Article

THE INTERNATIONAL JOURNAL of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
25(4) 428–439
© The Author(s) 2010
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342010387806
hpc.sagepub.com
SAGE

Trace-based performance analysis for the petascale simulation code FLASH

Heike Jagode[1], Andreas Knüpfer[2], Jack Dongarra[1], Matthias Jurenz[2], Matthias S Müller[2], and Wolfgang E Nagel[2]

Abstract

Performance analysis of applications on modern high-end petascale systems is increasingly challenging due to the rising complexity and quantity of the computing units. This paper presents a performance-analysis study using the Vampir performance-analysis tool suite, which examines application behavior as well as the fundamental system properties. This study was carried out on the Jaguar system at Oak Ridge National Laboratory, the fastest computer on the November 2009 Top500 list. We analyzed the FLASH simulation code that is designed to be scaled with tens of thousands of CPU cores, which means that using existing performance-analysis tools is very complex. The study reveals two classes of performance problems that are relevant for very high CPU counts: MPI communication and scalable I/O. For both, solutions are presented and verified. Finally, the paper proposes improvements and extensions for event tracing tools in order to allow scalability of the tools towards higher degrees of parallelism.

Keywords

collective I/O, collective MPI operations, event tracing, libNBC, Vampir

1 Introduction and background

Estimating achievable performance and scaling efficiencies in modern petascale systems is a complex task. Many of the scientific applications running on such high-end computing platforms are highly communication- as well as data-intensive. For example, the FLASH application is a highly parallel simulation with complex performance characteristics.

The performance-analysis tool suite Vampir is used to give deeper insights into performance and scalability problems of applications. It uses event tracing and post-mortem analysis to survey the runtime behavior for performance problems. This makes it challenging for highly parallel situations because it produces huge amounts of performance measurement data (Brunst, 2008; Jagode et al., 2009).

The performance evaluation of the FLASH software found two classes of performance issues that are relevant with very high CPU counts. The first class is related to inter-process communication and can be summarized as 'overly strict coupling of processes.' The second class is due to the massive and scalable I/O within the checkpointing mechanism where the interplay of the Lustre file system and the parallel I/O produces unnecessary delays. For both types of performance problems, solutions are presented that require only local modifications, not affecting the general structure of the code.

This paper is organized as follows: First we provide a brief description of the target system's features. This is followed by a summary of the performance-analysis tool suite Vampir. A brief outline of the FLASH code is provided at the end of the introduction and background section. In Sections 2 and 3 we provide extensive performance measurement and analysis results that were collected on the Cray XT4 system, followed by a discussion of the performance issues that were found, the proposed optimizations, and their outcomes. Section 4 discusses our experiences with the highly parallel application of the Vampir tools as well as future adaptations for such scenarios.

The paper ends with the conclusions and an outlook for future work.

1.1 The Cray XT4 system, Jaguar

We start with a short description of the relevant features of the Jaguar system, the fastest computer on the November

[1]The University of Tennessee, USA
[2]Technische Universität Dresden, Germany

Corresponding author:
Heike Jagode, The University of Tennessee, Suite 413, Claxton, Knoxville, TN 37996, USA
Email: jagode@eecs.utk.edu

2009 Top500 list.[1] The Jaguar system at Oak Ridge National Laboratory (ORNL) has evolved rapidly over the last several years. When this work was carried out, it was based on Cray XT4 hardware and utilized 7832 quad-core AMD Opteron processors with a clock frequency of 2.1 GHz and 8 GB of main memory (2 GB per core). At that time, Jaguar offered a theoretical peak performance of 260.2 Tflops/s and a sustained performance of 205 Tflops/s on Linpack.[2] The nodes were arranged in a three-dimensional torus topology of size $21 \times 16 \times 24$ with SeaStar2.

Jaguar had three Lustre file systems of which two had 72 Object Storage Targets (OST) and one had 144 OSTs (Larkin and Fahey, 2007). These file systems shared 72 physical Object Storage Servers (OSS). The theoretical peak performance of the I/O bandwidth was ~ 50 GB/s across all OSSes.

1.2 The Vampir performance-analysis suite

Before we show the detailed performance-analysis results, we will briefly introduce the main features of the performance-analysis suite Vampir (Visualization and Analysis of MPI Resources) that was used for this paper.

The Vampir suite consists of VampirTrace for instrumentation, monitoring, and recording as well as VampirServer for visualization and analysis (Brunst, 2008).[3.4] The event traces are stored in the *Open Trace Format* (OTF) (Knüpfer et al., 2006). The VampirTrace component supports a variety of performance features, for example MPI communication events, subroutine calls from user code, hardware performance counters, I/O events, memory allocation, and more (Knüpfer et al., 2008).[4] The VampirServer component implements a client/server model with a distributed server, which allows a very scalable interactive visualization for traces with over a thousand processes and an uncompressed size of up to 100 GB (Knüpfer et al., 2008; Brunst, 2008).

1.3 The FLASH application

The FLASH application is a modular, parallel AMR (Adaptive Mesh Refinement) simulation code, which computes general compressible flow problems for a large range of scenarios.[5] FLASH is a set of independent code units, put together with a Python language setup tool to create various applications. Most of the code is written in Fortran 90 and uses the Message-Passing Interface (MPI) library for inter-process communication. The PARAMESH library (MacNeice et al., 1999) is used for adaptive grids, placing resolution elements only where they are needed most. The Hierarchical Data Format, version 5 (HDF5), is used as the I/O library offering parallel I/O via MPI-IO (Yang and Koziol). For this study, the I/O due to checkpointing is most relevant, because it frequently writes huge amounts of data.

We looked at the three-dimensional simulation test case **WD_Def**, which is the deflagration phase of a gravitationally confined detonation mechanism for type Ia supernovae, a crucial astrophysical problem that has been extensively discussed in [Jordan et al. (2008). The **WD_Def** test case is generated as a weak scaling problem for up to 15,812 processors where the number of blocks remains approximately constant per computational thread.

2 MPI performance problems

The communication layer is a typical place to look for performance problems in parallel code. Although communication enables the parallel solution, it does not directly contribute to the solution of the original problem. If communication accounts for a substantial portion of the overall runtime, it implies that there is a performance problem.

Most of the time, communication delays are due to waiting for communicating peers. Usually, this becomes more severe as the degree of parallelism increases.

This symptom is indeed present in the FLASH application. Of course, it can easily be diagnosed on the basis of profiling, but the statistical nature of profiling makes it insufficient for detecting the cause of performance limitations and even more so for finding promising solutions.

In the following, three different performance problems are discussed, summarized as ‘overly strict coupling of processes’. The problems found are hotspots of **MPI_Sendreceive_replace** operations, hotspots of **MPI_Allreduce** operations, and unnecessary **MPI_Barrier** operations.

