NASA Contractor Report 178131

ICASE REPORT NO. 86-41

ICASE

Multiprocessing the Sieve of Eratosthenes

Shahid H. Bokhari

Contract No. NAS1-18107
June 1986

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Multiprocessing the Sieve of Eratosthenes

Shahid H. Bokhari
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Hampton, VA 23665, USA
and
Department of Electrical Engineering
University of Engineering and Technology
Lahore-31, Pakistan

ABSTRACT

The well known Sieve of Eratosthenes for finding prime numbers dates back to about 200 B.C. In recent years it has seen much use as a benchmark algorithm for serial computers while its intrinsically parallel nature has gone largely unnoticed.

We describe the implementation of a parallel version of this algorithm for a real parallel computer, the Flex/32, and discuss its performance. It is shown that the algorithm is sensitive to several fundamental performance parameters of parallel machines, such as spawning time, signaling time, memory access, and overhead of process switching. Because of the nature of the algorithm, it is impossible to get any speedup beyond 4 or 5 processors unless some form of dynamic load balancing is employed. We describe the performance of our algorithm with and without load balancing and compare it with theoretical lower bounds and simulated results.

It is straightforward to understand this algorithm and to check the final results. However its efficient implementation on a real parallel machine requires thoughtful design, especially if dynamic load balancing is desired. The fundamental operations required by the algorithm are very simple: this means that the slightest overhead appears prominently in performance data. The Sieve thus serves not only as a very severe test of the capabilities of a parallel processor but is also an interesting challenge for the programmer.

Research supported by NASA Contract No. NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center.

1. Introduction

More than two thousand years ago, Eratosthenes of Cyrene described a procedure for finding all prime numbers in a given range. This straightforward algorithm, known as the Sieve of Eratosthenes, is to this day the only procedure for finding prime numbers [2]. In recent years it has been of interest to computer scientists and engineers because it serves as a convenient benchmark to measure some aspects of a computer's performance. Specifically, the Sieve tests the power of a machine (or of a compiler) to rapidly and repeatedly access a very large array in memory. This is clearly influenced by memory access time, the speed at which indexing is done and the overhead of looping. Over the last decade we have seen numerous instances of its use as a benchmark in both professional journals and popular computer magazines [1],[7].

In this paper we describe the implementation of a parallel version of the Sieve on the Flex/32 shared memory multiprocessor and discuss its performance. We show that the algorithm is sensitive to several fundamental performance parameters of parallel machines, such as spawning time, signaling time, memory access, and overhead of process switching. Because of the nature of the algorithm, it is impossible to get any speedup beyond 4 or 5 processors unless dynamic load balancing is employed. We describe the performance of our algorithm with and without load balancing and compare the performance with theoretical lower bounds and simulated results. There is good agreement between simulated results and observed run times, indicating that the performance parameters used in our simulation are an accurate measure of the machine's performance.

We feel that the Sieve of Eratosthenes is very useful as a test of the capabilities of a parallel machine. It is straightforward to understand the parallel algorithm and to check the final results. However the efficient implementation of the algorithm on a real parallel machine, especially in the dynamic load balancing case, requires thoughtful design. The basic operations required by this algorithm are very simple. This means that the slightest overhead shows up prominently in performance data. The

Sieve thus serves not only as a very severe test of the capabilities of a parallel processor but is also an interesting challenge for the programmer.

2. The Sieve of Eratosthenes

Suppose we wish to find all prime numbers between 2 and N. Eratosthenes' algorithm proceeds as follows. The numbers 2 to N are first written down. We start with the first number (2) and step through the list crossing out all of its multiples (4, 6, 8,...). Having exhausted the list, we return to the starting point, look at the next uncrossed number on the list (3), and cross out its multiples (6, 9, 12,...). Coming back again to the beginning of the list, the next uncrossed number we encounter will be 5 (4 having been crossed out during the first pass). We repeat the crossing out sweep through the list with 5 and so on. At the end of this process (illustrated in Fig. 1) the uncrossed numbers are the primes.

Several issues need to be clarified. Firstly, we explicitly eliminate all multiples of 2 from out list despite the fact that we could have started with a list of odd numbers only and reduced our total work considerably. We prefer not to exclude even numbers because our purpose is not to generate prime numbers efficiently but rather to use the Sieve as a measure of a computer's performance.

Secondly, we need perform the crossing out sweeps only up to $\sqrt{N}$, because any number less than N cannot have all its factors greater than $\sqrt{N}$. For example, if N=100 we need only sweep through four times, removing the multiples of 2, 3, 5, and 7. There is no point in removing the multiples of 11 since 22 will have been removed when sweeping with 2, 33 when sweeping with 3 etc.

It is interesting that the first of these issues (listing only the odd numbers) was known to Eratosthenes and is mentioned in the ancient text "Introduction to Arithmetic" by Nicomachus [6], the oldest reference to the Sieve commonly available in translation. Eratosthenes does not, however, state the notion of restricting the sweeps to less than $\sqrt{N}$. This first appears explicitly in the works of Leonardo of Pisa (Fibonacci) more than 1400 years later [5].

3. A Parallel Version of the Sieve

E. Shapiro [8] proposes a parallel version of the Sieve as follows. Eratosthenes starts off with a large number of slaves and instructs the first one to write down the integers 2,3,4,... on the ground in a field. A second slave is dispatched with instructions to cross out all multiples of two. Eratosthenes now moves down the list of numbers and dispatches a new slave whenever an uncrossed number is encountered. Thus the third slave is sent off with instructions to cross out 3, 6, 9, etc., the fourth crosses out all multiples of 5 and so on. The uncrossed numbers that remain at the end of this process are the primes.

It is amusing to speculate if this is actually how Eratosthenes sieved prime numbers, whether he had an unlimited supply of slaves and if not how many slaves he used to sieve what range of numbers and in how much time. If this really happened, it would probably represent one of the earliest instances of the implementation of a parallel algorithm. However, this 'parallel' version of the Sieve is not alluded to in any of the standard reference works on classical mathematics [5],[9].

For our purposes it suffices to note that the Sieve is easily parallelized and that the scenario described above can be programmed on a real shared memory multiprocessor. There are, however, some subtle points that need to be brought out.

Note first of all that synchronization is implicit in the above description. Eratosthenes politely waits until all slaves he has dispatched have moved at least one number away from him before moving forward himself and examining the next number. This ensures that he does not dispatch a slave uselessly to cross out the multiples of a non prime. For example, when Eratosthenes looks at 6, the slaves crossing out multiples of 2 and 3 should already have passed over it. If neither of them have, then a new slave will needlessly be dispatched crossing out multiples of 6. This will not affect the correctness of the algorithm but will definitely reduce its efficiency. Furthermore, slaves who are moving rapidly through the field—either because they are fast runners or because they are striking out multi-

ples of large primes, or both—must not overrun the initializing slave, who is writing down the integers from 2 to N.

The second point to appreciate is that the total amount of work done by any slave is inversely proportional to the size of the prime number whose multiples it is eliminating. When sieving a finite range of numbers, we expect that the slaves handling the last few primes will quickly finish their work and be idle for most of the time. This phenomenon has a major impact on the efficiency of the parallel algorithm as we will discuss below.

