A Numerical Simulation of Heat Distribution: A Comparative Analysis of Serial and Parallel Implementations

Karolin Krzeminski
Dept. of Computing Sciences
Coastal Carolina University
Conway, SC, USA
kkrzemin@coastal.edu

Cole Morris
Dept. of Computing Sciences
Coastal Carolina University
Conway, SC, USA
camorri4@coastal.edu

***Abstract*—This paper investigates the performance differences between serial and parallel computing approaches through a numerical simulation program that simulates temperature distribution. By implementing and analyzing both serial and parallel (using OpenMP and Pthread) versions of a computation-intensive task, we measure overall execution, computation, and I/O times to assess work gains. The parallel version's scalability and efficiency are evaluated against the number of threads. Results demonstrate that while parallel computing reduces overall and computation time, the benefits reach a point of diminishing returns with increased threads. This study provides insights into optimizing computational tasks for parallel execution.**

I. Introduction

In computational engineering, numerical simulations serve as a tool for understanding and predicting the behavior of physical systems under various conditions. One application of numerical simulation is in the study of heat distribution within solid materials. This paper focuses on a numerical simulation designed to model the temperature distribution across a metal strip over time, a problem that is fundamental in materials science but also in applications ranging from aerospace engineering to electronics design.

The simulation employs a model of the metal strip, representing the temperature at various points along the strip as elements in an array. The temperature distribution is calculated iteratively, with each step of the simulation updating the temperature at each point based on the heat exchange with its immediate neighbors. This model captures the essence of heat distribution in a controlled, computationally efficient manner.

To explore the impact of computational strategies on the efficiency and scalability of such simulations, we implemented three versions of the program: a serial version and two parallel versions, one utilizing OpenMP, and the other Pthreads. The serial version operates sequentially, processing each point in the array, and serves as a baseline for performance comparison. In contrast, the parallel versions leverage the capabilities of multicore processors to update multiple points simultaneously, aiming to achieve a significant reduction in simulation time.

This paper details the development, implementation, and performance analysis of these three approaches. By comparing the execution times, speedup, and efficiency of the serial and parallel versions, we aim to provide insights into the benefits and limitations of parallel computing for numerical simulations.

II. Methods

The program is designed to conduct a numerical simulation, where elements of an array are iteratively updated based on the values of their neighboring elements. Three versions of the program were developed: a serial version and two parallel versions, one utilizing OpenMP and the other utilizing Pthreads. The core algorithm remains consistent across both versions, ensuring a fair comparison of execution times, speedup, and efficiency between the serial and parallel approaches. We will begin by describing the methods for our OpenMP implementation of the program.

A. Computation Functions for Parallel (OpenMP) and Serial

The serial version, *run_iterations*, performs the numerical simulation in a sequential manner. It operates on two arrays, A and B, each of size N. The simulation progresses through *numIterations*, with each iteration updating the elements of array A according to the average of each element's current value and the values of its immediate neighbors in array B. The updated states of A are then stored in a larger array, *all_data*, to keep a record of the simulation over time. A swap mechanism between arrays A and B at each iteration's end ensures that the updates are based on the newly calculated values.

Snippet 1: Serial Implementation

```
void run_iterations(double *A, double *B,
int N, int numIterations, double *all_data) {
    double *tmp;
    for(int i = 0; i < numIterations; i++) {
        for(int j = 1; j < N-1; j++) {
            A[j] = (B[j-1] + B[j] + B[j+1]) / 3.0;
        }
        memcpy(all_data + i * N, A,
        N * sizeof(double));
        tmp = A;
        A = B;
        B = tmp;
    }
}
```

The parallel version, *omp_run*, adapts the serial algorithm to utilize multiple threads for simultaneous execution, using the OpenMP library. The iterative update process is parallelized such that each thread is responsible for a portion of the array,

which distributes the computational workload. Synchronization mechanisms, including barriers and master-only sections, ensure that data consistency is maintained and that array swaps occur only after all threads have completed their computations for the current iteration. The parallel implementation aims to reduce the overall computation time by taking advantage of multiple cores available.

