CS 5220 Project 3 – Team 6 Final Report

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

November 20, 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 and parallelization. Our evaluation result will be shown in Section 5. Finally, Section 6 concludes the report.

2 PROFILING

2.1 PROFILING

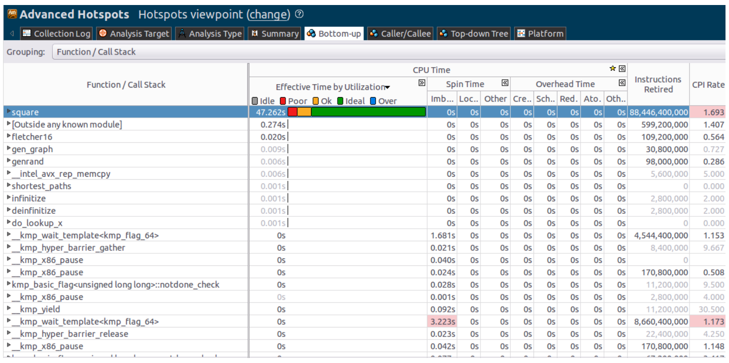
In order to find the bottlenecks of path, we first profiled our code using Intel's VTune Amplifier (It was broken in cluster, so we used it on local machine), as shown in Figure 1.

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

Figure 1: VTune Amplifier Command

2.2 Initial Profiling Result

Initial profile result can be found at Figure 2, and more detail in Figure 12.
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.



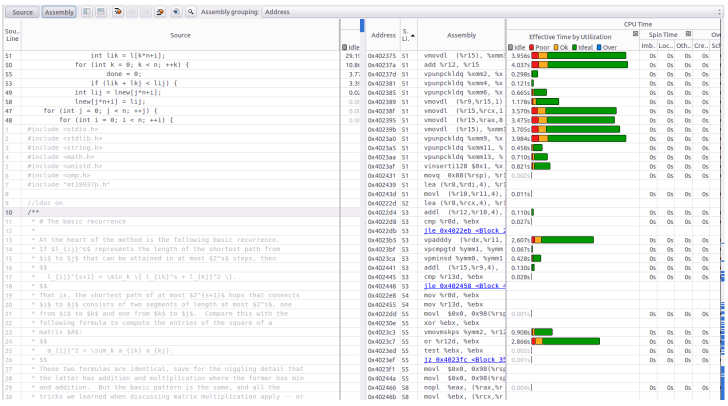
Advanced Hotspots Hotspots viewpoint (change)

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.	Ato.	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
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



Source | Assembly | Assembly grouping: Address

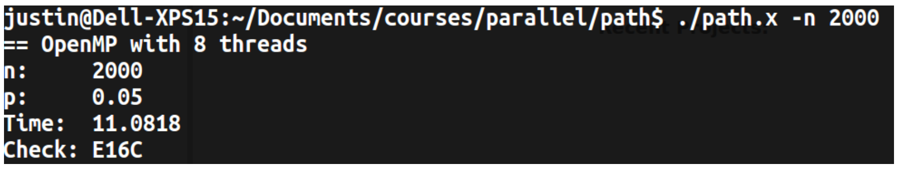
Sou.. Line	Source	▨ Idle
51	`int lik = l[k*n+i];`	29.1
50	`for (int k = 0; k < n; ++k) {`	10.8
55	`done = 0;`	3.7
53	`if (lik + lkj < lij) {`	3.3
49	`int lij = lnew[j*n+i];`	0.0
58	`lnew[j*n+i] = lij;`	0.0
47	`for (int j = 0; j < n; ++j) {`	0.0
48	`for (int i = 0; i < n; ++i) {`	0.0
1	`#include <stdio.h>`	
2	`#include <stdlib.h>`	
3	`#include <string.h>`	
4	`#include <math.h>`	
5	`#include <unistd.h>`	
6	`#include <omp.h>`	
7	`#include "mt19937p.h"`	
8		
9	`//ldoc on`	
10	`/**`	
11	`* # The basic recurrence`	
12	`*`	
13	`* At the heart of the method is the following basic recurrence.`	
14	`* If $l_{ij}^s$ represents the length of the shortest path from`	
15	`* $i$ to $j$ that can be attained in at most $2^s$ steps, then`	
16	`* $$`	
17	`* l_{ij}^{s+1} = \min_k \{ l_{ik}^s + l_{kj}^2 \}.`	
18	`* $$`	
19	`* That is, the shortest path of at most $2^{s+1}$ hops that connects`	
20	`* $i$ to $j$ consists of two segments of length at most $2^s$, one`	
21	`* from $i$ to $k$ and one from $k$ to $j$.  Compare this with the`	
22	`* following formula to compute the entries of the square of a`	
23	`* matrix $A$:`	
24	`* $$`	
25	`* a_{ij}^2 = \sum_k a_{ik} a_{kj}.`	
26	`* $$`	
27	`* These two formulas are identical, save for the niggling detail that`	
28	`* the latter has addition and multiplication where the former has min`	
29	`* and addition.  But the basic pattern is the same, and all the`	
30	`* tricks we learned when discussing matrix multiplication apply -- or`	

