Experiences with Belinda: A Synthetic Linda Benchmark for Parallel Computer Platforms

Srikanth Kambhatla, Jon Inouye, Jonathan Walpole

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-003

January, 1990

EXPERIENCES WITH BELINDA: A SYNTHETIC LINDA BENCHMARK FOR PARALLEL COMPUTING PLATFORMS

Srikanth Kambhatla, Jon Inouye and Jonathan Walpole

Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
19600 NW von Neumann Drive
Beaverton, OR 97006-1999

Abstract — Recent advances in the field of parallel processing have produced a diverse selection of architectures and programming styles. Such diversity presents significant problems for tasks such as program portability and performance evaluation. We argue that the issues of portability and performance evaluation are related, and that benchmarks for parallel machines should be easily portable. To this end, we have developed a benchmark for a portable software architecture based on Linda tuple space. Our benchmark is called BeLinda.

In this paper we overview the design of BeLinda, we present the results of running BeLinda on three radically different architectures, and we discuss our experiences with taking this particular approach to benchmarking parallel architectures. We identify some *primitives* that are generally used in parallel applications, perform our evaluation with respect to these primitives and attempt to combine the results to a single number. In combining the results, we are faced with the problems of assigning suitable weights to the different programs. These problems include the identification of a *typical* workload, determination of the frequency of occurrence of the primitives in the workload, and conversion of results into the same units. Our approach allows a quick and easy evaluation of the strengths and weaknesses of the machine being evaluated, but we conclude that it is not a realistic idea to attempt to reduce the results to a single overall number.

1. Introduction

Recent advances in the field of parallel processing have produced a diverse selection of architectures and programming styles. At the architectural level a broad distinction can be made between parallel machines that execute in single instruction stream, multiple data stream (SIMD) mode (an example is the Thinking Machines CM-2 [1]), and those which execute in multiple instruction stream, multiple data stream (MIMD) mode (examples are the Sequent Symmetry [2] and Intel iPSC/2 [3]). A further distinction can be made within the MIMD category based on the characteristics of the machine's address space(s). If the address space consists of a set of disjoint address spaces the machine is categorized as a distributed memory distributed address space machine (an example is the Intel iPSC/2). If on the other hand the machine has a common global address space it is categorized as a distributed memory shared address space machine (an example is the BBN Butterfly [4]). The above architectural categories have also been defined as Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA) and No Remote Memory Access (NORMA) [5]. Further architectural distinctions can be made within the NUMA category according to the degree of uniformity of a machine's memory access times.

Such diversity in architecture induces a diversity in programming styles, since most architectures have their own associated paradigm for parallel computation. This diversity is most apparent with respect to the issues of synchronization, communication and location.

The diversity in architectures and programming styles is a result of broad and varied research in the area of parallel processing. However, it also has several consequences. Among the most significant of these are the lack of portability of parallel programs, and the difficulty of making comparative performance evaluations between different machines and programming environments.

To satisfy the requirement for portability, a programming model must span the barriers of architectural diversity and present a uniform model of parallel computation to the programmer. Such a model must present parallel programming constructs at a sufficiently high level of abstraction and must be easy to implement efficiently. A number of programming languages have attempted to meet these demands, including: CSP [6], Concurrent Prolog [7], Strand [8], and Linda [9, 10].

The requirement for easy performance evaluation of different machines and programming environments is clearly related to the issue of portability. A benchmark for parallel architectures must also evaluate features at an appropriate level of abstraction and must be easy to implement. However, previous work in the area of benchmarking parallel architectures has either:

1. compared different implementations within the same architectural category. This approach was taken by Bomans [11], Kolawa [12], and Grunwald [13] to evaluate different versions of iPSC hypercube, MARK II and the NCube.

2. evaluated performance with respect to a specific application program. This is the approach followed by Martin [14], Fraboul [15], and Gustafson [16].
3. performed analytical studies [17].

Each method has problems of its own: the first is only applicable to machines which fall into the same architectural category. The second does not facilitate an analysis of the strengths and weaknesses of the machine under study. Further, the direct relevance of the results to applications in domains other than the one to which the benchmark program belongs is limited. The third does not provide any information on the performance of any specific implementation. It is often the case that users want to gain insight into the cost of running various distinct types of application programs either on a particular architecture or on a number of perhaps radically different architectures.

