Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem*

Vipin Kumar
Computer Science Department
University of Minnesota
Minneapolis, MN 55455
Internet: kumar@cs.umn.edu

Vineet Singh
MCC
3500 West Balcones Center Drive
Austin, Texas 78759
Internet: vsingh@mcc.com

March 21, 1991

Abstract

This paper uses the *isoefficiency* metric to analyze the scalability of several parallel algorithms for finding shortest paths between all pairs of nodes in a densely connected graph. Parallel algorithms analyzed in this paper have either been previously presented elsewhere or are small variations of them. Scalability is analyzed with respect to mesh, hypercube and shared-memory architectures. We demonstrate that isoefficiency functions are a compact and useful predictor of performance. In fact, previous comparative predictions of some of the algorithms based on experimental results are shown to be incorrect whereas isoefficiency functions predict correctly. We find the classic tradeoffs of hardware cost vs. time and memory vs. time to be represented here as tradeoffs of hardware cost vs. scalability and memory vs. scalability.

*This work was partially supported by Army Research Office grant # 28408-MA-SDI to the University of Minnesota and by the Army High Performance Computing Research Center at the University of Minnesota. A short version of this paper appeared in the proceedings of the 1990 International Conference on Parallel Processing.

1 Introduction

The problem of finding a shortest path in an explicit graph is an important problem encountered in the study of communication and transportation networks. Considerable attention has been devoted to solving this problem on sequential computers [7] as well as parallel computers [8, 11, 19, 3, 20]. This paper analyzes the scalability of a number of parallel algorithms for finding shortest paths between all pairs of nodes in a densely connected graph.

The scalability of a parallel algorithm is determined by its capability to effectively utilize increasing number of processors. The speedup obtained by a parallel algorithm is usually dependent upon characteristics of the hardware architecture (such as interconnection network, the CPU speed, speed of the communication channel, etc.) as well as certain characteristics of the parallel algorithm (such as the degree of concurrency, and overheads due to communication, synchronization, and redundant work). Given a parallel architecture and a problem instance of a fixed size, the speedup of any parallel algorithm does not continue to increase with increasing number of processors but tends to saturate or peak at a certain limit. This happens because either the number of processors exceeds the degree of concurrency inherent in the algorithm or because the overheads grow with increasing number of processors. For many parallel algorithms, a larger problem size (e.g., a shortest-path problem with a bigger graph) will have a higher speedup limit. Kumar and Rao have developed a scalability metric, called *isoefficiency*, which relates the problem size to the number of processors necessary for an increase in speedup in proportion to the number of processors used [14, 15]. The isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel algorithms [15, 21, 22, 16, 9, 13]. By doing isoefficiency analysis, one can test the performance of a parallel program on a small number of processors, and then predict its performance on a larger number of processors. As stated in [21], "... with this technique we can eliminate (or at least predict) the often reported observation that while a particular parallel program performed well on a small multicomputer, it was found to perform poorly when ported to a larger multicomputer."

The parallel algorithms analyzed in this paper are all based on two well-known sequential algorithms by Dijkstra and Floyd. These algorithms have either been previously presented elsewhere [11, 19, 3] or are small variations of them. The scalability of these parallel algorithms is analyzed for mesh, hypercube and shared-memory architectures. It is noteworthy that one of these modified algorithms has much better scalability than other algorithms over a variety of parallel architectures.

Some parallel algorithms require more aggregate memory (i.e., the sum of the memory needed on all the processors) than the corresponding sequential algorithm. Since memory is a costly resource, algorithms requiring less memory are better (provided they have similar performance otherwise). We have also analyzed the memory requirements for each algorithm-architecture pair. In some cases, an algorithm is better than others on the basis of both scalability and memory requirements. In other cases, only one of scalability or memory requirements is better. Another interesting observation is that for some parallel algorithms, addition of some special communication hardware leads to significantly better scalability as well as memory requirements.

The organization of this paper is as follows. Section 2 presents some preliminary definitions.

Section 3 defines the *isoefficiency* metric. Section 4 describes the parallel architectures studied and presents the communication models used for them in our analysis. Sections 5 through 11 describe the sequential algorithms by Floyd and Dijkstra and the parallel variants studied. These sections also contain the scalability analysis for each parallel algorithm. Section 12 presents theoretically computed speedup and efficiency for different values of the number of processors and problem sizes for some of the algorithm-architecture pairs studied in this paper. In Section 13, we show that a number of other commonly used metrics for evaluating parallel algorithms are not as useful as the isoefficiency function in capturing the utility of parallel algorithms on practically realizable parallel architectures. Finally, Section 14 summarizes the results.

A short version of this paper appeared in [17].

2 Definitions and Assumptions

In this section, we introduce some terminology used in the rest of the paper. All our algorithms work on directed graphs such as $G = (V, E)$ consisting of a set $V = \{v_i : 1 \leq i \leq n\}$ of n vertices and a set $E \subseteq V \times V$ of e edges. Since we are assuming that the graphs are dense, $e = \Theta(n^2)$. Each edge (v_i, v_j) has a length $l_{i,j}$.

We consider a parallel processor consisting of an ensemble of p processing units each of which – for purposes of determining the complexity – runs at the same speed. When a parallel algorithm is run on multiple processors, the time spent by an individual processor P_i can be split into two parts:

1. t_e^i (where e stands for essential), which is the time spent on computations that would also be performed by an optimal sequential algorithm; and,
2. t_o^i (where o stands for overhead), which is the sum of time spent on communication, synchronization, idle time, and other "non-essential" computation (i.e., computation that would not be performed by an optimal sequential algorithm).

The superscript i will be dropped whenever the time spent is the same for all processors. The execution time on p processors, T_p, satisfies[1]

$$T_p = t_e^i + t_o^i$$

The *problem size* T_e of a problem is defined to be the amount of computation (in units of time) taken by an optimal sequential algorithm. Note that we do not use the input size to refer to the problem size. Clearly, $T_e = \sum_{i=0}^{p-1} t_e^i$. We define $T_o = \sum_{i=0}^{p-1} t_o^i$. It follows that

$$T_o + T_e = p \times T_p$$

The *speedup* S of an algorithm on p processors is defined to be the ratio $\frac{T_e}{T_p}$. The *efficiency* is defined as follows:

[1] The reader should note that the sum of t_e and t_o is indeed exactly the same for all the processors. In this measure, the idle time of a processor is included in its t_o.

$$E = \frac{S}{p} = \frac{T_e}{T_p \times p} = \frac{T_e}{T_e + T_o}$$

Although isoefficiency analysis can be performed when I/O is considered, we will not consider the time spent in I/O. We assume that the input data is present in the desired locations in the parallel processor when the computation starts and the output data is similarly available at the desired locations when the computation terminates.

3 Scalability of Parallel Algorithms

If a parallel algorithm is used to solve a given problem instance of a fixed size (i.e., T_e), then the efficiency decreases as p increases. This property is true of all parallel algorithms. For a large class of parallel algorithms (e.g., parallel DFS [15], parallel shortest path algorithms [11]) the following additional property is also true:

- For any given number p of processors, the efficiency of the parallel algorithm goes up monotonically (i.e., it never goes down, and approaches a constant e, s.t. $0 < e \leq 1$) if it is used to solve problem instances of increasing size.

We call such algorithms *scalable* parallel algorithms. In these algorithms, efficiency can be maintained at a desired value (between 0 and e) for increasing number of processors provided that the problem size is also increased. Note that for different parallel architectures, the problem size may have to increase at different rates (w.r.t. the number of processors) in order to maintain constant efficiency. The rate of growth of the problem size, w.r.t. the number of processors, that is required to keep the efficiency fixed essentially determines the degree of scalability of these parallel algorithms (for a specific architecture). For example, if the problem size is required to grow exponentially w.r.t. the number of processors, then the algorithm-architecture combination has a poor scalability, as it would be difficult to obtain good speedups for a large number of processors (unless the problem size being solved is enormously large). On the other hand, if the problem size needs to grow only linearly w.r.t. the number of processors then the parallel algorithm is highly scalable; i.e., it can deliver linearly increasing performance with increasing number of processors for reasonable problem sizes. Since all problems have a sequential component (taken to be at least one arithmetic operation in this paper), the problem size must asymptotically grow at least linearly w.r.t. the number of processors to maintain a given efficiency. If the problem size needs to grow as $f(p)$, where p is the number of processors, to maintain an efficiency E, then $f(p)$ is defined to be the **isoefficiency function** for efficiency E and the plot of $f(p)$ w.r.t. p is defined to be the *isoefficiency curve* for efficiency E. It is possible to have different isoefficiency functions for different efficiencies. In this paper, isoefficiency functions are the same for all values of E within some range $0 < E \leq e \leq 1$. Therefore, for the sake of convenience in the rest of the paper, we will refer to isoefficiency functions and isoefficiency curves without reference to a specific efficiency.

