On the Viability of Component Frameworks for High Performance Distributed Computing: A Case Study

Dawid Kurzyniec, Vaidy Sunderam
Dept. of Math and Computer Science, Emory University
1784 North Decatur Road, Suite 100
Atlanta, GA 30322, USA
{dawidk,vss}@mathcs.emory.edu

Mauro Migliardi
University of Genoa, DIST
Via Opera Pia 13
16145 Genoa, Italy
om@dist.unige.it

Abstract

Software infrastructures that support metacomputing are evolving from traditional monolithic, platform-specific systems to component and service-based frameworks. In this paper we demonstrate that contrary to popular belief, such modular software systems are capable of delivering good to excellent performance, support legacy as well as new application programming paradigms, and deliver enhanced functionality. The Harness system, a software backplane enabling reconfigurable distributed concurrent computing is used to emulate the PVM programming environment. Numerical kernel benchmarks show that application performance results using the emulator and native versions are within a few percent of each other. Coupled with the ability to leverage pre-existing and specialized modules, our experiences suggest that service-oriented computational grids may be constructed rapidly and effectively via such component-based architectural frameworks that deliver full functionality without compromising efficiency.

1 Introduction

Effective systems software frameworks are critical for large scale loosely coupled high performance distributed computing. With the increasing adoption of grids and metacomputing platforms, modular software infrastructures that can adapt to varying hardware, networks, and applications are gaining acceptance. Such software systems, typically realized via component-based architectures, are ideally suited to heterogeneous environments, since specific components appropriate to a given situation can be assembled, and furthermore, the software evolves naturally as the underlying technologies change. However, there is a tendency to assume that owing to their very nature, such frameworks might deliver poor performance and thereby negate many of the advantages of high performance distributed computing. In this paper, we demonstrate via a case study that component-based software frameworks, written in Java, can and do deliver cluster computing performance comparable to that of monolithic native software systems.

Our experiences are based on Harness [15, 11], a metacomputing framework that is built on the concept of extensible reconfigurable virtual machines. The underlying motivation behind Harness was to develop a flexible metacomputing platform for the next generation, capable of rapidly incorporating new technologies as they emerge. Harness introduces a modular architecture, based on the concept of a lightweight software backplane configured according to users' needs by attaching additional software components, i.e. *plugins*. Some plugins are provided as part of the Harness system while others may be developed by individual users for special purposes; yet others could be obtained from third-party repositories. By configuring a Harness virtual machine using a suite of plugins appropriate to the particular hardware environment, the problem being solved, and resource constraints, users can achieve functionality and performance best suited to their specific needs. Furthermore, due to its modular architecture, plugins may be developed incrementally to facilitate emerging technologies like faster networks, new algorithms, or resource acquisition schemes, enabling their incorporation into the Harness system without a major reengineering effort.

Java has been the technology of choice for the implementation of the Harness framework, as it offers a convenient programming paradigm and substantially increased development efficiency in comparison to traditional programming languages and technologies. However, the applicability of Java for high performance computing infrastructures has been questioned in recent years due to performance problems caused by the portable, bytecode-based nature of the language, despite the emergence of optimized JIT compilers and other performance enhancing schemes.

Moreover, similar complaints have been traditionally reported against object- and component-oriented systems in general, as opposed to contained systems. In this paper, we compare the performance of the PVM [8] emulator, that we have implemented as a Harness plugin, against the original PVM implementation. We show not only that the overheads related to object oriented technologies and Java may, in many cases, turn out to be negligible, but in fact, that some new codes may even outperform native implementations on modern hardware architectures.

2 Related Work

A number of other scientific computing projects have adopted the component-oriented programming paradigm, and many have reported significant success. Cactus [2] is a problem solving environment that is configured by attaching application modules (*thorns*) into the core backplane (*flesh*). Cactus has been used successfully for large scale, geographically distributed astrophysical simulation experiments [1]. Legion [9] is component-based metacomputing system that provides an illusion of a virtual machine over heterogeneous, geographically distributed resources. XCAT [7] is a component framework based on XML standards that is designed for easy composition and integration of distributed applications. The OGSA model [6] integrates the service-based programming paradigm and Web Service technologies with the Globus toolkit [5].

