Trace-based Performance Analysis for the Petascale Simulation Code FLASH

Heike Jagode, Jack Dongarra
The University of Tennessee, USA
[jagode | dongarra]@eecs.utk.edu

Andreas Knüpfer, Matthias Jurenz,
Matthias S. Müller, Wolfgang E. Nagel
Technische Universität Dresden, Germany
[andreas.knuepfer | matthias.jurenz |
matthias.mueller | wolfgang.nagel]@tu-dresden.de

September 24, 2010

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 with the Vampir performance analysis tool suite that examines the application behavior as well as the fundamental system properties.

The study is done on the Jaguar system at ORNL, the fastest computer on the November 2009 Top500 list. We analyze the FLASH simulation code that is designed to scale towards tens of thousands of CPU cores. This situation makes it very complex to apply existing performance analysis tools. Yet, the study reveals two classes of performance problems that become relevant with 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.

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 those high-end computing platforms are highly communication- as well as data-intensive. As an example, the FLASH application is a highly parallel simulation code containing complex performance characteristics.

The performance analysis tool suite Vampir is used to gain deeper insight into performance and scalability problems of the application. It uses event trac-

ing 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 [3, 4].

This performance evaluation of the FLASH software exposes two classes of performance issues that become relevant for very high CPU counts. The first class is related to inter-process communication and can be summarized under the common heading "overly strict coupling of processes". The second class refers to 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.

The remaining 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 applied performance analysis tool suite Vampir. A brief outline of the FLASH code is provided at the end of the introduction and background section. In section 2 and 3 we provide extensive performance measurement and analysis results that are collected on the Cray XT4 system, followed by a discussion of the detected performance issues, the proposed optimizations and their outcomes. Section 4 is dedicated to experiences with the highly parallel application of the Vampir tools as well as to future adaptations for such scenarios. The paper ends with the conclusions and an outlook to future work.

1.1 The Cray XT4 System Jaguar

We start with a short description of the key features - most relevant for this study - of the Jaguar system, the fastest computer on the November 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 done, it was based on Cray XT4 hardware and utilized 7,832 quad-core AMD Opteron processors with a clock frequency of 2.1 GHz and 8 GBytes of main memory (2 GBytes 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 [17]. All three of these file systems shared 72 physical Object Storage Server (OSS). The theoretical peak performance of I/O bandwidth was $\sim$50 GB/s across all OSSes.

1.2 The Vampir Performance Analysis Suite

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

The Vampir suite consists of the VampirTrace component for instrumentation, monitoring and recording as well as the VampirServer component for

visualization and analysis [5, 6, 3]. The event traces are stored in the *Open Trace Format* (OTF) [7]. 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 [5, 8]. 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 one hundred GBytes [8, 3].

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 [9]. 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 [10] is utilized 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 [11]. For this study, the I/O due to checkpointing is most relevant, because it frequently writes huge amounts of data.

The examined three-dimensional simulation test case `WD_Def` is a deflagration phase of the gravitationally confined detonation mechanism for Type Ia supernovae, a crucial astrophysical problem that has been extensively discussed in [12]. The `WD_Def` test case is generated as a weak scaling problem for up to 15,812 processors where the number of blocks remain approximately constant per computational thread.

2 MPI Performance Problems

The communication layer is a typical place to look for performance problems in parallel codes. Although communication enables the parallel solution, it is not directly contributing to the solution of the original problem. If communication accounts for a substantial portion of the overall runtime, this indicts a performance problem.

Most of the time, communication delay is 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 that can be summarized under the heading “overly strict coupling of processes”. The

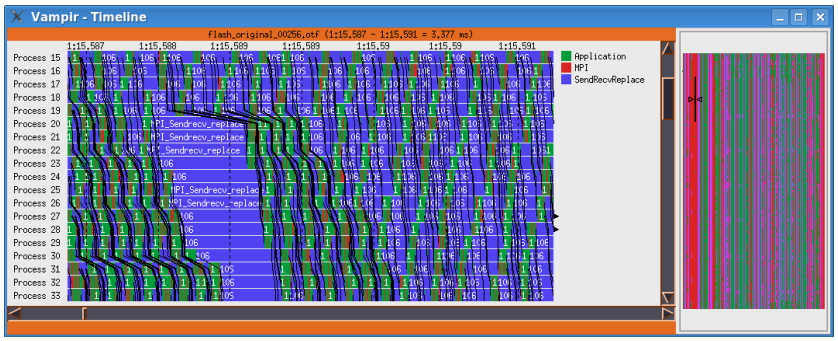

Figure 1: Original communication pattern of successive `MPI_Sendrecv_replace` calls. Message delays are propagated along the communication chain of consecutive ranks. See Figure 3 for an optimized alternative.

