Perbandingan Algoritma *Merge Sort* dengan menggunakan *Message Passing Interface* dan *Sequential*

Bagaspranawa Tirta Buana, Fajar Sakti Sanubari, Irsyad Abqori Fikri, Septian Luthfia Sanni

Ilmu Komputer/ Informatika, Universitas Diponegoro

Semarang, Indonesia

e-mail: bagaspranawa@students.undip.ac.id, fajarsakti@students.undip.ac.id, aleirsyad@students.undip.ac.id, septianluthfias@students.undip.ac.id

ABSTRAK

Penggunaan aplikasi paralel berbasis MPI adalah salah satu cara dalam rangka memenuhi kebutuhan komputasi yang terus meningkat. Lebih dari satu sumber daya dapat dimanfaatkan secara simultan dalam rangka melakukan komputasi. Pada sisi yang lain, Grid Computing hadir dengan menawarkan sumber daya komputasi besar. Lebih dari satu sumber daya yang tersedia, walaupun heterogen dan berada dalam lokasi yang tersebar, dapat dimanfaatkan bersama-sama dalam rangka menyediakan sumber daya komputasi yang besar. Penelitian ini dilakukan dalam rangka mencari tahu kecepatan eksekusi dengan menggunakan MPI pada urutan bilangan dengan algoritma *Merge Sort.* MPI dapat menjadi solusi dalam hal mengeksekusi *high-performance computing.*

Katakunci: Komputasi Tersebar dan Paralel, *Message Passing Interface*, *Merge Sort*

ABSTRACT

The use of MPI-based parallel applications is one way to meet the growing computing needs. More than one resource can be used simultaneously in order to perform computations. On the other hand, Grid Computing comes with offering large computing resources. More than one available resource, although heterogeneous and located in scattered locations, can be used together to provide large computing resources. This research was conducted in order to find out the speed of execution by using MPI in a sequence of numbers with the Merge Sort algorithm. MPI can be a solution in terms of executing high-performance computing.

Keywords: *Distributed Parallel Computing, Message Passing Interface, Merge Sort*

1. PENDAHULUAN

Teknologi yang semakin maju memicu beragam paradigma komputasi untuk terus berkembang. Paradigma komputasi diawali dengan proses komputasi menggunakan prosesor tunggal. Untuk meningkatkan kecepatan waktu komputasi, penggunaan multiprocessor bagi eksekusi proses komputasi mulai diperkenalkan. Paradigma yang kedua ini lebih dikenal sebagai *parallel computing* (komputasi paralel). Saat ini muncul kembali paradigma ketiga yang dikenal dengan *grid computing* (komputasi tersebar).

Penggunaan pendekatan multi-prosesor bagi eksekusi proses komputasi telah terbukti memberikan peningkatan

kecepatan proses eksekusi (jika dibandingkan dengan eksekusi proses komputasi menggunakan prosesor tunggal). Jelas bahwa pendekatan *parallel computing* telah memberikan peningkatan kecepatan waktu komputasi, yang dalam hal ini sering direpresentasikan dengan nilai *speedup*.

Pada penelitian kali ini akan dikaji suatu studi kasus berupa komputasi paralel untuk pengurutan bilangan menggunakan algoritma Merge Sort dengan menggunakan *Message Passing Interface*. Pengkajian dilakukan melalui eksperimen pengukuran running time kasus uji dan analisis hasil eksperimen yang bertujuan untuk mengetahui apakah MPI berpengaruh secara signifikan terhadap kecepatan prosesor dan kartu grafis dalam mengurutkan bilangan menggunakan algoritma *Merge Sort*.

2. DASAR TEORI

2.1 Pengurutan

Algoritma pengurutan adalah proses menyusun kembali rentetan objek-objek untuk meletakkan objek dari suatu kumpulan data ke dalam urutan yang logis [1]. Pada dasarnya, pengurutan (sorting) membandingkan antara data atau elemen berdasarkan kriteria dan kondisi tertentu [2]. Pengurutan dapat dilakukan dari nilai terkecil ke nilai terbesar (ascending) atau sebaliknya (descending). Ada dua kategori pengurutan [3]:

1. Pengurutan internal

   Pengurutan internal adalah pengurutan yang dilaksanakan hanya dengan menggunakan memori komputer, pada umumnya digunakan bila jumlah elemen tidak terlalu banyak.
2. Pengurutan eksternal

   Pengurutan eksternal adalah pengurutan yang dilaksanakan dengan bantuan memori virtual atau harddisk karena jumlah elemen yang akan diurutkan terlalu banyak.

