CS 5220: Project 3
All-Pairs Shortest Path

Saugata Ghose, Shreesha Srinath, Jonathan Tse
{*sg532, ss2783, jdt76*}*@cornell.edu*

***Abstract*—We implement a modified Floyd-Warshall algorithm to solve the all-to-all shortest paths problem in $O(n^3 \log n)$ time using MPI. Through the use of non-blocking round-robin message passing and other memory optimizations, we show super-linear speedup for a 4000 node graph problem, achieving a 23x speedup for 8 threads. In contrast, our reference OpenMP implementation achieved a 5x speedup for the same problem set executing 8 threads.**

I. Introduction

Several algorithms have been proposed to solve the all-pairs shortest path, where one must find the path between any two nodes in a graph such that the sum of the weights of the constituent edges is minimized. The Floyd-Warshall algorithm is one such example. The graph under inspection can be represented as an adjacency matrix, where each element corresponds to the weight of the edge between two nodes in the graph. In the case of a directed graph, element $(2, 1)$ represents the directed edge from node 2 to node 1, which is anti-parallel to the edge stored at element $(1, 2)$.

Starting with an adjacency matrix that represents the edges that directly connect nodes together, we can apply the Floyd-Warshall algorithm. A modified version of the original algorithm follows the recurrence shown in Equation 1 for the shortest path l_{ij}^{s+1} between nodes i and j for iteration $s + 1$.

$$l_{ij}^{s+1} = \min\{l_{ik}^{s} + l_{kj}^{s}, l_{ij}^{s}\} \tag{1}$$

l_{ik}^{s} and l_{kj}^{s} represent the length of the shortest path from i to k and from k to j, respectively, that can be attained in at most 2^s hops. Note that in our particular instance of the shortest paths problem formulation, we have an unweighted, directed graph.

The original Floyd-Warshall algorithm operates in $O(n^3)$. Given n nodes, and the restriction of the original recurrence computation that we can can only examine nodes 0 through k, all of the shortest paths through node $k + 1$ can be found in $2n^2$ operations. Since we need to iterate through $\forall k, k < n$, this gives us $2n^3$ operations to find all shortest paths. Our modified version takes a little longer, operating on order $O(n^3 \log n)$, as can be seen in Figure 1. We choose to use the recurrence shown in Equation 1 as it is identical to matrix-matrix multiply in its memory access pattern.

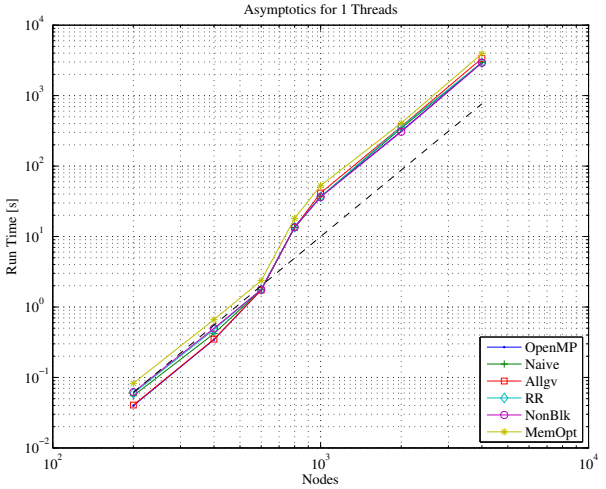

Fig. 1. Asymptotic behavior for single-threaded execution. The dotted black line is the function $10^{-9}x^3 \log_2 x$, which illustrates the asymptotic behavior.

TABLE I
Intel Xeon E5504 Processor Specifications [1]

Frequency	2.0 GHz
Number of cores	4
Cache line size	64 B
IL1/DL1 size	32 KB / 32 KB
IL1/DL1 associativity	4-way / 8-way
L2 size	256 KB, private
L2 associativity	8-way
L3 size	4 MB, shared
L3 associativity	16-way

We convert a reference OpenMP implementation of the modified Floyd-Warshall algorithm to use the MPI protocol. MPI is a popular standardized portable message passing system. It is used to program a parallel distributed-memory system, though it can also be implemented for shared-memory machines. In a distributed memory system, data is communicated between processors by passing explicit messages. Each processor owns a piece of the data, and does not exchange data unless required for any dependent computations. The MPI standard provides various language-independent communication protocols to distribute data.