4. The Flex/32 multiprocessor

The Flex/32 currently being installed at NASA Langley Research Center is made up of twenty processors based on the National Semiconductor 32032 microprocessor[4]. Each processor has about one Megabyte of local memory and all processors can access about two Megabytes of shared memory through a global bus.

Fig. 2 illustrates the current configuration of the machine. Pairs of processors reside on each of ten 'local' buses and all twenty machines must access the shared memory by going through the local and then the global bus. The processors are numbered 1 to 20; at Langley, Processors 1 and 2 operate under a Unix operating system and are used for program development and for loading and booting up the remaining processors when a parallel program is to be run. Thus, under normal circumstances, 18 processors are available for parallel processing—these run the MMOS concurrent operating system. From amongst these 18, only processor 3 has a console attached to it. It has been our experience that the overhead of handling console I/O can significantly impact the running times of processes on this processor. We have therefore excluded this processor from our experiments, which report timings for only 17 processors.

At the time the research reported in this paper was carried out, the machine was operating in an experimental mode, and not all of the ultimately envisaged facilities were available on it. Of greatest

importance is the fact that hardware locking of shared memory locations was not implemented—this was done in software with heavy overhead.

5. Basic Parallel Algorithm

In our parallel implementation of the Sieve of Eratosthenes there is one master process that orchestrates all work. We use one slave process per processor and time-share the master process with one of the slaves. The reason for this is discussed below. We use an array of dimension N to sieve N numbers. This array is of type character and its elements are all initialized to '*'. Locations whose indices are multiples of primes are then set to ' ' (blank) as the algorithm proceeds. In the end, the indices of all locations that are still '*' are the prime numbers. The initialization is done in parallel by all available slaves. Thus if there are p processors each initializes a subrange of length approximately N/p. This is in contrast with the scenario described in Section 3 above, where one slave does the initialization. It ensures that the initialization is done with the full power available to us and eliminates the need to synchronize in order to prevent a fast moving slave from overrunning the initializing slave and writing into a location which has not yet been initialized.

Once initialization is complete the master moves forward through the array, dispatching a slave whenever he encounters a location that is set to '*'. Slaves move through the array setting locations that are multiples of prime numbers to ' '. Should no idle slaves be available, the master waits until a currently working slave has finished his sweep and signaled his availability.

Fig. 3 lists this basic algorithm and Fig. 4 illustrates its execution for N=100 and p=3.

5.1. Details of Implementation

Examination of Fig. 3 will reveal that the bulk of the work in this algorithm is the setting to ' ' of locations in shared memory. This is perhaps the simplest memory operation that one can think of. It effectively precludes the use of any synchronization to ensure that the master processor does not

overshoot a slave—the overhead would be many times the amount of useful work being done. We have therefore not used any such synchronization. As discussed in Section 3 above, this does not invalidate the algorithm but does cause an overhead because some slaves are occasionally dispatched to strike out multiples of non-primes. This overhead is several orders of magnitude less than the overhead of synchronization and we are happy to live with it.

To minimize the incidence of the abovementioned 'false primes', we time-share the master process with another slave. This reduces the rate of progress of the master and almost eliminates the incidence of overruns. If we were not to do this, there would be a large amount of extra work caused by 'false primes' as illustrated in Fig. 5.

When running this algorithm with p processors, we spawn all p slaves at the beginning of the program. The slaves are given signals to proceed with a prime number as required by the master and to signal completion when they are done. A different approach would be to spawn a slave whenever required and to let it die upon completing its work. While this alternative results in a more elegant algorithm, the overhead of spawning a new process on most conventional multiprocessors is too high to permit this to be an efficient implementation. This issue is discussed by Jordan [3] for other applications.

The algorithm was implemented in C augmented with a few parallel processing primitives (Fig. 3). The parallel programming primitives are as follows.

when(condition) wait until the specified condition is true.

process(procedure_name(parameters),processor_number)

the specified procedure is started up on the specified processor.

lock(shared_variable) An attempt is made to lock a variable in shared memory. If that variable is currently not locked by any other process, the lock will be granted. Only the locking process will then be able to access this shared variable. If the variable

is found to be already locked by some other process, the locking process will wait until the lock is released.

unlock(shared_variable)release a locked variable.

We were able to take advantage of C's access to machine registers and obtain very good array access times, as indicated by the timings that are discussed in the following sections.

5.2. Observed Timings for Basic algorithm

Fig. 7 shows our measured times for the basic algorithm when sieving 0.5, 1 and 2 Million numbers. Each set of curves comprises 15 different runs. Inspection of these curves yields several interesting facts.

(1) There is no speedup beyond about 6 processors. This is because the slaves sieving the first few primes dominate the total computation time. This is illustrated in Fig. 6 for 7 processors. It can be seen that the slave sieving 2 goes on for the longest time while the others finish quickly. When there are fewer than 6 processors the total time to sieve primes other than 2 is greater than the time to sieve 2 and a speedup is in fact observed.

(2) There is a well defined spread of run times from about 2 to 7 processors. This is because of the uncertainty inherent in parallel processing. To appreciate this, note that the '**when**(condition)' statement is implemented on a conventional machine as a call to an operating system primitive. The operating system checks for 'condition' to become true at periodic intervals. There is no guarantee that a process will get its 'when' signal at precisely the same instant every time the program is executed. Because of this, a slave may get a signal to proceed slightly later during one run than in another. The master may assign slightly heavier load on one slave than on another. This spread does not occur for runs with one processor because there is essentially no parallel processing in that case. Nor does it occur for runs with very large numbers of processors

since there is not enough parallelism in the problem—there is an idle slave waiting for a new prime whenever the master needs to dispatch one.

6. Load Balancing

To improve the utilization of processors and thus reduce the running time, we developed a load balancing algorithm and measured its performance. The load balancing strategy is for each idle slave to take over half the load of the most heavily loaded slave. This load balancing phase is not started until the Master signals completion of its dispatching phase (i.e. when it has swept through the first $\sqrt{N}$ numbers.)

Fig. 8 shows *just one* instance of load balancing. At time 50 processor 1 takes over half the load of processor 2 thereby cutting off half the long 'tail' in Fig. 6. This load balancing step is executed by each processor as it becomes idle. (Fig. 8 shows just one load balancing step to avoid a congested diagram.)

There is a stiff cost for load balancing. This arises because each processor's limit and current location must be accessible to every other processor and thus must be stored in shared memory. This is in contrast with the basic algorithm (Fig. 3) where we can keep each processors location ('place') and limit ('local_limit') in registers and thus obtain very high rates of memory access. The penalty for load balancing shows up prominently in Fig. 9 which plots the observed run times for the basic and the load balancing algorithm. There are 15 runs for each of the two algorithms in this figure. It can be seen that the run time of the load balancing algorithm is initially higher than the basic algorithm but that the difference decreases as the number of processors increases. After about 10 processors, the benefit of load balancing exceeds the cost of using shared memory and the running time of the new algorithm drops below that of the old.

It is also noteworthy that there is a significant 'spread' of run times for all numbers of processors greater than 1. This is because the load balancing algorithm has greater parallelism with attendant

uncertainty than the basic algorithm. This is shown in detail in Fig. 10 which shows portions of plots for 0.5, 1 and 2 Million numbers.

