Chapter 1
Performance Analysis of MPI Programs *

Ed Karrels[†] Ewing Lusk[‡]

Abstract

The Message Passing Interface (MPI) standard has recently been completed. MPI is a specification for a library of functions that implement the message-passing model of parallel computation. One novel feature of MPI is its very general "profiling interface," that allows users to attach assorted profiling tools to the MPI library even though they do not have access to the MPI source code. We describe the MPI profiling interface and describe three profiling libraries that make use of it. These libraries are distributed with the portable, publicly available implementation of MPI.

1 Introduction

Parallel programs are more difficult to understand than sequential ones. It is common to conceive of a parallel algorithm, implement it, and then be puzzled by its disappointing performance, even though the program executes correctly. What is then needed is an instrumentation of the program, so that data can be collected that will lead to an *understanding* of the program's behavior.

The difficulty is that instrumenting all of the system calls is likely to be cumbersome and error prone, while any specific profiling mechanism provided by the vendor of the parallel computer or the parallel library is unlikely to be just exactly what a user wants. This is particularly the case if the user needs to tailor the profiling data to a specific application, such as when custom run time or post mortem graphical displays are used.

1.1 The MPI Message Passing Interface Standard

During 1993 and early 1994, a broadly-based group of parallel computer vendors, library writers, and application specialists met regularly to define a standard for message-passing libraries. The Standard has just been finalized (April, 1994) and is widely available [4]. A number of implementation efforts are under way, both public and proprietary, and porting of applications has begun. The MPI Forum, as it called itself, was eager to provide profiling functionality within the Standard, but believed it premature to standardize the profiling tools itself. In this paper we explain a solution to this problem, which was adopted as part of the MPI Standard. We then explain two ways to implement the specification, and illustrate the use of the MPI profiling interface with three profiling libraries that can be used with any conforming MPI implementation.

*This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under contract W-31-109-Eng-38.

[†]Computer Science Department, University of Wisconsin, Oshkosh

[‡]Mathematics and Computer Science Division, Argonne National Laboratory

2 The MPI Profiling Interface

In this section we describe the MPI profiling interface as it is defined in [4] and summarize two different methods for implementing it.

2.1 Definition

The idea behind the MPI profiling interface (due to James Cownie) is to use the linker to substitute a user-written "wrapper" function for the MPI function called by the application. In order to do this it must have the same name. Since we will assume that the author of the profiling library does not have access to the MPI source code, the profiling version of the MPI function must have a way to call the *real* MPI function to actually do the required task. The problem is that it has the same name as the profiling version. The way out of this dilemma is to require that every MPI function also be callable by a different name. The MPI definition requires that every MPI function `MPI_Xxx` also be callable by the name `PMPI_Xxx`. Then the profiling version of `MPI_Bcast`, say, called by the application, can itself call `PMPI_Bcast` to actually do the broadcast, performing whatever profiling work is desired before and after the call to `PMPI_Bcast`.

We also don't want the author of the profiling library to have to write profiling versions of *every* MPI routine. We would like the linker to use the profiling version of an MPI function if it has been defined, and a non-profiling version otherwise. There are at least two ways to do this: plain and fancy.

2.2 Plain implementation

One very straightforward way to solve the problem, requiring no unusual features in the compiler and linker, is just to build two complete copies of the MPI library, one in which every function has the prefix `MPI` and the other in which every function has the prefix `PMPI`. Then if the user code, profiling library, "PMPI" library, and MPI library are presented to the linker in that order, references will be resolved as in Figure 1. Here the profiling library

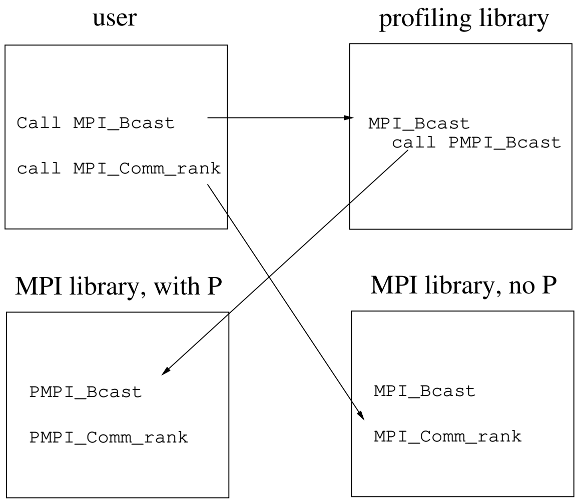

FIG. 1. *Link libraries for profiling*

has defined an alternative definition of `MPI_Bcast`, in order to intercept calls to it from the user, but has not provided an alternate definition for `MPI_Comm_rank`.

2.3 Fancy Implementation

If the compiler and linker support "weak" external symbols, as some do, then we can achieve the same effect without having essentially duplicate libraries. The declaration

```
#pragma weak MPI_Bcast = PMPI_Bcast
int PMPI_Bcast( ... )
{
    ...
}
```

defines `MPI_Bcast` to be `PMPI_Bcast` if there is no other definition of `MPI_Bcast`, but the linker does not complain if there is.

