A Proposal for a Parallel Programming Support for Multi-LAN platforms

Luciana Arantes[1]*, Bertil Folliot[1], Liria M. Sato[2], Pierre Sens[1]

[1] LIP6 Laboratory,
University of PARIS VI, Paris, France
email: [Luciana.Arantes, Bertil.Folliot, Pierre.Sens]@lip6.fr
[2] Escola Politécnica,
University of Sao Paulo, Sao Paulo, Brazil
email:liria@pcs.usp.br

Abstract—
In the first part of this article, we present our proposal for a distributed shared memory system (DSM) for an interconnection of local-area networks (LANs). Our multi-LAN DSM will be composed of a set of per LAN lazy release consistency (LRC) memory model DSM systems. For controlling shared-memory updates, the LRC protocol of each DSM will use the barrier-lock logical clocks, instead of the traditional per processor vector ones. This replacement provides modularity and scalability to some extent. The other enhancements to be added to the protocol aim the reduction of the number of messages and the volume of data exchanged between LANs for the sake of applications' performance. Data pre-fetching, simulation of a LAN-level cache and hierarchical execution of barriers are some of the strategies to be adopted. In the second part of the paper, we discuss the advantages of using the CPAR language for the development of parallel applications which run on top of shared-memory hierarchical platforms such as multi-LAN DSMs.

Keywords— **Distributed shared memory, multi-LAN platform, parallel programming language, hierarchical approach, modularity, scalability, logical clocks.**

I. INTRODUCTION

Distributed shared memory (DSM) systems provide shared memory abstraction on top of physically distributed memory systems. They simplify the development of an application since the programmer can use a shared-memory style. Distribution, placement and replication of data are completely transparent for the application.

However, the majority of the DSM systems found in literature have been implemented for local area networks (LANs) or symmetrical multiprocessors (SMPs). Our proposal is, then, to span a DSM over an interconnection of LANs. Such a platform is extremely interesting as it provides more computing power, larger amount of physical memory, and the possibility of profiting from the idleness of underutilized remote machines. On the other hand, the cost of communication between processors of different LANs is usually the major bottleneck for the feasibility of a multi-LAN DSM. High latency and, quite often, low bandwidth of inter-LAN links can strongly limit the performance of applications on top of a multi-LAN DSM.

Our proposal for a multi-LAN DSM is based on the **lazy release consistency (LRC)** memory model. LRC protocol was originally defined by TreadMarks [AMZ 96] [KEL 95], a state-of-art DSM, which presents quite a satisfactory performance on top of a LAN. In our multi-LAN solution, every LAN executes a copy of TreadMarks, adapted for a multi-LAN platform. As it is costly to request a data from a remote LAN, the DSM module of a LAN profits from the updated data already available within its local LAN, asking to a remote DSM only the ones that any of the processors of its local LAN does not have. Barriers execute in a tree-structure way in order to reduce the volume of data and the number of messages transferred between LANs. On the other hand, when a remote request is really necessary, some heuristics can be used for data pre-fetching as an attempt to minimize the number of future remote LAN accesses.

Modularity is also an important feature for a multi-LAN DSM. Thus, the multi-LAN LRC protocol will use the *barrier-lock* clocks [ARA 99a] for controlling the causality of shared memory accesses, instead of the traditional per processor ones [MAT 89] [FID 91]. The *barrier-lock* clock is a logical clock whose timestamps are based on LRC synchronization operations (barriers and locks). As its size does not depend on the number of nodes of the DSM, it is extremely adequate for interconnecting DSM systems.

Shared memory paradigm simplifies the programming of parallel applications to some extent but not completely. We believe that a language that turns less complex the programming of a DSM application itself can be quite useful. In the case of a multi-LAN DSM, it is particularly interesting a language that offers a programming model that provides constructions for exploiting the hierarchy of the architecture. The **CPAR** language [SAT 94] [SAT 95] has such features as it provides construction for task execution and memory organization in an hierarchical way. Thus, the second part of this paper discusses how we are going to adapt the CPAR processing support (**CPAR-DSM**) [ARA 98] to run on top of a

* PhD. scholar from CAPES (Brazil)

LRC multi-LAN DSM.

This paper is organized as follows. Section 2 gives an overview of the lazy release consistency memory model. In section 3, we describe our proposal for a multi-LAN DSM, as well as some preliminary results. Section 4 discusses the advantages of having the CPAR-DSM on top of a multi-LAN DSM. Section 5 briefly describes the current state of our project. In section 6, some related works are presented while the last section summarizes the contributions of this work and exposes some other ideas for the evolution of it.

