1-1990

Benchmarking Parallel Machines via a Software Architecture

Srikanth Kambhatla

Jon Inouye

Jonathan Walpole
Portland State University

BENCHMARKING PARALLEL MACHINES VIA A SOFTWARE ARCHITECTURE

Srikanth Kambhatla, Jon Inouye, Jonathan Walpole
Department of Computer Science and Engineering,
Oregon Graduate Institute of Science and Technology,

srikanth@cse.ogi.edu, jinouye@cse.ogi.edu, walpole@cse.ogi.edu

[Keywords: Performance Evaluation, Parallel Processing, Software Architecture, Synthetic Benchmark, Linda]

ABSTRACT

The diversity in parallel architectures and the programming styles induced thereof, make benchmarking of parallel machines a hard problem. Current benchmarking techniques are limited in applicability due to the issues of portability. We present a different approach to benchmarking parallel machines, which would make the resulting benchmarks portable, easy to use and widely applicable. The approach involves the use of a software architecture to mask the diversities of the underlying architectures. The benchmarks are based on the Linda Tuple Space, and is called BeLinda.

In this report, we motivate the use of a software architecture, give a specification of the benchmark suite, discuss its implementation on three radically different architectures, and present the results of the benchmark.

1. Introduction

Performance evaluation of parallel architectures is a hard problem. Firstly, one is faced with the aspect of diversity. We currently have parallel machines whose architectures are totally different from each other. Parallel machines can either execute in single instruction stream, multiple data stream (SIMD) mode like the Thinking Machines CM-2 [1], or multiple instruction stream, multiple data stream (MIMD) mode like the Sequent Symmetry [2] and Intel iPSC/2 [3, 4, 5]. MIMD machines can be further subdivided based on the characteristics of the address spaces of the constituent nodes. If the address spaces are disjoint, we get the distributed memory distributed address space machines like the iPSC2, and if the address spaces partition the common global address space, we get the distributed memory shared address space (DMSA) machines like the BBN Butterfly. DMSA machines are characterized by non uniform memory access times depending on the address being accessed.

This diversity in architecture induces a diversity in the programming styles. On a shared memory machine, the programmer has to identify shared data and explicitly protect them from concurrent accesses through some form of locking. The shared data structures are available to all the processes on the machine. On the other hand, in a distributed memory machine, the programmer is forced to adopt a network style of parallelism. The processes have to be split up keeping in mind the higher costs of communication and synchronization, with the failure in doing so resulting in ineffecient implementations. Cooperation is achieved through message passing as against shared data in the previous case.

Next we are faced with the crucial issue of portability. What we need is a language which spans the barriers of architectural diversity to provide a uniform model of parallel computation to the user. Under such circumstances it becomes easy to think of the problem purely in terms of the parallelism built into the application rather than addressing oneself to the vagaries of a particular architecture. In the hope of alleviating the problem of portability and uniform programming style, there has been a significant push to develop languages which offer a sufficiently high level of abstraction and yet are easily and efficiently implemented on a wide variety of machines. Languages such as CSP [6], Concurrent Prolog [7], and Strand [8] were developed for this purpose.

Because of the problems mentioned above, pervious work reported in the area of benchmarking of parallel machines has tended to be in either of two directions. The first one is comparing different versions of the same machine, or different implementations in the same architectural category. Bomans [9], Kolawa [10], and Grunwald [11] do it for evaluating different versions of iPSC hypercube, MARK II and the NCube. The other direction specifies the performance with respect to some application program. This is the approach followed by Martin [12], Fraboul [13], and Gustafson [14]. Both these methods have their own drawbacks. The first one limits applicability to the machines of that flavor only, while the second approach limits the usefulness of the results to applications in the same domain as the program being used as the benchmark. Furthur, a user wishing to get an idea about the cost he would incur in running a (any) program would not find the timings on another application program very useful. Other studies on the performance of parallel machines have been analytical, such as the one by Flatt & Kennedy [15], The problem here is that this approach does not provide any information on the performance of any specific implementation.

In this paper we present a different approach to benchmarking parallel machines. This approach involves the use of a software architecture. The stress is on portability, ease of implementation and usefulness of results. The benchmark, called BeLinda, is based on the Linda Tuple Space [16, 17, 18] which we believe defines an appropriate level of abstraction for comparing different parallel computing platforms. An advantage of using a Linda based benchmark is that it is easily portable over a wide variety of hardware architectures. However, the results of a benchmark at this level of abstraction are influenced not only by the performance of the underlying architecture, but also by the implementation of Linda on that architecture. Consequently, a good figure indicates that the combination of the implementation and the underlying architecture is good. However, a bad figure does not necessarily distinguish between a bad architecture, a bad Linda implementation, or both.

One usually talks about the instruction set architecture in uniprocessors. The idea there is the distinction between the architecture which is visible to the programmer, as against the implementation of the sequential machine and details like the presence of cache, pipelines etc., which improve performance but are not visible to the programmer. In parallel machines, we have components for evaluation other than the processors at various nodes viz. the communication and synchronization issues. This fact not withstanding, Linda provides us with a software architecture not unlike the instruction set architecture of sequential machines, because it masks out the details of how the parallel machine has been implemented, and on what model of computation it is based. This analogue is the basis on which we build the thesis that the way to go as far as generation of a benchmark suitable for parallel machines is via a software architecture.

Linda is a programming model based around a shared *tuple space* and several tuple space operations [19, 20]. These tuple space operations can be embedded into any standard language

such as C or FORTRAN, creating a new parallel language. Instead of having read and write operations over shared memory values, or using message passing, Linda provides the abstraction of the tuple space for both communication and synchronization.

A tuple consists of an ordered set of values. These values can be integers, floating point numbers, arrays, blocks, or even programs. The tuple space operations are : **out(), in(), inp(), rd(), rdp(),** and **eval()**. **in()** removes a selected tuple from the tuple space. If there is no tuple satisfying the **in()**, the process will block until such a tuple appears. **read()** acts like **in()** in that the tuples values are read, but the tuple remains in the tuple space. **out()** adds a tuple to the tuple space. **eval()** acts like **out()** except the tuples are active. An active tuple can be thought of as a process. In that sense, **eval()** acts somewhat like the UNIX fork() system call. However, it can eventually become passive, i.e. leave a result in tuple space. **inp()** and **rdp()** are asyncronous versions of **in()** and **rd()** respectively, in that they do not block if the matching tuple is not found. They return the status of the match. These operations are more fundamental than their synchronous counterparts, because the later can be built using **inp()** and **rdp()** but not vice versa. However, these are not incorporated into the present study.

The rest of the paper is organized as follows. Section 2 describes some attributes of a good benchmark, and we try to relate our benchmark with that list. In section 3, we detail our choice of the performance measures, which are of interest in any parallel computation. Section 4 gives the specification of BeLinda, our benchmark set. Section 5 describes the data collected about Linda primitives and how these figures were used to arrive at the weights for the individual performance measures. Section 6 describes the compiler and architectural features of the machines under consideration. Section 7 contains the results obtained for the three machines and tries to relate them with the underlying machine architecture. We discuss the results and arrive at the final BeLinda figures in section 8. Section 9 concludes the paper.

2. Attributes of a good benchmark

Gustafson [14] points out that a benchmark should have the following characteristics:

- It should be representative of actual applications.
- It should not artificially exclude a particular architecture or configuration.
- It should reduce to a single number to permit one-dimensional ranking.
- It should report enough details to be reproducible by an independent investigator.
- It should permit simple verification of correctness of results.
- It should use simple algorithms to fit into one page.

We shall now try and relate our set of benchmarks with respect to these attributes. Continueing the parallel to the instruction set architecture evaluation, what is needed is the identification of work primitives, or constructs which typify parallel computation and which occur in most applications. BeLinda attempts to identify such a set. The results obtained from these would be of generic use and each user would be in a position to relate to these, to their application.

