Heterogeneous Network Computing: The Next Generation

Vaidy Sunderam

Department of Math and Computer Science
Emory University, Atlanta, GA 30322, USA
vss@mathcs.emory.edu

Heterogeneous concurrent computing has emerged in recent times as an effective platform for high performance applications as well as for traditional distributed computing. Performance and functionality that such methodologies can provide, as well as their limitations and potential, have become reasonably well understood. In this paper, we discuss some selected aspects of heterogeneous computing in the context of the PVM system, and describe evolutionary enhancements to the system. These extensions, which involve performance optimization, light-weight processes, and client-server computing, suggest useful directions that the next generation of heterogeneous systems might follow. A prototype design of such a next-generation heterogeneous computing framework is also discussed.

1 Introduction

Parallel computing methodologies using clusters of heterogeneous systems have demonstrated their viability in the past several years, both for high-performance scientific computing as well as for more "general purpose" applications. This approach to concurrent computation is based on the premise that a collection of independent computer systems, interconnected by networks, can be transformed into a coherent, powerful, and cost-effective concurrent computing resource, through the use of software frameworks. The most common methodology for realizing such a mode of computing is exemplified by PVM (Parallel Virtual Machine) – a software framework that emulates a generalized distributed memory multiprocessor in heterogeneous networked environments.

⋆ Research supported by U. S. DoE, MICS grant DE-FG05-91ER25105, NASA grant NAG 2-828, and NSF awards ASC-9527186, ASC-9214149, and CCR-9523544.

Such an approach, which obviates the need to possess a hardware multiprocessor, has proven to be a viable and cost-effective technology for concurrent computing in many application domains. For a number of reasons, including portability, support for heterogeneity, robustness, and a simple but complete programming interface, the PVM system has gained widespread acceptance in the high-performance concurrent computing community.

In the course of experiences with PVM, both by the developers and by the numerous external users, substantial knowledge has been gained about the limitations, potential, and pitfalls of heterogeneous network concurrent computing. One of the most important factors concerns delivered performance for scientific supercomputing applications or their compute intensive mathematical kernels. It is well known, and accepted, that communications bottlenecks are the most serious obstacles to high performance in cluster environments, and numerous projects have tried to overcome these deficiencies with varying degrees of success. From the programming model viewpoint, a more natural and/or automatic computing model would be desirable, since the commonly used paradigm – based on sequential processes cooperating via explicit message passing – is cumbersome and prone to error. Again, language based and graphical toolkit oriented solutions have been proposed and developed but have yet to become mainstream methods. The third major issue in heterogeneous network computing concerns logistics and operational matters. Allocation of resources, load balancing, failure resilience, use of scavenged idle cycles, and security are factors that are many times more important in heterogeneous network platforms than on standalone machines.

The above issues have emerged and have been addressed by different methods during the early-to-mid 1990's when network computing flourished, driven primarily by scientific applications, and in synergy with message passing parallel computing models, algorithms, and tools. Recently however, there have been two important shifts in focus. Heterogeneous computing on a wide area basis is receiving renewed attention, with various forms of national or global virtual computing frameworks. Second, the coupling of parallel and concurrent computing to the World Wide Web, and/or its underlying technologies, is being investigated. Several projects based on one or both of the above are currently in progress, e.g. Netsolve [2], Globus [6], Ninf[10], Fafner [7], and Legion [3]. In this paper, we present a complementary approach – one that evolves incrementally from the existing PVM infrastructure and addresses issues concerning future heterogeneous computing systems.

2 PVM Overview and Experiences

The PVM software system is an integrated set of software tools and libraries that emulates a general purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architecture. The overall objective of the PVM system is to to enable such a collection of computers to be used cooperatively for a concurrent or parallel computation. Detailed descriptions and discussions of the concepts, logistics, and methodologies involved in this network-based computing process may be found in [4,1]; briefly, the principles upon which PVM is based include:

- Explicit message passing model and partitioning, process-based computation and scheduling.
- User configured host pool with dynamic reconfiguration ability, and translucent access to hardware.
- Heterogeneity support for machines and message, portability across multiple platforms.
- Computing entities decoupled in space, and loosely coupled in time.

