

## **In the Beginning**

In the beginning there was message passing.

Then came the IVY people who formalized the concept of Distributed Shared Memory.

But their version of the Distributed Shared Memory was not very efficient, therefore most people still used message passing in real life.

So a while later, some people at University of Tennessee, Oak Ridge National Laboratory, and Emory University came up with a design called Parallel Virtual Machine (PVM).

PVM provides a set of primitives for invocation of processes, message passing (transmission and reception), broadcasting, synchronization (via barriers), mutual exclusion, and shared memory\*.

\*Well, sort of. As far as I can tell, the shared memory part is available for the earlier versions of PVM, but it was scratched on the later versions.

## **The Good**

PVM supports heterogeneity at the application, machine, and network level:

### **Application Level:**

PVM supports different languages. It could be a library for C or Fortran, a class library for Java, or a Perl Extension.

### **Machine Level:**

The best thing about PVM is its independence towards different platforms. It has been ported and tested on the following machines: Sun, Sparc, Microvax, DEC, IBM RS/6000, HP-9000, SGI, Next, Sequent Symmetry, Alliant FX, Intel iPSC/860, Thinking Machines CM-2, Convex, CRAY Y-MP, and Windoz95 and NT compatible machines.

Because of this heterogeneous nature, it is necessary for user programs to send and receive typed data in a machine independent form. To enable this, a set of conversion routines has been incorporated. These routines are transparent to the users; it is dealt with by the PVM system when messages are sent and received.

## **The Good (cont)**

### **Network Level:**

Different types of physical network can make up a parallel virtual machine. For example, Ethernet, FDDI, token ring, etc.

## **The Bad**

In order to exploit the power of message passing, the programmers need to know exactly what the program is doing. In other words, programmers are forced to manage the data flows explicitly (hence the term explicit message passing). They have to know where a piece of data is located and when to set up the send/receive connection between two communicating processes.

Obviously this is a tedious task and error-prone. On top of that, programs written for PVM are not robust. Programs are written for one specific task, and if the task changes slightly, a lot of modifications need to be made.

Analogy: C vs. Assembly Language.

# The Ugly

This is how the messages are written:

```
/* Sending Process */
/*-----*/
init send();                /* Initialize send buffer */
putstring("The square root of "); /* Store values in */
putint(2);                  /* machine independent */
putstring("is ");          /* form */
putfloat(1.414);
send("receiver",4,99);     /* Instance 4; type 99 */

/* Receiving Process */
/*-----*/
char msg1[32],msg2[4];
int num; float sqnum;
recv(99);                  /* Receive msg of type 99 */
getstring(msg1);          /* Extract values in */
getint(&num);             /* a machine specific */
getstring(msg2);         /* manner */
getfloat(&sqnum);
```

## **PVM-How It All Works**

Daemons called “pvmd” (or “pvm3d”, depending on the version) are started by user programs on each host. The daemons wait to receive messages from other processes and send out messages when asked to.

When an application process makes an initiate request, the local pvmd process first determines a candidate pool of target hosts based upon the information in a description file. Then one host is selected from this pool based on the following algorithm:

1. Select next host from pool in round-robin manner, based upon all initiations that originated here.
2. Obtain load metric from this potential target host.
3. If this quantity is less than a pre-specified threshold, select this host.
4. Otherwise, repeat the process. If no host has a load factor below the threshold, the host with the lowest load is the selected target.

Note that it is still up to the programmer to remember which host was chosen for each process.

## **Take the Good, scratch the Bad, and hide the Ugly**

Most of the DSM systems available are not quite so portable. They have their own methods of passing data between the processes. These methods tend to be system dependent and therefore restrict the DSM from connecting to another system with a different architecture.

Two good ways around the problem:

1. Build the DSM system on a data passing mechanism already in existence and already ported to all architectures.
2. Build the DSM system in Java.

Building the DSM system on top of PVM implies taking the portability of PVM (the Good), taking the advantage of a DSM (scratching the Bad), and hiding all the data passing from the programmer (hide the Ugly).

In the end, there should be a DSM system that people will die for... so what's the problem??

## Implementations

I studied 4 implementations of DSM systems based on PVM:

1. Adsmith (A DSM In Tsing Hua university), Taiwan.  
Object Based, supports both Release Consistency and Strict Consistency.
2. Phosphorus (a mineral that is said to play an important role in the performance of the human memory),  
France.  
Based on Munin (Object Based, uses Release Consistency)
3. DOSMOS (Distributed Objects Shared Memory System), France.  
Object Based, uses the concept of “Groups”, or scopes
4. Dream (Distributed REgion And shared Memory),  
France.  
Weak Consistency, uses the concept of “Regions”  
much like “Groups” or scopes...

