Phosphorus[1]: Adding Shared Memory to PVM.

Rocío Cabrera-Dantart
Isabelle Demeure
Philippe Meunier

Télécom Paris - Département Informatique.
46, rue Barrault - 75634 Paris Cedex 13
demeure@inf.enst.fr

In this paper we present Phosphorus, a system that can be used together with PVM to provide PVM users with the opportunity to use shared variables.

1 Introduction

With the development of systems such as PVM (Parallel Virtual Machine), parallel programming on networks of computers has gained tremendous popularity. In loosely coupled distributed systems no common physical address space is available for data storage. Two approaches exist to share information among components of a parallel computation which may run on different computers: the **message passing** approach and the **distributed shared memory** approach.

In message passing systems each component of a parallel computation has its own variables. No support exists for sharing variables among components. If a component A needs access to the value of a variable x in a component B, then B has to pack a message in which it places the value of x and to send it to A. This is the PVM approach. By sending and receiving messages, application components can cooperate to solve a problem in parallel.

Distributed Shared Memory (DSM) systems provide a virtual address space shared among components on loosely coupled processors. Various DSM systems such as Munin, Ivy [Li89] and LINDA [Gel88] have been developed ([Nit91] provides references on these systems and several others).

[1] The phosphorus is a mineral that is said to play an important role in the performance of the human memory.

The message passing model is well understood and certainly well adapted to a number of data exchange patterns. However, its complexity increases with the complexity of the data sharing pattern. In distributed shared memory systems, programmers don't have to explicitly manage "data passing" between components. However, adding an extra layer in a system may cause some performance degradation. We believe that the two models are complementary. This is why, we have designed and prototyped the Phosphorus system. Phosphorus can be used together with PVM to provide programmers of parallel, distributed computations with both the message passing and the shared memory facilities.

2 Design choices

Several DSM systems have been developed. Some, like Linda, have gained a lot of popularity. Attempts where made in the past to develop DSM systems around PVM. The doPVM system, for example, enables distributed computing based on shared objects [Har93]. The GLENDA system implements LINDA on top of PVM. Our approach differs from the above mentioned ones in that we want to stay as close as possible to the PVM philosophy. We do not intend to develop "a whole new DSM system", but to complete the set of PVM primitives with data sharing primitives to provide PVM users with the shared memory abstraction.

This led us to the following choices:

(1) Phosphorus should be highly portable and efficient. Portability is achieved firstly by the use of the PVM library and secondly by avoiding the use of operating system dependent functions. Efficiency is achieved by the use of relaxed memory consistency models [Kel92] and by minimizing the amount of communication messages to keep the memory coherent.

(2) The Phosphorus interface should be as similar as possible to the PVM interface.

(3) Phosphorus should be able to evolve as PVM evolves.

When designing a DSM system, several choices have to be made:

(1) The design of our system was greatly inspired by the Munin System [Car93]. In particular, we chose to provide the same types of consistency protocols as implemented in Munin. Consistency protocols are used to keep the various copies of a shared variable coherent[2]. They also determine the sharing pattern (for more details about protocols see section 5).

(2) Another choice concerns the implementation level (hardware, software - with compiler support or with the support of the operating system and/or a library of functions). We chose to implement it as a library of functions on top of PVM.

(3) What is the unit of sharing from a logical point of view and from an implementation point of view? The shared memory unit chosen is the variable. To follow the PVM philosophy, the basic PVM types (those supported through the pack/unpack primitives) are offered as basic shared variable type. Shared arrays of these types may also be declared. In other systems, the unit of sharing is a memory page. This choice effects the performance. Since the unit of sharing is small, the overhead associated with each operation implying communications between the components is high. On the other hand, there is no false sharing and updates are only made when really necessary.

3 Implementation

Phosphorus is structured in a way that is very similar to PVM. Like PVM, the Phosphorus system, is composed of two parts. The first part is a daemon called *phosd* that resides on every machine making use of the shared memory (in practice, a *phosd* is a PVM component). There is a *phosd* daemon per user on each host participating in the user's applications. Any access to shared variables is translated into a PVM message that is sent to the local Phosphorus daemon. This daemon will eventually communicate with other daemons in the system to maintain memory coherence.

The second part of the system is a library. It provides routines to enroll and exit from Phosphorus, declare and free shared variables, read and write, and perform synchronization operations.

Every Phosphorus component must first of all enroll PVM and then enroll Phosphorus. Several Phosphorus components, belonging to the same application, may be running at any time on the same host (they all share the same *phosd*). Any access to shared memory is first directed to the *phosd* on the local host. If data is coherent locally, the request can be carried out. Otherwise, the local *phosd* communicates with remote *phosds*.

