Distributed Tuple Spaces in Highly Parallel Systems

Antonio Corradi

Dipartimento di Elettronica Informatica
e Sistemistica - Università di Bologna
2, Viale Risorgimento
40136 Bologna - ITALY
Ph.: +39-51-6443083
Fax: +39-51-6443073

Letizia Leonardi, Franco Zambonelli

Dipartimento di Scienze dell'Ingegneria -
Università di Modena
213/b, Via Campi
41100 Modena - ITALY
Ph.: +39-59-378517
Fax: +39-59-370040

E-mail: {acorradi, lleonardi, fzambonelli}@deis.unibo.it

Abstract

The tuple space abstraction is widely recognized as a powerful and general coordination model for parallel programming. However, it is based on the abstraction of a global space and therefore difficult to be efficiently implemented in highly parallel architectures. The paper discusses the implementation issues of the tuple space abstraction on large parallel systems and proposes a new implementation model suitable for massively parallel systems: the model achieves scalability by organizing the system in a hierarchical way and by encouraging the presence of multiple tuple spaces with a constrained scope. The paper describes and analyses the model for a transputer-based implementation: the hierarchical organization makes the access time to tuples proportional to the locality of the references and, in most cases, bounded by the logarithm of the system size.

***Keywords**: Linda, Multiple Tuple Spaces, Massively Parallel Architectures, Scalability, Locality.*

NOTE: The parts of the paper enclosed by a border suggest sidebars.

Distributed Tuple Spaces in Highly Parallel Systems

Antonio Corradi, Letizia Leonardi, Franco Zambonelli

1. Introduction

The interest in distributed memory architectures has been growing during the past decade, thank to their cost effectiveness and scalability. Nevertheless, computational models based on shared memory are far from disappearing from the scene of parallel and distributed technology: years of experience in sequential and super-computer programming make the shared memory a too well digested concept; in addition, the PRAM model based on shared memory has too deeply influenced the current background and solution understanding in parallel processing [1].

In parallel and distributed systems, the abstraction of a globally shared space can assume new roles, by playing as:

- a *general-purpose communication system*; communication between distributed processes can operate with the shared memory, making it easy the implementation of several communication modes;
- a *place to store monitoring and/or debugging information*; in a parallel or distributed environment, it could be difficult and expensive to obtain global information about the system and its applications, although the information is essential;
- a *way of access to I/O devices*; the mapping of I/O in shared memory solves the problem of granting distributed activities a fair access to external devices;
- a *virtual memory support*; shared memory gives any process an amount of addressable memory to overcome the limit of the local one.

A crucial point in implementing any abstraction is to choose the right abstraction level. For the shared memory, this means to choose the elementary units of access to the memory and its related operations. On the one hand, the level of variables to be "read" and "written" is too low a level and would require users to deal with low level synchronization issues. On the other hand, a shared objects level that embodies the interface and the synchronization policy of access is too high a level to grant generality and efficiency in the implementation.

The *Linda* model introduces the *tuple space* abstraction and can be considered at an intermediate abstraction level [2]: the elementary memory units - called *tuples* - are records of typed fields, associatively accessed via a few operations based on a pattern-matching mechanism whose semantics a priori solves synchronization issues. The simplicity and the generality of this model motivate its adoption in several parallel and distributed environments based on different paradigms and make it a *general coordination model* for parallel programming [3].

Despite its power and diffusion, the tuple space model face all the issues of its implementation. Excluding any centralized solution that introduces bottlenecks and limits the system scalability [4], large parallel systems require distributed tuple space implementations: tuples should be distributed among the nodes of a distributed memory architecture and made transparently available to every node of the system [5].

The paper adopts the distributed perspective and analyses different tuple distribution strategies toward an efficient and scalable implementation. Several implementations exist and have been proposed based on different strategies; however, they does not meet the scalability requirement needed for their effective use in large sized system. The extension of the Linda model toward *multiple tuple spaces* - each with a well defined scope - may enforce locality in applications but requires tuple distribution strategies capable of exploiting locality in order to achieve better scalability.

The implementation model proposed in the paper defines a hierarchy of tuple spaces distributed across a tree of nodes: tuples are replicated in the tree with a depth that depends on the scope of the tuples space they belong to. The paper shows that this solution avoids bottlenecks in the hierarchy and achieves tuple access times proportional to the *degree of locality* of the access itself and *logarithmically bounded* with respect to the system size.

2. The Tuple space model

The Linda model heightens the classical shared memory model by substituting it by the tuple space concept, a blackboard where concurrent process can asynchronously and anonymously interact by storing and by retrieving information in the form of tuples [2]. A *tuple*, the elementary unity of access to the tuple space, is an ordered set of typed fields (defining the tuple structure) with either a defined or an undefined value (actual and formal value, respectively). An additional actual field, a string, is usually adopted to classify tuples by type, i.e., by their structure. For example, the following notation may identify two tuples of type “T1” and two tuples of type “T2”:

```
int x, y, z;
("T1", x ?, 'a', 4.14);
("T1", 12, 'b', 7.37);
("T2", 12, y ?, 'p');
("T2", z ?, 5, 'p');
```

The question mark identifies a formal value.

Two tuples "*match*" if they belong to the same type and the values of their actual fields are the same. In the above example, the first two tuples, though belonging to the same type, do not match because their actual values differ. The third and the fourth tuples, instead, match. The result of the match is the unification of the two tuples w.r.t. their actual values. In the above example, the match of the third and of the fourth tuples produces the following tuple:

```
("T2", 12, 5, 'p');
```

The operation to interact with the tuple space are *Out*, *In* and *Read*, all with one tuple as a parameter:

```
Out("T2", 12, 3, 'f');
In("T2", 12, x ?, 'g');
Read("T3", y ?);
```

The Out operation stores the parameter tuple in the tuple space. The Read operation searches in the tuple space for one tuple matching with the parameter one, so to realize a form of associative access, i.e., by content, to the memory. In case one match occurs, the unified tuple is yielded to the reading process. The In operation is similar to the Read one, but the matching tuple is extracted from the tuple space and is no longer available to other matching operations. Both the In and Read operations can block the calling process: the execution does not continue until a match occurs. When operations produce multiple matches (one In call that matches with several tuples or one tuple that matches with several Ins) only one of them is committed in a non-deterministic way, so to grant fairness. No operations are defined that can modify a tuple *in loco* - i.e., within the tuple space: this grants high-level consistency and rules out any low-level synchronization issue.

There is a strict duality of the In (and Read) operations with the Out ones: when one In or Read operation does not produce any match, it is necessary to store the parameter tuple - "request tuple" - in the tuple space, so to test for future match possibilities (with incoming tuples produced by successive Out operations). Conversely, when one Out operation produces a tuple, it is necessary to test the presence of "request" tuples to match with. For sake of simplicity, in the following we indicate tuples as In-tuples, Read-tuples or Out-tuples, depending on the operation that produced them. In addition, the class of In-tuples and Read-tuples is indicated as dual of the Out-tuples class.

Though several other operations have been proposed to access to a tuple space, their semantic is not widely accepted and our analysis neglects them [6].

3. Implementing distributed tuple spaces

Any abstract model, apart from its conceptual power, can find its practical validity only by means of an efficient and scalable implementation.

The global abstraction upon which the tuple space model is based rules out any centralized implementation, either hardware via a shared-memory architecture or software via a global tuple space server: that would introduce a bottleneck hindering the system scalability [4]. The alternative solution is to implement the tuple space abstraction via a software layer onto a distributed shared memory architecture: the logical space is physically distributed among different nodes, each one with its local memory and autonomous execution capacity; tuples are allocated on the nodes of the systems and eventually replicated on several nodes and made transparently available to every node of the system [5].

The basic requirement for a distributed solution is to provide a *distribution policy* that grants any potential match to occur. In other words, the distribution policy must guarantee that any tuple T, if there exist a tuple S of the dual class with which to potentially match, the match must given the possibility of occur somewhere in the system, i.e., onto at least one of the system nodes.

Distribution policies and matching mechanisms are separate issues: when tuples are distributed and eventually replicated onto the nodes of a distributed architecture, the search for matching onto one node can occur only on the basis of the tuples locally stored there and without any knowledge of the global distribution policy; conversely, the tuple distribution policy can disregard the implementation of the local matching mechanisms. The implementation of efficient matching mechanisms is close to a database management problem; the paper concentrates on distribution policies.

