JAWABAN UJIAN TENGAH SEMESTER

IF 655 – Distributed Data Processing

Christianto Vinsen Budijanto

00000028917

Kelas : A

No. Absen : 24

Informatika

Fakultas Teknik & Informatika

UNIVERSITAS MULTIMEDIA NUSANTARA

2021

DAFTAR ISI

Nama **: Christianto Vinsen Budijanto**

NIM **: 00000028917**

Kelas **: A**

No. Absen **: 24**

1. Soal 1

Berikan penjelasan komputer yang digunakan untuk menjalankan program:

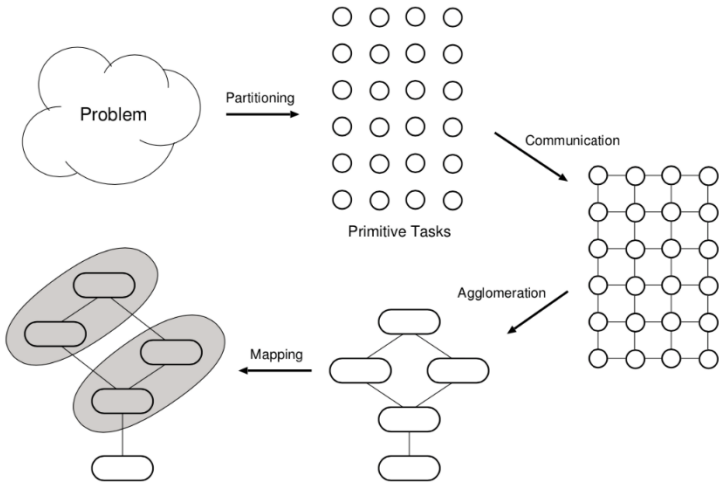
Sistem Operasi : Microsoft Windows

Jumlah Prosesor : 2 Core

Jumlah Thread : 4 Thread

2. Soal 2

Jelaskan bagaimana tahapan perancangan algoritma paralel menggunakan "Foster's Design Methodology" untuk menyelesaikan persamaan integral dengan metode trapesiodal.



Tahapan desain metodologi Foster's:

a. Partitioning

Pada tahap partisi terjadi proses pembagian data atau komputasi menjadi beberapa tasks.

Pada algoritma paralel disini, digunakan metode *Domain Decomposition*. *Domain Decomposition* adalah pendekatan desain algoritma paralel di mana pertama-tama membagi data menjadi beberapa bagian dan kemudian menentukan cara mengaitkan perhitungan dengan data, dan fokusnya pada struktur data terbesar dan atau paling sering diakses dalam program. Proses *Domain Decomposition* di algoritma paralel ini dilakukan dengan membagi data yang akan diiterasi pada *grids* (atau variabel 'iterasi' pada soal) menjadi beberapa bagian sesuai dengan jumlah proses yang digunakan dengan porsi yang kurang lebih sama untuk setiap bagian.

b. Communication

Komunikasi adalah media atau saluran yang menghubungkan suatu proses dengan proses lainnya dengan tujuan untuk mengirim dan menerima data dari atau ke proses lainnya.

Pada algoritma paralel disini, terjadi proses komunikasi pada semua proses ke satu proses utama yang dijadikan sebagai *root* proses dengan menggunakan *MPI (Message Passing Interface)*. Gambaran singkatnya, jika terdapat 4 proses yang dijalankan, maka akan dipilih 1 proses sebagai proses *root* atau utama yang akan berkomunikasi dengan proses lainnya. Komunikasi ini bertujuan untuk mengumpulkan semua data hasil perhitungan dari semua proses ke 1 proses utama (biasanya proses dengan id 0).

c. Agglomeration

Agglomeration merupakan proses pengelompokkan dari dua atau lebih *tasks* menjadi *tasks* yang lebih besar, dengan tujuan meningkatkan kinerja dan menyederhanakan program.