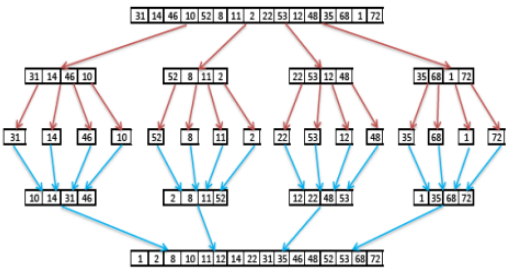
Algoritma-algoritma pengurutan dibedakan berdasarkan [4]:

1. Kompleksitas perbandingan antar elemen terkait dengan kasus terbaik, rata-rata dan terburuk.
2. Kompleksitas pertukaran elemen, terkait dengan cara yang digunakan elemen setelah dibandingkan
3. Penggunaan memori
4. Rekursif atau tidak rekursif
5. Proses pengurutannya (metode penggunaannya)

2.2 Algoritma Merge Sort

Algoritma merge sort merupakan algoritma yang dicetuskan oleh John von Neumann pada tahun 1945 [5]. Merge sort menggunakan prinsip divide and conquer. Divide and conquer adalah metode pemecahan masalah yang bekerja dengan membagi masalah (problem) menjadi beberapa sub-masalah (subproblem) yang lebih kecil, kemudian menyelesaikan masing-masing sub-masalah secara independen dan akhirnya menggabung solusi masing-masing sub-masalah sehingga menjadi solusi masalah semula [6].

Pada umumnya, merge sort membagi data menjadi 2 bagian. Namun pada algoritma 4 way merge sort membagi data menjadi 4 bagian sehingga diperoleh sub data yang terpisah. Kemudian sub data tersebut diurutkan secara terpisah lalu menggabungkannya sehingga diperoleh data dalam keadaan terurut. Proses pengurutan menggunakan 4 way merge sort pada gambar di bawah ini.



Gambar 1. Merge Sort

2.3 MPI (Message Passing Interface)

Message-Passing Interface atau MPI adalah sebuah standar untuk interface pada pemrograman paralel. Ada lima karakter dasar pada model pemrograman message passing, yang dapat diimplementasikan pada MPI, sebagai berikut:

1. Sebuah komputasi terdiri dari sekumpulan proses-proses besar (biasanya dalam jumlah yang ditentukan), dimana masing-masing memiliki identifikasi yang unik (bilangan bulat 0 sampai P-1 (Jumlah Prosesor-1)).
2. Proses-proses saling berinteraksi dengan saling bertukar pesan-pesan tertentu, dengan melibatkan diri pada operasi komunikasi kolektif maupun menunggu pesan-pesan yang dikirimkan.
3. Modularitas didukung melalui kelompok jaringan (communicator), yang membuat sub-program dapat mengkapsulasi operasi-operasi komunikasi dan dapat dikombinasikan dengan komposisi sekuensial maupun paralel.
4. Algoritma yang membuat pekerjaan-pekerjaan dinamis atau beberapa pekerjaan sekaligus pada sebuah prosesor, memerlukan perbaikan

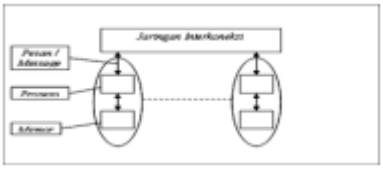
sebelum dapat diimplementasikan dengan MPI.

5. Sifat deterministik (beban yang sama) tidak terjamin, tetapi dapat diperoleh dengan pemrograman yang cermat.

Terdapat beberapa fungsi dalam MPI, termasuk fungsi inisialisasi dan fungsi dasar *send and receive.* Fungsi-fungsi tersebut ialah : MPI_Init, MPI_Send, MPI_Recv, MPI_Comm_size, serta fungsi-fungsi lain dengan tujuannya masing-masing.

2.3.1 Pertukaran Pesan

Message-passing dilakukan oleh prosesor-prosesor yang terlibat dalam komputasi paralel untuk melakukan pertukaran data dan sinkronisasi antar prosesor. Pertukaran data pada message-passing dibuat dengan menghubungkan sekumpulan komputer melalui sebuah interkoneksi jaringan. Setiap komputer memiliki sebuah prosesor dan memori lokal dan komunikasi antar komputer dilakukan melalui interkoneksi jaringan. Pesan tersebut berisikan data yang diperlukan dalam komputasi.



