Seminar Nasional Teknologi Informasi dan Kedirgantaraan (SENATIK)
Vol. II, 26 November 2016, ISSN: 2528-1666

Pemrosesan Paralel pada *Low Pass Filtering* Menggunakan *Transform Cosinus* di MPI (*Message Passing Interface*)

Astika Ayuningtyas
Sekolah Tinggi Teknologi Adisutjipto Yogyakarta,
Jl Janti Blok R Lanud Adisutipto, Yogyakarta
E-mail : astika.stta@yahoo.com

Abstract

Parallel processing is a process of calculating two or more tasks simultaneously through the optimization of the computer system resource, one treatment models is a desktop system. The model allows to perform parallel processing between computers with specifications different computers. An implementation of a model network of workstations using MPI (Message Passing Interface). In this study, applied to the case of the low-pass filtering (LPF), a process in the image or the image of the shape of the filter that retrieves the data at low frequencies. filtering programs lowpass using the cosine transform MPI implemented by modifying the algorithm in the process on each node (computer). Depending on the test results, so that the processing speed of a parallel system is influenced by the number of nodes / processes and the number of frequency components that are processed. In the treatment of single larger process, the time it takes more and more and the value prop affects only the amount of high frequency data is filtered on the field. While parallel processing of more and more computers involved in the filter calculation process low-pass, plus the time required to perform the calculation.

Keywords: *parallel processing, network of workstation, message passing interface, low pass filtering*

1. Pendahuluan

Pemrosesan paralel merupakan teknik komputasi menggunakan dua atau lebih komputer untuk menyelesaikan suatu tugas dalam waktu yang simultan dengan cara mengoptimalkan *resource* pada sistem komputer yang ada untuk dapat mencapai tujuan yang sama [4]. Sederhananya pada pemrosesan paralel semakin banyak hal yang bisa dilakukan secara bersamaan (dalam waktu yang simultan), maka semakin banyak pekerjaan yang bisa diselesaikan.

Proses kerja dari paralel adalah dengan membagi beban kerja dan mendistribusikannnya pada komputer-komputer lain yang terdapat dalam sistem untuk menyelesaikan suatu masalah. Salah satu model dari pemrosesan paralel adalah *network of workstations*. Model ini terdiri dari sejumlah komputer dengan spesifikasi yang berbeda-beda dihubungkan pada suatu jaringan yang sama, di mana mereka akan bekerja sama untuk menyelesaikan instruksi yang diberikan. Salah satu penerapan dari model tersebut adalah MPI (*Message Passing Interface*). MPI adalah standar *interface* dari model *message passing* yang mendukung komunikasi baik dengan tipe *point to point* maupun yang bersifat kolektif [5]. Di dalam MPI mendukung *thread safe* yang penting dalam *symmetric multiprocessor* pada lingkungan jaringan komputer yang heterogen. Pada penelitian ini akan menerapkan MPI untuk pemrosesan paralel untuk proses *Low Pass Filtering* (LPF) dengan *transform cosinus*. LPF adalah suatu proses pada citra dari bentuk filter yang mengambil data pada frekuensi rendah dan membuang frekuensi tinggi yang mempunyai tujuan mengurangi noise pada suatu citra [3].

Pada LPF di penelitian ini akan menghitung semua komponen frekuensi kecuali komponen frekuensi yang paling tinggi, dan kemudian mentransformasikannya kembali ke domain waktu, dengan meng-*overwrite array* x (dimana *array* x

adalah sebuah *array* dengan jumlah anggota n). Program *low pass filtering* menggunakan *transform cosinus* ini menerapkan MPI, maka diperlukan suatu modifikasi terhadap algoritma tersebut agar proses dikerjakan tiap node tidak sama. Setiap proses pada node akan diberikan nomor *id* yang unik untuk membedakan proses yang satu dengan yang lainnya. Tipe komunikasi yang digunakan pada MPI adalah *collective communication*, di mana operasi komunikasi tersebut berlangsung dalam sebuah grup proses yang melibatkan lebih dari dua sistem komputer. Tujuan dari penelitian ini adalah memperpendek waktu pemrosesan pada algoritma LPF dengan *transform cosinus* menggunakan pemrosesan paralel di MPI, sehingga lebih efisien dalam prosesnya. Adanya paralelisasi yang diimplementasikan pada algoritma LPF ini, diharapkan waktu pengerjaan untuk algoritma tersebut akan semakin cepat.