2.1 Hotspots of MPI_Sendrecv_replace calls

The first problem is a hotspot of **MPI_Sendrecv_replace** operations. It uses six successive calls, sending small to moderate amounts of data. Therefore, the single communication operations are latency bound and not bandwidth bound. Interestingly, it propagates delays between connected ranks, see Figure 1.

In the given implementation, successive messages cause a recognizable accumulation of the latency values. A convenient local solution is to replace this hotspot pattern with non-blocking communication calls. As there is no non-blocking version of **MPI_Sendrecv_replace** one can emulate the same behavior by non-blocking point-to-point communication operations **MPI_Irecv**, **MPI_Ssend**, and a consolidated final **MPI_Waitall** call. This would not produce a large benefit for a single **MPI_Sendrecv_replace** call but it will for a series of such calls, because for overlapping messages the latency values are no longer accumulated. Of course, it requires additional temporary storage, which is not critical for small and moderate data volumes.

The actual performance gain from this optimization is negligible at 1 to 2% at first. But together with the optimization described in Section 2.3 it will make a significant performance improvement.

The symptom of this performance limitation is easily detectable with profiling, because the accumulated runtime

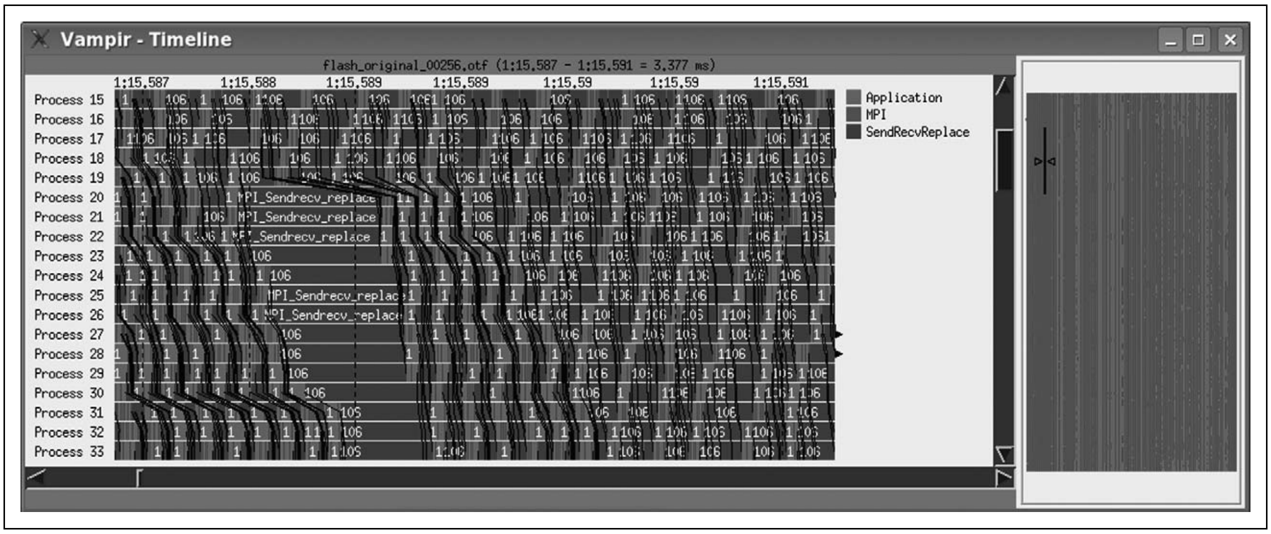

Figure 1. Original communication pattern of successive **MPI_Sendrecv_replace** calls. Message delays are propagated along the communication chain of consecutive ranks. See Figure 3 for an optimized alternative.

of **MPI_Sendrecv_replace** would stand out. Yet, neither the underlying cause nor the solution could be inferred from this fact alone. Plain profiling is completely incapable of providing any further details because all information is averaged over the complete runtime. With sophisticated profiling approaches like call-path profiling or phase profiling one could infer the suboptimal runtime behavior when studying the relevant source code. But this is tedious and time consuming especially if the analysis is not carried out by the author.

Only tracing allows convenient examination of the situation with all necessary information from one source. In particular, this includes the context of the calls to **MPI_Sendrecv_replace** within each rank as well as the concurrent situations in the neighbor ranks, see Figure 1. To keep the tracing overhead as small as possible and to provide a sufficient and manageable trace file, we recorded tracing information of the entire FLASH application using not more than 256 compute cores on Jaguar.

2.2 Hotspots of MPI_Allreduce calls

The most severe performance issue in the MPI communication used in FLASH is a hotspot of **MPI_Allreduce** operations. Again, there is a series of **MPI_Allreduce** operations with small to moderate data volumes for all MPI ranks. As above, the communication is latency bound instead of bandwidth bound.

In theory, one could also replace this section with a pattern of non-blocking point-to-point operations similar to the solution presented above. However, with **MPI_Allreduce** or with collective MPI operations in general, the number of point-to-point messages would grow dramatically with the number of ranks. This would make any replacement scheme more complicated. Furthermore, it would reduce performance portability since there is a high potential for producing severe performance issues. Decent MPI implementations introduce optimized communication patterns, for example tree-based reduction schemes and communication patterns adapted to the network topology. Imitating such behavior with point-to-point messages is very complicated or even impossible, because a specially adapted solution will not be generic and a generic solution will hardly be optimized for a given topology.

For this reason, the general advice to MPI users is to rely on collective communication whenever possible (Hoefler et al., 2007). Unfortunately, there are no non-blocking collective operations in the MPI standard. So it is impossible to combine a *non-blocking* scheme with a *collective* one, at least for now (Hoefler et al., 2007).

However, this fundamental lack of functionality has already been identified by the MPI Forum, the standardization organization for MPI. As the long term solution to the dilemma of *non-blocking* vs. *collective*, the upcoming MPI 3.0 standard will most likely contain a form of non-blocking collective operation. Currently, this topic is under discussion in the MPI Forum.[6]

As a temporary solution for this problem, libNBC can be used (Hoefler et al., 2007). It provides an implementation of non-blocking collective operations as an extension to the MPI 2.0 standard with an MPI-like interface. For the actual communication functionality, libNBC relies on non-blocking point-to-point operations of the platform's existing MPI library (Hoefler et al., 2007, 2008). Therefore, it is able to incorporate improved communication patterns but currently does not directly adapt to the underlying network topology (compare above).

Still, the FLASH application gets a significant performance improvement with this approach. This is mainly due to the overlapping technique of the successive **NBC_Iallreduce** operations (from libNBC) while multiple **MPI_Allreduce** operations are executed in a strictly sequenced manner.

