Parallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming

Hung-Hsun Su Max Billingsley III
Alan D. George
High-performance Computing and Simulation (HCS) Research Lab
Dept. of Electrical and Computer Engineering
University of Florida, Gainesville, Florida 32611-6200
{su,billingsley,george}@hcs.ufl.edu

Abstract

Given the complexity of parallel programs, developers often must rely on performance analysis tools to help them improve the performance of their code. While many tools support the analysis of message-passing programs, no tool exists that fully supports programs written in programming models that present a partitioned global address space (PGAS) to the programmer, such as UPC and SHMEM. Existing tools with support for message-passing models cannot be easily extended to support PGAS programming models, due to the differences between these paradigms. Furthermore, the inclusion of implicit and one-sided communication in PGAS models renders many of the analyses performed by existing tools irrelevant. For these reasons, there exists a need for a new performance tool capable of handling the challenges associated with PGAS models. In this paper, we first present background research and the framework for Parallel Performance Wizard (PPW), a modularized, event-based performance analysis tool for PGAS programming models. We then discuss features of PPW and how they are used in the analysis of PGAS applications. Finally, we illustrate how one would use PPW in the analysis and optimization of PGAS applications by presenting a small case study using the PPW version 1.0 implementation.

Keywords Performance analysis tool, partitioned global address space, UPC, SHMEM, profiling, tracing.

1 Introduction

To meet the growing demand for greater computing power, new shared-memory machines and clusters are constantly being built. In order to take advantage of these powerful systems, various parallel programming models have been developed, ranging from message-passing to partitioned global-address-space models. Given the added complexity (compared to sequential programming) of these parallel models, users often must rely on performance analysis tools (PATs) to help them improve the performance of their application. Among the available programming models, the Message Passing Interface (MPI) has received the majority of PAT research and development, as it remains the most well known and widely used parallel programming model. Almost all existing parallel PATs support MPI program analysis. These include tools such as HPCToolkit [1], KOJAK [2], MPE/Jumpshot [3], Paradyn [4], and TAU [5].

Recently, SPMD-based models providing the programmer with a partitioned global address space (PGAS) abstraction have been gaining popularity, including models such as Unified Parallel C (UPC) [6], SHMEM, Co-Array Fortran [7], and Titanium [8]. By extending the memory hierarchy to include an additional, higher-level global memory layer that is partitioned between nodes in the system, these models provide an environment similar to that of threaded sequential programming. This global memory abstraction is especially important in cluster computing, as it no longer requires programmers to manually orchestrate message exchange between processing nodes. Instead of the explicit, two-sided data exchange (i.e. send, receive) required by message-passing models[1], PGAS models allow for explicit or implicit one-sided data exchange (i.e. put, get) through reading and writing of global variables. These one-sided communication operations greatly reduce the complexity of data management from a programmer's perspective by eliminating the need to carefully match sends and receives between communicating nodes. Furthermore,

[1] MPI-2 does support one-sided operations, but with strong usage restrictions that limit their usefulness (e.g. no concurrent writes are allowed from different nodes to separate locations in the same window).

PGAS models can enable the development of new algorithms for large systems that are otherwise too complex to program under a message-passing environment [9].

However, the PGAS abstraction does force the programmer to give up some control over the communication between processing nodes, which can lead to a reduction in performance (due to increases in the likelihood of undesired program behavior). This situation is especially true in a cluster environment when inter-node operations are expensive compared to local operations. Therefore, PAT support for PGAS models becomes even more critical to the programmer. Unfortunately, no tool exists that fully supports the analysis of programs written using PGAS models.

In this paper, we present our work on Parallel Performance Wizard (PPW), a new parallel performance analysis tool that supports partitioned global-address-space program analysis. The remainder of the paper is organized as follows: Section 2 provides a brief overview of existing performance analysis tools. In Section 3 we present background research, and in Section 4 we introduce the high-level framework for PPW. We then discuss features of PPW and their usage in PGAS application analysis in Section 5. Next, in Section 6, we demonstrate how PPW can be used to optimize PGAS programs by presenting a case study. Finally, we conclude the paper and give future directions for PPW research and development in Section 7.

