**Hotplate, Andrew Stephenson**

**Introduction**

For this assignment, the purpose was to learn how to use OpenMP to parallelize a simple program (uniform heat distribution with the hotplate problem), and then learn to run jobs on the supercomputer.

**Methods**

Initially, I tried to just implement a naïve approach using two dimensional arrays and iteratively updating the arrays and checking to see if they had reached a steady state or not yet. I almost immediately ran into problems with array allocation though (computers don’t like allocating 16MB on the stack), and rather than playing with the settings, I opted to just use arrays on the heap. To help streamline input output operations via locality, I opted to use one single dimension array instead of a two dimensional array, since malloc provides no guarantee as to where on the heap each block will be allocated, but when memory is allocated as one block, it will be contiguous.

The single biggest problem that I ran into is that I forgot to exclude the fixed temperature cells in the middle of the grid from the check for steady state, which resulted in the simulation continuing to run forever. The naïve approach that I took actually worked very well, with some minor tweaking to account for the need to pass array pointers around. Once I fixed the steady state check to exclude the constant-temperature cells, everything went perfectly.

**Creative Contribution**

After the initial results, I tried multiple strategies to improve performance. The first was simply to parallelize everything that could be parallelized, with the exception of the check for a steady state (because it still comes out faster to short circuit the check for steady state rather than parallelizing it and doing the entire thing every time). This caused significant increases in performance, except when running on only a single thread (which makes sense, as the additional overhead of creating threads is encountered without the benefits of being able to work on anything in parallel).

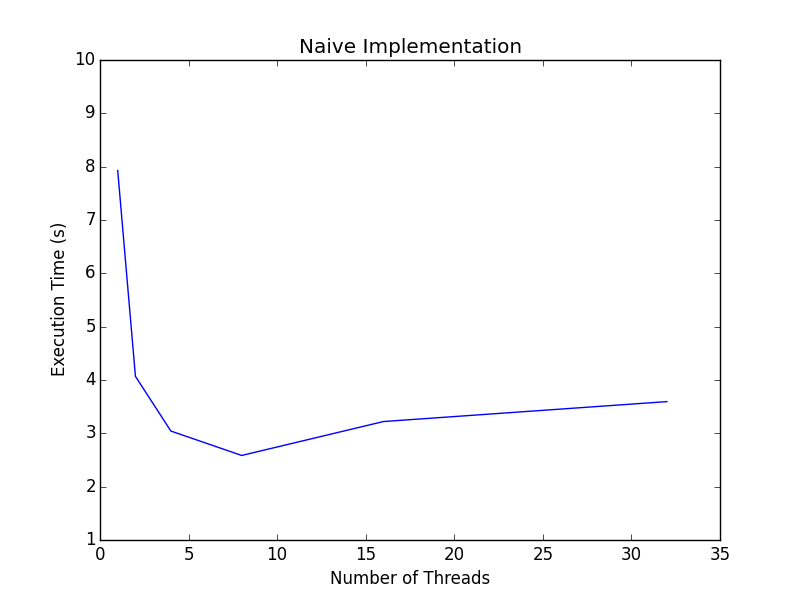
These increases in performance make sense, because they parallelize a number of frequently repeated operations (like the check for steady state, which occurs at each iteration). Given the size of these operations, how easy they are to parallelize, and how often they are repeated, the performance gains are expected.

I also added variables to track which cells have already converged. Once a cell has converged, it remains converged, so checking it every single time is a waste of computational resources. Instead, the function that checks for a steady state can simply start at the first cell that still hasn’t converged at each iteration. This improvement leads to significant performance gains as well.