2.1 The PVM Model and API

The PVM computing model is based on the notion that an application consists of several different sub-algorithms or "components". Components, depending on their functions, may execute in parallel with one another. Further, components that are parallelizable may be executed by replication, thereby reducing the processing time involved. Thus, PVM supports a two-level hierarchy of parallelism – with the relationship between components based on functional decomposition, and intra-component parallelism typically achieved via data decomposition, commonly using the SPMD (single-program multiple-data) model of computing.

The PVM software system contains library routines that enable a user process to register with/leave a collection of cooperating processes, routines to add and delete hosts from the virtual machine, to initiate and terminate PVM tasks, to synchronize with and send signals to other PVM tasks, and routines to obtain information about the virtual machine configuration and about active PVM tasks. Routines are provided for packing and sending messages between tasks. The core communication routines include an asynchronous send to a single task, and a multicast to a list of tasks. Messages can be received by filtering on source or message tag (both of which may be specified as wildcards), with either blocking or non-blocking receive routines. This application programming interface is provided as a library to which programs link; a collection of support software daemons executing on a user-configured host pool implements the process control, message exchange, and other mechanisms, by emulating a general purpose concurrent virtual machine.

2.2 The NAS Parallel Benchmarks on PVM

Cluster/network computing in general and PVM in particular are widely used for high-performance applications including scientific supercomputing. While the general purpose nature of the machines and networks involved preclude efficiencies attainable in dedicated MPP's, PVM performance can be quite satisfactory and frequently very good. To illustrate, we include a report on a recent performance exercise with the NAS computational fluid dynamics parallel benchmarks from NASA. These benchmarks are designed to substantially exercise the processor, memory, and communication systems of current generation parallel computers. The five NPB (NAS parallel benchmark) kernels, with the exception of the embarrassingly parallel application, are all highly communication intensive when parallelized for message passing systems. As such, they form a rigorous suite of quasi-real applications that heavily exercise system facilities and also provide insights into bottlenecks and hot-spots for specific distributed memory architectures. In order to investigate the viability of clusters and heterogeneous concurrent systems for such applications, the NPB kernels were ported to execute on the PVM system. Detailed discussions and analyses are presented in [5]; here we describe our experiences with two representative kernels, using Ethernet and FDDI based clusters.

The V-cycle multigrid kernel involves the solution of a discrete Poisson problem $\nabla^2 u = v$ with periodic boundary conditions on a $256 \times 256 \times 256$ grid. v is 0 at all coordinates except for 10 specific points which are +1.0, and 10 specific points which are -1.0. The PVM version of this application was derived by substantially modifying an Intel hypercube version; data partitioning along one dimension of the grid, maintaining shadow boundaries, and performing nearest neighbor communications. Several optimizations were also incorporated, primarily to maximize utilization of network capacity, and to reduce some communication. Results for the multigrid kernel under PVM are shown in Table 1; the PVM implementation performs at good to excellent levels, despite the large volume of communication which accounts for upto 35% of the overall execution time. It may also be observed that the communications bandwidths obtained, *at the application level*, are a significant percentage of the theoretical limit for both types of networks. Finally, the eight processor cluster achieves one-half the speed of a single processor of the Cray Y-MP, at an estimated one fourth of the cost.

The Conjugate Gradient kernel is an application that approximates the small-

Table 1
V-cycle Multigrid: PVM timings

Platform	Time (secs)	Comm. volume	Comm. time (secs)	Bandwidth (KB/sec)
4*IBM RS6000/550 (Ethernet)	293	96MB	101	973
4*IBM RS6000/550 (FDDI)	185	96MB	18	5461
8*IBM RS6000/320 (FDDI)	110	192MB	32	6144
Cray Y-MP/1	54	—	—	—

est eigenvalue of a symmetric positive definite sparse matrix. The critical portion of the code is a matrix-vector multiplication, requiring the exchange of sub-vectors in the partitioning scheme used. In this exercise also, the PVM version was implemented for optimal performance, with modifications once again focusing on reducing communication volume and interference. Results from our experiments are shown in Table 2.

