James Hahn
CS1645
High Performance Computing

## Verification

Below are the results of the verification test with source term $S(x, y) = xe^y$ and boundary conditions $T(0, y) = 0$, $T(2, y) = 2e^y$, $T(x, 0) = x$, and $T(x, 1) = xe$. The exact solution is $T = xe^y$. The contour map (Figure 1) can be found below plotted on the domain $0 \leq x \leq 2$ and $0 \leq y \leq 1$ with one threadblock. Additionally, Figure 2 displays the contour map with four threadblocks on a grid size of 400x200. A convergence rate of $10^{-12}$ was used on an iteration-by-iteration basis. All results in this report are from Jacobi algorithm requested in the project description.

| **Grid Size** | **L-∞ Error** |
|---|---|
| 5x5 | $1.0929 \times 10^{-3}$ |
| 11x11 | $1.8886 \times 10^{-4}$ |
| 21x21 | $4.8004 \times 10^{-5}$ |
| 41x41 | $1.2026 \times 10^{-5}$ |
| 51x51 | $7.6982 \times 10^{-6}$ |
| 101x101 | $1.9238 \times 10^{-6}$ |
| 201x201 | $4.7511 \times 10^{-7}$ |

The discretization is verified since the above tests were ran on successively finer grids. The algorithm was also tested on odd-dimension grids, such as 11x5, 21x11, and 41x21. The algorithm still worked and converged. It is important to note these lopsided dimensions will not necessarily work with more than 1 threadblock because the project specifications clarify each dimension is evenly divisible by the number of threadblocks in that dimension. For example, if there are 3 threadblocks in the y-dimension and 2 threadblocks in the x-dimension, the grid dimensions must be divisible by 3 in the y-dimension and divisible by 2 in the x-dimension or the program will exit with an error.

**Figure 1**
Contour Map of verification test with 1 threadblock

**Figure 2**
Contour Map of verification test with 4 threadblocks

## Performance

As grid size increases, so did the time taken to solve Poisson's equation. The grid sizes were pushed to significant and experimentally viable sizes.

All simulations were ran on the University of Pittsburgh's Center for Research Computing (CRC). Times were individually recorded for each algorithm as normal in the previous projects. Only the time spent on solving the equation is monitored; setup and error calculation are not considered in the measurements. Due to the nature of these simulations on a large 500x500 grid, the serial program was only ran once to save time, but the other algorithms were ran five times. Clearly, they all converge with approximately the same solution with similar error.

In order to grab a compute node for the first three simulations, a slurm script was written. To run simulations for OpenACC, the following commands were used:

```
crc-interactive.py -g -t 1 -n 1 -p gtx1080 -c 1
pgc++ -acc -Minfo=accel -ta=tesla:managed -fast poisson4.cpp && ./a.out
```

As such, unified memory was used with OpenACC on a GPU cluster.

| Grid Size | Method | Threads | Cores | Time (seconds) | Error |
|---|---|---|---|---|---|
| 500x500 | Serial | N/A | N/A | 25656.3 | 4.4003e-08 |
| 500x500 | MPI | N/A | 100 | 252.136 | 4.4001e-08 |
| 500x500 | OpenMP | 16 | N/A | 1436.71 | 4.3561e-08 |
| 500x500 | OpenACC | Default | 100 | 1613.91 | 4.4009e-08 |

The tricky part of this project is determining the number of threads and cores to use for each simulation. On the CRC, with my slurm script, MPI only allowed me to specify up to 100 cores; as such, this is all the cores available to me at the time. After that, it said no more resources were able to be allocated. Then, with OpenMP, I could specify more than 16 threads (4x4 layout), but execution time increased significantly for some reason. This counts as basically using all the threads available because even when I tried to increase the number of threads, it allowed me to set a maximum of 25 threads for some reason; so, I basically used all the threads available at the

node. As such, 16 threads worked fine for this purpose. Then, when I simulated Jacobi with OpenACC, the runtime was not too great at first, so I specified num_gangs to be 100. Then, runtime improved slightly, but I figured directly modifying the number of gangs is not worth the overoptimization for a slight increase in runtime if OpenACC can probably do a better job in certain cases. So, for future simulations, I allowed OpenACC to choose the number of gangs. Unfortunately, the runtime was still relatively poor.

As expected, the Serial program took by far the longest amount of time. No explanation is required for this. Then, surprisingly, MPI was by far the lowest execution time. It makes sense that OpenMP was significantly higher since MPI is distributed memory across many cores while OpenMP is one core and many threads in shared memory, requiring critical sections for some steps. The only additional overhead for MPI is communication cost to send the border value of one processor's matrix to bordering processors.

However, the only surprising result I found was the use of OpenACC. For some reason, the runtime was greater than both MPI and OpenMPI, despite the use of threads AND blocks (i.e. processors). I believe part of this is due to my implementation of the algorithm. I even removed the overhead of function calls by moving most of the code to main(), but the runtime barely improved. I suspect the issue is that the while(true) loop isn't parallelized, but then I have three separate clauses for parallel loop pragmas, which probably could be condensed significantly by optimizing my algorithm. Additionally, OpenACC could be the root cause as it might fail to optimize for the correct number of gangs, workers, and vectors. I know for a fact all of my loops are parallelizable since that is what it says when it compiles, but OpenACC perhaps optimized the code poorly.