7. Comparison of Observed and Predicted Timings

In order to verify our understanding of the various factors that influence the performance of the basic algorithm, we first timed several fundamental performance parameters of the Flex and then ran a simulation of the parallel Sieve. We felt that the extent to which simulated results agreed with actual experiments would indicate to us how well we understood the performance of the Flex.

Our timing experiments yielded the following numbers. Time to access a location in shared memory in a 'for' loop is 10 microseconds. The time for one process to signal another is 2 milliseconds (this is currently implemented in software.) The time to spawn a process is 13 milliseconds. This last number appears excessive but has minor impact on our Sieve experiments because we spawn each slave only once during the lifetime of the program.

These performance characteristics were used to develop a simulation of the basic algorithm. We also used the memory access times to obtain lower bounds on the time required to sieve. This was done by running a serial sieve algorithm, counting the number of times any memory location was referenced, and dividing by the number of processors.

Fig. 11 compares the simulated timings and lower bounds with the averages of 15 runs each of the basic algorithm and of the load-balancing algorithm. These curves are for a problem of size 1 Million. The measured timings shown here are averages of the timings shown in Fig. 9. The following sub-sections summarize our results.

7.1. Comparison with Simulated results

It is seen that the simulated and observed run times of the basic algorithm agree completely for $p>7$. For $1<p<7$ the simulation times are less than the observed times. This is due to the fact that the

simulation does not take into account the time-sharing of the master with one of the slaves. The overhead of process switching accounts for the gap between pairs of curves. This gap is maximum for p=1 and decreases smoothly as the number of processors increases. This is because as the number of processors increases the disruption caused by process switching on one processor becomes a smaller and smaller fraction of the total work done by all processors.

7.2. A lower bound on time to sieve

The lowermost curve in Fig. 11 plots the lower bound on time to sieve 1 million numbers. This curve was obtained by counting the number of accesses to memory by a serial sieve algorithm and dividing by the number of processors. Clearly, no parallel algorithm can do better than this lower bound. It can be seen that this curve is very close to the simulation curve at p=1 but that the two diverge as the number of processors increases. This is, of course, because the simulation takes signaling and spawning time into account while the lower bound only counts the number of memory accesses. Beyond p=6 the simulation curve flattens out because of the lack of parallelism.

Comparing the lower bound with the load balancing curve shows that the two have roughly the same shape. At p=1 the load balancing algorithm is about 1.65 times the lower bound. This factor increases smoothly to 2.4 for p=17. The initial difference at p=1 arises because the lower bound assumes register indexed memory access while the load balancing algorithm uses indirection through a shared memory location to access the shared array. The overhead of task switching also contributes to this factor, as it does in the non load balancing algorithm. This factor increases because of the increasing overhead of load balancing as the number of processors increases.

It may be possible to implement a more sophisticated load balancing algorithm which uses registers for the initial computation and shared memory locations during a second phase. Such an algorithm would be able to approach the lower bound more closely.

8. Conclusions

We have described an implementation of the Sieve of Eratosthenes on the Flex/32 Multiprocessor and discussed its performance as a function of the number of processors. Our basic algorithm does not show any speedup beyond 6 processors. The more complex load balancing algorithm is initially slower but beats the basic algorithm after about 10 processors. We have also compared the performance of the basic algorithm with a simulation. There is good agreement between the two, indicating that we understand the machine's performance well enough to predict running times with fair accuracy.

We also compared the performance plots with a plot of the theoretical lower bound to sieve a range of numbers. The proximity of the experimental run times and the theoretical bound is a measure of the machine's efficiency.

We expect that this parallel algorithm for the Sieve is a useful test of some aspects of multiprocessors performance. Its implementation on other commercially available processors will provide interesting comparisons.

9. Acknowledgements

I am indebted to R. G. Voigt for his unceasing encouragement of this research through some very difficult times. T. Crockett, G. Wright and S. Bostic were generous with their time and patience in helping me use the Flex. Discussions with J. H. Saltz, D. M. Nicol and H. Jordan were very useful and V. K. Naik continued in his role as C reference of last resort.

10. References

[1] J. R. Edwards and G. Hartwig, "Benchmarking the Clones," *Byte*, Vol. 10 , No. 11, pp. 195-201, November 1985.

[2] D. Hawkins, "Mathematical Sieves," *Scientific American*, Vol. 199, No. 6, pp. 105-112, December 1958.

[3] H. F. Jordan, "Structuring parallel algorithms in an MIMD shared memory environment," *Parallel Computing*, Vol. 3, No. 2, pp. 93-110, May 1986.

[4] N. Matelan, "The Flex/32 MultiComputer," *Proc. 12th International Symposium on Computer Architecture*, pp. 209-213.

[5] G. A. Miller, *Historical Introduction to Mathematical Literature*, New York: Macmillan, 1916.

[6] M. L. D'Ooge (translator), *Nicomachus of Gerasa: Introduction to Arithmetic*, New York: Macmillan, 1926.

[7] D. A. Patterson, "A performance evaluation of the Intel 80286," *Computer Architecture News*, (ACM Sigarch newsletter), Vol. 10, No. 5, pp. 16-18, September 1982.

[8] E. Shapiro, "Concurrent Logic-programming Techniques," tutorial presented at the 1985 International Conference on Parallel Processing, St. Charles, Illinois, August 23, 1985.

[9] D. E. Smith, *History of Mathematics*, Vol. 1, Boston: Ginn, 1923.

List of Figure Captions

1 Sieving the First 50 numbers. The original list (first column) contains all integers from 2 to 50. The first sweep removes all multiples of 2; the second sweep removes all multiples of 3 and so on. It is not necessary to continue the sweeping process beyond $\sqrt{100}$ as explained in the text. The numbers that remain in the last column are the primes.

2 Architecture of the Flex/32 Multiprocessor.

3 Basic parallel algorithm for the Sieve. Variables in shared memory have been given all uppercase names for convenience. Access to these variables is via LOCK/UNLOCK commands whenever the possibility of synchronization problems exists. This is not shown in the above listing for clarity. Synchronization is not, however, required when accessing LIST, since all slaves are writing the same value (' ').

4 Execution of the Parallel Sieve algorithm for N=100, p=3. Time advances from top to bottom. Entries that begin with 'M' represent the Master.
'M[1,i]' represents the master signaling processor 1 to start its initialization phase.
'M[1,5]' is the master signaling processor 1 to start sweeping with 5.
'M[-,8]' represents the master finding the number 8 already crossed out.
All other entries are of the form 'number being swept:location examined'.
All memory and signaling operations are assumed to take one unit of time. Spawning time is not shown in this figure. '—' represents an idle processor. Some sequences of entries (indicated '- - -') have been omitted for brevity.

5 When master proceeds very rapidly compared with slaves, there is the possibility of "false primes". In the figure some slaves are needlessly striking off multiples of 4, 6 and 8. This arises because the signaling time in this example is 5 times the memory access time. In general, this can occur whenever the master progresses faster than a slave. We minimize these "false primes" by always time-sharing the master with one of the slaves.

6 Execution of parallel Sieve for N=200, p=7, illustrating poor utilization of processors. After time=104 only one processor is active. ' — ' represents an idle processor. Some sequences of entries have been omitted for brevity.

7 Observed timings for the Basic algorithm, N=0.5, 1, and 2 Million.

8 Illustration of load balancing in the modified algorithm. At time 50, Processor 1 takes over half of the work on the most heavily loaded Processor, 2. To avoid a congested diagram only one load balancing step is shown; in the actual algorithm each idle processor would take over half the heaviest processor's load. For example, Processor 7 would take over half of Processor 3's load at time 63.

