Detecting Application Load Imbalance on High End Massively Parallel Systems

Luiz DeRose, Bill Homer, and Dean Johnson

Cray Inc.
Mendota Heights, MN, USA
{ldr,homer,dtj}@cray.com

Abstract. Scientific applications should be well balanced in order to achieve high scalability on current and future high end massively parallel systems. However, the identification of sources of load imbalance in such applications is not a trivial exercise, and the current state of the art in performance analysis tools do not provide an efficient mechanism to help users to identify the main areas of load imbalance in an application. In this paper we discuss a new set of metrics that we defined to identify and measure application load imbalance. We then describe the extensions that were made to the Cray performance measurement and analysis infrastructure to detect application load imbalance and present to the user in an insightful way.

1 Introduction

The current trend in high performance computing is to have systems with very large number of processors. In the latest list of Top 500 Supercomputing Sites [1], the smallest system in the top 10 has more than 9,000 processing elements, and the top 3 systems have more than 25,000 processors each. Moreover, with the recent shift in the microprocessor industry, which stopped riding the frequency curve and started increasing the number of processor cores in a chip, we will see a faster growth in the number of processing elements in these high end massively parallel systems. With the arrival of the "many-core" processors, we are going to see several massively parallel systems with tens and hundreds of thousands of processing elements in the near future. However, in order to perform at scale on these massively parallel systems, applications will have to be very well balanced. Thus, users will need performance analysis tools that can identify and display sources of load imbalance in an intuitive way.

A variety of performance measurement, analysis, and visualization tools have been created to help programmers tune and optimize their applications. These tools range from simple source code profilers [2], to sophisticated tracers and binary analysis tools. However, HPC performance tools currently tend to focus on processor performance, which is normally measured with hardware performance counters (e.g., Perfctr [3], PAPI [4], SvPablo [5], HPM Toolkit [6], HPCView [7]); analysis of communication (e.g., Vampir [8], VampirGuideView [9], Paraver [10], Jumpshot [11]); analysis of the memory subsystem (e.g., Sigma [12]), performance prediction (e.g., dimemas [13], Metasim [14]), or a combination of the above (e.g., TAU [15], KOJAK [16], Paradyn [17]). However, in general these performance tools do not focus on detection of load imbalance.

In order to address this problem, we extended the Cray performance measurement and analysis infrastructure [18], which consists of the CrayPat Performance Collector and the Cray Apprentice^2 Performance Analyzer, to automatically identify sources of performance imbalance, and present to the user in an insightful way. The main innovations presented in this paper include the definition of new metrics for evaluation of load imbalance in an application, and new insightful approaches for presenting load balance information in both textual and graphical forms.

The remainder of this paper is organized as follows: In Section 2 we briefly describe the Cray performance measurement and analysis infrastructure. In Section 3 we discuss our load balance metrics and demonstrate their use in textual form, using as an example the ASCI Sweep3d benchmark [19]. In Section 4 we discuss approaches for visualization of load imbalance, also using as example the Sweep3d benchmark. Finally, we present our conclusions in Section 5.

2 The Cray Performance Measurement and Analysis Infrastructure

The Cray performance measurement and analysis infrastructure consists of the CrayPat Performance Collector and the Cray Apprentice^2 Performance Analyzer. CrayPat provides an infrastructure for automatic program instrumentation at the binary level with function granularity. Users can select the functions to be instrumented by name or by groups, such as MPI, I/O, memory. CrayPat also provides an API for fine grain instrumentation. When instrumenting at a function level, users do not need to modify the source code, the makefile, or even recompile the program. CrayPat uses binary rewrite techniques at the object level to create an instrumented application, which is generated with a single static re-link, managed by CrayPat. When using the CrayPat API to instrument code regions source code modification and recompilation are needed, but other than the instrumentation differences, CrayPat and Cray Apprentice^2 treat code regions as user functions. Thus, for simplicity, in this paper we will refer to any instrumented section of the code (code regions, user functions, MPI functions, etc) as functions.

The second main component of the CrayPat Performance Collector is its runtime performance data collection library, which can be activated by sampling or by interval timers. Performance data can be generated in the form of a profile or a trace file, and its selection is based on an environment variable.

A third main CrayPat component is the report generator *(pat_report)*, which is a utility that reads the performance file that was created by the runtime library and generates text reports, presented in the form of tables.

