



Maseeh College of Engineering  
and Computer Science

PORTLAND STATE UNIVERSITY

ECE 688: WINTER 2024 - HW4  
ADVANCED COMPUTER ARCHITECTURE II

# HEAT DIFFUSION GRID HOMEWORK 4 MPI-IMPLEMENTATION

MOHAMED GHONIM  
AHLIAH NORDSTROM

03/03/2024

## Table of Contents

<i>Introduction:</i> .....	2
<b>Problem statement:</b> .....	2
<i>Solution</i> .....	3
<b>Disclaimer:</b> .....	3
<b>Initial conditions and how to run the programs:</b> .....	3
<i>MPI Calls/Functions used in our project:</i> .....	4
<b>MPI_Bcast</b> .....	4
<b>MPI_Gather</b> .....	4
<b>MPI_Finalize</b> .....	5
<b>MPI_Wtime</b> .....	5
<b>Synopsis</b> .....	5
<b>Return value</b> .....	5
<i>Implementation Considerations</i> .....	6
<b>MPI_Bcast and MPI_Gather implementation. [The best performing one]</b> .....	7
<i>A walk through our code:</i> .....	8
<b>Libraries used:</b> .....	8
<b>Constants</b> .....	8
<b>Simulation Process</b> .....	8
<i>Automation and output4.txt creation:</i> .....	10
<b>The results after running the program on a Mac with 11 cores.</b> .....	12
<b>The results after running the program on a Linux with 16 cores.</b> .....	13
<b>Results from an early experimentation on the program using MPI_Send and MPI_Recv and the Mac system.</b> .....	13
<i>Summary of Findings:</i> .....	15
<b>Mac System with 11 Cores (Using MPI_Bcast and MPI_Gather):</b> .....	15
<b>Linux System with 16 Cores (Using MPI_Bcast and MPI_Gather):</b> .....	15
<b>Early Experimentation on Mac System (Using MPI_Send and MPI_Recv):</b> .....	15
<b>Comparative Analysis:</b> .....	15
<b>Conclusions:</b> .....	16

## Introduction:

In this homework, we will use of MPI (Message Passing Interface) to model and simulate the temperature diffusion 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:

- a) Grid size = 1000 x 1000, spanning all points in the square between coordinates (1,1) and (1000,1000).
- b) 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.

- c) 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:

$$\begin{aligned} T(x,y)(t) = & T(x,y)(t-1) \\ & + Cx * (T(x+1,y)(t-1) + T(x-1,y)(t-1) - 2 T(x,y)(t-1)) \\ & + Cy * (T(x,y+1)(t-1) + T(x,y-1)(t-1) - 2 T(x,y)(t-1)) \end{aligned}$$

Where  $Cx=0.125$  and  $Cy=0.11$

- d) 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.
- e) **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

