tmPVM - Task Migratable PVM

C. P. Tan, W. F. Wong, and C. K. Yuen
{tanchung|wongwf|yuenck}@comp.nus.edu.sg

School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore 119260

Abstract

In this paper we introduce an implementation of PVM that exploit the homogeneity of a MPP network with user-level task migration. The target machine is the Fujitsu AP3000. It is basically a network of Sun workstations connected by a high-speed network. Furthermore, we explore the possibility of allowing the PVM host configuration to expand dynamically so as to include idle nodes into the virtual machine. The result is tmPVM, which allows the virtual machine to redistribute workload dynamically with low overhead.

1 Introduction

Parallel Virtual Machine (PVM) constructs a single large parallel platform using a collection of heterogeneous computers [1, 2] This allows large computational problems to be solved more cost effectively through the aggregated processing power and memory of many computers. The portability and flexibility of PVM allow the use of existing computer hardware to face the challenge of larger problem size.

The Fujitsu AP3000 is a Massively Parallel Processors (MPP) machine comprises of complete workstation nodes and a high-speed torus backplane [3, 4]. Each computing node is installed with a fully-functional Solaris operating system. The nodes can be used as standalone workstations or collectively as a cluster benefiting from existing distributed applications and the high-speed interconnection network.

The nodes are grouped into one or more partitions to support user applications in the AP3000 machine [4]. It is not uncommon for user applications to have parallel tasks which have different amount of computation work. Consequently, some nodes in the partition are usually idle while waiting for the slower processes (in other nodes). Therefore, we implemented a task-migration tool for PVM on the AP3000 machine to improve the utilization of the cluster as well as speed up certain applications.

1.1 Motivations

Each AP3000 node can be reserved for exclusive use or shared among multiple users [4]. The performance of a parallel application is affected by those tasks of it that run on overloaded nodes.

To alleviate the situation, we proposed and implemented a mechanism which can transfer load from one node to another to achieve dynamic load balancing.

The AP3000 machine provides a system-single image environment [4] through a shared file system. This allows a process to start and communicate without knowledge of its physical location. The shared file system presents a medium to preserve the process context for subsequent actions. The remote node can be instructed to restart the frozen process given the right environment. Since nodes in the AP3000 are homogeneous, binary compatibility is ensured.

2 Related Work

2.1 Process migration

Process migration is the act of transferring a process between two machines during its execution [5]. The transferred state includes the process address space, execution point (register contents), communication channels (open files) and other operating system dependent state. During migration, two instances of the migrating process exist: the source instance is the original process, and the destination instance is the new process created on the remote node. After the migration, the remote instance becomes a *migrated process*.

Migration can be classified according to the level at which it is implemented [5]. User level migrations are generally easier to implement but suffers from reduced performance although they are usually used for load distribution. Application level migrations have poor reusability as they requires duplicating most mechanism for each subsequent application, frequently involving effort beyond relinking the migration part with the application code. Lower-level migration is more complex to implement but has better performance, transparency and reusability.

2.2 PVM with task migration

SNOW MpPVM supports PVM process migration the *migration points* only [6, 7]. A pre-compiler modify the PVM source code to support the migration mechanism while the user may also select the migration points manually to suit their applications. The migration operations are performed at a high level by executing a set of programming language statements so that a process can migrate without any architectural restriction.

NSRC MPVM migrate a process by start a new PVM process with checkpoint data prepared by the old process itself [8]. Before migration, a process needs to freeze communication, handle pending messages, and pack checkpoint data. The new process will then unpack the data and resume the computation. It is implemented by a set of library over PVM. One problem of MPVM is the loss of messages during the migration. The MPVM users are also required to add the migration calls into the application codes.

DynamicPVM is an enhanced PVM with a PVM task migration facility using an extended version of Condor [9, 10]. Checkpoints are created and stored into a shared file system. For co-operating tasks, checkpoint requested while a task is in critical section will be postponed. The task migration protocol guarantees transparent suspension and resumption and an extended routing mechanism ensures that no message is lost. The PVM daemon (which the task is started) will reroute messages to the restarted PVM task at another node.

MIST MMPVM is a multi-user PVM system which is capable of transparent process migration and application checkpointing/restart [11, 12, 13]. Its migration protocol assumes that all binaries are available on a global file system. Each node has a single PVM daemon which maintain global knowledge about all PVM tasks not restricted to any particular user. Its protocol preserves message ordering by flushing all messages before the actual checkpoints are created.

Our migration mechanism includes the following features:

1. the process state is created directly from the `/proc` file system instead of relying on core dumps;
2. the process state is saved only at migration instead of checkpointing;
3. other than linking with the modified PVM library, the user can use all existing system libraries as system call wrappers are not required;
4. dynamically linked libraries are supported;
5. file I/Os are handled through runtime rewriting of the I/O routines to capture information for restart purposes;

The above distinguish our approach from that of MMPVM or DynamicPVM.

3 tmPVM

Our objective is to implement a PVM task migration tool to study the potential benefits of run-time load redistribution for a cluster of homogeneous nodes. Homogenety means that all the nodes in the PVM virtual machine (or at least those participating in the migration exercise) must be binary compatiable with one another. Our focus is the migration of CPU-bound PVM tasks which are most likely to benefit by being moved from heavily-loaded nodes to lightly-loaded nodes.