Gambar 2. Model komputer Message Passing

2.3.2 Komunikasi Pada Message-Passing

Pada sebuah proses komunikasi, ada dua aktivitas yang dilakukan, yaitu pengiriman dan penerimaan, masing-masing proses memiliki identifikasi yang unik. Identifikasi ini digunakan seperti penggunaan alamat dalam pengiriman surat. Sebuah pesan yang sederhana pada model pemrograman message-passing memiliki alamat penerima (identifikasi proses tujuan) serta alamat pengirim (identifikasi proses asal). Komunikasi sederhana dalam MPI yang hanya melibatkan dua buah proses. Komunikasi yang melibatkan lebih dari dua proses disebut komunikasi kolektif. Komunikasi kolektif ini melibatkan semua proses yang berada pada suatu kelompok jaringan (communicator) yang sama. Operasi pada komunikasi kolektif disertai

dengan proses sinkronisasi. Ini artinya setiap kali proses-proses melakukan operasi komunikasi kolektif, proses-proses tersebut tidak dapat melakukan pekerjaan selanjutnya sebelum semua proses selesai melakukan operasi tersebut. Jika sebuah proses belum selesai maka proses yang lain akan menunggu proses tersebut hingga selesai.

2.4 Sequential

Sequential adalah Pencarian berurutan sering disebut pencarian linear merupakan metode pencarian yang paling sederhana. Pencarian berurutan menggunakan prinsip sebagai berikut: data yang ada dibandingkan satu persatu secara berurutan dengan yang dicari sampai data tersebut ditemukan atau tidak ditemukan.

Pada dasarnya, pencarian ini hanya melakukan pengulangan dari 1 sampai dengan jumlah data. Pada setiap pengulangan, dibandingkan data ke-i dengan yang dicari. Apabila sama, berarti data telah ditemukan. Sebaliknya apabila sampai akhir pengulangan tidak ada data yang sama, berarti data tidak ada. Pada kasus yang paling buruk, untuk N elemen data harus dilakukan pencarian sebanyak N kali pula

3. METODOLOGI

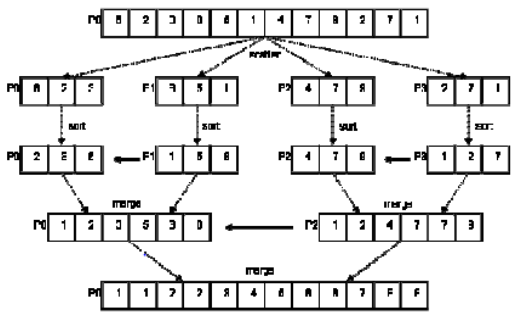
3.1 Implementasi *Merge Sort* menggunakan MPI

Disini kami melakukan *sorting* 1000 - 100000000 bilangan dengan menggunakan *merge sort*. Pada *sorting* ini, digunakan beberapa fungsi pada MPI, yaitu :

1. MPI_Send : untuk mengirim data
2. MPI_Recv : untuk menerima data
3. MPI_Finalize : untuk menghentikan eksekusi MPI pada *communicator*
4. MPI_Comm_size : untuk mendapatkan jumlah proses pada *communicator*
5. MPI_Comm_rank : untuk mendapatkan rank dari tiap proses
6. MPI_Init : untuk menginisialisasi program MPI
7. MPI_Scatter : untuk mendistribusikan data ke tiap proses
8. MPI_Wtime : untuk mencatat waktu eksekusi

Kami juga melakukan perbandingan antara MPI *merge sort* dengan *sequential merge sort*. Perbandingan ini dilakukan dengan membandingkan waktu yang dibutuhkan untuk melakukan *merge sort* dari kedua cara tersebut.

Berikut gambar ilustrasi mengenai proses *sorting* menggunakan *merge sort* :



