BACHELORARBEIT

Energy-Aware Instrumentation of Parallel MPI Applications

Universität Hamburg Fakultät für Mathematik, Informatik und Naturwissenschaften Fachbereich Informatik

Autor: Studiengang: Matrikelnummer: E-Mail: Fachemester:

Erstgutachter: Zweitgutachter: Betreuer: Florian Ehmke Informatik 6053142 8ehmke@informatik.uni-hamburg.de 8

Prof. Dr. Thomas Ludwig Prof. Dr. Winfried Lamersdorf Timo Minartz

Hamburg, 25. Juni 2012

Erklärung

Ich versiche, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen — benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Departments Informatik einverstanden

Hamburg, 25. Juni 2012

Abstract

Energy consumption in High Performance Computing has become a major topic. Thus various approaches to improve the performance per watt have been developed. One way is to instrument an application with instructions that change the idle and performance states of the hardware.

The major purpose of this thesis is to demonstrate the potential savings by instrumenting parallel message passing applications. For successful instrumentation critical regions in terms of performance and power consumption have to be identified. Most scientific applications can be divided into phases that utilize different parts of the hardware. The goal is to conserve energy by switching the hardware to different states depending on the workload in a specific phase. To identify those phases two tracing tools are used. Two examples will be instrumented: a parallel earth simulation model written in Fortran and a parallel partial differential equation solver written in C.

Instrumented applications should consume less energy but may also show a increase in runtime. It is discussed if it is worthwhile to make a compromise in that case. The applications are analyzed and instrumented on two x64 architectures. Differences concerning runtime and power consumption are investigated.

Contents

1	Intr 1.1	oducti Appro	on ach	$\frac{1}{2}$
2	Rela	ated W	Vork	4
3	Har	dware	Management	5
	3.1	Introd	uction to CPU governors	6
	3.2	Manua	al device state management	7
4	Pha	se Ide	ntification	11
	4.1	Descri	ption of the tracing and visualization environment	12
		4.1.1	HDTrace and Sunshot	14
		4.1.2	VampirTrace and Vampir	16
	4.2	Test a	pplications	18
		4.2.1	partdiff-par - partial differential equation solver	18
		4.2.2	GETM - General Estuarine Transport Model	21
	4.3	Relate	ed problems	23
		4.3.1	Overhead caused by tracing the application	23
		4.3.2	Size of the trace files	24
		4.3.3	Runtime variations	25
5	Inst	rumen	ntation of the Applications	26
	5.1	Test h	ardware	26
	5.2	partdi	ff-par: instrumentation and measurements	26
	5.3	GETN	I: reorganization of ncdf_sync	32
6	Con	clusio	n and Future Work	34

List of Figures

1.1	Draft of application behaviour to look for in traces.	2
3.1	eeDaemon overview	7
4.1	Tracing infrastructure	13
4.2	Main window of Sunshot	15
4.3	Detailed info for timeline elements	15
4.4	Main window of <i>Vampir</i>	16
4.5	Zoomed in timeline	17
4.6	MPI communication visualized in <i>Vampir</i>	17
4.7	partdiff-par phases	19
4.8	Communication during 1 iteration of the calculation phase	19
4.9	Communication during 1 iteration (highlighted area in figure 4.8)	19
4.10	I/O phase of partdiff-par	20
4.11	Trace of <i>GETM</i> in <i>Vampir</i> (<i>ondemand</i> governor)	21
4.12	Trace in <i>Vampir</i> with many flushes (blue areas)	23
4.13	Trace in <i>Vampir</i> with increased buffer size	24
4.14	Call to save_2d_ncdf which lasted much longer than previous ones	25
5.1	Trace of an instrumented 1 node job on an intel node	27
5.2	Utilization of the network when writing a checkpoint.	28
5.3	Length of an MPI_Sendrecv call used to exchange line data	28
5.4	Relative measurements of different CPU settings. Baseline is the fixed maximum	
	frequency setup. The setup is 1_node (see table 5.1)	29
5.5	Relative measurements of different CPU settings. Baseline is the fixed maximum	
	frequency setup. The setup is 4_nodes_artificial (see table 5.1)	30
5.6	Relative measurements of different CPU settings. Baseline is the fixed maximum	
	frequency setup. The setup is 4_nodes_realistic (see table 5.1)	30
5.7	Trace of GETM with reorganized ncdf_sync in Vampir	32

List of Tables

5.1	Overview of different setups for <i>partdiff-par</i>	27
5.2	Overhead caused by instrumentation of the CPU (10 runs each, one <i>Intel</i>	
	node). During the instrumented runs the 4 idle cores were set to the	
	highest P-State.	32
5.3	Measured values for new version (10 runs each)	33

Chapter 1 Introduction

The computational needs for science, industry and many other segments is growing since decades. Long ago the performance offered by a single machine stopped being enough — computers were clustered to drastically increase the performance. Today supercomputers consist of hundreds of nodes build in huge racks. The nodes are connected with high performance networks like $Infiniband^1$ or $Myrinet^2$. To unlock the potential of these supercomputers applications have to be parallelized. One way to parallelize applications on a large scale is to use the Message Passing Interface (MPI)³. The MPI standard specifies a library that contains several functions to exchange data between processes or to accomplish collective I/O.

The incredibly high demand for performance in High Performance Computing (HPC) will most likely not change soon. More performance requires more energy, a costly resource. Often the acquisition cost of a supercomputer is caught up by the maintenance costs after a few years. Hence lately the energy footprint of a new supercomputer plays an increasingly large role next to the actual performance of the system. The Sequoia supercomputer currently on rank one of the Top 500 list⁴ has a power consumption of 7890 kW. Rank two of that list, the K computer, draws even more power: 12659.9 kW (enough to power more than 10.000 suburban homes). The Sequoia supercomputer is not only 55 percent faster but also 150 percent more efficient in terms of energy. This shows that much research in this area is conducted.

Supercomputers are working at maximum utilization most of the time. Sometimes a few nodes are idle, but modern schedulers do their best to backfill those. This leaves very little room to conserve energy on an existing system. In desktop computing, especially on mobile devices, many hardware components are able to adjust their power consumption to a certain workload. Most of the time these adjustments do not affect the performance. The system still feels responsive and the user doesn't even notice that something has changed. But in High Performance Computing, where every cycle of the central processing unit (CPU) counts, this is not desired. Automatic changes to adjust to a workload have a major drawback. The adjustments are always late. If the CPU switches to a lower frequency because the system is idle, it does that after the system has gone idle,

¹http://www.infinibandta.org/

²http://www.myricom.com/

³http://www.mcs.anl.gov/research/projects/mpi/

⁴http://www.top500.org/

and not the moment it goes idle. Ideally there shouldn't be any idle times in HPC, but this isn't the case. While the applications running on the cluster do work all of the time, this is usually not true for every component of the utilized nodes. Scientific applications usually have different phases during their execution. Input data has to be processed before the calculation can start. The calculation phase gets interrupted by communication phases and at last the results have to be written to the disk. During the I/O or the communication phase the CPU is usually not utilized at full extent. During the calculation phases on the other hand the network interface controller and disk are often idle. These are exactly the starting points for automatic power saving in desktop and mobile computing — but not yet in HPC.

1.1 Approach

Our approach is to switch the hardware into the right (in terms of power consumption) mode, at the right time (without loosing performance). Briefly worded the approach is to analyze applications for interesting phases (for example an I/O phase) and then instrument those in the source code with the result that during these phases power saving modes are utilized. The analysis of an application can be tricky — especially parallel applications are sometimes hard to understand. For that purpose tools that visualize the flow of control of such applications as well as hardware utilization are used.



Figure 1.1: Draft of application behaviour to look for in traces.

