Dynamic Optimizations in Linda Systems*

Stefanos Kaxiras, Ioannis Schoinas
kaxiras@cs.wisc.edu, schoinas@cs.wisc.edu
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA

January 15, 1993

Abstract

We examine schemes for optimizing a Linda implementation in a message passing environment. We show that, with these schemes, it is possible to:

1. Capitalize on the locality of operations of Linda programs by using a distributed tuple space.
2. Increase such locality with replication of tuples.
3. Take advantage of the persistent communication paths, that are present in Linda programs, by dynamically routing templates and tuples.

We present the level of our success and the constrains involved in optimizing Linda in these three directions.

Keywords: Parallel computing, Linda, Distributed memory, Message passing, Locality.

*The Thinking Machines CM-5, of the Computer Sciences Department of the University of Wisconsin-Madison, was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. It is running the pre-release (beta) system software CMOST 7.2 S4, which may or may not reflect the performance of the final software.

1 Introduction

Linda [1] is an interesting model for the development of parallel applications. Its strength is centered in the simple, yet powerful, way it handles the communication and synchronization of parallel processes. Unfortunately due to its generality it is difficult to build efficient implementations. We have observed a certain behavior of Linda programs, that has to do with the locality of operations and the persistence of communication paths. We set out to exploit this behavior by providing a Linda implementation with optimizations based on this observed, specific, behavior. Furthermore, we implemented Linda using the cooperative shared-memory approach [2] and we present our observations.

In section 2 and 3 we briefly describe the Linda model and the underlying environment for the implementations. In section 4 we present the basic schemes for the Linda implementation in the message passing environment. In section 5 we describe the optimizations we examined for the message passing implementation. Section 6 presents the performance of five Linda applications for the different schemes. Section 7 discusses the ramifications of a Linda implementation using the cooperative shared-memory approach. Finally section 8 gives the conclusions of this work.

2 The Linda Model

Linda [1, 3] is a parallel programming model for creation and coordination of multiple processes that run in one or more processors. The Linda model is embedded in a computation language (C, Lisp, etc.) and the result is a parallel programming language.

Tuple space is Linda's mechanism for creating and coordinating multiple execution threads. The tuple space is a bag of tuples. A tuple is simply a sequence of values corresponding to typed

fields. Linda provides operators for dropping tuples into the bag, removing tuples out of the bag and reading them without removing them. Associative search is used to find tuples in the tuple space. Templates, comprising of values of a subset of the fields of a tuple, are used to select tuples for removal or reading. The Linda model defines four operations on the tuple space. These are:

- **out(t):** it causes tuple t to be added into the tuple space.
- **in(s):** it causes an arbitrary tuple t that matches the template s to be withdrawn from the tuple space. If such tuple does not exist, the call blocks.
- **rd(s):** it is the same as $in(s)$ expect that the matching tuple is not withdrawn from the tuple space.
- **eval(t):** it causes a process to be created to evaluate the fields of the tuple t. When the evaluation ends the tuple t is put in the tuple space. Since the native environment already offers process creation, this operation was not implemented.

3 Underlying Environment

For this work we used the CM-5 [4] parallel computer of the University of Wisconsin. The CM-5 is a distributed memory MIMD machine. Its processing nodes, each with its own memory, are connected with three fat tree networks. Besides the two normal message passing networks there is also a control network that supports operations like broadcast, global OR, reduction, parallel prefix and suffix. The cost of these operations is considerably higher than the cost of point to point messages. A host processor, which has access to the CM-5 networks, is responsible for coordinating the work done in the nodes.

On top of the CM-5 hardware, we designed and implemented an asynchronous message passing protocol. This protocol allowed us to adopt a remote procedure call convention in designing and building an implementation of Linda. With this protocol, each Linda operation is defined as a set of actions that must be performed locally and/or remotely. Since there cannot be two broadcast operations from two different nodes in progress at the same time, the protocol in each node forwards broadcast messages to the host processor. The host processor upon receiving a broadcast message performs the broadcast.

4 Basic Schemes

For a Linda implementation we need to define how to do the associative search. The common practice is to use hash tables to store the tuples and the templates. The key to access the hash tables is constructed from the the types of the fields that comprise a tuple (template) and the value of its first field. It falls to the programmer to supply an appropriate value for the first field, so that the accesses to the hash tables are efficient.

In a distributed memory system with message passing we need to define:

- where to store a tuple initially,
- where to send a template, so it can be checked against all the existing tuples that can potentially match it,
- how to guarantee that an unsatisfied template will be checked against all new tuples that can potentially match it.

The answers to the above questions differentiate the schemes presented below.

