Experiment Management and Analysis with *perfbase*

Joachim Worringen

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
joachim@ccrl-nece.de
http://www.ccrl-nece.de

Abstract

Achieving the desired performance with application software, middleware or operating system components on a parallel computer like a cluster is a complex task. Typically, a high-dimensional parameter space has to be reduced to a small number of core parameter which influence the performance most significantly, but still a large number of experiments is necessary to determine the optimal performance. Keeping track of these experiments to derive the correct conclusions is a major task. This paper presents *perfbase*, a set of frontend tools and an SQL database as backend, which together form a system for the management and analysis of the output of experiments. In this context, an experiment is an execution of an application or library on a computer system. The output of such an experiment are one or more text files containing information on the execution of the application. This output is the input for *perfbase* which extracts specified information to store it in the database and make it available for management and analysis purposes in a consistent, fast and flexible manner.

1 Introduction

Each time that a performance-critical part of a software system is changed, it is necessary to reevaluate the delivered performance to rate the effect of the change. For this purpose, a benchmark or application is run. Then, its output or externally measured data is typically transformed into a presentable form like a chart or a table for analysis. The same is true for testing correctness of a software. This can be considered a special case of a performance test with only a single result value, namely the number of errors that occurred.

The development of an MPI library is a good example for this. The MPI library consists of many independent subsystems which have to be changed for different reasons: to deliver better performance on the same platform; to change the behaviour of the software, i.e. to comply with given resource usage restrictions; to adapt the software to a new hardware platform or to avoid problems with a platform already supported or to implement new subsystems that perform new tasks.

An example from another research area is the price calculation of stock options [13]. To find the right model and parameters, a large number of parameterised simulation runs is

required. The results of these runs, which often depend on halve a dozen of parameters, need to be stored for further evaluation which compares different simulation results based on the parameters used.

The way that the described performance evaluation and result analysis is done is to store output data of the experiments (a run of a benchmark, application or simulation) in individual files. The format of the output data is typically ASCII text, as this is the easiest and most portable format to be generated from within any kind of software. Another advantage of ASCII files is that the content is still usable even when parts of the file are corrupted. All files are then organised by their name and sorted into directories. From these files, the data is in some way, either using some custom scripts or by manually copying the required data, fed into software for visualisation or analysis.

This naive, but widespread approach has a number of problems:

- Translating the benchmark output into the presentable form is often a tedious, manual task as the data needs to be extracted and transferred between different software tools.

- It is complex and error-prone to manage all results in a (big) number of files (of a certain type, usually text). It is not easy to discover which dimensions of the parameter space have not yet been measured precisely enough, or which one may need a closer look due to irregular results.

- Access to the output files is often difficult for people different from the one who performed the experiments that need the information contained within. Even with access, the way the output files are organised and the format of the file content can be hard to understand[1].

- Because of this complexity, the performance measurements are often limited in the range of the applicable test parameters, and in the number of samples taken for a certain set of test parameters. This leads to results of limited usefulness due to the unknown statistical variance in the results. This is especially true for application areas with a significant variation of results like testing of I/O performance, testing performance on a non dedicated system or running simulations which include error estimation.

- It is useful to track the performance development over a longer period of time or multiple software and hardware revisions. Typically, an analysis is done for a series of experiments at one point of time. Plotting and analysing data over time requires substantial additional efforts and is thus only performed by looking at the individual analysis – if the past analysis' are still available and have been generated in a way that allows a comparison.

- Generally, it is hard to perform a more complex analysis, like filtering the data by certain parameters or result values. Because the user knows this in advance, he will limit the parameter range of experiments to what be can processed with the naive approach he uses. This potentially leads to wrong conclusions.

[1] After some time, this problem typically applies to the original researcher, too!

In this paper, we describe *perfbase*, which is an open-source software package for experiment management and analysis. It is designed to resolve these problems without enforcing a change in the way the data is acquired. The latter requirement was to be fulfilled because we see that in many situations and environments where software developers have to work, any other way than printing ASCII formatted information to files is not possible or would involve a too large amount of work to set up the necessary framework[2]. In Section 2, we look at the characteristics of existing tools and frameworks for experiment management and analysis. The work flow concept and user-visible design of *perfbase* is presented in Section 3, with aspects of the internal architecture and implementation covered in Section 4. Section 5 presents an application example to illustrate the way *perfbase* can be used.

2 Related Work

While a number of academic projects deal with experiment management, analysis or both, only very few specialised solutions are offered commercially. Because of their specialisation to biology and chemistry or psychology, we do not consider these solutions here.

Commercial Statistics and Analysis Software Origin [9] and Matlab [5] are commercially available software packages for statistics and general math. They too can import ASCII data, but are limited in the flexibility concerning the specification which data in the input file is to be assigned to which variable of the experiment. They can exchange data with databases, but do not have a concept of an *experiment* and *runs* which contribute data to an experiment. This means that it might be possible to implement a system like *perfbase* on top of these packages, but they don't provide such functionality themselves. However, it is possible to have *perfbase* generate input data for the powerful statistical processing capabilities of these software packages.

ZOO The *Desktop Experiment Management Environment* ZOO [4] is an approach similar to *perfbase*. However, it has a different level of abstraction concerning a *process*, *data* and other objects, and features its own database server. The related software is is inaccessible and no longer maintained.

ZENTURIO ZENTURIO [10] is part of the ASKALON project and uses a directive-based language named ZEN to extract data from applications. ZENTURIO is a web-based grid-middleware system to run and control experiments and to visualise the performance data. Due to this grid-oriented implementation, combined with the required source-level instrumentation (which in turn requires a compiler and run-time libraries), the whole system is difficult to set up and run. The software is not publicly available.

PPerfDB and PPerfGrid PPerfDB [2], and its successor PPerfGrid, are designed to compare performance metrics of different executions of large-scale parallel programs with

[2]At least, this is how the developer sees it. The fact that existing advanced tools and frameworks are *not* used widely can be interpreted as a non-optimal adaptation to the work flow.

different optimisation parameters or on different machines/sites. This makes it similar to *perfbase*, but the technical approach is much more complex. To gather the performance data, it uses different techniques for tracing or (dynamic) probing of the application like Paradyn [6][3]. The current project PPerfGrid indicates that it is now using Grid technology for communication of the involved components. This does not simplify the usage of the software. Like ZENTURIO, the software is not publicly available.

