangan aplikasi sistem tersebar ini menggantungkan pada perangkatbantu (*tools*) yang berbasis tukar pesan (*message passing*). Beberapa hasil riset terdahulu misalnya aplikasi pemecah sistem PDO memakai metode VMRK berbasis MPI [3], implementasi *parallel* modifikasi metode Triangulasi Delunay untuk pembentukan permukaan objek dengan perangkatbantu PVM dan MPI [4] dan *analisis* unjuk kerja alat Bantu Java RMI dan Java Corba [5]. Namun demikian belum terdapat kajian yang melihat secara rinci hasil percobaan keseluruhan empat perangkatbantu gratis tersebut.

Makalah ini bertujuan mengkaji secara literature dan percobaan kinerja perangkatbantu tersebut dalam memecahkan suatu persoalan, kajian ini diharapkan dapat membantu pemakai untuk memilih perangkat yang sesuai dengan kebutuhannya. Bagian kedua membahas perangkatbantu yang dominan sudah dipakai, bagian tiga membahas percobaan memakai perangkatbantu tersebut, bagian empat membahas analisis kinerja perbandingan perangkatbantu tersebut, dan diakhiri dengan penutup dan beberapa kesimpulan.

2. Kajian Literatur Perangkatbantu Komputasi *Parallel*

Bagian ini mengulas secara ringkas prinsip-prinsip perangkatbantu yang dapat dipakai untuk memfasilitasi pengembangan aplikasi yang memanfaatkan komputer komputer dalam suatu jaringan menjadi SDK yang meniru SDK *multi*prosesor. Masing-masing komputer tersebut dapat berperan sebagai suatu pemroses (prosesor). Salah satu komputer akan bertindak sebagai *master* (*clients* dalam istilah *Java*) dengan tugas utama sebagai manajer dan lainnya sebagai slave (*server* dalam istilah *Java*) dengan tugas melakukan komputasi sesuai arahan *master*. Karena masing-masing pemroses tersebut harus turut melakukan komputasi, maka interaksi pengiriman data dari satu pemroses ke pemroses lainnya dilakukan dengan pengiriman data/pesan (*message passing*). Empat perangkatbantu yang menonjol dibahas dalam literatur, yakni *Parallel Virtual Machine* (PVM), *Message Passing Interface* (MPI), *Java Remote Method Invocation* (RMI), dan *Java Common Object Request Broker Architecture* (CORBA).

2.1 PVM

PVM adalah perangkat lunak yang membuat sekumpulan komputer menjadi tampak seperti sebuah sistem komputer *virtual* yang besar. Sekumpulan komputer yang akan terlibat dalam proses penyelesaian masalah harus didefinisikan trelebih dahulu, agar dapat menjalankan fungsinya. Komputer-komputer yang terlibat dalam komputasi bisa homogen, dengan *platform* yang sama, maupun heterogen, dengan *platform* yang berbeda, asal di antara mereka bisa saling berkomunikasi. PVM dapat menangani semua pengiriman proses, konversi data, dan penjadwalan *task*

secara *message passing* untuk sistem yang tidak kompatibel sekalipun.

Sistem PVM terdiri dari dua bagian. Bagian pertama adalah *daemon* yang diberi nama *pvmd*. *Pvmd* diaktifkan di setiap komputer yang akan membentuk mesin *virtual*. Bagian kedua adalah pustaka rutin antarmuka PVM yang berisi koleksi perintah-perintah primitif untuk mengoperasikan proses-proses pustaka tersebut. Pustaka rutin ini digunakan dalam program aplikasi paralel yang ditulis dengan bahasa pemrograman C, C++, atau FORTRAN 77. Aplikasi dalam bahasa pemrograman C dan C++ dihubungkan melalui pustaka *libpvm3.lib*, sedangkan aplikasi dalam bahasa pemrograman FORTRAN 77 dapat mengambil rutin-rutin dalam *libfpvm3.lib*. Kedua pustaka tersebut telah tersedia pada saat instalasi [6].

PVM memberi fasilitas untuk membuat sejumlah proses yang tidak tergantung dari jumlah prosesor. Setiap proses diidentifikasi menggunakan kode (*task* ID) dan dipetakan ke prosesor secara otomatis, atau dapat juga diatur oleh *programmer*. Program PVM umumnya diatur dengan model *master-slave*, yaitu satu proses yang dieksekusi pertama kali menjadi *master* dan mengaktifkan semua *client* dengan memanggil *pvm_spawn*. Rutin tersebut otomatis akan menjalankan semua proses dalam sistem PVM. Cara lain untuk menjalankan proses adalah dengan memanggil rutin *pvm_mytid* yang mengembalikan kode *task* ID dari proses tersebut. Sebelum keluar dari sistem PVM, semua proses *client* harus dimatikan dari PVM dengan memanggil rutin *pvm_exit*.

Komunikasi antar proses di dalam sistem PVM dilakukan secara *message passing* menggunakan perintah rutin PVM seperti *pvm_send* dan *pvm_recv*. Semua rutin pengiriman pesan dilakukan secara asinkron, sedangkan semua rutin penerimaan pesan dapat dilakukan secara sinkron maupun asinkron. Ada tiga tahap dalam mengirim pesan dalam PVM, yaitu :

- Menginisialisasi *buffer* pengiriman dengan rutin *pvm_initsend*
- Mengemas isi pesan ke dalam *buffer* dengan memanggil rutin *pvm_pk**. Data yang dikemas dapat bertipe *byte*, *complex*, *double*, *float*, *integer*, dan *character*. Tipe tersebut dinyatakan dengan mengganti tanda "*" dan memberi tipe yang sesuai untuk tiap parameter di dalam rutin *pvm_pk**. Misalnya data yang akan dikemas bertipe *float*, maka digunakan *pvm_pkfloat()*.
- Mengirim pesan ke prosesor tujuan dengan memanggil *pvm_send* atau *pvm_mcast*

Proses yang menerima pesan harus membuka paket pesan pada *buffer* penerima sesuai dengan format pengiriman pesan.

