Extending the Eclipse Parallel Tools Platform debugger with Scalable Parallel Debugging Library

Chao Jin, Liang Ding, David Abramson*

Faculty of Information Technology, Monash University, Clayton, VIC, 3168, Australia

Abstract

The Eclipse Parallel Tools Platform (PTP) is an open source Integrated Development Environment (IDE) aiding the development of Supercomputer applications. The PTP parallel debugger is used by a growing community of developers in scientific and engineering fields. This paper proposes a method of improving the communication infrastructure of the PTP debugger by taking advantage of a Scalable Parallel Debugging Library (SPDL). Unlike the present communication framework of PTP, the Scalable Debug Manager (SDM), SPDL provides a pluggable architecture that allows developers to select a communication protocol suitable for a targeted supercomputer. It currently supports a number of scalable protocols, including MRNet and SCI. The advanced features provided by these communication trees, like programmable filters and configurable topologies, allow developers to create more flexible solutions of efficient reduction and aggregation operations for parallel debugging. In particular, they allow parallel debuggers to handle the large amounts of back-end messages in peta-scale environments with better efficiency. The architecture of the PTP debugger is extended to support SPDL. The extended architecture combines the advantages of the PTP debugger at the front-end and SPDL at the back-end. It improves the scalability and performance of the PTP debugger. Consequently, it provides a flexible option of utilizing the PTP debugger with pluggable communication protocols to address the debugging challenges in peta-scale environments.

Keywords: parallel debugging; scalability

1. Introduction

As the number of CPU cores increases on supercomputers, programming large-scale scientific applications faces many challenges. In order to handle the increased complexity of programming HPC (High Performance Computing) applications with a large number of parallel processes, efficient developing tools are critical to help programmers to guarantee the correctness of their parallel applications, optimize the performance, and improve the productivity of development correspondingly.

* Corresponding author. Tel.: +61-3-9905-8681; fax: +61-3-9902-0193.
E-mail address: david.abramson@monash.edu.
The Parallel Tools Platform (PTP) [6] is a plug-in project for Eclipse. It aims to advance the development of scientific applications by taking advantage of the successful open-source Eclipse platform. It provides an IDE that supports coding, debugging, job scheduling, tuning, and revision control for developing scientific applications. Programmers are enabled to manage the complexity of HPC scientific code development and optimize the performance of HPC applications with an improved productivity [8].

The PTP debugger [10] supports debugging parallel C, C++, Fortran, and UPC programs on a number of parallel computers. It provides a graphical user interface (GUI) to allow users to perform complex actions of parallel debugging, like controlling processes, inspecting the status of variables, managing breakpoints and viewing source code. As an open-source parallel debugger, it is used by a growing community of developers for building scientific and engineering projects [9][10]. Currently, it deploys the Scalable Debug Manager (SDM) [12] to allow the PTP client at the front-end to control the parallel processes of applications at the back-end. The topology of the SDM communication tree is not configurable. Its way of aggregating redundant messages that are collected from the back-end machines is not programmable. However, these limitations have been addressed by several standalone communication libraries, like SCI [11] and MRNet [22], which provide optimum performance and scalability for general tools used in peta-scale environments.

SPDL (Scalable Parallel Debugging Library) [14] is a debugging middleware that provides a set of fundamental debugging functions supporting multiple communication protocols, including point-to-point connections and tree-based overlay networks, like MRNet and SCI. Specifically, it takes different communication protocols as plug-ins and it allows debugging tools to select the appropriate communication solutions on the targeted parallel computers. It allows developers to utilize the programmable filters provided by these protocols for reduction and aggregation operations with the configurable topology of communication trees. Consequently, more flexible solutions can be formulated for efficiently handling a large amount of back-end messages in different scenarios. Moreover, it provides a portable way to launch parallel applications on a number of parallel computers, including a Cray XE6 supercomputer.

