A Scalable Tuple Space Model for Structured Parallel Programming

Antonio Corradi, Franco Zambonelli

Dipartimento di Elettronica Informatica e Sistemistica - Università di Bologna
2, Viale Risorgimento - 40136 Bologna - ITALY
E-mail: {acorradi, fzambonelli}@deis.unibo.it

Letizia Leonardi

Dipartimento di Scienze dell'Ingegneria - Università di Modena
214, Via Campi - 41100 Modena - ITALY
E-mail: leonardi@dsi.unimo.it

Abstract

The paper proposes and analyses a scalable model of an associative distributed shared memory for massively parallel architectures. The proposed model is hierarchical and fits the modern style of structured parallel programming. If parallel applications are composed of a set of modules with a well-defined scope of interaction, the proposed model can induce a memory access latency time that only logarithmically increases with the number of nodes. Experimental results show the effectiveness of the model with a transputer-based implementation

1. Introduction

The lack of any globally shared resources makes massively parallel architectures intrinsically scalable. However, the need for a global space of interaction is difficult to be substituted for many reasons. In particular:

- global computational models are intrinsically simpler than local ones;
- even applications based on local computation models need shared resources, for instance to create a unique naming system.

The above problems can be solved by mapping onto the physical distributed resources of a parallel architecture the abstraction of a globally shared space, a distributed shared memory [1].

The problem of an abstract global interaction space is, again, non scalability: a distributed shared memory, in particular, can produce unacceptable latency times if the number of processors among which it is distributed is too large.

In this paper, we follow the idea that a scalable model for a distributed shared memory can be implemented by restricting the programming model to a **structured** one. In general, parallel applications follow predetermined patterns of interaction for their activities. These patterns identify, within an application, a well-defined spatial scope for references, in the same way as local variables have a restricted scope in sequential programming.

The above scenario leads us to a hierarchical model for a distributed shared memory, by allowing data to be accessed, but only within their scope and not globally. A shared memory is distributed across a tree of memory nodes, whose leaves are connected to a set of execution nodes. Shared data, tuples in our proposal, are replicated in the tree with a given depth, depending on their scope. This avoids bottlenecks in the hierarchy and achieves a data access time proportional to the **degree of locality** of the access itself. In any case, the memory access latency time is logarithmically bounded with respect to the system size. In the shared memory we distinguish between two roles: memory nodes are servers to implement the shared memory, execution nodes host the client applications. Because of this distinction, there is no overhead of the shared memory implementation on the application.

The paper is organised as follows. Section 2 shows the roles of a shared memory abstraction and its requirements, while the problem of implementing distributed tuple spaces is discussed in section 3. The proposed hierarchical model is described in section 4 and analysed in section 5. Section 6 and 7 describe, respectively, an extension to the model and how it meets the modern structured parallel programming style.

2. The shared memory abstraction

Shared memory is a well-accepted concept because of many years of experience in sequential and shared-memory supercomputers programming. Moreover, abstract computational models based on shared memory abstract architectures, such as the **PRAM** one [2],

produced and diffused know-how in parallel algorithms. In massively parallel programming, the shared memory abstraction plays new roles. It can behave as:

communication system. Communication between distributed processes can operate on the basis of the shared memory, making easy the implementation of several communication modes.

store for **monitoring** and/or **debugging** information. In a local environment, it could be difficult and even expensive to obtain global information about the behaviour of the system and of the applications, although the information is essential.

way to access **I/O** devices. The mapping of I/O in shared memory solves the problem of sharing external devices among distributed activities;

virtual memory: shared memory permits a process to access the amount of needed memory.

Shared memory and message passing computational models are not mutually excluding each other [3]. As an example, object-oriented parallel programming environments, whose computational model defines a message passing scenario without any shared resource, can greatly benefit of the presence of shared memory [4].

However, the physical implementation of a shared memory presents problems [5]: it introduces a centralisation point that limits the system scalability, becomes the bottleneck when the system size increases. A promising solution, not to avoid the use of a global interaction space, is represented by **distributed shared memory**, where the shared memory is implemented as a software abstraction onto distributed memory architectures.

