Jing-Fu Jenq and Sartaj Sahni
University of Minnesota

Abstract
Two plausible ways to implement Floyd's all pairs shortest paths algorithm on a hypercube mutiprocessor are considered. These are evaluated experimentally. A comparison with using Dijkstra's single source all desitination algorithm on each processor is also done.

Keywords and Phrases
parallel algorithms, all pairs shortest paths problem, hypercube multiprocessors

* This research was supported in part by the National Science Fundation under grants MCS-83-05567 and DCR-84-20935

Abstract
Two plausible ways to implement Floyd's all pairs shortest paths algorithm on a hypercube mutiprocessor are considered. These are evaluated experimentally. A comparison with using Dijkstra's single source all desitination algorithm on each processor is also done.

1 Introduction

Shortest path problems are amongst the most fundamental problems encountered in the study of transportation and communication networks. An annotated bibliography and taxonomy of related problems and uniprocessor algorithms appears in [4]. The *single source all destinations* and the *all pairs* shortest path problem are, perhaps, the most common forms of the shortest path problem. In each, the input is a directed graph $G=(V,E)$, $|V| = n$, $|E| = e$. A length $L(i,j)$, is associated with each edge in the graph. In the *single source* problem one is required to find a shortest path from a single vertex i to each of the remaining $n-1$ vertices $V-\{i\}$. In the *all pairs* problem one is required to find a shortest path between each pair (i,j), $i \neq j$ of vertices. Clearly, the all pairs problem may be solved by n applications of the solution to the single source problem. However, the fastest uniprocessor algorithms for the all pairs problem do not work in this way.

There have been many theoretical studies and developments of parallel algorithms for these two versions of the shortest path problem. As in the case of uniprocessor algorithms, these studies have often explicitly dealt with the problem of determining only the length of the shortest paths. A minor modification to these algorithms makes it possible to construct the actual shortest paths. [11] and [2] develop $O(\log^2 n)$ algorithms for the all pairs problem. Both use $\frac{n^3}{\log n}$ processors. While the algorithm of Savage [11] uses the concurrent read and concurrent write (CRCW) shared memory model, Dekel, Nassimi, and Sahni [2] assume a hypercube or perfect shuffle computer with each processor having its own local memory. Paige and Kruskal [9] have parallelized the Dijkstra and Ford single source algorithm under both CRCW, and EREW (exclusive read and exclusive write) models. For the CRCW model, for example, they obtain an $O(n)$ implemntation of both algorithms when $O(n)$ processors are avaliable. Frieze and Ruldolph [5] have developed an $O(\log\log n)$ expected parallel time n^2 processor algorithm for random graphs using the CRCW model.

Chen [1] and Lakhani [8] have developed distributed versions of the Ford-Moore-Bellman all pairs algorithm. The algorithm developed by Chen runs in $O(n^2 d)$ time and use n processors (d is the degree of the graph). The CRCW model is assumed. Lakhani's algorithm runs in $O(n(n + \log_2 d))$ time and uses a pipelined systolic n processor computer.

While there have been many theoretical studies of shortest path algorithms, very little experimental work on this problem has been conducted. This has been largely due to the nonavailability of parallel multiprocessors. [3] and [10] are two notable exceptions. Both report on experiences with parallel shortest

path algorithms on the Denelcor HEP (heterogeneous element processor). This is a shared memory MIMD multiprocessor computer. For the single source problem, the Pape D'Esopo version of Moore's algorithm was used and for the all pairs problem, Floyd's algorithm was used.

