Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Parallel Systems

Assignment 2

Name: Giorgos Nikolaou **sdi:** 1115202000154

Name: Helen Fili **sdi:** 1115202100203

1 Exercise 2.1

1.1 Description of the problem

In this problem, the task is to use the **Monte Carlo** method to approximate π by sampling random points in a 2×2 square (centered at $(0, 0)$) and observing how many of them fall within the unit circle. We approach the problem with both serial and parallel implementations (using `OpenMP`). As expected, the method produces better approximations as we increase the number of iterations, but that is not efficient when using the serial algorithm. To accelerate the computation, we utilize `OpenMP` to parallelize the random sampling among them.

1.2 Brief description of the solution

Description of Algorithms:

- **Serial**: The serial version of the algorithm is executed when the command line argument `<threads>` is set to 1.
- **Parallel**: The parallel version of the algorithm is executed when the command line argument `<threads>` is greater than 1. In this version, the samples of random points from arrow throwing are divided equally among the threads.

`serial_pi(LONG n)`: In this function, we compute the number of arrows that landed in the critical area (inside the unit circle). This calculation is performed using the serial algorithm, and the function returns an approximation of π.

`threaded_pi(size_t threads, LONG n)`: This function computes the number of arrows -that landed in the critical area- for the corresponding thread and then adds the result to the variable `arrows`, before calculating and returning the approximation of π. We use the directive `#pragma omp parallel` to create a group of threads, specifying the number of threads with `num_threads`. The term `reduction` is used to combine results from different threads, in our case, it involves the summation of these results. The `#pragma omp for` directive is employed to distribute the total number of arrows among threads.

Implementation Details:

- Type `long long` is macro-defined as `LONG`.
- For the point sampling, a thread-safe version of `rand()` is used (`rand_r()`).

1.3 Presentation of experiments and analysis

Below, we provide the results for various numbers of threads and sampling points. We also created the corresponding plot where `<n>` is the horizontal axis and presented logarithmically. See: `./monte_carlo/output/plot.png`

Specifically, we executed the program with every combination of the following parameters:

- `<threads>`: $[1, 2, 4, 8, 16, 32]$
- `<n>`: $[10^7, 10^8, 10^9, 5 \cdot 10^9, 10^{10}]$

To conduct this grid search and generate the corresponding plot, we developed the script `plots.py`. To reproduce, execute:

```
$ cd monte_carlo
$ make monte_carlo
$ python ./plots.py
```

Listing 1: Monte Carlo Plotting

To perform manual executions, run:

```
$ cd monte_carlo
$ make monte_carlo
$ ./monte_carlo <threads> <n>
```

Listing 2: Monte Carlo Execution

Experiments were conducted locally, 4 times each, using the following specifications:

- **OS**: `WSL: Ubuntu-22.04`.
- **CPU**: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`, featuring 6 cores with 2 threads each.
- **Compiler**: `gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0`

Similar to the `PThreads` implementation, we make the following observations:

- **Doubling** the number of threads results in **reduction** of the execution time by *half*, as expected due to the shared workload across multiple threads.
- In our configuration, featuring 6 cores with 2 threads each and totaling 12 threads (the maximum number of processor threads), we observe improvement up to the use of 12 threads. However, increasing the thread count beyond 12 does **not** yield further reductions in execution time.

The performance of the two methods appears to be consistent, which is expected given the straightforward nature of the parallelization applied to the problem.

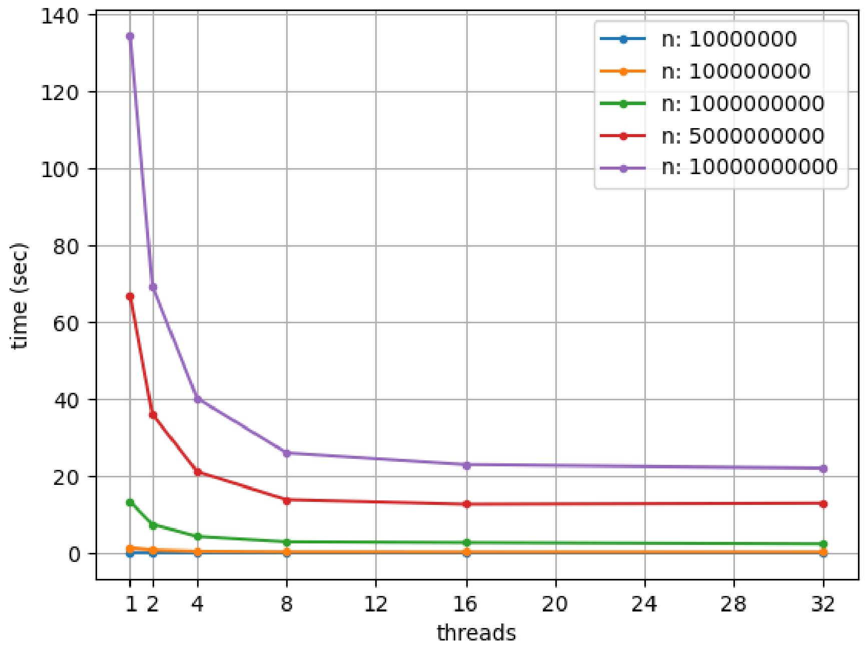


Figure 1: Monte Carlo Results

2 Exercise 2.2

2.1 Description of the problem

The goal of this exercise is to enhance the parallel **matrix-vector multiplication** program sourced from the book. The aim is to optimize the computation of multiplying an upper triangular matrix with a vector. This optimization entails avoiding unnecessary computations and leveraging the capabilities of `OpenMP` for parallelization.

