Proceedings of the 1989 Seventeenth Annual ACM Computer Science Conference, Louisville, KY, Feb. 21-23, 1989, pp. 232-238.

Parallel Sieve Algorithms on a Hypercube Multiprocessor

Roger L. Wainwright

Computer Science Department
The University of Tulsa

Abstract

Two sieve algorithms, a pipeline and a parallel version are presented for a distributed-memory multiprocessor. Traditionally, the sieve algorithm is solved using a pipeline approach. This algorithm has a high communication to computation ratio and as a result performs poorly. The parallel algorithm is a substantial improvement over the traditional pipeline algorithm. It has a low communication to computation ratio. The speedup for this algorithm is nearly linear. Further, unlike the pipeline algorithm, the parallel algorithm performs just as well or even better on a shared memory multiprocessor system.

Keywords: Sieve Algorithm, Parallel and Pipeline Algorithms, Shared and Nonshared Memory Multiprocessors

1. Introduction

Eratosthenes of Cyrene, head of the Alexandria library around 200 B.C., developed a procedure for finding all of the prime numbers in a given range. This algorithm, known today as the Sieve of Eratosthenes, is still the best procedure for finding prime numbers. The fundamental operations of the algorithm are fairly simple, however an efficient implementation requires some additional thought. The algorithm uses a large array in memory and accesses memory frequently. Depending on the range of values to search, the sieve algorithm can use all of the available memory and if that is not enough can repeatedly reuse all of the memory. A simple example of this is given in [4] where a sieve algorithm is presented to determine the primes in any range using as little as 64 KBytes of memory. Because of the nature of this algorithm, we have seen in recent years its popularity as a benchmark on many computer systems from the smallest microcomputers to the fastest supercomputers [1,2].

I have developed two parallel versions of the sieve algorithm using the NCUBE/10 hypercube with 64 nodes at the Amoco Research Center. The first algorithm is a simple straight forward pipeline parallelization of the sequential algorithm using a ring topology. The second algorithm is a true parallel algorithm where each node works only on its assigned range of values independently of the other nodes. Both algorithms test the fundamental performance parameters of the hypercube. The algorithms are sensitive to load balancing, memory limitations and message passing constraints, including the number, length, and distance of the messages. In Section 2, the sequential sieve algorithm is reviewed. In section 3, I review a parallel sieve algorithm implemented for a shared memory multiprocessor system. In sections 4 and 5 respectively the pipeline and parallel sieve algorithms are presented. Section 6 gives summary and conclusions.

2. Review of the Sieve Algorithm

The Sieve algorithm for finding all of the prime numbers between 2 and N is described in the follows steps:

(1) The integers 2 through N are written down in a list.
(2) Locate the leftmost integer, S, in this list that has not been crossed out.
(3) From the beginning of the list, step through the list in multiples of S crossing out all values.
(4) Repeat steps 2 and 3 until S >= Sqrt(N).
(5) All values of S and any remaining numbers left in the list form the set of prime numbers in the range 2 through N.

For example, consider finding all primes in the range 2..25 using the above algorithm: (numbers are crossed out by making it blank)

Step 1:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Step 2:	S is 2.																							
Step 3:		3		5		7		9		11		13		15		17		19		21		23		25
Step 2:	S is 3.																							
Step 3:				5		7				11		13				17		19				23		25
Step 2:	S is 5.																							
Step 3:						7				11		13				17		19				23		

At this point the algorithm is finished and the prime numbers in the range 2 through 25 are 2, 3, 5, 7, 11, 13, 17, 19, and 23. Notice that some numbers may be crossed out several times. In the above example 10 was crossed out when S was both 2 and 5. It is more efficient for the algorithm to blindly crossout numbers even if they have been crossed out before. S is called the sieve value to run through the list of numbers. The algorithm terminates when S >= Sqrt(N). This is because of the simple fact that any number less than or equal to N cannot have all of its factors larger than Sqrt(N). Therefore, it is unnecessary to sieve with values beyond Sqrt(N).