Finally, the Cray Apprentice^2 Performance Analyzer is a multi-platform, multi-function performance data visualization tool that takes as input the performance file generated by CrayPat and provides the familiar notebook-style tabbed user interface, displaying a variety of different data panels, depending on the type of performance experiment that was conducted with CrayPat and the data that was collected. Cray Apprentice^2 provides call-graph based profile information and timeline based trace visualization, supporting the traditional parallel processing and communication mecha-

nisms, such as MPI, OpenMP, and SHMEM, as well as performance visualization for I/O.

3 Load Imbalance Metrics

The first step in order to be able to report load imbalance in an insightful way is to define load imbalance metrics that are intuitive. Thus, we defined a couple of metrics: *"imbalance percentage"* and *"imbalance time"*. Assuming that n is the number of processing elements, we define the imbalance percentage of a parallel application[1] as:

$$\text{imbalance percentage} = \frac{\text{maximum time} - \text{average time}}{\text{maximum time}} \times \frac{n}{n-1} \qquad (1)$$

The goal of imbalance percentage is to provide an idea of the "badness" of the imbalance. Thus, to make it intuitive, we defined it to be in the range of 0 to 100, where a perfectly balanced code segment would have 0 imbalance percentage and a serial portion of a code segment on a parallel application (for example a serial I/O) would have imbalance percentage of 100. The imbalance percentage corresponds to the percentage of time that the rest of the team, excluding the slowest processing element, is not engaged in useful work on the given function. If they are idle, this is the "percentage of resources available for parallelism" that is wasted. In the worst case (as in the example above), one processing element does all the work, so that the others are 100% wasted.

Notice, however, that a section of code that has a high imbalance percentage should not necessarily be the main target of performance optimization. Following the example above, a serial I/O will always have imbalance percentage of 100, independently of the amount of time that it takes. If the fraction of time spent on that particular I/O operation is small, its impact on the overall performance of the application will not be significant. Thus, we also need a metric that is related to execution time, in order to identify regions of the program that should be considered for optimization. We opted to define a metric that would provide an estimation to the user of how much time in the overall program would be saved, if the corresponding section of code had a perfect balance. Thus we defined the imbalance time as:

$$\text{imbalance time} = \text{maximum time} - \text{average time} \qquad (2)$$

The imbalance time for a code section represents an upper bound on the *potential saving* that could result from perfectly balancing that particular code section. It is only an upper bound because it assumes that other processing elements are simply waiting without doing any useful work while the slowest member finishes.

Figure 1 shows a pat_report table displaying the profile output[2] with load imbalance information from a 48 processors execution of the Sweep3d benchmark, running 20 iterations with a $150 \times 150 \times 150$ grid, on a Cray XT3. As described in Section 2, CrayPat

[1] As defined, the imbalance percentage is only valid for parallel programs. Serial programs would have an imbalance percentage of 0.

[2] This is the summary version of the output, which only shows the lines where the percentage of time is at least 0.05% of the total time.

```
Table 1:  Profile by Function Group and Function

 Time % |      Time |Imb. Time |  Imb.  |  Calls |Group
        |           |          |Time %  |        | Function
        |           |          |        |        |  PE='HIDE'

 100.0% | 3.661935 |       -- |    -- | 675604 |Total
|------------------------------------------------------------
|  72.2% | 2.644437 |       -- |    -- | 245380 |USER
||-----------------------------------------------------------
||  97.1% | 2.568997 | 0.126365 |  4.8% |    576 |sweep_
||   1.8% | 0.047655 | 0.001252 |  2.6% |    576 |source_
||   0.3% | 0.008992 | 0.000212 |  2.4% |    576 |flux_err_
||   0.3% | 0.007960 | 0.000940 | 10.8% | 118080 |snd_real_
||   0.1% | 0.003182 | 0.001944 | 38.7% |     48 |MAIN_
||   0.1% | 0.002816 | 0.000591 | 17.7% | 118080 |rcv_real_
||   0.1% | 0.001637 | 0.039748 | 98.1% |     48 |inner_
||   0.1% | 0.001386 | 0.000055 |  3.9% |     48 |initialize_
||===========================================================
|  27.8% | 1.016999 |       -- |    -- | 238224 |MPI
||-----------------------------------------------------------
||  79.9% | 0.812923 | 0.232522 | 22.7% | 118080 |mpi_recv_
||  12.8% | 0.129678 | 0.125148 | 50.2% |   1536 |mpi_allreduce_
||   5.8% | 0.059340 | 0.010950 | 15.9% | 118080 |mpi_send_
||   0.8% | 0.007631 | 0.000221 |  2.9% |    192 |mpi_bcast_
||   0.7% | 0.007423 | 0.000347 |  4.6% |    144 |mpi_barrier_
|============================================================
```

