Tools for Benchmarking, Tracing, and Simulating SHMEM Applications

Mitesh R. Meswani[1], Laura Carrington[1], Allan Snavely[1], Stephen Poole[2]
[1]San Diego Supercomputer Center, [2]Oak Ridge National Laboratory
mitesh@sdsc.edu, lcarring@sdsc.edu, allans@sdsc.edu, stephen.w.poole@ugov.gov

***Abstract*— SHMEM communication library provides a low-latency one-side communication paradigm for parallel applications to co-ordinate their activity. Hence a trace of SHMEM calls is an important tool towards understanding and tuning SHMEM applications communication performance. Towards this end we present a suite of tools to benchmark, trace, and simulate SHMEM communication speedily and accurately.**

Specifically, in this paper we present the following three tools: (1) ShmemBench – a benchmark generator that generates user specified APIs and communication sizes to benchmark SHMEM communication, (2) ShmemTracer – a lightweight library to trace SHMEM calls in a running application, and (3) Shmem Simulator – a tool to accurately and speedily simulate SHMEM traces for different target HPC systems. The three tools presented provide a powerful experimentation tool for users to analyze and optimize performance of SHMEM applications.

I. INTRODUCTION

Two-sided communication has been the dominant protocol for developing high performance applications. However the cost of synchronization for point to point communication makes it challenging when developing systems with more than 100,000 cores. This problem will further be exacerbated when scaling for exascale systems where core counts may exceed 2^{30}. One-sided communication has been developed as a response to reduce cost of synchronization. One of the earliest implementation is the SHMEM one-sided communication library, available on Cray and SGI systems, providing a low latency mechanism for parallel tasks to co-ordinate their activities. One sided communication can take advantage of the RMA operations available to perform remote load and store operations without interrupting the remote processor. Thus, there is increasing research interest in using one sided communication such as SHMEM, UPC, to develop HPC applications.

While the choice of using SHMEM over two sided communication maybe apparent, however choice of a system to run a given SHMEM application may not be so apparent. Moreover the choice is not easy to deduce without actually running the application or in some cases the machine may not exist as maybe the case when designing new hardware. In such situations performance modeling can help the user decide about the choice of machine. For the performance model to be useful it should be accurate and with reasonable speed. To solve this issue in this paper we present a set of tools that can be used to benchmark, trace, and simulate SHMEM applications. In particular we present three tools that provide a powerful experimentation tool for users to analyze and optimize performance of SHMEM applications and summarized below:

- *ShmemBench*, a configurable benchmark generator that generates C, C++, and Fortran 90 benchmarks. This tool takes as input a user specified subset of SHMEM APIs and their corresponding message sizes and generates as output C, C++, and Fortran90 benchmarks with the specific APIs; each API call is timed. Hence, a user can run the benchmark on a target Cray System to benchmark its SHMEM communication performance.
- *ShmemTracer*, a lightweight tracing library that traces SHMEM API calls in a running SHMEM application. The library generates trace records for each SHMEM call and stores them in a compact binary format. Each trace record encapsulates the parameters passed to the call and time spent to complete the call. The tracing library can be used without binary modification and adds very little overhead. Additionally, we also provide a reporting utility that converts the binary trace into human readable ASCII text. The report provides detailed summary of each API call and as such users can easily parse the report to generate more complex reports.
- *Shmem Simulator*, a simulation framework that consumes SHMEM traces and applies user specified communication models to calculate applications communication time on a target system. The simulator is developed within the open source PSINS framework which is a network simulator for parallel architectures. The simulator takes as input SHMEM traces and a target system configuration and replays the traces and reports an applications communication time on the target system.

The outline of the rest of the paper is as follows: Section II describes ShmemBench, Section III describes ShmemTracer, and Section IV describes the Shmem Simulator. Experiments demonstrating the usability, efficiency and accuracy of the Tracer and Simulator are described in Section V Finally we discuss related work in Section VI and conclusions and future work is discussed in Section VII.