2. Metode Penelitian

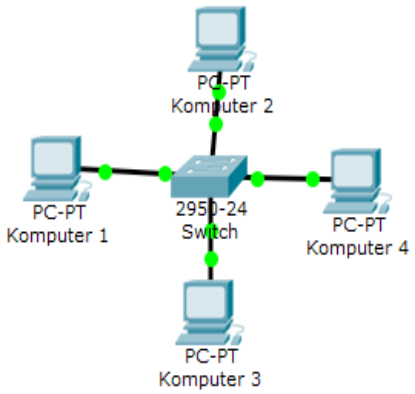
Metode Penelitian meliputi analisis yang menguraikan kebutuhan obyek data untuk pemrosesan paralel dengan MPI pada algoritma LPF dengan *transform cosinus*, arsitektur dari sistem pemrosesannya dengan memparalelkan algoritma LPF untuk dieksekusi dengan banyak komputer, metode yang dipakai untuk menyelesaikan masalah secara paralel, implementasi, dan pengujian dari algoritma yang sudah dimodifikasi secara *parallel processing* dengan membandingkan hasil pengujiannya menggunakan *single processing*.

2.1. Tahapan Analisis

Pada tahapan pertama menganalisis terkait kebutuhan obyek yang akan digunakan untuk membangun proses paralel pada pemrograman yang akan dibangun. Pada proses analisis ini meliputi analisis algoritma yang akan dimodifikasi secara paralel dengan menggunakan MPI. Algoritma yang dibangun pada sistem ini nantinya akan dimodifikasi menjadi suatu proses terkecil yang akan ditempatkan pada masing-masing komputer/*slave* yang terhubung pada satu jaringan yang sama, dimana proses yang dikerjakan setiap *node* tidak sama.

2.2. Arsitektur Sistem Pemrosesan Paralel Pada LPF

Pada program *low pass filtering* menggunakan *transform cosinus* ini menerapkan MPI, maka diperlukan suatu modifikasi terhadap algoritma tersebut agar proses dikerjakan tiap node tidak sama. Program LPF akan dijalankan pada beberapa komputer yang menggunakan MPI untuk proses paralelisasi-nya (lihat Gambar 1).



Gambar 1 Aristektur Jaringan Paralel yang dibangun di MPI

LPF menghitung semua komponen frekuensi kecuali untuk ***r = floor(prop x n)*** komponen frekuensi paling tinggi, dan kemudian mentransformasikan kembali ke domain waktu, dengan meng-*overwrite array* ***x***. Data domain waktu awalnya berada di ***x*** pada node 0, dan pada akhirnya, ketika semua komputasi sudah selesai, akan diletakkan kembali di ***x*** pada node 0. Dengan asumsi bahwa n-r dapat dibagi bulat oleh p, yaitu jumlah node MPI yang digunakan untuk pemrosesan data secara paralel, dan andaikan s adalah *chunk* yang diperoleh dari hasil bagi bulat dari (n-r) dengan p yang akan digunakan untuk paralelisme (lihat Gambar 2).

Xk	X1	..	..	Xn-r	..	..	Xn

Ck	C1	..	..	Cn-r	0	0	0

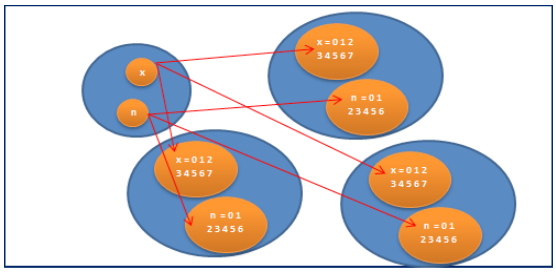
Gambar 2 Proses modifikasi LPF secara paralel

Di sini masing-masing node akan menghitung sebuah chunk *s* di *array c*, dan kemudian masing-masing node akan menghitung sebuah chunk berukuran *s* di *array x*. Rumus untuk transform cosinus satu dimensi dan inverse-nya pada nomor persamaan 1 dan 2 [2].

$$c_k = \sum_{j=0}^{n-1} x_j \cos(\frac{pi}{n}(j+0.5)k) \quad (1)$$

$$c_k = \sum_{j=0}^{n-1} x_j \cos(\frac{pi}{n}(j+0.5)k) \quad (2)$$

MPI_Bcast (broadcast) yang melakukan pengiriman data dari sebuah proses ke semua proses lainnya pada grup (lihat Gambar 3), dan fungsi MPI_Reduce untuk melakukan kebalikannya.