The goals of programming in a distributed programming environment are to reduce the amount of required communication, and to overlap communication with computation to hide the cost of passing messages. This differs from programming in a shared memory system, where the targets are to reduce the number of critical sections and synchronization phases. For both systems, one cannot ignore the structure of the memory hierarchy.

In the following sections, we demonstrate several modifications to the initial MPI program to better take advantage of the protocol behavior.

II. Experimental Setup

The tunings described by this paper are targeted for an Intel Xeon E5504 quad-core processor, which is part of the Core i7 family. Our code is restricted to using two of these processors, which are connected together using the Intel QuickPath Interface (QPI). Table I shows the relevant parameters of the processor for this project. All binaries were compiled using GCC version 4.4.3, with integrated OpenMP support, on an Intel Xeon E5405 quad-core processor–part of the Core 2 family. MPI version 2.1 was used.

Note that for our speedup comparisons, we use the OpenMP implementation, with a single thread, as our performance baseline. For all results, we verified checksums for all outputs with the reference checksums from the OpenMP executions. Our evaluations were performed with a probability of 0.05 that any two nodes would be directly connected.

III. OpenMP Implementation

We start by analyzing a reference implementation using OpenMP. We see that the shortest path problem has both data level parallelism and task level parallelism. The data parallelism is easy to visualize, as operations on each element of the adjacency matrix are independent of other data element operations within the same iteration of the recurrence. The provided OpenMP implementation maps the algorithm to available threads by assigning a subset of destination nodes to each processor, as seen in Figure 2.

The computation for all elements in a column is split between the threads by striping columns amongst the available parallel threads. The success of the parallel code depends on successive squaring

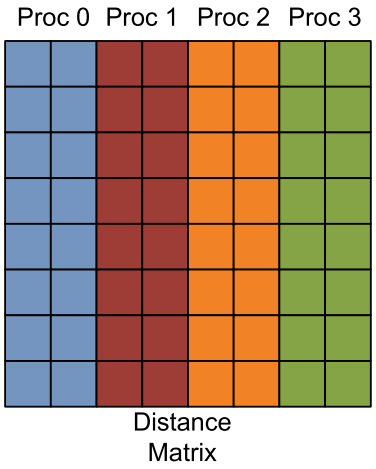

Fig. 2. Striping of the distance matrix over multiple processors.

operations until the squaring function no longer produces any changes in the successive matrix. This is based on the fact that for any loop-free path in a graph with n nodes, each node is passed through once. Therefore, the algorithm successfully can determine the shortest paths in $\lceil \log_2 n \rceil$ steps.

The `shortest_paths` function implements this by spinning on in a loop until it receives a `done` flag from the `square` function. The square function in the code implements the recurrence calculation in a nested `for` loop, as seen in Code Block 1.

Code Block 1. The `square()` function implemented using OpenMP.

```
1  int done = 1;
2  #pragma omp parallel for shared(l, lnew) \
3     reduction(&& : done)
4  for (int j = 0; j < n; ++j) {
5     for (int i = 0; i < n; ++i) {
6        int lij = lnew[j*n+i];
7        for (int k = 0; k < n; ++k) {
8           int lik = l[k*n+i];
9           int lkj = l[j*n+k];
10          if (lik + lkj < lij) {
11             lij = lik+lkj;
12             done = 0;
13          }
14       }
15       lnew[j*n+i] = lij;
16    }
17 }
18 return done;
```

A significant bottleneck at the end of each iteration in the OpenMP implementation is the reduction of the `done` flag and the subsequent implicit barrier due to the parallel `for` loop. This synchronization limits the scalability of the program. In addition, for larger node sizes, without systematic memory optimizations, the parallel performance will not scale. Tuning the memory performance by blocking for better cache locality should improve scalability.

Figures 3 and 4 show how the OpenMP implementation scales with the number of nodes. We see that the scaling is quite similar to that seen in Figure 1, and that the addition of threads decreases execution time. This can be seen more clearly in Figures 6 and 7, where it shows approximately a 5x speedup for both a smaller and larger number of nodes.

We discuss the effects of caching shown in Figures 3 and 4 further in Section V.