Table 2
Conjugate Gradient: PVM timings

Platform	Time (secs)	Comm. volume	Comm. time (secs)	Bandwidth (KB/sec)
4*IBM RS6000/550 (Ethernet)	203	130MB	124	1074
4*IBM RS6000/550 (FDDI)	82	130MB	24	5433
16*Sun Sparc SS1+ (Ethernet)	620	370MB	360	1074
Cray Y-MP/1	22	—	—	—

This table also exhibits certain interesting characteristics. Like the multigrid application, the conjugate gradient kernel is able to obtain near theoretical communications bandwidth, particularly on the Ethernet, and a four processor cluster of high-performance workstations performs at one-fourth the speed of a Cray Y-MP/1. Another notable observation is that with an increase in the number of processors, the communication volume increases, thereby resulting in lowered speedups. Our results from these two NPB kernels indicate both the power and the limitations of concurrent network-based computing with PVM — i.e. that with high-speed, high-capacity networks, PVM performance is competitive with that of supercomputers; that it is possible to harness nearly the full potential and capacity of processing elements and networks; but that scaling, load imbalance, and latency limitations are inevitable with the use of *general purpose* processors and networks that most cluster environments are built from.

3 Multithreaded Network Concurrent Computing

In an attempt to address two of the major factors contributing to performance degradation in PVM, viz. load imbalance and latency effects, we are investigating an alternative computing model using threads. 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.

3.1 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.

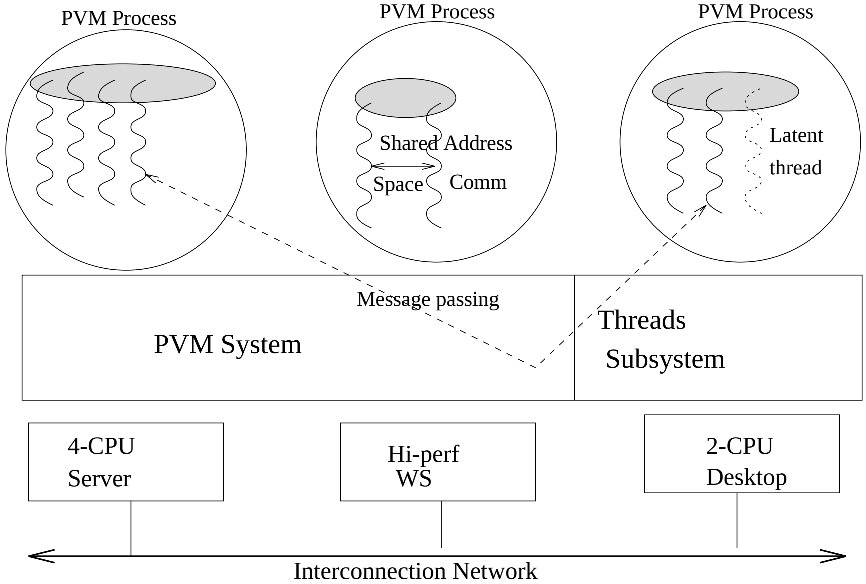

Fig. 1. TPVM Architectural Overview

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 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.

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.