The implementations of the four systems vary on the DSM level, but are almost identical on how they utilize PVM. Therefore we'll only look at Adsmith as an example.

## Adsmith

Adsmith is built on top of PVM at the library layer using C++, however, the PVM calls are transparent to the users.

Shared data in Adsmith have a “home”, or a owner. The selection of the ownership is done when the shared data is declared. By default, objects will be circularly allocated to daemons as their data owners (round-robin fashion). Programmers can also specify the owner of a specific shared data if they know the owner will be the processor that accesses that data the most.

Each shared data is annotated by either “MultiCopy with Cache”, “MultiCopy with NoCache”, “SingleCopy”, “MostRead”, and “MostWrite”. Like Munin, different shared objects have different methods associated for read and write.

During a write, the processor that has modified the shared data first updates the value of that object in its local memory. The local daemon sees the update and propegates the change to the data’s owner. Once the owner receives the change, it sends out invalidations to all other processes who have a copy of that object. Note that the processor which modified the data is not blocked when it waits for an acknowledgement from the owner of the object.

# Adsmith (cont)

Basic Structure:

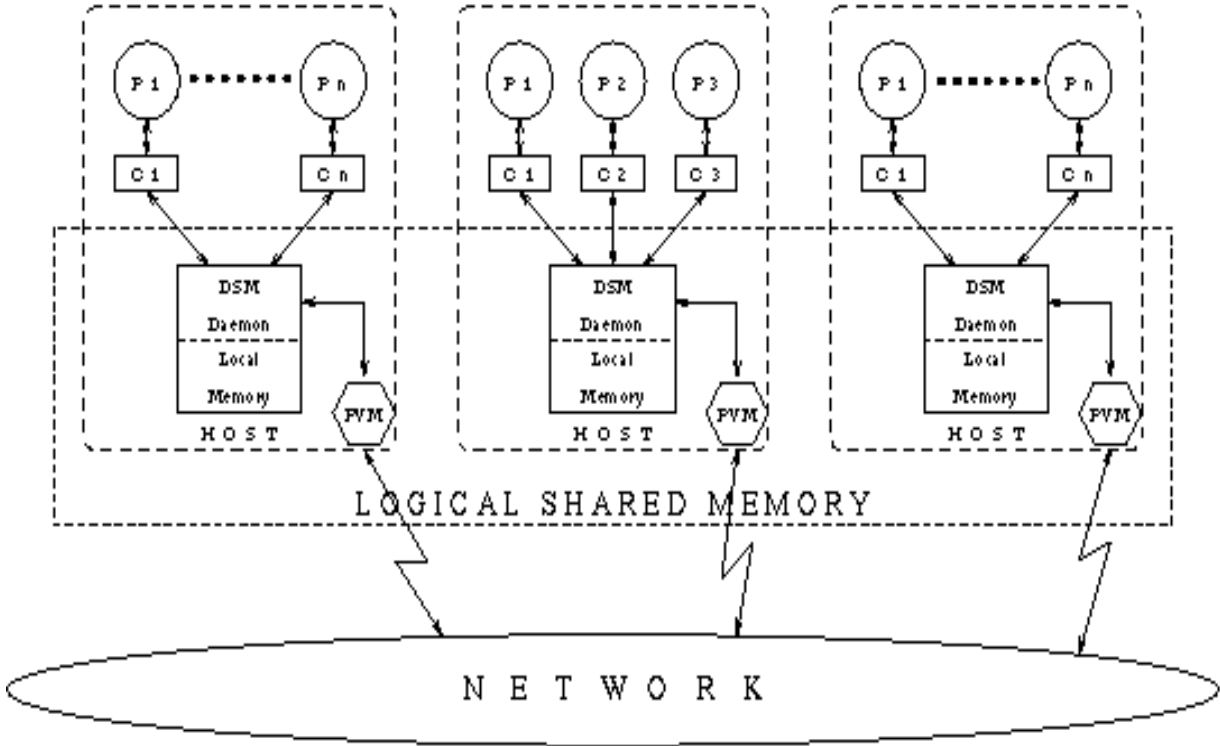


Figure 1: ADSMITH system architecture

## The More Interesting Stuff

### Access Style:

Adsmith claims that “ideally, users of a DSM need not distinguish between local and remote data accesses and issue calls to different objects differently. Unfortunately, an implementation to achieve this goal is very involved...” Therefore, Adsmith adopts a load/store access style for shared-object accesses. What it means is that a load (or called “refresh”) is required before accessing a shared object, and if the object is modified, a store (or called “flush”) is needed to tell the local daemon to update the object in the shared memory.

