# Parallelzing an All-Pairs Shortest Path Algorithm

Bangrui Chen (bc496), Markus Salasoo (ms933), and Calvin Wylie (cjw278)

November 19, 2015

## Introduction

In this report we describe the implementation and tuning of a parallel all-pairs shortest path algorithm.

## Profiling the given code

### VTune Amplifier

Given a working implementation of parallelized Floyd-Warshall algorithm, we can further increase the performance by profiling the computation. This will identify areas in the code that run much slower than the rest, giving opportunities to implement speedups. We used the VTune Amplifier to profile our code, calling our executable through the profiler with

```
amplxe-cl -collect hotspots ./path.x
```

| Function | Module | CPU Time |
|---|---|---|
| square | path.x | 42.751s |
| __kmp_barrier | libiomp5.so | 9.623s |
| __kmpc_reduce_nowait | libiomp5.so | 3.498s |
| __kmp_fork_barrier | libiomp5.so | 2.738s |
| genrand | path.x | 0.030s |
| fletcher16 | path.x | 0.020s |
| __intel_ssse3_rep_memcpy | path.x | 0.020s |
| __kmp_join_call | libiomp5.so | 0.010s |
| __kmp_launch_thread | libiomp5.so | 0.010s |
| __kmp_fork_call | libiomp5.so | 0.010s |

The hotspots analysis results tell us that the most expensive compute-time occurs in *square*(). This is expected because *square*() is the only method with nested loops carrying out the algorithm (*write_matrix*() only prints values before and after our algorithm has run).

There are two other methods in our top CPU time list from path.c however any effort put towards optimizing those would not be well spent because they are orders of magnitude faster than *square*().

We can dig deeper into *square*()'s performance by running the command

```
amplxe-cl -report hotspots -source-object function="square"
```

```
Source Line  Source                                  CPU Time
-----------  --------------------------------------  --------
47           for (int j = 0; j < n; ++j) {
48               for (int i = 0; i < n; ++i) {         0.020s
49                   int lij = lnew[j*n+i];            0.020s
50                   for (int k = 0; k < n; ++k) {     8.699s
51                       int lik = l[k*n+i];          25.457s
52                       int lkj = l[j*n+k];
53                       if (lik + lkj < lij) {        3.759s
54                           lij = lik+lkj;
55                           done = 0;                 4.796s
```

This tells us the CPU time spent at many points within the method. We can see the majority of the time is spent retrieving a value from the array in the innermost loop. Observe that the matrix accesses are not optimized because we fetch non-contiguous memory each iteration and it is taking a long time. We can re-order the loops to better access our desired values and improve the runtime.

### Vectorization Report

If we analyze the given code's vectorization report, we can try to adjust some code to take advantage of the compiler's vectorization capabilites. Compiling the code with flags

−qopt−report=5 −qopt−report−phase=vec

will produce a report summarizing the compiler's ability to vectorize our code. In summary, the given code path.c is reported to be mostly vectorized. Most estimated potential speedup values were greater than 5, yielding a large factor of speedup. Some loops were not vectorizeable because they contained print statements. Another non-vectorizeable loop was in $fletcher16()$, but our profiling results show this method to be insignificant in runtime comparison to $square()$ so we will focus our efforts towards $square()$.

## OpenMP Scaling Study

We did both strong scaling and weak scaling study using the provided OpenMP code.

