A Scalable Parallelization of All-Pairs Shortest Path Algorithm for a High Performance Cluster Environment

T. Srinivasan*, Balakrishnan R., Gangadharan S. A. and Hayawardh V
*Assistant Professor
Department of Computer Science and Engineering
Sri Venkateswara College of Engineering, Pennalur, Sriperambudur 602105, TN, India
tsrini1969@gmail.com, bsrealm@msn.com, {gangadharan, hayawardh}@gmail.com

Abstract

We present a parallelization of the Floyd-Warshall all pairs shortest path algorithm for a distributed environment. A lot of versions of the Floyd-Warshall algorithm have been proposed for a uniprocessor environment, optimizing cache performance and register usage. However, in a distributed environment, communication costs between nodes have to be taken into consideration. We present a novel algorithm, Phased Floyd-Warshall, for a distributed environment, which optimally overlaps computation and communication. Our algorithm is compared with a register optimized version of the blocked all pairs shortest path algorithm [6, 4, 1] which is adapted for a distributed environment. We report speedups of 2.8 in a 16-node cluster and 1.2 in a 32-node cluster for a matrix size of 4096.

1. Introduction

Supercomputers today are moving towards the paradigm of a large number of simple interconneced computers from the olden day mainframes. Distributed computing is sure to play a vital role in the future. Today, clusters of computers are used to solve huge number-crunching problems. The main challenge in effectively using a cluster of computers to solve a problem is in extracting the parallelism present in the problem. Developing parallel versions of basic algorithms like matrix multiplication and graph theoretical algorithms are of prime importance because they occur as part of bigger problems. The Floyd-Warshall algorithm [2, 7] is an algorithm for solving the all-pairs shortest path problem on weighted, directed graphs in cubic time. It is a very simple yet powerful algorithm. It has an abundance of uses in a large variety of fields, and is commonly found as a sub-problem in solving larger problems. Examples include transitive closure, finding a regular expression denoting the regular language accepted by a finite automaton, inversion of real matrices and optimal routing, among others. A lot of versions of the Floyd-Warshall algorithm have been proposed for a uniprocessor environment, optimizing cache performance and register usage.

Sahni et al [6] present a blocked version of the all-pairs shortest path algorithm, where they reorder computations to optimize cache performance. A tiled implementation with recursion to further optimize cache performance is presented in Park et al [4]. However, this blocked algorithm when adapted for a distributed environment accrues significant communication costs.

In this paper, we present Phased Floyd-Warshall, a novel algorithm for distributed environments, which optimally overlaps computation and communication.

We present experimental results comparing our algorithm with the blocked version. We report speedups of 2.8 in a 16-node cluster and 1.2 in a 32-node cluster for a matrix size of 4096.

The rest of the paper is organized as follows. Section 2 formally states the Floyd-Warshall all pairs shortest path algorithm, introduces the various parallel versions. Section 3 provides the motivation and major ideas behind our algorithm. Section 4 presents our algorithm. In section 5, we analyze the complexity of our algorithm, and compare it with the blocked version. Experimental results comparing our algorithm with the blocked algorithm is presented in section 6. We state our conclusions in section 7.

2. Preliminaries

2.1. The Floyd-Warshall Algorithm

We start with a brief description of the serial algorithm (Fig. 1).

Input: Adjacency matrix of n nodes a (with dimensions $n * n$), with $a_{i,j}$ representing weight of the direct path from

978-1-4244-1890-9/07/$25.00

```
01 for (int k = 1; k ≤ n; k++)
02    for (int j = 1; j ≤ n; j++)
03       for (int i = 1; i ≤ n; i++)
04          if (a_{i,j} > a_{i,k} + a_{k,j})
05                a_{i,j} = a_{i,k} + a_{k,j};
```

Figure 1. The Basic Serial Floyd-Warshall Algorithm

node i to node j.

Output: Matrix a (with dimensions $n * n$), with $a_{i,j}$ representing the shortest path between node i to node j, and ∞ if no such path exists.

In lines 6 and 7, we compare the current length of the path between i and j and the path from i to j through k. We then update the path length to be the minimum of those two paths. This is essentially a dynamic programming formulation.

The operation of checking the minimum and updating the path distance to that value is known as *relaxation*. In specific, lines 6, 7 shows the path between node i to node j being relaxed with respect to node k.

The working of the algorithm can be described as follows: at the end of the k^{th} iteration, the path from any node to any other node is relaxed with respect to the vertices $1, 2, \ldots, k$.

