Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global-Address-Space Applications

Hung-Hsun Su, Max Billingsley III and Alan D. George

Abstract
Given the complexity of high-performance parallel programs, developers often must rely on performance analysis tools to help them improve the performance of their applications. While many tools support analysis of message-passing programs, tool support is limited for applications written in programming models that present a partitioned global address space (PGAS) to the programmer such as UPC and SHMEM. Existing tools that support message-passing models are difficult to extend to support PGAS models due to differences between the two paradigms and the techniques used in their implementations. In this paper, we present our work on Parallel Performance Wizard (PPW), a performance analysis system for PGAS and MPI application analysis. We discuss new concepts, namely the generic-operation-type abstraction and GASP-enabled data collection, developed to facilitate support for multiple programming models and then give an overview of PPW’s automatic analysis and visualization capabilities. Finally, to show the usefulness of our system, we present results on PPW’s overhead, storage requirements and scalability before demonstrating its effectiveness via application case studies.

Keywords
Performance analysis tool, PGAS, UPC, SHMEM, GASP, generic-operation-type abstraction

1. Introduction
Parallel computing has become the dominant paradigm in high-performance computing. Over the years, there have been many parallel programming models, ranging from shared-memory models to message-passing models, developed so programmers can orchestrate the computation, communication, and synchronization between nodes of parallel architectures. However, due to the complexity of the models and parallel environments, achieving the desired parallel program performance is seldom guaranteed. Programmers often must undertake an iterative optimization process to improve the performance of their application to an acceptable level. Because this optimization process is cumbersome to perform manually, parallel performance analysis tools (PATs) were created to facilitate the process. Among the available parallel 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 model. Almost all existing parallel PATs support MPI program analysis to some degree.

Recently, newer parallel models providing the programmer with a partitioned global-address-space (PGAS) abstraction have been gaining popularity, including Unified Parallel C (UPC) (UPC Consortium, 2005), the SHared MEMory library (SHMEM) (see http://web1.quadrics.com/downloads/documentation/ShmemMan_6.pdf), Co-Array Fortran (CAF) (Numrich and Reid, 1998), Titanium (Yelick et al., 1998), and the upcoming Chapel (see http://chapel.cs.washington.edu) and X10 (see http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html) languages which are a part of the DARPA High-Productivity Computing Systems (Kepner and Koester, 2003) research effort. By extending the memory hierarchy to include the notion of a global shared memory which may be physically partitioned among nodes in the system, these models provide a simpler programming interface than the message-passing approach. Instead of the explicit, two-sided data exchange (i.e., send and receive operations) used in pure message-passing models (MPI 1.x), PGAS models allow for explicit or implicit one-sided data transfers (i.e., put and get operations) to take place through reading and writing global...
variables. This provision reduces the complexity of data management from a programmer’s perspective by eliminating the need to match send and receive operations and can facilitate the development of applications exhibiting fine-grained parallelism or using algorithms that are complex to program under a message-passing environment (Johnson, 2005). However, the PGAS abstraction requires programmers to handle new challenges (e.g., locating race conditions) and forces them to give up some control over the interaction between processing nodes. This loss of control increases the likelihood of a mismatch between the actual execution pattern and the one intended by the programmer, which can lead to an underperforming application, or worse, an application that does not work as intended. The performance impact of such a mismatch is most apparent in cluster environments, where inter-node operations are several orders of magnitude more expensive than local operations. For this reason, it is even more critical for PGAS programmers to have access to effective performance analysis tools to reap the benefits these models provide.

Unfortunately, tool support for PGAS models has been limited. Existing tools that support MPI are not equipped to handle several operation types provided by PGAS models, such as implicit one-sided communications and work-sharing constructs. In addition, the variety of PGAS compiler implementation techniques complicates the performance data collection process, making it difficult for existing tools to extend support to PGAS models. For these reasons, there exists a need for a new performance system capable of handling the challenges associated with PGAS models.

In this paper, we introduce Parallel Performance Wizard (PPW), a parallel performance analysis system that supports multiple PGAS programming models as well as MPI. The remainder of the paper is organized as follows. In Section 2, we provide an overview of performance analysis tools and PGAS programming models. In Section 3, we talk about our background research process and discuss key findings that shaped the development of the PPW system. Section 4 provides an overview of the PPW performance analysis system and introduces the generic-operation-type abstraction concept. In Section 5, we present our parallel programming event model based on generic operation types. Section 6 then discusses the PPW instrumentation and measurement approach using the new Global-Address-Space Performance (GASP) interface. In Section 7, we give an overview of the PPW automatic analysis system and data visualizations. Section 8 presents results on the overhead and storage requirements for PPW, along with tool scalability, and demonstrates the effectiveness of the system through two application case studies. Finally, we conclude the paper and give directions for future research in Section 9 and acknowledge parties instrumental in the creation of PPW in the acknowledgments.

2. Background

In this section we present an overview of performance analysis tools and partitioned global-address-space programming models.

2.1. Performance Analysis Tools

Performance analysis tools are software systems that assist programmers in understanding the runtime behavior of their application on real systems and ultimately in optimizing the application with respect to execution time, scalability, or resource utilization. To achieve this goal, the majority of tools make use of a highly effective experimental performance analysis approach, based on a measure–modify cycle in which the programmer conducts an iterative process of data collection, data analysis, data visualization, and optimization until the desired application performance is achieved (DeRose and Mohr, 2003). Under this approach, the tool first generates instrumentation code that serves as entry points for performance data collection (Instrumentation). Next, the application and instrumentation code are executed on the target platform and raw performance data are collected at runtime by the tool (Measurement). The tool organizes the raw data and can optionally perform various automatic analyses to discover and suggest resolutions to performance bottlenecks (Automatic Analysis). Both the raw and analyzed data are then presented in more user-friendly forms to the programmer through a text-based or graphical interface (Presentation) which facilitates the manual analyses process (Manual Analysis). Finally, the tool or programmer applies appropriate optimization techniques to the program or the execution environment (Optimization) and the whole cycle repeats until the programmer is satisfied with the performance level.

A tool can use (fixed-interval) sampling-driven instrumentation or event-driven instrumentation, depending on when and how often performance data are collected. In sampling-driven tools, data are collected regularly at fixed-time intervals by one or more concurrently executing threads. At each time step, a predefined, fixed set of metrics (types of performance data) are recorded regardless of the current program behavior (e.g., same types of data are recorded regardless of whether the program is performing computation or communication). The performance of the program is then estimated, often using only a subset of these metrics. In most cases, the monitoring threads access only a few hardware counters and registers and perform limited calculations, thus introducing a very low data collection overhead. As a result, this technique is less likely to cause changes in execution behavior that may lead to an inaccurate analysis of the program. However, sampling-driven tools typically have greater difficulty in presenting the program behavior with respect to the high-level source code, especially when the time interval is large enough to miss short-lived trends.

In contrast, event-driven tools record data only when specified events (such as the start of a function or communication call) occur during program execution. Together, events and metrics make up the event model that the tool uses to describe application behavior; the complete set of events and metrics is used to reconstruct the behavior of the program in direct relation with high-level source code, easing the analysis and optimization process. For each event, the tool records a select number of metrics (e.g., time, node,
etc.) relevant to that particular event but requires significantly more processing time than simply accessing a few hardware counters in the sampling-driven case. As a result, event-driven tools generally introduce a higher data collection overhead than sampling-driven tools and thus have a higher chance of introducing *heisenbugs*: bugs (caused by performance perturbation) that disappear or alter their behavior when one attempts to probe or isolate them. This problem is particularly applicable for frequently occurring, short-lived events that force substantial delay in order to collect performance data.

