MATA KULIAH KOMPUTASI LANJUT DAN BIG DATA DENGAN DOSEN PENGAMPU PROF. ALHADI BUSTAMAM, P.HD DAN RISMAN ADNAN, M.SI

KOMPUTASI LANJUT & BIG DATA

ALVIN FAIRUZ SANI
FAWWAZ CHIRAG SOFYAN
YANDRA WINDESIA

BUKU KUMPULAN TUGAS
MATA KULIAH KOMPUTASI LANJUT DAN *BIG DATA*

DISUSUN OLEH:

ALVIN FAIRUZ SANI (2206014712)

FAWWAZ CHIRAG SOFYAN (2206163402)

YANDRA WINDESIA (2106779560)

PROGRAM STUDI MAGISTER MATEMATIKA

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM

UNIVERSITAS INDONESIA

DEPOK

2022

KATA PENGANTAR

Bismillahirrahmanirrahim,

Puji syukur penulis panjatkan kehadirat Allah SWT, yang telah memberikan kekuatan dan ketekunan sehingga buku yang telah dipersiapkan sejak September 2022 lalu akhirnya dapat diselesaikan sebagai proyek akhir mata kuliah Komputasi Lanjut dan *Big Data* dengan dosen pengampu Prof. Alhadi Bustamam, Ph.D dan Risman Adnan, M.Si.

Sumber bacaan pokok dari pembuatan buku ini adalah berbagai buku dan jurnal-jurnal kredibel di bidang komputasi dan *big data*. Sehingga diharapkan keberadaan buku ini dapat menjadi panduan dalam penyusunan bahan perkuliahan komputasi dan *big data* bagi mahasiswa departemen Matematika Universitas Indonesia, khususnya bagi mahasiswa program studi Magister Matematika Universitas Indonesia.

Buku ini terdiri dari 4 bagian, bab pertama berisi perbedaan komputasi sekuensial dan komputasi paralel, bab kedua berisi pemaparan arsitektur komputasi paralel, bab ketiga berisi penjelasan studi kasus dan kinerja komputasi paralel, dan bab keempat berisi pemaparan analisis kinerja serta teknik perancangan pada algoritma paralel.

Penulis mengucapkan terimakasih kepada berbagai pihak yang telah membantu memberikan masukan dan perbaikan dalam penyusunan buku ini. penulis juga merasa bahwa buku ini jauh dari sempurna, oleh karena itu segala masukan baik berupa saran maupun kritik yang membangun sangat diharapkan. Akhirnya, semoga tulisan ini dapat bermanfaat bagi siapa saja yang ingin belajar dan mendalami mengenai komputasi dan *big data*.

Depok, 2 Januari 2023

Tim Penulis

DAFTAR ISI

Kata Pengantar ii

Daftar Isi iii

Bab I. Komputasi Sekuensial vs Komputasi Paralel 1

1.1.Pendahuluan 1

1.2 Komputasi Sekuensial 2

1.3 Komputasi Paralel 4

1.4 Kesimpulan 6

Bab 2. Arsitektur Komputasi Paralel 7

2.1 Taksonomi *Flynn* 7

2.2 Arsitektur Memori Komputasi Paralel 7

2.3 Komputasi *Shared Memory* 8

2.4 Komputasi *Distributed Memory* 9

2.5 Komputasi *Graphic Processing Unnit* 9

Bab 3. Studi Kasus dan Kinerja Komputasi Paralel 13

3.1 Arsitektur Komputasi Lanjut 13

3.2 *Tools* Pemrograman Paralel 14

3.3 Menghitung Kinerja dari Komputasi Paralel 27

Bab 4. Analisis Kinerja dan Teknik Perancangan Algoritma Paralel 30

4.1 Pendahuluan 30

4.2 Analisis Kinerja Algoritma Paralel 30

4.3 Implementasi Algoritma Paralel 35

4.4 Teknik Perancangan Algoritma Paralel 43

Daftar Pustaka 47

BAB I
KOMPUTASI SEKUENSIAL VS KOMPUTASI PARALEL

1.1 Pendahuluan

Berdasarkan penelitian yang pernah dikerjakan oleh mahasiswa Gunadarma mengenai pemanfaatan teknologi Nvidia Cuda pada aplikasi kompresi citra menggunakan algoritma DCT 8X8, dipaparkan bahwa dalam proses transmisi citra dibutuhkan guna menghemat bandwith dan penyimpanan data pada harddisk. Sejak Nvidia mengeluarkan teknologi Cuda maka kompresi citra berkembang menjadi proses yang dapat diolah secara paralel, sehingga diharapkan proses kompresi citra dapat menghemat waktu. Penelitian inipun mengungkapkan hasil perbandingan bahwa proses kompresi citra menggunakan komputasi paralel lebih cepat dibandingkan menggunakan komputasi sekuensial.

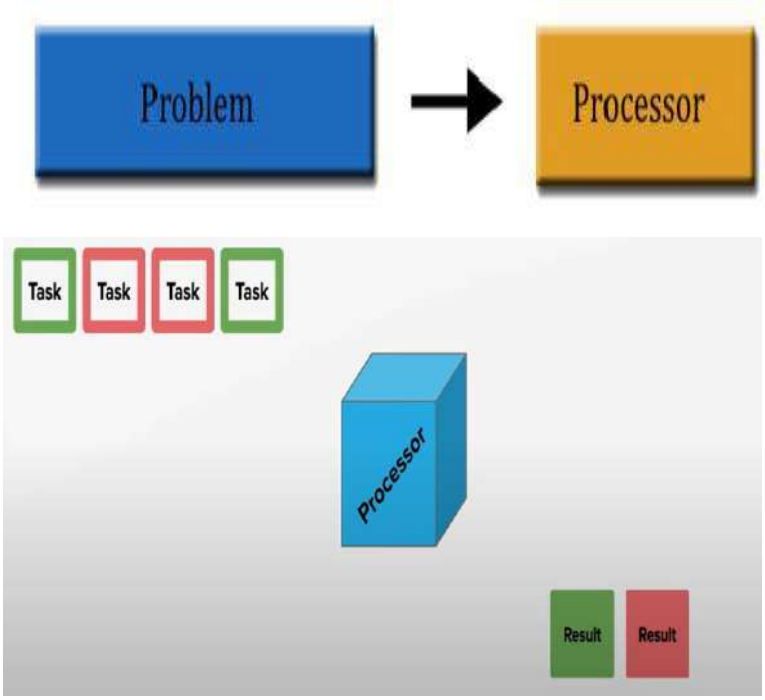
Perbedaan penggunaan komputasi sekuensial dan komputasi lanjut juga pernah diteliti oleh mahasiswa UI dengan penelitian terkait *Ant Colony Optimization* (ACO) yang merupakan salah satu algoritma *swarm intelligence* berdasarkan perilaku sekumpulan semut. ACO dapat digunakan untuk optimisasi pada *travelling salesman*, *quadratic assessment*, dan *routing* pada jaringan. Penggunaan algoritma ACO yang sekuensial ternyata membutuhkan waktu yang cukup lama dibandingkan penggunaan komputasi GPGPU secara paralel.

Berdasarkan dua penelitian di atas menunjukan pentingnya untuk mengetahui perbedaan terkait komputasi sekuensial dan komputasi paralel. Karena diharapkan komputasi sekuensial dan komputasi paralel mampu menyelesaikan masalah optimalisasi dengan waktu singkat dan biaya murah. Bahkan penggunaan komputasi sekuensial dan komputasi paralel juga diharapkan mampu menyelesaikan pekerjaan yang lebih banyak dalam waktu pengerjaan yang sama. Ini relevan dengan perkembangan teknologi komputer saat ini yang menuntut solusi terhadap kebutuhan mulai dari sumber daya komputasi, penyimpanan dan kecepatan. Konsep *high-performance computing* merupakan solusi untuk mengatasi berbagai masalah komputasi dengan menggunakan pendekatan komputasi sekuensial dan komputasi paralel.

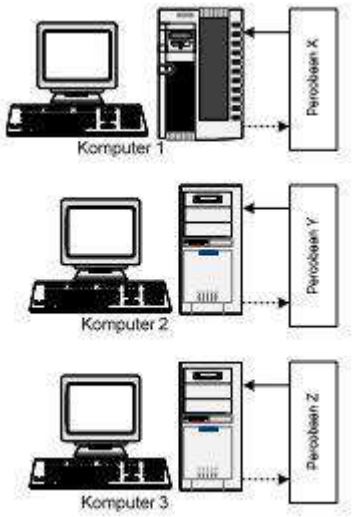
1.2 Komputasi Sekuensial

Komputasi sekuensial atau yang biasa disebut komputasi serial adalah komputasi yang hanya menggunakan satu prosesor untuk mengeksekusi program yang dipecah menjadi urutan instruksi diskrit, masing-masing dieksekusi satu demi satu tanpa tumpang tindih pada waktu tertentu. Ini maksudnya bahwa komputasi sekuensial adalah jenis komputasi dimana satu instruksi yang diberikan dan instruksi berikutnya harus menunggu instruksi pertama dieksekusi. Komputasi sekuensial tergolong metode komputasi tradisional karena semua instruksi dieksekusi secara berurutan. Pada komputasi sekuensial prosesor tunggal yang digunakan tentunya memberikan pendekatan yang lebih sederhana, tetapi secara signifikan terbatas oleh kecepatan prosesor dan kemampuannya untuk mengeksekusi setiap rangkaian instruksi. Mengukur kinerja dalam pemrograman sekuensial jauh lebih kompleks dan penting daripada tolok ukur karena biasanya hanya melibatkan identifikasi kemacetan dalam sistem. Kemampuan untuk menghindari kemacetan ini dengan memindahkan data melalui hierarki memori.

Komputasi sekuensial adalah komputasi yang dicirikan oleh arsitektur bit-sekuensial yaitu beroperasi secara internal pada satu bit. Prosesor dengan perangkat penyimpanan utama sekuensial seperti garis tunda akustik sehingga membutuhkan perangkat keras yang jauh lebih sedikit tetapi jauh lebih lambat. Berbicara mengenai perkembangan komputasi sekuensial, semua komputer digital yang dibuat sebelum tahun 1951 menggunakan algoritma sekuensial. Perancangan bit-sekuensial dikembangkan untuk pemrosesan sinyal digital pada 1960-an hingga 1980-an, termasuk struktur yang efisien untuk perkalian dan akumulasi. Seringkali prosesor seri N akan membutuhkan lebih sedikit area FPGA dan memiliki kinerja total yang lebih tinggi.



Gambar 2.1 Ilustrasi Komputasi Sekuensial



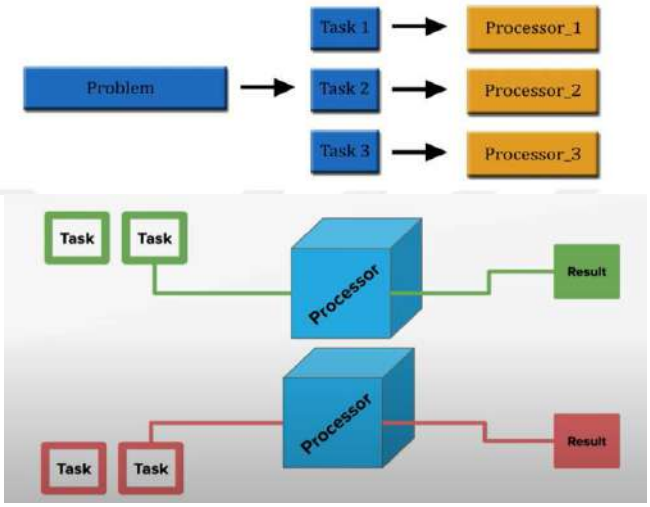
Gambar 2.2 Ilustrasi Prosesor pada Komputasi Sekuensial

Adapun kelebihan dari komputasi sekuensial adalah cenderung mencari *bug* lebih mudah dan tidak butuh *setup*. Sedangkan kekurangan dari komputasi sekuensial adalah cenderung lebih lambat dan hanya menyelesaikan satu tugas dalam satu waktu secara berurutan.

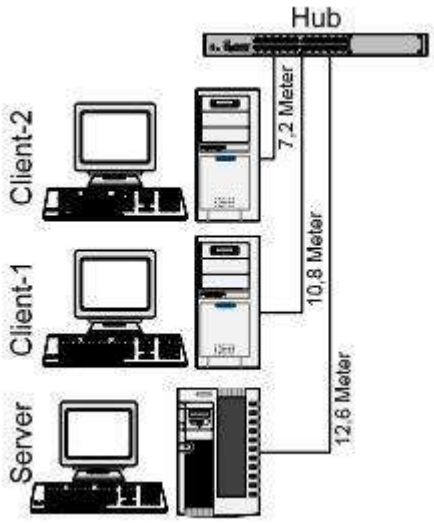
1.3 Komputasi Paralel

Komputasi paralel mengacu pada proses pemecahan masalah besar dan kompleks menjadi masalah yang lebih kecil yang dapat dieksekusi oleh beberapa prosesor yang hasilnya dapat digabungkan sebagai bagian dari keseluruhan algoritma. Oleh karena komputasi paralel didasarkan pada peningkatan ukuran prosesor, hal ini jelas mampu mengurangi jumlah instruksi yang harus dijalankan sistem untuk melakukan tugas pada data berukuran besar. Prosesor 8-bit harus menghitung jumlah dua bilangan bulat 16-bit dengan menjumlahkan 8 bit orde rendah, kemudian menambahkan 8 bit orde tinggi, sehingga membutuhkan dua instruksi untuk melakukan operasi. Tetapi pada komputasi paralel prosesor 16-bit dapat diproses hanya dengan satu instruksi yang dibatasi oleh pola manipulasi data yang tidak teratur dan bandwith memori. Selain itu, pada komputasi paralel sebuah prosesor hanya dapat menangani kurang dari satu instruksi untuk setiap fase siklus clock yaitu siklus yang nilainya naik menuju 1 kemudian mulai turun hingga nilainya nol. Instruksi ini dapat diurutkan ulang dan dikelompokkan yang kemudian dieksekusi secara bersamaan tanpa mempengaruhi hasil program.

Tujuan utama komputasi paralel adalah untuk meningkatkan daya komputasi yang tersedia untuk pemrosesan aplikasi dan pemecahan masalah yang lebih cepat. Data membutuhkan simulasi dan pemodelan yang lebih dinamis dan untuk memecahkan permasalahan tersebut dibutuhkan komputasi paralel yang menyediakan konkurensi dalam menghemat waktu dan uang. Data yang kompleks dan besar serta pengelolaannya dapat menggunakan pendekatan komputasi paralel guna memastikan keefektifan. Beberapa prosesor dijamin dapat digunakan secara efektif.

