# A Distributed, Programming Model-Independent Automatic Analysis System for Parallel Applications

Hung-Hsun Su, Max Billingsley III, Alan D. George
{su, billingsley, george}@hcs.ufl.edu

High-performance Computing and Simulation (HCS) Research Lab
Dept. of Electrical and Computer Engineering
University of Florida, Gainesville, Florida 32611-6200

**Abstract**
Developing a high performance parallel application is difficult. An application must often be analyzed and optimized by the programmer before reaching an acceptable level of performance. Performance tools that collect and visualize performance data can reduce the effort needed by the user in the non-trivial optimization process. However, as the size of the performance dataset grows, it becomes nearly impossible for the user to manually examine the data and find performance issues. To address this problem, we have developed a new analysis system to automatically detect, diagnose, and possibly resolve bottlenecks. In this paper, we present the architecture and the distributed, peer-to-peer analysis process of a programming model-independent analysis system, which includes a range of useful analyses such as scalability analysis and known-bottleneck detection. We then describe the details of an initial sequential implementation of the system that has been integrated into our Parallel Performance Wizard (PPW) tool. Finally, we provide correctness and performance results for this initial version and demonstrate the effectiveness of the system through two case studies.

**Keywords:** Parallel Performance Wizard, automatic analysis system, Unified Parallel C, SHMEM, MPI.

## 1. Introduction

Researchers from many scientific fields have turned to parallel computing in pursuit of the highest possible application performance. Due to the combined complexity of parallel execution environments and programming models, however, achieving good application performance is not always guaranteed. Developers often must pursue an iterative analysis and optimization process before their application reaches an acceptable level of performance. To facilitate this process, many performance analysis tools have been created over the years. Unfortunately, these existing tools were often developed to specifically target a small set of programming models, generally MPI and OpenMP, and are not easily extensible to support other models. As a result, programmers using newer models, such as those in the partitioned global-address-space (PGAS) family, have been forced to analyze their applications manually.

To facilitate tool support for newer parallel programming models, we devised the Parallel Performance Wizard (PPW) event-based performance tool framework previously introduced in [1]. Our design makes use of a generic-operation-type abstraction and a data collection process facilitated by a standardized global address space performance (GASP) interface [2]. The combination of these two techniques significantly lowers the dependency of the tool on its supported programming models and as a result minimizes the effort needed to include addition model support. Using the design, we successfully created the PPW performance tool that currently supports Unified Parallel C (UPC), SHMEM and MPI [3].

The original version of the PPW tool collects and manages performance data for applications under analysis and presents the data to the user via a set of intuitive visualizations. While we have previously demonstrated how the tool can be useful to users, a nontrivial amount of effort and expertise is still required of the user to successfully analyze and optimize an application. This situation arises in particular when evaluating the performance of an application that generates a very large amount of data (i.e. a long-running program or a program run on a large system), where it becomes nearly impossible to manually analyze the data even with the help of the tool. This problem exposes the need for an automatic analysis system that can detect bottlenecks, determine their causes, and possibly provide resolution strategies for the user.

In this paper, we present a new automatic analysis system that extends the capabilities of our PPW tool. The proposed system supports a range of analyses that no single existing system provides and uses novel techniques such as baseline filtering and a distributed analysis process requiring minimal raw data exchange. In addition, because it is based on the generic-operation-type abstraction introduced in [1], the analysis framework is applicable to any parallel programming model with constructs that can be mapped to the supported operation types.

The rest of the paper is organized as follows. In Section 2, we define common terms used throughout the paper. In Section 3, we provide an overview of automatic analysis and existing analysis systems. In Section 4, we present the architecture of our automatic analysis system and describe how to undertake the analysis process in an efficient manner. Next in Section 5, we describe the initial sequential implementation of this system and demonstrate the effectiveness of the system through correctness and performance tests and application studies. Finally, we conclude the paper and give directions for future research in Section 6.

## 2. Terminology

In this section, we define some important terms used in the remainder of the paper. A performance *property* (or *pattern*) defines an execution behavior of interest within an application. A performance *bottleneck* is a performance property with non-optimal behavior. Bottleneck *detection* (or *identification*,

*discovery*) is the process of finding the locations (system node, line of code, etc.) of performance bottlenecks. *Cause analysis* is the process of discovering the root causes of performance bottlenecks[1] (e.g., late barrier entrance caused by uneven work distribution). Bottleneck *resolution* is the process of identifying potential strategies that may be applied to remove the bottlenecks. *Automatic optimization* refers to source code transformation and/or changes in the execution environment made by the tool to improve application performance. Finally, a *hotspot* is a portion of the application that took a significant percentage of time to execute and thus is a good candidate for optimization.

## 3. Overview of Automatic Analysis and Systems

Automatic (or automated) analysis is a tool-initiated process to facilitate the finding and ultimately the removal of performance bottlenecks within an application. The entire process may involve the tool, with or without user interaction, performing some or all of the tasks illustrated in Figure 1 on the application under investigation. Note that in the figure, performance data collection refers to the gathering of additional data on top of what the tool collects by default.