Snippet 2: OpenMP implementation

```
void omp_run(double *A, double *B, int N,
int numIterations, double *all_data) {
    double *tmp;
    for(int i = 0; i < numIterations; i++) {
        #pragma omp parallel for
        for(int j = 1; j < N - 1; j++) {
            A[j] = (B[j-1]
            + B[j] + B[j+1]) / 3.0;
        }
        #pragma omp barrier

        #pragma omp master
        {
            memcpy(all_data + i * N, A,
            N * sizeof(double));
            // print_array(A, N);
            tmp = A;
            A = B;
            B = tmp;
        }
        #pragma omp barrier
    }
}
```

B. Computation Functions for Parallel (Pthreads) and Serial

The serial version, much like the serial version used for the OpenMP version, performs the numerical simulation in a sequential manner. It operates on two arrays, A and B, each of size N. It uses a main function that will loop over each iteration, and performs the same numerical simulation in a sequential manner. The function progresses through *numIterations*, with each iteration updating the elements of array A according to the average of each element's current value and the values of its immediate neighbors in array B. For each iteration, the the array is printed, and arrays A and B are swapped. Once the double nested for loop is done, the function keeps a note of the work time that it took to compute both arrays.

The main function of the serial program first checks to make the sure that there are 3 to 4 arguments used: the value of N, the number of iterations, and an optional output file name. If there are more or less than 3 or 4 arguments, then the program will print what the command line statement should be before exiting in failure. Once the value of N and the number of iterations are stored, memory is dynamically allocated for arrays A and B. All the bytes in array A are set to 0, then the first and last elements of array A to 1.0, then the contents of array A are copied into array B. The compute function is then called, then the arrays are printed once the computation is done. If an output file was used, then the program will save all the computation work to the specified output file name. The program then returns the Overall, Work, and IO time that it took the program before exiting.

Snippet 3: Main Function for Serial Program

```
int main(int argc, char* argv[]) {
    double startOverall, endOverall,
    startWork, endWork,
    startIO2, endIO2;
    double elapsedOverall, elapsedWork,
    elapsedIO;
    GET_TIME(startOverall)
    if (argc < 3 || argc > 4) {
        printf("usage: %s <length>"
        "<numIterations> [<out file>]\n",
        argv[0]);
        exit(1);
    }
    int N = atoi(argv[1]);
    int numIterations = atoi(argv[2]);
    char* outFile = NULL;

    if (argc == 4) {
        outFile = argv[3];
    }

    double *A = (double *)malloc(N *
    sizeof(double));
    double *B = (double *)malloc(N *
    sizeof(double));
    size_t size = (size_t)N *
    (size_t)numIterations * sizeof(size_t);
    double *all_data = (double *)malloc(size);


    // Initial conditions
    memset(A, 0, N * sizeof(double));
    A[0] = 1.0; A[N-1] = 1.0;
    memcpy(B, A, N * sizeof(double));


    GET_TIME(startWork)
    run_iterations(A, B, N, numIterations,
    all_data);
    GET_TIME(endWork)

    GET_TIME(startIO2)
    if (outFile != NULL) {
        write_to_file(outFile, all_data, N,
        numIterations);
    }
    GET_TIME(endIO2)

    // Clean up
    free(A);
    free(B);
    free(all_data);
    GET_TIME(endOverall)

    elapsedOverall = endOverall-startOverall;
    elapsedWork = endWork - startWork;
```

```
    elapsedIO= endIO2-startIO2;

    printf("Serial␣Overall␣Time:␣%.6f\n",
    elapsedOverall);
    printf("Serial␣Work␣Time:␣%.6f\n",
    elapsedWork);
    printf("Serial␣I/O␣Time:␣%.6f\n",
    elapsedIO);

    return 0;
}
```

The Pthreads program has a similar compute function with some notable differences. Since this is the Pthreads version of the serial program, it uses a ThreadData pointer to repeat the computation for the given number of iterations. When the function reaches a certain point (the data barrier), it waits for all threads to reach that point.