The implementation started with an user-level tool to migrate non-interacting Solaris processes across UltraSPARC nodes. For these processes, we only need to maintain their UNIX process context and associated I/O descriptors. An extended version of the tool was then implemented to capture both the UNIX process context and PVM task context in the PVM environment. Complications from inter-task communications result in some design decisions which imposes some limitations.

For a static group of PVM hosts, load balancing may be achieved by ensuring that load on each participating node are relatively similar. Consequently, the amount of waiting time between the faster workers and the slower workers will be reduced. We propose a scheme to exploit the task migration mechanism by allowing the set of PVM hosts to expand when some nodes are consistently heavily overloaded. Unlike the static virtual machine, the initial load balancing strategy here is to transfer the loads from the heavily overloaded nodes to the available slack nodes. After which, the previous strategy of spreading the load is exercised where appropriate. The migration process is automated by binding the load monitor modules with a centralized load balancer. Depending on whether the virtual machine is dynamic or static, different load balancing strategies may be adopted.

4 Implementation Issues

4.1 Process freeze and remote restart

A process is defined as an operating system entity that runs a program and provides an execution environment for it [14]. We implemented a mechanism to store the process state and use the information to restart the process at a remote node. The complete migration protocol for tmPVM tasks is illustrated in Figure 1.

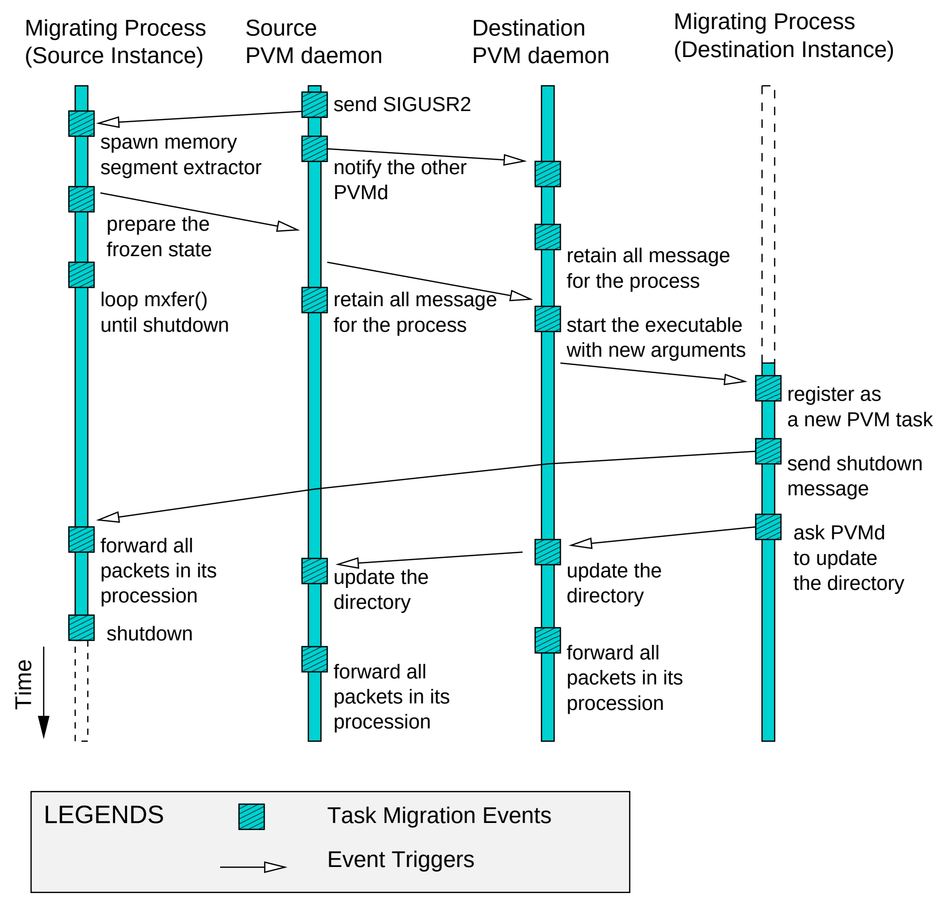
Migrating Process (Source Instance)
Source PVM daemon
Destination PVM daemon
Migrating Process (Destination Instance)
send SIGUSR2
spawn memory segment extractor
notify the other PVMd
prepare the frozen state
retain all message for the process
loop mxfer() until shutdown
retain all message for the process
start the executable with new arguments
register as a new PVM task
send shutdown message
ask PVMd to update the directory
forward all packets in its procession
update the directory
update the directory
shutdown
forward all packets in its procession
forward all packets in its procession
Time
LEGENDS
Task Migration Events
Event Triggers

Figure 1: Process migration protocol

Figure 2 illustrates the migration process. The process state capture process is fairly straight forward. First, the process installs a signal handler during its startup. When the process receives a (SIGUSR2) signal, the signal handler will spawn an external agent to extract memory segments of itself from the `/proc` file system. This mechanism differs from that of the Condor checkpoint system [15] which relies on core dumps.

The frozen process state includes all the memory segments that are attached during the point of migration. For the remote restart, the process is started using the same binary with additional arguments that indicate where to retrieve the frozen process state. The restarted process retrieves and loads all relevant memory segments. Once all the segments are installed, the process will continue the execution at the remote site from the point of interrupt using the `longjmp/setjmp` protocol.

