# Parallelizing a Serial Program - Hotplate Temperature Distribution

**Cody Heffner, CS 484**

The requirements for lab 1 were to code a solution to the hotplate problem listed on the course website and execute the solution serially. Once the solution was complete, the student was to parallelize the program using the OpenMP libraries, utilizing 1, 2, 4, 8, and 16 cores.

My solution initialized three arrays of length 4096 x 4096. The first two stored the previous and newly calculated temperature values; the bottom tiles and a few designated tiles in the middle were to be assigned a fixed temperature value of 100, the top tiles and the length of the two sides were to be assigned a value of 0 and fixed, and all other tiles were a variable temperature initialized with a value of 50. The third array stored a 0 or 1 to specify whether the temperature should be fixed; 0 was not fixed, 1 was fixed.

After initialization, my program began the process of distributing the temperature throughout the plate. It looked through each tile one at a time to calculate a new value. The new value was calculated with the following formula:

$$x_{i,j}^{(t)} = (x_{i+1,j}^{(t-1)} + x_{i-1,j}^{(t-1)} + x_{i,j+1}^{(t-1)} + x_{i,j-1}^{(t-1)} + \{4*x_{i,j}^{(t-1)}\}) / 8$$

The program calculated a new value by averaging the temperature of the current tile with the tiles above, below, to the left, and to the right. The current tile was given a higher weight in the average to offset the weight of the tiles on the side; the current tile's temperature was the most influential in determining the new temperature. If a tile was flagged as fixed the program would not perform this calculation. All the new temperatures were stored in an entirely separate array than the old temperatures to prevent tiles below being influenced by the current round's calculations.

When that code block finished, it repeated once more, performing a new set of calculations and storing those back in the first data structure. It then performed a check to examine whether or not the plate was in a "steady state." The steady state was defined as when all tiles meet the following condition:

$$ABS(x_{i,j}^{(t)} - (x_{i+1,j}^{(t)} + x_{i-1,j}^{(t)} + x_{i,j+1}^{(t)} + x_{i,j-1}^{(t)})/4 ) < 0.1$$

If the temperature of each non-fixed tile was determined to satisfy this condition, then the program exited.

Some problems I encountered while programming the serial portion of this assignment was that I initially stored the temperatures in each tile as an integer. This caused accuracy problems. To solve this, I changed my data structures to hold temperatures as doubles. I also had a few minor bugs here or there, but nothing significant enough to remember for this report.

Parallelization of the solution was straightforward. The first approach I took was to put the following line at the beginning of my `calculate_temperatures` function, immediately before my `for` loop that changed the temperatures of each tile:

```
#pragma omp parallel for shared(old, new, fixed)
```

When I tested this solution, it resulted in significantly longer run times than before with a single thread. I determined this was because this caused my threads to be erased each iteration, and with 360 total iterations resulted in a large performance hit (from 10 seconds to 60+ seconds runtime)

with 360 new sets of threads being created over the course of the program. My solution was to remove both the above line of code and the `for` loop to no longer be its own function, but instead implemented directly in the `while` iteration loop. This brought my run times back down to a reasonable duration (17 seconds) with a single thread.

However, in class today we discussed the proper placement of the parallel code to ensure proper creation/destruction of threads. That discussion led me to be believe that changing my implementation did not in fact do what I thought it did. At this point, I am at a loss as to what the problem was in the first place and what my solution changed to allow such faster runtimes.

When I ran the program on the supercomputer, I received times that showed me that parallel processing can in fact speed up a program's execution (figure 1). I improved my runtimes by using two or four processors to calculate the new temperatures. Improvement beyond four processors was countered entirely by the parallel execution being required to destroy and create new threads each iteration. In today's class we discussed what that problem was, and how to solve it by making each thread be in control of each iteration and manually dividing up the for loop based on which thread it is. This solution is something I plan on doing after this report, so I will not be detailing here the differences I see when programming with proper thread management.