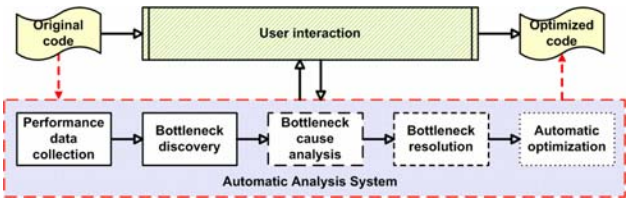


*Figure 1 – Tool-assisted automatic performance analysis process*

In the remainder of this section, we provide an overview of existing work relating to automatic analysis.

The APART Specification Language (ASL) [4] is a formal specification model introduced by the APART [5] working group to describe performance properties via three components: a set of *conditions* to identify the existence of the property, a *confidence* value to quantify the certainty that the property holds, and a *severity* measure to describe the impact of the property on performance. The group used this language to provide a list of performance properties for the MPI, OpenMP, and HPF programming models and noted the possibility of defining a set of base (model-independent) performance property classes.

HPCToolkit and TAU are examples of tools providing features to evaluate the scalability of an application using profiling data. HPCToolkit uses the timing information from two experiments to identify regions of code with scalability behavior that deviates from the weak or strong scaling expectation [6]. PerfExplorer is an extension of TAU that generates several types of visualizations that compare the execution time, relative efficiency, or relative speedup of multiple experiments [7]. In addition, PerfExplorer includes techniques such as clustering, dimension reduction, and correlation analysis to reduce the amount of performance data the user must examine.

[1] Because bottleneck detection and cause analysis are closely tied to each other, in some literature they are together referred to as the bottleneck detection process.

Periscope, KappaPI-2, and KOJAK are knowledge-based tools that support the detection of well-known performance bottlenecks defined with respect to the programming model. The advantage of a knowledge-based system is that little or no expertise is required of the user to successfully analyze the program. Periscope supports online detection of MPI, OpenMP, and memory system related bottlenecks (specified using ASL) through a distributed hierarchy of agents that evaluate the profiling data [8]. KappaPI-2 is a post-mortem, centralized, tree-based analysis system that supports bottleneck detection, cause analysis, and bottleneck resolution (via static source code analysis) using tracing data [9]. Finally, EXPERT is a part of KOJAK that supports post-mortem bottleneck detection and cause analysis of MPI, OpenMP, and SHMEM bottlenecks (specified using ASL). The developers recently introduced an event-reply strategy to allow parallel, localized analysis processing which has been successfully applied to MPI [10], but it remains questionable whether such a strategy works well for other programming models.

Hercules [11] is a prototype knowledge-based extension of TAU that detects and analyzes causes of performance bottlenecks with respect to the programming paradigm (such as *master-worker*, *pipeline*, etc.) rather than the programming model. An advantage of this system is that it can be used to analyze applications written in any programming model. Unfortunately, the system cannot handle applications developed using a mixture of paradigms or that do not follow any known paradigm at all, making it somewhat limited in applicability.

Paradyn's online $W^3$ search model was designed to answer three questions through iterative refinement: *why* is it performing poorly, *where* are the bottlenecks, and *when* did the problems occur [12]. The $W^3$ search system analyzes instances of performance data at runtime, testing a hypothesis which is continually refined along one of the three question dimensions. The $W^3$ system considers hotspots to be bottlenecks, and since not all hotspots contribute to performance degradation (they could simply be performing useful work), the usefulness of this system is somewhat limited.

The main idea behind the design of NoiseMiner, a component of the Projections tool, is that events of similar type should have similar performance under ideal circumstances [13]. Utilizing this assumption, the system makes a pass through the trace log, assigns an expected performance value to each event type, and then identifies specific trace events with performance that do not meet the expectations (i.e. noisy events).

Performance Assertions (PA) is a prototype source code annotation system for the specification of performance expectations [14]. Once performance assertions are explicitly added by the user, the PA runtime collects data needed to evaluate these expectations and selects the appropriate action (e.g. alert the user, save/discard data, call a specific function, etc.) during runtime. IBM has also developed an automated bottleneck detection system enabling the detection of arbitrary performance properties within an application [15]. The system supplies users with an interface to add new performance properties using pre-existing metrics and to add new metrics needed to formulate the new properties. With both of the above systems, a certain degree of expertise is required of the user to formulate meaningful assertions/properties.

Each of the above approaches has made a contribution to the field of automatic performance analysis. Each also has particular drawbacks that limit its effectiveness or applicability. In light of

ongoing progress in and the ever-increasing complexity of parallel programming models and environments, we have sought to make corresponding progress in effective analysis functionality for a variety of modern programming models.

## 4. PPW Automatic Analysis System Design

We present a new automatic analysis system that is novel in several aspects. First, our system performs a range of analyses including scalability analysis, load-balance analysis, common bottleneck detection, and cause analysis, whereas other systems support a small set of analyses. Second, we introduce a new baseline filtering technique[2] to identify performance bottlenecks via comparison to expected performance values. Our technique is more accurate compared to that used in NoiseMiner. Third, we have developed a scalable, distributed analysis processing technique that can be accomplished by multiple agents in parallel to reduce the analysis processing time. The process is also designed such that raw data transfers between agents are minimized. In most cases, our system can identify bottlenecks using only local data and requires only a small amount of remote data to diagnose the causes of bottlenecks.

