Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

B A C H E L O R T H E S I S

Energy usage analysis of HPC applications

vorgelegt von
Tim Jammer

MIN-Fakultät
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6527284
E-Mail-Adresse:	3jammer@informatik.uni-hamburg.de

Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Dr. Hermann Lenhart

Betreuer:	Dr. Hermann Lenhart
	Dr. Michael Kuhn

Hamburg, den 06.12.2016

Abstract

The importance of energy consumption of large scale computer systems will grow in the future, as it is not only a huge cost factor, it also leads to additional difficulty to cool those systems down. In order to gain experience on energy consumed by model simulation, I will analyze the energy consumed by the ECOHAM North Sea ecosystem model to deduct which parts of the application use the most energy.

First the influence of the measurement of the energy usage on the application will be discussed. It is important to keep this influence in mind, as one wants to know about the energy usage of the unchanged application, so that the gathered insights are transferable to the application when it is running without the energy measurement.
Furthermore my thesis will provide an overview on the energy which is needed by the different phases of the application. A focus is placed on the serial section where the output is written. The busy waiting implemented by the MPI implementation leads to an increased energy consumption. Without this busy waiting the application needs about 4 percent less energy. Therefore, I propose that the programmer of an MPI application should be able to choose which MPI calls should perform a non busy waiting.

Contents

1. Introduction **4**

2. Research question and setup **8**
2.1. Energy usage during the runtime of a HPC application 8
2.2. Related work 9
2.3. ECOHAM model 10
2.4. Test system 12
2.5. Measuring method 12

3. Influences of the measurement on the results **16**
3.1. Influence of the tracing 16
3.2. Influence of the MPI-implementation 18
3.3. Summing up the power data from different power sensors 18

4. Result analysis **21**
4.1. Simulating different months 21
4.2. Energy needed for initialization and finish 23
4.2.1. Description of the figures showing trace data 23
4.3. Energy usage for writing the output 26
4.4. Optimizing a serial section 28
4.5. Busy waiting in the MPI implementation 31
4.6. Energy usage of communication 39

5. Conclusion **42**

Bibliography **44**

A. Measured energy usage data **49**

B. Usage of pmlib **50**

C. Table from Goekkus's thesis **52**

D. The ArduPower device installed **54**

List of Figures **55**

List of Listings and List of Tables **56**

1. Introduction

The importance of energy aware computing

This section contains the motivation why energy aware computing is such an important topic.

A lot of emphasis is placed on energy efficiency of computer systems. Especially important on mobile devices, due to the limited battery life, its importance is growing. The efficiency of batteries cannot grow as fast as the demand on longer living and more powerful mobile devices like smartphones. Therefore adding new features for mobile devices needs to be coupled with a more energy efficient way of implementing those features as it is not feasible when those new features consume too much additional energy. It is also important to view the heat development in mobile devices as a CPU chip in a smartphone already has a higher energy density as a hot plate. (see Figure 1.2)

But it is not only an important topic for mobile devices.
With more and more data being processed, more and more energy is needed for that. As an example: Google used about 2.26 billions kilowatt hours of electricity in 2011. [Zei12]. In 2014 this value increased up to about 4.4 billions kilowatt hours [Rix16].
This high demand is a huge cost factor. In the EU this large amount of electricity would have cost already about 209,728,000 Euro for 2011. In 2014 the cost for the energy would have rised to about 403,223,000 Euro[1].
With this high cost, a saving of only about 4 percent of energy will result in over 16 million Euros less cost of energy. Therefore also seemingly small savings are really worth the effort.
Despite these high costs one should also consider the carbon footprint for this high amount of energy and its ecological influence.
This also affects the research facilities that uses high performance computing as a tool to calculate computational models in order to reveal the connections between processes in the real world.

Figure 1.1 shows the development of characteristics of Intel CPUs. One can see that the amount of transistor increases exponentially. But the exponential increase of the clock speed has stopped around 2003 alongside with the increase of power consumption which has also stalled around this point. This shows the ongoing trend towards more

[1] Source : Calculated with the average EU price from 2011 respectively 2014 for industrial consumers [Eur16]

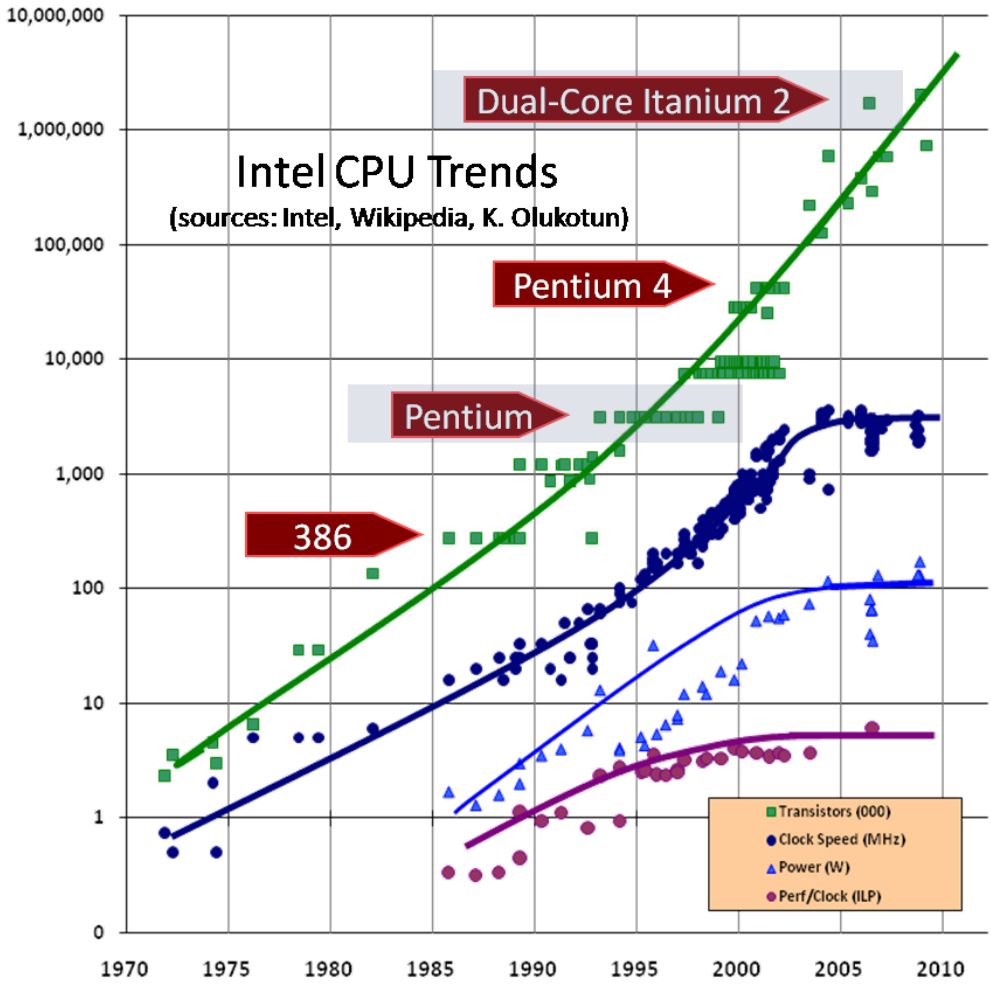


Figure 1.1.: Intel CPU Trends. Source [Sut09]

parallelization to further increase performance.
This leads to a mostly non linear tradeoff between power and energy used. Scaling performance using parallelization does often not work linearly, as not all parts of a program can work simultaneous and using more cores often lead to more communication overhead. When using more cores, the amount of energy needed will linearly increase, but the runtime will very often not improve linearly (Amdahl's law).

Figure 1.2 shows that the power density of an Intel CPU passed the density of a hot plate and is close to reach the density of a nuclear reactor [Mar13]. While a power density of a hot plate is still manageable even on mobile devices, the density of a nuclear reactor requires huge cooling efforts and is certainly not feasible for a mobile device.
With increasing power density it is also much harder to cool the components down. In large clusters with many installed computation devices this is a really big matter, as

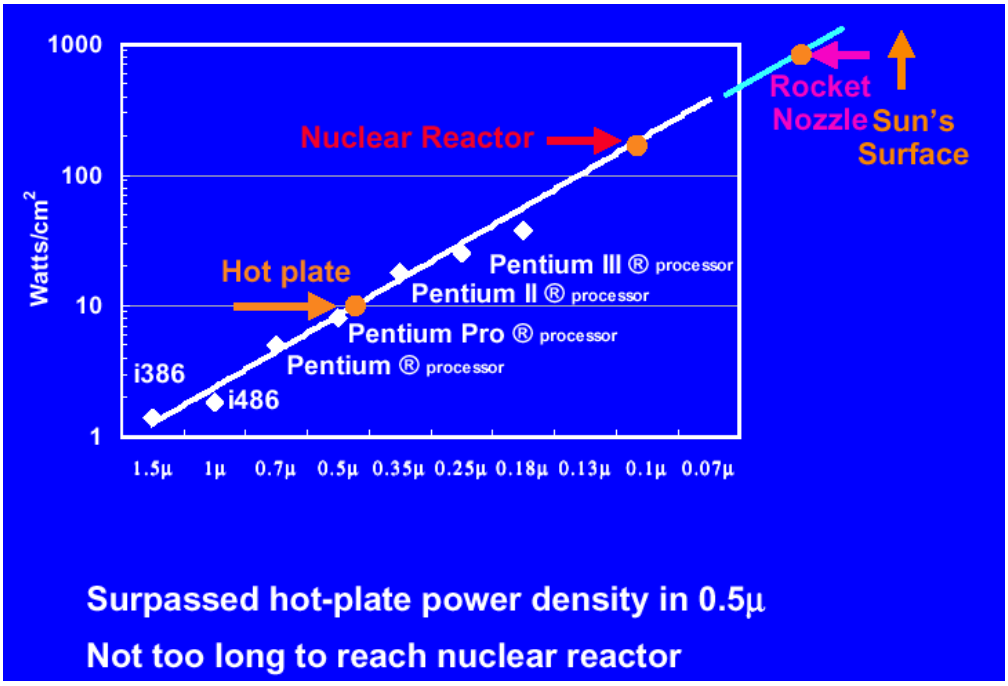


Figure 1.2.: Power density of CPU. X-axis denote the transistor size.
Source [Mar13]; These sizes were common between 1985 and 2006, so one can treat it like a timescale. For exact connection between time and the transistor size one can look at [Kle14].

all the heat from all the devices has to be transported away. This cooling will not only lead to a difficulty in designing and constructing large server farms, it will also consume additional energy.

Therefore we must ensure that our resources are used as efficiently as possible, which means that we cannot only focus on an efficient use of hardware to calculate as fast and as accurate as possible. We must also aim at using as little energy as possible, which will decrease our carbon footprint as well as reduce the cost of running such a great center of computation. This allows us to spend this savings otherwise, like further improvements to the provided service.
The DKRZ (The German Climate Computing Center) spends about 2.5 million Euro of its annual budget on energy [Lud15a]. Therefore only a small decrease in the needed energy like 4 percent will save 100 000 Euros. This will be enough to justify the assignment of more than one researcher just to accomplish this relatively small saving of four percent.

In the second chapter I will present the goal that I am aiming at with this work. it will also contain a discussion about previous work related to improving the energy efficiency of information systems in chapter two as well. The second chapter will conclude with a description of the setup of my analysis, consisting of a description of the test system and the analyzed application as well as measurement techniques, on which my analysis is based on.

In the third chapter I will discuss how the measuring will influence the analysis.
The fourth chapter consists of the discussion of the gathered results where I will show the impact various factors have on the energy consumed. I will take a special look on the sections where the application writes the output, as these sections are serial.

The final chapter will wrap up the thesis with my concluding thoughts.

2. Research question and setup

In this chapter I will present the goal that I am aiming at with this work. It will contain a discussion about previous work related to improving the energy efficiency of computer systems in Section 2.2. I will conclude this chapter with a description of the setup of my analysis, consisting of a description of the analyzed application (Section 2.3) and the test system (Section 2.4) as well as measurement techniques on which my analysis is based (Section 2.5).

2.1. Energy usage during the runtime of a HPC application

In this section I will describe the goal of my thesis.

Research to optimize an application mainly focus on optimizing its runtime. As a general rule of thumb - apart from choosing more efficient algorithms and data structures - it is advisable to optimize the part of the program which is executed most of the time, because a small saving in one execution will result in a big saving for the overall runtime [Lud15b]. Within my research I want to address the question, if this rule of thumb also applies to optimization for less energy consumption.

Therefore in this thesis I want to analyze how much energy is used by the different parts of a program. The goal is to deduct which parts of a program offers most potential for energy savings and therefore should be optimized first.
I will analyze the energy usage of the ECOHAM model to figure out which different parts of the model use which amount of energy.
I want to investigate the energy usage also to see if the energy usage of the different parts of the application correlate with their runtime. 5of this different parts.

Therfore my thesis will give an overview on which parts of the program the most energy will be used and where the critical sections are, where the energy consumption needs to be optimized first.
I will also consider how the energy consumption of these sections can be lowered. A special focus is placed on the serial output section of the application.

2.2. Related work

This section contains a small discussion of related work.

Research to reduce energy consumption of information systems mostly addresses reducing the energy that the hardware needed [Ran10]. Because the hardware directly uses energy, an efficient system relies on an efficient hardware.

A system does not only depend on its hardware layer, also the software has to be efficient in order to get an optimal overall result. Therefore I want to focus on improving the software to use the hardware as efficient as possible. Work related to increasing energy efficiency of the software layer of a system is mostly very specific and strongly connected to the used hardware. For example [UBPEG02] investigate the energy efficient use of a wireless network.

I consider approaches to dynamically scale the power of a component ([WAT09]), like the CPU or disk, as a connection between improving the energy saving of the hardware as well as improving the efficiency of the software. It is a hardware related approach, as techniques like this rely on the capability of the used hardware to suspend and consume lower or even no energy and efficiently get back to the full power working status. But it has to be decided from the software side when to scale down, which is also a very interesting topic.

Work that is related to improving the energy consumption of a general application does not state it explicitly, but you can find a common result if you read between the lines: "if it is faster, it uses less energy". For example you can take table 2 from the summary of Goekkus's thesis [Goe13] and substitute "energy" with "runtime"[1]. His statements will still stand to reason and are similar to results of work that tries to reduce the used runtime and not the used energy [Lud15b]. Intel's energy efficient software guidelines [Lar11] also point in this direction. The main point behind this is, that it is in most cases worth to execute the application as fast as possible, in order to power down the system components as erly as possible to achieve a minimal power consumption. As energy is the product of power and time, decreasing the needed time to finish a task result in less energy needed, even if that means to use a bit more power during that shorter time. But one have to take in account if this really holds on many systems, as the power used after an application has finished does not drop down to zero. Instead there is an idle power consumption in order to guarantee that the system stays ready for the next task.