2 Overview of Performance Analysis Tools

There are numerous performance tools based on the experimental measurement approach available for sequential and parallel programming [10]. In this approach, programs are executed and performance data are gathered through sampling or event recording under tracing or profiling mode. In tracing mode, instances of performance data are recorded separately from one another, while in profiling mode only statistical information is kept. Figure 1 illustrates the typical stages in the experimental measurement approach. In this cycle, user code is first instrumented by the tool to provide some entry point for performance data to be collected. This instrumented version is then executed and event data are measured and recorded during runtime. Based on these data, the tool then performs various data processing and automatic analyses, and the result is presented to the user through a text-based or graphical interface. Finally, optimizations are made to the original code and the whole cycle repeats until an acceptable level of performance is reached.

In terms of model support, tools supporting sequential C and MPI are the most common, with some also supporting shared-memory programming models such as OpenMP. Few tools have any support for SHMEM or UPC, and tools with basic support for these models are typically

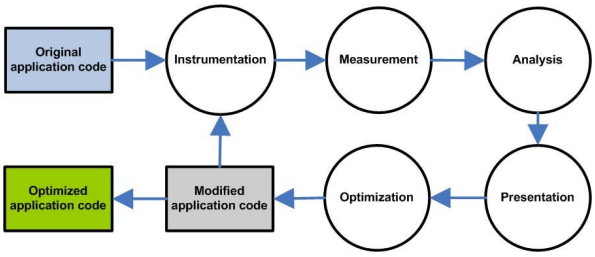

Figure 1. Experimental measurement stages

not portable and can only perform simple analyses. Many performance tools use the Performance Application Programming Interface (PAPI) [11], which provides platform-independent access to hardware counters on a variety of platforms. In addition, some tools, such as KOJAK and Paradyn, also attempt to automatically identify performance bottlenecks and quantify their impact on performance.

3 Background Research

In order to develop a suitable framework to fully support PGAS models, we conducted several studies, including an evaluation of existing performance analysis tools [12], an examination of PGAS programming models, and a study of tool usability. Based on the findings from these studies, we were able to identify several key characteristics and techniques common to successful PATs that are also applicable to PGAS models. These findings will be referenced throughout the remainder of this paper.

1. It is critical for a successful tool to provide source-code correlation so a user can easily associate performance data with actual code.

2. Profiling data allow users to quickly identify possible areas to focus their tuning efforts, while tracing data provide detailed information that is often necessary to determine the cause of performance degradation. Both of these types of data are useful in their own right.

3. The wrapper approach for instrumentation commonly found in MPI tools works well with SHMEM libraries, but would be inadequate for compiler-based implementations of other PGAS models. A new instrumentation method is needed to support PGAS models.

4. The inability of most existing tools in tracking and analyzing implicit one-sided communication remains one of the biggest roadblocks in extending them to support PGAS models. A new model is needed to handle implicit one-sided communication.

5. Although PATs are effective in troubleshooting performance problems, they are often too difficult to use and as a result are not used by programmers. An intuitive and easy-to-learn user interface is often the most critical quality of a productive tool. In addition, the tool should automate the optimization process as much as possible to alleviate the amount of effort needed from the user.

4 PPW Framework

Figure 2 shows the framework of Parallel Performance Wizard, which consists of modules corresponding to the stages of the experimental performance analysis cycle. While the overall design of our framework is similar to that of other performance tools, there are a few key differences in the design to accommodate the goal of supporting multiple PGAS models in a single tool. Note that the optimization stage is not included in the framework because automatic code optimization is still not practical with current technologies.