Since $E = 1/(1 + \frac{T_o}{T_e})$, in order to maintain a constant efficiency, $T_e \propto T_o$ or $T_e = KT_o$ must be satisfied, where $K = E/(1-E)$ is a constant depending on the efficiency to be maintained. Once T_e and T_o are known as functions of n and p, the isoefficiency function can often be determined by simple algebraic manipulations.

Let's go through an example to understand the concept of isoefficiency in more concrete terms. Assume that the actual isoefficiency function of an algorithm on an architecture is kp^3. Further assume that on p_0 processors and on a problem of size w_0, we get a speedup of $.8p_0$ (i.e., efficiency obtained is .8). If the number of processors is doubled to $2p_0$, then the problem size must increase by a factor of $2^3 = 8$ in order to obtain a speedup of $2 \times .8p_0$ (i.e., to keep the efficiency constant at .8). Clearly, if the problem size is increased less than eight-fold, then the efficiency obtained will be less than .8.

Some of the parallel algorithms analyzed in this paper are based on the sequential algorithm by Dijkstra and the others are based upon the one by Floyd. Both sequential algorithms have $\Theta(n^3)$ worst-case time complexity, but Floyd's algorithm has a lower constant of proportionality. Hence, the isoefficiency function of each parallel algorithm is analyzed with respect to sequential Floyd's algorithm.

We also derive *memory overhead factor* (MOF) for these algorithm-architecture pairs. Memory overhead factor is the ratio of the total memory required for all processors over the amount of memory required for the same problem size on a single processor.

4 Parallel Architectures

In this section, we present models of the parallel processor architectures for which we have analyzed the scalability of various parallel shortest path algorithms.

4.1 Mesh Multicomputer

In this paper, we consider only two-dimensional, square meshes. All processors in the interior of the mesh (except the boundary) are connected to four neighbors (up, down, right, and left). Processors cooperate with each other by sending and receiving messages. We will consider three variants of this type of multicomputer: simple mesh, mesh with cut-through routing hardware, and mesh with cut-through and multicast routing hardware.

4.1.1 Simple Mesh Multicomputer

In the simple mesh, the time to deliver a message containing m words between two processors that are d hops away (*i.e.*, there are $d-1$ processors in between) is given by $(t_s + t_w m) \times d$. Here t_s is the message startup time, and t_w is per-word communication time.[2] In some algorithms analyzed in this paper, we will also need to *multicast* messages (i.e., send the same message to multiple destinations) along the row or the column of the mesh separately. For the mesh multicomputer, a multicast along

[2] t_w is equal to $\frac{x}{B}$ where B is the bandwidth of the communication channel between the processors in bytes/second and x is the number of bytes per word.

a row or column will take time equal to the time required to send a message directly to the farthest destination. In particular, if the mesh is $\sqrt{p} \times \sqrt{p}$, then the time to multicast a message of size m from an edge processor to all the processors along the row or column perpendicular to the edge will be $(t_s + t_w m) \times (\sqrt{p} - 1)$.

The equivalent of barrier synchronization [4] can be implemented in a mesh at the cost of $4(t_s + t_w)(\sqrt{p} - 1)$ as follows. First a synchronizing message of one unit is passed along all rows left-ward starting at the processors in the right-most column. Then processors in the left-most vertical column pass a synchronizing message of length one word upwards starting at the bottom-most processor. This second message is sent up by these processors only after they have received the synchronizing message along the row. These steps take $2(t_s + t_w)(\sqrt{p} - 1)$ time. Once the top-left processor gets the synchronizing message from the row and column, it broadcasts a unit message to all the processors informing them that the barrier has been reached by all the processors. This also takes $2(t_s + t_w)(\sqrt{p} - 1)$ time.

"Simple Mesh Multicomputer" is abbreviated as "Mesh" in the rest of the paper.

4.1.2 Mesh Multicomputer with Cut-Through Routing Hardware

A simple mesh may be augmented with hardware for cut-through routing [6]. This can dramatically cut down the time to deliver a message. With cut-through routing, the time to deliver a message containing m words to a processor that is d steps away is reduced from $\Theta(m \times d)$ to $\Theta(m + d)$. For multicast of a message, time taken will be $\Theta(m \log p + \sqrt{p})$. The multicast takes place by sending two recursive multicast messages, one to the originator and one to the processor half-way to the end of the row/column. Both recursive multicasts now proceed with half the length of the original row/column.

The time for barrier synchronization remains the same as for Mesh. "Mesh Multicomputer with Cut-Through Routing Hardware" is abbreviated as "Mesh-CT" in the rest of the paper.

4.1.3 Mesh Multicomputer with Cut-Through and Multicast Routing Hardware

The hardware for cut-through routing can be enhanced to efficiently support multicast in hardware [5].[3] In particular, to multicast a message along a row/column, one need only take as much time as sending a *unicast* message to the end of the row/column (i.e., $\Theta(m + \sqrt{p})$). Copies of the message are deposited at the intermediate processors without any extra delay.

The time to send a message to a single destination as well as for barrier synchronization remain the same as for Mesh-CT. "Mesh Multicomputer with Cut-Through and Multicast Routing Hardware" is abbreviated as "Mesh-CT-MC" in the rest of the paper.

[3] Strictly speaking, the scheme in [5] is a general scheme in which all destination addresses must be listed explicitly. Therefore, the message size of a multicast message along a row would be larger than that of a message sent to just a single destination. However, one can implement a trivial variant of the scheme that would require just two extra bits to encode the four cases: unicast, multicast along row, multicast along column, and general multicast. We will ignore the effect of these two extra bits on our models of the mesh parallel processors.

4.2 Hypercube Multicomputer

An h-dimensional hypercube multicomputer is a set of $m = 2^h$ processors, where two processors i and j $(0 \leq i, j \leq m)$ are connected by a bidirectional communication link if and only if the binary representations of i and j differ in exactly one bit. The time to deliver a message containing m words between two processors that are d hops away (*i.e.*, there are $d-1$ processors in between) is given by $(t_s + t_w m) \times d$. Here t_s is the message startup time, and t_w is per-word communication time. A message containing m words can be broadcast from a processor to all other processors in time $(t_s + t_w m) \times \log p$, as a binary tree of depth $\log p$ can be naturally mapped on to a hypercube.

The equivalent of barrier synchronization can be implemented in a hypercube at the cost of $2(t_s + t_w)\log p$ as follows. A virtual binary tree of depth $\log p$ is mapped onto the hypercube. A synchronizing message of 1 unit is passed upwards in this tree starting at all the leaves. A non-leaf node sends a message of 1 unit up after receiving messages from its successors. It takes $(t_s + t_w)\log p$ time for the root processor to get the synchronizing message. Now the root processor broadcasts a unit message to all the processors informing them that the barrier has been reached by all the processors. This also takes $(t_s + t_w)\log p$ time.

"Hypercube Multicomputer" is abbreviated as "Cube" in the rest of the paper. As with Mesh Multicomputer, Cube can also be augmented with hardware for cut-through routing as well as with hardware for multicast routing to derive Cube-CT and Cube-CT-MC. It turns out that the scalability of the parallel shortest path algorithms discussed in this paper does not change on the Cube with these augmentations. Hence, we only discuss scalability on Cube in this paper.

Scalability analysis of parallel shortest path algorithms for pseudo-shared memory multiprocessors like BBN Butterfly[4] can be done using the same model as the Cube; however, the proportionality constants are much smaller for BBN Butterfly compared to a Cube such as Intel iPSC/2.[5]

4.3 Shared-Memory Parallel Architectures

There are many different models of shared-memory parallel processors [2]. The one we consider here is the CREW (concurrent read, exclusive write) PRAM model [2]. Memory latency for both reads and writes, when they are allowed, is uniform for any location in memory. Concurrent reads are allowed but only one write to a given location can take place at one time. "Shared-Memory parallel processor" is abbreviated as "SM" in the rest of the paper.

