INTERACTIVE MESSAGE DEBUGGER FOR PARALLEL MESSAGE PASSING PROGRAMS USING LAM-MPI

by

Hoimonti Basu

Bachelor of Technology (Honors)
Indian Institute of Technology, Kharagpur
1998

Bachelor of Science
San Jose State University, San Jose
2003

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2005

ABSTRACT

Interactive Message Debugger for Parallel Message Passing Programs using LAM-MPI

by

Hoimonti Basu

Dr. Jan B. Pedersen, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Many complex and computation intensive problems can be solved efficiently using parallel programs on a network of processors. One of the most widely used software platforms for such cluster computing is LAM-MPI. To aid develop robust parallel programs using LAM-MPI we need efficient debugging tools. The challenges in debugging parallel programs are unique and different from those of sequential programs. Unfortunately available parallel debuggers do not address these challenges adequately.

This thesis introduces IDLI, a parallel message debugger for LAM-MPI, designed on the concepts of multi-level debugging. IDLI provides a new paradigm for distributed debugging while avoiding many of the pitfalls of present tools of its genre. Through its powerful yet customizable query mechanism, adequate data abstraction, granularity, user-friendly interface, and a fast novel technique to simultaneously replay and sequentially debug one or more processes from a distributed application, IDLI provides an effective environment for debugging parallel LAM-MPI programs.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Matt Pedersen, for helping with theory, directing the implementation of our message debugger, IDLI and spending countless hours in proof reading and correcting the thesis. In addition, many thanks go to the Department of Computer Science at UNLV, particularly Professors Ajoy K. Datta and Evangelos Yfantis for providing financial support. I am grateful to Dr. Venkatesan Muthukumar for being my graduate faculty representative. I thank Professors Ajoy K. Datta and John T. Minor for being on my committee. I would like to thank all my friends at UNLV and in the city of Las Vegas, especially Erik Tribou, Doina Bein, Betty and David Stahl, Dr. Uma Nair and the CS office staff for lending me a helping hand whenever I needed it as well as treating me to delicious meals.

Finally, I want to thank the most important people in my life, my spiritual teacher Ammachi [Mata Amritanandamayi, reverently known as “The Hugging Saint” all over the world], my parents, Amalendu and Anima, my maternal uncle, RaghuRam, my husband, Amal, my brothers, Ardhendu and Dibyendu (alongwith their families), and my spiritual brothers and sisters, for their endless love, constant support and multifarious help. A special expression of gratitude for my husband Amal for his love, kindness, patience, never questioning any of my decisions especially regarding pursuing higher studies and sponsoring my summer tours across USA with our beloved guru Ammachi.

I dedicate this thesis at the lotus feet of our beloved guru Ammachi and to all souls through whom her eternal love and grace has poured into my heart, now and ever.

CHAPTER 1

INTRODUCTION

The craze for more computing power has been one of the main driving forces in the advancement of computers. Traditionally, developments at the high end of computing have been driven by the need for numerical simulations of complex systems such as weather, climate, mechanical devices, electronic circuits, manufacturing processes, and chemical reactions [FOS95]. However, the most significant forces behind the development of faster computers today are emerging commercial applications like video conferencing, computer-aided diagnosis in medicine, parallel databases used for decision support, advanced graphics and virtual reality. These applications require a computer to be able to process large amounts of data in sophisticated ways.

In the past decade there have been tremendous advances in microprocessor technology. Not only are the processors capable of executing multiple instructions in the same cycle, clock rates have increased from about 40 MHz to over 2.0 GHz. Desktop machines, engineering workstations, and computer servers with two, four, or even eight processors connected together are becoming common platforms for design applications. Large scale applications in science and engineering rely on larger configurations of parallel computers, often comprising hundreds of processors. Clusters of workstations that provide high aggregate disk bandwidth are often used by data intensive platforms like database or web servers and applications such as transaction processing and data mining. Applications requiring high availability rely on parallel and distributed platforms

for cost and performance efficient solutions [GRA03]. Although commercial applications may define the architecture of most future parallel computers, traditional scientific applications will remain important users of parallel computing technology.

Trends in applications, computer architecture, and networking suggest a future in which parallelism pervades not only supercomputers but also workstations, personal computers, and networks. In this future, programs will be required to exploit the multiple processors located inside each computer and the additional processors available across a network thus making concurrency a fundamental requirement for algorithms and programs. Because most existing algorithms are suited for a single processor, this situation implies a need for new algorithms and program structures able to perform many operations simultaneously [FOS95]. Therefore we will briefly discuss parallel computing systems.

Parallel and Distributed Systems

A parallel computer is either a single computer with multiple processors or multiple computers interconnected to form a coherent high performance computing platform. Thus there are two basic types of parallel computers (1) shared memory multiprocessor and (2) distributed memory multi-computer. In a shared memory multiprocessor system multiple processors are connected to multiple memory modules through some form of interconnection network, such that each processor can access any memory module which is employed in a single address space. The executable and data of a program is stored in the shared memory for each processor to execute and access respectively, thus enabling a program to access all the data if needed. From a programmer's point of view the shared memory multiprocessor is attractive because of the convenience of sharing data. Since it

is difficult to build hardware that has fast access to all the shared memory by all processors, most large shared memory systems have some form of hierarchical or distributed memory structure. In such a system processors can physically access nearby memory locations much faster than distant ones based on the principle of Non Uniform Memory Access (NUMA) [WIL05].

By connecting computers through an interconnection network we can enable each processor to send messages to other processors thus creating a system of message-passing multiprocessors. Generally such a system consists of self-contained computers that could operate separately which justifies the name distributed memory multi-computer. A common approach for programming for such a system uses message-passing library routines that are inserted into a conventional sequential program for message-passing. A problem is divided into a number of concurrent processes that may be executed on different computers. Different types of interconnection networks which are used in such systems are tree, multistage, mesh, hypercube, and crossbar switch [WIL05]. Also there are systems that are a hybrid of shared memory multiprocessor and distributed memory multi-computer known as Distributed Shared Memory.

Flynn [FLY72] created a classification for computers. Single processor computers which generated a single stream of instructions were termed as a single instruction stream-single data stream (SISD) computers. Multiprocessor systems where each processor had a separate program and one instruction was generated for each program for each processor were termed as multiple instruction stream multiple data stream (MIMD) computers. Computers where a single control unit is responsible for fetching the instructions from memory and issuing the instructions to the processors who execute the

same instruction in synchronism, but using different data were termed as single instruction stream-multiple data stream (SIMD) computers. The fourth combination multiple instruction stream single data stream (MISD) computer does not exist unless one classifies some fault tolerant systems or pipeline architecture in this category [WIL05].

A parallel programming structure which is often used is Single Program Multiple Data (SPMD). In SPMD a single source program is written and each processor will execute its personal copy of this program although independently and not synchronously. Depending on the identity of the computers, parts of the program will be executed on some specific computers and parts on other computers. For example in a master-slave program structure, the program would have parts for the master and parts for slaves, where master and slaves are different computers on the network.

Current trends in parallel computing reveal that message passing is more popular than the shared-memory programming paradigm. Since shared-memory constructs are simpler, using them to develop a parallel version of a program is often faster than a corresponding message passing version. While the gain in productivity may be appealing, the increase in performance depends on the sophistication of the compiler [KON]. A compiler for shared-memory parallel programs must generate code for thread creation, thread synchronization and access to shared data. In comparison, a compiler for message-passing parallel programs is much simpler. It consists of a base compiler with a communication library. However it is difficult to say definitely whether performance is enhanced using shared-memory constructs or message-passing. But we can admit that the message-passing model offers more scope for optimization [KON]. Data collection and distribution algorithms can be optimized for a particular application. Domain

decomposition can be performed based on communication patterns in the code. Another advantage of message passing is portability. A message-passing program written using a standard communication protocol can be ported from a network of PCs to a massively parallel supercomputer without major changes. Message passing can also be used on a network of heterogeneous machines [KON]. Consequently message passing parallel programming is more widely used than shared memory programming practices. Based on this trend we have focused on building a message debugger which is founded on the concepts of multilevel debugging [PED03] for a message passing programming environment.

Debugging in a Parallel Programming Environment

Debugging parallel programs can be a tedious and frustrating job. Parallelism introduced in the computation brings in a new dimension for errors and unexpected behavior to occur. Programs when tested on individual nodes may run correctly but when put together to run on a network concurrently might give unpredictable results. Also a program execution might not be consistent, sometimes they may run to completion as expected, at other times they might crash and not complete on inputs that had been tested successfully earlier. Such situations are often referred to as non-deterministic problems. Moreover, the issue of debugging is rarely covered in textbooks on parallel programming since the nature of debugging is heuristic. Debugging is strongly influenced by the sophistication (or lack thereof) of the commercial debugger available for the programming language selected and the parallel machine [THI]. It is often left to the programmer as an art to be mastered on one's own rather than a series of scientific rules to be learnt from established sources.

Often debuggers for sequential programming like the GNU Debugger (GDB) [GNU] or the Data Display Debugger (DDD) [DAT] are used to debug the sequential part of a parallel program at a particular node. Another popular technique is to insert print statements in the programs to track what is going on during execution. But the limitation of these processes is stark when a user has to debug a parallel application running on many nodes where both functionality and data has been distributed among various nodes of the network. In such a scenario the user needs to understand the whole picture since the state of the entire application is dependent on the states of all the involved nodes. Focusing on debugging programs at particular nodes with a sequential debugger without understanding the big picture does not go a long way in solving intricate and complex bugs. Moreover nodes need to communicate and coordinate successfully both in synchronous and asynchronous manner based on conformance of a protocol to ensure the completion of an execution. Also, we need to remember that the parallel system may be heterogeneous and the processes will be executing on machines which might have different operating systems, architecture, instruction sets, file systems and physical locations. All these issues make the task of debugging in the parallel scenario extremely difficult and sequential debugging tools are neither designed nor are sufficient to handle this mammoth challenge [TRI05].

Another issue which makes debugging quite hard in the parallel programming environment is the lack of infrastructure. In parallel programs, the cause and effect of an error are often separated by great distance in time and code making it difficult to locate and debug. Also, this difficulty is enhanced when the cause and effect do not lie in a single process [TRI05]. A parallel system is much larger and complex than a single

process and many available tools deluge the programmer with information overload, making it nearly impossible to zero down on useful information pertinent for debugging. This information overload is often caused by the tool trying to give the user a global view of the program [PED03]. To add to these issues we have the problems of detection of deadlocks and checking adherence to a protocol which are not present in sequential code development and debugging.

Multilevel Debugging

In contrast to the top down approach used in most parallel debuggers and visualization tools, multilevel debugging is a bottom up approach to debugging and was developed in [PED03]. Instead of providing a global view of a program and allowing the user to look for any kind of error using just one tool, the bottom up approach of multilevel debugging provides not only tools for creation of error hypothesis but specialized tools for handling each class of error. These tools assist the user to verify a hypothesis and refine it if necessary. They are also equipped with the ability to help the user to track the error back to its source code and fix it.

As discussed earlier, in contrast to sequential debugging there are many new types of errors that arise in parallel programming. To handle these new types of errors, new tools specific to each type which will provide detailed information to locate and debug the error are needed. The bottom up approach of multilevel debugging is very well suited for development of these tools since it not only provides information for hypothesis of an error but also helps in locating the source of the bug . In multilevel debugging errors are classified into three classes' namely sequential, message passing and protocol level errors. Each class has bugs that are specific to it and a developer has to have a good

understanding of the classes of errors to be able to develop the tools for catching them. We shall be focusing on debugging the second class of errors, that is, message passing.

Objectives of this Thesis

Our goal is to develop a simple, yet effective, message debugger based on the principles of multilevel debugging which locates errors due to faulty message passing in parallel programs. At the same time we intend to avoid the flaws present in current message debugging tools. Our message debugger shall be designed to avoid information overloading by providing built-in queries that shall fetch specific data for message communications done by each process. We wish to arm our message debugger with both sequential and parallel debugging abilities through the replay of a parallel application's execution for any number of processes at respective physical nodes. Also, our message debugger shall have the flexibility to integrate and work with a sequential debugger selected by the user. This will help the user to immediately use our message debugger without having to undergo a steep learning curve for another new sequential debugging tool. Along with that we want to provide a feature that will enable a user to write customized queries to get specific and pertinent data necessary to debug a particular type of errors.

The name of this project is IDLI which is an acronym for Interactive message Debugger for parallel message passing programs using LAM-MPI. Henceforth throughout the thesis we shall be referring to our message debugger as IDLI

Organization of this Thesis

A detailed discussion about the current tools for parallel and distributed computing

environment along with their limitations is provided in Chapter 2. IDLI is built to work with message passing library LAM-MPI. Chapter 3 introduces LAM-MPI and explains the functionalities of a set of most commonly used MPI routines for parallel programming. An introduction to IDLI along with meticulous examples explaining each of its features is provided in Chapter 4. We delve into the intricacies of IDLI's implementation details and algorithms in Chapter 5. Chapter 6 provides conclusions and recommendations for future work.

CHAPTER 2

BACKGROUND AND PREVIOUS WORK

For last two decades a great deal of research effort has been directed at development of tools for improving the performance of parallel applications and significant progress has been made [PAN01]. However, the reason for not having highly popular and standardized debuggers in the parallel domain akin to sequential debuggers like GDB [GNU] is that they can be extremely difficult to implement. Tool developers must cope with an inherently unstable environment where it may be impossible to reproduce program events or timing relationships [PAN99]. Moreover it is often difficult to find a comprehensive debugging tool capable of handling of all types of errors that arise in the parallel programming arena [PAN01]. This chapter briefly explains the process of parallel debugging, current tools available for debugging parallel programs using the Message Passing Interface (MPI), and their limitations.

The Need for a Debugger for Parallel Programs using MPI

The inherent nature of parallel programs is complex which makes debugging in the parallel programming environment a difficult and time consuming task. While debugging parallel programs a user not only has to deal with standard sequential errors but also with possible problems arising from communications among processes. This task is made all the more challenging by the lack of parallel infrastructure where tools integrate seamlessly in a heterogeneous massively parallel system.

Although there is a domain of different proposed High Performance Computing (HPC) software programming systems, the de facto standard programming model in use is a combination of multi-threading and message passing using MPI [DES05]. In particular, MPI is a difficult programming model to use due to the fact that it is not a compiled language but a library of routines. Existing tools do not perform static checks on MPI usage beyond correct use of prototypes. Moreover MPI is large and complex and there is lot of room for subtle errors. For example, its standard includes routines for fourteen send calls and five receive calls that can be combined arbitrarily for a total of seventy ways to implement a single point-to-point communication. A user has to clearly understand the functionalities and differences in the various point-to-point communication routines and their respective arguments to make a correct mapping to his needs. Also, distributed processing can obscure the location of errors thus leading to diffuse errors. For example, an error may become apparent many messages after an incorrectly matched message has been exchanged by a process. MPI is one of the most scalable programming models, which makes using a debugger quite hard when the number of processes keeps on increasing. Although debuggers can be made scalable, it is still extremely difficult for a user to debug an error arising in an application using 5000 processes. Changing to a different CPU or network or MPI implementation or changing the problem size can make potential or latent deadlocks and race conditions appear, leading to non deterministic errors [DES05]. In spite of the above potential disadvantages, MPI is widely used for cluster computing because of its portability and reasonably good performance.

In a survey performed in [BAL04] it was found that 75 percent of the users developing large parallel programs on the Grid used `printf` [KER88] statements for debugging! Among debuggers, GDB [GNU] came out on top with a usage by 65 percent of the community, followed by TotalView [ETN] at 45 percent, Compaq Ladebug [LAD] at 25 percent. 55 percent of the developers used a mixture of other debuggers like Visual Threads [HPV], Visual Studio [MIC], Visual MPI, DDD [DAT]. In another study, it was seen that 80 percent of the developers used only 0 to 4 processes while debugging, 35 percent used 5 to 16 processes, 20 percent used 17 to 32 processes, while less than 15 percent users used more than 32 processes for debugging [BAL04].

From the above discussions it is quite evident that there is a need for an efficient yet simple parallel debugger for MPI programs that can significantly reduce the debugging cycle time. Moreover, we also notice that there is a tendency among users to use less number of processes during debugging. The reason behind this behavior might be the inconveniences caused by information overload which increases the time and complexity of debugging an error. Also if a parallel MPI message debugger could integrate popular sequential debuggers like GDB [GNU], it would significantly decrease its learning curve thus enhancing usage and adoption among developer communities.

Current MPI Debuggers and Tools in use by Developers

At present, quite a few tools are available for debugging MPI programs. Most of them can be broadly classified into three categories based on the functionalities they provide. These categories are source level, graphical visualization and post processing debug tools as shown in Figure 1. Some of these tools with their versatile features might belong to more than one of the above mentioned categories.

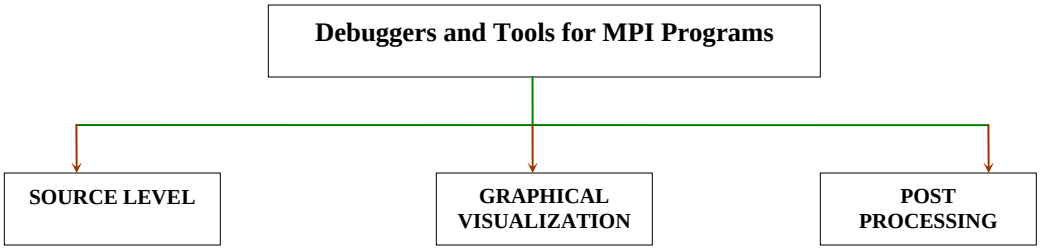


Figure 1: Categories of current debuggers for MPI programs.

Source Level Debuggers

Source level debugging tools being the simplest of the above mentioned three categories are extensions of traditional sequential debuggers like GDB [GNU]. Some tools just instantiate a copy of a standard sequential debugger for each process, while others may be more sophisticated and have a sequential debugger integrated in an Integrated Development Environment (IDE). Nevertheless, they do open multiple windows corresponding to each process being debugged. Since these debuggers are based on the sequential style of programming they do not fit well into the paradigm of parallel debugging which has quite a few new classes of errors.

To begin with, these debuggers typically operate at the level of source and assembly code. As a result they often overwhelm developers with a deluge of information and unnecessary details. Such a fine level of granularity makes it extremely difficult to debug an MPI program running over hundreds of processors. Often source level debuggers do not have views or data pertaining to the big picture containing all the nodes which will help a developer analyze and locate the exact source of bugs. A programmer has to sift through a vast amount of data to locate the actual bug which can often distract her from the real problem.

Each sequential debugging window is capable of providing lots of information about the attached process. The information is localized in context and is best used for debugging the sequential part of the code. Though the technique of debugging a parallel MPI program usually starts at the local context, it eventually requires information pertaining to a global view of the whole application. Consequently there is a need for manageable, yet relevant information, for both global and local domains of each process. However, source level debuggers are devoid of capabilities which provide information in global context, that is, the big picture. This makes debugging a massively parallel program using source level debuggers extremely difficult.

In an MPI program some of the most common type of errors arises due to faulty message passing involving several processes. As the number of processes increase, it becomes nearly impossible for the developer to manually manage, issue commands, monitor the output, and control each process in a separate window. It definitely demands enormous patience from a user to successfully debug a program with such a tool. Further, such a process is quite prone to human errors. Also, running a debug version of the program on hundreds of processes for the sake of debugging a set of errors prevalent only in a few processes is a very expensive and slow process with sub-optimal usage of resources. For example, it is surely not practical to open 256 windows for an MPI program running 256 processes! What is needed here is a set of higher level querying tools that collect information about the inter-process message communication and present relevant information in a user friendly manner.

Graphical Visualization Debuggers

Graphical visualization debug tools attempt in assisting the developer in a top-down debugging approach. They graphically present snapshots of the whole system indicating current states of processes, message queue, message route, pending messages and other relevant system features of the parallel machine. Their primary strength lies in depicting the complete system status at different points of time during execution of the program using various graphical charts and diagrams. Developers get a good idea of the overall system behavior through the means of different views provided by such a debugger.

Unfortunately these debuggers lie on the other end of the spectrum as compared to source level debuggers. Most of these debuggers lack sufficient granularity to aid a developer pin-point exactly what went wrong and where the errors are located. Typically in an MPI program the bulk of the code is sequential. Hence having no source level debugging capability at all seriously cripples the usability of such tools. For example, let us assume that a user has noticed that an MPI_Recv call has received incorrect data. But to figure out the source from where this data was sent, that is, line number of source code of a particular file running on a process of a specific rank, the user is left alone! Thus the burden of mapping an activity at the global level to its causal context at the local level is left entirely to the user. To further complicate the situation often the user cannot debug the sequential part of the code using the visualization debugger.

Graphical visualization debuggers typically have a predefined set of views. In other words they are not flexible or adaptable to a user's customized needs. Often the debugger designers' choice of important and relevant views may not match that of a user. For example, the user is at a loss when she wants to view all MPI communications that

originated from a particular file but this snapshot is not present in the built-in set of views that the visual debugger offers. Further more, these debuggers are not only limited by the imagination of its designers but also by the screen size and resolution of the computers being used.

Post Processing Debuggers

Post processing debug tools provide post-mortem debugging capabilities. The working principle behind such tools is that they log program execution in sufficient detail which enables replaying a part or whole of the program later on. As is apparent, the replaying and analysis is limited to the data snapshots taken by the tool and the events it was interested in. For example, a tool that was designed to record only message passing events would obviously lack any debugging capability for bugs in the sequential part of the code. Most of these debuggers also fail to provide sufficient granularity when needed. Generally they posses no integrated sequential debuggers and perform a replay based on past data stored in the log files. As a result, on-the-fly data manipulation cannot be supported by these tools thereby seriously limiting their debugging features.

Nevertheless, post-processing tools have their own share of merits. In real life, most MPI applications are computation intensive and massive in size, hence they take considerable amount of time to execute. The problems that are solved by models using cluster or grid computing techniques are inherently complex and large. Debugging problems of such a scale by repeated execution of the parallel programs using sequential debuggers or visualization tools not only consumes a lot of time but wastes resources too. On the contrary doing a replay can be tremendously fast since it does not involve the real execution of the parallel program. Replay is further sped up by the elimination of wait

times for resource availability and blocking communications, as well as eradication of the need for the message passing in the MPI communications layer. Figure 2 shows a list of some debuggers used in the parallel programming environment.

NAME	TYPE	WEBSITE
Buster	Post Processing	http://166.111.68.162/web/gelato/gelato-3-Buster.htm
Classic Guard	Source Level	http://www.guardsoft.com/classicguard.html
DDT	Source Level	http://www.allinea.com/?page=48
Etnus TotalView	Source Level	http://www.etnus.com/TotalView/index.html
Intel Trace Collector (Formerly Vampirtrace)	Graphical Visualization	http://www.cyf-kr.edu.pl/supeur96/Krotz/node1.html
Intel Trace Analyzer (Formerly Vampir)	Graphical Visualization	http://www.intel.com/cd/ids/developer/asmona/eng/95656.htm
KDevelop with MPI plugin	Source Level	http://freshmeat.net/projects/mpiplugin/
MQM	Graphical Visualization	http://web.engr.oregonstate.edu/~pancake/ptools/mqm/flyer.html
Panorama	Graphical Visualization	http://www-cse.ucsd.edu/users/berman/panorama.html
Paradyn	Graphical Visualization	http://www.cs.wisc.edu/~paradyn/
PDBX, PEDB and XPDBX	Source level	http://www.arc.unm.edu/~rsahu/pdbx.html http://www-03.ibm.com/systems/p/software/pe.html
PGDGB	Source level	http://www.pgroup.com/products/pgdbg.htm
Prism	Source level, Graphical Visualization	http://docs.sun.com/app/docs/doc/817-0088-10?q=PRISM
PVaniM (for PVM only)	Post Processing	http://www.cc.gatech.edu/gvu/softviz/parviz/pvanimOL/pvanimOL.html
XMPI	Graphical Visualization	http://www.lam-mpi.org/software/xmpi/

Figure 2: A list of some current parallel debuggers.

Thus if we could couple a sequential debugger with a post-processing debug tool which had built-in queries as well as the ability for a user to write customized queries then we can have best of both worlds! That is indeed what IDLI tries to achieve. The ability to write custom queries and the flexibility to work with a sequential debugger of the user's choice will make IDLI a very adaptable (message) debugger. Also, it would have a set of built-in queries for specific data retrieval and the ability to replay program

execution for user specified number of processes. This would enable it to cater to the needs of a wide domain of users without flooding them with unnecessary information.

Arguably a user may try a combination of currently available tools to achieve the above goal. But such a combination is often hindered by the following obstacles: (a) high learning curves for each tool, (b) lack of seamless integration since each product is from a different vendor, (c) different user interfaces and design philosophies for each tool, and (d) variances in compliance to standards, reliability, portability and levels of available support.

Summary

Though there many tools available to aid in cluster programming and high performance computing, only a handful of them are debuggers. Others belong to various tool classes like static or dynamic error checkers, profilers, event tracers and code analyzers. Primarily, the usability and effectiveness of available debuggers are severely reduced due to the following reasons:

- Most of them are designed and built to satisfy only one end of the spectrum, that is, they are either good at providing localized or global contexts but not both. For source level debuggers, which are typically modified sequential debuggers, the user hardly has a clue about the overall system behavior. On the other hand, graphical visualization tools provide considerable information in the global context but offer little help to the users to map it to the local context of individual processes.
- A large amount of data is often not of much relevance to the user and makes debugging extremely challenging. In source level debuggers a user has to

manage a sequential debugger for each process individually. One can imagine the scenario for a debugging session with 1024 processes and as many sequential debugging windows! Almost all graphical visualization tools provide lots of interesting views and data. But most of these views cannot be customized according to a user's needs and hence are insufficient by themselves to debug a massive MPI program. The post-processing tools usually take snapshots at certain predefined intervals or record a predefined set of events. But they lack the flexibility to adapt to the user's need for recording customized events or data. Once the user manages to localize the error, the next step is often a need to actually run and debug only a handful of processes. Unfortunately most of the post-processing tools do not have this capability.

- A serious shortcoming of existing tools is that they offer no flexibility to create customized views or alter existing views. This feature would have helped users locate relevant information from the vast amount of available data. It would be of immense help if the user could interactively query the data, a process which is fast enough in real time and obtain exactly what he is interested in.
- In order to debug MPI programs effectively, users often need to trace a particular message or a group of related messages. For example, a user might be interested in viewing all related message exchanges that participated in a group communication, like a call to MPI_Allgather at a particular location of a file. Most existing debuggers do not support such queries. Few tools do offer

tracing of messages but they require instrumentation of users' applications. Such instrumentations may be unnecessary if a tool is designed to provide similar functionality without modifying the source code of a user's application programs.

Our message debugger, IDLI, is a sincere attempt to precisely address the above mentioned issues. It has been designed to provide information at the global level by the means of several built-in queries. Yet at the same time it enables a user to do source level debugging within local context of a process using a sequential debugger of her choice. A Query Manager in IDLI offers a number of predefined built-in queries as well as gives a user the power to write customized queries to retrieve specific information. This feature ensures that IDLI does not inundate a user with huge amounts of irrelevant data. Last but not the least is IDLI's Replay feature which allows a user to replay a program's execution for a selected number of processes simultaneously. This allows a user to replay as many processes as he is comfortable with, thus helping focus attention on processes which might be probable sources of errors. Also, it is to be noted that users do not need to modify their source code in order to use IDLI. In short, IDLI is a message debugging tool which attempts to fill in the shortcomings present in current tools of its genre in a simple yet efficient manner.