In distributed shared memories, the logical shared memory space is distributed among different nodes, each one with its local memory and autonomous execution capability. The information in the memory are available, wherever they are allocated, to every node of the system and in a transparent way. **Data replication** can make data access time faster, by exploiting locality in a way similar to cache based systems [6]. This introduces the problem of **coherence** among different replicas and the necessity of protocols to guarantee it [7].

A crucial point when implementing a model is to choose the right **abstraction level**, in particular to choose the elementary unity of access of the memory. On the one hand, the level of variables (whose names permit them to be "read" and "written") is too low, and it can be confusing in a distributed environment where names do not necessarily correspond to a defined physical address. On the other hand, shared objects (that also define the interface for accessing them) represent too high a level and can lead to inefficiencies in implementation.

We choose an intermediate-level, the **tuple space** memory model [8]. The unity of access to the memory is the tuple, an ordered set of typed fields (defining the tuple structure), having a defined (actual) value or a not defined (formal) one. Two tuples are said to "match" if they have the same structure and the same value of corresponding actual fields. The operation allowed on the tuples are **Out**, **In** and **Read**. The Out operation simply stores a tuple in the tuple space. The Read operation searches in the tuple space for a tuple matching with a specified tuple. The In operation is similar to the Read operation except that it removes the matching tuple from the tuple space that is no longer available to other Read or In operations. Both In and Read operations can block for the calling process, if a match not found. In case of multiple matches (an In call that matches with different tuples or an Out call that matches with different In), a choice is made non-deterministically.

We emphasise the duality of In and Read operations with the Out one. One In or a Read produces a "request tuple", stored in tuple space, and successive Out calls must search for a match with already stored "request tuples". For sake of simplicity, in the following, we will indicate the tuples produced with In, Read, Out operations as, respectively, In-tuples, Read-tuples and Out-tuples.

We think the tuple space is not also an intermediate abstraction level but presents other advantages, being a well-known and defined model and guaranteeing both consistency (tuples cannot be modified *in loco*) and fairness (by managing conflicts in a non-deterministic way).

3. Distributed tuple spaces

The implementation of a distributed tuple space requires the definition of policies (**distribution policies**) to distribute and eventually replicate tuples onto different nodes. In the case of reconfigurable architectures, it is also necessary to define the interconnection topology that best suits from the adopted distribution policy. We emphasise the separation between distribution policies and matching mechanisms: the matching search occurs locally to a node, on the basis of the tuples locally assigned to that node, without any knowledge of the global distribution policy and of the system interconnection topology. Since the implementation of efficient matching mechanisms is a database management problem, we mostly concentrate on the distribution policy and on the system topology.

The simplest **distribution policy** (with no data replication and system independent) is based on a **hash**

solution. A tuple is allocated in one of the nodes of the system on the basis of the value of a hash function applied on one of its fields, assumed as "key" (it must always have an actual value). The problem with this solution is the difficulty in achieving a homogeneous distribution of tuples in the system, especially when the number of nodes is high with respect to the number of tuples keys. Moreover, the hashing solution cannot exploit locality.

A different approach is based on a replication technique: the replication space of the Out tuples should form a **non-null intersection** with the replication space of the tuple requests. A trivial solution that follows this approach is one where the Out-tuples are stored only on one node while the In and Read-tuples search globally for the match and are eventually replicated on every node of the system [9]. At the other end of the spectrum, the dual solution provides a full replication of the Out-tuples and a local search and a non-replicated storage of the In and Read-tuples. Both solutions, even if simple and system-independent, are not effective for large system. In fact, they are based on a global replication strategy that is not scalable and requires a global coherence protocol.

Hybrid solutions, where both tuples and requests are replicated with a limited degree, seem to provide better performance. However, their implementation and effectiveness are strictly related to the architecture topology. The replication strategy must take into account the structure of the communication system, that influence the implementation costs both of the replication strategy and of the coherence protocols. An interesting example is the Linda Machine [10]. The system is configures a 2-D mesh topology: a node is identified by the row and column numbers of its position in the mesh. Whenever a tuple enters the tuple space via an Out operation, it is replicated along the whole row of the mesh of the source node. Whenever a request enters the tuple space it is replicated along the whole column. The cost of the replication policy and of the coherence protocol increases proportionally with the square root of the system size, and makes the Linda Machine quite-well scalable.