The second issue deals with the portability of the benchmarks across architectures. The fact that Linda is portable has been seen before. It is also apparent that as long as one writes benchmarks in a language which projects its own architecture the benchmarks are going to be portable. As Gustafson [14] points out this aspect has not been addressed in many previous studies.

Apparently, the requirement of reduction to a single number came out of convenience. Gustafson mentions the fact that the Livermore Kernels present an entire statistical base of data, but

the performance of these kernels is often stated as a single number. The benchmarks listed here also produce a statistical base of data. The approach adopted here has been to weigh the figures obtained from various programs based on the frequency of occurrence of the particular construct being evaluated. In order to obtain the weights for the programs, we collected data from various applications developed in Linda and instrumented the code to find out the dynamic and the static frequencies of the constructs. The other issues are concerned with the ease of implementation and verification of the benchmark and have been met as far as possible in BeLinda.

3. Identification of work primitives

The work primitives identified fall into the following categories:

- Basic Linda primitives
- Primitives for communication and synchronization as used in Linda
- Primitives for computation
- Primitives for overlap of computation and communication
- Other issues in parallel computation

We shall now examine each one of these in turn.

3.1. Basic Linda primitives

Linda has four primitive operations in(), out(), rd() and eval(). The efficiency of the Linda implementation depends to a large extent on the methodology for the implementation of theseprimitives. Hence, the time to perform each of these has to be measured.

Secondly, Linda gets a lot of its power from its content addressability through structured naming. Therefore, an important measure is the number of fields in the tuple, and the number of actuals in these fields. It is clear that the larger the number of fileds in the tuple, the larger will be the time to perform the match. Many of the Linda compilers available, optimize out the tuple references and reduce the time for match by performing what is known as the tuple analysis. However, it is a known fact that compilers play a crucial role in the performance of the benchmarks.

3.2. Primitives for communication and synchronization

In Linda, synchronization is achieved through the tuple space. Thus it is no longer a primitive operation of the native architecture, but we get it via the software architecture portrayed by Linda. Synchronization is achieved by means of a single field tuple exchange between the two processes. In order to ensure correctness, the single field in the tuple is an actual.

Communication is an operation which is not visible to the user. This is intentional and is part of the Linda semantics. The user looks on the machine as a huge tuple space in which parallel processes crawl over distributed data structures. The communication primitives for an architecture are defined in terms of the following

Latency: Latency is the time required to send a zero length message between two nodes in the architecture. In the case of shared memory systems, the cost involved is only in terms of copying between different buffers and is an operating systems overhead. Where as in a distributed memory machine we require physical transmission of data between two nodes over the communication network.

Contention: This refers to the fact that messages going in opposite directions might influence the communication times of each other. This measure is highly dependent on the number of redundant paths in the network and also on the presence of full duplex communication in the network.

Multicast: This is the operation required when one node talks to k other nodes. If k happens to be the number of nodes in the network, this operation is called the broadcast.

Reverse-Multicast: This is a k-1 operation when many nodes send messages to a single node. This might happen when a process is waiting on a barrier from all the other nodes.

Message-Size: It is a measure of the effect of increasing message length on the transmission time. In some shared memory systems message passing may be implemented via memory mapping in which case message length does not effect the transmission time. Increasing message delay increases the point at which it would be less costly to distribute the task among several processors.

3.3. Primitives for computation

Although Linda is now being used in a variety of application domains, including systems programming [21], the majority of the applications continue to be in the scientific computation domain. The frequently reported primitives in this domain are the add and multiply times for integers, floats and double precision floats. Also, we report the timings for a+b*c operation on the various machines.

3.4. The overlap of computation and communication

Communication between non-neighboring nodes is typically handled by separate communications controllers in the sender, receiver and the intermediate nodes. So, the node processors of the intermediate nodes should not be affected by the passing traffic. A good way to measure the overlap in computation and communication in parallel machines would be to measure the time for computation and the time for communication separately and then put the communication and the computation parts together and time it again. The amount of overlap will be obtained by subtracting the last measure from the sum of the measures in the first two cases.

However, this would be tough to do in the context of Linda. The method we adopt is then to try and measure the amount of interference caused because of interaction between the computation and communication. We measure the effect in two ways. The first being the effect of communication on the computation off other processors, and, the effect of computation off other processors, on the communication between the cooperating processes. Although, this is the way it should be, Linda provides no way of ensuring that a particular process runs on a particular processor. Thus to increase the chances that the processes doing the computation are on different processors, we eval() several processes doing the same computation.

3.5. Other issues in parallel computation

There are several issues in parallel computation which should be considered in systems design but not all are relevant in the benchmarking phase. Some of the measures in this category are fairness and starvation. However there are some issues which could not be placed in any of the above categories but are relevant to measure the implementation. They are

Scaling-Effect: The biggest advantage in distributed memory machines is scalability. In most shared memory machines the bus becomes a bottleneck as the system size is increased

(exceptions may be in the form of the BBN Butterfly). We need to measure the time to compute a constant workload using an increasing number of eval()s.

Blocked-in()s: This is a measure directly aimed at testing the implementation of Linda. The point is that when a tuple is missing on an in(), the process is supposed to suspend. But in some shared memory implementations, the processes spin. If the process spins, clearly there will be a degradation. If instead the process is kept in a blocked queue of processes, we still have some amount of degradation because of the overheads associated with managing the queues and checking periodically if the tuple has arrived. The goal is to check if this results in any degradation as far as the overall computation goes.

We have tried to list out the various parameters to be included in evaluating the Linda benchmark. The fact remains that although Linda masks out the native architecture from the users viewpoint, inefficient implementation of the underlying architecture, or, bad architectural features can cause degradation in the performance on the applications. We have strived to incorporate in the benchmarks, tests which will reflect these features. The following section will explain the program set constituting the benchmarks, the specification of each module and the method of implementation.

4. The BeLinda Specification

BeLinda consists of a set of eleven individual programs which evaluate the various characteristics discussed above. A brief description of each of the BeLinda programs is given below:

primitives.l evaluates the cost of doing the Linda out(), in(), rd(), and eval() operations by performing *n* primitive primitive operations of each type[1]. The time is then divided by 'n' to obtain the average time for each individual operation. In the case of eval(), a null eval which does no processing is performed.

actuals.l evaluates the cost of executing a primitive operation with a varying number of actuals in the tuple. This is achieved by varying the number of actuals and timing rd() and in() operations.

formals.l evaluates the cost of executing a primitive operation with a varying number of formals in the tuple. This is achieved by varying the number of formals in the tuple and timing rd() and in() operations.

spin.l checks whether processes spin during in() or rd() operations by timing the execution of a task using four workers on a single processor. The task is then rerun with one of the workers blocked on an in().

scale.l determines how the time required to execute a constant amount of work varies as the number of eval()s[2] is increased.

latency.l measures the latency involved in communication between processes by bouncing the same message (with one actual field) back and forth between two processes. After N iterations, the elapsed time is divided by 2N to obtain a figure for latency.

contention.l measures the degradation in performance due to contention for channels by simultaneously exchanging tuples in opposite directions between two processes. The number of such interacting pairs is varied. To ensure that messages are indeed transmitted

1 where 'n' is large enough to ensure that the overall time is discernible.

2 The number of eval()s is guaranteed to be higher than the number of processors.

simultaneously, we transmit many messages in succession (this increases the probability of overlap but does not guarantee that every pair will overlap).

size.l measures the time to send messages of different sizes.

broadcast.l measures the time required to perform both multicast and reverse multicast communications. In both cases M processes are eval()ed from the main one in order to execute the required in()s and out()s. M is varied.

overlap.l checks the effect of computation on communication and vice-versa. This is determined by: measuring the time required to eval() a computation-bound process; measuring the time required to send a message between two other processes (involving no computation); and then by measuring the time required for each when they are both run together.

arithmetic.l Measures the basic arithmetic capability of the processors by measuring the time for integer, double, and floating point addition, multiplication and a+b*c operations.