Several projects have explored PVM enhancements and emulations as well. For example, Legion supports the PVM programming model with a dedicated version of the PVM library [12], thus making it possible to run PVM applications within preexisting Legion environments. "JPVM" [4], on the other hand, is a pure Java implementation of a PVM-like system. It supports a PVM-like communication API in Java, and as such is not interoperable with traditional PVM applications. A different approach is adopted in the "jPVM" project [22], whose goal is to enable cooperation between Java applications and traditional PVM runtime via Java language bindings to the PVM library. While these projects demonstrated viability under specific sets of circumstances, the Harness exercise highlights (1) the retention of full backward compatibility with an existing parallel programming model; (2) true exploitation of component architectures via the leveraging of existing general purpose and specialized modules; and (3) the attainment of performance levels equaling that of native environments despite the use of Java-based component technologies.

3 The Harness Framework

The fundamental abstraction in the Harness framework is the *Distributed Virtual Machine* (DVM), as shown in Fig. 1. A DVM is associated with a symbolic name that is unique in the Harness name space, but has no physical entities connected to it. *Heterogeneous computational resources* may enroll into a DVM at any time; however, at this level, the DVM is not yet ready to accept requests from users. In order for that to be possible, computational resources enrolled in a DVM need to load *plugins* – software components implementing specific *services*. By loading an appropriate set of plugins, the DVM can build a consistent *service baseline*. Users may *reconfigure* the DVM at any time both in terms of computational resources enrolled by having them *join* or *leave* the DVM, and in terms of services available by *loading* and *unloading* plugins.

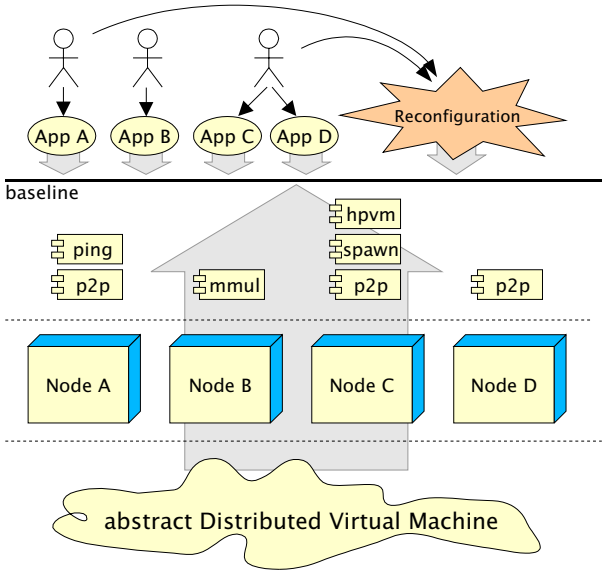


Figure 1. Model of a Harness Distributed Virtual Machine (DVM).

Harness is implemented as a software backplane or kernel, whose main function is to enable heterogeneous computational resources to form a DVM, and make them capable of delivering a consistent service baseline to users. This goal requires the components comprising the framework to be as portable as possible over as large a selection of systems as possible. Moreover, in order to make services easily available, the system must have the ability to acquire plugins from multiple, searchable software repositories without user intervention. This set of requirements motivated us to choose Java as the implementation technology for Harness, as it naturally supports both the above features, while also significantly simplifying the development of multithreaded, distributed applications. It must be noted, however, that various techniques and technologies, including JNI [13] and TCP sockets, may be used to achieve interoperability with

legacy and native codes.

The component-oriented nature of Harness can be exploited in many ways, including the emulation of specific concurrent computing environments to run legacy applications. Since Harness provides many task and message-management functions via built-in plugins, a specific environment (e.g. Linda [14] or Distributed Shared Memory or MPICH [10]) can be emulated by leveraging these built in facilities and writing a few environment-specific plugins. Both MPI [3] and PVM [16] plugins have been successfully developed and deployed to validate this approach. This paper discusses component framework performance in the context of Harness-PVM; the design of a PVM emulator, capable of substituting the PVM runtime in a manner that is seamless and transparent to legacy PVM application codes, is described in the next section.

4 Design of the Harness PVM Plugin

The architecture of the original PVM system is shown in the Fig. 2. A virtual machine is constituted by the set of computational nodes forming a host pool. Every node runs a dedicated resident process, termed the *PVM daemon*. Each daemon maintains a local copy of a *host table* containing globally synchronous information about all other daemons within the PVM. Thus, daemons share global state and communicate directly with each other.

