Multiparadigm Distributed Computing with TPVM *

Adam Ferrari
Department of Computer Science
University of Virginia, Charlottesville, VA 22903, USA
ferrari@virginia.edu

and

V. S. Sunderam
Department of Mathematics and Computer Science
Emory University, Atlanta, GA 30322, USA
vss@mathcs.emory.edu

Abstract

Distributed concurrent computing based on lightweight processes can potentially address performance and functionality limits in heterogeneous systems. The TPVM framework, based on the notion of "exportable services", is an extension to the PVM message passing system, but uses threads as units of computing, scheduling, and parallelism. TPVM facilitates and supports three different distributed concurrent programming paradigms: (a) the traditional, task based, explicit message passing model; (b) a data-driven instantiation model that enables straightforward specification of computation based on data dependencies; and (c) a partial shared-address space model via remote memory access, with naming and typing of distributed data areas. The latter models offer significantly different computing paradigms for network-based computing, while maintaining a close resemblance to, and building upon, the conventional PVM infrastructure in the interest of compatibility and ease of transition. The TPVM system comprises three basic modules: a

*Research supported in part by NASA, DoE, and NSF under grants NAG 2-828, DE-FG05-91ER25105, ASC-9214149

library interface that provides access to thread-based distributed concurrent computing facilities, a portable thread interface module which abstracts the required threads-related services, and a thread server module which performs scheduling and system data management. System implementation as well as applications experiences have been very encouraging, indicating the viability of the proposed models, the feasibility of portable and efficient threads systems for distributed computing, and the performance improvements that result from multithreaded concurrent computing.

1 Introduction

Distributed concurrent processing on heterogeneous networked computers is now commonplace. The standard methodology in network computing is for a software system to emulate general purpose concurrent computing facilities, and present a programming interface, typically based on explicit message passing. These software systems permit users to configure "virtual parallel machines" using networked computer systems, and execute their computations with varying degrees of logical and real concurrency – depending upon the number of physical machines used. For many classes of applications, loosely coupled concurrent computing in such environments has proven to be straightforward in terms of parallelization effort, satisfactory to very good, in terms of performance, and highly effective in terms of cost. Numerous software systems are in existence that support network based parallel computing [17, 3], although the majority of use is probably based on a few popular ones, such as PVM [9], MPI [11], and P4/Parmacs [2].

However, despite their widespread adoption and abundance of success stories [15, 5], network-based concurrent computing systems suffer from several critical shortcomings. Principal among them are factors influencing, or in some way related to, performance. The relatively slow networks and communications software typical in heterogeneous networked environments usually preclude high efficiencies. Cardinal rules for good performance in network-based computing are large granularity, infrequent communication using the largest possible messages, and avoidance of highly synchronous algorithms, even if these requirements are unnatural to the problem at hand. In addition, a serious obstacle to good performance is the general purpose, non-dedicated, nature of typical network-based environments — external influences, including those generated by other users as well as by operating systems and network software, lead to load imbalances and intractable scheduling issues eventually resulting in degraded overall performance. We believe that only some of the above factors are fundamentally inherent and unavoidable. Obviously, physical limitations such as network bandwidth and shared-medium design (as in Ethernet and FDDI) cannot be overcome. However, certain strategies, such as increasing asynchrony, enabling overlapped communication and computation, dynamic scheduling, and reducing load imbalances are feasible and will likely result in significant performance improvements.

Almost all software systems that support distributed concurrent computing use (1) a process based computing model; and (2) a message passing communications model. Processes have been the most convenient and universally supported task unit on most types of machines, and message passing has become the paradigm of choice for a number of reasons. Recently however, several developments have occurred to prompt an investigation of lightweight processes or "threads" as units of parallelism and scheduling. First, experiences with message passing parallel computing have highlighted greater asynchrony as one avenue for performance improvement. Second, especially in heterogeneous cluster environments, processors with multiple (2-8) CPU's and global memory are becoming commonplace; when collections of such machines are used as a parallel computing platform, the traditional process-oriented message passing model is suboptimal. Finally, new concepts such as active messages [18] and remote memory copy are attractive programming abstractions, but are cumbersome to implement efficiently within a process-oriented framework.

We propose an alternative paradigm, based on light-weight processes and event driven computation, to attempt to address some of the hindrances in network-based concurrent computing systems, and to exploit the new advances in architectures and operating systems. This approach, which is derived by combining variants of principles from multithreading systems, data flow computing, and remote procedure call, is believed to have the potential to enhance performance and functionality in heterogeneous systems, without too drastic a departure from the prevalent parallel programming methodologies. TPVM is an experimental system for investigating these issues. It is an auxiliary subsystem of PVM in which threads are the units of parallelism and scheduling; the system provides a supplementary programming interface that enables programs to avail of the extended facilities, and the system itself is implemented as a non-intrusive layer over the native release version of PVM. The computing models supported by TPVM and examples of their use, salient implementation features, and performance/functionality experiences are described in this paper.

2 Background

Heterogeneous concurrent computing offers the potential for optimal algorithm-architecture assignment, dynamic load allocation across geographically distributed resources, and other sophisticated parallelization and operation strategies. In reality, however, the complexity of exploiting heterogeneity and distribution is so high that most applications are

SPMD programs that are derived from, and partitioned in a manner identical to homogeneous MPP's. It is not surprising, therefore, that such regular and symmetric applications often perform poorly in typical heterogeneous and clustered network environments.

It is therefore tempting to conclude that the widespread use of network-based systems is by and large limited (a) to large-grained applications; and/or (b) small-scale settings; and/or (c) research or experimental arenas. Such a judgment is not necessarily derogatory; network-based systems have high value as stable, pragmatic, portable technologies that at least provide an entry level solution to parallel computing in terms of both hardware and software, i.e. zero-investment parallelism is made available to any user interested in high performance computing. The above discussion is not meant to imply that network-based systems perform poorly for applications with substantial communications content. To demonstrate that reasonable performance can be delivered by network-based systems when granularity is not excessively fine, especially when increasingly popular high-speed networks are used, we present in Table 1 results of recent experiments with the NAS Parallel benchmarks [1]. This table shows excerpted performance numbers from a detailed benchmarking effort [19] aimed at analyzing efficiency issues in network-based high-performance computing, using different types of environments, for a widely accepted and representative class of scientific applications.

Platform/ Benchmark	Cray Y-MP 1	i860 128	PVM			
			Comm. Volume	Num. Messages	8 RS6K, FDDI	8 SGI, Gswitch
Multigrid Method	22 secs	9 secs	192 MB	1808	110 secs (30 idle)	168 secs (50 idle)
Integer Sort	11 secs	14 secs	560 MB	2491	318 secs (129 idle)	258 secs (105 idle)
Fourier Transform	29 secs	10 secs	1500 MB	826	412 secs (34 idle)	228 secs (34 idle)

Table 1: Representative NPB timing results on PVM

Several points are worthy of note in the table. One is that realistic algorithms with substantial communication content execute on small PVM clusters about an order of magnitude slower than on a 1-CPU vector supercomputer or 128-node MPP. More important, however, is the apparent potential for improvement by up to 10 to 50%, judging by the ratio of idle time (spent blocking on receives within the slowest process) to overall execution time. This factor is due in most part to load imbalances — in network-based systems, dynamically varying external influences, operating system activity, paging, and background network traffic result in some processes executing at rates different from others. Further, the algorithms that were used are loosely synchronous — i.e., all processes compute, then communicate, then compute, and so on, thereby precluding any overlap of communication and computation without costly algorithmic re-design.

In this paper, we describe a novel experimental methodology devised as an enhancement to the PVM system that is aimed at attacking the above limitations and inefficiencies. This approach is based on the use of lightweight processes (or threads) as both the unit of parallelism and of scheduling, i.e. workload allocation and data/function partitioning in the new scheme will be in terms of threads, and the the computational entities managed by the system will also be threads — as opposed to processes in the conventional PVM system. In this model, each heavyweight process will be comprised of multiple lightweight processes that will, in the abstract, be asynchronous, independent threads of control, capable of communicating and synchronizing with other lightweight processes. This fundamental change in the nature of parallelism offered by PVM will, we believe

- increase the potential for, and the realization of, asynchrony and communication/computation overlap, with little programmer effort
- allow application parallelization at smaller, perhaps more natural, granularity without sacrificing efficiency, while enabling responsive scheduling of computational tasks in an adaptive and dynamically load-balanced manner
- enable alternative computing paradigms based on address-space sharing and data-driven computing, with substantial improvements in functionality and performance

2.1 The TPVM Model

The TPVM system retains, and is compatible with, the essence of the PVM computing model, *viz.* an application is comprised of "components" or sub-algorithms, each of which may be manifested as a collection of instances that cooperate via messages. This model is straightforward but nevertheless comprehensive enough to be suitable for most application programs. TPVM essentially supports this paradigm with component instances being realized as threads or lightweight processes rather than as standard heavyweight processes. Threads in TPVM are hosted within regular PVM tasks and form the units of parallelism and scheduling. In other words, a collection of TPVM threads that comprise an application are created and execute within the context of a (usually smaller) number of regular processes or "pods" (as in, e.g. SunOS LWP [16]). TPVM exploits multiple-CPU SMMP's by concurrently executing threads, but is also very beneficial in single-CPU systems in terms of program asynchrony and latency hiding.

