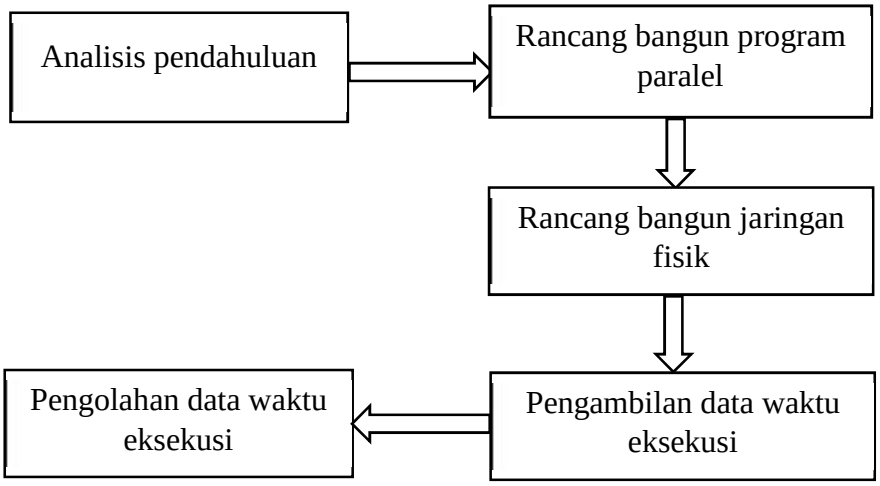
BAB III

METODE PENELITIAN

Pada penelitian ini, dilakukan analisis pendahuluan dari waktu eksekusi algoritma *Shell sort* dan *Quick sort* untuk mengetahui titik perpotongan data antara kedua algoritma. Setelah titik perpotongan data didapatkan, selanjutnya adalah mengkombinasikan (*hybrid)* algoritma *Shell sort* dan *Quick sort* yang ditulis ke dalam bentuk program serial, kemudian mengimplementasikan pemrograman paralel berbasis *massage passing interface* pada rutin komunikasi *point to point*. Metode yang digunakan pada penelitian ini adalah metode eksperimen, dimana program dieksekusi pada jaringan komputer fisik dengan menggunakan dua komputer yaitu satu komputer sebagai *master* dan satu komputer lagi sebagai *slave.* Adapun tahap-tahap yang dilakukan dalam metode penelitian ini seperti terlihat pada Gambar 3.1.



Gambar 3.1. Diagram blok tahapan penelitian

3.1 Analisis Pendahuluan

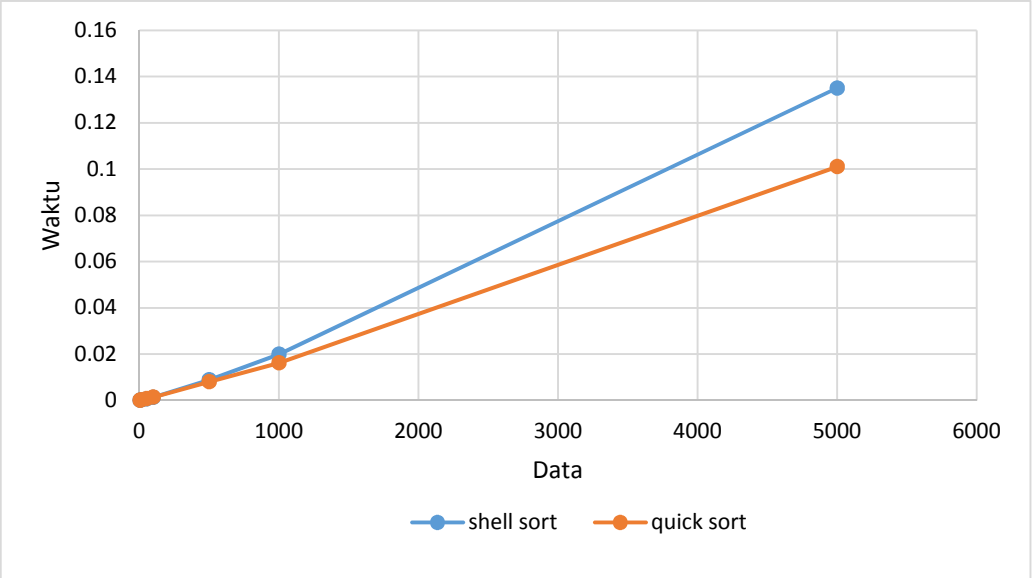
Analisis pendahuluan dilakukan agar dapat diketahui titik perpotongan data antara kedua algoritma yang digunakan pada tahap perancangan algoritma *Shell sort* dan *Quisk sort* kombinasi (*hybrid).* Adapun data yang dianalisis dari algortima *Shell sort* dan *Quick sort* adalah data waktu eksekusi *average case* berdasarkan hasil praktik. Kondisi *average case* dipilih karena tidak diketahui kondisi data acak sebelumnya apakah dalam keadaan terurut, terurut terbalik, berukuran kecil ataupun berukuran besar.