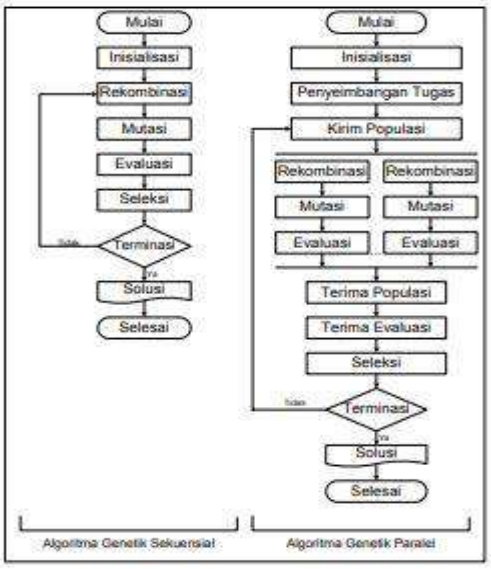


Gambar 2.3 Ilustrasi Komputasi Paralel



Gambar 2.4 Ilustrasi Prosesor pada Jaringan Paralel

Adapun kelebihan dari komputasi paralel adalah lebih cepat dibanding komputasi sekuensial dalam memroses instruksi dan dapat menyelesaikan banyak tugas dalam satu waktu. Sedangkan kekurangan dari komputasi paralel adalah sulit mencari *bug* karena banyak prosesor yang digunakan dan membutuhkan *setup*.



Gambar 2.5 Ilustrasi Perbedaan Pemrosesan pada Algoritma

1.4 Kesimpulan

Adapun perbedaan antara komputasi sekuensial dan komputasi paralel adalah sebagai berikut:

a. Performa yang dihasilkan komputasi paralel lebih baik dibanding komputasi serial.
b. Waktu eksekusi yang dibutuhkan oleh komputasi parallel lebih singkat dibanding komputasi serial.
c. Kapasitas yang diperlukan untuk komputasi parallel sangat besar dibanding komputasi serial.

Komputasi Sekuensial	Komputasi Paralel
Tipe proses untuk satu tugas yang dieksekusi satu per satu sesuai urutan dengan menggunakan satu prosesor	Tipe proses untuk banyak tugas sekaligus dengan menggunakan proses yang berbeda
Hanya ada satu prosesor tunggal	Ada banyak prosesor yang bisa digunakan
Kinerja rendah	Kinerja tinggi
Beban kerja prosesor tinggi	Beban kerja prosesor rendah
Data transfer menggunakan format bit	Data transfer menggunakan format byte (8-bit)
Membutuhkan waktu lebih lama	Membutukan waktu lebih singkat
Biaya lebih murah	Biaya lebih mahal

BAB II
ARSITEKTUR KOMPUTASI PARALEL

2.1 Taksonomi Flynn

Taksonomi Flynn, dalam arsitektur komputer, adalah sebuah klasifikasi yang dibuat oleh Michael J. Flynn pada tahun 1966. Klasifikasi ini dibuat berdasarkan jumlah instruksi yang berjalan simultan dan konkuren, dan juga aliran data yang diprosesnya. Dalam Taksonomi Flynn, komputer dibagi menjadi empat buah kelas, yakni :

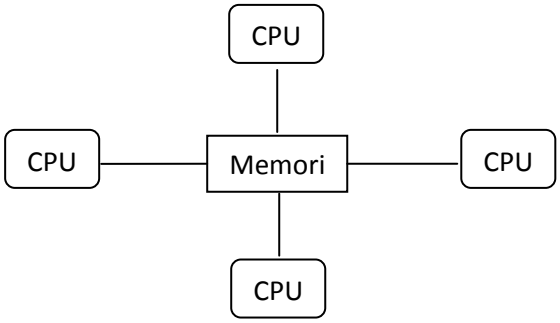
a. Single Instruction Single Data Stream (SISD), yaitu sebuah komputer yang tidak memiliki cara untuk melakukan paralelisasi terhadap instruksi atau data. Contoh mesin SISD adalah PC tradisional atau mainframe yang tua.
b. Multiple Instruction, Single Data Stream (MISD), yaitu sebuah komputer yang dapat melakukan banyak instruksi terhadap satu aliran data. Komputer ini, tidak memiliki contoh, karena meski pernah dibuat, hal itu dibuat sebagai purwarupa (prototipe), dan tidak pernah dirilis secara massal.
c. Single Instruction, Multiple Data Stream (SIMD), yaitu sebuah komputer yang mampu memproses banyak aliran data dengan hanya satu instruksi, sehingga operasi yang dilakukan adalah operasi paralel. Contoh dari SIMD adalah prosesor larik (array processor), atau GPU.
d. Multiple Instruction, Multiple Data stream (MIMD), yaitu sebuah komputer yang memiliki beberapa prosesor yang bersifat otonomus yang mampu melakukan instruksi yang berbeda pada data yang berbeda. Sistem terdistribusi umumnya dikenal sebagai MIMD, entah itu menggunakan satu ruangan memori secara bersama-sama atau sebuah ruangan memori yang terdistribusi.

2.2 Arsitektur Memori Komputer Paralel

a. Shared memory. Prosesor dapat mengaksessemua memory menjadi space alamat global. Shared memory terbagi menjadi dua kelas, yaitu UMA (Uniform Memory Access) dan NUMA (Non-Uniform Memory Access)
b. Distributed Memory. Arsitektur jenis ini memiliki memori local tersendiri, sehingga membutuhkan networking.
c. Hybrid distributed-shared memory. Arsitektur yang menggabungkan kedua arsitektur memori diatas.

Dalam shared memory architecture, semua inti CPU dapat mengakses memori yang sama, seperti beberapa pekerja di kantor yang berbagi papan tulis yang sama, dan semuanya dikendalikan oleh satu sistem operasi. Prosesor modern semuanya adalah prosesor multicore, dengan banyak CPU-core yang diproduksi bersama pada chip silikon fisik yang sama. Sedangkan, dalam distributed memory architecture, kami mengambil banyak komputer multicore dan menghubungkannya bersama-sama menggunakan jaringan, seperti pekerja di kantor yang berbeda berkomunikasi melalui telepon.

2.3 Komputasi *Shared Memory*

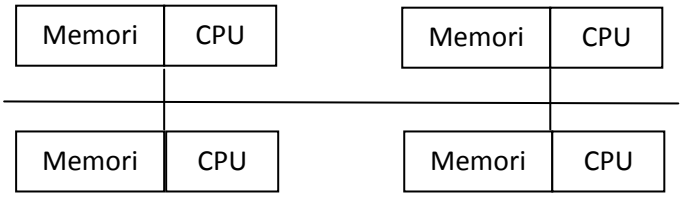


Arsitektur *shared-memory* adalah arsitektur yang menghubungkan beberapa prosesor ke dalam sistem tunggal yang menggunakan memori secara bersama. *Shared-memory* dapat diklasifikasikan menjadi dua jenis, yaitu *uniform memory access* dan *non-uniform memory access*. *Uniform memory access* memiliki karakteristik dimana prosesor dapat mengakses semua memori sebagai ruang secara umum. Miltiprosesor pada *uniform memory access* dapat beroperasi secara mandiri namum dapat saling berbagi memori. Sehingga perubahan di memori yang diakibatkan oleh satu prosesor dapat dilihat oleh prosesor lain yang terhubung di memori yang sama. Setiap prosesor pada *uniform memory access* mempunyai akses dan waktu akses yang sama. *Non-uniform memory access* memiliki karakteristik dimana prosesor memiliki sekumpulan alamat memori sendiri sehingga prosesor dapat mengakses memori lebih cepat. *Non-uniform memory access* biasanya digunakan untuk menghubungkan secara fisik dua atau lebih CPU, dimana satu CPU dapat mengakses memori secara langsung ke CPU yang lainnya. Perbedaan dengan *uniform memory access* adalah pada *non-uniform memory access* tidak semua prosesor mempunyai waktu akses sama ke memori. *Non-uniform memory access* memiliki kelemahan yaitu akses memori lebih lambat.

Shared-memory menyediakan perspektif pemrograman yang lebih menguntungkan ke pengguna terkait penggunaan memori. Pada *shared-memory*, berbagi

data antar perintah dapat dilakukan lebih cepat dan seragam karena dekatnya memori ke CPU. *Shared-memory* memiliki kelemahan adalah tidak terukur artinya menambahkan CPU dapat meningkatkan *traffic* di jalur *shared-memory*. Selain itu, *shared-memory* butuh sinkronisasi yang memastikan akses yang tepat ke memori secara luas. Penambahan jumlah prosesor akan berdampak semakin rumit dan mahal biaya yang dibutuhkan.

2.4 Komputasi *Distributed Memory*



Distributed memory dikenal sebagai Multikomputer yang dapat mereplikasi pasangan prosesor atau memori dan menghubungkannya melalui jaringan interkoneksi. Pasangan prosesor atau memori dikenal sebagai elemen pemrosesan (PE) dan PE bekerja kurang lebih terpisah satu sama lain karena satu PE tidak dapat secara langsung mengakses memori PE lainnya. Pada *distributed memory* setiap prosesor memiliki lokasi memorinya sendiri. Setiap prosesor tidak memiliki pengetahuan eksplisit tentang memori prosesor lain. Karena tidak ada memori bersama, *traffic* bukanlah masalah besar dengan pada *distributed memory*. Secara ekonomis tidak mungkin untuk menghubungkan beberapa prosesor secara langsung satu sama lain. Untuk mencegah banyaknya koneksi adalah dengan menghubungkan setiap prosesor hanya ke beberapa prosesor lainnya. *Distributed memory* dapat menjadi tidak teratur karena tambahan waktu yang dibutuhkan untuk meneruskan pesan dari satu prosesor ke prosesor lainnya di sepanjang jalur pesan.

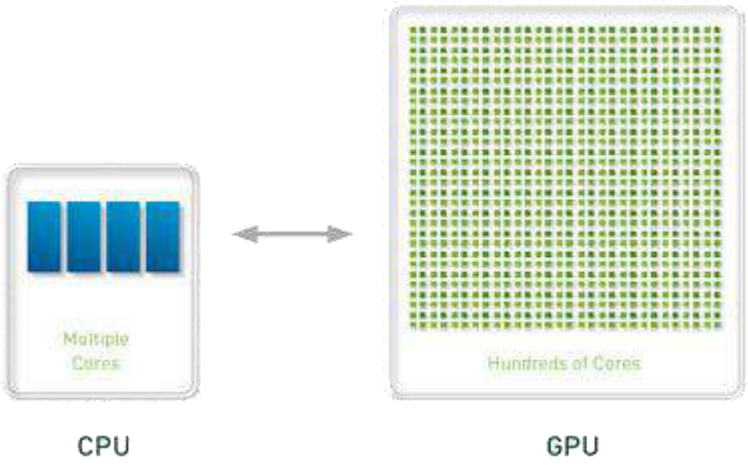
2.5 Komputasi *Graphic Processing Unit* (GPU)

Chip grafis dimulai sebagai pipa grafis dengan fungsi tetap. Selama bertahun-tahun, chip grafis ini menjadi semakin dapat diprogram, yang membuat NVIDIA memperkenalkan GPU pertama. Dalam jangka waktu 1999-2000, ilmuwan komputer, bersama dengan peneliti di bidang seperti pencitraan medis dan elektromagnetik, mulai menggunakan GPU untuk mempercepat berbagai aplikasi ilmiah. Ini adalah munculnya gerakan yang disebut GPGPU, atau komputasi GPU Tujuan Umum.

Tantangannya adalah GPGPU membutuhkan penggunaan bahasa pemrograman grafis seperti OpenGL dan Cg untuk memprogram GPU. Pengembang harus membuat aplikasi ilmiah mereka terlihat seperti aplikasi grafis dan memetakannya ke dalam masalah yang menggambar segitiga dan poligon. Ini membatasi aksesibilitas ke kinerja GPU yang luar biasa untuk sains.

NVIDIA menyadari potensi membawa kinerja ini ke komunitas ilmiah yang lebih besar dan berinvestasi dalam memodifikasi GPU agar sepenuhnya dapat diprogram untuk aplikasi ilmiah. Plus, itu menambahkan dukungan untuk bahasa tingkat tinggi seperti C, C++, dan Fortran. Hal ini menyebabkan platform komputasi paralel CUDA untuk GPU.

Komputasi GPU adalah penggunaan GPU (unit pemrosesan grafis) sebagai co-prosesor untuk mempercepat CPU untuk komputasi ilmiah dan rekayasa tujuan umum.



GPU mempercepat aplikasi yang berjalan pada CPU dengan membongkar beberapa bagian kode yang intensif komputasi dan memakan waktu. Sisa aplikasi masih berjalan pada CPU. Dari sudut pandang pengguna, aplikasi berjalan lebih cepat karena menggunakan kekuatan pemrosesan paralel besar-besaran dari GPU untuk meningkatkan kinerja. Ini dikenal sebagai komputasi "heterogen" atau "hibrida".

CPU terdiri dari empat hingga delapan inti CPU, sedangkan GPU terdiri dari ratusan inti yang lebih kecil. Bersama-sama, mereka beroperasi untuk mengolah data dalam aplikasi. Arsitektur paralel yang sangat besar inilah yang memberi GPU kinerja komputasi yang tinggi. Ada sejumlah aplikasi akselerasi GPU yang menyediakan cara mudah untuk mengakses komputasi kinerja tinggi (HPC).

GPU terintegrasi tidak memiliki kartu terpisah sama sekali dan malah disematkan di samping CPU. GPU diskrit adalah chip berbeda yang dipasang pada papan sirkuitnya

sendiri dan biasanya dipasang pada slot PCI Express. Pada prinsipnya, baik GPU maupun CPU (Central Processing Units) adalah produk dari teknologi yang sama. Di dalam setiap perangkat terdapat prosesor yang terdiri dari jutaan hingga miliaran komponen elektronik mikroskopis, terutama transistor. Komponen-komponen ini membentuk elemen prosesor seperti gerbang logika dan dari sana dibangun menjadi struktur kompleks yang mengubah kode biner menjadi pengalaman komputer canggih yang kita miliki saat ini.

Perbedaan utama antara CPU dan GPU adalah paralelisme. Dalam CPU modern, Anda akan menemukan beberapa inti CPU yang kompleks dan berkinerja tinggi. Empat core adalah tipikal untuk komputer mainstream, tetapi CPU 6- dan delapan-inti menjadi mainstream. Komputer profesional kelas atas mungkin memiliki lusinan atau bahkan lebih dari 100 inti CPU, terutama dengan motherboard multi-socket yang dapat menampung lebih dari satu CPU.

Setiap inti CPU dapat melakukan satu atau (dengan hyperthreading) dua hal sekaligus. Namun, pekerjaan itu bisa berupa apa saja dan bisa sangat kompleks. CPU memiliki berbagai macam kemampuan pemrosesan dan desain yang sangat cerdas yang membuatnya efisien dalam mengerjakan matematika yang rumit.

GPU modern biasanya memiliki ribuan prosesor sederhana di dalamnya. Misalnya, GPU RTX 3090 dari Nvidia memiliki 10496 core GPU. Tidak seperti CPU, setiap inti GPU relatif sederhana dibandingkan dan dirancang untuk melakukan jenis perhitungan yang khas dalam pekerjaan grafis. Tidak hanya itu, ribuan prosesor ini dapat bekerja pada bagian kecil dari masalah rendering grafis secara bersamaan. Itulah yang kami maksud dengan "paralelisme."