In this paper, we consider parallel solutions to the all pairs problem only. As with other studies, our development considers finding only the length of the shortest paths. We are interested in solving the all pairs problems on an MIMD hypercube in which each processor has local memory. Specifically, our experimentation is done on an NCUBE/7 hypercube. A block diagram of this computer is shown in Fig 1.1

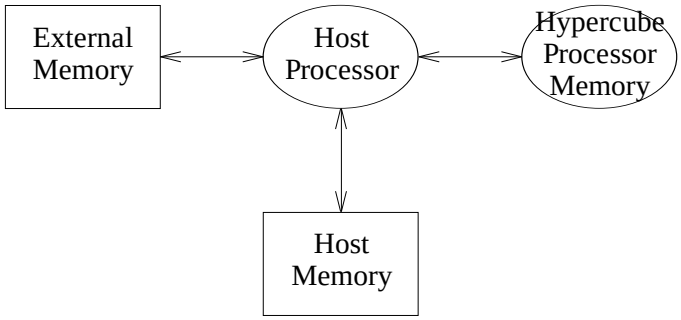


Figure 1.1 Block diagram of the NCUBE hypercube

A detailed description of the architecture of the NCUBE hypercube appears in [6]. This is an MIMD computer. Each hypercube processor is a custom 32 bit, 2 MIPS, 500,000 FLOPS processor. The local memory for each hypercube processor is either 128 K or 512 K bytes. In the configuration avaliable to us, each processor has 128 K bytes of local memory. The high level language support currently includes FORTRAN and C. Both have been extended to allow for communication between the host and the hypercube and among the hypercube processors.

When designing a parallel algorithm for a shared memory machine such as the HEP, one focus on the creation of multiple processes that avoid memory access coflicts. In the case of a computer such as the hypercube the focus is on partitioning the data so as to reduce the time spent in inter processor communication. We consider two methods of partitioning the data for the all pairs problem. These are called: by stripes and by rectangles. These two methods are evaluated experimentally and are also compared with the alternative of running a single source sequential algorithm on each of the hypercube processors.

2 Floyd's All Pairs Algorithm

Floyd's all pairs algorithm (see[3] for e.g.) is given in Fig 2.1. Here $L[i,j]$ is initially the length of the directed edge from $<i,j>$ (if present); it is zero if $i=j$; and ∞ otherwise. Upon termination, $L[i,j]$ is the length of a shortest path from i to j. The correctness of this algorithm is easily established for graphs that contains no negative cycle. On each iteration of the outermost **for** loop a new version of the matrix is computed. Let L^q be the version completed when $k = q$, $i \le q \le n$. Let L^0 be the initial version of L. It may be verified ([7] that the results are unaffected if we replace the fourth line by

$L[i,j] := \min\{ L^{k-1}[i,j], L^{k-1}[i,k] + L^{k-1}[k,j] \}$

```
-----------------------------------------------------------------
  for k:=1 to n do
   for i:=1 to n do
    for j:=1 to n do
     L[i,j] := min { L[i,j], L[i,k] + L [k,j] }
-----------------------------------------------------------------
```

Figure 2.1 Floyd's all pairs shortest path algorithm

This observation is important when developing a parallel version of the algorithm for a multiprocessor with local memory.

3 Parallel All Pairs Algorithm

In the parallel version of Floyd's algorithm, the L matrix is partitioned and each partition assigned to one of the hypercube processors. For each value of k, (cf. Fig 2.1) each processor computes the L (or L^k) values for its partition of L. For this, it needs the corresponding segments of the row $L^{k-1}[i,k]$ and the column $L^{k-1}[k,j]$. Hence on a multiprocessor with local memory the parallel version of Floyd's algorithm takes the form given in Fig 3.1

3.1 Data Partitioning and Assignment to Processors

We consider two ways to partition L: stripes and rectangles. Figure 3.2 shows how L may be partitioned into stripes and rectangles. In the case of partitioning into stripes, each partition consists of $\frac{n}{p}$ contiguous columns of L(p is the number of processor in the hypercube; we asssume n is divisible by p). Figure 3.2 corresponds to the case of $p = 16$. The number within a stripe is the processor to which the stripe is assigned. The arrows from stripe 0 identify the stripe assigned to the hypercube neighbors of stripe/processor 0 (note that processors 1 (0001), 2 (0010), 4 (0100), and 8 (1000) are the four hypercube neighbors of processor 0 (0000)).

