P^3: Parallel Peer to Peer
An Internet Parallel Programming Environment

Licínio Oliveira, Luís Lopes, and Fernando Silva

Departamento de Ciência de Computadores & LIACC
Faculdade de Ciências - Universidade do Porto
Rua do Campo Alegre, 823
4150 Porto, Portugal
{lsoliveira,lblopes,fds}@ncc.up.pt

Abstract. P^3 is a next-generation Internet computing platform, building upon other experiments and implementing new ideas for high-performance parallel computing in the Internet environment. This paper describes its run-time system, programming model and how it compares to current state-of-the-art systems.
Keywords. Peer to Peer, Distributed Computing, Parallel Computing, High Performance Computing.

1 Introduction and Motivation

In recent years there has been a growing interest in *High Performance Computing* using distributed systems, namely the Internet itself, as the resource provider [10, 17]. This focus on current distributed systems stems from the observation that individual machines in such systems spend most of their time either idle or with very modest workloads, and that their otherwise wasted computing cycles may be gathered and used to perform large scale computations [17, 7, 8].

Grid Computing establishes the foundations for an infrastructure capable of transparently providing computational cycles over wide area networks. Such systems behave much like service providers and the first implementations are very recent and subject to discussion namely on standardization [9, 11].

Peer-to-Peer Computing is often associated with Grid Computing as it provides a fundamental change in paradigm for programming distributed systems. It provides highly efficient communication by making nodes exchange data directly, without intermediate routing servers. It also improves service availability as these are not concentrated in a few servers with higher failure rates. Several such systems have been proposed to date, mostly focusing on the sharing of computational cycles or data-storage [17, 8, 7, 10, 12].

So far the work on Peer-to-Peer systems has been focusing on infrastructure for Grid systems, stand alone parallel distributed systems and file-sharing systems. Our main interest in Peer-to-Peer computing is in using it as a basis for the development of an efficient, highly available, parallel programming system

which we named P^3 (Parallel Peer-to-Peer). In this perspective we argue that current implementations lack adequate solutions on two fundamental issues.

First, to our knowledge, there is no system that integrates features such as: dynamic discovery and management of resources, scalability, accessibility, availability, portability and fault-tolerance, into an integrated environment optimized for parallel computing. In P^3 we aim to provide for these features by carefully designing the run-time system. In short, our proposal is as follows:

- dynamic discovery and management of resources is supported by dynamically changing sets of nodes that monitor the network for resources and manage the computational workload;
- the portability issue is solved by using an hardware independent format for the run-time system implementation, in this case Java;
- scalability is supported by the use of dynamic workload balancing between computing nodes, peer-to-peer communication and dynamically changing sets of resource manager nodes;
- fault-tolerance is supported by keeping redundant copies of the run-time system state and using checkpoint/rollback mechanisms;
- availability of resources is guaranteed by allowing any node who has the run-time system installed to join a computation dynamically;
- finally, accessibility, by not requiring any specific properties for a node to belong to the P^3 community.

A second, and most important, point is the lack of integration of an adequate parallel programming model into existing systems, as most of them deal with independent tasks. In our proposal, such a model must feature:

- builtin dynamic work balance, given the variable number of the computational resources;
- the specifications for the problem's computation and for data partitioning should be orthogonal; this allows a far cleaner and intuitive programming style;
- a shared object-space that provides a network-wide, dynamically changing, virtual shared memory.

The shared object-space, actually a Peer-to-Peer distributed file system, provides all the support required for interprocess communication and synchronization. The programmer must provide the code that solves the problem without thinking of data partition. The work partition strategy is specified independently by the programmer in a P^3 application and describes the behavior of a node when a dynamic work request comes from a node joining the computation.This model hides the architectural complexity from the programmer at the expense of additional run-time system complexity. The implementation of the P^3 framework is ongoing research work.

The remainder of the paper is organized as follows. Section 2 describes the software architecture of the P^3 system and its proposed implementation. Section 3 follows with a description of the programming model with an example. Finally, in section 4 we issue some conclusions and discuss future work.