3. A Parallel Sieve Algorithm for a Shared Memory Multiprocessor

Bokhari [1] presents the following parallel sieve algorithm for a shared memory multiprocessor system. In his algorithm an array of dimension N is used to hold the integers 2..N and there are p processors available. The first step of this algorithm initializes the array in parallel. Each processor initializes a subrange of the array (approximately N/p integers each). This corresponds to Step 1 in the above algorithm. Next each processor in turn locates the next S value in the array and performs a sweep over the array of integers crossing out multiples of S. This corresponds to Steps 2 and 3 in the above algorithm. His algorithm is a simple parallelization of the original sieve algorithm adapted to a shared memory multiprocessor.

There are several important observations concerning this algorithm that Bokhari points out. First, it is important after a processor has determined the next sieve value, S, that it be allowed to begin the sweeping process before the next available processor determines its S value. Some sychronization is required at this point. For example, if the first three processors are not synchronized, then it is possible for the following situation to occur. The first processor sieves 2, the second processor sieves 3, and the third processor erroneously picks up 4 to sieve, since the first processor has not yet been able to cross it out. The algorithm still works, but is inefficient since the third processor is wasting its time. Secondly, Bokhari noticed that the processor sieving 2 has the longest execution time, while the other processors all finish early. To improve utilization of the processors a load balancing algorithm is required is assist processors that are assigned low sieve values. All of this produces a complicated balancing act in order to obtain efficient results. He ran this algorithm using 0.5, 1, and 2 million numbers using up to 17 processors and it appears from his results that he was never able to obtain a speed up of more than six! His algorithm failed to utilize the full potential of the multiprocessor system.

In the remainder of this paper I present two different parallel sieve algorithms. The first algorithm is a simple parallelization of the original sieve algorithm adapted to a distributed-memory multiprocessor system. This algorithm behaves much the same way as Bokhar's algorithm with the same limitations. I was unable to obtain large speedups and the algorithm was limited in its performance capability due to routing large numbers of small messages throughout the ring of nodes. The second algorithm, however, is much simpler in concept. It is a true parallel algorithm, where each processor works independently of other processors. Message passing is minimized in this algorithm and near linear speedups are realized.

4. A Pipeline Sieve Algorithm

The cube manager obtains the value of N and the size of hypercube to use from the user. The cube manager sends N to each node then collects all of the prime numbers sent back from the various nodes. When each node finishes its work, it notifies the cube manager. When all nodes have finished, the process stops and the complete set of primes in the range 2..N are now available in the cube manager. The fundamental operations of the cube manager are shown below:

Pipeline Sieve Algorithm (Cube Manager)

```
SEND N to each node
Repeat
  RECEIVE prime numbers from the nodes
Until all nodes have completed
```

The nodes in the allocated hypercube are treated topologically as a ring. That is previous and next nodes are defined as nodes of distance one away. I considered only the cases where a complete ring is defined. Thus complete cubes from size zero to six (1, 2, 4, 8, 16, 32, and 64 nodes) were used. To begin the process, node 1 keeps sieve 2 and generates a list of odd numbers 3..N to send to the next node in the ring. The fundamental operations of each node is given below:

Pipeline Sieve Algorithm (Each Node 1..p)

```
RECEIVE N from the cube manager
IF node one THEN generate the list 3..N and SEND the list to the next node
Repeat
  RECEIVE a list of numbers from the previous node
  Strip off the first number in the list as the sieve, S
  IF S > Sqrt(N) THEN SEND the list of numbers to cube manager and STOP
  Use S to sieve the rest of the numbers in the list
  SEND surviving numbers of the list to the next node
Until end of List
SEND all S values to the cube manager with a termination notice.
```

Each node strips off the first value in the list as a sieve and uses it to sieve on the rest of the list. Surviving members of the list are forwarded to the next node. Eventually, a node will pick up a sieve value that is larger than Sqrt(N). When this happens, the node will send the remainder of the list back to the cube manager. There is no need to sieve and forward the list since all list members at this point are prime. The algorithm is basically finished except for minor house cleaning. This node sends a termination message to the next node. Each node thereafter in the ring will (1) receive the termination message, (2) send all its sieve values back to the cube manager, notifying the cube manager it is about to quit, and (3) forward the termination message to the next node in the ring. When the termination message makes a full circle the algorithm is done.

