CX 4803: Parallel 2D Heat Equation Solver

Suzan Manasreh

April 30 2025

1 Introduction

Generally, me model heat in 1D (x) space using the following equation [1].

$$\frac{\delta}{\delta T}T(x,t) - \alpha\frac{\delta^2}{\delta x^2}T(x,t) = q(x,t). \tag{1}$$

Where $T(x,t)$ is the temperature at time t for position x. α is the heat coefficient that controls how fast heat spreads— the thermal conductivity. We take $\alpha = .1$ simulation units for our experiments unless otherwise specified. $q(x,t)$ is the outside forcing function on position x at time t.

The 2D heat equation that we will be solving is defined as:

$$\frac{\delta}{\delta T}T(x,y,t) - \alpha(\frac{\delta^2}{\delta x^2} + \frac{\delta^2}{\delta y^2})T(x,y,t) = q(x,y,t). \tag{2}$$

We apply homogeneous Dirichlet boundary conditions in the x and y directions [3]:

$$T(0,y,t) = T(a,y,t) \qquad \text{for} \qquad 0 \le y \le b,\ t \ge 0 \tag{3}$$

$$T(x,0,t) = T(x,b,t) \qquad \text{for} \qquad 0 \le x \le a,\ t \ge 0 \tag{4}$$

This means that the area around the 2D grid is all initialized to 0, a constant boundary condition. The innermost space also gets an initial uniform temperature, so the forcing function q does not depend on time. This guarantees that the solution will converge to a steady state [1]. In general, if the initial temperature of all points inside the boundary $T(x,y,0) = 10$, we should reach $T(x,y,t) = 0$ at a later time as heat dissipates.

In 1D, we can discretize the heat equation at position j and time step k as:

$$T_j^{k+1} = T_j^k + \frac{\alpha\Delta t}{\Delta x^2}(T_{j-1}^k - 2T_j^k + T_{j+1}^k) + \delta t q_j^k \tag{5}$$

In our simulation, we can ignore the last term as q is constant. This discretization uses an explicit Euler forward time step. The temperature at the next time step is computing using only the surrounding values at the current step.

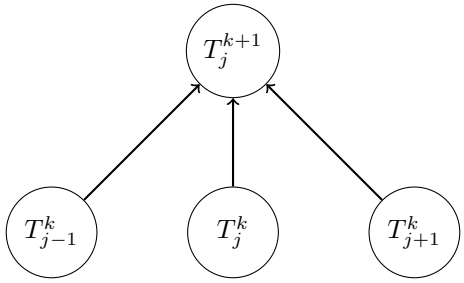

Figure 1: 1D Heat Equation Stencil Update

We can discretize the 2D heat equation in a similar manner:

$$T_{i,j}^{k+1} = T_{i,j}^k + \frac{\alpha \Delta t}{\Delta x^2}(T_{i-1,j}^k - 2T_{i,j}^k + T_{i+1,j}^k) + \frac{\alpha \Delta t}{\Delta y^2}(T_{i,j-1}^k - 2T_{i,j}^k + T_{i,j+1}^k) \quad (6)$$

Like the 1D heat equation, this is a simple stencil udpate at each point that depends on the surrounding values from the previous time step.

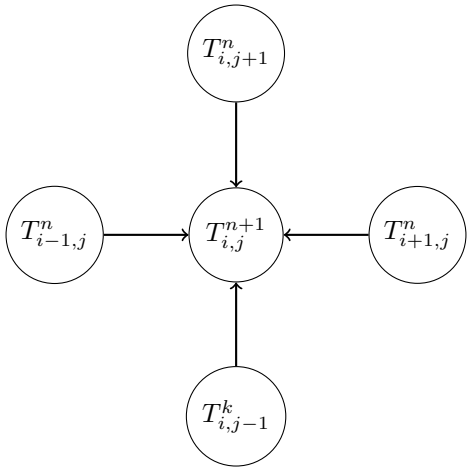

Figure 2: 2D Heat Equation Stencil Update

The goal of this project is to simulate the 2D heat equation on a large grid (tens of thousands of cells) for many time steps (tens of thousands). Since the temperature at each point is computed using only the surrounding cells, we can decompose our grid into blocks. Each processor computes the values for the cells in its block. Of course values at boundaries with other processor will have to be exchanged at each time step. To do this, we will use MPI to split work between processors and exchange information. Within each block that the

processor works on, we will also use OpenMP parallelization to take advantage of shares memory between processor cores. This will help us further decompose the cells within each processor's block and parallelize operations.

There are many 2D heat equation simulation codes with MPI available like that of Dournac's, but we will further optimize on space by having each processor only store the cells within the block it's responsible for [2]. This will increase cache hit rates because the processor will always be bringing in rows it's responsible for when taking advantage of spatial locality. The use of hybrid parallelization will also allow OpenMP to better handle shared memory cores instead of assigning MPI processes to each core on a physical processor and having the overhead of the MPI library increase execution time.