Ingatlah bahwa CPU tidak terspesialisasi dan dapat melakukan semua jenis perhitungan, terlepas dari berapa lama waktu yang dibutuhkan untuk menyelesaikan pekerjaan. Faktanya, CPU dapat melakukan apa pun yang dapat dilakukan GPU, hanya saja tidak dapat melakukannya dengan cukup cepat untuk berguna dalam aplikasi grafik waktu nyata.

Jika ini masalahnya, maka kebalikannya juga benar sampai batas tertentu. GPU dapat melakukan beberapa perhitungan yang sama seperti yang biasanya kita minta kepada CPU, tetapi karena mereka memiliki desain pemrosesan paralel seperti superkomputer, mereka dapat melakukannya lebih cepat. Itulah GPGPU: menggunakan GPU untuk melakukan beban kerja CPU tradisional.

Pembuat GPU utama (NVIDIA dan AMD) menggunakan bahasa dan arsitektur pemrograman khusus untuk memungkinkan pengguna mengakses fitur GPGPU. Dalam kasus Nvidia, itu adalah CUDA atau Compute Unified Device Architecture. Inilah sebabnya mengapa Anda akan melihat prosesor GPU mereka disebut sebagai inti CUDA.

Karena CUDA adalah hak milik, pembuat GPU pesaing seperti AMD tidak dapat menggunakannya. Sebagai gantinya, GPU AMD menggunakan OpenCL atau Open Computing Language). Ini adalah bahasa GPGPU yang dibuat oleh konsorsium perusahaan yang mencakup Nvidia dan Intel.

BAB III
STUDI KASUS DAN KINERJA KOMPUTASI PARALEL

3.1 Arsitektur Komputasi Lanjut

Taksonomi Flynn, dalam arsitektur komputer, adalah sebuah klasifikasi yang dibuat oleh Michael J. Flynn pada tahun 1966. Klasifikasi ini dibuat berdasarkan jumlah instruksi yang berjalan simultan dan konkuren, dan juga aliran data yang diprosesnya. Dalam Taksonomi Flynn, komputer dibagi menjadi empat buah kelas, yakni:

1. Single Instruction Single Data Stream (SISD), yaitu sebuah komputer yang tidak memiliki cara untuk melakukan paralelisasi terhadap instruksi atau data. Dalam SISD, instruksi mesin diproses secara berurutan dan komputer yang mengadopsi model ini secara populer disebut komputer sekuensial. Kebanyakan komputer konvensional memiliki arsitektur SISD. Semua instruksi dan data yang akan diproses harus disimpan dalam memori utama. Contoh mesin SISD adalah PC tradisional atau mainframe yang tua.
2. Multiple Instruction, Single Data Stream (MISD), yaitu sebuah komputer yang dapat melakukan banyak instruksi terhadap satu aliran data. Komputer ini, tidak memiliki contoh, karena meski pernah dibuat, hal itu dibuat sebagai purwarupa (prototipe), dan tidak pernah dirilis secara massal.
3. Single Instruction, Multiple Data Stream (SIMD), yaitu sebuah komputer yang mampu memproses banyak aliran data dengan hanya satu instruksi, sehingga operasi yang dilakukan adalah operasi paralel. Mesin berdasarkan model SIMD sangat cocok untuk komputasi ilmiah karena melibatkan banyak operasi vektor dan matriks. Agar informasi dapat diteruskan ke semua elemen pemrosesan, elemen data terorganisir dari vektor dapat dibagi menjadi beberapa set (N-set untuk sistem N elemen pemrosesan) dan setiap elemen dapat memproses satu set data. Contoh dari SIMD adalah prosesor larik (array processor), atau GPU.
4. Multiple Instruction, Multiple Data stream (MIMD), yaitu sebuah komputer yang memiliki beberapa prosesor yang bersifat otonomus yang mampu melakukan instruksi yang berbeda pada data yang berbeda. Sistem terdistribusi umumnya dikenal sebagai MIMD, entah itu menggunakan satu ruangan memori secara bersama-sama atau sebuah ruangan memori yang terdistribusi. Setiap elemen pemrosesan dalam model MIMD memiliki instruksi dan aliran data yang terpisah;

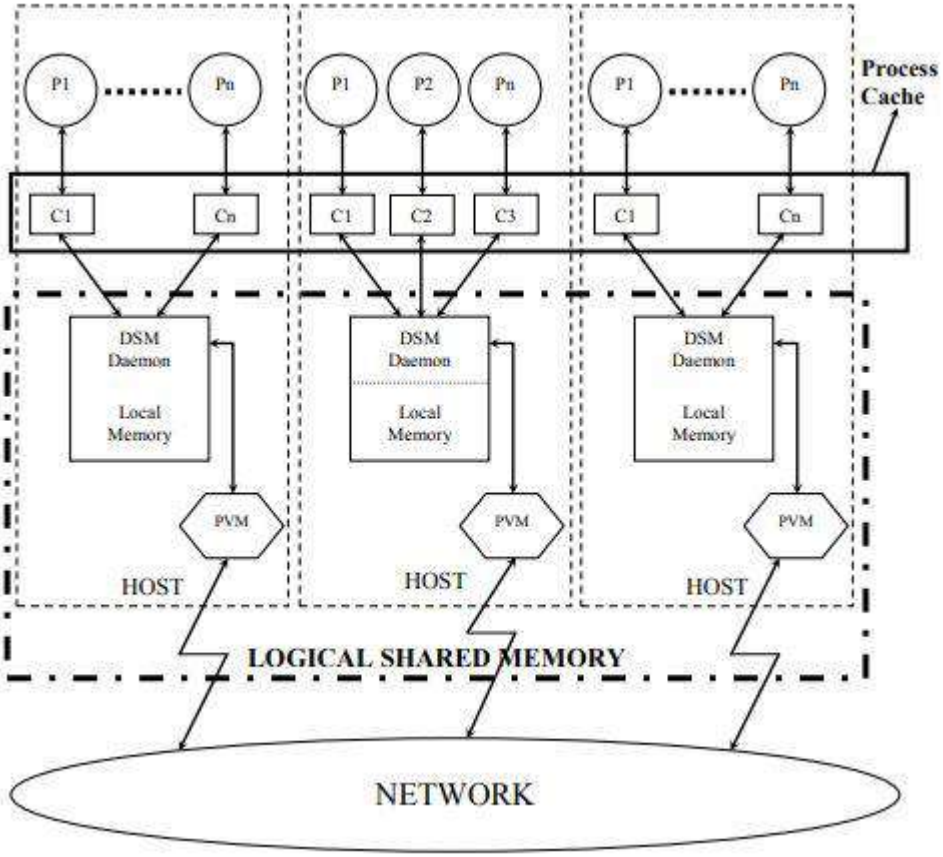
oleh karena itu mesin yang dibangun menggunakan model ini mampu untuk segala jenis aplikasi. Tidak seperti mesin SIMD dan MISD, elemen di mesin MIMD bekerja secara asinkron.

3.2 Tools Pemrograman Paralel

1. Adsmith sebagai Implementasi sistem DSM

Adsmith adalah implementasi distributed shared memory (DSM) berupa library pemrograman pada bahasa C++ (Changhun, 2002: 151-742). Adsmith diimplementasikan untuk berjalan diatas sistem parallel virtual machine (PVM).

Adsmith akan menyediakan pemrosesan paralel dengan implementasi DSM berdasarkan objek dengan cara menyediakan fungsi pemrograman shared memory bagi aplikasi paralel yang berjalan pada sistem komputer cluster yang menggunakan PVM sebagai middleware. Adsmith akan memanfaatkan PVM sebagai subsistem komunikasi, karena sistem PVM banyak digunakan sebagai middleware dari sistem komputer cluster yang bersifat heterogen. Library dari adsmith dan PVM dapat diakses secara bersamaan oleh program aplikasi yang membuat sistem komputer cluster yang menerapkan DSM dengan Adsmith mampu mendukung program aplikasi paralel yang menggunakan fungsi pemrograman message passing dan shared memory.



Arsitektur Adsmith

Adsmith membagi memori lokal dari masing-masing node cluster menjadi dua tingkatan yaitu:

a. Lapisan Logical Shared Memory

Pada lapisan ini terletak bagian dari memori lokal yang menjadi shared memory dimana penggunaannya dapat dilakukan secara bersama dengan proses lain yang terletak pada prosesor lain. Selain itu, terletak daemon dari Adsmith yang berfungsi untuk mengatur shared object yang terletak pada lapisan ini. Setiap daemon akan berkomunikasi dengan daemon yang lain menggunakan PVM untuk menyediakan layanan shared memory bagi program aplikasi. Setiap daemon lokal juga akan memiliki direktori pemetaan data yang akan berisi informasi mengenai status dari shared object pada lapisan Logical Shared Memory ini.

b. Lapisan Process Cache

Lapisan ini akan berisi data yang akan diproses oleh program aplikasi. Lapisan ini memiliki kemiripan dengan cache memory pada hirarki memori komputer.

Data yang dibutuhkan sebuah proses akan dituliskan pada lapisan ini sebelum proses dilakukan, dan setelah proses data selesai maka data akan tetap disimpan pada lapisan ini untuk mengurangi waktu latensi seandainya data ini diperlukan kembali oleh proses. Data tersebut juga dapat dituliskan ke lapisan Logical Shared Memory bila diperlukan.

Pengujian kinerja sistem DSM pada komputer cluster akan dilakukan pada dua sistem yaitu sistem Adsmith dan sistem PVM. Pengujian sistem PVM dilakukan untuk mengetahui kinerja cluster yang tidak mengimplementasikan Adsmith. Pengujian dilakukan pada cluster yang terdiri dari 1, 2, 3 dan 4 node.

Jumlah Node	Spesifikasi Tiap Node
4 Buah	○ Pentium 3- 866 Mhz ○ 256 MB SDRAM ○ 20 GB Hard Disk Drive ○ 3-com 100 Mbps NIC Ethernet Card
1 Buah	○ Nexus 8 port 100Mbps switch

Untuk meningkatkan beban kerja dari cluster parameter input program aplikasi akan ditingkatkan berdasarkan skenario pengujian dengan mengubah parameter input berupa jumlah kota pada program TSP serta jumlah proses slave pada cluster.

Skenario Pengujian	Parameter Input Jumlah Kota	Parameter input proses slave
Skenario 1	16	4
Skenario 2	16	8
Skenario 3	24	4
Skenario 4	24	8

Untuk masing-masing skenario pengujian, diukur waktu eksekusi yang dibutuhkan. Waktu rata-rata yang diperoleh untuk masing-masing skenario dibandingkan dengan waktu rata-rata eksekusi dengan menggunakan satu buah node untuk mendapatkan besarnya speedup.

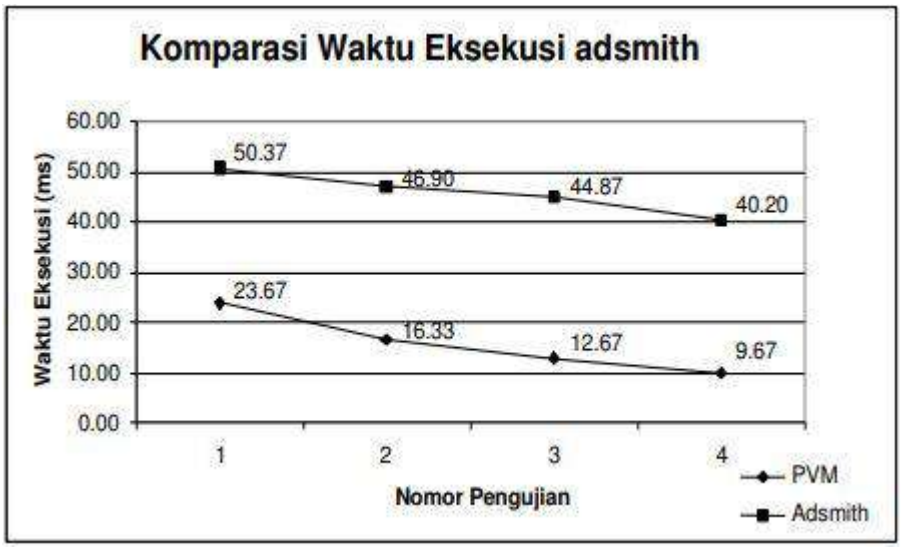
Skenario	Pengujian	Cluster	Waktu Eksekusi(msec)			Rata-rata	Speedup
1	1	1	23	24	24	23.67	1.00
	2	1,2	15	18	16	16.33	1.45
	3	1,2,3	14	12	12	12.67	1.87
	4	1,2,3,4	9	10	10	9.67	2.45
2	1	1	44	43	43	43.33	1.00
	2	1,2	26	26	28	26.67	1.63
	3	1,2,3	20	22	21	21.00	2.06
	4	1,2,3,4	16	16	16	16.00	2.71
3	1	1	31	30	30	30.33	1.00
	2	1,2	26	24	24	24.67	1.23
	3	1,2,3	21	22	21	21.33	1.42
	4	1,2,3,4	15	15	15	15.00	2.02
4	1	1	254	254	255	254.33	1.00
	2	1,2	216	211	207	211.33	1.20
	3	1,2,3	202	202	202	202.00	1.26
	4	1,2,3,4	197	196	196	196.33	1.30