5 Sequential Algorithm by Floyd

Figure 1 shows a slightly modified version of Floyd's sequential all-pairs shortest path algorithm [1] as given in [11]. A matrix P of dimension $n \times n$ is used to store the currently best known shortest distances between every pair of nodes. Initially, $P[i,j]$ is the length of the edge (v_i, v_j) if one exists, 0 if $i = j$, and ∞ otherwise. On termination, $P[i,j]$ is the length of the shortest path from v_i to v_j.

[4]Butterfly is a trade mark of the BBN Advanced Computers, Inc.

[5]iPSC/2 is a trade mark of Intel.

This algorithm works correctly for all graphs that contain no negative cycles. On each iteration of the outermost **for** loop, a new version of the P matrix is computed. P^i is the version at the end of iteration i, where $1 \leq i \leq n$. P^0 is the initial version of the matrix.

Let us assume that it takes time t_c to compute the next iteration value for one element in the matrix. Hence the sequential execution time of sequential Floyd's algorithm is $n^3 t_c$. Since Floyd's algorithm is the best known sequential algorithm for solving this problem, $T_e = n^3 t_c$.

Algorithm SF

for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $P^k[i,j] \leftarrow \min(P^{k-1}[i,j], P^{k-1}[i,k] + P^{k-1}[k,j])$
        end for
    end for
end for

Figure 1: Sequential All-Pairs Algorithm by Floyd

6 A Generic Parallel Version of Floyd's Algorithm

Figure 2 contains algorithm GPF, a generic parallel version of Floyd's Algorithm adapted from [11]. In this algorithm, the P matrix is partitioned and each partition is mapped to the memory of one of the processors. (For SM, no such physical mapping is done but the logical mapping is utilized.) For each value of k, each processor computes the P^k values for its partition of P. For this, it needs access to the corresponding segments of the row $P[k,j]$ and the column $P[i,k]$. We assume that the required initial P values are available on each processor and that the final P values stay on the processors.

Jenq and Sahni [11] describe two parallel algorithms based on this generic parallel algorithm. We briefly describe and analyze these in sections 7 and 9. In Section 8, we discuss a small variation of one of these algorithms that has better scalability on a variety of architectures. Although we have tried to make the discussion in Section 7 self-contained, the reader may find it useful to read reference [11].

7 Checkerboard Version of Parallel Floyd Algorithm

In this version (given in [11]), the cost matrix P is divided into equal parts of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, and each is allocated to a different processor. Each processor has the responsibility to update its allocated part of the matrix in each iteration. In Step 1 (Step 2) of the k^{th} iteration of GPF, each of the $\sqrt{p}$ processors that contain parts of the k^{th} column (the k^{th} row) send them to $\sqrt{p}$ other processors that need this

Algorithm GPF

Repeat steps 1 through 4 for $k := 1$ to n

Step 1: If this processor has a segment of $P^{k-1}[*, k]$, then transmit it to all processors that need it.
Step 2: If this processor has a segment of $P^{k-1}[k, *]$, then transmit it to all processors that need it.
Step 3: Wait until the needed segments of $P^{k-1}[*, k]$ and $P^{k-1}[k, *]$ have been received.
Step 4: For all i, j in this processor's partition, compute
$P[i, j] := \min\{P^{k-1}[i, j], P^{k-1}[i, k] + P^{k-1}[k, j]\}$;

End of repeat

Figure 2: Generic Parallel Algorithm Based on Floyd's Algorithm

information. In Step 3, all of the processors are synchronized. In Step 4, the essential computation is done.

Since at most n^2 processors can be kept busy using this parallel algorithm, n must grow as $\Theta(\sqrt{p})$. Or $T_e = \Theta(n^3)$ must grow as $\Theta(p^{1.5})$. Hence, the isoefficiency function due to concurrency is $\Theta(p^{1.5})$. Note that the isoefficiency function can be worse due to communication (or other factors) involved in each iteration (i.e., T_e may have to increase faster than $\Theta(p^{1.5})$ to mask the communication overhead). Of course, this will be different for different architectures.

Consider the case for Cube. As discussed in [11], it is possible to map a virtual mesh of $\sqrt{p} \times \sqrt{p}$ on a p-processor hypercube such that each row and column of this virtual mesh is a hypercube of $\sqrt{p}$ processors. Hence, the time spent in communication by each processor in each iteration in Step 1 (and Step 2) is $(t_s + t_w \frac{n}{\sqrt{p}}) \log \sqrt{p}$, as processors containing relevant information broadcast $\frac{n}{\sqrt{p}}$ units of information to $\sqrt{p}$ processors. Step 3 is essentially a barrier synchronization step to make sure that all the processors have received relevant data before they start executing Step 4. As discussed in Section 4.2, this can be performed at the cost of $2(t_s + t_w) \log p$. The time spent in Step 4 per processor in each iteration is $t_c \frac{n^2}{p}$. Hence,

$$T_p = n \times (2(t_s + t_w \frac{n}{\sqrt{p}}) \log \sqrt{p} + 2(t_s + t_w) \log p + t_c \frac{n^2}{p})$$

$$= n \times ((t_s + t_w \frac{n}{\sqrt{p}}) \log p + 2(t_s + t_w) \log p + t_c \frac{n^2}{p})$$

$$pT_p = 3npt_s \log p + 2npt_w \log p + n^2 \sqrt{p} t_w \log p + t_c n^3$$

$$T_o = pT_p - T_e$$

$$= (3t_s + 2t_w) np \log p + t_w n^2 \sqrt{p} \log p$$

Clearly, if T_e is increased (by increasing n) while keeping p fixed, T_o becomes smaller relative to T_e, thus increasing the efficiency. On the other hand, if T_e is fixed and p is increased, then T_o becomes more prominent, and efficiency decreases. If p increases, then in order to maintain the efficiency at some value E, n must grow such that

$$t_c n^3 = K((3t_s + 2t_w)np\log p + t_w n^2\sqrt{p}\log p)$$

where $K = \frac{E}{1-E}$.

Due to the first term in T_o, n must asymptotically grow as $\sqrt{K\frac{3t_s+2t_w}{t_c}p\log p}$, or T_e must asymptotically grow as $\Theta((p\log p)^{1.5})$. Due to the second term in T_o, n must asymptotically grow as $K\frac{t_w}{t_c}\sqrt{p}\log p$, or T_e must asymptotically grow as $\Theta(p^{1.5}(\log p)^3)$. Hence the overall isoefficiency function is $\Theta(p^{1.5}(\log p)^3)$.

In the scalability analysis for Cube, we took all the constants into account. To simplify the presentation, we perform analysis for the remaining architectures without taking constants into account. The reader can perform a more detailed analysis along the above lines.

In Mesh, $\Theta(n/\sqrt{p})$ information can be broadcast to p processors in $\Theta((n/\sqrt{p})\sqrt{p})$ time, and barrier synchronization takes $\Theta(\sqrt{p})$ time. Hence, the overhead due to communication and synchronization at each processor in each iteration is equal to $\Theta(n) + \Theta(\sqrt{p})$. Hence,

$$T_o = \Theta(n^2 p) + \Theta(np^{1.5})$$

For constant efficiency,

$$\Theta(n^3) \sim \Theta(n^2 p) + \Theta(np^{1.5})$$

The isoefficiency function due to the first term is $\Theta(p^3)$, and due to the second term is $\Theta(p^{2.25})$. Since $\Theta(p^3)$ is worst of all the terms including that due to concurrency, this is also the overall isoefficiency function of the parallel algorithm.

In Mesh-CT, $\Theta(n/\sqrt{p})$ information can be broadcast to p processors in $\Theta(\frac{n\log p}{\sqrt{p}} + \sqrt{p})$ time. Hence, the overhead due to communication and synchronization at each processor in each iteration is $\Theta(\frac{n\log p}{\sqrt{p}} + \sqrt{p})$. Therefore,

$$\begin{aligned} T_o &= \Theta(np(\frac{n\log p}{\sqrt{p}} + \sqrt{p})) \\ &= \Theta(n^2\sqrt{p}\log p + np^{1.5}) \end{aligned}$$

For constant efficiency,

$$\Theta(n^3) \sim \Theta(n^2\sqrt{p}\log p + np^{1.5})$$

The isoefficiency function due to the first term is $\Theta(p^{1.5}(\log p)^3)$, and due to the second term is $\Theta(p^{2.25})$. Hence the overall isoefficiency function is $\Theta(p^{2.25})$.

