GASP! A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models

Lawrence Berkeley National Lab Tech Report LBNL-61659

Hung-Hsun Su[1], Dan Bonachea[2], Adam Leko[1],
Hans Sherburne[1], Max Billingsley III[1], and Alan D. George[1]

[1] HCS Research Lab, Dept. of Electrical and Computer Engineering
University of Florida, Gainesville, FL 32611-62001
[2] Dept. of Electrical Engineering and Computer Sciences,
University of California at Berkeley, Berkeley, CA 94720-1770

Abstract. The global address space (GAS) programming model provides important potential productivity advantages over traditional parallel programming models. Languages using the GAS model currently have insufficient support from existing performance analysis tools, due in part to their implementation complexity. We have designed the Global Address Space Performance (GASP) tool interface that is flexible enough to support instrumentation of any GAS programming model implementation, while simultaneously allowing existing performance analysis tools to leverage their tool's infrastructure and quickly add support for programming languages and libraries using the GAS model. To evaluate the effectiveness of this interface, the tracing and profiling overhead of a preliminary Berkeley UPC GASP implementation is measured and found to be within the acceptable range.

1 Introduction

Parallel performance analysis tools (PATs) such as KOJAK [1] and TAU [2] have proven to be useful in tuning time-critical applications. By simplifying the instrumentation process, organizing performance data into informative visualizations, and providing performance bottleneck detection capabilities, these tools greatly reduce the time needed to analyze and optimize the parallel program under investigation. However, the majority of these tools support only a limited set of parallel programming models, focusing primarily on the message-passing model. As a result, programmers using newer parallel models are often forced to manually perform tedious and time-consuming ad-hoc analyses if they wish to optimize the performance of their parallel application.

While some work has been done in this area, the great majority of newer programming models remain unsupported by performance analysis tools due to the amount of effort that must be traditionally invested to fully support a

new model. In particular, models providing a global address space (GAS) abstraction to the programmer have been gaining popularity, but are currently underrepresented in performance analysis tool support. These models include Unified Parallel C (UPC) [3], Titanium [4], SHMEM, and Co-Array Fortran (CAF) [5]. Due to the wide range of compilers and techniques used to support execution of parallel applications using the GAS model, performance analysis tool writers face many challenges when incorporating support for these models into their tools. Among these problems are the technical issues associated with instrumenting code that may be highly transformed during compilation (e.g. by parallel compiler optimizations), and the challenge of adequately instrumenting parallel applications without perturbing their performance characteristics. The latter is especially challenging in the context of GAS languages where communication is one-sided and locality of access might not be linguistically explicit, such that statically indistinguishable accesses may differ in runtime performance by orders of magnitude.

In this paper, we present a Global Address Space Performance (GASP) tool interface [6] that is flexible enough to be adapted into current GAS compilers and runtime infrastructures with minimal effort, while allowing performance analysis tools to efficiently and portably gather valuable information about the performance of GAS programs. The paper is organized as follows. Section 2 provides the background and motivation in specifying such an interface. Section 3 gives a high-level overview of the interface, and Sect. 4 presents the preliminary results of the first implementation of the GASP interface. Finally, in Sect. 5, conclusions and future directions are given.

2 Background and Motivation

The traditional message-passing model embodied by the Message Passing Interface (MPI) currently dominates the domain of large-scale production HPC applications, however its limitations have been widely recognized as a significant drain on programmer productivity and consequently GAS models are gaining acceptance [7]. By providing a shared address space abstraction across a wide variety of system architectures, these models allow programmers to express interprocess communication in a way that is similar to traditional shared-memory programming, but with an explicit semantic notion of locality that enables high-performance on distributed-memory hardware. GAS models tend to heavily emphasize the use of one-sided communication operations, whereby data communication does not have to be explicitly mapped into two-sided send and receive pairs, a tedious and error-prone process that can significantly impact programmer productivity. As a result, programs written under these models can be easier to understand than the message-passing version while delivering comparable or even superior parallel performance [8, 9].

Most large-scale parallel systems employ communication hardware that requires explicit interaction with networking components in software, and consequently the compilers and libraries that support the execution of GAS programs

often need to perform a non-trivial mapping to convert user-specified one-sided communication operations into hardware-level communication operations. As a result, it can be challenging to determine the appropriate location for insertion of instrumentation code to track performance data. Furthermore, the one-sided nature of GAS model communication inherently biases available information to the initiator, making it more complicated for PATs to infer the state of the communication system and observe communication bottlenecks that may be incurred on passive participants. Finally, the instrumentation process has the potential to interfere with compiler optimization that normally takes place, which may perform aggressive rearrangement and scheduling of communication.