The tracing tools are thereby used to look for application behaviour similar to that sketched in figure 1.1. Phases during which the utilization of a hardware component is low, indicating that it can potentially do the same work in a lower performance state. This is done with two different applications. Once the interesting phases of those applications are identified they are instrumented. Instructions are added to the source code that initiate device mode changes before such a phase starts. Ideally the frequency graph in figure 1.1 would look exactly like the utilization graph. To control the hardware a daemon is running on every node the application is started. The instructions are sent to that daemon, which then decides if the device mode change can be executed. If another application requires a higher device mode, the change won't be executed.

The next chapter will start by presenting some related work in this field. In order to improve energy efficiency lots of work focuses around the dynamic voltage and frequency scaling of the processor. The general direction of the presented work is to improve the prediction of workload. In chapter 3 the different power saving modes of processor, hard disk drive and network interface controller are described. In the course of that, the software used to manage the device modes is introduced. Chapter 4 is about the software suites used for tracing and visualization and lists the test applications that are used in this thesis. Example traces are used to explain the usage of both tracing tools. After that the two test applications are traced and analyzed for interesting phases. In chapter 5 the previously discovered phases of the test applications are instrumented. Two different x64 architectures are used to evaluate the instrumentation. Chapter 6 concludes this thesis and presents ideas for future work.

Chapter 2

Related Work

In order to reach exascale computing a lot of research is being conducted. Much of it deals with dynamic voltage and frequency scaling (DVFS) of the processor. CPU MISER (CPU Management Infrastructure for Energy Reduction) is a run-time power aware DVFS scheduler [6]. The scheduling is completely automated and requires no user intervention. It has as an integrated performance prediction model that allows the user to specify acceptable performance loss for an application relative to application peak performance. CPU MISER predicts workload, for example communication and memory access phases, and lowers the CPU frequency accordingly. Experimental results have shown that this can save up to 20% energy with 4% performance loss. Another DVFS scheduler is Adagio [14]. It is an online scheduler that predicts computation time based on a current stack trace. It extracts information about MPI calls from that trace and then predicts the next MPI call. This information is then used for processor scheduling. Adaqio aims for significant energy savings with negligible performance loss (less than one percent). [5] proposes low power versions of two collective MPI functions that utilize DVFS. In particular those functions are MPI_Gather and MPI_Scatter. During these functions the CPU exhibits computational idle phases. These phases are then used to scale down the cpu frequency and voltage in order to save energy. The experimental results show that in case of low power MPI_Gather it was possible to save 45.9% energy and for low power MPI_Scatter it was even 55.7%. In [4] the potential of DVFS is analyzed. It is shown that the potential for energy savings with DVFS has significantly diminished in newer CPU technologies.

In [15] an alternative *Linux* CPU frequency governor is introduced. Other than the common governors *ondemand* and *conservative* the *pe-Governor* uses hardware performance counters to make decisions (as opposed to the CPU load). These decisions are designed to run the workload as power efficient as possible. More precisely the used metric is instructions per memory access. Test results show that the *pe-Governor* in average increases the runtime by 1.58% while the energy consumption gets reduced by 2.37%.

Chapter 3

Hardware Management

In the first part of this chapter the different power saving modes of processor, hard disk drive and network interface controller are presented. Further we explain why these modes are often disabled in high performance computing although they are enabled and used in desktop computing. In the course of that terminology like *Turbo Boost* and CPU governors are introduced. The next part discusses how the *energy efficient Daemon* (*eeDaemon*) can be used to utilize power saving modes via manual code instrumentation in high performance computing.

Most modern hardware components are capable of changing their performance to adjust to a certain workload. The benefit of this is to conserve power. The central processing unit (CPU) has several performance states (P-States) and operating states (C-States) for this purpose [2]. A CPU P-State represents an operating frequency and an associated voltage. Increased P-States mean lower operating frequencies and thus lower power consumption and performance. C-States are another measure to conserve power. The default operating state is C0 which means that no components of the CPU are shut down. If the CPU is idle it is possible to gradually turn off more and more components of the CPU by switching to higher C-States. The downside is that as more components are turned off the time needed to return to C0 increases.

Hard disk drives (HDDs) offer three different modes. The first mode is active/idle which is the normal operation mode. The second mode is standby (low power mode) which means that the drive has spun down and the last mode is sleeping. In this mode the HDD is completely shut down.

Common network interface-controllers (NICs) can switch between different transmission rates. If for example the fastest rate is Gigabit Ethernet (1000 Mbit/s) then Fast Ethernet (100 Mbit/s) and Ethernet (10 Mbit/s) can be used to reduce the power consumption. The power consumption difference of these three modes is however hardly noticable (around 1 Watt) which makes the NIC the least interesting component to conserve power.

In normal desktop computers switching between available performance modes is depending on the workload. The operating system decides which states of the CPU shall be used at a certain point of time. There are different so called governors which make different descisions at the same workload [13].

In high-performance computing (HPC) this behaviour is often not desired. When for

example the HDD enters a sleep mode it would take seconds to go back into the normal operating mode. In parallel applications this could lead to serious delays and thus these energy saving features are disabled to maximize the performance.

The *eeDaemon* allows programmers to directly control hardware by instrumenting their existing code. This has many advantages and is particularly interesting if the application has phases during which the CPU is less utilized or the HDD could enter a sleep mode. Usually the hardware would remain in the mode offering the highest performance. Using the *eeDaemon* a programmer can instrument the code responsible for an I/O phase so that the CPU enters a higher P-State before the I/O phase and goes back into the fastest P-State after the I/O phase. In the same matter the HDD would wake up / spin up just in time for the I/O phase and go back to standby afterwards. Of course these instrumentations have to be in the right place so that the modes are switched at the right time. This is especially important in case of the HDD where switching modes needs more time (in contrast to the CPU).

3.1 Introduction to CPU governors

The *Linux CPUfreq* subsystem allows it to dynamically scale the CPU frequency. The *CPUfreq* system uses governors to manage the frequency of each CPU. Different governors may make different decisions at the same workload [13]:

ondemand The ondemand governor is the default governor and dynamically sets the frequency based on the current workload. During idle phases the CPU will rest in the lowest frequency. When the current load surpasses a specified threshold the ondemand governor will switch the CPU to the highest frequency available. Once the load falls below that threshold the ondemand governor will switch to the next lowest frequency and continue to do so until the lowest frequency is reached (if the load stays below the threshold).

powersave The powersave governor will keep the CPU at the lowest frequency.

performance The performance governor will keep the CPU at the highest frequency.

- **conservative** The *conservative* governor works like the *ondemand* governor, based on the current workload, but it increases the frequency more gradually (decreasing is the same). The *conservative* governor only switches to the next highest frequency (once the load is higher than the threshold) and not to the highest frequency. The frequency will be continually increased as long as the load stays above the threshold until the highest frequency is reached.
- *userspace* The *userspace* governor allows the user to take full control over the CPU and it's P-States.

Newer technologies Newer Intel CPUs have a special P-State called Turbo Boost¹ [3]. The CPU activates this mode if high load is present and the CPU is running in the

¹Newer AMD CPUs have a similar technology called Turbo Core.

lowest P-State (P0). The *Turbo Boost* itself has several states depending on the CPU model. If load is present on every core the *Turbo Boost* won't be used, it is designed for scenarios where some cores are idle and others are under heavy load. In that case the active cores will be overclocked. The highest *Turbo Boost* will only be used if only one core is active and all other cores are idle.

3.2 Manual device state management



Figure 3.1: *eeDaemon* overview

The *eeDaemon* provides a programming interface to explicitly manage device power modes by manual instrumentation [12]. It is completely written in the C programming language and consists of a client library and a server process. The client library offers the necessary functions to manage the hardware and can be linked dynamically to the application. A server process must be running on every cluster node running the application. The client library sends the information to the server process which then decides which power state every device should use. This way only the server process must be executed in kernel space. If more than one application is running on one node the server process will prevent interferences between the two and use only modes that would not affect runtime of each application.

