Visualization Tool for Debugging Pilot Cluster Programs

by

Tianyi Bao

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Master of Science
in
Computer Science

Guelph, Ontario, Canada

© Tianyi Bao, December, 2016
ABSTRACT

VISUALIZATION TOOL FOR DEBUGGING PILOT CLUSTER PROGRAMS

Tianyi Bao
University of Guelph, 2016

Advisor: Professor W. B. Gardner

This thesis presents the adaptation and integration of MPI Parallel Environment (MPE) as a logfile producing library and Jumpshot-4 as a visualization tool to support a log visualization facility for the Pilot library. Pilot is a parallel programming library written in C that forms a thin layer on top of standard MPI (Message Passing Interface). Pilot does not require other third-party libraries, and it is designed particularly for novice programmers to keep them away from common MPI pitfalls. This is done through using high-level abstractions, referenced from Communicating Sequential Processes (CSP), to form a process-and-channel programming model. Due to non-deterministic execution order, among other reasons, parallel programs are more difficult to debug than sequential programs. After our research, we find that this can be helped by the use of MPE and Jumpshot-4, and their characteristics meet our requirements to trace a Pilot program's execution visually and aid debugging. This can help beginners to understand the actual run-time message-passing between processes more clearly and serve as a pedagogical aid for parallel programming instructors.
Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Dr. William Gardner, a responsible, respectable and resourceful mentor, who has provided me with priceless guidance and consistent encouragement in study and life throughout every stage of my entire master career. Without his enlightening instruction, impressive kindness and extraordinary patience, I cannot come this far and complete this thesis. Furthermore, I also would like to thank my advisory committee member Dr. Charlie Obimbo for the valuable assistance and sound suggestions on improving my thesis writing. Besides, I shall extend my thankfulness to all the mates and staff in SoCS. It is an unforgettable experience of learning and working with so many fantastic people during my master program. My sincere appreciation also goes to the kindhearted people in Parkview Church who keep praying for me in my trying periods. At last but not least, I would like to thank my family, especially my parents who provided me the opportunity to live and study in Canada, and give me support both in material and spiritual. I am overwhelmed with gratitude to them for their unconditional help and infinite love.
# Table of Contents

Acknowledgments ................................................................. iii

Table of Contents ............................................................... iv

List of Tables ................................................................. vi

List of Figures ............................................................... vii

Chapter 1 Introduction .................................................... 1

1.1 Motivation ............................................................... 1
1.2 Thesis statement ....................................................... 4
1.3 Thesis outline ........................................................... 4

Chapter 2 Pilot Background and Literature Review .................. 5

2.1 Terminology ............................................................. 5
2.2 Pilot ................................................................. 7
   2.2.1 Abstractions .................................................. 8
   2.2.2 Support for debugging ...................................... 11
2.3 Current visualization tools for parallel programming on MPI .... 13
   2.3.1 Objectives for tool support .................................. 14
   2.3.2 Overview of the tools tested in the experiments ............ 16
2.4 Jumpshot with MPE ................................................... 26
   2.4.1 MPE ............................................................ 27
   2.4.2 Jumpshot ...................................................... 33
   2.4.3 Conclusion ..................................................... 35

Chapter 3 Adapting MPE for Pilot ...................................... 36

3.1 Criteria ............................................................... 36
3.2 MPE’s capabilities with Jumpshot .................................. 36
   3.2.1 What MPE can record and Jumpshot display ............... 41
   3.2.2 Limitations and solutions .................................... 45
3.3 Visual design for displaying Pilot traces in Jumpshot .......... 47
   3.3.1 Plan for using colours ...................................... 47
   3.3.2 Plan for using graphical objects ......................... 50

Chapter 4 Integrating MPE into Pilot ................................ 56

4.1 Pilot as a legacy code ............................................... 56
   4.1.1 Run time option selection .................................. 57
   4.1.2 Source file organization .................................... 57
   4.1.3 Run-time execution phases ................................ 58
Chapter 5 Experimental Results

5.1 Screenshots of all the Pilot calls
5.1.1 Popup windows of state rectangles
5.1.2 Popup windows of message-passing arrows
5.1.3 Popup windows of event bubbles associated with a state
5.1.4 Popup windows of event bubbles not connected with a state
5.2 Demonstration application
5.2.1 Demonstration of the main execution time
5.2.2 Demonstration of MPE log file wrap-up time
5.3 Real instances of using MPE and Jumpshot to help debugging

Chapter 6 Conclusion and Future Work

6.1 Summary
6.2 Future work

Bibliography

Appendix A: Building and Using MPE with Pilot

A.1 Building Pilot with MPE
A.1.1 Installing MPE
A.1.2 Building Pilot
A.2 Building Pilot applications with MPE
A.3 User guide
List of Tables

Table 2-1. The result of each criterion for the experimented MPI visualization tools ............... 17
Table 2-2. Standard logfile formats and their associated tools .................................................. 18
Table 2-3. Basic MPE function calls for producing a CLOG-2 logfile ........................................ 30
Table 3-1. Colour plan for state rectangles and event bubbles ...................................................... 50
Table 4-1. The approach of logging the required data into the corresponding popup windows .... 71
Table 5-1. Execution times of the thumbnail application program .............................................. 96
Table 5-2. Execution times for MPE logfile wrap-up .................................................................... 99
List of Figures

Figure 2-1. Pilot’s process/channel architecture ................................................................. 11
Figure 2-2. A Pilot bundle for broadcast use ........................................................................ 11
Figure 2-3. Sample logfile for a Pilot program................................................................. 12
Figure 2-4. User Interface of DDT ...................................................................................... 20
Figure 2-5. User Interface of Vampir [Bur07] ...................................................................... 21
Figure 2-6. User Interface of Pajé ....................................................................................... 23
Figure 2-7. User Interface of ViTE ....................................................................................... 24
Figure 2-8. User Interface of Jumpshot reading a SLOG-2 file converted from a .trc file ...... 25
Figure 2-9. Jumpshot logfile converter .............................................................................. 34
Figure 2-10. Display of an MPI program in Jumpshot ....................................................... 34
Figure 2-11. Display an MPI program in Jumpshot (zoomed in) ....................................... 35
Figure 3-1. A detail popup window of state ....................................................................... 42
Figure 3-2. A detail popup window of event ...................................................................... 43
Figure 3-3. A detail popup window of message passing ..................................................... 43
Figure 3-4. A detail popup window of message passing ..................................................... 44
Figure 3-4. Legend window ............................................................................................... 44
Figure 4-1. A popup window in Jumpshot running in Java version 1.6 ....................... 73
Figure 4-2. A popup window in Jumpshot running in Java version 1.8 ....................... 73
Figure 4-3. Picture of distorted bubble and arrow positions .............................................. 75
Figure 4-4. Picture of showing collective function using usleep() .................................... 76
Figure 5-1. Demonstration program for displaying all Pilot calls in Jumpshot (Home size) ...... 83
Figure 5-2. Demonstration program zoomed to PI_Configure ........................................... 83
Figure 5-3. Demonstrating PI_Write, PI_Select, and PI_Read ........................................ 83
Figure 5-4. Demonstrating PI_Broadcast and PI_Gather .............................................. 84
Figure 5-5. Demonstrating PI_Scatter and PI_Reduce ...................................................... 84
Figure 5-6. Popup window of PI_Configure ....................................................................... 86
Figure 5-7. Popup window of Compute .............................................................................. 86
Figure 5-8. Popup window of PI_Write ............................................................................... 87
Figure 5-9. Popup window of PI_Read ............................................................................... 87
Figure 5-10. Popup window of PI_Broadcast ................................................................. 87
Figure 5-11. Popup window of PI_Gather ........................................................................... 88
Figure 5-12. Popup window of PI_Scatter ......................................................................... 88
Figure 5-13. Popup window of PI_Reduce ......................................................................... 88
Figure 5-14. Popup window of PI_Select .......................................................................... 89
Figure 5-15. Popup window of message passing (main to worker) ..................................... 89
Figure 5-16. Popup window of message passing (from worker to main) .......................... 89
Figure 5-17. Popup window of PI_Write bubble ............................................................... 90
Figure 5-18. Popup window of PI_Read bubble ................................................................. 90
Figure 5-19. Popup window of PI_Broadcast bubble ....................................................... 90
Figure 5-20. Popup window of PI_Gather bubble ............................................................. 91
Figure 5-21. Popup window of PI_Scatter bubble ............................................................... 91
Figure 5-22. Popup window of PI_Reduce bubble ............................................................... 91
Figure 5-23. Popup window of PI_StartTime ..................................................................... 92
Figure 5-24. Popup window of PI_EndTime ...................................................................... 92
Figure 5-25. Popup window of PI_TrySelect ..................................................................... 92
Figure 5-26. Popup window of PI_ChannelHasData ........................................................ 93
Figure 5-27. Popup window of PI_Log ............................................................................... 93
Figure 5-28. The thumbnail application running with 10 processes shown in Jumpshot............. 94
Figure 5-29. The thumbnail application shown in Jumpshot (zoomed in)............................ 95
Figure 5-30. Instance A of student’s program in Jumpshot..................................................... 100
Figure 5-31. Instance B of student’s program in Jumpshot..................................................... 100
Chapter 1

Introduction

1.1 Motivation

High-performance computing (HPC) has been applied in many areas of scientific research on account of its ability in quick, large-scale calculation. Commonly using one or more computer clusters connected by network, HPC programming can be distributed among these processors through message passing to execute calculations. HPC is focused on clusters, since clusters are more scalable than shared memory computers. With a shared memory computer having four, eight or sixteen cores, overhead involved in synchronization and one common bus for memory access can become a bottleneck to prevent improvement of speedup. In order to utilize hundreds or thousands of processors for HPC, parallel programmers are going to be committed to message passing programming. Thus, message passing programming plays a significant role in the communication between processors in clusters.

However, to debug HPC programming is more difficult than sequential programming due to nondeterministic execution order causing bugs to show up randomly, and to poor observability of program execution. Also, compared with sequential programming, HPC programming needs to trace both control flow, and interprocess synchronization and data communication to diagnose deadlocks and other timing and synchronization errors.
Besides the former two reasons, traditional debugging techniques like inserting printf statements become useless because they may affect execution order and mask the bug.

For the sake of providing message passing service on computer clusters, and developing debugging or any other beneficial facilities to improve the performance of parallel programs, a library with highly approved guideline is required as a foundation. Message Passing Interface (MPI) library, a traditional lingua franca of HPC, forms a standardized and portable protocol. By stipulating the syntax and semantics in its application programming interface (API), MPI is beneficial for developing portable message-passing applications in varied computer programming languages such as C, C++, and Fortran. MPI supports both point-to-point and collective communication, and could be applied in both distributed-memory systems and shared-memory systems. Aiming for high performance, portability, and scalability, versions of MPI can be installed on most distributed architectures. Furthermore, the MPI library has associated tools to visualize program execution for debugging, and profiling tools to improve correctness and performance of parallel programs.

Though the MPI library has been in long-term use and is regarded as a widely adopted standard for HPC, its tedious function and high possibility of misusing API calls make the new parallel programer flinch. Therefore, the Pilot library was developed to address these drawbacks. Pilot is a parallel programming library written in C that forms a thin layer on top of the standard MPI library. Pilot does not require other third-party libraries. It is designed as an educational tool to introduce the novice parallel programmers about message passing by shielding them from dealing with low-level communication issues. Considering that Pilot is oriented to beginners in parallel programming, it would be
particularly useful if there were visualization tools for debugging Pilot programs similar to those supporting MPI programmers.

Currently, Pilot has the capability via command line option to generate its own trace file. However, the records are in coded text and not user friendly to read and interpret. Hence, Pilot in its present form is not helpful enough for the novice programmers to have an intuitive view to see how their parallel programs work and message passing performs at run time.

While there are a number of existing visualization tools for parallel programming, they may not align with the features of Pilot, making them difficult to apply. Therefore it will be necessary to find or develop an appropriate tool to aid novice Pilot programmers, and this tool should satisfy the following requirements:

1. **Visualizer in Pilot’s own terms instead of MPI’s.** Since most visualization tools for parallel programming are designed for MPI, they can only identify and present MPI but not Pilot’s functions in pictures. However, for the novice parallel programmer, it is important to observe the operation of Pilot functions corresponding to their code.

2. **Debugging more than profiling.** The current visualization tools for parallel programming are more focused on large data analysis and profiling; however, since Pilot is targeted at novice parallel programmers, the functional requirement of the tool should be more concentrated on debugging than performance profiling.

3. **Freeware to go along with free/open source Pilot.** Due to the limitation of availability, a commercial tool may not be suitable for novice programmers in an educational environment to use. Also it can cause difficulties in future research with commitments to buy licenses and the possibility of a product being discontinued.
1.2 Thesis statement
Since the Pilot library is positioned as an educational tool for students in parallel program-
ing, a reliable visualization tool for debugging Pilot programs is needed. Finding or
developing an applicable visualization tool aligned with the features of the Pilot library, and
showing the appropriate way to adapt it to Pilot, can make a great contribution of aiding
novice parallel programmers to better comprehend the procedure of message passing.

1.3 Thesis outline
In Chapter 2, some background knowledge and related work will be covered including the
advantages and disadvantages of various existing MPI visualization tools. As a result of this
extensive research, the library MPI Parallel Environment (MPE) and its associated
viewer Jumpshot-4 were identified as good candidates. In Chapter 3, how to adapt MPE
for Pilot will be described and analyzed. Chapter 4 presents the details of integrating MPE
into Pilot, made up of user interface, software design, problems and solutions, and, finally,
comparing and contrasting Pilot’s native logging facility. Then Chapter 5 will show the
results and screenshots of visual logs for various Pilot programs, and analysis of the run-
time overhead that using MPE introduces. In Chapter 6, a conclusion will be drawn along
with limitations and future work.
Chapter 2

Pilot Background and Literature Review

For the purpose of achieving the visualization facility to support Pilot programming, the priority is to find out which current visualization tools are already available for message-passing programmers, for example, the ones for supporting MPI programming. Then, after experiments are conducted, the feasibility of utilizing each tool on the Pilot library can be determined to identify if an appropriate one exists. If not, the intended-designed tool can still use the capabilities and features of these tools as a referential model at least.

But first, some necessary terminology is mentioned in Section 2.1. Then in Section 2.2, in order for the reader to understand what is distinctive about Pilot vs. MPI (such that tools for MPI may not be immediately applicable), some more detailed background on Pilot is necessary. Based on this, the abstractions which the tool needs to reflect will be found, and the criteria for a good solution will be listed.

After that, Section 2.3 discusses some current parallel visualization tools for MPI, and finally Section 2.4 summarizes the findings and explains the direction that this research took.

2.1 Terminology

There are four significant terms—logging, tracing, profiling, and event—which need to be explained and distinguished in advance. These are normally reported in the context of some
“process” (in the operating system sense, same as MPI “rank”) they are associated with, and with reference to “time” (program execution time) [SMM13]. Visualization tools commonly display a graphical representation where time increases along the X axis and processes are distributed along the Y axis.

IBM developerWorks makes a nice distinction between logging and tracing: “Logging is typically used for significant events that you want to make note of in your logfiles, while tracing is typically used to record any information that could be useful in debugging problems with your code” [BBJ08].

Generally, logging is active all the time and generates data in low volume. It is to indicate important changes, warnings, or errors for system administrators to check. Tracing is enabled when a suspected problem needs to be diagnosed to find out when and where an error occurs. Nevertheless, it is normally disabled since trace events can occupy high volume in storage space.

In a colloquial sense, “logging” can be considered the general term, and “tracing” a particular case of logging. In the context we are talking about, the significant events are not for the system administrators, they are for the programmers. The kind of “logging” that we are talking about for Pilot is the specific kind called “tracing”.

Profiling is a form of dynamic program analysis from collecting relevant data about the execution of a program [RK15]. The main focus of profiling is to aid program optimization by uncovering the bottlenecks of performance, providing detailed information about source code function contents, and visualizing call graphs of the examined application.
An *event* is often defined as an instance of an MPI function call, but in some visualization tools, event, or more precisely called a solo event, is defined as a note created by programmers to record an important point in time. In that case, the other meaning of event, namely an MPI function call will be replaced by another term “state”. *State* is useful because it includes the notion of duration, not just a single point in time.

### 2.2 Pilot

The Pilot library was created by J. Carter and Dr. Gardner [GCG10, Gar]. They state that Pilot is a C library constructed as a thin layer on top of MPI and does not require other third-party libraries. They also present that the main purpose of Pilot is to instruct and educate novice scientific C programmers to understand message passing programming. Dr. Gardner states five main drawbacks using MPI which cause high possibility to misuse the message-passing functions in the beginning of studying, and explains by what means Pilot overcomes each of them [GC14]: (1) By using high-level abstractions in communication, Pilot prevents the user from being distracted by the low-level communication details in MPI such as tag, rank, and communicators; (2) In contrast with over 120 and over 500 functions in MPI-1 and MPI-2, respectively, Pilot has merely 25 functions, with limited complex features, which can be mastered and applied in a short time by its users; (3) Pilot only uses the Multiple Program, Multiple Data (MPMD) model instead of MPI’s potentially confusing mixed models, MPMD as well as Single Program, Multiple Data (SPMD) where the same collective functions, such as MPI_Bcast, should be called both in sender and receiver; (4) A deadlock detector integrated in Pilot can be activated to diagnose a deadlock-suspected program. Whereas, such problems in an MPI program cannot be easily detected since they need a third-party deadlock checker such as Umpire [VdS00], otherwise it can run to the
end of its time limit without giving any feedback, so beginners are not capable of identifying the difference between a long-running program and a deadlocked program; (5) The Pilot deadlock detector, but not MPI, can also discover any process with undelivered messages to support the user to find their faulty design. In these ways, Pilot programming is advocated as simpler and safer than with MPI.

