Outlier Detection in Performance Data of Parallel Applications

Katharina Benkert [†*], Edgar Gabriel [†] and Michael M. Resch [*]

[†] Parallel Software Technologies Laboratory, Department of Computer Science,
University of Houston, Houston, TX 77204-3010, USA.
Email: gabriel@cs.uh.edu

[*] High Performance Computing Center Stuttgart (HLRS),
Nobelstr. 19, 70569 Stuttgart, Germany.
Email: { benkert, resch }@hlrs.de

***Abstract*— When an adaptive software component is employed to select the best-performing implementation for a communication operation at runtime, the correctness of the decision taken strongly depends on detecting and removing outliers in the data used for the comparison. This automatic decision is greatly complicated by the fact that the types and quantities of outliers depend on the network interconnect and the nodes assigned to the job by the batch scheduler. This paper evaluates four different statistical methods used for handling outliers, namely a standard interquartile range method, a heuristic derived from the trimmed mean value, cluster analysis and a method using robust statistics. Using performance data from the Abstract Data and Communication Library (ADCL) we evaluate the correctness of the decisions made with each statistical approach over three fundamentally different network interconnects, namely a highly reliable InfiniBand network, a Gigabit Ethernet network having a larger variance in the performance, and a hierarchical Gigabit Ethernet network.**

Keywords: outlier detection, performance analysis, adaptive communication libraries

I. Introduction

The constant increase of complexity of clustered high performance computing environments entails resource-intensive tuning to exploit the capabilities of the given hardware and software environment. Certain software components can be tuned for a given platform before the execution of the application. This approach has been taken by several projects such as ATLAS [1] or ATCC [2]. However, several features influencing the performance of the application can only be determined while executing the application itself, since the performance depends on factors such as process placement [3], resource utilization (e.g. multiple parallel jobs running simultaneously) and application characteristics (e.g. message sizes depending on input data). To overcome the limitations of statically tuned software, we have recently introduced the Abstract Data and Communication Library (ADCL) [4]. The main goals of ADCL are threefold: firstly to define higher level abstractions for often occurring application level communication patterns, secondly to provide a large number of implementations for each communication pattern and thirdly to choose at runtime the implementation giving the best performance.

A key component of ADCL is the algorithm used to determine when a particular implementation is considered to be 'faster' than another implementation. Based on per-process performance data of each implementation, the algorithm has to be able to exclude incidental outliers, generate a consistent result across all processes while being at the same time computation and communication wise inexpensive in order to minimize the disruption of the application.

Outliers have mostly been defined in a strict statistical context as one ... *that deviates so much from other observations as to arouse suspicion that it was generated by a different mechanism* [5]. In general, outliers may occur as a result of wrong models under study, systematic errors or faulty data (errors in measurement). If the sample mean is calculated from the data, a reasonable handling of outliers is essential. Just one value $y_i \rightarrow \infty$ will make the estimated sample mean $\frac{1}{n}\sum_i y_i$ arbitrarily large and thus rendering it completely senseless.

The focus of this paper is to explore different statistical approaches for outlier detection. Firstly, removing outliers has an overwhelming influence on the decisions taken by the ADCL runtime selection logic. Secondly, the distinction between systematic errors and faulty data is particular difficult within the context of communication performance, since different types of network interconnects have different characteristics and produce therefore different types and quantities of outliers.

High performance network interconnects such as InfiniBand or Myrinet configured in a fully non-blocking mode will not get congested for the vast majority of applications and will produce highly reliable and repetitive performance data. One might still observe outliers in performance data over such high performance networks, they will however not be due to networking issues, but external sources such operating system jitter [6].

A Gigabit Ethernet switch providing the full bisection bandwidth required for the given number of cluster nodes offers similar overall characteristics as the networks described above. However, depending on the quality of the switch, the performance data might show higher variations compared to an InfiniBand or Myrinet network. These variations make the

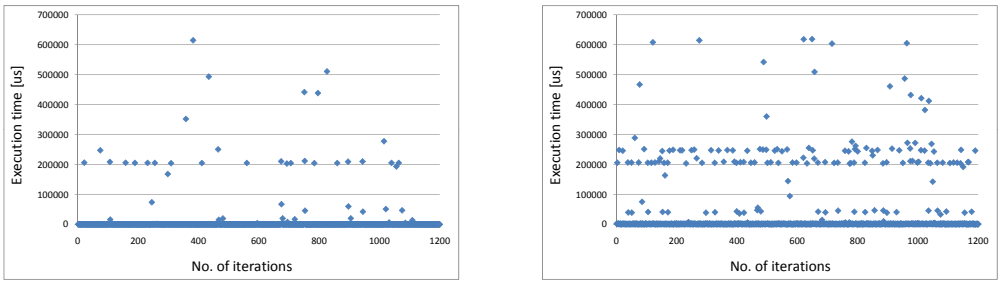

Fig. 1. Performance data of two different implementations collected on a hierarchical network on process zero.

identification of outliers more complex.