Additionally energy savings due to parallelization of an application were discussed in [CM10]. When processors can be individually turned off, the optimal energy usage is achieved with the fastest way of running the application. That is when it uses the maximum possible amount of processors in parallel sections of the program. Otherwise

[1] see Table C.2 in the appendix

when it is not possible to switch off the processors individually, there is a tradeoff between runtime and energy consumption, because taking more processors for faster calculation will result in more energy used. [CM10] investigates how to find the optimal tradeoff for a given requirement.

Also proposals for energy aware scheduling to favour energy efficient applications and reduce overall energy consumption (like [BSH+08] or [MRZ+03]) address the question how one can improve the energy efficiency of systems by adjusting its software, rather than relying on more efficient hardware to improve energy saving. This is a complex research field. In order to find efficient algorithms, the energy usage of the application needs to be estimated, so the scheduling algorithms can use the energy needed as an input variable [SBM09]. A pleasing technique to estimate the power consumption of an application was not found yet. Techniques that rely on hardware performance counters [CXDM10] are relatively easy to compute and do not need additional hardware to perform but they are not free of pitfalls. For example [MHEZ13] shows that "memory events like cache misses [do not] correlate with power consumption [...] on [their] multicore system". Other methods are only feasible for small embedded systems as they are too complex to use for large scale systems [Naj94].

2.3. ECOHAM model

For this study the biogeochemical ecosystem model ECOHAM (ECOsystem model HAMburg, version 5) is used. The model addresses questions on changes in carbon flow within the North Sea in relation to climate change [LPMK12] as well as questions related to different pressures upon the North Sea ecosystem related to nutrient input of nitrogen and phosphorus [LMBB+10].

Within the model the North Sea is subdivided into a three-dimensional grid and for each grid cell a number of numerical differential equations are solved. The different state variables represent the so-called functional groups within the ecosystem. These are selected in a way, that their interaction represent the cycle of a specific matter group, like nitrogen. In total ECOHAM represents the cycles of carbon, nitrogen, phosphorus, silicon and oxygen by calculating the changes in the state variables and the related processes per timestep.

The model domain extends from l5.25° W to 14.083° E and from 47.583 to 63.983° N and comprises the entire North Sea, large parts of the Northwestern European Continental Shelf and parts of the adjacent Northeastern Atlantic. The horizontal resolution is 1/5° with 82 grid points in latitudinal direction and 1/3° with 88 grid points in longitudinal direction, which corresponds to a resolution of 20 x 20 km. The horizontal grid of the model domain is shown in Figure 2.1. The vertical dimension with a maximum depth of 4000m is resolved by 31 z-layers with a surface layer of 10m. The vertical has a resolution

of 5m between 10 and 50m depth. Below 50 m, the layer thicknesses successively increase with depth.

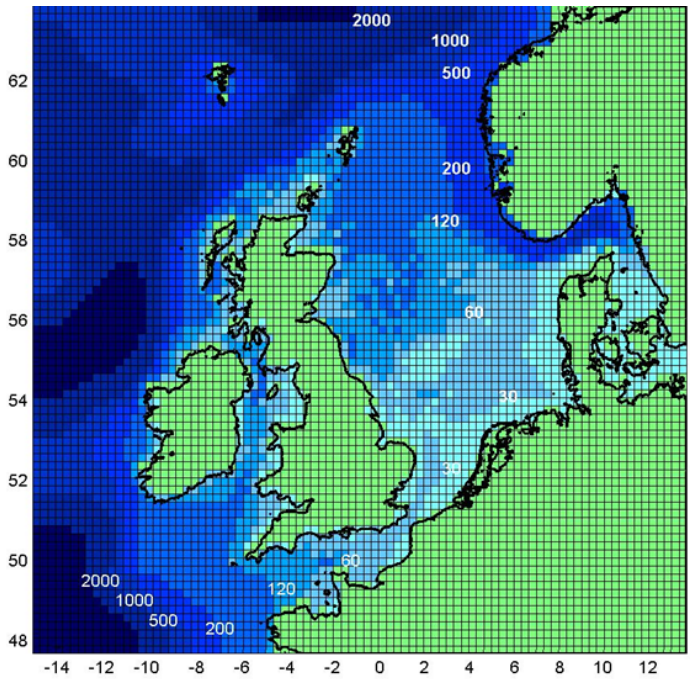


Figure 2.1.: Illustration of the grid used by the ECOHAM model.

ECOHAM can deliver simulation for the time interval between 1977 up to 2014 based on realistic forcing. The model runs in a so-called "off-line mode" with a time step of 30 min using the 24-hour averages of the hydrographic and hydrodynamic fields generated by the general circulation model HAMSOM. The meteorological forcing was derived from NCEP/NCAR reanalysis data which provides 6 hourly information for air temperature, cloud coverage, relative humidity, wind speed and direction which were interpolated onto the model grid. Daily freshwater run-off data for 249 rivers were provided by Cefas (Centre for Environment, Fisheries and Aquaculture Science) and represent an updated dataset of that used by [LMBB$^+$10] covering the entire simulation period. The same dataset encompassed nutrient loads used for the ECOHAM model. The data for the atmospheric nitrogen deposition represent mean deposition rates for the North Sea. These data are particulary important for the analysis of the so called oxygen deficiency [GLPB15].

The products of the biogeochcmical simulation from ECOHAM are stored as daily values. This output data consist of cumulative process information as well as snapshots of state variables which are stored for the entire model grid and the complete simulation time interval, but separated by years.

2.4. Test system

For obtaining the data presented in the next chapter, the cluster from the research group of Scientific Computing (WR cluster) were used.
The WR cluster is a system used for research and teaching. There are different kinds of nodes. I used the nodes which are based on 2 Intel(R) Xeon(R) CPU X5560 @ 2.80GHz with 4 cores each and are equipped with 12 GiB RAM. They are operated by Ubuntu 14.04.3 with Linux Kernel version 3.13. The nodes are connected with Gigabit-Ethernet (1GbE) and the cluster uses the distributed Lustre filesystem and SLURM for batch queueing. They use the OpenMPI implementation (version 1.6.5).

I executed the ECOHAM application on one of these nodes with 8 processes. Therefore the communication between the processes do not need the network. So the network will only be used for writing the output to the lustre server. Except for Section 4.6, where i show the results of executing the application on 4 nodes with 2 processes on each node, in order to investigate the cost of transmitting the messages over the network.

2.5. Measuring method

In this part I will describe the measurement techniques that I will use within my Thesis.

In order to get detailed data at what time the program uses which amount of energy, I use the pmlib plugin for VampirTrace.
VampirTrace is a software which automatically instruments an application in order to obtain performance tracing data.[2] The pmlib plugin extends the functionality of VampirTrace so that in addition to the normal performance trace also the energy usage data is collected. The collected trace data can later be visualized with the Vampir software. The plugin also allows Vampir to show the power data alongside the performance trace to inspect their connection.

The architecture of pmlib is illustrated in Figure 2.2.
As illustrated in Figure 2.2 an application runs on a target platform which is connected to an energy measurement device. While it is running, the application can also be traced to analyze its performance. The data for power consumption is collected separately. I will use VampirTrace in order to get the performance data, but the architecture of pmlib will allow any other way of tracing the performance. It is not even mandatory to trace the performance, but I would strongly recommend it due to the fact that the raw data of the energy consumption alone will not be as informative as when one can link it to the performance trace.

Pmlib consists of a server which collects the measured energy usage data from the measurement device (see Figure 2.2). It can collect the data from many different devices.

[2] `https://www.vampir.eu/`

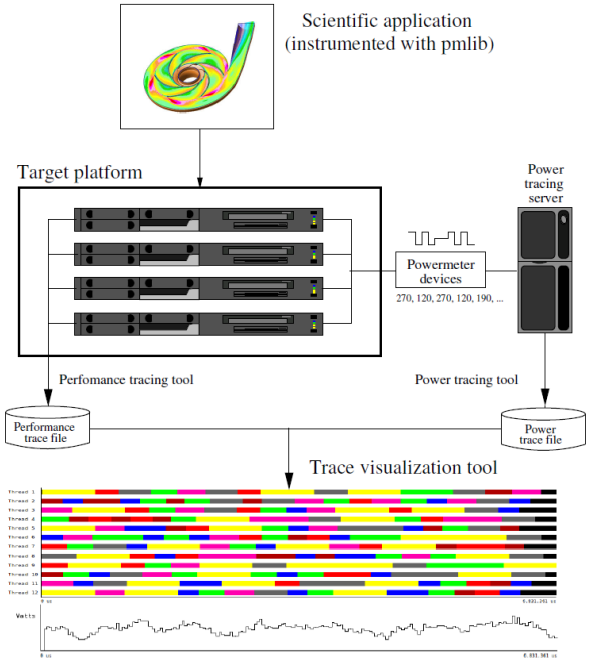


Figure 2.2.: Illustration on how pmlib works. Source [BBC+13]

This server is a Python program, which is designed to easily add new devices as the routine how the data is collected from a particular device should be written as an extra module. Thus a new added device does not have to be a measurement device which measures energy consumption it can measure any other value (such as temperature) as well.

For my analysis I have adapted the server program. As the server has to be close to the measurement device, for convenience reasons, it is running at the same node as the analyzed application. I will discuss the implication this has on the measured data in Section 3.1. As the server is running on the same node, it does not need to send the measured data over the network anymore. Instead of sending it over the network, it now writes the data to a file, which will be merged with the rest of the trace at the end. Writing the measured data into a file is faster, and therefore more efficient, than transmitting them over the network. As in my case the server programm is executed at the same node it is feasible to just dump the measured data to a local file. In Figure 2.2 it is illustrated, that there is a dedicated server hardware. Then the server has to send the measured data over the network at the end, so that it can be stored alongside the performance trace.

In order to tell the server when it should collect the data from which devices, the running application has to be instrumented with instructions that communicate with the server and ask it to start or stop the tracing and receive the trace-data from the server in the end to write them into an output file. The pmlib C library offers the required functions to do so.

I will use the pmlib plugin for VampirTrace which instruments the application automatically with the pmlib functions in addition to the VampirTrace instrumentation for tracing the performance.

Using the pmlib plugin is quite simple: The application has to be compiled with the VampirTrace instrumentation. Listing 2.1 shows the only additional steps required: First one has to set the library path so that the pmlib library can be found. Second one has to set the VT_PLUGIN_CNTR_METRICS environment variable. This variable will contain all metrics delivered by plugins, that one want to trace. It will list the metrics separated by a colon (":"). Each metric consists of the plugin name ("PmlibPlugin_10.0.0.20") and then separated by an underscore ("_") the name of the metric ("ArduPowerDevice_0"). Every input lane of the arduPower device is defined as its own metric, therefore the loop in Listing 2.1 sets the environment variable so that it contains all 16 different metrics so that the power on all 16 different input lanes is measured. The Plugin will parse the plugin name, in order to get the (underscore seperated) IP-address of the energy measurement server ("10.0.0.20").

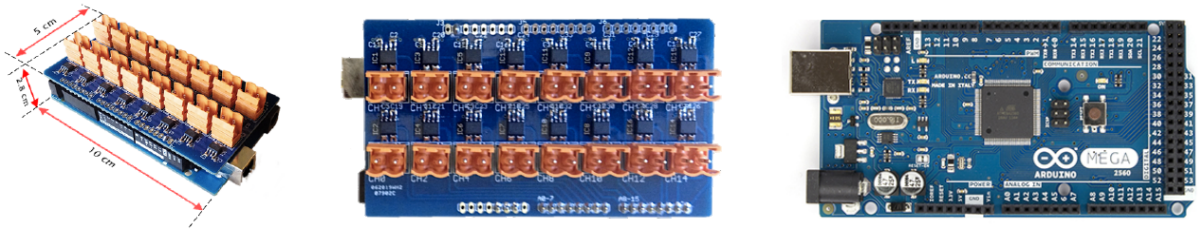
```
setenv LD_LIBRARY_PATH "${HOME}/pmlib/pmlib_client:${LD_LIBRARY_PATH}"

setenv VT_PLUGIN_CNTR_METRICS "PmlibPlugin_10.0.0.20_ArduPowerDevice_0"

foreach i (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

    setenv VT_PLUGIN_CNTR_METRICS "${VT_PLUGIN_CNTR_METRICS}:PmlibPlugin_10.0.0.20_ArduPowerDevice_${i}"
end

 mpiexec ${executable}
```

Listing 2.1: Usage of the pmlib Plugin for VampirTrace.

After collecting the trace-data, Vampir allows to show the energy trace-data alongside the performance trace-data in order to investigate the connection between the performance and the energy consumption.
More detailed description on how pmlib works can be found in [BBC+13].

An Arduino Shield served as a powermeter device (shown in Figure 2.3), which is capable of measuring the energy on 16 different cables with its hall effect sensors [DHK+15].

Although this device is small and cheap, it is very accurate (±1.5%). Earlier devices for measuring the power demand of a system are much more costly and often offer less samples per second or are not capable of distinguish the energy needed by the different hardware components [LPD13].



(a) ARDUPOWER wattmeter. (b) Power sensing shield of ARDUPOWER. (c) Arduino Mega 2560 board.

Figure 2.3.: Components of the ARDUPOWER wattmeter. Source [DHK+15]

[BH15] suggests to use the ArduPower device to be able to measure the energy consumption of the different hardware components of a computation node. But in my analysis I will simply sum up all this partial consumptions to analyze the node wide energy usage. I assume that the different hardware components themselves are very much optimized, because hardware producer already spent lots of effort to this. So I think for my analysis it is more informative to see the overall consumption as an output of the interaction of all this components, rather than just analyzing the energy consumption separated for each part of the hardware. But the possibility to see which energy each individual component needs will be very informative in order to explain why the energy needed by the overall system rises or falls.

3. Influences of the measurement on the results

In this chapter I want to show the impact that the various factors have on the energy consumption.
First I will discuss the influence the measurement has on the behaviour of the application, as one has to keep in mind that all results are subject to this influence.
In Section 3.2, I will continue this discussion while taking a look at the influence of the MPI implementation.
Section 3.3 will discuss the error encountered, when one sums up the measurement data from 16 different sensors.