Our analysis system is also based on the same (model-independent) generic-operation-type abstraction underlying our PPW tool. As a result, the system can be easily adapted to support any parallel programming model and naturally supports the analysis of mixed-model applications (i.e., programs written using two or more models). Furthermore, the use of this abstraction improves the system's capabilities by allowing in-depth analysis of some user-defined functions. For example, by simply telling the system to treat a user-defined upc_user_wait_until function in a UPC program as a *wait-on-value-change* operation (adding one line in the event type mapping), the system is able to determine the cause of any delays associated with this function.

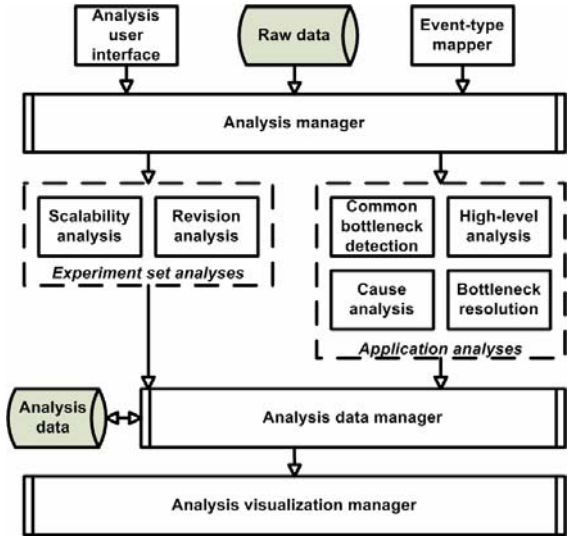


*Figure 2 – PPW automatic analysis system architecture*

[2] The technique is new for automatic analysis but readily used in system performance evaluation.

### 4.1. Design Overview

The high-level architecture of the PPW automatic analysis system is depicted in Figure 2. The analyses supported by the system are categorized into two groups: application analyses (common bottleneck detection, cause analysis, and high-level analysis) which deal with performance evaluation of a single run; and experiment set analyses (scalability and revision analysis) to compare the performance of related runs. The design focuses on providing analyses to help both novice and expert users in optimizing their application via source code modification.

We now present the distributed, localized analysis processing mechanism, which is peer-to-peer based and consists of up to N agents (where N is the application system size): 0 to N–1 non-head agents, each of which has (local) access to raw data from a set of nodes, and one head agent that also performs global analyses, which require access to (profiling) data from all nodes. This inherently parallel design is intended to support analysis of large-scale applications in a reasonable amount of time. In Figure 3, we illustrate the types of analyses conducted and the raw data exchange needed for all agents in an example 3-agent system.

Our analysis process can be broken down into several distinct phases. Figure 4 depicts the analysis workflow for an agent in the system, separated into the following four phases: the global analysis phase, detection phase, cause analysis phase, and resolution phase. Each of these phases is described in more detail in the following subsections.

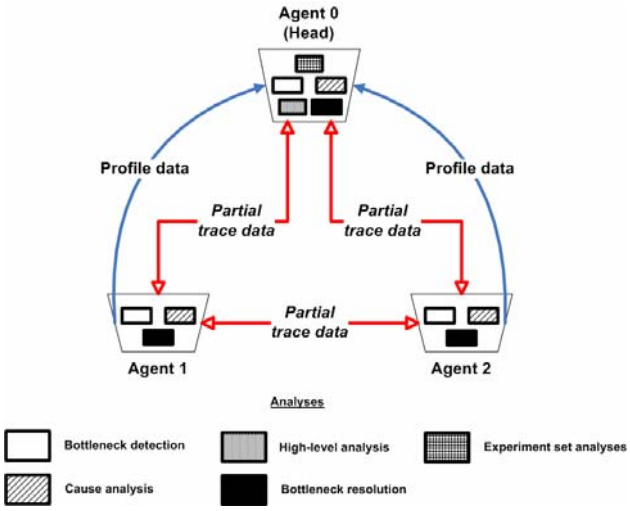


*Figure 3 – Example analysis processing system with 3 agents showing the analyses each agent performs and raw data exchanged needed between agents*

### 4.2. Global Analysis Phase

In the global analysis phase, each agent sends its portion of the profiling data to the head agent where one or more analyses are independently conducted. Analyses performed during this phase include scalability analysis, revision analysis, and high-level application analysis.