Snippet 4: Thread Work Function

```
void* thread_work(void* arg) {
    thread_data_t *data = (thread_data_t*)
    arg;
    int start = BLOCK_LOW(data->thread_id,
    data->num_threads, data->N);
    int end = BLOCK_HIGH(data->thread_id,
    data->num_threads, data->N);
    double *A = data->A, *B = data->B, *tmp;

    for (int iter = 0; iter < data-
    >numIterations; ++iter) {
        for (int j = start + (data-
        >thread_id == 0 ? 1 : 0); j <= end
        - (data->thread_id == data-
        >num_threads - 1 ? 1 : 0); ++j) {
            A[j] = (B[j - 1] + B[j] + B[j +
            1]) / 3.0;
        }

        my_pthread_barrier_wait(data->barrier);

        if (data->buffer != NULL) {
            int copyStart = start;
            if (data->thread_id == 0) {
                copyStart = 1;
            }
            int copyEnd = end;
            if (data->thread_id == data-
            >num_threads - 1) {
                copyEnd = data->N - 2;
            }
            memcpy(data->buffer + iter *
            data->N + copyStart, A +
            copyStart, (copyEnd - copyStart
            + 1) * sizeof(double));
        }

        my_pthread_barrier_wait(data->barrier);

        tmp = A;
        A = B;
        B = tmp;
    }
    return NULL;
}
```

The main function of the Pthreads program performs mostly the same as the serial version, but with many notable differences since this is the parallel version. An additional argument is used for the thread count in the argument validation. The command line argument for the number of threads is then parsed into variables much like the other arguments. Memory is then allocated for the threads just as memory was allocated for arrays A and B. The barrier and thread structure are initialized to synchronize threads. Then the program performs the iterative computation across threads, and then the results are saved to an output file if there was one specified at the command line. The program then prints the Overall, Work, and IO time it took the program before exiting.

Snippet 5: Pthread Main Function

```
int main(int argc, char* argv[]) {
    ...

    if (argc < 4 || argc > 5) {
        fprintf(stderr, "Usage:␣%s␣<length>"
        "<numIterations>␣[<out␣file>]"
        "<number␣of␣threads>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    ...

    int num_threads;

    if (argc == 5) {
        outFile = argv[3];
        num_threads = atoi(argv[4]);
    } else {
        num_threads = atoi(argv[3]);
    }

...

    my_pthread_barrier_t barrier;
    my_pthread_barrier_init(&barrier, num_threads);

    ...

    pthread_t threads[num_threads];
    thread_data_t thread_args[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        thread_args[i] = (thread_data_t){i,
        num_threads, N, numIterations, A,
        B, buffer, &barrier};
        pthread_create(&threads[i], NULL, thread_work,
        &thread_args[i]);
    }

    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], NULL);
```

```
    }
    GET_TIME(endWork);
    elapsedWork = endWork - startWork;

...

 my_pthread_barrier_destroy(&barrier);

...
}
```

C. I/O Operations

The write is designed to write the simulation data along with its metadata to a specified binary file.

Snippet 6: Write Function

```
void write_to_file(char *filename,
double *data,
int N, int numIterations) {
    FILE *fp = fopen(filename, "wb");
    if (fp == NULL) {
        perror("Error opening file");
        return;
    }
    // Write metadata
    fwrite(&N, sizeof(int), 1, fp);
    fwrite(&numIterations,
    sizeof(int), 1, fp);
    // Write the entire 2D array data
    fwrite(data, sizeof(double), N
    * numIterations, fp);
    fclose(fp);
}
```

D. Timing Mechanisms

To analyze the performance, timing mechanisms were used at various stages of the program. This includes measuring the overall time, computation (work) time, and I/O time, using the **GET_TIME** macro.

