Laporan Praktikum 1
IF3230 Sistem Paralel dan Terdistribusi
"Pengenalan dan Penggunaan OpenMP – Quicksort"

Disusun oleh :

Atika Firdaus **13514009**

Adam Rotal Y. **13514091**

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2016/2017

Deskripsi Solusi Paralel

Pada praktikum 1, kami diminta membuat implementasi permasalahan *quicksort* paralel menggunakan OpenMP. Solusi yang kami buat adalah membuat paralel pada empat titik berikut.

```
void quicksort_parallel(int *a, long lo, long hi) {
    long i, div;

    if (lo < hi) {
        int x = a[lo];
        div = lo;

        for (i = lo+1; i < hi; i++) {
            if (a[i] <= x) {
                div++;
                swap(a, div, i);
            }
        }

        swap(a, lo, div);

        #pragma omp task
        quicksort_parallel(a, lo, div);

        #pragma omp task
        quicksort_parallel(a, div+1, hi);
    }
}
```

Gambar 1
Penerapan paralel pada prosedur quicksort_parallel

Pada program utama, kami menginisiasi proses *sorting* dengan menggunakan **#pragma omp parallel num_threads(jumlah_thread)** agar proses *sorting* dilakukan secara paralel sebanyak jumlah thread yang didefinisikan *user*. Kemudian, digunakan **#pragma imp single nowait** agar tiap eksekusi *procedure* quicksort dibuat thread baru dan memastikan kelangsungan proses yang sudah selesai tanpa menunggu.

Pada *procedure* quicksort, setelah melakukan partisi, digunakan **#pragma omp task** sehigga eksekusi rekursif dilakukan oleh sebuah thread yang sedang aktif.

```c
int main(int argc, char *argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    int n;
    printf("Jumlah array : "); scanf("%d", &n);
    int a[n];
    int b[n];

    double start, end;

    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        int x = rand() % 999999;
        a[i] = x;
        b[i] = x;
    }

    start = omp_get_wtime();
    quicksort_serial(a, 0, n);
    end = omp_get_wtime();
    double elapsed1 = (end - start);

    start = omp_get_wtime();
    #pragma omp parallel num_threads(thread_count)
    {
        #pragma omp single nowait
        quicksort_parallel(b, 0, n);
    }
    end = omp_get_wtime();
    double elapsed2 = (end - start);

    printf("Time elapsed in serial: %f second\n", elapsed1);
    printf("Time elapsed in parallel: %f second\n", elapsed2);

    return 0;
}
```