Another common tool classification, tracing versus profiling, distinguishes how a tool handles the metrics each time instrumentation code is executed. A tool operating in tracing mode stores metric values calculated at each time instance separately from one another. From this data, it is possible for the tool to reconstruct the step-by-step program behavior, enabling application analysis in great detail. However, the large amount of data generated also requires significant storage space per program run, and the sheer amount of data could be overwhelming if it is not carefully organized and presented to the user. In addition, due to memory limitations, the tool often must perform file I/O during runtime, introducing additional data collection overhead on top of the unavoidable metric calculations. Dimemas/Paraver (Labarta et al., 1996), Intel Cluster Tools (see http://www.intel.com/software/products/cluster), MPE/Jumpshot (Chan et al., 2000), and MPICL/ParaGraph (Heath and Etheridge, 1991) are examples of tools that support the collection and viewing of trace data. In contrast, a tool operating in profiling mode performs additional on-the-fly calculations (minimum, maximum, average, count, etc.) after metric values are calculated at runtime and only statistical (profile) data are kept. This data can usually fit in memory, avoiding the need to perform file I/O at runtime. However, profiling data often only provide sufficient information to perform high-level analysis and may be insufficient for determining the causes of performance bottlenecks. Examples of popular profiling tools include DynaProf (Mucci, 2003), mpiP (Vetter and McCracken, 2001), and SvPablo (DeRose et al., 1998).

Finally, as the system used to execute the application grows in size, the amount of performance data that a tool must collect and manage grows to a point that it becomes nearly impossible for users to manually analyze the data even with the help of the tool. To address this issue, several tools such as HPCToolkit (Mellor-Crummey et al., 2002), Parodyn (Miller et al., 1995), Scalasca/KOJAK (Mohr and Wolf, 2003), and TAU (Shende and Malony, 2006) also include mechanisms to have the tool automatically analyze the collected performance data and point out potential performance bottlenecks within the application (examples of analyses supported include scalability analysis, performance bottleneck detection, and cause analysis).

### 2.2. PGAS Programming Models

The PGAS parallel programming model is based on the single process multiple data (SPMD) execution model and presents the programmer with a memory space logically divided into two parts: a private portion local to each node and a global portion partitioned among the nodes. The PGAS model provides the abstraction of a shared memory space that can be read and written directly, regardless of whether or not these operations result in network communication. This abstraction makes the PGAS model substantially more intuitive for many programmers than message-passing models such as MPI. Programmers using the PGAS model additionally have access to the notion of data within the shared memory space having affinity to a particular node, allowing them to exploit the important fact that local accesses are typically much faster than remote accesses and can be used to improve performance. The present family of PGAS programming models includes the existing Unified Parallel C, Co-Array Fortran, and Titanium languages and the SHared MEMory library as well as the Chapel and X10 languages currently in development as part of DARPA’s High-Productivity Computing Systems (HPCS) initiative.

In the remainder of this section, we provide an overview of UPC and SHMEM, the two PGAS models currently supported by PPW.

Unified Parallel C is an explicit parallel extension of the ANSI C language whose development started in the late 1990s based on experience with several earlier parallel C-based programming models. UPC exposes the PGAS abstraction to the programmer by way of several language and library features, including specially typed (shared) variables for declaring and accessing data shared among UPC program threads, synchronization primitives such as barriers and locks, a number of collective routines, and a unique, affinity-aware work sharing construct (upc_forall).

The organization responsible for the continuing development and maintenance of the UPC language is a consortium of government, industry, and academia, which released the latest UPC specification version 1.2 in June 2005. This specification has been implemented in the form of vendor compilers, including offerings from HP and IBM, as well as open-source compilers such as Berkeley UPC and the reference Michigan UPC. These provide for UPC support on a number of HPC platforms, including SMP systems, supercomputers such as the Cray XT series, and Linux clusters using a variety of commodity or high-speed interconnects.

The SHared MEMory library essentially provides the shared-memory abstraction typical of multi-threaded sequential programs to developers of high-performance parallel applications. First created by Cray Research for use on the Cray T3D supercomputer and now trademarked by SGI, SHMEM allows processes to read and write all globally declared variables, including those mapping to memory physically located on other nodes. SHMEM is distinct from a language such as UPC in that it does not provide intrinsically parallel language features; instead, the shared memory model is supported by way of a full assortment of API routines. In addition to the fundamental remote memory access primitives (get and put), SHMEM provides routines for collective communication, synchronization, and atomic memory operations. Implementations of the library are primarily available on systems offered by
SGI and Cray, though versions also exist for clusters using interconnects such as Quadrics. At the present time, no SHMEM standardization exists so different implementations tend to support a different set of constructs providing similar functionalities. However, an effort to create an OpenSHMEM standard (see https://email.orl.gov/mailman/listinfo/openshmem) is currently underway.

3. Background Research Findings

A substantial background research process has led to the formulation and development of the PPW system. In this section, we briefly describe this process and its resulting findings and insights that have shaped the PPW design.

We began our background research by studying the details of the PGAS programming model specifications and implementations in order to identify characteristics important in analyzing parallel application performance. In parallel, we surveyed existing works on performance tool research in order to identify characteristics important for the success of a performance tool (Leko et al., 2005). Using the knowledge gained from these studies, we then evaluated the applicability of existing performance tool techniques to various programming models and leveraged techniques that could be re-used or adopted. Additionally, we performed comparisons between related performance analysis techniques, identified characteristics common to these techniques, and made generalizations based on the commonalities (such generalization is desirable as it typically reduces tool complexity). Finally, we recognized new obstacles pertaining to PGAS performance analysis and formulated solutions to handle these issues.

A helpful performance tool must collect appropriate and accurate performance data. We found that it is useful for a tool to support both profiling and tracing measurement modes. Profile data guides users to program segments where they should focus their tuning efforts, while trace data provides detailed information often needed to determine the root causes of performance degradations. The tool should also make use of hardware counter monitoring systems, such as the portable Performance Application Programming Interface (PAPI) (London et al., 2001), which are valuable in analyzing non-parallel sections of an application but can also be used on parallel sections. Finally, to avoid performance perturbation, the data collection overhead introduced by the tool must be minimized. A general consensus from the literature indicates that a tool with an overhead of approximately 1–5% under profile mode and 1–10% under trace mode is considered to be safe from performance perturbation.

A productive tool must be easy to learn and use. While performance tools have proved effective in troubleshooting performance problems, they are often not used because of their high learning curve. To avoid this pitfall, a tool should provide an intuitive, familiar user interface by following an established standard or adopting visualizations used by existing performance tools. In addition, since source code is generally the primary level over which users have direct control (the average user may not have the knowledge or permission to alter the execution environment), performance tools should present performance data with respect to the application source code. This feature helps in attributing the causes of performance bottlenecks to specific source code regions, making it easier for the user to remove bottlenecks. A tool’s ability to provide source-line correlation and to work closer to the source level is thus critical to its success.

To efficiently support multiple programming models, a successful performance tool design must include mechanisms to resolve difficulties introduced due to diverse models and implementations. We noticed that the techniques for the measurement and the presentation stages are generally not tied to the programming model. The types of measurements required and difficulties one must solve are very similar among programming models (get timestamp, clock synchronization issues, etc.) and visualizations developed are usually applicable or easily extensible to a range of programming models. Furthermore, we noted that while each programming model supplies the programmer with a different set of parallel constructs to orchestrate work among nodes, the types of inter-node interaction facilitated by these constructs are quite similar between models. For example, both UPC and SHMEM include constructs to perform, among other operations, barrier, put, and get. Thus it is desirable to take advantage of this commonality and devise a generalization mechanism to enable the development of system components that apply to multiple models, helping reduce the complexity of tool design.