9 Comparison of observed timings of Basic and Load balancing algorithm for N=1M.

10 Detail of plots of observed timings of Basic and Load Balancing algorithms for N=0.5M, 1M, 2M.

11 Averages of observed timings of the two algorithms for N=1M compared with simulation and lower bound.

	2	3	5	7
2	2	2	2	2
3	3	3	3	3
4				
5	5	5	5	5
6				
7	7	7	7	7
8				
9	9			
10				
11	11	11	11	11
12				
13	13	13	13	13
14				
15	15			
16				
17	17	17	17	17
18				
19	19	19	19	19
20				
21	21			
22				
23	23	23	23	23
24				
25	25	25		
26				
27	27			
28				
29	29	29		
30				
31	31	31	31	31
32				
33	33			
34				
35	35	35		
36				
37	37	37	37	37
38				
39	39			
40				
41	41	41	41	41
42				
43	43	43	43	43
44				
45	45			
46				
47	47	47	47	47
48				
49	49	49	49	
50				

Fig. 1 Sieving the First 50 numbers. The original list (first column) contains all integers from 2 to 50. The first sweep removes all multiples of 2; the second sweep removes all multiples of 3 and so on. It is not necessary to continue the sweeping process beyond $\sqrt{50}$ as explained in the text. The numbers that remain in the last column are the primes.

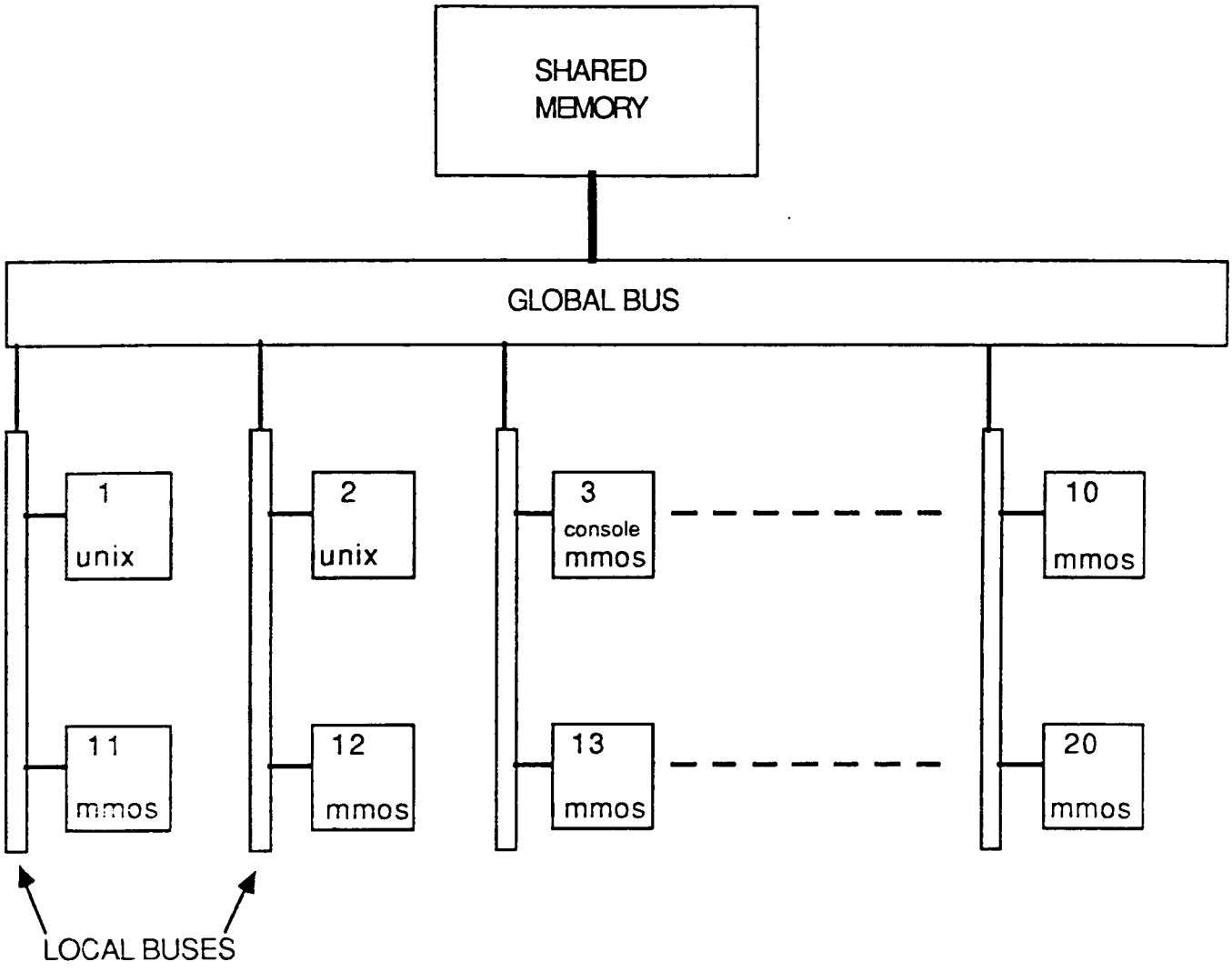

Fig. 2 Architecture of the Flex/32 Multiprocessor.

```
shared char LIST[limit];              /* These variables are in shared memory          */
shared int  ACTIVE[maxprocs];         /* limit is the range of numbers to be sieved    */
shared int  PRIME[maxprocs];          /* maxprocs is the number of processors to be used */
shared int  DONE;

main()                                /* the master procedure                          */
{
    DONE=false;
    for (processor=0;processor<maxprocs;processor++) {
        ACTIVE[processor]=false;
        process(slave(processor,limit),processor);
                                      /* starts up routine 'slave' on specified processor */
    }
    rootlimit=sqrt(limit);
    mainplace=1;
    when(all processors have finished initialization);
                                      /* causes processor to wait until condition satisfied */
    do {
        do mainplace++;
        while ((LIST[mainplace]==' ') && (mainplace<rootlimit));
        if (mainplace>=rootlimit)
            DONE=true;
        else {
             when(a processor is idle);
             PRIME[processor]=mainplace;
             ACTIVE[processor]=true;
        }
    }
    while (DONE==false);
    when(all processors have finished their ongoing sweeps);
}

slave(processor,limit)                /* a copy of this procedure runs on each processor */
int processor,limit;
{ register int stepsize, place, local_limit;
    initialization: compute start and stop points;
    for (place=start;place<=stop;place++)
        LIST[place]='*';              /* initialization */
    signal completion of initialization to master;
    do {
        when (ACTIVE[processor]==true);

        stepsize=PRIME[processor];
        place=stepsize+stepsize;
        local_limit=limit;            /*limit is copied into a register for speed */
        while (place<=local_limit) {
            LIST[place]=' ';          /* sieving      */
            place=place+stepsize;
        }
        signal completion of sweep to master;
    }
    while (DONE==false);
}
```

Fig.3 Basic parallel algorithm for the Sieve. Variables in shared memory have been given all upper-case names for convenience. Access to these variables is via LOCK/UNLOCK commands whenever the possibility of synchronization problems exists. This is not shown in the above listing for clarity. Synchronization is not, however, required when accessing LIST, since all slaves are writing the same value (' ').