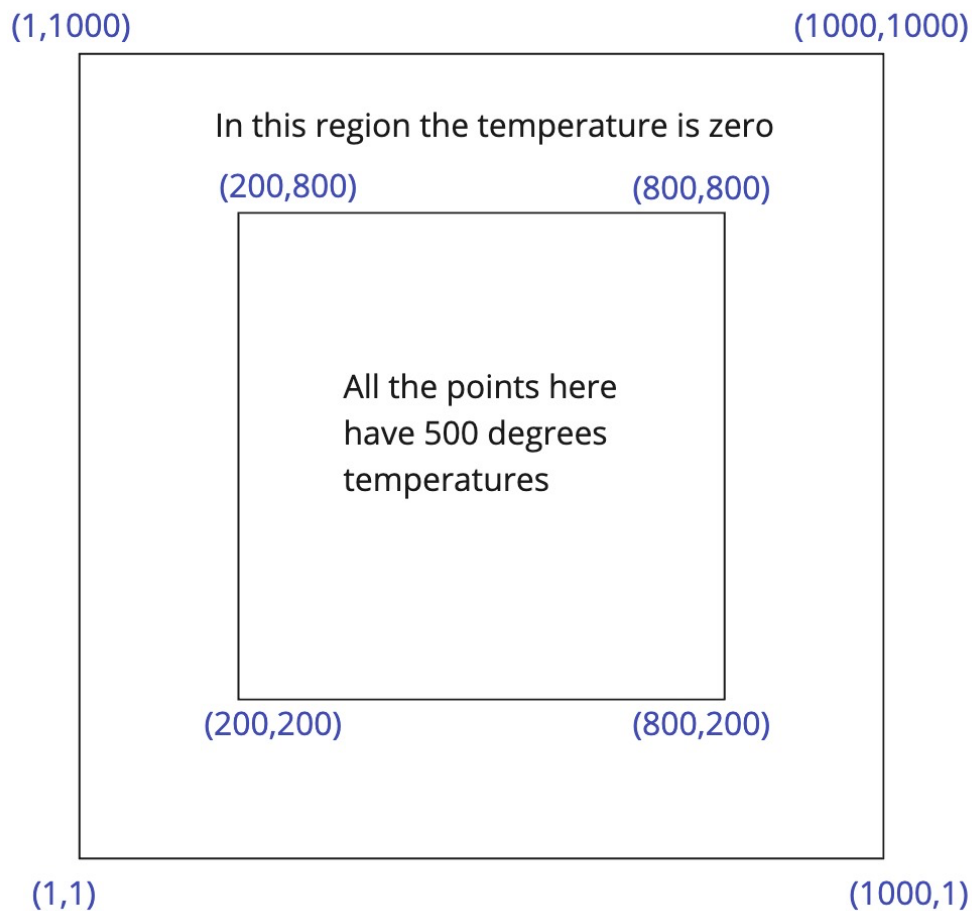
There are so many ways to divide up and process this problem, one might be more efficient than other given the hardware structure on which the program is running. To really compare the different ways this MPI code could be structured, we need to experiment and run the program in different configurations.

### Disclaimer:

This program was run on a Mac M3 pro machine with 11 cores, as well as the mo/auto PSU linux server which has up to 32 cores. 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:

### Initial Condition



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:

We referred to the mpich.org reference below for definitions and use cases of the different MPI components. [Man pages for MPI]

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

### 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:

### **MPI\_Wtime**

Returns an elapsed time on the calling processor.

#### **Synopsis**

double MPI\_Wtime(void)

#### **Return value**

Time in seconds of resolution of MPI\_Wtime

## Implementation Considerations

We explored various MPI (Message Passing Interface) approaches for efficiently parallelizing this program. The focus was on identifying the most efficient method for distributing work among processes and gathering the results.

Initially, a direct point-to-point communication strategy was considered, utilizing **MPI\_Send** and **MPI\_Recv**, which allows for explicit control over the data exchange between individual MPI tasks. These functions are well-suited for applications where tailored communication patterns are required, as they enable specific sender-receiver pairings.

However, the analysis led to the adoption of collective communication operations, specifically **MPI\_Bcast** and **MPI\_Gather**. The decision was based on performance metrics, where these collective operations outperformed the point-to-point counterparts in execution time. This communication strategy seems to work best for our program since we have a symmetric matrix that can easily be split as evenly as possible; hence we're broadcasting and gather very similar things to the different tasks/processors and there's no need to use many point-to-point communications in this case.

Collective operations like **MPI\_Bcast** and **MPI\_Gather** are designed for scenarios where a single data item needs to be shared across all processes (broadcasting), or where data from all processes need to be collected into a single process (gathering). These functions abstract away the complexity of setting up individual send/receive calls, offering a more streamlined approach that can lead to performance gains due to their implementation optimizations.

In the context of our program, the term "tasks" is used interchangeably with threads, cores, or processors. In MPI terminology, a task typically refers to a single MPI process, which may be running on a thread or core. The term reflects the MPI perspective, where each process performs a part of the computation, and collectively, they solve a problem in parallel.

From our initial experimentation and code "drafts" we concluded that this specific application, using **MPI\_Bcast** to distribute the workload and **MPI\_Gather** to collect the results was the most efficient strategy. This approach benefits from reduced communication overhead and optimized data transfer patterns that these collective operations can provide. It's important to note that the efficiency of collective versus point-to-point communication can depend on various factors, including the size of the data being transferred, the topology of the network, and the particular MPI library implementation.