Another category of cluster networks are hierarchical networks, where (relatively) small switches are used to connect the nodes, and an additional layer of switches is used to connect these smaller, lower level switches. Due to technical or financial reasons, switches are often constructed using a fat tree topology with reduced uplink bandwidths. Examples are InfiniBand switches configured in a 2:1 blocking mode as for instance in the Thunderbird cluster at Sandia National Laboratories [7], or hierarchies of Gigabit Ethernet switches configured in many low-cost clusters or as the secondary network.

Communication patterns on such hierarchical networks can produce data points which could be considered by a trivial outlier detection algorithm as an invalid data point. Figure 1 shows the execution time per instance of two different implementations for a neighborhood communication run across a hierarchical Gigabit Ethernet network. Both implementations produce a significant number of 'outliers' at the 200,000μs range, which are due to congestion and packet drops at the higher level Gigabit Ethernet switch. The total number of 'outliers' however varies greatly. Nevertheless, these 'outliers' contain in both cases valuable information regarding the performance of the implementation due to the systematic nature of their existence. Excluding them would lead to a wrong decision of the runtime logic, but, as stated earlier, ignoring the fact that performance data can contain outliers would lead to a wrong decision by the runtime selection logic for the first two classes of networks.

Therefore, we evaluate four statistical methods from very different backgrounds to tackle this problem, including a heuristic applied so far in ADCL, a standard outlier detection algorithm from basic statistics, an approach relying on cluster analysis, and a method based on robust statistics, which does not exclude any data points. In order to compare the behavior and the prediction quality of these four methods, we have collected performance data for a variety of scenarios on all three types of networks described previously. Using additional measurements, we can evaluate the correctness of the decisions taken by each statistical approach.

More generally, while we are executing the analysis detailed in this paper in the context of a runtime library, the same methodologies have to be applied when performing an offline performance analysis of a parallel application which focuses on individual operations within the application. Thus, the evaluation of the statistical methods presented in this paper applies to a wide variety of performance analysis scenarios. The remainder of the paper is organized as follows: section II gives a brief introduction to ADCL. Section III presents four different algorithms for outlier detection. Section IV compares these algorithms for three different network settings. Finally, section V summarizes the paper and gives an overview about the currently ongoing work.

II. The Abstract Data and Communication Library

ADCL is an application level communication library aiming at providing the highest possible performance for application level communication patterns, such as the n-D neighborhood communication, within a given execution environment. The library provides for each communication pattern a large number of implementations and incorporates a runtime selection logic in order to choose the implementation leading to the highest performance. Two different runtime selection algorithms are currently available within ADCL: the library can either apply a brute force search strategy which tests all available implementations of a given communication pattern or, alternatively, a heuristic relying on attributes characterizing an implementation which has been developed in order to speed up the runtime decision procedure [4].

The ADCL API offers high level interfaces of application level collective operations. These are required in order to be able to switch the implementation of the according collective operation within the library without modifying the application itself. The main objects within the ADCL API are:

- `ADCL_Topology`: provides a description of the process topology and neighborhood relations within the application.

- `ADCL_Vector`: specifies the data structures to be used during the communication. The user can for example *register* a data structure such as a vector or a matrix with the ADCL library, detailing how many dimensions the object has, the extent of each dimension, the number of halo-cells, the basic datatype of the object, and the pointer to the data array of the object.
- `ADCL_Function`: each ADCL function is the equivalent to an actual implementation of a particular communication pattern.
- `ADCL_Fnctset`: a collection of ADCL functions providing the same functionality. ADCL provides predefined function sets, such as for neighborhood communication (`ADCL_FNCTSET_NEIGHBORHOOD`). The user can however also register its own functions in order to utilize the ADCL runtime selection logic.
- `ADCL_Attribute`: abstraction for a particular characteristic of a function/implementation. Each attribute is represented by the set of possible values for this characteristic.
- `ADCL_Attrset`: a collection of ADCL attributes
- `ADCL_Request`: combines a process topology, a function-set and a vector object. The application can initiate a communication by starting a particular ADCL request.

Figure 2 gives a simple example for an ADCL code, using a 2-D neighborhood communication on a 2-D process topology.

```
double vector[...][...];
ADCL_Vector vec;
ADCL_Topology topo;
ADCL_Request request;

/* Generate a 2-D process topology */
MPI_Cart_create (MPI_COMM_WORLD, 2, cart_dims,
   periods, 0, &cart_comm);
ADCL_Topology_create (cart_comm, &topo );

/* Register a 2D vector with ADCL */
ADCL_Vector_register (ndims, vec_dims,
   NUM_HALO_CELLS, MPI_DOUBLE, vector, &vec);

/* Combine description of data structure and
   process topology */
ADCL_Request_create (vec, topo,
   ADCL_FNCTSET_NEIGHBORHOOD, &request );

/* Main application loop */
for (i=0; i<NIT; i++ ) {
   ...
   /* Initiate neighborhood communication */
   ADCL_Request_start (request );
   ...
}
```

Fig. 2. Code sample for a parallel application using a regular 2-D neighborhood communication based on ADCL and MPI.