2 P^3 System Overview

P^3 is a Java run-time system that uses user specified resources to provide support for a distributed programming environment in a Peer-to-Peer network. P^3 provides transparent access to network peers, persistence using shared disk storage, fault-tolerance, high availability, portability, support for parallel computations according to a specific programming model and an extensible framework.

The P^3 network is fully autonomous and self-healing in the sense that it does not need any administration nor centralized control procedures and provides the means for fault-tolerance and high availability. More specifically, peers can join and leave the network at any time and nodes do not need permanent Internet connections (although this is preferential).

The P^3 run-time tries to be as unobtrusive as possible to the host peer system, running in the background while possible and mandating child computations to have minimum operating system priority. If such restrictions should still be insufficient, the kernel scheduler could easily be adapted to include any other priority queueing policy, based, for example, on standard uptime information or collected statistical information of the runtime execution.

Although security is not a priority in the current implementation of the P^3 run-time, it is an obvious concern, especially in the Internet environment. As such, P^3 tries to be as confidential as possible by not transmitting any personal or unspecified system information through the network. As a policy, the previous is obviously unsatisfactory, but future work will address this problem more adequately.

2.1 Conceptual Organization

The P^3 network is organized in two distinct sets of nodes – *manager* nodes and *compute* nodes (fig. 1). The idea is borrowed from previous work in [10] and [14], where complete decentralization of the Peer-to-Peer network was proven, in practice, not to scale well. As such, a balance must be met between Peer-to-Peer common practice of complete decentralization of control and standard client-server techniques. P^3 uses the hybrid approach of maintaining volatile nodes marked as manager nodes[1], which perform coordination operations, maintaining quality of service. By volatile we mean that the set of manager nodes may dynamically change, expand or even contract (though it can never be empty), given the state of the current manager node set. Also, those changes are completely controlled by the run-time system, without user intervention.

Peers are known in the system by their PeerID. They are assigned by the responsible manager nodes at first login and are persistent through all sessions.

[1] Similar, in concept, to *Gnutella*'s notion of *ultra-peer*.

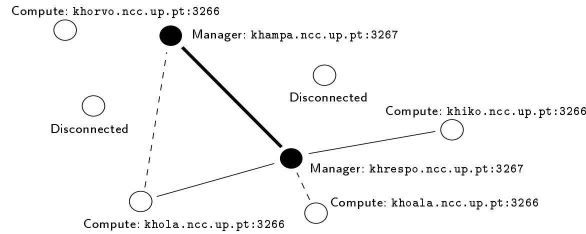

Fig. 1. Example of a possible P^3 network topology

P^3 provides an *object-space* abstraction that resembles the concept of a distributed, shared file-system. In this object-space, objects can be stored and retrieved (by name) with atomic run-time system operations and name-spaces can be created or deleted. Other operations are possible using the standard tools provided by the P^3 run-time, which include object meta-search (e.g. search by object type/contents) or storage of arbitrarily complex objects.

The shared file-system is stored across all compute nodes, thus providing almost as much disk storage as the sum of free space of all nodes in the network[2].

Manager Nodes. The manager node set is the collection of P^3 nodes that, at a given time, hold the responsibility of maintaining state coordination across the P^3 network. The state of the P^3 run-time is the union of the following:

- Peer routing and meta-data information
- File-system data, meta-data and caching information
- Global application status

Given the responsibilities of such nodes, their allocation scheme should be efficient and, by precedence, respect the following:

1. Permanent Internet connection
2. Low latency and high bandwidth connection
3. Availability of disk space
4. Good overall system performance

Nonetheless, as it is not necessarily possible to allocate peers that fulfill all of the specified goals, the choice should fall on the best rated nodes according to the requirements given above.

P^3 uses redundancy in the manager node set to provide high availability and fault-tolerance. For each manager node there is one or more shadow manager nodes. These nodes are considered secondary, in the sense that they do not hold authoritative data. They maintain consistency with the primary manager node through a round-robin differential state copy. Several primary manager nodes

[2] In reality, each node can configure the amount of disk space devoted to shared storage and the high availability kernel features use more space for data backup.

may exist but, in that case, they should be responsible for exclusive partitions of the system's state. Shadow manager nodes, however, need not comply to the later, to avoid allocation of excessive manager nodes. The primary and its shadow nodes must, at all times, know of each other's existence.