So, at the end of the n iterations, all nodes are relaxed with respect to nodes $1, 2, \ldots, n$. Hence, we have found the all-pairs shortest path.

Let us represent the path $a_{i,j}$ relaxed with respect to vertices $1, 2, \ldots, k$ by $d^k_{i,j}$. Hence, $d^0_{i,j}$ would represent the state of the initial input. In this representation, the relaxation operation becomes

$$d^k_{i,j} = min(d^{k-1}_{i,k}, d^{k-1}_{i,k} + d^{k-1}_{k,j}) \tag{1}$$

Hence, we can view the problem as the promotion of all elements from state 0 to state n. As can be easily observed, this basic serial algorithm runs in time $O(n^3)$.

It is important to note that though the three loops in the algorithm may look very similar to matrix multiplication, it is much more challenging to parallelize due to the extensive dependencies caused by the relaxation operation. The loop invariant is that at the start of the $(k+1)^{th}$ loop, all nodes are relaxed with respect to the nodes $1, 2, \ldots, k$. This eliminates an approach like divide and conquer

2.2. Blocked Version

Here, we describe the blocked version of Floyd Warshall [6, 4]. We begin by defining a few terms. The matrix is divided into *blocks*. Each block has a dimension of $b * b$. Hence, the entire matrix is partitioned into N^2 blocks, where $N = n/b$.

Let $A_{i,j}$ be the $(i,j)^{th}$ block.

$$A_{i,j} = \{a_{x,y} : (i-1)*b < x \leq i*b; (j-1)*b < y \leq j*b\} \tag{2}$$

In the k^{th} iteration:

- $A_{k,k}$ is the pivot block.
- Other blocks in the k^{th} row are the pivot row blocks.
- Other blocks in the k^{th} column are the pivot column blocks.
- All other blocks are non-pivot blocks.

Block $A_{i,j}$ when in state k has been relaxed with respect to nodes $1, 2, \ldots, b*k$. $D^k_{i,j}$ is used to represent the block $A_{i,j}$ in state k. It is similar to $d^k_{i,j}$, except that the latter is for a single element $a_{i,j}$ relaxed with respect to nodes $1, 2, \ldots, k$.

The dependencies arising from the blocking scheme are as follows:

- The pivot block requires only elements from itself to perform the current relaxation.
- The pivot row and column blocks require elements both from themselves and the pivot block to proceed with their relaxation.
- All other blocks require elements from two other blocks, apart from themselves, to perform relaxation.

We define the set $E^k_{i,j}$ as the set of external blocks in the appropriate states required to relax block $A_{i,j}$ during the k^{th} iteration.

$$E^k_{i,j} = \begin{cases} \emptyset; i = j = k \\ \{D^k_{i,k}\}; i = k, j \neq k \\ \{D^k_{k,j}\}; i \neq k, j = k \\ \{D^k_{i,k}, D^k_{k,j}\}; i \neq k, j \neq k \end{cases}$$

We define the relaxation operation which promotes the block $A_{i,j}$ from the state $k-1$ to state k using external block set $E^k_{i,j}$ as:

$$D^k_{i,j} = relax(D^{k-1}_{i,j}, E^k_{i,j}) \tag{3}$$

Hence, we can view the problem as the promotion of all blocks from state 0 to state N. The algorithm is in Fig. 2.

2.3. A Simple Parallel Version

We now move on to the straightforward parallel version. Each cluster node is assigned an equal number of contiguous rows in the adjacency matrix. Thus, if there are p cluster nodes and n rows, each node owns n/p rows.

The algorithm is as follows.

We note that this approach blocks at the beginning of every iteration for the broadcast to complete.

```
01  for (int k=1; k≤N; k++)
02  {
03    Relax the pivot block with respect to
    itself.
04    Relax the pivot row and pivot column blocks
05    Relax the non-pivot blocks.
06  }
```

Figure 2. Serial Blocked Version

```
01  for (int k = 1 k≤n; k++)
02  {
03    Processor owning the k^th row broadcasts the
    k^th row.
04    All processors relax their rows with respect
    to k.
05  }
```