A key concept of ADCL is its ability to select the fastest of the available implementations for a given communication pattern during the regular execution of the application. The approach chosen for ADCL is to use the first iterations of the application to determine the fastest available implementation. Although some of the tested implementations may not deliver the optimal performance, this approach avoids a separate 'planner' step which does not contribute at all to the solution of the problem and ignores other influences on the communication process such as work imbalances. A planner also has the disadvantage that once a decision is made, it is kept for the whole execution time, even if the networking conditions change significantly compared to the initial evaluation. With ADCL on the other hand, one could easily use a monitoring interface, which restarts the runtime decision logic whenever necessary. During the first iterations, ADCL will call each implementation multiple times in order to determine the fastest implementation from a given set of functions. Each process keeps track of the execution times of all implementations in a data array which is attached to the according `ADCL_Request`. After all implementations have been tested the required number of times, the measurements are analyzed mainly locally with a statistical method. Then, depending on the evalutation method used, the method judged fastest is chosen.

III. Techniques for Performance Evaluation

For each algorithm i on each process j, we perform n measurements and denote the execution time of the k-th measurement by $t(i,j,k)$. For the first three methods presented, outliers, i.e. measurements not fulfilling a certain condition $\mathcal{C}$, are removed from the data set. This leads to a filtered subset

$$M^f(i,j) = \{t(i,j,k) \,|\, t(i,j,k) \text{ fulfills } \mathcal{C}\} \tag{1}$$

of measurements with cardinality $n_f(i,j)$. Then, common to all methods, the performance measurements for every method are analyzed locally on each processor and characterized by an estimate of the mean. This can be the local average execution time

$$m(i,j) = \frac{1}{n}\sum_k t(i,j,k) \tag{2}$$

or its filtered counterpart

$$m_f(i,j) = \frac{1}{n_f}\sum_{k\in M^f(i,j)} t(i,j,k). \tag{3}$$

After a global reduction, a maximum average execution time for each algorithm over all processes is determined. Its minimum is chosen as the best-performing algorithm.

A. Standard approach

A common approach to remove outliers from a data set consists in using the interquartile range IQR which is defined as the difference between the third and first quartile [8]. A quartile, in turn, is any of the three values that divides a sorted data set into four parts of equal size. The first quartile Q1 marks the end of the first 25% of data, the third quartile Q3 the beginning of the last 25%. Every value being smaller than $\text{Q1}-1.5\,\text{IQR}$ or larger than $\text{Q3}+1.5\,\text{IQR}$ is regarded as outlier.

Outliers are discarded by setting $\mathcal{C}$ to

$$\mathrm{Q1} - 1.5\,\mathrm{IQR} \leq t(i,j,k) \leq \mathrm{Q3} + 1.5\,\mathrm{IQR}$$

in (1) and the filtered mean (3) is computed. After a global reduction, the maximum filtered average execution time for each implementation

$$m_f(i) = \max_j m_f(i,j) \tag{4}$$

is calculated and the best-performing algorithm i' is chosen as the one fulfilling

$$m_f(i') = \min_i m_f(i).$$

B. Heuristic approach

This procedure based on a heuristic was first presented in a simpler form in [4]. There it was shown that with the right choice of two parameters, namely a bound b, which defines a measurement to be an outlier, and the maximum number of accepted outliers $nmax_o$, satisfactory results can be obtained. Nonetheless, without preceding tests on an architecture a non-optimal selection of M^f and, consequently, of the best-rated implementation, is possible.

$\mathcal{C}$ is defined as $t(i,j,k) \,|\, t(i,j,k) < b \cdot \min_k t(i,j,k)$. Apart from eq. (2) and (3), the number of outliers $n_o(i,j) = n - n_f(i,j)$ is computed locally for each algorithm.

After the global reduction, the maximum average execution time

$$m(i) = \max_j m(i,j),$$

the maximum average execution time considering only filtered data (4) and the maximum number of outliers

$$n_o(i) = \max_j n_o(i,j)$$

for each implementation is determined. Finally, we select the maximum execution time including or excluding outliers by

$$r(i) = \begin{cases} m_f(i) & \text{if } n_o(i) \leq nmax_o \\ m(i) & \text{otherwise} \end{cases}$$

depending on whether the maximum number of outliers is exceeded or not. The implementation i' fulfilling $r(i') = \min_i r(i)$ is chosen as the best one.