5. Assigning Weights to the programs

In order to generate a single benchmark figure for the BeLinda it is necessary to combine the results obtained from the above programs. The importance of each particular benchmark is largely determined by the type of application to be run on the architecture in question. However, to obtain an overall generic performance indicator we collected data from a number of different applications and used the average occurrence of the work primitives in question to weight the results produced by the BeLinda benchmarks[3]. The significant information for BeLinda is the ratio of the number of occurrences of each of the work primitives relative to the others.

Table 1 contains the static (compile time) frequencies of some attributes of the tuple space. It may be noted that some attributes being evaluated like latency, contention etc., are nontangible. We adopt the approach of giving equal weightage to each of the categories, and use the relative frequencies to differentiate between the different factors in the same category. The factors which increase the number of primitive operations being performed include the number of workers, number of times the loop is executed etc. The first of these is the dominating factor & varies drastically from application to application. Therefore, we shall use the static frequencies as the basis for our weight computation.

Table 2 gives the weights obtained. The first category has the basic Linda primitives. In this we had evaluated the costs of the primitives, and also the cost of the primitives with varying number of fields. We give equal weightage to both of these. Based on the relative frequencies, we arrive at the weights for each test as shown in the table 2.

The next category is for communication and synchronization primitives. MOst of the parameters being measured in this category are nontangible. We recognize the fact that they occur each time a tuple operation happens in some implementations, but not necessarily in others. We therefore, give each of them a uniform weightage. A study of Table 1 shows that most synchronization is of the form 1-1, M-1 or 1-M, with M being the number of workers. The times for 1-1 communication is measured in terms of latency and contention. Thus, in the multicast/ reverse multicast tests, we shall only consider the timings of 1-4 and 4-1 with equal weights. The number of workers in the computation varies depending upon the application and also might vary between different invocations of the same computation. Our choice of 4 workers seems suitable, in the

3 The data was collected from the set of ten programs which are included with the SCA linda package.

	#1	#2	#3	#4	#5	#6	#7	#8	#9	Total
# in()s	1	3	4	2	1	2	1	5	4	23
# rd()s	0	0	0	2		3			2	7
# out()s	1	6	4	7	1	4	1	7	5	36
# eval()s	0	1	1	1		1		2	2	8
# actuals										
1		2	4	2	1	2	2	8	11	32
2	1	1		2		3		4		11
3										
>3										
# formals										
1			2	3		4		3		12
2	1	1		1						3
3		2								2
>3										
Size of msgs (in long)	1-256	1 1-256	2K	N 1-M	-	1 256	1	1 <100	1	
synchronization	--	1-1	1-1	M-1	-	M-1		1-1	1-1	

Table 1. Static Frequencies

absence of any supporting evidence for a commonly occuring number of workers.

The size of the message being transmitted also varies depending on the application. We shall use the time for 256 bytes of data. It may be noted that in some Linda implementations (eg iPSC2), the unit of data in the Linda Block structure is char, while in some others (eg Sequent, it is long). Thus different sizes of messages need to be sent on different machines depending on the transmission units of the implementation.

The rest of the table can be interpreted in a similar manner. Another point of interest is how to arrive at a figure for scalability. The expected behaviour when we increase the number of workers doing a computation is an initial decrease in the time for completion followed by a level off and a subsequent increase. We also want the unit of measurement to be in time. Furthur, in keeping with the other parameters being measured in time units, we would like a low figure to imply a more scalable implementation. We thus define a measure as:

Scalability = Slope of the upward curve

This defenition is quite ad hoc in recognition of the fact that the figure would depend quite a lot on the workload. If we were to have an application with a lot of interdependencies, the time for a sequential execution would be the lowest, and if we have a perfectly parallel problem, we would end up with an almost linear drop and then a level off after the number of workers becomes equal to the problem size. Also important is the granularity of the eval(). For the same amount of parallelism, we can get different timings based on the granularity (with the bottom case being when the amount of computation is so low that the average cost per worker becomes same as the cost of doing a null eval().

Primitives	Occurance	local wt	weight of subgroup	weight of group	overall weight
in()	34.8%	0.3	0.5	0.25	0.0375
out()	54.54%	0.5	0.5	0.25	0.0625
rd()	10.6%	0.1	0.5	0.25	0.0125
eval()	12.12%	0.1	0.5	0.25	0.0125
actuals					
1	53.33%	0.5	0.5	0.25	0.0625
2	18.33%	0.2	0.5	0.25	0.025
formals					
1	20%	0.2	0.5	0.25	0.025
>1	8.33%	0.1	0.5	0.25	0.0125
Latency		1.0	0.2	0.25	0.05
Contention		1.0	0.2	0.25	0.05
1 - 4		1.0	0.2	0.25	0.05
4 - 1		1.0	0.2	0.25	0.05
Msg size		1.0	0.2	0.25	0.05
a+b*c		1.0	1.0	0.25	0.25
overlap1		1.0	0.5	0.25	0.125
overlap2		1.0	0.5	0.25	0.125

Table 2. Weights Obtained

In recognition of all these problems, we feel that it is not practical to include scalability as a basic measure of the implementation. We feel that it is a property of the implementation but it is not easily quantifiable. Blocked-in()s also depends quite a bit on the implementation. We therefore do not include these features in the final BeLinda figures.

In the case of overlap of computation and communication, the parameters of interest are the increase in computation time due to communication, and, the increase in communication time due to computation. These are listed in table 2 as overlap1 and overlap2.

6. Machines being evaluated

We have used BeLinda to evaluate three different Linda based parallel architectures: the Sequent Symmetry, the Intel iPSC/2, and the Cogent XTM. The architectures used in the evaluation included the following features:

Sequent Symmetry

The architecture consists of a shared memory, 8x80386 processors each with 32KBytes of writeback cache, 53MBytes/sec pipelined bus, 32MBytes of main memory, a Wietek floating point accelerator and an SCA Linda compiler. The Linda implementation contained a number of performance optimizations.

Intel iPSC/2

The architecture consists of a distributed memory, 32x80386 processors in a 4 cube configuration, an additional 80387 numeric coprocessor, 8MBytes of local memory per node, a Direct Connect Module per node to improve communication, 2.8 Mbytes/sec communication rate and an SCA Linda compiler.

Cogent Research XTM

Our configuration included two Cogent Research XTM workstations each containing two Inmos T800 processors, connected to a resource server containing 8 T800 processors each with 4 MBytes of memory. The processors in the resource server use a hybrid network of a shared bus and a crossbar (the bus is used for short messages, while a crossbar connection is used for heavy communication between two processors). The Linda implementation used in this case was Cogent's Kernel-Linda [21].

Kernel Linda has the following distinguishing features:

- Tuples are implemented as dictionaries (a Key & Value pair)
- Provides multiple & nested tuple spaces
- Has no eval() instruction
- Provides support for variable weight process creation
- Provides a set of language independent data types
- Has the restriction that the tuples have a single key
- Changes the semantics so that ordering of tuples is guaranteed.

7. Results and their interpretation

Having described the benchmarks and the environment in all the machines, let us now take a look at the results and try to interpret them in light of the features which the particular architecture or the compiler has. We shall go over the results in order of the tests.

7.1. Basic Linda primitives

7.1.1. Primitives

Table 3. summarizes the results. We immidiately see the benefits of a shared memory architecture. The cost of doing any Linda primitive is substantially less than on any other machine. The tuples in iPSC2 are distributed across the cube and costs of communication are incurred in tuple operations. XTM falls between these two extremes. It places the tuples in the node running the process doing the out(). The distributed memory is not utilized, but the underlying assumption must have been that if one achieves load distribution of eval()s, distribution of tuples on an out() would be a consequence.