4.1 Distributed Hashing

Traditionally, the Linda implementations in distributed memory machines (IPCS/2, transputer based machines) [5, 6] have adopted the *Distributed Hashing (DH)* approach. In this scheme, there is only one, global, hash table, distributed over the nodes. The node in which each tuple will be stored is determined by applying a hash function on the key of the tuple. An *out*() operation sends the tuple to the node determined by the hash function. Similarly, *in*() and *rd*() operations send their template to the appropriate node. For a template to match a tuple, it is necessary both to have identical keys. The template will be sent to the node where all the tuples that can potentially match it will also be sent. If a template cannot be satisfied immediately, it is stored in that node and checked against all incoming tuples.

The advantage of this scheme is that the Linda operations have a constant cost. This can also be a disadvantage since the locality of Linda programs is not exploited. In addition, it is prone to contention at some nodes if the hash function fails to distribute the tuples evenly or if some tuples are accessed more frequently than others. A special case of the DH approach, is the centralized tuple manager approach, where all tuples are hashed to the same node.

4.2 Distributed Tuple Space

Another general scheme for implementing Linda in distributed memory machines is based on a *Distributed Tuple Space (DTS)*. In the CM-5, each processing node has a part of the tuple space in its local memory. Each node has its own local hash table. All *out*() operations in a node result in a tuple being inserted in the local hash table. All *rd*() and *in*() operations, that do not match a tuple in the local hash table, broadcast the template to all other nodes. When a node receives a template from another node and has a matching tuple in its local hash table, it moves this tuple to

the requesting node. Every node stores the templates that it cannot satisfy. Whenever it creates a tuple, it checks it against the stored templates.

In the DTS scheme, the tuples move from node to node in response to requests. Local operations involve no communications but remote operations require costly broadcasts. The cost of Linda operations is not always predictable. For example, an $in()$ or $rd()$ operation may result in the movement of many tuples to the requesting node.

4.3 Read Replication

The first step to exploit locality of operations is with replication of tuples for $rd()$ operations. We call this *Read Replication (RR)*. We can expand both DH and DTS schemes with RR. In Linda, the life of a tuple can be described with the sequence of operations $out() - \{rd()\}^* - in()$. RR improves the efficiency of repeated $rd()$ operations on the same tuple by more than one node. The basic idea of the scheme is the existence of a node which is declared as the tuple owner. The tuple owner is responsible for spreading and invalidating copies of the tuple. When a node issues a $rd()$ operation, it receives a copy from the tuple owner. The copy is stored locally in the node and can satisfy subsequent $rd()$ operations of this node without communication.

The implementation of the scheme resembles the $Dir_K B$ [7] coherence protocol: With the first $rd()$ request a node with a matching tuple becomes its owner. It satisfies the first k $rd()$ operations by giving out copies. With the $(k+1)th$ $rd()$ operation, it broadcasts the tuple to every other node. The copies are only visible to $rd()$ operations and not to $in()$ operations. When the owner of a tuple receives a matching template of an $in()$ operation, it invalidates all copies of the tuple.

In DH, $rd()$ operations satisfied locally with a copy, save two messages: one to move the template to the node where the tuple resides, and one to move the tuple back. The gain in the DTS scheme

from $rd()$ operations satisfied locally with copies, is considerable since the broadcast of the template and the message for the tuple are saved. For both DH and DTS schemes there is a slight cost when using RR. This cost is the invalidation of the copies which requires up to k messages or a broadcast.

5 Routing Schemes

We chose to expand the DTS scheme to exploit locality of operations and to take advantage of the persistent communication paths in the Linda programs. DTS has an advantage over DH, in that it only moves tuples upon receiving a request. In addition, in DTS, it is easier to dynamically decide how and where to move templates and tuples to improve performance.

The schemes presented in this section are based on the observation that a Linda program exhibits persistent communication paths between its parallel processes. According to Carriero and Gelernter [8], a Linda program can follow either the message passing paradigm or the distributed data structures paradigm. If the application follows the message passing paradigm, the paths are explicit and known to the application developer. If the application follows the distributed data structures paradigm, persistent communication paths may also be formed due to the locality of accesses to the abstract data objects represented by tuples.

In Linda, the need to efficiently access the tuples and the templates, in the hash tables, is in conflict with the locality of the accesses on the abstract objects, represented by the tuples. For example, tuples that represent rows of a matrix need to have different keys for good hashing. If that is the case, we have no information from the keys that the tuples represent the same abstract object (matrix). Such information should be present in at least one field of the tuples, exactly because they represent the same abstract object.

To address this problem, we define a secondary key that is constructed from the the types of the fields that comprise a tuple (template) and the value of its *second* field. For programs following the distributed data structures paradigm, the second field of the tuple (template) should be used to carry the information about the locality domain of the tuple. For programs following the message passing paradigm, the value of the second field should denote a message path. In the following sections, we call templates that have identical secondary keys *similar*.

