CSCD94 Project Report

Supervisor: Prof. Marcelo Ponce
Team: Pedram M. Haghighi & Tuan Ky Pham

Introduction

The International HPC Summer School (IHPCSS) is an annual workshop where there is a programming challenge at the end. The participants of these challenges try to improve the performance of some given program using the knowledge they gained from the workshop. In 2019, the IHPCSS challenge was a simulation of thermal diffusion on a metal plate of size 14560x14560 discretized units, and in 2024, it was a program that performs pagerank calculation on a graph of 1000 websites.

Our team, which consists of students from the summer course CSCD71, attempted these two challenges by running them on the Scinet Teach cluster and fine tuned them using performance improving techniques learned from the course, such as shared memory programming and distributed programming. In this report, we will go through our approaches and findings, as well as the challenges we faced, to gain significant performance improvement.

The Challenges / Teach Cluster

Both of the IHPCSS challenges begin with some starter code, but the level of content originally provided was vastly different. For the 2019 challenge, the starter code contained multiple implementations of the thermal diffusion simulation using the basics of different HPC techniques, such as distributed programming with MPI and utilizing accelerators with OpenACC. On the other hand, the 2024 challenge only provided a serial implementation of the pagerank program. This difference is significant for us because we can try a lot more things to gain performance in the 2024 challenge compared to the 2019 challenge. Other than the starter code, both challenges also provide some software ecosystem to work with C and FORTRAN.

Our high performance compute cluster, Scinet Teach, was not the cluster originally used for these challenges, so we had to make many modifications to the provided software ecosystem to run the programs. Since we only work with C, all FORTRAN related code was ignored or removed. The Teach cluster also does not have some compilers that the challenge wanted, so we swapped it for GNU compilers and used the compiler flags we deemed equivalent. The biggest problem with Teach, however, is that it only has 16 CPU cores per node and only lets you request 4 nodes per job, which led to us having to modify some job configurations in the 2019 challenge because we did not have enough resources. Despite that, we were able to run all programs for both challenges without any issue and were able to collect baseline performance measurements.

Optimization Philosophy

In tackling these projects, we employed a systematic approach to optimization. As Ruud van der Pas, coauthor of **Using OpenMP,** highlights in his presentation **Mastering OpenMP** :

> *"...scalability can mask poor performance (a slow code tends to scale better, but is often still slower)."*

With this in mind, our focus was not solely on scaling the algorithm but first on improving its intrinsic performance. Optimizing poor code and then scaling it ensures that the underlying inefficiencies do not carry over to the parallelized version. Thus, before leveraging tools and libraries to achieve parallelism, we scrutinized the single-threaded performance of the existing implementations.

Our optimization strategy involved the following steps:

1. **Baseline Optimization:** First, we improved the serial version of the code. This ensured that the algorithm was as efficient as possible before applying any parallelization techniques.
2. **OpenMP for Shared Memory Optimization:** Once we had an optimized serial version, we applied OpenMP to take advantage of shared memory and multi-core processing.
3. **MPI for Distributed Computing:** To scale beyond a single node, we applied domain decomposition and distributed computing principles using MPI.
4. **Hybrid OpenMP AND MPI:** To maximize performance, we combined OpenMP and MPI in a hybrid approach. OpenMP handled multi-threading within each node, while MPI distributed the workload across multiple nodes. This balance allowed us to fully utilize the computational power of each node while minimizing communication overhead between nodes.

IHPCSS 2024

Serial Code Optimization

The original implementation of the PageRank algorithm, although functionally correct, had several inefficiencies that could be improved to enhance both memory access patterns and the overall runtime. We identified and made the following optimizations:

Optimization 1: *Memory Initialization and Array Copying*
In the original code, loops were used for both array initialization and copying, which we replaced with `memset` and `memcpy` functions. While this did not yield a measurable performance gain, the change contributed to cleaner, more compact code, making it easier to manage. We initially expected that using library functions over custom loops would improve performance, but in C, the compiler may already optimize these cases efficiently.

Optimization 2: *Refactoring the PageRank Update Loop*
One of the largest bottlenecks in the original PageRank algorithm was the triply nested loop responsible for calculating the new PageRank values. The inner loop, which calculates the outdegree of each node, was particularly inefficient. We restructured the loop to eliminate one level of nesting and reorganized the matrix traversal to row-major order to reduce cache thrashing.

Original nested loop:

```
for(int i = 0; i < GRAPH_ORDER; i++) {
    for(int j = 0; j < GRAPH_ORDER; j++) {
        if (adjacency_matrix[j][i] == 1.0) {
            // Additional inner loop to count outdegree
        }
    }
}
```

Refactored loop:

```
for (int i = 0; i < GRAPH_ORDER; i++) {
    int outdegree = 0;
    for (int k = 0; k < GRAPH_ORDER; k++) {
        if (adjacency_matrix[i][k] == 1) outdegree++;
    }
    for (int j = 0; j < GRAPH_ORDER; j++) {
        if (adjacency_matrix[i][j] == 1) new_pagerank[j] += pagerank[i] /
(double)outdegree;
    }
}
```

This modification reduced the number of iterations from a nested triplet to a nested double loop, leading to substantial performance improvements. The number of iterations on the *nice* graph increased from **21 to 1195 iterations**, marking a significant boost in performance.

