2

CONF-890892--1

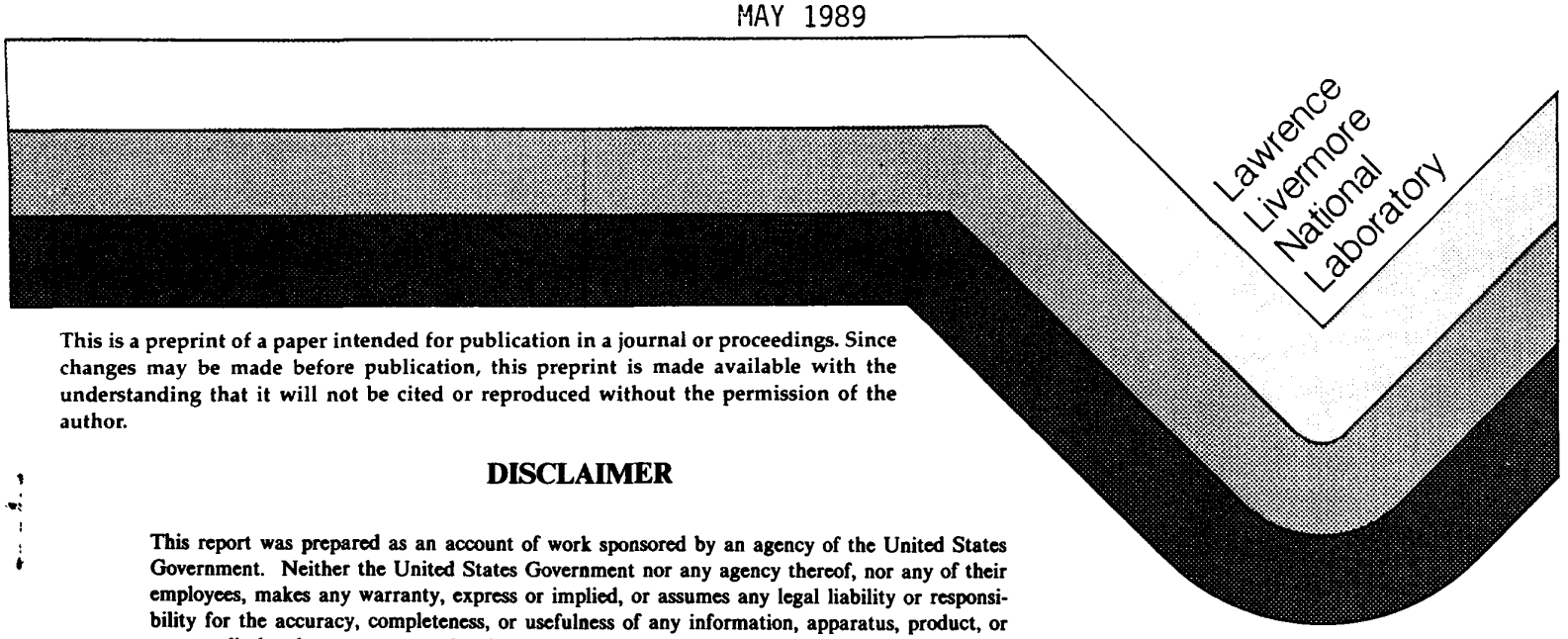
UCRL- 101085
PREPRINT

RUN-TIME SUPPORT FOR PARALLEL FUNCTIONAL PROGRAMMING ON SHARED MEMORY MULTIPROCESSORS

Ching-Cheng Lee
H.A. Fatmi

THIS PAPER WAS PREPARED FOR SUBMITTAL TO
12th INTERNATIONAL CONGRESS ON CYBERNETICS
NAMUR, BELGIUM
AUGUST 21-25, 1989

MAY 1989



Run-Time Support for Parallel Functional Programming on Shared-Memory Multiprocessors

Ching-Cheng Lee *
H. A. Fatmi **

UCRL--101085
DE89 013790

Abstract

The use of functional languages for parallel computing has been proposed for many years. Many functional languages were developed along with the design of new architectures such as data flow and reduction machines [1,2,3,7,8,10,11]. In this paper, we present a general model of a run-time system for a parallel functional language called SISAL [13] to be executed on shared-memory multiprocessors. The implementation of this run-time system is examined on two radically different architectures; i.e., a 32-way (symmetrical) Vax Research Multiprocessor M31 [15] and a 4-way Cray X-MP system. In order to properly evaluate the effectiveness of SISAL on shared-memory multiprocessors, we suggest exploring an interactive visual control mechanism that dynamically show run-time behavior in future research.