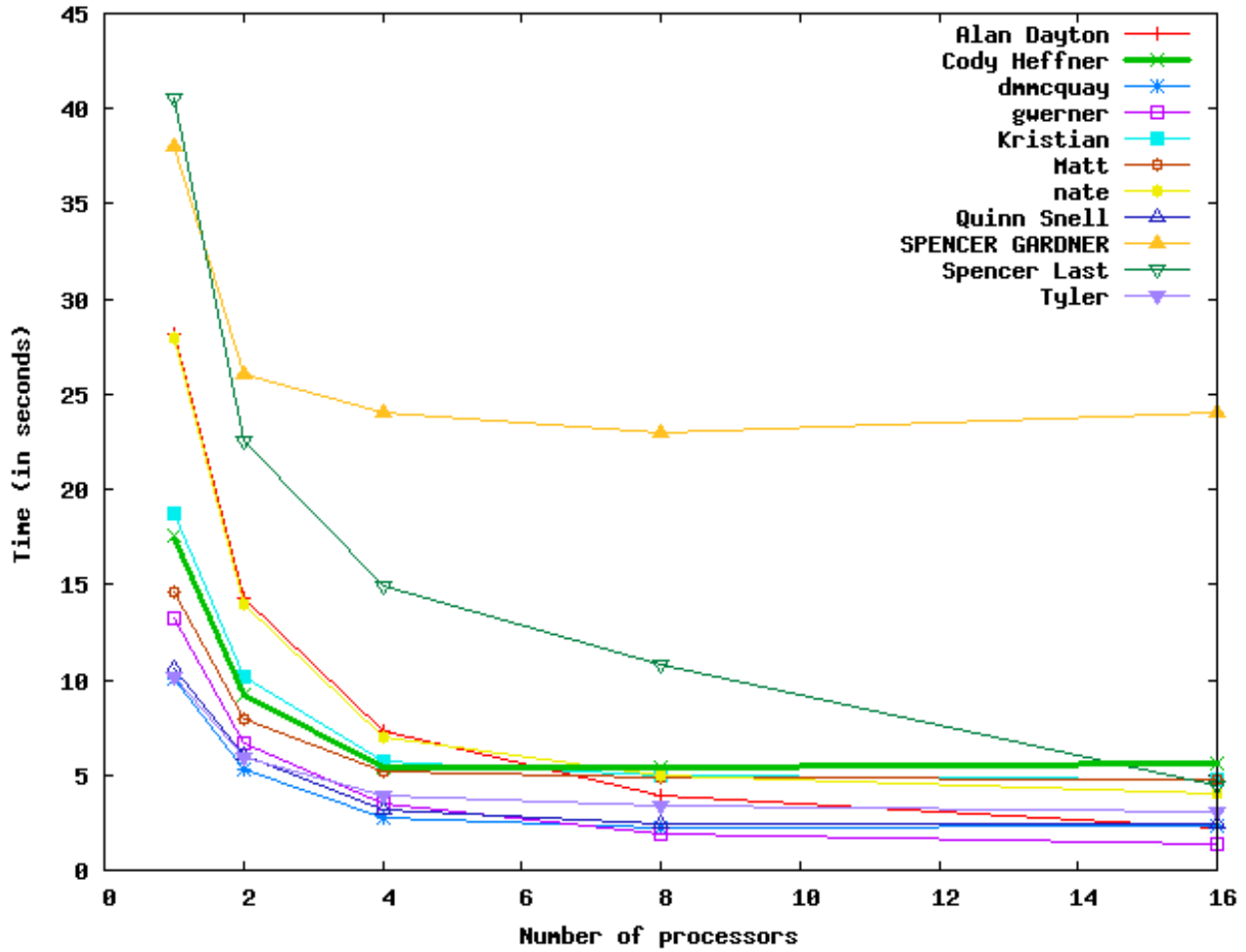


*Figure 1 - The relationship between number of processors and run time. Includes comparison with other members of the class. Highlighted for emphasis.*