



# **Modul Praktikum**

## Pemrograman Lanjut

**Laboratium Digital**



**Tim Penyusun Modul**

Tim Asisten Lab Digital 2021

Program Studi S1 Teknik Komputer

Fakultas Teknik

Universitas Indonesia

2021

# KATA PENGANTAR

Praktikum ini adalah bagian dari mata kuliah Pemrograman Lanjut dan Praktikum yang diberikan untuk mahasiswa Semester 2 Program Studi Teknik Komputer. Mata kuliah ini merupakan mata kuliah lanjutan yang diberikan kepada mahasiswa dari serangkaian mata kuliah untuk mendukung 2 CP Prodi, yaitu Mampu merancang algoritma untuk masalah tertentu dan mengimplementasikannya ke dalam pemrograman (C6) dan Mampu memanfaatkan teknologi informasi dan komunikasi (C3).

Capaian Pembelajaran Mata Kuliah (CPMK) dari mata kuliah ini adalah:

1. Mampu merancang program komputer prosedural kompleks dengan struktur data dinamis (C6)
2. Mampu menunjukkan sikap kritis, kreatif, dan inovatif dan menghargai orang lain dalam kelompok untuk memecahkan masalah bersama dan tugas kelompok Pemrograman Lanjut (C3, A3)
3. Mampu menggunakan software pemrogram komputer dengan mahir (C3)

Sedangkan Sub-CPMK yang akan dicapai adalah:

- 1.1. Mampu mengimplementasikan algoritma rekursif ke dalam pemrograman
- 1.2. Mampu mengimplementasikan algoritma searching dan sorting ke dalam pemrograman
- 1.3. Mampu membuat program komputer prosedural kompleks dengan linked list, stack dan queue
- 1.4. Mampu mengimplementasikan multi-threading dan parallel programming
- 1.5. Mampu merancang perangkat lunak sederhana dengan struktur data dinamis
- 2.1. Mampu menunjukkan proses berpikir kritis, kreatif dan inovatif dalam menyelesaikan permasalahan kelompok
- 2.2. Mampu berkomunikasi dengan sopan
- 2.3. Mampu menghargai pendapat orang lain
- 3.1. Mampu menggunakan software pemrogram komputer untuk program kompleks dengan mahir

Pada mata kuliah ini mahasiswa akan mengasah kemampuan cara berpikir dan penyelesaian masalah dengan membuat algoritma, kemudian menerjemahkan algoritma tersebut ke dalam bahasa pemrograman yang dapat dijalankan oleh komputer. Pada mata kuliah ini mahasiswa akan mempelajari Rekursif, Searching, Sorting, Linked list, Stack, Queue, Multi-threading, Parallel programming,, dan pada bagian akhir akan ditutup dengan proyek akhir pemrograman yang dibuat oleh mahasiswa.

Bahasa C merupakan bahasa pemrograman terstruktur, yang membagi program dalam sejumlah blok. Tujuannya adalah untuk mempermudah dalam pembuatan dan pengembangan program. Bahasa C menggunakan standarisasi ANSI (American National Standardization Institute) yang dijadikan acuan oleh para pembuat compiler C. Bahasa C terdiri dari fungsi-fungsi dan setiap program C memiliki fungsi utama yang disebut main. Program akan dieksekusi dimulai dari statement pertama pada fungsi main tersebut.

Akhir kata, diharapkan modul praktikum ini akan dapat menjadi referensi untuk membuat program dalam Bahasa C secara umum, dan menjadi panduan dalam menjalankan praktikum mata kuliah Pemrograman Lanjut, secara khusus.

Depok, Februari 2021

Tim Penyusun Modul

# MODUL 9: PARALLEL PROGRAMMING

## Definisi

Tujuan dari dilakukannya **Parallel Programming** adalah untuk menjalankan suatu proses secara *concurrent/parallel* sehingga dapat menjalankan suatu program dengan waktu eksekusi yang lebih singkat. Konsep dari melakukan Parallel Programming adalah untuk menggunakan *seluruh* resource yang dimiliki untuk mendistribusikan setiap *task* agar dapat meningkatkan sistem kerja.

Pada openMP setiap *task* dikumpulkan dalam sebuah *pool* dimana setiap thread yang sedang tidak melakukan proses apapun akan mengambil salah satu task dari *pool* tersebut. Namun, openMP tidak dapat mendeteksi jika terdapat beberapa *task* yang mengeksekusi instruksi yang sama. Oleh karena itu dalam *parallel region* perlu didefinisikan fungsi tertentu agar setiap thread dapat memproses secara paralel pada *range* yang berbeda tanpa menyebabkan Race Condition.

