LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-227812

Practical Differential Profiling

M. Schulz, B. R. De Supinski

February 6, 2007

EuroPar 2007
Rennes, France
August 28, 2007 through August 31, 2007

Practical Differential Profiling

Martin Schulz and Bronis R. de Supinski
{*schulzm,bronis*}*@llnl.gov*

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
PO Box 808, L-560, Livermore, CA 94551, USA

Abstract. Comparing performance profiles from two runs is an essential performance analysis step that users routinely perform. In this work we present *eGprof*, a tool that facilitates these comparisons through differential profiling inside *gprof*. We chose this approach, rather than designing a new tool, since *gprof* is one of the few performance analysis tools accepted and used by a large community of users.
eGprof allows users to "subtract" two performance profiles directly. It also includes callgraph visualization to highlight the differences in graphical form. Along with the design of this tool, we present several case studies that show how *eGprof* can be used to find and to study the differences of two application executions quickly and hence can aid the user in this most common step in performance analysis. We do this without requiring major changes on the side of the user, the most important factor in guaranteeing the adoption of our tool by code teams.

1 Motivation

Users can choose from a large variety of performance analysis tools to optimize their parallel applications, like TAU [1], Vampir [7], Paradyn [6], or Open|SpeedShop [9] to name just a few. However, it is our experience from working with code teams both at Lawrence Livermore National Laboratory (LLNL) and beyond that those tools are best suited to performance analysis experts. They provide rich functionality that naturally implies a somewhat steep learning curve. Since members of the code teams focus on adding functionality to their applications that increase the science they can achieve, they only have limited time to spend on performance optimization (often only 1 or 2 weeks of effort per year across the entire team). Thus, they encounter the learning curve with each use and choose to use relatively simple tools for that reason.

In this work we therefore pursue a different approach to improve performance tools for these occasional performance tool users. Rather than developing a new tool with more advanced functionality, we analyzed the usage patterns of those tools that are already accepted by the code teams and determined their main deficiencies with respect to their common usage pattern based on user feedback.

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-CONF-227812).

Based on this analysis, we are augmenting these tools to suit those usage patterns better while retaining their ease-of-use. Our new toolset can be deployed transparently to the user without requiring any workflow or instrumentation changes on their side since it merely provides additional functionality. It will therefore easily find acceptance by a large group of users.

In this work, we apply this strategy to profiling of parallel applications with *gprof*, a command line driven profiler that is installed on almost all systems. While it is limited in its scope, it has found a large acceptance among our code teams because of its simplicity, wide installation base, and existing support by almost any compiler. However, *gprof* (like most other profiling tools) does not provide any direct support for the most common analysis step used in profiling: the comparison of executions, e.g., before and after coding changes intended to improve performance. Instead the user is left with having to compare large text logs of profiles manually, which is both tedious and error-prone.

We therefore extend the *gprof* toolset to include differential profiling allowing the user to directly compare two execution profiles as well as callgraphs from two different application executions. In addition, we provide a graphical representation of both individual and differential callgraphs to visualize the often complex information encoded in *gprof*'s callgraph results.

Combined, this provides an easy way to study the impact of code optimizations and parameter changes, as well as code properties both within one rank and across ranks. We will demonstrate this using four case studies covering various scenarios for single and multi-node performance analysis. In all cases, our extensions concisely present the key differences between individual executions in a few lines without the need for long manual searches.

2 Related Work

Only few tools support differential or comparative performance analysis. One of the exceptions is Open|SpeedShop [9], a recently developed performance toolset for Linux clusters, which includes the ability to align and contrast results from multiple runs. Further, Karavanic has investigated difference operators for performance event maps in Paradyn [6] as part of her Ph.D. thesis [4].

Both PerfDMF [3] and PerfTrack [5] provide a base infrastructure capable of supporting differential performance analysis. They both deploy relational databases to store the results of performance analysis across multiple runs. This data can later be queried and then compared using external tools.

Most other tools, however, only have the ability to work with data gathered during a single run and leave the user with the task to manually contrast the individual results. Due to the complexity and size of performance data, in particular from large scale parallel applications, this often tedious task risks missing key aspects.