CUBE CUBE, part of the KOJAK project [8], is a multi-experiment performance analysis tool. It applies a specific performance algebra to the trace information that maps the information from the trace data into the three dimensions performance metric, call path and process (or thread). This algebra is an extension of the model used by PPerfDB. Data can be explored along these dimensions in different ways. To compare multiple experiments, it is useful to see the performance difference between two experiments. Alternatively, the mean values of multiple experiments can be displayed. The trace data required for the analysis can be imported from the TAU [7] system. The source code is freely available.

PerfDMF PerfDMF [3] aims to provide a framework for performance data management. PerfDMF addresses objectives of performance tool integration, inter-operation, and reuse by providing common data storage and analysis infrastructure for parallel performance profiles. PerfDMF includes a relational database to store profile data from various proprietary (parallel) profile formats in a generic format. Upon this, it features an abstract profile query and analysis programming interface and a toolkit of commonly used utilities for building and extending performance analysis tools.

Assessment of the presented approaches All these approaches are complex as they strive to provide a maximal solution (in their sense). This implies the use of a range of methodologies: controlling the execution of the (parallel) applications, gathering of detailed trace data gained by (semi-)automatic instrumentation, transportation of this data between components using grid technologies, storage of the data either as files or within a database, and finally a way to describe the data to be able to visualise it in a structured way. Such maximal solutions imply a huge overhead in terms of setup complexity, compatibility requirements, storage requirements, processing time and analysis complexity. The existing solutions are difficult to deploy as they require services to be running and accessible, libraries and tools to be available right where the data is to be gathered. Therefore, they are targeted at stable production environments with administrators taking care of the installation, and users who know how to use these complex tools and libraries in the given environment with their specific application.

Many users don't require such an approach. Examples are application developers who know their code well and want to track the effect on execution time for modified algorithms, or system analysts who need to evaluate the performance of applications and benchmarks on different cluster platforms. There are various reasons for why the complex approaches, while looking comfortable once everything is running, often do not meet the user's requirements and thus are not used:

[3] In fact, the project leader of PPerfGrid used to work within the Paradyn project.

- The user does not have enough time to set up a complex system in an environment he is using for development right now, maybe on a temporary basis.
- The administrators of a system have not set up the system the user might need, and the user does or not have the required administrative rights.
- External communication is not possible, which hinders the deployment of grid-based solutions.
- The user does know exactly which data he needs to gather for analysis, and does not want the automatic system to gather data for all sorts of events, effectively hiding the data in which the user is interested.
- The system is not compatible with the user's type of software, or does not extract the required information.
- Changes to the output of the application are required, or the application needs to be instrumented on source level. In the case of automatic instrumentation, the instrumentation can change the runtime behaviour of the application, just not capture what the user wants to know, or captures so much data that it is hard for the user to manage this amount data and find the information he needs.

3 Workflow Concept and Design

There is one method of gathering and storing data which always works, independent from any external conditions: gathering data either inside the system to be analysed or with external tools, and printing the derived information as ASCII-formatted text to a file. The universal and fail-safe characteristic of this method, which is typically very simple to set up, is the reason why it is still the most intensively used one. This does not only apply to software written by individual users or groups, but also to all sorts of publicly available benchmarks, libraries and applications.

The drawback of this method is the fact that these output files are not well suited for further processing as the information output is often poorly formatted and not specifically tagged. This makes the process of feeding this data into analysis tools cumbersome and error-prone. If information belonging together is even distributed across multiple files, like raw performance numbers and information on the execution environment, the user often faces a serious problem.

The central idea within *perfbase* is the *experiment*. An experiment is the software, or more generally the system, to be evaluated. Typically, this is a benchmark or an application which generates arbitrary ASCII output. This software is executed within certain constraints, given by the *input parameters*, and delivers a number of *result values*. Each execution of the software is a *run* within the experiment, and is stored as a set of input parameters and result values. A parameter or a result may have constant content throughout the run (*unique occurrence*), or may contain a vector of content (*multiple occurrence*). Such vectors of parameters and results are typically related element-wise when they represent the columns of a table. Each tuple of vector elements is then called a *data set*.

The experiment is analysed by performing queries on all or only a subset of the runs. Such queries derive and process data from selected runs and output it in a definable format, either as a plot, raw text or binary data, or structured representation like XML or LaTeX tables.

3.1 Experiment Definition

The first step when working with *perfbase* is the definition of the experiment (see Fig. 5 for an example). Like all control files for *perfbase*, the definition is provided as an XML formatted text file which conforms to a *perfbase*-specific DTD (*Document Type Definition*). The most relevant part of the definition are the *input parameters* and *result values*. For each parameter and value, a description and a short synopsis can be given, together with the physical or logical unit of the data, and the type of data (like integer, float, text or other types). Additionally, some meta information on the experiment is required. This includes a description and synopsis, the authors name and affiliation, and the users that are allowed to import or query experiment data.

Because experiments evolve over time, the experiment definition can be changed by the user. Values and parameters can be added, modified or removed, the meta information can be changed and access rights can be revoked or granted to users.

3.2 Input Description and Data Import

Each execution of the experiment is called a *run* and generates data contained in an arbitrary number of ASCII files. This data needs to be imported into *perfbase* for further processing. An *input description*, which is again an XML formatted file, tells *perfbase* how to extract the required data for the input parameters and result values from these ASCII input files (an example is shown in Fig. 6). A number of means are provided to describe where to find the content for a parameter or a value: a *named location* matches a given string or a regular expression and use the text behind (or in front of) this match as content of the parameter or value. Within this assignment, *perfbase* considers the data type of the variable, and provides different means to specify which part of the text to assign. A *fixed location* retrieves content from a defined row and column in the text file. Data sets are retrieved via a *tabular location* which contains an arbitrary number of *tabular values*. The start of a table is defined by a match of a string or regular expression and possibly an offset. It is also possible to retrieve content from the name of an input file using a *filename location*. To provide a parameter or result independent from the data files, it can be defined via a *fixed value* either within the XML file or from the command line. For parameters which can not be retrieved from the input files directly, but need to be derived from other parameters, a *derived parameter* provides the means to express such an arithmetic relation. Finally, a single input file may contain data of multiple runs. The separation of these runs can be defined by a *run separator*.