SPDL provides an easy way of improving the scalability of debugging tools across a wide range of computing platforms. It can be used to simplify the process of building new parallel debugging tools. It has been demonstrated to extend the scalability of Guard [4][5][19], a relative debugger, to support debugging more than several hundreds of thousands MPI processes on a Cray supercomputer [14].

This paper proposes a way to combine the advantage of the PTP debugger at the front-end and the advantage of SPDL at the back-end. This combination allows developers to utilize PTP’s graphical user interface while selecting a scalable communication protocol that is suitable for a targeted supercomputer. The goal is achieved with minimal engineering effort and all required updates are made transparently to the PTP users at the front-end. In particular, this paper presents the following contributions:

1) The back-end architecture of the PTP debugger has been extended to support SPDL. This improved architecture allows the PTP users to select using a proper back-end debugger, SDM or SPDL, for a targeted debugging scenario. With SPDL chosen as the back-end debugger, the PTP client can select a communication protocol, like MRNet or SCI, to control a large number of parallel application processes on a targeted parallel computer.

2) SPDL Gate is added between the PTP client and the SPDL client. It enables SPDL to fit into the present structure of the PTP debugger with minimal engineering effort. It wraps the debugging functions of SPDL with the interface of the SDM master to serve the debugging actions of the PTP client. The existing communication channel connecting the PTP client and the SDM master is re-used. Consequently, developers can use the same GUI of PTP to debug parallel applications with SPDL.

This extension improves the performance and scalability of the PTP debugger by taking advantage of proper communication protocols on the targeted computing platforms. It provides a flexible option allowing developers to utilize the PTP parallel debugger in a wider range of scenarios. The remainder of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 presents the design to extend the
back-end architecture of the PTP debugger. Section 4 discusses the implementation details of integrating SPDL with PTP. Section 5 evaluates its performance. Section 6 presents the conclusion.

2. Background and Related Work

The majority of modern parallel debuggers employ a client-server architecture in which a debug client running on a front-end (FE) machine is used to control a number of debug servers running on the back-end (BE) machines. The major process of parallel debugging consists of collecting the status of individual processes from the back-end servers and analyzing the data at the front-end to diagnose errors. Presently, however, parallel debuggers face the challenge of increasing number of CPU cores in peta-scale systems.

On one hand, an efficient debugging interface is critical to improve the productivity of parallel debugging. Normally, a GUI-based debug client is more productive than a CLI (Command Line Interface)-based client especially for many junior developers to conduct complex debugging actions. On the other hand, a tree-based communication structure (TBCS) is required to connect the FE client with BE servers in a scalable manner. Specifically, TBCS is able to decrease the number of socket connections between the FE client and BE servers, provides \( O(\log(n)) \) broadcast, and reduces the number of messages exchanged between the client and servers via data reduction.

Ladebug [23] is an enhanced parallel debugger on Alpha Linux and Tru64 Unix systems from HP. It pioneered the tree-based communication network; however, it is not actively supported at present. It only supports a command line interface.

Allinea’s DDT (Distributed Debugging Tool) [1] is a commercial parallel debugger. It provides a graphical user interface for debugging scalar, multi-threaded, and parallel applications written in C/C++ and Fortran. DDT supports a large number of different MPI implementations and batch queuing systems. The commercial versions of DDT support a proprietary tree-based communication mechanism. However, the technical details of its TBCS are not publicly available.

TotalView [2] is another commercial graphical debugger for C, C++, and Fortran applications. It supports MPI [20] and OpenMP [21]; and it has been deployed on a large number of supercomputers. It provides advanced data visualization and memory debugging features. It adopts MRNet [22], a standalone TBCS library, to handle communications between its front-end client and back-end servers.
Eclipse PTP [6] provides an open source parallel debugger to aid the development of scientific applications written with C, C++, Fortran, and UPC. The architecture of the PTP debugger consists of two major components: the Eclipse client and the debug server. The Eclipse client provides an interface for users to perform various debug actions. In contrast, its debug server, called Scalable Debug Manager (SDM)[12], provides a tree-based multicast/reduction network to control application instances, utilizing GDB (The GNU Project Debugger) to control individual processes.

