# ParLin: From a Centralized Tuple Space to Adaptive Hashing

João Gabriel Silva, João Carreira, Francisco Moreira

Universidade de Coimbra
Urb. Boavista, Lt.1-1
3000 Coimbra - Portugal
Email:{ jcar, fmoreira, jgabriel}@pandora.uc.pt

**Abstract.** In this paper we present a library for the PARIX[1] Operating System, called ParLin, that provides support for Linda-like programming in transputer arrays. The primitives offered by this library are quite efficient, due to some design decisions, explained in the paper, that lead to a very simple internal structure, with a centralized Tuple Space. We claim that this is the best solution for small to medium systems (up to several tens of processors) and present the results of some experiments that support this statement.

The mechanisms needed to scale up this library to larger systems are then discussed, with a proposal being made for a new scalable technique, called adaptive hashing. It consists of a partitioned (not replicated) Tuple Space, where the tuples are localized with the help of an hashing function that uses as input not only some fields of the tuples, but also the topology and size of the underlying system, and information about the distribution of tuples. This distribution is either gathered during run-time, or given by the programmer, or both. Since during successive runs the system can perfect it's knowledge of the tuple's behaviour, changing the hash function accordingly, we call it adaptive hashing.

**Keywords:** ParLin, Linda, Tuple Space, Transputers, Parix O.S, Adaptive Hashing.

## 1. Introduction

Parallel programming languages can be divided in two main classes: those languages where the communication between processes is done through shared-memory and those based on message passing. The programs written with them tend to be bound to shared-memory or distributed-memory architectures, respectively.

Linda[10] tries to overcome these restrictions by providing a high-level abstraction for the programmer, called the Tuple Space (TS). The Tuple Space is a virtual associative memory, shared by all processes joined in the computation.

[1] PARIX is a registered trademark of Parsytec Computer Gmbh, Aachen, Germany.

Linda was implemented in shared memory machines like the Encore, Sequent and Alliant machines as well as in distributed memory machines, like the Intel iPSC hypercube, Bell Labs S/Net [8], and more loosely coupled systems, e.g., VAX/VMS workstation clusters [31], PC networks [12] and Sun Workstations [24].

It provides a collection of primitives for process creation and interprocess communication that can be added to existing sequential languages.

Linda is nowadays a promising parallel language mainly due to its simplicity and portability. Many algorithms can be easily expressed in Linda [10].

The transputer is a common building block for distributed memory architectures, frequently programmed in Occam, a programming model that closely matches the hardware architecture. Although problems can be efficiently solved in Occam, the proposal of a new higher level programming paradigm that uncouples the algorithm design from the hardware is welcome.

With the increasing use of transputers, soon proposals were made to port Linda to these systems. In fact, some effort was made in this area [4][13][26][28]. Several approaches were suggested to implement the Tuple Space over a distributed memory: centralized tuple space [28], distributed using hashing techniques [26] and distributed Tuple Space based on semi-replication and hashing [13].

One of the main concerns is the scalability of the tuple space, particularly in massively parallel systems. While the distribution of the tuple space is at a glance the solution to this problem, it is still not clear which is the best distribution policy: a certain distribution can fit well a particular application, but can be a bottleneck in another. This is an open point among Linda researchers.

With these considerations in mind, we developed the ParLin Library, with a clear design objective: providing primitives as efficient as possible for small to medium size transputer arrays (up to several tens of transputers). The scaling up of the library for bigger systems was left for a second phase. The technique we intend to use to attain that objective is presented in this paper. We call it adaptive hashing.

We start by briefly introducing Linda in Section 2. Our own Linda implementation is described with some detail in Section 3. It should be pointed out that ParLin is not a full-featured Linda implementation. The rationale behind our version is presented. In Section 4 we discuss how the latency of TS operations can be partially compensated, while in Section 5, we evaluate the performance of the library. In Section 6 the scalability of the Tuple Space is discussed, and our proposal, adaptive hashing, is presented. Section 7 draws some conclusions and finishes the paper.

## 2. Linda Overview

Linda is a system for building parallel programs based on the abstraction of the Tuple Space(TS): an associative (i.e. content addressable) unordered bag of *tuples.* A tuple is an ordered sequence of data elements. All the communication between processes is done through the Tuple Space using the following operations:

**out(t)**: Adds a tuple *t* to the TS.

**in(t)**: The tuple *t* , with some data elements not given, is used to search the TS for a tuple matching the given data elements. If no such tuple exists in the TS, the calling process is suspended until one is deposited. If several matching tuples exist one is chosen arbitrarily.

