AN ASYNCHRONOUS ALL PAIRS SHORTEST PATH ALGORITHM FOR MULITPROCESSORS

TECHNICAL REPORT NO. CENG 89-16

AYDIN URESIN   MICHEL DUBOIS

JUNE 1989

This work was performed under a grant from AT&T

An Asynchronous All Pairs Shortest Path Algorithm for Multiprocessors[†]

Aydin Uresin   Michel Dubois

Department of Electrical Engineering-Systems

University of Southern California

Los Angeles, CA 90089-0781

Abstract

Three asychronous versions of Floyd's algorithm for multiprocessors are presented. Their difference from already existing counterparts is that they require no barrier synchronization. Therefore, they perform better for large numbers of processors. In the last part of the paper, the technique used for removing synchronization on the all pairs shortest path problem is identified and generalized.

1- INTRODUCTION

The class of shortest-path problems is one of the most important ones in the area of transportation and communication networks. Uniprocessor algorithms for this class are well studied and documented [1] , [4]. There have also been many parallel shortest-path algorithms studied in the literature [5], [8], [3], [10], [9]. Most of these algorithms assume an SIMD computational model which is not compatible with most existing machines. Some of them are indeed for loosely-coupled [6] or tightly-coupled [5] [10] multiprocessors, but they still have the drawback that the extensive use of some form of synchronization primitives degrades their performance, especially when the number of processors is large. At synchronization points processors are blocked: they either relinquish control to the operating system, spin on a single shared variable, or wait for access to a global queue.

In this paper, we propose a variation of Floyd's algorithm for the all-pairs shortest path problem. In the proposed scheme, no explicit synchronization is necessary. The processors are not blocked during the execution of the algorithm and there is no serial bottleneck. Cooperation among processors is restricted to shared data accesses. The cost of the scheme is the extra shared memory space requirement and the penalty due to accesses to this extra memory. Although we have only considered Floyd's algorithm, the technique we used for removing synchronization is more general and we will introduce it in Section 7.

2- SYNCHRONIZED PARALLEL FLOYD'S ALGORITHM

Given a directed graph in which each arc has a nonnegative cost, the problem of all-pairs shortest path is to find for each ordered pair of vertices (i, j) the smallest length of any path from i to j, where the length of a path is defined as the sum of the costs of the arcs on that path [1].

Floyd's Algorithm is a popular one for the solution of this problem and a parallel version of it is given in Figure 1 [5]. In this algorithm, n is the number of nodes in the graph and initially $A[i,j]$

```
for $ROUND := 1$ to $n$ do
    forall $1 \leq i, j \leq n$ do
        $A[i,j] := \min\{A[i,j], A[i, ROUND] + A[ROUND, j]\}$
```

Figure 1: Synchronized parallel Floyd's algorithm

is the cost of the arc from i to j for all $i \neq j$; if there is no arc from i to j, $A[i,j]$ is set to ∞. Each diagonal element, $A[i,i]$, is set to 0. The keyword **forall** indicates that the $A[i,j]$'s are computed in parallel. To distinguish the shared writable variables from the other ones we adopt the convention of using uppercase names for such variables ($A, ROUND$). The read/only shared variables (such as n) can be considered as local variables, because each processor can keep a copy of these in its local store and can efficiently access them. In the synchronized Floyd's algorithm, at the end of the k-th iteration (round), for all i,j : (i) $A[i,j]$ is the length of a path from i to j , (ii) $A[i,j]$ is not greater than the length of any path from i to j passing through nodes labelled k or less. Therefore, after the execution of the loop, matrix A contains the lengths of the mimimum paths for each ordered pair.

In the synchronized parallel version of Floyd's algorithm the computations of all components of A must be synchronized after each iteration. Therefore, a barrier synchronization is needed after each iteration of the loop. A *barrier* is a logical point in the control flow of an algorithm at which *all* the processors must arrive before any of them are allowed to proceed further, and it is the most deleterious form of synchronization [2]. Let p be the number of processors such that

$p \leq n^2$. If we ignore the execution of the barrier, each round of the algorithm in Figure 1 takes $O(n^2 p^{-1})$ time. However, in [2] it has been shown that the barrier introduces significant overhead and a straightforward implementation of it, that uses a single semaphore, takes $O(p\, t'(p))$ time where $t'(p)$ is the time complexity for executing the semaphore. A faster scheme is also given in [2] which uses $O(p \log p)$ memory and takes $O(t'(p) \log p)$ time. Here, we assume the second implementation without further elaborating on its complexity. As a result, the synchronized parallel Floyd's algorithm takes $O(n\,(n^2 p^{-1} + t'(p) \log p))$ time. When p is $O(n^2)$ then the second term dominates and the complexity becomes $O(n\, t'(n^2) \log n)$.