## MPI\_Bcast and MPI\_Gather implementation. [The best performing one]

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

```
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.

```
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 argument to MPI\_Bcast? We broadcasted the current (or you can say the old) grid to all the tasks, and we gathered 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 broadcast it again, this time as the current grid to all the processors.

[https://www.mpich.org/static/docs/latest/www3/MPI\\_Gather.html](https://www.mpich.org/static/docs/latest/www3/MPI_Gather.html)

```
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 walk through our code:

### Libraries used:

- MPI.h**: Facilitates parallel computing by providing an API for communication between processes in a distributed computing environment.
- Stdio.h**: Enables input/output operations, including printing simulation results to the console.
- Stdlib.h**: Provides general-purpose functions, including dynamic memory allocation, which is used for creating the simulation grid.
- Math.h**: While included, this library is not explicitly used in the current implementation and can be removed if no mathematical functions are required.

### Constants

- X\_SIZE, Y\_SIZE**: Define the dimensions of the simulation grid.
- Cx, Cy**: The diffusion coefficients for the x and y dimensions, respectively.
- CMin, CMax**: Specify the central region of the grid with initial heat presence.
- TIMESTEPS**: The total number of simulation cycles to be executed.
- PRINT\_STEPS**: The interval at which the grid's state is output to the console.
- MASTER**: Denotes the rank of the master MPI process responsible for coordinating the simulation.

### Simulation Process

1. **Initialization**: The master process initializes the simulation grid with predefined temperature values, representing the initial state of the heat source.
2. **Distribution**: The grid is then distributed to all processes using **MPI\_Bcast**. Each process receives a copy of the initial grid to work on a subset of rows.
3. **Computation**: Each process computes the temperature changes for its assigned rows based on the heat diffusion equation.

```

MPI_Bcast(grid, X_SIZE * Y_SIZE, MPI_FLOAT, MASTER, MPI_COMM_WORLD);

int rows_per_process = X_SIZE / size;
int start_row = rank * rows_per_process;
int end_row = (rank == size - 1) ? X_SIZE : start_row + rows_per_process;

double start_time = MPI_Wtime();

for (int step = 0; step <= Timesteps; ++step) {
    // Update the new_grid based on the current grid's values
    update_grid(grid, new_grid, start_row, end_row);

    // Synchronize and update the grid for the next step
    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);

    if (rank == MASTER) {
        print_cycle_temperatures(grid, step);
    }

    // Broadcast the updated grid for the next timestep
    MPI_Bcast(grid, X_SIZE * Y_SIZE, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
}

```

```

void update_grid(float (*grid)[Y_SIZE], float (*new_grid)[Y_SIZE], int
start_row, int end_row) {
    for (int i = start_row; i < end_row; ++i) {
        for (int j = 1; j < Y_SIZE - 1; ++j) {
            // Ensure we only update cells within the grid boundaries
            if (i > 0 && i < X_SIZE - 1) {
                new_grid[i][j] = grid[i][j] +
                                Cx * (grid[i + 1][j] + grid[i - 1][j] - 2 *
grid[i][j]) +
                                Cy * (grid[i][j + 1] + grid[i][j - 1] - 2 *
grid[i][j]);
            }
        }
    }
}

```

4. **Collection:** After each computation step, the master process collects the updated segments of the grid using **MPI\_Gather**.
5. **Synchronization:** The updated complete grid is then broadcasted again to all processes for the next computation cycle using **MPI\_Bcast**.

6. **Output:** At specified intervals, the master process prints the current state of the grid to the console.

