High Performance Computing

The Stencil Method

Zahra Younes Pour Langaroudi

University Of Trieste

Part 1

Introduction and Setup

Introduction and Setup

- **Problem:**

  Solve the 2D Heat Equation using the five-point stencil method.

- **Goal:**

  Parallelize the serial code using a hybrid MPI + OpenMP model to achieve high performance.

- **Platform:**

  Leonardo Supercomputer (2 sockets, 56 cores).

- **Setup:**

  OpenMP threads are placed on cores with a spread binding policy to maximize memory bandwidth.

Part 2

OpenMP

OpenMP

update_plane function

```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    double t0 = omp_get_wtime();

    #pragma omp for collapse(2) schedule(static)
    for (uint j = 1; j <= ysize; j++){
        for ( uint i = 1; i <= xsize; i++)
        {
            double alpha = 0.6;
            double result = old[IDX(i,j)] * alpha;
            double sum_i  = (old[IDX(i-1, j)] + old[IDX(i+1, j)]) / 4.0 * (1.0-alpha);
            double sum_j  = (old[IDX(i, j-1)] + old[IDX(i, j+1)]) / 4.0 * (1.0-alpha);
            result += (sum_i + sum_j );
            new[IDX(i,j)] = result;
        }
    }
    double t1 = omp_get_wtime();
    #pragma omp atomic
    g_per_thread_comp_time[thread_id] += (t1 - t0);
}
```

OpenMP scheduler:

Tested: static, dynamic, guided
Result: Static scheduling was optimal for our regular, uniform workload.

OpenMP

get_total_energy function

```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    double t0 = omp_get_wtime();

    #pragma omp for collapse(2) reduction(+:totenergy) schedule(static)
     for ( int j = 1; j <= ysize; j++ ){
        for ( int i = 1; i <= xsize; i++ ){
            totenergy += data[ IDX(i, j) ];
        }
    }
    double t1 = omp_get_wtime();
    #pragma omp atomic
    g_per_thread_comp_time[thread_id] += (t1 - t0);
}
```

reduction(+:totenergy)

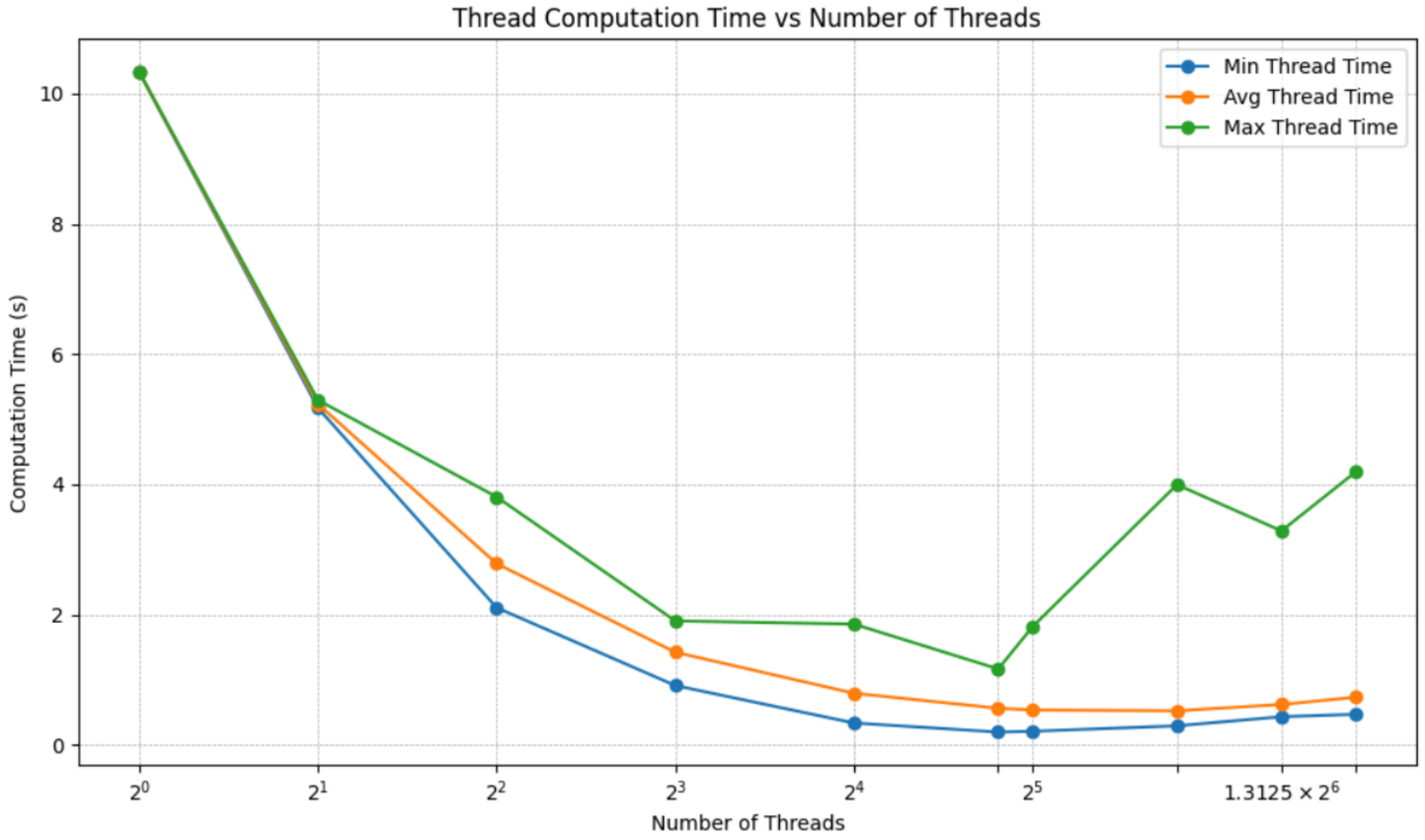
To accumulate the total energy across all threads