PVM juga menyediakan rutin *pvm_setopt* untuk mengatur pilihan dalam sistem PVM, seperti pencetakan pesan kesalahan secara otomatis, tingkat pencarian kesalahan (*debugging level*), dan metode pengaturan jalur komunikasi. Contoh yang paling umum dari penggunaan *pvm_setopt* adalah memungkinkan jalur komunikasi langsung antar *task* dalam PVM. *Pvm_setopt(PvmRoute, PvmRouteDirect);* Dengan perintah ini otomatis *bandwidth* komunikasi di dalam jaringan akan digandakan.

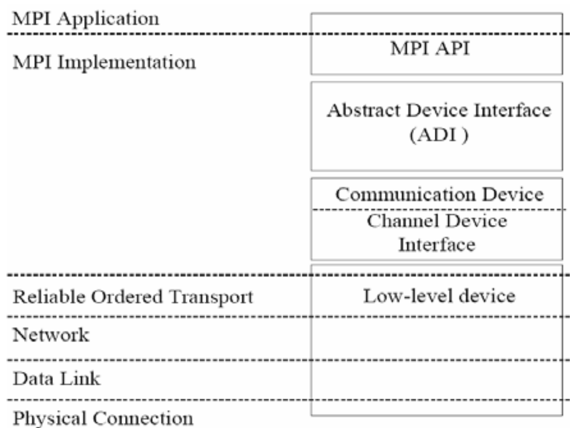
PVM adalah pustaka yang bersifat *opensource* yang tersedia dalam berbagai versi. Pada makalah ini yang digunakan adalah PVM versi 3.4 yang berjalan di bawah sistem operasi *Windows*. Sistem *Windows* tidak disertai fasilitas RSH untuk *remote login* dan *remote access* seperti pada sistem Unix. Untuk keperluan ini digunakan Ataman RSHD [7] yang dapat bekerja di bawah sistem operasi *Windows* mulai versi *Windows* 95, dan terpasang protokol TCP/IP. Rincian tentang PVM dapat dilihat pada [6], dan diulas juga secara ringkas di [4].

2.2. MPI

Dalam implementasinya MPI menggunakan fungsi-fungsi pustaka yang dapat dipanggil dari program C, C++, atau Fortran. Hampir sama dengan PVM, MPI juga *portable* untuk berbagai arsitektur. Salah satu implementasi terbaru pada saat penulisan makalah ini yang berjalan pada lingkungan Windows adalah MPICH. Versi terakhirnya adalah MPICH versi 1.2.5, yang tersedia secara bebas di *ftp://ftp.mcs.anl.gov/pub/mpi/nt/mpich.nt.1.2.5.exe*.

Ditinjau dari sisi aplikasi, MPI hanya dapat digunakan dengan model *single program multiple data* (SPMD), sedangkan PVM dapat digunakan dengan model SPMD maupun *multiple program multiple data* (MPMD). Model SPMD secara fisik ditunjukkan dengan program *master* dan *slave* yang menyatu, sedangkan MPMD ditunjukkan dengan program *master* dan *slave* yang terpisah, sehingga *slave* dapat mengerjakan tugas yang berbeda-beda antara satu *node* dengan *node* lainnya.

Arsitektur MPICH ditunjukkan pada Gambar 1. Abstraksi program aplikasi yang dibuat oleh *user* dinyatakan pada lapisan API. Fungsi-fungsi pustaka yang tersedia pada MPI dinyatakan dalam *header mpi.h*. Pengaktifan MPI dimulai dengan menjalankan perintah *MPI_Init(&argc, &argv)*; pada program utama, dilanjutkan dengan menentukan *ranking* dari tiap *node* yang menjalankan program aplikasi dengan perintah *MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)*; *my_rank* adalah bilangan bulat positif, bernilai nol berarti program berjalan pada komputer *master*, sebaliknya bernilai tidak sama dengan nol berarti program berjalan pada komputer *slave*. *MPI_COMM_WORLD* adalah konstanta yang telah



Gambar 1. Arsitektur MPICH [8]

terdefinisi untuk mengendalikan proses-proses yang ada pada saat MPI dimulai. Untuk mengetahui jumlah prosesor (*node*) yang aktif digunakan perintah *MPI_Comm_size(MPI_COMM_WORLD, &p)*.

Komunikasi dilakukan secara berurutan dari lapisan teratas sampai ke lapisan fisik, pada sisi penerima berlaku sebaliknya, yaitu dari lapisan fisik ke atas. Misalnya ada perintah *MPI_Send(message, strlen(message)+1, MPI_CHAR,dest, tag, MPI_COMM_WORLD)*; maka pesan (*message*) sepanjang *strlen(message)+1* dan bertipe karakter (MPI_CHAR) dikirim ke prosesor tujuan (*dest*). Pada lapisan ADI pesan tersebut diterima oleh *Send_handle* yaitu pengendali pengiriman data pada lapisan ADI, kemudian pada lapisan *Channel Device* paket tersebut diterima dengan pengendali *MPID_SendControl* dan *MPID_SendChannel [8]*. Selanjutnya, pada lapisan *low-level device* komunikasi dilakukan dengan protokol yang tersedia. Misalnya MPICH berjalan pada Windows 2000, maka protokol yang digunakan adalah TCP/IP. Pada sisi penerima, lapisan komunikasi atau *channel device interface* memiliki pengendali *MPID_ControlMsgAvail* dan *MPID_RecvAnyControl*. Kedua pengendali ini meneruskan paket ke lapisan ADI. Pada lapisan ini, terdapat dua pengendali, yaitu: *PostedRecv_Handles* dan *UnexpectedRecv_Handles*, masing-masing digunakan untuk mengetahui pengiriman paket yang terkirim dengan benar dan yang salah. Selanjutnya, pada lapisan aplikasi, data diterima dengan perintah *MPI_Recv* (*message, len, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status*); dan untuk mengakhiri MPI digunakan perintah *MPI_Finalize()*; Rincian tentang MPI dapat dilihat di [8,9] dan secara ringkas di [4].

