(NASA-CR-199393) HETEROGENEOUS CONCURRENT COMPUTING WITH EXPORTABLE SERVICES (Emory Univ.) 5 p

N96-10977

Unclas

G3/61 0066392

Heterogeneous Concurrent Computing with Exportable Services*

Vaidy Sunderam†

Abstract

Heterogeneous concurrent computing, based on the traditional process-oriented model, is approaching its functionality and performance limits. An alternative paradigm, based on the concept of services, supporting data driven computation, and built on a lightweight process infrastructure, is proposed to enhance the functional capabilities and the operational efficiency of heterogeneous network-based concurrent computing. TPVM is an experimental prototype system supporting exportable services, thread-based computation, and remote memory operations that is built as an extension of and an enhancement to the PVM concurrent computing system. TPVM offers a significantly different computing paradigm for network-based computing, while maintaining a close resemblance to the conventional PVM model in the interest of compatibility and ease of transition. Preliminary experiences have demonstrated that the TPVM framework presents a natural yet powerful concurrent programming interface, while being capable of delivering performance improvements of upto thirty percent.

1 Introduction

Parallel and concurrent processing on heterogeneous collections of networked computing systems has received widespread attention and adoption recently. The standard scenario, in the use of networked machines for parallel 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. For many classes of applications, loosely coupled concurrent computing in such environments has proven to be straightforward in terms of parallelization effort, at least satisfactory if not very good in terms of performance, and highly effective in terms of cost. Numerous software systems are in existence [1] although the majority of use is probably based on a few popular ones, such as PVM [4], P4/Parmacs [2], and Express [3] that are used extensively — it is estimated that nearly 10,000 sites or users worldwide have obtained one of the above three software systems, and a fair percentage actively use them.

However, despite their widespread adoption and abundance of success stories [5, 6], network-based concurrent computing systems suffer from several critical shortcomings. Principal among them are factors concerning, or in some way related to, performance – which in turn is almost completely dominated by communication speeds and capacities. It

*Research supported by NASA grant NAG 2-828, ONR N00014-93-1-0278, and NSF CCR-9118787.
†Dept. of Math & Computer Science Emory University, Atlanta, GA 30322

is a well known fact that commonly available networks upon which heterogeneous computing systems typically operate are relatively slow thereby precluding high efficiency in parallel applications requiring large volume and/or frequent communication. 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. We propose an alternative paradigm, based on light-weight processes and event driven computation, to attempt to address some of the above issues in network-based concurrent computing 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.

2 Background

Heterogeneous concurrent computing, as a technology and an evolving sub-discipline in parallel processing, is aimed at emulating general, flexible, and effective computing environments on networked collections of multifaceted machines. Software frameworks are the mainstay of this computing methodology, which is built on the premise that "the network *is* the high-performance computer", i.e. that independent but interconnected computer systems can be a viable alternative to, if not better than, monolithic hardware multiprocessors and vector supercomputers. Another fundamental theme is that real applications are comprised of several sub-algorithms, each with potentially different requirements and amenable to different concurrency paradigms — and that heterogeneous systems therefore enable execution of individual sub-algorithms on the best suited machines, to the maximal possible extent.

While the above (seemingly logical) issues continue to be long-term research goals, the practice of network-based computing has generally been confined to clusters or networks of workstations and sometimes general purpose parallel machines. Heterogeneity is often present but is addressed only in terms of instruction sets and data representations; most applications are SPMD programs that are partitioned in a manner identical to homogeneous MPP's, ignoring such differences as machine speeds, specific architectural and machine characteristics (pipelining, optimized scientific libraries, varying external loads etc), and variations in network capacities. The primary reason for the tendency to blindly port programs or algorithms designed for homogeneous MPP's is complexity — it is very difficult to accurately parametrize computing resources in a network-based concurrent computing system especially when it is subject to time-varying external influences. An algorithm or application optimized for specific clusters or heterogeneous environments might overcome

this obstacle with significant effort, but would then lose the portability advantage.

