Electrical and Computer Engineering Department

ENCS3390 Operating Systems

second Semester 2023/2024

Name : Mariam Turk

ID : 1211115

Instructor : Dr . Abel Salam Sayyad

Section : 1

• Abstract

This program calculates the average BMI (Body Mass Index) for a given dataset and compares three different approaches: the naive approach, multiprocessing approach, and multithreading approach,and measures the execution time for each approach to compare their performance. By analyzing the execution time, we can observe the impact of parallel processing on the overall efficiency of calculating the average BMI.

Contents

Table of figure

• Theory

- BMI (Body Mass Index)

BMI is a commonly used measurement to assess a person's body weight relative to their height. It is intended to provide an indication of a person's leanness or corpulence and is often used as a general indicator of whether an individual has a healthy body weight. Although BMI is not a direct measure of body fat percentage, it is a convenient and widely accepted tool for initial screening purposes. [1]

The following table outlines the categories commonly used to interpret BMI values:

BMI Categories:

- Underweight: BMI less than 18.5
- Normal weight: BMI between 18.5 and 24.9
- Overweight: BMI between 25 and 29.9
- Obesity - Class I: BMI between 30 and 34.9
- Obesity - Class II: BMI between 35 and 39.9
- Obesity - Class III: BMI 40 or higher

It is important to note that these categories may vary slightly depending on factors such as region and age. In some cases, additional subcategories may be used to describe the severity of underweight or obesity, such as "severely underweight" or "very severely obese."

- Naive approach

The naive approach is a straightforward and simplistic method commonly employed to address problems. It emphasizes fundamental assumptions and simplicity while disregarding intricate techniques. This report delves into the significance, relevance, and practical applications of the naive approach. It also includes a basic example and a graphical representation to illustrate its usage. [2]

It is important because it serves as a foundation for beginners. It helps them understand complex problems and encourages creative thinking. It provides a starting point and prepares individuals for more advanced techniques in problem-solving. [2]

- Multiprocessing approach

Multiprocessing is a way for computers to work faster by having multiple processors handle different parts of a program at the same time. Instead of relying on just one processor, a computer with multiprocessing can use two or more processors to process instructions simultaneously. [3]

The main advantage of multiprocessing is speed. With multiple processors, a computer can handle larger amounts of information more efficiently. Each processor is assigned specific tasks, allowing them to work on different sets of instructions simultaneously. For example, one processor may handle memory storage, another may handle data communications, and yet another may handle arithmetic functions. This division of tasks helps the computer process instructions faster. [3]

Multiprocessing was first used in large computers called mainframes before becoming more common in personal computers as the cost of including multiple processors decreased. [3]

- Multithreading approach

Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer. Multithreading can also handle multiple requests from the same user. [4]

It is a technique that enables multiple threads from different programs to be executed simultaneously on a single microprocessor. Although the processor can only handle one instruction at a time, the threads are executed rapidly, creating the illusion of concurrent execution. This improves performance by allowing programs to execute multiple tasks concurrently. However, careful attention is required from programmers to avoid issues like race conditions and deadlocks, which can arise when multiple threads access shared data or wait for each other. [4]

1- The API and functions of Multiprocessing and Multithreading

- Multiprocessing

```
#include <sys/types.h>

#include <sys/wait.h>
```

The **<sys/types.h>** header file provides the definition for **pid_t**, and the **<unistd.h>** header file includes the necessary system calls and function prototypes, including **fork( ).**

```
pid_t childPid;

int status;
```

pid_t childPid : This variable of type **pid_t** is used to store the process ID (PID) of the child process .

int status : This variable is used to store the termination status of the child process.

API’s :

```
childPid = fork();

      if (childPid == 0) // Child process

      {

        calculateBMI(start, end, gender, height, weight, BMI);

         exit(0);

      }
```

```
    else if (childPid < 0) // Fork failed
    {
        cout << "Fork failed." << endl;
        exit(1);
    }
  }

// Wait for all child processes to finish
  while ((childPid = wait(&status)) > 0)
  {
    // Do nothing
  }
```

fork( ) : This system call is used to create child processes. It is used inside a loop to create multiple child processes for parallel execution of the BMI calculation.

wait( ) : This system call is used to wait for child processes to finish. The parent process waits until all child processes have exited before proceeding.

These APIs are essential for measuring the execution time and achieving parallel processing using child processes

- Multithreading

```
#include <thread>
```

The **<thread>** header in C++ provides functionality for working with threads, allowing for concurrent execution and parallelism.

```
vector<thread> threads;
      for (int i = 0; i < numThreads; i++)
      {
         int start = i * dataPerThread;
         int end = (i == numThreads - 1) ? start + dataPerThread + remainingData : start +
dataPerThread;
         threads.push_back(thread(calculateBMI, start, end, gender, height, weight, BMI));
      }
      for (auto& t : threads)
      {
         t.join();
      }
```

API and functions are used:

- std::thread: It is a class that represents a single thread of execution. It is used to create and manage threads. In this code, an **std::vector<std::thread>** named threads is declared to store the thread objects.
- std::thread::thread: It is the constructor of the std::thread class. It creates a new thread and associates it with a given callable object (function, lambda expression, functor, etc.). In this code, the calculate BMI function is passed as the callable object to the thread constructor.
- std::thread::join: **t.join()** It is a member function of the std::thread class. It blocks the execution of the current thread until the thread object on which it is called finishes its execution. In this code, after creating all the threads, a loop is used to call the join function on each thread in the threads vector.