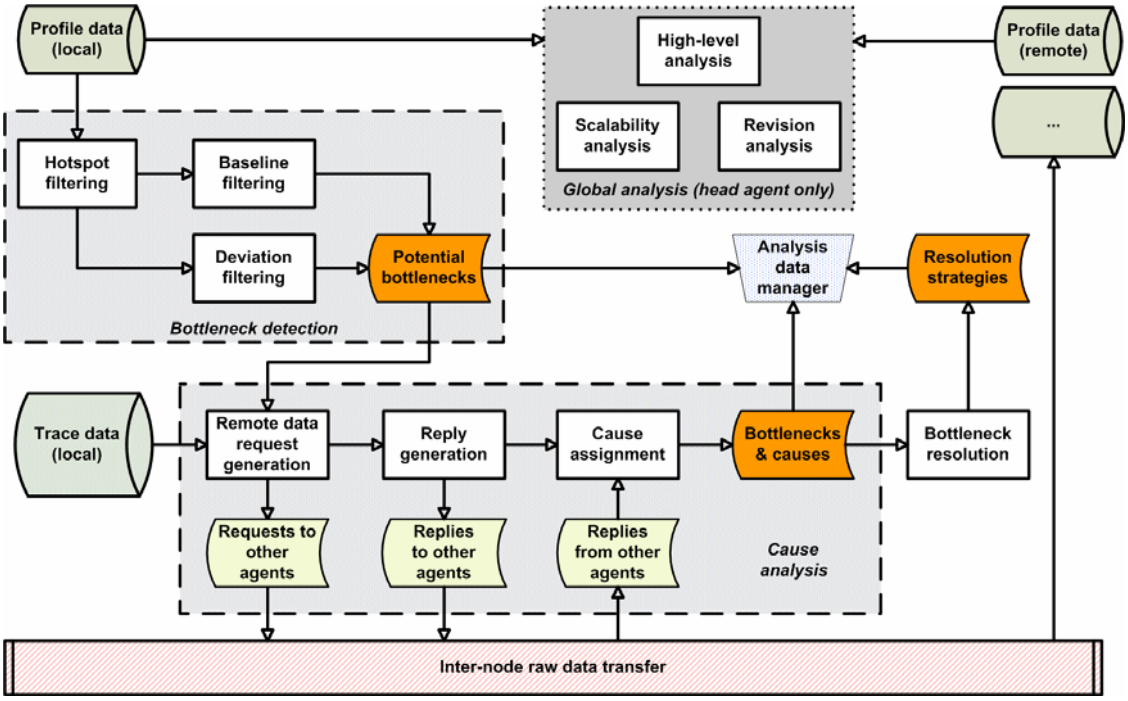


*Figure 4 – Analysis process flowchart for an agent in the system*

Scalability analysis and revision analysis are used to compare the performance of multiple related experiments. Scalability analysis is used to evaluate an application's scalability using two or more experiments on different system sizes. Using the experiment with the smallest number of nodes, the head agent calculates the scaling factor (a.k.a. relative speedup, the ratio of performance improvement over the size increase) for all other experiments. A scaling factor of 1 indicates that the application exhibits perfect scalability, while a value approaching 0 suggests very poor scalability. On the other hand, revision analysis is used to evaluate the performance of various versions of an application. In particular, it can be used to determine whether or not code changes in a particular version improved the program's performance.

High-level analysis is mainly used to detect bottleneck nodes that, when optimized, could improve the application performance for a single experiment. To do this, the head agent first calculates the computation, communication, and synchronization time for all nodes in the system and selects the nodes with largest % computation as the bottleneck nodes.

### 4.3. Detection Phase

The next analysis phase is the detection phase, during which each agent examines its portion of the (local) profiling data and identifies bottleneck profiling entries. For each of the profiling entries, the agent first checks whether or not that entry's total execution time exceeds a preset percentage of the total application time. The purpose of this filtering step is to focus the analysis effort on portions of program that would noticeably improve the performance of the application when optimized.

Next, the agent decides if the identified hotspot entry is a bottleneck by applying one of the following two expected value comparison methods. With the baseline comparison method, the agent marks the entry as a bottleneck if the ratio of its average execution time to its baseline execution time – the minimal amount of time needed by a given operation to complete its execution under ideal circumstances – exceeds a preset threshold.

If the baseline comparison method is not applicable (e.g., because the entry is a user function or no baseline value has been collected for the entry), the agent uses the alternative deviation evaluation method. With this method we make use of the following assumption: under ideal circumstances, when an event is executed multiple times, the performance of each instance should be similar to that of other instances (the same assumption is used in NoiseMiner). Thus for each hotspot entry, the agent calculates the ratio of its minimal execution time and of its maximum execution time to its average execution time. If one or both of the ratios exceeds a preset threshold, the agent marks the entry as a bottleneck.

### 4.4. Cause Analysis Phase

If tracing data is available, the agent next enters the cause analysis phase, which assigns names to bottlenecks and finds local and remote events that caused them. The agent carries out several activities using local tracing data in a two pass scheme. In the first trace-log pass, the agent identifies trace events with source location matching any of the profiling entries discovered in the detection phase[3]. For each matching trace event, the agent generates a request entry containing its name, start time, and end time along with the event's operation type (supplied by the event-type mapper in PPW) which is sent to other agents to retrieve appropriate trace data (Table I).

[3] Agents can choose to apply the filtering techniques on each trace event during this pass to further reduce the amount of data exchange needed between agents.

For example, for a upc_lock event on node 0 with start time of 2 ms and end time of 5 ms, the request entry {source node 0, 2ms, 5 ms, P2P unlock} would be issued to all agents. The logic behind this is the following: if at the time of the lock request, another node holds the lock, the P2P lock operation issued by node 0 will block until it is released by the lock holder. To find out which node(s) held the lock that caused the delay in P2P lock operation, we simply look at the P2P unlock operations issued between the start and end time of the lock operation. If no P2P unlocked operation was issued by any other nodes, we conclude that the delay was caused by uncontrollable factors that cannot be resolved by the user, such as network congestion due to concurrent execution of other applications. At the end of the first pass, the agent sends the requests out to all other agents and waits for the arrival of requests from all other agents.