Pada waktu eksekusi *average case*, analisis dilakukan dengan cara mencari titik potong data antara algortima *Shell sort* dan *Quick sort* yang ditampilkan menggunakan grafik yang terdapat pada *Microsoft Excel*. Data waktu eksekusi algoritma *Shell sort* dan *Quick sort* dari penelitian sebelumnya pada 5-5000 data acak seperti terlihat pada Tabel 3.1.

Tabel 3.1. Waktu Eksekusi Algoritma *Shell sort* dan *Quick sort* (Ocampo, 2008)

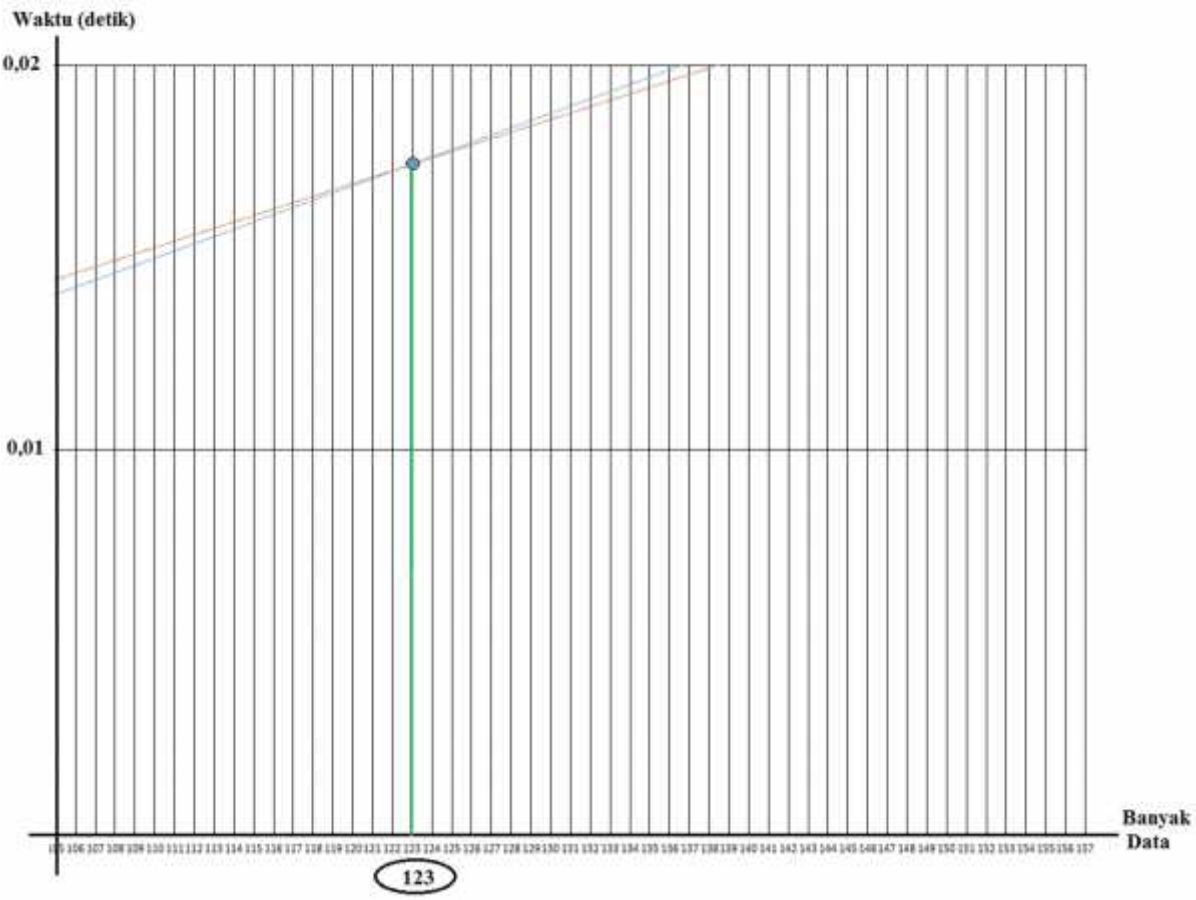
Algoritma	**Banyak data**						
	5	**10**	**50**	**100**	**500**	**1000**	**5000**
Shell sort	0,060 detik	0,117 detik	0,635 detik	1,311 detik	8,845 detik	19,894 detik	134,810 detik
Quick sort	0,067 detik	0,128 detik	0,656 detik	1,364 detik	8,035 detik	16,198 detik	100,949 detik