3 eGprof: Differential Profiling with gprof

Differential profiling is a useful tool in many scenarios. However, to make it practical and accepted by code teams, it is important to provide a tool that is familiar to users as well as easy to use. To achieve these goals we decided to integrate the concept of differential profiling into *gprof*, which is part of the *GNU binutils* package. Thus, it is available on almost any UNIX based system, familiar to users, and already in use by most code teams.

Our design follows two main guidelines: any extension must a) be optional and cannot alter the default behavior of the tool and b) follow the main philosophy of the original *gprof*. These guidelines ensure the tool can be transparently deployed and is easy to use, which ensures acceptance by existing *gprof* users.

eGprof, our extended *gprof*, provides two major new features that aid differential profiling: the ability to subtract two profiles from each other and thereby to show their differences; and the visualization of both single and differential callgraphs.

3.1 Profile Subtraction

To use the unmodified *gprof* users must instrument their code using the *-pg* switch. The resulting binary then produces a *gmon.out* file containing the execution profile. *gprof* reads this profile along with the symbol information contained in the binary itself and produces both an execution profile and a dynamic callgraph in a textual representation.

To enable differential profiling, we allow the user to "subtract" a second profile from a baseline. Both profiles are collected in the same way and specified as command line options[1]. In this case, we first load both profiles individually, create the performance histogram and the callgraph information as in the original *gprof*, and associate the performance data (both the "positive" data from the baseline profile and the "negative" data from the second one) with the corresponding symbols.

We then align the two histograms, subtract the respective timing information for each symbol, and then store the newly created histogram. To generate the common callgraph, we first start with the baseline or "positive" graph. For each edge in the "negative" graph that is already present in the baseline graph we subtract the number associated with this edge, i.e., the number of times this edge was traversed at runtime, from the value associated with the existing edge. If the edge does not already exist, we add it, but record the negative of the associated number for this edge. In both cases, histogram and callgraph, we must take special care of zero values: in contrast to the original *gprof* in which zero means no performance information and hence can be removed, a zero in *eGprof* means that the performance in the two profiles is equal, which is relevant information that must be retained.

[1] Note if only a baseline profile is specified, *eGprof* behaves exactly like the original *gprof* ensuring that this addition is fully transparent.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
50.37	5.51	5.51	31	0.18	0.18	f_d
29.16	8.70	3.19	1	3.19	8.52	f_c
10.51	9.85	1.15				main
9.96	10.94	1.09	1	1.09	1.27	f_b

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
35.28	3.08	3.08				main
23.37	5.12	2.04	22	0.09	0.09	f_d
21.88	7.03	1.91	1	1.91	3.76	f_c
19.47	8.73	1.70	1	1.70	1.89	f_b

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
47.60	3.47	3.47	9	0.39	0.39	f_d
-26.47	5.40	-1.93				main
17.56	6.68	1.28	0	-1.28	-4.76	f_c
-8.37	7.29	-0.61	0	0.61	0.62	f_b

% impact	+self seconds	-self seconds	diff seconds	+self calls	-self calls	diff calls	sym +-	name
47.60	5.51	2.04	3.47	31	22	9	XX	f_d
-26.47	1.15	3.08	-1.93	-	-	0	XX	main
17.56	3.19	1.91	1.28	1	1	0	XX	f_c
-8.37	1.09	1.70	-0.61	1	1	0	XX	f_b

Fig. 1. Histograms for two executions (top two); differential histograms in old (third) and new format (bottom).

This basic approach works as long as both input profiles originate from executions of the same binary and hence are associated with the same symbol information. In many usage scenarios, e.g., for the evaluation of code optimizations, however, each profile is generated from a separate executable. In such a case, we require the user to specify the second binary along with its profile. We then load the symbol tables from each binary, but keep them in separate data structures since each binary potentially has different code regions associated with the same symbol. Instead we sort and align the two symbol tables against each other, and then link matching symbols.

In both cases we print the resulting differential profile in the existing *gprof* format, as shown in Figure 1. The top two histograms show two profiles of a demo program with different parameters and the third histogram shows the generated differential analysis. It shows, for example, that 47.6% of all performance changes were caused by a runtime decrease in routine *f_d* of 3.47 seconds, while *f_b* and *main* ran slower in the second execution by 1.93 and 0.61 seconds respectively.

