

An Integrated Approach To Improving The Parallel Application Development Process

Gregory R. Watson
IBM T.J. Watson Research Center
grw@us.ibm.com

Craig E Rasmussen
Los Alamos National Laboratory
crasmussen@lanl.gov

Beth R. Tibbitts
IBM T.J. Watson Research Center
tibbitts@us.ibm.com

Abstract—The development of parallel applications is becoming increasingly important to a broad range of industries. Traditionally, parallel programming was a niche area that was primarily exploited by scientists trying to model extremely complicated physical phenomenon. It is becoming increasingly clear, however, that continued hardware performance improvements through clock scaling and feature-size reduction are simply not going to be achievable for much longer. The hardware vendor's approach to addressing this issue is to employ parallelism through multi-processor and multi-core technologies. While there is little doubt that this approach produces scaling improvements, there are still many significant hurdles to be overcome before parallelism can be employed as a general replacement to more traditional programming techniques. The Parallel Tools Platform (PTP) Project was created in 2005 in an attempt to provide developers with new tools aimed at addressing some of the parallel development issues. Since then, the introduction of a new generation of peta-scale and multi-core systems has highlighted the need for such a platform. In this paper, we describe some of the challenges facing parallel application developers, present the current state of PTP, and provide a simple case study that demonstrates how PTP can be used to locate a potential deadlock situation in an MPI code.

I. INTRODUCTION

Parallel computers have existed in one form or another almost since the first computers were available. The complexity introduced by parallelism was evident from a very early stage, and has been a major impediment to the adoption of parallelism in main stream application development. Many programming models and techniques have been used to improve the simplicity and reliability of parallel programs. Dozens of new languages and language features were introduced, however very few are still widely used. In the 1990's, the Message Passing Interface (MPI) standardization effort [1] was seen as a major step forward in parallel programming models. The predominant programming models still in use are asynchronous threads and MPI, although the use of partitioned global address space (PGAS) languages, such as Unified Parallel C (UPC) [2], appear to be increasing in popularity. Although the PGAS languages simplify the programmer's task to some extent, the potential for deadlocks and other synchronization issues still remain a significant challenge.

The first integrated development environment (IDE) was introduced when computer input devices became sophisticated enough to support the seamless integration of development activities. Due to performance and usability issues, however, there was often programmer resistance to the wholesale adoption of IDEs. The quality and productivity improvements achieved using IDEs has been well established [3], [4], [5], [6], and combined with improvements to the IDEs themselves, they are now the predominant environment for software

development. Although a considerable number of IDEs are available today, many are limited to a single operating system (e.g. KDevelop, Visual Studio), or are proprietary (e.g. Visual Studio, Xcode, Sun Studio). Eclipse [7] is one of the few truly cross-platform IDEs that has been designed for extensibility. Interestingly, although IDEs have been used in the past to aid parallel application development [8], [9], [10], [11], none of these are still available today. Few developers working on parallel scientific codes use IDEs at all.

The Eclipse Parallel Tools Platform (PTP) was launched in 2005 in an attempt to address this situation. At this time, commodity clusters had largely replaced custom proprietary hardware for high performance computing (HPC), however the predominant parallel application development environment was still command-line tools. At the same time, the move towards multi-core architectures for conventional applications was outpacing the ability of existing IDEs to provide the tools necessary to exploit the new technology. As the HPC and multi-core architectures begin to converge, the need for new programming models and more sophisticated tools now has a greater urgency.

PTP builds on the exemplary tools available in the Eclipse platform and the Eclipse C/C++ Development Tools (CDT) to provide support for C, C++, UPC, Fortran, and in the future other parallel languages. It is also a *platform*, so that while it provides a range of core services and tools, it is also designed to be extended to support new tools, architectures, and programming models. In addition to Eclipse's advanced editing, build, and integrated source code management functionality, PTP provides four additional features: advanced error checking and static analysis tools that assist the programmer to develop parallel applications; runtime monitoring and control of parallel jobs; debugging support for multi-process applications; and an external tools framework for the integration of dynamic analysis tools, such as tools for performance analysis and optimization. In the first of these, PTP provides a number of tools that are primarily aimed at the MPI and OpenMP [12] programmer, and that reduce much of the tedious and error-prone nature of these programming models. Runtime monitoring and control of parallel jobs abstracts the interaction between the developer and the parallel system, so that the developer is able to seamlessly launch and control applications without needing to focus on specific architecture details. The parallel debugging support provides a range of features to support debugging parallel applications, but that can also be extended to encompass the new debugging paradigms that will be required on peta-scale and multi-core systems. The external tools framework allows existing performance and other kinds of dynamic analysis tools to be easily integrated into the

Eclipse framework so they are accessible to the developer. Finally, recent work has added remote development capability to PTP. The Remote Development Tools (RDT) are a series of enhancements to CDT that allow projects to be physically located, built, and executed on a remote system, while Eclipse runs on the developer's local workstation.

In the following sections, we will outline some of the challenges faced by developers, describe some of the main features of PTP that are specifically designed to aid parallel application development, provide a simple case study that shows how a potential deadlock situation was discovered in an MPI code, and discuss future directions for the PTP project.