When choosing the system interconnection topology to implement a distributed tuple space, it is possible to divide the system resources into execution nodes, devoted to the applications, and memory nodes, devoted to the implementation of the distributed shared memory. This allows to define independently the interconnections of different parts: the execution nodes interconnection network, the memory nodes interconnection network and the connection between the memory nodes and the execution nodes. This solution is suitable in case one decides of maintaining both the message passing and the shared-memory paradigm [3]: the execution nodes can interconnect each other in application specific topologies while the memory nodes can interconnect to fit the implemented memory distribution strategy. Moreover, the lack of contention between the memory activities and the application execution avoids the shared memory overhead problem and achieves more predictable performances. This solution is adopted, for example, in the **P^{3}M** abstract machine for massively parallel computation [11].

4. The tree structured tuple space model

Our tuple space model follows a hybrid scheme for tuple replication, as seen in section 3, where the execution nodes are separated from the memory nodes. The main point of the project is to define a scalable scheme of tuple replication, growing slowly in cost with the system size.

The system is configured in a tree fashion (figure 1), whose leaves are the execution nodes. Each execution node is connected to a memory node of the immediately higher level. Each memory node is connected, in its turn, with another memory node of the higher level and some nodes (L-1, if L is the connectivity degree of a node) of the lower level, that can be either memory nodes or execution nodes. The root memory node represents the highest level of the tree and has no higher memory nodes to connect to. Such solution looks like the one adopted in hierarchical caching model for distributed architectures [12].

The model can be implemented in every architecture (it requires a very limited connection degree, at least 3), and minimises the number of nodes dedicated to the implementation of the tuple space. We emphasise that the execution nodes can be freely interconnected to each other, without influencing at all the tree structure.

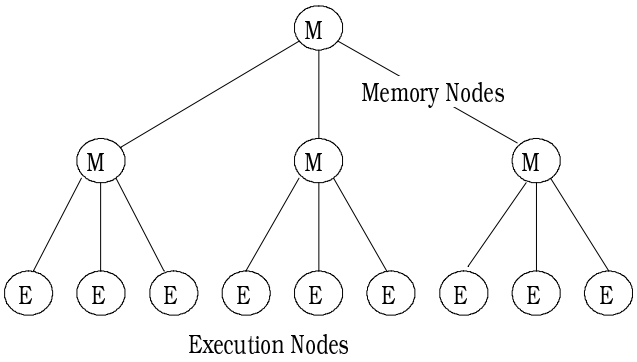

Figure 1. The tree distributed shared memory model

4.1. Tuple distribution strategy

The tuple distribution strategy adopts a hybrid replication scheme that follows the tree. Tuples (both In-tuples, Read-tuples and Out-tuples) are replicated along the whole path that starts from the execution node that

generated it and goes up to the root. It is clear that such a replication scheme presents the property of non-null intersection. Every tuple share with every other tuple the nodes that start from the lowest common parent in the tree and arrive to the root. Any two tuples share at least the root. Since the height of the tree grows logarithmically with the system size, the level of replication (and so the cost of the coherence protocol) also follows this law, making the model scalable.

When a tuple enters the tuple space, it climbs the tree by replicating itself in every node it crosses, after verifying the chances of matching at each node. Replication stops if a match is found (figure 2): this optimises the replication degree since, in a couple of matching tuples, only one of them (the first that has joined the tuple space, is replicated up to the root while the replication of the other is blocked at the first node of the intersecting replication path. Once the tuple is found, the Out-tuple must go down the tree in direction of the execution node that produced the matching In-tuple.

