XTPVM : An Extended Threaded Parallel Virtual Machine

Tarek Abdel-Radi and Muhammed Mudawwar
{ tradi, mudawwar }@aucegypt.edu
The American University in Cairo
113 Kasr Al Aini, Cairo, Egypt

ABSTRACT

Providing the parallel programming community with a single abstract view to a heterogeneous network of workstations is a very intrinsic task. Parallel programmers want a facility to combine a set of workstations and use this pool to create and execute threads. Alleviating the programmers from time consuming mechanisms such as managing communication among the threads over a network, scheduling threads and balancing the load through thread migration are all crucial. We developed XTPVM (eXtended TPVM) to be a transparent thread scheduling and migrating virtual machine based on TPVM (Threaded Parallel Virtual Machine) that provides parallel programmers with the ability of using a network of workstations for parallel processing. It considers heterogeneity and differences in processing power between workstations, and also the dynamics of the system as a whole. In such a computing environment, where the use of resources vary as other applications consume and release resources, transparent scheduling of parallel threads onto the least loaded hosts was achieved. As workstation loads vary due to their use by other users, XTPVM adapts by migrating ready threads before being committed from loaded workstations to less loaded ones using DGP (Distributed Global Plan). Experiments have been performed to demonstrate the effect of some parameters of XTPVM on total execution speed and to show the usefulness of migration.

Keywords: *XTPVM, TPVM, PVM, Virtual Machine, Thread Scheduling, Thread Migration, Distributed Dynamic Load Balancing.*

1. Introduction

Considerable advances have been made in recent years in both parallel and distributed computing. However, despite common interests, the work in the two areas has remained quite distinct. The main concern of the parallel community is with speed and processor scalability, resulting in specialized architectures and minimal reliance on system software, while the distributed community is concerned with wide area connectivity and resource sharing, resulting in open system platforms and largely functional system software [1]. The recent trend is using workstation clusters for parallel computation indicating that these distinctions are potentially disappearing. The emergence of high-powered workstations connected via fast communication networks has increasingly been considered as an alternative to dedicated high performance parallel computers. These workstation networks are not only cheaper, but also provide a general-purpose computing environment that is typically shared by both parallel and non-parallel application developers and users. Distributed Parallel Programming has been of major concern over the past few years, and is based on heterogeneous networked platforms, frequently comprising powerful workstations and software systems that together emulate the general purpose virtual parallel computers [2]. Networks of workstations can be used as large scale parallel machines, although at present their use is restricted to coarse grained computation.

It is thus required to allow parallel program writers to make use of the abundantly available idle workstation power on a LAN transparently by giving the ability to define and run threads transparently and remotely. This can be achieved by using a virtual machine as a layer of abstraction that allows the application programmer to view different heterogeneous computers in a single perspective. A virtual machine takes advantage of the underlying operating system and network resources and presents a usable programming interface to applications. A thread-based parallel virtual machine should be able to schedule and migrate threads transparently, and adapt to changing workstation loads. Just as normal threads can share memory, threads in the parallel virtual machine should also be able to share memory.

The goal of XTPVM is to develop a thread-based parallel virtual machine that provide a library of functions to develop parallel programs, that will schedule threads transparently on a network of workstations, and that will maintain a load balance through thread migration. Thus the application programmer will have no control over which workstations will be used for the execution of a particular thread. However, he can specify the pool of workstations over which threads will run. The programmer will only see an interface to a library of functions that can be used to execute relatively independent threads that can possibly share remote memory in a distributed way.

XTPVM will balance the load on a network using the user specified load band that is tolerable to make use of the idle workstations and remove the burden off heavily loaded workstations. Only ready but non-committed threads can migrate. A thread is committed to a workstation once a context has been created for it. Non-committed threads do not have a context. Thus, their migration is simplified and the cost of migration is greatly reduced because there is no need to transfer their contexts across a network. The share of each workstation in the number of non-committed threads thus grows and shrinks, as a program runs. A good load metric for migration is the number of non-committed threads queued at each workstation which must be balanced within a user selected load band.

This paper is organized into 7 sections. Section 2 presents a brief background of PVM and TPVM. Section 3 discusses the abstract machine developed, namely XTPVM. Section 4 investigates the scheduling and migration alternatives, and those selected for the implementation of XTPVM. Section 5 presents a performance evaluation of XTPVM. Section 6 presents related work in this field, and section 7 rounds off with the conclusion and discusses some future enhancements.

2. Background: PVM and TPVM

PVM enables a collection of different computer systems to be viewed as a single parallel virtual machine for hosting processes [3]. It provides an infrastructure where networks of workstations can be viewed by application developers as a large distributed-memory multiprocessor machine, making it convenient to create parallel applications by virtualizing the workstation network and by providing the necessary primitives for process communication (via message passing) and for process control [4]. The PVM task is a process that isn't thread safe. TPVM, on the other hand, is a thread-based virtual machine, implemented as a layer over the native release version of PVM[5]. TPVM's threads retain the original computing model in which an application is comprised of "components" or sub-algorithms, each of which may be manifested as a collection of instances that safely cooperate via message passing [6]. TPVM threads form the units of parallelism and are hosted within regular PVM processes or "pods". In other words, a collection of TPVM threads that comprise an application are created and executed within the context of a smaller number of pods which provide an environment shell and do not contribute to the computation as is typical with other thread systems such as SunOS LWP. Host processes or pods are initiated via normal native PVM mechanisms e.g. pvm_spawn. These pods must export thread entry points that specify the types and numbers of threads that the process is willing and able to host.

