BAB II
LANDASAN TEORI

2.1 *Cluster* Komputer

Cluster Komputer adalah sistem yang terdiri atas sejumlah *PC* yang terhubung dengan jaringan berkecepatan tinggi (*high speed switch*). Setiap anggota *cluster* merupakan sebuah sistem tersendiri yang mempunyai pengolah (*processor*), baik tunggal maupun jamak, memori, sistem operasi, dan perangkat I/O (*input/output*) sendiri. Anggota-anggota *cluster* ini dapat ditempatkan di sebuah tempat bersama-sama atau terpisah secara fisik dan dihubungkan oleh suatu jaringan komunikasi komputer (Buyya,1999 dikutip oleh Sumaryono dan Syahputra,2011).

Cluster komputer adalah sekumpulan dari beberapa komputer tunggal dengan performa rendah menjadi satu kesatuan sehingga memiliki performa tinggi, biasanya disebut sebagai super komputer. Komponen *cluster* biasanya saling terhubung dengan cepat melalui sebuah interkoneksi yang sangat cepat, atau bisa juga melalui jaringan local (*Local Area Network,LAN*) (Buletin Cuaca Antariksa,2014).

Dalam sebuah cluster terdapat beberapa komputer yang disebut sebagai *node.* Masing-masing *node* tersebut adalah mesin komputasi yang saling terhubung satu sama lain dalam satu jaringan. Masing-masing *nodes* di dalam cluster memiliki sistem operasi dan sistem *input* dan *output* . Jika semua *node* di dalam *cluster* tersebut memiliki arsitektur dan sistem operasi yang sama disebut sebagai cluster yang homogen. Jaringan interkoneksi yang menghubungkan *nodes* di dalam cluster sebisa mungkin memiliki *bandwidth* yang tinggi dan *litency* yang serendah mungkin karena kecepatan dan *litency* di dalam jaringan cluster sangat menentukan performa komputasi yang dilakukan oleh cluster.

Untuk membangun sebuah cluster dibutuhkan empat buah komponen, yakni dua buah komponen *hardware* dan dua buah komponen *software.* Komponen *hardware* yang pertama adalah *node* atau mesin komputasi dan yang kedua adalah jaringan interkoneksi, sedangkan komponen *software* yang pertama adalah *software*

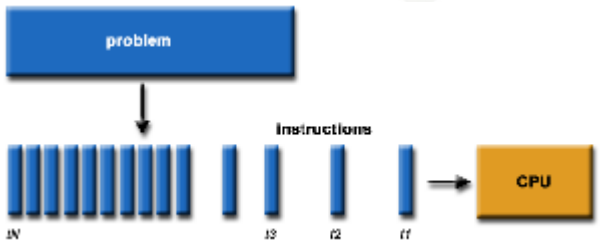
yang dapat digunakan oleh pengguna untuk mengembangkan aplikasi paralel dan yang kedua adalah *software* yang digunakan untuk memonitor dan mengatur aplikasi yang berjalan pada cluster (Zulfikar, 2010).

Cluster digunakan untuk melakukan komputasi dengan performa yang tinggi. Komputasi dengan performa tinggi tersebut dicapai dengan melakukan komputasi secara paralel yaitu menyelesaikan sebuah permasalahan komputasi dengan membagi permasalahan untuk menyelesaikan secara simultan oleh masing-masing *node* di dalam cluster (Widyaputra, 2008).

Pada dasarnya, hampir semua sistem operasi seperti Linux, Solaris, FreeBSD, dan Windows dapat digunakan untuk pengolalaan *node cluster*. Namun demikian masih dibutuhkan aplikasi yang ditulis sebagai libraries pada sistem *middleware* seperti PVM (*Parallel Virtual Machine*) dan MPI (*Message Passing Interface*). Koneksi *cluster* dapat digunakan sebagai sistem server dengan skalabilitas besar dan performa komputer parallel yang tinggi. Koneksi *cluster* yang diimplementasikan basisdata, mengakibatkan sejumlah besar data dapat ditrasfer secara kontinu dari satu *node cluster* menuju *node cluster* lainnya. Disamping itu dapat pula dilakukan eksekusi secara parallel, distribusi beban dan pengelolaan sistem (Prasetyo,2007).

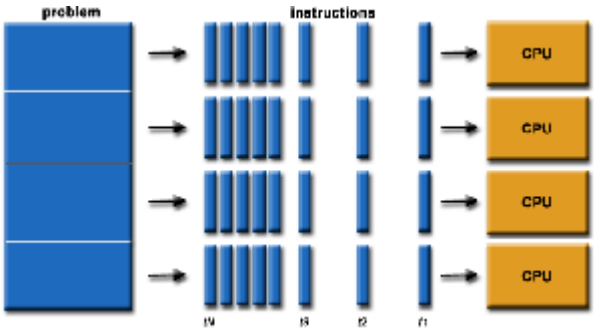
2.2 Komputasi Paralel

Komputasi parallel adalah penggunaan lebih dari satu sumber daya komputasi secara simultan untuk memecahkan persoalan komputasi. Software umumnya dibuat untuk komputasi serial yang dijalankan oleh prosesor tunggal dimana sebuah persoalan dipecah kedalam intruksi yang dieksekusi secara berurutan dan hanya satu intruksi yang boleh dieksekusi pada saat yang sama.



Gambar 2. 1 Komputasi Tunggal/Serial
(Sumber : Syahputra & Akbar, 2011)

Pada komputasi parallel, sebuah persoalan dipecah menjadi beberapa bagian yang dapat diselesaikan pada saat bersamaan. Setiap bagian selanjutnya dipecah menjadi instruksi yang berurutan dan masing-masing bagian dikerjakan oleh prosesor yang berbeda (Moleong, C, & dkk, 2013).



Gambar 2. 2 Komputasi Parallel
(Sumber : Syahputra dan Akbar,2011)

Komputasi parallel adalalah salah satu teknik melakukan komputasi secara bersamaan dengan memanfaatkan beberapa komputer independen secara bersamaan. Ini umumnya diperlukan saat kapasitas yang diperlukan sangat besar, baik karena harus mengelila data dalam jumlah besar ataupun karena tuntutan proses komputasi yang banyak.

Untuk melakukan aneka jenis komputasi paralel ini diperlukan infrastruktur mesin paralel yang terdiri dari banyak komputer yang dihubungkan dengan jaringan dan mampu bekerja secara paralel untuk menyelesaikan satu masalah. Untuk itu diperlukan aneka perangkat lunak pendukung yang biasa disebut sebagai *middleware* yang berperan untuk mengatur distribusi pekerjaan antar *node* dalam satu mesin paralel. Selanjutnya pemakai harus membuat pemrograman paralel untuk merealisasikan komputasi. Tidak berarti dengan mesin paralel semua program yang dijalankan diatasnya otomatis akan diolah secara paralel (Ali, 2011).