Device Modes

The eeDaemon offers 5 different modes that are all applicable to any device [10]:

MODE_TURBO Mode marking a very high utilization for the device - device must be switched to the highest performance mode.

- **MODE_MAX** Mode marking a high utilization for the device device must be switched to the high performance mode.
- **MODE_MED** Mode marking a mid-range utilization for the device if possible, device can be switched to a mid-range performance mode.
- **MODE MIN** Mode marking a low utilization for the device if possible, device can be switched to a low performance mode.
- **MODE_UNUSED** Mode marking the device as unused which means a device can possibly be switched to sleep.

It has to be kept in mind that not every device offers 5 different modes. A common HDD for example can spin down (MODE_MIN) and sleep (MODE_UNUSED). But there are no further performance modes which would map to MODE_TURBO, MODE_MAX or MODE_MED. Thus all these modes do the same — they wake the disk up if it was previously in MODE_MIN or MODE_UNUSED [10].

General Usage

The *eeDaemon* library interface provides two different methods to initialize an application on a cluster. Upon initialization applications have to provide a tag. That tag is used to register the application at the server and allows the server to tell the running applications apart. This tag can be provided by the programmer using the function ee_{init} . However it is important to make sure that the tag doesn't collide with other applications. Alternatively the more convenient function ee_{init_rms} can be used. This function reads the tag from an environment variable set by the Resource Management System (RMS). In our case this RMS is *Torque*¹ and the tag will be set to the jobid specified by *Torque*. It is necessary for the server to be able to distinguish the running applications. Obiviously it's not desired that application one is able to reduce the CPU frequency while application two is in a computational phase. That's why the server only sets a device to a lower power state if every (registered) application running on a certain node previously issued that particular change.

Changing the device modes from within the code can be done with the function ee_dev_mode. This initiates a device mode change to one of the device modes presented in section 3.2. The mode change will be initiated without any delay, however the device may take some time to finish the device mode change. In case of the CPU this is usually no problem, but a HDD or NIC can take several seconds to change the mode. To cope with that problem the *eeDaemon* provides the function ee_dev_mode_in(int device_id, int mode_id, int secs) which allows the programmer to specify that a completed device mode change is needed in secs seconds. If an application is structured in iterations and one iteration takes 1 second we could call ee_dev_mode_in(HDD, MODE_MAX, 100) before the calculation starts to indicate that we need a certain device state in iteration 100. This could be for example the HDD which is needed for an I/O phase in iteration 100 but idle in the other iterations.

Before the application exits one has to call ee_finalize to properly unregister the application at the server.

¹http://www.adaptivecomputing.com/products/open-source/torque/

Fortran wrapper for the eeDaemon The eeDaemon is written in the C programming language and thus can only be used in applications written in the C programming language itself. Many of the scientific applications in which the eeDaemon would be applicable are written in *Fortran*. It is possible to call C code from within *Fortran* applications. To achieve that functionality a wrapper for the eeDaemon interface was implemented in the course of this thesis.

Implementation

```
Listing 3.1: C function prototype of ee_init_rms
      /**
1
       * Distincts the tag by reading the environment variable containing
2
          the resource
       * management system jobid.
3
4
       *
       * Calls ee_init(). See ee_init() for details.
5
6
       * @param argc Pointer to count of commandline args
7
       * @param argv Pointer to commandline args
8
       * @param rank Rank for this process, e.g. the MPI rank
9
       */
10
      void ee_init_rms(int *argc, char ***argv, int rank);
11
  Listing 3.1: C function prototype of ee_init_rms
```

Listing 3.1 shows the prototype of the function $ee_init_rms()$ which is typically used to initialize the *eeDaemon* when an application is started by a resource management system like *Torque*. To achieve the same functionality in a *Fortran* application using the *eeDaemon* with it's *Fortran*-Interface a few more steps are needed. In a *C*-Application the needed argument vector which contains the program name and command-line arguments is directly available. *Fortran* has no direct equivalent to the *C* argument vector and thus the *Fortran*-version of $ee_init_rms()$ looks a little different.

```
Listing 3.2: Interface to ee_init_rms_fortran(), a wrapper for ee_init_rms
1
      INTERFACE
2
          SUBROUTINE EE_INIT_RMS (NAME, RANK) BIND(C, NAME='
             ee_init_rms_fortran')
              USE ISO_C_BINDING
3
              IMPLICIT NONE
4
              CHARACTER (KIND=C CHAR) :: NAME (*)
5
              INTEGER (C_INT), VALUE
                                        :: RANK
6
          END SUBROUTINE EE INIT RMS
7
     END INTERFACE
8
 Listing 3.2: Interface to ee_init_rms_fortran(), a wrapper for ee_init_rms
```

The Fortran interface for the *eeDaemon* uses a wrapper function as shown in listing 3.2. The function ee_init_rms() needs the C argument vector only for the program name, argc is not used. Therefore the Fortran function only has 2 arguments: NAME and RANK. NAME should be the same as the corresponding argv[0] (in C) and RANK should be the rank provided by the MPI library.

```
Listing 3.3: eeDaemon initialization

1 call get_command(program_name)

2 program_name=trim(program_name)//C_NULL_CHAR

3 call ee_init_rms(program_name, rank)

Listing 3.3: eeDaemon initialization
```

Since Fortran 2003 there is a new intrinsic module called iso_c_binding which makes it a lot easier to access C code from Fortran. As shown in listing 3.3 line 2 the string provided by get_command() can be passed to a C application as long as the necessary null character (\0) is appended via //C_NULL_CHAR. Further usage is not different to C applications using the *eeDaemon*. The functions in the Fortran interface eed_f have the same names as in C.

This chapter focused on hardware management in terms of power consumption and performance. Almost every device in a modern computer has its own ways to adjust the power consumption to a certain workload. It was explained why these capabilities are disabled (most of the time) in HPC — to avoid negative impact on the performance. The introduced *eeDaemon* provides a consistent interface to control the CPU, HDD and NIC from within an application. The programmer can decide whether or not power saving modes should be used. This explicit management reduces performance loss while power is conserved.

Chapter 4 Phase Identification

The previous chapter focused on device modes and how they can be used. It was described how manual code instrumentation can be used to utilize those modes in order to conserve power. This chapter is about the identification of phases that are suitable for that purpose. The key for optimal instrumentation is timing. If the correct power state for a certain phase in an application is applied too late, or too early, the overall result won't be better or maybe even worse. To aid in identifying the interesting phases during the execution of applications two different tracing tools are used. A tracing tool records information while the application is running and saves this information in so-called trace files. These trace files include things like function calls, time spent in functions, values of variables, hardware utilization etc.

An application can be traced synchronously or asynchronously. For example, it is a synchronous trace to record when a function call starts and when it ends. Those are two distinct events that also inherit the information how long the function call lasted $(time_{end} - time_{start})$. Tracing the function calls asynchronously would mean to check every interval seconds in which function the application is currently working. Periodically reading and storing the current cpu frequency is asynchronous. The cpu frequency could also be traced synchronously (every frequency change is one event). The advantage of doing this asynchronously is that it creates less overhead. Tracing every CPU frequency change would (in case of a governor that dynamically changes the frequency) create much more events. Additionally the overhead would be unsteady, there would be phases with lots of frequency changes, and phases with little to no changes. On the contrary tracing asynchronously most likely looses information. The frequency could change an unknown amount of times between two measuring points. Generally speaking the advantage of synchronously tracing is that no event is missed, but it can create a high overhead. Asynchronous traces create a controlled amount of overhead, but can be inaccurate. Asynchronous and synchronous trace files are not incompatible to each other. They can be synchronized using recorded timestamps.