There exist two main classes of tuple distribution policies: non replicative and replicative ones, depending on whether tuples exist uniquely in the system or they can be replicated on several sites.

3.1 Non replicative policies

Non replicative policies assign tuples to the nodes of the system on the basis of their structure, i.e., their type: tuples belonging to the same type have to be allocated onto the same node, so to grant any possible match to occur.

While application-dependent strategies can be conceived that statically decide which type of tuples - within an application - to allocate on which nodes of the system, a more general and application-independent approach may assign tuples to the system nodes on the basis of a hash function applied on the tuple type field (figure 1):

node_of_residence = hash_function("type")

Being a match between possible only between two tuples of the same type, the hash solution implements a correct tuple distribution policy: it allocates on the same node all tuples of the same type and grants any possible match to occur.

The hash solution is simple to implement and completely independent on the structure of the target architecture. However, it lack in scalability. On the one hand, it hardly achieves a homogeneous distribution of tuples in the system, especially when the number of nodes is high with respect to the number of different tuple types. On the other hand, it does not exploit any locality principle, because it does not take into any account the relation between the node of the system from where a tuple is produced by a given process (*source node*) and the node onto which the tuple is stored. Consequently, in large sized systems, a hash solution is likely to produce a high overhead of communication and to make the access times to remote tuples unacceptably high.

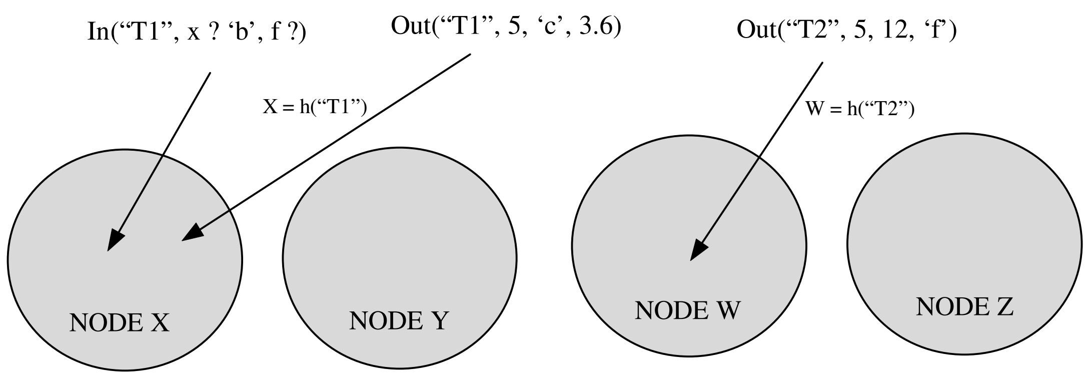

Figure 1. Hash Based Tuple Distribution Strategy

3.2 Replicative policies

The physical replication of one logical entity can provide locality in the access to the entity and better availability. In case of a distributed tuple space, a replicative distribution policy may provide to limit the access times to tuples by enforcing locality on matches, i.e., by making matches occur close to the nodes from where (one of the) tuples are produced.

Tuple replication - as any replication - requires the introduction of coherence protocol to grant consistency of different replicas: once an actions change the state of one replica, this must change must reflect on all other replicas. The consistency problem introduced by tuple replication is similar to the one of cache based or distributed shared-memory systems [7]: in those systems, inconsistency typically arises because of concurrent write operation to different replicas of the same logical memory cell; in the tuple space model, inconsistency may arise because of concurrent matches of the same tuple in different nodes.

The presence of two dual classes of tuples introduces two different sources of inconsistency:

- a replicated Out-tuple matches with two different In-tuples on different nodes and it is extracted twice from the tuple space;
- a replicated In-tuple matches with two different Out-tuples on different nodes and extract them both.

Figure 2 reports two different Out tuples replicated in node X and Y respectively. If a In-tuple that matches with both of them joins the tuple space and it is replicated in both node X and Y, two matches occur and committing both causes the extraction of two Out-tuples by a single In-tuple. Conversely, if different In-tuples are stored on two different nodes (W and Z in figure 2) waiting for incoming Out-tuple, when one Out-tuple joins the tuple space and is replicated in both nodes, committing both matches causes a single Out-tuple to be extracted from the tuple space by two different In-tuples. Not only these situations make the tuple space inconsistent, but also break the semantic of the operations.

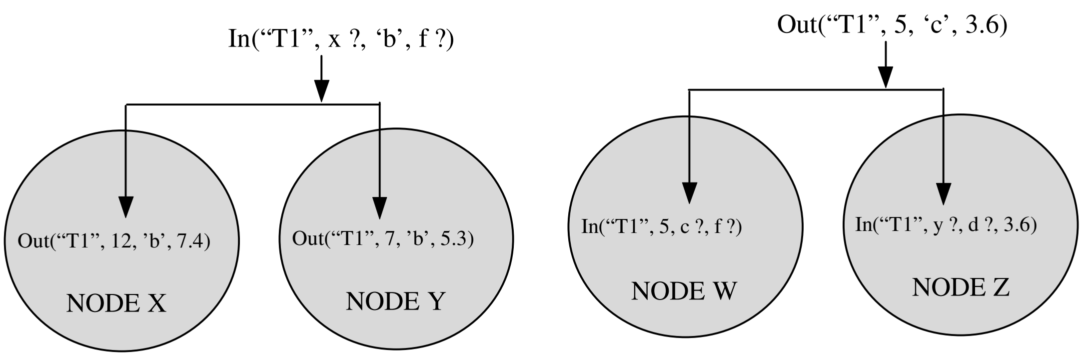

Figure 2. Inconsistencies due to concurrent matches

In a replicative tuple distribution strategy, coherence protocols are in charge to correctly implement the semantic of the operations - that allows only one match per operation - and to maintain the internal consistency of the tuple space. The degree of coordination among the nodes of the systems needed by the coherence protocols characterizes the scalability of replicative distribution policy.

Full replication techniques are based on a global replication scheme and can be implemented in two ways. On the one hand, Out-tuples can be fully replicated in every node of the system while In and Read-tuples are not replicated and have the duty of search for local potential match (this scheme is shown in figure 3). On the other hand, the dual solution stores each Out-tuple only on one node while In and Read-tuples are globally replicated so to search for a match in every node of the system. Any possible match is clearly allowed to occur by the globality of the above replication schemes.

Both solutions are very simple and makes no assumptions on the underlying system topology. However, they are based on a global replication scheme, intrinsically not scalable, that makes them not viable for large parallel systems: the costs of the needed coherence protocols increase with the system size and becomes the bottleneck of the implementation.

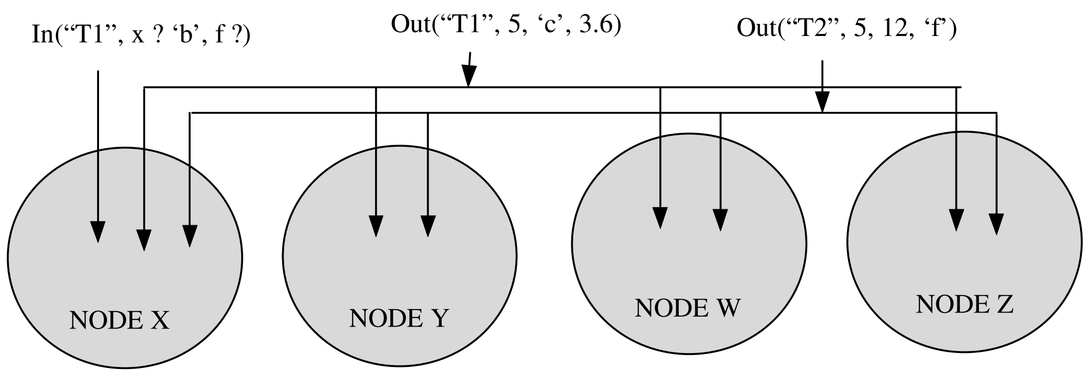

Figure 3. Full Replication Strategies

Better scalability can be achieved by limiting the replication degree of tuples and making it grows slower than the system size. *Partial replication* policies achieves this goal by replicating tuples only in a subset of the nodes of the system. Potential matches are granted if the nodes of replication of Out-tuples forms a *non-null intersection* with the nodes of replication of In and Read-tuples. For example, if the system topology supports a 2-D grid embedding, each Out-tuple can be replicated along the column of the grid of its source node, while each In and Read-tuples can be replicated along the row, as in figure 4. In this case, the replication degree of tuples - and the cost of the associated coherence protocols - grows only with the square root of the system size.