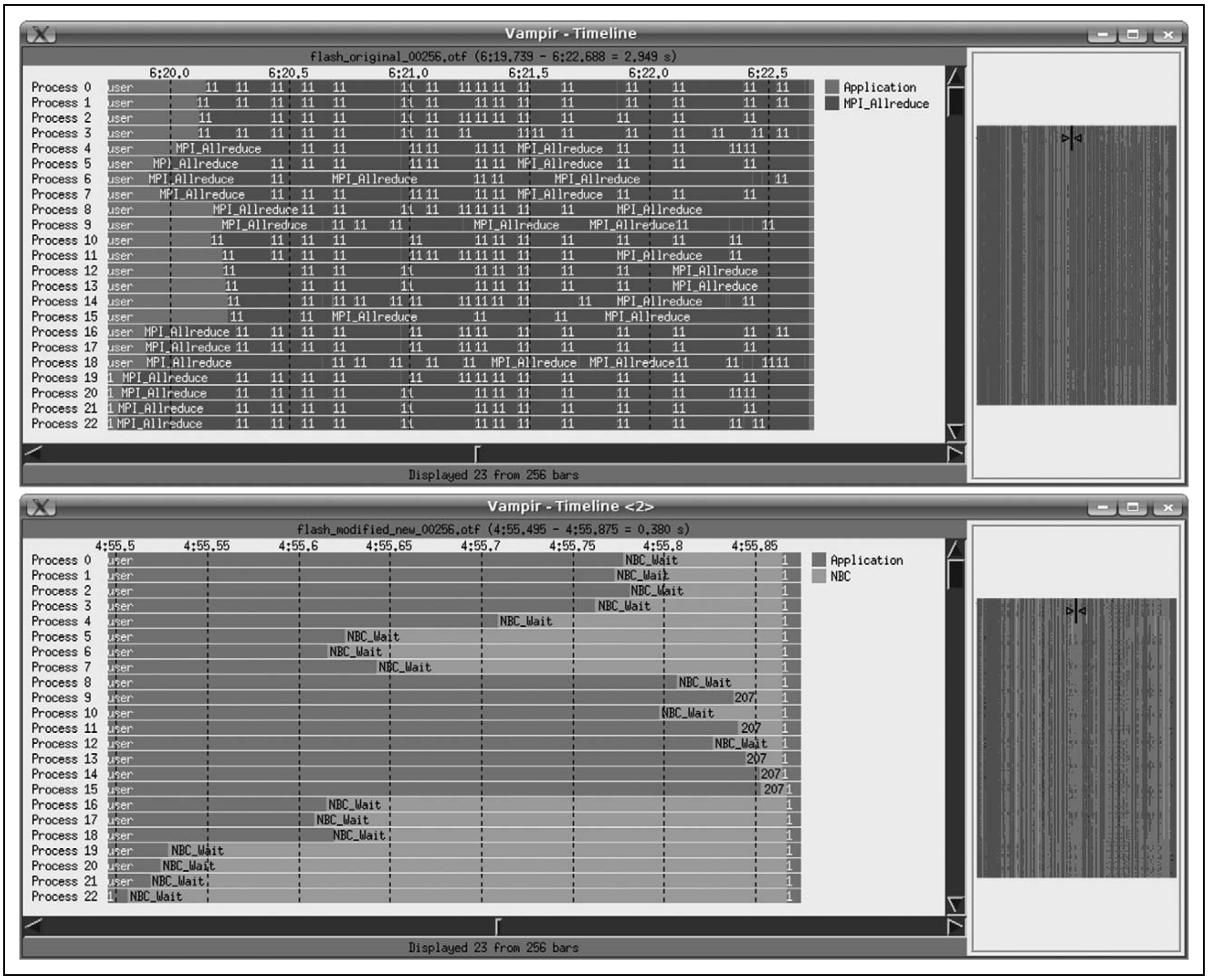

Figure 2. Corresponding communication patterns of **MPI_Allreduce** in the original code (top) and **NBC_Iallreduce** plus **NBC_Wait** in the optimized version (bottom). The latter is more than seven times faster, taking 0.38 s instead of 2.95.

In Figure 2, two corresponding allreduce patterns are compared.[7] The original communication pattern spends almost 3 s in **MPI_Allreduce** calls, see Figure 2 (top). The replacement needs only 0.38 s, consisting mainly of **NBC_Wait** calls because the **NBC_Iallreduce** calls are too small to notice with the given zoom level, compare Figure 2 (bottom). This provides an acceleration of more than seven times for the communication patterns alone. It achieves a total runtime reduction of up to 30% when using 256 processes as an example (excluding initialization of the application).

Again, the actual reason for this performance problem is easily comprehensible with the visualization of an event trace. But it would be lost in the statistical results offered by profiling approaches.

2.3 Unnecessary barriers

Another MPI operation consuming a high runtime share is **MPI_Barrier**. For 256 to 15,812 cores, it uses about 18% of the total execution time.

Detailed investigations with the Vampir tools reveal typical situations where barriers are placed. It turns out that most barriers are unnecessary for the correct execution of the code. As shown in Figure 3 (top) such barriers are placed before communication phases, probably in order to achieve strict temporal synchronization, that is, making communication phases start almost simultaneously.

A priori, this is neither beneficial nor harmful. Often, the time spent in the barrier would be spent waiting at the beginning of the next MPI operation if the barrier were removed. This is true, for example, for the **MPI_Sendrecv_replace** operation. Yet, for some other MPI operations the situation is completely different. Removing the barrier will save almost the total barrier time. This is found, for example, with **MPI_Irecv**, which starts without an initial waiting time once the barrier is removed. Here, unnecessary barriers are very harmful.

Now, reconsidering the hotspots of **MPI_Sendrecv_replace** calls discussed in Section 2.1, the situation has been changed from the former case to the latter. So, the earlier optimization receives a further improvement by removing