II. SHMEMBENCH

ShmemBench is a configurable benchmark generator designed to allow a user to verify and validate the installation of the ShmemTracer tool. The design of the benchmark is shown in Figure 1. The benchmark generator, written in python, takes as input a configuration file and test templates. The configuration file is used to specify one of five classes of SHMEM API calls, described below, and supply appropriate parameters for them. While, the test templates, one for each SHMEM API class, are used to store the static portion of the source code as well as place holders to allow the generator to write the dynamic portion of the code. Based on the configuration the generator produces the corresponding source code in C, C++, and Fortran90. The user need only compile the source codes and run it on the target system.

The configuration file has the following format. Each line specifies the test number, test class, number of elements to transfer, and stride. The test number is used to uniquely identify the test and thus allowing a user to test the same test class with different transfer size and strides. The test class specifies one of five tests: #1 – shmem put calls, #2 – shmem get calls, #3 – collective calls including reductions broadcasts and barriers, #4 – synchronization calls include fence, quiet, and wait calls, and finally #5 – atomic operations. The transfer size specifies the number of elements to be used by put, get, and collective calls; it is ignored by synchronization and atomic calls. The stride is used for strided iput and iget calls; stride is ignored for other calls. A typical configuration file is shown in figure 2. In this figure we specify 3 tests, test 1 and test 3 are used to specify put calls to transfer 100, 5000 elements and stride 1, 4 respectively, while test 2 is used to specify collective calls to transfer 900 elements.

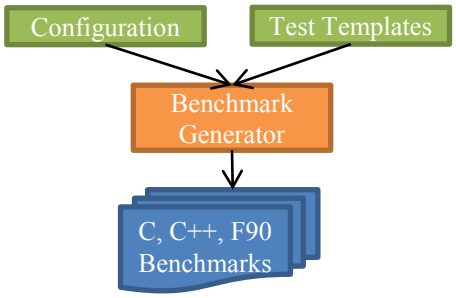

Figure 1. High-level design and flow of information of ShmemBench.

TESTNUM	TESTCLASS	NELEMS	STRIDE
1	1	100	1
2	3	900	1
3	1	5000	4

Figure 2. ShmemBench Configruation.

III. SHMEMTRACER

ShmemTracer is a tracing tool to collect information about SHMEM events in a running application. The tracer is built as a shared library and uses the LD_PRELOAD mechanism to trace SHMEM events. Briefly each SHMEM api call made by a task is captured by the tracer library, and the relevant parameters are stored in an event record and the tracer then redirects to allow the original SHMEM call to proceed. Those calls are timed, within the tracer and are also included in the event trace record.

The tracer has been designed to be flexible. The events of interest are listed in an XML specification file. To specify an event to be traced, relevant information about the event parameters are included in the XML specification file. The most relevant tags and an example specification for *shmalloc* is shown in figure 3. Based on the specification a python script then constructs the event wrappers. Thus, adding or deleting an event is easily facilitated by the XML specification and promotes greater code maintainability.

TAG	DESCRIPTION
Function	Declares the name of the SHMEM API
Arg	The arguments accepted by the function
Trace	The trace record destination to store argument

```
<function ret="void*"> shmemalloc
    <arg type="size_t"> size </arg>
    <trace dest="bytes"> size </trace>
</function>
```

Figure 3. XML Specification for Tracer.

For each event to be traced, the tracer builds an event wrapper with the same function signature. When the tracer library is loaded using LD_PRELOAD, the function calls are first redirected to the tracer. The event wrappers store the arguments for each call in an event record. The calls are then redirected to call the original SHMEM library and those calls are timed and recorded in the event trace. Additionally, the tracer also times spent between SHMEM communication events, or CPU time called as *CPUBurst*. To gather *CPUBurst* events, the library uses timers at the end and the beginning of each SHMEM routine replacement so that when a SHMEM function is called, the time spent since the end of the last SHMEM call to the current call is recorded in the trace.

