A Study of LISP on a Multiprocessor Preliminary Version

Peter Nuth* Robert Halstead, Jr.

October 12, 1988

Abstract

Parallel symbolic computation has attracted considerable interest in recent years. Research groups building multiprocessors for such applications have been frustrated by the lack of data on how symbolic programs run on a parallel machine. This report describes the behavior of Multilisp programs running on a shared memory multiprocessor. Data was collected for a set of application programs on the frequency of different instructions, the type of objects accessed, and where the objects were located in the memory of the multiprocessor. The locality of data references for different multiprocessor organizations was measured. Finally, the effect of different task scheduling strategies on the locality of accesses was studied. This data is summarized here, and compared to other studies of LISP performance on uniprocessors.

1 Introduction

Several research groups are now trying to build parallel architectures for symbolic computing [23, 22, 9, 1]. Traditionally, computer architects simulate how existing languages and algorithms would run on proposed new designs. But there is little known about how symbolic programs might run on parallel processors. This paper addresses this lack of data by analyzing the behavior of symbolic programs on an experimental multiprocessor.

Several published reports profile an implementation of the *LISP* language on a particular processor [21, 20, 24, 8, 6]. However, the behavior of LISP on

*Author's address: MIT Lab for Computer Science, 545 Technology Sq., Room NE43-415, Cambridge, MA 02139. Tel: 617-253-6048. Arpanet: nuth@zermatt.lcs.mit.edu

a multiprocessor has not yet been studied. The work summarized here profiles the execution of real parallel LISP programs on an existing shared memory multiprocessor. It was not simply a simulation of code running on a multiprocessor. Nor was it a prediction of program performance based on the expected frequency of instructions.

In many cases, execution on a multiprocessor is both more convenient and more convincing than simulations. With sequential simulations, it is difficult to accurately predict the effect of contention for resources on the speed of global communication. It is equally difficult to see the effect of different task scheduling algorithms. Our own experience with LISP on a multiprocessor has shown that intuition is frequently wrong in such cases [13].

Some data described here could have been generated on a uniprocessor. For instance, we show profiles of instructions executed and types of data objects accessed by several LISP programs. Other results only apply to parallel execution of the code. An example is the degree of parallelism in a program and the amount of interaction between tasks. Another result discussed here is the *distribution* of data objects in memory, the *locality of reference* to those objects, and how both are affected by task scheduling decisions.

2 Experimental Method

The language used in these experiments was *Multilisp* [12], a lexically scoped dialect of LISP, similar to *Scheme* [7]. Multilisp programs were compiled into pseudo-machine instructions [15], which were then interpreted by the *Nusim* [18] simulator which counts instructions, data accesses, and other low-level behavior during the execution of a benchmark.

Multilisp allows the programmer to spawn parallel tasks by using the *future* construct [12]. The expression:

```
(let ((x (future (fn arg))))
  ...
```

spawns a parallel task to evaluate `fn` applied to `arg`. The variable `x` is initially *undetermined*. When the spawned task finishes evaluating the expression

(fn arg), x is bound to the result. The tasks created by the *future* statement can be executed by any processor in the system. Idle processors compete for tasks to run.

The experiments were run on *Concert* [2, 14], a shared memory multiprocessor system. Concert consists of up to 32 Motorola MC68000 processors,[1] each with 500 kilobytes of local memory. The processors share an additional 8 megabytes of global heap memory. A copy of *Nusim* ran in the local memory of each processor. The global heap memory was dynamically partitioned among all the processors in the system.

The five Multilisp application programs studied in this research effort were chosen to be representative of of LISP applications. The programs were:

Compiler The compiler for Multilisp, which is itself written in Multilisp [11].

Consim A parallel digital logic simulator [4].

Fboyer A parallel version of the Boyer-Moore theorem prover [10].

Multilog A query language interpreter using parallel unification [19].

Pqsort A parallel version of the *Quicksort* program.

3 Results

3.1 Instruction Profiles

The pseudo-machine language interpreted by *Nusim* contains approximately 115 instructions. The machine language can be grouped into several general classes of instructions as follows:

- Arithmetic and logical operations, and stack manipulation.
- Load and Store instructions.
- Conditional and unconditional branch instructions.
- Call and return instructions.

[1] The benchmark programs were only run on 27 of the 32 processors in the system.

- Instructions to create and manipulate *futures*.

Figure 1 shows the distribution of instructions for the five benchmarks tested in this study.