5.1 Template Routing

The first mechanism to take advantage of the persistent communication paths in Linda programs, is *template routing (TR)*. Simply put, templates are dynamically routed to nodes where it is expected to find the appropriate tuples, according to information of the recent past. The mechanism of TR is quite simple: When an $in()$ or $rd()$ operation is not satisfied in the local tuple space, its template must be broadcasted to all other nodes. In order to avoid this expensive broadcast, the node requests the tuple from a node that has previously satisfied a similar template. Since there may be many other nodes that have satisfied a similar template in the past, we request the tuple from the node that has done so most often.

This mechanism is implemented using a cache in each node. In each cache entry, we store a template and information about the number of times each node has satisfied a similar template. This cache is accessed via the secondary keys, so with similar templates we access the same entry. If an $in()$ or $rd()$ cannot be satisfied locally, the cache is accessed. Then, instead of being broadcasted, the template is directed to the node that has satisfied a similar template, most often. If the node that receives this routed template has a matching tuple, it returns it immediately; otherwise, it broadcasts the template. Finally, when a tuple returns to the requesting node the cache is

updated. If the routing was unsuccessful, i.e., the tuple was not returned from the expected node, we penalize this node in the cache entry. Specifically, the number of times the node has satisfied a similar template is halved. This rather severe penalty is used for quick adaptation to changing behavior.

Unsuccessful routings result in one more message in addition to the broadcast. Without taking into account the overhead of maintaining the cache, a simple calculation shows that TR is successful if the hit ratio *(successful routings / all routings)* is greater than the ratio *(cost of message / cost of broadcast)*. In the CM-5, the ratio *(cost of message / cost of broadcast)* is very low. Thus, the hit ratio required for TR to be effective is very low. As we shall see later on, TR proved to be a success for most Linda programs.

5.2 Tuple Routing

Since we have a mechanism to direct templates, we moved a step further and tried to direct tuples to nodes where they might be needed in the near future. We call this mechanism *Tuple Routing (TuR)*. It can be considered as a form of 'pre-sending'. Nodes that have abundant tuples, distribute them to nodes that have requested similar tuples in the past, in anticipation of new requests. Similar tuples are the tuples that are matched by similar templates.

This mechanism is also implemented using a cache in each node. This cache, similar to the one used in TR, is accessed with the secondary keys. All similar templates access the same cache entry. In each cache entry, we store a template, a counter for every other node, and other bookkeeping information. The counter for a remote node represents the number of tuples that have been sent to that node as a result of its previous requests. All these tuples were matched by templates similar to the one in the cache entry.

If the node detects a steady state in communications, it starts transferring the tuples from its local tuple space to other nodes. We consider a steady state to be the case where more than two requests from a remote node have been satisfied with similar tuples. Each time a node receives a template which it can satisfy, it accesses its cache. The counter for the node, where the template originated, is incremented and a check is made for steady state. If there is a steady state, the node starts a transfer cycle. The number of tuples that will be pre-sent to each node is proportional to the number of previously satisfied requests for that node.

Since feedback comes only in the form of new requests, the transferring of tuples cannot continue indefinitely. New requests come only from the nodes that 'consume' the tuples they get and need more. Nodes that stay behind in the 'consumption' will accumulate tuples in their local tuple space, while there is a need elsewhere. The mechanism of TuR will increase the proportion of routed tuples to nodes that generate new requests. When a node pre-sends a certain amount of tuples, it exits the transfer cycle, i.e., it stops and clears the cache.

TuR saves the request template from being sent or broadcasted. It can reduce the latency of an $in()$ or $rd()$ operation by moving a tuple to a node before that node requests it. The problem with TuR is that it worsens the ping-pong effect. The ping-pong effect, an inherent deficiency of the DTS scheme, is what we call the excessive movement of tuples from node to node due to the unnecessary satisfaction of outdated templates. In DTS, a node stores the unsatisfied template of a remote node until it can satisfy it or a new one arrives from that remote node.

The following situation illustrates the problem. Two or more nodes broadcast templates that can be satisfied by the same kind of tuples. If one of them receives many tuples, it will use one and send the rest to the other nodes. In the worst case, the tuples will continue to travel from node to node until all the stored templates have been discarded (satisfied) throughout the system. The

ping-pong effect can be eliminated either by allowing only one answer as a result of a request or by not broadcasting the request. In DTS, for the phenomenon to occur, there must be more than one node having the appropriate tuples to satisfy the requests. TuR intensifies the ping pong effect by also allowing one node to send more than one tuples as a result of a request.

6 Application Results

To evaluate the success of the schemes presented, we use five Linda applications. With the first application, we establish that RR is essential for Linda programs with mostly read tuples. With the next four, we assert the effectiveness of the routing schemes.

6.1 Read Replication

The matrix multiplication program is one of the classical Linda demonstration programs. It adopts the master/workers paradigm. The master outputs tuples that contain the rows and columns of two matrices respectively. The workers request a job tuple, read a row of the first matrix and the columns of the second matrix, perform the multiplication and create a result row tuple. This process continues until all the rows of the result matrix have been computed. Since the second matrix must be read once for each row calculated, it becomes obvious why the RR is an essential feature of every approach. In table 6.1, it is shown that the addition of RR in DTS or DH results in reducing the number of messages by an order of magnitude. For DTS it practically eliminates broadcasts.

