Russell Folk
CS 491/CS 521 Parallel Programming
Homework 3 Report

OpenMP Optimizations

Disclaimer

For these tests, I chose to run on the computer named stonebreaker in the grad lab. It is uncertain if I was the only user when these tests were run so results may vary. Originally I tried running on the koss computer as last time but found troubles identifying improvements.

Changes to the algorithm

The most important change that I made to the algorithm was splitting the check for max diff outside of the generation of the xnew matrix. After many experiments I found that this was key to maximizing performance. By splitting these two operations I was able to see huge performance gains. The reason for this is that I am able to split the threads effectively to generate the matrices, also seen in the reassignment of xold from xnew. After this I am able to have each thread check it's diff then afterwards assign the maximum diff from all the threads utilizing OpenMP's critical section. This too offered great speedups over the original operation.

Tests run

For the final experiments I ran a series of tests and compared to the original file. To examine the differences of OpenMP and OpenMP + compiler optimizations I ran 2 sets of tests. The first is my modified code with 1, 2, 4, 8, and 16 threads. The second is the same code optimized by the compiler with -O3 run with the same numbers of threads.

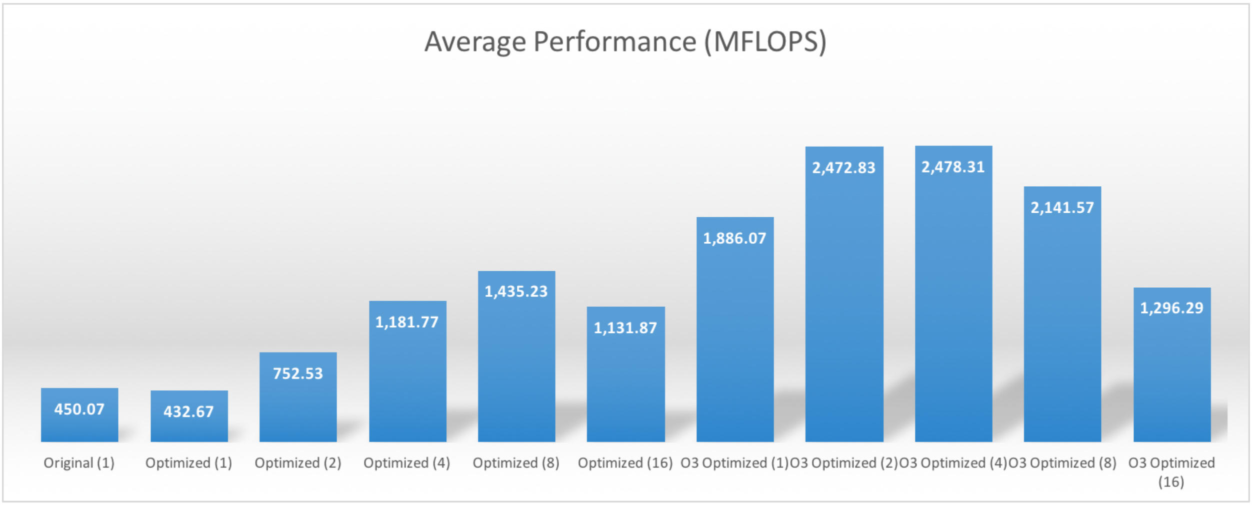

FIGURE 1: AVERAGE PERFORMANCE MEASURED IN MFLOPS
THIS GRAPH SHOWS THE THREE TYPES OF EXPERIMENTS RUN WITH THE THREADS UTILIZED IN PARENTHESIS.

Results

Obviously, as seen in Figure 1, as soon as we apply any multi-threading capabilities we see improvements to our performance yields; this is completely expected. Again, quite obviously, the compiler optimizations trump the plain performance of the rewritten algorithm utilizing OpenMP. Perhaps the most surprising finding though was the difference in threads utilized for best performance when comparing the raw compilation and the optimized compilation. If we examine Figure 2 we are able to see that for the non-compiler-optimized code the optimal number of threads is 8 with 4 and 16 performing similarly, with 4 being in the lead. This makes sense as the machine that I ran on was limited in cores and eventually we reach a point of saturation and additional threads cause additional overhead without generating increased performance. However, when we look at the compiler-optimized code we see that the best performance is at 4 and 2 threads with the difference between the two being negligible. Comparatively, 8 threads is nearly 100% less performance improvement than 2 and 4 threads. I am unsure exactly why this is but I am thinking the compiler-optimization is either matching the cores or the compiler is optimized to fewer cores by defaults. More research is needed on this front.

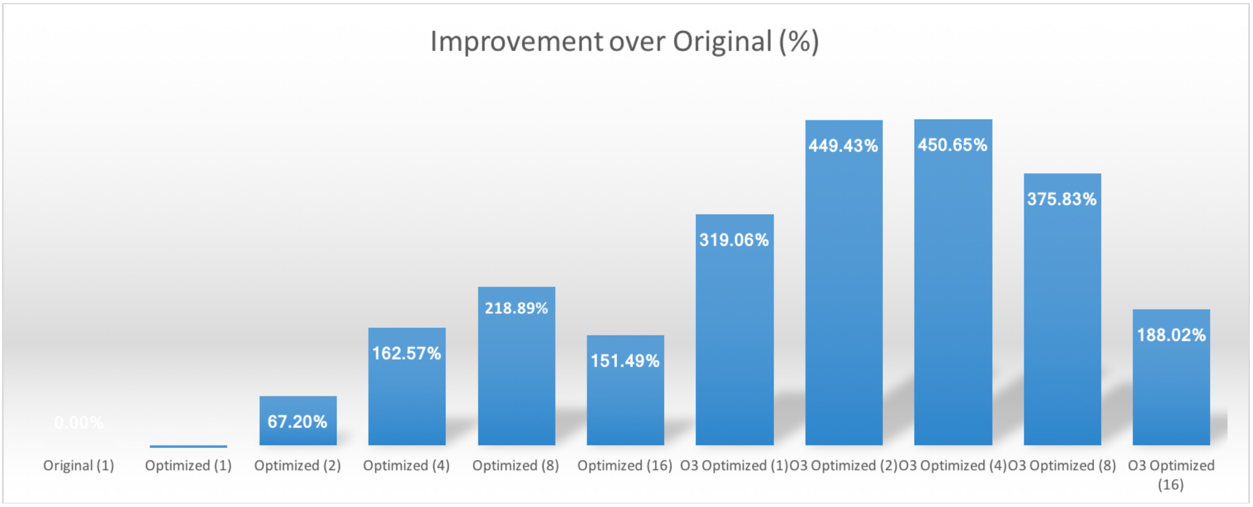

FIGURE 2: AVERAGE PERFORMANCE IMPROVEMENT
THIS GRAPH SHOWS THE PERCENTAGE IMPROVEMENT OF THREE TYPES OF EXPERIMENTS RUN WITH THE THREADS UTILIZED IN PARENTHESIS. EACH THREAD IS COMPARED TO THE ORIGINAL IMPLEMENTATION AND ONLY THE OPTIMIZED CODE RUNNING WITH 1 THREAD PERFORMS POORLY.