When stripes partitioning is used, each processor always has the segment of $L^{k-1}[k,*]$ it needs for the computation of L^k. Hence step 3 of Figure 3.1 is null and may be eliminated. However, exactly one processor will contain the entire column $L^{k-1}[*,k]$ needed to compute L^k. This column may be

---

Host:

Step1 : Send each processor the L values for its partition.

Step2 : Wait until all final L segments are received from the processors.

Multiprocessor nodes

Step1 : Receive initial L values. Repeat step 2 through step 5 for k:=1 to n

Step2 : If this processor has a segment of $L^{k-1}[*,k]$ then transmit it to all processors that need it

Step3 : If this processor has a segment of $L^{k-1}[k,*]$ then transmit it to all processors that need it

Step4 : Wait until the needed sements of $L^{k-1}[*,k]$ and $L^{k-1}[k,*]$ have been received

Step5 : Compute $L[i,j]$:= min $\{L^{k-1}[i,j], L^{k-1}[i,k]+L^{k-1}[k,j]\}$ for all i,j in this processor partition;
{end of repeat}

Step6 : Transmit final L partition to host

Figure 3.1 Multiprocessor Floyd's algorithm

transmitted to the remaining processors in $\log_2 p$ transmission steps using the standard binary tree transmission scheme for hypercubes (for example, assume processor 0 (0000) has column k initially. It first transmits this column to its neighbor processor, processor 8 (1000). Next processors 0 and 8 transmit the column to their neighbor along the second most significant address bit (i.e. processor 4 (0110) and processor 12 (1100) . Next the four processors 0, 4, 8, and 12 transmit the column to their neighbors along the third most significant bit (i.e. to processors 2 (0010), 6 (0110), 10 (1010), and 14 (1110). For p=16, the final transmission is to neighbors along the fourth significant bit which is also the least significant bit.). Since "writes" on the NCUBE are nonblocking, processor 0 can begin its transmission to processor 4 before processor 8 has received the transmission. Our implementation incorporates this feature.

The time to transfer a packet of data between two hypercube neighbors is given by :

$$T_{po} + nT_p$$

where T_{po} is a constant that denotes the time required to open the channel between the two processors; n is the packet length; and T_p is the time to transfer a unit packet. When partitioning into

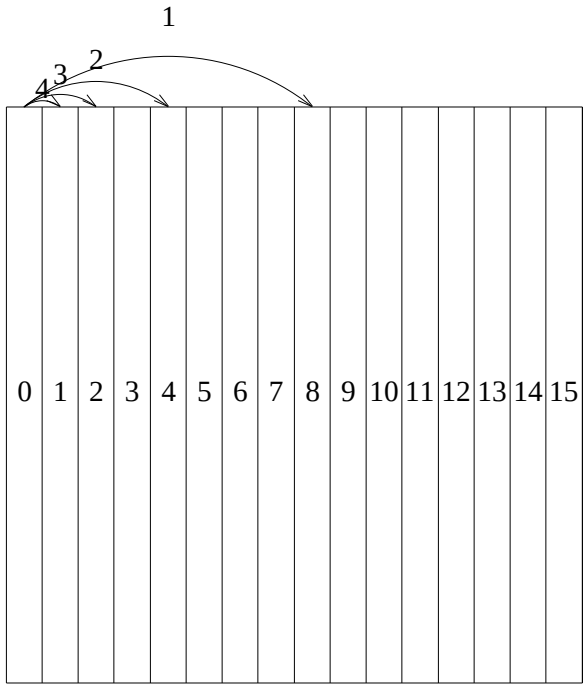


Figure 3.2 Partitioning into stripes and processor mapping

stripes is done, the time spent on inter processors data transfer is

$$T_{2D2}^{stripes} = n(T_{p0}+nT_p)\log_2 p$$

In practice, the time may be slightly larger as a processor may still be computing its L^{k-1} values from a previous iteration when the column for the next iteration is received. This will introduce a delay.