Figure 3. Simple Parallel Version

2.4. A Parallel Blocked Version

We now state the adaptation of the blocked all pairs shortest path algorithm for a distributed environment, as in [1]. This follows directly from the basic blocked algorithm [6, 4]. The parallel blocked version provides greater performance than the simple parallel version by allowing more overlap of computation and communication. In this algorithm, if there are p cluster nodes, then block size $b = n/p$. Each cluster node owns one row of blocks.

```
01  for (int k=1; k ≤ N; k++)
02  {
03    <all processors have updated pivot block
    A_{k,k}>
04    P_k sends pivot row blocks to the other
    processors
05    Others receive and compute one pivot row
    block and their pivot column block
06    Others broadcast pivot row blocks in turn
07    Others relax their own blocks (their row)
    using received pivot row blocks
08    P_{k+1} sends P_k block A_{k+1,k+1} (next pivot block),
    which computes and broadcasts it for the next
    iteration
09  }
```

Figure 4. Parallel Blocked Version

In our discussions, for ease of understanding, we use serial numbers running from 1 to N (as opposed to using 0 to $N1$). Hence, if $g\%h$ is 0, it has to be taken as h. p is the total number of nodes and the r^{th} node is known as P_r.

3. Motivations for the Phased Algorithm

There are several inefficient aspects to deploying the blocked algorithm in a distributed environment (where communication costs are important), as opposed to a uniprocessor environment:

- In every iteration, the node holding the pivot has to distribute the pivot row amongst other nodes. Here, one sender has to send specific blocks to each and every receiver. Hence, the receivers will be idle until their turn arrives (receiver receives their respective block).
- The pivot node is highly underutilized and remains idle for most part of the iteration.
- There are as many broadcasts in an iteration as there are nodes.

We aim to propose an efficient algorithm for a distributed environment by using the following design goals:

- To complete the transfer of blocks required for the next iteration during the current iteration (in parallel with the ongoing processing work) to the maximum extent possible, so that there is no idle time in this regard. Specifically, we would want the data for the x^{th} iteration to be ready at the beginning of the $(x-1)^{th}$ iteration. This way, we can perform transmission of this data during the computational work in the $(x-1)^{th}$ iteration. If this is not achievable, it should at least be possible to transmit after performing some minimum computation work in the $(x-1)^{th}$ iteration. We achieve this by designing iterations such that no iteration requires data that is computed during the same iteration.
- To distribute evenly the workload amongst the nodes during a given iteration to the maximum extent possible. Our algorithm has loops where the amount of data worked on in every iteration is either increasing (phase 2) or decreasing (phase 1). We balance the workload by interleaving the iterations of different loops. This is detailed in a later section.
- To prefer a single large transmission as opposed to multiple small transmissions to the maximum extent possible. This is achieved by designing the algorithm such that pieces of work done by a node during previous iterations will represent all the external data required by (one or many) nodes during a particular iteration. Hence, the external data for iteration will arrive from a single source rather than parts arriving from multiple sources.

We now formally establish the states in which the blocks will be required as external blocks to segregate the work into modules efficiently.

Theorem 3.1. *In performing relaxations, we require the block $A_{a,b}$ in exactly two states, first when it is in state $min(a,b)$ and next when it is in state $max(a,b)$.*

Proof. Every block $A_{a,b}$ belongs to two sets of external block sets , E_1 and E_2, where $E_1 = \{E^a_{i,b}\}; 1 \leq i \leq N$ and $E_2 = \{E^b_{a,j}\}; 1 \leq j \leq N$.

Hence, $A_{a,b}$ will be used as an external block only in the corresponding relaxations. In E_1, $A_{a,b}$ is present in the state a, and in E_2 in state b. Thus, we require the block $A_{a,b}$ as an external block in exactly two states, once when it is in state a and once when it is in state b. Obviously, one of a, b is $min(a, b)$ and the other is $max(a, b)$. Since we can only sequentially promote $A_{a,b}$ from state 0 to state N, the state $min(a, b)$ becomes available before the state $max(a, b)$. □

We note that as a special case, when $a = b$, the two states merge into one. In the light of the above theorem, we partition the algorithm into three phases:

- Phase 1: Promotion of blocks $A_{a,b}$ from state 0 to state $min(a, b)$.
- Phase 2: Promotion of blocks $A_{a,b}$ from state $min(a, b)$ to state $max(a, b)$.
- Phase 3: Promotion of blocks $A_{a,b}$ from state $max(a, b)$ to state N.

For cases where $a = b$, we make the task of promoting the block to state a a part of the first phase. Thus, there is no work to be done on those blocks in the second phase.