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 this will contribute a significant performance improvement.

The symptom of this performance limitation would be easily detectable with profiling, because the accumulated run-time 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 run-time. With sophisticated profiling approaches like call-path profiling or phase profiling one could infer the suboptimal run-time behavior when studying the relevant source code. But this would be most tedious and time consuming especially if

the analysts is different from the author.

Only tracing allows convenient examination of the situation with all necessary information from one source. In particular, this includes the contexts 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 as well as manageable trace file size, 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 disadvantages. 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.

On this account, the general advice to MPI users is to rely on collective communication whenever possible [13]. 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 [13].

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 operations. Currently, this topic is under discussion in the MPI Forum [14].

As a temporary solution for this problem, libNBC can be used [13]. 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 [13, 15]. 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 achieves a significant performance improvement with this approach. This is mainly due to the overlapping technique

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 0.38s instead of 2.95.

of the successive `NBC_Iallreduce` operations (from libNBC) while multiple `MPI_Allreduce` operations are executed in a strictly sequenced manner.

In Figure 2, two corresponding allreduce patterns are compared[1]. The original communication pattern spends almost $3s$ in `MPI_Allreduce` calls, see Figure 2 (top). The replacement needs only $0.38s$, 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 factor 7 for the communication patterns only. 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.

[1]Event tracing allows identification of exactly corresponding occurrences for compatible test runs. In this example both are at the middle of the total runtime.

2.3 Unnecessary Barriers

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

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, i.e. communication phases starting almost simultaneously.

A priori this is neither beneficial nor harmful. Often, the time spent in the barrier would be spent waiting in the beginning of the next MPI operation when the barrier is 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 can be found for example for `MPI_Irecv`, which starts without initial waiting time once the barrier is removed. Here, unnecessary barriers are most harmful.

Now, reconsidering the hotspots of `MPI_Sendrecv_replace` calls discussed in Section 2.1, this situation has been changed from the former case to the latter. Therefore, the earlier optimization allows another improvement when removing the unnecessary `MPI_Barrier` calls. Figure 3 (bottom) shows the result of this 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. They are mostly part of an internal measurement in the FLASH code which is aggregating coarse statistics about total runtime consumptions of certain components. Furthermore, barriers next to checkpointing operations are sensible.

By eliminating the 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 of unnecessary and useful ones would be completely obscured. The alternatives are 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 aspect for the overall performance of the 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 dramatically to dominate as the number of cores increased. A runtime breakdown over trials with increasing number of cores, shown in

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 removing the barrier there is an un-synchronized start (bottom). Yet, this imposes no additional time on the following MPI operations, the runtime per communication phase is reduced by approximately 1/3.

Figure 4, illustrates this behavior[2]. More precisely, Figure 4 (a) depicts the evolution of a selection of 5 important FLASH function groups without I/O where the corresponding runtimes grow not more than 1.5-fold[3]. The same situation but with checkpointing 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 API [11]. By default, the parallel mode of HDF5 uses an independent access pattern for writing datasets without extra communication between processes [9].

However, parallel HDF5 can also perform an aggregated mode, writing the

[2]Because of the great complexity of FLASH, we focus on those FLASH function groups that show poor scaling behavior and imply I/O function calls.

[3]As compared to the 256 core case. With ideal weak-scaling it should be constant.

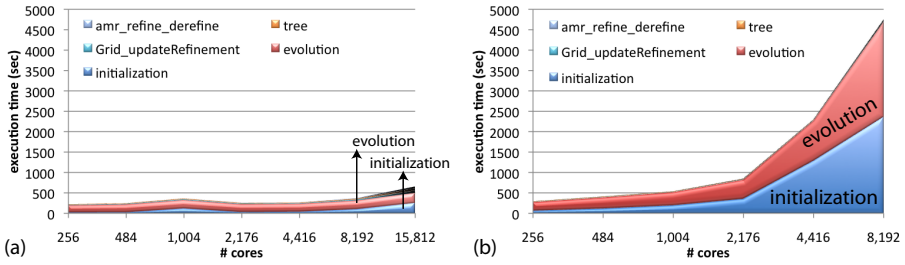

Figure 4: Weak-scaling study for a selection of FLASH function groups: (a) Scalability without I/O and (b) break-down of scalability due to checkpointing