The TPVM model requires that host processes be instantiated prior to the active computational entities, i.e. the threads, being started. Host processes or pods are initiated via normal native PVM mechanisms, e.g. `pvm_spawn()` or the command line. These PVM tasks are required to "export" thread entry points that specify the types and numbers of threads that the process is willing and able to host. These exports establish a set of latent threads within each host process; operational threads are instantiated at a later stage, depending on the application program structure and the particular variant of the TPVM model being used.

TPVM supports three different, composable basic programming paradigms that are described below, as well as a number of additional features having efficient implementations and specific utility given the underlying threads-based model. Figure 1 depicts the basic TPVM model – processing elements are realized as threads which are hosted within PVM tasks, and which exchange messages. The underlying communication medium through which messages are passed (sometimes a shared address space, sometimes a PVM managed communication network) is encapsulated by the TPVM programming interface.

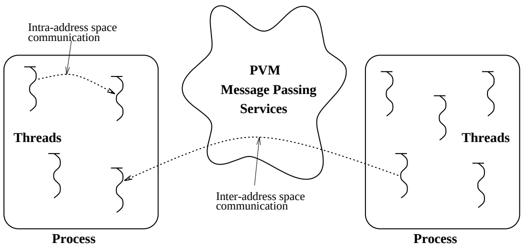

Figure 1: The basic TPVM model.

2.2 The Process Paradigm

The "default" or process-based TPVM computing paradigm is essentially the same as the PVM computing paradigm, in terms of the programming model. Parallel applications are comprised of sub-tasks with disjoint address spaces which cooperate via explicit asynchronous message passing. The TPVM process model retains this familiar and well-understood paradigm, while transparently (to the programmer) availing of the performance, concurrency, and shared-address-space benefits of multithreaded computing. The notion is simple – parallel programs are developed assuming a process oriented computational structure, but realized as lightweight processes or threads. Programs are developed assuming that parallel tasks will interact via message passing, thus naturally precluding the use and assumption of shared global variables. The model in fact suggests that program development follow this convention, i.e. one of thinking in terms of a distributed memory multiprocessor framework with no means of communication between component task units other than explicit message-passing. At the coding stage, a simple "shell" around the specification of the computational component program is constructed, that enables the TPVM system to realize the parallelism units as threads.

In the default TPVM process model therefore, PVM processes are first spawned, but are not themselves the computational units of parallelism and/or scheduling. The PVM processes "export" entry points describing computational units – these are subroutines (that may or may not be the interface to a collection of procedural modules) that describe the actions of a concurrent computational entity. Activations of these computational entities are generated by equivalents of the conventional process startup mechanism – in the case of TPVM, by the `tpvm_spawn()` primitive. In the TPVM system, these computational units are realized as threads, each with a unique addressable identifier (termed the thread id, or *ttid*). From this point on, the TPVM model simply presents a traditional message passing paradigm, i.e. threads send and receive messages based on *ttids*, and this is assumed to be the only mode of interaction.

2.3 Alternative Paradigms

The design of the TPVM system was influenced by two somewhat conflicting objectives – a desire to retain a well understood programming paradigm based on asynchronous communicating sequential entities, and, simultaneously, to exploit the performance and functionality enabled by lightweight processes. Our compromise was to provide related, but independent and separate, programming models within the TPVM system. The previous subsection outlined the traditional process-oriented legacy model in TPVM; here we briefly discuss the alternative paradigms supported.

2.3.1 The Data Driven Paradigm

Based on the observation that workload allocation and scheduling are critical in network based computing systems where resources are not dedicated, and further, that uncontrollable and random external resource demands are common, we propose a straightforward scheme that delays binding of work units to computational entities. The reasoning is that delayed binding and scheduling only when data is available, will assign resources to tasks more optimally. This concept, somewhat related to dataflow computing, has the potential to be very effective in heterogeneous environments. Essentially, "shells" of computational entities (i.e. threads) are initially created at appropriate locations in a heterogeneous platform, but are not activated. These threads specify their data requirements in terms of message tags – i.e. they indicate that activation requires the availability of a set of specified messages. Other components of the application inform the scheduling subsystem when they generate these messages; when a complete set has been formed, the scheduling system activates a thread at the most suitable location as determined by the instantaneous state of the system, and causes the trigger messages to be delivered.

A specific instance of the data-driven TPVM paradigm essentially resembles the notion of "active messages"; a thread dependent on a single message type is created at a specific location (i.e. within a specific process pod), and is activated immediately upon the arrival of a message of that type – an action equivalent to that of active messages, with their attendant advantages.

2.3.2 Remote Memory

Independent of the instantiation paradigm used in a TPVM application (process based, data-driven, or some composition of these), TPVM threads can avail of "remote memory", a translucent compromise between distributed shared memory and completely disjoint address spaces. Remote memory allows asynchronous read and write of explicitly exported portions of a thread's address space by another, even when they are in different process pods, possibly on different machines. Of course, consistency semantics have to be established, either by convention or by provided PVM and TPVM concurrency control constructs.

Remote memory in TPVM is useful for decoupling the location of data structures in a distributed application from the implementation of threads which define and/or use those data structures. In distributed memory parallel programs, data structures are typically sent and received in fixed patterns, or are handled by an explicitly programmed server. In TPVM, the asynchronous manipulation of remote data structures by clients can be made automatic and transparent to the programmer using the remote memory abstraction.

3 TPVM Programming

The logistics of programming the TPVM system are deliberately straightforward, and share little in common with traditional threads or SMMP programming. Programmers do not use native thread primitives, semaphores, locks, or other typical thread related management and synchronization abstractions. Instead, users write concurrent distributed applications using a set of provided TPVM library calls (as opposed to a language interface or automatic compilers, etc.) that present the portable TPVM thread-management, communication, synchronization, and other miscellaneous primitives. In other words, the

TPVM programming model involves the use of library functions that provide an interface to the TPVM run-time system. These library calls support the programming paradigms described above (a process-based/message passing model, a data-driven instantiation model, and remote memory) and other related features.

Programming using the lightweight process based message passing model is only cosmetically different from programming in standard PVM. The basic elements are the same — the program must spawn processing elements which must then use message passing and data marshaling primitives to cooperate in solving the problem at hand. This model allows existing PVM applications to be easily ported to take advantage of TPVM facilities, and supports a well known programming style which is easy to adapt or convert to from any traditional message passing system. The data-driven model, on the other hand, is intended to simplify the development of distributed parallel programs by providing automatic thread instantiation based on data dependencies. This model does not require explicit thread creation or message passing, and thus eases the complex state management tasks required of DMMP programmers and facilitates the construction of composable program elements. These two models may be mixed to form hybrid programs, and may also be combined with usage of the remote memory abstraction.

Before examining the details of TPVM programming, it is worth reiterating that the use of TPVM is implicitly dependent on the use of PVM. I.e., preliminary PVM actions such as virtual machine establishment and process creation are necessary before any of the TPVM constructs may be utilized. Since TPVM threads exist within the context of PVM tasks, the standard PVM library routines must be used to create and synchronize PVM processes. Besides the necessary usage of PVM in a TPVM program, many of the "incidental" details involved in properly using PVM are retained in TPVM for the sake of consistency and ease of porting existing PVM software. For example, the message buffer management routines included in PVM have corresponding versions in TPVM with essentially identical semantics.

The starting point for using both basic TPVM programming models is the `tpvm_export()` library function. A running PVM task may declare itself a potential "host" or "containing pod" for a given type of thread using the this routine. For example:

```
int rules[2] = { 1, 2 };
tpvm_export("IntSort", sortfunc(), PvmDflowModel, 2, rules);
/* flags might also be PvmProcModel */
```

The first parameter to `tpvm_export()` specifies the name of the declared thread, a label which may be used throughout the virtual machine to identify a given type of thread, just as a "component name" is used in PVM. The second parameter indicates the code entry point in the calling process for the thread type named. The entry point for similarly named threads in different processes may be different; the TPVM scheduler, however, does not differentiate between threads of the same name declared by different tasks in making scheduling decisions, so care must be taken to avoid name clashes.