Given *n* parallel SHMEM tasks, the tracer generates *n* trace files, one for each task. In order to inspect the traces we have provided a utility called *shmembin2txt* which parses the binary trace file and produces a user readable ASCII text report file. This file contains for each event the parameters passed to the SHMEM function and also reports the wall clock time spent executing the particular function call. A user may gather valuable statistics by parsing the ASCII report file.

In order to run simulations, the *n* binary trace files are post processed by another utility called *shmem2PSINS* which converts the trace and converts into a format that can be read by the network Simulator. The conversion utility reads the *n* trace files and repackages the events into a single trace file which can then be passed as input

to the SHMEM Simulator. In addition to the events the conversion utility also performs certain book keeping operations that are required by the SHMEM simulator.

IV. SHMEM SIMULATOR

Shmem Simulator is an event simulator and developed as part of the PSINS [1] framework. The flow of information in the network simulator for SHMEM is shown in figure 4. The simulator takes as input SHMEM communication event traces of an application and parameters that model the target machine and replays the trace in the simulator. The simulator simulates both the computation and communication time of an application and at the end of the simulation provides detail statistics and predicted runtime.

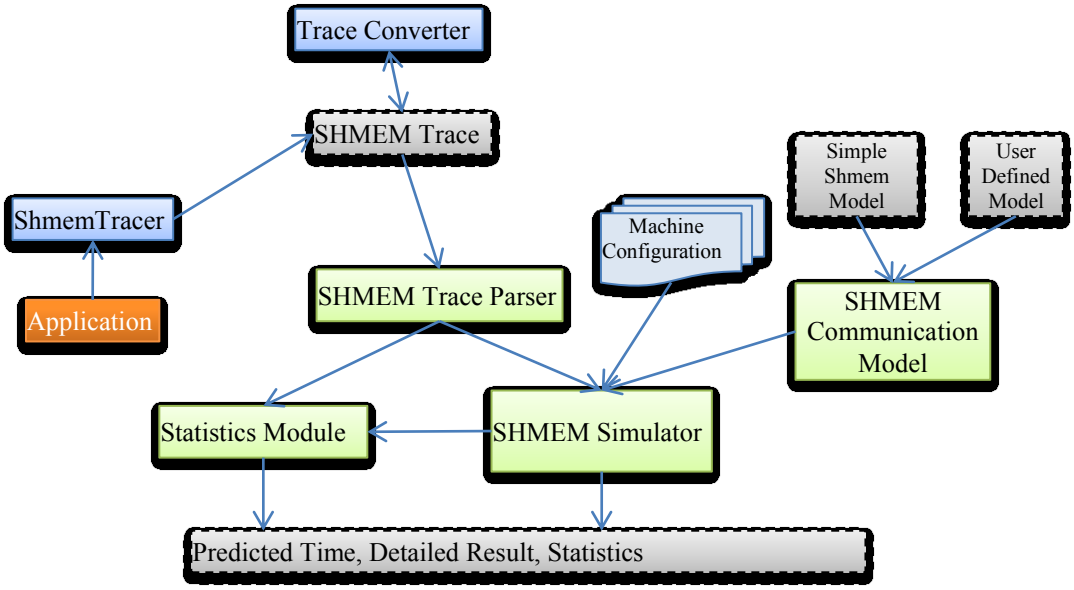

Figure 4. Design of SHMEM Simulator and flow of information.

