Path Stage 1 – Group 2 Mid Report

Batu Inal(bi49), Kenneth Lim(kl545), Wensi Wu (ww382)

November 7, 2015

1 Introduction

In this project, we analyze the OpenMP version of Floyd-Warshall algorithm by profiling the code using Vtune to identify bottlenecks. The Vtune profiling result is shown in section §2. To tune the code, we explore different compiler flags (section §**??**) and slightly vectorized the code (section §4). Finally we perform strong and weak scaling in section §5.

2 Profiling

2.1 Identify bottlenecks

To define the bottlenecks of `path.C`, we profiled our code using Intel's VTune Amplifier to understand which part of the code takes the most time to run. VTune Amplifier commands is shown in Figure 1.

```
amplxe-cl -collect advanced-hotspots ./path
    amplxe-cl -R hotspots -report-output vtune-report.csv -format csv -csv-delimiter comma
```

Figure 1: VTune Amplifier Command

2.2 Initial Timing Result

As shown in Figure 2, the majority of time is spent inside the `sqaure` function. We will take about how we optimize the performance of the `square` function in §4.

```
Function                     Module       CPU Time CPU Time:Ideal
---------------------------------------  --------     --------------
square              path.x         45.14s 39.03s
_kmpbarrier                  libiomp5.so    6.485s  0.16s
_kmpc_reduce_nowait          libiomp5.so    2.979s   0s
_kmp_fork_barrier            libiomp5.so    2.553s  0.03s
gen_graph               path.x        0.030s 0s
_intel_ssse3_memcpy          path.x        0.030s 0s
fletcher16                   path.x        0.030s 0s
```

Figure 2: Initial Profile Result

3 Compiler Flags

With some experience from the previous project, we decided to play around with different compiler flags to examine whether it will help us the speedup of the code. We found that the icc compiler flag shown below gave us 8 times speedup to the `deinfinitize` function.

```
OPTFLAGS=-O3 -no-prec-div -opt-prefetch -xHost -ansi-alias -ipo -restrict
```

4 Vectorization

As we discussed in section §2, the majority of time is spent in the `sqare` function. Looking at the vectorization report generated by the compiler default code, we see that the compiler did not vectorize the `for` loop inside the `sqaure` function because it assumes dependencies in the loop.

```
LOOP BEGIN at mt19937p.c(59,5) inlined into path.c(228,14)
remark #15344: loop was not vectorized: vector dependence prevents vectorization
remark #15346: vector dependence: assumed FLOW dependence between mt line 60 and mt line
LOOP END
```

To clear the dependency, we simply added `#pragma` vector aligned in the inner loop as shown below to ensure the compiler that the lij is indeed aligned. Adding the `#pragma` instruction gave us a 4 times speed up to the `square` function.

```
#pragma omp parallel for shared(l, lnew) reduction (&& : done)
for (int j = 0; j<n; ++j){
for (int k = 0; k < n; ++k) {
int lkj = l[j*n+k];

#pragma vector aligned
for (int i = 0; i < n; ++i){
int lij = lnew[j*n+i];
int lik = l[k*n+i];

if (lik +lkj < lij){
lij = lik+lkj;
done=0;
}

lnew[j*n+i] = lij;
}
}
}
```

5 Evaluation

5.1 What Worked

5.2 What did not work

6 Future Work

- **Blocking.** We plan on performing some kind of blocking to improve the perfomance of the code.
- **Using MPI.** We might try to implement MPI and compare the performance with OpenMP.