## Algorithm Presentation

Clearly, the project handout contains a brief, general, vague description of the algorithm and Poisson equation. After further investigation, the Jacobi algorithm was implemented for this project to solve any solution given the source term, domain constraints, and boundary conditions.

As noted in the handout, the second derivative can be used to solve the true values of the solution. It turns out, the following equation, after solving for $T_{i,j}$, can be used to approximate the desired value:

$$(T_{i-1,j} - 2T_{i,j} + T_{i+1,j}) / (\Delta x)^2 + (T_{i,j-1} - 2T_{i,j} + T_{i,j+1}) / (\Delta y)^2 = S_{i,j}$$

Which ends up being:

$$[\ (\Delta y)^2 T_{i-1,j} + (\Delta y)^2 T_{i+1,j} + (\Delta x)^2 T_{i,j-1} + (\Delta x)^2 T_{i,j+1} - (\Delta y)^2 (\Delta x)^2 S_{i,j}\ ] / (2(\Delta y)^2 + 2(\Delta x)^2) = T_{i,j}$$

So, this above equation was used to approximate the value of $T_{i,j}$ on each iteration, based on its neighbors and underlying source term, which was always provided for all coordinates.

An important note is this algorithm does not iterate over the boundaries because they are important boundary conditions to keep the algorithm stable. As such, $T_{i,j}$ is only a concern for all interior points of the grid.

One important challenge in the implementation of this algorithm, since it is utilizes threadblocks, is to ensure each threadblock maintains an MxN matrix containing only that threadblock's data and ensuring some threadblocks iterate over all M values and others iterate over M-1 values. For

example, in a four threadblock scheme, threadblock 0 possesses the overall top and left boundary values of the matrix. Thus, when updating values for the Jacobi algorithm, for loops must iterate from 1…M and 1…N, but threadblock 1, which possesses the top and right boundary values, iterates from 1…M and 0…N-1. Finally, threadblock 2, with the left and bottom boundary values, iterates from 0…M-1 and 1…N. Maintaining the integrity of these boundary conditions across any scheme and any number of threadblocks is imperative or the algorithm is at risk of never converging with poor values, thus producing high error. Nearly all of these implementation concerns were handled by OpenACC since it determined how the loop pragmas took care of the matrix and its indices. Nothing fancy on my end was required, contrasting with the previous two projects.

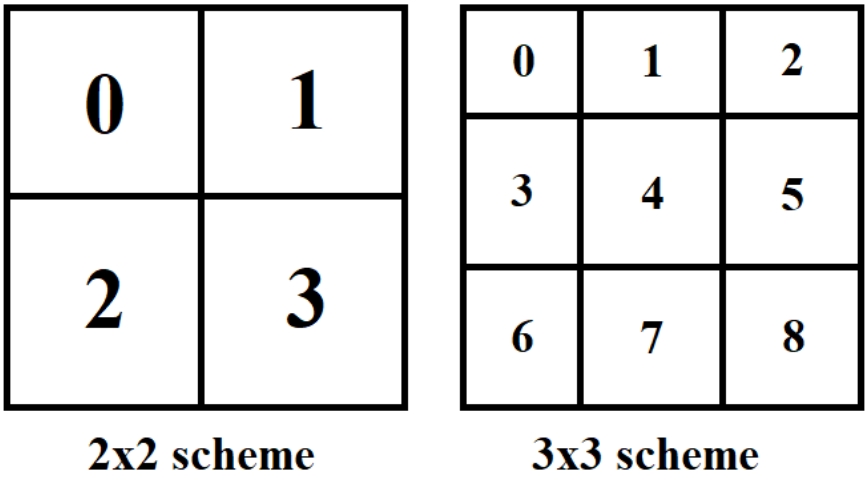


**Figure 4**
Illustration of partitioning scheme with several threadblocks

In the end, to find the error between the 'solved' grid and the 'true' grid, given in the verification test, we calculated the L-∞ norm (maximum difference between 'solved' value and 'true' value in the grid).

Satisfactory results were generated, as the grid both converged and produced useful contour maps related to the underlying solution values.

**What is your conclusion regarding your performance analysis?**

I have two differing thoughts on this question. The first is that OpenACC ***should*** be best medium for solving the Poisson equation. This is because it is on a GPU with much higher number of processors, each of which contain many threads. Additionally, OpenACC is a very technically advanced technology that should optimize my code correctly. However, I guess this is not what happened in reality and maybe resources were overallocated, leading to poor execution time results.

In reality, **MPI is the best** medium to solve this equation. This is not a stretch observation by any means. MPI utilizes distributed memory across many processors. As such, the matrix is truly being solved in parallel. In addition, the only overhead is communication between the processors to share border values, which does not cost much in computation time or memory. It should theoretically be better than OpenMP and it is indeed practically better. I believe MPI

bests OpenACC due to specific specification of the number of cores (100) without having to worry about thread configuration, which may have led to a worse runtime for OpenACC.