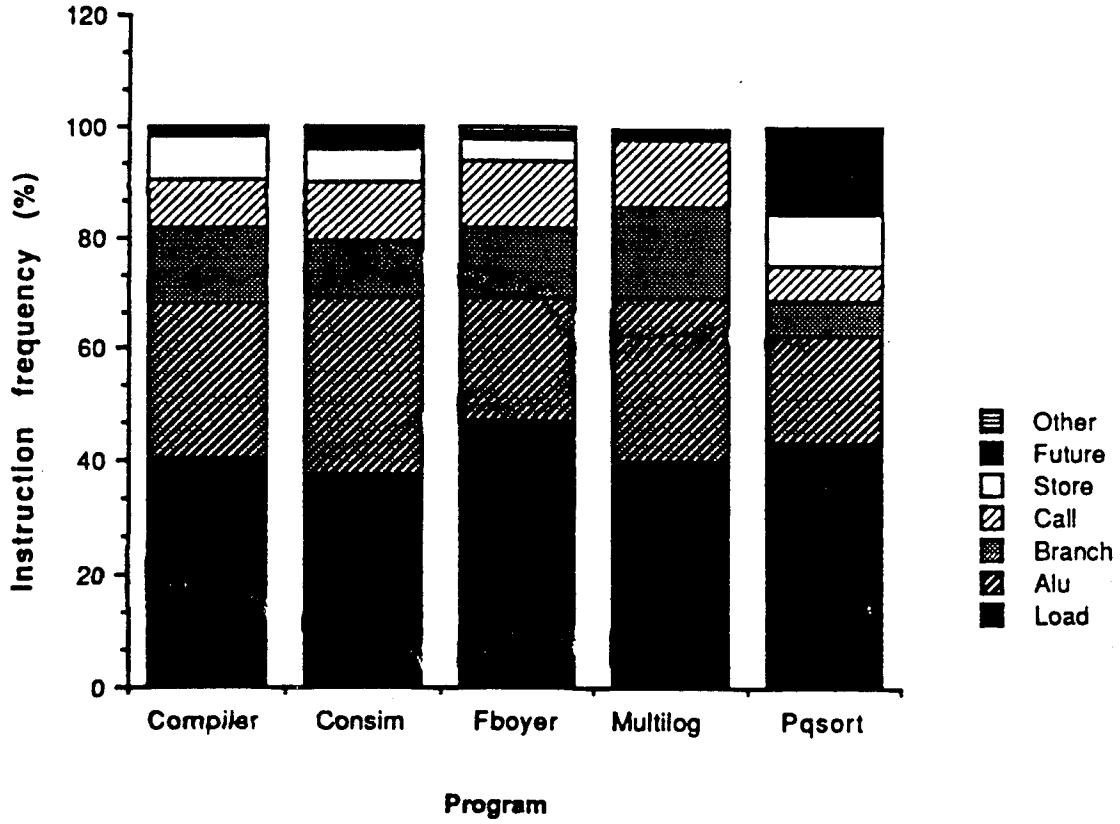

Figure 1: Distribution of instructions. (% per instruction class.)

The distribution of instructions summarized in Figure 1 differs from that reported in other recent studies of LISP, both on RISC processors [21, 23] and on conventional hardware [24, 8]. The major differences are the high proportion of load instructions, and the low proportion of ALU instructions for Multilisp programs. Also *Multilisp programs* use *futures* for explicit task synchronization. The following sections examine each of these results in turn.

3.2 Load Instructions

Figure 1 shows that an average of 42% of instructions load data from the heap onto the stack. Steenkiste [21] reported that only 28% of LISP instructions running on a MIPS-X processor were loads. For LISP running on the SPUR processor [23], only 17% of instructions were loads. The difference shows how difficult it is to apply the results of one low level LISP study to another implementation.

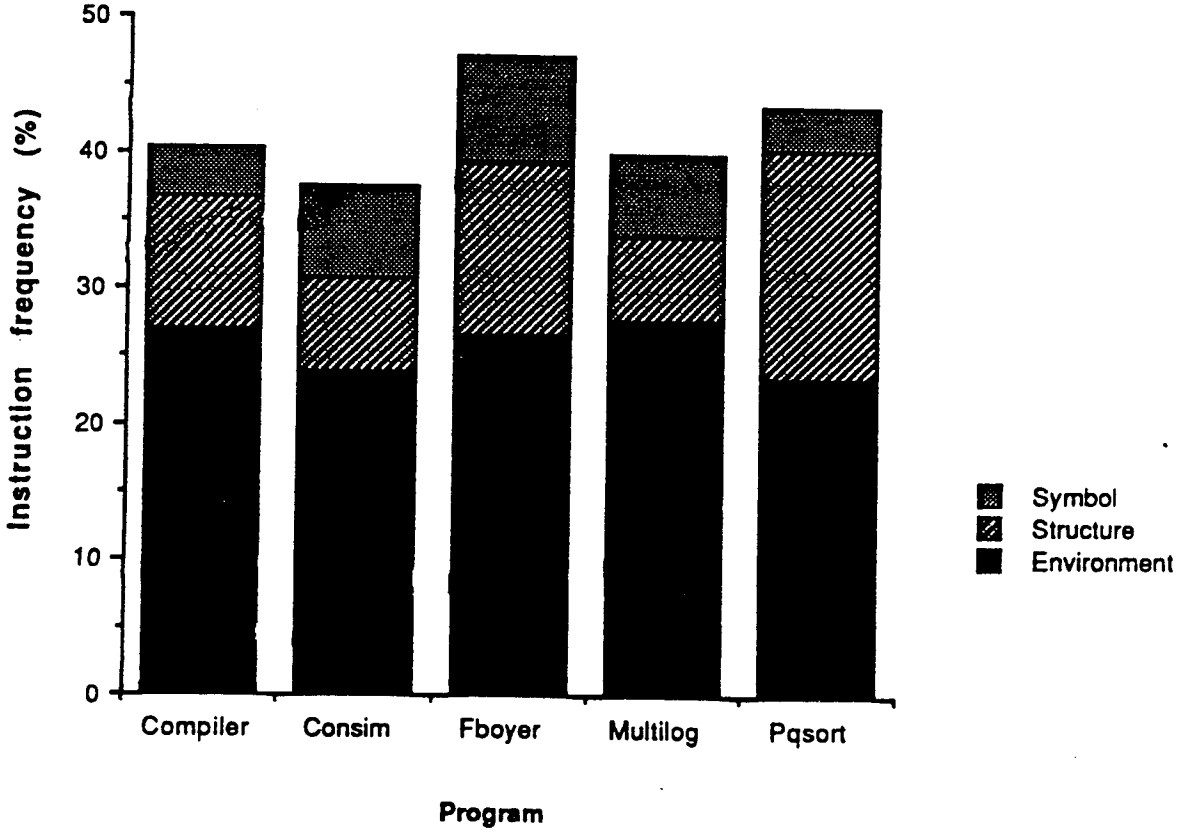

Figure 2: Load instructions by type: % of all instructions that read data from the environment, LISP structures or global symbols.