CHAPTER 3

THE MPI PARALLEL PROGRAMMING PARADIGM

Message passing is a widely used programming paradigm on parallel computers, especially so for a Network of Workstations (NOW) or Scaleable Parallel Computers (SPC) with distributed memory. MPI stands for Message Passing Interface and is a standard that defines the user interface and functionality for a wide range of message-passing capabilities [SNI96]. One of MPI's major goals is to achieve a good degree of portability across different architectures. Since IDLI is designed to work with MPI its wise to delve into a brief overview of MPI, and have a quick glimpse of a set of its most widely used functions.

MPI

During the period from 1980s to early 1990s, as parallel computing progressed, a number of incompatible tools for writing distributed programs were developed usually with tradeoffs between portability, performance, functionality and price. Soon recognition for the need of a standard arose. MPI resulted from the efforts of numerous individuals and groups over the course of a couple of years between 1992 and 1994. MPI is the only message passing library which can be considered a standard [MES]. It is supported on virtually all high performance computing (HPC) platforms. Practically, it has replaced all previous message passing libraries. MPI is highly portable since there is no need to modify the source code of an application when it is ported to different

platforms that support the MPI standard. A variety of implementations, both in the vendor and public domains make MPI widely available.

The programming model of MPI lends itself to most, if not all distributed memory parallel programming paradigms. In addition, MPI is commonly used behind the scenes to implement some shared memory models, such as Data Parallel, on distributed memory architectures [MES]. Originally, MPI was targeted for distributed memory systems. As shared memory systems became more popular, particularly symmetric multi processors (SMP) and Non Uniform Memory Access (NUMA) architectures, MPI implementations for these platforms appeared. MPI is now used on just about any common parallel architecture including massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks [MES]. MPI makes all parallelism explicit, that is, programmers are responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time.

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require the specification of a communicator as an argument. MPI_COMM_WORLD is the predefined communicator that includes all MPI processes. Within a communicator, every process has its unique integer identifier known as rank. Ranks are assigned by the system when the processes initialize. A rank is sometimes also called a “process ID”, it begins at zero and is contiguous. Programmers specify the source and destination of messages using ranks of processes. Often ranks are used conditionally by the application to control

program execution [MES]. The general structure of an MPI program along with a communicator is shown in Figure 3.

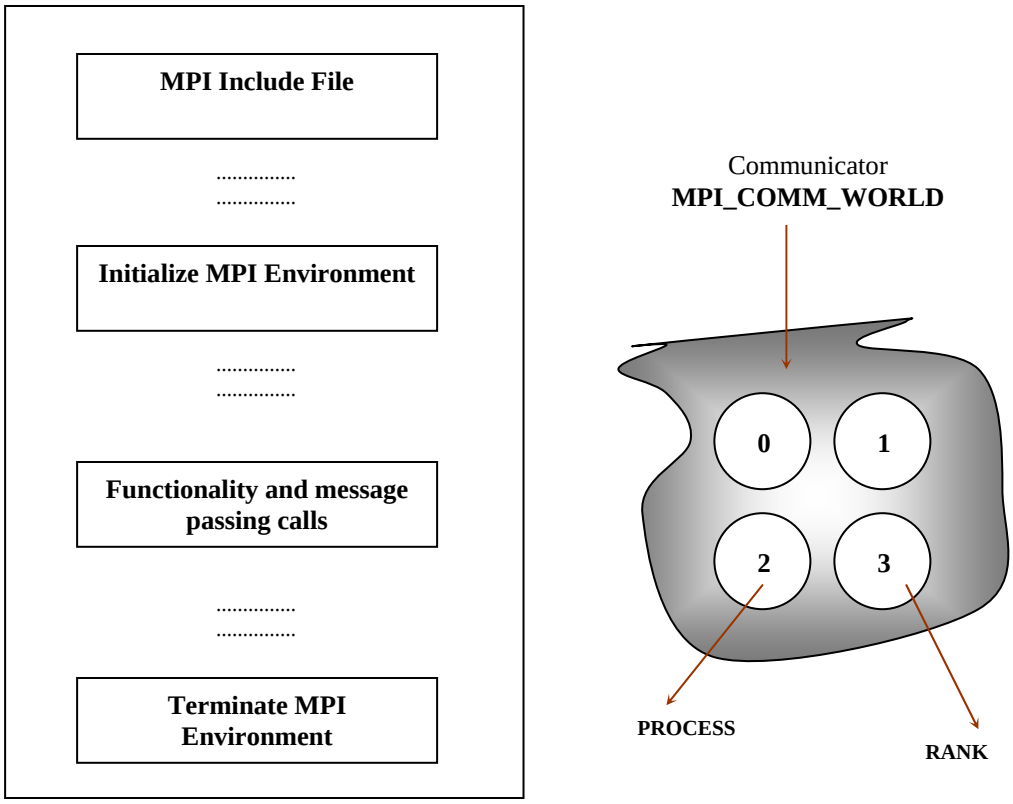


Figure 3: General structure of an MPI program and a communicator.

LAM-MPI

LAM-MPI is an excellent implementation of the MPI standard by the Open Systems Laboratory (OSL) at Indiana University. LAM-MPI provides high performance on a variety of platforms, from small off-the-shelf single CPU clusters to large symmetric multiprocessor (SMP) machines with high speed networks, even in heterogeneous environments. In addition to high performance, LAM provides a number of usability features key to developing large scale MPI applications. Some of these features are abilities to checkpoint or restart, fast job startup, high performance communication,

interoperable MPI in heterogeneous environment, run time tuning and remote process invocation (RPI) selection, SMP aware collectives, and extensible component architecture [LAM].

MPI Functions

In the message-passing library approach to parallel programming, a collection of processes execute programs written in a standard sequential language augmented with calls to a library of functions for sending and receiving messages [FOS95]. MPI is a complex system. In its entirety, it comprises of more than two hundred functions. As our goal is to convey the essential concepts of message-passing programming, and not to provide a comprehensive MPI reference manual, we focus on a set of twenty four functions which provide more than adequate support for a wide range of applications [WIL05]. To further simplify the study we have classified these functions into three groups of preliminary, point-to-point message passing and group communication routines as shown in Figure 4. Before we explain the details of each group of the twenty four commonly used MPI functions we shall briefly discuss blocking, non-blocking, synchronous and asynchronous communication using message passing.

Blocking and Non-blocking Message Passing

Routines that return after completing their local actions, even though the message transfer may not have been completed are blocking. Those that return immediately are non-blocking. In MPI, non-blocking routines assume that the data storage to be used for the transfer is not modified by the subsequent program statements prior to the data storage being actually used for the transfer [WIL05].

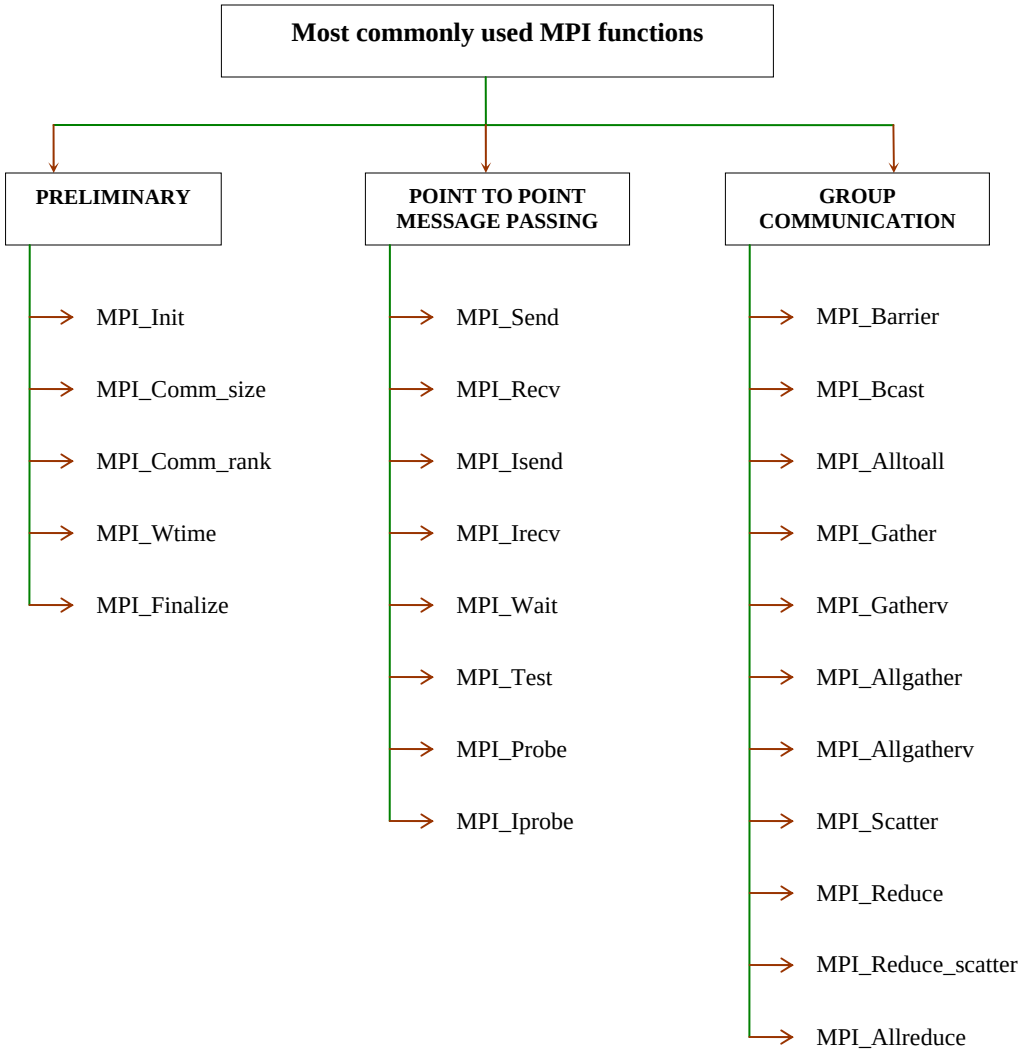


Figure 4: Classification and names of widely used MPI routines.

Synchronous and Asynchronous Message Passing

The term synchronous is used for routines that return when the message transfer has been completed [WIL05]. A synchronous send routine will wait until the complete message has been received by the receiving process before returning. A synchronous receive routine will wait until the message it is expecting arrives and has been stored before returning. A pair of processes, one with a synchronous send operation and one with a matching synchronous receive operation will be synchronized, with neither the

source process nor the destination process able to proceed until the message has been passed from the source process to the destination process [WIL05]. Asynchronous communications initiate the send or receive but do not block on their completion.

Preliminary Routines

The preliminary routines are needed for establishing the environment and related matters [WIL05]. They consist of five functions namely MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Wtime and MPI_Finalize. The signatures of each preliminary function are shown in Figure 5.

FUNCTION NAME	SIGNATURE
MPI_Init	`int MPI_Init(int *argc, char **argv)`
MPI_Comm_size	`int MPI_Comm_size(MPI_Comm comm, int *size)`
MPI_Comm_rank	`int MPI_Comm_rank(MPI_Comm comm, int *rank)`
MPI_Wtime	`double MPI_Wtime (void)`
MPI_Finalize	`int MPI_Finalize (void)`

Figure 5: Signatures of preliminary routines of MPI.

MPI_Init initializes MPI execution environment. This function must be called only once, before any other MPI functions in every MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

MPI_Comm_size determines the number of processes in the group associated with a communicator. It is generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by an application [MES].

MPI_Comm_rank determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors minus 1 within the communicator MPI_COMM_WORLD.

This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

MPI_Wtime returns an elapsed wall clock time in seconds (double precision) for the calling processor. This is the only MPI function that does not return an error code like all other MPI routines, instead it returns times in seconds. Figure 68 in Appendix 1 is an example of an MPI program that shows the use of the function MPI_Wtime.

MPI_Finalize terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it [MES]. Figure 56 in Appendix 1 shows the use of the functions MPI_Init, MPI_Comm_size, MPI_Comm_rank and MPI_Finalize in an MPI program.

Point-to-Point Message Passing Routines

A widely used set of point-to-point message passing routines comprises of MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, MPI_Wait, MPI_Test, MPI_Probe, and MPI_Iprobe. All these routines return an integer error message code. The signatures of each of the above functions are shown in Figure 6.

MPI_Send is a basic blocking send operation and returns only after the application buffer in the sending task is free for reuse. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send to implement the basic blocking send. MPI_Recv receives a message and blocks until the requested data is available in the application buffer in the receiving task. Figure 56 in Appendix 1 lists an MPI program which shows the use of the functions MPI_Send and MPI_Recv.

FUNCTION NAME	SIGNATURE
MPI_Send	`int MPI_Send (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
MPI_Recv	`int MPI_Recv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
MPI_Isend	`int MPI_Isend (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
MPI_Irecv	`int MPI_Irecv (void* buf, int count, MPI_Datatype datatype, int source,int tag, MPI_Comm comm, MPI_Request *request)`
MPI_Wait	`int MPI_Wait (MPI_Request *request, MPI_Status *status)`
MPI_Test	`int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)`
MPI_Probe	`int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status)`
MPI_Iprobe	`int MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)`

Figure 6: Signatures of point-to-point message passing routines of MPI.

MPI_Isend identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed [MES]. MPI_Irecv identifies an area in memory to serve as a receive buffer. Akin to MPI_Isend the program's execution continues immediately after this call is made. The pending message status is handled using the communication request handle. The program must use calls to MPI_Wait or MPI_Test to determine the completion of the non-blocking receive operation. On completion of the call, the requested message is available in the application buffer. Figure 66 in Appendix 1 lists an MPI program which shows the use of the functions MPI_Isend and MPI_Irecv.

MPI_Wait makes the program wait for a non-blocking operation to complete. MPI_Wait returns after the operation identified by request completes. Information on the

completed operation is found in the MPI_Status data structure. MPI_Test is used to determine if a non-blocking request has completed or not. MPI_Test returns `flag` equal to `true` (the signature of MPI_Test is in Figure 6) if the operation identified by `request` is complete. The MPI_Status object is set to contain information on the completed operation. MPI_Probe waits until a message matching `source`, `tag`, and `comm` (the signature of MPI_Probe is in Figure 6) arrives. It lets one check for an incoming message without actually receiving it. MPI_Probe is a blocking call that returns only after a matching message has been found. MPI_Iprobe is akin to MPI_Probe, but it returns `flag` equal to `true` when there is a message that matches the pattern specified by the arguments `source`, `tag`, and `comm` (the signature of MPI_Iprobe is in Figure 6) can be received [MES].

Group Communication Routines

Some of the most commonly used group communication routines are MPI_Barrier, MPI_Bcast, MPI_Alltoall, MPI_Gather, MPI_Gatherv, MPI_Allgather, MPI_Allgatherv, MPI_Scatter, MPI_Reduce, MPI_Reduce_scatter, and MPI_Allreduce. All these routines return an integer error message code. The signatures of each of the above group communication routines are shown in Figure 7.

MPI_Barrier subroutine blocks until all tasks have called it. Tasks cannot exit the operation until all group members have entered it. Figure 62 in Appendix 1 lists an MPI program which shows the use of the function MPI_Barrier.

MPI_Bcast broadcasts a message from the calling process, to all tasks in `comm`. The contents of the calling process's communication buffer are copied to all tasks on return. In

FUNCTION NAME	SIGNATURE
MPI_Barrier	`int MPI_Barrier (MPI_Comm comm)`
MPI_Bcast	`int MPI_Bcast (void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
MPI_Alltoall	`int MPI_Alltoall (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf,int recvcount,MPI_Datatype recvtype, MPI_Comm comm)`
MPI_Gather	`int MPI_Gather (void* sendbuf,int sendcount, MPI_Datatype sendtype, void* recvbuf,int recvcount,MPI_Datatype recvtype,int root, MPI_Comm comm)`
MPI_Gatherv	`int MPI_Gatherv (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)`
MPI_Allgather	`int MPI_Allgather (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
MPI_Allgatherv	`int MPI_Allgatherv (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)`
MPI_Scatter	`int MPI_Scatter (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
MPI_Reduce	`int MPI_Reduce (void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
MPI_Reduce_scatter	`int MPI_Reduce_scatter (void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
MPI_Allreduce	`int MPI_Allreduce (void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

Figure 7: Signatures of group communication routines of MPI.

Figure 63 in Appendix 1 lists an MPI program which shows the use of the function MPI_Bcast.

MPI_Alltoall sends a distinct message from each task to every task. Figure 61 in Appendix 1 lists an MPI program which shows the use of the function MPI_ Alltoall.

MPI_Gather collects individual messages from each task in `comm` at the root task and stores them in rank order. MPI_Gatherv operates in a similar way as MPI_Gather but with `recvcounts` as an array (the signature of MPI_Gatherv is in Figure 7), messages

can have varying sizes, and `displs` allows one the flexibility of where the data is placed on the calling process. Figure 64 and Figure 65 in Appendix 1 list MPI programs which show the uses of the functions MPI_ Gather and MPI_Gatherv respectively.

MPI_Allgather gathers individual messages from each task in the communicator and distributes the resulting message to each task. MPI_Allgatherv is akin to MPI_Allgather but here messages can have different sizes and displacements. Figure 58 and Figure 59 in Appendix 1 lists MPI programs which show the uses of the functions MPI_ Allgather and MPI_Allgatherv respectively.

MPI_Scatter distributes individual messages from the calling process to each task in the communicator. This subroutine is the inverse operation to MPI_Gather. MPI_Reduce applies a reduction operation to the vector `sendbuf` over the set of tasks specified by the communicator and places the result in `recvbuf` on the calling process (the signature of MPI_Reduce is in Figure 7). Users can define their own operations or use the predefined MPI operations. Figure 57 in Appendix 1 lists an MPI program which shows the uses of the functions MPI_ Scatter and MPI_Reduce.

MPI_Reduce_scatter applies a reduction operation to the vector `sendbuf` over the set of tasks specified by the communicator and scatters the result according to the values in `recvcounts` (the signature of MPI_Reduce_scatter is in Figure 7). Figure 67 in Appendix 1 lists an MPI program which shows the uses of the function MPI_ Reduce_scatter. MPI_Allreduce applies a reduction operation to the vector `sendbuf` over the set of tasks specified by the communicator and places the result in `recvbuf` (the signature of MPI_Allreduce is in Figure 7) on all of the tasks [MES]. Figure 60 lists an MPI program which shows the uses of the function MPI_ Allreduce.

This concludes our brief sketch of the set of the twenty four most widely used MPI functions, which are supported in our message debugger, IDLI.

Summary

In this chapter we gave a brief introduction to MPI and explored a set of twenty four most commonly used MPI functions for parallel programming. IDLI has been designed to support these twenty four MPI functions.

CHAPTER 4

INTRODUCING IDLI – AN MPI MESSAGE DEBUGGER

The concept of multi level debugging was developed and demonstrated on PVM by a debugger named Millipede [PED03] written in the programming language for the PVM message passing system. Later, a Java GUI was added to Millipede to improve its cross platform compatibility and incorporate new debugging features [TRI05]. IDLI is a message debugger designed to extend the concepts of multilevel debugging to LAM MPI which has become a popular message passing library for parallel computations. In the subsequent sections we shall explore the architecture and features of IDLI in detail.

Architecture and Overview

Our multilevel message debugger, IDLI, operates with a high quality implementation of the Message Passing Interface (MPI) standard from LAM which is associated with Open Systems Laboratory (OSL) of Indiana University [LAM]. The architecture of IDLI consists of three layers:

- a distributed relational SQL database which is used for persistent data storage comprises the backend,
- the middle layer is a native C [KER88] library which has wrappers for MPI functions and
- a simple shell user interface forms the front end. These three layers aid in logging, displaying and analyzing the debugging information gathered from calls

made to the MPI routines during the execution of a parallel application in a distributed environment.

Wrappers for MPI functions in the native C library of IDLI

During a debugging session, when a user's application is compiled in debug mode with IDLI, it is linked with the native C [KER88] library of IDLI instead of the standard MPI library. This is done by means of the wrapper functions present in the native C library. As explained in the previous chapter, IDLI supports twenty four commonly used MPI functions [WIL05]. Consequently the native C library has wrapper functions for each of these MPI routines. These wrapper functions intercept the MPI calls placed in the user's program. A detailed explanation of the manner in which the wrapper functions intercept the MPI calls is provided in the next chapter. In addition, the wrappers possess intelligence for MPI function specific (a) initialization, (b) database processing, (c) lock management for database transactions like writes and updates (more than one process might try to simultaneously write or update the same database tables) and (d) determining the mode for processing, that is, debug or replay mode. When a wrapper function intercepts an MPI call from an application program, it stores debugging information specific to the MPI routine in the SQL database at the backend. The stored data is furnished by the wrappers whenever required by the user for analysis, and debugging of erroneous scenarios. As a result, IDLI's native C library has a two way communication with the SQL database as shown in Figure 8.

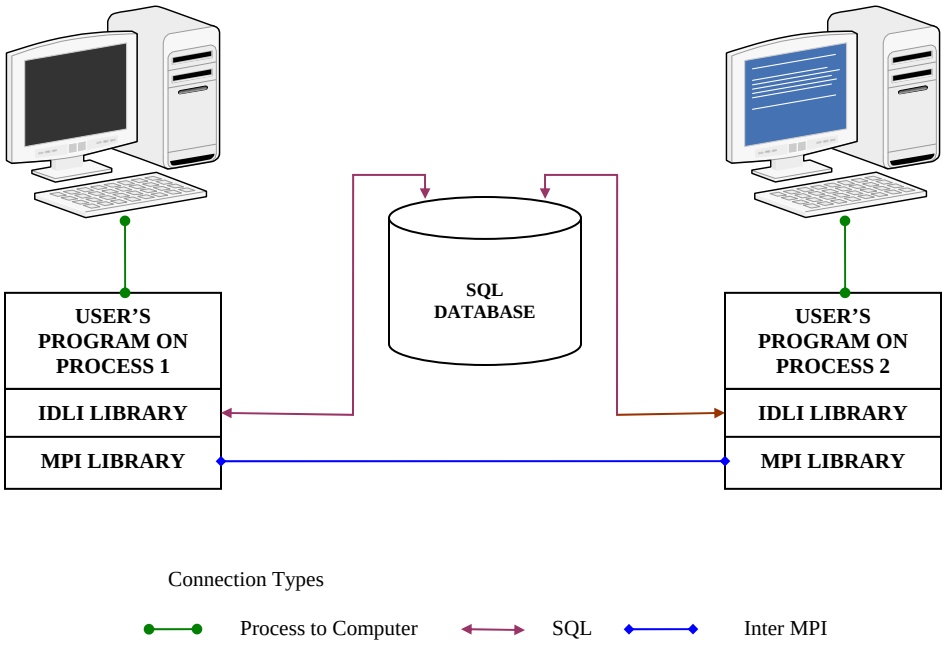


Figure 8: Overview of execution of an application in debug mode with IDLI.

The PostgreSQL Database

We have used the free open source SQL database system PostgreSQL [POS] for persistent storage [TRI05]. PostgreSQL is a well tested, widely used powerful and distributed object-relational database management system. It has an efficient and safe concurrency transaction management. The SQL database need not be installed on any of the nodes of the network where the user's application is executing. This is demonstrated in Figure 8. The architecture of IDLI enables the processes to insert data into the database in a distributed manner. Each process executing the program with MPI calls stores its data by opening a dedicated connection to the database server through a TCP/IP network connection.

Front-end User Interface of IDLI and Query Manager

IDLI has a simple shell user interface which acts as the front end. We will provide a brief overview of the functionality and architecture of the main features of IDLI that can be accessed from its front end. Later on, detailed explanations of each feature will be provided in subsequent sections of this chapter. IDLI, as a message debugger, can be used to view details of messages exchanged by MPI calls through a Query Manager. The Query Manager has a front-end, which is the shell user interface that interacts with a SQL database at the backend. A set of well defined built-in queries are provided by the Query Manager to aid the user in retrieving debugging data for analysis and hypothesis formation of errors [PED03]. It is also equipped with a feature that enables a user to write customized SQL queries which fetch specific data directly from database tables. A user can do post mortem analysis using IDLI's Query Manager.

Replay

An application's execution can be replayed simultaneously at a number of selected processes using a sequential debugger of the user's choice with IDLI's Replay feature. The functionality of Replay will be discussed later in an exclusive section. During replay all data related to MPI calls are fetched from stored data of a previous run of the application. Since the MPI communication layer is not invoked during Replay, the debugging process for a parallel application running on large networks is considerably speeded. A user may run a Replay of the application in a sequential debugger of her choice for a selected set of processes from any machine on the network which is enabled with TCP/IP connectivity. IDLI forks individual connections for each specified process. During replay of a program, IDLI forks a connection to the chosen process running on a

node using the Secure Shell 2 Protocol (SSH2) over TCP/IP and invokes the user's selected sequential debugger. The original session of the Query Manager, from which the replay was initiated, is the parent while the replays on different nodes are its children. The Replay feature can be used simultaneously for any number of processes of distinct ranks. IDLI automatically runs the Replay on the same node on which the process was originally run during an application's execution. We do so to simulate the exact hardware and software environment for the parallel application in which the errors arose. This is particularly relevant for processes running in a hetereogenous network where each node might have a different machine architecture and operating system.

It is noteworthy that multiple processes might have run on the same node. IDLI is capable of handling multiple such Replay sessions on the same node. Further it has built-in error checking to prevent a user from simultaneously invoking more than one Replay for the same process rank. IDLI has a robust Replay session management functionality. During exit from Replay, IDLI checks if there are any replays in action on any of the nodes. If that is the case, the user is notified to exit only when all nodes have quit replays. This is done to maintain consistency in replays' session management data stored in the user database. Figure 9 shows the overall architecture of IDLI's Query Manager and Replay at a couple of nodes.

Features of IDLI

After a user completes locating and correcting sequential errors present in a program, she proceeds to the next level in multi level debugging which is interactive message debugging [PED03]. This level involves communication between two or more processes. It is concerned with correcting errors arising from faulty messaging and incorrect

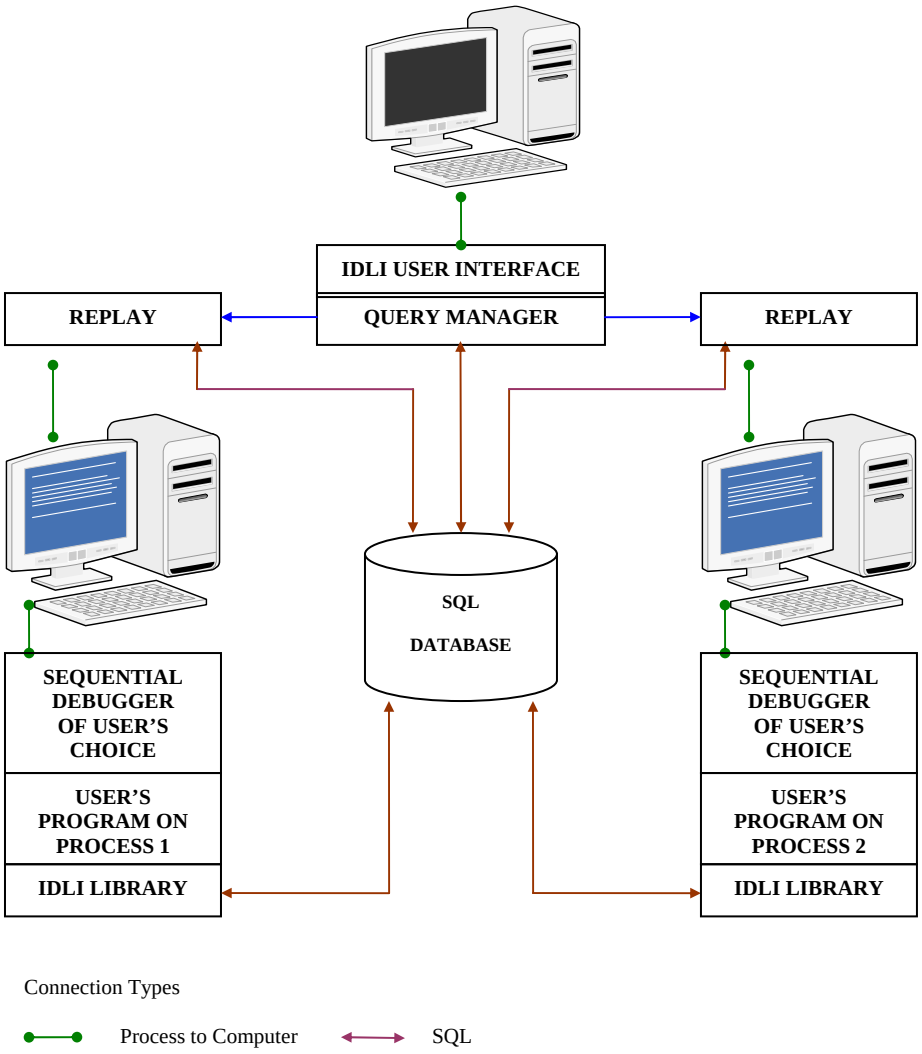


Figure 9: Overall architecture of IDLI's Query Manager and Replay.

message content. Our message debugger, IDLI, can be used to analyze, as well as view, the contents of communication messages exchanged by MPI routines in an application. It can also be used for debugging source code of a program. Thus IDLI is a tool that enables a user to do *post-processing* as well as *source level* debugging. We shall hereby explore each feature of IDLI, and demonstrate its usage by means of real time examples.

