PARALLEL PROGRAMMING ON TOP OF DSM SYSTEM AN EXPERIMENTAL STUDY

LAURENT LEFÈVRE*

Abstract.
Parallelization of an application with a message-passing model requires time, effort and an important parallel programming knowledge from the programmer.

By using a Distributed Shared Memory model, programming a parallel application seems easy but reaching good performances becomes more difficult.

In this paper we propose an experimental study of the way to program parallel applications with the DOSMOS DSM system which provides an original programming model with shared distributed objects. Moreover by integrating a development platform and monitoring facilities, the DOSMOS system has been designed to offer a performant user-friendly programming environment.

Key words. Distributed Shared Memory System, Programming Environment, Shared Objects

1. Introduction. Parallelization of an application with a message-passing model requires time, effort and an important parallel programming knowledge from the programmer.

The purpose of Distributed Shared Memory systems (DSM) is to implement, above a distributed memory architecture, a programming model allowing a transparent manipulation of virtually shared data. Thus, in practice, a DSM system has to handle all the communications and to maintain the shared data coherence. By this way, by using a Distributed Shared Memory model, programming a parallel application seems easy but reaching good performances becomes more difficult.

In our laboratory we have developped an original programming environment, the so-called DOSMOS[1] system. This system is based on a structural approach of parallel programming. In other words, DOSMOS proposes to the user to hierarchically structure the processes into *groups* and *sub-groups of processes* sharing a same set of variables. This feature, combined with weak consistency protocols reduces the amount of communications required for the management of the shared data, and, as a consequence, ensures efficiency and scalability of the applications.

However, it would be unrealistic to argue that DOSMOS, or any other DSM system, deals efficiently with any kind of applications. That is why DOSMOS allows the programmer to mix both message-passing (PVM [GBD+93]) and DOSMOS code. To complete the programming environment, DOSMOS integrates a dedicated monitoring tool (DOSMOS-Trace) which has been added to the system to allow the user to understand the behavior of his applications.

At last, this programming environment has been designed to run both on distributed systems and on parallel machines. Thus, to ensure the portability of both the system and the applications, DOSMOS (as well as DOSMOS-Trace) has been developed on top of PVM.

But the main and open problems are "How to program a Distributed Shared Memory System like DOSMOS ? Is it well adapted for beginners or only for expert end-users ? Which is the best programming model to use for implementing parallel scientific applications, a shared model or a message-passing one ?"

*Laboratoire de l'Informatique du Parallelisme, Ecole Normale Supérieure de Lyon, 69364 LYON Cedex 07, France, llefevre@lip.ens-lyon.fr

[1]DOSMOS is the acronym for a **D**istributed **O**bjects **S**hared **M**em**O**ry **S**ystem.

This year, for the first time, DOSMOS has been teached like any other system in parallel courses of our university. Thus we have chosen among the students, a few parallel programming beginners and we analysed their reactions and behaviour in front of our DSM system when they had to parallelize their sequential algorithms.

This paper is divided into five parts. After a short description of previous works (section 2), we analyse the basics of the DOSMOS DSM system in terms of management of shared data and process structuring (section 3). Then a description of the programming environment is proposed (development platform, basic routines and programming model). In section 5, we discuss the way to program a DSM system like DOSMOS and the differences of programming and of performances between a DSM and a Message-passing system. At last, section 6 proposes a discussion both on the features of this programming environment and on its interest in terms of parallel programming.

2. Purpose of Distributed Shared Memory systems and previous works. By allowing the programmer to share "memory objects" (i.e. programming variables) in a transparent way, Distributed Shared Memory Systems (DSM) propose a interesting trade-off between the easy-programming of shared memory machines and the efficiency and scalability of distributed memory systems. Basically, a Distributed Shared Memory system is a mechanism that allows application processes to access shared data in a transparent way. In other words, a DSM system relaxes the programmer from the management of all inter-process communications.