By maintaining the output format, users can continue to use any evaluation scripts that parse the *gprof* output or external visualization tools that postprocess the output (e.g., *xprofiler*). Since this format is, however, not easily readable we also provide the option to print the resulting analysis in a new, more detailed format that further eases the manual analysis of the results and also includes

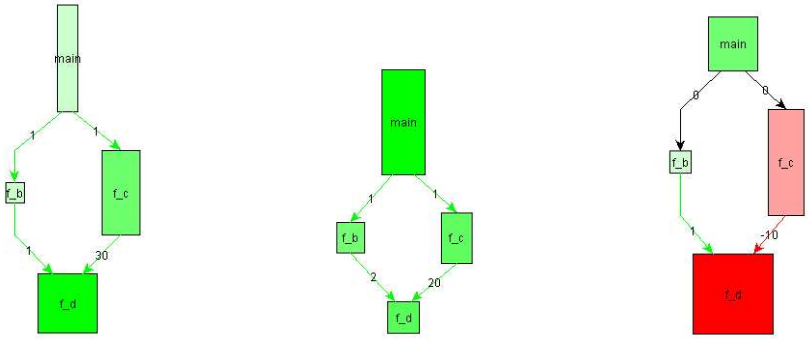

Fig. 2. Callgraphs of two executions (left, middle); differential callgraph (right).

the information from the separate profiles for easy comparison. The bottom histogram in Figure 1 shows the results of the above profile in the new format. For the remainder of the paper we will use this new format for illustration purposes.

3.2 Visualizing Callgraphs

gprof also provides dynamic callgraph information for all instrumented code pieces. However, this format is often difficult to read and adding differential information makes it even more complex. We therefore added support for callgraph visualization. On request, *eGprof* exports the callgraph as a GML (Graphic Markup Language) file. Figure 2 shows an example using the same data as above. The nodes represent the individual functions in the executable, and the edges show all calls between functions executed by the program together with the call frequency. Optionally, the size of the node can show both the scaled inclusive (height) and exclusive (width) time the code spent in this routine.

The left and the middle graph show the original execution profiles of the sample program and the right graph shows the differential callgraph. The latter is color coded to illustrate the direction of changes between the two original callgraphs. Again we see the large changes in *f_d* as well as in the exclusive time of *main*. Additionally, the graph shows an unchanged call frequency from *main* to *f_b* and *f_c*, while the second execution called *f_d* one more time from *f_b*, but ten times less from *f_c*.

By enabling *eGprof* to write GML files rather then visualizing the data itself, we ensure that the existing gprof maintains its command line philosophy and remains simple and easy to use. The created GML files can be visualized with any GML capable viewer. We use the freely available tool *yed* [2] for all graphs in this paper.

[2] http://www.yworks.com/en/products_yed_about.htm

4 Case Studies

Differential profiles can be used in many scenarios. In the following we present four case studies on parallel applications showing the use of eGprof for both single and multi-node performance analysis. All experiments were conducted on *mcr*, a large scale cluster installed at LLNL. Each node consists of two Intel Xeon CPUs running at 2.4 GHz and 4 GB of main memory. All nodes are connected using Quadrics's QsNet II (Elan3). The system runs CHAOS 3, a Linux distribution developed at LLNL. It is based on Red Hat's RHEL 4, but is optimized for high performance computing and provides a specialized MPI implementation for the Quadrics interconnect. All codes are compiled with gcc 3.4.4 and use *-O2* (unless otherwise noted) and *-pg* to activate the performance profiling in gcc. System libraries, including the MPI implementation, however, are unmodified and used as is, i.e., they are fully optimized and not compiled with *-pg*.

4.1 Single Node Performance

Performance Optimization: Performance optimization is usually an iterative process of selecting an appropriate optimization method, applying it to the code and then analyzing the performance of the newly generated code in comparison to the original one. The latter step is essential for understanding the impact of the chosen optimization and for selecting the one for the next step. This process is greatly aided by our differential profiling approach.