Fig. 1. CrayPat profile with load balance by function group and function

provides functionality for automatic performance instrumentation at the function level, and provides an API for hand instrumentation at a smaller granularity. The imbalance metrics described above are automatically calculated at the level that the code was instrumented. Pat_report splits the profile by the different instrumentation groups (user functions, MPI, I/O, Memory, etc), which provides an idea of the balance of the various phases of the application (computation, communication, I/O, and memory allocation). In the example shown in Figure 1, only MPI and user functions were instrumented, and we observe that 72.2% of the total time was spent in users functions, while 27.8% of the time was spent in MPI functions. We also observe that the function that has the highest potential savings (“`Imb. Time`”) is the “`MPI_recv`”, which is consistent with the Sweep3d application, since it communicates using a wave front approach, which creates a communication imbalance, since the higher ranks tend to wait longer on receives. Although not shown in the example, CrayPat can also display the call path, as well as source code and line number information for each function.

Notice that in the profile shown in Figure 1, by default, “`Time`” is the average time per processing element. Thus, the imbalance time will be greater than the average time

if the maximum time is more than twice the average. So it would not be unusual to see imbalance times that are larger than the average time for some functions, as is the case of function "inner" in Figure 1, but typically these would not be near the top of the profile.

The point of the imbalance metric is to reveal cases in which the average time underestimates the "contribution" of a function to the elapsed time of the program. The functions with the best opportunities for reducing runtime by improving load balance will be among those for which the maximum time spent by a single processing element exceeds the average time over all processing elements by an amount that is a significant fraction of the program runtime. Our "imbalance time" is precisely the excess of maximum over average. Notice that in the statement above, we have to say "are among" instead of just "are" because of cases like the following:

```
void F1()
{
   if (rank%2) G1()
   else        G2();
}
```

Here G1() and G2() will be very unbalanced individually, but if F1() has balanced inclusive time, then this section of the code will probably not have a load balance problem. Currently, CrayPat only shows the imbalance metrics for exclusive times. We are in the process of extending it to also provide imbalance metrics for inclusive times.

An alternative approach to display load balance information, which is also provided by CrayPat, is shown in Figure 2. This pat_report table displays the maximum, median, and minimum values for each function and corresponding PEs[3]. If desired, one can display the complete PE distribution for each function, but for runs with a large processor count, such table would not be practical. Another option is available, where the report presents a distribution with the top three and the bottom three PEs, in addition to the median.

4 Visualization of Load Imbalance

As described in Section 2 users can visualize the performance data generated by CrayPat with the Cray Apprentice^2 Performance Analyzer. One of the multiple views provided by Cray Apprentice^2 is the call graph profile shown in Figure 3. The call graph profile has a similar approach to the call graph visualization described in [20], where the size of the boxes are relative to the execution time of the function. In our case, the height of a rectangle represents the exclusive time of the function, while the width represents its children time (i.e., the inclusive time minus the exclusive time, which is the same as the sum of the time of all its children). Thus, when searching for the code segments that take most of the execution time, users should look for large tall boxes in the call graph.