SDM uses a client/server architecture that consists of a SDM master and multiple servers. With a typical configuration, the SDM master runs on a head node in a cluster, while the SDM servers are deployed on the computing nodes. The SDM master process manages the communication between the Eclipse client and back-end servers. At the back-end, each SDM server controls one application process via a GDB instance.

All of the SDM server processes are organized in a $k$-nomial topology [10]. Fig. 1.a illustrates the architecture of the PTP debugger with 10 servers. Its $k$-nomial topology is efficient for broadcasting commands to the back-end servers. However, this un-balanced topology makes it hard to provide optimum performance when aggregating large amount of redundant back-end messages. Furthermore, its filter of communication tree is not programmable and does not support a flexible way of aggregating redundant messages. These limitations prevent SDM from providing optimum performance for parallel debugging in peta- or exa-scale environments.

In order to improve the communication infrastructure of SDM, we propose to integrate SPDL with the PTP debugger. It provides developers a flexible option to select a proper communication framework for efficient parallel debugging in a targeted scenario.

3. Extending the Architecture of PTP debugger with SPDL

This section describes a way of extending the architecture of the PTP debugger with SPDL. There are different choices available in making the PTP debugger work with SPDL. The one presented in this section is for reusing the existing components of PTP as much as possible with minimal engineering overhead.

3.1. SPDL with Scalable Communication Protocols

Before presenting the integration, let us first review the architecture of SPDL with its pluggable communication protocols: MRNet and SCI. In comparison to SDM [12], MRNet [22] and SCI [11] provide multicast/reduction network with more advanced features.

SCI [11], released with Eclipse PTP by IBM, is a lightweight communication library that provides scalable message transmission functions for a client-server model, especially for a master FE node associated with a large number of slave BE nodes. The communication tree of SCI supports a $k$-nomial topology and its fan-out degree is configurable. SCI provides a plug-in mechanism to filter messages transferred across SCI agents. Message aggregation can be achieved through programming filters. In comparison to SDM, SCI provides better scalability with a programmable approach of aggregating a large amount of BE messages. The PTP debugger plans to replace the broadcast/reduction network of SDM with SCI. However, the engineering effort to achieve this replacement is not trivial.

MRNet [22] is a software overlay network using a tree of communication processes to connect FE and BE nodes. Its communication tree can be utilized to broadcast/multicast messages downstream and collect or aggregate messages upstream. Similar to SCI, it provides a pluggable mechanism for message filtering. Moreover, the tree organization is configurable and it supports common network layouts like $k$-ary and $k$-nomial trees, or custom layouts tailored to the specific requirements. In particular, the balanced topology of MRNet tree is able to provide an efficient framework of collecting a large amount of messages from BE nodes [18]. It has been used to scale STAT (Stack Trace Analysis Tool) up to supporting hundreds of thousands of CPU cores [7].
Although both MRNet and SCI utilize similar tree structures for efficient group communications, they provide significantly different methods to transfer messages and launch back-end processes. On a targeted computing platform, in order to make the best use of a specific communication protocol, special optimization must be conducted. The unified interface of SPDL provides an easy way of utilizing these two scalable communication protocols with minimal engineering overhead.

The architecture of SPDL consists of one front-end client and multiple back-end debug servers, as illustrated by Fig. 1.b. Its communication protocol is pluggable. Specifically, it takes different protocols, like MRNet, SCI, and point-to-point socket connections (which is not scalable, but provides a simple reference implementation for machines that has no support of MRNet and SCI), as plug-ins. SPDL is designed to meet the communication requirements of debugging SPMD (Single Program Multiple Data) programs. It has been demonstrated to provide a low latency for interactively debugging a large number of MPI parallel processes on a Cray XE6 supercomputer [14]. It also provides a generic launching infrastructure and presently supports a number of import native launch services, including Cray ALPS [3], IBM POE (Parallel Operating Environment) [13], the queuing system of Sun Enterprise Grid (SEG) [24], and MPIRUN [20].