4. Phased Floyd Warshall

4.1. The Algorithm

We first define sets that are used in the algorithm.

$$S1(x) = \begin{cases} A_{i,x}; x < i \leq N \\ A_{x,j}; x < j \leq N \end{cases}$$

$$S2(x) = \begin{cases} A_{i,x}; x \leq i < N \\ A_{x,j}; x \leq j < N \end{cases}$$

Thus, by definition, $S2(1)$ is $\emptyset$.

The brief algorithm for various phases is given in Fig. 5.

4.2. Proof of Correctness

4.2.1. Phase 1

Proof. We prove correctness by induction. Let x represent the current iteration number.

$x = 1$: Let us consider the iteration $x = 1$. $E^1_{1,1} = \emptyset$. Hence promotion of the pivot block $A_{1,1}$ from state 0 to state 1 does not violate dependencies. External blocks required for promoting blocks of $S1(1)$ from state 0 to state 1 is given by $\{E^1_{1,2} \cup E^1_{1,3} \cup \ldots E^1_{1,2}\} \cup \{E^1_{1,2} \cup E^1_{1,3} \cup \ldots E^1_{1,2}\}$, which is nothing but $E^1_{1,1}$.

This is available from the previous step. Hence the iteration with $x = 1$ has been shown to conform to dependency requirements.

$x = u$: Assuming that iteration with $x = u - 1, (u > 1)$ has completed, we now prove that iteration with $x = u$ can be carried out without violating dependencies. For performing the pivot block relaxation $A_{u,u}$, we need $E^1_{u,u} \cup E^2_{u,u} \cup \ldots \cup E^u_{u,u} = \{D^1_{u,1} \cup D^1_{1,u}\} \cup \{D^2_{u,2} \cup D^2_{2,u}\} \cup \ldots \cup \{D^{u-1}_{u,u-1} \cup D^{u-1}_{u-1,u}\}$. As can be seen, for $1 \leq y < u$, $E^y_{u,u}$ is available in $S1(y)$ and is in required state at the end of the iteration $x = y$. By hypothesis, those iterations have all completed. Hence the pivot block can be relaxed without violation of dependency requirements. Consider an element $A_{i,u}$ of $S1(u)$ to promote it from state 0 to u, the external blocks required are $E^1_{i,u} \cup E^2_{i,u} \cup \ldots \cup E^u_{i,u} = \{D^1_{i,1} \cup D^1_{1,u}\} \cup \{D^2_{i,2} \cup D^2_{2,u}\} \cup \ldots \cup \{D^{u-1}_{i,u-1} \cup D^{u-1}_{u-1,u}\} \cup D^u_{u,u}$.

As can be seen, for $1 \leq y < u$, $E^y_{i,u}$ is available in $S1(y)$ and is in required state at the end of iteration with $x = y$. $D^u_{u,u}$ is available from the previous step. Hence, dependencies of elements of the form $A_{i,u}$ of $S1(u)$ are satisfied.

Similarly dependencies are also satisfied for any element

```
Phase 1

01  for (int x=1;x≤N;x++)
02  {
03    promote A_{x,x} from state 0 to state x
04    promote S1(x) from state 0 to state x
05  }

Phase 2

01  for (int x=2;x≤N;x++)
02  {
03    promote S2(x) elements (A_{i,j}) from their
    current state min(i,j) to state x
04  }

Phase 3

01  for (int x=1;x≤N;x++)
02  {
03    for (int y=1; y≤N; y++)
04    {
05       promote A_{x,y} from state max(x,y) to N
06    }
07  }

Interleaved Phase 1 and Phase 2

01  for (int x=1;x≤N;x++)
02  {
03    if (x ≠ N)
04       perform iteration for this value of x for
    phase 1
05    if(x ≠ 1)
06       perform iteration for this value of x for
    phase 2
07  }
```

Figure 5. Short Algorithm for the Phases

from $S1(u)$ of form $A_{u,j}$. Hence it is possible to complete the iteration with $x = u$. □

4.2.2. Phase 2

Proof. As above, we prove correctness by induction. Let x represent the iteration number. $x = 2$: Let us consider the iteration $x = 2$. $S2(2) = A_{1,2}UA_{2,1}$. $min(i, j)$ for both $A_{1,2}$ and $A_{2,1}$ is 1. External blocks required to promote them from state 1 to state 2 is given by $E^2_{1,2} \cup E^2_{2,1} = D^2_{2,2}$.

