DVSA and SHOB: Support to Shared Data Structures on Distributed Memory Architectures

F. Baiardi D. Guerri P. Mori L. Moroni L. Ricci
Dipartimento di Informatica
Universitá di Pisa
Corso Italia 40, 56125 - Pisa (Italy)
{baiardi, guerri, mori, ricci}@di.unipi.it

Abstract

With reference to numerical iterative algorithms, this paper exemplifies a methodology to design the runtime support of applications sharing a set of data structures on a distributed memory architecture. According to the methodology, the support is decomposed into two layers: an application independent one, supplying the basic functionalities to access a shared structure, and an application dependent layer that implements the caching and prefetching strategies most appropriate for the considered application. Starting from this assumption, we introduce DVSA, a package that implements the application independent layer and SHOB, one of the packages that can be developed on top of DVSA. SHOB defines a weak consistency memory model where the user controls the amount of inconsistency due to caching and prefetching. The model is well suitable to implement iterative numerical algorithms. Experimental results of the methodology are presented in the case of an uniform multigrid method to solve partial differential equations.

1 Introduction

In most parallel applications, the interactions among a set of processes either involves the sharing of some data or may be naturally described in terms of shared data. Those architectures that do not provide a physical support for shared memory implement these interactions through a software layer.

We propose a methodology that decompose this software layer into two hierarchical layers:

- an application independent layer that includes the basic functionalities to define and access a shared data as well as to synchronize the accesses;
- an application dependent layer built on top of the previous one, to implement the caching and prefetching strategies most appropriate for the application of interest. These policies are fundamental for an efficient implementation, but they imply the adoption of a proper weak consistency memory model that, in turn, depends upon the application of interest.

Assuming this methodology, we present two packages, DVSA, Distributed Virtual Shared Areas, and SHOB, SHared OBjects. DVSA is the package that implements the application independent layer, while SHOB is one of the packages that can be built on top of DVSA. In particular, SHOB implements a weak consistency model where the user can choose the amount of allowed inconsistency between multiple copies of a shared data structure.

As implied by its name, the DVSA package structures the shared memory as a set of shared areas. Each area stores a set of related data and each operation on the shared memory involves a whole area. The package defines a set of functions that can be composed by higher level virtual machines, like SHOB, to implement caching and prefetching policy. Currently two implementations of the DVSA package have been developed: one on the Meiko CS2 and the other on a cluster of workstations. Obviously, both implementations define the same interface, to achieve a complete portability of the applications developed on top of the package. The implementation of DVSA on the Meiko CS2 has been described in [2]; this paper is focused on the implementation on a cluster of workstations.

SHOB implements a weak consistency memory model, but it can emulate also a sequential consistency model, because the shared data structure can be updated at predefined moments [6, 7]. Moreover, it relieves the developers from the task of mapping the data structures onto the physical memories of the processing nodes, p-nodes. Besides simplifying the access to a shared structure, by proper exploiting the weak consistency model, SHOB implements caching and prefetching strategies and it allows the application to

1066-6192/01 $10.00 © 2001 IEEE

choose an optimal compromise between the consistency of the copy of a data structure in the local cache and the overhead to update the cache itself. Due to the memory consistency models that can be supported by the SHOB primitives, the package is well suitable to easily and efficiently implement numerical iterative methods because they can easily overcome some degree of inconsistency of data in the local cache.

The rest of the paper is organized as follows: section 2 presents the main features and the functionality offered by the DVSA library; section 2.1 describes the DVSA primitives and section 2.2 describes the implementation of the DVSA library on a cluster of workstations. Section 3 presents the SHOB library; the SHOB primitives are described in section 3.1 while section 3.2 describes the implementation on top of the DVSA. Section 4 presents some experimental results of a parallel implementation of an uniform multigrid method for the solution of partial differential equations on a cluster of workstations using the SHOB library.

