# Showcasing the Impact of Differing Parallelization Techniques on a 1-Dimensional Heat Simulation

Devin Bucci
*Department of Computing Sciences*
*Coastal Carolina University*
Myrtle Beach, SC, USA
dtbucci@coastal.edu

Agit Yesiloz
*Department of Computing Sciences*
*Coastal Carolina University*
Myrtle Beach, SC, USA
ayesiloz@coastal.edu

***Abstract*—In this project, we explore the differing methods of parallelization of a 3-point 1-dimensional stencil simulation. The proposed simulation we opted to explore parallelization was heat transfer. Heat transfer is shown through a python script that showcases the temperature of a metal strip over time for a set length and time index. We first implemented a serial version of the heat diffusion simulation that measures computation time, optional IO time and overall time. Next, we opted to create another copy of the serial file which showcases the improvement parallelization is able to grant through the usage of omp directives. Then, we parallelized another copy through pthreading to showcase possible parallel techniques that are better/beneficial over another. Both methods of parallelization showcased speedup, even as we increased iterations, length of the array and especially thread amount (up to 16). All programs also showcased an optional file output due to limitations of memory allocation for testing where too large of executions showed failure during execution. We first tested for speedup on native windows machines under the linux subsystem, WSL, then moving on to test on the San Diego Supercomputer, Expanse. Shown through our results, we see a general speedup for both omp and pthreading. Expanding off of the execution, we discuss further the performance implications of pthreading versus that of openmp for parallelization. Overall, parallelization of both algorithms showcase speedup improvements and efficiency limitations.**

## I. Introduction

A one-dimensional three-point stencil algorithm is a computational method used to model heat diffusion. In this context, heat diffusion refers to the process by which temperature is able to spread through a metal plate over time. [1] The algorithm iterates through each time step (numIterations) and updates temperature based off of the relative adjacency each point has to each other and what it's associated temperature was recorded as. [2]In our case, the different arrays are swapped to avoid possible incorrect temperature values. By iteratively using this algorithm, the temperature distribution will generally increase then reach a level of equilibrium across the board for the metal plate (metal plate is simulated through our algorithm). [3]

The simulation is for heat diffusion but we opted to explore possible methods that can improve a serial implementation of the aforementioned. In this experiment, we used both pthreading and opemp.

Pthreads, or POSIX threads, is a method of program design that focuses on the idea that parallelization can be done by dividing a program into multiple threads of execution. These threads can run concurrently within a single process, allowing for simultaneous execution. [4] This allows for better management of resources for multi-core processors while improving overall performance and responsiveness of an application. This inherent splitting up of tasks "allow[s] the program to control multiple different flows of work that overlap in time. " (Wikipedia 2024) It's important to note that pthreading at least in the circumstance of a 1-dimensional stencil algorithm can showcase beneficial performance, but generally isn't the best method of parallelization. We use pthreading in this experiment, merely with the idea of comparing it to another popular method of parallelization; it is a thought experiment.

Expanding on the better method of parallelization is openmp. Openmp aims to simplify parallel programming by providing a high-level, API driven and user-friendly approach for shared-memory systems. For the purpose of this experiment, Openmp uses omp directives for parallelization with the use of one line. [5]

Openmp is generally a more hands off technique for parallelization while pthreading is more prone to race conditions from the nature of hands on.

Moving forward, this paper is organized as follows. Section II provides an outline for the design steps needed to deliver a serial heat simulation, a pthreading version and an openmp version. In Section III, the experiments are outlined and results are discussed to evaluate performance of the serial algorithm, pthreading algorithm and openmp algorithm. Section IV concludes the paper.

## II. Implementation and Design

To accurately discuss the importance of parallelization in regards to our stencil simulation, it is necessary to showcase the development that led us to that point.

To start, our experiment has the goal of showcasing the benefits of using parallel programming inside of a heat diffusion simulation with the usage of a 3 point stencil algorithm. We first need to setup a starting point typically done through that of a serial algorithm. This serial algorithm is done with a double nested for loop simulating an efficiency of O($n^2$). For the sake of reusability, we opted to use a "functions.c" file that stored any function capable of being used across several

different files. The idea in implementing the serial version of the heat simulation is to ensure the necessary parameters and build from there. Needed is the length of the array, N, and the number of iterations, numIterations. After using the parameters, we build off of it by implementing and initializing the arrays that we are going to use. Below are the two reusable functions for both:

```
void allocateAr(double**A, double **B, int N){
    *A = (double *)malloc(N * sizeof(double));
    if (*A == NULL) {
        fprintf(stderr, "Error A\n");
        exit(1);
    }
    *B = (double *)malloc(N * sizeof(double));
    if (*B == NULL) {
        fprintf(stderr, "Error B\n");
        exit(1);
    }
}

void initializeAr(double *A, double *B, int N){
    memset(A,0,N*sizeof(double));
    A[0] = 1.0;
    A[N - 1] = 1.0; // some initial values
    memcpy(B, A, N * sizeof(double));
}
```

After initialization and allocation, the serial computation can occur. Now, the function as mentioned, uses a double nested for loop where it is iteratively updated according to Array A based on the adjacent values. It uses double buffering to avoid future race conditions.