Dalam membangun sebuah *cluster* yang digunakan untuk komputasi paralel maka harus memenuhi poin-poin sebagai berikut :

a. *Cluster* harus memiliki kemampuan mendistribusi proses komputasi dari *master node* ke komputer *nodes* yang lain.

b. *master node* harus dapat berkomunikasi dengan komputer *nodes* tanpa harus melakukan proses otentikasi.
c. *Cluster* harus memiliki *file system* global yang dapat diakses oleh semua komputer untuk keperluan pengaksesan berkas-berkas.
d. *Cluster* harus memiliki sebuah sistem untuk memonitor proses-proses dan beban komputasi baik di masing masing *node* maupun beban komputasi dalam *cluster*.

Untuk dapat memenuhi poin-poin yang diatas, maka dalam *cluster* harus terdapat beberapa servis yang digunakan sebagai berikut (Widyaputra, 2008):

a. Pendistribusian proses komputasi menggunakan pustaka OpenMPI. OpenMPI adalah aplikasi yang mengimplementasikan MPI.
b. Proses komunikasi antara *master node* dan *nodes* menggunakan SSH (*Secure Shell)* tanpa proses otentikasi.
c. *File system* global menggunakan *Network File System* (NFS). NFS adalah *file system* yang dieksport oleh sebuah *server* untk diakses oleh *client* yang diijinkan oleh *server*.

Sistem operasi yang digunakan adalah sistem operasi distro Linux Ubuntu LTS 14.04. Sistem operasi Linux yang dipilih karena sistem operasi bersifat *open source* sehingga dapat digunakan secara bebas dan gratis serta banyak referensi yang dapat digunakan sebagai acuan. Distro Ubuntu dipilih karena memberikan stabilitas dan kompatibilitas untuk berbagai aplikasi *cluster* seperti OpenMPI, OCTAVE dan aplikasi-aplikasi untuk pengujian performa (Sumaryo dan Dedy, 2011).

2.2.1 Model Komputasi

Bagian penting dari proses komputasi adalah eksekusi terhadap suatu urutan instruksi atas sekumpulan data. Dengan dasar pemikiran tersebut, komputer dapat diklasifikasi berdasarkan pada aliran data dan aliran instruksi di dalam prosesor. Aliran data adalah lalu lintas pertukaran data antara prosesor dan memori, sedangkan aliran instruksi adalah urutan instruksi yang diolah oleh prosesor. Terdapat empat tipe komputer yaitu :

1. SISD (*Single Instruction/Single Data stream)*

   SISD merupakan tipe komputasi yang banyak digunakan pada beragam komputer sampai saat ini. SISD adalah istilah untuk sebuah arsitektur komputer dimana hanya terdapat satu prosesor yang melaksanakan aliran instruksi tunggal, yang mengeksekusi data yang tersimpan dalam memori tunggal pula.

2. SIMD (*Single Intruction/Multiple Data stream)*

   SIMD merupakan tipe komputasi yang menjelaskan suatu komputer dengan beberapa elemen pemrosesan yang secara besama-sama melakukan satu operasi pada suatu data. SIMD menggunakan banyak prosesor dengan instruksi yang sama, namun setiap prosesor mengolah data yang berbeda.

3. MISD (*Multiple Intruction/Single Data stream)*

   MISD adalah tipe arsitektur komputasi paralel dimana lebih dari satu unit fungsional prosesor melakukan beberapa operasi yang berbeda terhadap data yang sama. Tidak banyak implementasi dari arsitektur ini karena keterbatasan skalabilitasnya dibandingkan dengan tipe arsitektur SIMD dan MIMD.

4. MIMD (*Multiple Intruction/Multiple Data stream)*

   MIMD merupakan tipe komputasi yang menggunakan banyak prosesor dengan setiap prosesor memiliki instruksi yang berbeda dan mengolah data yang berbeda. Komputer dengan tipe MIMD ini dapat saling berbagi memori ataupun memiliki memori yang independen (Sumaryono dan Syahputra,2011).

2.2.2 Konsep Komputasi Paralel

Pemrosesan paralel adalah pendekatan komputasi untuk meningkatkan tingkat dimana satu set data diolah dengan penolahan bagian yang berbeda dari data pada waktu yang sama secara simultan atau bersamaan pada sebuah komputer dan

berfungsi memecahkan beban besar menjadi beberapa beban kecil untuk mempercepat proses penyelesaian masalah.

Secara umum, paradigm komputasi paralel dapat dibagi dua jenis yaitu *paralelisme implisit* dan *paralelisme eksplisit. Paralelisme implisit* merupakan suatu pendekatan dimana penulis program (*programmer*) tidak perlu memperhatikan masalah pembagian *task* ke beberapa prosesor dan memori serta sinkronisasinya. Pembagian *task* dan sinkronisasi ditangani sepenuhnya oleh sistem dibawah program atau *compiler.*

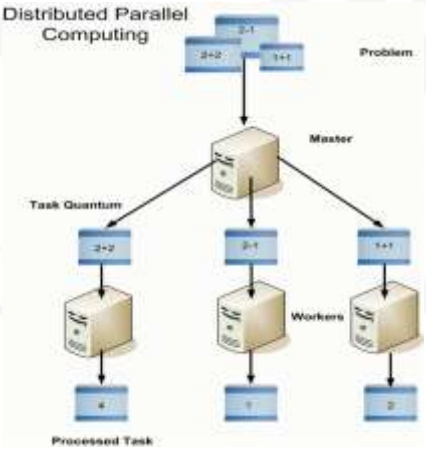
Berbeda dengan paradigm *eksplisit* harus memperhatikan dan menangani masalah pembagian *task* dan komunikasinya. Komunikasi antar dua atau lebih *task* pada umumnya dilakukan dengan menggunakan *shared-memory* atau *message-passing.*

1. *Shared memory*

   Pada model komunikasi ini, *task-task* menggunakan memori yang sama, dimana masing-masing *task* dapat menulis atau membaca secara *asinkron.*

2. *Message passing*

   Pada model komunikasi ini, setiap *task* mempunyai memori lokal masing-masing dan paradigma yang diguna adalah *parallelism eksplisit.* Contoh sumber daya komputasi yang cocok menggunakan pemodelan ini adalah beberapa *workstation* yang dijadikan *cluster.*



Gambar 2. 3 Alur Proses Paralel
(Sumber : Gaurav Verma, 2009)

2.3 MPI (Message Passing Interface)

Model *messagie passing* telah dikenal luas sebagai salah satu model pemrograman aplikasi paralel. Dengan *message passing*, setiap proses yang terlibat dalam penyelesaian masalah secara paralel dapat berkomunikasi dengan cara mengirim pesan. Standar ini kemudian dikenal sebagai *Message Passing Interface* (MPI). MPI berisi sekumpulan *Aplication Programming Interface* (API) yang dapat digunakan oleh pengembang aplikasi. API ini dirancang untuk dapat diterapkan pada bahasa pemrograman apa saja (Syahputra & Akbar, 2011).