3- SIMPLE ASYNCHRONOUS FLOYD'S ALGORITHM

To remove the barrier, we first observe that the single variable $ROUND$ in the above algorithm has the role of keeping track of the "progress" of the data A towards the solution. After all the components of A are updated once, one unit of progress has been made and the next "round" of computation is entered. In the first asynchronous version of the algorithm which is given in Figure 2, on the other hand, each processor only keeps track of the progress of the component it is assigned to. Therefore, instead of only one $ROUND$ number for the whole data, we have a $ROUND$ number for each individual component. The idea is as follows. Suppose that at a certain instance of the computations the component $A[i,j]$ is in the k-th round ($k = ROUND[i,j]$). After $A[i,j]$ is updated in this round, there is a unit progress in $A[i,j]$ towards the solution if its predecessors are at least in the same round. In this case, $ROUND[i,j]$ can be incremented. By predecessors, we mean the components on which $A[i,j]$ is dependent in the k-th round, i.e., $A[i,j], A[i,k+1], A[k+1,j]$.

Before further elaborating on the above idea, we need to describe the model of computations.

```
    $task(i,j)$;
        begin {$task$}
S1:         $k\_old := ROUND[i,j]$ ;
S2:         $k := k\_old + 1$ ;
S3:         $ki := ROUND[i,k]$ ; $kj := ROUND[k,j]$ ;
S4:         $A[i,j] := \min\{A[i,j]\ ,\ A[i,k] + A[k,j]\}$ ;
S5:         if ($k\_old \leq ki$) and ($k\_old \leq kj$) then
                begin
                    $ROUND[i,j] := k$ ;
                    if ($k = n$) then terminate
                end ;
            return
        end {$task$}
```

Figure 2: Simple asynchronous Floyd's algorithm

Figure 2 shows the program of the task that corresponds to a single update of the component $A[i,j]$. In our point of view, a task is an indivisible unit of execution that is held by the same processor from its start until the statement **return** or **terminate** is reached. There is a pool of tasks in the system: initially n^2 of them, one for each component. There are also a number of processors, and each available processor selects a task from the pool and executes it until the **return** statement. At this point, it returns the task to the pool and makes another task selection, or possibly continues with the same task. The execution of the **terminate** statement in $task(i,j)$ discards this task from the system for good, because $A[i,j]$ has reached the solution. It should also be mentioned that $ROUND[i,j]$ is initially assigned to 0, for all i,j. We also assume a computational system that supports atomic accesses (e.g. by hardware) to individual components of $ROUND$ and A. If not, accesses to $ROUND$ and A should be performed in critical sections.

The implementation of the task allocation policy that determines the selection rule is not made explicit here since a variety of schemes may be adopted. We only make two assumptions on the allocation. The first one is that the processors have exclusive hold of the tasks, i.e., at a given time instance $task(i,j)$ can be executed by at most one processor. This restriction is natural when each task is assigned statically to a fixed processor, throughout the computation. Static allocation has the advantage that it does not require an explicit task pool in the system: each processor repeatedly executes the task in its set with no need to access a global pool. If dynamic allocation is adopted, the exclusive hold condition should be enforced by some means. Another condition is that the allocation of tasks to processors is fair. In other words, in a finite period of time all tasks execute their code at least once. We will refer to these assumptions as the *exclusive hold* and *fairness* conditions.

The proof of correctness of the above described algorithm is based on the truth of the following property throughout the execution of the algorithm. Let $ROUND_t[i,j]$ and $A_t[i,j]$ be the values of $ROUND[i,j]$ and $A[i,j]$ at time t.