3.1. Influence of the tracing

In this section I will discuss how the performance tracing with VampirTrace and the energy measurement with pmlib will influence the applications behaviour.

When analyzing the energy usage of the application, we have to keep in mind that the tracing may alter the behaviour of the application as well as it needs additional energy.
In Section 4.3 we will discover some influence of the tracing. When the trace buffer is full, VampirTrace flushes it to the trace file. This I/O event needs more energy then the computation, but the trace buffer has to be filled with the tracing events in the first place and this will influence the behaviour of the application as well. Therefore I want to have a closer look on the influence of the tracing, especially on the additional energy which is needed to trace the application.

In order to measure the energy the application needs without the VampirTrace instrumentation, I manually requested the pmlib energy tracing server to start collecting data and receive the collected data from the server at the end of the measurement. No other communication than starting the energy measurement, stopping the measurement and receiving the measured data at the end, is needed. I have written a wrapper program which does this communication with the server[1]. First it asks the server to start the measurement. Then it starts the application without any instrumentalization. When the application has finished, the wrapper program communicates with the server again to

[1] see Listing B.1 in the appendix

finish the measurement and receive the energy usage data. So the application does not need to be changed.

The server program that collects the energy usage data is running on the same computation node on which the application is running. This is the only remaining influence the measurement has on the application. The server has to be close to the measurement device for convenience reasons.
But the server program is a background process which only collects the measured data from the Arduino device from time to time. At the test systems are running many more processes belonging to the operating system which interfere with the application the same way and cannot be switched off in order to guarantee a stable environment.
There is no significant difference in runtime for the manual instrumented code comparing it to the unchanged one running without the energy measurement.
Therefore I will assume that manual collecting the power data will not influence the application, as many other factors like the network usage by other users have a stronger impact on the application performance. The variance between the runtime of two runs ($\approx 2\%$)[2] is greater than the difference between the application running with the server collecting the energy usage data and the application running without this additional background process. So the effect of this energy measurement is not measurable.

The device is capable of sending over 475 samples per second, but I only used 100 samples per second in order to lower the load for the server. 100 samples per second are still enough to cover the different tasks which the application performs. For example, when looking at Figure 4.15, which is the most detailed view on the trace, between two dotted lines are still 50 samples.

As a downside for this measurement technique without additional tracing, one cannot connect the measured energy usage to any other data than the output of the application, in order to find the parts of the program where the most energy is used.
In order to connect the energy to the different phases of the application all other sections are subject to the influence of VampirTrace.

A one month run of the model uses 228 kJ of energy on average. With VampirTrace tracing its performance, the application uses 320 kJ of energy on average. So the tracing with VampirTrace needs an additional 92 kJ, which is about 40 percent more.
In terms of runtime the application needs 2037 seconds to finish an average run. With VampirTrace instrumentalization it needs 2769 seconds, which is only 35 percent more. When tracing the application VampirTrace needs to flush the full trace buffers from time to time. This I/O event causes a rising in the energy demand of the system. Therefore the tracing increases the needed energy even more then it increases the runtime. The reason why the tracing uses more energy in relation to its increased runtime, will be discussed in more detail in Section 4.3.

[2] see Appendix A for the measured values

3.2. Influence of the MPI-implementation

To show which influence on the efficiency of an application the used MPI implementation has, I also used the MPICH implementation (version 3.2). When using MPICH the application only needs 1951 instead of 2037 seconds (4.2 percent less) for the calculation of one month. With the MPICH implementation the system needs 113 Watt on average. With the OpenMPI implementation it only needs 112 Watt on average. But considering the lower runtime, MPICH is more efficient as the run only needs 219 kJ in total against 228 kJ with the OpenMPI implementation (4.045 percent less energy needed)[3]. For the analyzed application the MPICH implementation is about 4 percent more efficient.
In the next sections the OpenMPI implementation (version 1.6.5) will be used again, because the VampirTrace build on the cluster only works with OpenMPI.

3.3. Summing up the power data from different power sensors

This section takes a closer look on how the data from the 16 different sensors were summed up.

As the 16 different power sensors will gather the power usage data independently, one cannot simply sum up the data without taking care of the timing.

16 different sensors need to be synchronized. They should start collecting data at the same time. But as it is nearly impossible to send 16 different messages to the sensors at the same time, there will be a small delay, as illustrated in Figure 3.1. Also the time between two samples may differ slightly. For example a sampling rate of 1 Hz means 1 sample per second, but it is not stated if it should be taken from the first or second half of the second. When exactly the sample will be taken might vary, depending on when exactly the sensor is asked for the sample and how long the data transmission take.

[3]This results where gathered with manual instrumentation (see Appendix A)

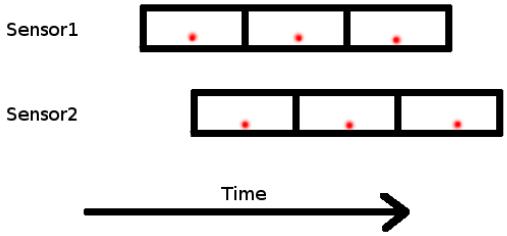


Figure 3.1.: Illustration of the timing error. The black boxes illustrate one sample. The red dots show when exactly the measured value was read.

Figure 3.1 illustrate the encountered error. The black boxes illustrate one sample and the red dots show when exactly the measured value was readed. When one wants to sum up the data from these two sensors, it has to be decided which points will be summed up. For example: which sample from sensor one belongs to the first sample from sensor two2?
To get the corresponding value for the point of time where the sensor value of sensor two was read, I will take the values from the next two timepoints to interpolate the value in between.

So if one just sums up the values grouped by order of time points, it can lead to an error where you sum up values that belong to different points in time.

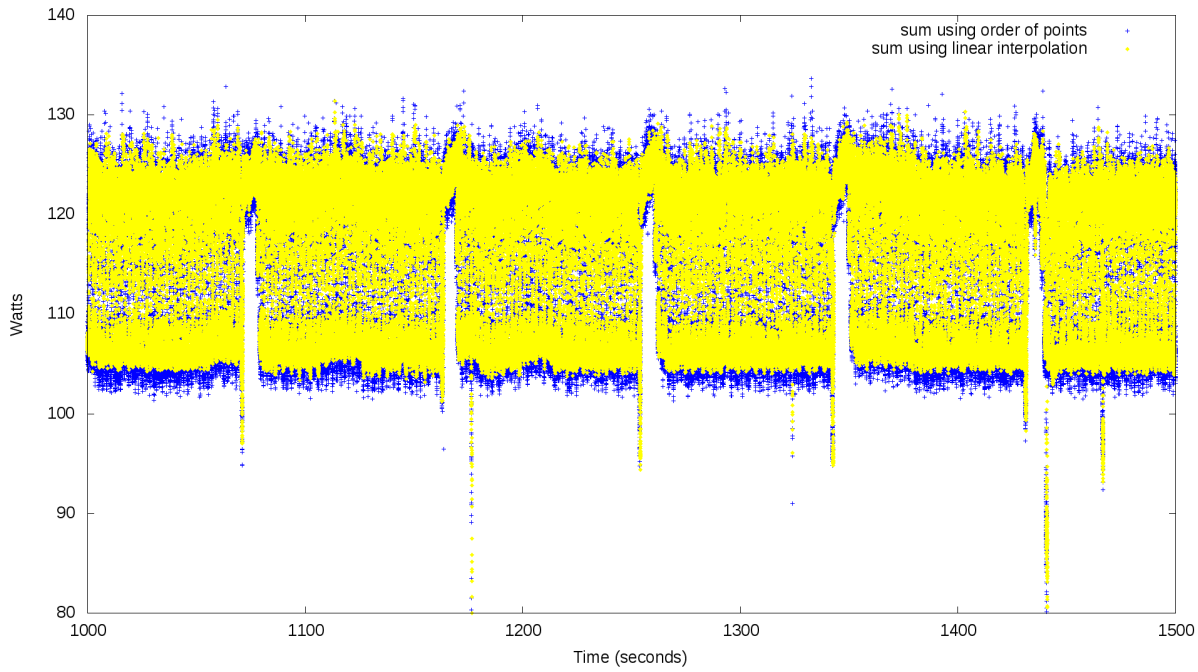


Figure 3.2.: Energy usage data summed up by order of points (blue) and using linear interpolation (yellow).

Figure 3.2 illustrates that error. It shows the difference when one sums up the data grouped by the order of measurement points contrary to the summation using linear interpolation. To get the data for a timepoint where no samplevalue is present, I used the two surrounding measurement points to interpolate the value of the timepoint I want to sum up. One can see that the variation decreases a bit, but the overall pattern remains the same.

The small deviations of the points of time where the sample values are taken from, are not a problem anymore, when one considers the amount of Joules used. One can simply calculate the Joules used on each lane separately and sum it up without taking care of timing, as time is already included in the unit "Joule".

4. Result analysis

In this Chapter I will describe and discuss the gathered results in relation to different aspects of the application.

In Section 4.1 I will take a look on how the application differs when simulating different months. Section 4.2 will continue my analysis of the applications energy consumption, especially laying the focus on the initialization and finialization parts of the application. In Section 4.3 the behaviour in the iterations of the main part are discussed. A more detailed view on the energy needed for writing the output is given. Section 4.4 shows an easy way to lower the amount of energy needed during serial sections of a program. In Section 4.5 I will take a closer look on the busy waiting which is implemented in the MPI implementation. Finally Section 4.6 gives a more detailed view of the energy needed for the communication between the processes.

4.1. Simulating different months

As the simulated ecosystem behaves different in the different months of a year, the months will also differ in the simulation. Therefore I will take a look in which way the energy needed for the simulation of the various months differs.

Figure 4.1 shows how much Jules are needed to calculate the different months of a year.

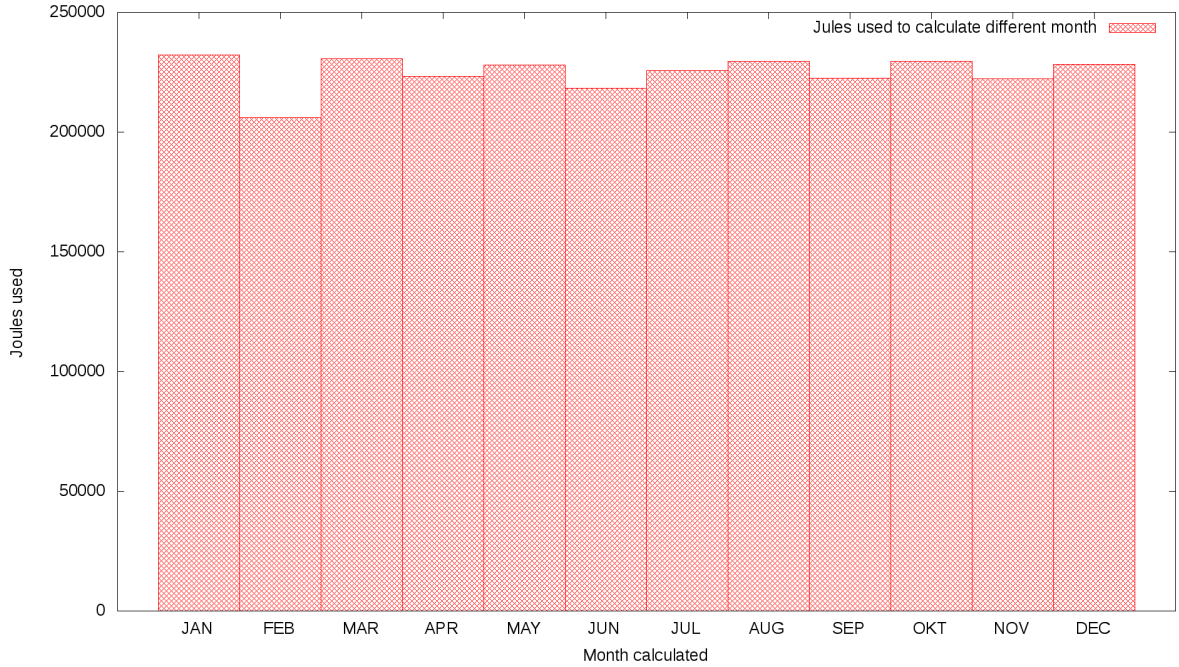


Figure 4.1.: Energy used to simulate different months.

One can see which month has 31 days, as more energy is needed to calculate the additional day. February stands out, as it has only 28 days and needs even less energy. There is no other significant distinction between the energy used to calculate different months.
The hypothesis that calculating the summer months needs more energy, because the simulated ecosystem is more active in the summer months, cannot be backed up by the data. The energy used by a calculation is not influenced by its result. Therefore simulating a more "active" ecosystem like in the months of spring use roughly the same amount of energy as simulating a less "active" one (with lower result values) like in the winter months, as the steps to obtain these result values are exactly the same only with different input data.

Therefore there is no reason to alter the input data, for example to scale it down to compute lower values.

4.2. Energy needed for initialization and finish

In this section the energy usage during the init and finish phase of the application are discussed.

Most applications have a dedicated initialization and finialization part, where the needed resources are allocated or the input data is read. As they wrap around the rest of the program, we will start to analyze them to see how they differ from the rest of the program.

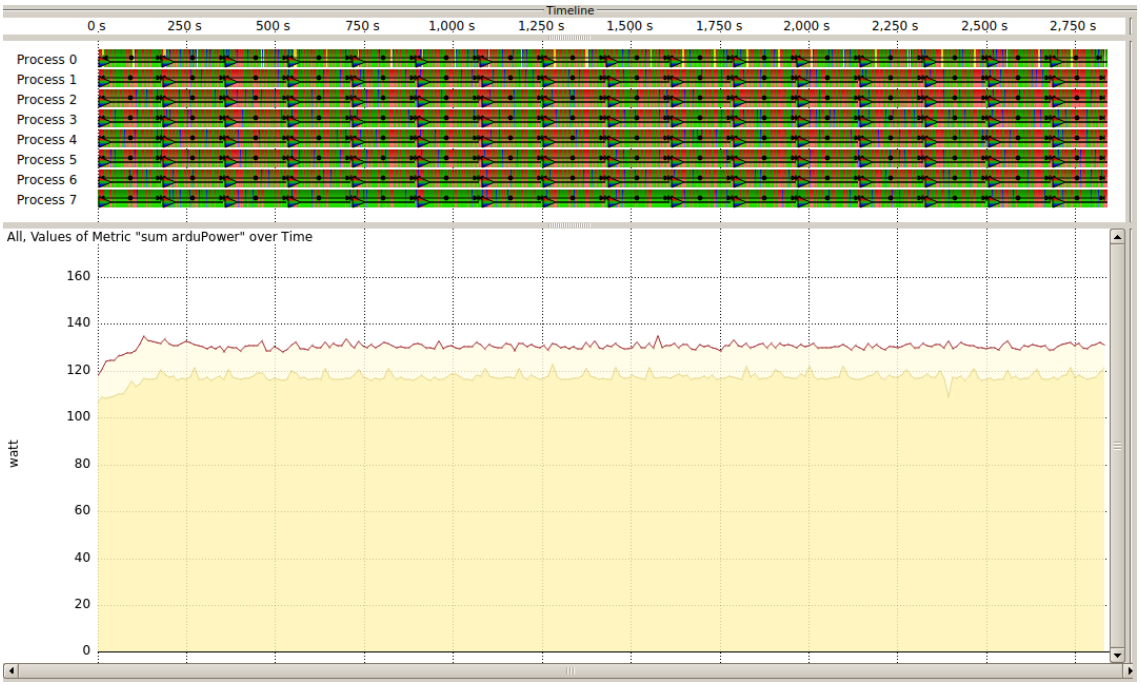


Figure 4.2.: Overview of the whole trace. Detailed description of the chart in Section 4.2.1.