Gambar 3. Mapping Task *Merge Sort*

Berikut merupakan kode program lengkap untuk *merge sort* menggunakan MPI :

```
/*
 * mergeSort.c
 * ...illustrates parallel merge
sort in MPI.
 *
 * Hannah Sonsalla, Macalester
College 2017
 *
 * Usage: mpirun -np N ./mergeSort
<arraySize>
 *  - arraySize must be a multiple
of N
 *  - N must be positive and a
power of 2
 *
 * Notes:
 *  - To view initial unsorted
array uncomment line A
 *  - To view local arrays of
processes before sorting uncomment
line B
 *  - To view final sorted array
uncomment line C
 *
 */

#include <mpi.h>     // MPI
#include <stdio.h>   // printf
#include <stdlib.h>  // malloc,
free, rand(), srand()
#include <time.h>    // time for
random generator
#include <math.h>    // log2
#include <string.h>  // memcpy
#include <limits.h>  // INT_MAX

 /* Declaration of functions */
void powerOfTwo(int id, int
numberProcesses);
void getInput(int argc, char*
argv[], int id, int numProcs, int*
arraySize);
void fillArray(int array[], int
arraySize, int id);
void printList(int id, char
arrayName[], int array[], int
arraySize);
int compare(const void* a_p, const
void* b_p);
int* merge(int half1[], int
half2[], int mergeResult[], int
size);
int* mergeSort(int height, int id,
int localArray[], int size,
MPI_Comm comm, int globalArray[]);

/*--------------------------------
--------------------------------
-
 * Function:   powerOfTwo
 * Purpose:    Check number of
processes, if not power of 2
prints message
 * Params:     id, rank of the
current process
 *
numberProcesses, number of
processes
 */

void powerOfTwo(int id, int
numberProcesses) {
    int power;
    power = (numberProcesses != 0)
&& ((numberProcesses &
(numberProcesses - 1)) == 0);
    if (!power) {
        if (id == 0)
printf("number of processes must
be power of 2 \n");
        MPI_Finalize();
        exit(-1);
    }
}

/*--------------------------------
--------------------------------
-
 * Function:   getInput
 * Purpose:    Get input from user
for array size
 * Params:     argc, argument
count
 *                   argv[],
points to argument vector
 *                   id, rank of
the current process
 *                   numProcs,
number of processes
 *                   arraySize,
points to array size
 */

void getInput(int argc, char*
argv[], int id, int numProcs, int*
arraySize) {
    if (id == 0) {
        if (id % 2 != 0) {
            fprintf(stderr,
"usage: mpirun -n <p> %s <size of
array> \n", argv[0]);
            fflush(stderr);
            *arraySize = -1;
```

```
        }
        else if (argc != 2) {
            fprintf(stderr,
"usage: mpirun -n <p> %s <size of
array> \n", argv[0]);
            fflush(stderr);
            *arraySize = -1;
        }
        else if ((atoi(argv[1])) %
numProcs != 0) {
            fprintf(stderr, "size
of array must be divisible by
number of processes \n");
            fflush(stderr);
            *arraySize = -1;
        }
        else {
            *arraySize =
atoi(argv[1]);
        }
    }
    // broadcast arraySize to all
processes
    MPI_Bcast(arraySize, 1,
MPI_INT, 0, MPI_COMM_WORLD);

    // negative arraySize ends the
program
    if (*arraySize <= 0) {
        MPI_Finalize();
        exit(-1);
    }
}

/*--------------------------------
---------------------------------
-
 * Function:   fillArray
 * Purpose:    Fill array with
random integers
 * Params:     array, the array
being filled
 *             arraySize, size of
the array
 *                  id, rank of
the current process
 */

void fillArray(int array[], int
arraySize, int id) {
    int i;
    // use current time as seed
for random generator
    srand(id + time(0));
    for (i = 0; i < arraySize;
i++) {
        array[i] = rand() % 100;
//INT_MAX
    }
}
```

```
/*--------------------------------
---------------------------------
-
 * Function:   printList
 * Purpose:    Prints the contents
of a given list of a process
 * Params:     id, rank of the
current process
 *                  arrayName,
name of array
 *                  array, array
to print
 *             arraySize, size of
array
 */

void printList(int id, char
arrayName[], int array[], int
arraySize) {
    printf("Process %d, %s: ", id,
arrayName);
    for (int i = 0; i < arraySize;
i++) {
        printf(" %d", array[i]);
    }
    printf("\n");
}

/*--------------------------------
---------------------------------
-
 * Function:    Compare - An
Introduction to Parallel
Programming by Pacheco
 * Purpose:     Compare 2 ints,
return -1, 0, or 1, respectively,
when
 *              the first int is
less than, equal, or greater than
 *              the second.  Used
by qsort.
 */

int compare(const void* a_p, const
void* b_p) {
    int a = *((int*)a_p);
    int b = *((int*)b_p);

    if (a < b)
        return -1;
    else if (a == b)
        return 0;
    else /* a > b */
        return 1;
}

/*--------------------------------
---------------------------------
-
```

```
 * Function:    merge
 * Purpose:     Merges half1 array
and half2 array into mergeResult
 * Params:      half1, first half
of array to merge
 *                half2, second
half of array to merge
 *
      mergeResult, array to store
merged result
 *                      size, size
of half1 and half2
 */
int* merge(int half1[], int
half2[], int mergeResult[], int
size) {
    int ai, bi, ci;
    ai = bi = ci = 0;
    // integers remain in both
arrays to compare
    while ((ai < size) && (bi <
size)) {
        if (half1[ai] <=
