The PerSyst Monitoring Tool

A Transport System for Performance Data Using Quantiles

Carla Guillen, Wolfram Hesse, and Matthias Brehm

Leibniz Supercomputing Centre, Germany*

Abstract. This paper presents a systemwide monitoring and analysis tool for high performance computers with several features aimed at minimizing the transport of performance data along a network of agents. The aim of the tool is to do a preliminary detection of performance bottlenecks on user applications running in HPC systems with a negligible impact on production runs. Continuous systemwide monitoring can lead to large volumes of data, if the data is required to be stored permanently to be available for queries. For system monitoring level we require to store the monitoring data synchronously. We retain the descriptive qualities by using quantiles; an aggregation with respect to the number of cores used by the application at every measuring interval. The optimization of the transport route for the performance data enables us to precisely calculate quantiles as opposed to quantile estimation.

1 Introduction

In order to have a running machine used as efficiently as possible we identified the need to do systemwide monitoring at application level. Inefficient applications prevent a petaflop system from producing more scientific results compared to an efficient used supercomputer. The preliminary detection of inefficient applications running in a petaflop system enables us to select the applications which need to be optimized. Thus, acquiring performance data of a supercomputer is necessary. Nevertheless, not all the performance data is necessary for analyzing performance; it is sufficient to retain a descriptive measure per application. The PerSyst Monitoring tool uses a fixed number of quantiles for performance monitoring. Quantiles have proven to be sufficient to retain the quality of the performance data for bottleneck detection [7]. The tool also features system level measurements. Thus, the synchronization of the measurements throughout the entire machine was required. The tool copes with a systemwide synchronization and extraction of data from a petaflop system. This is achieved with two main ideas: firstly, by using a tree agent hierarchy which extracts data with optimized routes; and secondly, by using statistical aggregation of data. Performance data is correlated with the job[1] information provided by the resource manager. The

* This work has been funded by the BMBF, grant 01IH13009A (the FEPA project).

[1] A job is a scheduled application that runs in a supercomputer.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 363–374, 2014.

job information and topology determines how data will be optimally extracted from the transport system. The reduction of the amount of data is done by aggregating at the application level using a fixed number of quantiles. We retain the descriptive qualities by calculating the quantiles with respect to the number of cores used by the application. Given that the number of quantiles is fixed, it is not necessary to store data ranges or histogram bins. By doing this we have a data agnostic database as we do not require previous knowledge of the ranges where the data lies. The percentiles adjust to the range of data available at a given monitoring interval.

Depending on the job size, we may use the tree topology partially and in the most efficient way. System level monitoring is possible by having the distribution of all the jobs together with the monitoring data from unused cores. The aggregations can then be performed for a monitoring interval at system level.

Jobs are assigned to agents that gather the performance data such that the distribution among these agents is as balanced as possible and take into consideration the topological closest distance to the entire job. If job information is collected centrally, calculating the accumulated frequency can be done without estimations.

Jobs that can't be handled at one collector are distributed to the nearest collectors in the tree of agents. These jobs will require an estimation of quantiles based on quantile data obtained at each collecting agent. The calculated quantile subsets are pushed upwards in the topology network using, only in this case, the already existing solutions of a reduction network. The monitoring system has already been deployed in an Itanium IA2 architecture based SGI supercomputer system with 9728 cores, in a BladeCenter HX5 supercomputer based on Intel Xeon architecture with 8200 cores, and has been adapted at a IBM System x iDataPlex Sandy Bridge-EP Supercomputer with 147,456 Cores.

In Section 2 related work is described. Section 3 deals the details of the PerSyst Monitoring tool's transport system. Estimation of quantiles is explained in Section 4. The approach for collecting performance data of jobs is explained in Section 5. Results of the Sandy Bridge-EP system are described in Section 6. We finally conclude in Section 7 and give a brief outlook of the tool.

