Faculty of Engineering & Technology
Department of Electrical & Computer Engineering
ENCS3390-OPERATING SYSTEMS
Report project #1

Prepared by:
Saja Asfour 1210737

Instructor:
Dr. Abdel Salam Sayyad
Section:
2

Date:
2-may-2024

Abstract:

This programming task aims to demonstrate multiprocessing and multithreading by calculating the average BMI using three different methods. The first approach is the naive method, which involves a single process without any child processes or threads. Following that, calculating the average BMI , will be performed using multiple processes with IPC facilitated through pipes. Additionally, a multithreaded application utilizing Pthreads will be developed for calculating average BMI , by employing joinable threads.

The primary objective is to analyze and compare the execution time of each method, thereby assessing their performance. By the end of this task, we aim to gain insights into the advantages and challenges associated with designing applications using multiple processes and multithreading.

Table of Contents

Table of Figure:

List of Tables:

Introduction:

Multiprocessing→ involves systems equipped with multiple central processing units (CPUs), each contributing to increased speed, power, and memory. With multiple CPUs, users can run multiple processes concurrently, as each CPU operates independently.

A child process in C language is usually made by calling fork () function. And the parent process should wait for its child's to finish their work by calling wait () function.

On the other hand, multithreading→ is a programming approach where multiple threads are assigned to a single process. This enables the simultaneous execution of multiple tasks within that process. Multithreading takes advantage of multi-core processors, enhancing computational speed and optimizing system memory usage. It ultimately improves the responsiveness and efficiency of software applications.

Within a program, a Thread is a separate execution path. It is a lightweight process that the operating system can schedule and run concurrently with other threads. The operating system creates and manages threads, and they share the same memory and resources as the program that created them. This enables multiple threads to collaborate and work efficiently within a single program.

Threads can be understood as individual sequences of execution within a process. They're often likened to lightweight processes because they share some characteristics with processes. Every thread is associated with precisely one process. In operating systems that facilitate multithreading , a single process can comprise multiple threads. However, for threads to function efficiently, it's beneficial to have more than one CPU available. Otherwise, in a scenario with only one CPU, the system must frequently switch between threads, impacting overall performance.

Joinable threads→ it's the default state in POSIX threads, necessitate explicit synchronization via **pthread_join()**. This function enables one thread to wait for another to finish its execution and release resources. Such synchronization is particularly valuable in scenarios where it's essential to obtain the results or ascertain the completion status of a thread.

Threads within the same process operate within a shared memory space, whereas processes have their own distinct memory spaces. Unlike processes, threads are not entirely independent entities; they share their code section, data section, and operating system resources, such as open files and signals, with other threads. However, similar to processes, each thread possesses its own program counter (PC), register set, and stack space.

In this project we will calculate the average BMI (Body Mass Index) for the attached dataset in three ways:

1- Naive approach, a program that does not use any child processes or threads.

2- Multiprocessing approach: a program that uses multiple child processes running in parallel. Try different numbers of child processes and compare the outcome.

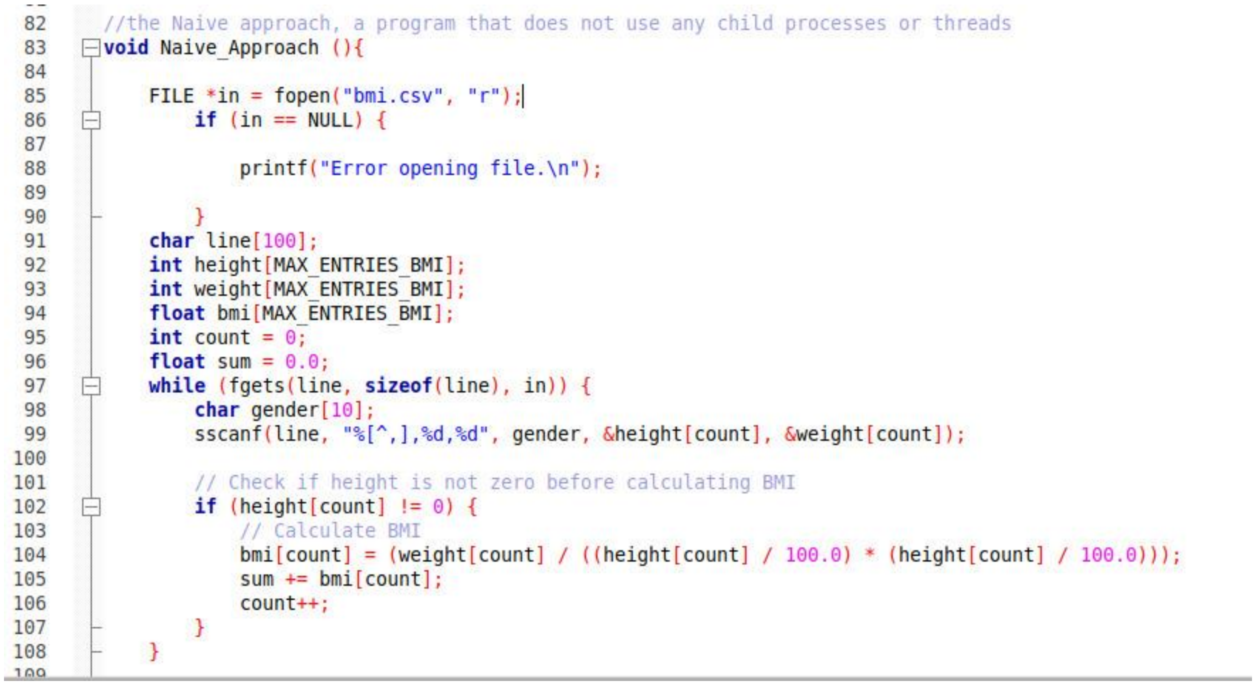
3- Multithreading approach: a program that uses multiple joinable threads running in parallel. Try different numbers of threads and compare the outcome.

And calculate the execution time and performance for each case then compare it in all cases.

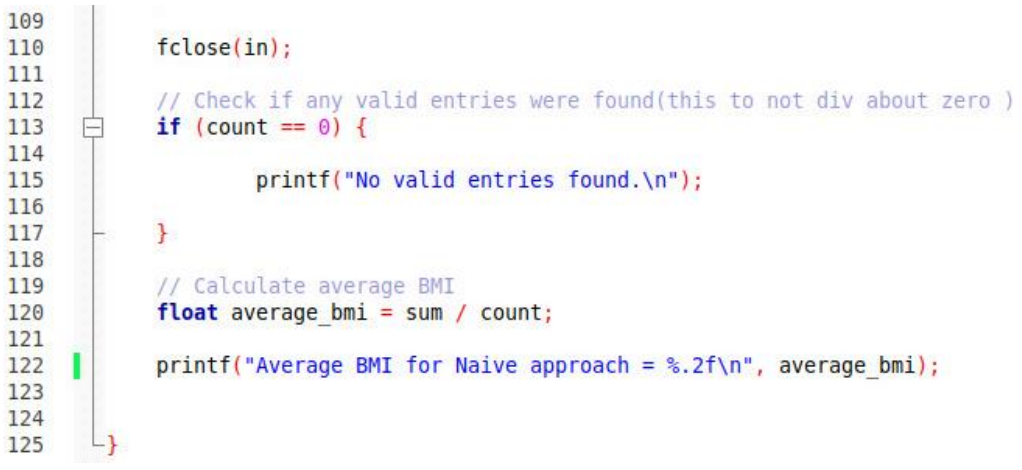
Discussion of codes:

1. Naive approach:

1.1 Code:

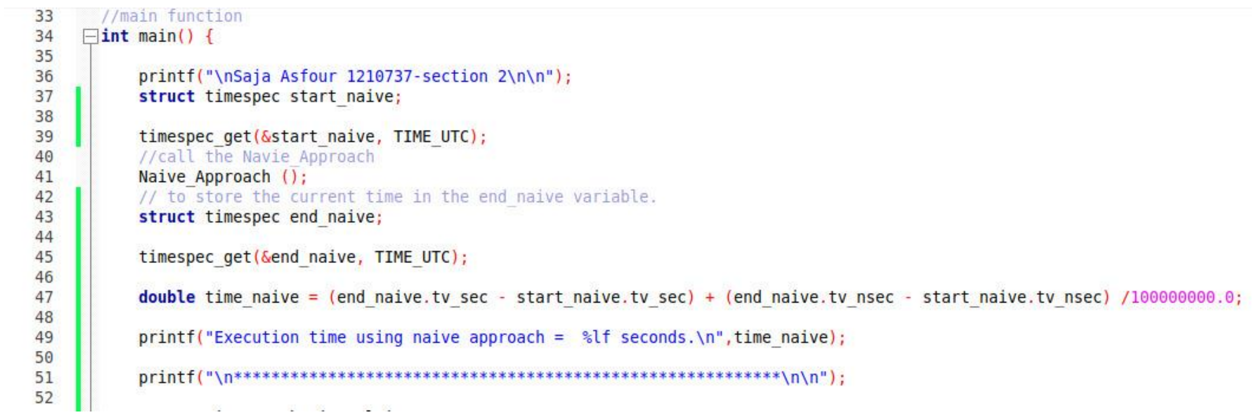


```c
//the Naive approach, a program that does not use any child processes or threads
void Naive_Approach (){

    FILE *in = fopen("bmi.csv", "r");
        if (in == NULL) {

            printf("Error opening file.\n");

        }
    char line[100];
    int height[MAX_ENTRIES_BMI];
    int weight[MAX_ENTRIES_BMI];
    float bmi[MAX_ENTRIES_BMI];
    int count = 0;
    float sum = 0.0;
    while (fgets(line, sizeof(line), in)) {
        char gender[10];
        sscanf(line, "%[^,],%d,%d", gender, &height[count], &weight[count]);

        // Check if height is not zero before calculating BMI
        if (height[count] != 0) {
            // Calculate BMI
            bmi[count] = (weight[count] / ((height[count] / 100.0) * (height[count] / 100.0)));
            sum += bmi[count];
            count++;
        }
    }
```



```c

    fclose(in);

    // Check if any valid entries were found(this to not div about zero )
    if (count == 0) {

            printf("No valid entries found.\n");

    }

    // Calculate average BMI
    float average_bmi = sum / count;

    printf("Average BMI for Naive approach = %.2f\n", average_bmi);


}
```

