
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 algorithm for finding the minimum paths 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. Parallelization : To make it parallel 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 4 discusses our approach for parallelization-related work and Section 3 discusses vectorization. Our evaluation result will be shown in Section 5. Finally, Section 6 suggests 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 machine), as shown in Figure 1.

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

Figure 1: VTune Amplifier Command

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.

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													Instructions Retired	CPI Rate
	Effective Time by Utilization▼					Spin Time			Overhead Time						
	Idle	Poor	Ok	Ideal	Over	Imb...	Loc...	Other	Cre...	Sch...	Red...	Ata...	Oth...		
*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
*infinite	0.001s					0s	0s	0s	0s	0s	0s	0s	0s	2,800,000	2.000
*deinfinite	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	0.000
*_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

Source Line	Source	Address	Assembly	Effective Time by Utilization	Spin Time	Overhead Time
51	int lik = l[k*n+1];	0x402375	vmovd1 (r15), %xmm0	3.956s	0s	0s
52	for (int k = 0; k < n; ++k) {	0x40237a	add r12, r15	4.037s	0s	0s
53	done = 0;	0x40237d	vpunpckldq %xmm2, %x	0.298s	0s	0s
54	if (lik + lik < lik) {	0x402381	vpunpckldq %xmm4, %x	0.121s	0s	0s
55	int lij = lnew[j*n+1];	0x402385	vpunpckldq %xmm6, %x	0.665s	0s	0s
56	lnew[j*n+1] = lij;	0x402389	vmovd1 (r9, r15, 1)	1.178s	0s	0s
57	for (int j = 0; j < n; ++j) {	0x40238f	vmovd1 (r15, r15, 1)	3.370s	0s	0s
58	for (int i = 0; i < n; ++i) {	0x402395	vmovd1 (r15, r15, 8)	3.475s	0s	0s
59	vmovd1 (r15), %xmm1	0x40239b	vmovd1 (r15), %xmm1	3.705s	0s	0s
60	vpunpckldq %xmm9, %x	0x4023a0	vpunpckldq %xmm9, %x	3.984s	0s	0s
61	vpunpckldq %xmm1, %x	0x4023a5	vpunpckldq %xmm1, %x	0.458s	0s	0s
62	vpunpckldq %xmm3, %x	0x4023aa	vpunpckldq %xmm3, %x	0.710s	0s	0s
63	vinseri128 \$0x1, %x	0x4023af	vinseri128 \$0x1, %x	0.821s	0s	0s
64	movq %xmm0(rsp), %r1	0x402431	movq %xmm0(rsp), %r1	0.002s	0s	0s
65	lea (r8, r1, 4), %r1	0x402439	lea (r8, r1, 4), %r1	0.011s	0s	0s
66	movl (r10, r11, 4), %r1	0x40243d	movl (r10, r11, 4), %r1	0.011s	0s	0s
67	lea (r8, r1, 4), %r1	0x402442	lea (r8, r1, 4), %r1	0.110s	0s	0s
68	addl (r12, r10, 4), %r1	0x402444	addl (r12, r10, 4), %r1	0.027s	0s	0s
69	cmpl %r8d, %ebx	0x4024d8	cmpl %r8d, %ebx	0.027s	0s	0s
70	jle 0x4024e0, @.lock_1	0x4024db	jle 0x4024e0, @.lock_1	2.607s	0s	0s
71	vpaddd (r15, r11, 1)	0x4024e0	vpaddd (r15, r11, 1)	0.067s	0s	0s
72	vpcomgtq %ymm1, %ymm	0x4024e3	vpcomgtq %ymm1, %ymm	0.428s	0s	0s
73	vpinsrd %ymm0, %ymm1	0x4024e6	vpinsrd %ymm0, %ymm1	0.130s	0s	0s
74	addl (r15, r9, 4), %r1	0x4024e9	addl (r15, r9, 4), %r1	0.028s	0s	0s
75	cmpl %r13d, %ebx	0x4024f5	cmpl %r13d, %ebx	0.028s	0s	0s
76	jle 0x4024f8, @.lock_1	0x4024f5	jle 0x4024f8, @.lock_1	0.001s	0s	0s
77	mov %r8d, %ebx	0x4024e8	mov %r8d, %ebx	0.000s	0s	0s
78	mov %r12d, %ebx	0x4024e5	mov %r12d, %ebx	2.866s	0s	0s
79	xor %ebx, %ebx	0x40230e	xor %ebx, %ebx	0.002s	0s	0s
80	or %r12d, %ebx	0x4023c7	or %r12d, %ebx	0.001s	0s	0s
81	test %ebx, %ebx	0x4023ed	test %ebx, %ebx	0.001s	0s	0s
82	jz 0x4023f0, @.lock_1	0x4023ef	jz 0x4023f0, @.lock_1	0.001s	0s	0s
83	movl %xmm0, %xmm0(rsp)	0x4023f1	movl %xmm0, %xmm0(rsp)	0.004s	0s	0s
84	movl %xmm0, %xmm0(rsp)	0x40244a	movl %xmm0, %xmm0(rsp)	0.004s	0s	0s
85	nopl %eax, (%rax, %r	0x402466	nopl %eax, (%rax, %r	0.004s	0s	0s
86	movl %ebx, (%rcx, %r	0x40246b	movl %ebx, (%rcx, %r	0.004s	0s	0s