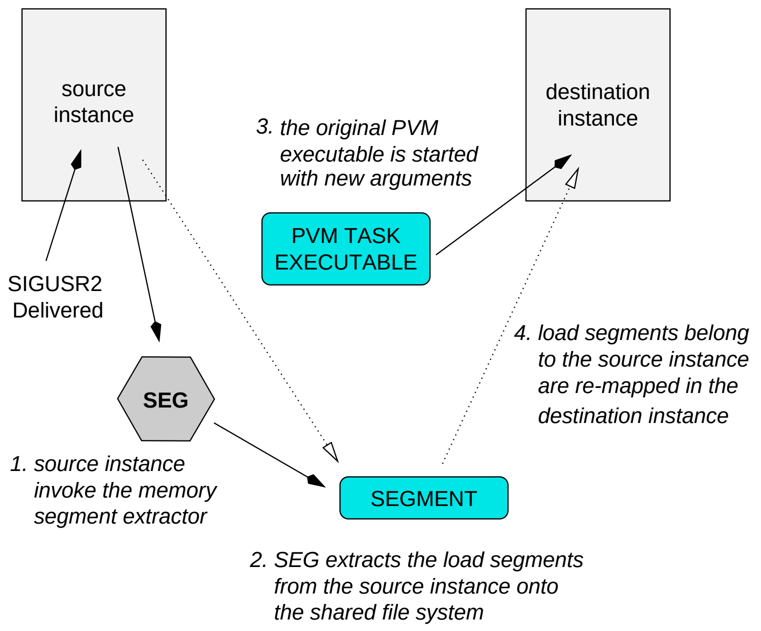
source instance
destination instance
3. the original PVM executable is started with new arguments
PVM TASK EXECUTABLE
SIGUSR2 Delivered
SEG
4. load segments belong to the source instance are re-mapped in the destination instance
1. source instance invoke the memory segment extractor
SEGMENT
2. SEG extracts the load segments from the source instance onto the shared file system

Figure 2: Unix Process Migration

Upon initialization, the `open` and `close` system routines are dynamically rewritten so that any calls to them are noted. This is necessary because one important piece of information, namely the file names, are not kept as part of the file descriptor data structure and subsequently after files are opened, are unavailable. During the process migration, the states of all opened file descriptors are captured. The file descriptors are restored by reopening the corresponding files with appropriate file mode relative to the state at the point of migration. Thus the restoration of file I/O is transparent to the user. This process assumes that the same file is available via the same path from any node within the system. This is possible because of NFS on the AP3000. This approach differs from that of MMPVM or Condor which requires the user to use their version of the system I/O libraries which have the necessary wrapper. Users can therefore make use of the latest version of the system libraries.

Each PVM task has two independent states - as an UNIX process (as described above) and in relation to the PVM application [13]. The UNIX process has a well-defined context which comprises of all the information needed to describe the process [14]. As part of a PVM application, the state of task includes its task ID and the message sent from/to that task. To implement the PVM task migration, all state information and messages should be captured, transferred and reconstructed.

Our approach is to make use of TID alias directory by supporting only the `indirect` routing for inter-task communication. This allows us to retain the encoding of the location information found in the original PVM TID format. The high-speed back-plane allows the directory update to be performed readily. Any message for the migrating task will be detained by the PVM daemon. The PVM daemon will then flush the detained message when the task has restarted. For routing of messages to migrated PVM tasks, the alias directory will be consulted so as to obtain the TID of the current instance.

4.2 Load Balancing

Load information is collected to implement a high level policy (see Figure 3) [5]. The decision to select a process for migration is based on the information gathered from the local and remote nodes.

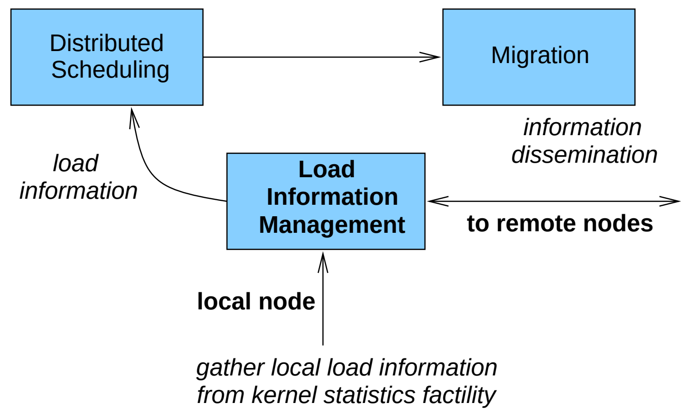
Distributed Scheduling
Migration
information dissemination
load information
Load Information Management
to remote nodes
local node
gather local load information from kernel statistics factility

Figure 3: Load information management module

A load monitor module is spawned on each node to collect local load information (see Figure 4). In tmPVM, a centralized resource manager gathers and maintains load information for each partipating node.

The system instrumentation gathers and maintains load information for each machine (or each process). The processor load is represented by one or multiple statistics. Statistics of interest include CPU utilization, amount of available physical memory, amount of paging activites as well as the number of active processes.

For process migration, the frequency of load information gathering should be inversely proportional to the migration cost [5]. Otherwise, the monitor of the system statistics and process progress may introduce too much overheads. Our load monitor module samples and sends load statistics to the resource manager at an interval of 2 seconds but otherwise it is asleep. Therefore, the additional overhead is minimal. As migration is usually applied to certain processes, it is also necessary to maintain a list of candidate processes within the node.