The possible mapping of input files to *runs* in the experiment is shown in Fig. 1. The most simple case is a), where a single input file is parsed according to one input description, resulting in one new run. Case b) uses run separators to retrieve multiple runs from a single input file. If multiple files are parsed with a single input description, they will be processed independently and multiple runs are created (case c)). Finally, as depicted in case

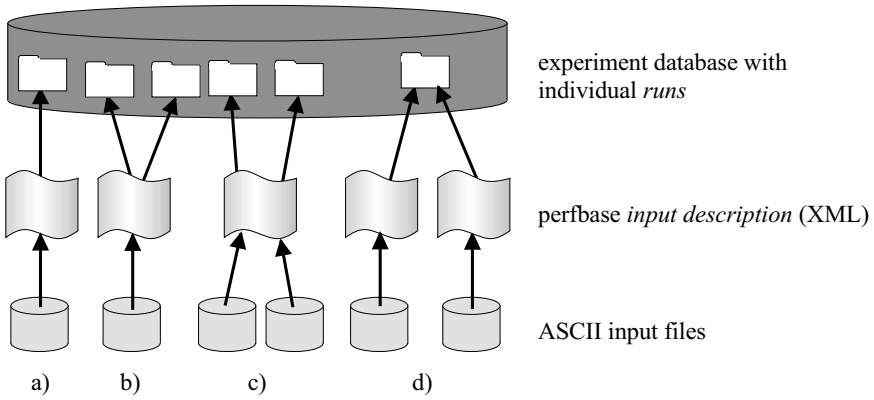


Figure 1: Possible mappings of input files to *runs*

d), processing multiple input files, each with one input description, will merge the data from all files into a single run. This mapping allows to collect outputs of different sources for a single run and process these without needing to merge them into a single input file.

For each of these means, *perfbase* uses meaningful default values and smart parsing to actually extract the content from the input files that the user intended to assign to the variables. Additionally, a range of additional XML attributes and elements can be used to successfully handle each variant of ASCII input files.

It happens that either an input description does not define content for all variables of an experiment, or that the input files do not provide content for all variables. Different approaches are possible to solve this problem: *perfbase* can use default values if such are defined for a variable (within the experiment definition). It is also possible to have a variable without content. However, the user may not choose either of these approaches if they could falsify the results for the case of corrupt or incomplete input files. In such cases, the user might either want to discard the data from such input files, or add the necessary information manually. The user can specify the desired behaviour of *perfbase* via command line switches of the respective *perfbase* command. This makes it possible to perform batch imports of a large number of input files without worrying about corrupt or incomplete experiment data. For such operations, it is also important that without explicit confirmation, importing data from the same input file more than once is not possible.

3.3 Query Specification

After a sufficient amount of runs has been performed for an experiment, the user wants to perform a query to gain knowledge from the data gathered. Again, the control file to perform a query is an XML formatted text file (see Fig. 7 for an example) and conforms to a DTD provided by *perfbase*. This flexible DTD allows for very powerful queries, with the result of the query being provided in one or more of multiple available output formats. The principle of how data is extracted from the database, processed via relations and operations, and formatted for the query output is shown in Fig. 2: four different elements, namely *source*, *operator*, *combiner* and *output* are connected by assigning the output of one element to be

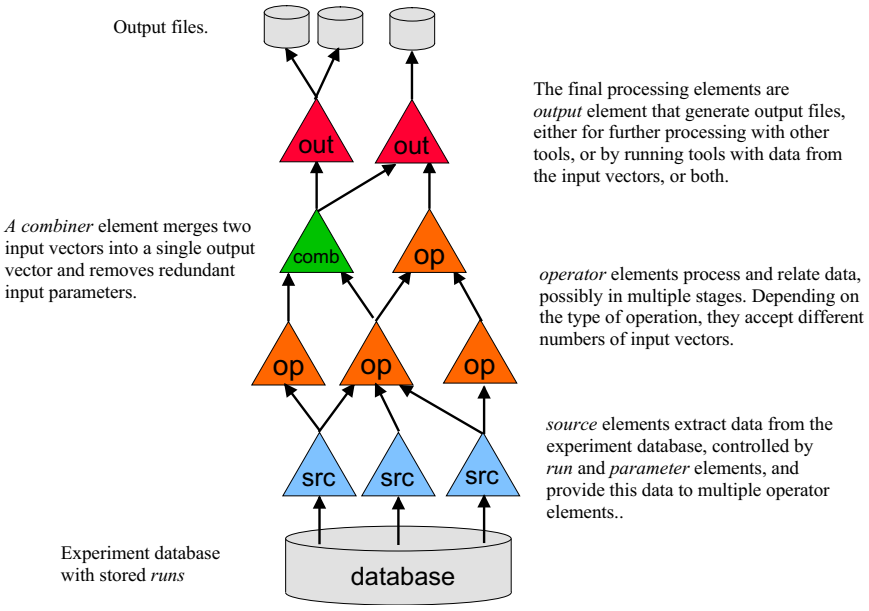


Figure 2: Possible relation of query elements within a query specification.

the input of another one. Within certain limits, these elements can be arbitrarily cascaded.

3.3.1 Source Element

A query may define an arbitrary number of instances of the *source* element. They retrieve data from the the database based on limiting properties of zero or more input parameters or the time stamp or index of a run, all given by *parameter* and *run* elements of the query specification. The output of a *source* element is a vector of data tuples which match the specified criteria. Each data tuple consists of the input parameters by which the database access was filtered and the result values that were specified in the *source* definition. It is typical that multiple tuples feature the same set of input parameters, but different result values. Along with the content of a variable in the output vector comes meta information of the variable.

3.3.2 Operator Element

The operator element is the element that actually performs operations and relations on the elements of a tuple. Available operator types are statistical functions (*avg*, *stddev*, *variance*, *count*) which can be applied on exactly one input vector, general reductions (*min*, *max*, *prod*) and arithmetic operations (*eval* for arbitrary function definitions, *scale* and *offset* for linear functions) which can be applied to any number of input vectors, and a set of operations that only can be applied to exactly two input vectors (*diff* and *div* for subtraction and division, and *percentof*, *above* and *below* for relative comparisons).

There are different modes of operation which are automatically differentiated by the number and type of the input vectors and the type of the operator. E.g., if the input vector

of a reduction operator stems from a *source* element, it will apply the reduction operation to multiple result values with identical sets of input parameters (*data set aggregation*)[4]. If the same operator is applied to a single input vector that stems from another non-source element, it will reduce all elements of the vector into a single element. Finally, if more than one input vector is assigned to this operator, it will perform an element-wise reduction on these vectors into a single output vector. This illustrates that *perfbase* is designed to work in a meaningful way without requiring the user to specify every detail.

3.3.3 Combiner Element

A *combiner* element is used to merge two input vectors into one output vector. All result values of the two input vectors are passed to the new output vector. Duplicate input parameters (parameters that exist in both input vectors) are removed by default. Combiners are sometimes required to match output vectors to the requirements of an operator's input vector.

