Achieving Parallelism 'easily' through *Pshell* – Architecture and Overview

Daniel F. Saffioti

School of Information Technology and Computer Science (SITACS)
University of Wollongong

<dfs@cs.uow.edu.au>

ABSTRACT

Communications and the scheduling of tasks are the two most important issues of parallel programming on clusters. Over time, various parallel computing programming models such as remote threads, transparent process migration, message passing, distributed shared memory and optimizing parallel compilers have emerged, assisting the programmer develop applications, which can work seamlessly in such environments. Considerable research has been done to optimize these models, which typically have large communications overheads, resulting in a detrimental effect on performance. In addition to this, their acceptance has varied by virtue of the fact that each has introduced new problems with reference to portability, scalability and most importantly usability. Sometimes these problems completely violate the underlying notion of such computing.

To overcome these issues *Pshell*, which provides transparent scheduling and communication of jobs between disparate hosts, has been developed. *Pshell* is the 'glue' for producing high performance parallel applications, which can work securely, and efficiently in heterogeneous environments. It represents a major shift in traditional parallel programming environments because it is a language using the syntax of the Bourne Shell sh(1). The Bourne shell process and communications models can easily be extended to parallel computing environments using the concurrent programming model.

This paper will examine the evolution of cluster computing and identify where deficiencies lie in current programming models, providing a justification for simple languages whilst providing the reader with an understanding of the *Pshell* programming environment. In addition to this the paper illustrates how such a language can be used to ease the process of writing parallel applications and to overcome some of the limitations inherent in traditional programming models without scarifying performance.

1. Introduction to Cluster Computing & Programming Models

High Performance computing initially consisted of large machines in a single room. The machine would typically have a number of central processing units and a high-speed shared memory interconnect. Such machines included the Silicon Graphics Origin 2000 or Sun Enterprise 10000. Over time a number of configurations for such infrastructure have been seen. Most high performance computers of this type are costly and difficult to gain access to.

Beowulf clusters are the aggregation of commodity computers using network interconnects. The Beowulf cluster provides the user with the illusion of a single computer, even though it is composed of several or even thousands of nodes. This illusion is created through software. At NASA, Becker designed the original Beowulf cluster using 16, 80486 DX CPU's connected to one another using a network switch (Becker, Merkey, Ridge, Sterling, 1997). This infrastructure sustained performance of 74Mflops. Later incarnations of such clusters were based on the same concepts but used Pentium class CPU's. These clusters yielded performance in excess of 2.4Gflops. The Beowulf cluster suddenly made it feasible to address computationally expensive problems in an environment that was economically viable for many organizations.

First generation clusters were clumsy. Ferri and Otero describe Beowulf cluster computing as initially having a number of stumbling blocks (Ferri, Otero 2002). These authors focussed on issues of scalability, software support for heterogenous nodes and the costs associated with the configuring of individual nodes. More importantly however, first generation Beowulf clusters did not provide the user with the impression that they were using a single computer. Such clusters were clumsy both to use and to administer.

Second generation Beowulf clusters emerged during the late 1990's. They retained the same underlying communications infrastructure as their predecessors but offered a number of advances in management tools and process space distribution. Management tools such as *OSCAR* emerged. *OSCAR* is a 'software stack', which contains all the software necessary to configure and manage clusters (Ferri 2002). The *OSCAR* software stack includes an operating system (typically open source), administration tools, job scheduling software such as *PBS* (Portable Batch Scheduler) (Ferri 2002) and conventional multi-host programming libraries such as *PVM* (Parallel Virtual Machine). To date *OSCAR* remains popular with cluster administrators and users. It represents a major turning point in the development of such technology.

Other work undertaken in the development of second-generation cluster environments included the introduction of distributed process spaces, to make the collection of nodes appear to be a single node. One notable example of this is *Bproc*, which permits the transparent migration of processes to hosts in the cluster (Hendricks, 2002). The original concept of a cluster described by Becker is 'a collection of commodity computers connected to one another' (Becker, Merkey, Ridge, Sterling 1997). There is no implication of uniformity in this definition and it therefore allows that computers within such a cluster might be of different architectures, possibly running different operating systems. It could be argued that second generation cluster tools do not abide by this idea entirely, largely due to the fact that process migration is architecture and platform dependant.