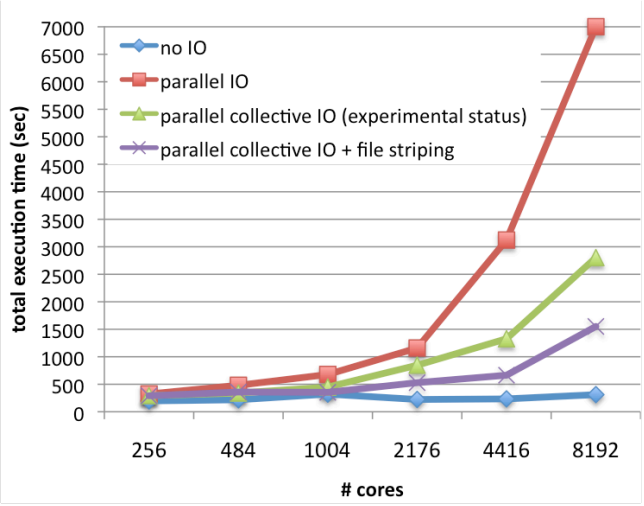

Figure 5: FLASH scaling study with various I/O options

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 [11]. 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 5 important FLASH function groups only, now 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.

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 [16]. 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 [9]. The size of such a checkpoint file grows linearly with the number of cores. As an example, in the 15,812 core case the size of the checkpoint file is approximately 260 GByte.

By default, files on Jaguar are striped across 4 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 may 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 [16]. For the investigation described in this section, the parent directory has been striped across all the OSTs on Jaguar, which is also suggested in [17]. 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 midsize 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 a single checkpoint phase, shown in Figure 6. It reveals that utilizing efficient collective I/O in combination with file striping (right) results in a faster as well as more uniform write speed, while the naïve parallel I/O implementation (left) behaves slower and rather irregularly.

3.3 Split Writing

By default, the parallel implementation of HDF5 for a PARAMESH [10] grid creates a single file and every process writes its data to this file simultaneously [9]. 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 a single file. Split file I/O can be enabled by setting the `outputSplitNum` parameter to the number N of files desired [9]. 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 back for the

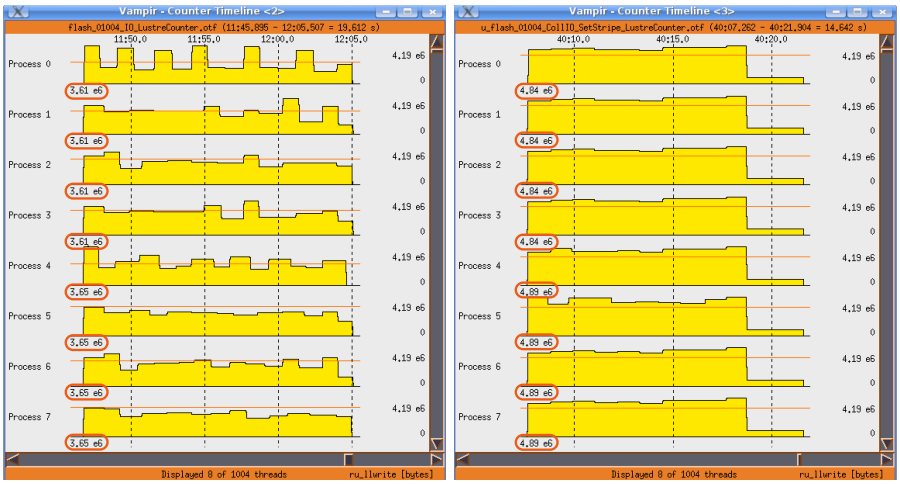

Figure 6: Performance counter displays for write speed of processes. The original bandwidth utilization is slow and irregular (left). It becomes faster and more uniform when using collective I/O in combination with file striping (right). All counters show the aggregated per-node bandwidth of 4 processes. (The rather slow maximum bandwidth of 6MB/s corresponds to share of the total bandwidth for 1,004 out of 31,328 cores for the scr72a file system.)

default case 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 when 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 in Lustre. For the 2,176 core run it appears that writing to 32 separate files delivers best performance. Even when compared with the 'collective I/O + file striping' trial that has a run time of $\sim$529 seconds, the split writing strategy decreases the run time to $\sim$381 seconds which delivers a speedup of approximately 28% for the entire application. For the same comparison, the 8,192 core run saw a run time 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 might be an issue due to the use of too many files. A future intent is to find the optimal file size or optimal number of files to obtain the best performance.