Even if both hardware and software implementations have been proposed, most of the systems require the implementation of an additional software layer :

- **Virtual Shared Memory systems** (V.S.M.) propose to share data pages, i.e. to merge into a single wide address space a set of memory pages distributed in the network like MIRAGE[FP89] or MUNIN[CBZ91].
- **Object-based Distributed Shared Memory systems** (D.S.M.) work at the program level, i.e. they implement a software layer that automatically generates, on the user's behalf, all the communications required to manipulate shared data. In other words, instead of defining (and writing in the code) the inter-process communications, the programmer only specifies which data are actually shared. Then he can use these data as if they were local. On its side, the DSM system takes into charge all the needed communications (as a message-passing programmer would do). Such DSM systems as ORCA[TKB92] or CLOUDS[RAK89] have been implemented on parallel or distributed architectures. DOSMOS[BL94, BL96] system belongs to this kind of systems.

While a lot of DSM systems have been proposed, only few of them [CBZ91, SN93, BEP93, BBP94] describe the way to program and to obtain good performances with them.

3. Basics of DOSMOS. DOSMOS is an object-based DSM system which allows processes to share in a transparent way a set of passive objects (i.e. of users programming variables) distributed and replicated in the network.

Moreover, DOSMOS integrates novel features :

- **DOSMOS Processes** : Basically, a DOSMOS application is composed of two types of processes:
  - **Application processes (A.P.)** contain and execute the code (written in C) of the application.

  - **Memory processes (M.P.)** manage the whole DSM system, i.e. they provide A.P. with the objects they request and maintain data coherence. Each A.P. is connected to one and only one memory process. On the contrary, an M.P. can be connected to several A.P. and several M.P.
- **Array allocation** : DOSMOS allows to manipulate both basic type variables (integer, float, char...) and distributed arrays. These arrays are split into several "system objects", replicated among the processes. Various splittings are provided : by row, by column, by block and by cyclic block. The system ensures a transparent access to arrays, whatever the splitting implemented.
- **Weak consistency protocols** : for efficiency and scalability purpose, DOSMOS gives the opportunity to duplicate shared objects. It is clear that these replicas have to be kept coherent. Most of actually implemented models are strong consistency oriented. DOSMOS implements a weak protocol: the release consistency. This model [GLL+90] provides two synchronization operators: *acquire* and *release*. These operators allow processes which want to modify shared objects to lock and unlock them in order to implement a mutual exclusion of accesses on shared objects.
- **Hierarchical structuring of the application processes**: Previous DSM systems have always proposed "*flat*" models in which any shared object is accessible from any process. Such "*anarchical*" models cannot be scalable. In DOSMOS, processes can be grouped into groups and sub-groups in order to optimize the management of the data coherence.
  When one observes the behaviour of a DSM application, and more particularly the behaviour of a process participating in the application, it appears that if some shared data are intensively accessed by this process, some are either scarcely or even never accessed. This leads us to introduce some definitions (see fig. 3.1) :
  - *G.V.S. : The* **Global Virtual Space** *(GVS) of a process is the set of the shared objects accessed (in read or write mode) by a process during the execution of the application.*
  - *L.V.S. : The* **Local Virtual Space** *(LVS) of a process is the set of the shared objects intensively accessed by this latter.*
  - *E.V.S. : The* **Extern Virtual Space** *(EVS) of a process is the set of the shared objects rarely accessed by the process.*

  Let P be a process. We have :

$$G.V.S.(P) = Local\ Virtual\ Space(P) + Extern\ Virtual\ Space(P)$$

  Usually, in previous systems, when an object O is modified, an invalidation message is sent to all the processes P such that $O \in GVS(P)$. This prevents, as noted before, to ensure a good scalability. By using a hierarchical grouping of processes, DOSMOS limits the invalidation messages to processes such that $O \in LVS(P)$.
  Basically, DOSMOS proposes to structure the application into hierarchical groups of processes sharing the same objects. In practice, a group is defined by a set of processes and a set of shared objects. Processes of a same group share all the objects attached to the group, i.e. if they request an object, they will receive a copy of this object which will be automatically updated by the system.
  Moreover, DOSMOS allows processes to access extra-group shared objects. For this purpose, in each group, a dedicated memory process, called Link