In this paper we present a benchmark that is portable, easy to implement and reflects performance measures for different primitives of parallel computation across several different machine types. The benchmark, called BeLinda, is based on Linda tuple space 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 for a particular program(s). However, a bad figure does not necessarily distinguish between a bad architecture, a bad Linda implementation or both. The experiences we had in building and using this benchmark are discussed later in the paper.

Our use of Linda is also based on the notion that Linda provides a software architecture not unlike the instruction set architecture of a sequential machine. In both cases there is a distinction between the architecture that is visible to the programmer and the lower level implementation of the machine. In a sequential machine the lower level details include features such as the presence of a cache, pipelines etc, which improve performance but are not visible to the programmer. In parallel architectures the lower level details include additional issues relating to the communication network, processor and memory configuration of the machine. In either case, however, a benchmark must provide insight into the performance of the architecture at a level which is visible to the programmer.

While high level languages like FORTRAN, and C are supported on a variety of architectures, the programmers see the underlying architecture and have to explicitly introduce code for message passing or shared data. Therefore the code written for a hypercube is no longer suitable for a shared memory machine like the Sequent Symmetry. Linda however defines its own software architecture which masks out the native machine architecture and makes its code highly portable. Thus, we were able to use the same benchmark suite, which has been written in C-Linda, on the iPSC/2 and the Sequent Symmetry with only minimal changes in the code (due to the minor differences in the implementation of Linda on these machines).

The BeLinda suite can be used in two ways. It can be used to generate a database of results indicative of the performance of the machines under study. This facilitates comparision of different machines with respect to specific parameters of interest. In many cases though, the user is interested in the performance of the machines in some particular application domain. For those cases, specific weights which are appropriate for the domain need to be chosen. Applications differ in issues like the amount of communication, amount of parallelism, granularity of tasks, communication patterns, and the amount of computation. This diversity prohibits the presence of a set of general purpose weights suitable for all applications.

In the following section we discuss the design of BeLinda, we define our choice of performance measures for parallel computation, and we give an outline specification for the individual benchmarks which together comprise BeLinda. Section 3 presents the results which we obtained by running BeLinda on three radically different parallel architectures. In section 4 we discuss our experiences with taking this approach. In section 5 we overview related work, and finally we conclude the paper in section 6.

2. Overview of BeLinda

The overall design and goals of BeLinda were influenced by the following characteristics listed by Gustafson [16]. A benchmark should:

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

In order to meet requirement "1" BeLinda identifies a set of work primitives, or constructs, which are common across various parallel applications. The performance figures associated with these primitives can then be used as performance indicators for different parallel applications.

Requirement "2" is concerned with the issue of portability. The use of Linda as a basis for BeLinda ensures that BeLinda shares the same high degree of portability as Linda.

Requirement "3" is largely a matter of convenience since a one dimensional ranking allows straight-forward comparisons between architectures. BeLinda provides a statistical base of data, however, this data can be weighted and combined to produce a single performance figure. The issue of assignment of weights to the various results produced by BeLinda is discussed in a later section. It has been our experience that reducing the results to a single number is unrealistic.

The remaining requirements are concerned with the ease of implementation and verification of the benchmark. Most of our algorithms indeed fit into one page. We hope that the details provided herein, and in the accompanying paper [18] are sufficient for reproducibility of results. Although "5" is desirable, it is, in general, difficult to obtain.

Identification of work primitives

The work primitives identified within BeLinda fall into a number of general categories: the basic Linda primitives; constructs for communication and synchronization; primitives for computation; evaluation of overlap of computation and communication; and others. Each of these is discussed in more detail in the following sections.

Basic Linda primitives: Linda defines four primitive operations: **in()**, **out()**, **rd()** and **eval()**. The efficiency of Linda on any architecture depends to a large extent on the implementation of these primitives. Therefore, these are included in the first set of BeLinda work primitives.

A second issue which must be addressed in this category is the use of Linda's features for content addressability. The important factors here include the number of fields in a tuple and the number of actuals in these fields. It is expected that as the number of fields in a tuple increases, so will the time required to perform the match. However, several Linda compilers optimize tuple references and reduce the time for matching by performing *tuple analysis*. Tuple analysis has an effect on the overall time only in the case of shared memory machines. In distributed memory machines the time for Linda operations are dominated by the communication costs between the processors. Thus the improvements resulting from optimizations of match operations are not as dramatic.