2 DVSA: Definition and Implementation

Distributed Virtual Shared Areas, DVSA, is a package that provides a shared memory abstraction on distributed memory parallel architectures. Similar abstraction of a shared memory on distributed memory architectures are ThreadMarks [1], Munin [5], Tempest and Typhoon [9] and HIVE [4]. While these packages have been developed to efficiently support user applications, the goal of DVSA is the development of application run time supports. The implications of this approach are discussed at the end of the next section.

2.1 DVSA Functions

DVSA defines a shared memory consisting of a set of areas. A shared area is a sequence of contiguous memory locations and is paired with an identifier; each process can access an area through the corresponding identifier, independently of the actual physical allocation of the area. Each area is considered as an atomic entity, and all the values stored in the area are either read or updated by each DVSA operation. In order to guarantee the correctness of the accesses, areas can be managed by the DVSA primitives only.

To efficiently execute the package on different architectures, the size of an area may be chosen in an architecture dependent range. In fact, the hardware/firmware support of an architecture determines a range of area sizes that can be efficiently supported by the architecture itself. In particular, the lower bound of this range is the most critical from the performance point of view, because it defines the smallest amount of data exchanged when accessing an area. For instance, in the Meiko CS2, where each p-node includes a coprocessor for message exchange and a fast interconnection network is available, the lower bound is lower than the one of a cluster of workstations, that provides little support for message exchange. On the other hand, in order to minimize problems such as false sharing, the size of an area must be chosen according to the semantic of the application too, taking into account the data structures recorded into one area and the typical behavior of the application itself. Porting of an application developed on top of DVSA is possible, provided that the target architecture supports the sizes of the areas that have been used in the application.

All the areas shared by the DVSA package have to be declared in advance. Each process declares only the areas it is willing to share. Both the physical allocation and the size of each area are fixed at this time and cannot be changed during the computation. The physical allocation of each area, i.e. the local memory of the p-node where the area is mapped, can be chosen either by the programmer or by the DVSA allocation procedure.

The DVSA primitives are described in table 1. The notification primitives are used to initialize and terminate the DVSA support. The synchronization primitives provide mechanism to explicitly synchronize processes operating on the same area. The utility primitives provide information on the areas. The access primitives implement the accesses to the areas; each of these primitives can adopt a different synchronization protocol and a different kind of termination. In terms of the synchronization protocol, we can distinguish among synchronized and non synchronized accesses, while in terms of the kind of termination, we can distinguish among blocking (synchronous) and non blocking (asynchronous) accesses. Hence, each access operation in table 1 corresponds to four DVSA operations.

A synchronized access is managed according to the operation, update or read, and to the existence of other concurrent synchronized accesses on the same area. An update is performed only if no other process is accessing the same area, otherwise the request is stored in a queue paired with the area, *area_queue*. A synchronized read is performed if no other process is accessing the same area as writer, otherwise the request is stored in *area_queue*. Hence, several processes can read the same area simultaneously, but only one process can write it. As soon as a synchronized access is terminated, one of the requests stored in *area_queue* is served. Write requests have priority over the read ones; then, if the *area_ queue* includes both a read and a write request, the write one is executed first, regardless of its position in the queue. If no write requests are waiting in the queue, then all the read requests can be executed simultaneously. A non synchronized access, instead, is immediately executed even if other accesses, synchronized or non syn-

Notification primitives	
Share	declares the areas
End	disallocates the areas
Access primitives	
Read(A,b)	reads the area A into the buffer b
Write(A,b)	writes the buffer b into the area A
Copy(A,B)	copies the area A into the area B
Read_Write(A,b,c)	read the area A into the buffer b and write the buffer c into the area A
Synchronization primitives	
Lock(A)	delays synchronized access on A
Unlock(A)	permits synchronized access on A
ReadLock(A)	delays synchronized updates of A
ReadUnlock(A)	permits synchronized updates of A
Test(h)	test if the operation identified by h is terminated
Wait(h)	waits till the operation identified by h is terminated
Utility primitives	
PagInfo(A)	returns information on A
Exist(A)	returns 1 if A exists
MyPages	returns information about the areas declared and allocated by the calling process

Table 1. DVSA primitives

chronized, are in progress on the same area. Hence, this kind of access can be safely used only if the semantic of the application enables concurrent accesses, or if other synchronizations in the application guarantee the mutual exclusion on the same area. In this way, useless synchronizations at DVSA level are avoided.

