Parallelization Report

Max Stritzinger

For this project, I needed to discretize the heat equation to steady state. This program considers a heat source function, S(x,y), boundary conditions, and calculates the heat at given points.

I first implemented this function serially. The code is given as twoD.cpp. The source function used is S(x,y) = -sin(x) – sin(y), however any function may be used. This makes the exact solution, u(x,y) = sin(x) + sin(y). I use the given equation and the boundary conditions to then iteratively calculate predicted values of the exact solution until we reach a specified error threshold between iterations or a maximum number of iterations. In our solution, I used a tolerance of 1E-15.

I then tested my code comparing the output predicted values with the exact solution to ensure that our approximation was correct.

I then attempted to parallelize the code. There are several loops in my serial implementation that are targets for parallelization. The most obvious is where all values of U^{n+1} are approximated. This was a successful parallelization that reduced runtimes for all m,n that were greater than 50.

There are two other candidate loops for parallelization. One is the calculation of the error between U^n and U^{n+1}. I attempted parallelizing first using a parallel for with a critical block for the update. This critical section slowed the code down so much it could not finish 100x100 on two threads. I then tried using an atomic update, however this too slowed down the code compared to the serial version. Our last attempt was creating a mxn error grid where each error value was calculated in parallel and then we looked for the max error serially. This solution only slowed down the execution by ~.2 seconds for a 100x100 grid. It's predicted that this solution would actually be faster for very large values of m,n. (given in twoD_parallel2.cpp).

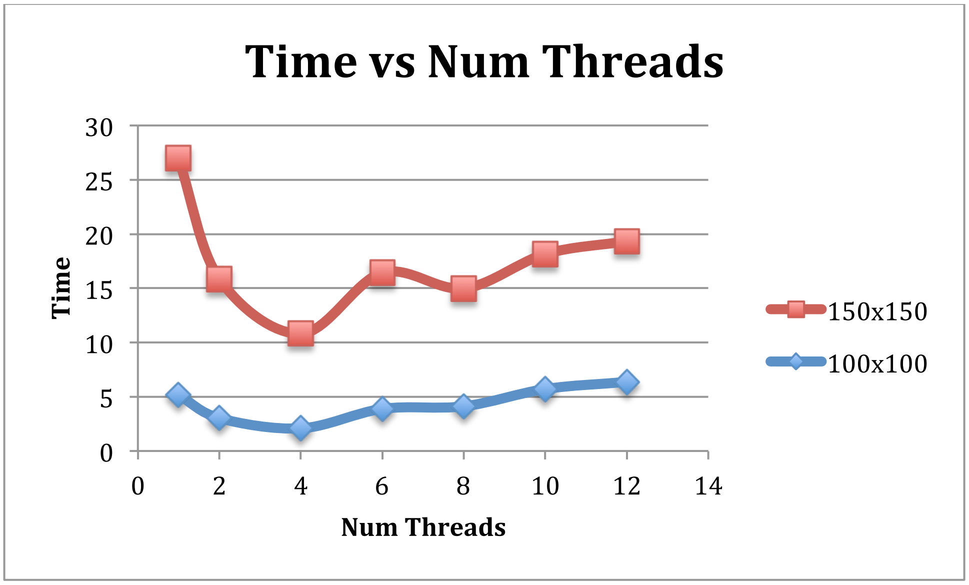
The other loop that could be parallelized is the update loop, however a quick test showed it was fastest to just do this serially.

