MPI Cluster System Software

Narayan Desai, Rick Bradshaw, Andrew Lusk, and Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, Illinois 60439

[1]

Abstract. We describe the use of MPI for writing system software and tools, an area where it has not been previously applied. By "system software" we mean collections of tools used for system management and operations. We describe the common methodologies used for system software development, together with our experiences in implementing three items of system software with MPI. We demonstrate that MPI can bring significant performance and other benefits to system software.

1 Introduction

In this paper we use the term "cluster system software" to describe the collection of tools used to manage a parallel machine composed of multiple nodes. Such a machine typically includes a local file system and perhaps a shared file system. We are not discussing per node software such as the node OS or compilers. Rather, the tools we refer to are the programs and scripts, either externally provided or locally written, that support the management of the cluster as a parallel machine and the execution of parallel jobs for users.

Research into system software is not by any means a new activity; previous efforts have been discussed in [8] and [11]. These efforts have focused around scaling serial unix services to large cluster scale. The approach described in this paper describes the use of MPI to create scalable, parallel system tools. As clusters have become larger, the lack of scalability in traditional system management tools is becoming a bottleneck. This situation has led to a gap in the amount and quality of parallelism used in system support programs as opposed to application programs.

Parallel *applications* have embraced MPI as a portable, expressive, scalable library for writing parallel programs. MPI is already available for applications on most clusters, but it has seldom been used for writing system programs. (An exception is the collection of Scalable Unix Tools described in [9]. These are MPI versions of common user commands, such as `ls`, `ps`, and `find`. In this paper we focus on a different class of tools, more related to system admainstration and

[1] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, SciDAC Program, Office of Science, under Contract W-31-109-ENG-38.

operation.) As we hope to show, MPI is a good choice for systems tools because MPI's scalability and flexibility, particularly its collective operations, can provide a new level of efficiency for system software. In this paper we explore the potential of using MPI in in three different system software tasks:

file staging Applications need both executable and data files to be made available to the individual nodes before execution, and output data may need to be collected afterward. This process is awkward and can be painfully slow when the cluster, for the sake of scalability, has no globally shared file system.

file synchronization The `rsync` program is a classical Unix tool for ensuring that the content of file systems on two nodes is consistent. A cluster may require many nodes to be synchronized with a "master" node.

parallel shell We have written a parallel shell called MPISH to supervise the execution of parallel jobs for users. It handles many of the same functions that a normal shell such as `sh` or `bash` does for serial applications (process startup, environment setup, `stdio` management, interrupt delivery) but in a parallel and scalable way.

In Section 2 we discuss the shortcomings of current approaches and elaborate on the advantages and disadvantage of an MPI-based approach. In Section 3 we briefly describe the Chiba City [3] cluster at Argonne, where our experiments took place. In Sections 4, 5, and 6 we describe the three example tasks mentioned above. In each case we describe the old and new versions of the tool we have developed for that task, and we assess the general usefulness of our new approach. Section 7 describes plans for applying the MPI approach to other system software.

2 Background

2.1 The Current Situation

The system software community uses various schemes to address scalability concerns. In some cases, large external infrastructures are built to provide resources that can scale to required levels. In other cases, *ad hoc* parallelism is developed on a task-by-task basis. These *ad hoc* systems generally use tools like `rsh` for process management functionality, and provide crude but functional parallel capabilities.

Such tools suffer from poor performance and a general lack of transparency. Poor performance can be attributed to naive implementations of parallel algorithms and coarse-grained parallelism. Lack of transparency can be attributed to the use of "macroscopic" parallelism: that is, the use of large numbers of independent processes with only exit codes to connect back to the overall computation, leading to difficulty in debugging and profiling the overall task.

2.2 Potential of an MPI Approach

The adoption of *comprehensive* parallelism, such as that provided by MPI, can provide many benefits. The primary benefit is an improved quality of parallelism. A complete set of parallel functionality is not only available but already optimized; whereas with *ad hoc* parallelism collective operations often are not available. Also, MPI implementations are highly optimized for the high-performance networks available on clusters. While these networks are available for use by serial tools through TCP, its performance, even over high performance networks, tends to lag behind the performance provided by the vendor's MPI implementation.

The use of MPI for system software does have some potential disadvantages. Since an MPI approach is likely to utilize collective operations, the simple mechanisms available for providing fault tolerance to pairs of communicating process may not apply. Techniques for providing fault tolerance in many situations are being developed [7] but are (MPI-)implementation-dependent. In addition, the manner in which MPI programs are launched (e.g. `mpiexec`, `mpirun` in various forms) is not as portable as the MPI library itself. System scripts that invoke MPI-based tools may have to be adapted to specific MPI installations.

