

# A Framework for Comparative Performance Analysis of MPI Applications

Edgar Gabriel,  
Department of Computer Science,  
University of Houston,  
4800 Calhoun Road, Houston, TX 77204, USA,  
gabriel@cs.uh.edu

Feng Sheng, Rainer Keller, Michael M. Resch  
High Performance Computing Center Stuttgart,  
University of Stuttgart,  
Nobelstr. 19, 70550 Stuttgart, Germany  
{fsheng, keller, resch}@hllrs.de

## Abstract

*Parallel application developers are facing a myriad of parameters when trying to understand the performance behavior of their code. Even within a single hardware configuration, the performance of any application will depend among others on factors such as the MPI library or some application level input parameters. This paper deals with the problem on how to determine the cause for performance variations of an application. Based on tracefiles of the application generated for several scenarios and an according documentation of the parameters used for each run, the PERDAC performance analysis tool is calculating statistical properties of the performance data gathered, such as average, standard deviation, maximum and minimum across the different runs. In the current implementation, the performance data comprises of the total execution time of an MPI function on a process, the number of occurrences of each MPI function and optionally some hardware performance counters such as cache hits or cache misses. In a second step, the results of the statistical analysis are traversed in search for parameters, which show a non-uniform behavior in the analyzed execution scenarios.*

## 1 Introduction

Parallel computing has reshaped science and industry over the last decade in many areas. The number of parallel systems available in the world is continuously growing. Thanks to standardized parallel programming paradigms such as MPI [12, 13] and OpenMP [2], writing parallel applications has become widespread. However, hurdles for application developers to write portable *and* performant code are still high. Even parallel computing experts struggle to understand the implication of any change in the software/hardware environment on the performance of the application. The key question arising is, how can an application developer determine, whether the reason for the sudden

performance degradation of his application is a modification in the source code, an upgrade of the MPI library done by his system administrator, or a slight modification in an input parameter.

The goal of the project presented in this paper is to provide a framework, which allows to determine reasons for performance variations of an application, by applying statistical analysis on performance data collected with the same application overall several runs with different parameters. Based on tracefiles of those runs, the analysis tool presented here calculates statistical properties of the performance data, and searches in a second step for values, which show non-uniform behavior across the analyzed scenarios.

Since the performance data available for the analysis is currently based on the tracing of MPI routines, the comparative performance analysis framework can only pinpoint to problems related to MPI functions. However, the methodology applied in this project can easily be extended to other performance data/parameters.

The paper is structured as follows: section 2 discusses the tools which we developed for comparative performance analysis. Section 3 presents an analysis of the High Performance Linpack (HPL) benchmark for a constant problem size, three different block sizes and three different MPI libraries using the tools presented in the previous section. In section 4 we compare our approach to related work in the area of performance analysis. Finally, section 5 summarizes this paper and presents the ongoing work within this project.

## 2 Description of the basic components

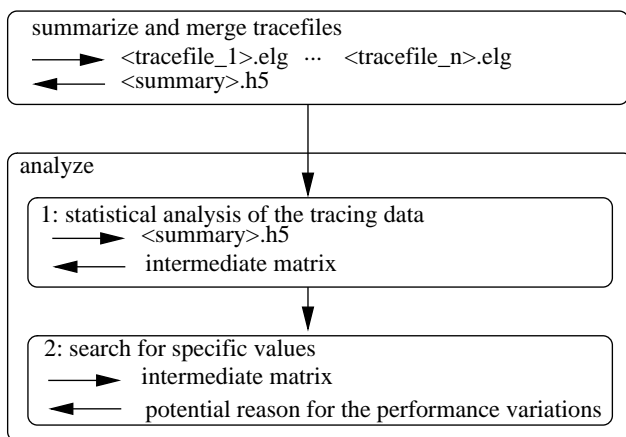
The overall goal of the PERDAC (Performance Data Comparator) project presented here is to create a framework for identifying key parameters/functions which influence the performance of an application. Three scenario's are especially relevant:

1. Identify the MPI functionality, which makes an application perform faster with one MPI library, compared

to a second/other MPI library

2. Identify the MPI functionality, which leads to load imbalance within an application (possibly again depending on the MPI library)
3. Identify the application parameter which influences the performance of an application

The PERDAC framework consists of the following steps: First, the user has to generate tracefiles of the applications with all parameters/variations which shall be used later on for the analysis. In the second step all tracefiles are summarized and merged into a single output file. Third, statistical properties of the performance data are being determined, and finally, the result of this statistical analysis will be traversed searching for 'special' values, which hint towards the reason for the performance variations. The sequence of action items when applying PERDAC for a performance analysis is also shown in Fig. 1. In the following, we would like to discuss each of these steps separately.



**Figure 1. Description of the workflow of the comparative performance analysis**

## 2.1 Generating Tracefiles

Generating tracefiles of MPI application are usually done based on the profiling interface of MPI [12]. There are several well established libraries available, such as vampirtrace [10], MPE [5], or EPILOG [18]. These libraries intercept every MPI function and store relevant data for each function call on a per process basis. As an example, for a send operation this data includes the time-stamp when the function has been entered, the rank/process id of the destination process, the amount of data sent to the

destination process, the communicator used, and the time-stamp when leaving the function. Furthermore, many tracing libraries support hardware performance counters such as cache misses or cache hits. A portable interface for accessing hardware performance is provided e.g. by the PAPI [3] library.

The information recorded by a tracing library can be used to visualize messages and the communication pattern of a parallel application. This can be done with tools such as Vampir [4], jumpshot [19] or paraver [11]. However, if the format of the tracefile is publicly available, one can apply arbitrary operations on tracefiles. In the frame of this project we decided to use the EPILOG tracing library, since its functionality includes a C++ interface for directly accessing the data structures within the tracefile.

## 2.2 Summarizing and Merging Tracefiles

Since the goal of this project is to identify parameters/MPI functions which are causing performance degradations in one scenario compared to another scenario, the user will inevitably has to produce multiple EPILOG tracefiles. Handling several tracefiles is however inconvenient, especially if the number goes beyond a certain threshold. Furthermore, we decided to compact the information given in a tracefile on a per MPI function basis. As an example, instead of using the record of every MPI\_Send function for the later analysis, we summarize all events including an MPI\_Send on a single process by storing the number of occurrences of this event, how much time has been spent all-in-all in this function on a single process and some additional, optional performance metrics provided by PAPI. Clearly this step removes some information from the original tracefile, since there are scenarios in which a single 'bad' function-call is causing the problem. Our approach is capturing however general trends and releases systematic performance problems of an MPI library, and reduces at the same time the data volume significantly.

Summarizing and merging the EPILOG tracefiles is done by a C++ code (which we will call for the sake of simplicity *summarize* for the rest of this paper) using the interface provided by the *Event Analysis and Recognition Library* (EARL) [17] tool. EARL allows arbitrary access to the events stored in an EPILOG tracefile. Amongst the information which can be extracted using EARL is the type of the event ('ENTER' or 'EXIT'), the time-stamp of the event, the location of the event presented as a tuple of (machine, node, processor, thread), the number of metrics available for this event and the list of available metrics and metrics names.

*summarize* generates an output-file, which can be stored either in a plain ASCII format or in HDF5 [9] format. HDF5 is a scientific data library developed at NCSA and used in

many scientific applications. The main feature of HDF5 used in this context is the ability to have different datasets within the same file and the possibility to attach names and attributes to each data point. Figure 2 shows the main components of a merged tracefile printed with `h5dump`.