Property 1 *$A_t[i,j]$ is the length of a path from i to j which is not greater than the length of any path passing through the nodes labelled $ROUND_t[i,j]$ or less.*

Thus, $A_t[i,j]$ is the length of a minimum path when $ROUND_t[i,j] = n$. The correctness immediately follows from the following facts :

(1) At any time instance t during the execution of the algorithm Property 1 holds for all i,j,

(2) $ROUND[i,j]$'s are nondecreasing,

(3) For all i,j and t there exists a finite period of time Δt such that $ROUND_{t+\Delta t}[i,j]$ is greater than $ROUND_t[i,j]$.

We can show (1) by induction. It is obvious that initially Property 1 holds for all i,j (Recall that initially $ROUND[i,j] = 0$ for all i,j). After the initial state, the only events that would affect the truth of (1) are the executions of statements S4 and S5 in Figure 2. Let us assume that Property 1 holds just before the execution of S4. Then, it will obviously be satisfied after the execution of S4, as well, because the new value of $A[i,j]$ will not be greater than the old value. Now, let the old value of A be A_o and consider the time instance just before the execution of S4. If $ki \geq k\_old$, then $A_o[i,k]$ is not greater than the length of any path passing through nodes labelled k_old or less. The reason is that by induction hypothesis this is true at the time when ki is fetched and even if $A[i,k]$ might have been changed since then, this change can only decrease $A[i,k]$ without affecting the truth of the above statement. Similarly, if $kj \geq k\_old$, then $A_o[k,j]$ is not greater than the length of any path passing through nodes labelled k_old or less. Therefore, $A_o[i,k] + A_o[k,j]$ is smaller than or equal to the length of any path passing through k once and not passing through nodes with a label larger than k. Consequently, $\min\{A_o[i,j], A_o[i,k] + A_o[k,j]\}$ is the length of a path which is not greater than the length of any path passing through nodes labelled k or less, and Property 1 holds just after the execution of S5.

(2) is obvious from statement S5 which is the only statement that updates $ROUND[i,j]$. (3) is not as obvious. Let l be the minimum of all the components of $ROUND$ at time t and i', j' be the coordinates of this minimum, i.e., $ROUND[i',j'] = l$. From the fairness assumption in a finite period of time $task(i',j')$ will execute its code. During this execution $k\_old = l$ and therefore the condition of S5 is satisfied and $ROUND[i',j']$ is incremented. This means that $\min\{ROUND[i,j]\}$ is increased in a finite period of time and all the components are increased in a finite period of time.

The time complexity of this asynchronous Floyd's algorithm generally depends on the task allocation scheme, which affects the number of executions of $task(x,y)$, although the number of useful executions of $task(x,y)$ in which $ROUND[x,y]$ is incremented is always n. However, unless the load balance is very poor, we can assume that the ratio between the useful and the wasted task executions is a constant independent of n and p. Then, each task is executed $O(n)$ times and between two consecutive executions of a certain task, there are $O(n^2 p^{-1})$ task executions by a single processor where p is the number of processors which is less than n^2. Section 6 discusses how the task allocation policy can reduce the number of executions of each task. If accessing a shared variable takes $O(t(p))$ time, then a single execution of a task takes $O(t(p))$ time. Also assume an allocation scheme that does not take more than $O(t(p))$ to initiate the next task. Then the time complexity of the algorithm is $O(n^3 p^{-1} t(p))$. In the case when we have sufficient number of processors such that p is $O(n^2)$, the complexity will be $O(n\, t(n^2))$, which is less than the synchronized case. Note that the time complexity of executing a semaphore ($O(t'(p))$) is at least as large as the time complexity of accessing a shared variable ($O(t(p))$).

4- ASYNCHRONOUS FLOYD'S ALGORITHM FOR PARTITIONED DATA

The synchronized Floyd's algorithm and the asynchronous Floyd's algorithm in the previous section are the two extreme cases with respect to the size of the $ROUND$ data. As mentioned before, in the synchronized case only one $ROUND$ number is maintained for the whole matrix A and in the simple asynchronous case, there is a $ROUND$ number for each component. Using an additional memory for each component of A may be considered a waste, especially when there are less processors available than the number of components and therefore partitioning is necessary. Consequently, in the case of partitioning, we can utilize a $ROUND$ number for each partition, instead of each component, and the result is the algorithm in Figure 3.