There are several subtle issues to point out. If each node were to collect the entire list before performing the sieve process, then this would essentially be a sequential process. Therefore the list must be sent in several packets in order to overlap processing. The size of each packet (or message) is an important issue in obtaining efficient performance. As a result of splitting the list into many separate messages, it is important to be able to tell if the incoming list is the continuation or the beginning of the list. In addition the list will probably be passed around the ring several times before coming to a halt. The first value in each message is examined. If it is smaller than the previous values then it is the beginning of the list. This is because a list is in increasing order.

An example of this algorithm is given in Figure 1. In this case four nodes are used and N is 45. This algorithm can be thought of as a ring of stationary processors where the list is passed around. Bokhari's algorithm, on the other hand, can be thought of as a stationary list (an array) where processors run through the list. In this example the sieve values are 2, 3, 5, and 7. The list stops at the fourth node. A synchronization problem occurs if the list makes a full cycle back to the first node. Most likely the first node is not finished with the generation of the list, and now the list has come around for processing. Care must be taken not to stack up too many messages or a deadlock may occur in the system. This is one of the fundamental performance problems with this algorithm.

Figure 2 shows the execution time versus the number of nodes. The size of the messages going around the ring greatly affects performance. In Figure 2, I show the effect of using four different message lengths (20, 50, 200, and 800) while finding all primes in the range 2..10,000. Message length in this case is measured by the number of integers sent. Multiple by four to obtain the message length in bytes. Sending only 20 values at a time from node to node proved to be very inefficient. For example message length 20 using two nodes took 8.5 seconds, while using message length 200 took 0.95 seconds. However, as the size of the cube increases this had less of an affect. In this case, there was little difference in using message sizes from 200 to 800, although the best message size appears to be around 200. I tested the case of sending message lengths of 2000, but this proved to be worse than messages in the 200 to 800 range. Generally, one should avoid sending very small or very large messages. It is difficult to tell a priori what message length should be used. The optimal value will vary with N.

Figure 2 also illustrates that this pipeline sieve algorithm has very poor performance. The best performance for N = 10,000 was 0.191 seconds occurring at 32 nodes using a message length of 200. The sequential sieve algorithm on one node takes 0.502 seconds. I define speedup as the ratio of the time using the best sequential algorithm on one node by the time from the parallel algorithm on multiple nodes. This algorithm achieved an extremely poor speedup of only 2.6 while using 32 nodes. The parallel algorithm on one node takes 1.63 seconds. This is over three times slower than the sequential version. This is due to all of the messages a single node will send to itself.

Even though the pipeline sieve algorithm is not the best approach to solve this problem, it does point out the importance of message passing constraints in a ring topology when the computation to communication ratio is small. Even when care is taken to optimize performance by adjusting the message lengths in this algorithm, it is still very easy to deadlock our system as explained in the following section.

NCUBE Communication System

The communication system in each of the NCUBE nodes is called VERTEX. It provides communication and process control functions. In our present system VERTEX uses only a few thousands bytes of memory. Approximately 30 KBytes of memory are reserved in each node for input/output message buffers. The rest of the 512 KBytes may be used for data and the executing program. Most hypercubes support the store and forward routing mechanism. This includes the NCUBE/10. A pair of adjacent processors communicate through links directly connecting the processors. This is a node to node (one hop) communication system.