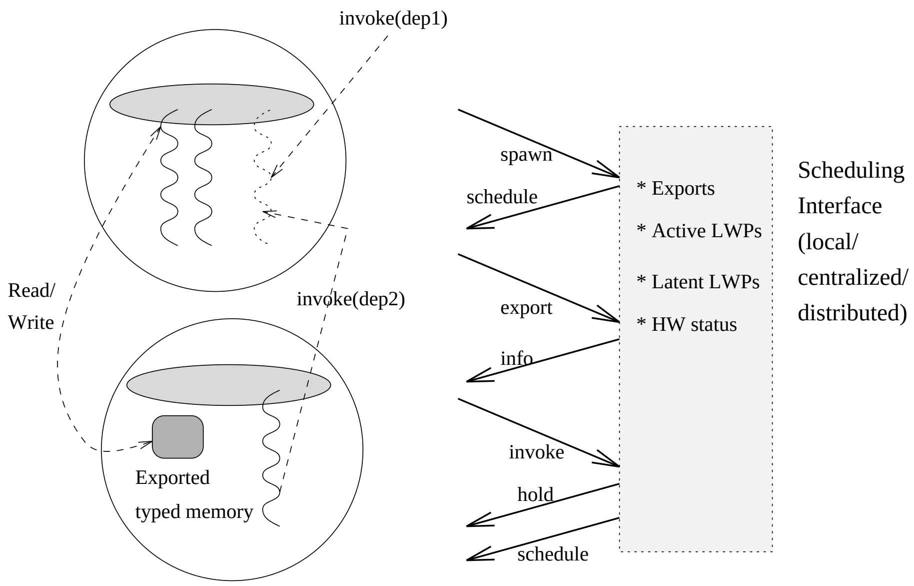

Fig. 2. Schematic of TPVM Implementation

3.2 Implementation and Experiences

Two different models are provided in the TPVM system and a preliminary version of each has been implemented and tested. The first is a process based 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 functions such as `tpvm_export(. . .)` which 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 which then communicate and synchronize via TPVM variants of the usual PVM routines.

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 3. 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		TPVM: CPU's/Threads		
	4/4	16/16	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 3
PVM vs. TPVM times in seconds (SS1+ workstations on Ethernet)

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, special invocation functions are provided. Table 4 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: CPU's/Threads			DflowModel: CPU's/Threads	
	4/16	16/64	16/36	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 4
TPVM ProcModel vs. DflowModel times in seconds (SS1+, Ethernet)

4 Client-Server Computing with PVM

The message passing model supported by PVM is powerful and well suited to SPMD and some MPMD applications, especially in scientific computing. However, providing only a message-passing interface has restricted PVM application categories significantly. Furthermore, the message-passing model requires a number of potentially complex tasks to be programmed explicitly by the user, including process identification and table maintenance; message preparation, transmission/reception, ordering, and discrimination; and task synchronization. In contrast, traditional distributed computing is based on the "client-server" model. In this paradigm, entities known as servers provide services that may be invoked by clients. Implicit in this abstract definition is the potential for varying levels of implementation semantics. To support client-server applications in a more natural manner, we have developed an RPC system for the PVM framework [8]. PVM-RPC is loosely based on the RPC model in which references to remote services mimic a procedure call. The service provider designs and implements a server which may export one or more services. These services can then be accessed by a client via an invocation that looks and behaves like a procedure call.

4.1 An Overview of PVM RPC

PVM-RPC is loosely based on the well-known RPC model in which references to remote services mimic a procedure call. The service provider designs and implements a server which may export one or more services. These services can then be accessed by a client via an invocation that looks and behaves like a procedure call. However, there are a number of issues which arise since the procedure call invokes a non-local service. These issues include name binding, parameter passing, asynchronous processing, and redundancy as it applies to fault tolerance and load balancing. Some of these issues are facilitated by the PVM system, while others are implemented in the PVM-RPC subsystem.

PVM-RPC has four major components, the pvm daemon (PVMD), the service broker (SB), the servers, and the clients. The remainder of this section describes the function and interaction of these components. The PVMD is an integrated part of the native PVM system, and is responsible for identifying and managing global resources (such as available hosts), message routing, and authentication. Before any PVM program can run, the master PVMD must be started. After the master PVMD starts and processes the list of available hosts, local PVMDs start on each available host. Once a local PVMD is running on a host, that host is available for use by the PVM system. The SB maps service names, as known to clients, to a tuple consisting of a server and service id recognizable to the server. In order to ensure that PVM-RPC is at least as reliable as PVM, with respect to hardware failure, the SB runs on the same machine as the master PVMD. While there are various ways that the SB can start, any server that is trying to register a service will start the SB if it is not already running.