Figure 2 shows the classes of load instructions in Nusim. *Environment* fetches copy data out of the lexically scoped environment of the current LISP procedure. *Structure* loads fetch an element of a LISP data structure, such as

the *car* or *cdr* of a cons-cell. The other load instructions fetch the value of a global *symbol*. Note that these load frequencies are shown as a percentage of all instructions executed, meaning that one quarter of all instructions in a Multilisp program explicitly fetch a value out of the current environment.

There are two factors which account for the high number of fetches from the environment in Multilisp. First, when a parallel task is spawned using the *future* construct, that task inherits the environment of its parent. Since a spawned task may migrate to another processor in the system, environments are always allocated in the heap. A procedure's arguments and local variables are allocated in this environment, never in processor registers or on the stack. This means that a fetch from the heap is used to get the value of a local variable in Multilisp, whereas in other LISP implementations it may only require a register move [17, 5]. Second, the lack of common sub-expression elimination or other optimizations in the Multilisp compiler tends to increase the frequency of data fetches.

3.3 ALU Instructions

Figure 3 shows a similar breakdown of *ALU* instructions. These instructions include arithmetic operations, logical operations and instructions to compare two values. These operations pop their operands off the evaluation stack and push the result. Instructions that *push*, *pop*, or *copy* a datum on the stack are included here as well. Note that all ALU instructions account for only about 26% of all instructions in our application programs. The proportion for MIPS-X and SPUR was 36% and 44% respectively.

The Nusim implementation of Multilisp executes few ALU instructions relative to other LISP implementations. There are several reasons for this. Most Nusim instructions perform high level functions, equivalent to LISP primitives. Nusim does not use any explicit shift and mask instructions to extract the type tag of operands. Instead, all instructions check the types of their operands implicitly. A typical pseudo-code instruction, such as *add*, can handle fixnums, bignums, floating-point numbers, or trap to an error routine depending on the types of its operands.

Nusim also does not use *add* and *subtract* instructions to generate effective

3 RESULTS

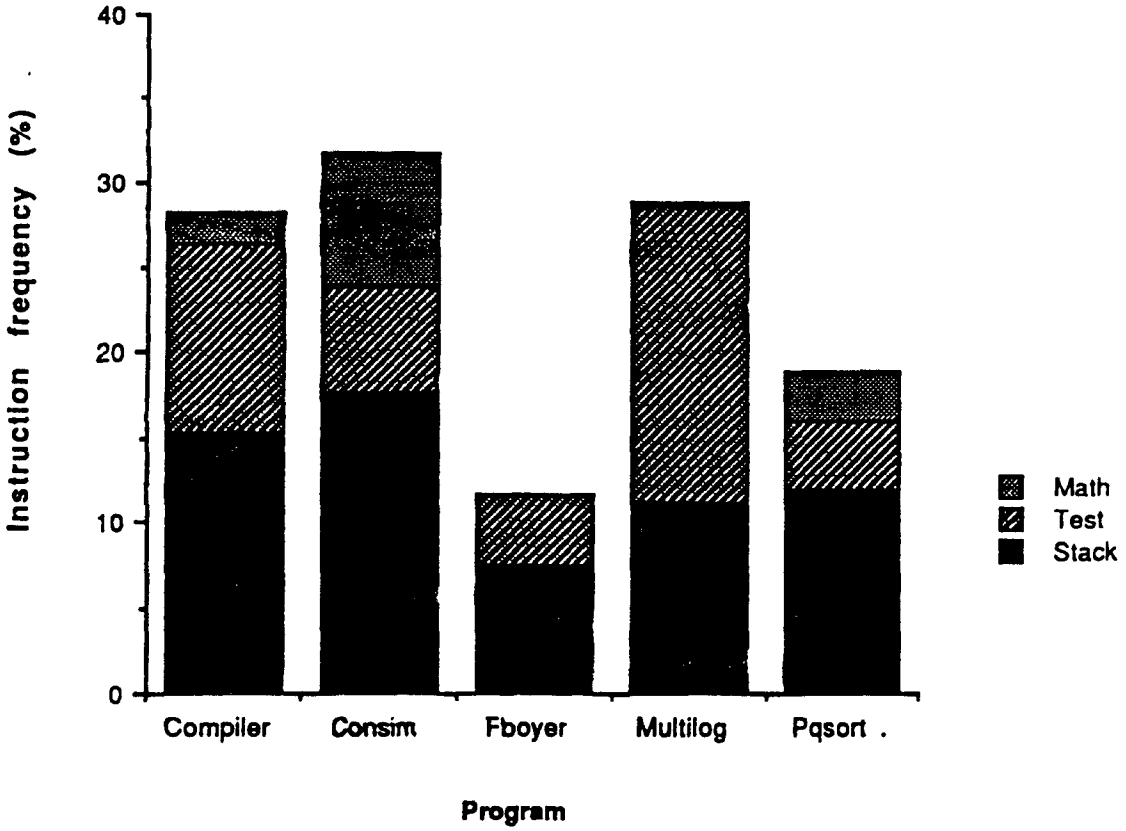

Figure 3: ALU instruction frequencies: % of all instructions that manipulate the stack, test a value, or do arithmetic.

addresses. Instructions which fetch an element of a data structure first calculate the address of that element. *Cdr* is a single instruction, not an *add* followed by a *load*. The pseudo-code also contains relatively complex *call* and *return* instructions, to build a call frame and an environment for the called procedure. In other LISP implementations, several *add* and *subtract* instructions are needed to adjust the stack pointer on procedure call and return [21].