2 Related Work

There are other tools which have a tree hierarchy architecture for extracting and/or storing data. The Multicast Reduction Network tool (MRNet) is a tool for parallel applications enabling high-throughput communications [10]. Although MRNet is not, per se, a performance measuring tool it can be used for these purposes [2]. MRNet uses the principle of a hierarchy of software in a tree topology, also referred to as a tree-based overlay network, for scaling to hundreds of thousands of cores. Multicast is done from the frontend downwards through the tree, until the command reaches the leaves of the tree-topology. Transport of data is done with a bottom-up logic, i.e. from the leaves of the tree to the frontend. Aggregation can be implemented via customisable filters to aggregate data

packets. The filters, however, can aggregate data only from piece-wise continuous aggregation functions. The NWPerf tool [9] uses a hierarchical structure to extract performance data without statistical aggregation. This tool provides systemwide monitoring of performance counters for high performance computers. Periscope [5] is a scalable tool for analyzing the performance of a single application. It enables a distributed on-line search for performance metrics based on hardware counters as well as metrics for MPI and OpenMP [6]. Periscope uses a hierarchy of agents to extract information and to send commands to the leaves of the tree hierarchy. Distributed hierarchical storage also use the idea of a tree structure to query performance data [3].

The PerSyst Monitoring tool has been developed as an overlay of distributed software with a tree agent hierarchy. Using a tree structure we overcome many scalability problems, just like other existing tools. However, data collection and extraction is done differently, making it a distinct tool from other tree overlay network tools. We exploit the topology of the running jobs to optimize the extraction of performance data on a large cluster. The storage of the performance data is done as close as possible to the measurement source, instead of sending the information through the entire tree of agents. A difference to other hierarchical tools is that the collecting agents of a job will have a common and smaller subtree whose root node will finally process the job instead of the frontend thereby avoiding the usage of the entire tree topology.

3 The Transport System

The PerSyst Monitoring tool has three types of agents. These are the synchronization agent, or *SyncAgent*; the *Collector Agent*; and the *PerSyst Agent*, as shown in Figure 1. The main functionality of the SyncAgent is to synchronize measurement, the Collector Agent collects the performance data, and the PerSyst Agent performs the measurements. Every type of agent has a core framework that implements the communication and the basic functionality. The framework provides interfaces which allow the use of ad-hoc delegates. The delegates interact with batch schedulers and system measuring interfaces. This ensures the portability of the tool.

The PerSyst Agents measure at the synchronized command of the frontend. The frontend is the SyncAgent at the root node and orchestrates the rest of the tree. The communication protocol used is TCP/IP, a reliable communication protocol compared to the UDP protocol. While the SyncAgents can only perform estimation of quantiles, the layer of Collector agents performs exact calculations of quantiles. If the collection of performance data is needed at a SyncAgent, the Collector and SyncAgents involved respect the parent-child relation of the original tree configuration. The PerSyst Agents, conversely, send the performance data to an optimized route in the agent tree.

The aggregation of subsets of percentiles is not possible using the definition and can only be done using estimations, thus two types of aggregations are

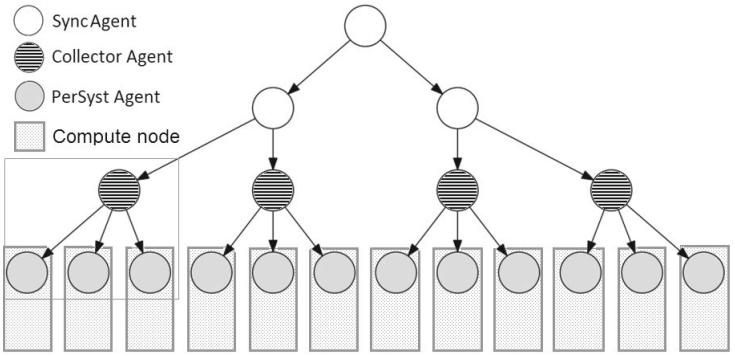

Fig. 1. Agent hierarchy

necessary at different levels of the hierarchy tree. Brim et al. [2] use the same aggregations function (or filters according to their terminology) among the software components which we changed to avoid estimations as much as possible. The top-down control of agents was kept, by sending the command through the tree structure of the agents, just like other hierarchical tools. However, the response of the PerSyst Agents is not necessarily directed to their Collector parent.