E. Testing Differences Between Results

In order to ensure that the serial and parallel implementations produce consistent results, a *mydiff* program was created. This program is specifically designed to compute the Total Sum of Squared Errors (TSSE), a statistical measure used to assess the variance between corresponding elements of two arrays, denoted as matrixA and matrixB. A TSSE of zero indicates that the two arrays are identical. Any non-zero TSSE value signals a variance, with larger values denoting greater disparities.

Snippet 7: mydiff function

```
    double tsse = 0.0;
    for (int i = 0; i < rowsA; ++i) {
        for (int j = 0; j < colsA; ++j) {
            double diff = matrixA[i * colsA
            + j] - matrixB[i * colsB + j];
            tsse += diff * diff;
        }
    }
```

Performance

Performance was measured by comparing the execution time of the numerical simulation in three areas: overall time, computation time, and I/O time. These measurements were taken under different conditions by varying the number of threads and the size of the computation values. The specific metrics used to evaluate performance include:

- **Speedup**: Calculated by dividing the execution time of the serial program by the execution time of the parallel program for overall, work, and I/O times. The formula used for calculation is:
$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$
- **Efficiency**: Calculated as the speedup divided by the number of threads (p), providing insight into how effectively the parallel resources are utilized. The efficiency formula is:
$$E = \frac{S}{P}$$

Experimental Setup

The experiments were conducted using values of sizes 500,000 to 4,000,000 in 500,000 increments, with varying iterations to maintain the overall work size. The number of threads was varied across 1, 2, 4, 8, 16, 32, and 64 to observe the impact of thread count on performance. The performance metrics were collected for each combination of value size, number of iterations and thread count.

The environment for the experiments was conducted on the Expanse supercomputer. The Expanse system is equipped with multi-core processors, providing a robust platform for executing parallel computing tasks.

By varying the conditions under which the numerical simulation was performed, we were able to analyze the effects of parallelization on computational efficiency and identify the optimal number of threads for different value sizes.

III. Results

To visualize the purpose of the program which simulates the temperature of a metal strip over time, a contour graph was created, view Figure 1. We will be comparing the experimental results of the OpenMP implementation against the Pthread implementation to see if there are any significant differences. Refer to Figure 2 - Figure 9.

IV. Discussion

Our results indicate a reduction in overall and computation times when transitioning from a serial to a parallel implementation for the numerical simulation. This is consistent with the theoretical advantages of parallel processing, where tasks are distributed among multiple threads, reducing the time required

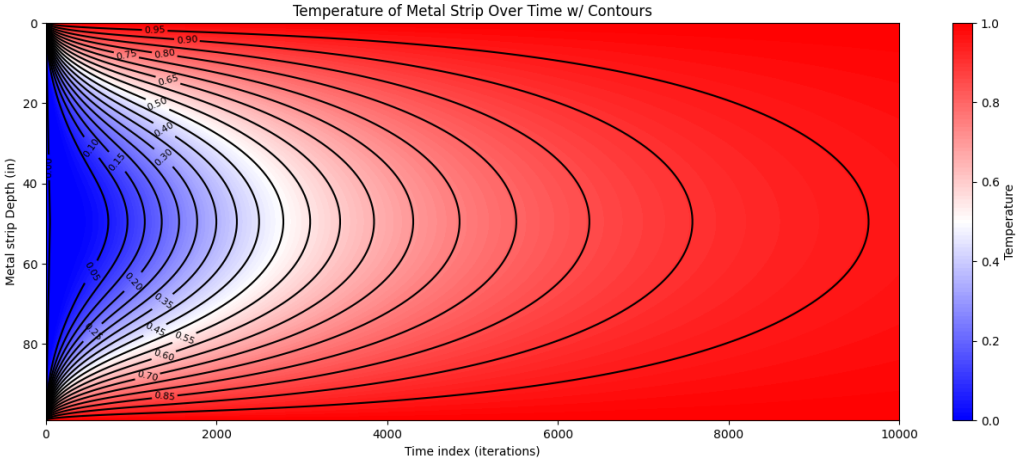

Figure 1: Temperature of Metal Strip Over Time Simulation - This image was created from a resulting binary file, where our program calculated the size value of 100, with 10,000 iterations. As the number of iterations increase, the metal strip gradually increases in temperature.