```
void print_cycle_temperatures(float grid[X_SIZE][Y_SIZE], int cycle) {
    if (cycle % PRINT_STEPS == 0) {
        printf("Cycle: %-4d. ", cycle);
        printf("[1,1]: %f, [150,150]: %f, [400,400]: %f, [500,500]: %f,
[750,750]: %f, [900,900]: %f\n",
            grid[1][1], grid[150][150], grid[400][400], grid[500][500],
            grid[750][750], grid[900][900]);
    }
}
```

7. **Finalization:** Once all time steps are computed, the master process calculates the total execution time and prints it. The MPI environment is then finalized, and allocated memory is freed.

## Performance Analysis

The execution time is recorded using **MPI\_Wtime()**, providing a high-resolution time elapsed for the simulation. The chosen MPI functions, **MPI\_Bcast** and **MPI\_Gather**, are optimized for collective communication and have demonstrated superior performance in distributing and collecting data compared to point-to-point communications such as **MPI\_Send** and **MPI\_Recv**.

```
> git push
Everything up-to-date
> mpicc -o HW4 1 HW4 1.c
> mpirun -np 11 ./HW4 1
Cycle: 0 . [1,1]: 0.000000, [150,150]: 0.000000, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 500.000000, [900,900]: 0.000000
Cycle: 200 . [1,1]: 0.000000, [150,150]: 0.000000, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 500.000000, [900,900]: 0.000000
Cycle: 400 . [1,1]: 0.000000, [150,150]: 0.000000, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 500.000000, [900,900]: 0.000000
Cycle: 600 . [1,1]: 0.000000, [150,150]: 0.000000, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 499.987640, [900,900]: 0.000000
Cycle: 800 . [1,1]: 0.000000, [150,150]: 0.000011, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 499.873962, [900,900]: 0.000000
Cycle: 1000 . [1,1]: 0.000000, [150,150]: 0.000182, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 499.479279, [900,900]: 0.000000
Cycle: 1200 . [1,1]: 0.000000, [150,150]: 0.001223, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 498.635529, [900,900]: 0.000000
Cycle: 1400 . [1,1]: 0.000000, [150,150]: 0.004856, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 497.254883, [900,900]: 0.000000
Cycle: 1600 . [1,1]: 0.000000, [150,150]: 0.013824, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 495.328705, [900,900]: 0.000000
Cycle: 1800 . [1,1]: 0.000000, [150,150]: 0.031480, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 492.900177, [900,900]: 0.000000
Cycle: 2000 . [1,1]: 0.000000, [150,150]: 0.061251, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 490.038086, [900,900]: 0.000000
Cycle: 2200 . [1,1]: 0.000000, [150,150]: 0.106207, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 486.819489, [900,900]: 0.000000
Cycle: 2400 . [1,1]: 0.000000, [150,150]: 0.168815, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 483.318909, [900,900]: 0.000000
Cycle: 2600 . [1,1]: 0.000000, [150,150]: 0.250847, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 479.604095, [900,900]: 0.000000
Cycle: 2800 . [1,1]: 0.000000, [150,150]: 0.353398, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 475.733612, [900,900]: 0.000001
Cycle: 3000 . [1,1]: 0.000000, [150,150]: 0.476959, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 471.756805, [900,900]: 0.000004
Cycle: 3200 . [1,1]: 0.000000, [150,150]: 0.621525, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 467.714691, [900,900]: 0.000010
Cycle: 3400 . [1,1]: 0.000000, [150,150]: 0.786698, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 463.640442, [900,900]: 0.000022
Cycle: 3600 . [1,1]: 0.000000, [150,150]: 0.971780, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 459.561005, [900,900]: 0.000046
Cycle: 3800 . [1,1]: 0.000000, [150,150]: 1.175863, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 455.497620, [900,900]: 0.000090
Cycle: 4000 . [1,1]: 0.000000, [150,150]: 1.397896, [400,400]: 500.000000, [500,500]: 500.000000, [750,750]: 451.467285, [900,900]: 0.000164
Execution Time: 15.192993 seconds
```

## Automation and output4.txt creation:

Similar to HW3, in which we created a simple Perl script to automate the creation of the output3.txt for the program using pthreads, we have slightly modified the same automation script to use it in our MPI program to create the output4.txt file. The scripts executes the HW4 program from 1 to 11 tasks on Mac, and 16 tasks on Linux, and calculates their speedup.