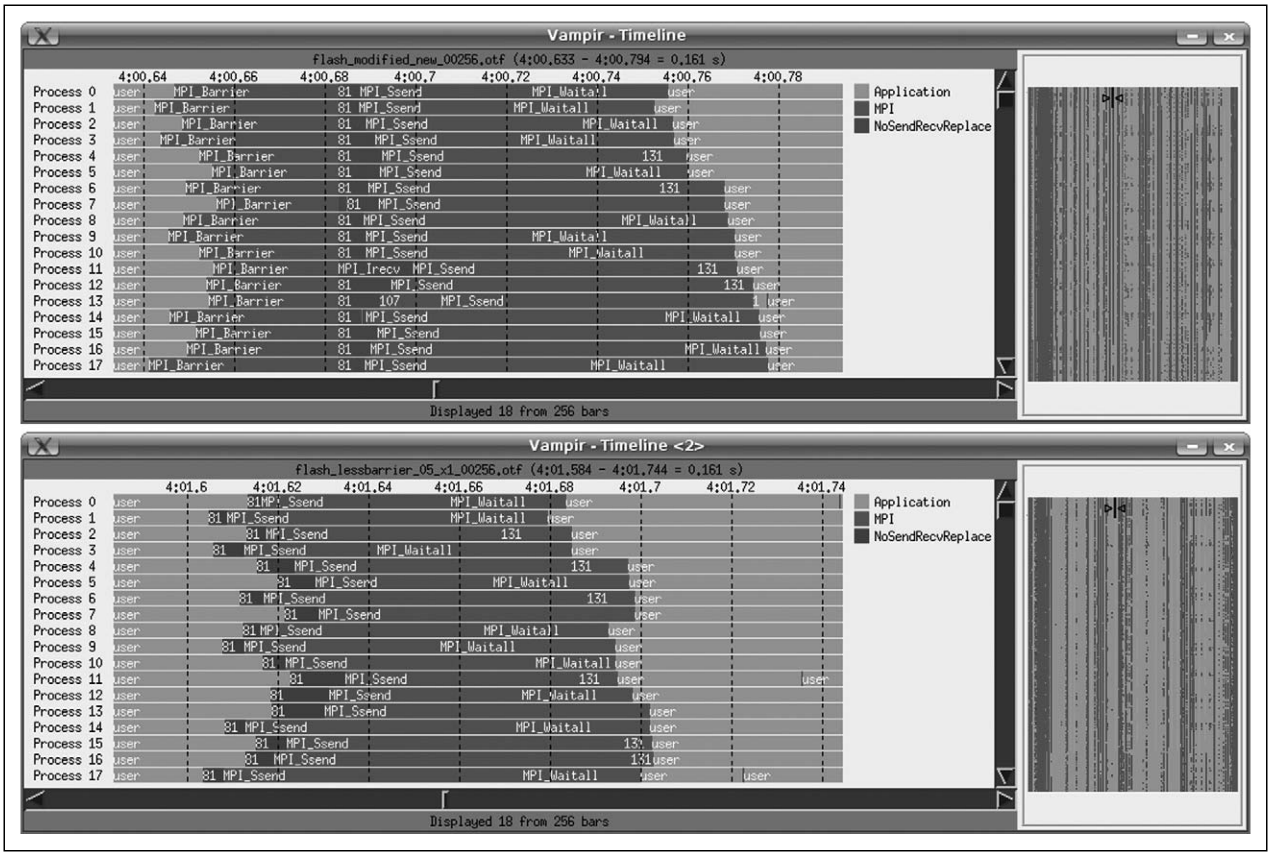
Vampir - Timeline
flash_modified_new_00256.otf (4:00.633 - 4:00.794 = 0.161 s)
Application
MPI
NoSendRecvReplace
Displayed 18 from 256 bars
Vampir - Timeline <2>
flash_lessbarrier_05_x1_00256.otf (4:01.584 - 4:01.744 = 0.161 s)
Application
MPI
NoSendRecvReplace
Displayed 18 from 256 bars

Figure 3. Typical communication pattern in the FLASH code. An **MPI_Barrier** call before a communication phase ensures a synchronized start of the communication calls (top). When the barrier is removed, the start operations are not synchronized (bottom). Yet, this imposes no additional time on the following MPI operations, the runtime per communication phase is reduced by approximately $^1/_3$.

unnecessary **MPI_Barrier** calls. Figure 3 (bottom) shows the result of the combined modification. According to the runtime profile (not shown), the aggregated runtime of **MPI_Barrier** is almost completely eliminated.

Besides the unnecessary barriers, there are also some useful ones. These mainly belong to internal measurements within the FLASH code, which aggregates coarse statistics about total runtime consumption of various components. Barriers next to checkpointing operations are also sensible.

By eliminating unnecessary barriers, the runtime share of **MPI_Barrier** is reduced by 33%. This lowers the total share of MPI by 13% while the runtime of all non-MPI code remains constant. This results in an overall runtime improvement of 8.7% when using 256 processes.

While the high barrier time would certainly attract attention in a profile, the distinction between unnecessary and useful ones would be completely obscured. The alternative is either a quick and easy look at the detailed event trace visualization or tedious manual work with phase profiles and scattered pieces of source code.

3 I/O performance problems

The second important issue for the overall performance of FLASH code is the I/O behavior, which is mainly due to the integrated checkpointing mechanism. We collected I/O data from FLASH on Jaguar for jobs ranging from 256 to 15,812 cores. From this weak-scaling study it is apparent that time spent in I/O routines began to dominate dramatically as the number of cores increased. A runtime breakdown over trials with an increasing number of cores, shown in Figure 4, illustrates this behavior.[8] More precisely, Figure 4(a) depicts the evolution of a selection of five important FLASH function groups without I/O where the corresponding runtimes grow not more than 1.5 times.[9] The same situation but with checkpointing, as in Figure 4(b), shows a 22-fold runtime increase for 8,192 cores, which clearly indicates a scalability problem.

In the following three sections, multiple tests are performed with the goal of tuning and optimizing I/O performance for the parallel file system so that the overall performance of FLASH can be significantly improved.

3.1 Collective I/O via HDF5

For the FLASH investigation described in this section, the Hierarchical Data Format, version 5 (HDF5), is used as the I/O library. HDF5 is not only a data format but also a software library for storing scientific data. It is based on a generic data model and provides a flexible and efficient I/O

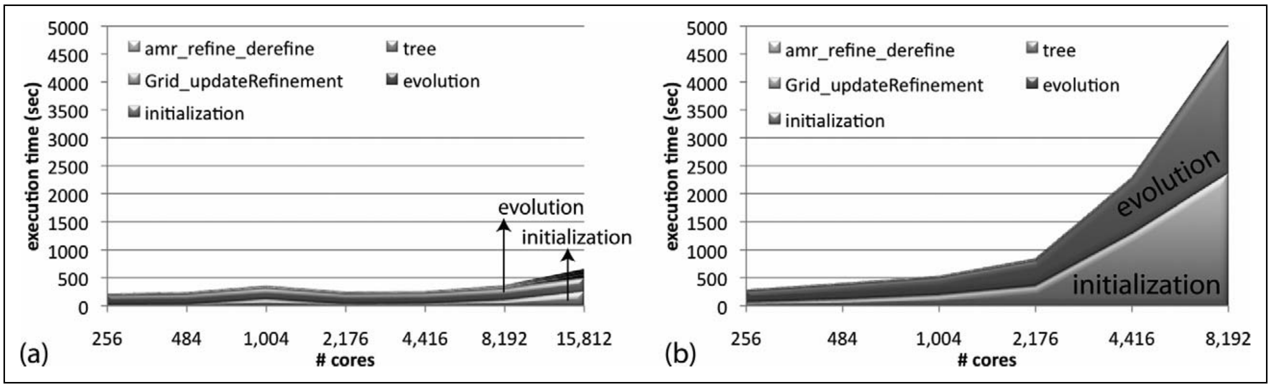

Figure 4. Weak-scaling study for a selection of FLASH function groups: (a) scalability without I/O and (b) break-down of scalability due to checkpointing

API (Yang and Koziol). By default, the parallel mode of HDF5 uses an independent access pattern for writing datasets without extra communication between processes.[5]

However, parallel HDF5 can also perform in aggregation mode, writing the data from multiple processes in a single chunk. This involves network communications among processes. Still, combining I/O requests from different processes in a single contiguous operation can yield a significant speedup (Yang and Koziol). This mode is still experimental in the FLASH code. However, the considerable benefits may encourage the FLASH application team to implement it permanently.

