ORNL/CP-99933

Invited paper at the EuroPVM/MPI 98 Conference, Liverpool, UK, September 6, 1998

HARNESS: THE NEXT GENERATION BEYOND PVM*

G. A. Geist
Computer Science and Mathematics Division
Oak Ridge National Laboratory
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

*Research supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U. S. Department of Energy, under contract No. DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation.

Harness: The Next Generation Beyond PVM

G. A. Geist *

Oak Ridge National Laboratory, USA

Abstract. Harness is the next generation heterogeneous distributed computing package being developed by the PVM team at Oak Ridge National Laboratory, University of Tennessee, and Emory University. This paper describes the changing trends in cluster computing and how Harness is being designed to address the future needs of PVM and MPI application developers. Harness (which will support both PVM and MPI) will allow users to dynamically customize, adapt, and extend a virtual machine's features to more closely match the needs of their application and to optimize for the underlying computer resources. This paper will describe the architecture and core services of this new virtual machine paradigm, our progress on this project, and our experiences with early prototypes of Harness.

1 Introduction

Distributed computing has been popularized by the availability of powerful PCs and software to cluster them together. The growth of the Web is also raising the awareness of people to the concepts of distributed information and computing. While originally focused on problems in academia and scientific research, distributed computing is now expanding into the fields of business, collaboration, commerce, medicine, entertainment and education. Yet the capabilities of distributed computing have not kept up with the demands of users and application developers. Although distributed computing frameworks such as PVM [2] continue to be expanded and improved, the growing need in terms of functionality, paradigms, and performance quite simply increases faster than the pace of these improvements.

Numerous factors contribute to this situation. One crucial aspect is the lag between development and deployment times for innovative concepts. Because systems and frameworks tend to be constructed in monolithic fashion, technology and application requirements have often changed by the time a system's infrastructure is developed and implemented. Further, incremental changes to subsystems often cannot be made without affecting other parts of the framework. Plug-ins offer one solution to this problem. Plug-ins are code modules that literally plug into a computing framework to add capabilities that previously did not exist. This concept has become popularized by Web browsers. For

* This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation

example, there are Netscape plug-ins for playing live audio files, displaying PDF files, VRML and movies. The central theme of Harness is to adapt this plug-in concept and extend it into the realm of parallel distributed computing.

By developing a distributed computing framework that supports plug-ins, it will be possible to extend or modify the capabilities available to parallel applications without requiring immediate changes in the standards, or endless iterations of ever-larger software packages. For example, a distributed virtual machine could plug in modules for distributed shared memory programming support along with message passing support, allowing two legacy applications, each written in their own paradigm, to interoperate in the same virtual machine. Virtual machine plug-ins enable many capabilities, such as adapting to new advanced communication protocols or networks, programming models, resource management algorithms, encryption or compression methods, and auxiliary tools, without the need for extensive re-engineering of the computing framework. In addition, Harness plug-ins will be dynamic in nature, or "hot-pluggable". Certain features or functionality will plug in temporarily, only while needed by an application, and then unplug to free up system resources. Distributed application developers no longer will need to adjust their applications to fit the capabilities of the distributed computing environment. The environment will be able to dynamically adapt to the needs of the application on-the-fly.

In Harness we have broadened this concept of pluggability to encompass the merging and splitting of multiple virtual machines that was pioneered in IceT [5], as well as the attachment of tools and applications [6]. Being able to temporarily merge two virtual machines allows collaborating groups at different sites to work together without feeling like they have to place their resources into some larger pool that anyone might use.

Looking at the new capabilities of Harness from the software and application's viewpoint, analysis tools will be able to plug into applications on-the-fly to collect information or steer computations. In addition, peer applications will be able to "dock" with each other to exchange intermediate results or even active objects (e.g. Java bytecode) thereby facilitating collaborative computing at the software level.

From the hardware perspective, as more and more high-performance commodity processors find their way into scientific computing, there is a need for distributed applications to interact between Window and UNIX systems. All of our Harness research is being done within a heterogeneous environment that can include a mixture of Unix and NT hosts; in fact, the plug-in methodology will permit greater flexibility and adaptability in this regard.