When a user begins a session with IDLI he gets a welcome screen with a list of menus. The complete menu navigation map of IDLI is shown in Figure 10. The front-end, which is a shell user-interface, possesses useful features like command history, command completion with tabs, prompts showing the selected user database for current session and other niceties. Data is predominantly displayed as a set of rows. Each row has columns for different types of data. Each column has a header row which has a suitable name for that column. IDLI's user interface uses a dark colored background for the header row and the text headings are highlighted in a white color. To easily discern the criteria of sort, columns of sorted data are generally displayed in colors which are different from the rest of the data.

The Commands: List and Drop

While debugging a stand alone application a developer might need to refer to data from the application's previous executions. This helps him to analyze how current changes incorporated to debug an error have affected the application with respect to its previous run. Also, most often applications are inter-linked with many executables, each of which have to be developed and unit tested.

In such a scenario, when a user is debugging an application, she may need to refer to a previous debugging session of another related application. If the previous session's debugging information is not stored or is inaccessible (some debuggers store such data in hidden files in binary formats), then the user is forced to rerun the previous application to refer to required data.

To avoid such scenarios and inefficient use of resources, IDLI allows a user to access and analyze the most recently executed programs, up to a maximum of five sessions.

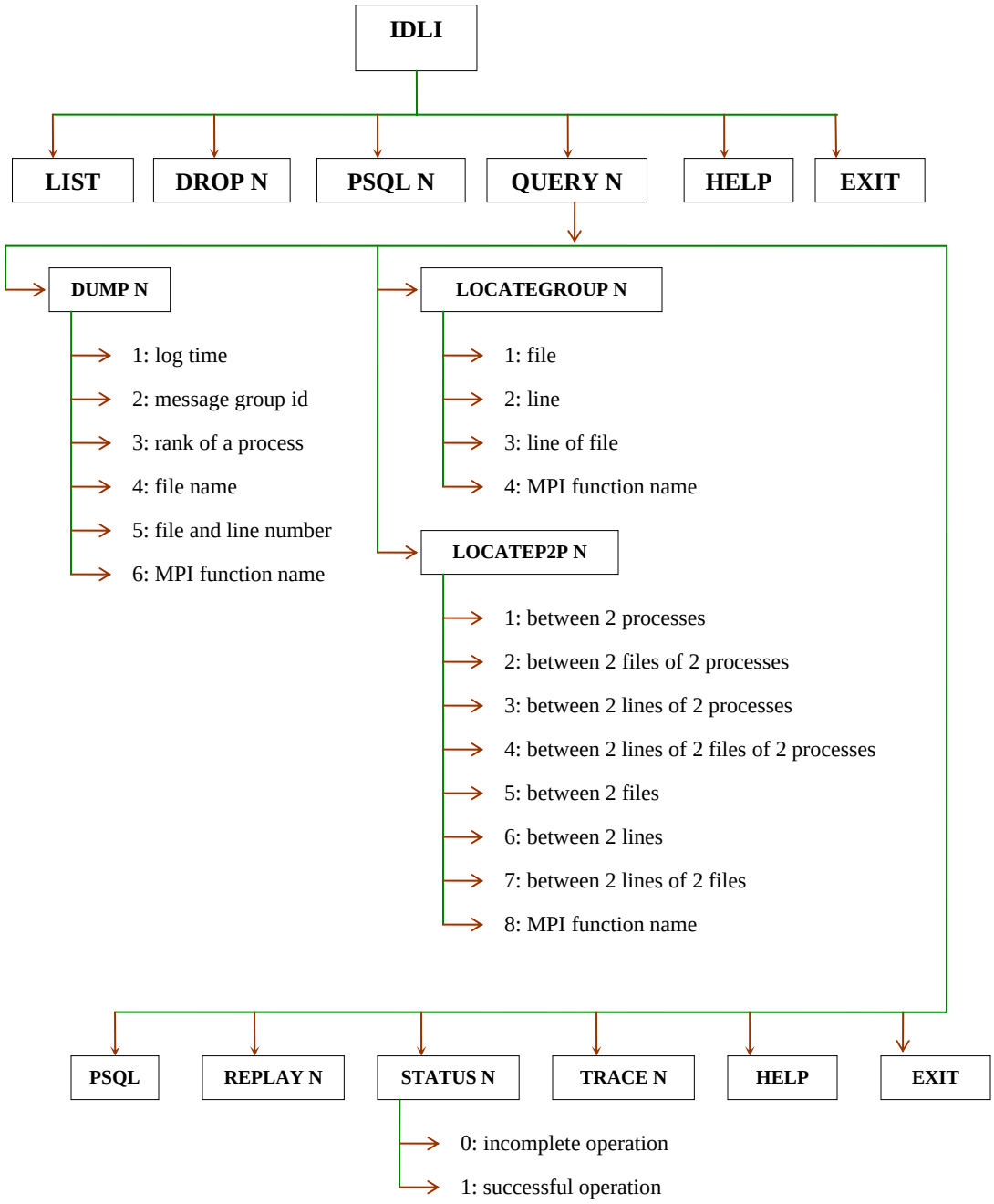


Figure 10: Menu navigation map of IDLI.

Once the maximum of five sessions is reached, the user database pertaining to the earliest session is deleted following the first in first out (FIFO) algorithm. The corresponding session number is reassigned to the current session and a new user database is created for storage of debugging data for present execution. To optimize space usage on the database server we have chosen this policy of recycling sessions by deleting previous user databases when they exceed the maximum allowed. User databases are named by appending the username with the session number.

When a user types the command *list* he is shown a list containing rows with details of databases which have debugging data from previous executions of his applications. The header row for the above list consists of headings like *sessiono, userdb, execname* and *sessiontime*. A user's session numbers are specified by the heading *sessiono*, names of user databases from previous executions of applications are headed by *userdb*, name of the executable as *execname* and *sessiontime* denotes session creation times. Figure 11 demonstrates the execution of the command *list*.

Deletion of a user database from a particular session can be accomplished using the command *drop N*. The session number of the user database to be deleted is specified by *N*. After a user deletes a database by entering the command *drop N*, she may type *list* to check if the database has been really deleted. Command execution for *drop* is shown in Figure 11.

Considerable error checking is done for the interface commands in IDLI. Instead of giving one generic message for all types of errors, meaningful messages are provided explaining the mistakes. This helps a user to understand the exact nature of his error.

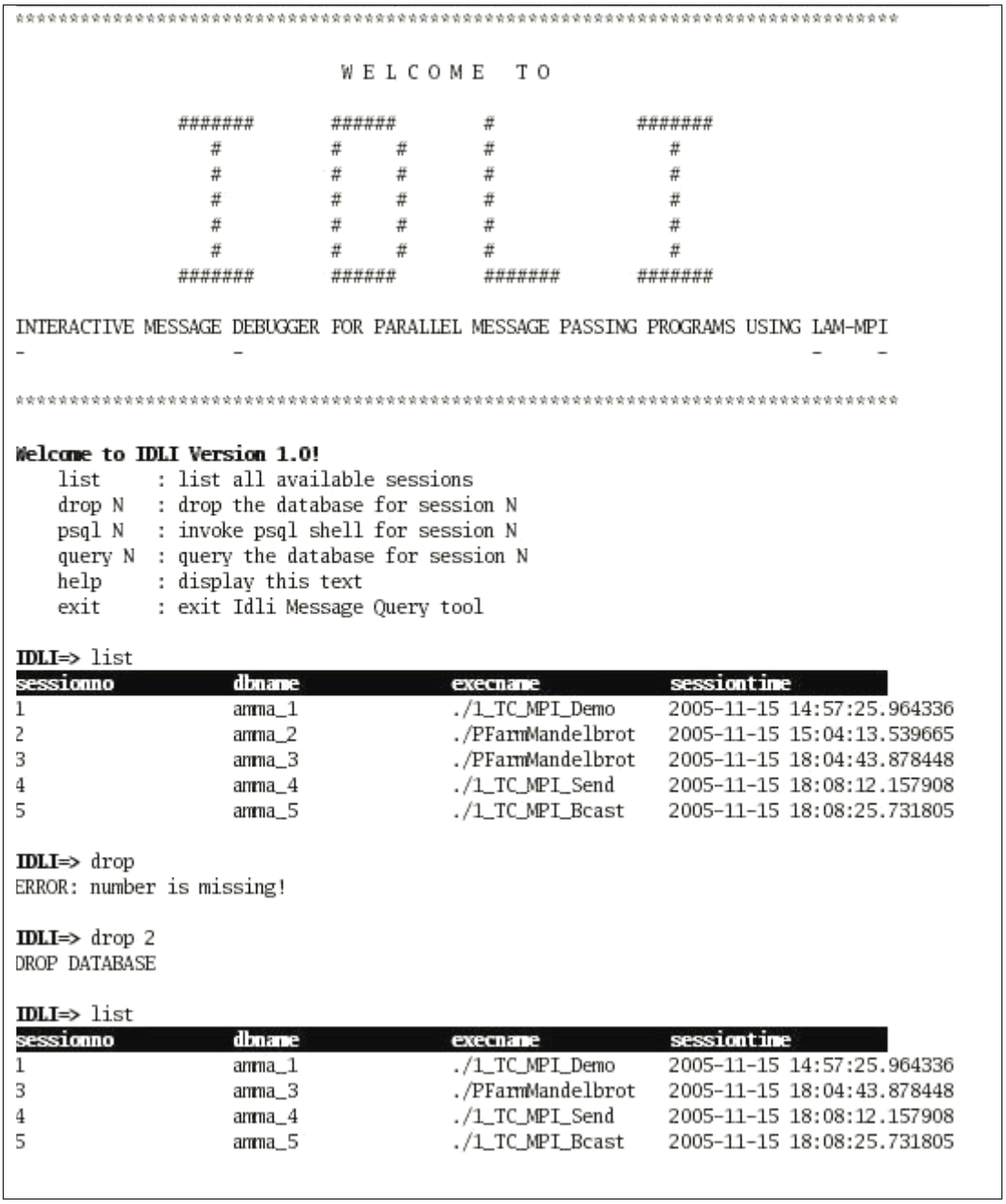


Figure 11: Commands list and drop being executed with error checks.

For example, if a user types in the command *drop*, the error message generated is "ERROR: number is missing!" Since zero cannot be a valid session number, if a developer enters *drop 0*, the error message is "ERROR: number is invalid!" Similarly, throughout the implementation of IDLI we have adhered to the goal of generating useful

customized error messages instead of generic ones. Figure 11 demonstrates some error checks.

Query Manager

IDLI's Query Manager can be used for postmortem analysis of communication messages. After an application has been run in debug mode, a developer can study messages exchanged by calls to MPI functions by using a suitable subset of commands from the set of built-in commands present in the Query Manager. The built-in features have been designed in a way as not to overwhelm the user with huge amounts of irrelevant data. A user can also get customizable views of data from the global to local context of the application by invoking a *psql* shell and writing custom SQL queries. Customized SQL queries are explained in a later section.

The Query Manager comes with a set of useful built-in commands for viewing the details of the executions of MPI routines and messages exchanged by them. The command *query N* invokes the Query Manager with its built-in queries for the specific database used in session number *N*. For example, if a developer wants to query the database used to store the debugging information of an application run in session three, the command for doing so is *query 3*. This command would bring up the Query Manager's menu of built-in query commands. The different column names in the common header row for the list of data displayed by all the commands in the Query Manager is explained in Figure 12.

The set of queries provided is *dump, locateGroup, locatep2p, status* and *trace.* Each of these queries has a lot of options to enable a user to view specific data according to her needs. The commands *help* and *exit* are self explanatory. A user may invoke replay of an

msgId	: Message Group Id assigned by IDLI
src	: Rank of source for an MPI function call
dest	: Rank of destination for an MPI function call
myRank	: Rank of process placing MPI function call
mpiFuncName	: Name of MPI function called
ok	: Denotes whether a group communication was successful (1) or not (0)
fileName	: Name of file from which MPI function was called
line	: Line Number of the source code at which MPI function was called
logTime	: Time (logged) at which MPI function was called
returnMsg	: Message denoting result of an MPI function call

Figure 12: Legend for common header row of data displayed by all query commands.

application's execution by using the command *replay*. In subsequent sections we shall explain each feature of the Query Manager in sufficient details along with examples of their use. The Query Manager's menu of built-in queries is shown in Figure 13.

```
*******************************************************************************

  ##      #    #   #######   ######    #     #      #     #     #####   ######
 #  #     #    #   #         #     #    #   #       ##    ##    #       #     #
#    #    #    #   #         #     #     # #        # #  # #   #       #     #
#    #    #    #   #####     ######       #         #   #  #   #  ####   ######
# ## #    #    #   #         #     #      #         #      #   #     #   #     #
 #  #     #    #   #         #     #      #         #      #    #    #   #     #
  ## #     #####   #######   #     #      #         #      #     #####   #     #
     #

                              IDLI QUERY MANAGER
*******************************************************************************
    dump N        : dump messages sorted by one of the following ways, N is the number of choice
                    1 : log time
                    2 : message group id
                    3 : rank of a process
                    4 : file name
                    5 : file and line number
                    6 : MPI function name
    locategroup N : locate group communication messages by one of the following ways, N is the number of choice
                    1 : file
                    2 : line
                    3 : line of file
                    4 : MPI function name
    locatep2p N   : locate point-to-point messages by one of the following ways, N is the number of choice
                    1 : between 2 processes
                    2 : between 2 files of 2 processes
                    3 : between 2 lines of 2 processes
                    4 : between 2 lines of 2 files of 2 processes
                    5 : between 2 files
                    6 : between 2 lines
                    7 : between 2 lines of 2 files
                    8 : MPI function name
    psql          : invoke psql shell for current database
    replay N      : replay process execution of rank N in gdb
    status N      : display messages sorted by status of MPI call for entire operation
                    0 : incomplete operations
                    1 : successful operations
    trace N       : trace all messages of a particular message group, N is the message group id
    help          : display this text
    exit          : exit query mode for this session
```

Figure 13: Menu of the built-in queries of IDLI's Query Manager.

Built-in Query: Dump

The query command *dump* is used to list details of all MPI functions executed during an application's run. If a user wants to see the exact chronological order in which the MPI routines were executed, she may use the command *dump 1*. This command lists the execution of MPI routines sorted by the column *logTime*. Viewing chronological order of calls helps locate any function calls that might not have been executed at all. Figure 14 shows the execution of *dump1*.

```
amma_1=> dump 1
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
    1    0    0      0 MPI_Init             1 1_TC_MPI_Send.c     9 2005-10-26 14:25:01.707574 MPI_SUCCESS: no errors
    2    1    1      1 MPI_Init             1 1_TC_MPI_Send.c     9 2005-10-26 14:25:01.839972 MPI_SUCCESS: no errors
    3    0    0      0 MPI_Comm_size        1 1_TC_MPI_Send.c    10 2005-10-26 14:25:01.875077 MPI_SUCCESS: no errors
    4    1    1      1 MPI_Comm_size        1 1_TC_MPI_Send.c    10 2005-10-26 14:25:01.882099 MPI_SUCCESS: no errors
    5    0    0      0 MPI_Comm_rank        1 1_TC_MPI_Send.c    11 2005-10-26 14:25:01.888407 MPI_SUCCESS: no errors
    6    1    1      1 MPI_Comm_rank        1 1_TC_MPI_Send.c    11 2005-10-26 14:25:01.9348   MPI_SUCCESS: no errors
    7    0    1      0 MPI_Send             1 1_TC_MPI_Send.c    17 2005-10-26 14:25:01.968825 MPI_SUCCESS: no errors
    8    0    0      0 MPI_Finalize         1 1_TC_MPI_Send.c    24 2005-10-26 14:25:02.001325 MPI_SUCCESS: no errors
    7    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c    21 2005-10-26 14:25:02.017125 MPI_SUCCESS: no errors
   10    1    1      1 MPI_Finalize         1 1_TC_MPI_Send.c    24 2005-10-26 14:25:02.039665 MPI_SUCCESS: no errors

amma_1=>
```

Figure 14: Chronological list of executions of all MPI routines by the command *dump1*.

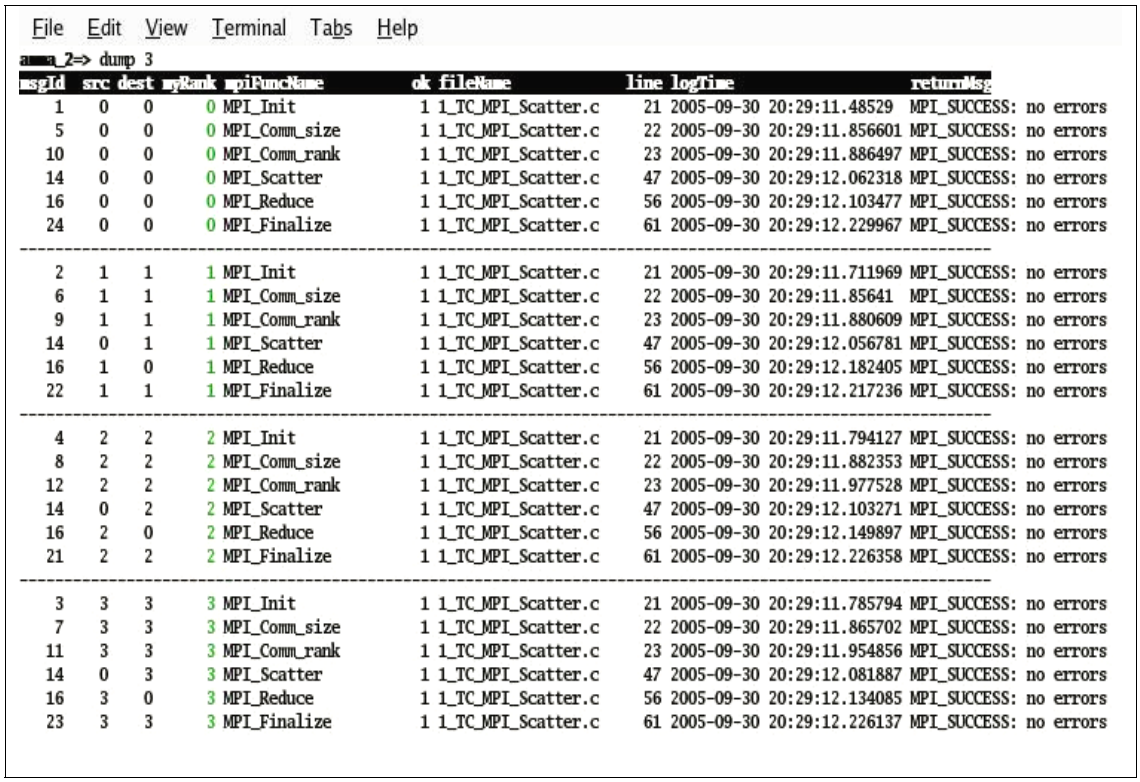
At times, it helps to view all the point-to-point as well as group communications grouped under their respective unique message group ids as assigned by IDLI. This facilitates better understanding of the execution of communication routines which aids in locating erroneous processes. For example, in a point-to-point communication with an MPI_Send – MPI_Recv pair, if the MPI_Send has a message id of *x* then there must be a corresponding MPI_Recv with the same message id. If a call to MPI_Recv with the

message id *x* is missing from the list, then we can safely conclude that there is a bug on the related process. The command *dump 2* lists the executions of MPI routines sorted by their message ids (*msgId).* The entire listing of communication routines executed by a program, using the command *dump 2* is shown in Figure 15. Note that the point-to-point communication with an MPI_Send – MPI_Recv pair is listed under one unique message group id.

```
a____1=> dump 2
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
    1    0    0      0 MPI_Init             1 1_TC_MPI_Send.c       9 2005-10-26 14:25:01.707574 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    2    1    1      1 MPI_Init             1 1_TC_MPI_Send.c       9 2005-10-26 14:25:01.839972 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    3    0    0      0 MPI_Comm_size        1 1_TC_MPI_Send.c      10 2005-10-26 14:25:01.875077 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    4    1    1      1 MPI_Comm_size        1 1_TC_MPI_Send.c      10 2005-10-26 14:25:01.882099 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    5    0    0      0 MPI_Comm_rank        1 1_TC_MPI_Send.c      11 2005-10-26 14:25:01.888407 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    6    1    1      1 MPI_Comm_rank        1 1_TC_MPI_Send.c      11 2005-10-26 14:25:01.9348   MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    7    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-10-26 14:25:01.968825 MPI_SUCCESS: no errors
    7    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-10-26 14:25:02.017125 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    8    0    0      0 MPI_Finalize         1 1_TC_MPI_Send.c      24 2005-10-26 14:25:02.001325 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
   10    1    1      1 MPI_Finalize         1 1_TC_MPI_Send.c      24 2005-10-26 14:25:02.039665 MPI_SUCCESS: no errors
```

Figure 15: Execution of the command *dump 2* which sorts all messages by their ids.

To get a clear picture of all the MPI routines executed by each process we have created the command *dump 3.* This command lists all MPI routines sorted by the rank of each process. When MPI programs are executed on huge networks with a large number of processes this command helps in focusing attention on a set of particular processes of interest. The entire listing of MPI routines executed by program which is sorted by their ranks using *dump 3* is shown in Figure 16. Note that the ranks of processes are shown in green color to make it easy to discern the criteria of sorting.



File Edit View Terminal Tabs Help

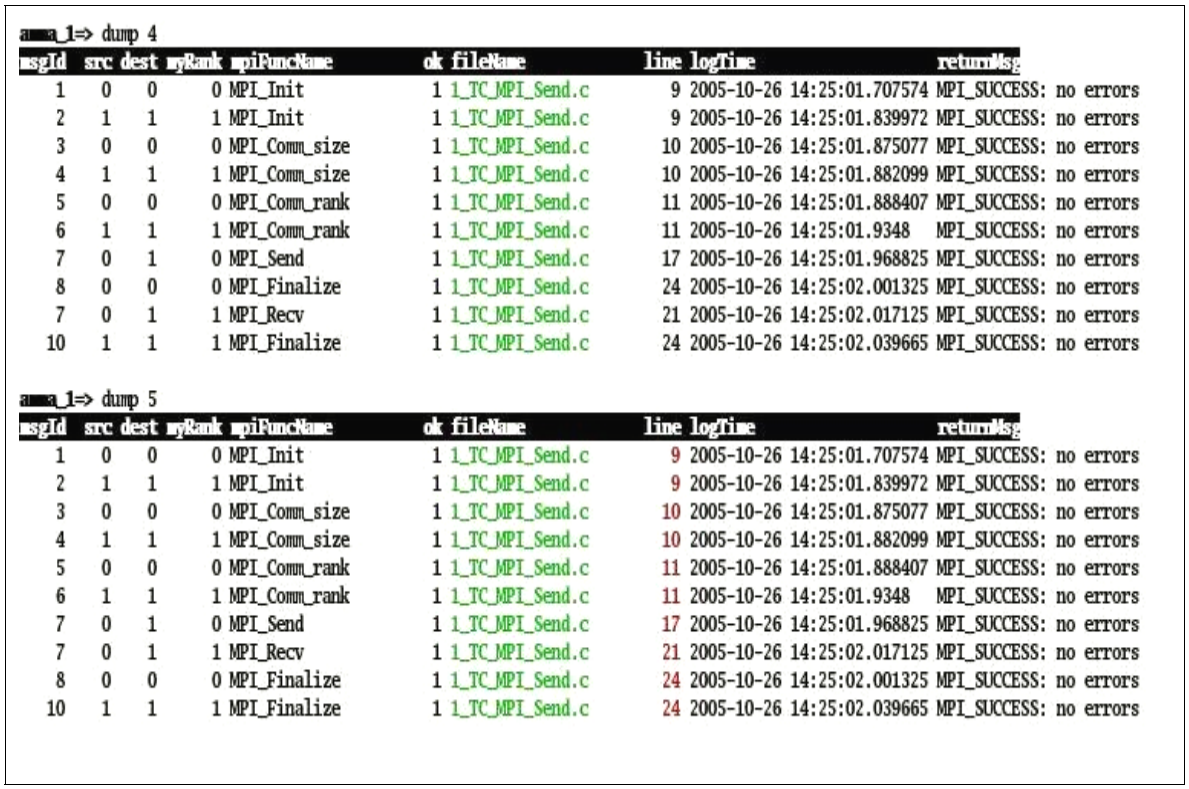
[redacted]_2=> dump 3

msgId	src	dest	myRank	mpiFuncName	ok	fileName	line	logTime	returnMsg
1	0	0	0	MPI_Init	1	1_TC_MPI_Scatter.c	21	2005-09-30 20:29:11.48529	MPI_SUCCESS: no errors
5	0	0	0	MPI_Comm_size	1	1_TC_MPI_Scatter.c	22	2005-09-30 20:29:11.856601	MPI_SUCCESS: no errors
10	0	0	0	MPI_Comm_rank	1	1_TC_MPI_Scatter.c	23	2005-09-30 20:29:11.886497	MPI_SUCCESS: no errors
14	0	0	0	MPI_Scatter	1	1_TC_MPI_Scatter.c	47	2005-09-30 20:29:12.062318	MPI_SUCCESS: no errors
16	0	0	0	MPI_Reduce	1	1_TC_MPI_Scatter.c	56	2005-09-30 20:29:12.103477	MPI_SUCCESS: no errors
24	0	0	0	MPI_Finalize	1	1_TC_MPI_Scatter.c	61	2005-09-30 20:29:12.229967	MPI_SUCCESS: no errors
2	1	1	1	MPI_Init	1	1_TC_MPI_Scatter.c	21	2005-09-30 20:29:11.711969	MPI_SUCCESS: no errors
6	1	1	1	MPI_Comm_size	1	1_TC_MPI_Scatter.c	22	2005-09-30 20:29:11.85641	MPI_SUCCESS: no errors
9	1	1	1	MPI_Comm_rank	1	1_TC_MPI_Scatter.c	23	2005-09-30 20:29:11.880609	MPI_SUCCESS: no errors
14	0	1	1	MPI_Scatter	1	1_TC_MPI_Scatter.c	47	2005-09-30 20:29:12.056781	MPI_SUCCESS: no errors
16	1	0	1	MPI_Reduce	1	1_TC_MPI_Scatter.c	56	2005-09-30 20:29:12.182405	MPI_SUCCESS: no errors
22	1	1	1	MPI_Finalize	1	1_TC_MPI_Scatter.c	61	2005-09-30 20:29:12.217236	MPI_SUCCESS: no errors
4	2	2	2	MPI_Init	1	1_TC_MPI_Scatter.c	21	2005-09-30 20:29:11.794127	MPI_SUCCESS: no errors
8	2	2	2	MPI_Comm_size	1	1_TC_MPI_Scatter.c	22	2005-09-30 20:29:11.882353	MPI_SUCCESS: no errors
12	2	2	2	MPI_Comm_rank	1	1_TC_MPI_Scatter.c	23	2005-09-30 20:29:11.977528	MPI_SUCCESS: no errors
14	0	2	2	MPI_Scatter	1	1_TC_MPI_Scatter.c	47	2005-09-30 20:29:12.103271	MPI_SUCCESS: no errors
16	2	0	2	MPI_Reduce	1	1_TC_MPI_Scatter.c	56	2005-09-30 20:29:12.149897	MPI_SUCCESS: no errors
21	2	2	2	MPI_Finalize	1	1_TC_MPI_Scatter.c	61	2005-09-30 20:29:12.226358	MPI_SUCCESS: no errors
3	3	3	3	MPI_Init	1	1_TC_MPI_Scatter.c	21	2005-09-30 20:29:11.785794	MPI_SUCCESS: no errors
7	3	3	3	MPI_Comm_size	1	1_TC_MPI_Scatter.c	22	2005-09-30 20:29:11.865702	MPI_SUCCESS: no errors
11	3	3	3	MPI_Comm_rank	1	1_TC_MPI_Scatter.c	23	2005-09-30 20:29:11.954856	MPI_SUCCESS: no errors
14	0	3	3	MPI_Scatter	1	1_TC_MPI_Scatter.c	47	2005-09-30 20:29:12.081887	MPI_SUCCESS: no errors
16	3	0	3	MPI_Reduce	1	1_TC_MPI_Scatter.c	56	2005-09-30 20:29:12.134085	MPI_SUCCESS: no errors
23	3	3	3	MPI_Finalize	1	1_TC_MPI_Scatter.c	61	2005-09-30 20:29:12.226137	MPI_SUCCESS: no errors

Figure 16: The command *dump 3* which sorts all messages by their rank.