Address	S. Li.	Assembly	CPU Time				
			Effective Time by Utilization	Spin Time			
			▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Imb.	Loc..	Oth..	Cre..
0x402375	51	`vmovdl  (%r15), %xmm3`	3.956s	0s	0s	0s	0s
0x40237a	51	`add %r12, %r15`	4.037s	0s	0s	0s	0s
0x40237d	51	`vpunpckldq %xmm2, %x`	0.298s	0s	0s	0s	0s
0x402381	51	`vpunpckldq %xmm4, %x`	0.121s	0s	0s	0s	0s
0x402385	51	`vpunpckldq %xmm6, %x`	0.665s	0s	0s	0s	0s
0x402389	51	`vmovdl  (%r9,%r15,1)`	1.178s	0s	0s	0s	0s
0x40238f	51	`vmovdl  (%r15,%rcx,1`	3.570s	0s	0s	0s	0s
0x402395	51	`vmovdl  (%r15,%rax,8`	3.475s	0s	0s	0s	0s
0x40239b	51	`vmovdl  (%r15), %xmm1`	3.705s	0s	0s	0s	0s
0x4023a0	51	`vpunpckldq %xmm9, %x`	3.984s	0s	0s	0s	0s
0x4023a5	51	`vpunpckldq %xmm11, %`	0.458s	0s	0s	0s	0s
0x4023aa	51	`vpunpckldq %xmm13, %`	0.710s	0s	0s	0s	0s
0x4023af	51	`vinserti128 $0x1, %x`	0.821s	0s	0s	0s	0s
0x402431	51	`movq  0x88(%rsp), %r1`	0.002s	0s	0s	0s	0s
0x402439	51	`lea (%r8,%rdi,4), %r1`					
0x40243d	51	`movl  (%r10,%r11,4),`	0.011s	0s	0s	0s	0s
0x40222d	52	`lea (%r8,%rcx,4), %r1`					
0x4022d4	53	`addl  (%r12,%r10,4),`	0.110s	0s	0s	0s	0s
0x4022d8	53	`cmp %r8d, %ebx`	0.027s	0s	0s	0s	0s
0x4022db	53	`jle 0x4022eb <Block 2`					
0x4023b5	53	`vpadddy  (%rdx,%r11,`	2.607s	0s	0s	0s	0s
0x4023bf	53	`vpcmpgtd %ymm1, %ymm`	0.067s	0s	0s	0s	0s
0x4023ca	53	`vpminsd %ymm0, %ymm1`	0.428s	0s	0s	0s	0s
0x402441	53	`addl  (%r15,%r9,4),`	0.130s	0s	0s	0s	0s
0x402445	53	`cmp %r13d, %ebx`	0.028s	0s	0s	0s	0s
0x402448	53	`jle 0x402458 <Block 4`					
0x4022e8	54	`mov %r8d, %ebx`					
0x402455	54	`mov %r13d, %ebx`					
0x4022dd	55	`movl  $0x0, 0x98(%rsp`	0.001s	0s	0s	0s	0s
0x40230e	55	`xor %ebx, %ebx`					
0x4023c3	55	`vmovmskps %ymm2, %r12`	0.908s	0s	0s	0s	0s
0x4023c7	55	`or %r12d, %ebx`	2.866s	0s	0s	0s	0s
0x4023ed	55	`test %ebx, %ebx`	0.002s	0s	0s	0s	0s
0x4023ef	55	`jz 0x4023fc <Block 35`	0.001s	0s	0s	0s	0s
0x4023f1	55	`movl  $0x0, 0x98(%rsp`					
0x40244a	55	`movl  $0x0, 0x98(%rsp`					
0x402466	58	`nopl  %eax, (%rax,%r`	0.004s	0s	0s	0s	0s
0x40246b	58	`movl  %ebx, (%rcx,%r`					

