BAB II

LANDASAN TEORI

2.1 Deteksi Tepi *(Edge Detection)*

Deteksi tepi (*edge detection*) adalah suatu proses yang menghasilkan tepi-tepi dari obyek-obyek citra, tujuannya adalah untuk memperbaiki detail dari citra yang kabur, yang terjadi karena *error* atau adanya efek dari proses akuisisi citra. Suatu titik (x,y) dikatakan sebagai tepi (*edge*) dari suatu citra bila titik tersebut mempunyai perbedaan yang tinggi dengan tetangganya (Ramdhani dan Murinto, 2013).

Deteksi tepi merupakan salah satu operasi dasar dari pemrosesan citra. Tepi merupakan batas dari suatu objek. Pada proses klasifikasi citra, deteksi tepi sangat diperlukan sebelum pemrosesan segmentasi citra. Batas objek suatu citra dapat dideteksi dari perbedaan tingkat keabuannya (Purnomo dan Muntasa, 2010).

Tepi atau sisi dari sebuah obyek adalah daerah di mana 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, contoh bernilai nol atau satu. Deteksi tepi akan menghasilkan nilai tinggi apabila ditemukan tepi dan nilai rendah jika sebaliknya (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 dua daerah yang berbeda (Yulianto dkk, 2009).

Pelacakan tepi merupakan operasi untuk menemukan perubahan intensitas lokal yang berbeda dalam sebuah citra. Gradien adalah hasil pengukuran perubahan dalam sebuah fungsi intensitas, dan sebuah citra dapat dipandang sebagai kumpulan beberapa fungsi intensitas kontinu sebuah citra. Perubahan mendadak pada nilai intensitas dalam suatu citra dapat dilacak menggunakan perkiraan diskrit pada gradien. Gradien disini adalah kesamaan dua dimensi dari turunan pertama dan didefinisikan sebagai vektor (Lusiana, 2013). Oleh karena itu teknik deteksi tepi sering digunakan sebagai dasar teknik segmentasi untuk proses segmentasi yang lain.

Ada beberapa metode yang terkenal dan banyak digunakan untuk pendektesian tepi di dalam citra, yaitu operator Robert, operator Prewitt dan operator Sobel. Metode Sobel

paling banyak digunakan sebagai pelacak tepi karena kesederhananaan 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 semaksimal mungkin (Ballard dkk, 1982).

Deteksi tepi operator Sobel diperkenalkan oleh Irwin Sobel pada tahun 1970. Operator ini identik dengan bentuk matriks 3x3 atau jendela ukuran 3x3 piksel (Lusiana, 2013). Operator Sobel melakukan perhitungan secara 2D terhadap suatu ruang di dalam sebuah citra. Operator ini biasanya digunakan untuk mencari gradien dari masing-masing piksel citra *input* yang telah dikonversi ke *grayscale* sebelumnya

Operator Sobel terdiri dari matriks 3x3 masing-masing adalah G_x dan G_y. Matriks *mask* tersebut dirancang untuk memberikan respon secara maksimal terhadap tepi objek baik horizontal maupun vertikal. *Mask* dapat diaplikasikan secara terpisah terhadap *input* citra. Operator Sobel menggunakan kernel operator gradien 3 x 3, dengan koefisien yang telah ditentukan. G_x dan $G_y$dapat dinyatakan sebagai berikut (Munir, 2004):

$$G_x=\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} \text{ dan } G_y=\begin{bmatrix}1 & 2 & 1\\ 0 & 0 & 0\\ -1 & -2 & -1\end{bmatrix} \tag{2.1}$$