When partitioning into rectangles, each partition is a $\frac{n}{2^{\lfloor d/2 \rfloor}} \times \frac{n}{2^{\lceil d/2 \rceil}}$ rectangle where $p = 2^d$ (recall that the number of processors in a hypercube is alway a power of 2). When d is even the rectangles are actually squares. Figure 3.3 shows the partitioning and processor assignment for the case $p = 2^4 = 16$, while Figure 3.4 shows these for the case $p = 2^3 = 8$. Once again, the arrows emanating from processor 0 points to this processor's hypercube neighbors.

Both step 3 and step 4 of Figure 3.1 are required when partitioning into rectangles is done (except for the case $p = 1$ when neither step is needed and $p = 2$ when we just get two stripes). The column packets being transmitted are $\frac{n}{2^{\lfloor d/2 \rfloor}}$ units long and the row packets are $\frac{n}{2^{\lceil d/2 \rceil}}$ units long. All the column packets can be transmitted in $\lceil d/2 \rceil$ transmission steps using the tree scheme outlined earlier. The row packets can be transmitted in an additional $\lfloor d/2 \rfloor$ transmittals. The NCUBE architecture

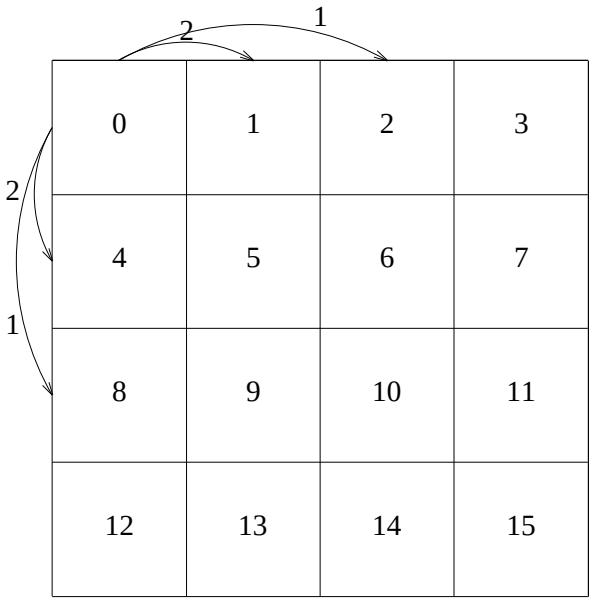


Figure 3.3. Partitioning into rectangles and processor mapping

supports the simultaneous transmittal of row and column packets along different channels. Using this capability we may overlap steps 3 and 4 of Figure 3.1

For the case when d is even, the above analysis yields:

$$T_{2D2}^{rectangle} = \alpha n(\frac{n^2}{p} + \frac{n}{\sqrt{p}}T_p)\log_2 p$$

where $1 \le \alpha \le 2$ accounts for the ability to overlap the row and column transmittals. $\alpha = 2$ if there is no overlap and $\alpha = 1$ if there is complete overlap. Comparing $T_{2D2}^{stripes}$ and $T_{2D2}^{rectangle}$ for the case d even and assuming $\alpha = 2$, we get

$$T_{2D2}^{stripes} > T_D^{rectangle} \quad iff \quad n(1 - \frac{4}{\sqrt{p}}) > \frac{T_{po}}{T_p}$$

From this, we see that if $p > 4$ then no matter how large the ratio $\frac{T_{po}}{T_p}$, there will always be values of n (perhaps quite large) for which $T_{2D2}^{stripes} > T_{2D2}^{rectangle}$. When $p = 4$, $T_{2D2}^{stripes} > T_{2D2}^{rectangle}$ for all n(unless $T_{po} = 0$ in which case the two are equal; again recall this is true only if $\alpha = 2$. If $p < 4$ then $p = 1$ or $p = 2$. In the former case, both T_D's are zero as there is no inter processor data transmission. In the latter case, d is odd and the formula