**rd(t)**: Same as in(t), but the matching tuple remains in the TS.

**inp(t)**, **rdp(t)**: Similar to in(t) and rd(t), except that these are non-blocking. If there is no matching tuple, a flag is returned giving feedback to the user.

**eval(t)**: Starts a different process to calculate the data elements of the tuple t, and then deposits the calculated tuple in the TS.

By incorporating these primitives in an high-level language like C [1], C++ [7] or Pascal [22], programmers can build parallel programs in a straightforward way.

Linda simplicity and elegance relies on some important factors, namely:

1. Programmers can be freed from having to deal with spatial and temporal relationships among parallel processes, because processes in Linda are spatially and temporally uncoupled. In fact processes do not need to be aware of each other nor even to be loaded at the same time.
2. Linda provides the ability to distribute processes dynamically at run-time.
3. We can use Linda to simulate shared memory, with the advantage of easily avoiding unwanted non-determinism, since individual tuples assure mutual exclusion. However, the use of Tuple Space as a shared data space is not limited to systems with physical shared memory; it can also be ported to distributed memory systems[2] .

Associated with Linda is a particular programming methodology based on distributed data structures, that is, structures that are directly accessible to many processes simultaneously [10].

The parallel programming paradigm best suited for Linda is probably the master-worker model. In this model, we replicate the program several times in the form of workers. Each worker grabs a job tuple from the Tuple Space, executes the required task, places a result tuple in TS and looks for another job. This paradigm has the advantages of being relatively easy to program, of balancing the load dynamically and scaling transparently. Of course, to achieve this scalability, the underlying hardware, the Tuple Space, and the communication system also need to be scalable to handle the proliferation of workers.

## 3. ParLin: Linda meets PARIX

### *3.1. User Interface*

The ParLin library provides a Linda-like coordination language to the Parix [21] operating system and the C programming language.

Unlike the original Linda there is no pre-processor involved. The Linda primitives are implemented as C functions and are supported by a run-time library. The main consequence of this fact is the loss of associative search. In ParLin, as in Brenda [4] and TsLib [28], tuples are seen as arbitrary arrays of bytes along with an integer tag.

The integer tag should be specified when adding, removing or reading a tuple from the TS and is the only value used for matching. The programmer can use tuples containing a C structure with several fields, but the ParLin routines will not recognize its individual elements.

This simple matching strategy has the clear advantage of strongly simplifying the TS search procedure. Additionally, it may be the key for an efficient hash-based distribution of the TS, because the search field is always the same.

---

[2] Obviously, we cannot expect the same performance as on an implementation on a shared memory machine.

Despite the clear performance advantage, this simplification could be unacceptable if it imposed significant difficulties to the programmer. But, from our experience, this is not the case. Most of our programs were easily constructed using only this simple integer tag. If the full associative capability were available, they would not have been structured differently. One exception was the Sieve of Erasthotenes, where having two search fields would have helped. With some other programs the need for a second key field is also felt. In rarer cases, a third field might be useful. An important observation in this context, also reported for instance by [30], is that the fields used to specify an associative search tend to be always the same. We have found no case where a field used to specify one associative search is a formal parameter in another associative search.

Moreover, many master-worker programs tend to have only two tuple types: tuples with jobs to be executed, and tuples with results from the jobs. For such applications, even a single integer search field is overkill, since a couple of bits would be enough.

This leads to the conclusion that using a fixed tag as in ParLin is not a serious limitation of the library, although having two could be useful. Having more than one integer tag is a trivial extension that we will consider in a future version of ParLin.

ParLin provides the six usual Linda primitives (to specify a tuple, the programmer must indicate, besides the tag, the size and address of a block of bytes):

```
void Out(int tag, int size, byte *buffer);
void In(int tag, byte *buffer);
void Rd(int tag, byte *buffer);
void Inp(int tag, byte *buffer);
void Rdp(int tag, byte *buffer);
void Eval(void (*proc) ());
```

Out(), In(), Rd(), Inp() and Rdp() functions perform just like Linda primitives with just one difference: a FIFO policy is followed for each type of tuple (identified with one tag). In the Tuple Space tuples are gathered by tags, allowing concurrent access to different tuple types.

When reading or removing from the TS, the system assumes that the programmer provides a buffer with enough space to store the tuple. This avoids having the size field as an extra parameter. There is no ambiguity because there cannot exist two tuples with the same tag and different sizes.

The last function, Eval(), is used to start a worker process somewhere in the transputer network.

This is another point where we depart from the original Linda primitive, since our *eval* only launches another process, and does not do any *out* of a tuple. This simplification is a consequence of the fact that the library does not know the structure of the tuples. A similar change to the original Linda specification was made in Glenda [25] and P4-Linda [6].