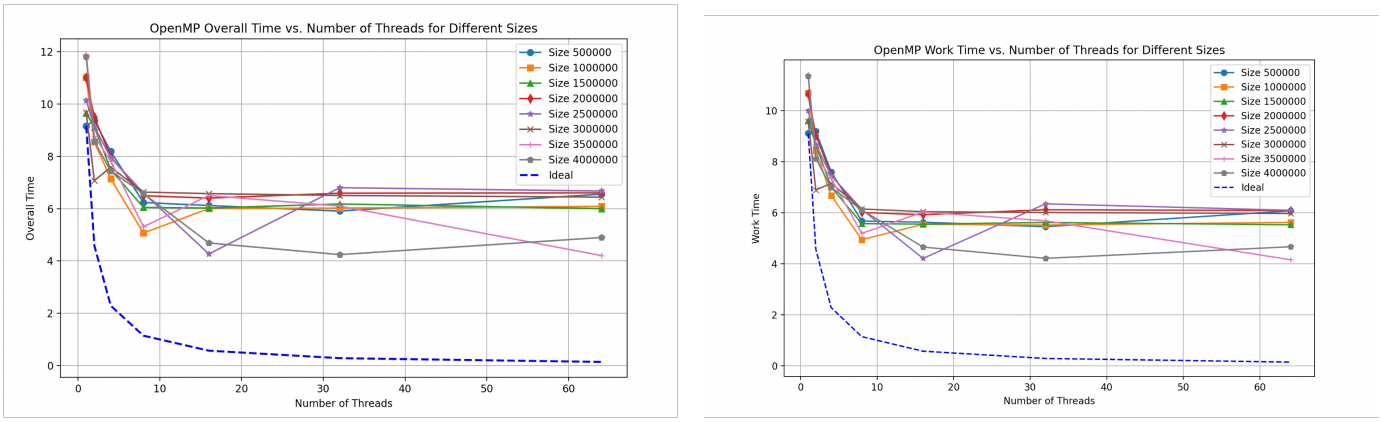

Figure 2: OpenMP Overall vs OpenMP Work Time - For all value sizes, the overall and computational time show a steady decrease as the number of threads increased from 1 to 8. After 8 threads, the time remains about the same for all value sizes.

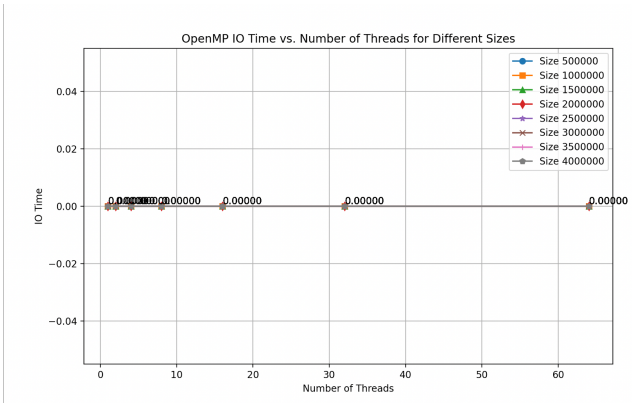

Figure 3: OpenMP IO Time - For all value sizes, I/O time remains at zero as writing to a file was excluded from our experiments, therefor I/O time is not applicable to the experiments run as it is not the portion that is parallelized.