Optimization 3: *Changing the Data Type of the Adjacency Matrix*
The adjacency matrix originally used the `double` data type, which was unnecessary since the graph was unweighted, consisting only of 1s and 0s. We experimented with several data types (`double`, `int`, `short int`, and `boolean`) to find the optimal type for performance. We discovered that using `short int` yielded the best performance without loss of accuracy.

The table below compares the number of iterations achieved with different data types for the adjacency matrix:

Data Type	Iterations Completed
Boolean	1155
Double	1195
int	1233
Short int	1239

Switching to `short int` provided the best results for this specific case.

Performance Metrics

After implementing the aforementioned optimizations, we measured the runtime performance of the code on two different graphs, the **nice** graph and the **sneaky** graph.

For the **nice** graph:
- **Original Code:** 21 iterations achieved in 9.58 seconds.
- **Optimized Code:** 1239 iterations achieved in 10.00 seconds.

For the **sneaky** graph:
- **Original Code**: 43 iterations achieved in 9.93 seconds.
- **Optimized Code:** 2223 iterations achieved in 10.00 seconds.

By optimizing the code's memory access patterns, reducing loop nesting, and switching data types, we were able to achieve substantial performance improvements for the PageRank algorithm. These optimizations reduced cache thrashing, improved memory locality, and maximized the number of iterations completed within a given timeframe. The results on both the **nice** and **sneaky** graphs demonstrate the success of our efforts, achieving up to 100x more iterations compared to the original code.

The next step involves parallelizing this optimized code using OpenMP and MPI, which we expect will further improve performance by distributing the computation across multiple cores.

OpenMP optimization

After optimizing the serial version of the PageRank algorithm, we applied OpenMP to leverage multi-threading and improve performance on multi-core systems. By carefully parallelizing key parts of the algorithm and minimizing overhead, we achieved significant performance gains.

Initial Optimization: Simple Loop Parallelization

The first step in optimizing with OpenMP was to parallelize each loop in the PageRank algorithm using `#pragma omp for`. Initially, we used separate parallel regions for each loop and applied reductions on key variables such as `diff` and `pagerank_total`.

1. Precomputing Outdegrees:

The outdegrees for each node are precomputed and stored in an array, which can be used later in the PageRank update step. Parallelizing this section yielded about 100 additional iterations on average.

```
```
#pragma omp parallel for
for (int i = 0; i < GRAPH_ORDER; i++) {
    for (int k = 0; k < GRAPH_ORDER; k++) {
        if (adjacency_matrix[i][k] == 1) outdegrees[i]++;
    }
}
```
```

2. Main PageRank Computation:

The primary PageRank update loop was also parallelized, distributing the computation of each node's new PageRank value across multiple threads.

```
```
#pragma omp parallel for
for (int i = 0; i < GRAPH_ORDER; i++) {
    for (int j = 0; j < GRAPH_ORDER; j++) {
        if (adjacency_matrix[j][i] == 1) {
            new_pagerank[i] += pagerank[j] / (double)outdegrees[j];
        }
    }
}
```
```

3. Setting New PageRank:

The next step involved applying the damping factor to the newly computed PageRank values, which was also parallelized to speed up this process.

```
#pragma omp parallel for
for(int i = 0; i < GRAPH_ORDER; i++) {
```

```
        new_pagerank[i] = DAMPING_FACTOR * new_pagerank[i] + damping_value;
    }
```

4. Reduction for `diff`:
To track the convergence of the PageRank algorithm, we needed to compute the difference between the current and new PageRank values. This step was parallelized using OpenMP's `reduction` clause to sum the differences across all threads.

```
    diff = 0.0;
    #pragma omp parallel for reduction(+:diff)
    for(int i = 0; i < GRAPH_ORDER; i++) {
        diff += fabs(new_pagerank[i] - pagerank[i]);
    }
```

5. Reduction for `pagerank_total`:
Similarly, we used the `reduction` clause to compute the total sum of the PageRank values, ensuring that the sum remains normalized across all threads.

```
    ```c
    double pagerank_total = 0.0;
    #pragma omp parallel for reduction(+:pagerank_total)
    for(int i = 0; i < GRAPH_ORDER; i++) {
        pagerank_total += pagerank[i];
    }
    ```
```

Further Optimizations:

Reducing Parallel Overhead and Synchronization:
Once the initial OpenMP parallelization was complete, we identified additional opportunities to reduce overhead and improve performance. Specifically, we focused on reducing the overhead of spawning new threads by combining the loops into a single parallel region and minimizing the use of barriers.

This alone provided a massive improvement over the baseline serial version.