	DTS	DTS+RR	DH	DH+RR
msgs	235995	13184	201663	13413
bcasts	102158	386	0	320

Table 6.1: Matrix Multiplication: 320x320 matrices, 32 CM-5 nodes

6.2 Routing Schemes

The following variations of DTS and DH are compared: DTS with RR, DTS with RR and TR, DTS with RR and TuR, DTS with RR and both TR and TuR, DH with RR. For each of the programs and for every variation the total number of broadcasts, the total number of messages and the speedup are given. Each program was run on 1,2,4,8,16 and 32 CM-5 nodes. While the metrics presented do not capture the whole picture, lack of space forbids including everything that was measured. Furthermore, it should be noted that no effort was made to restructure the programs in order to improve their performance and speedup. In fact, the majority was designed to run in a LAN of few workstations.

6.2.1 Newspaper Image Block Classifier

This program, described in [9], takes as input a newspaper image and classifies the blocks it contains. A block can be large text, small text, graphics, image, e.t.c. We can distinguish two phases in this program. In the first phase, the input image is statically divided among the workers, certain image transformations are applied to each part and then the workers exchange the parts. In the second phase, using a divide & conquer strategy, the image is subdivided into blocks, certain attributes are calculated for each block and finally these attributes are used to classify the block. In this phase, each worker gets a job description tuple and it proceeds to examine if the area it describes can be further partitioned. If so, it generates more job description tuples. The process continues until all the areas have been processed and classified.

The results for processing an image of 2384x2517 pixels with 39 blocks are depicted in figures 1-3. The parallelism of this program is limited by the small number of blocks in the image, hence the small speedup. All the DTS based schemes perform roughly the same. The reduction in broadcasts achieved with TR is negligible and negated by an increase in the number of messages. While DH produces less messages than DTS with TR or DTS with both TR and TuR, its overhead is higher. The reason is that the extra messages in the other schemes are small templates, while in DH most of the messages are due to the movement of the bigger tuples. Especially in the second phase, there is little communication in all the DTS based schemes because the workers request job description tuples from remote nodes, only after they consume all the tuples they created themselves.

6.2.2 Differential Equation Solver

In this program, described in [10], a differential equation is solved recursively for a given input of timesteps. The matrix of the problem is statically partitioned among the workers. In each timestep, each worker must exchange the boundary areas with its neighbors. The communication is highly localized between the neighbors in the matrix.

The results from solving the equation for a matrix of 3800x100 and 500 timesteps are depicted in figures 4-6. It is apparent that TuR is marginally effective in reducing broadcasts and messages. TR reduces the broadcasts considerably, roughly to one third of DTS. In DH, we observe the most messages because of the indirection involved in almost all movements of data. In the speedup curves we see that there is a breakdown for 32 nodes of all schemes except DTS+RR+TR+TuR. For the DTS variants this happens because of the excessive communication. DTS+RR+TR+TuR is saved because of the low number of broadcasts. We have discovered that DH breaks down because the hash function fails to distribute the tuples evenly. Roughly one third of all tuples are stored in one

node.

6.2.3 Prime Number Generator.

This is a classical Linda program, described in [8]. It adopts the master/workers paradigm. There is a single tuple which every worker withdraws, updates and reinstalls in the tuple space. This tuple is used to claim a subrange in which the worker searches for prime numbers. The worker reports the primes it finds back to the master. The master announces them to all the workers that need them to search within their subrange. The workers use rd() operations to get these primes. Whenever a worker finishes with its subrange, it gets a new subrange. An important point is that the master requests the primes in ascending order.

The results from finding all the primes in the range 1 to $3 * 2^{20}$ and a subrange size of 3072 are depicted in figures 7-9. TuR reduces the number of broadcasts and messages. The main reason is that the workers pre-send the primes before the master requests them. About one fourth of the broadcasts are attributed to RR. The success of TR is small. It is interesting to note, in figures 7 and 9, the additive property of the improvements by TR and TuR to DTS. The reason for the failure of DH is due to the contention on the node where the tuple used to assign ranges to workers is stored.

6.2.4 Mandelbrot Image Browser.

This program is a Linda port of MIT's 'xmandel' program. It creates images of the mandelbrot set. It adopts the master/workers paradigm and exhibits very simple communication patterns. The master creates job tuples that contain the number of the row in the mandelbrot image. Then it waits for result tuples. Every worker gets a job tuple, computes the row and creates a result tuple.

The workers continue until all the job tuples have been consumed.