In the basic PVM system without migration capability, load balancing may be achieved with process creation based on the collected load information from the participating nodes. The original PVM creates processes in a round-robin fashion. Using the information provided by the load monitor modules, the schedule can be sensitive to the load on each node. By dispatching of new PVM tasks to the idle nodes in a greedy fashion, we are more likely to do better than the simple round-robin policy.

Once augmented with the process migration mechanism, we can do even better than the above. Through careful decision of which process to migrate, it is possible to achieve a speed up over the greedy load sensitive, task dispatch scheduling. We can either try to balance the workload at each node, or acquire idle nodes to relieve the heavily loaded nodes. In both cases, the overall execution time of the parallel application will experience a speedup as long as the cost for migration is relatively small compared to the life span of the process.

The option of process migration is exercised when there is a serious disparity among the load of the participating nodes. Once the decision is committed, the load monitors at the selected

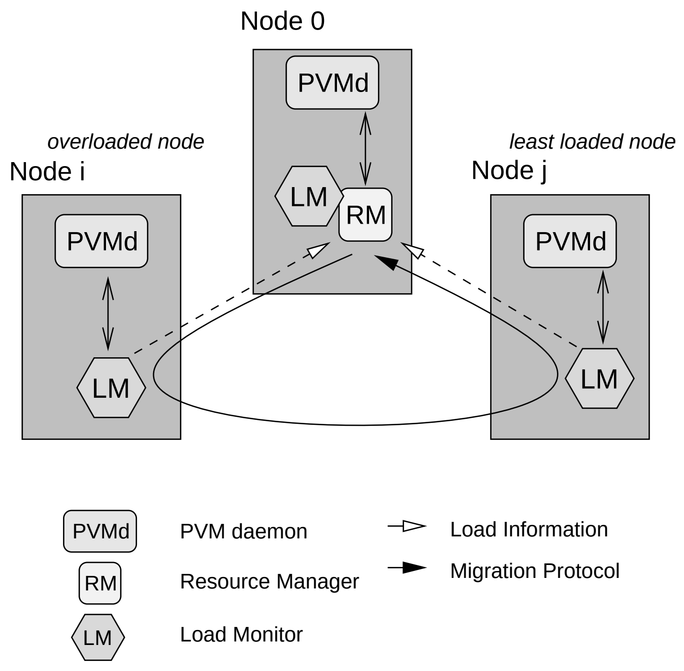

Figure 4: tmPVM virtual machine configuration

hosts are responsible to execute the process migration.

At the user-level, the available statistics are limited. Most statistics are gathered using the kstat facility of the Solaris operating system.

The following system statistics are captured:

- average number of process using the CPU in the last second;
- free physical memory available;
- probability of thrashing;
- number of migratable PVM tasks.

The cost of statistics gathering is relatively low as the load monitor module can retrieve the relevant data by refering to the appropriate symbols. Within the load monitor module, a thread is created to sample the data at fixed interval of one second. The statistics are averaged over a set of samples to reduce noises.

To crater for the growth of Unix process space over time, we decide to monitor the amount of available physical memory. When the available physical memory drops below some preset level, the load monitor module will warn the resource manager and avoid sending new PVM tasks to the node.

The probablity of thrashing is tied to the amount of page swapping activities. When there are many pages being swapped, the node may not have sufficient physical memory to support all its processes.

4.3 Modifications to Public Domain PVM

We modified the initialization and termination of PVM tasks. PVM tasks are linked with additional libraries and invoke an initialization routine during their startup. In this routine, the signal handler is installed, control structures are created and the local PVM daemon and load monitor module is informed of the existence of a new and migratable PVM task. The load monitor module will also be notified when the corresponding task terminates.

tmPVM can utilize the input from the resource manager to decide the target host when new tasks are spawned through `pvm_spawn()` calls. It can also use it for dynamic load balancing through the exercise of process migration.

Currently, tmPVM only supports `indirect` message routing. The TID alias directory is implemented as two separate tables on each PVM daemons. The `first TID` of a PVM task is considered as its unique identifier and TID of migrated instances are treated as aliases. Thus, application programs can assume that the task IDs involved are not modified during the execution time. For a migrated PVM task, the first table will map its current TID into its `first TID` while the second table will provide the TID of the latest instance.

4.4 Extensions to the Basic Migration Tool

We started with a migration tool for isolated Solaris processes on the UltraSPARC workstation node. To extend it to the PVM environment, we have to take care of messages on transit during the migration. Messages should not be lost since they are the synchronisation tools in the message-passing model.

The basic migration tool will shutdown the source instance once the memory segments are captured. If the PVM task is shutdown abruptly, message in transit may be lost due to broken sockets. Thus, the source instance does not terminate until the destination instance broadcast its presence in the tmPVM virtual machine. Between the point of migration to the termination, the source instance retains messages and forwards them to the destination instance.

For the basic migration tool, it is possible to have multiple destination instances of the process. Each instance restart the execution from the point of migration. In the tmPVM environment, however, this is not allowed as the restarted instance may influence other PVM tasks through inter-task communication.

By forwarding and disallowing multiple restart instances, tmPVM ensures that there is no message loss during task migration. However, as there may be messages forwarded by the source instance after the activation of the destination instance, message order during the migration cannot be guaranteed.