The rest of this paper describes the work of the Harness team, which includes researchers at Emory University, Oak Ridge National Laboratory, and the University of Tennessee. The work leverages heavely on other distributed computing projects the team is concurrently working on, particularly Snipe [1] and IceT [5]. In the next section we describe the basic architecture, then follow with a description of the design of the Harness daemon. In section 4, we describe the core services that are provided by the Harness library upon which the Harness daemon is built. Finally, we present the latest results and future development plans.

2 Harness Architecture

The Harness architecture is built on the concept of a distributed set of daemons making up a single distributed virtual machine (DVM). The high level architectural view is similar to the very successful PVM model where there is one daemon per IP address within a given DVM. Where Harness deviates sharply from PVM is in the composition of the daemon and the control structure.

The Harness daemon is composed of a minimal kernel, which understands little more than how to plug-in other modules, and a required set of plug-in modules, which supply the basic virtual machine features. The required modules are: communication, process control, and resource management. The design also allows the user to extend the feature set of their DVM by plugging in additional modules written by the user community.

Figure 1 shows how the plug-ins and daemons fit into the overall Harness architecture.

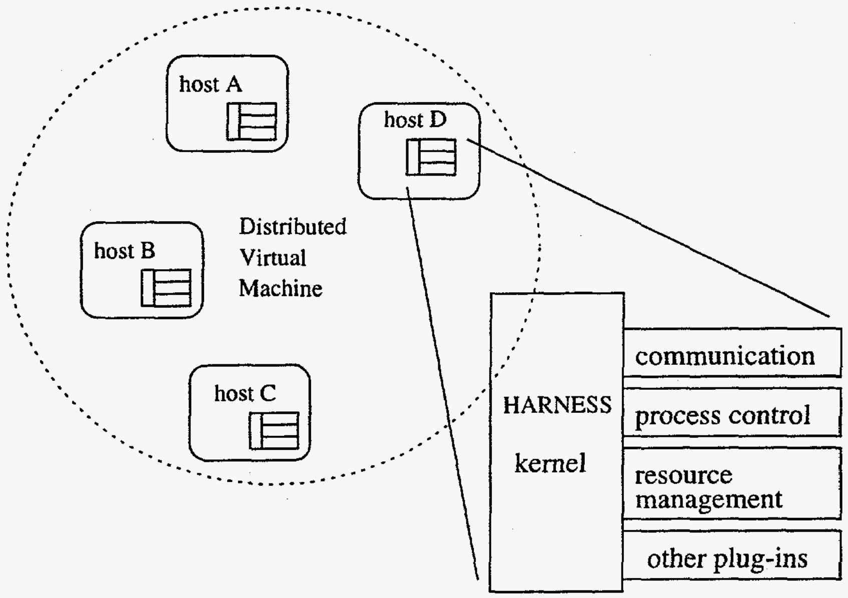

Fig. 1. Distributed Virtual machine is composed of daemons running on each computer and each daemon is composed of 3 required modules (plus possibly others)

2.1 Distributed Control

One goal of Harness is to be much more adaptable and robust than PVM. One weakness in PVM is that it has a single point of failure. When PVM first starts up it selects one daemon to be the "master daemon" responsible for arbitrating conflicting requests and race conditions. All other hosts and daemons in PVM can come and go dynamically, but if contact with the master daemon ever fails the entire DVM gracefully shuts down. We have designed Harness without a single or even multiple point of failure short of all the Harness daemons failing at the same time. This flexibility is not just a nice new feature, but is necessary when you consider the dynamic plug-in environment with constantly changing available resources that Harness is designed to work in.

Two distributed control designs are being investigated for the Harness project. The first is based on multicast and the dynamic reconstruction of the DVM state from remaining daemons in the case of failure. This design is described in [7]. The second design is based on all daemons being peers with full knowledge of the DVM state. The second design is described in detail in [4].

Both designs meet the following requirements that we have specified for the Harness control structure.

- Changes to the DVM state are seen in the same order by all members.
- All members can inject change requests at the same time.
- No daemon is more important than any other i.e. no single point of failure.
- Members can be added or deleted fast.
- Failure of host does not negate any partially committed changes, no rollback.
- Parallel recovery from multi-host failure.

3 Harness Daemon

As seen in the above description, the Harness daemon is the fundamental structure of the Harness virtual machine. The cooperating daemons provide the services needed to use a cluster of heterogeneous computers as a single distributed memory machine.