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/experimental arenas. Such a judgement is not necessarily derogatory; network-based systems are generally accepted as having 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 NASA Parallel benchmarks [7]. This table shows excerpted performance numbers from a detailed benchmarking effort [8] 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	Multigrid Method	Integer Sort	Fourier Transform
Cray Y-MP/1	22 secs	11 secs	29 secs
i860/128	9 secs	14 secs	10 secs
PVM Comm. Vol.	192 MB	560 MB	1500 MB
No. of msgs.	1808	2491	826
8 RS6K/FDDI	110 secs	318 secs	412 secs
(idle secs)	(30)	(129)	(34)
8 SGI/Gswitch	168 secs	258 secs	228 secs
(idle secs)	(50)	(105)	(34)

TABLE 1

Representative NPB results on PVM

Several points are worthy of note in the table. One is that realistic algorithms with substantial communication content may be executed on small clusters using PVM about an order of magnitude slower than on a single processor vector supercomputer or on an MPP with 128 processors. More relevant to this discussion however, is the fact that there is often the potential for further improvement by upto 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.

In this paper, we describe an 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, with little or no additional effort required of the programmer
- permit and achieve overlap of communication and computation when possible, in a manner transparent to the user
- allow application parallelization at smaller, perhaps more natural, granularity without sacrificing efficiency
- enable dynamic scheduling of computational tasks in a dynamically load-balanced manner
- provide for the exploitation of adaptive parallelism, i.e. changes in the resources available within a network-based computing environment

3 The TPVM Framework

TPVM is a collection of extensions and enhancements to the PVM computing model and system. In TPVM, computational entities are threads. TPVM threads are runtime manifestations of imperative subroutines or collections thereof that cooperate via a few simple extensions to the PVM programming interface. In the interest of straightforward transition and to avoid a large paradigm shift, one of the modes of use in TPVM is identical to the process-based model in PVM, except that threads are the computational units. TPVM also offers two other programming models — one based on data driven execution, and the other supporting remote memory access. The individual models supported by TPVM are discussed in later sections; a general architectural overview of TPVM is depicted in Figure 1, and a brief description follows.

Architecturally, TPVM is a natural extension of the PVM model. In terms of the resource platform, TPVM also emulates a general-purpose heterogeneous concurrent computer on an interconnected collection of independent machines. However, since TPVM supports a threads-based model, it is potentially capable of exploiting the benefits of multithreaded operating systems as well as small-scale SMMP's — both of which are becoming increasingly prevalent in general purpose computing environments. Further, multiple computational entities may now be manifested within a single process. In combination, these aspects enable increased potential for optimizing interaction between computational units in a user-transparent manner. In other words, inter-machine communication may continue to use message passing, while intra-machine communication, including that within SMMP's, may be implemented using the available global address space. In addition, "latent" computational entities in the form of dormant threads may be instantiated either asynchronously or during initialization, at negligible or low cost. In many cases, this helps reduce the overhead of spawning new computational entities during application execution. Further, the concept of latent threads extends naturally to service-based computing paradigms that are more appropriate in general purpose, non-scientific, distributed and concurrent processing.

TPVM is designed to be layered over the PVM system, and does not require any modifications or changes to PVM. User level primitives are supplied as a library against which application programs link; operational mechanisms are provided in the form of

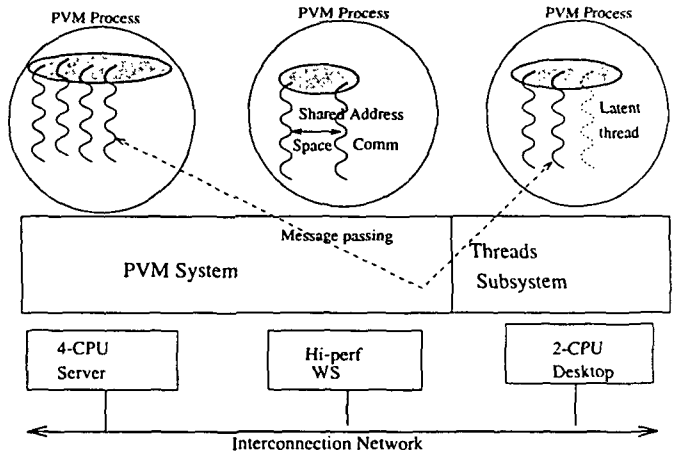

FIG. 1. *TPVM Architectural Overview*

standalone PVM programs. Central to the TPVM implementation is the concept of a scheduling interface that is responsible for controlling thread spawning as well as other facilities that are available in TPVM. Thus interface is defined in functional terms, thereby enabling implementations to evolve from a centralized mechanism in preliminary implementations to one based on distributed algorithms. A schematic of this the scheduling interface and the principal facets of a TPVM implementation are shown in Figure 2.