In Mesh-CT-MC, $\Theta(n/\sqrt{p})$ information can be broadcast to p processors in $\Theta(n/\sqrt{p} + \sqrt{p})$ time. Therefore,

$$T_o = \Theta(n^2\sqrt{p} + np^{1.5})$$

Hence, for constant efficiency,

$$\Theta(n^3) \sim \Theta(n^2\sqrt{p} + np^{1.5})$$

The isoefficiency function due to the first term is $\Theta(p^{1.5})$, and due to the second term is $\Theta(p^{2.25})$. Hence the overall isoefficiency function is $\Theta(p^{2.25})$.

For SM, the broadcasting of the information is not even necessary, as each processor can read the required value as needed. Hence, there is no overhead due to communication in steps 1 and 2. Since SM is a SIMD model of computation, step 4 would finish at the same time in each processor, hence the synchronization of Step 3 is attained at zero cost. Thus the isoefficiency function is $\Theta(p^{1.5})$, which is the isoefficiency due to concurrency.

It is easy to see that the memory overhead factor of this algorithm on all architectures is $\Theta(1)$ because the total memory requirement of all the processors is $\Theta(n^2)$. All results from this section are summarized in Table 3. We may refer to this algorithm as "Floyd Checkerboard" in the rest of the paper.

8 Pipelined Checkerboard Version of Parallel Floyd Algorithm

We will discuss this version only for multicomputers (i.e., for Mesh, Mesh-CT, and Mesh-CT-MC, and Cube). In this version, as in the version of the preceding section, each processor has the responsibility to compute the successive iteration values for the elements contained in a $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ square of the matrix. In the previous version, all the iterations are synchronized; i.e., iteration $i+1$ starts for all the processors only after iteration i has finished at all the processors and the relevant subsets of the iteration-i values of row i and column i have been communicated to all the processors. The new version differs from the previous version in that the processors are not synchronized. A processor starts the work for iteration $i+1$ as soon as it has the relevant subset of the iteration-i values of row i and column i. Whenever a processor has elements of row i (and/or column i) and has computed iteration $i-1$ values, then it immediately sends out these values to other relevant processors so that they may start working on the next iteration. All the communications are done within the mesh structure. Thus, in a setup containing p processors, the old algorithm takes $\Theta(\sqrt{p})$ steps for the iteration i values to get through to all the processors. In the new version, by the time the last processor gets the iteration-i values, the other processors that got the values earlier would already have started on the $i+1$ iteration.

Recall that it takes time t_c to compute the next iteration value for one element in the matrix, and time $t_s + t_w$ to communicate one value from one processor to its immediate neighbor in Mesh. In [18], we prove[6] that for this algorithm,

$$T_p = \frac{n^3}{p} t_c + 4(\sqrt{p} - 1)(t_s + \frac{n}{\sqrt{p}} t_w)$$

Hence, the total work done by all the processors in all the iterations is

$$pT_p = p \times (\frac{n^3}{p} t_c + 4(\sqrt{p} - 1)(t_s + \frac{n}{\sqrt{p}} t_w))$$

$$= n^3 t_c + 4p(\sqrt{p} - 1)(t_s + \frac{n}{\sqrt{p}} t_w)$$

[6] As stated in [18], the given expression for T_p is correct only when the number of elements per processor is more than 1. The time taken for the 1-element-per-processor-case is $nt_c + (4n-6)(t_s + t_w)$.

The total work done by the best sequential algorithm is $T_e = n^3 t_c$. Hence, $T_o = 4n\sqrt{p}(\sqrt{p}-1)(t_s + t_w)$. To maintain constant efficiency,

$$n^3 t_c \sim 4p(\sqrt{p}-1)(t_s + \frac{n}{\sqrt{p}} t_w)$$

or, the isoefficiency due to communication is $\Theta(p^{1.5})$.

Note that in this algorithm, p can be only as high as n^2; hence, the isoefficiency function due to concurrency is also $\Theta(p^{1.5})$. Therefore, the overall isoefficiency function of the algorithm is $\Theta(p^{1.5})$.[7]

When there is exactly one element per processor, the pipelined-checkerboard algorithm reduces to the one described by Bertsekas and Tsitsiklis [3]. Our development of this parallel algorithm was motivated by the algorithm in [3]. The algorithm in [3] can be easily adapted to work for the case when $p < n^2$ by using time-slicing to emulate n^2 virtual processors on the p processors available. The run time of this emulated algorithm $= \frac{n^2}{p}(nt_c + (4n-6)(t_s + t_w))$, and efficiency $= \frac{1}{1+\frac{(4n-6)(t_s+t_w)}{nt_c}}$. Even for the rather optimistic case of $t_s + t_w = t_c$, the efficiency of this parallel algorithm is bounded from above by .2. On the other hand, our relaxed version of Jenq and Sahni's Checkerboard algorithm can achieve efficiency close to 1 (w.r.t. to Floyd's sequential algorithm) as long as $n \approx \Omega(\sqrt{p})$.

8.1 Improved Pipelined Checkerboard Version of Floyd's Algorithm

In this section, we present a new variant of Floyd's algorithm that has $\Theta(p^{1.5})$ isoefficiency function. The total memory required is no more than that for the sequential algorithm. Further, the nature of the algorithm makes it very easy to prove these results.

The new twist compared to the previous Pipelined Checkerboard version of Floyd's algorithm is as follows. When a processor with a segment of the kth row or column receives relevant (k-1)th iteration values from the processor responsible for the matrix element in the kth row and kth column, only then does it send its relevant (k-1)th iteration values to other processors. This ensures that all processors that need to receive values from both columns as well as rows in other processors will receive them at the same time as illustrated in Figure 3. The time is shown with the symbol "t" and is given relative to the time that $P^{(k-1)}[k,k]$ is available. It is assumed that the value $P^{(k-1)}[k,k]$ takes time a to get to the processor with element $P^{(k-1)}[i,k]$ and that it takes time b to get to the processor with element $P^{(k-1)}[k,j]$.

As shown in Appendix A and B of [18], the time taken for this algorithm remains the same as the previous Pipelined Checkerboard version when there is more than one matrix element per processor. As opposed to the previous case, this expression is also valid for the one element per processor case for the current algorithm. Clearly, the isoefficiency function remains $\Theta(p^{1.5})$.

[7] Another assumption made in deriving T_p is that the message-startup overhead t_s is incurred only by the communication subsystem attached to the node CPU; i.e., the node CPU is able to proceed with other computation immediately after issuing the send message command. If we assume that the node CPU becomes unavailable for time t_s after issuing the send message command, then it can be shown that $T_p \leq \frac{n^2}{\sqrt{p}}(\frac{n}{\sqrt{p}}t_c + t_s) + 4(\sqrt{p}-1)(t_s + \frac{n}{\sqrt{p}}t_w)$. Even for this revised expression for T_p, isoefficiency function remains $\Theta(p^{1.5})$.

Figure 3: Improved Pipelined Checkerboard Version of Floyd's Algorithm

The total memory requirement (over all the processors) of this algorithm is $\Theta(n^2)$ and the memory overhead factor is $\Theta(1)$. The results from this section are summarized in Table 3. We may refer to this improved algorithm as "Floyd Pipelined" in the rest of the paper.

9 Striped Version of Parallel Floyd Algorithm

In this version (given in [11]), the cost matrix P is divided into equal parts each containing p columns, and each part is allocated to a different processor. Each processor has the responsibility to update its allocated part of the matrix in each iteration. At the beginning of the k^{th} iteration, the processor that contains the k^{th} column broadcasts it to all the other processors. Since p can at most be equal to n, the isoefficiency function due to concurrency is $\Theta(p^3)$. Note that the isoefficiency can be worse due to communication involved in each iteration. Of course, this will be different for different architectures, and can be derived by doing analysis as in Section 7. Results are summarized in Table 3. We may refer to this algorithm as "Floyd Striped" in the rest of this paper.

10 Sequential Algorithm by Dijkstra

We will now briefly present the sequential single-source algorithm by Dijkstra [1]. This can be used to solve the all-pairs shortest path problem by repeating the algorithm with every possible source. We will refer to this as Dijkstra's all-pairs shortest path algorithm. Although both this algorithm and

Floyd's all pairs shortest path algorithm have $\Theta(n^3)$ worst-case time complexity, Floyd's algorithm is preferred because of its lower constant of proportionality. However, as we will see later, a parallel version of Dijkstra's algorithm has better scalability than some parallel versions of Floyd's algorithm on some architectures.