Communication and synchronization: In Linda based systems synchronization is no longer a primitive operation of the native architecture: rather, it is achieved through the tuple space by means of a single field tuple exchange between two processes. In order to ensure correctness, the single field in the tuple is an actual.

Similarly, communication also takes place via the Linda tuple space. Programmers regard their machine as a large tuple space in which parallel processes crawl over distributed data structures. BeLinda considers the following work primitives for communication:

Latency: is a measure of the time required to send a zero length message between two nodes in the architecture. In a distributed memory machine this requires the transmission of data between two nodes over a communication network, whereas for shared memory systems it reduces to the cost of copying or mapping between memory locations.

Contention: is a measure of the overhead caused by interference between concurrent messages. This is highly dependent on the number of redundant paths and on the presence of full duplex communication in the interconnection network.

Multicast: is the cost of sending a message from one node to k other nodes. In the degenerate case k may be the number of nodes in the network in which case the operation becomes a broadcast.

Reverse Multicast: is the cost associated with sending a message from each of k-1 nodes to a single node. This is intended to model operations such as a process waiting on a barrier.

Message Size: is a measure of the effect on transmission time of increasing message length. In some shared memory systems, message passing may be implemented via memory mapping in which case message length does not affect transmission time.

The work primitives listed above are intended to model typical operations in parallel applications. In BeLinda these primitive operations are defined in terms of sets of Linda operations. Several other communication and synchronization parameters are also of interest in the context of parallel machines, for example the use of blocking or non-blocking message exchange and the design of the routing algorithm in the interconnection network. However, these are not immediately relevant for a benchmark at this level of abstraction.

Computation: Despite the fact that Linda is now being used in a variety of application domains, including systems programming [19], the majority of applications continue to be in the scientific computation domain. Frequently reported work primitives for computation in this domain are: the *add* and *multiply* times for *integers*, *floats* and *double precision floats*. BeLinda also makes use of these computation based work primitives.

The overlap of computation and communication: In some parallel architectures communication between non-neighboring nodes is handled by separate communications controllers in the sender, receiver and intermediate nodes. In this case the node processors of the intermediate nodes should not be affected by the passing traffic. In order to measure the interference between computation and communication we consider both the effect of communication on the computation of other processors, and the effect of computation of other processors on the communication between two cooperating processes.

Others: A number of other significant issues that do not fit well into any of the above categories are also considered in BeLinda:

Scaling-Effect: is a measure of the scalability and performance characteristics of a system. BeLinda calculates this factor by measuring the time required to perform a constant workload using an increasing number of **eval()**s.

Blocked in()s: is intended to determine whether processes block or spin when waiting on an **in()**. Spinning may or may not cause a degradation in performance depending on the average amount of time processes spend waiting on **in()** operations and the amount of processing required to manage a queue of blocked processes. The goal for BeLinda is to determine the level of degradation associated with blocked **in()**s.

The BeLinda Specification

BeLinda consists of a set of eleven individual programs[a] that 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 operations of each type[b]. 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[c] 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 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.

Combining the BeLinda 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 motivation is that a single performance figure facilitates one-dimensional ranking among the machines being evaluated. The general approach is to assign weights to each program based on the frequency of occurrence of the primitives and to perform a weighted sum of the results to get a single figure. However there are several problems here.

The importance of each particular benchmark is largely determined by the type of application to be run on the architecture in question. This makes the set of weights specific to the application domain. However, to obtain an overall generic performance indicator we collected data from different applications and used the average occurrence of the work primitives in question to weigh the results produced by the BeLinda benchmarks[d]. The significant information for BeLinda is the ratio of the number of occurrences of each of the work primitives relative to the others.

It is interesting to note that we had considerable difficulty arriving at weights for some of the work primitives. This was due either to the fact that the primitives were non-tangible (e.g. scalability and blocking **in()**s), or it was due to the difficulty of analyzing the sample applications for certain features. These problems will be addressed in more detail in section 4.

3. A BeLinda Based Evaluation

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

(a) The programs total about 800 lines of C-Linda code. (b) Where *n* is large enough to ensure that the overall time is discernible.

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

(d) The data was collected from the set of ten programs which are included with the SCA Linda package.

Sequent Symmetry

The architecture consists of a shared memory, 8 80386 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, 32 80386 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 Research's Kernel-Linda [20].