II. Lazy Release Consistency

In **Lazy Release Consistency** memory model [AMZ 96] [KEL 95], ordinary memory accesses are distinguished from synchronization ones, i.e., the **acquire** and **release** operations. Information about updates made by a process on its local page copies are propagated out to a second one only when the latter performs an acquire operation. This postponement reduces much of the communication that is required to make shared data consistent.

The execution of each process is divided into **intervals**. A new interval begins at each acquire or release operation. Acquire and release operations set up the causality of shared memory updates. Usually, two synchronization variables are provided: **barriers** and **locks**. A barrier is used to sequence the execution of the parallel program while a lock is used to control processes' access to shared variables. Operations on locks and barriers can be mapped onto acquire/release operations: a lock operation corresponds to an acquire, an unlock to a release while an arrival at a barrier can be modeled as a release and the departure from it as an acquire operation.

Partial ordering of acquire and release operation is controlled by assigning a per process vector timestamp to each interval [FID 91] [MAT 89]. Each process P_j keeps a vector clock v_j of N entries, where N is the total number of processes of the system. P_j controls the intervals created by itself using the jth entry of its vector clock. The other entries store the current knowledge that this process has of the updates made by other processes. Process P_j updates its vector clock v_j, at each synchronization operation, as follows:

- if it is a remote acquire operation, P_j updates its vector clock with the maximum of its current value and the releaser's v_r, i.e., $v_j = max(v_j, v_r)$;
- P_j adds 1 to the jth entry of its vector clock $v_j[j]$.

Thereby, process P_j, at an acquire operation, sends its current clock value to the releaser process P_r. This one sends back all the intervals covered by its local clock but not by P_j 's, including the identification of the pages that have been modified in each interval. Each identification is stored in a structure called *write-notice*. When receiving the *write-notices*, P_j invalidates the local copies of the corresponding pages (invalidate protocol). Hence, the first access to an invalid page will cause a page fault. The faulting process, P_j in this case, will, then, ask for the missing updates to all processes that last modified the page. It will apply them in the order defined by the causality of the intervals. These updates come in the form of *diffs*, a word-by-word comparison between a copy of the original page and its last version. Therefore, a *diff* of a page contains only the data that have been modified during an interval. It is generated, at the end of the interval, by comparing the page to a copy (*twin*) saved at the beginning of the interval.

III. The LRC Multi-LAN DSM

Our proposal is to have a modular, hierarchical and scalable DSM. We consider that these three features are extremely important for several reasons: the number of processors (machines) of a multi-LAN platform can be quite significant; the cost of sending a message between processors belonging to distinct LANs is much higher than between processors within the same LAN; it might be the case that the same DSM module is not loaded on all LANs; the configuration of the platform can dynamically change (e.g., temporary inaccessibility of remote LANs machines, network partitioning).

The scalability, modularity and hierarchy features are going to be provided by the protocol itself, i.e., the LRC protocol will be modified. The **barrier-lock** clocks will replace the traditional per process ones in order to deal with the scalable and modular aspects while the simulation of a **LAN-level cache**, **data pre-fetching** and **two-level execution of barriers** will be responsible for the hierarchical approach and, therefore, for the reduction of data transferred between LANs. These features will be discussed in the subsections that follow.

A. The barrier-lock clocks

The traditional per process vector logical clock [MAT 89] [FID 91] keeps an entry for each process (node) of the system. This means that in order to be able to connect several per LAN DSM modules we need a larger vector clock, whose size comprises the total number of processors (processes) of all LANs, which restricts the scalability and modularity of the multi-LAN DSM to some extent. The ideal would be to have a logical clock whose size does not depend on the number of nodes. This would allow the comparison of the timestamps of two distinct DSM modules, which is impossible with the tradition per processor vector timestamps.

Considering the above needs, restrictions, and the fact that our solution is composed of an interconnection of DSM modules, we have conceived a new logical clock. We have named it the *barrier-lock* clock [ARA 99a]. As causality in LRC DSM is induced by locks and barriers operations, the barrier-lock clocks have been modeled based on these oper-

ations.

A *barrier-lock* clock timestamp i is represented by the tuple $(b, vl)_i$, where b is a barrier call counter and vl a per lock vector variable.