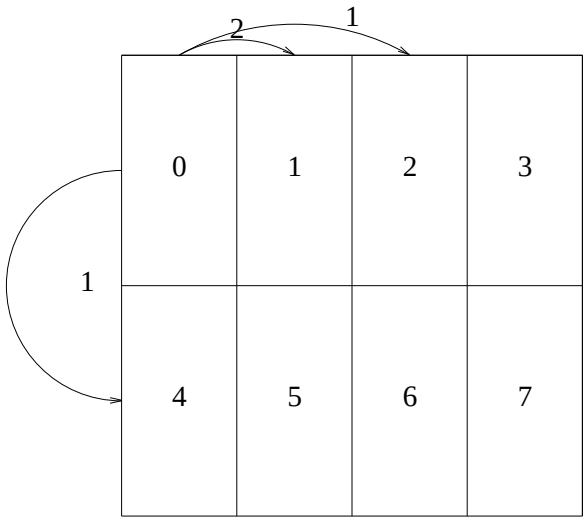


Figure 3.4. Rectangle partition and Data Transfermations for 8 processors

does not apply (also, when p =2 the rectangle and stripes partitions are the same).

Our analysis shows that T_D will generally be less for the rectangles method. This is expected even when p =4 as α is expected to be smaller than 2. The comparison between the two T_D's is significant because the host to hypercube and hypercube to host data transfer times are the same for both partitioning methods. Further, the time spent by each processor in computing the L values for its partition is the same for all processors.

4 Using A Single Source Algorithm

The inter processor data transfer time may be reduced to zero by using a single source algorithm on each of the p processors of the hypercube. In this case, each processor is assigned $\frac{n}{p}$ of the graph vertices. The single source algorithm is run once using each of these as a source vertex . I.e. each vertex essentially computes $\frac{n}{p}$ rows of the final L matrix. If the single source algorithm on a uniprocessor ran in $\leq \frac{T_A}{n}$ time (where T_A is the uniprocessor run time for the all pairs algorithm) then this strategy would outperform both the strategies outlined in the previous section. However, as pointed out earlier, the single source algorithm requires more than $\frac{T_A}{n}$ time . However, because the multiprocessor implementation of Floyd's all pairs algorithm requires interprocessor data transfer, we expect that for p suitably large, using a single source algorithm as mentioned above will result in a faster computation of the shortest paths

In practice, the size of the local memory will limit the use of this method. When the all pairs algorithm is used, each processor need be able to store only its data partition (i.e., $\frac{n^2}{p}$ entries of L). However, when the single source strategy is used, each processor needs to store the entire initial array L.

5 Experimental Results

When evaluating the effectiveness of a multiprocessor algorithm, two measures: speedup and efficiency are often used. The speedup, S, is defined to be :

$$S = \frac{time\ \ taken\ \ by\ \ best\ \ sequential\ \ algorithm\ \ using\ \ one\ \ processor}{time\ \ taken\ \ by\ \ multiprocessor\ algorithm}$$

The efficiency is :

$$E = \frac{S}{p}$$

In reporting our experimental results, we use the time taken by Floyd's uniprocessor algorithm running on one processor of the hypercube as the numerator in the ratio for S.

The algorithms descibed earlier (stripes, rectangles, single source) were programed in FORTRAN and run on the NCUBE/7 hypercube. For the single source algorithm, we used that proposed by Dijkstra. For each value of n, randomly generated L matrices were used. The average of the time for these was used to compute S (and hence E). The measured run time is the time elapsed from the initiation of step 1 of the host program (Figure 3.1) to the completion of step 2. It is assumed that the L matrix is initially in the host processor memory and that the hypercube processors are already loaded with their programs. Figure 5.1 is a plot of S vs. n for different values of p. The solid line denotes S for the rectangle method and the dotted line represents this for the stripes method. As can be seen, regardless of which method is used, the speedup increases as the number of processors increases (provided n is suitably large). Further, for any fixed p, the speedup increases as n increases and then finally saturates. The important conclusion from this is that the overhead of interprocessor communication does not overshadow the benefits of having many processors.