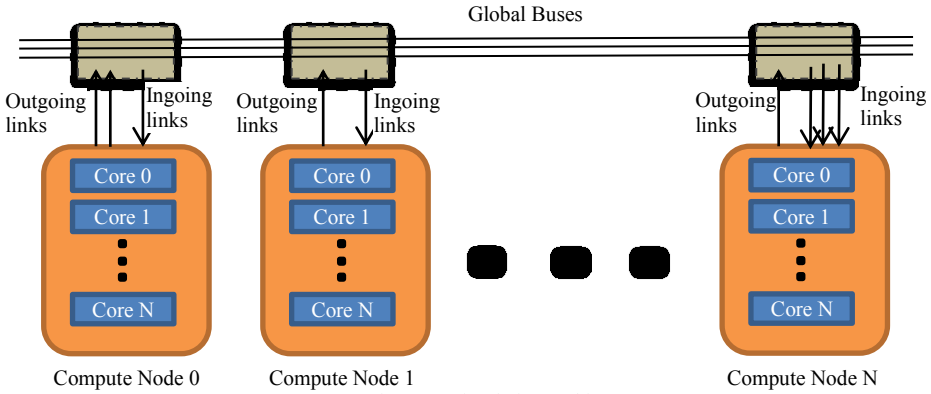

Figure 5. Simulation architecture.

The original PSINS framework supported only simulation of MPI communication and in this research we have added support for SHMEM. The SHMEM simulation shares many components of the MPI simulation such as the representations of the target machine described in Section VA and the general event queue implementation of the simulation described in Section VB. The major difference between MPI and SHMEM simulation is the communication model which is described in Section VC. Additionally we also developed a parser for SHMEM traces.

A. Machine Representation

To simulate the target system, the network simulator assumes that the machine is composed of compute nodes interconnected by global buses as shown in figure 5. Each compute node is made up of processing elements that are connected to the global buses by ingoing and outgoing links. The description of the machine model is flexible and the user can configure the number of each individual elements. Hence, for example the user can specify different speeds for each individual processing element or specify different number of ingoing and outgoing bus links. This flexibility allows the description to cover a wide range of machine architectures found in HPC.

The description of the target machine system for simulation is given to the simulator in an ASCII configuration file. The parameters include the number of nodes, the number of buses, and for each bus the

bandwidth and latency for inter-node communication. For each compute node, the user needs to specify the number of cores, the number of incoming and outgoing links from/to busses, bandwidth and latency for intra-node communication, CPU ratios for each core with respect to base system. The simulator uses the CPU ratios to estimate the computation time. The ratio is used as a factor to project the speed of CPU burst on target system relative to the base system. Prior research [2, 3] has shown that CPU ratio is an effective means to predict CPU burst. The user can also specify the task to node mapping for each SHMEM task, if not given a default round robin mapping is assumed.

B. Simulation Overview

The simulator has been implemented as an event queue simulator based on priority queues and replays the entire input event trace. When an event is read from the input trace its priority is calculated based on the earliest time it will be ready for execution. This is calculated using the events task timer to calculate the priority at the time of insertion of the event in the queue. If the event is not ready to execute for example a blocking event such as a barrier or global reduction, then the event is reinserted into the event queue with reduced priority. When an event executes, it is deleted from the event queue and its time is tagged with the execution time plus the time it had to wait. The wait time of an event is the time an event had to wait to begin execution, for example global events such as barriers may have to wait for other tasks to reach the barrier. The execution time of the event is also used to update its task timer.

The execution of an event during simulation depends on the type of the event and the state of the system at each event execution such as the load on the network, contention and so forth. CPU burst events use the CPU ratio for calculating the execution time. Peer to Peer communications are posted as soon as the event is ready to execute. While blocking events such as global synchronization and reductions are kept in the queue until all participating tasks post the same event. When the events are ready to execute, the execution time for the events are calculated using the communication models which are described in Section VC.

The simulator also includes a statistics module to report detailed information about the simulation. The statistics module collects information such as frequencies of execution per event; break down of compute and communication times per task, and the execution time for each event type on the target system. It also reports the waiting time for each event which can be used to indicate the load imbalance in a system. The module also reports metrics such as average communication size by event, by task, and on node and off node communication. The detailed report provides valuable insights about bottlenecks in the system and opportunities for optimization.