The SPDL client provides a command set of Basic Debugging Functions (BDFs). The debugging functions are chosen to be compatible with the HPDF (High Performance Debug Forum) standard [15] with an extension of supporting scalable group operations. Each SPDL server controls one application process with one instance of GDB (GNU Project Debugger). Each debug server is assigned a unique identifier, called a SID (Server ID), which corresponds to the MPI rank of its MPI process.

The architectures of SPDL and SDM share several similarities. First, the debugging functions of SPDL client are almost equivalent to the debugging commands of PTP. Second, both SDM and SPDL utilize GDB to control application processes at the back-end. Third, the two systems are both C programs. These similarities enable a successful integration of SPDL with the PTP debugger after making a proper improvement.

### 3.2. The Architecture of PTP debugger with SPDL

The PTP parallel debugger has been extended to support SPDL. Users are allowed to choose SPDL as the back-end debugger by using the panel of Debug Configurations, as illustrated in Fig. 2. Currently, SPDL only supports loading gdb-mi as debugger back-end. Other configurations are the same as in configuring SDM.
The extended architecture of the PTP debugger with SPDL is illustrated in Fig. 3. Different from the architecture in Fig. 1, SPDL substitutes SDM to aid the PTP client to control the back-end application processes. Similar to the architecture of PTP with SDM, the debug servers of SPDL still use GDB to conduct low-level debugging operations.

In order to make SPDL work with PTP properly, the SPDL Gate is added between the PTP client and SPDL client at the front-end, as illustrated in Fig. 3. It serves the debugging actions of the PTP client by invoking the debugging functions of SPDL. Specifically, the SPDL Gate exposes the debugging functions of SPDL with the interface of the SDM master. Keeping the same interface allows us to use the existing proxy layer to connect the PTP client and SPDL client. Therefore, it requires no significant updates in the Eclipse client to make SPDL work with the PTP debugger. As a consequence, the SPDL Gate enables SPDL fitting into the present structure of the PTP debugger with a minimal engineering effort.

However, the SPDL Gate needs to fill the gap caused by the mismatched part in the two sets of debugging commands. Specifically, the debugging commands received by the SPDL Gate are in a PTP format and correspondingly, the response for each debugging command also expects the PTP format that the Eclipse client can understand. Therefore, the SPDL Gate is responsible for: 1) accepting the debugging requests sent by the PTP client and realizing them by invoking corresponding SPDL debugging commands; 2) collecting responses from the SPDL servers, converting them to the PTP format and forwarding them to the PTP client.

At the back-end, SPDL is able to use an MI (Machine Independent) interpreter to control GDB, which is also used by SDM. Therefore, no updates are required in the SPDL server.

4. Implementations

There are several challenges to realize this architecture. First, although both SPDL and SDM utilize GDB at the debug back-end and they provide equivalent debugging functions, the principles of implementing both command sets are different at the front-end. Realizing each debug command involves sending requests to the back-end servers, collecting corresponding responses, and then forwarding them to the PTP client. The SDM master implements this in an asynchronous manner, while SPDL employs a synchronous approach. More importantly, the process of invoking a debug session is different in both systems. Specifically, the PTP client creates the SDM servers by calling an external launch service, like MPIRUN. In contrast, launching SPDL servers is totally controlled by the SPDL client.

4.1. Presentations of back-end servers

Currently, the PTP client only maintains one active debugging session. Within each debugging session, the PTP client performs debugging actions on a group of MPI processes. The debug command is sent to each SDM
debug server that controls one MPI process of the targeted group. Each SDM server is identified by the MPI rank of its MPI process. PTP uses a \textit{bitset} to represent a group of back-end servers, which is a realization of a bit array. Each bit in the array represents one debug server. The \textit{bitset} is also used to in each aggregated response to represent a group of message senders. However, the implementation of \textit{bitset} is not efficient when dealing with a large and sparse bit array.