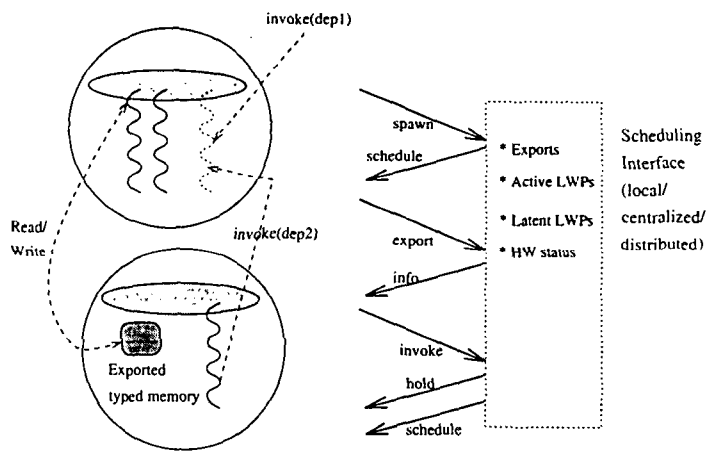

FIG. 2. *Schematic of TPVM Implementation*

A thread in TPVM is essentially a subroutine/procedure (or a code segment, including nested procedure invocations, identified by one entry point). TPVM requires "strong encapsulation" of threads, i.e. shared variables are not permitted, although the system

cannot enforce this restriction. Thus, a TPVM thread is a (sequence or collection of) subroutine(s) that, when initiated, possesses a thread of control and executes on its own stack, with its own data segment. Threads in TPVM exist within the context of PVM processes. However, processes in TPVM do not indulge in computation and/or communication; they only serve as "shells" or environments for threads to exist in, and play active roles only at certain points during execution e.g. for thread creation. A process may play host to (be the "pod" of) one or more threads, each of which may be an instantiation of the same or of different entry points. Application programmers access facilities in TPVM by invoking model-dependent functions that are described below.

4 The Process Oriented Model

TPVM supports a "traditional" concurrent computing model based on multiple interacting threads cooperating via explicit message passing. In this model, multiple threads, each with its unique thread id, are spawned and subsequently exchange messages using `send` and `receive` calls in a manner analogous to process spawning and interaction in PVM. However, to define threads that may be spawned, PVM pod (host) processes utilize the function

```
int
tpvm_export(name, func, limit, flags, 0, 0)
char *name;
void (*func)();
int limit, flags;
```

This function declares a potential thread identified by a symbolic string valued name, and associated with a function entry point; additional parameters specify options, and limits to the number of threads allowed within this pod. An already existing thread or a pod process may subsequently spawn one or more instantiations of an exported thread, using the function

```
int
tpvm_spawn(name, flag, where, nthread, ttids)
char *name, *where;
int flag, nthread, *ttids;
```

specifying the symbolic name of the service, the number desired, and optional parameters indicating spawning locations and scheduling strategies. The thread ids of all spawned threads are returned, which may then be distributed to all threads if necessary. Subsequently, these threads exchange messages using the calls

```
tpvm_send(ttid, msgtag)
int ttid, msgtag;

tpvm_mcast(ttids, nthread, msgtag)
int *ttids, nthread, msgtag;

tpvm_recv(ttid, msgtag)
int ttid, msgtag;

tpvm_nrecv(ttid, msgtag)
```

```
int ttid, msgtag;

tpvm_probe(ttid, msgtag)
int ttid, msgtag;
```

As apparent, the above calls are direct equivalents of their PVM counterparts, with thread ids substituted for task ids; other functions, such as `pvm_initsend` and `pvm_pkxxx` are inherited as is from PVM. Our experiences with a preliminary implementation of this process-oriented model for TPVM have been encouraging, both from the viewpoints of functionality as well as performance. Two textbook applications, namely matrix multiplication and sorting, were written to conform to this model and tested on our experimental implementation; results are shown in Table 2. As can be seen from this table, the TPVM versions perform better, by a factor of upto 24% — with the largest gains occurring when the granularity and the number of threads are "ideal". As some entries show, TPVM performance is worse when the overheads of thread management offset any gains due to the increased asynchrony.