The non specification of the place where the new process is going to be created is maintained in ParLin: the Eval calls are forwarded to the Tuple Space node, where it is decided in which node the worker should be started. This decision is made based on actual information about the processors load[3].

With ParLin more than one worker can be running in each node. This can be useful in some communication intensive applications that usually have poor performance in Linda. With several workers per node, when a process is blocked during an In() operation, the other co-resident worker can use that time to perform its tasks, effectively overlapping

[3] The information recorded is the number of workers in each node, not effective CPU load.

communication and computation. The results of some experiments made to validate this assumption are presented in section 4.

Two other primitives were included for initialization and termination of the program:

```
void InitParLin()
void TermParLin(int level)
```

InitParLin() should be the first ParLin call in the user main() function. It initializes the ParLin environment and spawns all the threads that make the ParLin runtime environment.

TermParLin() should be the last ParLin call of the user main() function, and is used to terminate the ParLin runtime environment. This primitive provides three levels of termination (SOFT, HARD, SMART).

The first level - SOFT - is useful for debugging purposes, it just causes the termination of the ParLin system threads. If any user thread remains active it will not be terminated, causing the Parix program to be blocked.

The second level - HARD - simply terminates the ParLin program by aborting the Parix server at the host computer. It does not care whether user threads are still running or not.

The third level - SMART - provides an elegant ParLin program termination: The system is terminated when all workers are blocked in an In() or Rd() operation. This is a sufficient condition for the termination of any Linda program, provided that it happens after the master has terminated its task.

### *3.2. Tuple Space structure*

An important design decision we made, for the sake of simplicity, was to have only one repository of tuples in the transputer network. As our implementation is oriented to small to medium systems (up to several tens of processors) this centralized Tuple Space does not pose scalability problems.

Previous Linda implementations followed this approach, even for larger and loosely coupled systems, as is the case of Glenda [25], P4-Linda [6] and TsLib [28].

In other implementations of Linda in transputer networks, several Tuple Space replication schemes were tried. However, as far as we know, those implementations always led to unscalable systems. The main reason was the overhead associated with Tuple Space search operations. Other problems are the memory overhead (a large replicated TS consumes a lot of memory in each node), and the slow-down of workers due to the coexistence with the TS Managers in the same node.

For small systems the centralized TS is in fact the best solution. The Tuple Space management is greatly simplified. For performance reasons, we decided that the master is the only one allowed to run in the TS node and, thus TS operations do not slow-down workers. The master is allowed to run in the TS node, if that is the programmer's will, because usually the master does not have too much to do; usually he does a bunch of *outs* with job tuples and sits down waiting for the results to arrive.

As there is no replication of tuples, the memory overhead is kept to a minimum, and no time is lost with the algorithms needed to maintain the consistency of the several copies of the tuple space.

## 4. Performance of ParLin

In this section we present some measurements of our Linda implementation for transputer networks. The measurements were carried out in a Parsytec X'plorer machine with eight nodes with the INMOS T805 and 4Mb RAM. The nodes are connected in a 4x2 grid.

To assess the performance of programs written using ParLin we first decided to estimate the cost of a basic TS operation. For that purpose we used the well-known *ping-pong* program [8]. This program creates two processes *ping* and *pong* that stay in a loop making in() of ping-tuples and out() of pong-tuples, and vice versa.

Table 1 summarizes the results obtained with ParLin along with results obtained with the same program for other Linda or Linda-like implementations.

**Table 1:** Results for the ping pong program

| | ParLin | Modula-L best case | Modula-L worst case | S/Net Linda Kernel | Minix Linda |
|---|---|---|---|---|---|
| Basic Tuple Space operation | 0,576 ms | 1,32 ms | 2,39 | 1,4 ms | 1,165 ms |
| Time for 40000 tuples | 24 s 243 ms | 53 s | 95 s | 56 s | ____ |
| Time for 40000 tuples with 8 workers | 24 s 305 ms | 68 s | ____ | ____ | ____ |

Columns 2 and 3 lists figures for Modula-L, a Linda programming environment having Modula-2 as a base language, that implements a distributed Tuple Space using hashing techniques. The results presented were obtained with a system quite similar to ours [30]. The best/worst case column results were obtained for an optimal/worst placement of tuples in the nodes.

Column 4 lists figures for the S/Net Linda Kernel.

Finally column 5 lists figures for a MINIX Linda implementation[12].

The good performance of ParLin can be explained in part due to:

1) the simple matching algorithm used. Remember that the matching is done using only the *tag* field of the tuple;
2) a simple and efficient centralized Tuple Space implementation;
3) the underlying operating system communication facilities, that proved to be extremely fast [21].