Communications management and the efficient scheduling of processes are the two most fundamental issues in cluster computing. A number of solutions have emerged allowing processes to be migrated between disparate hosts. Solutions like *Open Mosix* (Barak, La'Adan 1998) and *Condor* (Douglas, Milojicic, Paindaveinem Wheeler, Zhou 2000) provide varying levels of transparency and portability. That said, a number of programming models have emerged over time, which make it easier for the user to develop parallel applications. These models focus on the provision of appropriate infrastructures, which enable programmers to create remote processes and facilitate communications between them. Additionally, some programming models provide primitives to enable concurrency and serialisation of tasks.

The major programming models include message passing implemented by libraries such as *PVM* (Parallel Virtual Machine (Sunderam 1990) or *MPI* (Message Passing Interface) (The MPI Forum, 1993). Message passing techniques allow the creation of remote processes in clusters and provide infrastructure to facilitate communications between them. Sunderam initially proposed *PVM* in the late 1980's and it remains the most accepted and widely used of the available tools (Sunderam 1990). Other programming models include distributed shared memory (DSM) regimes, which allow the transparent sharing of memory regions between nodes in a cluster. A number of packages including *Ivy* (Li, Huddak, 1989), *Mirage* (Fleish, Popek 1989), *Quarks* (Swanson, Stoller 1998) and *Agora* (Bisiani, Forin 1998) have implemented varying distributed shared memory algorithms. One of the largest problems inherent in any of these programming models is the latency associated with the network interconnects in cluster environments and the additional overheads in the protocols involved with process delegation and the sharing of data.

The loose coupling of the nodes, along with the overheads imposed by interconnects, causes programming paradigms based on process migration, message passing or shared memory regimes to introduce extra complexities which do not benefit the end user. In some programming models, the programmer needs to articulate in great detail the communications between the processes – the programmer needs to be mindful of issues such as deadlock, data consistency and concurrency. Many programmers simply do not have the experience or background to do this effectively.

Whether it be message passing or distributed shared memory, most of these accepted programming models for clusters are 'add ons' to pre-existing languages. This can be viewed as a negative aspect for programming because the syntax of the underlying languages do not support parallel concepts. This, in turn, requires greater effort on the part of the programmer. Lastly, most of the existing programming models, place additional costs on development time. More often than not, a programmers needs to write a parallel program while keeping in mind the entry points to a communications or process library. This can be a cumbersome and daunting experience.

'Beowulf economics and sociology are poised to kill off the other architectural lines – and will likely effect traditional super computer centres' (Bell, Gray 2002). This indicates the growth and acceptance of cluster computing environments and the major challenge they present to traditional high performance computing. An examination of the top five super computers in

the world illustrates how such computing is proving popular and successful. Two of these five super computers defies the norm, being a Beowulf cluster made up of many dual processor nodes. (Top 500 Super Computers, 2003), these clusters are located at Virginia Tech and at MCR Livermore. In order for this trend to continue, new techniques must be devised which allow users from diverse backgrounds to access the resources provided by cluster computing. This is a core goal of grid computing environments.

The following paper examines a number of the currently accepted programming models and illustrates how a scripting language for cluster environments, based on a concurrent programming model, can be used to ease the process of writing parallel applications and to overcome some of the limitations inherent in these traditional environments.

2. The Pshell Programming Environment

Our current research is into the design of a scripting/programming language called *Pshell*. The language is based on a concurrent programming language model, which allows programs to be expressed as a series of interleaved processes. Each process contributes to an overall solution of a problem (Harrison 2002). Hoare devised this original programming model concept in his Communicating Sequential Processes paper (Hoare 1978).