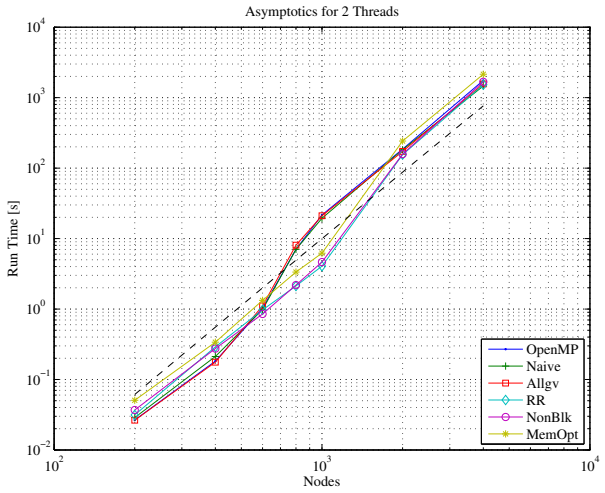

Fig. 3. Asymptotic behavior for 2 thread execution. The dotted black line is the function $10^{-9}x^3 \log_2 x$, which illustrates the asymptotic behavior.

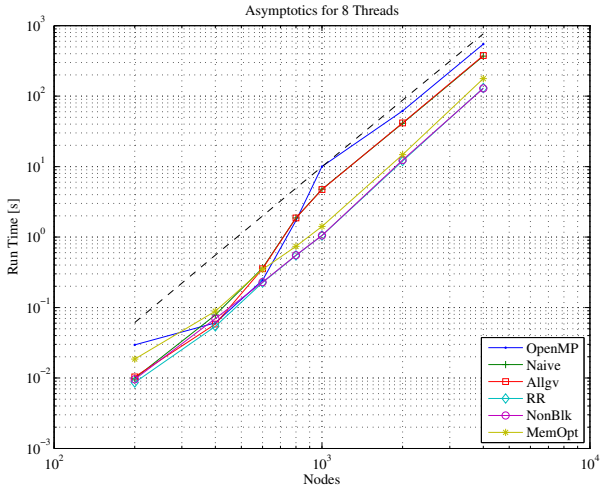

Fig. 4. Asymptotic behavior for 8 thread execution. The dotted black line is the function $10^{-9}x^3 \log_2 x$, which illustrates the asymptotic behavior.

IV. MPI-Based Floyd-Warshall Algorithm

A. Translating the OpenMP Code

A straightforward way to parallelize the shortest path algorithm using MPI is to maintain a copy of the original adjacency matrix in each processor, and distribute the recurrence operations amongst the p processors available. We call this version *Naive*. Each processor is assigned an equal number of contiguous columns in the adjacency matrix. If there are n nodes in the graph, each processor would be responsible for computing the shortest paths for $\lfloor n/p \rfloor$ columns of elements–we floor the operation due to integer truncation in the C programming language. Note that in this implementation, the task division is not fair, and loads the last processor when the number of nodes is not evenly divisible by the number of threads available in the system.

At the beginning of the program execution, we must broadcast the entire adjacency matrix from the master node to the other nodes in the system. This involves a broadcast of $(p-1)$ messages, each of length n^2 integer data elements. We used the `MPI_Bcast` collective operation to implement the broadcast of messages between processors. At the end of each iteration, each processor needs to broadcast the updated values of the columns it has been assigned,

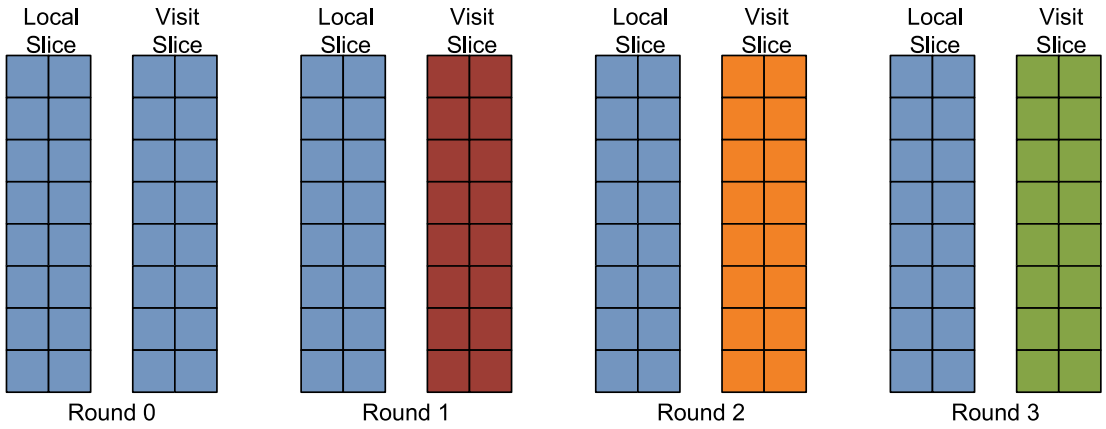

Fig. 5. Partitioning the stale matrix across the multiple processors in the *RR* implementation, as seen from the perspective of Processor 0. In each round, Processor 0 receives a slice of the previous iteration matrix from a neighboring core, and performs updates on its local slice.