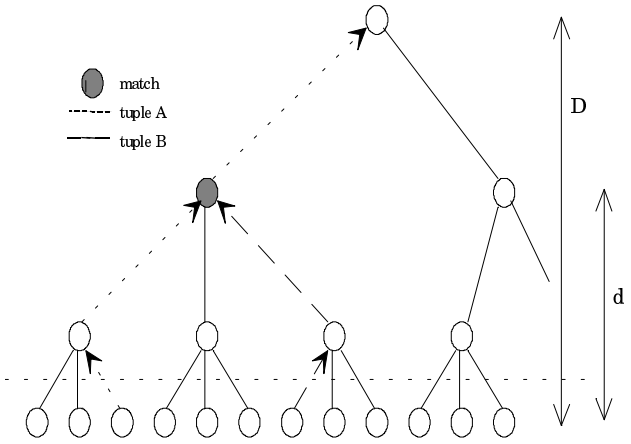

Figure 2. Replication and match

the tuple A enters the tuple space and it is replicated up to the root; when the tuple B enters the tuple space match with A and stops to replicate itself

4.2. Matching and coherence protocols

Data replication introduces the problem of coherence and the need of granting protocols. In a tuple space, we argue that coherence can be guaranteed if:

1) any given Out-tuple is extracted by only one In operation (note: this is meaningless in the case of the Read operations);
2) any given In operation extracts only one Out-tuple (the same for Read operations);

The two relationships are independent and so are the protocols to guarantee them: one can grant one of them without granting the other and, viceversa, the choice of one coherence protocol does not influence the another; only efficiency can be influenced.

Two protocols can be adopted to obtain the **first coherence condition**, we call it respectively UP-DOWN and DOWN-UP.

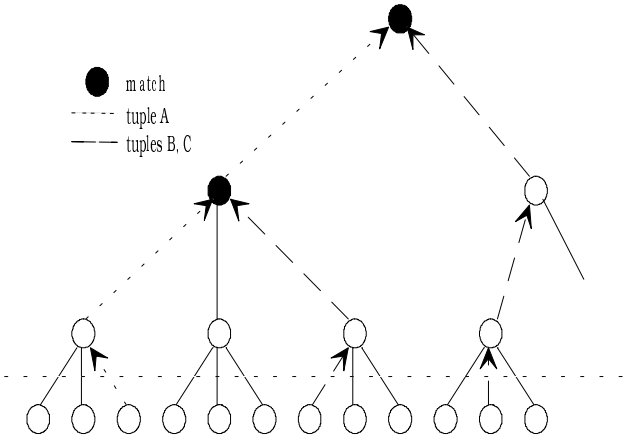

Figure 3. Multiple matches

two tuples B and C match, on different nodes, with replicas of the same tuple A

Let us suppose that in a node one In-tuple finds a match with one Out-tuple. We recall that both tuples are stored in all lower levels of the tree following the path down to the respective leaves that generated them. After the match, the Out-tuple should be assigned to the In-tuple and extracted from the tuple space after verifying that no other In-tuple has already extracted a different replica of the same tuple. The **UP-DOWN protocol** (figure 4), before assigning an Out-tuple to a given In-tuple, explores the tree from the node of the current match down to the leaf. The Out-tuple is extracted from every node in the path of generation. If the Out-tuple is not found in a node, this means it has been already extracted by another matching tuple-In. In this case, a NOT-OK message is returned to the node where the match has occured and the In-tuple is stored and replicated up in the tree, waiting for another match. If and only if the Out-tuple extraction process proceeds down to the leaf, the match succeeds. In this case, an OK message is returned to the node of the match and the Out-tuple is assigned to the matching In-tuple, because no other In-tuple can extract the same tuple down to the leaf level.

When the matching protocol succeeds at a given level of the tree, the Out-tuple must be necessarily extracted also from the upper levels up to the root. Even if this tuple cannot be assigned to other In-tuples, its presence in the tree could cause a waste of memory resources and Read-tuples to match with an already extracted tuple. The Out-tuple process extraction from the upper levels of the

tree, however, can be executed in parallel with the UP-DOWN protocol and does not cause any additional access costs. Let us note that the protocol works correctly if every Out-tuple replicated on a node contains the information of the DOWN node where it comes from.