The function also implements a double buffering technique as shown through the swapping arrays for data accuracy. After each iteration, the function will store the contents of array A into a buffer that is used in the one time file write after the function call. Below is the function used for the computation and shows file write with the buffer.

```
double *buff=malloc(N*numIter*sizeof(double));
for(int i=0; i<numIter; i++) {
    for(int j=1; j<N-1; j++) {
        A[j] = (B[j-1] + B[j] + B[j+1]) / 3.0;
    }
    if (filenameA != NULL) {
        memcpy(buff+i*N,A,N*sizeof(double));
    }
    print_array(A, N);
    tmp = A; // double buffer approach
    A = B;
    B = tmp;
    }
    if (filenameA != NULL) {
    write_to_file(fileA, buffer,N*numIter);
    }
```

It's important to also note that the output file after the write will contain necessary row and column metadata at the beginning of the file and is in binary format, row major.

After the development of the serial function, it was necessary to create a function that can read the data from the file and print it to the screen to ensure that the serial function is providing the correct data to our output file. Below is that function:

```
void print_File(char *filename) {
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Error opening %s\n", fname);
        exit(1);
    }
    int N, numIterations;
    fread(&N, sizeof(int), 1, file);
    fread(&numIter, sizeof(int), 1, file);
    double *data = (double *)...
    malloc(N *(numIter+1) * sizeof(double));
    for (int i = 0; i < numIter+1; i++) {
        for(int j = 0; j < N; j++) {
            fread(&data[i*N + j]...
             sizeof(double), 1, file);
            printf("%1.2lf ", data[i*N + j]);
        }
        printf("\n");

    }
    fclose(file);
    free(data);
}
```

Once the desired results are achieved for the serial function and is compared with data that was confirmed to be true, we moved to a parallel implementation. To experiment with threading and showcase performance impact, it is necessary to display the scope of experiment along with necessary parameters for execution. Parameters were first listed in small 2 digit amounts for both length and iterations to test for accuracy then moved to larger million sized length with tens of thousands for number of iterations. Along with these parameters we used a thread amount of 1 to 64 increasing in a multiple of 4 each time. Once parameters were established and tested natively to ensure the accuracy of the native function at larger amounts we would be able to move to using a parallel version. [6]

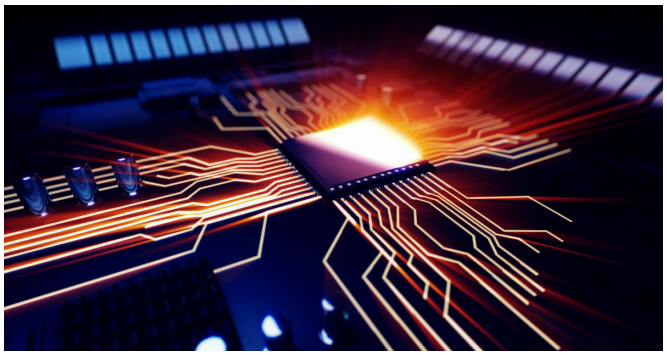

Fig. 1. San Diego Super Computer

Moving on to our first parallel version of the serial algorithm, we opted to use the openmp library with omp directives for parallelization. The program is generally the same except for the fact that, seperate threading functionality has been added to

parallelize the computational work inside of our main function. We opted to call the omp file, "omp-midterm-question.c", it takes 4 arguments two for length and iterations, one for the output file and one for the amount of threads to be used. As mentioned previously, it uses the openmp library to achieve parallelization.

Openmp is an acronym for Open Multi-Processing. It is a directive-based API (Application Programming Interface) used for developing parallel programs. It uses compiler directives, which is a statement or command used to force the compiler to commit a specific action during the compilation of code. [7] These compiler directives specify what type of code should be generated, for example one such directive is "pragma omp parallel for" which would then parallelize a for loop within its bounds. It does the threading and parallelization behind the scenes within the API. Our specific compiler directive is shown below:

```
#pragma omp parallel for
```

The directive will automatically parallelize what is within its bounds using a determined amount of threads called from the command line. The statement was placed within the second loop for accurate parallelization as parallelizing the outer loop will not affect the results for this particular experiment.

Once the implementation of the omp parallel version was completed, we opted to test for difference using a new program inside of our directory. This program finds the total sum of squared errors and the average percent relative errors, if there are none it will print out to the terminal window that the data being compared is identical. Below is a part of the implementation for this function:

```
fileA = fopen(argv[1], "rb");
if (fileA == NULL) {
    printf("Cannot open file %s\n",argv[1]);
    return 1;
}

fileB = fopen(argv[2], "rb");
if (fileB == NULL) {
    printf("Cannot open file %s\n",argv[2]);
    return 1;
}

while(fread(&valA,sizeof(double),1,fileA)&&
fread(&valB, sizeof(double), 1, fileB)) {
    double diff = valA - valB;
    TSSE += diff * diff;
    AVGPRE += fabs(diff) / fabs(valA);
    count++;
}

AVGPRE /= count;
```

To ensure the serial and omp versions are producing the same results, the "mydiff.c" program was created. Upon testing, we are able to determine that the upon using the same parameters for both omp and serial we can see that mydiff.c produces that there is no difference between the two. Below is the mentioned equivalency between the two .raw files "A.raw" being from the serial code and "C.raw" being from the omp code. ci

```
CSCI473/HW05/Code$ ./mydiff B.dat C.dat
The matrices are identical.
```

Upon completion of the omp version of the code, the experiment shifts to use another method of parallelization.

