Trace-Based Analysis and Optimization for the Semtex CFD Application – Hidden Remote Memory Accesses and I/O Performance

Holger Mickler, Andreas Knüpfer, Michael Kluge,
Matthias S. Müller, and Wolfgang E. Nagel

Technische Universität Dresden, Center for Information Services and High Performance Computing (ZIH), 01062 Dresden, Germany
{holger.mickler,andreas.knuepfer,michael.kluge}@tu-dresden.de
{matthias.mueller,wolfgang.nagel}@tu-dresden.de

Abstract. In this paper we present the analysis and optimization of the Semtex CFD application on the basis of trace data obtained with VampirTrace and visualized by Vampir. In the course of the paper the evaluation of I/O performance with regard to globally shared I/O resources and the detection of hidden remote memory accesses with the help of special hardware performance counters will be highlighted.

Keywords: Tracing, Performance Analysis, Remote Memory Access, I/O.

1 Introduction

This paper presents two aspects of the analysis and optimization process of the parallel CFD (computational fluid dynamics) application Semtex on the SGI Altix 4700 platform.

Computation and communication are the most prominent targets for performance improvement. Yet, in this paper we investigate two effects related to remote memory access in a NUMA environment and to extensive I/O activity.

The following Sect. 2 gives an overview of instrumentation and trace collection with VampirTrace and trace visualization with the Vampir and VampirServer tools. Section 3 depicts the checkpointing and communication mechanisms of the Semtex CFD code and presents detailed detection of two interesting performance flaws as well as the successful optimization of both. The paper ends with a short conclusion and outlook.

2 Trace Collection and Visualization with Vampir

VampirTrace is a scalable and portable event tracing software for sequential and parallel applications. It features tracing of applications on UNIX platforms in C, C++ and Fortran supporting MPI, OpenMP and hybrid parallelism. It includes support for automatic code instrumentation and a sophisticated run-time measurement library. VampirTrace is developed at the Center for Information Services and High Performance Computing (ZIH) at Technische Universität Dresden in collaboration with the KOJAK project of research center Jülich [1] and is available under a BSD open source license.

Vampir is an interactive trace visualization and analysis tool developed at ZIH, TU Dresden. It allows detailed post-mortem investigation of dynamic parallel run-time behavior as well as statistical summaries of arbitrary intervals of run-time [2,3]. The successor version VampirServer uses a client-server approach with distributed processing of trace data that allows an interactive work-flow for very large data sets [4].

2.1 Performance Counter Support

Even though VampirTrace focuses on event tracing and collecting event-specific information, it utilizes additional statistical information about dynamic run-time performance. Most notably, it supports the PAPI performance counter library, that defines a common interface for reading hardware performance counters [5]. On one hand, it makes common performance counters available with standard names on almost all platforms. This includes the counters for floating point operations or cache misses/hits for various cache levels. On the other hand, PAPI allows to query a huge number of platform specific performance counters that are rarely used. Yet, sometimes such counters allow insight into very special performance issues, as shown in this paper.

2.2 Application Specific and System Wide I/O Tracing

Besides the classical targets of performance analysis computation and communication, another important component of HPC applications is input/output (I/O) from/to files on mass storage systems. In particular, with expanding storage sizes and working sets, the time spent for data accesses on the storage system makes up more and more of the total application run time. Yet, the speed of storage systems does not increase accordingly. Like the so called *memory wall* [6] that inhibits faster computation because of inadequately slow memory accesses there is a similar *input/output wall* for accesses to the storage systems.

This makes the analysis and optimization of the I/O behavior increasingly important. This is especially true for data-intensive applications that scale well with the number of processors. Usually, such codes scale with a constant working set per CPU. Thus the data to be transferred to/from the storage system grows linearly with the degree of parallelism exceeding the I/O capacity eventually.

Therefore, the recent development of VampirTrace contains some approaches for gaining insight into dynamic I/O behavior of parallel applications [7,8]. This includes instrumentation and tracing of application-level I/O calls as well as system-wide I/O throughput of the global SAN (Storage Area Network) infrastructure. The former is important for detailed examination of user-space I/O requests. The latter is necessary to include the effects from concurrent I/O activities of other applications – the SAN infrastructure cannot be used exclusively like CPU or (parts of) the communication infrastructure (to some extent).