The **DOWN-UP protocol** (figure 5) is the dual of the above described one. In this case, a match occured on a given node is validated only if the Out-tuple extraction process succeeds up to the root. As in the above protocol, the Out-tuple must be extracted from the whole tree: again, the extraction from the lower levels can proceed in parallel with the protocol execution.

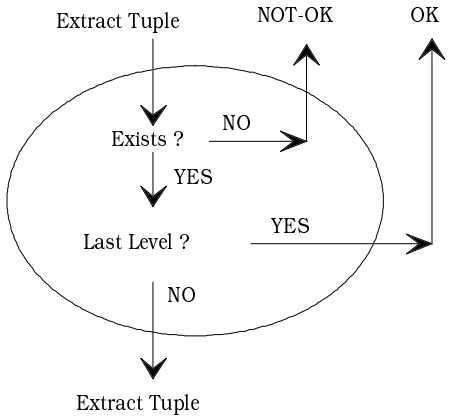

Figure 4. The UP-DOWN protocol on a memory node

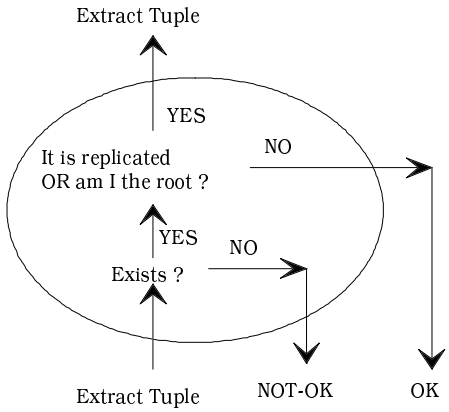

Figure 5. The DOWN-UP protocol on a memory node

The **second coherence condition** states that an In-tuple must extract only one Out-tuple. In other words, if replicas of the same In-tuple on different nodes match with different Out-tuples, only one of these Out-tuples gets extracted from the tree. This coherence condition must be honoured by Read-tuples too: one must not permit a Read-tuple to match with two or more replicas of the same Out-tuple. The difference is that no incoherence can result because a Read does not modify memory.

Even in this case, two dual protocols could be possible: 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 climbs down the tree trying to extract itself from every node of its replication path. If this process succeeds, the In-tuple joinsthe execution node that generated it (carrying along the information obtained with the match including the Out-tuple itself). If the In-tuple replica is no longer present on a node, a match has already occurred of the same In-tuple with another Out-tuple. In this case, the carried Out-tuple should not be extracted: it is "outed" again in the tree and made available to other In-tuples. In parallel with this protocol, In-tuple replicas are extracted from the upper levels of the tree. Therefore, as before, an In-tuple on a node must contain the information of its DOWN node.

The dual scheme (DOWN-UP protocol) is not viable. Trying to extract the In-tuple up to the root, in fact, goes against the natural direction for the matched In-tuple, whose final goal is to return information down to the execution node that has generated it.

5. Evaluation of the model

The cost C of complete matching process (a successful one), both in case of an In-Out match and of a Read-Out one, can determine the memory access time. If D is the height of the tree, i.e., the number of levels including the root and the execution nodes, and d (see figure 1) is the distance between the execution nodes and the memory node at which the match is occurred, then:

$C_{Read} = O(d^2)$

$C_{In} = O(d^2)$ by adopting the UP-DOWN and by adopting the DOWN-UP one in the case the Out-tuple joins the tuple-space before the In-tuple;

$C_{In} = O(D^2)$ by adopting the DOWN-UP protocol in the case the In-tuple joins the tuple space before the Out-tuple.

In fact, the protocols act in the direction of the tree, extending their actions along d levels in the first two cases, along the whole tree height in the latter. In every memory node of action of the protocol, the matching mechanisms cost is proportional to the distance between the execution nodes and the memory node itself. It is important to point out that, since the height D of the tree is approximately the logarithm of the number of nodes of the system, and since d<=D, the application cost of the protocols increases slowly with the system size (and it is scalable).