The "pth-midterm-question.c" program orchestrates parallelization of our 1-dimensional 3-point stencil algorithm through dividing the computation by thread amount. It operates by setting a unique identifier for each thread then calculates the workload for an available thread with the variable "local_m". The remainder is then calculated to ensure that parallel work occurs even if there is uneven distribution. The next variables are to show the range of data each thread will encompass and the for loop then parallelizes the computation from the previous serial version. The function also has implementation for writing to a file with the filenameA being passed in but for executing the program this won't be used.

A previous iteration of the parallel computation consisted of spawning thousands of threads and forking them together after each iteration which led to a heavy decrease in computational speedup/efficiency. The algorithm now spawns threads once and joins them once after being called. With the usage of this implementation it would be easier to run into race conditions so to solve this issue we opted for the usage of barriers. Barriers ensure synchronization where before continuing all threads reach a synchronization point. They are used before several key areas where thread synchronization is pertinent. To avoid the unnecessary waste of computation work we used one thread to swap the pointers for the associated arrays A, B and tmp.

Ideally, this would speedup computation time as threads are increased. Below is the parallelized version for pthreading.

```
void *thread_Function(void* rank) {
    ThreadArgs *thread_args = (ThreadArgs *)rank;
    //double *buffer = thread_args->buffer;
    char *filenameA = thread_args->filenameA;
    long my_rank = thread_args->rank;

    double *buffer = NULL;
    if (filenameA != NULL) {
        buffer = malloc(N * numI * sizeof(double));
        if (buffer == NULL) {
            fprintf(stderr, "Allocation Fail. \n");
            exit(1);
        }
    }


    // Determines range for threads
    int local_n = BLOCK_SIZE(my_rank, thread_count,
    N-2);
    int myfirstrow = BLOCK_LOW(my_rank, thread_count,
    N-2) + 1;
    int mylastrow = my_first_row + local_n - 1;

    for (int iter = 0; iter < numI; iter++) {
```

```
      for (i = myfirstrow; i <= mylastrow; i++) {
        A[i] = (B[i-1]+B[i]+B[i+1])/3.0;
      }
      if (filenameA != NULL) {
        pthread_barrier_wait(&barrier);
        memcpy(buffer + iter * N, A,
        N * sizeof(double));
      }

      pthread_barrier_wait(&barrier);
      if (my_rank == 0) {
        double *tmp = A;
        A = B;
        B = tmp;
      }
      pthread_barrier_wait(&barrier);
    }

    pthread_barrier_wait(&barrier);
    if (filenameA != NULL && my_rank == 0) {
      write_to_file(filenameA, buffer, N*numI);
      free(buffer);
    }
    return NULL;
}
```

Quinn macros were effectively utilized to parallelize the program of updating an array based on the values of its neighboring elements. By assigning each thread a specific portion of the array to process, you ensured an even distribution of workload. The use of BLOCKSIZE, BLOCKLOW, and BLOCKHIGH macros allowed each thread to identify its respective segment of the array, thereby applying simultaneously to different sections of the array without race conditions. This approach not only optimized the computation by leveraging parallel processing but also provided a practical application of parallel programming principles, showcasing how theoretical concepts can be implemented to achieve significant improvements in performance.

After designing both the serial and parallel code needed to test for performance improvement and an accurate heat plate diffusion, we opted to move from natively testing the code on our own separate machines to something that can handle higher thread amounts. The switch from using a native machine to something akin to a supercomputer is due to the fact hardware limitations could impact performance or that peformance in general could be vastly different from machine to machine. So for the sake of consistency and algorithmic speedup, we opted to use the San Diego Supercomputer, Expanse.

Expanse is a supercomputer based out of San Diego that is a cluster of ”13 SDSC Scalable Compute Units” designed to deliver at its peak a performance of 5.16 peak petaflops. Each expanse node has access to a parallel file system consisting of 12 petabytes and a storage system consisting of 7 petabytes. We opted to run our files on the compute node under the debug partition. The capabilities of the compute node are listed as such: 728 nodes, 2 sockets, 64 cores, 2.25 GHz, 4608 GFlop/s, 256 GB DDR4 Dram, etc (Expanse retrieved 2024).