### Computation and Communication Overlapping:

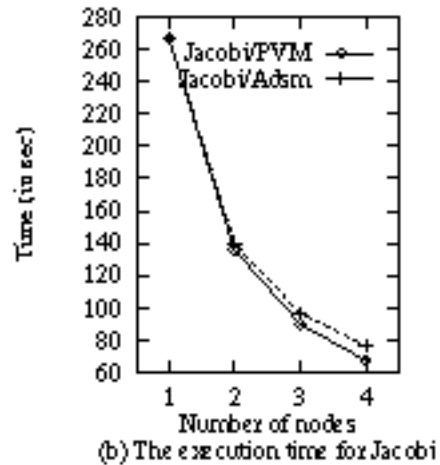
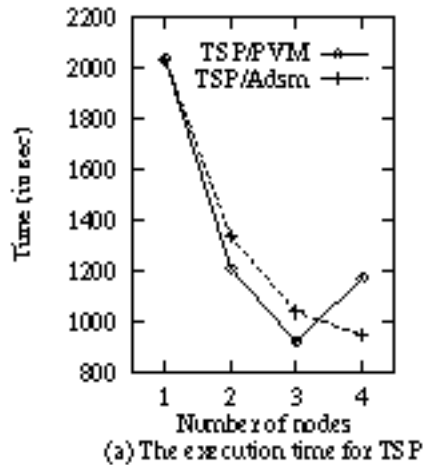
Because of the load/store style, prefetching and data aggregation becomes much easier. A load command could be issued long before the data is actually needed (prefetch). Before it is sent, the data is combined with other objects that the processor requests for and the whole thing is sent as a “bulk” message (overlapping)

### Active Accesses:

In active accesses, a short code segment, such as an expression, is transmitted to the host which requests for the data. This eliminates the need to transfer the data back and forth between the owner and the local node.

# Performance-Adsmith

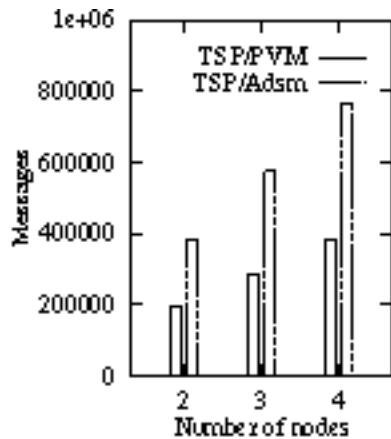
Adsmith:



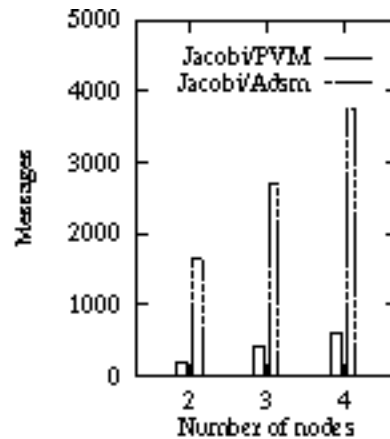
It is very hard to believe that Adsmith, based on PVM, would out perform PVM as indicated in the case of the TSP problem. They claim that PVM is out-performed because “the master process becomes a bottleneck during the reduction phase for calculating the maximum and minimum.”

## Performance-Adsmith (cont)

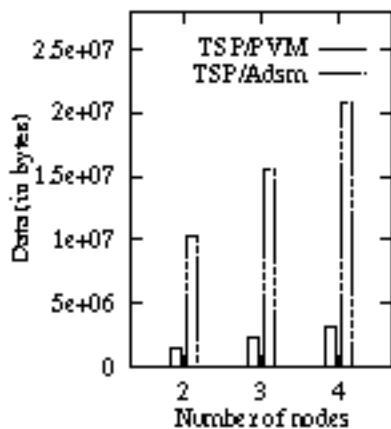
This is even harder to believe when combined with the following tables