3 Tracing and Analysis of the Semtex Application

Semtex is a parallel CFD code which scales very well over 512 CPUs [9]. It is very data intensive and is used regularly for highly parallel and long-running simulations on the HPC infrastructure of ZIH.

Semtex employs an integrated checkpoint/restart mechanism in order to divide a single simulation into convenient sections that fit well into the batch system policy and make it robust against system failures. Additionally, multiple checkpoints retrieved at small intervals can be used to visualize the simulation. This allows for verification of the correctness and refinement of the simulation, respectively.

The checkpoint mechanism saves the complete parallel working set of a simulation after a given number of time step iterations. For a visualization of the simulation checkpoints are written every 200 iterations. If no visualization is needed, the simulation runs for 5000 time steps (ca. 4h real-time on 128 CPUs) after which a checkpoint is taken that is used as starting point by the next job.

Most of the simulations running at ZIH use a working set suitable for running on 128 processors that results in checkpoints of 5 gigabytes in size. The less often used next larger working set has checkpoints occupying 20 GB of disk space.

As simulations carried out with Semtex account for a large share of the CPU hours used per year at ZIH, its I/O behavior has been subject to closer performance analysis.

3.1 Instrumentation and Tracing of Semtex

For the analysis Semtex was at first instrumented using the automatic compiler instrumentation offered by VampirTrace. A tracing run on 128 CPUs took 2h 16min for 2000 time steps including 10 checkpoints, and the trace data accumulates to 56 gigabytes – including function calls, MPI specific information and extensive I/O records for both, per-process I/O calls as well as system-wide I/O throughput records. This data provides fine-grained information about the application. Figure 1 shows a section of the overall run-time including two checkpoint phases which can easily be identified.

Although VampirServer handles such large traces without problems, the overwhelming details make an analysis cumbersome. The level of detail (and hence the size of trace files) was reduced by employing VampirTrace's filter abilities which allow to record only a given number of calls per function or to skip certain functions completely.

Nevertheless, the overhead caused by the automatic instrumentation still was very large – tracing time was about 2.5 times of original runtime.[1] As a

[1] This is a known problem when compilers have to decide whether to instrument or inline a function – the Intel compilers used in this example favor instrumentation over inlining, therefore much performance is lost.

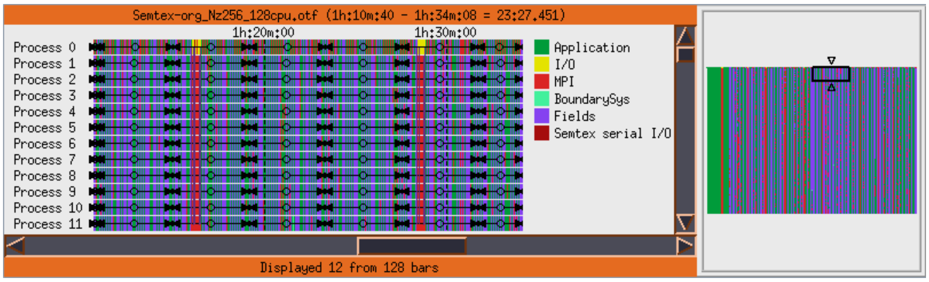

Fig. 1. Vampir's Process Timeline display for a typical execution of Semtex

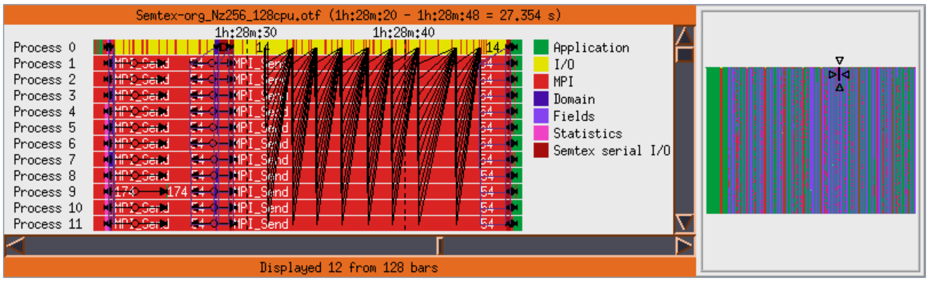

Fig. 2. Timeline display zoomed to a single checkpoint phase

consequence, the code was manually instrumented and afterwards, the tracing introduced only a marginal overhead.