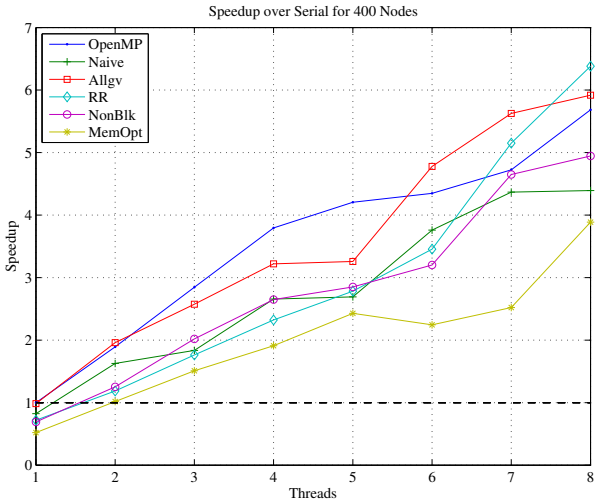

Fig. 6. Speedup over serial execution for 400 nodes.

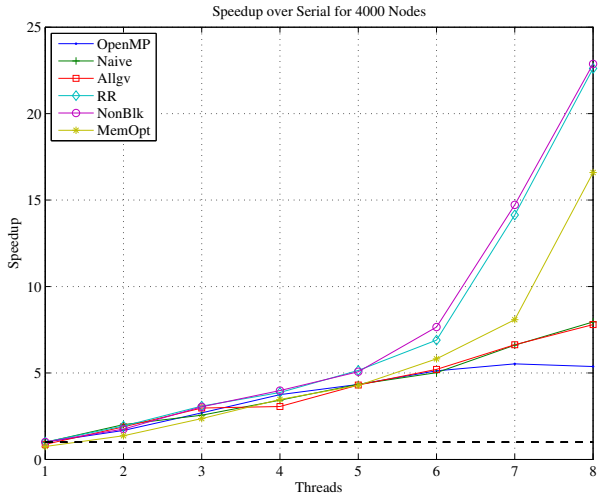

Fig. 7. Speedup over serial execution for 4000 nodes.

again using `MPI_Bcast` in a round-robin fashion. At the end of each iteration, there is also a `MPI_Allreduce` operation to perform the `done` flag aggregation.

We see in Figure 6 that, as we increase the number of nodes, *Naive* performs slightly worse than the *OpenMP* implementation. However, in Figure 7, we can see that *Naive* modestly outperforms *OpenMP* as we increase the number of threads. Figures 1, 3, and 4 show that, for various thread counts, the *Naive* implementation scales similarly to the *OpenMP* version. We expect that this is because we are trying to replicate the OpenMP communication style in MPI, which is not as well-suited to this type of communication pattern, as discussed in Section I.

B. Improving Per-Iteration Communication

Instead of using the `MPI_Bcast` call at the end of each iteration, we can replace this with an all-to-all gather operation to reduce latency. `MPI_Allgatherv` lets each processors gather data from all other processors, where each processor can send different amounts of data based on the task division. This *Allgv* implementation improves the time spent in communication of messages by overlapping the data movement between processors

As we can see in Figures 6 and 7, as we approach 8 threads, *Allgv* manages to outperform *OpenMP*. This is because our reduction in communication times eliminates some of the bottlenecks associated with the *Naive* implementation, which likely become magnified for smaller node sizes, due to the reduction in the computation:communication ratio.

V. A Memory-Scalable MPI Implementation

Unfortunately, both *Naive* and *Allgv* suffer from a lack of memory scalability, as each processor still maintains a full copy of the distance matrix for performing all the local computations. We can instead exploit logical ring topology amongst the processors.