4 Estimation of Quantiles

For practical purposes, the definition and implications of using percentiles will be used hereafter. Other quantiles (for example quintiles, quartiles, or deciles) can be adapted to the definitions and usage.

The standard definition [4,8] is the kth percentile P_k is a value within the range of x, say x_k, which divides the data set into two groups. The fraction of the observation specified by the percentile falls below and its complement falls above. Thus, it is necessary to obtain the empirial cumulative distribution function, hereafter *cdf*, of the variate x to calculate any given percentile. To calculate the kth percentile of a distribution, P_k, the value of x_k which corresponds to the element position $\frac{Nk}{100}$ in the cdf is taken, where N is the sample size. When $\frac{Nk}{100}$ is not an integral value the linear interpolation of the cdf between the value corresponding to $\lfloor \frac{Nk}{100} \rfloor$ in the cdf and the next value corresponding to the cdf (i.e. $(\lfloor \frac{Nk}{100} \rfloor + 1)$) is calculated.

A feature of monitoring systems with tree topologies is that they can be configured to perform meta-aggregations[2]. If percentiles are used, estimations are required when an application requires the use of the entire tree topology, i.e. to apply meta-aggregation of percentiles. The percentiles per job are collected

[2] The term meta-aggregation refers to performing aggregations of aggregated sets. For example, calculating averages from the averages of multiple sets.

within an agent that aggregates subsets of percentiles. These percentiles are collected and estimated at each common parent of the Collectors. At each common parent the estimates are done by inferring the population of each Collector per job. For example, take

$$P_1 = \{p_0^1, p_1^1, ..., p_{100}^1\} \tag{1}$$

as the percentiles from Collector 1, C_1 and

$$P_2 = \{p_0^2, p_1^2, ..., p_{100}^2\} \tag{2}$$

as the percentiles from Collector 2, C_2. Both P_1 and P_2 belong to the same job such that the new percentiles need to be estimated from both of them. Given that a distribution is not known a priori, the entire set of observations from P_1 and from P_2 is estimated assuming a uniform random distribution between each percentile P_k and P_{k+1}. As seen in Figure 2, uniform random distribution assumes that the data between two deciles is uniformly increasing and curves in the cdf are replaced with a line joining two deciles. The percentile values themselves do not need to be changed; they are part of the newly recreated set. For example:

$$S_1 = \{p_0^1, r_1^1, r_2^1, ..., p_1^1, r_n^1..., p_{100}^1\} \tag{3}$$

and

$$S_2 = \{p_0^2, r_1^2, r_2^2, ..., p_1^2, r_n^2..., p_{100}^2\} \tag{4}$$

where r are the random values, and S_1 and S_2 are the recreated sets. The new estimated set is then $S = S_1 \cup S_2$. The random values are produced in such a way that they lie within the range of two neighboring percentiles, thus the value r_i lies between $p_k \leq r_i \leq p_{k+1}$. The number of random values $R(k, k+1)$ between two neighbouring percentiles, k and $k+1$, where $k \geq 1$ is

$$R(k, k+1) = \frac{N_o}{n_p} - 1 \tag{5}$$

where N_o is the total number of observations and n_p is the number of percentiles (example: $n_p = 100$ when all percentiles are used, and $n_p = 10$ when only deciles are used). This formula applies except for the first interval, given that

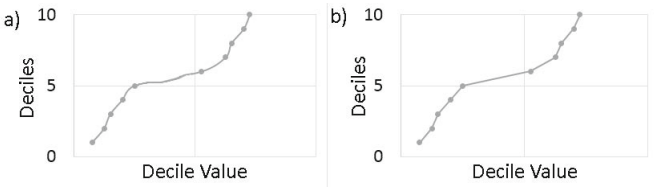

Fig. 2. Example of approximating a population with uniform distribution. Graph a) represents the real distribution. Graph b) represents an estimation using uniform distribution.