Results

We ran the Belinda benchmarks on each of the above three architectures, and combined the weighted results to produce a single figure for ease of comparison. In this section some of the more interesting results are discussed. For a more complete description of the benchmark and the results obtained see [18].

One effect of using a shared memory machine is immediately apparent in the results of the benchmark for the basic Linda primitives (see Table 1). For all the Linda primitives the shared memory architecture of the Sequent Symmetry exhibits significantly better performance than the distributed memory architectures.

Another area in which the Sequent machine "wins" is in the benchmark which tests the effect of the number of fields in a tuple (see tables 2 and 3 and figures 3, 5 and 6). Surprisingly, the number of fields does not affect the cost of an **in()** or a **rd()** on the Sequent. However, in this case the improvement is due to analysis at compile time rather than architectural advantages. A peculiarity of the XTM Linda implementation causes the cost of doing **in()** and **rd()** operations to increase linearly with the number of fields in the tuple[e].

The benchmark of primitives for communication and synchronization again demonstrated an advantage of using a shared memory architecture such as the Sequent. The shared memory architecture exhibited considerably better figures for both latency and the two types of multi-cast tested (see tables 4 and 6). In all cases the cost of communication dominated the final figures. The only system which demonstrated any contention problems was the iPSC/2 (see tables 4 and 5). A significant problem encountered in all these tests however, was the construction of the equivalent of a null message transfer. In Linda the degenerate case for communication must involve the transfer of a tuple containing at least one field.

The benchmark to test the effect of message size reveals an interesting characteristic of the Sequent Linda implementation (see figure 4). In the case of the iPSC/2 and XTM architectures the cost of transferring varying sized messages remains fairly constant, whereas the Sequent implementation demonstrates a linear increase in cost with increasing message size. We suspect that this is due to the cost of copying tuples between applications. A better approach might be to implement large tuple exchanges in shared memory using memory mapping rather than copying.

The test for interference between computation and communication again showed the Sequent architecture coming out on top (see table 7). The Sequent demonstrated virtually no interference, whereas the XTM machine showed a definite degradation in computation times in the presence of communication. This was true to a lesser extent on the iPSC/2.

An area in which we expected the distributed memory machines to demonstrate an advantage was in the effect of scalability (see figure 1). The iPSC/2 did in fact come out ahead in this benchmark, however the Sequent was second and the XTM third. A close examination of the results and the various Linda implementations has led us to believe that our results are completely dominated by the effects of the Linda implementation rather than the underlying hardware architectures: the iPSC/2 implementation uses its own distributed virtual memory system to optimize the performance of eval; the Sequent implementation builds eval on top of a fork() system call; and since the current Cogent Kernel Linda implementation doesn't have an **eval()**, the underlying operating system mechanism for process creation must be used.

The only system which causes processes to spin when they are blocked on **in()**s is the Sequent (see table 8).

Combination of the results

The final BeLinda figures for the three implementations show a clear "win" for the Sequent, with the iPSC/2 second and the XTM third. A large part of the overall result can be attributed to the relative costs for communication in the various architectures since the basic costs of communication underly many of the work primitives that were tested in the benchmarks. Another very significant factor in the overall result is the choice of weights for conveying the relative importance of the figures that were produced by each benchmark. It was particularly difficult to arrive at a suitable weight for the figures generated by benchmarks such as the one for scalability. A final issue which confuses the results somewhat is the effect of different levels of optimization in the Linda compilers of the various architectures. These issues will now be discussed.

(e) In Cogent Linda, tuples with multiple fields must be supported using multiple tuple spaces since individual tuples can only contain two fields. Consequently, as the number of actuals (or formals) increases a correspondingly deeper level of nesting of tuple spaces is required.

4. Experiences with BeLinda

Our experiences with designing and using BeLinda fall into two general categories: good and bad. On the positive side, we were pleasantly surprised by the small amount of time and effort required to implement and run BeLinda on three radically different parallel architectures. We believe that this success is largely due to the approach of using a portable software architecture as the basis for the benchmark. In addition, the generation of a single overall performance measure facilitates quick comparisons. Consequently, BeLinda provides a fast and efficient method for comparing different parallel computing platforms.

