CITS5507 Project 1: Parallel Implentation and Optimal Findings of Compressed Matrix Multiplication

Dayu Liu (24188516)

Semester 2, 2024

Introduction

Matrix multiplication becomes increasingly expensive as the dimension size grows, particularly with sparse matrices, which contain mostly zero values and only a few non-zero elements. This results in unnecessary computations involving zeros, making the process inefficient in terms of both time and memory. In this project, we implemented a customized, simplified version of compressed matrix multiplication to tackle these inefficiencies. Our goal was to parallelize the compressed matrix multiplication and conduct a series of experiments to evaluate the impact of various parameters and parallelization strategies on execution time. All code implementations and lab results are included in the submitted zip file.

Project Workflow

The project can be divided into two main parts:

1. Implementation:
In the first step, we implemented a compressed matrix multiplication algorithm. A helper function for ordinary matrix multiplication was also created to check the integrity of results and ensure correctness.

2. Parallel Experiments:
In the second step, we conducted experiments using different thread counts and scheduling strategies. We monitored the performance using OpenMP parallelization on Pawsey supercomputer. Performance was evaluated based on execution time across various parameter settings. To avoid performance degradation over time and minimize the impact of individual unit behavior on Setonix, our experiments were split into multiple sub-tasks, each handling a specific range of thread counts.

Experiment Setup

The following parameters were explored during the experiments:

Fixed Parameters:

- Matrix dimension size: 100,000 x 100,000

Experiment 1: Thread Count

- Number of Threads: 1-120
- Percentage of non-zero values in matrices: 1

In the first experiment, the number of threads ranged from 1 to 120. Three matrix multiplications were performed for each thread count, with varying percentages of non-zero elements (1%, 2%, and 5%).

Experiment 2: Scheduling Strategies

- Thread Scheduling strategies: Static, Dynamic, Guided, Runtime

- Chunk size: Default (0), 100, 200, 500

In the second experiment, the matrix with 1% non-zero elements was used to explore the impact of different scheduling strategies. The optimal thread count for 1% matrix multiplication (60 threads) was used throughout. Since the matrix size is large (100,000 x 100,000), chunk size plays a crucial role in performance. Thus, experiments with different chunk sizes (100, 200, 500) were also conducted, with the default chunk size (indicated as 0) serving as a baseline.

Matrix Generation

The `generateMatrices()` function is responsible for creating matrices with a specified density of non-zero values. Each element in the matrix is assigned a value based on a random selection process that determines whether it will be a non-zero value, according to the specified density (percent).

This function generates the original sparse matrix and two compressed matrices (values and indices) row by row. When a non-zero value is selected, a numerical value between 1 and 10 is assigned. These non-zero elements, along with their column indices, are stored in two corresponding compressed matrices.

Finally, the generated matrices are saved into local files for future use, with a suffix added to distinguish between different matrices. For example, in `FileB_matrixX_percent_1`, "FileB" indicates that it's the value matrix, "matrixX" refers to the first matrix in the multiplication, and "percent_1" denotes that the matrix was generated with a 1% density of non-zero elements.

Compressed Matrix Multiplication Implementation

The function `compressedMatrixMultiply()` leverages OpenMP to parallelize the multiplication of compressed matrices. It operates by iterating over the compressed representation, and the loop is parallelized using OpenMP's `parallel for` directive, which distributes the work among a given number of threads.

For each element in the compressed matrix, its value and corresponding index are extracted. These values are then used to compute the product, and the result is added directly to the appropriate position in the result matrix.

Ordinary Matrix Multiplication & Check Integrity

To validate the correctness of the compressed matrix multiplication, we implemented a standard matrix multiplication function, `matrixMultiply()`. This function was used to generate the results of a full, uncompressed matrix multiplication.

To verify accuracy, the `checkIntegrity()` function compared the outputs of the ordinary and compressed multiplication methods. This comparison was performed row by row and column by column within a two-layer loop, ensuring that each corresponding element in both result matrices matched.

