Transparent Real–Time Monitoring in MPI *

Samuel H. Russ[1], Rashid Jean–Baptiste[2], Tangirala Shailendra Krishna Kumar[1], and Marion Harmon[2]

[1]Mississippi State University
NSF Engineering Research Center for Computational Field Simulation
Box 9627
Mississippi State, MS 39762 USA
{russ,krishna}@erc.msstate.edu
http://www.erc.msstate.edu

[2]Florida A&M University
Center for Distributive Computing
Florida A&M University
Tallahassee, FL 32307
{rjean,harmon}@cis.famu.edu
http://www.cis.famu.edu

Abstract. MPI has emerged as a popular way to write architecture–independent parallel programs. By modifying an MPI library and associated MPI run–time environment, transparent extraction of timestamped information is possible. The wall–clock time at which specific MPI communication events begin and end can be recorded, collected, and provided to a central scheduler. The infrastructure to create and collect these events has been implemented and tested, and a future architecture that can use this information is described.

1 Introduction and Background

Distributed processing is an increasingly popular architecture for execution. Because it relies on numerous, typically commodity systems, it offers low cost, inherent fault tolerance, and flexibility. A common problem with commodity systems is lack of software support for parallel execution, fault tolerance, and ability to support applications with real–time constraints.

Countless systems exist today to support distributed computing over commodity workstations. They vary in the degree to which individual workstations retain their identity, the mechanism for and degree of support for various features such as task migration and fault tolerance, and support for legacy applications and common programming standards. While a complete review is well outside the scope of this paper, one good survey of cluster computing systems may be found in [1].

One system that supports a variety of desirable features is the Hector Distributed Run–Time Environment under development at Mississippi State University and Florida A&M University [2]. The environment supports task migration, fault tolerance via checkpoint–and–rollback [3], collection of detailed run–time information [4], and automatic performance optimization. All of these features are provided automatically and transparently to MPI applications. That is, with no source code modifications, MPI applications can run with these features on networked workstations. (A more detailed description of these capabilities is found in the appendix at the end of this article.)

* This work was funded in part by NSF Grant No. EEC–8907070 and NSF Grant No. EEC–9730381.

An alternate approach to the software–based timestamping described here is to augment the system with special–purpose hardware [5]. The hybrid approach has been shown to be minimally intrusive and offer great accuracy, but at the expense of lower portability. It was decided to maintain Hector's software–only, commodity–only design.

The environment had no support for priority–based scheduling or for applications with real–time constraints, and so it was decided to add such features to the environment. The modifications needed to support the creation and collection of timestamps are detailed below, as is an architecture that can use the information for real–time scheduling and performance optimization.

2 An Infrastructure to Timestamp Events

The next step was to specify the API for sending timestamps and to design a way to collect them. The API was designed to match that of the MPI standard that had already been embraced. Collection was a matter of reusing existing modifications for instrumentation.

2.1 Appearance to the Programmer

It was decided that automatically timestamping every MPI event might overwhelm the collection infrastructure, and so an extension to the MPI standard was made. This extension indicates that the next MPI call that is made is to be timestamped and reported.

The format of the extension is *HCT_RT_Tag(int Marker, char *User_Str)*, where Marker and User_Str are user–defined structures that uniquely specify the tag and are reported via the collection mechanism.

For example, if the beginning and end wall–clock time of an MPI_Send is to be tagged, the code may be modified as follows:

```
...
HCT_RT_Tag(Event++,"About to send a number");
...
MPI_Send(...);
...
```

The value of the "Event" variable, the string "About to send a number", and the wall–clock time of the beginning and ending of the call to MPI_Send are reported. In this example, by incrementing Event every time an MPI event is tagged, every MPI event will have a unique value.

2.2 Modifying MPI Wrapper Functions

Modifications had already been made to Hector's MPI library in order to track CPU time spent inside and outside of MPI. This information is used to estimate time spent communicating and computing. These modifications consisted of "wrapper functions" inserted before and after every MPI call in order to update this instrumentation. The wrappers are invoked by using a special #define structure that redirects the function call from the MPI library to the wrapper library, as shown below in Fig. 1.