Each processor maintains a local slice of the matrix, upon which it performs all calculations. Inside each recurrence iteration, an inner loop rotates a visiting piece of the matrix, which is owned and is being updated by a remote processor. Figure 5 shows how the slices are rotated from the perspective of Processor 0, assuming the initial slicing of Figure 2. For iteration $s+1$, each processor maintains three slices: (a) a stale copy of the slice that it owns from iteration s, (b) a second copy of the owned slice, which will be updated with new values for iteration $s+1$, and (c) a visiting stale slice from iteration s. This visiting stale slice is used to perform partial updates on the slice owned by the processor. Once these partial updates are finished, each processor sends the visiting slice to its neighbor in a logical 1D ring topology. The processor then receives a new visiting slice from its other neighbor, and repeats the partial update calculation until a complete rotation of visitors has been completed.

In this implementation, *RR*, each processor now only has a copy of $3n \times (n/p)$ elements. The all-to-all communication pattern observed is now reduced to localized `MPI_Sendrecv` operations between the neighboring nodes, removing the need to globally synchronize communication within an interval.

We see that in Figure 6, the performance of the algorithm is again below that of *OpenMP* for 400 nodes. However, Figure 7 is more representative of memory scaling, showing that for 4000 nodes, *RR* does significantly better than *OpenMP*, *Naive*, and *Allgv*. What this suggests is that, while the round-robin-based communication incurs a significant penalty when there is not a lot of work to be done, the reduction of large communications greatly increases the efficiency of our algorithm for large problem sizes.

However, the superlinear speedups have another cause, which can be better observed in Figure 3. While all previous implementations see a jump in performance when going from 600 to 800 nodes, *RR* does not experience such a jump until between 1000 and 2000 nodes. This is because, through the reduction of data stored in each processor, our memory footprint for higher node counts fits within the L1 caches. This effect can also be seen in Figure 4, where *RR* does not see any spike in its execution timing trend.

A. Removing Blocking Communications

While we greatly reduce the footprint of the matrix stored in each processor, we are still performing relatively fine-grained synchronization at the end of each round, which is currently not being overlapped by any computation. To alleviate this bottleneck, we create a new implementation, *NonBlk*, which removes the blocking `MPI_Sendrecv` calls and replaces them with the non-blocking commands `MPI_Isend` and `MPI_Irecv`. We then initiate this communication before our recurrence calculations, so we can transfer the visiting matrix to the next processor concurrently.

Figure 6 shows *NonBlk* slowing down compared to the *RR* implementation. One possible explanation for this is that `MPI_Sendrecv` is optimized to perform the send and receive operations concurrently, and so the overhead in calling two separate MPI functions in *NonBlk* is greater than the communication and computation for smaller node sizes. Figure 7 shows a speedup over *RR*, but one that is relatively small. This likely indicates that while overlapping communication with computation helps somewhat, there is a large imbalance between the time it takes for computation and the time it takes for communication. This means that one of them is a relatively insignificant portion of the code, and as a result, the overlap saves little time. We will investigate this further in Section VII.

B. Optimizing Memory Usage

For our final optimization, we take a look at memory usage. What we find is that, at every iteration, we are creating a large portion of memory inside Processor 0 that stays unused. It is possible that this large memory block creates unnecessary cache pollution, slowing down the core (and therefore overall execution). In order to do this, we parallelize the `gen_graph` function across all processor, with each processor allocating their slice of memory and generating the paths. Since the programs are all seeded identically, simply consuming random numbers until we arrive at our desired slice will allow us to maintain consistency with the matrices generated by previous implementations. We also eliminate the need to perform a `memcpy` after each `square` operation, by allocating two visit matrix buffers and swapping pointers each iteration.

We see that despite this expectation, this version of the code, *MemOpt*, does worse than *NonBlk* and even *RR*. While the serialized graph generation does impact Processor 0, it is not a part of the critical path, and its only effect is for possible reduction of cache collisions. However, since the initial n^2 block of memory is used once and never accessed again, it likely has little effect on the performance of the overall program, and does not impact the actual algorithm computation. We can also see in Figures 3 and 4 that the cache behavior noted for *RR* and *NonBlk* holds true for *RR* as well, indicating no major change in cache utilization.

Another impact could be an inadvertent effect of using pointer swapping as opposed to `memcpy`. Our arrays were originally declared using the `restrict` directive, which is a guarantee to the compiler that the accesses do not overlap with those of other pointers. However, by swapping the pointers inside the `shortest_paths` function, we are effectively violating this rule. It is possible that the compiler had to disable any unrolling optimizations that it had previously been able to implement because of the `restrict` keyword. This could have a potentially large impact on performance.