Communication between neighboring nodes proceeds as follows. The sending node will transfer the message to its buffer area. If the buffer is full then the sending node "blocks" and will wait until buffer space is available. Next a request is sent to the receiving (adjacent) node indicating how long the message is. This defines the amount of buffer space needed for the message. If sufficient buffer space is available an acknowledgment message is returned to the sending node and the message is transferred and placed in the buffer area of the receiving node. If there is insufficient buffer space in the receiving node then the acknowledgment message is held up until space is available. If this happens, the sending node is "blocked". Data sent between two nonadjacent processors is copied from processor to processor in a series of one hop messages. Notice that each node stores the entire message before sending it on to the next node in the path. This uses CPU and memory resources of the processor, however it is the simplest mechanism to implement. One major side effect is that a "blocked" node will NOT allow any message traffic to go through it since its buffer space is full or it is waiting on an acknowledgment. This can easily cause a deadlock if the programmer is not careful. Also in this mechanism a message must always be sent in the reverse direction of the data (acknowledgment), thus increasing message traffic load. Both the SEND and RECEIVE commands are blocking commands. A SEND blocks until the message has been copied so when it returns the message buffer can be reused. The RECEIVE command is a blocking function and does not return until the message is received or an error is returned [3].

I have found message passing in the NCUBE system to be a delicate matter. It is very easy to deadlock the system. It is not the least bit difficult to arrive at a situation where two blocked nodes are waiting on each other. I have found in some instances that VERTEX does not seem to use a safe buffer management system. In order to ensure that a deadlock not occur, It has been suggested to never allow more than 30K bytes of messages in the entire cube at any one time. This avoids the unlikely (but possible) event that all of the messages have migrated to the same node and overwritten the buffers. My experience has been that even with much less than 30k bytes of messages in the cube, deadlock often occurs [5,6].

5. A Parallel Sieve Algorithm

In the parallel sieve algorithm the cube manager obtains the value of N and the size of the hypercube to use from the user and communicates this to node one. The cube manager is not used in this algorithm. Instead a dedicated node (node one) performs the task of determining the set of primes in the range of 2..Sqrt(N). This set of primes is called the Sieve Set. Once the Sieve Set has been determined, it is broadcasted to all of the remaining nodes, 2..p. At this point, each node including node one is responsible for using the Sieve Set to sieve N/p values each. Each node works independent of other nodes and the workload is balanced. The fundamental operations of node one are given below.

Parallel Sieve Algorithm (Node One)

(1) RECEIVE N,p from the Cube Manager
(2) Determine the Sieve Set
(3) Broadcast the Sieve Set to Nodes 2..p
(4) Using Sieve Set, sieve the range Sqrt(N) .. N/p
(5) SEND resulting primes from this range to Cube Manager

The fundamental operations of each of the other nodes, 2..p, is described below.

Parallel Sieve Algorithm (Node i, (2 <= i <= p))

(1) RECEIVE the Sieve Set from Node One
(2) Using the Sieve Set, sieve range (N(i-1))/p +1 .. Ni/p
(3) SEND resulting primes from this range to Cube Manager

An example of this algorithm is given in Figure 3. In this case, p is 4 and N is 4096. Node one determines the Sieve Set. This can be done recursively by first determining the primes in the range 2..Sqrt(Sqrt(N)) and so forth. In this case node one determines the primes in the range 2..8 and uses this to sieve values through 64. The Sieve Set is then broadcasted to nodes 2..4. At this point each of the nodes 1..4 are responsible for determining primes in the respective ranges 65..1024, 1025..2048, 2049..3072, and 3073..4096.

The data communication in this algorithm is much less than that of the pipeline version. Each node receives the Sieve Set, and returns the primes in its assigned range. There is no required internode communication as in the pipeline sieve algorithm. This means that any number of nodes may be used to solve the problem; the number of nodes used does not need to be a power of two. In order to overlap computation of the Sieve Set by node one and the computation of the other nodes, the Sieve Set is sent in separate packets. For example, after the first three values of the Sieve set (2, 3 and 5) are determined, they are sent to the other nodes. The amount of computation required by the nodes to sieve their respective ranges using these three values is generally more the the amount of time required by node one in determining the rest of the Sieve Set. Thus when the sieving nodes require additional sieve values, they are usually ready and waiting. Therefore, in this algorithm all of the nodes can begin to process in parallel almost immediately.