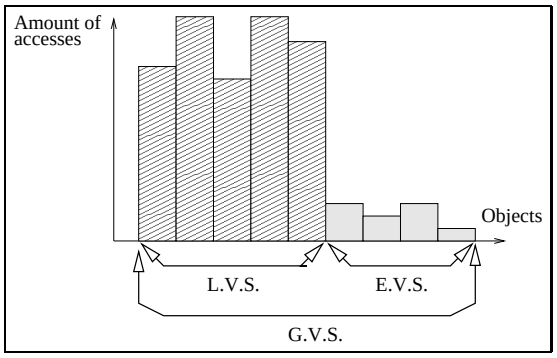

FIG. 3.1. *Accesses distribution of an application using an object-based DSM system*

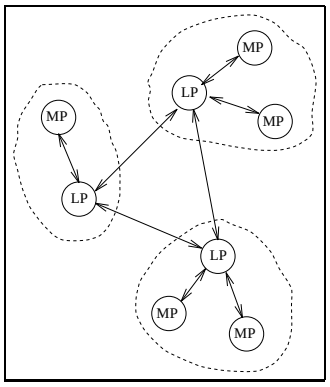

FIG. 3.2. *Groups and link processes in DOSMOS*

Process (LP), plays the role of link between groups (see Fig 3.2). Thus, these special MPs take into charge all the communications between groups.
With this model, access to shared objects is optimized and maintenance of the consistence is kept cheap. We obtain good performances improvements with experiments (like in figure 5.7) using hierarchical groups on network of workstations.

4. Programming environment.

4.1. Development platform. The implementation of DOSMOS is based on two different layers :

- **Preprocessing level**: this layer analyses the user's application in order to detect and generate accesses to shared objects (fig. 4.2). This layer makes the system "transparent";
- **DSM level**: this layer assumes the creation and management of shared objects, groups and of various processes involved in execution of the application (APs, MPs, LPs). This management is performed using message-passing routines hidden to the user.

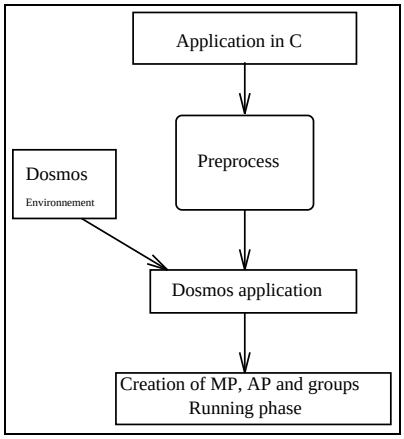

FIG. 4.1. *DOSMOS Environment : from an application written in C to the DSM execution*

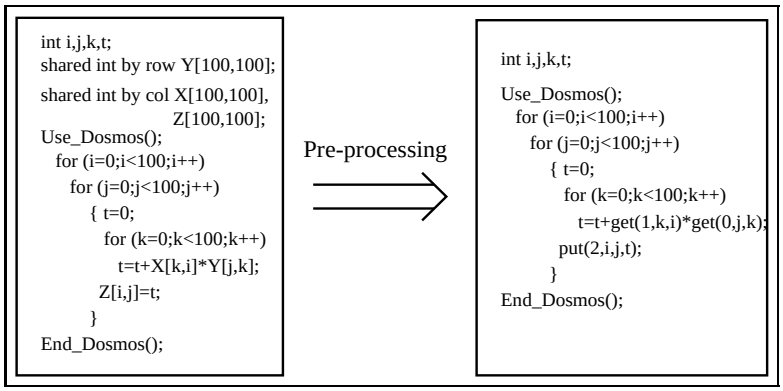

FIG. 4.2. *Example of pre-processing on a matrix multiplication application*

4.2. DOSMOS primitives. By adding only a few new primitives, DOSMOS system stays well-adapted for a beginner user. All accesses (except exclusive ones) are totally hidden to the user.