Figure 3: Initial Assembly Result

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 vertices.

```

justin@Dell-XPS15:~/Documents/courses/parallel/path$ ./path.x -n 2000
== OpenMP with 8 threads
n: 2000
p: 0.05
Time: 11.0818
Check: E16C

```

Figure 4: Initial Timing Result

3 VECTORIZATION

In order to vectorize properly vectorize our code, we looked at the output of `ipo_out.optprtpt` 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.

4 PARALLELIZATION

Parallel..

5 EVALUATION

5.1 TIMING AFTER VECTORIZATION

Grouping: Function / Call Stack		CPU Time										Instructions Retired		CPU Rate		Module		Function (Full)	
Function / Call Stack		Effective Time by Utilization	SpinTime	Loc..	Other	Cre..	Sch..	Red..	Atto.	Oth..									
*square		9.097s										28,224,000,000	0.998	1.108	path.x			square	
*[Outside any known module]		0.055s										86,800,000	1.484	0.836				[Outside any known module]	
*fletcher16		0.020s										109,200,000	0.564	1.100	path.x			fletcher16	
*shortest_paths		0.016s										42,000,000	1.600	1.500	path.x			shortest_paths	
*_intel_awk_rep_memcpy		0.013s										3,400,000	7.000	1.077	path.x			_intel_awk_rep_memcpy	
*gen_graph		0.010s										36,400,000	0.769	1.000	path.x			gen_graph	
*genrand		0.005s										89,600,000	0.312	2.000	path.x			genrand	
*initialize		0.003s										2,800,000	3.000	1.000	path.x			initialize	
*deinitialize		0.002s										2,800,000	2.000	1.000	path.x			deinitialize	
*rml:internal:Backend:indexedbins:reset		0.001s										0	0.000	0.000	libomp5.so			rml:internal:Backend:indexedbins:reset	
_kmp_wait_template:kmp_flag_64		0s	0.052s									117,600,000	1.238	1.000	libomp5.so			_kmp_wait_template:kmp_flag_64*	
*_kmp_hyper_barrier_gather		0s										2,800,000	0.000	0.000	libomp5.so			_kmp_hyper_barrier_gather	
*[import thunk_kmpc_for_static_fix]		0s										0	0.000	0.000	path.x			[import thunk_kmpc_for_static_fix]	
*func@ox00d8		0s										2,800,000	0.000	0.000	libtntestify_collector.so			func@ox00d8	
*_kmp_yield		0s	0.006s									2,800,000	12.000	2.000	libomp5.so			_kmp_yield	
_kmp_wait_template:kmp_flag_64		0s	0.346s									898,800,000	1.209	1.125	libomp5.so			_kmp_wait_template:kmp_flag_64*	
*_kmp_hyper_barrier_release		0s	0.004s									0	0.000	1.000	libomp5.so			_kmp_hyper_barrier_release	
*_kmp_x86_pause		0s	0.003s									11,200,000	1.000	1.333	libomp5.so			_kmp_x86_pause	
*_kmp_x86_pause		0s	0.006s									0	0.000	1.000	libomp5.so			_kmp_x86_pause	
*kmp_basic_flag:unsigned long long:cnoddone_check		0s	0.010s									2,800,000	9.000	0.900	libomp5.so			kmp_basic_flag:unsigned long long:cnoddone_check	

Figure 5: Profile Analysis After Vectorization

As show in the picture, for the same problem size, the function square is more than 5 times faster than it was before, when we are running it with in Vtune, where one core is used to trace other threads. In fact, running it with command line and time it with build in function `omp_get_time()` shows the saving gives us a factor of more than 9 times faster.