Figure 5.2 is a plot of the efficiency E. As is evident, very good efficiencies are obtained. When $p = 2$ an efficiency slightly in excess of 0.9 is obtainable (for large n). We were unable to compute the saturation E and S for large p´s as the local memory of 128 K per processor is insufficient to solve large instances on a single hypercube processor. Figure 5.3, however plots run time for large values of n. this is in conformance with our analysis which predicted that for n sufficiently large, the rectangle method will outperform the stripes method.

Figures 5.4 and 5.5, respectively, plot speedup and efficiency obtained by the single source method. The speedup and efficiency obtained by this method are inferior to those obtained by the stripes and rectangle methods when $p \leq 16$. However, this situation is reversed when $p \geq 32$ and $n \geq 70$ and $p = 64$ and $n \geq 64$. However, because of the memory requirement of the

single source scheme, its application is limited to cases where L fits into the local memory of each processor.

6 Conclusions

Based on our experiments, we conclude that Floyd's all pairs algorithm works better on a hypercube when the rectangle method is used rather than when the stripes method is used. For small values of n, the two methods are very competitive. When p is large (say 32 or 64), the single source method is the better way to compute the shortest paths, provided enough memory is avaliable in each processor to accomodate L.

7 References

[1] C. C. CHEN, "A Distributed Algorithm for Shortest Paths", IEEE Tranctions on Computers, VOL C-31, 1982, pp 898-899.

[2] E. Dekel, D. Nassimi and S. Sahni "Parallel Matrix and Graph Algorithms", 1981, SIAM J Computing, pp 657-675.

[3] N. Deo , C. Y. Pang and R. E. Lord, "Two Parallel Algorithms for Shortest Path Problems", IEEE International Conference on Parallel Processing, 1980, pp 244-253.

[4] N. Deo and C. Pang, "Shortest Path Algorithms : Taxonomy and Annotation", Networks, 1984, pp 275-323.

[5] Alan Frieze and Larry Rudolph, "A Parallel Algorithm for All Pairs Shortest Paths in a Random Graph", Proc. 22nd Annual Allerton Conf. on Communication, Control, and Computing, 1984, pp 663-670.

[6] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "Architecture of a Hypercube Supercomputer", International Confeerence on Parallel Processing, 1986, pp 653-660.

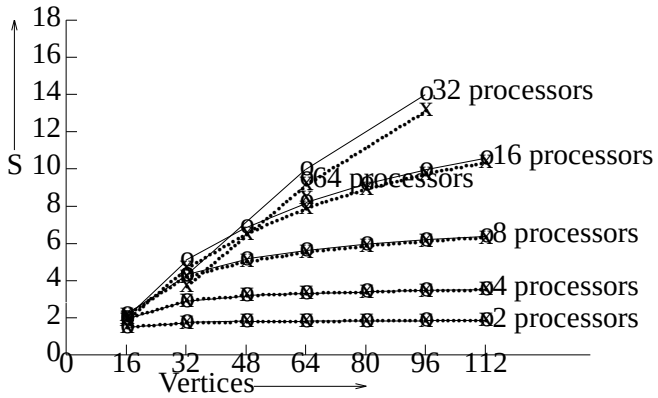
[7] E. Horowitz, and S. Sahni, "Fundamentals of Data Structures in Pascal", Computer Science Press, Maryland, 1985.

[8] Gopal D. Lakhani, "An improved Distribution Algorithm for Shortest Path Problem", IEEE Transcations on Computers, 1984, VOL C-33, pp 855-857.

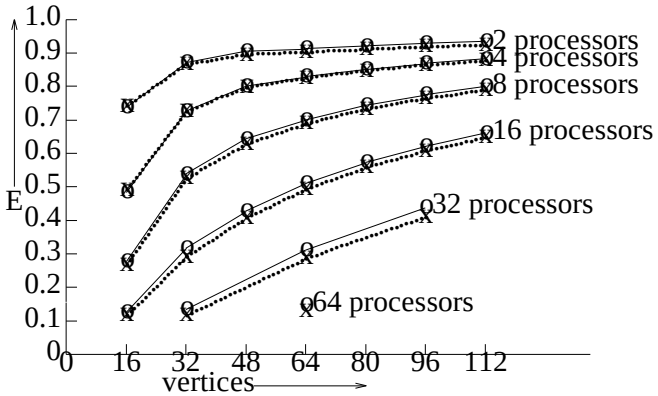
[9] R. C. Paige and C. P. Kruskal, "Parallel Algorithms for Shortest Path Problems", IEEE International Conference on Parallel Processing, 1985, pp 14-20.