2.2.1 Abstractions

Process and Channel are two main abstractions of the Pilot library. Both of these abstractions are borrowed from Communicating Sequential Processes (CSP) [GC14, Hoa02] which is a formal model of computation with message passing. Also, care has been taken to keep the API compatible with CSP and resist inflating the library with ad hoc capabilities that break the formalism. A process is real, and it is capable of having behavior that can be executed on one of the nodes in an MPI cluster, which enables hundreds of processes to compute concurrently. A channel is abstract, and acts like a communication medium between any pair of reading and writing processes. There are four characteristics of CSP (and Pilot) channels. The first is point-to-point: A channel is bound to two processes at creation time. The second is one-way: A process writes to the channel and another process reads from it, and the direction of writes and reads in one channel cannot be inverted. If return communication is desired, another channel must be created. The third is synchronous: Execution of the read and write processes only proceed after the message passing is finished. The last is untyped: Any type of data, also any quantity, can be communicated via a given channel. The data type and quantity are expressed using conventions from C’s stdio library (fprintf and fscanf), for example “%100f” means an array of 100 floating point numbers.
Synchronous channel communication, which comes from CSP, is not strongly enforced due to being implemented via MPI_Send, the most common MPI primitive for sending a message. MPI_Send has so-called “eager” semantics (see Chap. 8 of [GLS14] and MPI Message Passing Protocols in [Law14]) in that it prefers to be asynchronous (i.e., the function returns to its caller as soon as the contents has been moved out of the write buffer, even if it is only moved to an MPI buffer and has not physically left the sender's rank) but can fall back to synchronous (the function doesn't return until the message has been received). The behaviour of any given MPI_Send call depends on a number of factors including the size of the message, the current utilization of MPI and operating system buffers, among others. When the programmer activates Pilot's built-in deadlock checker, PI_Write is changed to use MPI_Ssend with its true synchronous communication, or “rendezvous” semantics, at the cost of adding latency in the sending process. Deadlock checking takes place in the context of synchronous channel semantics with the assumption that if a deadlock were detected under those strict conditions, it had the potential to occur under the “loose” semantics of MPI_Send, thus it is dangerous for the programmer to depend on PI_Write behaving asynchronously.

Pilot programmers need not concern themselves with this aspect anymore than MPI programmers typically think about it when coding MPI_Send. But once tracing and visualization are brought into play, programmers will be able to see that in some, even most, instances, PI_Write does return right away and the sending process goes on to do other things, while in other instances it does not; i.e., its run-time behaviour is not consistent.

The third abstraction is the Bundle. A bundle represents a collection of channels for reading or writing that all have the same process endpoint. It allows the Pilot library to
access MPI’s efficient collective operations while not breaking the theoretical process/channel model. Five types of bundles are provided by Pilot: (1) broadcast, for writing the same value to all processes in the collection; (2) scatter, for writing different values to all processes in the collection; (3) selector, for determining which channel in the collection is ready to be read; (4) gather, for reading a distinct value from every process; and (5) reduce, for reading a value from every process while applying a reduction operation (such as sum) over the contributions. Bundles and collective operations do not really add any additional functionality that cannot be obtained with simple PI_Read and PI_Write. However, programmers using them can avoid calling I/O functions repeatedly in loops, to accelerate the time during which data are collected from multiple sources at one time. Bundles can not only save time, but also give an alternative for the structure of a Pilot program.

Using bundles or not can categorize Pilot application structure into two styles. The basic one without using bundles, as shown in Figure 2-1, is made up of processes, represented by clouds, including a master and a number of workers, and channels, represented by arrows, connecting the master process and each work process. In addition, the direction of the arrow represents the direction of message passing. The other style using bundles, as shown in Figure 2-2, employs a dashed ring to represent a bundle for the channels sharing the same endpoint, the master process, for writing.

Pilot programs are organized in two phases to utilize this process/channel model. The first is the configuration phase, executed on all MPI ranks, where the application architecture comprised of processes, channels, and bundles is defined. Then, the second is the execution phase where all the calculations and communications are performed by their respective processes, namely, the individual MPI ranks.
2.2.2 Support for debugging

Based on this process/channel model, Pilot also has its own specific diagnostic facilities for debugging. Pilot debugging facilities for message passing are mainly made up of the Pilot deadlock detector, mentioned in Section 2.1, and runtime error-checking at a user-specified level. The purpose of the latter is to detect erroneous use of the API and abort the program.
with a diagnostic message including source file and line number. Error checking has four levels to be selected depending on what degree of debugging users need, and it can be set by "-picheck=n" in command line or a global variable PI_CheckLevel. These two facilities help the user with disclosing errors in message passing. Besides the former two approaches, Pilot also provides a logging facility, described next.

Pilot has the capability to log traces of all channel communications as a debugging aid. This facility is also used internally to update a dependency matrix of process/channel communications when deadlock detection is enabled.

The traces can record and distinguish each process, channel, and bundle. It is able to record the process number, the time, and API function call. It also records channel or bundle number, data type (in terms of Pilot format code), memory address, the number of array elements, and the first element in array.

```
000971 P 3 FIN 0
001013 C 0 Wri 1 pilot_test.c:64 %d [1] 3
001025 C 2 Rea 1 pilot_test.c:40 %d [1] 0x7fff76376e70
001031 P 0 FIN 0
001043 P 2 FIN 0
```

**Figure 2-3.** Sample logfile for a Pilot program

Figure 2-3 shows an example of Pilot’s native logfile which is recorded from a simple Pilot program. The nine columns from left to right represent: (1) time of the event in nanoseconds from the start of execution. This is actually the time that the report arrives at the logging process, not necessarily the time of the event in the reporting process, since the clocks of the cluster nodes are not synchronized. (2) Event type. The typical type is “C”
standing for an API call by the user application. Type “U” (not shown) means that the user called PI_Log to place an arbitrary message in the log. Type “P” is an internal status message from Pilot. Types reserved for future use are “T” (dump of Pilot tables) and “S” (statistics on API calls, messages sent, etc.). These latter have not been implemented. (3) MPI rank. It is the same as the process’s number, for reporting where an event happened. (4) Abbreviated name of event. For example, “Rea”, “Wri”, and “FIN” mean the abbreviation of PI_Read, PI_Write, and process is finished. (5) Channel or bundle number for which the Pilot I/O function is called. (6) Source file name and line number in the Pilot program where the Pilot function is called. (7) Data type that the programmer coded as a format string. (8) Length of a the data array. (9) First array element value (for sending function such as PI_Write) or buffer address (for receiving function such as PI_Read) in an array.

As shown above in Figure 2-3, the logfile is just textual, with its own format not conforming to any standard. Its contents are obscure to understand, and its event times are not very accurate. Hence, Pilot without a standard logfile and visualization tool is not helpful enough for the novice programmers to have a intuitive view to see how their parallel programs work and message-passing performs.

2.3 Current visualization tools for parallel programming on MPI

The purpose of this section is both to review the state of the art in visualization tools for message-passing programming and to evaluate whether any could be candidates for adaptation or outright adoption for Pilot. In order to make that judgment, we start by stating our objectives for providing a good solution to log visualization for Pilot. After an initial scan of the literature, including a helpful survey paper [Mar09], it was found that many current
visualization tools for parallel computing are no longer limited to basic functions like debugging with a small number of processes. Instead, their main focus has shifted to performance profiling, such as analyzing large data [WZYK14], mapping 3D graphs [SHN10, LLB+12], and manufacturing animation [SMM13], but profiling is not the scope of this thesis. Therefore, we proceed to present tools that are more suitable for our debugging emphasis.

2.3.1 Objectives for tool support

There are two fundamental requirements for a visualization tool to support Pilot programming users. First, it is able to analyze and draw an appropriate graphical representation from reading a standard parallel programming logfile. Second, the picture produced by the visualizer should show the time of each event and the relationship between writing and reading processes in message passing. While there may be a number of existing visualization tools for parallel programming, they may or may not align with the features of Pilot. Therefore the following characteristics or criteria are proposed to help with filtering when surveying literature and the state of the art.

1. Visualizer in Pilot’s own terms instead of MPI’s terms

Most existing visualization tools for parallel programming are designed for MPI, so they can only identify and present MPI’s functions in pictures, but not Pilot’s. Furthermore, since a single Pilot library function may result in multiple MPI calls “under the hood,” such graphical representations will show a large number of low-level operations confusing to Pilot programmers using the latter’s higher-level abstractions (such as channels and bundles). The worst cases are: 1) setting PI_CheckLevel at 2 (the writer sends an extra MPI message so that the reader can check that both are using the same format strings, to ensure
data type and length consistency); 2) invoking the Pilot deadlock detection and/or logging facilities (each Pilot I/O function sends an MPI message to the special process that runs the deadlock detector and/or writes the logfile); 3) using multiple data formats for a single read/write (e.g., “%d %100f”) which results in multiple MPI messages; and 4) using the formats “%^” and “%s” to send variable-length arrays and strings, respectively (results in an extra array length message).

1. **Debugging more than profiling**

Some visualization tools for parallel programming focus more on large data analysis and profiling; however, since Pilot is targeted at novice parallel programmers, the functional requirement of the tool should be more focused on debugging.

2. **Freeware to go along with free/open source Pilot**

Because of the cost, a commercial tool is not suitable for the educational environment. Also, it can cause obstacles in the future, since with the development and updating of the third-party libraries, it may require more capital to purchase the latest revision to keep this facility in proper functioning. While commercial tools may be available to researchers with accounts on large-scale HPC consortia, such as Compute Canada, educators, especially of undergraduates, may need to teach using inexpensive multicore platforms and free software such as OpenMPI and gcc. Pilot is equally compatible with both the HPC consortium and the quad-core laptop, but that would change if it were tied to a commercial visualization tool.

3. **Visualizer with less dependency on additional libraries**

Not all libraries are readily available on all cluster platforms, or may require special attention from system administrators to install properly. Because any lack or wrong version of
libraries may cause failure when installing, or even a small updating in libraries may also lead to invalidation of the visualizer, the fewer libraries are needed, the more likely that the visualizer can be successfully installed on any given platform.

4. **Visualizer should be widely used or still being updated now**

Both for current use and future development, a tool used by a large amount of people usually has a long life cycle, and one can get more guidance in using it and solving problems that arise.

5. **Able to be used “as is”**

This means the third-party library does not need to be modified to meet the features of Pilot specially, so there is no future maintenance commitment for Pilot library maintainers and developers, which would not be practical in an academic setting.

### 2.3.2 Overview of the tools tested in the experiments

After surveying, eleven tools concerned with visualizing MPI parallel programs were found which may have capabilities to be applied to Pilot. The description follows the order we discovered and evaluated these tools. They are DDT [All09], Pajé [dKSB00, SD], Vampir [Nag16, NAW+96], ViTE [CFJ+], Triva [SHN10, Sch10], Viva [LBGM14, Sch16b], Jedule [HHS10], Jumpshot-4 (referred to as Jumpshot hereafter) [CALG07], TAU [SM06, Dep16], Score-P [SDK+14], and MPE [CGL98]. More precisely, they can be classified into three categories which are visualization tools, debugging tools, and systems that provide a measurement infrastructure for profiling. Except for DDT, TAU, Score-P, and MPE, the rest of the tools are complete visualizers. DDT is a debugging tool with visualization facility, and TAU, Score-P, and MPE are measurement infrastructures generating
a logfile to support visualizers. For example, Score-P is for profiling execution of parallel programs. In addition, TAU has its own affiliated visualizer ParaProf [BMS03], but instead of ParaProf the name “TAU” is commonly used to represent the whole system.

Table 2-1 summarizes the features of each tool according to the characteristics/criteria listed in Section 2.3.1. A check mark means the tool meets the criterion, while an “X” means a failure. No mark means that either the characteristic/criterion is not applicable for that tool, or since it clearly became disqualified, the characteristic was not further researched and is therefore “unknown.”

<table>
<thead>
<tr>
<th>Categories and tool names</th>
<th>Show Pilot functions (not MPI)</th>
<th>Debugging more than profiling</th>
<th>Free to use</th>
<th>Fewer library dependences</th>
<th>Widely used or being updated</th>
<th>Able to be used “as is”</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Debugging tool</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Visualizer</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pajé</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Vampir</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ViTE</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Triva</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Viva</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Jedule</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Jumpshot-4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Measurement infrastructure</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAU (ParaProf)</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Score-P</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MPE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Visualization tools may need input from logfiles produced during application program execution, so we must also consider how to generate these logfiles. While an
application program can, in principle, be altered to produce any logfile format by itself—
similar to the way that Pilot presently produces its own simple log format via fprintf state-
ments—there are logging libraries that greatly ease production of the various standard for-
mats. Some are highly complex, because they are designed for compact representations of
voluminous data, specification of drawable objects, and global time synchronization when
merging logs from separate cluster nodes. There are basically two methods of employing
these libraries, some of which come with the visualization tools. The first one, an “easy”
way, is to simply execute the parallel program with the instrumented library. The other one,
dubbed a “custom” way, is to insert calls to the tracing library’s API into the application
program. After that, a standard logfile could be read in and displayed as a picture by a com-
patible visualization tool. The following Table 2-2 lists several main formats of logfiles
that have been developed over the years (in order from oldest to most recent), and along
their tools for generation and display. Outdated versions (CLOG and SLOG) are not
included. A tool is listed in the right column if it can display the format directly or after an
internal conversion step. (The columns do not take into account the availability and possi-
ble use of external logfile conversion utilities.)

Table 2-2. Standard logfile formats and their associated tools

<table>
<thead>
<tr>
<th>Name of logfile format (oldest first)</th>
<th>Method of generation</th>
<th>Compatible display tools</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Easy way</td>
<td>Custom way</td>
</tr>
<tr>
<td>TRACE file [SD]</td>
<td>EZTrace, Akypuera</td>
<td>GTG</td>
</tr>
<tr>
<td>OTF file [Nag16]</td>
<td>VampirTrace</td>
<td>VampirTrace, GTG</td>
</tr>
<tr>
<td>TRC file [Dep16]</td>
<td>TAU</td>
<td>TAU</td>
</tr>
<tr>
<td>CLOG-2 file [CGL98]</td>
<td>MPE</td>
<td>MPE</td>
</tr>
<tr>
<td>SLOG-2 file [CGL98]</td>
<td>MPE</td>
<td>MPE</td>
</tr>
<tr>
<td>OTF2 file [EWG+12]</td>
<td>Score-P</td>
<td>Score-P</td>
</tr>
</tbody>
</table>
After having given a brief introduction to the criteria of these visualization tools, we will introduce the details of each of them in the following sections. The most important ones for each will be presented in an independent section and the others will be listed together in the end. MPE and Jumpshot are covered in Section 2.4. All the following screenshots except for Vampir were obtained by running the tools on a cluster of our local HPC consortium, Shared Hierarchical Academic Research Computing Network (SHARCNET).

### 2.3.2.1 DDT
Distributed Debugging tool (DDT) [All09], which is marketed by Allinea, is an intuitive, scalable, graphical parallel debugger. DDT can be used as a single-process or a multi-process MPI program debugger, and it provides all the standard debugging features for each running thread such as stack trace, breakpoints, watches, and view variables by threads. Moreover, DDT supports C, C++, and Fortran along with a large number of platforms, compilers and all known MPI libraries. In addition, DDT has a memory debugging feature to detect some errors before they cause a crash, and it can also display MPI internal message queues. DDT is an expensive commercial tool, and it is available on SHARCNET where it helps with debugging of Pilot programs. Since DDT is only licensed to run on four of the clusters—not including the one designated for teaching the school's undergraduate parallel programming course—having to awkwardly rebuild a Pilot application just for execution on another cluster throws up an impediment for students wishing to use this tool, and experience shows that they refrained from using it, even when it would have been helpful.
Figure 2-4 shows the User Interface of DDT. The upper blue bar shows the process or thread number. By clicking a certain number, the programmer could set breakpoints and check the values of variables in the corresponding process or thread for debugging.

We experimented with DDT at first, since it is available on SHARCNET and more familiar to us than other tools. There was no obstacle to running DDT with Pilot; however, the User Interface of DDT is still more like a traditional sequential debugging tool, and its visualization facility focuses on statistics but does not produce a picture displaying the message passing between processes. In other words, DDT does not give an intuitive view of message passing to Pilot programmers, so it may not be a suitable visualizer for Pilot. Therefore, we gave up using it.

2.3.2.2 Vampir

Vampir [Nag16, NAW+96], developed by two science centres Julich and Dresden, provides an easy-to-use analysis framework which enables developers to quickly display program behavior. Vampir can filter processes, functions, messages and collective operations.

Figure 2-4. User Interface of DDT
to be visible or not according to users’ choices and definitions, and has hierarchical grouping of threads, processes and nodes. In addition, Vampir has powerful adjustment and processing functions in graphical displays such as zooming, scrolling, and comparing (for example, the accumulated execution time of each kind of function, and performance between the programs having the same functionality but different structure and calls being used). It also allows users to check the source code that generated a given event.

Vampir reads Open Trace Format (OTF) trace files produced by VampirTrace [ZIH] which is an open source parallel program tracing library. Figure 2-5 depicts the User Interface of Vampir. Process and global time are presented as Y and X axes, respectively. The green and red blocks represented Application and MPI functions, respectively, and the black lines show the message passing between each process.

![Figure 2-5. User Interface of Vampir [Bur07]](image)

Vampir was the second tool that came into our vision. Its user interface meets our requirement, it is frequently updated, and in wide use for MPI application profiling. Nevertheless, we found that Vampir is focusing more on profiling, as shown by its choice of
colours: green is application execution, red is MPI functions, so green is “good” and red is “bad” (communication overhead). A more serious deficiency is that Vampir is a commercial tool, so it is not available to go further in our research.

2.3.2.3 Pajé

Pajé [dKSB00, SD], developed in a collaborative open source environment, is an interactive and scalable trace-based visualization tool. Pajé has two views for graphical presentation, a Space/Time view and a Statistical view. Moreover, it has a number of filters to control the visualization such as grouping the processes (e.g., the processes which have the same work function), selecting customized processes or objects to be visible in the picture, and repositioning the display order of objects. For example, related processes can be shown close together, or terminated objects, like short-living processes, can be recombed to reuse the screen space. The other significant feature is that Pajé and most of its related log-file producing libraries are free, open-source, and available online.