4 User interface

4.1 Process Control

int phos_init(void)

enrolls a component in Phosphorus (similar to pvm_mytid()). The call creates a new server (*phosd*) on the local host, if it does not exist.

int phos_end(void)

called to exit Phosphorus (similar to pvm_exit()).

4.2 Variable Declaration

int phos_declare(int desc, int type, int count, int protocol)

used to declare a shared variable. **desc** is a unique identifier given by the programmer as a descriptor for the shared variable; it must be common to every task sharing it. **type** corresponds to the PVM types (byte, cplx, dcplx, double, float,

[2] The terms "coherence" and "consistency" are sometimes used interchangeably in the literature. Following [Nit91], we use "coherence" as the general term for the semantics of memory operations, and "consistency" to refer to a specific kind of memory coherence.

int, long, short, uint, ulong, ushort). **count** indicates the number of items of the given type that are being declared. **protocol** determines the **protocol** associated with this variable. Possible values for the argument protocol are READ_ONLY, CONVENTIONAL, MIGRATORY, WRITE_SHARED and SYNCH (for synchronization variables). For more details on protocols see next section.

int phos_free(int desc)
to be called to free a Phosphorus variable when it is no longer needed in a task.

4.3 Access to Variables

int phos_read(int desc, void *buffer)
called to obtain a coherent value of the variable associated with the descriptor **desc**. It is the user's responsibility to ensure that enough storage is available in **buffer**.

int phos_write(int desc, void *buffer)
called to update the value of the shared variable identified by **desc** with that pointed to by **buffer**.

4.4 Synchronization

int phos_lock(int desc)
desc may be either the descriptor of a synchronization variable (SYNCH) or that of a shared variable (READ_ONLY variables may not be locked). The call blocks until the caller is granted the lock.

int phos_unlock(int desc)
liberates a previously acquired lock.

int phos_barrier(char *group, int count)
like a PVM barrier, it blocks the calling task until all tasks in the **group** have reached it. When a barrier is reached, all the shared variables are updated.

5 Consistency protocols

This system enforces two consistency protocols: **strict consistency** where each read operation returns the most recently written value and **release consistency** where the programmer enforces consistency using synchronization primitives.

Phosphorus implements the four protocols implemented in Munin. Three of them follow the strict consistency model (MIGRATORY, CONVENTIONAL and READ_ONLY) and one of them the release consistency model (WRITE_SHARED). In any case, the user is not aware of data replication nor migration; shared memory accesses are done in the same way as local accesses.

5.1 MIGRATORY

In this case, the variable is migrated to the host from which it is accessed. There is a unique owner of the variable at a time, and the owner is the only one who may read from or write to the variable. This protocol, also called "Single Reader/Single Writer" (SRSW), is the access pattern typical of shared data that is accessed only inside a critical section.

5.2 CONVENTIONAL

In this case, one host (the owner) has a read-write copy of the variable and multiple hosts have read-only copies. This consistency protocol is also called "Multiple Readers/Single Writer" (MRSW).

A read operation on a host with no read access rights, causes a read fault.In this case, the faulting host has to communicate with the owner host to acquire a read-only copy. The owner then changes the access rights to read-only, if necessary, and sends a read-only copy to the reader hosts.

A write operation on a host with no write access rights, causes a write fault. In this case the ownership is transferred to the host where the write fault occurred. At this time, an invalidate message is sent to every host keeping a read-only copy before the write operation can complete.

Once a host has acquired the ownership of a conventional variable, it does not reply to requests from other hosts for a period of time (100 msecs). This is done to avoid data ping pong between writers.

5.3 WRITE_SHARED

Multiple copies with read-write access rights may reside on different hosts. This protocol, also called "Multiple Readers/Multiple Writers" (MRMW), is implemented with a Release Consistency Protocol (Lazy Hybrid Release Consistency Protocol). Each task may modify locally a portion of the variable. It is important for the programmer to be aware that each task must write to independent portions of the variable. Updates occur whenever a task acquires a lock or arrives at a barrier.

5.4 READ_ONLY

In this case, once the variable has been initialized, no further updates occur. Thus, the consistency protocol simply consists of replication on demand with no write access rights.

6 Early performance results

In order to compare the relative performance of PVM and Phosphorus, we have implemented the same parallel algorithm in both systems. The chosen algorithm is the merge sort algorithm. In our version of merge sort, the list to be sorted out is divided into a number of sublists each of which can be sorted out by a different worker component. A master component then merges the resulting sublists. In the PVM version, the master maintains a list (called the job list) of the sublists that have not been sorted out yet. In the Phosphorus version, both the job list and the various sublists are shared variables.