One important, novel aspect of the design of this framework is the use of generic operation types instead of model-specific constructs whenever possible. This approach is vital in providing support for multiple PGAS models successfully while minimizing the number of components needed. For each supported programming model, the *Event Type Mapper* maps the model-specific constructs to their appropriate generic operation types (e.g. `upc_memget` maps to one-sided get, `shmem_barrier` maps to all-to-all, etc.). Once this classification has been made, the majority of the components in PPW then work with the generic operation types and consult the *Event Type Mapper* only when model-specific information is needed. As a result, only a single implementation of a component is often needed, which significantly reduces the effort needed to support multiple programming models. Currently, PPW includes the following operation types applicable to existing PGAS models: startup, termination, barrier, collective, point-to-point synchronization, one-sided put, one-sided get, non-blocking operation synchronization, and work sharing. The following paragraphs describe each component in the framework in more detail.

The *Instrumentation Units* deal with how to enable recording of performance data and specify when these data are gathered (i.e. what is considered an event and how it will trigger the recording of data). Due to the differences in techniques used in the development of PGAS programming environments, no single existing instrumentation technique works well for all PGAS model implementations[2]. Source instrumentation, in which entry points to data collection are added in the original source code, may prevent

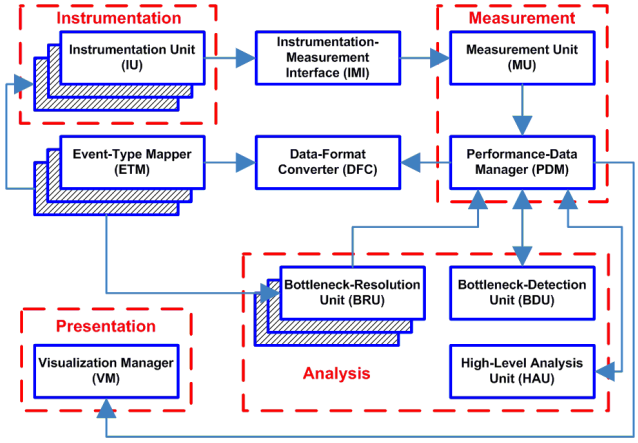

Figure 2. The high-level framework of PPW organized with respect to the stages of experimental measurement

compiler optimization and lacks the means to handle a relaxed memory model. Binary instrumentation, in which the compiled binary is modified to facilitate data collection, is not available on some architectures, and it is often difficult to relate the data back to the source code. Finally, the intermediate library approach (such as PMPI), in which library functions are wrapped with versions invoking measurement code, does not work for "pure" compilers, such as GCC UPC, that translate parallel constructs directly into byte code.

To accommodate the many instrumentation techniques appropriate for various PGAS model implementations, without introducing multiple measurement components, we proposed an *Instrumentation-Measurement Interface* called the Global Address Space Performance (GASP) interface [14]. Such an interface shifts the responsibility of where to add instrumentation code and how to obtain source code information (finding 1) from the tool writer to the compiler writer, which improves the accuracy of the data gathered. The simplicity of the interface minimizes the effort required from the compiler writer to add PAT support to their system. In addition, a PAT can quickly add support for any PGAS compiler which includes an implementation of the GASP interface by simply defining the body of a generic event notification function (findings 1, 3, 5). We were able to demonstrate that this is a suitable approach for PGAS models with the Berkeley UPC GASP implementation[3] (for which we observed profiling overhead $< 3\%$ and tracing overhead $< 5\%$).

The *Measurement Unit* deals with what type of data to record and how to acquire the data accurately. Like many

[2] See [13] for more information on various instrumentation techniques.

[3] Developed by the Berkeley UPC group for BUPC version 2.4+ [15].

Table 1. Example events recorded for each generic operation type

Operation Type	Event of Interest[4]
all-to-all	time, system size (num nodes)
point-to-point	time, sender, receiver
one-sided data transfer	time, sender, receiver, data size
all other	time

existing PATs, this unit supports both tracing and profiling of data and allows the recording of PAPI hardware counters (finding 2). However, instead of dealing with model-specific constructs as existing PATs do (which often leads to multiple measurement units being needed for multiple model support), this unit works with a generic data format. This format, which is based on generic operation types defined in the *Event Type Mapper*, describes a set of events of interest to record for each operation type (Table 1). Using this generic data format, additional model support requires no modification of the *Measurement Unit* as long as the constructs of the new model can be associated with the existing generic operation types of the *Event Type Mapper*.