Here, the rows and the columns of matrix A are partitioned and a column partition x together with a row partition y defines a submatrix of A, i.e., a rectangular domain of indices which is denoted by $s[x, y]$ (Figure 4). $task(x, y)$ corresponds to a single update of the whole domain $s[x, y]$ instead of a single component update. The column partition that contains the k-th column is denoted by $Col\_Partition(k)$ and similarly the row partition that contains the k-th row is denoted by $Row\_Partition(k)$. The predecence relationships which existed among the individual components in the previous algorithm are now among the partitions.

The relationship between A and $ROUND$ is stated by Property 2 below, which is analogous to Property 1.

Property 2 *$A_t[i, j]$ is the length of a path from i to j which is not greater than the length of any*

```
    $task(x, y)$;
        begin {$task$}
S1:         $k\_old := ROUND[x, y]$ ;
S2:         $k := k\_old + 1$ ;
S3:         $zx := Col\_Partition(k)$ ; $zy := Row\_Partition(k)$ ;
S4:         $kx := ROUND[x, zy]$ ; $ky := ROUND[zx, y]$ ;
S5:         for $(i, j) \in s[x, y]$ do
                $A[i, j] := \min\{A[i, j], A[i, k] + A[k, j]\}$ ;
S6:         if $(k\_old \le kx)$ and $(k\_old \le ky)$ then
                begin
                    $ROUND[x, y] := k$ ;
                    if $(k = n)$ then terminate
                end
S7:         return
        end {$task$} ;
```

Figure 3: Asynchronous Floyd's algorithm for partitioned data

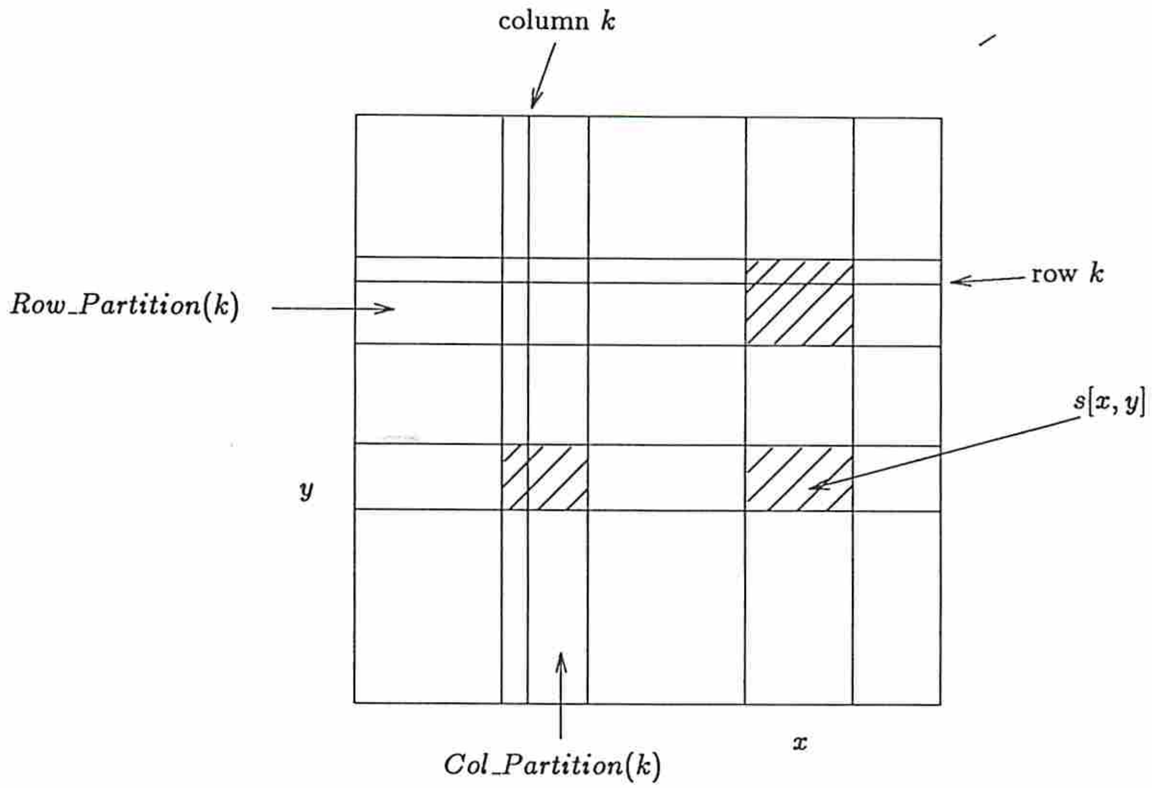

Figure 4: Partitioning A

path passing through the nodes labelled $ROUND_t[x, y]$ or less, for all $(i, j) \in s[x, y]$.