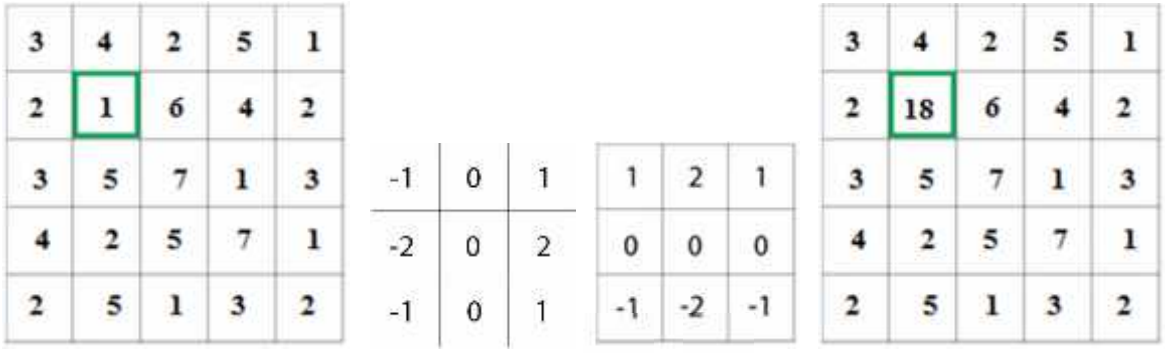
Kernel di atas dirancang untuk menyelesaikan permasalahan deteksi tepi baik secara vertikal maupun horizontal. Penggunaan kernel-kernel ini dapat digunakan bersamaan ataupun secara tepisah (Purnomo dan Muntasa, 2010). Untuk mendapatkan nilai maksimum dari operator Sobel, proses selanjutnya adalah dengan menghitung kekuatan tepi citra terhadap warna kecerahannya dengan cara mencari nilai *magnitude* yang dapat dihitung dengan persamaan sebagai berikut (Munir 2004):

$$M = \sqrt{{G_x}^2 + {G_y}^2} \tag{2.2}$$

Karena menghitung akar adalah persoalan rumit dan menghasilkan nilai real, maka dalam mencari kekuatan tepi *(magnitude)* dapat disederhanakan perhitungannya. Besarnya *magnitude* gradien dapat dihitung lebih cepat lagi dengan menggunakan persamaan sebagai berikut (Munir, 2004):

$$M = |G_x| + |G_y| \tag{2.3}$$

Pada Gambar 2.1 diperlihatkan deteksi tepi dengan operator Sobel. 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*) (Munir, 2004).



(A) (B) (C) (D)

Gambar 2.1. (A) Citra asli, (B) G_x, (C) G_y, (D) Hasil konvolusi

Nilai 18 pada citra hasil konvolusi diperoleh dengan perhitungan berikut (Munir, 2004):

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

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

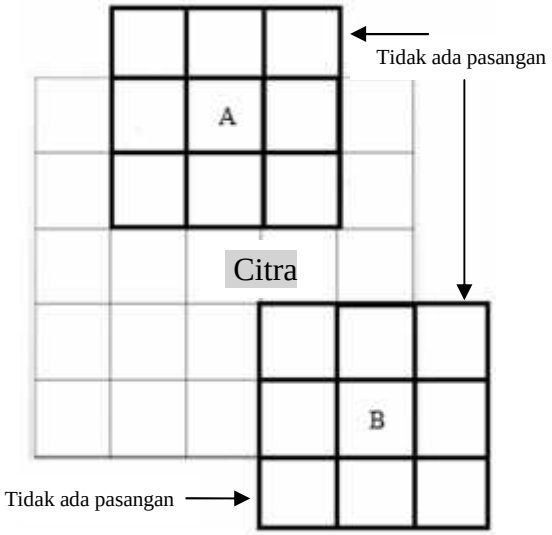
$$M = \sqrt{G_x^2 + G_y^2} = \sqrt{(11)^2 + (-7)^2} \cong |G_x| + |G_y| = |11| + |-7| = 18$$

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

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

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

Pada matriks Sobel dengan kernel 3 x 3, terlihat bahwa tidak semua piksel dikenai konvolusi yaitu baris dan kolom yang terletak di tepi citra (*border*). Hal ini disebabkan karena piksel yang berada pada tepi citra tidak memiliki tetangga yang lengkap sehingga rumus konvolusi tidak berlaku pada piksel seperti itu (Kadir dan Susanto, 2013). Gambar 2.2 menjelaskan contoh tentang hal ini. Sebagai contoh, konvolusi tidak mungkin dilakukan pada posisi A dan B.