Part 3

OpenMP Results

Computation Time per Thread

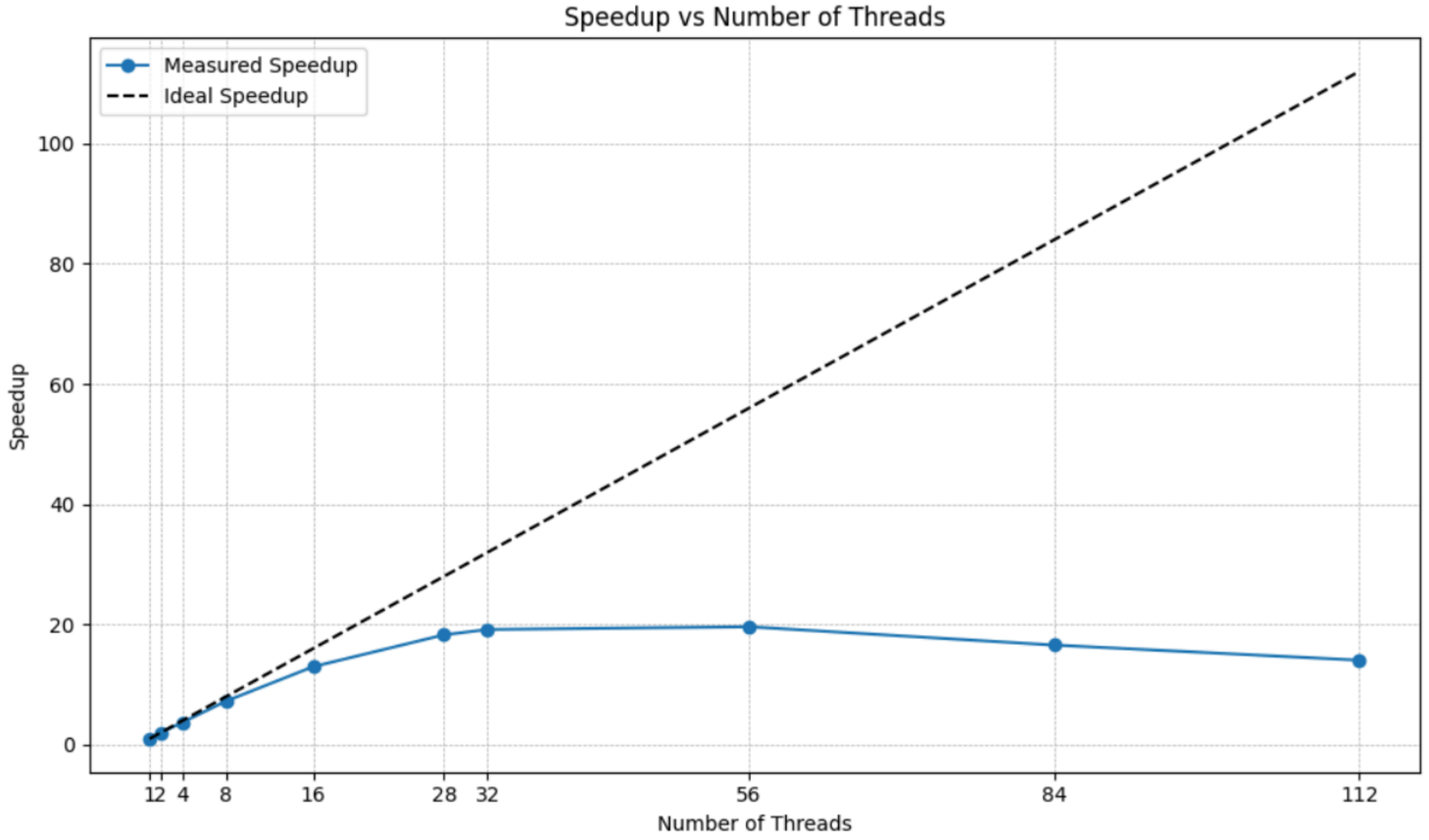
1 Node, 1 MPI Task, Threads number in (1, 2, 4, 8, 16, 28, 32, 56, 84, 112)



- Slowdown after 32 threads
- NUMA effects and memory access slowdowns.
- The guided scheduler.
- Work imbalance grows as more cores are used.

Speedup

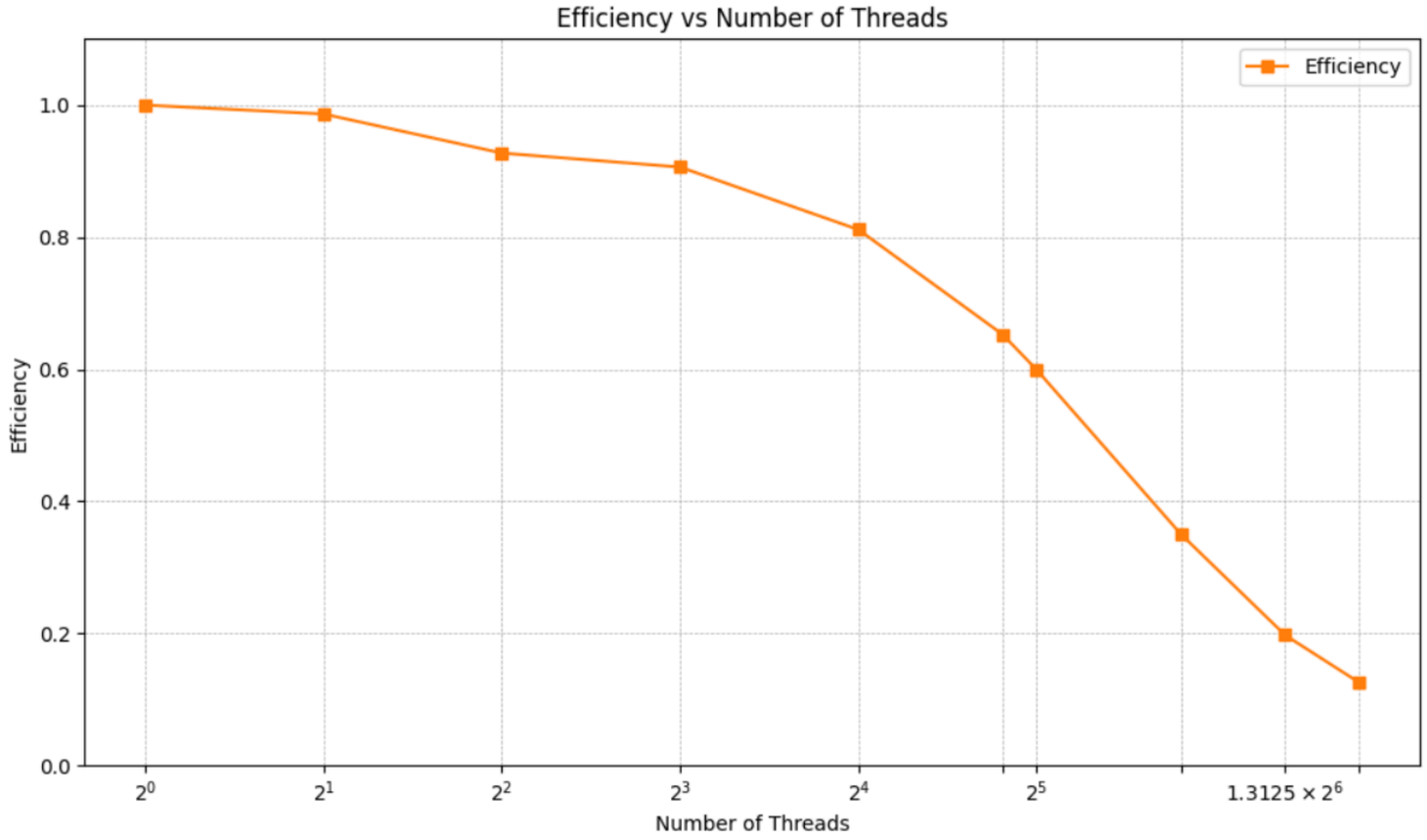
1 Node, 1 MPI Task, Threads number in (1, 2, 4, 8, 16, 28, 32, 56, 84, 112)