In the third row of the table the same *ping-pong* program is used with the same number of "balls", but with 8 workers in the match. The results show that the basic Tuple Space operations duration does not increase significantly.

### *4.1. Hiding the latency of the Tuple Space operations*

The purpose of this section is to examine whether the effect of having more than one worker per node can lead to a significant increase in performance.

The obtained effect is called "TS latency hiding", because when a worker is blocked in an In() or Rd() operation, another worker in the same transputer can be doing useful work, thus effectively "hiding" the latency of the In() or Rd() operation.

We claim that this effect is only visible for communication intensive algorithms. In those programs the processes spend most of the time communicating and thus communication is

the limiting factor of performance. In computation intensive algorithms this mechanism is not very useful, and can even lead to a small decrease in performance due to the overhead introduced by the existence of extra workers.

**Table 2:** Overlapping computation and communication

| | 21000 jobs Load = 0,64 ms | 21000 jobs Load = 6,4 ms | 21000 jobs Load = 64 ms |
|---|---|---|---|
| 7 workers one per node | 19s 830 ms | 23s 618 ms | 3m 3s 0 ms |
| 14 workers two per node | 19s 837 ms | 20 s 895 ms | 3m 0s 430 ms |
| 21 workers three per node | 19s 865 ms | 19s 703 ms | 3m 0s 333 ms |

To assess this mechanism, a simple program was written. It is a typical bag-of-tasks program, where a master spawns several workers through the network, places a bunch of *job tuples* in the TS and waits for *result tuples.* Workers are structured as an infinite loop with a sequence In(job)-WorkLoad-Out(result). We varied the number of workers per node and the duration of each job executed in the worker, obtaining different program granularities.

Table 2 shows the results.

The program in the second column achieved a speedup of 1,2 when we enrolled three times more workers in the computation, for the same number of processors. The time a worker spent in communications for each job of 6,4 ms was 1,152 ms, having thus a communication to computation ratio of 0.18.

For bigger job granularities, or very fine grained ones, the effect on the performance is negligible. The first result was expected according to the previous discussion. The results for very fine grained jobs can also be explained: workers communicate so frequently that there is almost no computation to do by a worker when the others communicate.

More measurements were made using the benchmarks that will be described in the next section, with similar results.

Our conclusion is that this feature only leads to a significant speedup for applications with a communication to computation ratio in a restricted range.

Nevertheless, we think that the possibility of having more than one worker per node is important, since it allows the programmer to write and structure his application with a number of workers independent of the number of processors that will be available in each particular system where his program is going to be executed. The applications are, in this way, more hardware independent.

## 5. Program Performance Measurements

To assess the performance of programs written with the support of ParLin we used six typical benchmarks that follow the master-worker paradigm of programming.

### *5.1 Benchmarks*

**Nqueens:** Counts the number of solutions to the n-queens problem, with n equal to 12. The problem is distributed by several jobs assigning to each job a possible placement of the first two queens.

**πCalculation:** Computes an approximate value of p by numerically calculating the area under the curve $4/(1+X^2)$ [10].

**TSP:** Solves the Traveling Salesman Problem for a map of 15 cities, using a branch and bound algorithm. The jobs are divided by the possible combinations of the first 3 cities. When a worker finds a new minimum it updates the tuple that keep the minimum solution.

**Matmult:** Multiplies two square integer matrices of size 400 x 400. The matrices are divided in 4 blocks of 100 x 100. Under each sub-multiplication there are four iterations to calculate each block of the result matrix, requiring a total of 16 multiplications.

**Knapsack:** Solves the knapsack problem for a list of 25 numbers. The division of jobs is made by assigning to each job the task of testing only those possible solution vectors whose least significant bits correspond to the job id.

**Alpha:** Searches a game tree using the alpha-beta algorithm. The top of the tree is initially built by the master by traversing the first two levels of the tree. The fanout of each tree is 38 (common value in a chess game). A job consists of evaluating a subtree having as root one of the leaves of the top part of the tree.

*5.2 Speedup Curves*

Speedup curves have become the most straightforward and easily recognized measure of parallel computer utility [20]. In the following figures we plot the speedup chart for the benchmarks described above. This curves measure the utility of parallel computing, not raw speed.

For each benchmark we computed the *absolute* and the *relative* speedup. Absolute speedup is obtained by dividing the execution time obtained using a sequential version of the algorithm by the execution time of the parallel program with *n* workers - *Ts/Tp(n)*, while the relative speedup is computed using the same parallel version with only one worker, instead of the sequential version.