The third "flags" parameter of the `tpvm_export()` call is used to determine if the declared thread will be of the message passing variety (`PvmProcModel`) or of the data-driven variety (`PvmDflowModel`). Like named threads may not be declared with different "flags" values or a run-time error will occur. In the case illustrated, the "flags" parameter indicates that the declared thread is of the data-driven type, and thus the `tpvm_export()` call must also define the thread's data dependencies. This is accomplished via the fifth and sixth "rules" specification parameters, which indicate the number and message tags of the declared thread's data dependencies. Similarly named data-driven threads declared by different tasks must specify the same data dependencies. For `PvmProcModel` threads, the "rules" specification parameters are ignored. The semantics of data-driven model threads are examined further below.

3.1 Message-based TPVM Programming

Once threads have been declared, the user may deploy the TPVM message passing model of programming using the `tpvm_spawn()` instantiation function and TPVM message passing primitives. Programmers should include synchronization in their TPVM code to ensure that calls to `tpvm_export()` to declare a given thread precede calls to `tpvm_spawn()` to create instances of that thread. An attempt to spawn an undeclared thread will cause a run-time error, even if that thread is declared at some later point (i.e., concurrency control is left to the programmer). Use of `tpvm_spawn()` is similar to `pvm_spawn()`:

```
tpvm_spawn("MatMult", PvmThreadWild, 0, nthreads, ttids);
```

The first parameter to `tpvm_spawn()` indicates the exported name the type of thread to be instantiated (as it was specified by a call to `tpvm_export()`). The second parameter specifies the scheduling discipline which will be employed; currently implemented disciplines include `PvmThreadLocal` which indicates that the thread should be created in the calling PVM task, `PvmThreadWild` which indicates that the scheduling decision should be left to the TPVM run-time system, and `PvmThreadTask` which indicates that the thread should be created in the task (i.e. within the process pod) indicated by the third parameter. If `PvmThreadTask` is not specified, the third parameter is ignored. The fourth parameter indicates the number of thread instances to create.

The final parameter is an array into which system-wide unique thread identifiers (commonly referred to as *ttids*) of the newly spawned threads will be stored upon return.

A call to `tpvm_spawn()` blocks until all requested threads have been created and the *ttid* array can be filled, after which the number of threads instantiated by the call is returned.

Use of the thread identifiers returned by `tpvm_spawn()` as message source and destination identifiers is central to the TPVM message passing model; they must be disseminated by the `tpvm_spawn()` invoker to all threads which need to communicate with the newly created threads. A thread may invoke `tpvm_mytid()` to determine its own thread identifier value, or `tpvm_parent()` to determine the identity of the thread that spawned it. This scheme is similar to the situation with task identifiers (*tids*) in PVM.

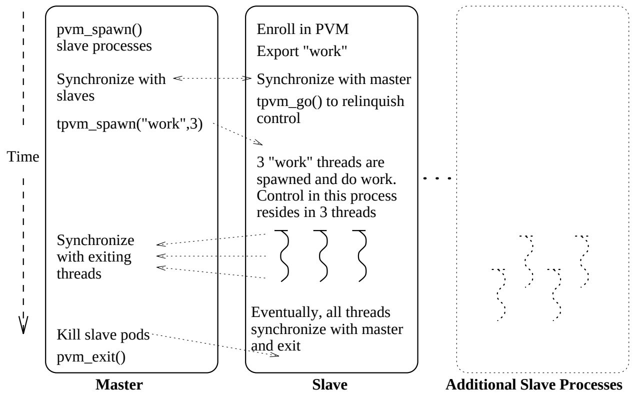

Figure 2: Typical sequence of events in a TPVM message passing application.

A typical model for a simple master slave TPVM application is illustrated in figure 2. Here we see the essential elements already discussed. Also illustrated is the fact that each pod in TPVM is created with a "default" thread of control (which is defined to begin execution at the function "`tpvm_main()`"). This default thread will often permanently relinquish control after exporting the thread types which which may be scheduled in its process. The `tpvm_go()` library routine is provided to accomplish this. This call is essentially equivalent to `tpvm_exit()` (which terminates a calling thread), but has the safety features that it may only be called by the default thread of control, and will flag a run-time error if no thread exports are active (a local deadlock situation). It should be noted that the default thread of control in a process is a first class TPVM processing element and is not required to relinquish control via `tpvm_go()`; it may participate in TPVM message passing no differently than any other thread and is assigned a thread identifier identical to the pod's PVM task identifier in order to allow the interoperation of PVM and TPVM processes. However, the scheme of using only explicitly spawned threads as active computational entities is simpler, provides symmetrical scheduling for all of the threads which will comprise an application, and is thus the typical case.

Executing threads cooperate using a set of message marshaling and communication primitives implemented in the TPVM library. For example, buffer management and data marshaling are supported by `tpvm_initsend()` and `tpvm_pkXXX()` routines that manipulate an implicit, per-thread unique send buffer – these correspond directly to their PVM counterparts. To send a message, the `tpvm_send()` and `tpvm_mcast()` primitives are used. Correspondingly, to receive and unpack TPVM messages, the `tpvm_recv()`, and `tpvm_upkXXX()` library functions are provided. In order to determine information about incoming messages (e.g. size, or the tag and source of messages received when wildcards are specified for the desired tag or source) the `tpvm_bufinfo()` function is also provided. For example, a simple message creation, transmission (on left) and reception (on right) might look like:

```
/* Sender... */                      /* Receiver... */
tpvm_initsend(PvmDataDefault);       Buf = tpvm_recv(MessageTag,-1);
tpvm_pkint(&VecSize,1,1);            tpvm_bufinfo(Buf,&Source,&Size,&Tag);
tpvm_pkdouble(Vector,Vecsize,1);     printf("Message from id %d\n",Source);
tpvm_send(MessageTag,ttids[dest]);   tpvm_upkint(&VecSize,1,1);
                                     tpvm_upkdouble(Vector,VecSize,1);
```

Complete discussion of the details of PVM style message passing and buffer management primitives can be found in the PVM Users' Guide [9] and the TPVM Manual [7].

3.2 Data-driven TPVM Programming

The TPVM data-driven programming model is intended to mask some of of the inherent complexity of the message passing model, and facilitate the construction of composable program elements. While straightforward in principle, explicit message passing based programming models typically force the programmer to explicitly manage all of task creation and synchronization in a program. These tasks can become unmanageable in large systems with hundreds or thousands of concurrent activities. The TPVM data-driven model attempts to automate thread instantiation and message passing in order to address these issues. It essentially requires that threads be declared with named data dependencies; when the data dependencies for a certain type of thread are satisfied, the TPVM run-time system automatically schedules and executes a thread of that type. This process if described in the following discussion.

As introduced above, a `tpvm_export()` call that declares a thread of the data-driven variety also enumerates that thread type's data dependencies. These data dependencies are denoted by integer message tags. The data dependencies associated with a given thread type are satisfied by calls to `tpvm_invoke()`, the analog of send in the TPVM data-driven thread model. For example, in order to make a message available to fill a future thread instance's data dependencies, a thread might execute:

```
tpvm_pkint(Image,ImageSize,1); /* Pack data as usual */

/* Announce the availability  of new messages */
tpvm_invoke(MsgTag,Nfilters,PvmCopyBuf);
```

In this example, we see that the first parameter to `tpvm_invoke()`, like `tpvm_send()`, indicates the message tag of the available message. In contrast to `tpvm_send()`, a `tpvm_invoke()` call specifies no intended destination. The second parameter to `tpvm_invoke()` indicates to how many destinations the given message may be delivered in order to satisfy a data dependency (supporting the data-driven model analog of multicast). In the given example, we presumably have a number of different "filter" thread types which we want to operate on the same image.

The last parameter to `tpvm_invoke()` is a flag which indicates whether the TPVM buffer containing the invoked message may be re-used. If the flag is set to `PvmCopyBuf`, the thread may safely continue using the packed send buffer (at the cost of copying the buffer memory). If the flag is set to `PvmUserBuf` the thread must not use the active send buffer after the invoke call (i.e. `tpvm_initsend()` must be called before a new message is created), but the cost of copying the buffer is avoided.

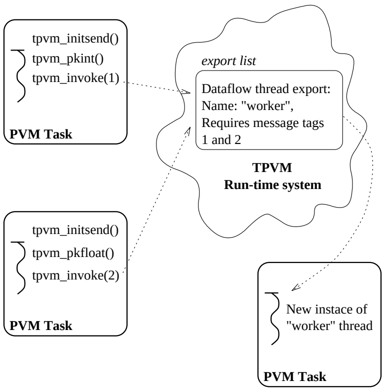

Figure 3: Automatic creation of data-driven threads.