2.3 Java RMI

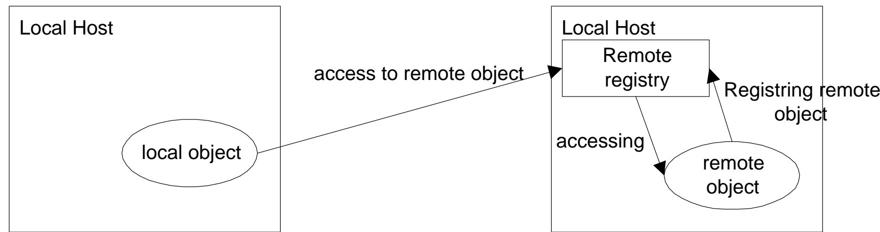
Java Remote Method Invocation (RMI) dibuat berdasarkan ide dasar dari objek lokal dan objek remote. Dalam hal ini kesatuan program yang akan dijalankan disebut sebagai objek. Konsep ini bersifat relatif. Objek lokal adalah objek-objek yang dijalankan pada *host* tertentu. Objek remote adalah objek-objek yang dijalankan pada *host* yang lain. Objek-objek pada *remote host* diekspor (*exported*) sehingga dapat diminta (*invoked*) secara *remote*. Suatu objek diekspor dengan cara meregistrasi dengan suatu *remote registry server*. Server remote registry membantu objek pada *host* yang lain untuk mengakses secara remote objek yang teregister oleh server itu. Hal itu dilakukan dengan memelihara *database* nama dan objek yang terasosiasi dengan nama itu [10].

Objek yang mengekspor dirinya sendiri untuk akses secara *remote* harus mengimplementasi *Remote interface*. *Interface* ini mengidentifikasi objek sebagai dapat diakses secara *remote*. Setiap metode yang diminta secara *remote* harus "membuang" (*throw*) *Remote Exception*. Eksepsi ini digunakan untuk mengindikasi *error* yang terjadi selama proses RMI [10].

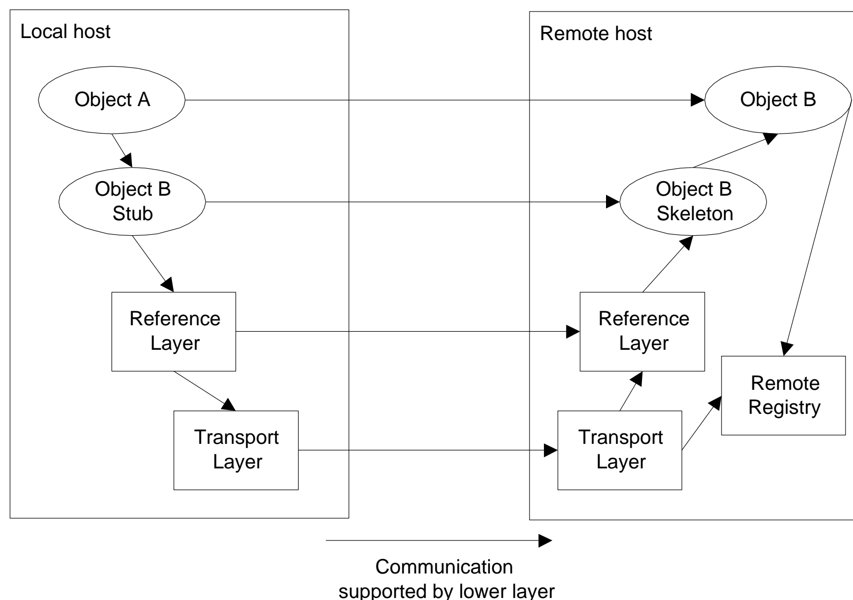
Metode pendekatan RMI pada *Java* (*Java* RMI) adalah objek-objek diorganisasikan dalam suatu *framework* klien/*server*. Suatu objek lokal yang meminta (*invoke*) metode dari objek *remote* dijadikan sebagai *client object* atau *client*. Sebuah objek *remote* yang metodenya diminta oleh objek lokal dijadikan sebagai sebagai *server object* atau *server* [10]

Java RMI menggunakan *stub* dan *skeleton*. *Stub* adalah suatu objek lokal yang berlaku sebagai *local proxy* untuk objek *remote*. *Stub* menyediakan metode yang sama seperti objek *remote*. Objek lokal meminta (*invoke*) metode dari *stub* seakan seperti metode dari objek *remote*. *Stub* kemudian mengkomunikasikan permintaan (*invocation*) metode kepada objek *remote* melalui suatu *skeleton* yang diimplementasikan di *remote host*. *Skeleton* adalah suatu *proxy* pada objek *remote* yang berada pada *host* yang sama dengan objek *remote*. *Skeleton* berkomunikasi dengan stub lokal dan menyampaikan invokasi metode pada *stub* kepada objek *remote* yang sebenarnya. *Skeleton* kemudian menerima nilai yang dihasilkan dari RMI (jika ada) dan menyampaikan nilai itu kembali ke *stub*. *Stub* selanjutnya mengirim nilai tersebut kepada objek lokal yang menginisiasi RMI [10].