Next, the agent makes a second pass through its trace log and generates the correct replies – consisting of {event name, timestamp} tuples – and sends them back to the requesting agents. Finally, the agent waits for the arrival of replies and completes the cause analysis by assigning a bottleneck pattern name to each matching trace event, along with the remote operations that contributed to the delay.

*Table I – Generic-operation-type specific bottleneck pattern, request targets, and remote data type needed for cause analysis*

| ***Local data type*** | ***Bottleneck patterns*** | ***Request targets*** | ***Remote data type*** |
|---|---|---|---|
| Global sync. / comm. | Wait on group sync. / comm. (load-imbalance) | All other | Global sync. / comm. |
| P2P lock | Wait on lock availability | All other | P2P unlock |
| P2P wait on value | Wait on value change | All other | One-sided data xfer |
| One-sided put / get | Competing put / get | All other | One-sided Put / get |
| Two-sided send | Late sender | Receiver node | Two-sided receive |
| Two-sided receive | Late receiver | Sender node | Two-sided send |

### 4.5. Resolution Phase

In the final step of the analysis process, the resolution phase, the agent aims at identifying hints useful to the user in removing the bottlenecks identified in the other phases. This process is the only part of the system that may need to be model-dependent, as a given resolution strategy may not always work for all programming models. For example, a technique to fix the performance degradation stemming from upc_memget, versus from shmem_get, could be different even though they are both classified as one-sided get operations.

One example of this is to automatically identify the best blocking factor to use in the definition of a UPC shared array. When the system detects an excessive communication issue associated with a shared array, the agent would try to find an alternative blocking factor that would yield the maximum local-to-remote memory access ratio for all nodes in the system.

## 5. System Evaluation and Results

We have implemented an initial sequential version of our analysis system (supporting UPC, SHMEM, and MPI) which has been integrated into the latest version of the PPW tool. The system adds to PPW a number of analysis components, corresponding to those shown in Figure 2, to perform the necessary processing, management of analysis data, and presentation of analysis results to the tool user. To perform any of the analyses, the user brings up the analysis user interface (Figure 5), selects the desired analysis type, and adjusts any parameter values (such as *percentage program threshold* that defines the minimum hotspot percentage) if desired.

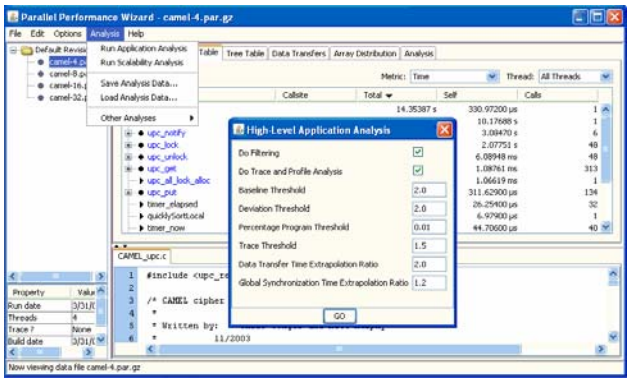


*Figure 5 – PPW analysis user interface*

The initial version of the system then employs a single agent to complete the selected analyses in the order depicted in Figure 6. To acquire the appropriate baseline values needed for the baseline filtering technique, we created a set of bottleneck-free benchmark programs for each of the supported models. These benchmarks are then executed on the target system, and the generated data files are processed to extract the baseline value for each model construct. Once all the analyses are completed, the results are sent to an analysis visualization manager which generates the appropriate visualizations.

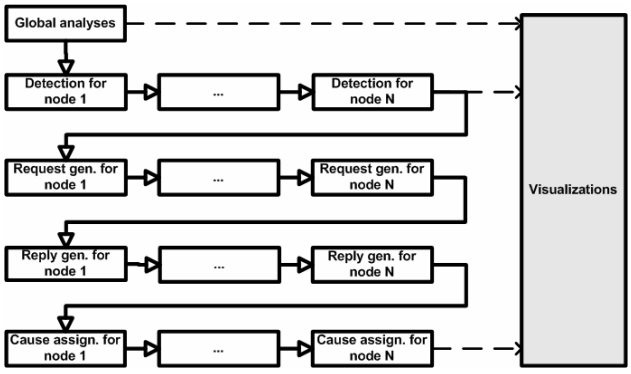


*Figure 6 – Sequential implementation workflow*

Our extended PPW tool provides several analysis visualizations, including the following: a scalability-analysis visualization that plots the scaling factor values against the ideal scaling value (Figure 7), a revision-comparison visualization that facilitates side-by-side comparison of observed execution times for regions

within separate versions of an application (Figure 8), a high-level analysis visualization displaying the breakdown of computation, communication, and synchronization time for each node executing an application (Figure 9), and a multi-table analysis visualization which lists all the results from the detection and cause-analysis phases along with source code correlation (Figure 9 left).