3.4 Access Types

The pseudo-machine emulated by Nusim has explicit load and store instructions as discussed in Section 3.2. However, other instructions may also implicitly access data from memory. [2] We can classify explicit and implicit accesses by the type of data that they fetch or store. A useful characterization emphasizes that some data must be shared among all processors in the system, while other data could be allocated in memory local to a processor. Here are five such classes of data:

- **Constant** data includes Lisp instructions and constants that are kept with the code stream. This immutable data could be copied to the local memory of each processor, although *Nusim* does not currently allocate it there.
- The **stack** holds procedure linkage information and temporary values that are local to a single task. Nusim maintains a *stack cache* for each processor to hold the top of the stack. Only loads and stores to fill or flush the stack cache are counted here.
- **Environment** frames may be shared between tasks that have common lexical parents, and must be allocated in the heap.
- **Global** accesses count fetches of Lisp global symbols and any structured data allocated in the heap.
- Accesses to **future** objects are counted separately from other structured data, because of their role in synchronizing parallel tasks.

[2] An example is the *call* instruction, which builds a new environment frame in the heap.

3 RESULTS

Figure 4 shows the percentage of explicit and implicit fetches from each class of data. The distribution of access types varies for the benchmarks tested. The measurements show that keeping constant data in the local memory of a processor could reduce global memory accesses by one third. Currently, all environment frames are also allocated in the heap. If the frames were only copied out into the heap when necessary to share data between tasks, global memory traffic could be reduced by as much as 20%.

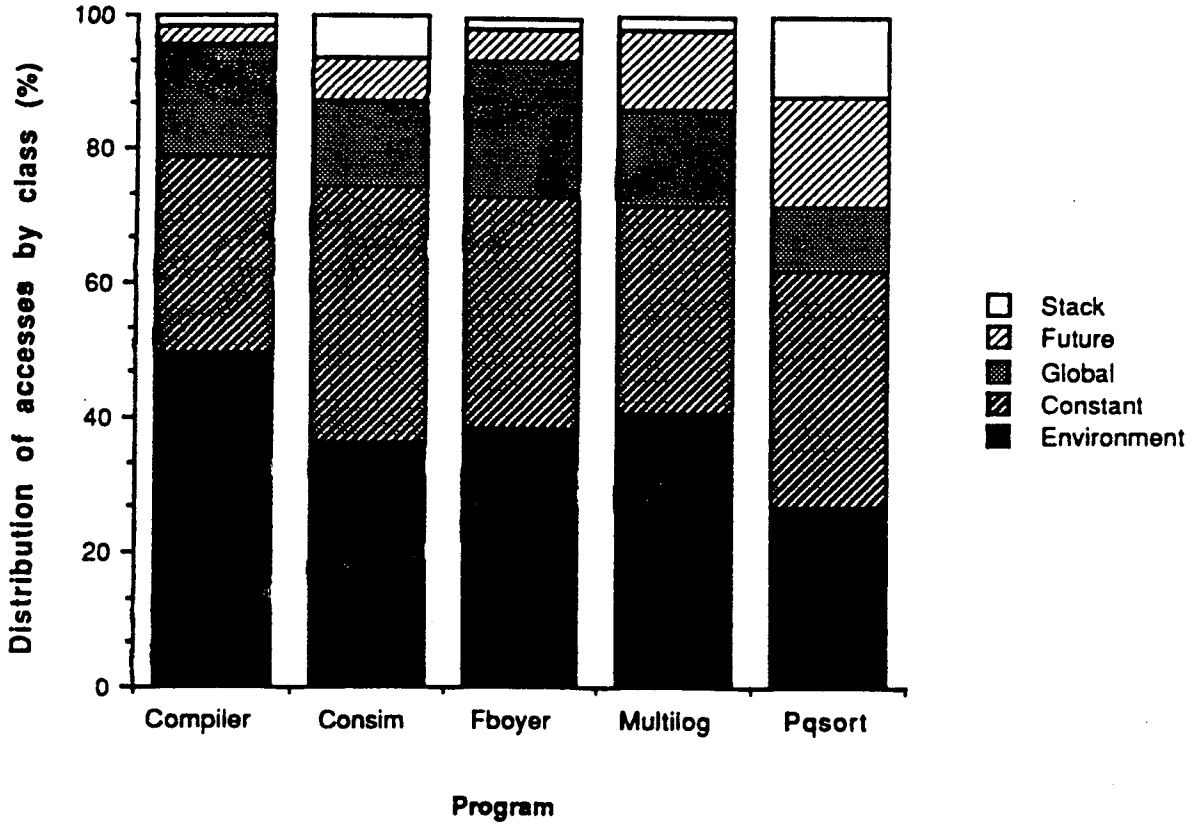

Figure 4: Class of data fetched by programs.

3.5 Parallelism

The *future* construct is a mechanism for spawning parallel tasks, and is intended to be as common as procedure calls. Proposed processor architectures for Mul-

tilisp [16] have devoted considerable resources to handling futures efficiently. This study shows how often they occur in practice.