Stub dan *skeleton* berkomunikasi melalui *remote reference layer*. *Layer* ini meyediakan *stub* dengan kemampuan untuk berkomunikasi dengan *skeleton* melalui suatu *protocol transport*. RMI biasanya menggunaan TCP untuk transpor informasi, walaupun bersifat fleksibel untuk dapat menggunakan protokol lainnya [10]. Rincian lebih lanjut tentang Java RMI dapat dilihat di [10] dan [11].



Gambar 2. Registrasi dari suatu *remote object* untuk *remote access* [10]



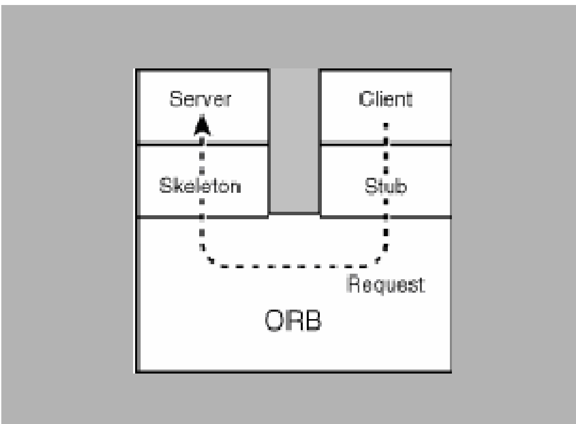
Gambar 3. Java RMI menggunakan *stub* dan *skeleton* untuk mensupport komunikasi klien / server [10].

2.4 **Java CORBA**

Teknologi *Common Object Request Broker Architecture (CORBA)* merupakan teknologi standar yang digunakan untuk melakukan komputasi dengan menggunakan bahasa pemograman yang heterogen [12,13].

CORBA terdiri dari beberapa lapisan (*layer*). *Layer* terendah adalah *Object Request Broker (ORB)*. *ORB* merupakan bahasa netral, yang artinya kita dapat membuat objek dengan menggunakan bahasa pemrograman apapun dan menggunakan ORB untuk dapat mengakses metode-metode yang terdapat pada objek tersebut.

Jika kita menggunakan bahasa pemrograman yang berbeda-beda, maka diperlukan bahasa pemetaan antara bahasa yang digunakan, yaitu dengan menggunakan *CORBA's Interface Definition Language (IDL)*. Ketika kita membuat IDL maka kita akan membuat *stub* dan *skeleton* di dalam bahasa pemrograman yang kita gunakan. *Stub* merupakan *interface* antara klien dan ORB. *Skeleton* merupakan *interface* antara ORB dan objek yang terdapat di *server*. *Stub* dan *skeleton* berkomunikasi melalui ORB.



Gambar 4. Clients *CORBA menggunakan ORB untuk dapat mengakses metode yang terdapat pada server CORBA*.

Gambar ini menunjukkan hubungan antara ORB dan klien yang akan menggunakan metode yang terdapat pada objek di *server*. Rincian lebih lengkap tentang *Java* CORBA dapat dilihat misalnya di [12].

3. Percobaan

Bagian ini membahas persoalan yang akan diuji coba dengan memakai beberapa perangkatbantu. Persoalan yang dibahas adalah persoalan perkalian dua matrix. Kemudian akan dibahas secara ringkat lingkungan SDK yang dipakai, dan bagaimana percobaan itu dilakukan untuk masing-masing perangkatbantu.

3.1 Persoalan yang diuji coba

Meskipun banyak persoalan yang mahal dan harus diatasi dengan komputasi *parallel* untuk mempercepat prosesnya, dalam riset ini adalah persoalan perkalian *matrix* A dan B yang hasilnya disimpan dalam *matrix* C. Yang ingin dilihat adalah bukan jumlah dan jenis persoalan tersebut, namun banyaknya data yang dapat diproses secara bersamaan. Dalam persoalan perkalian *matrix*, proses paralelisasi dapat dirancang dengan beragam jumlah prosesor dan ukuran *matrix* dapat disesuaikan dengan besar data yang ingin diproses.

Secara umum elemen hasil perkalian *matrix* tersebut di definisikan oleh

$$c_{i,j} = \sum_{k=0}^{l-1} a_{i,k} b_{k,j} \tag{2}$$

dimana ***a*** adalah *matrik n x l dan* ***b*** adalah *matrik l x m,* dan $c_{i,j}$ adalah element *matrik* ***c*** yang merupakan hasil perkalian baris ke I dari *matrik* A dengan kolom ke j dari *matrik* B. Perkalian *matrik* sering muncul dalam beberapa aplikasi rekayasa (*engineering*), dimana beberapa bagian aplikasi tersebut banyak tergantung pada persoalan aljabar liner terutama terkait dengan operasi *matrik*.

Dapat dilihat bahwa dengan memberikan beberapa potongan baris *matrix* A yang berbeda dan seluruh elemen *matrix* B ke prosesor yang berbeda, maka prosesor itu akan menghasilkan bagian dari *matrix* C yang berbeda baris dan kolomnya. Karena data yang diproses berbeda, maka masing masing prosesor dapat melakukan perkalian secara serempak (*parallel*) sehingga diharapkan proses makin cepat jika menggunakan jumlah prosesor banyak.