Figure 4 shows speedup of the parallel sieve algorithm versus the number of nodes for N = 10,000, 60,000, 400,000 and one million. The speedup is almost perfectly linear in each case up to a certain point. After a certain point the computation to communication ratio decreases to a point where it begins to degrade performance. In some cases performance was actually worse when additional nodes were added. Generally, this algorithm works best for large values of N where the computation to communication ratio is kept high. Note the case where N is one million. Speedup was nearly linear up to around 30 nodes. The speedup was about 44 when 64 nodes were used.

6. Summary and Conclusions

I have described two parallel implementations of the sieve algorithm on the NCUBE/10, a distributed-memory multiprocessor system. The first algorithm is the traditional pipeline sieve algorithm. This algorithm uses a ring topology. The nature of the algorithm is such that the ratio of communication to computation ratio is extremely high. The size of the messages going around the ring plays an important part in the overall efficiency of this algorithm. The pipeline sieve algorithm only works for small values of N due to communication limitations. It was very easy to cause a deadlock in the system with this algorithm due to the high amount of message traffic. Ultimately, message traffic was artifically slowed down to bring the computation to communication

ratio more in line. This greatly affected efficiency. The performance of the algorithm is extremely poor. The best performance, for example, for N = 10,000 was a speedup of only 2.6 using 32 nodes. Not only is the amount of message traffic a major performance factor, but the work load is unbalanced among the nodes. Nodes lower in number have more work to do than nodes higher in number, due to the pipeline nature of the algorithm. The poor performance of this algorithm is no different from that obtained by Bokhari [1] in his pipeline algorithm on a shared memory multiprocessor system. His pipeline algorithm exhibited the same load balancing problems and relatively poor speedups. The pipeline approach is clearly inefficient regardless of memory configuration.

The second algorithm described in this paper is a parallel sieve algorithm. This algorithm takes a different approach compared to the first algorithm and is much more efficient than Bokhar's algorithm. The work load among the processors is divided evenly. The processors do not communicate among each other; they work completely independent of each other. Communication is only with the cube manager. Because of this the parallel sieve algorithm will work just as well on a shared memory system, perhaps better. There are no memory contention problems. The parallel algorithm is a substantial improvement over the traditional pipeline algorithm. It has a low communication to computation ratio and works best when this ratio is keep low, that is for large values of N. For size one million, the speedup was nearly linear up to a 30 nodes before beginning to level off.

Clearly, the parallel version of this algorithm is substantially superior in every measurement over the traditional pipeline sieve algorithm, regardless of shared or distributed-memory. I feel that both the parallel and pipeline sieve algorithms will continue to be useful test procedures for multiprocessor systems. The pipeline sieve algorithm tests a systems ability to handle algorithms with high communication to computation ratios with heavy message traffic. The parallel sieve algorithm tests a systems ability to handle a truly parallel algorithm with the hope of obtaining nearly linear speedup.

Acknowledgments

The work reported in this paper was supported by Amoco Research Center Tulsa, Oklahoma.

References

[1] S.H. Bokhari, Multiprocessing the Sieve of Eratosthenes, Computer, Vol. 20, No. 4, April 1987, 50-58.

[2] J.R. Edwards and G. Hartwig, Benchmarking the clones, Byte, Vol. 10, No. 11, Nov. 1985, 195-201.

[3] NCUBE Corporation, NCUBE System Manual, 1987.

[4] T.A. Peng, One Million Primes Through the Sieve, Byte, Vol. 10, No. 11, Nov. 1985, 243-244.

[5] R.L. Wainwright, Deriving Parallel Computations from Functional Specifications: A Seismic Example on a Hypercube, International Journal of Parallel Programming, Vol. 16, No. 3.

[6] R.L. Wainwright, Message Passing Considerations for Hypercube Multiprocessors, Proceedings of the Second Workshop on Applied Computing, University of Tulsa, Tulsa, Oklahoma, March, 1988.