Dari Tabel 3.1 terlihat bahwa pada data 5-100, algoritma *Shell sort* lebih cepat daripada algoritma *Quick sort.* Pada data 500-5000, algoritma *Shell sort* menjadi lebih lambat, dan terlihat bahwa algoritma *Quick sort* lebih cepat daripada algortima *Shell sort.* Agar dapat terlihat titik potong, maka data waktu eksekusi tersebut ditampilkan pada sebuah grafik seperti terlihat pada Gambar 3.2.



Gambar 3.2. Grafik perpotongan data (Skala 1 : 1000)

Pada Gambar 3.2 titik perpotongan data tidak bisa terlihat dengan jelas karena ukuran grafik yang kecil. Agar bisa terlihat, pembesaran grafik dari Gambar 3.2 seperti terlihat pada Gambar 3.3.



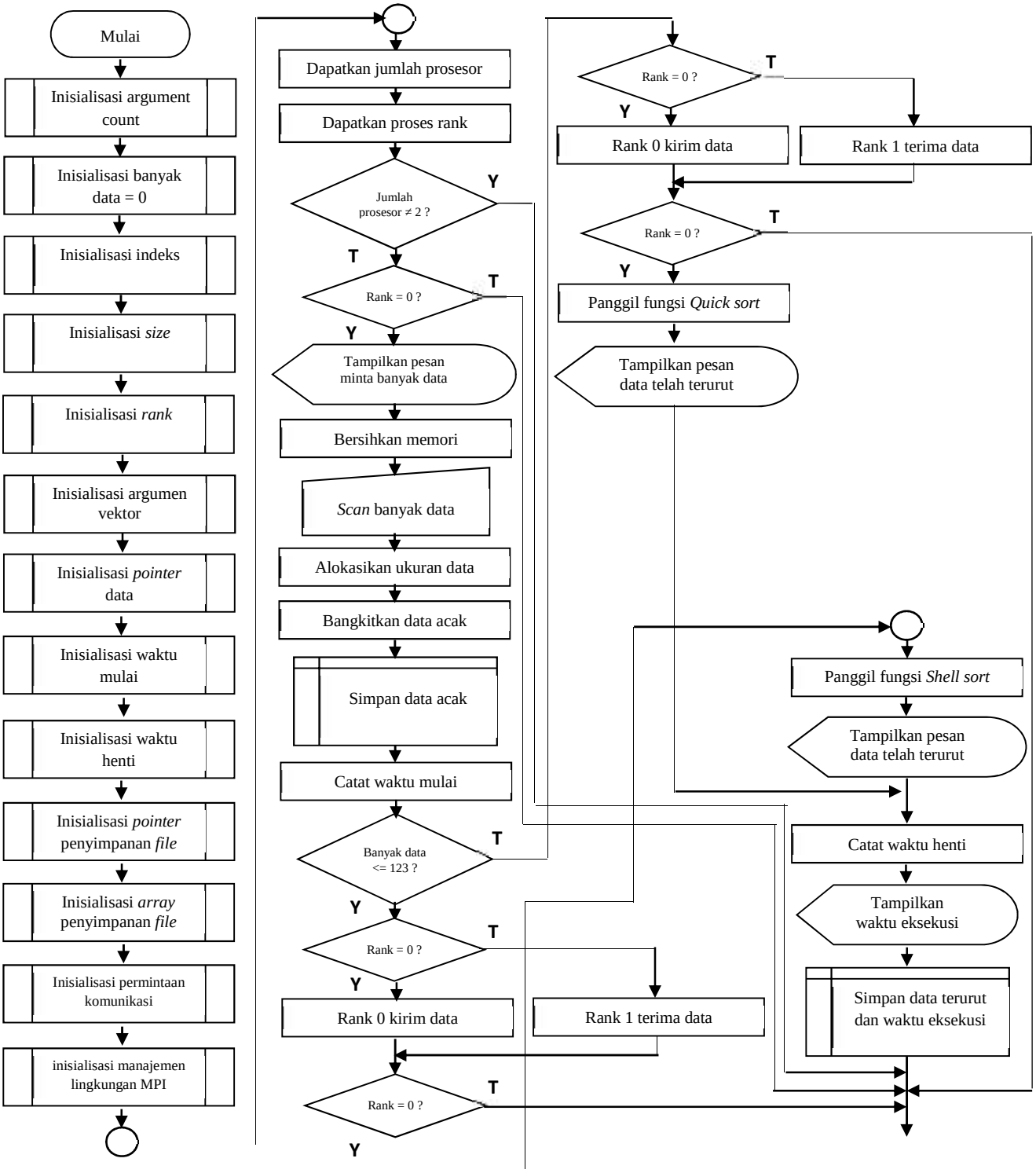
Gambar 3.3. Pembesaran grafik dari Gambar 3.2 (Skala 1 : 1)