However, we encountered several problems during our research. Firstly, the use of a software architecture means that the effectiveness of evaluation depends on both the implementation of that software architecture (Linda) and on the underlying hardware architecture. In several cases it was not possible to determine whether a particular performance measurement was due to the hardware architecture or the Linda implementation. This means that the results must be interpreted as representative of the combination of architecture and implementation. The usefulness of the results for evaluating the underlying hardware architecture alone depends on the degree of optimization in the associated Linda implementations. This argument is also true for many sequential machine benchmarks, the results of which are generally dependent on compiler optimizations.

A second problem encountered because of the use of Linda was that it hides certain architectural features that are required for benchmarking some of the work primitives we identified. In particular, Linda does not allow the identification of specific nodes for running **eval()**s. This increased the difficulty of measuring features like latency and contention.

The choice of software architecture is also very important. It should not be biased towards any particular architecture. We chose Linda as our software architecture because it is a fairly simple, low level paradigm for parallel programming and it is easily portable across most available hardware architectures.

The identification of work primitives is crucial to a synthetic benchmark. In the ideal case the set of work primitives partitions the typical workload. We found this particularly hard to achieve. Firstly, the granularity of any particular work primitive is restricted by the constructs of the language or software architecture (Linda). Secondly, it is difficult to avoid overlaps between the work primitives. This can skew the results by allowing some characteristics to be measured more often than others.

There are also difficult issues associated with assigning weights to the results of the benchmarks in order to produce a single figure for comparison. Firstly, there is the problem of combining the results of benchmarks which generate measurements in incomparable units. This is a major difficulty in incorporating the benchmark for scalability into the overall result. Secondly, the measurements produced by comparable benchmarks must be weighted according to the frequency of occurrence of the associated work primitives in a *typical* work load. For some primitives this requires a complicated analysis of different application programs. In the absence of such analysis we were forced to make estimates of the appropriate weighting factor for some of the benchmarks. We believe that the problems associated with assigning appropriate weights to the various benchmark results probably outweigh the benefits of having a single overall performance figure for comparison. The real benefit of running the BeLinda benchmark set is that it produces a database of results which can be used to identify specific problems and areas for improvement.

5. Related Work

Work reported in the area of performance evaluation of parallel machines can be divided into three general categories. The first approach is to benchmarked different versions of the same architectural model: Bomans and Roose [11] compare different versions of the Intel iPSC; Kolawa and Otto [12] compare the Mark II with the Intel Hypercube; and Grunwald and Reed [13] compare the NCube and the iPSC. The second approach is to do benchmarking by timing of application programs. This approach has been followed by Martin [14] to compare different supercomputers, and by Fraboul [15] and Gustafson and Hawkinson [16] to benchmark different parallel machines. The third approach to benchmarking is to use analytical tools. This approach is taken by Flatt and Kennedy [17]. We are not aware of any other work that uses a portable software architecture such as Linda tuple space to benchmark different parallel architectures.

Work has also been reported in the area of instrumenting parallel programs [21,22]. Miller and Yang [23] discuss interactive performance tools, while, Guarna *et al.* [24] describe an integrated environment for the development of parallel programs. Such tools are of considerable benefit in building benchmarks.

6. Conclusions

In this paper we have introduced a new approach to benchmarking parallel computing platforms based on the use of a suite of Linda programs. Our experiences with designing the BeLinda benchmark and using it on three radically different architectures have been mixed. We are encouraged by the speed and efficiency with which the benchmark can be implemented and run on new architectures. This factor is particularly important given the absence of any other readily available, portable benchmarks for measuring the performance of different parallel architectures.

An alternative which has comparable portability and ease of implementation to BeLinda is to time the execution of a specific application program on different architectures. This approach would solve many of the problems that we mentioned in section 4 concerning the

choice of work primitives and their associated weights. However, such an approach would not produce the detailed analysis of the strengths and weaknesses of the architectures in question that is available using BeLinda. A major conclusion of the paper therefore is that the BeLinda approach to benchmarking parallel architectures is useful, but that it is not a realistic idea to attempt to reduce the results of such a benchmark set to a single overall figure.

Acknowledgements

The paper was improved by comments from the anonymous referees. We would like to thank Tony Capitano of OACIS for allowing us to use their iPSC/2, and to the system staff at OGI who gave us an exclusive access to the Sequent Symmetry to run our benchmarks.