Several instrumentation techniques are used by existing parallel PATs. Unfortunately, none of these techniques provide a fully effective approach for GAS programming models. Source instrumentation may prevent compiler optimization and reorganization and lacks the means to handle relaxed memory models, where some semantic details of communication are intentionally underspecified at source level to allow for aggressive optimization. Binary instrumentation is unavailable on some architectures of interest, and with this method it is often difficult to correlate the performance data back to the relevant source code, especially for systems employing source-to-source compilation. An intermediate library approach that intersperses wrappers around functions implementing operations of interest does not work for compilers that generate code which directly targets hardware instructions or low-level proprietary interfaces.

Finally, different compilers and runtime systems may use wildly different implementation strategies (even for the same source language), which further complicates the data collection process. 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 MuPC). These divergent approaches imply sufficient differences in compilation and execution such that no single existing instrumentation approach would be effective for all implementations. A naïve way to resolve this issue is to simply select an existing instrumentation technique that works for one particular implementation. Unfortunately, this approach forces the writers of performance analysis tools to be deeply versed in the internal and often fluid or proprietary details of the implementation, and can result in system-dependent tools that lack portability, to the detriment of the user experience.

It is clear that a new instrumentation approach must be found to handle these GAS models. The alternative we have pursued is to define a standardized performance interface between the compiler and the performance analysis tool. With this approach, the responsibility of adding appropriate instrumentation code is left to the compiler writers who have the best knowledge about the execution environment. By shifting this responsibility from tool writer to compiler writers, the chance of instrumentation altering the 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. Concomitantly,

this simple interface makes it easy for performance analysis tool writers to add support for new GAS languages into existing tools with a minimum amount of effort.

3 GASP Interface Overview

The Global Address Space Performance interface is an event-based interface which specifies how GAS compilers and runtime systems communicate with performance analysis tools (Fig. 1). Readers are referred to the GASP specification [6] for complete details on the interface – this paper restricts attention to a high-level overview for space reasons.

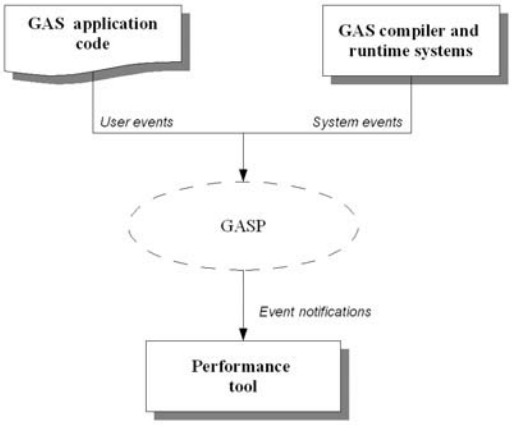

Fig. 1. High-level system organization of a GAS application executing in a GASP-enabled implementation

The most important entry point in the GASP interface is the event callback function named `gasp_event_notify` (Fig. 2), whereby GAS implementations notify the measurement tool when events of potential interest occur at runtime, providing an the event ID, source code location, and event-related arguments to the performance analysis tool. The tool is then free to decide how to handle the information and what additional metrics to record. In addition, the tool is permitted to make calls to routines that are written in the source language or that use the source library to query model-specific information that may not otherwise be available. Tools may also consult alternative sources of performance information, such as CPU hardware counters exposed by PAPI [10] for monitoring serial aspects of computational and memory system performance in great detail.

```
typedef enum {
  GASP_START,
  GASP_END,
  GASP_ATOMIC,
} gasp_evttype_t;

void gasp_event_notify(gasp_context_t context,
                       unsigned int event_id,
                       gasp_evttype_t event_type,
                       const char *source_file,
                       int source_line, int source_col, ...);
```

Fig. 2. Structure of GASP event notification

The `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 GAS model threads may be implemented as threads within a single process and virtual address space.

3.1 GASP Events

The GASP event interface is designed to be highly extensible, allowing language- and implementation-specific events that capture performance-relevant information at varying levels of detail. Additionally, the interface allows tools to intercept just the subset of events relevant to the current analysis task.