Experiment 1

In the main program (`project1.c`), command-line arguments divide the workload by specifying different thread ranges and sparsity percentages for the compressed matrix multiplication task. This enables us to break the task into sub-tasks, each responsible for a specific range of threads on a certain set of matrices.

Matrices are first loaded from pre-generated files using the `loadMatrices()` function, which reads in compressed values and indices, to ensure every sub-task is working on the same set of data.

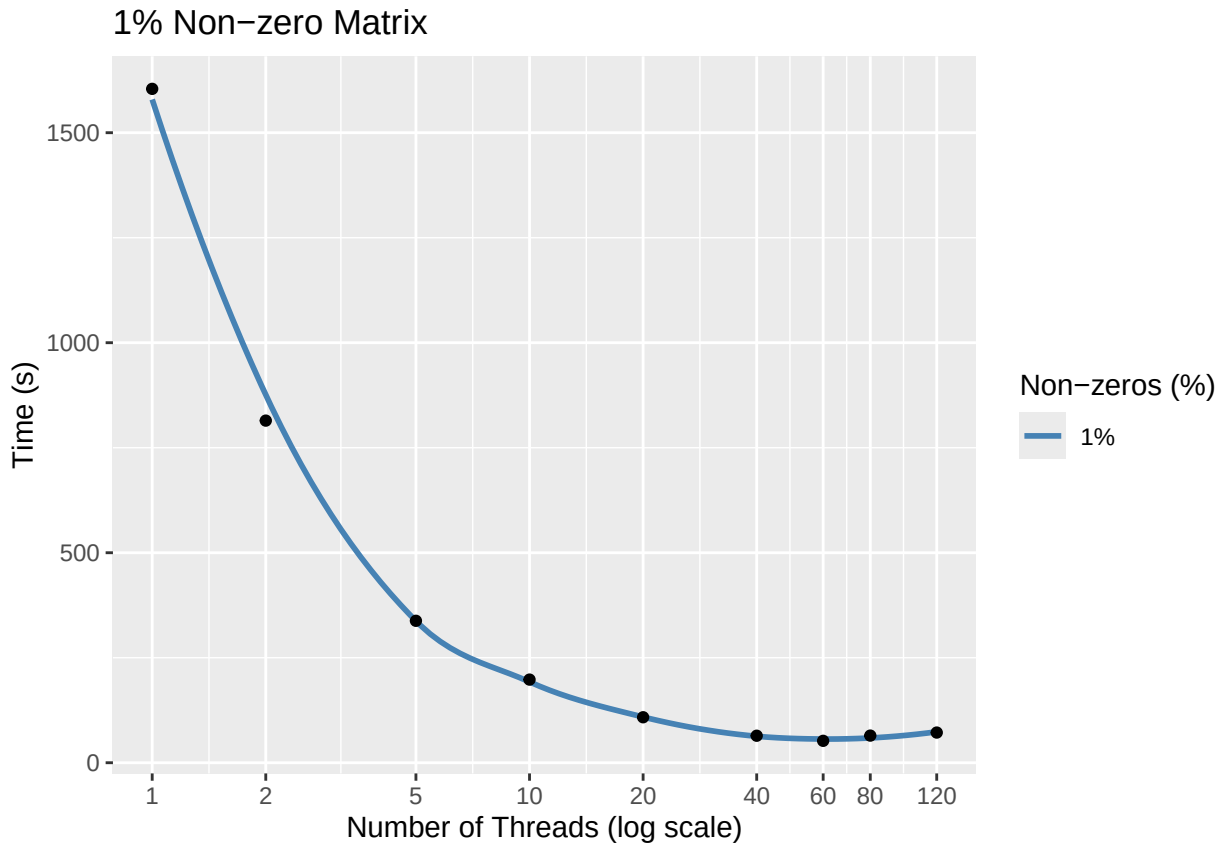
Once the matrices are loaded, the `compressedMatrixMultiply()` function is executed across varying thread counts, with OpenMP used to parallelize the operation. The execution time is recorded for each thread count using `omp_get_wtime()`, allowing us to evaluate the performance at different levels of parallelization.