MPI adalah teknik pemrograman yang berdasarkan data paralel dengan *Single Program Multi Data* (SPMD). Maksudnya adalah setiap proses mengeksekusi program yang sama tetapi menggunakan data yang berbeda. Untuk *sharing* data, suatu proses secara eksplisit mengirimkan data kepada proses penerima yang juga menerima data secara ekspilit. MPI bukan merupakan bahasa pemrograman baru, tetapi MPI adalah subprogram *library* yang dapat dipanggil dari program C dan FORTRAN 77 (Pahlevi,2008).

MPI adalah perangkat lunak yang memungkinkan sekumpulan komputer seperti suatu sistem komputer paralel dan dapat digunakan sebagai sebuah sumber daya komputasi yang koheren. Komputer tersebut bisa berupa *workstation, multiprocessor, specialized grapich engine* sampai dengan *vector super computer* yang dihubungkan dengan jaringan. MPI memungkinkan eksekusi program pada setiap mesin dapat dikendalikan oleh *user* dan menjadi lingkungan komputasi yang *powerfull*. MPI digunakan untuk komputasi paralel dalam sistem yang terdistribusi. Pengguna MPI dapat menuliskan programnya dengan bahas C, C++, FORTRAN 77 dan FORTRAN90 untuk menjalankannya secara paralel dengan memanggil rutin *library* yang sesuai. Karena datanya terdistribusi, biasanya komputasi pada suatu proses akan membutuhkan suatu nilai data yang di *copy* dari proses lainya. Oleh karena itu, bila proses A membutuhkan nilai data X yang disimpan pada memori di data B, maka program tersebut harus meng-*input*-kan suatu baris seperti :

```
If (I am processor A) then
call MPI_Send (X)
```

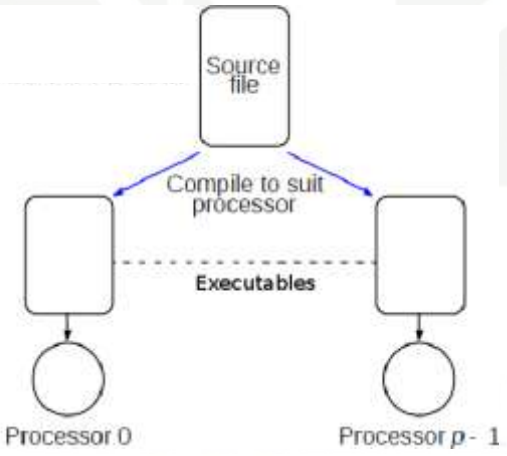
```
else if (I am processor B) then
call MPI_Recev (X)
end
```

Jelas bahwa untuk mengeksekusi sebuah program yang menggunakan MPI akan terlihat berbeda dengan bila menggunakan versi sekuensial. Pengguna harus membagi data masalah ke beberapa proses dan menambah beberapa program untuk mengirimkan nilai yang dibutuhkan dari sebuah proses dimana data tersebut berada ke sebuah proses yang membutuhkan nilai tersebut.

MPI dikembangkan pada tahun 1993-1994 oleh sekelompok peneliti yang berasal dari kalangan pemerintahan, industry dan academia. MPI merupakan salah satu standar pertama untuk menjalankan program paralel prosesor, dan yang merupakan pelopor dalam *basic message passing* (Zulfikar, 2010).

2.3.1 Konsep MPI

Pada MPI, konsep yang di terapkan adalah arsitektur komputer paralel SPMD (*Single Program Multiple Data)* yakni program yang sama dieksekusi oleh tiap prosesor dan statemen control memilih bagian berbeda untuk eksekusi di tiap prosesor.



Gambar 2. 4 Metode MPI Dasar
(Sumber : Eko Didik Widianto, 2012)

Pada MPI, proses-proses yang terdefinis dalam grup komunikasi diberi rank, rang dimulai dari 0 dan seterusnya. Pada program MPI menggunakan statemen kontrol untuk melakukan eksekusi spesifik. Kontrol pada model *master-node*, satu proses (*master*) melakukan satu set action dan semua proses lainnya (*node*s) melakukan action indentik namunn dengan data yang berbeda.

Dalam pembuatan proses, terdapat dua cara yakni:

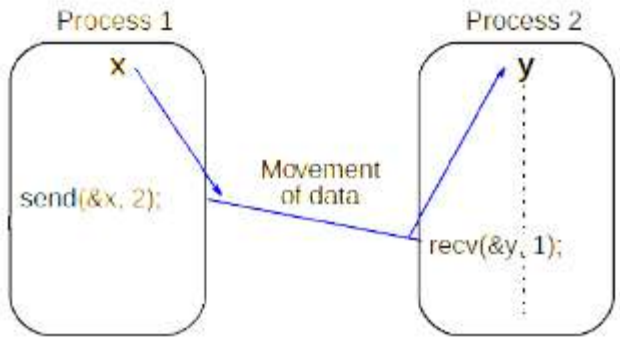
1. Pembuatan proses secara statik

   Semua executable dijalankan bersamaan dan dilakukan saat satu mesin menjalankan program terkompile.

2. Pembuatan proses secara dinamik

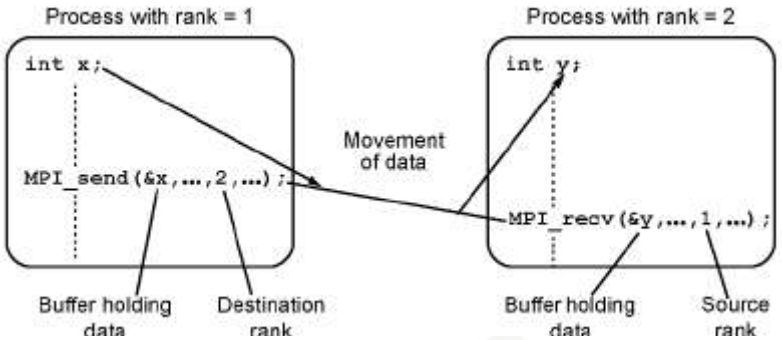
   Satu prosesor mengeksekusi proses *master*. Prosesor lain menjalankan proses dari proses *master*.

Metode pengiriman dan penerimaan message pada MPI yakni mengirimkan sebuah message antar proses menggunakan rutin send() dan receive () dan MPI menggunakan MPI_send() dan MPI_receive.



Gambar 2. 5 Passing message antar proses menggunakan library call send() dan recv()

(Sumber : Eko Didik Widianto, 2012)



Gambar 2. 6 Passing message antar proses menggunakan library call MPI_send() dan MPI_recv()

(Sumber : Eko Didik Widianto, 2012)

Proses MPI dapat dikelompokkan ke dalam gup, tiap message dikirim dalm suatu konteks dan harus diterima dalam konteks yang sama. Grup dan konteks membentuk communicator yakni mendefinisikan domain komunikasi dan set proses yang diijikan untuk berkomunikasi antar mereka. Sebuah proses dapat diidentifikasi dengan rank dalam grup yang berasosiasi dengan suatu communicator (Eko Widianto, 2012).

2.3.2 Fungsi-fungsi Dasar MPI

Dibawah ini adalah daftar fungsi-fungsi MPI yakni fungsi inisialisasi dan fungsi dasar *send/receive*.