A blocking operation terminates only after the access has been completed, i.e. when the shared area has been copied into the local buffer or when the local buffer has been copied into the shared area. Hence, after a blocking read operation, the value in the target buffer can be safely used. Non blocking operations terminate immediately, without returning any information about the status of the operation. Instead, they return an handle that identifies the issued operation. To test the status of a non-blocking operation, the DVSA library defines specific operations on the handle, test and wait. In this case, the value in the target buffer can be safely used only after a positive result of the test operation. Non blocking operations are used to overlap useful computation with the execution of the access operation. As an example, after issuing a non blocking read access, the program can execute some computations concurrently with the access.

While DVSA offers a wide range of methods to access the areas, it does not provide either caching or prefetching strategies. These strategies, along with any other optimization strategy, are to be implemented at higher levels, by the support of specific applications. At the DVSA level, no information is available on the data structures recorded in the shared areas and on the application behavior. Moreover, as far as a single area is concerned, the DVSA provides a sequential consistency memory model [8]. Alternative consistency models can be implemented by properly composing the DVSA functions.

2.2 Implementation on a Cluster of Workstations

This section describes the implementation of the DVSA on a cluster of workstations. In the following, we assume that one process runs on each p-node; we denote by P_h both the h-th p-node and the h-th process.

The DVSA data structures and the areas are allocated in the physical memory when the DVSA initialization function, share, is invoked. In particular, the share function creates the Address Translation Table, ATT, that records, for each shared area, the identifier of the p-node where it is physically mapped and the corresponding physical address. The ATT does not change during the computation and it can be replicated on each process.

To guarantee an efficient and correct computation, each P_h has to simultaneously manage both its accesses to all the areas and the accesses to the areas mapped into its memory from processes running in other p-nodes. For this reason, the DVSA library has been implemented using threads. Moreover, the DVSA library is thread-safe, i.e. the application can generate concurrent threads that use the DVSA primitives. In our implementation, to avoid the overhead due to the TCP protocol and to exploit at best the reliability of the current interconnection structure, threads communicate through UDP sockets.

Besides the threads defined by the application, P_h includes two sets of threads: the *client threads*, that manage both the access requests to local and remote areas issued by the local application, and the *server threads*, that manage the access requests to the local areas issued by threads of P_k, $h \neq k$. To avoid the large overhead of dynamic thread creations, both the numbers of client and server threads is fixed at system start up and all the threads are created by the share function.

The application is executed by a client thread, THap, that can generate other threads. THap manages the synchronous operations on both local and remote areas. If the area to be accessed is stored in the local memory, then THap tries to access the area; if the access requires the synchronization with the other p-nodes, then the *area_queue* is used. If the area is allocated into the local memory of P_k, the THap thread of P_h sends the request to the server threads of P_k, through the server socket of P_k. THap can easily determine where an area has been stored through the ATT. Furthermore, the address of the server socket of each p-node is known by any other p-node. The request sent through the server socket includes the address of a private socket to be used by the thread of P_k to return any operation result.

A set of other n client threads, THas, manage asynchronous operations. THap inserts the request of an asynchronous operation in a queue, the *async_queue*. One thread of THas extracts a request from the *async_queue* and executes it in the same way as THap executes a synchronous operation.

