Performance Analysis Framework for GASNet Middleware, Tools, and Applications

Prashanth Prakash Max Billingsley III Alan D. George Vikas Aggarwal

High-performance Computing and Simulation (HCS) Research Lab
Dept. of Electrical and Computer Engineering
University of Florida, Gainesville, Florida 32611-6200
{prakash,billingsley,george,aggarwal}@hcs.ufl.edu

Abstract

Design of efficient communication libraries and languages for high-performance, parallel computer systems is difficult, requiring rigorous performance analysis and optimization to yield acceptable performance. Developers of such libraries and languages often rely upon various lower-level communication APIs. The GASNet communications middleware from UC Berkeley and LBNL, which provides a set of primitives for partitioned global-address-space programming models, has grown to become the underlying foundation for a variety of languages and libraries such as Berkeley UPC, Titanium, Chapel, and some implementations of SHMEM (e.g. GSHMEM). Despite the increased popularity and usage of GASNet, tools for analyzing its performance are lagging. An efficient framework for performance analysis of GASNet can be beneficial to tools developers, as well as advanced applications developers. In this paper, we present our research investigation and the challenges involved in design of a performance-analysis infrastructure for GASNet, leveraging the PPW performance tool and GASP interface developed at Florida. The proposed framework enables a comprehensive view of various performance-related events from different layers within a user application (e.g. GASNet, UPC, or SHMEM, as well as the application itself). Our framework for GASNet-related performance analysis is highly modular, allowing seamless integration with different tools built atop GASNet. We evaluate our approach for instrumentation of GASNet and illustrate the benefits of our approach through two case studies representing different usage scenarios involving Berkeley UPC and GSHMEM.

PGAS '11 Galveston, Texas.

Keywords GASNet, partitioned global-address space, UPC, OpenSHMEM, GASP, PPW, performance tools

1. Introduction

Due to the complexity involved in design of efficient, high-performance parallel applications, tools for performance analysis have become a critical component in the application-development process. A variety of performance-analysis tools are currently available that assist developers with identifying performance bottlenecks in their applications. While there are a plethora of tools [1] [2] that support performance analysis of applications, support for languages and libraries based upon partitioned global address space (PGAS) has been very limited. Parallel Performance Wizard (PPW) [3] developed at Florida pioneered research in exploring and resolving challenges involved in performance analysis of PGAS-based languages and libraries such as UPC [4] and SHMEM [5].

The design and implementation of a language or a communication library based upon PGAS is difficult, requiring performance analysis, bottleneck detection, and optimization on a target platform to yield an acceptable level of performance. System developers often rely on various lower-level, network-independent communication APIs for implementing such libraries and languages. GASNet (Global Address Space NETworking) [6] is one such communications middleware from UC Berkeley and LBNL, which provides a set of primitives for building PGAS programming models. Due to its support for a large variety of systems and interconnect technologies, GASNet has become a popular building block for implementing a number of PGAS-based languages and libraries such as Berkeley UPC (BUPC) [7], Chapel [8], Titanium [9], Co-Array Fortran [10], and some implementations of SHMEM (e.g. GSHMEM from Florida).

Data from performance analysis of GASNet behavior could be useful in numerous scenarios, since it would provide details about a variety of performance-related GASNet events. Consider an example of a *shmem_barrier* routine in a SHMEM library designed atop GASNet. As shown in Fig-

ure 1, with the support of performance analysis of GASNet, performance data can be captured about different GASNet events triggered by the *shmem_barrier* function.

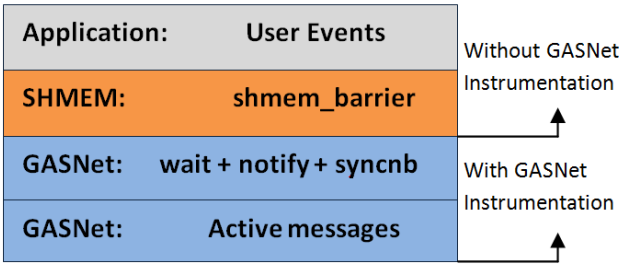


Figure 1. Example showing the availability of additional details, while using SHMEM with GASNet instrumentation.

Data related to the performance of GASNet routines and its underlying mechanisms can be very useful for a variety of users and purposes. For example, performance data can provide developers of programming languages such as UPC or SHMEM a top-level view of the way an application is using the library or runtime. Additionally, when performance data from GASNet is provided within the context of parallel-programming constructs (such as barriers or collectives), it can provide a comprehensive view of the events that occur in different layers of software (on top of which a particular application is built), as well as provide information about interaction between these different layers. Without such a comprehensive view, application developers have to typically employ different analysis tools to retrieve performance-related information from different software layers. Performance data related to GASNet can also be useful for developers of GASNet itself in identifying future enhancements by understanding the way different PGAS programming languages stress different features of GASNet.

The approach presented in this paper to support performance analysis of GASNet is by augmenting the PPW framework to support multiple programming models simultaneously. Additionally, we identify important performance-related events in GASNet and extend it to include GASP event notification mechanism. This approach enables us to present performance data to users by taking advantage of existing PPW visualizations. The main goal of our research and design is to provide a unified view of various events from a programming model (UPC or SHMEM in our prototype), GASNet, and an user application, such that an application developer can understand the relationship between events from different layers. Since GASNet is employed by a variety of languages and libraries, our framework is modular to allow seamless integration with any parallel-programming language. Our approach presented in this paper incurs minimal overhead while extracting valuable performance data from GASNet as well as other software layers present in an application. We illustrate the merits and utility of our approach using two case studies, one each with BUPC and GSHMEM. We also present different visualizations available in our framework for analyzing data and determining potential performance bottlenecks.