**Results**

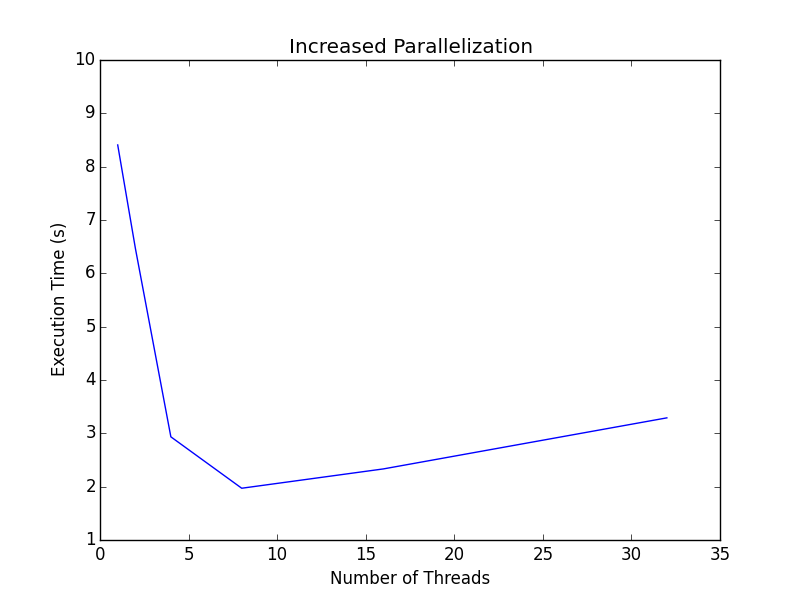
Original Naïve Implementation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Threads | 1 | 2 | 4 | 8 | 16 | 32 |
| Execution Time (s) | 7.92 | 4.06 | 3.04 | 2.58 | 3.22 | 3.55 |

****

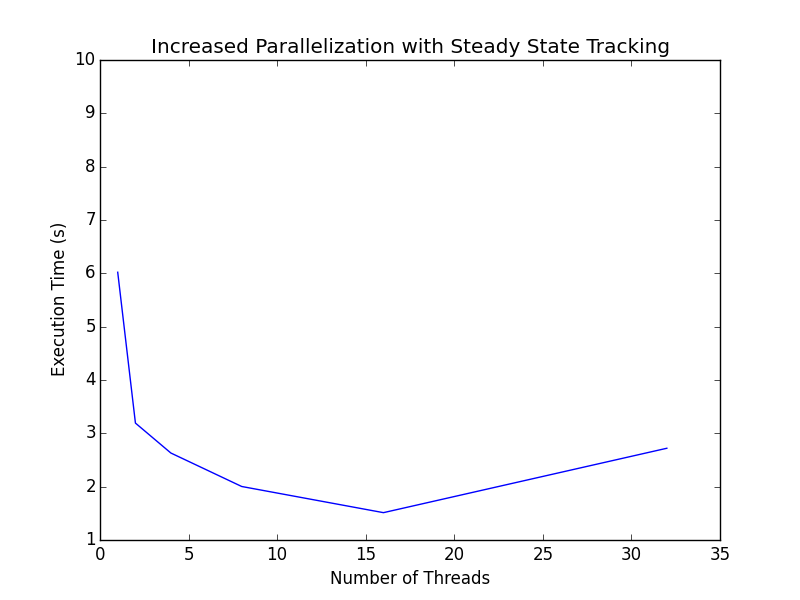
Second Implementation With Increased Parallelization:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Threads | 1 | 2 | 4 | 8 | 16 | 32 |
| Execution Time (s) | 8.41 | 6.46 | 2.93 | 1.97 | 2.33 | 3.29 |



Final Implementation with Increased Parallelization and Tracking State Stability

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Number of Threads | 1 | 2 | 4 | 8 | 16 | 32 |
| Execution Time (s) | 6.02 | 3.19 | 2.63 | 2.00 | 1.51 | 2.72 |



I haven’t included any observations or discussion of the results in this section because I already discussed them in the earlier sections.

**Conclusions**

Parallel programming can lead to significant performance gains when done correctly and on the right kind of problem. If used excessively, or on problems that aren’t really parallelizable, it can actually harm performance. Additionally, parallel programming requires significantly more thought and planning than serial versions of the same program, and if not done correctly can cause problems.