Figure 1 shows the TPVM system. It consists of three components : a library, a portable thread interface, and a thread server module, which performs scheduling and system data management [6,7].

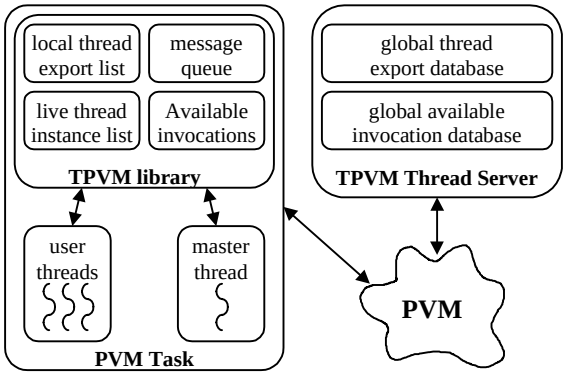

Figure 1: The TPVM system

The user interacts with the TPVM system via library calls that provide a number of required support services. The TPVM library is used for thread-based distributed computing. The library routines are based on PVM primitives for services such as message passing, on the portable thread interface module for services such as thread creation and scheduling, and on a TPVM thread server task for scheduling and export database services. Thus, the three modules interact together to provide the user with the TPVM system.

The Portable Thread Interface module handles all thread-related services such as thread management, communication and synchronization. It abstracts basic thread services required to implement the TPVM library routines. This abstraction allows the implementation of the TPVM library to be decoupled from the available thread services on various OS and machine platforms. It provides for thread creation, exiting, yielding, obtaining and releasing mutual exclusion, and determining a unique identifier number associated with the running thread. There are fairly basic operations, which can be supported by most typical thread systems with little implementation effort.

TPVM relies on a centralized thread server task which provides support for the thread export database, scheduling, and data-driven thread creations that is neither scalable nor failure resilient.

3. XTPVM

3.1 The Need for XTPVM

The goals of having a simple, complete and easy to use library for a virtual machine that is easy to set up has been met by both PVM and TPVM, but in different degrees. PVM has existed for quite some time, and its interface has gone over several enhancements and modifications, and thus has a more complete set of library functions. TPVM on the other hand is more recent, and is not as clear as PVM, and thus lacks in some points. For example, it does not support non-blocking receive between threads while PVM supports it between processes, making TVPM not complete enough. Also, lack of documentation on concepts such as remote memory, makes programming with TPVM difficult at first until some rules are discovered by experience. An example is the inability to declare remote memory anywhere in the program except by the main thread of only one pod which should then call *tpvm_go*. To program with TPVM, the user should be familiar with both the PVM and TPVM libraries and call appropriate functions when necessary. XTPVM can supply the user with only one set of library calls that provides consistency and simplifies programming [8].

Despite the success of PVM, there are areas such as resource allocation where PVM lacks support. In a computing environment where the availability of resources changes over time, the allocation and reallocation of resources in response to these changes is essential to utilize the resources effectively. Also, PVM has no provisions for thread safe processes, which are its basic unit of scheduling rather than threads. Parallel programmers are more familiar with threads and thus cannot port their parallel algorithms easily to PVM. TPVM, on the other hand, handles thread definition, creation, termination and invocation and has also introduced the idea of dataflow computation for threads over PVM. PVM and TPVM are neither abstract nor transparent enough, requiring the parallel programmer to concentrate on many non-application specific tasks. For example, to schedule tasks, an application must use one of PVM's three modes of scheduling processes when spawning. These are *PvmTaskDefault* (PVM can chose any machine to start task), *PvmTaskHost* (PVM should chose a particular host), and *PvmTaskArch* (PVM should chose any workstation of a particular architecture). TPVM, on the other hand, has 2 modes for scheduling its threads at spawn time: *PvmThreadLocal* (to start the thread in the spawning pod) and *PvmThreadWild* (to start the thread on any of the pods running in a workstation in the current virtual machine.

One of the limitations of TPVM is the use of round robin scheduling, rather than load sensitive scheduling, to schedule threads. Neither PVM nor TPVM supports process/thread migration. TPVM requires strong encapsulation of threads, and so shared variables are not permitted. XTPVM alleviates some of the problems of PVM and TPVM for the parallel programming community. The next section discusses its features.

3.2 Features of XTPVM

The features of the proposed virtual machine are:

- *A simple, complete and easy to use library that runs under the application layer*. The user interface is a small set of library functions whose declaration is provided in a header file to be included for compilation with the user's program. A library called *libxtpvm.a* should be linked with the user's code to generate the executable.
- *Ease of virtual machine setup over a heterogeneous network.* A simple text file called *hosts.ini* should be edited by the user to specify a list of host names to use as the pool of workstations in the virtual machine setup as well as the location of the user's application on each workstation.
- *A simple method of thread management*. XTPVM is thread-based. The XTPVM library provides a complete set of operations for thread definition, creation, termination and invocation.
- *Transparent thread scheduling.* XTPVM thread scheduling gives priority to least loaded workstations to receive the newly created threads, rather than the round robin scheduling of threads provided by TPVM.
- *Transparent thread migration.* A major contribution of XTPVM not available in PVM and TPVM is load balancing through thread migration, which is in fact an implementation of the decentralized global plan algorithm discussed later. Providing location transparency allows the programmer to deal with threads without having to know where on the network they are located. The user can easily and transparently create and terminate these threads without any knowledge of their location. This location transparency is achieved by assigning a unique name for each thread function. All thread creation, termination, invocation, sending and receiving is done using this unique name and not the TPVM integer thread id, since a thread might migrate dynamically, and any id returned to the user might be invalid in later library calls.
- *Allowing threads to cooperate*. XTPVM provides send and receive primitives between threads as well as remote shared memory for inter-thread communication. To use remote shared memory, the programmer would first specify the shared variables to be registered by XTPVM at initialization time, and then threads can get and put values to them easily, using *xtpvm_get* and *xtpvm_put*.