The remainder of the paper is organized as follows. Section 2 provides background on the different components involved in our framework. Section 3 is an overview of our framework with emphasis on its support for BUPC and GSHMEM. In Section 4, we provide two case studies to demonstrate our approach and illustrate the benefits of performance analysis by instrumentation of GASNet. Finally, Section 5 summarizes the work with conclusions and directions for future work.

2. Background and Related Research

In this section, a brief overview is provided for various tools related to our performance-analysis framework for GASNet. These tools include GASNet, BUPC, GSHMEM, GASP, and PPW.

2.1 Berkeley UPC and GASNet

Berkeley Unified Parallel C (BUPC) is a portable, high-performance implementation of UPC for multiprocessors and clusters, developed at the Lawrence Berkeley National Laboratory (LBNL) and the University of California (UC) at Berkeley. The UPC runtime system is a lightweight layer that implements language-specific features, such as shared-memory allocation and shared-pointer manipulations. The BUPC runtime runs atop GASNet, so we use it to demonstrate the capabilities of our performance-analysis framework.

The GASNet [11] middleware provides a set of lower level, high-performance communication primitives specifically designed to implement PGAS libraries or runtime. GASNet supports a large number of conduits over UDP, Quadrics, InfiniBand, Myrinet GM, MPI (Message Passing Interface), SHMEM, Cray XT and IBM BG/P among others. Table 1 shows the classification of APIs provided by GASNet. At a high level, it consists of two sets of APIs, namely the core API and the extended API.

Table 1. Organization of GASNet APIs

GASNet API	
Core API	**Extended API**
Job-control interface	Memory-memory operations
Active-message interface	Register-memory operations
Atomicity interface	Barriers

The GASNet core API is based upon active messages (AM) [12] and implemented directly on top of each network architecture also known as conduit. The core API mainly consist of (a) a job-control interface for bootstrapping, job termination, and job environment queries, (b) an active-messaging interface for implementing requests, replies, and

their handlers, and (c) an interface for handler signal safety and atomicity locks.

Active messages are implemented by matching requests and reply operations. When a request is received, based on the handler ID argument, a request handler is invoked. Similarly, when a reply is received, a reply handler is invoked. The handlers may run asynchronously or concurrently with respect to main computation thread. AM handlers are not interruptible, in other words a thread running a handler will not be interrupted to run another handler. The only exception is in the case of *gasnet_AMreply* call made at the end of a request handler, which can run AM reply handlers synchronously. GASNet provides special locks called Handler Safety Locks (HSL) to protect the objects shared between handlers and main thread. Once a thread acquires HSL it cannot be interrupted to run a handler till the HSL is released.

The GASNet extended API provides high-level functionality, such as memory-to-memory data transfers, register-to-memory data transfer, and notify and wait for split-phase barriers. Whenever possible, functions in the extended API are implemented on top of network hardware in order to exploit available features such as RDMA to achieve higher performance. There is also a generic implementation of the extended API that uses only the core API.

2.2 GSHMEM

SHMEM is a lightweight, shared-memory programming model and belongs to PGAS family. SHMEM is known for its simplicity, performance, and support for features such as one-sided communication, various collective operations, among others. GSHMEM [13] (a.k.a. GatorSHMEM) is reference design of a portable OpenSHMEM library, which leverages GASNet communication libraries and is being developed at the University of Florida. Similar to MPI [14], GSHMEM also provides a name-shifted version of the API, which can be used by performance tools to assist developers in performance analysis of their applications. We employ GSHMEM as a tool that will benefit from our performance-analysis framework for GASNet as shown in the case studies.

2.3 GASP and PPW

The Global-Address-Space Performance (GASP) interface [15] allows Global-Address-Space (GAS) compiler and runtime developers to add performance-analysis support quickly, while requiring fewer modifications in performance-analysis tools to support newer GAS models. When a user executes an application, the GAS runtime makes callbacks to the performance tool, which in turn stores the details of the specified event. In this work, the GASP interface is used to notify necessary events from GASNet to our performance tool. The GASP interface callback structure mainly involves two functions:

- *gasp_init* - This function is provided by the performance tool and is collectively called by all threads.
- *gasp_event_notify* - This function is called by runtime code to notify an event to the performance tool. The main parameters include *event-tag*, which identifies the event as described specifically in each programming model, and *event-type*, which indicates whether the event is a begin event (GASP_START), an end event (GASP_END), or an atomic event (GASP_ATOMIC).

The Parallel Performance Wizard (PPW) [16] is a performance-analysis tool designed specifically for PGAS languages. The current version of the tool supports the GASP instrumentation for several languages such as UPC, SHMEM, and MPI. We use PPW to provide us with the tool-side framework, so that we can add support and any additional features required to support GASNet instrumentation. The design of PPW provides us with the flexibility to introduce new operation types and extend the measurement module in PPW for new operation types. This flexibility is necessary for modeling active messaging, which is a major component of the core API of GASNet.