However, the automatic instrumentation was by no means useless. The available tracing data delivered invaluable insights into the workflow of Semtex that were used to quickly identify interesting source code locations where instrumentation calls have been inserted.

3.2 I/O-Related Performance Bottlenecks

For investigation of I/O related performance we looked closer on the checkpoint phases in between the time step simulation. Figure 2 shows a zoomed time line for a typical checkpoint phase. Process 0 is dominated by I/O activity shown in yellow while the remaining processes spend this time in MPI calls. This is a classic *single writer* situation, where one process is collecting all data from its peers via message passing in order to write it to the file system.

This scheme is obviously unfavorable for a massively parallel program and limits the otherwise good performance. In particular, it inhibits scalability as checkpoint phases will grow linearly for growing CPU counts.

Further investigating the I/O behavior of Process 0 revealed additional performance problems. All checkpoint phases turned out to follow a very regular pattern of receiving and writing constant sized blocks of approximately 1.4 MB

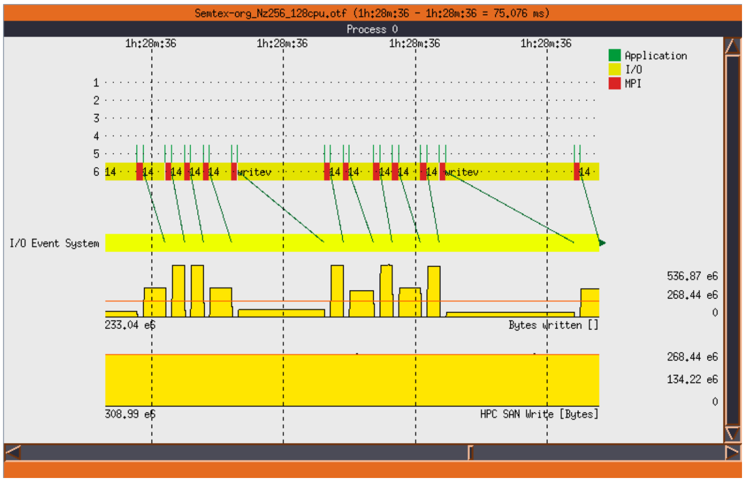

Fig. 3. Regular pattern of MPI receive (red) and write (yellow) activities with notable variation in throughput. The upper counter shows the process-related write speed while the lower one shows the global SAN throughput.

as shown in Fig. 3. Yet, the single write operations show quite differing speed as shown in the *bytes written* performance counter, compare Sect. 2.2.

When inspecting I/O performance of any single process it is important to consider the current utilization of the SAN infrastructure. Usually, the I/O network is not exclusively used by an application but globally shared. Therefore, the effect could have been caused from outside, i.e. any other application with extensive I/O utilization. This is not the case in this example. The comparison of the local counter *bytes written* and the global one *HPC SAN write* in Fig. 3 shows the same I/O throughput on average. Furthermore, the global I/O speed, which is only available with a resolution of one sample per second (compare Sect. 2.2), is almost constant during all checkpoint phases.

3.3 Optimization of I/O Performance

Our analysis of I/O behavior revealed two bottlenecks: On one hand, the *single writer* problem, and on the other hand the fluctuating local write speed.

Both of these problems were solved by modifications to the checkpoint code. From the trace analysis we learnt the following facts:

– The speed of single write calls of constant size is heavily fluctuating. Even though most calls are fast, the regularly occuring very slow calls destroy the over-all I/O performance – compare peaks in *bytes written* counter in Fig. 3.
– The speed of SAN I/O is the same during all checkpoint phases and near zero just before and after these phases which confirms the reliability of the application's I/O measurements.

– The time step iteration is interrupted by checkpoint phases from time to time. Therefore, every I/O phase is followed by a computation phase without I/O activity – see Fig. 1.

This led to the following hypothesis: The delays in I/O API calls on the application level are caused by the caching I/O subsystem of the operating system. Two different solutions are available to eliminate this problem. The first is using direct I/O which bypasses the operating system's file cache. Yet, it is rather difficult to implement as special alignments and request sizes have to be used. The second solution does not suffer from those restrictions. Since the write-only scheme of checkpointing does not require instant write to disk, asynchronous I/O can be used. This has the advantage of decoupling the I/O calls, that happen during the checkpoint phases, and the actual I/O operations issued by the operating system, that may be performed concurrent to the following computation phase. Furthermore, with modifying the checkpoint routine so that each process writes its data on its own to the checkpoint file, the single writer problem is addressed as well.