Remark: This procedure is related to the trimmed mean, which deletes a certain percentage of observations from each end of the data and then computes the mean in the usual way. Despite the fact that the trimmed mean is a simple robust estimator of location (see III-D for more details on robust methods), its application in this context is not beneficial since outliers from below hardly appear and have much less impact on the mean value than outliers from above.

C. Cluster Analysis

Cluster analysis [9] is a common name for a variety of mathematical methods to partition a data set into subsets called clusters such that similar objects are grouped together. The similarity or dissimilarity is expressed by a distance measure. The average linking method employed in this paper belongs to the group of hierarchical clustering methods which agglomeratively build up a hierarchy. The visual representation of the hierarchy in a treelike structure is denoted as dendrogram. Since outliers become small, isolated clusters, cluster analysis provides a natural mean to decide whether a point is an outlier or not. This eliminates the need to specify a bound b as in sec. III-B, but a proper choice of the parameter $nmax_o$ remains.

For the analysis of performance data, we remove, starting from the top of the dendrogram, iteratively those clusters, which have little similarity to the others and which in total do not exceed the number of accepted outliers $nmax_o$. This procedure ensures that the most isolated clusters are removed first. The current implementation utilizes the average linking method of the Open Source package Cluster 3.0 [10] with an Euclidean distance measure without prior standardization. After having removed the outliers, we proceed as described sec. (III-A).

D. Robust statistics

Robust statistics [11], [12], [13], [14] provide a statistically justified way of procurement. Its development was triggered due to problems where the clear decision whether to keep or reject a data sample can not be decided easily. Down-weighting dubious observations is considered more appropriate than removing them completely form the data set, i.e. outliers are handled automatically and appropriately by the method. This distinguishes robust statics from all methods mentioned so far which require the removal of outliers.

Real data can deviate from the often assumed normal distribution in classical methods. This departure can manifest itself in the form of longer tailed or skewed distributions. Therefore, robust parametric statistics tend to rely on replacing the normal distribution with the longer-tailed t-distribution with low degrees of freedom or with a mixture of two or more distributions. Within the framework of this analysis, the first approach is applied.

The sample data recorded for the n units, i.e., the measurements of execution time $t(i',j',k)$ for one algorithm on one processor, are denoted for the sake of simplicity by $y_1, y_2, \ldots, y_n$. Let $\mathcal{N}(\mu, \sigma^2)$ denote the normal distribution with mean μ and variance σ^2. Then the normal model assumes

$$y \overset{\text{ind}}{\sim} \mathcal{N}\left\{\mu(\theta), \sigma^2(\phi)\right\}, \tag{5}$$

where μ is a function of known form indexed by a unknown parameter θ, and $\sigma^2(\phi)$ is the variance of known form indexed by a unknown parameters ϕ. The t model replaces the normal distribution assumption (5) by

$$y \overset{\text{ind}}{\sim} t\left\{\mu(\theta), \psi(\phi), \nu\right\},$$

where $t\{\mu, \psi, \nu\}$ denotes the t-distribution with location parameter μ, scale parameter ψ and ν degrees of freedom. The meaning of ν will be explained later.

A maximum likelihood (ML) estimation is applied to make inferences about parameters of the underlying t-distribution from the given data set analogous to those for the normal model (5).

Maximizing the log-likelihood function

$$l(\theta, \phi, \nu) = \sum_{i=1}^{n} f(y_i | \, \xi) \tag{6}$$

with respect to $\xi = (\theta, \phi, \nu)$ yields ML estimates $\hat{\xi} = (\hat{\theta}, \hat{\phi}, \hat{\nu})$, where $f(y_i | \, \xi)$ is the log-density function for $t\,\{\mu(\theta), \psi(\phi), \nu\}$. Since the density function of the univariate t-distribution is

$$p(y \,|\, \xi) = \frac{\Gamma\left(\frac{\nu+1}{2}\right) \psi^{-1/2}}{\sqrt{\pi\nu}\Gamma\left(\frac{\nu}{2}\right)} \cdot \left(1 + \frac{(y-\mu)^2}{\psi\nu}\right)^{-(v+1)/2}$$

the log-density function for each component y_i is

$$\begin{aligned} f(y_i | \, \xi) \quad &= \quad -\frac{\nu+1}{2} \ln\left[1 + \frac{(y_i - \mu(\theta))^2}{\nu\psi(\phi)}\right] \\ &\quad -\frac{\ln(\psi(\phi))}{2} + g(\nu) + c, \end{aligned}$$

where

$$g(\nu) = \ln\left[\Gamma\left(\frac{\nu+1}{2}\right)\right] - \frac{\ln(\nu)}{2} - \ln\left[\Gamma\left(\frac{\nu}{2}\right)\right], \tag{7}$$

and

$$c = -\frac{1}{2}\ln(\pi).$$

As explained in [15], for $\nu < \infty$, maximum likelihood estimation of θ and certain functions of ϕ are robust in the sense that outlying cases with large Mahalanobis distances $\delta_i^2 = (y_i - \mu)^2/\psi$ are down-weighted. The degree of down-weighting of outliers depends reciprocally on ν. If ν is fixed a priori at some reasonable value (degree of freedoms between 4 and 6 have often been useful in practice), it is a robustness tuning parameter. For larger data sets, ν can also be estimated from the data by ML, yielding an adaptive robust procedure.