When executables contain a large number of different MPI files instead of just one file, it becomes a challenge to understand from which file and line number the errors were introduced. To help a user locate the file and line number of an erring MPI call we have created the commands *dump4* and *dump5*. They sort all MPI routines executed by *filename,* and *filename* and *line* respectively, as shown in Figure 17.

The last variation of dump is the command *dump 6*. It lists all MPI routines executed at each process sorted by MPI function names. This is particularly useful when we have a lot of different MPI function calls in a program which is executing on a big network. In such a scenario using *dump 6* we can check if all the processes executed calls to a particular type of MPI function correctly.



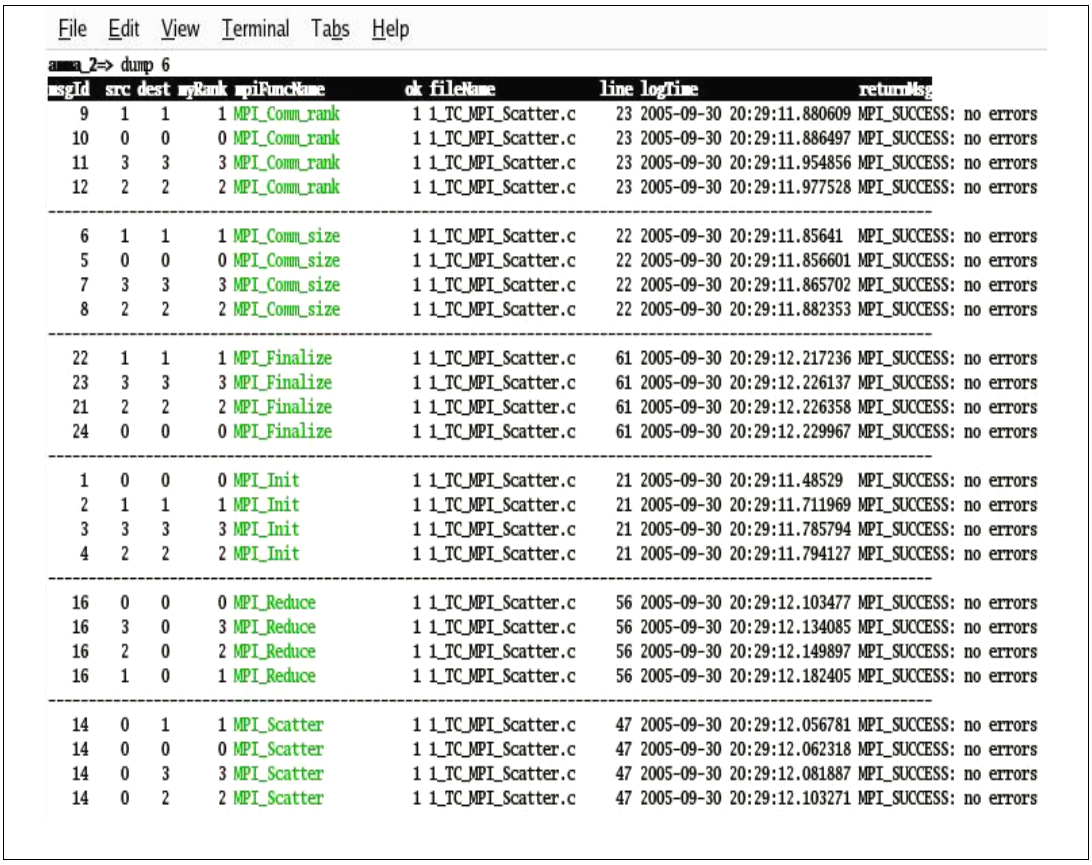
```
[redacted]_1=> dump 4
msgId  src dest myRank mpiFuncName      ok fileName          line logTime                  returnMsg
    1    0    0      0 MPI_Init         1 1_TC_MPI_Send.c     9 2005-10-26 14:25:01.707574 MPI_SUCCESS: no errors
    2    1    1      1 MPI_Init         1 1_TC_MPI_Send.c     9 2005-10-26 14:25:01.839972 MPI_SUCCESS: no errors
    3    0    0      0 MPI_Comm_size    1 1_TC_MPI_Send.c    10 2005-10-26 14:25:01.875077 MPI_SUCCESS: no errors
    4    1    1      1 MPI_Comm_size    1 1_TC_MPI_Send.c    10 2005-10-26 14:25:01.882099 MPI_SUCCESS: no errors
    5    0    0      0 MPI_Comm_rank    1 1_TC_MPI_Send.c    11 2005-10-26 14:25:01.888407 MPI_SUCCESS: no errors
    6    1    1      1 MPI_Comm_rank    1 1_TC_MPI_Send.c    11 2005-10-26 14:25:01.9348   MPI_SUCCESS: no errors
    7    0    1      0 MPI_Send         1 1_TC_MPI_Send.c    17 2005-10-26 14:25:01.968825 MPI_SUCCESS: no errors
    8    0    0      0 MPI_Finalize     1 1_TC_MPI_Send.c    24 2005-10-26 14:25:02.001325 MPI_SUCCESS: no errors
    7    0    1      1 MPI_Recv         1 1_TC_MPI_Send.c    21 2005-10-26 14:25:02.017125 MPI_SUCCESS: no errors
   10    1    1      1 MPI_Finalize     1 1_TC_MPI_Send.c    24 2005-10-26 14:25:02.039665 MPI_SUCCESS: no errors

[redacted]_1=> dump 5
msgId  src dest myRank mpiFuncName      ok fileName          line logTime                  returnMsg
    1    0    0      0 MPI_Init         1 1_TC_MPI_Send.c     9 2005-10-26 14:25:01.707574 MPI_SUCCESS: no errors
    2    1    1      1 MPI_Init         1 1_TC_MPI_Send.c     9 2005-10-26 14:25:01.839972 MPI_SUCCESS: no errors
    3    0    0      0 MPI_Comm_size    1 1_TC_MPI_Send.c    10 2005-10-26 14:25:01.875077 MPI_SUCCESS: no errors
    4    1    1      1 MPI_Comm_size    1 1_TC_MPI_Send.c    10 2005-10-26 14:25:01.882099 MPI_SUCCESS: no errors
    5    0    0      0 MPI_Comm_rank    1 1_TC_MPI_Send.c    11 2005-10-26 14:25:01.888407 MPI_SUCCESS: no errors
    6    1    1      1 MPI_Comm_rank    1 1_TC_MPI_Send.c    11 2005-10-26 14:25:01.9348   MPI_SUCCESS: no errors
    7    0    1      0 MPI_Send         1 1_TC_MPI_Send.c    17 2005-10-26 14:25:01.968825 MPI_SUCCESS: no errors
    7    0    1      1 MPI_Recv         1 1_TC_MPI_Send.c    21 2005-10-26 14:25:02.017125 MPI_SUCCESS: no errors
    8    0    0      0 MPI_Finalize     1 1_TC_MPI_Send.c    24 2005-10-26 14:25:02.001325 MPI_SUCCESS: no errors
   10    1    1      1 MPI_Finalize     1 1_TC_MPI_Send.c    24 2005-10-26 14:25:02.039665 MPI_SUCCESS: no errors
```

Figure 17: Executions of the commands *dump 4* and *dump 5.*

Figure 18 shows a listing of all MPI routines executed by an application, sorted by MPI function names, which has been generated using *dump 6*.

Built-in Query: Locategroup

The command *locategroup N,* where N equals *1, 2, 3, 4,* is used to query executions of group communication routines. This command locates communications exchanged, from a specific file (N = 1), at given line number (N = 2), a particular line number in a file (N = 3), and by the name of an MPI function (N = 4) respectively. This feature is particularly useful when we want to check the details of exchanged messages from a specific MPI group communication function at a given line number of a file. It immediately locates the call and displays its details. When there are lots of files in an executable, this feature saves considerable time since the user does not need to sift



```
File  Edit  View  Terminal  Tabs  Help
amma_2=> dump 6
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                  returnMsg
    9    1    1      1 MPI_Comm_rank        1 1_TC_MPI_Scatter.c    23 2005-09-30 20:29:11.880609 MPI_SUCCESS: no errors
   10    0    0      0 MPI_Comm_rank        1 1_TC_MPI_Scatter.c    23 2005-09-30 20:29:11.886497 MPI_SUCCESS: no errors
   11    3    3      3 MPI_Comm_rank        1 1_TC_MPI_Scatter.c    23 2005-09-30 20:29:11.954856 MPI_SUCCESS: no errors
   12    2    2      2 MPI_Comm_rank        1 1_TC_MPI_Scatter.c    23 2005-09-30 20:29:11.977528 MPI_SUCCESS: no errors
-----------------------------------------------------------------------------------------------------------------------
    6    1    1      1 MPI_Comm_size        1 1_TC_MPI_Scatter.c    22 2005-09-30 20:29:11.85641  MPI_SUCCESS: no errors
    5    0    0      0 MPI_Comm_size        1 1_TC_MPI_Scatter.c    22 2005-09-30 20:29:11.856601 MPI_SUCCESS: no errors
    7    3    3      3 MPI_Comm_size        1 1_TC_MPI_Scatter.c    22 2005-09-30 20:29:11.865702 MPI_SUCCESS: no errors
    8    2    2      2 MPI_Comm_size        1 1_TC_MPI_Scatter.c    22 2005-09-30 20:29:11.882353 MPI_SUCCESS: no errors
-----------------------------------------------------------------------------------------------------------------------
   22    1    1      1 MPI_Finalize         1 1_TC_MPI_Scatter.c    61 2005-09-30 20:29:12.217236 MPI_SUCCESS: no errors
   23    3    3      3 MPI_Finalize         1 1_TC_MPI_Scatter.c    61 2005-09-30 20:29:12.226137 MPI_SUCCESS: no errors
   21    2    2      2 MPI_Finalize         1 1_TC_MPI_Scatter.c    61 2005-09-30 20:29:12.226358 MPI_SUCCESS: no errors
   24    0    0      0 MPI_Finalize         1 1_TC_MPI_Scatter.c    61 2005-09-30 20:29:12.229967 MPI_SUCCESS: no errors
-----------------------------------------------------------------------------------------------------------------------
    1    0    0      0 MPI_Init             1 1_TC_MPI_Scatter.c    21 2005-09-30 20:29:11.48529  MPI_SUCCESS: no errors
    2    1    1      1 MPI_Init             1 1_TC_MPI_Scatter.c    21 2005-09-30 20:29:11.711969 MPI_SUCCESS: no errors
    3    3    3      3 MPI_Init             1 1_TC_MPI_Scatter.c    21 2005-09-30 20:29:11.785794 MPI_SUCCESS: no errors
    4    2    2      2 MPI_Init             1 1_TC_MPI_Scatter.c    21 2005-09-30 20:29:11.794127 MPI_SUCCESS: no errors
-----------------------------------------------------------------------------------------------------------------------
   16    0    0      0 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.103477 MPI_SUCCESS: no errors
   16    3    0      3 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.134085 MPI_SUCCESS: no errors
   16    2    0      2 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.149897 MPI_SUCCESS: no errors
   16    1    0      1 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.182405 MPI_SUCCESS: no errors
-----------------------------------------------------------------------------------------------------------------------
   14    0    1      1 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.056781 MPI_SUCCESS: no errors
   14    0    0      0 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.062318 MPI_SUCCESS: no errors
   14    0    3      3 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.081887 MPI_SUCCESS: no errors
   14    0    2      2 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.103271 MPI_SUCCESS: no errors
```

Figure 18: Execution of the command *dump 6* which sorts by MPI function name.

through the entire list of function calls. The execution of the commands *locategroup 1, 2, 3, 4* are shown in Figure 19.

Built-in Query: Locatep2p

For debugging MPI point-to-point communication routines, it is necessary to locate the messages exchanged by specific files at different line numbers as well as get a view of all messages exchanged by different processes. The query command *locatep2p N* has been created to meet the above goal. The details of *locatep2p* are as follows:

- *N=1*: locates messages exchanged between two processes
- *N=2:* locates messages exchanged between two files of two processes
- *N=3*: locates messages exchanged between two lines of two processes

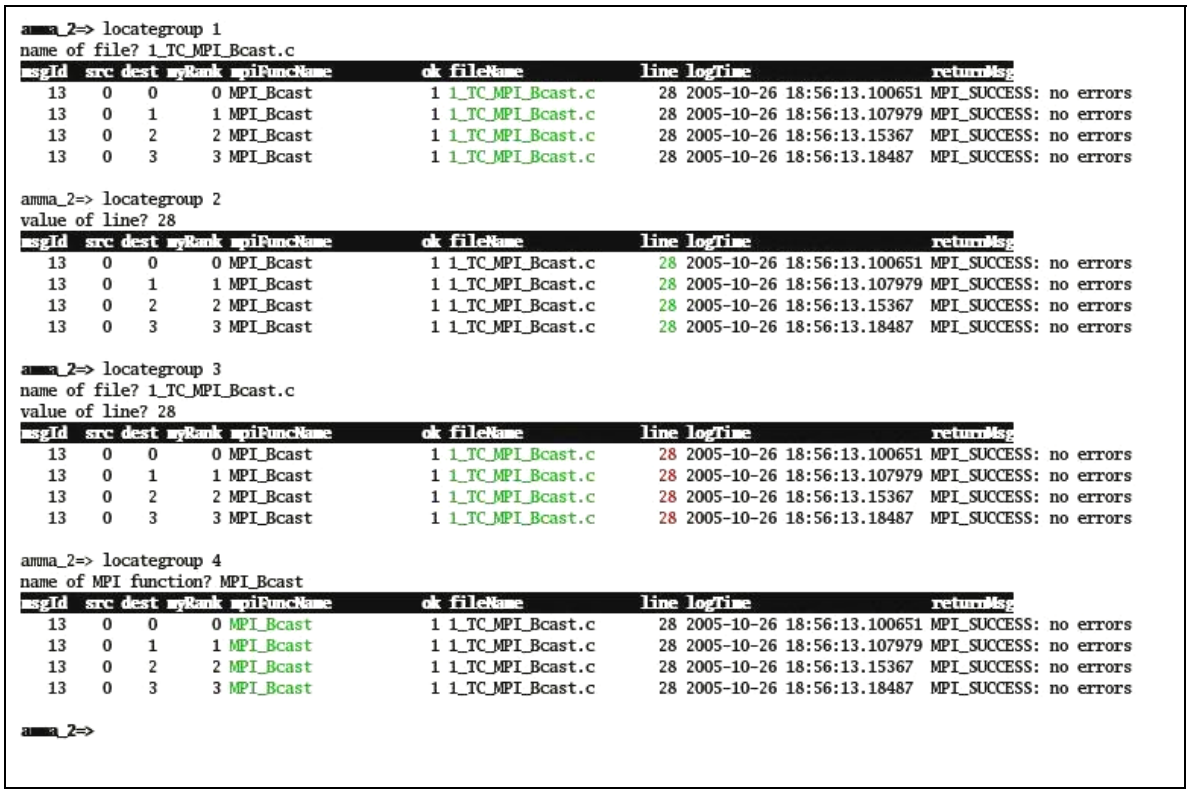


Figure 19: Executions of the command *locategroup N*, N = 1, 2, 3, and 4.

- *N=4*: locates messages exchanged between two lines of two files of two processes
- *N=5*: locates messages exchanged between two filess
- *N=6*: locates messages exchanged between two lines
- *N=7*: locates messages exchanged between two lines of two files
- *N=8*: locates messages by MPI function names.

For example, if we want to find messages exchanged between processes with ranks 0 and 1 we type in the command *locatep2p 1*. We enter 0 when a prompt appears asking for the rank of first process. Similarly we enter 1 when another prompt demands the input for the rank of second process. This brings up a detailed list of messages exchanged between above two processes. Similarly, for other choices of N, various prompts demand different

input information from the user, based on which relevant data is furnished from the user database.

A complete execution of all the options (N = 1 to 8) for *locatep2p* for an MPI program with point-to-point communication routines MPI_Send and MPI_Recv is shown in the Figure 20. Different colors are used for highlighting different columns of data, thus helping distinguish the criteria by which the messages have been sorted. This feature is useful for tracking errors in point-to-point communication since we can zero in on the information we want using a suitable value of N.

Built-in Query: Status

In the built-in queries discussed so far, we had to go to each row and see the data under the column *ok* (0 – incomplete, 1 – complete) to check if an operation that resulted from a call to an MPI routine had completed or not. For large programs running on a big network, this is an extremely time consuming and painful process which is highly error prone. To aid such problems, we have introduced the command *status N,* where N = 0 signifies an incomplete operation while N = 1 denotes a complete operation. A user can type the command *status 0* to get a list of all MPI functions that did not terminate successfully. This command paves the way for faster and accurate debugging. Executions for commands *status 0* and *status 1* are shown in Figure 21.

Built-in Query: Trace

With our current set of built-in queries, if we want to see all messages exchanged by a specific call to a group or a point-to-point communication routine, we have to use the commands *locategroup 4* or *locatep2p 8*. In these queries, a prompt asks the user to

```
amma_1=> locatep2p 1
rank of 1st process? 0
rank of 2nd process? 1
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    6    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-09-30 21:07:58.985435 MPI_SUCCESS: no errors

amma_1=> locatep2p 2
rank of 1st process? 0
name of 1st file? 1_TC_MPI_Send.c
rank of 2nd process? 1
name of 2nd file? 1_TC_MPI_Send.c
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    6    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-09-30 21:07:58.985435 MPI_SUCCESS: no errors

amma_1=> locatep2p 3
rank of 1st process? 0
value of 1st line? 17
rank of 2nd process? 1
value of 2nd line? 21
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    6    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-09-30 21:07:58.985435 MPI_SUCCESS: no errors

amma_1=> locatep2p 4
rank of 1st process? 0
name of 1st file? 1_TC_MPI_Send.c
value of 1st line? 17
rank of 2nd process? 1
name of 2nd file? 1_TC_MPI_Send.c
value of 2nd line? 21
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
-------------------------------------------------------------------------------------------------------------------
    6    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-09-30 21:07:58.985435 MPI_SUCCESS: no errors

amma_1=> locatep2p 5
name of 1st file?  1_TC_MPI_Send.c
name of 2nd file?  1_TC_MPI_Send.c
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    1    0    0      0 MPI_Init             1 1_TC_MPI_Send.c       9 2005-09-30 21:07:58.772073 MPI_SUCCESS: no errors
    2    1    1      1 MPI_Init             1 1_TC_MPI_Send.c       9 2005-09-30 21:07:58.814114 MPI_SUCCESS: no errors
    3    0    0      0 MPI_Comm_size        1 1_TC_MPI_Send.c      10 2005-09-30 21:07:58.847061 MPI_SUCCESS: no errors
    5    0    0      0 MPI_Comm_rank        1 1_TC_MPI_Send.c      11 2005-09-30 21:07:58.860598 MPI_SUCCESS: no errors
    4    1    1      1 MPI_Comm_size        1 1_TC_MPI_Send.c      10 2005-09-30 21:07:58.901841 MPI_SUCCESS: no errors
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
    7    0    0      0 MPI_Finalize         1 1_TC_MPI_Send.c      24 2005-09-30 21:07:58.92273  MPI_SUCCESS: no errors
    8    1    1      1 MPI_Comm_rank        1 1_TC_MPI_Send.c      11 2005-09-30 21:07:58.940017 MPI_SUCCESS: no errors
    6    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-09-30 21:07:58.985435 MPI_SUCCESS: no errors
   10    1    1      1 MPI_Finalize         1 1_TC_MPI_Send.c      24 2005-09-30 21:07:59.011622 MPI_SUCCESS: no errors

amma_1=> locatep2p 6
value of 1st line? 17
value of 2nd line? 21
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
    6    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-09-30 21:07:58.985435 MPI_SUCCESS: no errors

amma_1=> locatep2p 7
name of 1st file?  1_TC_MPI_Send.c
value of 1st line? 17
name of 2nd file?  1_TC_MPI_Send.c
value of 2nd line? 21
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
    6    0    1      1 MPI_Recv             1 1_TC_MPI_Send.c      21 2005-09-30 21:07:58.985435 MPI_SUCCESS: no errors

amma_1=> locatep2p 8
name of MPI function? MPI_Send
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                   returnMsg
    6    0    1      0 MPI_Send             1 1_TC_MPI_Send.c      17 2005-09-30 21:07:58.907827 MPI_SUCCESS: no errors
```

Figure 20: Executions of the command *locatep2p N*, N = 1, 2, 3, 4, 5, 6, 7, and 8.

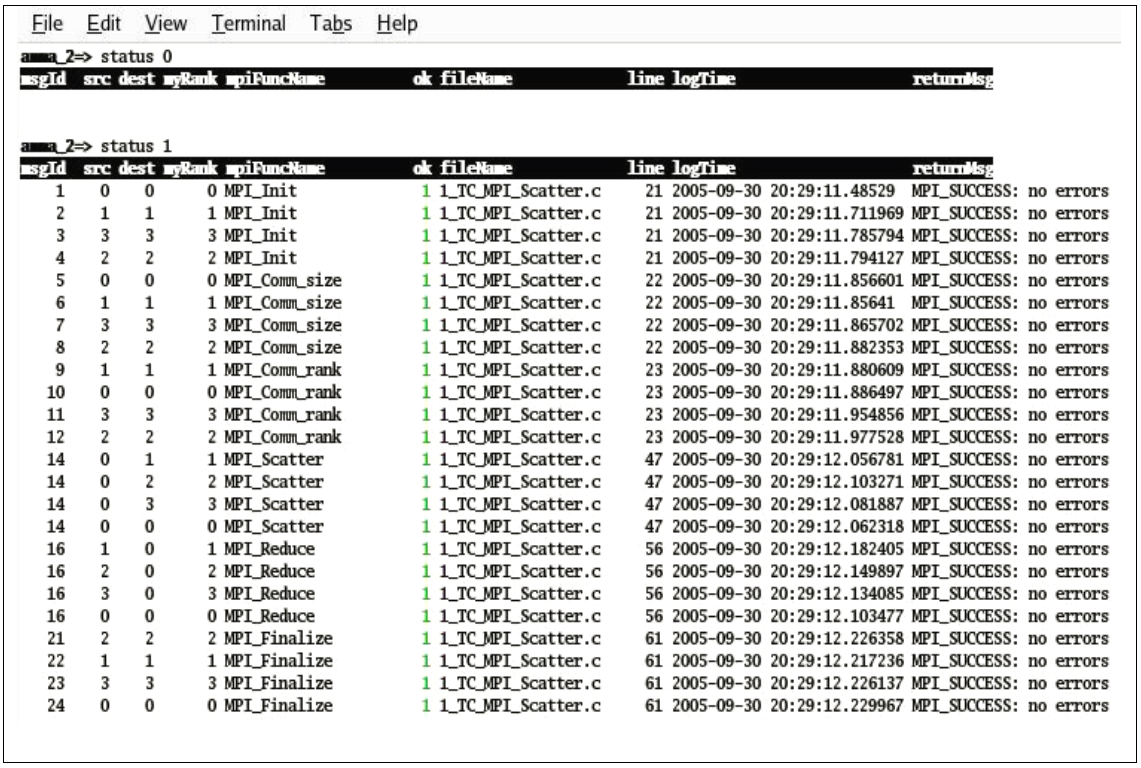


Figure 21: Executions of the commands *status 0* and *status 1*.

enter the name of the MPI function to be traced. Based on the name of the MPI routine, data from the corresponding user database will be displayed. The caveat of these commands is that the list generated will be for all calls in the application to the above function. Consequently we have to search for our particular function call by scanning through the line numbers and file names till we arrive at our desired call. This process is not only time consuming but has high probability for human errors. The risk of making a mistake increases when we have big applications running on a network of thousands of nodes.

To simplify the process we have introduced a built-in query named *trace N*. Using this query a user can track down a particular MPI function by its unique message group id, *N*. For example, let us study a particular MPI_Scatter call, which has been dynamically assigned a message group id of 14 by IDLI when the application had

executed. When we enter the command *trace 14* all messages exchanged by the entire group, whose message group id is 14, is listed. This saves us the effort of seeking out an MPI_Scatter call (whose message group id is 14) by its file name and line number from a list of all MPI_Scatter calls made by an application. Executions for the command *trace N* with various message group ids are as shown in Figure 22.

The listings for the C programs, 1_TC_MPI_Send.c and 1_TC_MPI_Scatter.c, used for testing the built-in queries are shown in Figure 56 and Figure 57 respectively in Appendix 1.