Pada algoritma paralel disini, proses pengelompokkan terjadi ketika jumlah proses yang dijalankan lebih besar daripada total core pada komputer. Karena setiap core hanya dapat menjalankan 1 proses setiap waktunya, maka dilakukan pengelompokkan proses-proses menjadi *tasks* yang lebih besar agar jumlahnya tidak lebih dari jumlah core pada komputer. Sehingga dapat meningkatkan dan mengoptimalkan kinerja dari program.

d. Mapping

Mapping merupakan proses untuk melakukan *assign tasks* ke *physical resources* (processor, cores).

Pada algoritma ini, setelah proses-proses dilakukan pengelompokkan menjadi *tasks-tasks* yang lebih besar, langkah selanjutnya adalah memetakan setiap *tasks* tersebut ke core pada komputer guna mengeksekusi dan menyelesaikan program

3. Soal 3

Listing program paralel dengan MPI:

```
#include <stdio.h>
#include <Windows.h>
#include <mpi.h>

// Function to get low block for specified process
int Low(int process_id, int number_of_processes, long number_of_grids);
// Function to get high block for specified process
int High(int process_id, int number_of_processes, long number_of_grids);
// Function to get size of block for specified process
int Size(int process_id, int number_of_processes, long number_of_grids);

// Function to calculate integral value
double F(double x);
// Function Integral Trapezoidal
double Trapezoidal(double lower_limit, double upper_limit, long number_of_grids, long
first_block, long last_block);
// Function to get system times
double GetTime();

// Main Function Using min max block before edit
int main(int argc, char* argv[]) {
    // Variable Declaration and Initialization

    // Process ID and Total Process
    int process_id, number_of_processes;
    // Lower and Upper Limit in equation
    double lower_limit = 10;
    double upper_limit = 50;
    // Total iteraions or grids
    long number_of_grids = 40000000;
    // First and Last Block in each Process
    long first_block, last_block;
    // Total time used
    double elapsed_time, max_elapsed_time;
    // Variable to store the result of equation
    double local_result, global_result;

    // Initialize the MPI Execution Environment
    MPI_Init(&argc, &argv);
    // Blocks until all processes in the communicator have reached this routine
    MPI_Barrier(MPI_COMM_WORLD);
    // Get the Initial Time
    elapsed_time = -GetTime();

    // Determines the rank of the calling process in the communicator
    MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
    // Determines the size of the group associated with a communicator
    MPI_Comm_size(MPI_COMM_WORLD, &number_of_processes);

    // Get first and last block of this process
    first_block = Low(process_id, number_of_processes, number_of_grids - 1) + 1;
    last_block = High(process_id, number_of_processes, number_of_grids - 1) + 1;

    // Get the local result of process from Trapezoidal function
    local_result = Trapezoidal(lower_limit, upper_limit, number_of_grids, first_block,
last_block);

    // Get total values of result on all processes to a single value
    MPI_Reduce(&local_result, &global_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    // Blocks until all processes in the communicator have reached this routine
```

```
    MPI_Barrier(MPI_COMM_WORLD);
    // Get the Final Time
    elapsed_time += GetTime();
    // Get maximum values of elapsed_time on all processes to a single value
    MPI_Reduce(&elapsed_time, &max_elapsed_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    //printf("%d) local count = %lf with time = %lf second\n", process_id, local_result,
elapsed_time);
    //fflush(stdout);

    if (process_id == 0) {
        // Print the results
        printf("Nilai Integral = %lf \nWaktu (Tp)     = %lf second\n", global_result,
max_elapsed_time);
        fflush(stdout);
    }

    // Terminates MPI execution environment
    MPI_Finalize();
    return 0;
}

// Function to get low block for specified process
int Low(int process_id, int number_of_processes, long number_of_grids) {
    return((process_id * number_of_grids) / number_of_processes);
}
// Function to get high block for specified process
int High(int process_id, int number_of_processes, long number_of_grids) {
    return((process_id + 1) * number_of_grids / number_of_processes - 1);
}
// Function to get size of block for specified process
int Size(int process_id, int number_of_processes, long number_of_grids) {
    return(Low(process_id + 1, number_of_processes, number_of_grids) - Low(process_id,
number_of_processes, number_of_grids));
}

// Function to calculate integral value
double F(double x) {
    double h = 3 * x * x + 4 * x + 12;
    return (h);
}
// Function Integral Trapezoidal
double Trapezoidal(double lower_limit, double upper_limit, long number_of_grids, long
first_block, long last_block) {
    double xi, yi;
    double h = (upper_limit - lower_limit) / number_of_grids; // Grid spacing
    double y = 0.5 * h * (F(upper_limit) + F(lower_limit));

    for (int i = first_block; i <= last_block; i++) {
        xi = lower_limit + i * h;
        yi = F(xi);
        y += h * yi;
    }

    return y;
}
// Function to get system times
double GetTime() {
    SYSTEMTIME time;
    GetSystemTime(&time);
    LONG time_ms = (time.wSecond * 1000) + time.wMilliseconds; // msec
    return(time_ms / 1000.0); // second
        }
```