II. CHALLENGES

With the growing popularity of multi-core systems as a means of improving application performance, parallel programming is set to enter the main stream. The challenge posed by these systems is twofold: existing applications will need to be modified to make use of the new architectures if any performance improvements are to be obtained; and new programming models and languages will be required to manage the extra complexity that parallelism introduces. Although explicit threading has been used effectively as the predominant programming model for shared memory architectures, it is neither easy to program correctly, nor conducive to retrofitting applications in order to utilize the new architectures. How existing applications will benefit from the new age of parallelism without huge investments in re-engineering is still very much an unanswered question.

In scientific computing, explicit parallelism has been employed with varying degrees of success for many years. Unfortunately, the homogeneous architectures that have facilitated these programming models have reached a practical limit in the search for peta-scale performance and beyond. One approach to addressing this is to offload large portions of the computation load onto some form of accelerated hardware. The result is a very heterogeneous environment that introduces significant complexity into the application development process. In an attempt to address these problems, a large scale effort is underway to develop new programming models and languages that will reduce the complex and error-prone nature of parallel application development, and to develop new tools that will aid both legacy and new applications to extract the maximum performance from the new architectures. Many of the current issues encumbering scientific application developers have been discussed elsewhere [13], and we will consider the implications of some of these here. However, few major scientific applications have yet to face the challenges that the next generation of peta-scale machines will introduce.

In many computing environments it is already unusual for computational resources to be available locally, and development processes are becoming complex enough to require significant resources in themselves (e.g. building large applications can take many hours). In these situations, the ability to develop applications remotely will be an important requirement, as will launching and debugging applications on the remote systems. Further, as many of these systems employ batch schedulers, the developer must be able to easily submit jobs for execution, and be notified when the job has completed execution. Providing a development environment

that is flexible enough to deal with the many possible system configurations, but without burdening the developer with additional complexity, is a major challenge.

Effectively debugging applications is an area that still presents significant problems to the developer. The current generation of parallel debuggers have simply extended the sequential debugging paradigm to multiple processes (or threads), but do not address problems such as algorithmic debugging, nor how to deal with errors resulting from concurrency issues, scaling, or other non-deterministic failures. In addition, the next generation of peta-scale machines are expecting in the order of one million executing tasks, and existing threaded applications, which already exhibit thousands of threads, are likely to also increase in size significantly. Dealing with large numbers of objects (threads, processes, etc.) raises many scalability issues with the debugger itself, both in the ability of the user interface to display and manage the objects, and in the communication services that are used between remote systems and the local environment.

As system complexity increases, the process of building an application is likely to become more complicated. Currently, applications are built with a (albeit intricate) linear sequence of compile and link steps. In the future, however, it is possible that many more activities will be required to produce an optimized application executable. For example, multiple programming models may need to be combined (as is already required for IBM's Cell Broadband Engine), or dynamic performance information gathered at runtime may be required to augment the static analysis performed by the compiler. Tools designed for building sequential applications on SMP architectures are unlikely to suffice for this purpose.

The DARPA HPCS Language Project [14] is driving the development of programming models and languages for scientific computing, and a similar evolution will also likely to be necessary for multi-core systems. However, a huge amount of legacy language code development will continue for the foreseeable future, and new tools will be essential if these codes are to adapt to the new architectures. Techniques such as the static analysis of programs and refactoring will play a vital part in making this happen. The infrastructure required for these tasks is significantly more complex than anything required by development environments before (apart from compiler internals), so a large engineering effort will be required to build the appropriate frameworks and tools.

There are many other challenges that still prove to be significant obstacles to the effective and productive development on scientific codes, such as the difficulties in porting applications to new hardware, simplifying the use of defect tracking tools as part of the development cycle, providing comprehensive unit and regression testing, managing application data requirements and visualization, and verification and validation. Many of these are also likely to impact on multi-core development practices.

The Parallel Tools Platform project's aim of building on the unique capabilities of Eclipse in order to address as many of these challenges as possible is an ambitious one. While much work still remains to be done, we are confident that the functionality described here will demonstrate that PTP is well on the way to meeting this goal.

III. INTEGRATED TOOL PLATFORM

Eclipse is an open development platform that provides an extensible framework for integrating tools to support the software development lifecycle. It is an open source, portable, flexible, plug-in management system that forms the core of a fully featured integrated development environment. The Eclipse SDK adds a number of plug-ins that provide the wide range of software development tools, services, and documentation expected in an IDE. These include integrated features such as syntax aware editors with content assist and code formatting, context-sensitive help, language-specific searching and navigation, code refactoring, managed and un-managed build systems, application launch services, local and remote multi-thread debugging, version control, and defect tracking. Eclipse also provides multi-language support, including C, C++, Fortran, UPC, and a range of scripting languages, such as Python. As these features have been discussed elsewhere [15], we will not consider them in detail here.