At each barrier call, the scalar counter b of all processes is incremented while their lock vector variable vl is reset.

Process P_j, at each acquire/release operation on lock l, updates its vl vector variable as follows:

- if it is a remote acquire operation, $vl_j = max(vl_j, vl_r)$, where vl_r is the per lock vector vl of the release process P_r;
- P_j always adds 1 to the entry of vl_j that corresponds to lock l ($vl_j[l]$).

If L is the number of locks required by the application, then the size of a *barrier-lock* clock timestamp is $L + 1$. In other words, the size is proportional to the number of locks, independently of the number of processors of the system.

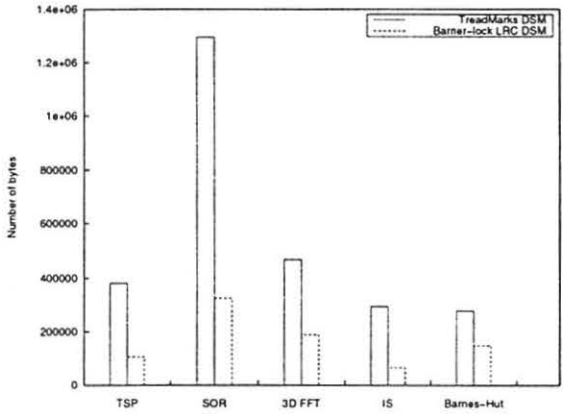

Fig. 1. Volume of data at synchronization operations

The replacement of the traditional logical clocks by the barrier-lock ones turns feasible the idea of having a global DSM which is composed of an interconnection of several DSM systems since a timestamp of a DSM module does not depend on the number of processors of its LAN. More details about the *barrier-lock* clocks can be found in [ARA 99a] and the proof that they precisely capture causality of synchronization operations is presented in [ARA 99b].

Figure 1 shows the volume of data exchanged at synchronization operations by 5 applications running on top of TreadMarks software DSM, version 0.10.1, and the same software where the traditional per process vector timestamps were replaced by the *barrier-lock ones*. We believe that these measures are quite representative since a synchronization message contains only timestamps and *write-notices*.

The tests have been made on top of 8 Sun-sparc-5 stations linked by a 100 Mbit/s Ethernet backbone with the following applications: SOR and TSP (distributed by TreadMarks); IS and 3D FFT (NAS benchmark [BAI 93]); Barnes-Hut (SPLASH benchmark [SIN 92]). We have simulated a platform with 32 processors by increasing the constant that defines the number of processors of the system. The input parameters (size and iterations), number of requested locks and number of barrier calls are summarized in Table I.

TABLE I

APPLICATION CHARACTERISTICS

Applic.	*Size,Iterat.*	*# of locks*	*# of barrier calls*
TSP	19 cities	2	3
SOR	512x512,200	0	401
IS	256x128,10	0	82
3D-FFT	256x16x16,50	0	104
Barnes-Hut	16K,3	0	13

Compared to the original TreadMarks, we can remark a significant decrease in the amount of data exchanged at synchronization operations for all the applications when running on top of our *barrier-lock*-based prototype. The reason for this is that all the applications employ a small number of locks (from zero to 2). Hence, the size of the timestamps is smaller than the traditional per processor ones (32 in this case). This reduction may result in better performance for systems whose links have a high per byte transmission cost (e.g. low bandwidth inter-LAN links of multi-LAN platforms).

B. Reduction of Inter-LAN Data Transfers

If an application defines a small number of locks, the use of *barrier-lock* clocks by the LRC protocol may reduce the volume of data transferred at synchronization operations, as shown in the previous subsection. However, this reduction is not so significant as to improve the application's performance. The major volume of inter-LAN communication is due to updated data requests, i.e., the transfers of the data themselves (*diffs*). Thus, in order to have less traffic over the inter-LAN links, the number and the size of *diff* messages must be reduced. We propose, then, the *pre-fetching* of *diffs* at synchronization operations, the simulation of a *LAN-level-cache* and the execution of global barriers in a *tree-structure* way.

B.1 Pre-fetching of Data

In the LRC invalidate protocol, *diffs* are sent to a process only when the latter gets a page fault. This laziness in propagating the *diffs* in fact results in an extra message whenever a process accesses an invalidated page. Thus, instead of including only the *write-notices*, (i.e., the identifications of the pages that have been modified) in the synchronization mes-

sages to a remote-LAN acquiring process, a releasing processor can also piggyback in this message the *diffs* that it believes, based on heuristics, that the acquiring remote LAN will demand in the near future. This protocol is called **lazy hybrid protocol** [KEL 95]. Good prediction of future accesses of remote LAN processors can significantly decrease the number of access misses and, consequently, the number of messages sent across inter-LAN links. On the other hand, bad prediction entails useless data sent across inter-LAN links.