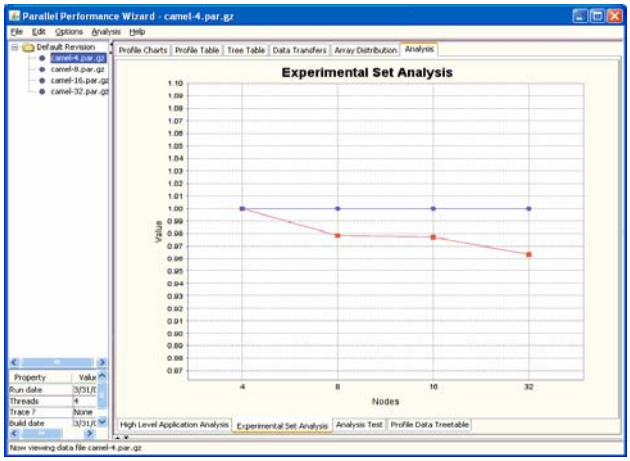


*Figure 7 – PPW scalability-analysis visualization*
*(blue line = expected scaling, red line =observed scaling)*

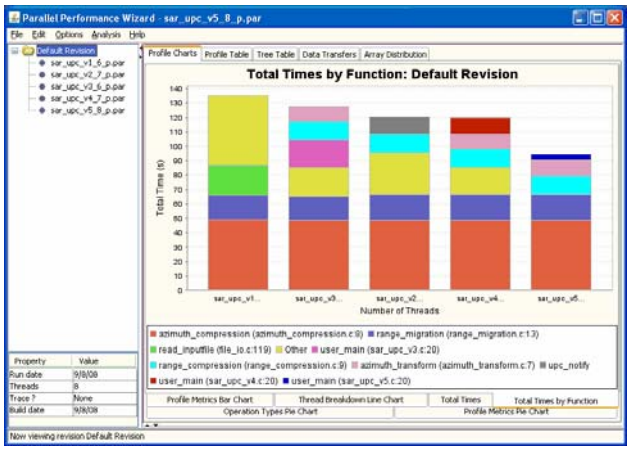


*Figure 8 – PPW revision-comparison visualization*

### 5.1. Correctness and Performance Testing

We created a set of test programs written in UPC, SHMEM, and MPI to verify the correctness of the sequential version of our analysis system. This analysis test suite consists of control programs with no bottlenecks and test programs which each contain a bottleneck pattern listed in Table I. We applied the analysis process on these programs and verified that the system is able to detect bottlenecks correctly.

To measure the speed of analysis, we applied the analysis process on the FT, CG, and IS NAS 2.4 benchmarks on a Pentium 4, 2.8 GHz processor workstation with 3 GB of RAM. The results are given in Table II, showing the analysis speed to be linearly proportional to the number of events in the raw data file (on average, the system processes 4-5 million trace events per minute).

These initial performance results are encouraging, especially since future, parallel implementations of the system should execute substantially faster on multi-core workstations or other parallel systems.

*Table II – Analysis speed for NPB 2.4 FT, CG, and IS benchmarks*

| | ***FT*** | ***CG*** | ***CG*** | ***IS*** |
|---|---|---|---|---|
| ***System size*** | *32* | *4* | *8* | *16* |
| ***Raw data file size (GB)*** | *0.14* | *0.57* | *1.14* | *4.25* |
| ***# of trace events (million)*** | *8* | *34* | *68* | *268* |
| ***Execution time (min.)*** | *1.5* | *7* | *14* | *65* |

### 5.2. Fourier Transform Case Study

For this case study, we performed automatic analysis on the Fourier Transform (FT) benchmark (which implements a Fast Fourier Transform algorithm) from the NAS benchmark suite version 2.4. The performance data used in the analysis was collected on an Opteron cluster with a Quadrics QsNet$^{II}$ high-speed interconnect executed with 16 nodes using GASP-enabled Berkeley UPC version 2.6.

In an earlier experiment presented in [1], we manually analyzed the FT benchmark using the visualizations provided by PPW and identified a significant bottleneck associated with upc_barrier inside the main *fft* function. A more in-depth investigation led us to conclude that the cause of the bottleneck is the serialization of multiple unrelated upc_memget operations called immediately before the barrier. We replaced this blocking upc_memget with a non-blocking bupc_memget_async and were able to improve the performance of the program by 14%.

In this experiment, we applied the automatic analysis process to the same FT data file to check whether or not the system could find the bottlenecks that we identified earlier in [1]. Looking at the multi-table analysis visualization, we saw that the system found 4 bottlenecks, including the most significant upc_barrier bottleneck. In addition, the system was able to determine the cause of delay for each occurrence of the barrier operation that took longer than expected. For example, the system found that the barrier called by thread 7 with a starting time of 2.6s took longer than expected to execute because threads 8 and 15 entered the barrier later than thread 7 (Figure 9, left). As we observed in the annotated Jumpshot view (Figure 9, right), this was indeed the case. Switching to the high-level analysis visualization, we saw that each node spent 5-15 % of the total execution time inside the barrier call, further validating the existence of a barrier-related bottleneck. This percentage drops to 1-2 % of the total execution time for the revised version using the non-blocking get operation.

In this brief case study, we were able to show that our automatic analysis system was able to correctly identify and determine the cause of significant bottlenecks in the FT benchmark within a short period of time (less than 5 minutes). The results match the findings from our earlier manual analysis effort, which took us a few hours to identify using the PPW tool (along with our own expertise).