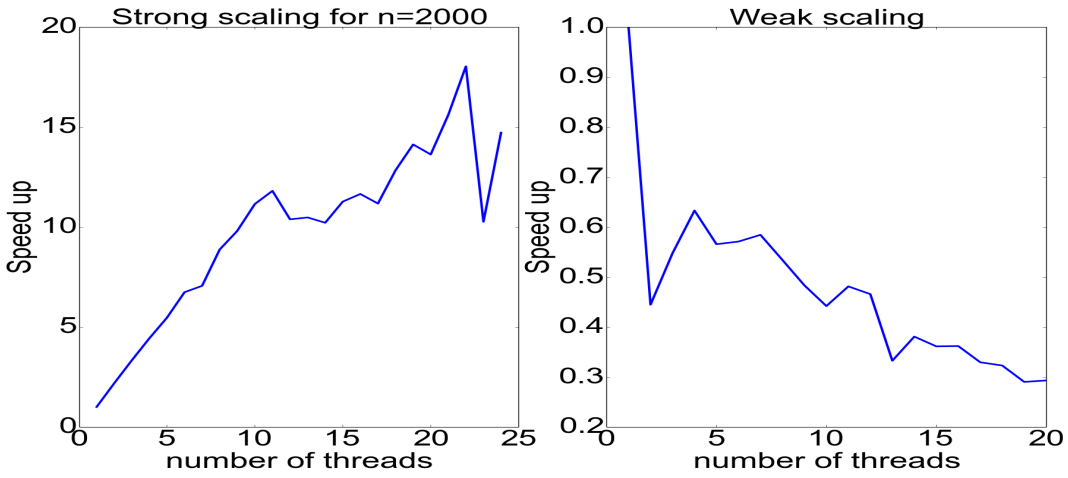
### Strong Scaling

In the strong scaling, we compare parallel to serial time for a fixed size problem. Since the graph is randomly generated, for each experiment, we did 10 simulation and use the average time as their running time. Here we choose the problem size $n = 2000$ and the speedup is summarized in left plot of Figure 1. As can be seen from the left plot of Figure 1, when we increase the number of threads from 1 to 12, the speed up grows linearly with respect to the number of threads. After 12 threads, the speedup doesn't grow linearly due to the excess amount of overheading and synchronization.

### Weak Scaling

In weak scaling, you need to increase both the number of processors and the amount of jobs such that each processor has the same amount of jobs. In this problem, the time complexity is $O(n^3 \log n)$

Figure 1: scaling study




and we let each processor has approximately $O(2000^3 \log(2000))$ amount of work. The results are summarized in the right figure of Figure 1. As can be seen from the right figure of Figure 1, the speedup is actually decreasing when you increase the amount of work and the number of processors. This could due to the overheading and synchronization when the number of processors is large.

## MPI Implementation

We implemented a parallel version of the shortest path algorithm using the Message Passing Interface (MPI). In our implementation, each processor is responsible for updating a portion of the shortest path matrix $l$. Specifically, we partition by column, so that the processor labelled $k$ (for $k = 1, \ldots, p$), updates $l_{ij}$ for $j \in J_k$, where the $(J_k)_{1 \leq k \leq p}$ define a partion of $\{1, 2, \ldots, n\}$. With this strategy the computation pattern for updating the shortest path matrix remains the same, we are simply looping over a smaller sets of columns.

After each processor has finished updating it's corresponding columns in the update $l_{ij}^{s+1} = \min_k \{l_{ik}^s + l_{jk}^s\}$, we call *MPI_Allgather* so that each processor gets the fully updated $l^{s+1}$ for the next step. Each processor also keeps a local "done" flag that will be true if nothing was updated. In order to determine whether to terminate or not, we call *MPI_Allreduce* with a logical AND operator on the local done flags.

### Predicting Speedup

We considered a simple alpha-beta model to predict the speedup of our MPI implementation. This models the communication time as