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. 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.
4. W. Crowther et al., "Performance Measurements on a 128 Node Butterfly Parallel Processor," *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 531-540, August 1985.
5. R. F. Rashid, "Designs for Parallel Architectures," *UNIX Review*, April 87.
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. 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.
10. R. Bjornson, N. Carriero, D. Gelernter, and Jerrold Leichter, "Linda, the Portable Parallel," Yale Univ. Dept. Comp. Sci. RR-520, January 1988.
11. L. Bomans and D. Roose, "Benchmarking the iPSC/2 hypercube multiprocessor," *Concurrency: Practice and Experience*, vol. 1, no. 1, pp. 3-18, September 1989.
12. A. Kolawa and S. W. Otto, "Performance of the Mark II and Intel Hypercube," Technical Report 254, Caltech Concurrent Computation Program, February 86.
13. D. C. Grunwald and D. A. Reed, "Benchmarking Hypercube hardware and software," Technical Report, University Of Illinois, November 86.
14. 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.
15. C. Fraboul, "MIMD parallelism expression, exploitation and evaluation," in *Supercomputing*, ed. A. Lichnewsky and C. Saguez, pp. 155-170, Elsevier Science, 1987.
16. J. L. Gustafson and S. Hawkinson, "A Language-Independent Set of Benchmarks for Parallel Procesors," Technical Report, Floating Point Systems, April 1986.
17. H. Flatt and K. Kennedy, "Performance of parallel processors," *Parallel Computing*, vol. 1, no. 12, pp. 1-20, 1989.
18. S. Kambhatla, J. Inouye, and J. Walpole, "Benchmarking Parallel Machines via a Software Architecture," Technical Report 90-002, Oregon Graduate Institute , 1990.
19. W. Leler, "Linda meets UNIX," *IEEE Computer*, pp. 43-55, February 1990.
20. Cogent Research Inc., "Kernal-Linda Specification 4.0," Technical Report, March 1989.
21. Y. Gaur, V. A. Guarna, Jr., and D. Jablonowski, "An Environment for performance experimentation on multiprocessors," CSRD Rpt No 865, University of Illinois, April 89.
22. Z. Segall and L. Rudolph, "Pie: a programming and instrumentation environment for parallel processing," *IEEE Software*, pp. 22-37, November 1985.
23. B. P. Miller and C. Q. Yang, "Ips: an interactive and automatic tool for parallel and distributed programs," *Proceedings of the seventh International Conference on Distributed Computing Systems*, pp. 482-489, September 1987.
24. V. A. Guarna Jr., D. Gannon, D. Jablonowski, A. D. Malony, and Y. Gaur, "Faust: an integrated environment for development of parallel programs," *IEEE Software*, pp. 20-28, July 89.

Appendix

Table 1. Time for primitives

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

Table 2. Time to **rd()** actuals

System	Actuals(rd())				
	1	2	4	8	16
Sequent	14us	14us	14us	14us	--
iPSC2	1046us	1038us	1048us	1038us	--
XTM	437us	808us	1551us	--	--

Table 3. Time to **in()** actuals

System	Actuals(in())				
	1	2	4	8	16
Sequent(8)	14us	14us	14us	14us	15us
iPSC2(16)	1047us	1039us	1050us	1038us	--
XTM(8)	467us	838us	1581us	--	--

Table 4. Latency

System	Latency
Sequent	0.0455ms
iPSC2	1.051ms
XTM	4.217ms

Table 5. Contention

System	Contention
Sequent	0.044ms
iPSC2	1.575ms
XTM	4.248ms

Table 6. Multicast

System	Multicast			
	1-2	1-4	1-8	1-16
Sequent	0.10ms	0.60ms	2.0ms	4.3ms
iPSC2	1.0ms	2.9ms	6.8ms	14.0ms
XTM	4.05ms	7.46ms	15.1ms	59.2ms

Table 7. Reverse multicast

System	Reverse Multicast			
	2-1	4-1	8-1	16-1
Sequent	0.30ms	1.3ms	3.1ms	6.7ms
iPSC2	2.0ms	5.2ms	11.6ms	23.8ms
XTM	9.67ms	14.9ms	44.7ms	93.1ms

Table 8. Interference time

System	Comn	Comn & comptn	comptn	comptn & comn
Sequent	0.27	0.27	0.22	0.22s
iPSC2	21.134	21.244	0.752	1.815s
XTM	4.67	4.94	0.0874	0.124s

Table 9. Time for computation in the presence of blocked workers

System	10 workers	10 wrkrs + 10 blkd
Sequent	400ms	1100ms
iPSC2	307us	581us
XTM	583ms	891ms