4.5 Abstract Implementation

Our mechanism does not require the replacement of any system-level libraries. This provides more flexibility for the users. Implemented at the user-level, the mechanism is abstracted from the underlying system-level libraries. On the other hand, checkpointing systems (such as Condor [15]) require the user to recompile the modified system libraries and relink the PVM application whenever the system libraries have changed. Furthermore, in systems supporting dynamic linking, system routines are allowed to supersede one another. For example, the Solaris thread library redefines all the basic I/O routines of the libc library.

In our approach, the association between the application and the system-level libraries is captured at the point of migration. As long as the system libraries are not modified during the migration, the association remains valid. The mechanism works for both memory segments mapped from the static and dynamic system libraries.

We feel that this level of abstraction for the migration tool is important for the user. If the migration tool is tightly coupled with the lower level system libraries, the usage of the user-level migration tool may require further support from the system administrators for example in managing the dynamic link paths.

4.6 Limitations

Our decision to implement the mechanism at the user level introduces several limitations. These limitations are also present in other process migration and checkpointing systems.

The process freeze facility assumes relevant directories are mounted identically across all the participating nodes. For instance, the scratch pad in the `$PVM_ROOT/TM` directory must be accessible on all of the nodes. Some debuggers may not work with the migrated processes since there are additional low level codes which bring in the frozen process state of the source instance. The current mechanism does not honours timer routines. Also, multithreading is not supported.

The scheduler which is implemented in the resource manager supports only one task migration at any one time. It also assumes that the user has sufficient disk space to store the segment files on the scratch pad for the subsequent subsequent process to restart. All load monitor modules are assumed to be active till the shutdown of the tmPVM virtual machine.

tmPVM is currently available for clusters of UltraSPARC workstation on the Fujitsu AP3000 machine. User are required to include a call to `TCP_TM_init()` before any statements. The routine will setup the necessary mechanism, data structures and notifcaition to the load monitor modules. The current implementation does not support `direct` message routing in the public domain PVM.

5 Performance Evaluation

The evaluation of tmPVM was conducted on the Fujitsu AP3000 at the National University of Singapore. The machine has a total of 32 143-MHz UltraSPARC nodes which has a total of 256 MB of physical memory each. The nodes are connected by a 200MB/s backplane known as the AP-Net. The backplane is a two-dimensional torus-network which guarantees reliable and in-order message transfers. The Public Domain PVM (or PD PVM) used refers to the version 3.3.11 distribution.

5.1 Overhead of Basic Communication

The communication overhead is measured by using the ping-pong program against some number of migrated tasks in the tmPVM virtual machine. For this purpose, we modify the nntime program in the Public Domain PVM distribution to improve the timing acccuracy. To conform with the limitations of tmPVM, the PvmDirectRoute option was also disabled.

Table 1 suggests that our implementation impose a penalty of about 7.7% for inter-task communication over the original version even when there is no task migration. We attribute this overhead to the additional codes in the modifed PVM daemons.

The TID directory is active when at least one PVM task has migrated and remain active. The directory is consulted for the routing of all messages. A maximum cost of 12.2% and 18.0%

message size (in bytes)	PD PVM (Indirect)	tmPVM with # Migrated Process					
		0		5		10	
0	2515	2524	*(0.4%)*	2666	*(6.0%)*	2776	*(10.4%)*
80	2736	2904	*(6.1%)*	3028	*(10.7%)*	3228	*(18.0%)*
800	2904	3129	*(7.7%)*	3258	*(12.2%)*	3381	*(16.4%)*
8000	5941	6048	*(1.8%)*	6137	*(3.3%)*	6273	*(5.6%)*
80000	48141	48348	*(0.4%)*	49612	*(3.1%)*	49814	*(3.5%)*
800000	466856	469541	*(0.6%)*	484688	*(3.8%)*	486051	*(4.1%)*

Table 1: Overhead of Basic Communication (in microsec)

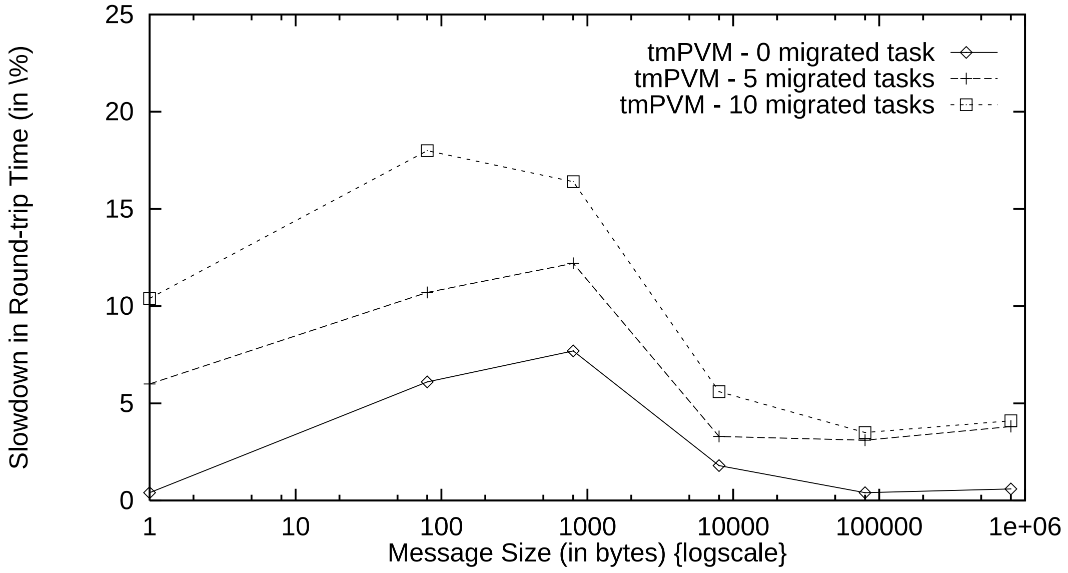

Figure 5: Overhead in Basic Communication (Log Scale)

for 5 and 10 migrated task entries respectively. As the number of active migrated PVM tasks in the tmPVM virtual machine increases, so too will the communication overhead. Thus, we should only exercise the process migration option when the benefit from the improved communication and reduced CPU contention outweight this penalty.