The command: PSQL

The advanced users can write custom SQL [DAT96] queries to analyze specific data from MPI function calls. To achieve this, IDLI provides a way to invoke a PSQL shell. The command so for doing from IDLI's parent menu is *psql N* where N is the session number corresponding to the user database to be queried. If a developer wants to query the user database of session two, the command for invoking the psql shell is *psql 2*. Let us study a customized query. When a big application is running on a large number of processes on a huge network, a user might want to get a quick view of all the MPI routines that were executed successfully on a particular process. For example, for viewing details of all successful calls to MPI routines from a process of rank 0, he types a custom SQL query as shown in Figure 23.

```
amma_2=> trace 14
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
   14    0    1      1 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.056781 MPI_SUCCESS: no errors
   14    0    0      0 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.062318 MPI_SUCCESS: no errors
   14    0    3      3 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.081887 MPI_SUCCESS: no errors
   14    0    2      2 MPI_Scatter          1 1_TC_MPI_Scatter.c    47 2005-09-30 20:29:12.103271 MPI_SUCCESS: no errors

amma_2=> trace 16
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
   16    0    0      0 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.103477 MPI_SUCCESS: no errors
   16    3    0      3 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.134085 MPI_SUCCESS: no errors
   16    2    0      2 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.149897 MPI_SUCCESS: no errors
   16    1    0      1 MPI_Reduce           1 1_TC_MPI_Scatter.c    56 2005-09-30 20:29:12.182405 MPI_SUCCESS: no errors

amma_2=> trace 1
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
    1    0    0      0 MPI_Init             1 1_TC_MPI_Scatter.c    21 2005-09-30 20:29:11.48529  MPI_SUCCESS: n
o errors

amma_2=> trace 24
msgId  src dest myRank mpiFuncName          ok fileName          line logTime                    returnMsg
   24    0    0      0 MPI_Finalize         1 1_TC_MPI_Scatter.c    61 2005-09-30 20:29:12.229967 MPI_SUCCESS: n
o errors
```

Figure 22: Executions of the command *trace N*.

```
IDLI=> psql 1
Welcome to psql 7.4.8, the PostgreSQL interactive terminal.

amma_1=> select
amma_1->        myrank, msggroupid, mpifuncname, mpifuncreturnmsg
amma_1-> from
amma_1->        loginfo, mpifuncsigid
amma_1-> where
amma_1->        opdone = 1 AND
amma_1->        myRank = 0 AND
amma_1->        loginfo. mpifuncid = mpifuncsigid.mpifuncid
amma_1-> order by
amma_1->        logtime;
 myrank | msggroupid |  mpifuncname  |    mpifuncreturnmsg
--------+------------+---------------+------------------------
      0 |          1 | MPI_Init      | MPI_SUCCESS: no errors
      0 |          3 | MPI_Comm_size | MPI_SUCCESS: no errors
      0 |          5 | MPI_Comm_rank | MPI_SUCCESS: no errors
      0 |          7 | MPI_Send      | MPI_SUCCESS: no errors
      0 |          8 | MPI_Finalize  | MPI_SUCCESS: no errors
(5 rows)
```

Figure 23: PSQL shell invoked to write customized SQLs to access specific data.

Replay

When we have tracked down the errors and zeroed in on a set of erroneous processes, we might want to debug the application using a step by step execution of the program on a particular process (from the above set). In other words, from a global context we want to zoom in on to a local context. But to rerun the entire program execution involving a huge set of nodes would not only be time consuming but expensive. On the contrary, if the user is given a tool which can be used to replay the program's execution at specific erroneous processes, it will tremendously enhance the speed of the debugging process. Moreover it would relieve the user from going through a huge amount of data generated during a rerun of an entire application on the whole network.

From the Query Manager a user can replay one or more processes, using the command *replay N*, where N is the rank of the chosen process for replay. We have designed Replay in IDLI with the following features:

- A user can replay an application over the network on her chosen set of processes. She might choose as many erroneous processes as she is comfortable with debugging simultaneously. Consequently information overloading is avoided.
- From one terminal where IDLI is running, a user can simultaneously replay on one or more processes at different nodes of the network. It is to be noted that during the execution of an application on the network, more than one process might have run on the same node (which is a physical machine on the network). IDLI's Replay feature allows a user to simultaneously replay the application for all the processes executed on a particular node of the network. The user may run multiple replay sessions for processes of different ranks and debug them simultaneously. The rank of

the process for which replay is being done is displayed as the title of the xterm window. For example, for a replay of an application on process number 2 the title shows *Idlimq_replay_for_rank_2*. Other than the title of the xterm window, the command prompt for GDB [GNU] also shows the rank as "*[gdb on rank 2]"*. These features help a user to identify a process window correctly. Figure 24 shows simultaneous replays on four processes from a single terminal, using GDB [GNU] as a sequential debugger.

- An application can be replayed multiple times in the same session on any number of processes simultaneously. Once a user completes replay of a process, he may rerun the replay for that process any number of times on the same terminal or quit.

A developer can replay using the sequential debugger of his own choice, for example GNU DeBugger (GDB) [GNU] or Data Display Debugger (DDD) [DAT]. This is possible assuming that the chosen debugger is available on the node on which replay is being run. When a user enters *replay N* (N = rank of process), she is asked to enter the name of a debugger to be used for replay. Also, a programmer can replay an application on different nodes of the network using different sequential debuggers simultaneously for each distinct process. This feature provides the flexibility to use GDB at node 1 or DDD at node 2 depending on the availability and need for specific remote machine and the debugger is opened in a xterm window with display set to the local machine where IDLI is running. Depending on the debugging scenario, a user may use different debuggers for each process with a distinct rank running on the same node. This is demonstrated in Figure 25 where replay is running with two different sequential debuggers GDB and DDD on a couple of processes at a single

Figure 24: Simultaneous replay of four processes, checks and error messages.

node.

- A user can perform sequential debugging and examine contents of communication messages at the same time. He may set break points in the program particularly at the MPI function calls if his intention is to check the content of the messages being

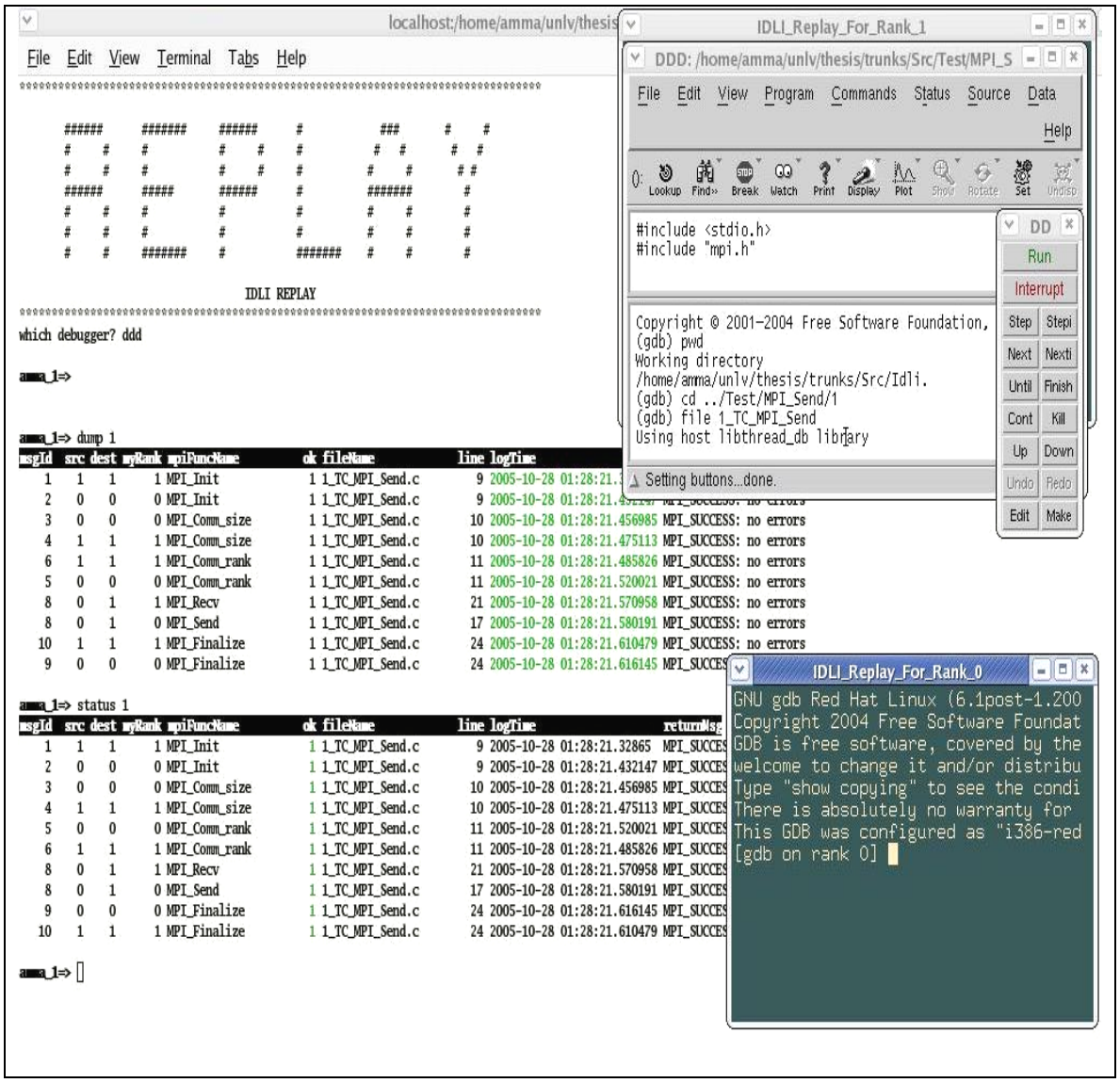


Figure 25: Simultaneous use of the Query Manager and Replay.

exchanged. When the replay reaches the break points at MPI function calls, no calls are made using the MPI communication layer. The data for the MPI calls is furnished from the current user database for the application which is being replayed. If a user finds that the data received by an MPI call was incorrect, he may make changes to the data that was exchanged by MPI routines using the sequential debugger. This will help him see how the outcome of the program changed by correcting faulty data

exchanged by messages. A developer may set breakpoints in the sequential part of the program which does not involve MPI function calls. The sequential debugger will handle them according to its design and methodology. Figure 26 shows replay being done with the Data Display Debugger (DDD) [DAT] where breakpoints have been inserted at calls to MPI functions.

- Query Manager and Replay can be used simultaneously in the same session. A user can use the Query Manager for *post-processing* of communication messages while replay is being used simultaneously for *source level* debugging on a number of processes at a node as shown in Figure 25.

Replay has robust error checking. Since we allow simultaneous replays of processes of multiple ranks, it is necessary to check for the case when a user tries to invoke replay for the same process more than once at the same time. If a user tries to do so, error messages are provided and she is not allowed to start more than one replay session of the same process simultaneously. For example, let us consider the scenario when a replay is already running for a process of rank 0, and the user enters the command *replay 0*. A check is made and the error message "*ERROR: currently there is an active replay on the given rank"* is shown to the user. Then control is returned to IDLI's Query Manager Shell which can be used for querying. Another erroneous situation is when a user tries to exit from the Query Manager while there are ongoing sessions of replay at various processes. Suppose we have replays running for ranks 0, 1, 2, and 3 and the user types in exit to quit the Query Manager. Immediately a warning of the following type is issued to the user "*Replays for ranks: 0, 1, 2, 3 are still running! Please finish or terminate the active replays before exiting.*

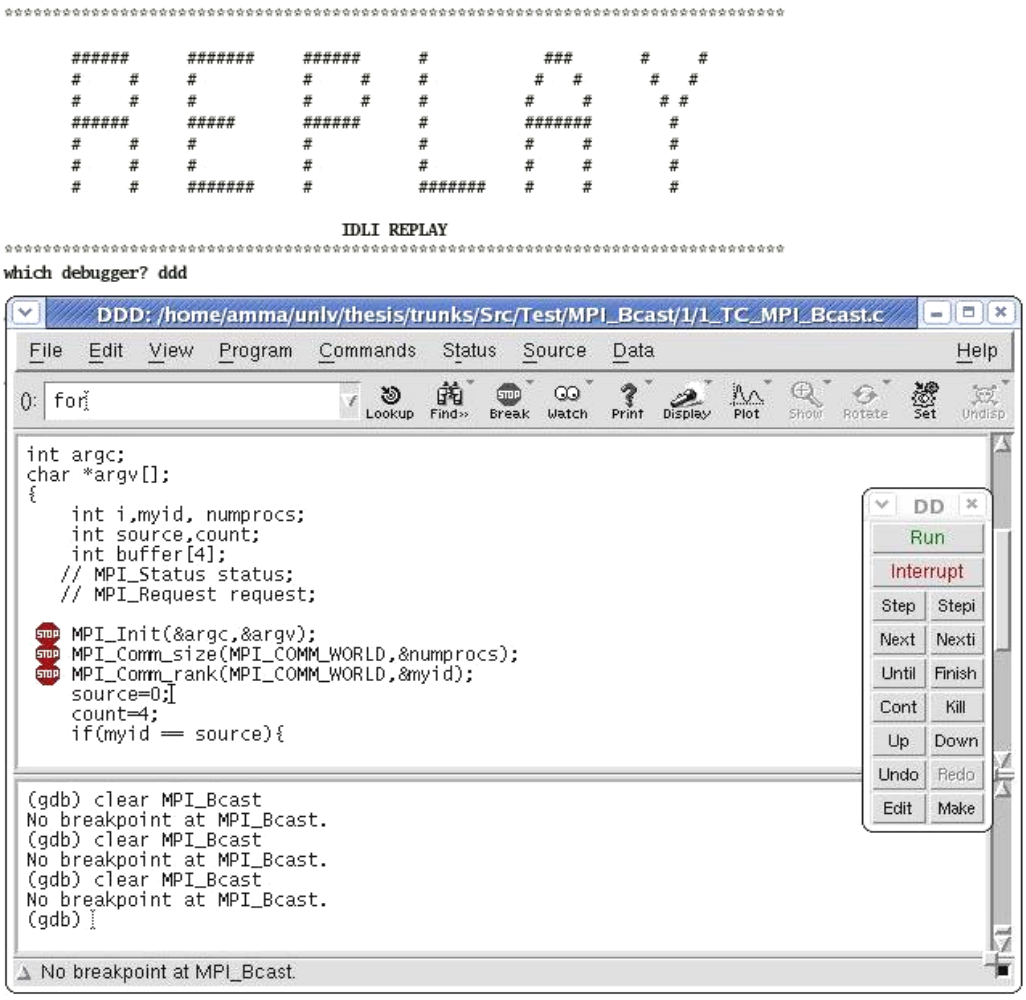


Figure 26: IDLI's Replay in action with the sequential debugger DDD.

Exiting this mode now may lead the database to an inconsistent state." If a user tries to exit the Query Manager when there are replay sessions in action then the database tables managing the secssions would have inconsistent data. For example, a scenario may arise where replays going on for processes with ranks 0 and 1, when the user decides to exit the Query Manager without quitting the replays first. If we allow the user to exit in such a scenario there would be inconsistency in the process of exit. Consequently the data in the tables storing information for replays' sessions will not be updated, that is, sessions will still be marked as active for processes 0 and 1

respectively. As a result when the user tries to replay again on processes 0 and 1 he will not be able to do so since we do not allow more than one replay for a distinct process for an application. Figure 24 shows a session with simultaneous replays for four different processes. When a user tries replay on a process which is already running one, an error message is thrown as depicted in Figure 24. It also demonstrates similar checks done during exit.

While replaying a particular process on a specific node, the MPI communication calls must behave exactly the way it did during actual execution of the program. IDLI ensures that by furnishing data for MPI calls from a user database which stores data from the application's previous execution. It is merely a database read operation and hence extremely fast. Also, all MPI calls will return the same error code as during the original execution. Since the data is provided from a past run, the MPI communication layer used for exchanging messages over the network is not involved. There is no actual execution of the MPI routines in real time during the replay of an application. Consequently, the time that was used by the MPI routines for blocking or synchronization, and message exchanges is thus saved in replay. Hence the process of debugging is significantly speeded up. In a nutshell, Replay enables a user to debug a large MPI application with optimal resource usage and without information overloading [PED03].

This concludes our exploration of the features of IDLI. The key motivational factors during the entire design and development cycles of IDLI were to provide features that would do the following:

- enhance the quality of debugging capabilities of the tool,
- make it user friendly and intuitive which helped flatten the learning curve,

- provide specific debugging data to the user thus avoiding information overload,
- make it faster and reliable with robust error checking, and
- aid in optimal usage of resources.

The demonstrations also showed how IDLI's Query Manager and Replay in conjunction with a sequential debugger enables a user to:

- view global context information and map it to local context,
- customize it by writing user specific SQL queries,
- generate relevant and precise debugging information, and
- simultaneously debug source level sequential code as well as errors in the messages exchanged by processes.

Debugging IDLI with IDLI during its development cycle

During the development cycle of IDLI, first the databases were modeled and created, followed by the coding and testing of the Query Manager. After that we started implementing the wrapper functions in the native C library. The intention behind doing so was to use the software while it was being developed to debug itself with the aid of the Query Manager. Quite a few errors were trapped and resolved during the development process using the above methodology of debugging a product using itself. For example while developing IDLI, there was a bug which we were not aware of, in a call to the MPI function named MPI_Comm_rank. While debugging another bug we did a *dump 1*, when we noticed that for all processes, there were no entries for MPI_Comm_rank after execution of MPI_Comm_size. In our program, execution of MPI_Comm_rank should have followed that of MPI_Comm_size. This brought into view this unknown bug which

was then resolved.

Another instance of the tool's self-debugging occurred while implementing wrappers for MPI_Isend and MPI_Irecv. A bug was caught using *dump 3* where we noticed that a call to MPI_Irecv was not getting executed at its destined node. A complex bug was encountered while writing the wrapper for the MPI function MPI_Wtime. This subroutine returns the current value of time as a double precision floating point number of seconds. Each time we printed the time being returned by MPI_Wtime we got the same value! When we wrote a custom SQL query to see the value of the return data which was stored in the database, it showed different values for each call to the above function. Finally we resolved it by correcting the pointer references and arguments being passed in the wrapper and related functions. In this manner we used our tool several times to debug itself which also helped in streamlining many of the features for easier usability. Also, considerable time and effort that would have been spent in writing print statements for debugging were saved by using IDLI's Query Manager to debug IDLI.

Summary

In the first section of this chapter we discussed the architecture and organization of our message debugger IDLI. In the next section we explored the features of IDLI in details. We demonstrated the functionality of each feature with examples as well as screenshots of executions of corresponding commands. We also discussed how new features were developed as a result of reflection upon the limitations of the inbuilt queries already developed in the debugger. An interesting aspect that demonstrated our tool's utility and convenience was the discussion about how IDLI was used to debug its own software while it was being developed.

CHAPTER 5

IMPLEMENTATION DETAILS OF IDLI

The architecture of IDLI comprises three layers. A distributed relational SQL database forms the backend. The middle layer is a native C [KER88] library which has wrappers for MPI functions and a simple intuitive user shell interface is the front-end. The user interface can be used to query messages, replay a program and write interactive customized SQL queries to directly access data from database tables. This chapter gives implementation details of all three layers, highlighting complexities faced, algorithms used and some interesting SQL queries.

Backend: Distributed Relational SQL Database

A distributed relational SQL database [POS] at the backend is used to achieve the following:

- generation and management of data for multiple users' sessions,
- logging meta-data generated for each MPI call, and
- storing data from messages exchanged by MPI calls in a user's application.

To facilitate segregation and remove redundancy we have divided data storage into two parts. The first part consists of data for management of multiple users' simultaneous sessions. This data is stored in a database known as system database. It does not contain debugging information from MPI calls of the application. The second part stores meta-data and details of messages exchanged by MPI calls in each session. This data is stored

in databases known as user databases. There is a distinct user database for each session of a user. Unique names of user databases are generated by appending the username and session number. Figure 27 shows a detailed overview of IDLI's Relational DataBase Management System (RDBMS).

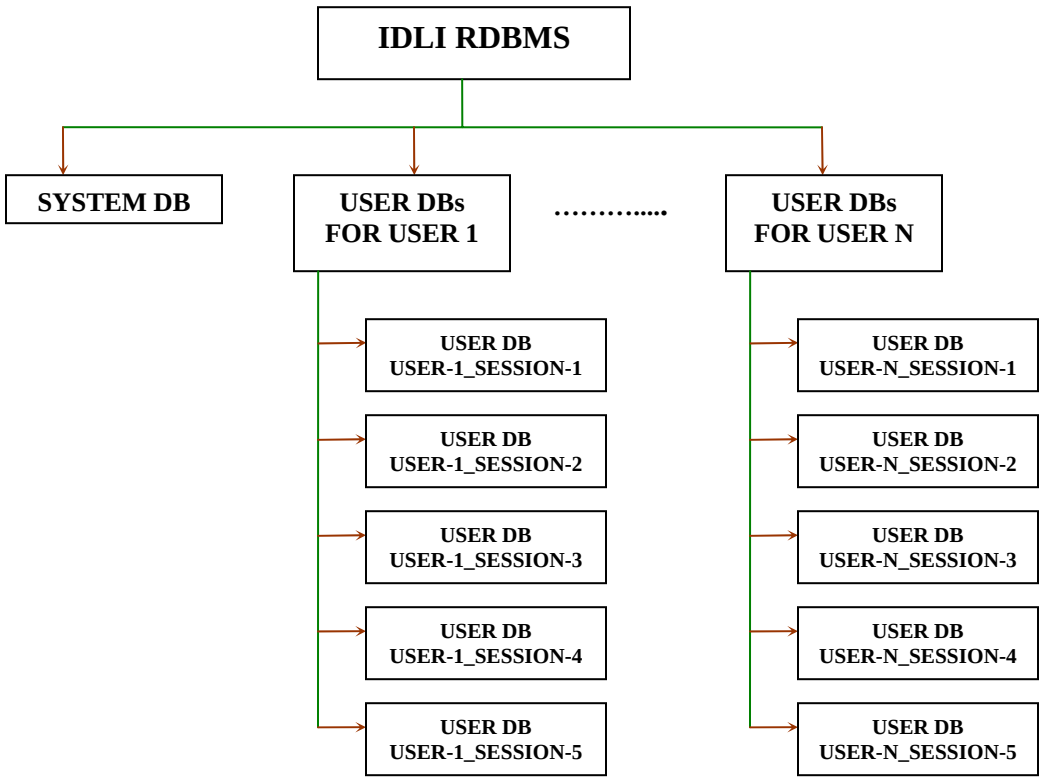


Figure 27: Details of IDLI's RDBMS.

Backend: Multiple users' sessions data generation and management

As explained in the previous chapter, IDLI supports sessions for multiple users and stores debugging data for each user's session up to a maximum of five sessions. Consequently a user can access the debugging data of an application executed in any of his previous five sessions. Once the maximum of five sessions is reached for a user, the user database for the earliest session (out of the five allotted sessions) is deleted. The

session number thus freed is assigned to the user's current session. A new user database for storing debugging information of the present session is created. Thus to help optimize the usage of space on the database server, session numbers are cyclically reused and limited to the maximum number for each user. A future version of IDLI might allow the user to choose the maximum number of sessions he wants to store. A relation named *userinfo,* which has been created in IDLI's system database, stores the users' session management data. Figure 28 shows details of the relation *userinfo* along with some example data.

user Name	session No	session Time	db Name	session Active	db Created
amma	1	2005-08-28 23:15:40.809558	amma_1	F	T
amma	2	2005-08-28 23:15:40.809558		F	F
amma	3	2005-08-28 23:15:40.809558		F	F
amma	4	2005-08-28 23:15:40.809558		F	F
amma	5	2005-08-28 23:15:40.809558		F	F

Figure 28: The *userinfo* relation and tuples of data for a user named *amma.*

Backend: Logging Meta data generated for messages for each MPI call

LAM MPI does not provide user accessible identification numbers for messages. To distinctly identify a message we generate unique message group ids. During point-to-point and group communication, all messages related to a call are grouped together, hence the name message group id. The message group id facilitates storing of message related information as a tuple in a relation. During a call to an MPI routine, a substantial amount of meta-data is generated for each message. This meta-data aids in querying and debugging. To store meta-data we have created a relation named *loginfo* in the user database. Figure 29 shows the details of the relation *loginfo* along with example data.

COLUMN NAME	DESCRIPTION	EXAMPLE DATA
msgGroupId	Message Group Id [Primary Key]	6
comm	Communication Handle	0x80c77a0
rankSender	Rank of Sender	0
rankReceiver	Rank of Receiver	1
myRank	Rank of process where application is running	0
opDone	Operation Done [0 – incomplete, 1 – complete]	1
tag	Tag of message	-
hostIPAddress	IP Address of Host	192.168.0.2
hostName	Host Name	localhost.localdomain
filename	Name of File from which MPI call is made	1_TC_MPI_Send.c
lineNo	Line Number from which MPI call is made	17
mpiFuncId	Unique Integer Id for MPI routine instead of storing its name as a string	155
mpiFuncReturnCode	Return code from call to MPI routine	0
mpiFuncReturnMsg	Return message corresponding to return code from call to MPI routine	MPI_SUCCESS: no errors
logTime	Time when call was placed to MPI routine	2005-09-2221:36:38.667086

Figure 29: The *loginfo* relation and example data for the function MPI_Send.

Backend: Storing data from messages exchanged by MPI routines

In point-to-point and group communication routines, along with the meta-data, we also need to store the data received, and the return values sent, by an MPI function. We do so by storing data specific to a particular MPI function in a relation with the same name as that of the function. For example, all data related to calls to the MPI function, MPI_Recv, are stored as separate rows within a table named *MPI_Recv* in the user database. The relation *MPI_Recv* uses the columns msgGroupId and myRank as a reference key to refernce the relation *loginfo*.

If we had stored data for each MPI call in the relation *loginfo* instead of a separate table for each function, it would have made a single relation huge in size. This in turn would have increased retrieval times. All data related to each MPI call could have been stored in a binary format in the relation *loginfo.* In that case, we would have to convert

the data back to the data type of each stored parameter, during data retrieval. This would have required the maintenance of an ordered list with the count and data type of each of the stored parameters. This process would have unnecessarily increased the complexity and processing time.

Hence to facilitate quick retrieval, optimize space requirements and remove unnecessary complex processing we have stored data specific to a particular type of MPI function in a relation of that name. Figure 30 shows the relation *MPI_Recv* for the function MPI_Recv. For example, if we had stored this information in the relation *loginfo*, we would have to convert the values in all the columns of *MPI_Recv*, that is, `pBuf, count, datatype, pStatus, statusSource, statusTag, statusError, statusLength`, and `data` into binary format and then store them. On the contrary, since we have stored the data in a separate relation *MPI_Recv* the retrieval is much faster. This is due to the fact that most of the parameters passed in the arguments of the function MPI_Recv are stored in separate columns in their original formats, so no conversion is required except for the column named `data`. The column for `data` stores the information received (in arrays) by MPI_Recv from a corresponding MPI_Send. As the number of conversions from binary format to original data types is considerably reduced during data retrieval, processing is faster and simpler.

Data from the messages exchanged by MPI functions is stored to aid a user to query the data from a specific function call by writing customized SQL queries. For example, when a developer has tracked down a specific MPI function call, and needs to know the data exchanged by that routine, he can write a customized SQL query for specific data retrieval.

COLUMN NAME	DESCRIPTION	EXAMPLE DATA
pBuf	Stores initial address of receive buffer	0xbffff634
Count	Number of elements to be received	1
Datatype	Datatype of each receive buffer element	0x80c62a0
Source	Rank of the source task in comm or MPI_ANY_SOURCE	0
Tag	Message tag or MPI_ANY_TAG	1234
Comm.	Communicator handle	0x80c77e0
pStatus	Stores address of status object	0xbffff620
statusSource	Source rank stored in object status	0
statusTag	Source tag stored in object status	1234
statusError	Error stored in object status	0
statusLength	Message length stored in object status	4
msgGroupId	Message Group Id [Primary Key]	8
rankReceiver	Rank of current process [Primary Key]	1
data	Data received in binary format	.\026\000\000

Figure 30: The *MPI_Recv* relation.

For a huge parallel application running on a large number of processes, writing a customized SQL to view the requisite data is probably a faster debugging process than replaying the entire application.