Baseline Performance (Sneaky Graph): 2198 iterations in 10.00 seconds
Improved Performance: 10299 iterations in 10.00 seconds

Combining Loops into a Single Parallel Region:

Instead of creating separate parallel regions for each loop, we combined them into a single parallel region. This reduced the overhead associated with spawning and destroying thread teams and allowed the code to run more efficiently.

```
```
#pragma omp parallel default(none) shared(adjacency_matrix,
new_pagerank, pagerank, outdegrees, diff, pagerank_total)
firstprivate(damping_value, diff_local, pagerank_total_local)
{
    // PageRank computation
    #pragma omp for schedule(static)
    for (int i = 0; i < GRAPH_ORDER; i++) {
        for (int j = 0; j < GRAPH_ORDER; j++) {
            if (adjacency_matrix[j][i] == 1) {
                new_pagerank[i] += pagerank[j] /
(double)outdegrees[j];
            }
        }
    }

    // Updating the new PageRank values and accumulating diff and
pagerank_total locally
    #pragma omp for nowait schedule(static)
    for(int i = 0; i < GRAPH_ORDER; i++) {
        new_pagerank[i] = DAMPING_FACTOR * new_pagerank[i] +
damping_value;
        diff_local += fabs(new_pagerank[i] - pagerank[i]);
        pagerank_total_local += pagerank[i];
    }

    // Atomically updating the global diff and pagerank_total
    #pragma omp atomic
    diff += diff_local;

    #pragma omp atomic
    pagerank_total += pagerank_total_local;
}
```
```

This change improved performance by reducing the synchronization overhead, as fewer barriers were required between different phases of the algorithm.

Additional Optimizations:
We also applied a few more advanced techniques, including:

- Load Balancing: Using static scheduling to ensure balanced workloads across threads.
- Barrier Minimization: Removing unnecessary barriers and using the `nowait` clause to avoid synchronization where it wasn't needed.
- Loop Fusion: Combining loops where possible to reduce cache misses.
- Scaling Analysis: Identifying the optimal number of threads (16 in our case) to maximize performance.

These optimizations further boosted the performance, particularly on the **sneaky** graph, from **~10K to ~15K** iterations in 10 seconds.

Example of Final Optimization Using Reduction:

The final version of the code used OpenMP's `reduction` clause instead of atomic operations to update `diff` and `pagerank_total`. This provided a further performance boost, as reductions are generally faster than atomic operations.

```
#pragma omp parallel default(none) shared(adjacency_matrix, new_pagerank,
pagerank, outdegrees, diff, pagerank_total) firstprivate(damping_value)
{
    // PageRank computation
    #pragma omp for schedule(static)
    for (int i = 0; i < GRAPH_ORDER; i++) {
        for (int j = 0; j < GRAPH_ORDER; j++) {
            if (adjacency_matrix[j][i] == 1) {
                new_pagerank[i] += pagerank[j] / (double)outdegrees[j];
            }
        }
    }

    // Updating new PageRank values and performing reductions
    #pragma omp for nowait schedule(static)
    for(int i = 0; i < GRAPH_ORDER; i++) {
        new_pagerank[i] = DAMPING_FACTOR * new_pagerank[i] + damping_value;
    }

    // Reduction for diff and pagerank_total
    #pragma omp for nowait schedule(static) reduction(+:diff,
pagerank_total)
    for(int i = 0; i < GRAPH_ORDER; i++) {
        diff += fabs(new_pagerank[i] - pagerank[i]);
```

```
        pagerank_total += new_pagerank[i];
    }
}
```

Performance Metrics

- **Sneaky Graph:**
    - Baseline: 2198 iterations in 10.00 seconds
    - Optimized: 16843 iterations in 10.00 seconds

- **Nice Graph:**
    - Baseline: 1266 iterations in 10.00 seconds
    - Optimized: 14521 iterations in 10.00 seconds

These optimizations using OpenMP demonstrate the significant improvements that can be made by leveraging multi-threading, reducing overhead, and fine-tuning parallel regions and synchronization points.

MPI Optimization:

Simultaneously along the application of OpenMP, we also attempted distributed computing using MPI, specifically OpenMPI. In order to move from a serial implementation to distributed computing, there are a few problems we have to solve:

1. How to distribute the computation across the multiple processes.
2. How to get all processes the information they need for calculations and reporting.
3. How to synchronize the processes so they run for the same amount of iterations.

After solving these problems, we run multiple tests on different number of processes on different number of compute nodes to determine the resource combination that would give us the highest performance.

1. Distribution of PageRank computation

Given that there are 1000 websites in the graph and an arbitrary number of MPI processes, we use static chunking to distribute consecutive graph nodes to the processes. Each process will be responsible for calculating the pagerank of its delegated nodes, but they must communicate with other processes to gather the data to do so.

```
// MPI variables for gathering
// Assume mpi_size > 1
int recvcounts[mpi_size];
int displs[mpi_size];
```

```
int block_size = GRAPH_ORDER / (mpi_size - 1);

displs[0] = 0;
recvcounts[0] = block_size;
for (int i = 1; i < mpi_size; i++) {
    displs[i] = displs[i-1] + block_size;
    recvcounts[i] = i == mpi_size-1 ? GRAPH_ORDER % (mpi_size - 1) :
block_size;
}

// Local block size and block starting position
int block_pos = displs[mpi_rank];
if (mpi_rank == mpi_size - 1)
    block_size = recvcounts[mpi_size-1];
```

- `block_pos`: the starting position of the contiguous block of nodes that the current MPI process is responsible for.
- `block_size`: the number of nodes the MPI process is responsible for.