• #include	`Dosmos.h`
• Declaration	`shared` ...
• Begin-End	`use_dosmos()` - `end_dosmos()`
• Exclusive access	`acquire(`*`object`*`)`
	`release(`*`object`*`)`
• Synchronization	`sync(`*`object or group`*`)`

FIG. 4.3. *DOSMOS primitives*

Use_Dosmos() and *End_Dosmos()* allow the beginning and the end of sharing of the objects. All data accesses are not specified by the user but in order to access exclusively shared objects, two operators are proposed to the user : *Acquire* and *Release*. The synchronization routine allows to synchronize all processes sharing a given object or all processes of a given group.

4.3. Programming model. As soon as the *Use_Dosmos()* primitive has been executed, the user can access the shared objects in a transparent way. However, DOSMOS, as any DSM system, does not pretend to be efficient in all the situations. Consequently, in order to help the user to optimize his applications, DOSMOS allows the combination of different programming models for user's comfort. Consequently, three programming models are available :

- **Local programming** : in order to minimize accesses to shared objects, it is sometimes more performant to work on local variables before modifying shared variables.
- **D.S.M. programming** : the user can use DOSMOS primitives to declare and access shared variables.
- **Mixing of DSM and message-passing programming** : the user can integrate message-passing communications (with PVM routines) into DOSMOS applications. This feature presents two advantages. First, it permits to deal with specific applications. Second, it allows to port PVM applications on DOSMOS with slight modifications of the code.

5. Programming on top of DOSMOS.

5.1. Efficiency of data accesses. First of all, a battery of tests has been designed and implemented in order to validate the system efficiency. Usually, most of scalability problems encountered with DSM systems occur when several processes try to modify the same object. To understand DOSMOS behaviour with such problems, the next experiments involve a LAN of 8 workstations. Each processor runs both an Application Process and its dedicated Memory Process. No grouping of processes is performed. The 8 APs recursively attempt to access the same shared object in order to modify it. Figure 5.1 displays the execution time of these applications with respect to the number of write accesses performed. In the first example, relaxed write modifications are performed without synchronization (i.e. without acquire operations). In the second experiment, before each write access, an acquire operation is triggered in order to lock the object. As we can see, both experiments express quasi-linear results which prove the efficiency. The system is not overloaded when the amount of concurrent accesses is high (e.g. 500 write operations performed per second).

Write Operations	Relaxed Accesses	Acquire/Release Accesses
100	0.14	15.3
200	0.35	32.3
300	0.57	49.6
400	0.71	70.8
500	1.03	91.2

FIG. 5.1. *Write accesses to a single shared object with or without acquire/release calls*

Furthermore, these experiments outline the cost of the acquire/release operations (response time multiplied by 100). This argues for an optimization of the consistency protocols implemented (see section 3) and for the introduction of group management facilities (see section 5.3).

5.2. Scalability. In the following two applications, as previously, an Application Process and its dedicated Memory Process are mapped on each processor.

AP	Exec. Time	Efficiency
1	3.33	
2	1.8	0.92
4	0.93	0.9
8	0.5	0.83

FIG. 5.2. *Concurrent accesses to a shared object : a distributed array filling*

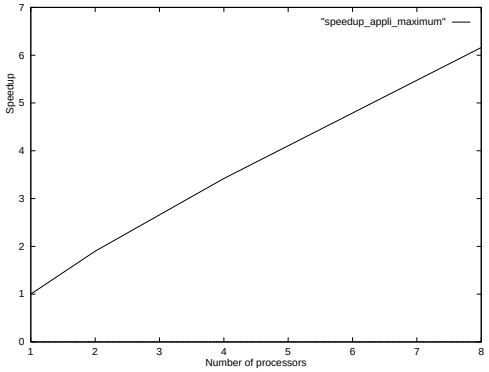

FIG. 5.3. *Speedup of a DOSMOS application computing a distributed maximum*

Figure 5.2 displays execution times of an application involving processes which

concurrently fill and modify a distributed shared array. Modifications are done using weak consistency protocols.