Similar to SDM, SPDL employs the same way of identifying back-end servers with MPI ranks. However, SPDL utilizes a \textit{bitvector}, a \textit{compressed bit array}, to represent a group of back-end servers. The interface of \textit{bitvector} is similar to a normal bit array. However, its implementation provides a more scalable solution even for large and sparse bit vectors. It is able to save a significant overhead when aggregating back-end messages for peta- or exa-scale systems.

For each debugging command, SPDL Gate needs to translate its \textit{bitset} into a \textit{bitvector} before invoking its corresponding SPDL command. Accordingly, the responses collected from the back-end servers are aggregated and each unique message is tagged with a \textit{bitvector} indicating theirs senders. The \textit{bitvector} needs to be converted back to a \textit{bitset} before forwarding the response to the PTP client.

### 4.2. Debugging commands

To serve each PTP debugging command, the SPDL Gate needs to invoke its corresponding debug function in SPDL. The SPDL Gate defines a command translation table, illustrated in Table 1, to convert each PTP debugging command into its correspondence in SPDL.

Specifically, most SDM commands can find a direct correspondence in the SPDL command set, except those denoted with ‘*’ in Table 1. These commands are not directly supported by SPDL, but SPDL exposes GDB/MI commands to handle them. The DbgMasterCreateSession command corresponds to a null command in SPDL, because a debugging session is initialized in DbgInvoke with SPDL.

<table>
<thead>
<tr>
<th>Category</th>
<th>SDM Commands</th>
<th>Functionality</th>
<th>SPDL Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>DbgMasterInit</td>
<td>Set callback functions &amp; initialize proxy</td>
<td>DbgInit</td>
</tr>
<tr>
<td></td>
<td>DbgMasterCreateSession</td>
<td>Connect to Eclipse PTP Front-end</td>
<td>Null</td>
</tr>
<tr>
<td></td>
<td>DbgMasterStartSession</td>
<td>Start debugging session</td>
<td>DbgInvoke</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>DbgMasterSetLineBreakpoint</td>
<td>Set a breakpoint at specified line</td>
<td>DbgSetLineBreak</td>
</tr>
<tr>
<td>operations</td>
<td>DbgMasterSetFuncBreakpoint</td>
<td>Set a breakpoint at the start of specified function</td>
<td>DbgSetFuncBreak</td>
</tr>
<tr>
<td></td>
<td>DbgMasterDeleteBreakpoint</td>
<td>Delete a specified breakpoint</td>
<td>DbgDeleteBreak</td>
</tr>
<tr>
<td>Process</td>
<td>DbgMasterGo</td>
<td>Start / Continue executing program</td>
<td>DbgGo</td>
</tr>
<tr>
<td>Control</td>
<td>DbgMasterStep</td>
<td>Execute the program line by line</td>
<td>DbgStep</td>
</tr>
<tr>
<td>operations</td>
<td>DbgMasterTerminate</td>
<td>Terminate program execution</td>
<td>DbgKill</td>
</tr>
<tr>
<td></td>
<td>DbgMasterSuspend</td>
<td>Interrupt an executing program</td>
<td>DbgInterrupt</td>
</tr>
<tr>
<td></td>
<td>DbgMasterQuit</td>
<td>Quit debugger</td>
<td>DbgFinish</td>
</tr>
<tr>
<td>Stack frame</td>
<td>DbgMasterListStackframes</td>
<td>List current or all stack frames</td>
<td>DbgShowFrame</td>
</tr>
<tr>
<td>operations</td>
<td>DbgMasterGetCurrentStackframe</td>
<td>Move up or down a specified level of stack frames</td>
<td>DbgMoveFrame</td>
</tr>
<tr>
<td></td>
<td>DbgMasterListStackDepth</td>
<td>List the depth of current stack frame</td>
<td>DbgShowFrame</td>
</tr>
<tr>
<td>Expression</td>
<td>DbgMasterListLocalVariables</td>
<td>List local variables</td>
<td>DbgGetVars</td>
</tr>
<tr>
<td>and Variables</td>
<td>DbgMasterListArguments</td>
<td>List arguments of specified function</td>
<td>DbgGetVars</td>
</tr>
<tr>
<td>operations</td>
<td>DbgMasterListGlobalVariables</td>
<td>List global variables</td>
<td>DbgGetVars</td>
</tr>
<tr>
<td></td>
<td>DbgMasterGetType</td>
<td>Find native type of specified variable</td>
<td>DbgGetType</td>
</tr>
<tr>
<td></td>
<td>DbgMasterEvaluateExpression</td>
<td>Evaluate specified expression</td>
<td>DbgEvalExpr</td>
</tr>
<tr>
<td></td>
<td>DbgMasterEvaluatePartialExpression</td>
<td>Evaluate a partial expression</td>
<td>DbgEvalExpr</td>
</tr>
<tr>
<td>Thread</td>
<td>DbgMasterListInfoThread</td>
<td>List the current thread’s information</td>
<td>*</td>
</tr>
<tr>
<td>operations</td>
<td>DbgMasterSetThreadSelect</td>
<td>Set specified thread as current thread</td>
<td>*</td>
</tr>
<tr>
<td>Memory</td>
<td>DbgMasterDataReadMemory</td>
<td>Read specified memory</td>
<td>*</td>
</tr>
<tr>
<td>operations</td>
<td>DbgMasterDataWriteMemory</td>
<td>Write data to specified memory</td>
<td>*</td>
</tr>
</tbody>
</table>
Moreover, for each command, the Gate also performs additional services like error processing, checking program status and addressing the difference between two command sets. For instance, the parameters of each debugging function are different in both systems. The SPDL Gate decodes the parameters of the PTP debugging commands and re-orders them to follow the sequence required by the SPDL debugging functions.