Matrix yang dipakai adalah matrix bujur sangkar, dan besarnya dimulai dari ukuran terkecil hingga ukuran terbesar yang dapat secara optimum diproses pada komputer. Peningkatan besaran *matrik* ini untuk melihat seberapa jauh percepatan dapat dicapai dan seberapa persen proses tambahan (*overhead*) terjadi. *Matrix* yang digunakan adalah *matrix* A dengan elemennya 1 dan *matrix* B dengan elemen –1. Alasan memakai bilangan tipe integer ini adalah agar komputer dapat menyimpan data sebanyak mungkin, sehingga dapat banyak dilihat pengaruh besarnya data dalam proses *parallel*.

Urutan proses yang dilakukan diawali dengan proses 0 inisialisasi *matrik*s, dilanjukan dengan Proses *master* berID 0 mengirim *Matrik*s B dan potongan A ke Proses lain, Proses lain menerima *Matrik*s, Seluruh Proses menghitung perkalian *matrik* berdasarkan data yang diterima, Proses selain 0 mengirim potongan *matrik*s C ke proses 0, Proses 0 menerima potongan *Matrik*s C dan mengumpulkannya sebagai hasil seluruh proses perkalian.

3.2 Lingkungan komputasi.

Percobaan dilakukan pada lingkungan komputasi yang berbeda dengan kumpulan komputer yang berspesifikasi berbeda. Untuk percobaan pertama dengan perangkatbantu masing-masing MPI, JavaRMI dan JavaCorba dilakukan pada komputer dengan Sistem Pengoperasi *WindowXP* dengan *IP address* serta spesifikasi berikut

1. 152.118.27.40 : Pentium IV 3.01 GHz, 256Mb RAM
2. 152.118.27.39 : Pentium IV 3.01 GHz, 256Mb RAM
3. 152.118.27.34 : Pentium IV 2.01 GHz, 256Mb RAM
4. 152.118.27.31 : Pentium IV 2.01 GHz, 256Mb RAM

Percobaan yang membutuhkan satu prosesor dilakukan pada komputer 1; untuk dua prosesor dilakukan pada komputer 1 dan 2. Dan untuk empat prosesor semua

komputer dipakai. Percobaan ini belum mempertimbangkan *load balanching* (penyeimbangan beban) terhadap data yang diproses di antara prosesor, sehingga percobaan dengan dua dan empat prosesor tidak akan terlalu berpengaruh karena spesifikasi komputer 1 dan 2 lebih baik dengan komputer 3 dan 4. Percobaan dengan *load balanching* merupakan bahan kajian tersendiri dan akan dilakukan pada eksperimen berikutnya dan serta akan dilaporkan dalam makalah yang terpisah.

Sedangkan percobaan lainnya dengan perangkatbantu PVM dilakukan pada jaringan empat komputer dengan spesifikasi Prosesor: Pentium 4, 3 Ghz, RAM : 512 MB. Sekilas akan tanpa jelas bahwa percobaan ini jika memakai perangkatbantu yang sama akan memakan waktu yang lebih cepat karena spesifikasi prosesor-nya yang lebih baik. Namun demikian hasil percobaan akan memberikan analisis yang menarik seperti ditayangkan di bawah.

Percobaan dilakukan dengan ukuran matrix berturut-turut, 256, 512, 1024, 1536, dan 2048. Eksekusi dilakukan dengan satu, dua dan empat komputer. Masing-masing eksekusi dilakukan minimum lima kali dan waktu perkalian dan seluruhnya dicatat, dan rata waktu tersebut yang akan dianalisis. Akan dilihat bagaimana pengaruh besar ukuran *matrik* terhadap percepatan relatif terhadap satu prosesor dan waktu *overhead* (tambahan) yang diakibatkan proses inisialisasi dan komunikasi.

3.3 Percobaan dengan MPI dan PVM

Percobaan MPI ini memakai *MPICJH2* versi Win32IA32 yang dapat di*download* dari [9]. Jalankan *executable*, *mpich2-1.0.3-1-win32-ia32.msi*. Perlu diperhatikan bahwa perangkat ini membutuhkan .NET *framework* versi 2.0. Beberapa *variable environtment* perlu disesuaikan, lihat referensi untuk informasi rinci pemasangan. Perlu diingat bahwa MPICH2 harus di*install* ke seluruh *node*/komputer. Kompilasi *file* program yang akan diproses, misalnya ia menghasilkan *file* bernama *matmult.exe*, kemudian *copy*-kan *file* ini ke seluruh *node* dan pastikan ia berada pada direktori yang bernama sama. Masukkan *file exe* dalam *exception firewall*. Sebagai contoh untuk menjalankan proses dengan 2 prosesor/*node* dapat dilakukan pada komputer *master* perintah: *mpiexec –n 2 NIP1 NIP2 matmult.exe*, dimana NIP1 dan NIP2 adalah nomor IP *address* dua komputer yang akan dipakai.

Percobaan PVM diawali dengan membuat *file hostfile* yang disimpan pada komputer *master*. *File* ini berisi daftar node komputer dan nama *user* yang akan dipakai untuk komputasi *parallel*. Bila nama *user* pada semua komputer sama misalnya nama user riset pada komputer C1, C2,C3 dan C4, maka *hostfile* ini boleh tidak ada. Kemudian *daemon* PVM dijalankan pada masing-masing *node*. Daftarkan IP masing-masing komputer pada *file/etc/hosts/hosts.allow* dan */etc/hosts/hosts.equiv*. Penambahan dan penghapusan *host* secara dinamis dapat dilakukan melalui konsole PVM. Program yang akan dijalankan, harus dikompilasi dulu di masing-masing komputer yang akan dilibatkan dan diletakkan pada direktori yang sama. Kompilasi ini tak perlu dilakukan jika spesifikasi prosesor komputernya sama, tinggal *file* hasil kompilasi dari suatu komputer di-*copy*-kan ke komputer lain. Proses dilakukan lewat daemon PVM pada *master* dengan mengeksekusi *file* hasil kompilasi tadi.