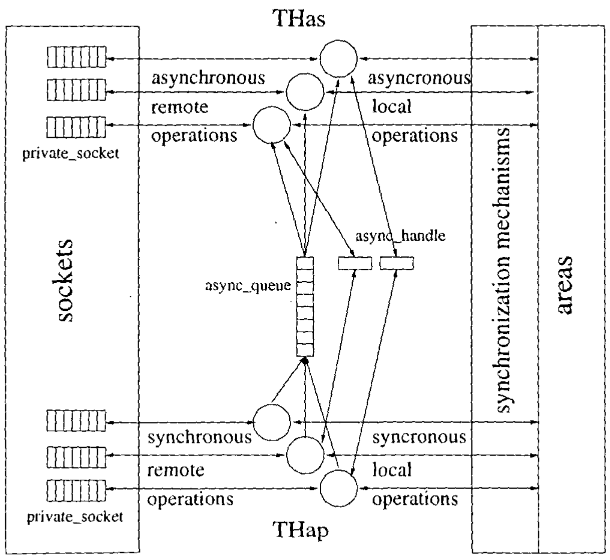

Figure 1. Client threads cooperation scheme

The set of server threads is partitioned into two subsets, THrem and THaw, that include, respectively, the threads that manage the requests of other p-nodes on the local areas and those that complete a suspended operations.

A thread in THrem extracts the requests sent by the other p-nodes from a queue paired with the server socket. If the requested operation can be executed, its results are returned through the private socket paired with the request. Otherwise, the request is stored in the *area_queue*, the operation is suspended and the thread extracts the next request from the queue paired with the server socket. As soon as the operation on an area A has been completed, if the *area_queue* of A is not empty, the THrem thread extracts the first request and records it in the *aw_queue*. One of the THaw threads extracts the request from the *aw_queue* and executes the corresponding operation.

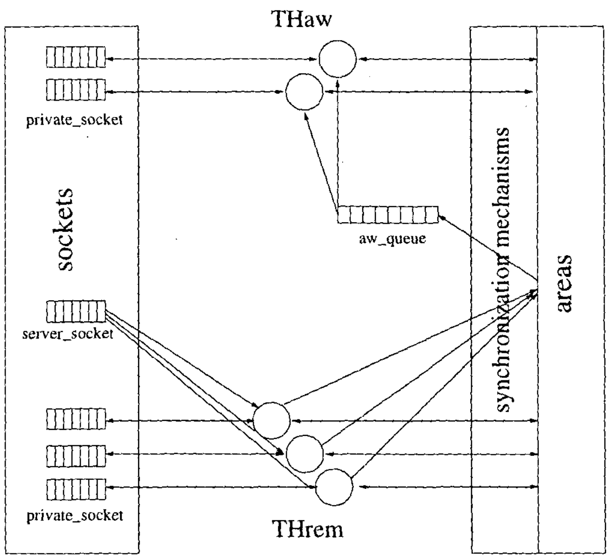

Figure 2. Server threads cooperation scheme

3 SHOB

The SHared OBject package, SHOB, exploits the unsynchronized operations of the DVSA to implement caching and prefetching strategies that can be driven by the SHOB user, that chooses the trade off between consistency and synchronization overhead. Since the SHOB library is built on the DVSA layer and does not require other cooperation mechanisms, the portability of the SHOB package only depends upon the availability of the DVSA library on the target architectures.

3.1 SHOB Functions

The SHOB library enables the programmer to define shared data structures that can be accessed by any process on any p-node by specifying the handle of the structure only. In the following, we focus on two dimensional matrixes. In this case, any element of the matrix declared through the package can be accessed by specifying the handle of the matrix and the indexes of the element.

First of all, the shared matrixes have to be declared. Each process declares all and only the matrixes that it is going to share. A matrix is shared only among processes that have declared it, i.e. if a process does not declare the shared matrix A, it cannot access any element of A but it avoids any computational overhead related with A. The declaration function returns the handle to refer the shared matrix. For each shared matrix, an allocation strategy has to be chosen at declaration time. Two allocation strategies for matrixes are currently supported. The main difference between them is the definition of the allocation unit, i.e. the set of elements that are allocated in the same DVSA area. The first strategy maps an user-defined number of columns, or rows, into a DVSA area. The second strategy partitions the matrix into square submatrixes and maps each submatrix into a DVSA area. Both strategies have been inspired by the HPF blocking distribution strategy. 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. Both the allocation strategy and the physical allocation of the data are static, i.e. they cannot be modified at runtime. Obviously, to obtain the best performances, the allocation strategy and the physical allocation of the data, should be chosen according to the behavior of the application.