2.4 User Control of Profiling

The author of the profiling library needs a way to provide some control over the behavior of the profiling library to the application program. Since profiling libraries will have quite different types of controls, there is no way to completely standardize the control calls. MPI does require the existence of a single library function `MPI_Pcontrol`, which has an undefined, variable-length argument list. This function doesn't do anything, but, just like the other MPI functions, can be redefined in order to provide whatever controls the profiling library needs.

3 Three Profiling Tools for MPI programs

In this section we describe three profiling libraries that are supplied with the public, portable implementation of MPI being developed at Argonne National Laboratory and Mississippi State University. Two of them rely in turn on another set of routines, called the MPE library (for Multi-Processing Environment) that is part of the public MPI as well. Parts of MPE started life as part of Chameleon [1].

The function of the MPE library is to serve as a companion to MPI (It uses MPI for communication) and that supplies useful functionality, not part of the MPI standard, that is useful in creating a parallel programming environment. Two aspects of this functionality that we exploit here are event logging and simple graphics. The relevant MPE functions are described where used below.

3.1 Accumulation of Time Spent in MPI Routines

The first profiling library is simple. The profiling version of each `MPI_Xxx` routine just calls `PMPI_Wtime` (which delivers a time stamp) before and after each call to the corresponding `PMPI_Xxx` routine. The times are accumulated in each process and written out, one file per process, in the profiling version of `MPI_Finalize`. The files are then available for use in either a global or process-by-process report. This version does not take into account nested calls, which occur when `MPI_Bcast`, say, is implemented in terms of `MPI_Send` and `MPI_Recv`.

3.2 Logfile Creation and Teeshot

The second profiling library generates *logfiles*, which are files of timestamped events. During execution, calls to `MPI_Log_event` are made to store events of certain types in memory, and these memory buffers are collected and merged in parallel during `MPI_Finalize`. During execution, `MPI_Pcontrol` can be used to suspend and restart logging operations. The logfile produced at the end can be analyzed by a variety of tools. One that we use is called Teeshot, which is a derivative of Upshot [3], written in Tcl/Tk. A screen dump of Teeshot in use is shown in Figure 2. It shows parallel time lines with process states, like one of

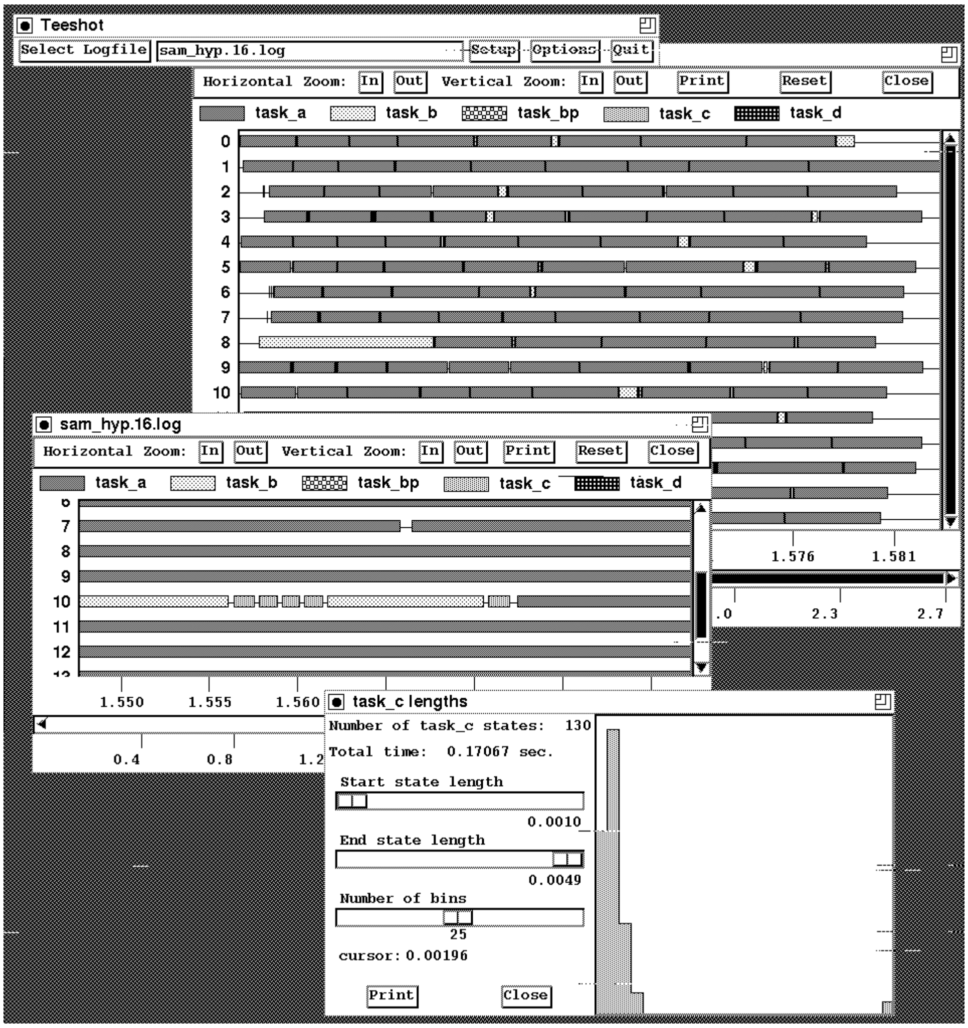

FIG. 2. *A screendump from Teeshot*

the paraGraph [2]. The view can be zoomed in or out, horizontally or vertically, centered on any point in the display chosen with the mouse. In Figure 2, the middle window has resulted from zooming in on the upper window at a chosen point to show more detail. The window at the bottom of the screen show a histogram of state durations, with several adjustable parameters.