- Speedup is nearly linear up to 16 threads, then flattens.
- The gap between measured and ideal speedup is caused by the serial fraction of the code (Amdahl's Law) and parallel overhead.

Efficiency

1 Node, 1 MPI Task, Threads number in (1, 2, 4, 8, 16, 28, 32, 56, 84, 112)



$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

- Efficiency remains high for a low thread count, but decreases by adding more threads.
- **16 threads** per MPI task were chosen as the optimal for the next studies.

Part 4

MPI

MPI

Blocking Version

- Used an even-odd strategy:

  -Even-ranked processes receive first, then send.

  -Odd-ranked processes send first, then receive.
- Added conditional logic to handle each neighbor direction (north, south, east, west).
- Encountered deadlocks during testing
- Commented out the blocking version and switched to the non-blocking MPI

MPI

Non-Blocking Version | Receive

```
// Post all RECEIVES first
if (neighbours[NORTH] != MPI_PROC_NULL) {
    MPI_Irecv(&current_plane[0 * full_sizex + 1], sizex, MPI_DOUBLE, neighbours[NORTH], 1, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
if (neighbours[SOUTH] != MPI_PROC_NULL) {
    MPI_Irecv(&current_plane[(sizey + 1) * full_sizex + 1], sizex, MPI_DOUBLE, neighbours[SOUTH], 0, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
if (neighbours[EAST] != MPI_PROC_NULL) {
    MPI_Irecv(buffers[RECV][EAST], sizey, MPI_DOUBLE, neighbours[EAST], 3, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
if (neighbours[WEST] != MPI_PROC_NULL) {
    MPI_Irecv(buffers[RECV][WEST], sizey, MPI_DOUBLE, neighbours[WEST], 2, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
```

MPI

Non-Blocking Version | Send

```
// Then post all SENDS
if (neighbours[NORTH] != MPI_PROC_NULL) {
    MPI_Isend(&current_plane[1 * full_sizex + 1], sizex, MPI_DOUBLE, neighbours[NORTH], 0, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
if (neighbours[SOUTH] != MPI_PROC_NULL) {
    MPI_Isend(&current_plane[sizey * full_sizex + 1], sizex, MPI_DOUBLE, neighbours[SOUTH], 1, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
if (neighbours[EAST] != MPI_PROC_NULL) {
    MPI_Isend(buffers[SEND][EAST], sizey, MPI_DOUBLE, neighbours[EAST], 2, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
if (neighbours[WEST] != MPI_PROC_NULL) {
    MPI_Isend(buffers[SEND][WEST], sizey, MPI_DOUBLE, neighbours[WEST], 3, myCOMM_WORLD, &requests[request_count]);
    request_count++;
}
```

MPI

Non-Blocking Version

```
// --- Wait for all communications to complete ---
// MPI_Waitall will pause here until every single Isend and Irecv is finished.
MPI_Waitall(request_count, requests, MPI_STATUSES_IGNORE);
```