1. Introduction

Ultrahigh speed computations for complex numerical and nonnumerical problems simulate our human intelligence, such as learning, understanding, reasoning, and problem solving. The physical constraints of hardware has made parallel computing the only solution that meets the demands of high-speed computation. To enhance parallel computing, the development of effective ways to program high speed computations is equally important. Conventional language programs are made of sequences of statements that alter the values of variables in memory. This makes it complicated for both programmers and automatic-analysis software to discover which program segments can safely execute in parallel. One of the better approaches to programming parallel computers today is to use functional languages. In a functional language, a program is composed of a set of function definitions that describe the computations without any side effect (caused by an assignment), and only data dependencies constrain the order of execution. This makes the details of the underlying architectures transparent to the user and allows the compiler to easily detect and exploit the parallelisms of the underlying architectures and in the programs.

---

* LLNL, Livermore, CA 94550 USA. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. This is work in partial fulfillment for the Ph.D. degree at King's College, University of London.

** King's College, University of London, Strand, London WC2R 2LS UK

Traditionally, functional languages were developed simultaneously with the design of new architectures such as data flow and reduction machines [1,2,3,7,8,10,11], most of which realized parallelism at the finest granularity level (single instructions). A coarser granularity is appropriate for more conventional multiprocessor systems. In this paper, we present a general run-time system designed to support a parallel functional language called SISAL (Streams and Iteration in a Single Assignment Language) [13] to be executed on conventional shared-memory multiprocessors. SISAL was derived primarily from the data flow language VAL[1], but unlike VAL it is developed for conventional sequential machines, shared-memory multiprocessors, and vector processors, as well as the Manchester data flow machine. The work on SISAL was collaborated research between the University of Manchester, Lawrence Livermore National Laboratory, Digital Equipment Corporation, and Colorado State University.

The general issues of mapping functional language programs onto an arbitrary multiprocessor has been addressed by Hudak [9]. The run-time support described in this paper is an extension of the previous run-time system developed for the Sequent Balance multiprocessor [14], which was redesigned to make it portable on general shared-memory multiprocessor systems. The design started with the implementation on Clustered Vax 11/784 [4]. The current version is now running on many different multiprocessor systems, including the (symmetrical) Vax Research Multiprocessor M31, and the Cray X-MP. The initial performance evaluations of this run-time system on the (asymmetrical) Vax Research multiprocessor M31 and the Cray X-MP have been discussed previously [12]. In this paper, we describe the internal workings of the run-time system with their implementations on the (symmetrical) Vax Research Multiprocessor M31 and the Cray X-MP in more detail. In addition, one experiment is conducted to show the effectiveness of parallelism on both the M31 and Cray X-MP.

2. The Run-time Model

2.1 Task management

The environment of the run-time system supports a large number of concurrent instruction streams constituting a single SISAL program. We call such an instruction stream a task. In SISAL, a task defines a function call, a parallel loop slicing, and a stream producing/consuming sequential loop. The code generator determines this partition and generates the calls to the run-time system. To support the above parallelism, the task management has been designed as a user-level, run-time library for task scheduling and control. This reduces the significant amount of state information maintained for a task since the current operating-system state never changes; only the processor state needs to be saved and restored.

In our run-time system, when a SISAL program is started, an operating system process called a worker is created for each processor that will be used. A worker either executes a task or executes a "busy wait" until a task is available to be executed. Without busy waiting, an idle worker would have to relinquish its processor to the operating system and significant overhead would occur. The following discussions give more details on the concept of a task and the role of a worker.

Tasks

A task is a basic execution unit in SISAL. When a task is created, a data structure called task control block (TCB) and a run-time stack that holds local variables and supports run-time library calls are allocated and initialized. The TCB defines the current state of an executing task, which includes a task's current processor state, extant child count, and execution status. The processor state defines the current program counter, argument pointer, register contents, etc. The extant child count defines the current number of nonterminated children belonging to a task and is incremented during task creation. The execution status identifies a task's current mode of operation: READY, RUNNING, VBLOCKED (Value BLOCKED), PBLOCKED (Producer BLOCKED), and CBLOCKED (Consumer BLOCKED). The last two modes are discussed in later sections. All tasks available for execution, but not yet executing, have an executing status identified as READY. The tasks are maintained on a global ready list in FIFO (First In First Out) order. An executing task has state RUNNING. When a RUNNING task requires the data produced by one of its extant children, it blocks with the status VBLOCKED until all the children tasks are completed. The following are some task management run-time routines:

- **GetStack:** Allocates and initializes a task descriptor; i.e., TCB when a task is to be created.
- **RListEnQ:** Adds the task to the ready queue.
- **RListDeQ:** Removes a task (if any) from the ready queue.
- **Schedule:** Performs a task context switch to run a newly selected task.
- **Sync:** Blocks the requesting task and waits for the children's completion. The invoking task will be VBLOCKED.
- **TermMe:** Terminates execution and decrements the parent's child count. If this is the last extant child, it will awaken a VBLOCKED parent task.