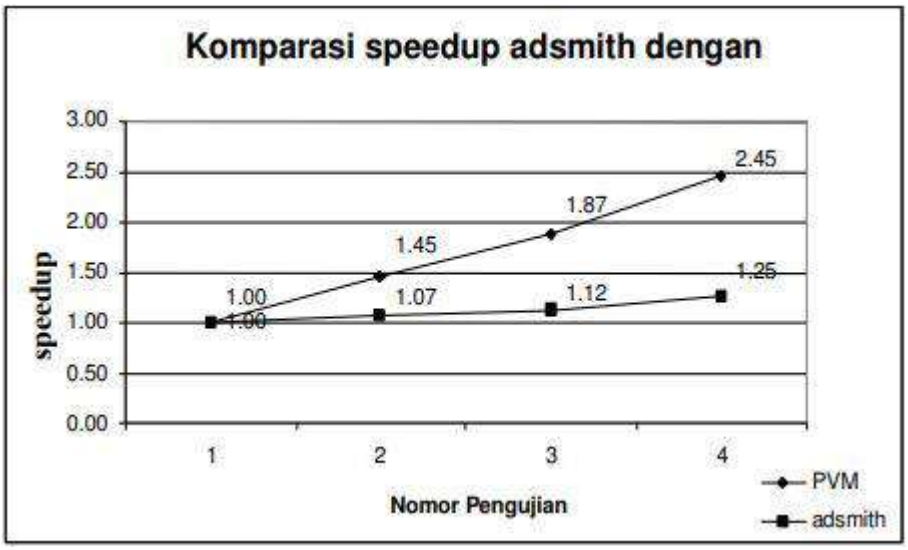
Gambar. Hasil Pengujian Sistem PVM

Tabel. Hasil Pengujian Sistem Adsmith

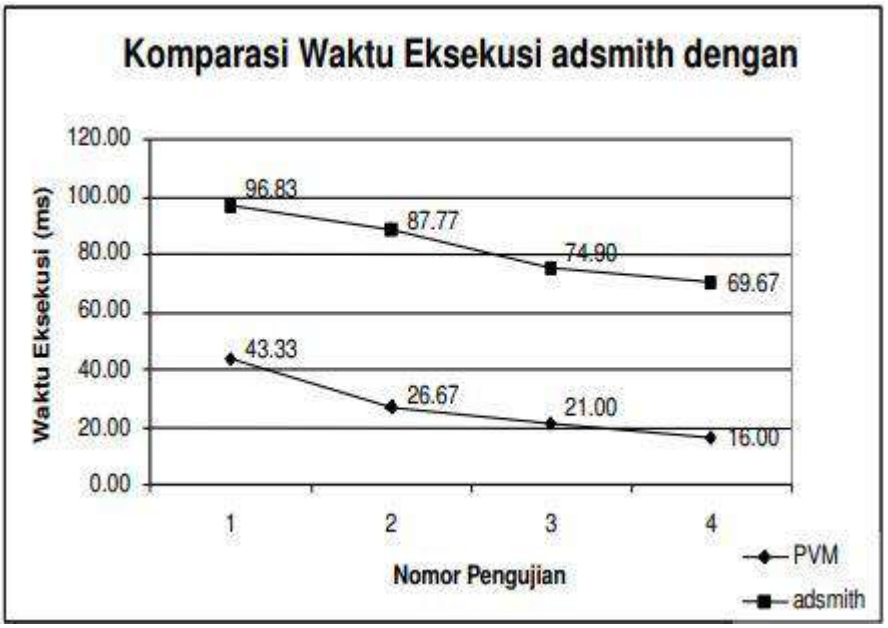
Skenario	Pengujian	Cluster	Waktu Eksekusi(msec)			Rata-rata	Speedup
1	1	1	50.6	50.3	50.2	50.37	1.00
	2	1,2	46.7	47.4	46.6	46.90	1.07
	3	1,2,3	44	44.8	45.8	44.87	1.12
	4	1,2,3,4	39.5	40.1	41	40.20	1.25
2	1	1	97	97	96.5	96.83	1.00
	2	1,2	88	87	88.3	87.77	1.10
	3	1,2,3	75.7	73	76	74.90	1.29
	4	1,2,3,4	69	70	70	69.67	1.39
3	1	1	74	73	74	73.67	1.00
	2	1,2	69	69	70	69.33	1.06
	3	1,2,3	66	65	65	65.33	1.13
	4	1,2,3,4	53	55	55	54.33	1.36
4	1	1	141	139	144	141.33	1.00
	2	1,2	119	118	122	119.67	1.18
	3	1,2,3	91	92	91	91.33	1.55
	4	1,2,3,4	87	87	87	87.00	1.62

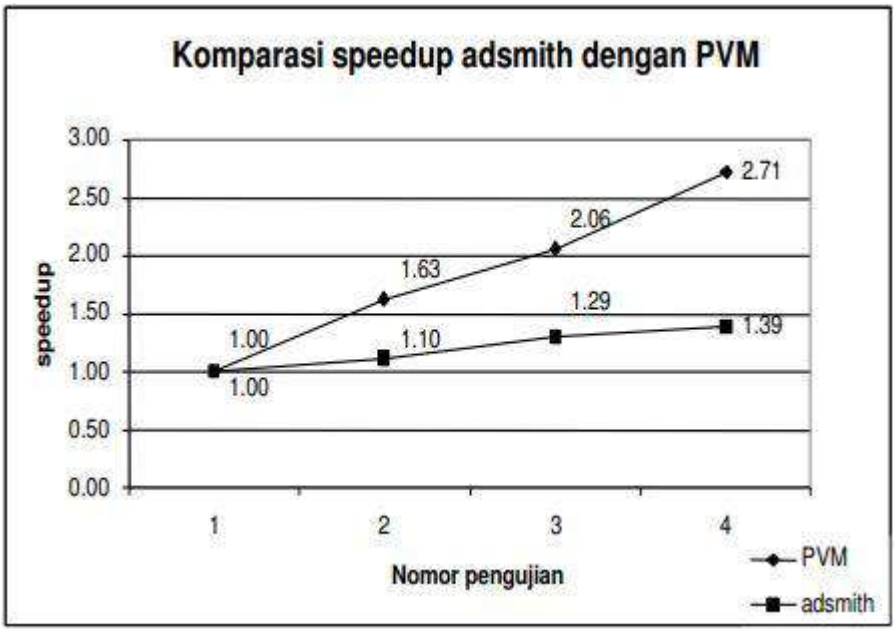
Dari hasil pengukuran dan perhitungan untuk berbagai skenario baik untuk Adsmith maupun untuk PVM, dapat kita gambarkan grafik perbandingan antara kedua sistem.



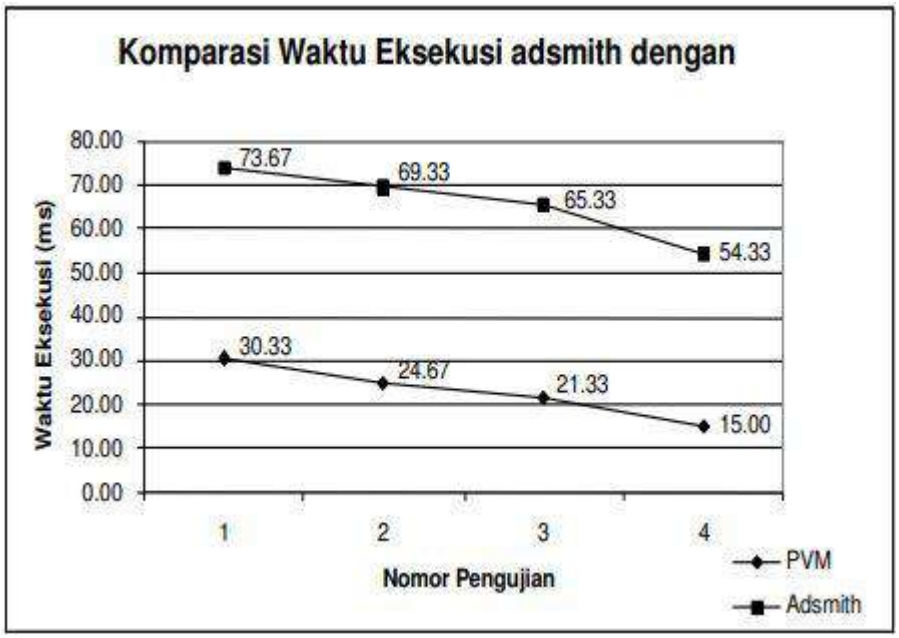


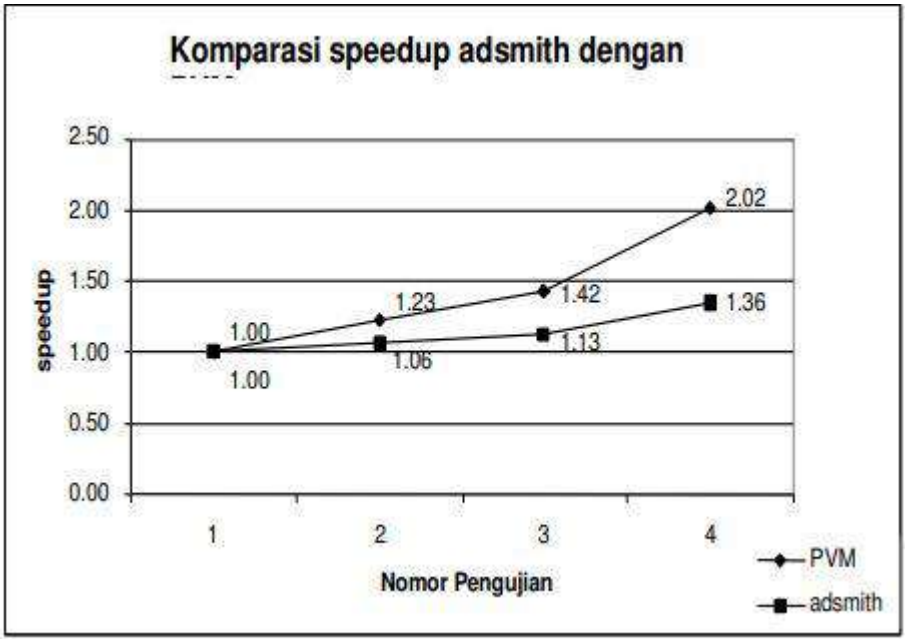
Gambar : Skenario 1



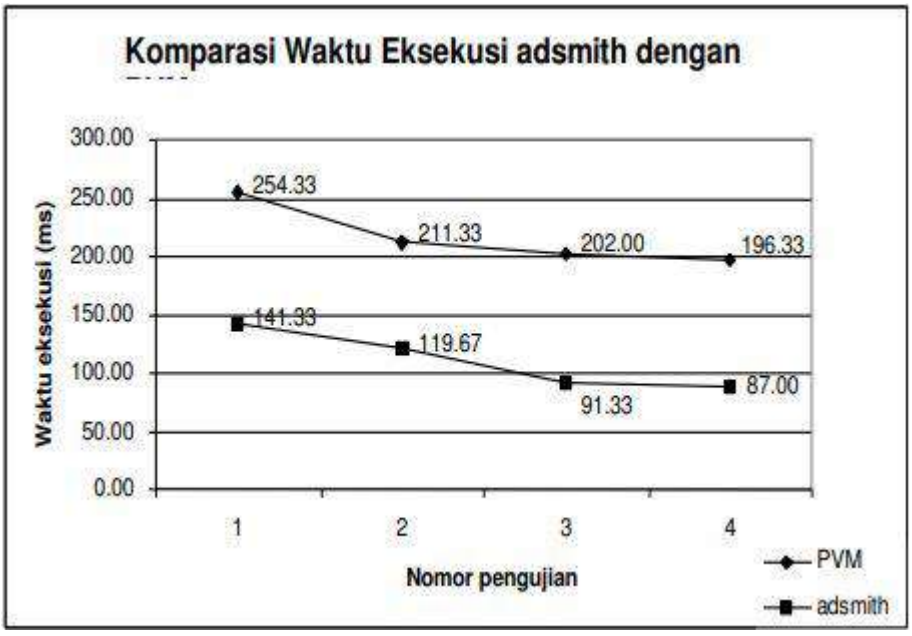


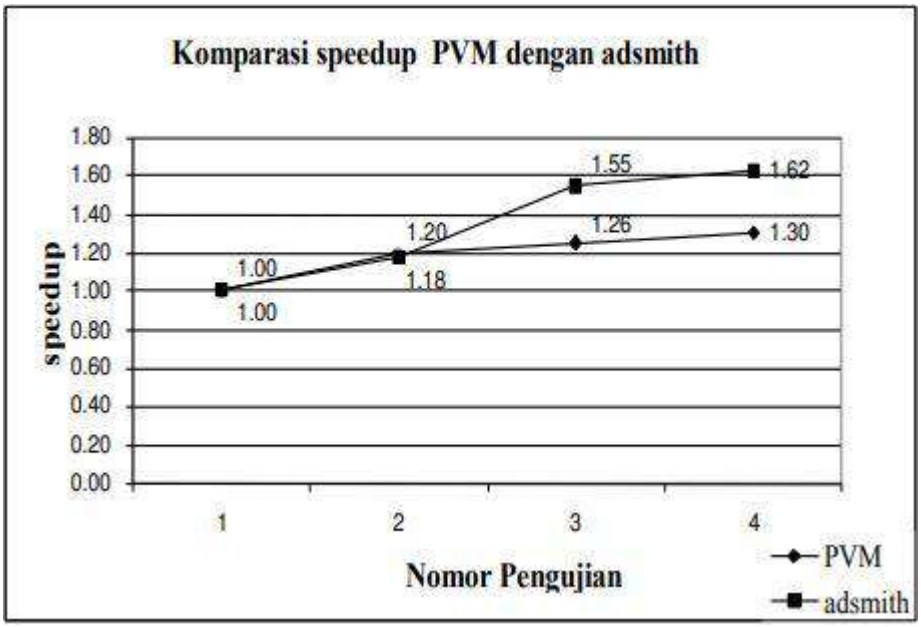
Gambar : Skenario 2





Gambar : Skenario 3





Gambar. Skenario 4

Berdasarkan hasil skenario di atas, dapat disimpulkan sebagai berikut:

a. Kinerja komputer cluster yang menggunakan sistem PVM cenderung lebih baik dibandingkan dengan sistem Adsmith pada beban kerja yang relatif lebih rendah. Hal ini dilihat pada gambar 8, 9 dan 10 diatas dimana waktu eksekusi yang lebih rendah dan tingkat speedup yang lebih tinggi pada sistem PVM. Pada skenario pertama, kedua dan ketiga tersebut tercatat waktu eksekusi sistem PVM lebih cepat rata-rata 30 milli detik, 55.54 milli detik, 42,8 milli detik daripada sistem adsmith, dengan tingkat speedup maksimum – dengan 4 node – sistem PVM rata-rata 2,4 kali dibandingkan dengan sistem adsmith yang rata-rata hanya 1.33 kali (lebih tinggi rata-rata 44.6%).