As the absolute and the relative speedup curves are very similar, we only present the chart for the absolute speedup. In each chart, the ideal speedup curve with 45 degrees is also plotted, to be used as a comparison.

As can be seen in Figure 1 and Figure 2, the benchmarks Nqueens, Knapsack and Matmult, show nearly linear speedup. At least for small systems, and for such kinds of applications, having a centralized Tuple Space is not a bottleneck. However, for the other benchmarks (TSP & Alpha) the results were not as good as the previous ones.

For the Traveling Salesman Problem we have two versions of the algorithm: one version (V1), where each worker receives the current minimum with each job and works with it until the end, and the other version (V2) where the worker gets the current minimum with the job, but periodically inspects the global minimum. This version showed a better speedup than version 1 because, since in version 2 each worker looks more often at the current global minimum, he will be faster in abandoning his current job if that job leads to an higher value than has already been found elsewhere. Less useless work get's done, and the solution is found more quickly.

A problem with this frequent polling of a tuple with a global value is a significant increase in the communication traffic. Since most of the reads of that tuple will return the same value, if that tuple were cached in each processor the communication overhead would be significantly reduced.

For a future version of ParLin, we are considering the implementation of a tuple cache mechanism.

ALPHA did not have a speedup as good as the other applications, and we can point out some of the reasons. We used a tree with fanout equal to 38, thus generating 1444 jobs with a quite fine granularity and a high ratio communication/computation. Most of the times, when a worker finishes a job, it's result is discarded because there is already a better value, which also leads to poor performance.