The resulting file of the summarize and merge step contains thus several data sets, one for each EPILOG tracefile. A data set contains the name of the original tracefile, and a three dimensional data space: the extent of the first dimension is equal to the number of MPI functions used in this application, the second is referring to the number of MPI processes used, and the third to the number of metrics stored with each event. Along with the data space, the file also contains attribute fields, which document the names of the MPI functions used (e.g. `MPI_Send` or `MPI_Recv`), and the names of performance metrics (e.g. no. of occurrences, time spent in function or `L1_D_ACCESS`).

Technically, *summarize* can merge two non-related tracefiles; the result of this operation is very probably meaningless. It remains the responsibility of the user to ensure, that the tracefiles are from the very same application with exactly the same number of processes.

```
HDF5 "/home/gabriel/Software/PERDAC/all.h5" {
GROUP "/" {
  DATASET "ftmpi-240.elg" {
    DATATYPE H5T_IEEE_F64LE
    DATASPACE SIMPLE { ( 11, 4, 4 ) / ( 11, 4, 4 ) }
    DATA {
      (0,0,0): 1, 77942, 1.92572e+07, 1.39144,
      ...
    }
    ATTRIBUTE "function names" {
      DATATYPE H5T_STRING
      DATASPACE SIMPLE { ( 11 ) / ( 11 ) }
      DATA {
        (0): "MPI_Init", "MPI_Send", "MPI_Recv", "MPI_Comm_split",
        (4): "MPI_Type_struct", "MPI_Type_commit", "MPI_Sendrecv",
        (7): "MPI_Ssend", "MPI_Type_free", "MPI_Comm_free", "MPI_Finalize"
      }
    }
    ATTRIBUTE "metric names" {
      DATATYPE H5T_STRING
      DATASPACE SIMPLE { ( 4 ) / ( 4 ) }
      DATA {
        (0): "No of occ.", "L1_D_MISS", "L1_D_ACCESS", "Time in fct"
      }
    }
  }
  DATASET "lam7-240.elg" {
    ...
  }
}
```

**Figure 2.** An example of the HDF5 file generated by *summarize*

### 2.3 Statistical Analysis of the Merged Tracefile

The resulting file of *summarize* is the input to an *analyzer* application. The *analyzer* calculates in an initial pre-processing step derived values for each metric, e.g. by dividing the total time spent in an MPI function on a process by the number of occurrences of the same MPI function on the same process. Therefore, the application has additionally to the absolute values for each metric also the average

values available for the analysis.

Next, the *analyzer* converts all available performance data into a five dimensional matrix: the extent of the first dimension is the total number of MPI libraries used in this analysis, the extent of the second dimension is the number of MPI processes. The third dimension is reserved for variations of an application parameter, while the last two dimensions correspond to the number of MPI functions used in the application and the number of metrics stored with each event.

A key feature of the *analyzer* is its ability to calculate certain statistical values on the matrix in any matrix direction. As an example, if the user would like to compare the performance of an application with two different MPI libraries, as described in scenario 1, the analyzer would apply its statistical methods over the first dimension of the matrix. For the second scenario described above, the same analysis will be performed over the 2nd dimension of the matrix, while for scenario 3 the 3rd dimension of the matrix will be used. This approach greatly simplifies the source code and the maintainability of the *analyzer*.

The current set of statistical analysis which can be performed by the *analyzer* includes calculating the average and the standard deviation over a set of measurements, and determining for each set the minimum and maximum together with their location. As an example, for the 1st scenario described at the beginning of this section, this would lead to how long an MPI library spent on average for each process e.g. in `MPI_Send`, which MPI library spent the least and the most time in this MPI function on this process.

### 2.4 Identifying Special Values

The result of the analysis step is a new matrix. This temporary matrix is one order lower than previous one, contains however instead of a single value always a structure with the average, minimum, location of the minimum, maximum, location of the maximum, absolute and relative standard deviation. In the next step, the program tries to extract from this statistical data the values, which might be the reason for the performance variation.