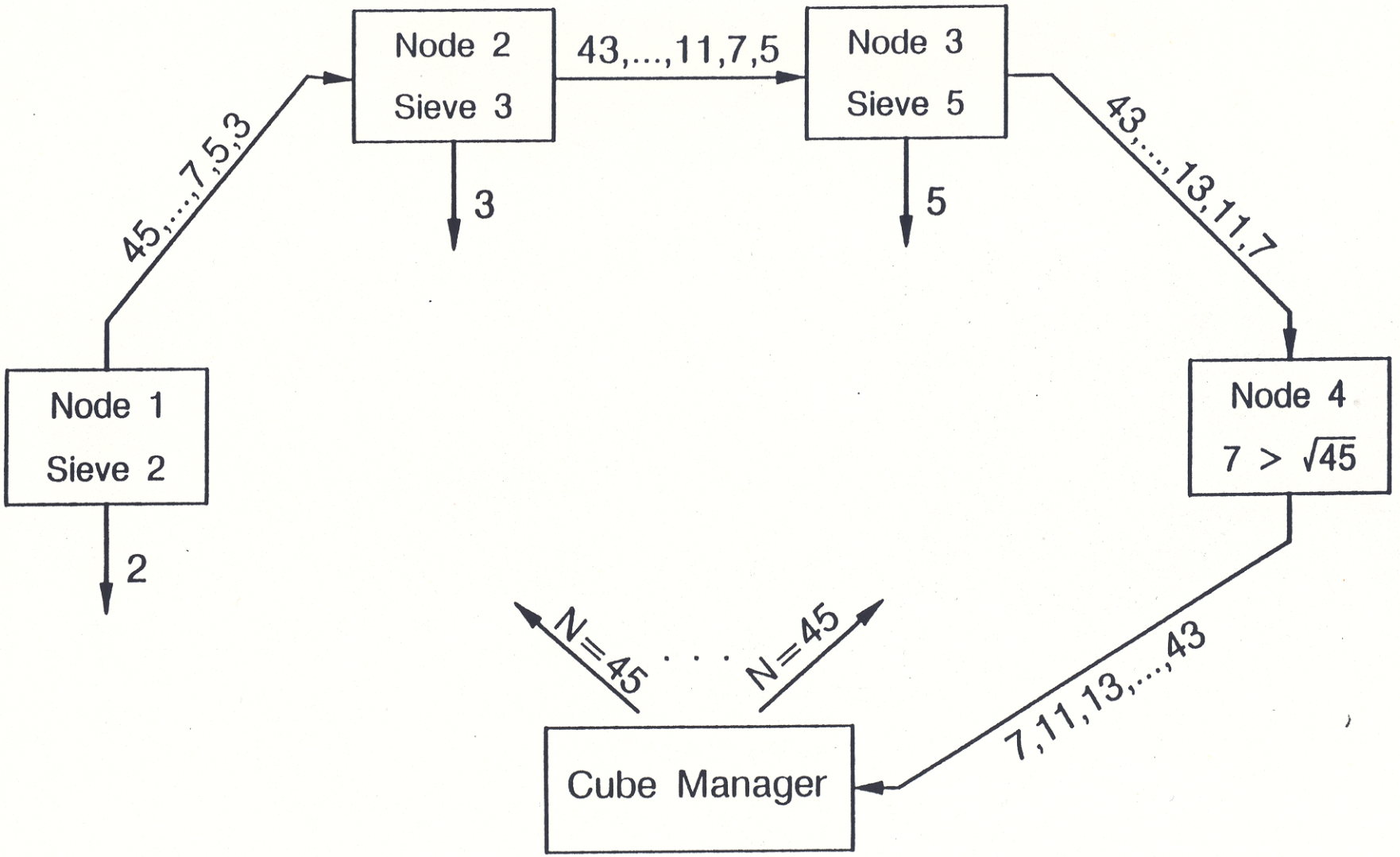