the minimum (considered to be the percentile zero) is in this range, there is one less random value to produce:

$$R(0,1) = N_o/n_p - 2 \tag{6}$$

Both sets S_1 and S_2 are grouped together and they form the estimated observations of the collectors $C_1 \cup C_2$. The cdf is calculated from S, the estimated population. The percentiles are then determined from the estimated population. Analogously, this method can be applied to more than two sets, i.e. percentiles coming from more than two Collectors. Once all the estimated sets are joined together an estimated but complete population is obtained whose cdf can be determined as well as its global percentiles.

5 Collection of Jobs

The decision as to where and at what point the information will be processed is calculated by a job balancer which is integrated in the frontend of the collection system. For every measuring interval, the job balancer will assign the jobs to a collection route (in a large cluster new jobs may appear, while other jobs are terminated and removed). This also ensures that a same job which is reassigned to other nodes will also be reassigned to a new collection route[3].

The PerSyst Agents do not have knowledge of all of the available collectors only of their parent Collector. When the measuring command arrives they also receive information of the route in the tree where they should send the data. The route specifies either the Collector to whom they should send the performance data or if the agent itself can aggregate the performance data and perform the output. After the measurement cycle is completed this information is then lost. The only information kept is the communication address of the parent. Algorithm 1 is the main algorithm which performs this balancing.

When the job size fits exactly in one compute node[4], the job is processed locally. Requests that exceed the capacity of a database, or file system, or any other storage method, will create a bottleneck. Thus, if these requests are exceeding the limits imposed by the storage medium, the jobs are sent through the network tree. l_j and l_{max} are called loads, and the terms represent the amount of performance data of a job l_j or the maximum amount of performance data a Collector can take l_{max}. For jobs where $l_j \leq l_{max}$, it is only necessary to use one Collector and not the entire tree structure for extracting and collecting data. Using the entire tree rather than a part of it implies using more communication. l_{max} depends on the HPC System and the amount of performance data collected. These jobs are defined as medium sized jobs (i.e, jobs whose load $l_j \leq l_{max}$ and

[3] The tool would, therefore, redistribute the job collection even with migration of computations to another hardware architecture, if the new job placement information is made available by the batch scheduler.

[4] A compute node refers to an operating system instance which runs on one or more cores with shared memory.

Algorithm 1. Algorithm to distribute jobs to collectors.

Require:

$$\left\lceil \frac{\sum_J l_j}{l_{max}} \right\rceil \leq C \tag{7}$$

Where C is the set of collectors and l_j is the amount of performance data, or load, from job j and J is the set of all jobs at a measuring interval. l_{max} is the maximum performance data amount a Collector can take.

1: Sort jobs J in descending order of load l_j {A job j is running on different compute nodes monitored by agents A_j.}
2: Initialize all collectors in C:
3: **for all** load l_c of $c \in C$ **do**
4: $l_c \leftarrow 0$
5: **end for**
6: Set loads from jobs to collectors:
7: **for all** $j \in J$ **do**
8: **if** $l_j = 1$ **then**
9: Mark j to be processed directly at A_j
10: Continue to next j in the for-loop
11: **end if**
12: $c \leftarrow$ FindBestCollector(C, A_j)
13: $l_{temp} \leftarrow l_j + l_c$.
14: **if** $l_{temp} > l_{max}$ **then**
15: DistributeLoadOnCollectors(C, l_j, A_j)
16: **else**
17: Assign $l_c \leftarrow l_{temp}$
18: **end if**
19: **end for**

which run in more than one compute node). Medium sized jobs are collected at one Collector and the aggregation is done with a precise calculation of the percentiles. In this case applying Algorithm 1 with the `FindBestCollector` algorithm (Algorithm 3) will be sufficient to determine where the job should be sent to.