Several works have proposed some heuristics for data prefetching or for the use of update protocol instead of the invalidate one [AMZ 99] [KEL 95] [MON 98]. For instance, a good heuristic for programs with lock-protected migratory data (i.e., when the same shared data, protected by the same lock, are sequentially updated by a set of processors [AMZ 99]) is to send to an acquiring process those *diffs* which the latter does not have and which correspond to previous updates to the shared data, that have been protected by the lock being acquired [MON 98]. In the case of the multi-LAN LRC, the releasing LAN should send to the acquiring LAN all the lock-protected *diffs* which any of the processors of the releasing LAN possesses. The ideal is to aggregate all *diffs* in a large message as the overhead of sending small messages over a slow link can be quite high [LU 97]. For barrier-like programs, a good heuristic may be the set of *diffs* whose pages were modified by the majority of processors of the remote LAN in the previous barrier call. Both heuristics can be implemented using the information provided by the *barrier-lock* timestamps as their values specify the type of synchronization operation that was performed.

B.2 Simulated LAN-level Cache

In our multi-LAN DSM, each LAN behaves like a physically shared-memory SMP node, i.e., any *diff* previously requested by any of the processors within a local LAN is "available" to all of them. Basically, the idea is to simulate a **LAN-level cache** in order to reduce the volume of cross-LAN data.

Using LRC protocol (invalidate mode) when a process gets a page fault, it needs to obtain the missing *diffs* and apply them to the corresponding page. The process knows which are these *diffs* since it holds all the *intervals* and *write-notices* received at previous synchronization operations. In the original LRC, the faulting process, q, always requests these *diffs* to the last processes that have modified the page (if a process p has modified a page at a timestamp t then p has all the *diffs*, including those from other processes, corresponding to timestamps that causally precede t [AMZ 96]). In the case of the multi-LAN LRC, whenever p belongs to a LAN different from q's, the requested list of causal-related *diffs* is split in two. Process q identifies for each list which is the last process of its local LAN that is included in the list. Thus, q asks to this process for the *diffs* that it holds and asks to the distant-LAN process p for the others, i.e., the ones not available within its local LAN. In this way, the amount of *diffs* sent across the inter-LAN is reduced.

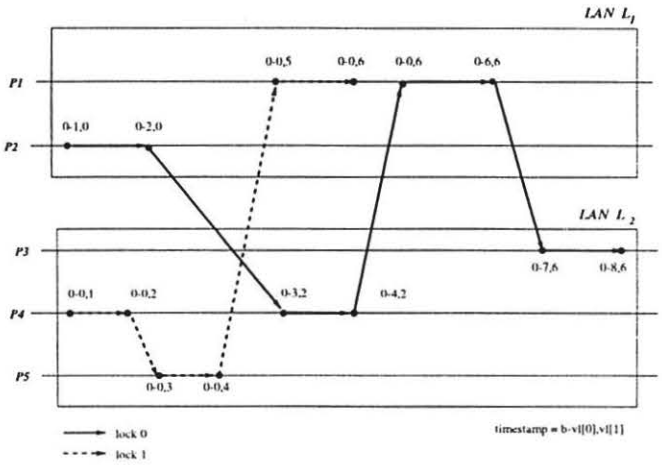

Fig. 2. LAN-level cache approach

Figure 2 illustrates better the LAN-level cache approach. It shows five processors divided into two LANs: P_1 and P_2 belonging to LAN_1 and P_3, P_4, and P_5 belonging to LAN_2. Two locks (lock 0 and lock 1) are requested by the application. *Barrier-lock* clocks are used by both LRC DSMs. The $b - vl[0], vl[1]$ timestamp (barrier counter plus per lock vector) at each acquire or release operation is indicated. For instance, suppose that after acquiring lock 0 at 0-7,6, processor P_3 attempts to access an invalidated page. It will get a page fault and, thus, it will need to request the related *diffs*. If we consider that P_1 was the last processor to have modified the page, in the original LRC, P_3 would address the *diff* requests to this processor (2 *diff* requests: one concerning lock 0 and the other lock 1 paths of causality). However, at this point, P_3 knows that some local processors of its LAN already keep part of these *diffs* (information received when it acquired lock 0). Thereby, in the case of the multi-LAN LRC, P_3 will split each *diff* request in two: for P_5 it asks for the *diffs* until the timestamp 0-0,4 (when P_5 lost lock 1); for P_4 the *diffs* until the timestamp 0-4,2 (when P_4 lost lock 0) and for P_1 the remaining *diffs*.