Pajé reads a .trace file produced by Akypuera (a tracing library that provides a low intrusion and low memory footprint for MPI applications) [Sch16a], EZTrace (a tool that generates execution traces of parallel applications) [TFR+16], otf2paje (a library that creates the Pajé’s generic file format by converting OTF2 archives) [Sch12], and other formats listed on their website [SD]. Figure 2-6 shows the User Interface of Pajé. It is similar to the one of Vampir. Process and global time makes up Y and X axes, respectively, but it is noteworthy that the meaning of colour in these blocks are different from Vampir’s colour coding. Green represents a function to send, while red represents a function to receive in each process. The white arrows represent message passing between them.
In experiments, Pajé succeeded in displaying events and message passing after reading a .trace file. Nonetheless, the .trace file based on MPI cannot visualize Pilot’s own API functions. As a result, we stopped the experiment on Pajé.

2.3.2.4 ViTE
ViTE [CFJ+], developed by INRIA, is a powerful, portable and open-source profiling tool designed to visualize traces produced by parallel applications. ViTE is able to open several trace formats (OTF and TRACE, with TAU not fully tested), browse, export, filter, retrieve information, and draw graphical statistics from these trace files.

Both the two trace generation methods could be used on ViTE. Either ViTE can read a TRACE file produced by EZTrace, or using Generic Trace Generator (GTG) [Tra12] library by inserting functions into the parallel program to create an OTF file, and then read by ViTE. Figure 2-7 shows the User Interface of ViTE. Like the pictures output by Vampir and Pajé, process and time make up Y and X axes, respectively, and the blocks and arrows represent the functions and message passing.

ViTE was experimented at the same period of time as Pajé, since they both could read .trace files. However, the disadvantages of ViTE are similar to Pajé’s because it shows a cluttered view with undesired “under hood” MPI calls without providing any facilities to log customized functions (such as Pilot API functions). Moreover, ViTE is no longer under
development, and some of its planned facilities, like the filtering capabilities (which could potentially reduce the clutter), are not implemented.

2.3.2.5 TAU
Tuning and Analysis Utilities Performance System (TAU) [SM06, Dep16], developed by University of Oregon and Los Alamos National Laboratory, is not a visualization tool per se, but is instead a portable profiling and tracing toolkit for performance analysis of parallel programs. TAU provides an API to support multiple languages such as Fortran, C, C++, UPC (Unified Parallel C), Java, and Python. TAU uses a selective instrumentation file to organize and control profiling of different parts of the program. It is capable of gathering performance information through instrumentation of functions, methods, and basic blocks. It can also generate event traces that can be displayed with the Vampir, Paraver, or Jump-shot trace visualization tool.

First, by running a parallel program with the TAU library, several .trc files having the same quantity as the processes can be generated. Then by using tau_treemerge.pl which is a Perl script, these .trc files can be merged into one .trc file named “tau.trc”. After
that, using the Java program “tau2slog2 tau.trc tau.edf -o tau.slog2” converts the .trc file to a .slog2 file. Finally, Jumpshot, a Java-based visualization tool can read this .slog2 file and export a picture to analyze the performance of the program. Figure 2-8 exhibits the User Interface of Jumpshot displaying a SLOG-2 file converted from a .trc file. Processes and global time are shown on the Y and X axes, respectively. The rectangular blocks and arrows present MPI functions and message passing, respectively. The bubbles are to record the significant events during program execution.

![User Interface of Jumpshot reading a SLOG-2 file converted from a .trc file](image)

Figure 2-8. User Interface of Jumpshot reading a SLOG-2 file converted from a .trc file

TAU aligns with the most features of Pilot, and it could provide a self-defined selective instrumentation for a customized library like Pilot, but we could not get it to work consistently. It was found that if the instrumented Pilot program was run with a selective instrumentation file (even if it was empty) the Perl script would crash in the phase of integrating the per-process .trc files, or if using TAU’s default instrumentation file, then in the tau2slog2 Java conversion phase it would crash and only present the main process of the Pilot program in Jumpshot. Since this happened even for very simple programs, and we
observed references to TAU’s segmentation faults on the Web, we jettisoned TAU as too unreliable.

2.3.2.6 Other unsuitable tools
For the rest of the tools, they also have the same or even more serious deficiencies and flaws than the previous five, so we gave them up immediately and did not conduct many further experiments with them. For Triva [SHN10, Sch10] and Viva [LBGM14, Sch16b], though they are upgraded versions of ViTE, they still do not provide any facility to achieve showing Pilot’s terms instead of MPI’s, and their newest version is still in test. Score-P [SDK+14] is a scalable performance measurement infrastructure for parallel codes that generates OTF2 logfiles. It aims to consolidate and replace previous trace libraries such as VampirTrace for improving measurement. As a tool focused on performance, it is not the tracing infrastructure we are looking for. Its OTF2-compatible display tool Scalasca [GWW+10, FTG16] presents complex pictures that would not be helpful for novice users. Jedule [HHS10] is a tool for helping researchers to develop scheduling algorithms by visualizing the graphical representation, but it has ceased development for four years and there has been no conference paper about it since.

2.4 Jumpshot with MPE
Although the results of experiments on most of these tools were not positive, the user experience for Jumpshot was impressive. After further considerable searching and reading, we concluded it could be great to be utilized and applied to Pilot, rather than building another visualizer ourselves. Since the logfile for Jumpshot is SLOG-2, two proposals were considered to generate logfiles: (1) adapt Pilot's trace logger to directly output SLOG-2 format, or
another (simpler) format that is convertible to SLOG-2; (2) create a “pilot2slog2” utility that can read Pilot’s native logfile format and produce SLOG-2 format. Both of these approaches were rejected because Pilot’s native logger does not record sufficient accurate information. On the one hand, it logs only the start of an API event, but for blocking communication the duration of the event is important. On the other hand, its timestamps record when the event report arrived at the logger process, not when the event actually took place on its own process (which could be on another computer). A superior approach was discovered, that of making Pilot use the MPE library, which has a simple (though scantily documented) API and generates standard CLOG-2 logfiles suitable for conversion to SLOG-2 format and visualization with Jumpshot. This approach overcomes the aforementioned deficiencies of the Pilot native logger. MPE and the accompanying visualization tool Jumpshot are described in detail in the subsections following.

2.4.1 MPE

The Multi-Processing Environment (MPE) library was developed by University of Chicago and Argonne National Laboratory [CGL98]. Based on the characteristics of MPI, MPE supplies various extension functions for debugging, graphics, and some common utility routines. Its main purpose is to generate logfiles of MPI calls automatically, and it also allows customized logging via its API. In addition, its tracing library can produce a textual log of MPI calls on stdout similar to Pilot’s native logging feature, and its animation API allows a program to display some real-time graphics via X windows.

MPE can generate SLOG-2 format logfiles or CLOG-2 format, which is convertible to SLOG-2 for input to Jumpshot. CLOG-2 is the default, but SLOG-2 can be selected via
the MPE_LOG_FORMAT environment variable. Even though it requires an extra step, the literature calls the conversion approach “preferred” for two reasons:

1. The program may turn out to be “non well-behaved” (which is never explained) and it may produce a defective SLOG-2 file that cannot be properly displayed. This is a risk.
2. The conversion step can be useful for (a) diagnosing problems with the log contents, say, due to improper use of the logging API, and (b) adjusting conversion parameters that affect the subsequent display such as the “frame size” (the amount of data initially displayed by the visualization tool).

The second point is well taken: Regrettably, the user documentation about MPE’s API is inadequate and comes with limited examples, so considerable experimentation was necessary. Incorrect assumptions would show themselves as errors reported by the CLOG-2 conversion utility, enabling us to finally discover how the API is intended to be used.

When MPE is built from its source distribution and installed, it creates several libraries with similar names, yet the documentation does not clearly describe their differences. Investigation yielded these explanations:

• libmpe.a is the “normal” library to link with MPI programs (including Pilot applications) and it implements the logging API (MPE_ functions).

• libmpe.a includes the above, and also the ability to intercept calls to all MPI functions and automatically log those. Since we specifically do not wish to log MPI calls, we do not use this one.

• libampe.a and libtmpe.a are the animation and tracing libraries, respectively, which we do not use.
• libmpe_nompi.a: The “nompi” library is for use with serial programs which do not call MPI. A serial program can thus utilize MPE’s logging capabilities. Erroneously using a nompi library with an MPI program was found to output only rank 0’s events into the logfile, and was apt to produce a conversion error from CLOG-2 to SLOG-2.

• libmpe_null.a, lib_nompi_null.a: Linking a program that has embedded MPE calls in it with a “null” library will deactivate the calls and produce no logfile when executed. That is, the calls still take place, but they effectively become no-ops. This option can be easily exercised at the link step, and is superior to physically removing (e.g., via #ifdef) the calls from the program and recompiling. Thus, a production code can be written to output a logfile, but that capability can be conveniently suppressed until it is needed, say, for debugging purposes.

Table 2-3 lists the basic MPE calls used for custom logging in an MPI program and producing a .clog2 file. The documentation was scraped out of comments in the source file mpe_log.c. A simple example code for using MPE in an MPI program can found in an MPE installation under share/examples/logging/pilog.c. The Pilot library is itself an MPI program, so, in principle, these functions can be invoked from within the Pilot library on behalf of the user’s Pilot application. MPE assumes that MPI is running and uses MPI for some internal operations. Thus, MPE functions should be called after MPI_Init() and before MPI_Finalize().

For the basic MPE logging concepts, the user has a choice of recording events that have duration, called “state” (having start time and end time), and those that do not, called (by us) “solo events” (having only event time). Recording a particular event in the log can be thought of as instantiating one or another “event ID”, which is simply an MPE-generated
<table>
<thead>
<tr>
<th>Table 2-3. Basic MPE function calls for producing a CLOG-2 logfile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization functions</strong></td>
</tr>
<tr>
<td><strong>MPE_Init_log()</strong></td>
</tr>
<tr>
<td>Invoke and initialize the MPE logging facility.</td>
</tr>
<tr>
<td>**MPE_Log_get_state_eventIDs(int <em>statedef_startID, int <em>statedef_finalID)</em></em></td>
</tr>
<tr>
<td>Get a pair of unique event numbers to be defined as a drawable state for functions through</td>
</tr>
<tr>
<td><strong>MPE_Describe_state().</strong></td>
</tr>
<tr>
<td>1. statedef_startID: Starting eventID (time) for the definition of state</td>
</tr>
<tr>
<td>2. statedef_finalID: Ending eventID (time) for the definition of state</td>
</tr>
<tr>
<td>**MPE_Describe_state(int state_startID, int state_finalID, const char <em>name, const char <em>color)</em></em></td>
</tr>
<tr>
<td>Describe the attributes of a state without byte informational data in MPI_COMM_WORLD. Then adding a state definition to the logfile.</td>
</tr>
<tr>
<td>1. state_startID: Event number (time) for the beginning of the state</td>
</tr>
<tr>
<td>2. state_finalID: Event number (time) for the ending of the state</td>
</tr>
<tr>
<td>3. name: Name of the state. The maximum length is 32 bytes</td>
</tr>
<tr>
<td>4. color: Colour of the state. The maximum length is 24 bytes</td>
</tr>
<tr>
<td>(Note: The maximum length is according to the storage space CLOG-2 file format).</td>
</tr>
<tr>
<td>**MPE_Describe_info_state(int state_startID, int state_finalID, const char *name, const char <em>color, const char <em>format)</em></em></td>
</tr>
<tr>
<td>Describe attributes of a state with byte informational data in MPI_COMM_WORLD. The first 4 parameters are the same as <strong>MPE_Describe_state().</strong> Then adding a state definition to the logfile.</td>
</tr>
<tr>
<td>5. format: Printf style format control string (with % conversion specification) for the state. The maximum length is 40 bytes. If format is NULL, it is equivalent to calling <strong>MPE_Describe_state().</strong></td>
</tr>
<tr>
<td>*<em>MPE_Log_get_solo_eventIDs(int <em>eventdef_eventID)</em></em></td>
</tr>
<tr>
<td>Get a single unique event number to be used to a drawable event to mark important point in time through</td>
</tr>
<tr>
<td><strong>MPE_Describe_event().</strong></td>
</tr>
<tr>
<td>1. eventdef_eventID: eventID (time) for the definition of event</td>
</tr>
<tr>
<td>**MPE_Describe_event(int eventID, const char <em>name, const char <em>color)</em></em></td>
</tr>
<tr>
<td>Describe the attributes of an event without byte informational data in MPI_COMM_WORLD. Then adding an event definition to the logfile.</td>
</tr>
<tr>
<td>1. eventID: Event number (time) for the event occurring.</td>
</tr>
<tr>
<td>2. name: Name of the event. The maximum length is 32.</td>
</tr>
<tr>
<td>3. color: Colour of the event. The maximum length is 24.</td>
</tr>
<tr>
<td>**MPE_Describe_info_event(int eventID, const char *name, const char <em>color, const char <em>format)</em></em></td>
</tr>
<tr>
<td>Describe attributes of an event with byte informational data in MPI_COMM_WORLD. Then adding an event definition to the logfile. The first 3 parameters are the same as <strong>MPE_Describe_event().</strong></td>
</tr>
<tr>
<td>4. format: Printf style format control string (with % conversion specification) for the event. The maximum length is 40 bytes. If format is NULL, it is equivalent to calling <strong>MPE_Describe_event().</strong></td>
</tr>
<tr>
<td><strong>Event logging functions</strong></td>
</tr>
<tr>
<td>**MPE_Log_pack(MPE_LOG_BYTES bytebuf, int <em>position, char tokentype, int count, const void <em>data)</em></em></td>
</tr>
<tr>
<td>Pack the informational data into the byte buffer to be stored in a informational event.</td>
</tr>
<tr>
<td>Output Parameters:</td>
</tr>
<tr>
<td>1. bytebuf: Output buffer declared as char[32] by MPE_LOG_BYTES</td>
</tr>
<tr>
<td>2. position: When called, position gives the byte offset.</td>
</tr>
<tr>
<td>3. tokentype: A character token type indicator, currently supported are 's', 'h', 'd', 'l', 'x', 'X', 'e' and 'E'</td>
</tr>
<tr>
<td>4. count: The number of continuous storage units as indicated by tokentype</td>
</tr>
<tr>
<td>5. data: Pointer to the beginning of the storage units being copied</td>
</tr>
</tbody>
</table>
integer. States require a pair of event IDs, one for the start and one for the end. State and solo event IDs are defined and given properties of name and displayable colour. Event instances inherit those properties and also add time(s) and optional text. Therefore, one must anticipate all the kinds of events that one will want to record, and define each one by generating an event ID.

In detail, the user program must include “mpe.h” first. MPE logging function is initiated by calling MPE_Init_log() and terminated by calling MPE_Finish_log().
MPE_Log_get_state_eventIDs() and MPE_Log_get_solo_state_eventIDs() are called to assign event IDs for user-defined states and solo events, respectively. Name and colour are defined for each state and solo event by MPE_Describe_(info_)state() and MPE_Describe_(info_)event(), respectively. These colours are specified in MPE using X11 colour names [Com]. Based on error messages coming from the CLOG-2-to-SLOG-2 conversion step, it seems that if a colour name is a single word, all the letters should be in lower case like “red”. If it is multiple words, the first letters of these words should be capitalized in camel case like “ForestGreen”. After initialization, calling MPE_Log_event() in pairs can log a state, for example, MPE_Log_event( event1a, 0, NULL ) and MPE_Log_event( event1b, 0, NULL ). The first call logs the start point, and the matching one logs the end point. This will result in displaying a coloured rectangle from the start time to the end time. Also, MPE_Log_event() can be called individually with event ID for recording solo events. This will result in a displaying a small coloured circle or bubble.

Both rectangles and bubbles can be right-clicked in Jumpshot to display times and additional text (if any). In order to show the relationship of message passing, MPE_Log_send() and MPE_Log_receive() should be called in pairs, and have the same tag number and length of data as parameters to correspond with each other. This will result in a white arrow being drawn from the timeline corresponding to the sending process to that of the receiving process.

At the program’s end, MPE_Log_sync_clocks() is called to synchronize or recalibrate all MPI clocks to minimize the effect of time drift, and MPE_Finish_log() is called to write the single merged CLOG-2 logfile.
2.4.2 Jumpshot

Jumpshot [CALG07] was developed by Argonne National Laboratory. Jumpshot is a visualization program for the SLOG-2 logfile format. It permits seamless scrolling at any zoom level of an entire logfile. In addition, Jumpshot provides a number of functionalities, such as dragged-zoom, grasp and scroll, instant zoom in/out, easy vertical expansion of timelines, and cut and paste of timelines. Jumpshot has a search-and-scan facility that helps users locate graphical objects which are hard to find. Also, Jumpshot can draw a picture from user-selected duration which allows for ease of data analysis on the statistics of a logfile. For example, it enables easy detection of load imbalance across processes among timelines. The legend table categorizes different objects, such as arrows, states, and events, and provides manipulation on visibility and searchability. Also, all known SLOG-2 convertible logfile formats, such as CLOG, CLOG-2, RLOG (the internal MPICH2 logging format) [BBB+11], and IBM’s Unified Tracing Environment (UTE) [IBM06], could be transformed by an integrated logfile converter in the visualizer. In this way, Jumpshot satisfies many users’ expectation of look and feel for a standard visualization tool.

Figure 2-9 shows the Jumpshot logfile “convertor”[sic] window. When given a CLOG-2 file, this window can allow conversion of a supported trace file format to SLOG-2 format.