A PVM application is a collection of *tasks* distributed over computational nodes. Unlike daemons, tasks do not share any global state information; they only communicate with a local daemon that provides them with a point of contact, message routing, authentication, process control, and fault detection. Through these facilities, application processes interact with each other to carry out concurrent activities via distributed memory multiprocessing. Every task must be linked against the *PVM library* that provides a PVM access endpoint and communicates with the local daemon through TCP or a Unix domain socket. The link between the library and the daemon is defined in terms of an application-level communication protocol. This particular design aspect of PVM is especially advantageous in the context of our project, because it makes the PVM library oblivious to the technology used for daemon implementation as long as the wire protocol is properly supported. Thus, this design aspect allowed us to substitute the daemon with its Harness counterpart, thereby retaining binary compatibility with existing PVM applications.

The architecture of the Harness PVM emulator is shown in Figure 3. As illustrated, the Harness version of the PVM daemon (*hpvmd*) is not self-contained but depends on other, general-purpose plugins running within the Harness kernel on every PVM node:

- the *transport plugin*, providing point-to-point and multicast message passing capabilities;
- the *spawner plugin*, giving the ability to start processes on remote machines;
- the *notifier plugin*, tracing DVM events (like node crashes) and forwarding them to registered listeners;
- the *database plugin*, providing simple registration and lookup functions.

It is important to note that dependencies between these plugins are soft, and purely interface-based. For instance, consider the transport plugin. For our tests, we use the TCP/IP-based realization; however, another implementation (e.g. based on Myrinet GM [17]) may be substituted without any changes to the *hpvmd* component. Similarly, an extended version of the spawner plugin, capable of downloading binaries from the network before starting a process, will immediately enhance the emulator with a *staging* facility, making it no longer necessary to pre-install executables on computing nodes before running an application.

The current version of Harness-PVM constitutes a near-complete equivalent of the PVM 3.4.3 distribution. A full set of point-to-point and multicast communication routines is supported, along with process control, signaling, and notification functions, and task output redirection features.

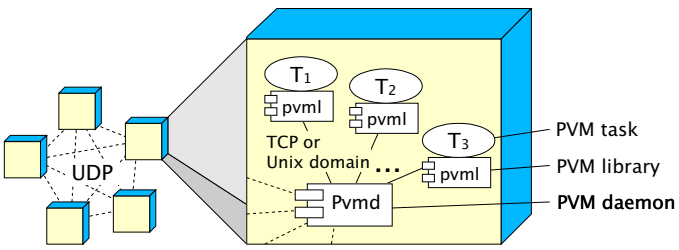


Figure 2. PVM architecture

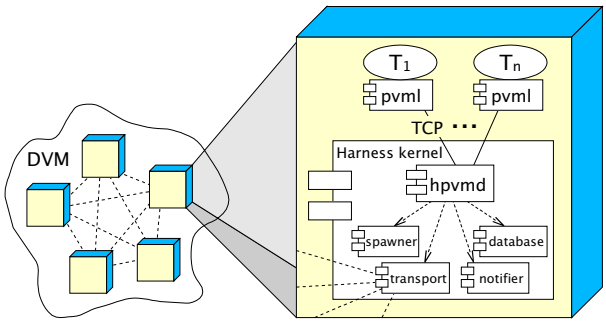


Figure 3. PVM emulation in Harness

Missing functionality includes lack of support for Unix domain sockets (thus requiring that the PVM library be compiled with the `NOUNIXDOM` flag), dynamic process groups, tracing, and logging. This missing functionality typically affects only PVM tools like `xpvm` which therefore cannot yet be used with the Harness-PVM emulator; we expect this support to be provided in the near future. Most other applications run seamlessly, even if they use advanced PVM functions, as for instance does the original PVM console program.

5 Performance Results

Design, implementation, and engineering details regarding the Harness-PVM system are presented elsewhere [16]; in this paper we focus on the performance implications of emulating PVM via component-based frameworks written in Java. Our performance tests consisted of a series of runs of various PVM applications that were executed using the Harness-PVM emulator as well as the original PVM system (for both Unix domain and TCP socket versions). Applications include simple throughput and latency tests, a matrix multiplication benchmark, and the set of five kernels from the NAS Parallel Benchmarks suite (NPB) [18, 23].