To maximize (6), we applied for the sake of simplicity the simplex algorithm of Nelder and Mead [16] which is part of the GNU Scientific Library. For constant ν, $g(\nu)$ in (7) does not enter the maximization process and was therefore omitted. We choose the median, the second quartile (see sec. III-A), as an estimate of location and a multiple of the Median of Absolute Deviations

$$\text{MAD}(y) = \text{median}_i | \, y_i - \text{median}_j(y_j) \, |$$

as a scale estimate. A factor of 1.483 is most commonly used.

After obtaining the estimate mean value μ, the same steps as in sec. III-A were applied to select the best implementation.

E. Complexity estimates

As mentioned earlier in this paper, the methods discussed within this section will be applied within a runtime library. This necessitates to minimize the computational costs of the algorithms handling outliers. Table I gives an overview about the complexity estimate for each of the four statistical approaches, with n being the number of measurements per implementation.

Statistical approach	Complexity
Interquartile range	$O(n \log n)$
Heuristic approach	$O(n)$
Cluster analysis	$O(n^3)$
Robust statistics	exponential in worst case, a lot faster in most real scenarios

TABLE I
COMPLEXITY ESTIMATE FOR EACH OF THE FOUR STATISTICAL APPROACHES

IV. EVALUATION

In the following, we evaluate the behavior each statistical method presented in the previous section for different scenarios, i.e., whether the decision of the statistical method based on a small number of measurements is comparable to a ranking obtained by long-term runs. For this, we generate performance data using a TFQMR [17] solver with Jacobi preconditioning for a convection-diffusion equation discretized with center differences. One iteration of the solver requires six neighborhood communications to calculate the necessary matrix-vector products.

Since the neighborhood communication is implemented using ADCL, the library will evaluate, given a brute-force search, all available implementations for this communication pattern a given number of times during the initial iterations of the solver. The different implementations of the neighborhood communication provided by ADCL differ in the number of simultaneous communication partners for each process (e.g., pair-wise or all-to-all), the data transfer primitives used (e.g., synchronous, asynchronous, one-sided communication) and the handling of non-contiguous data (e.g., derived data types, pack/unpack). As of now, ADCL provides twelve different implementations for the neighborhood communication which are detailed in [4]. The execution time of each individual instance of the neighborhood communication is stored in an output file on a per process basis. These files are used as the input for the subsequent analysis. Using the statistical methods presented, we calculate an estimated mean value for every implementation and select the method with the smallest one.

In order to evaluate the 'correct' decision of the statistical approaches, we additionally executed the same application forcing ADCL to use a single implementation at a time by circumventing the runtime selection logic. In order to make the conditions as comparable as possible, the reference data was produced within the same batch scheduler allocation and thus had the same node assignments. We will refer to these measurements as verification runs throughout the rest of this section.

The runs generating input for the analysis using a brute-force search as well as the verification runs have been executed three times using the MPI library Open MPI v1.2 [18]. During each run using the runtime selection logic, 50 data points have been generated per implementation and process. The prediction quality of the four evaluation methods was analyzed separately for each run. During each verification run, the

execution times of 700 iterations per implementation were stored and subsequently averaged over all three runs.

A. Shark IB

The first set of tests have been executed on the *shark* cluster at the University of Houston. This cluster consists of 24 single processor, dual core 2.2 GHz AMD Opteron nodes. Each node is equipped with a 4xInfiniBand and a Gigabit Ethernet card. We first evaluate the statistical approaches using ADCL performance data from runs over the InfiniBand network with 16, 32 or 48 MPI processes. As described in the introduction, the InfiniBand network produces highly reliable performance data with only few outliers. Furthermore, our tests showed that the performance of ten out of the twelve available implementations for the neighborhood communication is comparable to each, i.e., the data points of different implementations are fairly close to each other. One might argue that any of the ten good performing implementations would be considered an acceptable choice, but this scenario is highly relevant for any network interconnect. With an increasing number of available implementations in ADCL, more and more implementations will only differ in nuances. Thus, any statistical method applied within ADCL must be able to handle performance data of highly comparable implementations for 'fine tuning'.