Using our standard notation, the graph G is represented as (V, E). Each edge (v_a, v_b) is assigned a non-negative *length* $l_{a,b}$. Dijkstra's single-source algorithm (shown in Figure 4) will find the shortest distance from a specified source vertex v_s to all other vertices in G, if a path exists. Each vertex v_i in V is assigned a number d_i representing the length of the shortest path known from the source v_s. T represents the set of vertices to which a shortest path has not been found. In each iteration of the algorithm, a vertex $v_m \in T$ with the minimum d value is removed from T. The neighbors of v_m are then examined to see if a path through v_m will lead to a shorter path. If so, their d value is updated.

Algorithm SD

$d_s \leftarrow 0$
$d_i \leftarrow \infty$, for $i \neq s$
$T \leftarrow V$
for $i \leftarrow 1$ **to** n **do**
    Find a vertex $v_m \in T$ such that d_m is minimum
    for each edge (v_m, v_t) such that $v_t \in T$ **do**
        if $d_t > d_m + l_{m,t}$ **then**
            $d_t \leftarrow d_m + l_{m,t}$
        end if
    end for
    $T \leftarrow T - v_m$
end for

Figure 4: Sequential Single-Source Algorithm by Dijkstra

Let us assume that it takes time fnt_c to execute the body of the outer for loop once.[8] Hence the sequential execution time of sequential Dijkstra's algorithm is $n^2 t_c f$ for finding shortest path from one node to all other nodes, and is $n^3 t_c f$ for finding shortest path among all pairs. Clearly, the penalty factor w.r.t. the best sequential algorithm (i.e., Floyd's algorithm) is f.

[8]To simplify the analysis, we assume that the amount of work done in each iteration of the outer for loop is the same. In reality, the amount of work done in each iteration may differ slightly, as the distance of a node in the inner loop of the algorithm is revised only if an improved path to it is found.

11 Parallel Variants of Dijkstra's All Pairs Shortest Path Algorithm

11.1 Source-Partitioned Variant

In this variant of Dijkstra's algorithm, n copies of Dijkstra's sequential, single-source algorithm are equally distributed over p processors and executed in parallel. Processor p finds shortest paths from each vertex in the set $\{k \times p \mid k$ is an integer, and $k \times p \leq n\}$ to all other vertices in the graph.

Since at most n processors can be kept busy using this parallel algorithm, n must grow as $\Theta(p)$. Or $T_e = n^3$ must grow as $\Theta(p^3)$. Hence, the isoefficiency function due to concurrency is $\Theta(p^3)$.

Note that this parallel algorithm involves no inter-processor communication,[9] as each processor can figure out by itself which nodes to work on. Furthermore, if $n = k \times p$, then each processor has equal amount of work; hence, in this case, the efficiency should be 1. The total absence of communication might make one believe that this is an excellent parallel algorithm. On the contrary, as discussed above, the isoefficiency function of this algorithm due to concurrency is $\Theta(p^3)$. Since the isoefficiency function of most other parallel algorithms analyzed in this paper is better than $\Theta(p^3)$, this algorithm is less scalable than them.

The memory-overhead factor is $\Theta(p)$ because each processor has to store the length information for all $\Theta(n^2)$ edges. All results are summarized in Table 3. We may refer to this algorithm as "Dijkstra Source-Partitioned" in the rest of this paper.

11.2 Source-Parallel Variant

The major problem with the previous parallel formulation is that it can keep at most n processors busy doing useful work. Here we discuss another variant that can keep more than n processors busy. This parallel variant of Dijkstra's algorithm for all pair shortest path is similar to the previous one except that the single-source algorithm is also run on many processors as described in [19]. Thus, each of the n copies of single-source Dijkstra's algorithm is run on a disjoint subset of processors. The p processors are divided into n partitions, each with $\frac{p}{n}$ processors. (This variant is of interest only if $p > n$.) Each partition runs one copy of the parallel algorithm for the single-source shortest path problem. Thus, we parallelize at the source level first and then all available parallelism in the single-source algorithm is extracted using whatever processors are available in each partition. Clearly, this version has much more concurrency than the previous one, as it allows up to $\Theta(n^2)$ processors to be utilized. Hence, isoefficiency due to concurrency is $\Theta(p^{1.5})$. Next, we analyze isoefficiency due to communication and synchronization.

First, we consider the case of Cube. Referring again to the code for Dijkstra's sequential, single-source algorithm in Figure 4, we can see that the minimum d_m can be computed in parallel using a

[9] It is not completely correct to say that the algorithm has no communication cost. In reality, some central authority (perhaps one of the processors) needs to direct all the processors to start executing, and it also needs to receive a message from them after they all have finished executing. We have ignored this startup and finishing synchronization cost for all algorithm-architecture pairs considered in this paper. But the reader can verify that even after taking this cost into account, all the isoefficiency expressions remain unchanged.

virtual tree structure. Since there are n possible vertices to minimize over and these are distributed evenly over the p/n processors, this computation will take time $\Theta(\frac{n^2}{p} + \log\frac{p}{n})$ for SM. Of this, $\Theta(\frac{n^2}{p})$ is the useful work that is to be done even by the sequential algorithm, and $\Theta(\log\frac{p}{n})$ is the extra time spent by all the processors in the parallel algorithm. To be precise, this extra time spent in communication for determining m and d_m is $(t_s + 2t_w)\log\frac{p}{n}$. Now, m and d_m can be broadcast to all processors using a virtual tree structure in $(t_s + 2t_w)\log\frac{p}{n}$ time for Cube.

We know that the total amount of computation (other than communication and synchronization) performed by the algorithm is fn^3t_c. Hence,

$$pT_p = fn^3t_c + 2np(t_s + 2t_w)\log\frac{p}{n}$$

$$E = \frac{T_e}{pT_p}$$

$$= \frac{n^3t_c}{fn^3t_c + 2np(t_s + 2t_w)\log\frac{p}{n}}$$

$$= \frac{1}{f + \frac{2np(t_s+2t_w)\log\frac{p}{n}}{n^3t_c}}$$

To maintain constant E between 0 and $\frac{1}{f}$, n^3t_c must grow as $2np(t_s + 2t_w)\log\frac{p}{n}$. Or n^2 must grow as $\Theta(p\log\frac{p}{n})$. Therefore, $T_e = n^3$ must grow as $(p\log\frac{p}{n})^{1.5}$, or $T_e \sim (p\log p)^{1.5}$ (correct within a double log factor). In other words, the isoefficiency function due to communication is $\Theta(p\log p)^{1.5}$. Note that the isoefficiency function due to pure concurrency is $\Theta(p^{1.5})$. But the overall isoefficiency function of the parallel algorithm is the higher of the two functions (i.e., higher of the one due to communication and the one due to concurrency).

Now, consider the isoefficiency function for Mesh. Broadcasting a unit of information on a mesh of $\frac{p}{n}$ processors takes $O(\sqrt{\frac{p}{n}})$ time (as opposed to $O(\log\frac{p}{n})$ time). Hence for Mesh,

$$pT_p = \Theta(n^3) + \Theta(np\sqrt{\frac{p}{n}})$$

For constant E, n^3 must grow as $\Theta(np\sqrt{\frac{p}{n}})$, or n must grow as $\Theta(p^{\frac{1.5}{2.5}})$. Therefore, the isoefficiency function of this algorithm on Mesh is $\Theta(p^{3\frac{1.5}{2.5}}) = \Theta(p^{1.8})$.

The reader can verify that the isoefficiency functions for this algorithm on Mesh-CT and Mesh-CT-MC are also the same. The reason is that the size of all the messages that need to be passed in this parallel algorithm is a small constant; for this case, Mesh, Mesh-CT and Mesh-CT-MC have identical performance.

The memory overhead factor is n for all architectures because each group of p/n processor uses as much memory as the sequential algorithm (assuming $n > p$). Results are summarized in Table 3. We may refer to this algorithm as "Dijkstra Source-Parallel" in the rest of this paper.

12 Performance Predictions

To illustrate the predictive power of isoefficiency analysis, we computed efficiency of the parallel algorithms on Cube for a wide range of problem sizes and number of processors. The value of f was assumed to be 1.6.[10] As a result, the efficiencies of parallel algorithms based upon Dijkstra's algorithms have an upper bound of .625. Two sets of values were chosen for parameters t_c, t_s, t_w for two different kinds of CPU and communication technology. In the first set, chosen for a machine with relatively fast communication technology, $t_c = 1$, $t_s = 5$, and $t_w = .5$. In the second set, chosen for a machine with relatively slow communication technology, $t_c = 1$, $t_s = 50$, and $t_w = 1.0$.

These results are given in Tables 1 and 2. For each combination of n and p, efficiency is given for each algorithm provided it is applicable. For example, Floyd Striped is applicable only if $p < n$. Even though the efficiency figures are different in the two tables, the overall patterns in absolute and relative performance remain the same.