Table 1. Comparison of original and optimized Semtex checkpoint phases

# CPUs	Checkpointing Time (% of total runtime) Original version		Optimized version		Improvement
8	12.1 s	(1.3%)	6.3 s	(0.9%)	47.9%
128	106.8 s	(8.4%)	35.5 s	(3.9%)	66.8%
256	381.7 s	(12.8%)	107.7 s	(5.1%)	71.8%

Table 2. Comparison of checkpoint times, *intermediate checkpoint phases only*

# CPUs	Checkpointing Time (% of total runtime) Original version		Optimized version		Improvement
8	7.5 s	(0.8%)	3.1 s	(0.4%)	58.7%
128	64.7 s	(5.1%)	15.8 s	(1.8%)	75.6%
256	249.3 s	(8.4%)	37.0 s	(1.7%)	85.2%

Based on this hypothesis the checkpoint code of the Semtex application was modified to use asynchronous MPI I/O functions. This yielded an improvement in checkpointing speed of up to 85%, compare Tables 1 and 2.

Table 1 shows the times spent for checkpointing including the final checkpoint. There the gain from asynchronous I/O is not as large as within intermediate checkpoints (see Table 2). This comes from the fact that the application must wait for completion of the I/O within the last checkpoint whereas this is not necessary for intermediate checkpoints.

Table 2 further shows that the proportionate time needed for intermediate checkpoints does not grow when increasing the number of processes from 128 to 256 (1.8% to 1.7%) which underlines the potential of asynchronous I/O.

3.4 Performance Problems in Memory Copy Operations

In the course of the performance evaluation of Semtex's I/O activities, an exceptional high fraction of almost 40% was observed to be spent for communication during time step iterations. Figure 4 shows the profile of one such iteration. One time step needs 1.5 seconds to complete from which 0.59 seconds are used for data exchange. A look into the source code revealed that besides MPI functions, the only other time-consuming calls could be those to `memcpy` which were then enclosed by tracing calls for making them available for analysis (already included in Fig. 4).

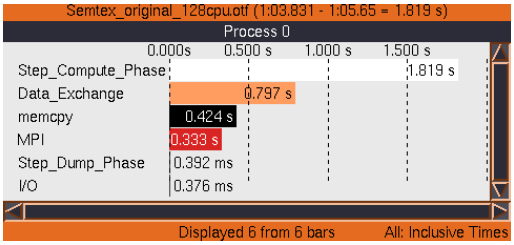

Fig. 4. A major part of one simulation step is spent on data exchange, which `memcpy` is responsible for, besides MPI

Further investigation showed that the speed of `memcpy` calls is not constant. On Process 0 the first call is faster than the remaining ones, compare Fig. 5. The same happens for Process 1 whereas on Processes 2 and 3, the third call is faster.

Apart from the difference in speed the copy operations themselves are surprisingly slow. Each `memcpy` call copies 22 kBytes of data in 35 μs (fast case) or in 165 μs (slow case) resulting in transfer rates of 640 MB/s and 130 MB/s, respectively – the underlying architecture would allow for much more.

We found a combination of three phenomena responsible for these effects:

1. Glibc's `memcpy` is not tuned for the Itanium architecture which causes the copy operations to be rather slow.
2. *Single-copy transfers* of SGI's Message Passing Toolkit (MPT, the MPI library in use) introduce hidden remote memory accesses.
3. The different speeds of memory copy operations arise from characteristics of the Altix 4700 architecture in conjunction with the second phenomenon.