half2[bi]) {
            mergeResult[ci] =
half1[ai];
            ai++;
        }
        else {
            mergeResult[ci] =
half2[bi];
            bi++;
        }
        ci++;
    }
    // integers only remain in
rightArray
    if (ai >= size) {
        while (bi < size) {
            mergeResult[ci] =
half2[bi];
            bi++; ci++;
        }
    }
    // integers only remain in
localArray
    if (bi >= size) {
        while (ai < size) {
            mergeResult[ci] =
half1[ai];
            ai++; ci++;
        }
    }
    return mergeResult;
}

/*--------------------------------
--------------------------------
-
 * Function:    mergeSort
```

```
 * Purpose:     implements merge
sort: merges sorted arrays from
 *              processes until we
have a single array containing all
 *                integers in
sorted order
 * Params:        height, height
of merge sort tree
 *                       id, rank
of the current process
 *
      localArray, local array
containing integers of current
process
 *                       size, size
of localArray on current process
 *                       comm, MPI
communicator
 *
      globalArray, globalArray
contains either all integers
 *                              if
process 0 or NULL for other
processes
 */

int* mergeSort(int height, int id,
int localArray[], int size,
MPI_Comm comm, int globalArray[])
{
    int parent, rightChild,
myHeight;
    int* half1, * half2, *
mergeResult;

    myHeight = 0;
    qsort(localArray, size,
sizeof(int), compare); // sort
local array
    half1 = localArray;  // assign
half1 to localArray

    while (myHeight < height) { //
not yet at top
        parent = (id & (~(1 <<
myHeight)));

        if (parent == id) { //
left child
            rightChild = (id | (1
<< myHeight));

            // allocate memory and
receive array of right child
            half2 =
(int*)malloc(size * sizeof(int));
            MPI_Recv(half2, size,
MPI_INT, rightChild, 0,
                MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
            // allocate memory for
result of merge
            mergeResult =
(int*)malloc(size * 2 *
sizeof(int));
            // merge half1 and
half2 into mergeResult
            mergeResult =
merge(half1, half2, mergeResult,
size);
            // reassign half1 to
merge result
            half1 = mergeResult;
            size = size * 2;  //
double size

            free(half2);
            mergeResult = NULL;

            myHeight++;

        }
        else { // right child
           // send local array to
parent
            MPI_Send(half1, size,
MPI_INT, parent, 0,
MPI_COMM_WORLD);
            if (myHeight != 0)
free(half1);
            myHeight = height;
        }
    }

    if (id == 0) {
        globalArray = half1;   //
reassign globalArray to half1
    }
    return globalArray;
}

/*---------------------------------
---------------------------------
-*/

int main(int argc, char** argv) {
    int numProcs, id,
globalArraySize, localArraySize,
height;
    int* localArray, *
globalArray{};
    double startTime, localTime,
totalTime;
    double zeroStartTime,
zeroTotalTime, processStartTime,
processTotalTime;;
    int length = -1;
    char
myHostName[MPI_MAX_PROCESSOR_NAME]
;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
&numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD,
&id);


MPI_Get_processor_name(myHostName,
&length);

    // check for odd processes
    powerOfTwo(id, numProcs);

    // get size of global array
    getInput(argc, argv, id,
numProcs, &globalArraySize);

    // calculate total height of
tree
    height = log2(numProcs);

    // if process 0, allocate
memory for global array and fill
with values
    if (id == 0) {
        globalArray =
(int*)malloc(globalArraySize *
sizeof(int));
        fillArray(globalArray,
globalArraySize, id);
        //printList(id, "UNSORTED
ARRAY", globalArray,
globalArraySize);  // Line A
    }

    // allocate memory for local
array, scatter to fill with values
and print
    localArraySize =
globalArraySize / numProcs;
    localArray =
(int*)malloc(localArraySize *
sizeof(int));
    MPI_Scatter(globalArray,
localArraySize, MPI_INT,
localArray,
        localArraySize, MPI_INT,
0, MPI_COMM_WORLD);
    //printList(id, "localArray",
localArray, localArraySize);   //
Line B

    //Start timing
    startTime = MPI_Wtime();
    //Merge sort
    if (id == 0) {
        zeroStartTime =
MPI_Wtime();
```

```
        globalArray =
mergeSort(height, id, localArray,
localArraySize, MPI_COMM_WORLD,
globalArray);
        zeroTotalTime =
MPI_Wtime() - zeroStartTime;
        printf("Process #%d of %d
on %s took %f seconds \n",
            id, numProcs,
myHostName, zeroTotalTime);
    }
    else {
        processStartTime =
MPI_Wtime();
        mergeSort(height, id,
localArray, localArraySize,
MPI_COMM_WORLD, NULL);
        processTotalTime =
MPI_Wtime() - processStartTime;
        printf("Process #%d of %d
on %s took %f seconds \n",
            id, numProcs,
myHostName, processTotalTime);
    }
    //End timing
    localTime = MPI_Wtime() -
startTime;
    MPI_Reduce(&localTime,
&totalTime, 1, MPI_DOUBLE,
        MPI_MAX, 0,
MPI_COMM_WORLD);

    if (id == 0) {
        //printList(0, "FINAL
SORTED ARRAY", globalArray,
globalArraySize);  // Line C
        printf("Sorting %d
integers took %f seconds \n",
globalArraySize, totalTime);
        free(globalArray);
    }

    free(localArray);
    MPI_Finalize();
    return 0;
}
```