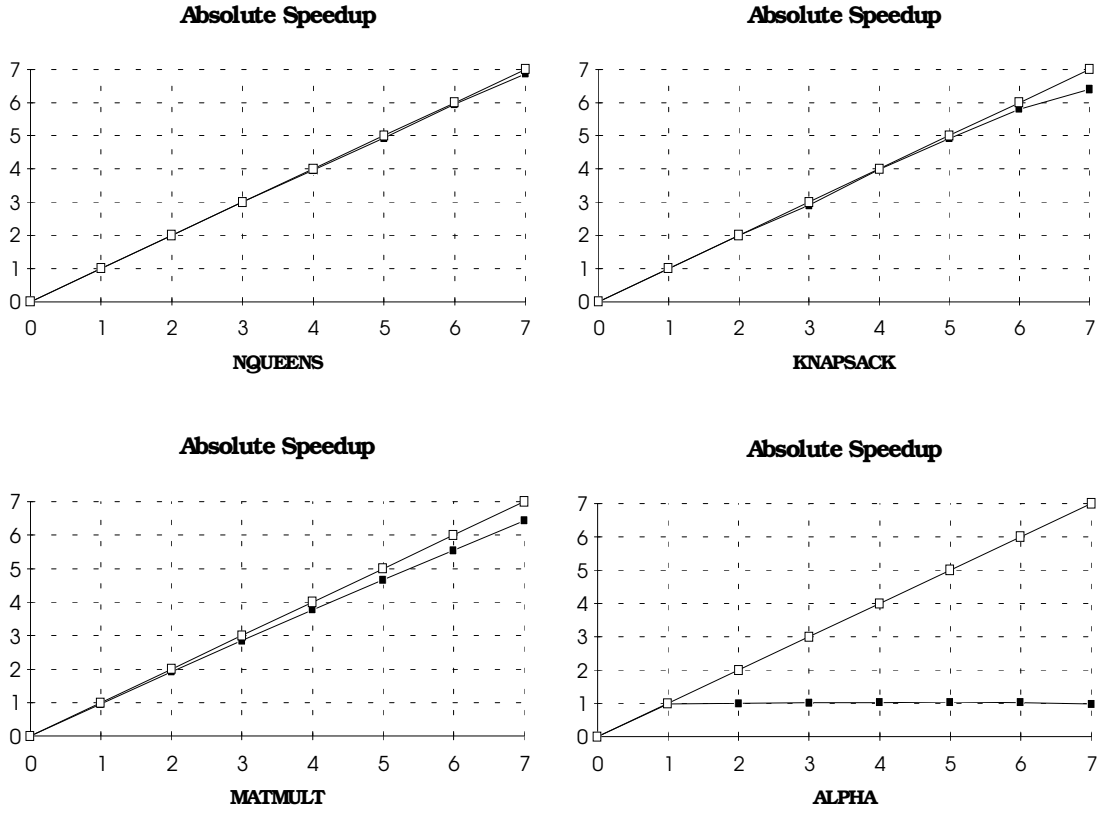


**Figure 1** - Speedup curves for Nqueens, Knapsack, Matmult, Alpha.

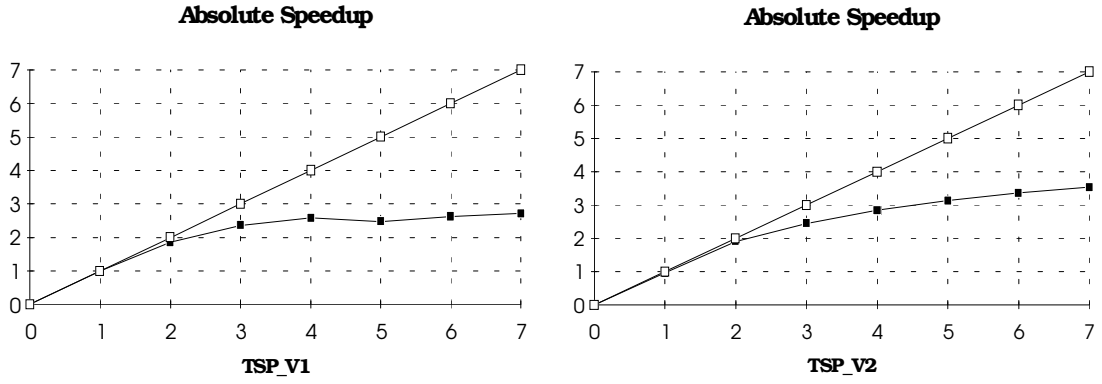


**Figure 2** - Speedup curves for TSP

From this performance study we can draw some conclusions:

1) The centralized Tuple Space is not a bottleneck, at least for small transputers systems.
2) Applications with a medium/large job granularity scale almost linearly with the number of processors; on the contrary, those applications which have a small job granularity may not scale so well, in spite of having the fastest Linda primitives, as shown with the *ping-pong* example (see section 4);

## 6. Tuple Space Scalability

In this section we will discuss how to implement a scalable Tuple Space. The goal is that the time taken to execute a basic tuple space operation remains constant, or at least does not increase significantly when we add more processors to the system.

An approach followed by the Linda authors in their C-Linda implementation [11], makes use of compiler technology. The C-Linda compiler converts knowledge of the program's pattern of Tuple Space accesses into faster code. However, the compiler has a limited capacity of understand the particular details of each application.

To overcome that limitation, [32] suggested the use of explicit mapping functions, which are user-defined and application specific. These functions should give the programmer control over the way in which the Tuple Space is decomposed, providing optimization at the cost of transparency. But we think that one of the greatest Linda advantages is transparency, and it should not be sacrificed.

In the following we will assess some usual Tuple Space distributions schemes, and look for one that meets our needs.

### *6.1. Tuple Space Distribution Policies*

Various approaches to implement a distributed Tuple Space in distributed memory systems have been suggested so far. The most significant are:

- Uniform distribution
- Intermediate uniform distribution
- Hashing

In a uniform distribution scheme [2], tuples are broadcast by their generating nodes to all nodes within a predetermined node's "out-set", and requests for tuples are broadcast to all nodes in the node's "in-set". Within this general model, there is a large spectrum of possibilities. We can implement out(T) by broadcasting T to all the nodes in the network, and thus have *ins* and *rds* making local searches and global invalidation due to the full replication of tuples; or we can have an inverse scheme where an out(T) stores T locally, and *ins* and *rds* make networkwide broadcasts to locate matching tuples.

This scheme is intrinsically not scalable due to the broadcasts and has a big memory overhead due to the full tuple replication.

Another possibility is an intermediate uniform distribution [2]. It works by having a restricted replication of Tuples through the network. To implement out() we broadcast T to $\sqrt{N}$ nodes, the "out-set"; in() and rd() work by broadcasting requests to different $\sqrt{N}$ nodes, the "in-set". We must design in-sets and out-sets in such a way that each in-set includes at least a member of each out-set. This scheme is a significant improvement upon the pure uniform distribution. While the previous scheme required that all N nodes participate in a given out-in transaction, this only requires $2*\sqrt{N}$. It has also lower storage

costs because tuples are not fully replicated, requiring only $\sqrt{N}$ times more space instead of the N times for the pure uniform case.

Thus, restricted uniform distribution seems to be a good solution for transputer networks. But unfortunately it is not. [13] has implemented this distribution scheme in transputer networks and concluded that it was too inefficient to be of practical use due to communications overheads.

There is still another popular solution, the hash-based scheme [2] which is used in several Linda implementations, as is the case of the Linda kernel for the iPSC Hypercube or Modula-L [30]. Tuples are distributed across the network and mapped to nodes according to an hash function applied to the "type signature" of the tuples. This scheme has the advantage of having low costs in terms of storage and communication bandwidth because there is no tuple replication and no need for message broadcast.