The *Pshell* language is also based on the concepts presented in the P^3L language. P^3L is a high performance language initially designed for transputers, which utilises a series of communicating sequential processes (Danelutto, Di Meglio, Orlando, Pelagatti, Vanneschi 1992). The processes have clearly defined inputs and outputs, which can be derived or used by other processes. Processes execute concurrently, using built in constructs such as farms and pipelines. This, in turn, provides a simple programming model that does not require the programmer to articulate all communication flows in the system.

Harrison has demonstrated the success of such languages in modelling problems. His 'Initial Concurrent Programming Language' illustrates how problems can be decomposed and expressed as a series of communicating processes. He argues that this is an excellent teaching language as it allows users to clearly articulate data flows, which form the basis of functions (Harrison 2002). The concurrent language paradigm is suitable for the resolution of many of the issues in parallel programming. The problems associated with parallel programming are largely due to communications and the distribution of processes. The underlying nature of the concurrent programming model is believed to simplify the process of writing parallel code by eliminating the need to express every aspect of process scheduling and communications.

2.1 Rationale behind the development of the *Pshell* Programming Model

The design of the *Pshell* programming language differs from that of a number of other programming languages. This is largely due to its design and syntax. *Pshell* permits the execution of *Pshell* code on heterogenous hosts using syntax similar to that of the Bourne shell. The beauty of the language lies in its tying together of ideas which have existed separately in the parallel computing and operating system fields for some time. An example of this includes distributed shared memory.

Current programming models for cluster environments include message passing, shared memory and process migration techniques. *Pshell* specifically aims to overcome many of the limitations of current programming models for clusters.

a) Complexities in the software development process; in other words, the linking of shared objects as required by libraries such as *PVM*, *MPI* and *Quarks*. This takes away precious time from the programmer, which can be better spent writing code.

b) The complexities in expressing communications and interactions within programs. For example in environments such as *PVM* and *MPI* a programmer needs to clearly articulate the data flows between different threads of execution. In distributed shared memory environments the programs need to express how data integrity is maintained. These environments typically provide a number of low-level primitives to control the flow of data.

c) The homogenous nature of such programming models conflicts with the ideas central to the Beowulf cluster concept. Programming models such as *PVM*, *MPI* and *DSM* all function best in clusters of homogenous nodes. Their performance and usefulness varies in heterogenous environments but, typically, degrades to some degree. This is an unacceptable feature for clusters and future grid computing environments as they tend to be made of

computational elements of *different* architectures and capabilities.

Many distributed shared memory regimes simply do not work in heterogenous computing environments due to differing page sizes or kernel implementations (Carter, Khandekar, Kamb 1995). To further illustrate this point, a number of the process migration technologies in current widespread use do not scale well to heterogenous clusters. This is due to the underlying requirements in operating system kernels. *Open Mosix* is an excellent example of a process migration technology, which requires a 'patch' to the Linux kernel.

d) The lack of scalability and reliability in such models is an issue which needs to be resolved.

   For example the programming model offered by *Open Mosix* tends not to scale with large clusters. Protocols used by this technology have high communications overheads adversely effecting scalability, whist the pre-emptive process model (used for load balancing) affects portability (Hendriks 2002). If a process in Open Mosix terminates abnormally there is no mechanism for it to be restarted – this is a serious reliability issue. These issues are particularly important as cluster computing begins to evolve into grid computing.

e) Environments such as *PVM*, *MPI* and many of the shared memory systems are unaware of the computing environment in which they function. It could be argued that such environments are generally not adaptable to their environment. They are typically unaware of latencies in network connections and the utilisation of resources. This does not allow the best usage of resources in a cluster environment. In such programming models, processes can be allocated to resources which are busy or have experienced degraded performance. There have been several attempts to optimise *PVM*'s performance by making it aware of the surrounding environment (Hendriske, Iskra, Overreinder, Sloot, Van Albada, Van Dan Lindern 2000).

f) There are many legacy applications for which source code no longer exists. Often it is important for such legacy applications to be integrated into parallel programming environments. Few programming models currently support the integration of legacy applications in any way.

2.2 *Pshell* Architecture and Implementation.