The results from computing the mandelbrot image of 300x300 pixels four successive times are depicted in figures 11-12. This application fits perfectly with DTS. The number of messages exchanged, using this scheme, is very close to the theoretical minimum for the task (2400 messages: one to assign the work and one to get the result, for every row). While TR is marginally effective in reducing the broadcasts, it does not impose any severe overhead. TuR fails completely because of the ping-pong effect. In fact, more than half of the total number of tuples reaching a node were immediately forwarded to another node. DH performs badly and the indirection involved in the data movement is to blame for this.

6.2.5 In Short

From the results presented here, we can conclude that DTS outperforms DH. It fails only when the number of broadcasts becomes very large. TR generally enhances the performance of DTS. Even when it fails, the extra overhead is relatively small. TuR can reduce the number of messages exchanged in the system, but it may also lead to excessive network traffic in other cases. When both TR and TuR perform reasonably the same applies to their combination.

7 Discussion: Linda & Shared Memory

Generally, it is difficult to reason about the movements of the data within the shared memory hardware with most shared memory architectures. The ability to make such reasoning is important, in order to use the specific knowledge about the access patterns on the data for code optimization. Fortunately, the cooperative approach to shared memory [2] gives exactly this ability. In our perspective, the key point of the approach is that it defines advisory hardware primitives that