Once the performance data are collected by the *Measurement Unit*, they are passed to the *Performance Data Manager*. The primary responsibilities of this unit include access and storage of data, merging of data, and simple post-processing of data (e.g. calculating averages). Like the *Measurement Unit*, this data manager operates on the generic data format to support multiple models through a single component implementation.

The analysis module aims to provide the tool with semi-automatic program analysis capability. A new bottleneck detection and resolution model based on the generic operation types is currently being developed (the details of which are beyond the scope of this paper). The roles of each of the three units in the analysis module are briefly mentioned here. The *Bottleneck Detection Unit*, which operates on the generic operation types, is responsible for the identification of potential performance bottlenecks in a given application. Once these are identified, the model-specific *Bottleneck Resolution Unit* then tries to provide additional insights into what caused these performance degradations and possibly provide suggestions on how to remove them from the application. Since resolution techniques can be different for particular models even for the same type of operation (e.g. a technique to fix the performance degradation stemming from `upc_memget` versus from `shmem_get` could be different even though they are both classified as one-sided get operations), multiple model-specific resolution units are needed. Finally, the *High-Level Analysis Unit* provides analyses applicable to the entire program, such as critical-path analysis, scalability analysis, and load-balancing analysis.

The *Data Format Converter* unit is solely responsible for converting PPW performance data into other established performance data formats. Again, only a single component implementation is needed because of the use of generic operation types (e.g. converting `upc_memput` and `shmem_memput` to another format is essentially the same, since the data recorded for both is the same).

The *Visualization Manager* provides the user-friendly, graphical interface of the tool (finding 5). It is responsible for generating useful visualizations from the performance data gathered by the *Measurement Unit* or generated by the analysis module. Version 1.0 of PPW includes traditional visualizations such as charts and tabular views of profile data, a graphical data-transfer view, and a novel shared-array distribution diagram (see Section 4 for more information regarding these), all with source code correlation when appropriate (finding 1). In addition, viewing of trace data in a timeline format is achieved by exporting through the *Data Format Converter* to viewers such as Jumpshot and Vampir [16]. With these, we were able to avoid replicating the capabilities of these full-featured timeline viewers, while bringing these capabilities to the world of PGAS programming.

5 PPW Features and Usage

In this section, we present several features of PPW and outline how each is used in the optimization process. Many of these features are borrowed from existing tools, helping minimize the time needed to learn a new tool. Where appropriate, new features have been developed as needed for analyzing the performance of PGAS programs.

5.1 Semi-Automatic Instrumentation

One of the key features of PPW is the ability to enable performance data collection with little or no user input. Through GASP implementations, all system events (user functions and parallel model constructs) are automatically instrumented by the tool (finding 5). In addition, users are free to track the performance of any region of the program, as GASP supports user-defined events that can be manually defined anywhere in the application. During execution time, PPW then automatically collects the data for all system and user-defined events under trace or profile mode (as specified by the user). At the same time, PPW also captures information regarding the execution environment such as op-

[4]Note that PPW can record PAPI counter events for any type of operation, though they are mainly used for local analysis.

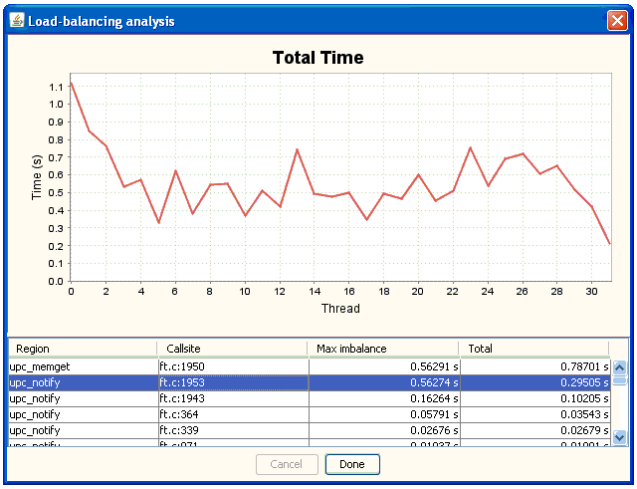

Figure 3. PPW Region-based Load Balancing Analysis visualization to facilitate the identification of load imbalance issues