To understand the latter, both the workflow of the data exchange code and the architecture of the Altix 4700 have to be taken into account. The Semtex application uses a special data exchange scheme between all processes, which is performed multiple times during every time step iteration of the simulation. It is implemented as follows:

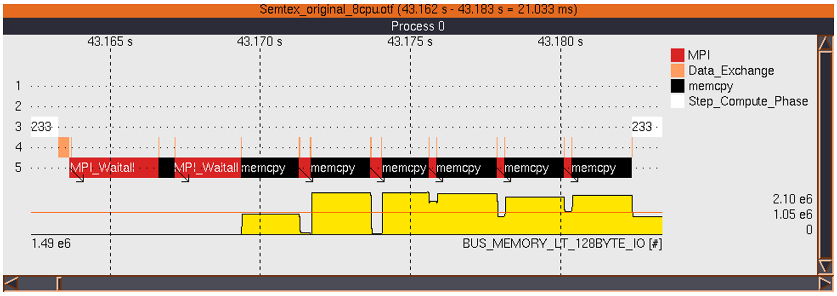

Fig. 5. The Itanium's hardware performance counter indicates unoptimized memory accesses from remote processors

1. Allocate a temporary buffer for data exchange.
2. For each of the other processes do the following:
   - Send own buffer to partner via `MPI_Isend`.
   - Receive data from partner via `MPI_Irecv` into the temporary buffer.
   - Wait for completion of the communication via `MPI_Waitall`.
   - Copy contents of temporary buffer to own buffer.

Therefore, Process 0 exchanges data with Process 1 first, then with Process 2, 3, etc. The sequence is analogous for Process 1. Process 2 communicates first with Process 0, then with Process 1, 3, 4, etc.

Figure 5 shows the details of one complete exchange on Process 0 for a small run with 8 CPUs. The send-receive-copy pattern is executed for every peer process, i.e. seven times in this example.

From this algorithm and the architecture of the Altix 4700, which consists of dual-core Intel Itanium 2 processors[2], we can conclude that the faster `memcpy` calls belong to the data exchange happening between the cores of one CPU.

The findings so far suggest that the `memcpy` calls after communication with processes located on other CPUs access remote memory and are therefore slower. This is somewhat counter-intuitive because the code looks as if the copy routine would access two local buffers – the temporary one and the permanent one.

Yet, the behavior can be explained considering the second phenomenon. For saving memory bandwidth, MPT maps the communication buffers into the peer's address space and then uses a single copy operation to transfer the contents to the destination buffer [10]. Unfortunately, the memory pages of the receiver's temporary buffer seem to be located at the remote party after this operation.

Evidence for this hypothesis is given by a special hardware performance counter of the Itanium 2 processor. Below the time line, Fig. 5 shows the rate of `BUS_MEMORY_LT_128BYTE_IO`, which counts the number of less than full cache line[3] transactions from remote parties. The counter shows a high rate indicating excessive remote memory accesses that transfer less than 128 bytes. Those

[2] At ZIH, in particular, there is one CPU per system board.

[3] L2 and L3 cache line size is 128 bytes on the Itanium 2 processor.

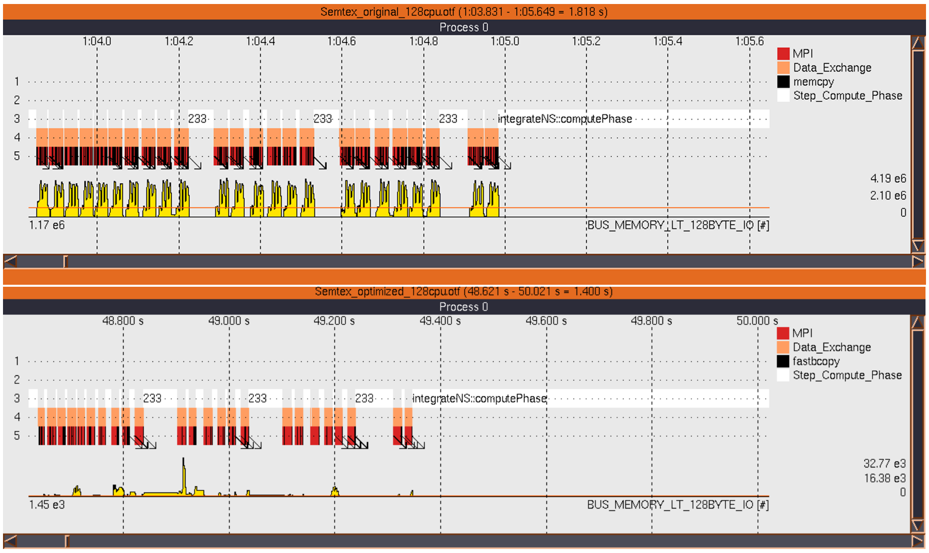

Fig. 6. Comparison of the original data exchange pattern (top) with the optimized counterpart (bottom) for Process 0 for the 20'th time step iteration. Note the much smaller duration of the optimized version (1.4s vs. 1.8s) and the significantly reduced value of the counter showing small-sized remote memory accesses.