VI. Scalability

We study the scalability of our parallel MPI implementations by considering both strong and weak scaling.

A. Strong Scaling

Figure 8 and Figure 9 show the results for the strong scaling of our MPI implementations–i.e. scaling for a fixed global problem size of 400 and 4000 nodes respectively while varying the number of processors between 1 and 8. Figure 8 shows that for the smaller fixed problem size of 400 nodes, the *Naive* and *Allgv* implementations do not scale linearly beyond 5 threads. This can be attributed to the global communication costs, which overwhelm the computation time on each processor.

The *NonBlk* and *RR* implementations scale fairly well for increasing number of processors, due to improved localized communication patterns. *NonBlk* scales linearly up to 4 processors. Since each worker node on the cluster consists of a pair of quad-core processors (see Section II), the communication costs show up when we increase the number of processors beyond 4, at which point some of the communications will use the QPI. The *MemOpt* implementation scales well up to 5 processors, after which the it does not perform as well as the other round-robin implementations. We believe this is due to losing the benefits of the `restrict` keyword, as described in Section V-B.

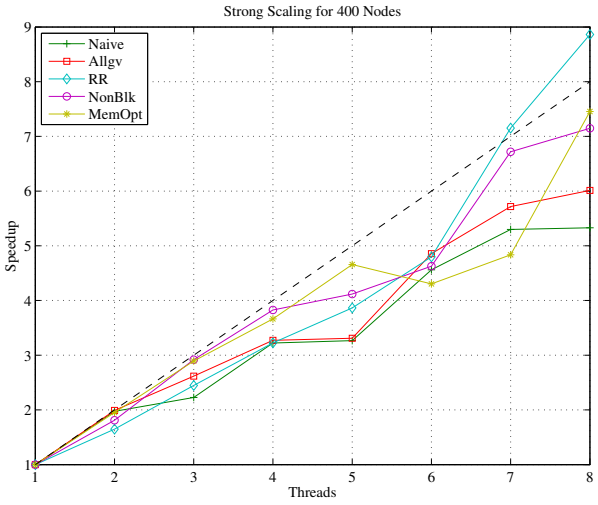

Fig. 8. Strong scaling study for 400 nodes. Ideal 1:1 scaling is plotted as a dashed black line.

For the larger problem size of 4000, Figure 9 shows superlinear scaling for the *MemOpt*, *NonBlk* and *RR* implementations, indicating the benefits of tuning the algorithm for memory scaling. For all three algorithms, each processor holds only its assigned columns. Thus, for increasing number of processors, the memory footprint is lower, and our data can reside in the L1 cache, as described in Section V. *MemOpt* still lags behind in performance, for the reasons described earlier.

In contrast, the *Naive* and *Allgv* scale linearly versus thread count, as the work per processor decreases linearly with the number of processors, as discussed in Section VII.

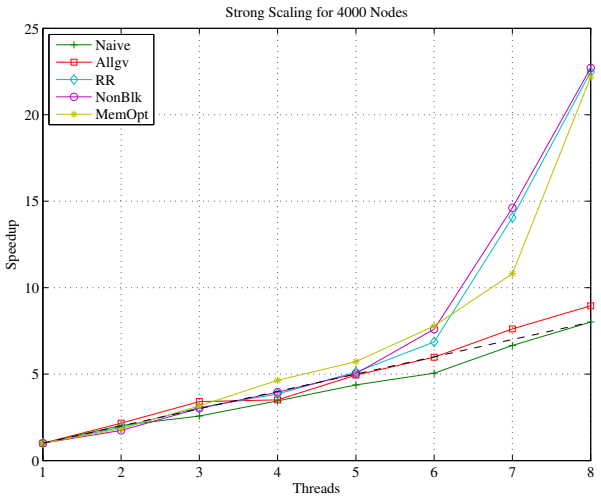

Fig. 9. Strong scaling study for 4000 nodes. Ideal 1:1 scaling is plotted as a dashed black line.

B. Weak Scaling

For the weak scaling studies, the y-axis of the graphs must be adjusted to account for the asymptotic increase in algorithm performance, as the amount of work per node must be kept constant. A linear increase in n, which represents the values swept in our data set, translates to a quadratic increase in the total amount of work n^2. As a result, we choose to calculate a corrected run time for the weak scaling figures, normalizing against a unit amount of work.