5.2 Migration Cost

The migration cost is essentially the sum of the obtrusiveness cost and the cost of restarting the process[12]. Originally, the obtrusiveness measures how fast a workstation owner can regain exclusive access to the local machine [16, 12]. In our context, it represents how fast the frozen process state is available on the shared file system. We obtain a lower bound of the migration cost by monitoring the process migration within a single node.

While ideally the time difference should be determined by executing the process migration across two different nodes, this measurement is complicated by the problem of clock synchronisation and the associated message propogation delays. The result denotes the lower bound as the network file transfer and additional messages are ignored. Other overheads have to be considered for practical situations.

A PVM task which obtain memory reserved for a two-dimensional array of `double` data item of a specific size is 'migrated' to the same node. The aim of the experiment is to obtain the cost of freezing and unfreezing a process of particular sizes within the same machine.

Matrix Size (# double)	Frozen Process Size (in KB)	Obtrusiveness Cost (in sec)	Restart Time (in sec)	Minimum Migration Cost (in sec)
		(a)	(b)	(c) = (a) + (b)
20 x 20	1680	0.509	0.131	0.640
40 x 40	1712	0.546	0.131	0.677
80 x 80	1776	0.582	0.123	0.705
100 x 100	1840	0.627	0.132	0.759
200 x 200	2304	1.042	0.134	1.176
300 x 300	3088	1.662	0.135	1.797
400 x 400	4176	2.747	0.147	2.894
500 x 500	5584	4.024	0.137	4.161
600 x 600	7312	5.397	0.135	5.532
700 x 700	9344	6.934	0.139	7.073
800 x 800	11680	8.886	0.145	9.031
900 x 900	14336	11.121	0.168	11.289
1000 x 1000	17312	13.638	0.191	13.829

Table 2: Migration cost vs Process state size

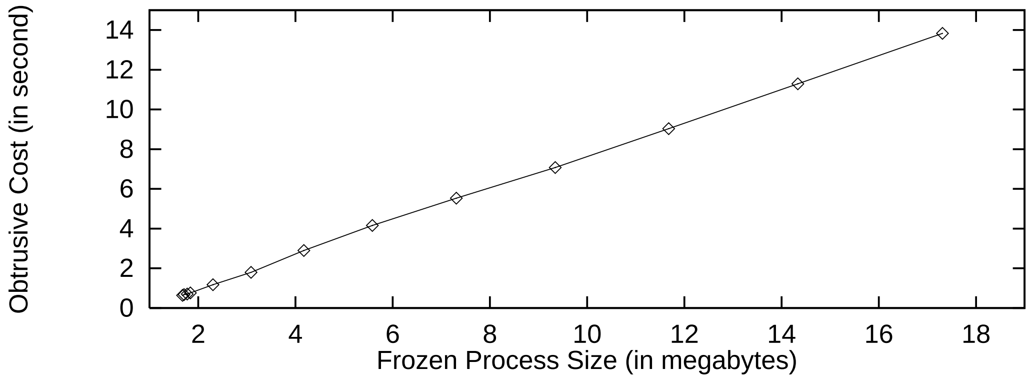

Figure 6: Migration cost vs Process state size

Table 2 summarizes the average of the best 10 observations for each test configuration. The result is an important factor for the decision whether a suggested process migration will improve the overall execution time of the application. Profiles of the active tmPVM task may be considered together with the migration cost during decision making.

The data suggests that process may be migrated in less than 1 second. The graph also suggest that the cost is approximately linear with respect to the size of frozen process state. The restart of the process involved the mapping of the memory addresses to the segments in the frozen process state. The cost of this mapping operation is relatively stable at less than 0.2 second.

5.3 CPU-Bound Tasks

We constructed an "embarassingly parallel" application to evaluate the effectiveness of migration using 5 U-143 nodes. With the information supplied by the load monitor modules, the resource manager can use the load information to dispatch the worker tasks to the nodes. Once the execution has started, the tmPVM virtual machine can apply process migration to redistribute the workload. For this expertiment, we did not allow the virtual machine to expand when there are overloaded nodes.

The test application consists of a group of `worker` synchronized by the `master` PVM task. Each worker has given a work that runs for approximately 600 seconds. The execution time of the application is dominated by that of the slowest `worker`. Table 3 summarizes the results obtained from 20 samples for each configuration. The best observed timings are highlighted. The theoretical optimal execution time is computed assuming that task migration has no overhead.

The results compare the execution time with and without the resource manager. With the resource manager, the tmPVM virtual machine benefits from load-sensitive task dispatch and process migration capabilty. For the case without the resource manager, the new PVM tasks are dispatched to the test nodes by the default round-robin policy [1].