The algorithm applied currently at this point is to report all values, whose relative standard deviation is larger than a certain threshold. The reasoning behind this approach is, that the standard deviation indicates whether the measured values are close to the average or not. If all values which are used for the comparison are close to the average, the probability that they point to the reason for the performance variation is low, while if the data points are widely scattered around the average, the average does not really point out a 'common behavior' in all tests.

Blocksize	FT-MPI	LAM/MPI	MPICH2
48	160.0	154.9	157.09
80	152.6	146.6	149.18
240	160.3	152.6	155.33

**Table 1. Execution time in seconds of HPL runs using three different blocksizes and three different MPI implementations.**

### 3 A Usage Scenario

In the following, we would like to present a usage scenario of the PERDAC comparative performance analysis framework. This test case uses the High Performance Linpack benchmark (HPL) [14] for a constant problem size (6000) on four processors. We ran tests with three MPI libraries, namely FT-MPI v0.9 [6], LAM/MPI v7.0 [16] and MPICH2 v0.96 [8]<sup>1</sup>. With each MPI library we executed three tests by varying the blocksize-parameter of HPL, using 48, 80, 240. The overall execution times with each MPI library and blocksize are summarized in table 1. All tests have been executed on the same platform, namely a Pentium III cluster connected by a Fast Ethernet network. The tracefiles captures additionally to the execution time for each function also the hardware counters for the first level cache, namely `L1_D_ACCESS` and `L1_D_MISS`.

Generally, MPICH2 has the lowest execution time in all test. Since we used for all tests the same compiler and the same hardware, the reason for the performance difference has to be directly related to the MPI libraries. To understand the reasons for the performance difference between the MPI libraries, we originally conducted a set of micro-benchmarks, such as simple ping-pong benchmarks or testing the derived datatypes used within HPL. However, these micro-benchmarks did not reveal any significant differences between the MPI libraries. The question which we would like to address in this analysis is therefore twofold:

1. Which MPI function used in HPL performs best with MPICH2 and might therefore be the reason for the better performance?
2. Which MPI function is affected mostly by the different blocksizes?

In our analysis, we used as the threshold for reporting special values (as described in subsection 2.4), a relative standard deviation larger than 10%. Figure 3 shows an example for the output file generated by the analyzer. The analyzer only displays values, whose relative standard deviation

<sup>1</sup>Please note, that newer releases are available for all three MPI libraries. The performance results might vary compared to the results presented here

is above the threshold defined by the user. The first column shows the name of the specific metric, which crossed the threshold, followed by the average value for this metric, the absolute and relative standard deviations. Next, the minimum and the location of the minimum are displayed. Since we compared different MPI libraries for this particular example, the location of the minimum is determined by *which MPI library* had the minimal value. The same display is used for the maximum and the location of the maximum. The four last columns indicate, which MPI routine on which process and for which application parameter (in this example blocksize NB) has been applied in the measurement producing this line of output.

#### 3.1 Comparing different MPI libraries

The MPI functions used within HPL can be distinguished into process management functions, such as `MPI_Init` or `MPI_Finalize`, functions involving explicit communication, e.g. `MPI_Send`, and management functions, which might involve indirectly communication, such as `MPI_Comm_split`. Since the process management functions used within HPL are only called once, they will be ignored for the rest of the analyses.

HPL uses all-in-all four MPI functions, which directly imply a communication between two processes. These functions are `MPI_Send`, `MPI_Recv`, `MPI_Ssend`, and `MPI_Sendrecv`. The analysis of these functions can be summarized as follows:

1. For the functions `MPI_Send`, `MPI_Recv`, and `MPI_Ssend` the tracefiles showed significant differences in the cache-behavior of the MPI libraries. Generally, LAM7 showed the best values for the level one cache hits and cache misses as well (`L1_D_ACCESS` and `L1_D_MISS`). MPICH2 had the highest counters for the `L1_D_MISS`, while FTMPI had the worst values for `L1_D_ACCESS`. The best and worst values for each of the counters differ by a factor between two and four. However, the level-one cache behavior does not hint for this (fairly slow) network to similar behavior of the execution time. In fact, the summary of the execution times for each function does not cross the specified threshold for reporting them as a potential problem, they all are within 10% of the average.
2. MPICH2 has the best overall execution time for `MPI_Sendrecv`, FT-MPI is showing the highest values for the summarized execution time.