Floyd Checkerboard algorithm performs substantially better than Floyd Striped for both architectures for most cases. The performance of Floyd Striped is almost as good as Floyd Checkerboard only when the problem size is very large compared with the number of processors. This is predicted by our scalability analysis. On Cube, the isoefficiency function of Floyd Checkerboard is $\Theta(p^{1.5}(\log p)^3)$ which is substantially better than $\Theta((p \log p)^3)$ (the isoefficiency function of Floyd Striped). In the experiments reported by Jenq and Sahni [11], the difference in the performance of these algorithms was very small. The reason is that they computed speedup after taking the input time into consideration. If the cost matrix has to be loaded serially from outside the multicomputer (which is the case in [11]) or from a single processor in the multicomputer, then the reader can verify that no parallel shortest path algorithm has isoefficiency better than $\Theta(p^3)$. In particular, the isoefficiency functions on Cube for Floyd Checkerboard and Floyd Striped would be $\Theta(p^3)$ and $(p \log p)^3$ respectively.

Source-Parallel version of Dijkstra's algorithm outperforms Floyd Checkerboard for many combinations of n and p despite the fact that the base sequential algorithm (sequential Dijkstra's algorithm) is not as efficient as Floyd's sequential algorithm. This is again explained by scalability analysis, as the isoefficiency function of the Source-Parallel version of Dijkstra's algorithm is better than Floyd Checkerboard.

In [11], it is mentioned that Source-Partitioned version of Dijkstra's algorithm will outperform Floyd Checkerboard when p is large. ¿From Table 3, it is clear that Floyd Checkerboard is much more scalable than the Source-Partitioned version of Dijkstra's algorithm. Indeed, from the tables, it is clear that for large number of processors, Source-Partitioned version of Dijkstra's algorithm is outperformed by Floyd Checkerboard. Dijkstra Source-Partitioned does better than Floyd Checkerboard only for moderate values of p and for small values of n.

Floyd Pipelined consistently outperforms all other algorithms, as it has substantially better scalability than them.

We did not find it necessary to validate the theoretically derived efficiency figures given in this section by actual experimentation for the following reasons. The time complexity of the two sequential

[10]This value is computed from the experimental results given in [11].

P→ / N↓		4	16	64	256	1024	4096	16384	65536
64	Floyd Pipeline	0.9987	0.9905	0.9420	0.7091	0.2560	0.0441		
	Floyd Checkerboard	0.9411	0.7272	0.3478	0.1000	0.0229	0.0050		
	Floyd Striped	0.9142	0.5714	0.1818					
	Dijkstra Partition	0.6250	0.6250	0.6250					
	Dijkstra Parallel				0.3225	0.0735	0.0135		
256	Floyd Pipeline	0.9999	0.9995	0.9977	0.9882	0.9362	0.6989	0.2514	0.0436
	Floyd Checkerboard	0.9903	0.9552	0.8421	0.5714	0.2424	0.0689	0.0165	0.0037
	Floyd Striped	0.9827	0.8767	0.5423	0.1818				
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250				
	Dijkstra Parallel					0.5063	0.2173	0.0510	0.0102
1024	Floyd Pipeline	0.9999	0.9999	0.9998	0.9994	0.9975	0.9876	0.9347	0.6964
	Floyd Checkerboard	0.9979	0.9912	0.9715	0.9142	0.7619	0.4705	0.1860	0.0526
	Floyd Striped	0.9959	0.9687	0.8379	0.4923	0.1624			
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250			
	Dijkstra Parallel						0.5904	0.4255	0.1639
4096	Floyd Pipeline	1.0000	0.9999	0.9999	0.9999	0.9998	0.9994	0.9974	0.9875
	Floyd Checkerboard	0.9995	0.9979	0.9938	0.9827	0.9534	0.8767	0.6956	0.4000
	Floyd Striped	0.9990	0.9921	0.9548	0.7987	0.4425	0.1419		
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250		
	Dijkstra Parallel							0.6159	0.5594
16384	Floyd Pipeline	1.0000	1.0000	1.0000	0.9999	0.9999	0.9999	0.9998	0.9994
	Floyd Checkerboard	0.9998	0.9995	0.9985	0.9959	0.9897	0.9743	0.9360	0.8421
	Floyd Striped	0.9997	0.9980	0.9883	0.9410	0.7615	0.3995	0.1247	
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	
	Dijkstra Parallel								0.6227
65536	Floyd Pipeline	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999	0.9999	0.9999
	Floyd Checkerboard	0.9999	0.9998	0.9996	0.9990	0.9975	0.9939	0.9856	0.9660
	Floyd Striped	0.9999	0.9995	0.9970	0.9846	0.9275	0.7271	0.3635	0.1110
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
	Dijkstra Parallel								
262144	Floyd Pipeline	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999
	Floyd Checkerboard	0.9999	0.9999	0.9999	0.9997	0.9993	0.9985	0.9965	0.9920
	Floyd Striped	0.9999	0.9998	0.9992	0.9961	0.9808	0.9142	0.6956	0.3333
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
	Dijkstra Parallel								
1048576	Floyd Pipeline	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
	Floyd Checkerboard	0.9999	0.9999	0.9999	0.9999	0.9998	0.9996	0.9991	0.9980
	Floyd Striped	0.9999	0.9999	0.9998	0.9990	0.9951	0.9770	0.9014	0.6666
	Dijkstra Partition	0.6250	0.62501	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
	Dijkstra Parallel								

Table 1: Efficiencies of different parallel algorithms for a range of input size and number of processors on a machine with relatively fast communication technology. $t_c = 1$, $t_s = 5$, and $t_b = .5$.

P→ N↓		4	16	64	256	1024	4096	16384	65536
64	Floyd Pipeline	0.9950	0.9538	0.7160	0.2401	0.0381	0.0049		
	Floyd Checkerboard	0.7356	0.2758	0.0625	0.0126	0.0025	0.0005		
	Floyd Striped	0.7032	0.2285	0.0470					
	Dijkstra Partition	0.6250	0.6250	0.6250					
	Dijkstra Parallel				0.0684	0.0094	0.0015		
256	Floyd Pipeline	0.9998	0.9986	0.9913	0.9430	0.6949	0.2313	0.0373	0.0048
	Floyd Checkerboard	0.9669	0.8258	0.4812	0.1600	0.0384	0.0084	0.0018	0.0004
	Floyd Striped	0.9525	0.7150	0.2949	0.0727				
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250				
	Dijkstra Parallel					0.2061	0.0362	0.0063	0.0012
1024	Floyd Pipeline	0.9999	0.9999	0.9997	0.9983	0.9903	0.9403	0.6898	0.2292
	Floyd Checkerboard	0.9949	0.9757	0.9069	0.7032	0.3575	0.1126	0.0277	0.0063
	Floyd Striped	0.9911	0.9330	0.6989	0.3033	0.0801			
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250			
	Dijkstra Parallel						0.4145	0.1234	0.0246
4096	Floyd Pipeline	1.0000	0.9999	0.9999	0.9999	0.9996	0.9982	0.9901	0.9396
	Floyd Checkerboard	0.9989	0.9955	0.9850	0.9525	0.8540	0.6124	0.2844	0.0869
	Floyd Striped	0.9979	0.9840	0.9113	0.6585	0.2783	0.0743		
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250		
	Dijkstra Parallel							0.5545	0.3100
16384	Floyd Pipeline	1.0000	1.0000	0.9999	0.9999	0.9999	0.9999	0.9996	0.9982
	Floyd Checkerboard	0.9997	0.9989	0.9968	0.9911	0.9752	0.9304	0.8069	0.5423
	Floyd Striped	0.9995	0.9960	0.9768	0.8879	0.6131	0.2482	0.0660	
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	
	Dijkstra Parallel								0.6057
65536	Floyd Pipeline	1.0000	1.0000	1.0000	1.0000	0.9999	0.9999	0.9999	0.9999
	Floyd Checkerboard	0.9999	0.9997	0.9992	0.9979	0.9947	0.9867	0.9657	0.9094
	Floyd Striped	0.9998	0.9990	0.9941	0.9696	0.8645	0.5708	0.2218	0.0586
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
	Dijkstra Parallel								
262144	Floyd Pipeline	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	0.9999	0.9999
	Floyd Checkerboard	0.9999	0.9999	0.9998	0.9995	0.9987	0.9969	0.9927	0.9823
	Floyd Striped	0.9999	0.9997	0.9985	0.9922	0.9623	0.8420	0.5331	0.1999
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
	Dijkstra Parallel								
1048576	Floyd Pipeline	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
	Floyd Checkerboard	0.9999	0.9999	0.9999	0.9998	0.9996	0.9992	0.9982	0.9959
	Floyd Striped	0.9999	0.9999	0.9996	0.9980	0.9903	0.9552	0.8204	0.4999
	Dijkstra Partition	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250	0.6250
	Dijkstra Parallel								

Table 2: Efficiencies of different parallel algorithms for a range of input size and number of processors on a machine with relatively slow communication technology. $t_c = 1$, $t_s = 50$, and $t_b = 1.0$.

algorithms (Dijkstra and Floyd) is well-known, both theoretically and experimentally. In addition, various researchers have experimentally verified the expressions for communication delays that are used for the various parallel architectures in our analysis. Hence, the analytically derived execution times (efficiencies) would differ from the experimentally derived times only if our theoretical derivations are incorrect.