2.2 Brief description of the solution

In this implementation, we **flatten** the $2D$ matrix into $1D$ vector and perform the multiplication accordingly. The total computations are divided among the threads, with each one handling the computations for a specific range rows.

We experiment with all the options for the `schedule` clause in `OpenMP` to observe how they affect the parallel execution of the loop.

Implementation Details:

- Only one dimension of the matrix needs to be provided since the matrices are square.
- To avoid unnecessary computations in the loops, calculations are performed exclusively for the elements on and above the diagonal.
- To ensure reproducibility of the results in our measurements, we keep the seed constant in the `my_rand` function.
- To verify the results of the multiplication, compile with `-DVERIFY`, run the desired algorithm, and then execute the script `verify.py` with the appropriate parameter `<n>`.

2.3 Presentation of experiments and analysis

Below, we provide the results for various numbers of threads, matrix dimensions, scheduling types and chunk sizes.

Specifically, we executed the program with every combination of the following parameters:

- `<threads>`: $[1, 2, 4, 8, 12, 16]$
- `<n>`: $[4096, 16384]$
- `<sch type>`: [baseline, auto, static, static_cs, dynamic, dynamic_cs, guided, guided_cs]
- `<chunk size>`: $[4, 8, 16, 32, 64, 128]$

To perform the grid search described in the assignment and produce the corresponding plots, we created the script `plots.py`. To reproduce, execute:

```
$ cd mat_vec_mul
$ make mvm
$ python ./plots.py
```

Listing 3: Matrix Multiplication Plotting

To perform manual executions, run:

```
$ cd mat_vec_mul
$ make mvm
$ ./mvm <threads> <n> <sch type> <chunk size> <log file, optional>
```

Listing 4: Matrix Multiplication Execution

Experiments were conducted locally, 4 times each, using the following specifications:

- **OS**: `WSL: Ubuntu-22.04`.
- **CPU**: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`, featuring 6 cores with 2 threads each.
- **Compiler**: `gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0`

The results are presented side-by-side for both sizes, with $n = 4096$ on the left and $n = 16384$ on the right.

2.3.1 Performance

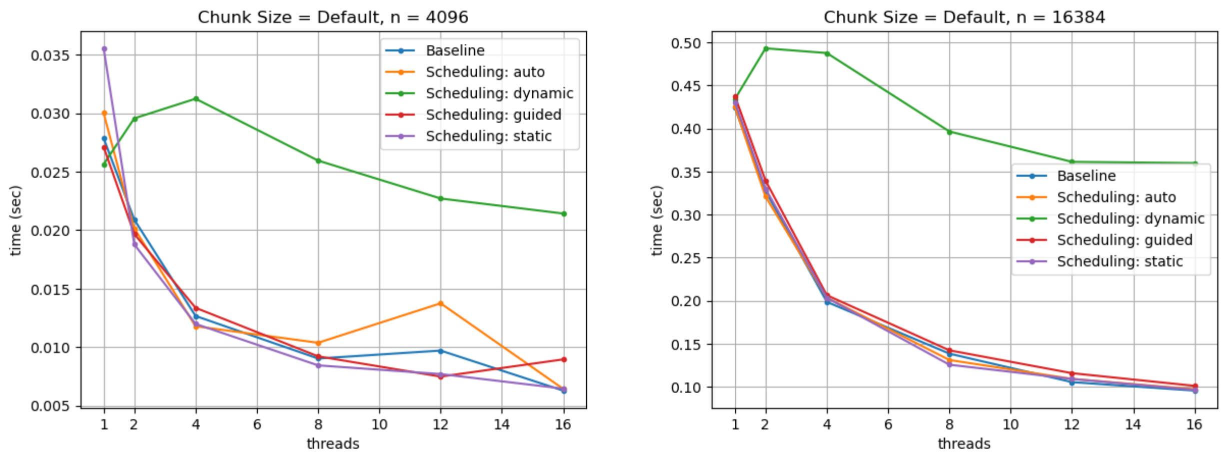
When applicable, we also plot a baseline, which is just parallelization with no scheduling.

2.3.2 Scheduling with default chunk size

In this case, the following schedulers are available:

- `auto`: Scheduling is determined by the number of available cores and the number of iterations.
- `static`: Iterations are assigned to threads in equal-sized chunks before the execution.
- `dynamic`: Additional iterations are assigned to threads once they finish.
- `guided`: Iterations are assigned to threads in progressively smaller chunks.

Figure 2: Default Scheduling Results



We make the following observations:

- As anticipated, an increase in the number of threads corresponds to a decrease in execution time.
- The `dynamic` scheduling method consistently demonstrates inferior performance compared to other methods across various thread counts. This can be attributed to its additional runtime overhead. Specifically, the default chunk size is 1, resulting in constant chunk reassignment on each thread and, consequently, a significant decrease in performance.
- With their default parameters, the `auto`, `static`, and `guided` scheduling methods show comparable performance. This similarity arises from the use of default parameters, resulting in a distribution of iterations for `static` and `guided` that is very similar, leading to the observed results. Similarly for the `auto` and `baseline` scheduling.
- At 12 threads, the maximum supported number in our configuration, a notable increase in execution times is observed for $n = 4096$. This phenomenon is likely attributed to potential false-sharing, as the number of threads doesn't allow for optimal memory alignment.

2.3.3 Scheduling with various chunk sizes

Below we present the plots for different chunk sizes.