4.2.1. Description of the figures showing trace data

In the upper half of the figure, below the timescale, the trace of the application is shown. Red sections represent MPI library calls where the processes uses the MPI interface to communicate. Yellow sections represent I/O events when input data is read or the output is written. Blue sections represent the VampirTrace instrumentalization where the trace buffer is flushed. The green sections represent the other application code where the computing is done. The lower half of the figure shows the chart of the power the system needs. As more than one measurement point needs to be displayed in one pixel, the red line is the maximum measured for this time. The line where the strong yellow filled section starts, below the light yellow filled one, denotes

the average of all measured points for this time.

Figure 4.2 shows the overview of the whole trace. The only trend one can see at this high level view is, that the power demand of the system is lower at the beginning and therefore rises a bit in the very first part of the program. This rise can be explained by the fact that more cooling is needed because all devices are active now. The cooling system has to power up to secure an appropriate temperature.

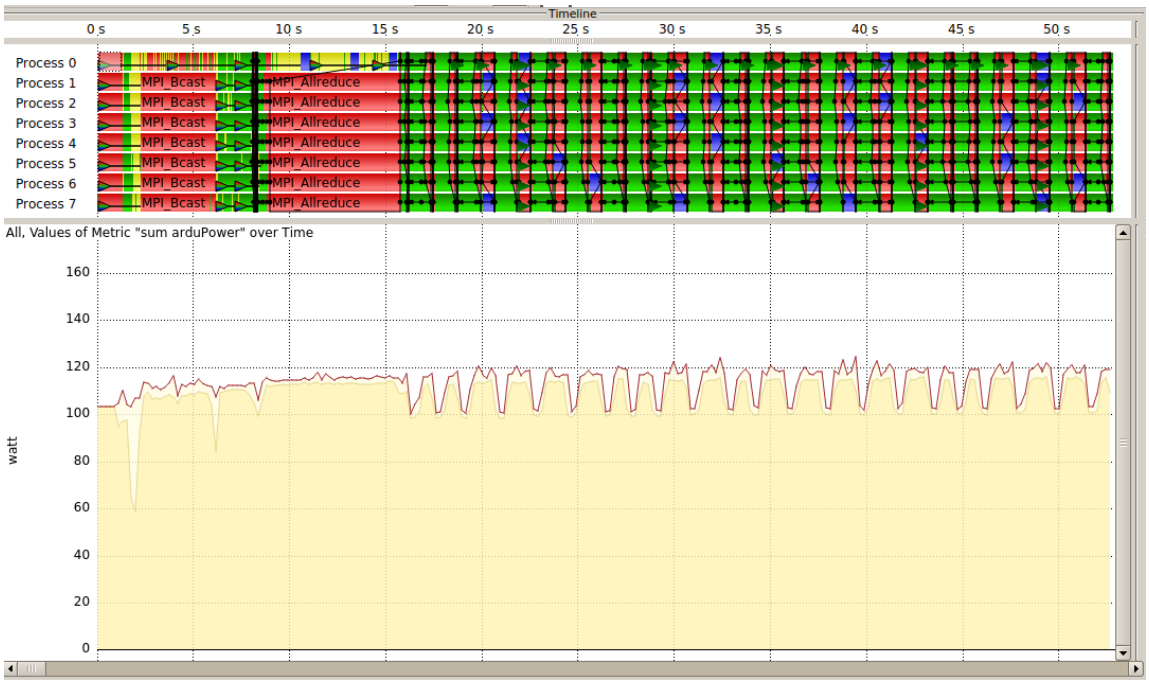


Figure 4.3.: Vampir showing the initialize section of the application. Detailed description of the chart in Section 4.2.1.

Figure 4.3 takes a closer look to the init section at the beginning of the program. After the `MPI_INIT` call the power demand of the system rises slightly while the input data is read. As the reading of the input continue while the data structures are allocated and filled with the initial data, the power needed by the system does not drop. At about 9 seconds the first output section starts writing the initial state to the output file. Then the program continues with a pattern which will be discussed in Section 4.3.

The section where the input data is read is very short, compared to the section where the output is written, and it only occurs once. Therefore, as the saving potential is greater, I only take a look on analyzing these other I/O sections more in detail.

Figure 4.4 shows the last section of the trace. After the output from the last iteration is written, the resources are freed and the master process conclude the output file, while all other processes are waiting to terminate. This last wrap-up can also be

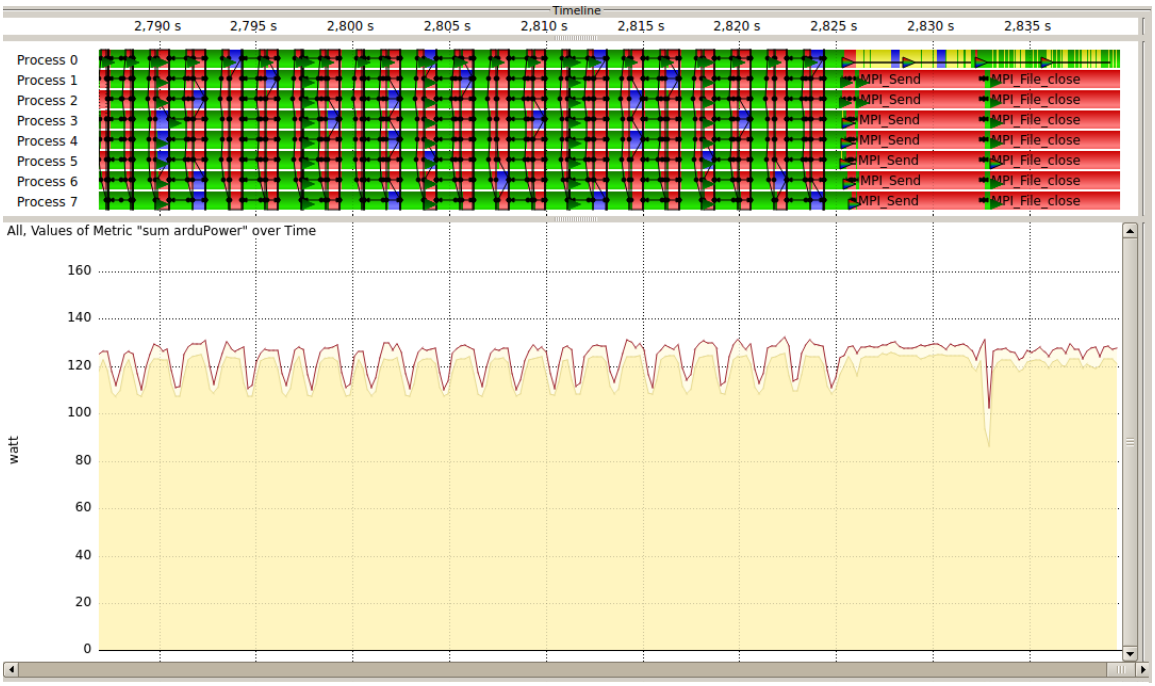


Figure 4.4.: Vampir showing the finialization section of the application. Detailed description of the chart in Section 4.2.1.

subject to an optimization like the one presented in Section 4.4.
But as initialization and finalization only occur once, they do not have a huge impact on the overall performance.

One run of the model calculating 3 months uses 676 kJ of energy in 6263 seconds of runtime. Three independent runs for the same 3 months with one month each need 669 kJ in 6225 seconds of runtime together. The run calculating all 3 months at once uses even slightly more energy (about 1 percent more) than the three independent ones. This small deviation can be explained with measurement inaccuracy as the power consumption of the operating system is also recorded. But another explanation can be found in Figure 4.2 where one can see that the power used in the first few seconds is slightly lower than the power the system needs during the rest of the runtime. As this pattern emerges in all traces, it is an indicator that the system uses slightly less power during the first few seconds where an application is executed. Therefore the impact of the initialization phase of the application is lowered even further. In the measured data it is even so low that starting multiple runs with one month each uses slightly less energy.

When discussing the impact on the runtime it has to be mentioned that the time the scheduler needs to finish the job and load the next one is not included as only the runtimes were summed up. There has to be a small break which allows the system to cool down, so that the next application can start on a cooler system which do not need the full cooling power right at the start.

4.3. Energy usage for writing the output

This section will discuss the energy needed to perform the output.

We now will take a closer look on the pattern mentioned in Section 4.2. To see the behaviour after the init phase, Figure 4.5 shows an iteration in the middle of the program. The yellow section, where the output is written, clearly stand out in the power consumption graph. The spikes in the section where the computing is done, refers to the I/O events where VampirTrace flushes the trace buffer, which will be discussed in more detail in the paragraphs below. In all other iterations of the program the same pattern emerges for the power demand of the system.

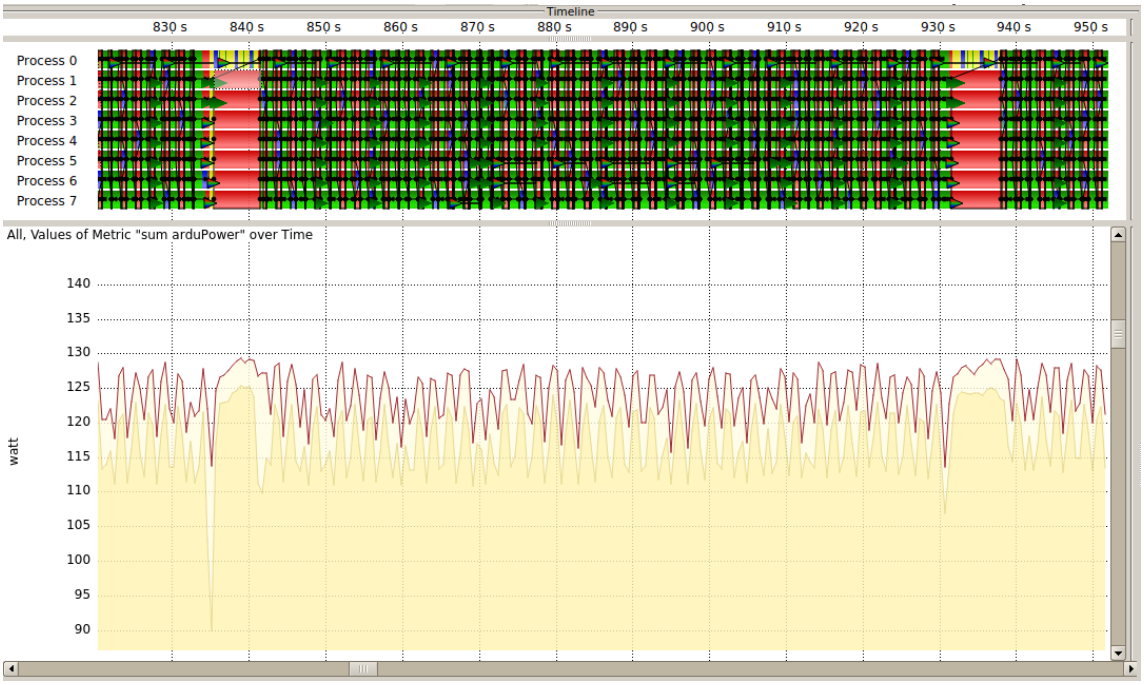


Figure 4.5.: Vampir showing a full iteration surrounded by the section where the output is written. Detailed description of the chart in Section 4.2.1.

To have a more detailed view, Figure 4.6 shows Vampir visualizing the trace data for the output phase after the end of the computation of one day, alongside the start of the computation of the next day. The yellow section is the end of the section when the master process does the output.

One can see that the power consumption of the system while it processes I/O events is 10 to 15 Watt higher compared to times where no I/O event is present. While doing I/O the system consumes approximately 125 Watt.
The first section where 125 Watt are used, is the section where first the output from the

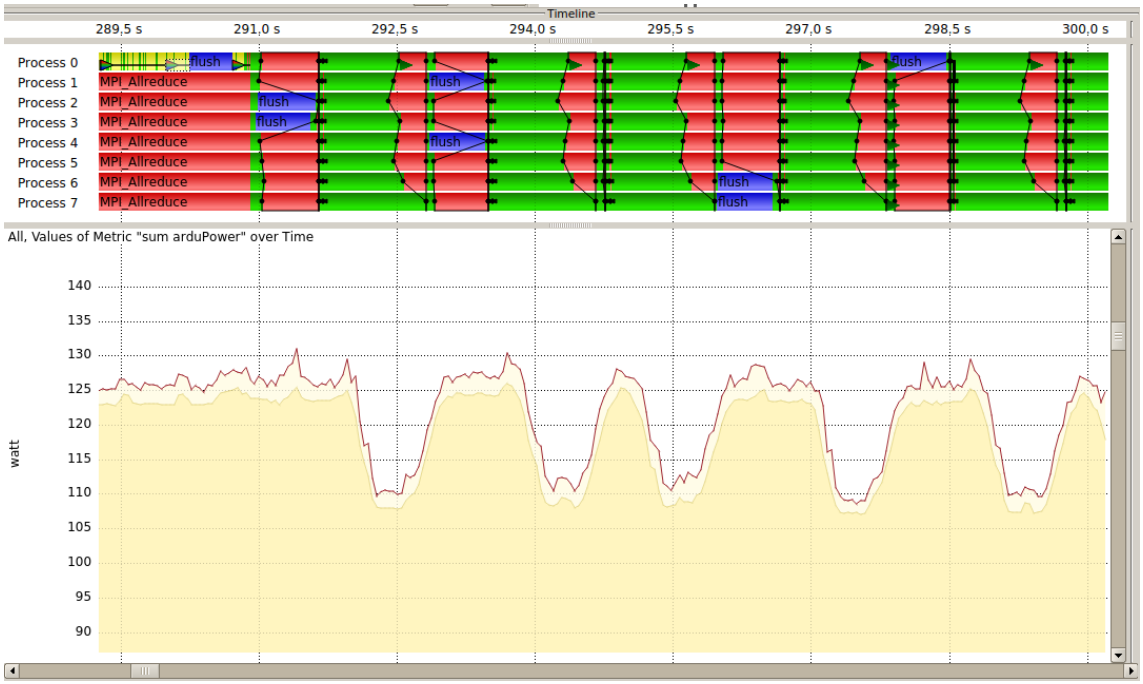


Figure 4.6.: Vampir showing the output phase alongside with the start of next iteration. Detailed description of the chart in Section 4.2.1.

last iteration is written. The writing of the output is immediately followed by a flush of the tracing-buffer which is required to obtain the trace data.
After this section the power used by the system drops to approximately 110 Watt while the computation starts again.
Shortly after that, the power needed by the system rises to approximately 125 Watt. This is caused by the need of flushing the trace buffer for processes 1 and 4. So VampirTrace dumps the collected trace data to the trace file in order to free the buffer for the next events to be traced. This is another output event which results in a higher power usage of the system. As stated in Section 3.1 it is important to note that these trace buffer flushes are related to the tracing with vampirtrace and therefore will not occur when the application is executed without additional performance tracing.
After the buffers are flushed, the needed power drops again to approximately 115 Watt while the computation continues. The next spike in the power demand graph is referring to some information about the status of the iteration, which is written to the logfile. This causes another activation of the disk and therefore results in more power needed. This pattern continues and is the same for all other iterations.