In contrast, the choice of the best instrumentation technique is highly dependent on the strategy used to implement the target compiler. Many diverse implementation methods are used by compiler developers to enable execution of target model applications. For instance, all SHMEM implementations are in the form of linking libraries while UPC implementations range from a direct compilation system (e.g., Cray UPC) to a system employing source-to-source translation complemented with extensive runtime libraries (e.g., Berkeley UPC). We noticed that while it is possible to select an established instrumentation technique that works well for a particular implementation strategy (for example, using the wrapper instrumentation approach for linking libraries), none of these techniques work well for all compiler implementation strategies (see Section 6 for additional discussion). Thus, any tool that wishes to support a range of compilers must include mechanisms to handle these implementation strategies efficiently.

4. PPW System Overview

Parallel Performance Wizard is a performance data collection, analysis, and visualization system for parallel programs (University of Florida, 2004). The goal is to provide a performance tool infrastructure that supports a wide range of parallel programming models with ease; in particular, we focus on the much-needed support for PGAS models. PPW’s high-level architecture is shown in Figure 1, with arrows illustrating the steps involved in the PPW-assisted application optimization process. A user’s source program is first compiled using PPW’s commands.
to generate an instrumented executable. This executable is then run and either profiling or tracing data (as selected by the user) are collected and managed by PPW. The user then opens the resulting performance data file and proceeds to analyze application performance in several ways: examining statistical performance information via profiling data visualizations supplied by PPW; converting tracing data to SLOG2 or OTF format for viewing with Jumpshot or Vampir; or using the PPW analysis system to search for performance bottlenecks.

PPW currently supports analysis of UPC, SHMEM, and MPI 1.x applications and is extensible to support other parallel programming models. To facilitate support for a variety of models, we developed a new concept: the generic-operation-type abstraction. In the remainder of this section, we discuss the motivations behind and advantages of using this abstraction.

Existing performance tools are commonly designed to support a specific model or are completely generic. Model-specific tools interact directly with model-specific constructs and have the advantage of being able to collect a varying set of operation-specific data (such as memory address and data transfer size for put and get). However, the cost of adding support for additional models to these tools is usually high, often requiring updates to a significant portion of the system, due to a need to re-implement the same functionality for each model. In contrast, completely generic tools such as MPE work with generic program execution states (such as the beginning and the end of function calls) and thus can be easily adopted to support multiple models. Unfortunately, being completely generic forces these tools to collect a standard set of metrics (e.g., source line, timestamp) each time data collection occurs, and as a result, these tools lose the capacity to obtain useful operation-specific metrics (e.g., data size for data transfer operations). To avoid the unnecessary tight-coupling of a tool to its supported programming models while still enabling the collection of useful operation-specific metrics, we developed a generic-operation-type abstraction that is a hybrid of the model-specific and the completely generic approaches. The idea is to first map model-specific constructs to a set of model-independent generic operation types classified by their functionality (the mapping of UPC, SHMEM, and MPI 1.x constructs to the generic operation types is provided in Table 2). For each generic operation, the tool then collects operation-specific events and metrics and may later analyze and present these data differently depending on the operation type (Figure 2).
The generic-operation-type abstraction has influenced many of the components of the PPW system, including its event model, instrumentation and measurement approach, and analyses and visualizations. We now describe these components in the following sections.

5. Parallel Programming Event Model

The effectiveness of an event-driven tool is directly impacted by the events and metrics (i.e., event model) that it uses. Events signify important instances during program execution when performance data should be gathered, while metrics define the types of data that should be measured and subsequently stored by the tool for a given event. In this section, we present the generic parallel programming event model PPW uses to describe the behavior of a given parallel application.

5.1. Overview of the PPW Event Model

PPW focuses on providing detailed information needed to analyze parallel portions of a program while maintaining sufficient information to enable a high-level sequential analysis. For this reason, the PPW event model includes mostly parallel events. Table 1 summarizes the theoretical event model using the generic-operation-type abstraction to describe the behavior of a given program; this event model is compatible with both PGAS models and MPI; Table 2 shows the mapping of UPC, SHMEM, and MPI constructs to generic operation types.

We organized Table 1 so that operation types with the same set of relevant events are shown together as a group. In addition to metrics useful for any event (i.e., calling node ID, code location, timestamp, and operation-type identifier), for each event, we provide a list of additional metrics that would be beneficial to collect. For one-sided communication, metrics such as the data source and destination (node ID and memory address), amount of data being transferred, and synchronization handler (for non-blocking operations only) provide additional insights on the behavior of these operations. For two-sided communication, it is necessary to match the node ID, transfer size, message identifier, and synchronization handler in order to correlate related operations. For lock acquisition or release operations and wait-on-value change operations, the lock identifier or the wait-variable address help prevent false bottleneck detection. For collective global-memory allocation, the memory address distinguishes one allocation call from another. For group communication, the data transfer size may help understand these operations. For lock acquisition or release operations and wait-on-value change operations, the lock identifier or the wait-variable address help prevent false bottleneck detection. For collective global-memory allocation, the memory address distinguishes one allocation call from another. For group communication, the data transfer size may help understand these operations. Finally, for group synchronization and communication that do not involve all system nodes, group member information is useful in distinguishing between distinct but concurrently executing operations.

<table>
<thead>
<tr>
<th>Table 1. PPW's Generic-Operation-Type-Based Event Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic operation type</td>
</tr>
<tr>
<td>-----------------------</td>
</tr>
<tr>
<td>Group synchronization</td>
</tr>
<tr>
<td>Group communication</td>
</tr>
<tr>
<td>Initialization</td>
</tr>
<tr>
<td>Termination</td>
</tr>
<tr>
<td>Global memory management</td>
</tr>
<tr>
<td>Atomic read/write</td>
</tr>
<tr>
<td>Blocking implicit put/get</td>
</tr>
<tr>
<td>Blocking explicit put/get</td>
</tr>
<tr>
<td>Non-blocking explicit put/get</td>
</tr>
<tr>
<td>Explicit communication synchronization</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Blocking send/receive</td>
</tr>
<tr>
<td>Non-blocking send/receive</td>
</tr>
<tr>
<td>Lock acquisition or release</td>
</tr>
<tr>
<td>Wait-on-value change</td>
</tr>
<tr>
<td>User-defined function/region</td>
</tr>
<tr>
<td>Work-sharing</td>
</tr>
<tr>
<td>Environment inquiry</td>
</tr>
<tr>
<td>Exit</td>
</tr>
</tbody>
</table>
It is important to point out that event timestamp information is often the most critical metric to monitor. With a proper set of events and accurate timing information for these events, it is possible to calculate (or at least provide a good estimate of) the duration for various computation, communication, and synchronization calls throughout program execution. In some cases, it is also possible to calculate program-induced delays\(^6\) (PI delays – delays caused by poor orchestration of parallel code such as uneven workload-distribution, competing data access, or lock acquisition) that point to locations in the program that can be optimized via source-code modification. By examining the durations of various operations and identifying PI delays that can be removed, it is much simpler for a programmer to devise optimization techniques to improve the execution time of the application.

In the following subsections, we discuss the events and means to calculate operation duration and PI delay for each of the logical groups of generic operation types shown in Table 1. For each group, we diagram some typical examples of the UPC, SHMEM, and MPI 1.x equivalent calls that can be used to calculate these metrics.
execution patterns via a set of operation-specific events (each indicated by an arrow with a number at the end) ordered with respect to time (x-axis) for each of the processing nodes (nodes X, Y, and Z) involved. We discuss means to calculate the duration of non-blocking operations (the duration for a blocking operation is always the time difference between its Enter and Exit events) and PI delay with these events, discuss why the inclusion of some events affects the accuracy of the calculations, and mention how a tool can track these events in practice. In addition, we point out performance issues typically associated with each operation group.
5.2. Group-Related Operations

In Figure 3, we illustrate the events for the category of operations that involves a group of nodes working together, including group synchronization, group communication, initialization, termination, and global memory allocation operations. The execution behavior of these operations is commonly described in terms of participating nodes running in one of two phases. First is the notification phase when the calling node sends out signals to all other nodes in the group indicating its readiness in performing the operation. The second is the wait phase where the calling node blocks until the arrival of signals from all nodes before completing the operation. Two versions of these group operations are typically provided to programmers: the standard (blocking) single-phase version where a single construct is used to complete both phases and the more flexible (non-blocking) split-phase version using separate constructs for each phase that allows for overlapping operation (generally restricted to local computation). With respect to existing programming models, the single-phase version is available for all operations in this category while the split-phase version is typically only available for group synchronization and group communication. PI delays associated with these operations normally mark the existence of load-imbalance issues.