A comprehensive set of events has been defined for capturing performance information from the UPC programming model and includes the following basic categories. Shared variable access events capture one-sided communication operations occurring implicitly (through shared variable manipulation) and explicitly (through bulk transfer and asynchronous communication library calls). Synchronization events, such as fences, barriers, and locks, serve to record synchronization operations between threads. Work-sharing events handle the explicitly parallel regions defined by the user. Start-up and shutdown events deal with initialization and termination of each thread. There are also collective events which capture broadcast, scatter, and similar operations, and events which capture memory management operations on the shared and private heaps.

The GASP interface provides a generic framework for the programming model implementation to interact with the performance analysis tool, and the GASP approach is extensible to new GAS models through the definition of model-appropriate sets of events. The GASP interface is also designed to support mixed-model applications whereby a single performance analysis tool can record and analyze performance information generated by each GAS model in use and present the results in a unified manner.

Finally, GASP provides facilities for user-defined, explicitly-triggered performance events to allow the user to give context to performance data. This facilitates phase profiling and customized instrumentation of specific code segments.

3.2 GASP Instrumentation and Measurement Control

Several user-tunable knobs are recommended by the GASP specification to provide finer control over instrumentation and measurement overheads. First, the `--inst` and `--inst-local` compilation flags are used to request instrumentation of operations excluding or including events 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, instrumenting these events can add a significant runtime overhead to the application. By contrast, shared local access information is useful in some analyses, particularly those that deal with optimizing data locality and performing privatization optimizations, and thus may be worth the additional overhead. Instrumentation #pragma directives are provided, allowing the user to instruct the compiler to avoid instrumentation overheads for particular lexical regions of code at compile time. Finally, a programmatic control function is provided to toggle performance measurement for selected program phases at runtime.

4 Preliminary Results

A preliminary GASP implementation was added to Berkeley UPC [11] to test the effectiveness of the GASP interface. To test this implementation, we ran the UPC implementation of the NAS parallel benchmark suite version 2.4 ("class B" workload) under varying instrumentation and measurement conditions. For the CG, MG, FT, and IS benchmarks, we first compiled each benchmark using an installation of Berkeley UPC version 2.3.16 with all GASP code disabled. We used the best runtime for each benchmark as a baseline, and then recompiled each application against the same version of Berkeley UPC with GASP support enabled. We subsequently re-ran each benchmark under the following scenarios:

Instrumentation: a trivial GASP tool was linked to each benchmark that intercepted all `gasp_event_notify` calls and immediately returned. This scenario records the absolute minimum overhead that results as a consequence of GASP instrumentation being inserted into a program.

Instrumentation, local: the same trivial GASP tool used in the previous scenario was linked to each benchmark, and the `--inst-local` flag was also passed to the compiler. This scenario demonstrates the absolute minimum overhead imposed by GASP instrumentation that tracks both remote and local references in the global address space.

Measurement (profiling): we linked each benchmark against an actual performance analysis tool named Parallel Performance Wizard (PPW) [12] that records statistical information about each benchmark's runtime characteristics.

In particular, the total amount of time spent executing one-sided memory operations is collected and stored relative to the source line in which the operation was initiated. This scenario does not include local data accesses in the instrumentation.

Measurement (profiling with PAPI): this scenario used the same tool as the profiling scenario, but in addition to raw temporal data the PAPI hardware counter library was used to collect the total number of cycles and number of floating-point instructions consumed by each profiled entity. This scenario does not include local data accesses in the instrumentation.

Measurement (tracing): in this scenario, the same performance analysis tool recorded a full trace of the program's activity, storing information about each UPC operation such as byte count and source/destination threads in one-sided memory operations. This scenario does not include local data accesses in the instrumentation.

Each benchmark was run under each scenario a total of ten times, and an average was used to determine the percentage increase in overall execution time against the baseline after any data outliers were discarded. The measured variance for the data set was low, with standard deviation peaking at less than a few percent of overall runtime for the MG benchmark.

Figure 3 presents the results from our profiling and tracing experiments, giving a breakdown of the overheads obtained from each scenario listed above. In all cases, the overhead imposed by profiling was less than 5%, and the worst overhead for tracing (which is typically more expensive than profiling due to the disk I/Os needed to capture a complete record of events) was less than 9%. These overheads are well within acceptable limits for obtaining representative performance data.