Dari Gambar 3.3 terlihat titik perpotongan terdapat pada data yang ke-123. Pada data kurang dari 123, algoritma *Shell sort* lebih cepat daripada algoritma *Quick sort,* sebaliknya pada saat data lebih dari 123 algoritma *Shell sort* menjadi lebih lambat daripada algoritma *Quick sort.*

Dari hasil analisis pendahuluan di atas, nilai 123 digunakan untuk menentukan batas data yang akan diurutkan oleh algoritma *Shell sort* dan *Quick sort* yang akan di kombinasikan dan dirancang terlebih dahulu dengan *flowchart* (diagram alir) kemudian program ditulis ke dalam bahasa pemrograman C. Pada saat data kurang dari atau sama dengan 123, maka data akan diurutkan menggunakan algoritma *Shell sort.* Pada saat data lebih dari 123, maka data akan diurutkan menggunakan algoritma *Quick sort.*

3.2 Rancang Bangun Program Paralel

Program paralel *hybrid* algoritma *Shell sort* dan *Quick sort* dirancang menggunakan *flowcart* (diagram alir). *Flowchart* sangatlah penting sebagai tahap awal dari perancangan sebuah algoritma, karena *flowchart* menggambarkan langkah-langkah program yang akan dibuat. Setelah *flowchart* dirancang, selanjutnya adalah penulisan kode program paralel ke dalam bahasa C berbasis *massage passing interface* yang menggunakan rutin komunikasi *point to pont*.



Gambar 3.4. *Flowchart* algoritma *Shell sort* dan *Quick sort hybrid* paralel

Berdasarkan Gambar 3.4 di atas, maka dapat diketahui dan ditulis subrutin-subrutin program yang perlu digunakan, serta urutan atau langkah-langkah dalam perancangan program *hybrid* algoritma *Shell sort* dan *Quick sort* paralel.

Subrutin kode program inisialisasi yaitu:

```
….
int main ()
….
int argc, n=0, i, size, rank;
char **argv;
float *data;
double start, end;
FILE *ifp;
char inputFilename[] = "Data Acak.doc";
FILE *ofp;
char outputFilename[] = "Data Terurut.doc";
MPI_Request request;
MPI_Init(&argc, &argv);
….
```

Subrutin program untuk mendapatkan jumlah prosesor, proses *rank* dan banyaknya prosesor yang digunakan yaitu:

```
….
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (size != 2)
{
      printf("Maaf, jumlah pemroses/prosesor harus lebih dari 2 !!\n");
      MPI_Finalize();
      return 0;
}
….
```

Subrutin program untuk menampilkan pesan meminta banyak data acak, membersihkan memori dan men-*scan* banyak data yang ingin diacak yaitu:

```
….
printf("Masukkan banyak data yang ingin diacak : ");
fflush (stdout);
scanf("%i", &n);
….
```