Events associated with this category of operations are the following:

- **Enter (Notification_Begin):** Event denoting the beginning of cooperative operation (beginning of notification phase). The calling node starts sending out Ready signals (plus data for group communication operations) to all other nodes.
- **Notification_End:** Event denoting the point in time when the calling node finishes sending Ready signals (plus data for group communication operations) to all other nodes (end of notification phase). For the split-phase version, the calling node is free to perform overlapping operations after this point until the wait phase. In the single-phase version, this event is normally not traceable directly but is estimated to occur a short time after the Enter event.
- **Wait_Begin:** Event denoting the beginning of the wait phase (where the calling node blocks until Ready signals are received from all other nodes). Normally only traceable for the split-phase version.
- **Transfers_Received:** Event denoting the arrival of Ready signals from all other nodes on the calling node. This event is usually not traceable directly but is estimated to occur a short time after the last participating node enters the operation.
- **Exit (Wait_End):** Event denoting the completion of the cooperative operation.

An example execution pattern exhibiting bottlenecks (on nodes X and Y) caused by uneven work distribution for the single-phase version is diagramed in Figure 3(a). In this scenario, node Z entered the operation after it has received a Ready signal from both nodes X and Y (i.e., a Transfers_Received event occurred before an Enter event) so it was able to complete the operation without blocking. In contrast, nodes X and Y finished sending out signals before receiving all incoming Ready signals so they were unable to complete the operation optimally; each node became idle until it reached the Transfers_Received event. The PI delay for the single-phase version is given by the time difference between the Transfers_Received and Notification_End events (Figure 3, bottom).

Figure 3(b) shows an example execution pattern for the split-phase version. In this scenario, node Z received all signals before entering the notification phase so it is free of any PI delay. Node Y entered the wait phase before receiving all signals so it remained idle for a period of time before completing its operation (the idle time is given by the time difference between the Transfers_Received event and the Wait_Begin event). Finally, node X shows a situation where overlapping computation is used to remove potential delays (the advantage of the split-phase version). Node X entered the notification phase first so it logically required the longest wait time (i.e., the largest difference between the Transfers_Received and the Enter events). However, by performing sufficient computation, node X no longer needed to wait once it entered the wait phase and thus was free of delay. For the split-phase version, the total operation duration is given by the combined duration of the notification phase (the time difference between the Enter and Notification_End events) and the wait phase (the time difference between the Wait_Begin and Exit events) and the PI delay is given by the time difference between the Transfers_Received and Wait_Begin events (Figure 3, bottom).

5.3. Data Transfer Operations

In Figure 4, we illustrate the events for operations relating to one-sided, point-to-point data transfers such as atomic operations, blocking/non-blocking explicit or implicit put and get operations, and explicit-communication synchronization (e.g., fence, quiet). Note that we present the get operation as a reverse put (where the calling node sends a request to the target node and the target node performs a put) since get operations are often implemented this way in practice in order to improve their performance. For this class of operations, we are interested in determining the time it takes for the full data transfer to complete, from beginning of read/write to when data is visible to the whole system. The precise duration for the non-blocking version could be calculated if the Transfer_Complete event is available; unfortunately, it is often not possible for model implementations to supply this event. For such systems, the duration can only be estimated from the end time of either the explicit- or implicit-communication synchronization (Synchronization_End event) that enforces data consistency among processing nodes. As illustrated in Figure 4(c), this estimated duration could be much higher than the precise duration and as a result compromises the reliability of subsequent analyses. Furthermore, any PI delays caused by the
Fig. 4. Events for one-sided communication and synchronization operation.
synchronization operations increase the duration time further away from the actual transfer time. Finally, if an explicit synchronization operation is used to force the completion of multiple data transfers, a PI delay in one transfer will affect the duration calculation for all other transfers as well, further decreasing the accuracy of the performance information. To calculate the PI delay, either the \textit{Transfer\_Begin} or the \textit{Transfer\_Complete} event is needed; they are sometimes obtainable by examining the NIC status. The PI delay for a blocking put/get is the time difference between the \textit{Transfer\_Begin} and the \textit{Enter} events. For non-blocking put/get using implicit synchronization, the PI delay is the time difference between the \textit{Transfer\_Complete} and the \textit{Synchronization\_Begin} events. For non-blocking put/get using explicit synchronization, the PI delay is the time difference between the \textit{Transfer\_Begin} and the \textit{Synchronization\_Begin} events. PI delays associated with these operations often signify the existence of competing data accesses.

In Figure 5, we illustrate the events for operations (of node X) relating to two-sided, point-to-point data transfers such as blocking/non-blocking \texttt{send} and \texttt{receive} operations and explicit-communication synchronization. As with one-sided communication, we are interested in determining the time it takes for the full data transfer to complete. The duration for the non-blocking versions is the time difference between the \textit{send} (receive) \textit{Enter} and the \textit{Exit} events plus the time difference between the \textit{Signal\_Received} and the \textit{Wait\_End} events. Unfortunately, the \textit{Signal\_Received} event (an event denoting the point in time when the calling node received a \texttt{Ready} signal from the matching node) is nearly impossible to obtain; thus the duration can only be estimated from the exit time of the synchronization call that guarantees data transfer completion, resulting in a much higher than normal duration calculation. The PI delay for a blocking send (receive) is the time difference between the \textit{Enter} and the \textit{Matching\_Enter} events while the PI delay for a non-blocking send (receive) with matching blocking
receive (send) is the time difference between the \textit{Wait\_Begin} and the \textit{Matching\_Enter} events. For a non-blocking send (receive) with matching non-blocking receive (send), the PI delay cannot be calculated (since \textit{Signal\_Received} is not available). These PI delays signify a potentially inefficient calling sequence of send/receive pairs that typically stem from a load-imbalance prior to calling of these operations.

5.4. Lock, Wait-On-Value, and Locally Executed Operations

In Figure 6(a), we illustrate the events for the lock mechanisms and the wait-on-value operation. The duration is calculated from the time difference between the \textit{Enter} and \textit{Exit} events. To calculate the PI delay, the \textit{Condition\_Fulfilled} event is needed, which indicates when the lock becomes available/unlocked or when a remote node updates the variable to have a value satisfying the specified wait condition. This \textit{Condition\_Fulfilled} event is generally not traceable directly by the tool but instead can be estimated from other operations’ events (the last unlock or data transfer completed). PI delays associated with these operations generally stem from poor orchestration among processing nodes (such as lock competition and late updates of wait variables).

Finally in Figure 6(b), we illustrate the events for the locally executed operations such as the user-defined function/region, work-sharing, and environment inquiry operations. Tracking the performance of these operations is important as it facilitates the analysis of local portions of the program. Since we can consider each to be a blocking operation, the duration is simply the time difference between the \textit{Enter} and \textit{Exit} events. Without extensive sequential performance tracking and analysis, it is not possible to determine if any PI delay exists.

5.5. Implementation Challenges and Strategies

In this section, we briefly discuss the challenges and strategies used to implement the event model with respect to data collection (instrumentation and measurement), (automatic) data analysis, and data presentation. A more detailed discussion of each of these stages will be given in Sections 6 and 7.