There are text based tracing tools but also tools that generate more complex data which can then be visualized with a trace viewer. Text based tracing tools usually create less overhead and are easier to use and setup. Tools that are able to visualize the data are more complex but can provide more insight. In this thesis the latter is used. Having a graphic record of the program execution can also help debugging applications. Especially in the field of parallel programming understanding the flow of control can be complicated. In such a case (instead of manually inspecting the code) looking at the graphic representation can help to identify problems. Some tracing tools can visualize trace data while the application is running (online). In this thesis offline trace viewers are used which visualize the data after the execution.

Theory Tracing applications has various purposes. It can aid in debugging applications, it can help to identify bottlenecks and it can simply help understanding a program better. In this work tracing is explicitly used to identify phases that are interesting for instrumentation. The obvious things to look for are communication and I/O phases. The knowledge that these phases exist isn't enough. It is mandatory that the phase is exposed enough to be instrumented. That means the phases shouldn't overlap. A communication phase could be implemented in such a way, that the actual MPI calls return immediately (non-blocking) and the computation continues with little to no interruption by the communication. The data will be sent through the network in any way, with the difference that the actual data isn't tangible for instrumentation if it's implemented non-blocking. So one has to make sure that the instrumentation doesn't have a negative impact on the performance by ruling out that the computational phase and the communication (or I/O) phase overlap. For that purpose a visualization of the program execution is very helpful.

This chapter starts with a description of the tracing and visualization environment that is going to be used. Two tracing tools are presented and by use of example traces their functionality is explained. In the course of that it is shown how the generated graphs can be interpreted. In the following section the two applications that are going to be used in this work are introduced. With the help of the tracing tools phases of interest in those applications are identified.

4.1 Description of the tracing and visualization environment

In parallel applications it is sometimes not trivial to identify phases which would be obvious in serial applications. It is very helpful to have a graphical representation of concurrent events as opposed to looking at traditional logfiles or the code itself.

For that reason two different tracing tools are used which should help identifying interesting phases, find problems and evaluate the results. The first tracing tool is *HDTrace* which visualizes the MPI communication of different MPI Processes as well as system information like hardware utilization, power consumption, network and I/O. *HDTrace* is licensed under the GPL license and developed at the University of Hamburg in the department Scientific Computing. *HDTrace* consists of libraries that generate trace files and *Sunshot* which is then used to visualize those traces.

The second tracing tool is *Vampir* which is a proprietary trace viewer that can visualize trace data of different formats including the Open Trace Format (OTF). To generate the necessary OTF trace files *VampirTrace* is used. *VampirTrace* is developed at ZIH Dresden in collaboration with the KOJAK project and licensed under the BSD Open Source license.



Figure 4.1: Tracing infrastructure

4.1.1 HDTrace and Sunshot

HDTrace consists of several different components (see fig. 4.1 for a selection of components used in this thesis) [11]. Especially interesting are the components that trace calls of the MPI library as well as the *Resource Utilization Tracing Library (RUT)* and the *PowerTracer*. The *RUT* is used to periodically gather information about the hardware utilization and started as a daemon on every cluster node the traced application is running. The *PowerTracer* is also running as a daemon but on the master node on which no calculation is done. It pulls the power consumption of each node from LMG450devices. Both the *PowerTracer* and the *RUT* store the data in a database. The data in this database is then used to populate the trace files after the execution. This reduces the overhead of tracing. In case of the *PowerTracer* no overhead at all is generated (because everything is done on the master node). The *RUT* daemon however does create overhead, the utilization data has to be sent through the network. This overhead could be severely reduced by utilizing a service network (different from the network used for normal applications).

To generate trace files for an application run the application has to be linked against the libraries of *HDTrace*. Upon execution the application will then generate 3 types of files [9]:

- .trc The generated .trc files contain the MPI events in XML format. Each rank has its own .trc file that stores the MPI events that occured during the execution of the application. To each entry of an MPI event in that file belongs a start and end timestamp.
- .stat These files contain external statistics in a binary format gathered for example from the *Resource Utilization Tracing Library*. They are used to store data like CPU utilization and power consumption. The data is collected periodically (asynchronous) and upon visualization synchronized with the .trc files via timestamps.
- .info The .info files contain structural information such as MPI data types.

Once these files are present a project file (.proj) has to be generated. This is done with a python script (project-description-merger.py) that needs the .info files as input data. That .proj file can then be used to open the trace with *Sunshot*, the trace viewer of *HDTrace*.

Example Trace

Figure 4.2 shows the main window of Sunshot. To the left one can see the names of the different timelines. The first timelines are representing the activities of the MPI library. Each process on each node has its own timeline. In this example one node with 8 processes was used. Below the MPI timeline external statistics from the .stat files are shown. Hardware components like the main memory, each CPU core, the NIC and the HDD each can have several timelines indicating their utilization at a certain point during the application execution. When looking at that data it has to be kept in mind that the data is collected periodically. This is particularly important for the CPU frequency timelines. The CPU frequency can change very fast and very often in a short period.



Figure 4.2: Main window of Sunshot

If such a sequence of frequency changes happens between two measuring points of the *Resource Utilization Tracing Library*, and before and after it the same frequency was used, *Sunshot* would show a constant frequency for that period of time. In this example only the average CPU utilization and frequency for all cores are shown, it is however possible to show the data for each core individually.



Figure 4.3: Detailed info for timeline elements

The elements shown in the MPI timelines can be right-clicked to show detailed information as can be seen in figure 4.3. Information like the exact duration, the timestamp when the function call was executed, involved ranks and files and the exact function name is shown. For functions like MPI_File_write it also shows the amount of data written, the file name and the offset that was used for writing the file.



Figure 4.4: Main window of Vampir

4.1.2 VampirTrace and Vampir

Vampir is a proprietary trace file viewer that supports different trace file formats. In this thesis the OpenTraceFormat (OTF) is used [8]. The trace files will be generated if a special compiler wrapper shipped with VampirTrace is used (for example mpicc-vt or mpif90-vt). These wrappers then trace user functions as well as MPI events at execution time and store them in trace files (.z). This naturally causes overhead. In section 4.3.1 ways to deal with the overhead at execution time as well as exceptionally huge trace files are presented. Additionally external statistics like hardware utilization can be integrated (see figure 4.1) using the VampirTrace Plugin Interface [16]. After the program execution the trace files can be viewed with Vampir.

Example Trace

Figure 4.4 shows the main window of *Vampir*. To the top right one can see the main timeline of the application run. It shows a histogram of the time spent per function group. The window "Function Summary" shows that in this example 97 seconds were spent in functions of the application, 91 seconds using functions of the MPI library and 41 seconds were used for the *VampirTrace* library. The 4 graphs in the top left corner which are named "Process 0-3" show timelines of the function calls of each process that participated in executing the application. Aligned to these timelines in the window below is an additional chart that in this case shows the power consumption over time. Other possible charts are for example the cpu utilization or the the cpu frequency over time. These charts can be shown at the same time.







Figure 4.6: MPI communication visualized in Vampir

It is possible to zoom in on an area of the main timeline which will affect all other charts. As one can see in figure 4.5 the power consumption chart is now more precise and in the process timeline the function names are shown. The areas representing function calls can be clicked and then show information like call duration, interval, name and involved processes in the "Context View" to the right. This is similar to the detailed info in *Sunshot* (see section 4.3).

Vampir furthermore visualizes the MPI events. If process one sends data to process two by use of MPI_Send and MPI_Recv the two calls will be connected with a black line in the process view. The relations are also clickable. Figure 4.6 shows such a communication phase. It can be seen that process three receives data from process one (through MPI_Isend) but process three is further ahead and thus has to wait for process one. As soon as the call to MPI_Waitall finishes process 3 receives the data (the function name is not shown, because the MPI_Irecv call is too short).

4.2 Test applications

Two different applications were used in the scope of this thesis. One written in C and one *Fortran* application. The first application is *partdiff-par*, a partial differential equation solver parallelized using MPI. The *Fortran* application *GETM* is a scientific model also parallelized in MPI.