When we partition the algorithm across p threads, we find that our work is of order $O((n/p)n^2 \log n)$, as while we divide the amount of nodes to be updated per core by column, we must still visit all n other nodes within the matrix for a total of $\log n$ iterations. We select a fixed amount of work m for each thread as the basis of our weak scaling studies. If we assign m columns to each thread, we will have $n = mp$ nodes calculated globally. We can then use this to derive the total amount of work done in terms of m, as seen in Equation 2.

$$n^3 \log n = \frac{mp}{p}(mp)^2 \log mp \tag{2}$$

To determine how we normalize our plots, we first simplify Equation 2:

$$n^3 \log n = m^3 p^2 \log mp \tag{3}$$

Since we determined that the initial amount of work given is m, a single processor should have work on the order of $m^3 \log m$. We can divide Equation 3 by the amount of work done by a single processor to determine how much additional work each core is doing for larger values of n:

$$\frac{n^3 \log n}{m^3 \log m} = p^2 \left(1 + \frac{\log p}{\log m}\right) \tag{4}$$

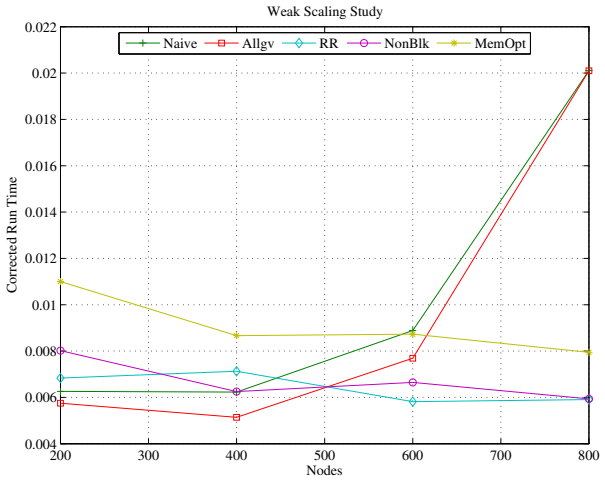

Fig. 10. Weak scaling study for 200 to 800 nodes, where the number of nodes per thread is a constant 100. Note that the y-axis has been adjusted for the $O(m^3 \log m)$ asymptotic behavior.

Therefore, if we divide each of our run times by Equation 4, we should expect to see a zero-slope behavior for ideal scaling.

Figure 10 shows the results for our weak scaling study. For this portion of our study, the problem size is increased from 200 to 800 nodes, keeping m at a fixed value of 100 nodes. *MemOpt*, *NonBlk* and *RR* scale well in this mode because of their balanced communication pattern. Each node in these algorithms only communicates with only their neighboring nodes, hence communication of messages can be masked by the computation time when increasing the number of processors.

Naive and *Allgv* do not exhibit flat-line scaling in the graphs, instead showing a rise in run time for increasing work. These implementations are largely communication-bound and consume more system resources, limiting weak scaling.

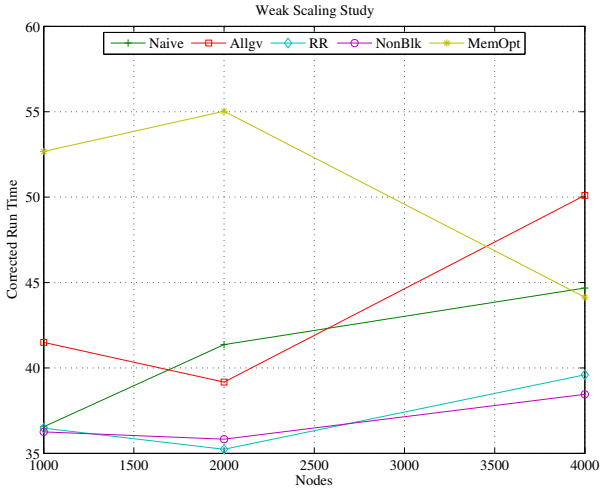

Fig. 11. Weak scaling study for 1000 to 4000 nodes, where the number of nodes per thread is a constant 1000. Note that the y-axis has been adjusted for the $O(m^3 \log m)$ asymptotic behavior.

Figure 11 shows the results for work increasing from 1000 to 4000 nodes, where m is fixed at 1000 nodes. The *MemOpt*, *RR*, and *NonBlk* implementations perform well at these larger sizes, and the curves are relatively flat, as expected. *Naive* and *Allgv* also exhibit better scaling compared to weak scaling at 100 nodes, as for the increased work

size, the computation time increases and masks the overhead incurred due to communication.