13 Other Metrics for Parallel Computation

There are a number of other metrics that have been used by various researchers to determine how good a parallel algorithm is. If an unlimited number of processors are freely available, then the time taken by a parallel algorithm (while using as many processors as possible) is a good metric. We will call this metric **Partime**. In reality, processors are costly, and, for example, it is unlikely that anyone would want to use a 10,000-processor parallel machine to solve the all-pairs shortest path problem for a 100-node graph. Hence, this metric is not very useful when the parallel algorithms being compared are meant to run on practically feasible parallel computers. For example, Partime for Checkerboard version of Floyd's algorithm is $\Theta(n^2)$ for the Mesh, Mesh-CT and Mesh-CT-MC architectures. (This time is obtained when $\Theta(n^2)$ processors are used.) But the isoefficiencies of this algorithm on the three architectures are very different (see Table 3). ¿From the isoefficiency functions, it is clear that the performance of the algorithm on the Mesh-CT-MC architecture will be substantially better than that on the Mesh architecture. If the problem size grows as $\Theta(p^3)$ (while p increases), then on Mesh, the efficiency will merely remain constant, whereas on Mesh-CT-MC it will increase. If the problem size grows as $\Theta(p^{2.25})$, then on Mesh-CT-MC, the efficiency will remain constant, whereas on Mesh it will decrease. Similarly, the Source-parallel version of Dijkstra's algorithm and the Checkerboard version of Floyd's algorithm have $\Theta(n \log n)$ as their Partime on the Cube architecture (this is obtained in each case when $\Theta(n^2)$ processors are used). But the Source-parallel version of Dijkstra's algorithm has a better scalability than the Checkerboard version of Floyd's algorithm on the Cube architecture.

Another very widely used metric is the PT-product (also called cost in [2]); i.e., the product of the run time of the parallel algorithm and the number of processors used by it. A lower bound on the PT-product is the run time of the best sequential algorithm for solving the same problem. A parallel algorithm is called **cost optimal** if its PT-product is equal to the sequential run time of the best sequential algorithm. The PT-product of the parallel algorithms considered in this paper can be computed for two cases: (i) the algorithms use as many processors as possible (in this case, T of the PT-product is identical to Partime); (ii) the algorithms use fewer processors so that the overall efficiency is high and thus the speedup is close to linear. If we consider the PT-product of these algorithms for the second case, then it turns out to be $\Theta(n^3)$ for all of them; hence the metric does not offer us any insights into the relative merits of these parallel algorithm and architecture combinations. Let's now consider the usefulness of the PT-product for the first case.

The Source-Partitioned variant of Dijkstra's algorithm uses n processors, and finishes in $\Theta(n^2)$ time. Hence, the PT-product of this algorithm is $\Theta(n^3)$ which is ideal, as it is also the complexity of the sequential algorithm. The Source-Parallel variant of Dijkstra's algorithm uses n^2 processors, and finishes in $\Theta(n \log n)$ time for Cube. Hence, the PT-product of this parallel algorithm is $\Theta(n^3 \log n)$, which is worse than that for the Source-Partitioned variant. It should be clear to the reader that the Source-Parallel variant has a much better scalability than the Source-Partitioned variant, and yet the PT-product fails to capture this. In fact, the reader can verify that the PT-product of every parallel algorithm and architecture combination discussed in this paper is no better than that for the Source-Parallel variant, and yet a majority of them are better parallel algorithms than the Source-Parallel variant (in the sense that they will provide better speedups on a reasonable set of problem instances and architecture sizes). The PT-product is also used to distinguish between the parallel algorithms that are tied according to the Partime metric. Again, the usefulness of this is limited to the case in which as many processors are available as can be used by the algorithm.

A recently introduced metric that is useful for assessing the performance of parallel algorithms on practically feasible architectures is **Scaled Speedup** [10]. This metric is defined to be the speedup obtained when the problem size is increased linearly with the number of processors. There is a close relation between this metric and isoefficiency. If the parallel algorithms under consideration have linear or near-linear isoefficiency functions, then the scaled speedup metric provides results very close to those of isoefficiency analysis. For algorithms with much worse isoefficiencies, this metric is not necessarily very useful. For example, for all parallel algorithms considered in this paper, if the problem size (i.e., T_e) is increased only linearly, then the efficiency becomes very poor.

The concept of isoefficiency functions is related to the concept of "Parallel Efficient" (**PE**) problems, defined by Kruskal et al.[12]. **PE** class of problems have algorithms with a polynomial isoefficiency function at some efficiency. The class **PE** of problems makes an important delineation between algorithms with polynomial isoefficiencies and those with even worse isoefficiencies. Kruskal et al. prove the invariance of the class **PE** over a variety of parallel computation models and interconnection schemes. The benevolent consequence of this result is that an algorithm with a polynomial isoefficiency on one architecture will have a polynomial isoefficiency on many other architectures as well. But, as seen in this paper, isoefficiency functions for a parallel algorithm can vary across architectures and understanding of this variation is of considerable practical importance. Also, parallel algorithms whose isoefficiency is a large polynomial are no good in practice. Thus the concept of isoefficiency function helps us in further dissecting the **PE** class of problems. It helps us in identifying better algorithms for problems in the **PE** class, in terms of scalability.

14 Discussion

The scalability analysis of various parallel algorithms for solving the all-pairs shortest-path problem presented in this paper gives us insights into the usefulness of these algorithms for practically feasible parallel computers. In particular, it helps us decide which algorithm-architecture combinations are superior to others. ¿From the analysis, we can also determine the effects of various architectural features on the performance of the parallel algorithms. As discussed below, for some parallel algorithms, addition of certain hardware features makes a major impact on their scalability. Table 3 summarizes the scalability results derived in this paper for various combinations of parallel algorithms and archi-

Base Alg.	Parallel Variant	Architecture	Isoefficiency	MOF
Dijkstra	Source-Partitioned	SM, Cube, Mesh, Mesh-CT, Mesh-CT-MC	p^3	p
Dijkstra	Source-Parallel	SM, Cube	$(p \log p)^{1.5}$‡	n
		Mesh, Mesh-CT Mesh-CT-MC	$p^{1.8}$	n
Floyd	Stripe	SM	p^3	1
		Cube	$(p \log p)^3$	1
		Mesh	$p^{4.5}$	1
		Mesh-CT	$(p \log p)^3$	1
		Mesh-CT-MC	p^3	1
Floyd	Checkerboard	SM	$p^{1.5}$	1
		Cube	$p^{1.5}(\log p)^3$	1
		Mesh	p^3	1
		Mesh-CT	$p^{2.25}$	1
		Mesh-CT-MC	$p^{2.25}$	1
Floyd	Pipelined Checkerboard	SM, Cube, Mesh, Mesh-CT, Mesh-CT-MC	$p^{1.5}$	1

Table 3: Isoefficiency functions and memory requirements for several parallel algorithm and architecture combinations.

tectures. Entries marked with a ‡ are correct within a double log factor. Also, all expressions are Θ expressions.

¿From Table 3, it is clear that Pipelined Checkerboard version of Floyd's algorithm has better scalability than all other parallel algorithms for architectures such as Cube, Mesh, Mesh-CT, and Mesh-CT-MC. Even on the SM architecture (which is practically unrealizable), only one other parallel algorithm (Checkerboard version of Floyd's algorithm) has the same scalability (others have worse scalability). Scalability of the Checkerboard version of Floyd's algorithm on Mesh improves substantially by the addition of cut-through routing hardware. Results such as these can be very valuable in deciding whether certain features are worth adding to the parallel hardware.

