Two Layers Distributed Shared Memory

F. Baiardi[1], D. Guerri[2], P. Mori[1], L. Moroni[1], and L. Ricci[1]

[1] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 - Pisa (Italy)
[2] Synapsis S.r.l., P.zza Dante 19/20, 57124 - Livorno (Italy)
{baiardi,guerri,mori,ricci}@di.unipi.it

Abstract. This paper presents a methodology to design a distributed shared memory by decomposing it into two layers. An application independent layer supplies the basic functionalities to access shared structures and optimizes these functionalities according to the underlying architecture. On top of this layer, that can be seen as an application independent run time support, an application dependent layer defines the most suitable consistency model for the considered class of applications and it implements the most appropriate caching and prefetching strategies for the consistency model. To exemplify this methodology, we introduce DVSA, a package that implements the application independent layer and SHOB, an example of the second layer. SHOB defines a release consistency model for iterative numerical algorithms and it implements the corresponding caching and prefetching strategies. We present some experimental results of the methodology and discuss the performance of a uniform multigrid method developed through SHOB on a massively parallel architecture, the Meiko CS2, and on a cluster of workstations.

1 Introduction

A Distributed Shared Memory, DSM, is a software layer that emulates a shared memory on distributed memory machines. A DSM allows the application developer to focus its interest on the application problems, rather than managing the physical allocation of the data onto the local memories and the coordination of data movement. We introduce a methodology that defines a DSM by composing two layers:

- an application independent layer, that includes the basic functionalities to access shared data and to synchronize the accesses. The functionalities of this application independent layer are optimized according to the concurrent mechanisms supported by the underlying architecture;
- an application dependent layer built on top of the previous one to implement a given consistency model. This layer implements the most effective caching and prefetching strategies for the considered model. This is an application dependent layer because different classes of applications require alternative consistency models.

B. Hertzberger et al. (Eds.): HPCN Europe 2001, LNCS 2110, pp. 302–311, 2001.

To exemplify this methodology, we present DVSA, Distributed Virtual Shared Areas, a package that implements the application independent layer and structures the shared memory as a set of areas. Currently, two DVSA implementations have been developed on, respectively, the Meiko CS2 [4] and a cluster of workstations, COW.

As an example of how the application dependent layer can be developed using the DVSA functions, we present SHOB, SHared OBjects, a package that defines a release consistency model well suited for numerical iterative algorithms. The caching and prefetching strategies supported by SHOB have been designed to exploit at best this consistency model to minimize the cost of accesses to shared data. Another package implemented on top of DVSA is HPF-Share [4], that defines a consistency model to support HPF applications.

The decomposition of the DSM into two layers simplifies the porting of the second layer, and of the corresponding programming tools, onto any architecture where the DVSA package has been implemented.

Notice that the second layer is not implemented by the application developer. Rather, it is the developer of the DSM that implements this layer to offer a given consistency model. The application developer should choose the consistency model and, consequently, the most appropriate second layer for the application of interest.

The rest of the paper is organized as follows. Sect. 2 presents the main DVSA features, sect. 3 presents the SHOB package and its implementation. Sect. 4 presents some experimental results of an uniform multigrid method developed through SHOB.

2 DVSA

Distributed Virtual Shared Areas, DVSA, is a package that provides a shared memory abstraction on distributed memory parallel architectures. Similar abstractions of a shared memory are TreadMarks [1], HIVE [3], Munin [5] and Tempest and Typhoon [10]. The DVSA functionalities are simpler than those of these systems because, while these packages have been developed to support user applications, DVSA is application independent and its goal is the development of application dependent run time supports. For this reason, DVSA offers a wide range of primitives to access areas and to synchronize processes, but it does not impose a given consistency model. In our approach, this model as well as the corresponding caching and prefetching strategies are defined and implemented by a second layer, that depends upon the applications of interest.