The manager node set grows as soon as each primary manager node determines that it cannot keep up with peer connections or runs out of disk space; the set can also grow if shadow manager nodes run out of available disk space but in this case another shadow manager node can help and the set does not need to increase in size.

Compute Nodes. All nodes that do not have manager capabilities[3] are called compute nodes and contribute to the system with usable resources. The resources that a compute peer may bring to the system are, in the current run-time implementation, of two kinds: processor time and disk storage. Several other resources may be harnessed in the future, by using the run-time system extensibility properties.

Each compute node may only be contributing to a single P^3 application at any given instant, thus avoiding local resource competition and simplifying the run-time system. The amount of work each compute node has currently in hand is dictated by the manager node that was responsible for its assignment. Compute nodes expect to receive work from the manager nodes, upon request. It is the manager's responsibility to correctly select the application that each compute node receives work from.

The Shared File-System. P^3 has builtin support for a Peer-to-Peer file-system, much in the style of current Peer-to-Peer file-sharing networks [10, 14]. The distinction between those systems and P^3 is the ability to search and store arbitrarily complex Java objects by reference, transforming the file-system into a shared associative memory or object-space. The file-system also supports the existence of arbitrary name-spaces, where objects can be independently stored. A name-space is a subset of the object-space where objects can be stored without name clashing problems. This means that object `"A"` living in name-space `"X"` is not the same as `"A"` in name-space `"Y"`. Name-spaces are identified with reference keys, NamespaceIDs, which are strings.

The proposed standard for P^3, imposes the presence of at least one, global, name-space, named `"global"`, and a local name-space for each P^3 application. This ensures that each application has, at least, one absolutely safe name-space in which to store its objects (accessible through the p3lspc method). This local name-space identifier is assigned at application startup time, by an adequate manager node. Additional management of name-spaces can be done at any time, using name-space creation (p3create) and deletion (p3destroy) methods of the appropriate P3NameSpace Java class.

[3] The role of a node may change at any time, given the needs of the system.

Each object in the file-system has an associated key, ObjectID, and name-space tag, through which it is referenced. Optional meta-data can be specified for each object, through appropriate methods (p3metadata) in the main Java shared object class, P3Object. Several methods are also devoted to basic object sharing: synchronous read/write (p3get/p3set), asynchronous read/write (p3bind/p3send) and object search (p3search). The synchronous read/write methods are multiplexed into the basic Java data-types to provide commodity to the application developer, just like many other parallel programming libraries.

Objects are stored in a Peer-to-Peer fashion, across P^3's network nodes. Every P^3 node may store objects and objects can be stored in any P^3 node. Object access information is stored and kept consistent by the manager node set. Synchronization is needed when updating the state of an object in the object-space and as such, manager nodes have the ability to block access to objects during the time frame in which such actions occur. The principle is simple, as each manager node must be informed of state changes for each object it is responsible for, object locking is just a matter of disallowing or blocking the operation if a given flag is set. Certain applications might be interested in using such synchronization primitives and for that purpose one provides synchronization methods, such as p3readLock, p3readUnlock, p3writeLock and p3writeUnlock within the P3Object class.

In this context, file-system caching is of vital importance to provide higher system performance, high availability, fault-tolerance and avoiding resource deadlocking. Caching is implemented through redundancy, in a hierarchical two level scheme. Nodes are able to store local, non authoritative, object copies and asynchronously verify its consistency (local caching). Manager nodes may at any time propagate copies of objects for which request rates rise above a given threshold or for which the storage nodes disconnect from the network (global caching).

2.2 P^3 Run-Time System

The P^3 run-time system is being implemented in Java. Java was chosen due to its portability, extensibility and amazing number of features already present in the language, and not readily available in other programming languages, like C or C++. Besides, Java is the acknowledged standard for portable high performance computing projects and, as such, lessons can be learned from the implementation of those projects.

Java presents the possibility of implementing the run-time system as an applet. However, the run-time system could not have been implemented with its current features, due to the security policy enforced by the environment where applet based applications run. For example: peers could not communicate directly and a high performance persistent shared file-system could not be implemented. As such, P^3's run-time system is implemented as a Java application. Other Internet parallel computing systems follow the opposite (applet) path [3, 6].