Figure 1 : Naive approach code



```
//main function
int main() {

    printf("\nSaja Asfour 1210737-section 2\n\n");
    struct timespec start_naive;

    timespec_get(&start_naive, TIME_UTC);
    //call the Navie Approach
    Naive_Approach ();
    // to store the current time in the end_naive variable.
    struct timespec end_naive;

    timespec_get(&end_naive, TIME_UTC);

    double time_naive = (end_naive.tv_sec - start_naive.tv_sec) + (end_naive.tv_nsec - start_naive.tv_nsec) /100000000.0;

    printf("Execution time using naive approach =  %lf seconds.\n",time_naive);

    printf("\n************************************************************\n\n");
```

Figure2 : Naive approach in the main part code

1.2 Discussion of Naïve approach code:

The Naïve approach is a simple method used to solve problems. It is basic and straightforward. In the Naïve approach code , I first read the datasheet from file and count the number of BMI after find it for each line of the code by this equation: BMI=$\frac{weight}{height^2}$ which height must be in meter , and while reading I skip the string part in the bmi.csv file to calculate the average of BMI correctly . after I finish reading from file I calculate the average of BMI by this equation→ average BMI = $\frac{sum\ of\ BMI}{count}$ then print this average to screen.
In the main function I found the execution time for Naïve approach → Time execution is calculated by using a function (timespec_get (& struct timespec variable, TIME_UTC) which provides the real elapsed time or the wall time and provides a nanosecond precision.

In the above code→ start_naive.tev_sec provides the current time in seconds at the start of the code, start_naive.tev_nsec provides the nanosecond part. The time_naive is calculated by taking the difference between the end and beginning and it is measured in seconds.

The use of a VmWare and several additional factors lead the execution times to vary with each run.so I make many run and get the average from them to be the execution time.

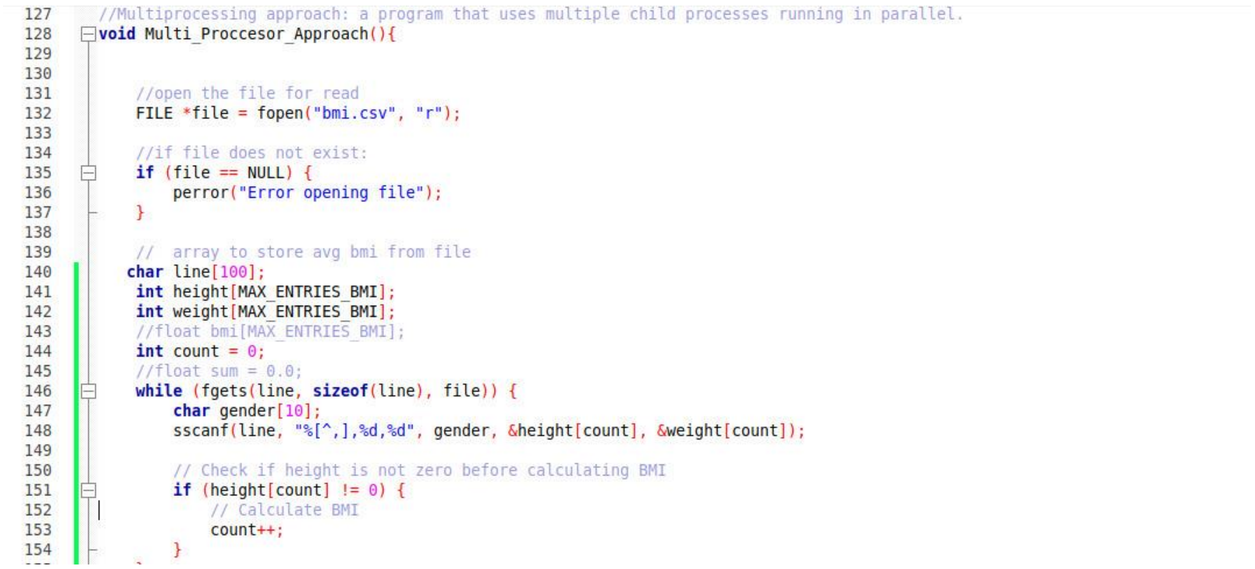
Table 1: Naive approach results

Run 1	Run2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Avg
0.003842	0.000570	0.000479	0.000429	0.000652	0.000421	0.001341	0.001074	0.000621

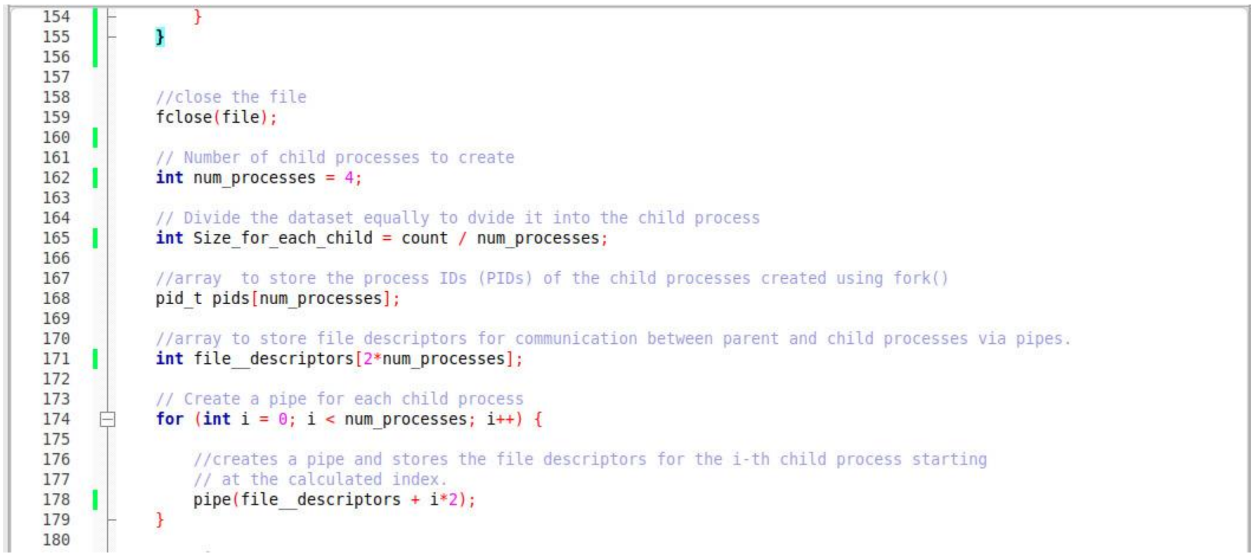
Performance= $\frac{1}{Execuiton\ Time}$ = $\frac{1}{0.003579}$ = 1610.306

2. Multiprocessor approach:

2.1 Code :

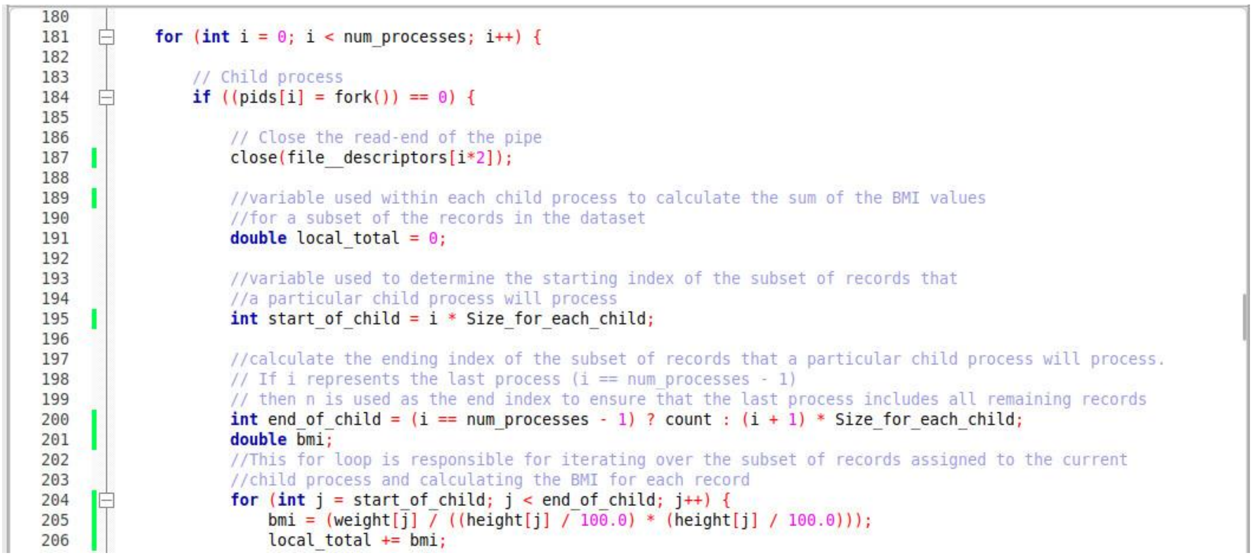


```c
//Multiprocessing approach: a program that uses multiple child processes running in parallel.
void Multi_Proccesor_Approach(){


    //open the file for read
    FILE *file = fopen("bmi.csv", "r");

    //if file does not exist:
    if (file == NULL) {
        perror("Error opening file");
    }

    //  array to store avg bmi from file
   char line[100];
    int height[MAX_ENTRIES_BMI];
    int weight[MAX_ENTRIES_BMI];
    //float bmi[MAX_ENTRIES_BMI];
    int count = 0;
    //float sum = 0.0;
    while (fgets(line, sizeof(line), file)) {
        char gender[10];
        sscanf(line, "%[^,],%d,%d", gender, &height[count], &weight[count]);

        // Check if height is not zero before calculating BMI
        if (height[count] != 0) {
            // Calculate BMI
            count++;
        }
```