To run our experiments as stated previously, we used the Expanse supercomputer within the shared partition. We submitted a batch job using our run_experiments.sh script that automated the process of loading modules needed for execution and the actual steps needed to run all of our programs at once. Below is a snippet of the code from our sbatch script:

```
i=0
iter=(7300...58400)

for((N=4000000;N>=500000; N-=500000)); do

    iter_i=${iter[$i]}
    ./midterm-question $N $iter_i
    for ((P=1; P<=64; P*=4)); do
        echo "N=$N, P=$P, iter=$iter_i"
        ./omp-midterm-question $N $iter_i $P
        ./pth-midterm-question $N $iter_i $P

    done
    ((i++))
done
```

All of the SBATCH directives are for expanse to know what is needed for processing this script, it shows the memory we need, the account we are using, the output file name, etc. Shown in the script is a makefile (“make clean all”) which was created with the purpose of compiling all of our files at the same time to mitigate the amount of unnecessary manual work done by the user. It also will remove all .raw files after all necessary comparisons. It links all of our source files together with shell commands written in the file, shown below is the exact Makefile we used for this project.

```
CC=gcc
CFLAGS=-g -Wall -Wpedantic -std=c99
LDFLAGS=-lm -lpthread
PROGS= midterm-question ...  mydiff

all: $(PROGS)

functions.o: functions.c functions.h
$(CC) $(CFLAGS) -c functions.c -o functions.o

midterm-question: midt.c functions.o
$(CC) $(CFLAGS) -o ... $(LDFLAGS)

omp-midterm-question.o: ...
$(CC) -fopenmp $(CFLAGS) -c ...

omp-midterm-question: omp-midterm functions.o
$(CC) -fopenmp $(CFLAGS) -o ... $(LDFLAGS)

pth-midterm-question: pth-midterm functions.o
$(CC) $(CFLAGS) -o ...$(LDFLAGS)

mydiff: mydiff.c
$(CC) $(CFLAGS) -o mydiff mydiff.c $(LDFLAGS)

clean:
rm -f $(PROGS) *.o *.raw
```

## III. Experimentation and Results

Moving on to experimentation and results, the objective we set out to meet is to display different parallelization techniques while hopefully being able to speedup the serial technique of a heat plate diffusion simulation.

It's important to note, at least briefly until results, that caching plays a roll in the complete performance speedup of the algorithms as limits of cache size may affect the direct speedup of the algorithm.

All performance results were measured in the shared partition on the shared node of the expanse supercomputer. No inconsistencies were recorded with regards to *where* the testing occurred. A simple "run_experiments.sh" script was used to run all the instances of each length, iteration amount and thread size.

### A. Experimentation

So, we used the file generated by the bash script which created a .out file and transcribed all the data into a csv file which could be automated to be ran with a python script. The python script would then output graphs needed to showcase improvement or negative performance of efficiency, time or speedup. Shown is the Overall OMP Time versus the Overall Pthreading Time. Both graphs are parallelized just through different aforementioned ways.

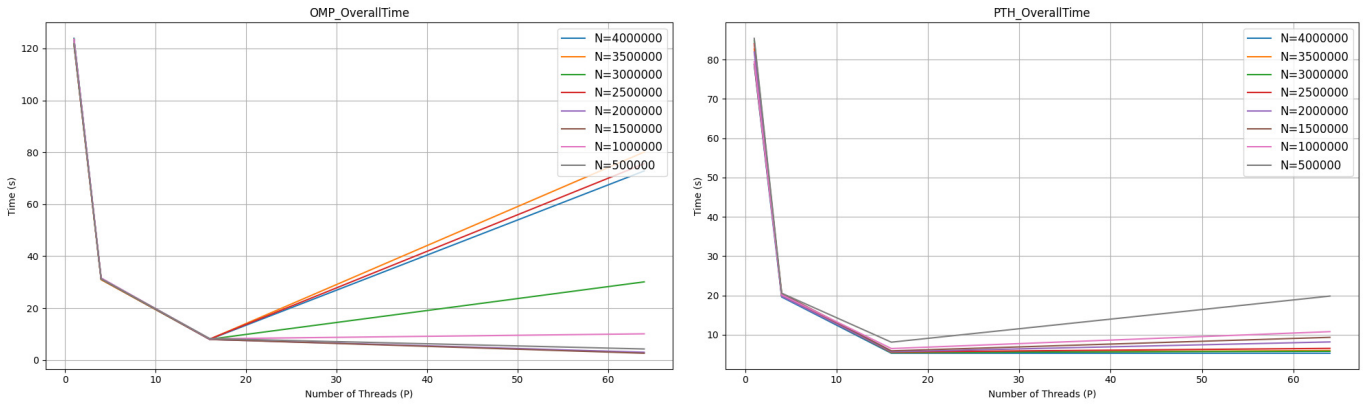


Fig. 2. The figure above shows the overall time it takes to execute a 1-Dimensional 3-Point heat simulation stencil with increasing threads for both omp and pthreads.

Figure 2 shows Pthreading has a higher rate of performance than that of OMP. The curves follow a very similar path as threads increase time is lowered. The figure shows however, when OMP reaches a thread amount of 64 time to execute separates depending on array size. Generally higher iteration amounts showcase worse performance and higher time while lower iteration amounts showcase similar performance to pthreading.

After the creation of the time graph, speedup is able to be calculated for both OMP and Pthreads and subsequently created into a graph to be viewed visually.

The speedup formula is given by:

$$S_p = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

We then used the formula to visualize overall speedup for both OMP and pthreading, shown below is figure 4.