We go over our code, parameter values, and parallelism/design choices in the Methods section. Results from our strong and weak scaling experiments are provided in the Results section along with verification test results. Lastly, we finish with key results summarized in our Conclusion.

2 Methods

For the simulation code, I used `C++`, so I can take advantage of the `C++` standard library and other libraries like Boost for contiguous vectors in memory. My codebase contains a sequential 1D heat simulation code called `main_1d.cpp` that I used to introduce myself to solving PDE's.

Then, I implemented a sequential version of the 2D heat equation solver described above. The thread of execution is responsible for the entire grid of cells and there's no domain decomposition. This made it simple to write out the grid points to a VTK file as a structured points dataset and visualize it in Paraview. This sequential code was useful for my verification tests against the parallel version of the 2D heat equation solver.

Finally, I implemented the parallel version of the 2D heat equation solver. This was similar to my sequential code, but I introduced several new parameters to help with grid decomposition.

In the sequential code, I had `x_dim` and `y_dim` variables to define the number of dimensions in the x and y directions. The x and y dimensions can be split across different numbers of x and y-domains. For example, we can have a 100×100 grid with 2 x-domains and 4 y-domains, so we get 8 blocks on our grid and need 8 processors in total specified by `MPI`. We visualize the layout of the processors on this grid in the figure below when we use `MPI` to assign processor ranks in a new Cartesian grid communicator. `MPI` also causes the x and y-axes to be flipped, but this is not a problem if we take it into account in our code.

Using `MPI` this way takes care of the load balancing problem as each processor is only responsible for a subset of the full grid. In our example, this would be $1,250$ cells per processor instead of $10,000$.

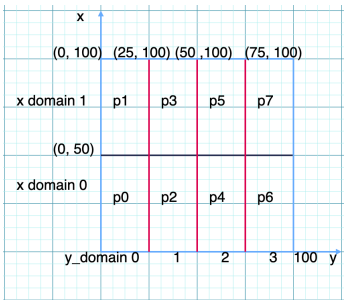

Figure 3: Processor ranks on simulation grid.

As mentioned earlier, each processor performs a stencil update for all the cells in its block. When this cell is on a border with no other `MPI` processes— such as proccessor 0's left and down borders— we can just use the value 0 at the border from our Dirichlet boundary conditions. However, when the border is between another processor, this processor has to send its border value to the other processor. Instead of using another array to store our border values, we use the original 2D array extended with ghost cells.

There's a set of ghost cells on every processor's up, down, left, and right boundary. These ghost cells are stored with each processors local 2D array, but they only add 2 rows in the x-direction and 2 rows in the y-direction. The figure below provides a visual of what these ghost cell areas at the end of each processor's domain look like. Note that this figure is not drawn to scale as each ghost area is just one row or column.

Using these ghost areas, each processor can easily pass contiguous blocks of cells in the up/down directions using `MPI_Sendrecv`. However to pass ghost cells in the left/right direction, we create an `MPI` vector that stores the values for all x cells in that column with a stride equal to the number of y cells in a row. Then, we can use `MPI_Sendrecv` to send columns from one processor to its processor on the right in the Cartesian grid.

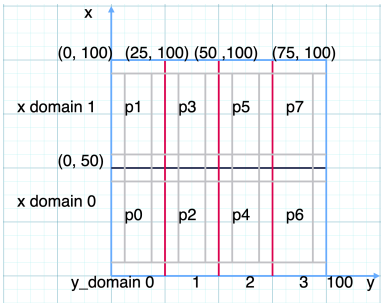

Figure 4: Processor ranks with ghost areas on the simulation grid.

Finally, for the time stepper section of our code, we first check if the parameter dt provided is too large for a stable solution, and if it is, we set it to its maximum allowed value. The following equation constrains dt where $h_x = 1/x_{dim}$ and $h_y = 1/y_{dim}$. We think of 1 as the fixed length in the x and y dimensions.

$$dt \leq \frac{min(h_x, h_y)}{4\alpha} \tag{7}$$

Our time stepper loops through all the time steps and updates the value of all cells belonging to that block. Then, we do the series of `MPI` send/receives described above and check if the solution converged. We define a threshold of .01 where if the difference between the previous iteration and the current iteration is less than this threshold, the solution has converged, and we can quit the time stepper loop.