Experiment 1 Result

For the 1% non-zero matrix, the minimum execution time occurred at 60 threads. This can be attributed to the relatively smaller computational workload, where the overhead of thread management and switching outweighs the benefit of utilizing more threads. When dealing with a smaller-size matrix like this, excessive threading can introduce inefficiencies, as the workload is too small to be effectively divided among many threads.

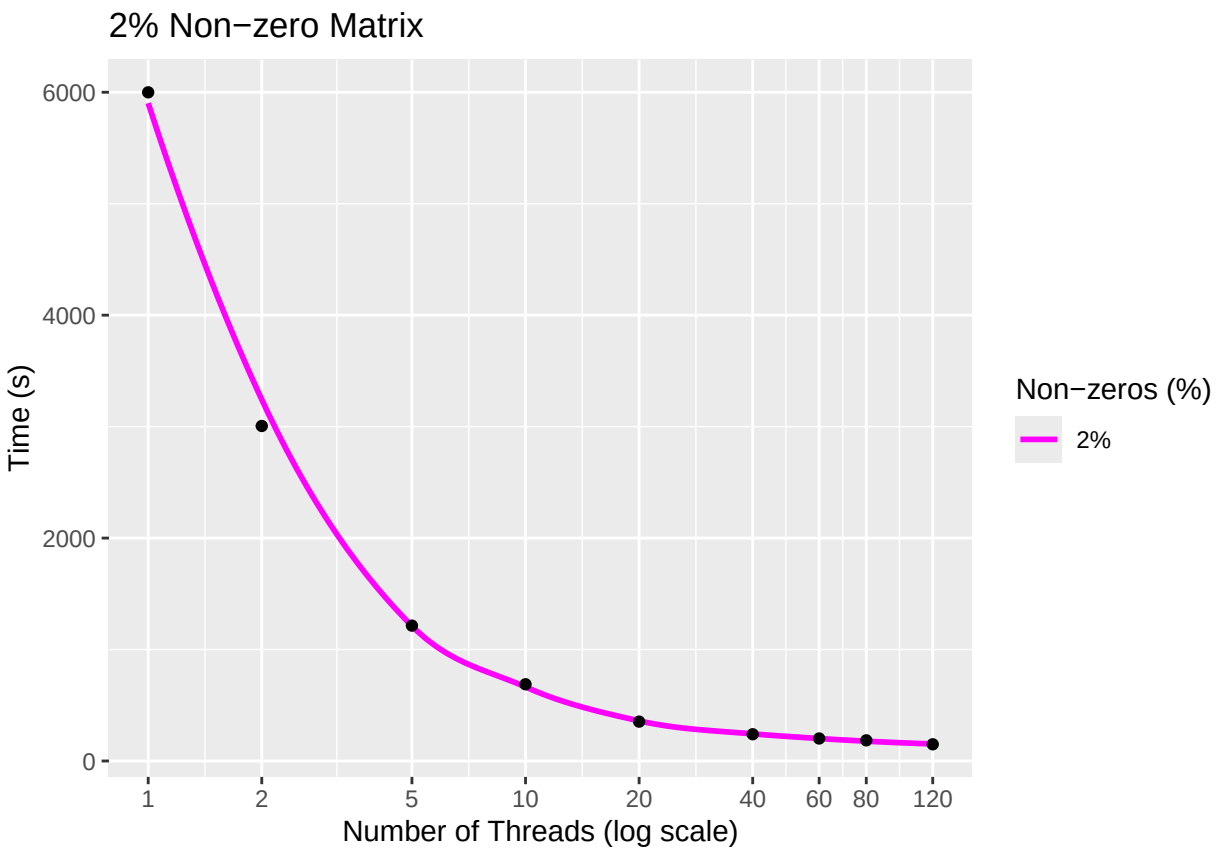
Non-zero Density	Minimum Time (s)	Optimal Thread Count
1%	52.1606	60
2%	149.679	120
5%	632.536	116