To summarize the results of the InfiniBand tests, all four statistical approaches presented in the previous section performed reasonably well. The method using the robust statistics performed best, since it managed to determine the fastest implementation most often for the data sets used in this subsection. The results of the standard method were slightly inferior to those of the other methods. Table II details the results obtained for each test-case. For each statistical method this table shows the rank obtained from verification runs of the implementation judged best-performing. The table also lists the decisions which would have been made by the runtime selection logic without any data filtering. Due to the very reliable performance of this network, many of the decisions without data filtering can be regarded as reasonable. However, in most scenarios, the runtime selection logic was capable of choosing a faster implementation when using a data filtering. Additionally, already minor disturbances such as in the 2nd run of the 32 processes case, lead without any data filtering to a less optimal decision.

An interesting detail in the 32 processes test case is, that the average execution time of the fastest implementation has been determined by the verification runs to take 105.41s, followed by the ones taking 105.44s and 105.56s. However, a closer look into the data sets of the verification runs reveals that the implementation ranked 9th was in two out of three instances the fastest implementation (104.99s and 105.09s), but the third measurement was significantly higher (112.96s). This signifies that the third verification run did not show the average behavior of the network. Since the input data for the methods was generated in between, we also expect a significant change. Therefore we added the ranking stemming from the execution times averaged over just the first two runs in brackets. This

Method		best-rated methods for test cases using 16 processes	32 processes	48 processes
standard	run 1	3rd	2nd (2nd)	2nd
	run 2	3rd	3rd (4th)	3rd
	run 3	3rd	2nd (2nd)	3rd
heuristic	run 1	3rd	2nd (2nd)	1st
	run 2	3rd	2nd (2nd)	3rd
	run 3	1st	2nd (2nd)	2nd
cluster analysis	run 1	3rd	9th (1st)	3rd
	run 2	3rd	9th (1st)	3rd
	run 3	1st	9th (1st)	2nd
robust statistics	run 1	1st	9th (1st)	1st
	run 2	1st	9th (1st)	1st
	run 3	1st	9th (1st)	1st
no data filtering	run 1	3rd	2nd (2nd)	2nd
	run 2	3rd	3rd (4th)	3rd
	run 3	3rd	2nd (2nd)	3rd

TABLE II
RANK IN THE VERIFICATION RUNS OF EACH IMPLEMENTATION JUDGED BEST-PERFORMING BY THE DIFFERENT METHODS FOR THE 16, 32 AND 48 PROCESSES TEST CASES OVER INFINIBAND ON *shark*

makes it less surprising that robust statistics and cluster analysis chose twice (correctly) this implementation to be the fastest one. The heuristic approach chose the second-best in all three data sets, whose performance is slightly below that of the one ranked 9th for run 1 and 2. The standard interquartile range approach made acceptable choices by judging twice the second-best as winner and once the third-best.

B. Shark GE

In this subsection, we repeat the same tests with 16, 32 and 48 processes as in the subsection IV-A using however the Gigabit Ethernet interconnect. The Gigabit Ethernet switch used within this network provides a full duplex 1GBit connection for each node. However, for the 32 and 48 processes test cases, two MPI processes are running on each node and have to share one physical link. This leads to a higher number of outliers for these two scenarios compared to the 16 processes test case.

To summarize the results achieved over the Gigabit Ethernet network on *shark*, the quality of the predictions for all statistical methods strongly depends on the number of processes. For the 16 processes runs, the execution times within the verification runs show only a small standard deviation and all methods produced good results. For 32 and 48 processes the standard deviation is significantly higher, which causes a problem for the evaluation methods. The standard method fails for the 32 processes test case and cluster analysis as well as robust statistics fail for the 48 processes case. An in-depth analysis for the different test-cases is provided in the following paragraphs.

Unfortunately, the variability of the network for the 32 and 48 processes cases also has a bearing on the verification runs. Although we produced large data sets (700 points during each verification run for each implementation), the standard deviation remains rather large when we compute the average execution times $t(i) = 1/3 \cdot \sum_{r=1}^{3} t_r(i)$ for each implementa-

tion i over the three verification runs r, where $t_r(i)$ denotes the average execution time for each implementation $1/700 \cdot \sum_{k=1}^{700} t_r(i,k)$ within one verification run. As a consequence, the ranking obtained from the averaged execution times may differ significantly from the ranking obtained from each of the verification runs due to the errors in measurement. We are no longer able to refer to the former one as the only "correct" one, but a classification in fast, fair and slow implementations is of course still possible. So, instead comparing the evaluation methods to a fixed ranking, we divide the implementations into categories based on the averaged execution times $t(i)$ (with somewhat arbitrary bounds), and analyze if the evaluation methods roughly follow this categorization or not.