As shown in table 2, the SHOB library defines three access primitives to handle shared matrixes: read, write and sync.

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

Table 2. SHOB primitives

The read and the write primitives work on one element of the shared matrix and they are executed in a non synchronized, non blocking mode. Hence, the same matrix element can be accesses in parallel by several processes. The package does not provide any mechanism to implement mutual exclusive accesses. Any synchronization among the processes has to be implemented by the application. In this way, release memory consistency model may be implemented.

The sync(A) primitive implements a barrier among all the processes sharing A. When all the processes have reached the barrier, pending operations on A are completed. By choosing the number of operations on A in between two invocations of sync(A), the programmer chooses a compromise between consistency and synchronization overhead.

3.2 SHOB Implementation

When a shared matrix is declared, the SHOB support returns an handle to a structure, replicated in each p-node, to refer the matrix in the SHOB primitives. The structure records a set of information regarding the matrix, namely the identificator of the DVSA area containing the first allocation unit of the matrix, the allocation strategy adopted and the information to manage the caching and prefetching strategies. The allocation units of a shared matrix are DVSA areas whose identifier are contiguous. In the case of programmer-defined allocation, the DVSA library manages the physical allocation of the areas in the local memories of the p-nodes according to the programmer directives. When an element e of a shared matrix is accessed by a read or write primitive, the SHOB support computes the identifier of the area where e is stored, using the identifier of the initial DVSA area and the allocation method chosen for this matrix. Then, the SHOB package decides, according to the caching and prefetching strategies, whether a remote access has to be executed. These strategies characterize the functionalities of the SHOB package with respect those of DVSA and depend upon the allocation strategies chosen for the shared matrix.

When the process P_h reads, for the first time, the element *A[i][j]* of a shared matrix, the whole allocation unit u including *A[i][j]*, i.e. a DVSA area, is copied into the cache C_h in the local memory of P_h provided that u is not stored in the local memory of P_h. Hence, all the other elements in u have been prefetched. If, later on, P_h reads an element *A[k][l]*, $i \neq k$ or $j \neq l$, that belongs to u, the primitive returns the value of this element from C_h. If another read operation is executed on *A[i][j]*, then the copy of u in C_h is updated by reading again the proper DVSA area. Moreover, as soon as all the elements of u have been read from C_h, the SHOB support automatically updates C_h by starting a read operation on the corresponding DVSA area. Hence, between two consecutive read operations on the same element, the value of the cache is updated. We notice that the overhead introduced by these strategies is very low. In fact, if S, the shared area to be read, is not in the cache memory or has to be reloaded, the only overhead is due to the determination of a free cache position and to the copy of S in the cache. If, instead, S is already in the cache, no remote access is executed.

The strategies adopted for the write operations must consider the problem of the cache consistency. When a process P_h writes the element *A[i][j]* of a shared matrix A, if a copy of the allocation unit u containing this element is not present in the local cache C_h of P_h, then P_h copies u in C_h and it updates the local copy of *A[i][j]*. The updated value of u will be copied back into the corresponding DVSA area either when a sync(A) is invoked or when all the elements in u have been updated in C_h. In the latter case, an asynchronous DVSA access is used to execute the update in parallel with the user operations. In this way, the overhead of the update is overlapped with the user operations. Instead, when a sync(A) is issued, the application is suspended till all updates are terminated. However, if a process P_k, $h \neq k$, reads *A[i][j]* before a sync(A), it could receive an out-of-date value. No consistency problem on A arises provided that distinct processes updates distinct allocation units of A.