The above presented evaluation implicitely assumes an infinite memory and execution capacity for each memory node. However, when the number of tuples increases, the memory management can require a high number of memory and execution resources, making the memory access time untolerable. Let us suppose that, in the time unit, the memory has to manage M matches and, for sake of simplicity, M In operations and M Out operations. Half of the tuples joining the tree, either In- or Out-tuples, are replicated up to the root: in fact, they do not find a partner tuple to match with. The other tuples stop replication when they find a match. Let us call p(x) the probability that one tuple finds a match at a distance from the execution nodes greater or equal to x. Because every tuple that touches a node triggers a matching mechanism, and by considering also the matching mechanisms triggered by the coherence protocols, the total execution load[1] at a given level x of the tree is:

$$C_{TOT}(x) = MC(\frac{1}{2} + k) + MC(\frac{1}{2} + h)p(x) \qquad \text{(i)}$$

where C is the cost, in terms of execution resources, of the matching mechanism, k and h are constant terms depending on the adopted coherence protocol. In the load $C_{TOT}(x)$ the dominant term is the first constant item, while the second item decreases with the increasing of x. For a given level, the load is split into several nodes whose number decreases while x increases: the higher the level in the tree, the lower the number of memory nodes. By considering that M increases with the system size (the bigger the system the bigger the number of processes that access the memory producing matches), it is clear that the highest levels of the tree could not be able to manage them. A high degree of locality in applications can force a high probability of producing matches at the lowest levels of the tree: this diminishes the cost $C_{TOT}(x)$ at the highest level of the tree, by reducing the probability factor, i.e., the second item of the (i). However, in $C_{TOT}(x)$, the constant factor (independent on the height where the match occurs) still loads the highest levels of the tree even in presence of high locality in applications: the root can still be the bottleneck in large sized system.

Experimental results in a T800 transputer-based implementation of the model confirm the above considerations. The time needed for an In-tuple to retrieve information from the tuple-space by matching with an already present Out-tuple, defines the memory access time. To evaluate it, we have simulated the presence, on the execution nodes, of application processes accessing the tuple space. Several patterns of memory accesses have been simulated and with a different frequency of the accesses to the shared memory.

Figure 6 reports the average time - in case of a tree composed of 40 nodes distributed in four layers (1, 3, 9, 27 nodes, respectively) and 81 execution nodes to access to the shared memory - depending on the level of the tree where the match occurs. In case application processes are very CPU-bound, the memory access time is proportional to the level of the tree where the match occurs. However, when application processes are more memory-bound, the higher levels of the tree tend to become overloaded. The access time increases exponentially for those matches that occur at the higher level of the tree.

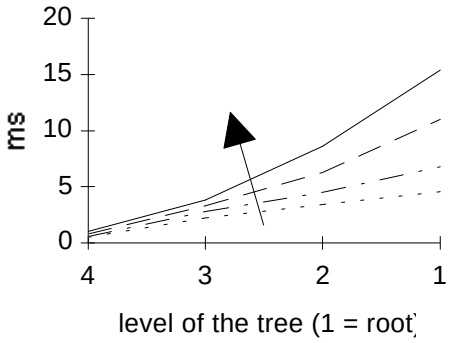

Figure 6. Tuple space access times

(arrow direction = increasing load)

To make our model viable in large systems, still exploiting potential scalability, it is necessary to avoid the growth of the ratio between the load at the higher levels of the tree and the number of nodes that must support.

6. Extensions to the model

The previous section has identified the bottleneck of the presented model: the higher levels of the tree must use a limited number of resources for a load that increases with the system size. One could ignore this problem: even if higher levels are overloaded, lower levels can always offer good performance, allowing by efficiently exploiting locality of the applications. However, the matches occurring at the higher level of the tree - representing interactions between very far nodes - produce higher and higher latency times even if they are a few. Then, two solutions arise to limit the access time for those high levels matches:

- ### to modify the model in order to allow the load at the higher levels of the tree to be shared among the needed number of resources;
- ### to eliminate the costant item in C_{TOT}, by constraining the tuple replication level.