This time stepper loop is also where we bring in the bulk of the `OpenMP` parallelization:

```
for (k = 1; k <= num_steps; k++) {
    double diff = 0;
    double global_diff = 0;

    #pragma omp parallel for reduction(+:diff)
    for (int i = x_start; i <= x_end; i++) {
        for (int j = y_start; j <= y_end; j++) {
            prev[i][j] = x[i][j];
            x[i][j] += (sx * (x[i + 1][j] - (2*x[i][j])
              + x[i - 1][j])) + (sy * (x[i][j + 1]
              - (2*x[i][j]) + x[i][j - 1]));
            diff += square(x[i][j] - prev[i][j]);
        }
    }
    // MPI_SendRecv's
    // Convergence checks
}
```

We use `OpenMP` to perform a reduction on the `diff` variable for each cell. The reduction clause reduces the contention between `OpenMP` threads on the `diff` variable [4]. When I added this and tested it on 4 MPI processors with 8 cores each, it resulted in a $5.76x$ speedup over the MPI-only code on 4 MPI processors.

We also used `OpenMP` when we gather all the grid cell values onto one processor and rearrange them so they can be written out in VTK format, but the speedup here from using `OpenMP` was negligible compared to the previous speedup.

3 Results

To evaluate our code, we performed strong scaling experiments first where we set `x_dim = 1000` and `y_dim = 1000` for a 1000×1000 grid. We run it for `num_steps = 100`, and the solution does not converge early, so it runs to completion. In Figure 6, we calculate speedup as the execution time on 1 processor with the same number of OMP threads over the execution time with more MPI processors.

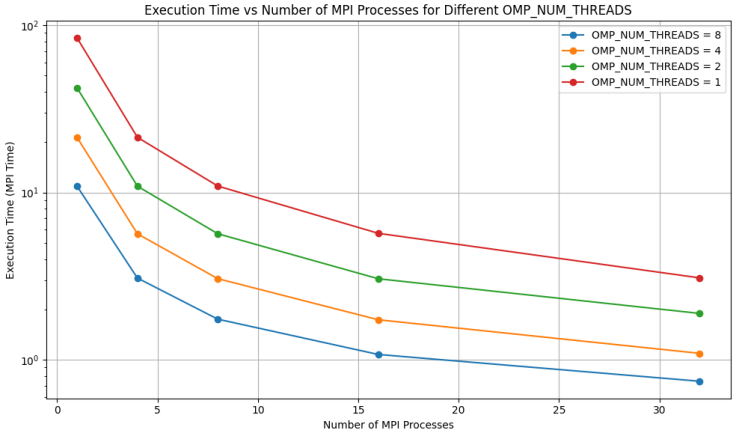

Figure 5: Strong Scaling Experiment with Run-times

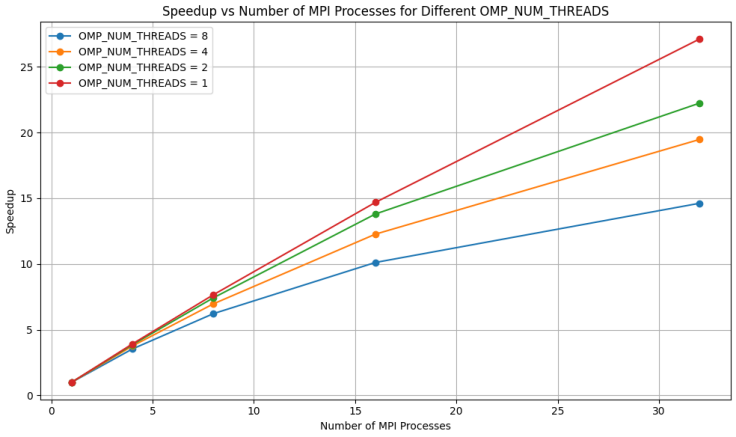

Figure 6: Strong Scaling Experiment with Speedups

As we can see, our code provides near perfect scaling as the speedup is close to the number of MPI processes. The speedup is less close to perfect as we increase the number of `MPI` processes. This is most likely due to increased communication load on each processor as they have to communicate in all four directions instead of one or two. The `OMP_NUM_THREADS = 1` case is the closest to perfect speedup and this tapers off as we increase the number of OMP threads. This is probably due to `OpenMP` having to manage more threads as we increase the thread count and adding more overhead to the execution time.

Next we perform weak scaling experiments where we fix the block size per processor to a 100×100 grid. For example, if we have 16 processors, so that `x_domains = 4` and `y_domains = 4`, we set `x_dim = 400` and `y_dim = 400`. The simulation has 100 time steps, so it runs to completion without converging.