In contrary to communication functions, the management functions show the tendency of a correlation between the execution time and the level one cache behavior, at least for those MPI functions, which do not imply communication, such as `MPI_Type_struct/commit/free`.

Function:rank	metric	average	stdev	re_sdev	min	minloc	max	maxloc	NB
MPI_Send: 0	Ll_D_MISS	8529.000	4852.3	56.9	3915.000	lam7	15235.000	mpich2	240
MPI_Send: 0	Ll_D_ACCESS	278860.667	223263.2	80.1	92339.000	lam7	592750.000	ftmpi	240
MPI_Send: 0	Ll_M_Pro_oc	167.235	95.1	56.9	76.765	lam7	298.725	mpich2	240
MPI_Send: 0	Ll_A_Pro_oc	5467.856	4377.7	80.1	1810.569	lam7	11622.549	ftmpi	240
MPI_Send: 1	Ll_D_MISS	8397.000	5380.4	64.1	3741.000	lam7	15937.000	mpich2	240
MPI_Send: 1	Ll_D_ACCESS	213838.000	159552.6	74.6	64205.000	lam7	434918.000	ftmpi	240
MPI_Send: 1	Ll_M_Pro_oc	254.455	163.0	64.1	113.364	lam7	482.939	mpich2	240
MPI_Send: 1	Ll_A_Pro_oc	6479.939	4834.9	74.6	1945.606	lam7	13179.333	ftmpi	240
MPI_Send: 2	Ll_D_MISS	8826.667	4514.9	51.2	4522.000	lam7	15063.000	mpich2	240
MPI_Send: 2	Ll_D_ACCESS	280489.333	227877.4	81.2	90787.000	lam7	600955.000	ftmpi	240
MPI_Send: 2	Ll_M_Pro_oc	180.136	92.1	51.2	92.286	lam7	307.408	mpich2	240
MPI_Send: 2	Ll_A_Pro_oc	5724.272	4650.6	81.2	1852.796	lam7	12264.388	ftmpi	240
MPI_Send: 3	Ll_D_MISS	7677.333	4305.6	56.1	3742.000	lam7	13669.000	mpich2	240
MPI_Send: 3	Ll_D_ACCESS	209378.333	163009.9	77.9	64192.000	lam7	437049.000	ftmpi	240
MPI_Send: 3	Ll_M_Pro_oc	232.646	130.5	56.1	113.394	lam7	414.212	mpich2	240
MPI_Send: 3	Ll_A_Pro_oc	6344.798	4939.7	77.9	1945.212	lam7	13243.909	ftmpi	240
MPI_Comm_split: 0	Ll_D_MISS	18965.333	17183.8	90.6	1091.000	mpich2	42161.000	lam7	240
MPI_Comm_split: 0	Ll_D_ACCESS	924954.667	1093523.2	118.2	23528.000	mpich2	2463906.000	lam7	240
MPI_Comm_split: 0	Time in fct	0.064	0.1	96.1	0.003	mpich2	0.149	lam7	240
MPI_Comm_split: 0	Ll_M_Pro_oc	6321.778	5727.9	90.6	363.667	mpich2	14053.667	lam7	240
MPI_Comm_split: 0	Ll_A_Pro_oc	308318.222	364507.7	118.2	7842.667	mpich2	821302.000	lam7	240
MPI_Comm_split: 0	Time_Pro_oc	0.021	0.0	96.1	0.001	mpich2	0.050	lam7	240
MPI_Type_struct: 0	Ll_D_MISS	9545.333	6334.4	66.4	3376.000	lam7	18255.000	ftmpi	240
MPI_Type_struct: 0	Ll_D_ACCESS	1428610.667	1382473.1	96.8	312860.000	mpich2	3376872.000	ftmpi	240
MPI_Type_struct: 0	Time in fct	0.005	0.0	75.3	0.002	mpich2	0.011	ftmpi	240
MPI_Type_struct: 0	Ll_M_Pro_oc	381.813	253.4	66.4	135.040	lam7	730.200	ftmpi	240
MPI_Type_struct: 0	Ll_A_Pro_oc	57144.427	55298.9	96.8	12514.400	mpich2	135074.880	ftmpi	240
MPI_Type_struct: 0	Time_Pro_oc	0.000	0.0	75.3	0.000	mpich2	0.000	ftmpi	240