[3] In the interest of brevity, this table only shows lines where the percentage of time is at least 5.0% of the total time. This threshold can be selected by the user when running pat_report. The default is to show only functions that execute at least 0.05% of the time.

```
Table 2: Load Balance across PE's by Function
  This table shows only lines with Time% > 5.0.

 Time % |   Cum. |     Time |  Calls |Group
        | Time % |          |        | Function
        |        |          |        |  PE[mmm]

 100.0% | 100.0% | 3.661935 | 675604 |Total
|-----------------------------------------------
|  72.2% |  72.2% | 2.644437 | 245380 |USER
||----------------------------------------------
||  97.1% |  97.1% | 2.568997 |    576 |sweep_
|||---------------------------------------------
3||   2.2% |   2.2% | 2.695363 |     12 |pe.0
3||   2.1% |  52.7% | 2.590278 |     12 |pe.31
3||   2.0% | 100.0% | 2.487526 |     12 |pe.37
||==============================================
|  27.8% | 100.0% | 1.016999 | 238224 |MPI
||----------------------------------------------
||  79.9% |  79.9% | 0.812923 | 118080 |mpi_recv_
|||---------------------------------------------
3||   2.7% |   2.7% | 1.045446 |   1440 |pe.47
3||   2.0% |  56.9% | 0.786257 |   2880 |pe.16
3||   1.4% | 100.0% | 0.563250 |   1440 |pe.0
|||=============================================
||  12.8% |  92.7% | 0.129678 |   1536 |mpi_allreduce_
|||---------------------------------------------
3||   4.1% |   4.1% | 0.254826 |     32 |pe.0
3||   2.1% |  71.1% | 0.128529 |     32 |pe.21
3||   0.0% | 100.0% | 0.002649 |     32 |pe.47
|||=============================================
||   5.8% |  98.5% | 0.059340 | 118080 |mpi_send_
|||---------------------------------------------
3||   2.5% |   2.5% | 0.070290 |   2880 |pe.27
3||   2.0% |  58.6% | 0.056842 |   2160 |pe.6
3||   1.2% | 100.0% | 0.033176 |   1440 |pe.0
|===============================================
```

Fig. 2. CrayPat profile with Load Balance across PE's by Function

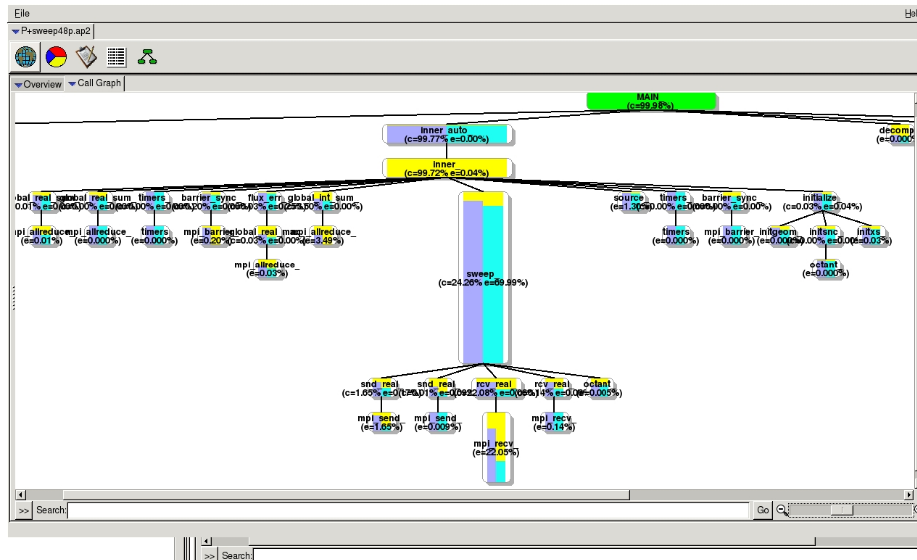

Fig. 3. Cray Apprentice[2] call graph view

As described above, when looking at the overall call graph, users have an idea of how each of the code segments contributes to total execution time of the application. We extended this call graph representation by adding a second level of abstraction, such that users could quickly observe load balance information when looking at a single rectangle. With this second level of abstraction, the height of each individual rectangle in the call graph represents the maximum execution time across all processing elements. The size of the darker blue bar in the left side of the rectangle is proportional to the average time, while the size of the light blue bar in the right side is proportional to the minimum time across all processing elements. The rest of the rectangle is filled with yellow. Thus, when looking at a single rectangle, users have an idea of the load balance of the code segment considering all processing elements. In order to identify functions that have a high percentage of load imbalance users should look for rectangles with a large yellow area. In particular, a large amount on yellow in the left side of a tall rectangle will indicate a high potential saving, since the amount of yellow in the left side of a rectangle represents the difference between the maximum time and the average time for the function, as defined in Equation 2. A rectangle almost completely filled with yellow (e.g., the function "`inner`" in Figure 3) normally indicates a function that is executing on a single PE.