A complete scenario of SB interaction with the rest of the PVM-PRC system is now illustrated: when a server starts, it first locates the SB from the PVMD using the table facility. If the SB is not running, then the server starts an SB using *pvm_spawn* and, again, attempts to locate the SB from the PVMD (the second lookup of the SB is done to prevent race conditions between two servers). The server finally registers its services with the SB and no further communication between the server and the SB is required. Similarly, when a client wishes to use a service, it first locates the SB from the PVMD and then gets a list of available services from the SB. In the client's case, a request for a list of registered services results in a complete list of all known services and addresses, which the client caches locally to eliminate unnecessary communication between the client and SB. While the list of servers and services may change during the client's execution, this is not an issue unless the client requires a service whose address is not cached, or there is a problem with a known provider of a service (due either to server failure or congestion). In either of these cases, the client can update its list of known services. If more than one server is available for a particular service, the client will, by default, interleave requests among available servers.

4.2 PVM-RPC Programming

In using PVM-RPC, server programs and client programs need to be written. To create a server, services must be declared and coded, using a syntax similar to standard C. The primitive *pvm_rpc_service* is used to declare a service. Its syntax is:

```
pvm_rpc_service(service_name, [[IN, OUT, INOUT]
                 {PVM_TYPE} variable_name, ...]);
```

This declaration is then followed by standard C code that defines the implementation of the service. All parameters are treated as pointers, with array notation favored for element access. The key words IN, OUT and INOUT are used to give the server a hint as to whether variables need to be copied from client to server (IN), server to client (OUT) or bidirectionally (INOUT). If none of IN, OUT or INOUT is specified the variable is assumed to be INOUT. The type of a parameter is PVM_TYPE, which must be specified, and must be a legal PVM type such as PVM_INT (PVM_CPLX and PVM_DCPLX are not supported). Finally, for each parameter the programmer specifies there is a 'hidden' integer constant with the name {parameter_name}_size (i.e., _size appended to the parameter name) that specifies the number of elements the parameter named variable_name contains. A complete example demonstrating the usage of *pvm_rpc_service* is shown in Fig. 3. After all services are declared,

```
pvm_rpc_service(add_vec, IN PVM_DOUBLE a, IN PVM_DOUBLE b,
                OUT PVM_DOUBLE c) {
    int i;
    for(i=0; i < a_size; i += 1) {
        c[i] = a[i] + b[i];
    }
}
```

Fig. 3. Example showing how to declare and implement a service that sums two vectors.

a call to *pvm_rpc_main* terminates the program. This function will generate a main program containing the code for registering the defined services and processing requests from clients. There are no restrictions on the number of services a server may contain, nor on references to external functions by a service as allowed by C. A server may in itself behave as a client. While a service may call external functions, all services must be declared in the same file that contains the *pvm_rpc_main* statement. Figure 4 shows a complete server containing one service, *add_vec*.

Clients may invoke a remote procedure via one of two calls. The first call, *pvm_rpc_sinvoke* is synchronous; the second, *pvm_rpc_ainvoke* calls the server function asynchronously. In the latter case, a subsequent *pvm_rpc_wait* statement acts as a synchronization point for the RPC. To limit unpredictable behavior, parameters used in an RPC are not modified by the RPC routine until completion of the remote procedure (i.e, the parameters are only modified during the *pvm_rpc_wait* statement). An example of a client invocation of an RPC call is:

```
#include <pvm3.h>
#include <pvm_rpc.h>
pvm_rpc_service(add_vec, IN PVM_DOUBLE a, IN PVM_DOUBLE b,
                OUT PVM_DOUBLE c) {
    int i;
    for(i=0; i < a_size; i += 1) {
        c[i] = a[i] + b[i];
    }
}
pvm_rpc_main;
```

Fig. 4. Example of a complete PVM-RPC server.

```
pvm_rpc_invoke(add_vec,a,SIZE,b,SIZE,c,SIZE);
```

where a, b, and c are double precison vectors a[SIZE], b[SIZE], and c[SIZE].