Kode diatas merupakan kode untuk melakukan proses *sorting* menggunakan merge sort. Dilakukan 4 proses, dimana proses ini akan membagi (*scatter)* data menjadi 2 bagian array. Selanjutnya, data-data yang sudah terbagi tadi akan diurutkan satu persatu perbagiannya. Setelah itu, hasil pengurutan akan disatukan kembali (*merge)* menjadi 2 bagian array, yang selanjutnya akan di-*merge* kembali menjadi satu kesatuan data. Berikut fungsi-fungsi yang digunakan untuk melakukan proses tersebut :

- powerOfTwo : mengecek apakah proses yang digunakan adalah proses "power of two".
- getInput : mendapatkan input dari pengguna untuk ukuran array.
- fillArray : mengisi array dengan integer acak
- printList : mencetak isi dari list proses yang dijalankan
- compare : membandingkan 2 buah integer, akan mengembalikan nilai -1 jika bilangan pertama bernilai lebih kecil dari bilangan kedua, mengembalikan nilai 0 apabila kedua bilangan bernilai sama, dan mengembalikan nilai 1 jika bilangan pertama lebih besar dari bilangan kedua.
- merge : untuk menggabungkan dua bagian array yang sudah terbagi menjadi setengah kedalam mergeResult. Dimana half1 merupakan setengah bagian awal dan half2 merupakan setengah bagian akhir, sementara mergeResult merupakan array untuk menyimpan ukuran hasil penggabungan array half1 dan half2.
- mergeSort : fungsi untuk mengimplementasikan *merge sort*. Fungsi ini digunakan untuk menggabungkan array

dari semua proses sampai menjadi satu buah array yang berisi semua bilangan yang sudah terurut.

- main : driver utama untuk menjalankan program.

Berikut merupakan kode program untuk *sequential merge sort* :

```
/*
 * mergeSortSeq.c
 * ...illustrates sequential merge
sort.
 *
 * Hannah Sonsalla, Macalester
College 2017
 *
 * Usage:  ./mergeSort <arraySize>
 *
 * Notes:
 *  - To view initial unsorted
array uncomment line A
 *  - To view final sorted array
uncomment line B
 */

#include <stdio.h>   // printf
#include <stdlib.h>  // malloc,
free, rand(), srand()
#include <time.h>    // time for
random generator and timing
#include <string.h>  // memcpy
#include <limits.h>  // INT_MAX

 /* Declaration of functions */
void getInput(int argc, char*
argv[], int* arraySize);
void fillArray(int array[], int
arraySize);
void printList(char arrayName[],
int array[], int arraySize);
int compare(const void* a_p, const
void* b_p);
int* mergeSort(int array[], int
arraySize);



/*-------------------------------
---------------------------------
-
 * Function:   getInput
 * Purpose:    Get input from user
for array size
 * Params:     argc, argument
count
 *                   argv[],
points to argument vector
 *                   arraySize,
points to array size
 */

void getInput(int argc, char*
argv[], int* arraySize) {
    if (argc != 2) {
        fprintf(stderr, "usage:
%s <number of tosses> \n",
argv[0]);
        fflush(stderr);
        *arraySize = -1;
    }
    else {
        *arraySize =
atoi(argv[1]);
    }

    // 0 totalNumTosses ends the
program
    if (*arraySize <= 0) {
        exit(-1);
    }
}

/*-------------------------------
---------------------------------
-
 * Function:   fillArray
 * Purpose:    Fill array with
random integers
 * Params:     array, the array
being filled
 *             arraySize, size of
the array
 */

void fillArray(int array[], int
arraySize) {
    int i;
    // use current time as seed
for random generator
    srand(time(0));
    for (i = 0; i < arraySize;
i++) {
        array[i] = rand() % 100;
//INT_MAX
    }
}

/*-------------------------------
---------------------------------
-
 * Function:   printList
 * Purpose:    Prints the contents
of a given list of a process
 * Params:     arrayName, name of
```

```
array
 *                   array, array
to print
 *              arraySize, size of
array
 */

void printList(int array[], int
arraySize) {
    for (int i = 0; i < arraySize;
i++) {
        printf(" %d", array[i]);
    }
    printf("\n");
}

/*--------------------------------
---------------------------------
-
 * Function:    Compare - An
Introduction to Parallel
Programming by Pacheco
 * Purpose:     Compare 2 ints,
return -1, 0, or 1, respectively,
when
 *              the first int is
less than, equal, or greater than
 *              the second.  Used
by qsort.
 */

int compare(const void* a_p, const
void* b_p) {
    int a = *((int*)a_p);
    int b = *((int*)b_p);

    if (a < b)
        return -1;
    else if (a == b)
        return 0;
    else /* a > b */
        return 1;
}

/*--------------------------------
---------------------------------
-
 * Function:    mergeSort
 * Purpose:     implements merge
sort: merges sorted arrays from
 *              processes until we
have a single array containing all
 *                integers in
sorted order
 * Params:        array, array on
which to perform merge sort
 *                     arraySize,
size of array
 */
```

```
int* mergeSort(int array[], int
arraySize) {

    qsort(array, arraySize,
sizeof(int), compare); // sort
    return array;
}

/*--------------------------------
---------------------------------
-*/

int main(int argc, char** argv) {
    int arraySize;
    int* array;
    clock_t startTime, endTime;

    // get size of array
    getInput(argc, argv,
&arraySize);

    // allocate memory for global
array and fill with values
    array = (int*)malloc(arraySize
* sizeof(int));
    fillArray(array, arraySize);
    printList(array, arraySize);
// Line A

    //Start timing
    startTime = clock();

    //Merge sort
    array = mergeSort(array,
arraySize);

    //End timing
    endTime = clock();

    printList(array, arraySize);
// Line B
    printf("Sorting %d integers
took %f seconds \n", arraySize,
(double)(endTime - startTime) /
CLOCKS_PER_SEC);
    free(array);

    return 0;
}
```