2.4 Related Research

HPCToolkit [17] is an integrated suite for measurement and analysis of program performance. It uses call-stack sampling to collect performance data, which makes it possible to provide visualizations that shows the interaction between GASNet and programming model running on top of it. While HPCToolkit extracts the data by sampling the call stack, our approach involves classifying the GASNet operations into different categories and collecting data related to them through the GASP interface. With call-stack sampling it is possible to extract the call-stack information from an application without instrumentation, our approach requires instrumentation but makes it possible to get operation counts and provides the option to identify the communication endpoints. Other tools that implement call-stack sampling (such as TAU) will be able to present the interaction between the GASNet other components in the form of call-stack, but support for performance analysis of GASNet using instrumentation is nonexistent to the best of our knowledge.

GASNet has an inbuilt tracing and statistical collection utility [18] for monitoring communication events in GASNet applications. When tracing is enabled, trace data from each thread is written to a unique text file, which can be used with the help of a text (terminal) based, trace summarization tool called *upc_trace* [19], which is provided by Berkeley UPC.

3. Performance-Analysis Framework for GASNet

In order to collect performance data from GASNet, we expanded the PPW framework to support GASNet, and we extended GASNet to include GASP event-notification func-

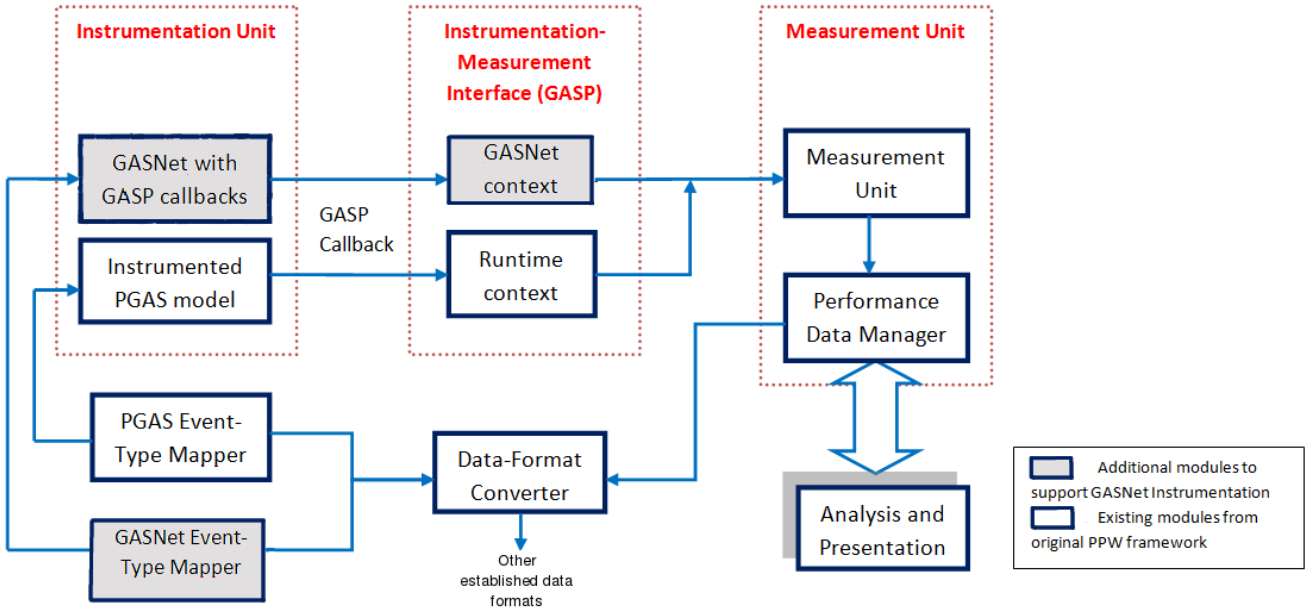


Figure 2. PPW framework to support GASNet instrumentation

tionality. In the following section, we describe the extensions required in the PPW framework in order to support GASNet instrumentation and provide a comprehensive view of the performance data collected from various parts of an application. Next, we present and discuss a list of important performance-related events in GASNet and their support in our framework. Finally, we describe the effort required to support a new parallel-programming model, using GSHMEM as an example.

3.1 PPW Support for GASNet

To enable collection of performance data from GASNet, PPW's support for performance analysis must be extended to include GASNet, in addition to existing support of UPC and SHMEM. Figure 2 illustrates at a high level the modified framework of PPW. The shaded modules are the new modules (not a part of the original PPW framework) added to support collection of performance data from GASNet.

The *instrumentation unit* is responsible for recording of data related to each model supported (e.g. UPC or SHMEM or GASNet). The techniques used for instrumentation in each model can be different, but they should provide a way to make GASP callbacks on specific events. Consider the examples of BUPC and GSHMEM. In BUPC, the GASP callbacks are added to the runtime, and same approach is followed for GASNet. GSHMEM provides a name-shifted version of all APIs, which can be wrapped by performance tools to form a different library. When an application developer requests profiling, the application can be linked with this wrapper library.