When a subset of the available invoked messages in the system matches the data dependencies of an exported thread, an instance of that thread is created by the TPVM run-time system. This general process is illustrated in figure 3. In the event that all the data dependencies of multiple threads are met, but a decision must be made regarding which will be created, the outcome is non-deterministic. As soon as a TPVM data-driven model thread begins executing, it can assume that its data dependencies have been filled and may thus perform non-blocking receives (`tpvm_nrecv()`) in order to obtain its required inputs. These may be performed in any order and whenever needed. As with message-passing based receives, the TPVM buffer management routines should be used to handle and unpack the incoming messages.

3.3 Remote Memory

An additional TPVM programming abstraction which may be used by both the process-model and data-driven varieties of threads is a facility related to distributed shared memory [13] called "remote memory". Remote memory in TPVM extends the ability of TPVM pods to export services (i.e. available latent threads which may be activated) to apply also to passive entities, i.e. memory regions. This feature is based on system a system primitive to assign a system-wide addressable name to a local memory region: `tpvm_exportmem()`. An example call to `tpvm_exportmem()` is:

```
tpvm_exportmem("Vector",0,Vector,VecSize,PVM_FLOAT);
```

Memory regions exported by `tpvm_exportmem()` are identified by a name and number pair (to support the concept of a logical "global" name and partition number for exported regions) specified in the first two parameters. The third parameter specifies the local memory location to which this name will refer. Finally, the size of the exported memory region must be specified (in terms of number of objects, in this example integers), as well as the data type stored in the named region, in order to allow transfer of the stored data among machines with differing data representations.

Use of exported memory regions is supported via the `tpvm_rcpy()` primitive, which implements an asynchronous copy to or from a named memory export buffer. An example of `tpvm_rcpy()` is:

```
int MyVectorCopy[VecSize];
tpvm_rcpy("Vector",0,MyVectorCopy,PvmMemGet);
```

The `tpvm_rcpy()` call requires a flag as its fourth parameter specifying the desired action: `PvmMemGet` specifying that the data be copied from the named remote memory region to the local memory location given in the third parameter, or `PvmMemPut` specifying that the data at the given local memory location should be copied into the named memory export. This above example would copy the data stored in the exported memory region identified by the pair `("Vector", 0)` into the local memory location `MyVectorCopy`. The `tpvm_rcpy()` primitive is atomic in that the results of one `tpvm_rcpy()` operation are never mingled with the results of another. For example, if two threads performed logically concurrent `tpvm_rcpy()` operations intended to store data into the same named memory export region, the final result would reflect all of one thread's data, and none of the other's (the other's operation would be performed, then overwritten entirely by the second operation serviced).

3.4 Additional TPVM interface features

In addition to the basic programming model interface already described, the TPVM library supports a number of features to enhance the logistical convenience of distributed concurrent programming with TPVM. An example of these additional features is the `tpvm_killtid()` library function. This function is the TPVM analog of `pvm_kill` which is used to force a PVM process to exit. The additional semantics associated with `tpvm_killtid()` are related to the management of thread exports. In order for the TPVM scheduling system to operate efficiently (avoiding a costly time-out/retry protocol), it must have current, accurate information about which thread exports are valid. Thus, before a task exits (e.g using `pvm_exit`), it must use a `tpvm_unexport()` function to invalidate its thread exports. In the event that the process must be abnormally aborted or has relinquished control via `tpvm_go()`, performing this "clean exit" protocol may be quite inconvenient. In these cases, the `tpvm_killtid()` call may be used as it will notify the run-time system that all exports associated with the killed process should be invalidated. Even in situations where a "clean exit" is being coded, it may be more convenient an less error-prone to write ''`tpvm_killtid(pvm_mytid())`" in place of a series of calls to `tpvm_unexport()` followed by a `pvm_exit`.

4 Programming Examples

The utility of programming primitives such as those introduced in the previous section are difficult to evaluate outside of the context of actual programs. Thus, we now examine some brief examples of TPVM programming which will illustrate the three central TPVM features described thus far: the process-based model, the data-driven model, and remote memory.

4.1 A Process Model Stenciler

Stencil algorithms arise as natural solutions in a number of scientific application areas including the numerical solution of partial differential equations and digital image processing. In general, stencil algorithms operate on an array of input data, each point of which is iteratively modified by a function of the local data value and some set of neighboring data values. The resulting communication pattern of parallelized stencil algorithms is thus defined by the set of neighboring values needed to operate on a single data point. For simplicity of exposition, we will consider a stenciler that operates on a two-dimensional array of integers using a North-East-West-South (NEWS) stencil.

```
tpvm_main() {
  tpvm_export("Stenciler",Stenciler,PvmProcModel,0,NULL);
  tpvm_initsend(PvmDataDefault);
  tpvm_send(pvm_parent(),EXPORT_DONE);
  tpvm_go();
}

Stenciler() {
  /* Declare local variables (omitted)... */

  /* Receive initial data */
  tpvm_recv(-1,INITIAL_MESSAGE);
  tpvm_upkint(MyPartition,MyDim*MyDim,1);
  tpvm_upkint(ttids,NTHREADS,1);
  tpvm_upkint(&MyId,1,1);

  for(i=0;i<NUM_ITERS;i++) {
    /* Send and receive N,E,W, and S border regions */
    ExchangeBorders(ttids,MyId);

    /* Perform the stencil operation */
    for(j=0;j<MyDim;j++)
      for(k=0;k<MyDim;k++)
        MyPartition[j][k] = Stencil(MyPartition,i,j,N,E,W,S);
  }
  tpvm_initsend(PvmDataDefault);
  tpvm_pkint(MyPartition,MyDim*MyDim,1);
  tpvm_send(pvm_parent(),STENCIL_DONE);
}
```

Figure 4: TPVM stenciler slave code.

Pseudocode for the pod program which implements and exports the stenciler thread is given in figure 4, while an outline of the master program to start up the algorithm and gather the results is given in figure 5. The code is not significantly different from the code that would correspond to a PVM version. The main cosmetic difference is the need for the

```
tpvm_main() {
  /* Declare local variables (omitted)... */

  InitializeInput(Input);
  pvm_spawn("stencil_pod",0,0,0,NUM_PODS,tids);
  for(i=0;i<NUM_PODS;i++)
    tpvm_recv(tids[i],EXPORT_DONE);
  tpvm_spawn("Stenciler",PvmThreadWild,NULL,NTHREADS,ttids);

  /* Distribute array among workers */
  for(i=0;i<NTHREADS;i++) {
    NextPartition=GetPartition(Input,i);
    tpvm_initsend(PvmDataDefault);
    tpvm_pkint(NextPartition,PartDim*PartDim,1);
    tpvm_pkint(ttids,NTHREADS,1);
    tpvm_send(ttids[i],INITIAL_MESSAGE);
  }

  /* Gather results */
  for(i=0;i<NTHREADS;i++) {
    NextPartition=GetPartition(Input,i);
    tpvm_recv(ttids[i],STENCIL_DONE);
    tpvm_upkint(NextPartition,PartDim*PartDim,1);
  }
  WriteResults();
}
```

Figure 5: TPVM stenciler master code.

worker task to export the available thread for scheduling, then relinquish control. This leads directly to the question of what the advantage of programming this application in TPVM rather than PVM might be. One important answer in this case lies in the automatic ability of TPVM to transparently overlap communication with computation.

Each task in a PVM implementation of this stenciler must block and wait at the beginning of each iteration in order to obtain the border rows needed to apply the stencil function on the outside edge of it's partition of the array. Tasks in the TPVM version of this application are made up of multiple threads of control. A good distribution of the array to the worker threads would assign adjoining regions to threads in the same task, maximizing the number of common borders in each task. Communication of border values among threads in the same pod is performed by the TPVM system through the shared memory space of the interacting threads. Thus, while threads sharing borders with threads in a remote pod will likely need to block and wait at the end of each iteration, "internal" threads will likely have their message dependencies filled relatively quickly and be allowed to continue computing, overlapping computation time with the time other threads spend waiting for messages from remote pods. The result is a decrease in the total time the task as a whole spends blocking for incoming messages, and thus improved performance.

4.2 Data-Driven Pipeline Example

As described in section 3.2, the data-driven TPVM programming model is intended to support automatic instantiation of computational instances and facilitate the construction of composable program elements. Here we examine a pipeline example based on the data-driven model. Pseudocode for a pod exporting a single generic pipeline stage is given in figure 6.

First, it is worth pointing out the basic features of the data-driven programming model in use. Upon export, the default thread of this pod specifies a data dependency rule for the pipe stage it is exporting. Whenever a message tagged with that rule is announced by a `tpvm_invoke()` call, a thread of the type named "`PipeStage`" will automatically be instantiated in some pod running this program. When a thread of the "`PipeStage`" variety is instantiated, it can safely invoke a non-blocking receive primitive (`tpvm_nrecv()`) to satisfy its data dependency. When a `PipeStage` thread has completed processing, it uses the `tpvm_invoke()` primitive to announce the availability of an input for the next stage of the pipeline. A thread that implements this next pipe stage will be instantiated automatically to process the produced input.