In general, an executing SISAL program defines a hierarchical tree of tasks. The root task, called the SISAL program initiator, creates the main SISAL task and its

required stream I/O tasks and triggers worker termination on completion of its children. Figure 1 summarizes the state transitions and the run-time routines that cause them, including those described in Section 2.3.

Workers

Before the SISAL program is executed, the run-time data structures are initialized and the operating system facilities are used to create the workers. In order to synchronize all the workers to start for parallel computation, all the workers start execution on a common barrier. Once all the processes arrive on this barrier, parallel execution begins safely. Each worker then attempts to acquire a READY task from the read list. If such a task exists, the old processor state is saved and that of the acquired task is installed. Otherwise, the worker waits busily. We refer to the act of saving and restoring processor states as *worker reassignment*. When a RUNNING task requests suspension by calling run time routine Sync, the newly freed worker attempts to acquire another task to execute. When a RUNNING task requests termination by calling run-time routine TermMe, its processor state is not saved during worker reassignment. Instead, its resources are released after the new processor state is installed. Thus, task execution is multiplexed based on voluntary suspension and termination requests. All worker processes thus repeat their normal scheduling pattern; i.e., busy waiting on the ready list and searching for work until shutdown occurs.

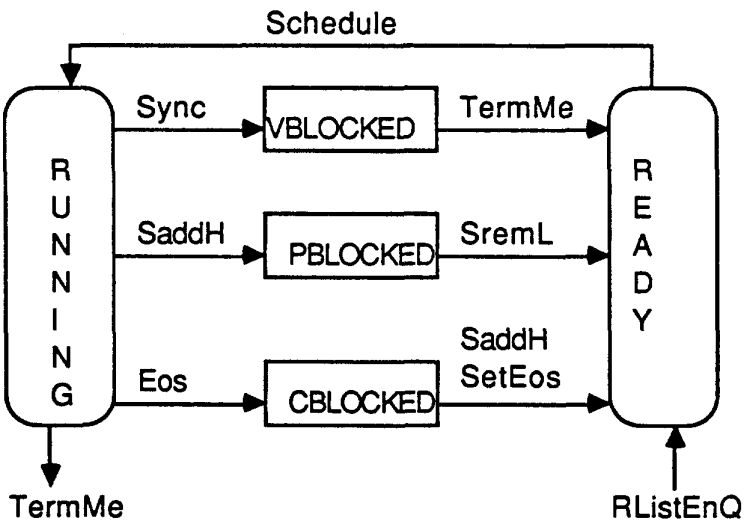


Figure 1: Task State Transitions

2.2 Loop Slicing

SISAL loops come in two forms: sequential and parallel. The parallel loop is a forall construct that includes an index range for each independent loop-body execution and thus provides the opportunity for parallel execution. In our run-time system, a routine called LoopSlicer is used to divide a SISAL forall loop into independent tasks for parallel execution. Each divided slice spans a continuous set of index values. Figure 2 summarizes the operation of LoopSlicer. The routine takes the entry point of the task defining the loop and the full index range of the loop as arguments. Dividing the total number of iterations by Slices, a run- time parameter determines slice thickness. The original SISAL loop is thus replaced by a call to LoopSlicer with the address of the loop code as its argument.

```
LoopSlicer (LoopAddress, LowIndex, HighIndex) {
      IndexRange = HighIndex - LowIndex;          /* determines the range of the index */
      SliceThickness = IndexRange/Slices;         /* determines each slice thickness */
      while (LowRange <= HighRange) {
            ThisHi = LowIndex + SliceThickness;   /* compute new high index for this slioce */
            NewTCB = GetStack(LoopAddress);       /* allocate new TCB for this slice task */
            NewTCB->Low = LowIndex;               /* record this task low and high index */
            NewTCB->High = ThisHi;
            RListEnQ(NewTCB);                     /* put this new task on READY queue */
            LowIndex = ThisHi + 1;                /* for next slice, start the new low index */
      }
}
```

Figure 2: LoopSlicer Source Code in Pseudo C.