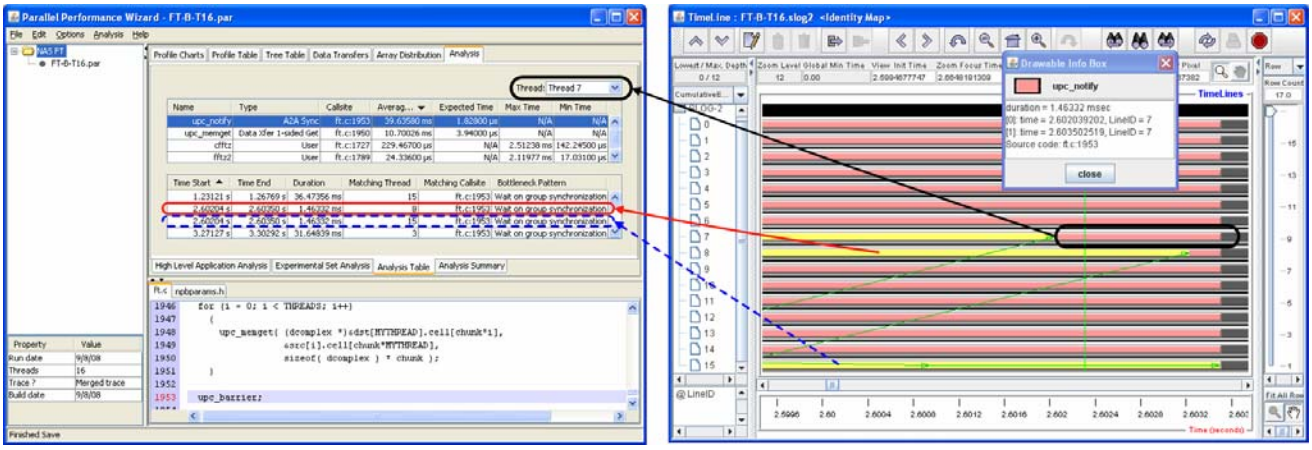

*Figure 9 – Multi-table analysis visualization for FT benchmark showing bottlenecks (top), causes (middle), and location (bottom) with annotated Jumpshot visualization that verifies the cause analysis results*

### 5.3. Synthetic Aperture Radar Case Study

For a second case study, we performed automatic analysis on an in-house SHMEM implementation of the Synthetic Aperture Radar (SAR) algorithm. The performance data for the analysis was collected on an Opteron cluster with a Quadrics QsNet$^{II}$ high-speed interconnect executed with 12 processing nodes using GASP-enabled Quadrics SHMEM.

SAR is a high-resolution, broad-area imaging process algorithm used for reconnaissance, surveillance, targeting, navigation, and other operations requiring highly detailed, terrain-structural information. The raw image gathered from the downward-facing radar is first divided into patches with overlapping boundaries so they can be processed independently from each other. Each patch undergoes a two-dimensional, space-variant filtering that can be decomposed into two domains of processing, the range and azimuth, to produce the result for a segment of the final image.

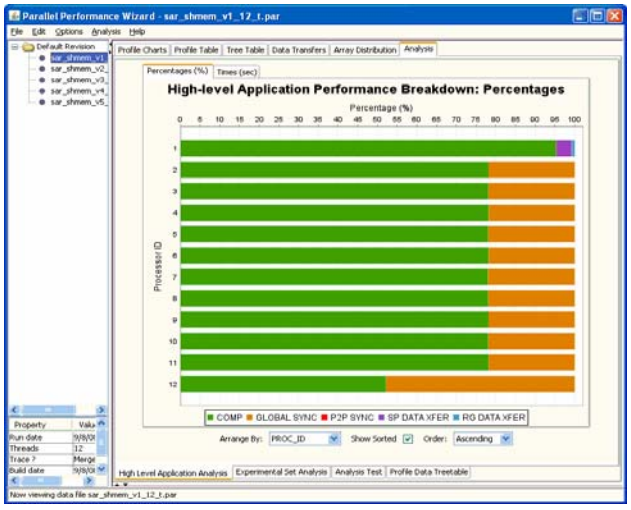


*Figure 10 – High-level analysis visualization for the original version (v1) of SAR application with load-imbalance issue*

Based on the sequential version of SAR from the Scripps Institution of Oceanography and MPI versions provided by two fellow researchers in our lab [16], we developed an initial SHMEM version of this application (denoted *v1*, it uses a single master node to handle all the I/O operations and also perform processing of patches). We collected performance data for version *v1* using PPW and applied the automatic analysis process. From the high-level analysis visualization (Figure 10), we observed that a significant amount of time is lost doing global synchronization. This is reconfirmed by looking at the multi-table analysis visualization which lists two shmem_barrier_all bottlenecks.

We came up with two optimization strategies to improve the performance. The first strategy was the use of a dedicated master node (performing no patch processing) to ensure that I/O operations could complete as soon as possible. The second strategy was to replace the all-to-all barrier synchronization with point-to-point flag synchronization (which implements a *wait-until-value-changes* operation) so processing nodes could work on the patches as early as possible. We systematically applied one or both of these strategies to the application and after each revision, the program was executed and analyzed again. We were finally able to successfully remove all bottlenecks from the application in version 5, which uses 2 master nodes and the flag synchronization together (Figure 11; note that the analysis system did not find any bottlenecks). This was verified when we looked at the high-level analysis visualization and saw that all nodes spent the majority of their time performing computation.