The runtime platform consisted of a local network of 16 Sun Blade 1000 workstations running 64-bit Solaris 8, each equipped with two UltraSPARC-III 750 MHz CPUs, a 150 MHz system clock, 1GB of RAM, and a Fast Ethernet network adapter. All executable and data files were installed on a shared NFS filesystem and the machines were exclusively dedicated to the benchmarks. To minimize standard error, tests were repeated in three series consisting of 2 to 5 runs of every benchmark. We performed all tests for the normal, dual-CPU configuration, as well as for a single-CPU configuration (with one CPU disabled on every host). The Java platform used was Java 2 Standard Edition, version 1.4.0 beta 3 for Solaris/SPARC.

5.1 Simple experiments

The throughput tests involved transmitting a unidirectional stream of messages between two distinct nodes. 32 MB of data was being sent in every case except for PVM runs with small messages (256 bytes or less). In those cases, the total number of messages was limited to 2^{16} in order to obtain reasonable execution times. The results are shown in figures 4(a) and 4(b) for uni-CPU and dual-CPU configurations, respectively. We measured the throughput of pure TCP transfer for C and Java, the throughput of three different PVM configurations (direct route, Unix domain sockets, and TCP sockets), and finally, the Harness-PVM. As can be seen in the figures, direct communication via a TCP link was able to saturate the maximum 100Mbps bandwidth very easily. Good performance was also demonstrated by PVM with the direct route option, that allows the two tasks to bypass daemons and perform communication through a direct TCP socket. Enabling default routing (via daemons on the sending and receiving nodes) limited PVM throughput to about 4.7 MB/s, regardless of the socket type used for library-daemon communication. All test cases showed little or no dependence on the number of CPUs user per node.

While most of the measurements were as expected, the results for the Harness-PVM plugin in the default-routing mode were surprising – the throughput peaks at 7 MB/s (47% improvement over PVM) in uni-CPU mode, and as much as 10 MB/s (110% improvement over PVM) in dual-CPU mode (note that in direct route mode, the emulator exactly matches the performance of direct-routing PVM since the communication bypasses daemons completely). The causes for this surprising speedup are twofold. First, the TCP protocol that our emulator uses for inter-daemon communication (via the Harness TCP transport plugin) is much better suited to handle uni-directional streams of data than the UDP protocol used by the original PVM. TCP provides ordering and reliability guarantees directly on top of IP, whereas PVM has to provide the same over UDP. Second, modern versions of Java Virtual Machines (including the SDK/JRE used in our experiments) are able to perform aggressive run-time optimizations, mostly unavailable to static C compilers, and are able to take full advantage of particular run-time platform characteristics [19, 20] – in this case, a powerful SPARC-III architecture, 64-bit operating system and large memory. [1] It is important to note that the UltraSPARC-III processors that we used are designed with efficient Java support in mind [21], and that the Sun HotSpot JVM is especially well optimized for these processors [19]. The difference in Harness-PVM performance between uni- and dual-CPU modes can be explained by the multithreaded nature of the emulator that exploits the JVM allocation of workload more evenly between both processors. In contrast, PVM is single-threaded by design (it performs context-switching internally within a single process) and the daemons cannot benefit from additional CPUs.

To measure latency, we performed a series of 100,000 ping-pong tests. The results are shown in Figure 5. Not surprisingly, the smallest latency of about 0.1 ms was associated with direct TCP communication. PVM direct route communication ranked second, adding about 0.05 ms to the pure TCP overhead. Unix domain- and TCP-based PVM routing exhibited latencies of about 0.27 and 0.3 ms, respectively. These measurements are reasonable considering that they include the accumulated latency of three store and

[1] In fact, our earlier results for SPARC-II and Linux/x86 with a smaller amount of RAM was less spectacular, showing a 10-20% performance *overhead* of the Harness-PVM emulator as compared against PVM in the uni-CPU mode.

forward stages (socket transfers) through which the packet is routed. Finally, the Harness-PVM emulator exhibited the largest latency of 0.55 ms / 0.48 ms for uni- and dual-CPU modes, respectively, which is a 75% / 55% overhead in comparison to PVM with default TCP routing. [2]

In dedicated parallel computers and homogeneous multiprocessors like the IBM SP3, this increased latency would likely affect application performance to a noticeable degree. However, in general-purpose clusters and distributed environments, nodes are never in lock-step due to uneven system loads and variations in network delays. This fact substantially alleviates the impact of increased latency on application performance. Nonetheless, these ambivalent differences in throughput and latency between the emulator and native PVM can actually be noticed in the remainder of the results, with Harness-PVM gaining the edge over PVM in throughput-critical applications, and vice-versa.