Problem (size)	PVM: CPUs/Processes 4/4	16/16	TPVM: CPUs/Threads 4/16	16/64	16/36
Matmul (500x500)	201	78	182	94	60
Matmul (1000x1000)	1618	756	1610	710	663
Sort (2M integers)	1423	366	1201	351	309

TABLE 2

PVM vs. TPVM times in seconds (SS1+ workstations on Ethernet)

5 The Data Driven Model

Motivated by the well known advantages of data driven computing, as well as by observations that scheduling is of critical importance for high performance, the TPVM system also supports a computing model that is different from the process-based paradigm. In this scheme, thread entry points are exported as before, but contain a list of "firing rules" that are required to be satisfied before a thread can be instantiated. The last two arguments of `tpvm_export` contain the size of, and a pointer to, an array containing a list of message tags — implying that the specified thread may be instantiated when one message of each tag type is available. Typically, as threads complete some portion of their allocated work, they are able to (partially) satisfy such a dependency. In order to indicate that a thread is able to satisfy a firing rule of an exported service, and to deliver a message containing the required data, the following function is used:

```
int
tpvm_invoke(name, msgtag, number, flags)
char *name;
int msgtag, number, flags;
```

As an example, consider a PVM process called `P1` that exports the symbol `''matmul''`, specifying that a thread manifesting a matrix multiply service may be instantiated when messages of tag `77` and `88` (e.g. representing corresponding subblocks of the original

matrices) are available. When threads `Ta` and `Tb`, in the course of their computations, generate the appropriate subblocks required by `matmul`, `Ta`, for example, executes:

```
pvm_initsend(0);
pvm_pkdouble(Asubblock, 100*100, 1);
tpvm_invoke("matmul", 77, 1, 0);
```

with `Tb` executing a corresponding sequence of functions. When both `tpvm_invokes` have been executed, the TPVM system instantiates the `matmul` thread at the most appropriate location, and delivers the two messages to it. Exported threads based on this model would typically begin execution by calling the appropriate `tpvm_recv` functions, followed by the computational code corresponding to the service they provide. The potentially significant benefits of this model are to delay thread spawning until all dependencies are satisfied, thereby enabling load-balanced scheduling at low cost. Table 3 indicates the measured performance for the matrix multiplication and sorting examples mentioned earlier; results from the process-based model are also included for convenient comparison. As can be seen, the data driven model performs at approximately the same levels in some cases, and significantly better in others.

Problem (size)	ProcModel: CPUs/Threads 4/16	16/64	16/36	DflowModel: CPUs/Threads 4/?	16/?
Matmul (500x500)	182	94	60	194	62
Matmul (1000x1000)	1610	710	663	1582	640
Sort (2M integers)	1201	351	309	1167	309

TABLE 3

TPVM ProcModel vs. DflowModel times in seconds (SS1+ workstations on Ethernet)

6 Remote Memory Access

In addition to the two message-based computing models described above, TPVM also supports a third, which is based on the notion of logical access to remote address spaces. This facility, referred to as remote memory copy, is a compromise between the need for explicit participation by both entities involved as in traditional message passing and completely transparent access as in (distributed) shared memory. Essentially, the idea is that computational entities export data areas which may then be read from or written to, asynchronously, by other entities thereby requiring only the proactive participant to be explicitly involved in the interaction.

In TPVM, threads or processes may indicate that certain local data areas are available for remote access, with strongly typed semantics that are required when operating in a heterogeneous environment, by using the

```
tpvm_exportd(name, partno, address, type, size, flags);
char *name;
void *address;
int piecenum, type, size, flags;
```

primitive, that associates a (portion of a) local data structure identified by an address, a datatype, and a length, with a symbolic identifier composed of a string valued logical

name and an integer qualifier. This naming scheme is useful, for instance, when portions of a distributed data structure reside within different thread address spaces. External threads may place or retrieve typed data into or from such exported data areas by using the `tpvm_read` and `tpvm_write` constructs. Options, settable using the `flags` parameter, may be used to ensure synchronized access when needed.