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

3.1 Timing After 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

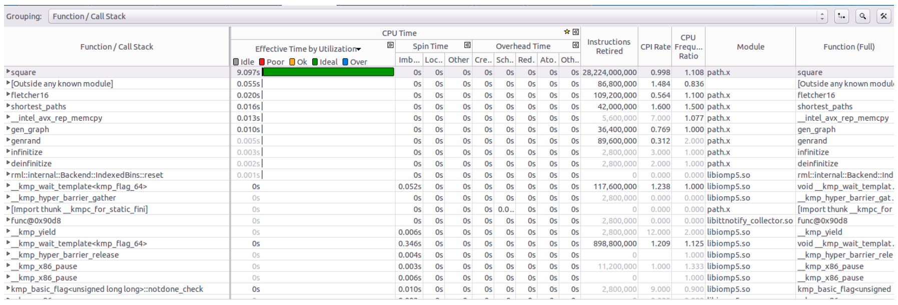
```
int lik = l[k*n+i]
```

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.

3.2 Timing After Vectorization

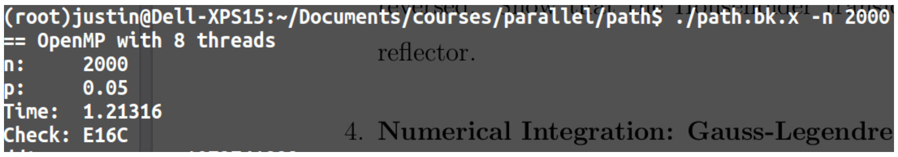


Grouping: Function / Call Stack

Function / Call Stack	CPU Time									Instructions Retired	CPI Rate	CPU Frequ... Ratio	Module	Function (Full)
	Effective Time by Utilization	Spin Time			Overhead Time									
	Idle Poor Ok Ideal Over	Imb...	Loc..	Other	Cre..	Sch..	Red.	Ato.	Oth..					
square	9.097s	0s	0s	0s	0s	0s	0s	0s	0s	28,224,000,000	0.998	1.108	path.x	square
[Outside any known module]	0.055s	0s	0s	0s	0s	0s	0s	0s	0s	86,800,000	1.484	0.836		[Outside any known modul
fletcher16	0.020s	0s	0s	0s	0s	0s	0s	0s	0s	109,200,000	0.564	1.100	path.x	fletcher16
shortest_paths	0.016s	0s	0s	0s	0s	0s	0s	0s	0s	42,000,000	1.600	1.500	path.x	shortest_paths
__intel_avx_rep_memcpy	0.013s	0s	0s	0s	0s	0s	0s	0s	0s	5,600,000	7.000	1.077	path.x	__intel_avx_rep_memcpy
gen_graph	0.010s	0s	0s	0s	0s	0s	0s	0s	0s	36,400,000	0.769	1.000	path.x	gen_graph
genrand	0.005s	0s	0s	0s	0s	0s	0s	0s	0s	89,600,000	0.312	2.000	path.x	genrand
infinitize	0.003s	0s	0s	0s	0s	0s	0s	0s	0s	2,800,000	3.000	1.000	path.x	infinitize
deinfinitize	0.002s	0s	0s	0s	0s	0s	0s	0s	0s	2,800,000	2.000	1.000	path.x	deinfinitize
rml::internal::Backend::IndexedBins::reset	0.001s	0s	0s	0s	0s	0s	0s	0s	0s	0	0.000	0.000	libiomp5.so	rml::internal::Backend::Ind
__kmp_wait_template<kmp_flag_64>	0s	0.052s	0s	0s	0s	0s	0s	0s	0s	117,600,000	1.238	1.000	libiomp5.so	void __kmp_wait_templat.
__kmp_hyper_barrier_gather	0s	0s	0s	0s	0s	0s	0s	0s	0s	2,800,000	0.000	0.000	libiomp5.so	__kmp_hyper_barrier_gat.
[Import thunk __kmpc_for_static_fini]	0s	0s	0s	0s	0s	0.0..	0s	0s	0s	0	0.000	0.000	path.x	[Import thunk __kmpc_for
func@0x90d8	0s	0s	0s	0s	0s	0s	0s	0s	0s	2,800,000	0.000	0.000	libittnotify_collector.so	func@0x90d8
__kmp_yield	0s	0.006s	0s	0s	0s	0s	0s	0s	0s	2,800,000	12.000	2.000	libiomp5.so	__kmp_yield
__kmp_wait_template<kmp_flag_64>	0s	0.346s	0s	0s	0s	0s	0s	0s	0s	898,800,000	1.209	1.125	libiomp5.so	void __kmp_wait_templat.
__kmp_hyper_barrier_release	0s	0.004s	0s	0s	0s	0s	0s	0s	0s	0		1.000	libiomp5.so	__kmp_hyper_barrier_rele
__kmp_x86_pause	0s	0.003s	0s	0s	0s	0s	0s	0s	0s	11,200,000	1.000	1.333	libiomp5.so	__kmp_x86_pause
__kmp_x86_pause	0s	0.006s	0s	0s	0s	0s	0s	0s	0s	0		1.000	libiomp5.so	__kmp_x86_pause
kmp_basic_flag<unsigned long long>::notdone_check	0s	0.010s	0s	0s	0s	0s	0s	0s	0s	2,800,000	9.000	0.900	libiomp5.so	kmp_basic_flag<unsigned