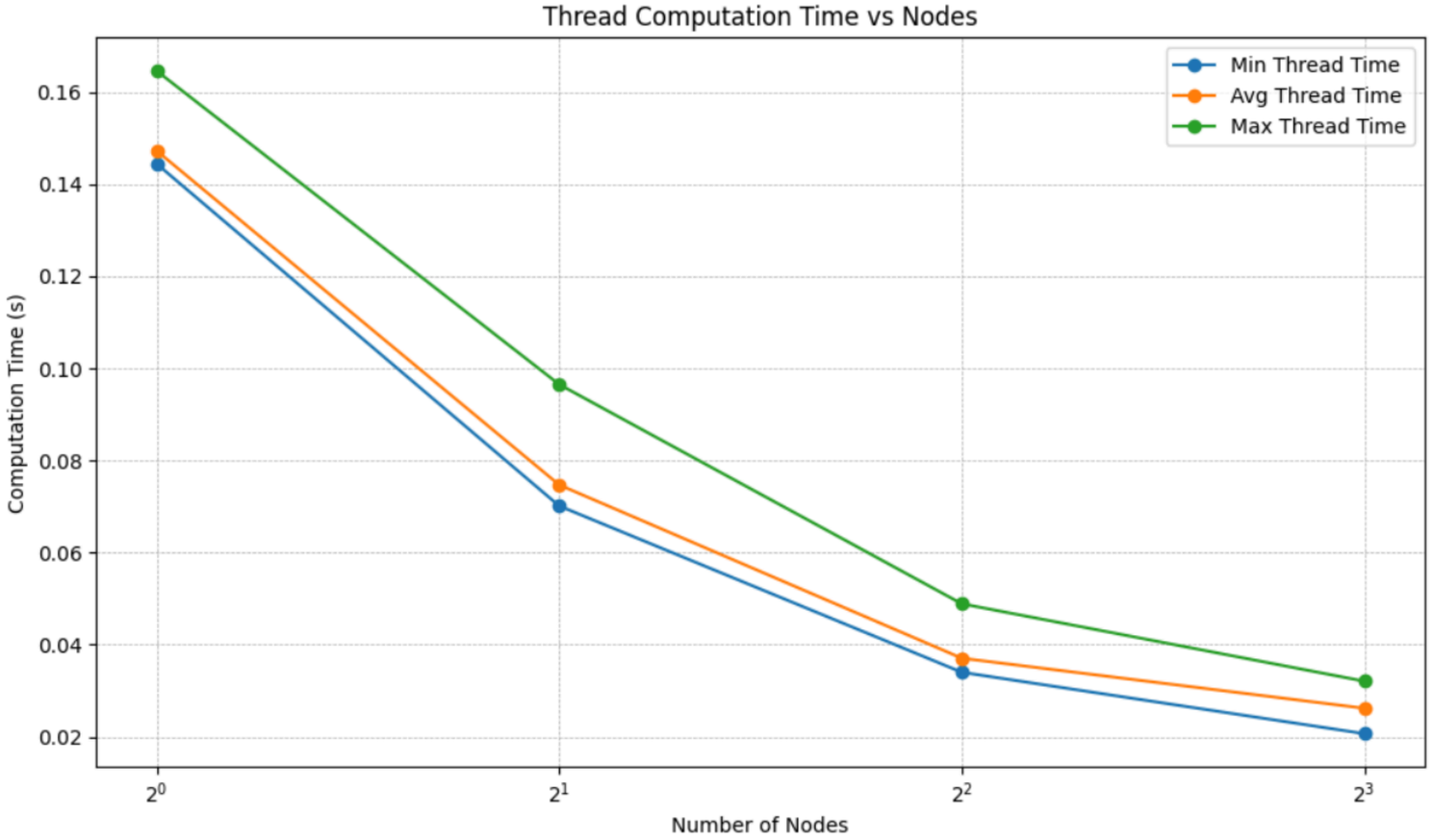
To enable true communication-computation overlap:
Split the update into interior and boundary
Compute the interior while waiting for the halo exchange to finish
Then update the boundary after MPI_Waitall()

Part 5

Strong Scaling Results

Computation Time per Node

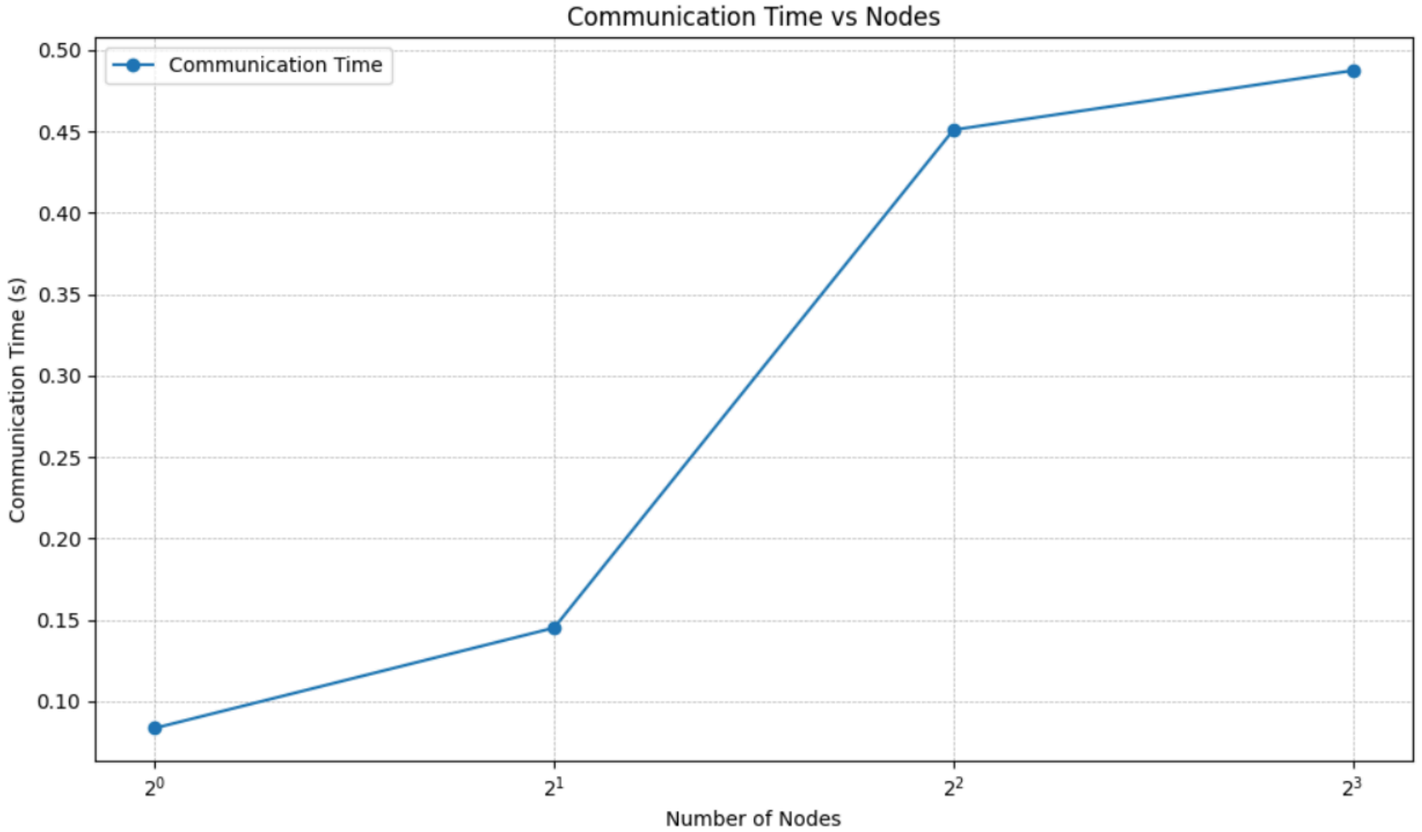
Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads



- Adding more nodes reduces computation time.
- Min, avg, and max thread times all decrease, indicating good parallel efficiency.
- Static scheduler

Communication Time

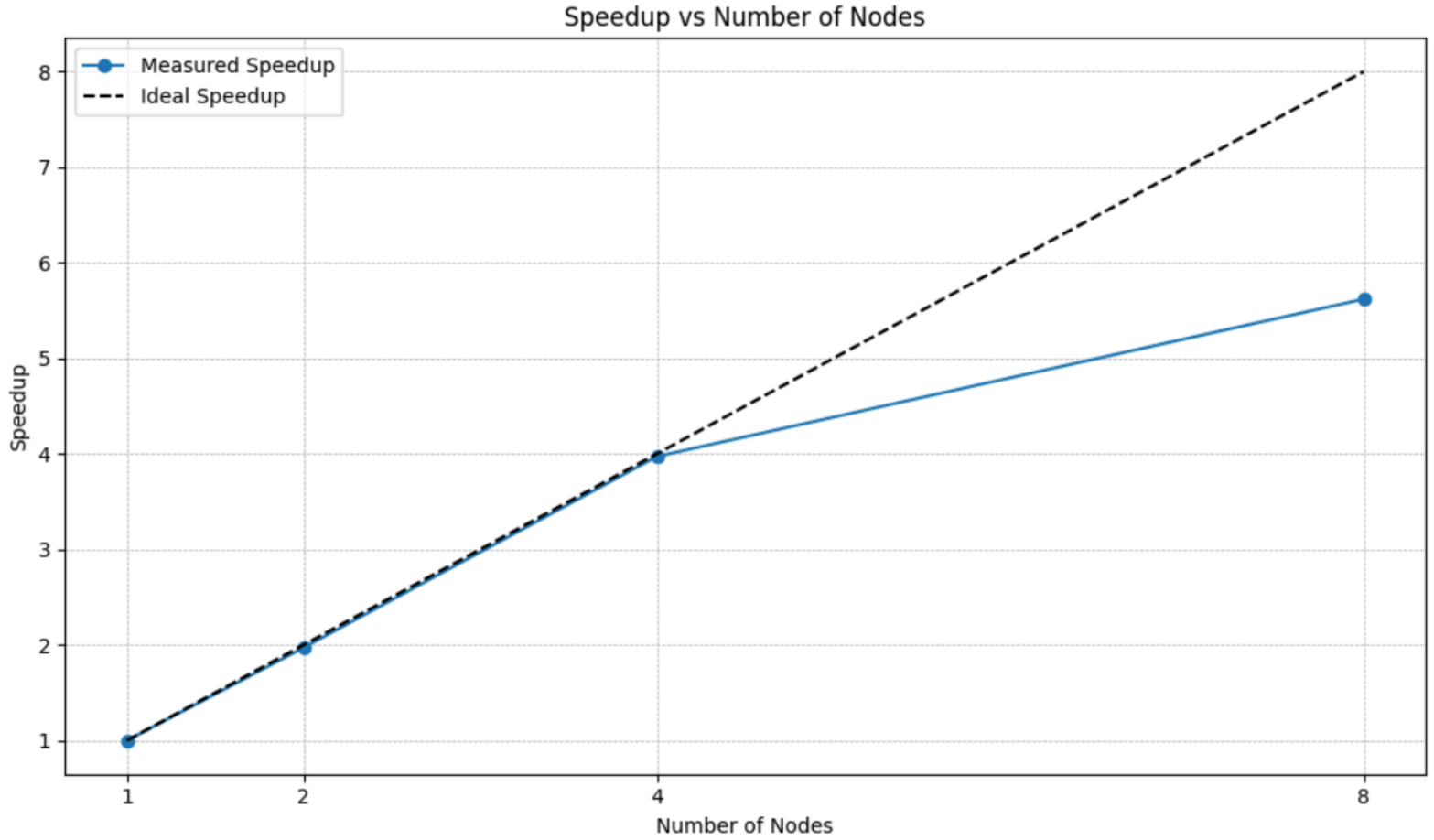
Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads



- As more nodes are added, the communication time increases.
- This is because the total communication volume grows.
- Network Latency