The argument to show that the elements of $ROUND$ eventually increase in a finite period of time till the solution, is the same as in the previous case; therefore, we will not repeat it here. The proof that Property 2 holds for all i, j and t is also similar to the proof of Property 1. We only have to notice that the only events that could affect the truth of Property 2 are the executions of S5 or S6. As a result, after $task(x, y)$ terminates the value of $ROUND[x, y]$ is n and from Property 2 $A[x, y]$ contains the solution.

As in the previous case, the time complexity of this version can be obtained easily as $O(n^3 p^{-1} t(p))$. Although the complexity is the same, the size of $ROUND$, in this version, is smaller. Also, this version is faster due to the less frequent accesses to $ROUND$.

5- EARLY UPDATING OF *ROUND*

Notice that in the previous algorithm for partitioned data, $ROUND[x, y]$ is fetched at the beginning of $task(x, y)$ and it is updated at the end. This may cause the execution to slow down unnecessarily. To see this consider the following scenario. Consider the first executions of the tasks $task(x, y)$ and $task(x', y')$. $task(x, y)$ updates its partition and then updates $ROUND[x, y](= 1)$. After this, it starts the second execution: it fetches $ROUND[x, y](= 1)$ and later $ROUND[x, zy](= 0)$ (at S4). Let $y' = zy$. If $task(x, y')$ is slower such that it updates $ROUND[x, zy]$ in the first execution after $task(x, y)$ fetches it in the second execution, then in the second execution of $task(x, y)$ $ROUND[x, y]$ will not be incremented, because $task(x, y)$ will still see $ROUND[x, zy] = 0$. In general, the worst case is that a task increments its $ROUND$ component in one of the two consecutive executions.

One of the ways to reduce the slowdown due to the above effect is to update $ROUND[x, y]$ as early as possible in $task(x, y)$. This is done in the algorithm in Figure 5.

The only difference between this version and the previous one is that $task(x, y)$ updates $ROUND[x, y]$ prematurely: $task(x, y)$ first updates the points in $s[x, y]$ either coordinate of which is $k\_next = ROUND_t[x, y] + 2$, then $ROUND[x, y]$ is updated before the remaining components in $s[x, y]$ are processed.

Property 3 *$A_t[i, j]$ is the length of a path which is not greater than the length of any path passing through the nodes labelled $ROUND_t[x, y]$ or less, for all i, j such that*

- *$(i, j) \in s[x, y]$, and*
- *$i = ROUND_t[x, y] + 1$ or $j = ROUND_t[x, y] + 1$.*

Since the only different part of the algorithm is between S7 and S9, the only possible cause of incorrect behavior of the algorithm may be this part. Particularly, the only reason the algorithm would not be correct could be as follows. $task(x, y)$ might update $ROUND[x, y]$ and another $task(\overline{x}, \overline{y})$ might fetch this premature data before $task(x, y)$ finishes the execution of S8, and then it may use this premature data to produce incorrect data. On the other hand, $task(\overline{x}, \overline{y})$ can only fetch $ROUND[x, y]$ at S5. Therefore, after S5 of $task(\overline{x}, \overline{y})$, either $\overline{kx} = ROUND[x, y]$ or $\overline{ky} = ROUND[x, y]$. Without loss of generality, we assume the first case; from symmetry the argument for the second case is the same. $\overline{kx}$ is used only in S7 and since the difference between the premature and the old values of $ROUND[x, y]$ is 1, the correctness may be affected only when $\overline{k\_old} = \overline{kx}$, therefore $\overline{k} = ROUND[x, y] + 1$. On the other hand, the only elements of A that $task(\overline{x}, \overline{y})$ uses as inputs are the ones either coordinate of which is $\overline{k} = ROUND[x, y] + 1$. Property 3 states that Property 2 is satisfied for such elements. Therefore, the early update of $ROUND[x, y]$ does not make the algorithm incorrect.

```
     $task(x, y)$;
         begin {$task$}
S1:          $k\_old := ROUND[x, y]$ ;
S2:          $k := k\_old + 1$ ;
S3:          $k\_next := k + 1$ ;
S4:          $zx := Partition\_x(k)$ ; $zy := Partition\_y(k)$ ;
S5:          $kx := ROUND[x, zy]$ ; $ky := ROUND[zx, y]$ ;
S6:          for $((i, j) \in s[x, y]$ and $(i = k\_next$ or $j = k\_next))$ do
                 $A[i, j] := \min\{A[i, j], A[i, k] + A[k, j]\}$ ;
S7:          if $(k\_old \leq kx$ and $k\_old \leq ky)$ then
                 $ROUND[x, y] := k$ ;
S8:          for $((i, j) \in s[x, y]$ and $i \neq k\_next$ and $j \neq k\_next$ do
                 $A[i, j] := \min\{A[i, j], A[i, k] + A[k, j]\}$ ;
S9:          if $ROUND[x, y] = n$ then terminate else return ;
         end {$task$}
```