Two changes were made to support the real–time instrumentation. First, the modified wrappers checked to see if the HCT_RT_Tag function had been called and, if so, tagged and timestamped the MPI library entry (or exit) and sent the information. Second, it was discovered that some MPI library function calls recursively called other MPI library functions. (For example, MPI_Bcast calls MPI_Send.) The wrappers were modified so

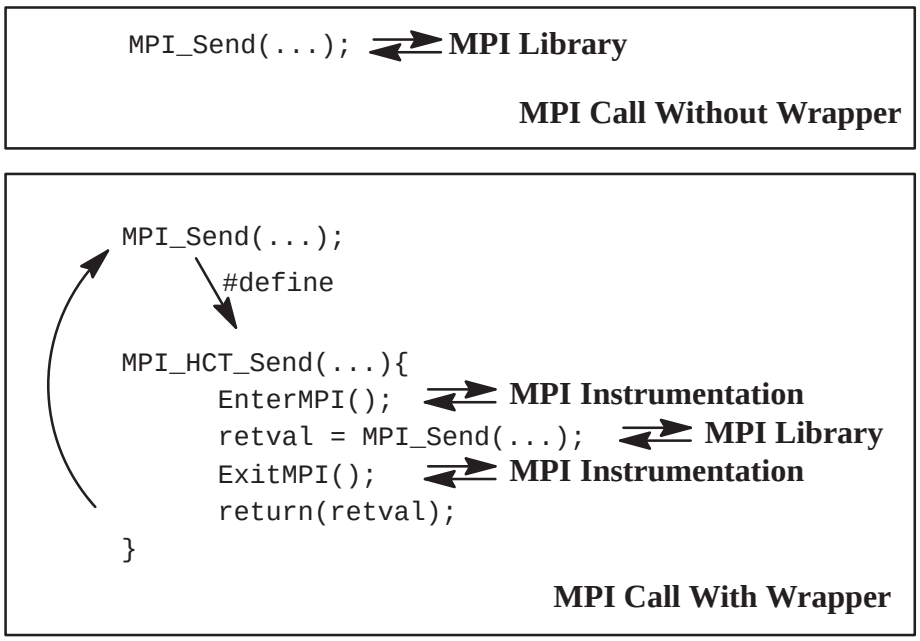

Fig. 1. Wrapper Structure

that only the outermost MPI function invocation was timestamped. This also fixed a bug in the CPU–time–gathering infrastructure.

The wrapper itself works as follows. First, when the HCT_RT_Tag() function is called, the user–defined integer and character string are copied into local static variables and a flag is set. Second, when EnterMPI() is called, the flag is checked and, if set, a timestamp is obtained (using gettimeofday()) and the information (timestamp, integer, and string) are sent over the "real–time socket" via a call to write(). Third, when ExitMPI() is called, a process similar to EnterMPI() ensues and then the flag is reset.

3 An Infrastructure to Gather Information

Once the infrastructure was designed to emit timestamped data, the means of collecting the data centrally had to be developed.

3.1 Using Redirection

A system had already been developed to redirect stdout, stderr, and stdin to and from parallel MPI jobs. The redirection system not only collects the stdout correctly, it also supports task migration correctly, and so the stdout stream (and stdin and stderr) is maintained even as tasks migrate. The original purpose of the redirector was to support users who wanted to invoke parallel jobs inside a shell script and who used redirection in legacy shell scripts.

A redirected task works by closing the stdin, stdout, and stderr file descriptors, opening sockets back to the central collection task (called a "redirector"), and using dup() to make sure the file descriptors match. Special command–line arguments notify the task of the hostname and port number of the redirector. By closing and opening the descriptors, all subsequent calls (such as printf and scanf) work correctly. The central collection