3 Experimental Environment

Chiba City [3] is a 256-node cluster at Argonne National Laboratory devoted to scalable software research, although "friendly users" also run applications. It is not, however, dedicated to applications use and hence is available for parallel software development and experimentation, particularly in the area of system software. For the past year, its software stack has consisted largely of components developed in the context of the SciDAC Scalable System Software project [13]. That is, the scheduler, queue manager, process manager, configuration manager, and node monitor are all implemented as communicating peer components [4]. Extending these components to include necessary system utilities and support programs initiated the experiments described in this paper.

In addition to being large enough that scalability of system operation is an issue, Chiba City presents particular challenges for application use because of its lack of a global file system. This was a deliberate choice to force the development of economical, scalable approaches to running user applications in this environment.

Instead, Chiba City is divided into "towns" consisting of 32 nodes and a "mayor" node. The eight mayors are connected via gigabit Ethernet to a "president" node. The mayors mount the individual file systems of each node and are connected to them by Fast Ethernet. The nodes all communicate with each other over both a separate Fast Ethernet connection and Myrinet. The MPI implementation on Chiba City is the latest release of MPICH2 [6]. The experiments reported here were all carried out using TCP over Fast Ethernet as the underlying transport layer for the MPI messages, pending finalization of the Myrinet version of MPICH2.

4 File Staging

Chiba City has no shared file system for user home directories. Over the past five years Chiba City has been in operation, we have used two systems that attempt to provide seamless, on-demand access to home-directory data on nodes. The first, *City Transit*, was developed during Chiba City's initial deployment. The second system, implemented in MPI, has been in use for the past year. We will describe each system's implementation in detail, discuss usage experiences, compare performance, and assess both systems.

4.1 City Transit

City Transit was implemented in Perl in the style of a system administrator tool. Parallel program control is provided by `pdsh`, which executes commands across nodes. The staging process starts with a request, either from an interactive user or from a job specification, identifying the data to be staged and the nodes that need the data. The requested data is archived (uncompressed) into a tar file. This tar file is copied via `NFS` from the home directory file server to the cluster master and then to mayors who manage destination nodes. Finally, the nodes unarchive the tar file, available via an `NFS` mount from the mayors. Once this process has completed, the user has the pertinent portion of his home directory on all assigned compute nodes.

Chiba City's management infrastructure (the president and mayors) is connected using Gigabit Ethernet, while the remainder of the system is connected using Fast Ethernet. This process was optimized to improve performance through knowledge of network topology. By preferentially using the faster Gigiabit Ethernet links, we substantially accelerated the file staging process.

Nevertheless, this process has several shortcomings. First, the tar file of user data is written to disk on the cluster master and some mayors during intermediate staging steps. This process can be wasteful, as each tar file is useful only once. For larger file-staging runs, multiple stages of disk writes can constitute a large fraction of the overall run time. Second, process control is provided by multiple, recursive invocations of `pdsh`. The return-code handling capabilities of `pdsh` require that in each invocation, multiple return codes are compacted into one. While this allows for basic error detection, complex error assessment isn't possible. Third, the coarse-grained parallelism provided by this staged approach effectively places several global barrier synchronizations in the middle of the process. Fourth, several shared resources, including file servers, the cluster master, and a subset of the mayors are heavily used during the staging process. Thus users can seriously impact setup performance of one another's jobs. Fifth, since all data transmissions to compute nodes originate in the system management infrastructure, full network bisection bandwidth isn't available.

4.2 Userstage

The second file staging process in use on Chiba City is implemented in MPI. Users (or the queue manager by proxy) specify a portion of the home file system to be used on compute nodes. This data is archived into a compressed tar file and placed in an HTTP-accessible spool. The nodes start the `stagein` executable, which is an MPI program. Rank 0 of this program downloads the stage tar file from the file server via HTTP. The contents of this file are distributed to all nodes using `MPI_Bcast`, and then all nodes untar the data to local disk.

This approach addresses many of the shortcomings of the City Transit approach. MPI provides fine-grained parallelism and synchronization and highly optimized collectives that use available networks effectively. Errors can be easily detected and handled properly. Also, only the initial tar file download is dependent on system infrastructure. Hence, users are better isolated from one another during parallel job setup. Moreover, the use of MPI allows us to largely skip the optimization process, which was quite troublesome during the design process of City Transit.