Tabel 2. 1 Daftar Fungsi-fungsi Dasar MPI (Berney,2009 dikutip oleh Zulfikar,2010)

Nama Fungsi	Tujuan Fungsi	*Syntax*
MPI_Init	Inisialisasi lingkungan MPI, fungsi ini hanya boleh dipanggil sekali, sebelum penggunaan fungsi-fungsi MPI yang lain	MPI_Init (&argc,&argv) MPI_INIT (ierr)
MPI_Comm_size	Untuk menentukan jumlah prosesor yang tergabung dalam sebuah *communicator* (contohnya	MPI_Comm_size (comm.,&size) MPI_COMM_SIZE (comm.,size,ierr)

Nama Fungsi	Tujuan Fungsi	*Syntax*
	MPI_COMM_WORLD)	
MPI_Comm_rank	Untuk menetukan *rank* dari prosesor yang memanggil fungsi ini dalam sebuah *communicator* (contohnya MPI_COMM_WORLD)	MPI_Comm_rank (Comm,&rank) MPI_COMM_RANK (comm.,rank,ierr)
MPI_Abort	Untuk menghentikan semua proses MPI yang berjalan dalam *communicator*	MPI_Abort (comm, errorcode) MPI_ABORT (comm.,errorcode,ierr)
MPI_Get_processor_name	Untuk mendapatkan nama prosesor tiap *node*	MPI_Get_processor_name (&name, &resultlength) MPI_Get_PROCESSOR_Name (name,resultlength,ierr)
MPI_Initialized	Menunjukkan apakah telah dipanggil MPI_Init	MPI_Initialized (&flag) MPI_INITIALIZED (flag,ierr)
MPI_Wtime	Untuk melihat waktu yang telah berjalan semenjak panggilan terakhir terhadap fungsi ini sebelumnya	MPI_Wtime () MPI_WTIME ()
MPI_Wtick	Untuk melihat resolusi dalam detik dari MPI_Wtick	MPI_Wtick () MPI_WTICK ()
MPI_Finalize	Untuk menghentikan lingkungan eksekusi MPI pada *communicator*	MPI_Finalize () MPI_FINALIZE (ierr)
MPI_Send	Fungsi dasar untuk mengirim data	MPI_Send (&buf, count, datatype, dest, tag, comm) MPI_SEND (buf, count,datatype, dest, tag, comm.,ierr)
MPI_Recv	Fungsi dasar untuk memerintahkan komputer agar menunggu data yang akan masuk	MPI_Recv (&buf,count, datatype, source, tag, comm, &status) MPI_RECV (buf, count, datatype,source, tag, comm.,status,ierr)
MPI_Bcast	Mem-*broadcast* sebuah pesan ke semua prosesor yang tergabung dalam sebuah *communicator* (contohnya MPI_COMM_WORLD)	MPI_Bcast (&buffer, count, datatype, root, comm.) MPI_BCAST (buffer, count,datatype, root, comm, ierr)

2.3.3 OpenMPI

OpenMPI merupakan implementasi *Massege Passing Interface* menggunakan *open source* yang dikembangkan dan dikelola oleh *Cosnsortium of academic,*

research dan industry partners. OpenMPI mampu menggabungkan teknologi dan sumber daya dari *High Performance Computing* untuk membangun pustaka MPI terbaik. Fitur-fitur yang terdapat dalam OpenMPI yakni (www.open-mpi.org) :

1. Mendukung jaringan heterogenitas
2. Mendukung *job schedulers* yang banyak
3. *Single library* mendukung untuk semua jaringan
4. Mendukung banyak sistem operasi (32 bit dan 64 bit)
5. Performa yang tinggi pada semua *platform*

2.4 Deteksi Tepi

Secara umum, pengolahan citra digital menunjukkan pada pemrosesan gambar 2 dimensi menggunakan komputer. Dalam konteks yang lebih luas, pengolahan citra mengacu pada pemrosesan setiap data 2 dimensi. Citra merupakan sebuah larik (*array)* yang nilai-nilai real maupun komplek yang direpresentasikan dengan deretan bit tertentu. Citra atau *image* adalah istilah lain untuk gambar sebagai salah satu komponen multimedia yang memegang peranan sangat penting sebagai bentuk informasi visual. Citra mempunyai karakteristik yang tidak dimiliki oleh teks, yaitu citra kaya akan informasi.

Citra atau gambar merupakan salah satu komponen dalam deteksi tepi yang bersifat 2D. Teknologi dasar untuk menciptakan dan menampilkan warna pada citra digital berdasarkan pada penelitian bahwa sebuah warna merupakan kombinasi dari tiga dasar yaitu merah, hijau, dan biru.

Dalam pemrosesan citra terdapat suatu proses dengan menggunakan deteksi tepi, yang mana proses ini ditujukan untuk mendapatkan batas garis pada suatu tepian citra merepresentasikan objek-objek yang terkandung dalam citra tersebut, baik bentuk, ukurannya dan informasi tentang teksturnya.

Tepi atau sisi dari sebuah obyek adalah daerah dimana terdapat perubahan intensitas warna yang cukup tinggi. Proses deteksi tepi (*edge detection)* akan melakukan konversi terhadap daerah ini menjadi dua macam nilai yaitu intensitas warna rendah atau tinggi.

Gradient adalah hasil pengukuran perubahan dalam sebuah fungsi intensitas, dan sebuah citra dapat dipandang sebagai kumpulan beberapa fungsi intensitas kontinyu sebuah citra. Perubahan mendadak pada nilai intensitas dalam suatu citra dapat dilacak menggunakan perkiraan diskrit pada gradient. Gradient disini adalah kesamaan dua dimensi dari turunan pertama dan didefinisikan sebagai vector (Lusiana, 2013).

Deteksi tepi banyak dipakai untuk mengidentifikasi suatu objek dalam sebuah gambar. Tujuan dari deteksi tepi adalah untuk menandai bagian yang menjadi detail citra dan memperbaiki detail citra yang kabur karena adanya kerusakan atau efek akuisisi data. Dalam citra, sebagian besar informasi terletak pada batas antara kedua daerah yang berbeda (Yulianto, Nataliani, & Kurniawan, 2009).

Ada beberapa metode terkenal dan banyak digunakan untuk pendektesian tepi di dalam citra, yaitu operator Robert, operator Prewiit dan operator Sobel. Medote Sobel paling banyak digunakan sebagai pelacak tepi karena kesederhanaan dan keampuhannya (Munir, 2004). Kelebihan dari metode ini adalah kemampuan untuk mengurangi *noise* sebelum melakukan perhitungan deteksi tepi. Masing-masing metode deteksi memiliki sub metode yang cukup banyak, tetapi metode deteksi citra yang baik adalah metode yang dapat mengeliminasi derau (*noise)* yang semaksimum mungkin (Ballard & Brown, 1982).