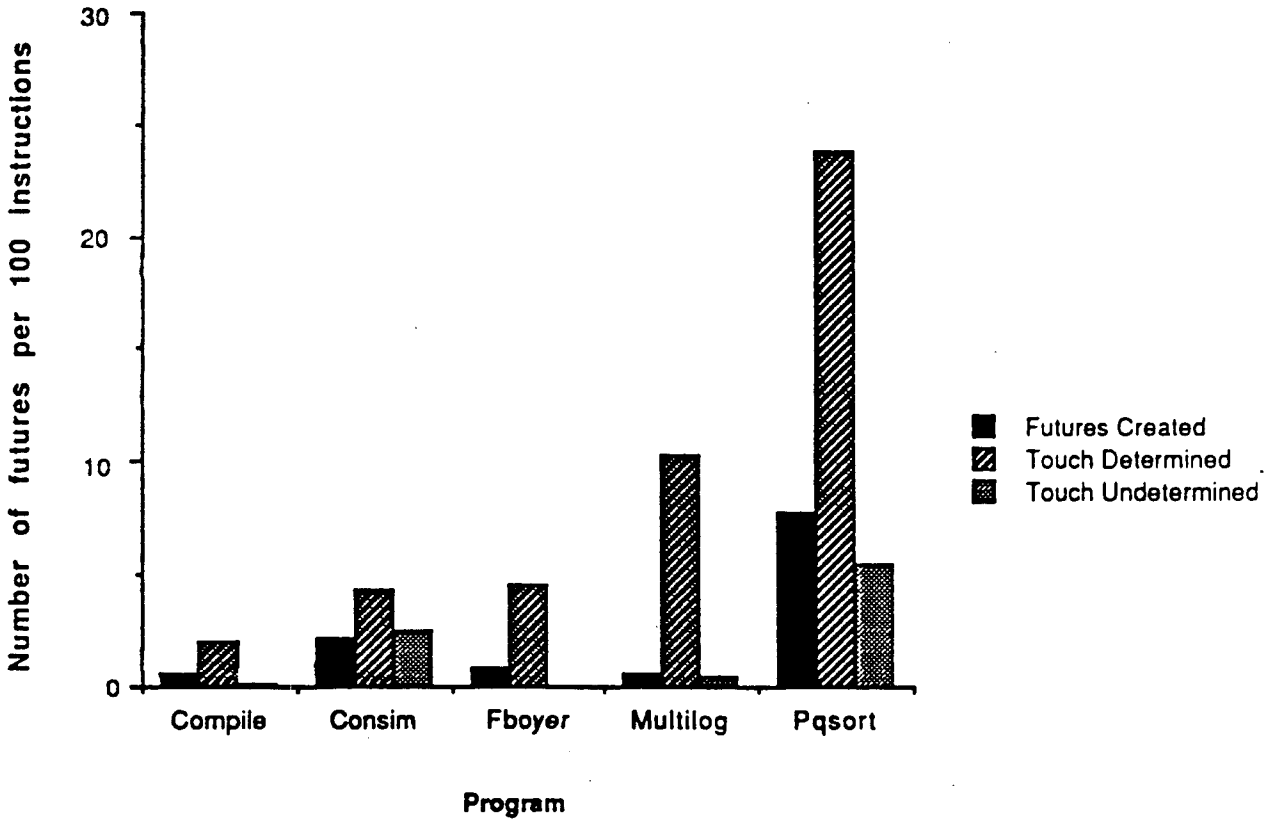

Figure 5: Future object creation and accesses as a percentage of instructions.

The benchmarks shown differ widely in their use of futures. The Compiler, Multilog, and Boyer-Moore have approximately 100 instructions per future created, while Pqsort tasks are only 13 instructions long. Each future object created is typically read several times by tasks that depend on its value. Most of the accesses touch *determined* futures, after their value has been computed. Occasionally, a task will touch an *undetermined* future, forcing the task to queue up and wait for the value to be determined. Only Consim and Pqsort touch a significant number of undetermined futures.

This data shows the danger of making assumptions about the frequency of

operations from the behavior of a particular implementation. Aside from the small Pqsort program, none of the benchmarks studied created many future objects. They are certainly not as frequent as function calls or branches. Yet this may reflect the cost of future creation and access in Nusim. Programmers could afford to spawn more parallelism if it was less expensive in time and memory. See [18] for a more thorough discussion of parallelism and use of futures for these programs.

3.6 Locality of Reference

While *Nusim* runs on a real multiprocessor, it can simulate other multiprocessor topologies. We can describe such a topology as a set of connected nodes, each of which contains a processor and some memory that is shared with the rest of the system. *Nusim* can pretend that physical processors and memory are located at the nodes of the simulated topology. It also simulates partitioning the global heap among all node memories. The 'distance' of a memory reference is then the simulated number of hops between two nodes. By tracking the distance of all accesses made during the execution of a program, we can compute the locality of reference for that program on a particular topology.

For these experiments, we simulated three different topologies using *Nusim*. See Figure 6. The *Ring* topology assumed that the nodes were distributed along the circumference of a circle, each connected to its nearest neighbor. The *Grid* assumes that nodes are connected in a two-dimensional rectangular grid, so that each node has four nearest neighbors. (The grid wraps around at the edges). Finally, a *Segmented* topology assumes that nodes are split into a number of groups, each with a private bus. The local buses are then tied together by a global bus. All three topologies were simulated with 27 nodes, since that was the number of real *Concert* processors available at the time.