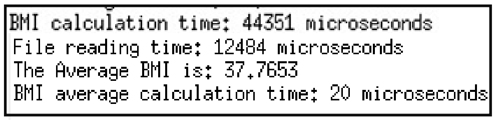
The code divides the task of calculating BMI for different ranges of data points among multiple threads. Each thread operates on a separate range of data points, which helps in parallel execution and improves performance by utilizing multiple CPU cores.

Please note that multithreading introduces concurrency, and proper synchronization mechanisms should be employed to avoid data races and ensure thread safety. In this code, since the threads are operating on .

2- Analysis according to Amdahl’s law

What percentage is the serial part of your code? What is the maximum speedup according to the available number of cores?

. Naive approach :



```
BMI calculation time: 44351 microseconds
File reading time: 12484 microseconds
The Average BMI is: 37.7653
BMI average calculation time: 20 microseconds
```

Figure 1 - for serial Naive

Time to read from file = 12484 microseconds

Time for calculate Avg BMI = 20 microseconds

Serial time = Time to read from file + Time for calculate Avg BMI

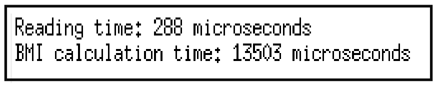
= 12484 microseconds + 20 microseconds

= 12504 microseconds

Serial Portion = $= \frac{Serial\ time}{Total\ time}$ * 100% $= \frac{12504}{44351}$ * 100% = 28.19 %

Maximum Speed up = $\frac{1}{S+\frac{(1-S)}{N}} = \frac{1}{0.2819+\frac{(1-0.2819)}{4}} =$ 2.169005965

. Multiprocessing approach:



```
Reading time: 288 microseconds
BMI calculation time: 13503 microseconds
```

Figure 2 - serial for multiprocessing

Time to read from file = 288 microseconds

Time for calculate Avg BMI = 13503 microseconds

Serial time = Time to read from file + Time for calculate Avg BMI

= 288 microseconds + 13503 microseconds

= 13791 microseconds

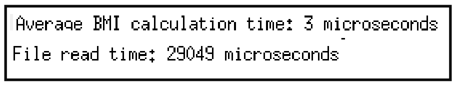
Total time (from execution code 5 time with different number of process) =

$\frac{(26033+27496+29445+32560+34261\ )}{5}$ = 29959

Serial Portion $= \frac{Serial\ time}{Total\ time} * 100\% = \frac{13791}{29959} * 100\% = 46.03\ \%$

Maximum Speed up $= \frac{1}{s+\frac{(1-s)}{N}} = \frac{1}{0.4603+\frac{(1-0.4603)}{4}} = 1.679768$

- Multithreading approach :



```
Average BMI calculation time: 3 microseconds
File read time: 29049 microseconds
```

Figure 3 - serial for Multithreading

Time to read from file = 29049 microseconds

Time for calculate Avg BMI = 3 microseconds

Serial time = Time to read from file + Time for calculate Avg BMI

= 29049 microseconds + 3 microseconds

= 29052 microseconds

Total time (from execution code 5 time with different number of process) =

$\frac{(16970+17909+20503+22109+25919\ )}{5} = 20682$

Serial Portion $= \frac{Serial\ time}{Total\ time} * 100\% = \frac{29052}{20682} * 100\% = 140.2897\%$

Maximum Speed up $= \frac{1}{s+\frac{(1-s)}{N}} = \frac{1}{1.404+\frac{(1-1-404)}{4}} = 0.7669.$

In these calculations , the code calculate the time of reading from file and the time from calculate the avg of BMI , because there tow parts are the serial part of the code of Naive approach , Multiprocessing approach and Multithreading approach .

What is the optimal number of child processes or threads?

- Process :

Well, when we go back to the schedule up, let’s begin with **the processes** optimal number, first note that the time execution increases, which means the performance being worse with increasing the number of processes, (but in usual, the time execution decrease until it reach a certain point and then the time increases, this doesn’t happened with us because I am using a virtual box), and at the table it was obvious that the optimal number of processes seems to be around 1 – 11 processes, as the time is relatively lower compared to higher process counts.