We show an example of this use on SMG2000, a Semicoarsining Multigrid Solver based on the hypre library [2]. We compile the base version of the application using the *-O2* flag and then optimize by allowing inlining using *-O2 -finline*. Figure 3 shows the resulting output for the top ten routines. While the performance only marginally improves, we clearly see that the 69536 invocations of *hypre_ExchangeLocalData* were inlined by the compiler eliminating all calls to this routine in the second version. The callgraph (omitted due to space constraints) shows that this routine was originally called by *hypre_InitializeCommunication* and the histogram correspondingly shows an increase in time spent in this routine due to the inlining. The top 2 routines,

% impact	+self sec.	-self sec.	diff sec.	+self calls	-self calls	diff calls	sym +-	name
21.65	55.25	54.83	0.42	6247	6247	0	XX	hypre_SMGResidual
7.73	33.68	33.53	0.15	4881	4881	0	XX	hypre_CyclicReduction
5.15	0.10	---	0.10	69536	-	69536	XX	hypre_ExchangeLocalData
5.15	2.89	2.79	0.10	1176	1176	0	XX	hypre_SemiInterp
5.15	0.82	0.72	0.10	980	980	0	XX	hypre_StructAxpy
4.12	0.65	0.57	0.08	1014	1014	0	XX	hypre_CycRedSetupCoarseOp
4.12	0.48	0.40	0.08	168	168	0	XX	hypre_StructVectorClearGhostValues
-4.12	0.66	0.74	-0.08	336	336	0	XX	hypre_StructVectorSetConstantValues
-3.61	0.05	0.12	-0.07	69536	69536	0	XX	hypre_InitializeCommunication
2.58	0.42	0.37	0.05	84	84	0	XX	hypre_SMGSetupInterpOp

Fig. 3. Comparing performance of SMG2000 with and without inlining.

% impact	+self sec.	-self sec.	diff sec.	+self calls	-self calls	diff calls	sym +-	name
60.72	536.56	---	536.56	-	-	0	XX	ATL_dupKBmm10_10_2_b1
-22.51	---	198.88	-198.88	-	-	0	XX	ATL_dJIK80x80x80TN80x80x0_a1_b1
-10.80	---	95.46	-95.46	-	-	0	XX	ATL_dJIK0x0x20TN20x20x0_a1_bX
1.60	14.12	---	14.12	-	-	0	XX	ATL_dJIK0x0x10TN10x10x0_a1_bX
0.60	20.15	14.86	5.29	500	50	450	XX	HPL_dlaswp00N
-0.43	---	3.79	-3.79	-	-	0	XX	ATL_dJIK0x0x20TN1x1x20_a1_bX
-0.40	---	3.57	-3.57	-	-	0	XX	ATL_dJIK0x0x50TN50x50x0_a1_bX
-0.34	4.51	7.48	-2.97	-	-	0	XX	ATL_dtrsmKLLNU
-0.33	---	2.89	-2.89	-	-	0	XX	ATL_dupMBmm0_1_0_b1_loopa
0.23	4.22	2.23	1.99	-	-	0	XX	ATL_dcol2blk_a1
-0.22	---	1.95	-1.95	-	-	0	XX	ATL_dJIK0x0x26TN26x26x0_a1_bX
-0.17	0.47	1.97	-1.50	-	-	0	XX	ATL_drow2blkT_a1
0.16	1.41	---	1.41	-	-	0	XX	ATL_dJIK0x0x10TN1x1x10_a1_bX
0.13	8.98	7.82	1.16	2	2	0	XX	HPL_pdmatgen
0.12	14.22	13.15	1.07	55042118	50541398	4500720	XX	HPL_lmul

Fig. 4. Comparing two executions of HPL with different blocksizes.

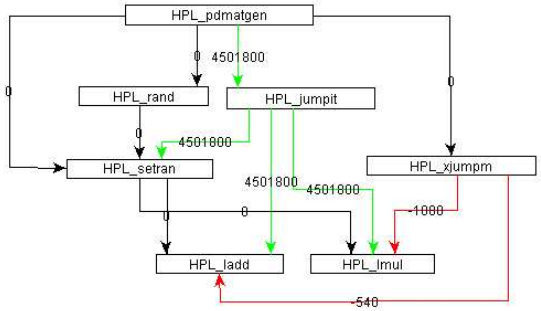

Fig. 5. Differential callgraph for HPL with varying block sizes.

however, benefit from inlining, most likely due to compiler optimizations enabled by the inlined routines.