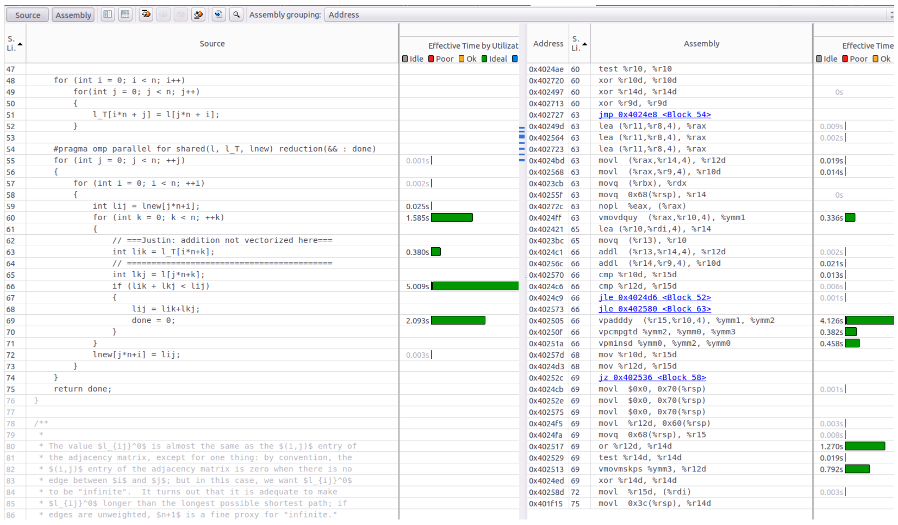
Figure 5: Profile Analysis After Vectorization

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.



```
(root)justin@Dell-XPS15:~/Documents/courses/parallel/path$ ./path.bk.x -n 2000
== OpenMP with 8 threads
n:      2000
p:      0.05
Time:   1.21316
Check:  E16C
```