Note that on Cube, the isoefficiency function of Source-Parallel version of Dijkstra's algorithm is better than that of Checkerboard version of Floyd's algorithm. It follows that the first one can obtain higher overall speedup than the second one (even when the speedup is being computed with respect to sequential Floyd's algorithm) on a range of problem sizes and processors. This analytically driven observation is validated by our theoretically computed performance estimates in Tables 1 and 2. Note that a similar relationship between the isoefficiency functions and performance results exists for other pairs of algorithm-architecture combinations studied in this paper.

It is often incorrect to judge the merit of a parallel algorithm from speedup experiments on a sample of problem sizes and number of processors. For example, [11] concluded from a limited set of experiments that Dijkstra Source-Parallel is better than Floyd Checkerboard for large number of processors. Our results show that this is true only for moderate number of processors. Since the scalability of Dijkstra Source-Parallel is much worse than Floyd Checkerboard, it has much worse performance on very large number of processors.

A table such as Table 1 contains all the information needed to evaluate the performance of parallel algorithms on a specific architecture with specific technology dependent constants (e.g., CPU speed, communication speed). The table provides speedup obtained by the parallel algorithm (w.r.t. the best available sequential algorithm) for various combinations of problem sizes and number of processors. If such a table is available for all competing parallel algorithms for the given parallel computer,[11] then the user can easily choose the best parallel algorithm for the given problem instance that needs to be solved on a given number of processors.

Clearly, for a given parallel computer and parallel algorithm, isoefficiency function has less information than such a table; the isoefficiency function can be computed from such a table but not vice versa. But isoefficiency function is remarkably compact compared with such a table. As we see in Section 12, algorithms with better isoefficiency function outperform the other algorithms for a wide range of combinations of problem sizes and number of processors. In particular, if one wants to make efficient use of a large number of processors, then it is better to use an algorithm with better isoefficiency.

The entries in Tables 1 and 2 can change dramatically if the technology dependent factors (such as CPU speed, communication speed) change. (The reader should compare the entries in Tables 1 and 2.) But the impact of such changes can be easily predicted from the isoefficiency functions. For example, consider the case in which the ratio $\frac{t_w}{t_c}$ goes up by a factor of 10 (due to improvements in CPU technology). We can determine from the isoefficiency analysis in Section 7 that Floyd Checkerboard will give similar efficiency on problems that are bigger by a factor of 10^3. We can also determine that for Dijkstra's Source Parallel version, such a change will have no impact whatsoever (i.e., the efficiency will remain the same for each combination of problem size and number of processors).

The actual efficiencies are dependent upon such diverse factors as the degree of concurrency in the parallel algorithm and the amounts of communication and synchronization overheads relative to the rest of the computation. Isoefficiency function is a useful predictor of performance that combines the effects of all these factors in a single expression.

The isoefficiency metric primarily tells us the required growth in problem size to be able to efficiently utilize an increasing number of processors. If the problem size does not grow as rapidly, then it is clear that the efficiency will drop. But in two different parallel algorithms, the efficiency may drop at different rates (i.e., one algorithm may be more sensitive to decreased problem size than the other). For example, assume that the isoefficiencies of two parallel algorithms A1 and A2 are $\Theta(p^2)$ and $\Theta(p^3)$, respectively. Clearly, if the problem size is increased at the rate between $\Theta(p^2)$ or $\Theta(p^3)$, then A1 will

[11] Tables 1 and 2 are each equal to 5 tables, one for each parallel algorithm.

outperform A2 (asymptotically). (If the problem size is increased at a rate faster than $\Theta(p^3)$, then both algorithms will provide near-linear speedup, hence there will be little difference in their speedup performance.) But if the problem size is increased at a rate less than $\Theta(p^2)$, then it is possible for A2 to outperform A1, provided A2 is less sensitive to decreased problem size. Therefore, it follows that isoefficiency functions of A1 and A2 indicate the superiority of A1 over A2 only when problem sizes are increased in the range between the two isoefficiency functions. As a corollary, if the difference between the isoefficiency functions of two algorithms is wider, there is a wider range of problem size and architecture size combinations when the algorithm with superior isoefficiency dominates.

In this paper, we assumed that the cost matrix is available at each processor before the execution starts. If the cost matrix has to be loaded serially from outside the multicomputer or from a single processor in the multicomputer, then the reader can verify that no parallel shortest path algorithm has isoefficiency better than $\Theta(p^3)$. However, parallel I/O on the perimeter of the mesh or even along one edge will remove I/O as the primary bottleneck for scalability.

All of the parallel formulations discussed in this paper could use at most n^2 processors to solve the n-node problem. Hence, due to degree of concurrency, none of them could have isoefficiency better than $\Theta(p^{1.5})$. The following questions need to be investigated further. Is it possible to derive a parallel algorithm that has better scalability than $\Theta(p^{1.5})$? What is the lower bound on isoefficiency functions for the shortest path problem for different parallel architectures?

Acknowledgements

We would like to thank Anshul Gupta, Ananth Grama, and Chris Tomlinson for making a number of useful suggestions.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Computer Science and Information Processing. Addison-Wesley, 1983.

[2] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[3] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*, chapter 4, page 307. Prentice Hall, 1989.

[4] Steven Brawer. *Introduction to Parallel Programming*. Academic Press, Inc., 1250 Sixth Avenue, San Diego, CA 92101, 1989.

[5] Gregory T. Byrd, Nakul Saraiya, and Bruce Delagi. Multicast Communication in Multiprocessor Systems. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–196 to I–200, August 1989.

[6] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. PhD thesis, California Institute of Technology, March 1986.

[7] N. Deo and C. Pang. Shortest path algorithms : Taxonomy and annotation. *Networks*, pages 275–323, 1984.

[8] N. Deo, C.Y. Pang, and R.E. Lord. Two parallel algorithms for shortest path problems. In *Proceedings of IEEE International Conference on Parallel Processing*, pages 244–253, 1980.

[9] Anshul Gupta and Vipin Kumar. On the scalability of FFT on parallel computers. In *Proceedings of the Frontiers 90 Conference on Massively Parallel Computation*, October 1990. An extended version of the paper is available as a technical report from the Department of Computer Science, and as TR 90-20 from Army High Performance Computing Research Center, University of Minnesota, Minneapolis, MN 55455.

[10] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[11] J. Jenq and S. Sahni. All Pairs Shortest Paths on a Hypercube Multiprocessor. In *International Conference on Parallel Processing*, pages 713–716, 1987.

[12] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. Technical Report RC13572, IBM TJ-Watson Research Center, NY, 1988.

[13] Vipin Kumar and Anshul Gupta. Analyzing the scalability of parallel algorithms and architectures: A survey. In *Proceedings of the 1991 International Conference on Supercomputing*, June 1991.

[14] Vipin Kumar and V. N. Rao. Scalable parallel formulations of depth-first search. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, 1990.

[15] Vipin Kumar and V. Nageshwara Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16 (6):501–519, 1987.

[16] Vipin Kumar and V. Nageshwara Rao. Load balancing on the hypercube architecture. In *Proceedings of the 1989 Conference on Hypercubes, Concurrent Computers and Applications*, pages 603–608, 1989.

[17] Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem: A Summary of Results. In *Proceedings of the International Conference on Parallel Processing*, 1990. Extended version available as a technical report from the department of computer science, University of Minnesota, Minneapolis, MN 55455 and as MCC TR ACT-OODS-058-90.

[18] Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem. Technical Report ACT-OODS-058-90 (revised Jan. 1991), MCC, 1990. To appear in Journal of Parallel and Distributed Computing (Special Issue on Massively Parallel Computation. A shorter version appeared in proceedings of the 1990 International Conference on Parallel Processing.

[19] Richard C. Paige and Clyde P. Kruskal. Parallel Algorithms for Shortest Path Problems. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 14–19, 1985.

[20] Michael J. Quinn and Narsingh Deo. Data structures for the efficient solution of graph theoretic problems on tightly-coupled MIMD computers. In *Proceedings of International Conf. on Parallel Processing*, pages 431–438, 1984.

[21] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition.* Springer-Verlag, New York, 1990.

[22] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. Scalability of parallel sorting on mesh multicomputers. In *Proceedings of the Fifth International Parallel ProcessingSymposium*, March 1991. Extended version available as a technical report (number TR 90-45) from the department of computer science, University of Minnesota, Minneapolis, MN 55455, and as TR ACT-SPA-298-90 from MCC, Austin, Texas.