For validating this approach, we have implemented a prototype, modifying the same version of TreadMarks (0.10.1), where two LANs are simulated. Remote accesses are uniform, i.e., intra-LAN latency and bandwidth are equal to inter-LAN ones. Table II shows some measures obtained when the split of *diff* requests is applied (LAN cache) and when it is not for the IS application, using the same 8 Sunsparc-5 workstations platform. We have chosen the IS appli-

cation, from NAS benchmark [BAI 93] because the volume of data exchanged at each *diff* request is quite high since this application presents the problem of *diff* accumulation. The *diff* accumulation problem occurs in the case of migratory data.

TABLE II
COMPARISON OF IS APPLICATION PERFORMANCE

	TreadMarks	*LAN cache*
Inter-LAN vol.of diffs	$22.8x10^6$*	$16.1x10^6$*
Diff req. waiting time	32.1s	23.8s
Barrier waiting time	27.6s	16.5s
Overall time	41.1s	33.7s

* bytes

We observe that the volume of data transferred between the processors of different LANs was reduced when a LAN-level cache is provided. This decrease plus the overlap of *diff* requests were both responsible for the reduction of the time that a processor waited for the *diffs*. Therefore, the application was less unbalanced (i.e. smaller *diff* request waiting time) and the maximum time that a process was blocked at a barrier was reduced. Thus, the simulation of a LAN-level cache resulted in an overall better performance for the IS application. It is worth remarking that we have not considered a higher latency or lower bandwidth for the inter-LAN links. When we have added these parameters, we believe the results will be even more promising.

B.3 Two-level Hierarchical Execution of Barriers

The idea of implementing the execution of barriers in a tree-structure way, as proposed in [BIL 98] or [HU 99], is quite interesting and appropriate for a multi-LAN DSM.

Usually, in most DSM systems, a single centralized barrier manager is responsible for assembling all causality information from the other processes and propagating them back after these ones have reached the barrier. This centralization may, in the case of a multi-LAN DSM, represent a strong bottleneck. In order to minimize this bottleneck, a tree-structure approach for the execution of barriers is particularly adequate. Each LAN will have a local barrier manager that gathers the causality information of the processes within its local LAN. After having received all local information, each manager will send the aggregate causality information to a global manager. Thus, the execution of the barrier will be done in two levels, increasing the parallelism of a barrier execution. Furthermore, the grouping of data, done by each local manager, will reduce the number of messages sent over inter-LAN links. Hence, the hierarchical execution of barriers will probably result in decrease of barriers execution time as a whole.

IV. CPAR LANGUAGE AND A MULTI-LAN PLATFORM

The **CPAR-DSM** system [ARA 98] provides a programming language, the **CPAR** [SAT 94] [SAT 95], and a processing support that ease the development of parallel applications that run on top of distributed shared memory (DSM) systems. The first version of CPAR-DSM system was implemented on top of SunOS workstation network, using the Quarks DSM [KHA 96]. This DSM offers the **release consistency** memory model (eager). Basically, in eager release consistency model, the updates made by a process on its local shared variables' copies are propagated out to other processes when the former executes a release operation, i.e., not at the next acquire operation as in the lazy release consistency.

Pure DSM systems, in general, offer only a small set of programming primitives, basically for shared memory allocation, task synchronization, and control of critical sections. The CPAR language, an extension of the C language, provides more powerful constructions that simplify the definition and programming of parallel applications.

For generating a executable code, the CPAR programming is first parsed by a pre-compiler, the CCPAR, which converts the original source program into a C program, including calls to the CPAR library. The generated code is then compiled and linked to the CPAR-DSM processing support (CPAR library plus the DSM library).

CPAR-DSM task execution assumes SPMD computation model (Single Program Multiple Data), where the same program is loaded on all processors but due to conditional programming, they execute different segments of code or work on different data-sets.