Figure 6: Timing Result After Vectorization


Source
Assembly
Assembly grouping: Address
Effective Time by Utilizat
🟥 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦
Effective Time
▨ Idle 🟥 Poor 🟧 Ok

```
S. Li.  Source                                                             Effective Time
47
48          for (int i = 0; i < n; i++)
49              for(int j = 0; j < n; j++)
50              {
51                  l_T[i*n + j] = l[j*n + i];
52              }
53
54          #pragma omp parallel for shared(l, l_T, lnew) reduction(&& : done)
55          for (int j = 0; j < n; ++j)                                      0.001s
56          {
57              for (int i = 0; i < n; ++i)                                  0.002s
58              {
59                  int lij = lnew[j*n+i];                                   0.025s
60                  for (int k = 0; k < n; ++k)                              1.585s
61                  {
62                      // ===Justin: addition not vectorized here===
63                      int lik = l_T[i*n+k];                                0.380s
64                      // ==========================================
65                      int lkj = l[j*n+k];
66                      if (lik + lkj < lij)                                 5.009s
67                      {
68                          lij = lik+lkj;
69                          done = 0;                                        2.093s
70                      }
71                  }
72                  lnew[j*n+i] = lij;                                       0.003s
73              }
74          }
75          return done;
76      }
77
78      /**
79       *
80       * The value $l_{ij}^0$ is almost the same as the $(i,j)$ entry of
81       * the adjacency matrix, except for one thing: by convention, the
82       * $(i,j)$ entry of the adjacency matrix is zero when there is no
83       * edge between $i$ and $j$; but in this case, we want $l_{ij}^0$
84       * to be "infinite".  It turns out that it is adequate to make
85       * $l_{ij}^0$ longer than the longest possible shortest path; if
86       * edges are unweighted, $n+1$ is a fine proxy for "infinite."
```

```
Address    S. Li.  Assembly                                      Effective Time
0x4024ae   60      test %r10, %r10
0x402720   60      xor %r10d, %r10d
0x402497   60      xor %r14d, %r14d                              0s
0x402713   60      xor %r9d, %r9d
0x402727   63      jmp 0x4024e8 <Block 54>
0x40249d   63      lea (%r11,%r8,4), %rax                        0.009s
0x402564   63      lea (%r11,%r8,4), %rax                        0.002s
0x402723   63      lea (%r11,%r8,4), %rax
0x4024bd   63      movl  (%rax,%r14,4), %r12d                    0.019s
0x402568   63      movl  (%rax,%r9,4), %r10d                     0.014s
0x4023cb   63      movq  (%rbx), %rdx
0x40255f   63      movq  0x68(%rsp), %r14                        0s
0x40272c   63      nopl  %eax, (%rax)
0x4024ff   63      vmovdquy  (%rax,%r10,4), %ymm1                0.336s
0x402421   65      lea (%r10,%rdi,4), %r14
0x4023bc   65      movq  (%r13), %r10
0x4024c1   66      addl  (%r13,%r14,4), %r12d                    0.002s
0x40256c   66      addl  (%r14,%r9,4), %r10d                     0.021s
0x402570   66      cmp %r10d, %r15d                              0.013s
0x4024c6   66      cmp %r12d, %r15d                              0.006s
0x4024c9   66      jle 0x4024d6 <Block 52>                       0.001s
0x402573   66      jle 0x402580 <Block 63>
0x402505   66      vpadddy  (%r15,%r10,4), %ymm1, %ymm2          4.126s
0x40250f   66      vpcmpgtd %ymm2, %ymm0, %ymm3                  0.382s
0x40251a   66      vpminsd %ymm0, %ymm2, %ymm0                   0.458s
0x40257d   68      mov %r10d, %r15d
0x4024d3   68      mov %r12d, %r15d
0x40252c   69      jz 0x402536 <Block 58>
0x4024cb   69      movl  $0x0, 0x70(%rsp)                        0.001s
0x40252e   69      movl  $0x0, 0x70(%rsp)
0x402575   69      movl  $0x0, 0x70(%rsp)
0x4024f5   69      movl  %r12d, 0x60(%rsp)                       0.003s
0x4024fa   69      movq  0x68(%rsp), %r15                        0.008s
0x402517   69      or %r12d, %r14d                               1.270s
0x402529   69      test %r14d, %r14d                             0.019s
0x402513   69      vmovmskps %ymm3, %r12d                        0.792s
0x4024ed   69      xor %r14d, %r14d
0x40258d   72      movl  %r15d, (%rdi)                           0.003s
0x401f15   75      movl  0x3c(%rsp), %r14d
```

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.

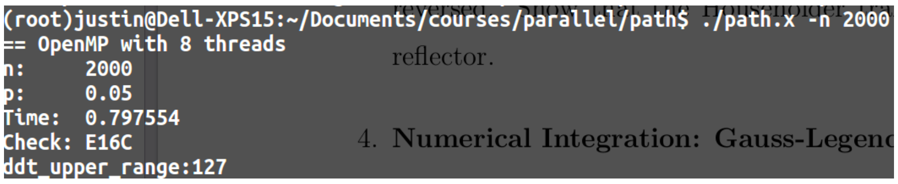
3.3 Data-Type Optimization

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.