We carried out a set of additional experiments in which the file was broadcast in various sizes of individual chunks rather than all at once, in order to benefit from pipeline parallelism. This approach did indeed improve performance slightly for small numbers of nodes, where the MPICH2 implementation of `MPI_Bcast` uses a minimal spanning tree algorithm. For more than four nodes, however, MPICH2 uses a more sophisticated scatter/allgather algorithm, and the benefit disappears. This experience confirmed our intuition that one can rely on the MPI library for such optimizations and just code utility programs in the most straightforward way. The only chunking used in the experiment as reported here was to allow for the modest memories on the nodes; this points the way for further optimization of `MPI_Bcast` in MPICH2.

4.3 Practical Usage Experiences

City Transit was used for four years. During this period, there were frequent problems with transient failures. These problems were difficult to track, as debugging information was split across all systems involved in file staging. The debugging process was improved with the addition of network logging. However, many failure modes could be detected only by manual execution of parts of City Transit, because of poor implementation and `pdsh`'s execution model. More important, the tree design of control used didn't allow communication of errors between peers. In case of errors, non-failing nodes could finish the file staging process. As this process could take upwards for 30 minutes and put a heavy load on the system infrastructure, single node failures would cause a large expenditure of resources for no benefit. Also, even in cases where everything worked properly, file staging operations performed poorly, and users were generally unhappy with the process.

Our experiences over the past year with the MPI-based staging mechanism show a stark contrast with our previous experiences. Performance is substantially improved, as shown in Figure 1. This improvement can be attributed to the highly optimized parallel algorithms implemented in MPICH2. The MPI version performs better overall, even though it does not exploit knowledge of the network topology, and as one expects, the advantages are greater both as the number of nodes increases for fixed file sizes and as the file size increases for a given number of nodes. At 128 nodes for a gigabyte file, the new code is writing an aggregate of 216 Mb/sec compared with only 57 Mb/sec for City Transit.

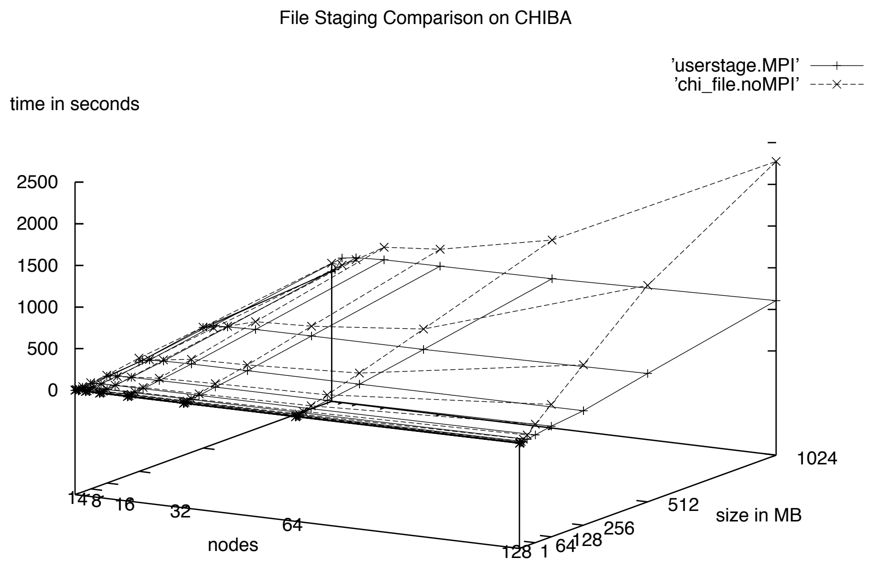

Fig. 1. Performance of old vs. new file staging code.

More important, the reliability of the staging process has been substantially improved. We attribute this to a number of factors. First, the execution model of the new staging tools provides an easier environment for error detection and handling. Because programs aren't called recursively, error codes and standard output are readily available to the executing program. Also, the communication constructs available to MPI programs allow for errors to be easily disseminated to all processors.

As a result of the parallel infrastructure provided by MPI, the volume of code required to implement file staging has been dramatically reduced. *City Transit* consisted of 5,200 lines of Perl code, whereas the MPI-based code is only 84 lines of C. This size reduction makes the code easier to write, debug, and test comprehensively.