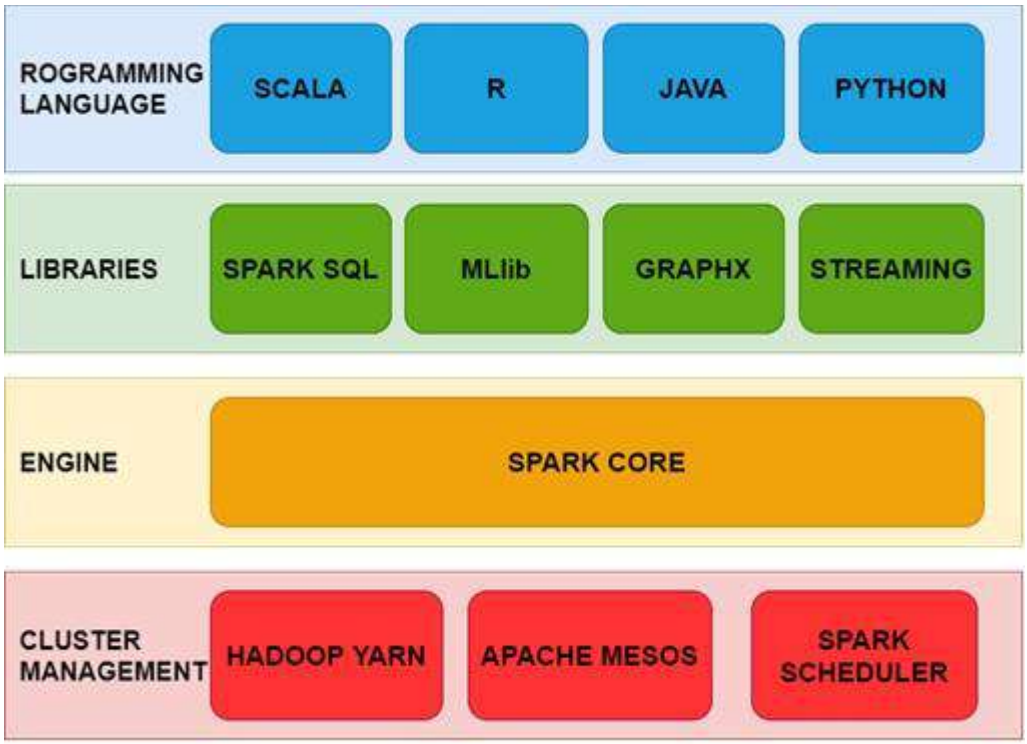
b. Kinerja komputer cluster yang menggunakan sistem Adsmith cenderung lebih baik dibandingkan dengan sistem PVM pada beban kerja yang relatif lebih tinggi. Hal ini dapat dilihat pada gambar 11 dimana parameter input yang diberikan kepada program lebih besar dari skenario yang lain - yang menunjukkan waktu eksekusi yang lebih rendah dan tingkat speedup yang lebih tinggi pada sistem Adsmith. Pada skenario 4 ini sistem Adsmith tercatat rata-rata lebih cepat 106,2 millidetik dibandingkan sistem PVM. Speedup sistem Adsmith terlihat cenderung lebih tinggi pada pengujian ketiga dan keempat skenario keempat dengan perbedaan speedup mencapai 18.7 % untuk pengujian

ketiga dan 19.7 % untuk pengujian keempat masing-masing untuk keunggulan sistem Adsmith.

2. Apache Spark sebagai Implementasi Distributed Memory

Apache Spark merupakan sebuah *framework* atau *environtment* yang dapat digunakan untuk mengakses data dari berbagai sumber berbeda, kemudian mengolah data tersebut, kemudian menyimpannya kedalam penyimpanan data untuk dianalisis. Fitur yang dimiliki oleh *Apache Spark* memungkinkan para data *engineer* untuk membangun sebuah aplikasi pipa pemrosesan *Big Data*. *Apache Spark* adalah *framework* yang digunakan untuk memproses, menanyakan, dan menganalisis *Big Data*. *Apache Spark* melakukan pemrosesan data melalui *in-memory*, sehingga waktu pemrosesan lebih cepat daripada *framework* sejenis seperti *MapReduce* dan lainnya. Perkembangan data dalam tingkat *terabyte* data diproduksi setiap hari, menjadikan kebutuhan akan solusi yang dapat memberikan *real time analysis* dengan kecepatan tinggi.

Apache Spark memiliki beberapa komponen dan dukungan dari berbagai bahasa pemrograman.



Fitur yang ada pada Apache Spark, yaitu sebagai berikut:

a. Spark Core

Spark Core adalah mesin dasar untuk pemrosesan data paralel dan terdistribusi skala besar. *Library* tambahan dapat dibangun di atas *Spark Core* sehingga memungkinkan beragam pemrosesan seperti untuk *streaming, SQL*, dan *Macine Learning* untuk mendukung berbagai aktivitas pemrosesan data. Komponen ini berisi fungsionalitas dasar *Spark* seperti penjadwalan tugas, manajemen memori, interaksi dengan sitem penyimpanan, dll. Tanpa *Spark Core* berbagai *library Spark* lainnya tidak dapat dijalankan pada suatu mesin atau server.

b. Spark SQL

Spark SQL adalah library yang mengintegrasikan pemrosesan data relasional dengan *Spark functional programming API*. *Library* ini mendukung pengolahan data menggunakan kueri, baik melalui *SQL* atau melalui Bahasa Kueri Hive. *Spark SQL* menggunakan antarmuka seperti *SQL* untuk berinteraksi dengan data dari berbagai *format* seperti *CSV, JSON, Parket,* hingga ke berbagai *database engine* seperti *MySQL* dan *SQL Server*.

c. MLlib

MLlib adalah *library* yang berisi berbagai macam Algoritma *Machine Learning* yang ditawarkan oleh *Spark. MLib* menyediakan berbagai *function* yang dapat dipanggil untuk melakukan pembelajaran *Supervised* maupun *Un-supervised, Regression* maupun *Classification*. *Library* ini dapat memenuhi kebutuhan analisis untuk melihat pola tersembunyi dari data yang ada, setelah data dari berbagai sumber didapatkan dan diolah.

d. GraphX

Library ini adalah *API Apache Spark* untuk menjalankan komputasi grafik secara paralel. *Library* ini dapat mengolah data yang tersimpan dalam format *RDD (Resilient Distributed Dataset),* kemudian membuat grafik yang memiliki arah pada setiap *vertex* dan *edge*. Setiap *vertex* dan *edge* dapat memiliki properties seperti nama, *cost* atau jarak, arah, dan menyimpan informasi lainnya yang dibutuhkan.

e. Spark Steaming

Komponen ini merupakan komponen dari *Spark* yang dapat digunakan untuk memproses data secara *real time streaming. Streaming* memungkinkan data yang masuk, dapat diolah dan disimpan secara *real time*. *Spark*

Streaming memberikan fasilitas untuk aplikasi pemrosesan data yang dibangun menggunakan *Spark*, dapat berjalan secara *real time. Spark Streaming* akan berkomunikasi dengan sumber data dan tempat penyimpanan data. Sumber data dapat terdiri dari satu atau beberapa *server* atau aplikasi berbeda, kemudian *Spark* menerima data dari sumber tersebut, lalu mengolahnya dan mengirim data hasil olahan menuju tempat penyimpanan data secara terus menerus (tidak terputus).

Apache Spark merupakan *tools Big Data* yang sangat berguna untuk membangun jalur pemrosesan data dengan mudah, didukung oleh beberapa jenis bahasa pemrograman dan menyediakan berbagai *library* yang dapat memenuhi kebutuhan pemrosesan data. Kita dapat mengakses hingga *petabyte* data dari berbagai sumber pengimpanan berbeda dan memprosesnya secara cepat dengan menyiapkan beberapa *node server* yang terinsall *framework Apache Spark*. *Apache Spark* juga dilengkapi dengan library untuk memenuhi kebutuhan analisis data seperti *Graph X* untuk komputasi grafik, dan *MLlib* untuk memenuhi kebutuhan pengolahan data menggunakan *machine learning.* Eksekusi dari aplikasi yang dibangun menggunakan *Spark* dapat mendukung pemrosesan data secara *real time,* sehingga dapat digunakan untuk membangun pipa pemrosesan *Big Data* dari berbagai sumber data, menuju penyimpanan data secara terus menerus.

Implementasi jurnal: "IMPLEMENTASI APACHE SPARK PADA BIG DATA BERBASIS HADOOP DISTRIBUTED FILE SYSTEM"

3. Apache Cordova sebagai Implementasi Hybrid Memory

Apache Cordova adalah framework yang digunakan untuk pengembangan aplikasi seluler yang dapat digunakan untuk membuat aplikasi seluler lintas platform menggunakan HTML5 dan JavaScript murni. Dengan lintas platform, basis kode aplikasi dapat ditulis satu kali menggunakan HTML5 dan JavaScript yang dapat dijalankan di beberapa platform seluler target seperti Android, iOS, atau Windows seluler. Biasanya, aplikasi berbasis web dijalankan dalam sandbox, artinya aplikasi tersebut tidak memiliki akses langsung ke berbagai fitur perangkat keras dan perangkat lunak pada perangkat digital yang biasa digunakan. Database yang berisi nama, nomor telepon, email, dan informasi lainnya tidak dapat diakses oleh aplikasi web. Selain menyediakan framework dasar untuk menjalankan aplikasi web dalam

aplikasi native, Cordova juga menyediakan API JavaScript untuk memungkinkan akses ke berbagai fitur perangkat, seperti database kontak.

Kemampuan dari framework ini dapat diekspos melalui penggunaan kumpulan plugin. Plugin menyediakan jembatan antara aplikasi web dan fitur asli perangkat. Tim Ionic telah melangkah lebih jauh dan menciptakan Ionic Native dengan interface TypeScript untuk lebih dari 200 plugin yang paling umum dan versi yang didukung perusahaan untuk tim yang menginginkan manfaat dari Ionic yang mengelola pembaruan yang sedang berlangsung, patch keamanan, kompatibilitas dan aspek lainnya akses perangkat asli.

Saat aplikasi ini diluncurkan pertama kali, Apache Cordova memuat halaman awal aplikasi default (biasanya index.html) di WebView aplikasi dan meneruskan kontrolnya ke WebView. Pada WebView ini pun memungkinkan pengguna untuk berinteraksi dengan aplikasi dengan memasukkan data di bidang input, mengklik tombol tindakan, dan melihat hasil di WebView aplikasi. Untuk mengakses fungsionalitas asli seluler seperti audio atau kamera, Cordova menyediakan paket API JavaScript yang dapat digunakan oleh pengembang dari kode JavaScript mereka. Panggilan ke Cordova JavaScript API akan diterjemahkan ke dalam panggilan API perangkat asli dengan menggunakan lapisan jembatan khusus. API asli dapat diakses dari plugin Apache Cordova. Beberapa fungsi dari Apache Cordova, yaitu sebagai berikut:

a. Fungsi Apache Cordova : Pembuatan Aplikasi Mobile

Fungsi utama yang dimiliki oleh aplikasi Apache Cordova adalah untuk menunjang proses pembuatan aplikasi berbasis mobile. Jadi bisa dikembangkan suatu aplikasi pada platform mobil dengan memanfaatkan platform Apache Cordova tersebut. Aplikasi mobile yang dibuat tersebut akan bisa diluncurkan pada beberapa jenis OS perangkat ponsel pintar. Mulai dari Android, iPhone, dan bahkan bisa juga untuk Windows Seluler. Nantinya, bisa melakukan *deploy* aplikasi sebelum diluncurkan pada perangkat *smartphone*.

b. **Mengakses Device Pada Perangkat Mobile**

Apache Cordova merupakan perangkat lunak komputer yang berisi API (*Application Programming Interface*) yang bisa untuk mengakses devide dari perangkat mobile. Beberapa jenis device yang bisa diakses dari *smartphone* tersebut antara lain adalah kamera, GPS, storage, dan lain sebagainya. Jadi dengan platform tersebut, proses pengembangan aplikasi akan bisa lebih

maksimal. Sebab nantinya aplikasi tersebut akan bisa langsung terhubung dengan berbagai jenis device pada *smartphone*. Hal ini tentunya akan membuat proses pembuatan aplikasi dan ataupun tampilan web menjadi semakin maksimal.

Implementasi Jurnal: "Implementasi Framework Cordova sebafai solusi pengembangan aplikasi cross-platform"

3.3 Menghitung Kinerja dari Komputasi Paralel

1. Isoefisiensi

Ada sejumlah metrik yang dapat digunakan untuk mengukur kinerja suatu sistem paralel. Metrik yang sering digunakan adalah peningkatan kecepatan atau yang lazim disebut sebagai speedup dan efisiensi prosesor. Speedup adalah perbandingan antara waktu eksekusi algoritma sekuensial tercepat dengan waktu eksekusi algoritma paralel, atau waktu eksekusi algoritma paralel pada satu prosesor dengan waktu eksekusi algoritma tersebut pada sejumlah prosesor. Secara matematis speedup dinyatakan dengan Persamaan :

$$S_p = \frac{T_1}{T_p} \dots (1)$$

Speedup pada satu prosesor adalah sama dengan satu, dan speedup pada p prosesor bernilai $1 \leq Sp \leq p$. Secara ideal speedup meningkat sebanding dengan bertambahnya jumlah prosesor. Jadi jika digunakan p prosesor, speedup idealnya adalah p. Dalam beberapa kasus dapat terjadi superlinear speedup ($Sp > p$). *Superlinear speedup* tidak mencerminkan peningkatan kinerja yang sebenarnya, karena peningkatan kecepatan ini diakibatkan oleh fitur unik dari arsitektur paralel, misalnya ukuran cache yang lebih besar pada lingkup pemrograman paralel dibandingkan dengan ukuran *cache* pada lingkup pemrograman sekuensial.

Efisiensi merupakan suatu ukuran kinerja yang sangat erat hubungannya dengan *speedup*. Secara matematis efisiensi dinyatakan dengan :

$$E = \frac{S_p}{p}, dengan\ \frac{1}{p} \leq E \leq 1 \dots (2)$$

Speedup dan efisiensi merupakan fungsi dari ukuran data dan jumlah prosesor. Umumnya *speedup* tidak akan meningkat secara linear dengan meningkatnya jumlah prosesor, melainkan cenderung mencapai suatu titik jenuh. Dengan perkataan lain, efisiensi akan menurun jika jumlah prosesor meningkat. Nilai *speedup* dan efisiensi

yang tidak ideal ini dikarenakan adanya *overhead* pada sistem paralel, misalnya komputasi tambahan yang hanya dibutuhkan pada sistem paralel, komunikasi antar-prosesor, dan proses sinkronisasi. Hal ini berlaku untuk semua sistem paralel, dan gejala saturasi dari speedup dan efisiensi ini mengikuti Hukum Amdahl.

Namun, *speedup* dan efisiensi akan meningkat jika ukuran data juga ditingkatkan, hal ini sesuai dengan Hukum Gustafson. Jika peningkatan jumlah prosesor akan menurunkan efisiensi, dan peningkatan ukuran data akan meningkatkan efisiensi, berarti efisiensi dapat dibuat konstan jika jumlah prosesor dan ukuran data sama-sama ditingkatkan. Efisiensi yang konstan ini disebut sebagai isoefisiensi. Kinerja suatu sistem paralel dapat dianalisis dengan menggunakan fungsi isoefisiensi. Sistem paralel yang memiliki fungsi isoefisiensi sebesar $\Theta(p)$ dikatakan sebagai suatu sistem paralel yang *scalable* (*scalable parallel system*)

2. Scalable Parallel System

Untuk memperoleh efisiensi yang konstan, ukuran data perlu ditingkatkan seiring dengan meningkatnya jumlah prosesor. Peningkatan ukuran data akan berbeda untuk sistem paralel yang berbeda, dan peningkatan ini dipengaruhi oleh berbagai faktor, seperti pendistribusian data antarprosesor dan kanal komunikasi pada suatu sistem paralel.

Skalabilitas sistem paralel dapat dianalisis dengan menggunakan suatu fungsi yang menghubungkan ukuran data dengan $overhead$ dari sistem paralel. Fungsi ini dinamakan fungsi isoefisiensi. Suatu sistem paralel yang $scalable$ adalah suatu sistem paralel yang mampu mempertahankan kinerjanya berdasarkan suatu metrik, seiring dengan peningkatan jumlah prosesor.