C. Communication Models

The communication models are used to calculate the execution time of an event. We provide built in models for SHMEM simulation. In addition the PSINS framework allows a user to easily define new models.

1) Built-in Models

We implement a simple model for our built-in model. The simple models uses the best bandwidth and latency and assume that infinite resources are available, that is we assume that there is not contention for network resources such as buses and a task can send messages without any contention. For peer to peer communication for blocking events the execution time is the latency plus the time to transfer the message, whereas for non-blocking the time is calculated as simply the latency. For collective communication the time for each event is either linear, logarithmic or some other scaling factor with respect to the number of participating tasks. In this sense our simple model simply gives the lower bound on execution time. While this model is simplistic, it is sufficient to demonstrate the simulator and we leave addition of more complex models as part of future work.

2) Adding new models

The simulation framework allows user to easily add new models. The simulator provides the base class *SHMEMModel*, with some C++ virtual methods. The user need only implement the virtual methods to implement the calculation of execution time and scheduling of resources. The models can be added as an extension of the base class and implement the virtual functions. The design of the virtual functions is model specific and can be done in few hundred lines of code.

V. Experiments and Results

To demonstrate the efficacy of our tracing and simulation tools we used them to trace and simulate a parallel 3dfft (p3dfft) application [10]. P3dfft is a library to perform 3DFFT operation using 2D decomposition. 3D FFT is an algorithm widely used in fields such as turbulence studies, climatology and material science. The P3dfft library has been optimized for large data sets and has been shown to scale to large core counts [10]. In our research we used a driver which is available with the library. The driver program uses the library to perform a forward transform and then a backward transform on a 3D array.

The traces were collected on two large Cray HPC machines Hopper and Jaguar. Hopper is a Cray XE6 supercomputer hosted at NERSC with a peak performance of 1.26 petaflops. Hopper has 6,384 compute nodes, with each node made up of 2 twelve-core AMD MagnyCours processors. The nodes are interconnected by a custom cray network and uses Cray Gemini network to route communication. Jaguar is a Cray XK6 supercomputer hosted by NCCS with a peak performance of 3.3

petaflops. Jaguar is composed of 18,688 compute nodes, with each node made up of a single 16-core AMD processor. Jaguar also uses the Gemini interconnect to route communication. Both hopper and Jaguar support RMA operations.

The traces were collected for processor counts from 128 to 16,386. The data sets were scaled with core count. The 3dfft test application is a small kernel that runs within minutes. However, the test produces at the largest core count approximately 1 billion events which is a reasonable problem size to test scalability and overhead of the tools. In future full application tests are planned. The traces were then simulated on the login nodes of Jaguar and Hopper. However we only simulated processor counts 1024 to 4096. For larger CPU counts we found that we are limited by the number of open file handles that are allowed to user program. We are working to find a workaround around these limitations in the simulator.

A. *Trace Size and Simulation Times*

Shown in figure 6 is the number of SHMEM events collected per CPU count. This figure illustrates that the number of events is a linear multiple function of CPU count. The sizes of traces collected for each event count is given in figure 7. The figure illustrates that the size of event traces grows linearly as the event count grows. The sizes range from 0.008GB to 56GB.

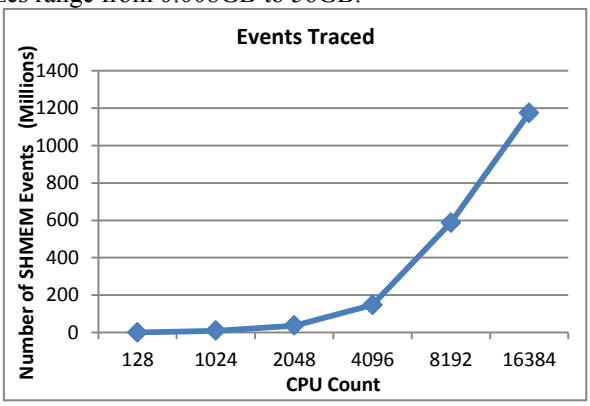

Figure 6. SHMEM Events Per CPU Count.