Figure 7 shows the locality of data references made by our benchmarks on the three simulated topologies. The distance of each access is averaged over all accesses to produce the *mean distance of access* for each benchmark. Programs with greater locality have a lower mean distance of access on a particular topology. The data labeled 'Random' indicates what the locality would have been if the accesses had been uniformly distributed among the nodes of the system.

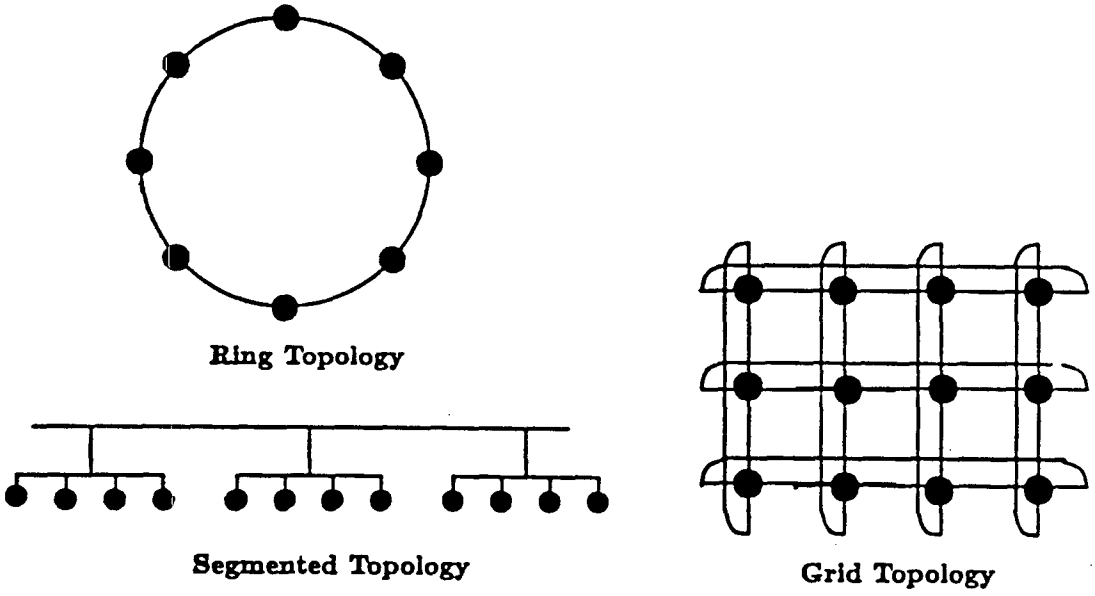

Figure 6: Multiprocessor topologies simulated by Nusim.

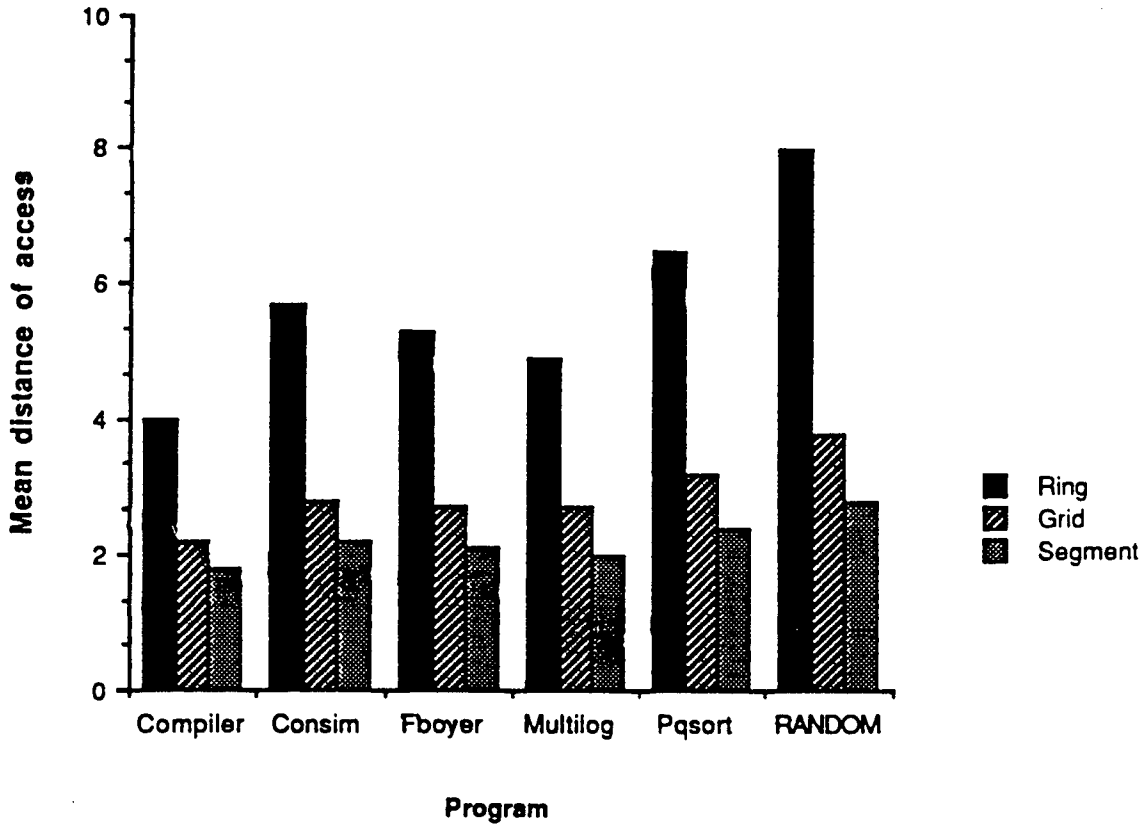

Figure 7: Basic locality of reference: Mean distance of accesses for different simulated topologies. Column "random" shows the locality for purely random accesses.