```
#!/usr/bin/perl

use strict;
use warnings;

# Output file path
my $output_file = '../output/Ghonim_Nordstrom_output4_(1)_mac_os1.txt';
# my $output_file = '../output/Ghonim_Nordstrom_output4_(2)_mac_os.txt';
# my $output_file = '../output/Ghonim_Nordstrom_output4_(1)_linux.txt';
# my $output_file = '../output/Ghonim_Nordstrom_output4_(1)_linux.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_process_time;

# Run the program with different numbers of processes
for (my $num_processes = 1; $num_processes <= 11; $num_processes++) {
    # Execute the MPI program and capture its output
    my $output = `mpirun -np $num_processes ./HW4_1`;
    #my $output = `mpirun -np $num_processes ./HW4_2`;

    # Extract the time from the output.
    my ($time_in_seconds) = $output =~ /Execution Time: ([\d\.]+) seconds/;

    if (defined $time_in_seconds) {
        # Calculate speedup. Avoid division by zero by checking if
$num_processes is 1.
        my $speedup = $num_processes == 1 ? 1 : ($single_process_time /
$time_in_seconds); # // is the defined-or operator, prevents use of
uninitialized value
        $single_process_time = $time_in_seconds if $num_processes == 1;


        # Write the results to the output file with fixed widths for each
column
        printf $fh "%-20d %-15f %-15f\n", $num_processes, $time_in_seconds,
$speedup;
    } else {
        warn "Could not extract execution time for $num_processes processes";
    }
}


# Close the file
```


```
close $fh;  
  