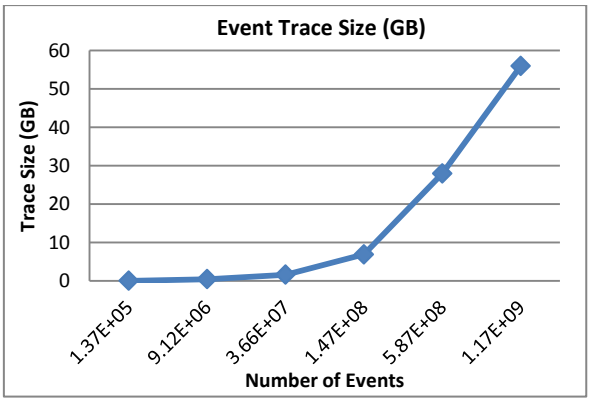

Figure 7. Trace Size (GB) as function of event count.

The traces are then fed to the simulator and the time for running the simulation is shown in Figure 8. The figure plots the time to finish the SHMEM simulation for CPU counts 128 to 4096. The results show that simulation in the worst case takes about 15X time than actually running the application. The simulation time grows linearly with increasing CPU counts and suggests that a parallel implementation maybe required as we move towards larger CPU counts. Note that we are not able to simulate larger core counts because of limitations of number of open file handles per user program.

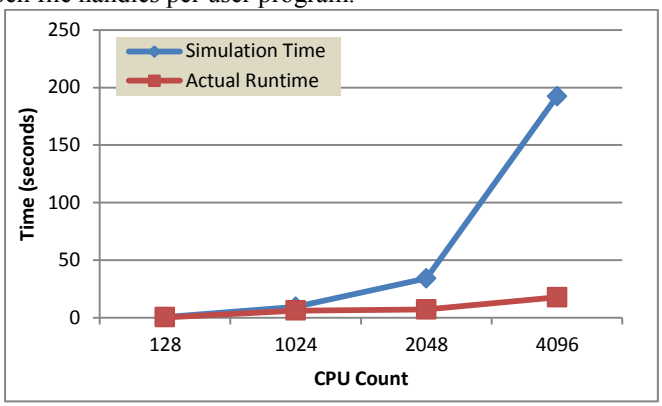

Figure 8. Simulation time comparison for different CPU counts.

It is also important to quantify the overhead of collecting traces by Shmem Tracer. We observed that the overhead for CPU counts below 4096 was negligible, and for the largest core count of 16384 cores the overhead was only 13%.

B. *Simulation Accuracy*

Next we show results of simulation accuracy. We investigate the prediction accuracy of the simple model for predicting times on Jaguar. We compare the prediction time for total communication time for a 128, 1024, and 2048 core job. The results are shown in Figure 9. The error in the prediction stems from our usage of the simple model, and we can improve upon the model to improve accuracy of prediction. Complex models are planned as part of future work.

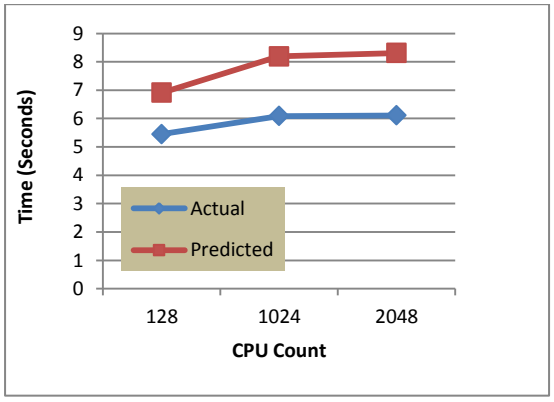

Figure 9. Prediction accuracy.