The `FindBestCollector` Algorithm finds the Collector with the minimum assigned performance data load (l_c in Algorithm 1). When minima are found the algorithm considers also the topological distance of a Collector and a PerSyst agent so jobs will be sent to their closest Collector. The topological distance, *td*, of two tree nodes (leaves or nodes) has been defined to be the longest distance between each node and their common collection node, i.e. the longest distance that the data has to travel such that it is collected centrally at the root of the smallest sub-tree. As described in Algorithm 2, the jobs are distributed to their closest Collector or Collectors. The closest Collector to a job is defined as the Collector with the minimum total *td* of itself with respect to all the PerSyst Agent nodes were the job is running. By calculating the *td*, the algorithm

Algorithm 2. Algorithm to find Collector with minimum load and minimum topological distance.

```
1: Algorithm FindBestCollector(C, A)
2: C' ← all c_i with minimum load.
3: if |C'| > 1 then
4:    for all c' ∈ C' do
5:       Set d ← 0 where d is the topological distance from c to a.
6:       for all a ∈ A do
7:          d ← d+ TopologicalDistance(c, a)
8:       end for
9:       Insert c' and d in Collector-Distance ordered map. {c' ∈ C' is mapped to the
         total distance d_c' ∈ D with f(c') ↦ d_c' with a surjective mapping f : C' ↦ D
         }
10:   end for
11:   return c'' {where c'' = f^-1(min(D)), i.e. the collector with minimum distance.
      If |f^-1(min(D))| > 1, ie more than one collector, only the first one is returned.}
12: else
13:   return c' {where c' ∈ C' with minimum load.}
14: end if
```

guarantees that the normal parent-child relations are used as much as possible. This avoids sending additional Collector information to the PerSyst Agents more than necessary.

When $l_j > l_{max}$ the job size is handled with percentile estimation and use the tree partially to fit the collection in the lowest possible number of collectors to extract the information; these jobs are called for convenience big jobs. Algorithm 3 shows how this distribution is done. The idea is to use the tree structure only when it is necessary, otherwise aggregate with exact calculations and store information as quickly and as closest to the source as possible.

The `DistributeLoadOnCollectors` algorithm is similar to the previous algorithm `FindBestCollector`. The main difference is that the number of collectors n_c where the job will be collected is determined. Once the algorithm determines which Collectors will be used, the remaining load is distributed among them. The last remaining task is to calculate the common collection node among the SyncAgents of an entire job. With tree operations the agent responsible for big jobs can be determined. Medium sized jobs finish their collection at one Collector. One-node jobs finish their collection at the PerSyst agent in charge of monitoring it's node. Figure 3 shows the different possibilities of retrieving a job.

The algorithm that calculates the topological distance is not shown but has a time complexity of $O\left(log(n)\right)$ as it reduces to a tree search. The complexity of the calling algorithm, including all the calls, is therefore $O\left(n^2 log(n)\right)$, where n is the number of the measuring agents.

Even though the measurements are done synchronously, the collection is done asynchronously, i.e. which ever process finishes collecting a job's data will start

Algorithm 3. Algorithm to distribute performance data load in several collectors.

Algorithm DistributeLoadOnCollectors(C,l_j, A_j)

$$n_c \leftarrow ceiling(\frac{l_j}{l_{dist}}) \tag{8}$$

{where n_c is the number of Collectors that will receive the job performance data, also referred to as load, from all agents A_j. l_{dist} refers to a defined distribution load the Collectors will take, thus $l_{dist} < l_{max}$}

iter $\leftarrow 0$
$A'_j \leftarrow A_j$ where A'_j is a temporary variable for agents of a job.
while iter $< n_c$ **do**
  $c \leftarrow$ FindBestCollector(C, A'_j)
  insert c in C' set.
  Child agents of c allocate their load in c
  Remove all agents a which are children of c from A'_j
  $iter \leftarrow iter + 1$
end while
Place the rest of the load on the first n_c Collectors found:
for all $a \in A'_j$ **do**
  $c \leftarrow$ FindBestCollector(C', A_j)
  place load of a in c:
  $l_c \leftarrow l_a$
end for

performing the output. The measurements are associated to the synchronized measuring interval. The collection of the performance data asynchronously alleviates the amount of synchronized communication of extracted data on a large cluster.