This pattern leads me to the result: I/O events need more power than computation. At the test system the power consumption rises from approximately 110 to approximately 125 Watt during the I/O, which is an increase of about 13.64 percent (see Figure 4.6).

This increased power demand is explained in more hardware components are needed

in order to perform the output. These additional components (the network for the distributed filesystem) will power up and increase the needed power as long as they are used to execute the output. When the writing of the output has finished, these components can suspend again, in order to lower the amount of power used by the system.
But this only accounts for a small amount of the observed increase (approx. 5 Watts). In Section 4.5 we will take a look on the other factor, which increases power consumption. The processors, will power up in order to perform a busy wait. As the ECOHAM model deals with very many different variables, the application performance is strongly influenced by the main memory. Because the processing is therefore slowed down by the Von Neumann bottleneck, the CPU are still capable of powering up when there are less memory accesses during such a busy wait phase.

4.4. Optimizing a serial section

In this section I will present an easy way to lower the amount of energy needed during the serial output phase.

In the current state the model uses serial output, which means that all data is first sent to the master process, which then merges them and handls the output. One can see this in the trace charts in Section 4.3, as the other processes stop in an `MPI_ALLREDUCE` operation waiting for the master to finish I/O.

There are many ways of optimizing this behaviour such as using parallel I/O or letting the other processes start with some calculation that do not depend on synchronisation with the master process. But assuming that this serial section has to be serial and the further processing depends on its output, we cannot alter this serial behaviour.

Just a small extension to the code will lead to a significant energy saving. It is remarkable that this optimization is not part of the MPI implementation.

First I present the optimization and its influence. Then that I will discuss why this adjustment is not part of the MPI implementation.

To optimize the program I added only one sleep statement which causes the processor to sleep for a given amount of time. So while the master process is in the serial section the other processes will sleep.

Figure 4.7 shows the trace of the optimized program similar to Figure 4.5 for the non-optimized version. One can clearly see the impact of the adjustment where the power demand of the system decreases during the serial output phase.. To get a more

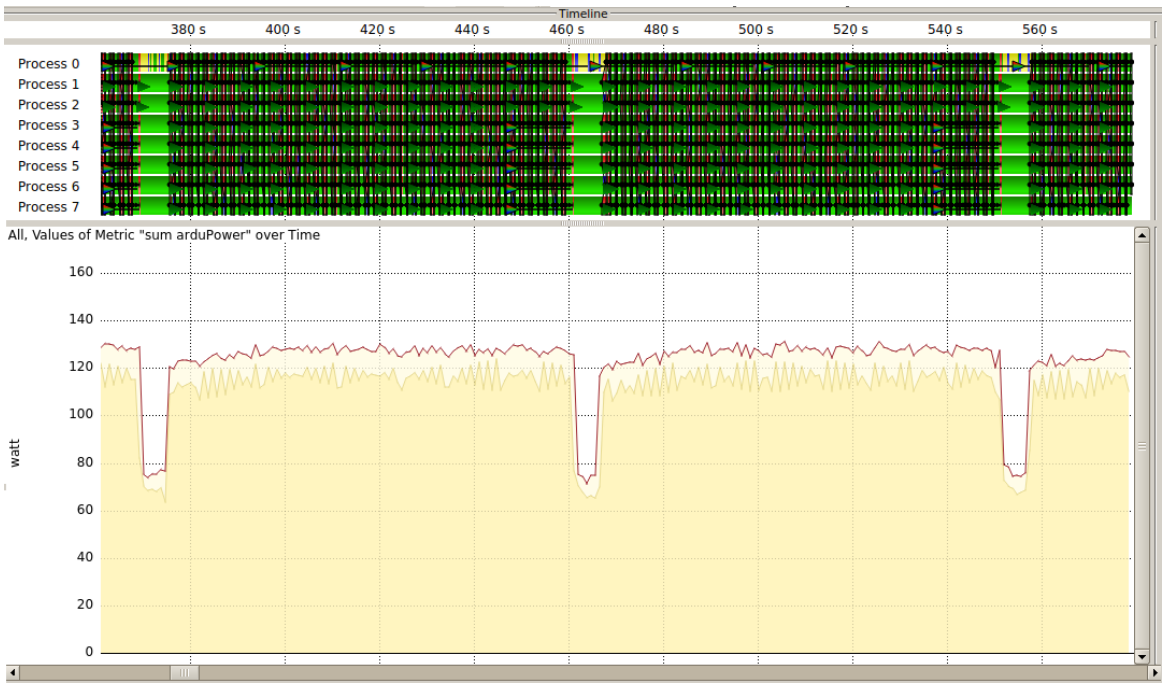


Figure 4.7.: Vampir showing two full iteration, surrounded by the section where the output is written, for the adjusted program. Detailed description of the chart in Section 4.2.1.

detailed view, Figure 4.8 shows the end of the iteration alongside the start of the next one similar to Figure 4.6. When comparing the power graphs to the graphs from Section 4.3, note the different scales of the Y-axis. In Section 4.3 they ranged from 140 to 90 Watt while in this section the axis range from 140 to 0 Watt in order to show the full graph.

On the test system the power needed by the system during the serial output section has decreased to approximately 70 Watt. This is a decrease of 41.67 percent.
The power usage of all other sections of the program has not changed. This fits to the expectation, because all other parts should not be influenced by this adjustment.

When looking at the trace, one will acknowledge that the processes are not waiting in the `MPI_ALLREDUCE` call while the master process is in the I/O section, now they are following the mentioned sleep instruction which is shown as application-time in the trace.

When taking a closer look to the end of the output part, one can recognize that the time spent in the new sleep instruction ends slightly before the master has finished the I/O task. This is leaving enough time for VampirTrace to flush the full trace buffers before all processes will go into the `MPI_ALLREDUCE` call.

So the sleep instruction is finished before the processors need to be back in a high power state again in order to process the `MPI_ALLREDUCE`. Therefore the added sleep

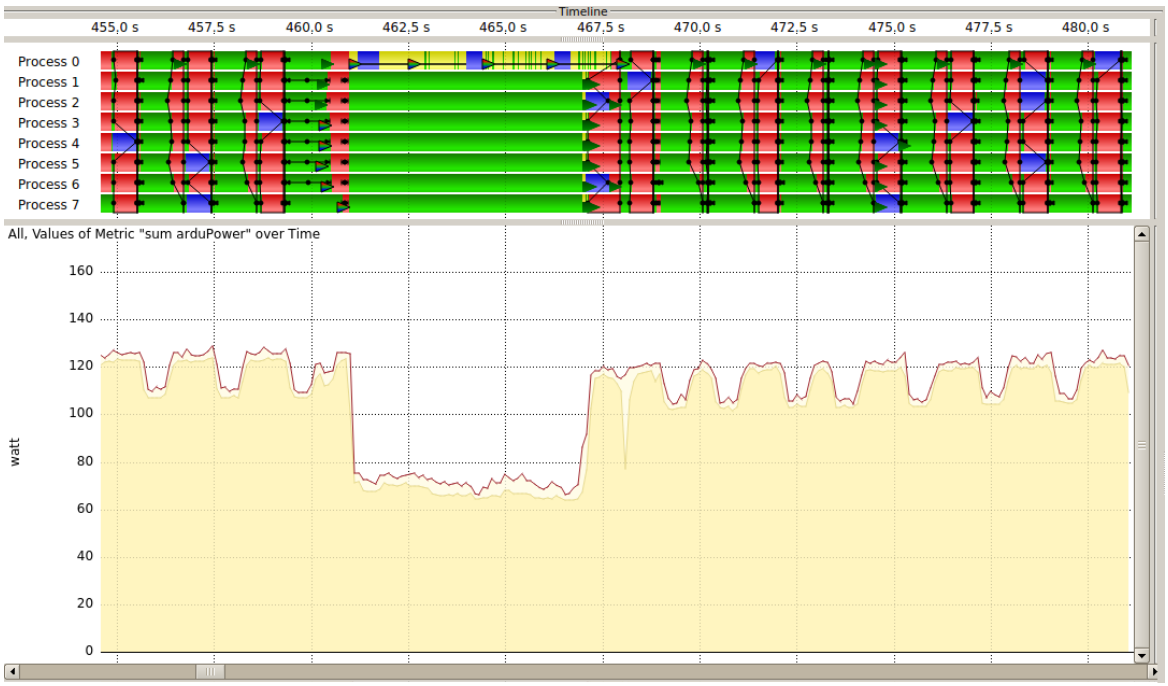


Figure 4.8.: Vampir showing the output phase, alongside with the start of next iteration, for the adjusted program. Detailed description of the chart in Section 4.2.1.

instruction has no influence on other parts of the program and does not increase the overall time needed for the output section.

This leads us to the key point of this optimization: choosing the time to sleep.
Sleeping too long will force the master process to wait for the waking up of the sleeping processes. This wasted idle time increase the runtime of the program. But sleeping too short will result in less energy saving. This is the main reason why this is not part of the MPI implementation.
The MPI implementation is optimized for performance and therefore it will not tell the processors to sleep, because waking up the processors from a sleep state will take some time. The processors are at full load the whole time. The MPI implementation implemented a "busy waiting" during which the processor can not power down, as the processor actively check if the needed information is available. This active waiting ensures that the processor continue the program with the full performance immediately when the information is available. Focus is placed to immediately continue processing, so that the latentency for messages is as low as possible. The busy waiting also ensures that the processor is in C0 state[1], when the message arrives, which is often only seen as a positive side effect. I will discuss this in more detail in the next section.

In order to achieve maximum performance all processors must be back in C0 state

[1]C0 refers to the full power state.

right before the communication finishes as then no time is wasted for waking up.
So finding the correct time requires insight in the overall behaviour of the application. It might also depend on the parameters for the program run as they may require different amounts of output.
As discussed in Section 3.1 It is important to keep the effect of the tracing in mind. In this particular case one can see in the figures of the traces, that VampirTrace has to flush the trace buffer several times during the I/O phase which further increases the overall time the other processors have to wait. Therefore the chosen time[2] to sleep will not fit for the application without VampirTrace.
With the optimization, the program runs 2071 seconds requiring 224 kJ without VampirTrace. With VampirTrace tracing its performance it needs 2768 seconds and 306 kJ. In comparison to Section 3.1 the used energy only increases about 36 percent instead of 40 percent for the not optimized version, As the adjustment also takes in account these trace buffer flushes during the I/O phase.

It might be an interesting topic for future work, to see if it is possible to extract a pattern from the waiting times in a running program and use this pattern to tell the processors to sleep in order to save energy, while the processors still wake up early enough to not decrease the performance. Therefore it should be evaluated if the additional overhead for the computation of the waiting times is worth the savings.
As work like [Ehm12] shows, the saving potential is great, but also the potential to waste energy is great.

4.5. Busy waiting in the MPI implementation

In this section I will discuss the busy waiting implemented in the MPI implementation. I will also discuss a way to not wait as busy as it is implemented now.

As noticed in Section 4.4 the MPI implementation implements a busy waiting in order to ensure maximum performance. This leads to a significant waste of energy. More than four percent of the energy is spent to perform this busy waiting. Therefore in this section I will take a closer look on this.

I will analyze it in the MPICH implementation, because this implementation is more efficient (see Section 3.2).

```
if (!MPID_Request_is_complete(request_ptr))
    {
    MPID_Progress_state progress_state;

    MPID_Progress_start(&progress_state);
        while (!MPID_Request_is_complete(request_ptr))
```

[2] I have chosen six seconds. Without instrumentation four seconds fit.

```
    {
        mpi_errno = MPID_Progress_wait(&progress_state);
        if (mpi_errno) { MPIR_ERR_POP(mpi_errno); }
    }
    MPID_Progress_end(&progress_state);
    }
```

Listing 4.1: MPI implementation to wait for the request to complete (from MPICH implementation).

Listing 4.1 shows an excerpt of the implementation of the `MPI_Wait` function. This function is called whenever a blocking MPI-function is called. One can see that the `MPID_Progress_wait()` function is called repeatedly until the request is completed. The `MPID_Progress_wait()` function "Wait for some communication since '`MPID_Progress_start`'"[3]. This waiting is implemented with a busy waiting loop polling again and again if there is communication (see Listing 4.4).

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank=0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Barrier(MPI_COMM_WORLD);
    if (rank==0)
        printf("ALL: busy 10\n");
    busy(10);
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    if (rank==0)
        {
            printf("MASTER: busy 10\n");
            busy(10);
        }
    MPI_Barrier(MPI_COMM_WORLD);

    // Continue with:
    // All Busy
    // Master only
    // All Busy

    MPI_Finalize();
    return 0;
}
```

Listing 4.2: Test program to show the busy waiting.

In order to analyze this behaviour, I have written a small test application (shown in Listing 4.2). After the initial `MPI_Barrier` all processes call the busy function which

[3]MPICH Source code file mpiimpl.h, line 3427 (version 3.2)

will simulate that the process is doing some work (Listing 4.3). After the processes have synchronized again, only the master process will be busy in the next step. All other processes will be waiting in the next `MPI_Barrier`. These two steps will repeat once, then in the last section all processes are busy again. As the program has 5 sections with 10 seconds each, we would expect that one run needs a minimum of 50 seconds to finish.