Pshell is an interpreted language, which uses an interpreter similar to that utilised by the Bourne shell. This interpreter allows the distribution of processes as expressed by a programmer using conventional shell constructs such as pipes and redirection. The familiar shell language has been extended enabling a programmer to articulate areas of code which can be parallelised.

Pshell acts as the glue for combining executables and shell-like code to form parallel applications. The execution of *Pshell* code and the processes it controls can be distributed over a cluster of computational elements using the models offered by concurrent programming languages. The *Pshell* interpreter is responsible for identifying which processes can be executed on a given machine (as such execution is platform/architecture dependant). It is also responsible for parsing *Pshell* script code and identifying fragments which can be distributed and executed in parallel. Parallelisation of code can be explicitly expressed by the programmer or inferred by analysing the consequences of operators such as pipelines.

Additionally *Pshell* enables the sharing of data using a distributed shared memory model similar to that used in *Agora* (Pinherio, Chen, Dwarkadas, 2000). Data can be shared between distributed *Pshell* processes (scripts). The data-sharing model is typeless, therefore enabling the programming environment to function on differing architectures. It is even possible for executables to inherit variable values.

The following core components of the *Pshell* programming model and their interactions between one another are illustrated in Figure 1.

a) Pshell Virtual Machine (Sand Box).
   The virtual machine is responsible for the execution of instructions as specified by the interpreter. The virtual machine permits *Pshell* code to be executed on any architecture/platform. This therefore enables the environment to function in heterogenous computing environments. In addition to this the virtual machine provides a mechanism similar to chroot(3c) which prevents users from accessing resources beyond the Virtual Machine. *Pshell* interpreter instructions are executed on this virtual machine. In addition to this the *Pshell* Virtual Machines keeps vital performance information about a host e.g. CPU load, I/O load and Network load. This information is stored in internal data structures and can be accessed by other *Pshell* constructs through a number of IPC

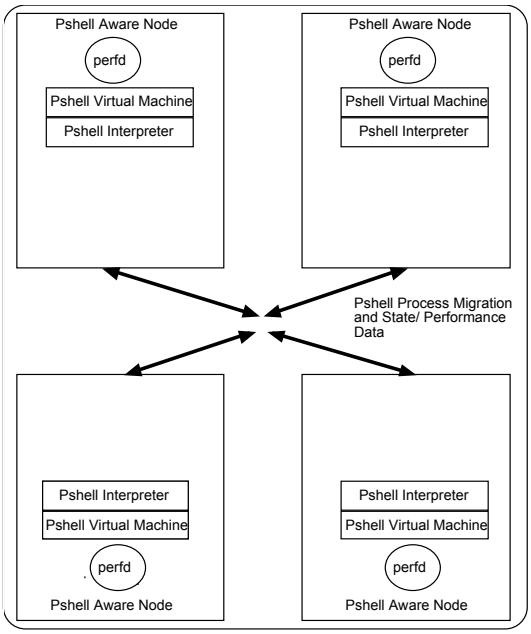

Figure 1: A typical cluster illustrating Pshell's architecture.

facilities e.g. Sockets and Shared Memory.
In addition to doing this, the *Pshell* Virtual Machine provides an I/O subsystem to communicate with spawned processes by mimicking conventional standard input, output and error.

b) *The Pshell Engine – Bourne Shell Style Interpreter.*
The *Pshell* interpreter is responsible for interpreting *Pshell* script code written by the programmer and delegates it to appropriate hosts at runtime. *Pshell* script code contains calls to executables written using a number of languages and possibly for different platforms. The *Pshell* scripting language provides shell-like semantics to 'glue' together executables and shell-like code. The *Pshell* interpreter spawns processes on remote hosts by using a remote method invocation technique.

A process would be spawned on remote hosts when pipelines, calls to executables or explicit parallelisation requests are made in the *Pshell* script. A process can be either a running *Pshell* script or an executable built for a particular target architecture.