Then the SLOG-2 file could be read by the visualizer and exported as a picture. An example has been shown in Figure 2-10. The legend table is shown on the left and the main picture is on the right of it. The dense blue state rectangles represent computing, and in this example took most of the time of each process. Figure 2-11 enlarges the white translucent
The red part is for broadcast, and the arrows from the process 0 to other processes represent procedure of the message passing of MPI_Broadcast.
2.4.3 Conclusion

While the other tools are not very suitable for adaptation to Pilot’s use, MPE and Jumpshot were found to meet all six characteristics/criteria proposed in Section 2.3.1. Both of them are free and widely used. Also, they do not focus a lot on profiling and do not have much dependency on other libraries. What is more, MPE and Jumpshot can be used “as is”, and they can record and display functions in Pilot’s own terms. The details will be explained in Section 3.1.

In summary, after weighing the pros and cons, it is highly feasible to adapt MPE and Jumpshot with the Pilot library to achieve a log visualization facility. The next chapter will show the strategy for adapting MPE to Pilot, walking through from the current criteria to introducing visual design for how to use and display Pilot traces in Jumpshot.
Chapter 3

Adapting MPE for Pilot

After having selected MPE and Jumpshot as a logfile producer and visualization tool with high compatibility to be adapted to Pilot, it is also essential to decide how to adapt them to the Pilot library and how to utilize their functions to present the features of a Pilot program. The result of the decision can largely impact on users’ experience, especially for the novice parallel programmers.

First, Section 3.1 will review our current criteria for a good solution and show in detail how MPE and Jumpshot meet them. Then, Section 3.2 will describe MPE’s logging capabilities when coupled with Jumpshot. Finally, Section 3.3 elaborates a design plan for applying those capabilities to Pilot. The implementation of that design is the subject of Chapter .

3.1 Criteria

Before offering a proposal of a detailed visual design for displaying Pilot traces in Jumpshot, the first thing is to look back the six criteria listed in Section 2.3.1 and to confirm how MPE and Jumpshot can meet them in detail. Additionally, five criteria discovered during the first attempt of using MPE and Jumpshot have been added.

According to Software Quality Characteristics [Fle], we can logically group these criteria under the following six characteristics which are functionality, reliability, usability,
efficiency, maintainability, and portability. Each criterion is listed below with an explanation of how MPE and Jumpshot satisfy it.

1. Functionality

- *Must display data in Pilot’s terms, not MPI’s:* MPE provides a customized logging facility which can be used to insert MPE logging calls into any program. If we insert into the Pilot library MPE calls to define and log events, we can log the user application’s calls to Pilot's high-level API and ignore Pilot's underlying calls to MPI. When this is displayed as a picture for a high level vision, users will not be confused by low-level details of which they are not aware. Thus MPE and Jumpshot can be used to record and display events in terms of Pilot's processes, channels, and API functions rather than MPI’s, just as the criterion states.

- *Should be geared toward debugging more than profiling:* MPE is simply a trace logging facility, and is not optimized to collect performance data, say, at the level of individual functions (such as number of calls and amount of time spent in the function). One could conceivably post-process the log to extract such statistics, but timings would not be accurate and would include time waiting for I/O, not purely execution. Since we want to do tracing for debugging and not collect performance profiles, MPE’s abilities are compatible with our needs.

2. Reliability

- *Must not generate errors or crash:* During the learning curve for the MPE library, we encountered errors using its API and initially produced faulty CLOG-2 files that could not be successfully converted to SLOG-2 and visualized in Jumpshot. After learning
from these mistakes, Pilot now generates logfiles that are error free and displayable. We were never able to get to this stage with some other packages, due to installation problems, incompatibility between library versions, and poor user documentation.

3. Usability

- **Must not require the user to modify application code (to generate logfiles):** For the Pilot application programmer, any third-party tools should be transparent, which means that they should not have to insert extra library calls in their application code. Since MPE could be inserted in the Pilot library, which is discussed in detail later, besides the installation of MPE and Jumpshot, the only new thing is to modify the user's makefile to link with the MPE library. We will provide them several essential documents, included in Appendix A: Building and Using MPE with Pilot, to simplify this stage: (1) A detailed integrated MPE and Jumpshot user instruction of installation and function explanation; (2) A skeleton makefile to set up the path directing to MPE and Pilot libraries.

- **Must not require the user to carry out many manual steps (to view the log):** For the Pilot application programmer, the step of producing a logfile by MPE is also simpler than many other libraries like TAU. Since undue complicated setup operations can daunt users even for an attractive application, we do not want to burden them with manual steps. By specifying “-pisvc=j” on the command line, the logging facility will be invoked and executed with the application to produce a CLOG-2 file without any other conversion or integration step. Then, this CLOG-2 file can be converted into a SLOG-2 file by one simple operations in a converter window of Jumpshot, which was mentioned in Section 2.4.2.
4. Efficiency

- Testing showed that the run time overhead of adding MPE logging to a Pilot program can be very minimal. See Section 5.2 for details. Therefore, this solution is both light-weight and efficient.

5. Maintainability

- *Must use MPE and Jumpshot “as is”: MPE and Jumpshot can work with Pilot without changing their source code. If this were not the case, then when a new revision of MPE or Jumpshot were issued, the subsequent developers and maintainers of Pilot would have to become committed to maintaining these modifications, and this is not realistic for academic freeware.*

- *Should be in wide use with a sizeable user base: MPE and Jumpshot are developed by Argonne National Laboratory. They have a large user base and are not orphan software. The latest version of MPE (including Jumpshot-4) available for public downloading is 2.4.9b, dated Dec. 2015. Also, these tools are currently in use and mentioned in recent publications. For example, *Using MPI Portable Parallel Programming with the Message Passing Interface* (2014 edition) has an appendix devoted to MPE [GLS14]. The recent book, *Multicore and GPU Programming* [Bar14], despite what the title would suggest, also covers distributed memory programming with MPI. It recommends and gives instructions on the use of MPE—both the “easy way” and the “custom way”—and Jumpshot (via the CLOG-2 to SLOG-2 conversion route) for debugging cluster programs. This currency ensures that Pilot can continue using MPE and Jumpshot for the reasonably foreseeable future.*
6. Portability

- **Should not depend on many other libraries:** The libraries that an MPE installation requires are CLOG, pthreads, and Xlib. The first one comes with MPE, and pthreads and Xlib are standard parts of Linux distributions. So it does not have any new library dependencies. As for Jumpshot, it only needs a JRE which is easy to obtain on any platform. Furthermore, by decoupling logfile generation (via MPE) and log viewing (via Jumpshot), these can take place on different platforms according to the user's convenience (provided they are willing to transfer the logfile).

- **Must not cause Pilot installation or execution to become dependent on a third-party library:** One of Pilot's strengths is that it can be installed on any platform that simply has an MPI library (such as OpenMPI); it does not need any other third-party libraries. Therefore, tying it to another library such as MPE was done with reluctance. However, this dependency can be relaxed by making its installation with MPE optional. The installer can choose whether or not to utilize MPE. In Pilot's code, this is reflected by making all MPE calls and data structures conditional using #ifdefs (see Section 4.1.2). If Pilot was installed without MPE, the only thing the user will notice is that “-pisvc=j” prints a warning that logging for Jumpshot is not available.

- **Must be compatible with Pilot’s free/open source license:** Since both MPE and Jumpshot are free/open source software, and they are available online, these two noncommercial softwares are compatible with Pilot to build an educational or research environment without spending any money.
After a long period of researching and analyzing, it can be said that MPE and Jumpshot meet all these eleven criteria according with recognized software quality principles of software engineering. Next, we explore further to find out the capabilities of MPE and Jumpshot that can potentially be adapted to Pilot’s logging needs.

3.2 MPE’s capabilities with Jumpshot

To propose a visual design for displaying Pilot traces in Jumpshot, only satisfying the criteria is far from enough. MPE’s capabilities with Jumpshot should also be taken into consideration. In this subsection, the abilities of MPE and Jumpshot are going to be presented first in Section 3.2.1, and the limitations of using them will be discussed in Section 3.2.2.

3.2.1 What MPE can record and Jumpshot display

MPE's logging abilities have been largely described in Section 2.4.1. It can record states—loggable events with duration—solo events having just a time, and arrows representing message-passing information. It should be noted that nested states are possible: If state A runs from time 3 to 20, and state B from 5 to 8, then state B is fully nested within A. When displayed in Jumpshot, A will be shown as an outer rectangle and B as another rectangle within A.

States and solo events have attributes of a name and a displayable colour, and specific instances of them can have optional additional text along with the actual instance times. In addition to recording instances of states and events that occur in individual processes, it can also record a message being passed from one process to another.

The above are written into per-process logs which are merged at the end of the program when MPE_Finish_log() is called and become a single .clog2 file written on the file
system of the rank 0 process. After conversion to SLOG-2 format, the logfile can be displayed by Jumpshot.

Jumpshot’s displays, as shown in Figure 2-10 and Figure 2-11, are drawn on coordinates axes presenting processes and global time in seconds as Y and X axes, respectively. The process number starts from zero which represents the main process. States, events, and message passing are displayed as rectangles, bubbles, and arrows, respectively. Nested states are shown as one or more small rectangles positioned inside a large one. The colours of state rectangles and event bubbles can be specified via MPE, whereas the colours of message arrows are fixed by Jumpshot. Details of states, events, and message passing by MPE can be shown in a detail popup window after right-clicking the graphical object, shown from Figure 3-1 to 3-3, respectively. These figures were obtained by running an MPI program (not a Pilot program) with MPE.

![Figure 3-1. A detail popup window of state](image)

A detail popup window of state is shown in Figure 3-1. At the top of the window, it shows the name of this state which is “Reduce” with an icon of a coloured rectangle that comes from the definition step of state. In the main body of the window, the first line shows the duration of this state calculated automatically from the difference of the start and end times when MPE_Log_event() was called (shown on the second and third lines as [0] and

42
“World_rank” is the MPI rank where this state exists. The last line presents the extra information added by the programmer by calling MPE_Log_pack().

Figure 3-2. A detail popup window of event

In Figure 3-2 a detail popup window of an event is shown. At the top is an icon of the coloured event bubble, and the name “MPE_Comm_finalize” that comes from the event definition step. Its main body shows the time when this event was logged by MPE_Log_event() and the MPI rank of its process.

Figure 3-3. A detail popup window of message passing

Figure 3-3 illustrates a detail popup window of message passing. The title “message” is a fixed name given by Jumpshot with an arrow icon. The body is similar to the state popup, but here the ranks differ because the message was sent from one rank and received by the other, as reported by calling MPE_Log_send() and MPE_Log_receive(), respectively. The last line presents the tag of this message which is 1, and the size of it which is 8.
Jumpshot has a “legend” button that brings up a listing of all the events and states in use by the logfile being displayed. Figure 2-10 on page 34, for example, has the legend on the left side. It is shown fully expanded in Figure 3-4.

![Legend window]

For each state or event, it gives the coloured icon, the name, and some simple statistics (which can be sorted): a “count” of the number of instances (which could represent the number of times a function was called) and two durations marked “incl” and “excl.” Inclusive means the sum of the duration of its state instances (equal to adding the widths of all its state rectangles, e.g., all the states named “Reduce”). Exclusive is the inclusive time minus any nested states, i.e., subtracting interior rectangles, which amounts to the time
spent computing purely in the state and not in its substates. These statistics are potentially useful for performance purposes in the absence of special-purpose profiling tools.

Another feature is that if users right click and drag on the main window, a popup window titled “Duration Info Box” will appear showing the time duration, the start time, and the end time of the selected interval. The popup also has a “statistics” button. If this button is clicked, another main window titled “Histogram for the duration [...]” will popup, zoomed to the same selected interval. This window uses the relative length of each coloured bar to depict the proportion of time that each process spent in that function. For example, a long red bar represents summing up the times spent in instances of PI_Read. It will be disappointing if its length is greater than that of the gray bar representing the total time spent in useful computing. Right-clicking on a bar pops up an information box giving the name of the Pilot function and the “ratio” (percentage) of the selected interval devoted to executing that function. Since this facility is more concerned with profiling, we will not discuss it further.

3.2.2 Limitations and solutions

The recording capabilities of MPE and the displaying capabilities of Jumpshot are impressive; however, there is no reason to deny that some limitations exist. We found two kinds of limitations, the practical kind (where we tried to do something but could not succeed), and the potential kind.

The first practical limitation is that message passing recorded in MPE facility is not able to note any extra text information, unlike for states and events, which means the detail popup window of an arrow shown by Jumpshot cannot provide information tied to a
message instance, such as the Pilot process names of sender and receiver and the message contents. Also, the message bubble cannot be given a different name or colour. In this way, without the extra text information the detail popup window and a specified name and colour, users are more likely to be confused with the sequence of each message passing, especially the ones between the same sender and receiver, since they look similar when displayed as non-distinctive arrows.

There are two alternatives for showing detailed message-passing information in Jumpshot. The first is using an additional event bubble, and the second is using a nested state rectangle. Compared with the state rectangle approach, which requires calling twice to denote a rectangle, the event bubble approach is more flexible and comprehensible for recording much more information. Event bubbles only require one call for each message-passing arrow, so the overhead of calling MPE functions can be reduced.

Another limitation is that the size of the extra text information is limited to forty bytes. If the length of content exceeds that, the rest will be truncated. There is no way to get around that limitation, and we must minimize the use of the space as far as possible in our design and implementation.

In terms of potential limitations, Jumpshot is able to display the SLOG-2 file format which was developed later and has more capabilities than the CLOG-2 format produced by MPE. (Even when MPE is directed to produce SLOG-2 output, it only uses the CLOG-2 features.) This means that using MPE to generate logfiles gives up, at least in theory, the ability to draw more sophisticated displays. The alternative would be to enable Pilot to write SLOG-2 format directly, but there is no ready-to-use C API for that purpose. Since we judged that we could obtain the results we need via MPE, we decided not to try writing
SLOG-2 format files with its more elaborate graphical objects. This limitation is just a potential one because we might be able to produce something better, or at least fancier, by working directly in SLOG-2, but we don't actually know that.

3.3 Visual design for displaying Pilot traces in Jumpshot

According to the criteria in adaptation and the capabilities of MPE and Jumpshot, a visual design for displaying Pilot traces in Jumpshot is proposed for each Pilot function regarding how they can be displayed in Jumpshot. In other words, this plan is for what a Pilot programmer is going to see in Jumpshot corresponding to each Pilot API call in their Pilot application.

First, the strategy for assigning colours is explained. It will be seen that colours are not used in an ad hoc, arbitrary fashion, but that a meaningful system was devised based on breaking Pilot functions into different categories. Then, the general plan for utilizing the available graphical objects (state rectangles, event bubbles, and message arrows) will be presented.

3.3.1 Plan for using colours

Since the colours of states and events can be defined by using MPE functions, a plan should be designed to specify the colour of each different state rectangle representing a Pilot function and each event bubble.

Prior to describing the plan, the whole set of Pilot functions are broken down into four categories according to their characteristics. The first category is the Pilot output functions including PI_Write, PI_Broadcast, and PI_Scatter. The second category is the Pilot input functions including PI_Read, PI_Gather, PI_Reduce, and PI_Select. The third
category is the Pilot administrative functions (meaning that they do not perform message I/O) including PI_Configure, PI_StartAll, PI_ChannelHasData, PI_TrySelect, PI_Log, PI_StartTime, PI_EndTime, PI_StopMain, and PI_Abort. There is a special problem with PI_Abort that could not be solved, see note 7 in Section 3.3.2. The last category is the Pilot functions which are not significant enough to warrant displaying in Jumpshot, either because they belong to the brief, one-time configuration phase (which is already displayed in total as PI_Configure), or because they are simple utilities with no communication implications. They are PI_CreateProcess, PI_CreateChannel, PI_CreateBundle, PI_CopyChannels, and PI_SetCommWorld in the configuration phase; PI_GetBundleChannel and PI_GetBundleSize in the execution phase; and PI_IsLogging, PI GetName, and PI_SetName which can be called in any phase.

The first principle is that all the functions in the same category should have similar colours. The second principle is that within a category we can distinguish simple channel-based communication (PI_Write and PI_Read) versus bundle-based collective communication (e.g., PI_Reduce, PI_Broadcast) by light versus dark shades of the same colours.

We decided to adopt a red theme for input and a green theme for output. This is the same plan used by Pajé (see Figure 2-6 in Section 2.3.2.3). There is an intention of meaningful symbolism here: The word “red” is similar to “read,” and since reading always blocks, it can also be associated with “red means stop.” In contrast, “green means go” can readily be connected with writing because in the context of message passing, sending a message has an interprocess synchronization effect—signalling to wake up a waiting reader—as well as a communication effect. The colours used could be distinguishable for all the people, even for the ones who have some colour blindness, since the author is himself
red-green colour blind, but still sees these pictures clearly. However, even if an individual user does not feel inspired by this symbolism, he or she will still recognize right away that input-type functions can easily be distinguished visually from output-type functions. Then, to apply the second principle, it is intended that collective Pilot I/O functions should use darker shades of PI_Read or PI_Write. For example, PI_Read and PI_Write are red and green. PI_Broadcast and PI_Gather are ForestGreen and IndianRed. The darker colour means that this function is called from the process at the start point or end point of a bundle.

After experimenting with colours, we found that some of them are distracting. For example, it is too overwhelming to use the colour blue in compute for each process—like the default for MPE logging of MPI programs (see Figure 2-10 on page 34)—since so much of the program execution time is spent within compute, and then this is represented in all these blue keeping staring at us.

The colours of state rectangles and event bubbles adopted for showing each Pilot function are listed in Table 3-1. The special one in the table is Compute which is not a real function call in the Pilot library. The time outside of the Pilot API calls is considered “computing,” then Compute refers to the full duration of each process’s execution phase shown by an outer gray rectangle, with Pilot API calls shown by coloured rectangles within it (this is an example of nested states). Moreover, yellow is used for the event bubbles connected with a state, while cyan is used for those not connected with a state, since they are eye-catching enough and have not been used for any other graphical objects.