Time	P1	P2	P3	Comments
1	M[2,i]	--	—	Master signals P2
2	M[3,i]	2: 1	—	Master signals P3, P2 starts
3	M[1,i]	2: 2	3: 35	Master signals P1; P3 starts
4	1: 69	2: 3	3: 36	Master idle; P1 starts
5	1: 70	2: 4	3: 37	
6	1: 71	2: 5	3: 38	
- - -	- - - -	- - - -	- - - -	
34	1: 99	2: 33	3: 66	
35	1:100	2: 34	3: 67	
36	—	—	3: 68	Init. finished; master informed
37	—	—	—	
38	M[2,2]	—	—	Master signals P2 to sweep 2;
39	M[3,3]	2: 4	—	Master signals P3 to sweep 3; P2 starts
40	M[-,4]	2: 6	3: 6	Master finds 4 crossed out ; P3 Starts
41	M[1,5]	2: 8	3: 9	Master signals P1 to sweep 5;
42	—	2: 10	3: 12	Master now time-shared with sweep 5;
43	M[-,6]	2: 12	3: 15	Master finds 6 crossed out ;
44	5: 10	2: 14	3: 18	P1 starts sweep 5;
45	5: 15	2: 16	3: 21	Master blocked:no processor idle
- - -	- - - -	- - - -	- - - -	
61	5: 95	2: 48	3: 69	
62	5:100	2: 50	3: 72	P1 finishes sweep 5 and signals master
63	—	2: 52	3: 75	
64	M[1,7]	2: 54	3: 78	Master signals P1 to sweep 7
65	—	2: 56	3: 81	
66	M[-,8]	2: 58	3: 84	Master now time-shared with sweep 7
67	7: 14	2: 60	3: 87	P1 starts sweep 7
68	M[-,9]	2: 62	3: 90	Master finds 9 crossed out
69	7: 21	2: 64	3: 93	
70	M[-,10]	2: 66	3: 96	Master finishes, having reached $\sqrt{100}$
71	7: 28	2: 68	3: 99	P3 finishes sweep 3
72	7: 35	2: 70	—	
73	7: 42	2: 72	—	
- - -	- - - -	- - - -	- - - -	
81	7: 98	2: 98	—	P1 finishes sweep 7
82	—	2: 90	—	
83	—	2: 92	—	
84	—	2: 94	—	
85	—	2: 96	—	
86	—	2: 98	—	
87	—	2:100	—	P2 finishes sweep 2
88	—	—	—	Program terminates

Fig.4 Execution of the Parallel Sieve algorithm for N=100, p=3. Time advances from top to bottom. Entries that begin with 'M' represent the Master.
'M[1,i]' represents the master signalling processor 1 to start its initialization phase.
'M[1,5]' is the master signalling processor 1 to start sweeping with 5.
'M[-,8]' represents the master finding the number 8 already crossed out.
All other entries are of the form 'number being swept:location examined'.
All memory and signalling operations are assumed to take one unit of time. Spawning time is not shown in this figure. '—' represents an idle processor. Some sequences of entries (indicated '- - -') have been omitted for brevity.

Time	P1	P2	P3	P4	P5	P6	P7
31	M[2 , 2]	—	—	—	—	—	—
32	M[3 , 3]	—	—	—	—	—	—
33	M[4 , 4]	—	—	—	—	—	—
34	M[5 , 5]	—	—	—	—	—	—
35	M[6 , 6]	—	—	—	—	—	—
36	M[7 , 7]	2: 4	—	—	—	—	—
37	M[1 , 8]	2: 6	3: 6	—	—	—	—
38	—	2: 8	3: 9	4: 8	—	—	—
39	M[- , 9]	2: 10	3: 12	4: 12	5: 10	—	—
40	—	2: 12	3: 15	4: 16	5: 15	6: 12	—
41	M[- , 10]	2: 14	3: 18	4: 20	5: 20	6: 18	7: 14
42	—	2: 16	3: 21	4: 24	5: 25	6: 24	7: 21
43	—	2: 18	3: 24	4: 28	5: 30	6: 30	7: 28
44	—	2: 20	3: 27	4: 32	5: 35	6: 36	7: 35
45	8:16	2: 22	3: 30	4: 36	5: 40	6: 42	7: 42
46	8:18	2: 24	3: 33	4: 40	5: 45	6: 48	7: 49

Fig. 5 When master proceeds very rapidly compared with slaves, there is the possibility of "false primes". In the figure some slaves are needlessly striking off multiples of 4, 6 and 8. This arises because the signalling time in this example is 5 times the memory access time. It general, this can occur whenever the master progresses faster than a slave. We minimize these "false primes" by always time-sharing the master with one of the slaves.

Time	P1	P2	P3	P4	P5	P6	P7
39	M[- , 4]	2: 6	3: 6	—	—	—	—
40	M[4, 5]	2: 8	3: 9	—	—	—	—
41	M[- , 6]	2: 10	3: 12	5: 10	—	—	—
42	M[5, 7]	2: 12	3: 15	5: 15	—	—	—
43	M[- , 8]	2: 14	3: 18	5: 20	7: 14	—	—
44	M[- , 9]	2: 16	3: 21	5: 25	7: 21	—	—
45	M[- ,10]	2: 18	3: 24	5: 30	7: 28	—	—
46	M[6,11]	2: 20	3: 27	5: 35	7: 35	—	—
47	M[- ,12]	2: 22	3: 30	5: 40	7: 42	11: 22	—
48	M[7,13]	2: 24	3: 33	5: 45	7: 49	11: 33	—
49	M[- ,14]	2: 26	3: 36	5: 50	7: 56	11: 44	13: 26
50	—	2: 28	3: 39	5: 55	7: 63	11: 55	13: 39
51	—	2: 30	3: 42	5: 60	7: 70	11: 66	13: 52
60	—	2: 48	3: 69	5:105	7:133	11:165	13:169
61	—	2: 50	3: 72	5:110	7:140	11:176	13:182
62	—	2: 52	3: 75	5:115	7:147	11:187	13:195
63	—	2: 54	3: 78	5:120	7:154	11:198	—
64	—	2: 56	3: 81	5:125	7:161	—	—
65	—	2: 58	3: 84	5:130	7:168	—	—
66	—	2: 60	3: 87	5:135	7:175	—	—
67	—	2: 62	3: 90	5:140	7:182	—	—
68	—	2: 64	3: 93	5:145	7:189	—	—
69	—	2: 66	3: 96	5:150	7:196	—	—
70	—	2: 68	3: 99	5:155	—	—	—
71	—	2: 70	3:102	5:160	—	—	—
79	—	2: 86	3:126	5:200	—	—	—
80	—	2: 88	3:129	—	—	—	—
102	—	2:132	3:195	—	—	—	—
103	—	2:134	3:198	—	—	—	—
104	—	2:136	—	—	—	—	—
105	—	2:138	—	—	—	—	—
129	—	2:186	—	—	—	—	—
130	—	2:188	—	—	—	—	—
131	—	2:190	—	—	—	—	—
132	—	2:192	—	—	—	—	—
133	—	2:194	—	—	—	—	—
134	—	2:196	—	—	—	—	—
135	—	2:198	—	—	—	—	—
136	—	2:200	—	—	—	—	—
137	—	—	—	—	—	—	—

Fig. 6 Execution of parallel Sieve for N=200, p=7, illustrating poor utilization of processors. After time=104 only one processor is active. ' — ' represents an idle processor. Some sequences of entries have been omitted for brevity.

