CS 5220 Project 3 – Team 6 Mid Report

Taejoon Song, Junteng Jia, Joshua Cohen
{ts693, jj585, jbc264}@cornell.edu

November 12, 2015

1 INTRODUCTION

The Floyd–Warshall algorithm, in computer science, is an alogorithm for finding the minimum patahs in a weighted graph. (i.e. All-pairs shortest path problem)
Our goal in this assignment includes:

1. Profiling: To find the bottleneck of the code.
2. Paralleization : To make it paralle by using MPI.
3. Optimization (Tuning) : Finally, tune it aggressively to get highest performance.

The rest of the report is organized as follows. In Section 2, we introduce a baseline timing result from initial copy, and show our profiling result. Section **??** discusses our approach for parallelization-related work and Section 3 discusses vectorization. Our evaluation result will be shown in Section **??**. Finally, Section 4 suggestes what should be done more after peer reviews.

2 TIMING

2.1 PROFILING

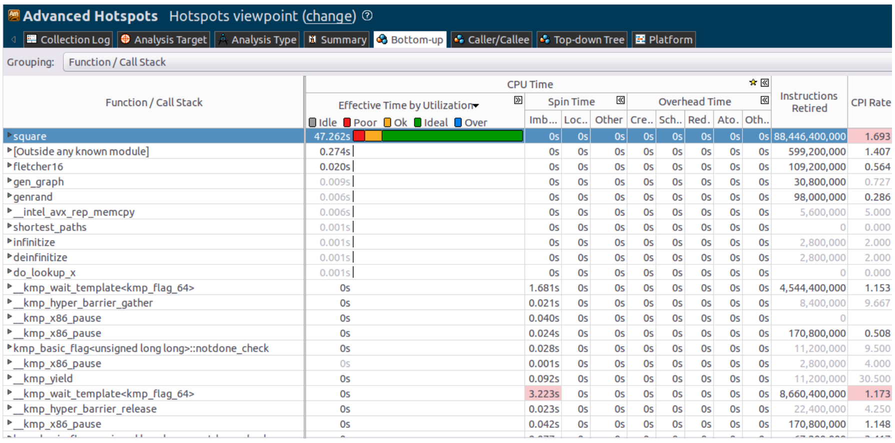
In order to find the bottlenecks of path, we first profiled our code using Intel's VTune Amplifier (It was broken in cluster, so we used it on local machie), as shown in Figure 1.

2.2 INITIAL PROFILE RESULT

Initial profile result can be found at Figure 2, and more detail in Figure 3.
We found that "square" function is the main bottleneck and do the most critical steps for this program, so this point is where we started to tune the code.

```
amplxe-cl -collect advanced-hotspots ./path
amplxe-cl -report hotspots -source-object function=<NAME>
```

Figure 1: VTune Amplifier Command



Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Platform

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Spin Time: Imb...	Spin Time: Loc..	Spin Time: Other	Overhead Time: Cre..	Overhead Time: Sch..	Overhead Time: Red.	Overhead Time: Ato.	Overhead Time: Oth..	Instructions Retired	CPI Rate
square	47.262s	0s	0s	0s	0s	0s	0s	0s	0s	88,446,400,000	1.693
[Outside any known module]	0.274s	0s	0s	0s	0s	0s	0s	0s	0s	599,200,000	1.407
fletcher16	0.020s	0s	0s	0s	0s	0s	0s	0s	0s	109,200,000	0.564
gen_graph	0.009s	0s	0s	0s	0s	0s	0s	0s	0s	30,800,000	0.727
genrand	0.006s	0s	0s	0s	0s	0s	0s	0s	0s	98,000,000	0.286
__intel_avx_rep_memcpy	0.006s	0s	0s	0s	0s	0s	0s	0s	0s	5,600,000	5.000
shortest_paths	0.001s	0s	0s	0s	0s	0s	0s	0s	0s	0	0.000
infinitize	0.001s	0s	0s	0s	0s	0s	0s	0s	0s	2,800,000	2.000
deinfinitize	0.001s	0s	0s	0s	0s	0s	0s	0s	0s	2,800,000	2.000
do_lookup_x	0.001s	0s	0s	0s	0s	0s	0s	0s	0s	0	0.000
__kmp_wait_template<kmp_flag_64>	0s	1.681s	0s	0s	0s	0s	0s	0s	0s	4,544,400,000	1.153
__kmp_hyper_barrier_gather	0s	0.021s	0s	0s	0s	0s	0s	0s	0s	8,400,000	9.667
__kmp_x86_pause	0s	0.040s	0s	0s	0s	0s	0s	0s	0s	0	
__kmp_x86_pause	0s	0.024s	0s	0s	0s	0s	0s	0s	0s	170,800,000	0.508
kmp_basic_flag<unsigned long long>::notdone_check	0s	0.028s	0s	0s	0s	0s	0s	0s	0s	11,200,000	9.500
__kmp_x86_pause	0s	0.001s	0s	0s	0s	0s	0s	0s	0s	2,800,000	4.000
__kmp_yield	0s	0.092s	0s	0s	0s	0s	0s	0s	0s	11,200,000	30.500
__kmp_wait_template<kmp_flag_64>	0s	3.223s	0s	0s	0s	0s	0s	0s	0s	8,660,400,000	1.173
__kmp_hyper_barrier_release	0s	0.023s	0s	0s	0s	0s	0s	0s	0s	22,400,000	4.250
__kmp_x86_pause	0s	0.042s	0s	0s	0s	0s	0s	0s	0s	170,800,000	1.148