This example also illustrates the use of the data-driven programming model to develop composable program components. The pod code depicted here highlights the fact that the data dependency associated with a thread type can be determined at run time. In this example, the pod receives a message from it's parent to determine what the data dependency of it's exported thread will be. As part of one program, the depicted thread may be pipe stage six, while in another it may be stage one. Also note that the thread in this case dynamically computes the dependency of the next pipe stage based on it's own dependency - it simply increments it's own dependency (available via `tpvm_bufinfo()`) and invokes the next stage of the pipe, whatever that may be.

Although not explicitly depicted here, an application may arbitrarily combine the data-driven and process based TPVM programming models. For example, the "`ProcessInput()`" routine called by an instance of the depicted "`PipeStage`" thread could be computationally expensive, in which case it might be parallelized using either the process based or data-driven TPVM programming model. For example, the implementation of this routine might be similar to the master-slave stenciler example described in section 4.1.

4.3 A Remote Memory Bag-of-Tasks

Remote memory in TPVM enables decoupling the location of a data structure in a distributed program from the implementation of threads which export or use that data structure. As an example of programming with remote memory, we consider the design of a generic "bag-of-tasks" to implement dynamic load balancing. The typical scheme with a bag-of-tasks application requires that some process act as a server, producing partitions of a data structure to be operated on (units of work) upon asynchronous demand by processing elements running on machines of various speeds and instantaneous loads. Using the remote memory paradigm we can decouple the implementation of the bag-of-tasks server from the data structures being operated on, thus producing a re-usable program element.

The basic operation of the generic TPVM bag of tasks is to accept a message indicating the number of partitions in a remote memory data structure, and then reply to requests for work with increasing integers up to the number of partitions. Users of these generic bag-of-tasks servers simply request a work partition from the appropriate servers, and then call `tpvm_rcpy()` to obtain the assigned data partitions. For example, a matrix-vector product code might divide the operand matrix into a set of exported remote memory partitions corresponding to groups of rows. A generic bag-of-tasks server could then be used to control access to the matrix operand, while the vector operand could be read initially by all worker threads. Pseudocode for this example is given in figure 7. Note that the worker thread determines the logical remote memory name of its operands dynamically before running its main loop. Partition numbers corresponding to remote memory partitions are obtained from a known generic bag-of-tasks server, and the appropriate data is obtained via `tpvm_rcpy()` and operated on.

5 Implementation Overview

The TPVM system is implemented as a non-intrusive layer over the version 3.3 release of the PVM system. The system has been ported to run on Sun workstations and multiprocessors running SunOS 4.x or Solaris, Silicon Graphics workstations running IRIX 5.x, and IBM RS/6000 workstations running IRIX. The system is comprised of three basic modules: The user interface library module, the portable thread interface

```
tpvm_main() {
      int Rules, Nrules = 1;

      /* Receive message to determine rule of this stage */
      tpvm_nrecv(pvm_parent(),STAGE_CONFIGURE);
      tpvm_upkint(&Rules,1,1);

      /* Export pipe stage and relinquish control */
      tpvm_export("PipeStage",PipeStage,PvmDflowModel,Nrules,&Rules);
      tpvm_initsend(PvmDataDefault);
      tpvm_send(pvm_parent(),EXPORT_DONE);
      tpvm_go();
    }

    PipeStage() {
      /* Declare local variables (omitted)... */

      /* Receive my data dependency */
      InBuf = tpvm_nrecv(-1,-1);

      /* Determine my rule */
      tpvm_bufinfo(InBuf,&InSize,&MyRule,&LastStage);

      /* Next pipe stage uses my rule plus one */
      NextStageRule = MyRule+1;

      /* Unpack and operate on data */
      tpvm_upkfloat(InputData,INPUT_SIZE,1);
      ProcessInput(InputData);

      /* Pass data to next stage of the pipe */
      tpvm_initsend(PvmDataDefault);
      tpvm_pkfloat(InputData,INPUT_SIZE,1);
      tpvm_invoke(NextStageRule,1,PvmUserBuf);
    }
```

Figure 6: TPVM pipeline-stage code.

```
mat_vec_mult() {

  /* Learn remote memory name of operands */
  tpvm_recv(-1,MAT_NAME);
  tpvm_upkstr(mat_name);
  tpvm_upkstr(vec_name);

  /* Get operand vector */
  tpvm_rcpy(vec_name,0,vec,PvmMemGet);

  /* Loop until all work is done... */
  do {
    tpvm_initsend(PvmDataDefault);
    tpvm_send(bag_ttid,REQUEST);
    tpvm_recv(bag_ttid,REPLY);
    tpvm_upkint(&next_part,1,1);

    if(next_part<0) break;

    /* Get next work unit */
    tpvm_rcpy(mat_name,next_part,mat_part,PvmMemGet);

    multiply(mat_part,vec);
  } while (next_part>0);
}
```

Figure 7: TPVM matrix-vector multiply code.

module, and the thread server module. The key design principles behind the prototype implementation were:

- to support efficient, transparent cooperation among both threads in the same address space and threads in different tasks
- to support transparent cooperation among threads running on different architectures and operating system, as well as different underlying thread systems
- to require no modifications to the native PVM release
- to design the system to be easily portable to other architecture, operating system, and thread implementation combinations

Before describing the implementation and interoperation of the basic modules in in detail, it is worth presenting the general features of TPVM run-time operation. First, as has been described, user code interacts with the TPVM system via library calls. These library routines in turn rely on the PVM library for services such as message passing and remote process instantiation. TPVM uses the portable thread interface module for services such as thread creation, process-local thread scheduling, and basic synchronization primitives (e.g. semaphore operations). The TPVM thread server module supports the library implementations of operations such as remote thread instantiation and remote memory moves by providing and export database and global scheduling services. Each PVM process using TPVM facilities contains a dedicated daemon thread of control that receives and enqueues all messages from external pods in the system, and also interacts with the thread server implement activities such as the asynchronous scheduling of threads and remote memory manipulation. The daemon thread and the internal state of the TPVM library on each process, combined with the thread server module, comprise the TPVM run-time system. Figure 8 depicts the basic elements of the run-time system.

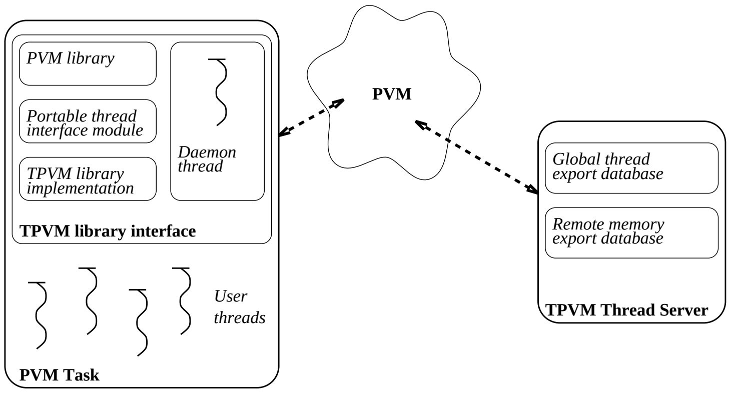

Figure 8: The TPVM run-time system.

5.1 Library Module Implementation

The TPVM library module implements the user interface to the TPVM system and a number of required support services. The library module also contains the implementation of the dedicated "daemon thread" which manages asynchronous activities in a TPVM task. The library functions fall into four main classes: the export functions, the message passing functions, and the instantiation primitives, and remote memory copy.

5.1.1 Managing Exports

The first group of library calls provide the program with the ability to export services. The `tpvm_export()` and `tpvm_unexport()` functions interact with the TPVM thread server module to register or un-register thread declarations with the system. These routines also modify an internal list of valid thread exports for the calling task which is used by the daemon thread to satisfy asynchronous scheduling requests from the TPVM scheduler. Similarly, the `tpvm_exportmem()`, and `tpvm_unexportmem()` communicate the name and location of exported remote memory regions to the thread server module, and add these exports to a local list of valid remote memory regions for the pod.

5.1.2 Buffer Management and Communication

The next group of library routines are responsible for implementing TPVM message passing and buffer handling both within the address space of the local pod, and with threads in remote address spaces. The first task in sending a message is buffer management. Each thread of control has associated with it implicit active send and receive buffers which must remain invariant in the presence of preemption. In order to ensure this invariance, the library maintains a table of active buffer handles associated with each live thread. When a thread uses any of the buffer manipulation routines (e.g. `tpvm_pkXXX()`, `tpvm_initsend()`, etc.), the table is consulted to determine the actual buffer that the call should affect. Once the appropriate buffer has been determined, physical buffer manipulation simply defers to the native PVM implementation.