Tabel 1. berikut menyajikan waktu percobaan dalam detik untuk beberapa ukuran *matrix* yang berbeda. Untuk mengetahui waktu *overhead*, maka ditampilkan juga waktu proses perkalian dan waktu keseluruhan, sehingga akan nampak selisihnya sebagai waktu *overhead*.

Pada Tabel 1. N menyatakan besarnya *matrik*, NP1, NP2 dan NP4 berturut-turut adalah percobaan dengan satu, dua dan empat prosesor.

Pada tabel itu dengan memperhatikan waktu perkalian dan waktu proses total akan tampak jelas bahwa terdapat waktu *overhead* terutama pada saat ukuran *matriks* masih relative kecil, dan waktu ini relative terabaikan jika ukuran *matrik* sangat besar. Waktu proses perkalian akan sangat mendominasi sehingga waktu *overhead* akan kelihatan kecil sekali. Sementara itu percepatan yang diperoleh empat prosesor tidak terlalu berarti terutama untuk percobaan dengan MPI, ini mengingat pertambahan prosesor yang kinerja kurang baik seakan terabaikan jika dibandingkan

Table 1. Waktu dalam percobaan dengan PVM dan MPI

Percobaan dgn PVM				Percobaan dengan MPI		
Waktu Perkalian saja						
N	NP1	NP2	NP4	NP1	NP2	NP4
256	0.14	0.07	0.03	0.12	0.06	0.05
512	1.67	0.82	0.41	0.97	0.46	0.46
1,024	78.11	39.37	20.32	7.40	3.66	3.62
1,536	275.75	136.85	71.28	24.74	12.34	11.82
2,048	646.70	323.57	165.56	60.29	29.39	28.46
Waktu Proses Total						
256	0.14	0.07	0.03	0.12	0.09	0.11
512	1.67	3.19	4.80	0.97	0.56	0.98
1,024	78.11	44.64	35.06	7.40	4.07	4.33
1,536	275.75	174.15	116.16	24.74	13.20	13.33
2,048	646.70	407.69	223.31	60.29	30.98	31.22

Table 2. Prosentase waktu *Overhead* dan Speed Up

	Percobaan PVM		Percobaan MPI	
N	NP2	NP4	NP2	NP4
	Waktu *Overhead* dalam %			
256	-	-	34	51
512	74	91	19	53
1,024	12	42	10	16
1,536	21	39	7	11
2,048	21	26	5	9
	Speed Up (Percepatan)			
256	2.0	4.5	1.3	1.0
512	0.5	0.3	1.7	1.0
1,024	1.7	2.2	1.8	1.7
1,536	1.6	2.4	1.9	1.9
2,048	1.6	2.9	1.9	1.9

dengan dua prosesor yang lebih baik. Rincian ini dapat dilihat pada Tabel 2. berikut.

Walaupun percepatan yang dicapai percobaan PVM dengan empat prosesor lebih besar dibanding dengan MPI, namun jika dilihat waktu total proses kedua perangkat ini, tampak proses memakai MPI lebih cepat disbanding dengan memakai PVM.

3.4 Percobaan dengan Java RMI dan CORBA

Untuk melakukan percobaan RMI maka pada setiap komputer client/*master* dan server/slave cukup diinstall Java Development Kit (JDK). Lalu perlu disiapkan tiga proses yang berpartisipasi untuk mendukung *remote method invocation* yakni *Proses Client* yakni proses yang meminta sebuah metode pada *remote object*; proses *Server* yakni proses yang memiliki *remote object*; dan *Object Registry*: nama yang digunakan untuk menghubungkan proses *Client* dan server. Server mendaftarkan objek dengan *object Registry*, *client* dapat mencari objek yang sesuai dengan *object Registry*. Kalau kita perhatikan istilah *Server* identik dengan konsep Slave pada MPI dan PVM, demikian juga *Client* identik dengan *Master*.

Sebagai contoh untuk kasus ini, ada beberapa langkah rinci yang perlu dilakukan yakni:

1. mendefinisikan *remote* objek yang terdiri dari definisi *remote* interface dan class yang mengimplementasikan *remote interface*, untuk kasus riset ini dapat dilakukan misalnya dengan berturut-turut membuat *file* terkait yakni ***ComputableMatriks.java*** dan ***CompMatriksImpl.java***.
2. mengkompilasi dan membuat stub. Kompilasi dilakukan terhadap kedua *file* di atas, sedangkan pembuatan stub dilakukan dengan kompilasi memakai rmic (*rmi compiler*). Setelah itu dilakukan pendaftaran (*registring*) *remote* class dengan menjalankan proses registry dari java yakni *start rmiregistry*.
3. mempersiapkan server dalam suatu program server yang melakukan register dengan nama unik. Misalnya *file* yang dibuat adalah ***MatriksServer.java***. Server dilakukan dengan perintah *start java MatriksServer*
4. mempersiapkan client dengan cara membuat program client, misalnya ***MatriksClient.java***, kemudian program client dijalankan.
5. pembagian program/*class* pada komputer *client* dan *server*. Dalam kasus ini yang diletakkan pada masing-masing server adalah *class-class MatriksServer.class*, *ComputableMatriks.class* dan *CompMatriksImpl.Class*. Kemudian dilanjutkan dengan menjalankan *rmiregistry* dan *server*. Sedangkan pada *clients* diletakkan *MatriksClient.class*, *ComputableMatriks.class* dan *CompMatriksImpl_stub. Class*.