- Threads :

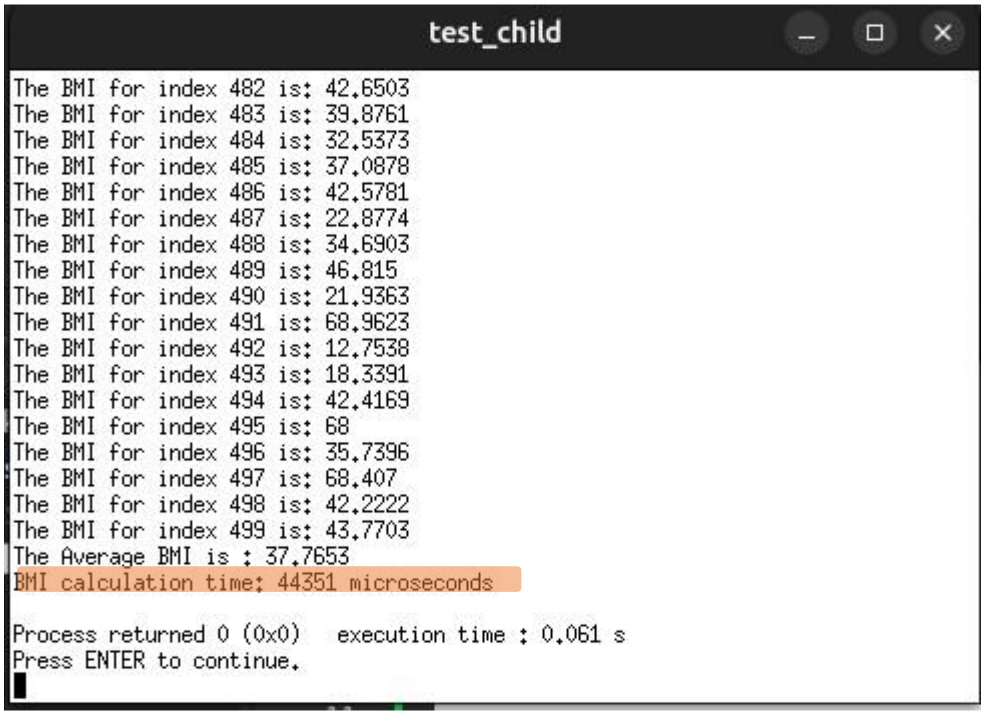
However, **the Joinable threads**, it’s obvious that they won the race, their performance here is the best, but the overhead here of joining threads does not seem to significantly impact the performance, but at the same time we can’t ignore the time increasing for increasing the number of threads, and the optimal number here is around 51 threads, as it shows a good balance between parallelization and overhead.

In conclusion, the optimal number for child processes for my case is between 1-11 children, which is a good number, and the optimal number for threads between 1-51 threads which is also a good number. And here the order of the performance from the best till the worse as following (Joinable threads, processes and then the native approach). Were a good balance between parallelisms and overhead is achieved! And it’s essential to consider that I used only one processor for these codes, the characteristics of the problem and the hardware requirements should be also considered.

3- compares the performance of the 3 approaches.

- Naive approach:

As it’s obvious at the picture below, the execution time is 44351 microsecond, which is long time for this program.



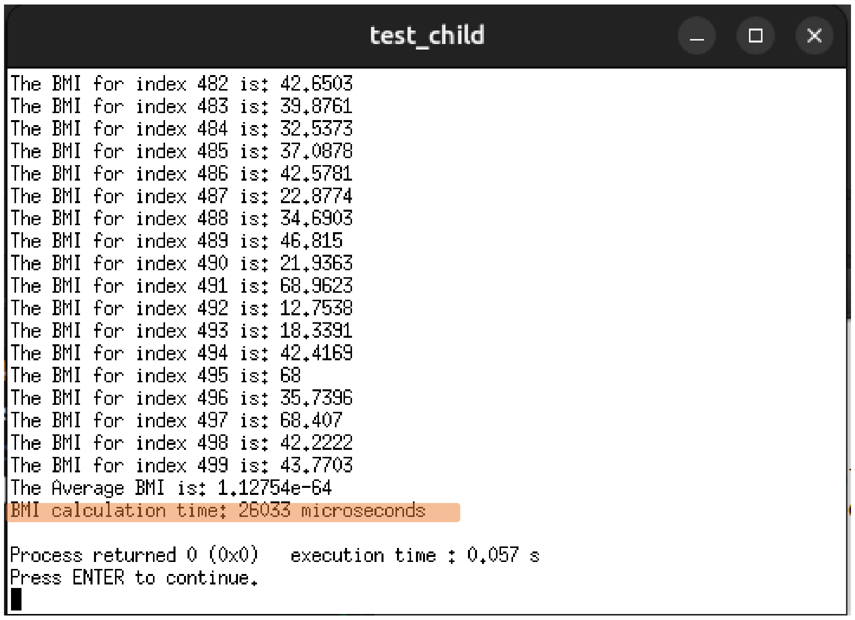
```
test_child
The BMI for index 482 is: 42.6503
The BMI for index 483 is: 39.8761
The BMI for index 484 is: 32.5373
The BMI for index 485 is: 37.0878
The BMI for index 486 is: 42.5781
The BMI for index 487 is: 22.8774
The BMI for index 488 is: 34.6903
The BMI for index 489 is: 46.815
The BMI for index 490 is: 21.9363
The BMI for index 491 is: 68.9623
The BMI for index 492 is: 12.7538
The BMI for index 493 is: 18.3391
The BMI for index 494 is: 42.4169
The BMI for index 495 is: 68
The BMI for index 496 is: 35.7396
The BMI for index 497 is: 68.407
The BMI for index 498 is: 42.2222
The BMI for index 499 is: 43.7703
The Average BMI is : 37.7653
BMI calculation time: 44351 microseconds

Process returned 0 (0x0)   execution time : 0.061 s
Press ENTER to continue.
```