4.3 RPC and Coordination

The PVM-RPC model is a straightforward, yet effective mechanism that allows the user to perform distributed computation in an elegant and natural fashion. Furthermore, PVM-RPC enhances PVM by adding support for client-server computation. The current implementation achieves these features with minimal performance penalty. In preliminary benchmark experiments, simple textbook applications in scientific computing such as matrix multiplication and particle-in-cell codes were implemented using PVM-RPC, with partitioning and work allocation methods chosen such that total communication volume and computation were equal (or very close) to their message passing counterparts agains which they were compared. In all cases, PVM-RPC implementations executed in elapsed times that were within 8% of standard DMPP versions of the applications, with the RPC implementations performing better in some cases.

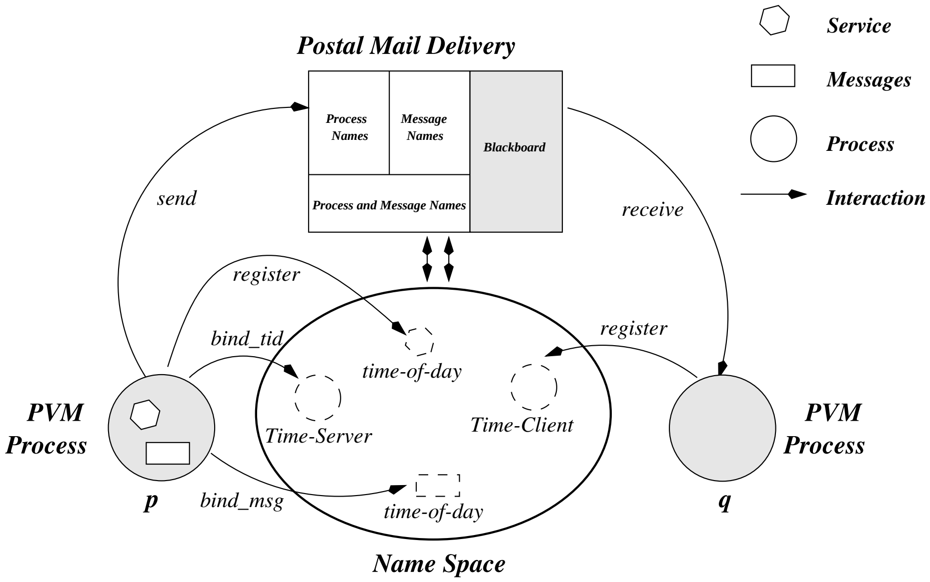

Fig. 5. PVM Coordination Model

Having established that a client-server computing model is feasible and useful in the context of message passing systems, a more general and comprehensive abstraction was considered. This model, known as the coordination model, provides for cooperation and interaction between *independent* entities; furthermore, the coordination may be completely decoupled in space and in time. In order to enhance the PVM model with coordination facilities, a global name space for processes, messages, and services, is needed. In addition, "postal" messages to enable time-decoupling are introduced into the model. These messages may have persistent, hold, or instant delivery semantics, and, in a flexible and dynamic manner, provide message exchange capabilities ranging from synchronous send/receive to mailbox modes. The PVM coordination model is shown in Figure 5; a detailed paper [9] contains more specifics of the preliminary implementation, as well as the client-server API supported.

5 Agents, Scripts, and Heterogeneous Computing

Our experiments with threads, client-server computing, and coordination, all within the context of a traditional, process-oriented message passing system, have demonstrated the potential for the unification of paradigms from distributed computing and parallel processing. The thread-based concurrent computing infrastructure was found to be very valuable in permitting finer granularity, hiding latency, and exploiting multi-CPU workstations. The client-server and coordination models showed that even for traditional applications, there are several benefits of using a request-response paradigm especially with regard to expressiveness and ease of use. Considering that there is little or no performance degradation, this is very valuable; especially given the enhanced potential for transparent fault tolerance and load balancing that can be built into RPC servers.