print "Results saved in $output_file\n";
```

```
> ./perl_automate.pl  
Results saved in ../output/Ghonim_Nordstrom_output4_(1)_mac_os1.txt  
~/Desktop/Winter24/ECE688/Homework/ECE688_HW4/src | main !3 ?1
```

The results after running the program on a Mac with 11 cores.

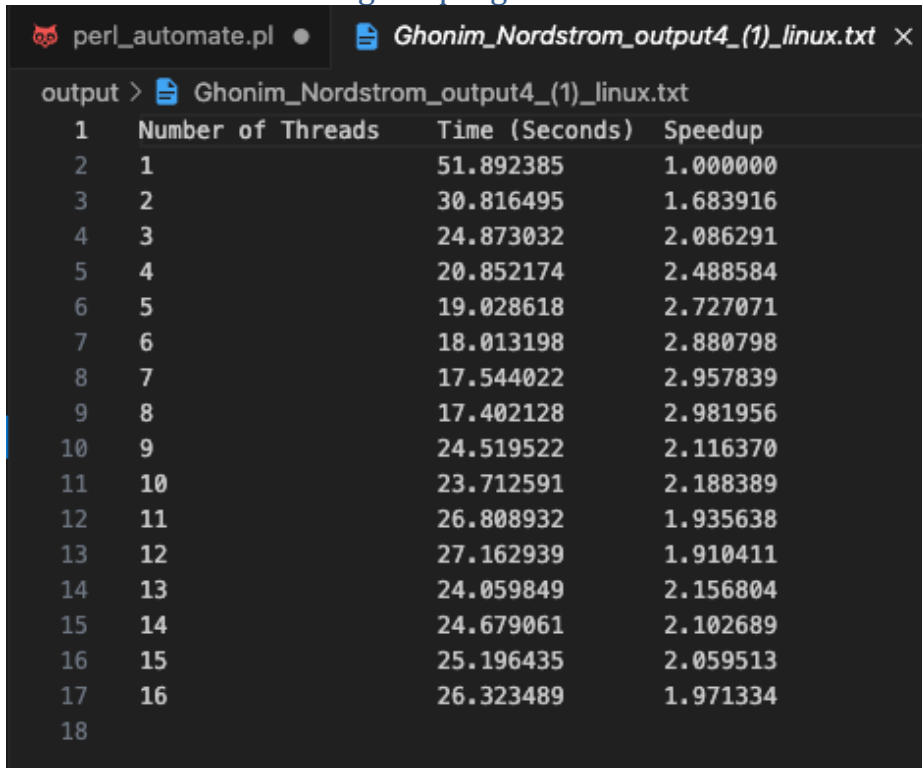

perl\_automate.pl M


Ghonim\_Nordstrom\_output4\_(1)\_mac\_os.txt
×

output >

Ghonim\_Nordstrom\_output4\_(1)\_mac\_os.txt

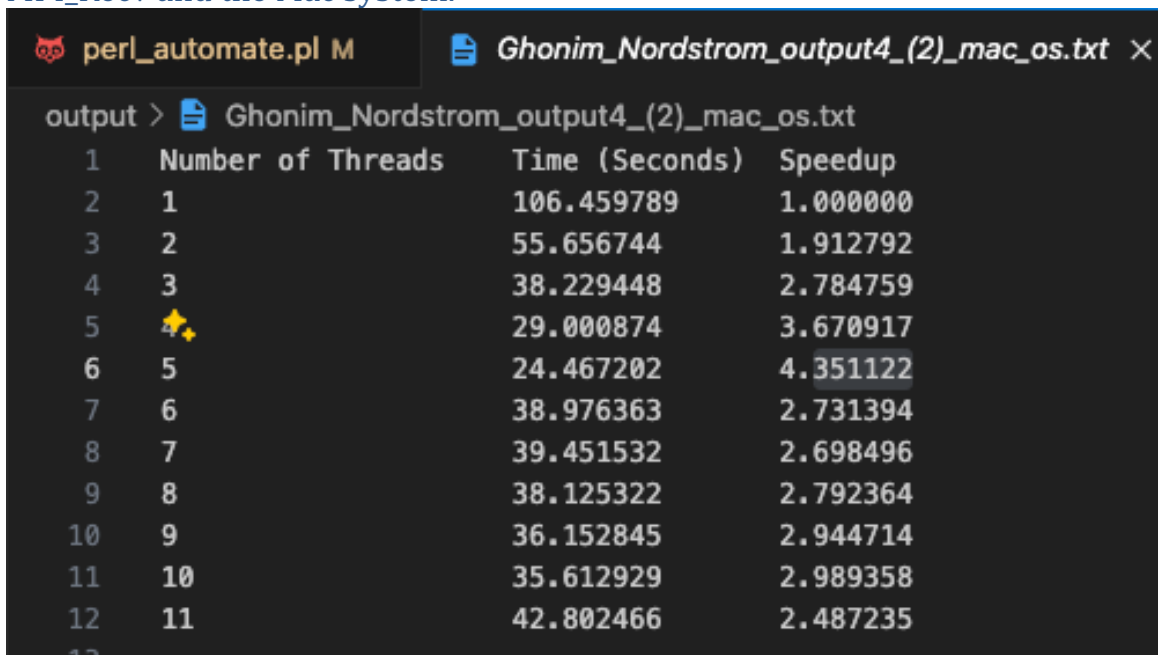
	Number of Threads	Time (Seconds)	Speedup
1	1	13.645267	1.000000
2	2	7.457060	1.829845
3	3	5.491243	2.484914
4	4	4.840131	2.819194
5	5	4.827044	2.826837
6	6	8.842850	1.543085
7	7	9.738743	1.401132
8	8	10.079142	1.353812
9	9	10.226562	1.334297
10	10	10.768287	1.267172
11	11	14.611524	0.933870

The results after running the program on a Linux with 16 cores.



	Number of Threads	Time (Seconds)	Speedup
1	1	51.892385	1.000000
2	2	30.816495	1.683916
3	3	24.873032	2.086291
4	4	20.852174	2.488584
5	5	19.028618	2.727071
6	6	18.013198	2.880798
7	7	17.544022	2.957839
8	8	17.402128	2.981956
9	9	24.519522	2.116370
10	10	23.712591	2.188389
11	11	26.808932	1.935638
12	12	27.162939	1.910411
13	13	24.059849	2.156804
14	14	24.679061	2.102689
15	15	25.196435	2.059513
16	16	26.323489	1.971334

Results from an early experimentation on the program using MPI\_Send and MPI\_Recv and the Mac system.



	Number of Threads	Time (Seconds)	Speedup
1	1	106.459789	1.000000
2	2	55.656744	1.912792
3	3	38.229448	2.784759
4	4	29.000874	3.670917
5	5	24.467202	4.351122
6	6	38.976363	2.731394
7	7	39.451532	2.698496
8	8	38.125322	2.792364
9	9	36.152845	2.944714
10	10	35.612929	2.989358
11	11	42.802466	2.487235

We checked the linux system and found that it has 32 cores.