Gambar 2.2. Masalah pada konvolusi

Sumber : Kadir dan Susanto, 2013

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

a. Abaikan piksel pada bagian tepi

   Oleh karena pada bagian tepi citra tetangga tidak lengkap, sehingga piksel 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 diproses tidak diikutkan dalam citra hasil. Akibatnya, ukuran citra hasil mengecil.

b. Buat baris dan kolom tambahan pada bagian tepi

   Baris dan kolom ditambahkan pada bagian tepi sehingga proses konvolusi dapat dilaksanakan. Dalam hal ini, baris dan kolom baru diisi dengan nilai nol.

2.2 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 (mata), 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 homogen berdasarkan kriteria keserupaan tertentu antara derajat keabuan suatu piksel dengan derajat keabuan piksel-piksel tetangganya (Wijaya dan Prayudi, 2010) .

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

Visi komputer merupakan proses otomatis yang mengintegrasikan sejumlah besar proses untuk persepsi visual, seperti akuisisi citra, pengolahan citra, klasifikasi, pengenalan (*recognition*) dan membuat keputusan. Visi komputer terdiri dari teknik-teknik untuk mengestimasi ciri-ciri objek di dalam citra, pengukuran citra berkaitan dengan geometri objek dan menginterpretasi informasi geometri tersebut (Jain dkk, 1995).

Proses-proses di dalam visi komputer dapat dibagi menjadi tiga aktivitas yaitu sebagai berikut (Jain dkk, 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 dan menggunakan hasil pemrosesan untuk tujuan tertentu, misalnya memandu robot, mengontrol peralatan, mamantau proses manufaktur, dan lain-lain.

Pengenalan Pola (*pattern recognition*) digunakan untuk mengelompokkan data numerik dan simbolik (termasuk citra) secara otomatis yang dilakukan oleh komputer dengan tujuan untuk mengenali suatu objek di dalam citra dan hasil keluarannya berupa deskripsi objek.

Visi komputer meliputi pengolahan citra dan pengenalan pola. Pengolahan citra berkaitan dengan manipulasi dan analisis gambar. Sub area utama pada pengolahan citra, yaitu (Kulkarni, 2001) :

a. Digitisasi dan kompresi

b. *Enhancement*, restorasi, dan rekonstruksi

c. Pencocokan

d. Pendeskripsian dan pengenalan

Digitisasi 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 lainnya, pembagian gambar menjadi beberapa bagian, serta pengukuran hubungan antara bagian-bagian tersebut (Kulkarni, 2001).

Ada beberapa perangkat lunak yang digunakan untuk visi komputer diantaranya 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 X. OpenCV dirancang untuk efisiensi komputasional dan dengan fokus yang kuat pada aplikasi *real-time*. Tujuannya adalah agar komputer mempunyai kemampuan yang mirip dengan cara pengolahan visual pada manusia. OpenCV memiliki API (*Application Programming Interface*) untuk pengolahan tingkat tinggi maupun tingkat rendah. Pada OpenCV juga terdapat fungsi-fungsi siap pakai untuk me-*load*, menyimpan, serta mengakuisisi gambar dan video (Bradski dan Kaehler, 2008).

Tujuan OpenCV adalah untuk menyediakan infrastruktur visi komputer yang mudah digunakan yang membantu orang-orang dalam membangun aplikasi - aplikasi visi yang mutakhir dengan cepat. *Library* pada OpenCV berisi lebih dari 500 fungsi yang menjangkau berbagai area dalam permasalahan visi, meliputi inspeksi produk pabrik, pencitraan medis, keamanan, antarmuka pengguna, kalibrasi kamera, visi stereo, dan robotika (Bradski dan Kaehler, 2008).