Deteksi tepi operator Sobel dipekenalkan oleh Irwin Sobel pada tahun 1970. Operator ini identik dengan bentuk matriks 3x3 atau jendela ukuran 3x3 pixel, dengan Gx gan Gy dihitung menggunakan kernel (*mask)* seperti pada Gambar 2.6 (Munir, 2004).

-1	-2	-1
0	0	0
1	2	1

G_x

-1	0	1
-2	0	2
-1	0	1

G_y

Gambar 2. 7 Matriks Operator Sobel

Untuk mendapatkan nilai maksimum dari operator Sobel adalah dengan menghitung *magnitude* dari gradient untuk mendapatkan kekuatan tepi citra terhadap warna kecerahannya dengan persamaan berikut :

$$M = \sqrt{G_x^2 + G_y^2}$$

Gambar 2.7 Persamaan Menghitung *Magnitude* Gradien

Karena menghitung akar sulit menghasilkan nilai real, maka dalam mencari *magnitude* dapat disederhanakan perhitungannya. Besarnya *magnitude* gradient dapt dihitung lebih cepat dengan menggunakan persamaan berikut :

$$M = |G_x| + |G_y|$$

Gambar 2.8 Persamaan Lain Menghitung *Magnitude* Gradien

Operasi konvolusi bekerja dengan menggeser kernel piksel per piksel, yang hasilnya kemudian disimpan dalam matriks baru. Konvolusi pertama dilakukan terhadap piksel yang bernilai 1 (di titik pusat *mask).*

3	4	2	5	1
2	1	6	4	2
3	5	7	1	3
4	2	5	7	1
2	5	1	3	2

A

-1	0	1
-2	0	2
-1	0	1

B

1	2	1
0	0	0
-1	-2	-1

C

3	4	2	5	1
2	18	6	4	2
3	5	7	1	3
4	2	5	7	1
2	5	1	3	2

D

Gambar 2. 8 A. Citra Asli B. G_x C. G_y D. Hasil Konvolusi

Hasil konvolusi citra yang bernilai 18 diperoleh dengan perhitungan berikut :

$G_x = (3)(-1) + (2)(-2) + (3)(-1) + (2)(1) + (6)(2) + (7)(1) = 11$

$G_y = (3)(1) + (4)(2) + (2)(1) + (3)(-1) + (5)(-2) + (7)(-1) = -7$

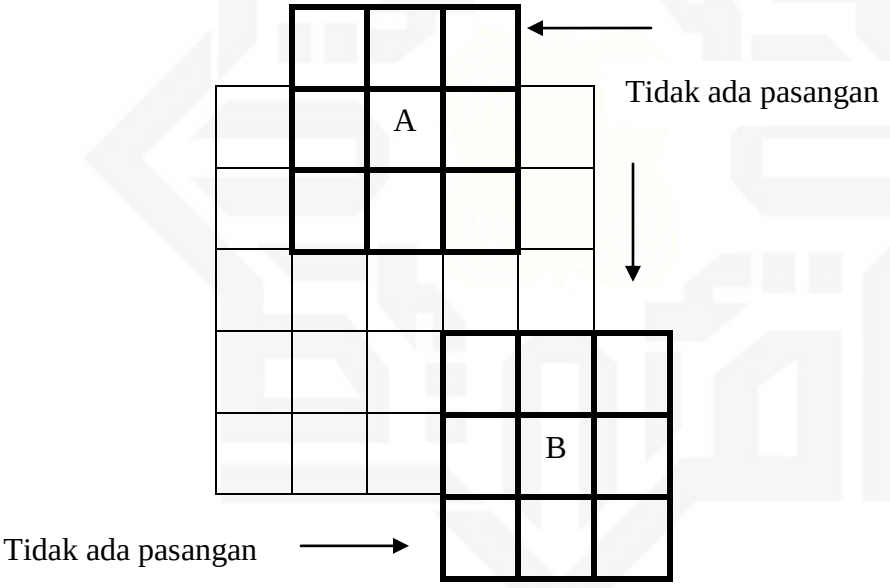
$M = |G_x| + |G_y| = |11| + |-7| = 18$

Dengan demikian, nilai 1 diubah menjadi nilai 18 pada citra keluaran.

Dalam konvolusi terdapat dua kemungkinan yang ditemukan, yakni diselesaikan dengan cara berikut (Munir, 2004):

a. Untuk hasil konvolusi menghasilkan nilai negative, maka nilai tersebut dijadikan 0.
b. Jika hasil konvolusi nilai piksel lebih besar daripada nilai keabuan maksimum, maka nilai tersebut dijadikan nilai keabuan maksimum.

Pada matriks Sobel dengan *mask* 3 x 3, terlihat bahwa tidak semua piksel dikenai konvolusi yaitu baris dan kolom yang terletak di tepi citra. Hal ini disebabkan karena piksel yang berada pada tepi citra tidak memiliki tetangga yang lengkap sehingga rumus konvolusi tidak berlaku pada piksel seperti itu. Gambar 2.10 menjelaskan contoh tentang hal ini. Sebagai contoh, konvolusi tidak mungkin dilakukan pada posisi A dan B.



Gambar 2. 9 Masalah Pada Konvolusi

Masalah konvolusi pada pixel yang tidak mempunyai tetangga selalu terjadi pada pixel-pixel pinggir kanan, kiri, atas, bawah. Solusi untuk masalah ini adalah (Kadir & Susanto, 2013) :

a. Abaikan pixel pada bagian tepi

   Oleh karena pada bagian tepi citra tetangga tidak lengkap, sehingga pixel pada posisi tersebut tidak dikenai konvolusi. Sebagai konsekuensinya, citra yang tidak mengalami konvolusi akan diisi nol atau diisi sesuai pada citra asal. Alternatif lain, bagian yang tidak dip roses tidak diikutkan dalam citra asli. Akibatnya, ukuran citra hasil mengecil.

b. Buat baris dan kolom tambahan pada bagian tepi

   Baris daln kolom ditambah pada bagian tepi sehingga proses konvolusi terdapat dilaksanakan. Dalam hal ini, baris dan kolom baru diisi dengan nilai 0.

Algoritma pembuatan program deteksi tepi pada gambar/citra menggunakan operator Sobel yakni sebagai berikut (Zulfikar, 2010):

1. Mulai
2. Memanggil informasi rank pada *communicator*
3. Waktu keseluruhan
4. Waktu mulai baca gambar
5. Baca gambar/citra RGB
6. Ubah gambar/citra RGB menjadi gambar/citra *grayscale*
7. Hitung jumlah element pada matriks *grayscale*
8. Tentukan baris dan kolom pada matriks *grayscale*
9. Henti waktu baca gambar
10. Mulai waktu pengiriman gambar
11. *Broadcast* jumlah element matriks ke setiap *nodes*
12. Periksa apakah rank = size-1, jika rank = size-1 maka lanjutkan langkah selanjutnya jika tidak maka lanjutkan ke langkah 17
13. Bagi kolom dan citra dengan banyaknya jumlah *nodes*
14. Kirim semua data ke masing-masing *node/*rank dengan menggunakan MPI_Scatter
15. Hentikan waktu pengiriman gambar
16. Mulai waktu operasi Sobel