4.2.1 partdiff-par - partial differential equation solver

partdiff-par is a parallel differential equation solver. The program has several input parameters which allow to use it as a benchmark as well as an application that behaves very similar to "real" scientific applications. It is very easy to create scenarios that represent realistic workload and/or artificial I/O heavy scenarios. partdiff-par uses the Jacobi method to solve the system of linear equations. The application runs through a userspecified amount of iterations (alternatively it is possible to specify a desired precision for the result, the calculation will stop if the precision is reached). Each participating process gets an equal share of the matrix. The matrix is distributed line by line (every process has one contiguous set of lines). Each iteration consists of a calculation phase and a communication phase. During such a communication phase the lines needed to continue the calculation in the next iteration are exchanged. Additionally, the application can perform checkpoints which will result in an I/O phase. The checkpoints are written using MPI I/O functions. MPI I/O provides an I/O interface for parallel MPI programs. Using MPI I/O is much faster than normal, sequential I/O and also enables the MPI library to apply further optimizations. During such a checkpoint the complete matrix is dumped. Every process writes its share of the matrix into the checkpoint file.

Parameters The most important parameters of *partdiff-par* are listed below:

- interlines This parameter specifies the size of the matrix that is going to be solved. With 1000 interlines a matrix with the dimension 8008 will be calculated which uses 0.513 gigabytes memory. The memory usage of the matrix doesn't grow linearly but exponentially with the specified interlines.
- **iterations** Specifies the amount of iterations that will be calculated. More iterations means a higher precision of the result but also a higher runtime.
- checkpoint iterations Specifies the number of iterations before a checkpoint is
 written. For example, if iterations is set to 100 and checkpoint iterations
 to 40 the complete matrix will be written to the disk in iteration 40 and 80.
- visualization iterations Same as checkpoint iterations but instead of a checkpoint the visualization data is written. Writing this data takes much less time than writing a checkpoint (because only the matrix diagonal is written). This parameter will always be set to the same value as checkpoint iterations to simplify matters.



Figure 4.7: partdiff-par phases

Phases In figure 4.7 one can see the different phases during execution of partdiff-par. In this figure only the MPI activities are shown.

Initialization During the initialization phase the MPI library as well as the matrix and some global variables are initialized. This phase is extremely short and therefore not interesting for our purpose.



Figure 4.8: Communication during 1 iteration of the calculation phase



Figure 4.9: Communication during 1 iteration (highlighted area in figure 4.8)

Iteration The matrix is calculated spread across the participating ranks. Each iteration consists of a calculation phase and a communication phase. Before the calculation starts the different ranks have to communicate with each other to acquire the necessary data for the calculation. The matrix is distributed between the ranks line by line. A matrix with 8 lines calculated by 4 ranks would be distributed as follows: line 1-2: rank1, line 3-4: rank2, line 5-6: rank3, line 7-8: rank4. Each rank only has to communicate with his direct neighbours. In this example rank2 would have to communicate with rank1 and rank3 after each iteration. The communication is implemented using MPI_Sendrecv(). Figure 4.8 and figure 4.9 visualize that rank0 and rank7 call MPI_Sendrecv() only once per phase because they have only one direct neighbour to communicate with.



Figure 4.10: I/O phase of partdiff-par

- I/O phase Every <checkpoint iterations> an I/O phase takes place during which a checkpoint is written. Figure 4.10 shows the MPI calls during this phase as well as relevant hardware utilization. In that trace the ondemand governor was used (see section 3.1). Most of the time during MPI_File_write_at calls the governor set the CPUs to high P-States. This is clearly visible when looking for example at the graph of timeline CPU_FREQ_AVG_2 which shows the clock speed of core two. Notable is that during calls to MPI_File_close the utilization of the CPU is at 100% and thus the ondemand governor does not set the CPU to a higher P-State. That's because MPI_File_close is a collective operation and for example rank0 spends 95% of the I/O phase just with waiting for other ranks to finish their MPI_File_write_at calls so that they can finish the collective MPI_File_close operation. This can be seen in the timelines CPU_FREQ_AVG_0, CPU_TOTAL_0 (utilization of core zero) and the MPI timeline of rank0.
- **Finalization** In this phase the MPI library will be finalized and every rank sends some data to rank0 which then visualizes the matrix. For the purpose of conserving energy this phase is not interesting as it's almost as short as the initialization phase.

4.2.2 GETM - General Estuarine Transport Model

The short form *GETM* stands for *General Estuarine Transport Model* [1][7]. *GETM* is a three dimensional MPI parallelized modular *Fortran 90/95* model which can be used among others to simulate tides for the Sylt-Rømø Bight. *GETM* requires *NetCDF*¹ input data and writes the output data through *NetCDF* as well. *NetCDF* is a short form for *Network Common Data Form*, a set of libraries and a (open, cross platform) file format to exchange scientific data. *GETM* comes with several setups. Each setup represents a different case that is going to be simulated. In the course of this thesis the setup box_cartesian is used. The box_cartesian setup can be run sequentially or parallel with 4 MPI processes.

Phases To identify the phases of interest in this case only *Vampir* was used. It would have been possible with *Sunshot* as well but due to the internal structure of *GETM* the written trace files by *HDTrace* quickly exceed magnitudes that do no longer fit into the main memory when the trace files are opened with *Sunshot*. *VampirTrace* offers more flexibility in this case. Figure 4.11(a) shows a trace of *GETM* using the *ondemand* governor on one *Intel* node. The main window shows 2 additional graphs:

intel2_util_cpu_freq_avg_0 the cpu frequency over time.

intel2_power the power consumption over time.



(a) Both the CPU frequency and the power consumption graph are very unsteady.

(b) Calls to save_<2d|3d>_ncdf interrupt the calculation in every iteration.

Figure 4.11: Trace of *GETM* in *Vampir* (ondemand governor).

Both of these graphs appear very unsteady which is very suspicious. Figure 4.11(b) reveals the reason for this unsteadiness. The functions save_2d_ncdf and save_3d_ncdf are called frequently. These functions are obviously I/O functions. This is a quite

¹http://www.unidata.ucar.edu/software/netcdf/

unattractive pattern for instrumentation. These calls are very short which causes the overhead of the instrumentation to shadow the actual gain of executing these phases at a lower CPU frequency. To the right in figure 4.11(b) some information about one call to save_2d_ncdf is shown. That particular call lasted only 91.9 ms. In section 5.3 the assumption that it is not feasible to instrument these calls will be validated.

To see how the model performs without tracing it 10 runs with 4 MPI processes on an *Intel* node were performed. During these runs the model calculated 10 days of the input data which are split into 86400 timesteps (iterations). The 10 runs averaged for about 223 seconds execution time. That is 387 iterations per second. Every 10 iterations save_2d_ncdf is called and every 70 iterations save_3d_ncdf which means they both are executed several times each second. Both of these subroutines end with a call to nf90_sync (found out after inspecting the source code files save_2d_ncdf.F90 and save_3d_ncdf.F90) which synchronizes the *NetCDF* data in the main memory with the data on the HDD.

4.3 Related problems

Using tracing tools to identify phases or just to debug an application can sometimes be problematic. Naturally compiling and running an application linked against a trace library causes overhead. More code needs to be executed and trace files have to be written. This overhead can eventually choke off the benefits of using such tools.

4.3.1 Overhead caused by tracing the application

The overhead that originates from tracing the application calls and writing the trace files is a serious problem which can't be ignored. When for example in a trace several calls to MPI_Wait appear to be very long this doesn't have to mean that these calls have the same length when executing the application without trace libraries. Maybe these MPI_Wait's only exist because one process is writing trace files while others have already finished or didn't even need to.



Figure 4.12: Trace in *Vampir* with many flushes (blue areas).