3.3 Overall Architecture

XTPVM is built over PVM and TPVM as shown in Figure 2, thus making use of previous research efforts and experimental experience to develop a more powerful tool for the parallel programming community. In summary, XTPVM has a somewhat abstract structure, and the user only needs to know a small set of XTPVM library calls. No prior knowledge of PVM or TPVM is required.

XTPVM has access to both PVM and TPVM subsystems. The user application will have access to XTPVM, TPVM and PVM even though it is recommended that the XTPVM interface should be used solely and not bypassed via calls to TPVM or PVM routines. The XTPVM interface is complete enough for its specified purpose.

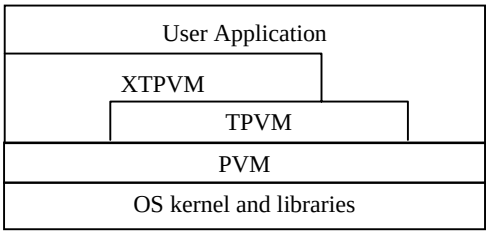

Figure2 : Layered Diagram relating XTPVM to PVM and TPVM

3.4 The XTPVM Interface

The XTPVM library is written in C and makes calls to TPVM and PVM. Both TPVM and PVM are portable across many platforms, and this makes the XTPVM library also portable. XTPVM is intended to work on a heterogeneous group of workstations and this has been accounted for (e.g,. whenever a pod is created on a machine, all function pointers are reevaluated).

The following is a list of all the library calls the programmer can use. For a more complete description, refer to [8].

- *xtpvm*_init(*threads, memnames, memlocs, memtypes*): The user should supply this function. In it, thread names, thread function pointers, shared memory names, and shared memory addresses should be set.
- *xtpvm_main*(): The user's main function.
- *xtpvm_end*(): This function terminates all the pods, frees all allocated buffers, and then halts the PVM system for a clean termination.
- *xtpvm_beginthread*(*functionname, instance*): starts a user thread with the XTPVM name given in *functionname* and *instance*.
- *xtpvm_beginthreads*(*functionname, startinstance, number*): starts a set of user threads with the XTPVM name given in *functionname*, starting with a *startinstance* and ending with *startinstance+number*.
- *xtpvm_endthread*(*functionname, instance*): broadcasts a message to all thread spawners to terminate a thread.
- *xtpvm_endthreads*(*functionname, startinstance, number*): broadcasts a message to all thread spawners to terminate a group of threads.
- *xtpvm_getexitcondition*(): checks if a thread is marked for ending condition or not.
- *xtpvm_setloadband*(*band*): sets the value of *delta* to be used.
- *xtpvm_send*(*threadname, instance, messagetag*): non-blocking send of a message with a tag to a thread.
- *xtpvm_receive*(*threadname, instance, messagetag*): blocking receive waiting for a message with a tag from a thread.
- *xtpvm_nreceive*(*threadname, instance, messagetag*): non-blocking receive, returns false if no message exists, else true.
- *xtpvm_initsend*(): initializes the send buffer. should be called before *xtpvm_send*().
- *xtpvm_remoteput*(*name, number, loc*): copy to a named shared memory buffer from a user specified variable.
- *xtpvm_remoteget*(*name, number, loc*): copy to a user specified variable from a named shared memory buffer.
- *xtpvm_log*(*string*): writes *string* to the log file. A single log file is created for each pod.

- *xtpvm_exit*(): removes the calling thread from XTPVM.
- *xtpvm_getinstance*(): returns the instance number of a thread.
- *xtpvm_upkXXX*(*pointer, count, stride*): unpacks data of type *XXX* (can be byte, double, int, long, etc.) to address *pointer*. *count* items are unpacked and spaced *stride* appart.
- *xtpvm_pkXXX*(*pointer, count, stride*): packs data of type *XXX*.

3.5 XTPVM Internal Design

The *xtvpm_init* function associates user thread names with thread function pointers. It should be the first *xtpvm* function to be called. The user can also declare names to remote shared memory regions, and sets appropriate pointers. Since every pod will first call this function, the problem of a function pointer having a different value on different architectures is alleviated by reloading its value whenever a pod is created, thus heterogeneity is provided.