```
// keep the calling process busy for a given amount of time
void busy(double s)
{
    struct timeval start_time;
    struct timeval cur_time;
    gettimeofday(&start_time, NULL);
    int i=0;
    double time= 0.0f;

    while (time < s)
    {
        i++;// do something
        gettimeofday(&cur_time, NULL);
        time = (cur_time.tv_sec - start_time.tv_sec)
        + (cur_time.tv_usec - start_time.tv_usec) * 1e-6;
    }
}
```

Listing 4.3: The busy function used in Listing 4.2.

I have executed this test program with the normal MPICH implementation as well as with the small adjustment shown in Listing 4.4. This adjustment alters the behaviour of the MPI implementation, so that the processors sleep between the polling.

```
1     int stime=0;
2     while (MPID_nem_queue_empty (MPID_nem_mem_region.my_recvQ) ||
         ↪ !MPID_nem_recv_seqno_matches (MPID_nem_mem_region.my_recvQ))
3     {
4         usleep(stime);
5         stime++;
6         // do the actual polling...
```

Listing 4.4: Changes to the `MPID_nem_mpich_blocking_recv function` (Lines 1, 4 and 5 are new).

In [FGPT16], several ways of reducing the energy consumption while waiting on a lock, which guards some critical section, are discussed. Other techniques, which are not based on sleeping between polling, rely on the process currently holding the lock, that to wake up the waiting process. In the context of locks to secure a critical code section, this is not an optimal solution because it requires additional time when releasing the lock to figure out who is waiting and which of the waiting processes is eligible to hold the lock next for fairness reasons. This additional overhead might overcome the benefit of setting the processor to sleep.

This is especially true for MPI communication, in particular for immediate send operations, where the sending process marks the sendbuffer, so that it can not be used anymore. The sending process can now start in the background at any time. After this immediate send, the process immediatly continues without taking care if the receiving process is ready to receive the data. Then it is not feasible to spend any effort to try to wake up the receiving process, as it may not be ready to receive the data yet. So techniques that rely on the sender to wake up the process, which is waiting for the transmission, is incompatible with the concept of the immediate send/receive operations [VVH+15], at least within one node. When one considers inter-node communication, the network hardware can activate the waiting process.
A similar approach (discussed by [VVH+15]) proposes DVFS (dynamic voltage frequency scaling). The approach proposes to lower the frequency of the CPU in order to save energy, while the CPU is busy polling if there was some communication. But this is often not feasible as it is nearly impossible to just scale down the voltage for one specific process. Appart from the problem, that the operating system might swich the process to another core in between, scaling down the voltage of a CPU will influence all of its cores and therefore might also slow down other cores which are busy with computation.

Therefore I have chosen to sleep between polling. The time that the processor sleeps between two polls increases with every time the polling loop is executed. The intention behind the adjustment is that at the beginning one expects that the application is efficient and does not waste much runtime waiting on other processes. Therefore it is best to sleep only for short intervals to guarantee that the processing can continiue as soon as the communication finished. But when the communication does not finish early, the importance of saving energy grows. While the importance of an immediate continuation of the processing declines, as the wasted waiting time is much larger than the additional time spent in the non busy waiting.
In my particular test (Listing 4.2) the non busy waiting takes 0.001 seconds longer.[4] This is very small in relation to the 10 seconds the process was waiting anyhow.

Figure 4.9 shows the power usage of the test program (Listing 4.2) with the standard implementation (busy waiting) in green and the power usage with sleep between polling in blue.
One can see that in the sections where only one process is active the power demand of the system is halved when the other processors sleep between polling. This is a significant saving.

But this huge saving is not without an impact on the runtime. In this particular test the runtime increases by approximately 0.002 seconds.

In the worst case, the communication is finished right after the processor goes to sleep between two pollings. Let Y denote the time (in microseconds), the processor

[4] calculated with the function h of Figure 4.12

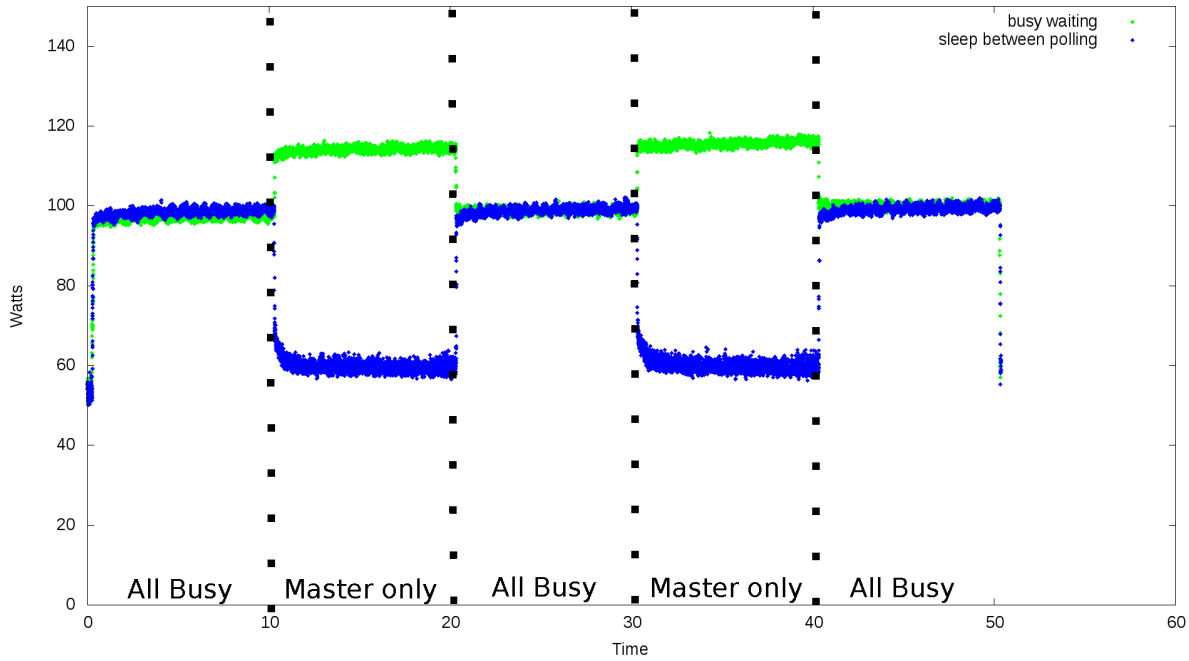


Figure 4.9.: Power usage of the test program (Listing 4.2) with busy waiting (green) and sleep between polling (blue).

waited until that point. It can be written as $Y = \sum_{i=1}^{x} i$ (+ the additional time needed for the actual polling, which I will leave out here). So before the next polling, the processor will first sleep for $x + 1$ microseconds. To get the maximum proportion of the additional waited time I define $f(Y) = \frac{x+1}{Y}$, which describes the proportion of the additional time the processor has to wait in the worst case depending on the time that was spent waiting before. The sum $Y = \sum_{i=1}^{x} i$, where x describes the number of pollings done by the processor so far, describes a triangular number and can therefore also be written as

$$Y = \sum_{i=1}^{x} i \tag{4.1}$$

$$Y = \frac{x(x+1)}{2} \tag{4.2}$$

$$2Y = x \cdot (x+1) \tag{4.3}$$

$$2Y = x^2 + x \tag{4.4}$$

$$2Y + \frac{1}{4} = x^2 + x + \frac{1}{4} \tag{4.5}$$

$$2Y + \frac{1}{4} = x^2 + 2 \cdot \frac{1}{2}x + \frac{1}{4} \tag{4.6}$$

$$2Y + \frac{1}{4} = (x + \frac{1}{2})^2 \tag{4.7}$$

$$\sqrt{2Y + \frac{1}{4}} = x + \frac{1}{2} \tag{4.8}$$

$$\frac{1}{2} + \sqrt{2Y + \frac{1}{4}} = x + 1 \tag{4.9}$$

with Y as the time, which was spent waiting and x as the number of pollings done by the processor so far. Therefore is $f(Y) = \frac{x+1}{Y} = \frac{0.5+\sqrt{2Y+0.25}}{Y}$.

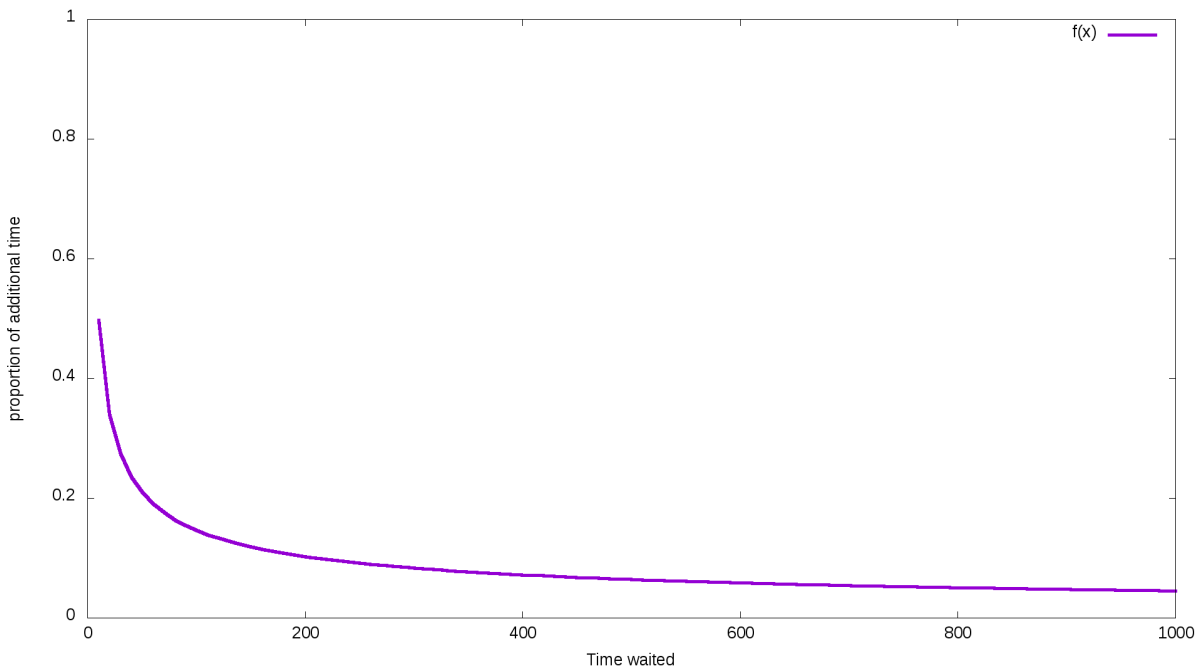


Figure 4.10.: Plot of $f(x) = \frac{0.5+\sqrt{2x+0.25}}{x}$ which describes the proportion of the additional time the processor have to wait in the worst case depending on the time that was waited before.

For this particular test system, when the proportion of additional time is below 1, no additional energy is wasted, as the non busy waiting consumes only half as much power as the busy waiting does, waiting twice as much time will result in no additional energy wasted. In the plot of the function in Figure 4.10, one can see that the proportion of additional waited time declines with more time spent in waiting. As the the proportion of the additional waiting time is higher at the beginning, it is a good idea to perform the first few polls without any additional wait. So that the addditional waiting starts a bit later, when small extra delays are more acceptable, as they are relatively small in proportion to the the time, wich was spent waiting for the message to arrive.

To get a overview on the absolute values of the additional waited time, Figure 4.11 shows the waited time. The time for busy wait is shown in bluish green and the time when the implementation waits between polling in purple. One can see that the time

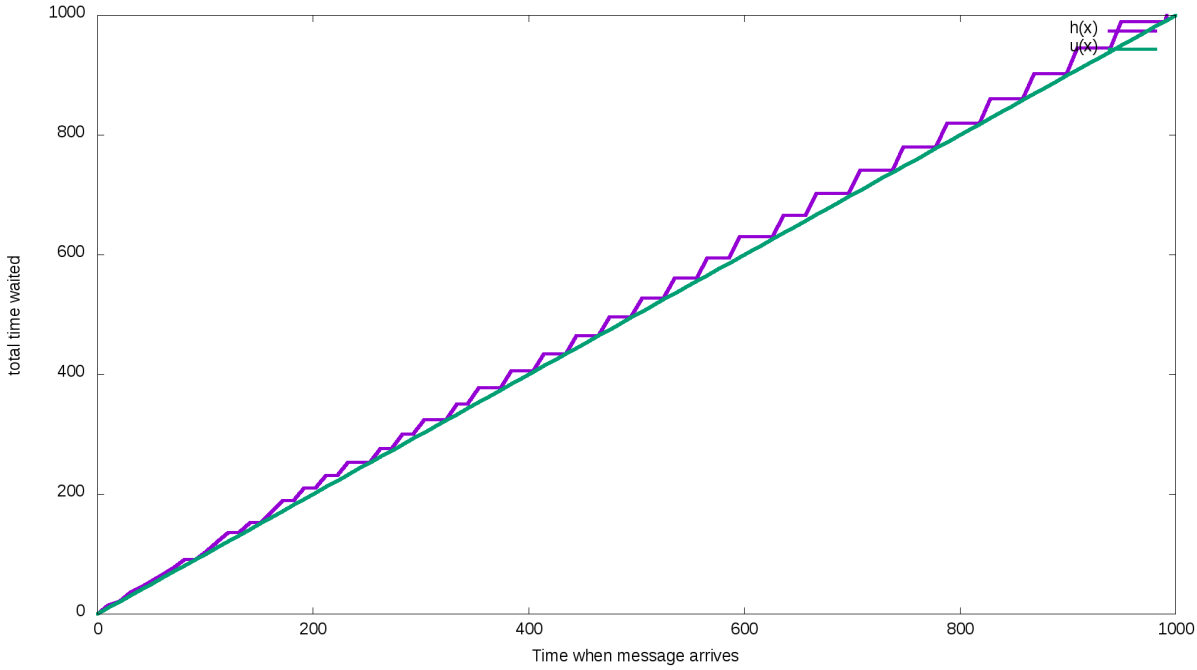


Figure 4.11.: Plot of the waited time. Busy wait $u(x)$ (bluish green), wait between polling $h(x)$ (purple).

waited is a sort of step function. Every step is linked to the time which was waited between two polls. The function which describes the waiting time can be denoted as $h(Y) = 0.5 * (\lceil \sqrt{2Y + 0.25} + 0.5 \rceil - 1) * \lceil \sqrt{2Y + 0.25} + 0.5 \rceil$ with x as the time when the message arrives. As above, the time spend in waiting is a triangular number $Y = \frac{x*(x+1)}{2}$. With x as the number of pollings the processor have done so far. As the processor can only poll for an integer amount of times, we denote $Y = \frac{\lceil x \rceil * (\lceil x \rceil + 1)}{2} = \frac{(\lceil x+1 \rceil - 1) * \lceil x+1 \rceil)}{2}$. We can now get the function h using equation 4.9. When looking at it on a larger scale like in Figure 4.12 it shows that the loss of runtime when waiting between polling is negligible in comparison to the time that was spent waiting anyhow, as the two lines are so close they can not be distinguished. It also shows, that the sleeping time between two polls does not increase very fast. In fact, its increase will even slow down as the time until the next poll and therefore the next increase of the waiting time becomes longer and longer. In this setup this effect prevents the waiting time to become too large so that the latentency grows too much. Nevertheless, it is a good idea to set a maximum waiting time, as the results have shown that it will still be enough to allow the processor to power down. Looking at Figure 4.9, one can see that the power demand of the system does not decline with an increase in the waited time after the first seconds of waiting.

Therefore I suggest that in future MPI implementations programmers can choose whether they expect a MPI call to be called at different times from the involved processes and therefore, the application might have to wait longer, or they expect it to be called by all involved processes at roughly the same time. When the call is expected to not