The Parallel Tools Platform (PTP) is an extension of the Eclipse platform that provides features specifically designed to aid parallel application development. It currently focusses three main areas: the tools and infrastructure necessary for advanced error checking and analysis of parallel applications; a runtime environment that allows developers greater transparency into the systems on which they are developing applications; and a parallel debugger that will allow developers the ability to more easily locate errors and anomalies in program behavior. In the following sections we will describe each of these three aspects of PTP in more detail. The underlying architecture of PTP has already been described in [16]. PTP also provides a framework for integrating performance tools into Eclipse, but discussion of this will be the subject for a future paper.

A. Static Analysis Tools

The PTP analysis tools are aimed at providing Eclipse with an additional feature set that is designed to aid developers writing parallel applications. These tools are currently targeted at the MPI and OpenMP programming models, but we fully expect them to be extended to other models or languages as the need arises.

1) *Advanced Help and Content Assist*: Eclipse includes an integrated help system that provides a help browser and context sensitive help that can be accessed directly from the user's editor session. PTP augments this help system with MPI- and OpenMP-specific information in order to improve the developer experience when using these programming models. Reference information about the MPI and OpenMP API, including arguments, return type, and a functional description, are available via the help browser or by simply placing the cursor over an API in the editor view to activate *hover help*. The Eclipse content assist has also been augmented to enable auto completion of APIs and arguments while typing.

2) *Artifact Analysis*: This analysis tool allows the developer to more easily work with MPI and OpenMP codes by providing a higher level abstraction of the APIs. Like the *outline view*¹, the *artifact view* shows a list of all MPI function calls,

¹The Eclipse outline view provides an outline of the program showing its structural elements.

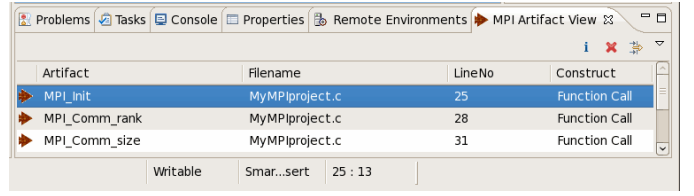


Fig. 1. View showing MPI artifacts discovered in the source code.

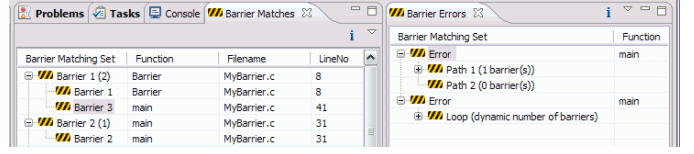


Fig. 2. View showing barrier matching sets and barrier errors that were discovered using static analysis.

OpenMP pragmas, and other artifacts in the program. Figure 1 shows the MPI artifact view. Navigation to the source code location of these artifacts is achieved by clicking on the artifact in the view, or by using icons in the navigation bar of the currently open editor.

In addition to augmented views, static analysis also provides more advanced error checking features than are typically available in Eclipse. These are the types of checks that could be made by compilers, but by providing an integrated tool it is possible to provide error reporting much earlier in the development cycle. Currently, checks for many of the known OpenMP programming errors are provided.

3) *Barrier Analysis*: The barrier analysis tool was the first example of a static analysis tool provided by PTP, and can be used to detect potential deadlocks in MPI applications. The tool does this by identifying the location of all MPI barrier statements² in the application (which may be scattered throughout the source code), and constructs *barrier matching sets*. Each set comprises all the barrier statements that could execute in parallel. Using this information, it is possible to determine if there are any barrier statements that do not have a matching barrier, and flag these as potential deadlock errors. The technique used to compute the barrier sets is described in detail in [17]. The tool also provides a *barrier view* to enable the easy navigation to barrier statements in the source code. Figure 2 shows an example of the barrier view containing a list of barrier sets.

4) *Concurrency Analysis*: Like the barrier analysis tool, the concurrency analysis tool was provided to detect potential concurrency problems, but for OpenMP (threaded) applications. Details on the analysis technique used by the tool are provided in [18]. The concurrency analysis tool allows the developer to choose a particular expression, and then evaluate and identify all expressions that could execute concurrently with the selected expression. Since it is important to ensure that only expected expressions execute in parallel, this tool can be used to detect potential race and deadlock conditions.

²An MPI barrier causes each process to wait until all processes have reached a barrier. It is used to synchronize all processes.

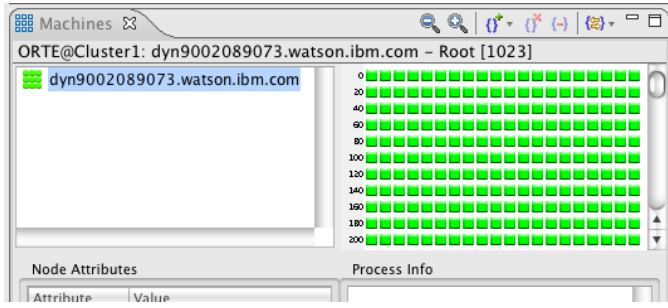


Fig. 3. View showing the status of the first 220 nodes of a 1024 node cluster.