Depending on the programming model, the chosen instrumentation technique, and the tool design decision, the set of events collected by the tool during runtime will be selected. The programming model dictates the \textit{Meaningful event set} to collect as specified by the event model. From this set, a subset of \textit{Measurable event set} that can be collected directly during runtime given the constraints imposed by the chosen instrumentation technique is identified. Finally, some tool design decisions may further limit the \textit{Actual event set} the tool supports. Once the \textit{Actual event set} is known, metrics (common metrics plus additional metrics in Table 1) associated with events in this set are collected during runtime.

Depending on the \textit{actual event set} collected, analyses performed during the analysis phase will differ. For example, to calculate the barrier duration and delay in MPI and SHMEM, the single-phase formulas are used, while for Berkeley UPC, the split-phase formulas are used. Programming model capabilities also play a role in what kind of analyses are performed. Analyses specific to barriers can be applied to all
Table 3. GASP Events and Arguments for Non-Blocking UPC Communication and Synchronization

<table>
<thead>
<tr>
<th>Operation identifier</th>
<th>Event type</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP_UPC_NB_GET_INIT</td>
<td>Enter</td>
<td>int * relaxed, void * dst, gasp_upc_PTS_t * src, size_t n</td>
</tr>
<tr>
<td>GASP_UPC_NB_GET_INIT</td>
<td>Exit</td>
<td>int * relaxed, void * dst, gasp_upc_PTS_t * src, size_t n, gasp_upc_nb_handle_t handle</td>
</tr>
<tr>
<td>GASP_UPC_NB_GET_DATA</td>
<td>Enter, Exit</td>
<td>gasp_upc_nb_handle_t handle</td>
</tr>
<tr>
<td>GASP_UPC_NB_PUT_INIT</td>
<td>Enter</td>
<td>int * relaxed, void * dst, gasp_upc_PTS_t * src, size_t n</td>
</tr>
<tr>
<td>GASP_UPC_NB_PUT_INIT</td>
<td>Exit</td>
<td>int * relaxed, void * dst, gasp_upc_PTS_t * src, size_t n, gasp_upc_nb_handle_t handle</td>
</tr>
<tr>
<td>GASP_UPC_NB_PUT_DATA</td>
<td>Enter, Exit</td>
<td>gasp_upc_nb_handle_t handle</td>
</tr>
<tr>
<td>GASP_UPC_NB_SYNC</td>
<td>Enter, Exit</td>
<td>gasp_upc_nb_handle_t handle</td>
</tr>
</tbody>
</table>

6. Instrumentation and Measurement

In this section, we introduce known approaches for instrumentation, discuss their strengths and limitations within the context of the goals of PPW, and then present our data collection solution based on a novel standardized performance interface called GASP.

6.1. Overview of Instrumentation Techniques

While several techniques have proven to be effective in application instrumentation (Shende, 2001), the differences in compilation and execution among the divergent compilation approaches prevent the selection of a universal instrumentation strategy. With source instrumentation, the instrumentation code is added as part of the high-level source code prior to execution time. Because the source code is altered during the instrumentation process, this technique may prevent compiler optimization and reorganization and also lacks the means to handle global memory models where some semantic details of communication are intentionally underspecified at the source level to allow for aggressive optimization (for example, implicit read/write of a shared variable in UPC is difficult to handle using source instrumentation, especially under the relaxed memory consistency mode where a given compiler may reorder the implicit calls to improve performance). With binary instrumentation, the instrumentation code is added to the machine code before or during program execution. A direct benefit of modifying the machine-code rather than the source-code is that recompilation is often not needed after each program modification. Unfortunately, binary instrumentation is unavailable on some architectures and yields performance data that is often difficult to correlate back to the relevant source code, especially for systems employing source-to-source translation. Finally, with library instrumentation (such as PMPI), wrappers are placed around functions implementing operations of interest. During execution time, a call to a function first executes the appropriate wrapper code that enables data collection and then invokes the original function. This approach is very easy to use but does not work for programming model constructs that are not in the form of a function call (such as an implicit put in UPC) or for compilers that generate code which directly targets hardware instructions or low-level proprietary interfaces.

A brute force approach to having a tool simultaneously support multiple programming models and implementations is simply to select an existing instrumentation technique that works for each particular model implementation. Unfortunately, this approach forces the writers of performance analysis tools to be deeply versed in the internal and ongoing changing or proprietary details of the implementations, which can result in tools that lack portability. In addition, the use of multiple instrumentation techniques forces the tool to handle each model implementation disjointly and thus complicates the tool development process.

6.2. The Global-Address-Space Performance Interface

The alternative we have pursued is to define an instrumentation-measurement interface, called the Global-Address-Space Performance interface (Su et al., 2006), that specifies the relationship between programming model implementations and performance analysis tools. This interface defines the events and arguments of...
importance for each model construct (see Table 3 for GASP events and arguments related to non-blocking UPC communication and synchronization calls). Insertion of appropriate instrumentation code is left to the compiler writers who have the best knowledge about the execution environment, while the tool developers retain full control of how performance data are gathered. By shifting the instrumentation responsibility from tool writer to compiler writer, the chance of instrumentation altering program behavior is minimized. The simplicity of the interface minimizes the effort required from the compiler writer to add performance analysis tool support to their system (and once completed, any tool that supports GASP and recognizes PGAS operations can support application analysis for that compiler). Concomitantly, this approach also greatly reduces the effort needed for performance analysis tool writers to add support for PGAS implementations (a single tool-side GASP implementation is sufficient for all compilers with GASP support).

The most important entry point in the GASP interface is the event callback function named \texttt{gasp\_event\_notify} that compilers use to notify when events of potential interest occur at runtime and provide useful information (e.g., event identifier, source code location, and event-related arguments) to the performance analysis tool. The tool then decides how to handle the information and what metrics to record. In addition, the tool is permitted to make calls to routines that are written in the source programming model or that use the source library to query model-specific information which may not otherwise be available. The tool may also consult alternative sources of performance information, such as CPU hardware counters exposed by PAPI, for monitoring serial aspects of computational and memory system performance in great detail. The \texttt{gasp\_event\_notify} callback includes a per-thread, per-model context pointer to an opaque, tool-provided object created at initialization time, where the tool can store thread-local performance data.

The GASP specification is designed to be fully thread-safe, supporting model implementations where arbitrary subsets of programming model threads may be implemented as threads within a single process and virtual address space. It is highly extensible by allowing a tool to capture model- and implementation-specific events at varying levels of detail and to intercept just the subset of events relevant to the current analysis task. It also allows for mixed-model application analysis whereby a single performance analysis tool can record and analyze performance data generated by all programming models in use and present the results in a unified manner. Finally, GASP provides facilities to create user-defined, explicitly-triggered performance events which allow the user to give context to performance data. This user-defined context data facilitates phase profiling and customization of specific code segments.

Several user-tunable knobs are also defined by the GASP specification to provide finer control over the data collection process. First, several compilation flags are included so a user can control the event types that the tool will collect during runtime. For example, the \texttt{-inst-local} compilation flag is used to request instrumentation of data transfer operations generated by shared local accesses (i.e., one-sided accesses to local data which are not statically known to be local). Because shared local accesses are often as fast as normal local accesses, enabling these events can add a significant runtime overhead to the application so by default, the tool does not collect these data. However, shared local access information is useful in some analyses, particularly those that deal with optimizing data locality (a critical consideration in PGAS programming) and performing privatization optimizations, and thus may be worth the additional overhead. Second, instrumentation \texttt{#pragma} directives are provided, allowing the user to instruct the compiler to avoid instrumentation overheads for particular regions of code at compile time. Finally, a programmatic control function is provided to toggle performance measurement for selected program phases at runtime.