Hasil Output:

```
C:\Users\Christianto Vinsen\source\repos\UTS_Final\Debug>mpiexec -n 2 UTS_Final.exe
Nilai Integral = 129280.004032
Waktu (Tp)     = 0.995000 second

C:\Users\Christianto Vinsen\source\repos\UTS_Final\Debug>mpiexec -n 4 UTS_Final.exe
Nilai Integral = 129280.012096
Waktu (Tp)     = 0.710000 second
```

4. Soal 4

Tabel eksekusi program paralel dengan menggunakan 4 proses

Jumlah Iterasi atau Grids (N)	Waktu Eksekusi (Second)	Screenshot

10.000	0	`Jumlah Iterasi = 10000`
		`Nilai Integral = 129328.384320`
		`Waktu (Tp) = 0.000000 second`

50.000	0,001	`Jumlah Iterasi = 50000`
		`Nilai Integral = 129289.676813`
		`Waktu (Tp) = 0.001000 second`

100.000	0,002	`Jumlah Iterasi = 100000`
		`Nilai Integral = 129284.838403`
		`Waktu (Tp) = 0.002000 second`

500.000	0,008	`Jumlah Iterasi = 500000`
		`Nilai Integral = 129280.967680`
		`Waktu (Tp) = 0.008000 second`

1.000.000	0,017	`Jumlah Iterasi = 1000000`
		`Nilai Integral = 129280.483840`
		`Waktu (Tp) = 0.017000 second`

2.000.000	0,051	`Jumlah Iterasi = 2000000`
		`Nilai Integral = 129280.241920`
		`Waktu (Tp) = 0.051000 second`

5.000.000	0,088	`Jumlah Iterasi = 5000000`
		`Nilai Integral = 129280.096768`
		`Waktu (Tp) = 0.088000 second`