Figure 4.12 shows the master timeline of a *Vampir* trace with the default buffer size (32 M). The buffer is used to store all kinds of recorded events. Once it is full the data has to be written on the disk (flushed). The application ran for 72 seconds and as one can see much time was spent in calls of the MPI library (red areas). When looking at the process timeline it becomes clearly visible that between the 25 seconds and the 65 seconds mark the buffer flushes (blue) of the *VampirTrace* library stopped being synchronized which introduced very long calls to MPI_Waitall.

Vampir offers some configuration options to cope with the overhead [17]. For instance it is possible to manually instrument the source code. With manual instrumentation it is possible to reduce the amount of events that are traced. When less things are traced, the buffer doesn't fill up so fast. That way the amount of long buffer flushes can be reduced. To apply manual instrumentation the application has to be compiled with -DVTRACE. It can be used together with the automatic compiler instrumentation or without. To use only manual intrumentation the VT compiler wrapper needs the option -vt:inst manual. This is ideal to reduce the overhead because it allows to simply skip the tracing of sections of no interest. This flexibility makes it possible to have different tracing scenarios like I/O phases, initialization or calculation and in each run only the interesting sections will be traced and thus the buffer doesn't get jammed with needless data.

```
Listing 4.1: VampirTrace manual instrumentation

    #include "vt_user.h"
    VT_USER_START("name");
    ...
    VT_USER_END("name");
    Listing 4.1: VampirTrace manual instrumentation
```

Additionally, it is possible to completely turn off (and on again) the tracing by using the VT_OFF() and VT_ON() macros. By default *VampirTrace* stops tracing as soon as it's buffer is full for a second time (flushed once), that means nothing after that point will be traced. This is often not enough for a complete trace. To change this behaviour two environment variables can be changed: VT_BUFFER_SIZE and VT_MAX_FLUSHES. To get a complete trace VT_MAX_FLUSHES must be 0 or something high enough that *VampirTrace* doesn't stop tracing. Unfortunately flushing the buffer takes a considerable amount of time (the buffer is written to the disk) and is able to "ruin" traces (see figure 4.12). To guarantee that interesting parts of the trace don't get interrupted by a buffer flush it is possible to manually initiate a buffer flush by calling VT_BUFFER_FLUSH().



Figure 4.13: Trace in *Vampir* with increased buffer size.

Figure 4.13 shows a trace of the same application with the same parameters as in 4.12. The only thing that has been changed is the VT_BUFFER_SIZE (from 32 Mb to 768 Mb). As can be seen the time spent in the *VampirTrace* library has been reduced significantly which also led to much less time being spent in the MPI library.

4.3.2 Size of the trace files

Another problem similar to the overhead created by tracing applications is the size of the generated trace files. Depending on the traced application the file size can exceed several gigabytes very fast. This is a problem for several reasons. On the one hand the trace file viewers *Sunshot* and *Vampir* may not be able to visualize the trace because they can't fit the data into their main memory and on the other hand these large files may not even fit onto the specific HDD (less likely). Most solutions presented in section 4.3.1 also reduce the trace file size.

4.3.3 Runtime variations

The previous problems were solely caused by tracing the application. Runtime variations however also appear when executing the application normally. This is a problem that affects not only the identification of phases of interest by tracing the application but also the normally executed runs. The variations go up to 20% which is a serious problem because it means that the scope of these variations exceeds the expected results. These variations have many causes. One cause is the usage of the *Network File System (NFS)*. If an applications writes data on a *NFS* volume and at the same time another application is also writing data naturally the results will be different compared to an exclusive access. This problem can be easily solved by making sure that no other users or applications are utilizing the *NFS* volume. But there are also other causes that are not as apparent and whose impact on the results can only be minimized by repeatedly measuring again and the elemination of evident outliers.



Figure 4.14: Call to save_2d_ncdf which lasted much longer than previous ones.

Runtime variations are not restricted to multi node jobs. Figure 4.14 shows a trace of *GETM* during which one call to save_2d_ncdf for some reason lasted much longer than previous and subsequent ones. As so often when one process is spending more time during a function call than the other participating processes he slows down the whole process group at the next call to MPI_Waitall or similar functions like for example MPI_Barrier.

Tracing and identification of relevant phases was the main topic of this chapter. Terms like asynchronous and synchronous tracing, text based versus grahpic traces and online/offline visualization were explained. Two different tracing suites were introduced, both with an offline graphic visualization tool. The usage of both tools was described by use of the two test applications *partdiff-par* and *GETM*. Both applications were analyzed for phases that can potentially be instrumented by the *eeDaemon* which is the topic of the next chapter. The last part of this chapter described related problems that occured during the usage of the tracing tools.

Chapter 5

Instrumentation of the Applications

The previous chapter focused on the analyzation of the two test applications *partdiff-par* and *GETM*. This chapter is about the instrumentation of these applications. At first the cluster on which the applications are tested is described. The next sections focus on the instrumentation of the phases identified in chapter 4. The *eeDaemon* is then used to utilize the present device modes of the test hardware as described in chapter 3.

5.1 Test hardware

The eeClust (energy efficient cluster) consists of ten nodes. Five of these nodes are powered by an AMD CPU (Opteron 6168 @ 1.900 MHz), the other five nodes by an Intel CPU (Xeon Nehalem X5560 @ 2.800 MHz). The Intel nodes have 12 Gb of main memory, the AMD nodes have 32 Gb. Two switches are used for networking. An Allnet 4806W takes care of the service network (IPMI) while a D-Link DGS-1210-48 is used for all the other networking tasks. The power consumption of every node is measured through a LMG 450 Power Meter and stored in a database every 100 ms. One NAS nodes provides the necessary storage capacity for jobs with very large input and output data. It is important to distinguish between jobs that write on the NAS systems and jobs that write on the master-node that stores the home directories because their performance is different which could lead to corrupted test results.

5.2 partdiff-par: instrumentation and measurements

In partdiff-par the I/O phase identified in section 4.2.1 was instrumented. The CPU was set to MODE_MIN during the I/O phase (writing a checkpoint and the visualization data). During the other phases the CPU was set to MODE_MAX. Additionally some tests with MODE_TURBO instead of MODE_MAX were made. The runs without instrumentation were made in four (three for AMD) different CPU frequency settings. Once with the ondemand governor and for comparability with fixed frequencies set to the minimum frequency available on the specific node as well as the maximum frequency and the Turbo Boost (only on Intel). Neither the NIC, nor the HDD was instrumented — doing so could have saved a couple of watts but we focused on the CPU.

jobname	interlines	iterations	checkpoint	processes	nodes
1_node	3000	40	30	8	intel1
1_node_amd	4500	40	30	24	amd1
4_nodes_artificial	1500	250	120	32	intel1-4
4_nodes_artificial_amd	1500	250	120	96	amd1-4
4_nodes_realistic	1500	4000	1500	32	intel1-4
4_nodes_realistic_amd	1500	4000	1500	96	amd1-4

Table 5.1: Overview of different setups for *partdiff-par*

Setups Table 5.1 shows an overview of the different setups that were used for *partdiffpar*. Both the 1_node setup and the 4_nodes_artificial setup are more a benchmark than a realistic scenario that is likely to happen in the real world. However these are still useful to analyze the behaviour of the application and the cluster. The 4_nodes_realistic scenario has a much lower I/O - calculation ratio and can therefore be considered as a realistic example.



Figure 5.1: Trace of an instrumented 1 node job on an intel node

Trace Figure 5.1 visualizes how the behaviour of the hardware changes (with instrumentation) compared to the trace with the *ondemand* governor shown in section 4.2.1 (Figure 4.10). In area one it is clearly visible that the CPUs remains in the highest P-State throughout the whole I/O phase although towards the end of it most CPUs are actually at 100% utilization. This is interesting because the *ondemand* governor would interpret that CPU utilization as load and shift the CPUs to lower P-States resulting in a higher power consumption when in fact, the only thing those processes do is actively waiting for other processes to finish writing data. This can safely be done in the highest P-State without loosing too much performance. Area three shows the drastic results for the power consumption (again compared to the *ondemand* governor). Lastly area two shows that during the I/O phase indeed I/O is happening (as opposed to data being

cached and written later on). That behaviour is not optimal in terms of performance. The calculation could continue once the checkpoint data is cached (and not yet completely sent) since the completion of the checkpointing is not actually required for the calculation.