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

6.1. Releasing the tree structure

To subdivide the load of the higher levels of the tree among a larger number of resources, one can think to release the physical structure of the tree while maintaining its logical one. One can assign to each logical node of the higher levels of the tree more physical nodes, in a number proportional to the load it must bear. This solution, however, cannot permit a logarithmic increase in the memory access time. The increased number of nodes for a single logical memory node makes the cost of both matching mechanisms and coherence protocols increase: in fact, they must be applied to a distributed set of items. Moreover, the implementation would require too many memory nodes with respect to the number of execution nodes.

An alternate solution can be thought. Not only the physical structure of the tree is released at the higher levels, but also the logical one. The higher levels of the tree, no longer able to manage their load, are substituted with a "flat" structure, such as a ring or a mesh; this solution can become efficient even for large systems. However, asymptotically, the behaviour of the flat structure tends to dominate the memory behaviour, with a growth of memory access time, linear in case of a ring, square rooted in the case of a mesh. In any case, the logarithmic trend cannot be produced any longer.

6.2. Constraining tuples replication

The best solution to overcome the bottleneck of the model and, at the same time, to maintain a logarithmic growth in the memory access time, is to eliminate the constant factor in C_{TOT}. The only way to achieve that is to avoid the replication of tuples up the root. In general, tuples will be replicated only in a given limited sub-tree.

It is clear that if a tuple is not replicated up to the root, it will not be accessible by every execution node. In fact, the replication space of tuples no longer respect the property of non-null intersection (see section 3). A tuple will not be globally accessible but could be accesses only by the execution nodes in the sub-tree whose root is the highest level memory node onto which the tuple is replicated (figure 7): a tuple acquires a limited **visibility scope**.

In other words, we can state that tuples are not partially replicated in the memory but fully replicated in a **sub-memory**, identified by a sub-tree contained in the main shared memory. Each node of the memory identifies a sub-tree it is the root of, and it identifies also a tree-structured sub-memory with the same property w.r.t. the main one. Tuples whose scope is a given sub-tree can be considered not to be stored in the main memory but only to be stored in the sub-memory identified by the sub-tree.

We claim coherence protocols need not to be modified, but they will be applied by considering the sub-memory scheme. This solution to the bottleneck problem is viable only if the parallel programming methodology is very structured.

Next section analyses the parallel programming methodology that permits tuples to constrain replication at a given level.

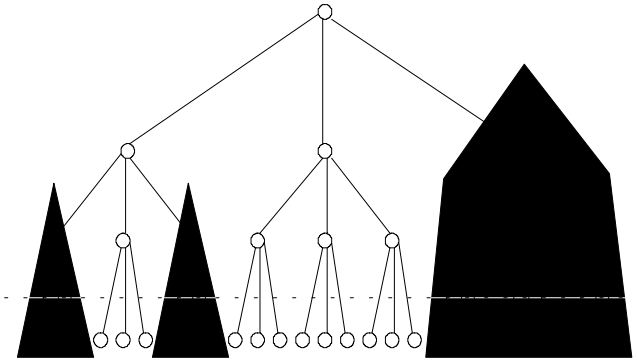

Figure 7. Sub-trees identifying different visiblity scopes and submemories

7. Structured Parallel Programming

The restriction of the replication degree of tuples in our memory model does not represent a limit if the adopted programming methodology is structured. A **structured approach** to parallel programming starts from the idea that most parallel applications follow a few predetermined patterns of execution (such as pipelines, forms, trees) for the parallel activities and their communications [13, 14, 15]. In other words, the components of a parallel application normally do not interact arbitrarily but follow regular basic communication patterns.

Most large applications can be composed starting from these basic structures. The result is a structured application and the scope of the accesses, both to other processes and to shared variables, is confined at the internal of the parallel pattern that uses it, in a way similar to the local variables of a sequential procedure. The temporal locality of references typical of structured sequential programming is extended to spatial dimension in the case of structured parallel programming.