3.3.4 Output Element

The output element generates arbitrarily formatted output from its input vectors. Currently implemented output formats are input files for the Gnuplot plotting program, supporting a variety of plotting styles and direct control of Gnuplot, and raw ASCII tables of data. Planned output formats include LaTeX tables, XML tables (i.e. for import into spreadsheet software like MS Excel), and other plotting tools like Grace or OpenDX.

3.4 Status Retrieval

To manage an experiment, it is possible to list the runs contained by different criteria, display the content of selected variables or meta information, or see the actual content of variables for a run. This allows to determine which parameter settings might still be missing for a parameter sweep.

4 Architecture and Implementation

perfbase is implemented as a collection of Python [12] scripts, launched via a sh script frontend. Currently, it consists of about 10.000 lines of original Python code, and it uses third-party Python modules to access a relational SQL database and to parse and generate XML files. No installation is required if the database server is found in the command path. It is invoked by providing the *perfbase* command (like `setup`, `input` or `query`) plus required arguments to the frontend script.

[4] In most cases, it makes sense to reduce the data from a *source* element via data set aggregation before processing it further.

4.1 Class Hierarchy

Classes are used in nearly all parts of *perfbase*: within the *input* command, all different ways to parse data from an input file are implemented in classes derived from the same base class, featuring a common set of methods with identical interfaces by which they are accessed from upper layers. The same is true for the functionality of the *query* command where the *source*, *operator*, *combiner* and *output* elements from the XML description are mapped onto respective class implementations based on a common base class. These clean interfaces make it easy for third-party developers or users to contribute to *perfbase* or adapt it for their special requirements.

4.2 Database Access

perfbase stores all persistent data in an SQL database. We chose the PostgreSQL database server [1] for this purpose as it is freely available on different platforms and very powerful. A user can either run a personal database server on his local workstation, or store his data on any connected PostgreSQL server. In this case, multiple users can access the same experiments in a protected manner. This is realised by having different user classes: *query users* which can only perform queries on an experiment, *input users* which can create new runs by importing data, and *admin users* which have full access to the database.

Each experiment database has some tables for meta information and one table for parameters and results with a unique occurrence per run. These tables are created during the initialisation of the experiment. For each new run, one table is created which contains the tabular data. The most database accesses occur for the *query* operations as the query elements communicate through temporary tables of the experiment database. This allows to use SQL database functionality for many of the operators, which results in better performance than to process the data within a Python script. The input vectors are read from other elements output tables, and an element can perform SQL operations on a single or multiple of these input tables and store the resulting data in a new temporary table. This means, each query element stores its output vector into its own temporary table. A reference to this table (its name) is passed on to the element by which it was invoked.

4.3 Parallelisation Potential

perfbase is not only well suited to track and evaluate performance data for typical cluster applications, but can itself benefit from running on a cluster. The most time-consuming and most frequently performed operation with *perfbase* is the *query* operation. Complex queries with multiple stages of operators take several seconds to complete, and it would make working with *perfbase* a more interactive experience if this delay could be reduced by some factor.

As described above, when performing a query, the individual elements of a query communicate through temporary database tables. Normally, all these tables are created within a single database server. On a cluster, these operations can be distributed across multiple nodes, each running an independent database server. In such a setup, the output vector of each query element is stored on the node on which the query element(s) run which use this

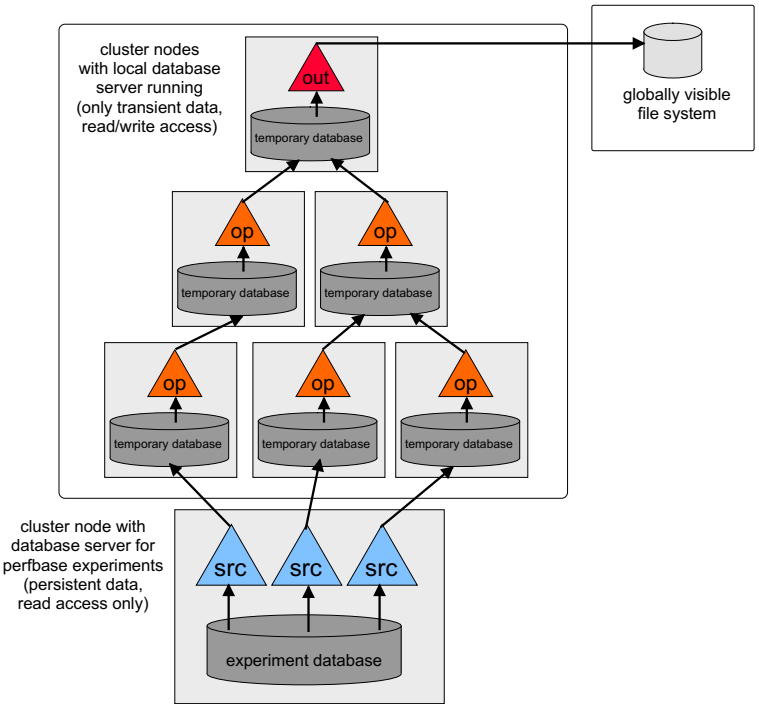


Figure 3: Parallelisation of *perfbase* across a cluster.

data for their input vector. The access to the database servers on remote nodes is performed via sockets, possible using a high-speed interconnection network. This principle is illustrated in Fig. 3.

The cluster or frontend node which runs the database server with the persistent experiment data might be at risk of becoming a bottleneck as all processing is rooted there. However, we profiled the *perfbase* query command and could see that in fact, the fraction of time spent within the source elements is typically only about 10%. This fraction decreases with increasing complexity of the query, and can be further reduced if performed concurrently, even on a single (SMP) server. This is understandable as the source elements do only perform simple read acccess on the shared database tables, and write data into independent temporary tables.

In contrast, the operator elements perform more complex queries, possibly across multiple tables with additional temporary tables before they finally store the results in the dedicated temporary table. We therefore do not expect the source elements to create a bottleneck. Otherwise, it would still be possible to replicate the experiment database onto more nodes to distribute the load. Other interesting problems with this approach are the distribution of the query elements across the cluster nodes: a 1:1 mapping is not efficient as once an element has performed its query, it will sit idle. This means, the number of cluster nodes that can be used efficiently is limited to the effective degree of parallelism in the query processing. However, for parameter sweeps, this degree can be significant, making a parallelisation worthwhile.

5 Application Example: MPI-IO Benchmarking

A current example of the application of *perfbase* is our continuous development of the MPI-IO part of an MPI library. This work includes running benchmarks which perform I/O operations. Each platform, and each site, has a different I/O environment. Additionally, I/O benchmarks feature a much higher variance in the results than message-passing benchmarks as the I/O system behaves less deterministic and is often a shared resource, even if the nodes running the benchmarks are dedicated. All this makes it necessary to run a lot of tests to get statistically valid results.