The *instrumentation-measurement interface* in our case is a GASP interface that provides a communication channel through which instrumentation code from runtime or GASNet can relay data to the measurement unit in an efficient manner. *Runtime context* and *GASNet context* are software abstractions that collectively represent the structures and interfaces in the instrumentation-measurement interface. The *measurement unit* records the data of interest with every event reported. Data collection is based upon the generic operation types, and thus it is independent of the parallel-programming model. The *performance data manager* is responsible for access, storage, and merging of data, and it also performs some post-processing of data. The *analysis and presentation unit* is responsible for providing tools for analysis and visualizations of the collected performance data. The *data-format converter* is solely responsible for converting the PPW performance data into other established performance-data formats such as TAU profile, CUBE profile, OTF and SLOG2.

As can be seen from Figure 2, the system uses a common measurement unit for recording all the events from both UPC (or SHMEM) and GASNet. This method is required to have a unified view of the events. Consider an example, where we have *upc_memget* using *gasnet_get_bulk*, and we want our performance data to show this relationship and all other related data. By contrast, a different approach would be to have two different measurement units, one for GASNet and the other for UPC (or SHMEM), but this separation results in the loss of data representing the relationship between UPC (or SHMEM) events and GASNet events.

One of the novel aspects of PPW is its use of generic operation types instead of language-specific constructs. The use of these generic operation types makes it very easy to extend PPW to support new parallel-programming languages and libraries. For example, instead of having a model-specific construct such as *shmem_barrier*, PPW uses a generic operation type named *PPW_TYPE_BARRIER*. When a component requires information about a model-specific construct, it obtains this data from the *event-type mapper* (shown in Figure 2). The measurement unit and the analysis and presentation unit are designed using these generic operation types, so they are easily extensible and language-independent.

Performance-analysis tools use a number of features of parallel languages or libraries for their operations, such as data transfer, synchronization, among others. This set of features is usually referred to as *upcalls* in PPW. Consider an example, where PPW is used with UPC. A particular component within PPW can use the barrier operation by using *ppw_upcall_barrier* which internally uses *upc_barrier*. When a new programming language is to be supported, the upcalls are mapped as required and thus components of PPW that use these upcalls need not be modified. In the case of GASNet, all upcalls cannot be implemented in a simple manner as with other parallel languages using their language constructs (For example, *shmem_get*, *upc_memget*, among others). If we are to design a set of GASNet upcalls, then the internals of UPC (or SHMEM) and GASNet must be taken into consideration. Consider an example of implementing send or receive upcalls using *gasnet_get*. In order to use *gasnet_get*, we require a memory region in the shared-memory space registered by UPC (or SHMEM) runtime. As a result, performance tools should be aware of memory management of shared-memory region or request memory from UPC (or SHMEM) runtime, which makes the implementation of upcalls complex and runtime dependent. Therefore, we follow a very simple approach, which involves using the upcalls from UPC (or SHMEM) and avoiding GASNet upcalls. In this way, the selection of a proper set of upcalls is performed by having a predefined variable that gives information about the specific parallel-programming language in use.

3.2 Extensions to GASNet

The collection of performance data from the GASNet layer requires identification of important performance-related events within GASNet. Once these events are identified, event-notification functionality is required to record information about each event. We provide such functionality by adding appropriate GASP callbacks in GASNet for various events of interest.

GASNet supports several network architectures, with its core API implemented directly on top of network-specific functionality. As a result, instrumentation of the core API is performed on a case-by-case basis for each conduit using appropriate GASP callbacks. By contrast, the extended API can be implemented directly on top of hardware-specific functionality or over GASNet's core API. Instrumentation of the extended API depends upon how the extended API is implemented.

Most of the performance-related events map directly to routines in the corresponding GASNet APIs except for the handler events. Table 2 lists the different categories of GASNet events that we monitor, with few examples for each category. The core-API events include start-up and exit events, active-messaging events, and atomicity events. Active-message events include distinct events for short, medium, and long messages for requests, replies, and their corresponding handlers. The extended API includes events for blocking memory operations, non-blocking memory operations (with explicit and implicit handles), register-memory operations, and split-phase barriers.

We can model handlers in a manner similar to other events with well-defined start and end events. Similarly, holding a handler safety lock (HSL) results in an implicit no-interrupt section, such that no other asynchronous events can occur while holding the lock. This property of HSLs allows us to model them in a manner similar to handlers. However, such an approach for active-message handlers and HSLs may lead to a problem during presentation of performance data (through existing visualization techniques in PPW), as these events can be nested between other synchronous events, leading to a representation that may not be desirable to some users. Consequently, our design provides an option to skip the profiling of active-message handlers and HSLs or treat them as atomic events.

Table 2. GASNet events by category

Category	**Example Events**
CORE API EVENTS	
Start-up and exit events	ATTACH
Active Messaging Events	AM_REQ_SHORT AM_REQ_SHORT_HANDLER AM_REPLY_SHORT AM_REPLY_SHORT_HANDLER
Atomicity Events	HSL_TRYLOCK INTERRUPT_HOLD
EXTENDED API EVENTS	
Blocking get and put	GET GET_BULK
Non-blocking get and put (explicit handle)	GET_NB_INIT GET_NB_DATA WAIT_SYNCNB TRY_SYNCNB
Non-blocking get and put (implicit handle)	GET_NBI_INIT GET_NBI_DATA WAIT_SYNCNBI_GETS TRY_SYNCNBI_ALL
Register put and get	PUT_VAL PUT_NBI_VAL WAIT_SYNCNB_VALGET
Barrier	BARRIER_NOTIFY BARRIER_WAIT BARRIER_TRY