B. Runtime Tools

One of the difficulties facing the parallel application developer is the lack of transparency about the behavior and status of applications and the machines that they run on. Further, many parallel systems have a more complex interface than POSIX-style command-line execution, and because they are typically a scarce resource, many employ a batch scheduler to manage access to the computational resources. Not only must the developer spend time learning the interfaces and integrating these with their development processes, but each iteration of the development cycle can be encumbered with unnecessary and tedious activities.

To facilitate a more productive development environment, PTP provides a number of abstractions that simplify the interaction with target systems. The first of these is a *runtime model*³ that provides an abstract representation of the parallel system that the developer is interacting with. This model forms the core of a model-view-controller design pattern. Information about the parallel system, and applications running on the system, is fed into the model to update the attributes of model elements. PTP provides a number of views into this model that enable the developer to monitor the status of the system and the applications as they are executing. The advantage of this approach is that new views can be easily added to expose different features of the underlying system (e.g. network topology), or to display information in a completely different way (e.g. to scalably display racks of a large parallel system).

The second abstraction that PTP provides is the notion of a *resource manager*, which represents any subsystem that manages resources on a target system. Examples of possible resource managers include: MPI runtime systems; job schedulers; virtual machines; and simulators. PTP allows multiple resource managers to be configured simultaneously, and places no restrictions on the location of the resources, so they can be local or remote to the Eclipse environment. Internally, a resource manager is just another part of the runtime model hierarchy, so the model views can be used to provide a display of the status of any resource managers that have been configured. In addition to monitoring activities, the resource manager is also responsible for submitting jobs for execution, and initiating debug sessions. The protocol used to interact with remote resource managers is user-selectable during the configuration process. This includes allowing communication

³Not to be confused with a programming model. The runtime model only provides a model of the parallel machine for monitoring and control purposes

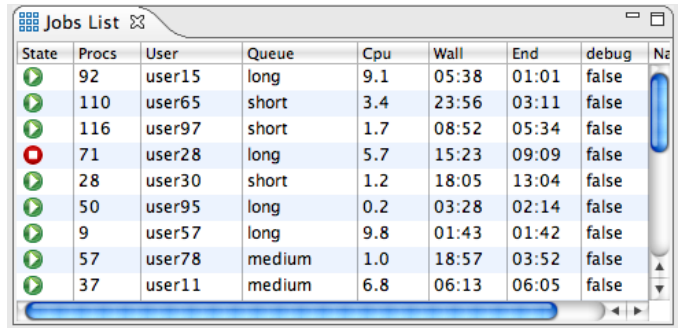


Fig. 4. View showing a list of jobs (red - completed, green - running).

to be tunneled over a secure ssh connection to address the security requirements of many installations. Figure 3 shows one type of view for monitoring a large cluster.

Launching of parallel applications is managed through the normal Eclipse launch configuration mechanism. PTP adds a parallel application launch type that allows the developer to select the resource manager that will be used to control job submission, and supply resource manager specific attributes that specify resource constraints on the job. Once a job has been submitted, the runtime views allow the user to monitor progress of the job on the target system. Figure 4 shows an example of a view for monitoring job status.

C. Debug Tools

A key aspect of any development process is the ability to effectively locate and correct program errors. Debugging has traditionally been a difficult area for parallel application developers, since traditional debugging methodologies only apply when the number of parallel tasks remains small, and the very act of debugging can perturb the application enough to make identifying temporal issues very difficult. Very few parallel debuggers currently exist, so developers have, until recently, only had a relatively few options: purchase a commercial parallel debugger, attempt to use a sequential debugger (such as gdb) or a debugger wrapper (such as mpigdb), or use debug tracing statements (`printf` or equivalent).

As only a small number of commercial parallel debuggers exist⁴, there is little competition to drive innovation and new functionality, and with only a small potential market, this can be an expensive debug solution. Also, these debuggers suffer from scalability problems when debugging applications larger than a few thousand processes. The gdb or mpigdb options, while cheaper, also suffer from scalability and usability issues. Neither the commercial nor open source solutions are integrated with a complete development environment, so launching a debug session can be a challenging exercise. Using tracing, while neither scalable nor powerful, is at least ubiquitous and easy to use. As a result, this has become the defacto debugging paradigm for parallel programming in many situations.

PTP attempts to overcome these limitations, by providing an integrated parallel debugger that can be activated whenever the developer requires detailed debugging information about the

⁴At the time of writing only three: TotalView, DDT and the Intel Debugger (IDB).