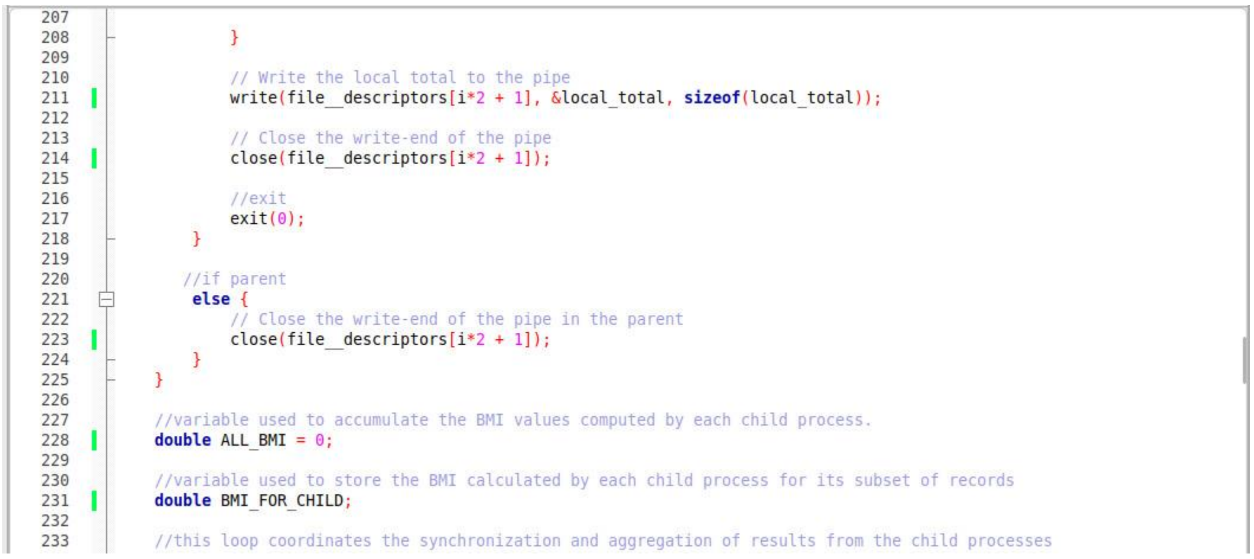


```c
        }
    }


    //close the file
    fclose(file);

    // Number of child processes to create
    int num_processes = 4;

    // Divide the dataset equally to dvide it into the child process
    int Size_for_each_child = count / num_processes;

    //array  to store the process IDs (PIDs) of the child processes created using fork()
    pid_t pids[num_processes];

    //array to store file descriptors for communication between parent and child processes via pipes.
    int file__descriptors[2*num_processes];

    // Create a pipe for each child process
    for (int i = 0; i < num_processes; i++) {

        //creates a pipe and stores the file descriptors for the i-th child process starting
        // at the calculated index.
        pipe(file__descriptors + i*2);
    }
```

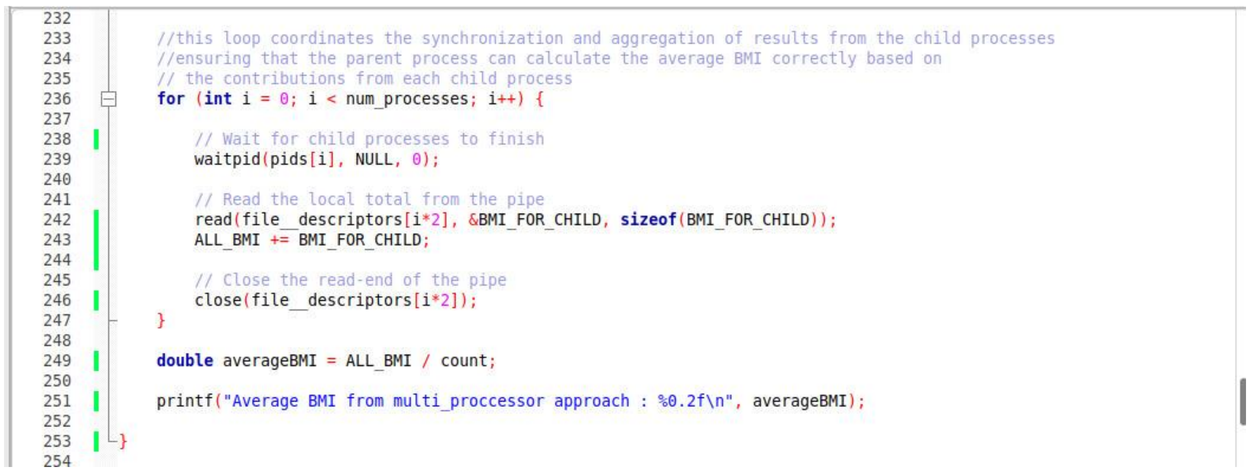
Figure 3: Multiprocessor Approach code part1



```c

    for (int i = 0; i < num_processes; i++) {

        // Child process
        if ((pids[i] = fork()) == 0) {

            // Close the read-end of the pipe
            close(file__descriptors[i*2]);

            //variable used within each child process to calculate the sum of the BMI values
            //for a subset of the records in the dataset
            double local_total = 0;

            //variable used to determine the starting index of the subset of records that
            //a particular child process will process
            int start_of_child = i * Size_for_each_child;

            //calculate the ending index of the subset of records that a particular child process will process.
            // If i represents the last process (i == num_processes - 1)
            // then n is used as the end index to ensure that the last process includes all remaining records
            int end_of_child = (i == num_processes - 1) ? count : (i + 1) * Size_for_each_child;
            double bmi;
            //This for loop is responsible for iterating over the subset of records assigned to the current
            //child process and calculating the BMI for each record
            for (int j = start_of_child; j < end_of_child; j++) {
                bmi = (weight[j] / ((height[j] / 100.0) * (height[j] / 100.0)));
                local_total += bmi;
```

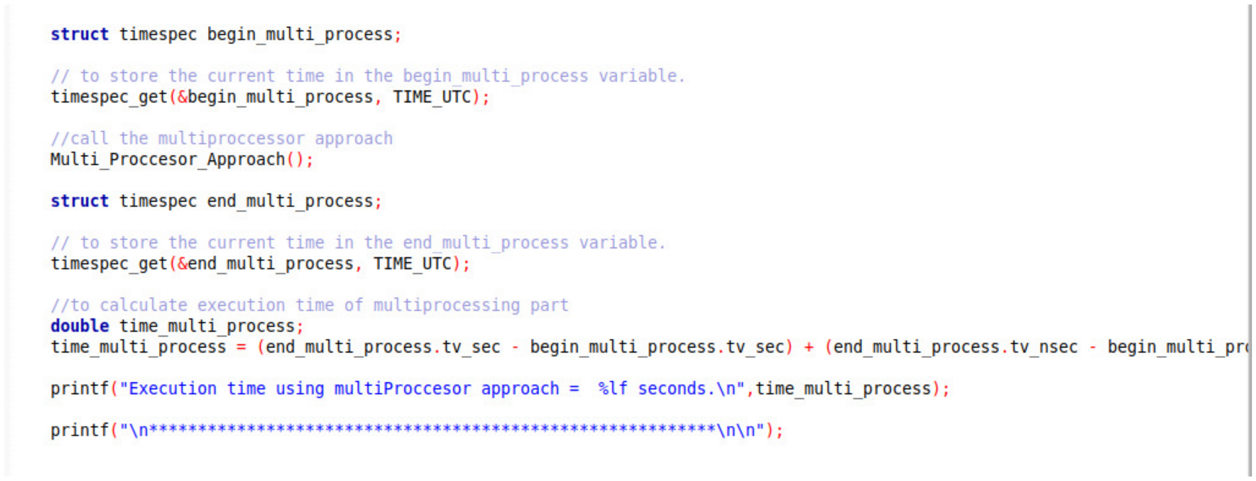


```c

            }

            // Write the local total to the pipe
            write(file__descriptors[i*2 + 1], &local_total, sizeof(local_total));

            // Close the write-end of the pipe
            close(file__descriptors[i*2 + 1]);

            //exit
            exit(0);
        }

       //if parent
        else {
            // Close the write-end of the pipe in the parent
            close(file__descriptors[i*2 + 1]);
        }
    }

    //variable used to accumulate the BMI values computed by each child process.
    double ALL_BMI = 0;

    //variable used to store the BMI calculated by each child process for its subset of records
    double BMI_FOR_CHILD;

    //this loop coordinates the synchronization and aggregation of results from the child processes
```



```c

    //this loop coordinates the synchronization and aggregation of results from the child processes
    //ensuring that the parent process can calculate the average BMI correctly based on
    // the contributions from each child process
    for (int i = 0; i < num_processes; i++) {

        // Wait for child processes to finish
        waitpid(pids[i], NULL, 0);

        // Read the local total from the pipe
        read(file__descriptors[i*2], &BMI_FOR_CHILD, sizeof(BMI_FOR_CHILD));
        ALL_BMI += BMI_FOR_CHILD;

        // Close the read-end of the pipe
        close(file__descriptors[i*2]);
    }

    double averageBMI = ALL_BMI / count;

    printf("Average BMI from multi_proccessor approach : %0.2f\n", averageBMI);

}