```

ghonim@mo:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             32
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                42
Model name:            Intel Xeon E312xx (Sandy Bridge, IBRS update)
Stepping:              1
CPU MHz:               2599.998
BogoMIPS:              5199.99
Virtualization:        VT-x
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              4096K
L3 cache:              16384K
NUMA node0 CPU(s):     0-31
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge m
rdtscp lm constant_tsc rep_good nopl xtopology eagerfpu pni pclmulqdq vmx ss
xsave avx hypervisor lahf_lm ssbd rsb_ctxsw ibrs ibpb stibp tpr_shadow vnmi t
ctrl intel_stibp arch_capabilities
ghonim@mo:~$

```

## Summary of Findings:

### Mac System with 11 Cores (Using MPI\_Bcast and MPI\_Gather):

- **Observations:**
  - Optimal performance was noted up to 5 cores, with a speedup reaching approximately 2.83 times faster than the single-threaded execution.
  - Beyond 5 cores, the performance gain plateaued, and further increments in the number of cores resulted in decreased speedup, with a notable decrease in efficiency at 6 cores and beyond.
  - Using all 11 cores resulted in a speedup less than that of a single core, indicating significant overhead that negates the benefits of parallelization.

### Linux System with 16 Cores (Using MPI\_Bcast and MPI\_Gather):

- **Observations:**
  - The Linux system showed steady speedup increments up to 8 cores, peaking at almost a 3 times speedup.
  - Beyond 8 cores, the speedup gain diminished, and with more than 9 cores, the overhead began to outweigh the parallelization benefits, similar to the Mac system.
  - The highest observed speedup was with 8 cores, after which the efficiency started to degrade.

### Early Experimentation on Mac System (Using MPI\_Send and MPI\_Recv):

- **Observations:**
  - The speedup increased consistently up to 4 cores, yielding a 3.67 times faster execution than the single-core run.
  - Beyond 4 cores, the speedup did not improve significantly and even decreased when using more than 5 cores.

### Comparative Analysis:

- The collective communication implementation using **MPI\_Bcast** and **MPI\_Gather** proved more efficient than the point-to-point communication method. The collective approach provided a more consistent and significant speedup across both systems.
- Both systems exhibited diminishing returns in speedup beyond a certain point: after 5 cores on the Mac system and 8 cores on the Linux system. This trend suggests that the algorithm's inherent parallelization potential may be limited by factors such as communication overhead and the diminishing returns of adding more processing units.
- The speedup's drop beyond the optimal number of cores indicates that the overhead of managing additional MPI tasks becomes detrimental. This overhead could include the time taken to distribute and gather data, synchronization between processes, and possible contention for shared resources.
- The initial implementations using **MPI\_Send** and **MPI\_Recv** exhibited less efficiency compared to the revised version with **MPI\_Bcast** and **MPI\_Gather**. This difference highlights the importance of choosing the

right MPI primitives that align with the communication pattern of the application.

#### Conclusions:

- The parallelized heat diffusion simulator achieves considerable speedup through parallel execution but only up to a certain point, which varies depending on the system's architecture and the number of available cores.
- The use of collective communications (**MPI\_Bcast** and **MPI\_Gather**) is validated as a more efficient strategy compared to direct point-to-point communication methods for this particular application.
- Identifying the optimal number of cores for execution is critical to maximizing efficiency. In the case of the Mac system, this was up to 5 cores, whereas for the Linux system, it was 8 cores.
- As the core count increases beyond the optimal number, the performance gain from parallelization is outweighed by the overhead introduced by additional MPI tasks, which suggests the need for careful consideration of the computing resources relative to the workload.

These findings can guide future optimization and scalability considerations for similar parallel applications.