While Figure 4 depicts the evolution of five important FLASH function groups only, Figure 5 summarizes the weak-scaling study results of the entire FLASH simulation code for various I/O options. It can be observed that collective I/O yields a performance improvement of 10% for small core counts while for large core counts the entire FLASH code runs faster by up to a factor of 2.5. However, despite the improvements so far, the scaling results are still not satisfying for a weak-scaling benchmark. We found two different solutions to notably improve I/O performance. The first one relies only on the underlying Lustre file system without any modifications of the application. The second one requires changes in the HDF5 layer of the application. Therefore, the latter is of an experimental nature but more promising in the end. Both solutions are discussed below.

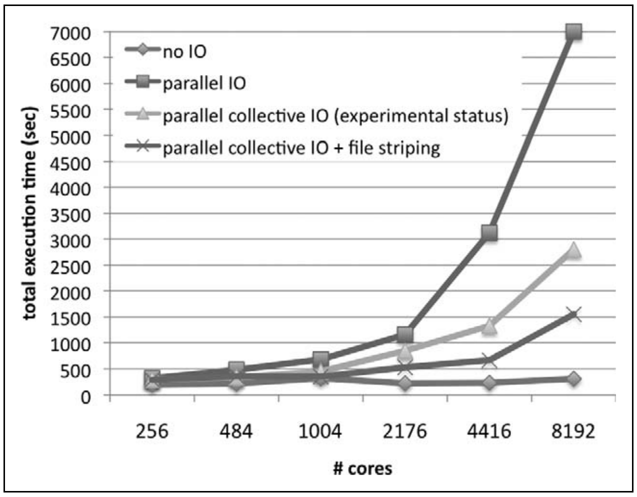

Figure 5. FLASH scaling study with various I/O options

3.2 File striping in Lustre FS

Lustre is a parallel file system that provides high aggregated I/O bandwidth by striping files across many storage devices (Yu et al., 2007). The parallel I/O implementation of FLASH creates a single checkpoint file and every process writes its data to this file simultaneously via HDF5 and MPI-IO.[5] The size of such a checkpoint file grows linearly with the number of cores. For example, in the 15,812-core case the size of the checkpoint file is approximately 260 GB.

By default, files on Jaguar are striped across four OSTs. As mentioned in Section 1.1, Jaguar consists of three file systems of which two have 72 OSTs and one has 144 OSTs. Hence, by increasing the default stripe size, the single checkpoint file can take advantage of the parallel file system, which should improve performance. Striping pattern parameters can be specified on a per-file or per-directory basis (Yu et al., 2007). For the investigation described in this section, the parent directory has been striped across all the OSTs on Jaguar, which is also suggested in Larkin and Fahey (2007). More precisely, depending on what file system is used, the Object Storage Client (OSC) communicates via a total of 72 OSSes – which are shared between all three file systems – to either 72 or 144 OSTs.

From the results presented in Figure 5, it is apparent that using parallel collective I/O in combination with striping the output file over all OSTs is highly beneficial. The results show a further improvement by a factor of 2 for mid-size and large core counts by performing collective I/O with file striping compared to the collective I/O results. This yields an overall improvement for the entire FLASH code by a factor of 4.6 when compared to the results from the naïve parallel I/O implementation.

This substantial improvement can be verified by the trace-based analysis of the I/O performance counters for

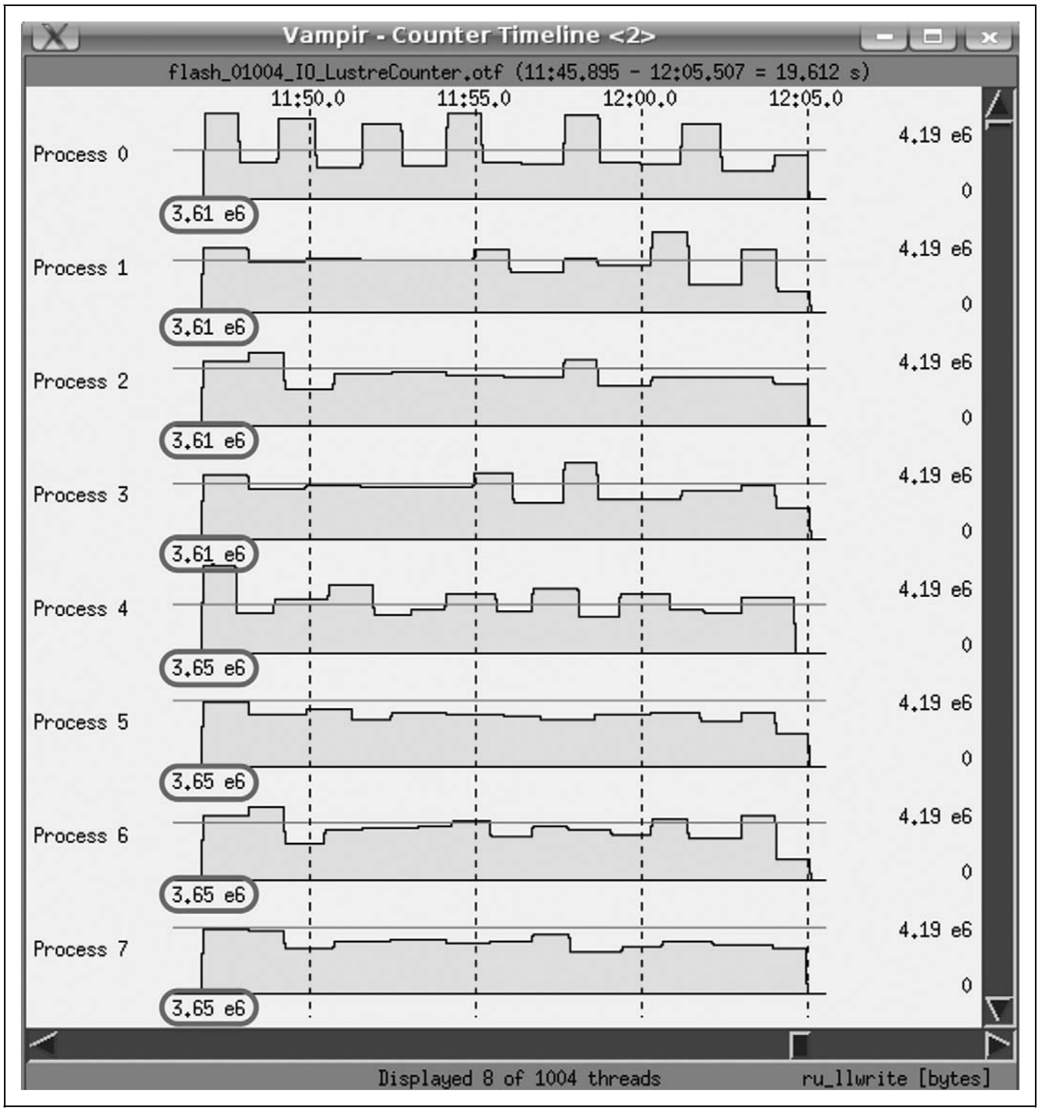

Figure 6. Performance counter displays for the write speeds of processes. The original bandwidth utilization is slow and irregular (left). It becomes faster and more uniform when using collective I/O in combination with file striping (right). All counters show the aggregated per-node bandwidth of four processes. (The rather slow maximum bandwidth of 6 MB/s corresponds to a share of the total bandwidth for 1004 out of 31,328 cores for the scr72a file system.)

a single checkpoint phase, as shown in Figure 6. This reveals that utilizing efficient collective I/O in combination with file striping (right) results in a faster as well as a more uniform write speed, while the naïve parallel I/O implementation (left) behaves more slowly and rather irregularly.