Figure 3 shows an example of how a shared matrix is managed through the SHOB operations. In the example, the allocation unit is one row of the shared matrix, i.e. each shared area records one row of the shared matrix A. When the process P_h issues a write operation on the element *A[i][0]*, the DVSA area containing the i-th row is copied into the local cache of P_h and the update is executed in the local cache (figure 3 a). When P_h issues a write primitive on another element of the i-th row, i.e. *A[i][1]*, the update is executed on the previously cached copy of the i-th row (figure 3 b). At this point, all the elements of the i-th row have been updated and the SHOB support flushes the local cache (figure 3 c).

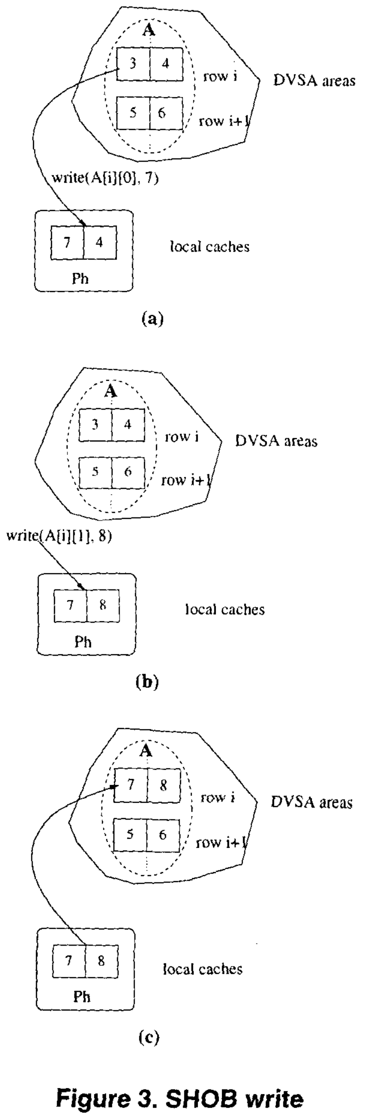

Figure 3. SHOB write

In order to fully exploit these strategies, the allocation unit of a shared matrix has to be chosen according to the application behavior. In particular, the order of the accesses to the elements must be considered. As an example, if the application implements a multiplication among two matrixes, the best strategy allocates each row of the first matrix and each column of the second matrix in a distinct area. In this way, when the process that multiplies the i-th row r_i of the first matrix and the j-th column c_j of the second one, reads the first element of r_i and the first element of c_j, the SHOB support records r_i and c_j in the local cache. From now on, the process finds in the local cache any other element of r_i and c_j.

4 Experimental Results

This section presents some experimental results to evaluate the performances of the DVSA and of the SHOB libraries. The considered architecture is a cluster of workstations; each workstation is a PC with an Intel Pentium II CPU (266 Mhz) and 128 Mbyte of local memory; the interconnection network is a 100Mbit Fast Ethernet switch and the operative system running on each p-node is LINUX.

Figure 4 shows the time to access a DVSA area for synchronous blocking read and write primitives, in the cases of local access, while Figure 5 shows the same results in the case of remote access. Note that the scale of both axis is logarithmic.

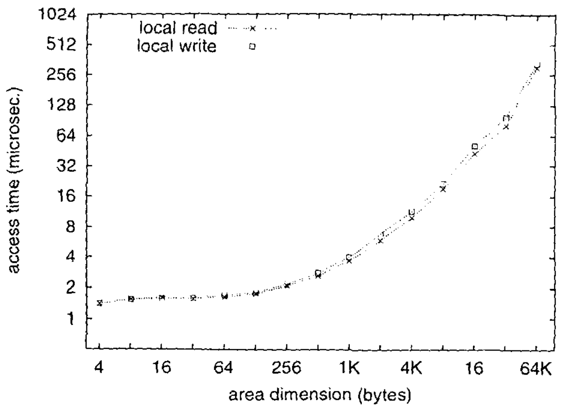

Figure 4. Access time of DVSA primitives on local areas

We notice that the time of a local access is two orders of magnitude different from a remote one. This confirms the importance of caching and of prefetching.