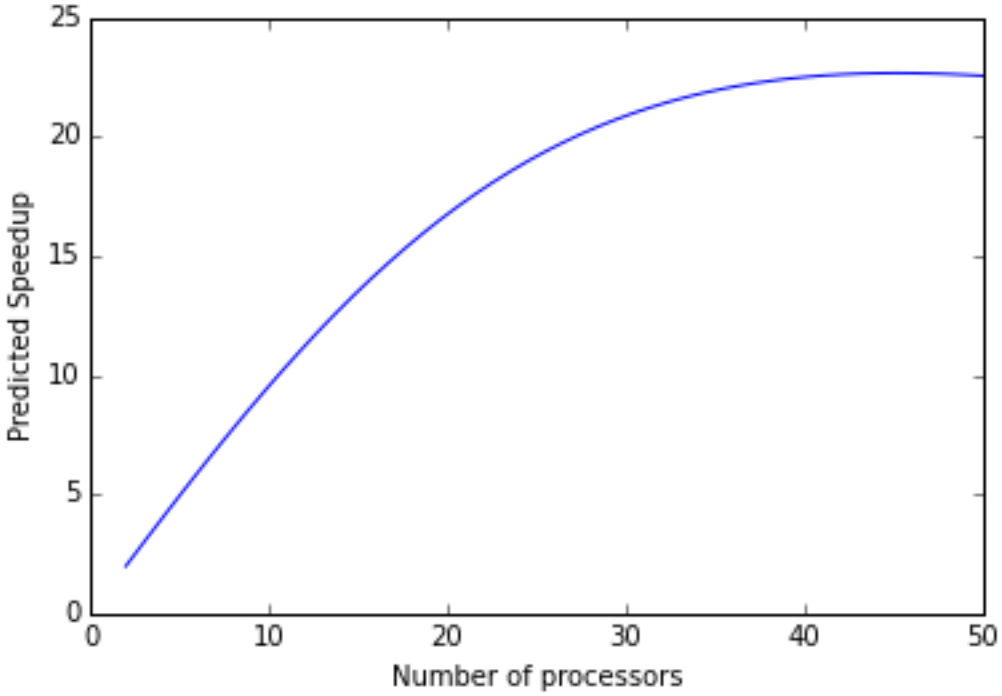
$$t_{comm} = \alpha + \beta * M$$

where $\alpha$ is latency, $\beta$ is inverse bandwidth, and $M$ is the message size. On the totient cluster on a single node, the time per kilobyte to communicate was previously estimated [1] to be

$$0.28 + 0.097 \times kB \quad \mu s$$

[1] `http://cornell-cs5220-f15.github.io/slides/2015-10-08-model.html`

Figure 2: Predicted MPI Speedup




Thus the total time for a single iteration in seconds can be modeled as

$$\begin{aligned} t_{total} &= t_{serial} + t_{comm} \\ &= O\left(\left(\frac{n}{p}\right)^3\right) + \frac{0.28 + 0.097(4n^2)}{10^6} \end{aligned}$$

We estimated the correct value of $t_{serial}$ by comparing against serial timing experiments, and we estimate the number of iterations to be $\log n$. A plot of estimated speedup for $n = 2016$ can be seen in Figure 2.

As seen in the plot, we should expect to get roughly linear speedup up until around 40 processors, after which communication costs would start to overtake the gains from paralellization.

### Scaling Study

## Tuning

We chose to focus on tuning the OpenMP version of the code.

Since this problem is closely related to the matrix multiplication, we used the same tricks as we did in tuning matrix multiplication, which are copy optimization and blocking. Using copy optimization, we store the transpose of the shortest path matrix to increase the possibility of cache hits. Implementing it will give us 3x to 4x speed up. Similarly, by using blocking, we can better parallellize independent jobs and increase the possibility of cache hits. Combining these two methods together, we achive the speed up seen in Figure 5. As you can see, the speedup can be up to 10x.