[10] Michael J. Quinn and Year Back Yoo, "Data Structure for the Efficient Solution of Graph Theoretic Problems on Tightly Coupled MIMD Computers", IEEE International Conference on Parallel Processing, 1984, pp 431-438.

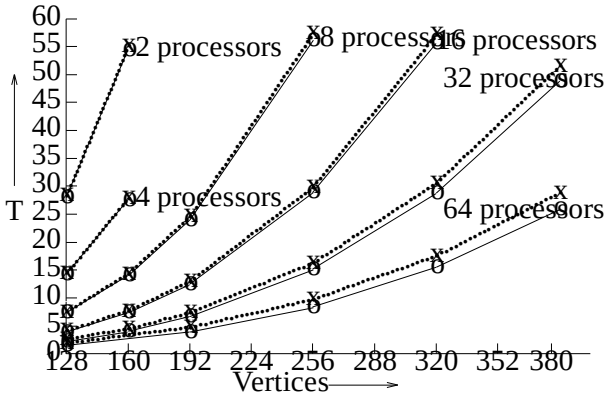
[11] C. Savage, "Parallel Algorithms for Graph Theoretic Problems", Ph. D Thesis, University of Illinois, Urbana, Aug. 1977.



dotted ------ stripes partition solid ------ rectangle partition
Figure 5.1. Speedup vs. Vertices



dotted ------ stripes partition solid ------ rectangle partition
128 elements is next n for p=64
Figure 5.2 Efficiency vs. Vertices



dotted ------ stripes partition solid ------ rectangle partition

Figure 5.3. Runtime vs. Vertices (time unit in sec)

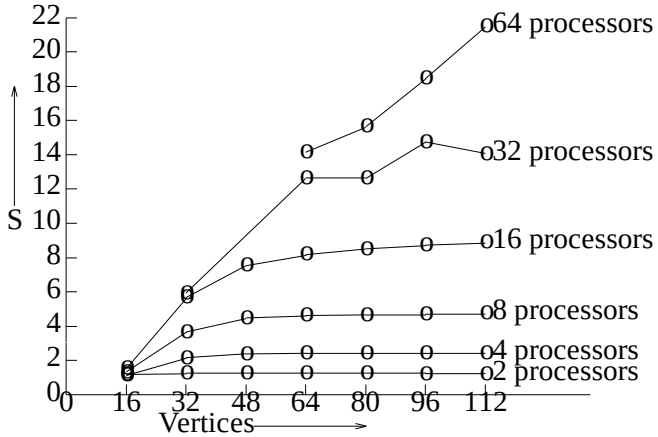


Figure 5.4. Speedup vs. Vertices Single source method

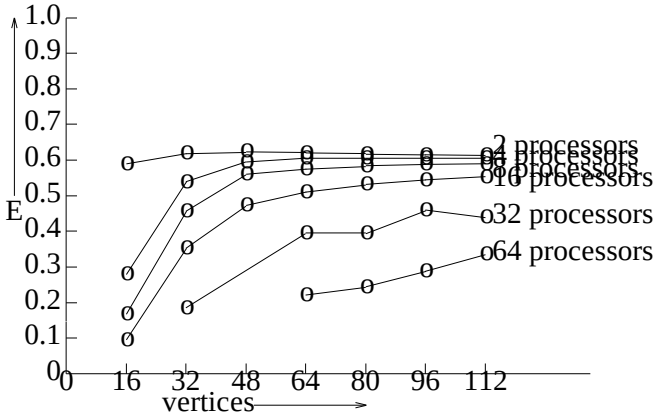


Figure 5.5 Efficiency vs. Vertices Single source method