DVSA defines the shared memory as a set of areas. An area is a sequence of contiguous memory locations and it is paired with an identifier. Each process can access an area through the corresponding identifier, independently of the actual physical allocation of the area itself. Each area is considered as an atomic entity, and all its locations are either read or updated by each DVSA operation. In order to guarantee the correctness of the accesses, these areas can be managed through the DVSA primitives only, see Tab. 1.

To efficiently execute the package on different architectures, the size of each area may be freely chosen in an architecture dependent range. In fact, the hardware/firmware support of an architecture determines a range of area sizes that the architecture itself can efficiently support. The lower bound of this range is the most critical one from the performance point of view, because it defines the smallest amount of data exchanged when accessing an area. The lower bound has to be chosen according to architectural parameters such as the time to set up a communication and the bandwidth of the interconnection network. For instance, in the Meiko CS2, that includes specialized co-processors for message exchange and a fast interconnection network, this value is lower than that of a COW, that provides little hardware/firmware support for data exchange. On the other hand, to minimize problems such as false sharing, the largest size of an area should be chosen according to the semantic of the application too, according to the data structures recorded into an area.

The DVSA primitives can be partitioned into four subsets: notification, access, synchronization and utility.

The *notification* primitives initialize and terminate the DVSA support. At initialization time, each process declares, through the *Share* primitive, all and only the areas it is willing to share. Both the physical allocation, i.e. the local memory of the processing node, p-node, where the area is mapped, and the size of each area are fixed at this time and cannot be changed at run time. The physical allocation of an area can be chosen either by the user or by the built-in DVSA allocation procedure.

The DVSA *synchronization* primitives support the coordination of processes operating on the same area. A *Lock* primitive issued by a process P delays synchronized accesses from other processes to the area until the corresponding *Unlock* primitive is issued by P. A *Lock(read,A)* primitive delays any update of A, while a *Lock(write,A)* delays any access to A.

The *utility* primitives return information on the areas. The *AreaInfo(A)* primitive returns the size and the local memory where A is stored. The *MyAreas* primitive returns the list of the identifiers of the areas declared by the invoking process P and of the areas allocated in the local memory of P.

Four *access* primitives are available: *Read* and *Write*, to, respectively, read and write an area, *Copy*, to copy an area into another one and *Read_Write*, to read an area before updating it. The synchronization protocols and the kind of termination of each access primitive can be chosen through the parameter m, see table 1. In terms of the synchronization protocol, the DVSA accesses are partitioned into *synchronized* and *non synchronized* while, from the point of view of termination, they are partitioned into *blocking* and *non blocking*.

A *synchronized* access may be delayed when other synchronized accesses on the same area are concurrently executed. An update is delayed if at least one other process is accessing the same area. A synchronized read, instead, is delayed iff another process is updating the same area. Hence, several processes can read the same area simultaneously, but only one process can update it. As soon as a synchronized access on an area is terminated, pending accesses on the same

Table 1. DVSA Primitives.

Notification primitives	
Share(n,L(A_i))	declares a list of n areas $A_0, A_1, ...A_{n-1}$
End()	free all the areas
Access primitives	
Read(m,A,b)	reads the area A into the buffer b
Write(m,A,b)	writes the buffer b into the area A
Copy(m,A,B)	copies the area A into the area B
Read_Write(m,A,b,c)	reads the area A into the buffer b, writes the buffer c into A
Synchronization primitives	
Lock(w,A)	delays synchronized access on A
Unlock(A)	permits synchronized access on A
Test(h)	test if the operation returning h is terminated
Wait(h)	waits till the operation returning h is terminated
Utility primitives	
AreaInfo(A)	returns information on A
Exist(A)	returns 1 if A exists
MyAreas()	returns information about the areas of the process

area are considered. Write accesses have priority over the read ones. Then, if both read and write accesses have been delayed, a write one is executed first. This scheduling algorithm is unfair, but we assume that the average number of delayed accesses is so low that fairness is not an issue. If no write accesses are pending, then all read accesses are concurrently executed. A *non synchronized* access, instead, is immediately executed even if other accesses, synchronized or non synchronized, are in progress on the same area. Hence, this kind of access can be safely used only if either the semantics of the application enables concurrent accesses, or if proper process synchronizations in the application guarantee the mutual exclusion on the area.