## Perbandingan dengan Multithreading

Pengimplementasian **Parallel Programming** memiliki tujuan spesifik yaitu untuk mempercepat proses kerja dari suatu komputasi dengan membagi *task* pada thread yang berbeda agar dapat dijalankan bersamaan. Dapat diibaratkan sebuah tukang yang mengerjakan suatu konstruksi rumah akan memakan waktu yang lebih lama dibandingkan 4 tukang yang mengerjakan pembangunan rumah tersebut. Sedangkan dalam multithreading suatu proses akan akan dijalankan oleh beberapa thread tetapi pembagian untuk setiap *thread* tersebut tidak diatur secara spesifik. Sehingga untuk setiap thread yang tidak memiliki proses untuk dijalankan akan mengambil *task* lain dari *pool*.

Contoh program tanpa parallel programming:

```

#include <stdio.h>
#include <omp.h>
#define SLEEP 10000
#define ARR_SIZE 500000 // 5e5
int main(int *argc, char *args){
    int a[ARR_SIZE];
    int i, sum = 0;
    double t1, t2;
    for (i = 0; i < ARR_SIZE; i++) //init
        a[i] = 1;
    t1 = omp_get_wtime();
    for (i = 0; i < ARR_SIZE; i++){ //counting
        sum += a[i];
        int j;
        for (j = 0; j < SLEEP; j++);
    }
    t2 = omp_get_wtime();
    printf("jumlah = %d. durasi = %g\n", sum , t2-t1);
    return 0;
}

```

Pada potongan program diatas, dilakukan pencarian lama waktu proses eksekusi diantara interval t2 dan t1. Interval antara t1 dan t2 sendiri mengapit blok kode untuk menjumlahkan data sebanyak 500000 buah. Outputnya sebagai berikut:

```

jumlah = 500000. durasi = 7.571
-----
Process exited after 14.46 seconds with return value 0
Press any key to continue . . .

```

Contoh program yang menggunakan multi-threading:

```

#include <stdio.h>
#include <omp.h>
#define SLEEP 10000
#define ARR_SIZE 500000 // 5e5
int main(int *argc, char *args){
    int a[ARR_SIZE];
    int i, sum = 0;
    double t1, t2;
    for (i = 0; i < ARR_SIZE; i++) //init
        a[i] = 1;
    t1 = omp_get_wtime();
    #pragma omp parallel private(i)
    {
        for (i = 0; i < ARR_SIZE; i++){ //counting
            sum += a[i];
            int j;
            for (j = 0; j < SLEEP; j++);
        }
    }
    t2 = omp_get_wtime();
    printf("jumlah = %d. durasi = %g\n", sum , t2-t1);
    return 0;
}

```

Pada potongan program diatas, diberi syntax `#pragma omp parallel private(i)` pada kode utama yang memungkinkan untuk menjalankan prgram secara multithread. Kecepatan eksekusinya pun lebih cepat dengan output sebagai berikut:

```

jumlah = 5899858. durasi = 10.057
-----
Process exited after 18.09 seconds with return value 0
Press any key to continue . . . _

```

Selain menggunakan syntax `#pragma omp parallel private(i)`, terdapat beberapa syntax lain yang dapat digunakan agar mempercepat proses eksekusi program. Diantaranya adalah:

- `#pragma omp for`  
Membuat compiler akan membuat serta mengalisis sendiri cara untuk membagi kode program.

```

t1 = omp_get_wtime();
#pragma omp parallel
{
    int localsum = 0;
    #pragma omp for
    for (i = 0; i < ARR_SIZE; i++){ //counting. i will be private
        localsum += a[i];
        int j; for (j = 0; j < SLEEP; j++); //add delay
    }
    #pragma omp critical
    sum += localsum;
}
t2 = omp_get_wtime();

```

- Reduction variable

Membuat variabel “sum” sementara pada tiap thread, lalu mengakumulasinya secara otomatis kedalam variabel “sum” yang sebenarnya.

```

t1 = omp_get_wtime();
#pragma omp parallel
{
    int localsum = 0;
    #pragma omp for reduction (+: sum)
    for (i = 0; i < ARR_SIZE; i++){ //counting. i will be private
        sum += a[i];
        int j; for (j = 0; j < SLEEP; j++); //add delay
    }
}
t2 = omp_get_wtime();

```

## Solusi dengan Parallel Programming

Agar setiap proses yang dikerjakan dapat terbagi pada setiap thread secara merata, maka dapat dilakukan sinkronisasi menggunakan metode barrier atau taskwait.