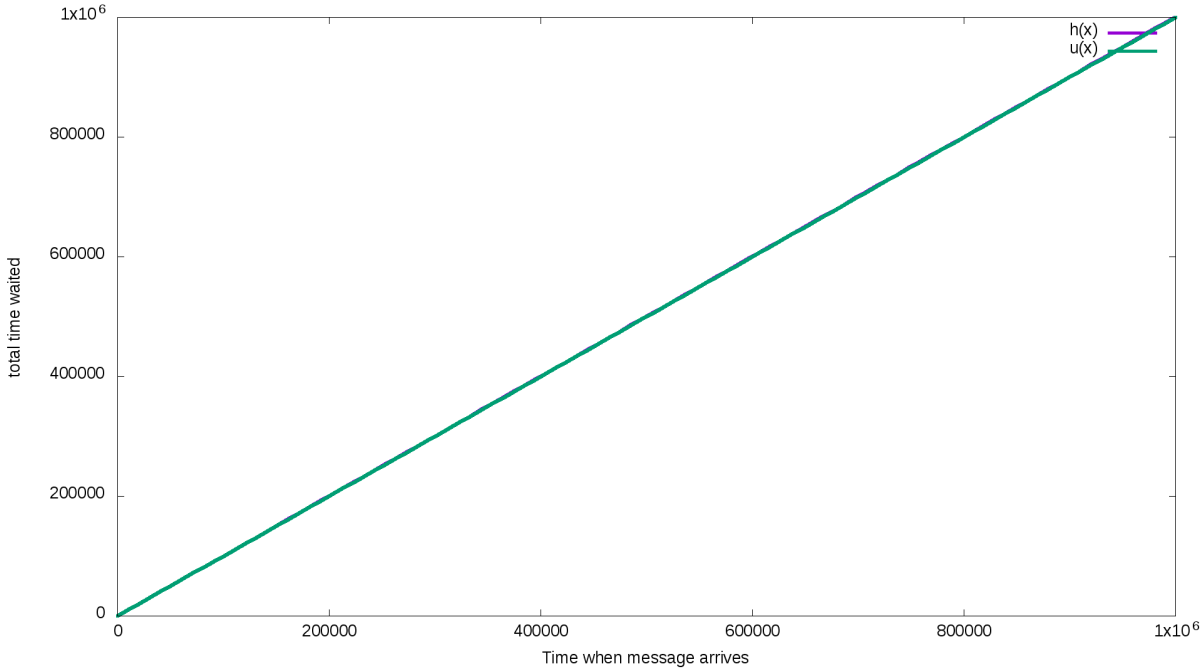


Figure 4.12.: Plot of the waited time on a larger scale. Busy wait $u(x)$ (bluish green), wait between polling $h(x)$ (purple).

be executed by all involved processes at roughly the same time, the MPI implementation should not wait as busy as it does now. Otherwise the busy waiting ensures the maximal performance and is still the better option for the calls where all involved processes are expectd to be at rughly the same time. This counts in particular for any use of `MPI_Barrier`, as the concept of `MPI_Barrier` is to synchronize asynchronous processes.

It is very interesting to see that the developers of the MPICH implementation have been thinking about this problem for several years now. But they have not found a pleasant solution yet. As "MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard"[5] the small latency introduced with a proposal like this will hurt the message rate and is therefore not compatible with their goal of a high performance implementation.

Considering that in an efficient application the waiting time is already minimized, one should ask the question, if it is really worth the effort. For the ECOHAM application more than 14 kJ are wasted performing busy waiting for the I/O to finish. This accounts for 4 percent of the energy needed by the ECOHAM model.
As Minartz determined in his PhD Thesis, also savings in the lower percent range are in general worth the effort [Min13].

[5]Source: `https://www.mpich.org/` [as of November 28, 2016]

4.6. Energy usage of communication

In this section we will take a closer look on the energy needed for the communication between the processes.

Another very important aspect of parallel applications is communication. In this section I will analyze the amount of energy needed to perform the communication.

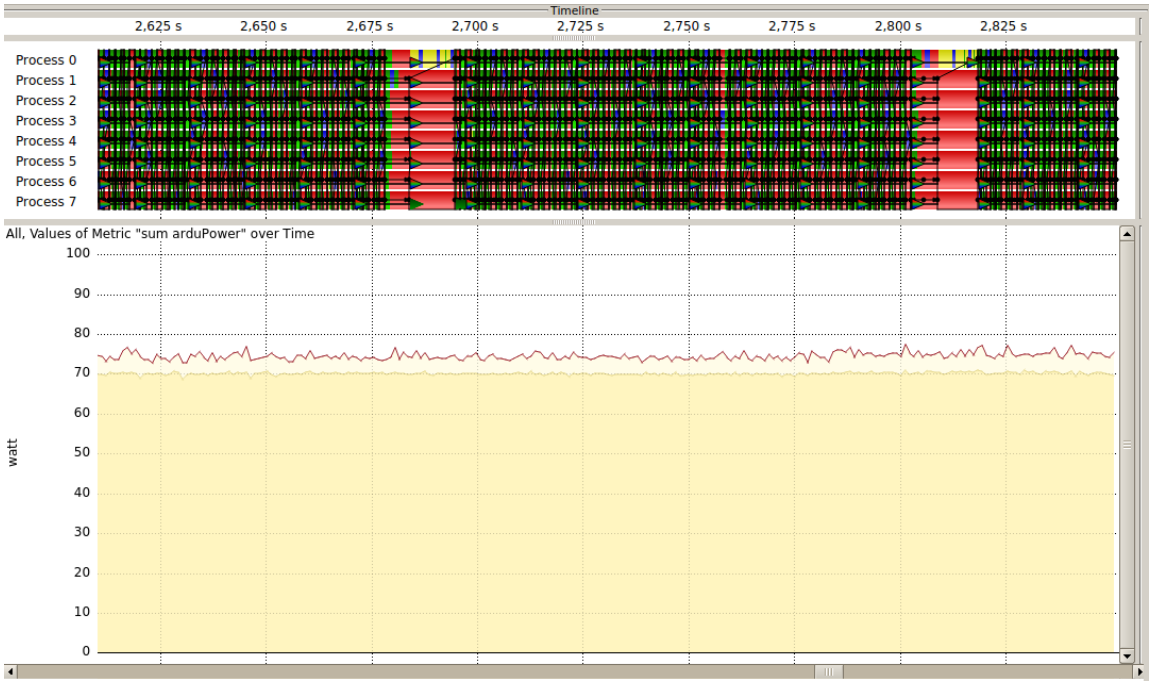


Figure 4.13.: Vampir showing the program running on 4 different nodes. The power chart is only the power usage of one node. Detailed description of the chart in Section 4.2.1.

In Figure 4.13 one can see the trace of the program when running on four different computation nodes. So every node hosts only 2 processes. The shown power consumption is only the consumption of one node. The master process which does the I/O is not hosted on the node where the power measurement was done, in order to see only the influence of communication.

The result is that the power needed for communication is indistinguishable from the power demand during computation.

To further strengthen this result one also can have a look at Figure 4.14, which shows this part in more detail. I have also displayed the minimum (blue) line. When looking at the average line it is clearly a straight line and the power consumed does not change significantly.

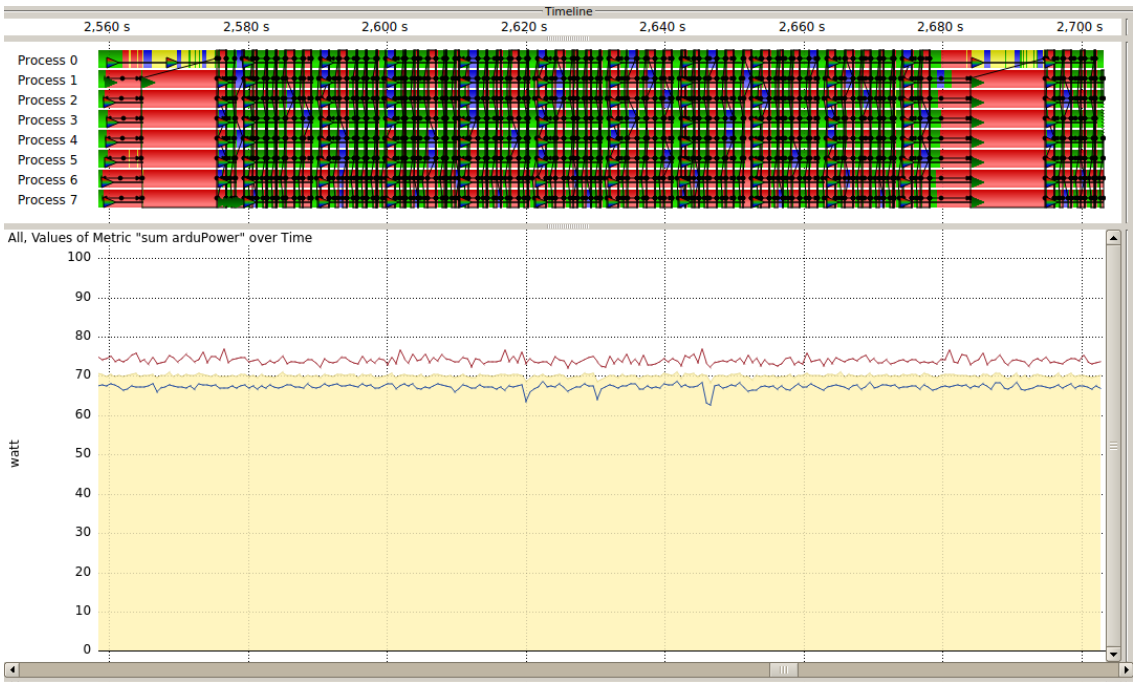


Figure 4.14.: Vampir showing the program running on 4 different nodes. The power chart is only the power usage of one node. Detailed description of the chart in Section 4.2.1.

When looking at an even more detailed view like in Figure 4.15, one can see that the power needed for the system stays at approximately 70 Watt. It does not change over time. The effect discovered earlier (Section 4.5), that the busy waiting needs more power, cannot be observed in this setup. As only two processors on the measured node were active, the effect would not be as large as in Section 4.3. Furthermore, as discussed in Section 4.5, when considering inter-node communication there are ways to wait until a message arrives, because the network hardware can awake all processes that are waiting for the message. Also for communication between several nodes the network hardware has to be in a higher power state the whole time, as the processes will communicate very often. Therefore the power needed during the execution of the application does not change, so that we cannot distinguish the communication phases from the computation phases in the power graph.

It is also interesting to see that all 4 nodes together use much more power then one node hosting all the processes. As every node uses approximately 70 Watt, together they need 280 Watt. This is more then twice of the amount needed by only one node hosting all of the 8 processes, which only needs approximately 125 Watt.
The reason for that is, that some components of a computation node cannot shut down in order to save energy. Components like the mainboard or memory has to be switched on. Without them the node cannot be available as the operating system services are not available. Therefore the idle power consumption is approx. 50 Watt at the test system.

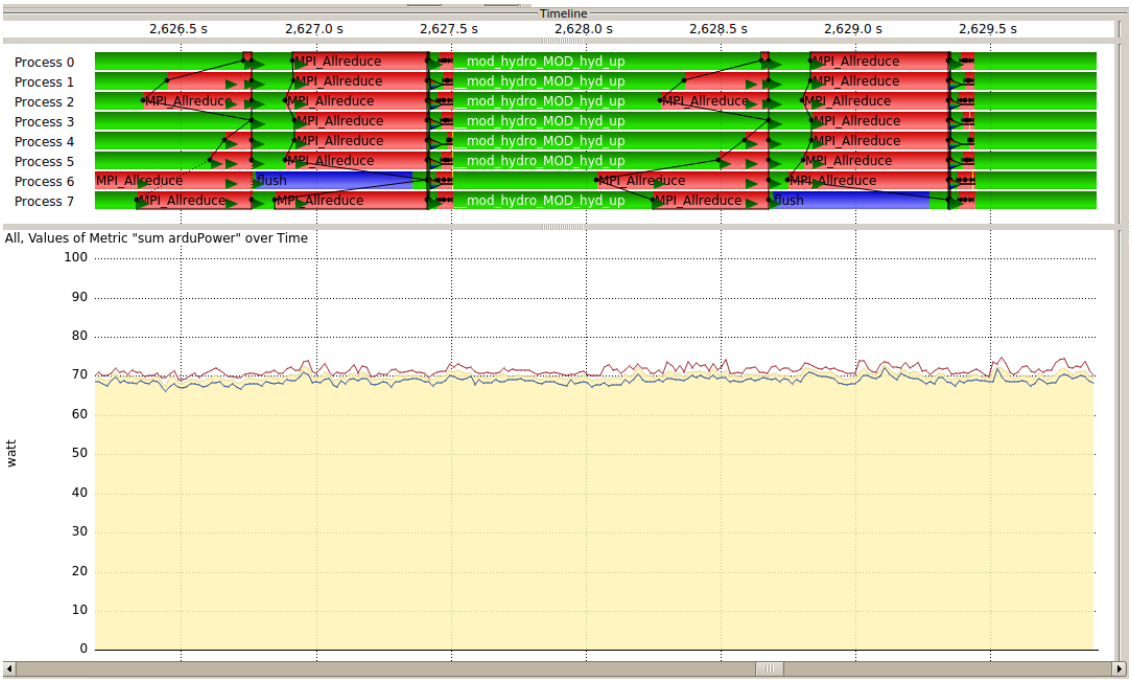


Figure 4.15.: Vampir showing the program running on 4 different nodes. The power chart is only the power usage of one node. Detailed description of the chart in Section 4.2.1.

The measured node uses 257 kJ in the 3772 seconds the job needs to complete. When the application is executed with all 8 processes on one node it uses 228 kJ in only 2103 seconds of runtime. This high increase of the needed runtime (about 79 percent) is explained in the longer time the communication takes, as communication between two nodes takes much longer than communication between two processes within the same node. As the runtime increases, the needed energy increases as well. In this particular setup, the energy needed for only one node hosting a quarter of the processes, while the other processes are running on other nodes, is even higher than the amount of energy needed when the node hosts all 8 processes.

So when distributing an application onto several nodes, one should keep in mind that more power is used. This should be considered for performance reasons anyway as additional time is needed to communicate between nodes and the not used nodes stay available for other applications.
In terms of energy efficiency using another processor from an already allocated node is not as costly as allocating another node for the next needed processor. In particular, this should be considered if more resources are needed that are nodewide and not per processor, like network bandwidth. One should evaluate if the increase in performance is worth the high increase of the energy needed.

5. Conclusion

This thesis has shown that it is important to take care about energy consumption.

In Section 4.2 the result have shown that optimization of the initialization and finalization part is indeed not as important as optimizing the longer main part. This counts especially for the initialization part, as the test system always need slightly less power in the first few seconds of the program. Therefore the rule of thumb that it is more advisable to optimize the part of the program which is executed most of the time (Section 2.1), also generally holds when optimizing an application for less energy consumption, as the saving potential is larger there.

Nevertheless, Section 4.3 showed that the output parts, where more components are needed, uses more energy. As the output is a serial section, one can lower the amount of energy significant when the not used processors are powered down, as shown in Section 4.4.
This is due to the fact that the MPI implementation implements a busy waiting. Therefore in Section 4.5 a closer look on this busy waiting has shown that it is feasible to sleep between the polling in order to save energy. This will work without wasting too much additional runtime when it is expected that the processes have to wait for the other involved processes for the MPI call to complete.
Future work may evaluate if it is feasible to power down for all calls of `MPI_Barrier`.

The main result of Section 4.6 is that one has to be aware that using multiple nodes consume more power than using more processors from one node. This is especially important if one wants to allocate more resources that are shared among the processors of one node, like network bandwidth or main memory. Then it has to be evaluated if these additional resources are worth the relatively large increase of the needed power.

The input data being processed does not influence the energy needed by the system, as long as the way they are being processed is the same, as shown in Section 4.1. Therefore it is not important to use energy in a pre-processing to alter the input data (e.g. scale it down to compute with lower values), as long as it is feasible for the specific application.

Section 3.1 and Section 3.2 showed that one should not forget the influence of the performance tracing and the MPI implementation when optimizing an application.
This is especially important as optimal improvements of the energy consumption require insights in the applications behaviour. When gaining the needed insight through a tracing tool, it is important to keep the effect of the tracing in mind as one wants to optimize

the version without the instrumentalization for the tracing. The task of optimizing the energy usage of an application cannot be delegated to the compiler or the MPI implementation yet.

In the future when considering optimizing an application performance, we should also consider the impact for the energy used as not all of them have a positive impact. Furthermore we should not rely on other system layers like the compiler or the hardware to do sufficient optimizations for us.

