Table of Contents

[Introduction: 2](#_Toc159068959)

[Problem statement: 2](#_Toc159068960)

[Disclaimer: 3](#_Toc159068961)

[Initial conditions and how to run the programs: 3](#_Toc159068962)

[Different Implementations and analysis of the program: 4](#_Toc159068963)

[1- A basic row-major approach. (TempGrid\_HW3.c) 4](#_Toc159068964)

[2- Enhancing the code by using swappable arrays for old and new, and fewer barriers [only when necessary] (HW3\_1.c) 4](#_Toc159068965)

[3- Exploring Further enhancement by utilizing the code symmetry to minimize calculations. 5](#_Toc159068966)

[Automation and output3.txt creation: 9](#_Toc159068967)

[The results after running the first approach. (1) on a Mac with 11 cores. 10](#_Toc159068968)

[The results after running the first approach. (1) Linux with 16 cores. 11](#_Toc159068969)

[The results after running the pointer approach. (2) on a Mac with 11 cores. 12](#_Toc159068970)

[The results after running the pointer approach. (2) Linux with 16 cores. 13](#_Toc159068971)

[Summary and closing notes: 14](#_Toc159068972)

# Introduction:

In this homework, we will use of MPI (Message Passing Interface) to model the temperature distribution across a 1000x1000 grid, where we explore the temperature evolution under specified initial conditions over 4000 time steps. This program, through its iterations, explores the performance advantages and complexities of concurrency and synchronization, with an emphasis on optimizing computational efficiency.

# Problem statement:

This is a program using MPI to estimate the temperature of all points on a grid.

Parameters and specifications:

1. Grid size = 1000 x 1000, spanning all points in the square between coordinates (1,1) and (1000,1000).
2. Initial condition: All center points in the region (200, 200) to (800, 800) have a temperature of 500 degrees, and all other points have a temperature of zero. Points outside the grid (i.e., neighbors of points on the boundary) always have a temperature of zero that does not change.
3. At each time step t, the temperature of a point at coordinates (x,y) is computed from the temperatures of the neighboring points in the previous time step (t-1) according to the following equation:

T(x,y)(t) = T(x,y)(t-1)

+ Cx \* (T(x+1,y)(t-1) + T(x-1,y)(t-1) – 2 T(x,y)(t-1))

+ Cy \* (T(x,y+1) (t-1) + T(x,y-1) (t-1) – 2 T(x,y) (t-1))

Where Cx=0.125 and Cy=0.11

1. Run the program for 4000 time steps. Note that depending on how you split your data, you may need to communicate information to neighboring processors after each time step.
2. After each 200 time steps, you should print the temperatures of the following points: (1, 1), (150,150), (400, 400), (500, 500), (750, 750), and (900,900).

4000/200 = 20 lines. => Temperature of 6 points

Solution

There are so many ways to divide up and process this problem, one might be more efficient than other given the hardware structure on which the program is running. To really compare the different ways this MPI code could be structured, we need to experiment and run the program in different configurations.

## Disclaimer:

Those programs were run on a Mac M3 pro machine with 11 cores, as well as the mo/auto PSU server. As expected, we noticed that the performance saturates beyond 11 threads on the mac, while it improves a little bit on Linux, overall, the mac was much more efficient than the Linux server, even with a fewer core.

## Initial conditions and how to run the programs:

A diagram of temperature

Description automatically generated

To compile the code, we use:

mpicc -o hw4 hw4.c

On macos, the -lrt flag is not needed.

On Mac, this is the command need to run the code.

mpirun -np 11 ./sum

On linux, this is the command need to run the code:

mpirun –oversubscribe -H localhost -np 11 ./hw4

To run 11 threads for example

# MPI Calls/Functions used in our project:

**MPI\_Bcast**

Broadcasts a message from the process with rank "root" to all other processes of the communicator.