Next experiment (figure 5.3) shows the speed-up obtained by an application which computes the maximum value of a distributed set of data mapped on different processors. Thus, we obtain speedups of more than 6 and a good efficiency of more than 0.8 when using 8 processors.

In all experiments, a quasi-linear scalability is reached which confirms the effectiveness of the concepts introduced in the DOSMOS system.

5.3. Improving DOSMOS applications. Previous experiments have shown that designing small applications with DOSMOS permits to easily program parallel applications with efficiency. But reaching good performances with more important applications can be difficult. As DSM systems hide all communications and optimizations to the user, the performances can greatly vary depending on the user knwoledge of the system.

In figure 5.4, we can see the easiness of programming with DOSMOS but also the differences of codes between two DOSMOS users, a beginner and an expert one.

```
for( i=istart ; i<=iend ; i++ )
    {
         acquire( Matrice1[ligne,ligne]);
         pivot=get(Matrice1[ligne,ligne]);
         release( Matrice1[ligne,ligne]);

         acquire( Matrice1[i,ligne]);
         coef= -1.00 * get( Matrice1[i,ligne]) / pivot;
         release( Matrice1[i,ligne]);

         for( j=ligne; j<Matrice1:col ; j++ )
         {
            double d1,d2;

            acquire(Matrice1[i,j]);
            d1=get( Matrice1[i,j]);
            release(Matrice1[i,j]);

            acquire(Matrice1[ligne,j]);
            d2=get( Matrice1[ligne,j]);
            release(Matrice1[ligne,j]);

            if( !EstNul( coef ) )
              buf=d1+ coef * d2;
            else
              buf=d1;

            acquire(Matrice1[i,j]);
            put( Matrice1[i,j],buf);
            release(Matrice1[i,j]);
         }
    }
```

```
pivot=get(Matrice1[ligne,ligne]);

for( k=ligne; k<Matrice1:col ; k++ )
        d2[k]=get( Matrice1[ligne,k]);

for( i=istart ; i<=iend ; i++ )
  {
    coef= -1.00 * get( Matrice1[i,ligne]) / pivot;

    for( j=ligne; j<Matrice1:col ; j++ )
      {

        d1=get( Matrice1[i,j]);

        if( !EstNul( coef ) )
          buf=d1+ coef * d2[j];
        else
          buf=d1;

        put( Matrice1[i,j],buf);
      }
  }
```

FIG. 5.4. *From a first version of DOSMOS Gauss (left side) to an optimized code (right side)*

As DOSMOS provides various consistency protocols, beginner users prefer to guaranty the data accesses by choosing strong consistency schemes (first code). All put and get operations are exclusive and synchronized. So performance of the application decreases as all accesses are sequentialized. But this code can be greatly improved by using local variables and by relaxing some matrix accesses which do not need to be modified in a exclusive way for this application. An execution of DOSMOS Gauss with the first version takes 26 seconds instead of 13.1 seconds with the optimized version.

When the code is cleaned of all artifacts, we can reach scalable performances with

a DOSMOS execution. We can see in figure 5.5 that when the amount of data treated increases of more than two times, execution times follow the same behaviour.

Matrix Size	Execution Time
$10 * 10$	1.05
$20 * 20$	6.6
$30 * 30$	13.1

FIG. 5.5. *Execution times of Gauss with DOSMOS*

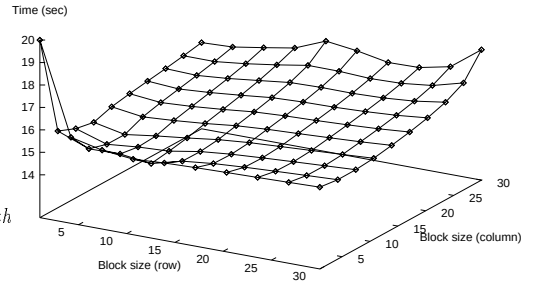

FIG. 5.6. *Finding optimal block size for a Gauss* $30 * 30$

After optimizing the code itself, many parameters concerning the shared objects must be tuned to improve performances. The previous Gauss experiment has to deal with various shared matrices which can be split in several blocks. On figure 5.6 we can see the great importance of that parameter on execution time with matrices of $30 * 30$ elements. If the matrix is not splitted, performances decrease because each access generates a sending of the whole shared matrix. On the other side, choosing a really small object size is worst because the system has to deal with a lot of very small messages for each access. For this experiment, the trade-off is to choose blocks with medium size. Here, blocks of 2x5 elements permit to reach the best execution time of 13 seconds oppositely of the worst case with up to 20s of execution time.