Gambar 3 Proses MPI_Bcast

Operasi broadcast juga diperlukan ketika suatu proses dalam suatu spesifik group mengirim ke seluruh proses pada spesifik group, yaitu semua dataset (array) satu dimensi x dengan jumlah anggota sebanyak n.

2.3. Metode Paralelisasi

Metode untuk memparalelkan proses pada algoritma LPF di penelitian ini memanfaatkan MPI yang merupakan salah satu API (*Application Programming Interface*) pada model *network of workstation.* Model ini lebih menitikberatkan pada komputasi yang berjalan di beberapa komputer yang terhubung dengan satu jaringan yang sama yang mendukung *thread safe* yang penting dalam *symmetric multiprocessor* pada lingkungan jaringan yang heterogen [1]. Heterogen yang dimaksudkan adalah komputer yang dibangun menggunakan spesifikasi yang berbeda-beda. MPI bersifat *extensible*, dimana dapat terus dikembangkan dan memenuhi kebutuhan komputasi pada masa yang akan datang. *Semantic behaviour* yang dimiliki MPI telah terspesifikasi dengan jelas, sehingga dapat menghindari permasalahan kritis seperti *race-conditions* dan *dead lock.* Gambar 3 menunjukkan data x dari indeks 0 hingga n-r semua x harus di-*broadcast* karena untuk menghitung setiap c dibutuhkan semua x. Misalkan jumlah anggota dicontohkan ada 8 data sesuai dengan contoh pada Gambar 2.

2.4. Implementasi Paralel LPF

Tahap implementasi dengan membangun program LPF menggunakan paralel di MPI. Program ini menggunakan bahasa C yang ditambahkan *library* MPI untuk paralelisasi-nya. Input dari program ini menggunakn *argc* dan *argv* yang sering digunakan dengan *command line intepreter*. Input yang diterima bertipe string dengan spasi sebagai penanda dari inputan tersebut. Input yang diminta pada program ini adalah jumlah anggota dan besar nilai dari proposisi (variabel *prop*) serta perintah cetak hasil atau tidak pada layar. Dua inputan tersebut digunakan untuk menghitung jumlah komponen frekuensi paling tinggi yang akan difilter atau dilewatkan untuk tidak dihitung pada LPF.
Misal dapat diilustrasikan sebagai berikut :

1. Misal jumlah anggota n = 100
   Inputan prop = 0.1 sehingga nilai r = floor (prop x n) = 0.1 x 100 = 10
2. Didapatkan jumlah komponen frekuensi paling tinggi sebesar 10, sehingga yang akan diproses LPF adalah sebanyak n-r (100-10).
3. Kemudian dihitung nilai chunk (s) = (n-r)/p = 90/3=30 (misal p=3), untuk dilakukan perhitungan secara paralelisme pada masing-masing node MPI. Data x dari indeks 0 hingga n-r semua x harus dibroadcast karena untuk menghitung setiap c dibutuhkan semua x.

```
 MPI_Bcast(x,n,MPI_INT,0,MPI_COMM_
WORLD);
for(i=start; i<=finish; i++)
  {
    temp =0;
    for(j=0; j<n; j++)
    {
 temp+=x[j]*(cos((M_PI/)*(j+0.5)*i));
    }
    c[i]= floor(temp);     }
```

4. Sebanyak chunk berukuran s dihitung masing-masing node di array c. Pada perhitungan di *array* c dilakukan *MPI_Allgather* karena semua nilai c dibutuhkan oleh semua *node* untuk menghitung x yang menjadi bagiannya.

```
MPI_Allgather(c+start,s,MPI_
INT,c,s,MPI_INT,MPI_COMM_WORLD);
  for (i=start; i<=finish; i++)
  {
  temp =0;
  for (j=1; j<n; j++)
```

```
{
temp+=c[j]*(cos((M_PI/n)*(i+0.5)*j));
}
    x[i] = floor(0.5*c[0] + temp);
}
```

5. Setelah proses berhasil dikerjakan secara paralel pada masing-masing *node*, selanjutnya akan dilakukan pengambilan data dari semua proses *MPI_Gather* dipilih karena hanya node 0 (server 1) yang membutuhkan nilai x yang baru untuk ditampilkan ke suatu proses pada spesifik *communicator*.

```
MPI_Gather(x+start,s,MPI_
INT,x,s,MPI_INT,0,MPI_COMM_WORLD);
```