The colour names listed in Table 3-1 are from Xlib [Com], since this is what MPE_Describe_state(), MPE_Describe_info_state(), MPE_Describe_event(), and MPE_Describe_info_event() expect. When using Jumpshot, it is also possible to adjust the
colours to individual taste via a popup colour chooser dialog by pressing one of the icon buttons of objects in the legend window, but this setting only persists for the current Jumpshot session.

### Table 3-1. Colour plan for state rectangles and event bubbles

<table>
<thead>
<tr>
<th>Category and Pilot functions</th>
<th>State rectangle colour</th>
<th>Event bubble colour</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output category (green theme)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL_Write</td>
<td>green</td>
<td>yellow</td>
</tr>
<tr>
<td>PL_Broadcast</td>
<td>ForestGreen</td>
<td>yellow</td>
</tr>
<tr>
<td>PL_Scatter</td>
<td>ForestGreen</td>
<td>yellow</td>
</tr>
<tr>
<td><strong>Input category (red theme)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL_Read</td>
<td>red</td>
<td>yellow</td>
</tr>
<tr>
<td>PL_Gather</td>
<td>IndianRed</td>
<td>yellow</td>
</tr>
<tr>
<td>PL_Reduce</td>
<td>IndianRed</td>
<td>yellow</td>
</tr>
<tr>
<td>PL_Select</td>
<td>IndianRed</td>
<td></td>
</tr>
<tr>
<td><strong>Administrative category</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL_Configure</td>
<td>bisque</td>
<td></td>
</tr>
<tr>
<td>PL_StartAll</td>
<td></td>
<td>cyan</td>
</tr>
<tr>
<td>Compute</td>
<td></td>
<td>gray</td>
</tr>
<tr>
<td>PL_ChannelHasData</td>
<td></td>
<td>cyan</td>
</tr>
<tr>
<td>PL_TrySelect</td>
<td></td>
<td>cyan</td>
</tr>
<tr>
<td>PL_Log</td>
<td></td>
<td>cyan</td>
</tr>
<tr>
<td>PL_StartTime</td>
<td></td>
<td>cyan</td>
</tr>
<tr>
<td>PL_EndTime</td>
<td></td>
<td>cyan</td>
</tr>
<tr>
<td>PL_StopMain</td>
<td></td>
<td>cyan</td>
</tr>
<tr>
<td>PL_Abort</td>
<td></td>
<td>See note 7 in Section 3.3.2</td>
</tr>
</tbody>
</table>

3.3.2 Plan for using graphical objects

MPE's API allows us to draw coloured rectangles to represent states, coloured bubbles to represent events, and (white) arrows to represent message passing, as well as to attach some additional text to objects (except for the arrows). Each of these graphical objects, when displayed in Jumpshot, can be right-clicked to bring up a popup window containing more
information, as was shown in Section 3.2.2. In the paragraphs below we explain the general strategy for using these objects.

A state rectangle will be used to display, on a given process’s timeline, each instance of its calling of a Pilot function. The time at the left edge of the rectangle will be just after the function is invoked, and the time at the right edge just before it returns. The associated popup will show the line number where it is called in the original .c file, the name of the calling process, and its index argument. Each Pilot process has a default simple name like “P3” meaning the third process created (equivalent to MPI rank 3); however, programmers can call PI_SetName to assign a more meaningful name, and they may wish to do so precisely for the purpose of logging and debugging. The reason for showing the first argument (an integer) is because it is common in master/worker patterns for all workers to execute the same function, distinguishing their instances only by the value of the first argument, which usually serves as an array index. Therefore, the argument denotes which of the workers issued the call. If there is a bundle argument involved in a Pilot collective function call, the name of the bundle (e.g., “B4”) will be shown. As with processes, the programmer can assign meaningful names to any channel or bundle.

Event bubbles will be used to annotate the state rectangles with the precise times of, and information about, important “milestones” during the execution of Pilot functions. For example, a PI_Read call can spend considerable time waiting for a message to arrive, so the rectangle could be rather long. The bubble in its rectangle will mark the moment when the message arrives, and its associated popup will display detailed information about the name of the channel involved in this PI_Read (e.g., “C3”). (On the output side, e.g., PI_Write, the data length and the value of the first element are also shown.) Furthermore, since a
single PI_Read may involve multiple messages (e.g., the format “%d %100f” sends two MPI messages, one for a single integer and another for an array of 100 floats), there will be a bubble inside each rectangle indicating when each individual message arrives.

A message arrow will be drawn between any output source and its input destination. In the case of collective operations like PI_Broadcast, a bundle with N channels will result in N arrows being drawn. The popups associated with message arrows always display the start and end times of the transmission, its duration, the MPI tag, and message size. No way was found to attach additional data.

PI_Select is a slight exception to the above patterns. On the one hard, it acts like PI_Read in that it blocks until a message is received on any of its bundle’s channels, therefore it should be represented as state. On the other hand, no message is actually received until a subsequent call to PI_Read, therefore it does not have an event bubble. Its information popup gives the index of the channel that is ready to read.

The above descriptions apply primarily to categories 1 and 2, the output/input functions. For category 3, administrative functions, bubbles are used to show related information and message arrows are not used at all. The treatment of these functions is shown below:

1. PI_Configure
This function is used to initialize the lower level MPI functions in Pilot and must be called before any processes, channels, and bundles are created. Since PI_Configure has a clear execution time period, i.e., the configuration phase stretching from PI_Configure to PI_StartAll, it can be represented by a state rectangle. However, since the logging facility of MPE cannot be invoked until after MPI functions have been initialized, the state
rectangle will not include the time for the initialization of MPI, but the bulk of the configuration phase, including creation of Pilot processes, channels, and bundles, is included in it. The additional text in the popup window of the state rectangle can show the process name, its index argument, and the source code line number where PI_Configure was called.

2. **PI_StartAll and PI_StopMain (Compute)**

PI_StartAll triggers each created process to execute its work function, while code following it continues with the identity of PI_MAIN. This marks the start of the execution phase, which ends when each work function returns and when PI_MAIN calls PI_StopMain. Thus, PI_StartAll and PI_StopMain bracket a clear execution time period. Therefore, they can be represented by a state rectangle, named as Compute, and an event bubble to note that Compute is finished at the end of the function. The additional text in the popup window of the state rectangle can show the process name and its index argument.

The following functions are independent events, which we cannot represent as a state rectangle. Thus, they are represented as bubbles, and the information shown in the related popup window is described. In all cases, the line number where the function was called is reported.

3. **PI_ChannelHasData**

This function is used to return a value (1 if the channel has a queued read, 0 if it does not) to indicate whether the specified channel can be read without causing a block. The return value of the function is shown in the information popup.
4. PI_TrySelect
This function is the non-blocking version of PI_Select. If no channels are ready to read, it will immediately return -1. The return value of the function is shown in the information popup.

5. PI_Log
This function is used by the programmers to enter textual remarks in the log. Since it is an independent notation, we cannot represent it as a state rectangle. The loggable text is shown in the information popup.

6. PI_StartTime and PI_EndTime
These functions are used by the programmer to time segments of their application. We cannot control how they are called (e.g., StartTime could be called once followed by several calls to EndTime) so drawing state rectangles is not feasible. Therefore, they are represented as event bubbles. For PI_EndTime, the elapsed time (since the last call to PI_StartTime) is shown in the information popup.

7. PI_Abort
This function is used by the programmer to halt execution of the Pilot program on all nodes, typically after a fatal problem has been detected by one process. Ideally, we would like to enter PI_Abort in the MPE log, but there is a problem. PI_Abort calls MPI_Abort, which shuts down all MPI message infrastructure. There is no other way for the programmer (and Pilot) to terminate a cluster program’s execution except to call MPI_Abort, which enlists MPI in killing the program executing on every rank.

Unfortunately, MPE uses MPI messages to collect the per-node in-memory logs and combine them into a single file written by rank 0. Thus, when MPI_Abort is called,
there is no way to avoid the loss of the MPE log. One might imagine that this is a recognized problem for MPE, yet we were unable to find MPE literature dealing with the issue. So we do not currently provide any logging treatment for PI_Abort. Pilot’s existing native log does not have this vulnerability because it writes each log entry onto a disk file when it is received (making it far less efficient than MPE logging).

After designing the plan of adapting MPE logging calls for Pilot, the following chapter will show the approach of how to integrate and implement these logging calls into Pilot.
Chapter 4
Integrating MPE into Pilot

This chapter presents the implementation methodology used to integrate the planned use of MPE (described in Section 3.3) into Pilot in a sound way from a software engineering standpoint. On the one hand, a prime goal was to insert all MPE interactions into the Pilot code base so that use of MPE will be transparent to the user's application (i.e., we don't make them write any MPE calls themselves). On the other hand, Pilot is now a “legacy code,” therefore it is important to understand and accommodate its existing patterns and structures such that successive modifications do not tend to make it less maintainable.

Accordingly, this chapter starts with Section 4.1 analyzing the legacy features that figured in the choices and trade-offs concerning the software design for integrating MPE. Section 4.2 details where and how the MPE logging calls were inserted into the Pilot code base. Along the way, a series of problems were encountered due in no small part to the scanty documentation and examples provided with MPE. These are described in Section 4.3 along with their solutions. Finally, Section 4.4 gives a comparison of the new MPE/Jumpshot facility with Pilot's native logging feature.

4.1 Pilot as a legacy code
As stated earlier, we need to recognize and fit in with Pilot's current source code organization rather than introduce styles that are out of character, in order to keep the code maintainable. In addition, integrating MPE's fresh logging capability cannot break the existing
logging feature nor the deadlock detector which is closely tied to it. In the subsections below are described Pilot's source file organization, run-time execution phases, and the operation of its existing logging feature.

4.1.1 Run time option selection
Pilot uses command line arguments to turn on run time options such as logging. When the user’s application calls PI_Configure( &argc, &argv ), Pilot removes any arguments that it recognizes and processes them. (Then it calls MPI initialization with &argc and &argv so it can do the same with any MPI-specific options that the user may have specified.) To be consistent with that approach, we add another option abbreviation to the “-pisvc=” command line argument: “j” standing for “Jumpshot log”. This is the same way that Pilot’s native call logging is enabled presently: -pisvc=c. Options can be combined, e.g., -pisvc= cj. MPE logging will not be enabled by default because of the possible overheads involved, in terms of execution time (every time a Pilot function is called, and upon termination to collect the single log), memory consumed on each node for accumulating MPE events, and disk space for the combined logfile. These overheads are measured and reported in Chapter.

4.1.2 Source file organization
Pilot is written in C, and its communication calls follow C’s syntax for fprintf and fscanf. It does have a FORTRAN API, but that’s a thin layer that redirects calls to the C API. All of the relevant code is in pilot.c, with some data and macro definitions in pilot_deadlock.h. The public API is defined in pilot.h and no changes are needed there. The deadlock detector, also unaffected, is in pilot_deadlock.c. Therefore, all the changes will be introduced
into pilot.c, while adding some additional fields to data structures defined in `pilot_private.h`.

In pilot.c and pilot_private.h, all the calls and parameters related to the MPE logging facility were set up to make building Pilot with MPE optional by means of conditional compilation macros. The administrator can choose whether or not to install MPE at any site. In practice, the inserted MPE functions are wrapped by “#ifdef” and “#endif”. If the user asks for an MPE log (-pisvc=j) but without MPE being built in their Pilot installation, a warning will be printed to show that logging for Jumpshot is not available. In this way, Pilot does not gain a hard dependency on the MPE library.

4.1.3 Run-time execution phases

MPE usage requires three steps: 1) invoking MPE initialization function and defining event variables for later use; 2) calling MPE functions to record the start/end of each event in the log; 3) invoking MPI finalization to collect the log data from all the processes and write out the CLOG-2 file.

Compared with the two phases of Pilot application execution which has been mentioned in Section 2.2.1, the three MPE steps also follow the order of starting with configuration and being followed with execution. Therefore, they can fit into the Pilot phases nicely. According to that, the first step of MPE can be inserted at the end of the Pilot configuration phase; the second can be used to wrap each Pilot function call in the Pilot execution phase; and the last can be placed where the Pilot execution phase finishes. The details will be explained in Section 4.2.
4.1.4 Operation of existing logging feature

The existing deadlock detection and call logging features both use the same software pipeline:

- Loggable Pilot API events (say, a user call to PI_Write) are recorded in pilot.c using the LOGCALL macro (defined in pilot_private.h).

- If call logging and/or deadlock detection was selected at run time (-pisvc=c and/or d), LOGCALL grabs some information from the environment (such as the application's source file and line number) and formats a call to LogEvent() in pilot.c with some text describing the event.

- LogEvent() breaks up long text into PI_MAX_LOGLEN length messages, if necessary, and sends them using MPI_Send to the special online process running on rank 1.

- That process receives log messages from PI_MAIN and worker processes, and time-stamps each one upon receipt. If call logging is enabled, it writes the message to the logfile. If deadlock detection is enabled, it calls the detection function to update its dependency graph with the new event and to check for cycles.

Due to this tight integration of existing call logging and deadlock detection with an independent Pilot process, it was decided that it was better not to try integrating MPE calls into this pipeline, but instead to call MPE directly from the same places in pilot.c that LOGCALL is currently being called, even though this would result in some redundant coding. In this way, we avoid breaking or complicating the existing mechanism that works well.

Furthermore, MPE does not need an intermediary process, as Pilot logging does, to collect events in one file since it has a completely different architecture: Rather than writing
each event to the logfile when it occurs, it collects event data in memory on each MPI rank, spilling events to /tmp files as needed, until log finalization, at which point it uses MPI to collect all the saved event data and merge it onto a single CLOG-2 file while synchronizing the time stamps of each rank. Thus, there was no reason and no advantage to involve Pilot's special logging process, which made it simpler to integrate MPE into Pilot.

While we now have a superior logging facility with MPE, we still want to retain the existing Pilot logging feature. The reasons are as follows:

1. If some site doesn't want MPE or can't install it, they can still use the existing feature.
2. Even if they did have MPE, maybe the user just wants a quick look at a text file without involving a Java-based viewer (Jumpshot) or there's a problem with their Java installation.
3. Since deadlock detection uses the same mechanism as call logging, almost no code is saved by deleting just the latter.
4. There could be future uses for the existing log, or it may be enhanced.

After drawing a positive conclusion from investigating whether MPE could meet with Pilot’s legacy features, considerations about where to place MPE calls within Pilot’s source code will be discussed in the next section.

4.2 Where to put the MPE calls

Since a goal of logging is to display the execution of Pilot programs accurately, it is important to find appropriate positions for inserting MPE calls into pilot.c. First, Section 4.2.1 will explain some predefined variables used for MPE configuration purposes. Then, in Section 4.2.2 the position to place the basic MPE initialization and finalization functions
will be described. Next, the details of the loggable Pilot functions will be described in Section 4.2.4 and Section 4.2.5 in terms of the three categories of Pilot functions presented in Section 3.3.1. After that, Section 4.2.6 discusses how to implement the extra information required by each state rectangle or event bubble, and finally Section 4.2.7 tells how to customize colour use in Jumpshot if a user is not satisfied with the preprogrammed defaults.

4.2.1 Streamlining MPE configuration

After some simple experiments, it was found that the configuration steps of MPE are too complex to apply to the state rectangles and event bubbles needed for each Pilot function individually. The optimum way is to create a table-driven approach to initialize the states and events for all cases at once. The approach is as follows:

To configure states and bubbles for logging, MPE_Describe_info_state and MPE_Describe_info_event need to be called to generate and return internal event IDs required for use in subsequent calls to MPE_Log_event. At configuration time, we save these IDs in two arrays. The one for states, named mpe_statese (meaning “state start and end”), holds a pair of IDs, one for marking a state's start and the other to mark its end. Similarly, another array mpe_event holds the IDs for recording event bubbles. In order to ease the indexing of correct entries in these arrays, we define two kinds of preprocessor symbols LOG_NAME and LOG_NAME_MSG corresponding to each Pilot API function PI_name. For example, the symbols for PI_Write are defined as LOG_WRITE and LOG_WRITE_MSG. That is, the IDs needed to log the start and end of a PI_Write call would be mpe_statese[LOG_WRITE][0] and [LOG_WRITE][1], and the ID for its event bubble is mpe_event[LOG_WRITE]. Two exceptions to this pattern are the symbol
LOG_CHANNEL, used to record the channel number for message passing, and LOG COMPUTE, used for the state mentioned in Section 3.3.1.

### 4.2.2 MPE logging initialization, configuration and finalization

The MPE logging facility should be invoked only after MPI has been initialized, so MPE_Init_Log should be called at the end of PI_Configure (which calls MPI_Init_thread). Then the configuration of MPE state rectangles and event bubbles (described in the previous section) are placed right after MPE_Init_Log. For MPE log finalization, MPE_Log_sync_clocks and MPE_Finish_log are called in the very beginning of PI_StopMain before any other MPI functions. The reasons behind that are, firstly, there will not be any further Pilot API events to record; secondly, since MPE uses MPI to collect the log data, its calls have to come before Pilot shuts down MPI; and thirdly, just in case anything went wrong during Pilot or MPI termination, at least we can be sure that the MPE log is safely written out.

### 4.2.3 Strategies for placement of MPE calls relative to Pilot actions

Since MPE logging calls cannot be executed simultaneously with the internal actions of Pilot functions, there exists a slight time difference between them. In order to make the start/end times of Pilot function calls viewed in Jumpshot as accurate as possible, erring on the side of overstating durations, some strategies were developed: For state rectangles, we create the left edge immediately upon entering the Pilot function, and the right edge just before returning from it. For event bubbles of Pilot input and output functions, we place the bubbles right before messages have been sent and, and right after being received, respectively. For event bubbles of Pilot administrative functions, the placement is not
consequential because such functions are brief compared to the blocking potential of I/O actions. As explained below, our use of MPE can come close to this ideal.