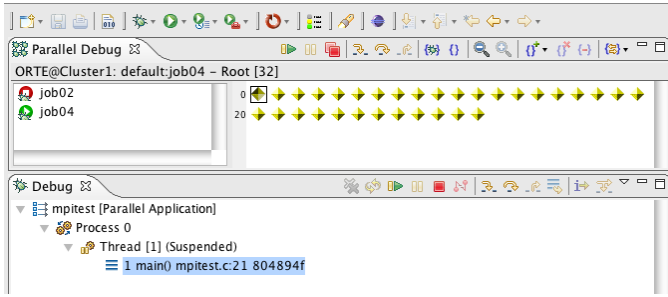


Fig. 5. Parallel debug view showing a 32 process job being debugged.

application under development. In addition to normal debugging functionality, such as setting breakpoints, single stepping, viewing and altering variables, etc., the debugger also gives the developer the ability to control and manipulate arbitrary sets of processes associated with a parallel application as it is executing. By default, the debugger establishes a set of all processes in the application run, and commands such as setting a breakpoint, single stepping, or resuming execution can be applied to this set of processes. The set can be subdivided into an arbitrary number of subsets (including individual processes) that allow finer control of application execution. Figure 5 shows the *parallel debug view* which allows manipulation of sets of processes.

Debugger scalability is always an issue, and the PTP debugger is no exception. However, the debugger infrastructure has been designed to scale, and so far has proved effective up to the same application sizes that can be handled by the commercial debuggers. In addition, because the PTP debugger is an open architecture, we hope that it will be used as a platform to develop new debugging paradigms that will be necessary to deal with applications that comprise hundreds of thousands or millions of parallel tasks.

D. External Tools Framework

PTP recognizes the fact that there are already many powerful tools available for assisting developers to maximize the performance and productivity of their applications. Most of these tools are stand-alone or command-line based, and are typically quite complicated to use, but the benefits of these tools to the development process can be significant. Although it is possible to integrate external tools into the Eclipse framework, the cost and complexity of doing so can be high, and many tool developers do not see the benefits that integration with Eclipse can bring. To better facilitate the integration of such tools, the PTP External Tools Framework (ETF) has been developed to reduce the burden on the tool developer and simplify the process of integrating the tool with Eclipse.

The ETF defines five integration points that can be specified using only an XML description file. These integration points are:

- *Instrumentation*. This includes the automatic and selective instrumentation of the application for data collection.
- *Build*. These define activities that must take place when the application is built, and may be transparent to the user.
- *Launch*. This defines how the instrumented application is launched.

- *Data Management*. This defines how tool generated data is managed and accessed.
- *Visualization*. This defines how tools can provide their own browser or GUI in order to visualize or analyze data, or make use of Eclipse views directly.

By specifying corresponding actions for each integration point, the tool developer can easily incorporate their tool into the Eclipse development workflow in a manner that is transparent to the developer.

E. Remote Development Tools

Although PTP supports remote application monitoring, execution, and debugging, until recently the application source code and build environment was restricted to the local workstation on which Eclipse was running. However, with the recent addition of the Remote Development Tools (RDT) to PTP, all aspects of an application can be managed remotely. Although still in an early development stage, RDT promises to significantly enhance the development of complex parallel and scientific applications. It does this by allowing developers to continue to utilize the existing development platforms they are currently using, but at the same time to take advantage of the features provided by PTP without requiring any changes to the application build environment.

RDT introduces the notion of a *remote project*, which appears in the Eclipse workspace as if it was local to the developer's workstation, but the application actually resides on a remote machine. In order to operate on the remote application, RDT defines a number of *services* which can be independently configured to provide selective remote operations. The three primary services that are currently supported by RDT are:

- *Remote File Service*. This service is used to access source files that reside on the remote system to support local editing and file copy operations.
- *Remote Indexing Service*. This service is used to provide parsing and index generation on the remote system in order to avoid the cost of accessing every source file across the network.
- *Remote Build Service*. This service enables the application to be built remotely by executing an arbitrary build command on the remote system. Currently only projects built in this way can be used with RDT, the CDT "managed projects" are not supported in this release.

When used in conjunction with PTP's remote execution and debug facilities, RDT allows the developer to transparently undertake the full range of development activities on projects the completely reside on a remote system.

IV. A SIMPLE CASE STUDY

In the following section, we will present a simple case study on using PTP for developing an MPI application. This will include describing the steps necessary to import and configure an existing MPI application, locate a potential deadlock situation, then launch the application under debugger control. The code we chose for this example is the freely available discrete Fourier transform code, FFTW⁵, containing about 75,000 lines of source code and nearly 500 source files.

⁵Available from <http://www.fftw.org>

A. Importing

Eclipse offers a range of methods for importing an existing application so that it can be developed using PTP. These include: making a copy of the code into the workspace; linking to an external project; or checking out the code from a source code repository. The particular method chosen will depend on the developers environment. In this case, we didn't have access to a source code repository, and downloaded the code from the web as a gzipped tar file. After unpacking to source into a temporary directory, the code was imported into Eclipse. As FFTW already provides its own build system, we first created an empty unmanaged type of *C Project*, known as a *Makefile project*. Right-mouse clicking on this project provides access to the **Import...** menu. We then selected the **File System** import wizard and selected the directory that contained the unpacked source code. At this point the files were copied into the workspace ready for use.

Eclipse is scalable enough to support very large projects (thousands of files, millions of lines of code). Activities such as indexing the source code (used for advanced searching, content assist, type and call hierarchy views) are potentially long running, so automatically take place in the background without affecting the developer.