We are currently working on the improvements to the Harness-PVM implementation to reduce the latency overhead, and we expect to be able to present better results in the future. For instance, we have identified at least one thread context switch during the message dispatch that could potentially be avoided. However, even despite this issue our current implementation yields competitive performance results, as will be shown in the remainder of this paper.

The matrix multiplication tests, illustrated in Figure 6, involved multiplying two 1200x1200 matrices using the pipe-multiply-roll algorithm that adapted a simple block approach for multiplication of local submatrices.[3] These tests showed little dependence on the underlying runtime environment, and any differences are within the standard error range; the closeness of the results can be explained by the relatively low communication volume involved in this experiment.

5.2 NAS kernel benchmarks

Experimental results for all five NAS kernel benchmarks are shown in Figures 7 to 11. Each graph shows the total execution time in seconds as reported directly by the benchmark code. For all kernels except EP, the total communication time is also shown by the solid horizontal line drawn within the appropriate result bar. The bottom part of the bar (from the X-axis up to the level of that line) represents communication time, and the remainder of the bar represents computation time. All NAS kernels were executed with data size "A" (adequate for moderately powerful workstations), that allowed us to perform tests even on a small number of computing nodes, including a single node. In most cases, the differences in performance between PVM and Harness-PVM were only caused by communication differences, with computation time remaining essentially constant. This is not surprising, since the same application codes were in use. As mentioned earlier and as expected, Harness-PVM tended to exhibit better performance in throughput-bound benchmarks, as well as more significant improvement in dual-CPU mode over the uni-CPU mode. This is consistent with the results shown in the previous section suggesting that throughput of Harness-PVM is better than that of PVM whereas the latency is worse, and that both of these parameters improve in the dual-CPU mode for Harness-PVM, but not as much for native PVM.

The EP kernel executes multiple iterations of a loop in which a pair of random numbers are generated and tested for if they satisfy a specific condition, and falls into the category of *embarassingly parallel* codes. The results shown in Figure 7 are therefore not surprising, showing linear scalability and marginal dependencies on the underlying runtime environment.

The CG kernel, shown in Figure 8, uses gradient methods to find an eigenvalue of a sparse matrix with a random pattern of nonzeros. It is characterized by long-distance, unstructured communication patterns. As the results suggest, the scalability of this problem is particularly poor and the communication load rapidly increases with the number of nodes, exceeding 80% of the total time. The emulator slightly outperformed PVM (up to 14%) in this test in all cases in dual-CPU mode. In the uni-CPU mode, PVM was marginally faster on 9 and 16 nodes.

Results for the FT benchmark, which solves a 3-dimensional partial differential equation using FFT, are shown in the Figure 9. Sizes of messages exchanged in this test are especially large, thereby exploiting the improved throughput of Harness-PVM, resulting in significantly better results and up to 21% performance improvement over PVM. In this test, Harness-PVM outperformed PVM it in all cases except the 2-node uni-CPU run in which the numbers are almost equal.

Figure 10 displays the results of the MG benchmark that executes 4 iterations of the multigrid algorithm to solve the discrete Poisson problem on a three-dimensional grid. Highly structured communication patterns in MG resulted in relatively low communication overhead and little variance among the different runtime environments; however, the Harness-PVM emulator introduced a constant time overhead of about 4-5 seconds per run regardless of the number of nodes the application was running on, most likely as a result of process initialization overheads that are being further investigated.

Finally, figure 11 depicts the results of the IS benchmark that involves sorting over a large array of integer keys. Communication in this benchmark is very frequent and rela-

[2] Of course, Harness-PVM can also work in the direct route mode, matching the throughput and latency results of direct routed PVM.

[3] the use of a naive triple-nested loop would result in confusing superlinear speedup on 4 nodes as compared to 1 node, due to very poor utilization of the CPU cache in the 1-node case.

tively low-volume, and the communication pattern is a fully interconnected graph. As shown in the figure, in some cases communication accounts for as much as 90% of the total execution time. The effects of competition between throughput and latency aspects are clearly reflected, as neither PVM nor Harness-PVM gained a definite performance advantage across all test cases (although PVM performed significantly better on 4 and 8 nodes in uni-CPU mode). As with the FT and CG tests, this benchmark unambiguously demonstrates that Harness-PVM performance improves significantly in dual-CPU mode, whereas the addition of the second CPU has little effect on the performance of native PVM.