Events are notified to PPW by using the *gasp_event_notify* callback function. Before any notify callbacks can be invoked, the GASP interface must be initialized by using the *gasp_init* callback, which returns GASP context (of type *gasp_context_t*). This GASP context contains information specific to GASNet and is used by the subsequent invocation of notify callbacks. The GASP context allows a performance tool to differentiate between events from different parts (software layers) of an application. The extension involving additions of event notification is a non-recurring effort, since all of these modifications can be reused by different programming tools built over GASNet. The semantics of the *gasp_event_notify* callback requires us to define a number of specific events, which can be atomic or have well-defined start and end events. Blocking operations such as *get* and *barrier* are modeled to have specific start and end events, whereas events such as *interrupt-hold* can be modeled as an atomic event.

To support active messages, we need to associate generic operation types with different active-messaging operations. Since active messages do not map to any of the existing types of generic operations, we introduce some new generic operation types for requests, replies, and their corresponding handlers. Table 3 lists the generic operation types used in GASNet. Operation types PPW_TYPE_AM_REQUEST and PPW_TYPE_AM_REPLY map directly to the active message request and reply function calls of the core API, whereas operation types PPW_TYPE_AM_REQUEST_HANDLER and PPW_TYPE_AM_REPLY_HANDLER map to the events corresponding to invocation of handlers.

Table 3. Generic operation types for GASNet

Operation Type	Description
PPW_TYPE_AM_REQUEST	Active message request
PPW_TYPE_AM_REPLY	Active message reply
PPW_TYPE_AM_REQUEST_HANDLER	Active message request handler
PPW_TYPE_AM_REPLY_HANDLER	Active message reply handler
PPW_TYPE_LOCK	Handler safety lock
PPW_TYPE_GET	Blocking get operation
PPW_TYPE_NBGET	Non-blocking get operation
PPW_TYPE_PUT	Blocking put operation
PPW_TYPE_NBPUT	Non-blocking put operation
PPW_TYPE_BARRIER	Barrier

To have a comprehensive view of performance-related events (from application, UPC/SHMEM, and GASNet), we need to have event callbacks from BUPC and GSHMEM as well. In the case of GSHMEM, we can use the name-shifted API routines to provide a PPW-specific wrapper library that includes the necessary GASP callbacks for SHMEM-specific events. In the case of BUPC, GASP callbacks were added by the Berkeley UPC group for versions 2.4 and above of their tool. User application events are received as a part of BUPC's (or GSHMEM's) GASP context as per the GASP specification, whereas GASNet will have only GASNet-specific events. With all the callbacks in place, the modified PPW framework records these events in a way that can be presented in a comprehensive manner. Since the performance data is collected based upon generic operation types, the existing graphical user interface and other visualizations can be used with the new performance data.

3.3 Supporting GASNet Instrumentation for a New Programming Model

In addition to UPC and SHMEM, many different programming models and tools are available on top of GASNet. Thus, one of the goals of our design is to keep the effort required to support a new programming tool fairly small. In this section, we provide a brief account of the source-code modifications that were required in order to support GSHMEM running on GASNet, after first supporting BUPC. The modifications for the GASNet and PPW measurement unit can be reused, since they are not dependent upon a parallel-programming model. The only work required is to modify the GASP callbacks provided by the performance tool in order to have two different models and provide a set of upcalls. PPW had support for GSHMEM, so we could utilize the upcall routines from the same. The total amount of source code modified to add support for GSHMEM/GASNet was less than 300 lines.

In a scenario where a particular programming model is not supported by PPW, even so our performance tool can obtain GASNet performance data by providing additional upcall routines for sending and receiving data and an upcall routine for barrier. If the programming model under consideration does not have performance-analysis support, then by reviewing performance data from GASNet a user can get a top-level view of their application behavior. This flexibility is possible because there is a one-to-one mapping between a significant number of constructs of the programming model (such as UPC or SHMEM) to the constructs in GASNet. For example, remote-memory access usually maps to one of the *gasnet_get* functions, barriers map to *gasnet_barrier_wait*, and so on.

4. Results

In this section, we analyze and experiment with the capabilities of our framework. We present case studies with both BUPC and GSHMEM. Additionally, we provide a quantitative measure of overhead involved for performance profiling. Results presented are obtained from experiments on a cluster of 16 Linux servers connected by DDR (20 Gb/s) InfiniBand (IB), where each server is equipped with a quad-core Xeon E5520 running at 2.26 GHz with 6 GB of memory.