2.3 Stream Parallelism

A SISAL stream is a data structure defining a possibly infinite sequence of homogeneous values. Streams differ from arrays in that the values can only be accessed sequentially. The compiler compiles the stream producing/consuming loops into functions that contain sequentially executing loops and allocates a buffer in memory for stream values. A producer of values may execute in parallel with consumers, and only a substream must exist at any point during execution. Streams are the only values in SISAL required to be nonstrict; i.e., the value can be used as soon it is available instead of waiting for the entire structure to be operated together. In constrast, arrays in SISAL must be completely constructed before any consumer can operate on it. To eliminate copies for side-effect-free semantics, reference counts are associated with each stream value. The following are some of the run-time routines that manage a stream:

- **SaddH**: This routine is called by a producer loop that adds data elements to a stream until it reaches MaxStreamSize, a run-time parameter. Then it becomes producer blocked; i.e, PBLOCKED (refer to Figure 1). This prevents a fast stream producer from exhausting memory. Once the consumed streams (by SremL routine described in the following) are below a producer threshold value, the blocked producer is awaken again.

- **SetEos**: This routine is called by a producer loop that sets the end of the stream mark so that a blocked consumer; i.e., CBLOCKED task, can be awaken (refer to Figure 1).

- **Eos**: This routine is called by a consumer loop that finds no more elements in the current stream and desires to know the current end-of-stream status of its stream instance. If the routine returns false, the calling consumer will become consumer blocked; i.e., CBLOCKED (refer to Figure 1).

- **SremL**: This routine is called by a consumer loop that implements a SISAL stream_rest operation. It returns a stream similar to the input stream with the first element removed. Instead of copying the stream, a reference count is used to indicate the usage. The consumer of a stream may proceed faster than its corresponding producer and discover that the stream is empty but the end of stream has not been reached. Such a consumer will block. The SaddH routine will awaken CBLOCKED tasks when a producer threshold value is reached (refer to Figure 1).

3. Implementations on Shared-Memory Machines

This section discusses the issues of implementing the run-time software on shared-memory machines. The major machine and operating system dependencies in our run-time software implementations are: process creation, synchronization, and shared-memory allocation.

3.1 Vax Research Multiprocessor (M31)

The M31 system is a large-scale multiprocessor machine for supporting research and experiments within Digital Equipment Corporation. The machine has 32 processors and run a single VMS (V5.0) operating system to support symmetrical (no master and slave) multiprocessing.

In M31, the creation of worker process is as follows: The VMS facility LIB$SPAWN can be used to spawn a subprocess to execute a defined command procedure on each individual processor. This command procedure runs the same program image as that of the spawner process and therefore the parallel execution becomes SPMD (Single Program Multiple Data stream) mode. All the worker processes are thus created. The synchronization among these workers is however via the use of shared memory described in the following paragraphs.

To allocate shared memory, the system requires that a global section in shared memory be explicitly created and mapped to a program's virtual memory space at run time. Creation of the global section involves two steps: (1) Open (by using the VMS RMS facility) a file for global section mapping, and (2) Invoke a

Create and Map Section ($CRMPSC) system call. After the creation of global section by the first worker, the global section will be mapped by subsequently arriving workers using the VMS global section mapping facility; i.e., $MGBLSC system call. When the program terminates, an exit handler is defined to delete the global section and to do the final cleanup.

3.2 Cray X-MP

The run-time system was implemented on Cray X-MP, running NLTSS (Network Livermore Time Sharing System). The NLTSS operating system currently has multiprocessing tasking library [6] to support user-level multiprocessing and implement the lower level machine/system dependent primitives for process management. The following are some of the Cray NLTSS tasking primitives that has been integrated into the SISAL run-time library.

- **tasktune:** Modifies tuning parameters that include maxcpu (maximum number of CPU) and other parameters to simulate busy waiting of the run-time behavior.
- **taskstart:** Creates a Cray task by allocating and initializing the task descriptor (TCB).
- **tqwait:** Implements the P operation of counting semaphore, which is called when the task is blocked.
- **tqsignal:** Implements the V operation of a counting semaphore, which is called when a task needs to wakeup a blocked task.