At its most fundamental level the Harness daemon is an event driven program that receives and processes requests from user tasks or remote daemons. Processing a request may include sending requests to other Harness daemons or activating user defined plug-ins. Here is the outer loop of the Harness daemon.

```
loop till shutdown
    recv request
    validate request
    carry out request
    reply to the requester
endloop
```

The first step after a request is received is to check that the requester is authorized to execute this service on the specified resource. Since PVM was a single user environment this was an unnecessary step, but since Harness allows multiple DVMs to merge, there can be multiple users sharing a DVM. Harness must provide some measure of authorization and protection in its heterogeneous distributed environment, just as an operating system supplies in a multiple user parallel computer. The act of registering with a DVM will return a certificate that will be used to specify the scope of permissions given to a user.

There are three classes of requests that the daemon understands. The first class of request is to perform a local service. The daemon checks its table of available services which includes the core services such as "load a new plug-in", required plug-in services such as spawn a task, and user plug-in services such as balance the load. If available, and the user is authorized, the requested service is executed. Otherwise an error message is returned to the requester. The second class of request is to handle distributed control updates. There are steady pulses of information constantly being exchanged between daemons so that changes to the state of the DVM are consistently made. Each daemon's contribution to the arbitration of state changes is passed around in the form of requests. For example, a request may come in that says, "daemon-X wants to add host-Y to the DVM." The local daemon can pass it on as a way of saying, "OK" or the daemon may know that host-Z needs to be added to the DVM state first and reacts accordingly. (Exactly how it reacts depends on which of the two distributed control methods is being used.) The third class of request is to forward a request to a remote daemon. For example, a local task may send a request to the local daemon to spawn a new task on a remote host. The local daemon could forward the request to the daemon on this host rather than try to invoke a remote procedure call.

4 Core Services

A key realization is that the set of daemons can be thought of as just another parallel application, not unlike a user's parallel application. The daemons need to send information back and forth, they need to be able to coordinate, and to keep track of the state of the virtual machine. These are the same kinds of needs every parallel program has. Therefore the Harness approach is to design a library of functions required to build the daemons and to make this library available to application developers. This library is called the core services and can be divided into four areas: plug-in/run interface, data transfer, global storage, and user registration.

4.1 Plug-in/Run Interface

The basic concept behind Harness is that pluggability exists at every level. At the lowest level, a Harness daemon or a user application can plug-in a new feature, library, or method. At the middle level, two parallel applications can plug-in to one another and form a larger multiphase application. At the highest level, two virtual machines can merge (plug-in) to each other to share computational resources.

Since Harness is object oriented in its design, the plug-in function can be generalized to accept a wide range of components from low level modules to entire DVM. For symmetry there is also an unplug function that reverses the plug-in operation for a given component.

There are times when it is convenient to load and run a plug-in in a single step. A common example is the spawning of a parallel application. There are other times when it is more efficient to preload plug-ins and run them only when necessary. For this reason the core plug-in service presently allows three different instantiation options. The first option is "load only". This is for plug-ins such as libraries that are not runable, for preloading runable processes, and for joining two already running components together. The second option is "load and run". This provides a convenient means to add new hosts to the DVM by starting a remote daemon and also for spawning parallel applications. The third option is "load, run, and wait till completion". This option provides the same model as Remote Procedure Call (RPC) in Unix and has the same utility. As users get more experience with Harness, other options may be added depending on the application needs.

From a practical point of view it may make sense to consider plugging in a single component in a single location separately from plugging in components across many (possibly heterogeneous) locations. In the latter case there are situations, such as spawning a parallel application, where no coordination is required, and other situations, such as loading a new communication plug-in, where strict coordination is required. The latter case also requires the specification of a set of locations as well as a vector of return codes to specify the success or failure of each of the components. For these reasons there are separate functions for these two cases.

Here are the four functions being considered for the plug-in/run interface.

```
object = plugin( component, args, options )
         run( object, args )
         stop( object )
         unplug( component )
```

4.2 Global Storage

The Harness virtual machine needs to store its state information in a robust, fault tolerant database that is accessible to any daemon in the DVM. User applications also need to have a robust information storage and retrieval. For Harness we propose to use the persistent message interface that was developed for the final version of PVM.