$D^2_{2,2}$ is available as all blocks $A_{i,i}$ are in state i at the end of phase 1 iteration with $x = i$. Hence the iteration $x = 2$ been shown to conform to dependency requirements. $x = u$: Assuming that iteration with $x = u - 1, (u > 2)$ has completed, we now prove that the iteration with $x = u$ can be carried out without dependency violations. Consider an element of form $A_{i,u}$ of $S2(u)$. To promote it from state i to state u, the external blocks required are $E^{i+1}_{i,u} \cup E^{i+2}_{i,u} \cup \ldots \cup E^{u}_{i,u} = \{D^{i+1}_{i,i+1} \cup D^{i+1}_{i+1,u}\} \cup \{D^{i+2}_{i,i+2} \cup D^{i+2}_{i+2,u}\} \cup \ldots \cup \{D^{u-1}_{i,u-1} \cup D^{i}_{u-1,u-1}\} \cup D^{u}_{u,u}$.

As can be seen, for $i < y < u$, $D^y_{i,y}$ is available in $S2(y)$ and is in the required state at the end of iteration with $x = y$ of phase 2. $D^y_{y,u}$ is available in $S1(y)$ and is also in required state at the end of iteration $x = y$ in phase 1. $D^u_{u,u}$ is available in appropriate state at the end of iteration $x = u$ of phase 1. Hence, dependencies of elements of the form $A_{i,u}$ of $S2(u)$ are satisfied. Similarly dependencies are also satisfied for any element from $S2(u)$ of form $A_{u,j}$. Hence it is possible to complete the iteration with $x = u$. □

4.2.3. Phase 3

Proof. In phase 3, we promote all blocks $A_{i,j}$ from state $max(i, j)$ to N. In phase 3, the following external blocks are required to promote the block from state $max(i, j)$ to N:

$$E^{max(i,j)+1}_{i,j} \cup E^{max(i,j)+2}_{i,j} \cup \ldots \cup E^{N-1}_{i,j} \cup E^{N}_{i,j} =$$
$$\{D^{max(i,j)+1}_{i,max(i,j)+1} \cup D^{max(i,j)+1}_{max(i,j)+1,j}\} \cup$$
$$\{D^{max(i,j)+2}_{i,max(i,j)+2} \cup D^{max(i,j)+2}_{max(i,j)+2,j}\} \cup \ldots \cup$$
$$\{D^{N}_{i,N} \cup D^{N}_{N,j}\}$$

As can be seen, all external blocks $A_{i',j'}$ are required in state $max(i', j')$. As they are already in that state at the end of phase 2, it is possible to process blocks in phase 3 in any sequence. It is to be remembered that being in a state higher than required is permissible. □

4.2.4. Interleaving of phases 1 and 2

From the proof of correctness, it is clear that phase 1 iteration depends only on previous phase 1 iterations. Hence the interlacing will not affect phase 1 processing. From the proof of correctness, it is clear that a phase 2 iteration depends only on phase 2 iterations with $x < u$ and phase 1 iteration with $x = u$. As for a given iteration we perform phase 1 processing before phase 2, the dependencies are not violated.

4.3. The Distributed Algorithm

As stated earlier, the data distribution and task allocation is done in a manner so as to keep up the design goals as much as possible. The data distribution is done such that $S2(x) \cup A_{x,x}$ is present at node $x\%p$.

4.3.1. Phase 1

For every iteration $x = u$ of phase 1, a cluster node processes all blocks that it owns. The subset of $S1(x)$ owned by the r^{th} node is given by

$$S1(x) = \begin{cases} A_{i,x}; x < i \leq N, i\%p = r \\ A_{x,j}; x < j \leq N, j\%p = r \end{cases}$$

On computing the required external blocks, it can be seen that only non locally available external blocks for iteration $x = u$ are $S2(u) \cup A_{u,u}$. All these blocks are available in the node $u\%p$. However, the blocks will reach the required states only in the iteration $x = u$.To achieve the design goal of transmitting the external requirements of the $x = u$ iteration during the $x = u - 1$ iteration, a look ahead operation can be performed as shown in Fig. 6.

```
01  promote A_{x,x+1} and A_{x+1,x} from state 0 to state x
02  promote A_{x+1,x+1} from state 0 to state x+1
```

Figure 6. Look Ahead Operation

The correctness of performing this procedure at the beginning of the iteration $x = u - 1$ can be easily established by checking the sequence of operations against external block requirements. If this function were invoked at the beginning of the $x = u - 1$ iteration, $S2(u) \cup A_{u,u}$ will reach the state in which they are ready for transmission. Hence non local data required during iteration $x = u$ can be transmitted during iteration $x = u - 1$.