We make the following observations:

- Like before, an increase in the number of threads corresponds to a decrease in execution time. Additionally, utilizing scheduling we further increase the performance significantly.
- As anticipated, an increase in the chunk size results in improved performance for `dynamic` scheduling (Figure 6), eventually surpassing other methods (Figure 5). This improvement is evident in a speedup of up to 1.5 (consistently observed with up to the maximum amount of supported threads, in comparison to the `baseline`).
- As discussed in subsubsection 2.3.2, we reconfirm the suboptimal performance of the default parameters for `dynamic` scheduling, now in comparison to different chunk sizes within the same scheduling protocol (Figure 6). Furthermore, although there is a noticeable improvement with larger chunk sizes, we observe that it becomes negligible beyond a chunk size of 64.

Figure 3: Chunk Size = 16

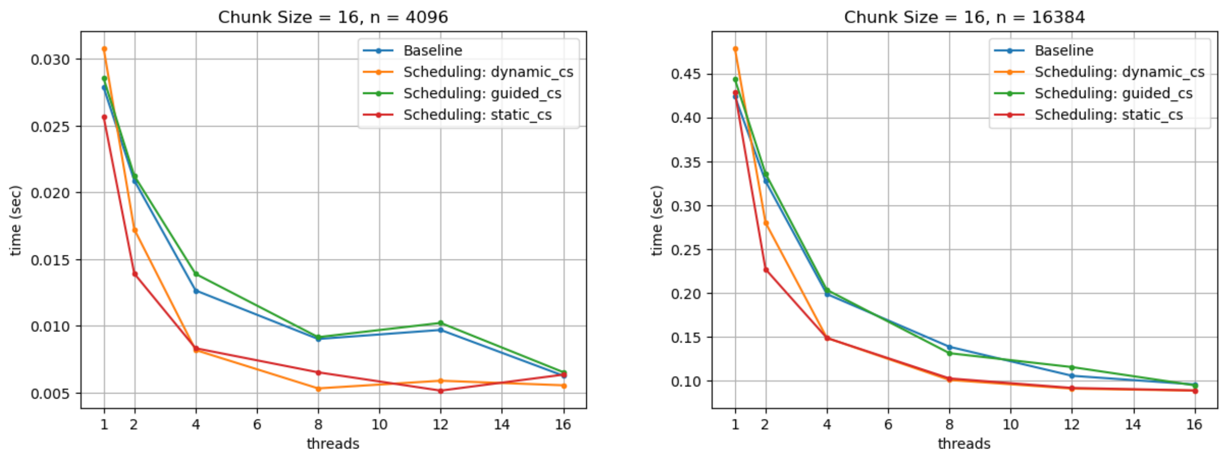


Figure 4: Chunk Size = 64

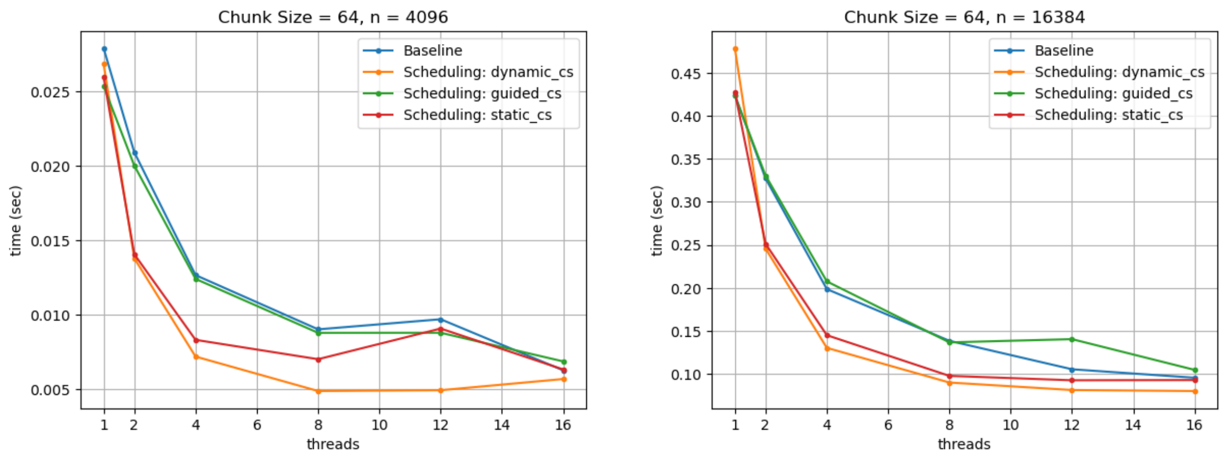
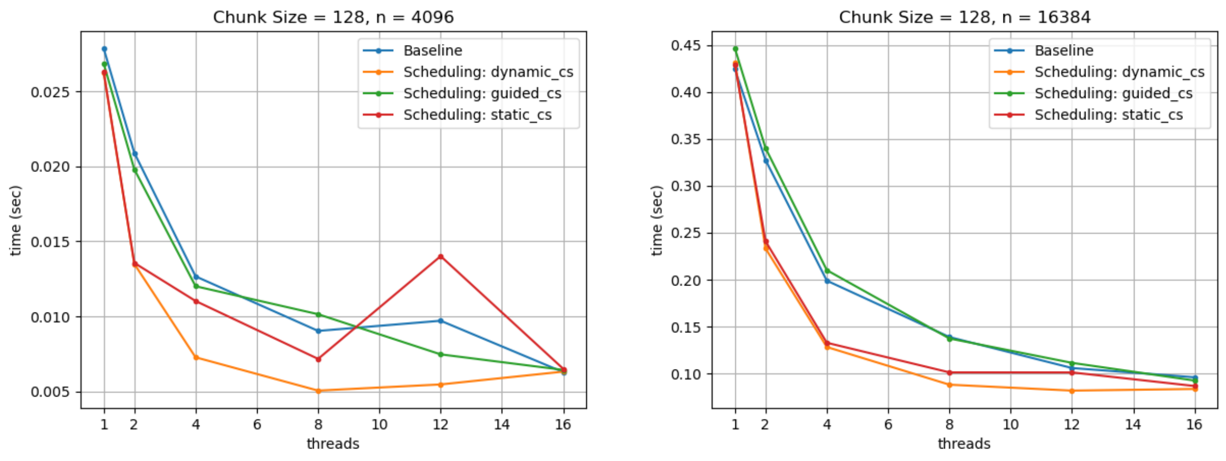