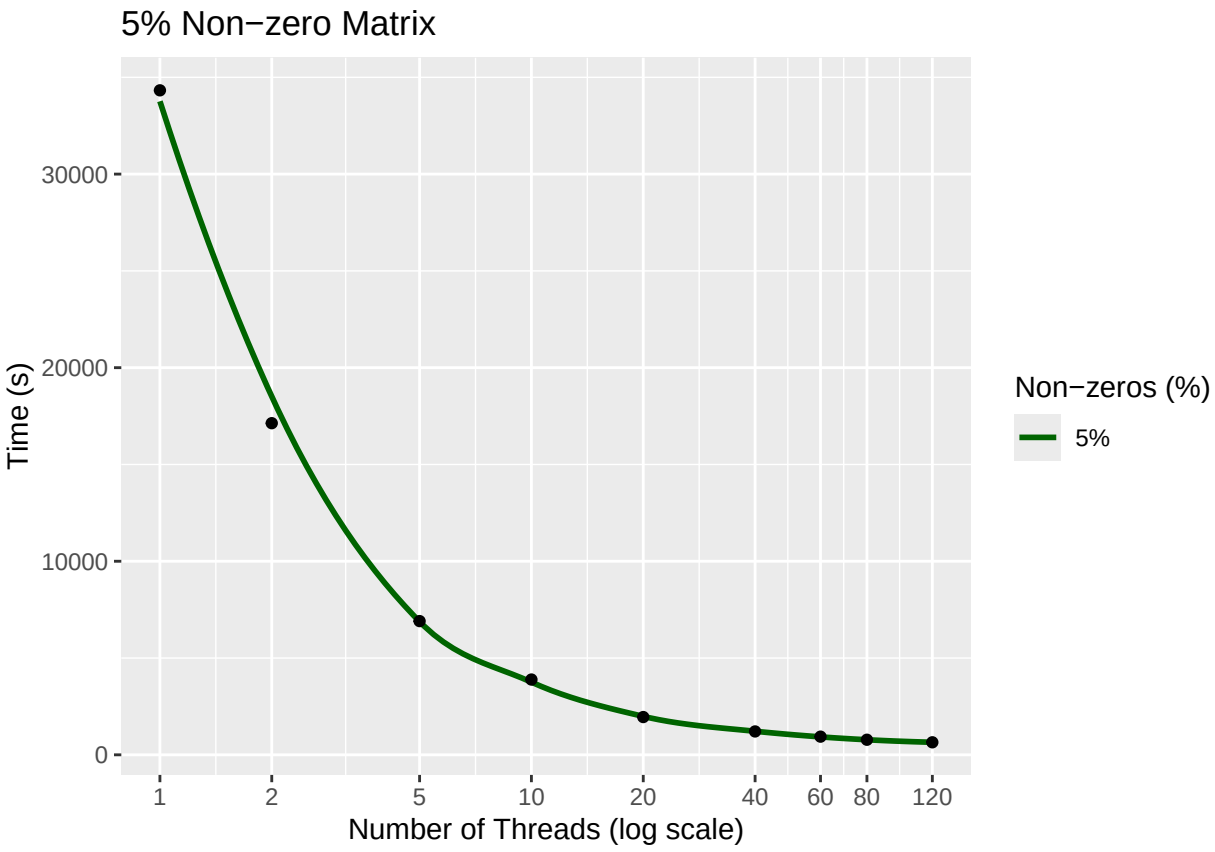
Some of the graphs below use logarithmic scales or uneven breaks on the X/Y axis to normalise the visual representation for easier comparison over a wide range of values.