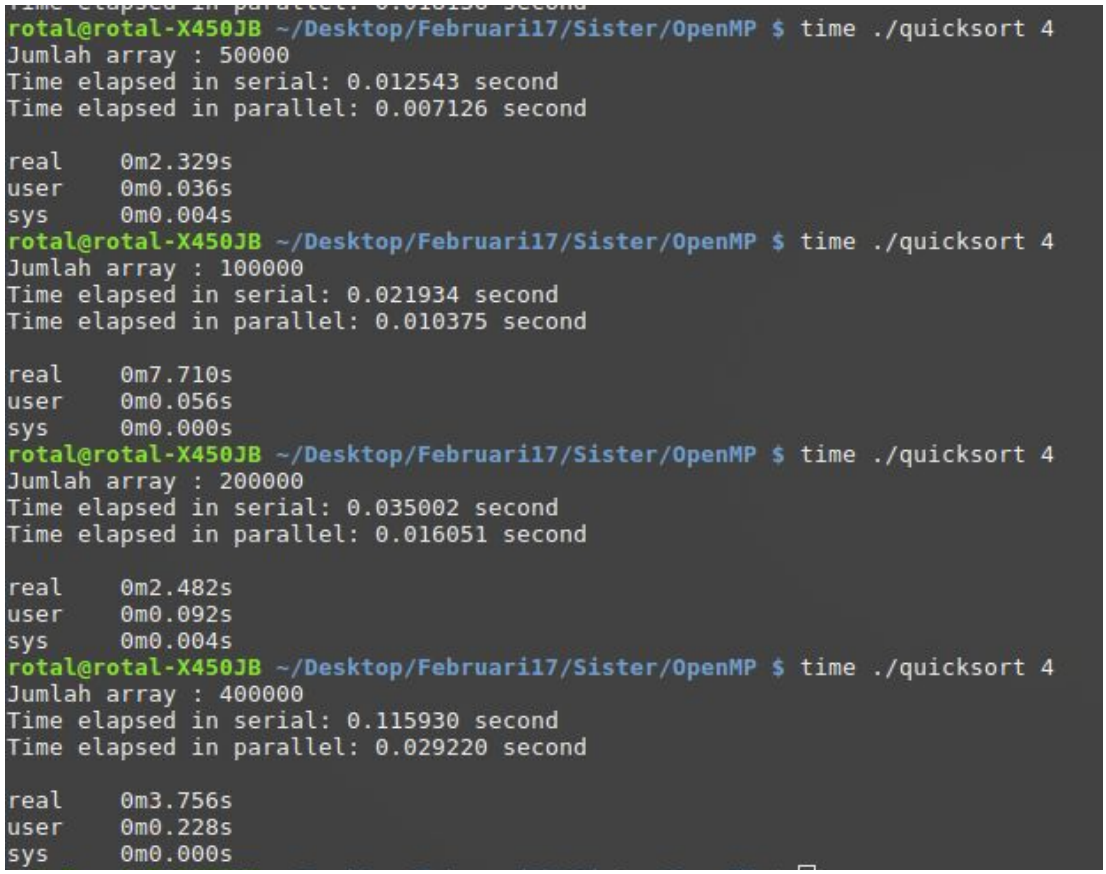
Gambar 2
Penerapan paralel pada *main program*

Analisis Solusi

Pada proses rekursif di *procedure* quicksort, kami membuat setiap rekursif dilakukan oleh thread yang berbeda dengan menggunakan **#pragma omp task**. Namun dikarenakan terdapat barrier maka kurang maksimalnya eksekusi paralel dibandingkan eksekusi secara serial. Untuk itu ditambahkan **#pragma omp parallel num_threads(jumlah_thread)** dan **#pragma omp single nowait** agar ketika suatu proses selesai melakukan eksekusi, maka proses tersebut tidak perlu menunggu proses yang lain karena tidak adanya barrier sehingga proses akan menjadi lebih cepat.

Pengukuran Performansi

Pengujian kami lakukan dengan mengeksekusi program menggunakan kedua metode, yaitu *quicksort* serial dan paralel, dengan jumlah *thread* sebanyak empat buah, dan menggunakan empat kasus yaitu *array of integer* berukuran 50.000, 100.000, 200.000, dan 400.000. Hasil eksekusi program dapat dilihat dalam gambar berikut.



Gambar 3
Hasil eksekusi program

Analisis Perbandingan Performansi Serial dan Paralel

Dari eksekusi program yang dilakukan, dapat terlihat bahwa pada seluruh kasus program paralel membutuhkan waktu yang lebih singkat daripada program serial untuk menyelesaikan tugasnya. Perbedaan waktu eksekusi antara program serial dan program paralel cukup signifikan, yaitu program paralel hampir dua kali lebih cepat dieksekusi daripada program serial.

Hal ini terjadi karena dengan penggunaan banyak *thread* pada program paralel berarti proses yang terjadi dapat dibagi-bagi menjadi proses kecil yang masing-masing dilakukan oleh *thread.* Dibandingkan program serial yang seluruh prosesnya dilakukan oleh satu *thread* yang mengerjakan seluruh pekerjaan, program paralel memanfaatkan banyak *thread* sehingga setiap *thread*-nya mengerjakan beban pekerjaan yang lebih sedikit.