Lisensi *open source* pada OpenCV telah distrukturisasi sehingga pengguna dapat membangun produk komersial menggunakan seluruh bagian pada OpenCV. OpenCV telah digunakan pada banyak aplikasi, produk, dan usaha-usaha penelitian. Aplikasi-aplikasi ini

meliputi penggabungan citra pada peta web dan satelit, *image scan alignment*, pengurangan *noise* pada citra medis, sistem keamanan dan pendeteksian gangguan, sistem pengawasan otomatis dan keamanan, sistem inspeksi pabrik, kalibrasi kamera, aplikasi militer, serta kendaraan udara tak berawak, kendaraan darat, dan kendaraan bawah air. OpenCV juga telah digunakan untuk pengenalan suara dan musik, dimana teknik pengenalan visi diaplikasikan pada citra spektogram suara (Bradski dan Kaehler, 2008).

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

1. Manipulasi data gambar (mengalokasi memori, melepaskan memori, menduplikasi gambar, mengatur serta mengkonversi gambar).
2. *Image*/Video I/O (bisa menggunakan kamera yang sudah didukung oleh *library* ini).
3. Manipulasi matriks dan vektor, serta terdapat juga *routines* aljabar linear (*products, solvers*, *eigenvalues*).
4. Pengolahan citra dasar (penapisan, pendeteksian tepi, *sampling* dan interpolasi, konversi warna, operasi morfologi, histogram, piramida citra).
5. Analisis struktural.
6. Kalibrasi kamera.
7. Pendeteksian gerakan.
8. Pengenalan objek.
9. GUI dasar (menampilkan gambar/video, mengontrol *mouse*/*keyboard*, *scrollbar*).
10. *Image labelling* (garis, kerucut, poligon, penggambaran teks).

Libraries OpenCV menyediakan banyak algoritma visi komputer dasar, dengan keuntungan bahwa fungsi-fungsi tersebut telah diuji dengan baik dan digunakan oleh para peneliti di seluruh dunia. *Libraries* OpenCV juga menyediakan sebuah modul untuk pendeteksian tepi objek yang menggunakan algoritma deteksi Sobel.

OpenCV berisi lebih dari 500 fungsi, meskipun demikian peneliti masih dapat menyelesaikan algoritma deteksi tepi menggunakan metode operator Sobel cukup dengan menggunakan beberapa fungsi OpenCV .

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 deskriptor, *voxel volume*, dan lain - lain. *Class mat* memiliki sintaks sebagai berikut (OpenCV, 2014):

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

Keterangan sintaks di atas dapat dilihat pada Tabel 2.1.

Tabel 2.1. Parameter *mat*

Parameter	Keterangan
rows	Banyaknya baris dalam *array* 2D
cols	Banyaknya kolom dalam *array* 2D
type	Tipe *array*

Imread adalah salah 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 sintaks sebagai berikut (OpenCV, 2014):

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

Keterangan sintaks di atas dapat dilihat pada Tabel 2.2.

Tabel 2.2. 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, 2014):

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

Keterangan sintaks di atas dapat dilihat pada Tabel 2.3.

Tabel 2.3. Parameter *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, 2014):

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

Keterangan sintaks di atas dapat dilihat pada Tabel 2.4.

Tabel 2.4. Parameter *cvtcolor*

Parameter	Keterangan
src	Citra *input*
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 transformasi 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, 2014):

```
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 absolut dari setiap elemen matriks. *Class abs* memiliki sintaks sebagai berikut (OpenCV, 2014):

```
MatExpr abs (const Mat & m)
MatExpr abs (const MatExpr & e)
```

Keterangan sintaks di atas dapat dilihat pada Tabel 2.5.

Tabel 2.5. 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* berfungsi untuk menghitung jumlah elemen *array* dan

mengembalikan jumlah elemen *array*, secara independen untuk setiap *channel*. *Class sum* memiliki sintaks sebagai berikut (OpenCV, 2014):

```
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 dari *array. Clone* memiliki sintaks sebagai berikut (OpenCV, 2014):

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

Keterangan sintaks di atas dapat dilihat pada Tabel 2.6.