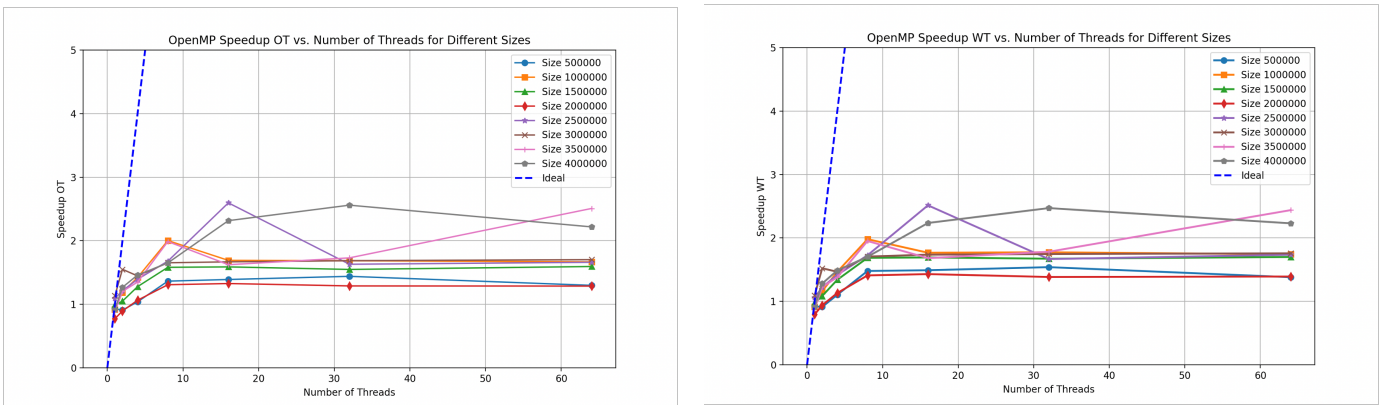

Figure 4: OpenMP Speedup Overall vs Work Time - For all value sizes, speedup increases for both overall and work time up to 8 threads, with diminishing results past this point.

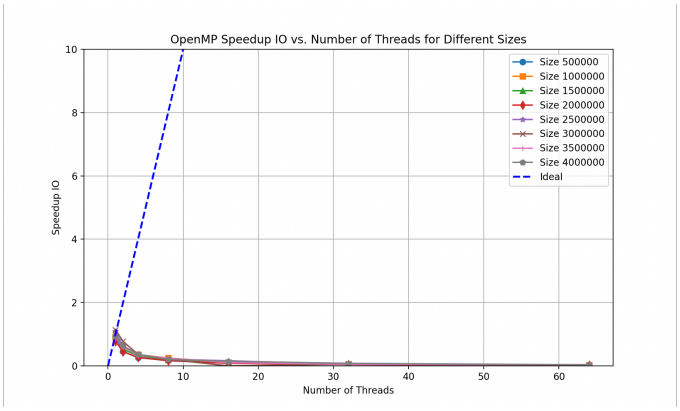

Figure 5: OpenMP Speedup I/O Time - For all value sizes, speedup for I/O time remains at zero as writing to a file was excluded from our experiments, therefor speedup for I/O time is not applicable to the experiments run.