A *blocking* access terminates only after the access has been completed. A blocking *Read* terminates as soon as the shared area has been copied into the local buffer, while a blocking *Write* terminates as soon as the local buffer has been copied into the shared area. Instead, *non blocking* accesses terminate immediately, returning an handle h that identifies the issued operation. To test the status of a non-blocking access, the DVSA library defines two operations on the handle h: *Test(h)*, to test whether the access is terminated, and *Wait(h)*, to wait till the access has been terminated. Non blocking accesses are used to overlap the execution of the application with the DVSA accesses.

As far as a single area is concerned, the DVSA provides a sequential consistency memory model [9]. The next section shows how an alternative consistency model can be implemented on top of DVSA.

3 SHOB

The SHared OBject, SHOB, package is an application oriented support corresponding to the second layer of our DSM system. It implements operations on shared data structures according to a release consistency model that is well suited for numerical iterative algorithms. To minimize the overhead due to remote accesses, i.e. accesses to data stored into a remote memory, SHOB implements caching and prefetching strategies suited to this model. In the following, we describe the implementation of these strategies and their effectiveness for numerical iterative applications. Furthermore, we focus on shared structures that are two dimensional matrices.

3.1 SHOB Primitives

SHOB requires that each process declares, through the primitive *Declare*, all and only the structures that it is going to share. A structure is shared only among the processes that have declared it, i.e. a process that does not declare a structure S cannot access it but avoids any overhead related to S. The primitive *Declare* returns an handle that has to be specified by any process on any p-node to access the structure. In the case of two dimensional matrices, one of two allocation strategies is chosen at declaration time. The strategies differ in the definition of the *allocation unit*, i.e. the set of elements that are allocated in the same DVSA area. The first strategy maps K columns, or rows, into a DVSA area. The second strategy partitions the matrix into KxK square submatrices and maps each submatrix into a DVSA area. The most appropriate strategy depends upon the application behavior. The value of K and the physical allocation of the data, i.e. the p-node where each allocation unit is stored, can be chosen either by the programmer in the declaration or by the SHOB support and cannot be updated at runtime. As shown in Tab. 2, SHOB defines three access primitives to handle a matrix: *read*, *write* and *sync*. The *read* and the *write* primitives work on one element of the shared matrix and they are executed in a non synchronized mode. Hence, the same element can be accessed in parallel by several processes. SHOB does not supply any mechanism to implement mutual exclusive accesses. Moreover, the *write* primitive updates the values of shared data in an asynchronous mode. Any process synchronization exploits a *sync(M)* primitive, that implements a barrier among all the processes sharing M. The barrier can be passed only when all the pending accesses on M have been terminated and all the processes have flushed back their local caches. No consistency

Table 2. SHOB Primitives.

Access primitives
Read(M,i,j,b) reads M[i][j] into the local variable b
Write(M,i,j,b) copies the local variable b into M[i][j]
Sync(M) completes the update of M

problem arises because of the updates, provided that distinct processes updates distinct allocation units of *M*. This corresponds to a release consistency model where the number of operations on M in between two successive invocations of *sync(M)* determines the trade off between data consistency and synchronization overhead.

3.2 SHOB Implementation

The SHOB handle of a shared matrix M is implemented as pointer to a structure, replicated in each p-node. The structure records the identifier of the DVSA area containing the first allocation unit of *M*, the adopted allocation strategy the information to implement the caching and prefetching strategies and the identifier of the DVSA area to implement *sync(M)*. The allocation units of a shared matrix are DVSA areas whose identifiers are contiguous. In the case of programmer-defined allocation, the DVSA library manages the physical allocation of the areas according to the programmer directives. When an element of *M* is read or written, the SHOB support computes the identifier of the area where the element is stored, using the identifier of the initial DVSA area and the allocation method of *M*.