What makes CPAR language specially appropriate for a multi-LAN platform is the fact that it offers constructions for exploring hierarchy in terms of task execution and organization of shared memory:

- A program can be organized into nested logical parallel execution blocks. *Macrotasks* are logical independent unit within the main function of the program. *Macrotasks* can execute in parallel. Once a *macrotask* is active, it can be subdivided into one or more *microtasks*, which are responsible for the finer grain of parallelism. This one can be of two types: data and control. In the former, the same operation is applied over multiple elements of the same data structures, while the latter refers to parallel execution of distinct segments of code. Each *microtask* is executed by a processor. Within a *macrotask* sequential code can be interleaved with parallel code. At the end of a *macrotask*, its *microtasks* are always synchronized. Besides, primitives are offered to the application (master process) to either wait for the

completion of all *macrotasks* or for the completion of a single *macrotask*;
- CPAR shared variables can be defined either as *global* or *local* to a macrotask. The *global* shared variables can be accessed by any *microtask* of any *macrotask* while *local* variables are declared within the scope of a *macrotask*, being accessible only by the *microtasks* of the *macrotask* in question.

The CPAR hierarchical constructions are quite adequate for a multi-LAN platform. Each *macrotask* can be assigned to a single LAN. This means that the *microtasks* of a *macrotask* are dispatched to the processors of the assigned LAN. *Local* shared variables are accessed only by the *microtasks* (processes) of this LAN, while the *global* shared variables by any *microtask* of any LAN. This hierarchy of shared memory is very important as only the pages related to global variables may cross LAN boundaries. Therefore, latency and bandwidth problems of inter-LAN links concern only these pages.

```
task spec mtask1 ( );
task spec mtask2 ( );

shared int vet[N];

task body mtask1 ( )
 { shared int a,b;
   ...
   parbegin
    ...
   also
    ...
   parend
  ...}
task body mtask2 ( )
{ ....
   forall i=1 to N {
   ...}
 ...}
```

```
main ( ){
 printf ("begin\n");
 alloc_proc (5);

 / * mtask 1 executes on LAN 1 */
 create 2, mtask1 ( );

 / * mtask 2 executes on LAN 2 */
 create 3, mtask2 ( );

/* synchronization of all processes */
 wait_all ( );

 printf ("end\n");
}
```

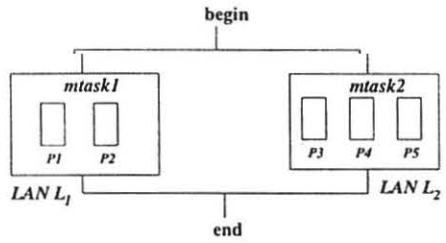

Fig. 3. CPAR program source and execution diagram

Figure 3 shows an example of CPAR program source and its parallel logical execution diagram. The program executes on a multi-LAN platform (the same one as figure 2). Two *macrotasks* **mtask1** and **mtask2** are defined. Each one is assigned to a different LAN. The shared variable **vet** is global to both *macrotasks* while variable **a** and **b** are only accessible to the *microtasks* of *macrotask* **mtask1**.

V. WORK-IN-PROGRESS

We have developed a first prototype with the **barrier-lock** clocks and where the **LAN-level cache** is provided. We are now working in a second prototype where data pre-fetching and multi-level execution of barriers will be implemented. A higher inter-LAN latency and lower bandwidth will be simulated for validating the prototype.

The porting of CPAR-DSM system to the LRC multi-LAN DSM basically consists of adapting its library to **TreadMarks** DSM. However, we believe that it will be easely done as both **Quarks** and **TreadMarks** offer quite the same set of API primitives and both of them are based on release consistency model (eager and lazy, respectively). Furthermore, when the first version of CPAR-DSM was developed, its porting to other DSM systems was foreseen [ARA 98]. Thus, the current CPAR-DSM library groups all the primitives offered by a DSM into a single module, adopting a standard interface that can be easily adapted to other DSM systems.

A future work would be to change the **CCPAR** precompiler for providing useful information for the DSM support, as in [DWA 99]. Recognition of future shared access patterns and parallel support for reductions [KEL 97] are some examples of the features that could be offered by the pre-compiler CCPAR. s

VI. RELATED WORKS

Bal et al. [BAL 98] make a performance study of medium-grain parallel applications running on top of a multi-LAN platform. They were written in Orca, a parallel object-based language. Contrary to our solution, the optimizations to minimize inter-LAN latency and bandwidth restrictions were introduced to the application and not to the shared memory support. Their strategy consists in changing the application algorithms themselves in order to have less traffic over the inter-LAN links, taking into consideration the multi-level network structure. The optimizations comprise, for instance, the coalescence of messages, the caching of data at LAN level, the replacement of centralized queues by distributed ones or the reduction of the number of synchronization points. Each application has received one or more of these optimizations and most of them have presented a better performance.