6. Setelah proses paralel selesai pada LPF, maka hasilnya akan ditampilkan dengan lamanya waktu eksekusi (total komputasi paralel LPF).

```
Time2 = MPI_Wtime();
```

3. Hasil dan Pembahasan

Berdasarkan arsitektur sistem pada program paralel LPF yang ada pada Gambar 1 dan 2 serta proses implementasinya, dilakukan pengujian dengan menggunakan empat buah komputer yang bertindak sebagai *server* ataupun *slave*. Komputer-komputer tersebut terlebih dahulu dilakukan pengaturan *network of workstation*-nya (lihat Gambar 4).

Gambar 4 Pengujian Sistem dengan *Network of Workstation*

Setelah dilakukan pengaturan jaringan, langkah berikutnya melakukan pengaturan untuk proses paralelisasinya dengan meng-*autorisasi* komputer yang akan digunakan untuk membagi proses pada algoritma LPF yang sudah dimodifikasi secara paralel.

```
$ ssh-keygen -t dsa
$ cp /home/mpiuser/.ssh/id_dsa.pub /
home/mpiuser/.ssh/authorized_keys
$ scp /home/mpiuser/.ssh/id_dsa.pub
mpiuser@192.168.1.108:.ssh/authorized_
keys
$ scp /home/mpiuser/.ssh/id_dsa.pub
mpiuser@192.168.1.124:.ssh/authorized_
keys
$ scp /home/mpiuser/.ssh/id_dsa.pub
mpiuser@192.168.1.104:.ssh/authorized_
keys
$ chmod 700 /home/mpiuser/.ssh
$chmod600/home/mpiuser/.ssh/
authorized_keys
```

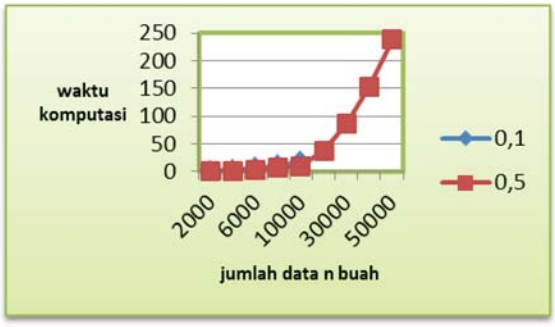
Pengujian pertama dilakukan di *single processing*. Salah satu hasil tampilan di *single processing* dengan data berjumlah 100 pada Gambar 5.

```
PROGRAM LPF DENGAN TRANSFORMASI COSINUS
---------------------------------------------------------------------
Array x:
  1  3  6  7  5  3  5  6  2  9  1  2  7  0  9  3  6  0  6  2  6  1  8  7  9  2  0  2  3  7  5
  9  2  2  8  9  7  3  6  1  2  9  3  1  9  4  7  8  4  5  0  3  6  1  0  6  3  2  0  6  1  5  5
  4  7  6  5  6  9  3  7  4  5  2  5  4  7  4  4  3  0  7  8  6  8  8  4  3  1  4  9  2  0  6  8
  9  2  6  6  4
------------------
Hasil Array c:
------------------
 456 -13   8 -14 -14  15  19 -19  -9  17  -6 -24  28 -12 -18 -20 -12  24   0   6 -42   3 -17 -17  -4  -6
  30   7  -8   4 -46 -16 -19  11  22 -18 -25 -46   0   1  18 -18  19 -12   9 -17  -2   0 -32  17   1  23 -23
  30 -15 -19 -17  21  20  -3 -31  14  14  13  25  -1  -9   6  35  -9 -31   5 -18   8 -37 -27   1   9  -8 -25
  15  27  24   7  16  -4  11   2  26 -12   0   0   0   0   0   0   0   0   0   0
---------------------------------------------
Hasil Array x setelah proses filtering:
---------------------------------------------
  27 130 339 300 307  89 305 254 130 440  38 138 283  88 335 278 164 147 154 241 171 161 306
 417 405 122  -4  83 177 312 295 403 140  65 427 434 356 162 277  86  59 495 104  93 411 229
 329 407 203 233  30 109 352  -7  64 236 213  41  54 255  90 228 265 200 338 325 217 343 393
 210 284 264 187 160 192 253 313 232 177 161  -4 341 409 280 425 377 222 124  71 176   9   2
   0   6   8   9   2   6   6   4
-----------------------------------------------
Node :0 -- Start: 0 -- Finish:89 OKE
Waktu komputasi LPF Transform Cosinus : 0.002123
```

Gambar 5 Pengujian Sistem di *Sigle* ***Processing***

Keseluruhan hasil percobaan di *single processing* terdapat pada Gambar 5.