We have measured the execution times in both cases. Figure 1 shows some measurements made for 20 sublists. We have observed that the two implementations give very similar results. However, we need to go further with these measurements and to experiment with other applications before we can draw any general conclusion.

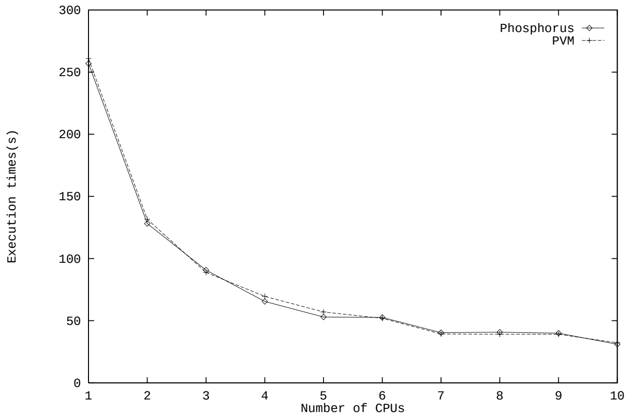

Figure 1: Merge Sort - Execution times - PVM vs Phosphorus

7 Related work

An earlier version of PVM included shared memory features [Sun90]. One of the main advantages provided by this approach is the way in which shared data is allocated. Read and write accesses are performed directly on the shared segment, located at daemon level. In other words, memory for shared variables does not have to be allocated twice at user's level and at daemon level. However, it is necessary to protect every access to a shared segment by a lock; this may cause a segment contention when several accesses to the same argument are carried out at the same time. In addition, PVM shared memory provides a unique shared data access pattern (i.e. only one protocol exist).

Another project that is closely related to ours is ADSMITH [Wen94], a system providing DSM on top of PVM. An advantage of ADSMITH over Phosphorus, is that it provides the possibility to retrieve parts of a variable and it supports shared structures. In contrast, ADSMITH variable ownership and memory coherence management is fixed. This may cause more traffic on the network and a bottleneck in the central daemon. Another disadvantage of ADSMITH is that is structured in a three layer memory system (process, cache and daemon). Therefore, variables may be triplicated on the three levels which can saturate the main memory.

In conclusion, we must say that Phosphorus was carefully designed with efficiency in mind. Four consistency protocols were adapted to suit four different access behaviors, one of them being a relaxed memory consistency protocol. A distributed shared memory manager and numerous low level implementation improvements were developed to reduce network traffic. We therefore expect better performance than those of the previously mentioned systems (as seen in the previous section, the preliminary performance measurements are encouraging).

8 Status of the project and future work

At the time this abstract is being written, a prototype of Phosphorus has been developed on SUN 4 Workstations (only the C version of the library exists). It implements all protocols except for the WRITE_SHARED one. It is currently being tested and we are developing applications to get some idea of its performance.

Future work includes (but is not limited to) completing the development and tests, providing support for shared structures, experimenting with other consistency protocols, tuning the system to improve performance and comparing Phosphorus to other DSM systems.

References

[Car93] J. B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency.* PhD thesis. Rice University. September, 1993.

[Gel88] David Gelernter *Getting the Job Done. "LINDA" a parallel-programming language, is easy, efficient, and portable.* Byte, November 1988.

[Har93] C.L. Hartley and V.S. Sunderam. *Concurrent Programming with Shared Objects in Networked Environments.* Papers related to PVM, obtained through netlib.

[Kel92] P. Keleher, A. L. Cox and W. Zwaenepoel. *Lazy Release Consistency for Software Distributed Shared Memory.* March 9, 1992.

[Li89] K. Li and P. Hudak. *Memory Coherence in Shared Virtual Memory Systems.* ACM Transactions on Computer Systems, Vol. 7, No. 4, pages 321-359. November, 1989.

[Nit91] B. Nitzberg and V. Lo. *Distributed Shared Memory: A Survey of Issues and Algorithms.* Computer, Vol 24, No 8, pages 52-60. University of Oregon. August, 1991.

[Sun90] V.S. Sunderam. *PVM: A framework for parallel distributed computing. Concurrency: Practice and experience.* Vol 2(4), pages 315-339. Department of Math and Computer Science. Emory University. Atlanta. 1990.

[Wen94] Wen-Yew Liang. *ADSMITH: A Structure-Based Heterogeneous Distributed Shared Memory on PVM.* PhD thesis. Institute of Computer Science. National Tsing Hua University. June 1994.