Backend: Mapping MPI function names to unique integer function ids

As explained in the previous chapter, the wrapper functions (in IDLI's native C library) frequently need to fetch specific data for MPI functions from the relation *loginfo*, through SQL queries. Hence, along with other meta-data, we need to store the name of the MPI function in the relation *loginfo*. LAM-MPI does not provide user accesible unique integer ids for the MPI functions. So the names of the MPI functions have to be stored as strings in each tuple which is not space optimal. Also, in the source code, a developer has to hard code the names of the MPI functions in the WHERE clause of the embedded SQL queries. Figure 31 shows an embedded SQL query where the function name 'MPI_Allgatherv' has been hard coded. Moreover, during data retrieval, string matching has to be done for each tuple with the given MPI function's name. This will

```
sprintf (fgRequestString,
         "SELECT min(msgGroupId) FROM loginfo
          WHERE mpiFuncName = 'MPI_Allgatherv' AND opDone = %d",
         FALSE);
```

Figure 31: Example of SQL with MPI function names as strings

slow down the retrieval process.

A better way of doing the same is to use integer comparison instead of string matching. To do so unique integer ids are assigned to each of the MPI functions. These unique integer ids are stored along with their names and signatures in a relation named *MPIFuncSigId.* Consequently in the embedded SQL queries, a developer can use the unique integer ids for each MPI function instead of hard coding its name, as shown in Figure 32 . This makes the SQL queries generic. Hence they remain unaffected by future addition or change of names of the MPI routines. Also, data retrieval is fastened, since integer matching can be done in each tuple using the integer ids instead of string matching with the function's name.

```
sprintf (fgRequestString,
         "SELECT min(msgGroupId) FROM loginfo
          WHERE mpiFuncId = 9 AND opDone = %d",
      FALSE);
```

Figure 32: Example of SQL with MPI function ids instead of names.

As more routines are added or their names are modified in LAM MPI, we need to add or modify these functions' details only in the relation *MPIFuncSigId.* Consequently the source code remains unaffected since no function names are hard coded in it. It is to be noted that in the relation *loginfo* the unique integer id of each MPI function is stored instead of its name. As mentioned earlier, this is done to optimize space and processing time. Figure 33 shows the details of the relation *MPIFuncSigId.*

mpiFunc Id	mpiFunc Return	mpiFunc Name	mpiFunc ArgNum	mpiFunc Arg
9	int	MPI_Allg atherv	8	{“void *”, int, MPI_Datatype, “void *”, int *, int *, MPI_Datatype, MPI_Comm }

Figure 33: The *MPIFuncSigId* relation with data for MPI_Allgatherv.

Middle Layer: Native C Library

A native C [KER88] library provides a layer between the user’s program being debugged and LAM-MPI library as demonstrated in Figure 34. It consists of wrapper functions for each MPI routine that provides interaction between the SQL database [POS] at the backend, and the user’s program (in debug mode) or the debugger’s front-end user interface (in replay mode). The wrappers also possess intelligence for initialization, database processing and locks, understanding whether the debugger has to be started in debug or replay mode, and MPI function specific processing. When an application is executed in debug mode, a wrapper function in the native C library does the following:

- inserts initial data in the SQL database,
- calls the original MPI function,
- stores data from the call in the user database after the MPI function returns, and
- returns control to the user’s program.

In replay mode, the native C library’s wrapper functions provide data for MPI calls from stored data of a previous execution of the application. No calls are made to MPI functions during replay.

Middle Layer: Methodology used for interception of MPI calls

MPI calls are intercepted through the use of C preprocessor #define macros that

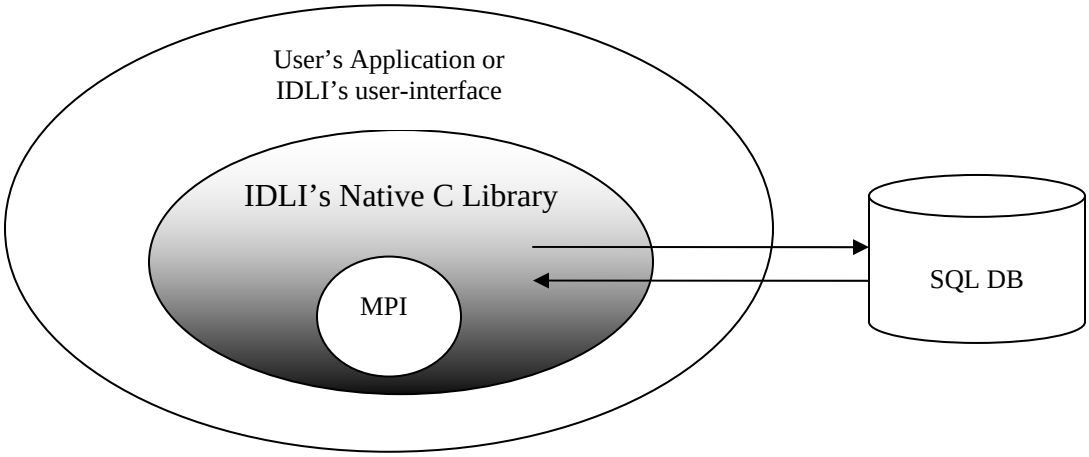


Figure 34: Schematic Illustration of Native C Library's Role.

redirect the MPI call to a wrapper function in IDLI's native C library. For example, an interception of a call to MPI_Init, is done in the following manner:

1. The signature of MPI_Init is `int MPI_Init (int *pargc, char ***pargv)`. The corresponding wrapper function in native C library is `int IDLI_MPI_Init (int *pargc, char ***pargv)`.
2. The call to MPI_Init is intercepted by the C preprocessor `#define` macro: `#define MPI_Init (A, B) (SET_FILE_LINE, IDLI_MPI_Init (A, B))`. The a macro which sets global variables with the file name and line number from which the MPI call was made is `SET_FILE_LINE`.
3. The MPI library is included in the user program with `#include "mpi.h"`. The standard MPI header file `mpi.h` has been renamed to `OrigMPI.h` and the header file `mpi.h` has been replaced with a header file of the same name. This new header file has has a `#inculde IDLImpi.h` followed by all the `#define` macros like the one shown above. `IDLImpi.h` is a header file that has the prototypes for wrapper functions like above mentioned `IDLI_MPI_Init`. `IDLImpi.h` has a

#include `OrigMPI.h` which enables the inclusion of the MPI libraries. Figure 35 demonstrates the header file swapping process.

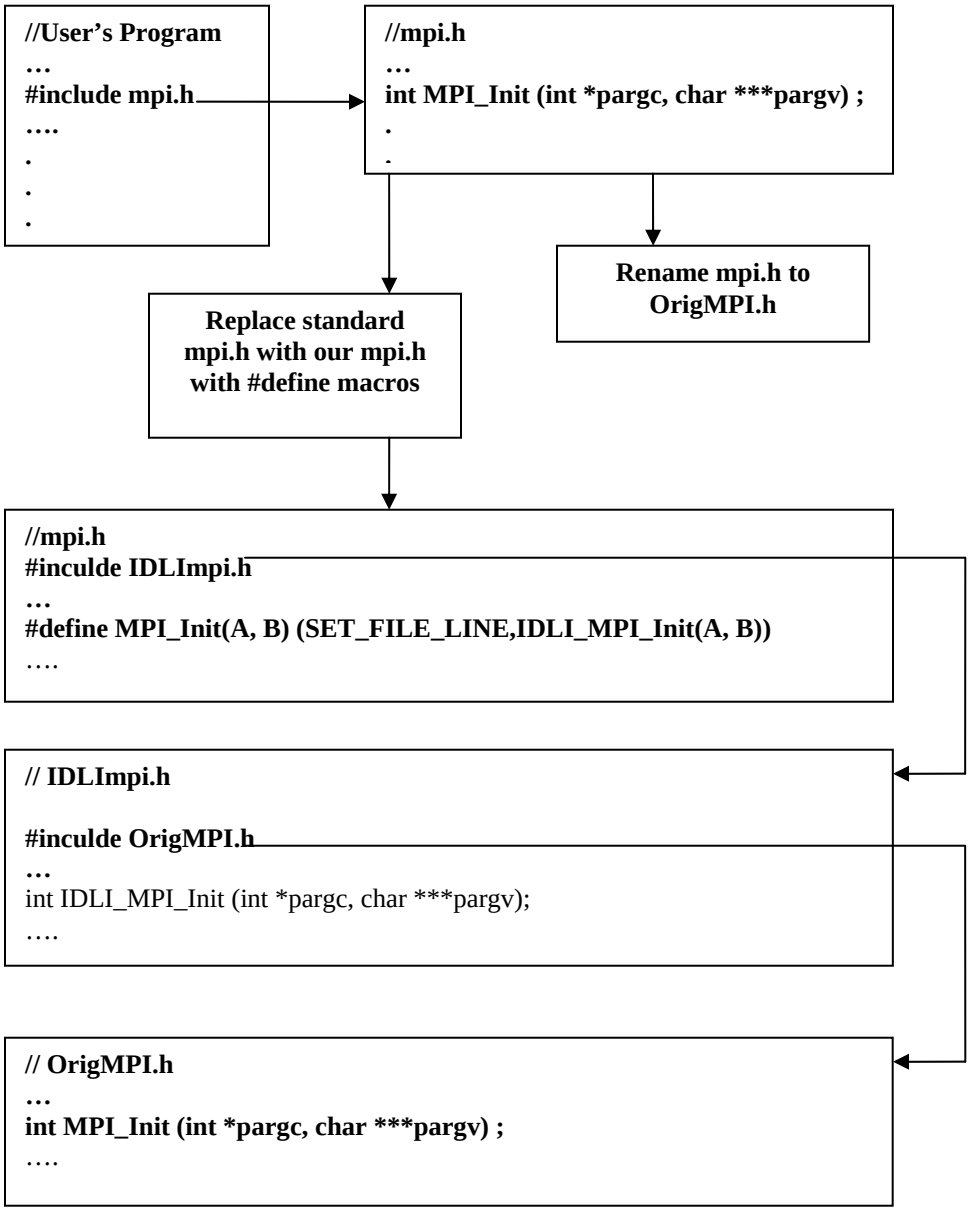


Figure 35: Diagram showing swapping of header files to intercept MPI calls.

4. During compilation of a user's program, he needs to specify the IDLI directory in the include path, as in –I…/IDLI. Consequently, the IDLI specific mpi.h is included

instead of the standard mpi.h. Also, he must link the IDLI library by using the flags –L../IDLI –lidli during the linking phase in gcc. At compile time a flag is checked (-DDEBUG as an argument in gcc [GCC]) in order to determine whether the program should be compiled in IDLI debug mode or not. If it is not defined the standard MPI header file is included else we replace the standard MPI header file with our own header files as explained above.

Middle Layer: Flow of control in the wrapper functions

Once the MPI calls are intercepted we check whether IDLI is running in replay or debug mode. When a user enters the command to replay an application, a flag is set to replay mode. The flag is set to debug mode during compile time as explained in the previous section. At the start of each wrapper function we check this flag to determine whether the processing should be for replay or debug mode. In debug mode, data is written to the user database, while in replay mode the stored data is retrieved from it.Each process writes debugging information directly to the database. During initialization, which is done in the wrapper function IDLI_MPI_Init, each process opens a connection to the system database to check the user's information. The first check is to determine whether the current (logged on) user is a new user or a returning one who has had previous sessions with IDLI.

If she is a new user, five rows (maximum number of sessions allowed for each user is five) are created in the relation *userinfo.* A session number of 1 is assigned to the user's current session and a user database (for storing debugging data) is created with a name formed by appending the username and session number. The `dbCreated` and `sessionActive` flags in the relation *userinfo* are set to true. This enables other

processes spawned by the user's application to know that the current user exists, there is an active session for her and that a user database has already being created. Consequently, they can go ahead and retrieve the requisite connection information from the relation *userinfo*, open a connection to the relevant user database and start writing to the tables. The user database is created by the process which executes first (on any of the nodes of the network being used). While creating the user database a lock is held on the relation *userinfo* so that other processes wanting to do the above mentioned checks for the user do not get ghost data. After the user database is created and the data in relation *userinfo* is updated, the lock is released. This enables other processes that were waiting to continue with their checks to proceed.

If the user already exists, we check whether there is an active session present for the user. If so, the user database information is retrieved and a connection is opened to that database. If not, we check whether an inactive session number is available for that user that can be assigned to her current session. If more than one free (inactive) session number exists, the minimum one is chosen. After that, the process of user database creation as mentioned in above paragraph is performed. If no free session exists for the user, that is, she has already had five previous sessions of debugging, then we cyclically free a session number. Based on the algorithm first in first out (FIFO), we assign the earliest (used) session number to her current session, delete the corresponding existing user database and create a new one.

Once each process has created a connection to the correct user database to store the current session's debugging data, the original connection to the system database containing the relation *userinfo* is closed. Next each wrapper function gathers data

specific to the MPI function call, generates a unique message group id and inserts a row in the relation *loginfo* in the user database. Then a call to the MPI function is made and the corresponding tuple in *loginfo* is updated with the return parameters from a successful call. Since we update the tuple after the call is made, we can keep track of the execution times of each MPI call.

When an MPI call returns successfully, all return data along with data associated with parameters passed in the function call are stored in a table with the same name as the corresponding MPI function. In addition, for point-to-point and group communication routines, processing intelligence which is unique to each routine is incorporated in the wrappers.This is done to assign each group the same message group id, to check whether the group operation has ended successfully or not, and other requisite verifications. Locks are held on database tables whenever required to stop more than one process from updating or writing to a table simultaneously and to prevent them from reading ghost data.

Middle Layer: Implementation details of the MPI wrapper functions

One of the most important tasks performed by the wrapper functions is to group together related messages exchanged during an MPI call. Whether it is a point-to-point or group communication, to successfully locate errors occurring at different nodes of the network a user needs to know the details of messages sent from a node to a group and vice versa. Unfortunately, there is no built–in mechanism under LAM-MPI which assigns unique identification numbers to related messages exchanged by point-to-point or group communication routines. Consequently for each wrapper function, we had to implement different algorithms for assigning unique message group ids to related messages. As

mentioned in the previous chapter, in IDLI we have implemented wrappers for twenty four commonly used MPI functions [WIL05]. In the following sections we shall discuss the implementation details for some of the challenging point-to-point, group and preliminary routines' wrapper functions.

Point-to-point communication routines: MPI_Isend and MPI_Irecv

Our discussion starts with the implementation details for an asynchronous point-to-point communication done by the functions MPI_Isend and MPI_Irecv. Figure 36 shows details of these functions [MES].

Our task is to group together each set of corresponding MPI_Isend and MPI_Irecv calls and assign them the same message group id number, say GID. Since these function calls are asynchronous, once the calls are made the program control immediately goes to the next program statement, without waiting for the completion of the copying of data from the application buffer. In other words, an MPI_Isend call returns as soon as copying of data from application buffer to system buffer begins; an MPI_Irecv call returns as soon as copying of data from system buffer to application buffer starts. When a process invokes, MPI_Isend, a row with a unique message group id (generated afresh) is inserted in the relation *loginfo*. When the MPI_Isend returns successfully, a corresponding row is inserted in the table MPI_Irecv with all data about the call. But the `opDone` column in relation *loginfo* is not updated to `true (1)` since the operation will be considered successful only when the corresponding MPI_Irecv goes through successfully.

MPI_Isend has to specify a destination in its function parameter dest. But MPI_Irecv can receive from any source when its function parameter source is set to `MPI_ANY_SOURCE`. In that case, we have to determine the source of the message to

Function	MPI_Isend is a point-to-point communication routine.
Signature	`int MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request)`
Arguments	buf - is the initial address of the send buffer (choice) (IN) count - is the number of elements in the send buffer (integer) (IN) datatype - is the datatype of each send buffer element (handle) (IN) dest - is the rank of the destination task in comm (integer) (IN) tag - is the message tag (positive integer) (IN) comm - is the communicator (handle) (IN) request - is the communication request (handle) (OUT)
Work	Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to MPI_Wait or MPI_Test indicate that the non-blocking send has completed.
Function	MPI_Irecv is a point-to-point communication routine
Signature	`int MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request)`
Arguments	buf - is the initial address of the receive buffer (choice) (OUT) count - is the number of elements in the receive buffer (integer) (IN) datatype - is the datatype of each receive buffer element (handle) (IN) source - is the rank of source or MPI_ANY_SOURCE (integer) (IN) tag - is the message tag or MPI_ANY_TAG (positive integer) (IN) comm - is the communicator (handle) (IN) request - is the communication request (handle) (OUT)
Work	Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to MPI_Wait or MPI_Test to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

Figure 36: Details of functions MPI_Isend and MPI_Irecv.

match the call. The list of function arguments of MPI_Irecv does not have an object of the type `MPI_Status.` As a result, we have to devise a way to extract the source from the opaque object MPI_Request which is a communication handle (refer to Figure 36 for the function arguments of MPI_Irecv). This is done using the function MPI_Test on the MPI_Request object. MPI_Test tests for the completion of a send or receive and when the operation completes successfully it outputs an object of type `MPI_Status` which contains the message's `source, tag, error` and `length`.

Once we have extracted the source, the next step is to obtain the message group id of the corresponding MPI_Isend from the relation *loginfo* (see Figure 29) and assign it as the message group id of current MPI_Irecv call. To do so, we retrieve the message group id of the tuple in the relation *loginfo* whose:

- `mpifuncsigid` is equal to unique integer id of MPI_Isend,
- `rankSender` is that of source obtained above,
- `rankReceiver` is the rank of process which has received the message through MPI_Irecv, and
- `opDone` is `false(0)`.

The result set for the above query might return more than one tuple for the case when more than one message has been sent from the same source to the same destination. An example of such a case is the repeated execution of an MPI_Isend in a *for* loop which is sending different messages to the same destination. The challenge now is to identify the correct message group id, GID, pertaining to our specific MPI_Isend from the above result set. LAM-MPI preserves the order in which messages are sent from a source to a destination, that is, if there are two messages sent from source A to destination B, the message sent first will be received first, and so on. Though the order is preserved, in asynchronous communication the time when the destination process will receive the message is not guaranteed. Consequently to extract our GID we have to choose the minimum message group id from the result set obtained above. Message group ids are generated in increasing order. Once we get the GID, we assign it as the message group id for our specific MPI_Irecv call in the relation *loginfo*. Next, we update the necessary columns in the tuple for our specific MPI_Irecv call and mark the operation as complete

(`opDone is true (1)`). In the relation *loginfo* the corresponding MPI_Isend call's `opDone` is also updated to `true (1)`. After that, a row is inserted in the table *MPI_Irecv* with all the data received in binary format. This completes the operation of grouping together a set of corresponding MPI_Isend and MPI_Irecv calls and storing their data for debugging purposes. Figure 37 shows the C code with the embedded SQL for getting the GID for a message received by MPI_Irecv. The unique integer id for the function MPI_Isend is 125.

```
sprintf (fgRequestString,
         " SELECT min(msggroupid) FROM loginfo \
           WHERE mpifuncid = 125 \
           AND ranksender = %d AND myrank = %d \
           AND rankreceiver = %d AND opdone = %d;  ",
         rankSender, rankSender, rankReceiver, FALSE);
```

Figure 37: C code with embedded SQL used to retrieve the GID for MPI_Recv.

Group Communication Routines

In LAM-MPI all group communications are blocking. It is very important that we clearly understand the meaning of blocking in LAM-MPI. Blocking means that the program execution at the node is blocked only for the time when the data is copied from the application buffer of the program to the system buffer. Once that is done, the MPI function returns and the program control goes to the next statement in the program. Blocking does not mean that the program control (at the process initiating a group routine) will wait till all the other related processes have executed their corresponding group MPI functions and returned or acknowledged.

MPI_Barrier

MPI_Barrier is a group communication routine. In this routine LAM-MPI implementation enforces synchronization. Hence if an MPI_Barrier call returns it implies that all the other nodes executing the program have reached (not necessarily completed) their corresponding MPI_Barrier calls. Figure 38 shows the details of the function MPI_Barrier [MES].

Function	MPI_Barrier is a group communication routine.
Signature	`int MPI_Barrier(MPI_Comm comm)`
Arguments	comm - is a communicator (handle) (IN)
Work	Creates barrier synchronization in a group. Each task, when reaching the MPI_Barrier call, blocks until all tasks in group reach the same MPI_Barrier call.

Figure 38: Details of function MPI_Barrier.

When an MPI_Barrier call is encountered, its wrapper inserts a row for it with a unique message group id in the relation *loginfo*. We must identify the correct group of MPI_Barrier calls to which this particular MPI_Barrier call belongs and update its message group id with that of the group's message group id, GID, thus creating a group. If no such group exists then this is the first instance of an MPI_Barrier call, hence its message group id is the GID. All message group ids are increasing in order with time. For an MPI_Barrier call the `opDone` flag, which indicates whether an operation is complete or not, is updated in the relation *loginfo* only when all processes of the group communication return successfully.

To find the unique GID, to which the current MPI_Barrier call belongs, we first select the set of distinct message group ids from the relation *loginfo* where the `mpifuncsigid` is equal to MPI_Barrier's unique integer id of 18 and `opDone` is `false (0)`. But this is not sufficient for an MPI program having more than one call to

MPI_Barrier. In that case, the above result set may contain message group id of a previous set of MPI_Barrier calls whose operation has not yet completed. This is possible because a call to MPI_Barrier returns when all processes *reach* the MPI_Barrier call in their program execution; it does not wait for them to complete the calls. Thus some processes might have completed the first call to MPI_Barrier, reached the second call to MPI_Barrier and are currently blocked while others might still be executing their first MPI_Barrier call. Further, since we assign GID at the start of the wrapper function, the previous set of MPI_Barrier calls must have been already assigned a GID. But since all the processes of the previous group have not yet completed their MPI_Barrier calls, the `opDone` flag for the group will be `false (0)` and thus they will be included in the above result set.

We have to exclude the message group id of the previous set of MPI_Barrier calls, if any, from above obtained results. We create this previous set by collecting rows in relation *loginfo* where the `mpifuncsigid` is equal to MPI_Barrier's unique integer id of 18, `opDone` is `false (0)` and `mpifuncreturnmsg` is `not null` (since at least one of the calls has returned hence a return message is present). Then we extract our desired set of message group ids by excluding the above obtained set from the original result set using a set difference. After that we choose the minimum message group id from the set difference to get the GID for the current group. The tuple in the relation *loginfo* corresponding to the current MPI_Barrier call is updated with the GID obtained above.

An example would illustrate the above explained scenario. Let the processes be r_0, r_1, r_2 where r_0 is the root, that is, calling process. Let us assume that there are two

MPI_Barrier calls in the program ba_1 and ba_2 and we have a scenario as demonstrated in Figure 39. We are at a stage where process r_1 is trying to assign GID for ba_2. The result set form our first query as explained in previous paragraphs, will contain the message group ids {1, 4, 5} while the second query will return the set {1}. The set difference of the above two sets is {4,5} and the minimum element in the set difference is {4}. Thus r_1 gets the GID of 4.

callNo	GID	myRank	root	opDone	returnCode	
ba_1	1	r_0	r_0	false	Success	
ba_1	1	r_1	r_0	false	Success	
ba_1	1	r_2	r_0	false		
ba_2	4	r_0	r_0	false		
ba_2	5	r_1	r_0	false		← We are here, r_1 has to get GID for ba_2

Figure 39: Example of a scenario with multiple MPI_Barrier calls.

Figure 40 shows the C code with the embedded SQL for the query to get the GID for MPI_Barrier. After getting the GID, the actual call to MPI_Barrier is made; when the call returns all the relevant columns in the corresponding row, in relation *loginfo*, are updated. Next we have to check whether all the processes in the communication group have completed executing their corresponding MPI_Barrier calls. We do so by counting the number of rows in relation *loginfo* where msgGroupId is same as the GID obtained above and `mpifuncreturncode = MPI_SUCCESS`. If this count equals the number of processes in `comm` (communication object) then all the processes have returned successfully from the corresponding MPI_Barrier calls. Consequently, we update `opDone` to `true(1)` for all processes with the above GID in the relation *loginfo*. Last but not the least a row is inserted in the table *MPI_Barrier* with debugging data from this call.

```
sprintf (fgRequestString,
         "SELECT min(msgGroupId) FROM (\
          (\
            SELECT DISTINCT msgGroupId FROM loginfo \
            WHERE mpifuncid = 18 \
            AND opDone = %d \
           )\
           EXCEPT \
           (\
             SELECT DISTINCT msgGroupId FROM loginfo \
             WHERE mpifuncid = 18 \
             AND opDone = %d AND mpifuncreturnmsg IS NOT NULL \
            )\
           )\
           AS set1",
         FALSE, FALSE);
```

Figure 40: C code with embedded SQL used to find the GID for MPI_Barrier.

MPI_Bcast

In MPI_Bcast the process which broadcasts is the root. A call to MPI_Bcast at the root can return as soon as data is copied from the application buffer to a system buffer. Other processes which receive the broadcast will be blocked till the data is copied from the system buffer to their application buffer. A scenario may arise where several MPI_Bcast calls might have been executed at the root while other processes were still blocked on their first MPI_Bcast call. Further, one or more processes might have also executed several (less than or equal to the number of calls made by the root) MPI_Bcast calls (from the same root), while the rest might not have completed their first call to MPI_Bcast. Hence, if an MPI_Bcast at a non-root process returns then the root must have at least entered MPI_Bcast, and sent data to its system buffer. The details of the function MPI_Bcast [MES] are shown in Figure 41.

In the wrapper for MPI_Bcast, a tuple is inserted in the relation *loginfo* with a unique message group id, and then a call is made to MPI_Bcast. If the call returns successfully updates are made to the tuple with above message group id in the relation *loginfo* with the

Function	MPI_Bcast is a group communication routine.
Signature	`int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
Arguments	buffer - is the starting address of the buffer (choice) (INOUT) count - is the number of elements in the buffer (integer) (IN) datatype - is the datatype of the buffer elements (handle) (IN) root - is the rank of the root task (integer) (IN) comm - is the communicator (handle) (IN)
Work	Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.

Figure 41: Details of the function MPI_Bcast.

return codes and messages. Next, if the current process is not the root, we must locate the message group id of the root that initiated this broadcast message. We then update the tuple for this call in the relation *loginfo* with the message group id of the root, say GID, thus creating the broadcast group.

To find the unique GID to which the current MPI_Barrier call belongs, we first select the set of distinct message group ids from the relation *loginfo* where the `mpifuncsigid` is equal to the unique integer id 19 assigned for the function MPI_Bcast, `myRank` is that of root and `opDone` is `false (0)`. This result set will contain the GIDs of all MPI_Bcast group operations that have yet not completed.

But this is not sufficient, since the above result set will contain GIDs of previous incomplete operations of MPI_Bcast groups for which the current process might have already completed its call and assignment of correct GID. So we have to exclude this set of message group ids from above obtained results. We create this set by collecting message group ids from rows in the relation *loginfo* where the `mpifuncsigid` is equal to the unique integer id 19 assigned for the function MPI_Bcast, `myRank` is that of the current process, `rankSender` is that of root and `opDone` is `false (0)`. We obtain the desired set of message group ids by excluding the above set from the original result set. We then choose the minimum message group id from the set difference which gives

us the unique GID for the group. The message group id of the tuple inserted into the relation *loginfo* for the current MPI_Bcast call is updated with the GID obtained above.