Subrutin program untuk alokasikan ukuran data acak yaitu:

```
….
data=(float*)malloc(n*sizeof(int));
….
```

Subrutin program untuk membangkitkan banyaknya data acak yang diminta lalu menyimpannya ke dalam *file* yaitu:

```
….
for (i=0; i<n; i++)
{
      data[i]= (float) rand() * (float) 0.1;
      fprintf(ifp, "%f\n", data[i]);
}
fclose(ifp)
….
```

Data acak yang dibangkitkan adalah bilangan *floating point* yang dikalikan dengan nilai 0,1 sebagai konstanta untuk menghasilkan bilangan acak sebanyak enam angka di belakang koma, sehingga bilangan *floating point* tersebut tidak hanya menghasilkan bilangan nol saja di belakang koma. Hasil data acak tersebut disimpan ke dalam sebuah *file* dokumen bernama “Data Acak.doc”.

Setelah itu, dicatat waktu mulai eksekusi, adapun subrutin programnya yaitu:

```
….
start=MPI_Wtime ();
….
```

Kemudian program akan memilih fungsi yang akan digunakan dalam melakukan pengurutan data. Pada saat banyak data acak (n) kurang dari atau sama dengan 123, maka pengurutan data dilakukan dengan algoritma *Shell sort,* sedangkan pada saat banyak data acak (n) lebih dari 123, maka pengurutan data dilakukan menggunakan algoritma *Quick sort.* Adapun subrutin program untuk memilih fungsi yang akan digunakan yaitu:

```
….
if (n<=123)
{
….  /*algoritma Shell sort*/
}
else
{
…. /*algoritma Quick sort
}
….
```

Proses komunikasi paralel terjadi jika *rank* 0 mengirim data ke *rank* 1, dan *rank* 1 menerima data dari *rank* 0. Subrutin programnya yaitu:

```
….
if (rank==0)
{
        MPI_Isend(data, n, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &request);
}
else if (rank==1)
{
        MPI_Irecv(data, n, MPI_FLOAT, 1, 1, MPI_COMM_WORLD, &request);
}
….
```

Dari proses di atas, *rank* 0 melakukan pengiriman data dan *rank* 1 melakukan penerimaan data. Kedua *rank* tersebut masing-masing menginisialisasi *buf* yang dikirim dan diterima yaitu data, kemudian menghitung (*count*) banyaknya elemen data yang akan dikirim dan diterima yaitu n. MPI_FLOAT adalah tipe data yang dikirim dan diterima yaitu berupa bilangan *floating point.* Nilai *dest* pengirim adalah 0 (*rank* 0), sedangkan nilai *dest* penerima adalah 1 (*rank* 1). Nilai *message tag* pengiriman dan penerimaan adalah 1. Komunikator yang digunakan adalah MPI_COMM_WORLD, dan alamat memori permintaan komunikasi yang digunakan adalah *&request*.

Pada saat banyak data kurang dari atau sama dengan 123, maka program memanggil fungsi *Shell sort* kemudian mengurutkan data tersebut dengan menggunakan algoritma *Shell sort*. Cara pemanggilan fungsi *Shell sort* yaitu:

```
….
if (rank==0)
{
      ShellSort(data,n);
….
```

Dengan algoritma *Shell sort* yaitu:

```
….
void ShellSort(float data[], int n)
{
     int jarak, i, j;
     float simpan;
     for(jarak=n/2; jarak>0; jarak/=2)
     {
          for(i=jarak; i<n; i++)
          {
               simpan=data[i];
               for(j=i; j>=jarak; j-=jarak)
               {
                    if(simpan<data[j-jarak])
                    {
                         data[j]=data[j-jarak];
                    }
                    else
                    {
                         break;
                    }
               }
               data[j]=simpan;
          }
     }
}
….
```

Jika data berhasil diurutkan menggunakan algoritma *Shell sort*, maka program menampilkan pesan bahwa algoritma tersebut berhasil mengurutkan data menggunakan algoritma *Shell sort*. Adapun subrutin programnya yaitu:

```
….
printf("\nData telah berhasil diurut menggunakaan program Shell sort. \n");
….
```