Jika T_1 dan T_p masing-masing menyatakan waktu eksekusi pada satu dan p prosesor, maka $overhead$ sistem paralel dapat dinyatakan dengan :

$$T_o = pT_p - T_1 \dots (3)$$

atau

$$T_p = \frac{T_1 + T_o}{p} \dots (4)$$

Karena waktu eksekusi pada satu prosesor hanya ditentukan oleh beban kerja pada prosesor, maka $T_p = W$, sehingga Persamaan (4) dapat ditulis sebagai :

$$T_p = \frac{W + T_o}{p} \dots (5)$$

Dengan mensubstitusikan Persamaan (5) ke Persamaan (1) dan Persamaan (2), diperoleh :

$$S = \frac{W \times p}{W + T_o} \dots (6)$$

$$E = \frac{1}{1 + T_o/W} \dots (7)$$

Dengan mendefinisikan suatu $konstanta\ K = E\ /(1 - E)$, maka Persamaan (8) dapat dinyatakan dengan :

$$W = KT_o \dots (9)$$

Persamaan (9) merupaka fungsi yang menunjukkan seberapa besar peningkatan beban kerja W yang dibutuhkan seiring dengan peningkatan jumlah prosesor untuk mempertahankan efisiensi sistem. Fungsi ini disebut sebagai fungsi isoefisiensi. Suatu fungsi isoefisiensi yang kecil menunjukkan bahwa untuk mempertahankan efisiensi sistem seiring dengan bertambahnya prosesor, hanya dibutuhkan sedikit peningkatan beban kerja, atau dengan perkataan lain sistem tersebut adalah suatu *scalable parallel system*.

BAB IV
ANALISIS KINERJA, CONTOH KASUS, SERTA TEKNIK PERANCANGAN PADA ALGORITMA PARALEL

4.1 Pendahuluan

Dalam pemecahan masalah yang kompleks seringkali komputasi sekuensial kurang memadai, terutama dalam hal waktu eksekusi yang sangat lama sehingga menjadi demikian tidak fisibel. Alternatif yang menarik untuk mengatasi waktu komputasi yang lama adalah penggunaan komputasi paralel. Komputasi kinerja paralel dapat diukur dengan berbagai metrik, yaitu dari kinerja (*time complexity, speed up, efficiency*) ataupun skalabilitas. Contoh kasus yang menggunakan komputasi paralel diantaranya adalah menggunakan algoritma pengurutan (*sorting*) beserta implementasinya seperti *mergesort* dan *quicksort*. Terdapat juga beberapa teknik perancangan yang dapat dilakukan, diantaranya arsitektur von neumann, arsitektur risc dan cisc , arsitektur data flow, arsitektur systolic array.

4.2 Analisis Kinerja Algoritma Paralel

1. Time Complexity

Untuk persamaan

$$T(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

maka big-oh dari $T(n)$ adalah $O(n^k)$. Untuk persamaan

$$T(n) = 2^n + n^3$$

maka big-oh dari $T(n)$ adalah $O(2^n)$.

$T(n)$ adalah fungsi yang diturunkan dari algoritma yang akan diukur big-oh nya dan n adalah variabel dalam algoritma yang paling dominan dalam menentukan persamaan tersebut. Kompleksitas waktu adalah big-oh dari persamaan dengan variabel yang dominan dalam algoritma dimana persamaan ini menyatakan jumlah langkah yang harus dilalui oleh algoritma tersebut pada kondisi terjelek.

```
begin
for (i=0;i<m;i++)
   for (j=0;j<n;j++)
      { p[i,j]=q[i,j]+r[i,j];
      }
   end
end
```

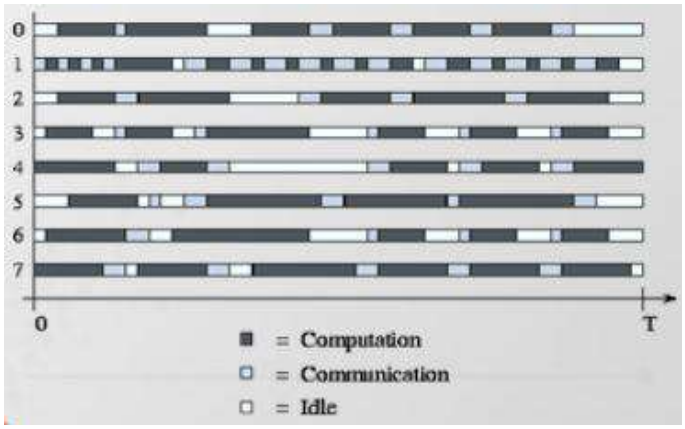
Variabel yang dominan adalah m dan n, sehingga kompleksitas waktu nya adalah $O(mn)$.

```
SUM(EREW PRAM)
Initial condition: List of n>=1 elements stored in A[0...(n-1)]
Final condition: Sum of elements stored in A[0]
Global variables: n, A[0...(n-1)],j
begin
spawn (P0,P1,P2,...,P((n/2)-1)
   for all Pi where 0<=i<=[n/2]-1 do
   for j<-0 to [log n]-1 do
      if i modulo 2^j=0 and 2i+2^j<n then
         A[2i]<-A[2i]+A[2i+2^j]
      endif
    endfor
    endfor
end
```

Maka $T(n) = \log\left(\frac{n}{2}\right) + konstan + konstanxlog(n)$ dan kompleksitas waktunya adalah $O(\log n)$. *Lower bound* adalah kompleksitas waktu tercepat secara teoritis yang bisa dicapai oleh suatu algoritma. Sedangkan *upper bound* adalah kompleksitas waktu tercepat yang dapat dicapai oleh pembuat algoritma.

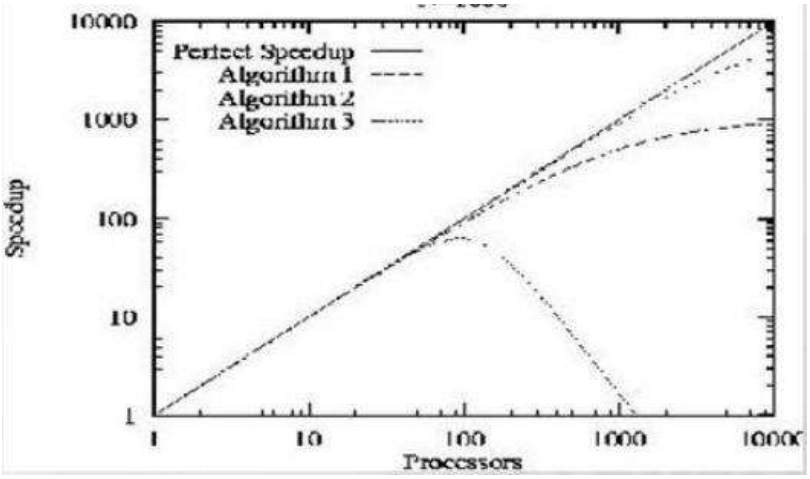
2. *Speed-Up*

Sebuah analogi, jika 8 prosesor sedang bekerja paralel, maka masing-masing prosesor mempunyai kemungkinan fungsi yaitu mampu menjalankan komputasi, mengolah informasi, dan tidak melakukan apapun (*idle*).



Gambar 1. Analogi *Speed-up*

Secara intuisi, dapat dikatakan bahwa jumlah prosesor makin banyak, maka tidak selalu menjamin bahwa waktu komputasi menjadi makin cepat. Cara penyelesaian algoritma dan jenis beban kerja dari algoritma tersebut menentukan kecepatan dari komputasi.



Gambar 2. Kinerja *Speed-up* Berdasarkan Jumlah Prosesor dan Algoritma Berbeda

Perhitungan *speed-up* adalah sebagai berikut:

$$speedup = \frac{kompleksitas\ waktu\ sekuensial}{kompleksitas\ waktu\ paralel}$$

Secara definisi, *speed-up* adalah perbandingan antara waktu yang diperlukan oleh algoritma sekuensial yang paling efisien dengan waktu untuk algoritma yang sama tetapi dijalankan pada prosesor dengan *pipeline* atau data paralel. Setelah memahami konsep kompleksitas waktu, maka *speed-up* dapat ditentukan juga dari perbandingan kompleksitas waktu algoritma terbaik untuk sekuensial dengan kompleksitas waktu algoritma paralel untuk kasus yang sama.

Untuk meningkatkan *speed-up* dari suatu algoritma, maka dapat menggunakan hukum *Amdahl*, yaitu prinsip dasar dalam peningkatan kecepatan proses suatu prosesor jika hanya sebagian dari peralatan perangkat peralatan perangkat keras atau perangkat lunak nya yang diperbaharui atau ditingkatkan kinerja nya. Persamaan hukum *Amdahl* adalah sebagai berikut:

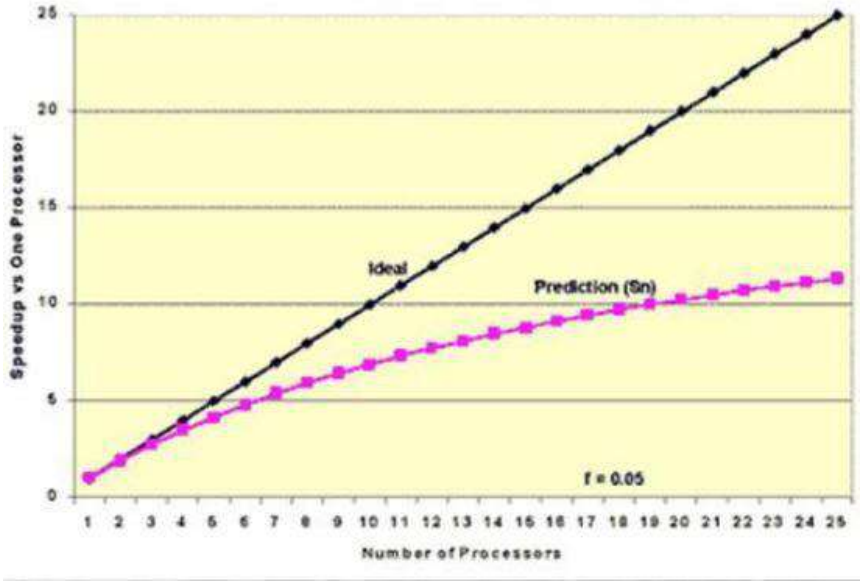
$$S \leq \frac{1}{f + \frac{1-f}{p}}$$

dimana

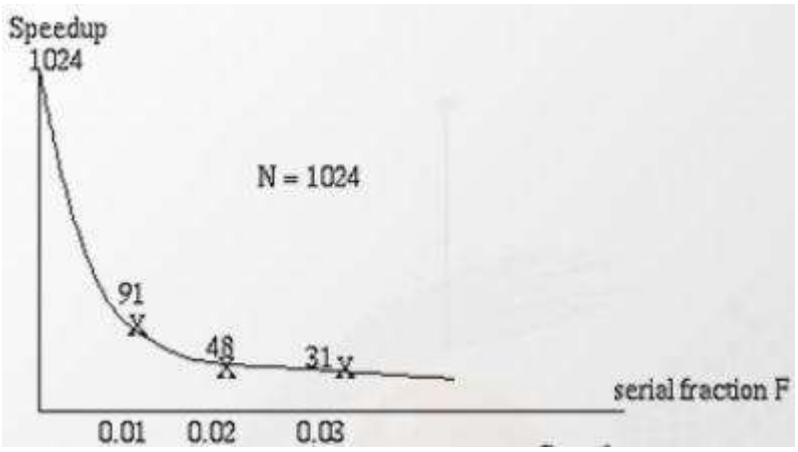
$S = speed - up.$

$f = bagian\ proses\ yang\ harus\ dijalankan\ secara\ sekuensial\ 0 \leq f \leq 1.$

$p = jumlah\ prosesor.$



Gambar 3. Kinerja Hukum *Amdahl*



Grafik 1. *Speed-up* pada 1024 Prosesor

3. Efficiency

Yang dimaksud dengan *efficiency* adalah berkaitan dengan *cost*, yaitu suatu nilai yang diperoleh dari perkalian antara jumlah prosesor yang digunakan dengan kompleksitas waktu dari algoritma paralel yang dipakai.

$$cost = jumlah\ prosesor\ x\ kompleksitas\ waktu\ paralel$$

Cost disebut optimal jika nilainya mempunyai order sama dengan *lower bound* kompleksitas waktu algoritma sekuensial. Jika optimal *cost* sulit ditentukan

karena *lower bound* kompleksitas waktu sekuensial tidak diketahui, maka bisa digunakan nilai *efficiency* untuk mengevaluasi *cost*.

$$efficiency = \frac{kompleksitas\ waktu\ sekuensial\ yang\ diketahui}{cost}$$

Jika $efficiency > 1$, maka menggunakan prosesor tunggal bisa lebih cepat tetapi *cost* tidak optimal. Jika $efficiency = 1$, maka *cost* masih diragukan. Jika $efficiency < 1$, maka diperoleh *cost* optimal.

4.3 Contoh Kasus dan Implementasi Algoritma Paralel

1. Implementasi Algoritma Pengurutan (*Sorting*)

Pengurutan bilangan merupakan tahap merangkai sebuah deretan bilangan menjadi terurut secara naik atau menurun adalah operasi dasar yang dipakai pada banyak aplikasi. Jika bilangan terduplikasi pada urutannya, maka proses pengurutan dapat didefinisikan sebagai penempatan bilangan pada urutan yang tidak menurun atau tidak menaik. Pengurutan juga dapat diaplikasikan pada data non-numeris, misalnya mengurutkan suatu rangkaian secara alfabet. Pengurutan seringkali dilakukan agar perintah *search* (pencarian) dan perintah lainnya dapat dilakukan dengan lebih mudah. Buku-buku di perpustakaan diurutkan menurut subjek /nama agar pencarian terhadap suatu buku tertentu lebih mudah.

Quicksort dan *mergesort* merupakan jenis algoritma penyortiran sekuensial yang terkenal. Kedua algoritma tersebut tercakup dalam keluarga algoritma penyortiran berbasis-perbandingan (*comparison-based*), yaitu algoritma penyortiran berbasis pada pembandingan sepasang bilangan. Kompleksitas waktu terburuk dari *mergesort* dan kompleksitas waktu rata-rata dari *quicksort* adalah sama, yaitu n log n untuk n bilangan asli. Pada kenyataannya, O(n log n) optimal untuk berbagai jenis algoritma penyortiran sekuensial berbasis perbandingan tanpa menggunakan *property* khusus apapun dari bilangan yang diurutkan. Untuk itu, kompleksitas waktu paralel terbaik yang dapat kita harapkan berdasarkan algoritma penyortiran sekuensial dan penggunaan prosesor berjumlah p adalah:

$$\text{Kompleksitas waktu paralel optimal} = \frac{O(n \log n)}{p} = O(\log n) \quad \text{jika } p = n$$

Algoritma pengurutan dengan kompleksitas waktu O(log n) dan dengan n prosesor telah ditunjukan oleh Lighton (1984) berdasarkan sebuah algoritma pada Ajtai, Komlos dan Szemeredi (1983), tetapi konstan *hidden* pada notasi terurut cukup besar. Algoritma pengurutan dengan kompleksitas waku O(log n) juga diperlihatkan oleh Leighton (1994) untuk sebuah n-prosesor *hypercube* menggunakan operasi random. Akl (1985) mendeskripsikan 20 algoritma pengurutan paralel yang berbeda-beda, beberapa diantaranya mencapai kompleksitas yang rendah untuk suatu jaringan interkoneksi tertentu. Meskipun demikian, pada umumnya sebuah algoritma dengan komplesitas O(log n) yang realistis dengan n-prosesor merupakan sebuah tujuan yang sulit dicapai dengan menggunakan algoritma pengurutan berbasis perbandingan. Jumlah prosesor yang dipakai mengkin lebih daripada n.