Figure 5.2: Utilization of the network when writing a checkpoint.

In figure 5.2 it can be seen that during 4-node jobs the data of the checkpoint written with MPI_File_write_at is sent over the network only during the actual "I/O" phase and not cached and sent later (during calculation phases).



Figure 5.3: Length of an MPI_Sendrecv call used to exchange line data.

Length of the communication and calculation phase In *partdiff-par* only the I/O phase was instrumented. Although communication of the line data between ranks takes up a considerable amount of time it is not feasible to instrument these phases. A rather long MPI_Sendrecv call (that is used to exchange line data) lasts around 0.1 seconds as one can see in figure 5.3. This problem also exists in *GETM* (see paragraph 5.3). In *partdiff-par* however with enough main memory (and an appropriate amount of interlines) it would be possible reach regions where these MPI_Sendrecv calls last considerably longer. Under these circumstances it would be feasable to apply instrumentation to the communication.

Measurements (runtime, energy, and power consumption) Every setup was executed 15 times, after what evident outliers were eliminated. The *Turbo Boost* results are sometimes hard to interpret. That is because one has no guarantee that the *Turbo Boost* will be used although the lowest P-State is active. That descision isn't made by the operating system but by the CPU.



Figure 5.4: Relative measurements of different CPU settings. Baseline is the fixed maximum frequency setup. The setup is 1_node (see table 5.1).

Figure 5.4(a) shows how the different CPU settings compare to a fixed frequency set to 2,8 GHz. It can be seen that the minimum frequency needs considerably longer (24%) while only 16% power consumption is saved. This results in an increased energy consumption. The *ondemand* governor shows an increase in power consumption similar to the Turbo Boost setup. This indicates that the ondemand governor switched to the lower P-State and the *Turbo Boost* was utilized, otherwise the power consumption wouldn't be so much higher than the maximum frequency. That drastic power consumption increase results in a much higher energy consumption although the runtime is only increased by six percent. The instrumented runs also show an increase in runtime (three percent) but the by ten percent decreased power consumption outweighs this increase which results in a eight percent decrease in energy consumption. The jobs for AMD shown in figure 5.4(b) performed different compared to *Intel*. The minimum frequency (800 MHz) shows an increase in runtime of 80% compared to the maximum frequency (1900 MHz). This is more than three times the increase that was measured on *Intel*. Since the 1_node setup is very I/O heavy this leads to the assumption that by reducing the CPU frequency also the memory bandwidth suffers. The results of the instrumented runs match with this theory.

Figure 5.5(a) visualizes the results for the 4_node_artificial jobs. In this setup the network was utilized during the checkpoint phase. Notable is that the min setup saved energy although the runtime increased by nine percent. This is not very much which indicates that utilizing the network doesn't need much CPU power although the packages have to be prepared and packed before they can be sent. Since the min setup conserved energy it isn't surprising that the instrumented setup was able to achieve the same.



Figure 5.5: Relative measurements of different CPU settings. Baseline is the fixed maximum frequency setup. The setup is 4_nodes_artificial (see table 5.1).

The AMD graphs shown in figure 5.5(b) look very different to those in figure 5.4(b). Reducing the CPU frequency thus doesn't affect the network performance. That results in energy savings for both the min and the instrumented setup.



Figure 5.6: Relative measurements of different CPU settings. Baseline is the fixed maximum frequency setup. The setup is 4_nodes_realistic (see table 5.1).

The runs of the setup 4_nodes_realistic of both AMD and Intel are visualized in figure 5.6. The first thing that stands out is that the runtime of the minimum frequency settings is much higher than in the previous setups. In case of Intel it shows an increase in runtime by 82% and for AMD it is with 128% even higher. These runtimes result in much higher energy consumptions. In addition, it is clearly visible that any setting that involved the *Turbo Boost* (ondemand, turbo, instrumented (turbo)) has a much higher energy consumption. Although the runtime is decreased by five to eight percent the much higher power consumption (20 to 24 percent) results in around 15% more energy consumption. The instrumented setup looks very similar to the fixed maximum frequency because the I/O phase is rather short compared to the calculation phase. The ondemand governor on AMD performed similar to the instrumented setup and the fixed maximum frequency. Since on the AMD architecture no *Turbo Boost* like feature is available this is as expected.

The "artificial" setups 1_node and 4_node_artificial showed savings in energy consumption of five to eight percent. The "realistic" setup showed similar results to the fixed maximum frequency. This was expected since the share of instrumented execution time was very small in this setup (much computation). This doesn't mean, that in "realistic" cases nothing can be saved. There are certainly applications that perform relative amounts of I/O or communication closer to the artificial setups. In these I/O or communication heavy setups potential savings with only reasonable performance loss exist. The *Turbo Boost* has proven to be very inefficient for our setups. Although sometimes the runtime was decreased, that gain was outweighed by the much higher power consumption resulting in an increased energy consumption. In serial workload however, where the load is distributed very uneven, it may be worth using the *Turbo Boost*. The used *AMD* architecture wasn't suitable for I/O instrumentation. The huge decrease in performance caused by using the highest P-State leads to the assumption that the memory bandwidth is decreased along with the CPU frequency.

5.3 GETM: reorganization of ncdf_sync

As presented in section 4.2.2 the structure of GETMs phases is unfortunate for the purpose of this thesis. The naive approach to just instrument the calls to ncdf_sync which undertake I/O and thus don't need much CPU time fails because there are simply to many calls in short periods.

Table 5.2: Overhead caused by instrumentation of the CPU (10 runs each, one *Intel* node). During the instrumented runs the 4 idle cores were set to the highest P-State.

setup	runtime	power	energy	
default	$222.673\mathrm{s}$	$221.194\mathrm{W}$	$48524.8\mathrm{J}$	
instrumented	$245.016\mathrm{s}$	$217.272\mathrm{W}$	$56230.5\mathrm{J}$	

Overhead Table 5.2 shows that indeed the overhead is too large when instrumenting the I/O phases in *GETM*. Although the mean power consumption is slightly lower (about 4 W) the runtime increase (10%) is just too large and causes the overall consumed energy to rise severely.



Figure 5.7: Trace of *GETM* with reorganized ncdf_sync in *Vampir*

ncdf_sync only every 24 hours (model time) Calling ncdf_sync that often makes sense to a certain degree. If the program execution crashes due to hardware failure or something similar, the data should be unaffected since it is already written to the disk. This makes it possible to restart the calculation at the last time ncdf_sync was called and only very little calculated data could be lost (at most data of 9 iterations). The question arises if it is really necessary to sync that often. At this point one has to weigh things up. For testing purpose the save_2d_ncdf and save_3d_ncdf routines have been modified to only call ncdf_sync every 24 hours (model time) which then are instrumented to run at the lowest CPU frequency possible.

The impact of this modification together with the instrumentation becomes clearly visible when looking at traces of this version. Figure 5.7(a) shows a much more plain pattern and less unsteadiness in the charts. In figure 5.7(b) it can be seen how the power consumption drops along with clocking down the CPU.

setup	runtime	power	energy
default	$64.6035\mathrm{s}$	$195.594\mathrm{W}$	$12612.2\mathrm{J}$
instrumented	$64.0554\mathrm{s}$	$196.412\mathrm{W}$	$12559.7\mathrm{J}$

Table 5.3: Measured values for new version (10 runs each)

Table 5.3 shows the measured values (power consumption, runtime and energy consumption) for the new version (sync every 24 hours). It can be seen that there is no longer an overhead between the instrumented and the default version. However there is also no measurable gain in energy efficiency. This is due to the fact that now that the call to $ncdf_sync$ only happens every 24 hours (model time) the overall time spent with I/O is too low in contrast to the time spent with communication and calculation. Nevertheless it is remarkable how the runtime has changed towards the old version. It now averages at around 64s as opposed to 222s (see table 5.2).