3.3 Split writing

By default, the parallel implementation of HDF5 for a PARAMESH (MacNeice et al., 1999) grid creates a single file and the processes write their data to this file simultaneously.[5] However, it relies on the underlying MPI-IO layer in HDF5. Since the size of a checkpoint file grows linearly with the number of cores, I/O might perform better if all processes write to a limited number of separate files rather than to a single file. Split file I/O can be enabled by setting the **outputSplitNum** parameter to the number *N* of files desired.[5] Every output file will be then broken into *N* subfiles. It is important to note that the use of this mode with FLASH is still experimental and has never been used in a production run. This study uses collective I/O operations but the file striping is set to the default position on Jaguar. Furthermore, it is performed for two test cases only but with various numbers of output files. Figure 7 shows the total execution time for FLASH running on 2,176 and 8,192 cores while the number of output files varies from 1 (which is default) to 64 and 4,096 respectively. In this figure the results from the split-writing analysis are compared with those from collective I/O investigations where data is written to a single file.

For the investigated cases, it is noticeable that writing data to multiple files is more efficient than writing to a single file followed by striping the file across all OSTs. This is most likely due to the overhead of the locking mechanism

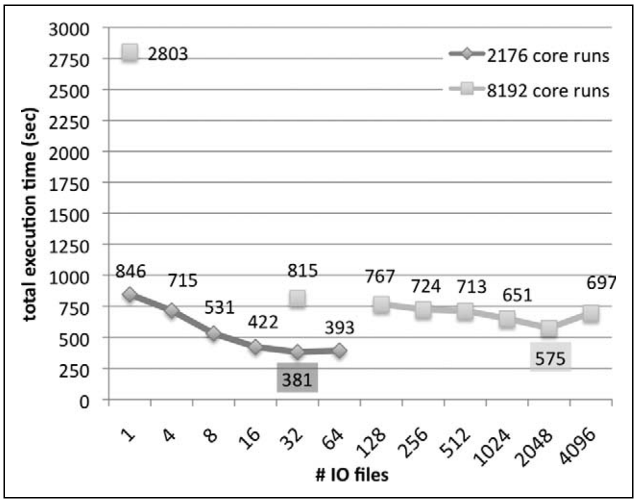

Figure 7. I/O analysis of writing data to a single file vs. multiple files.

in Lustre. For the 2,176-core run it appears that writing to 32 separate files delivers the best performance. Even when compared with the 'collective I/O + file striping' trial that has a runtime of $\sim$ 529 seconds, the split writing strategy decreases the runtime to $\sim$ 381 seconds and delivers a speedup of approximately 28% for the entire application. For the same comparison, the 8,192-core run saw a runtime reduction from $\sim$ 1551 to $\sim$ 575 seconds when data is written to 2,048 separate files. This results in a performance gain of nearly a factor of 2.7. Note the slowdown for the 8,192-core run when going from 2,048 files to 4,096 files. This issue might be due to using too many files. It is intended to carry out further research to find the optimal file size and the optimal number of files to obtain the best performance.

3.4 Limited I/O-tracing capabilities on Cray XT4

The I/O tracing capabilities of VampirTrace are very limited on the Jaguar system, because two important features cannot be used. The first is the recording of POSIX I/O calls, which is deactivated because of the absence of shared library support on the compute nodes. The second is the global monitoring of the Lustre activity, which would require administrative privileges. Both features are extensively described in Mickler et al. (2008) and Jurenz.[4]

Therefore, the only alternative was to rely on client-side Lustre statistics, which are shown in Figure 6. They represent the total I/O activity per compute node with a maximum granularity of $1/s$.

This compromise solution is sufficient for a coarse analysis of the checkpoint phases and the I/O speed. It allows us to observe the I/O rate over time, the load balance across all I/O clients for each individual checkpoint stage, and in general to observe the distributions of I/O among the processes. Due to the limitations and due to the coarse sampling rate, the I/O performance information comes close to what an elaborate profiling solution could offer. Still, to the best of our knowledge, there is no such profiling tool for parallel file systems available. However, more detailed insights into the behavior of the HDF5 library would be desirable, for example, concerning block sizes and scheduling of low-level I/O activities. An I/O monitoring solution (which works on this platform) as described in Mickler et al. (2008) would also allow observation of the activities on the metadata server, the OSSes, and the RAID systems.

4 Lessons learned with tracing

Event tracing for highly scalable applications is a challenging task, in particular due to the huge amount of data generated. The default configuration of VampirTrace is limited to record not more than 10,000 calls per subroutine and rank (MPI process) and to 32 MB of total uncompressed trace size per rank. This avoids excessively huge trace files and allows the generation of a custom filter specification for successive trace runs. These filters reduce frequent subroutine calls completely and keep high-level subroutines untouched. Usually, this results in an acceptable trace size per process and a total trace size that grows linearly with the number of parallel processes. Filtering everything except MPI calls is a typical alternative if the analysis focuses on MPI only. With the FLASH code, the filtering approach works well and creates reasonably sized traces. As an exception, additional filtering for the MPI function **MPI_Comm_rank** was necessary, because it is called hundreds of thousands of times per rank.

The growth of the trace size is typically not linear with respect to the runtime or the number of iterations. Instead, there are high event rates during initialization with many different, small, and irregular activities. Afterwards, there is a slow linear growth proportional to the number of iterations. This can be roughly described by the following relation

$$\text{trace size} = 6\ \text{MB/rank} + 0.1\ \text{MB/iteration/rank} \quad (1)$$

(in compressed OTF format) where the first part relates to initialization.

On the analysis and visualization side, VampirServer provides very good scalability because of its client/server architecture with a distributed server. It is able to handle 1 to n trace processes with one analysis process and requires approximately the uncompressed trace file size in distributed main memory. This combined approach is feasible up to a number of several hundred to a few thousand processes but not for tens of thousands because of the following reasons:

1. the total data volume grows to hundreds of GB,
2. the distributed memory consumption for analysis, and
3. the limited screen size and limited human visual perception.