Understanding Parameter Impact: Many applications have a large set of tunable algorithm parameters that enable the adjustment of codes to new platforms or problems. However, complex interactions among the parameters as well as with the target architecture often makes understanding their impact difficult. Differential profiling can help analyze the impact of parameter variations.

The High Performance LINPACK (HPL) [8] benchmark from the University of Tennessee uses a two-dimensional, block-cyclic data distribution and LU factorization with row partial pivoting featuring multiple look-ahead depths. Details of the algorithm can be fine-tuned with a large range of parameters. However, only rough guidelines exist on how to choose the best settings for a given target architecture, forcing the user to rely on hand-tuning for each platform.

In our experiment we use a constant problem size of N=5000 and vary one of the most significant parameters, the blocking size NB between 10 and 100, which causes more than a factor of two difference in performance. Figure 4

presents the differential profile for NB=10 and NB=100. It shows that HPL uses a drastically different set of routines in the underlying BLAS library for its computation depending on the value of NB[3]. This difference is further illustrated in the excerpt of the callgraph shown in Figure 5.

4.2 Cross-Node Performance

Understanding Load Balancing: The efficiency of parallel applications depends to a large degree on correct load balancing, i.e., ensuring that all tasks have roughly the same amount of work and hence do not incur long wait times at synchronization points. Comparing performance profiles of different ranks is one way to check for correct load balancing.

To illustrate this point we use Sweep3D from the ASCI Blue Benchmark suite. This code solves a 1-group time-independent discrete ordinates 3D cartesian geometry neutron transport problem. We run this code with 32 processes and create a separate profile for each rank. Using *eGprof* we then compare two representative tasks[4], in this case 0 and 1.

Figure 6 shows that the execution profiles of the individual ranks of Sweep3D are very similar indicating a well balanced code. All routines are called the same number of times (except for the timing routine, which is only called on the master node, rank 0) and the time spent in each routine is nearly identical.

Synchronous vs. Asynchronous Communication: Synchronous communication in parallel programs can lead to long blocking delays. In some cases this delay can be hidden using non-blocking communication calls: these routines allow send and receive operations to start without immediately waiting for their completion. Instead the call returns and the application can overlap computation with the message transfer.

We show the effects of non-blocking communication by contrasting the performance of a synchronous and an asynchronous version of SMG2000. For this

[3] The missing call number information is caused by the ATLAS library (all routines starting with *ATL_*) since this library was preinstalled and not compiled with *-pg*.
[4] All rank combinations exhibit similar behavior.

% impact	+self seconds	-self seconds	diff seconds	+self calls	-self calls	diff calls	sym +-	name
-98.69	975.14	981.94	-6.80	12	12	0	XX	sweep_
-0.58	2.20	2.24	-0.04	12	12	0	XX	flux_err_
-0.15	0.00	0.01	-0.01	1	1	0	XX	initgeom_
0.15	0.96	0.95	0.01	1	1	0	XX	initialize_
0.15	0.01	0.00	0.01	1	1	0	XX	inner_auto_
-0.15	0.00	0.01	-0.01	1440	1440	0	XX	snd_real_
0.15	18.06	18.05	0.01	12	12	0	XX	source_
0.00	---	---	0.00	2	-	2	XX	timers
0.00	0.00	0.00	0.00	1	1	0	XX	MAIN__

Fig. 6. Comparing rank 0 and 1 of Sweep3D.