```

Figure 4: Multiprocessor approach code part 2



```
struct timespec begin_multi_process;

// to store the current time in the begin_multi_process variable.
timespec_get(&begin_multi_process, TIME_UTC);

//call the multiproccessor approach
Multi_Proccesor_Approach();

struct timespec end_multi_process;

// to store the current time in the end_multi_process variable.
timespec_get(&end_multi_process, TIME_UTC);

//to calculate execution time of multiprocessing part
double time_multi_process;
time_multi_process = (end_multi_process.tv_sec - begin_multi_process.tv_sec) + (end_multi_process.tv_nsec - begin_multi_pr

printf("Execution time using multiProccesor approach =  %lf seconds.\n",time_multi_process);

printf("\n*********************************************************\n\n");
```

Figure5: Multiprocessor approach code in the main function

2.2 Multiprocessor approach code discussion :

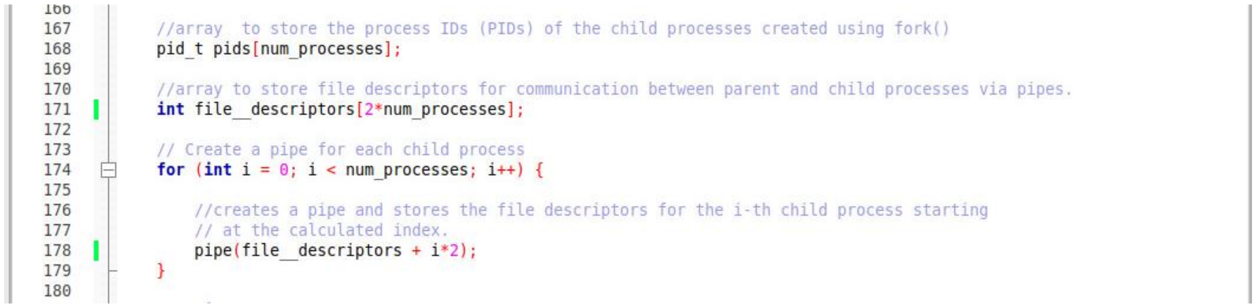
This approach allows the workload to be distributed across multiple processes, enabling parallel execution and leveraging the available CPU cores for improved performance.

First I read datasheet from file in the same way I read in Naïve approach above. But in each line I just count the number of BMI in the file to use it later.

Then I define a variable with name num_processes to set the number of CPUs that do the work.

Then I dived the number of BMI to the number of CPUs to dived the work between them and store the result in variable called segment_size. The dataset is divided equally among the child processes. Each child process calculates BMI for a subset of records.

In the code below I created a pipes for each child process, to perform intercrosses communication:



```
    //array  to store the process IDs (PIDs) of the child processes created using fork()
    pid_t pids[num_processes];

    //array to store file descriptors for communication between parent and child processes via pipes.
    int file__descriptors[2*num_processes];

    // Create a pipe for each child process
    for (int i = 0; i < num_processes; i++) {

        //creates a pipe and stores the file descriptors for the i-th child process starting
        // at the calculated index.
        pipe(file__descriptors + i*2);
    }
```

pid_t pids[num_processes] → is an array that holds the process IDs (PIDs) of the child processes created using fork().pid_t is a data type defined in the <sys/types.h> header file, which represents process IDs. It is used to store process IDs returned by system calls like fork ().

For example, if num_processes is 4, pid_t pids[4] declares an array capable of storing the PIDs of 4 child processes. After creating child processes using fork(), the PIDs of these child processes will be stored in elements pids[0], pids[1], pids[2], and pids[3], respectively.

int file_discriptor [2*num_processes] is typically used to hold file descriptors for communication between parent and child processes via pipes.

For example, if num_processes is 4, file_discriptor [8] declares an array capable of holding file descriptors for communication between 4 child processes and the parent process. Each child process will have two file descriptors associated with it: one for reading (file_discriptor [i*2]) and one for writing (fd[i*2+1]). These file descriptors are used to transfer data between the parent and child processes via pipes.

By calling pipe(file_discriptor + i*2) , the parent process creates a pipe and stores the file descriptors for communication with the current child process at the specified memory location within the fd array.

Then creates a specified number of child process using fork (). Fork system call creates a new process and return -1 if the creation failed, 0 if it is the child process and a positive value for the parent. This value represents the process id of the newly created process. Each child process is responsible for processing a portion of the dataset.

Pipes are used for communication between the parent process and child processes. Child processes write their local results (partial BMI) to the pipe, and the parent process reads these results to compute the total BMI.

Then the Child processes execute concurrently, processing their respective subsets of the dataset in parallel.

And finally, the parent process aggregates the partial BMI values obtained from each child process to compute the total BMI for the entire dataset.

In the main function I found the execution time for multiprocessor approach → Time execution is calculated by using a function (timespec_get (& struct timespec variable, TIME_UTC) which provides the real elapsed time or the wall time and provides a nanosecond precision.

In the above code→ begin_multi_process.tev_sec provides the current time in seconds at the start of the code, begin_multi_process.tev_nsec provides the nanosecond part. The time_multi_process is calculated by taking the difference between the end and beginning and it is measured in seconds.

The use of a VmWare and several additional factors lead the execution times to vary with each run.so I make many run and get the average from them to be the execution time.

Table 2: Multiprocessor approach results

Number of processor	Run 1	Run 2	Run 3	Run4	Run5	Run6	Run7	Avg
2	0.003818	0.003718	0.003563	0.002532	0.002639	0.002689	0.003866	0.003261
3	0.002234	0.002094	0.001691	0.002555	0.002148	0.002450	0.001966	0.002163
4	0.001508	0.001931	0.001629	0.001726	0.001964	0.001734	0.001025	0.001645
5	0.004959	0.004046	0.004510	0.005118	0.004808	0.004133	0.003922	0.004499

Performance for 2 processors = $\frac{1}{0.003261}$ = 306.654

Performance for 3 processors = $\frac{1}{0.002163}$ = 462.321

Performance for 4 processors = $\frac{1}{0.001645}$ = 607.903

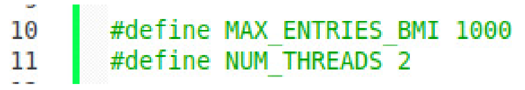
Performance for 5 processors = $\frac{1}{0.004499}$ = 222.271

The number of processes affects directly the execution time and the overall efficiency. When a greater number of processes is used, the code takes less time to execute the task. And hence it will have a higher performance. On the other hand, having too many processes can result in overhead, which can compete for resources and reduce productivity.

I used VmWare to run the code in Linux environment. And my system has only 4 CPUs. And that clarifies why 5 processes take greater time than 4 processes when I run the code. The optimal number of processes was 4 which equals the number of CPUs.

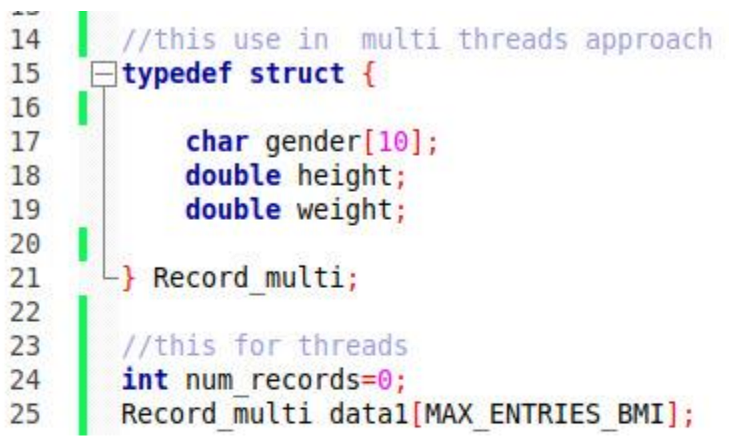
3. Multithreading approach:

3.1 Code and comment of it :



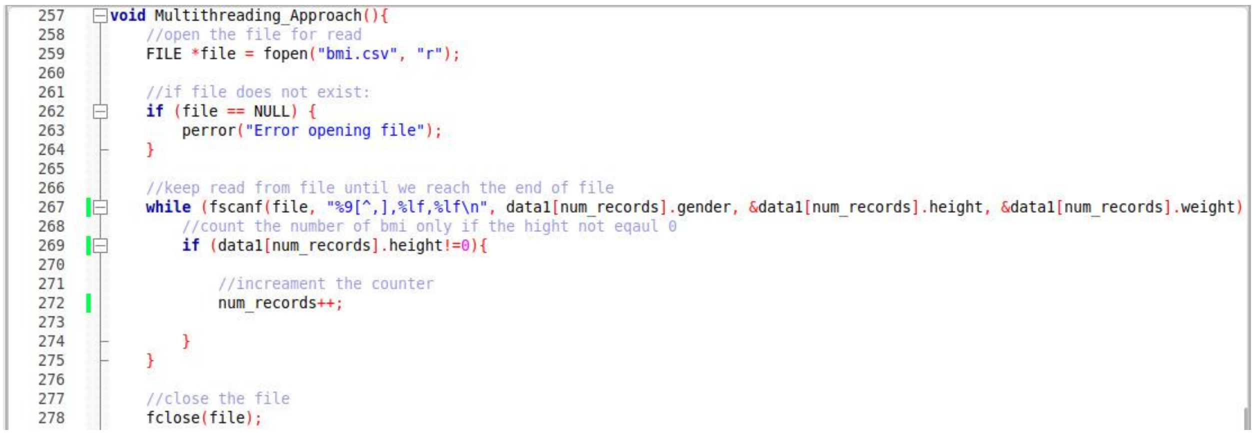
```
#define MAX_ENTRIES_BMI 1000
#define NUM_THREADS 2
```