The SISAL implementation on the Cray has integrated these NLTSS tasking primitives into the run-time system. At the beginning of the job execution, the tasking kernel is initialized with the number of hardware processes needed for the job and the parameters that simulate the busy waiting of the scheduling. Since no hardware process has been created initially except the startup process, the subsequent calls of taskstart will create the additional worker processes. Since task creation is relatively expensive for the Cray, a run-time option has been designed to allow the user to specify an additional number of dummy tasks to be created at the begining along with the worker-process creations. These additional dummy tasks will all start on an execution entry, waiting on a binary semaphore. When a task spawn occurs during run time, these tasks can be awakened and tagged with additional context to be run. Although this scheme can save some task-creation overhead during the parallel execution of the job, it introduces some additional synchronization overhead such as signaling and queuing/dequeing operations.

The task-state transitions in the Cray implementation are as follows: If a task is VBLOCKED, PBLOCKED, or CBLOCKED, the worker will call TQWAIT and is

assigned to another ready task. On the other hand, the workers can wake up the value-blocked parent task, or stream-blocked producer/consumer via TQSIGNAL.

To implement the shared memory, all the processes within the same job share the same user memory space. The processes created by LIB$SPAWN start with the same calling process execution environment. One noticeable difference of shared memory from that of M31 is that the Cray does not have virtual memory and hence no memory mapping is involved in the memory referencing.

4. Performance Evaluations

To evaluate the effectiveness of SISAL multiprocessing, the algorithm of The Sieve of Eratosthenes was performed on the (symmetrical) VAX Research Multiprocessor M31 and the Cray X-MP/416. Because the Cray resource is rare, the experiments on the Cray were conducted in the time-shared environments. The following performance measurements were used:

T(n) = execution time times on n processors as shown in Table 1.
SP(n) = parallel speedup i.e. T(1)/T(n) as shown in Table 1.

- **The Sieve of Eratosthenes**

This algorithm has the property of stream parallelism and works as follows: The function Integers produces a stream of odd integers. The reference to Integers in function Sieve causes the production in Integers to occur but execution continues in Sieve concurrently. Each prime is found in Sieve by removing the first element of the input stream. From the rest of this stream, a new stream is generated by filtering out (through the function Filter) each multiple of the produced prime. This resulting new stream is used again in the prime-finding process in Sieve until a newly obtained prime is greater than the square root of the input parameter Limit (all primes have been found). The SISAL code for the implementation is shown in Figure 3.

To evaluate the Sieve prime finder with the parameter 20000, we found that a pipeline with 35 segments will form during execution. The first segment is the function Integer, 33 Filter segments follow (since there are 33 odd primes less than square root of 20000), and the final segment is Sieve itself. Table 1 shows the performance for the M31 and the Cray X-MP/416, respectively. The results are reasonably good, with the speedup saturating near 10 processors for the M31. The degradation of performance as more processors added is partly caused by the stream run-time software spending a large percentage of time on testing the end-of-stream condition in the pipeline. The performance of this algorithm on Cray was discussed previously [12] and is now shown here for comparisons.

```
type StrmInt = stream[Integer];
 function Integer(Limit: integer returns StrmInt)
% Produce a stream of odd integers i.e. 3, 5, 7, 9..Limit
   for  initial
        I:=3
        while I<= Limit repeat
          I:= old I + 2
        returns stream of I
   end for
end function

 function Filter(S: StrmInt; M: integer returns StrmInt)
% Produce a stream of values obtained from the argument stream S excepting those multiple of M
   for I in S
        returns stream of I unless mod(I,M) = 0
   end for
end function

 Function Sieve(Limit: integer returns StrmInt)
% Generate a stream of primes by inserting a filter on the stream against each prime produced
   Let
         Maxt := Integer(Sqrt(real(Limit)))
   in
        for initial
          S:=Integers(Limit);
          T:=2
         until stream_empty(S) repeat
          T:=stream_first(old  S);
          S:= if T != Maxt then
                 Filter(stream_rest(old  S),  T)
              else stream_rest (old S)
              end if
          returns stream of T
        end for
   end let
end function
```

Figure 3: SISAL Implementation: the Sieve of Eratothenes

5. Conclusions

In this paper, we have presented the internal mechanisms of task management to support parallel execution of SISAL. We have also discussed the parallel loop slicing and stream pipeline parallelism. The run-time support routines have been designed in a user-level library and were made transparant to different architectures with minimum architectural dependencies. We have shown the implementation and performance of this run-time system on two radically different architectures; i.e., one with a large number of small processors (M31) and another with only a few powerful processors (Cray X-MP). The performance results show that the implementation is promising given the nature and newness of the compiler and run-time system. However, the well-known inefficiencies of

functional languages were clear; in particular, memory consumption and unnecessary copying. In order to better evaluate the effectiveness of functional languages on multiprocessor systems, new, interactive, visual control mechanisms need to be developed to dynamically show the running behavior so that parallel computing can be evaluated more effectively. This deserves further research.