After calling *xtpvm_init*, XTPVM sets up the virtual machine by starting the PVM system if it is not running, adding the hosts specified in the *hosts.ini* file to the virtual machine, and spawning one pod per host. The *hosts.ini* file specifies the location of each pod on its corresponding host so that different pod versions run on their corresponding architectures. Every pod first exports all the threads declared by the user (so that threads can migrate to any pod in the virtual machine) and then exports and spawns two private XTPVM threads, the *thread_spawner* and the *load_monitor* threads as shown in Figure 3. If *xtpvm_init* returns TRUE, XTPM converts the master process into a pod, creating for it these two private threads, otherwise, a pod is spawned on the local machine with these two threads. The advantage of converting the master into a pod is that inter-process communication is eliminated on the local machine, thus increasing performance. Another advantage is that the *xtpvm_main* function is managed as a normal migratable thread.

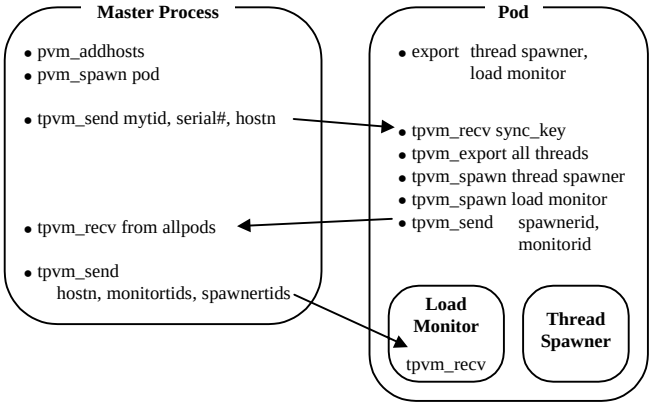

Figure 3: Creating a pod and setting the load monitor and thread spawner.

The pod and the master processes are in fact the same executable, but after calling *xtpvm_init* as the first function, both call *xtpvm_start* which will execute differently for a master process than a pod process. A process knows if it is a pod or a master just by making the PVM call *pvm_parent*, since only pods have a parent that spawns them and assigns them a serial number. The master process has no parent. This ensures that only the master process performs the virtual machine setup.

The *thread_spawner* and the *load_monitor* threads are non-migratable. They are exported and spawned by the main thread of every pod. Master processes that have changed into pods also own these two threads. All other threads that run in a pod are migratable user threads and the programmer can reference them by name.

The *thread spawner* is responsible mainly for user thread creation and termination. Send and receive commands require mapping from thread name to id for transparent access to threads, and this is done by multicasting a message to all thread spawners requesting them a lookup of a thread name. The thread spawner that has the thread currently running on its host replies. Thus, send and receive require thread names as the destination and source respectively and not thread ids. This is how location transparency is achieved. A mapping is also provided from thread ids to thread instances so that threads can know their instance number. This is useful for data parallelism, where each thread works on a separate section of the data.

The *load monitor* thread is responsible for calculating the load average at the local pod. It then sends this system load and the number of queued threads to the neighboring load monitor (the one in the pod with the next serial number) in a chain fashion to construct one common *load array*. The load array is used for initial scheduling as well as for migration (see section 4). The DGP algorithm is used to select threads for migration. The load monitor on the most loaded machine migrates the selected threads to the pod on the least loaded machine, while other load monitors do nothing.

4. Scheduling and Migration in XTPVM

4.1 Scheduling in XTPVM

Several static scheduling schemes have been proposed. Some of these include Source Processor Scheduling (SPS), Sorted Spiral Scheduling (SSS), and Global Plan Scheduling (GPS) [9]. All these can be categorized as static off-line scheduling since a schedule for the entire system is determined at design time, before the execution of the system. The advantages of this approach are its low run-time overhead, its deterministic behavior, its provision for system-wide optimizations, and the ease in which task dependencies and resource conflicts can be resolved, eliminating the need for costly resource locking and synchronization operations. The disadvantages of any static off-line scheduling include its inflexibility to adapt to a changing environment, and the difficulty of finding an optimal schedule (which is an NP-hard problem). Heuristics are often used to find a feasible schedule, which might result in low system utilization.

In XTPVM, we are concerned with the dynamic scheduling of threads over a set of workstations whose local loads might fluctuate as the threads run. Thus, we require a distributed scheduling scheme that takes into account local workstation loads and tries to balance these loads via migration of threads. The load metric used for initial scheduling of threads is the reported system load average since this adds a prediction factor as to where a thread might finish quicker. For migration, the load metric used is the number of ready threads queued at each workstation by a particular application owner. Thus,

XTPVM scheduling = initial scheduling + migration

Whenever the user wishes to spawn a thread, a call to *xtpvm_beginthread* is made, specifying the thread name and thread instance. The thread name is the name associated with a particular function specified in *xtpvm_init* by the user. A name, rather than a TPVM thread id, is used for location transparency since threads might migrate after being spawned. Since several instances of a thread function might be spawned, each instance is qualified by its

instance number, which is also required for communication and termination of threads.

Once this call is made, XTPM retrieves the *Load Array* from the *Load Monitor* thread in the caller's pod, and the least loaded workstation is selected as the destination to spawn this new thread. Once a destination host has been selected, a message is sent to the *Thread Manager* in the pod on that workstation requesting it to create a new thread in its encapsulating pod. The *Thread Manager* can choose either to directly spawn this new thread thus creating a context and committing it, or to place it on a queue until some currently running threads terminate. The maximum number of committed threads per host is determined as the number of processors available on the workstation multiplied by a thread *Commit Factor*. Thus, the *Commit Factor* is the number of threads committed per processor. Since flooding a processor with many threads simultaneously slows down the system, a *Commit Factor* is chosen to be the suitable number of threads a single processor can handle efficiently. Once committed, a thread cannot migrate. If a thread performs a blocking wait, XTPVM would be counting it as committed, when in fact it is not running, and another thread could be scheduled to run instead. To avoid this, just before blocking on a receive another thread on the queue is allowed to commit. This means that at times more than *Commit Factor* threads may be running to avoid deadlocks.