One of the MPI-IO benchmarks we use is *b_eff_io* [11]. It creates an output file (see Fig. 4) which gives information on the benchmark setup and summarises the results of a large range of MPI-IO operations.

The *perfbase* experiment related to this benchmark will contain most of the information from the *b_eff_io* output file either as an input parameter or a result value. We might need additional information not found in the output file, like file system type and configuration, or the number of nodes on which the application did run. Such information can be encoded in the filename of the output file, or be provided by additional input files. Fig. 5 shows an excerpt of the experiment definition which contains all variables for this example.

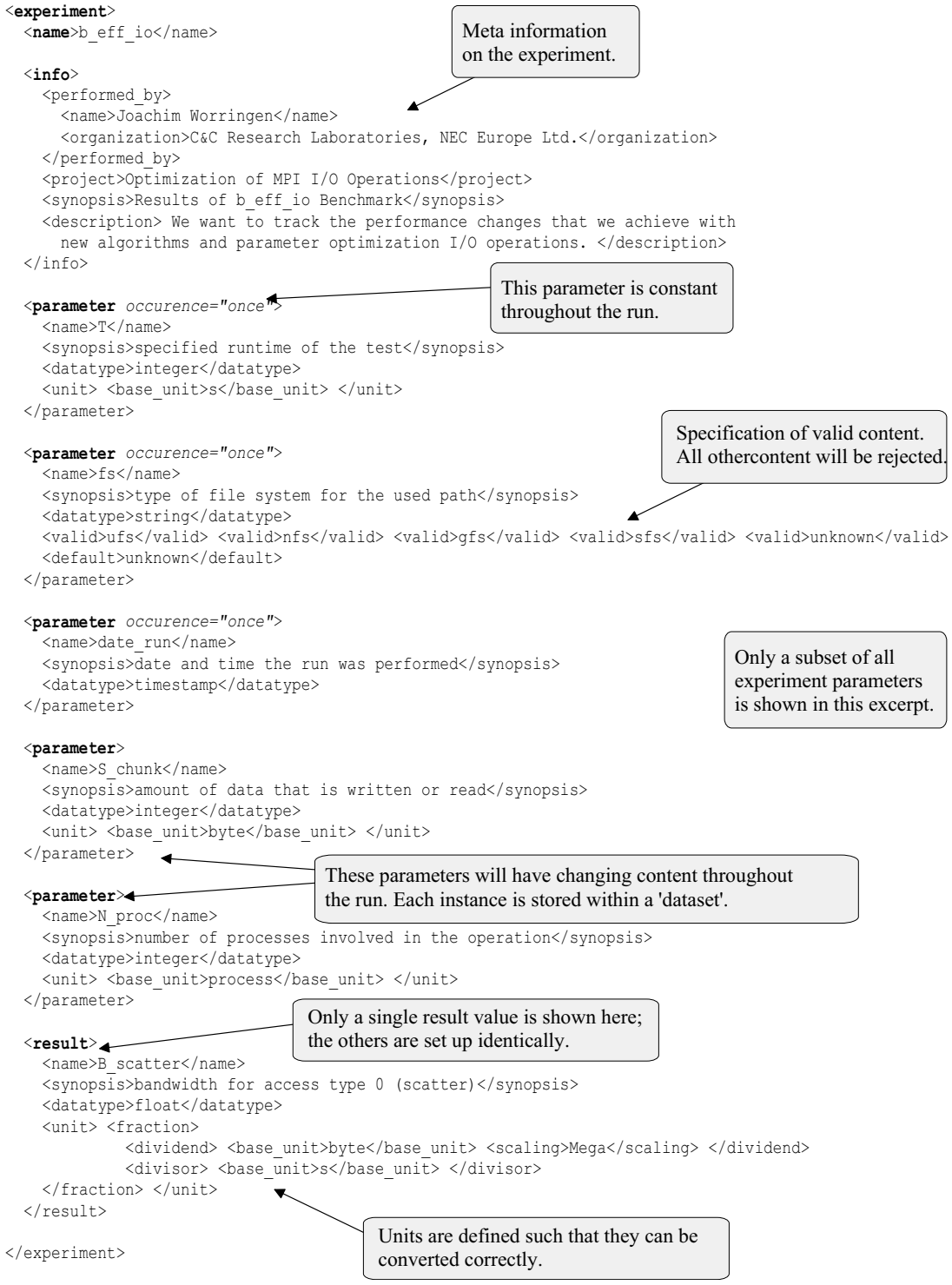
The input description shown in Fig. 6 lets *perfbase* read all variables from *b_eff_io*'s output file (and its name). It parses the filename for the name of the file system used, locates a number of input parameters via keyword matching, and finally parses the data tables with additional input parameters and the result values.

For this simple example, we want to evaluate the performance effect of a new list-less technique for non-contiguous I/O [14]. We ran *b_eff_io* on our cluster for a number of times in different configurations concerning the number of nodes and processes and the file system used. The new technique is enabled by default, and we performed additional runs with the old technique enabled. We then made sure that we gathered a sufficient amount of data by having *perfbase* calculate the average and standard deviation (this query is not shown due to space restrictions), and in fact some configurations required additional runs to reduce the standard deviation. After this, we could perform the query that would show us the relative performance difference between the old and new technique for all related test cases of *b_eff_io*. We chose the maximum value over all runs, and let *perfbase* create a bar chart from the derived numbers (see Fig. 7).