In [LU 98], the authors present a system that allows **OpenMP** programs to run on top of network of workstations. The OpenMP is an emerging standard for parallel programming on shared-memory multiprocessors. Their solution consists of a compiler, based on SUIF, that generates appropriate calls for a multi-threaded version of **ThreadMarks** DSM.

The evaluation of the use of a DSM as the target for parallelizing compilers is discussed in [COX 97]. The ARP

shared-memory (**SPF**) compiler is used to generate programs to **TreadMarks**. The authors conclude that this platform is adequate for irregular applications as they have presented better performance than their equivalent compiler-generated message passing ones. A similar approach is presented in [KEL 97], where the **CVM** DSM offers the support for **SUIF** compiler-generated shared-memory parallel programs. Some enhancements were added to the CVM DSM as, for instance, the piggyback of updated data at barrier synchronization messages. In [DWA 99], the **Parascope** parallel programming environment was extended in order to provide future shared access information to the DSM support, **TreadMarks**.

Several works [COX 99] [STE 97] [SAM 98] present the extending of some uniprocessor software DSM to **networks of SMPs** (symmetric multiprocessors). As within a SMP node, local shared memory coherence is performed in hardware, the performance of the system is supposed to be better than the performance of its equivalent uniprocessor DSM. Hu et al. present in [COX 99] a version of TreadMarks (multi-threaded), adapted for SMP network architectures. Their solution exploits the intra-SMP hardware shared memory, using POSIX threads. The Cashemere-2L [STE 97] implements a multi-writer, directory-based release consistency protocol. HLRC-SMP [SAM 98] implements the home-based multiple-write LRC protocol (HLRC) across SMP nodes. Processes within an SMP share the intra-node physical address space and take advantage of its synchronization mechanism, but each process has a separate page table.

In terms of logical clocks, we can find in the literature some clocks that present scalability since they can be constructed with a constant number of entries, independent of the number of nodes of the system. On the other hand, they do not precisely capture causality between events. For instance, the **plausible** clocks [TOR 96] provide a high level of ordering accuracy. However, they do not guarantee that certain pairs of concurrent events will not be ordered. Hence, they are not appropriate for implementing LRC protocol, as this ordering of concurrent events could lead to unnecessary consistency operation and remote requests, which will probably give rise to communication overhead.

VII. CONCLUSIONS

We have present in this paper our proposal for a multi-LAN programming support for shared-memory parallel applications.

The support for the shared memory consists of an interconnection of LRC DSM systems. Modifications have been made to the LRC protocol in order to adapt it to the hierarchy of the architecture. The use of the *barrier-lock* clocks provides scalability and modularity for the solution while the LAN-level cache, data pre-fetching and multi-level execution of barriers approaches take latency and bandwidth gap between inter-LAN and intra-LAN processors into account. The reduction of inter-LAN communication aims the speed up of applications.

The CPAR-DSM system will be ported to the multi-LAN DSM, as the CPAR language constructions are extremely appropriate for the development of parallel applications which run on top of multi-LAN platforms.

Future works will consider the CCPAR pre-compiler being able to analyze the source code in order to predict shared access patterns, the problem of network partition and other issues related to scalability such as global synchronization (barriers, garbage collection support), lock serialization, initial placement of pages, and consumption of memory.

REFERENCES

[AMZ 96] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony; W. Yu and W. Zwaenepoel. TreadMarks: shared memory computing on networks of workstations. *IEEE Computer*, 29(2): 18-28, February 1996.

[AMZ 99] C. Amza, L. Cox, S. Dwarkadas, J. Jin, K. Rajamani; W. Yu and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proceedings of the IEEE*, 87(3):467-475, March 1999.

[ARA 98] L. Arantes, and L. Sato. CPAR-DSM : a support for parallel programming on top of DSM. In *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications*, Las Vegas, July 1998.

[ARA 99a] L. Arantes, B. Folliot, and P. Sens. A customized logical clock for timestamp-based relaxed consistency DSM systems. In *Proceedings of the 1999 Workshop on Software Distributed Shared Memory held in conjunction with ICS'99 (ACM/SIGARCH)* Rhodes, Greece, pages 1-6, June 1999.