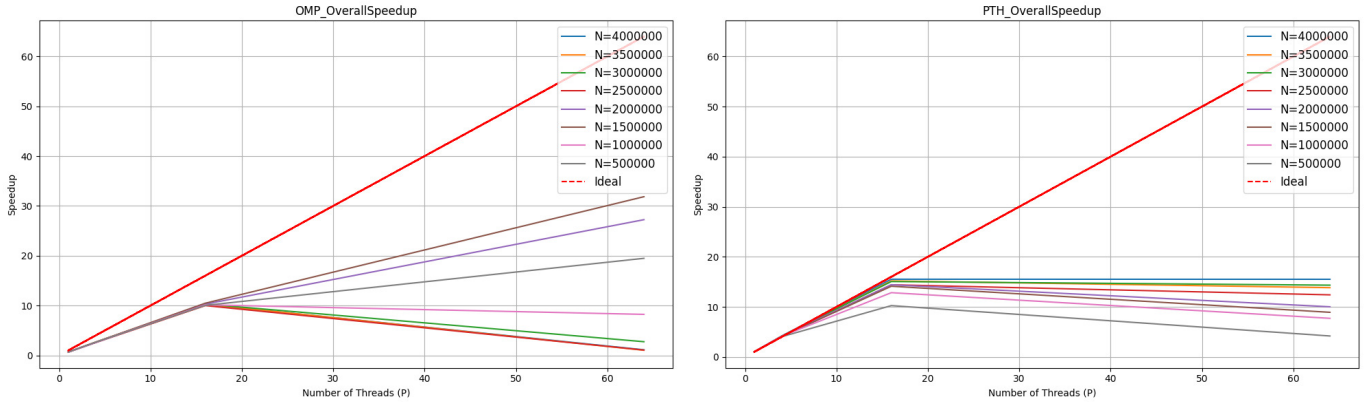


Fig. 3. The figure above shows the overall speedup it takes to execute a 1-Dimensional 3-Point heat simulation stencil with increasing threads for both omp and pthreads.

According to figure 3, it seems that speedup follows a general positive trend for both pthreading and omp until 64 threads. Speculation of possible reasons for this drop in productivity led us to possible cache contention or some thread overhead limiting speedup. Depending on the length given for the array , thread speedup was wildly different at 64 threads across the board for OMP. OMP shows negative trends for some array sizes while showing a linear positive trend for other arrays.

Pthreading was very similar to OMP, but showed less of a possible theoretical speedup than OMP. To elaborate, OMP showed possible speedup of 30x at its best while pthreading seemed to cap out at around 15-18x for all array sizes. Now, pthreading did show a much better job of staying consistent across all sizes for 64 threads but at it's best never beat OMP's best but on average did a lot better than OMP. Pthreading from lower thread amounts also showcased almost ideal level's of speedup but plateau'd at 64 threads.

We now move to look at the potential efficiencies of both, Efficiency is defined as the speedup being measured divided by the amount of processes being used. Note that this is overall efficiency for omp and pthreads. The efficiency ($E_p$) of parallelization can be calculated using the formula:

$$E_p = \frac{\text{Speedup}}{\text{Number of Processes}}$$

Using the efficiency formula, we use the previous data we gathered to create graphs for both omp and pthreads. Shown below in figure 4 are the graphs to visualize our programmatic efficiency.

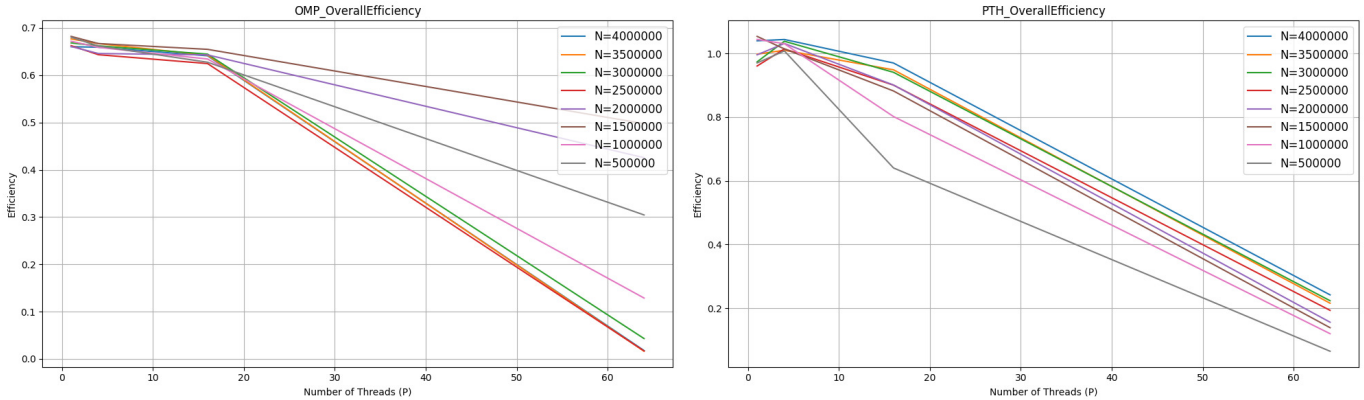


Fig. 4. The figure above shows the overall efficiency for execution of a 1-Dimensional 3-Point heat simulation stencil with increasing threads for both omp and pthreads.