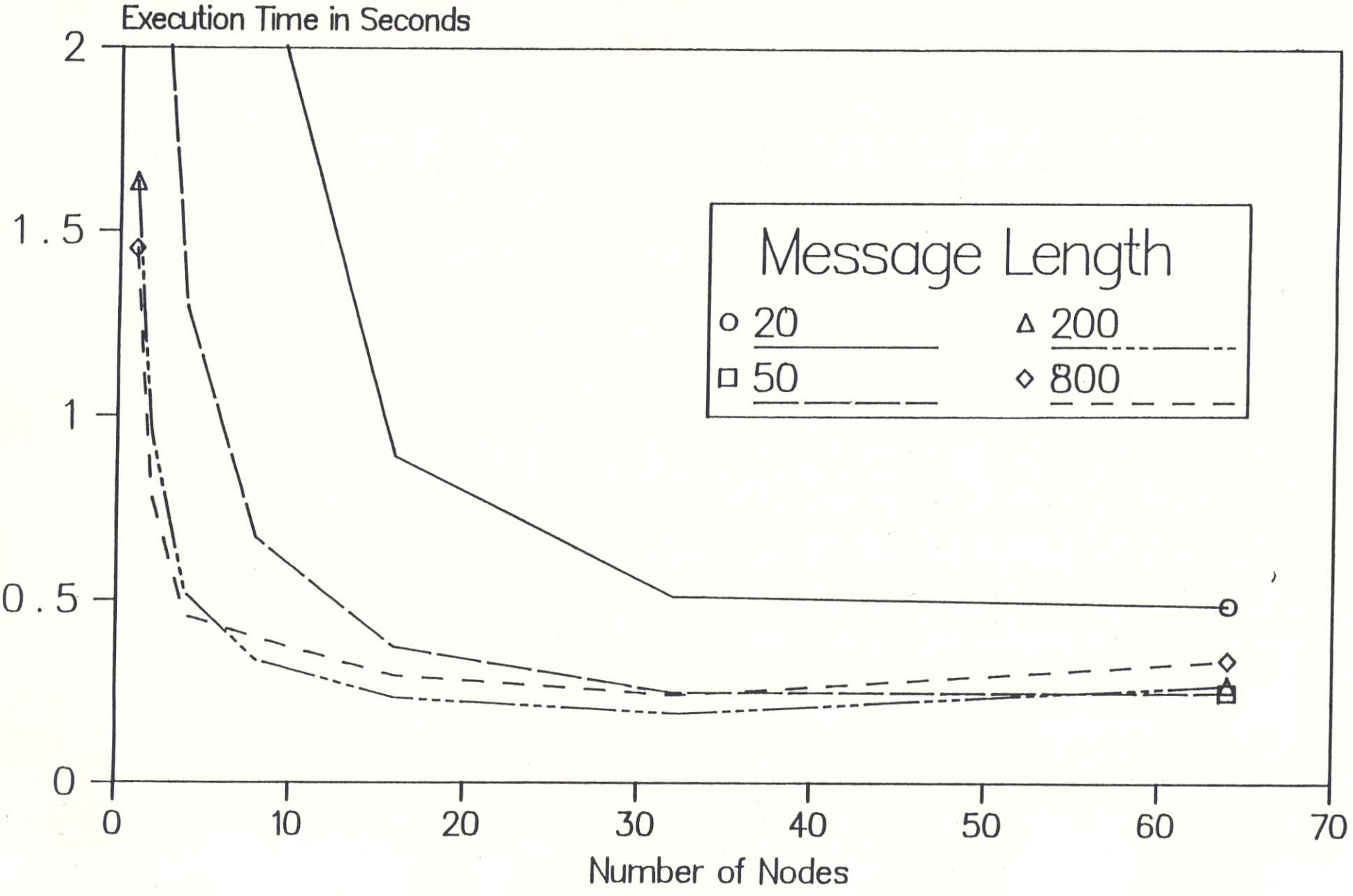