4.3.2. Phase 2

The phase 1 work load decreases with every iteration and the phase 2 work load increases with every iteration. Hence, in order to ensure that enough time is available for the broadcast of $S1(u) \cup A_{u,u}$ to complete during the iteration $x = u - 1$, we interleave iterations of phases 1 and 2. This has been proven possible in Section 4.2.4.

The broadcasts of $S2(u) \cup A_{u,u}$ during the $x = u-1$ iteration has made available the entire $S2(u)$ to all nodes in the up to date state. This makes it easy to distribute the workload of phase 2 such that all external blocks required are locally available and workload distribution is almost even. We define the subset of $S2(x)$ assigned to the node r for processing as:

$$S2(x) = \begin{cases} A_{i,x}; x \leq i < N, i\%p = r \\ A_{x,j}; x \leq j < N, j\%p = r \end{cases}$$

On checking the external block requirements, it can be seen that all are available locally in the appropriate state at the beginning of the iteration. The combined algorithms of phases 1 and 2 are stated in Fig. 7, 8.

```
01  /* get_start(a, b) returns first element e
    such that e > a and e % p = b */
02
03  void relax_phase1(int x, int r, int y_start)
04  {
05    for (int y=y_start;y≤N;y+=p)
06       for (int k=1;k≤x;k++)
07       {
08          relax (x, k, y);
09          relax (y, k, x);
10       }
11  }
12
13  void relax_phase1_priority (int x) /* Look
      ahead operation */
14  {
15    for (int k=1;k≤x;k++)
16    {
17       relax (x, k, x+1);
18       relax (x+1, k, x);
19    }
20    for (int k=1;k≤x+1;k++)
21       relax (x+1, k, x+1);
22  }
23
24  void relax_phase1_rest (int x)
25  {
26    relax_phase1 (x, (x+1)%p, x+1+p);
27  }
28
29  void relax_phase2 (int x, int r)
30  {
31    for (int y=r;y<x;y+=p)
32       for (int k=y+1;k≤x;k++)
33       {
34          relax(x, k, y);
35          relax(y, k, x);
36       }
37  }
```

Figure 7. Phase 1 and 2 - Distributed Algorithm

4.3.3. Phase 3

As has been established in the proof of correctness, we can process blocks in phase 3 in any order. We assign the

```
01  void phase1and2 (int r)
02  {
03    if (r == 0)
04    {
05       relax_phase1_priority(0);
06       bcast_async_begin(1);
07    }
08    else
09       bcast_recieve_async_begin(1);
10    wait_bcast(1);
11
12    for (int x=1;x≤N;x++)
13    {
14       if (x ≠ N)
15       {
16          if ((x+1)%p == r)
17          {
18             phase1_priority(x);
19             bcast_async_begin(S2(x+1) U
    Ax+1,x+1);
20             relax_phase1_rest (x);
21          }
22          else
23          {
24             bcast_recieve_async_begin((x+1)%p,
    S2(x+1) U Ax+1,x+1 ); /* sender, data */
25             relax_phase1(x,r,get_start(x,r));
26          }
27          relax_phase2(x,r);
28          wait(); /* Wait for broadcast to
    complete */
29       }
30       else
31       {
32          relax_phase1(x,r,get_start(x,r));
33          relax_phase2(x,r);
34       }
35    }
36  }
```

Figure 8. Phase 1 and 2 - Distributed Algorithm - Continued

processing of blocks in t^{th} row to cluster node $t\%p$. The lower triangle is the set of blocks $\{A_{i,j} : j < i \leq N\}$. Note that we do not include the diagonal blocks in the lower triangle. We define the subset of the lower triangle initially available with the r^{th} cluster node as: $LT_r = \{A_{i,j} : j < i \leq N; j\%p = r\}$. We now establish what processing can be done at a particular node when a particular subset of the lower triangle (which may come from another node) is locally available to it. Let us assume LT_s is available at the r^{th} cluster node. Based on external block requirements, it is seen that we can process elements of the set $S3(r, s)$, where $S3(x, y) = \{A_{i,j} : i\%p = x, j\%p = y\}$. Thus, all cluster nodes will need all lower triangle subsets to finish phase 3 processing of rows assigned to them. To make the entire lower triangle available to all cluster nodes, we transmit data in a ring-like manner. Formally, a cluster node P_r keeps receiving data from cluster node $P_{(r+1)\%p}$ and keeps sending data to cluster node $P_{(r-1)\%p}$. The sequence of processing for cluster node P_r in phase 3 proceeds as in Fig. 9:

```
01  for (int x=1;x≤p; x++)
02  {
03    If (x≠p)
04    {
05       initiate send of LT(r+x-1)%p to P(r-1)%p
06       initiate receive of LT(r+x)%p from
    P(r+1)%p
07    }
08    process S3(r,(r+x-1)%p)
09    If (x≠p)
10    {
11       wait for completion of transmissions
12    }
13  }
```