Tabel 2.6. Parameter *clone*

Parameter	Keterangan
const	Objek yang akan di-*clone*

Image.at<uchar>(row,column) 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, 2014):

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

Keterangan sintaks di atas dapat dilihat pada Tabel 2.7.

Tabel 2.7. Parameter *Image.at<uchar>(row,column)*

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

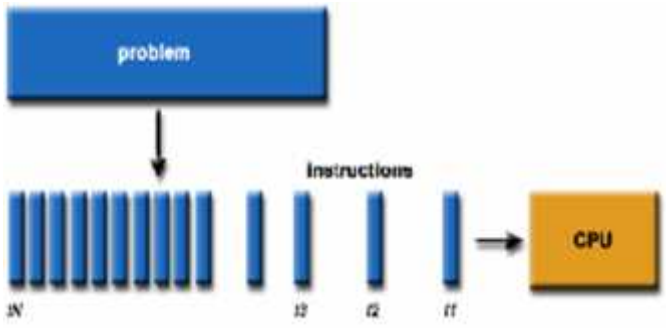
2.3 Pemrograman Paralel

Pemrograman paralel adalah komputasi dua atau lebih *task* pada waktu bersamaan dengan tujuan untuk mempersingkat waktu penyelesaian *tasks* tersebut dengan cara mengoptimalkan *resource* pada sistem komputer yang ada untuk mencapai tujuan yang sama. Pemrosesan paralel dapat mempersingkat waktu eksekusi suatu program dengan cara

membagi suatu program menjadi bagian-bagian yang lebih kecil yang dapat dikerjakan pada masing-masing prosesor secara bersamaan (Hidayat, 2006).

Tujuan dari penggunaan prosesor paralel adalah untuk mengatasi kendala kecepatan dan kapasitas memori, dengan asumsi bahwa sumber daya paralel sudah tersedia. Seperti komputasi pada prosesor tunggal, kinerja komputasi paralel dipengaruhi oleh teknik pemrograman, arsitektur, atau keduanya (Barney, 2009).

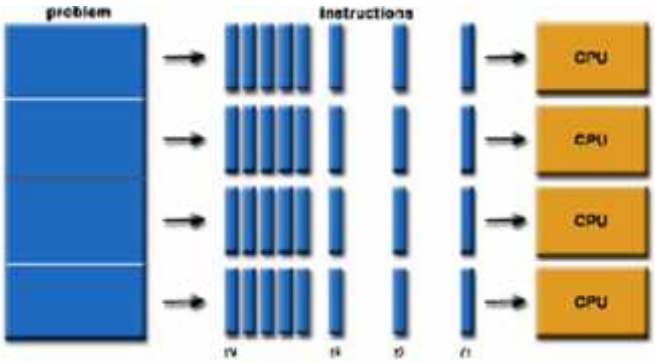
Kinerja prosesor pada dasarnya dilakukan dengan cara menghitung waktu sebelum dilakukan proses pekerjaan dimulai dan diakhiri saat proses pekerjaan selesai. Proses ini pada dasarnya membagi sejumlah *array* ke dalam *sub array* dimana setiap *sub array* dikerjakan oleh satu buah prosesor, secara berurutan dalam kurun waktu tertentu, seperti terlihat pada Gambar 2.3 (Barney, 2009).



Gambar 2.3. Penyelesaian Sebuah Masalah pada Komputasi Tunggal

Sumber : Barney, 2009

Proses paralel adalah proses yang dilakukan pada p buah prosesor, sehingga data yang ada dipecah dalam beberapa bagian, dimana setiap bagian data tersebut di serahkan ke prosesor masing-masing untuk diolah, seperti terlihat pada Gambar 2.4 (Barney, 2009).