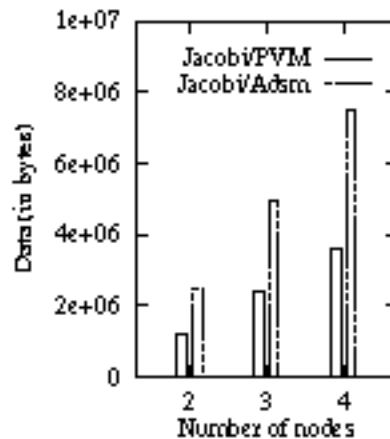
(c) The number of messages for TSP



(d) The number of messages for Jacobi



(e) The amount of data for TSP



(f) The amount of data for Jacobi

They claim that “the execution time does not increase in proportion to the number of messages and the amount of data transferred. This is a good evidence that Admish implementation is able to hide the communication latency by overlapping communication with computation”

# Performance-Others

Here we evaluate Phosphorus performance on running merge sort and matrix multiplication (100x100):

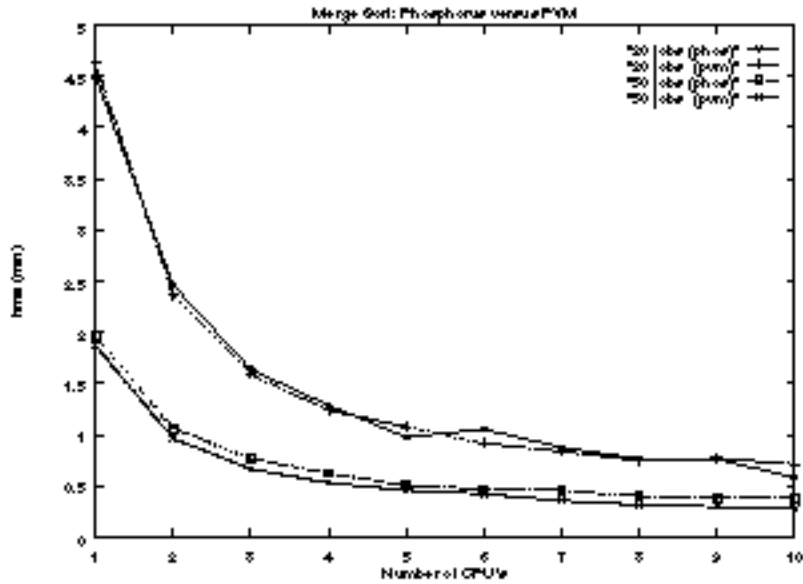


Figure 6.4: Merge sort: shared memory versus message passing

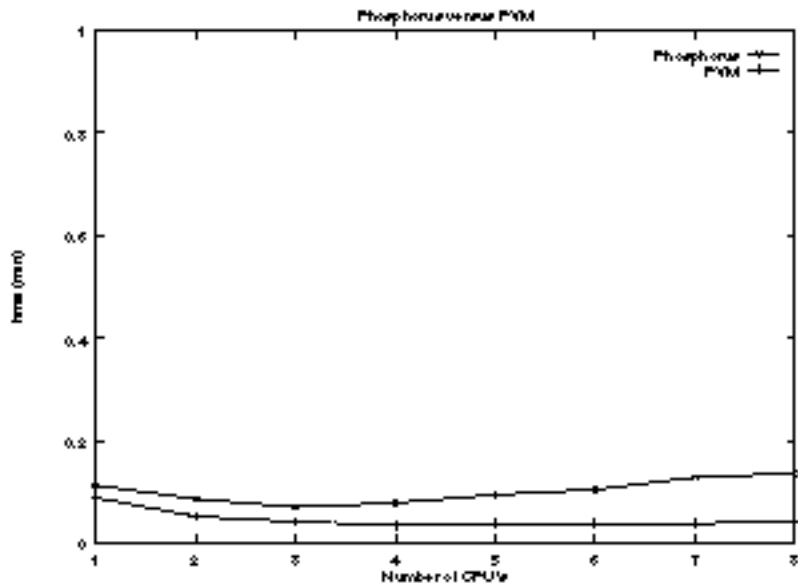
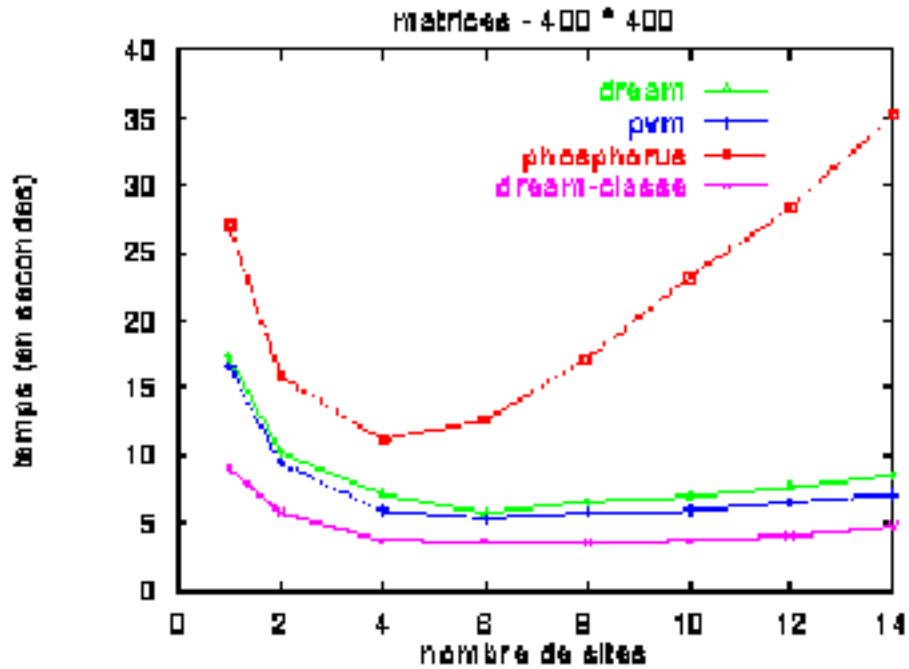