Figure 1 plots the variation in the time for primitives as the number of nodes in the system changes. The figure is for the iPSC2. There are a couple of points of interest here. Firstly, the cost

	Linda Primitives			
system	in	rd	out	eval
Sequent (8)	15	14	14	29ms
iPSC2 (16)	1025	1163	358	1.7ms
XTM (8)	467	437	460	2.12s

Table 3. Cost of Primitives (in us)

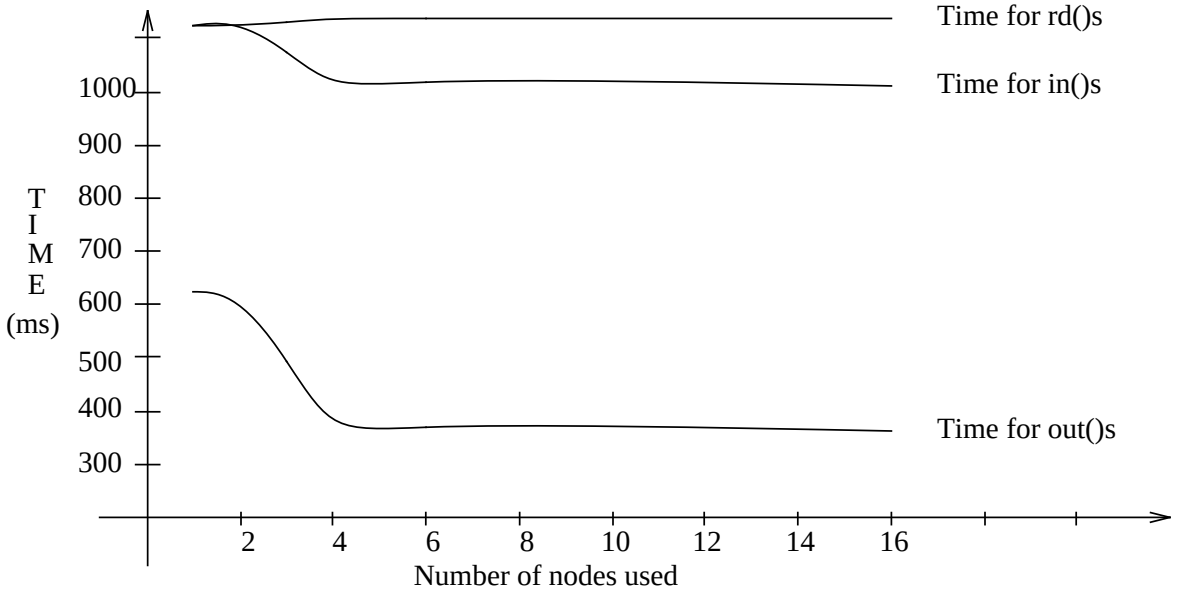

Figure 1. Time for Primitives in iPSC2 (in ms)

of the operations is higher for lower number of nodes and eventually becomes constant as the nodes are increased. This is surprising because when only one node is being used, communication costs are eliminated. But, since all the tuples are hashed onto the same node, there would be more collisions. The effect is most pronounced for an out() as expected. The out() semantics do not specify that the process should wait till the tuple has reached its final destination. The communication costs are therefore never part of the picture. Figure 2 is for the XTM. It says that the costs remain constant as the number of nodes change. This is because we have only one process doing all the operations and as such all the tuple operations are independent of the number of nodes in the system. The cost of rd() and in() are lower than that of out() because there is no communication involved in the tests.

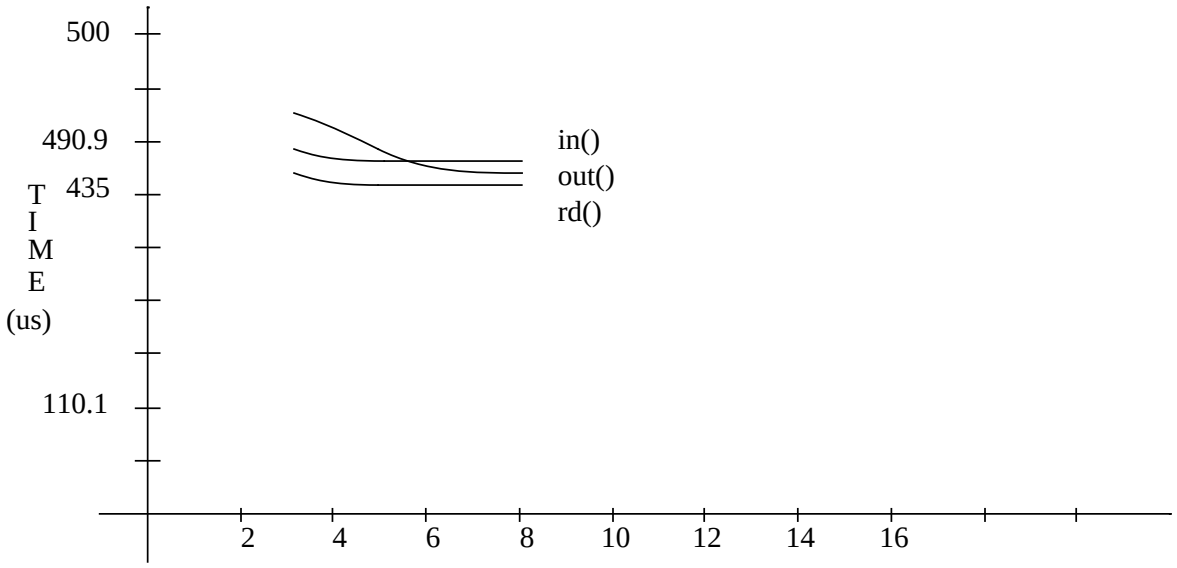

Figure 2. Time for primitives (on XTM)

7.1.2. actuals

The times to do a rd() and an in() are shown in tables 4. and 5.

	Actuals (read)				
system	1	2	4	8	16
Sequent (8)	14	14	14	14	14
iPSC2(16)	1046	1038	1048	1038	--
XTM(8)	437	808	1551		

Table 4. Time to rd() actuals (in us)

From the figures of iPSC2 and Sequent we see that the number of fields in the tuple does not effect the time to do a rd() or an in(). This is clearly because of the compiler preprocessing where the fields are all hashed onto one number. Also apparent is the fact that the times to do an in() or a rd() are almost the same. The dominating cost is therefore one of locating the matching tuple and copying to the applications address space, rather than the cost of removing it from the TS. The cost of doing a rd() or an in() increases linearly with the number of fields in the XTM. This is because of the fact that Kernel Linda does not allow you to access more than one value at a time. Thus, as the number of actuals (or formals) increases, correspondingly larger nesting of multiple

	Actuals (in)				
system	1	2	4	8	16
Sequent(8)	14	14	14	14	15
iPSC2 (16)	1047	1039	1050	1041	--
XTM(8)	467	838	1581		

Table 5. Time to in() actuals (in us)

tuple spaces needs to be used. Thus the cost to access more than 3 actuals is more in XTM than in iPSC2.

7.1.3. formals

We now change the number of formals in the tuple and find out the time to match. The times are listed in tables 6 and 7.