Another approach for the initial scheduling of *n* threads is to fairly distribute them over the available workstations and let migration move them around to achieve quickest execution. As the results presented later show, the initial scheduling scheme does not affect execution time since migration is performed frequently.

4.2 Distributed Global Plan (DGP) Algorithm

In XTPVM, we are concerned with the dynamic scheduling of threads over a set of workstations whose local loads might fluctuate as the threads run. Thus, we require a distributed scheduling scheme that takes into account local workstation loads and tries to balance these loads via migration of threads. One method is to use a Genetic Algorithm. We have implemented one that has chromosomes selected, reproduced and mutated in order to converge to the best possible distribution of a fixed size block of totally independent tasks that are determined prior to scheduling. When loads change on the workstations, the GA will adapt to account for such a change with a new distribution strategy of these blocks [10]. Even though a GA is adaptable and performed well in the experiments conducted, yet it is not useful for XTVPM, which requires scheduling of threads that might start and end at different times.

An alternative way of developing a scheduling policy for XTPVM is to investigate multiprocessor scheduling algorithms. These suffer from the global state problem, since it is impossible to know the current state of the entire system exactly, due to the latency of acquiring the information. When this problem is looked at from a distributed perspective, then the latency in acquiring information about the current state is much greater. Some systems such as the MARS architecture escape this problem via static scheduling. Kara has introduced a new algorithm called DGP (Distributed Global Plan) that addresses the problem of coherence and coordination and makes good local scheduling decisions without jeopardizing global goals. DGP is distributed since control is decentralized and no host has a true image of the overall state of the system. A scheduler is replicated on each host, and each scheduler:

- accounts for local decisions made by other schedulers,
- accounts for the effect of its local decisions on the system, and
- ensures load balancing

DGP thus prevents host overloading which occurs when several hosts target the least loaded host, which in turn becomes heavily loaded. Its use of the parameter *delta* (Δ) avoids job thrashing which is when jobs infinitely move around the network and hosts spend their time in redistributing jobs and little on executing local jobs. The DGP algorithm is based on a strategy called Global Plans (GP) that aims at maintaining all computational loads of a distributed system within a band Δ. We have analyzed DGP as a static scheme for scheduling on multiprocessors, and experiments have shown good performance both in speedup and efficiency [9], and thus it looks promising to use for XTPVM's migration algorithm.

The Global Goal is defined as follows:

A network of hosts is balanced if the load on all hosts are within a band called Δ, (where Δ is constant, and has the same unit as the load of the hosts). Furthermore, $\Delta_r(t)$ is defined as the minimum band that contains all loads at a time t. In other words, a system is balanced at time t if $\Delta \geq \Delta_r(t)$ [1,2]

To illustrate this definition, assume that two snapshots of a multiprocessor with five processors were taken at time t1 and t2. Also assume that the load metric is the number of jobs queuing for execution. Using the above definition Figures 4 represents the system at these two times and shows the individual loads of each host. Assuming Δ = 3, Figure 4A exhibits an unbalanced state because Δ(t1)=7 and therefore Δ(t1)> Δ. Figure 4B shows a balanced system since all loads belong to the band Δ since Δ ≥ Δ(t2)

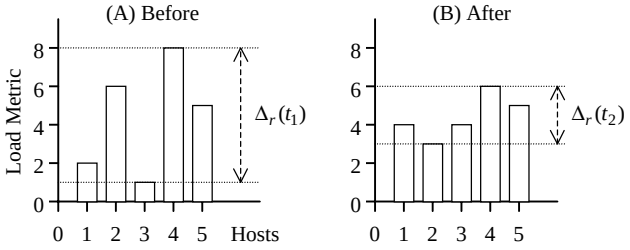

Figure 4: Loads of 5 hosts before and after balancing

The Global Plan Strategy aims to satisfy the global goal by ensuring that all loads are within *delta* and are thus balanced, preventing instability and host overloading. This is done via

- An Input Loads vector X
- A parameter Δ
- Output loads vector Y (withing the interval Δ)
- A table of global allocations T = { (p,q,r) .. } p units from host q to host r

In general GP can be described by the following pseudo-code

1. Y=X
2. compute the processor with lest load (l)
3. compute the processor with the highest load (h)
4. while (load of processor h - load of processor l) ≥ delta do
   - Search T for a matching triple (p,h,l) for any p
   - if Search successful then increment triple by 1 => (p+1,h,l)
   - else insert new entry (1,h,l)
   - Decrement load of processor h
   - Increment load of processor l
   - Compute the new l
   - Compute the new h

For example, if a network consists of 5 hosts, and we use Δ = 2 for the initial load vector X = (2,7,3,6,3), then T would develop as follows:

Step #	Y	Δ at t	Status	T
0	(2,7,3,6,3)	5	unbalanced	{}
1	(3,6,3,6,3)	3	unbalanced	{(1,2,1)}
2	(4,5,3,6,3)	3	unbalanced	{(2,2,1)}
3	(4,5,4,5,3)	2	balanced	{(2,2,1) ,(1,4,3)}

Table 1 : The development of the table of global allocations in GP.