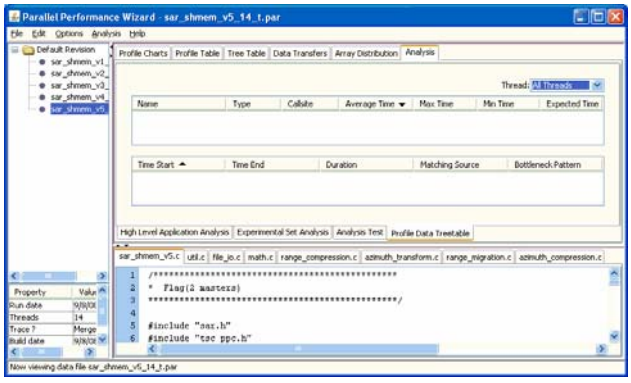


*Figure 11 – Multi-table analysis visualization for the optimized version of SAR application without any bottlenecks*

In this second case study, we have demonstrated how we used the automatic analysis system in the optimization of a SHMEM SAR application. The analysis system was able to detect significant bottlenecks in earlier program versions and, for the last version, provide results to verify the lack of bottlenecks. With the help of the analysis system, we were able to produce an optimized version that is 18% faster than the first version.

## 6. Conclusions and Future Directions

Parallel applications have the potential to achieve very high performance – but this potential is often not realized. In pursuit of better performance, programmers often use special-purpose tools that collect and visualize performance data to aid them in understanding and hopefully improving the performance of their applications. However, the sheer amount of data may make it impossible for the user to diagnose performance programs in such a way. To combat this problem and further facilitate the optimization process, a number of tools have added the capability to automatically identify performance issues within an application.

We originally created the Parallel Performance Wizard (PPW) framework and tool to facilitate performance tool support of parallel programming models. Earlier versions of our tool provided features to collect and visualize performance data for applications written in UPC, SHMEM, and MPI. In this paper, we presented a substantial enhancement to the PPW tool: a novel automatic analysis system to detect, diagnose, and resolve bottlenecks. We presented the architecture of this model-independent system and discussed how the system carries out all the analyses in a distributed fashion. We then showed correctness and performance results for a sequential implementation of the system that has been integrated into our PPW tool and demonstrated its effectiveness through two case studies.

Future work for this system includes development and evaluation of a parallel implementation of the system, enhancements to the existing analyses, support for additional analyses such as frequency analysis and bottleneck resolution, expansion of the number of common bottleneck patterns the system detects, and development of functionality to allow users to define new bottlenecks themselves.

## Acknowledgements

This work was supported in part by the U.S. Department of Defense. We would like to acknowledge members of the UPC group at UF, Armando Santos and Balaji Subramanian, for their involvement in the development of this system.

## References

[1] H. Su, M. Billingsley, and A. George, "Parallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming," 9th IEEE International Workshop on Parallel & Distributed Scientific and Engineering Computing (PDSEC) of IPDPS 2008, Miami, FL, Apr. 14-15, 2008.

[2] 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," Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06), Umeå, Sweden, June 18-21, 2006.

[3] Parallel Performance Wizard tool project website, http://ppw.hcs.ufl.edu/

[4] T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J.L. Traff, "Knowledge Specification for Automatic Performance Analysis", APART Technical Report Revised Version, August 2001.

[5] Automatic Performance Analysis: Real Tools (APART) IST Working Group website, http://www.fz-juelich.de/apart/

[6] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko, "Scalable Analysis of SPMD Codes Using Expectations," ICS 07, Seattle, Washington, June 16-20, 2007.

[7] K.A. Huck and A.D. Malony, "PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing," Supercomputing 05, Seattle, Washington, Nov. 12-18, 2005.

[8] K. Furlinger and M. Gerndt, "Automated Performance Analysis using ASL Performance Properties," Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06), Umeå, Sweden, June 18-21, 2006.

[9] J. Jorba, T. Margalef, and E. Luque, "Search of Performance Inefficiencies in Message Passing Applications with KappaPI-2 Tool," Lecture Notes in Computer Science, number 4699, Pages 409-419, 2007.

[10] M. Geimer, F. Wolf, B.J.N. Wylie, and B. Mohr, "Scalable Parallel Trace-Based Performance Analysis," Pages 303-312, PVM/MPI 2006, LNCS 4192, 2006.

[11] L. Li and A.D. Malony, "Model-Based Performance Diagnosis of Master-Worker Parallel Computations," Lecture Notes in Computer Science, Number 4128, Pages 35-46, 2006.

[12] J.K. Hollingsworth, "Finding Bottlenecks in Large Scale Parallel Programs," Ph.D. Dissertation, University of Wisconsin-Madison, 1994.

[13] I. Dooley, C. Mei, and L. Kale, "NOISEMINER: An Algorithm for Scalable Automatic Computational Noise and Software Interference Detection," 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Miami, Florida, April 14-18, 2008.

[14] J.S. Vetter and P.H. Worley, "Asserting Performance Expectations," Conference on High-Performance Networking and Computing, Baltimore, Maryland, 2002.

[15] I. Chung, G. Cong, and D. Klepacki, "A Framework for Automated Performance Bottleneck Detection," 13th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Miami, Florida, April 14-18, 2008.

[16] A. Jacobs, G. Cieslewski, C. Reardon, and A. George, "Multiparadigm Computing for Space-Based Synthetic Aperture Radar," Proc. of 2008 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, NV, July 14-17, 2008, to appear.