Partial replication strategies are normally architecture dependent: the replication strategy must take into account the structure of the communication topology, that influence the implementation costs of both replication strategy and coherence protocols. Thus, they achieve better scalability but are less general than non replicative or full replication strategies.

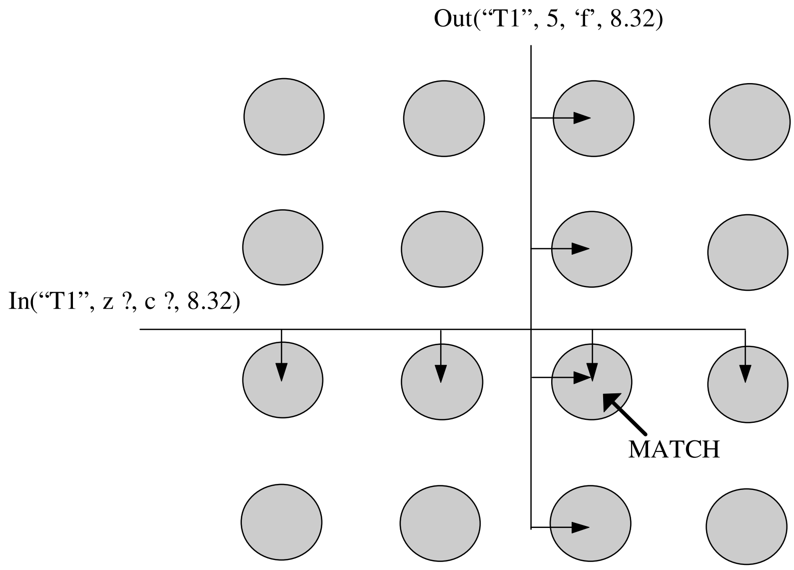

Figure 4. Partial Replication on a 2-D Grid

4. Multiple tuple spaces

Whatever the implementation, the global model intrinsic to the tuple space abstraction tends to introduce scalability problems, not only in terms of performance but also of application management: the growth in size of parallel and distributed systems and the increasing complexity of parallel applications requires a modular approach to enforce locality and composition-based approach.

Modern parallel programming models and environments tends to apply the same modularity principles that have guided the evolution of sequential programming models, by locally confining the references of parallel applications. Instead of designing a parallel application as an unordered bag of interacting processes, modern parallel programming styles recognize that the interactions between the processes of an application are rarely unstructured. Instead, clusters of intensively interacting processes can be identified and composed in a higher-order abstraction that encapsulates local references and defines a precise interface to the external world. Large applications can then be incrementally composed starting from these structures, in a bottom-up style. The result is a structured application that enforce the modularity principle and in which most references (let them be references to other processes or to shared variables) are usually confined within clusters, similarly to local variables of a sequential procedure.

The urge for a structured style of parallel programming can be found not only in several parallel and distributed programming environment [8, 9] but also in extensions of the PRAM model toward the nesting of multiple PRAMs with a limited scope [10] and, again, in distributed logic programming environments that discipline and confine the scope of knowledge basis [11, 12].

With reference to the tuple space model, the above considerations encourage extensions toward the presence of multiple tuple spaces [13]: application processes can structure their accesses to several tuple spaces and make use of them as local interaction spaces.

In a multiple tuple space model, one identifier is associated with a tuple space:

```
TupleSpace *ts1, *ts2;
```

The In, Read and Out operations must specify the identifier of the referred tuple space:

```
ts1.In(tuple1);
ts2.Out(tuple2);
```

The matching mechanism is restricted to act within the specified tuple space.

Multiple tuple space model can grant better scalability if the logical locality of the model is enforced in the implementation too. In other words, the presence of multiple and well-confined tuple spaces must produce locality at the architectural level. The presence of multiple tuple space does not achieve any benefits if the tuple distribution policy globally distribute tuples in the system without taking into account their scope; this is the case for example, of a hash distribution policy.

5. The tree Structured tuple space model

Our tuple space implementation model follows a partial replication policy, as in section 2: the main point is to make the replication degree of tuples (and the costs of the introduced coherence protocols) grow slower than the system size and to lead to the implementation of a multiple tuple spaces model that physically exploits logical locality.

In addition, the implementation model adopt an a priori solution to limit the overhead of the tuple space activity, i.e., matching search, replication and coherence protocol, on applications: the nodes devoted to the execution of application processes (called execution nodes) are separated from the nodes devoted to the implementation of the tuple space (called memory nodes). Not only this separation avoids any overhead of the memory on the applications, but also grants more predictable performances.

5.1 System configuration

The system is shaped after a tree whose leaves are the execution nodes (see figure 5). Each execution node is connected to a memory node of the immediately higher level. Each memory node is connected, in its turn, with one memory node of the higher level and some node (L-1, if L is the connectivity degree of a node) of the lower level, that can be memory nodes or execution nodes. The root memory node has no higher memory nodes to connect to.

The proposed model can be implemented in any kind of architecture with a limited connection degree (at least 3), and minimizes the number of nodes dedicated to the implementation of the tuple space. In a 100 T800 transputer based architecture, for example, the memory nodes required to realize the tree are 34, number that can be lowered only by connecting the memory nodes in a ring.

We emphasize that the execution nodes can be freely interconnected to each other, without influencing at all the tree structure. This can be usefully exploited in making the message-passing and the shared memory paradigm coexist: environments can mix both forms of interaction to exploit locality via message passing and to solve any globality by means of the shared memory [14].

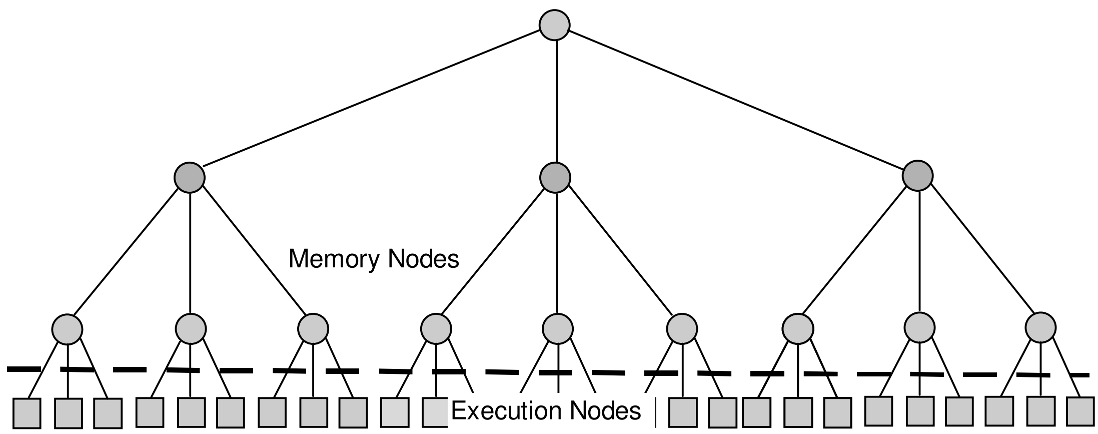

Figure 5. The tree structured shared memory model

5.2 Tuple distribution strategy

Let us temporarily disregard the presence of multiple tuple spaces and consider the tree as a global and unique tuple space.

The adopted tuple distribution strategy is based on a replication scheme that follows the vertical direction of the tree. Tuples (both In-tuples, Read-tuples and Out-tuples) are replicated along the whole path that starts from the source execution node and goes up to the root. When a tuple enters the tuple space, it climbs the tree by replicating itself in every node it crosses, after a control of the possibilities of match in those nodes. Since the height of the tree grows logarithmically with the system size, the replication degree of tuples (and so the cost of the coherence protocol) also follows this trend, making the model fairly-scalable.