	Formals (read)			
system	1	3	7	15
Sequent	.34	.41	.56	.80
iPSC2	1.23	1.29	1.66	---
XTM	same as actuals			

Table 6. Time to rd() formals (in ms)

Figures 3 & 4 give the variation in time to rd() and in() formals as the number of computing nodes changes. As the number of nodes increases in the system, the cost of doing a rd() or an in() becomes constant after an initial increase. The point is that as the number of nodes increases, the percentage of hashes to remote bins also goes up. Therefore the communication cost adds to the time. Another way of looking at it is to vary the number of formals keeping the number of nodes fixed. Again we notice an increase in the time for computation. Also apparent is the fact that the cost of removing a tuple from the TS is negligible. In the XTM the times for actuals cannot be differentiated from those of the formals, because of the way multiple tuple spaces are configured. Thus, again we can expect a linear increase in time as the number of formals increases.

	Formals (in)			
system	1	3	7	15
Sequent	.37	.47	.68	1.04
iPSC2	1.23	1.31	1.66	---
XTM	same as actuals			

Table 7. Time to in() formals (in ms)

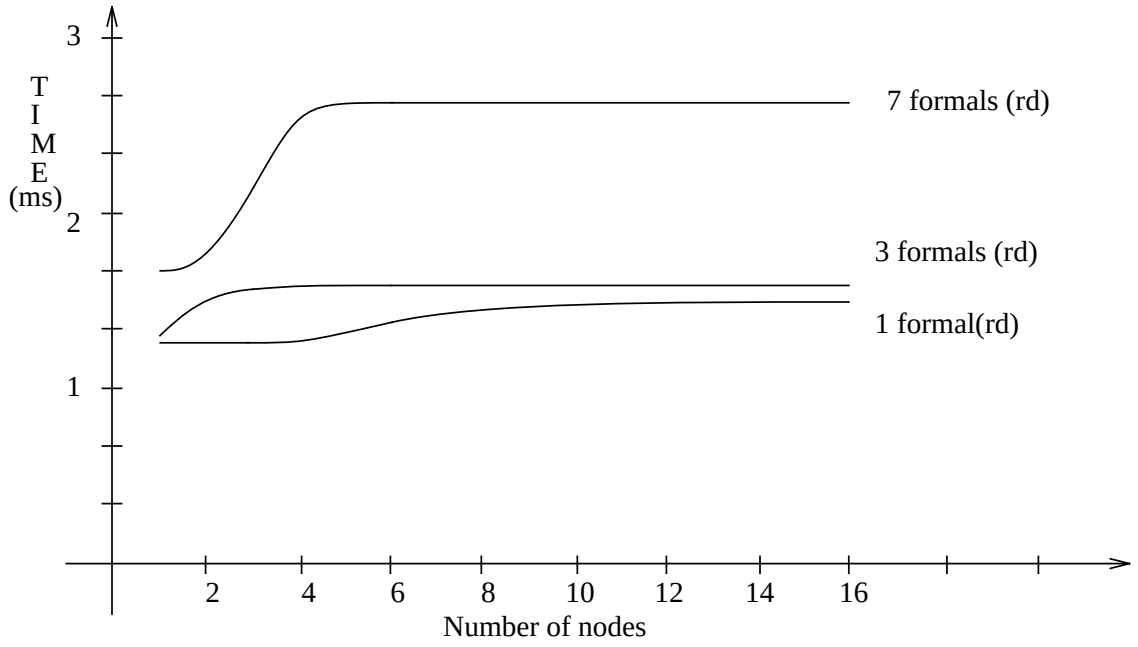

Figure 3. Time for rd() with different number of formal fields

7.2. Primitives for communication and synchronization

7.2.1. Latency

The time required to send a zero byte message cannot be measured in Linda. The first problem is that Linda presents us with a model of computation in which the user cannot explicitly ask an eval to be running on a particular node. Secondly, all out()s and in()s have to contain at least one key. Therefore, in measuring latency in Linda would be equivalent to the cost of synchronization between two processes. The times measured for the various systems are given in table 8. The cost of communication seems to be the highest in the XTM. Of course, the time obtained in the shared memory machine is the best.

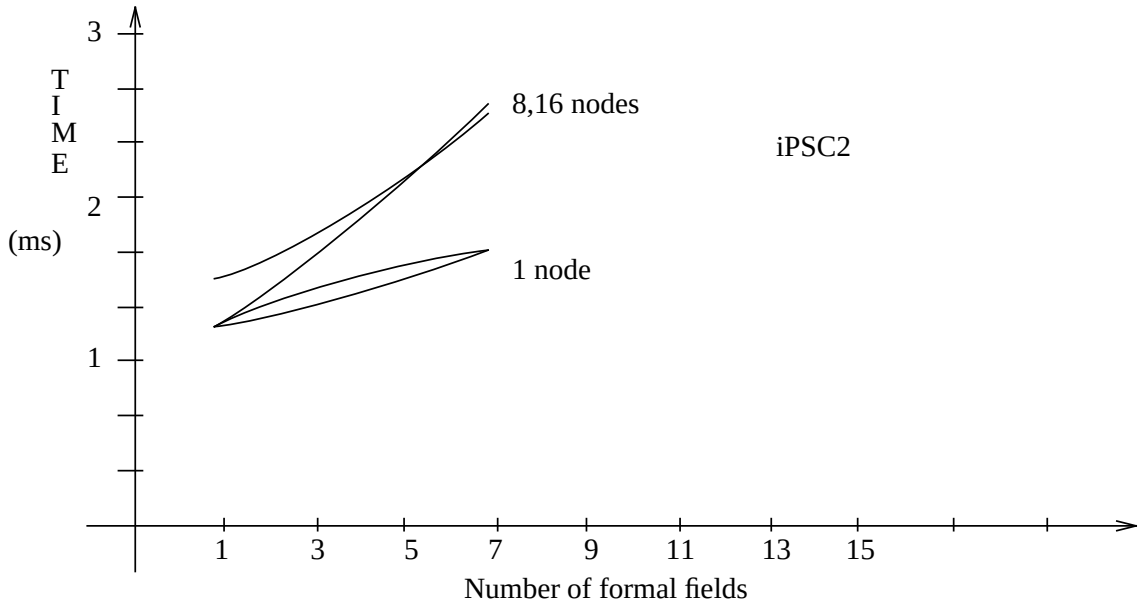

Figure 4. Time for rd() - another view

System	Latency
Sequent	.0455
iPSC2	1.051
XTM	7.631

Table 8. Latency (in ms)

7.2.2. Contention

The method for measuring contention has been detailed before. If the figure obtained here is any different from the figure for latency, we have degradation due to conflict in the use of channels. The times are given in table 9. We see a performance degradation by 33% in the case of the iPSC2 due to channel contention. In the case of the Sequent and the XTM, the timings are uneffected.

7.2.3. Multicast & reverse multicast

Table 10 presents the timings of M to 1 and 1 to M communications for different values of M. These represent the primitives for different kinds of synchronization between different numbers of processes. The time increases as synchronization is required among increasing number of processes. The time again is the highest in the XTM.

System	Contention
Sequent	.044
iPSC2	1.575
XTM	7.466

Table 9. Contention (in ms)

	1 - M				M - 1			
System	1 - 2	1 - 4	1 - 8	1 - 16	2 - 1	4 - 1	8 - 1	16 - 1
Sequent	.10	.60	2.0	4.3	.30	1.3	3.1	6.7
iPSC2	1.0	2.9	6.8	14.0	2.0	5.2	11.6	23.8
XTM	4.05	7.46	15.1	59.2	9.67	14.9	44.7	93.1

Table 10. Multicast & reverse multicast (in ms)

7.2.4. Message size

Figure 5 plots the time taken for different size messages for the three machines. While the units of transmission in Sequent is longs, the unit in iPSC2 is char. What is of interest is the trend. While the time for transmission remains constant (or increases at a very small rate) in the XTM and the iPSC2, the change is more dramatic in the Sequent. The iPSC2 uses the DCM for its internode communication, by which it establishes the path using a probe, after which the data is DMAed between the nodes. Thus the overhead with increase in message size is minimal. Whereas in Sequent, the message has to be copied explicitly between the processes. The cost therefore, increases linearly with message size.