Figure 4 compares the overhead of the "Instrumentation" and "Instrumentation, local" scenarios showing the minimum incremental cost of profiling local memory accesses in addition to remote accesses. It is encouraging that the instrumentation overhead alone for each benchmark was under 1.5% and 3.0% for remote and remote+local instrumentation (respectively), even in this relatively untuned GASP implementation. This outcome shows that the overall design of GASP is sound enough to accurately capture the fine-grained performance data typically associated with GAS models.

5 Conclusions and Future Directions

This paper introduces the Global Address Space Performance (GASP) interface that specifies a standard event-based performance interface for global address space languages and libraries. This interface shifts the responsibility of where to add instrumentation code from the tool writer to the GAS compiler/library writer, which improves the accuracy of the data collection process and reduces measurement perturbation. In addition, any performance analysis tool can quickly add support for GAS models by simply defining the body of a single, generic `gasp_event_notify` function when corresponding GASP im-

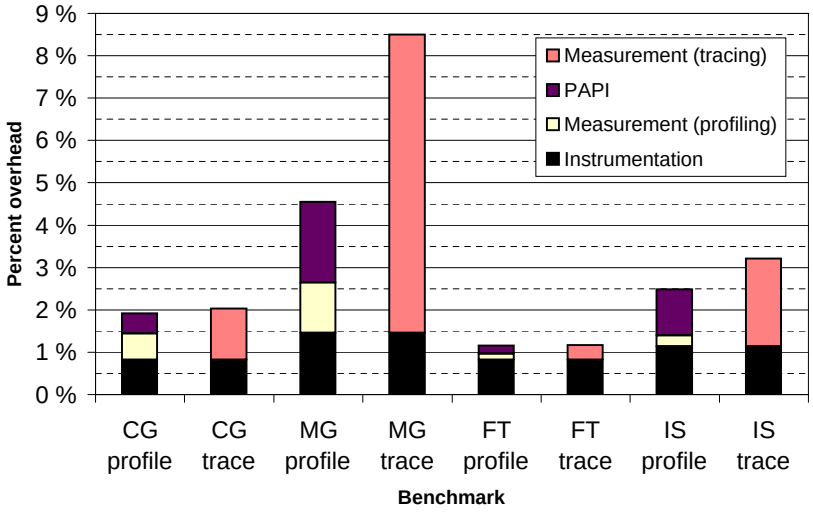

Fig. 3. Berkeley UPC GASP overhead for NAS benchmark 2.4 class B on a 32-node, 2-GHz Opteron/Linux cluster with a Quadrics QsNet^{II} interconnect

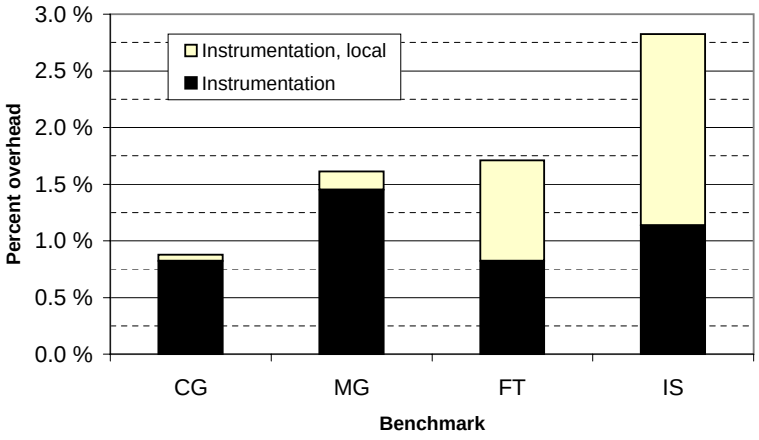

Fig. 4. Incremental raw instrumentation cost for profiling remote and local GAS accesses in the Berkeley UPC GASP implementation

plementations are available. To evaluate the effectiveness of such an interface, a preliminary version of the GASP interface was implemented in Berkeley UPC and overhead was measured and was found to be within an acceptable range.

To further evaluate and improve the interface, we're currently working to define GASP event sets for additional GAS models and plan to integrate GASP instrumentation into several GAS implementations including the Titanium compiler and several UPC and SHMEM implementations. In doing so, we aim to encourage more compiler developers to adopt the GASP interface in their own implementations. In addition, we are currently developing a new comprehensive parallel performance analysis tool called Parallel Performance Wizard (PPW) that makes full use of GASP. Figure 5 shows a screenshot of the tool's user interface, and Fig. 6 shows our tool integrating with the Jumpshot timeline viewer to allow the user to browse a visualization of trace data. Finally, we hope to extend the GASP interface to support other parallel programming models such as MPI-2, OpenMP, Chapel, Fortress, and X10.