### 4.2.4 Logging Pilot’s input and output functions

Pilot input functions are PI_Read, PI_Reduce, PI_Gather, and PI_Select, whereas Pilot output functions are PI_Write, PI_Broadcast, and PI_Scatter. According to the plan for using graphical objects in Section 3.3.1 and Section 3.3.2, they can be represented by state rectangles, message-passing arrows, and event bubbles in Jumpshot.

#### 4.2.4.1 State rectangles of Pilot input and output functions

These seven functions are intended to be presented as state rectangles, so the positions to place the start and end points, which form the left and right edge of the rectangle, should be decided first.

Since these functions are all Pilot I/O functions, they will all be diagnosed for the argument error checking before the message has been passed or received. If any errors are detected, the program will be aborted. If not, the LOGCALL macro (in pilot_private.h) will be called to format the message text and pass it to LogEvent(), which in turn invokes MPI_Send to deliver the event to the special online process (for writing to the logfile and/or dispatching to the deadlock detector). Hence, to capture as much processing time as possible, the first MPE_Log_event function, representing the left edge of the state rectangle, is called right after the argument error checking and before LOGCALL being called. Correspondingly, the second MPE_Log_event function, representing the right edge of the state rectangle, is called at the end of each Pilot I/O function when it is totally finished. These two MPE function calls must be carefully matched in order to properly complete each state
rectangle as mentioned earlier in Section 2.4.1 and Section 3.2.1. If this is neglected, it results in an error “No matching State end-event for Record RecHeader” when MPE’s CLOG-2 logfile is converted to SLOG-2 format for Jumpshot viewing. This error message means that the ID numbers of MPE_Log_event for logging the start point and the end point were not equivalent.

Besides the basic state rectangle for these seven Pilot I/O functions, extra information can be recorded and shown in the associated popup window since the state rectangles were initialized by calling MPE_Describe_info_state. Therefore, before the matching call to MPE_Log_event, MPE_Log_pack can be called to store the extra logging information. According to the design as mentioned earlier in Section 3.3.2, the additional texts in each popup windows of state rectangle of these I/O functions show the name of the calling process, the index argument, and the line number of the Pilot call. Since all the line numbers of where Pilot functions have been called will be shown in the popup windows of their own graphical objects, repetitions will not be in the following paragraphs. Then the name of the bundle is also presented if applicable. For PI_Select we want to report which channel was selected, which could be important debugging information.

4.2.4.2 Message-passing arrows of Pilot input and output functions

As mentioned in Section 2.4.2, message passing is represented by arrows in Jumpshot, and MPE_Log_send and MPE_Log_receive are used to log the tail and the head of an arrow, respectively. Therefore, for all the Pilot output functions, MPI_Log_send is called just before the message is sent (e.g., MPI_Send), since this is the point in time when the I/O function is about to send data. Whereas for all the Pilot input functions, MPI_Log_receive is called just after the message is received (e.g., MPI_Recv), since this is the point in time
when the communication is finished. In this way, a whole arrow spanning two processes and representing the full duration of the communication is formed for Jumpshot to display.

PI_Write and PI_Read can be called both for either point-to-point or collective communication. That is to say, PI_Write and PI_Read will be participants in collective operations such as PI_Gather and PI_Broadcast if their respective channels arguments are part of a bundle. (PI_Gather, PI_Broadcast, and other collective operations take a bundle argument.) Thus, in PI_Write, MPE_Log_send is called in every case where a message is sent to a PI_Read, PI_Gather, or PI_Reduce. While in PI_Read, MPE_Log_receive is called in every case where a message is received from a PI_Write, PI_Broadcast, or PI_Scatter.

In contrast to PI_Write and PI_Read with their multiple use cases, the four collective Pilot I/O functions using bundles only have one, since PI_Gather and PI_Reduce receive from PI_Write, and PI_Broadcast and PI_Scatter send to PI_Read. If a bundle has, say, 10 channels, then Jumpshot will display 10 message arrows involving 11 processes. Thus, these four Pilot I/O functions need a loop to call MPE_Log_send or MPE_Log_receive, using their bundle size as the number of iterations, otherwise, only one arrow spanning two processes would be shown in the graphical representation.

4.2.4.3 Event bubbles of Pilot input and output functions
As mentioned in Section 3.2.2, neither the data length and the value of the first element nor the name of the involved channel(s) can be logged by MPE_Log_send or MPE_Log_receive in message passing. To make up this deficiency and display the information, we decided to use popup windows of event bubbles and place them in the relevant state rectangles of Pilot I/O functions. Therefore, MPE_Log_pack for recording these
information and MPE_Log_event for creating the event bubble are called right after MPE_Log_receive if it is in the Pilot input function, or before MPE_Log_send if it is in a Pilot output function.

4.2.5 Logging Pilot’s administrative functions

The remaining loggable Pilot functions which are designed to be shown in Jumpshot are the administrative functions including PI_Configure, PI_StartAll and PI_StopMain (Compute), PI_ChannelHasData, PI_TrySelect, PI_Log, PI_StartTime. According to the plan for using graphical objects in Section 3.3.2, they can be presented by state rectangles or event bubbles in Jumpshot.

4.2.5.1 State rectangle of Pilot administrative functions

There are two cases of administrative functions which can be shown in state rectangles, PI_Configure and Compute, the latter representing the pair of PI_StartAll and PI_StopMain. Since both of them are shown as state rectangles they should follow the same rule to insert the MPE logging calls as mentioned in Section 4.2.3. Also, the name of the calling process and the index argument are logged as the extra information in each of their state rectangles as well. The details are as follows.

For PI_Configure, the first MPE_Log_event, representing the left edge of the state rectangle, is called right after all the MPE state rectangles and event bubbles are initialized in PI_Configure, since it cannot be called before MPE initialization is finished. The matching MPE_Log_Event is called at the very beginning of PI_StartAll before Compute is called. Actually, it is not precise and slightly late to log the end point of PI_Configure here, but there is no other good earlier place to log it. The matching MPE_Log_Event cannot be
called at the end of any other Pilot functions (which are to create processes, channels, or bundles, or set up the name of them) between PI_Configure and PI_StartAll, since these functions may be called repeatedly and not in any fixed sequence.

   For Compute, the first MPE_Log_event is called right after the end of the state of PI_Configure and before PI_StartAll does anything else. The matching one is called in PI_StopMain right before MPE_Log_sync_clocks and MPE_Finish_log.

4.2.5.2 Event bubbles of Pilot administrative functions
   The use of event bubbles for Pilot administrative functions is to note important information produced by the rest of the Pilot loggable functions. They will produce extra information (at least the line number where they have been called in a Pilot program) in the popup window.

   For PI_StartTime and PI_EndTime, the purpose of recording their calls with event bubbles is to annotate the visual log in case the programmer finds it useful (they may be inserting timing calls along with important phases of computation). Since the MPE calls have to come either before or after the resulting calls to MPI_Wtime, the times recorded in the bubbles will be slightly different from the times returned by MPI. MPE_Log_event is called in the beginning of PI_StartTime before MPI_Wtime is called. Since this event bubble is just to log the time, which is shown in the popup window anyway as default information, there is no need to add any other information into it except for the line number of its Pilot call.

   For PI_EndTime, the elapsed time (since the last call to PI_StartTime) returned by the function is shown as additional information. Thus, MPE_Log_pack to record the time
duration and MPE_Log_event for creating the event bubble are called right after the elapsed time being calculated in PI_EndTime.

For PI_TrySelect we want to report which channel was selected, or -1 if none was ready to read, which could be important debugging information. Thus, MPE_Log_pack to record the index of the channel returned by PI_TrySelect and MPE_Log_event for creating event bubble are called right after the bundle is checked.

PI_Log is used by the programmer to deliberately insert text into the Pilot log and/or the MPE log. We use an event bubble to display that text. Hence, MPE_Log_pack to the text and MPE_Log_event for creating the event bubble are called right after the text has been sent to LogEvent in PI_Log.

For PI_ChannelHasData, this function is used to report the result of whether the specified channel can be read. Hence, it can be treated the same as PI_TrySelect.

4.2.6 How to implement showing the extra popup information

In order to display extra information in a popup window of a state rectangle or an event bubble, there are four relevant functions (see Table 2-3): MPE_Describe_info_state, MPE_Describe_info_event, MPE_Log_pack, and MPE_Log_event Since the positions where to insert them have already been introduced in the previous paragraphs, this section is going to show the basic essentials to assign the parameters for extra information into these functions in pilot.c, and then describe how to get the particular data to be logged in terms of the requirements of each popup window. It is done in a roundabout three-step fashion.
First, as mentioned in Section 4.2.2, MPE_Describe_info_state/event is used to initialize a state rectangle or an event bubble in the MPE configuration. Those functions’ fifth parameter specifies a format control string following C’s printf style with placeholders marked by escape characters (“%”). They specify both the relative locations and the data types where content will be inserted for a particular state or event instance. For example, to initialize the state rectangle of PI_Write, the fifth parameter is “%s (%d) L.%d”. Those three placeholders prepare for the process name, process index, and source line number to be inserted when PI_Write is called.

Second, when an event is to be logged, MPE_Log_pack is called to store the data as many times in succession as there are placeholders in the associated format string to be substituted by variable values. The function has five parameters: The first, bytebuf, is a local buffer to be filled, of type MPE_LOG_BYTES (which gives it a fixed length of 32 bytes). The second, position, is an int pointer that tracks the next available position in the buffer, and should be initialized to 0 prior to the first call. The third parameter specifies a tokentype (e.g., ‘s’) to match with a placeholder in the format string (“%s”). The fourth and the fifth parameters give the count (1 for a scalar, more for an array) and address of the data to be logged, respectively. With each call to MPE_Log_pack, the next matching placeholder is substituted with the value(s) of the data converted to ASCII according to the tokentype. For the example in the last paragraph, ‘s’, ‘d’, and ‘d’ should be set in three successive MPE_Log_pack calls to substitute values for the three placeholders.

Third, after using MPE_Log_pack to store the data in bytebuf, then MPE_Log_event can be called with bytebuf as its argument, to enable the state rectangle or the event bubble to log the extra information in its popup window.
After giving the fundamental use of logging extra information in a popup window of a state rectangle or an event bubble, Table 4-1 lists the approaches to obtaining the required values which need be logged for each popup window. It should be noted that some of the listed format control strings were found to trigger problematic displays by Jumpshot as described Section 4.3.3. Those with problems were prepacked via sprintf and then a simple “%s” format string was used. Also in the table entries, PL_CallerLine refers to Pilot’s global variable that is automatically set to a source file line number via the preprocessor macro __LINE__ whenever a call is made to a Pilot API function.

4.2.7 Customizing colour use in Jumpshot

To implement and simplify the design in Section 3.3.1, we created a header file for color assignments in MPE logs, called pilot_log_colors.h. Pilot functions are classified into eight categories of state rectangle and events bubbles using a default colour scheme given in Table 3-1. If users are not satisfied with the default colour setting, they can alter the colours by modifying this file (always choosing valid names taken from the X11 colour chart [Com]) and recompiling Pilot.

4.3 Problems and solutions

While carrying out the design of integrating MPE into Pilot, it is wrong to say that the implementation was without any problems. Several problems emerged during the experiments. Some of them have been solved satisfactorily, but some of them still remain. The problems and their solutions are described next.
<table>
<thead>
<tr>
<th>Name of each MPE state rectangle or event bubble</th>
<th>Contents need to be logged</th>
<th>Printf style format control string in MPE_Log_info_state/event</th>
<th>Approach to obtain the required value for each popup window</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State rectangles</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_WRITE</td>
<td>1. The name of the calling process; 2. The index argument; 3. The line number of the calling place.</td>
<td>“%s (%d) L.%d”</td>
<td>1. thisproc.processes [thisproc.rank].name; 2. thisproc.processes [thisproc.rank].argument; 3. PI_CallerLine</td>
</tr>
<tr>
<td>LOG_READ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGCONFIGURE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_BROADCAST</td>
<td>1. The name of the calling process 2. The index argument 3. The name of the calling bundle 4. The line number of the calling place</td>
<td>“%s (%d) %s L.%d”</td>
<td>Same as above except for the name of the calling bundle. 3. b-&gt;name.</td>
</tr>
<tr>
<td>LOG_SCATTER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_GATHER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_REDUCE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_SELECT</td>
<td>1. The channel number having one or more message to be sent; 2. The line number of the calling place.</td>
<td>“%s (%d) R.%d %s L.%d”</td>
<td>1. i (refers to a channel number); 2. PI_CallerLine.</td>
</tr>
<tr>
<td>LOG_COMPUTE</td>
<td>1. The name of the calling process; 2. The index argument.</td>
<td>“%s (%d)”</td>
<td>Same as the first one except for the line number of the calling place, since Compute is not a Pilot function, mentioned in Section 3.3.1.</td>
</tr>
<tr>
<td><strong>Event bubbles</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_WRITE_MSG</td>
<td>1. The data length and the value of the first element.</td>
<td>“Data %s”</td>
<td>1. mybuff obtained from interpArg (mybuff, PI_MAX_LOGLEN, “code”, &amp;mpiArgs[i] ). (Note: “code” refers to the abbreviation of Pilot I/O calls; for example, “Wri” refers to PI_Write.)</td>
</tr>
<tr>
<td>LOG_BROADCAST_MSG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_SCATTER_MSG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_GATHER_MSG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_REDUCE_MSG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG_READ_MSG</td>
<td>1. The channel number to pass a message.</td>
<td>“Channel %s”</td>
<td>1. c-&gt;name.</td>
</tr>
<tr>
<td>LOG_STARTTIME</td>
<td>1. The line number of the calling place.</td>
<td>“L.%d”</td>
<td>1. PI_CallerLine.</td>
</tr>
</tbody>
</table>
When experiments were conducted on the SHARCNET cluster called mako, which has JDK version 1.6, there was a problem of rendering fonts, and this problem will also occur in such kind of computers or clusters not supporting a higher version JDK. Figure 4-1 depicts the issue in detail. It can be observed that the font is in italics, which is neither expected nor desired. This issue affects all the windows in Jumpshot. In addition, the title text of the popup window is truncated with “...”, and this issue takes place on various buttons and menus as well.

### Table 4-1. The approach of logging the required data into the corresponding popup windows (Cont.)

<table>
<thead>
<tr>
<th>Name of each MPE state rectangle or event bubble</th>
<th>Contents need to be logged</th>
<th>Printf style format control string in MPE_Log_info_state/event</th>
<th>Approach to obtain the required value for each popup window</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG_ENDTIME</td>
<td>1. The elapsed time from PI_StartTime; 2. The line number of the calling place.</td>
<td>“Elapsed time %E Sec. L.%d”</td>
<td>1. elapsed_time; 2. PI_CallerLine.</td>
</tr>
<tr>
<td>LOG_TRYSELECT</td>
<td>1. The negative result (“-1”) when no channel has data to pass in a bundle; 2. The line number of the calling place.</td>
<td>“TrySelect result %d L.%d”</td>
<td>1. tryselect_mpe (with the value “-1”); 2. PI_CallerLine.</td>
</tr>
<tr>
<td>LOG_CHANNELHASDATA</td>
<td>1. The result of whether the specified channel can be read (1, if the channel has a queued read; 0, if the channel does not have a queued read); 2. The line number of the calling place.</td>
<td>“HasData %d L.%d”</td>
<td>1. flag; 2. PI_CallerLine.</td>
</tr>
<tr>
<td>LOG_LOG</td>
<td>1. The text logged in a PI_Log call; 2. The line number of the calling place.</td>
<td>“%s L.%d”</td>
<td>1. text; 2. PI_CallerLine.</td>
</tr>
</tbody>
</table>

### 4.3.1 Fonts in Jumpshot

When experiments were conducted on the SHARCNET cluster called mako, which has JDK version 1.6, there was a problem of rendering fonts, and this problem will also occur in such kind of computers or clusters not supporting a higher version JDK. Figure 4-1 depicts the issue in detail. It can be observed that the font is in italics, which is neither expected nor desired. This issue affects all the windows in Jumpshot. In addition, the title text of the popup window is truncated with “...”, and this issue takes place on various buttons and menus as well.
After checking the contents of the CLOG-2 and SLOG-2 logfiles by comparing with the original code in pilot.c, it was identified that there was no data loss during logfile production and conversion. Therefore, the version of Java came under suspicion. After running Jumpshot with Java version 1.8, all the windows have been shown in a suitable font without any truncation of words. The result is shown in Figure 4-2 concerning the same popup window, but without any issues of the font.

There are three possible solutions for this problem: The first one is to download the CLOG-2 or SLOG-2 logfile from the cluster to another computer with JDK 1.8 or higher, and run Jumpshot there. This solution is especially suitable when the cluster does not support Java anyway, or is strictly for computation and does not encourage exporting live X sessions. The second is to install a JDK of 1.8 or higher on such computers or clusters that need to execute Jumpshot. However, both of these two solutions have their deficiencies.
The process of the first one is more complex, since it requires Jumpshot to be installed on another computer and takes additional steps so that logfile cannot be viewed immediately after the Pilot program has been executed. Whereas for the second solution, to run Java version 1.8 needs a large amount of memory, more than a cluster’s login node may supply (and this is the case with mako). Thus, a Pilot user who wants to run Jumpshot on mako needs to ssh to one of the compute nodes that does have a lot of memory and run the application there. However, this practice runs afoul of system policy that compute nodes are for computation, and in an instructional setting, the students could occupy a large number of compute nodes running Java interactively.

The third solution is provided by SHARCNET’s Visualization Systems. They have large amounts of memory, high-end graphics cards, and can be accessed through a web browser by means of a VNC (Virtual Network Computing) session, so that the viewer’s computer does not need a lot of memory. Since the visualization systems mount the user’s home directory which is shared across all SHARCNET clusters, it is a simple matter to generate a logfile on mako or another cluster, and view it on one of these systems.

It should be noted that the above issue was specific to SHARCNET, though it might occur on another HPC cluster. But if a user were, for instance, to install MPI, Pilot, MPE, and Jumpshot on a multicore Linux laptop with a recent version of Java, this issue would not arise.