5 File Synchronization

File synchronization is an extremely common operation on clusters. On Chiba City, we use this process to distribute software repositories to mayors from the cluster master. This process is executed frequently and can highly tax the resources available on the cluster master. We will describe two schemes for implementing this functionality. The first is based on `rsync`. The second is a file synchronizer written in MPI. We will discuss their implementations and standard usage cases. We will also describe our experiences using each solution and compare relative performance.

5.1 Rsync

`Rsync` is a popular tool that implements file synchronization functionality. Its use ranges from direct user invocation to system management tools. For example, SystemImager [5], a commonly used system building tool, is implemented on top of `rsync`.

`Rsync` is implemented using a client/server architecture. The server inventories the canonical file system and transmits file metadata to the client. The client compares this metadata with the local file system and requests all update files from the server. All communication transactions occur via TCP sockets. This process is efficient for point-to-point synchronizations; however, the workloads we see on parallel machines tend to be parallel, and therefore this process can be wasteful. During a parallel synchronization, the server ends up inventorying the file system several times, and potentially transmitting the same data multiple times. Usually, the files between slaves are already synchronized, so the odds of unnecessary retransmission are high. Many more technical details about `rsync` are available at [12].

5.2 MPISync

To address several issues with our usage of `rsync`, we have implemented a parallel file synchronization tool, which performs a one-to-many synchronization. The steps used in `mpisync` are similar to those used by `rsync`, with the replacement of serial steps by parallel analogues where appropriate. The first step is a file inventory on the master. The result of this process is broadcast to all nodes, which in turn perform an inventory of the local file system. A list of files that need distribution is produced by a reduction of the set of inventories on all nodes. All processors then iterate through files needing distribution. File contents are then broadcast singly. Through the use of `MPI_Comm_Split`, only the processors requiring the data participate in these broadcasts.

5.3 Experiences and Assessment

As we mentioned, `mpisync` was designed to address several shortcomings in the usage for `rsync` for parallel synchronization. It easily handles several of the shortcomings of the serial approach; that is, local files are read only once on the server, and the MPI library provides much more efficient distribution algorithms than a set of uncoordinated point-to-point links can provide. Considerably improved scalability and performance are shown by the benchmark shown in 2. Standard `rsync` performance drops off relatively quickly, and 32-way execution runs are the largest ones that properly complete. On the other hand, `mpisync` runs properly execute concurrently on 128 nodes, and its efficiency is closely related to the underlying MPI implementation. For this reason, we expect that its scalability would remain good at levels larger that this.

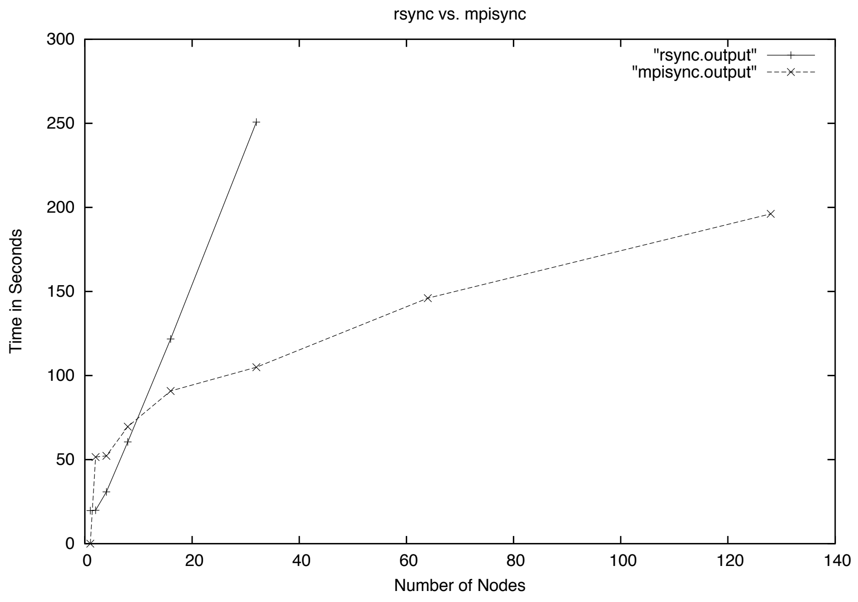

Fig. 2. Performance of old vs. new `rsync` code.

6 A Parallel Execution Environment

Parallel applications are run on clusters in a variety of ways, depending on the resource management system and other tools installed on the cluster. For example, `pdsh` uses a tree to execute the same program on a given set of nodes. MPD [2] is a more flexible process manger, also targeting scalability of parallel job startup, particularly, but not exclusively, for MPI applications. A number of systems (e.g. LAM [1]) require a setup program to be run on one node, which then builds an execution environment for the application on the entire set of allocated nodes.