Upon initial analysis of both graphs for efficiency, we are hit with a similar result with a clear winner. OMP starts roughly at an efficiency of 65-70% while pthreading starts around 100% even achieving slightly higher than that for some amounts of length. While note that due to Amdahl's law, a theoretical limit on speedup, it is impossible for the 100% efficiency to be a result of the algorithm. The algorithm is likely being improved through caching or better memory utilization. Pthreading shows a general linear negative trend with all of the array sizes following that example. Pthreading reaches an efficiency at 64 threads of around 30% to 10%. OMP showcases a much different trend where it is negative and linear generally, it changes

at 64 threads. At 64 threads for OMP, some array sizes keep an effciency of above 40% where other sizes go as low as just above 0% efficiency.

It's important to note that the timing, speedup and efficiencies were tested without the usage of IO for this particular test and the next few graphs will be referring to just the computational work. The computational work did not use the writing to file so the trend of graphs will be the same as above.

Moving on to the timing data for the computational work for both pthreading and OMP, results will likely be very similar as IO was not used during this execution. Below is figure 5 for the first of the graphs for computational work related to the time required for execution of both OMP and Pthreading.

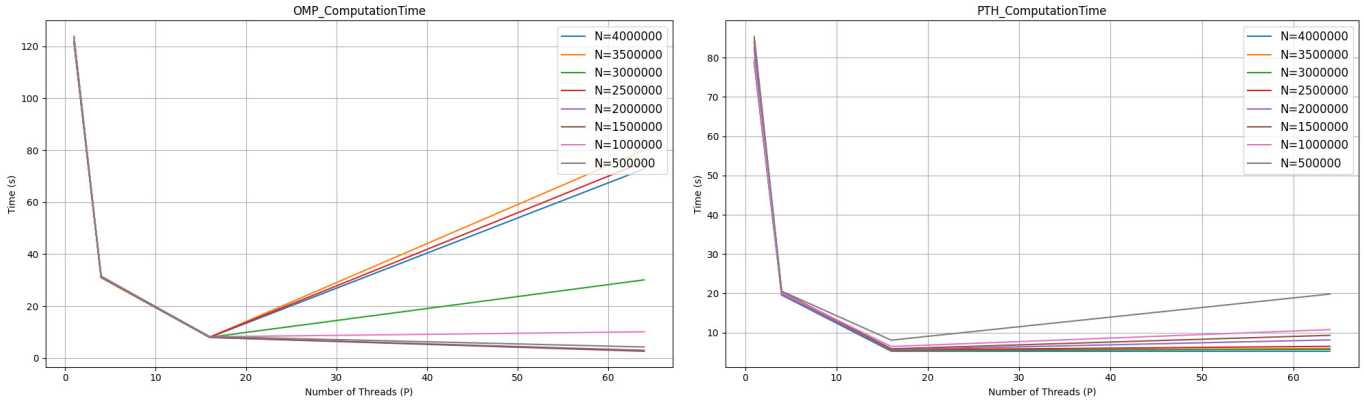


Fig. 5. The figure above shows the computation time for execution of a 1-Dimensional 3-Point heat simulation stencil with increasing threads for both omp and pthreads.

Shown above is almost the same exact graph as our overall time for omp and pthreading. As previously talked about, for OMP time it decreases exponentially until 64 threads. At 64, the algorithm seems to bottleneck at varying degrees of size and iteration. Perhaps, the method at how openmp handles this parallelization is causing the variations in performance. Generally lower size saw the most impact of a higher thread amount. Pthreading also once again showed improvement flat across the board, with a higher precision.

Moving on, we need to once again visit speedup but this time in regards to the computational work being done. It is the same formula however we have different graphs this time to showcase the visualization. Shown in figure 6 are the results of speedup in relation to increasing thread amounts for both omp and pthreads.

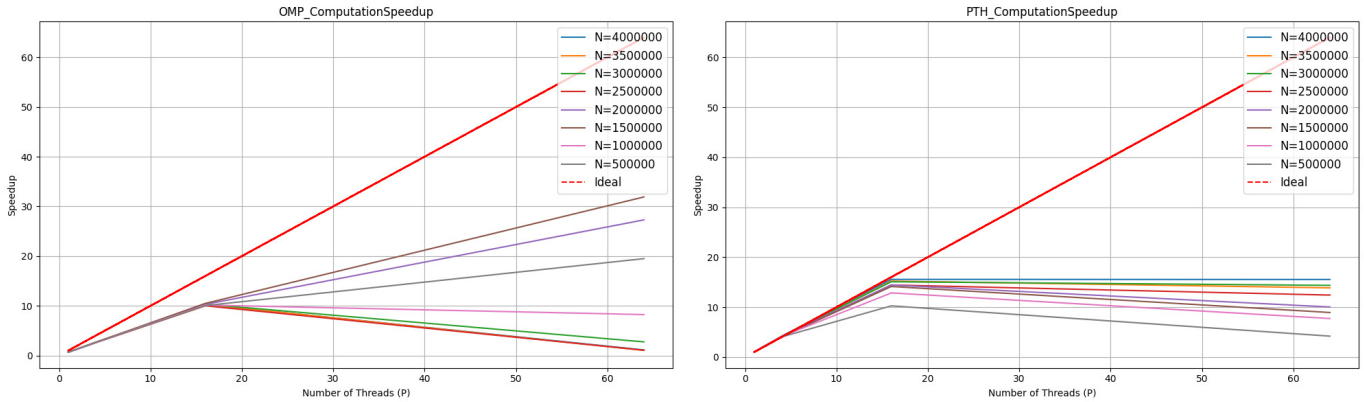


Fig. 6. The figure above shows the computation speedup for execution of a 1-Dimensional 3-Point heat simulation stencil with increasing threads for both omp and pthreads.