Figure 5: Chunk Size = 128



- The `guided` scheduling exhibits nearly identical performance with the `baseline`. This is attributed to the fact that, when no scheduler is specified, a variant of `guided` protocol is likely employed.
- As discussed in subsubsection 2.3.2, at 12 threads, we once again observe an increase in execution time for similar reasons.
- Overall, `dynamic` scheduling with a chunk size of 128 emerges as the best performing method.

Figure 6: `dynamic` Scheduling Comparison

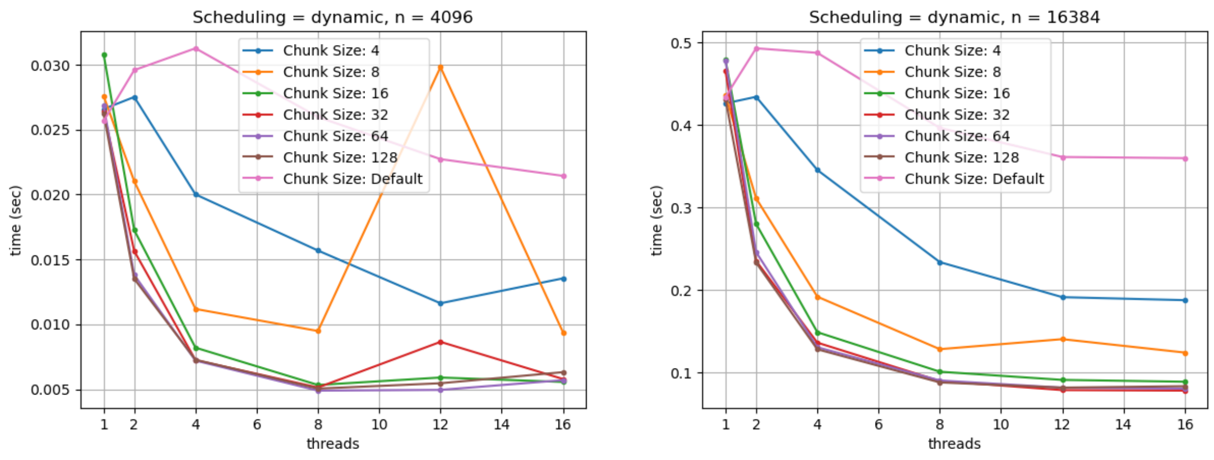


Figure 7: `guided` Scheduling Comparison

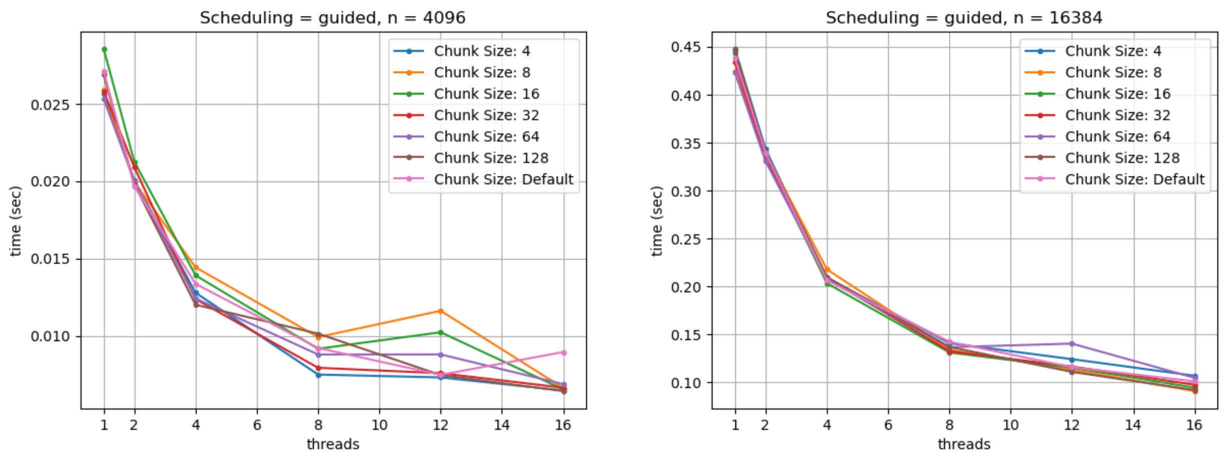
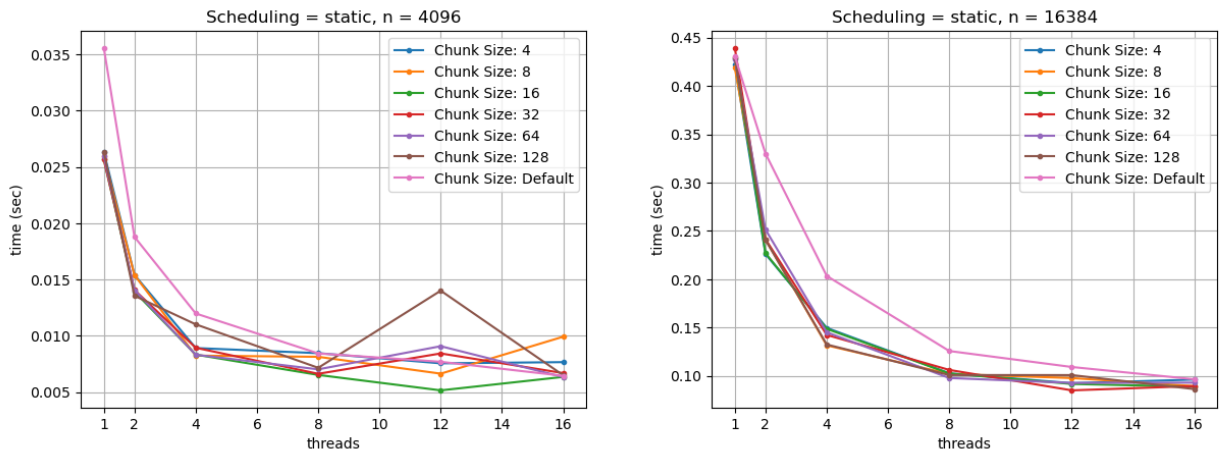