Program *sequential merge sort* kurang lebih menggunakan fungsi yang sama dengan MPI *merge sort*. Jika pada MPI *merge sort* pengurutan diselesaikan dengan menggunakan fungsi-fungsi MPI, pada

sequential merge sort proses pengurutan diselesaikan dengan cara sekuensial.

Hasil output program MPI :

- 1000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortMPI\Debug>mpiexec -n 4 mergeSortMPI.exe 1000
Process #3 of 4 on MSI took 0.000397 seconds
Process #0 of 4 on MSI took 0.000523 seconds
Sorting 1000 integers took 0.001862 seconds
Process #2 of 4 on MSI took 0.000491 seconds
Process #1 of 4 on MSI took 0.000426 seconds
```

- 10000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortMPI\Debug>mpiexec -n 4 mergeSortMPI.exe 10000
Process #2 of 4 on MSI took 0.000646 seconds
Process #3 of 4 on MSI took 0.000546 seconds
Process #1 of 4 on MSI took 0.000571 seconds
Process #0 of 4 on MSI took 0.000700 seconds
Sorting 10000 integers took 0.000742 seconds
```

- 100000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortMPI\Debug>mpiexec -n 4 mergeSortMPI.exe 100000
Process #3 of 4 on MSI took 0.004825 seconds
Process #1 of 4 on MSI took 0.004530 seconds
Process #0 of 4 on MSI took 0.005393 seconds
Sorting 100000 integers took 0.005433 seconds
Process #2 of 4 on MSI took 0.005044 seconds
```

- 1000000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortMPI\Debug>mpiexec -n 4 mergeSortMPI.exe 1000000
Process #0 of 4 on MSI took 0.049659 seconds
Sorting 1000000 integers took 0.049717 seconds
Process #3 of 4 on MSI took 0.044245 seconds
Process #2 of 4 on MSI took 0.045966 seconds
Process #1 of 4 on MSI took 0.043814 seconds
```

- 10000000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortMPI\Debug>mpiexec -n 4 mergeSortMPI.exe 10000000
Process #3 of 4 on MSI took 0.484727 seconds
Process #1 of 4 on MSI took 0.474127 seconds
Process #2 of 4 on MSI took 0.507547 seconds
Process #0 of 4 on MSI took 0.541623 seconds
Sorting 10000000 integers took 0.541686 seconds
```

- 100000000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortMPI\Debug>mpiexec -n 4 mergeSortMPI.exe 100000000
Process #0 of 4 on MSI took 5.268437 seconds
Sorting 100000000 integers took 5.268515 seconds
Process #2 of 4 on MSI took 4.938231 seconds
Process #1 of 4 on MSI took 4.703627 seconds
Process #3 of 4 on MSI took 4.616351 seconds
```

Hasil output program menggunakan *sequential* :

- 1000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortSeq\Debug>mpiexec -n 1 mergeSortSeq.exe 1000
Sorting 1000 integers took 0.000000 seconds
```

- 10000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortSeq\Debug>mpiexec -n 1 mergeSortSeq.exe 10000
Sorting 10000 integers took 0.002000 seconds
```