However, as the network gets larger, the tuple traffic increases and the same hashing function is no longer useful. The hashing function should be capable of mapping the same tuples to a greater number of nodes to avoid communications bottlenecks in the tuple buckets, but that is not always possible. In fact, for bag-of-tasks applications, where tasks are usually represented as tuples of the form ("TASK",<task description>), the hash functions would map all the task tuples to the same node, creating a bottleneck. A solution was proposed for this problem by using extended keys [30]. The authors suggest the use of more fields than the tuple identifier as a key to the hash function, but then the value of those fields must always be given in an In() operation. This is quite artificial in many cases, as with the bag-of-tasks example above.

### *6.2. Adaptive Hashing*

In spite of the shortcomings presented before, we think that hashing is the most promissing technique from the scalability point of view, since its communication requirements can be kept bounded and, because there is no replication of tuples, no consistency algorithm is needed. Additionally, the fixed search key of ParLin is particularly adequate for hashing, since there is never any doubt about what part of the tuple should be used as input for the hash function.

But where we depart from previous proposals is in our claim that the hash function should use as input not only the value of the search key, but also the size and topology of the underlying system, and information about the run-time distribution of search keys. The information about the distribution of the search keys could be gathered by the system in some test runs, and used to enhance the tuple distribution in the following runs. It is important to note that this test runs need not be a new development phase, since during the final debugging of a program many test runs are made anyhow. The last ones could be used to simultaneously gather the required statistics.

This statistical information can go on being perfected each time the program is run, so that successive runs of the same program execute faster and faster. The programmer can also give the system some hints about the distribution of search keys, but this is not a requirement - the programmer can choose not to do it, and even if he does, it is something totally separated from the program itself, with no impact on the way the program is structured.

This statistical information would include information like the different values that the search key takes, and which processes do Out()s and In()s on each kind of tuples.

We call this technique adaptive hashing.

For the case of the bag-of-tasks application, for instance, where most of the tuples are job tuples and have the same search key, the tuple space could be simply partitioned, with

part of the "job" tuples going to each partition, and with each In() operation being directed to the nearest repository of such tuples. The result tuples could be kept centralized in the same node where the master resides, since the system would quickly verify that only that process ever executes an In() on that kind of tuple.

Our present research goes along this line, and intends to test the feasibility of this adaptive hashing.

## 7. Conclusions

A library called ParLin, implementing a Linda-like programming environment for transputer networks running the Parix operating system, has been presented. It is currently only meant for small to medium systems (several tens of processors), but work is under way to scale it up to larger systems.

The current implementation has been shown to be quite efficient, due mainly to the very simple internal structure of the supporting run-time system, that uses a centralized tuple space. Speedup curves have been shown for several typical programs.

The problem of tuple space distribution has been discussed. We claim that some variation of hashing in the most promissing way to achieve an efficient distribution of the Tuple Space for larger systems, and we presented our proposal, that we call adaptive hashing. According to it, the hash function (perhaps more appropriately called mapping function) should use as input not only the search fields of the tuple, but also the topology and size of the underlying system, along with information about the tuples' search key distribution. This distribution can be gathered at run-time, and perfected in successive runs, or given by the programmer (or both).

Our research presently tries to assess the feasibility of this approach.

For small to medium systems, we think that the centralized approach used in the current version of ParLin is the most efficient one.

## Acknowledgements

We would like to thank Luis Silva for motivating our research, for providing a lot of literature about Linda, and for his thoughtful comments on an earlier version of this paper.

The last two authors are partially supported by the EEC under the FTMPS ESPRIT project - 6731.

## References

[1] S.Ahuja, N.Carriero, D.Gelernter. "Linda and Friends", IEEE Computer, pp.26-34, August 1986.

[2] S.Ahuja, N.Carriero, V.Krishnaswamy. "Matching Language and Hardware for Parallel Computation in the Linda Machine", IEEE Transactions on Computers, Vol. 37,No. 8, August 1988.

[3] D.Bakken, R,Schlichting. "Supporting Fault-Tolerant Parallel Programming in Linda", Technical Report 93-18, Dept. of Computer Science, Univ. of Arizona, June 1993.

[4] M.Braner, et al. "Trolius: A Software Solution for Transputers and Other Multicomputers", NATUG1 - Transputer Research and Applications, IOS Press, 1990.

[5] A.Bruell, H.Kuchen. " Implementierung von Interprozesskommunikation ueber Tupelraeume auf Transputersysteme", Diplomarbeit, Rheinisch Westfaelische Technische Hochschule Aachen, 1993.