Figure 3. Example for the output generated by the *analyzer*

MPICH2 showed for these functions the best values among the three MPI libraries. The overall execution time of these functions is however too small to have an overall effect on the performance of HPL. `MPI_Comm_split` typically will involve communication, since all processes have to agree on a common identifier. For this function, MPICH2 had once again the best cache behavior and the lowest execution time.

### 3.2 Sensitivity of MPI functions to the blocksize

In this analyses, we analyze the sensitivity of MPI functions to the blocksize parameter of HPL. The blocksize parameter influences in HPL not only the size of the matrix/data chunks calculated at once, but also the size and the number of messages communicated between the processes. As an example, for a constant problem size, a small blocksize will split the data, which has to be sent between different processes into smaller pieces, and will thus generate

more messages than a large blocksize, which will enforce sending fewer, but larger messages.

This analysis shows little variance for the different MPI libraries. As expected, the average execution time per function call for all four communication routines decrease for smaller block sizes. At the same time the level one cache behavior is reciprocal to the execution time per function call, the number of level one cache access and cache misses decreases with increasing blocksize. Similarly, the data type creation function `MPI_Type_struct` shows decreasing execution times per function call and increasing level one cache counters with smaller block sizes. However, the analysis did not reveal any significant differences for the overall execution times of the MPI functions, hinting therefore, that the difference in the overall execution time of each HPL run for different block sizes is **not** related to communication.

## 4 Related Work

The most profound application of non-trivial statistical analysis of a parallel application has been conducted up to now by Ahn and Vetter in [1]. Using cluster analysis and factor analysis, they managed to reduce the volume of the performance data by identifying correlated values and tasks-groups with similar behavior. In contrary to our work here, they did not apply their methods for comparing multiple experiments.

Comparative performance analysis of parallel applications is not a new research topic as well, many groups and approaches have been presented in the last years. As an example, in the frame of the KOJAK project [15] Wolf et al. developed an algebra for cross-experiment performance analysis. The main feature of this approach is the support for repetitive application of its methodology, since the format of the result of the algebraic operation is identical to the input format of the performance data. Currently, three different operators are supported: the merge operator integrates several tracefiles into a single one, the difference operator calculates, as the name suggests, the difference between two similar experiments and the mean operator calculates the average values over an arbitrary number of experiments.

The performance analysis tool Paraver [11], developed by the Barcelona Supercomputing Center can load multiple traces for analyzing them simultaneously. One basic functionality of Paraver is to copy and align the time range of the traces in separate displays. Similarly to our approach, this is only practical within traces of similar execution profile, which is not the case when executing the same application with different number of processes. Another way to properly cope with comparing two traces is to insert calls of the tracing API into the application and rerun to manually align two traces, or use the extensive functionality of Paraver to semantically analyze the trace-file. For collective communication, this would involve counting the MPI operations and align on the traces to compare onto the same collective operation (providing that two traces with different number of processes use the same communication pattern).