4.3.2 Distortions of bubble and arrow positions
When event bubbles and arrows are created within an extremely short time period, it was found that they could end up superimposed upon each other. This condition can also raise
a warning message called “Equal Drawables” when converting CLOG-2 file to SLOG-2 file. The message indicates that two or more graphical objects having the same eventID have identical start and end times. This can result from the limited resolution of MPI_Wtime (returning wallclock time in double precision seconds), such that multiple MPE logging functions are too adjacent in global time for the MPI library to distinguish them [Cha11].

An example is shown in Figure 4-3, which uses an earlier colour scheme (we now show I/O bubbles in yellow only). In the first line (which stands for the main process), the one orchid and two green state rectangles represent the PI_Scatter and PI_Broadcast functions, respectively, and the several event bubbles in wheat and cyan show message passing to each work process. Normally, the number of bubbles should appear as a set of five, corresponding to the five work processes that receive the messages, and a message-passing arrow should start from the position each bubble’s position (with some very slight

![Figure 4-3. Picture of distorted bubble and arrow positions](image-url)
deviation) However, in the picture, some of the message-passing arrows start from the same event bubble as another arrow. In addition, there are only four medium blue bubbles in the third and fourth sets of them. This problem emerges, like in this example, particularly in the loop that calls MPE_Log_send/receive for Pilot collective functions, such as PI_Broadcast and PI_Gather, and it results in superimposed (lost) event bubbles and wrong-position message-passing arrows in the graphical representation.

To prevent this problem of superimposed objects, a compromise way is to artificially spread the time of each iteration in the loop by inserting delays using usleep() (found in unistd.h). The result is shown in Figure 4-4. It is observed that the picture now looks regular, especially its arrows. This is achieved by setting just 1 ms in usleep(), but the result is still very noticeable, and yet the injected delay (6 ms in this example) hardly impacts the program’s execution. Also, the previous arbitrary order of drawing arrows has been reversed to avoid the crossed lines. However, there is a drawback to the solution. Since the MPE logging functions cannot be inserted into the MPI library itself, but have to be

Figure 4-4. Picture of showing collective function using usleep()
executed by Pilot, each iteration in the loop is not executed together with the real message passing. In other words, the arrows and event bubbles shown in the graphical representation have some delay from the real position where the messages were sent or received. Nevertheless, the result shown in the picture is more consistent with what Pilot collective functions do than the superimposed bubbles and arrows. Moreover, while in this case there are five messages going out, they may not necessarily all go directly from the master to each of the workers. MPI might organize the workers into a distribution tree, then start by sending one message and have other processes forward messages to save time, so the key point is that these messages are going to the receivers but they do not necessarily go in a predictable order. Thus, the sequence of message passing shown in the picture is in an arbitrary order which may not be completely accurate, but there is no way to make it in the real order because we do not know “under the hood” how MPI is going to send the messages. Ideally, what we want is to have all of the arrows emanating from the close or superimposed bubbles, but then we run into other problems (Equal Drawables).

Another solution that was considered for eliminating the superimposition problem was by simply using a single bubble on the common (collective operation) end of the bundle. This is suitable for PI_Broadcast, which sends the same data to every receiver and only needs to report the data once, and for PI_Gather/Reduce, which are receivers and do not need to report (since the data is reported on the send side of the arrow). But PI_Scatter breaks the pattern by sending different data to each receiver, and we would not be able to report all the different data, but maybe we could compromise and just show the first channel’s data.
4.3.3 Extra information out of order in popup windows

After succeeding in logging Pilot functions into graphical objects and displaying them in Jumpshot, we realized that some text in the popup windows was not shown in the proper order. In further experiments, the problem was found that in the printf style format control strings, if there was an escape character placed at the \textit{beginning} of the whole string, and one or more characters of literal text placed later, the latter text would be shown first. For example, if a string was \texttt{"%s logged"}, the extra information in the popup window would come out as \texttt{"logged \%s"}.

At first, we tried to work around the problem by adding more blanks in the string, but this failed. Then, suspecting that this was an MPE bug, we checked the ASCII content logged in both CLOG-2 and SLOG-2 files, and found the sequence there was correct, so we thought the problem might be caused by an internal operation of Jumpshot. Rather than attempt to change the related code of Jumpshot (which we were committed to using \texttt{“as is”}), we replaced problematic format strings with a simple string tokentype (\texttt{“\%s”}), and used \texttt{snprintf} to store the original pack the output in the correct order. Therefore, the printf format string, listed in Table 4-1, for the first row (LOG\_WRITE/READ/CONFIGURE), has been changed from \texttt{“\%s (%d) L.\%d”} to \texttt{“\%s \%s \%s”}, the second row (LOG\_BROADCAST/SCATTER/GATHER/REDUCE) from \texttt{“\%s (%d) \%s L.\%d”} to \texttt{“\%s \%s \%s \%s”}, the third row (LOG\_SELECT) from \texttt{“\%s (%d) \%s R.\%d L.\%d”} to \texttt{“\%s \%s \%s \%s \%s”}, the fourth row (LOG\_COMPUTE) from \texttt{“\%s (%d)”} to \texttt{“\%s \%s”}, and the last row (LOG\_LOG) from \texttt{“\%s L.\%d”} to \texttt{“\%s \%s”}, just to work around this problem.
4.4 Comparison with Pilot's native logging facility

It is worthwhile looking back and comparing the MPE log with Pilot’s original log facility to find the differences between them. Ideally, the MPE/Jumpshot log should display at least as much information as the textual Pilot log.

Most of the event data that is recorded in the native Pilot log (-pisvc=c) is also captured in the MPE CLOG-2 logfile. In addition, the MPE log features a more accurate global time axis and more complete time information for each Pilot call, and it collects events in a binary format. Most important, the MPE log can be visualized graphically using Jumpshot, whereas the Pilot log (which is based on logging the processes and channels of Pilot program along with the message-passing data in channels) is technically human-readable (since it is text), but is a nuisance to interpret, especially when given more than a handful of entries. Thus, the performance of MPE logging facility provides a more favourable opportunity than the one of Pilot to ease the learning curve for novice scientific Pilot programmers.

What is missing in the MPE log compared to the Pilot log are two items: (1) the message datatype, and (2) the caller’s source file name. The reason is that the size of the MPE message buffer is so limited, it was considered that such items could be left out to conserve space. The message datatype can be easily determined by checking the source line number—and in any case, the data itself is displayed in the correct format—and since calls will tend to be from the same source file, it would be redundant to keep displaying that name.

After accomplishing the implementation of integrating MPE logging functions into the Pilot library, and solving most technical issues, in the next chapter we are going to show
the demonstration result of all loggable Pilot functions and Pilot programs to examine whether MPE and Jumpshot could be used in the educational environment to aid Pilot programmers.
Chapter 5

Experimental Results

This chapter will show several experimental results after the implementation. First, in Section 5.1 the screenshots of all the Pilot calls will be shown to confirm that MPE logging calls are functioning properly in the real execution according to the design and implementation in Section 3.3 and Section 4.2. Then in Section 5.2 an application program which is an assignment for teaching Pilot will be used as a demonstration program to measure how much overhead will be taken up by calling MPE logging calls on a Pilot application. After that, in Section 5.3 two more examples will be produced to support how important MPE logging and Jumpshot can be in helping novice learners to understand the actual run-time message passing between processes, and then to diagnose and fix their logic.

5.1 Screenshots of all the Pilot calls

A program was created covering all Pilot calls which have been decided to display in Jumpshot. The program is only for testing the MPE logging calls inserted into these Pilot functions and without any other particular purpose.

The program uses nine processes: one for main and eight for instances of the same work function. In the main execution step, the program can be divided into three parts according to the three bundles created. First, one PI_Select followed by PI_Reads involve two channels for PI_Write from two work processes to the main process. Second, one PI_Broadcast and one PI_Gather utilize three channels each between the main process and
three work processes. Third, one PI_Scatter and one PI_Reduce involve three channels each between the main process and the remaining three work processes.

After executing the program and converting its CLOG-2 to SLOG-2 format without any errors, Figure 5-1 presents the performance of it which is shown in Jumpshot. From this picture we can clearly see that all the Pilot calls which have been custom-logged via MPE calls are listed on the left in the legend window. They are ranked according to their graphical objects types and alphabetically, and their colours following the design in Table 3-1 are displayed correctly in both the legend window and the main window. Moreover, in the main window, we can clearly see the execution has been divided into four parts corresponding to the code in the test program. The first part is the state rectangle for PI_Configure which is colour bisque (enlarged in Figure 5-2). The other three parts are the main execution and nested state rectangle for Compute which is colour gray. Portions are enlarged in Figure 5-4 to Figure 5-5. By these pictures, we can find all the message passing (represented by arrows) between the processes with great distinctness. In Figure 5-3 the Pilot point to point message-passing functions (PI_Write and PI_Read) use light green and red for their state rectangles, while in Figure 5-4 and Figure 5-5 the Pilot collective message-passing functions (PI_Broadcast, PI_Gather, PI_Scatter, and PI_Reduce) have dark green and red for writing and reading, respectively.

In Figure 5-4 it is clear how the upper long red bar shows PI_Gather receiving data from the workers, shown by the three narrow green bars of their PI_Writes. (The PI_Writes complete quickly, but the PI_Gather is prolonged by the injected millisecond delays in order to properly display the message arrows.) However, the PI_Reduce in Figure 5-5 shows a different pattern: The green bar of process 6 is much longer than those of processes
Figure 5-1. Demonstration program for displaying all Pilot calls in Jumpshot (Home size)

Figure 5-2. Demonstration program zoomed to PI_Configure

Figure 5-3. Demonstrating PI_Write, PI_Select, and PI_Read
7 and 8. The reason is that the messages from the work processes to the main processes are in a reduce bundle. For internal reasons, Pilot carries out a reduction in two steps. The first uses MPI_Reduce in processes 6-8 to collect the result in process 6. The second uses MPI_Reduce in processes 6-8 to collect the result in process 6. The second uses

1. With MPI_Reduce, all callers must provide a “contribution” to the result that will be reduced into the designated “root” process. Even the root process must contribute data. This usage is compatible with MPI’s SPMD style of coding, but not with Pilot’s MPMD style and Pilot’s PI_Reduce does not work that way. The caller does not have any contribution to make, but is simply waiting to receive the reduced result from the worker processes. Therefore, if PI_Reduce were to utilize MPI_Reduce directly, it would have to generate a “fake” contribution. This is problematic because the contribution would need to be an identity value, say, 0 for a “+” operation, 1 for “*”, and so on. It gets worse when the reduced value is an array, since initialized temporary storage would need to be allocated/freed, and completely breaks down when a custom operation (“mop”) is specified, since no identity value could reasonably be inferred. Due to those problems, it was decided to let the workers all call PI_Reduce to carry out the reduction operation, followed by the root process sending the result to the caller of PI_Reduce, even though this is admittedly less efficient than making a single call to MPI_Reduce.
MPI_Send to relay the result to the process that called PI_Reduce. It is because of the second step that process 6 shows a long green bar for PI_Write. Its PI_Write takes longer to process than those of processes 7 and 8 due to the extra step. This shows that the Jumpshot log accurately exposes the timing of Pilot operations while still portraying them in Pilot’s terms and hiding the low-level details of internal MPI calls.

Another question could be raised about process 8’s red PI_Read bar. Why is it longer than those of process 6 and 7, considering they are all executing the same code? The reason is because the compute nodes on the mako cluster have 8 cores each. When the job scheduler launched the 9-process test program, it started processes 0-7 on one node and process 8 on another. Thus, messages passed among the first 8 processes are actually passed in RAM, with consequent low latency. However, as soon as process 8 is involved, messages have to go on the physical network and latency increases. This is an important principle for novice parallel programmers to grasp if they are to understand the true cost of interprocess communication in their programs, and it is clearly shown by the Jumpshot log.

Note that the titles of popup windows will reflect the names of the corresponding Pilot functions, e.g., the title of the popup for a red PI_Read bar will be “PI_Read”. However, users can actually change these names by double-clicking on them in the legend window and typing new text. Such changes will be reflected in the popup titles, and persist only for the current Jumpshot session.

Based on the three kinds of graphical objects, we can subdivide the event bubbles and classify all the popup windows into state rectangles, message-passing arrows, event bubbles associated with a state, and other event bubbles not connected with a state. These objects are all demonstrated in the following sections.
5.1.1 Popup windows of state rectangles

The details of popup windows of state rectangles are shown from Figure 5-6 to Figure 5-14. All these windows show their duration time (msec), start time [0], end time [1], caller’s process number (world_rank), caller’s process name, and index argument (n). (For main, it is not strictly correct to display “main (0)” because main does not have an index argument, however, this has not presently been special-cased.) Except for Compute (since Compute is not actually a Pilot call), they all show the line number where they were called in the application program. For collective functions, the name of the bundle argument is also shown, e.g., “B2@P0” means bundle no. 2 (from the second call to PI_CreateBundle) with its common end at process 0. For PI_Select, its return value (the index of the channel that is ready for reading) is shown.

![Figure 5-6. Popup window of PI_Configure](image)

![Figure 5-7. Popup window of Compute](image)
5.1.2 Popup windows of message-passing arrows

The details of popup windows of message-passing arrows are shown in Figure 5-15 and Figure 5-16. The first one is for message passing from the main process to a work process, and the second is for the one sending back. Both of them show their duration time, start
time, end time, sender’s and receiver’s process numbers (world_rank), channel tag (msg_tag), and the length of the message-passing array (msg_size).

For channels that are not part of a bundle, tag N is assigned to channel no. N, from the Nth call to PI_CreateChannel. Channels that are part of a bundle all have the same tag.
This is a low-level detail that most users would not care about, but msg_tag and msg_size are shown automatically so they may as well give meaningful information.

5.1.3 Popup windows of event bubbles associated with a state

The details of popup windows of event bubbles associated with a state are shown from Figure 5-17 to Figure 5-22. These are information bubbles that mark the end of an
operation. To associate their titles with the matching state rectangles we added “Data” after their state names. For example, the popup title of PI_Broadcast’s bubble is “Broadcast Data”. All these popups show the time when the message finished being sent or received, and most show the length of the data array [N] and the first element (value) with “...” indicating that the array continues. Most windows show integer data, but Figure 5-17 shows char[1] data and Figure 5-19 float[5] data.

Figure 5-17. Popup window of PI_Write bubble

Figure 5-18. Popup window of PI_Read bubble

Figure 5-19. Popup window of PI_Broadcast bubble
There are two special cases which are the name of the bubbles for PI_Write and PI_Read. PI_Write is basic point-to-point message passing and displays the first data value, so we give the name “Message Passing Data” to it. For PI_Read, it shows the channel name (but no data), so we give the name “Message Passing Channel” to it. (One can click on the PI_Write side to find out the data value, so it is redundant to display it on the PI_Read side as well.)
5.1.4 Popup windows of event bubbles not connected with a state

The details of popup windows of event bubbles not connected with a state are shown from Figure 5-23 to Figure 5-27. Since all of them are Pilot functions, the place and the time where and when they are called will be shown in the popup window. Also, according to the design in Section 3.3.2, their respective return values will be shown.

Figure 5-23. Popup window of PI_StartTime

Figure 5-24. Popup window of PI_EndTime

Figure 5-25. Popup window of PI_TrySelect
In summary, all the Pilot functions which are logged by MPE have been tested and executed without any errors. The following section will give an application for demonstration to evaluate the overhead cost of MPE logging calls.

### 5.2 Demonstration application

The application program was used as an assignment for teaching Pilot before. Its purpose is, given more than one thousand of JPEG files, to rapidly produce a corresponding set of thumbnail images (also JPEG). It is designed to use a pipeline of three kinds of processes which are PL_MAIN, multiple Di (standing for decompressor), and C (compressor). PL_MAIN is responsible for opening each .jpg file in the specified input directory, inputting each file as binary data, and shipping it off to the next available worker Di. A D process does the work of decompressing the JPEG data, cropping out the center 32% of the pixel...
array, and then down-sampling the array so as to send only 32% (i.e., every third one) of the pixels to the compressor process C. The C process obtains each thumbnail image as pixels from a D₁ worker, compresses it back into JPEG format, and ships the binary data back to PI_MAIN which writes it to the output directory. A constraint is that only PI_MAIN is permitted to do disk I/O. The application scales by adding additional data parallel D processes, since this is the time-consuming stage of the whole task parallel pipeline.

Figure 5-28 shows the timeline of the application in Jumpshot running with PI_MAIN plus 10 work processes, including 8 decompressors. Since the resulting SLOG-2 file can be successfully read by Jumpshot after calling thousands of Pilot functions without any conversions errors from CLOG-2, it demonstrates that the MPE logging calls are robust in a large and complex program.

![Figure 5-28](image)

**Figure 5-28.** The thumbnail application running with 10 processes shown in Jumpshot

Figure 5-29 shows a zoomed-in portion of Figure 5-28. We can see at a glance that the Pilot I/O functions only take a small proportion of the time, since the red and green colours are tiny in comparison to the gray. In contrast, most of the execution time is used
for computation (the gray state rectangles), namely decompressing, cropping, and recompressing the pictures instead of being idle. This means the parallel application program is well-designed and can be a good experimental case study to measure the overhead of MPE logging calls.

The demonstration is divided into two parts, which are the main execution time and the MPE logfile wrap-up time, since the latter represents a one-time burden that occurs upon program termination. This contrasts with the steady overhead contributed by the MPE logging calls during the program’s execution phase. The MPE logging overhead is compared with that of (1) the four different levels of Pilot’s error checking, (2) deadlock detection, and (3) native Pilot logging. The details are presented in the following sections.

5.2.1 Demonstration of the main execution time

Since the time needed for initialization is not scaled by increasing the number of processes, the main execution time is without Pilot and MPE initialization, and MPE log file wrap-up. The plan to measure the main execution time is shown with nine cases in Table 5-1 by

![Figure 5-29. The thumbnail application shown in Jumpshot (zoomed in)](image-url)
running the application with various combinations of error checking level, deadlock detection, Pilot native call logging, and MPE logging. Then we double the number of the cases by running with six processes (which are one main process and five work processes) and eleven (one main process and ten work processes). Recalling that deadlock detection and/or native logging takes an extra process, this means that for those cases (marked with * in the table), the number of decompressor processes will be reduced by one, resulting in longer run time.