Once a message is ready for transfer, a TPVM send routine (e.g. `tpvm_send()`, `tpvm_mcast()`) must be used. The send primitive must determine to which PVM tasks to send a message based on a thread identifier. In order to facilitate efficient message source/destination mapping in a distributed fashion, the PVM task id of the the pod containing a thread is encoded into that thread's identifier. If the PVM destination of a message is the local containing task, that message is simply enqueued on the internal incoming message queue, saving the cost of inter-address space communication. If the message is destined for a thread in a remote PVM task, a native PVM message passing function is used to transfer the message to that task.

The `tpvm_recv()` function is responsible for blocking the calling thread (but not the containing process) until some requested message is available, at which time the caller is allowed to proceed with a new active receive buffer containing the received data. The implementation of `tpvm_recv()` uses the following steps:

1. The incoming TPVM message queue is checked, as the requested message may already have arrived at the pod.

2. If a suitable message is not found on the message queue, the calling thread is entered on a "blocked receive list" managed by the TPVM library. Each entry on the blocked receive list contains the identity of a blocked thread, the tag and source of the message desired by that thread (or wildcards, if requested), and a semaphore on which the blocked thread is is waiting.

3. After waiting on the blocked receive list, the thread may immediately dequeue an available message and continue.

Messages are added to a pod's received message queue from two sources: first, threads within the pod send to other threads within the pod by enqueing messages directly, and second, the daemon thread polls for messages from remote pods (using the non-blocking `pvm_nrecv()` function) and enqueues any received. Whenever a message is placed on the received message queue, the blocked receive list is consulted to determine if any threads are waiting for the newly enqueued data. If such a blocked thread is found, the semaphore in its blocked receive record is posted, and the thread will awaken to find the message it requires.

5.1.3 Instantiation Primitives

The third group of library routines implements thread instantiation. The first of these is the explicit `tpvm_spawn()` instantiation function, the operation of which depends on the scheduling discipline requested. If `PvmThreadLocal` is specified, `tpvm_spawn()` simply calls the underlying portable thread interface module to create the requested threads. A list of active thread descriptor data structures is updated to reflect the new live thread instances. If `PvmThreadTask` is specified, a PVM message is sent to the daemon thread of the desired thread host task. On that task, the daemon thread interacts with the portable thread interface module to create the requested threads, and the thread identifiers of the new threads are returned to the requester. If `PvmThreadWild` is specified, the request is forwarded to the TPVM thread server module. The thread server then determines where to schedule the requested threads and forwards the requests to the daemon threads on the appropriate tasks.

The second instantiation related library routine is the `tpvm_invoke()` primitive. It should be noted that the invoke function could also be considered a communication primitive, as it must also manage the delivery of a data-driven thread's activation dependency messages. The invoke routine performs two major actions. First, it updates a local list of all available invoked messages in the pod to record the newly available message for later delivery upon the instantiation of a dependent data-driven thread. Next, the function alerts the thread server to the presence of a newly available invocation, since the new message may satisfy a data-dependency and thus result in the instantiation of a new thread.

5.1.4 Remote Memory Copy

The remote memory copy routine, `tpvm_rcpy()` is responsible for transferring data to or from the owning pod of an exported remote memory region. The first part of this task involves determining the location of a given export based on its logical string name and partition number. In the worst case, the task will need to consult the thread server module to determine the location, as all memory exports are registered with the thread server. In the expected case, the thread will find the location of the memory export in a "memory export location cache" which is managed on a per pod basis. Thus, the first time a pod uses a memory export location, a delay is experienced while the thread server is queried. Whenever a memory export query is made on the thread server, the result is cached and will thereafter be available locally.

After the location of the memory export is determined, a request for services is sent to the daemon thread on the owning pod. If the request is a write (`PvmMemPut`), the request is accompanied with the data to be written. If the request is a read (`PvmMemGet`), a protocol similar to the message receive protocol is employed — the thread enters a "waiting for remote memory read" record on a pod-wide "blocked remote memory read list". The thread then sleeps until the memory read is received by its local daemon thread, which examines the blocked remote memory read list upon each receipt of a message tagged as a remote memory read and awakens the appropriate blocked thread. Of course, in the case where the location of the memory export is determined to be the local pod, the remote memory copy is simply performed as a direct memory copy within the local address space. Synchronization using semaphores ensures that the atomic semantics are preserved in this case.

5.2 The Portable Thread Interface Module

The portable thread interface module is intended to abstract the basic thread services required to implement the TPVM library routines. This level of abstraction allows the implementation of the TPVM library to be decoupled from the available thread services on various OS and architecture platforms. The basic functionality provided by the thread interface module includes the creation of threads, terminating the running thread, yielding the running thread to any other live thread, basic semaphore operations, and determining an identifier number associated with the running thread which is unique within the pod (the pod-wide unique id of the thread, combined with the system wide task id of the pod comprises a thread identifier).

The definition of the "thread" or "lightweight process" abstraction presented by the portable thread interface module is intentionally loosely specified. For example, threads may be preemptively scheduled within a process, or may be scheduled only by explicit user intervention. Scheduling of the threads within the process may be performed by the underlying operating system, or by a user-space library. The threads may operate with true concurrency (i.e. on a multiprocessor), or might only multiplex a single sequential process among one ore more execution contexts. The only fundamental assumptions that the TPVM library module makes of the threads module are that:

- Threads are independent execution contexts which may or may not execute concurrently, and may be switched among, perhaps preemptively, or perhaps via a "`yield()`" operation.
- The threads in a process share the address space of that process.
- The threads in a process can identify themselves using a unique identifier assigned by the threads module; a running thread can determine its own identifier.
- The threads within a process can be synchronized using a standard counting semaphore abstraction. Furthermore, if some thread in the process is not blocked on any semaphore, it will eventually be permitted to execute (to ensure liveness).

It seems likely that almost any implementation of "threads" or "lightweight processes" could satisfy this loose set of requirements. To illustrate this point, we implemented our prototype design implemented over three significantly dissimilar thread systems:

- For use with SunOS 4.x, we implemented the thread interface over the Rex [6] thread system, which operates in user space, is non-preemptive, does not support true concurrent thread execution, and performs scheduling only at the request of the user (via a `yield()` operation).
- We also implemented the thread interface over the Solaris threads library [16], a POSIX compliant package which provides kernel supported, preemptive threads which will operate with true concurrency on a multiprocessor.
- We also ported the thread interface to use the Cthreads package developed at the Georgia Institute of Technology [14]. This thread system is implemented as a user space library without preemptive scheduling, but supports concurrent execution on multiprocessors. It is supported on a number of architectures, of which we utilized the versions running on IBM RS/6000 workstations and Silicon Graphics machines running IRIX 5.x.

These packages represent a fair sampling of the threads implementation spectrum, yet did not present any significant challenges to implementing our portable threads interface module as specified. In most cases, the functions provided by our threads module were implemented as simple transliterations of native thread services.

The result of this loose coupling with the notion of a threads package, combined with the portability of the PVM system, is the somewhat surprising attribute of easy portability of the TPVM system. While it is clear that programs written using TPVM are portable to different TPVM implementations, the portability of the system itself is a matter of concern, as it is with all software developed using lightweight processes. Due to the design of the thread interface module, TPVM avoids the common pitfall of non-portable threads-based software.

Beyond portability, due to support for heterogeneity in the underlying PVM system, all of the TPVM implementations are fully interoperable – cooperating threads need not be concerned with the underlying architecture, operating system, or threads implementation of their peers.

5.3 The Thread Server Module

As was noted in our discussion of the TPVM library module implementation in section 5.1, our prototype TPVM design relies on a centralized thread server task which provides support for the thread and remote memory export database, scheduling, and data-driven thread creation (as well as certain other incidental tasks). While this scheme is neither scalable nor failure resilient, we employed it in our initial design in order to develop a prototype system rapidly and begin conducting initial experiments. Since TPVM library interaction with the thread server is generally infrequent and inexpensive, the thread server does not introduce a bottleneck for small to medium scale applications. An improved design would distribute the functions of the thread server to a group of thread servers which could cooperate to perform similar services. The TPVM library was designed with this inevitable future enhancement in mind to enable straightforward migration to a distributed thread-server scheme.

The primary task of the thread server is the management of the global thread export database and the scheduling of `PvmThreadWild` discipline spawned threads. Thread scheduling is performed in a modified round robin fashion on the appropriate processes, with multiprocessors receiving a greater number of threads in proportion to their number of processors. This scheduling scheme (like the PVM scheduling scheme which is a simple round robin policy) is clearly non-optimal, especially in the presence of heterogeneous computing resources, varying external influences, and irregular computations. However, the focus of this research was not to address the scheduling problem, and the simple scheduling policy described above was adequate for the test applications we ran.