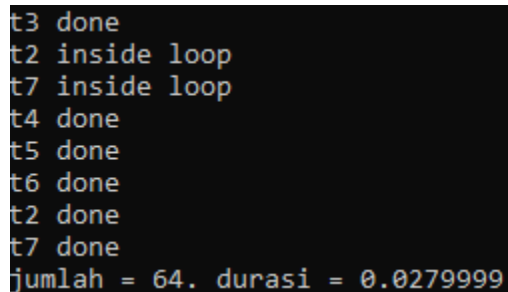
Metode barrier menggunakan #pragma omp barrier menyebabkan setiap thread yang telah selesai tidak dapat lanjut ke proses selanjutnya sebelum thread-thread lainnya menyelesaikan bagiannya. Proses ini dapat juga dispesifikan untuk thread tertentu agar tidak berada pada kondisi ini dengan #pragma omp nowait. Namun perlu diketahui bahwa barrier tidak boleh didefinisikan dalam pragma omp for karena dapat menyebabkan deadlock.



Contoh dari penggunaan nowait:

```
== barrier and nowait
#include <stdio.h>
#include <omp.h>
#define SLEEP 10000
#define ARR_SIZE 64 // 5e5
int main(int *argc, char *args){
    int a[ARR_SIZE];
    int i, sum = 0;
    double t1, t2;
    for (i = 0; i < ARR_SIZE; i++){a[i] = 1;} //init
    t1 = omp_get_wtime();
    #pragma omp parallel
    {
        int localsum = 0;
        #pragma omp for nowait
        for (i = 0; i < ARR_SIZE; i++){ //counting. i will be private
            localsum += a[i];
            printf("t%d inside loop\n",omp_get_thread_num());
            int j; for (j = 0; j < SLEEP; j++); //add delay
        }
        #pragma omp critical
        printf("t%d done\n",omp_get_thread_num());
        sum += localsum;
    }
    t2 = omp_get_wtime();
    printf("jumlah = %d. durasi = %g\n", sum , t2-t1);
    return 0;
}
```

Hasil Output Program:



```
t3 done
t2 inside loop
t7 inside loop
t4 done
t5 done
t6 done
t2 done
t7 done
jumlah = 64. durasi = 0.0279999
```

Dengan penggunaan nowait maka thread yang melakukan proses dapat menyelesaikannya tanpa menunggu thread lainnya yang belum menyelesaikan pekerjaannya. Sedangkan jika tidak digunakan nowait maka setiap thread akan menunggu satu sama lain.

Contoh dari penggunaan taskwait:

```

// === recursive task, correct
#include <stdio.h>
#include <omp.h>
#define ARR_SIZE 4
#define STEP 1
int a[ARR_SIZE];
int main(int *argc, char *args){
    int i, sum = 0;
    for (i = 0; i < ARR_SIZE; i++){a[i] = 1;} //init
    #pragma omp parallel num_threads(4)
    #pragma omp single
    sum = doSum(0,ARR_SIZE-1);
    return 0;
}
int doSum(int start, int end){
    int mid, x, y, res;
    if(end == start){
        res = a[start];
    }else{
        mid = (end + start)/2;
        printf("t%d sum(%d,%d) = sum(%d,%d) + sum(%d,%d)\n",
            omp_get_thread_num(),start,end,start,mid,mid+1,end);
        #pragma omp task shared(x)
        x = doSum(start, mid);
        #pragma omp task shared(y)
        y = doSum(mid+1, end);
        #pragma omp taskwait
        res = x + y;
    }
    printf(" t%d sum(%d,%d) = %d\n",omp_get_thread_num(),start,end,res);
    return res;
}

```

Output dari program:

```

t1 sum(0,3) = sum(0,1) + sum(2,3)
t0 sum(0,1) = sum(0,0) + sum(1,1)
t1 sum(2,3) = sum(2,2) + sum(3,3)
t3 sum(0,0) = 1
t0 sum(1,1) = 1
t0 sum(0,1) = 2
t1 sum(3,3) = 1
t2 sum(2,2) = 1
t1 sum(2,3) = 2
t1 sum(0,3) = 4

```

Dengan penggunaan taskwait maka proses kerja dari setiap thread akan tersinkronisasi untuk penjumlahannya dan tidak terdapat proses yang dijalankan lebih dari sekali.

Contoh Parallel Programming lainnya menggunakan OpenMP dapat dilihat pada referensi berikut:

<https://docs.microsoft.com/en-us/cpp/parallel/openmp/a-examples?view=msvc-160>