3.4 Tuned OpenMP Timing Result

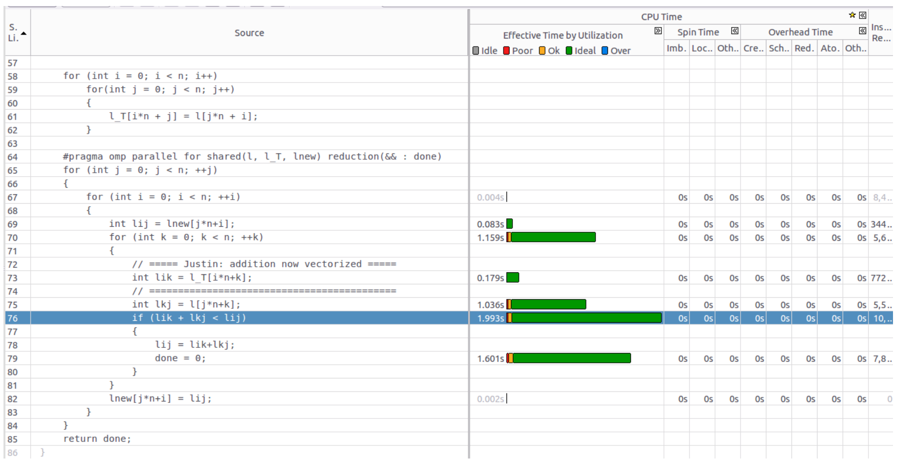


```
(root)justin@Dell-XPS15:~/Documents/courses/parallel/path$ ./path.x -n 2000
== OpenMP with 8 threads
n:      2000
p:      0.05
Time:   0.797554
Check:  E16C
ddt_upper_range:127
```

Figure 8: Timing Result After OpenMP tuning

Timing result before introducing MPI is shown in Figure 8. As we can see the program is running with 8 threads using OpenMP and it takes 0.7976 seconds for 2000 vertices.

3.5 Tuned OpenMP Profiling Result



S. Li.	Source	CPU Time									Ins... Re...
		Effective Time by Utilization	Spin Time			Overhead Time					
		▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	Imb.	Loc..	Oth..	Cre..	Sch..	Red.	Ato.	Oth..	
57											
58	`for (int i = 0; i < n; i++)`										
59	`    for(int j = 0; j < n; j++)`										
60	`    {`										
61	`        l_T[i*n + j] = l[j*n + i];`										
62	`    }`										
63											
64	`#pragma omp parallel for shared(l, l_T, lnew) reduction(&& : done)`										
65	`for (int j = 0; j < n; ++j)`										
66	`{`										
67	`    for (int i = 0; i < n; ++i)`	0.004s	0s	0s	0s	0s	0s	0s	0s	0s	8,4..
68	`    {`										
69	`        int lij = lnew[j*n+i];`	0.083s	0s	0s	0s	0s	0s	0s	0s	0s	344..
70	`        for (int k = 0; k < n; ++k)`	1.159s	0s	0s	0s	0s	0s	0s	0s	0s	5,6..
71	`        {`										
72	`            // ===== Justin: addition now vectorized =====`										
73	`            int lik = l_T[i*n+k];`	0.179s	0s	0s	0s	0s	0s	0s	0s	0s	772..
74	`            // ==========================================`										
75	`            int lkj = l[j*n+k];`	1.036s	0s	0s	0s	0s	0s	0s	0s	0s	5,5..
76	`            if (lik + lkj < lij)`	1.993s	0s	0s	0s	0s	0s	0s	0s	0s	10,..
77	`            {`										
78	`                lij = lik+lkj;`										
79	`                done = 0;`	1.601s	0s	0s	0s	0s	0s	0s	0s	0s	7,8..
80	`            }`										
81	`        }`										
82	`        lnew[j*n+i] = lij;`	0.002s	0s	0s	0s	0s	0s	0s	0s	0s	0
83	`    }`										
84	`}`										
85	`return done;`										
86	`}`										

Figure 9: 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 `ljk`, which consist with what we would expect, since we can now fit more additions into vector register to be calculated in one cycle.

4 Parallelization

4.1 OpenMP

Parallelization with OpenMP is already implemented in the original code. Within a single node, the jobs are statically distributed among different threads. Depending on the way of array accessing, we won't have problem with cache coherence, because we are reading from and writing to different memory locations.

4.2 Boardcast Algorithm

To make our code scales better, we implement an algorithm similar to broadcast algorithm for matrix multiplication. For details, see:

http://www.cs.berkeley.edu/ yelick/cs267-sp04/lectures/13/lect13-pmatmul-6x.pdf

Assuming we are working with number of ranks being a perfect square. Then, the procedure of our algorithm can be outlined as follows:

1. Generate the random graph from rank0.

2. On rank0, partition the whole matrix into a 2-d array of blocks. Each rank will be in charge of one block.
3. Rank0 send the information about the whole matrix to different processors.
4. Calculate the shortest path:
    a) MPI ranks within the same column share their blocks.
    b) MPI ranks within the same row share their blocks.
    c) Calculate the shortest path by the "square" method along the cols and rows. The "done" variable is determined by the "logical_and" operation.
    d) Block until all the MPI ranks return, then use MPI_reduce to decide whether the iteration has finished.
5. Once all the job has been finished, gather the updated matrix back to rank0.