The above replication scheme grants any potential match: every tuple of one class shares with every tuple of the other class all the nodes included in the common replication path, from the lowest common ancestor to the root; at least, two tuples share only the root. An immediate optimization of the scheme makes tuple replication stop as soon as one match is found on one node (figure 6): in a couple of matching tuples, only one of them (the first that joins the tuple space) is replicated up to the root while the replication of the other is blocked at the lowest common ancestor.

As usual, when one match occurs, the unified tuple must be yielded to the execution node that requested it.

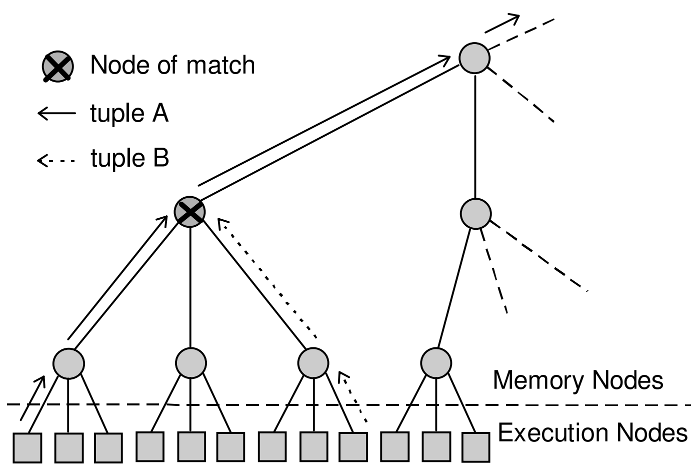

Figure 6. Replication and Match
the tuple A enters the tuple space and it is replicated up to the root
when a dual tuple B enters the tuple space and match with A stops to replicate itself

5.3 Multiple tuple spaces in the tree

The introduction of multiple tuple spaces in the hierarchical model is simple: since each memory node of the tree identifies a sub-tree of which it is the root, any of these sub-trees can be considered as one tuple space in itself, independently on the fact that it is part of a larger tree. Thus, several logically separated tuple spaces can coexist within the same physical tree of memory nodes (see figure 7).

With regard to the tuple distribution strategy, each tuple is assigned a given tuple space, extending along a sub-tree of memory nodes, and it is replicated only in the sub-tree corresponding to its scope. At one extreme, a tuple can belong to the minimal tuple space composed of one memory node only, directly connected to a few execution nodes that are the only one allowed to accessed to this tuple space. At the other extreme, a tuple can be replicated along the whole tree, thus belonging to a global tuple space that every execution node can access to.

The main point of this solution for accommodating multiple tuple space in a system is that the locality of references within a tuple space reflects the physical locality in a sub-tree: the enforcement of the locality principle is met not only by the programming model but also by the implementation.

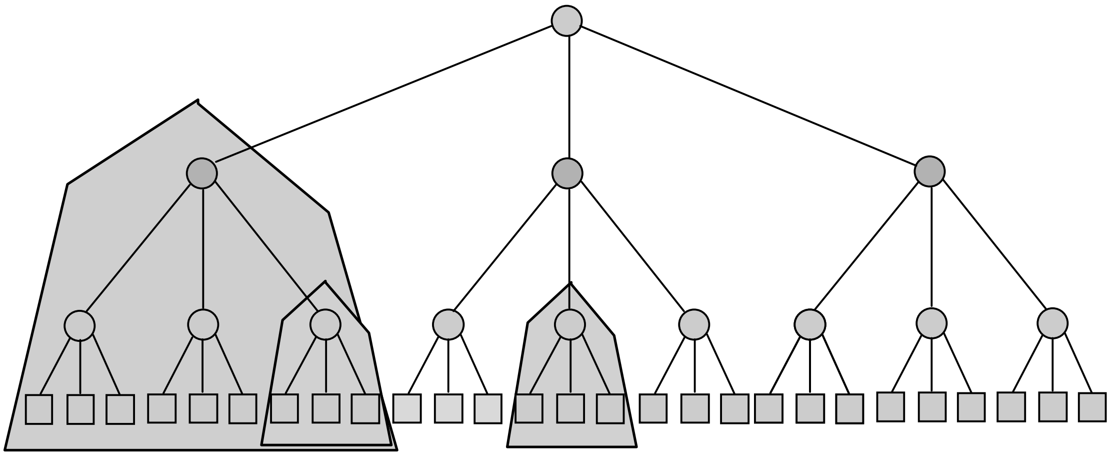

Figure 7. Different sub-trees may identify tuple spaces with different globality scope

6. Matching and coherence protocols

A replicative distribution policy requires coherence protocols to grant the following coherence conditions:

1) any given Out-tuple is extracted by only one In-tuple;

2) any given In (Read) tuple extracts (read) only one Out-tuple;

The duality of these conditions leads to the implementation of a protocol composed of two independent phases, that can be considered complete coherence protocols in themselves. For any of the two, different solutions may be conceived for their implementation.

For sake on simplicity, we concentrate only on matches occurring between In-tuple and Out-tuples and disregard the presence of Read-tuples, by recalling that they must honour only the second coherence condition.

6.1 First coherence condition

Two dual solutions can be adopted to grant the first coherence condition within a tuple space (i.e., a sub-tree) in the hierarchical model. We call them *UP-DOWN* and *DOWN-UP* protocol, respectively.

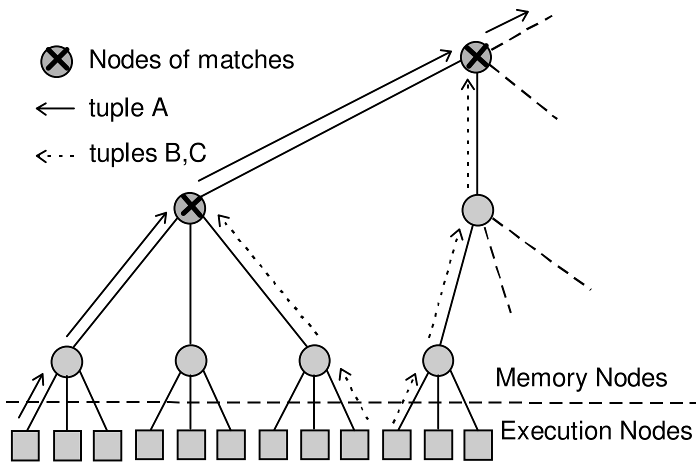

Figure 8. Two tuple B and C match, on different nodes, with the same replica of a dual tuple A

Let us suppose that, in a node, an In-tuple belonging to a given sub-tree matches with an Out-tuple belonging to the same sub-tree. By recalling that all tuples are stored in the lower levels of the sub-tree down to the path to the respective source execution nodes, the Out-tuple can be assigned to the matching In-tuple - and extracted from the tuple space - only if no other In-tuple has already matched - and extracted - a different replica of the same Out-tuple. The *UP-DOWN protocol* (figure 9a), before assigning one Out-tuple to one In-tuple, explores the sub-tree from the matching node down in the path that goes to the source node of the tuple, and extracts the Out-tuple from every node. If the Out-tuple cannot be found in one of these nodes, it has been already extracted from another matching In-tuple: in this case, a *NOT-OK* message notifies to the matching node the unavailability of the Out-tuple and causes the matching In-tuple to be stored and replicated up in the sub-tree, waiting for another match. The Out-tuple extraction process succeeds if it reaches the leaf: an *OK* message goes up to the matching node and the Out-tuple is assigned to the In-tuple, because no other In-tuple could have extracted the same tuple - nor will be able to do it in future times - from the tuple space.

When the matching protocol succeeds at a given level of the sub-tree, the Out-tuple must be extracted also from the upper levels. Even if the protocol makes it impossible its assignment to another In-tuple, being no longer present in the lower levels, its presence in the upper levels of sub-tree could wastes memory resources. The extraction process of the Out-tuple from the upper levels does not influence neither the above protocol nor the client processes and, then, can go on in parallel with the UP-DOWN protocol without inducing any additional access cost. Let us note that every Out-tuple replica contains the information of the node of the lower level from where it was produced.