The delegation of code/process to remote hosts is influenced by the semantics of the language and factors such as CPU utilisation, network latencies and executable binary format e.g. ELF executables could only be executed on a Sparc based machine. If the interpreter is preparing to execute a process, the interpreter identifies an appropriate host to handle it by issuing a UDP broadcast or multicast which forces all hosts to respond with state information managed by the Virtual Machine. The techniques employed permit dynamic resource discovery allowing the cluster to grow and contract in size during runtime. All this ensures that the programming environment is aware of the state of the cluster, which may influence the performance of a task. There has been considerable work in optimising *PVM* for this very purpose (Hendriske Z, Iskra K, Overreinder J, Sloot P, Van Albada G, Van Der Linden F, 2000)

It should be noted that the *Pshell* Interpreter and Virtual Machine are one component of the system. Each node in the cluster would be running the *Pshell* interpreter and Virtual Machine. A *Pshell* script would typically be executed on one host and, over the course of the program's execution, other hosts would be called upon to perform discrete tasks. A Parent-Child relationship drives the process model implemented by *Pshell*. It is similar to that used in all Unix shells.

c) *Pshell* Language Constructs and Infrastructure.

The ideas presented by Hoare (Hoare, 1979) and Harrison (Harrison 2002) form the basis of the *Pshell* language. The language uses constructs similar to the Bourne shell to model process delegation and communications. *Pshell* code consists of a number of scripted processes working together. The language permits the gluing together of these constructs and enables the sharing of data between them. In the *Pshell* environment executables have clearly defined inputs and outputs, just as in concurrent programming languages. Functions implemented in the shell can also be delegated to remote hosts. An example of the *Pshell* language can be seen in Figure 2.

d) *Pshell* Performance Daemon.

The *Pshell* performance daemon provides real time information to the nodes in a cluster about the status of each node. This information is communicated back to the virtual machine, which use it to delegate jobs. During the runtime process *Pshell* Scripts will query hosts for load – this information is derived from conventional network broadcasts. This in turn allows the cluster to adapt process scheduling in accordance to how resources such as CPU, Disk and network are being utilised.

The driving force behind the *Pshell* programming model is to provide a highly portable, scalable and easy to use programming

```
#
# In for loops,list can be a series of values (variables),
# calls to Pshell functions or other executables. Each
# iteration of the loop would be distributed to nodes in t
# cluster NOTE: No dependant variables are used in the loo
#

for i in list
do parallel
      statement
      statement
      . . .
done

#
# It should be noted that in the above construct the
# distributed processes inherit variables using the
# semantics of the sh(1). Executables called can see such
# variables but not alter them. Distributed Pshell script
# code can modify inherited variables.
#
# parallelisation of pipelines results in the distributio
# the processes in the pipline. Pipelines in the pshell
# language may invoke functions created by the programmer
# Typically the results of a pipeline would be assigned t
# variable as illustrated below.
#

A=`process a | process b | process b | pshell func`

#
# Other operators such as <, > and & have special
# meanings in the Pshell environment. For exampe & means
# to spawn a background process and detach it. It may
# be allocated to any node in the cluster. The parent can
# continue execution.
```

Figure 2: Basic Syntax of Pshell

environment. By developing a programming model, which utilises a familiar syntactical base, process, and communications models, it is believed simplicity can be achieved in cluster programming without sacrificing portability and efficiency. The nature of the programming environment basically requires a programmer to associate tasks with processes and express them in a way that could be executed in parallel. The programmer need not worry about the intricacies of communicating information back and forth. This represents a major improvement on some of the existing DSM or message passing technologies.

3. A Sample Application

Consider the example of ray tracing. In order to operate, a ray tracer requires information about all objects in the image to be rendered; geometry data and information about light sources. Decomposing the final resultant image into blocks can parallelize this task. The generation of these blocks can be performed concurrently.