Figure 2: Initial Profile Analysis

2.3 Initial Timing Result

Initial timing result is shown in Figure 4. As we can see the program is running with 8 threads using OpenMP and it takes 11.0818 seconds for 2000 nodes.

3 Vectorization

In order to vectorize properly vectorize our code, we looked at the output of `ipo_out.optrpt` after compiling with flags `-qopt-report=5 -qopt-report-phase=vec`. We first were able to vectorize our call to `square` within `shortest_paths` within `path.c` by explicitly precomputing the transpose of `l` during each call to `square`, and then replacing the assignment of `lik` directly from `l`, as

```
int lik = l[k*n+i]
```

to an assignment instead from the transpose, as

```
int lik = l_T[i*n+k]
```

We also attempted to solve the issue of unaligned memory access from within `l` and `l_T` by replacing calls to `malloc` with `_mm_malloc` (and, correspondingly, calls to `free` with calls to `_mm_free`), using a byte alignment of 32 since we're compiling with AVX2 (using the flag `-xcore-avx2`). This solved some of the issues with unaligned access, according to the vectorization report, but there are still cases with unaligned access reported.

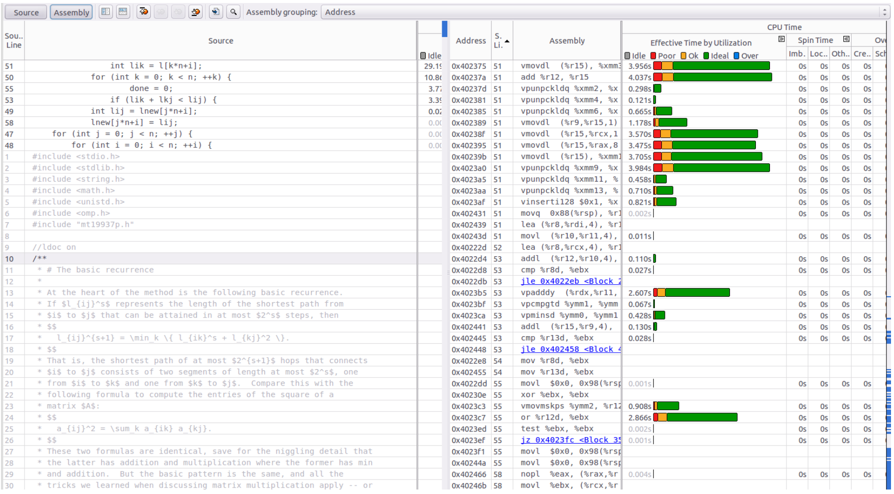


Figure 3: Initial Assembly Result

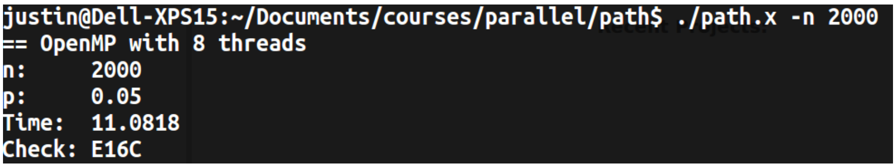


Figure 4: Initial Timing Result

4 Future Work

- **Vectorization.** According to the vectorization report, we fixed most of the issues with loops not being properly vectorized. We still are having some issues with unaligned memory accesses, so we need to make sure that we're using the correct byte alignment and figure out how to properly make our indexing and memory accesses properly aligned. However, we are planning on focusing mainly on our MPI implementation.

- **Work Minimization.**

- **Compile Time Sizing.**

- **Blocking.**