7.3. primitives for computation

Sequent and iPSC2 both have intel 80387 on their node processors. But XTM does not have any floating point coprocessor. We see the effect of this reflected in the times for floting point computations in table 11.

7.3.1. Primitives for overlap of computation and communication

In the case of Sequent we see that the times for computation or communication are uneffected by the presence of the other (see table 12). Mostly the communication is not affected by the presence of computation. But we see a defenite degradation in computation times in the presence of communication in XTM and to a lesser extent on the iPSC2.

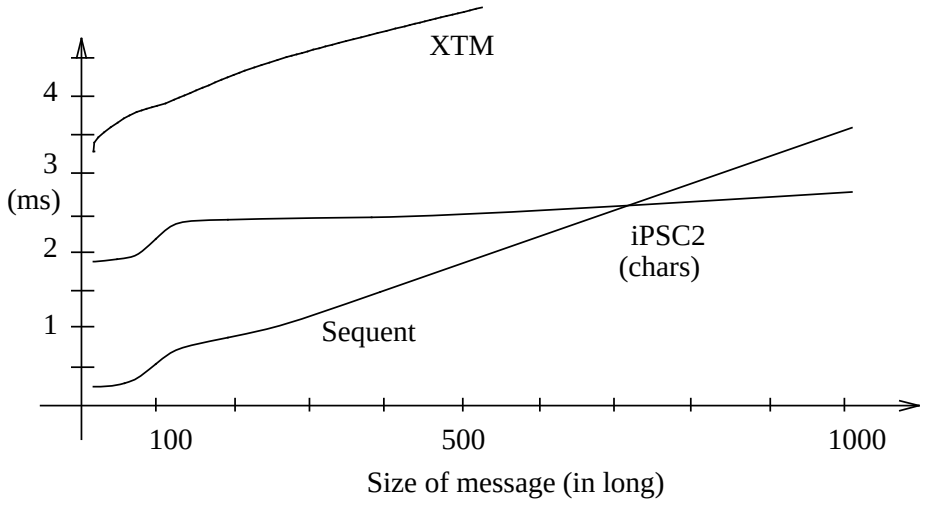

Figure 5. Message Transmission time

System	int			float			double		
	add	mult	a+b*c	add	mult	a+b*c	add	mult	a+b*c
Sequent	3.0	3.0	4.0	1	1	1	1	1	-
iPSC2	1.3	1.3	1.3	2	1	4	1	1	-
XTM	3.6	4.0	3.8	4	4.1	6.2	4.6	5.4	-

Table 11. Basic arithmetic time (in us)

A measure of performance which has been proposed is the *overlap time*. It is defined as
Overlap Time = Time for computation + time for communication - time for both
By this measure Sequent should have the maximum overlap, iPSC2 the next and XTM the least.

7.4. Other issues

7.4.1. Scaling effect

Figure 6 shows the variation in times with a constant workload and changing the number of eval()s. What one expects is an initial decrease in the time as the number of workers increases. But eventually this should level off and then go upwards as we keep increasing the number of eval()s. This is pretty much the story in Cogent XTM and Sequent. The XTM was running with three processors, while the Sequent had 8. The iPSC2 also shows the effect but the upward curve is less pronounced. It was running with 32 processors. Some of the factors contributing to this performance would be the low cost of starting a worker of on another node, low cost of

System	Communication	communication and computation	Computation	computation and communication
Sequent	.27	.27	.22	.22
iPSC2	21.134	21.244	0.752	1.815
XTM	4.67 sec	4.94 sec	.0874 sec	.124 sec

Table 12. Interferance times

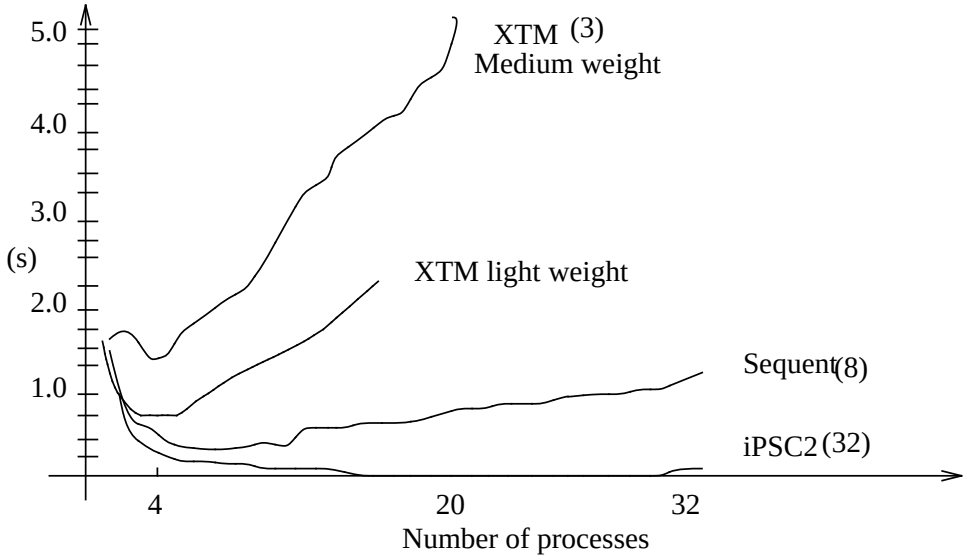

Figure 6. Scaling effect - constant workload

communication & more number of processors.

Figure 7 shows that time for various number of **evals()** on each host using eight processors. Amazingly, the iPSC/2 comes out ahead, in fact it is more than twice as fast as the Sequent for more than eight **evals()**. One possible explaination could be the fact that the iPSC/2 implementation loads a copy of the main program on each node processor. Evals could then be activated by simply sending a message to another processor requesting it to exeucute a particular procedure. On the other hand the Sequent needs to perform a fork of an extremely heavyweight process. Still, this matter needs to be futher investigated.

The Cogent Research XTM has the worst times. Kernel Linda doesn't have an eval(), rather it provides routines for creating light to heavyweight processes. Unlike SCA's eval() which makes use of the UNIX fork(), Kernel Linda must create a process from an executable file.