B. Configuring

Like many open source projects, the FFTW code uses the `autoconf` package to configure the build for a particular architecture. Eclipse doesn't provide explicit support for `autoconf`, but will still work with this type of project. In order to create the `Makefile` required to build the code, the `configure` script must first be run. This can be done either from the command line (by changing to the project directory and typing `./configure`) or from within Eclipse (by creating an **External Tools** launch configuration to run the `configure` script.) In order to build FFTW with MPI support, the `--enable-mpi` option had to be passed to the `configure` script. We also prefixed the command with `CFLAGS=-g` to ensure that debugging information was included in the executable.

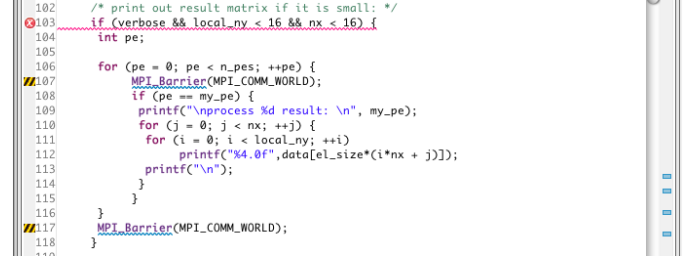
Once the appropriate `Makefiles` have been created, the only remaining action required to complete the configuration is to create a *make target* in order to run the build command. This is achieved by right-mouse clicking on the 'fftw' project in the **Make Targets** view and selecting **Add Make Target** from the context menu. We named the target 'build fftw' and left the **Make Target** field blank in order to run the `make` command with no arguments.

C. Analyzing

The barrier analysis tool is invoked on the project using a special **Parallel Analysis** menu on the Eclipse *toolbar*⁶. The analysis will scan all source code in the project and compute the barrier sets. Markers indicating the location of potential errors will be placed on corresponding source files and when the source file is opened, at the source line location in the file. The developer can then use this information to correct a potential deadlock situation.

⁶The toolbar provides quick access to commonly used functions via a series of icons at the top of the Eclipse window.

In the case of the FFTW code, we had no idea if there were any MPI barriers used in the code (although it seemed likely) or where they were located, so we simply ran the barrier analysis on all the source files. This was done by selecting the top-level directory, and then the **MPI Barrier Analysis** command from the **Parallel Analysis** menu. The result of this analysis was the discovery of six barrier statements in a single source file, and one potential deadlock situation. Figure 6 shows the source code annotated with markers indicating the location of the barrier statements and the potential error.



```
102 /* print out result matrix if it is small: */
103 if (verbose && local_ny < 16 && nx < 16) {
104     int pe;
105
106     for (pe = 0; pe < n_pes; ++pe) {
107         MPI_Barrier(MPI_COMM_WORLD);
108         if (pe == my_pe) {
109             printf("\nprocess %d result: \n", my_pe);
110             for (j = 0; j < nx; ++j) {
111                 for (i = 0; i < local_ny; ++i)
112                     printf("%4.0f", data[el_size*(i*nx + j)]);
113                 printf("\n");
114             }
115         }
116     }
117     MPI_Barrier(MPI_COMM_WORLD);
118 }
```

Fig. 6. Annotated source code showing the location of barrier statements and a potential barrier deadlock.

While this potential deadlock was only evident in a test harness, it nevertheless demonstrates the value of this kind of analysis, and also the difficulty in locating such errors. In this case, the error was not immediately obvious, even after inspecting the code. None of the variables used in evaluating the `if` statement at line 103 (`verbose`, `local_ny`, or `nx`) appear to be dependent on the rank of the individual processes, so it was not obvious why this might be an error. However, a more detailed inspection of the code revealed that the value of `local_ny` is computed in `transpose_mpi_get_local_size` in a different source file. This function calculates the size of the local segment of the array which varies depending on the process rank. The result is that it may be possible for some processes to execute the body of the `if` statement, while others do not, and so miss the barrier statement at line 117.

D. Building

Eclipse projects can be configured to automatically build each time an editor change is saved, or by manually invoking the build command from an Eclipse menu. While the build is running, the developer is able to continue to modify the source, perform analysis, or undertake other activities that are not dependent on the build completing. Build progress is displayed in a special *progress view*, that provides an estimate of the percentage completed. Detailed output from the build is available in the *console view*. If any errors are detected by the compiler or linker steps, the build will terminate, and a list of the errors will be displayed in the *problems view*. Error markers will also be placed on source files and displayed in the editor.

In order to build the FFTW project, it was necessary to invoke the make target that we created previously. This was done by switching to the **Make Targets** view and double-clicking on the 'build fftw' target. The build completed successfully with only one error, the previously discovered barrier error, still evident.

E. Launching

Once the build has completed, the developer must create a launch configuration to run the application. A single configuration is used for both running and debugging, and specifies the attributes needed to launch the application, such as the resource manager, executable name, command line arguments, environment variables, etc. These attributes are saved in the configuration, so they only need to be specified once. After creating the configuration, the application can be run or debugged by clicking a single button on the toolbar.