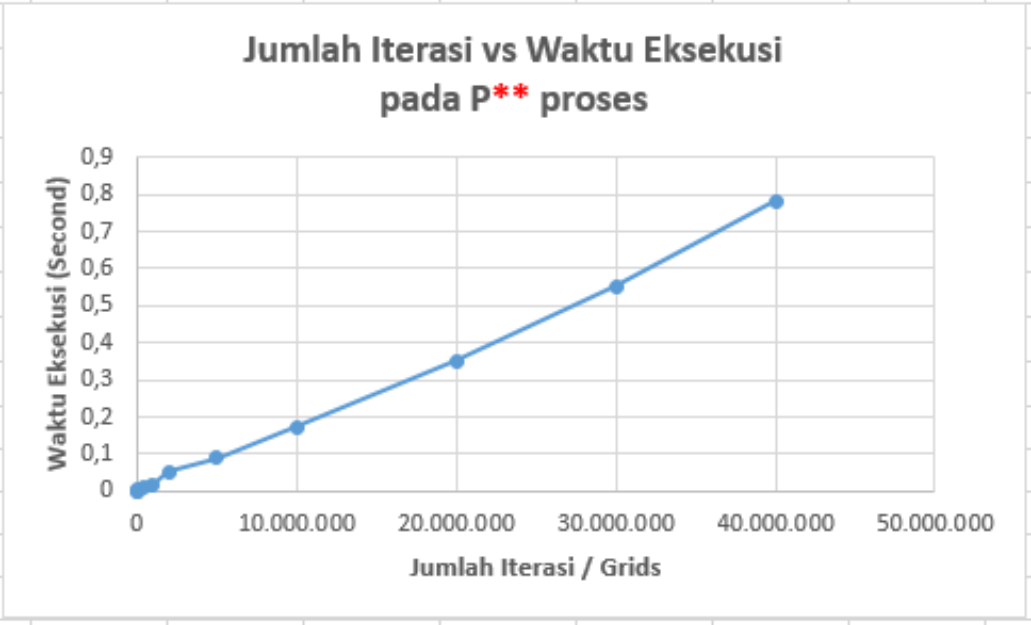
10.000.000	0,173	`Jumlah Iterasi = 10000000`
		`Nilai Integral = 129280.048384`
		`Waktu (Tp) = 0.173000 second`

20.000.000	0,351	Jumlah Iterasi = 20000000 Nilai Integral = 129280.024192 Waktu (Tp) = 0.351000 second
30.000.000	0,552	Jumlah Iterasi = 30000000 Nilai Integral = 129280.016128 Waktu (Tp) = 0.552000 second
40.000.000	0,782	Jumlah Iterasi = 40000000 Nilai Integral = 129280.012096 Waktu (Tp) = 0.782000 second

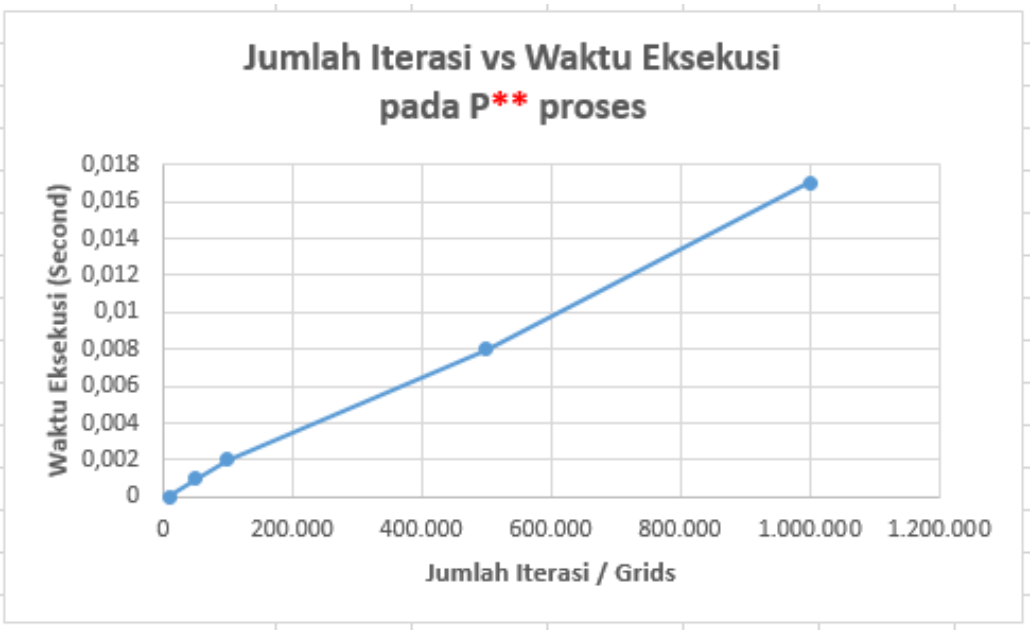
5. Soal 5

Grafik eksekusi program paralel dengan menggunakan 4 proses dari data di soal no 4