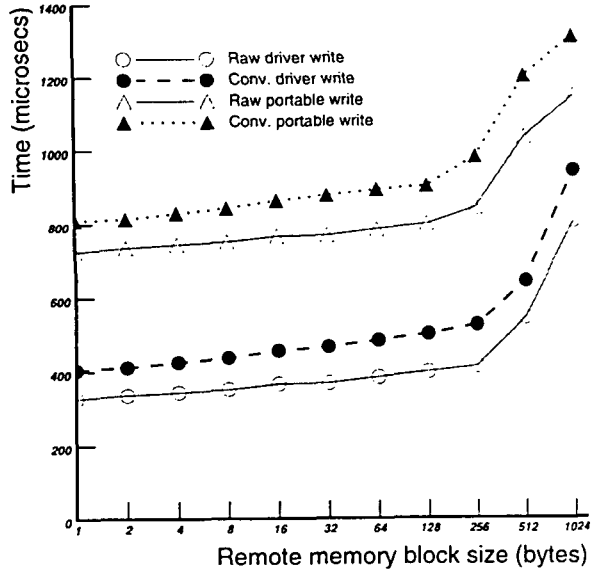

FIG. 3. *TPVM Remote Memory Copy Performance (SS1+ workstations on Ethernet)*

The preliminary TPVM implementation supports a prototype version of the remote memory access facility that was used to ensure that the overheads in emulating a translucent shared address space are acceptable. Figure 3 indicates the time required to perform remote memory writes for different sizes of exported data areas. Two sets of times are reported; one based on the generic PVM implementation using high-level Internet protocols, and the other based on a research version of PVM modified to use low level Ethernet protocols by directly accessing the data link device drivers. Also shown are the costs incurred in data format conversion that is necessary to support heterogeneous remote memory copy.

7 Discussion

In this paper, we have described an extension to the PVM heterogeneous concurrent computing system that supports three distinct and novel paradigms within the same unified framework, while retaining almost complete compatibility with the existing PVM model. The key concept in these extensions is the use of multiple threads or lightweight processes within each PVM process, thereby increasing the available and exploitable asynchrony, and at the same time, providing for alternative computational models that can

be implemented in a message-passing environment. From the program development point of view, the process-oriented model in TPVM is closest to the existing, and traditional, program structuring scheme. Translating existing message passing programs to this model is straightforward, and can potentially result in performance gains of the order of 30%. The other two schemes supported by TPVM, i.e. the data driven model and facilitiies for remote memory copy are likely to be of value for applications where expressing parallelism and dependencies does not naturally fit the explicit message passing paradigm. Further, our experiences show that these facilities for computational entities to interact in TPVM can also benefit from increased asynchrony, computation/communication overlap, and dynamic load balancing, and are capable of delivering significantly improved performance over comparable implementations based on traditional message passing. While the success and value of the TPVM system can only be determined after substantial evolution and use, early experiences have been encouraging and indicate that these paradigms enhance the viability and usability of heterogeneous, network-based, concurrent computing.

References

[1] Louis Turcotte, "A Survey of Software Environments for Exploiting Networked Computing Resources", Draft Report, Engineering Research Center for Computational Field Simulations, Mississippi State, January 1993.

[2] R. Butler and E. Lusk, "User's Guide to the P4 Programming System", Argonne National Laboratory, Technical Report ANL-92/17, 1992.

[3] A. Kolawa, "The Express Programming Environment", *Workshop on Heterogeneous Network-Based Concurrent Computing*, Tallahassee, October 1991.

[4] V. S. Sunderam, G. A. Geist, J. J. Dongarra, and R. Manchek, "The PVM Concurrent Computing System: Evolution, Experiences, and Trends", *Journal of Parallel Computing*, **20**(4), pp. 531-546, March 1994.

[5] J. Dongarra, G. A. Geist, "PVM Users' Group Meeting 1993; 1994", *Abstracts and Talks*, Oak Ridge National Laboratory and University of Tennessee, Netlib, 1993, 1994.

[6] D. Duke, "Workshop on Heterogeneous Network-Based Concurrent Computing 1991; 1992; 1993", *Abstracts and Talks*, Florida State University, `ftp.scri.fsu.edu`, 1991, 1992, 1993.

[7] D. Bailey, et al. "The NAS Parallel Benchmarks." *International Journal of Supercomputer Applications*, Vol. 5, No. 3, pp.63-73, Fall 1991.

[8] S. White, A. Alund, and V. S. Sunderam, "Performance of the NAS Parallel Benchmarks on PVM Based Networks", *Journal of Parallel and Distributed Computing*, to appear, 1994.