One of the resulting bar charts, created through Gnuplot, is shown in Fig. 8. It should be noted that this chart is shown unedited as it was created by *perfbase*. All labels and the legend are derived from the experiment definition and the query specification shown above. This plot shows a scenario in which the new list-less technique is about 60% slower than the old list-based technique for large read accesses. In fact, this was a performance bug which we could then fix. Without *perfbase*, it would have been very hard to compare the large number of result files for different configurations. Moreover, manually locating a single case of such a performance problem has little meaning when thinking of possible reasons for a transient drop in I/O performance. Only the statistically solid information across multiple runs as provided by *perfbase* made it definite that the problem is with the software.

```
MEMORY_PER_PROCESSOR =  256 MBytes  [1MBytes = 1024*1024 bytes, 1MB = 1e6 bytes]
Maximum chunk size   =    2.000 MBytes

-N  4 T=10, MT=1024 MBytes -i list-based_io.info, -rewrite
PATH=/tmp, PREFIX=bio_T10_N4_listbased_ufs_grisu_run1
       system name : Linux
       hostname    : grisu0.ccrl-nece.de
       OS release  : 2.6.6
       OS version  : #1 SMP Tue Jun 22 14:37:05 CEST 2004
       machine     : i686
Date of measurement: Tue Nov 23 18:30:30 2004

Summary of file I/O bandwidth accumulated on   4 processes with  256 MByte/PE
------------------------------------------------------------------------------
 number pos chunk-   access   type=0   type=1   type=2   type=3   type=4
 of PEs     size (l) methode scatter  shared  separate segmened seg-coll
            [bytes]  methode  [MB/s]   [MB/s]   [MB/s]   [MB/s]   [MB/s]
 ------------------------------------------------------------------------

   4 PEs 1       32 write     35.504    0.924    2.046    8.078    1.537
   4 PEs 2     1024 write     59.088    0.011   38.085    4.149    3.860
   4 PEs 3     1032 write     60.846    4.489   29.165   46.070   26.685
   4 PEs 4    32768 write     57.678   32.370   79.422   78.025   75.847
   4 PEs 5    32776 write     47.860   29.212   80.488   78.455   68.962
   4 PEs 6  1048576 write     67.328   77.992   81.427   82.589   82.988
   4 PEs 7  1048584 write     50.938   44.520   84.160   74.681   78.649
   4 PEs 8  2097152 write     62.658   80.023   86.121   82.971   84.693
   4 PEs      total-write     58.579   48.697   77.944   77.036   73.113

   4 PEs 1       32 rewrite   47.341    1.456   20.764   19.918    1.719
   4 PEs 2     1024 rewrite   55.452    0.223   70.707   44.763   25.665
   4 PEs 3     1032 rewrite   64.561    6.714   84.172   82.667   37.562
   4 PEs 4    32768 rewrite   63.221   34.205   90.227  102.996   88.864
   4 PEs 5    32776 rewrite   66.642   32.040   85.841   84.367   72.504
   4 PEs 6  1048576 rewrite   70.583   82.269   93.732   90.947   95.950
   4 PEs 7  1048584 rewrite   61.124   72.735   94.227   93.300   82.767
   4 PEs 8  2097152 rewrite   67.095   85.119   88.854   88.774   89.671
   4 PEs      total-rewrite   63.841   61.525   89.142   89.221   81.972

   4 PEs 1       32 read      76.680    1.850   32.768   32.860    1.767
   4 PEs 2     1024 read     227.183   36.553  677.280  418.002   52.442
   4 PEs 3     1032 read     226.023   60.290  671.384  676.151   56.184
   4 PEs 4    32768 read     242.524 1020.334 1661.961 1670.895  914.146
   4 PEs 5    32776 read     241.514  992.418 1657.528 1669.052  917.171
   4 PEs 6  1048576 read     465.409 1056.584 1155.196 1198.228 1173.667
   4 PEs 7  1048584 read     478.095 1000.797 1168.338 1184.615 1088.690
   4 PEs 8  2097152 read     516.540 1102.414 1173.111 1193.911 1185.697
   4 PEs      total-read     300.390  811.358 1073.572 1088.574  575.428

This table shows all results, except pattern 2 (scatter, l=1MBytes, L=2MBytes):
  bw_pat2=   60.848 MB/s write,   63.429 MB/s rewrite,  235.483 MB/s read

weighted average bandwidth for write   :   65.658 MB/s on 4 processes
weighted average bandwidth for rewrite :   74.924 MB/s on 4 processes
weighted average bandwidth for read    :  691.619 MB/s on 4 processes

b_eff_io of these measurements =  214.516 MB/s on 4 processes with 256 MByte/PE and scheduled time=0.2 min

Maximum over all number of PEs
 b_eff_io =  214.516 MB/s on 4 processes with 256 MByte/PE, scheduled time=0.2 Min, on Linux grisu0.ccrl-
nece.de 2.6.6 #1 SMP Tue Jun 22 14:37:05 CEST 2004 i686, NOT VALID (see above)
```

Figure 4: Excerpt from summarising output file of *b_eff_io* benchmark.




```
<experiment>
  <name>b_eff_io</name>

  <info>
    <performed_by>
      <name>Joachim Worringen</name>
      <organization>C&C Research Laboratories, NEC Europe Ltd.</organization>
    </performed_by>
    <project>Optimization of MPI I/O Operations</project>
    <synopsis>Results of b_eff_io Benchmark</synopsis>
    <description> We want to track the performance changes that we achieve with
      new algorithms and parameter optimization I/O operations. </description>
  </info>

  <parameter occurence="once">
    <name>T</name>
    <synopsis>specified runtime of the test</synopsis>
    <datatype>integer</datatype>
    <unit> <base_unit>s</base_unit> </unit>
  </parameter>

  <parameter occurence="once">
    <name>fs</name>
    <synopsis>type of file system for the used path</synopsis>
    <datatype>string</datatype>
    <valid>ufs</valid> <valid>nfs</valid> <valid>gfs</valid> <valid>sfs</valid> <valid>unknown</valid>
    <default>unknown</default>
  </parameter>

  <parameter occurence="once">
    <name>date_run</name>
    <synopsis>date and time the run was performed</synopsis>
    <datatype>timestamp</datatype>
  </parameter>

  <parameter>
    <name>S_chunk</name>
    <synopsis>amount of data that is written or read</synopsis>
    <datatype>integer</datatype>
    <unit> <base_unit>byte</base_unit> </unit>
  </parameter>

  <parameter>
    <name>N_proc</name>
    <synopsis>number of processes involved in the operation</synopsis>
    <datatype>integer</datatype>
    <unit> <base_unit>process</base_unit> </unit>
  </parameter>

  <result>
    <name>B_scatter</name>
    <synopsis>bandwidth for access type 0 (scatter)</synopsis>
    <datatype>float</datatype>
    <unit> <fraction>
             <dividend> <base_unit>byte</base_unit> <scaling>Mega</scaling> </dividend>
             <divisor> <base_unit>s</base_unit> </divisor>
    </fraction> </unit>
  </result>

</experiment>
```

Figure 5: Excerpt from experiment definition for *b_eff_io* benchmark.