Figure 5: Early updating of $ROUND$

Obviously, the complexity of this version of the asynchronous algorithm is the same as the previous one. However, since the processors make the $ROUND$ data available to other processors earlier than in the previous version, the components of $ROUND$ are incremented more frequently. Therefore, this algorithm executes faster.

6- TASK ALLOCATION

In the above sections we have shown that the asynchronous Floyd's algorithm is always correct when the exclusive hold and fairness conditions are satisfied. We also mentioned that the complexity of the algorithm is the same for any task allocation scheme as long as each task is executed $O(n)$ times. However, it is clear that the actual execution time of the algorithm is directly related to the number of executions of the tasks. Two possible allocation schemes are as follows:

1. Static allocation: the set of tasks is partitioned into disjoint subsets and each subset is preassigned to a processor. Each processor executes the tasks in its subset in a round-robin fashion.

2. Dynamic allocation: a given task is not executed by the same processor throughout its lifetime.

Static allocation has the advantage that it automatically satisfies the exclusive hold assumption, therefore no extra overhead is necessary to enforce it. Its disadvantage is that if the processors are not homogeneous or if the size of each partition cannot be made equal, slower tasks determine the speed of the computations. Through dynamic allocation, on the other hand, a better distribution of the total useful work among the processors is possible. In other words, efficiency of the processors can be increased, and therefore, the execution time can be reduced. However, exclusive hold condition must be enforced by some means. One way to accomplish this is to maintain a global queue of tasks and let the available processors choose tasks for execution, according to a selection rule. Ignoring the overhead associated with the implementation of the allocation, the selection rule is optimum, i.e., the execution time is minimum, when each available processor selects the task with minimum $ROUND$ value. This simply follows from the fact that the condition of incrementing the $ROUND$ number of a task (S5 in Figure 2, S6 in Figure 3 and S7 in Figure 5) is always satisfied, in this case. Therefore, each task increments its $ROUND$ value, when executed, and is executed exactly n times, which is the minimum number. However, strictly enforcing this rule introduces an unnecessary overhead, because the minimum finding procedure adds to the total complexity. In any case, the idea gives sufficient insight to propose the following scheme. An additional processor is dedicated solely to check the queue and to keep it sorted with respect to $ROUND$ values as best as possible. Even though this does not guarantee that each task increments its $ROUND$ value, the chances are good. Therefore, wasted task executions are reduced.

A third allocation scheme that can be adopted is intermediate between the static and dynamic allocations. In this scheme most of the tasks are partitioned into equal sized subsets and the remaining tasks are allowed to float among the processors.