We demonstrate the above explained process with an example. Let the processes have ranks r_0, r_1, r_2 where r_0 is the root. Assume that there are four MPI_Bcast calls in the program, bc_1, bc_2, bc_3 and bc_4 respectively and we are in the scenario as demonstrated in Figure 42.

callNo	GID	myRank	root	opDone	returnCode	
bc_1	1	r_0	r_0	true	success	
bc_1	1	r_1	r_0	true	success	
bc_1	1	r2	r_0	true	success	
bc_2	4	r_0	r_0	false	success	
Bc_2	4	r_1	r_0	false	success	
bc_3	6	r_0	r_0	false	success	
bc_3	7	r_1	r_0	false	success	← We are here, r_1 has to get GID for bc_3
bc_4	8	r_0	r_0	false	success	

Figure 42: Example of a scenario with multiple MPI_Bcast calls.

We are at a stage where process r_1 is trying to obtain its GID for the call bc_3. We assume that prior to this point, r_1 has successfully assigned correct GIDs for MPI_Bcast calls bc_1 and bc_2 as shown in Figure 42. The first query fetches the result set {4, 6, 8} which contains GIDs for all incomplete MPI_Bcast calls. In this set, the message group id of 4 has already being successfully assigned by r_1 to bc_2. Hence there is a need to eliminate the GIDs already assigned by the current process.

Our second query fetches the set {4, 7}. When we do a set difference of the above result sets {4, 6, 8} and {4, 7}, we get the set {6, 8}. As mentioned earlier, in LAM-MPI order of calls is preserved. In IDLI all message group ids are assigned in increasing order

with time. So the minimum of the set {6, 8}, which is {6}, is the GID for the call bc_3 for current process of rank r_1.

To summarize, to get GID for an MPI_Bcast for process r_i where i = [1, 2, … , (no of processes) -1] we select the minimum message group id from the set difference of {all unfinished root GIDs} and {all unfinished root GIDs for r_i}. The C code with the embedded SQL statement used to find GID is shown in Figure 43.

```
sprintf (fgRequestString,
         "SELECT min(msggroupid) FROM ( \
          ( \
             SELECT DISTINCT msgGroupId FROM loginfo \
             WHERE mpifuncid =  19 AND myrank = %d AND opdone = %d \
           ) \
           EXCEPT \
           ( \
             SELECT DISTINCT msgGroupId FROM loginfo
             WHERE mpifuncid = 19
             AND myrank = %d AND rankSender = %d  AND opdone = %d \
           ) )  AS set1",\
        root, FALSE, gMyRank, root, FALSE);
```

Figure 43: C code with embedded SQL for extracting the GID for MPI_Bcast.

Next, we check whether all the processes in the communication group have completed executing their corresponding MPI_Bcast calls. We do so by counting the number of rows in relation *loginfo* where `msgGroupId` is same as the message GID obtained above and `mpifuncreturncode = MPI_SUCCESS`. If this count equals the number of processes in `comm` (communication object) then the MPI_Bcast calls at all processes have returned successfully. Consequently, we update `opDone` to `true (1)` for all processes with the above GID in the relation *loginfo*. According to our norm, a row is inserted in the table *MPI_Bcast* with debugging data from this call.

MPI_Gather

For a non-root process MPI_Gather returns as soon as data is copied from the application buffer to a system buffer. However, when MPI_Gather at the root returns, we can conclude that all other processes must have completed their corresponding MPI_Gather calls (at least data must have been copied from the application buffer at each process to a system buffer). Hence, in an application with multiple MPI_Gather calls, there can be a scenario where some processes might have completed multiple MPI_Gather calls while the root may be still blocked at its first MPI_Gather call. The details of the function MPI_Gather [MES] is shown in Figure 44.

In the wrapper for MPI_Gather, a tuple is inserted in the relation *loginfo* with a unique message group id, and then a call is made to MPI_Gather. If the call returns successfully updates are made in the tuple with the above message group id, in the relation *loginfo*, with the return codes and messages.

Function	MPI_Gather is a group communication routine.
Signature	`int MPI_Gather(void* sendbuf,int sendcount,MPI_Datatype sendtype, void* recvbuf,int recvcount,MPI_Datatype recvtype,int root, MPI_Comm comm)`
Arguments	sendbuf - is the starting address of the send buffer (choice) (IN) sendcount - is the number of elements in the send buffer (integer) (IN) sendtype - is the datatype of the send buffer elements (handle) (IN) recvbuf - is the address of receive buffer (choice, significant only at root) (OUT) recvcount - is the number of elements for any single receive (integer, significant only at root) (IN) recvtype - is the datatype of the receive buffer elements (handle, significant only at root) (IN) root - is the rank of the receiving task (integer) (IN) comm - is the communicator (handle) (IN)
Work	Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

Figure 44: Details of the function MPI_Gather.

If the current node placing a call to MPI_Gather is not the root then a tuple is inserted in the table *MPI_Gather* with data for this call but there will be no binary data from

`recvbuf.` This data will be put only in the tuple inserted by the root. Since the nodes can race ahead with MPI_Gather calls as explained above, we synchronize with an MPI_Barrier call. This ensures that all inserts into relation *MPI_Gather* by all processes in the network are complete before the root updates the message group ids in the appropriate tuples with its own message group id, which is the GID in this case.

If the node is the root, then a row is inserted in the relation *MPI_Gather* with the data in `recvbuf`. Next the root has to ensure that all inserts in *MPI_Gather* by other processes have been completed so it synchronizes with a call to MPI_Barrier (in correspondence to the MPI_Barrier call placed by the non-root nodes after their inserts in table *MPI_Gather*). When MPI_Barrier at the root node returns we now know that all the non-root processes have returned from their MPI_Gather calls and might have at most invoked one more MPI_Gather call with the same root. They cannot invoke more than one MPI_Gather call because they need to synchronize with the root through the MPI_Barrier call. As a result, the root now updates the message group ids of all the tuples (for all the processes for this MPI_Gather call) with its message group id which is the GID. Figure 45 shows the embedded SQL for the above update.

```
sprintf (fgSQLString,
         "UPDATE loginfo SET msggroupid = %lld \
          FROM ( \
                   SELECT myRank as rank, min(msggroupid) as id \
                   FROM loginfo \
                   WHERE rankReceiver = %d AND opDone = %d \
                         AND mpifuncid = 77 \
                   GROUP BY myRank \
                ) AS temp \
          WHERE msggroupid = temp.id AND myRank = temp.rank",
          groupMessageId, root, FALSE);
```

Figure 45: C code with embedded SQL for updating the relation *loginfo* with the GID.

Next, we check whether all the processes in the communication group have completed executing their corresponding MPI_Gather calls. We do so by counting the number of rows in relation *loginfo* where `msgGroupId` is same as the message GID obtained above and `mpifuncreturncode = MPI_SUCCESS`. If this count equals the number of nodes in `comm` (communication object) then the MPI_Gather calls at all processes have returned successfully. Consequently root updates `opDone to true` (1) for all nodes with the above GID in the relation *loginfo*.

MPI_Allgather

The details of the function MPI_Allgather [MES] are shown in Figure 46. If any process returns from MPI_Allgather, all other processes must have started MPI_Allgather and sent data to their system buffers. This also implies all previous MPI_Allgather calls must have completed.

Function	MPI_Allgather is a group communication routine.
Signature	`int MPI_Allgather(void* sendbuf,int sendcount, MPI_Datatype sendtype, void*  recvbuf,int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
Arguments	sendbuf - is the starting address of the send buffer (choice) (IN) sendcount - is the number of elements in the send buffer (integer) (IN) sendtype - is the datatype of the send buffer elements (handle) (IN) recvbuf - is the address of the receive buffer (choice) (OUT) recvcount - is the number of elements received from any task (integer) (IN) recvtype - is the datatype of the receive buffer elements (handle) (IN) comm - is the communicator (handle) (IN)
Work	Gathers individual messages from each task in comm and distributes the resulting message to each task.

Figure 46: Details of the function MPI_Allgather.

In the wrapper for MPI_Allgather, a tuple is inserted into the relation *loginfo* with a unique message group id, and then a call is made to MPI_Allgather. If the call returns successfully updates are made in the tuple with above message group id, in the relation

loginfo, with the return codes and messages. Now we must identify the common message group id, GID, for this MPI_Allgather call. We do so by selecting all those message group ids from the relation *loginfo* where `mpifuncsigid` is equal to the unique integer id 8 assigned for the function MPI_Allgather, and where `opDone` is `false (0)`. Since LAM_MPI preserves the order in which function calls are made and in IDLI all message group ids are assigned in increasing order with time, we select the minimum message group id from above result set as the GID. We then access the relation *loginfo* and update the tuple for MPI_Allgather for this process with the GID obtained above. Next, we check whether all the processes in the communication group have completed executing their corresponding MPI_Allgather calls. We do so by counting the number of rows in the relation *loginfo* where `msgGroupId` is same as the message GID obtained above and where `mpifuncreturncode = MPI_SUCCESS`. If this count equals the number of processes in `comm` (communication object) then the MPI_Allgather calls at all processes have completed successfully. In that case the `opDone` flag is set to `true (1)` for all tuples with above GID in the relation *loginfo.*

Preliminary Routine: MPI_Finalize

MPI_Finalize should be called exactly once in an MPI program. The details of the function MPI_Finalize [MES] is shown in Figure 47.

Function	MPI_Finalize is a Preliminary or Environment routine.
Signature	`int MPI_Finalize(void)`
Arguments	None
Work	Terminates all MPI processing. Although MPI_FINALIZE terminates MPI processing, it does not terminate the task. It is possible to continue with non-MPI processing after calling MPI_FINALIZE, but no other MPI calls (including MPI_INIT) can be made.

Figure 47: Details of the function MPI_Finalize.

In the wrapper for MPI_Finalize, a tuple is inserted in the relation *loginfo* with a unique message group id, and then a call is made to MPI_Finalize. If the call returns successfully updates are made in the tuple with above message group id, in the relation *loginfo,* with the return codes, messages and `opDone` is set to `true (1)`. We then check whether all the processes have executed MPI_Finalize. We do so by counting the number of rows returned from the relation *loginfo* where `mpifuncsigid` is equal to the unique integer id 74 assigned to the function MPI_Finalize, and where `opDone` is `true (1)`. If number of MPI_Finalize calls that have returned successfully is equal to the number of processes in the `comm` (communication handle) we can then conclude that all MPI calls in the program have completed execution. Consequently we close the connection to the user database and a connection to system database is opened. In the relation *userinfo* updates are done for current user whose `sessionActive` column is set to `false`. After that the connection to the system database is closed and memory is freed for subsequent operations.

Middle Layer: Wrappers for other MPI Routines

Wrapper functions for MPI_Gatherv and MPI_Reduce are implemented along the lines of the wrapper for MPI_Gather as explained above. MPI_Scatter's wrapper follows the implementation details of MPI_Bcast's wrapper. The wrappers for MPI_Allgatherv, MPI_Alltoall, MPI_Reduce_scatter, MPI_Allreduce are based on the implementation of the wrapper for MPI_Allgather. Wrappers for the preliminary routines MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Wtime, MPI_Wait, MPI_Test, MPI_Probe, MPI_Iprobe, MPI_Send, MPI_Recv follow the general design of our wrappers and are straight forward to implement.

Front End: User Interface for Query Manager and Replay

IDLI has an intuitive shell user interface which has been specifically written to provide features like command completion, command history, and robust error checking. Each command is checked for errors and the user is informed of the specific error. This helps the user correct the problem quickly rather than having to figure out the error himself. When IDLI starts, its parent menu of commands is shown in Figure 48.

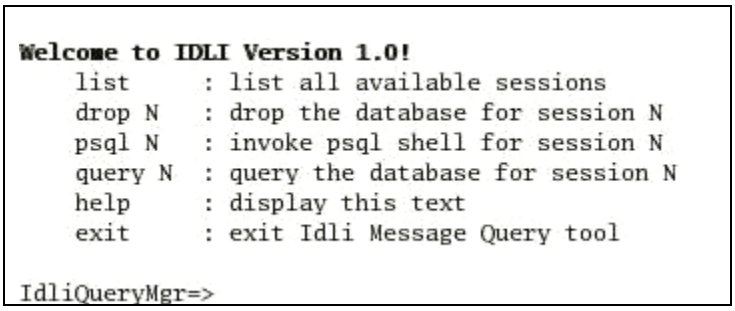


Figure 48: First menu of commands for Message Query Manager.

In this section we discuss the implementation details of the commands. The functional features of each command have already been explained in Chapter 4. When a user starts a session in IDLI, we obtain the name of the current user. We then open a connection to the system database for IDLI so that we can query the system table *userinfo* which stores data for all users and their multiple sessions.

Commands: *list N, drop N, psql N, help* and *exit*

The command *list* is used to list all the user databases storing information for programs debugged with IDLI up to a maximum of five sessions for each user. We do so by selecting all tuples from the relation *userinfo* where `userName` is that of current user, `dbName` is not `null` and then sort the result set in ascending order with respect to `sessionTime`. The C code with the embedded query for *list* is shown in Figure 49.

```
sprintf (fgRequestString,
          "SELECT sessionNo, dbName, sessionTime \
           FROM  userinfo \
           WHERE userName = '%s' AND dbName IS NOT NULL \
           ORDER BY sessionTime ASC",
         gUser);
```

Figure 49: C code with embedded SQL query for the command *list*.

The command *drop N* deletes the user database created in session number *N*. After the database is deleted, we update the corresponding tuple for current user and session, in the relation *userinfo*, by setting `dbName` to `null`, `dbCreated` to `false`, and `sessionTime` with current time.

The command psql N invokes a PostgreSQL shell for the user database of session *N*. Using this shell a user can write customized SQLs to query tables for retrieval of specific data. The text shown in Figure 48 can be printed by a user using the command *help*. The command *exit* quits the debugger after closing connection to system database.

Command: Query N

The command *query N*, where *N* is a session number, takes the user to the built-in-queries shown in Figure 50. These queries can be used for querying the user database for the program already executed in debug mode for session *N*. The built-in queries with its various options provide different sets of data abstraction.

A user can write customized SQL queries, without having to go back to the parent menu shown in Figure 48, using the command *psql* which invokes a PostgreSQL shell for the current user database. He has access to all the tables used for storing the debugging information in the user database. However, a user does not have access to the system database.

```
                              IDLI QUERY MANAGER
*******************************************************************************
    dump N        : dump messages sorted by one of the following ways, N is the number of choice
                    1 : log time
                    2 : message group id
                    3 : rank of a process
                    4 : file name
                    5 : file and line number
                    6 : MPI function name
    locategroup N : locate group communication messages by one of the following ways, N is the number of choice
                    1 : file
                    2 : line
                    3 : line of file
                    4 : MPI function name
    locatep2p N   : locate point-to-point messages by one of the following ways, N is the number of choice
                    1 : between 2 processes
                    2 : between 2 files of 2 processes
                    3 : between 2 lines of 2 processes
                    4 : between 2 lines of 2 files of 2 processes
                    5 : between 2 files
                    6 : between 2 lines
                    7 : between 2 lines of 2 files
                    8 : MPI function name
    psql          : invoke psql shell for current database
    replay N      : replay process execution of rank N in gdb
    status N      : display messages sorted by status of MPI call for entire operation
                    0 : incomplete operations
                    1 : successful operations
    trace N       : trace all messages of a particular message group, N is the message group id
    help          : display this text
    exit          : exit query mode for this session

amma_2=>
```

Figure 50: The Query Manager's menu of queries.

A user can replay the execution of an application at a particular process by using the command *replay N*, where *N* is the rank of the process. She can choose to replay in a debugger of her own choice, for example, GDB (GNU DeBugger) [GNU], or DDD (Data Display Debugger) [DAT]. The *help* and *exit* commands to enable a user to print the Query Manager's menu shown in Figure 50 and exit respectively.

In the following sections we discuss the implementation details of the queries *dump N, locategroup N, locatep2p N, status N,* and *trace N.* We also devote a section to the

command *replay N*. For each command we describe the type of debugging information being displayed, explain some of the complex SQL queries used to retrieve the data and demonstrate C code with embedded queries whenever necessary.

Data displayed by each built-in Query

The header row for each message displays the following:

- message group id as `msgid`,
- the source from which the message was sent as `src`,
- the destination to which the message was sent as `dest`,
- the rank of the process as `myRank`,
- the name of the MPI function as `mpiFuncName`,
- the result of the operation, as `ok` (0-incomplete, 1-complete),
- the name of the file and line number from which MPI function was called as `fileName` and `line` respectively, and
- the time when the information was inserted in the relation *logInfo* as `logTime`.

Each built-in query fetches the above information and the criteria of sorting used in the query is highlighted by coloring corresponding sorted columns of data in each row with a different colors.

Query: *dump N*

The query *dump N*, where *N* denotes a column to sort by, lists details of all MPI functions executed during an application's run. *Dump N* can sort the above information in various orders and the column by which it is sorted is displayed in a different color for

ease of identifying the sorting criteria. The different sorting criteria chosen by selecting different values of *N* are as follows:

- *N* = 1: sort by `logTime`,
- *N* = 2: sort by `msgid`,
- *N* = 3: sort by `myRank`,
- *N* = 4: sort by `filename`,
- *N* = 5: sort by `filename` & `line`,
- *N* = 6: sort by `mpiFuncName`.

The C code with the embedded SQL used for the command *dump 1* is shown in Figure 51. The SQL queries for the command *dump* with other values of *N* are very similar to the one shown in Figure 51. In those queries the parameter for the clause `order by` is changed to the criteria selected by the choice of *N*.

```
sprintf (fgRequestString,
         "SELECT msgGroupId, rankSender, rankReceiver, \
               myRank, mpiFuncName, opDone, fileName, lineNo, \
               logTime, mpiFuncReturnMsg \
          FROM logInfo \
          NATURAL INNER  JOIN mpiFuncSigId
          ORDER BY logTime ASC"
         );
```

Figure 51: C code with embedded SQL for the command *dump 1*.

Query: *locategroup N*

The command *locategroup N*, where *N* indicates the way to locate the messages, locates all messages exchanged only by group communication routines during the execution of an application. The different criteria by which a message can be located by selecting different values of *N* are as follows:

- *N = 1: locates messages exchanged between two files,*
- *N = 2: locates messages exchanged between two line numbers,*
- *N = 3: locates messages exchanged between two line numbers of two files,*
- *N = 4: locates messages exchanged between two MPI functions.*

The C code with the embedded SQL used for the command *locategroup 1* is shown in Figure 52. The SQL queries for the command *locategroup* with other values of *N* are very similar to the one shown in Figure 52. In those queries the parameter for the inner selection clause `where` is changed according to the criteria selected by the choice of *N*.

```
sprintf (fgRequestString,
         "SELECT msgGroupId, rankSender, rankReceiver, myRank,\
                 mpiFuncName, opDone, fileName, lineNo, \
                 logTime, mpiFuncReturnMsg \
          FROM logInfo \
          NATURAL INNER  JOIN mpiFuncSigId \
          WHERE msgGroupId IN  (\
                SELECT DISTINCT msgGroupId FROM logInfo \
                WHERE fileName = '%s' AND mpifuncid IN (\
                      SELECT mpifuncid FROM mpifuncsigid
                      WHERE mpifuncname IN (\
               'MPI_Barrier' , 'MPI_Bcast' , 'MPI_Gather' ,\
               'MPI_Gatherv' , 'MPI_Allgather' , 'MPI_Allgatherv' ,\
               'MPI_Alltoall' , 'MPI_Allreduce' , 'MPI_Reduce' ,\
               'MPI_Scatter' , 'MPI_Reduce_scatter' ) \
                ) \
          ) ORDER BY logTime ASC", \
         file1);
```

Figure 52: C code with embedded SQL for the command *locategroup* 1.

Query: locatep2p N

The command *locatep2p N*, where *N* denotes the way to locate the messages, locates all messages exchanged only by point-to-point communication routines during the execution of a program. The different criteria by which a message can be located by selecting different values of *N* are as follows:

- *N=1: locates messages exchanged between two processes*
- *N=2: locates messages exchanged between two files of two processes*
- *N=3: locates messages exchanged between two lines of two processes*
- *N=4: locates messages exchanged between two lines of two files of two processes*
- *N=5: locates messages exchanged between two filess*
- *N=6: locates messages exchanged between two lines*
- *N=7: locates messages exchanged between two lines of two files*
- *N=8: locates messages by MPI function names.*

The C code with the embedded SQL used for the command *locatep2p 1* is shown in Figure 53. The SQL queries for the command *locatep2p* with other values of *N* are very similar to the one shown in Figure 53. In those queries the parameter for the selection clause `where` is changed according to the criteria selected by the choice of *N*.

```
sprintf (fgRequestString,
         "SELECT msgGroupId, rankSender, rankReceiver,\
                 myRank, mpiFuncName, opDone, fileName, \
                 lineNo, logTime, mpiFuncReturnMsg FROM logInfo \
             NATURAL INNER  JOIN mpiFuncSigId WHERE ( \
                  (rankSender = %d AND rankReceiver = %d) OR \
                  (rankSender = %d AND rankReceiver = %d) \
          ) \ ORDER BY logTime ASC", rank1, rank2, rank2, rank1);
```

Figure 53: C code with embedded SQL for the command *locatep2p* 1.

Query: *status N*

The command *status N*, where *N = 0* denotes incomplete operations and *N = 1* denotes complete operations, is used to list details of executions of all MPI routines based on the status of their operation, that is, complete or incomplete. For example, *status 1* lists details of all executions of MPI routines where the entire operation terminated

successfully. The command *status 0* lists details of all executions of MPI routines where the operations terminated with an error. Operation does not refer to a particular MPI call, it implies all related MPI calls required to complete a communication. For example, MPI_Send and MPI_Recv together form an operation. For group routines operation implies the execution of the corresponding MPI call across all processes of the communication handle or object `comm`. If we execute an MPI_Bcast then 'operation' means execution of MPI_Bcast by all processes of the group. In such a group if one process fails then the operation is unsuccessful, though there might be successful execution of MPI_Bcast calls by all other processes. The SQL used to implement *status* is shown in Figure 54.

```
sprintf (fgRequestString,
         "SELECT msgGroupId, rankSender, rankReceiver,\
                 myRank, mpiFuncName, opDone, fileName,\
                 lineNo, logTime, mpiFuncReturnMsg
          FROM logInfo \
          NATURAL INNER  JOIN mpiFuncSigId \
          WHERE opDone = %d ORDER BY msgGroupId ASC",
        choice);
```

Figure 54: C code with embedded SQL for the command *status N*.

Query: *trace N*

The command *trace N* displays all the messages belonging to the same message group id (denoted by *N*). In a group communication, this command is very useful in viewing messages exchanged by processes, as segregated groups. For example, if we executed a call to MPI_Scatter whose message group id is *N*, then *trace N* will list tuples with information for this specific MPI_Scatter executed by all processes of the communication group. This command enables us to see errors, if any, and the processes on which they have occurred quickly since we trace specific MPI calls instead of going

through a list of all MPI_Scatter calls in an application. The SQL used to implement *trace N* is shown in Figure 55.

```
sprintf (fgRequestString,
         "SELECT msgGroupId, rankSender, rankReceiver,\
                 myRank, mpiFuncName, opDone, fileName, \
                 lineNo, logTime, mpiFuncReturnMsg
          FROM logInfo \
          NATURAL INNER  JOIN mpiFuncSigId \
          WHERE msgGroupId  = %d
          ORDER BY msgGroupId ASC",
        Id);
```

Figure 55: C code with embedded SQL for the command *trace N*.

Query: *replay N*

The command *replay N* replays the execution of the program at a process of rank N without actually running LAM-MPI. The data for LAM-MPI functions, which have already executed when the program was run in debug mode, is fetched from the tables in user database. The user can locate the process which had erred using the Query Manager and then replay the execution for that particular process using a sequential debugger of choice. This will enable the user to see the exact data returned by the MPI function calls and aid in mapping the error back to the source. A unique feature of the command *replay N* is that it allows the user to simultaneously replay multiple processes of distinct ranks in separate windows which may be running on the same physical node. Secure Shell 2 Protocol (SSH2) over TCP/IP is used to open a connection to the node on which replay is to be executed. IDLI's replay automatically locates the node for the selected process of rank *N*.

The information about the number of replays running simultaneously at different nodes is stored in the relation *replayInfo.* Information is stored about each process in different columns in the following manner:

- a process's rank as `myRank`,
- the ip address of the node as `hostIpAddress`,
- the user database as `dbName`,
- the current state of replay as `state` whose permissible values are `replay_not_started` or `replay_running` or `replay_`done,
- the process id of debugger as `gdbPid`, and
- the time when replay was initiated as `logTime`.

Initially there would be no entries in the *replayInfo* table. Each time the user chooses to *replay* a particular process of rank *N* from the Query Manager, an entry is entered into the *replayInfo* table. The entry is deleted when its corresponding replay is terminated or shutdown. In the wrapper function for IDLI_MPI_Init (in the native C library of IDLI), it is first checked if there are any entries in the *replayInfo* table. If any entry is found IDLI runs in replay mode else it runs in normal debug mode. A global flag is set to indicate the mode.

At the beginning of each wrapper function in the native C library for IDLI, the global flag is checked to determine the mode of processing for the MPI function calls. During replay mode, stored data is fetched from a previous execution of the application in debug mode (with IDLI). However, to the user program it appears as if the MPI function is actually executing. Since no MPI calls are actually executed through the MPI interface, no time spent on message blocking or synchronization. Consequently, replay is extremely fast as it performs database reads only. Along with sequential debugging, the data retrieved for the MPI calls can be examined in the sequential debugger selected by the

user for replay. When a user quits replay, that is, when replay is over at the relevant nodes, all corresponding tuples for those nodes are deleted from the relation *replayInfo.*

For a replay, the user specifies the rank of the process and the preferred sequential debugger at the interactive prompt in the shell of Query Manager. The ip address and hostname of the node on which the process of chosen rank got executed in its normal debug run is retrieved from the table *logInfo* in the user database. A global array of flags indicates if a replay session for a given rank is already running or not. If there is a replay already running for the process of rank *N*, then an error message is generated. Otherwise the appropriate flag denoting that there is a replay for the process of rank *N* is set to true. This prevents the user from simultaneously running more than one replay for a process of the same rank.

If the user is allowed to quit the Query Manager while there are active replay sessions, it may leave the database in an inconsistent state. This is because the corresponding tuples for the active replays will not be deleted in the relation *replayInfo.* When a user tries to quit the Query Manager by entering the command *exit,* global array of flags is checked to determine if any active replay sessions existed. If so, an error message is displayed and the user is prevented from quitting by returning the control the prompt of the Query Manager.

When our checks reveal that there are no active replay sessions for a process of rank *N,* then a child process is forked. This process handles the tasks of invoking the replay at the remote machine and exiting gracefully at the end. The parent process simply returns to the interactive command prompt in the Query Manager thus enabling simultaneous replay and query processing.

Summary

In this chapter we explored the implementation details of our message debugger, IDLI, in three distinct parts. The first part explains the design of the SQL databases at the backend and the relations used to store debugging information and meta data. The design of the native C library, its wrapper functions for MPI routines, their algorithms and examples of different challenges that were encountered for the group routines are explained in the second part. The user interface at the front-end, IDLI's Query Manager with all its built-in queries and Replay are explained in the third part. This part provides information about the features of each query, its implementation details and C code snippets of SQL queries as and when deemed necessary.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we have described IDLI, a message debugger for parallel programs running in a distributed environment using LAM-MPI. Our message debugger is based on the concepts of multilevel debugging [PED03] which is a bottom-up parallel debugging approach. We have designed IDLI after a careful analysis of the limitations present in the usability [PAN03] of current parallel debuggers. We believe that IDLI provides effective sequential and message level debugging along with precise mappings between global and local contexts. Hence it provides sufficient granularity like its predecessors yet avoids many of their inherent flaws.