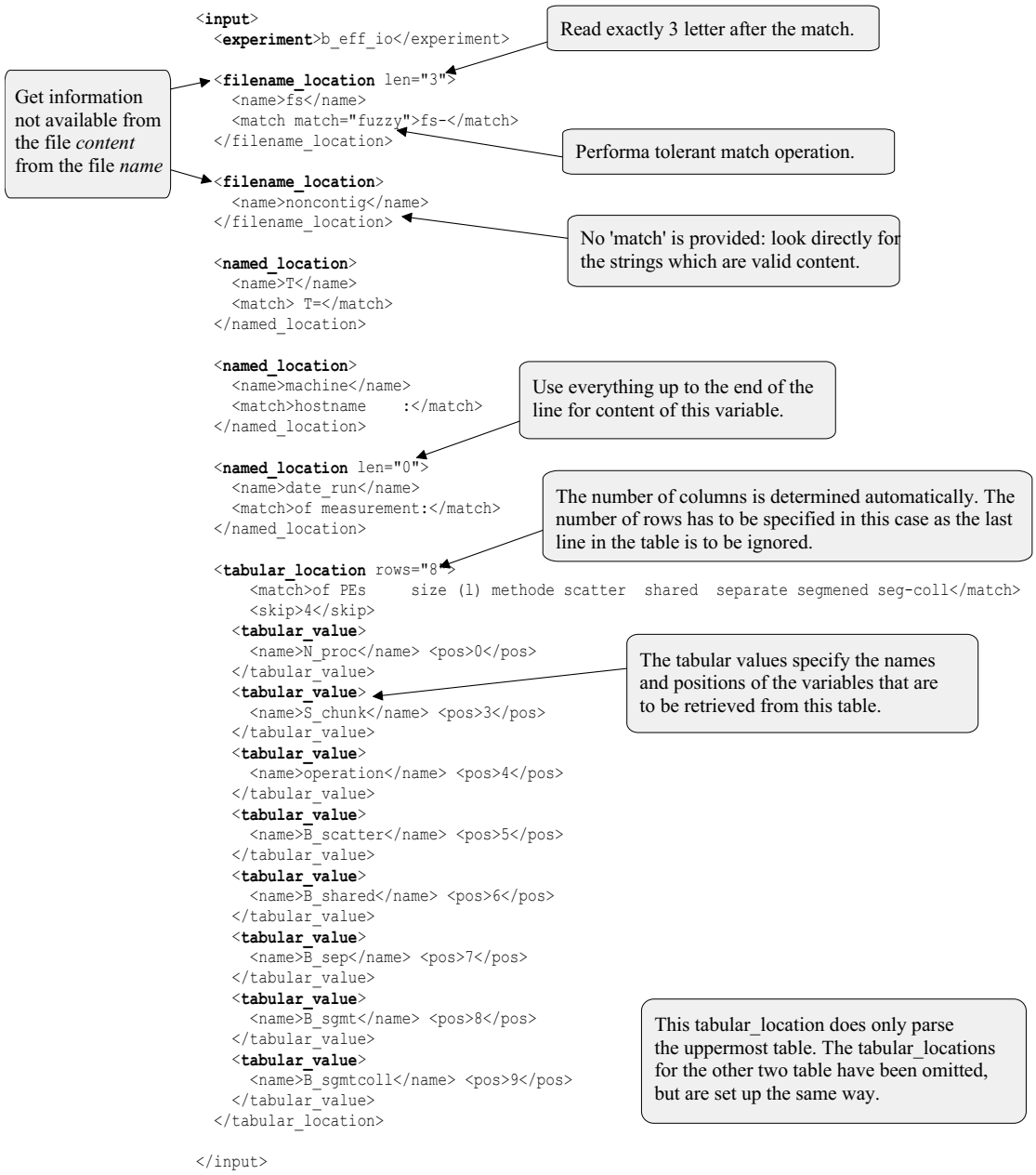


Figure 6: Excerpt from input description for *b_eff_io* experiment.

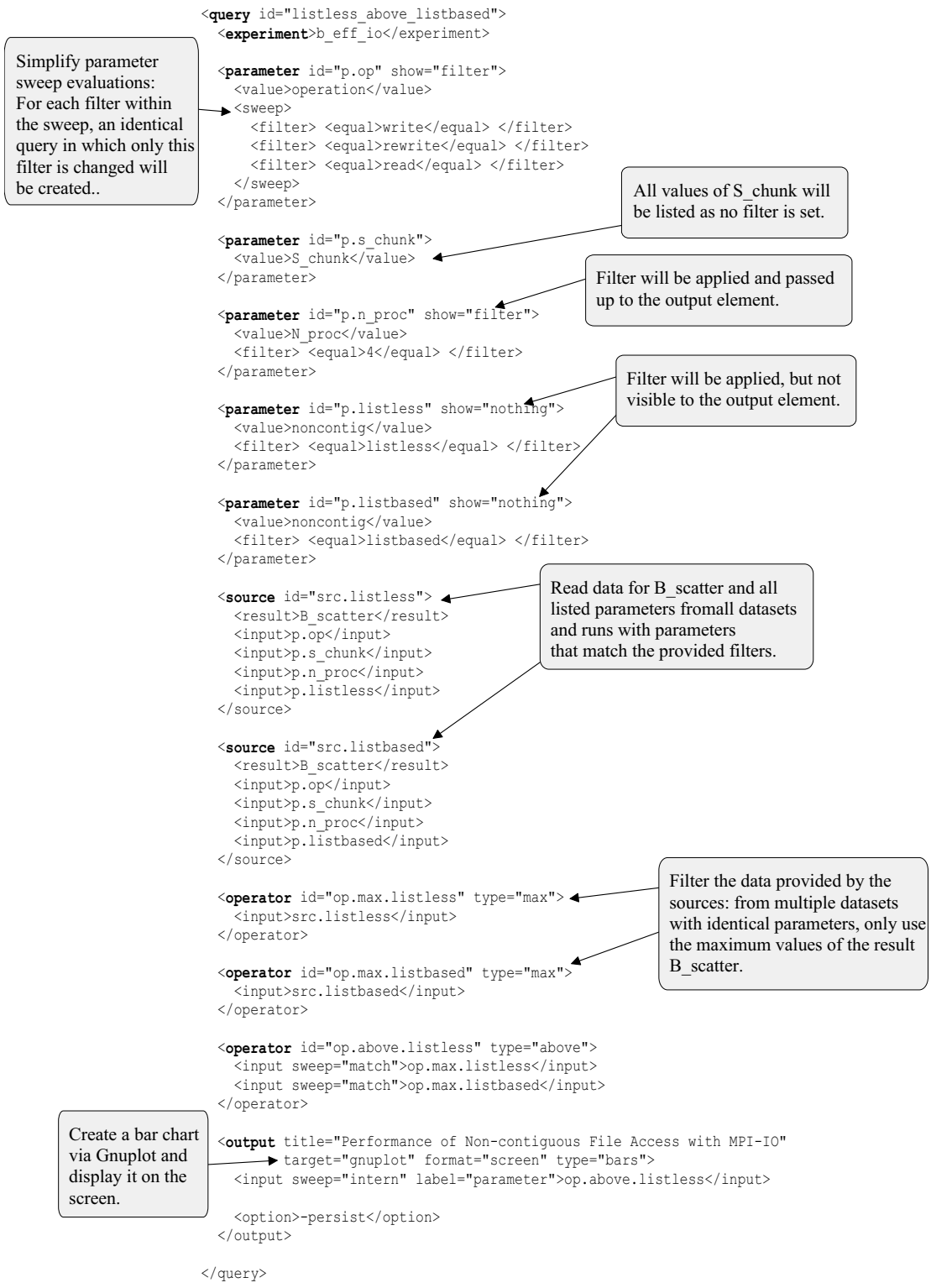


Figure 7: Query specification for *b_eff_io* experiment.