Figure 8: `static` Scheduling Comparison



- The consistent performance of `guided` scheduling across various chunk sizes can be attributed to the fact that the distribution of iterations is not solely determined by the specified chunk size. Specifically, during the initial, more resource-intensive iterations, the distribution between the threads remains constant, contributing to the observed uniformity in performance.

- As anticipated, when employing `static` scheduling, larger chunk sizes result in slower performance. However, this effect is noticeable only for extremely large chunk sizes, attributed to the progressively smaller workload assigned to each iteration.

3 Exercise 2.3

3.1 Description of the problem

For this exercise, we aim to use `OpenMP` to create a parallel program for solving linear systems through Gauss elimination. When dealing with large-scale linear systems, the commonly adopted method involves applying **Gauss elimination** followed by **back substitution**. This method effectively converts an $n \times n$ linear system into an upper triangular form. Back substitution is used to find the values of the unknown variables.

3.2 Brief description of the solution

For Gauss elimination and back substitution, we implement the algorithms provided in the assignment. We observe opportunities for parallelization in certain loops, allowing us to optimize the performance of our program. More specifically:

Gauss Elimination:

- First Loop: In a given snapshot at the i^{th} row, the Gauss elimination algorithm requires that for each subsequent row, all elements in the first $i-1$ columns are set to 0. Attempts to parallelize the loop may result in inconsistent results due to dependencies between the operations.
- Second Loop: Can be parallelized.
- Third Loop: Can be parallelized.

Before the initial `for` loop, we utilize the `#pragma omp parallel` directive to instantiate the specified number of threads. These threads are then employed to parallelize either of the nested loops, resulting in improved performance as the threads are not created and destroyed in each iteration. Additionally, when parallelizing the innermost loop, it is necessary to use the `#pragma omp single` directive to execute the calculation of the `ratio` and update the vector `b`, as these operations must occur only once.

Back Substitution:

- First Loop: As the name suggests, the substitution must be performed linearly, starting from the end and working backward. It is evident that there is a dependency between iterations, and thus, the outer loop cannot be parallelized.
- Second Loop: Can be parallelized.

The parallelization is implemented as before, utilizing the `#pragma omp parallel` and `#pragma omp single` directives.

Implementation Details:

- Since the matrices are square, only one dimension of the matrix needs to be provided.
- We flatten the $2D$ matrix into a $1D$ vector and perform the operations accordingly.
- To ensure the correctness of the algorithm, compile the code with the `-DVERIFY` flag, run the program, and then execute the `verify.py` script with the appropriate parameter `<n>`.

To compile and run the program, use the following commands:

```
$ cd gauss
$ make gauss FLAGS="<define flags>"
$ ./gauss <threads> <n> <log file , optional>
```

Include the following in `FLAGS` to compile with the desired:

- `-DVERIFY`: Write the matrices and vectors in order to verify the results.
- `-DTRI1`: Parallelize the second loop of trigonalization.
- `-DTRI2`: Parallelize the third loop of trigonalization.
- `-DBACK`: Parallelize the second loop of back-substitution.

3.3 Presentation of experiments and analysis

Below, we provide the results for various numbers of threads, matrix dimensions and parallelization methods.

Specifically, we executed the program with every combination of the following parameters:

1. - `<threads>`: $[2, 4, 8, 12]$
   - `<n>`: $[1024, 4096]$
   - `<parallelism>`: [Trig. Second loop, Trig. Third loop, Back-Substitution]
2. With compilation flag `-O3`:
   - `<threads>`: $[2, 4, 8, 12, 16]$
   - `<n>`: $[1024, 4096, 10240]$
   - `<parallelism>`: [Trig. Second loop, Trig. Third loop, Back-Substitution]

We also ran the serial version of the algorithm, and its results are represented on the plots when the `threads` parameter is set to 1.