VI. BACKGROUND

Profiling and tracing of SHMEM one-sided communication is supported in Kojak [8] and Tau [9]

frameworks. Tau traces and profiles entry and exit of SHMEM call and however, it does not record the transfers in tracing mode. While Kojak profiles both SHMEM enter and exit calls and records transfer statistics in traces. The profile information collected by Kojak is then post processed to identify areas of performance bottlenecks and other performance properties which may provide insight into the behavior of programs. However, both TAU and Kojak do not provide a network simulator and to the best of our knowledge ours is the first work implementing a network simulator for SHMEM.

There have been numerous efforts for tracing and simulating parallel architectures. Proteus [4] was one of the earliest to implement parallel architecture simulation. The design of the simulator was modular in nature and this separation allowed it to be easily customized. Some of these design principles have been followed in PSINS [1] framework with support for MPI simulation. PSINS allows a user to customize many pieces of the architecture. PSINS also allows users to define their own parser and communication model modules. In this research we have incorporated SHMEM simulation within the PSINS framework. We implemented our own parser, models, and reused the general simulation flow in PSINS.

Simulators such as LAPSE [5], MPI-SIM [6] and Wisconsin Wind Tunnel [7] implemented parallel simulations but typically are execution driven and full system simulators. These simulators are more complex and generally simulation time grows because of the need to simulate the entire system. The Dimemas [3] project separates network simulation from the remaining system and has been implemented and has many similarities to PSINS. However, Dimemas is not open source and may not be suitable for community development.

VII. Conclusions and Future work

Performance modeling can be a valuable tool to provide insights about tuning of application and systems and help design future systems. Fast accurate simulations can be a valuable resource when evaluating choice of current and future systems. One-side RMA communications can reduce the time spent in communication and possible scale to exascale architectures. Modeling and simulating tools for one-sided communications such as SHMEM may prove valuable to improve understanding of RMA operations.

The tools presented in this paper are a first at simulating SHMEM applications. We provide a low overhead tracing too and fairly fast simulation tools and simple models for prediction. The framework itself allows users to add more powerful complex models and tracers. In the future we will add support for Open SHMEM and continue adding complex communication models.

Acknowledgment

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, Supported by the Extreme Scale Systems Center at ORNL, which is supported by the Department of Defense.

References

[1] Mustafa M. Tikir, Michael Laurenzano, Laura Carrington, and Allan Snavely, “PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications,” in *Proceedings of Euro*-Par, 2009.

[2] D.H. Bailey and A. Snavely, “Performance Modeling: Understanding the Present and Predicting the Future”, EuroPar, 2005.

[3] R. Badia, J. Labarta, J. Giménez and F. Escalé. "Dimemas: Predicting MPI Applications Behavior in Grid environments," Workshop on Grid Applications and Programming Tools, 2003.

[4] E. Brewer, C. Dellarocas, A. Colbrook and W. Weihl. "Proteus: A High-Performance Parallel Architecture Simulator," MIT Technical Report MIT/LCS/TR-516, 1991.

[5] P. Dickens, P. Heidelberger and D. Nicol. "A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs," Proceedings of the 8th Workshop on Parallel and Distributed Simulation, 1994.

[6] S. Prakash and R. Bagrodia. "MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs," Proceedings of the Winter Simulation Conference, 1998.

[7] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis and D. Wood. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1993.

[8] Mohr, B., Kühnal, A., Hermanns, M.-A., Wolf, F. "Performance Analysis of One-sided Communication Mechanisms," Mini-Symposium "Tools Support for Parallel Programming", Proceedings of Parallel Computing (ParCo), Malaga, Spain, September, 2005.

[9] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Proling and Tracing for Parallel Scientic Applications using C++. In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134–145. ACM, Aug. 1998.

[10] http://code.google.com/p/p3dfft/