3.3 Real-Time Animation

The third library does a simple form of real-time program animation. The MPE graphics library contains routines that allow a set of processes to share an X display that is not particularly associated with any one specific process. Our prototype uses this capability to draw arrows that represent message traffic as the program runs. An example snapshot is shown in Figure 3. Here a parallel Mandelbrot computation is proceeding, with output of the computation being drawn in one MPE window while the message pattern is being drawn in another MPE window. (In this case the manager-worker scheduling algorithm is apparent; the manager process has been stopped to take the screen dump, and animation quickly has reached the state in which every other process has completed it current task and has sent a message to the manager that it has not yet received, precisely because it has been stopped.)

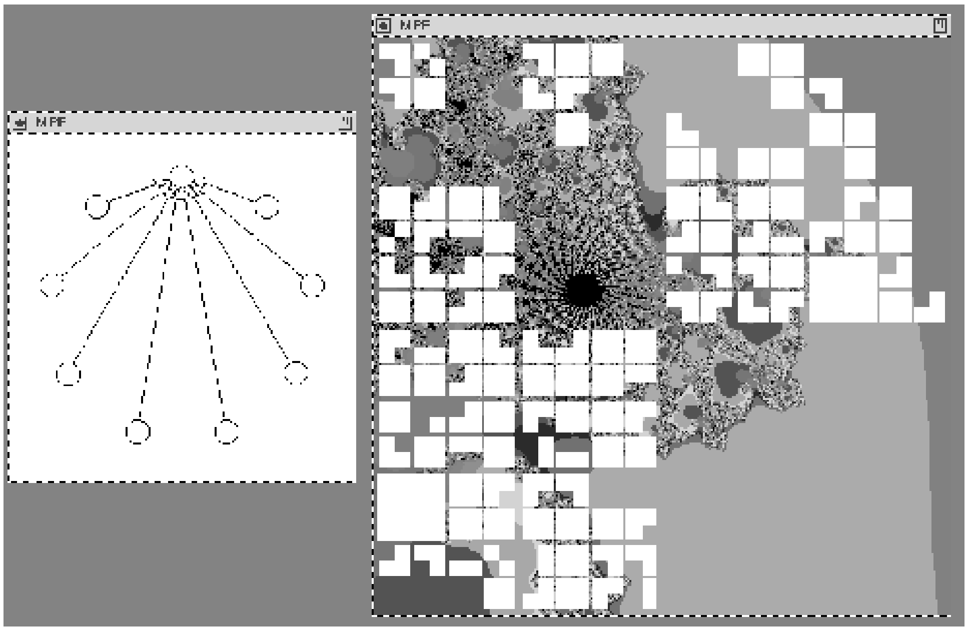

FIG. 3. *Program Animation with the MPI Profiling interface*

3.4 Automatic Generation of Profiling Libraries

For each of these libraries, the process of building the library was very similar. First, profiling versions of `MPI_Init` and `MPI_Finalize` must be written. The profiling versions of the other MPI routines are quite similar in style. The code in each looks like

```
   int MPI_Xxx( . . . )
{
       do something for profiling library
    retcode = PMPI_Xxx( . . . );
       do something else for profiling library
    return retcode;
}
```

We generate these routines by writing the "do something" parts only once, in schematic form, and then wrapping them around the `PMPI_` calls automatically. It is thus extremely easy to generate profiling libraries.

4 Summary

Performance analysis is critical for the understanding of the behavior of parallel programs. A wide variety of research projects continue to explore a range of approaches to performance analysis. The MPI message passing interface standard has partially filled the needs of users and performance analysis researchers by providing not a specific performance analysis tool but rather a standard interface that can host a wide variety of approaches to performance analysis. Three tools have been presented here: simple time accounting, logfile creation and examination, and runtime animation.

References

[1] W. D. Gropp and B. Smith, *Chameleon parallel programming tools users manual*, Tech. Rep. ANL-93/23, Argonne National Laboratory, Mar. 1993.

[2] M. T. Heath, *Recent developments and case studies in performance visualization using ParaGraph*, in Performance Measurement and Visualization of Parallel Systems, G. Haring and G. Kotsis, eds., Amsterdam, The Netherlands, 1993, Elsevier Science Publishers, pp. 175–200.

[3] V. Herrarte and E. Lusk, *Studying parallel program behavior with* `upshot`, Tech. Rep. ANL–91/15, Argonne National Laboratory, Argonne, IL 60439, 1991.

[4] Message Passing Interface Forum, *MPI: A message-passing interface standard*, Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. (To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994).