17. Periksa batas wilayah baris pertama, jika batas wilayah baris pertama = 0 maka lanjutkan ke langkah berikutnya jika tidak maka lanjutkan ke langkah 19
18. Salin nilai 1 ke wilayah baris pertama
19. Periksa batas wilayah baris terakhir, jika batas wilayah baris terakhir > -1 maka lanjutkan ke langkah berikutnya jika tidak maka lanjutkan ke langkah 21
20. Salin objek *grey* dengan *method rows* ke wilayah baris terakhir kemudian dikurangi 1
21. Looping baris pada citra secara keseluruhan
22. Hitung nilai gradient x
23. Hitung nilai gradient y
24. Hitung *magnitude* dari gradient x dan gradient y
25. Periksa apakah nilai matriks 255<sum<0, jika nilai matrik 255<sum<0 maka lanjutkan langkah berikutnya jika tidak maka kembali ke langkah 24
26. Akses nilai piksel dan ubah nilai matriks menjadi nilai intensitas citra
27. Hentikan waktu operasi Sobel
28. Mulai waktu penyatuan gambar
29. Kirim semua data ke rank 0 menggunakan MPI-Gather
30. Hentikan waktu penyatuan gambar
31. Hentikan waktu keseluruhan
32. Periksa apakah rank = 0, jika rank = 0 lanjutkan ke langkah berikutnya jika tidak proses selesai
33. Simpan citra deteksi tepi pada folder yang telah ditentukan
34. Tampilkan waktu eksekusi
35. Selesai

2.5 Visi Komputer (*Computer Vision*)

Visi komputer adalah proses otomatis yang mengintegrasikan sejumlah besar proses untuk citra, pengenalan dan pembuatan keputusan. Visi komputer mencoba

meniru cara kerja sistem visual manusia (*human vision)* yang sesungguhnya sangat kompleks, bagaimana manusia melihat objek dengan indra penglihatan, lalu citra objek diteruskan ke otak untuk diinterpretasi sehingga manusia mengerti objek apa yang tampak dalam pandangan mata. Selanjutnya hasil interpretasi ini digunakan untuk pengambilan keputusan.

Dalam melakukan pengenalan sebuah objek di antara banyak objek dalam citra, komputer harus melakukan proses segmentasi terlebih dahulu. Segmentasi adalah memisahkan citra menjadi bagian-bagian yang diharapkan merupakan objek-objek tersendiri atau membagi suatu citra menjadi wilayah-wilayah yang homogeny berdasarkan kriteria keserupaan tertentu antara derajat keabuan suatu piksel dengan derjat keabuan piksel-piksel tetangganya (Wijaya & Prayudi, 2010).

Definisi pengolahan citra adalah pengolahan suatu citra dengan menggunakan komputer secara khusus untuk menghasilkan suatu citra yang lain. Visi komputer dapat didefisikan setara dengan pengertian pengolahan citra yang dikaitkan dengan akuisisi citra, pemrosesan, klasifikasi, pengambilan keputusan yang diikuti dengan pengidentifikasian citra. Tugas-tugas seperti pengidentifikasian tanda tangan, pengidentifikasian tumor pada suatu citra resonansi magnetic, pengenalan objek pada citra satelit, pengidentifikasian wajah, penempatan sumber daya mineral dari suatu citra dan penghasilan gambaran tiga dimensi dan potongan citra dua dimensi dianggap berada dalam ruang lingkup visi komputer (Fadlisyah, 2007).

Proses-proses di dalam visi komputer dapat dibagi menjadi tiga aktivitas yaitu sebagai berikut : (Jain, Kasturi, & Schunck, 1995) :

a. Memperoleh atau mengakuisisi citra digital
b. Melakukan teknik komputasi untuk memproses atau memodifikasi data citra (operasi-operasi pengolahan citra)
c. Menganalisis dan menginterpretasi citra serta menggunakan hasil pemrosesan untuk tujuan tertentu, misalnya memandu robot, mengontrol peralatan, memantau proses manafaktur dan lain-lain.

Visi komputer meliputi pengolahan citra dan pengenalan pola. Pengolahan citra berkaitan dengan manipulasi dan analisis gambar. Sub area utama pada pengolahan citra yakni :

a. Digitisasi dan kompresi
b. *Enchancement,* restorasi dan rekonstruksi
c. Pencocokan
d. Pendeskripsian dan pengenalan

Digitasi adalah proses pengkonversian gambar menjadi bentuk diskrit dan kompresi mencakup coding efisien atau pendekatan gambar digital untuk menghemat tempat penyimpanan atau kapasitas *channel.* Teknik perbaikan dan restorasi berkaitan dengan peningkatan kualitas dari citra dengan kontras yang rendah, blur, ataupun *noisy.* Teknik pencocokan dan pendeskripsian berkaitan dengan perbandingan dan pelapisan gambar yang satu dengan gambar yang lainya, pembagian gambar menjadi beberapa bagian serta pengukuran hubungan antara bagian-bagian tersebut (Kulkarni, 2001).

Ada beberapa perangkat lunak yang digunakan untuk visi komputer diantranya Aforge.Net, VXL dan OpenCV. OpenCV adalah singkatan dari *Open Computer Vision*, yaitu *library open source* yang dikhususkan untuk melakukan pengolahan citra. *Library* ini ditulis dengan bahasa C dan C++, serta dapat dijalankan dengan Linux, Windows dan Mac OS. OpenCV dirancang untuk efesiensi komputasional dan fokus yang kuat pada aplikasi *real-time.* Tujuannya adalah agar komputer mempunyai kemampuan yang mirip dengan pengolahan visual pada manuasia. OpenCV memiliki API (*Application Programming Inetrface)* untuk pengolahan tingkat tinggi maupun tingkat rendah. Pada OpenCV juga terdapat fungsi-fungsi siap pakai untuk me-*load*, menyimpanm serta nebgakuisisi gambar dan video (Bradski & Kaehler, 2008).

Library OpenCV memiliki fitur-fitur sebagai berikut (Bradski & Kaehler, 2008) :

a. Manipulasi data gambar (mengalikasi memori, melepaskan memori, menduplikasi gambar, mengatur serta mengkonversi gambar).

b. *Image*/Video I/O (bisa menggunakan kamera yang sudah didukung oleh *library* ini).
c. Manipulasi matriks dan vector, serta terdapat juga *routines* aljabar linier (*products, solvers, eigenvalues).*
d. Pengolahan citra dasar (penapisan, pendekstesian tepi, *sampling* dan interpolasi, konversi warna, operasi morfologi, histogram, piramida citra).
e. Analisis structural.
f. Kalibrasi kamera.
g. Pendektesian gerakan.
h. Pengenalan objek.
i. GUI dasar (menampilkan gambar/video, mengontrol *mouse/keyboard, scrollbar).*
j. *Image labeling* (garis, kerucut, polygon, penggambaran teks).