[ARA 99b] L. Arantes, B. Folliot, and P. Sens. A node-count independant logical clock for scaling Lazy Release Consistency Protocol. To appear in the *Proceedings of Europar'99*, Toulouse, France, Septembre 1999.

[BAI 93] D. Bailey, J. Barton, T. Lansinski, and H. Simon. *The NAS parallel benchmark*. Technical Report 103863, NASA, July 1993.

[BAL 98] H. Bal, A. Plaat, M. Bakker, P. Dozy and R. Hofman. Optimizing parallel applications for wide-area clusters. In *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, April 1998.

[BIL 98] A. Bilas, L. Iftode, R. Samanta and J. P. Singh. Supporting a coherent shared address space across SMP nodes: An application-driven investigation *IMA Volumes in Mathematics and its Applications (Algorithms for Parallel Processing)*, 105, Springer-Verlag, New York, 1998.

[COX 97] A.Cox, S. Dwarkads, H. Lu and W. Zwanenepoel. Evaluating the Performance of Distributed Shared Memory as a Target for Parallelizing Compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 447-482, April, 1997.

[COX 99] A.Cox, Y. Hu, H. Lu and W. Zwanenepoel. OpenMP on Networks of SMPs. In *Proceedings of the 13th International Parallel Processing Symposium*, April 1999.

[DWA 99] S. Dwarkadas, H. LU, A. Cox, R. Rajamony and W. Zwaenepoel. Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory. *Proceedings of the IEEE*,87(3), March 1999.

[FID 91] C. Fidge Logical Time in Distributed Computing Systems. *IEEE Computer*, pages 28-33, July 1991.

[IFT 96] L. Iftode, C. Dubnicki, E. Felten and K. Li. Improving Release Consistency Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[HU 99] W. Hu, W. Shi and Z. Tang. Reducing System Overheads in Home-based Software DSMs. In *Proceedings of the 13th International Parallel Processing Symposium*, April 1999.

[KHA 96] D. Khandekar. *Quarks: distributed shared memory as a building block for complex parallel and distributed systems*. Master's Thesis, Department of Computer Science, University of Utah, Salt Lake City, (EUA),1996.

[KEL 95] P. Keleher. *Lazy release consistency for distributed shared memory*. Ph.D. Thesis, Rice University, January 1995.

[KEL 97] P. Keleher and C. Tseng. Enhancing Software DSM for Compiler-Parallelized Applications. In *Proceedings of the 11th International Parallel Processing Symposium*, April, 1997.

[LU 97] H. Lu, S. Dwarkadas, A. L. Cox and W. Zwanenepoel. Quantifying the Performance Differences Between PVM and TreadMarks. *Journal of Parallel and Distributed Computation*. 43(2):65-78, June 1997.

[LU 98] H. Lu, Y.C. Hu, and W. Zwanenepoel. OpenMP on Networks of Workstation. In *Proceedings of Supercomputing'98*, Orlando, EUA, November 1998.

[MAT 89] Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*. Elsevier Science Publishers, 1989.

[MON 98] L. R. Monnerat and R. Bianchini. Efficiently Adapting to Sharing Patterns in Software DSMs. In *Proceedings of the 4th Symposium on High Performance Computer Architecture*, January, 1998.

[SAM 98] R. Samanta, A. Bilas, L. Iftode and J. Singh. Home-based SVM protocols for SMP clusters: Design and Performance. In *Proceedings of the 4th Symposium on High Performance Computer Architecture*, February 1998.

[SAT 94] L. M. Sato. Programming language for multiprocessor systems with memory hierarchy. In: *Simposio Nipo-brasileiro de Ciência e Tecnologia*, pages 227-35, Sao Paulo, Brazil, 1994.

[SAT 95] L. M. Sato. *Ambientes de Programacao para Sistemas Paralelos e Distribuidos*. Tese de livre docência, Escola Politécnica da Universidade de Sao Paulo, Sao Paulo (Brazil), 1995.

[SIN 92] P. Singh, W. Weber and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Technical Report*, Stanford University, April 1991.

[STE 97] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, H. Kontothannassis,S. Parhasarathy and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote Write-Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Octobre 1997.

[TOR 96] F.Torres-Rojas and M. Ahamad. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG96, Bologna, Italy, Octobre 1996.