2. Implementasi Algoritma Pengurutan Compare-Exchange

Suatu perintah yang menjadi dasar dari berbagai algoritma pengurutan sekuensial klasik adalah perintah *compare-exchange* (penggabungan-penukaran). Pada sebuah operasi *compare-exchange*, dua bilangan misalnya A dan B dibandingkan. Jika A>B, A dan B ditukarkan, yaitu isi dan lokasi yang menampung A dipindahkan ke lokasi yang menampung B, isi dari lokasi tersebut tidak dipertukarkan, maka *compare* and *exchange* diilustrasikan pada kode sekuensial sebagai berikut

```
if (A > B) {
   temp = A;
   A = B;
   B = temp;
}
```

Compare and *exchange* sangat cocok untuk diaplikasikan dalam *system message-passing*. Misalkan *compare* and *exchange* dilakukan pada dua bilangan A dan B dengan A berada pada proses P_1 dan B pada proses P_2, maka cara sederhana untuk mengimplementasikan *compare* and *exchange*. Alternatif lain yang dapat ditempuh adalah P_1 mengirim A ke P_2 dan P_2 mengirim B ke P_1. Selanjutnya, kedua proses tersebut akan sama-sama melakukan operasi

perbandingan. P_1 menyimpan salah satu di antara A dan B yang memiliki nilai yang lebih kecil dan P_2 menyimpan salah satu di antara A dan B yang memiliki nilai terbesar, kode tertulis sebagai berikut.

Proses P_1

```
send(&A, P2);
send(&B, P2);
if (A > B) A = B
```

Proses P_2

```
recv(&A, P1);
send(&B, P1);
if (A > B) A = B
```

Pada awalnya, proses P1 menjalankan *send( )* dan proses P2 pertama menjalankan *recv( )* untuk mencegah *deadlock*. Pada versi pertama, *send( )* dan *recv()* otomatis berada pada urutan *non-deadlock*. Kemungkinan lain, P1 dan P2 dapat menjalankan *send( )* dahulu jika *locally blocking* sedang digunakan dan ada jaminan bahwa *buffering* cukup. Selanjutnya, kedua proses dapat mulai mengirim pesannya secara simultan untuk menutup *overhead* pada proses pengiriman pesan. Dari segi MPI, pemrograman ini dirasa kurang aman karena *deadlock* akan terjadi jika tidak terdapat *buffering* cukup.

Catatan ketepatan dalam komputasi terduplikasi. Kode sebelumnya mengasumsikan bahwa jika A>B, maka akan menghasilkan jawaban *Boolean* yang sama pada semua prosesor. Prosesor berbeda beroperasi pada presisi yang berbeda. Hal ini berarti jawaban yang dihasilkan akan berbeda saat bilangan real dibandingkan. Situasi ini terjadi ketika komputasi terduplikasi pada prosesor yang berbeda untuk mereduksi *message-passing*. Pada kasus tersebut, *message-passing* tidak direduksi, tetapi membuat kode pada tiap prosesor terlihat serupa memungkinkan sebuah program tunggal menjadi lebih mudah dibuat untuk semua proses.

Pembagian Data, walaupun sejauh ini kita telah mengasumsikan bahwa satu prosesor digunakan untuk satu bilangan, sebenarnya terdapat lebih banyak bilangan daripada jumlah prosesor. Pada kasus seperti itu, sekelompok bilangan akan dibagi kepada tiap-tiap prosesor. Pendekatan seperti ini dapat diterapkan pada semua

algoritma pengurutan. Sebagai contoh, terdapat p prosesor dan n bilangan, sebuah list bilangan dengan anggota sebanyak n/p akan diberikan pada tiap prosesor. Operasi *compare-exchange* hanya ada satu prosesor yang mengirimkan partisinya ke prosesor lain.yang kemudian melakukan operasi penggabungan dan mengembalikan setengah dari *list* terbawah ke prosesor pertama. Semua prosesor saling bertukar grup. Satu prosesor akan tetap mempertahankan n/p bilangan kecil dan satu prosesor yang lain akan mempertahankan n/p bilangan besar dari total 2n/p bilangan. Metode umum yang biasa digunakan adalah dengan menjaga *list* terurut pada tiap prosesor dan menggabungkan *list* yang telah disimpan dengan *list* yang dating. Dengan n bilangan, terdapat n-1 perintah *compare* dan *exchange* pada fase pertama, yaitu memposisikan bilangan terbesar pada akhir *list*. Pada fase selanjutnya, yaitu memposisikan bilangan terbesar berikutnya, sehingga terdapat n-2 operasi *compare-exchange* dan seterusnya. Untuk itu, jumlah operasi keseluruhan dirumuskan sebagai berikut:

$$\text{Jumlah operasi compare-and-exchange} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Rumus di atas mengindikasikan bahwa kompleksitas waktu $O(n^2)$ yang dimiliki *compare-exchange* tersebut bernilai konstan. Dimisalkan bilangan yang terdapat pada *array* adalah a[], kode sekuensialnya adalah sebagai berikut:

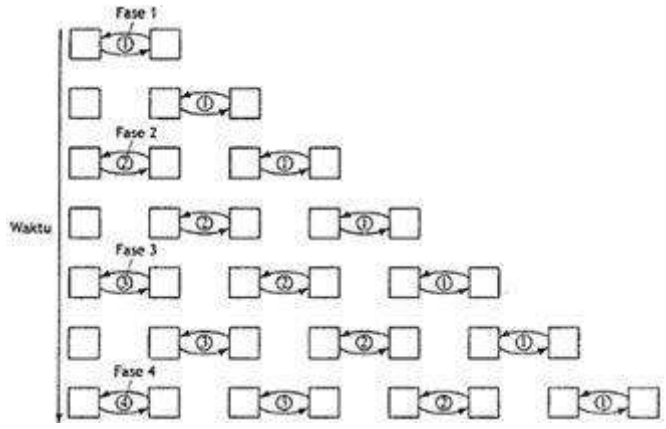
```
for (i = n - 1; i > 0; i--)
   for (j = 0; j < i; j++) {
      k = j +1;
      if (a[j] > a[k]) {
         temp = a[j];
         a[j] = a[k];
         a[k] = temp;
      }
   }
```

Bubble sort adalah sebuah algoritma pengurutan murni. Setiap langkah pada loop dalam dilakukan sebelum loop dalam berikutnya dan loop dalam keseluruhan selesai dilakukan sebelum perulangan berikutnya dari loop luar. Meskipun demikian, bukan berarti *bubble sort* tidak dapat diformulasikan dalam sebuah algoritma paralel hanya karena kode sekuensial menggunakan *statement* yang bergantung pada

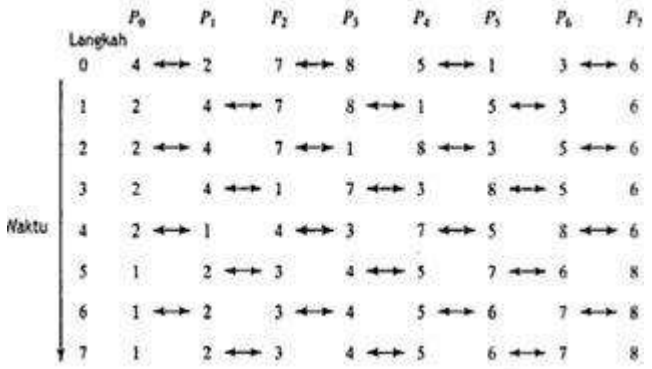
statement sebelumnya. Aksi penggelembungan (*bubling*) dari perulangan loop dalam selanjutnya dapat dimulai sebelum perulangan sebelumnya berakhir dengan syarat bahwa selama aksi tersebut tidak mengambil alih aksi penggelembungan sebelumnya. Hal ini mengindikasikan bahwa penggunaan sebuah struktur implementasi *pipeline* lebih disarankan.

Sebagian aksi penukaran dari *bubble sort* dapat dilakukan di belakang aksi penukaran lainnya pada sebuah *pipeline*. Kita melihat bahwa perulangan 2 dapat dilakukan di belakang perulangan 1 pada waktu yang sama jika dipisahkan oleh sebuah proses. Hal yang sama juga terjadi pada perulangan 3, yang dapat dilakukan di belakang perulangan 2.

Ide ini menghasilkan sebuah varian *bubble sort*, yaitu *odd-even* (*transposition*) *sort*, yang beroperasi pada dua fase alternatif, yaitu fase genap (*even*) dan fase ganjil (*odd*). Pada fase genap, proses bernomor genap menukarkan bilangan dengan tetangga sebelah kanannya. Begitu pula, pada fase ganjil proses bernomor ganjil akan menukarkan bilangannya dengan tetangga sebelah kanannya. *Odd-even transposition sort* tidak akan dibahas dalam pemrograman sekuensial, karena tidak memiliki keuntungan yang lebih dibandingkan *bubble sort* jika diimplementasikan dalam proses sekuensial. Meskipun demikian, implementasi secara paralel dapat mengurangi kompleksitas waktu menjadi O(n). *Odd-even transposition sort* dapat diimplementsikan pada sebuah jalur jaringan, dan hal ini dinilai optimal untuk jaringan tersebut.



Gambar 4. Aksi *Overlap Bubble Sort* pada sebuah *Pipeline*



Gambar 5. *Odd-even Transpotition Sort* Mengurutkan 8 Bilangan

Karena dalam kasus yang terburuk hanya membutuhkan n langkah untuk mereposisi sebuah bilangan. Sebuah penerapan *odd-even transposition sort* pada sebuah deretan delapan bilangan dengan sebuah bilangan yang disimpan pada setiap proses.

Pertama, marilah kita perhatikan dua fase alternatif yang berbeda tersebut secaraterpisah. Pada fase genap, terdapat proses *compare-exchange*, P_0<->P_1, P_2<->P_3, dan seterusnya. Dengan menggunkan bentuk compare and exchange. Kode yang harus dibuat adalah seperti ini.

```
P_i i = 0, 2, 4, ..., n – 2 (genap)
recv(&A, P_(i+1));
send(&B, P_(i+1));
if (A < B) B = A;
```

```
P_i i = 1, 3, 5, ..., n – 1 (ganjil)
send(&A, P_(i-1)); /* even phase */
recv(&B, P_(i-1));
if (A < B) A = B; /* exchange */
```

Bilangan yang disimpan di P genap adalah B, dan bilangan yang disimpan di P ganjil adalah A. Pada fase ganjil terdapat operasi *compare-exchange*, P_0<->P_1, P_2<->P_3, dan seterusnya.

```
P_i, i = 1, 3, 5, ..., n – 3 (ganjil)
send(&A, P_(i+1));
recv(&B, P_(i+1));
if (A > B) A = B;
```

```
P_i, i = 2, 4, 6, ..., n – 2 (genap)
recv(&A, P_(i-1));      /* odd phase */
Send(&B, P_(i-1));
if (A > B) B = A;   /* exchange */
```

Pada kedua fase tersebut, proses bernomor ganjil mengerjakan rutin *send()* terlebih dahulu dan proses bernomor genap mengerjakan rutin *recv()* terlebih dahulu. Jika dikombinasikan, maka segmen-segmen kode tersebut dapat

dikombinasikan ke dalam bentuk SPMD dimana identitas proses digunakan untuk memilih bagian program yang akan dieksekusi oleh prosesor.

```
P_i, i = 1, 3, 5, ..., n - 3 (ganjil)
send(&A, P_i-1);
recv(&B, P_i-1);
if (A < B) A = B;
if (i <= n-3){
 send(&A, Pi+1);
 recv(&B, Pi+1);
 if (A > B) A = B;
}
```

```
P_i, i = 0, 2, 4, ..., n - 2 (genap)
recv(&A, P_i+1);      /* even phase */
send(&B, P_i+1);
if (A < B) B = A;
if (i >= 2){
 recv(&A, Pi-1);
 send(&B, Pi-1);
 if (A > B) B = A;
}
```

3. Implementasi Algoritma *Mergesort-Quicksort*

Mergesort adalah sebuah algoritma pengurutan sekuensial klasik menggunakan pendekatan *divide-and-conguer*. *List* yang belum terurut pertama dibagi dua bagian. Setiap bagian kemudian dibagi lagi menjadi dua bagian lagi. Hal ini terus dilakukan hingga mendapatkan setiap bilangan pada *list* tersebut. Setelah itu, pasangan-pasangan bilangan tersebut digabungkan sebagai sebuah *list* terurut yang terdiri atas dua bilangan. Pasangan dari *list* dua bilangan tersebut digabungkan menjadi *list* empat bilangan terurut. Hal ini dilakukan terus hingga didapatkan keseluruhan *list* yang terurut.

Quicksort (Hoare, 1962) adalah sebuah algoritma pengurutan sekuensial yang sangat terkenal. Algoritma ini memiliki kinerja yang bagus dengan kompleksitas waktu O(n log n). Pertanyaan yang harus dijawab adalah apakah sebuah versi paralel langsung dari algoritma ini dengan menggunakan n prosesor mampu mencapai kompleksitas waktu O(log n). Kita tidak menjawab pertanyaan serupa untuk algoritma *mergesort* sebagaimana analisis yang telah kita lakukan. Untuk itu, kita jadikan *quicksort* sebagai dasar algoritma pengurutan paralel.

Dalam pemograman sekuensial, *quicksort* mengurutkan sebuah *list* bilangan dengan membagi *list* tersebut menjadi dua *sublist* seperti halnya dengan *mergesort*. Semua bilangan pada suatu *sublist* dirangkai hingga menjadi lebih kecil dari semua bilangan *sublist* yang lain. Hal ini dicapai dengan pertama memilih sebuah bilangan disebut *pivot* dan kemudian dibandingkan dengan bilangan yang lainnya. Jika suatu bilangan lebih kecil daripada *pivot*, bilangan tersebut diletakkan pada sebuah *sublist*. Jika tidak, maka bilangan tersebut diletakkan pada *sublist* yang

lain. *Pivot* dapat berupa sebuah bilangan yang ada pada *list*. Tetapi seringkali bilangan yang dipilih sebagai *pivot* adalah bilangan pertama pada *list*. Pivot diletakkan pada salah satu *sublist* atau dapat dipisahkan dan diletakkan pada posisi akhir. Dalam pembahasan ini, kita akan memisahkan *pivot*.