No. of `worker`	Execution Time with tmPVM (in sec)						
	without RM/migration			with RM/migration			
	average	min	max	average	min	% speedup	max
		(a)			(b)	[(a)-(b)] ÷ (a)	
5	618.5	616.4	624.3	654.5	631.4	*(-2.4%)*	694.0
6	1225.2	1219.0	1233.6	997.7	994.0	*(18.5%)*	1000.3
7	1225.2	1221.3	1228.1	999.5	996.4	*(18.4%)*	1004.9
8	1226.3	1221.5	1232.8	1213.4	1161.3	*(4.9%)*	1304.1
9	1226.7	1223.1	1239.2	1250.5	1246.5	*(-1.9%)*	1258.1
10	1226.0	1223.3	1229.9	1265.3	1253.2	*(-2.4%)*	1279.6
11	1829.9	1825.4	1836.5	1522.2	1515.1	*(17.0%)*	1528.6
12	1838.0	1827.0	1850.0	1591.8	1571.0	*(14.0%)*	1617.2
13	1833.2	1828.8	1837.2	1763.3	1733.9	*(5.2%)*	1809.8
14	1837.1	1829.5	1850.5	1805.6	1770.5	*(3.2%)*	1875.3
15	1835.7	1828.7	1849.5	1884.5	1876.2	*(-2.6%)*	1897.8
16	2435.2	2430.7	2440.0	2070.6	2055.0	*(15.5%)*	2096.1
17	2437.7	2430.8	2441.8	2209.4	2188.9	*(10.0%)*	2245.8
18	2468.9	2435.7	2575.5	2337.0	2310.5	*(5.1%)*	2402.0
19	2445.2	2435.0	2458.4	2511.0	2413.2	*(0.9%)*	2650.9
20	2450.4	2437.6	2490.8	2516.6	2496.5	*(-2.4%)*	2568.5
21	3044.0	3039.1	3058.4	2701.1	2683.1	*(11.7%)*	2711.7

Table 3: Performance of CPU-bound PVM tasks in tmPVM

From the results, the completion time for the tmPVM applications augmented by resource manager are generally shorter. But, the collection and dissemination of load statistics by the load monitor modules introduces overhead of not exceeding 2.5% of the total execution time. When tmPVM's task migration mechanism is employed appropriately, the application performance may be improved upto 18.4%.

The results also suggest that the variance in execution time of application with resource manager is larger. Other than the noise introduced by the load monitor modules, the variance

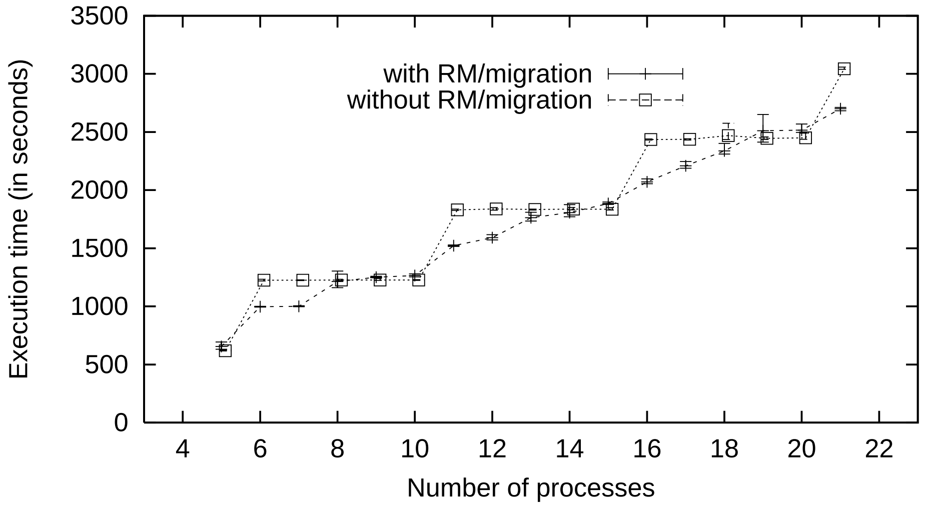

Figure 7: Performance of CPU-bound PVM tasks in `tmPVM`

can be attributed to the quality of the decision to do process migration. If the opportunity of load redistribution is identified promptly, the application may finish the execution earlier. In fact, the decision for this particular application is relatively simple since all `worker` are given the same amount of work. Thus, the slower `worker` tasks are among those which are dispatched to the overloaded nodes and redistribution of workload is only possible after the faster `task` has completed.

5.4 Dynamic Reconfigurable PVM

This set of experiments is essentially the same as the previous one. However, in this case, we allow the virtual machine to *dynamically* expand when there are some consistently overloaded nodes. Probes are sent on a random interval to search for idle nodes in the cluster. When the resource manager decides that load redistribution is unlikely to improve the execution time, it can co-opt new hosts into the virtual machine. These chosen new hosts are those with low loads and will be targets of subsequent process migration.

We compared the execution time of heavily overloaded nodes. In the experiment, twenty workers were spawned on an initial configuration of two PVM hosts. Each worker was given a artificial workload of approximately 200 seconds. The comparison is based on the execution time of static and dynamic `tmPVM` virtual machine. For the static virtual machine, we used 20 samples of the execution time to obtain the results shown in Table 4. The samples for the dynamic virtual machine is plotted in Figure 8.