The execution times were obtained by calling PI_StartTime after PI_Configure and PI_EndTime before PI_StopMain. To ensure the accuracy, these 18 cases were run ten times each. We obtained both the average and the median of them, especially for the cases which ran with eleven processes spreading across two nodes. (One node on mako only has 8 cores, and internode communications times will be much greater than intranode communication which can occur in shared memory.) The results are listed in Table 5-1.

<table>
<thead>
<tr>
<th>Experimental cases</th>
<th>5 work processes</th>
<th>5 work processes</th>
<th>10 work processes</th>
<th>10 work processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Level 0</td>
<td>30.676</td>
<td>30.965</td>
<td>14.635</td>
<td>14.41</td>
</tr>
<tr>
<td>Check Level 1</td>
<td>30.784</td>
<td>30.98</td>
<td>14.833</td>
<td>14.38</td>
</tr>
<tr>
<td>Check Level 2</td>
<td>30.701</td>
<td>30.995</td>
<td>15.28</td>
<td>14.44</td>
</tr>
<tr>
<td>Check Level 3</td>
<td>30.651</td>
<td>30.97</td>
<td>15.111</td>
<td>14.42</td>
</tr>
<tr>
<td>Check Level 3_D*</td>
<td>40.296</td>
<td>39.95</td>
<td>17.034</td>
<td>16.415</td>
</tr>
<tr>
<td>Check Level 3_C*</td>
<td>40.663</td>
<td>40.64</td>
<td>16.935</td>
<td>16.2</td>
</tr>
<tr>
<td>Check Level 3_CD*</td>
<td>41.322</td>
<td>41.66</td>
<td>17.031</td>
<td>16.33</td>
</tr>
<tr>
<td>Check Level 3_M</td>
<td>30.297</td>
<td>30.025</td>
<td>14.813</td>
<td>14.42</td>
</tr>
<tr>
<td>Check Level 3_MD*</td>
<td>40.296</td>
<td>39.95</td>
<td>16.943</td>
<td>16.36</td>
</tr>
</tbody>
</table>

Table 5-1. Execution times of the thumbnail application program

a. D: Deadlock detection facility; C: Pilot native call logging facility; M: MPE logging facility; *
*: one less work process due to displacement by the special online process.
There are four noteworthy points in the table:

1. Compared with the results of five work processes, the execution times are less than one half after the number of work processes doubled. This shows that the data parallel division of labour from increasing the number of D type processes is successful in achieving speedup with favourable efficiency.

2. From Checking Level 0 to Level 3 the time does not vary obviously, even when the Checking Level is upgraded from Level 1 to Level 2 (which doubles the message traffic in order to carry out PI_Read/Write format verification). This shows that the overhead of even maximum Pilot error-checking, at least in this compute-intensive cluster program, is of little consequence.

3. The execution times of enabling deadlock detection or Pilot native logging are much longer than the others, since they need one more process to run these facilities, and the former also converts “eager” MPI_Send use to synchronous MPI_Ssend to make the channels obey synchronous CSP semantics for the purpose of deadlock detection. That means there will be one less work process, causing the execution time to become noticeably longer. Therefore, we can deduce that the MPE logging facility saves more time than the Pilot native one under the same condition with limited processes. However, if the program is aborted, a native logfile will still be produced, whereas the MPE logging facility cannot achieve that (mentioned in Section 3.3.2).

4. The Level 3_MD case shows that MPE logging coexists with deadlock detection, as one would hope.

5. The significant comparison is Level 3 vs. Level 3_M, which shows that the overhead for MPE logging is negligible. Even as the number of processes is scaled up, MPE
logging calls only add extremely slight overhead on Pilot programs, at least of a compute intensive nature (which is the norm for HPC production codes). Note, however, that this disregards log wrap-up time, which is the subject of the next section.

5.2.2 Demonstration of MPE log file wrap-up time

After getting such positive results, we also want to detect how much time is required for logfile wrap-up. In contrast with Pilot native call logging, where the time to write the logfile overlaps computation in other processes (but a price is paid by consuming a process just for logging), the output time for the MPE logfile is deferred to the end of program execution where it stands out by itself. With the number of processes being increased, the wrap-up time could be longer, so we want to measure how seriously this added step will impact the total execution time. Strictly speaking, the wrap-up time starts from MPE_Log_sync_clocks and MPE_Finish_log, and ends before PI_StopMain finishes, so measuring the execution time of PI_StopMain is the closest we can get to evaluating the wrap-up time. However, there is an obstacle in that PI_EndTime cannot be called after PI_StopMain because MPI is shut down at that point. Therefore, we use gettimeofday() instead, which returns the current wallclock time. We call it twice, right before and after PI_StopMain.

The demonstration is based on two basic test cases which are Checking Level 3 with and without the MPE logging facility, and the cases are doubled by running six and eleven processes. For each case, we still collected results ten times and obtain the average and the median of them. The experimental results are shown in Table 5-2.
From these data, we find that the overhead for wrap-up (calculated as wrap-up time over total L3 execution time) varied from 2.4% to 5.9%, since the wrap-up time increased slightly with the number of processes while the total time shrank. The key point is that an additional <1 second is not an excessive price to pay to obtain the MPE log.

After reporting on the overhead of inserting MPE logging calls into Pilot programs, the next section will give two instances of how Jumpshot does help to diagnose problems in Pilot programs.

### 5.3 Real instances of using MPE and Jumpshot to help debugging

The two instances were collected from another assignment for teaching message-passing programming with Pilot. The assignment was to read, in parallel, a 316MB .csv file of data on automotive collisions in Canada, with different worker processes starting from different file offsets, and then carry out a series of queries in parallel, merging the results. Figure 5-30 and Figure 5-31 show logfiles from two students’ programs. These students got no speedup, yet they never diagnosed why. But when we view their timelines in Jumpshot, one

<table>
<thead>
<tr>
<th>Experimental cases&lt;sup&gt;a&lt;/sup&gt;</th>
<th>5 work processes average (Sec.)</th>
<th>5 work processes median (Sec.)</th>
<th>10 work processes average (Sec.)</th>
<th>10 work processes median (Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check Level 3</td>
<td>0.026</td>
<td>0.022</td>
<td>0.032</td>
<td>0.031</td>
</tr>
<tr>
<td>Check Level 3_M</td>
<td>0.769</td>
<td>0.765</td>
<td>0.918</td>
<td>0.873</td>
</tr>
<tr>
<td>Log wrap-up time</td>
<td><strong>0.743</strong></td>
<td><strong>0.743</strong></td>
<td><strong>0.868</strong></td>
<td><strong>0.842</strong></td>
</tr>
<tr>
<td>Check Level 3 total (from Table 5-1)</td>
<td>30.651</td>
<td>30.97</td>
<td>15.111</td>
<td>14.42</td>
</tr>
<tr>
<td>Percent overhead</td>
<td>2.4%</td>
<td>2.4%</td>
<td>5.9%</td>
<td>5.8%</td>
</tr>
</tbody>
</table>

<sup>a</sup> M: MPE logging facility.
is able to instantly see their mistakes. To start with, that something is wrong is obvious from the very unfavourable ratio of gray computation to red blocking-read time.

In instance A, file reading runs from 0 to 1.1 seconds, then query processing continues on to 2 seconds. During file reading, the partial overlapping of gray bars show that the program was unable to fully parallelize the I/O. But more seriously, during query processing, it looks like pairs of PI_Write and PI_Read were called for each worker in a loop.

Figure 5-30. Instance A of student’s program in Jumpshot

Figure 5-31. Instance B of student’s program in Jumpshot
instead of all the PI_Writes (to distribute the work parcels) followed by all the PI_Reads. Thus, the program inadvertently serialized the calculations and the workers never did query processing in parallel.

In instance B, the workers were kept waiting till PI_MAIN did 11 sec. of initialization, showing that the program did not succeed in parallelizing reading this big file, so the total run time always stayed nearly the same (since the calculations were fast).

Ironically, these students did not utilize the logging tool after it was demonstrated in a lab session. Nonetheless, when the instructor displayed the timelines to the whole class, others were readily able to diagnose their classmates’ problems. These were not “bugs” in the sense of causing incorrect results, but they were bugs in parallelization that undermined the whole purpose of parallel programming: obtaining correct results in reduced time. Log visualization could also expose load imbalances among the worker processes and help the programmer, for example, to adjust work granularity to provide a more even distribution, or perhaps switch from a static to a dynamic work allocation scheme.

Thus, from the instances shown above, we can illustrate that using the new MPE logging facility and observing the timeline in Jumpshot has the potential to help new learners diagnose their problem in Pilot programs. Aside from the above, the instructor also reported that showing to the class the timeline of simple programs that are used in the half-day Pilot tutorial is an excellent pedagogical aid. It allows one to visually walk through what each process is doing and to see precisely how they interact through message passing. This simple aid brings the abstract into the realm of the concrete, and helps students make the difficult paradigm shift from sequential to parallel programming.
Chapter 6
Conclusion and Future Work

6.1 Summary

Pilot aims to ease the learning curve for novice scientific programmers and keep them away from common MPI pitfalls. Pilot made promising progress after being released. Due to nondeterministic execution order, among other reasons, parallel programs are more difficult to debug than sequential programs. However, Pilot without a standard logfile and visualization tool is not helpful enough for the novice programmers to have an intuitive view to see how their parallel programs work and message-passing performs. In order to aid Pilot programmers, it was necessary to find or develop an appropriate visualization tool. This can be helped by the use of such tools as DDT, Vampir, Pajé, TAU and Vite, for tracing a program's control flow and data flow visually. However, until now, many third-party visualization tools for MPI programs do not perfectly align with the features of Pilot, such as some of them are commercial tools which are not suitable for the educational environment, or some of them would display low-level MPI operations confusing to Pilot programmers using the latter’s higher-level abstractions (such as channels and bundles).

In the process of this research, the MPE library and the portable, Java-based visualization tool Jumpshot-4 were found as a result of extensive searching and experimentation. After formulating a customized design, solving problems that emerged during the experiments using MPE and Jumpshot, and integrating MPE calls into the Pilot library, user
programs can now automatically generate CLOG-2 logfiles with very low overhead suitable for visualization using Jumpshot. Pilot API calls and even message data are displayed. In this way, programmers can understand the actual run-time message passing between processes more clearly, helping to diagnose and fix their logic, and even improve their execution times. After being tested for a considerable time, now this feature is released and was used with an undergraduate parallel programming course, CIS*3090, in Fall 2016, where it proved to be a helpful pedagogical aid. It will be available publicly with the release of Pilot V3.1.

6.2 Future work

Although we have obtained certain progress, there are still some inadequacies that need to be improved and perfected in the future. The future work stemming from this thesis should be conducted as the following two parts.

First, the difficult problem about handling PI_Abort, described in Section 3.3.2, needs to be fixed. Otherwise, the MPE log is in danger of being lost when Pilot detects a fatal error or the user calls PI_Abort.

Second, after collecting the feedbacks of the first assignment in CIS*3090, we find most of the students did not try to generate the MPE log for the extra help, and some of them even complained about the extra steps of logging into the SHARCNET visualization workstations and converting the CLOG-2 file to SLOG-2. Though this phenomenon can be predicted and explained by the fact that the assignment was not considered difficult to debug (from the message passing standpoint) and users can find any additional procedural steps to be something of a barrier to adopting another tool, it still reveals the deficiencies
in user experience. Thus, future work might be possible to focus on designing an interface with higher integration of these two libraries to encourage the students to use MPE and Jumpshot more frequently. Regardless of that, the instructor could also adopt strategies such as (1) designing lab exercises that utilize Jumpshot timelines to diagnose specific problems, or (2) requiring students to try the logging feature and submit screenshots with their assignments. It is likely that this forced exposure would overcome their aversion to a novel tool, and convince them of its usefulness. Certainly the two students whose case studies were shown in the last chapter would have benefited from that.
Bibliography


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Website</th>
<th>Date of Publication</th>
<th>Cited Date</th>
<th>Availability</th>
</tr>
</thead>
</table>
Appendix A: Building and Using MPE with Pilot

Section A.1 tells how to install MPE for Pilot’s use, and Section A.2 how to build user applications when Pilot has been built with MPE. Finally, Section A.3 is the text of user instructions given to the students of CIS*3090 Parallel Programming in Fall 2016. It reflects the SHARCNET environment, and can be adapted as required to local situations.

A.1 Building Pilot with MPE

The Pilot code base needs only to compile with MPE (specifically, mpe.h), and Pilot applications only need to link (not compile) with MPE. The necessary Makefile flags to accomplish this are shown in the sections below.

A.1.1 Installing MPE

After downloading and unpacking the tarball, the configure command is used to set up the build. On SHARCNET clusters (mako and orca have had successful installations), the only options needed are those designating which compilers to use, how jobs are submitted to the cluster scheduler, and where the files should be installed. Beyond that, configure does not need to be told where to find MPI.

The following configure command was used:

```
./configure CC=icc MPI_CC=mpicc F77=ifort MPI_F77=mpif77 \
MPERUN=sqsub --prefix=/work/wgardner/orca-mpe-2.4.8
```
CC and MPI_CC designate the C compiler, the latter for MPI programs. Similarly, F77 and MPI_F77 give the Fortran compiler. We use the Intel compiler on SHARCNET clusters. MPERUN specifies the job submission command, and the prefix option gives the destination directory where “make install” will copy the files.

After the configuration step succeeds, type “make” to build the binaries, then “make install” to place them in the designated directory.

A.1.2 Building Pilot

Very few changes were required in Pilot’s Makefile. A new variable MPEHOME was introduced, which should be set to the “prefix” directory of the above MPE installation. If MPEHOME is set, then the following line is executed to augment CFLAGS:

```
CFLAGS += -DPILOT_WITH_MPE -I$(MPEHOME)/include
```

The preprocessor symbol PILOT_WITH_MPE turns on all the MPE related #ifdef C code in various Pilot source files, and the -I option locates mpe.h. If it is desired to build Pilot without MPE, the simplest method is to define “MPEHOME=” so that its value is empty, then this CFLAGS statement will be disabled.

If the installer wishes to customize the Jumpshot colour scheme, the file pilot_log_colors.h should be modified before compiling.

Compile and link using “make”, or “make cpilot” if the Fortran API is not needed. Then, “make install” will copy the files to the location given in the PREFIX variable.
A.2 Building Pilot applications with MPE

The user’s Makefile needs to set two variables: PILOTHOME (same as PREFIX directory above), and MPEHOME (where MPE was installed). In order to link the Pilot application with the correct MPE library, libmpe.a, the following line is needed:

```
LDFLAGS += -L$(PILOTHOME)/lib -lpilot -L$(MPEHOME)/lib -lmpe
```

With the previous version of Pilot (V3.0) the same line looked like this:

```
LDFLAGS += -L$(PILOTHOME)/lib -lpilot
```

This change is reflected in the specimen Makefile that is distributed with the lab exercises. No other changes are needed.

A.3 User guide

Debugging parallel cluster programs can be a challenge due to lack of visibility into what your processes are doing at any point in time. Pilot can help by outputting a log of all your PL_API calls. After the program runs, there will be a file in the working directory called pilot.clog2 (CLOG2 is the log format). You can visualize this log graphically by running the Jumpshot tool, a Java program. The logfile needs to be converted to another format, SLOG2, for display, but this is a quick step. NOTE: If the program aborted due to an error, the log file might not be created. You can use option `-pisvc=c` (for calls) to get a text-based file pilot.log.

What you will see:

- Y axis with one row per process; X axis with time increasing to the right
- Process history as a series of nested colour-coded rectangles (see Legend window):
• Bisque: configuration phase

• Grey: computation

• Green: output related (e.g., PI_Write, PI_Broadcast)

• Red: input related (e.g., PI_Read, PI_Reduce)

• Arrows showing messages going from one process to another

• Yellow bubbles showing events (e.g., PI_TrySelect result, PI_StartTime)

• Right-clicking on above features brings up info boxes with duration, start/end times, MPI process number (“world_rank”), Pilot process name (with index argument value) or channel/bundle name, data array length and first value, line number of call in source file

Steps to do log visualization:

1. Because running the log viewer requires Java and a lot of memory, it is not feasible to run it on the mako login node. Instead, open a shell on one of the SHARCNET visualization systems. You can leave this shell open while running your job on mako in another window. Open this link in your local browser (scroll down to Visualization Systems): https://www.sharcnet.ca/my/systems

2. Click on a blue screen icon, which starts a VNC client in your browser (this way, the big memory requirement is on the SHARCNET system, which has lots). Login, then open a terminal window. The first time, open your ~/.bashrc file and add the line “export PATH=$PATH:/work/wgardner/bin” so you can find the executables (scripts and jarfiles).

3. “cd” to the directory where you will run your mako job.
4. In your mako shell, **sqsub** your job with Pilot option `-pisvc=j` (stands for Jump-shot). If you want to change the logfile name to foo.clog2, **add** option `-pilog=foo`.

5. After the job completes you should see the .clog2 file.

6. In your visualization shell, type “jumpshot *.clog2” or whatever the logfile name is.

7. Jumpshot will open and ask if you want to convert the .clog2 file to .slog2 format. Click Yes. A Logfile Convertor window will open; click Convert (good status = 0), then OK.

8. Jumpshot will display the log. You can scroll right/left for later/earlier times, and drag the mouse to zoom into features you want to inspect. Right click to bring up info boxes.

    Or, you can connect to the visualization systems from your own computer using a VNC client like TigerVNC. Instructions: [https://www.sharcnet.ca/help/index.php/Remote_Graphical_Connections](https://www.sharcnet.ca/help/index.php/Remote_Graphical_Connections)