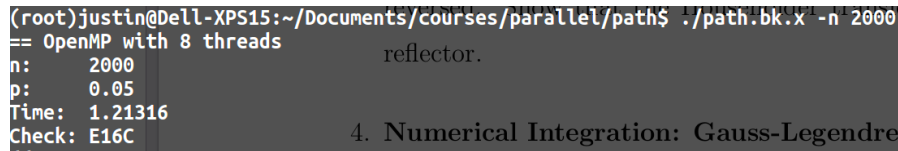


Figure 6: Timing Result After Vectorization

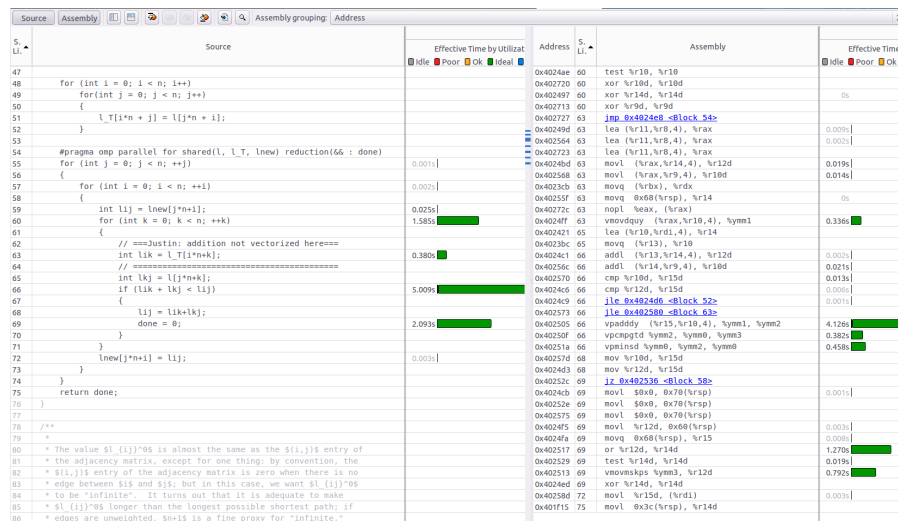


Figure 7: Assembly Analysis After Vectorization

The assembly analysis is more informative as it tells how well our functions have been vectorized as well as the timing. From the assembly analysis after vectorization, we can tell that the computations in the inner has been vectorized. Which is where the savings come from.

5.2 DATA-TYPE OPTIMIZATION

As we went through tests, we found that the maximum distance between two vertices is always below 20, even when we are calculating a 10000 by 10000 graph. Then it naturally follows that, we don't necessarily need a 4-byte `int` to store the distance information.

To fit more data into register so that we can carry out more operation per cycle, we used a prototype called `ddt`, which can be any data type such as `long`, `short`, and `char`. In fact, this gives us 30% saving when we are using `char` to store the distance between two vertices.

Notice in Figure 8, we printed out a variable called `ddt_upper_range` which tells us the largest distance we can have, in order to use one certain data-type. In the case with `char`, the largest distance is 127, which is proved to be more than enough by our tests.

5.3 MIDTERM PROFILE RESULT

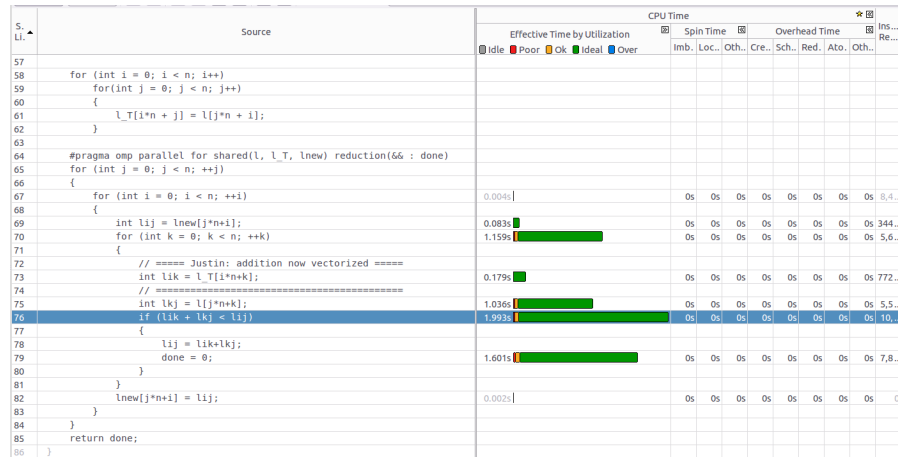


Figure 8: Initial Assembly Result

The assembly analysis after using data-type `char` is down below. Comparing with Figure 7, we can tell that the major saving comes from the addition between `lij` and `lkj`, which consist with what we would expect, since we can now fit more additions into vector register to be calculated in one cycle.

5.4 INITIAL TIMING RESULT

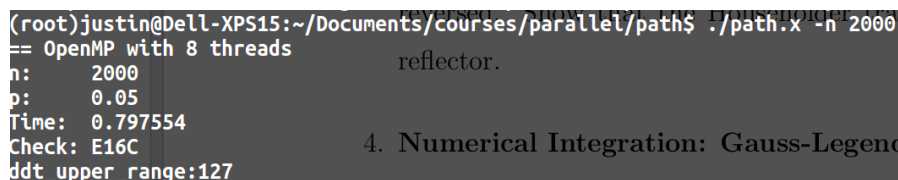


Figure 9: Midterm Timing Result

Midterm timing result is shown in Figure 9. As we can see the program is running with 8 threads using OpenMP and it takes 0.7976 seconds for 2000 vertices.

6 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.**