6.3. GASP Implementations

Here we briefly discuss considerations for the compiler-side implementation of the GASP interface, focusing on UPC as it is the more interesting case. There are several UPC compilers with existing GASP implementations: Berkeley UPC, GCC UPC, and HP UPC (see http://h21007.www2.hp.com/portal/site/dsppp/PAGE.template/page. document?ciid=c108e1c4dde021120e1c4dde021102756de10 RCRD). Berkeley UPC translates UPC code to standard C code with calls to the Berkeley UPC runtime system. As a result, much of the corresponding GASP implementation consists of appropriate GASP calls made within the runtime system. However, several features of the GASP specification must be implemented within the compiler itself, including the \texttt{#pragma} directives for controlling instrumentation of program regions and support for instrumentation of user-function entry and exit. In addition, to providing appropriate UPC source code correlation, the compiler must pass source code information down through the translation process. By contrast, the GCC UPC and HP UPC compilers both use a direct compilation approach, generating machine code directly instead of translating UPC into C. With this architecture, the GASP implementation involves more changes to the compiler itself than with Berkeley UPC. In the case of GCC UPC, for example, changes were needed in one of the UPC compilation phases (called the “gimplification” phase because intermediate representations of functions are converted to GCC’s GIMPLE language) to determine if instrumentation is enabled and generate appropriate code if so.

7. Data Analysis and Visualizations

An effective performance system must provide the means to present and facilitate the analysis of collected performance data. In this section, we briefly describe the PPW analysis system developed to automatically detect and discover causes of performance bottlenecks, along with visualizations supplied by PPW to view collected profiling and tracing data. Due to paper length considerations and the amount of details necessary to present the PPW analysis system, only the high-level
overview of the system is provided here. For an in-depth discussion of our analysis system design, prototypes, and preliminary results, please see Su et al. (2009).

7.1. Automatic Analysis

As the size of the performance dataset grows, it becomes nearly impossible for the user to manually examine the data and find performance issues. To address this problem, we developed a new generic-operation-type-based PPW analysis system that supports two categories of analyses: application analyses (common bottleneck detection, cause analysis, and high-level analysis) which deal with performance evaluation of a single run; and experiment set analyses (scalability and revision analysis) to compare the performance of related runs. To improve analysis speed, we developed a distributed, localized analysis processing mechanism, which is peer-to-peer based and consists of up to \( N \) agents (where \( N \) is the application system size); up to \( N-1 \) non-head agents, each of which has (local) access to raw data from a set of nodes, and one head agent that also performs global analyses (requiring access to profile data from all nodes). The inherently parallel design is intended to support analysis of large-scale applications in a reasonable amount of time. In Figure 7, we illustrate the types of analyses conducted and the raw data exchange needed for all agents in a three-agent system.

In Figure 8, we depict the analysis processing phases each agent goes through, namely the global analysis phase, the detection phase, the cause analysis phase, and the resolution phase. In the global analysis phase, the head agent collects profiling data from all other agents in the system and performs analyses that require the access of global data. In the detection phase, each agent examines its portion of the (local) profiling data and identifies bottleneck profiling entries using one or more filtering techniques. In the cause analysis phase (if tracing data is available), each agent first examines the local tracing data, determines the remote operation data needed to determine the cause of bottlenecks and sends out requests to appropriate agents to obtain these data. Once requests are received, each agent examines its local tracing data again and sends the appropriate data back to the requesting agents. Each agent then waits for the arrival of replies and completes the cause analysis by assigning a bottleneck pattern name to each matching trace event, along with the remote operations that contributed to the delay. Finally, in the resolution phase, agents seek to provide hints to the user in removing bottlenecks identified in the other phases.

It is important to point out that the execution speed of an analysis system is directly related to the total number of tracing events generated rather than the execution time of the application. An application with shorter execution time might generate significantly more trace records than a longer one and thus would require longer analysis time. In the case of PPW, our experiment shows that the analysis speed of our sequential prototype (with one agent) is directly proportional to the total number of events; on average, the system requires 1 minute of processing time for every 4–5 million tracing events generated on a Pentium-4 2.8 GHz workstation.

7.2. Visualization Overview

PPW provides both graphical and text-based interfaces to view collected profile data and automatically generated analysis data. Most of these visualizations have been...
designed to have a similar look and feel to those provided by other tools so users already familiar with other tools can quickly learn and effectively use PPW. The following list summarizes the visualization-related features (with each supporting source-code correlation whenever possible) currently provided by PPW:

Fig. 8. Analysis process workflow for an agent in the system.

Fig. 9. (a) Load-balancing analysis visualization for CG 256-node run, (b) experimental set comparison chart for Camel 4-, 8-, 16-, and 32-node runs.
A view summarizing the application execution environment (optimization flags used, machine hostnames, etc.).

Charts to facilitate the identification of time-consuming application segments (10 longest-executing regions).

Flat and call-path tables to display high-level statistical performance information.

A visualization to detect and show load-balancing issues (Figure 9(a)).

A chart to compare related experimental runs (Figure 9(b)) such as runs of the same program using various system sizes or runs of different versions of the same program.

A display showing the inter-node communication volume for all data transfer operations in the program (providing processor-to-processor or processor-to-global-memory communication statistics).

A unique, UPC-specific array distribution display that depicts the physical layout of shared objects in the application on the target system (Figure 10).

Exports of trace data for viewing with the Jumpshot and Vampir (see http://www.vampir.eu) timeline viewers.

8. Experimental Results

We have performed a number of experiments to evaluate the effectiveness of the PPW system. In this section, we give PPW overhead and storage requirements, present the results of tool scalability testing, and discuss two case studies performed using the FT benchmark and the SAR application.

8.1. Overhead and Storage Requirements

In Table 4, we provide the data collection overhead and storage space needed for PPW. For this test, we executed the George Washington University’s UPC NPB version 2.4 benchmark suite (class B) (see http://www.gwu.edu/~upc/download.html) compiled with Berkeley UPC (see http://upc.lbl.gov) version 2.6, and executed an in-house SHMEM SAR application (see Section 8.4 for an overview of SAR) with Quadrics SHMEM using 16 nodes on an Opteron cluster connected via QsNet quadrics interconnect. Both sets of programs were instrumented and monitored using PPW, with performance data collected for all UPC/SHMEM constructs and user functions in each program. In all cases, the data collection overhead numbers...
<2.7% for profile, < 4.3% for trace) are comparable to existing performance tools. Tracing data size is linearly related to the total number of events instrumented by the tool; on average, PPW requires 17 MB of storage space per 1 million trace events.

In Table 5, we show a brief overhead and data-size comparison of the PPW, TAU, and Scalasca tools for a 16-node run of the MPI IS Benchmark.