Gambar 2.4. Penyelesaian Sebuah Masalah pada Komputasi Paralel

Sumber : Barney, 2009

Dari kedua gambar di atas, dapat disimpulkan bahwa kinerja komputasi paralel lebih efektif dan dapat menghemat waktu untuk pemrosesan data yang banyak daripada komputasi tunggal. Banyak parameter yang dapat digunakan untuk mengukur kinerja sistem paralel, diantaranya adalah waktu komputasi dan *speedup* komputasi paralel (Barney, 2009).

Dalam suatu sistem paralel biasanya *programmer* perlu mengukur kinerja dengan membandingkan waktu proses algoritma paralel dengan waktu proses sekuensial. Pemrograman paralel bertujuan untuk meningkatkan performa komputasi. Semakin banyak hal yang bisa dilakukan secara bersamaan, semakin banyak pekerjaan yang bisa diselesaikan. Batas bawah *speedup* adalah 1 dan batas atasnya adalah jumlah prosesor yang digunakan (*p*). Performa dalam pemrograman paralel diukur dari berapa banyak peningkatan kecepatan (*Speed Up*) yang diperoleh dalam menggunakan teknik paralel. Dengan mendefinisikan t_s dan t_p sebagai waktu proses pemrograman serial dan pemrograman paralel, maka *speedup* dapat dihitung dengan persamaan (Wilkinson dan Allen, 2010):

$$\boldsymbol{SpeedUp} = \frac{\mathbf{T_{serial}}}{\mathbf{T_{paralel}}} \qquad \textbf{(2.4)}$$

T_{serial} = waktu eksekusi menggunakan prosesor tunggal (pemrosesan serial)
$T_{paralel}$ = waktu eksekusi menggunakan N-buah prosesor (pemrosesan paralel)

Speed up pada satu prosesor adalah sama dengan satu, dan *speed up* pada p prosesor bernilai $1 \leq S_p \leq p$. Secara ideal *speed up* meningkat sebanding dengan bertambahnya jumlah prosesor. Jadi jika digunakan p prosesor, *speed up* idealnya adalah p (Wilkinson dan Allen, 2010).

2.4 *Open Multi-Processing* (OpenMP)

OpenMP pertama kali dirilis pada tahun 1997 dan merupakan sebuah *Application Programming Interface* (API) untuk pemrograman paralel yang mendukung multi *processing shared memory* di C, C++ dan Fortran. OpenMP memiliki keuntungan mudah diterapkan pada arsitektur *multicore portable* untuk mengembangkan aplikasi paralel pada *platform* mulai dari *desktop* sampai super komputer. OpenMP terdiri dari satu set perintah *compiler*, rutin *library*, dan variabel lingkungan yang mempengaruhi *runtime* (Kiessling, 2009).

Direktif *compiler* muncul sebagai komentar dalam kode sumber dan diabaikan oleh *compiler* kecuali jika *programmer* memberitahu sebaliknya, biasanya dengan menetapkan bendera *compiler* yang sesuai. Direktif *compiler* OpenMP digunakan untuk berbagai tujuan sebagai berikut (OpenMP, 2013):

- Pemijahan wilayah paralel
- Membagi blok kode di antara *thread*
- Mendistribusikan iterasi *loop* di antara *thread*
- Serialisasi bagian kode
- Sinkronisasi kerja di antara *thread*

Direktif *compiler* memiliki sintaks sebagai berikut:

#pragma omp nama-direktif *clause* (argumen)

Keterangan sintaks di atas dapat dilihat pada Tabel 2.8.