Mat adalah salah satu *static method* dari *class additional core. Class mat* mewakili *array* 2D (dua dimensi) atau *array* multi-dimensi. *Mat* berfungsi untuk menyimpan matriks, gambar, histogram, fitur descriptor, *voxel volume,* dan lain-lain. *Class mat* memiliki sintaks sebagai berikut (OpenCV, 2016):

```
Mat(int rows, int cols, int type)
```

Keterangan sintaks diatas dapat dilihat pada Tabel 2.2.

Tabel 2. 2 Parameter mat

Parameter	Keterangan
Rows	Banyaknya baris dalam *array* 2D
Cols	Banyaknya kolom dalam *array* 2D
Type	Tipe *array*

Imread adalah satalh satu *static method* dari *class additional highgui. Imread* berfungsi untuk me-*load* citra dari *file* yang telah ditentukan dan mengembalikannya. Jika citra tidak dapat di-*load,* hal itu terjadi karena *file* yang hilang, perizinan yang tidak tepat, dan format yang didukung tidak valid. *Imread* memiliki sintak sebagai berikut (OpenCV, 2016):

```
Mat imread (const string& filename, int flags)
```

Keterangan sintaks diatas dapat dilihat pada tabel 2.3.

Tabel 2. 3 Parameter imread

Parameter	Keterangan
Filename	Nama *file* yang akan di-*load*
Flags	Menentukan jenis warna dari citra yang di-*load (*optional)

Imwrite adalah salah satu *static method* dari *class additional highgui. Imwrite* berfungsi untuk menyimpan citra ke *file* yang ditentukan. Format citra dipilih berdasarkan nama ekstensi *file* yang juga ditentukan. *Imwrite* memiliki sintaks sebagai berikut (OpenCV, 2016):

```
bool imwrite(const string& filename, Input img)
```

Keterangan sintaks diatas diatas dapat dilihat pada tabel 2.4.

Tabel 2. 4 Paremeter imwrite

Parameter	Keterangan
Filename	Nama *file* yang akan disimpan
Input img	Citra yang akan disimpan

CvtColor adalah *static method* dari *class additional imgproc. CvtColor* berfungsi untuk mengkonversi citra dari satu ruang warna ke yang lain. *CvtColor* memiliki sintaks sebagai berikut (OpenCV, 2016):

```
void ctvColor (Input src, Output dst, int code)
```

Keterangan sintaks diatas dapat dilihat pada tabel 2.5.

Tabel 2. 5 Parameter cvtColor

Parameter	Keterangan
src	Citra *input*

Parameter	Keterangan
dst	*Output* citra dengan ukuran yang sama dengan citra *input*
code	Kode konversi ruang warna

Code pada sintaks di atas adalah parameter tambahan yang menunjukkan jenis trasformasi yang akan dilakukan. Dalam hal ini menggunakan CV_BGR2GRAY yang berfungsi untuk mengubah citra dari BGR ke format *Grayscale.* OpenCV memiliki fungsi yang sangat bagus untuk melakukan hal ini, sintaks yang digunakan adalah (OpenCV, 2016):

```
cvtColor (src, gray_image, CV_BGR2GRAY);
```

Abs adalah salah satu operasi pada *array. Abs* juga merupakan *static method* dari *class additional core. Abs* berfungsi untuk menghitung nilai absolute dari setiap elemen matriks. *Class abs* memiliki sintaks sebagai berikut (OpenCV, 2016) :

```
MatExpr abs(const Mat & M)

MatExpr abs(conts MatExpr & e)
```

Keterangan sintaks di atas dapat dilihat pada Tabel 2.6.

Tabel 2. 6 Parameter abs

Parameter	Keterangan
M	Matriks
E	Ekspresi matriks

Sum adalah salah satu operasi pada *array. Sum* juga merupakan *static method* dari *class additional core. Sum* untuk menghitung jumlah elemen *array* dan mengembalikan jumlah elemen *array*, secara independen untuk setiap *channel. Class sum* memiliki sintaks sebagai berikut (OpenCV, 2016):

```
sum (InputArray src)
```

Clone adalah *method* dari *class additional core* yang berfungsi untuk membuat salinan lengkap dari objek dan data yang mendasarinya. Metode ini menciptakan salinan lengkap *array*. *Clone* memiliki sintaks sebagai berikut (OpenCV, 2016):

```
Mat Mat :: clone () const
```

Keterangan sintaks diatas dapat dilihat pada Tabel 2.7.

Tabel 2. 7 Parameter clone

Parameter	keterangan
const	Objek yang akan di *clone*

Image.at<uchar>(row,colomn) adalah cara yang paling umum untuk mengakses nilai piksel. Objek *mat* yang disediakan oleh OpenCV digunakan untuk menyimpan data setiap piksel sebagai matriks untuk mempermudah bekerja dengan piksel. Sintaks yang digunakan untuk mengakses nilai piksel cukup sederhana, yaitu (OpenCV, 2016):

```
image.at<uchar> (rows, cols)
```

Keterangan sintaks diatas dapat dilihat pada tabel 2.8.

Tabel 2. 8 Parameter Image.at<uchar>(row,colomn)

Parameter	Keterangan
image	*Pointer* objek *mat*
at	Menyediakan fasilitas untuk mengakses nilai-nilai piksel
<uchar>	Untuk mendapatkan nilai sebagai *unsigned character*
rows	Argumen baris yang mendefinisikan lokasi dalam matriks
cols	Argumen kolom yang mendefinisikan lokasi dalam matriks

2.6 High Performance Linpack

High Performance Linpack merupakan sebuah kumpulan *subroutine* fortan yang dapat menyelesaikan dan menganalisa persamaan linier. Linpack berguna untuk

mengukur kemampuan suatu komputer dalam melakukan komputasi. Linpack di perkenalkan oleh Jack Dongarra dimana untuk mengukur kecepatan komputer yang dimilikinya dalam menyelesaikan permasalahan nilai kepadatan n dari sistem persamaan linier $Ax = b$ yang mana permasalahaan ini merupakan permasalahan umum diantara para insinyur.

Target dari Linpack yakni memperkirakan seberapa cepat sebuah komputer dapat menyelesaikan permasalahan nyata. Permasalahan tersebut yakni tidak ada satupun proses komputasi dapat memberikan hasil yang akurat pada subuah sistem komputer. Dalam mengukur nilai tertinggi performa suatu sistem komputer dapat menggunakan linpack yang dapat memberikan hasil yang akurat. Puncak dari performa adalah nilai maximal yang dimiliki oleh nilai teori yang dapat di capai, hal ini berkaitan dengan perhitungan frekuensi mesin yang dimiliki, jumlah *cycle* perdetik yang dimiliki, dan nilai waktu yang dimiliki pada setiap *cycle* perdetiknya. Meskipun demikian pada kenyataannya nilai performa nyata selalu lebih rendah dari pada nilai puncak yang dimiliki.