2. Obtain neighbouring graph nodes data from other MPI processes

The PageRank calculation of each website for each iteration requires the pagerank of all of that website's *incoming* neighbors from the previous iteration. This means that our MPI processes must communicate to each other to exchange this information.

We decided to simply have each MPI process send and gather the previously calculated pageranks from all other MPI processes, and fortunately for us, the MPI method `MPI_Allgatherv()` is exactly what we needed. This way of communication could introduce a large performance overhead, but it is simple.

Another way to achieve a better communication pattern that we considered is to partition the graph into regions such that all regions have the least number of neighboring regions. However, this problem on its own is hard, and the computation cost to get a partitioning may not be worth it for just 1000 websites.

`MPI_Allgatherv()` gathers into an array, and it must know where data from each MPI process must to write to within that array, so we created two arrays: `displs[]` and `recvcounts[]` that stores the displacement (location) of each graph node block and the number of nodes within the block.

```
MPI_Allgatherv(new_pagerank, block_size, MPI_DOUBLE, pagerank, recvcounts,
displs, MPI_DOUBLE, MPI_COMM_WORLD);
```

At the end of each iteration, every MPI process will use this method to send their newly calculated pageranks for their designated graph nodes, which are stored at `new_pagerank`, and gather all the pageranks across the network into `pagerank`.

3. Synchronize the iterations for all MPI processes

Ensuring the MPI processes run for the exact same number of iterations is important. Without synchronization, it is possible that a process may run for one more iteration while other processes have stopped, which leads to that process waiting for a communication that is not happening.

One way to synchronize all the MPI processes is to use an MPI barrier, which will block processes reaching the barrier and release them when all processes reach the barrier. However, MPI barrier is often the costliest way to implement synchronization and should only be used when timing is critical, so we did not consider this for our implementation. Instead, we decided to elect a MPI process as the time keeper instead.

For the MPI implementation, we elected the last MPI process, where `mpi_rank=mpi-size-1` as the time keeper. This process will be the only process that update the timer and broadcasts this timer to other processes in the network. This way, a process only needs to wait for the message from the time keeper and can then move on without waiting for other processes. Furthermore, because the exit condition of the main loop solely uses the timer value, all MPI processes will see the same timer value and exit after the same amount of iterations.

Additional changes: MPI reduction of pagerank total and difference

At the end of each iteration, the program sums up the total pagerank difference from the previous iteration as well as the total of the newly calculated pageranks. The total of pageranks is just used for validation because it should always be 1.0. The pagerank difference is used for the report at the very end of the program, where the min and max difference per iteration, as well as the total difference across all iterations will be logged.

For our MPI implementation, because we already distributed the pagerank calculation, we decided to also distribute the calculation of pagerank difference and total. Each iteration, each MPI process will calculate the pagerank difference and total within their own designated block of graph nodes, then we use a single `MPI_Reduce()` to reduce all the differences and totals to a root process. We also elected the last MPI process, `mpi_rank=mpi_size-1` as the root for the reduction, which means that this process is also responsible for logging out the results at the end.

We did consider another way of calculating the pagerank differences and total, which is to not use `MPI_Reduce()` and instead perform a linear scan similar to the original serial code after all the newly calculated pageranks are gathered using `MPI_Allgatherv()`. This may seem slower, but because we only have 1000 graph nodes to calculate, it may be faster than incurring

the cost of communication. However, this turns out to be slower than using MPI reduction and often resulting in a loss of **~1k** iterations.

Additional considerations: Graph generation and pre-calculation of outdegrees

Since our graph does not change overtime, and both the generation of the graph and the pre-calculation of the outdegrees are done once before the timer starts, we decided to not distribute this code. Thus, all MPI processes will do their own graph generation and outdegrees calculation before the actual computation.

Resource configuration

With distributed computing using MPI, we can expand our calculation to multiple processes and even compute nodes, enabling us to fully utilize the maximum resources we request from Teach. However, horizontally scaling does mean we would have to incur the additional cost of communication. We found out that inter-node communication was very costly for our program, and using more compute nodes does not mean more performance.

- 1 node, 16 tasks each: **15.5K iterations**
- 2 nodes, 16 tasks each: **13K iterations**
- 3 nodes, 16 tasks each: **12.7K iterations**
- 4 nodes, 16 tasks each: **10.5K iterations**

We noticed that the performance is not scaled with an increase in the number of processes when the processes are distributed across multiple compute nodes, so we decided to stay with 16 processes on 1 compute node and move on to the next implementations, where we will fine tune the resource configuration more.

An explanation for this behavior could be that the number of graph nodes, 1000, is simply too quick to compute that the communication overhead starts to dominate in the total cost in performance.