Figure 1: Newspaper - Broadcasts

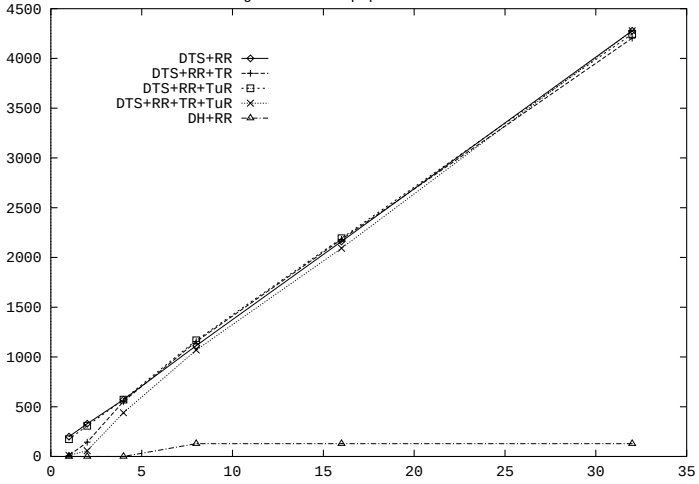


Figure 2: Newspaper - Messages

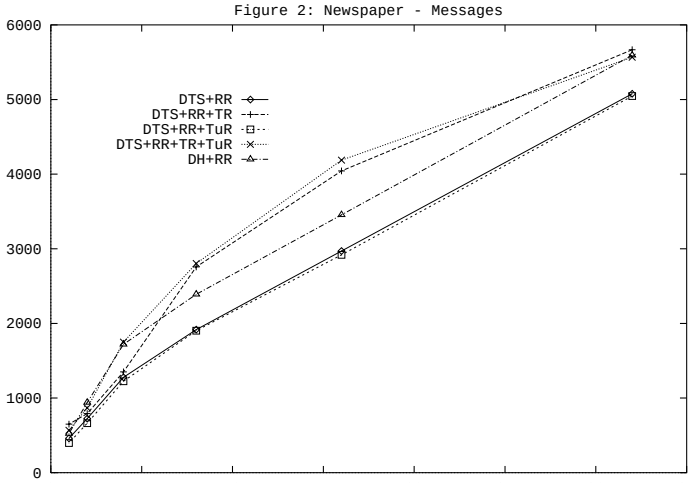


Figure 3: Newspaper - Speedups

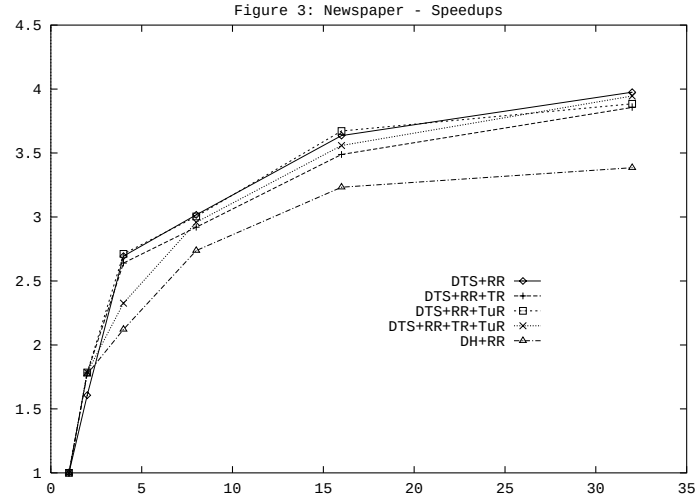


Figure 4: Differencial Equation Solver - Broadcasts

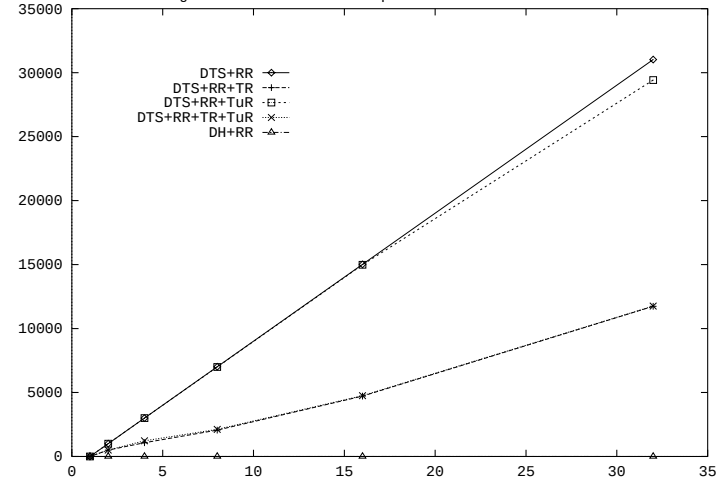


Figure 5: Differencial Equation Solver - Messages

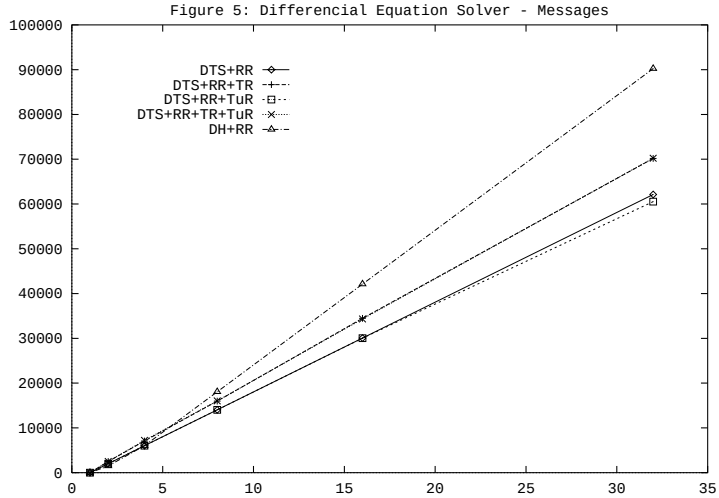


Figure 6: Differencial Equation Solver - Speedup

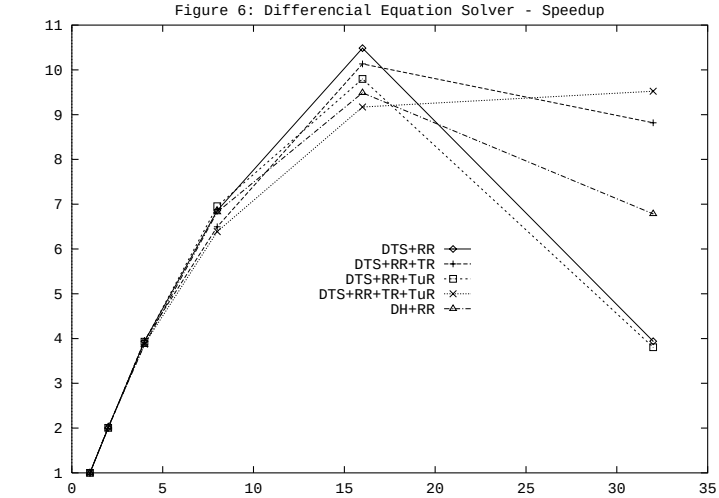