Table 4. Per-Program Profiling/Tracing File Size and Overhead for 16-Node Runs of UPC NPB 2.4 Benchmark Suite (Class B) and SHMEM SAR Application

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>MG</th>
<th>SAR v1</th>
<th>SAR v3</th>
<th>SAR v5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of events</td>
<td>6.13E7</td>
<td>1388</td>
<td>8.33E6</td>
<td>2.68E8</td>
<td>1.39E6</td>
<td>2366</td>
<td>2.24E5</td>
<td>3.37E5</td>
</tr>
<tr>
<td>Profiling data size</td>
<td>276 kB</td>
<td>113 kB</td>
<td>369 kB</td>
<td>195 kB</td>
<td>840 kB</td>
<td>256 kB</td>
<td>215 kB</td>
<td>215 kB</td>
</tr>
<tr>
<td>Tracing data size</td>
<td>1.05 GB</td>
<td>0.15 MB</td>
<td>142 MB</td>
<td>4.56 GB</td>
<td>34 MB</td>
<td>0.26 MB</td>
<td>5.8 MB</td>
<td>8.6 MG</td>
</tr>
<tr>
<td>Execution time (no instrumentation)</td>
<td>18.66 s</td>
<td>18.19 s</td>
<td>19.19 s</td>
<td>0.96 s</td>
<td>1.86 s</td>
<td>107.81 s</td>
<td>81.01 s</td>
<td>59.45 s</td>
</tr>
<tr>
<td>Profiling overhead</td>
<td>1.66%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>2.69%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>Tracing overhead</td>
<td>2.84%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>2.08%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
</tbody>
</table>

Table 5. Profiling/Tracing Overhead and File-Size Comparison of PPW, TAU, and Scalasca Tools for a 16-Node Run of the MPI IS Benchmark

<table>
<thead>
<tr>
<th></th>
<th>PPW</th>
<th>TAU</th>
<th>Scalasca</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profiling overhead</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>Tracing overhead</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>Profile data size</td>
<td>133 kB</td>
<td>77.5 kB</td>
<td>56 kB</td>
</tr>
<tr>
<td>Trace data size per million events</td>
<td>16.16 MB</td>
<td>23.75 MB</td>
<td>11.44 MB</td>
</tr>
</tbody>
</table>

Table 6. Profiling/Tracing Overhead and File Size for Medium-Scale UPC NPB 2.4 Benchmark Suite Runs

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>128</th>
<th>256</th>
<th>512*</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td>0.87%</td>
<td>0.31%</td>
<td>N/A**</td>
</tr>
<tr>
<td>Data size</td>
<td>4.04 MB</td>
<td>14.10 MB</td>
<td>N/A</td>
</tr>
<tr>
<td>EP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td>6.34%</td>
<td>0.91%</td>
<td>0%</td>
</tr>
<tr>
<td>Data size</td>
<td>3.24 MB</td>
<td>11.82 MB</td>
<td>45.13 MB</td>
</tr>
<tr>
<td>FT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td>4.44%</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Data size</td>
<td>4.92 MB</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>IS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td>0%</td>
<td>4.06%</td>
<td>N/A</td>
</tr>
<tr>
<td>Data size</td>
<td>3.74 MB</td>
<td>12.80 MB</td>
<td>N/A</td>
</tr>
<tr>
<td>MG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overhead</td>
<td>4.55%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Data size</td>
<td>8.00 MB</td>
<td>21.32 MB</td>
<td>64.05 MB</td>
</tr>
</tbody>
</table>

8.2. Scalability

To evaluate the scalability of PPW, we conducted 128-, 256- , and 512-node runs of GWU’s UPC NPB version 2.4 benchmarks (class B) using Berkeley UPC 2.8 (via the GASNet MPI conduit using MPICH2 version 1.0.8) on an 80-node Intel Xeon 3220 quad-core cluster with a Gigabit Ethernet interconnect (all UPC/SHMEM constructs and user functions are instrumented). As shown in Table 6, the data collection overhead numbers are higher than with runs on smaller system sizes but are still within the acceptable range (< 6.34% for profiling, < 5.61% for tracing). In all cases, the profile data size remained in the manageable MB range. In contrast, the trace data size for larger runs is significantly greater for some of these benchmarks that exhibit weak scaling, such as CG and MG (for benchmarks
such as IS that exhibit strong scaling, the data size stays relatively constant); this characteristic could become an issue as the system size continues to increase. In Figure 11, we show the communication statistics visualization for a 256-node run of CG (Figure 11(a)) and the zoomed-in Jumpshot view of an MG 512-node run (Figure 11(b)).

8.3. Fourier Transform Case Study
For the first case study, we ran the Fourier Transform (FT) benchmark (which implements a fast Fourier transform algorithm) from the NAS benchmark suite version 2.4 using GASP-enabled Berkeley UPC version 2.6. Initially no change was made to the FT source code, and the performance data were collected for the class B setting executed using 16 nodes on an Opteron cluster with Quadrics QsNetII high-speed interconnects.

From the Tree Table (Figure 12), it was immediately obvious that the \texttt{fft} function call (third row) constituted the bulk of the execution time (18 s out of 20 s of total execution time). Further examination of performance data for events within the \texttt{fft} function revealed the \texttt{upc\_barrier} operations (represented as \texttt{upc\_notify} and \texttt{upc\_wait}) in \texttt{transpose2\_global} (fifth row) as potential bottleneck locations. We came to this conclusion by observing that the actual average execution times for \texttt{upc\_barrier} at lines 1943 (78.71 ms) and 1953 (1.06 s) far exceed the expected value of 2 ms on our system for 16 nodes (we obtained this latter value by running a simple benchmark). Looking at the code between the two barriers, we saw that multiple time-consuming \texttt{upc\_memget} operations (line 1950) were issued and speculated that the bottleneck was related to these operations. However, we are unable to fully verify this speculation and determine the cause of this bottleneck based solely on the statistical data.

Thus, we then converted the trace data into the Jumpshot SLOG-2 format and looked at the behavior of \texttt{upc\_barrier} and \texttt{upc\_memget} operations in a timeline view. We discovered that the \texttt{upc\_barrier} at line 1953 was waiting for the \texttt{upc\_memget} operation to complete. In addition, we saw that \texttt{upc\_memget} operations issued from the same node were unnecessarily serialized, as shown in the annotated Jumpshot screenshot (Figure 13; note the zigzag pattern for \texttt{memget} operations). For instance, looking at the start and end times of consecutive \texttt{upc\_memget} operations issued from node 0 to all nodes in the system (see the info box in Figure 13), we saw that the later \texttt{upc\_memget} operations were forced to wait for the earlier \texttt{upc\_memget} operations to complete before initiating (due to the blocking nature of \texttt{upc\_memget}), even though the data obtained were from different sources and stored locally at different private memory locations.

A solution to improve the performance of the FT benchmark is to use a non-blocking (asynchronous) bulk-transfer \texttt{get} such as \texttt{bupc\_memget\_async} provided by Berkeley UPC. When this code transformation was made, we were able to improve the performance of the program by 14.4\% over the original version. In this brief case study, we have shown how PPW was used to optimize a UPC 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.

8.4. SAR Case Study
For the second case study, we performed analysis of both UPC and SHMEM in-house implementations of the Synthetic Aperture Radar (SAR) algorithm using GASP-enabled Berkeley UPC version 2.6 and Quadrics SHMEM on an Opteron cluster with a Quadrics QsNetII interconnect.
SAR is a high-resolution, broad-area imaging processing algorithm used for reconnaissance, surveillance, targeting, navigation, and other operations requiring highly detailed, terrain-structural information. In this algorithm, the raw image gathered from the downward-facing radar is first divided into patches with overlapping boundaries so they can be processed independently of each other. Each patch then undergoes a two-dimensional, space-variant convolution that can be decomposed into two domains of processing, the range and azimuth, to produce the result for a segment of the final image (Figure 14).

The sequential version from Scripps Institution of Oceanography and the MPI version provided by two fellow researchers in our lab (Jacobs et al., 2008) were used as the templates for the development of UPC and SHMEM versions. The MPI version follows the master–worker approach where the master node reads patches from the raw image file, distributes patches for processing, and writes the result to an output file, while the worker nodes perform the actual range and azimuth computation on the patches (note, master nodes also perform computations). For this study, we used a raw image file with parameters set to create 35 patches, each of size 128 MB. While all patches could be executed in parallel in a single iteration on a system with more than 35 nodes, smaller systems, such as our 32-node cluster, execute over multiple iterations (in each iteration, $M$ patches are processed where $M$ equals the number of computing nodes). We assume only sequential I/O is available throughout the study, a fair assumption since neither UPC nor SHMEM currently includes standardized parallel I/O.