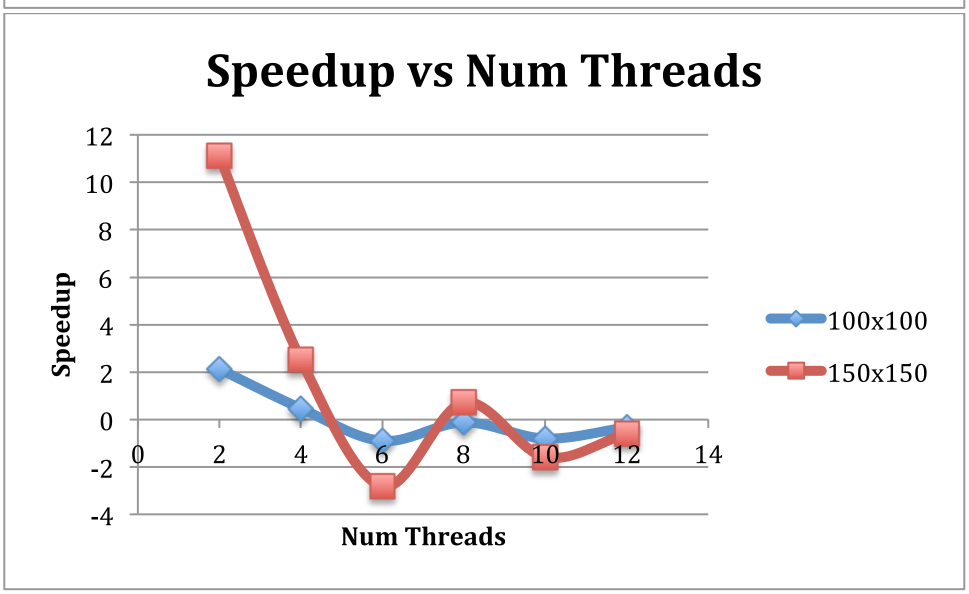
I then ran timed test of the parallel code to see how numbers of threads affected runtime, speedup, and efficiency. These tests were run for 100x100 and 150x150 grids using the icpc compiler on a 4 core processor. The results table is included as results.xlsx.

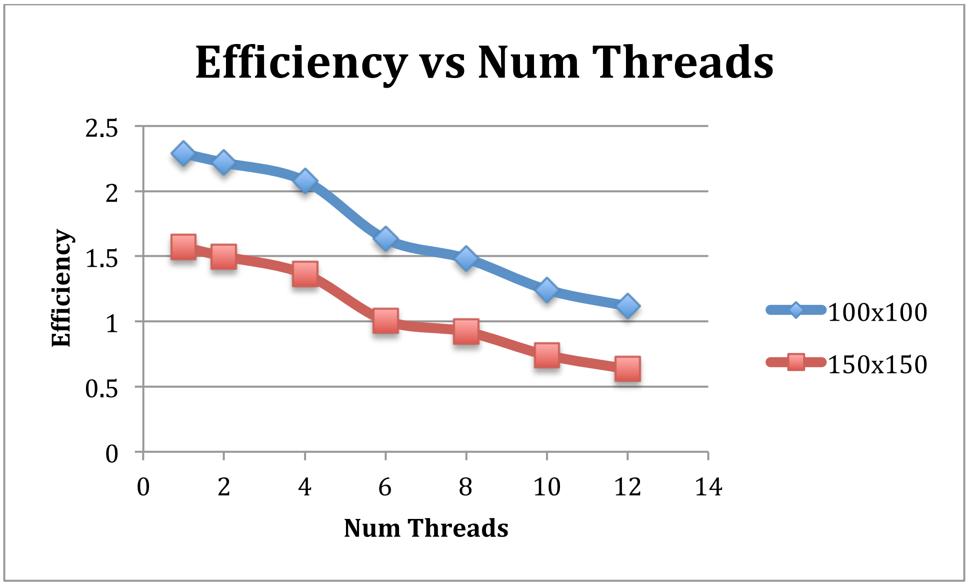
Speedup was calculated as $-\frac{(Time_{n-1} - Time_n)}{(NumThreads_{n-1} - NumThreads_n)}$. For both 100x100 and 150x150 the highest speedup came between going from one thread to 2 threads. Speedup increased up until 4 threads after which additional threads slowed the program down.

Efficiency was calculated as $\log \frac{1}{Time * NumThreads} + c$ where c is a constant used to normalize the values to be positive. The maximum efficiency came from using a single thread and decreased with each additional thread.

Graphs shown below.






Efficiency vs Num Threads
2.5
2
1.5
1
0.5
0
Efficiency
0
2
4
6
8
10
12
14
Num Threads
100x100
150x150