The PTP client expects a SDM event as the response for each debugging command. However, the SPDL servers return a SPDL event as response to each debugging command. In order to take advantage of the existing PTP proxy that transfers each SDM event as a proxy message to the PTP client, the SPDL Gate needs to convert the SPDL event to its correspondent SDM event.

In terms of communication, the SPDL Gate needs to intercept the messages transferred bi-directional, and translate them before forwarding them. The interface of the SDM master expects an asynchronous way of handling all the messages collected from the back-end servers. However, SPDL treats them differently. Specifically, each debugging command of SPDL returns a synchronous response, while the events and I/O generated by debug servers are collected as asynchronous messages. Therefore, for the messages of events and I/O, the SPDL Gate forwards them directly after translating them. However, for the response of each debugging command, the SPDL Gate needs to simulate an asynchronous response.

4.3. Startup

The startup process of SPDL is different from SDM. The SPDL client itself controls launching the back-end servers and application processes, which is done via the DbgInvoke command. In contrast, launching SDM servers is conducted by the PTP client. In particular, the PTP client launches the SDM servers and master by invoking the launch method specified by submit-interactive-debug, which is defined in the system configuration of targeted resource. After that, a set of startup commands, including DbgMasterInit, DbgMasterCreateSession and DbgMasterStartSession, are executed by the SDM master. The startup process completes after the SDM master receives the connection requests from the SDM servers and acknowledges the PTP client. Fig. 4 illustrates the difference with using Open MPI as the resource manager.

The existing mechanism of launching the SDM servers is re-used to invoke a SPDL debugging session in order to avoid any significant changes in the PTP client. With SDM, to invoke a debugging session, users need to set the name of executable program to debug and the path where SDM is located in the target parallel computer via the PTP GUI. Then, the PTP client conducts the invoke method defined by submit-interactive-debug for the resource manager on the target computer to launch the SDM servers and application processes. Presently, PTP supports a number of resource managers, including 1) Open MPI-Generic-Interactive; 2) IBM Parallel Environment and 3) Torque-Generic-Interactive. With Open MPI, the launch process is conducted by...
invoking a startup script, *start_job.pl*. Specifically, it consist of 3 main steps: 1) launch the SDM server processes onto the compute nodes; 2) generate a routing file containing the rank, hostname, and port number for each server process in the communication tree; and 3) launch the process of SDM master and pass it the session address and port to connect to the Eclipse client. Fig. 4.a illustrates the process of launching SDM with Open MPI.