So to move from the unbalanced state (2,7,3,6,4) where host1 has load 2, host2 has load 7 and so on, to the balanced state (4,5,4,5,3), T tells us to take 2 tasks from host2 and give it to host1, and 1 task from host4 to host3. As one can see, the final state has a maximum host load of 5 and a minimum host load of 3, and since Δ was selected to be 2, then such a state would have the hosts balanced. The GP strategy is executed periodically depending on how important system balance is to the user. Since XTPVM will use this strategy, the library calls *xtpvm_setloadband* and *xtpvm_setloaddelay* are required for programmers to set the load band and the delay between load broadcast periods respectively.

The application of the GP policy to a distributed algorithm leads to several considerations. The algorithm based on the GP policy is called Distributed GP or DGP because of the distributed nature of the environment in which GP is applied. Each host only has control over its own resources and local load information is periodically broadcast. The GP algorithm is executed at each machine to create *T* and each machine *i* executes all entries that include it as a source (*, *i*, *) or as a destination (*, *, *i*) depending whether it is sender initiated or receiver initiated.

4.3 Dynamic Load Balancing and Migration in XTPVM

Recall that load monitors cooperate in a chain fashion, passing their loads around back to the first load monitor which then broadcasts this load. In fact, not only are the load averages circulated this way, but with them the load managers circulate the length of their ready queue in terms of the number of threads queued in it. This in effect generates the input loads vector, X, discussed in DGP that must be balanced to be within Δ.

XTPVM uses a third parameter in its specialization of DGP other than Δ and the *Commit Factor* discussed previously. This is the *Threshold_to_start_DGP* parameter. To prevent excessive migration, we can control migration to initiate it only when queues are empty enough. Whenever the minimum queue load is less than *Threshold_to_start_DGP* then migration is invoked, otherwise nothing happens and execution of committed threads continues as normal.

5. Performance Evaluation

This section examines the effect of changing XTVPM parameters on the overall execution time of an application. To assess the performance, we declared 100 thread instances of the same function but with different instance numbers. The thread function loops one million times to consume CPU cycles. Upon termination, a message is sent from a thread instance to its parent thread indicating its completion before calling *xtpvm_exit* to remove itself from the list of running threads. The parent thread waits for 100 such messages and calculates the time difference between starting these threads and receiving all 100 messages. This is the total time of execution of the threads, which changes as different values are chosen for XTPVM parameters.

5.1 The Effect of *Commit Factor*

The first experiment aims at finding the optimal number of committed threads per processor. Recall that a committed thread is the one for which a context has been created. Once committed, a thread cannot migrate. The *Commit Factor* in this experiment was ranged from 1 to 10 while other parameters were kept constant. The load monitor was set to run once every second (*Load Delay* =1), a DGP *delta value* of 2 was chosen, and the *Threshhold_to_start_DGP* was set to a relatively large value, 100, so that migration can take place anytime when enabled, and its effects can be obvious. A value of 100 was chosen since a queue will always have less than 100 threads in this experiment. The experiment was first performed with migration disabled on 2 hosts, and with migration enabled on 2 hosts and then 5 hosts (Sparc 5 workstations running Solaris 2.x). All hosts had a relatively low load average, ranging from 1 to 1.5 as reported by the system during the experiment. Figure 5 shows the results.

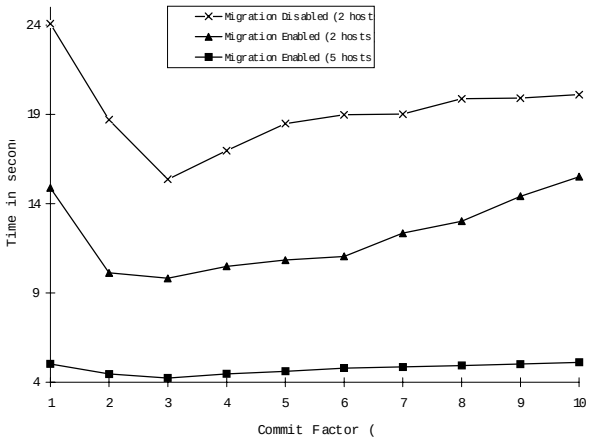

Figure 5: Effect of commit factor on execution time

The curves of Figure 5 have a common pattern, regardless of whether migration is enabled and the number of hosts used. As the number of committed threads per host increases, the overall execution time improved up to a certain point (3 in our example), due to the ability of each host to run more than one thread. The optimal commit factor value can vary from one application to another depending on what threads are doing. For example, if a committed thread waits on I/O, another committed thread can run making better use of CPU time and decreasing overall execution time. Increasing the number of committed threads beyond a certain value made XTPVM perform worse, since more time is wasted in creating new contexts for threads and switching between them. Furthermore, increasing the number of committed threads will decrease the number of threads that can be migrated and will imbalance the loads of workstations.

Besides the effect of *Commit Factor*, Figure 5 demonstrates the advantage of using migration in XTPVM and the speedup achieved when using many hosts. The advantage of migration is apparent, especially when initial scheduling is not proper or when the hosts have different loads or speeds. Although the upper and middle curves of Figure 5 were obtained by running the experiment on two hosts, the middle curve, with migration enabled, has much better performance. This is because the initial scheduling algorithm did not distribute the threads properly resulting in a load imbalance. However, with migration enabled, the *Load Monitor* maintained a

load balance, achieving high CPU utilization since all hosts are almost always busy.