4.3 MPI Implementation

1. To distribute blocks to all MPI ranks, MPI_Scatter method is used.

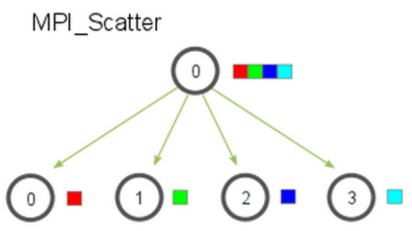


Figure 10: Initial Assembly Result

2. During the calculation of the whole graph, Ranks in the same Column/Row are sharing blocks through MPI_Allgather.

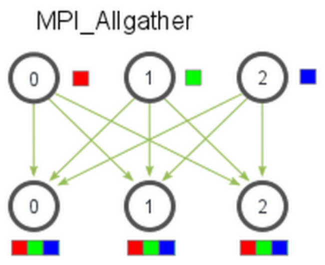


Figure 11: Initial Assembly Result

3. To gather the final result to rank 0, MPI_Scatter method is used.

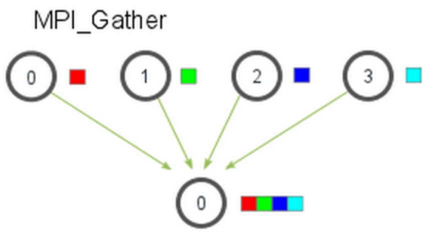


Figure 12: Initial Assembly Result

5 Evaluation

5.1 Performance Model

We build a performance model as follows: (We referred to GROUP 11)

Serial Performance
$S = N^3 * T$

Parallel Speedup
$Speedup = S/(Overhead + S/p)$

where N is the number of vertices T is time to execute atomic operations, $Overhead$ is the overhead time for parallelization such as communicating and p is the number of cores we can use.
The basic intuition behind this simple model is that as we can increase the N, the number of vertices, we could achieve the higher speed up, and the overhead becomes smaller relatively.

5.2 OpenMP Scaling

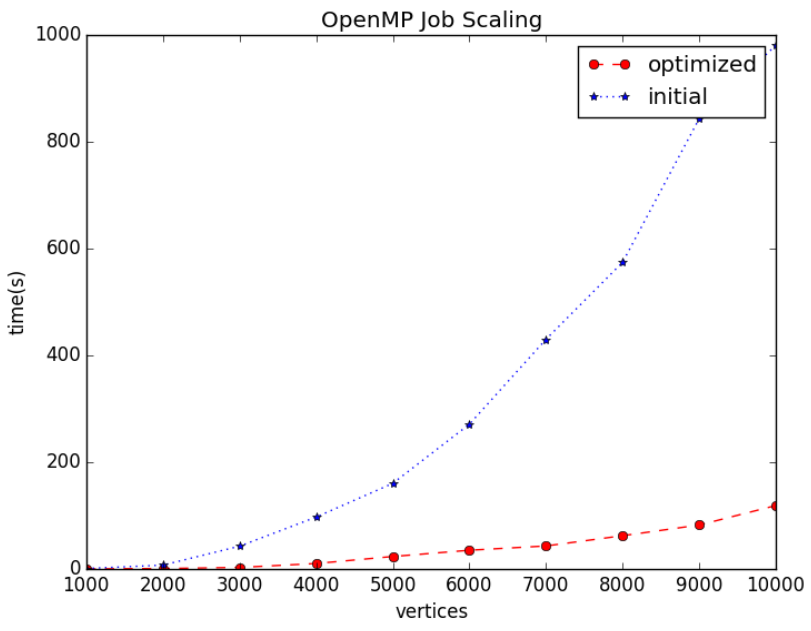


Figure 13: OpenMP Job Scaling with 1 Node - 24 Threads

The picture shows the scaling paradigm of the original code and optimized OpenMP code. From this picture, we can see there is a factor of 10 improvement after our optimization. This picture also gives information about the scaling of the calculation. Non-linear regression fit of this curve gives a result of $O(N^{3.22})$.

In theory, this algorithm should have a $O(N^3)$ scaling with respect to one single iteration. The result different from the idea case because for a smaller graph, it can be fit into cache.