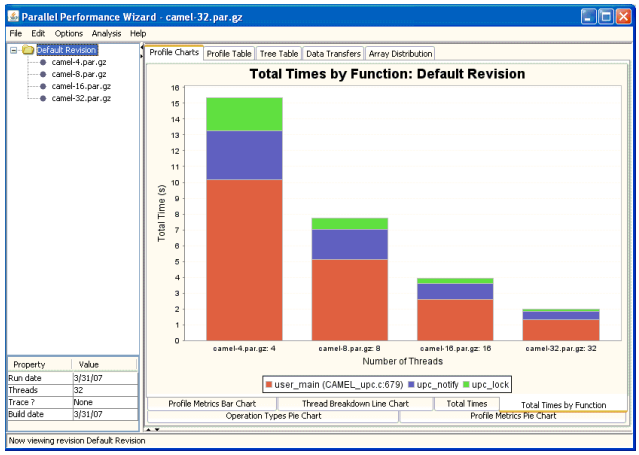

Figure 4. PPW Experimental Set Comparison chart providing side-by-side comparison of data files loaded

timization flags used, machine hostnames, etc. These data are then processed and used to create various visualizations described in the following sections.

5.2 Visualizations for Program Performance Overview

Amdahl's Law suggests that to maximize the performance gain from optimization, one should concentrate on optimizing the most time-consuming parallelizable part of the program. As a result, many programmers begin performance analysis by identifying application regions that took the longest time to run. Once identified, programmers then concentrate their effort on finding performance issues related to these regions. To this end, PPW provides charts and diagrams to facilitate the identification of time-consuming application segments (for example, one of the pie charts included in PPW shows the breakdown of time spent in the 10 longest running regions with respect to the total program execution time).

5.3 Statistical Data Tables

Like other successful tools, PPW displays high-level statistical performance information in the form of two tables that show statistical data for all regions of a program. The Profile Table reports flat profile information while the Tree Table (Figure 7) separately reports the performance data for the same code region executed in individual callpaths. Selecting an entry in these tables highlights the corresponding line in the source code viewer below the tables, allowing the user to quickly associate performance data and corresponding source code (finding 1). These statistical data, such as total time, percentage of whole program, average time, minimum time, maximum time, and number of calls, are useful in identifying specific code regions with long execution time. Moreover, the user can use them to identify potential bottlenecks. A region with the minimum or maximum time far below or exceeding its average execution time could potentially be a bottleneck in the application. An example is when particular instances of the region's execution took much longer than others to complete, which suggests that these instances were not executed optimally (PPW also provides a Load Balancing Analysis visualization to detect this behavior, as shown in Figure 3). In addition, if the user has some idea of the region's expected execution time (e.g. `shmem_barrier` takes around 100μs to execute when no bottleneck exists), he or she can quickly identify potential bottleneck regions if the actual average execution time far exceeds the expected value.

5.4 Multiple Data Set Support

PPW supports the loading of multiple performance data sets at once, a feature that proves useful when working with multiple related experimental runs. Included also in PPW is the Experiment Set Comparison chart (Figure 4) that the user can use to quickly compare the performance of data sets loaded. For example, when the user loads the data sets for the same program running on different system sizes, he or she can quickly determine the scalability of the program and identify sequential and parallel portions of the program. Another use is to load data sets for different revisions of the