Performa *cluster* adalah masalah yang saling terkait yakni aplikasi, algoritma, ukuran masalah, bahasa,kemampuan *compiler*, sistem operasi, arsitektur komputer dan karakteristik *hardware*. Performa yang dimaksud adalah jumlah jutaan *floating point* operasi per detik diukur dalah megaflops (Mflop s-1). Dalam konteks Linpack *benchmark* menggunakan gigaflops (Gflop s-1) sebagai jumlah miliaran *floating point* operasi perdetik. Performa didapatkan dengan menghitung jumlah *floating point* dalam penambahan dan perkalian yang dapat diselesaikan dalam jangka waktu tertentu (Jack J. Dongarra, 2003).

Linpack juga di gunakan pada standarisasi supercomputer yang ada di dunia. Top500 merupakan sebuah project pengukuran performa dimana supercomputer dunia dibuat dengan tujuan menyediakan sebuah basis pengukuran standarisasi yang handal untuk melacak dan mendeteksi trend dari *high-performance computing* di dunia.

High Performance Linpack adalah sebuah paket *software* yang dapat menyelesaikan permasalahan linier dengakn ketepatan 64 Bits pada sebuah *cluster*

komputer. Paket yang di sediakan HPL yakni menguji waktu sebuah program ketika menjalankan sekumpulan tugas. Variable yang dibutuhkan HPL dalam menguji sebuah performa *cluster* adalah sebagai berikut (Linpack) (http://www.netlib.org/benchmark/hpl/, 2016) :

1. Parameter N,

   Parameter N yakni menunjukkan nilai jumlah seberapa besar permasalahan yang akan di ujikan terhadap sistem. Nilai N berguna untuk mengetahui seberapa performa yang dimiliki sebuah komputer. Untuk pemilihan nilai N sebaiknya menggunakan 80 % dari total memory yang ada. Pada penelitian ini, besar memory pada masing-masing *nodes* yakni sebesar 512 sehingga 512 x 4 MB yakni 2 GB, maka dapat dicari besar N yang digunakan pada penelitian dengan menggunakan rumus sebagai berikut :

$$N = \sqrt{\frac{jumla\ h\ memory\ \ yang\ \ digunakan\ \ (GB) * 1024^3}{8}} * 0.8$$

$$= \sqrt{\frac{2 \times 1024^3}{8}} \times 0.8$$

$$= \sqrt{268435456} \text{ x } 0.8$$

$$= 165794 \times 0.8$$

$$= 13107$$

2. Parameter NB,

   Parameter NB yakni menunjukkan nilai *block size* yang digunakan untuk pendistribusian data. Biasanya ukuran blok memberikan hasil yang baik yang di rekomendasikan antara lain berkisar [96, 104, 112, 120, 128, …, 256]. Nilai NB yang digunakan dalam penelitian ini, akan dilakukan percobaan satu persatu dengan menggunakan nilai N yang sama.

3. Parameter P dan Q,

   Parameter P dan Q yakni menunjukkan nilai dari jumlah *core* yang dimiliki masing-masing *nodes* pada. Nilai P sebaiknya lebih kecil dari nilai Q. Pada perhitungan nilai P dan Q, dapat dicari menggunakan rumus :

Jumlah *nodes* x Jumlah CPU pada setiap *nodes*

Hasil dari perkalian diatas difaktorkan sehingga mendapat nilai-nilai faktor dari bilangan tersebut. Pilihlah bilangan faktor yang terdekat dan nilai P < Q. Pada penelitian ini, jumlah *nodes* yang digunakan berjumlah 4 dan masing-masing *nodes* mempunyai 2 *core* pada setiap komputernya, maka nilai P dan Q yang digunakan adalah sebagai berikut :

Jumlah *nodes* x Jumlah CPU pada setiap *nodes*

= 4 x 2

= 8

Faktor-faktor dari 8 adalah 1 x 8, 2x 4 dan akan menghasilkan tabel P dan Q sebagai berikut :

Tabel 2. 9 Menentukan nilai P dan Q

P	X	Q
1	*	8
2	*	4

2.7 Penelitian Terkait

Dalam melakukan penelitian ini, penulis mengambil beberapa referensi-referensi dari penelitian-penelitan sebelumnya. Penelitan yang menjadi referensi penulis yakni :

a. "*Parallel Approach of Sobel Edge Detector in Multicore Platform"* yang dilakukan oleh N.E.A.Khalid, S.A.Ahmad, N.M.Noor, A.F.A. Fadzil dan M.N.Taib pada tahun 2011. Penelitian ini melakukan deteksi tepi pada citra menggunakan operator Sobel dengan menggunakan *multicore* dalam komputasi paralel. Penelitian ini menggunakan citra sebesar 1Kx1K, 2Kx2K, 3Kx3K piksel, dalam mengeksekusi gambar yang akan diolah, penulis meng-*split* gambar menjadi 2 bagian yang akan dikerjakan oleh *multicore* yang ada.

Hasil dari penelitian tersebut yakni pemrosesan paralel lebih baik dari pada pemrosesan sekuensial dalam kecepatan. Dengan meningkatnya jumlah ukuran data yang akan diproses, *multicore* dapat sebagi salah satu alternative untuk mendapatkan proses yang cepat.

b. *"An Efficient Egde Detection Algorithm for Facial Image in Image Mining"* yang dilakukan oleh Vijayarani dan Vinupriya pada tahun 2013. Penelitian ini membandingkan kecepatan waktu eksekusi operator Canny dengan operator Sobel. Penelitian ini menghasilkan bahwa operator Canny melakukan eksekusi lebih cepat dibandingkan dengan operator Sobel. Penelitian ini masih menggunakan pemrograman secara serial untuk membandingkan kedua operator tersebut.

c. "*Uji kerja OCTAVE dengan MPITB menggunakan pustaka LAM/MPI dan OpenMPI*" yang dilakukan oleh M.Dedy Syahputra dan Irfan Akbar tahun 2013. Pada penelitian ini, peneliti membangun *cluster* dengan menggunakan 4 buah unit komputer yakni 1 unit *master* dan 3 lainnya sebagai *node*. Dalam penelitian ini, peneliti memilih Ubuntu 9.10 (Karmic Koala) sebagai sitem operasi dalam *cluster* dan topologi yang digunakan penulis yakni topologi *star*. Dalam pengujian penelitian ini, peneliti menguji kemampuan dalam melakukan komputasi dan melakukan perbandingan antara pustaka LAM/MPI dan OpenMPI. Di dapatkan hasil bahwa kemampuan dari pustaka OpenMPI lebih memiliki kelebihan untuk mendukung komputasi paralel dibandingkan dengan pustaka LAM/MPI.

d. "*Perancangan Cluster Linux untuk Komputasi Paralel OCTAVE"* yang dilakukan oleh Gentur Widyputra pada tahun 2008. Pada penelitian ini, peneliti merancang *cluster* dengan menggunakan 3 unit komputer, sistem operasi yang peneliti gunakan yakni Linux Fedora *Core* 6 dan menggunakan pustaka pemrogaman LAM/MPI serta topologi yang digunakan adalah topologi *star*. Hasil dari penelitian tersebut yakni *cluster* yang dibangun oleh peneliti dapat menjalankan perkalian matriks menggunakan OCTAVE dalam komputasi paralel.