Pada saat banyak data lebih dari 123, maka program memanggil fungsi *Quick sort* kemudian mengurutkan data tersebut dengan menggunakan algoritma *Quick sort*. Cara pemanggilan fungsi *Quick sort* yaitu:

```
….
if (rank==0)
{
      QuickSort(data, 0, n-1);
….
```

Dengan algoritma *Quick sort* yaitu:

```
….
int Partition(float data[], int awal, int akhir)
{
        int i=awal, j;
        float pivot=data[awal];
        for(j=awal+1; j<=akhir; j++)
        {
                if(pivot>data[j])
                {
                        data[i]=data[j];
                        data[j]=data[i+1];
                        data[i+1]=pivot;
                        i=i+1;
                }
        }
        return i;
}
void QuickSort(float data[], int awal, int akhir)
{
        if(awal<akhir)
        {
                int p=Partition(data, awal, akhir);
                QuickSort(data, awal, p-1);
                QuickSort(data, p+1, akhir);
        }
}
….
```

Pada algoritma *Quick sort* di atas terdapat dua fungsi, yaitu fungsi pembagi dan fungsi pengurutan data. Sebelum dilakukan pengurutan, data terlebih dahulu dibagi menjadi beberapa bagian menggunakan fungsi Partition(), kemudian setiap bagian dilakukan pengurutan menggunakan fungsi QuickSort(). Jika data berhasil diurutkan, maka program menampilkan pesan bahwa algoritma *Quick sort* berhasil mengurutkan data. Subrutin program yang digunakan yaitu:

```
….
printf("\nData telah berhasil diurut menggunakaan program Quick sort. \n");
….
```

Selanjutnya dicatat waktu berhentinya proses pengurutan data. Subrutin program untuk mencatat waktu henti yaitu:

```
….
end=MPI_Wtime ();
….
```

Setelah algoritma *Shell sort* atau *Quick sort* berhasil mengurutkan data, maka dihitung waktu eksekusinya, lalu disimpan hasil waktu eksekusi dan data terurut tersebut ke dalam suatu *file* bernama “Data Terurut.doc”. Subrutin program untuk menghitung waktu eksekusi dan menyimpannya ke dalam *file* data terurut yaitu:

```
….
ofp = fopen(outputFilename, "w");
fprintf(ofp,"Hasil pengurutan menggunakaan program Shell sort : \n\n");
for(i = 0 ; i < n ; i++)
{
        fprintf(ofp,"%f\n", data[i]);
}
end=MPI_Wtime();
printf("\nWaktu eksekusi = %lf Detik\n", end-start);
fprintf(ofp,"\nWaktu eksekusi = %lf Detik\n", end-start);
fclose(ofp);
….
```

Selanjutnya diakhiri proses pemrograman paralel. Adapun subrutin programnya yaitu:

```
….
MPI_Finalize ();
return 0;
….
```

3.3 Rancang Bangun Jaringan Fisik

Pada penelitian ini, agar kode program *hybrid* algoritma *Shell sort* dan *Quick sort* bisa dieksekusi di dalam jaringan komputer fisik, maka perangkat lunak yang digunakan adalah *Virtualbox* versi 4.1.4 yang di-*install* di dalam sistem operasi *Windows* 8.1 *Pro with Media Center*. *Virtualbox* digunakan untuk membangun sebuah jaringan komputer v*irtual.*

Jaringan komputer yang dibangun pada penelitan ini menggunakan topologi jaringan komputer *point to point* yang terdiri dari dua komputer, yaitu satu komputer bertindak sebagai *master* dan satu komputer lagi bertindak sebagai *slave* dengan menggunakan sistem operasi *Linux PelicanHPC* versi 2.2 (32 *bit*).

Gambar 3.5. Topologi jaringan komputer *point to point*

Dari Gambar 3.5, komputer *master* dan *slave* dihubungkan di dalam jaringan secara langsung menggunakan kabel *Local Area Network* (LAN) tipe *Cross over* dengan topologi jaringan komputer *point to point*. Rancang bangun jaringan komputer fisik ini dilakukan agar kedua komputer tersebut dapat melakukan pengiriman dan penerimaan data pada program yang dijalankan atau dieksekusi.