4.1 Case Study 1: BUPC/GASNet

In this section, we demonstrate a use case of performance analysis of GASNet with a number of existing visualizations

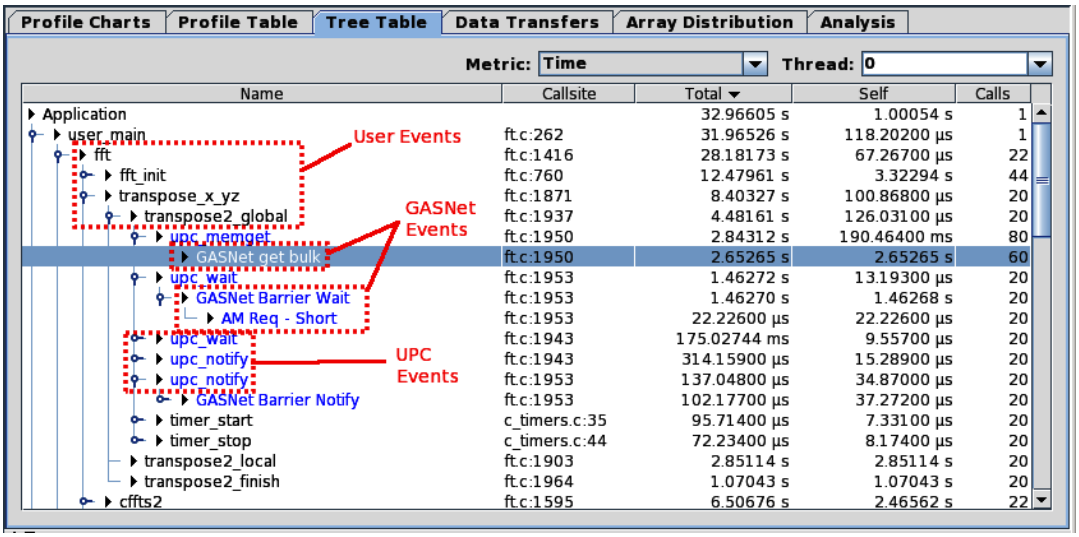


Figure 3. Tree-table visualization for FT benchmark of NPB2.4

of PPW. The performance data presented in this section is obtained by running the FT benchmark from the NAS parallel benchmark suite [20].

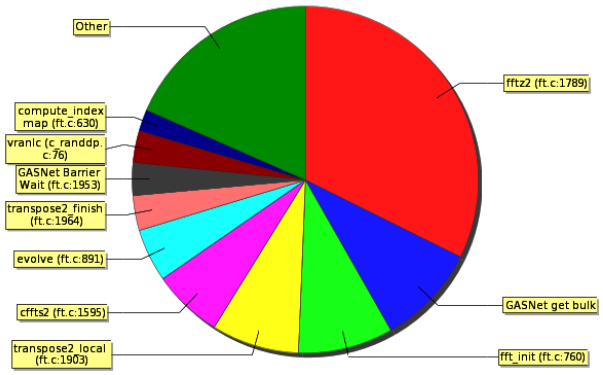


Figure 4. PPW profile metrics pie chart - self time, for FT benchmark of NPB2.4

Figure 3 shows a tree-table visualization with annotations identifying some performance-related events from BUPC, GASNet, and function-call invocations from the user application. Even though this visualization looks similar to a call-stack, it includes only the performance related events that are of interest to the programmer and not all the function calls within the application. The *user_main* in the tree-table is the *main* function in the source code of the application. With the help of tree-table view, we clearly see the relationship between different events. For example, we can see the different events starting from *user_main* until the application invokes *transpose2_global*, which triggers UPC events when it uses *upc_memget* and *upc_barrier*. These calls are handled by the BUPC runtime by using GASNet APIs. To handle *upc_memget*, the BUPC runtime uses *gasnet_get_bulk*, whereas *upc_barrier* is split into *upc_notify* and *upc_wait*, which internally uses the GASNet extended API namely, *gasnet_barrier_wait* and *gasnet_barrier_notify*. For IB, the implementation of *gasnet_barrier_wait* and *gasnet_barrier_notify* is based on the generic implementation of the extended API (built on top of the core API), which can be seen by the active-message request calls. The application was executed with four threads running across four different nodes, so as a result we see the calls to *upc_memget* invokes *gasnet_get_bulk*. If all the threads were running on the same server, we would not have seen the invocation of *gasnet_get_bulk*, since the intra-node transfer is handled by UPC runtime itself. In addition to providing a unified view of different events from application, the tree table includes other useful information, which includes call-site information (particular line of code in the application that triggered the event), total time and self time for events, and the number of invocations.

Many other visualizations are available within the PPW user interface, which can be used with performance data collected with our modified framework for GASNet. For example, Figure 4 shows the profile metrics pie chart, which provides information about total application time split as per the self time of different functions and system-related events and thus providing a top-level view of how time is split between computing, synchronization, and I/O. For example, the *fft_init* is a user-defined function, whereas the *gasnet_get_bulk* event corresponds to a blocking data transfer to read data from a different node, and the *gasnet_barrier_wait* event corresponds to synchronization with all other threads.

Another useful visualization in PPW shows the average amount of time spent by each thread on a particular event. For example, Figure 5 shows the amount of time spent by each thread waiting on *gasnet_barrier_wait* triggered at the end of function *transpose2_global* in the FT benchmark of

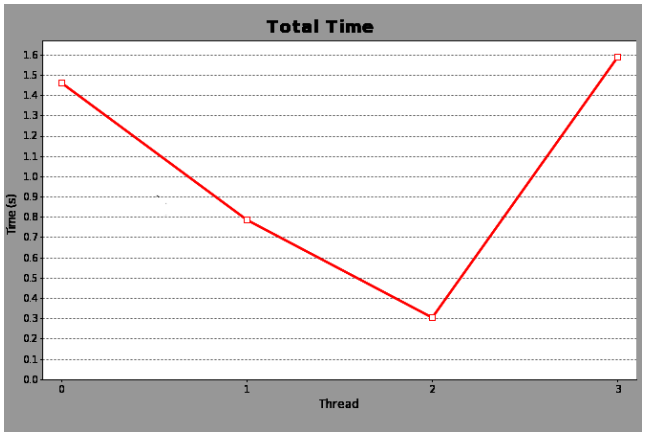


Figure 5. Visualization showing the time spent waiting on *gasnet_barrier_wait* by different threads

the NAS suite. This type of visualization is useful when there is a difference in the execution path (or computation load) among different threads.