The data shows that benchmarks have a greater locality of reference than purely random accesses. Accesses to memory that is close to a processor occur more frequently than distant references. In the Nusim implementation, each processor allocates data in its partition of the global heap. Typically, 30% to 40% of the references made by processors are to the local partition. All other references are uniformly distributed through the rest of memory. This data allocation protocol therefore improves the locality of reference of programs in Nusim. The mean distance of access for the Compiler, for instance, is only half of the random access distance.

3.7 Effect of Task Scheduling on Locality

In an attempt to further increase the locality of reference of programs, we varied the way that tasks are scheduled in *Nusim*. When a task is spawned by a processor, it is added to a queue of tasks in that processor's node. Idle processors then search the task queues of other nodes for work to do.

An improved strategy was to make the searching process aware of the topology of the system. Thus, in searching for tasks to run, a processor might first check local nodes. The effect of this strategy is shown in Figure 8.

With an intelligent task search algorithm, programs made an average of 10% more accesses to local node memory than before. However, the locality of reference improved by 5% to 30% over the primitive task scheduler. All programs showed the most improvement on the ring topology, in which the diameter was greatest.

The locality of some data accesses improved more than others in response to the new task scheduling algorithm. Accesses to *future* objects, the main points of synchronization between parallel tasks, improved the most. The effect was greatest for the *Pqsort* program, which uses futures more extensively than any other. This indicates that good task scheduling strategies can reduce the amount of global communication required in a shared memory multiprocessor.

3.8 Other work

Most of the data in this report was taken from [18]. which discusses in more detail the types of accesses made by Multilisp programs and the locality of

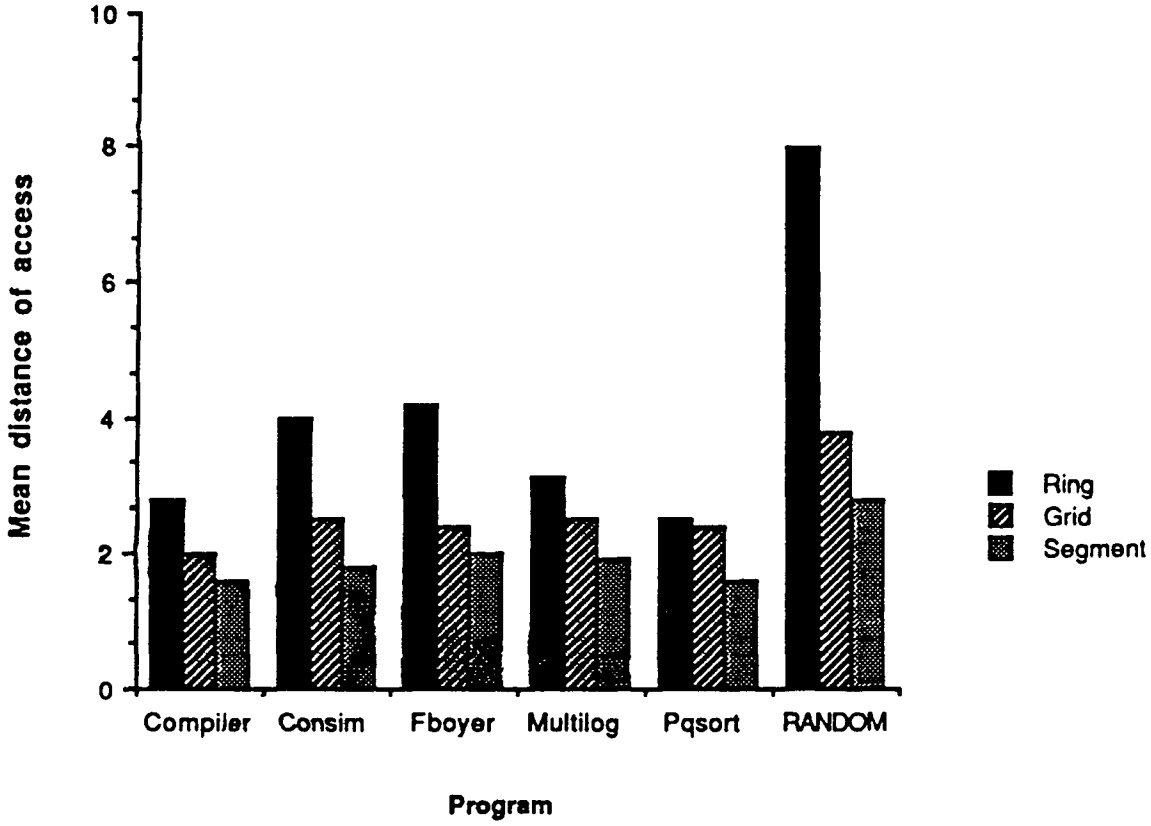

Figure 8: Locality of reference for different simulated topologies, using an improved task scheduling algorithm.

each type of reference. It also discusses the effect of limiting the parallelism of Multilisp programs, and the trade-offs between parallelism and locality.

A later study used *Nusim* to investigate the effect of caching in a symbolic multiprocessor [3]. The simulator was used to tag data words to show what data is shared between different processors in the system, and what data is private to a processor. This was used to predict the best case performance of very large data caches on each processor.

4 Implications for Architecture

The instruction profiles presented here differ from what has been reported for other studies of LISP on uniprocessors. The implementation of a language has a great effect on the mix of instructions that will be executed for typical LISP programs. This shows how difficult it is to make assumptions about the behavior of LISP programs when evaluating architectures.