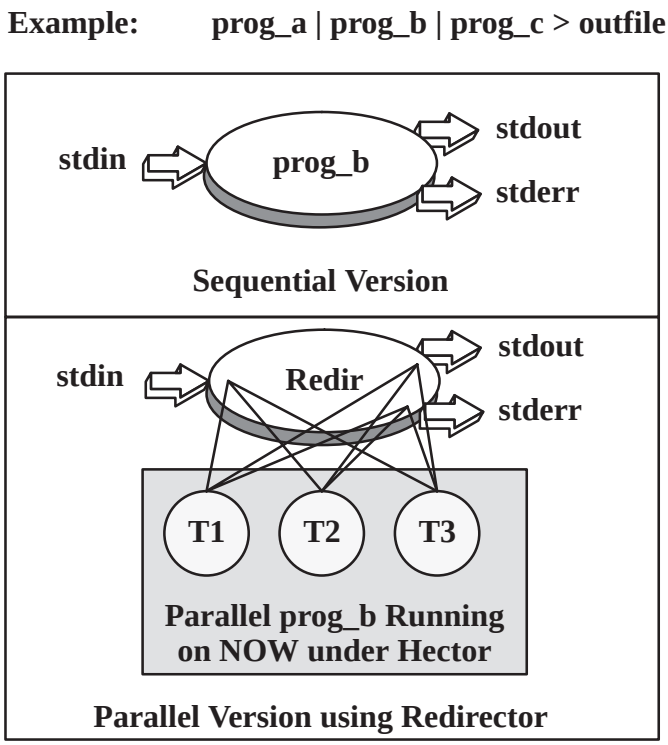

Fig. 2. Example Using the Redirector

task also submits the original job–launch request to Hector, and so acts as a single Unix command–line point of interface. The structure and an example is shown below in Fig. 2.

3.2 Adding the Real–time Stream

The redirector and the tasks were modified to add a fourth stream, nicknamed the "stdtim" stream. Since the opening of sockets was already supported, only incremental changes had to be made to add the real–time socket. Sending is accomplished by converting the timestamp and other information to a single ASCII string and calling a write().

The redirector polls to make sure all sockets have been connected and, once accomplished, polls the sockets and redirects the information streams. The current version of the redirector sends the real–time timestamp information to stdout for ease of debugging. Future versions will send the stream to a real–time scheduler and/or modified optimizer.

3.3 A Complete Example

A simple matrix multiply program was modified (by adding 6 HCT_RT_Tag() calls) to show how real–time information appears to the user. HCT_RT_Tag calls were inserted before several MPI calls, and a distributed version was launched on several workstations under Hector. The complete stdout and stdtim stream was redirected into a file that shows the results below in Fig. 3. The timestamped information is shown in boldface.

Each timestamp has the following format:

Task *Number Enter/Exit***:** *String Integer Timestamp*

where *Number* is the task ID of the task sending the timestamp, "*Enter*" or "*Exit*" indicates whether the timestamp was generated entering or exiting the MPI library, *String* is

```
0:  There are 3 other processes
Opening matrix file mm.small
Current working directory = /projects/cps/hector/HCT2.1/taskalloc/mi-
grate/testing
Task 0 Enter:Sending assignment...        0       913757003.257503
Task 0 Exit :Sending assignment...        0       913757003.258122
Task 0 Enter:About to bcast N and K...  1       913757003.259126
0: Got Matrix of 100 X 100 X 100
Task 0 Exit :About to bcast N and K...  1       913757003.259828
Task 0 Enter:Bcast some rows...        2       913757003.426740
Task 0 Exit :Bcast some rows...        2       913757003.427490
Task 0 Enter:About to send some rows... 3       913757003.777638
This is 0.....Multiplying matrices!
Task 0 Exit :About to send some rows... 3       913757003.778076
Task 0 Enter:About to send some rows... 4       913757003.789029
Task 0 Exit :About to send some rows... 4       913757003.789413
This is 0.....Freeing A and B
Task 0 Enter:About to recv some rows... 5       913757004.496749
*--*0:A rows*--*0:A*--*0:B rows*--*0:BThis is 0.....Ready to receive
results from 1
Task 0 Exit :About to recv some rows... 5       913757004.617608
This is 0.....Got results from 1
Task 0 Enter:About to recv some rows... 6       913757004.652495
Task 0 Exit :About to recv some rows... 6       913757004.652667
This is 0.....Ready to receive results from 2
This is 0.....Got results from 2
Start is 2.375451
End is   3.865196
```

Fig. 3. Actual Redirected Output Stream

the user–specified string, *Integer* is the user–specified number, and *Timestamp* is the machine local time in seconds since the epoch. On the Sun workstations used in testing, time was available with microsecond resolution.