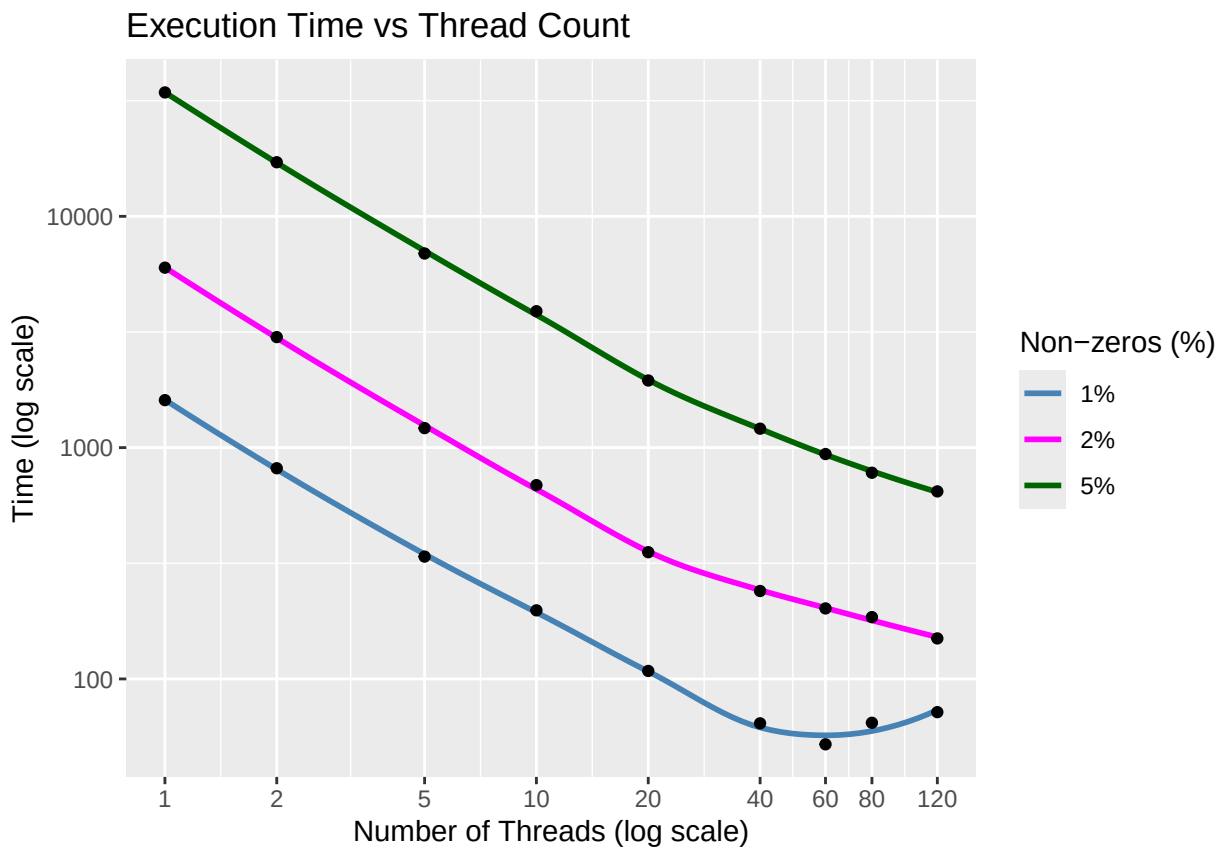


However, for the larger 2% and 5% non-zero matrices, the optimal performance was reached with 116-120 threads, which is the maximum thread count allowed by Pawsey for our experiments. The increased density of non-zero elements results in a larger computational workload, which can be more efficiently parallelized across more threads, reducing the overall time as more threads can handle the larger task in parallel.

Another plausible explanation is that for larger matrices like the 2% and 5% cases, the increased memory demands can hinder performance gains, thus requiring a higher thread count to achieve optimal performance. To address these memory-bound limitations, reducing memory consumption by using smaller data types, such as representing numerical values from 0-10 with char instead of int, might shift the local minimum execution time to a lower thread count.


2% Non−zero Matrix
6000
4000
2000
0
Time (s)
1
2
5
10
20
40
60
80
120
Number of Threads (log scale)
Non−zeros (%)
2%


5% Non−zero Matrix
Time (s)
30000
20000
10000
0
1
2
5
10
20
40
60
80
120
Number of Threads (log scale)
Non−zeros (%)
5%



Experiment 2

Experiment 2 tested how different OpenMP scheduling strategies (static, dynamic, guided, and runtime) impact performance in matrix multiplication. The 1% matrix was selected because it provided optimal performance at 60 threads in previous experiments.