Initially, we intended to run only the first set of parameters. However, upon experimenting with compiler optimizations, we were surprised to observe a significant impact on parallelization. Subsequently, we proceeded to test the second set (with the `-O3` optimization flag), and a detailed analysis is presented in subsubsection 3.3.2.

The code that performs the grid search as described and generates the corresponding plots is located in the script `gauss/plots.py`.

Experiments were conducted locally, 4 times each, using the following specifications:

- **OS**: `WSL: Ubuntu-22.04`.
- **CPU**: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`, featuring 6 cores with 2 threads each.
- **Compiler**: `gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0`

3.3.1 Results without compiler optimizations

We make the following observations:

- Parallelizing the algorithm yields clear benefits. In the trigonalization step, a notable performance boost of up to 60% is observed by parallelizing the second `for` loop (Figure 9). The increased performance is most noticeable in larger dimensions, where even parallelizing the innermost loop demonstrates clear performance gains, although not as pronounced as parallelizing the second loop.
- For $n = 1024$, we observe that parallelization does not improve back-substitution. This is attributed to the additional overhead of the threads and the potential for false sharing, outweighing any performance gains.
- At 12 threads, a notable performance reduction is observed, likely attributed to memory alignment and the distribution of workload among threads, as discussed in the previous exercise.

Figure 9: Trigonalization, no optimizations

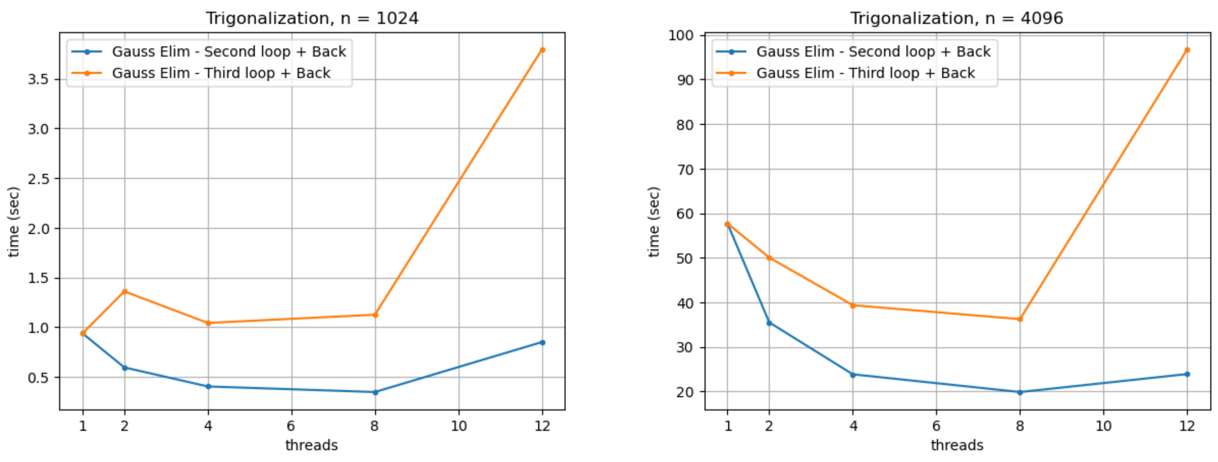
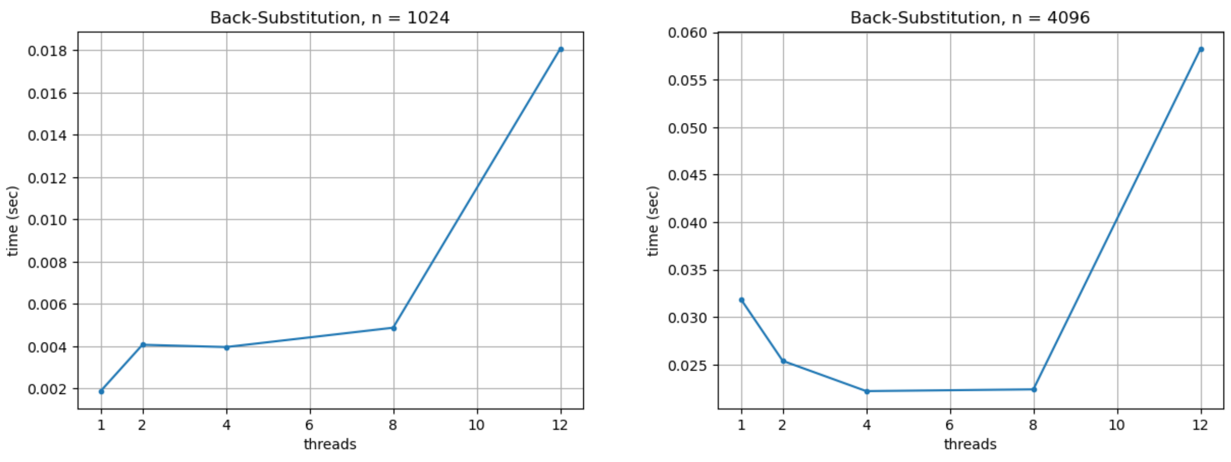


Figure 10: Back-Substitution, no optimizations

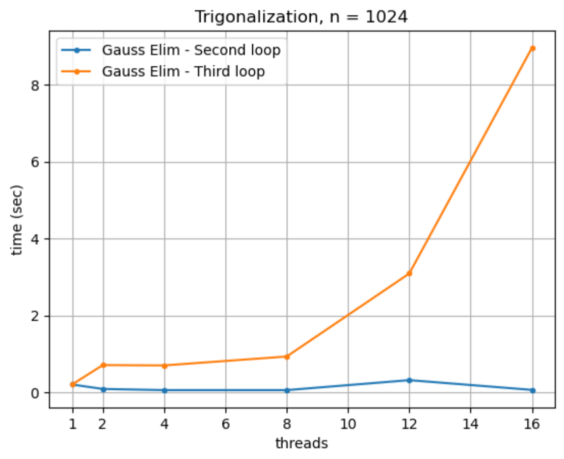


3.3.2 Results with compiler optimizations

We make the following observations:

- For $n = 4096$, the serial version of the algorithm completes the trigonalization in under 20 seconds (Figure 11), outpacing even the best-performing parallel version without optimizations.
- For smaller dimensions, employing compiler optimizations proved counterproductive in both the trigonalization and back-substitution steps, resulting in significantly worse performance as the number of threads increased, particularly when parallelizing the third `for` loop.
- In contrast, for large dimensions, optimization flags facilitated the completion of execution within a reasonable time. Specifically, for $n = 10240$, we observed performance gains when increasing the number of threads, particularly when parallelizing the second loop of the trigonalization and the back-substitution.

Figure 11: Trigonalization, with `-O3`



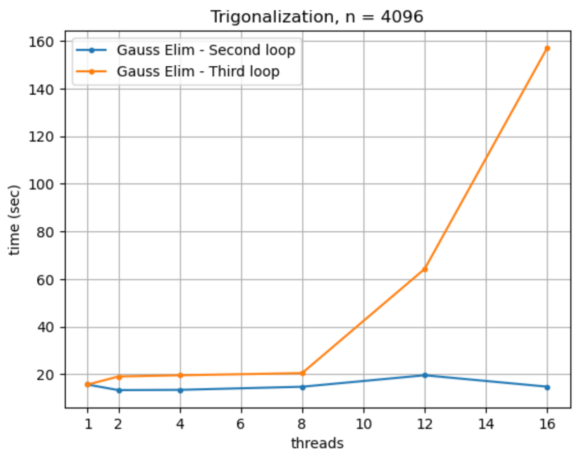
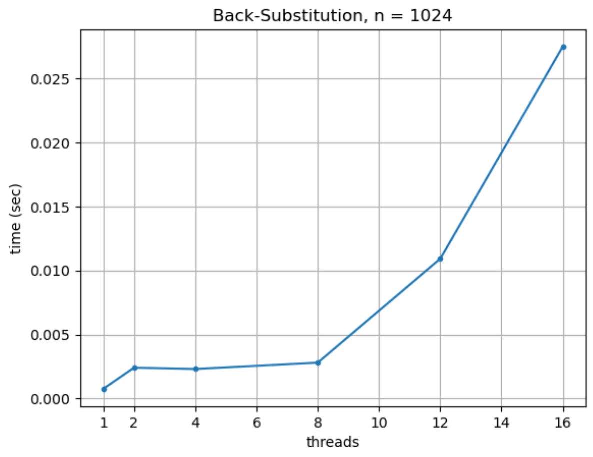


Figure 12: Back-Substitution, with `-O3`



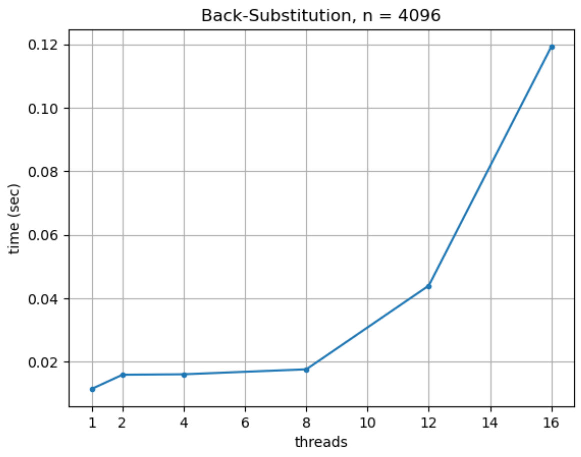
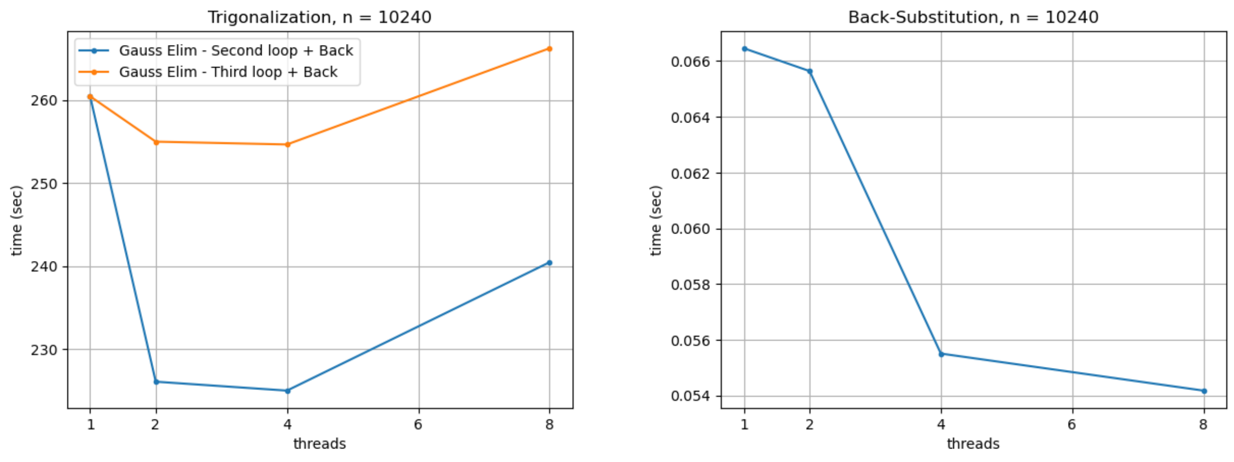


Figure 13: $n = 10240$, with `-O3`



4 Exercise 2.4

4.1 Description of the problem

In this exercise, our objective is to parallelize the top-down implementation of the merge sort algorithm using `OpenMP`. **Merge sort** is a widely used and efficient sorting algorithm that operates based on the divide-and-conquer strategy.