	Execution Time (in seconds)		
Configuration	Average	Minimum	Maximum
static	2057.0 ± 51.2	2032.2	2261.5
dynamic	1376.2 ± 261.5	867.1	1991.6

Table 4: Execution Time On Heavily Overloaded Nodes

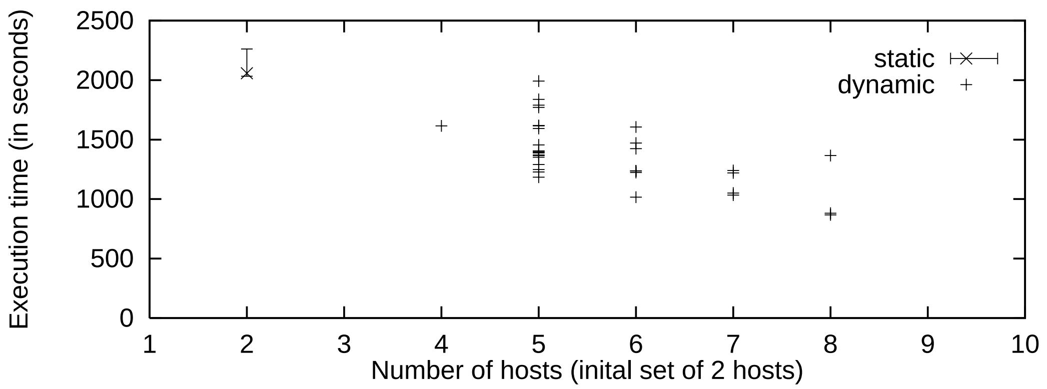

Figure 8: Execution Time On Heavily Overloaded Nodes

The results suggest that by coupling the migration mechanism with the flexibility of allowing the virtual machine to expand, the user application can adjust its execution environment. This is useful for users who are unable to maintain exclusive controls of computing resources.

6 Conclusion

We have demonstrated the feasibility of providing process migration at user-level for PVM applications. Starting from a mechanism that migrate Solaris process during its run-time, we extend the tool to work within the PVM. This remove one of the main advantages of PVM which is heterogeneity. However, it is not clear if process migration on a heterogeneous network is generally achievable and if the benefits derived are worth the added complexity.

Using the information gathered by load monitor modules, tmPVM can migrate processes away from heavily loaded nodes. It is also possible to allow the virtual machine to expand its configuration temporarily so that load can be redistributed more effectively. Although the work described was done on the Fujitsu AP3000 MPP, we see no problem in porting it to a network of Sun-Solaris workstations. Indeed we believe it is also quite feasible to port the migration mechanism to other flavors of Unix.

References

[1] A. Geist, A. Beguelin, J. Dongarra, W.Jiang, R.Mancheck and V.Sunderam, *PVM : Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994,

[2] V. Sunderam, "PVM: a framework for parallel distributed computing," *Concurrency, Practice and Experience*, vol 2 no 4, pp 315-339, December 1990, http://www.netlib.org/ncwn/pvmsystem.ps

[3] H. Ishihata, M. Takahashi, and H. Sato, "Hardware of AP3000 Scalar Parallel Servers," *Fujitsu Scientific & Technical Journal*, vol 33 pp 24–30, June 1997.

[4] H. Oyake, Y. Iguchi, and T. Yamane, "Operating System of AP3000 Series Scalar-Type Parallel Servers," *Fujitsu Scientific & Technical Journal*, vol 33 pp 31–38, June 1997.

[5] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S.N. Zhou, *Process Migration Survey*, The Open Group Research Institute, vol 5, March 1997.

[6] K. Chanchio, and X.S. He, "Efficient Process Migration for Parallel Processing on Non-Dedicated Networks of Workstations," *Institute for Computer Applications in Science and Engineering Technical Report TR-96-74*, December 1996.

[7] K. Chanchio, and X.S. He, "MpPVM : A Software System for Non-Dedicated Heterogeneous Computing," *Proceedings of the International Conference on Parallel Processing*, August 1996.

[8] J.J. Song, H.K. Choo, and K.M. Lee, "Application-level load migration and its implementation on top of PVM," *Concurrency, Practice and Experience*, vol 9 no 1, pp 1-19, January 1997.

[9] L. Dikken, *DynamicPVM : Task Migration in PVM*, November 1993.

[10] L. Dikken, F. Linden, J. Vesseur, and P. Sloot, "DynamicPVM : Dynamic load balancing on parallel systems," *Lecture Notes in Computer Science*, vol 797, pp 273-277, 1994.

[11] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "Adaptive load migration systems for PVM," *Proceedings of Supercomputing '94*, pp 390-399, November 1994.

[12] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MPVM : A Migration Transparent Version of PVM," *Computing Systems*, vol 8 no 2, pp 171-216, Spring 1995.

[13] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole, *MIST : PVM with Transparent Migration and Checkpointing*, ACM, 1993.

[14] U. Vahalia, *UNIX Internals - The New Frontiers*, Prentice Hall, 1996.

[15] A. Bricker, M. Litzkow, and M. Livny, *Condor Technical Summary*, Computer Sciences Department, University of Wisconsin-Madison, October 1991.

[16] M. Litzkow, and M. Livny, "Experience With The Condor Distributed Batch System," *Second IEEE Workshop on Experimental Distributed Systems*, pp 97-101, October 1990.