We describe now the implementation of a *SHOB read operation* and the caching and prefetching strategies. The first time a process P_h invokes a SHOB read on an element *M[i][j]* of a shared matrix, the DVSA area including *M[i][j]* is copied, through a DVSA read, into the cache C_h in the local memory of P_h. If, later on, P_h invokes a SHOB read on an element *M[k][l]*, $i \neq k$ or $j \neq l$, that belongs to the same area, SHOB returns the value of this element from C_h. As soon as all the elements of the area have been read from C_h, the SHOB support automatically updates C_h by starting a non blocking DVSA read operation on the corresponding area. In this way, all the elements in the area are *prefetched*. Moreover, between two consecutive SHOB read operations on the same element, the value of the cache is always updated through a DVSA read operation. The overhead of these strategies is very low, because most read accesses are overlapped with the application.

The implementation of a *SHOB write operation* is similar to that of a read one. When a process P_h writes *M[i][j]*, if a copy of the area containing this element is not present in its local cache C_h, then P_h copies the area in C_h through a DVSA read and it updates the local copy of *M[i][j]*. The updated value of the area will be copied back through a DVSA write into the corresponding area either when a $sync(M)$ is invoked or as soon as all the elements of the area have been updated in C_h. In the latter case, a non blocking DVSA write updates the area in parallel with the user computation.

sync(M) is implemented through DVSA operations on a further DVSA area paired with *M*. When a $sync(M)$ is issued, any copy of an area that includes at least an updated element of M, is copied back into the area, provided that SHOB has not autonomously issued such an update. The update is implemented through a DVSA write. The application is suspended till all pending updates are terminated.

The goal of these caching strategies is to fully exploit any value in the local cache and to access through a simple operation any data in a DVSA area. To take into account updates to the area due to other processes, SHOB updates through a DVSA read any copy of a DVSA area in the local cache as soon as it any data in the copy has been read. However, some updates of an element may be lost, because the corresponding DVSA area may be read just before it is updated by another process. Hence, the consistency model guarantees that only the last update of an area before a *sync(M)* will be seen by the other processes. These strategies try to exploit at best the properties of numerical iterative algorithms, that update each element of a shared data only once for each iteration. Furthermore, due to their iterative behaviour, these algorithms can easily tolerate the delay of some updates. Obviously, to fully exploit these strategies, the allocation units should be chosen according to the application behaviour.

4 Experimental Results

To evaluate on a real application the performances and the effectiveness of the DSM that composes DVSA and SHOB, we have implemented a parallel version of an non adaptive multigrid method. Before discussing the performance of this application, we consider that of the DVSA primitives. Currently, two versions of DVSA have been developed: one on the Meiko CS2 and the other on a COW. The CS2 system is a tightly coupled network of HyperSPARC (100 Mhz) processors with 128 Mbyte of local memory running the Solaris operating system. Processors are interconnected by a multi-stage switched network optimized for high performance inter-processor communications. Each workstation of the COW is a PC with an Intel Pentium II CPU (266 MHz) and 128 Mbytes of local memory and it runs the Linux operative system. The interconnection network is a 100Mbit Fast Ethernet Switch.

In the Meiko CS2, the time to read/write an area ranges from 10 μ sec for a 1 Kbyte local area to 160 μ sec for a 16 Kbyte remote area. In the COW architecture, instead, the times to read/write an area ranges from 2 μ sec for 1 Kbyte local area to 2 msec for a 16 Kbyte remote area. In the CS2 architecture, for the same area, there is a one order of magnitude difference between the times of, respectively, a local access and a remote one. In the COW architecture, instead, there is a two orders of magnitude difference.

To achieve satisfactory performances on such a broad range of architectures, the effectiveness of the caching and prefetching strategies supported by the consistency model of interest is fundamental. Furthermore, the ability of implementing these strategies starting from an architecture independent level, as the one defined by the DVSA, strongly simplifies the implementation.

The next step of our experimentation investigates the performance of a uniform multigrid method on the Meiko CS2 and on the COW. The implementation uses the SHOB library primitives embedded in the C programming language.