In a typical message passing system, messages are transitive and the focus is on making their existence as brief as possible by decreasing latency and increasing bandwidth. But there are a growing number of situations in distributed applications in which programming would be much easier if there was a way to have persistent messages. This was the purpose of the *Message Box* feature in PVM 3.4 [3]. The Message Box is an internal tuple space in the virtual machine.

The six functions that make up the Message Box in Harness are:

```
index = putinfo( name, msgbuf, flag )
        recvinfo( name, index, flag )
        delinfo( name, index, flag )
        searchmbox( pattern, matching_names, info )
        subscribe_for_notification()
        cancel_subscription()
```

Tasks can use the Harness data transfer routines to create an arbitrary message, and then use `putinfo()` to place this message into the Message Box with an associated name. Copies of this message can be retrieved by any Harness task that knows the name. And if the name is unknown or changing dynamically, then `searchmbox()` can be used to find the list of names active in the Message Box. The flag parameter defines the properties of the stored message, such as, who is allowed to delete this message, does this name allow multiple instances of messages, does a *put* to the same name overwrite the message? The flag also allows extension of this interface as users give us feedback on how they use the features of Message Box.

The `recvinfo()` function generates a request to the local daemon to find the message associated with the specified name in the global storage and to send it to the reqesting task. The task receives and decodes the message using the Harness data transfer routines.

Here are a few of the many uses for the Message Box feature. A visualization tool can be started that queries the message box for the existence and instructions on how to attach to a large distributed simulation. A performance monitor can leave its findings in the Message Box for other tools to use. A multipart, multiphase application can use the Message Box as a means to keep track of the different parts of the application as they move to the best available resources.

The capability to have persistent messages in a distributed computing environment opens up many new application possibilities, not only in high performance computing but also in collaborative technologies.

4.3 Data Transfer

Because the daemons need to be able to send and receive requests, the core services need to provide a means to transfer data between two components in a DVM. Experience with PVM has shown that message-passing is the most portable and heterogeneous of the possible paradigms, so this is what is provided in Harness as the basic data transfer mechanism.

The core services provide the blocking and nonblocking send and receive routines that users have grown accustomed to in MPI [8] and PVM. The core services also provide the test, wait, and cancel operations required with the nonblocking sends and receives.

The architecture and basic design of Harness have been completed, based on successful proof-of-concept experiments performed over the past year. A prototype implementation of Harness written entirely in Java and based on the first of two distributed control schemes is now running [7]. Another prototype implementation written in C++ and based on the second distributed control scheme should be running by the Fall of 1998.

A publicly available version of Harness is planned for the Fall of 1999.

Acknowledgements

The ideas expressed in this paper are the product of the entire Harness development team with includes: Micah Beck, Jack Dongarra, Graham Fagg, Al Geist, Paul Gray, James Kohl, Mauro Migliardi, Keith Moore, Terry Moore, Philip Papadopoulos, Stephen Scott, and Vaidy Sunderam. It is a collaborative effort between Oak Ridge National Laboratory, Emory University, and the University of Tennessee.

References

1. G. Fagg, K. Moore, J. Dongarra, and A. Geist, Scalable Networked Information Processing Environment (SNIPE), In *Proc. SC97*, November 1997.
2. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.
3. A. Geist, J. Kohl, R. Manchek, and P. Papadopoulos. New features of PVM 3.4 and beyond. In Dongarra, Gengler, Tourancheau, and Vigouroux, editors, *EuroPVM'95*, pages 1–10. Hermes Press, Paris, 1995.
4. A. Geist, and P. Papadopoulos. Symmetric Distributed Control in Network-based Parallel Computing Journal of Computer Communications, 1998. (http://www.epm.ornl.gov/harness/dc2.ps).
5. P. Gray and V. Sunderam. *The IceT Project: An Environment for Cooperative Distributed Computing*, 1997. (http://www.mathcs.emory.edu/~gray/IceT.ps).
6. T. Ludwig, R. Wismuller, V. Sunderam, and A. Bode. Omis – on-line monitoring interface specification. Technical Report TUM-19609, Technische Universität München, February 1996.
7. M. Migliard, J. Dongarra, A. Geist, and V. Sunderam. *Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System*, ISCOPE 1998. (http://www.epm.ornl.gov/harness/om.ps).
8. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, 1996.