**Synopsis**

int MPI\_Bcast( void \*buffer, int count, MPI\_Datatype datatype, int root,

MPI\_Comm comm )

**Input/Output Parameters**

**buffer**

starting address of buffer (choice)

**Input Parameters**

**count**

number of entries in buffer (integer)

**datatype**

data type of buffer (handle)

**root**

rank of broadcast root (integer)

**comm**

communicator (handle)

**MPI\_Gather**

Gathers together values from a group of processes

**Synopsis**

int MPI\_Gather(const void \*sendbuf, int sendcount, MPI\_Datatype sendtype,

void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root,

MPI\_Comm comm)

int MPI\_Gather\_c(const void \*sendbuf, MPI\_Count sendcount,

MPI\_Datatype sendtype, void \*recvbuf, MPI\_Count recvcount,

MPI\_Datatype recvtype, int root, MPI\_Comm comm)

**Input Parameters**

**sendbuf**

starting address of send buffer (choice)

**sendcount**

number of elements in send buffer (non-negative integer)

**sendtype**

data type of send buffer elements (handle)

**recvcount**

number of elements for any single receive (non-negative integer)

**recvtype**

data type of recv buffer elements (handle)

**root**

rank of receiving process (integer)

**comm**

communicator (handle)

**Output Parameters**

**recvbuf**

address of receive buffer (choice)

**MPI\_Finalize**

Terminates MPI execution environment

**Synopsis**

int MPI\_Finalize( void )

**Notes**

All processes must call this routine before exiting. The number of processes running *after* this routine is called is undefined; it is best not to perform much more than a return rc after calling MPI\_Finalize.

We initialized the MPI and used the MPI\_COMM\_WORLD to initialize the MPI\_Comm\_rank and MPI\_Comm\_size as shown below:

# Different Implementations and analysis of the program:

HW4\_1

This implementation is our most efficient one. After experimenting with different ways to distribute the work between MPI tasks, this one provided the best execution time.

In this implementation instead of using MPI\_Send and MPI\_Recv, we’re using MPI\_Bcast and MPI\_Gather to broadcast the work to different tasks, then collect “gather” it once the tasks are done.

Here we refer to threads, cores, or processors as tasks.

In this implementation, we’re allocating memory area for our 2 multi-dimensional arrays and pointing to them using points.

float (\*grid)[Y\_SIZE] = (float (\*)[Y\_SIZE])malloc(X\_SIZE \* Y\_SIZE \* sizeof(float));

float (\*new\_grid)[Y\_SIZE] = (float (\*)[Y\_SIZE])malloc(X\_SIZE \* Y\_SIZE \* sizeof(float));

Now we initially tried to implement a swapping function, basically if the rank of the current task is 0 (the MASTER), then the master will swap the pointers to between the old and current grids, similar to what we did in the pthreads program. We realized however that this is not really needed as we can encapsulate this in the MPI\_Gather operation.

We refer to the mpich.org reference below:

<https://www.mpich.org/static/docs/latest/www3/MPI_Gather.html>

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

We’re then broadcasting the message below to all the processors. The first argument is the address of the buffer, which is the grid in our case. The second argument is the count (which is the number of entries in the buffer/grid) hence we’re multiplying the dimensions of the grid to get the total number of points in the grid, and we’re assigning this data MPI\_Datatype = MPI\_FLOAT, we are setting the root, which is the rank of the broadcast as MASTER, and we’re using the MPI\_COMM\_WORLD as the communicator or handle.

MPI\_Bcast(grid, X\_SIZE \* Y\_SIZE, MPI\_FLOAT, MASTER, MPI\_COMM\_WORLD);

Did you notice that we passed the grid as the arugment to MPI\_Bcast? We broadcasted the current (or you can say the old) grid to all the tasks, and we gatherd the newly calculated one as the new\_grid. This is where our swapping of the old and new grids happen.

MPI\_Gather(new\_grid[start\_row], rows\_per\_process \* Y\_SIZE, MPI\_FLOAT,

grid, rows\_per\_process \* Y\_SIZE, MPI\_FLOAT, MASTER, MPI\_COMM\_WORLD);

After we gather the new grid for this time step from MPI\_Gather, we pass it along to MPI\_Bcast to broadcase it again, this time as the current grid to all the processors.

MPI\_Bcast(grid, X\_SIZE \* Y\_SIZE, MPI\_FLOAT, MASTER, MPI\_COMM\_WORLD);

The MPI\_Bcast, MPI\_Gather, and MPI\_Bcast again are repeated in a loop for each timestep in our analysis, and once we’ve covered all the time steps, we stop the timer, calculate and print the execution time, call MPI\_Finalize, and free the memory allocated for grid and new\_grid

MPI\_Finalize();

We use this to Terminates MPI execution environment.