Figure 9. Overview of Phase 3 - Distributed Algorithm

5. Complexity Analysis

As in [1], we perform register level optimizations that provide a 2.5 fold speedup in relaxing non-pivot blocks in the blocked Floyd Warshall algorithm. This is made possible by making the k loop the innermost loop and thereby reducing the number of loads and stores. This same optimization can and has been applied in our algorithm for non-pivot blocks (that is, when the elements $a_{i,j}$, $a_{i,k}$ and $a_{k,j}$ all come from different blocks). Let:

- C_{B1} be the cost of relaxing one element in a pivot block, pivot row block or pivot column block.
- C_{B2} be the cost of relaxing one element in a non-pivot block.
- B_l be the broadcast latency and B_b be the inverse broadcast bandwidth.
- S_l be the point to point latency and S_b be the inverse point to point bandwidth.

For simplicity, we have assumed that b is picked such that N is a multiple of p and derived the complexity. After adding computation and communication costs for all phases, the overall complexity (computation and communication cost) for the r^{th} cluster node comes out to be

$$
\begin{aligned}
&totalcostphased(n,p,b,r) = \\
&\left(\frac{n}{12pb}\right)(12n^2 - 21nb - 6rnb + 3pnb + 3p^2b^2 + 3pb^2 \\
&- 18prb^2 - 6rb^2 + 18r^2b^2 + 12b^2)C_{B2} + \\
&\left(\frac{n}{p}\right)(2n-b)\,C_{B1} + \left(\frac{n}{b}\right)B_l + n^2B_b + (p-1)S_l + \\
&\left(\frac{n}{2p}\right)(np + 2rb - n - 2pb)\,S_b \qquad (4)
\end{aligned}
$$

```
01  void relax_phase3 (int r, int s)
02  {
03    /* starting point of k will vary depending
    on whether i > j or not.
04  Hence two loops. */
05    for (int i = r; i ≤ N; i += p)
06      for (int j = s; j ≤ i; j += p)
07        for (int k = i+1; k ≤ N; k++)
08          relax (i, k, j)
09    for (int i = r; i ≤ N; i += p)
10      for (int j=get_start(i,s); j ≤ N; j += p)
11        for (int k = j+1; k ≤ N; k++)
12          relax (i, k, j)
13  }
14
15  void phase3 (int r)
16  {
17    int next = (r+1)%p;
18    int prev = (r-1)%p;
19    lt curr, temp; /* lt is a data structure to
    hold LTr */
20    curr.init_local (); /* Initialise with ltr
    which is locally available */
21    for (int x = 1; x ≤ p; x++)
22    {
23      if (x ≠ p)
24      {
25        send_async_begin (prev, curr);
      /* receiver, data */
26        receive_async_begin (next,temp);
      /*sender, data */
27        relax_phase3( r, (r + x - 1)%p );
28        wait (); /* Wait for transmissions to
    complete */
29      }
30      else
31        relax_phase3 ( r, (r + p - 1)%p );
32    }
33  }
```

Figure 10. Phase 3 - Distributed Algorithm

We now compare our algorithm with the optimized implementation [1] of parallel blocked Floyd-Warshall. In this implementation of blocked Floyd-Warshall, one cluster node owns an entire row of blocks. The algorithm has the following complexity:

$$
\begin{aligned}
&totalcostblocked(n,p,b) = \\
&\left(\frac{n^3}{p^3}\right)(2p-1)C_{B1} + \left(\frac{n^3}{p^3}\right)(p^2 - 2p - 1) * C_{B2} + \\
&p^2B_l + n^2B_b + p(p-1)S_l + \left(\frac{n^2}{p}\right)(p-1)S_b \qquad (5)
\end{aligned}
$$

We note that block size for the optimized parallel blocked Floyd-Warshall algorithm is a function of n and p ($b = n/p$) but is variable in the Phased Floyd-Warshall algorithm. Comparing the complexities of the two algorithms, we note the following:

- The total amount of point to point send data (S_b coefficient) in the proposed algorithm is lesser as compared to the blocked version.