Final Performance measurement

- **Sneaky Graph:**
  - Baseline: 2198 iterations in 10.00 seconds
  - Optimized: 16097 iterations in 10.00 seconds

- **Nice Graph:**
  - Baseline: 1266 iterations in 10.00 seconds
  - Optimized: 15564 iterations in 10.00 seconds

MPI + OpenMP hybrid optimization

Once we had the implementation for OpenMP and MPI finalized, we combined the two together to create a hybrid version of both distributed memory and shared memory. The final code is MPI

code with the for loops from the OpenMP code and some adjustments we made that resulted in some performance increase because of the hybridization.

Adjustment 1: Remove MPI reduction
In the MPI section, we mentioned how we used MPI reduction instead of linear scan for calculating the pagerank difference and total each iteration. However, with OpenMP available to us, we considered using one MPI process with multithreading to perform the summation of pagerank difference and total to be more performant than incurring the cost of MPI communication, especially since we only have 1000 graph nodes. We were right about this judgment and were seeing an average increase of **1K to 2K iterations** depending on the resource requested.

Adjustment 2: Choosing a different MPI process to be the time keeper
For the MPI only implementation, the last MPI process, `rank = size - 1` was chosen to be the timekeeper of the network. However, we were also electing this process to perform other heavy lifting tasks such as calculating the pagerank difference and total each iteration, as well as logging. In the hybrid version, with the removal of MPI reduction, we consider the amount of work this process has to do before it can broadcast the timer value will cause more MPI processes to wait. We decided to try electing MPI process 0 as the new time keeper because it has a more similar amount of work compared to other MPI processes. With this adjustment, we gained an average of **2K iterations** depending on the resource requested.

Fine-tuning the Slurm configuration by trying different combination of resources
With the MPI and OpenMP hybrid code implemented, we moved on to selecting the combination of node, MPI processes, and OpenMP threads per process that yields the best performance. As previously mentioned, having more MPI processes means each process will have to compute less, but the cost of communication will be higher, especially when processes on different compute nodes communicate. Also, having more MPI processes means we will have less OpenMP threads per process, because we only request for at most 4 compute nodes from the Teach cluster.

Compute Nodes	Procs per node	Threads per proc	Average Iterations
4	16	1	11.5k
4	8	2	13.5k
4	4	4	14k
4	2	8	12k
4	1	16	14k
-------	-------	---------	----------
3	16	1	15.5k

3	8	2	10.8k
3	4	4	9.5k
3	2	8	20k
3	1	16	13.5k
-------	-------	---------	----------
2	16	1	17k
2	8	2	11k
2	4	4	12k
2	2	8	14.5k
2	1	16	15.5k
-------	-------	---------	----------
1	16	1	15.5k
1	8	2	14.2k
1	4	4	12.4k
1	2	8	8k
1	1	16	X

After the testing, we arrived at the conclusion that running the hybrid program on 3 nodes, each with 2 MPI processes and each process using 8 OpenMP threads, yields us the best performance.

Final Performance measurement

- **Sneaky Graph:**
  - Baseline: 2198 iterations in 10.00 seconds
  - Optimized: 23931 iterations in 10.00 seconds

- **Nice Graph:**
  - Baseline: 1266 iterations in 10.00 seconds
  - Optimized: 20685 iterations in 10.00 seconds

IHPCSS 2019

Unlike the 2024 challenge, the 2019 challenge starts by giving us starter code of different basic implementations of different HPC models, such as MPI, OpenMP, and OpenACC. Because of this, our goal for this challenge was to improve the provided implementation of MPI and OpenMP hybrid, which was called Hybrid CPU, instead of improving the provided serial code. To achieve such a goal, we started with improving the implementation of MPI and OpenMP, then brought the code changes over to the Hybrid CPU implementation.

Teach Compatibility

The 2019 project required several modifications to compile and run successfully on the Teach cluster. The project included two versions, small and big, which differ in the size of the grid used for the simulation.

In the provided Makefile, there were numerous instructions related to the compilation of Fortran source code, which were removed. The compiler was switched from the PGI compiler to GNU GCC, along with necessary changes to the corresponding flags. Additionally, the OpenACC implementations and GPU compilation instructions were removed, as well as any Fortran-related compilation steps. Constants that define row and column values were left unchanged, but any references to OpenACC and Fortran were also adjusted in both the run.sh and submit.sh scripts.

The project came with scripts for submitting and running jobs, with the correct flags and options for ease of use. The submit.sh script dispatches Slurm jobs, and run.sh executes the correct binary file with pre-specified commands.

Slurm scripts included in the project required modifications to be compatible with the Teach cluster. Flags such as `-A ac560tp` and `partition=RM` were removed. Parameters in run.sh, such as `mpirun -n 112` and `OMP_NUM_THREADS=28`, were modified to 64 and 16, respectively, to fit the 16 CPU cores per node on the Teach cluster.

These modifications allowed the project to compile and run efficiently on the Teach cluster, adapting it to the system's configuration and ensuring compatibility with its available resources.

OpenMP Improvement

The provided implementation of shared memory using OpenMP was somewhat basic, and we were able to identify potential improvements. However, it is worth to point out that the algorithm for simulating thermal diffusion is very suitable for multithreading as there is no irregular workload or any critical regions, so the original implementation was already quite performant. Because of this, we did not increase the performance of the OpenMP implementation by much. Moreover, it is interesting to point out many potential changes, which we thought would improve the performance, but lowered the performance instead.