Figure 7: Prime Number Generator - Broadcasts

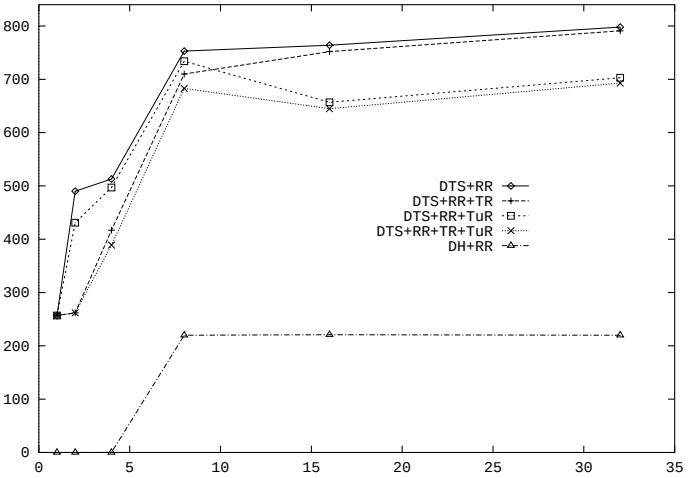


Figure 8: Prime Number Generator - Messages

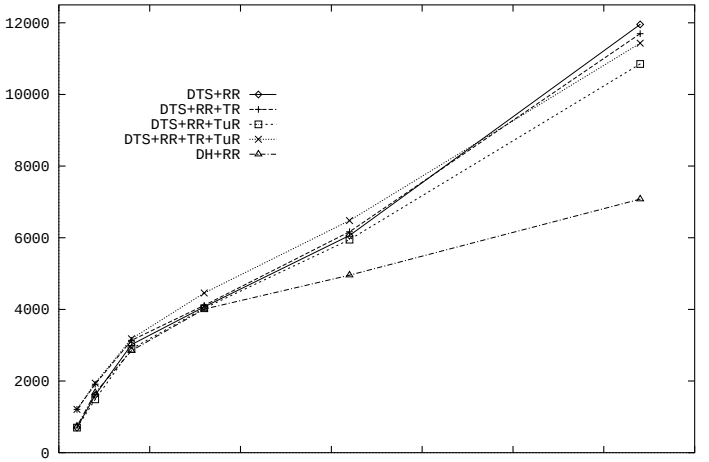


Figure 9: Prime Number Generator - Speedup

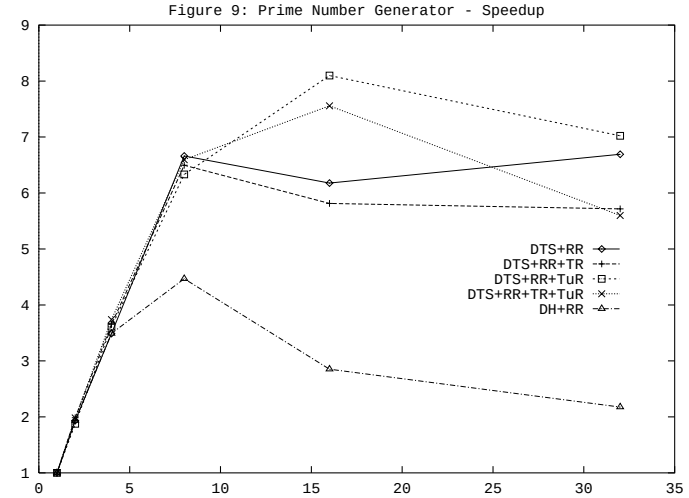


Figure 10: Mandelbrot Browser - Broadcasts

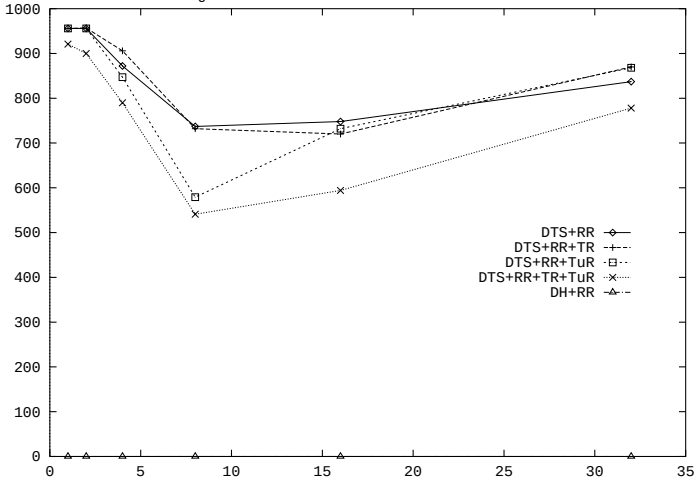


Figure 11: Mandelbrot Browser - Messages

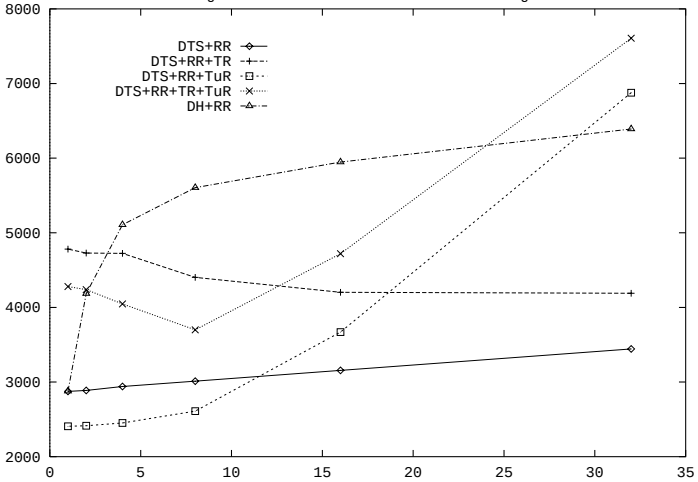
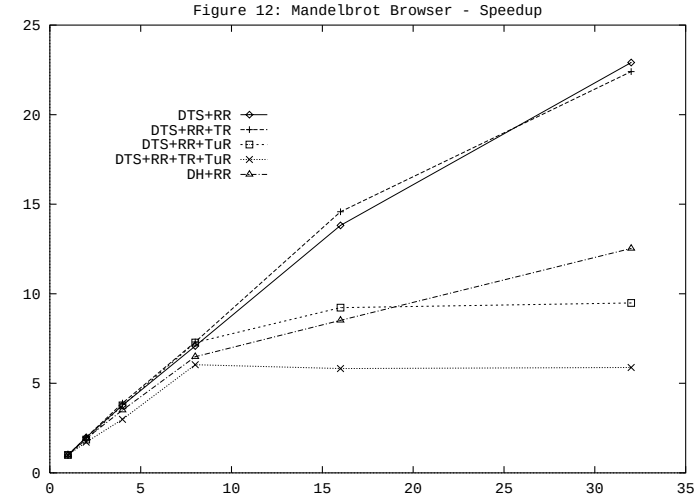