We began this PGAS case study by developing a UPC baseline version (which mimics the MPI version) using a single master node to handle all the I/O operations and that also performs processing of patches in each iteration. Between consecutive iterations, all-to-all barrier synchronization is used to enforce the consistency of the data. After verifying the correctness of this version, we used PPW to analyze the performance on three system sizes of computing nodes: 6, 12 and 18; these system sizes were chosen so that in each iteration, at most one worker node is not performing any patch processing. By examining several
visualizations in PPW (one of which is the Profile Metrics Bar Chart shown in Figure 15), we noticed that with six computing nodes, 18.7% of the execution time was spent inside the barrier and that the percentage increased with the number of computing nodes (20.4% for 12 nodes, 27.6% for 18 nodes). Using the timeline view to further investigate the issue (Figure 16), we then concluded that the cause of this bottleneck was that worker nodes must wait until the master node writes the result from the previous iteration to storage and sends the next patches of data to all processing nodes before they can exit the barrier.

We devised two possible optimization strategies to improve the performance of the baseline version. The first strategy was the use of dedicated master node(s) (performing no patch processing) to ensure that I/O operations could complete as soon as possible. The second strategy was to
replace all-to-all barrier synchronization with point-to-point flag synchronization (implementing the wait-until-value-changes operation) so processing nodes could work on the patches as early as possible. We expected that the first approach would yield a small performance improvement while the second approach should greatly alleviate the issue identified.

We then developed five revisions of the program using one or both of these strategies: (1) dedicated-master, (2) flag synchronization, (3) dedicated master with flag synchronization, (4) two dedicated masters (one for read, one for write), and (5) two dedicated masters and flag synchronization. These revisions were again run on system sizes with 6, 12, and 18 processing nodes, and the
performance of the revisions was compared to that of the baseline version (Figure 17). As expected, the dedicated master strategy alone did not improve the performance of the application. Surprisingly, the flag synchronization strategy by itself also did not improve the performance as we expected. After some investigation, we discovered that while we eliminated the barrier wait time, we introduced the same amount of idle time waiting for the shared flags to be set. The combination of both strategies, however, did improve the performance of the program by a noticeable amount, especially in the two dedicated masters and flag synchronization version where the percentage of the patch execution time increased (from 77.95%, 78.09%, and 70.71% for the baseline version) to 97.05%, 94.38%, and 87.97% of the total time for 6, 12, and 18 processing nodes, respectively (the remaining
time is mainly spent on unavoidable sequential I/O and bulk data transfer).

This case study was also performed using SHMEM implementations of SAR based on the same approaches outlined above for the UPC version (the performance comparison for these versions are also shown in Figure 17). For SHMEM, we noticed that the dedicated master strategy improved the performance by a small amount, while the flag synchronization strategy still did not help. The combination of both strategies again improved the performance by a noticeable percentage, with the two dedicated masters and flag synchronization version exhibiting 6.1%, 13.6%, and 15.8% improvement over the baseline version for 6, 12, and 18 processing nodes.

Additionally, PPW enabled us to observe that the performance of the SHMEM versions was 15–20% slower than the corresponding UPC versions (Table 7). This observation was surprising since we used the same Quadrics interconnect with communication libraries built on top of the same low-level network API. We examined the performance of the data transfers for UPC and SHMEM versions and found that the performance of these operations is actually better in SHMEM. After some investigation, we determined that the difference between the two versions came from the significant increase in execution time for read/write and put/get processing functions (i.e., the azimuth and range functions) in the SHMEM versions. We concluded that this behavior is most likely due to the overhead introduced by the SHMEM library to allow access to the shared memory space, which was incurred even for accesses on data physically residing on the same node. For UPC, a cast of shared pointer to local pointer made before entering these functions eliminates the overhead associated with global memory access (not available in SHMEM).

In this case study, we have shown how PPW was used to facilitate the optimization process of an in-house SAR application. We were able to use PPW to discover performance bottlenecks, compare the performance of both UPC and SHMEM versions side-by-side, and discover properties of the Quadrics SHMEM environment that should be considered when performing global memory accesses.

9. Conclusions and Future Directions

As hardware technologies for parallel computing mature, so 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 potential is often not realized due to the complexity of the execution environment. To tackle this problem, many parallel performance analysis tools were developed to help users optimize their code, with most tools supporting the message-passing model while having no or limited support for newer models such as partitioned global-address-space models. In this paper, we have outlined the bridging of this gap with the introduction of the first fully functional PGAS performance analysis system, called Parallel Performance Wizard, which currently supports UPC, SHMEM, and MPI 1.x models. First, we provided an overview of the PPW performance analysis system and discussed novel concepts developed to facilitate support for multiple programming models. We introduced the generic-operation-type abstraction, provided an in-depth discussion of the generic parallel programming event model applicable to both PGAS and message-passing models, and discussed the new GASP interface developed to solve challenges related to PGAS data collection. Finally, we gave an overview of PPW visualizations, briefly described the new PPW automatic analysis system, and demonstrated the effectiveness of PPW in PGAS application analysis via several experimental studies.

Future work on this project includes adding additional analyses to the PPW analysis system; completing integration of PPW into an Eclipse-based integrated development environment; enhancing the scalability of PPW’s visualizations; improving data collection overhead, management, and storage on large system sizes; and providing lower-level (e.g., programming model runtime and network-related) performance information using GASP.

Notes

1. Note that some of these challenges are applicable to MPI v2 that supports one-sided communication.
2. For tracing mode, these statistical calculations are often performed after execution time.
3. Note that it is possible but impractical to collect all metrics each time, as part of the collected metrics will not be meaningful.
4. The source and destination of a transfer may be different from the calling node.
5. Identifier used by the explicit/implicit synchronization operations to force completion of a particular put/get.
6. Example of a delay which is not a PI delay includes a data transfer delay due to network congestion, slowdown due to multiple applications running at the same time, etc.
7. A canonical get would have events similar to those illustrated for a put operation.
8. For example, existing UPC implementations include direct, monolithic compilation systems (GCC-UPC, Cray UPC) and source-to-source translation complemented with extensive runtime libraries (Berkeley UPC, HP UPC, and Michigan UPC).
9. Bottleneck resolution is still in development.
10. Note that while this code transformation is not portable to other compilers, the optimization strategy of using a non-blocking transfer is.
11. The overall system size for versions with dedicated master(s) are larger than the baseline version but the number of nodes that performs the patch processing remains constant. This setup provides a more meaningful comparison than using a constant overall system size.

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

Author’s Biographies
Hung-Hsun Su is a PhD candidate of the Department of Electrical and Computer Engineering (minor in computer and information science engineering) at the University of Florida. He received B.S. degrees in computer science and biochemistry from the University of California, Los Angeles in 1996, an M.S. in biochemistry from the University of Southern California in 1999, and an M.S. in computer science from the University of Southern California in 2002. His research includes the development and performance analysis of high-performance parallel applications, development of high-performance portable communication systems, and performance evaluation of high-performance systems.

Max Billingsley III is a research software engineer at the High-performance Computing and Simulation Research Lab at the University of Florida (UF). His main research interests are in parallel programming and performance.
analysis of high-performance parallel applications. He obtained a Master of Engineering degree in 2007 and a Bachelor of Science degree in 2005, both from UF.

*Alan D. George* is professor of Electrical and Computer Engineering at the University of Florida, where he serves as the director of the NSF Center for High-performance Reconfigurable Computing (CHREC) and the High-performance Computing and Simulation (HCS) Research Laboratory. He received a B.S. degree in computer science and an M.S. in electrical and computer engineering from the University of Central Florida, and a PhD in computer science from Florida State University. His research interests focus upon high-performance architectures, networks, systems, and applications in reconfigurable, parallel, distributed, and fault-tolerant computing.