As shown in the figure, the computational speedup graph is almost identical to the overall speedup graph. Hence, to discuss once again the reasons for this result would be redundant. Shown in figure 8 is the computational efficiency graph and is also identical to the overall efficiency graph. So once again, it would be redundant and not necessary to explain the possible reasons for this. The reason for identical is because IO wasn't used in this test and hence memory allocation would then take less than a second to run.

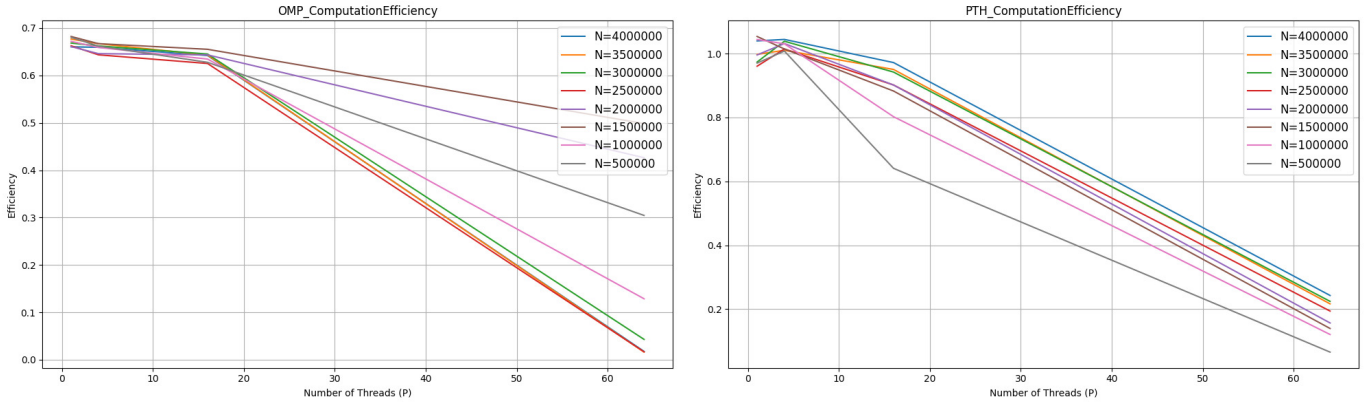


Fig. 7. The figure above shows the computation efficiency for execution of a 1-Dimensional 3-Point heat simulation stencil with increasing threads for both omp and pthreads.

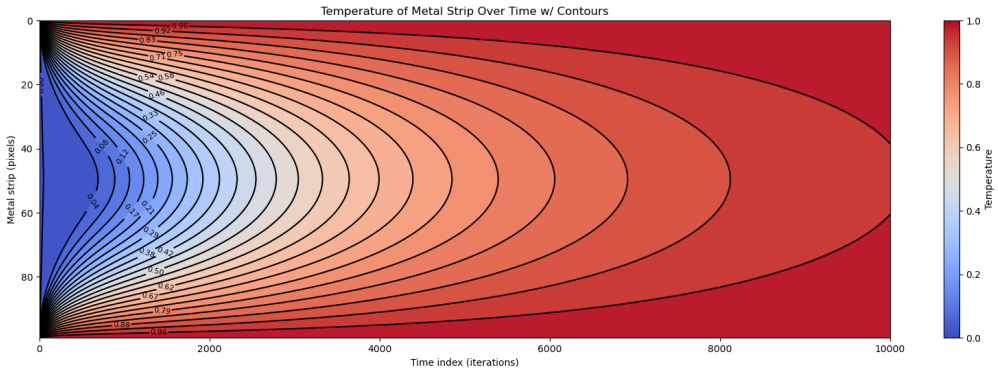


Fig. 8. The figure above shows the diffusion process where heat moves from warmer to cooler areas until a uniform temperature reached.

Contour plot displaying the temperature variation of a metal strip over time, which is likely the outcome of simulating a one-dimensional heat conduction problem using a stencil algorithm. In computer science, stencil algorithms are often used to model heat distribution or other physical processes where each point's value is updated based on neighboring values over discrete time steps.

On the x-axis, the 'Time index (iterations)', showing that the simulation is iterated over time—a standard approach in numerical simulations where the physical time is approximated by a series of time steps. The y-axis, labeled 'Metal strip (Pixels)', suggests that the metal strip's physical domain has been divided into a grid for computational purposes, each 'pixel' representing a discrete segment of the strip. [3] The contours are quite smooth, implying a steady and gradual transition of heat, which is typical in diffusion processes where heat moves from warmer to cooler areas until a uniform temperature is reached, assuming no sources or sinks of heat are introduced after the initial conditions. This kind of plot is a quick way to visualize and analyze the results of the simulation, giving insights into both spatial and temporal temperature dynamics of the system.