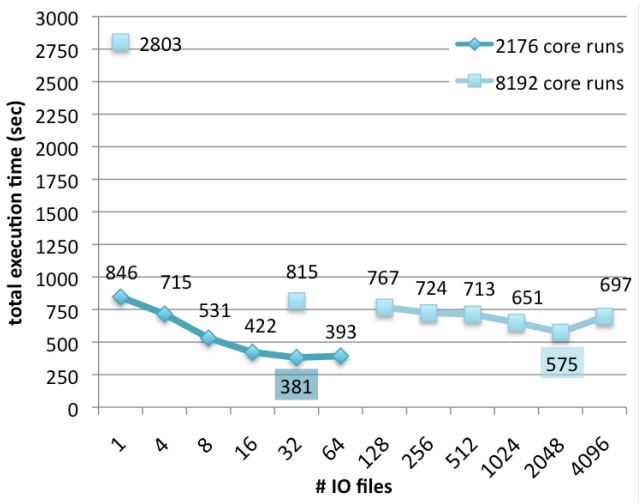

Figure 7: I/O analysis of writing data to a single file versus multiple files.

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 missing 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 [5, 18].

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 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. Yet, more detailed insight into the behavior of the HDF5 library would be desirable, e.g. concerning block sizes and scheduling of low-level I/O activities. An I/O monitoring solution (which is working on this plattform) as described in [18] 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 generated data. 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 growing 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 in order to create reasonably sized traces. As one 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 described coarsely by the following relation

$$\text{trace size} = 6\,\text{MB/rank} + 0.1\,\text{MB/iteration/rank} \tag{1}$$

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

On the analysis and visualization side, VampirServer provides very good scalability by 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 that grows to hundreds of GBytes,
2. the distributed memory consumption for analysis, and
3. limited screen size and limited human visual perception.

For the three problems, there are different solutions. The general method for this paper was to do 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 certain 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[4]) selection is loaded by VampirServer. This results in few warnings about incomplete data, yet the remaining analysis works as before.

[4] by modifying the anchor file of an OTF trace.

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, e.g. 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 about regular areas of a trace run as well as anomalous parts. Then, a single instance of a repeated pattern can serve as the basis for a more detailed inspection that represents many similar occurrences. 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 challenge. Currently, this seems to produce two trends. On the one hand, profiling approaches are being enriched by additional data, e.g. with phase profiles or call path profiles [19, 20]. On the other hand, data intensive event tracing methods are adapted towards data reduction, e.g. the extension in Paravar by Casas et al. [21]. A compromise between profiling and tracing was proposed by Fürlinger et al. [22]

A way to cope with huge amounts of event trace data was initially proposed by Knüpfer et al. [23, 24] with the Compressed Call Graph method, followed by Müller et al. [25] 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 approached with different methods by Roth et al. [26] or Samples [27]. A good overview about different approaches by Mohror et al. can be found in [28].

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 insight 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 already rather well optimized. In our opinion, the fact that we were able to identify notable potential for improvement still shows that high scale performance is a very complex topic and that special 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 derivation of time spent in FLASH routines shows in particular that time spent in I/O routines began dramatically to dominate as the number of CPU cores increase. 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 as an example. Since the size of the output file grows linearly with the number of cores, it is a future intention to find the optimal file size or optimal number of output files to obtain 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 with the application. Furthermore, Jeff Larkin (Cray) is greatly acknowledged for providing valuable insights on the Lustre file system on Jaguar. The authors also would like to thank David Cronk (UTK) for appreciated discussions about various MPI I/O implementations.

This research was sponsored by the Office of Mathematical, Information, and Computational Sciences of the Office of Science, U.S. 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.

References

[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] H. Brunst, "Integrative Concepts for Scalable Distributed Performance Analysis and Visualization of Parallel Programs", Ph.D. thesis, Shaker Verlag, 2008.