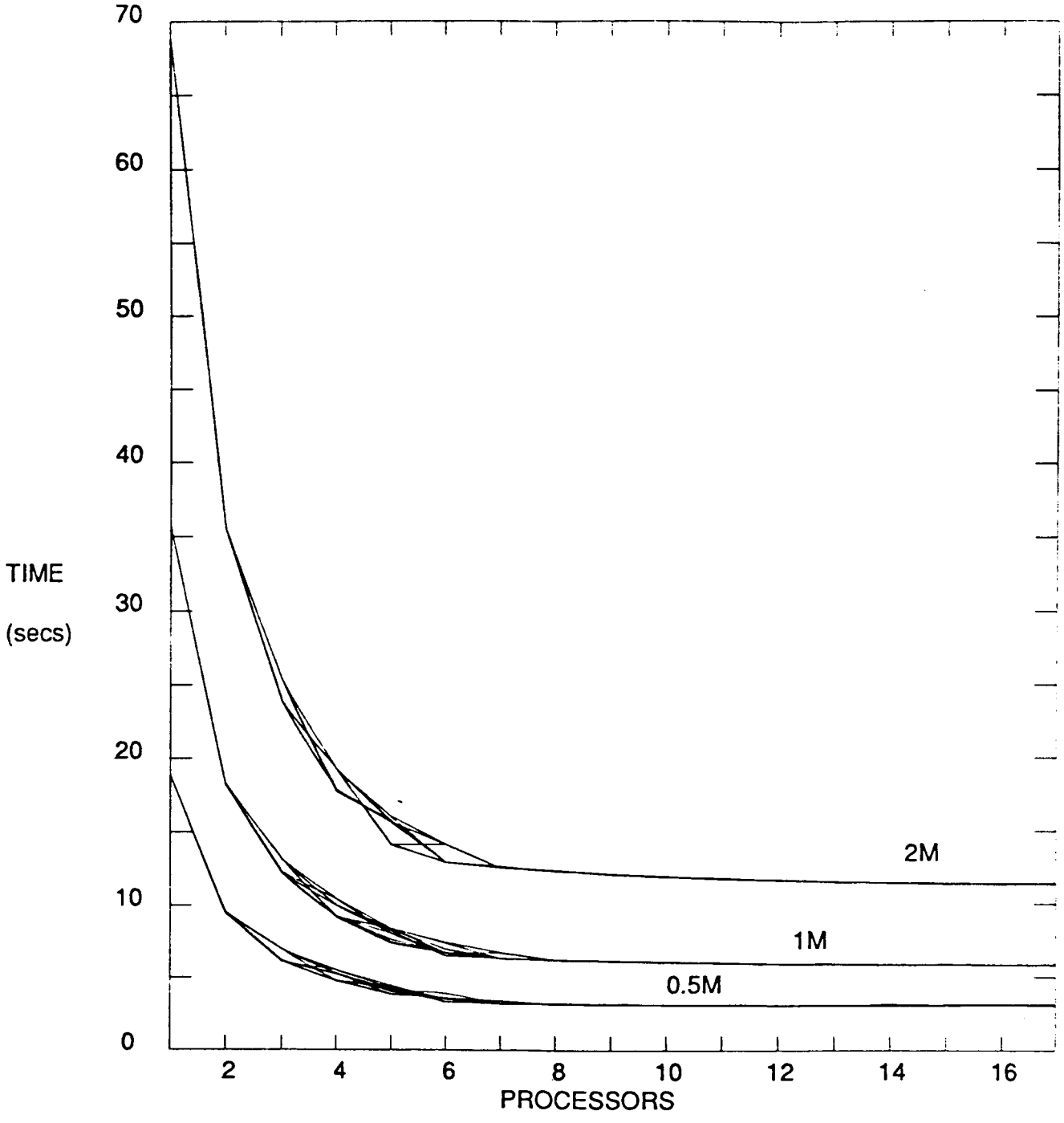

Fig. 7 Observed timings for the Basic algorithm, N=0.5, 1, and 2 Million.

P0	P1	P2	P3	P4	P5	P6	P7
39	M[- , 4]	2: 6	3: 6	—	—	—	—
40	M[4, 5]	2: 8	3: 9	—	—	—	—
41	M[- , 6]	2: 10	3: 12	5: 10	—	—	—
42	M[5, 7]	2: 12	3: 15	5: 15	—	—	—
43	M[- , 8]	2: 14	3: 18	5: 20	7: 14	—	—
44	M[- , 9]	2: 16	3: 21	5: 25	7: 21	—	—
45	M[- ,10]	2: 18	3: 24	5: 30	7: 28	—	—
46	M[6,11]	2: 20	3: 27	5: 35	7: 35	—	—
47	M[- ,12]	2: 22	3: 30	5: 40	7: 42	11: 22	—
48	M[7,13]	2: 24	3: 33	5: 45	7: 49	11: 33	—
49	M[- ,14]	2: 26	3: 36	5: 50	7: 56	11: 44	13: 26
50	—	2: 28	3: 39	5: 55	7: 63	11: 55	13: 39
51	2: 116	2: 30	3: 42	5: 60	7: 70	11: 66	13: 52
52	2: 118	2: 32	3: 45	5: 65	7: 77	11: 77	13: 65
53	2: 120	2: 34	3: 48	5: 70	7: 84	11: 88	13: 78
54	2: 122	2: 36	3: 51	5: 75	7: 91	11: 99	13: 91
55	2: 124	2: 38	3: 54	5: 80	7: 98	11:110	13:104
56	2: 126	2: 40	3: 57	5: 85	7:105	11:121	13:117
57	2: 128	2: 42	3: 60	5: 90	7:112	11:132	13:130
58	2: 130	2: 44	3: 63	5: 95	7:119	11:143	13:143
59	2: 132	2: 46	3: 66	5:100	7:126	11:154	13:156
60	2: 134	2: 48	3: 69	5:105	7:133	11:165	13:169
61	2: 136	2: 50	3: 72	5:110	7:140	11:176	13:182
62	2: 138	2: 52	3: 75	5:115	7:147	11:187	13:195
63	2: 140	2: 54	3: 78	5:120	7:154	11:198	—
64	2: 144	2: 56	3: 81	5:125	7:161	—	—
65	2: 146	2: 58	3: 84	5:130	7:168	—	—
66	2: 148	2: 60	3: 87	5:135	7:175	—	—
67	2: 150	2: 62	3: 90	5:140	7:182	—	—
68	2: 152	2: 64	3: 93	5:145	7:189	—	—
69	2: 154	2: 66	3: 96	5:150	7:196	—	—
70	2: 156	2: 68	3: 99	5:155	—	—	—
71	2: 158	2: 70	3:102	5:160	—	—	—
72	2: 160	2: 72	3:105	5:165	—	—	—
73	2: 162	2: 74	3:108	5:170	—	—	—
74	2: 164	2: 76	3:111	5:175	—	—	—
75	2: 166	2: 78	3:114	5:180	—	—	—
76	2: 168	2: 80	3:117	5:185	—	—	—
77	2: 170	2: 82	3:120	5:190	—	—	—
78	2: 172	2: 84	3:123	5:195	—	—	—
79	2: 174	2: 86	3:126	5:200	—	—	—
80	2: 176	2: 88	3:129	—	—	—	—
81	2: 178	2: 90	3:132	—	—	—	—
90	2: 196	2: 108	3:159	—	—	—	—
91	2: 198	2: 110	3:162	—	—	—	—
92	2: 200	2: 112	3:165	—	—	—	—
93	—	2: 114	3:168	—	—	—	—
94	—	—	3:171	—	—	—	—
95	—	—	3:174	—	—	—	—

Fig. 8 Illustration of load balancing in the modified algorithm. At time 50, Processor 1 takes over half of the work on the most heavily loaded Processor, 2. To avoid a congested diagram only one load balancing step is shown; in the actual algorithm each idle processor would take over half the heaviest processor's load. For example, Processor 7 would take over half of Processor 3's load at time 63.