Gambar 5 Grafik Hasil Pengujian di *Single Proccessing*



Pada Gambar 5 terlihat dari keseluruhan hasil percobaan menunjukkan bahwa semakin banyak jumlah data yang diproses semakin banyak pula waktu komputasi yang diperlukan. Data menunjukkan warna merah bekerja pada *prop* 0,5 dan warna biru

pada *prop* 0,1. *Prop* hanya menentukan banyaknya proposisi frekuensi paling tinggi yang akan difilter. Jadi makin banyak nilai *prop*-nya maka makin banyak pula frekuensi tinggi yang difilter. Setiap penambahan jumlah data dua kali lipat dari data sebelumnya menunjukkan selisih waktunya (lihat Tabel 1).

Tabel 1 Selisih Waktu Komputasi Percobaan di *Single Proccessing*

n_{2000}	n_{4000}	**Selisih waktu**	n_{4000}	n_{6000}	**Selisih waktu**
0,685	2,736	**2,051**	2,736	6,157	**3,421**
0,383	**1,519**	**1,136**	**1,519**	**3,423**	**1,904**
n_{6000}	n_{8000}	**Selisih waktu**	n_{8000}	n_{10000}	**Selisih waktu**
6,157	10,927	**4,77**	10,927	17,102	**6,175**
3,423	**6,083**	**2,66**	**6,083**	**9,483**	**3,4**

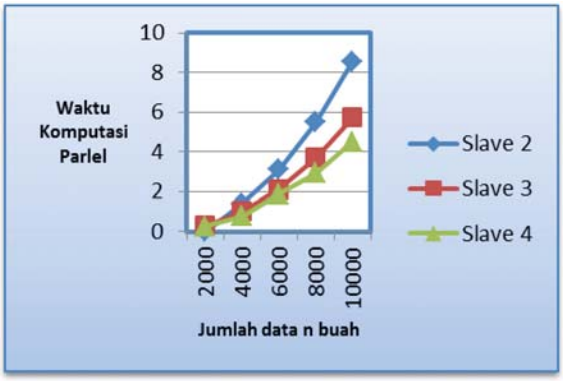
Pada Tabel 1 terlihat bahwa semakin banyak data yang diproses semakin banyak waktu yang diperlukan untuk melakukan perhitungan LPF. Semakin banyak nilai *prop* semakin banyak pula frekuensi tinggi yang terfilter sehingga mengakibatkan waktu pengerjaan semakin sedikit.

Pengujian kedua dilakukan di *parallel processing* dengan menggunakan beberapa *slave* yaitu *slave* 2, *slave* 3, dan *slave* 4 dengan nilai prop 0,1 (lihat Tabel 2).

Tabel 2 Hasil Percobaan di *Parallel Proccessing*

n	**2000**	**4000**	**6000**	**8000**	**10000**
Slave 1 (192.168.1.107 & 192.168.1.108)	0.379	1,395	3,097	5,489	8,555
Slave 2 (192.168.1.107 & 192.168.1.108 & 192.168.1.124)	0,291	0,999	2,104	3,69	5,738
Slave 3 (192.168.1.107 & 192.168.1.108 & 192.168.1.124 & 192.168.1.104)	0,278	0,782	1,865	2,919	4,513

Keseluruhan hasil percobaan di *single processing* terdapat pada Gambar 6. Pada Gambar 6 terlihat dari keseluruhan hasil percobaan menunjukkan bahwa semakin banyak jumlah data yang diproses semakin banyak pula waktu komputasi yang diperlukan. Data menunjukkan warna merah bekerja dengan 3 slave, warna biru dengan 2 slave, dan warna hijau dengan 4 slave. Semakin banyak slave atau node yang bekerja memproses maka semakin sedikit waktu komputasi yang dibutuhkan untuk menghitung LPF-nya. Setiap penambahan jumlah data dua kali lipat dari data sebelumnya menunjukkan selisih waktunya (lihat Tabel 2).