At last, the hierarchical structuring of application processes must be done to benefit of hierarchical groups provided by DOSMOS to reduce coherence costs.

In the next experiments (figure 5.7), we compute an approximation of π with an interval discretizing method. We use 12 Application Processes, mapped on 12 processors. With no group structuring, the computation requires 3.95 seconds. However, with the same number of processors, structuring the processes into two groups (each of them being in charge of one half of the interval), improves drastically the performances (new computation time: 2.56 s).

Like the other parameters, hierarchical grouping allows to obtain very good performances, on condition that it is pertinently used. Thus, we can obtain performances really close to the message-passing ones (see figure 5.8). The DOSMOS application is really easier to implement and only 8% slower than the PVM version with 16 processes. But the choice of the parallel programming model depends also on various parameters.

5.4. DOSMOS or Message-passing. To choose its programming model, the user must know which is the best-fitted to his applications' requirements. First, we see the differences between DOSMOS and PVM on basic accesses to data and then we compare the two environements in terms of performance and easiness of programming.

Processors	Groups	Exec.Time
12	0	3.95
12	2	2.56

FIG. 5.7. *π computation times (in sec.). One MP also runs on each processor.*

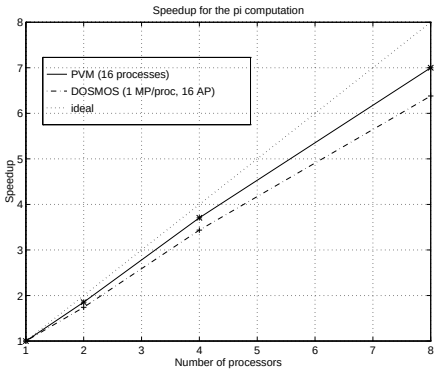

FIG. 5.8. *Computation of π ; comparison of the speedups obtained when using DOSMOS and when using PVM*

5.4.1. Read and Write operations. In DOSMOS, an Application Process asks to its dedicated M.P. the value of a shared object. This M.P. tries to know the value from the object owner and sends it back. Meanwhile, for the PVM Read version, a slave sends a message to a given process and waits for the value to come back. We can see in figure 5.9 that the DOSMOS Read operations are a little slower than PVM ones due to the DSM management layer. But with write operations, DOSMOS obtains better performances than the PVM version due to the bottleneck which occurs with the message-passing model. The DOSMOS relaxed consistency allows 5 processes to perform 1000 write modifications in less than 2 seconds. These experiments (see also figure 5.8) exhibit that PVM and DOSMOS can be equally compared on the basic access concepts.