### B. Results

Results are within the scope of what we expected from this experiment. OMP showed a large increase in speedup whiel pthreading showed even better improvement. The only time a portion of our parallel programs was worse than the serial code was at 1 thread which would be a result from increased overhead as a result of all the unused threading operations. An earlier iteration of this project showed a general negative trend for pthreading reaching very low performance improvement, but now with the usage of sychronization techniques pthreading showcases great speedup. Pthreading, for computational speedup showcased a general trend that was very close to the ideal speedup until 64 threads. Granted, the experiment jumped in multiples of 4 so we don't have much information to go off of from 16 threads to 64. We can visually see that there was a massive plateau in speedup, efficiency and time comparing 16 threads and 64 threads.

A possible reason for this might be where we were able to execute the computation inside of Expanse. We had to use the Shared partition which only has access to 128 threads in totality. We had to execute this project alongside of other people, as it is the nature of the shared partition. Due to running large scripts alongside other large scripts from other users, if both were requesting access to say 64 threads it is extremely likely that we wouldn't see as good of speedup due to the nature of limited resources on a shared node.

Efficiency inside of this project had interesting results as well. At the beginning with 1 thread, efficiency for both overall and computational showed pthreading having higher efficiency than the serial. We directly observed superlinear speedup. Superlinear speedup is this theoretical speedup amount where the algorithm speedups more than it should be physically allowed to. It is only achievable from a hardware standpoint and is incapable of producing those results from strictly software. In our example, as we are running on the Expanse supercomputer, it is likely due to improvements of Cache or better memory utilization as a result of how the system is set up. As thread amount 64 generally limited the parallel programs, it shows the efficiency much lowered at that amount, which is expected.

OMP encountered that strong drop in performance for several different lengths of the array, measuring effiency close to zero and speedup lowered as a result. Possible speculation of reasoning for these results could be cache contention, thread overhead and even possible algorithmic design.

To explore more with cache contention, with an increase of thread amount there is a higher chance for cache thrashing where threads or cores compete to access resources inside of the cpu cache. This might result in cache invalidations that would lead to increased time, false sharing where different threads modify data that are in the same space in the cache and possibly more.

Algorithmic design of how Openmp handles large thread sizes for this particular algorithm might affect how higher number of threads are handled and if they are handled poorly, increased time to use the appropriate resources might occur. There might also be bottlenecks in performance as well as a result.

Something that was genuinely unexpected was how long it would take for the OMP to run the algorithm on 1 thread compared to serial. It was typically 30 seconds longer than that of the serial algorithm which is a strange result. Pthreading showed similar timing to serial but on early iterations showed the same changes of OMP. It's possible that as we changed the pthread code to use better synchronization practices, it changed the 1 thread handling for time. Whereas, OMP uses a single pragma directive where synchronization is done either entirely behind the scenes by openmp or not at all. OMP has pragma barriers that are able to synchronize threads but if used improperly can either break the algorithm or slow it down immensely.

## IV. Conclusions

Overall, this project sought out to see if a heat plate simulation was able to be parallelized in a way that improved algorithmic execution time. Not so suprisingly, both parallel versions of our program showcased faster execution time, speedup and efficiency. Openmp had a higher variability to it's time where depending on the number of iterations being called, the time to run was heavily affected being either faster or slower. Pthreading showcased a better precision where no matter the number of iterations, it would generally showcase improvement. The project focused on improvements of computational time but did explore IO at a local level to determine areas of improvement. Further study on areas of OMP being much faster than pthreads at different levels of iteration amounts need to occur in a following experiment. Any negative trends resulting from OMP were likely a result of the environment/node it was ran in or were a result of caching limitations, resource contention or algorithmic design. To finish, the project showed improvement against the serial design up to 16 threads even coming close to the ideal line for pthreading while 64 threads limited the rate of improvement for both parallel designs.

## REFERENCES


[1] P. H. Gunawan. (2016) Scientific parallel computing for 1d heat diffusion problem based on openmp. [Online]. Available: https://ieeexplore.ieee.org/document/7571960

[2] Wikipedia. (2024) Heat transfer. [Online]. Available: https://en.wikipedia.org/wiki/Heat$_{t}ransfer$

[3] C. E. Leiserson. (2010) Stencil computing. [Online]. Available: https://courses.csail.mit.edu/6.884/spring10/labs/

[4] randu. (2024) Multithreaded programming. [Online]. Available: https://randu.org/tutorials/threads/

[5] T. Mattson. (2024) Openmp in a nutshell. [Online]. Available: https://tildesites.bowdoin.edu/ ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html

[6] M. explained. (2024) Multithreading explained. [Online]. Available: https://www.ionos.ca/digitalguide/server/know-how/multithreading-explained/

[7] R. M. LEONARDO DAGUM. (1998) Openmp: An industry standard api for shared memory programmin. [Online]. Available: https://pages.cs.wisc.edu/ david/papers/ieeecse1998$_{o}penmp.pdf$