The *DOWN-UP protocol* (figure 9b) is the dual of the above described one. In this case, a match occurred on a given node is valid only if the Out-tuple extraction process succeed up to the root of the sub-tree. We recall that if one In-tuple joins the tuple space before the matching Out-tuple, the Out-tuple is not replicated up to the root: in this case, the protocol can then immediately validate the match. As in the UP-DOWN protocol, the Out-tuple must be extracted from the whole sub-tree, but the extraction from the lower lever can proceed in parallel with the protocol execution.

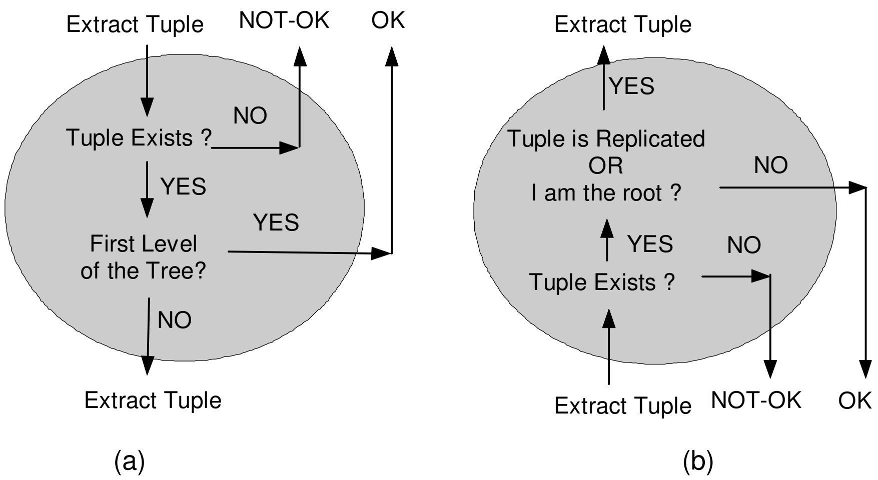

Figure 9. Coherence protocols for a memory node: UP-DOWN (a) and DOWN-UP protocol (b)

6.2 Second coherence condition

The second coherence condition states that one In-tuple must extract only one Out-tuple. In other words, if a replica of the same In-tuple on different nodes matches with different Out-tuples, only one Out-tuple must disappear from the sub-tree. Even in this case, two dual protocols could be identified: an *UP-DOWN* protocol and a *DOWN-UP* one.

In the *UP-DOWN protocol*, once a match has been validated by the first coherence protocol, the In-tuple descends the sub-tree by extracting its replicas from every node of its replication path. If and only if the extraction process succeeds, the unified tuple generated from the match must join the execution node from where the In-tuple was generated. Instead, if the In-tuple replica is not found on a node, another Out-tuple has already matched with a replica at a lower level of the sub-tree and the unified tuple is not returned down to the execution node. In this case, the potentially matching Out-tuple must no longer be extracted: it is "outed" again in the sub-tree and made available to other Ins. In parallel with this protocol, the In-tuple is extracted from the upper level of the tree. To implement this protocol, any In-tuple replica contains the information of the DOWN node that generated it.

The dual scheme (*DOWN-UP protocol*) is less viable. The attempt of extracting the In-tuple up to the root, in fact, forces the protocol to span down to the tree before finally validating the match and returning the unified tuple up to the execution node.

6.3 Costs of the protocols

The cost C of one complete matching process (a successful one) expresses the access time to retrieve information from a tuple space. Given a tuple space with a sub-tree of height *D* (i.e., the number of levels including the root of the sub-tree and the execution nodes) and by considering a match occurring on a memory node at a distance *d* from the execution nodes, these costs are:

- $C_{Match} = \mathrm{O}(d)$ by adopting the UP-DOWN protocol or the DOWN-UP one if the Out-tuple joins the tuple-space before the In-tuple;
- $C_{Match} = \mathrm{O}(D)$ by adopting the DOWN-UP protocol if the In-tuple join the tuple space before the Out-tuple.

In fact, the protocols operate in the sub-tree extending their action along *d* levels in the first case, moving to the whole height of the sub-tree in the latter. Since the height *D* of the sub-tree is proportional to the logarithm of the number of execution nodes that can have access to the given tuple space, and since $d \leq D$, the application cost of the protocols grows slower than the size of the tuple space, making them scalable.

A more complete analysis should take into account concrete resources, i.e., the limits of memory, communication and execution capability of each memory node. When the number of tuples increases, the management of the memory requires a large amount of resources. Because of the hierarchical structure of our model, resource requirements in not homogenous: the higher the level of the tree, the less the physical resources available for the management of the tuple space. This characteristic can introduce a bottleneck in large sized systems.

Let us suppose that the tuple space has to manage, in its whole, *M* matches in the unity of time and that these are caused by *M* Out operations and by *M* corresponding In operations issued from the application processes. In case all tuples belong to a single, global tuple space, about half of the tuples joining the tree is going to be replicated up to the root (as explained in section 3.2). Let us suppose the other half stops their replication path, because of a match, at a distance $d \geq l$ from the execution nodes. By considering that every tuple in a node verifies a match and that the coherence protocols too issue a matching mechanisms to verify the presence of a tuple in one node, the total execution load[1] at the level l of the tree can be expressed as:

$$C_{TOT}(l) = kMC_M + hMC_M p(l)$$

where C_M is the cost, in terms of execution resources, of the matching mechanism, k and h depend on the adopted coherence protocols while $p(l)$ is the probability that one match occurs at the level $d \geq l$ in the tree. At the highest levels of the tree, The load C_{TOT} is shared among a lowering level of memory nodes. Since M is likely to grow with the system size (the bigger the system the more the client processes that access to the tuple space), the higher levels of the tree, in a large system, may be short of resources to manage the C_{TOT} load and represent the bottleneck of the system. Locality in applications can force a high probability $p(l)$ of producing matches at the lower levels of the tree. However, as long as tuples belong to a single global tuple space, the first term in C_{TOT} is constant and represents a load necessarily present at any level of the tree, independently of the application characteristics.

When tuples are partitioned in multiple tuple spaces with a scope limited to given sub-trees, the load at a given level l of the tree is no more caused by all the tuples accessing the tree, but only by those tuples belonging to a tuple space whose sub-tree extends its height at the level l. In this case, the load at a the level l the tree can be expressed as:

$$C_{TOT}(l) = q(l)(kMC_M + hMC_M p(l))$$

where $q(l)$ represents the percentage of tuples replicated at the level l.

The presence of multiple tuple spaces with a well-defined scope makes the constant term in $C_{TOT}(l)$ disappear: the load imposed on the memory nodes - and so the performances achieved by the model - depends on the distribution $q(l)$, thus becoming application dependent.

The $q(l)$ distribution can be tuned in order to grant a fair usage of the memory resources, assumed constant for all memory nodes: we exclude the case of providing more resources to the nodes in the higher levels of the tree. Since the tree structure of our model makes the number of memory resources exponentially diminish with the height of the tree, a solution that grants a balanced distribution of the load among memory nodes must assume that the number of tuples replicated at a given level of the tree exponentially diminish with the height of the tree, i.e., $q(l) \propto exp(-l)$. However, in many practical cases, less restrictive assumptions on the distribution of tuples can grant good performances in terms of memory access time, as in the following.

---

[1] The situation of the communication and memory load is very similar and it will be not analysed

7. The transputer implementation

The presented tuple space model has been implemented on a transputer-based architecture, a Meiko CS-1 with 256 nodes.

7.1 Structure of one memory node

The project of the memory node support to implement the tuple space model for a transputer-based architecture separates mechanisms (tuples storing, extraction and matching) from policies (tuples distribution and replication, coherence protocols) and, then, identifies two layers: the *tuple layer* and the *distribution layer*, respectively (see figure 10). This separation permits to change the distribution policy (and the tuple space model) without influencing the mechanisms and, viceversa, to optimize mechanisms without changing the policy.