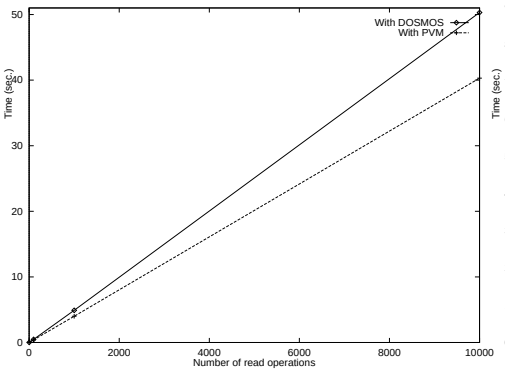

FIG. 5.9. *DOSMOS and PVM Read operations with 5 processes*

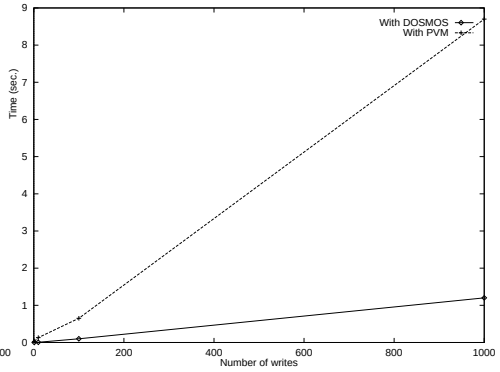

FIG. 5.10. *DOSMOS and PVM Write operations with 5 processes*

5.4.2. Choosing between DOSMOS and PVM. Our users have implemented a wide range of applications (Gauss, Cholesky, Matrix Multiply, Distributed Sorts...) with DOSMOS or PVM.

After these experiments, we can note several behaviours (also summarized in figure 5.11) :

- Programming with DOSMOS is easier than with PVM, and the code is close to the sequential version ;
- Designing an application with a message-passing model like PVM takes 4 more times than with a DSM model like DOSMOS ;
- Already written PVM codes can be easily ported and transformed to benefit of distributed shared objects because DOSMOS hidden underlying layer is based on PVM ;
- The code of the application is twice smaller by using DOSMOS primitives. Moreover, maintenance of a PVM code is much more difficult ;
- Understanding the behaviour and optimizing application performances are easier with PVM for an experimented programmer ;
- DOSMOS is more adapted to coarse grain or irregular parallel applications while PVM more designed for regular parallel algorithms ;
- Differences of performances between the two environments are slight with a small decrease for the DSM due to DOSMOS management layer;

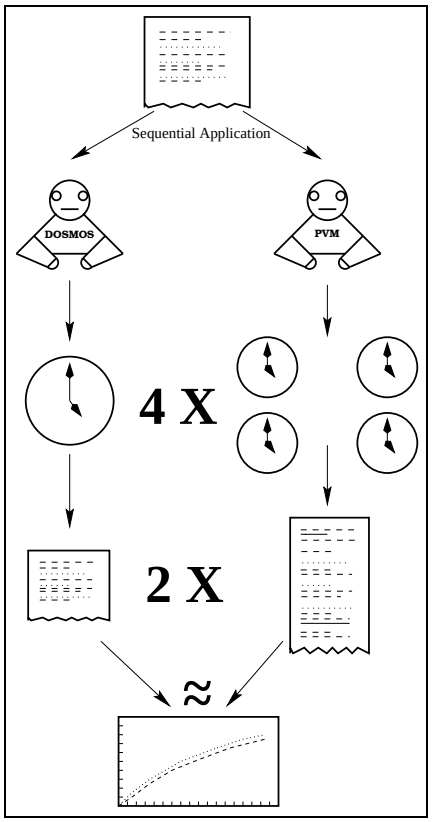

FIG. 5.11. *Programming a parallel application : DOSMOS or PVM ?*

By analysing the behaviour of users implementing DOSMOS applications, we can detect advantages and drawbacks of our DSM system. Moreover this study allows a better understanding of users requirements for a DSM environment comparing to the message-passing one.

6. Discussion and future works. This paper shows that DSM systems can be an effective way to implement distributed and parallel applications. But implementing a DOSMOS application deals with two main problems : the easiness of programming and the execution performances.

All communications of a parallel application can be totally hidden to the user by the DSM system. DOSMOS allows to program applications in a quick and easy way. Programming, testing and debugging are simplified and guaranteed by the reliability and efficiency of the system. But such a system can also appear as a *black box* as it is really difficult to understand applications behaviour. That's why, we have added a dedicated tool (called DOSMOS-Trace [BLR96]) which allows to precisely understand data access patterns of applications.