The call graph view also provides a list of functions, which can be sorted by exclusive time, imbalance percentage, or imbalance time, as shown in Figure 4. Functions that have multiple call sites will appear multiple times in the list (with numbers added to the names for disambiguation). The sorted list helps users to quickly locate the main sources of load imbalance, which is helpful especially with large call graphs. When

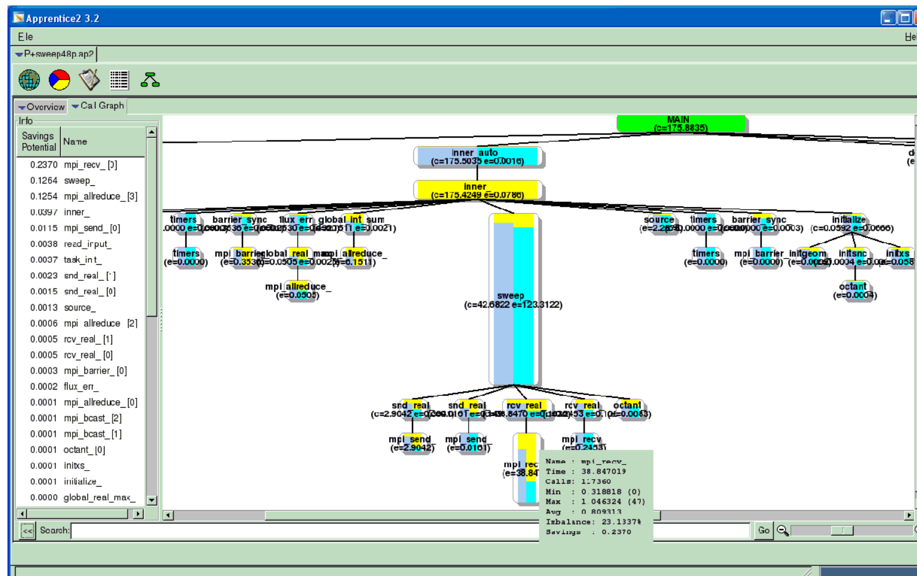

Fig. 4. load balance view

clicking on a function name the call graph pane will focus on the corresponding rectangle and highlight it. Also, when the user puts the cursor on top of a rectangle in the call graph, a popup window will appear, as shown for the function "`mpi_recv`" in Figure 4. The popup window displays the measured information for the function, including the corresponding PE that had the minimum and maximum times, as well as the average time, imbalance percentage, and imbalance time. Whenever collected during runtime, hardware counters data is also displayed in the popup window.

Cray Apprentice[2] also provides a view with the load balance distribution for any function in the call graph, as shown in Figure 5. The PEs in the load balance view can be sorted by time or number of calls. In addition, this load balance view display lines indicating the minimum, average, and maximum times for the function, as well as marks indicating the range of plus and minus one standard deviation from the mean, which can be used for a better understanding of the load balance distribution.

5 Conclusions

Applications will need to be well balance in order to achieve high scalability on current and future high end massively parallel systems. However, there is no standard for the measurement of load imbalance in an application and the current state of the art in performance tools does not focus on detection of load imbalance, which makes harder for users to tune applications on these massively parallel systems.

In this paper we presented the extensions to the Cray performance measurement and analysis infrastructure to support measurement, identification, and visualization of sources of performance imbalance in an application. The main contributions presented

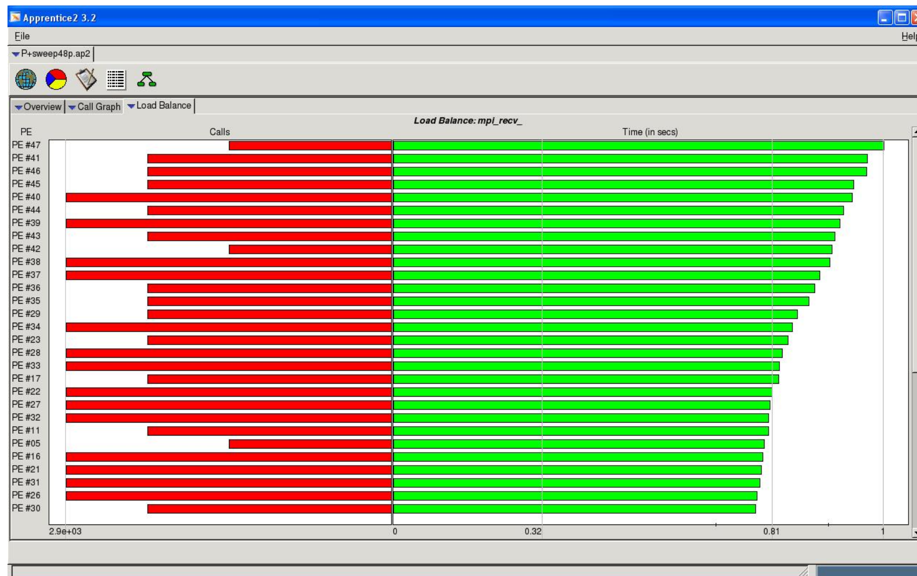

Fig. 5. load balance view

here were the definition of new metrics for evaluation of load imbalance in an application, and approaches for presentation in both textual and graphical form of load balance information that are insightful for the user. These approaches were exemplified with the ASCI Sweep3d benchmark.