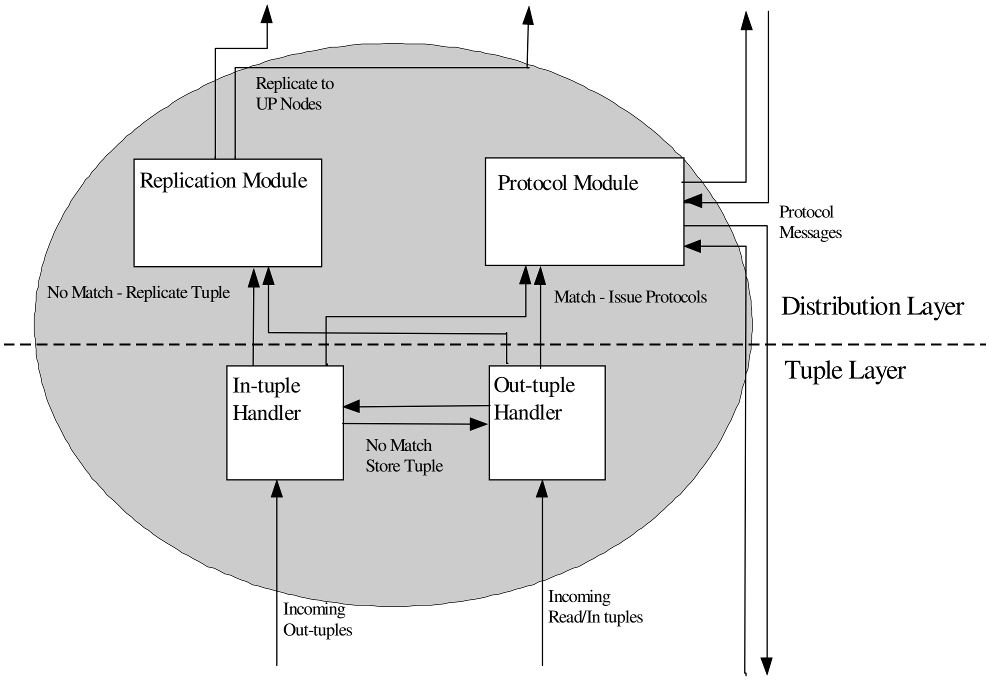

Figure 10. Modular layout of a memory node

At the tuple layer, the node design favors the greatest possible concurrency, in particular for In and Out-tuples processing. Two modules implements these functions:

- the *Out-tuple Handler* is dedicated to Out-tuples storage and extraction;
- the *In-tuple Handler* manages In- (and Read-) tuples, stored in the node waiting for a match.

The arrival of a In-tuple onto a node is managed by the above handlers by following the below steps:

- when one In-tuple arrives in a memory node, the Out-tuple handler checks for possible matches with local Out-tuples;

- if (at least) one match occurs, the matching Out-tuple is extracted and the control is passed to the distribution layer, in charge of the coherence protocols;
- if no match is possible, the Out-tuple handler passes the tuple to the In-tuple handler, to store it locally. The tuple is also sent to the replication module of the distribution layer too, that manages tuple replication depending on its scope.

Any Out-tuples incoming on a node follows a complementary scheme, by asking the In-tuple handler for a possible match before requiring the Out-tuple handler to store them locally.

The replication and the coherence module are in charge of managing tuple replication and of implementing the coherence protocols, respectively.

The *replication module* on one node receives, from both Out- and In-handlers, all tuples that could not match. If the tuple scope extends at the upper level of the tree, the replication module send the tuple to the proper handler up to the tree.

The *protocol module* implements the coherence protocols previously described and it is activated when a match occurs on a node. In addition the protocol module on a node can be activated because of a match in a different node.

7.2 Performance

To evaluate the effectiveness of the implementation, we have allocated several processes on the execution nodes and made them access to the tuple spaces by performing In and Out operations. The performance of the implementation can be measured with the time a process can retrieve information with one In operation, in case the Out tuple is already present in a tuple space. A wide range of possible situations have been experienced with regard to the patterns of the accesses:

- different distributions of the tuple spaces in the system, from global situations with only one global tuple space extending along the whole tree to local situations, where most tuples are confined within a high number of locally limited tuple spaces;
- different traffic of the accesses to the tuple spaces, from situations where the client processes access only rarely to them to situations where processes access almost continuously to tuple spaces.

In addition, to evaluate the scalability of the model, the above experiments have been performed with different system configurations - from a 2 level tree with only 9 execution nodes up to a 4 layer tree with 81 execution nodes.

The distribution of tuples across multiple tuple spaces and the frequency of the accesses greatly influences the retrieval time of information. Figures 11, 12, 13 refer to a system composed of a four layers memory tree (1, 3, 9, 27 nodes, respectively) and of 81 execution nodes: they show the impact of different tuple distributions (global, linear, exponential respectively) and of different traffic of the accesses (from 5% to 50% of the client processes execution time spent in accessing to the tuple space) to the access times.

When only one global tuple space is considered (figure 11), all tuples are replicated up to the root. Then, in high traffic conditions, the highest levels of the tree becomes the bottleneck: the access times to the memory, tends to exponentially grow with the depth at which matches occur, far from being proportional to it.

The access times worsening is less significant when tuples belong to a local tuple space, thus avoiding complete replication. When the assignment of tuples to multiple tuple spaces is such that the number of tuples replicated up to a given level linearly decreases, i.e., when $q(l) \propto l$, the situation is normally fair and becomes unviable only for very high traffic conditions, when the access times significantly increase for those matching occurring at the highest levels of he tree (figure 12).

When tuples are locally distributed so to make the number of tuples replicated at a given level of the tree exponentially decreases with the height of the tree, i.e., $q(l) \propto exp(-l)$, the access times to the memory is directly proportional to the depth of the tree at which a match occurs, without any deviation even for very high traffic conditions (see figure 13), as expected from the previous analysis.

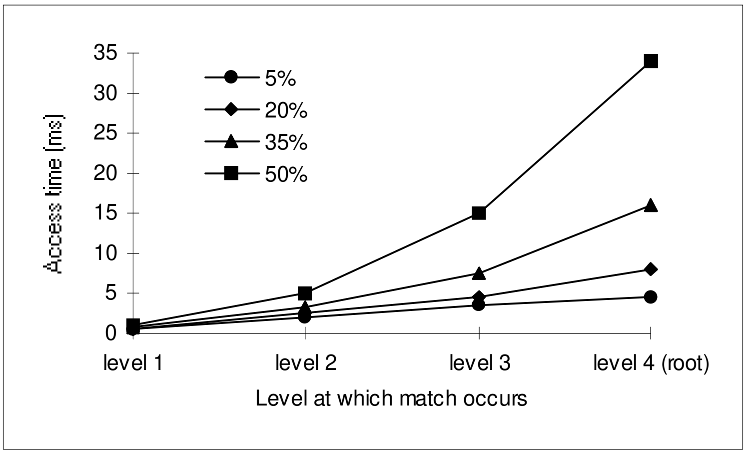

Figure 11. Tuple space access times with global replication of tuples for different traffic of the accesses (from 5 to 50%)

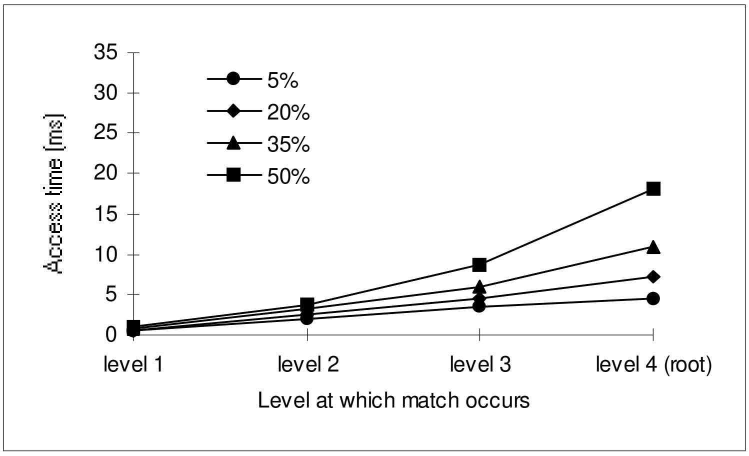

Figure 12. Tuple space access times with linear distribution of tuple scopes tuples for different traffic of the accesses (from 5 to 50%)