The second task of the thread server is the automatic creation and scheduling of data-driven model threads and the management of an available invocation database. The points in execution at which data-driven instantiation may be called for correspond to the receipt of new invocation database entries at the thread server (which in turn correspond directly to `tpvm_invoke()` calls. Thus, whenever a new invocation message arrives, the thread server updates the invocation database, then examines each thread type in the thread export database which lists the new invocation as a data dependency. Whenever a thread is found which is dependent on the newly arrived message tag, the other dependencies of the thread are checked versus the invocation database. If a thread type is found which may be scheduled, the thread server follows the same scheduling procedure as with spawned threads. In addition to this, the thread server notifies the daemon threads of the invocation owning processes to send the invoked messages to the newly scheduled thread. When a scheduling request for a new data-driven thread and all of the new threads data dependencies have arrived, the daemon thread on the selected process can instantiate the new thread using the underlying thread system.

In addition to these basic tasks, the thread server also maintains the global database of remote memory exports. As was described in section 5.1.4, this function of the thread server module is expected to be used infrequently due to remote memory location caching within the TPVM library module in individual tasks.

6 Empirical Evaluation

6.1 Basic TPVM Costs

We ran a number of experiments to determine the costs associated with some basic TPVM actions. SunOS 4 based tests were run on a set of SparcStation-1+ workstations with 18MB of RAM, and running SunOS 4.1.2. These tests were compiled with version 2.6.1 of `gcc` with the `-O2` optimization flag. Solaris based tests were run on a set of SparcStation-10 workstation with 20 MB of RAM running Solaris 2.3 (SunOS 5.4), except where noted. These tests were compiled with `gcc` version 2.6.3 with the `-O2` optimization flag. In both cases the hosts used were all on the same thin Ethernet subnet, and the machines and network were generally under very light loads when the tests were run.

The first test was performed to determine the relative costs of PVM task creation, remote TPVM thread creation (`PvmThreadWild`), and local TPVM thread creation. Task creation was timed using a PVM net consisting of eight nodes. Thread creation was tested using eight PVM tasks on different the same PVM net. Elapsed time between the call of the appropriate spawn routine and its completion (the return of the task identifier set) is recorded. The results are given in table 2

Number Spawned	SunOS			Solaris		
	PVM Tasks	Remote Threads	Local Threads	PVM Tasks	Remote Threads	Local Threads
1	110.0	35.6	2.9	56.7	14.8	1.4
8	798.5	90.4	12.7	206.5	45.3	5.8
16	2826.1	330.3	83.5	874.2	153.3	22.5

Table 2: Task/Thread creation costs (times in milliseconds).

These tests indicate that the creation of concurrent processing elements on a fixed number of hosts is significantly less expensive when light-weight processes are used. The creation of local threads is relatively negligible (compared to PVM tasks), while the cost of creating remote threads adds only the cost of two small messages and the time to make a scheduling decision in the thread server. In contrast, the creation of a PVM task requires an equal amount of message passing and scheduling time, but also adds the heavyweight process creation overhead and introduces significantly larger memory usage overheads on the target host. It should be noted that the cost of creating a TPVM light weight process assumes that the cost of creating a heavyweight pod has already been paid. The cost of creating a certain number of heavyweight processes (most likely one per host) is unavoidable under TPVM. However, for applications with large numbers of logically concurrent activities, the cost of creating the needed processing elements will on average be much lower under TPVM, as is supported by our experiments.

Our next experiment measured the cost of communication, timing the round-trip delivery of messages of various sizes. We measured these communications costs using PVM tasks running without TPVM active, then between two remote TPVM threads, and finally between two local TPVM threads. All versions of the test used the `PvmDataInPlace` buffer option without the direct routing (TCP) option enabled. The results are given in table 3.

As expected the cost of message passing among local threads is significantly lower than that among remote threads. The difference in these costs is generally an order of magnitude, as would be expected since communication between local threads occurs via shared memory. It should be noted that the cost of communication among local threads is not as low as theoretically possible, given that message passing semantics require that the transmitted data be copied from the source threads buffer into the destination threads buffer. If explicit use of shared memory were supported, the threads could simply

Message Size	SunOS			Solaris		
	PVM Tasks	Remote Threads	Local Threads	PVM Tasks	Remote Threads	Local Threads
1 B	11.5	14.9	1.5	5.6	8.5	0.9
1 KB	16.8	19.8	2.5	9.5	17.9	1.1
32 KB	203.3	214.4	18.11	126.3	132.0	17.1
1 MB	5754.9	5967.9	847.8	3766.7	4061.5	252.3

Table 3: Round-trip message costs (times in milliseconds).

access the same memory region, likely incurring much lower overheads (depending on the memory subsystem design). In TPVM, the potential performance penalty of disallowing explicit shared memory cooperation is accepted as a trade off for orthogonal, location independent cooperation primitives which lead to a cleaner programming model.

6.2 Matrix Multiply Application

The most basic evaluation of any parallel programming system lies in its performance on actual codes. As a first example application for performance evaluation, we implemented the matrix multiplication algorithm described in [10]. This algorithm employs square subblock decomposition of the operand and result matrices, where each processing entity is initially is distributed one block of each operand and computes a single result subblock. Communication is regular and occurs in a mesh pattern. The data partitioning scheme employed by this algorithm requires that an even square number of processing entities be used. Furthermore, the algorithm is not optimized with a send-ahead scheme, and thus could benefit from the overlap of computation with communication.

To examine the potential benefits of communication/computation overlap, we ran the TPVM implementation of this matrix multiplication algorithm, along with a native PVM implementation of the same algorithm on identical 4 node PVM nets (using the machine configurations, compiler options, etc. described above), as well as a sequential benchmark case. The data-parallel versions and the sequential control case all utilized the same matrix multiplication inner loop. The results for matrix multiplication of 512x512 operands multiplied on a SunOS 4 based net are given in table 4, while the results for the same problem size on a 4 node Solaris based net are given in table 5. As expected, the TPVM version achieves better processor utilization. The best results are observed when a balanced number of threads are assigned to each process. In the Solaris version, a speedup of 12% over the best PVM version is observed, while the SunOS version speeds up by 19% the PVM implementation. We noticed in this experiment that the Solaris implementation introduced somewhat higher overheads at lower numbers of threads than did the SunOS version. This is due in part to the polling action of the daemon thread. There are a number of alternative strategies to avoid this polling such as a user thread polling policy or interrupt driven message handling that are being considered for future versions of the system.

	Sequential	PVM	TPVM 4 Threads	TPVM 9 Threads	TPVM 16 Threads
Time (msecs.)	296.2	100.1	100.7	114.6	84.1
Speedup	-	2.95	2.94	2.58	3.52

Table 4: 512x512 Matrix Multiply on 4 SunOS 4.1 nodes (times in seconds).

	Sequential	PVM	TPVM 4 Threads	TPVM 16 Threads	TPVM 36 Threads
Time (msecs.)	118.4	37.9	49.4	38.3	33.7
Speedup	-	3.12	2.39	3.07	3.51

Table 5: 512x512 Matrix Multiply on 4 Solaris nodes (times in seconds).

As described above, the implemented matrix multiplication algorithm requires an even square number of processing elements. Given a number of available hosts which is not an even square, this algorithm would probably not achieve good load balance. The cost of using many additional tasks with smaller data partitions would quickly outweigh the performance gained by better load balancing. Intuitively, it seems likely that programming with light weight processes could address this issue. We verified this empirically by running the PVM and TPVM versions of the application presented in the last experiment, but on a 5 node SunOS based network. The results are given in table 6. The results of this experiment indicate that load balancing in situations such as this can be accomplished without additional programming effort using TPVM. The use of threads significantly reduced the overheads associated with finer-grained decompositions, allowing processor load assignments to be more finely balanced. It might be argued that a better algorithm ought to have been used in this case to achieve better load balance. While this is true with respect to our simple matrix multiplication example, in less trivial cases algorithmic refinement may be inappropriate or poorly suited to fully address the load balancing issue. In these situations, TPVM offers a low-effort, low-overhead alternative.

Processing Elements	PVM	TPVM
4	100.2	100.8
9	80.1	78.4
16	89.0	83.3
25	79.7	68.0
36	89.4	75.8

Table 6: 512x512 Matrix Multiply on 5 SunOS nodes (times in seconds).

Our next experiment using this matrix multiplication application was intended to test the efficiency of our prototype implementation on a small scale shared memory multiprocessor. Machines with a small number (2-8) of processors and global shared memory are becoming increasingly available resources in heterogeneous networked environments. We ran various problem sizes on a 6 CPU SparcCenter-2000 server with 240 MB of memory running solaris 2.3. The results, shown in table 7, are encouraging. As would be expected, programming using a distributed memory, message passing paradigm can achieve respectable performance on a shared memory multiprocessor, with speedups ranging from nearly 2 in the finest grained case, to 5.6 in the largest problem size.