Figure 4 - Naive approach time execution

- Multiprocessing approach :

Here using **2 processes, 1 for the child and one for the parent process**, and the time here is 26033 microseconds

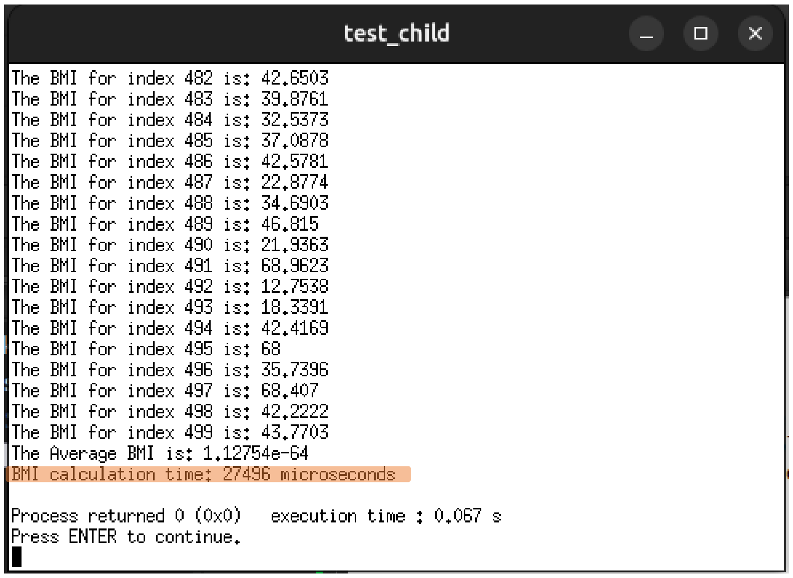


```
The BMI for index 482 is: 42.6503
The BMI for index 483 is: 39.8761
The BMI for index 484 is: 32.5373
The BMI for index 485 is: 37.0878
The BMI for index 486 is: 42.5781
The BMI for index 487 is: 22.8774
The BMI for index 488 is: 34.6903
The BMI for index 489 is: 46.815
The BMI for index 490 is: 21.9363
The BMI for index 491 is: 68.9623
The BMI for index 492 is: 12.7538
The BMI for index 493 is: 18.3391
The BMI for index 494 is: 42.4169
The BMI for index 495 is: 68
The BMI for index 496 is: 35.7396
The BMI for index 497 is: 68.407
The BMI for index 498 is: 42.2222
The BMI for index 499 is: 43.7703
The Average BMI is: 1.12754e-64
BMI calculation time: 26033 microseconds

Process returned 0 (0x0)   execution time : 0.057 s
Press ENTER to continue.
```

Figure 5 - 2 process time execution

Here using **3 processes, 2 for the child and one for the parent process**, and the time here is 27496 microseconds .

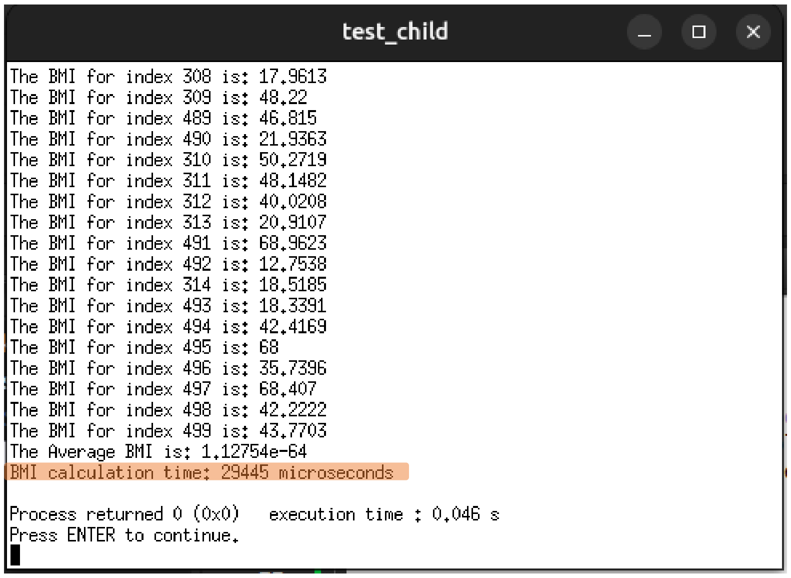


```
The BMI for index 482 is: 42.6503
The BMI for index 483 is: 39.8761
The BMI for index 484 is: 32.5373
The BMI for index 485 is: 37.0878
The BMI for index 486 is: 42.5781
The BMI for index 487 is: 22.8774
The BMI for index 488 is: 34.6903
The BMI for index 489 is: 46.815
The BMI for index 490 is: 21.9363
The BMI for index 491 is: 68.9623
The BMI for index 492 is: 12.7538
The BMI for index 493 is: 18.3391
The BMI for index 494 is: 42.4169
The BMI for index 495 is: 68
The BMI for index 496 is: 35.7396
The BMI for index 497 is: 68.407
The BMI for index 498 is: 42.2222
The BMI for index 499 is: 43.7703
The Average BMI is: 1.12754e-64
BMI calculation time: 27496 microseconds

Process returned 0 (0x0)   execution time : 0.067 s
Press ENTER to continue.
```

Figure 6 - 3 process time execution

Here using **12 processes, 11 for the child and one for the parent process**, and the time here is 29445 microseconds .