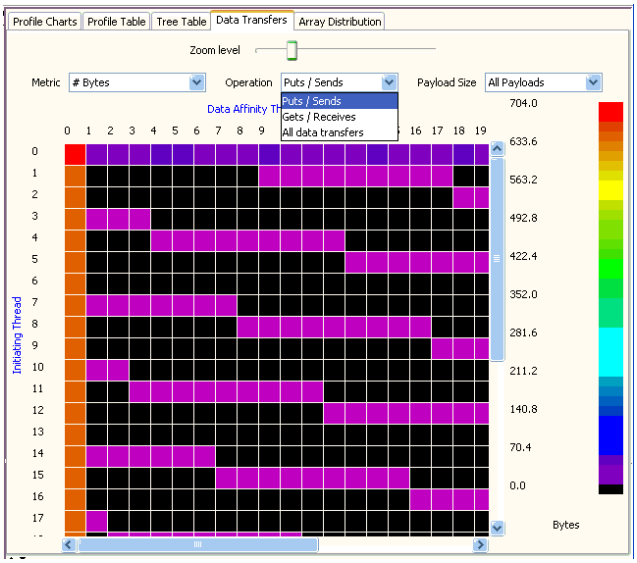

Figure 5. PPW Data Transfers visualization showing the communication volume between processing nodes for put operations

Figure 6. PPW Array Distribution visualization showing the physical layout of a 7x8 array with blocking factor of 3 on a system with 8 nodes

program running on the same system size and compare the effect of the revisions on performance.

5.5 Communication Pattern Visualization

In many situations, especially in cluster computing, significant performance degradation is associated with poorly arranged inter-node communications. The Data Transfer visualization (Figure 5) provided by PPW is helpful in identifying communication related bottlenecks such as communication hot spots. While this visualization is not unique to PPW, ours is able to show communications from both implicit and explicit one-sided data transfer operations in the original program. By default, this visualization depicts the inter-node communication volume for all data transfer operations in the program, but users are able to view the information for a subset of data transfers (e.g. puts only, gets with payload size of 8-16kB, etc.). Threads that initiate an inordinate amount of communication will have their corresponding blocks in the grid stand out in red. Similarly, threads that have affinity to data for which many transfers occur will have their column in the grid stand out.

5.6 Array Distribution Visualization

A novel visualization provided by PPW is the Array Distribution, which graphically depicts the physical layout of shared objects in the application on the target system (Figure 6). Such a visualization helps users verify that they have distributed the data as desired, a particularly important consideration when using the PGAS abstraction. We are also currently investigating the possibility of incorporating communication pattern information with this visualization to give more insight into how specific shared objects are being accessed. This extension would allow PPW to provide more specific details as to how data transfers behaved during program execution.

6 Case Study

In this section, we present a small case study conducted using PPW version 1.0 [17]. We ran George Washington University's UPC implementation [18] of the FT benchmark (which implements an FFT algorithm) from the NAS benchmark suite 2.4 using Berkeley UPC 2.4, which includes a fully functional GASP implementation. Tracing performance data were collected for the class B setting executed on a 32-node Opteron 2.0 GHz cluster with a Quadrics QsNetII interconnect. Initially no change was made to the FT source code.

From the Tree Table for the FT benchmark (Figure 7), it was immediately obvious that the `fft` function call (3rd row) constitutes the bulk of the execution time (9s out of 10s of total execution time). Further examination of performance data for events within the `fft` function revealed that `upc_barriers` (represented as `upc_notify` and `upc_wait`) in `transpose2_global` (6th row) appeared to be a potential bottleneck. We came to this conclusion from the fact that the actual average execution time