Improvements upon current parallel debuggers

The guiding philosophy behind the development of IDLI was to implement a message debugger for the LAM-MPI environment based on the principles of multi-level debugging [PED03]. This concept was designed to avoid several flaws prevalent in current parallel debugging tools most of which employ a top-down debugging approach. Also, to improve upon many of the shortcomings, as revealed by our extensive survey of the current parallel debuggers, we intended to develop an innovative yet simple parallel message debugger from scratch. We briefly review the limiting issues of the present tools and debuggers that were identified in Chapter 2 and explain how we used IDLI and the multilevel debugging concepts to transcend them.

- Partial view of the debugging spectrum:
  - Most parallel debuggers are capable of providing debugging information specific to either an individual process, that is, local context, or the system as a whole, that is, global context. Some try to provide information for both global and local contexts but often they are not satisfactory. This is due to the fact that they are rigid with respect to a user's specific needs.
  - IDLI provides both global and local context debugging information and is flexible to a user's specific needs.
    - Global level views of all the processes are supported by the means of data abstraction through a set of built-in queries.
    - Detailed information on the local context is provided through built-in queries. They provide mappings for communication messages, exchanged between processes of distinct ranks, to their origins at the specific line number of the source code in a particular file.
    - IDLI adapts itself to a user's need for specific information through a feature which enables a user to write custom SQL queries.
    - Further, the ability to replay an application in a sequential debugger of the user's choice provides some of the finest levels of granularity.
- Information Overloading:
  - Some of the existing debugging tools generate an overwhelming amount of data which is detrimental to efficient debugging of parallel programs. Often the cause and effect of an error are separated by a long distance in a parallel program. Information overloading thus makes it increasingly difficult to extract

the essential information needed for debugging an error by tracing it back to its cause.

- Our message debugger IDLI, provides various levels of data abstraction to display relevant information according to a user's needs.
  - The built-in queries have various options that help to trim down the debugging data and retrieve requisite specific information.
  - Moreover, the user can unleash the full potential of custom SQL queries according to his debugging needs.
  - IDLI provides the flexibility to replay an application simultaneously on a chosen number of processes. This enables a user to choose as many processes as she is comfortable debugging simultaneously. Thus the user is in total control of the amount of data she wants to simultaneously view and process for debugging.

- Inability to alter or create custom views:
  - The lack of this feature in most of the graphical visualization debugging tools paralyzes a user's ability to access specific information. Thus a user's debugging ability becomes limited by the fixed views provided by the designers of the graphical tools.
  - In IDLI a user has complete freedom to take advantage of the entire range of Postgres [POS] SQL commands to virtually create any desired view of the available data in the user database.

- Lack of querying features at message level:

- During debugging of parallel programs, often users need to inspect communication messages and map them to their origins at the exact line numbers on the source code on a particular process. In present parallel debuggers, absence of features enabling queries to provide such details of the communication messages slows down the debugging process significantly. Often a user is left to trace the source of an erring MPI call to its exact location in a file.
- IDLI has a whole set of built-in queries that cater to fine granularity at the message level.
    - It has a range of built-in queries that trace a message to its origin at the line number of the source code in a particular file. They also locate the rank of the process from which the message originated. This enables a user to quickly zero in on the cause of the error. To provide detailed information on the exchanged messages IDLI does not require any modification of the user's application source code.
    - In addition, a user can modify the data exchanged by the MPI calls through messages. He can do so by replaying the application on a process with a sequential debugger of his choice. During replay the data for the MPI calls is furnished from the stored data of a previous execution of the application. The user can view the data through the sequential debugger and if he feels that it is erroneous he can modify it. Next, he can step through the application using the sequential debugger to see how the processing would have progressed with the modified data instead of the erroneous one.

Hence we see that IDLI effectively addresses most of the drawbacks of current available debugging tools. To summarize in a nutshell, IDLI provides:

- specific debugging information through sufficient levels of data abstraction,
- connects global data with local context,
- has a simple front-end user interface,
- has built-in queries for querying messages and viewing details of executions of MPI routines,
- allows custom SQL queries to be written by a user, and
- enables fast and multiple simultaneous replays on any process (at its node, that is, physical machine) with a sequential debuggers of the user's choice.

These features enable IDLI to act as a *source-level* as well as a *post-processing* debugging tool without overwhelming a user with information overloading.

Future Work

We would like to enhance the features of our message debugger IDLI to include the following:

- Protocol Conformation:

  This feature would allow a user to write specifications of the behavior of the protocol. Then using information from the actual messages, IDLI would automatically check that the messages satisfy the above specifications [PED03].

- Deadlock detection

  A feature for automatic detection of deadlocks may be provided in IDLI. An algorithm that provides automatic suggestions for a deadlock induced state given

a protocol specification can be found in [PED01]. Implementation of this algorithm would add automatic correction to automatic detection of deadlocks [TRI05].

- Support for all MPI functions implemented by LAM-MPI

  IDLI's current version supports twenty-four commonly used functions of LAM-MPI. This support can be extended to cover all the functions of the LAM-MPI interface which would make IDLI a comprehensive tool for debugging any MPI routine.

- Execution with simultaneous multiple communication worlds

  At present IDLI is designed to work with message passing in one communication world denoted by the communication handle `comm`. It would be nice to extend the functionality to multiple communication worlds which is used in many large real time applications for parallel computing.

- Real time interactive debugging

  A great feature which would make IDLI a complete multi-level debugger would be to add capabilities for real time interactive debugging of sequential code as well as MPI messages on selected processes at different nodes. This feature should display contents in the variables of a program as they are populated during its execution, aid a user to change data and synchronize MPI function calls in real time.

- Graphical visualization of global level information

  Currently IDLI provides global level views of the whole system through data abstraction. It would be a nice idea to extend this feature to a graphical display of

the entire system complete with pictures of active processes at various nodes, their executions and contents of messages exchanged [PAN98]. We can also add profilers to view the performance of the system as a whole.

- A Graphical User Interface (GUI)

  Last but not the least a worthy feature could be the addition of a GUI which supports real time debugging, manages various replay sessions, and enables querying of messages along with the above mentioned future features.

These future features would provide a complete debugger for LAM-MPI based on the concepts of multi-level debugging.

APPENDIX 1

PROGRAMS USED FOR TESTING IDLI

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
  int myid, numprocs, tag, source, destination, count, buffer;

  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  tag=1234;  source=0;  destination=1;  count=1;

  if(myid == source){
    buffer=5678;
    MPI_Send(&buffer, count, MPI_INT, destination, tag, MPI_COMM_WORLD);
    printf("processor %d  sent %d\n", myid, buffer);
  }
  if(myid == destination){
    MPI_Recv(&buffer, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
    printf("processor %d  got %d\n", myid, buffer);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 56: Listing of the C program used for testing MPI_Send and MPI_Recv.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
/*
! This program shows how to use MPI_Scatter and MPI_Reduce
! Each processor gets different data from the root processor
! by using MPI_Scatter.  The data is summed and then sent back
! to the root processor using MPI_Reduce.  The root processor
! then prints the global sum.
*/
/*  globals */
int numnodes,myid,mpi_err;
#define mpi_root 0
/* end globals  */
void init_it(int  *argc, char ***argv);
void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
     mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
     mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}

int main(int argc,char *argv[])
{
     int *myray,*send_ray,*back_ray;
     myray = send_ray = back_ray = NULL;
     int count;
     int size, i, total,gtotal;
     init_it(&argc,&argv);
/* each processor will get count elements from the root */
     count=4;
     myray=(int*)malloc(count*sizeof(int));
/* create the data to be sent on the root */
     if(myid == mpi_root){
          size=count*numnodes;
          send_ray=(int*)malloc(size*sizeof(int));
          back_ray=(int*)malloc(numnodes*sizeof(int));
          for(i=0;i<size;i++)
               send_ray[i]=i;
     }
/* send different data to each processor */
     mpi_err = MPI_Scatter(send_ray, count, MPI_INT, myray, count,
                           MPI_INT, mpi_root, MPI_COMM_WORLD);
/* each processor does a local sum */
     total=0;
     for(i=0;i<count;i++)
          total=total+myray[i];
     printf("myid= %d total= %d\n ",myid,total);
/* send the local sums back to the root */
     mpi_err = MPI_Reduce(&total, &gtotal, 1, MPI_INT, MPI_SUM,
                          mpi_root, MPI_COMM_WORLD);
/* the root prints the global sum */
     if(myid == mpi_root){
          printf("results from all processors= %d \n ",gtotal);
     }
     mpi_err = MPI_Finalize();
     return 0;
}
```

Figure 57: Listing of the C program used for testing MPI_Scatter and MPI_Reduce.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

/*
! This program shows how to use MPI_Allgather
! Each node sends its rank and negative of rank to all.
*/

/*  globals */
int numnodes,myid,mpi_err;
#define mpi_root 0
/* end globals  */

void init_it(int  *argc, char ***argv);

void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
     mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
     mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}


int main(int argc,char *argv[])
{
     int i, sendarray[2];
     init_it(&argc,&argv);

/* each processor will send its rank and negative of rank to all */
     sendarray[0] = myid;
     sendarray[1] = -1 * myid;

     int *rbuf = (int *)malloc(numnodes*2*sizeof(int));
     MPI_Allgather( sendarray, 2, MPI_INT, rbuf, 2, MPI_INT, MPI_COMM_WORLD);

/* each processor prints what is received */
     printf("myid= %d\n ",myid);
     for(i=0;i<numnodes;i++)
     {
         int j = 2*i;
         int k = j+1;
         printf("\trbuf[%d] = %d rbuf[%d] = %d\n", j, rbuf[j], k, rbuf[k]);
     }

     mpi_err = MPI_Finalize();
     return 0;

}
```

Figure 58: Listing of the C program used for testing MPI_Allgather.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
/*
! This program shows how to use MPI_Allgatherv
! Each node sends data, rank number of times to all.
! ie rank 0 sends 0
! ie rank 1 sends 1
! ie rank 2 sends 2, 102
! ie rank 3 sends 3, 103, 203
! ie rank 4 sends 4, 104, 204, 304
! etc
*/
/*  globals */
int numnodes,myid,mpi_err;
#define mpi_root 0
/* end globals  */
void init_it(int  *argc, char ***argv);
void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
     mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
     mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}
int main(int argc,char *argv[])
{
    int i, j;
    init_it(&argc,&argv);
    int *recvcnts = (int *) malloc (numnodes * sizeof (int));
    int *displs = (int *) malloc (numnodes * sizeof (int));
    int cnt = 0;
    for (j = 0; j < numnodes; ++j)
    {
        recvcnts[j] = j;
        cnt += recvcnts[j];
    }
    int *rbuf = (int *) malloc (cnt * sizeof (int));
    displs[0] = 0;
    for (j = 1; j < numnodes; ++j)
    {
        displs[j] = displs[j-1] + recvcnts[j-1];
    }
    int *sendarray = (int *) malloc (myid * sizeof (int));
    for (j = 0; j < myid; ++j)
    {
        sendarray[j] = 100*j + myid;
    }
    MPI_Allgatherv( sendarray, myid, MPI_INT, rbuf, recvcnts, displs, MPI_INT,
MPI_COMM_WORLD);
    for(i=0;i<cnt;++i)
    {
        printf("\tmyid = %d rbuf[%d] = %d\n", myid, i, rbuf[i]);
    }
    mpi_err = MPI_Finalize();
    return 0;
}
```

Figure 59: Listing of the C program used for testing MPI_Allgatherv.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

/*
This program shows how to use MPI_Allreduce
A sum of first n non-negative integers are done
and sum of first n non-positive integers are done
*/

/*  globals */
int numnodes,myid,mpi_err;
#define mpi_root 0
/* end globals  */

void init_it(int  *argc, char ***argv);

void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
     mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
     mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}

int main(int argc,char *argv[])
{
    int sendarray[2];
    init_it(&argc,&argv);

    /* each processor will send its rank and negative of rank to root */
    /* root prints the sum */
    sendarray[0] = myid;
    sendarray[1] = -1 * myid;

    int *rbuf = (int *)malloc(numnodes*2*sizeof(int));
    MPI_Allreduce(sendarray, rbuf, 2, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    printf("\tmyid = %d rbuf[0] = %d rbuf[1] = %d\n", myid, rbuf[0], rbuf[1]);

    mpi_err = MPI_Finalize();
    return 0;

}
```

Figure 60: Listing of the C program used for testing MPI_Allreduce.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
/*
! This program shows how to use MPI_Alltoall
! rank 0 sends   0,   1,   2,   3, ...
! rank 1 sends 100, 101, 102, 103, ...
! rank 2 sends 200, 201, 202, 203, ...
! rank 3 sends 300, 301, 302, 303, ...
! rank 4 sends 400, 401, 402, 403, ...
! etc
! expected results are as follows
! rank 0 gets 0, 100, 200, 300, ...
! rank 1 gets 1, 101, 201, 301, ...
! rank 2 gets 2, 102, 202, 302, ...
! rank 3 gets 3, 103, 203, 303, ...
! rank 4 gets 4, 104, 204, 304, ...
! etc
*/

/*  globals */
int numnodes,myid,mpi_err;
#define mpi_root 0
/* end globals  */
void init_it(int  *argc, char ***argv);
void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
     mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
     mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}
int main(int argc,char *argv[])
{
     int i;
     init_it(&argc,&argv);
     int *sendbuf = (int *) malloc (sizeof (int) * numnodes);
     for(i = 0; i < numnodes; ++i)
     {
         sendbuf[i] = myid * 100 + i;
         printf("\tmyid = %d sendbuf[%d] = %d\n", myid, i, sendbuf[i]);
     }
     int *recvbuf = (int *) malloc (sizeof (int) * numnodes);
     MPI_Alltoall ( sendbuf, 1, MPI_INT, recvbuf, 1, MPI_INT, MPI_COMM_WORLD);
     for(i = 0; i < numnodes; ++i)
     {
         printf("\tmyid = %d recvbuf[%d] = %d\n", myid, i, recvbuf[i]);
     }
     mpi_err = MPI_Finalize();
     return 0;
}
```

Figure 61: Listing of the C program used for testing MPI_Alltoall.

```
#include <stdio.h>
#include "mpi.h"

int main(argc,argv)
int argc;
char *argv[];
{
  int myid, numprocs, tag, source, destination, count, buffer;

  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  tag=1234;  source=0;  destination=1;  count=1;

  if(myid == source){
    buffer=5678;
    MPI_Send(&buffer, count, MPI_INT, destination, tag, MPI_COMM_WORLD);
    printf("processor %d  sent %d\n", myid, buffer);
  }
  if(myid == destination){
    MPI_Recv(&buffer, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
    printf("processor %d  got %d\n", myid, buffer);
  }
  MPI_Barrier(MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

Figure 62: Listing of the C program used for testing MPI_Barrier.

```
/*************************************************************
This is a simple broadcast program in MPI
*************************************************************/

#include <stdio.h>
#include "mpi.h"


int main(argc,argv)
int argc;
char *argv[];
{
    int i,myid, numprocs;
    int source,count;
    int buffer[4];
   // MPI_Status status;
   // MPI_Request request;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    source=0;
    count=4;
    if(myid == source){
      for(i=0;i<count;i++)
        buffer[i]=100+i;
    }
    MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);
    for(i=0;i<count;i++)
      printf("\n buffer[%d] = %d ",i, buffer[i]);
    printf("\n");
    MPI_Finalize();
    return 0;

}
```

Figure 63: Listing of the C program used for testing MPI_Bcast.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

/*
! This program shows how to use MPI_Gather
! Each node sends its rank and negative of rank to root.
*/

/*  globals */
int numnodes,myid,mpi_err;
#define mpi_root 0
/* end globals  */

void init_it(int  *argc, char ***argv);

void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
     mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
     mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}


int main(int argc,char *argv[])
{
     int sendarray[2];
     init_it(&argc,&argv);

/* each processor will send its rank and negative of rank to root */
/* root prints what is received */
     sendarray[0] = myid;
     sendarray[1] = -1 * myid;

     if (myid == mpi_root)
     {
         int *rbuf = (int *)malloc(numnodes*2*sizeof(int));
         MPI_Gather(  sendarray,  2,  MPI_INT,  rbuf,  2,  MPI_INT,  mpi_root,
MPI_COMM_WORLD);
         printf("myid= %d\n ",myid);
         int i;
         for(i=0;i<numnodes;i++)
         {
            int j = 2*i;
            int k = j+1;
            printf("\trbuf[%d] = %d rbuf[%d] = %d\n", j, rbuf[j], k, rbuf[k]);
         }
     }
     else
     {
         MPI_Gather(  sendarray,  2,  MPI_INT,  NULL,  0,  MPI_INT,  mpi_root,
MPI_COMM_WORLD);
     }

     mpi_err = MPI_Finalize();
     return 0;

}
```

Figure 64: Listing of the C program used for testing MPI_Gather.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
/* ! This program shows how to use MPI_Gatherv
! Each node sends its rank, rank number of times to root.
! ie rank 0 sends 0 ! ie rank 1 sends 1 ! ie rank 2 sends 2, 102
! ie rank 3 sends 3, 103, 203 ! ie rank 4 sends 4, 104, 204, 304 ! etc */
int numnodes,myid,mpi_err;
#define mpi_root 0
void init_it(int  *argc, char ***argv);
void init_it(int  *argc, char ***argv) {
     mpi_err = MPI_Init(argc,argv);
     mpi_err = MPI_Comm_size( MPI_COMM_WORLD, &numnodes );
     mpi_err = MPI_Comm_rank(MPI_COMM_WORLD, &myid);
}
int main(int argc,char *argv[])
{
     int i, j;
     init_it(&argc,&argv);
/* each processor will send its rank, rank times to root */
/* root prints what is received */
     if (myid == mpi_root)
     {
         int *recvcnts = (int *) malloc (numnodes * sizeof (int));
         int *displs = (int *) malloc (numnodes * sizeof (int));
         int cnt = 0;
         for (j = 0; j < numnodes; ++j)
         {
            recvcnts[j] = j;
            cnt += recvcnts[j];
         }
         displs[0] = 0;
         for (j = 1; j < numnodes; ++j)
         {
            displs[j] = displs[j-1] + recvcnts[j-1];
         }
         int *rbuf = (int *) malloc (cnt * sizeof (int));
         int dummy = 0;
         MPI_Gatherv( &dummy, myid, MPI_INT, rbuf, recvcnts, displs, MPI_INT,
mpi_root, MPI_COMM_WORLD);
         printf("myid= %d\n ",myid);
         for(i=0;i<cnt;++i)
         {
            printf("\trbuf[%d] = %d\n", i, rbuf[i]);
         }
     }
     else
     {
        int *sendarray = (int *) malloc (myid * sizeof (int));
         for (j = 0; j < myid; ++j)
         {
            sendarray[j] = 100*j + myid;
         }
         MPI_Gatherv(  sendarray,  myid,  MPI_INT,  NULL,  0,  NULL,  MPI_INT,
mpi_root, MPI_COMM_WORLD);
     }
     mpi_err = MPI_Finalize();
     return 0;
}
```

Figure 65: Listing of the C program used for testing MPI_Gatherv.

```
#include <stdio.h>
#include "mpi.h"
#include <math.h>

/*************************************************************
This is a simple isend/ireceive program in MPI
************************************************************/

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;

    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    tag=1234;
    source=0;
    destination=1;
    count=1;
    request=MPI_REQUEST_NULL;

    if(myid == source){
      buffer=5678;
      MPI_Isend(&buffer,count,MPI_INT,destination,tag,
                MPI_COMM_WORLD,&request);
    }

    if(myid == destination){
     // MPI_Irecv(&buffer,count,MPI_INT,source,tag, MPI_COMM_WORLD,&request);
     MPI_Irecv(&buffer,count,MPI_INT,MPI_ANY_SOURCE,tag,
MPI_COMM_WORLD,&request);
    }

    MPI_Wait(&request,&status);

    if(myid == source){
         printf("processor %d  sent %d\n",myid,buffer);
    }

    if(myid == destination){
         printf("processor %d  got %d\n",myid,buffer);
    }

    MPI_Finalize();
    return 0;

}
```

Figure 66: Listing of the C program used for testing MPI_Isend and MPI_Irecv.

```
/*
This program shows how to use MPI_Reduce_scatter
This program runs on only 3 nodes.
*/

#include <mpi.h>

int main( int argc, char* argv[] )
{
    int i;
    int rank, nproc;
    int isend[6], irecv[3];
    int ircnt[3] = {1,2,3};

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &nproc );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    for(i=0; i<3; i++)
    {
        irecv[i] = 0;
    }

    for(i=0; i<6; i++)
    {
        isend[i] = i + rank * 10;
        printf("myid = %d isend[%d] = %d\n", rank, i, isend[i]);
    }

    MPI_Reduce_scatter(isend, irecv, ircnt, MPI_INT, MPI_SUM,
                       MPI_COMM_WORLD);

    for(i=0; i<3; i++)
    {
        printf("myid = %d irecv[%d] = %d\n", rank, i, irecv[i]);
    }

    MPI_Finalize();

    return 0;
}
```

Figure 67: Listing of the C program used for testing MPI_Reduce_scatter.

```
#include <stdio.h>
#include <unistd.h>
#include "mpi.h"

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    double timeVal;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    sleep (myid * 3);
    timeVal = MPI_Wtime ();

    printf ("\nmyid = %d timeVal = %50.20lf timeVal = %g timeVal = %50.20e",
myid, timeVal, timeVal, timeVal);
    printf ("\nsizeof double = %d", sizeof (double));

    MPI_Finalize();
    return 0;
}
```

Figure 68: Listing of the C program used for testing MPI_Wtime.

BIBLIOGRAPHY

[BAL04] Balle, Susanne M. & Hood, Robert T. "*Global Grid Forum User Program Development Tools Survey*", Global Grid Forum, 2004. http://www.gridforum.org/documents/GFD.33.pdf

[DAT] Data Display Debugger. http://www.gnu.org/software/ddd/

[DAT96] Date, C.J. & Darwen, H. "*A Guide to SQL Standard*", Fourth Edition, Addison Wesley, November 1996.

[DES05] DeSouza, J., Kuhn, B., De Supinski, Bronis R. "*Automated, scaleable debugging of MPI programs with Intel Message Checker*", Second International Workshop on Software Engineering for High Performance Computing System Applications, St. Louis, Missouri, USA, May 15, 2005.

[ELL03] Ellen, S., Figgins, S., & Weber, A. "*Linux In A Nut Shell*", Fourth Edition, O' Reilly Media, Inc., June 2003.

[ETN] Etnus Total View. The Most Advanced Debugger on Linux and Unix. http://www.etnus.com/TotalView/index.html

[FOS95] Foster, Ian T. "*Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*", Addison Wesley, February 1995.

[GCC] GCC Home Page – GNU Project – Free Software Foundation (FSF). http://gcc.gnu.org/

[GNU] GNU DeBugger. http://www.gnu.org/directory/gdb.html

[GRA03] Grama, A., Gupta, A., Karypis, G., & Kumar, V. "*Introduction to Parallel Computing*", Second Edition, Addison Wesley, January 2003.

[HPV] HP Visual Threads. http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,5074,00.html

[KER88] Kernighan, B. & Ritchie, D. "*The C Programming Language*", Prentice Hall, 1988.

[KON] Konchady, M. Parallel Computing Using Linux. http://www2.linuxjournal.com/article/1350

[LAD] Ladebug Debugger Manual. Compaq Tru64 UNIX Version 5.1A, June 2001.http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V51A_HTML/LADEBUG/TITLE.HTM

[LAM] LAM / MPI Parallel Computing. http://www.lam-mpi.org/

[MES] Message Passing Interface (MPI). http://www.llnl.gov/computing/tutorials/mpi/

[MES02] "*Message Passing Interface (MPI)"*, a presentation by National Partnership for Advanced Computational Infrastructure, San Diego Supercomputer Center, August 2002.

[MIC] Microsoft Visual Studio. http://msdn.microsoft.com/vstudio/

[PAN01] Pancake, Cherri. M. "*Performance Tools for Today's HPC: Are We Addressing the Right Issues?"* in Parallel Computing, Vol. 27, pp. 1403-1415, 2001.

[PAN03] Pancake, Cherri. M. "*Usability Issues in Developing Tools for the Grid – And How Visual Representations Can Help*," in Parallel Processing Letters, Vol. 13, No. 2, June 2003.

[PAN98] Pancake, Cherri. M. "*Exploiting Visualization and Direct Manipulation to Make Parallel Tools More Communicative*," in Applied Parallel Computing, ed. B. Kagstrom et al., Springer Verlag, Berlin, , pp. 400-417, 1998.

[PAN99] Pancake, Cherri. M. "*Applying Human Factors to the Design of Performance Tools"*, Proceedings of Euro-Par '99, pp. 440-457, 1999.

[PED01] Pedersen, Jan B. & Wagner, A. "*Correcting Errors in Message Passing Systems"*, High-Level Parallel Programming Models and Supportive Environments, 6th international workshop, HIPS 2001, San Francisco, LNCS 2026, Springer Verlag, April 2001.

[PED03] Pedersen, Jan B. "*Multilevel Debugging of Parallel Message Passing Systems"*, PhD Thesis, University of British Columbia, Vancouver, British Columbia, Canada, June 2003.

[POS] PostgreSQL. http://www.postgresql.org/

[SNI96] Snir, M., Steve, O., Huss-Lederman, S., Walker, D., & Dongarra, J. "*MPI: The Complete Reference*", MIT Press, 1996.

[THI] Thiebaut, D. Parallel Programming in C for the Transputer. http://maven.smith.edu/~thiebaut/transputer/descript.html

[TRI05] Tribou, Erik H. *"Millipede: A Graphical Tool for Debugging Distributed Systems with a Multilevel Approach"*, Masters Thesis, University of Nevada Las Vegas, Las Vegas, Nevada, USA, August 2005.

[WIL05] Wilkinson, B. & Allen, M. *"Parallel Programming Techniques and Applications using Networked Workstations and Parallel Computers"*, Second Edition, Pearson Prentice Hall, 2005.

[FLY72] Flynn, M. *"Some Computer Organizations and Their Effectiveness"*, IEEE Trans. Comput., Vol. C-21, pp. 94, 1972.

VITA

Graduate College
University of Nevada, Las Vegas

Hoimonti Basu

Local Address:
3955 Algonquin Dr, Apt 57
Las Vegas, NV 89119-5373, USA

Home Address:
C/O Mr. A. K. Basu
Santosh Mitra Road, Prembazar
P.O. Kharagpur, West Bengal, INDIA 721306

Degrees:

Bachelor of Technology (Honors), Metallurgical & Materials Engineering, 1998
Indian Institute of Technology (IIT), Kharagpur, India

Bachelor of Science, Computer Science, 2003
San Jose State University, San Jose, USA

Special Honors and Awards:

- Cum Laude Graduate, San Jose, 2003
- Second place at Graduate Level Mathematics Competition, San Jose, 2002
- Indranil Award for Metallurgical Engineering in India, Calcutta, India, 1999
- President's Silver Medal for leading the curve at IIT KGP, India 1998
- Sarat Memorial Award, Best woman undergrad at IIT KGP, India 1998
- Usha Martin Award, Best Senior Project Work at IIT KGP, India 1998
- J. C. Ghosh Memorial Award, Highest Senior GPA at IIT KGP, India 1998

Publications:

Basu, H., Godkhindi, M. M., & Mukunda, P.G. "*Investigations on the reactive sintering of porous silicon carbide"*, Journal of Material Science Letters, London, U.K., Vol.18, No. 5, pp. 389-392, March 1999.

Thesis Title: Interactive Message Debugger for Parallel Message Passing Programs using LAM-MPI

Thesis Examination Committee:

Chairperson, Dr. Jan Pedersen, Ph. D.
Committee Member, Professor Ajoy K. Datta, Ph. D.
Committee Member, Professor John T. Minor, Ph. D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph. D.