The most significant advantage of the asynchronous algorithm is that it has no barrier at which the processors have to wait idle. Barrier synchronization is an important cause of performance degradation, especially when the number of processors is high. On the other hand, the asynchronous algorithm requires extra memory space for $ROUND$ and extra overhead for accessing the elements of $ROUND$. Extra contention for accessing $ROUND$ can be reduced by placing the same components of $ROUND$ and A at consecutive locations in the shared memory and accessing them together. Also, there is no serial bottleneck such as accessing a single shared variable.

```
for $k := 1$ to $m$ do
    forall $1 \leq i \leq n$ do
        $X[i] := Fi(X, k)$
```

Figure 6: A synchronized loop

7- GENERALIZATION OF THE TECHNIQUE

The above described technique of mapping a synchronized loop to an asynchronous one can readily be applied to transitive closure problems [1] and to the minimum spanning tree algorithm given in [7]. Furthermore, the technique can be generalized. The generalization follows from the observation that at a certain time instance, $ROUND[i,j]$ contains information about the range of values that $A[i,j]$ can take at this instance. In other words, $A[i,j]$ takes values from the domain that contains all the possible values of path lengths that are not greater than the length of any path passing through $1, 2, \ldots, ROUND[i,j]$. As $ROUND[i,j]$'s increase these domains shrink and finally when $ROUND[i,j] = n$ the domain contains a single element, the solution.

Suppose, we are given a synchronized loop as in Figure 6. Let there exist $m + 1$ domains, $D(0), D(1), \ldots, D(m)$, such that the operator $F = F1 \times F2 \times \cdots \times Fn$ and $D(k)$ satisfies the following conditions:

[C1] $D(k) = D1(k) \times D2(k) \times \cdots \times Dn(k)$,

[C2] $X_0 \in D(0)$ (the initial value of X),

[C3] $D(k+1) \subseteq D(k)$,

```
$task(i)$ ;
    begin $\{task\}$
        $k := \min\{ROUND[j]\}$ ;
        $X[i] := Fi(X, k)$ ;
        $ROUND[i] := k + 1$;
        if ($ROUND[i] = m$) then terminate else return;
    end $\{task\}$
```

Figure 7: Asynchronous loop

[C4] $D(m) = \{\xi\}$,

[C5] $Y \in D(k) \;\Rightarrow\; F(Y, k) \in D(k+1)$,

From the above conditions it is easy to see that the result of the loop in Figure 6, i.e., the contents of X at the end of the execution is ξ. By using a similar argument as in Section 3 we can show that the result of the asynchronous loop in Figure 7 is $\xi[i]$. There are n tasks executing asynchronously and they satisfy exclusive hold and fairness assumptions. $ROUND$ values are initially 0. Notice that,

1. At all times, and for all i, $X[i] \in Di(ROUND[i])$.

2. Therefore, at all times, $X \in D(\min\{ROUND[i]\})$,

8- CONCLUSION

As we approach the era of massively parallel computing, overheads due to the interaction of processors such as synchronization and memory contention appear as important factors in the design and analysis of algorithms. Motivated by this fact, we have presented three versions of an asynchronous all pairs shortest path algorithm for shared-memory multiprocessor systems, free of barrier synchronization. Since, no barrier synchronization is involved and memory contention is minimized in these algorithms they perform better than previously announced ones, especially for a large number of processors. We showed the performance improvement by a simple complexity analysis. We finally generalized the idea used in the design of the asynchronous all pairs shortest path algorithms.

References

[1] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] AXELROD, T. S. Effects of synchronization barriers on multiprocessor performance. *Parallel Computing 3* (1986), 129–140.

[3] DEKEL, E., D.NASSIMI, AND SAHNI, S. Parallel matrix and graph algorithms. *SIAM J. Computing* (1981), 657–675.

[4] DEO, N., AND C.PANG. Shortest path algorithms: taxonomy and annotation. *Networks* (1984), 275–323.

[5] DEO, N., PANG, C., AND LORD, R. Two parallel algorithms for shortest path problems. In *IEEE International Conference on Parallel Processing* (1980), pp. 244–253.

[6] JENQ, J., AND SAHNI, S. All pairs shortest paths on a hypercube multiprocessor. In *International Conference on Parallel Processing* (1987), pp. 713–716.

[7] MAGGS, B., AND PLOTKIN, S. Minimum cost spanning tree as a path finding problem. *Inf. Proc. Lett. 26* (1988), 291–293.

[8] PAIGE, R., AND KRUSKAL, C. Parallel algorithms for shortest path problems. In *IEEE International Conference on Parallel Processing* (1985), pp. 14–20.

[9] QUINN, M. J. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[10] QUINN, M. J., AND YOO, Y. B. Data structures for the efficient solution of the graph theoretic problems on tightly-coupled MIMD computers. In *IEEE International Conference on Parallel Processing* (1984), pp. 431–438.