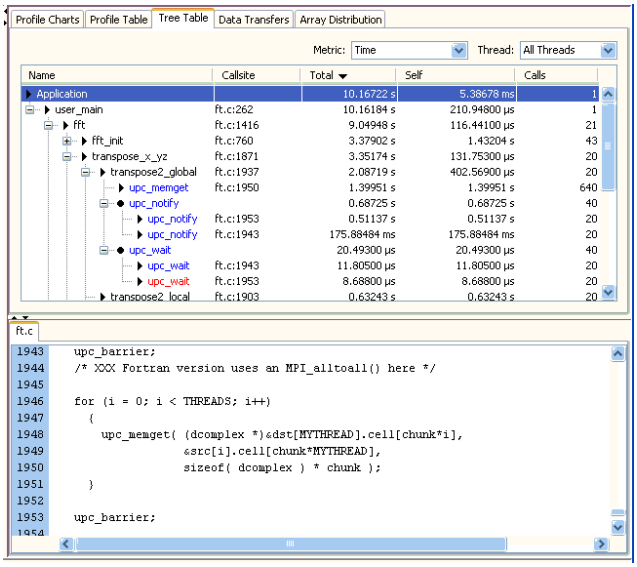

Figure 7. PPW Tree Table visualization for the original FT shown with code yielding part of the performance degradation

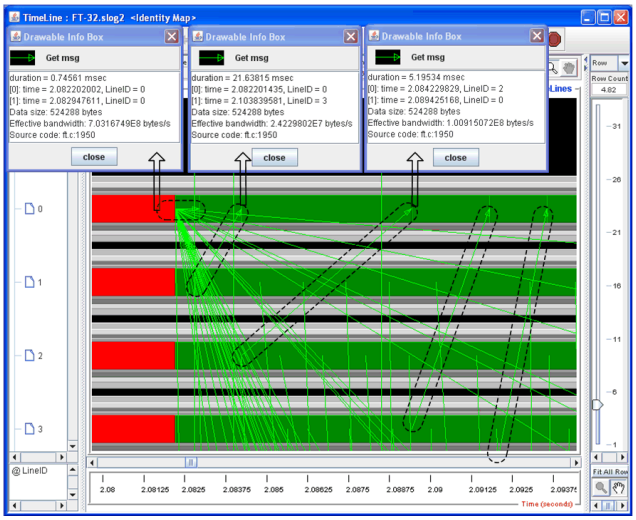

Figure 8. Annotated Jumpshot view of the original version of FT showing the serialized nature of `upc_memget`

for `upc_barrier` at lines 1943 (176 ms) and 1953 (0.5s) far exceeds the expected value of 2ms on our system for 32 nodes (we obtained this value by running a benchmark with no bottlenecks). Looking at the code between the two barriers, we saw that multiple `upc_memgets` were issued and suspected that the performance degradation might be related to these `upc_memgets`. However, we were unable to confirm this suspicion based on the data provided by the Tree Table alone, since we did not know the expected execution time for `upc_memget` for that payload size. More detailed performance data regarding these lines of code were needed.

We next converted the performance data from the PPW format into the Jumpshot SLOG-2 format in order to look at the behavior of `upc_barriers` and `upc_memgets` in a timeline format. With the viewer, we discovered that the `upc_barrier` at line 1953 was waiting for `upc_memget` calls to complete. In addition, we saw that `upc_memget` calls issued from the same node were unnecessarily serialized, as shown in the annotated screenshot of the Jumpshot viewer in Figure 8 (note the zigzag pattern for these memgets). Looking at the start and end time of the `upc_memget` calls issued from node 0 to all nodes in the system, we saw that the later `upc_memget` must wait for the earlier `upc_memget` to complete before initiating, even though the data obtained are not related.

The apparent solution to improve the performance of the FT benchmark was to replace the serialized version of `upc_memget` with a parallel one. Fortunately, Berkeley UPC includes such an asynchronous memget operation (`bupc_memget_async`). When the original `upc_memget` was replaced with the asynchronous version, we were able to cut down the execution time of the FT benchmark by 1.56s, an improvement of 14% over the original version.

In this small case study, we have shown how PPW was used to optimize a PGAS program. With little knowledge of how the FT benchmark works, we were able to remove a major bottleneck in the program within a few hours of using PPW.