Building further on the idea of a service-oriented computing framework, with well-encapsulated multithreaded computing entities, a natural progression is to move towards agent-based computing. When imposing the constraints or requirements of portability and heterogeneous computing, scripting languages appear to be promising. In order to investigate the potential for these ideas, we propose the IceT heterogeneous network computing framework.

5.1 The IceT Model

The IceT model is based on the premise that the internet infrastructure holds tremendous promise as a global computer. Despite its current and limited use as an information retrieval system, the internet consists of thousands of fast processors and is capable of yielding gigaflop performance provided that:

(i) heterogeneous processors can be organized to cooperate on a single user's problem,
(ii) remote operations can be initiated in a machine-independent manner,
(iii) highly efficient protocols exist, to support reliable uni- and multicast communication, with varying and selectable qualities of service, and

(iv) software to support all this functionality can be created.

We have undertaken a pilot project to investigate the feasibility, using the IceT model, of how a variety of information transformations ranging from voice-mail collection systems to high-performance distributed computations can be performed efficiently on the internet. Given a simple, but highly effective object naming and location scheme (i.e., URLs) and versatile servers, simple browsers become interfaces to powerful computing environments. This is made possible by IceT servers that operate as daemons – utilizing standard protocols to interact with browsers and IceT protocols to talk to other servers – and IceT's CMPI (core message-passing-interface) servers. In contrast to usage of the Web as a passive information retrieval source, IceT transforms the Web into an active resource (computational engine) that can repeatedly distribute and route both operations and data, perform distributed data transformations, fuse results and compute both iteratively and recursively.

An IceT environment is established once a set of IceT servers has been initiated on a distributed set of machines. Utilizing current technology, the internet is capable of transporting objects (i.e., datafiles, object files, source files, and IceT scripts) between IceT servers and IceT browsers. IceT browsers determine which objects are to be transported and where these objects are to be transported to. Effecting such an operation may be as simple as naming (and thus invoking) a local, user-defined IceT script. At specified target locations, IceT servers accept and interpret IceT scripts, carrying out tasks as specified. Failed servers are replaced as necessary. IceT scripts contain interpretive code to:

- generate and transport IceT scripts to servers at specified locations,
- invoke well-defined executables,
- invoke object files through dynamic linkage,
- invoke other iceT scripts, possibly remotely, including scripts that fetch source, object or executable files from other sites, and
- replicate IceT scripts along with associated data.

A schematic of the proposed operational model of IceT is shown in Figure 6. To enable the utilization of remote executables, IceT scripts invoking executables allow specification of inputs and outputs that are read from and written into data-envelopes. Indeed, data-envelopes act as caches for storing transient data obtained during functional compositions, and consist of special IceT commands enveloping the data. IceT scripts are code segments processed by an interpreter (e.g., PERL), and are subject to an imperative style of programming. That is, all tasks are performed sequentially, iteratively and conditionally. Nevertheless, IceT operations will enable and encourage recursion through recursive interpretation of scripts or simply recursive instantiation of executables. With the exception of IceT scripts, all computational invocations