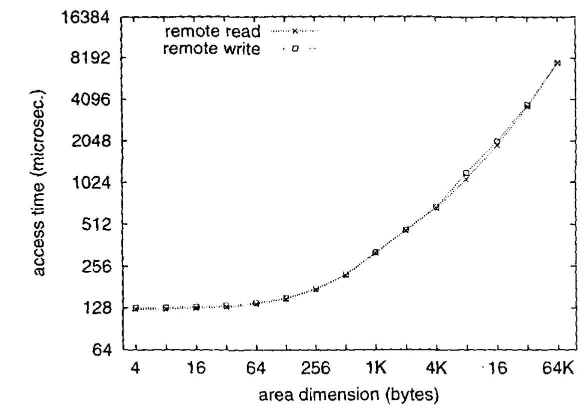

Figure 5. Access time of DVSA primitives on remote areas

4.1 Uniform Multigrid Methods

In order to evaluate the performances and the effectiveness of the SHOB package on a real application, we have implemented a parallel version of an uniform multigrid method.

Multigrid methods are fast iterative methods to solve partial differential equations in two or more dimensions that discretize the equation domain through a hierarchy of grids. Each grid represents the whole domain at a distinct abstraction level; the grids at the higher levels of the hierarchy discretize the domain using a larger number of points than those at the lower levels, because the grid at level l has twice the points on each dimension than the grid at level $l-1$. The highest level grid is called *finest grid* and the lowest level one is called *coarsest grid.* To solve the partial differential equation, a set of operators, multigrid operators, is applied to the grid hierarchy in a predefined order, V-cycle. Each operator, if applied to a grid g, updates the current value of each point p of g using the values of the neighbors of p. The neighborhood stencil of a point p depends upon the specific operator and it may include points on the same grid of p or on the grids above or below in the hierarchy [3]. The V-cycle is iteratively applied on the grid hierarchy until the evaluation of the current error on the finest grid is below a fixed threshold. The discrete solution of the partial differential equation is represented by the values of the points in the finest grid.

We use the multigrid method to solve 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 $h(x,y) = 10$.

Our parallel version of the uniform multigrid method has been implemented using the SHOB library embedded in the C programming language.

In our implementation, each grid of the hierarchy is represented by a shared matrix; each point on the grid correspond to an element in the shared matrix. The allocation strategy chosen for these matrixes maps one column into each shared area, and the areas are allocated in the local memories of the p-nodes in blocks of k, i.e. the columns from $k * h$ to $k * (h + 1) - 1$ are allocated onto the local memory of P_h, where k is the ratio between n, the number of columns of the matrix, and p, the number of processes of our parallel application.

Each P_h applies the multigrid operators to the columns mapped into its local memory. We recall that, to update the value of an element, the multigrid algorithm needs the value of the element itself and the values of its neighbors. In such a way, most of the accesses to the shared matrixes refer to the local memory. As an example, if the neighborhood stencil of e includes the elements whose indexes are obtained by adding or subtracting 1 to at least one index of e, P_h accesses elements stored in a remote memory only to apply the operators to the first and to the last column of the matrix slice allocated in its local memory.

The allocation strategy reflects the order in which the elements are updated by the multigrid method, i.e. if an element in the column c has been updated, then all the elements in c are updated before any element in any other column. To update an element e in c, P_h needs the values of the neighbors of e in $c - 1$ and in $c + 1$. Due to the caching and prefetching strategies, when P_h reads the first element of the columns $c - 1$ and $c + 1$, the whole columns that are not stored in the local memory of P_h are recorded in the cache of P_h. In this way, P_h finds in the local memory all the values to update all the other elements of c. Moreover, the SHOB library transparently tranfers the new values from the cache of P_h to the shared memory as soon as all the elements of the column have been updated.

Even if SHOB does not guarantee that all updates to an element will be seen by a process reading this element, the solution can be reached even if some processes use an out-of-date value because of the iterative behavior of multigrid algorithms. Hence, it is interesting to compare the results of the multigrid method using alternative consistency models.