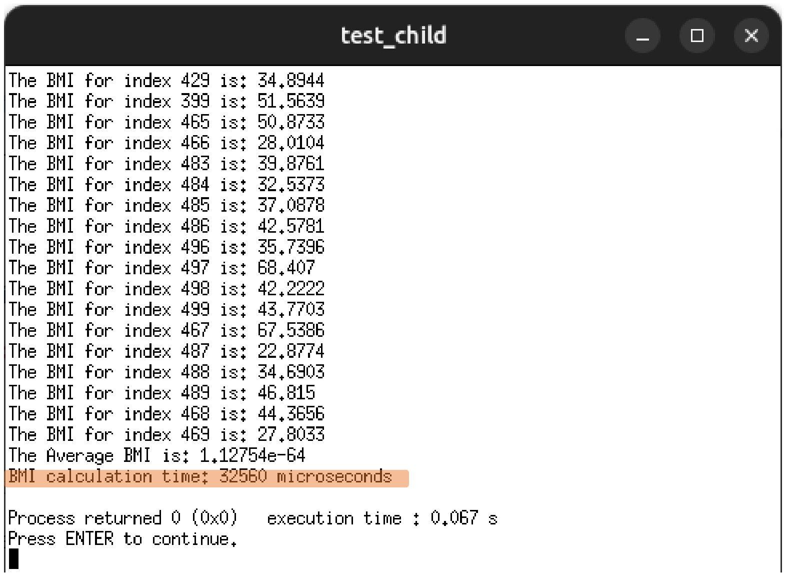

test_child

```
The BMI for index 308 is: 17.9613
The BMI for index 309 is: 48.22
The BMI for index 489 is: 46.815
The BMI for index 490 is: 21.9363
The BMI for index 310 is: 50.2719
The BMI for index 311 is: 48.1482
The BMI for index 312 is: 40.0208
The BMI for index 313 is: 20.9107
The BMI for index 491 is: 68.9623
The BMI for index 492 is: 12.7538
The BMI for index 314 is: 18.5185
The BMI for index 493 is: 18.3391
The BMI for index 494 is: 42.4169
The BMI for index 495 is: 68
The BMI for index 496 is: 35.7396
The BMI for index 497 is: 68.407
The BMI for index 498 is: 42.2222
The BMI for index 499 is: 43.7703
The Average BMI is: 1.12754e-64
BMI calculation time: 29445 microseconds

Process returned 0 (0x0)   execution time : 0.046 s
Press ENTER to continue.
```

Figure 7 - 12 process time execution

Here using **51 processes, 50 for the child and one for the parent process**, and the time here is 32560 microseconds .

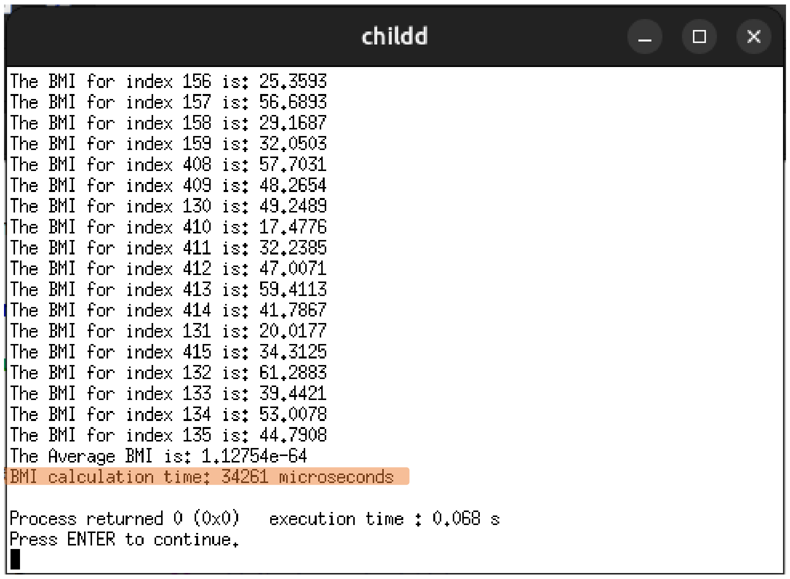

test_child

```
The BMI for index 429 is: 34.8944
The BMI for index 399 is: 51.5639
The BMI for index 465 is: 50.8733
The BMI for index 466 is: 28.0104
The BMI for index 483 is: 39.8761
The BMI for index 484 is: 32.5373
The BMI for index 485 is: 37.0878
The BMI for index 486 is: 42.5781
The BMI for index 496 is: 35.7396
The BMI for index 497 is: 68.407
The BMI for index 498 is: 42.2222
The BMI for index 499 is: 43.7703
The BMI for index 467 is: 67.5386
The BMI for index 487 is: 22.8774
The BMI for index 488 is: 34.6903
The BMI for index 489 is: 46.815
The BMI for index 468 is: 44.3656
The BMI for index 469 is: 27.8033
The Average BMI is: 1.12754e-64
BMI calculation time: 32560 microseconds

Process returned 0 (0x0)   execution time : 0.067 s
Press ENTER to continue.
```