## A basic row-major approach. (TempGrid\_HW3.c)

* We updated the code to ensure that only one thread (thread 0) is responsible for updating the **old** grid with **new** values after each timestep, which prevents race conditions.
* The initialization of the **old** grid to set initial conditions is streamlined with a single nested loop and a conditional statement. Compared to the given example which assigned 0 to the whole grid, then did an override and assigned 500 to the center grid, wasting computational resources.
* Added sanity checks to make sure we’re not performing any calculations outside the grid.

A screenshot of a computer screen

Description automatically generated

## Enhancing the code by using swappable arrays for old and new, and fewer barriers [only when necessary] (HW3\_1.c)

* **Dynamic Allocation**: Both **old** and **new** grids are dynamically allocated, allowing for pointer swapping.
* **Single Barrier**: Only one synchronization point (barrier) per iteration is used, reducing synchronization overhead.
* **Pointer Swapping**: After each iteration and synchronization point, pointers to **old** and **new** grids are swapped, eliminating the need for element-wise copying.

A screenshot of a computer

Description automatically generated

Comparing this version of the program to the initial one, we can see that we’re generating the same temperatures, but now the time is much, much less.

## Exploring Further enhancement by utilizing the code symmetry to minimize calculations.

To further enhance the code by utilizing the symmetry of the grid, we can calculate the temperature for just a quarter or half of the grid and mirror these values to the other parts. This approach can potentially halve or quarter the computation time.

Given the grid's symmetry, we will start by calculating temperatures for half of the grid (either vertically or horizontally) and then mirroring these values. For simplicity, let's assume we calculate for the left half and mirror to the right half. We need to adjust the logic slightly if the symmetry or the pattern of temperature distribution is different.

Here's what the temp function is when calculating only the left half of the grid and mirroring the values to the right half.

void\* Temp(void\* tmp) {

long int threadId = (long int)tmp;

int halfX = X\_SIZE / 2; // Calculate only for the left half

int start = threadId \* (halfX - 1) / NumThreads + (threadId < Remainder ? threadId : Remainder);

int end = start + (halfX - 1) / NumThreads - 1 + (threadId < Remainder ? 1 : 0);

for (int block = 1; block <= TIMESTEPS; block++) {

for (int j = start; j <= end; j++) {

for (int k = 1; k < Y\_SIZE - 1; k++) {

if (j > 0 && j < halfX) {

new[j][k] = old[j][k] + Cx \* (old[j + 1][k] + old[j - 1][k] - 2 \* old[j][k]) + Cy \* (old[j][k + 1] + old[j][k - 1] - 2 \* old[j][k]);

// Mirror the calculated value to the right half

new[X\_SIZE - 1 - j][k] = new[j][k];

}

}

}

Barrier();

if (threadId == 0) {

float (\*temp)[Y\_SIZE] = old;

old = new;

new = temp;

if (block % PRINT\_STEPS == 0) {

printf("\nThe Temperature values at points:[1,1]=%f [150,150]=%f [400,400]=%f [500,500]=%f [750,750]=%f [900,900]=%f", old[1][1], old[150][150], old[400][400], old[500][500], old[750][750], old[900][900]);

}

}

Barrier();

}

return NULL;

}

The new program is called HW3\_2, as compared to the program from step (2), which is called HW3\_1 here.

A screenshot of a computer screen

Description automatically generated

We see that we are able to cut some time, not quite 50% of the time because we have introduced some overhead logic to “mirror” the left half into the right half, but we also notice that the calculated values are not exactly the same, though in general they seem to be on the right direction. We explored what might be causing this discrepancy:

1. **Barrier Synchronization Overhead**: While the intention was to reduce computation by leveraging symmetry, the additional overhead of synchronizing threads (especially with multiple barriers) and the added complexity of mirroring the data might not offset the gains from reduced computation, particularly if the computation itself is not the primary bottleneck.
2. **Data Consistency Issues**: If the mirroring of data occurs while other threads are still computing their portions, or if the mirroring itself is not correctly synchronized, this could lead to inconsistencies in the grid data, affecting accuracy.