Prosedur dikerjakan secara berulang pada *sublist*, sehingga didapatkan empat buah *sublist*. Konsepnya adalah dengan meletakkan bilangan pada area nya. Perbedaannya adalah area tersebut dikenali dengan menggunakan *pivot* yang dipilh pada setiap langkah, prosedur dikerjakan secara berulang, sehingga didapatkan *sublist-sublist* yang beranggotakan masing-masing satu bilangan. *Sublist-sublist* tersebut kemudian diurutkan sehingga didapatkan *list* yang terurut.

Quicksort biasanya dideskripsikan menggunakan sebuah algoritma rekursif. Sebagi contoh, misalnya sebuah array *list[ ]* berisi sederetan bilangan dan *pivot* adalah *index* pada array tersebut yang menunjukkan posisi akhir dari *pivot*. Kita dapat membuat kode sebagai berikut:

```
quicksort(list, start, end)
{
   if (start < end) {
      partition(list, start, end, pivot)
      quicksort(list, start, pivot-1); /* recursively call on sublist*/

        quicksort(list, pivot+1, end);
     }
  }
```

Partition( ) memindahkan bilangan pada *list* antara *start* dan *end*. Apabila bilangan tersebut lebih kecil daripada *pivot*, maka bilangan tersebut diletakkan pada urutan sebelum *pivot*, sedangkan jika bilangan tersebut lebih besar daripada *pivot*, maka bilangan tersebut diletakkan pada urutan setelah *pivot*. *Pivot* berada pada posisi akhir ketika *list* tersebut telah terurut.

Satu hal yang membingungkan untuk memparalelkan *quicksort* adalah cara untuk memulai dengan sebuah prosesor dan melewatkan salah satu *call* rekursif ke prosesor lain, sementara operasi *call* rekursif yang lainnya tetap berlangsung. Hal ini akan membentuk sebuah pohon struktur seperti yang telah kita kenal saat membahas *mergesort*.

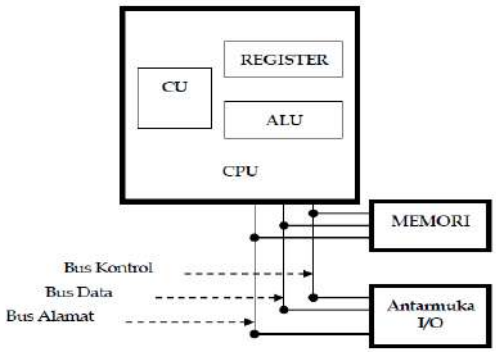
4.4 Teknik Perancangan Algoritma Paralel

1. Arsitektur Von Neumann

Konsep dasar yang melandasi arsitektur *Von Neumann* adalah kemampuan menyimpan instruksi di dalam memori bersamaan dengan data yang dioperasikan oleh instruksi-instruksi tersebut. Sebelum *Von Neumann* mengajukan jenis arsitektur ini, mesin-mesin komputasi dirancang dan dikembangkan hanya untuk tugas-tugas tunggal tertentu saja dan semua pemrograman dilakukan dengan mengatur ulang kabel-kabel sirkuit, sehingga pekerjaan pemrograman begitu sangat membosankan, terlebih-lebih jika terjadi suatu kesalahan akan sulit untuk mendeteksi letak-letak kesalahannya dan terlebih lagi memperbaiki kesalahan kesalahan tersebut.

Arsitektur *Von Neumann* disusun berdasarkan tiga komponen yang berbeda yaitu satu unit CPU, memori, dan antar muka I/O.

a. CPU, perannya sebagai otak sistem komputasi, meliputi tiga komponen utama yaitu control unit (CU), satu atau lebih ALU, dan berbagai register. CU menentukan perintah instruksi yang harus dieksekusikan dan mengontrol pencarian berbagai operand yang tepat.
b. Memori digunakan untuk menyimpan instruksi program dan data. Dua jenis umum memori yang sering digunakan adalah RAM dan ROM. RAM bertugas melakukan penyimpanan data dan berbagai program yang dieksekusi oleh mesin secara sementara atau dengan kata lain isi dari memori RAM dapat diubah pada sebarang waktu dan terhapus ketika aliran listrik pada komputer terputus. ROM bersifat permanen, bertugas menyimpan berbagai instruksi mesin awal.
c. Antarmuka I/O memungkinkan memori komputer berinteraksi dengan perangkat output, mengirim dan menerima data. Juga memungkinkan komputer berkomunikasi ke *user* dan perangkat penyimpanan *secondary* seperti *disk* dan *drive tape*.



Gambar 6. Arsitektur Von Neumann

2. Arsitektur RISC dan CISC

Arsitektur komputer telah mengalami perkembangan pesat dan lebih kompleks dengan kekompleksitasan dalam berbagai fitur seperti set instruksi yang lebih besar, lebih banyak mode pengamatan unjuk kerja instruksi yang lebih komputasional, terdapat lebih banyak register-register dengan fungsi khusus dan sebagainya. Berbagai mesin yang memiliki berbagai peningkatan kekompleksitasan di atas sering diistilahkan sebagai *complex instruction set computers* (CISCs).

Berdasarkan konsep peniadaan berbagai instruksi tersebut ke dalam set instruksi, maka muncullah gagasan inovatif untuk mendesain suatu arsitektur komputer alternatif yang sering diistilahkan *reduced instruction set computer* (RISC). Filosofi dibalik perancangan arsitektur RISC adalah keuntungan performansi dengan hanya instruksi-instruksi tertentu saja yang ditambahkan ke set instruksi. Mesin arsitektur RISC pertama kali dibangun pada tahun 1982 oleh perusahaan IBM, minicomputer 801. Karakteristik umum pada perancangannya adalah kepemilikan set instruksi yang sederhana dan terbatas, memori cache yang bersifat on-chip, memiliki kompiler yang bertugas memaksimalkan penggunaan register sehingga meminimalkan akses memori, dan penekanan kepada optimisasi pipeline instruksi.

Secara umum, sebuah arsitektur RISC memiliki berbagai karakteristik berikut :

a. Sebagian besar instruksi mengakses berbagai operand dari register kecuali intruksi-instruksi tertentu, seperti LOAD/STORE, yang mengakses memori, dengan kata lain arsitektur RISC merupakan mesin load-store.
b. Eksekusi sebagian besar instruksi hanya membutuhkan satu siklus prosesor tunggal, kecuali intruksi-instruksi tertentu, seperti LOAD/STORE. Tetapi dengan kehadiran cache on-chip, instruksi LOAD/STORE dapat dilakukan pada satu siklus atau periode.
c. Instruksi memiliki format tetap dan tidak melewati batas word memori utama.
d. Control unit telah terintegrasi menyatu, sehingga arsitektur RISC tidak dimikroprogramkan.
e. Memiliki sejumlah kecil format instruksi.
f. CPU memiliki arsip register yang besar, alternatif yang lain adalah memori cache on-chip.
g. Kekompleksitasan pada kompiler

h. Relatif memiliki instruksi yang lebih sedikit (sering < 150) dan memiliki mode pengalamatan yang sangat sedikit.
i. Mendukung operasi HLL (High-Level Language) dengan pemilihan instruksi yang tepat dan dengan memanfaatkan kompiler yang optimal.
j. Memanfaatkan penggunaan pipeline instruksi dan berbagai pendekatan untuk bertransaksi dengan percabangan (branch), seperti multipel prefetch dan berbagai teknik prediksi percabangan.

3. Arsitektur Systolic Array dan Arsitektur Data Flow

Di dalam arsitektur *data flow* sebuah instruksi siap untuk dieksekusi ketika data untuk operandnya juga telah tersedia. Ketersedian data diperoleh dengan menyalurkan berbagai hasil dari eksekusi instruksi sebelumnya ke dalam berbagai operand instruksi tunggu.

Arsitektur *systolic array* memiliki sejumlah prosesor sederhana yang identik, atau elemen pemrosesan (PEs). PEs disusun ke dalam bentuk yang terorganisasi baik, seperti array dua dimensi atau array linier.

Komputer MIT terdiri dari lima unit utama : (1) unit pengolahan, terdiri dari berbagai elemen pemrosesan khusus, (2) unit memori, terdiri dari sel instruksi yang menyimpan instruksi dan berbagai operand, (3) jaringan arbitrasi, yang mengirim instruksi ke elemen pemrosesan untuk perlakuan eksekusi, (4) jaringan distribusi, untuk memindahkan data hasil dari elemen pemrosesan ke memori, dan (5) unit kontrol, yang mengatur keseluruhan unit. Satu sel intruksi menyimpan sebuah instruksi yang terdiri dari opcode, operand, dan satu alamat tujuan. Instruksi di-*enable*-kan ketika semua operand dan sinyal kontrol yang dibutuhkan telah diterima. Jaringan arbitrasi mengirim intruksi yang telah di- *enable*kan sebagai suatu paket operasi ke elemen pemrosesan yang tepat. Setelah instruksi dieksekusi, hasilnya dikirim kembali melalui jaringan distribusi ke lokasi tujuan di dalam memori. Masing-masing hasil dikrim sebagai sebuah paket, yang terdiri dari sebuah nilai dan sebuah alamat tujuan.

Mesin *data flow* dapat diklasifikasikan menjadi dua kelompok : statis dan dinamis. Mesin *data flow* statis, sebuah intruksi di*enable*- kan ketika seluruh operand yang diperlukan telah diterima dan terdapat sebuah intruksi lainnya yang menunggu hasil dari instruksi tersebut. Gambar 5.6 merepresentasikan sampel mesin *data flow* statis. Mesin *data flow* dinamis, sebuah intruksi di-*enable*kan ketika

seluruh operand yang diperlukan telah diterima. Dalam kasus mesin *data flow* dinamis, berbagai operand bisa saja dipergunakan oleh intruksi dalam satu waktu, sehingga satu busur graf *data flow* dinamis dapat memuat lebih dari satu token.

Dibandingkan dengan mesin *data flow* statis, pendekatan dinamis lebih memungkinkan pengeksekusian secara paralel, karena sebuah instruksi tidak perlu menunggu lokasi yang kosong dalam instruksi lainnya untuk pengeksekusian sebelum menempatkan berbagai hasil.

Pada bidang komputasi sains, kita sering dihadapi pada berbagai kasus persamaan linier secara serempak, atau secara khusus, persamaan sistem linier skala besar. Berbagai kajian diperlukan untuk menemukan suatu metode yang cepat, akurat, dan dengan biaya yang efektif untuk penyelesaian berbagai kasus di atas. Menghadapi berbagai komputasi yang besar seperti perkalian, inversi pada matriks, dikembangkanlah sebuah software komputasi berbasis komputer digital *high-speed*. Pengolahan aljabar matriks menggunakan software berbasis komputer *general-purpose* membutuhkan waktu komputasi yang lama, selain keterbatasan memori untuk mengakomodasi berbagai matriks berskala cukup besar.

Oleh karena itu, untuk mengurangi permasalah di atas, maka dikenalkanlah suatu pendekatan paralel yang diterapkan pada sebuah komputer, yang dikombinasikan dengan berbagai prilaku mesin *special-purpose*. Salah satu solusi terhadap kebutuhan komputasional paralel yang cukup tinggi adalah dengan mengkoneksikan sejumlah besar prosesor identik sederhana atau pengkoneksian berbagai emen pemroses (PEs). Masing-masing PE memiliki *storage* privat yang terbatas, dan setiap PE hanya dikoneksikan ke PE tetangganya yang terdekat. Oleh karena itu PE disusun ke dalam suatu struktur yang terorganisasi secara baik seperti array dua dimensi atau array linier. Karakteristik struktur inilah yang dapat diistilahkan sebagai *systolic array*. *Systolic array* memberikan suatu tatanan ideal untuk pengimplementasian VLSI, ditambah penerapan memori *interleaved* yang lebih difungsikan untuk mensuplai data ke berbagai array tersebut.

DAFTAR PUSTAKA

Abdullah Dahlan. (2016). *Pemrosesan Paralel*. Unimal Press.

Amdahl, G.M. 1967. Validity of The Single Processor Approach to Achieving Large-Scale Computing Capability. Proc. AFIPS Conference, pp 483-485 [5].

Ayuningtyas, Astika. 2019. Arsitektur Memori Komputer. Diakses pada 27 September 2022 dari *http://astika.stta.ac.id/2016/09/arsitektur-memori-komputer-paralel.html*.

Brey, Barry B. 2003. *The Intel Mikroprosesor 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro Prosessor, Pentium II, Pentium III, Pentium 4 Architecture, Programming, and Interfacing* (6thEdition). Prentice Hall.

Denyer, Peter B.; Renshaw, David (1985). *VLSI signal processing: a bit-serial approach*. VLSI systems series. Addison-Wesley. ISBN 978-0-201-13306-6.

Fadlisyah. 2005. *Pemrosesan Paralel* : Seri Diktat Kuliah. Universitas Malikussaleh.

Fadlisyah, dkk., 2008. *Robotika : Reasoning, Planning, Learning*. Graha Ilmu.

Foster, I. 1995. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Massachuttes: Addison-Wesley Publi-shing Company.

Freeman, T. L. and Phillips, C. 1992. *Parallel Numerical Algorithms.* New York: Prentice-Hall, Inc.

Geeks for Geeks. 2019. Computer Architecture Flynn Taxonomy. Diakses pada 10 Oktober 2022 dari https://www.geeksforgeeks.org/computer-architecture-flynns-taxonomy/

Gustafson, J.L. Reevaluating Amdahl's Law. Comm. ACM, 31 (5): 532-533.

Mano, M. Morris. 1993. *Computer System Architecture*. PrenticeHall International.

Phoenixnap. 2019. Install Spark on Windows 10. Diakses pada 10 Oktober 2022 dari https://phoenixnap.com/kb/install-spark-on-windows-10

S. Rastogi and H. Zaheer, "Significance of paralel computation over serial computation," 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), 2016.

Spark. 2019. Spark Apache. Diakses pada 10 Oktober 2022 dari https://spark.apache.org/

Stalling, William. 1998. *Organisasi dan Arsitektur Komputer Perancangan Kinerja (jilid 1)*. Prentice Hall yang diterjemahkan oleh PT Prenhallindo, Jakarta.

The University of Edinburg. 2021. Shared Memory vs Distributed Memory. Diakses pada 27 September 2022 dari *https://www.futurelearn.com/info/courses/supercomputing/0/steps/24028*.

Toward Data Science. 2019. Beginner Guide to Apache Spark. Diakses pada 10 Oktober 2022 dari https://towardsdatascience.com/a-beginners-guide-to-apache-spark-ff301cb4cd92

Towards Data Science. 2019. Create Your First ETL Pipeline in Apache Sparks and Python. Diakses pada 10 Oktober 2022 dari https://towardsdatascience.com/create-your-first-etl-pipeline-in-apache-spark-and-python-ec3d12e2c169

Wikipedia. 2019. Apache Sparks. Diakses pada 10 Oktober 2022 dari https://en.wikipedia.org/wiki/Apache_Spark

Wikipedia. 2019. Taksonomi Flynn. Diakses pada 10 Oktober 2022 dari https://id.wikipedia.org/wiki/Taksonomi_Flynn

Zargham, Mehdi R., 1996. *Computer Architecture Single and Parallel Systems*. Prentice-Hall International.