Processing Elements	64x64	128x128	256x256	512x512
Sequential	.25	2.2	18.4	171.3
4	.13	1.6	5.8	42.2
9	.16	1.8	5.5	31.9
16	.17	2.1	4.9	30.7

Table 7: Matrix Multiply on a 6 CPU MP nodes (times in seconds).

As a final experiment with this application, we ran the code on a more realistic, heterogeneous network consisting of the 6 processor SparcCenter used in the last experiment, a four processor SparcStation-20 with 28 MB of memory, and 3 of the SparcStation-10 workstations used in the first experiment. Results for two problem sizes are presented in table 8. The TPVM implementation in this case is 67% faster on the smaller problem size, and 54% faster on the larger problem size. These performance gains are due to the combined affects of computation / communication overlap (latency tolerance), the reduced costs of spawning more processing elements, and the improved TPVM scheduling heuristic which placed more work on the multiprocessors than did the PVM scheduler. As expected, the PVM version performs better on the courser grained problem size, but is still outperformed by the TPVM implementation.

Processing Elements	PVM 512x512	TPVM 512x512	PVM 1024x1024	TPVM 1024x1024
4	39.3	36.1	358.3	363.4
9	33.0	21.2	331.7	181.2
16	41.1	18.8	336.2	173.1
25	29.8	17.8	205.5	132.9
36	31.2	17.9	216.0	144.7

Table 8: Matrix Multiply on a heterogeneous network (times in seconds).

6.3 Multigrid Solver Application

As a second test application, we implemented the Multigrid Solver Kernel described as described in [1]. The application executes four iterations of the V-cycle multigrid algorithm to obtain an approximate solution to the discrete Poisson problem $\nabla^2 u = v$ on a three dimensional grid with periodic boundary conditions. Our implementation of this program was based on porting the PVM implementation described in [19], where the grid is partitioned among processing elements in one dimension, and processing elements communicate in a ring topology in order to keep their border regions up to date. This application involves substantial communication between neighboring processors, as they must repeatedly communicate the values of vectors at border regions as they change. It would be expected that this program could benefit significantly from the overlap of computation and communication transparently afforded by TPVM.

We ran this application on a dedicated Ethernet of 8 SparcStation-1+ workstations along with the unmodified PVM version described in [19]. The results for various problem sizes are presented in table 9. Once again the benefits of multithreading are evident. While overheads result in slightly reduced speedup in the finest grained problem size, an improvement of 6% is observed in the middle problem size, and a 21% speedup is achieved in the largest problem size.

Problem Size	Sequential	PVM 8 Tasks	TPVM 8 Threads	TPVM 16 Threads	TPVM 32 Threads
32x32x32	7.4	6.9	7.2	7.0	-
64x64x64	58.7	23.4	24.0	22.0	23.8
128x128x128	-	115.6	119.7	95.4	120.7

Table 9: Multigrid Solver on an 8 node SunOS network (times in seconds).

7 Related Research

The concept of employing lightweight processes in a distributed memory parallel computing environment is not unique to TPVM. An example of a similar project is Chant [12]. This package combines the POSIX pthreads interface with the MPI communications interface, supporting both of these interfaces and allowing them to interoperate. While essentially similar in it's components to the TPVM process based model, this scheme differs in that the use of thread based primitives is supported, allowing a mixture of traditional shared memory and distributed memory concurrent programming. The ability of Chant to support *explicit* shared memory cooperation among threads could lead to better performance in many cases. The tradeoff in such cases would be in additional programming complexity.

Another example of threads as the basis of a distributed memory parallel programming system is the ATHAPASCAN-0 module of the APACHE project, which builds a non-blocking RPC-based language on top of PVM. This system utilizes threads, which are assumed to be non-preemptive, to implement the computation associated with each RPC, while employing PVM for transfer of parameters. This system differs from TPVM in its "control parallel" programming model, and also lack of support for concurrent thread execution on multiprocessors. As the authors mention in [4], an asynchronous RPC programming model such as this might be implemented over TPVM.

It is worth mentioning another effort to introduce multithreading into the PVM system: the LPVM project at Oak Ridge National Laboratory [20]. This project differs significantly from the TPVM effort, in that it is aimed at providing an efficient, threads-based implementation of the native PVM library for shared memory multiprocessors. In the LPVM scheme, an entire PVM application operates within a single address space, with tasks being realized as threads which may execute in parallel. While both TPVM and LPVM are based on the well-known fact that a message passing abstraction can easily and efficiently be implemented by processes sharing an address space, LPVM does not attempt to combine this concept with the domain of distributed memory parallel computing. While it is clearly valuable to increase the support for heterogeneity in toolkits such as PVM by incorporating support for efficient execution in shared memory environments, it is part of the TPVM philosophy that this goal should not hinder the ultimate goal of facilitating the interoperation of distributed heterogeneous resources.

8 Conclusions

Our prototype design and implementation of TPVM, a multi-paradigm parallel programming system, has proven a solid starting point from which to begin exploring the utility of threads in distributed memory parallel processing. TPVM provides support for multiple composable programming abstractions all sharing an underlying model of lightweight processes as the units of scheduling and parallelism. Using the "default" TPVM programming model, programs can be developed which are only cosmetically different from their PVM counterparts. This opens up the possibility of availing of threads performance benefits without significant programming investment.

We have designed the system to be highly portable, and have had encouraging experiences porting the system to new platforms. The system makes very weak assumptions about the underlying threads subsystem, and can be configured to operate using most thread implementations. Furthermore, the system requires no changes to the underlying PVM system to operate correctly.

Perhaps the most encouraging results obtained with this prototype system are application performance improvements over unenhanced PVM. Even on homogeneous networks of uniprocessors, PVM's traditional strong point, TPVM transparently provides speedups of 5-20%. In heterogeneous environments the performance gains are even more significant. It should be noted that these performance gains have been achieved using the process based programming model. In future work, we hope to examine the performance benefits of data-driven threads, which provide an elegant programming abstraction, and can be used to easily implement latency tolerance schemes such as active messages.

Other future work on the system will be directed at improving the implementation, most notably the thread server module, the buffer handling implementation in the library, and the daemon polling scheme for receiving messages from remote tasks. Furthermore, additional functionality to support load balancing through automatic thread migration seems attractive in the TPVM environment, especially since the state of individual threads will typically be smaller than that of full tasks. This will, we believe, afford a load balancing mechanism more fine-grained control of processor loads, and reduce the cost of migration. We are investigating compiler based schemes for supporting the automatic migration of threads in heterogeneous environments.

References

[1] D. Bailey, E. Barszcz, et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–67, 1991.

[2] R.M. Butler and E.L. Lusk. Monitors, Messages, and Clusters: The P4 Parallel Programming System. *Parallel Computing*, 20:417–444, 1994.

[3] Doreen Y. Cheng. A Survey of Parallel Programming Tools. Technical Report RND-91-005, NASA Ames Research Center, Moffett Field, CA, May 1991.

[4] Michel Christaller and Martha-Rosa Castaneda Retiz. Control Parallelism on top of PVM. In *Proceedings of the Second European PVM User's Group Meeting*, 1995.

[5] *Workshop on Cluster Computing, Florida State University.* Tallahassee, FL, 1992,1993 `ftp://ftp.scri.fsu.edu/pub/cluster-workshop/`.

[6] S. Crane. The Rex Lightweight Process Library. Technical report, Imperial College of Science and Technology, London, England, 1993.

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

[8] A.J. Ferrari and V.S. Sunderam. TPVM: Distributed Concurrent Computing with Lightweight Processes. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, Washington, D.C., August 1995. IEEE.

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM (Parallel Virtual Machine): A Users' Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[10] Geoffrey C. Fox et al. *Solving Problems On Concurrent Processors.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[12] Matthew Haines, David Cronk, and Mehrotra Piyush. On the Design of Chant: A Talking Threads Package. In *Proceedings of Supercomputing 94*, pages 350–359, Washington, DC, November 1994.

[13] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4), November 1989.

[14] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991.

[15] *Proceedings of the 1995 PVM Users' Group Meeting.* Pittsburgh, PA, May 1995.

[16] Sun Microsystems, Mountain View, CA. *SunOS 5.3 Reference Manual*, 1993.

[17] Louis H. Turcotte. A Survey of Software Environments for Exploiting Networked Computing Resources. Technical report, Engineering Research Center for Computational Field Simulation, Mississippi State, MS, June 1993.

[18] Thorsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, May 1992.

[19] S. White, A. Alund, and V.S. Sunderam. Performance of the NAS Parallel Benchmarks on PVM Based Networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, April 1995.

[20] Honbo Zhou and Al Geist. LPVM: A Step Towards Multithreaded PVM. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, 1995.