We further explored mirroring vertically instead of horizontally, to see if maybe this works better with our cache.

void\* Temp(void\* tmp) {

long int threadId = (long int)tmp;

int start = threadId \* (X\_SIZE - 1) / NumThreads + (threadId < Remainder ? threadId : Remainder);

int end = start + (X\_SIZE - 1) / NumThreads - 1 + (threadId < Remainder ? 1 : 0);

int halfY = Y\_SIZE / 2; // Only calculate for the top half

for (int block = 1; block <= TIMESTEPS; block++) {

for (int j = start; j <= end; j++) {

for (int k = 1; k < halfY; k++) { // Iterate only over the top half

if (k > 0 && k < halfY) {

new[j][k] = old[j][k] + Cx \* (old[j + 1][k] + old[j - 1][k] - 2 \* old[j][k]) + Cy \* (old[j][k + 1] + old[j][k - 1] - 2 \* old[j][k]);

}

}

}

Barrier();

// Mirror the calculated values to the bottom half

if (threadId == 0) {

for (int x = 0; x < X\_SIZE; x++) {

for (int y = 1; y < halfY; y++) {

new[x][Y\_SIZE - y - 1] = new[x][y];

}

}

}

Barrier();

if (threadId == 0) {

// Swap the pointers

float (\*temp)[Y\_SIZE] = old;

old = new;

new = temp;

if (block % PRINT\_STEPS == 0) {

printf("\nThe Temperature values at points:[1,1]=%f [150,150]=%f [400,400]=%f [500,500]=%f [750,750]=%f [900,900]=%f", old[1][1], old[150][150], old[400][400], old[500][500], old[750][750], old[900][900]);

}

}

}

return NULL;

}

This however degraded the performance.

A screen shot of a number

Description automatically generated

# Automation and output3.txt creation:

We created a simple Perl script to automate the creation of the output3.txt file and run the HW3 program from 1 to 16 threads and calculate their speedup. In large, this was a modification from the Perl automating script in HW2.

#!/usr/bin/perl

use strict;

use warnings;

# Output file

my $output\_file = 'Ghonim\_Nordstrom\_output3.txt';

# Open the file for writing

open(my $fh, '>', $output\_file) or die "Could not open file '$output\_file' $!";

# Print the header to the file with fixed widths for each column

printf $fh "%-20s %-15s %-15s\n", "Number of Threads", "Time (Seconds)", "Speedup";

# Variable to store time for single-thread execution

my $single\_thread\_time = 0;

# Run the program with different numbers of threads

for (my $num\_threads = 1; $num\_threads <= 16; $num\_threads++) {

# Execute the program and capture its output

my $output = `./TempGrid\_HW3 $num\_threads`;

# Extract the time from the output

my ($time\_in\_seconds) = $output =~ /Time = \d+ nanoseconds\s+\(([\d\.]+) sec\)/;

# Calculate speedup

my $speedup = $num\_threads == 1 ? 1 : $single\_thread\_time / $time\_in\_seconds;

$single\_thread\_time = $time\_in\_seconds if $num\_threads == 1;

# Write the results to the output file with fixed widths for each column

printf $fh "%-20d %-15f %-15f\n", $num\_threads, $time\_in\_seconds, $speedup;

}

# Close the file

close $fh;

print "Results saved in $output\_file\n";

## The results after running the first approach. (1) on a Mac with 11 cores.

A screenshot of a computer

Description automatically generated

## The results after running the first approach. (1) Linux with 16 cores.

A screenshot of a computer

Description automatically generated

## The results after running the pointer approach. (2) on a Mac with 11 cores.

A screenshot of a computer

Description automatically generated

## The results after running the pointer approach. (2) Linux with 16 cores.

A screenshot of a computer

Description automatically generated

It looks like either the linux system has only 8 cores, or the performance drops after 8 threads due to the communication overhead on linux, as compared to mac. We checked the linux system and found that it has 32 cores.

A screenshot of a computer

Description automatically generated

Therefore it’s probably due to more communication overhead that causes performance to degrade on the linux systems as compared to macOS.

# Summary and closing notes:

In conclusion, we have experimented with different implementations of threads starting an obvious implementation that uses multiple loops to copy the old grid into the new grid every timestep to an enhanced program utilizing pointers, then the exploitation of symmetry. This has revealed the intricate balance between computation, synchronization, and memory management in parallel processing. The optimization techniques employed, from swappable arrays to symmetry-based calculations have provided us valuable insights into the behavior of parallel systems. The experimental results highlighted the importance of thread management and data consistency, showcasing the potential for significant speedups while also cautioning against the pitfalls of overhead resulting from complexity.

Top of Form

Bottom of Form