[6] R. Butler, E. Lusk. "Pa-Linda: A Portable Implementation of Linda", Proc. 2nd Int. Symposium on High-Performance Distribution Computing, IEEE Computer Society Press, 1993.

[7] C.Callsen, I.Cheng & P.Hagen. " The AUC C++ Linda System", Aalborg University Center Denmark, EPCC Technical Report 91-13, pp 39-73, June1991.
[8] N.Carriero, D.Gelernter. "The S/Net's Linda Kernel", ACM Transactions on Computer Systems, Vol. 4, No. 2, pp.110-129, May 1986.
[9] N.Carriero, D.Gelernter. "Applications Experience with Linda", ACM /SIGPLAN, Parallel Programming: Experience with Applications, Languages and Systems, 1988.
[10] N.Carriero, D.Gelernter. "How to write Parallel Programs - A first course", MIT Press, ISBN 0-262-03171-X, 1990.
[11] N.Carriero, D,Gelernter. "Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler", Languages and Compilers for Parallel Computing", Pitman Pub. pp114-125, 1990.
[12] P.Ciancarini, N.Guerrini. "Linda Meets Minix", ACM Operating Systems Review, vol 24, no. 4, pp 76-92, October 1993.
[13] Craig Faasen."Intermediate Uniformly Distributed Tuple Space On Transputer meshes", Lecture Notes on Computer Science 574, pp. 157-173, June 1991.
[14] D.Gelernter, D.Kaminsky. "Supercomputing out of recycled garbage: Preliminary Experience with Piranha", Proc. ACM International Conference on Supercomputing, 1992.
[15] "Transputer Reference Manual", INMOS Limited - Prentice Hall, 1988.
[16] "Transputer Technical Notes", INMOS Limited - Prentice Hall, 1989.
[17] M.Kaashoek, H.Bal, A.Tanenbaum. "Experience with the Distributed Data Strucure Paradigm in Linda", Proc. 1st Usenix Workshop on Experiences with Distributed and Multiprocessor Systems, pp. 175-191, October 1989
[18] S.Kambhatla, J.Walpole. "The Interplay Between Granularity, Performance and Availability in a Replicated Linda Tuple Space", Proc. 6th International Parallel Processing Symposium, March 1992.
[19] W.Leler. "Linda Meets Unix", IEEE Computer, pp. 43-54, Feb. 1990.
[20] T.Lewis, H.El-Rewini. "Intoduction to Parallel Computing", Prentice-Hall International Editions, ISBN 0-13-498916-3,1992.
[21] "PARIX 1.2 Reference Manual and User Manual", Parsytec Computer GmbH, March 1993.
[22] J.Pinakis, C.McDonald. "The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language", Presented at the 16th Australian Computer Science Conference , Brisbane, February 1993.
[23] J.Pinakis. "Remote Thread Execution", 16th Australian Computer Science Conference, February 1993.
[24] G.Schoinas. "POSYBL: Implementing the blackboard model in a distributed memory environment using Linda", Department of Computer Science, Univ. of Crete, EPCC Technical Report 91-13, pp 105-116, June 1991.
[25] B.Seyfarth,J.Bickham,M.Arumughum. "Glenda Installation and Use", December 1993
[26] K. Shekhar, Y.Srikant. "Linda Sub System on Transputers", Proc. Transputing '91, IOS press, pp 246-261, 1991.
[27] E.Siegel, E.Cooper. "Implementing Distributed Linda in Standard ML", Carnegie Mellon University, CMU-CS-91-151, October 1991.
[28] Luis Moura Silva, Bart Veer, João Gabriel Silva. "The Helios Tuple Space Library", Proc. 2nd Euromicro workshop on Parallel and Distributed Processing, pp 325-333, January 1994.
[29] Luis M. Silva, Bart Veer, João G. Silva. "A Fault-Tolerant Tuple Space Library", Technical report DEE-UC-023-93, Univ. Coimbra, 1993.
[30] J.Trescher, F.Bieler, C.Hinrichs. "Modula-L: Implementation of the Linda Model for Arbitrary Transputer Networks", Parallel Computing: From Theory to sound Practice, IOS Press, 1992.
[31] R.Whiteside, J.Leichter. "Using Linda for Supercomputing on a Local Area Network", Proc. Intl. Conference on Supercomputing, pp 192-199,1988.
[32] G.Wilson. "Improving the Performance of Generative Communication Syatems by Using Application-Specific Mapping Functions", EPCC Technical Report 91-13, pp 129-142, June 1991.