To implement the sequential consistency model, for each matrix A, we issue a sync(A) after each write operation; to implement a released consistency memory model we issue a sync(A) only when a programmer defined number of operations on A has been issued.

Intuitively, in a iterative algorithm such as a multigrid method, the adoption of a release consistency model results in a larger number of iterations to compute the final solution. This does not imply that the algorithm takes more time to compute the final solution, because the time spent to synchronize accesses to shared data structures can be larger than the one due to the additional iterations.

Figure 6 shows the efficiency of our implementation on a hierarchy with a 1024x1024 finest grid and a 256x256 coarsest one. The efficiency e is defined by the following formula:

$$e = \frac{seqtime}{partime \times numproc}$$

where $seqtime$ and $partime$ are, respectively, the execution times of the sequential and of the parallel algorithm, while $numproc$ is the number of processors used in the parallel execution.

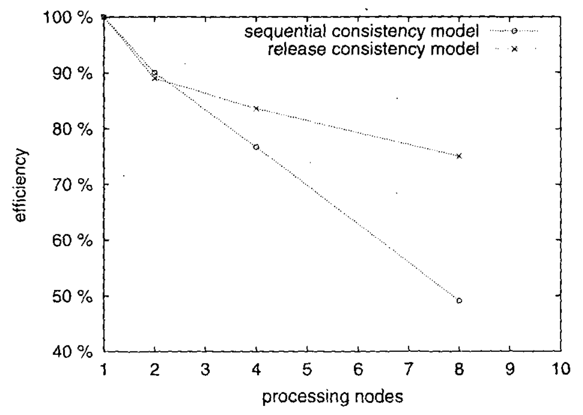

Figure 6. Efficiency of the Uniform Multigrid Method

The figure compares two version of the uniform multigrid method adopting, respectively, a sequential consistency model, and a released one. In the first version each process issues a synchronization after updating one column of a matrix. In the second one, each process issues a synchronization at the end of each V-cycle, i.e. after updating all the matrixes. The two versions produce the same numerical results, using a different number of V-cycles, respectively 50 and 52.

The figure shows that the performances are considerably better in the case of a released consistency model. In fact, the sequential consistency model achieves an efficiency of 50% on 8 p-nodes, while the released consistency model achieves an efficiency of 75% on the same number of p-nodes. This is due to the fact that an iteration in the released consistency model takes 65% of the time of an iteration in the sequential consistency one and the released consistency algorithm requires only two additional iterations to produce the same result of the sequential consistency one.

Finally, the effort to convert the sequential multigrid code into the parallel one using the SHOB library has been modest. In fact, we have modified the matrix declaration, chosen the allocation strategy and the physical allocation of the columns and we have added the synchronization operations. Also the syntax of the read and write operations has been modified, but this is a trivial substitution.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Threadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 2(29):18–28, 1996.

[2] F. Baiardi, D. Guerri, P. Mori, and L. Ricci. Evaluation of a virtual shared memory by the compilation of data parallel loops. *8-th Euromicro Workshop on Parallel and Distributed Processing*, jan 2000.

[3] W. Briggs. *A multigrid tutorial.* SIAM, 1987.

[4] P. F.Baiardi, G.Dobloni and L.Ricci. Hive: Implementing a virtual distributed shared memory in java. *DAPSYS - 3rd Austrian Hungarian Workshop on Distributed and Parallel Systems*, sep 2000.

[5] W. Z. J.K. Bennett, J.B. Carter. Munin: Distributed shared memory based on type-specific memory coherence. *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 168–176, 1990.

[6] W. Z. J.K. Bennett, J.B. Carter. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 1(3), aug 1995.

[7] P. H. K. Li. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), nov 1989.

[8] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[9] S. Reinhardt, J. Larus, and D. Wood. Tempest and typhoon: User-level shared memory. *International Symposium om Computer Architecture*, 1994.