Aksum [7] is a multi-experiment performance analysis tool. Based on an initial performance properties defined by the user, the tool can automatically detect performance critical code regions. An experiment launcher modifies input parameters and evaluates the resulting performance data. The result of this analysis is used to suggest new experiments. Both of the latter two approaches do not incorporate however per se sophisticated statistical analysis of the performance data.

## 5 Summary

In this paper, we presented a framework for comparative performance analysis of MPI applications. The main goal of the framework is to be able to point out parameters/functions, which show a non-uniform behavior across several runs. The data used for the performance analysis is currently based on tracing of MPI routines. An analyzer calculates initially certain statistical properties of the gathered data, and searches the results of this analysis for parameters/values, which show significant differences over the various runs.

The current framework will be extended in the near future in several directions. First, the current interfaces only support the variation of a single application level parameter (additionally to the variance in the MPI library). A more general approach supporting  $n$  application level parameters is however inevitable for many real-world application scenarios. Second, the statistical methods applied in our analysis will be extended to include correlation analysis between the parameters as well as a more sophisticated sensitivity analysis. Sensitivity analysis can also deliver a measure, whether the sensitivity of a certain metric (e.g. execution time) to a certain parameter (e.g. blocksize in our example) can be determined at all based on the current performance data.

The algorithm searching for a special value will be extended such that it can incorporate the description of an 'expected behavior'. This is required for all application level parameters which are known to lead to differences in the execution time (e.g. the problem size), and where a special value might not a parameter/function which shows non-uniform behavior across the different executions, but a parameter which does not follow the expected behavior. Finally, the output of the analyzer needs to be improved with respect to the user-friendliness, including for example also a cluster-analysis mechanism to group the output according to certain events and reduce the initial data volume presented to the end-user.

## References

- [1] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the Supercomputing Conference*, 2002.
- [2] O. A. R. Board. *OpenMP Application Program Interface*. Version 2.5, May 2005.
- [3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.
- [4] H. Brunst, M. Winkler, W. E. Nagel, and H.-C. Hoppe. Performance Optimization for Large Scale Computing: The

- Scalable VAMPIR Approach. In V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, editors, *Computational Science – ICCS 2001, Part II*, number 2074 in LNCS, pages 751–760, San Francisco, CA, USA, may 2001. Springer.
- [5] A. Chan, W. Gropp, and E. Lusk. Scalable log files for parallel program trace data(draft), 2000.
- [6] G. E. Fagg, E. Gabriel, Z. Chen, T. A. G. Bosilca, J. Pjesivac-Grbovic, , and J. J. Dongarra. Process fault-tolerance: Semantics, design and applications for high performance computing. *International Journal of High Performance Computing Applications*, 19(4):465–477, 2005.
- [7] T. Fahringer and C. S. Jr. Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In *Proceedings of the HiPC*, 2002.
- [8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.
- [9] H. D. F. Group. *HDF5 Reference Manual*, September 2004. Release 1.6.3, National Center for Supercomputing Application (NCSA), University of Illinois at Urbana-Champaign.
- [10] V. Intel.
- [11] G. Jost, H. Jin, J. Labarta, J. Gimenez, and J. Caubet. Performance analysis of multilevel parallel applications on shared memory architectures. In *International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003. Nice (France).
- [12] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. <http://www.mpi-forum.org>.
- [13] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. <http://www.mpi-forum.org>.
- [14] A. Petit, R. Whaley, J. Dongarra, and A. Cleary. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Version 1.0.
- [15] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An algebra for cross-experiment performance analysis. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 63–72. IEEE Computer Society, August 2004.
- [16] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users’ Group Meeting*, number 2840 in LNCS, Venice, Italy, Sept. 2003. Springer.
- [17] F. Wolf. *EARL-API Documentation*, 2003. Version 2.0, University of Tennessee, Forschungszentrum Jülich.
- [18] F. Wolf. *EPILOG Binary Trace-Data Format*, 2003. Version 1.1, University of Tennessee, Forschungszentrum Jülich.
- [19] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.