Berikut adalah grafik eksekusi program mulai dari iterasi atau grids 10.000 sampai 40.000.000:



Karena terdapat kendala perbedaan jumlah data yang sangat besar dari data awal sampai data akhir 10.000 : 40.000.000, maka sulit untuk melihat grafik data awal, oleh karena itu saya melakukan visualisasi grafik lagi untuk 5 data pertama, berikut hasilnya:



6. Soal 6

Lakukan eksekusi program dengan jumlah iterasi 40.000.000 (Empat puluh juta) dengan jumlah proses yang berbeda-beda, berikut hasilnya:

Jumlah Proses	Waktu Eksekusi Sekuensial / Ts (Second)	Waktu Eksekusi Paralel / Tp (Second)	Speed Up (Ts/Tp)
2	2,031	0,975	2,083076923
4	2,804	0,687	4,081513828
8	5,496	0,688	7,988372093
16	11,138	0,7	15,91142857

7. Soal 7

Grafik speedup dari program sekuensial dan program paralel dengan menggunakan 40.000.000 data iterasi / grids dengan jumlah proses yang berbeda-beda:

8. Soal 8

Kesimpulan dari percobaan yang dilakukan

Dari hasil percobaan menggunakan algoritma sekuensial dan paralel pada perhitungan persamaan integral dengan metode numerik Trapezodial, didapat kesimpulan bahwa algoritma paralel jauh lebih baik dan optimal daripada algoritma sekuensial.

Pada percobaan dengan menggunakan jumlah proses yang sama dengan jumlah iterasi atau grids yang berbeda. Didapat hasil bahwa jumlah iterasi atau grids selalu berbanding lurus dengan waktu eksekusi program tersebut. Semakin sedikit jumlah iterasi atau grids, maka waktu eksekusi akan rendah, sedangkan semakin banyak jumlah iterasi atau grids, maka waktu eksekusi juga menjadi semakin tinggi. Namun semakin banyak jumlah iterasi atau grids, hasil perhitungan integral yang dihasilkan akan menjadi semakin akurat.

Pada percobaan dengan menggunakan jumlah iterasi atau grids yang sama, dengan jumlah proses yang berbeda. Didapat hasil bahwa waktu eksekusi pada program sekuensial selalu berbanding lurus dengan jumlah prosesnya, semakin banyak jumlah proses, maka semakin tinggi waktu eksekusi programnya. Sedangkan pada program paralel, waktu eksekusi program berbanding terbalik dengan jumlah prosesnya, semakin tinggi jumlah prosesnya, maka waktu eksekusi akan semakin rendah. Namun pada program paralel terdapat kasus ketika jumlah proses mencapai jumlah tertentu maka waktu eksekusinya tidak akan berkurang lagi, dalam hal ini waktu eksekusi akan menjadi sama atau bahkan lebih tinggi dari waktu eksekusi dengan jumlah proses yang lebih sedikit. Hal ini disebabkan karena semakin banyak

jumlah proses yang digunakan, maka akan berpengaruh pada waktu komunikasi setiap proses yang meningkat. Waktu komunikasi disini terjadi ketika proses-proses mengumpulkan nilai hasil perhitungan ke suatu proses utama guna dilakukan penggabungan hasil-hasil tersebut.

Pada percobaan menghitung *speed up* antara algoritma sekuensial dan paralel, didapat bahwa *speed up* berbanding lurus dengan jumlah proses yang digunakan. Bisa saja terdapat kasus dimana *speed up* tidak meningkat saat mencapai suatu jumlah proses tertentu namun pada percobaan ini belum dapat dibuktikan karena keterbatasan prosesor. Selain itu, *speed up* juga berbanding lurus dengan jumlah iterasi atau grids. Hal tersebut dikarenakan jika ukuran iterasi kecil, maka *overhead* komunikasi tidak tertutupi.