Potential improvement: Using one OpenMP parallel region

Similar to the 2024 challenge, we also considered a single parallel construct for the two for loops that perform the main calculation and the temperature difference calculation. Our reasoning behind this consideration is that a single parallel region will have less overhead for spawning and destroying threads compared to using two `parallel for` constructs as done in the starter code.

```
#pragmam omp parallel
{
      #pragma omp for
      for(unsigned int i = 1; i <= ROWS; i++)
      for (unsigned int j = 1; j <= COLUMNS; j++)
            // Update temperature[i][j]

      #pragma omp for reduction(max:dt)
      for(unsigned int i = 1; i <= ROWS; i++)
      for (unsigned int j = 1; j <= COLUMNS; j++)
            // update to a higher dt
            // copy new temperature[i][j] to temperature_last[i][j]
}
```

Single parallel region construct

However, when testing this implementation, we found out that using a single parallel construct yielded slower time than using two separate `parallel for` constructs. For the small test (672x672), the time increase is **~0.5 seconds** and for the big test (14560x14560), the time increase is around **10 - 15 seconds**.

Potential improvement: Improving OpenMP pragmas

The `parallel for`'s in the starter code does not contain any additional details, so we considered adding to the `pragma`'s the `shared()` clause to indicate shared variables among the OpenMP threads and the `schedule(static)` clause to specify static scheduling. The `shared()` clause tells the compiler to handle variables, such as `temperature` and `temperature_last`, as shared memory, so that they are not determined on runtime. The `schedule(static)` specifies that the distribution of the loop iterations should be done following the static rule, which can be faster than the implicit default scheduling scheme that GNU OpenMP uses.

```
#pragma omp parallel for default(none) shared(temperature,
temperature_last) schedule(static)
// Main update loop
```

```
#pragma omp parallel for default(none) shared(temperature,
temperature_last) schedule(static) reduction(max:dt)
// Loop to update max temperature change and update temperature_last
```

These two changes did yield us a performance gain of around **~4 seconds** when running the big test.

Reordering the formula - a confusing improvement that works
We considered re-ordering the formula of the calculation within the main update loop like so:

Original:

```
temperature[i][j] = 0.25 * (temperature_last[i+1][j ] +
                            temperature_last[i-1][j ] +
                            temperature_last[i ][j+1] +
                            temperature_last[i ][j-1]);
```

New:

```
temperature[i][j] = 0.25 * (temperature_last[i-1][j ] +
                            temperature_last[i ][j+1] +
                            temperature_last[i ][j-1] +
                            temperature_last[i+1][j ]);
```

In C, 2-D arrays are stored in a row major manner, so elements of the previous rows are stored before the elements of the next rows. We considered this change because we thought that accessing the elements of the array `temperature_last` in the same order they are stored would be more efficient that the original order of access. Surprisingly, this change did yield us a performance gain of **~4 seconds**.

Final Performance Measurements

1. **Baseline OpenMP implementation:**
    a. Small Grid - 2.1 seconds
    b. Big Grid - 509.9 seconds
2. **Improved OpenMP implementation:**
    a. Small Grid - 2.0486 seconds
    b. Big Grid - 501.3169 seconds

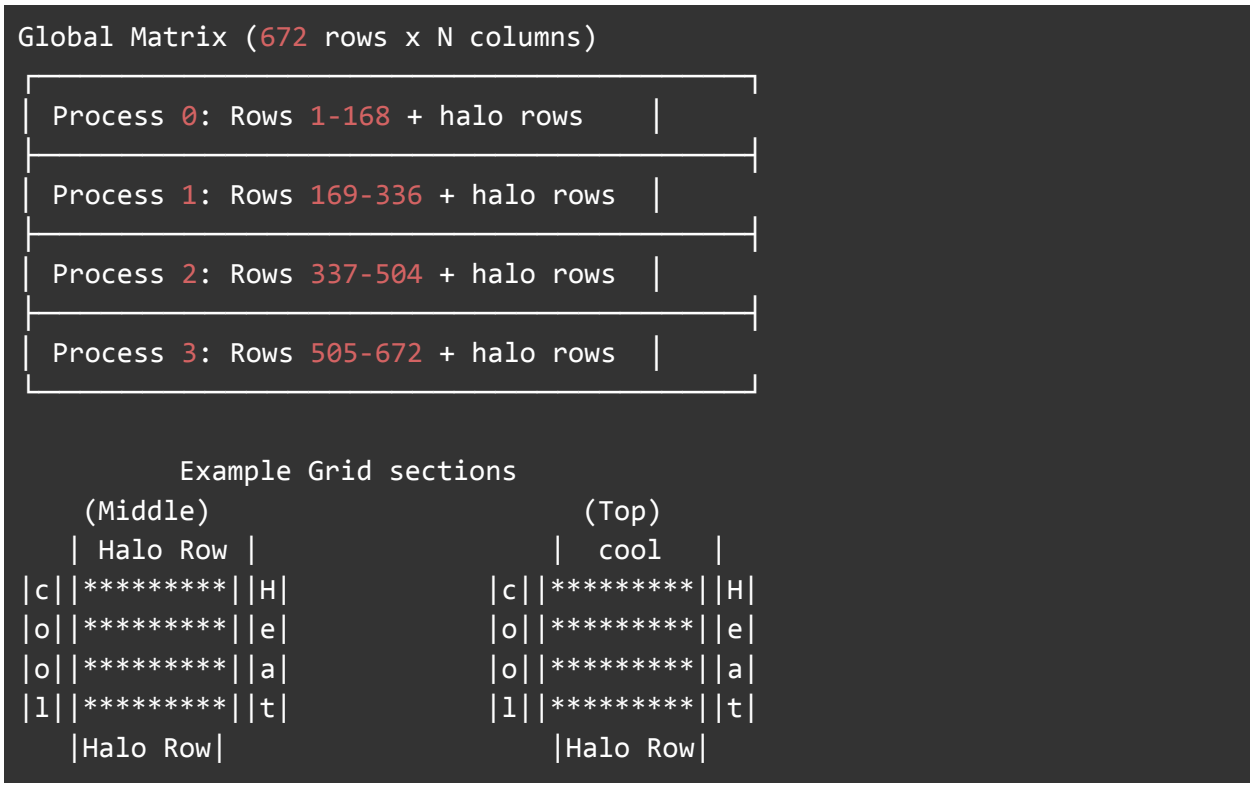
MPI Optimization:

Unlike the 2024 challenge, the 2019 MPI implementation was already in place and functional, leaving little room for new design or major implementation changes. However, the existing code required performance tuning to optimize the communication between processes.

Domain Decomposition and 2-D Halo Exchange:

The provided MPI implementation used a 2-D Halo Exchange communication pattern, where the matrix (or grid) was decomposed row-wise. Each MPI process handled a block of consecutive rows, with additional halo rows at the top and bottom to exchange boundary information with neighboring processes.

For instance, in the small grid version (672 rows), with 4 MPI processes, each process would be responsible for **168 rows**, but would also maintain extra halo rows for communication:



Each process computes its assigned rows while periodically exchanging boundary (halo) rows with its neighboring processes. The **top and bottom rows** are sent to neighboring processes, ensuring that each process has the necessary boundary data to perform its calculations.

Communication Bottleneck and Optimization:

The original MPI implementation used separate `MPI_Send` and `MPI_Recv` calls to exchange the halo rows. This approach caused inefficiencies because each process would wait for its neighbors to complete their send/receive operations before continuing. This cascading

wait caused delays and slowed down the overall performance, especially as the number of processes increased.

Original Communication Pattern:

The original implementation handled communication in stages, with separate calls for sending and receiving data:

```
// If we are not the last MPI process, we have a bottom neighbor
if(my_rank != comm_size-1) {
    MPI_Send(&temperature[ROWS][1], COLUMNS, MPI_DOUBLE, my_rank+1, 0,
MPI_COMM_WORLD);
}

// If we are not the first MPI process, we have a top neighbor
if(my_rank != 0) {
    MPI_Recv(&temperature_last[0][1], COLUMNS, MPI_DOUBLE, my_rank-1,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
}

// If we are not the first MPI process, we have a top neighbor
if(my_rank != 0) {
    MPI_Send(&temperature[1][1], COLUMNS, MPI_DOUBLE, my_rank-1, 0,
MPI_COMM_WORLD);
}

...<similar pattern continues>
```

In this design, each process had to wait for both send and receive operations, causing unnecessary delays.

Refactoring with MPI_Sendrecv:

To resolve this, we replaced the separate `MPI_Send` and `MPI_Recv` calls with the more efficient `MPI_Sendrecv` operation. This function allows for simultaneous sending and receiving of data, reducing the communication overhead and eliminating the potential for cascading delays.

```
// Send down, receive up phase
```

```
to = my_rank == comm_size-1 ? MPI_PROC_NULL : my_rank + 1;
from = my_rank == 0 ? MPI_PROC_NULL : my_rank - 1;
MPI_Sendrecv(&temperature[ROWS][1], COLUMNS, MPI_DOUBLE, to, 0,
             &temperature_last[0][1], COLUMNS, MPI_DOUBLE, from,
MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);

// Send up, receive down phase
to = my_rank == 0 ? MPI_PROC_NULL : my_rank - 1;
from = my_rank == comm_size-1 ? MPI_PROC_NULL : my_rank + 1;
MPI_Sendrecv(&temperature[1][1], COLUMNS, MPI_DOUBLE, to, 0,
             &temperature_last[ROWS+1][1], COLUMNS, MPI_DOUBLE, from,
MPI_ANY_TAG,
             MPI_COMM_WORLD, &status);
```

By combining the send and receive operations into a single `MPI_Sendrecv` call, we significantly improved the efficiency of the halo exchange. Each process can now send and receive its boundary rows in a single operation, minimizing waiting times and reducing the synchronization points between processes. Due to the large number of iterations, the small gain that we get by minimizing potential Send-Locks that happen when a large number of processes are sending at the same time amplifies in the **big grid** evaluation, as it will be seen in the next section.