Figure6 : Define variable to use it in Multithreading approach



```
//this use in  multi threads approach
typedef struct {

    char gender[10];
    double height;
    double weight;

} Record_multi;

//this for threads
int num_records=0;
Record_multi data1[MAX_ENTRIES_BMI];
```

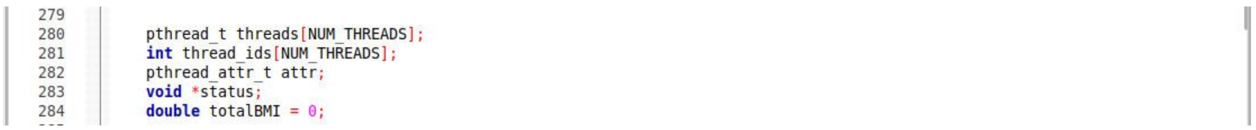
Figure7 : struct to store data from file + variable uses in multithreading approach



```
void Multithreading_Approach(){
    //open the file for read
    FILE *file = fopen("bmi.csv", "r");

    //if file does not exist:
    if (file == NULL) {
        perror("Error opening file");
    }

    //keep read from file until we reach the end of file
    while (fscanf(file, "%9[^,],%lf,%lf\n", data1[num_records].gender, &data1[num_records].height, &data1[num_records].weight)
        //count the number of bmi only if the hight not eqaul 0
        if (data1[num_records].height!=0){

            //increament the counter
            num_records++;

        }
    }

    //close the file
    fclose(file);
```

Figure 8: Multithreading approach code part1

In part1 above I read data from file and store it inside struct instead of array like in naïve and multiprocessor approaches and count the number of records BMI in the file then store it in variable called num_records.



```

    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    pthread_attr_t attr;
    void *status;
    double totalBMI = 0;
```

Figure9 : Multithreading approach code part2

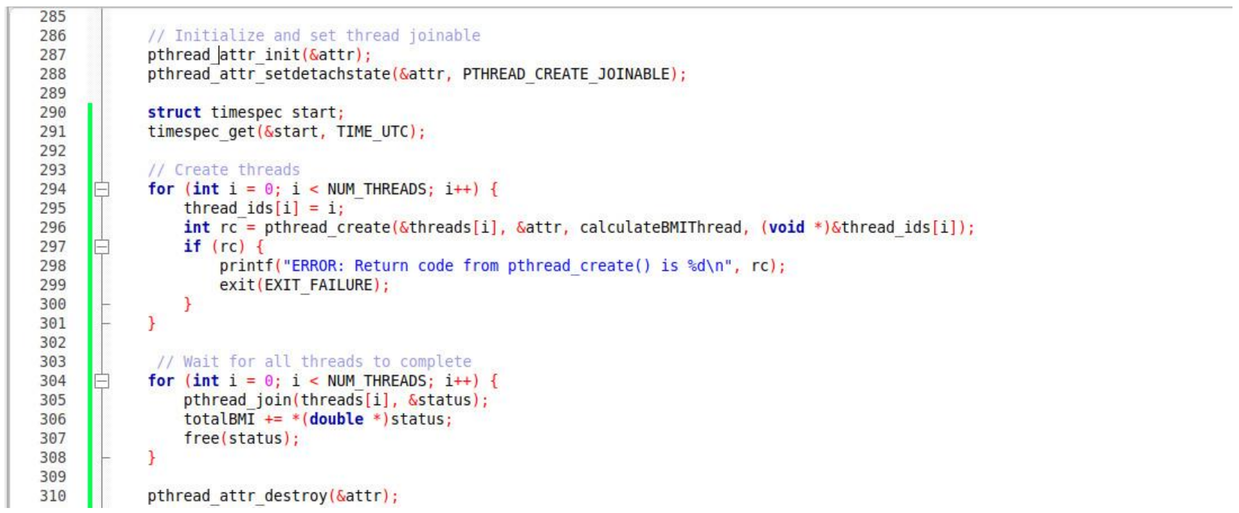
In part 2→ first I created an array threads of type pthread_t, which represents thread identifiers. NUM_THREADS is the number of threads want to create. This array will hold the thread identifiers for each thread created.

Then I declares an array thread_ids to store the IDs of the threads to be useful if I need to identify individual threads within my program.

Then I declares a thread attribute object attr that can be used to specify various attributes for the threads being created.

Then I declares a variable that will be used to store the exit status of the threads when they terminate. In pthreads, threads can return a status value when they finish executing, which can be retrieved by joining the threads.

Then I initializes a variable totalBMI to 0. This variable will be used to accumulate the total BMI calculated by all threads. It's initialized outside of the loop where threads are created, and it will be updated by each thread as they compute their partial BMI values.



```

    // Initialize and set thread joinable
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    struct timespec start;
    timespec_get(&start, TIME_UTC);

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        int rc = pthread_create(&threads[i], &attr, calculateBMIThread, (void *)&thread_ids[i]);
        if (rc) {
            printf("ERROR: Return code from pthread_create() is %d\n", rc);
            exit(EXIT_FAILURE);
        }
    }

     // Wait for all threads to complete
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], &status);
        totalBMI += *(double *)status;
        free(status);
    }

    pthread_attr_destroy(&attr);
```

Figure 10: Multithreading approach code part 3

In part3 → pthread_attr_init(&attr); : This function initializes the thread attribute object referenced by attr with default attribute values. This is necessary before setting any specific attributes using other functions.

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); : This function sets the detach state attribute of the thread attribute object attr. The PTHREAD_CREATE_JOINABLE constant indicates that the threads created with this attribute can be joined by other threads. When a thread is joinable, other threads can wait for it to finish its execution and retrieve its return value or status.

After that I start the timer to calculate execution time by the same way in naïve and multiprocessor approaches.

Then create for loop that creates multiple threads, each running the calculateBMIThread function (I will add it code later in my code) concurrently with potentially different arguments. This loop iterates over the number of threads to be created. It assigns a unique identifier to each thread. This identifier can be useful for distinguishing between threads, especially when dealing with thread-specific data or operations.

int rc = pthread_create(&threads[i], &attr, calculateBMIThread, (void *)&thread_ids[i]); : This line creates a new thread. Here's a notes about the parameters:

&threads[i] : A pointer to a pthread_t variable where the thread ID will be stored after creation.

&attr: A pointer to the thread attribute object specifying the attributes for the new thread.

calculateBMIThread: The function to be executed by the thread. This function must have the signature void *function (void *arg).

(void*)&thread_ids[i]: An argument to be passed to the thread function. In this case, it's a pointer to the thread identifier.

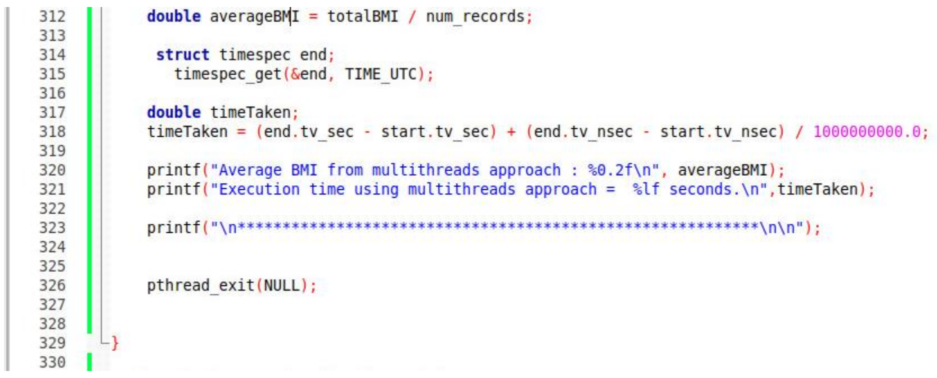
Then I create for loop to waits all threads to finish execution and accumulates their individual BMI calculation results into the totalBMI variable. And here some notes about it:

→pthread_join(threads[i], &status); : This function call blocks the current thread until the specified thread terminates. It waits for the thread with ID threads[i] to finish execution. The &status argument is a pointer to a location where the exit status of the joined thread will be stored.

→ totalBMI += *(double *)status; : After a thread completes, its exit status (which is a pointer to a double value) is retrieved from the status variable and added to the totalBMI variable. The (double *) status casts the void pointer status to a pointer to a double so that it can be dereferenced and its value added to totalBMI.

→ free (status); : Since memory was allocated dynamically to store the exit status of each thread, this line frees that memory to prevent memory leaks.

Then after this loop → pthread_attr_destroy(&attr);: This function call destroys the thread attribute object referenced by attr, releasing any resources associated with it. It should be called after the threads have been created and are no longer needed to clean up any resources allocated for the attribute object.