Given the large matrix size (100,000 x 100,000), chunk size likely plays a significant role in the efficiency of each scheduling type. Therefore, additional experiments were conducted with chunk sizes of 100, 200, and 500, which seemed suitable for parallelization on 60 threads. The default chunk size, indicated by 0, served as a baseline reference.

For runtime scheduling, which relies on the `OMP_SCHEDULE` environment variable, we couldn't set its parameter explicitly, but it still provides a valuable comparison between different combinations.

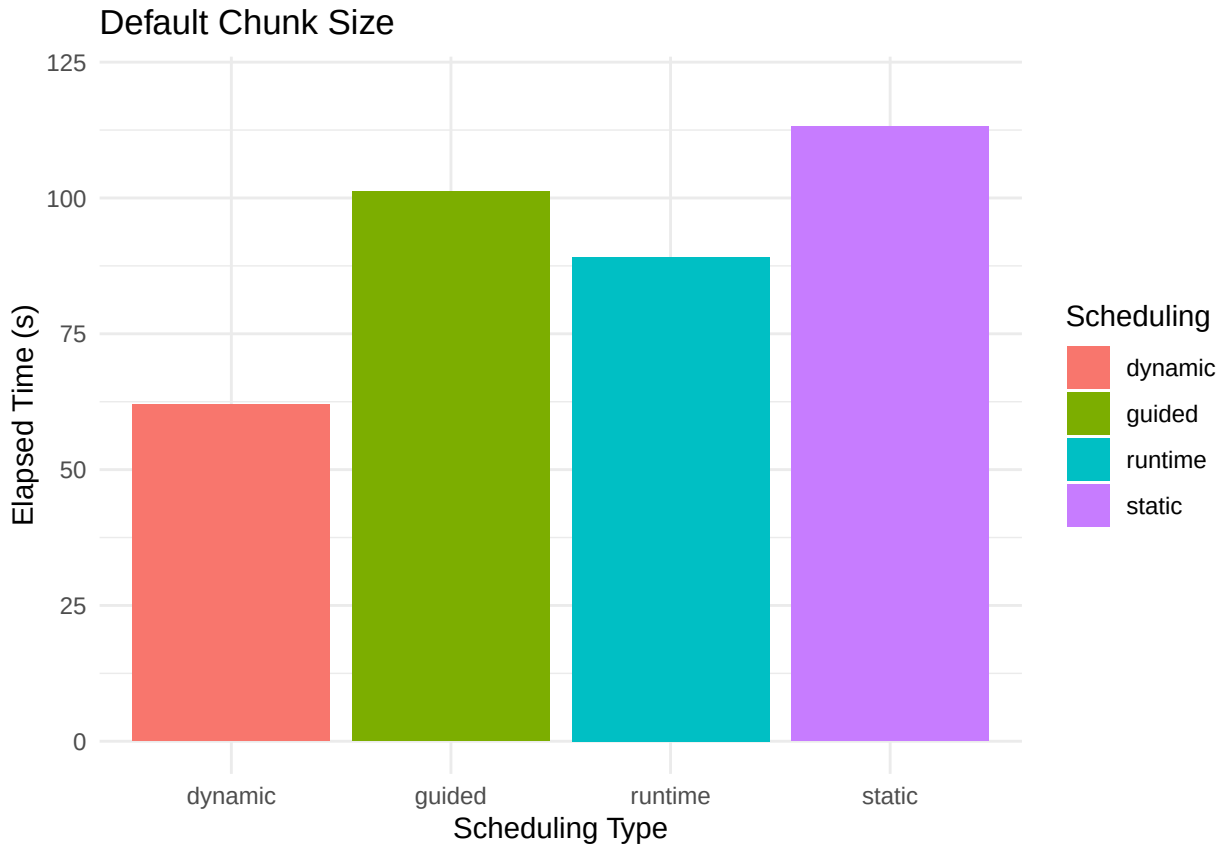
Experiment 2 result

At chunk size 0, which represents the default chunk size, the four scheduling strategies demonstrated distinct performance patterns. Static scheduling performed the worst among all, as its sensitivity to load imbalances caused slower performance compared to more adaptive strategies. Dynamic scheduling achieved the best performance, as it effectively minimized idle time by distributing smaller chunks of work to threads and ensuring balanced workloads. Runtime scheduling was the second fastest, likely due to the environment variables being well-optimized on Pawsey's system. Guided scheduling, while not as fast as other candidates, performed slightly better than static, maintaining a rather balanced workload distribution.