4.1 The Performance of a Uniform Multigrid Method

Multigrid methods solve partial differential equations (PDE) by discretizing the domain through a hierarchy of grids. The non adaptive version uses a statically defined hierarchy, [2], where each grid is a discrete representation of the whole domain at a distinct abstraction level. The grids at the higher levels of the hierarchy use a larger number of points than those at the lower levels. The highest level grid is the *finest grid*, while the lowest level one is the *coarsest grid*. Several operators, multigrid operators, are applied to the grid hierarchy in a predefined order, V-cycle. Each operator updates the current value of each point p of each grid g using the values of the neighbors of p. The neighborhood stencil of p depends upon the considered operator and it may include points on the same grid of p or on the grids above or below in the hierarchy [6]. The V-cycle is iteratively applied on the grid hierarchy until the current error on the finest grid is lower than a fixed threshold. The discrete solution of the PDE is represented by the values of the points in the finest grid. We apply the method to the Poisson problem, i.e. the Laplace equation along with the Dirichlet boundary condition:

$$-\frac{d^2u}{dx^2} - \frac{d^2u}{dy^2} = f(x,y) \qquad in \qquad \Omega =]0,1[\times]0,1[$$
$$u = h(x,y) \qquad in \qquad \delta\Omega$$

with $f(x,y) = 0$ and $h(x,y) = 10$.

In the experiments, the finest and the coarsest grids are implemented by, respectively, a 1024x1024 matrix and a 256x256 one. Our implementation represents each grid as a shared matrix that is allocated by mapping K columns into each DVSA area. If n is the number of columns, p the number of processes, we assume that K divides n/p and map onto the local memory of P_h all the areas storing the matrix columns from $(n/p) * h$ to $(n/p) * (h+1) - 1$. To port the application, we only had to change the value of K, the number of columns in a DVSA area.

Because of the neighborhood stencil of the operators, this allocation guarantees that most of the accesses to a shared matrix are served by the local cache. To show this, consider that each P_h applies the multigrid operators to all the columns mapped onto its local memory and that all the elements of a column c are updated before considering the next column. To update an element e in c, P_h needs the values of the neighbors of e in $c-1$ and in $c+1$. When P_h reads the first element of $c-1$ or of $c+1$, SHOB copies the DVSA area containing the corresponding column into the cache of P_h. Hence, P_h finds in its local cache all the values to update all the other elements of c. Moreover, SHOB transparently copies back the values from the local cache to the proper DVSA area as soon as all the elements of the column have been updated. Even if SHOB cannot guarantee that all the updates to an element will be seen by a process reading this element, the solution of the PDE can be reached even if some processes use an out-of-date value. However, the use of out-of-data values results in a larger number of iterations to compute the solution. This does not imply

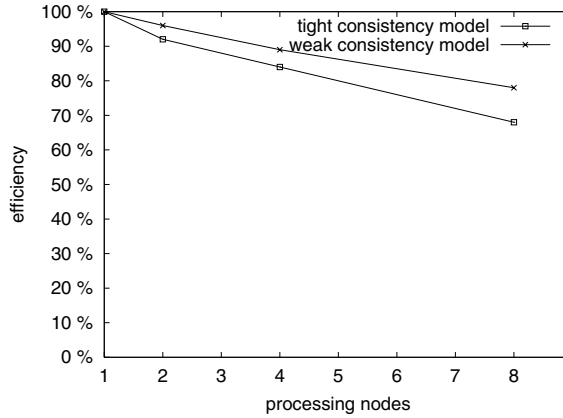

Fig. 1. Efficiency of the Uniform Multigrid Method on the CS2 Architecture.

that more time is required to compute the solution, because the time due to synchronizations may be larger than the one due to the additional iterations. To investigate the execution times of the multigrid method as an increasing degree of inconsistency is allowed, we have considered two versions that define the range of solutions supported by the consistency model of SHOB. In the version with the lowest degree of inconsistency, tight version, the processes issue a $sync(A)$, for each matrix A, after updating a column of A. In the version allowing the largest degree of inconsistency, weak version, the processes issue a $sync(A)$ only at the end of each V-cycle.