Multilisp programs use *future* objects to spawn and synchronize between tasks less frequently than was expected. The relative cost of operations seems to have an effect on the style of parallel programs written for an architecture. Therefore it is difficult to evaluate hardware mechanisms to support parallelism in future LISP architectures while using code that was written to run on existing implementations.

The types of load instructions used by the benchmarks show that many explicit memory fetches could be eliminated in a different implementation of Multilisp. A more sophisticated compiler could initially allocate lexical environments on the stack or in machine registers [17]. Constant data could be copied into the local memory of each processor.

Multilisp programs had more locality of reference than purely random accesses. Two mechanisms were shown to increase the locality of reference. The first simply allocated all data created by a processor in the processor's local memory. The second used a more sophisticated task scheduling algorithm to grab tasks from nearby processors whenever possible. The implication is that simple techniques can reduce the communication bandwidth required by parallel symbolic programs.

References

[1] ANDERSON, J., COATES, W., DAVIS, A., HON, R., ROBINSON, I., ROBISON, S., AND STEVENS, K. The architecture of the FAIM-1. *IEEE Computer 20*, 1 (Jan. 1987), 55–65.

[2] ANDERSON, T. The design of a multiprocessor development system. Tech. Rep. TR-279, Laboratory for Computer Science, M.I.T., Cambridge, Mass., September 1982.

[3] BERGSTEIN, S. H. Best-case caching in a symbolic multiprocessor. Bachelor's thesis, February 1988.

[4] BRADLEY, E. Logic simulation on a multiprocessor. Tech. Rep. TR-380, M.I.T. Laboratory for Computer Science, Cambridge, Mass., November 1986.

[5] BROOKS, R. A., POSNER, D. B., MCDONALD, J. L., WHITE, J. L., BENSON, E., AND GABRIEL, R. P. Design of an optimizing, dynamically retargetable compiler for Common Lisp. In *Proceedings of 1986 ACM LISP and Functional Programming Conference* (Boston, Aug. 1986), ACM, pp. 67–85.

[6] CLARK, D. Measurement of dynamic list structure use in Lisp. *IEEE Trans. Softw. Eng. 5*, 1 (Jan. 1979), 51–59.

[7] CLINGER, W., ET AL. The revised revised report on scheme, or an uncommon Lisp. Memo 848, M.I.T. Artificial Intelligence Laboratory, Cambridge, Massachusetts, August 1985.

[8] FODERARO, J., AND FATEMAN, R. Characterization of VAX Macsyma. In *Proceedings of 1981 ACM Symposium on Symbolic and Algebraic Computation* (Aug. 1981), ACM, pp. 14–19.

[9] GABRIEL, R., AND MCCARTHY, J. Queue-based multi-processing Lisp. *ACM Symposium on LISP and Functional Programming* (August 1984).

[10] GABRIEL, R. P. *Performance and Evaluation of Lisp Systems.* MIT Press Series in Computer Science. M.I.T. Press, Cambridge, MA, 1985.

[11] HALSTEAD, JR., R. H. Implementation of Multilisp: Lisp on a multiprocessor. In *ACM Symposium on Lisp and Functional Programming* (Austin, Texas, August 1984).

[12] HALSTEAD, JR., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 501–538.

REFERENCES

[13] HALSTEAD, JR., R. H. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Programming 16*, 6 (Dec. 1986).

[14] HALSTEAD, JR., R. H., ANDERSON, T., OSBORNE, R., AND STERLING, T. Concert: Design of a multiprocessor development system. In *13th Annual Symposium on Computer Architecture* (Tokyo, June 1986), pp. 40–48.

[15] HALSTEAD, JR., R. H., LOAIZA, J. R., AND MA, M. H. The Multilisp manual. PPG Group Working Paper, September 1986.

[16] JR., R. H. H., AND FUJITA, T. MASA: a multithreaded processor architecture for parallel symbolic computing. In *15th Annual Symposium on Computer Architecture* (May 1988), IEEE Computer Society, pp. 443–451.

[17] KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction* (June 1986), ACM SIGPLAN, pp. 219–233.

[18] NUTH, P. R. Communication patterns in a symbolic multiprocessor. Tech. Rep. MIT/LCS/TR-395, M.I.T. Lab for Computer Science, June 1987.

[19] SOLOMON, S. A query language on a parallel machine. MIT EECS Bachelor's Thesis, June 1985.

[20] STEENKISTE, P. LISP on a reduced-instruction-set processor: Characterization and optimization. Tech. Rep. CSL-TR-87-324, Stanford University, Mar. 1987.

[21] STEENKISTE, P., AND HENNESSY, J. LISP on a reduced-instruction-set-processor. In *Proceedings of 1986 ACM LISP and Functional Programming Conference* (Boston, Aug. 1986), ACM, pp. 192–201.

[22] SUGIMOTO, S., AGUSA, K., TABATA, K., AND OHNO, Y. A multimicroprocessor system for concurrent LISP. In *Proceedings of International Conference on Parallel Processing* (June 1983).

[23] TAYLOR, G. S., HILFINGER, P. N., LARUS, J. R., PATTERSON, D. A., AND ZORN, B. G. Evaluation of the SPUR lisp architecture. In *Thirteenth International Symposium On Computer Architecture* (June 1986).

[24] URMI, J. A machine independent Lisp compiler and its implications for ideal hardware. Linkoping Studies in Science and Technology Dissertations 22, Linkoping University, Linkoping, Sweden, 1978.