4 Architecture of a Complete Real–time Scheduler

The modifications that have been made and tested enable the design of a complete real–time scheduler for distributed computing. Modifications and additions are needed to the optimizer and master allocator.

4.1 Modifying the Optimizer

A real–time scheduler for Hector is under development at Florida A&M University. The scheduler will use user–specified priority information and real–time timestamp information to schedule, optimize, and allocate MPI tasks.

One example of how the timestamp information could be used is to diagnose that the margin by which real–time tasks are meeting specific deadlines is decreasing. This would cause the optimizer to suspend less important tasks so that real–time tasks have more processor power at their disposal. This is very similar to the approach taken in [6], but more general–purpose in the sense that only very minor code modifications are needed and very broad classes of applications are supported.

In the example, Hector would have several ways to respond to a real–time task with a declining performance margin. First, it could suspend less important tasks. It can do this by migrating the less important tasks to slower machines, by suspending them to disk, or by signalling them (which is akin to control–Z). Hector already supports these methods, and so it is simply a matter of giving the real–time monitor the authority to issue the appropriate commands. Second, it could move the real–time tasks to faster machines. This latter strategy will only be successful if the time to migrate is small enough.

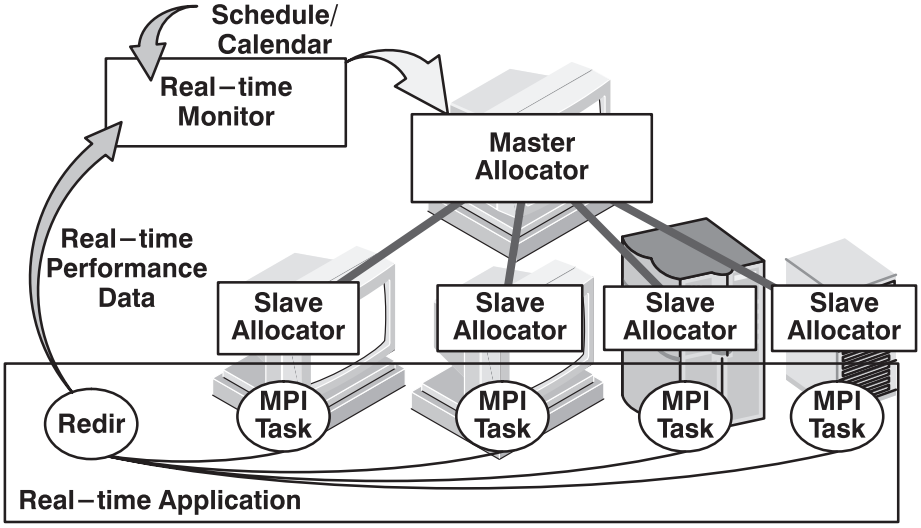

Fig. 4. Hector Running an Application with Real–time Support

4.2 Assembling the Pieces

In Hector, the optimizer is periodically invoked by the central controller (which is called the "master allocator" or "MA"). The optimizer is simply a function that is periodically called and is therefore not persistent. This design was intentional, so that it could easily be tested and replaced. An additional real–time monitor will be needed to collect all of the real–time streams and compare the information to some sort of schedule or calendar, and the modified optimizer can communicate with the real–time monitor. An example of Hector running with complete real–time instrumentation and optimization is shown below in Fig. 4. (More information about Hector's architecture and capabilities may be found in the Appendix at the end of this article.)

The real–time monitor can be implemented either as a standalone process that issues commands to the master allocator or as a separate thread of the MA. The latter is more efficient but requires more extensive modifications. Specifically, the MA is intentionally single–threaded and acts as a point of synchronization for many protocols.