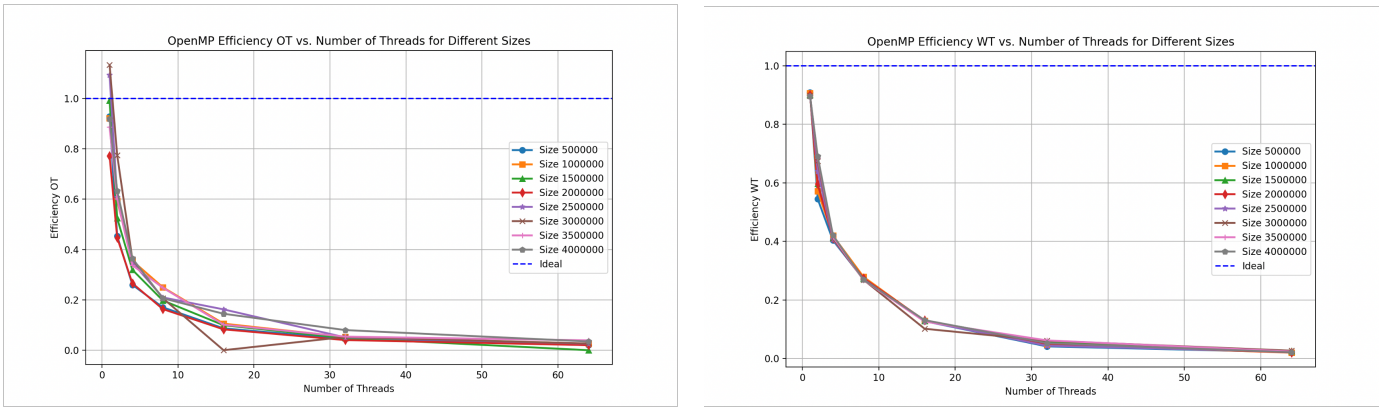

Figure 6: OpenMP Efficiency Overall vs Work Time - For all value sizes, efficiency for overall and work time gradually decrease as the number of threads used increases.

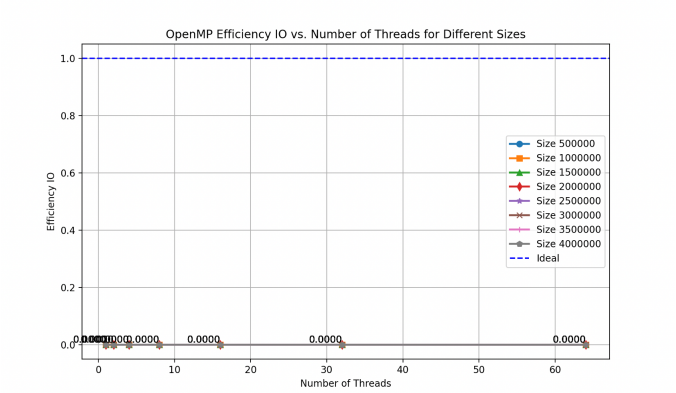

Figure 7: OpenMP Efficiency I/O Time - For all value sizes, efficiency for I/O time remains at zero as writing to a file was excluded from our experiments, therefor efficiency for I/O time is not applicable to the experiments run.

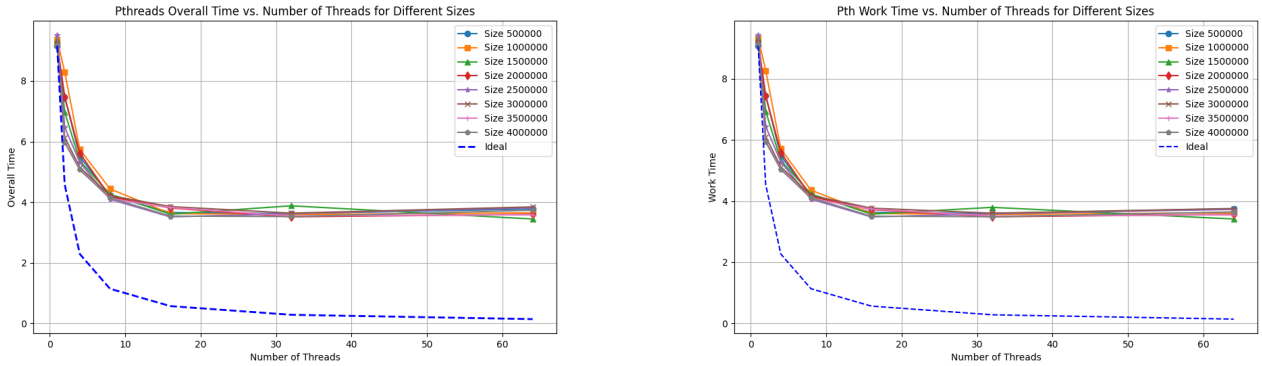

Figure 8: Pthread Overall Time VS Pthread Work Time- For N sizes, the overall time decreases as the number of threads increased from 1 to 4. After 4 threads, the time remains about the same for all value sizes.