A user may write an executable called raytracer that takes command line arguments indicating the dimensions of the image and quadrant number. The program produces a data file consisting of part of the final ray traced image. When the user prepares to perform such a task not only do they provide the executable, geometric and light source data (geometric.dat) but they also provide a script which explains how to execute the program and with what input data. This allows the user to achieve a degree of coarse-grained parallelism. This script is written using *Pshell*. *Pshell* as described earlier provides a real time distributed shell environment. Additionally, other executables can be provided to do pre/post processing. In this example another application called merge is used to combine all the resultant images to produce the final rendered image (post processing). Thus we end up with *Pshell* script like this;

```
#!/bin/psh

for quad in 1 2 3 4
do parallel
   cat geometric.dat | raytracer \
         > output.${quad}
done
merge output.* > finalimage
```

In this particular example, the for loop is iterated four times. The environment runs all iterations of the loop in parallel as expressed by the parallel keyword on remote nodes.

The remote nodes are allocated the process raytracer with the data and evaluated arguments (in this case representing the iteration number or image segment to generate). Each process is allocated using the scheduling algorithms described previously, choosing the node with the lowest CPU and I/O loads. When the remote nodes finish processing, control is passed back, to the source to combine the results using the provided merge program. The merge program takes the output of each ray tracer. The resultant image file, finalimage, generated by merge is the result.

Any program in general can have one or more arguments passed to it. This argument processing technique has become common in a number of other grid/cluster environments such as Nimrod-G (Giddy, Abranson, Buyya, 2000) and Apple's X-Grid (Apple Computer Inc, 2003) to achieve parallelism. As you can see *Pshell* hides the complexity of parallel programming by allowing users to combine 'smaller building' blocks to produce a complete solution. The idea of joining small programs via input/output to create larger programs is a core feature of 'Art of Unix Programming' (Raymond, 2003).

4. Limitations

One of the potential issues with the *Pshell* programming environment is the lack of support for floating point arithmetic. Shell scripting languages such as the Bourne-shell and C-shell do not support floating-point arithmetic. Because *Pshell* is based on these languages, this limitation is present here. It may be necessary to write *Pshell* scripts that require the use of

floating point arithmetic; in which case future work on *Pshell* will be required.

Another limitation of the *Pshell* programming environment is the method in which it interprets the file system space. In this environment, where the nodes are loosely coupled, the interpretation of filenames is node-centric – this therefore means there may be inconsistencies between the underlying layout of the file system and the user's perception. It may be necessary therefore for nodes to have access to the same files – providing a consistent view of the overall file system. It may be unacceptable for the *Pshell* interpreter or virtual machine to have access to other parts of the node's file system. Future incarnations of *Pshell* need to address the need for a consistent and possibly insulated file system. A possible solution would be the development of a sandbox environment, which allows *Pshell*-aware nodes to share data and resources transparently. This would require the layering of a network file system into the *Pshell* programming environment.

Future work on *Pshell* will focus on comparing it against the conventional message passing, process migration and shared memory regimes. Testing would be focused on micro and application level benchmarking (Akinlar, Gueve, Hollingsworth, 2003). Measuring the efficiency of the *Pshell* communications protocols and establishing the types of problems best suited to this programming model drives our research. We are currently exploring techniques, which will help in quantifying the environment's performance on particular problems.

It is often assumed that parallel computing programming paradigms could readily be migrated to grid environments. The assumption being that a grid is essentially a distributed cluster (Berman, Geoffrey 2003). In reality, this has proven to be difficult. Grids tend to be dynamic computing environments with heterogenous nodes. Latencies in interconnects, processing power variations, along with ownership and security issues introduces a new realm of problems (Baker, De Rouke, Jennings 2003). To date, grids have been used as a platform to integrate loosely coupled applications or resources. Just like cluster computing, the coordination and distribution of processes is of great importance. However, in this case, the underlying programming models simply do not scale. This means that the users of such environments need to learn new techniques and processes to leverage services. It is our belief that the *Pshell* programming model may fit nicely into grid environments, because it provides a *platform-neutral* programming model.

Grid-computing environments can be described as having a number of traits analogous with the power grid. Grids, just like the power grid, should be dependable, cost effective, consistent and have pervasive infrastructures (Foster, Kesselman 1999). This analogy lies at the core of grid technologies, which are designed to provide easy access to a wealth of resources while hiding the underlying complexity from the user. Currently, grids exhibit complexities in process distribution and communications due to the dynamic nature of resources, ownership, security requirements and network latencies. The *Pshell* programming model may a viable programming environment for grids by virtue of its simplicity and elegance. For this to be done, the *Pshell* environment needs to be made 'grid aware', possibly using an existing framework such as Globus (Globus 2003).