Sedangkan untuk percobaan dengan Java CORBA, hanya perlu di*install*, baik pada komputer *master* maupun pada komputer *slave, Java 2 Standard Edition* (J2SE) yang secara *default* telah mengikutsertakan *package* untuk Java CORBA, yaitu *package org.omg.CORBA*.

1. Langkah pertama dalam implementasi aplikasi perkalian *matriks* pada Java CORBA adalah dengan mendefinisikan *interface* antara *client* dan *server*. *Interface* tersebut dituliskan dalam bahasa yang bebas *vendor* sebagaimana telah ditentukan spesifikasi bahasanya oleh CORBA [13]. Penulisan interface ini disimpan dalam suatu *file*, misalnya distMatrix.idl. *File* itu kemudian dikompilasi *dengan* ***#idlj –fall matrix.idl***, dan ini akan menghasilkan *file file* pendukung
2. Kemudian, Implementasi *slave* pada CORBA berupa program. Pada *slave* terdapat sebuah fungsi untuk mengalikan dua buah *matriks*. *Slave* ini hanya menunggu panggilan dari *master* untuk melakukan operasi perkalian dua buah *matriks*. Class yang dibuat pada *slave* adalah sebagai *MatrixSlave.class*, berisi program utama dan MatrixImpl.class, sebuah class yang merupakan subclass dari *MatrixPOA.class* yang dihasilkan proses kompilasi *file* IDL di butir 1.
3. Program *master* berfungsi sebagai pengatur pembagian proses dan dekomposisi data yang akan dikirimkan ke masing-masing *slave.* Implementasi CORBA untuk program *master* ini disebut sebagai *client Dengan mengasumsikan bahwa file program master mempunyai class* ***distMatrixClient.ClientMain,*** maka hasil kompilasi berikut dapat mengimplementasikan proses secara keseluruhan.

4. Kompilasi implementasi Java CORBA
   - Jalankan service orb dengan perintah sebagai berikut: **# orbd –ORBInitialPort <port>**
   - Jalankan *slave-slave* dengan perintah sbb: **# java distMatrix*Server*.*Server*Main**
   - Jalankan *master* dengan perintah sb: **# java distMatrix*Client*.*Client*Main <row> <col> <rowcol>, dimana** <row> adalah ukuran baris *matriks* A, <col> adalah ukuran kolom *matriks* B, dan <rowcol> adalah ukuran kolom *matriks* A/ ukuran baris *matriks* B

Tabel 3. berikut menyajikan waktu percobaan dalam detik untuk beberapa ukuran *matrixs* yang berbeda.

Table 3.
Percobaan dengan perangkat bantu Java RMI dan Corba

Percobaan dgn Java RMI				Percobaan dengan Java Corba		
Waktu Perkalian saja						
N	NP1	NP2	NP4	NP1	NP2	NP4
256	0.43	0.23	0.11	1.70	0.93	0.99
512	3.72	2.40	1.02	15.52	11.11	5.53
1,024	40.78	28.00	12.19	203.10	120.76	52.25
1,536	287.74	138.82	71.21	512.35	257.94	139.74
2,048	1,105.15	489.54	252.70	1,437.39	865.75	463.44
Waktu Proses Total						
256	0.65	0.50	0.42	2.19	1.42	1.49
512	4.01	2.74	1.42	16.17	11.76	6.21
1,024	41.16	28.48	12.75	203.72	121.38	52.86
1,536	288.22	139.53	71.93	512.99	258.61	140.40
2,048	1,105.74	490.24	253.77	1,438.12	866.47	464.16

Table 4. Prosentase waktu *Overhead* dan Speed Up

	Percobaan RMI		Percobaan Corba	
N	NP2	NP4	NP2	NP4
	Waktu *Overhead* dalam %			
256	55	74	34	33
512	13	28	6	11
1,024	2	4	1	1
1,536	1	1	0	0
2,048	0	0	0	0
	Speed Up (Percepatan)			
256	1.3	1.6	1.5	1.5
512	1.5	2.8	1.4	2.6
1,024	1.4	3.2	1.7	3.9
1,536	2.1	4.0	2.0	3.7
2,048	2.3	4.4	1.7	3.1

Untuk mengetahui waktu *overhead*, maka ditampilkan juga waktu proses perkalian dan matriks keseluruhan, sehingga akan nampak selisihnya sebagai waktu *overhead*.

Seperti pada percobaan MPI dan PVM, dengan memperhatikan waktu perkalian dan waktu proses total pada Tabel 3. tersebut akan tampak jelas bahwa terdapat waktu *overhead* terutama pada saat ukuran matriks masih relatif kecil, dan waktu ini relatif terabaikan jika ukuran matriks sangat besar. Waktu proses perkalian akan sangat mendominasi sehingga waktu *overhead* akan kelihatan kecil sekali.

Jika kita perhatikan Table 4, maka nampak terjadi percepaan super linear (melebihi perkiraan dengan jumlah prosesor yang dipakai) pada percobaan dengan RMI dengan ukuran matriks 1,536 dan 2,048 untuk dua prosesor serta ukuran matriks 2048 dengan empat prosesor. Percepatan yang dicapai melebihi jumlah prosesor yang dipakai.