Figure 8 - 51 process time execution

Here using **101 processes, 100 for the child and one for the parent process**, and the time here is 34261 microseconds.



childd

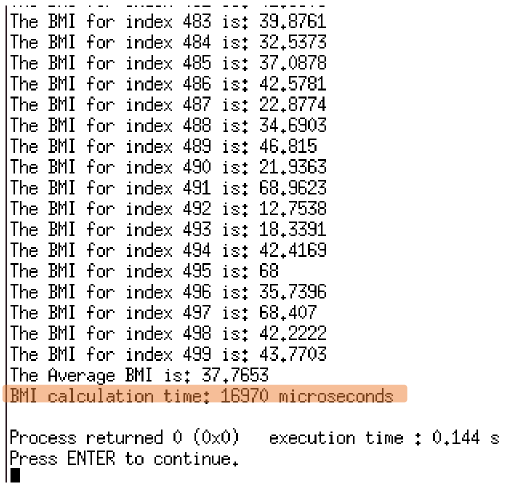
```
The BMI for index 156 is: 25.3593
The BMI for index 157 is: 56.6893
The BMI for index 158 is: 29.1687
The BMI for index 159 is: 32.0503
The BMI for index 408 is: 57.7031
The BMI for index 409 is: 48.2654
The BMI for index 130 is: 49.2489
The BMI for index 410 is: 17.4776
The BMI for index 411 is: 32.2385
The BMI for index 412 is: 47.0071
The BMI for index 413 is: 59.4113
The BMI for index 414 is: 41.7867
The BMI for index 131 is: 20.0177
The BMI for index 415 is: 34.3125
The BMI for index 132 is: 61.2883
The BMI for index 133 is: 39.4421
The BMI for index 134 is: 53.0078
The BMI for index 135 is: 44.7908
The Average BMI is: 1.12754e-64
BMI calculation time: 34261 microseconds

Process returned 0 (0x0)   execution time : 0.068 s
Press ENTER to continue.
```

Figure 9 - 101 process time execution

- **Multithreading approach** :

Here **using 1 thread,** and the time here is 16970 microseconds

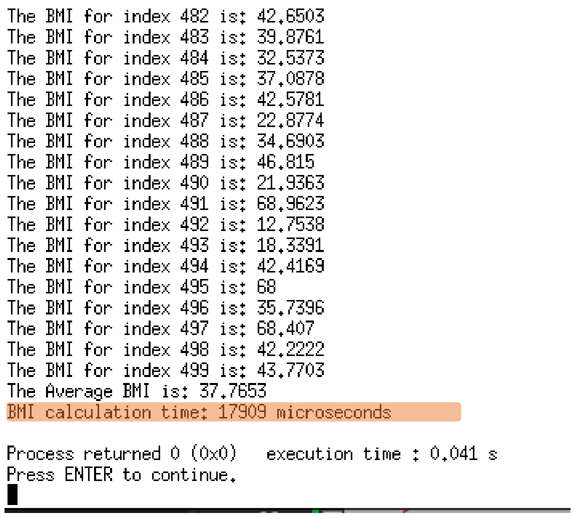


```
The BMI for index 483 is: 39.8761
The BMI for index 484 is: 32.5373
The BMI for index 485 is: 37.0878
The BMI for index 486 is: 42.5781
The BMI for index 487 is: 22.8774
The BMI for index 488 is: 34.6903
The BMI for index 489 is: 46.815
The BMI for index 490 is: 21.9363
The BMI for index 491 is: 68.9623
The BMI for index 492 is: 12.7538
The BMI for index 493 is: 18.3391
The BMI for index 494 is: 42.4169
The BMI for index 495 is: 68
The BMI for index 496 is: 35.7396
The BMI for index 497 is: 68.407
The BMI for index 498 is: 42.2222
The BMI for index 499 is: 43.7703
The Average BMI is: 37.7653
BMI calculation time: 16970 microseconds

Process returned 0 (0x0)   execution time : 0.144 s
Press ENTER to continue.
```