The collection of performance data typically comes with additional cost (in terms of execution time), which is mainly due to operations such as event notification, aggregation, and basic processing of performance data. The additional overhead observed due to instrumentation, during the execution of application should be significantly small. To measure the additional overhead due to GASNet instrumentation, we executed a number of benchmark applications from the NAS suite. The measured execution time was from the first call of *gasnet_init* until the last call of *gasnet_exit*. Figure 6 shows the overhead observed for some of the NAS benchmark applications (compilation parameters CLASS=C and NP=8). The average execution overhead observed by including GASNet instrumentation compared to UPC-only instrumentation was approximately 2%, whereas the overall execution overhead for profiling compared to baseline behavior of the program was under 5%. The additional overhead incurred is quite small, considering the additional information we obtain by enabling performance analysis of GASNet.

4.2 Case Study 2: GSHMEM/GASNet

In this section, we present a case study for a 2D-FFT application written using GSHMEM. We experiment with two implementations of this application, one with *shmem_fcollect* and the other *shmem_get*. Collectives such as *shmem_fcollect* in GSHMEM are implemented on top of GASNet's collective functions [21] [22]. Our experiments involved 4 processing elements of SHMEM mapped onto different server nodes with active-message handler profiling enabled.

Figure 7 shows the tree-table views for the implementation of 2D-FFT using *shmem_fcollect*[1] and *shmem_get*, for

[1] concatenates block of data from multiple threads to an array in every thread

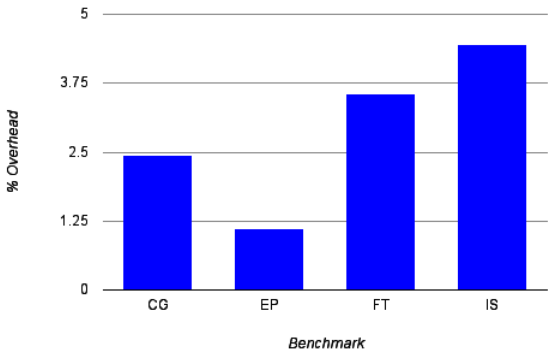
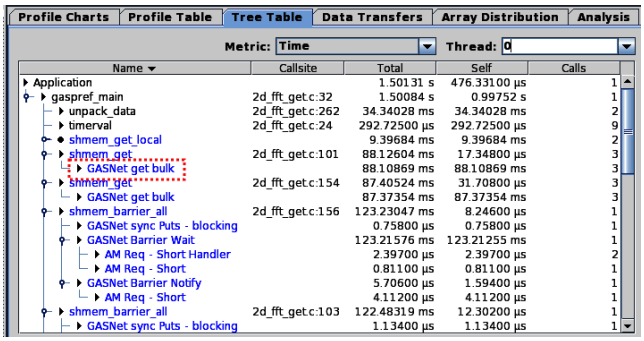
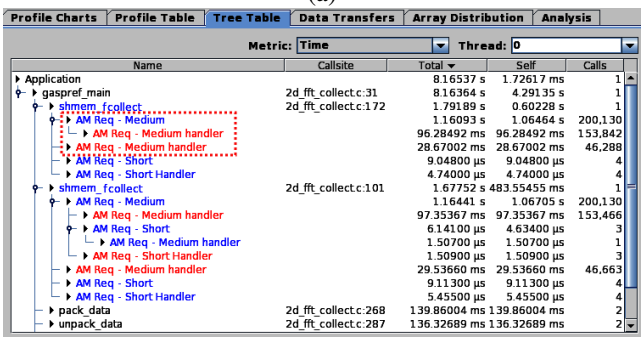


Figure 6. Overhead observed for different applications in NAS benchmark



(a)



(b)

Figure 7. Tree-table view for 2D-FFT for thread 0 using (a) *shmem_get* or (b) *shmem_fcollect*

thread 0. As in the previous case study with UPC, we can observe various events from different layers, but in addition we also notice the active message handler invocations. The events triggered by *shmem_fcollect* include a large number of active-message requests and these requests are symmetric across the different threads. The number of these active-message requests and handlers can be mapped to the communication geometry used by the collective. Our results indicate that the 2D-FFT application based on *shmem_get* performs better, as there is an implicit entry and exit barrier for every *shmem_fcollect* call. Additionally, we observe

that a large number of active-message calls are triggered by *shmem_fcollect*, in order to communicate the data between different threads, which can be seen from Figure 7(b).