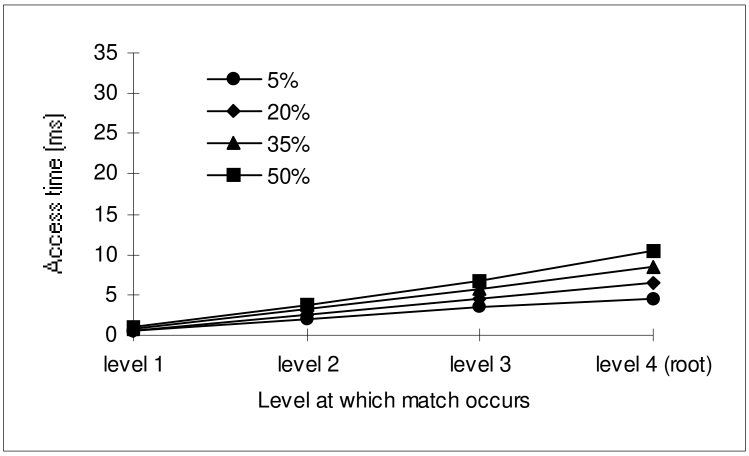

Figure 13. Tuple space access times with exponential distribution of tuple scopes tuples for different traffic of the accesses (from 5 to 50%)

The internal measure of the latency time for the coherence protocol messages on a memory nodes reflects the above behavior. As figure 14 reports that a high percentage of protocol messages experience very high latency time in the case of a single global tuple space. This percentage decreases when the distribution of tuples lowers the load in the highest levels of the tree.

The above experiments have been repeated for differently sized systems to prove the scalability of the model. The result is that the access time to tuple spaces are independent of the global system size; instead, a tuple is accessed in a time that depends only on the locality of the reference, i.e., the scope of the sub-tree it belongs to. If the system enlarges, processes are allowed to access to larger tuple spaces whose accesses are, obviously, subject to higher latency times. However, limiting the number of the accesses to them permits to hold the logarithmic proportionality to the scope of the accesses and avoid the bottlenecking of the highest levels of the tree.

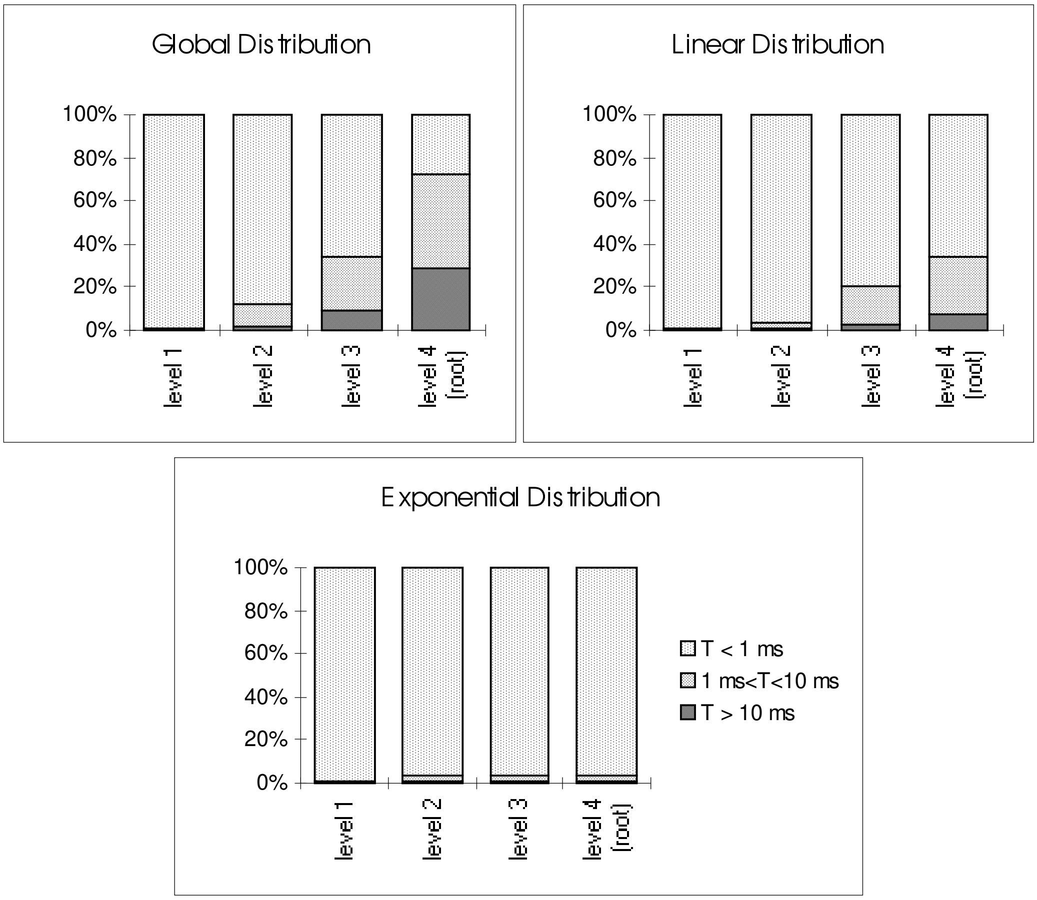

Figure 14. Percentage distribution of latency times (in milliseconds) for the protocol messages on a memory node

As a final remark, it is interesting to verify how the distribution policy of the hierarchical model exploits locality of reference within a tuple space. Figure 15 and 16 reports the percentage of matches that occurs at the different levels of a 3 levels sub-tree and of a 4 levels one, respectively, depending on the probability of a tuple to find a matching tuple of the dual class. For instance, a 10% of probability means that one tuple is likely to match with one tenth of the tuples of the dual class.

When one In-tuple has a high probability to find one matching Out-tuple, a high number of matches occurs at the lower levels of the sub-tree, i.e., close to the execution nodes. As soon as the probability of finding a match decreases, a larger number of tuples need to search in the higher levels of the sub-tree to find a match. However, even with the probability of match is very low, the number of matches at the lowest levels is still dominant. We stem that this situation is local to a single tuple space (i.e., a single sub-tree), and it is independent both on the global size of the system.

The above experiment shows that not only the access time to a tuple space is limited by the depth of the sub-tree that contains it, but most of the accesses to the memory will be even faster because producing matches at the lowest levels of the tree.

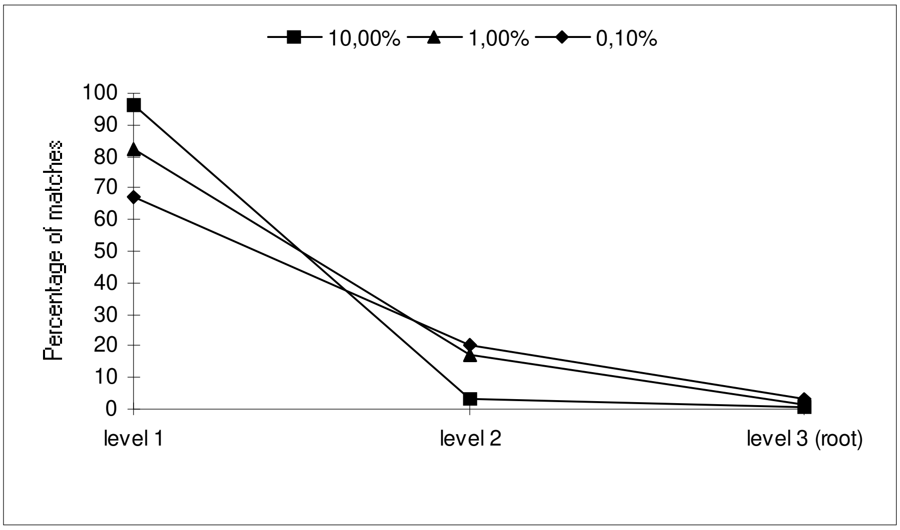

Figure 15. Percentage of matches occurring at the different levels of a 3 level sub-tree depending on the probability of match (10, 1, 0.1 %)