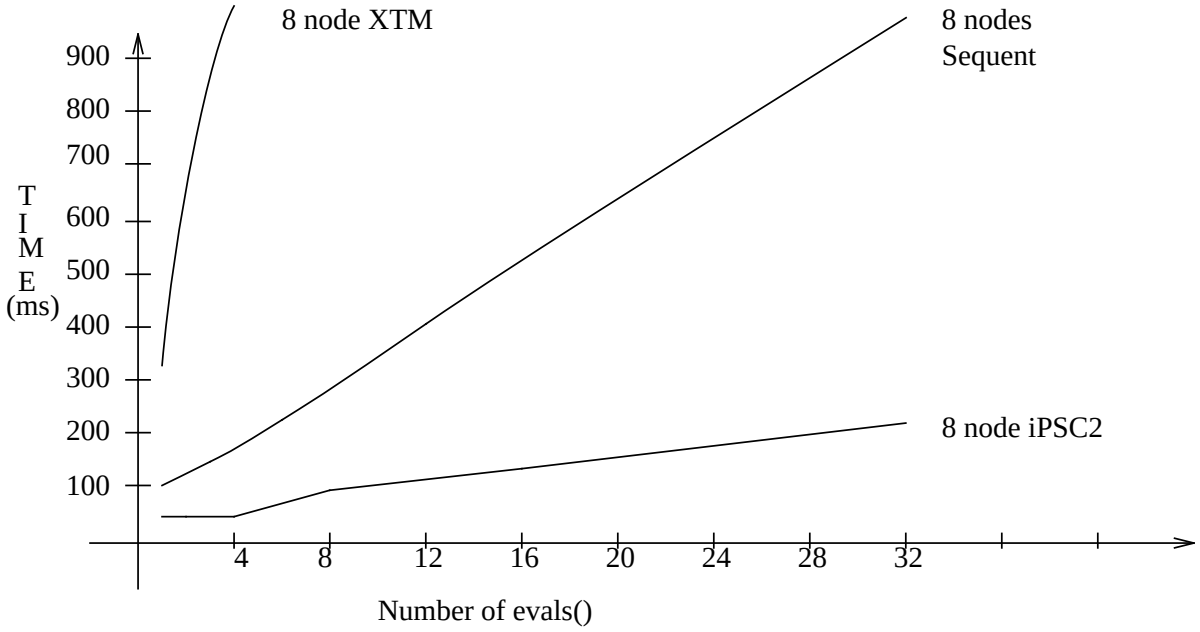

Figure 7. Cost w/ varying workload & eval()s

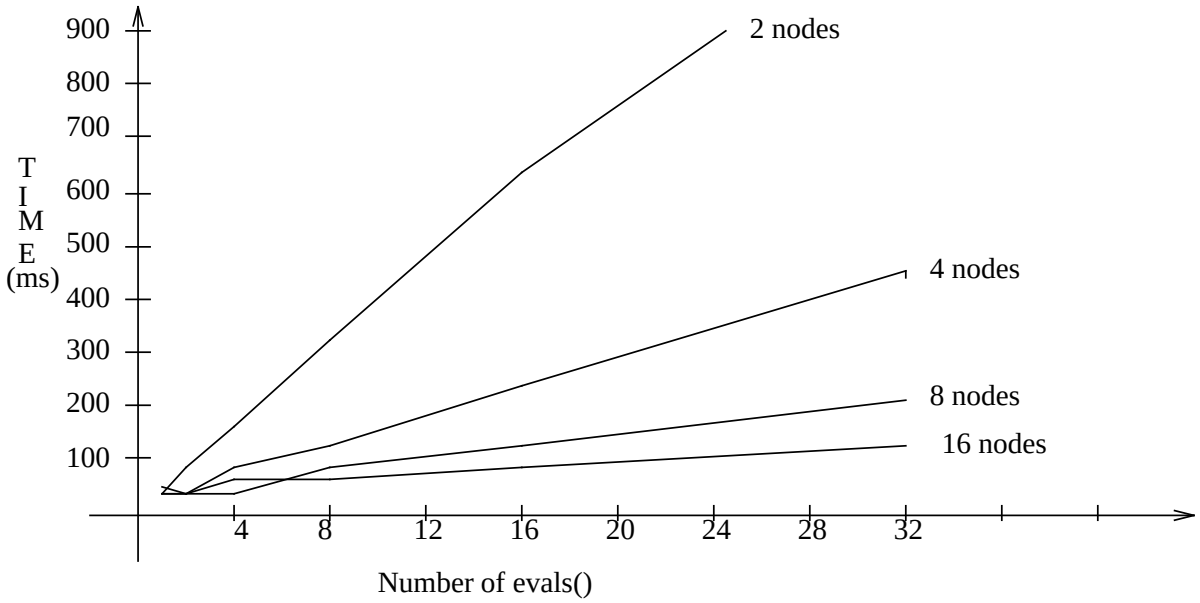

Figure 8. Cost with varying number of nodes, iPSC2

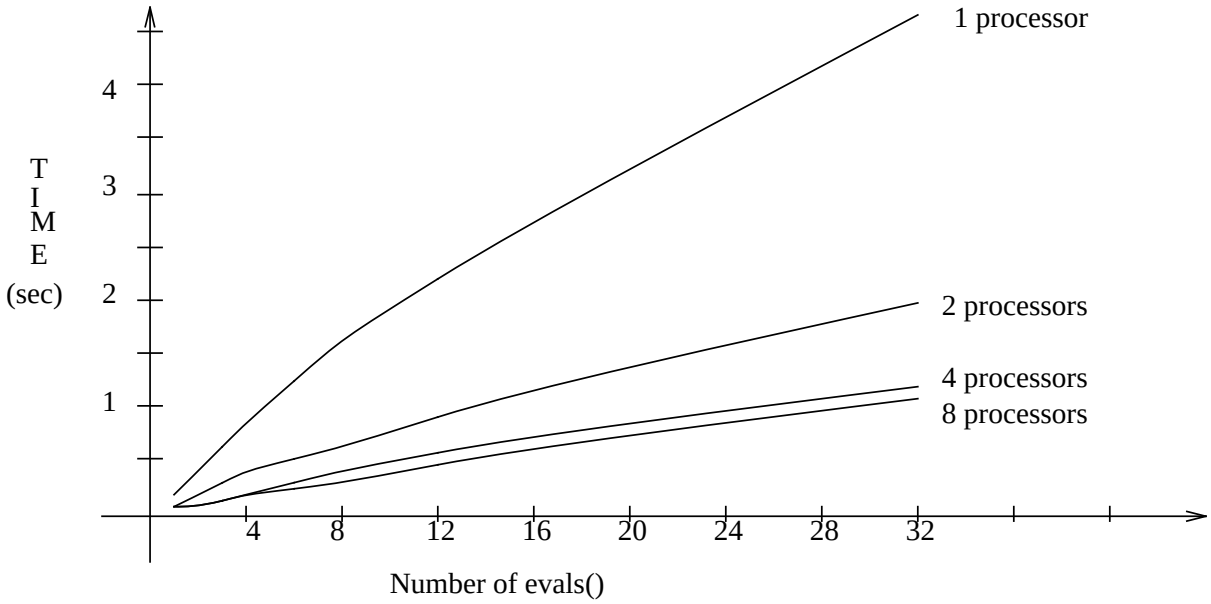

Figure 9. Cost with varying number of nodes, Sequent

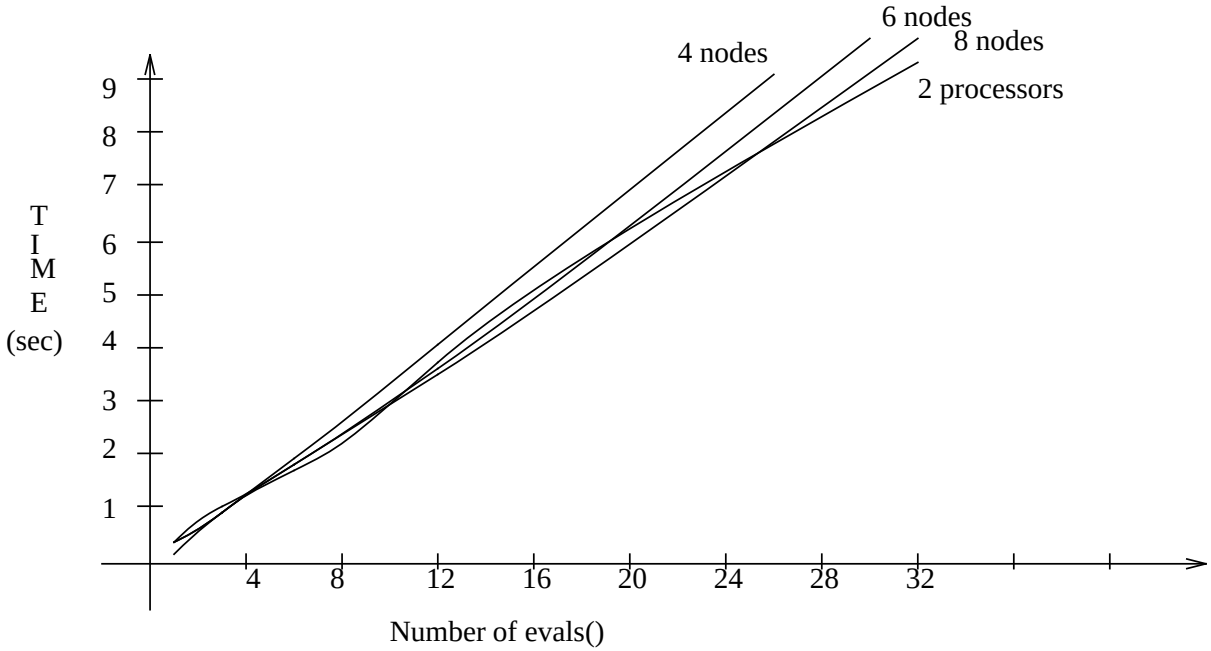

Figure 10. Cost with varying number of nodes, XTM

Process creation may involves transfering an entire program from one processor to another. Note that figure 10 shows the best time for a two processor systems. One of the reasons is that all processes are limited to two processors so there isn't a great need to transfer programs between processors. The figure clearly shows that the XTM does not scale very well. As the number of processors is increased in the system, the time for the eval()s remains the same.