Gambar 6 Grafik Hasil Pengujian di *Parallel Proccessing*

Tabel 3 Selisih Waktu Komputasi Percobaan di *Parallel Proccessing*

Slave 2	n2000	n4000	Selisih waktu	n4000	n6000	Selisih waktu
	0.379	1,395	1,016	1,395	3,097	1,702
	n6000	n8000	Selisih waktu	n8000	n10000	Selisih waktu
	3,097	5,489	2,392	5,489	8,555	3,066
Slave 3	n2000	n4000	Selisih waktu	n4000	n6000	Selisih waktu
	0,291	0,999	0,708	0,999	2,104	1,105
	n6000	n8000	Selisih waktu	n8000	n10000	Selisih waktu
	2,104	3,69	1,586	3,69	5,738	2,048
Slave 4	n2000	n4000	Selisih waktu	n4000	n6000	Selisih waktu
	0,278	0,782	0,054	0,782	1,865	1,083
	n6000	n8000	Selisih waktu	n8000	n10000	Selisih waktu
	1,865	2,919	1,054	2,919	4,513	1,594

Pada Tabel 2 terlihat bahwa semakin banyak data yang diproses semakin banyak waktu yang diperlukan untuk melakukan perhitungan LPF. Semakin banyak *node* yang terlibat maka semakin kecil waktu yang dibutuhkan untuk melakukan perhitungan LPF. Konsep paralel membuat waktu

komputasi menjadi lebih kecil dibandingkan diproses secara *single*, selain itu banyaknya *slave* mempengaruhi hasil akhir pada waktu komputasi LPF yaitu semakin banyak node/*slave* akan semakin kecil waktu yang diperoleh.

4. Kesimpulan

Berdasarkan percobaan-percobaan yang dilakukan dapat disimpulkan bahwa kecepatan pemrosesan suatu sistem paralel dipengaruhi oleh jumlah node/proses dan jumlah komponen frekuensi yang diproses. Pada komputer tunggal atau *single* semakin besar proses maka waktu yang dibutuhkan semakin banyak dan nilai prop hanya berpengaruh pada banyaknya data frekuensi tinggi yang terfilter pada domain. Sedangkan pada komputer paralel semakin banyak komputer yang terlibat untuk memproses perhitungan LPF maka semakin kecil waktu yang dibutuhkan untuk melakukan perhitungan. Begitu pula dengan bertambahnya jumlah komponen frekuensi yang dihitung (nilai *prop* kecil dan data besar) serta sedikit *slave*/komputer yang terlibat mengakibatkan waktu yang diperlukan semakin besar karena proses yang dihitung banyak dengan pembagian kerja node yang terlibat sedikit. Dari keseluruhan hasil percobaan dan analisanya dapat disimpulkan bahwa konsep *paralelisme* pada perhitungan LPF dengan *transform cosinus* memberikan hasil efektif dan efisien pada proses pengerjaan.

5. Saran

Adapun saran untuk penelitian selanjutnya, sebaiknya program ini dikembangkan untuk penerapan LPF pada proses pengolahan citra untuk pemrosesan yang lebih cepat sehingga dapat memberikan hasil yang efektif dan efisien dalam proses perhitungan setiap piksel di dalam citra yang akan diolah dengan LPF dan untuk proses pengujiannya dikembangkan dengan skalabilitas data dan jaringan komputer yang lebih besar lagi agar semakin kecil waktu yang digunakan untuk pemrosesannya sehingga lebih efektif dan efisien.

DAFTAR PUSTAKA

[1] Gebali, Fayez, 2011, *Algorithms and Parallel Computing*, Vol. 1, Ed.1, John Wiley & Sons Inc., Canada.

[2] Gonzales, R., P., 2004, *Digital Image Processing* (*Pemrosesan Citra Digital*), Vol. 1, Ed.2, diterjemahkan oleh Handayani, S., Andri Offset, Yogyakarta.

[3]Hanafi, 2013, *Simulasi Hasil Perancangan LPF (Low Pass Filtering) Digital Menggunakan Prototip Filter Analog Butterworth*, Litek Journal, Vol.10 No.1.

[4]Martins, Simone de L., Riberio, Celso C., dan Rodriguez, Noemi., 2011, *Parallel Computing Environments*, National Research Network, Estrada Dona Castorina.

[5]Matloff, Norm., 2011, Programming on Parallel Machines ,http://heather.cs.ucdavis.edu/~matloff/ParProcBook.pdf, diunduh pada tanggal 23 Agustus 2016.