With SPDL, the startup process is simplified into 2 major steps: 1) generate the startup information required by the SPDL client; 2) launch the SPDL client process and pass it the session address and port to connect to the Eclipse client. The detailed startup process of invoking a debugging session with SPDL is illustrated in Fig 4.b. The invoke method defined in *submit-interactive-debug* for the target system is ignored. In contrast, a SPDL specific script, *Start_SPDL.pl*, is called to launch the SPDL client and provide it the important information that is missed in the startup process of SDM. For instance, the number of application processes is not required by the SDM master, but is used by the SPDL client. Different from the startup process of SDM, *Start_SPDL.pl* does not directly launch parallel application and SPDL servers. In contrast, they are launched by the SPDL client. Currently, SPDL supports a number of launch services, including ALPS, IBM PoE, the queuing system of Sun Enterprise Grid (SEG) and MPIRUN with Open MPI and MPICH2.

5. Performance Evaluation

This section briefly examines the performance of the extended PTP with SPDL. We debugged a simple parallel application of 64 processes running on a cluster of 32 nodes with PTP. Each node is equipped with 2 Quad-Core Intel Xeon E5310 1.6GHz processors and 8 GB RAM. The selected resource manager is Open MPI.

The latencies perceived by the PTP debugger for the *Step* and *ListStackframes* commands were examined with SPDL and SDM respectively. During the experiment, MRNet 4.0.0 was used by SPDL. Each command was conducted 5 times and the averaged values are presented.

With SPDL, the latency of *Step* is 294 ms, while the latency of *ListStackframes* is 55 ms. In comparison, with SDM, *Step* requires 252 ms to complete and *ListStackframes* needs 85 ms. Overall, the performance of the PTP debugger with SPDL and MRNet is acceptable.

6. Conclusion

Scalable programming tools are critical to address the challenge of developing scientific applications in peta-scale environments. The Eclipse PTP parallel debugger is used by a growing community of developers for building scientific and engineering projects. However, the current communication framework of the PTP debugger, SDM, has a number of limitations, including non-programmable approach of aggregating back-end messages and non-configurable topology of its communication tree. These missed functions are essential to provide efficient parallel debugging tools running on extremely large-scale systems [4]. In contrast, SCI and MRNet are standalone scalable communication libraries to improve the scalability of general tools running in peta-scale environments. For instance, MRNet has been demonstrated on the tools running on many largest computing platforms, including Jaguar (Cray XT5 system) and IBM BlueGene/L (BG/L) supercomputers [4][18], while SCI aims to been used in tools running on a wide range of supercomputers [11]. Our project is motivated by improving the scalability of the PTP debugger and accordingly broadening the range of platforms it supports. We have integrated SPDL with the PTP debugger by extending its back-end architecture. With SPDL, PTP users can debug parallel applications by selecting an appropriate communication protocol for the targeted computing platforms. Consequently, it enables the PTP debugger to take advantage of the advanced features of these scalable multicast/reduction networks and their performance optimized for the targeted computing platforms.
Acknowledgements

This project is supported by the Australian Research Council, the MURPA (Monash Undergraduate Research Projects Abroad) program which is funded by Monash University. We acknowledge Greg Watson at IBM and Jay Alameda and his team members at NCSA (National Center for Supercomputing Applications) for their contribution to this project. We thank Jeff Tan for improving the quality of this paper. The project is also supported by the Office of Science of the U.S. Department of Energy under Agreement No. DE-FG02-09ER25929. Liang Ding performed the work in partial fulfillment of his Bachelor of Software Engineering research project.

References