Figure 10 - 1 thread time execution

Here **using 2 thread,** and the time here is 17909 microseconds

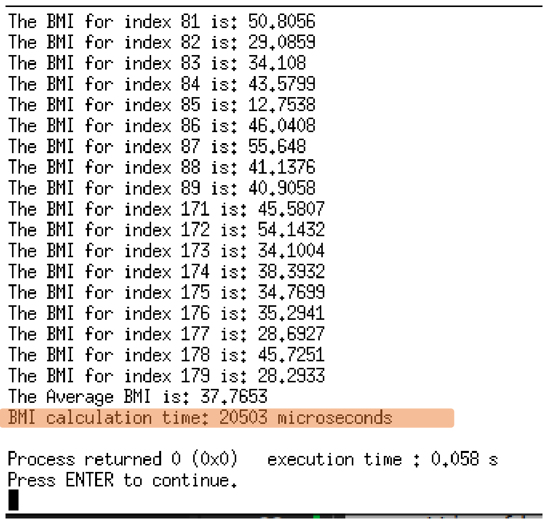


```
The BMI for index 482 is: 42.6503
The BMI for index 483 is: 39.8761
The BMI for index 484 is: 32.5373
The BMI for index 485 is: 37.0878
The BMI for index 486 is: 42.5781
The BMI for index 487 is: 22.8774
The BMI for index 488 is: 34.6903
The BMI for index 489 is: 46.815
The BMI for index 490 is: 21.9363
The BMI for index 491 is: 68.9623
The BMI for index 492 is: 12.7538
The BMI for index 493 is: 18.3391
The BMI for index 494 is: 42.4169
The BMI for index 495 is: 68
The BMI for index 496 is: 35.7396
The BMI for index 497 is: 68.407
The BMI for index 498 is: 42.2222
The BMI for index 499 is: 43.7703
The Average BMI is: 37.7653
BMI calculation time: 17909 microseconds

Process returned 0 (0x0)   execution time : 0.041 s
Press ENTER to continue.
```

Figure 11 - 2 thread time execution

Here **using 11 thread,** and the time here is 20503 microseconds

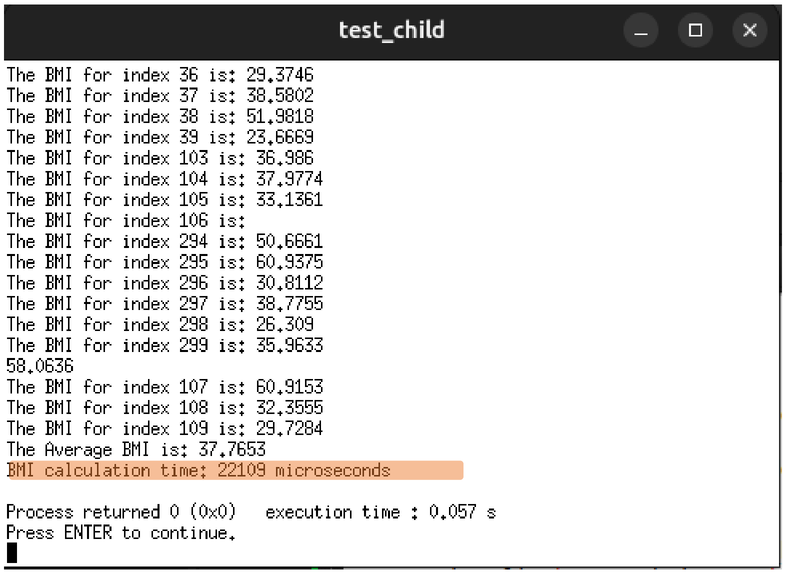


```
The BMI for index 81 is: 50.8056
The BMI for index 82 is: 29.0859
The BMI for index 83 is: 34.108
The BMI for index 84 is: 43.5799
The BMI for index 85 is: 12.7538
The BMI for index 86 is: 46.0408
The BMI for index 87 is: 55.648
The BMI for index 88 is: 41.1376
The BMI for index 89 is: 40.9058
The BMI for index 171 is: 45.5807
The BMI for index 172 is: 54.1432
The BMI for index 173 is: 34.1004
The BMI for index 174 is: 38.3932
The BMI for index 175 is: 34.7699
The BMI for index 176 is: 35.2941
The BMI for index 177 is: 28.6927
The BMI for index 178 is: 45.7251
The BMI for index 179 is: 28.2933
The Average BMI is: 37.7653
BMI calculation time: 20503 microseconds

Process returned 0 (0x0)   execution time : 0.058 s
Press ENTER to continue.
```