5.3 MPI Scaling

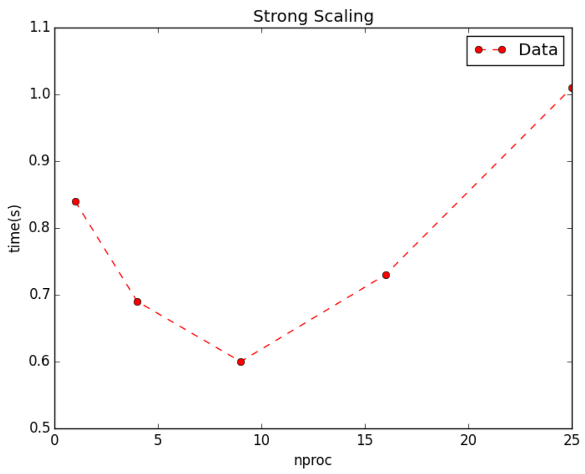


Figure 14: MPI Strong Scaling Naive

This is our first implementation of MPI code, tested with 2000 by 2000 graph. This strong scaling diagram shows the communication overhead is very big, and it's performance is no better than serial code at very large number of ranks.

5.4 Non-Blocking Communication

Notice that we don't need the whole column and row of the matrix to begin the shortest path calculation, we just need the first column block and first row block, then we can do the calculation. So we implemented a new version of MPI based on non-blocking communication MPI_Ibcast. The procedure is as follows:

1. The rank owning the first column block or row block do a blocking board-cast.
2. The rank owning the second column block or row block do a non-blocking board-cast.
3. Wait for the first column block and row block.
4. Update the graph based on the first column block and row block.
5. The rank owning the third column block or row block do a non-blocking board-cast.
6. Wait for the second column block and row block.
7. Go to the 4th step.

After the optimization, we can see improvement.

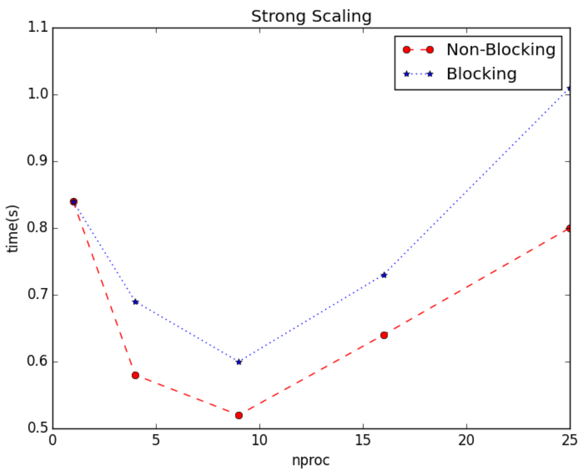


Figure 15: MPI Strong Scaling Non-Blocking

5.5 MPI Weak Scaling

Since this algorithm is not linear scaling, it is hard to define weak scaling, because the problem size grows $O(N^3)$. The data tested is listed below:

	1	4	9	16	25
2000	0.84				
4000	5.74	2.79			
6000	13.30		16.40		
8000	23.59			68.91	
10000	33.18				162.419

From this graph, we can calculate the speed up factor represented in the following graph.

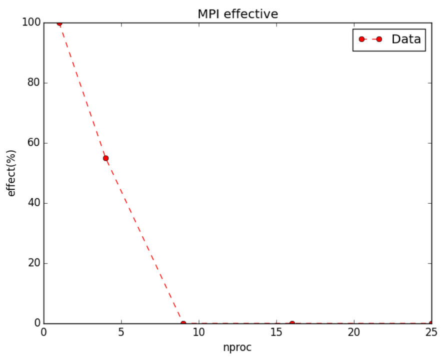


Figure 16: MPI Effective

From the result above, MPI is not used effectively. We think one reason is that the calculation of one single job is too small, because this algorithm uses additions to update the graph. So the fraction of Communication vs Calculation will be really high.

6 Summary

In this project, we did the following:

1. Profiled to find the hotspots, look into the vectorization report.
2. Optimized our code based on our finding, get a factor of more than 10.
3. Successfully parallel our code through MPI, board-cast algorithm, get the right answer.
4. Use non-blocking communication to speed up the MPI code.
5. Build up a mode to analysis the performance of our MPI code in terms of strong scaling and MPI efficiency.

From our result, we learned:

1. Non-blocking communication can help speed up the program, but it might not be that ideal.
2. When the size of a job is not large enough, the efficiency of MPI calculation maybe influenced.