Extremely big jobs, like those which take up an entire petaflop system, are also handled. The solution is to collect them like the typical procedure other hierarchical tools would do, having aggregation at the middleware of the tree topology that provide's quantile estimations.

6 Results

The PerSyst Monitoring Tool runs currently in production mode in an IBM X Series Cluster system, hereafter SuperMUC, which is based on Intel Sandy Bridge-EP processors and Mellanox FDR-10 Infiniband technology. SuperMUC comprises 18 thin node islands, among other systems. Each thin island has 516 nodes each having two Sandy Bridge-EP Intel Xeon E5-2680 processors with a total of 16 cores per node (a Sandy Bridge-EP processor has 8 cores). A thin island consists of 512 nodes (8256 cores). All individual islands are connected internally via a fully non-blocking infiniband network. SuperMUC has, thus, 9,216 nodes with a total of 148,608 cores in the thin islands. Faster interconnects are available at the level of the island. The batch scheduler does not allow users

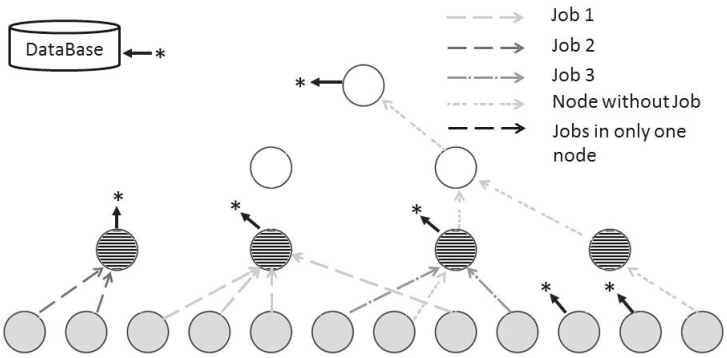

Fig. 3. Example of retrieval of performance data

to share a compute node, thus, a compute node is taken exclusively for a job. There are four job classes; each allows submissions with a different range of job sizes.

The PerSyst Monitoring tool was configured to run as one instance (one tree of agents with one frontend) on SuperMUC. The fanout of the tree hierarchy consisted on one SyncAgent as a frontend, 12 SyncAgents as a middle layer, 216 Collectors, and 9288 PerSyst Agents at every compute node. The 13 SyncAgents were placed at an external node which is used for administrative tasks. Six Collectors per island were placed having each 43 PerSyst Agents (child agents). Parent-child relations among Collectors and PerSyst Agents were placed in the same island. Thus, the tree agent topology exploited the faster interconnects with these placements. The tool was configured to run and aggregate using deciles (10 percentiles). Table 1 shows the collection of jobs at the different levels and the average values from 10 measurements of the job sizes running at the same time. The collection per job at a single point for percentile aggregation is done only at the Collector Agent level and at the PerSyst Agent level. The estimation of percentiles is on average 91% circumvented.

The topology network can be used fully when all the jobs travel to the frontend and are processed at each tree node. The alternative is to try to collect at selected

Table 1. Distribution of jobs in agent tree. Taken from 10 measurements in 10 days.

Tree level	Average number of Jobs	Percentage
Frontend	4.8	2.59%
SyncAgents	11.8	6.37%
Collector Agents	157.5	85%
PerSyst Agents	11.2	6.04%
Total number of jobs	185.3	100%

Table 2. Usage of the topology network for 58 measurements taken during a week

Job retrieving method	Average number of nodes used
Jobs travel through established topology connections until frontend	905.81
Jobs travel to selected nodes with job load balancing algorithm	217.59

Table 3. Collection time from PerSyst Agents to Collectors at a measuring interval

	Performance data one Collector	Performance data of SuperMUC
Bytes	204,426	44,156,016
Average time [s]	0.85	0.85
Used bandwidth [MiB/s]	0.22	49.54

nodes with the job load balancing algorithm (described in Section 5) and perform the output when the job has been collected. Table 2 shows the topology network usage with these two different methods.