5. Conclusion

Conventional high performance computing models, such as super computers, offer tightly integrated hardware and software solutions. The move to cluster and grid computing presents a hardware and software solution which is loosely coupled. In order to utilise these infrastructures, elegant programming models must be devised and used. In cluster environments, software ensures the coupling of system resources. The lack of integration typically leads to reduced performance (Foster, Kesselman 1999) and more complex models for programming.

The illusion of the cluster as a single machine is created by a number of programming models, which provide programmers with the appropriate primitives and techniques to create and distribute tasks. In addition to this, these models provide an infrastructure to share information. Whether it is writing a parallel application utilising *Quarks* or *PVM* the programmer needs to have a deep understanding of the problem and its interactions with data.

The application of some of the current software models for cluster computing incur additional overheads in terms of development time whilst others produce programs which have considerable latencies, by virtue of their lack of understanding of the surrounding environment and high communications overheads. Clusters are heterogenous computing environments and even though a number of programming techniques exist for them some of them simply do not appreciate the potential diversity of infrastructure.

In this paper, the *Pshell* scripting language was discussed. *Pshell* provides a framework for 'gluing' together executables and shell-like code. The *Pshell* language provides the ability to communicate with and delegate code to remote hosts. *Pshell* provides the programmer with a simple syntax, based on the Bourne shell. The nature of the shell programming language is one which is suitable for distributed environments, because it is based on processes and inter-process communication (IPC). These two aspects form the core of any cluster program model. The language allows parallel programs to be constructed by using ideas from concurrent programming languages such as *ICPL* and P^3L.

Pshell attempts to overcome many of the limitations identified in current programming models without scarifying performance. Our results show that the performance of Pshell is comparable to that of other tools such as PVM-3. In addition to this our results to date suggest that Pshell performs better then PVM in environments where the behaviour of nodes varies and is dynamic. This performance improvement in Pshell is partially due to the smarts the interpreter posses. Pshell is a flexible, easy to use, highly scalable programming model which eliminates the need for explaining complex data interactions in process-level code. In addition to this, it removes the need for compilation, allows legacy code to be integrated into parallel applications and creates a highly adaptable environment. Future work will focus on further development of the *Pshell* programming model and identifying which problems and environments it is best suited to. Our belief is that this model will provide researchers with the power to easily write code which leverages the power of computational resources be they are clusters or grids. It is also our belief that such a tool may be used in tertiary education to teach the fundamentals of parallel programming.

References

AKINLAR C, GUVEN E, HOLLINGSWORTH (2003): *Benchmarking a Network of PC's Running Parallel Applications* Workshop in IPCCC98, http://www.cs.umd.edu/~hollings/talks/ipccc98/ipccc98.pdf. Accessed 1 May 2004.

APPLE COMPUTER INC, *Apple Xgrid*, (Online) http://www.apple.com/acg. Accessed 30 May 2004.

BAKER M, DE ROUKE D, JENNINGS R (2003), *The evolution of the Grid* in "Grid Computing: Making the global infrastructure a reality', New York, Wiley Press.

BARAK A, LA'ADAN O (1998): *The MOSIX Multi Computer Operating System for High Performance Cluster Computing*. In: *Journal of Future Generation Computer Systems*, 4(5):361 – 372. ACM Press.

BECKER D, MERKEY P, RIDGE D, STERLING T (1997): *Beowulf: Harnessing the Power of Parallelism in a Pile of PC's* In: *Proceedings, IEEE Aerospace*. 1 – 13.

BELL G, GRAY J (2002): *What's next in High Performance Computing*. In: *Communications of the ACM*, 45(2):91 – 95. ACM Press.