Figure 12 - 11 thread time execution

Here **using 50 thread,** and the time here is 22109 microseconds

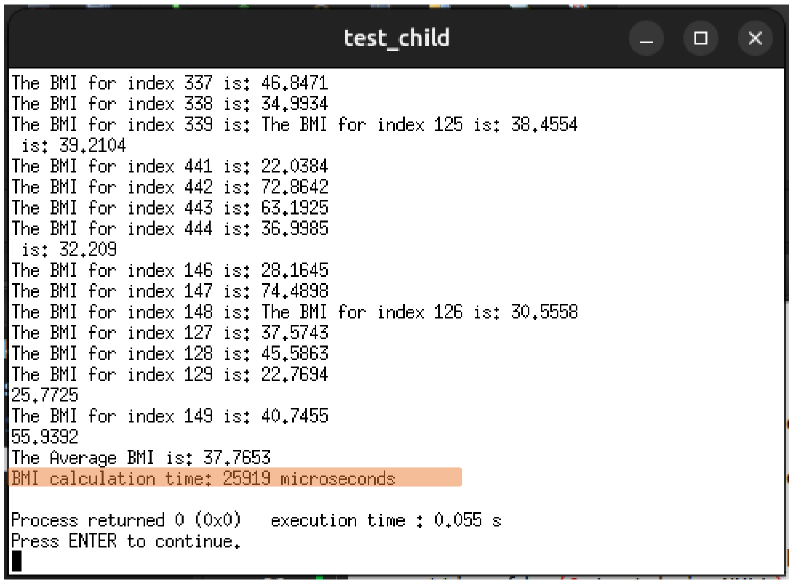


```
The BMI for index 36 is: 29.3746
The BMI for index 37 is: 38.5802
The BMI for index 38 is: 51.9818
The BMI for index 39 is: 23.6669
The BMI for index 103 is: 36.986
The BMI for index 104 is: 37.9774
The BMI for index 105 is: 33.1361
The BMI for index 106 is:
The BMI for index 294 is: 50.6661
The BMI for index 295 is: 60.9375
The BMI for index 296 is: 30.8112
The BMI for index 297 is: 38.7755
The BMI for index 298 is: 26.309
The BMI for index 299 is: 35.9633
58.0636
The BMI for index 107 is: 60.9153
The BMI for index 108 is: 32.3555
The BMI for index 109 is: 29.7284
The Average BMI is: 37.7653
BMI calculation time: 22109 microseconds

Process returned 0 (0x0)   execution time : 0.057 s
Press ENTER to continue.
```

Figure 13 - 50 thread time execution

Here **using 100 thread,** and the time here is 25919 microseconds



```
The BMI for index 337 is: 46.8471
The BMI for index 338 is: 34.9934
The BMI for index 339 is: The BMI for index 125 is: 38.4554
 is: 39.2104
The BMI for index 441 is: 22.0384
The BMI for index 442 is: 72.8642
The BMI for index 443 is: 63.1925
The BMI for index 444 is: 36.9985
 is: 32.209
The BMI for index 146 is: 28.1645
The BMI for index 147 is: 74.4898
The BMI for index 148 is: The BMI for index 126 is: 30.5558
The BMI for index 127 is: 37.5743
The BMI for index 128 is: 45.5863
The BMI for index 129 is: 22.7694
25.7725
The BMI for index 149 is: 40.7455
55.9392
The Average BMI is: 37.7653
BMI calculation time: 25919 microseconds

Process returned 0 (0x0)   execution time : 0.055 s
Press ENTER to continue.
```

Figure 14 - 100 thread time execution

Time of Execution	Number of processes||threads	throughput
0.044351 s		22.54740592 s^{-1}
0.026033 s	2 processes (1 child & 1 parent)	38.41278377 s^{-1}
0.027496 s	3 processes (2 child & 1 parent)	36.36892639 s^{-1}
0.029445 s	12 processes (11child & 1 parent)	33.96162337 s^{-1}
0.032560 s	51 processes (50child & 1 parent)	30.71253071 s^{-1}
0.034261 s	101processes(100child&1 parent)	29.18770614 s^{-1}
0.01697 s	1 thread	58.92751915 s^{-1}
0.017909 s	2 threads	55.83784689 s^{-1}
0.020503 s	11 threads	48.77335024 s^{-1}
0.022109 s	50 threads	45.23044914 s^{-1}
0.025919 s	100 threads	38.58173541 s^{-1}

The table shows how different methods of doing things at the same time affect how fast we can multiply 100x100 matrices. So here I am comparing different methods to see which one works best.
The regular way and using multiple processes were tried, and we saw that as we used more processes, the benefits started to decrease. We also looked at threads,. Threads turned out to be the better choice..

For multiplying 100x100 matrices specifically, joinable threads performed really well. They're great when we need to share memory and do things in small pieces. threads are good because they can grow when needed, work well together,
and don't cost too much to start or communicate with each other. This study helps us see the pros and cons of using different methods, reminding us to think about the task and the system when picking the best way to do things, and to pay more attention and see how much the architecture of the computer, and the CPU speeds affects the results.

•conclusion

In conclusion the naive approach executes tasks sequentially, resulting in suboptimal performance for complex tasks. Multiprocessing divides tasks into independent processes for true parallelism, ideal for tasks that can be easily separated. Multithreading creates multiple threads within a single process, allowing for concurrent task execution and data sharing, but requiring careful synchronization. Choosing between these approaches depends on task characteristics and requirements.

.References

[1] : https://www.calculator.net/bmi-calculator.html?cage=20&csex=m&cheightfeet=5&cheightinch=10&cpound=160&cheightmeter=167&ckg=81&printit=0&ctype=metric&x=Calculate

[2] : https://www.linkedin.com/pulse/naive-approach-gateway-problem-solving-success-pranav-nigam

[3] : https://www.britannica.com/technology/multiprocessing

[4] : https://www.techtarget.com/whatis/definition/multithreading