- 100000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortSeq\Debug>mpiexec -n 1 mergeSortSeq.exe 100000
Sorting 100000 integers took 0.018000 seconds
```

- 1000000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortSeq\Debug>mpiexec -n 1 mergeSortSeq.exe 1000000
Sorting 1000000 integers took 0.168000 seconds
```

- 10000000 Bilangan

```
C:\Users\bagas\source\repos\mergeSortSeq\Debug>mpiexec -n 1 mergeSortSeq.exe 10000000
Sorting 10000000 integers took 1.689000 seconds
```

- 100000000 Bilangan

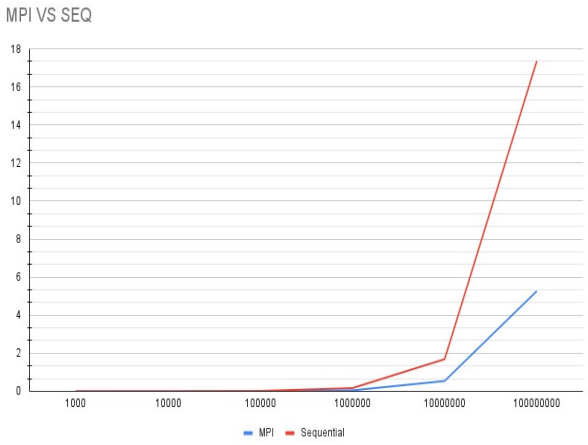
```
C:\Users\bagas\source\repos\mergeSortSeq\Debug>mpiexec -n 1 mergeSortSeq.exe 100000000
Sorting 100000000 integers took 17.366000 seconds
```

Berdasarkan kedua hasil diatas, dapat disimpulkan bahwa MPI terbukti lebih cepat dalam melakukan *merge sorting*, terutama jika terdapat banyak bilangan yang ingin diurutkan. Hal ini dibuktikan pada *sorting* seratus juta bilangan, *sequential merge sort* membutuhkan waktu 17.36 seken, sementara MPI *merge sort* hanya membutuhkan waktu 5.26 seken.

Berikut disajikan tabel mengenai perbandingan waktu pengurutan *merge sort* antara MPI dengan *sequential*.

Jumlah Data	MPI	SEQ
1000	0.001862	0
10000	0.000742	0.002
100000	0.005044	0.018
1000000	0.049717	0.168
10000000	0.541686	1.689
100000000	5.268515	17.366

Dibawah ini juga disediakan grafik mengenai tabel perbandingan diatas.



Gambar 4. Grafik Perbandingan Waktu Pengurutan MPI dengan Sequential

4. PENUTUP

4.1 Kesimpulan

Message Passing Interface atau yang disingkat MPI merupakan sebuah standar untuk interface dalam pemrograman paralel. MPI menggunakan proses komunikasi mengirim dan menerima untuk menyelesaikan suatu tugas, seperti *sorting* menggunakan *merge sorting*. *Merge Sort* merupakan salah satu cara *sorting* dimana sorting akan dilakukan dalam beberapa bagian data, yang nantinya akan menjadi satu buah array data yang sudah terurut. Kami mengimplementasikan MPI menggunakan MSI GF63 Thin, Sistem Operasi: Windows 10 Prosesor: Intel core i5-11400H , memory: RAM 8 GB. Untuk mengetahui efisiensi waktu pengurutan, dibandingkan dua cara yaitu MPI *merge sort* dan *sequential merge sort*. MPI terbukti memiliki waktu pengurutan yang lebih efisien dalam melakukan *sorting* menggunakan *merge sort*.

DAFTAR PUSTAKA

[1] Cormen, Thomas H. et al. 2009. Introduction to Algorithms (3rd ed.). Cambridge : MIT Press

[2] Indrayana & Ihsan, M.F. 2005. Perbandingan Kecepatan/Waktu Komputasi Beberapa Algoritma Pengurutan (Sorting). Bandung: Program Studi Teknik Informatika, Institut Teknologi Bandung.

[3] Suarga. 2012. Algoritma dan Pemrograman . Andi : Yogyakarta.

[4] Erzandi, M.O. 2009. Algoritma Pengurutan Dalam Pemrograman. Bandung: Program Studi Teknik Informatika, Institut Teknologi Bandung.

[5] Knuth, D. 1998. "Section 5.2.4: Sorting by Merging ". Sorting and Searching. The Art of Computer Programming 3 (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0201-89685-0.

[6] Munir, R. 2005. Diktat Kuliah IF2251 Strategi Algoritmik. Bandung : Lab. Ilmu dan Rekayasa Komputasi, Departemen Teknik Informatika, Institut Teknologi