6. Acknowledgement

The authors acknowledge the discussions of this paper with Professor Meera Blattner, The University of California at Davis, and the assistance with experiments on the M31 from John Sopka of DEC.

References

[1] W. B. Ackermann and J. B. Dennis, "VAL: A Value-Oriented Algorithmic Language," Tech. Report LCS/TR-218, Massachusetts Inst. of Technology, Cambridge, Mass., June 1979.

[2] K. P. Arvind and K. P. Gostelow. "The U-Interpreter". *Computer,* Vol. 15, No. 2, pp 42-49, February 1982.

[3] K. P. Arvind, K. P. Gostelow and W. Plouffe. "The (Preliminary Id Report". University of California Ivine, Department of Information and Computer Science, Technical Report TR-114a, Irvine, California, California, May 1978.

[4] David C. Cann, Ching-Cheng Lee, R. R. Oldehoeft, and S. K. Skedzielewski. SISAL *Multiprocessing Supprot.* Technical Report UCID-21115, Lawrence Livermore National Laboratory, Livermore, CA, 1987.

[5] K. Crispin and R. Strout II. *NSYSLIB Library reference Manual*, LCSD 912, Draft2. Lawrence Livermore National Laboratory, Livermore, CA, January 1988.

[6] W. P. Crowley, C. P. Henderson, and T. E. Rudy. *The Simple code.* Technical Report, UCID 17715, Lawrence Livermore National Laboratory, Livermore, CA, February 1978.

[7] A. Davis. "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine." *Proceedings of the 5th Annual Symposium on Computer Architecture, Computer Architecture News,* Vol. 6, No. 7, pp. 210-215, April 1978.

[8] A. Davis. "DDNs - A Low Level Programming Schema for Fully Distributed Systems." *Proceedings on the Workshop in Data Driven Languages and Machines,* J. C. Syre (Ed.), Toulouse, France, section XVI, February 12-13, 1979.

[9] P. Hudak and L. Smith, "Parafunctional Programming", A Paradigm for Programming Multiprocessor Systems, ", *Conf. Record Symp. Princ. Programming Languages*, ACM, New York, 1986, pp243-254.

[10] R. M. Keller, B. Jayaraman, D. Rose, and G. Lindstrom. "FGL (Function Graphical Language) Programmer's Guide", University of Utah, Department of Computer Science, AMPS Technical Memorandum No. 1, Salt Lake City, Utah, 1980

[11] R. M. Keller and F. C. H. Lin, "Simulated Performance of a Reduction-based Multiprocessor," *Computer*, Vol. 17, No. 7, July 1984, pp 70-82.

[12] Ching-Cheng Lee, S.K. Skedzielewski, John Feo. *On* the implementation of Applicative Languages on Shared-Memory, MIMD Multiprocessors. *Parallel Programming: Environments, Applications, Languages, and Systems Conference*, New Haven, CT. July, 1988.

[13] J. McGraw, S. K. Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirham, Bill Noyce and Robert Thomas. SISAL: *Streams and Iteration in a Single Assignment Language, Reference manual Version 1.2.* M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, march 1985.

[14] R. R. Oldehoeft and D. C. Cann. Applicative parallelism on a shared memory multiprocessor. *IEEE software*, 5(1): 62-70, January 1988.

[15] M. H. Reilly and J. R. Sopka. M31 : a large-scale multiprocessor vax for parallel processing research. *In Proceedings of the Spring COMPON*, pages 200-206, March 1988.

	(a)		(b)	
n	T(n)	SP(n)	T(n)	SP(n)
1	181.72	1.00	3.77	1.00
2	102.36	1.77	2.14	1.76
4	53.31	3.41	1.55	2.43
8	28.94	6.28	-	-
10	24.50	7.42	-	-
12	21.98	8.27	-	-
16	22.11	8.22	-	-
20	20.83	8.72	-	-

Table 1 : Parallel Speedup of the Sieve of Eratothenes
(a) (symmetrical) Vax Research Multiprocessor M31 (upto 20 processors)
(b) Cray X-MP/416 (upto 4 processors)