Figure 3: MPI Strong Scaling

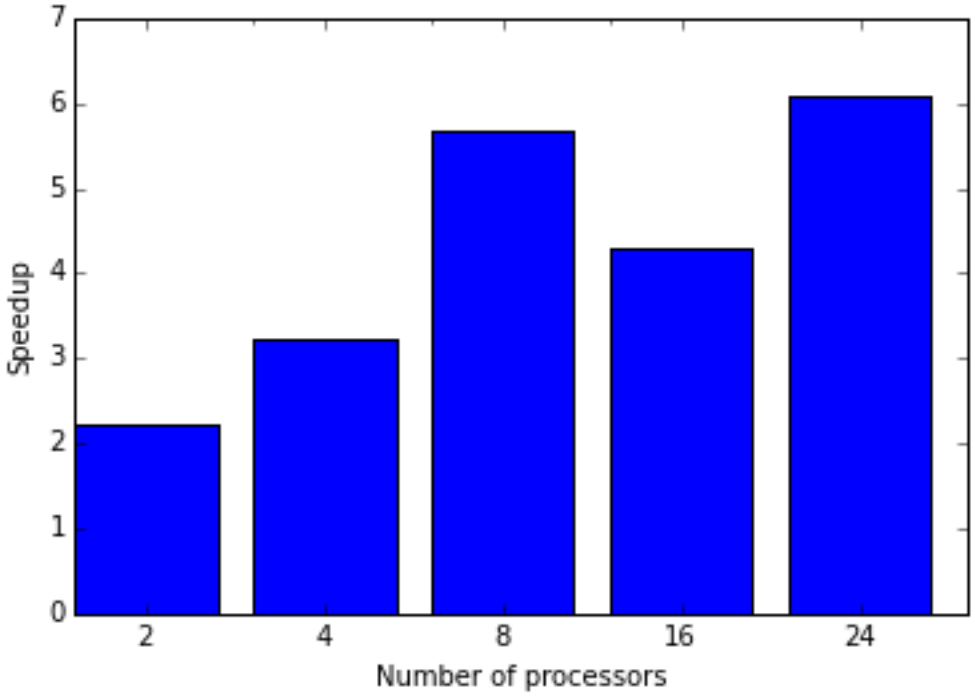



Figure 4: MPI Weak Scaling

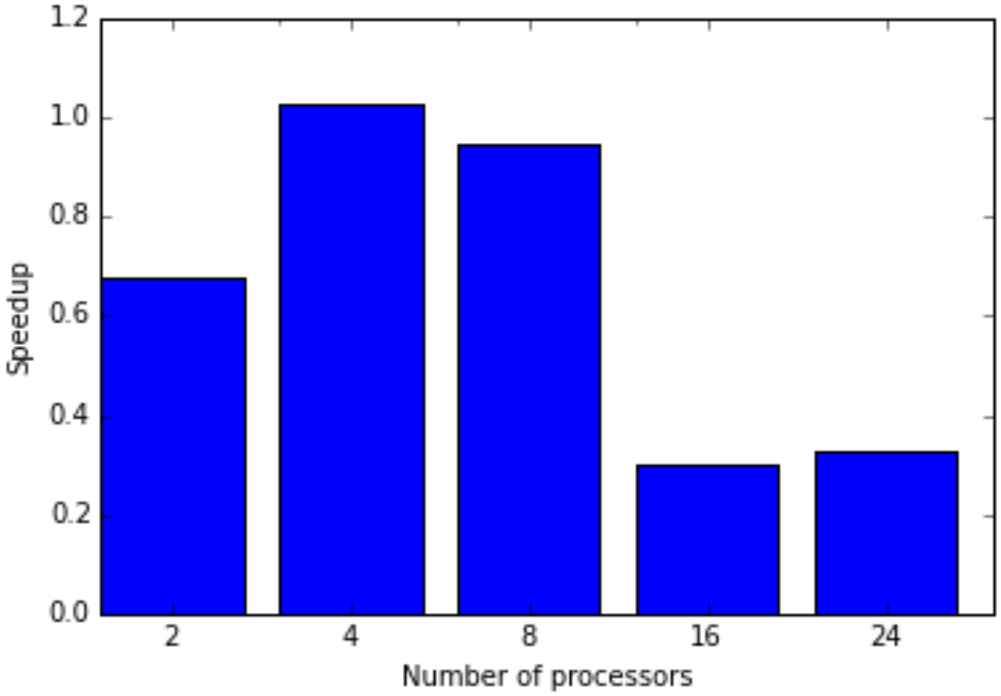
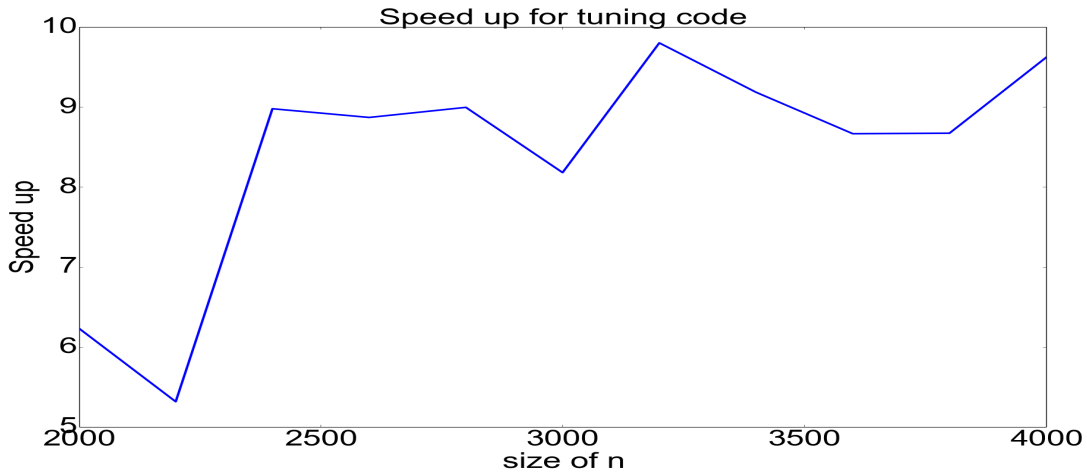


Figure 5: speed up for the tuned code




The benchmark is the original code using OpenMP with 24 threads.