For FFTW, we created a new launch configuration to run the `test_transpose_mpi` program with arguments '100 90 10' corresponding to a 100x90 matrix containing 10 elements. We had previously configured a resource manager to control a remote Linux machine, and used this to start a four process job from our workstation. The job completed successfully, and output was visible from the process with MPI rank 0.

F. Debugging

In order to debug the application, no further configuration was necessary. A single button click is all that is required to invoke the debugger, and Eclipse automatically switches to display views for controlling the application (e.g., single stepping), examining stack frame location, viewing variables, etc. Breakpoints can be set directly in the source code editor view by clicking on the left edge of the view. Once the debug session is completed, the developer can switch back to the runtime and editor views with a single click.

We again launched the same four process job, but this time under the control of the debugger. Placing a breakpoint at line 103 allowed us to observe that the value of `local_ny` for each process was 23, 23, 23, and 21 respectively, showing that the values *do* vary based on the process rank. Although we did not attempt to identify the specific cases where a deadlock would occur, we were able to verify that the deadlock situation was a feasible one.

V. FUTURE WORK

There are many aspects of parallel application development for both peta-scale and the emerging multi-core systems that still remain major challenges. As mentioned before, the current programming models are unlikely to be adequate for applications designed to run on peta-scale systems, and much more powerful tools will be required to optimize performance for the next generation of heterogeneous hardware. If multi-core systems are going to become the performance panacea, then application developers will need programming models and languages that are as simple and easy to understand as those being used today. There are also many issues relating to the interaction between developers and the systems for which they are developing parallel applications. Eclipse and PTP are now well placed to begin addressing many of these challenges. In the following sections, we briefly examine a few key areas where future development of PTP appears promising.

A. Analysis Tools

The barrier and concurrency analysis tools provided by PTP were chosen to address some immediately obvious sources of errors, however there is significant scope for expanding the types of analysis that can be undertaken. There are also

a number of tools already available that provide analysis information derived from running the application, such as trace and profile information, and that could be used to augment the static analysis to provide greater insights into program operation. In addition, there are opportunities to better utilize compiler generated information to assist in the application development process. One such tool being actively developed will use compiler generated parallelization analysis to display the analysis results to the user, and possibly aid the developer in parallelizing selected code regions.

B. Performance Tools

PTP provides an external tools framework for integrating performance tools with Eclipse, however this is only a small part of the functionality required to support integrated performance analysis and optimization of parallel applications. Ideally, the developer should be able to invoke a performance analysis tool as easily as launching or debugging the application, have the data automatically collected and analyzed, and the results used to annotate the source code. The Tuning and Analysis Utilities (TAU) have already been integrated with PTP, and a number of other performance tools groups are also exploring Eclipse as a delivery platform. However, there is still much work to do to ensure that performance tools can be easily and effectively used as part of the development workflow.

C. Multi-core Tools

The current PTP tool set has been targeted primarily at distributed memory architectures and programming models (with the exception of OpenMP), however there is a growing requirement for tools to ease the transition from existing architectures to multi-core systems. At least three kinds of tools could benefit these applications: tools to aid in parallelizing sequential applications in order to make better use of the increased compute resources; performance analysis tools specifically targeting applications running on multi-core systems; and debugging tools that better manage the extra complexity introduced by multi-core architectures.

D. Languages and Programming Models

A variety of efforts are underway to develop new languages and programming models for parallel computing. In addition to the DARPA HPCS Language Project, there are also projects aimed at enhancing existing languages, such as UPC, Co-Array Fortran (CAF) [19], and Titanium [20], that add new functionality to better support parallel programming. New programming models, such as Asynchronous Partitioned Global Address Space (APGAS), on which IBM's X10 language [21] is based, are being developed. There is work under way to add support for PGAS-style languages and programming models to PTP, but there is still much development required to add support for the languages themselves.

E. Refactoring

Refactoring, or source-to-source transformations that preserve behavior, is emerging as a significant solution to many issues facing parallel programmers. This ranges from enabling legacy codes to take advantage of new language features (e.g. refactoring Fortran 77 codes to add Fortran 95 features), adding data parallelism (e.g. adding UPC directives to C code), to performance optimizations through source transformations

(e.g. cache optimization). Enabling refactoring requires access to compiler front-end infrastructure, as well as a framework for specifying refactoring algorithms. Eclipse provides both of these, but significant development is still required to implement the kinds of refactorings that would be beneficial to parallel application developers.

F. Debugging Methodologies

The existing interactive debugging methodology for parallel applications is not significantly different from that used for sequential applications. However, as the size of applications increases to peta-scale and beyond, it is not clear that this methodology will remain effective. In particular, if applications comprise millions of concurrently executing tasks, just identifying which tasks are the source of the errors is likely to become a highly challenging activity. The rich user interface and extensibility of Eclipse provides an exciting opportunity to investigate new techniques for analyzing, locating, and correcting errors in parallel programs. In addition, the integrated nature of PTP now provides opportunities to combine performance analysis and debugging tools into a single performance debugging paradigm.