Fig. 1 Pipeline Sieve Algorithm P=4, N=45

Figure 2: Execution Time Versus Number of Nodes



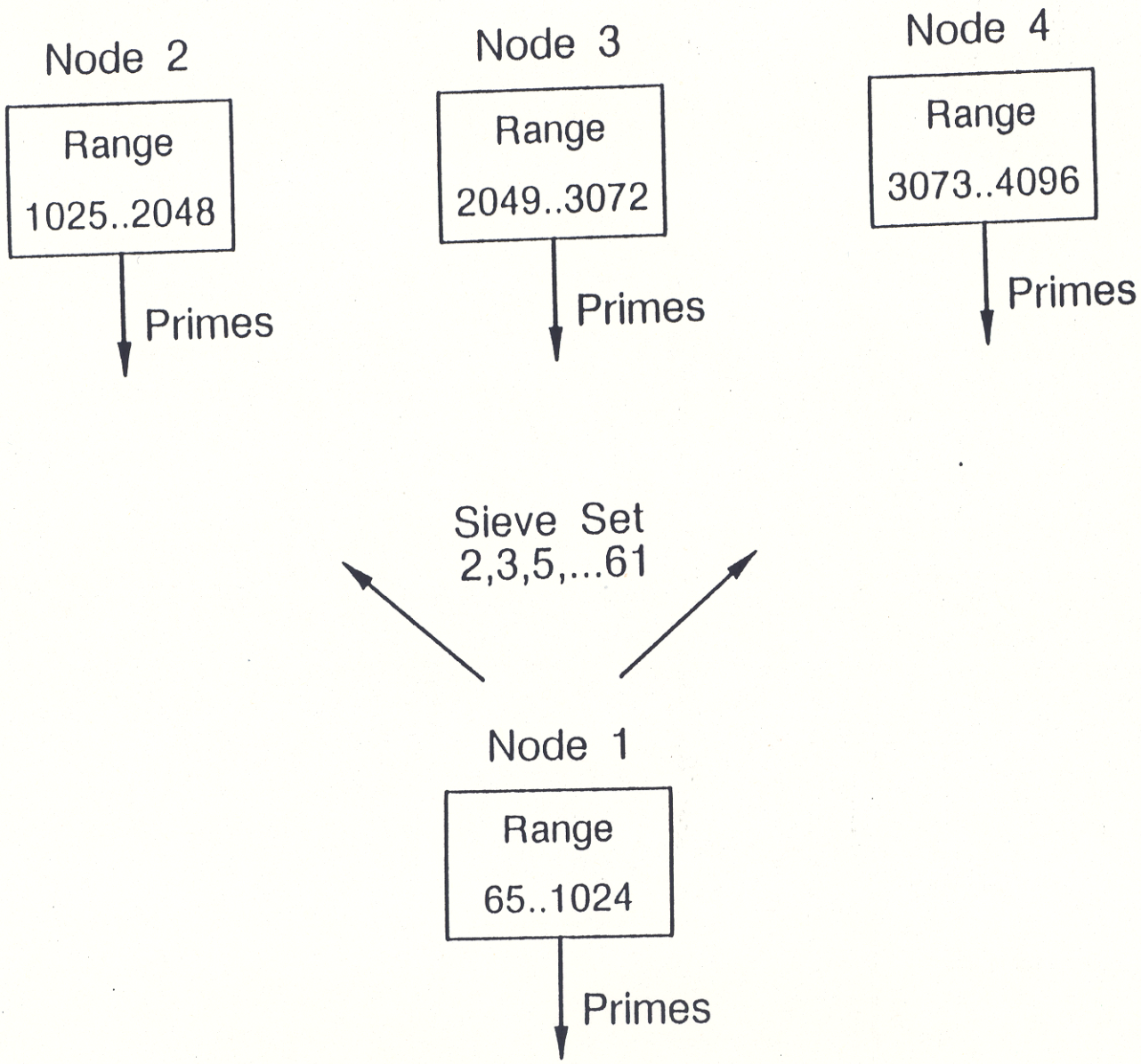


Fig. 3 Parallel Sieve Algorithm N=4096, P=4

Figure 4: Speedup of Parallel Sieve Algorithm