Assignment 5

David Warren
University of Alabama
dlwarren2@crimson.ua.edu
CWID: 11703777

I. Introduction

This report reflects the observations and analyses OpenMP parallel implementations of matrix multiplication and modeling of heat distribution. In Part I, we explore parallelization via two methods: parallelizing the outer loop, and then the middle loop. These each give different speed up values. Then we explore parallelization of modeling heat distribution. This is shown to be very scalable, just like matrix multiplication.

II. Ease of Use

A. *CPU and Compiler Information*

CPU used was Intel Xeon Gold 4180 CPUs on normal nodes to gather performance metrics. Compiler used was gcc (GCC) 7.3.0. Optimization flag was set to default -O0 least optimized for parts I-III. During Part II, testing 3dheat on my personal machine was on an AMD Ryzen 9 3900X 12-core Processor, 3.8GHz CPU, gcc (GCC) 7.5.0. Optimization flag was set to -O0 least optimized as well.

B. *Using the Makefile*

- "Make" to compile *parllelMM_Outer.c*, *parallelMM_Middle.c*, *3dheat.c*, *3dheatParallel.c*.
- "Make clean" to remove binary and executable files
- "Make submit" to submit mmOuter.job, mmInner.job, 3dHeat.job, 3dheatParallel.job.

III. Part I Observations

In part I, both parallelized versions of the outer and middle loop were tested, with the times averaged over 5 runs. The parallelization of the middle outperformed the parallelization of the outer loop as you can see below in the diagram. Each optimization was tested with 1, 4, 8, 16, and 32 threads.

Figure (1) Execution time for thread counts over both methods

exec time(s)/#threads	1	4	8	16	32
Outer	47.134993	16.891299	9.955691	4.999199	2.494766
Middle	41.731975	11.237745	5.823337	2.959071	1.532257

As you can see, middle parallelization consistently outperformed outer parallelization. Below we will compare the speedups of each method, outer speedup values, middle speedup values, and then out of interest, middle compared to outer.

Figure (2) Speedup values of outer, middle, and middle compared to outer

speedup (%) / #threads	1	4	8	16	32
middle to outer	1.1294695	1.50308616	1.70961959	1.68944882	1.6281642
outer	1	2.79048953	4.7344773	9.42850905	18.8935527
middle	1	3.71355419	7.1663335	14.1030665	27.2356237

The middle parallelization pulls way ahead with 32 threads with a 27.23x speed up. This is 1.628x faster than the outer parallelization of 32 threads.

IV. Part II Observations

In Part II, the sequential code was modified to use a 3d array, alternating between two levels for each iteration, up to 5000 iterations for size NxN, N=1000. It then displays N/8xN/8 points of the heat space to give an indication of how the heat spreads. Below are the execution times for the sequential algorithm in Part II on my personal machine and on the PantaRhei cluster. There was a small difference in my machine running faster the sequential version, but not too noticeable of a difference.

Figure (3) Personal machine

exec time(s) / #threads	1 (seq)
3dheat	56.352817

Figure (4) PantaRhei Cluster

exec time(s) / #threads	1 (seq)
3dheat	61.108625

V. Part III Observations

In Part III, code from Part II was modified to parallelize the sequential algorithm. The sequential code was included, but a parallelized version was added in which the initializations of the heat space are parallelized (not necessary for speedup of algorithm but I added it to decrease execution time on the

cluster), and then the base algorithm has each iteration launch in parallel the calculations to compute the means of surrounding points, and then in parallel updates the opposite level of the 3-dimensional array for the heat space. Every point of the graph was compared to the sequential version to find the maximum difference between points. Each run had a max difference of 0.000000 indicating that the sequential and the parallel versions came to the same conclusion. Like in Part II, Part III prints N/8xN/8 points from the heat space, but then gives a speedup from the sequential version. Below you may see the sequential execution time compared to the parallelized execution time, and the speedup metric below that. I ran the sequential algorithm each time before running the parallel to give a closer estimate for speedup, as each sequential run happens during the same job and under the same load for the cluster (or at least closer to the same).

Figure (5) Sequential and Parallelized heat space algorithm execution time in seconds

exec time(s)/#threads	1	4	8	16	32
sequential	63.680373	65.690996	63.174745	63.662311	63.329978
parallel	71.034636	16.404268	8.145428	4.204559	2.375604

Figure (6) Speedup metric of parallel algorithm over sequential (%)

speedup(%)/#threads	1	4	8	16	32
parallel over sequential	0.896469	4.004506	7.755853	15.141258	26.658477

Like Part I, Part III can be shown to be very scalable, with similar metrics to the middle parallelization from Part I. Like Part I as well, we see that scalability starts to level away from perfectly scalable.

VI: Source Code And Results

Source code is attached in tar file. Sample outputs are included in the output/console folder in each folder for the respective parts (part1 & part2-3). The error logs for the sample jobs can be found in output/error in the same folders. Overall, Part I showed both methods to be scalable, with the middle parallelization to be more scalable. Part III showed the parallelization of the heat space problem to be very scalable as well. All iterations finished with the same results as the sequential, as we can see by the findMaxDifference() function in parallelDistribution().