Every machine wishing to participate in running computations must install the Java application which encapsulates the run-time system. Once installed, the run-time will need no additional setup, apart from configuration tuning.

In P^3 there is support for off-line computations; the run-time engine does not need to have network access at all time. In fact, all P^3 computations occur off-line. It is only in the presence of a blocking operation that network access must be obtained. This would be the case, for example, of a blocking read/write operation.

Futhermore, when network access is first obtained, in each session, a network login procedure is executed. This procedure is important for several reasons, the first being to refresh the manager node tables with peer information, avoiding probing every node in the network for available resources. Another reason is checking the status of the current node in respect to the task it is running.

When dealing with distributed applications, good resource management and allocation policies are vital to system performance. In the Internet environment, such concerns must be even stronger, because volatile resources are extremely common. P^3 uses a meta-data approach to the problem; manager nodes are responsible for collecting and storing relevant meta-data information for every compute node. However, as each manager node has to store information for each compute node it is responsible for, collected meta-data must be kept to a bare minimum. Usually, the interesting information to retrieve from each compute node includes: the executing P^3 application and its state, the available disk space, machine load and additional routing information.

As already stated, P^3's main goal is to reuse computing cycles of Internet connected machines. However, the P^3 run-time does not limit itself to that network environment, it is possible to use P^3 to harness the computing cycles of institutional intranets, for example. It should be emphasized that P^3 is not only a run-time system for parallel computation; it is easy to envision several non-parallel applications that could be implemented in P^3. A large scale multimedia database constructed as a P^3 application that interacts with several non P^3 client applications is an interesting example.

Run-Time System Organization. Internally, the run-time system subdivides itself into two layers: the kernel module layer, which includes the most important and necessary features like communication, fault-tolerance, persistence, resource management and discovery; and the service layer, which are all non-basic standard features of the P^3 run-time that use the kernel to access the P^3 network resources. The parallel programming model is an example of such a service. Dividing the run-time in two layers, induces a three-tier model for a P^3 application (fig. 2). This widely used paradigm allows a higher degree of control and specialization of each layer, while making project development easier. The P^3 three-tier

model is not opaque, in the sense that it allows the top layer (applications) to access both lower layers (kernel and services) of the run-time system.

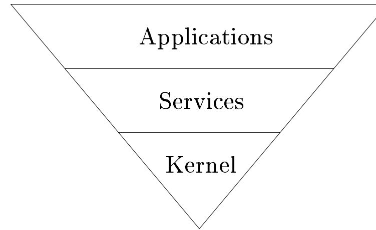

Fig. 2. P^3's three-tier model

Kernel. The P^3 run-time system kernel is the set of basic system features, like communication support, object-space access libraries and so on. The implementation tries to be expandable and modular, permitting quick and easy implementation of new replacement modules. The modules present in the current run-time implementation are those in table 1.

Table 1. Current run-time system kernel modules

Name	Description
Configuration	Permits static and dynamic kernel configuration
Communication	Communication interface and default TCP/IP backend
Object-space	Name-space management and object search/storage/retrieval procedures
Management	Manager node sub-system
Kernel	Kernel management and entry point

Although modular, the run-time engine expects some intercommunication between modules to take place, e.g. the communication module works with the management module to permit interaction compute/manager and manager/manager communications to take place.

Services. The three tier-model already explained, has the interesting property of permitting disconnected development paths to the three layers of project development, namely the kernel, the services and all applications wishing to use the system, because once a kernel/services interface becomes stable, there is no need to make any change to an existing service (or application), for each kernel (or services) update.

All a P^3 service needs, in order to be considered as such, is to subclass the P3Service class. From that point on, apart from defining some obligatory methods, the service can interact with the kernel layer to access P^3's facilities and provide whatever functionality it desires.

Service libraries should be distributed with the run-time system due to their static registration within the kernel. A useful P^3 feature would be to permit services to register within the kernel dynamically, providing a plug-in based approach as in [12]. However, at current development stage, the run-time system is not capable of doing so; registration must be statically configured and known at run-time startup time.