BERMAN F, GEOFFREY F, HEY T (2003): *The Grid: Past, Present and Future* in: *Grid Computing: Making the global infrastructure a reality,* New York. Wiley Press.

BISIANI R, FORIN A (1998): *Multilanguage Parallel Programming on Heterogenous Machines,* IEEETC 37(8):930 – 945. IEEE Press.

BAKER M, Buyya R, LAFORENZA D, *Internet-Wide Global Supercomputing* in: *Australian Unix Users Group 2000 Conference (Enterprise Security, Enterprise Linux)* in proceedings pp 229–251, Canberra, June 2000.

CARTER J, KHANDEKAR D, KAMB L (1995): *Distributed Shared memory: Where we are and Where we Should be Headed.* In: *Proceedings of the 5th Workshop on Hot Topics in Operating Systems, IEEE, 5:*119 – 122. IEEE Press.

DANELUTTO M, DI MEGLIO R, ORLANDO S, PELAGATTI S, VANNESCHI M (1992): *A Methodology for the Development and Support of Massively Parallel Programs* In: *Future Generation Computer Systems*, 8:205 – 220.

DOUGLAS F, MILOJICIC D, PAINDAVEINE Y, WHEELER R, ZHOU S (2000): *Process Migration* In: *ACM Computing Surveys*, 32(3):241 – 299. ACM Press.

FERRI R, OTERO G (2002): *The Beowulf Evolution*, in *Linux Journal*, 2002:100. IEEE Press.

FERRI R (2002): *The OSCAR Revolution*, in *Linux Journal*, 2002:98. IEEE Press.

FLEISH B, POPEK G (1989): *Mirage: a Coherent Distributed Shared Memory Design*. In: *Proceeding of the twelfth ACM symposium on Operating System Principles,* New York, 12:213-223. ACM Press.

FOSTER I, KESSELMAN C (1999): *Computational Grids* in *The Grid: Blueprint for a new computing infrastructure*, Morgan Kaufman, San Francisco.

Globus: The Globus Framework, Globus Software Foundation, `http://www.globus.org`. Accessed 1 May 2004

HARRISON, C (2002): *ICPL: An initial Concurrent Programming Language* In: *SIGCSE Bulletin* 34(4):101 – 105. ACM Press.

HENDRIKIS E (2002): *Bproc: The Beowulf Distributed Process Space* In: *Proceedings of the 16th International Conference on Supercomputing*, New York, 16:129 – 136. IEEE Press.

HENDRISKE Z, ISKRA K, OVERREINDER J, SLOOT P, VAN ALBADA G, Van Der Linden F (2000): *Implementation of Dynamite – an Environment for Migrating PVM Tasks*, In: *ACM SIGOPS Operating System Review*, 34(3):40 - 55. ACM Press.

HOARE C (1978*): Communicating Sequential Processes*, In: *Communications of the ACM* 21(8):666 – 667. ACM Press.

LI K, HUDDAK P (1989): *Memory Coherence in Shared Virtual Memory Systems*, In: *ACM Transactions on Computer Systems*, 321 – 359. ACM Press.

PINHERIO E, CHEN D, DWARKADAS S (2000): *S-DSM for Heterogenous Machine Architectures* In: *Second Workshop on Software Distributed Shared Memory,* United States, 1 – 7. IEEE Press.

RAYMOND E, "*The Art of Unix Programming*, Addison Wesley Press, 24 – 27, Boston, 2003.

SUNDERAM VS (1990): 'PVM: A Framework for Parallel Distributed Computing' In: *Concurrency Practice and Experience,* 2(4):315 – 339, ACM Press.

SWANSON M, STOLLER L, CARTER J (1998): 'Making Distributed Shared Memory Simple, Yet Efficient'. In: *Proceedings of the 3rd International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Washington, United States, 2 – 13.

THE MPI FORUM: (1993) MPI: *A Message Passing Interface*, In: *Proceedings of 1993 ACM/IEEE Conference on Supercomputing*, Portland, United States. 878 – 883, ACM Press.

Top 500: Top 500 Super Computer Sites, http://www.top500.org. Accessed 1 May 2004