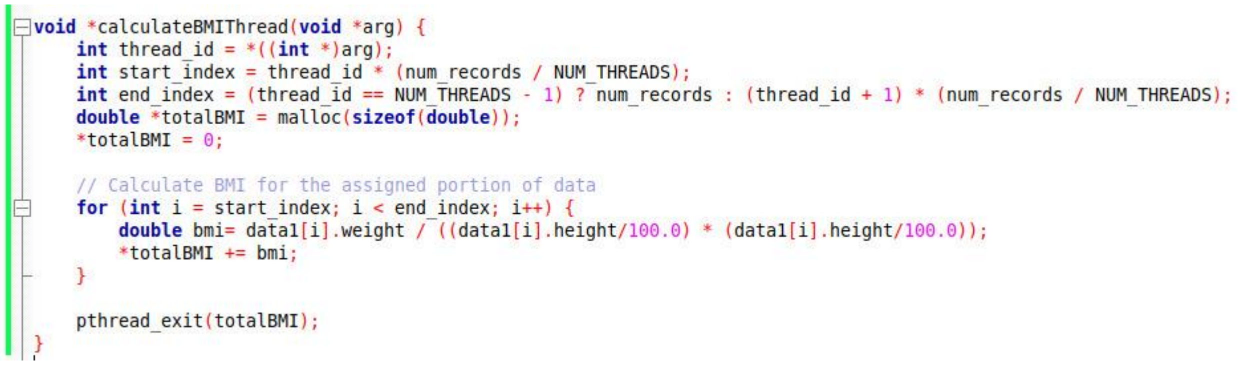
```c
    double averageBMI = totalBMI / num_records;

     struct timespec end;
       timespec_get(&end, TIME_UTC);

    double timeTaken;
    timeTaken = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1000000000.0;

    printf("Average BMI from multithreads approach : %0.2f\n", averageBMI);
    printf("Execution time using multithreads approach =  %lf seconds.\n",timeTaken);

    printf("\n************************************************************\n\n");


    pthread_exit(NULL);


}
```

Figure 11: Multithreading approach code part 4

Then I calculated the averageBMI by this equation → averageBMI= $\frac{totalBMI}{num\_records}$

And stop the timer then print both the execution time and the average result.

Finally, terminate the calling thread and return a value to the thread's joiner (if any) by pthread_exit(NULL).



```c
void *calculateBMIThread(void *arg) {
    int thread_id = *((int *)arg);
    int start_index = thread_id * (num_records / NUM_THREADS);
    int end_index = (thread_id == NUM_THREADS - 1) ? num_records : (thread_id + 1) * (num_records / NUM_THREADS);
    double *totalBMI = malloc(sizeof(double));
    *totalBMI = 0;

    // Calculate BMI for the assigned portion of data
    for (int i = start_index; i < end_index; i++) {
        double bmi= data1[i].weight / ((data1[i].height/100.0) * (data1[i].height/100.0));
        *totalBMI += bmi;
    }

    pthread_exit(totalBMI);
}
```

Figure 12 : Multithreading approach code part 5

in part 5→ this is function that responsible for computing the BMI for a portion of the dataset assigned to each thread and returning the total BMI calculated by the thread. And below some notes about it:

→ void *calculateBMIThread(void *arg): This function takes a void pointer arg as an argument and returns a void pointer. This signature is in compliance with the requirements for a thread function in pthreads.

→ int thread_id = *((int *)arg) ; : It extracts the thread ID from the arg parameter, which is passed as a void *. This ID is used to determine the portion of the data that the thread will process.

→ int start_index = thread_id * (num_records / NUM_THREADS); : Calculates the start index of the portion of the data to be processed by the current thread.

→ int end_index = (thread_id == NUM_THREADS - 1) ? num_records : (thread_id + 1) * (num_records / NUM_THREADS); : Calculates the end index of the portion of the data to be processed by the current thread. If it's the last thread, it processes the remaining records.

→ double *totalBMI = malloc(sizeof(double)); : Allocates memory to store the total BMI calculated by the thread. This memory is dynamically allocated to ensure that each thread has its own space for the total BMI.

→ then we have a loop to Calculates BMI for the assigned portion of data within the loop. The loop iterates over the records assigned to the thread based on its start and end indices.

- ➔ double bmi = data1[i].weight / ((data1[i].height / 100.0) * (data1[i].height / 100.0)); : Calculates BMI for each record using its weight and height.

→ pthread_exit(totalBMI); : Exits the thread and returns the total BMI calculated by the thread. The memory allocated for totalBMI is returned to the main thread for cleanup.

3.2: Measured execution time:

The use of a VmWare and several additional factors lead the execution times to vary with each run.so I make many run and get the average from them to be the execution time.

Table 3: Multithreading approach result

#of threads	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Avg
2	0.004801	0.003258	0.002456	0.002586	0.004089	0.004484	0.003363	0.004111	0.003337
3	0.002232	0.001372	0.001109	0.001185	0.002203	0.002373	0.002438	0.002194	0.001889
4	0.000981	0.000727	0.001012	0.000864	0.001670	0.001489	0.000952	0.001700	0.001174
5	0.005627	0.007951	0.003308	0.003929	0.003450	0.005805	0.003342	0.003900	0.004664

Performance for 2 joinable threads= $\frac{1}{0.003337}$ = 299.670

Performance for 3 joinable threads= $\frac{1}{0.001889}$ = 529.381

Performance for 4 joinable threads= $\frac{1}{0.001174}$ = 851.789

Performance for 5 joinable threads= $\frac{1}{0.004664}$ = 214.408

When comparing different number of threads (2, 3 and 4), it seems that larger number of threads leads to better performance and reduce execution time. However, this is not always correct when the number of threads exceeds the CPU numbers which is 4 in my Linux environment (I used VmWare). So the optimal number of joinable threads is 4 in my system.

4. Performance and executions time comparison:

Time execution is calculated for each part by using a function (timespec_get (& struct timespec variable, TIME_UTC) which provides the real elapsed time or the wall time and provides a nanosecond precision.

The use of a VmWare and several additional factors lead the execution times to vary with each run. I average them, and the results are displayed in the table below:

Table 4: All approaches result executions time and performance

Approach used	Execution time	Performance
Naïve approach	0.000621	1610.306
2 processors	0.003261	306.645
3 processors	0.002163	462.321
4 processors	0.001645	607.903
5 processors	0.004499	222.271
2 joinable threads	0.003337	299.670
3 joinable threads	0.001889	529.381
4 joinable threads	0.001174	851.789
5 joinable threads	0.004664	214.408

Putting the three ways in order of performance and throughput, we can say that a Naïve approach program performs better, followed by multithreading in second place, and finally, a multiprocessor approach longer execution time and worse performance.

This not happen every time, usually the multithreading or multiprocessors has the best performance and naïve approach has worst performance. But it happen to some reason some of it:

1-Small Dataset Size: For small datasets, the overhead of creating and managing multiple processes (as in the multiprocessing approach) can outweigh any potential parallelization benefits.The naive approach, which performs calculations sequentially, might be faster in such cases.

2-Overhead of Process Creation and Synchronization: Creating and synchronizing processes (forking, waiting, and communication) introduces overhead.If the actual computation time per record is small compared to this overhead, the naive approach might be faster.

3- I/O-Bound Tasks :If the task involves significant I/O operations (e.g., reading from/writing to files, network requests), the naive approach might be more efficient. Multiprocessing doesn't necessarily speed up I/O-bound tasks.

4-Uneven Workload Distribution: If the dataset isn't evenly divided among processes (due to uneven record sizes or other factors), some processes may finish early, leading to idle time.Load balancing is crucial for efficient multiprocessing.

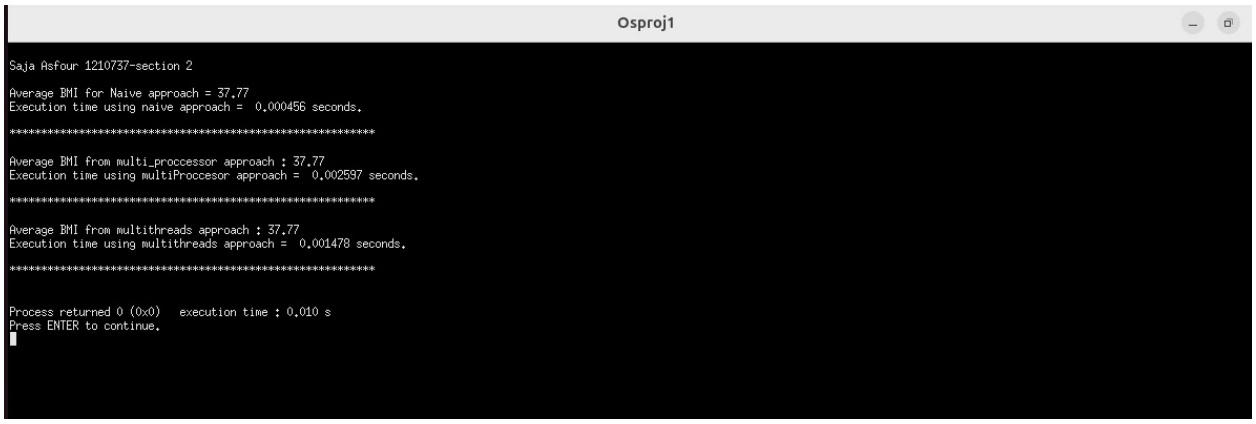
5-Data Dependencies: If the calculations have dependencies (e.g., one record's result depends on another), parallelization becomes challenging.The naive approach avoids these complexities.