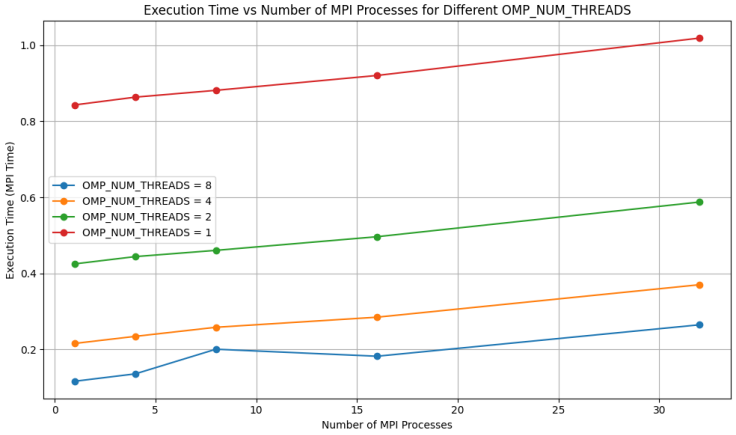

Figure 7: Weak Scaling Experiment with Run-times

Our weak scaling is near ideal because the execution time is fairly constant as the number of processors increases. Execution time does increase slightly however as number of processes increases. That's most likely due to increased communication overhead between processes as our algorithm does perform `MPI` communication at every step. This steady increase does not seem to vary with different numbers of OMP threads per processor.

Lastly, we also did verification tests to ensure that our parallel code did not provide different results from our simple, sequential code. Of course, we can verify this visually by seeing thermal diffusivity occur in the simulation for both sequential and parallel code with no visual difference. Figure 8 shows the data captured at time step 100 with default maximum dt for the 1000×1000 grid case.

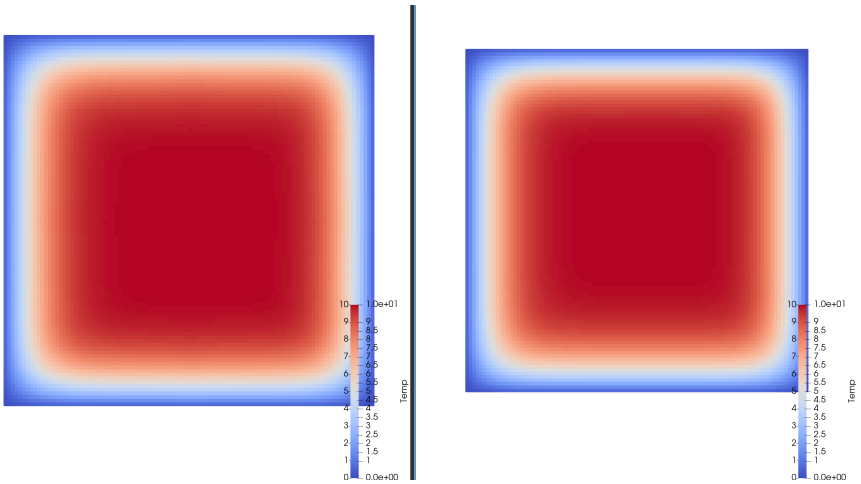

Figure 8: Thermal Diffusivity in Parallel (Left) and Sequential (Right) Case

However, we also ran a Python script (`diff.py`) to check that the actual values at a specific time step for both cases were not far enough. The initial condition set every value in the grid equal to 10, so as time progressed, the values all went to 0, and the gap between the parallel and sequential code was less. However, even when it was at its highest at an earlier time step, say step 100, our script said 54.08% of values were different with a total difference of 1.517 and an average difference of 0.0001517. Therefore, at its worst the parallel case was only off by a second or third of a decimal point. To be specific, this result was from our 100×100 grid case, so there were 1000 grid points and dividing $1.517/1000 = .001517$.

4 Conclusion

This code achieved the goals originally set out for a project because we developed an accurate 2D heat equation solver that ran in parallel and provided near ideal strong and weak scaling. Since we used a combination of both `MPI` and `OpenMP`, our code can be adapted to environments where there are both shared, distributed, and hybrid computers.

We also took care to optimize our code so that during the main time stepper loop, we did not use extra space than the block size, and we do not have any unnecessary loops contributing to our run-time. We used correct OMP directives so the parallel code would compute the correct values and our MPI code runs without communication error in the ghost cell approach.

However, we could still gather more results from here by taking different parallel programming approaches and running our code on different types of machines. We only tested our code on CPUs; however, we could have used

`OpenMP` or `OpenACC` for GPU parallelization. Our code would likely benefit from GPU parallelization when we update the values in each block— where we use `OpenMP` currently. However, this is a communication-heavy algorithm, so MPI would still be the ideal choice for the halo exchange (ghost cell sends and receives).

References

[1] Victor Eijkhout. 2012. Introduction to High Performance Scientific Computing. Lulu.com.

[2] Fabien Dournac. 2015. MPI Parallelization for numerically solving the 2D Heat equation. Dournac.org/info/parallel_heat2d.

[3] Ryan Daileda. 2012. The two dimensional heat equation. Ramanujan.math.trinity.edu/rdaileda/teach/s12/m3357/lectures/lecture_3_6_short.pdf.

[4] Victor Eijkhout. 2022. Parallel Programming in MPI and OpenMP. Theartofhpc.com.