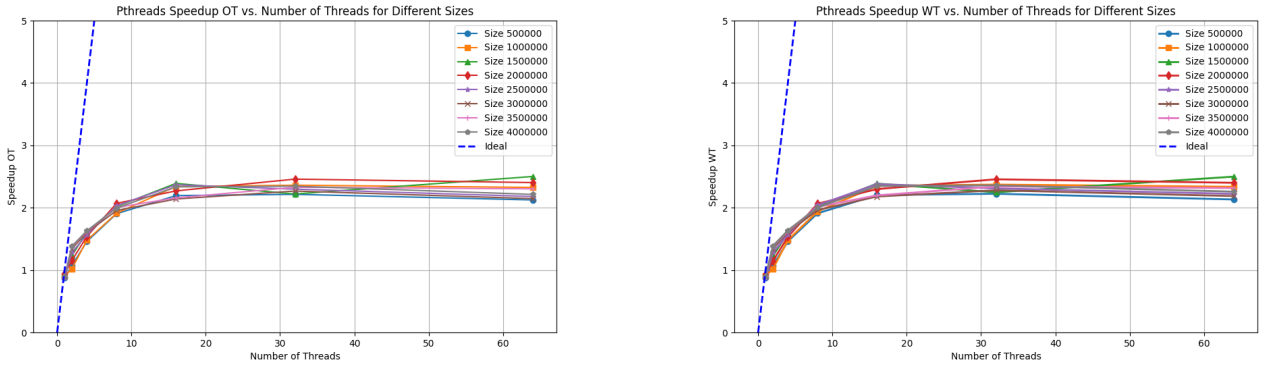

Figure 9: Pthread Overall Time vs Pthread Work Time Speedup - For thread sizes 1-4, the time goes down, but as more threads are used, the time goes up.

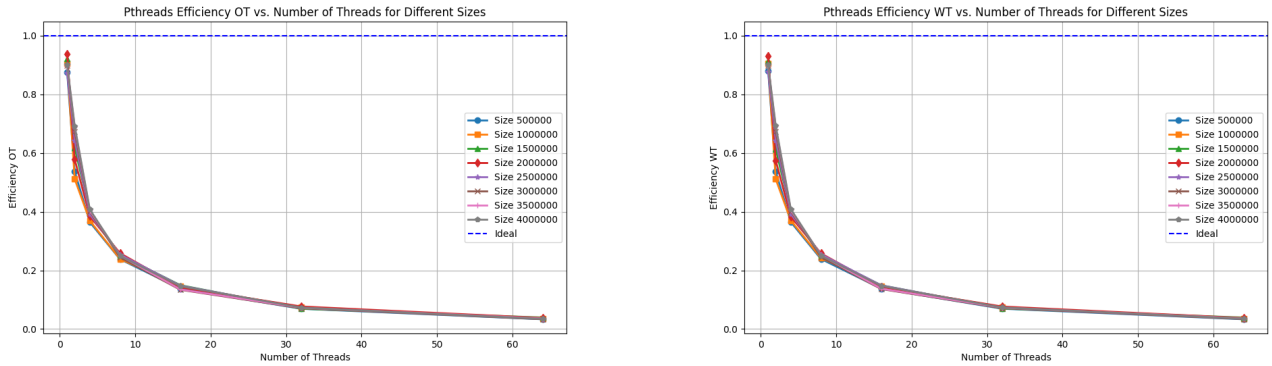

Figure 10: Pthread Overall Time vs Pthread WorkTime Efficiency - The efficiency rate for both the overall and work time decreases as more threads are used.

to complete the same amount of work. The OpenMP and Pthread implementations both showed improved performance over the serial approach.

Although, a point of diminishing returns was observed as the number of threads increased. Beyond 8 threads, the results diminished. This can be attributed to the overhead associated with managing a larger number of threads, which eventually outweighs the benefits of parallel processing.

In conclusion, this study demonstrates the potential of parallel computing to reduce the time required for computationally intensive numerical simulations. However, the benefits of parallelization must be balanced against the complexities of thread management and the diminishing returns associated with increasing the number of threads.

CONTENTS

REFERENCES

[1] P. S. Pacheco and M. Malensek, *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers, an imprint of Elsevier, 2022.

[**?**] [1]