3.3.1 Komputer *Master*

Komputer *master* adalah komputer yang terhubung di dalam jaringan komputer, yang bertindak memberikan *task-task* atau perintah kepada komputer *slave* sebagai pekerja pada saat program dijalankan atau diksekusi di dalam jaringan komputer. Adapun spesifikasi komputer *master* seperti terlihat pada Tabel 3.2.

Tabel 3.2. Spesifikasi komputer *master*

Perangkat	Spesifikasi
Prosesor	CPU *Intel Core* 2 *Duo* @2.0 GHz
RAM (*Random Acces Memory*)	512 *megabyte*
Hardisk	320 *gigabyte*
Sistem Operasi	*Linux Peli*can*HPC* versi 2.2 (32 *bit*)

3.3.2 Komputer *Slave*

Komputer *slave* adalah komputer yang terhubung di dalam jaringan komputer, yang bertindak sebagai pekerja yang menerima *task-task* atau perintah dari komputer *master* pada saat program dijalankan atau diksekusi di dalam jaringan komputer. Adapun spesifikasi komputer *slave* terlihat pada Tabel 3.3.

Tabel 3.3. Spesifikasi komputer *slave*

Perangkat	Spesifikasi
Prosesor	CPU *Intel Core* 2 *Duo* @2.0 GHz
RAM (*Random Acces Memory*)	384 *megabyte*
Hardisk	320 *gigabyte*
Sistem Operasi	*Linux PelicanHPC* versi 2.2 (32 *bit*)

3.4 Pengambilan Data Waktu Eksekusi

Untuk pengambilan data waktu eksekusi dari beberapa sampel data acak yang dilakukan pengurutan, program *hybrid* serial (Lampiran B) dan *hybrid* paralel (Lampiran

C) dari algoritma *Shell sort* dan *Quick sort* dijalankan di dalam jaringan komputer fisik pada komputer *master* yang telah terhubung dengan komputer *slave.* Kode program tersebut di-*compile* untuk mendapatkan *output file* kemudian dieksekusi pada komputer *master*, sehingga didapatkan hasil waktu eksekusi dan data terurut. Adapun perintah *compile* dan *execution* program *hybrid* serial dan *hybrid* paralel dari algoritma *Shell sort* dan *Quick sort* seperti terlihat pada Tabel 3.2.

Table 3.4. Perintah *compile* dan *execution* program serial dan paralel dari algoritma *Shell sort* dan *Quick sort*

Program	***Compile***	***execution***
Serial	*gcc filename.c –o outputfilename*	*./outputfilename*
Paralel	*mpicc filename.c –o outputfilename*	*mpiexec –n 2 outputfilename*

Kode program *hybrid* serial algoritma *Shell sort* dan *Quick sort* di-*compile* pada terminal *Linux* dengan *compiler* GCC (*GNU C Compiler*) versi 4.3.2. Pada kode program *hybrid* paralel algoritma *Shell sort* dan *Quick sort* juga di-*compile* pada terminal *Linux* dengan menggunakan *compiler* MPI untuk bahasa C (*mpicc*) versi 1.4.2.

Banyaknya data yang diambil untuk pengambilan waktu eksekusi adalah kurang dari atau sama dengan 123, dan lebih dari 123. Data tersebut adalah 5, 10, 50, 100, 500, 1000 dan 5000 data acak agar diketahui waktu eksekusi program serial dan program paralel dari *hybrid* algoritma *Shell sort* dan *Quick sort*.

3.5 Pengolahan Data Waktu Eksekusi

Pengolahan data waktu eksekusi diambil dari waktu rata-rata 10 kali percobaan eksekusi program *hybrid* serial dan *hybrid* paralel algoritma *Shell sort* dan *Quick sort*. Dari hasil rata-rata waktu eksekusi tersebut, data kemudian dianalisis dengan cara merasiokan rata-rata waktu eksekusi program serial dengan program paralel agar diketahui kecepatan pemrosesan (*speed up*) yang dicapai oleh program *hybrid* paralel dari program *hybrid* serial algoritma *Shell sort* dan *Quick sort*.

Setelah nilai s*peed up* didapatkan, dilakukan perhitungan nilai efisiensi untuk mengevaluasi seberapa signifikan penggunaan dua komputer. Efisiensi merupakan indikator atas tingkat kinerja *speed up* yang dicapai dibandingkan dengan nilai maksimum yang dapat dicapai.