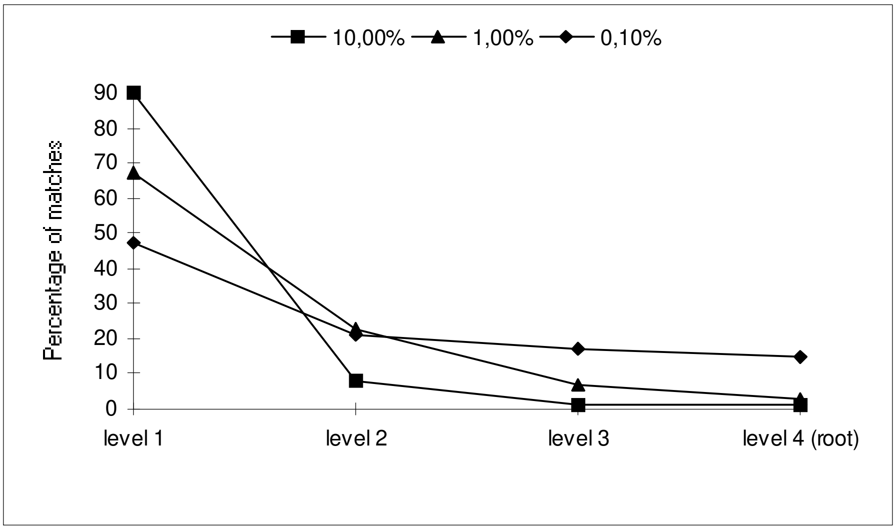

Figure 16. Percentage of matches occurring at the different levels of a 4 level tree depending on the probability of match (10, 1, 0.1 %)

8. Related Works

A large amount of literature deals with tuple space systems and proposals. However, most work focuses on tuple space applications and extensions to the model rather than on its implementation. Thus, the majority of the approaches are centralized or, at least, adopt a hash solution to distribute tuples to the nodes of the system, as the iPSC Linda Kernel [15].

An interesting extension to the hash solution for a transputer-based architecture is described in [16]: being the hash function unable of granting homogeneity in tuple distribution, an overloaded node can distribute part of its tuples to underloaded neighbors to achieve a most balanced situation.

A well known implementation of a full replication strategy is the S/Net's Linda Kernel [17], based on the global Out-tuple replication scheme: the scalability of this solution is limited even in presence of hardware support to broadcast.

A widely known implementation of a partial replication strategy is the Linda Machine [18]: the system is shaped as a 2-D grid of buses: tuples are replicated by row or by column depending on their class. This implementation is based on a specialized hardware support: the nodes of the system are connected by a grid of buses to support broadcast and, so, to keep low the cost of the coherence protocol.

Few proposals focuses on the implementation of multiple tuple space models. One solution that exploit locality in the accesses is adopted by the P^3M abstract machine [19]: dedicated nodes implements separated tuple spaces, any of them directly connected to a given set of application nodes. When the size of each tuple space is little, the P^3M model is scalable: even a simple hash distribution strategy applied on each tuple space allows client processes connected to efficiently access to them. As soon as the size of one tuple space increases, or when processes need to access to one global tuple space, the P^3M solution seems not viable.

A proposal for hierarchically organizing multiple tuple spaces to grant both the access to a local tuple space and to a global one is described in [20]. The hierarchical organization is only at the logical level, while each tuple space - the global one included - is implemented in one node of the target architecture in centralized way.

9. Conclusions and future works

The abstraction power of the tuple space model makes it a very general coordination model for parallel and distributed applications. However, only efficient and scalable implementations of the model can achieve large acceptance for parallel and distributed systems.

The proposed implementation attains this goal by organizing the system in a hierarchical way and by encouraging the presence of multiple tuple spaces.

Despite the promising results of the presented model, several issues have still to be faced at different levels. Firstly, the programming model must help users in structuring their applications to permit the partitioning of the references in multiple tuple spaces. Mapping tools must grant a wise exploitation of the system resources by providing a fair mapping of a block structured application onto the hierarchical system. Finally, different coherence protocols and distribution strategy may exist which improve the presented model.

We feel the hierarchical organization of the presented model can attract new interest in efficient implementations of multiple tuple space models for different architectures: in this direction, we are porting the implementation model to a network of workstations, by maintaining its hierarchical structure at the logical level.

10. Acknowledgments

The authors thank the Research Institute for Parallel Computing Systems (IRSIP) of Naples (I) for the availability of its computing resources. This work has been partially supported by the Italian MURST ex 40%.

References

[1] D. B. Skillicorn, 'Architecture-Independent Parallel Computation', *IEEE Computer*, Vol. 23, No. 12, Dec. 1990, pp. 38-50.

[2] S. Ahuja, N. Carriero, D. Gelernter, 'LINDA and Friends', *IEEE Computer*, Vol. 19, No. 8, Aug. 1986, pp. 26-34.

[3] D. Gelernter, N. Carriero, 'Coordination Languages and Their Significance'; *Communications of the ACM*, Vol. 35, No. 2, Feb. 1992, pp. 96-107.

[4] R. Duncan, 'A Survey of Parallel Computer Architectures', *IEEE Computer*, Vol. 23, No. 2, Feb. 1990, pp. 5-17.

[5] B. Nitzberg, V. Lo, 'Distributed Shared Memory: A Survey of Issues and Algorithms', *IEEE Computer*, Vol. 24, No. 8, Aug. 1991, pp. 52-60.

[6] P. Butcher, A. Wood, M. Atkins, 'Global Synchronisation in Linda', *Concurrency: Practice and Experience*, Vol. 6, No. 6, Sept. 1994, pp. 505-516.

[7] P. Stenstrom, 'A Survey of Cache Coherence Schemes for Multiprocessors', *IEEE Computer*, Vol. 23, No. 6, June 1990, pp. 12-25.

[8] B. Bacci et al., 'P3L: a Structured High-Level Parallel Language and its Structured Support', *Concurrency: Practice and Experience*, Vol. 7, No. 3, May 1995, pp. 225-255.

[9] J. Darlington et al., 'Parallel Skeletons for Structured Compositions', Symposium on Principle and Practice of Parallel Programming, *ACM Sigplan Notices*, Vol. 30, No. 8, Aug. 1995, pp. 19-28.

[10] T. Heywood, S. Ranka, 'A Practical Model of Parallel Computation', *The Journal of Parallel and Distributed Computing*, Vol. 16, No. 3, Nov. 1992, pp. 212-232.

[11] P. Ciancarini, 'Distributed Programming with Logic Tuple Spaces', *New Generation Computing*, Vol. 12, No. 3, 1994, pp. 251-284.

[12] E. Denti et al., "An Extensible Framework for the Development of Coordinated Applications", *Proceedings of Coordination '96*, Lecture Notes in Computer Science, Vol. 1061, Springer Verlag (D), 1996, pp. 305-320.

[13] D. Gelernter, 'Multiple Tuple Spaces in Linda', *Proceedings of PARLE '89*, Springer-Verlag (D), 1989, pp. 20-27.

[14] D. Kranz et al, 'Integrating Message-Passing and Shared-Memory: Early Experience', Symposium on Principles and Practice of Parallel Programming, *ACM Sigplan Notices*, Vol. 28, No. 7, July 1993, pp. 54-63.

References of the related works section

[15] R. Bjornson et al., 'Linda, the Portable Parallel', *Technical Report No. RR-520 - Department of Computer Science*, Yale University, Jan. 1988. Available at *www.cs.yale.edu.*

[16] M. J. Manthey et al., 'Ariadne - a Linda Kernel for Transputer Networks', *Master's Thesis - Department of Mathematics & Computer Science*, Aalborg University Center, 1990. Available at *www.cs.indiana.edu:800.*

[17] N.Carriero, D.Gelernter, 'The S/Net's LINDA Kernel', *ACM Transactions on Computer Systems*, Vol. 4, No. 2, May 1986, pp. 110-129.

[18] S.Ahuja et al, 'Matching Language and Hardware for Parallel Computation in the LINDA Machine', *IEEE Transactions on Computers*, Vol. 37, No. 8, Aug. 1988, pp. 921-929.

[19] F. Baiardi, M. Jazayeri, 'P^3M: a Virtual Machine Approach to Massively Parallel Computations', *International Conference on Parallel Processing*, IEEE CS press, Los Alamitos (CA), 1993, pp. 340-344.

[20] G. Matos, J. Purtilo, 'Reconfiguration of Hierarchical Tuple-Spaces: Experiments with Linda-Polylith', *Technical Report No. CS-TR-31-53 - Department of Computer Science*, University of Maryland, Oct. 1993. Available at *www.cs.indiana.edu:800.*