PBS [10] executes user scripts on a single node, which are in turn responsible for initiating MPI jobs via some MPI-implementation-dependent mechanism.

Just as serial jobs typically are run as children of a shell program, which manages process startup, delivery of arguments and environment variables, standard input and output, signal delivery, and collection of return status, so parallel programs need a parallel version of the shell to provide these same services in an efficient, synchronized, and scalable way. Our solution is `MPISH` (MPI Shell), an MPI program that is started by whatever mechanism is available from the MPI implementation, and then manages shell services for the application. An initial script, potentially containing multiple parallel commands, is provided to the MPI process with rank 0, which then interprets it. Its has all the flexibility of MPI, including the collective operations, at its disposal for communicating to the other ranks running on other nodes. It can thus broadcast executables, arguments, and environment variables, monitor application processes started on other nodes, distribute and collect `stdio`, deliver signals, and collect return codes, all in a scalable manner provided by MPI.

`MPISH` implements the PMI interface defined in MPICH2. Thus any MPI implementation whose interface to process management is through PMI (and there are several) can use it to locate processes and establish connections for application communication. Thus `MPISH` both is itself a portable MPI program and supports other MPI programs.

We have used `MPISH` to manage all user jobs on Chiba City for the past year, and found it reliable and scalable. `MPISH` implements a "PBS compatibility mode" in which it accepts PBS scripts. This has greatly eased the transition from the PBS environment we used to run to the component-based system software stack now in use.

7 Summary and Plans

Using MPI for system software has proven itself to be a viable strategy. In all cases, we have found MPI usage to substantially improve the quality, performance, and simplicity of our systems software. Our plans call for work in several areas.

First, we can improve our existing tools in a number of ways. We can incorporate MPI datatype support into applications, specifically `mpisync`, to improve code simplicity. We also plan the development of other standalone system tools. A number of other common tasks on clusters are currently implemented serially, and could greatly benefit from this approach. Parallel approaches could be taken in the cases of several common system management tasks; system monitoring, configuration management, and system build processes could be substantially accelerated if implemented using MPI.

Use of other advanced MPI features for system software is under consideration. I/O intensive systems tasks could easily benefit from the addition of MPI-IO support. The implementation of the MPI-2 process management features in MPICH2 will allow the implementation of persistent systems software components in MPI, using `MPI_Comm_connect` and `MPI_Comm_spawn` to aid in fault tolerance.

A final area for further work is that of parallel execution environments. `MPISH` provides an initial implementation of these concepts. The idea of an execution environment that allows the composition of multiple parallel commands is quite compelling; this appeal only grows as more parallel utilities become available.

References

1. Greg Burns, Raja Daoud, and James Vaigl. LAM: An open cluster environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.
2. R. Butler, N. Desai, A. Lusk, and E. Lusk. The process management component of a scalable system software environment. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER03)*, pages 190–198. IEEE Computer Society, 2003.
3. `http://www.mcs.anl.gov/chiba`.
4. N. Desai, R. Bradshaw, A. Lusk, E. Lusk, and R. Butler. Component-based cluster systems software architecture: A case study. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER04)*, 2004.
5. Brian Elliot Finley. VA SystemImager. In USENIX, editor, *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, page 394, Berkeley, CA, USA, 2000. USENIX.
6. William Gropp and Ewing Lusk. MPICH. World Wide Web. `ftp://info.mcs.anl.gov/pub/mpi`.
7. William Gropp and Ewing Lusk. Fault tolerance in mpi programs. *High Performance Computing and Applications*, To Appear.
8. J. P. Navarro, R. Evard, D. Nurmi, and N. Desai. Scalable cluster administration - chiba city i approach and lessons learned. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER02)*, pages 215–221, 2002.
9. Emil Ong, Ewing Lusk, and William Gropp. Scalable Unix commands for parallel processors: A high-performance implementation. In Y. Cotronis and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 410–418. Springer-Verlag, September 2001. 8th European PVM/MPI Users' Group Meeting.
10. `http://www.openpbs.org/`.
11. Rolf Riesen, Ron Brightwell, Lee Ann Fisk, Tramm Hudson, Jim Otto, and Arthur B. Maccabe. Cplant. In *Proceedings of the Second Extreme Linux Workshop at the 1999 Usenix Technical Conference*, 1999.
12. `http://rsync.samba.org/`.
13. `http://www.scidac.org/scalablesystems.`