6- CPU Cache Effects :Multiprocessing can lead to cache thrashing if different processes access the same memory locations.The naive approach (sequential) might benefit from better cache locality.

So, the choice between them depends on the specific problem, dataset size, system resources, and the nature of the workload. Profiling and experimentation are essential to determine the most efficient approach for your use case.

Multiprocessing and multithreading appear to be comparable in a number of aspects. They both make parallelism possible, which can enhance responsiveness and performance. They do, however, differ in other respects. Multiple threads sharing the same memory space but operating within a single process. This makes communication easier, but because shared resources are involved, careful synchronization is required. However, multiprocessing uses several processes, each of which has its own memory space.

This is an example of run when I have num_process=3 and NUM_THREADS=4:



```
Saja Asfour 1210737-section 2

Average BMI for Naive approach = 37.77
Execution time using naive approach =  0.000456 seconds.

**********************************************************

Average BMI from multi_proccessor approach : 37.77
Execution time using multiProccesor approach =  0.002597 seconds.

**********************************************************

Average BMI from multithreads approach : 37.77
Execution time using multithreads approach =  0.001478 seconds.

**********************************************************


Process returned 0 (0x0)   execution time : 0.010 s
Press ENTER to continue.
```

Figure 13 : Run Example

5. An analysis according to Amdahl’s law:

The serial part of my code is the part that use to read data from file and the part use to calculate the average BMI (final calculate). So to find the percentage of the serial part we first find the execution time of it (in the code) and then find the execution time for the whole code and divide it about the execution time of serial part multiply by 100%.

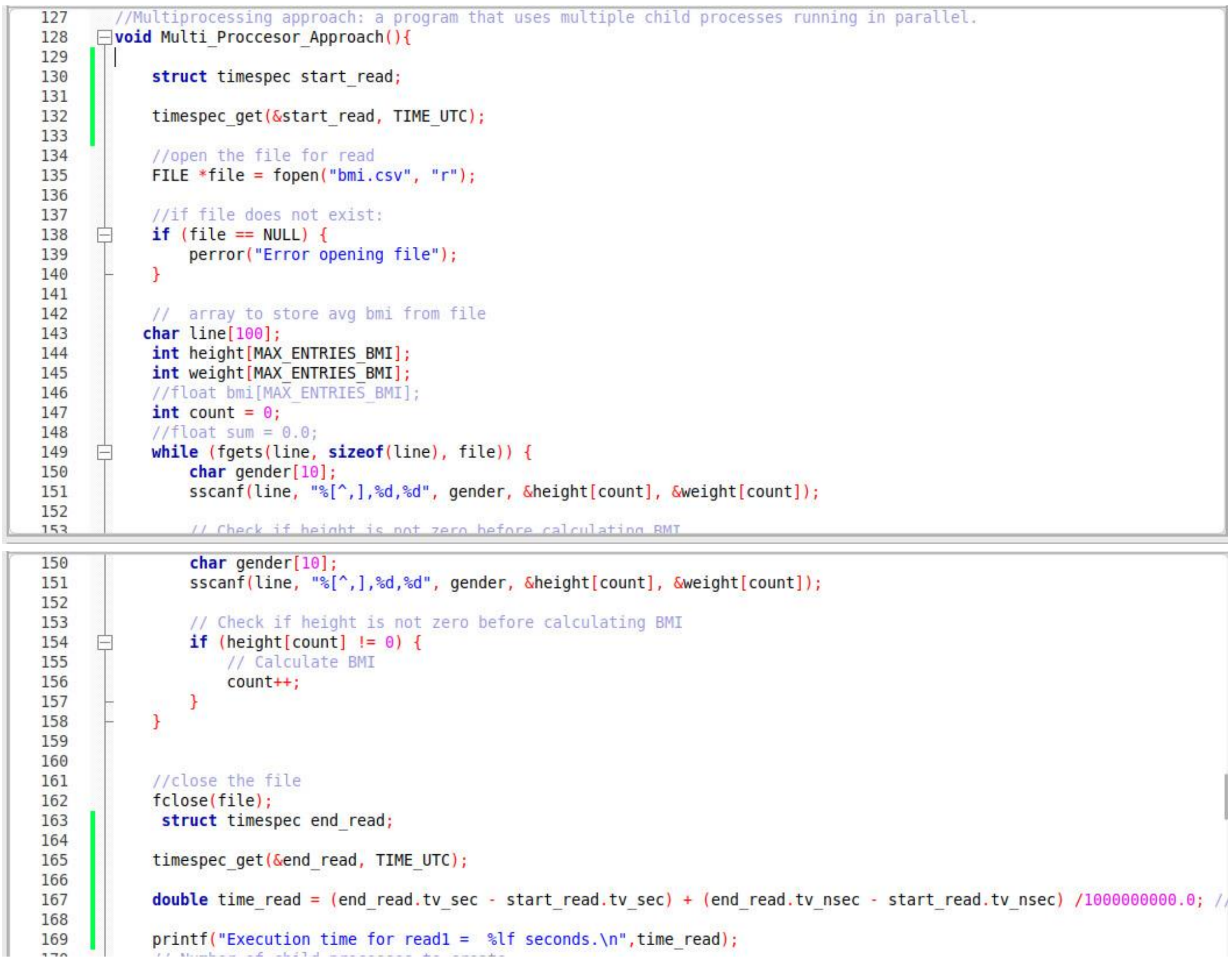
5.1 Naïve approach:

For this approach we don’t have any parallel part in it, all is serial so the percentage of serial is 100%

Speed up = $\frac{\mathbf{1}}{s+\frac{1-s}{N}} = \frac{\mathbf{1}}{\mathbf{1+0}}$ = 1

5.2 Multiprocessors approach:

First I found the execution time for read data from file which have a small different in each read so I take 5 run and average them.



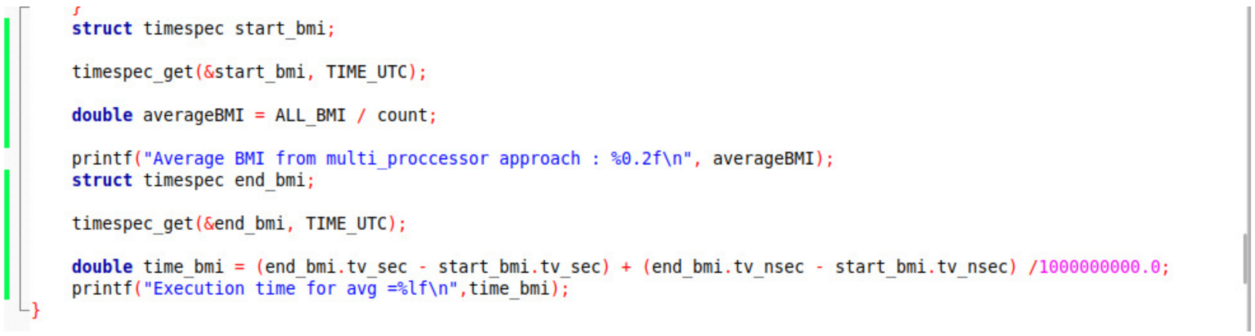
```
//Multiprocessing approach: a program that uses multiple child processes running in parallel.
void Multi_Proccesor_Approach(){

    struct timespec start_read;

    timespec_get(&start_read, TIME_UTC);

    //open the file for read
    FILE *file = fopen("bmi.csv", "r");

    //if file does not exist:
    if (file == NULL) {
        perror("Error opening file");
    }

    //  array to store avg bmi from file
   char line[100];
    int height[MAX_ENTRIES_BMI];
    int weight[MAX_ENTRIES_BMI];
    //float bmi[MAX_ENTRIES_BMI];
    int count = 0;
    //float sum = 0.0;
    while (fgets(line, sizeof(line), file)) {
        char gender[10];
        sscanf(line, "%[^,],%d,%d", gender, &height[count], &weight[count]);

        // Check if height is not zero before calculating BMI
        if (height[count] != 0) {
            // Calculate BMI
            count++;
        }
    }


    //close the file
    fclose(file);
     struct timespec end_read;

    timespec_get(&end_read, TIME_UTC);

    double time_read = (end_read.tv_sec - start_read.tv_sec) + (end_read.tv_nsec - start_read.tv_nsec) /1000000000.0; /

    printf("Execution time for read1 =  %lf seconds.\n",time_read);
```

Figure 14: Code to find the execution time for read file in multiprocessor approach

Table 5: Read file in multiprocessor approach execution time

Run1	Run 2	Run 3	Run 4	Run 5	Avg
0.000280	0.000229	0.000241	0.000227	0.000241	0.000244

Then I found the execution time for calculated the average BMI in the same way (just for the Arithmetic mean).



```
    struct timespec start_bmi;

    timespec_get(&start_bmi, TIME_UTC);

    double averageBMI = ALL_BMI / count;

    printf("Average BMI from multi_proccessor approach : %0.2f\n", averageBMI);
    struct timespec end_bmi;

    timespec_get(&end_bmi, TIME_UTC);

    double time_bmi = (end_bmi.tv_sec - start_bmi.tv_sec) + (end_bmi.tv_nsec - start_bmi.tv_nsec) /1000000000.0;
    printf("Execution time for avg =%lf\n",time_bmi);
}
```

Figure 15 : Code to find the execution time for calculate average BMI in multiprocessors

Table 6: calculate average BMI execution time in multiprocessor approach

Run1	Run2	Run3	Run4	Run5	Avg
0.000058	0.000051	0.000054	0.000066	0.000064	0.000059