The original internal structure of GETM was unsuited for instrumentation. The amount of iterations per seconds was too high — the overhead worsens the results. Reorganization of the I/O phase decreased the runtime, created a better suited structure for instrumentation but also decreased the relative amount of I/O in one execution of GETM. There is no longer an overhead due to instrumentation, but the results are not measurable. It is likely that setups other than the used box_cartesian have longer communication phases. The biggest constraint is that only 4 MPI processes could be used. More processes would mean longer communication phases.

This chapter applied the techniques presented in chapter 3 to conserve energy during the phases identified in chapter 4. The results show that reducing the CPU frequency on the used AMD architecture isn't feasible during local I/O phases. On the *Intel* architecture however this showed the best results of all three used setups (up to eight percent). During communication phases however it was possible to conserve energy on both architectures, but not as much as during the local I/O. This indicates that the process of preparing the data before it can be sent utilizes the CPU more than I/O. As for *GETM*, reorganizing the I/O phase resulted in theoretical savings, unfortunately nothing measurable. The I/O phase duration was too short compared to the time spent in calculation phases. Executing *GETM* on a larger productive cluster however should show measurable results as only 4 processes don't introduce long enough communication phases.

Chapter 6

Conclusion and Future Work

This thesis focused on improving the energy efficiency by using idle and performance states of hardware. In HPC performance is no longer the only important metric, energy efficiency plays an increasingly large role. Newer supercomputers not only surpass their predecessors in terms of performance but also in energy efficiency. As can be seen in mobile and desktop computing much power can be conserved when the system is idle. Because in HPC slowing down applications isn't desired, functionalities that automatically use these power saving modes are usually disabled. In HPC most of the time only one application per time is running on one node. Therefore one can be relatively sure that during phases that stress the HDD or the NIC the CPU is not working at maximum capacity and can potentially perform the same work in the same time with a lower operating frequency. Such phases were instrumented in this work using the *eeDaemon* with the result that the CPU switches to a lower frequency. In order to analyze applications for interesting phases, and to verify that the instrumentation works as intended, tracing tools can be used. Two graphical tracing suites were used for that purpose. The instrumentation and tracing was carried out on two different applications (one written in C and one in *Fortran*) and on two different x64 architectures.

With manual instrumentation of high performance applications it is possible to conserve energy by using device idle states without harming the performance too much. The looked-for opportunities to utilize these idle states can be identified with the help of tracing tools. Although the overhead of tracing applications can be challenging, the gained insight proved to be very valueable. The identified phases were successfully instrumented and it was possible to conserve energy. However there are things to look out for; in our case instrumentation of the I/O phase was counterproductive on the used AMDarchitecture. The increased performance when using the *Turbo Boost* did not justify the severely increased power consumption. To conclude, utilizing idle and performance states with code instructions is a powerful measure that can be worth the effort; but thorough evaluation is very important — if the instructions don't fit to the program's phases the results can be awfully bad.

Future work includes evaluation of the presented methods on larger clusters and the instrumentation of NIC and HDD. Our test applications are not optimal for the test cluster. It would be advantageous to test *partdiff-par* and *GETM* on a larger productive

cluster. Executing applications with thousands of processes also introduces much longer communication and I/O phases — these are the bottlenecks that work against scalability of parallel programs. In theory, this promises good results. Furthermore, applications that exhibit longer, more complex communication schemes can be evaluated and the poor I/O performance of the used AMD Magny-Cours architecture in higher P-States has to be analyzed.

Bibliography

- [1] Hans Burchard, Karsten Bolding, and Lars Umlauf. General Estuarine Transport Model - Source Code and Test Case Documentation. Tech. rep. 2011.
- [2] Hewlett-Packard Corporation et al. Advanced Configuration and Power Interface Specification. 2011. URL: http://www.acpi.info/.
- [3] Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. 2008.
- [4] Gaurav Dhiman, Kishore Kumar Pusukuri, and Tajana Rosing. "Analysis of Dynamic Voltage Scaling for System Level Energy Management". In: *HotPower'08 Proceedings of the 2008 conference on Power aware computing and systems*. San Diego, California: USENIX Association, 2008, pp. 9–14. URL: http://dl.acm. org/citation.cfm?id=1855610.1855619.
- [5] Yong Dong et al. "Low Power Optimization for MPI Collective Operations". In: International Conference for Young Computer Scientists (2008), pp. 1047–1052.
 DOI: http://doi.ieeecomputersociety.org/10.1109/ICYCS.2008.
 500.
- [6] Rong Ge et al. "CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters". In: *ICPP '07 Proceedings of the 2007 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 18–25. ISBN: 0-7695-2933-X. DOI: http://dx.doi.org/10.1109/ ICPP.2007.29.
- [7] GETM General Estuarine Ocean Model. June 2012. URL: http://http:// getm.eu/.
- [8] Andreas Knüpfer et al. "Introducing the Open Trace Format OTF". In: Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part II. Lecture Notes in Computer Science 3992. Graduate University of the Chinese Academy of Sciences. UK: Springer-Verlag GmbH, 2006, pp. 526–533. ISBN: 978-3-540-34381-3.
- [9] Stephan Krempel. "Design and Implementation of a Profiling Environment for Trace Based Analysis of Energy Efficiency Benchmarks in High Performance Computing". MA thesis. Ruprecht-Karls Universität Heidelberg, 2009.
- [10] Timo Minartz. *eeDaemon documentation*. 2012.
- [11] Timo Minartz, Julian M. Kunkel, and Thomas Ludwig. "Tracing and Visualization of Energy-Related Metrics". In: 26th IEEE International Parallel & Distributed Processing Symposium Workshops. Shanghai, China: IEEE Computer Society, 2012.

- Timo Minartz et al. "Managing Hardware Power Saving Modes for High Performance Computing". In: Green Computing Conference and Workshops (IGCC), 2011 International. Orlando, Florida, USA, 2011, pp. 1–8. ISBN: 978-1-4577-1222-7. DOI: http://dx.doi.org/10.1109/IGCC.2011.6008581.
- [13] Venkatesh Pallipadi and Alexey Starikovskiy. "The Ondemand Governor". In: Proceedings of the Linux Symposium, 2006, Ottawa, Canada. Ottawa, Canada, 2006, pp. 215 –230.
- Barry Rountree et al. "Adagio: Making DVS Practical for Complex HPC Applications". In: ICS '09 Proceedings of the 23rd international conference on Supercomputing. Yorktown Heights, NY, USA: ACM, 2009, pp. 460-469. ISBN: 978-1-60558-498-0. DOI: http://dx.doi.org/10.1145/1542275.1542340.
- [15] Robert Schöne and Daniel Hackenberg. "On-line analysis of hardware performance events for workload characterization and processor frequency scaling decisions". In: Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering. Karlsruhe, Germany: ACM Press, 2011, pp. 481–486. ISBN: 978-1-4503-0519-8. DOI: http://dx.doi.org/10.1145/1958746.1958819.
- [16] Robert Schöne et al. "The VampirTrace Plugin Counter Interface: Introduction and Examples". In: Euro-Par 2010 Parallel Processing Workshops. Ed. by Mario Guarracino et al. Vol. 6586. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 501–511. ISBN: 978-3-642-21877-4. URL: http://dx.doi. org/10.1007/978-3-642-21878-1_62.
- [17] VampirTrace 5.12.2 User Manual. TU Dresden, Center for Information Services and High Performance Computing (ZIH). 01062 Dresden, Germany,