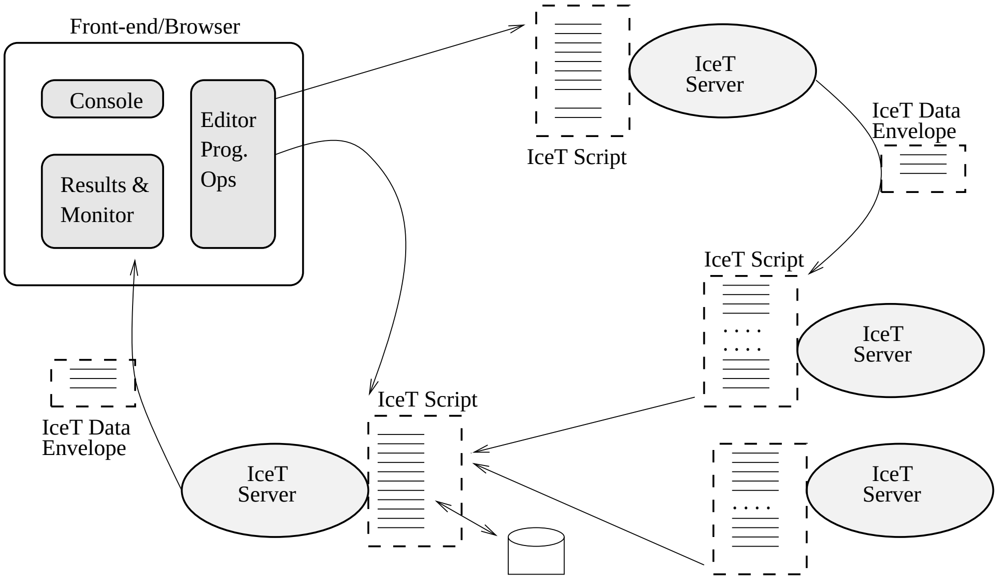

Fig. 6. IceT Operational Overview

must involve *basic blocks*. Starting with such an architectural model, we are attempting to build a prototype system in which experimentation can be performed, and the concepts evolved into contemporary heterogeneous computing systems.

6 Conclusions

Heterogeneous concurrent computing on networks of independent computer systems is a technology that has evolved during the last six or seven years. It has been extremely valuable as an effective and ubiqutous high-performance computing platform, and has permitted parallel computing without hardware investment. At the same time, the limitations of network computing have also become evident – some of which have been overcome by novel approaches, tools, and implementations. These experiences, in conjunction with complementary advances and developments, such as the MBone and the World Wide Web, have prompted the quest for the next generation of heterogeneous network-based concurrent computing systems. In this paper, we have outlined an incremental and evolutionary approach towards this goal.

References

[1] 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.

[2] H. Casanova and J. Dongarra, "NetSolve: A Network Server for Solving Computational Science Problems", *Proc. Supercomputing '96*, November 1996 (to appear).

[3] A. Grimshaw, et. al., "A Synopsis of the Legion Project", *University of Virginia Technical Report CS-94-20*, June 8, 1994.

[4] V. S. Sunderam, "PVM : A Framework for Parallel Distributed Computing", *Journal of Concurrency: Practice and Experience*, **2**(4), pp. 315-339, December 1990.

[5] A. Alund, S. White, and V. S. Sunderam, "Performance of the NAS Parallel Benchmarks on PVM Based Networks", *Journal of Parallel and Distributed Computing*, Vol. 26, No. 1, pp. 61-71, April 1995.

[6] I. Foster, S. Tuecke, "Enabling Technologies for Web-Based Ubiquitous Supercomputing", *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, Syracuse, August 1996.

[7] G. Fox, W. Furmanski, "Web Based High Performance Computing and Communications", *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, Syracuse, August 1996.

[8] A. Zadroga, A. Krantz, S. Chodrow, V. Sunderam, "An RPC Facility for PVM", *Proceedings – High-Performance Computing and Networking '96*, Brussels, Belgium, Springer-Verlag, pp. 798-805, April 1996.

[9] O. Krone, M. Aguilar, B. Hirsbrunner, V. Sunderam, "Integrating Coordination Features in PVM", *Proceedings – First International Conference on Coordination Models and Languages*, Cesena, Italy, Springer-Verlag, pp. 432-435, April 1996.

[10] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka and U. Nagashima, "Ninf : Network based Information Library for Globally High Performance Computing", *Proc. of Parallel Object-Oriented Methods and Applications (POOMA)*, Santa Fe, Feb., 1996.