6 Conclusions and Future Work

In this paper, we have described an exercise to emulate the PVM programming environment using the Harness framework, an adaptive, component-based software infrastructure that facilitates rapid deployment of metacomputing systems. In terms of functionality, this approach demonstrates that component software modules, developed independently, may be assembled with assistance from a backplane or kernel, to construct metacomputing support infrastructures most appropriate to the application and to the underlying resources. With the recent interest in grid-service architectures (see OGSA[6]), assembling service components to configure computational grids is a promising possibility, and our experiences show that component frameworks work well in high performance distributed computing settings, even when deployed via portable Java-based technologies.

While functionality and suggested approaches to service-grids are shown to be viable, an important goal of this paper was to demonstrate that high performance is not sacrificed by the use of Java and component technologies. Our benchmarking experiments indicate that this is indeed the case; Harness-PVM performs at levels comparable to, if not equaling or bettering, native PVM performance. We believe that even better performance is attainable through system level optimizations that may result in latency improvements; we are currently in the process of conducting these modifications. This paper, however, is presented as a case study in the use of *component architectures* as an *evolution path* in metacomputing systems and computational grids. By supporting legacy applications at full levels of functionality and performance while enabling both incremental and radical infrastructural modifications, this approach adapts to architectural, software technology, resource, and application changes directly in production mode.

Acknowledgements

This work was supported in part by NSF grant ACI-9872167 and DoE grant DE-FG02-99ER25379. Thoughtful comments by several anonymous reviewers led to a number of improvements in the revised version of this paper.

References

[1] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. In *Supercomputing 2001 Conference*, Denver, Colorado, USA, November 10-16 2001. Available at `http://www.cactuscode.org/Papers/GordonBell_2001.ps.gz`.

[2] G. Allen, T. Goodale, G. Lanfermann, T. Radke, and E. Seidel. The Cactus Code: A problem solving environment for the grid. In *Proceedings of the First EGrid Meeting*, Poznań, Poland, 2000. Available at `http://www.cactuscode.org/Papers/Egrid_2000.ps.gz`.

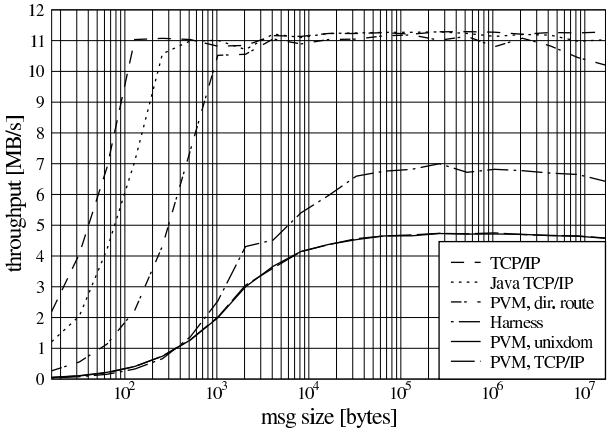
[3] G. Fagg, A. Bukovsky, and J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1496, Oct. 2001. Available at `http://icl.cs.utk.edu/publications/pub-papers/2001/harness-ftmpi-pc.pdf`.

[4] A. Ferrari. JPVM: The Java parallel virtual machine. `http://www.cs.virginia.edu/~ajf2j/jpvm/`.

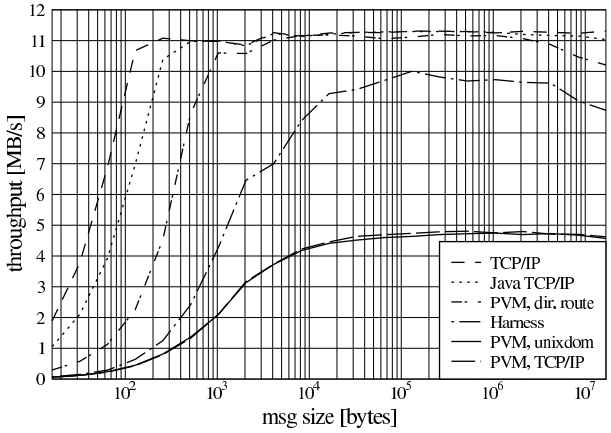
[5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, Jan. 2002. Available at `http://www.globus.org/research/papers/ogsa.pdf`.

[7] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz. Programming the Grid: Distributed software components, P2P and Grid Web Services for scientific applications. Available at `http://www.extreme.indiana.edu/~gannon/ProgGrid/ProgGridsword.PDF`.