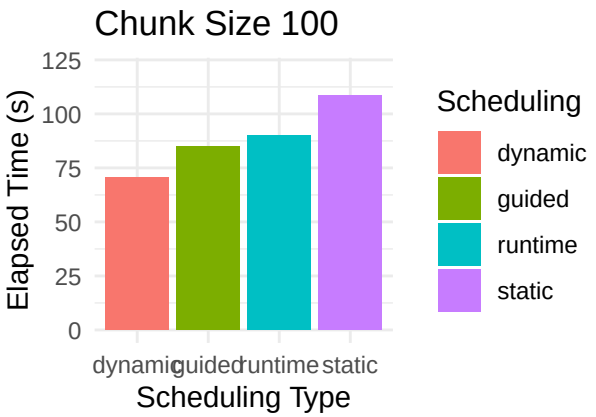
> The static scheduling task with the default chunk size took longer in Experiment 2 compared to Experiment 1. This can be attributed to variations in individual node performance and Pawsey's slower transfer speeds during the experiment on September 18th. To account for these

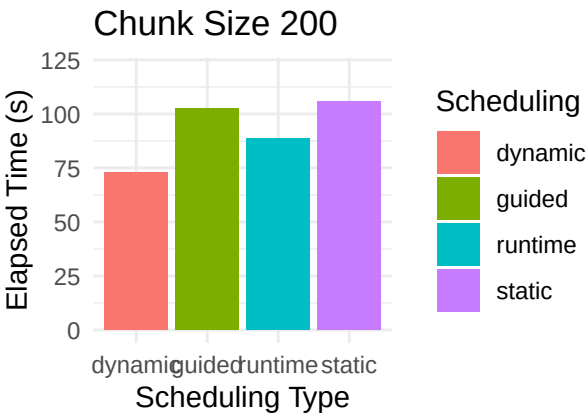
inconsistencies, we reran the static scheduling with the default chunk size in Experiment 2 for a more reliable comparison.

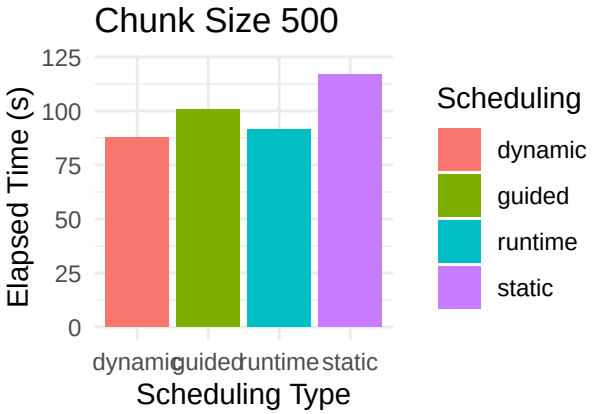
Scheduling Type	Time w/o Chunk size
static	113.135
dynamic	61.9773
guided	101.167
runtime	89.1588



In our side quest, we observed that the benefits of a more adaptive strategy diminished as chunk sizes increased. In the dynamic strategy, performance dropped significantly with larger chunk sizes due to the inability to evenly distribute the workload across threads. The guided strategy, however, performed better with a chunk size of 100 compared to the default, suggesting that the default chunk size may not be well-suited for larger workloads like this.


Chunk Size 100
Elapsed Time (s)
125
100
75
50
25
0
dynamic
guided
runtime
static
Scheduling Type
Scheduling
dynamic
guided
runtime
static


Chunk Size 200
Elapsed Time (s)
125
100
75
50
25
0
dynamic
guided
runtime
static
Scheduling Type
Scheduling
dynamic
guided
runtime
static


Chunk Size 500
Elapsed Time (s)
125
100
75
50
25
0
dynamic
guided
runtime
static
Scheduling Type
Scheduling
dynamic
guided
runtime
static