VII. Modeling Communication Overheads

As a rough, first-order model of execution time breakdown, we compared our MPI tunings against completely serial code–i.e. a code with no MPI or OpenMP calls. As discussed in Section I, the asymptotic behavior for our implementation of the Floyd-Warshall algorithm is $O(n^3 \log n)$, and as discussed in Section VI, the asymptotic behavior for each thread is $O((n/p)n^2 \log n)$, where p is the number of threads.

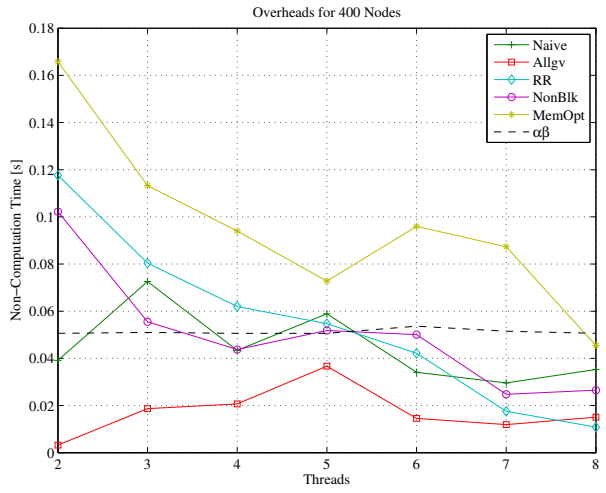

Fig. 12. Estimation of overheads in addition to computation. α-β model for communication is shown as a dashed black line.

Thus, to calculate the computation time for parallel threads, it is reasonable to divide the serial execution time by p. The overhead of non-computation tasks is the difference between the total parallel execution time and the estimated computation time, as shown in Equation 5.

$$t_{\text{overhead}} = t_{\text{parallel}} - \frac{1}{p} t_{\text{serial}} \tag{5}$$

As a point of comparison, we ran a ping-pong MPI test code to determine an α-β model for performance. Our values for α (base latency) and β (inverse bandwidth) were 78.78 μs and 36.54 ns/int, respectively. For each run of the algorithm, we can calculate a conservative estimate of the number of integers transferred between threads, as shown in Equation 6.

$$pn\left(\frac{n}{p} + n \mod p\right)\log_2 n \tag{6}$$

Using this model for communication as a conservative one, we can convert the number of integers communicated to time using the α and β values, the result of which is shown as the dashed black line in Figure 12. In an attempt to remove as much of the effects from caching as possible, we chose to plot the overheads for 400 nodes. Note this communication model does not account for the reduction on the `done` flag, nor does it account for any initial scatter, gather, or broadcast operations.

It is clear in the case of our *MemOpt* code that we are not accounting for some overheads in addition to the communication overheads. Our *Naive* MPI code best fits our α-β model, which is to be expected. However, we see that our *Allgv* code is performing better than our model. We suspect this has to do with the way the `MPI_Allgatherv` code is implemented, in that it does some communications in parallel. Thus, our model from Equation 6 is overly-conservative. We do not have sufficient data to conclusively explain the trends shown for our *RR* and *NonBlk*, but we believe that it is possible that the synchronization costs are hurting us more for low thread counts, i.e. because each processor has more work, the time spent waiting for another processor can be longer. In other words, the period of each processor's runtime function is longer, so there is more room for a phase shift. Another possible explanation is we are seeing some caching effects for 400 nodes, although this is unlikely.

VIII. Conclusion

We have shown our implementation of a modified Floyd-Warshall algorithm to solve the all-to-all shortest paths problem in $O(n^3 \log n)$ time using MPI. Through the use of non-blocking round-robin message passing and pointer swapping to eliminate unnecessary `memcpy` calls, we show super-linear speedup for a 4000 node graph problem, achieving a 23x speedup for 8 threads, a 4x improvement over our reference OpenMP implementation.

This speedup is a direct result of improved cache locality–for a problem size of 4000 nodes, dividing the graph data set amongst 8 threads allowed us to have better cache behavior than naively storing a copy of the entire graph for each thread. Any future work should include a detailed profiling of cache behavior.

References

[1] Intel Corp. "Intel Xeon Processor 5500 Series Specification Update." April 2011.