For the three problems, there are different solutions. The general method for this paper was to perform trace runs with medium-scale parallelism (several hundred to a few thousand ranks). Then identify and investigate interesting situations based on these experiments, interpolating the behavior for even larger rank counts. This successfully reveals various performance problems and allows the design of effective solutions. Yet, it is not sufficient for detecting performance problems that emerge only for even higher degrees of parallelism.

Some of the current investigations are also based on analyzing partial traces where all processes are recorded but only a (manual, by modifying the anchor file of an OTF trace) selection is loaded by VampirServer. This results in few warnings about incomplete data, yet the remaining analysis works as before.

5 Future plans

For a future solution, we propose a new *partial tracing* method as the result of the presented study. It will apply different levels of filtering, based on the assumption that (most) processes in SPMD (Single Program Multiple Data) applications behave very similarly. Only a selected set of processes is considered for normal tracing including normal filtering. For another set, there will be a reduced tracing, that collects only events corresponding to the first set, for example, communication with peers in the first set. All remaining processes will refrain from recording any events.

The longer term development will focus on the automatic detection of regular sections in an event trace in order to reduce the amount of data for a deeper analysis. Based on this, the visualization will provide a high-level overview of regular areas of a trace run as well as anomalous features. Then, a single instance of a repeated pattern will serve as the basis for a more detailed inspection. Single outliers with notably different behavior can be easily identified and compared with the regular case.

6 Related work

Detailed performance-analysis tools are becoming more crucial for the efficiency of large scale parallel applications and at the same time the tools face the same scalability challenges. Currently, this seems to produce two trends. On the one hand, profiling approaches are enriched by additional data, for example, with phase profiles or call path profiles (Malony et al., 2006; Szebenyi et al., 2009). On the other hand, data intensive event tracing methods are being adapted for data reduction, for example, the extension in Paravar by Casas et al. (2007). A compromise between profiling and tracing was proposed by Fürlinger and Skinner (2009).

A way to cope with huge amounts of event trace data was initially proposed by Knüpfer and Nagel (2006) and Knüpfer (2008), with the Compressed Call Graph method, followed by Noeth et al. (2009) with the ScalaTrace approach for MPI replay traces. Both can be used for automatic identification and utilization of regular repetition patterns. In the past, the same goal has been sought with different methods (Roth et al., 1996; Samples, 1989). A good overview of the different approaches used by Mohror et al. can be found in (Mohror and Karavanic, 2009).

7 Conclusions

This paper presents a performance-analysis study of the parallel simulation software FLASH that examines the application behavior as well as the fundamental high-end petascale system hierarchies. The approach is performed using the scalable performance-analysis tool suite called Vampir on the ORNL's Cray XT4 Jaguar system. The trace-based evaluation provides important insights into performance and scalability problems and allows us to identify two major bottlenecks that are of importance for very high CPU counts.

The FLASH application was considered to be already rather well optimized. In our opinion, the fact that we were able to identify notable possibilities for improvement still shows that high-scale performance is a very complex topic and that specially tailored tools are crucial.

The use of the Vampir suite allows not only the detection of severe hotspots in some of the communication patterns used in the FLASH application but is also beneficial in pointing to feasible solutions. Consequently, a speedup of the total runtime of up to 30% can be achieved by replacing multiple, strictly successive **MPI_Allreduce** operations by non-blocking **NBC_Iallreduce** operations (from libNBC) that permit overlapping of messages. Furthermore, another MPI-related bottleneck could be eliminated by substituting the latency-bound **MPI_Sendrecv_replace** operations with non-blocking communication calls; as well as by removing unnecessary **MPI_Barrier** calls. This reduces the total portion of MPI in FLASH by 13% while the runtime of all non-MPI code remains constant.

A deeper investigation of the causes for the time spent in FLASH routines shows in particular that time spent in I/O routines began to dominate dramatically as the number of CPU cores increased. A trace-based analysis of the I/O behavior allows a better understanding of the complex performance characteristics of the parallel Lustre file system. Using various techniques, like aggregating write operations, allowing the data from multiple processes to be written to disk in a single path, in combination with file striping across all OSTs, yields a significant performance improvement by a factor of 2 for midsize CPU counts and approximately 4.6 for large CPU counts for the entire FLASH application. An additional investigation shows that writing data to multiple files instead of a single file delivers a performance gain of nearly a factor of 2.7 for 8,192 cores, for example. Since the size of the output file grows linearly with the number of cores, it is planned to find the optimal file size and optimal number of output files to obtain the best performance for various core cases.

Acknowledgements

The authors would like to thank the FLASH application team, in particular Chris Daley for his continuous support of the application. Furthermore, Jeff Larkin (Cray) is highly acknowledged for providing valuable insights into the Lustre file system on Jaguar. The authors would also like to thank David Cronk (UTK) for discussions about various MPI I/O implementations.

Funding

This research was sponsored by the Office of Mathematical, Information, and Computational Sciences of the Office of Science, US Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. This work used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725. This resource was made available via the Performance Evaluation and Analysis Consortium End Station, a Department of Energy INCITE project.

Conflict of interest statement

None declared.

Notes

1. Top500 list, Nov. 2009, http://www.top500.org/list/2009/11/100.
2. Top500 list, June 2008, http://www.top500.org/list/2008/06/100.
3. VampirServer User Guide, http://www.vampir.eu.
4. M. Jurenz, VampirTrace Software and Documentation, ZIH, Technische Universität Dresden, http://www.tu-dresden.de/zih/vampirtrace.
5. ASC FLASH Center University of Chicago, FLASH Users Guide Version 3.1.1, January 2009.
6. MPI: A Message-Passing Interface – Standard Extension: Nonblocking Collective Operations (draft), Message Passing Interface Forum, Jan 2009, https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/NBColl.
7. Event tracing allows identification of exactly corresponding occurrences for compatible test runs. In this example both are at the middle of the total runtime.
8. Because of the extreme complexity of FLASH, we focus on those FLASH function groups that show poor scaling behavior possibly due to I/O function calls.
9. When compared to the 256-core case. With ideal weak-scaling it should be constant.

References

Brunst H (2008) Integrative concepts for scalable distributed performance analysis and visualization of parallel programs, PhD thesis, Shaker Verlag.

Casas M, Badia RM and Labarta (2007) Automatic structure extraction from mpi applications tracefiles. In: Proceedings of Euro-Par 2007, Springer LNCS 4641, Rennes, France.

Fürlinger K and Skinner D (2009) Capturing and visualizing event flow graphs of MPI applications. In: Proceedings of Workshop on Productivity and Performance (PROPER 2009) in conjunction with Euro-Par 2009, Delft, The Netherlands, Aug.

Hoefler T, Gottschling P and Lumsdaine A (2008) Leveraging non-blocking collective communication in high-performance applications. In: SPAA'08, Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, ACM, pp. 113–15.

Hoefler T, Kambadur P, Graham RL, Shipman G and Lumsdaine A (2007) A case for standard non-blocking collective operations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, EuroPVM/MPI 2007, Springer LNCS 4757, pp. 125–34.