The results show that the usage of the topology nodes is more than a factor of four with the traditional bottom-up retrieval of job information. To obtain the time it took to transmit the performance data, nine islands were measured and the average per Collector Agent was taken. Note that the transmission through the network interconnect between nodes includes measurement times and processing times within the PerSyst Agents. As soon as a performance datum is available, depending on the available data, it is sent in groups to the Collectors. This is done in order to not congest the network with performance data and explains the low bandwidths obtained.

By using deciles, we are able to reduce the amount of data more than 91% of the total amount of performance data in a week. No matter how big the job is, its information is compressed to 13 data points per monitoring interval: the deciles, the minimum (considered decile zero), the number of observations, and the average.

7 Conclusions

Percentiles have proven to be effective in data reduction. Due to the use of percentiles, two different kinds of aggregations are needed that produce exact calculations at certain nodes and other type of aggregations that estimate the new set of percentiles from meta-aggregation. In order to avoid meta-aggregation of percentiles, the transport systems adapts to the jobs' topological placement

in the supercomputer. Not only the estimations are avoided, but the extraction of data is optimized compared to the traditional extraction that uses the entire tree topology.

Future work includes using a PerSyst Agent also as a Collector for doing collecting tasks in order to further optimize the amount of resources deployed in the supercomputer. Furthermore, optimizations in the algorithms presented will be carried out in order to reduce the time complexity of the job balancer.

References

1. Benedict, S., Brehm, M., Gerndt, M., Guillen, C., Hesse, W., Petkov, V.: Automatic performance analysis of large scale simulations. In: Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A. (eds.) Euro-Par 2009. LNCS, vol. 6043, pp. 199–207. Springer, Heidelberg (2010)
2. Brim, M.J., DeRose, L., Miller, B.P., Olichandran, R., Roth, P.C.: Mrnet: A scalable infrastructure for development of parallel tools and applications. In: Cray User Group 2010 Proceedings (2010)
3. Focht, E., Jeutter, A.: AggMon: Scalable Hierarchical Cluster Monitoring. In: Proceedings of the Joint Workshop on High Performance Computing on Vector Systems (2012)
4. Frank, I.E., Todeschini, R.: The data analysis handbook, vol. 14. Elsevier Science B.V (1994)
5. Gerndt, M., Fuerlinger, K.: Automatic performance analysis with periscope. In: Journal: Concurrency and Computation: Practice and Experience, Wiley InterScience. John Wiley & Sons, Ltd. (2009)
6. Gerndt, M., Fuerlinger, K., Kereku, E.: Periscope: Advanced techniques for performance analysis, parallel computing: Current & future issues of high-end computing. In: International Conference ParCo 2005. NIC Series, vol. 33 (2006) ISBN 3-00-017352-8
7. Guillen, C., Hesse, W., Brehm, M.: A new scalable monitoring tool using performance properties of hpc systems. In: Bischof, C., Hegering, H.-G., Nagel, W.E., Wittum, G. (eds.) Competence in High Performance Computing 2010, pp. 51–60. Springer, Heidelberg (2012) 10.1007/978-3-642-24025-6.5
8. Mendenhall, W., Sincich, T.: Statistics for engineering and the sciences, 4th edn. Prentice-Hall International, Inc. (1995) ISBN 0-13-181017-0
9. Mooney, R., Schmidt, K.P., Studham, R.S.: NWPerf: a system wide performance monitoring tool for large Linux clusters. In: IEEE International Conference on Cluster Computing, pp. 379–389. IEEE Computer Society, Los Alamitos (2004)
10. Roth, P.C., Arnold, D.C., Miller, B.P.: Mrnet: A software-based multicast/reduction network for scalable tools. In: Proc. IEEE/ACM Supercomputing (2003)