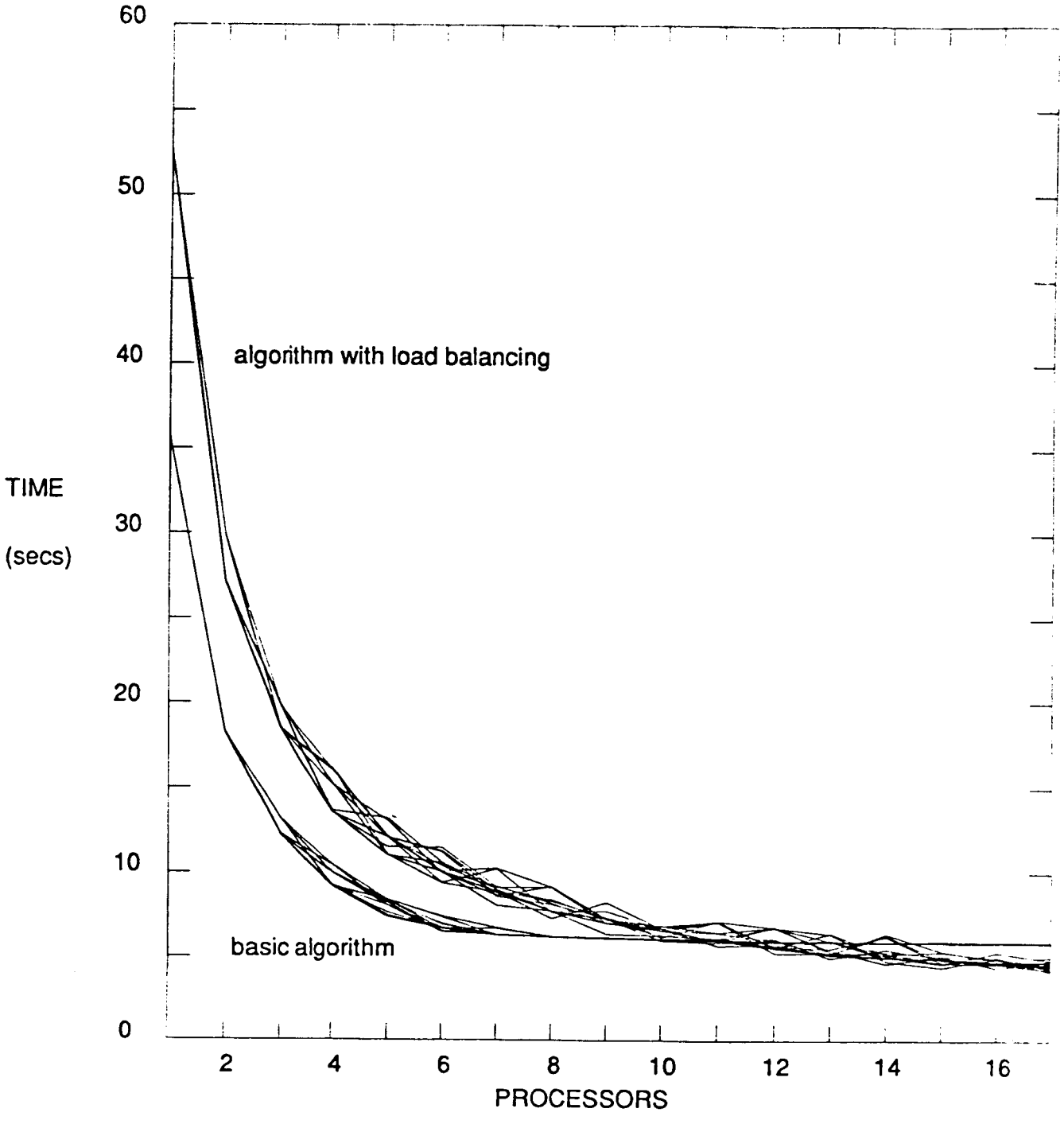

Fig. 9 Comparison of observed timings of Basic and Load balancing algorithm for N=1M.