The sequential (non-XTPVM) version of the application ran in 22.81 seconds. The XTPVM version ran in 14.88 seconds on two hosts and 5.02 seconds on five hosts with one thread committed per host at a time, thus achieving a speedup of 1.53 and 4.54 respectively. When 3 threads are committed per host, the execution times were 9.81 seconds on two hosts and 4.23 seconds on five hosts. The speedups were 2.32 and 5.39 respectively. The super-linear speedups were because the threads were doing I/O writing data to a file. Committing more than one thread per processor made better use of CPU time. While a thread is waiting for I/O, another thread ran reducing the overall execution time.

5.2 Changing the Initial Scheduling policy

We now examine the effect of initial scheduling policy on overall execution time. Figure 6 depicts the number of non-committed threads in the ready queues of two hosts at each migration step. The initial scheduling policy of Figure 6A assigned all the newly created threads to one host. At first, Q2 was empty due to the initial scheduling scheme, but after the first migration, the queues were balanced out with 48 threads each. One host was faster than the other, and was able to run more of its queued threads. When the *Load Monitor* ran for the second time, it balanced the loads to 42 and 41 (difference less than Δ=2). The end effect is that threads are migrated from slower to faster hosts. Even though the initial execution policy was improper, migration had compensated for it.

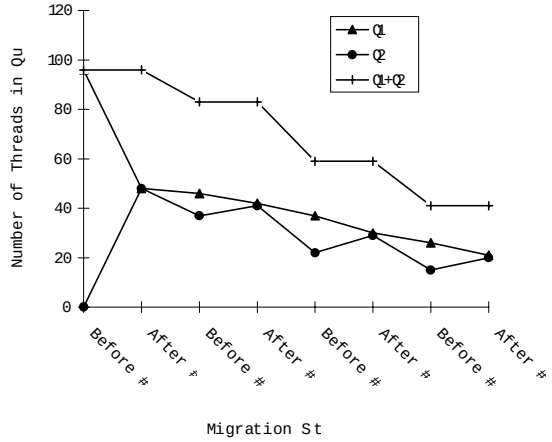

Figure 6A: Balancing the thread queues with initially imbalanced queues.

When the initial scheduling policy fairly divides the threads among the hosts, the changes in queue sizes were very similar to the first situation as shown in Figure 6B. The overall execution times in both cases were very close (14.9 and 14.5 seconds). Thus, initial scheduling does not play a significant role when the number of threads is large and migration is enabled. It should be emphasized that migration has a very small overhead in XTPVM, since only one message is required to specify the *n* threads to be migrated, just by sending *n* thread names and instances. The message is also small in size since no thread context is transferred. This is to be contrasted with alternative methods of migration in which the entire thread context (code, data, and state) is migrated.

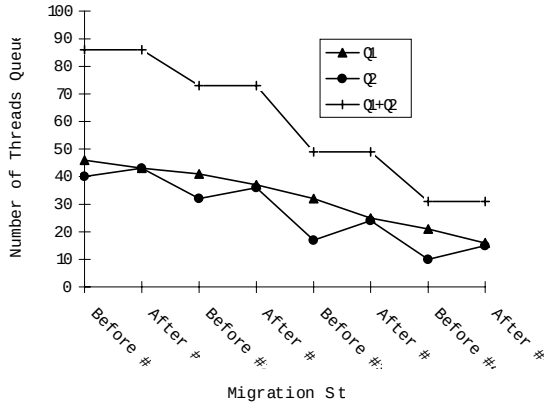

Figure 6B: Balancing the thread queues with initially balanced queues.

5.3 The Effect of *Threshhold_to_start_DGP*

An experiment similar to that in section 5.1 was performed but with a different *Threshhold_to_start_DGP* value. Instead of setting it to 100, a value of 10 was chosen this time. It was noticed that migration took place usually once only rather than 4 or 5 times as with a value of 100, and this had a noticeable effect on the total execution time only when the hosts had noticeably different loads. More frequent migration of threads to the fast host meant that the fast host was executing most of the threads while less frequent migration didn't migrate too many threads to the fast host. For example, out of 100 threads, the fast host executed an average of 66 threads when *Threshhold_to_start_DGP*=100 compared to an average of 64 when *Threshhold_to_start_DGP*=10.

6. Related Work

Introducing a new abstract machine necessitates a comparison with the existing ones to see the similarities and differences, the advantages and disadvantages. We already discussed the differences between XTPVM, PVM and TPVM. We will now compare XTPVM with other abstract machines surveyed.

6.1 Orca/Panda and HAWK

These deal with objects as the major item of abstraction and not threads, and thus use the Object-Oriented programming paradigm, which is at a higher level of abstraction. In both VMs, objects are shared and replicated transparently, and in HAWK they are also partitioned transparently. Unlike XTPVM, these VMs do not have provisions for threads and the process model of programming. XTPVM, on the other hand, does not replicate or partition shared regions as in Orca/Panda and HAWK [11,12].