In the 32 processes case, we use four categories, cat. 1 with times ranging from 233 to 236s, cat. 2 from 243 to 247s, cat. 3 from 251 to 256s and cat. 4 from 260 to 262. The standard interquartile range method produces in all three runs inconsistent rankings and mixes randomly good and bad implementations. The other methods all manage to select methods from the first category in places one to three. An interesting detail is, that a cat. 1 implementation was judged quite bad by all methods. Its measurements had two clusters which could be regarded as outliers. Since one contained a regular pattern, and the second had too many data points, one would not tempt to remove them as did the evaluation methods. However, these clusters did not influence the total performance and should have been discarded.

The 32 processes test cases also provide an excellent motivation for the usage of data filtering at all. In case not data filtering is being applied on the input data for the runtime decision logic, the runtime selection logic would have chosen twice implementations, which are considered to belong into the 'slow' group and once an implementation which has been categorized as 'fair'.

For the 48 processes runs, the evaluations for the first and the last two runs differ significantly. During the first run, all four methods were experiencing problems with the correct selection of the best-performing implementation and a correct internal ranking. The data was split into four categories: cat. 1 with average execution times from 127 to 132s, cat. 2 with those around 137, cat. 3 with those from 145 to 146s and finally cat. 4 with those over 200s. The results displayed in figure 3 depict the ordered average execution times of the verification runs for each implementation in the first column. The remaining columns list the estimated mean of the measured communication times for each run and every method. Every entry is color-coded: cat. 1 implementations have a white background, cat. 2 implementations a light grey one, cat. 3 implementations a dark grey one and cat. 4 implementations a black one.

The standard method, being quite conservative in removing measurements, does well, obtaining roughly the optimal ordering except for ranking a specific cat. 4 implementation in second place or third place. This cat. 4 implementation is the same which is rated in second place by cluster analysis. Looking more detailed into the measurements, one might be tempted to remove a few sporadic measurements. Apparently, these 'outliers' do influence the performance. The heuristic approach selects for the first run at an outlier ratio of 20% a cat. 4 implementation in first place, since a large cluster of data points with values between 205 and 208s is removed together with a number of true outliers. With only 10% of accepted outliers the heuristic approach would have selected a cat. 1 implementation. In the following two runs, the heuristic approach achieves good results, conserving nearly the optimal order for the first five places. The cluster analysis chooses a cat. 1 implementation in first place, but selections from cat. 4 follow in between. The method using robust statistics performs worst in this scenario. It judges three different cat. 4 implementations as well-performing. In order to improve the prediction quality, we tried to estimate ν from the data set instead of using $\nu = 4$. However, this change had only little influence on the ordering. The false ratings might originate from the distinct clusters which now appear at greater distances. They are down-weighted because of the small Mahalanobis distances and not treated correctly.

C. Cacau GE

The third set of tests are executed on the *cacau* cluster at High Performance Computing Center of Stuttgart (HLRS), Germany. The cluster consists of 200 dual processor 3.2 GHz EM64T processors, connected by a 4xInfiniBand network interconnect. For our tests, however, we utilized the secondary network of the cluster, namely a hierarchical Gigabit Ethernet network. A total of six 48-port switches are used to connect the nodes, each 48-port switch has four links to the upper level 24 port Gigabit Ethernet switch. Thus, this network has a 12:1 blocking factor. We have executed tests using 64 processors on 64 nodes in order to ensure that communication between the processes has to use two or more of the 48-port switches.

This hardware setup exposes tremendous differences in the performance of various implementations. The best performing implementation takes 110.85 seconds for 700 iterations, while the worst one needs nearly twice the time of that with 210.64 seconds. Due to the large differences in the execution times, all four methods identified correctly the best implementation in all three test runs. Interestingly, the heuristic approach had to fall back to the unfiltered average for all implementations, since it considered too many data points to be outliers. At the same time, neither the standard interquartile formula nor the cluster analysis removed any data points, but still came to the same conclusion.

V. Conclusion

This paper discussed four different approaches for handling outliers in parallel performance measurements, namely a standard approach using interquartile ranges, a heuristic derived from the trimmed mean value, an approach based on cluster analysis and finally a method using robust statistics. Although using fundamentally different approaches, all four methods in our evaluation have shown the capability to handle outliers in most scenarios. In case the differences between the average