7 Conclusions

As hardware technologies for parallel computing mature, so too does the development of programming models to support the execution of parallel applications. While such applications have the potential to achieve very high performance, this is often not realized due to the complexity of the execution environment. To tackle this problem, many parallel performance analysis tools have been developed to help users optimize their code, with most tools supporting the message-passing model. However, newer models such as those for partitioned global-address-space programming have very limited tool support. In this paper, we have outlined the bridging of this gap with the introduction of our PGAS performance analysis tool called Parallel Performance Wizard. We presented the high-level framework design of PPW based on generic operation types, and we then

discussed features and usage of these features provided by the tool. We also demonstrated the effectiveness of PPW in PGAS application analysis through a small case study using the version 1.0 release of the tool.

We are currently working to complete the development of the bottleneck detection and resolution model mentioned above and to improve the usability of the tool's user interface. In addition, we are in the process of extending the design and implementation of PPW to fully support a variety of additional programming models, including MPI and Reconfigurable Computing (RC) systems. Finally, we are also investigating the possibility of mixed-model analysis for applications written using multiple programming models.

Acknowledgments

This work was supported in part by the U.S. Department of Defense. We would like to acknowledge former members of the UPC group at UF, Adam Leko, Hans Sherburne, and Bryan Golden, for their involvement in the design and development of PPW. Also, we would like to express our thanks for the helpful suggestions and cooperation of Dan Bonachea and the UPC group members at UCB and LBNL.

References

[1] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. "HPCVIEW: A tool for top-down analysis of node performance". The Journal of Supercomputing, 23(1):81–104, August 2002.

[2] B. Mohr and F. Wolf. "KOJAK – a tool set for automatic performance analysis of parallel applications". European Conference on Parallel Computing (EuroPar), pages 1301–1304, Klagenfurt, Austria, LNCS 2790, August 26–29, 2003.

[3] A. Chan, W. Gropp, and E. Lusk. "Scalable log files for parallel program trace data(draft)", 2000.

[4] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. "The paradyn parallel performance measurement tool". IEEE Computer, 28(11):37–46, November 1995.

[5] S. S. Shende and A. D. Malony. "The Tau Parallel Performance System". International Journal of High-Performance Computing Applications (HPCA), 20(2):297–311, May 2006.

[6] The UPC Consortium, "UPC Language Specifications", May 2005. http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf

[7] B. Numrich and J. Reid, "Co-Array Fortran for Parallel Programming", ACM Fortran Forum, 17(2), pp. 1–31, 1998.

[8] 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", Workshop on Java for High-Performance Network Computing, Las Vegas, Nevada, June 1998.

[9] A. Johnson, "CFD on the Cray X1E Using Unified Parallel C", a PowerPoint presentation, 5th UPC Workshop, September 2005. http://www.gwu.edu/~upc/upcworkshop05/ahpcrcUPC_User_Forum.pdf

[10] L. DeRose, B. Mohr and K. London, "Performance Tools 101: Principles of Experimental Performance Measurement and Analysis," SC2003 Tutorial M-11.

[11] K. London, S. Moore, P. Mucci, K. Seymour, R. Luczak, "The PAPI Cross-Platform Interface to Hardware Performance Counters", Department of Defense Users' Group Conference Proceedings, Biloxi, Mississippi, June 2001.

[12] S. Shende, "The Role of Instrumentation and Mapping in Performance Measurement", Ph.D. Dissertation, University of Oregon, August 2001.

[13] A. Leko, H. Sherburne, H. Su, B. Golden, A.D. George. "Practical Experiences with Modern Parallel Performance Analysis Tools: An Evaluation". http://www.hcs.ufl.edu/upc/toolevaluation.pdf

[14] H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley III, and A. George, "GASP! A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models", Proc. of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06), Umeå, Sweden, June 18–21, 2006. http://gasp.hcs.ufl.edu/

[15] Berkeley UPC project website. http://upc.lbl.gov/

[16] Vampir tool website. http://www.vampir-ng.de/

[17] PPW Project website. http://ppw.hcs.ufl.edu/

[18] GWU UPC NAS 2.4 benchmarks. http://www.gwu.edu/ upc/download.html