Serial percentage = $\frac{read\ file\ execution\ time + average\ bmi\ execution\ time}{execution\ time\ for\ the\ function}$ x 100%

Speed up = $\frac{1}{s+\frac{1-s}{N}}$

Read file execution time + average bmi execution time = 0.000244+ 0.000059=0.000303 sec

When we have 2 processor:

Serial percentage = $\frac{0.000303}{0.003261}$ x 100%=9.29%

Speed up = $\frac{1}{0.0929+\frac{0.9071}{2}}$ =1.82

When we have 3 processor:

Serial percentage = $\frac{0.000303}{0.002163}$ x 100%=14%

Speed up = $\frac{1}{0.14+\frac{0.86}{3}}$ =2.34

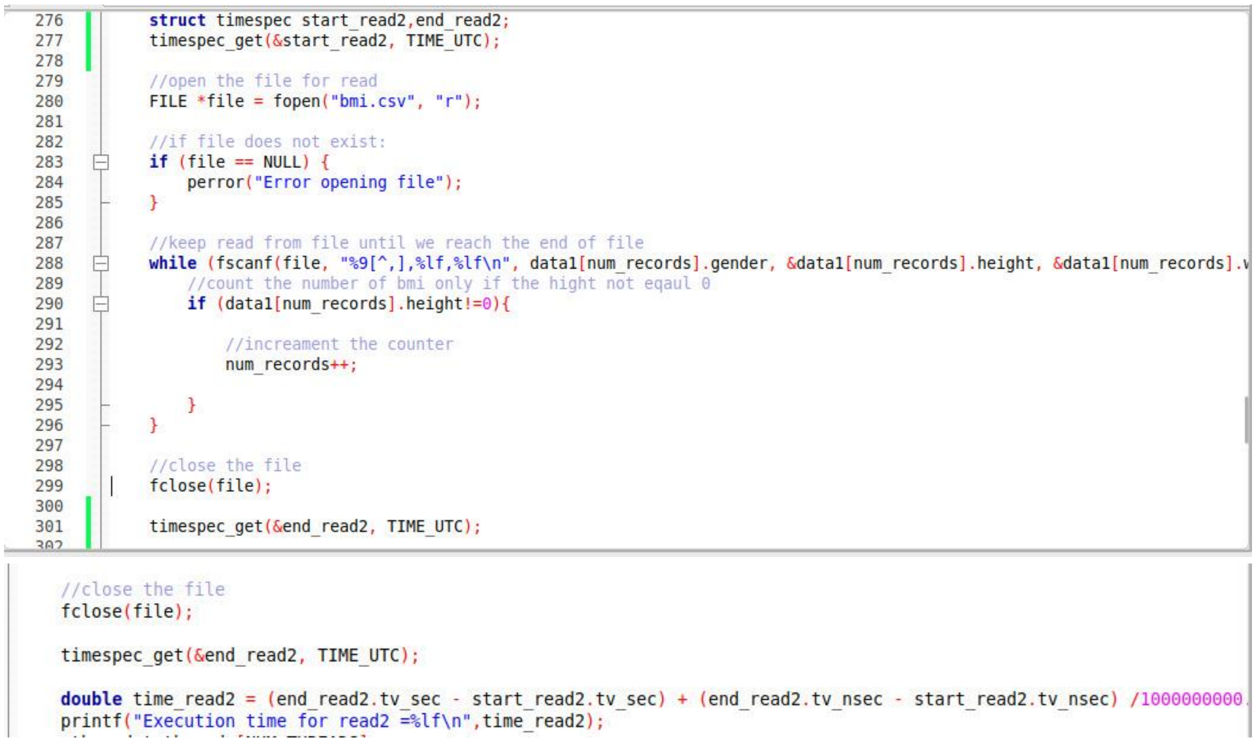
When we have 4 processor:

Serial percentage = $\frac{0.000303}{0.001645}$ x 100%=18.42%

Speed up = $\frac{1}{0.1842+\frac{0.8185}{4}}$ =2.57

5.3 Multithreading approach:

First I found the execution time for read data from file which have a small different in each read so I take 5 run and average them.



```
    struct timespec start_read2,end_read2;
    timespec_get(&start_read2, TIME_UTC);

    //open the file for read
    FILE *file = fopen("bmi.csv", "r");

    //if file does not exist:
    if (file == NULL) {
        perror("Error opening file");
    }

    //keep read from file until we reach the end of file
    while (fscanf(file, "%9[^,],%lf,%lf\n", data1[num_records].gender, &data1[num_records].height, &data1[num_records].w
        //count the number of bmi only if the hight not eqaul 0
        if (data1[num_records].height!=0){

            //increament the counter
            num_records++;

        }
    }

    //close the file
    fclose(file);

    timespec_get(&end_read2, TIME_UTC);
```

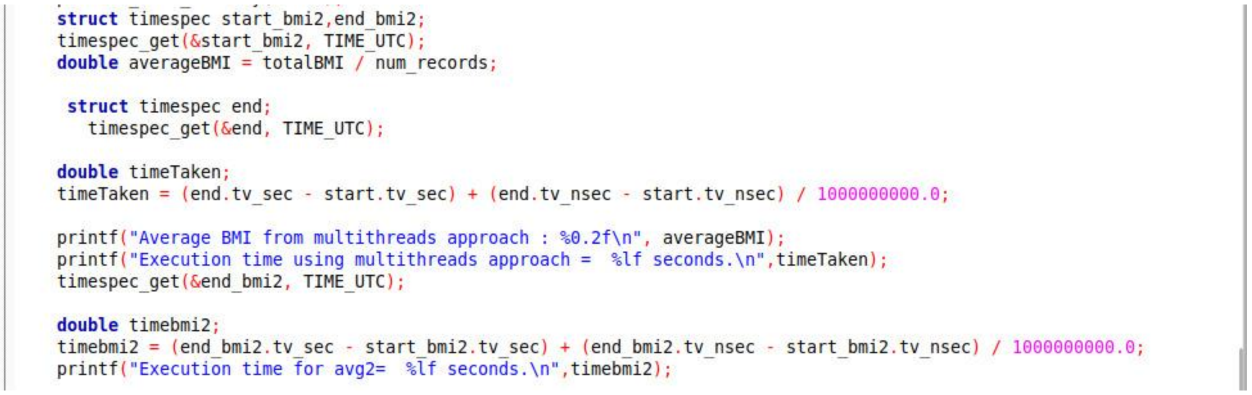
```
    //close the file
    fclose(file);

    timespec_get(&end_read2, TIME_UTC);

    double time_read2 = (end_read2.tv_sec - start_read2.tv_sec) + (end_read2.tv_nsec - start_read2.tv_nsec) /1000000000.
    printf("Execution time for read2 =%lf\n",time_read2);
```

Figure16 : Code for determine the execution time for read file in multithreading approach

Table 7: execution time result for read file in multithreading approach

Run1	Run 2	Run 3	Run 4	Run 5	Avg
0.000393	0.000387	0.000375	0.000443	0.000222	0.000364

Then I found the execution time for calculated the average BMI in the same way (just for the Arithmetic mean).



```
struct timespec start_bmi2,end_bmi2;
timespec_get(&start_bmi2, TIME_UTC);
double averageBMI = totalBMI / num_records;

 struct timespec end;
   timespec_get(&end, TIME_UTC);

double timeTaken;
timeTaken = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1000000000.0;

printf("Average BMI from multithreads approach : %0.2f\n", averageBMI);
printf("Execution time using multithreads approach =  %lf seconds.\n",timeTaken);
timespec_get(&end_bmi2, TIME_UTC);

double timebmi2;
timebmi2 = (end_bmi2.tv_sec - start_bmi2.tv_sec) + (end_bmi2.tv_nsec - start_bmi2.tv_nsec) / 1000000000.0;
printf("Execution time for avg2=  %lf seconds.\n",timebmi2);
```

Figure 17: Code to determine the execution time for average BMI in multithreading

Table 8 : result of execution time for calculate average BMI in multithreading approach

Run1	Run 2	Run 3	Run 4	Run 5	Avg
0.000063	0.000065	0.000124	0.000119	0.000061	0.000086

Serial percentage = $\frac{read\ file\ execution\ time + average\ bmi\ execution\ time}{execution\ time\ for\ the\ function}$ x 100%

Speed up = $\frac{1}{s+\frac{1-s}{N}}$

Read file execution time + average bmi execution time = 0.000364+ 0.000086=0.000450sec

When we have 2 joinable threads:

Serial percentage = $\frac{0.000450}{0.003337}$ x 100%=13.49%

Speed up = $\frac{1}{0.1349+\frac{0.8651}{2}}$ =1.76

When we have 3 joinable threads:

Serial percentage = $\frac{0.000450}{0.001889}$ x 100%=23.82%

Speed up = $\frac{1}{0.2382+\frac{0.7618}{3}}$ =1.86

When we have 4 joinable threads:

Serial percentage = $\frac{0.000450}{0.001174}$ x 100%=38.33%

Speed up = $\frac{1}{0.3833+\frac{0.6167}{4}}$ =2.03

The maximum speed up equal to 2.57 which is when we have 4 cores

Conclusion:

To summarize, this study compares various strategies for handling small computational tasks. The naive approach, while straightforward, is more efficient for smaller tasks to some reason I provided above. The child processes approach work by distributing the task across multiple child processes, with throughput increasing as the number of processes grows. And joinable threads, which allow for effective synchronization and result collection, generally exceeds child processes in efficiency, with throughput also rising with more threads. This analysis highlights the importance of carefully selecting a method based on task size, required synchronization, and efficiency.