Figure 12: Mandelbrot Browser - Speedup



announce to the hardware the intention to access a specific cache block or the completion of the access. In effect, they can be used to control the data movement within the shared memory hardware.

The cooperative shared memory approach has been realized within the Wisconsin Wind Tunnel simulator (WWT) [11]. WWT is a parallel discrete event simulator that is currently running on CM-5. It cleverly uses the memory hardware of the processing nodes to simulate the parallel machine to hardware speed in the case of cache hits. We used the cooperative shared memory simulator to implement a shared-memory Linda. The goal was to show in practice that you can reason about the data movements.

In the implementation, there are two main shared data structures, both with a similar layout. The first one is the *Tuple Space (TS)* where the tuples are stored. The second one is the *Template Buffer (TB)* where the unsatisfied templates are stored. They are implemented as a linked list hash table, i.e., the entries of the table are pointers to a null-terminated linked list. The indexing on the hash table is done using the key of a tuple or a template.

Since these hash tables are shared data structures, there must be a synchronization policy defined for accessing them. The policy for TS is 'multiple readers/single writer' while the policy for TB is 'exclusive access'. The synchronization is done independently for each linked list.

By using the CICO primitives, when a processor accesses the elements of a linked list, it brings the relevant memory blocks in its cache. Respectively, when it no longer has any use for these blocks, it flushes them out.

Conceptually, the shared-memory Linda resembles the DH scheme of the message-passing Linda. The tuples data will be distributed throughout the memory modules. However, there is an important fundamental difference. While in the message-passing Linda we send the query (template) to

the data (tuples), in the shared-memory implementation we send the data to the query. Currently, we are examining the effect of this difference in the design of an efficient Linda implementation for shared memory. In addition, we are investigating the applicability of techniques similar to the message-passing routing schemes.

8 Conclusions

In this work we have shown that it is possible to design and implement a Linda System that adapts its behavior to the actual information paths that are present in a Linda program. We have chosen DTS as the basis of our design. RR proved to be essential for an efficient design. TR can, in most of the cases, reduce the number of broadcasts in the system. TuR can reduce the number of messages in the system, but the existence of the ping-pong effect makes it unstable under certain conditions. The schemes examined complement DTS in a natural way, making it preferable to DH.

9 Acknowledgments

We are indebted to Mark D. Hill for his guidance throughout this work. We are grateful to Miron Livny and Marvin Solomon for discussions and suggestions. We also appreciate comments on an early draft of the paper made by Yannis Ioannidis. Finally, we would like to thank those who developed the Linda applications we used in this work: Clemens Cap & Volker Strumpen for the differential equation solver program, Ioannis Kavaklis for the newspaper program and the Linda port of xmandel, David Gelernter & Nicholas Carriero for designing the prime program.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer*, 18:26–34, August 1986.

[2] M. Hill, J. Larus, S. Reinhart, and D. Wood. Cooperative Shared Memory: Software and hardware for scalable multiprocessors. Technical report, CSD, UW-Madison, March 1992.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32, Number 4:444–558, April 1989.

[4] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[5] Leichter J. Shared tuple memories, shared memories, buses and lan's - Linda implementations across the spectrum of connectivity. Technical Report 714, DCS, Yale University, July 1989.

[6] S. Zenith. Linda coordination language; subsystem kernel architecture (on transputers). Technical Report 794, DCS, Yale University, May 1990.

[7] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM transactions on Computer Systems*, November 1986.

[8] N. Carriero and D. Gelernter. How to write parallel programs:A guide to the perplexed. *ACM Computing Surveys*, 21:323–357, September 1989.

[9] I. Kavaklis. Classification of Newspaper Image Blocks Using Texture Analysis, July 1991. Project Report, CSD, University of Crete.

[10] C. Cap and V. Strumpen. The PARFORM: A High Performance Platform for Parallel Computing in a Distributed Environment, June 1992. Institut fur Informatik, Universitat Zurich, draft paper.

[11] S. Reinhart, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Ccomputers. Technical report, CSD, UW-Madison, September 1992.