(a) uni-CPU configuration



(b) dual-CPU configuration

Figure 4. Throughput test results

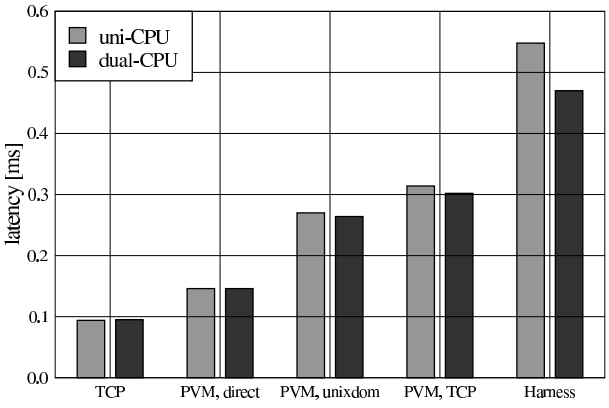


Figure 5. Latency test

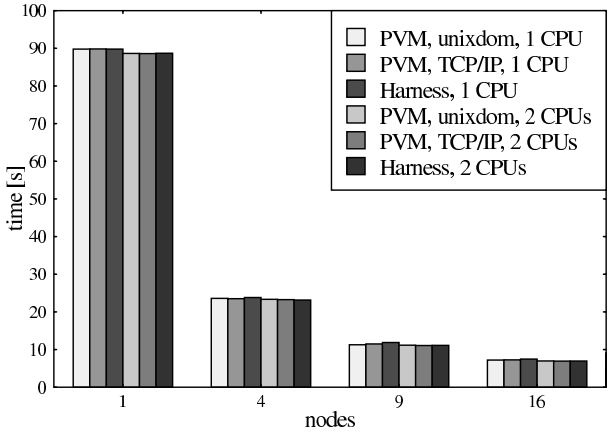


Figure 6. Matrix multiplication, two 1200x1200 matrices.

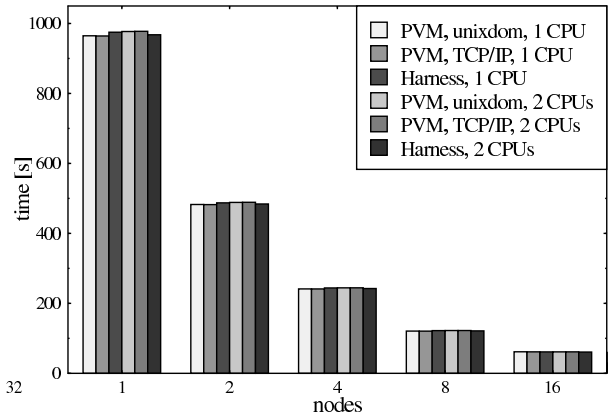


Figure 7. NAS EP benchmark results, problem size 2^{28}.

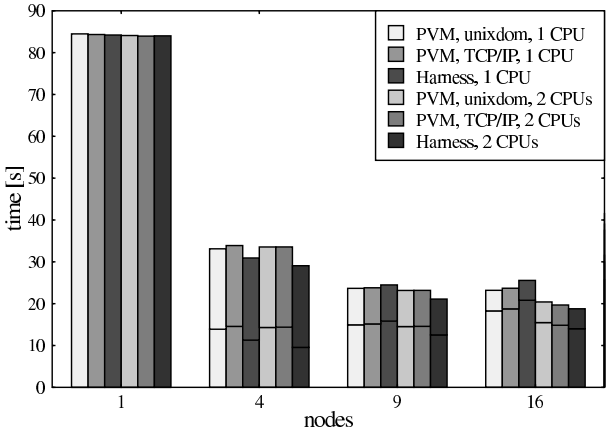


Figure 8. NAS CG benchmark results, solving an unstructured square linear system with matrix size 14000 in 15 iterations.

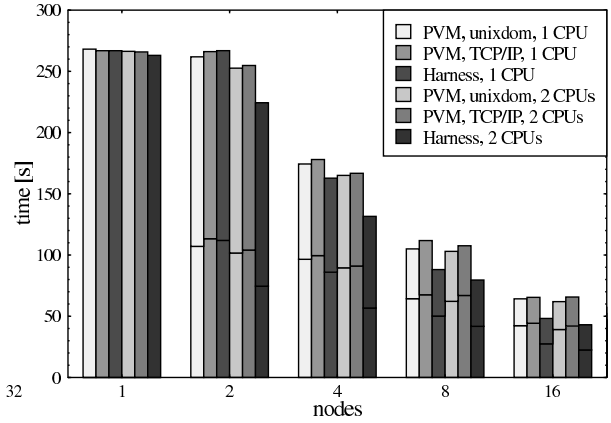


Figure 9. NAS FT benchmark results, solving 3-D partial differential equation using FFT on 256x256x128 grid.