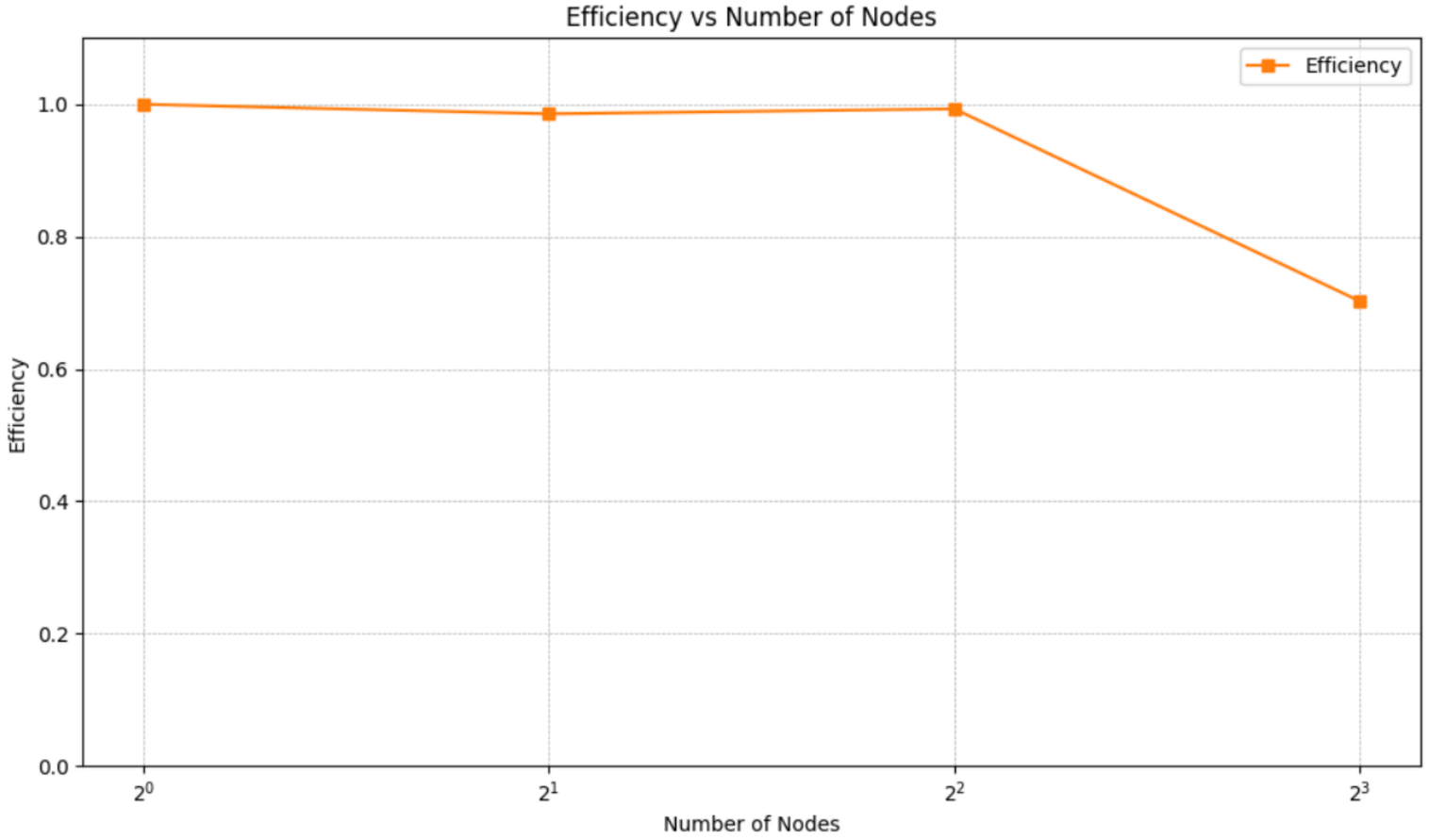
Speedup

Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads



Efficiency

Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads

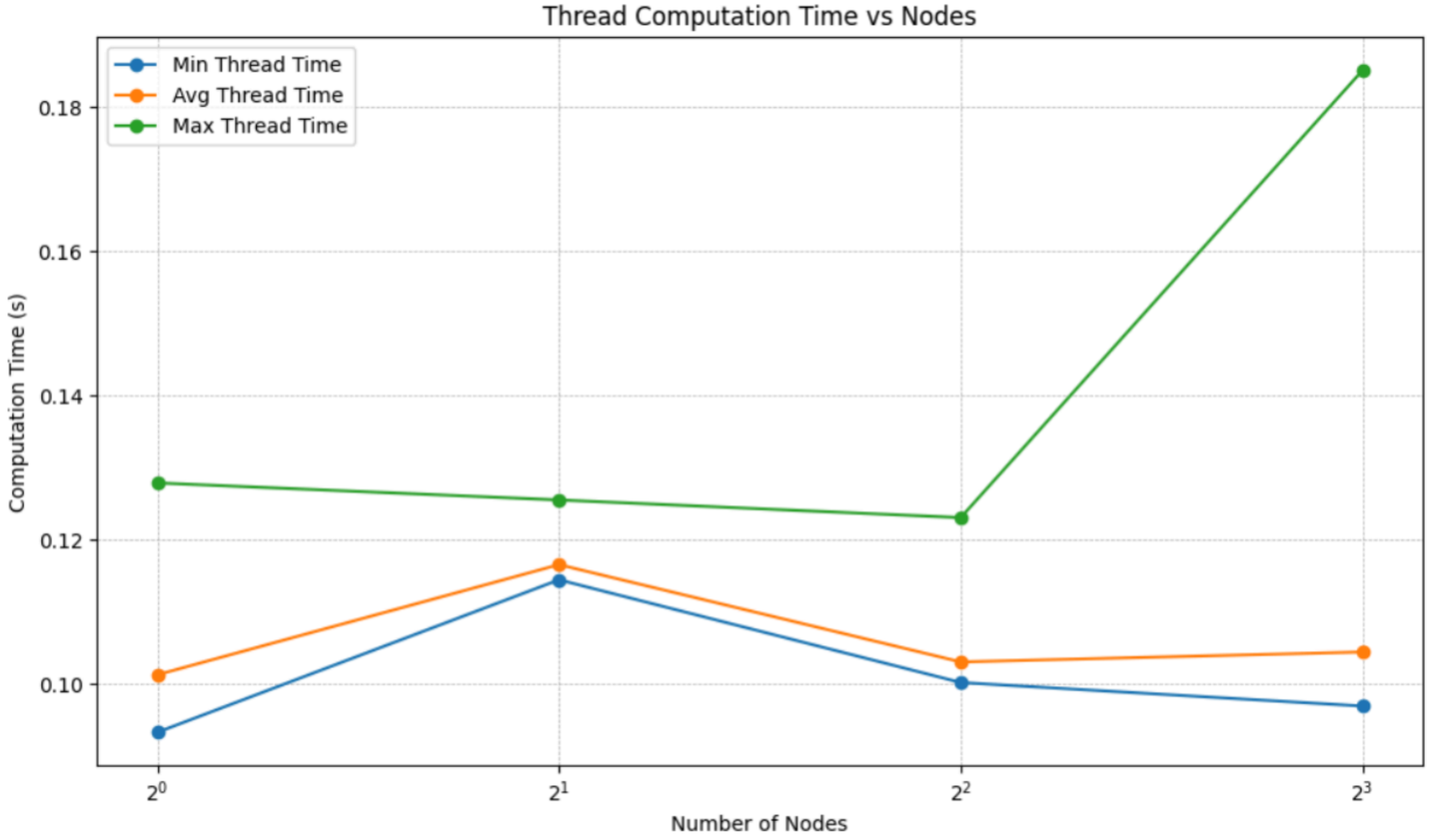


Part 6

Weak Scaling Results

Computation Time per Node

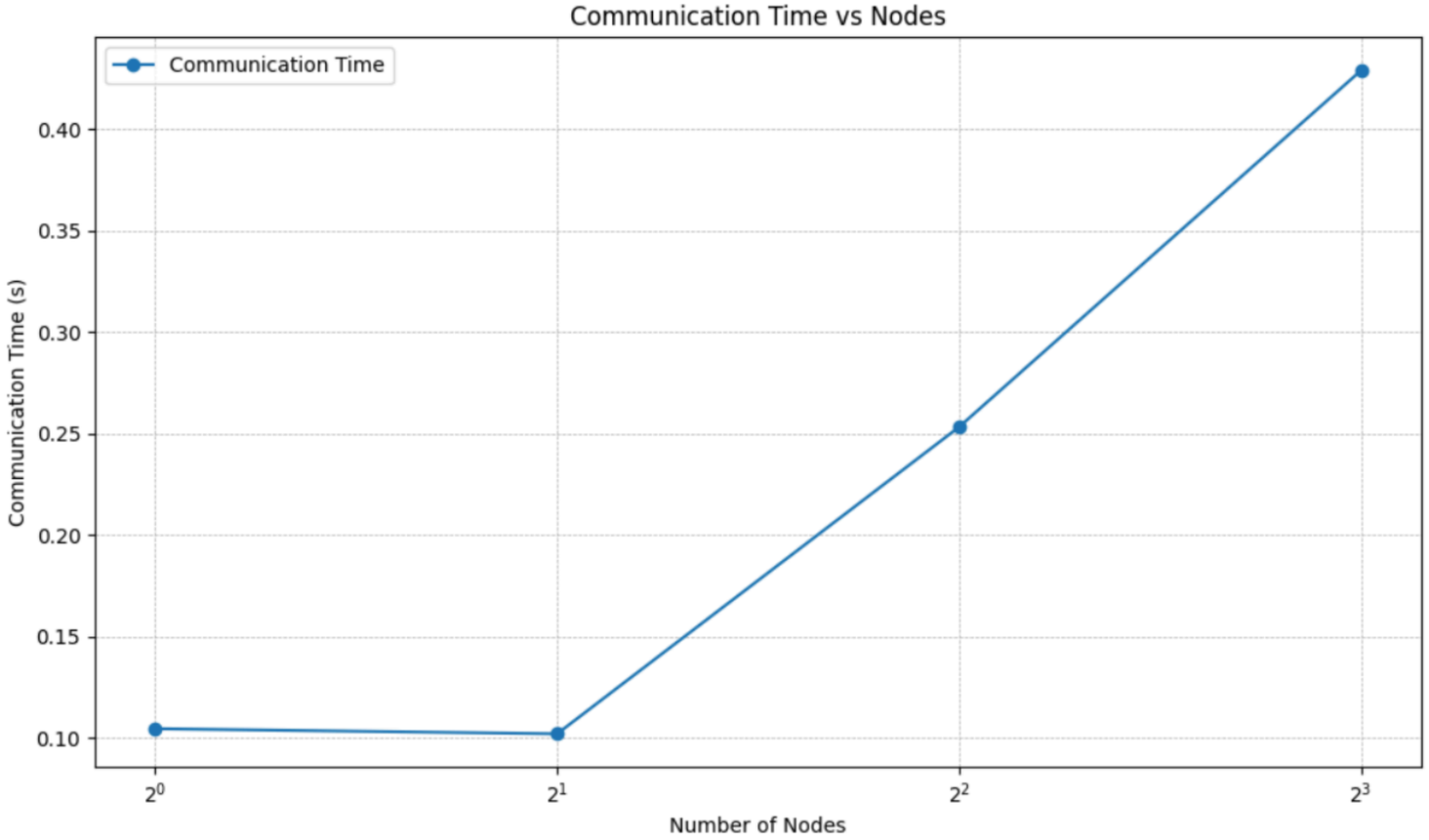
Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads



- The computation time per node remains nearly constant.
- Load imbalance for very large total problem sizes.

Communication Time

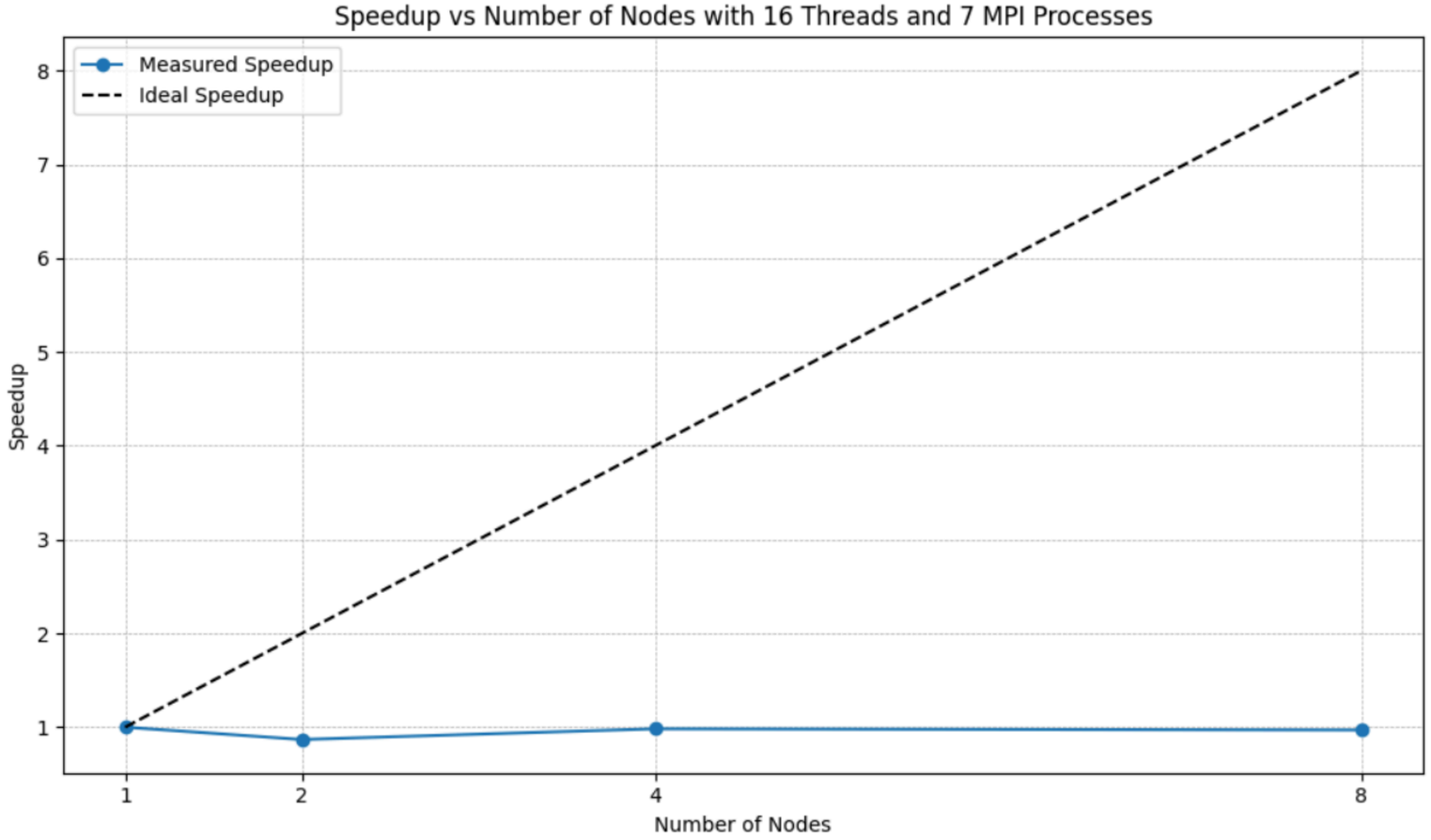
Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads



- Communication time increases with the number of nodes.
- More total data is being exchanged over the network.

Speedup

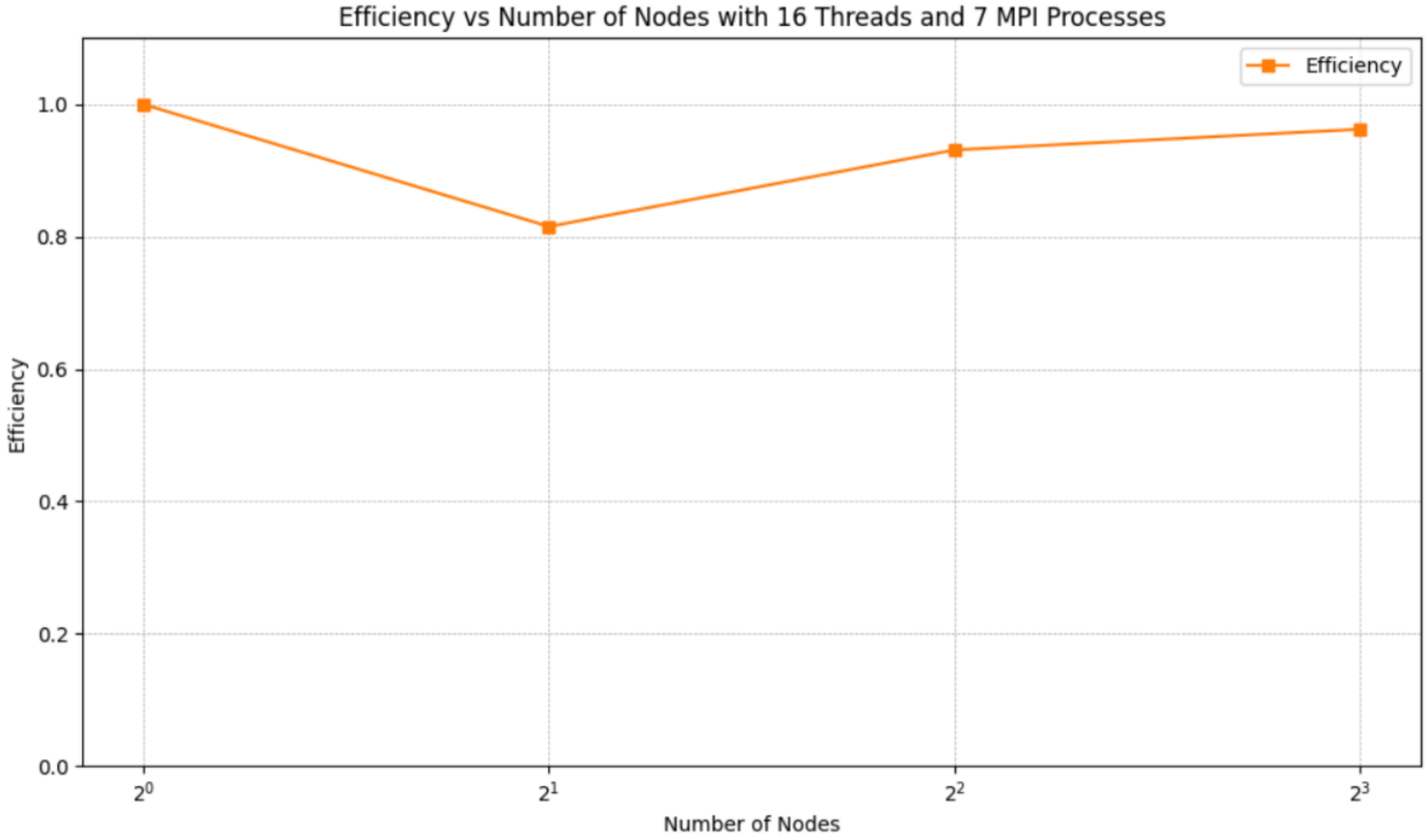
Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads



- The weak scaling speedup is nearly flat.
- Fix problem size per node.
- The code is capable of solving larger problems with more resources.

Efficiency

Nodes number in (1, 2, 4, 8), 7 MPI Task, 16 Threads



Part 7

Conclusion and Future Work

Conclusion

- Successfully parallelized a 5-point stencil code with a hybrid MPI+OpenMP model.
- The OpenMP study identified an optimal configuration of 16 threads per task.
- The MPI studies showed good strong scaling limited by communication overhead and effective weak scaling for solving larger problems.

Future Work

- Overlap Communication and Computation: Implement a split update_plane function to compute the grid interior while the non-blocking MPI halo exchange runs in the background. This would hide communication latency.
- See the result for OpenMP bind close to the core.
- See the result for more nodes in strong and weak scaling.

Thank you & Questions