CS 5220 Project 3 Preliminary Report

Team 15 - Xiang Long (XL483), Yalcin Ozhabes (AO294), Bryce Evans (BAE43)

November 10, 2015

1 Introduction

Our task is to profile and optimize an implementation of the all-pairs-shortest-paths algorithm that computes its result through repeated squaring of the minimum paths distance matrix. The parallelization of the implementation is done by OpenMP, and our task is to eventually produce another version using MPI instead. We have decided that for this report, we will profile and tune the current OpenMP version of the code, leaving the MPI implementation to be completed in the rest of the project.

2 VTune Profiling

We first profiled the given code using VTune Amplifier to discover hotspots in the code. Due to license limitations we were only able to profile the code on the Totient head node, but we believe similar results would have been obtained if we were able to perform profiling on the compute nodes. The relevant sections of the report is shown in Table 1.

Function	CPU Time	CPU Time:Idle	CPU Time:Poor	CPU Time:Ideal
square	30.320s	0s	0.121s	30.199s
gen_graph	0.052s	0.010s	0.042s	0s
fletcher16	0.030s	0s	0.030s	0s
__intel_ssse3_memcpy	0.020s	0s	0.020s	0s
genrand	0.019s	0.010s	0.009s	0s
infinitize	0.009s	0s	0.009s	0s

Table 1: VTune Report of Original Code

The results are for the default 2000×2000 matrix, and we have omitted columns that are all zeros and rows that are functions not part of path.c. It is clear that the vast majority of the time is taken by the square function, and that is where we should concentrate our optimization efforts.

3 Scaling Studies of Original Code

We also performed strong and weak scaling studies on the original code, and the performance varying with number of threads is shown in graphs of figures 1 and 2 respectively. Since the original code was parallelized using OpenMP, we expect some performance gains if more worker threads are available.

In the strong scaling graph we can see that performance does improve if more worker threads are created, although there are diminishing returns and improvement is not significant after around 12 threads.

In the weak scaling study, we keep the workload of each thread constant as we increase the number of threads. This was done by starting with a 1000×1000 matrix initially, and multiplying the dimensions by the square root of the number of threads. Here we find that the time taken did not stay constant, which means the parallelization is optimal. With tuning and a better parallelized implementation we can hope to improve on this.

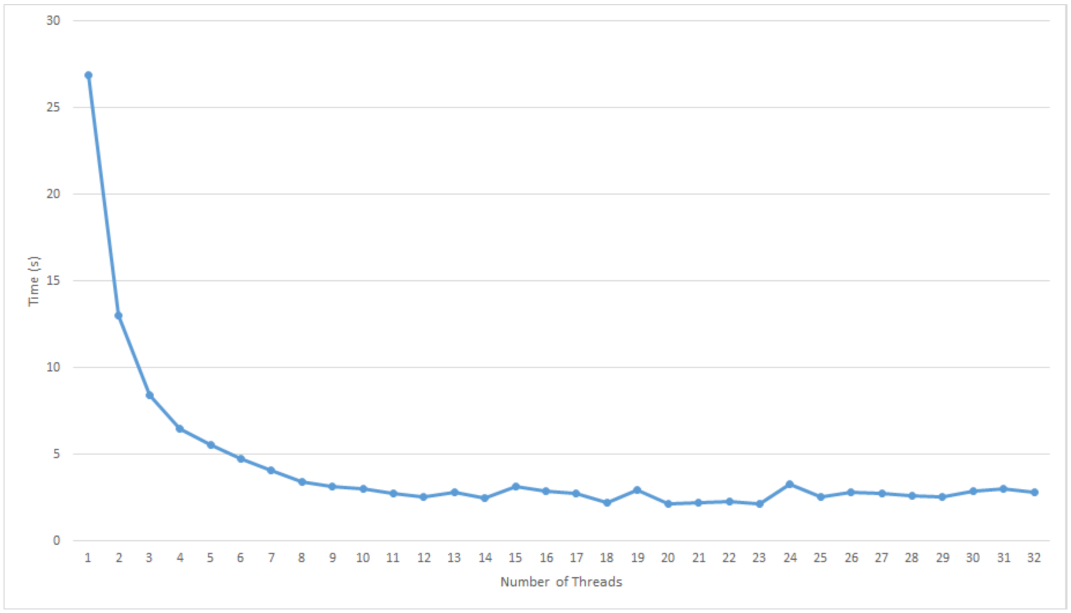

Figure 1: Time taken for original algorithm on a fixed 2000×2000 matrix varying with number of threads

4 Tuning

5 Strategy for Implementation with MPI

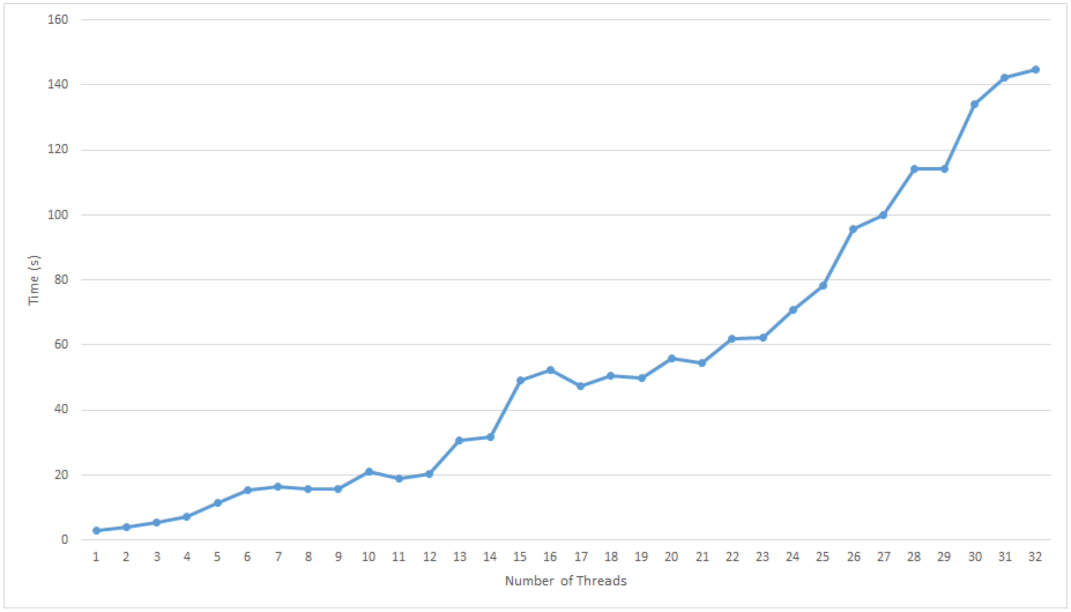

Figure 2: Time taken for original algorithm on varying with number of threads with workload of each thread held constant