The experiments confirm that both versions produce the same numerical results using a different number of V-cycles. The tight version produces the final results in 50 V-cycles, while the weak version requires at most two more V-cycles to produce the same results. Figure 1 shows the experimental results of the CS2 implementation, whereas Fig. 2 shows the results of the COW implementation. In the both platforms, the weak version results in a better performance.

While the best performance has been achieved on the CS2, the performance increase achieved by the weak version is larger in the COW architecture. In the case of the CS2 the difference between the performances of the two version is negligible, whereas in the COW architecture an iteration of the weak version takes 65% of the time of an iteration of the tight version.

These results show the effectiveness of a two layers DSM: it is important to have an application independent layer, that optimizes the basic functionalities to access shared data, according to the tools and the hardware support of the architecture. However, an application dependent layer is important too, because it specializes caching and prefetching strategies to hide the latency to read and update remote data according to the consistency model most appropriate for the class of applications.

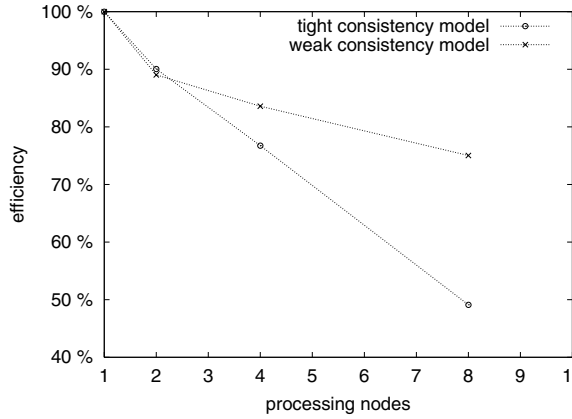

Fig. 2. Efficiency of the Uniform Multigrid Method on the COW Architecture.

References

1. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer **2**(29) (1996) 18 – 28
2. Baiardi, F., Chiti, S., Mori, P., Ricci L.: Parallelization of Irregular Problems Based on Hierarchical Domain Representation. Proceedings of HPCN 2000: Lecture Notes in Computer Science 1823 (2000) 71 – 80
3. Baiardi, F., Dobloni, G., Mori, P., Ricci L.: Hive: Implementing a virtual distributed shared memory in Java. DAPSYS - 3rd Austrian Hungarian Workshop on Distributed and Parallel Systems Kluwer Press (2000)
4. Baiardi, F., Guerri, D., Mori, P., Moroni, L., Ricci, L.: Evaluation of a Virtual Shared Memory by the Compilation of Data Parallel Loops. 8th Euromicro Workshop on Parallel and Distributed Processing (2000)
5. Bennett, J.K., Carter, J.B., Zwaenepoel, W.: Munin: Distributed shared memory based on type-specific memory coherence. ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (1990) 168 – 176
6. Briggs, W.: A Multigrid Tutorial. SIAM (1987)
7. Coelho, F., Germain, C., Pazat, J.L.: State of Art in Compiling HPF. In The parallel programming Model: Foundations, HPF Realization and Scientific Application, Lecture Notes in Computer Science 1132 (1996)
8. Gupta, S.K.S., Kaushik, S.D., Sharma, S., Huang, C.H., Sadayappan, P.: Compiling Array Expression for Efficient Execution on Distributed-Memory Machines. Journal of Parallel and Distribute Computing **32** (1996) 155 – 172
9. Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. on Computers **C-28**(9) (1979) 690 – 691
10. Reinhardt, S.K., Larus, J.R., Wood, D.A.: Tempest and Typhoon: User-level Shared Memory. Int. Symposium om Computer Architecture (1994) 325 – 336
11. Stichnoth, J.M., O'Hallaron, D., Gross, T.R.: Generating Communication for Array Statements: Design, implementation, and evaluation. Journal of Parallel and Distribute Computing **21**(1) (1994) 150 – 159