have double negative effects on transfer speed. Firstly, smaller transfer sizes mean more remote accesses are needed for copying the same amount of data, and secondly, each remote access suffers from higher latencies compared to local memory accesses. This in turn leads to the poor performance of `memcpy` when it eventually accesses remote memory.

For Process 0, the counter is low for the first `memcpy` belonging to the data exchange with Process 1, and rises afterwards indicating ineffective accesses to remote memory caused by `memcpy`. Looking at Process 2 (not shown here), the third `memcpy` shows no peak, and this scheme continues to the last process.

3.5 Optimization of Communication Scheme

The performance of the data exchange code could be dramatically improved by replacing the `memcpy` calls with the highly optimized `fastbcopy` call available from MPT. This routine achieves copying speeds of 4500 MB/s for local memory and 1300 MB/s when remote memory is involved.

A comparison of the original version with the optimized one is shown in Fig. 6. The architectural optimization of the `fastbcopy` routine is depicted by the `BUS_MEMORY_LT_128BYTE_IO` counter which shows very low rates in the optimized version (lower picture). Usage of this routine doubled the speed of data exchange which allowed for more than 20% improvement in run-time per time step iteration. Together with the I/O optimization, the total runtime of Semtex was reduced by 25%.

4 Conclusion and Outlook

This paper presented two interesting aspects of the performance analysis process for the Semtex application. The proposed optimization steps were confirmed with notable performance improvements for this application. Since this code is used for long-term simulations with large degree of parallelism on the HPC resources of ZIH, TU Dresden, the optimization accounts for a substantial number of CPU hours saved!

Further work will focus on the I/O tracing components of VampirTrace, in particular the availability of system-wide I/O monitoring in a platform independent manner.

References

1. Wolf, F., Mohr, B.: KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications (Demonstrations of Parallel and Distributed Computing). In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 1301–1304. Springer, Heidelberg (2003)
2. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. In: Supercomputer 63, vol. XII(1), pp. 69–80 (1996)
3. Brunst, H., Winkler, M., Nagel, W.E., Hoppe, H.-C.: Performance optimization for large scale computing: The scalable VAMPIR approach. In: Alexandrov, V.N., Dongarra, J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) ICCS 2001. LNCS, vol. 2074, pp. 751–760. Springer, Heidelberg (2001)
4. Brunst, H., Nagel, W.E.: Scalable Performance Analysis of Parallel Systems: Concepts and Experiences. In: Joubert, G.R., Nagel, W.E., Peters, F.J., Walter, W.V. (eds.) Parallel Computing: Software, Alghorithms, Architectures Applications, pp. 737–744. Elsevier, Amsterdam (2003)
5. PAPI group: Papi - performance application programming interface, `http://icl.cs.utk.edu/papi/`
6. Wulf, W.A., McKee, S.A.: Hitting the memory wall: Implications of the obvious. Computer Architecture News 23(1), 20–24 (1995)
7. Müller, M.S., Knüpfer, A., Jurenz, M., Lieber, M., Brunst, H., Mix, H., Nagel, W.E.: Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In: Bischof, C., Bücker, M., Gibbon, P., Joubert, G., Lippert, T., Mohr, B., Peters, F. (eds.) Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing, vol. 15, pp. 637–644. IOS Press, Amsterdam (2007) ISBN 978-1-58603-796-3
8. Mickler, H.: Kombinierte Messung und Analyse von Programmspuren und systemweiten I/O-Ereignissen. Master's thesis, Technische Universität Dresden (2007)
9. Blackburn, H.M., Sherwin, S.J.: Formulation of a Galerkin spectral element-fourier method for three-dimensional incompressible flows in cylindrical geometries. J. Comput. Phys. 197(2), 759–778 (2004)
10. SGI: Linux Application Tuning Guide, `http://techpubs.sgi.com/library/manuals/4000/007-4639-008/pdf/007-4639-008.pdf`