In this paper we focus on the discussion of imbalance for time, which is calculated by default on CrayPat. However, CrayPat extends the concepts of *"imbalance percentage"* and *"imbalance (time)"* for any metric of interest. Using Figure 1 as an example, with the appropriate user options, pat_report could calculate these metrics for "CALLS", or for any other metric, including hardware counter metrics, such as FLOPS or cache misses.

References

1. Top500 Supercomputer Sites: The 28th TOP500 List. (2006) http://www.top500.org/.
2. Graham, S., Kessler, P., McKusick, M.: gprof: A Call Graph Execution Profiler. In: Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, MA, Association for Computing Machinery (June 1982) 120–126
3. Pettersson, M.: Linux X86 Performance-Monitoring Counters Driver. http://user.it.uu.se/~mikpe/linux/perfctr/. Computing Science Department; Uppsala University - Sweden. (2002)
4. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. The International Journal of High Performance Computing Applications **14**(3) (Fall 2000) 189–204
5. DeRose, L., Reed, D.: Svpablo: A Multi-Language Architecture-Independent Performance Analysis System. In: Proceedings of the International Conference on Parallel Processing. (August 1999) 311–318

6. DeRose, L.: The Hardware Performance Monitor Toolkit. In: Proceedings of Euro-Par 2001. (August 2001) 122–131
7. Mellor-Crummey, J., Fowler, R., Marin, G., Tallent, N.: HPCView: A tool for top-down analysis of node performance. The Journal of Supercomputing (23) (April 2002) 81–101
8. Nagel, W., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: Vampir: Visualization and Analysis of MPI Resources. Supercomputer **12** (January 1996) 69–80
9. Kim, S., Kuhn, B., Voss, M., Hoppe, H.C., Nagel, W.: VGV: Supporting Performance Analysis of Object-Oriented Mixed MPI/OpenMP Parallel Applications. In: Proceedings of the International Parallel and Distributed Processing Symposium. (April 2002)
10. European Center for Parallelism of Barcelona (CEPBA): Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual. (November 2000) http://www.cepba.upc.es/paraver.
11. Wu, C., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., Gropp, W.: From trace generation to visualization: A performance framework for distributed parallel systems. In: Proceedings of Supercomputing 2000. (November 2000)
12. DeRose, L., Ekanadham, K., Hollingsworth, J.K., Sbaraglia, S.: SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In: Proceedings of SC2002, Baltimore, Maryland (November 2002)
13. Labarta, J., Girona, S., Cortes, T.: Analyzing scheduling policies using Dimemas. Parallel Computing **23**(1–2) (1997) 23–34
14. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A.: A framework for performance modeling and prediction. In: Proceedings of SC2002, Baltimore, Maryland (November 2002)
15. Bell, R., Malony, A.D., Shende, S.: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In: Proceedings of Euro-Par 2003. (2003) 17–26
16. Wolf, F., Mohr, B.: Automatic performance analysis of hybrid mpi/openmp applications. Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing' **49**(10–11) (November 2003) 421–439
17. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tools. IEEE Computer **28**(11) (November 1995) 37–46
18. DeRose, L., Homer, B., Johnson, D., Kaufmann, S.: The New Generation of Cray Tools. In: Proceedings of Cray Users Group Meeting – CUG 2005. (May 2005)
19. Lawrence Livermode National Laboratory: the ASCI sweep3d Benchmark Code. (1995) http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/.
20. DeRose, L., Pantano, M., Aydt, R., Shaffer, E., Schaeffer, B., Whitmore, S., Reed, D.A.: An Approach to Immersive Performance Visualization of Parallel and Wide-Area Distributed Applications. In: Proceedings of 8th International Symposium on High Performance Distributed Computing - HPDC'99. (August 1999)