In our memory model, if applications are structured, the reduction of the visibility scope of the tuples is not so strict a limit. Once the scope of a tuple within an application has been identified, the replication path of this tuple can be stopped at the corresponding tree level. In this way, only tuples that effectively need to be globally accessed from the whole system are replicated up to the

root: only these tuples are allowed to load the higher levels of the tree.

Figure 8 reports the memory access time - in the same system configuration described in section 5 - when using the above described restricted model. In particular, we report the access time in the same traffic condition of the experiments reported in section 5 by varying the visibility scope of the tuples in such a way that balance the load shared among the memory nodes of any level. In this case, as figure 8 reports, the obtained access times are linearly dependent on the depth of the tree where the match occurs and, consequently, logarithmically dependent on the system size.

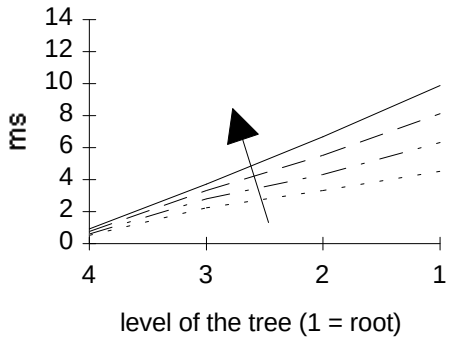

Figure 8. Restricted model access times

(arrow direction = increasing load)

8. Conclusions and Future Work

The paper has presented a model for the implementation of a distributed shared memory for massively parallel architectures. The hierarchical structure of the model allows a logarithmic degree of scalability and the exploitation of the locality present in applications.

The paper evaluates the model and identifies the limitations that could clash with scalability. However, the increasing use of structured parallel programming can produce applications whose properties can overcome these limitations and make our model not only very powerful but very effective.

Future work is in the direction of testing our model with real application in order to verify its effectiveness "on the field".

References

1. B.Nitzberg, V.Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", IEEE Computer, Vol. 24, No. 8, Aug. 1991.
2. D.B.Skillicorn, "Architecture-Independent Parallel Computation", IEEE Computer, Dec. 1990.
3. D. Kranz et alii, "Integrating Message-Passing and Shared-Memory: Early Experience", ACM Sigplan Notices, Vol. 28, No. 7, July 1993.
4. S. Matsuoka, S. Kawai, "Using Tuple Space Communication in Distributed Object-Oriented Languages", OOPSLA '88 Conference Proceedings, San Diego (CA), Sept. 1988.
5. R.Duncan, "A Survey of Parallel Computer Architectures", IEEE Computer, Feb. 1990.
6. P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors", IEEE Computer, June 1990.
7. M. Stumm, S. Zhou, "Algorithms Implementing Distributed Shared Memory", IEEE Computer, May 1990.
8. S. Ahuja, N.Carriero, D.Gelernter, "LINDA and Friends", IEEE Computer, Aug. 1986.
9. N.Carriero, D.Gelernter, "The S/Nets's LINDA Kernel", ACM Transactions on Computer Systems, Vol. 7, No. 7, July 1985.
10. S.Ahuja et alii, "Matching Language and Hardware for Parallel Computation in the LINDA Machine", IEEE Transaction on Computers, Vol. 37, No. 8, Aug. 1988.
11. F. Baiardi et alii, "P3M: an Abstract Architecture for Massively Parallel Machines", Workshop on Abstract Machine Models for Highly Parallel Computing, Leeds (UK), April 1991.
12. E. Hagersten, A. Landin, S. Haridi, "DDM - A Cache-Only Memory Architecture", IEEE Computer, Sep. 1992.
13. J. Darlington et alii, "Parallel Programming Using Skeleton Functions", Proceedings of PARLE '93, Munich (D), June 1993.
14. A. Corradi, L. Leonardi and F. Zambonelli, "How to Structure Parallel Applications: Local Nested Aggregates", Proceedings of the Joint Modular Languages Conference, Ulm (D), Sept. 1994.
15. B. Bacci et alii, "A Structured High-Level Parallel Language and its Structured Support", Concurrency: Practice and Experience, Vol. 7, No. 3, May 1995.