As an example, the parallel programming model proposed in the following section is implemented as a run-time system service.

3 Programming Model

In P^3 we want to introduce a programming model for the distributed environment that both hides the underlying architectural complexity of the system and, that is intuitive for the programmer. The first major task for a programmer in a parallel system is that of implementing the partition of data among a statically defined set of computing nodes. Given this partition the computational task for each node is then coded and this usually includes frequent interprocess communication for synchronization purposes or to share intermediate results. The implementation of this computation is usually highly dependent on the data partition chosen and thus not appropriate for environments where availability of computational resources changes dynamically.

In P^3 we solve this problem by asking the programmer to code a solution for the problem (method p3compute) and to specify a work partition procedure (method p3divide), orthogonally. For a matrix multiplication, for example, p3compute would compute the result matrix from available input matrices. On the other hand, p3divide describes how to divide the current matrix into smaller slices to assign to other nodes joining the computation. Thus, our P^3 computation involves a set of nodes calculating a number of sub-matrices of the result matrix, possibly of distinct sizes. It is the programmer that indicates the way in which a node's work is split and sent to a new node (method p3divide). This is similar to the task of implementing data partition in parallel programming except that in P^3 this procedure is invoked dynamically and adapts to the evolution of the computation.

Process synchronization and, in general, message passing are written as read and write operations on a shared object-space.

Fault-tolerance is introduced in the programming model by using system primitives supporting computation checkpointing (method p3checkpoint). These primitives create a full dump of the computing object into a standard Java format that then gets marshaled into a persistent medium. If at any time during the computation, a node failure is detected, a copy of the node's previous

checkpointed computing object is fetched and the computation resumes (method p3restart).

3.1 The Execution Model

The programming model is implemented in the P^3 run-time as a service. We shall now describe the associated execution model encapsulated in the p3.services.parallel Java package.

First of all, application startup is done through a standard programming model service component, defined in the Java class P3ParallelUploader. This small application uploads the pre-compiled Java class (which must be a subclass of P3Parallel) onto the manager, making it aware of its existence. The first time the manager decides to allocate a node for this application, it will flag the node's run-time that it must start the application by running the p3main method. After doing so, the allocated node's run-time system begins to compute, executing the method p3compute. All these actions are invoked asynchronously and automatically by the service.

The programming model service is built on top of the following, simple, loop of execution:

This means that when a node's run-time is not computing a P^3 task, it is probing the managers for more work. Examining fig. 3 we can see the main algorithm for work subdivision in P^3. A node (node A, in the figure) requests work, not directly from another node, but through a manager node, which may query other manager nodes, and will notify the node it determines to be the most suitable (B) to share some work with the first.

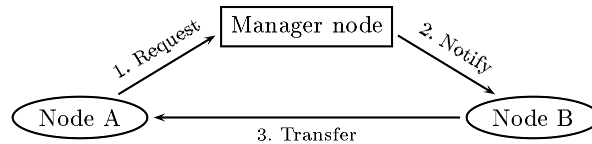

Fig. 3. Work request and attribution scheme

The work subdivision strategy is implemented by the programmer under the p3divide method. It could be as simple or as complex as one may desire. The events that follow the invocation of p3divide are depicted in fig. 4.

The division starts when a node (the nodes are the same as those in figure 3) receives a notify message from a manager node, saying it must send a work task to another node.

During this process a clone of the current node's compute object is produced and the first object is left almost intact. The difference between the two final

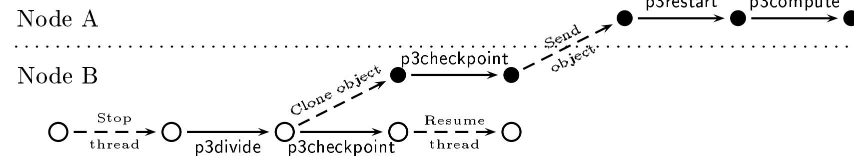

Fig. 4. Simplified execution flow after node B receives a work request from node A

compute objects is that one runs p3restart while the other does not. Besides that, the objects are identical, as their state was preserved by using Java serialization facilities. They are also different in the sense that they resume computation in different locations; in node B, execution resumes from the point where it first stopped and node A restarts by invoking p3restart and then p3compute.