5 Status, Future Work, and Conclusions

As described above, timestamping, sending, and collecting the real–time stream has been implemented and tested. Hector has already demonstrated several sophisticated migration and suspension mechanisms that can be used to remediate performance degradation. The modified optimizer and the creation of a real–time monitor are under development now.

With minimal modifications, a runtime system for MPI programs on networked workstations has been extended to timestamp and collect information about the wall–clock time of MPI events. This infrastructure can enable real–time scheduling of MPI programs with minimal source–code modifications on commodity workstations, and Hector's ability to suspend jobs can be used to free up resources for real–time applications automatically.

References

1. Mark A. Baker, Geoffrey C. Fox, and Hon W. Yau, ''Cluster Computing Review'', Northeast Parallel Architectures Center, Syracuse University, 16 November 1995. Available via *http://www.npac.syr.edu/techreports/hypertext/sccs–0748/cluster–review.html*.
2. Samuel H. Russ, Jonathan Robinson, Dr. Brian K. Flachs, and Bjorn Heckel, ''The Hector Distributed Run–Time Environment'', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 11, November 1998, pp. 1104–1112.
3. Samuel H. Russ, "An Architecture for Rapid Distributed Fault Tolerance", 3rd International Workshop on Embedded High–Performance Computing, in J. Rolim, Editor, *Parallel and Distributed Processing*, Lecture Notes in Computer Science, vol. 1388, Springer–Verlag, 1998, pp. 925–930.
4. Samuel H. Russ, Brad Meyers, Chun–Heong Tan, and Bjørn Heckel, ''User–Transparent Run–Time Performance Optimization'', *Proceedings of the 2nd International Workshop on Embedded High Performance Computing*, Associated with the 11th International Parallel Processing Symposium (IPPS 97), Geneva, April 1997.
5. James Harden, Cedell Alexander, Donna Reese, Marlene Evans, and Charles Hudnall, "In Search of a Standards–Based Approach to Hybrid Performance Monitoring," , *IEEE Parallel and Distributed Technology*, Winter, 1995, pp. 61–71.
6. Rakesh Jha, Mustafa Muhammad, Sudhakar Yalamanchili, Karsten Schwan, Daniela Ivan–Rosu, and Chris de Castro, "Adaptive Resource Allocation for Embedded Parallel Applications", *Proceedings of the International Conference on High–Performance Computing*, 1996, pp. 425–431.

Appendix: About Hector

The Hector Distributed Run–Time Environment has been under development at Mississippi State University for about four years. During that time it has demonstrated numerous capabilities relevant to running MPI–based parallel programs on networked resources.

First, it can migrate tasks from one workstation to another. This capability is essential in order to harness idle workstation resources, so that workstations can be released back to their "owner" on demand. It has been tested both for reliability and speed. Time to migrate is dominated by the time to transfer the program state over the network, and is therefore program size divided by network bandwidth.

Second, it can use its migration capability to suspend tasks onto remote disks. This can be used to create "backup copies" of programs that are running in order to provide fault tolerance and to free up resources.

Third, it has an extensive information–gathering infrastructure to remain well–informed about platforms and applications. Each candidate platform runs a small utility task called a "slave allocator" or SA. The SA obtains system–level information from the machine's kernel and application–level information from each task's self–instrumentation and forwards the information periodically to the central master allocator or "MA".

Fourth, it can optimize program allocation by taking into account how busy each machine is, the relative CPU speeds, and other performance–relevant information. One recent addition is an affinity function that tends to map tasks from the same program onto the same machine when possible. The MA performs this optimization, and so is a repository for global performance information.

The MA also acts as a point of synchronization for certain protocols, such as the migration and termination protocols. Its single–threaded, event–driven design make such synchronization straightforward.

Fifth, Hector provides these services transparently (with no source–code modifications) to MPI programs by a specially modified MPI implementation. The implementation permits consistent checkpointing and the ability to switch to shared–memory when two tasks share a machine.

More information about Hector may be found on the Hector Home Page at **`http://www.erc.msstate.edu/thrusts/cps/hector/main/index.html`**.