% impact	+self seconds	-self seconds	diff seconds	+self calls	-self calls	diff calls	sym +-	name
-19.09	614.20	640.62	-26.42	263328	263328	0	XX	hypre_SMGResidual
7.70	15.90	5.24	10.66	-	-	0	XX	elan_tportBufFree_locked
5.27	12.11	4.82	7.29	-	-	0	XX	elan_tportGCBufPool
5.19	55.36	48.18	7.18	-	-	0	XX	elan_progressFragLists
4.94	331.77	324.93	6.84	14144	205728	-191584	XX	hypre_CyclicReduction
-4.54	44.23	50.51	-6.28	-	-	0	XX	MPIR_Unpack2
-4.39	---	6.08	-6.08	-	-	0	XX	PMPI_Irecv
3.87	5.35	---	5.35	-	-	0	XX	PMPI_Recv
3.37	70.38	65.72	4.66	-	-	0	XX	elan_pollWord
2.65	28.65	24.99	3.66	-	-	0	XX	elan_tportRxStart

Fig. 7. Comparing synchronous and asynchronous communication in SMG2000.

experiment we statically link the MPI library to the application (to give *eGprof* access to its symbols) and execute both versions using 32 processes. We then add the data from all ranks into a single profile for each run using *gprof*'s sum option and contrast the two global profiles.

The resulting differential profile in Figure 7 shows that the first version uses blocking receives (*MPI_Recv*), while the second version uses the non-blocking counterpart (*MPI_Irecv*). The performance difference between those two calls is, however, negligible since these routines merely start the receive. The actual operation is conducted inside Quadric's Elan library, for which the first version shows longer execution times due to the blocking operations. The main computation routine (*hypre_SMGResidual*), however, executes slower in the non-blocking version. This is caused by the concurrent message transfer in the Elan library. Overall, however, the non-blocking version performs better showing that the code benefits from this style of communication.

4.3 Discussion

The case studies discussed above show that differential profiling can help users to quickly identify the key difference between two application executions, typically by just looking at the top ten or fifteen routines in the profile. Without this feature, users must manually examine the entire profile for both runs and match up the symbols, which may be presented in a different order, to understand execution time differences completely. For SMG2000, a relatively simple benchmark, the *gprof* output has around 370 lines for the histogram and over 2700 for the callgraph. Realistic applications with millions of lines of code, as are typical in environments like national laboratories, generate significantly more output so manual comparison is intractable. Together with the callgraph visualization, our *eGprof* implementation automates this activity to support quick comparison of any two performance profiles.

5 Conclusions

Code teams demand easy solutions in familiar environments for their performance analyses. They usually do not have the time to learn complex and new

tools, which instead are normally used by a few selected users specializing in performance optimization. We therefore take the approach to analyze those simple tools that are accepted and in use by our code teams and add missing functionality that directly aids their typical workflow.

Following this approach, we designed and implemented *eGprof*, an extension of the popular and widely available *gprof* profiler. Our version adds support for differential profiling to enable a quick and efficient comparison of the performance observed during two executions, the most common step in performance analysis. Further we support callgraph visualization to give users a quick graphical overview of the performance of their codes and the differences between two executions. In our cases studies we showed how these new features can significantly aid both single and multi-node performance analysis. In all cases, the output of *eGprof* was able to point to the key difference between application executions within the top 15 routines of the resulting profile.

Our *eGprof* implementation gives code teams access to powerful performance analysis techniques in a familiar environment. It does not require any new setup or performance measuring techniques on their side and therefore guarantees the lowest possible learning curve and easy acceptance.

References

1. R. Bell, A. Malony, and S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 17–26, Aug. 2003.
2. R. Falgout and U. Yang. hypre: a Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, pages 632–641, Apr. 2002.
3. K. Huck, A. Malony, R. Bell, and A. Morris. Design and Implementation of a Parallel Performance Data Management Framework. In *Proceedings of the 2005 International Conference on Parallel Processing*, Aug. 2005.
4. K. Karavanic. *Experiment Management Support for Parallel Performance Tuning*. PhD thesis, Department of Computer Science, University of Wisconsin, 1999.
5. K. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh. Integrating Database Technology with Comparison-Based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool. In *Proceedings of IEEE/ACM Supercomputing '05*, Nov. 2001.
6. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, Nov. 1995.
7. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.
8. A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack be nchmark for distributed-memory computers. Available at http://www.netlib.org/benchmark/hpl/.
9. The *Open|SpeedShop* Team. *Open|SpeedShop* for Linux. `http://www.openspeedshop.org/`, Nov. 2006.