Figure 6.6: Performance of Phosphorus versus PVM

## Performance-Others (cont)

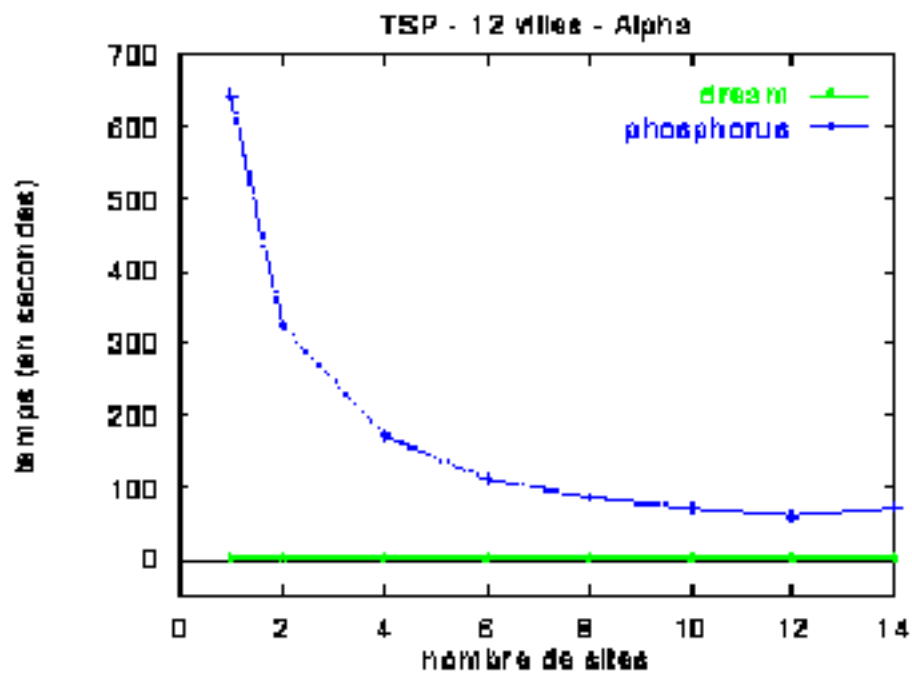
Here we evaluate the performance of Dream running matrix multiplication (400x400):



# The CrackDown

Hmmm.... notice the difference in the performance curves indicated by Phosphorus and Dream.

But is the Dream's data reliable??



This should help you answer your question... (this is a graph of the TSP problem running on 12 Alphas)

## **So What Does It All Mean??**

Well, even if we give Adsmith the benefit of the doubt and believe that their graphs are correct, one problem still exists: Adsmith is more like PVM than it is like a DSM from the programmers' perspective.

To write an efficient Adsmith program, you will need to annotate all the variables correctly, manually set up the shared variables so that they reside on processes that will eventually access them the most, load every single shared variable ahead of time so that the prefetch and bulk transfer could come into play, manually force the daemons to update the shared variables, etc, etc, etc, etc...

As of Phosphorus and Dream, after looking at their graphs, I'm pretty skeptical about their claims...

Q: So are DSM systems based on PVM better or worse compared to conventional DSMs?

A: My question exactly...

Moral of the Day:

If you want something really fast, go back to programming in Assembly Language. And if you want a DSM that's fast, portable, and requires no extra effort by the programmers, well.... Good Luck