6.2 TAM and LAM

These expose communication, synchronization, and scheduling of threads to allow compilers to optimize for important special cases, whereas no compiler optimization is provided in XTPVM. They also deal with threads but do not transparently migrate them [13,14]. They run on a single multiprocessor with only one running activation frame at any moment in time possibly with several threads, while XTPVM can work on multiprocessors and effectively exploit them, allowing multiple threads in different address spaces (pods) to concurrently run on the same machine. Thus, in TAM and LAM, synchronization is only among threads of same activation frame rather than threads of different activations, whereas in XTPVM

threads in different pods, possibly on different hosts, can synchronize.

6.3 LPVM

LPVM is a Lightweight process version of PVM but with a different user interface. It has threads that are thread safe as its basic unit, but it is specifically targeted at symmetric multiprocessors that support threads and global shared memory and does not provide smart scheduling and migration of these threads over a network of workstations[15].

6.4 MPVM and MIST

Just like XTPVM, MPVM is also based on PVM, but migrates processes rather than threads over a pool of workstations, and the system MIST has been build to support task migration, application checkpointing, and multi-user application execution, having an MPVM kernel [16,17]. MIST has a Multi-user Migratable PVM kernel, which is an enhanced version of PVM that supports transparent task migration, application checkpointing and multi-user application execution, but not thread migration. It makes use of the enhanced version of the resource manager interface provided by PVM [16].

7. Conclusion and Future Enhancements

XTPVM, a transparent thread scheduling and migrating virtual machine, has been implemented as a layer on top of TPVM and PVM to simplify the task of parallel programmers producing applications for a virtual parallel computer constructed by the cooperation of several hosts on a LAN. The DGP algorithm was used to initiate the migration of ready non-committed threads queued in a FIFO queue, waiting to be committed on the current host. An experiment was performed to show the advantage of queuing threads and not committing them as soon as they are ready.

A future enhancement to include in XTPVM is fault tolerance since PVM makes no attempt to automatically recover tasks that are killed because of host failure but leaves this task to the application programmer. Another feature is to make the commit factor change dynamically at runtime according to the behavior of threads.

REFERENCES

[1] M. Kara, "Simulation and Prototyping of a Coherent Distributed Dynamic Load Balancing Algorithm", Research Report Series, School of Computer Studies, The University of Leeds, Report 97.17, May 1997.

[2] M. Kara, "Using dynamic load balancing in distributed information systems" Research Report Series, School of Computer Studies, The University of Leeds, Report 94.18, May 1994.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, "PVM: A User's Guide and Tutorial for Networked Parallel Computing", MIT Press, 1994.

[4] A.S. Tanenbaum, H.E. Bal, and M.F. Kaashoek, "Programming Multicomputers Using Shared Objects", In *Proceeding of the Third International Workshop on Object Orientation in Operating Systems* (IWOOOS'93), pages 199--202, December 1993.

[5] A. Ferrari, and V. Sunderam, "Multiparadigm Distributed Computing with TPVM", Technical Report CSTR-951201, Department of Mathematics and Computer Science, Emory University, December 1995, Submitted to the *Journal of Parallel and Distributed Computing*, Special Issue on Multithreading for Multiprocessors

[6] A. Ferrari, and V. Sunderam, "TPVM: Distributed Concurrent Computing with Lightweight Processes", *Proceedings of IEEE High Performance Distributed Computing 4*, Washington, D.C., pp. 211-218, August 1995.

[7] A. Ferrari, and V. Sunderam, "TPVM: A Threads-Based Interface and Subsystem for PVM", Technical Report CSTR-940802. Department of Math and Computer Science, Emory University, Atlanta, August 1994.

[8] T. Abdel-Radi, "XTPVM: A Transparent Thread Scheduling and Migrating Abstract machine", Master thesis, April 98, AUC.

[9] M. Mahmoud, A. Abdelbar and T. Abdel-Radi, "A Framework for Analyzing Multiprocessor Scheduling", Submitted to *PDCS-98*, Chicago, Illonois.

[10] A. Sameh, T. Abdel-Radi, and I. Khalil. "Scheduling jobs using a Genetic Algorithm in a Distributed Environment", 6th *International Conference on Artificial Intelligence Applications* '98 (ICAIA'98).

[11] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, "Orca: A Language For Parallel Programming of Distributed Systems", *IEEE Transactions on Software Engineering*, 18(3):190-205, March 1992.

[12] R. Bhoedjang, T. Ruhl, R. Hofman, K. Langendoen, H. Bal, and M.F. Kaashoek, "Panda: A Portable Platform to Support Parallel Programming Languages," *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, San Diego, pp. 213-226, Sep. 1993.

[13] T. Eicken, D. Culler, S.C. Goldstein, and K.E. Schauser, "TAM - a Compiler Controlled Threaded Abstract Machine", *J. Parallel and Distributed Computing*, 1992.

[14] S. Davis, "The Liquid Abstract Machine", MIT Transit Project, Transit Note #86, October 1993.

[15] H. Zhou, and A. Geist, "LPVM: A Step Towards Multithreaded PVM", Oak Ridge National Laboratory. Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge.

[16] J. Casas, D.L. Clark, P.S. Galbiati, R. Konuru, S.W. Otto, R.M. Prouty, and J. Walpole, "MIST: PVM with Transparent Migration and Checkpointing", presented at the 3rd Annual *PVM Users' Group Meeting*, Pittsburgh, PA, May 7-9, 1995.

[17] J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R.M. Prouty, and J. Walpole, "MPVM: A Migration Transparent Version of PVM," Technical Report CSE-95-002, Oregon Graduate Institute of Science and Technology, February 1995.