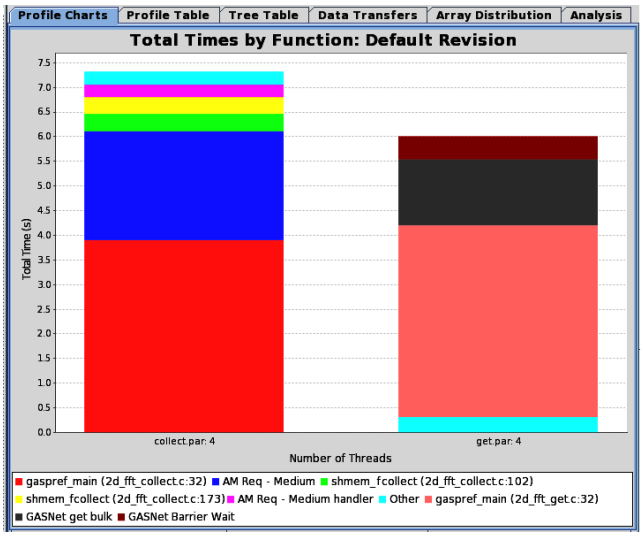


Figure 8. Snapshot showing time spent on different events in 2 different implementations of 2D-FFT

Another useful visualization within PPW is provided in Figure 8, which shows a comparison between the two different implementations (collect vs. get) and the time spent in user-defined function or events. In terms of performance, the 2D-FFT based on *shmem_get* is faster compared to the one based on *shmem_fcollect*. The application with *shmem_fcollect* spends a significant portion of time in the collective and active-message requests triggered by *shmem_fcollect*, which is comparatively higher compared to the time spent on *gasnet_get_bulk*, which is used by *shmem_get*. We can also observe that the time spent in computation by both implementations is almost identical. Similarly, there are a number of other useful visualizations for the performance data within PPW that can be used for performance analysis of applications developed using languages and libraries running atop GASNet.

5. Conclusions and Future Work

In this paper, we presented research and results with a performance-analysis framework to obtain the performance data from multiple layers, namely UPC or SHMEM, GASNet, and user applications. The addition of GASNet support enables collection of performance data with rich information to analyze the performance of the overall system. We illustrated the merits of our approach using two case studies and by showing various performance-related data from different layers in a comprehensive manner. Even though multiple layers were instrumented and measured, we observed a low overall execution overhead of under 5%. We also showed that our approach is modular and can be simply and easily integrated with other programming models by providing an account of changes made for supporting GSHMEM.

Future work on GASNet instrumentation will expand beyond profiling and focus on support for tracing, which will enable users to see the sequence of events that took place during the execution of a program and provide data on a specific instance of an event. We also plan to focus on further exploration in handling asynchronous events like active-message handlers, so that the user can see the relationship between the original request that triggered the handler and the invocation of handler itself.

Acknowledgments

This work was supported in part by the U.S. Department of Defense and the Lawrence Berkeley National Laboratory.

References

[1] S. Shende and A. Malony, *The Tau Parallel Performance System*, International Journal of High-Performance Computing Applications (HPCA), vol. 20, no. 2, pp. 297–311, 2006.

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

[3] H. Su, M. Billingsley III, and A. George, *Parallel Performance Wizard: A Performance System for the Analysis of Partitioned Global Address Space Applications*, International Journal of High-Performance Computing and Applications, Vol.24, No.4, Nov 2010, pp 485-510.

[4] The UPC Consortium, *UPC Language Specification v1.2*,http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf, 2005.

[5] *SHMEM API for Parallel Programming*, http://www.shmem.org/.

[6] Dan Bonachea, *GASNet Specification* U.C.Berkeley Tech Report (UCB/CSD-02-1207).

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

[8] *The Chapel Parallel Programming Language*, http://chapel.cray.com/

[9] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilnger, S. Graham, D. Gay, P.Colella, and A. Aiken, *Titanium: A High-Performance Java Dialect*, Workshop on Java for High-Performance Network Computing, Las Vegas, Nevada, June 1998.

[10] Numrich, R. W. and Reid, J. 1998. *Co-array Fortran for parallel programming*. SIGPLAN Fortran Forum 17, 2 (Aug. 1998).

[11] *GASNet* website, http://gasnet.cs.berkeley.edu

[12] A. Mainwaring and D. Culler, *Active Message Applications Programming Interface and Communication Subsystem Organization*, U.C. Berkeley Computer Science Technical Report, 1996.

[13] Changil Yoon, Vikas Aggarwal, Vrishali Hajare, Alan D. George, Max Billingsley III, *GSHMEM: A Portable Library for Lightweight, Shared-Memory, Parallel Programming*, Proc. of Partitioned Global Address Space, Galveston, Texas, October 2011.

[14] *Message Passing Interface Standard* Version 2.2, September 4 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[15] H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley III, and A. George, *GASP! A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models*, Proc. of Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06), Umeå, Sweden, 2006.

[16] *Parallel Performance Wizard* (PPW) tool project website, http://ppw.hcs.ufl.edu.

[17] *HPCToolkit* website, http://hpctoolkit.org/

[18] *GASNet Trace*, http://gasnet.cs.berkeley.edu/dist/README

[19] Manual page of UPC trace, http://upc.lbl.gov/docs/user/upc_trace.html

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

[21] D. Bonachea, R. Nishtala, P. Hargrove, M. Welcome, K. Yelick. GASNet Collectives Poster at SuperComputing 2006 (PDF), *Optimized Collectives for PGAS Languages with One-Sided Communication* , Nov 2006

[22] *GASNet collectives design notes*, http://gasnet.cs.berkeley.edu/dist/docs/collective_notes.txt