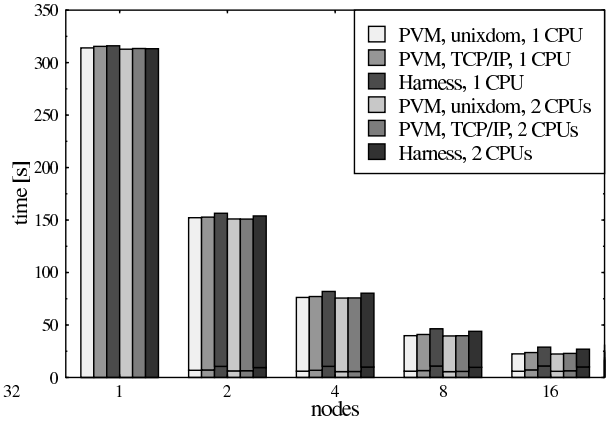


Figure 10. NAS MG benchmark results, solving Poisson problem on 256x256x256 mesh in 4 iterations.

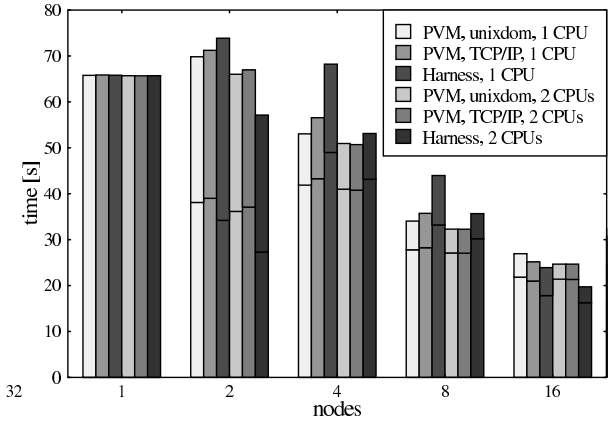


Figure 11. NAS Integer Sort benchmark results, sorting 2^{23} keys in range $[0..2^{19}]$.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.

[9] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. R. Jr. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, June 1994.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[11] Harness project home page. `http://www.mathcs.emory.edu/harness`.

[12] R. R. Harper. Interoperability of parallel systems: Running PVM applications in the Legion environment. Technical Report CS-95-23, Department of Computer Science, University of Virginia, May 03 1995.

[13] Java Native Interface. `http://java.sun.com/j2se/1.3/docs/guide/jni/index.html`.

[14] A. Matrone, P. Schiano, and V. Puoti. LINDA and PVM: A comparison between two environments for parallel programming. *Parallel Computing*, 19(8):949–957, Aug. 1993.

[15] M. Migliardi and V. Sunderam. The Harness metacomputing framework. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, USA, March 22-24 1999. Available at `http://www.mathcs.emory.edu/harness/PAPERS/pp99.ps.gz`.

[16] M. Migliardi and V. Sunderam. PVM emulation in the Harness metacomputing system: a plug-in based approach. In *Proceedings of 6th EuroPVM-MPI99 Conference, Lecture Notes in Computer Science*, volume 1697, pages 117–124, Barcelona, Spain, September 26-29 1999. Springer Verlag.

[17] Myricom. The GM message passing system. Available at `http://www.myri.com/scs/GM/doc/gm.pdf`.

[18] NAS parallel benchmarks. `http://www.nas.nasa.gov/NAS/NPB/`.

[19] Sun Microsystems. Java 2 platform, Standard Edition version 1.3.1 for the Solaris operating environment. `http://www.sun.com/software/solaris/ds/ds-j2se131/`.

[20] Sun Microsystems. Java HotSpot technology. `http://java.sun.com/products/hotspot/`.

[21] Sun Microsystems. UltraSPARC III technical highlights. `http://www.sun.com/sparc/UltraSPARC-III/USIIITech.html`.

[22] D. Thurman. jPVM: The Java to PVM interface, Dec. 1996. `http://www.chmsr.gatech.edu/jPVM`.

[23] S. White, A. Alund, and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM based networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, Apr. 1995.