This behavior allows the use of save/restore variables that permit the partitioning of the problem, by carefully coding p3compute. It should be noted that all interaction is implicit and hidden from the application programmer, by the run-time system control logic.

3.2 The Programming Interface

The programming interface for P^3's parallel programming model is implemented in one major class named P3Parallel. All P^3 parallel applications are instances of extensions to this class.

The P3Parallel Class The application programmer has to provide an implementation for the following methods of the P3Parallel class:

- p3main, is the method that sets up the P^3 computation. It can be used to perform some I/O or for the initialization of data-structures before the actual computation begins;
- p3compute, is the method implementing the unit of parallel work in the P^3 system for a given application;
- p3divide, describes how the work block allocated to the current computation is divided when a request for work is received from another node;
- p3restart, is invoked whenever a branch of the computation is resuming after a node failure or on the arrival of new work. In general, the computation must have been previously checkpointed at some point in the past, and execution will resume from that point (note that, a checkpoint is automatically created right after p3divide is invoked).

The P3Parallel superclass implements other methods, such as p3checkpoint, which may be used as a default action or re-implemented by the programmer in case some application specific needs must be fulfilled.

An Example: Matrix Multiplication In the following we describe an implementation of the usual matrix multiplication operation. To implement a solution, we define a class for the application, P3MatrixMultiply, that extends P3Parallel. The implementation of the application simply requires the definition of its attributes and methods.

In this example written in P^3, we compute the matrix C, from A and B, parameterized on a few attributes. The code for the computation has no reference whatsoever to data-partition among cooperating nodes. It is apparently a sequential code. The program then provides a procedure that describes how to obtain sub-matrices for C, dynamically, when a request for work arrives.

- global class attributes: a shared object `n` for the size of the matrix, shared objects `a`, `b` and `c` to contain, respectively, matrix A lines, matrix B columns and each cell of matrix C. Additional attributes include `line`, `column`, `l`, `m` and `max` which are integers used to keep track of the evolution of the computation.
- method p3main() is used to perform the initial matrix setup:

```
void p3main() {
    try {
        n.p3bind(p3lspc(), "n");
    } catch(P3BindException e) { p3abort(); }

    n.setInt(10000);    // Set matrix size to 10000x10000
    for ( int i = 0 ; i < 10000 ; i++ ) {
        // Code for initializing line i of matrix A
        // Code for initializing column i of matrix B
    }
}
```

- method p3compute() computes the matrix C. Before computing the new entry at row `line` and column `column`, it binds line `line` for A, column `column` for B and `c` for the resulting cell. Then, a cycle computes the value for the cell. Finally, a check is made to verify whether it is time to advance to the next line in the current work block.

```
void p3compute() {
    try {
        n.p3bind(p3lspc(), "n");
    } catch(P3BindException e) { p3abort(); }

    while ( line < max && column < n.getInt() ) {
        try {
            a.p3bind(p3lspc(), "A"+line);
            b.p3bind(p3lspc(), "B"+column);
            c.p3bind(p3lspc(), "C"+line+"_"+column);
        } catch(P3BindException e) { p3abort(); }
```

```
        for ( int i = 0 ; i < n.getInt() ; i++ )
            c.setInt(c.getInt()+a.getIntPos(i)*b.getIntPos(i));
        if ( ++column > n.getInt() ) { line++; column = 0; }
    }
}
```

– method p3divide() defines the way the computation reacts to a request for work from a coordinator node. The method is invoked synchronously and, in this case, it divides the work into two similar blocks of size `(max-line+1)/2`. The block to be given away will compute from line `line+(max-line+1)/2` to line `max`. The current computation will continue from `line` to the new value of `max`.

```
boolean p3divide() {
    l = line + (max - line + 1) / 2;
    m = max;
    max = line - 1;
    return(true);
}
```

– method p3restart() describes the computation restart procedure. We initialize the attributes `line` and `max` to the previously checkpointed values:

```
void p3restart() {
    line = l;
    column = 0;
    max = m;
}
```

As seen, the main task for the programmer lies in the code for p3compute and p3divide. These define work partition and computation for the problem orthogonally. The big advantage of this programming model relative to the usual models used in current parallel distributed systems stems from the following features:

– we allow the computing node pool to grow dinamically. The way work is partitioned among compute nodes is controlled by the programmer at the application level;
– by implementing a solution for the problem abstracting away from data partitioning among nodes we make the code for the problem more explicit and intuitive.