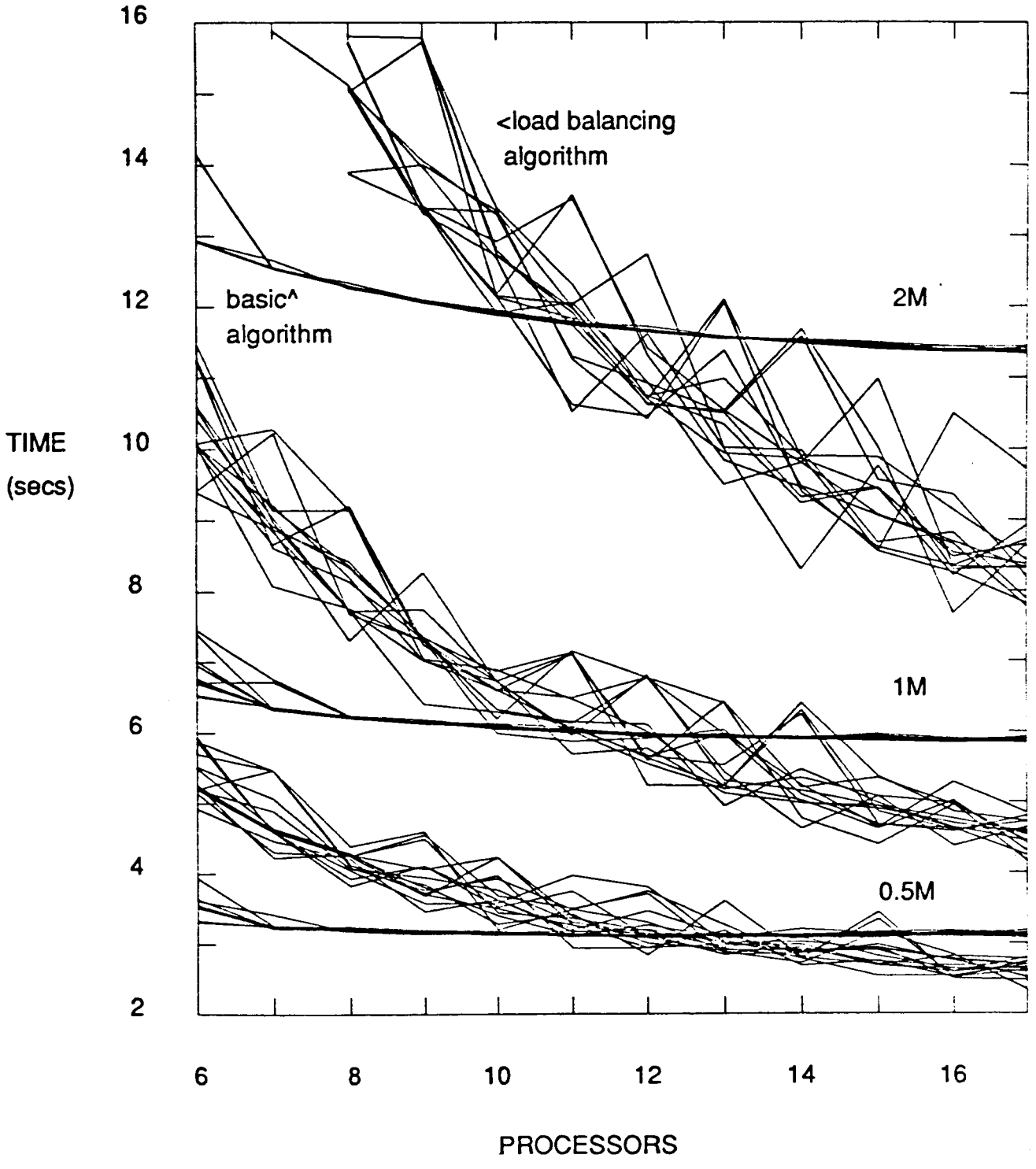

Fig. 10 Detail of plots of observed timings of Basic and Load Balancing algorithms for N=0.5M, 1M, 2M.

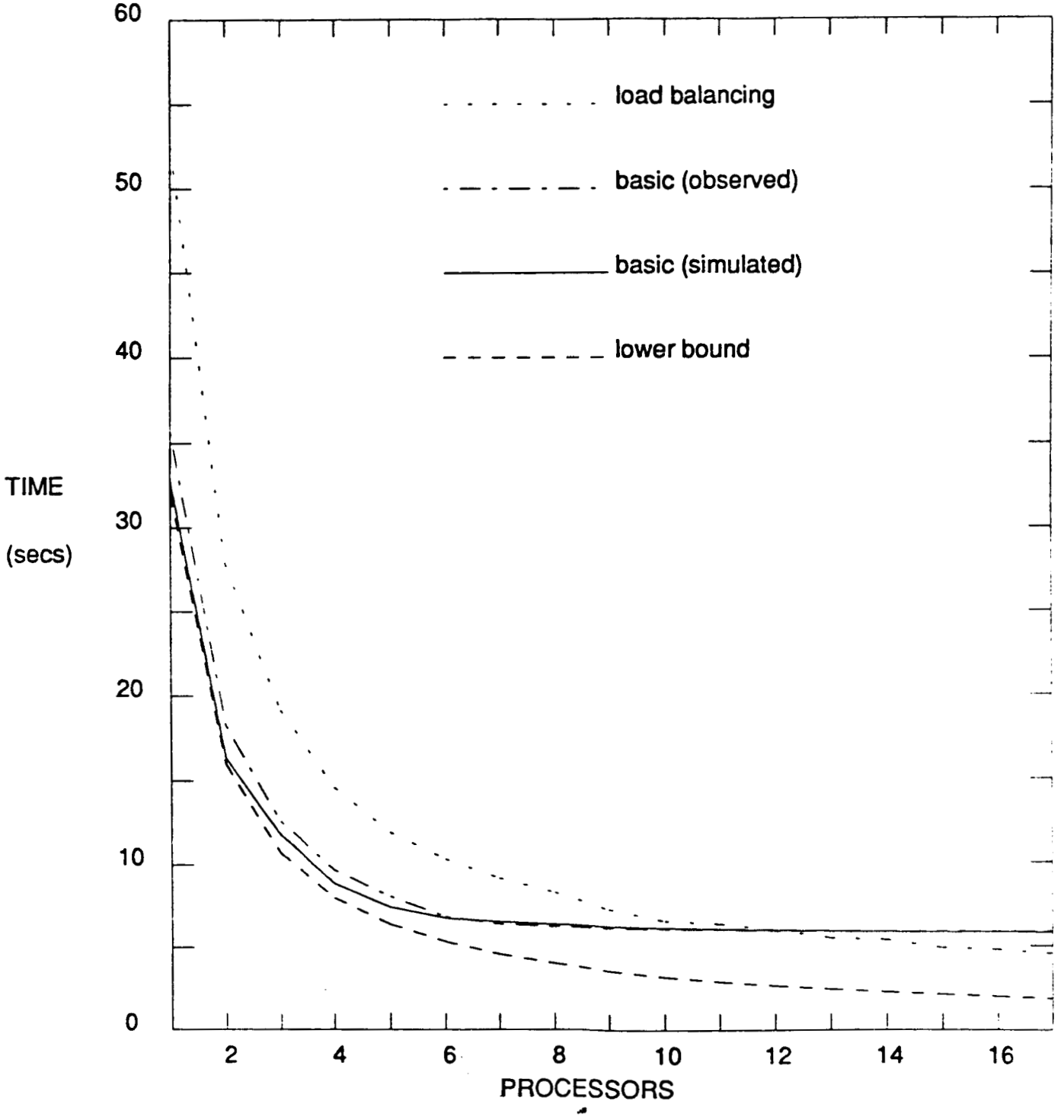

Fig. 11 Averages of observed timings of the two algorithms for N=1M compared with simulation and lower bound.

Standard Bibliographic Page

1. Report No. NASA CR-178131 ICASE Report No. 86-41	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle MULTIPROCESSING THE SIEVE OF ERATOSTHENES		5. Report Date June 1986 6. Performing Organization Code
7. Author(s) Shahid Bokhari		8. Performing Organization Report No. 86-41
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report 14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes

Langley Technical Monitor: J. C. South

Submitted to IEEE Computer

Final Report

16. Abstract The well known Sieve of Eratosthenes for finding prime numbers dates back to about 200 B.C. In recent years it has seen much use as a benchmark algorithm for serial computers while its intrinsically parallel nature has gone largely unnoticed.

We describe the implementation of a parallel version of this algorithm for a real parallel computer, the Flex/32, and discuss its performance. It is shown that the algorithm is sensitive to several fundamental performance parameters of parallel machines, such as spawning time, signaling time, memory access, and overhead of process switching. Because of the nature of the algorithm, it is impossible to get any speedup beyond 4 or 5 processors unless some form of dynamic load balancing is employed. We describe the performance of our algorithm with and without load balancing and compare it with theoretical lower bounds and simulated results.

It is straightforward to understand this algorithm and to check the final results. However, its efficient implementation on a real parallel machine requires thoughtful design, especially if dynamic load balancing is desired. The fundamental operations required by the algorithm are very simple: this means that the slightest overhead appears prominently in performance data. The Sieve thus serves not only as a very severe test of the capabilities of a parallel processor but is also an interesting challenge for the programmer.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
multiprocessing, shared-memory machines, Sieve of Eratosthenes, prime numbers, load balancing	60 - Computer Operations and Hardware 61 - Computer Programming Software Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	26	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)