verification	standard			heuristic			cluster			robust		
average	run 1	run 2	run 3	run 1	run 2	run 3	run 1	run 2	run 3	run 1	run 2	run 3
126.95	3,855	5,260	3,842	3,223	3,818	3,752	3,859	3,857	3,752	3,790	3,654	3,808
131.94	4,919	5,825	3,908	3,397	3,932	3,842	3,869	3,915	3,817	3,850	3,779	3,851
132.15	5,436	9,325	4,643	3,855	3,988	3,902	4,152	5,205	3,830	3,857	3,800	3,852
136.77	5,647	9,857	4,740	4,018	4,082	3,908	4,621	5,229	3,875	3,861	3,816	3,916
137.02	8,708	10,990	5,448	4,039	4,123	3,943	4,850	5,373	4,959	3,877	3,877	3,971
145.22	9,116	13,786	5,682	4,069	4,150	3,952	4,917	5,512	4,971	3,933	4,137	4,086
146.32	9,716	17,437	9,976	4,078	5,927	5,448	5,252	5,607	5,020	4,543	4,226	5,512
201.95	9,820	17,632	14,266	4,102	6,463	5,682	5,399	5,829	5,113	4,917	4,596	6,050
203.03	23,199	29,148	27,496	5,427	28,041	27,375	5,538	5,927	5,360	4,944	5,428	6,982
221.49	31,080	37,073	27,510	5,647	35,430	27,447	5,677	6,493	5,671	5,407	5,596	7,687
237.64	37,008	41,867	36,619	35,658	41,867	36,619	5,687	7,806	5,723	5,643	5,714	8,707
237.85	39,065	44,300	41,819	39,065	43,092	41,798	5,743	12,110	6,944	6,591	6,980	9,606

Fig. 3. Estimated means of the methods for each run vs. the correct ranking for the 48 processes test case over Gigabit Ethernet on *shark*.

execution times of different implementations are fairly large, all four methods determined uniformly the fastest implementation. The approach applying techniques from robust statistics does show a superior behavior in case the performance data has a low variance, and the main task is to determine subtle differences between the implementations. On the other hand, the same approach had significant problems in case the data was widely distributed in multiple, large data clusters. The approach using cluster analysis did have similar problems in the latter scenario, mainly due to the removal of too many clusters. The standard approach using interquartile range did not show an outstanding behavior for any of the analyzed scenarios, but did not have a major break down either. The heuristic based on the trimmed mean value did show the most reliable performance, delivering the optimal or close to optimal decisions in most of the analyzed scenarios.

The major advantage of the interquartile range method and of the heuristic approach are that their computational complexity is low compared to the cluster analysis and robust statistics. This is especially important for a library which has to perform the operations outlined in this paper at application runtime.

The future work in this area includes the development of a meta-outlier detection method which incorporates all four techniques discussed in this paper. The goal is to determine the method leading to the most accurate decision based on a few characteristics of the data set, such as number of clusters or the maximum distance between data points.

ACKNOWLEDGMENTS

This research was funded by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems. We would like to thank Shishir Shah for the fruitful discussions and Holger Berger as well as Danny Sternkopf for their support at the HLRS.

REFERENCES

[1] R. C. Whaley and A. Petite, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, 2005.

[2] J. Pjesivac-Grbovic, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "MPI Collective Algorithm Selection and Quadtree Encoding," *Parallel Computing*, vol. 33, pp. 613–623, 2007.

[3] J. J. Evans, C. S. Hood, and W. D. Gropp, "Exploring the Relationship Between Parallel Application Run-Time Variability and Network Performance," in *Proceedings of the Workshop on High-Speed Local Networks (HSLN), IEEE Conference on Local Computer Networks (LCN)*, October 2003, pp. 538–547.

[4] E. Gabriel and S. Huang, "Runtime optimization of application level communication patterns," in *Proceedings of the 2007 International Parallel and Distributed Processing Symposium, 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2007, p. 185.

[5] D. Hawkins, *Identification of Outliers*. Reading, London: Chapman and Hall, 1980.

[6] F. Petrini, D. J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 55.

[7] "Thunderbird cluster at Sandia National Laboratories." [Online]. Available: http://tbirdweb.sandia.gov

[8] A. M. Mood, F. A. Graybill, and D. C. Boes, *Introduction to the theory of statistics*. McGraw-Hill Inc., 1974.

[9] H. C. Romesburg, *Cluster Analysis for Researchers*. North Carolina: Lulu Press, 2004.

[10] M. J. L. de Hoon, S. Imoto, J. Nolan, and S. Miyano, "Open source clustering software," *Bioinformatics*, vol. 20, pp. 1453–1454, 2004.

[11] P. J. Huber, *Robust statistics*. New York: John Wiley & Sons, 1981, republished 2003.

[12] F. R. Hampel, E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel, *Robust Statistics – The Approach Based on Influence Functions*. New York: Wiley, 1986, republished 2005.

[13] R. A. Maronna, D. R. Martin, and V. J. Yohai, *Robust Statistics: Theory and Methods*, ser. Wiley Series in Probability and Statistics. New York: Wiley, 2006.

[14] P. J. Rousseeuw and A. M. Leroy, *Robust Regression and Outlier Detection*. New York: John Wiley & Sons, 1987, republished 2003.

[15] K. L. Lange, R. J. A. Little, and J. M. G. Taylor, "Robust statistical modeling using the t distribution," *Journal of the American Statistical Association*, vol. 84, pp. 881–890, 1989.

[16] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal*, vol. 7, pp. 308–315, 1965.

[17] R. W. Freund, "A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 470–482, 1993.

[18] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.