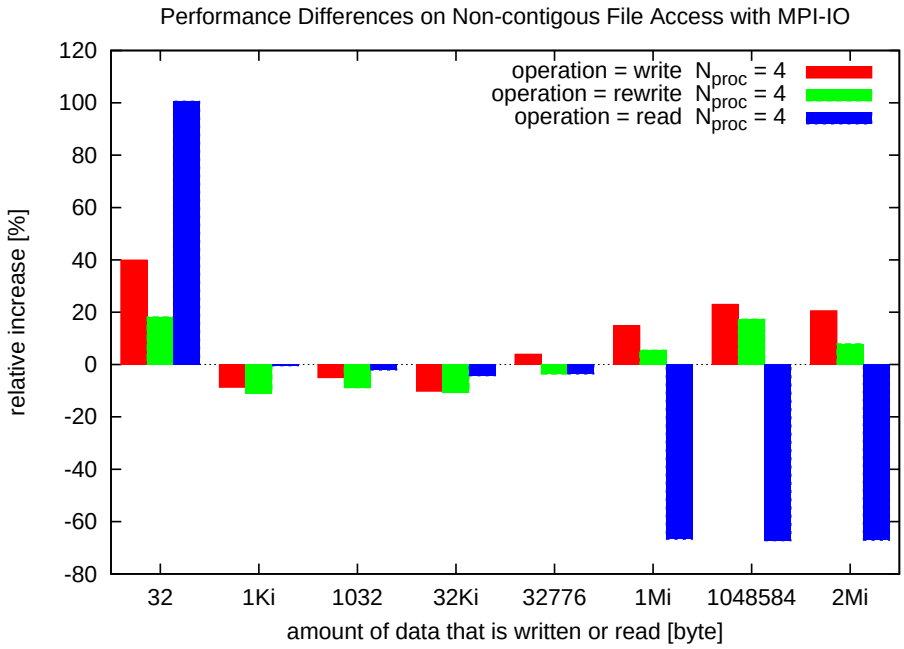


Figure 8: Relative difference of performance of two algorithms for non-contiguous I/O.

6 Summary and Outlook

We have presented *perfbase*, an open-source software package for the management and analysis of experiments based on the flexible processing of ASCII-formatted input files. To the best of our knowledge, *perfbase* features a unique approach which is oriented along the requirements of a very significant group of users and developers on clusters. This approach makes *perfbase* easy to deploy and use, yet powerful enough for even complex software environments, input files, and query requirements.

Next to its original purpose of performance analysis for HPC software and systems, *perfbase* can also be used for other tasks. A related application is the management and analysis of the output of test suites not only for performance, but also for correctness. Also, the use of *perfbase* is not limited to the operation of HPC clusters. The evaluation of sequential processing has similar problems, and *perfbase* could even be deployed in non-computer-science disciplines. However, with the parallel query processing in place, it will benefit from being run on a cluster.

Although *perfbase* is already a valuable tool, we have identified various directions for additional development and research: the parallelisation of the queries, the capability to analyse results automatically and only show suspicious or unusual results or deviations from previous runs, more operators and output formats, graphical frontends, processing of non-ASCII input files (like traces), and many more topics are on our list.

References

[1] E. Geschwinde and H.-J. Schönig. *PostgreSQL Developer's Handbook*. Sams, 2001.

[2] C. Hansen. Towards Comparative Profiling of Parallel Applications with PPerfDB. Master's thesis, PSU CS Department, 2001.

[3] K. A. Huck, A. D. Malony, R. Bell, and A. Morris. Design and Implementation of a Parallel Performance Data Management Framework. In *Proc. International Conference on Parallel Processing (ICPP 2005)*. IEEE Computer Society, 2005.

[4] Y. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A Desktop Experiment Management Environment. In *Proc. 22nd International VLDB Conference*, Bombay, India, September 1996.

[5] W. L. Martinez and A. R. Martinez. *Exploratory Data Analysis With Matlab (Computer Science and Data Analysis)*. Chapman & Hall/CRC, 2004.

[6] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, November 1995.

[7] B. Mohr, A. Malony, and J. Cuny. *Parallel Programming using C++*, chapter TAU. M.I.T. Press, 1996.

[8] S. Moore, F. Wolf, J. Dongarra, and B. Mohr. Improving Time to Solution with Automated Performance Analysis. In *Proc. Second Workshop on Productivity and Performance in High-End Computing (P-PHEC) at 11th International Symposium on High Performance Computer Architecture (HPCA-2005)*, San Francisco, February 2005.

[9] N.N. Originlab Website http://www.originlab.com.

[10] R. Prodan and T. Fahringer. ZENTURIO: A Grid Middleware-based Tool for Experiment Management of Parallel and Distributed Applications. *Journal of Parallel and Distributed Computing*, 6(64):693–707, 2004.

[11] R. Rabenseifner, A. E. Koniges, J.-P. Prost, and R. Hedges. *The Parallel Effective I/O Bandwidth Benchmark: b_eff_io*, chapter 4. Kogan Page Ltd., 2004.

[12] G. V. Rossum. *The Python Language Reference Manual*. Network Theory Ltd., 2003.

[13] J. Schumacher, U. Jaekel, and A. Basermann. Parallelization and Vectorization of Simulation Based Option Pricing Methods. In *Proc. International Conference on Computational Science and its Applications (ICCSA)*, Montreal, Canada, 2003.

[14] J. Worringen, J. L. Träff, and H. Ritzdorf. Fast parallel non-contiguous file access. In *Proc. of International Conference for High Performance Computing and Communications (SC2003)*, Phoenix, USA, November 2003.