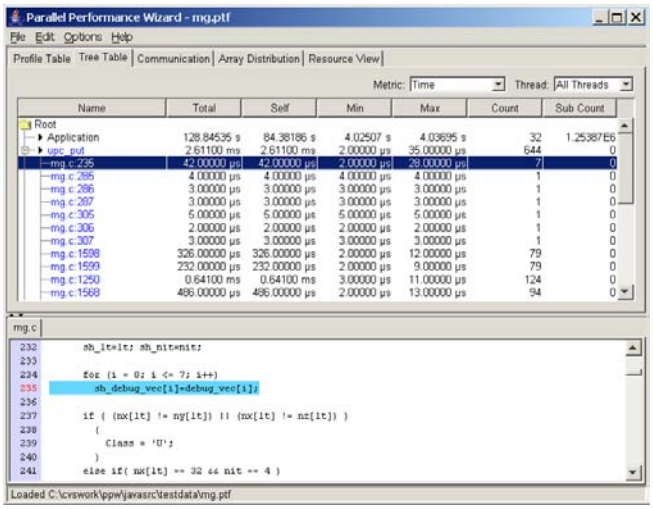

Fig. 5. Parallel Performance Wizard interface displaying performance data for the NPB MG benchmark

References

1. Mohr, B., Wolf, F.: KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003). Klagenfurt, Austria (September 2003)
2. Shende, S., Malony, A.D.: TAU: The TAU Parallel Performance System. International Journal of High Performance Computing Applications. **20:2** (2006) 287-331

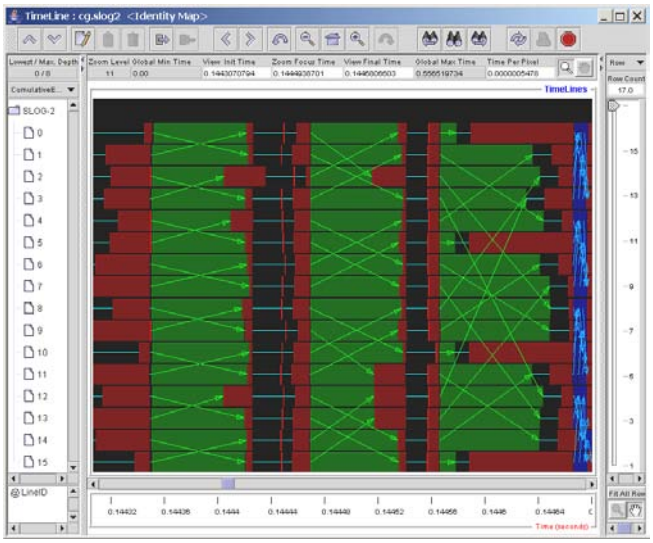

Fig. 6. Jumpshot timeline view of NPB CG benchmark

3. UPC Consortium: UPC Language Specifications v1.2. Lawrence Berkeley National Lab Tech Report LBNL-59208 (2005)
4. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A High-Performance Java Dialect. Concurrency: Practice and Experience, **10:11-13** (1998)
5. Numrich, B., Reid, J.: Co-Array Fortran for Parallel Programming. ACM Fortran Forum. **17:2** (1998) 1-31
6. Leko, A., Bonachea, D., Su, H., George, A.D.: GASP: A Performance Analysis Tool Interface for Global Address Space Programming Models, Specification Version 1.5. Lawrence Berkeley National Lab Tech Report LBNL-61606 (2006)
7. DARPA High Productivity Computing Systems (HPCS) Language Effort http://www.highproductivity.org/
8. Bell, C., Bonachea, D., Nishtala, R., Yelick. K.: Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. 20th International Parallel & Distributed Processing Symposium (IPDPS), 2006
9. Datta, K., Bonachea, D., Yelick K.: Titanium Performance and Potential: an NPB Experimental Study. Languages and Compilers for Parallel Computing (LCPC), 2005
10. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications (IJHPCA), **14:3** (2000) 189-204
11. Berkeley UPC Project: University of California at Berkeley and Lawrence Berkeley National Lab. http://upc.lbl.gov/
12. Parallel Performance Wizard Project: University of Florida, HCS Research Lab. http://www.hcs.ufl.edu/upc/