Jagode H, Dongarra J, Alam S, Vetter J, Spear W, Malony A (2009) *A Holistic Approach for Performance Measurement and Analysis for Petascale Applications, ICCS 2009, Part II, LNCS 5545*. Berlin, Heidelberg: Springer-Verlag, 686–95.

Jordan GC, Fisher RT, Townsley DM, ACalder AC, Graziani C, Asida S, et al. (2008) Three-dimensional simulations of the deflagration phase of the gravitationally confined detonation model of type Ia supernovae. *The Astrophysical Journal*, 681: 1448–57.

Knüpfer A (2008) Advanced memory data structures for scalable event trace analysis, PhD thesis, Technische Universität Dresden.

Knüpfer A and Nagel WE (2006) Compressible memory data structures for event-based trace analysis. *Future Generation Computer Systems* 22(3): 359-68.

Knüpfer A, Brendel R, Brunst H, Mix h and Nagel WE (2006) Introducing the Open Trace Format (OTF). In: Proceedings of the ICCS 2006, part II. pp. 526–533, Reading, UK.

Knüpfer A, Brunst H, Doleschal J, Jurenz M, Lieber M, Mickler H, Müller M and Nagel WE (2008) The Vampir performance analysis tool-set. In: *Tools for High Performance Computing*, Springer Verlag, 139-55.

Larkin J and Fahey M (2007) Guidelines for efficient parallel I/O on the Cray XT3/XT4. In: Proceedings of Cray User Group.

MacNeice P, Olson KM, Mobarry C, deFainchtein R, Packer C (1999) PARAMESH: A parallel adaptive mesh refinement community toolkit, NASA/CR-1999-209483.

Malony AD, Shende SS and Morris A (2006) Phase-based parallel performance profiling, parallel computing: current & future issues of high-end computing. In: Proceedings of ParCo 2005, Jülich, Germany.

Mickler H, Knüpfer A, Kluge M, Müller M and Nagel WE (2008) Trace-based analysis and optimization for the Semtex CFD application – hidden remote memory accesses and I/O performance. In: Euro-Par 2008 Workshops – Parallel Processing, Las Palmas de Gran Canaria, pp. 287-296, Springer LNCS 5415, Aug.

Mohror K and Karavanic KL (2009) Evaluating similarity-based trace reduction techniques for scalable performance analysis. In: Proceedings of SC '09, pp. 1–12, Portland, Oregon.

Noeth M, Ratn P, Mueller F, Schulz M and de Supinski BR (2009) ScalaTrace: Scalable compression and replay of communication traces for high performance computing. *Journal of Parallel and Distributed Computing* 69(8): 696–710.

Roth PC, Elford C, Fin B, Huber J, Madhyastha T, Schwartz B and Shields K (1996) Etrusca: Event Trace Reduction Using Statistical Data Clustering Analysis, PhD thesis.

Samples AD (1989) Mache: No-loss trace compaction. In: Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, pp. 89–97, Oakland, California, USA.

Szebenyi Z, Wolf F and Wylie BJN (2009) Space-efficient time-series call-path profiling of parallel applications. In: SC09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, USA.

Yang M and Koziol Q. Using collective IO inside a high performance IO software package – HDF5, www.hdfgroup.uiuc.edu/papers/papers/ParallelIO/HDF5-CollectiveChunkIO.pdf

Yu W, Vetter J, Canon RS and Jiang S (2007) Exploiting Lustre file joining for effective collective IO. In: Int. Conference on Clusters Computing and Grid (CCGrid '07), Rio de Janeiro, Brazil, IEEE Computer Society.

Authors Biographies

Heike Jagode received BSc and MSc degrees in Applied Mathematics from the University of Applied Sciences, Mittweida, Germany, in 2001. She earned a second MSc in High Performance Computing from the University of Edinburgh, Edinburgh Parallel Computing Centre (EPCC), in Scotland in 2006. She was a Research Associate at the Center for Information Services and High Performance Computing (ZIH) at Dresden University of Technology in Germany from 2002 to 2008. Since March 2008 she has been a Senior Research Associate at the Innovative Computing Laboratory (ICL) at the University of Tennessee in Knoxville (UTK). Her current research interests include studies in computer science for performance of high-performance computing applications and architectures, focusing primarily on developing methods and tools for scalable performance analysis, tuning, and optimization of HPC applications. She is also currently enrolled in a PhD program at the Department of Computer Science at UTK (since fall 2009).

Andreas Knüpfer is a research scientist at the Center for Information Services and HPC at Technische Universität Dresden. His fields of interest are parallel programming paradigms and HPC performance analysis. He received a diploma in mathematics in 2002 and a doctorate in computer science in 2008, both from TU Dresden.

Jack Dongarra is a University Distinguished Professor of Computer Science in the Electrical Engineering and Computer Science Department at the University of Tennessee and is a Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turing Fellow in the Computer Science and Mathematics Schools at the University of Manchester, and an Adjunct Professor in the Computer Science Department at Rice University. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing, and documentation of high-quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, Open-MPI, and PAPI. He has published approximately 300 articles, papers, reports, and technical memoranda and he is co-author of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions to the application of high-performance computers using innovative approaches and in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement. He is a Fellow of the AAAS, ACM, IEEE, and SIAM, and a member of the National Academy of Engineering.

Matthias Jurenz is a technical employee at the Center for Information Services and HPC at Technische Universität Dresden. He is the main developer of the performance measurement software tool VampirTrace. He completed an apprenticeship as software engineer at TU Dresden in 2002.

Matthias S Müller is deputy director and CTO of ZIH at TU Dresden. He received a PhD in Computational Physics from Stuttgart University in 2001. From 1999 to 2005 he worked at the High Performance Computing Center in Stuttgart, Germany, becoming deputy director. His research interests include programming methodologies and tools, computational science on high-performance computers, and grid computing. He is the author or co-author of more than 70 refereed publications on these topics. He is also head of the VampirTrace development group. He is a member of the German Physical Society (DPG) and Vice Chair of SPEC's High Performance Group.

Prof Dr Wolfgang E Nagel holds the Chair for Computer Architecture at Technische Universität Dresden (TUD). After his university studies of computer science at RWTH Aachen from 1979 to 1985, he worked in the area of parallel computing at the Central Institute for Applied Mathematics, Research Center Jülich, and at the Center for Advanced Computing Research (CACR), Caltech. Since 1997, he has been director of the Center for Information Services and High Performance Computing (ZIH) - the former Center for HPC

(ZHR) - at TUD, which was founded by him. From 2006 to 2009, he served as dean of the Computer Science department at TUD. His research profile covers modern programming concepts and software tools to support complex compute intensive applications, analysis of innovative computer architectures, and the development of efficient algorithms and methods. He has published about 110 papers in those areas, and has contributed as program committee member, program chair, or general chair at more than 45 conferences and workshops.