Tabel 2.8. Sintaks dan keterangan format direktif C/C++ OpenMP

No.	Sintaks	Keterangan
1.	#pragma omp	Dibutuhkan untuk seluruh direktif C/C++ OpenMP
2.	Nama-direktif	Direktif OpenMP valid
3.	*Clause* (argumen)	Opsional

Sumber : OpenMP (2013)

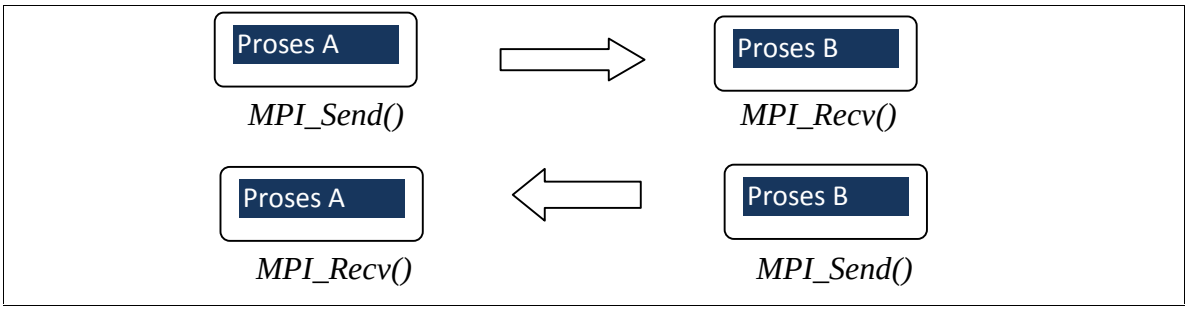
Direktif *parallel for* adalah konsep paralel yang membentuk tim *threads* dan menetapkan bahwa iterasi *looping* akan didistribusikan untuk diksekusi oleh tim *threads*. Untuk menghitung waktu eksekusi pada pemrograman paralel OpenMP digunakan rutin pewaktuan. Rutin pewaktuan digunakan untuk mendukung pengaturan waktu. Fungsi rutin pewaktuan yang digunakan adalah *omp_get_wtime()* yang berfungsi untuk mengembalikan waktu berlalu dalam detik (OpenMP, 2013).

2.5 *Message Passing Interface* (MPI)

Ada banyak sekali bahasa pemrograman paralel yang diperkenalkan dan sebagian besar merupakan bahasa tingkat tinggi untuk menyederhanakan kompleksitas pemrograman paralel. *Message passing interface* (MPI) merupakan API yang umum digunakan untuk membuat aplikasi paralel (Kurniawan, 2010). Konsep dasar mekanisme komunikasi dari MPI adalah perpindahan data dari satu prosesor ke prosesor yang lain

menggunakan fungsi dari MPI *run-time library,* dimana satu prosesor mengirim data dan yang lainnya menerima data tersebut (Kurniawan, 2010).

Hampir sebagian besar komunikasi yang terjadi pada MPI didasarkan pada komunikasi *point to point*, sehingga komunikasi ini sangatlah penting dan sebagai dasar untuk komunikasi pada MPI. Komunikasi *point to point* merupakan proses pertukaran data pada sepasang proses dimana satu proses sebagai pengirim dan satu proses lagi sebagai penerima, seperti terlihat pada Gambar 2.5 (Kurniawan, 2010).



Gambar 2.5. Proses komunikasi *point to point*

Sumber : Kurniawan, 2010

Dari Gambar 2.5 terlihat bahwa proses A mengirim pesan ke proses B, kemudian proses B kembali mengirim pesan ke proses A. Kedua proses tersebut saling melakukan pertukaran pesan dengan saling mengirim dan menerima data diantara keduanya. Pada MPI ada beberapa fungsi yang disediakan untuk mengirim dan menerima data baik secara *blocking* maupun *nonblocking* (Kurniawan, 2010).