Appendix

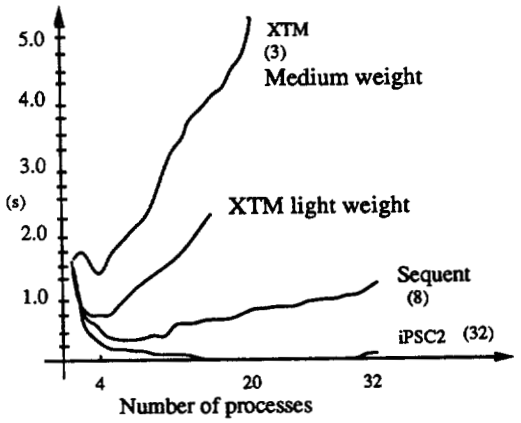

Fig 1. Scaling effect

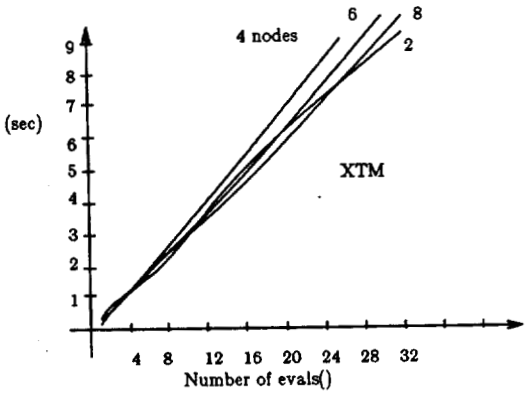

Fig 2. Cost of computation

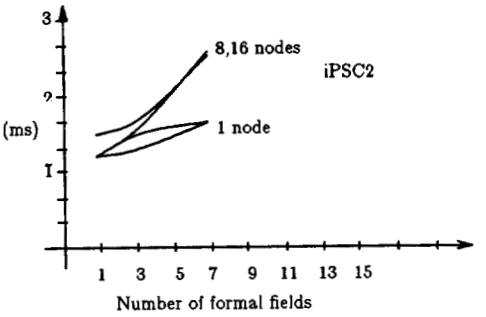

Fig 3. Time for formals

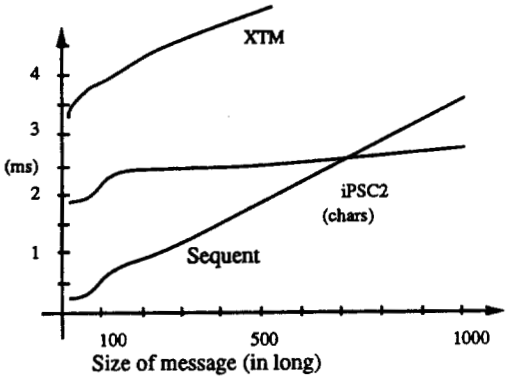

Fig 4. Transmission time

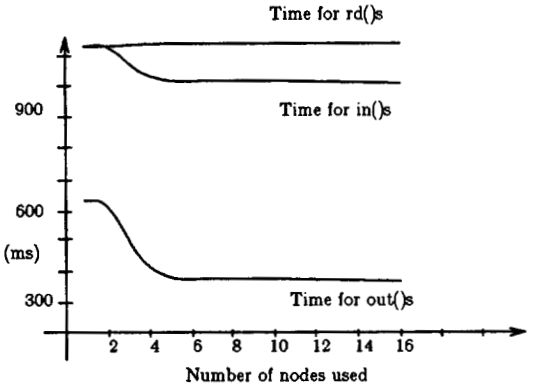

Fig 5. time for primitives (iPSC2)

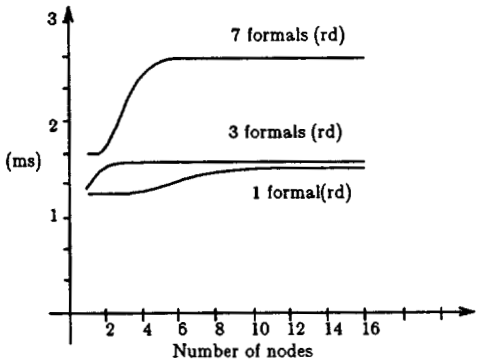

Fig 6. formals (iPSC2)