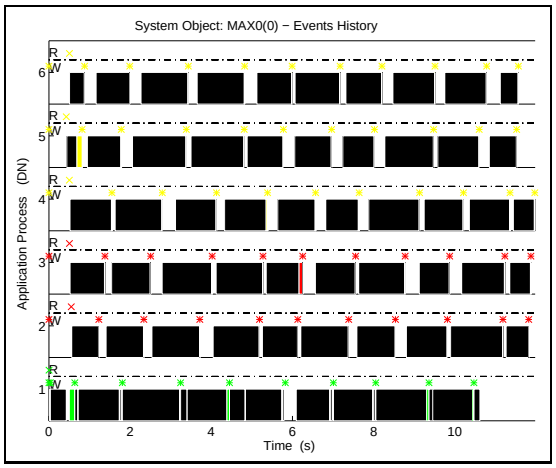

FIG. 6.1. *Object activity vs execution time : a bottleneck occurs during execution. All the processes exclusively access same shared data.*

This tool provides several visualizations and informations about the execution (like statistics on shared objects, histories...). Such views are extremely useful for the user to correct problematical situations. Indeed they allow to detect ping-pong effects, over-accessed variables, bottlenecks (like in figure 6.1), not actually shared variables, etc...So DOSMOS is a programming environment which can be used by beginners (at least as their first parallel programming approach). DOSMOS is also well designed for expert programmers which can improve and optimize the code as with any other parallel programming environment.

Many parallel programmers usually think that a DSM application will never beat an optimized message-passing application performances. Due to shared objects management, it is often true. But the costs (in terms of time, effort, debugging, maintenance...) to implement such performant message-passing applications are usually too expensive compared to the good results of DSM version. However, a lot of users prefer to implement themselves all the communications of the application. One possible trade-off is to use one of the interesting features of DOSMOS which allows to mix together PVM and DOSMOS code in a same application. By this way, the user can take advantage of each programming model without their drawbacks. It is also a good way to implement PVM code on DOSMOS to add shared objects with only small modifications on the original program. More experiments using both programming models are currently done on this domain.

REFERENCES

[BBP94] Didier Badouel, Kadi Bouatouch, and Thierry Priol. Distributing data and control for ray tracing in parallel. In *IEEE Computer graphics and applications*, pages 69–77. IEEE, July 1994.

[BEP93] François Bodin, Jocelyne Erhel, and Thierry Priol. Parallel sparse matrix vector multiplication using a shared virtual memory environment. In *6th SIAM Conference on parallel processing for scientific computing*, Norfolk, Virginia (USA), March 1993.

[BL94] L. Brunie and L. Lefèvre. Modèle de mémoire distribuée-partagée pour machine massivement parallèle. In *RenPar'6*, Ecole normale Supérieure de Lyon, France, June 1994.

[BL96] Lionel Brunie and Laurent Lefèvre. New propositions to improve the efficiency and scalability of DSM systems. In IEEE, editor, *1996 IEEE Second International Conference on Algorithms & Architectures for Parallel Processing - ICA3PP '96*, pages 356–364, Singapore, June 1996.

[BLR96] Lionel Brunie, Laurent Lefèvre, and Olivier Reymann. Execution Analysis of DSM Applications: A Distributed and Scalable Approach. In ACM Press, editor, *SPDT'96 : SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 51–60, Philadelphia, Pennsylvania, USA, May 1996. FCRC.

[CBZ91] John B. Carter, John K. Bennet, and Willy Zwaenepoel. Implementation and performance of MUNIN. *ACM - Operating Systems Review*, 25(5):152–164, 1991.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. In ACM PRESS, editor, *Proceedings of the twelfth ACM Symposium on Operating Systems Principles*, volume 23, pages 211–223, The Wigwam Litchfield Park, Arizona, December 1989.

[GBD+93] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, Tennesse, May 1993.

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, 1990.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: unifying synchronization and data transfer. In *1989 International conference on parallel processing*, volume II, pages 160–169, 1989.

[SN93] Ambuj Shatdal and Jeffrey F. Naughton. Using parallel virtual memory for parallel join processing. In *ACM-SIGMOD Conference*, March 1993.

[TKB92] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel programming using shared objects and broadcasting. *IEEE computer*, 25(8):10–19, August 1992.