Sebelum mengimplementasikan MPI ke dalam suatu program, seorang *programmer* harus menentukan *header*, komunikator dan rutin manajemen lingkungan apa yang akan digunakan. *Header* diperlukan oleh program untuk memanggil fungsi yang ada pada *library* MPI, *header* yang digunakan dalam implementasi MPI adalah *#include <mpi.h>*. Hampir semua fungsi-fungsi atau *routine* menggunakan komunikator sebagai argumen. Salah satu komunikator yang paling sering digunakan adalah MPI_COMM_WORLD (Barney, 2013). MPI juga memiliki beberapa rutin manajemen lingkungan yang sering digunakan. Rutin manajemen lingkungan MPI tersebut antara lain (Kurniawan, 2010):

a. *MPI_Init*, digunakan untuk menginisialisasi eksekusi MPI. Kode program yang digunakan adalah *MPI_Init (&argc, &argv);*

b. *MPI_Comm_size*, digunakan untuk mengembalikan jumlah proses MPI dalam komunikator. Kode program yang digunakan adalah *MPI_Comm_Size (MPI_COMM_WORLD, &size);*

c. *MPI_Comm_rank*, digunakan untuk memanggil proses MPI dalam komunikator. *Rank* sering juga disebut *task ID* dan digunakan untuk menentukan *source* dan *destination* dari sebuah pesan. Kode program yang digunakan adalah *MPI_Comm_rank (MPI_COMM_WORLD, &rank);*

d. *MPI_Wtime*, digunakan untuk menentukan waktu yang dipakai saat prosesor dipanggil (dalam detik). Kode program yang digunakan adalah *MPI_Wtime ();*

e. *MPI_Finalize*, digunakan untuk mengakhiri proses eksekusi MPI. Kode program yang digunakan adalah *MPI_Finalize ();*

Operasi *Nonblocking send/receive* adalah proses yang akan mengembalikan suatu nilai meskipun *buffer* belum penuh dengan data yang akan dikirim/diterima, artinya *nonblocking send/receive* akan mengembalikan nilai walaupun data yang dikirim/diterima belum dieksekusi. Operasi *nonblocking* MPI dikenali dengan nama operasinya dimana dapat dikategorikan menjadi empat bagian yaitu (Kurniawan, 2010):

a. *Immediate,* disingkat dengan I atau i
b. *Buffer,* disingkat dengan B atau b
c. *Synchronous,* disingkat dengan S atau s
d. *Ready,* disingkat dengan R atau r

Pada operasi *non-blocking immediate* digunakan *MPI_Isend()* dan *MPI_Irecv,* karakter I menunjukkan operasi MPI dengan mode segera (*Immediate).* Operasi MPI_*Isend()* dan *MPI_Irecv* merupakan operasi untuk pengiriman dan penerimaan data secara *nonblocking* dan bersifat segera (*immediate*) (Kurniawan, 2010). Deklarasi kode *non-blocking* mode *immediate* seperti terlihat pada Tabel 2.9.

Tabel 2.9. Deklarasi kode *non-blocking* mode *immediate*

Fungsi MPI	**Format penulisan**
Send	int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Recv	int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

Sumber : Kurniawan (2010)

Tabel 2.9 menunjukan parameter-parameter yang digunakan dalam operasi *non-blocking*. Fungsi dan keterangan parameter-parameter tersebut dapat dilihat pada Tabel 2.10 dan Tabel 2.11.

Tabel 2.10. Parameter *non-blocking send immediate*

Parameter	Keterangan	I/O
Buf	*Buffer* data yang akan dikirim	IN
Count	Jumlah *buffer* data	IN
Datatype	Tipe data dari *buffer* data yang akan dikirim	IN
Dest	*Rank* tujuan	IN
Tag	*Message tag*	IN
Comm	*Communicator* yang digunakan	IN
Request	*Communicator request*	OUT

Sumber : Kurniawan, 2010

Tabel 2.11. Parameter *non-blocking recv immediate*

Parameter	Keterangan	I/O
Buf	*Buffer* data yang akan diterima	OUT
Count	Jumlah *buffer* data yang diterima	IN
Datatype	Tipe data dari *buffer* data yang diterima	IN
Source	Sumber *rank* yang akan ditunggu data yang masuk	IN
Tag	*Message tag*	IN
Comm	*Communicator* yang digunakan	IN
Request	*Communicator request*	OUT

Sumber : Kurniawan, 2010