VI. CONCLUSION

The quest for greater hardware performance is driving a significant change in the application development landscape. Both the scientific and mainstream computing communities are facing the challenge of developing parallel applications that are able to extract maximum performance from the new hardware. There is no doubt that new tools, languages, and programming models will be needed to assist the developer to reach this goal.

Although a number of integrated parallel tool environments have been developed in the past, none are still in wide use today. It's possible to speculate on the reasons for this, but one factor is clear: none have been based on a framework that enjoys the enormous popularity and the advanced features of the Eclipse platform. In addition to an open, portable and robust platform, Eclipse also provides an extensive array of advanced tooling that can be used by tool developers to create an integrated solution to a wide array of programming activities. The Parallel Tools Platform builds on this solid foundation, and provides an additional framework for developing and integrating tools for developing parallel applications. Currently, PTP provides a range of tools that provide advanced error checking, static analysis, runtime monitoring and control, and debugging services.

In addition to the existing tools, there are a number of efforts underway to improve the range of tools and functionality that PTP provides. This includes extending the analysis support to encompass dynamic analysis, and better integration for performance analysis tools. There are also active projects to enhance the ability of Eclipse to work in distributed development environments, and to improve the refactoring support that is available for existing programming languages.

PTP is still a very young project, and there are many opportunities for improving the capabilities to suit the advances in computing technology that will be introduced over the next few years. The integrated nature of the platform also offers scope for developing new tools, that may have not been

possible in the past, to deal with programming challenges that will be faced by both the peta-scale and many-core communities.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the efforts of many contributors without whom the Parallel Tools Platform would not exist. This includes the Eclipse Foundation, Los Alamos National Laboratory, Monash University, IBM Corporation, University of Oregon, Oak Ridge National Laboratory, and Technische Universität München, along with the many individuals who have shared their ideas and suggestions.

This material is partly based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

REFERENCES

- [1] "MPI: A Message Passing Interface Standard," <http://www.mpi-forum.org>, June 1995.
- [2] "UPC Language Specification v1.2," 2005.
- [3] A. Frazer, "CASE and its Contribution to Quality," *The Institution of Electrical Engineers, London*, 1993.
- [4] M. J. Granger and R. A. Pick, "Computer-aided Software Engineering's Impact on the Software Development Process: An Experiment," in *Proceedings of the 24th Hawaii International Conference on System Sciences*, January 1991, pp. 28–35.
- [5] P. H. Luckey and R. M. Pittman, "Improving Software Quality Utilizing an Integrated CASE Environment," in *Proceedings of the IEEE National Aerospace and Electronics Conference*, May 1991, pp. 665–671.
- [6] R. J. Norman and J. F. N. Jr., "Integrated Development Environments: Technological and Behavioral Productivity Perceptions," in *Proceedings of the Annual Hawaii International Conference on System Sciences*, January 1989, pp. 996–1003.
- [7] "Eclipse - An Open Development Platform," <http://www.eclipse.org>.
- [8] B. Q. Brode and C. R. Warber, "DEEP: A Development Environment For Parallel Programs," in *Proceedings of the International Parallel Processing Symposium*, 1998, pp. 588–593.
- [9] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon, "ParaScope: A Parallel Programming Environment," in *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987.
- [10] J. Cownie, A. Dunlop, S. Hellberg, A. J. G. Hey, and D. Pritchard, "Portable Parallel Programming Environments - The ESPRIT PPPE Project," *Massively Parallel Processing Applications and Development*, Netherlands, June 1994.
- [11] P. Kacsuk, J. C. Cunha, G. Dózs, J. L. co, and et. al., "A Graphical Development and Debugging Environment for Parallel Programs," *Parallel Computing*, vol. 22(13), pp. 1747–1770, February 1997.
- [12] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, "Parallel Programming in OpenMP," *Morgan Kaufmann*, 2000.
- [13] L. Hockstein and V. R. Basili, "The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development," *IEEE Computer*, vol. 41(3), pp. 50–58, March 2008.
- [14] E. Lusk and K. Yelick, "Languages for High-Productivity Computing: The DARPA HPCS Language Project," *Parallel Processing Letters*, vol. 17(1), pp. 89–102, 2007.
- [15] "Eclipse Documentation - Latest Release," <http://help.eclipse.org/help33/index.jsp>, July 2007.
- [16] G. R. Watson, "A Model Based Framework for the Integration of Parallel Tools," in *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, September 2006.
- [17] Y. Zhang and E. Duesterwald, "Barrier matching for programs with Textually Unaligned Barriers," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2007, pp. 194–204.
- [18] Y. Lin, "Static Non-concurrency Analysis of OpenMP Programs," in *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, June 2005.
- [19] R. Numrich and J. Reid, "Co-Array Fortran For Parallel Programming," *ACM Fortran Forum*, vol. 17(2), pp. 1–31, 1998.
- [20] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A High-Performance Java Dialect," *Concurrency: Practice and Experience*, vol. 10, pp. 825–836, 1998.
- [21] "The X10 Programming Language," <http://x10-lang.org>.