- The number of point to point sends initiated (S_l coefficient) in the proposed algorithm is lesser as compared to the blocked version.
- The amount of broadcast data (B_b coefficient) is the same for both algorithms.
- The number of broadcasts initiated (B_l coefficient) is more for the Phased algorithm as compared to the blocked version (as usually, $n/b > p^2$).

Hence, the proposed version scores better in the number of point to point sends initiated and the total point to point data sent. The only factor in which the blocked version scores better is in the number of broadcasts initiated.

However, the complexity coefficients alone do not convey the complete picture. In the blocked version, during every iteration, each cluster node cannot complete its work until it has received the broadcasts of the pivot row blocks. In the Phased algorithm, all communication during an iteration (phases 1 and 2, or 3) is in preparation for the next iteration and therefore the current iteration need not block.

In one iteration of the blocked parallel Floyd-Warshall, each cluster node broadcasts its portion of the pivot row. Hence, there are multiple broadcasts happening in the same iteration. In the Phased algorithm, in the main loop of `phase1and2()` (Fig. 8), only one broadcast (of $S2(x+1) \cup \{A_{x+1,x+1}\}$) is taking place in the xth iteration. In every iteration of `phase3()` (Fig. 10), every cluster node is involved in exactly one point to point send and one point to point receive.

During every iteration of the blocked version, one cluster node is underutilized as it has to relax only one pivot row block and the pivot block for the next iteration. The load distribution is not that uneven in the Phased Floyd-Warshall Algorithm.

6. Experimental Results

Our experiments were carried out on a 32 node Beowulf cluster. Each machine was powered by a 200 MHz Pentium 2 processor with 128MB RAM. We used the LAM/MPI [3, 5] environment and the programs were written in C++. Duration was measured ignoring the time to distribute the input adjacency matrix and agglomerate the shortest path cost matrix.

We present the speedup (Fig. 11) of the phased FW with respect to the blocked FW algorithm. Due to the nature of distribution of work, some processors will have q and others $(q-1)$ blocks to work on. Hence, to minimize the difference in workload, we keep the block size b at a minimum. We have found $b = 4$ to work best in our experiments.

Non-blocking broadcast `MPI_Ibcast` is not implemented for i386 architecture. We have used `MPI_Bcast`.

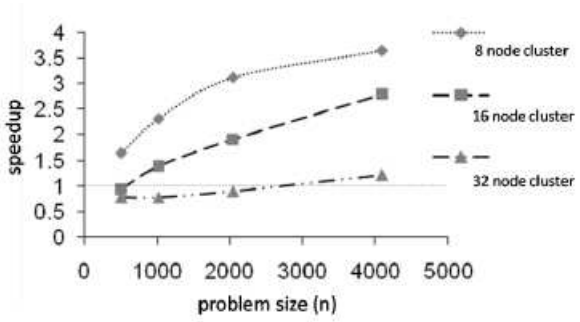

Figure 11. Speedup of Phased Floyd-Warshall Algorithm over the Blocked Algorithm

Hence, there is a blocking broadcast at the end of every iteration of `phase1and2()`, which could have been done in parallel with computation as per the algorithm. It is believed that there will be a significant performance improvement if non-blocking broadcast is made possible (it is currently possible in the IBM architecture).

It is seen that for any number of cluster nodes, the speedup exceeds 1 beyond a particular problem size.

7. Conclusion

We presented a new parallel algorithm for a distributed environment for the all pairs shortest path problem which aims to optimize the overlap between computation and communication and uniformly distribute computation load. A comparison with the parallel blocked algorithm shows speedups greater than 1.

References

[1] B. Diament and A. Ferencz. Comparison of parallel apsp algorithms.

[2] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[3] http://www.llnl.gov/computing/tutorials/mpi/. Message passing interface (mpi).

[4] M. Penner. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel Distrib. Syst.*, 15(9):769–782, 2004. Student Member-Joon-Sang Park and Fellow-Viktor K. Prasanna.

[5] J. M. Squyres and A. Lumsdaine. A component architecture for LAM/MPI. In *Proceedings, Euro PVM/MPI*, October 2003.

[6] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya. A blocked all-pairs shortest-path algorithm. In *SWAT '00: Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, pages 419–432, London, UK, 2000. Springer-Verlag.

[7] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.