While matrix multiplication is a fairly common programming example, allowing very regular data partition strategies we feel that our model provides adequate programming support for more complex applications, namely grid-based or hierarchical algorithms. In fact, the mappings used to construct the data partition in these systems may be easily adapted to produce an implementation for method p3divide. Also, the object-space may be used to store arbitrarily complex data-structures by assuming some adequate naming convention (e.g., directory style as in file-systems, or URL style as in web documents).

4 Conclusions and Future Work

We have described a development platform for high performance parallel computing in the Internet environment, based on recent and open research areas. Our belief is that the Internet and large, fast, intranets will, in the near future, be the *de facto* standard for high performance computing. Grid computing [11, 9] is an effort to take this idea even further, providing on demand computing power from high performance computers with fast interconnections.

Systems like HARNESS [12], Charlotte [3] and Javelin [6] also try to provide parallel programming environments for the Internet. However all these systems fall into the same computational and organizational model; they use the master/worker computation model and the client/server communication paradigm. P^3 is distinct for it uses none of these concepts; P^3 proposes a more intuitive and adaptive parallel programming model and uses Peer-to-Peer techniques to guard against common problems in those systems (server failures and connection bottlenecks), while providing a wide range of additional functionalities. The result should be a much more scalable, easier and richer parallel programming environment.

The P^3 run-time system is currently being implemented and major work is underway for creating a capable run-time system with all the features exposed in this document. Our concerns, at present, are in building a high performance, stable kernel with support for parallel computations according to the proposed model. The programming model was our first priority while kernel scalability, fault tolerance, high availability and object-system performance are now the most important concerns. Active research points are scalability issues and tolerance to volatile resources in extreme conditions.

In the future, P^3 will address topics such as security or dynamic system reconfiguration, but these are not short term objectives. Finally, profound tests will be conducted in three major areas: performance, scalability and fault-tolerance.

Acknowledgments. The authors are partially supported by FCT's projects MIMO and APRIL (contracts POSI/CHS/39789/2001 and POSI/SRI/40749/2001, respectively).

References

[1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. SuperWeb: Research issues in Java-based global computing. *Concurrency: Practice and Experience*, 9(6):535–553, 1997.

[2] J. Baldeschwieler, R. Blumofe, and E. Brewer. ATLAS: An infrastructure for global computing, 1996.

[3] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proc. of the 9th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-96)*, 1996.

[4] M. Beck, J. Dongarra, G. Fagg, G. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, and V. Sunderam. HARNESS: A next generation distributed virtual machine, 1999.

[5] A. Bricker, M. Litzkow, M. Livny, T. Summary, and V. Report. Condor Technical Summary, 1992.

[6] P. Cappello, B. Christiansen, M. Ionescu, M. Neary, K. Schauser, and D. Wu. JAVELIN: Internet based parallel computing using Java, 1997.

[7] *distributed.net*. `http://www.distributed.net/`.

[8] *Entropia*. `http://www.entropia.com/`.

[9] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[10] *Gnutella*. `http://www.gnutella.com/`.

[11] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, and Paul F. Reynolds Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, 8, 1994.

[12] M. Migliardi and V. Sunderam. Heterogeneous distributed virtual machines in the HARNESS metacomputing framework, 1999.

[13] M. Migliardi, V. Sunderam, A. Geist, and J. Dongarra. Dynamic reconfiguration and virtual machine management in the HARNESS metacomputing system, 1998.

[14] *Napster*. `http://www.napster.com/`.

[15] *Parabon*. `http://www.parabon.com/`.

[16] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, 2001.

[17] *SETI@Home*. `http://setiathome.ssl.berkeley.edu/`.

[18] D. Skillicorn and D. Talia. Models and languages for parallel computation, 1998.