3.5 Analisis kinerja

Jika kita perhatikan table 2 dan table 4, tampak jelas bahwa percobaan dengan Java RMI dan Corba mampu memberikan percepatan yang lebih baik serta waktu *overhead* yang lebih kecil. Namun jika kita perhatikan waktu proses keseluruhan dalam detik, seperti ditampilkan pada table 1 dan table 3, tampak jelas bahwa percobaan dengan MPI memakan waktu paling sedikit dibandingkan lainnya. Kemudian percobaan dengan Java baik RMI dan Corba memakan waktu relative lebih lama dibandingkan dengan MPI dan PVM. Untuk lebih jelasnya, kita bisa lihat Table 5 berikut yang mencatat perbandingan waktu percobaan lainnya dengan MPI. Tabel ini dibentuk dengan cara membagi kolom percobaan tertentu, misalnya PVM dengan np2, dengan kolom percobaan terkait, misalnya, kolom MPI dengan np2. Angka tersebut menyatakan percepatan yang dicapai percobaan MPI dibandingkan dengan percobaan lainnya.

Table 5. Perbandingan waktu percobaan non MPI relatif terhadap MPI

	PVM			JRMI			JCORBA		
N	np1	np2	np4	Np1	np2	np4	np1	np2	np4
256	1.2	0.8	0.3	5.6	5.5	3.7	18.9	15.8	13.3
512	1.7	5.6	4.9	4.1	4.9	1.5	16.7	20.8	6.3
1,024	10.6	11.0	8.1	5.6	7.0	2.9	27.5	29.8	12.2
1,536	11.1	13.2	8.7	11.6	10.6	5.4	20.7	19.6	10.5
2,048	10.7	13.2	7.2	18.3	15.8	8.1	23.9	28.0	14.9

Pada Table 5, semakin besar angka tersebut menyatakan semakin lambat percobaan tersebut dibandingkan dengan percobaan MPI. Walaupun percobaan non Java mempunyai waktu *overhead* dan *speed up* yang kurang baik dibanding dengan percobaan dengan Java, namun percobaan dengan Java relatif lebih lambat dibandingkan dengan percobaan dengan non Java.

4. Kesimpulan

Dari hasil percobaan tersebut maka dapat diambil beberapa kesimpulan bahwa

- Perangkatbantu MPI menunjukkan waktu paling cepat yang kemudian diikuti oleh PVM, Java RMI dan Java CORBA.
- Dari segi pemrograman, implementasi MPI dan PVM jauh lebih mudah dibandingkan dengan Java RMI dan CORBA yang memerlukan definisi *interface* dan proses kompilasi tersendiri.

Penulis memberikan saran bahwa

- Maka untuk aplikasi yang dikembangkan di atas web lebih baik menggunakan Java RMI dan Java CORBA. Khusus untuk aplikasi yang terdiri dari beberapa bagian yang dibuat dalam bahasa pemrograman yang berbeda, sebaiknya memakai Java CORBA
- Untuk aplikasi yang tidak berbasis web terutama engineering sebaiknya memakai MPI atau PVM.

Untuk mendapatkan wawasan yang lebih luas maka perlu dilakukan percobaan yang mempertimbangkan pemerataan beban di antara prosesor dan struktur atau topologi jaringan SDK yang dipakai.

Ucapan Terimakasih

Penulis mengucapkan terima kasih kepada Eko Sediyono yang dalam bimbingan Disertasi S3 bersama penulis menghasilkan review makalah pada bagian 2.1 dan bagian 2.2, Iis Haryono yang dalam bimbingan tugas akhir S1 bersama penulis menghasikan review makalah bagian 2.3 dan 2.4, dan para mahasiswa Magister Ilmu Komputer Universitas Indonesia sebagai pengikut kuliah topik dalam komputasi *parallel* semester genap 2005 – 2006 yang telah turut membantu jalannya percobaan yang dilakukan.

Daftar Acuan

[1] H. Suhartanto, Parallel Iterated Techniques based on Multistep Runge-Kutta Methods of Radau Type, Ph.D Thesis, University of Queensland, Australia, 1998.

[2] Zlatev, Z. and Berkowicz, R. (1988), Numerical treatment of large-sclae air pollutant models, Comput. Math. Applic., 16, 93 – 109.

[3] H. Suhartanto, A. Bustamam, T. Basaruddin, 2003, The Development of Parallel Iterated Multistep Runge-Kutta codes on cluster of workstation, International Conference on CMMSE, Alcante, Spain.

[4] Eko Sediyono, Modifikasi algoritma Triangulasi Deluenay dan implementasi *parallel* pada Sistem Komputasi *Parallel* berbasis jaringan PC, Disertasi, Fakultas Ilmu Komputer UI, 2006.

[5] Iis Haryono, (2004) , Studi Banding Implementasi *Java RMI* dan *Java CORBA* pada Sistem Terdistribusi dengan Kasus Komputasi Numerik, tugas akhir S1, Fakultas Ilmu Komputer UI.

[6] Geist, A., Beguelin,A., Dongara,J. (1994). *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge.

[7] Ataman Software Inc. (1998). "The Ataman RSHD Service User's Manual" [Online]. *Available: http://www.ataman.com* [11/2005].

[8] Protopopov Boris V and Anthony Skjellum. (1998). "A Multi-threaded Message Passing Interface (MPI) Architecture : Performance and Program Issue". *Journal of Parallel and Distributed Computing, vol 2. no.2. July, 1998.*

[9] MPICH2, http://www-unix.mcs.anl.gov/mpi/mpich2/

[10] Jaworski, Jamie, *Java™ 2 Platform Unleashed* (Sams, 1999).

[11] Java Remote Method Invocations, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html, access on July 10, 2006.

[12] Sun Microsystem, Inc. *Java 2 SDK Standard Edition Documentation version 1.4.2*, 2003

[13] Object Management Group, CORBA FAQ, http://www.omg.org/gettingstarted/corbafaq.htm , 2006.