7.4.2. Blocked-in()s

System	10 workers	10 workers and 10 blocked workers
Sequent	400ms	1100 ms
iPSC2	307 us	581 us
XTM	583 ms	891 ms

Table 13. Time (in us) for computation in presence of blocked workers

When a process blocks, the process spins on the tuple in Sequent (see table 13). Therefore all the other processes suffer a degradation in their time. The effect is less pronounced in the others (since they suspend).

8. Discussion of results

The final beLinda figures for the three implementations are given in table 15.

System	BeLinda Figure
Sequent	489.68 uS
iPSC2	1006.10 uS
XTM	66650.61 uS

Table 14. Final BeLinda figures

As expected, Sequent gives the best figures for BeLinda (see table 14). More surprising is the fact that iPSC2 does only twice as bad as the shared memory machine. A lot can be attributed to the fact that iPSC2 scales pretty well, and also that its communication costs are almost constant and independent of the message size. The figure for XTM is pretty bad. A lot can be attributed to its extremely bad communication between processes. Because of this, every communication and synchronization primitive suffers. Also, it does not scale very well. The concept of multiple tuple spaces causes the time for primitives on more than one value field to increase linearly. Thus, although it has better time than the iPSC2 on the basic in(), rd() operations, it winds up far behind

overall.

Based on the BeLinda figures we can say the following

- Communication plays a big role in the implementation of any parallel architecture. What is apparant is that even with the most effecient implementation of the Linda primitives, if the communication primitives are not effeciently implemented, the end result is not good. Even a shared memory machine like Sequent has bad communication overheads, and can clearly benefit from memory mapping between address spaces of processes.
- The choice of the weights determines the final time obtained from the benchmark suite. In the face of some nontangible properties being measured, we have attempted to make a selection of weights based on equal importance to all the categories. This was partly in recognition of the fact that the implementations in which the these factors contribute to other primitives being measured (like communication issues effect Linda primitives in distributed memory implementations), the factors would inherently be given more importance anyway. The major point is that we are able to obtain figures for different parameters, thus each user can choose his own weights to identify the usefulness of the results to his application.
- Finding a good measure of scalability is very difficult. This is partly because of its extreme sensitivity to the workload at hand, amount of parallelism inherent in the workload, the amount of communication overheads involved, and the granularity of the eval()s. In the face of so many variables, we approached the problem from several angles, but could not come up with a practical way in which to answer the question. We therefore concluded that scalability is not a primitive operation, but is a property of the implementation which depends on several other primitives.

9. Conclusions

In the absence of portable benchmarks and formal analytical methods of measuring the performance of parallel machines, the experiences with using a software architecture seems encouraging. It provides a convenient and quick way to benchmark parallel machines, possibly with different architectures.

Apparant is the fact that the degree to which conclusions can be drawn about the native architecture is limited by the level of optimizations in the implementation of the software architecture. The choice of the software architecture is important, in that it should not be biased towards any particular model of parallel computation.

We have implemented the set of benchmarks on three parallel machines with different architectures: a shared memory machine, a distributed memory machine and a machine with a hybrid architecture. The fact that we could do that seems to imply that the use of a software architecture would satisfy the demand for portable and easy to use benchmarks for parallel machines.

References

1. W. D. Hillis, *The Connection Machine*, MIT Press, 1985.
2. T. Lovett and S. S. Thakkar, ‘‘The Symmetry Multiprocessor System,’’ *International Conference on Parallel Processing*, pp. 303-310, 1988.
3. G. C. Fox et.al., in *Solving problems on Concurrent processors*, Prentice Hall, 1988.

4. S. Arlauskas, ‘‘iPSC/2 system : a second generation hypercube,’’ in *Proceedings of 3rd conference on Hypercube Concurrent computers and applications*, ed. G. C. Fox, pp. 38-42, ACM, 1988.

5. Intel, *iPSC/2 Users Guide*, Intel Scientific Computers, 1987.

6. C.A.R. Hoare, ‘‘Communicating Sequential Processes,’’ *Communications of the ACM*, vol. 21, pp. 666-677, August 1978.

7. E. Shapiro, ‘‘Concurrent Prolog: A Progress Report,’’ *IEEE Computer*, pp. 44-58, August 1986.

8. Strand Software Technologies, *STRAND88 Technical Description*, August 1989.

9. L. Bomans and D. Roose, ‘‘Benchmarking the iPSC/2 hypercube multiprocessor,’’ *Concurrency: Practice and Experience*, vol. 1, no. 1, pp. 3-18, September 1989.

10. A. Kolawa and S. W. Otto, ‘‘Performance of the Mark II and Intel Hypercube,’’ Technical Report 254, Caltech Concurrent Computation Program, February 86.

11. D. C. Grunwald and D. A. Reed, ‘‘Benchmarking Hypercube hardware and software,’’ Technical Report, University Of Illinois, November 86.

12. J. Martin, ‘‘Performance Evaluation of Supercomputers and Their Applications,’’ in *Parallel Systems and Computations*, ed. G. Paul and G.S. Almasi, pp. 221-235, Elsevier Science, 1988.

13. C. Fraboul, ‘‘MIMD parallelism expression, exploitation and evaluation,’’ in *Supercomputing*, ed. A. Lichnewsky and C. Saguez, pp. 155-170, Elsevier Science, 1987.

14. J. L. Gustafson and S. Hawkinson, ‘‘A Language-Independent Set of Benchmarks for Parallel Procesors,’’ Technical Report, Floating Point Systems, April 1986.

15. H. Flatt and K. Kennedy, ‘‘Performance of parallel processors,’’ *Parallel Computing*, vol. 1, no. 12, pp. 1-20, 1989.

16. R. Bjornson, N. Carriero, D. Gelernter, and Jerrold Leichter, ‘‘Linda, the Portable Parallel,’’ Yale Univ. Dept. Comp. Sci. RR-520, January 1988.

17. N. Carriero, ‘‘Implementing Tuple Space Machines,’’ Doctoral Dissertation, Yale University, 1987.

18. N. Carriero and D. Gelernter, ‘‘The S/Net’s Linda Kernel,’’ *ACM Transactions on Computer Systems*, pp. 110-129, May 1986.

19. S. Ahuja, N. Carriero, and D. Gelernter, ‘‘Linda and Friends,’’ *IEEE Computer*, vol. 19, no. 8, pp. 26-34., August 1986.

20. N. Carriero, D. Gelernter, and Jerrold Leichter, ‘‘Distributed Data Structures in Linda,’’ *Proceedings of the thirteenth ACM Symp. on Principles of Prog. Lang.*, January 1986.

21. W. Leler, ‘‘Linda meets UNIX,’’ *IEEE Computer*, pp. 43-55, February 1990.