4.2 Brief description of the solution

Description of Algorithms:

- **Serial**: The serial version of the merge sort algorithm is executed when the command line argument `<threads>` is set to 1.
- **Parallel**: The parallel version of the merge sort algorithm is executed when the command line argument `<threads>` is greater than 1.

`merge(int* array, size_t left, size_t middle, size_t right)`: This function compares elements from the two sorted sub-arrays and arranges them in the correct order.

`serial_mergeSort(int* array, size_t left, size_t right)`: This function implements the classic mergesort algorithm.

`parallel_mergeSort(int* array, size_t left, size_t right)`: This function is designed to implement a parallelized version of the mergesort algorithm. During each recursive call, a separate task is dynamically created, based on the directive `#pragma omp task if (right-left > THRESHOLD)`, enabling parallel execution. To ensure synchronization, we employ `#pragma omp taskwait` before proceeding to sort the sub-arrays.

Implementation Details:

- The minimum sub-array size for parallelization is defined as `#define THRESHOLD (n / (threads * 512))`, where `n` is the size of the array and `threads` is the number of parallel threads.
- To ensure the correctness of the algorithm, compile the code with the `-DVERIFY` flag and run the program. We utilized the `execlp()` command to execute the `verify.py` program, validating the accuracy of our implementation.

4.3 Presentation of experiments and analysis

Below, we provide the results for various numbers of threads and array sizes. We also created the corresponding plot where `<n>` is the horizontal axis and presented logarithmically. See: `./merge_sort/output/plot.png`

Specifically, we executed the program with every combination of the following parameters:

- `<threads>`: $[1, 2, 4, 8, 12, 16, 32]$
- `<n>`: $[10^7, 2 \cdot 10^7, 5 \cdot 10^7, 10^8]$

To conduct this grid search and generate the corresponding plot, we developed the script `plots.py`. To reproduce, execute:

```
$ cd merge_sort
$ make merge_sort
$ python ./plots.py
```

Listing 5: Merge Sort Plotting

To perform manual executions, run:

```
$ cd merge_sort
$ make merge_sort
$ ./merge_sort <n> <threads>
```

Listing 6: Merge Sort Execution

Experiments were conducted locally, 4 times each, using the following specifications:

- **OS**: `WSL: Ubuntu-22.04`.
- **CPU**: `Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz`, featuring 6 cores with 2 threads each.
- **Compiler**: `gcc (Ubuntu 11.3.0-1ubuntu1 22.04) 11.3.0`

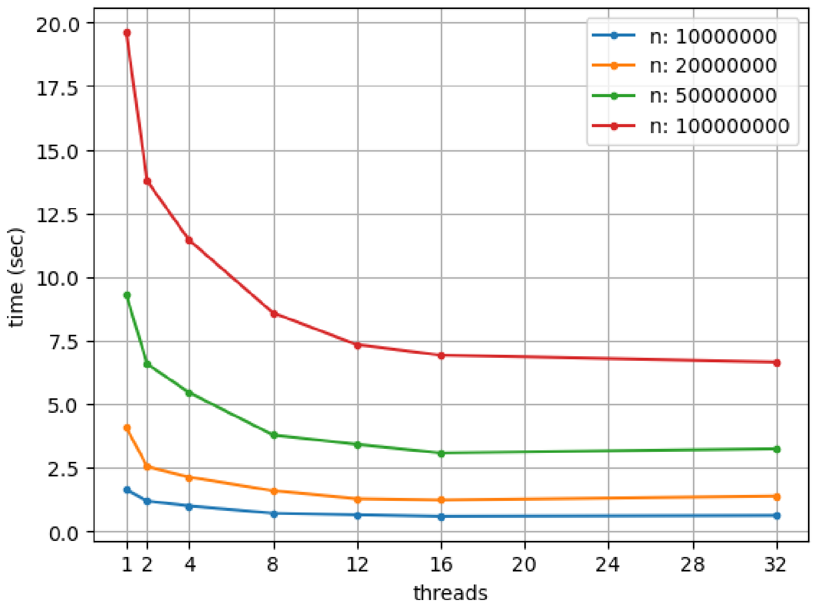


Figure 14: Merge Sort Results

We make the following observations:

- An increase in the number of threads results in improved performance. However, unlike `Monte Carlo` (subsection 1.3), the improvement is not linearly dependent on the number of threads, likely due to the influence of the `-O3` optimization flag. Nevertheless, there is a significant improvement of more than 50% compared to the serial algorithm when utilizing a substantial number of threads, such as 12 or 16.
- In our configuration, featuring 6 cores with 2 threads each and totaling 12 threads (the maximum number of processor threads), we would anticipate that increasing the threads beyond 12 would **not** result in further reductions in execution time. Surprisingly, however, we observe improvement with 16 threads, possibly attributable to better memory alignment of the input array.
- Beyond 16 threads, no further improvement was observed.