[4] H. Jagode, J. Dongarra, S. Alam, J. Vetter, W. Spear, A. Malony, "A Holistic Approach for Performance Measurement and Analysis for Petascale Applications," Springer-Verlag Berlin Heidelberg 2009, ICCS 2009, Part II, LNCS 5545, pp. 686–695, 2009.

[5] M. Jurenz, "VampirTrace Software and Documentation", ZIH, Technische Universität Dresden, http://www.tu-dresden.de/zih/vampirtrace.

[6] "VampirServer User Guide", http://www.vampir.eu.

[7] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, W. E. Nagel, "Introducing the Open Trace Format (OTF)", Proceedings of the ICCS 2006, part II. pp. 526–533, Reading/U.K., 2006.

[8] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. Müller and W.E. Nagel, "The Vampir Performance Analysis Tool-Set", in: Tools for High Performance Computing, pp 139-155, Springer Verlag, 2008.

[9] ASC FLASH Center University of Chicago, "FLASH Users Guide Version 3.1.1", January 2009.

[10] P. MacNeice, K. M. Olson, C. Mobarry, R. deFainchtein, C. Packer, "PARAMESH: A parallel adaptive mesh refinement community toolkit", NASA/CR-1999-209483, 1999.

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

[12] G. C. Jordan, R. T. Fisher, D. M. Townsley, A. C. Calder, C. Graziani, S. Asida, et al., "Three-Dimensional Simulations of the Deflagration Phase of the Gravitationally Confined Detonation Model of Type Ia Supernovae", The Astrophysical Journal, 681, pp. 1448–1457, July 2008.

[13] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, "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. 125–134, Oct 2007.

[14] "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`.

[15] T. Hoefler, P. Gottschling, and A. Lumsdaine, "Leveraging Nonblocking Collective Communication in High-Performance Applications." in SPAA'08, Proceedings of the 20'th Annual Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, ACM, pp. 113–115, 2008.

[16] W. Yu, J. Vetter, R. S. Canon, S. Jiang, "Exploiting Lustre File Joining for Effective Collective IO", Int'l Conference on Clusters Computing and Grid (CCGrid '07), Rio de Janeiro, Brazil, IEEE Computer Society, 2007.

[17] J. Larkin and M. Fahey, "Guidelines for Efficient Parallel I/O on the Cray XT3/XT4", in: Proceedings of Cray User Group, 2007.

[18] H. Mickler, A. Knüpfer, M. Kluge, M. Müller, and W.E. Nagel, "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 2008.

[19] A.D. Malony, S.S. Shende, A. Morris, "Phase-Based Parallel Performance Profiling", Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of ParCo 2005, Jülich, Germany, 2006.

[20] Z. Szebenyi, F. Wolf, B.J.N. Wylie, "Space-efficient time-series call-path profiling of parallel applications", SC09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, Portland, Oregon, USA, Nov 2009.

[21] M. Casas, R.M. Badia, J. Labarta, "Automatic Structure Extraction from MPI Applications Tracefiles", in Proceedings of Euro-Par 2007, Springer LNCS 4641, Rennes, France, Aug 2007.

[22] K. Fürlinger and D. Skinner, "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 2009.

[23] A. Knüpfer and Wolfgang E. Nagel, "Compressible Memory Data Structures for Event-Based Trace Analysis", in: Future Generation Computer Systems 22:3, pp. 359-368, 2006.

[24] A. Knüpfer, "Advanced Memory Data Structures for Scalable Event Trace Analysis", Ph.D. Thesis, Technische Universität Dresden, Dec 2008.

[25] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B.R. de Supinski, "ScalaTrace: Scalable compression and replay of communication traces for high performance computing", in Journal of Parallel and Distributed Computing, Volume 69 (8) p. 696–710, 2009, Academic Press, Orlando, FL, USA.

[26] P.C. Roth, C. Elford, B. Fin, J. Huber, T. Madhyastha, B. Schwartz, and K.Shields, "Etrusca: Event Trace Reduction Using Statistical Data Clustering Analysis", Ph.D. Thesis, 1996

[27] A.D. Samples, "Mache: No-Loss Trace Compaction", in Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systemsm, p. 89 – 97, Oakland, California, USA, 1989

[28] K. Mohror and K.L. Karavanic,"Evaluating similarity-based trace reduction techniques for scalable performance analysis", in Proceedings of SC '09, p. 1–12, Portland, Oregon, 2009