Results of MPI Optimization

The performance gained from the optimization of the Halo communication pattern using a simple swap to the simultaneous `MPI_Sendrecv` is documented as follows.

3. **Baseline MPI implementation:**
    a. Small Grid - 2.2929 seconds
    b. Big Grid - 117.3 seconds
4. **Improved MPI implementation:**
    a. Small Grid - 2.2913 seconds
    b. Big Grid - 99.074 seconds

Important to mention that the run.sh script that the challenge was advised to use only allows **8 MPI processes** for each small grid run and **64 MPI processes** for the bigger grid. For MPI only performance, this should not be a limitation as the larger number of processes should yield a faster result in this communication pattern, but in the Hybrid - OMP X MPI implementation, there are pre-defined #Threads and #Processes that the run.sh script runs the experiment and to further lock you in, the pre-defined **Partial Rows and Columns** used for the MPI domain decomposition, are hardcoded to be only for **8 MPI** processes in total. As we will see, we could

only do a full scaling analysis with combinations of NODE X MPI PROCESSES that were a total of **8 MPI processes.**

Hybrid OMP X MPI Optimization

Our approach to improving the original `hybrid_cpu` implementation was based on combining the optimizations we made to both the OpenMP and MPI components individually. The enhancements we applied included:

- **Modified Halo Swapping with `MPI_Sendrecv`**: We replaced the existing halo swapping mechanism with `MPI_Sendrecv`, which helped reduce communication overhead by facilitating simultaneous data exchange between processes. This change was particularly effective in minimizing the communication bottlenecks associated with boundary data transfer between MPI processes.

- **Improved OpenMP Pragmas**: We refined the OpenMP parallel regions by explicitly specifying shared variables. This reduced unnecessary data sharing and improved memory access patterns, leading to more efficient utilization of shared memory and faster computation.

However, there were two key OpenMP optimizations that we deliberately did not apply, as they introduced new issues:

1. **Static Scheduling:** Adding `schedule(static)` to OpenMP loops would cause a significant delay of ~2.4 seconds during the execution of the big test case. Static scheduling distributes iterations evenly among threads for OpenMP only version but failed to improve performance in this case, which could be due to uneven workload distribution across the threads, especially with the larger grid size.

2. **Reordered Variable Access:** Reordering the variable access within the calculation phase caused the results to become inaccurate, only matching up to 13 decimal places instead of the expected 18. This introduced numerical instability, likely due to precision-related issues. Since this reordering no longer had a noticeable performance impact in the OpenMP version, we decided not to include it in the hybrid implementation.

In addition, we explored the idea of using `MPI_Allreduce` instead of separate `MPI_Reduce` and `MPI_Bcast` calls for reducing the maximum temperature change. However, this resulted in a delay of ~5 seconds in the big test case, likely due to specific implementation details of OpenMPI. Thus, we reverted to the original combination of `MPI_Reduce` and `MPI_Bcast`.

Performance Comparison

1. **Baseline MPI implementation:**
    c. Small Grid - 4.0878 seconds
    d. Big Grid - 151.3637 seconds
2. **Improved MPI implementation:**
    e. Small Grid - 3.8125 seconds
    f. Big Grid - 143.2960 seconds

Full Scaling Analysis
A full scaling analysis was conducted, similar to the 2024 challenge, but due to the specific structure of the 2019 challenge, we were limited to using only 8 MPI processes. This limitation stems from the way the data matrix is initialized—using predefined `NROWS` and `NCOLUMN` variables that were specifically designed for 8 MPI processes.

The scaling analysis for the hybrid CPU implementation on the big C version was carried out using all valid combinations of nodes, MPI processes, and threads, ensuring comparability with the original implementation.

Scaling Analysis for Hybrid (Big Grid):

Compute Nodes	Procs per node	Threads per proc	Time
1	8	1	990.8353s
1	8	2	495.7244
—---	—---	—---	—----
2	4	1	721.6636
2	4	2	501.3176
2	4	4	259.6706
—---	—---	—--	—--
4	2	1	629.2334
4	2	2	377.3591
4	2	4	263.3787
4	2	8	143.7170

CSCD94 Project Concluding Statement

Our final implementation of the 2024 challenge has a 10,000x performance gain compared to the original serial code. To achieve this gain, we combined MPI collective communication with OpenMP multithreading and created a hybrid implementation of the PageRank algorithm that uses both distributed memory and shared memory. Our decisions behind the details of the implementation are guided by solving the challenges we faced, such as the high cost of inter-node communication and the small problem size of 1000 websites. We also arrived at the best combination of resources within the restriction of the Teach cluster that works best with our hybrid implementation through rigorous testing.

Similar to the 2024 Hybrid Scaling Analysis, achieving optimal performance in the 2019 MPI implementation required striking a balance between intra-node and inter-node communication. The best performance was found when we minimized the intra-node communication overhead of OpenMP and balanced the CPU's processing load across threads and MPI processes. By fine-tuning the number of threads and processes, and improving the efficiency of the communication through the use of MPI_Sendrecv, we were able to maximize the computational throughput and minimize communication bottlenecks, resulting in a significantly more efficient solution.