Therefore I propose, that the developers of an MPI application should be able to choose which MPI calls should be executed in a "power saving" mode and for which MPI calls the application needs the full performance. For Example this can be accomplished, when for MPI functions like `MPI_Barrier` another energy aware version like `MPI_PBarrier` (P for Power saving) will be available for the programmer to choose. Then the Programmers can decide at which moments the application does not need the full power, in order to save energy with a minimal impact on the runtime.
In the analyzed applicatioin the MPI calls while the Master process is performing the output should use a power saving version (see Section 4.4), while for the other communication the full power version is suitable, as theese calls are called by the processes at rughly the same time, so the small latentency is likely to hurt the applications performance more than the accomplished saving.
Also the operators of a computation center might want to decide that certain applications are only allowed to run with an energy aware MPI implementation, without the need of supporting two different MPI implementations and recompiling the applications. Therefore it might be a good idea to set an environment variable to indicate if the energy aware version of the MPI functions should be used. Then the operators can decide if the lower message rate is worth the power saving on a per application basis.

Bibliography

[BBC+13] Sergio Barrachina, Maria Barreda, Sandra Catalán, Manuel F Dolz, Germán Fabregat, Rafael Mayo, and ES Quintana-Ortí. An integrated framework for power-performance analysis of parallel scientific workloads. *Energy*, pages 114–119, 2013.

[BH15] Manuela Beckert and Janosch Hirsch. A low cost power measurement device to improve energy efficiency of hpc devices. `https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2014_2015/pre-1415-ardupower-report.pdf`[as of November 28, 2016], 2015.

[BSH+08] Diana Bautista, Julio Sahuquillo, Houcine Hassan, Salvador Petit, and José Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7. IEEE, 2008.

[CM10] Sangyeun Cho and Rami G Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):342–353, 2010.

[CXDM10] Xi Chen, Chi Xu, Robert P Dick, and Zhuoqing Morley Mao. Performance and power modeling in a multi-programmed multi-core environment. In *Proceedings of the 47th Design Automation Conference*, pages 813–818. ACM, 2010.

[DHK+15] Manuel F. Dolz, Mohammad Reza Heidari, Michael Kuhn, Thomas Ludwig, and Germán Fabregat. ArduPower: A low-cost wattmeter to improve energy efficiency of HPC applications. In *Sixth International Green Computing Conference and Sustainable Computing Conference (IGSC) 2015*, pages 1–8. IEEE, 12 2015.

[Ehm12] Florian Ehmke. Energy-aware instrumentation of parallel mpi applications. `http://edoc.sub.uni-hamburg.de/informatik/volltexte/2012/180/pdf/bac_ehmke.pdf` [as of November 28, 2016], 2012.

[Eur16] Eurostat. Electricity prices by type of user. `http://ec.europa.eu/eurostat/tgm/table.do?tab=table&plugin=1&language=en&pcode=ten00117` [as of November 28, 2016], 2016.

[FGPT16] Babak Falsafi, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. Unlocking energy. In *USENIX Annual Technical Conference (USENIX ATC '16)*. USENIX, 2016.

[GLPB15] Fabian Große, Christian Lindemann, Johannes Pätsch, and Jan O. Backhaus. The influence of winter convection on primary production: A parameterisation using a hydrostatic three-dimensional biogeochemical model. *Journal of Marine Systems*, pages 138–152, 2015.

[Goe13] Fethullah Goekkus. Energy efficient programming. `https://files.ifi.uzh.ch/hilty/t/examples/bachelor/Energy_Efficient_Programming_G%C3%B6kkus.pdf` [as of November 28, 2016], 2013.

[Kle14] Frank Klemm. Entwicklung der strukturgrößen von 1971 bis 2014. `https://commons.wikimedia.org/wiki/File:IntelFeatureSize.png` [as of November 28, 2016], 2014.

[Lar11] Petter Larsson. Energy-efficient software guidelines. *Intel Software Solutions Group, Tech. Rep*, 2011.

[LMBB+10] Hermann-J. Lenhart, David K. Mills, Hanneke Baretta-Bekker, Sonja M. Van Leeuwen, Johan Van Der Molen, Job W. Baretta, Meinte Blaas, Xavier Desmit, Wilfried Kühn, Geneviève Lacroix, et al. Predicting the consequences of nutrient reduction on the eutrophication status of the north sea. *Journal of Marine Systems*, 81(1):148–170, 2010.

[LPD13] James H Laros, Phil Pokorny, and David DeBonis. Powerinsight-a commodity power measurement capability. In *Green Computing Conference (IGCC), 2013 International*, pages 1–6. IEEE, 2013.

[LPMK12] Ina Lorkowski, Johannes Pätsch, Andreas Moll, and Wilfried Kühn. Interannual variability of carbon fluxes in the north sea from 1970 to 2006–competing effects of abiotic and biotic drivers on the gas-exchange of co 2. *Estuarine, Coastal and Shelf Science*, 100:38–57, 2012.

[Lud15a] Thomas Ludwig. Energieeffizienz und total cost of ownership. `https://files.wr.informatik.uni-hamburg.de/index.php/s/861a1cc4ebd9bea52e95e635dac45b94/download?path=%2F&files=hr-1516-tco.pdf` [as of November 28, 2016], 2015.

[Lud15b] Thomas Ludwig. Leistungsoptimierung sequentieller anwendungen. `https://files.wr.informatik.uni-hamburg.de/index.php/s/861a1cc4ebd9bea52e95e635dac45b94/download?path=%2F&files=hr-1516-seqopt.pdf` [as of November 28, 2016], 2015.

[Mar13] Peter Marwedel. Embedded system hardware -processing-. http://ls12-www.cs.tu-dortmund.de/daes/media/documents/teaching/courses/ws1213/es/lecture/es-marw-3.2-processing.pdf [as of November 28, 2016], 2013.

[MHEZ13] Jason Mair, Zhiyi Huang, David Eyers, and Haibo Zhang. Myths in pmc-based power estimation. In *European Conference on Energy Efficiency in Large Scale Distributed Systems*, pages 35–50. Springer, 2013.

[Min13] Timo Minartz. *Design and Evaluation of Tool Extensions for Power Consumption Measurement in Parallel Systems.* PhD thesis, Universität Hamburg, Von-Melle-Park 3, 20146 Hamburg, 2013.

[MRZ+03] Ramesh Mishra, Namrata Rastogi, Dakai Zhu, Daniel Mossé, and Rami Melhem. Energy aware scheduling for distributed real-time systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9–pp. IEEE, 2003.

[Naj94] Farid N Najm. A survey of power estimation techniques in vlsi circuits. *Urbana*, 51:61801, 1994.

[Ran10] Parthasarathy Ranganathan. Recipe for efficiency: principles of power-aware computing. *Communications of the ACM*, 53(4):60–67, 2010.

[Rix16] Kim Rixecker. Google senkt stromverbrauch im rechenzentrum – mittels künstlicher intelligenz. http://t3n.de/news/google-stromverbrauch-ki-deepmind-727798/ [as of November 28, 2016], 2016.

[SBM09] Karan Singh, Major Bhadauria, and Sally A McKee. Real time power estimation and thread scheduling via performance counters. *ACM SIGARCH Computer Architecture News*, 37(2):46–55, 2009.

[Sut09] Herb Sutter. The free lunch is over. http://www.gotw.ca/publications/concurrency-ddj.htm [as of November 28, 2016], 2009.

[UBPEG02] Elif Uysal-Biyikoglu, Balaji Prabhakar, and Abbas El Gamal. Energy-efficient packet transmission over a wireless link. *IEEE/ACM Trans. Netw.*, 10(4):487–499, August 2002.

[VVH+15] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar (DK) Panda, Darren Kerbyson, and Adolfy Hoisie. A case for application-oblivious energy-efficient mpi runtime. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 29:1–29:12, New York, NY, USA, 2015. ACM.

[WAT09] Adam Wierman, Lachlan LH Andrew, and Ao Tang. Power-aware speed scaling in processor sharing systems. In *INFOCOM 2009, IEEE*, pages 2007–2015. IEEE, 2009.

[Zei12] Süddeutsche Zeitung. Energiebilanz des internet-konzerns - google verbraucht strom für eine ganze stadt. `http://www.sueddeutsche.de/digital/energiebilanz-des-internet-konzerns-google-verbraucht-strom-fuer-eine-ganze-stadt-1.1141053` [as of November 28, 2016], 2012.

Appendices

A. Measured energy usage data

MPI-Implementation	Instrumentization	Runtime(seconds)	Energy(kJ)	Watts
OpenMPI	none	2107	-	-
OpenMPI	none	2084	-	-
OpenMPI	none	2079	-	-
MPICH	manual	1934	217.550	112.48
MPICH	manual	1997	224.689	112.51
MPICH	manual	1938	217.912	112.44
OpenMPI	manual	2069	232.420	112.33
OpenMPI	manual	2032	227.504	111.96
OpenMPI	manual	2037	228.330	112.09
OpenMPI	VampirTrace	2769	320.365	115.69
OpenMPI	VampirTrace	2773	320.961	115.72
OpenMPI	VampirTrace	2745	317.787	115.76

Table A.1.: Measured energy consumption. Watts are the average that the system needed during the runtime.

MPI-Implementation	Instrumentization	Runtime(seconds)	Energy(kJ)	Watts
MPICH	manual	2033	221.305	108.85
MPICH	manual	2077	225.438	108.54
MPICH	manual	2054	223.476	108.80
OpenMPI	manual	2030	219.389	108.07
OpenMPI	manual	2071	224.489	108.39
OpenMPI	manual	2103	223.704	106.37
OpenMPI	VampirTrace	2768	306.137	110.59
OpenMPI	VampirTrace	2775	306.917	110.60
OpenMPI	VampirTrace	2755	304.855	110.65

Table A.2.: Measured energy consumption with the adjustment presented in Section 4.4. Watts are the average that the system needed during the runtime.

B. Usage of pmlib

```
1  #include "vt_plugin_cntr.h" /* VT plugin interface definition */
2
3  int main(int argc, char** argv) {
4
5      int i = 0;
6      char* command;
7
8      if (argc >= 2) {
9          command = argv[1];
10
11     } else {
12         printf("Usage: \n arg1 is the command to measure\n");
13         return 0;
14     }
15
16     printf("init Power measurement\n");
17     if (init() == E_INIT_FAILURE) {
18         printf("init failure\n");
19         exit(-1);
20     }
21     int32_t counterID[16];
22
23     // Init measurement for all 16 Lanes
24     char* metric_pattern = "10.0.0.20_ArduPowerDevice_II"; // length
       ↪ = 28
25
26     char* metric_name = malloc(sizeof(char) * 28);
27
28     memcpy(metric_name, metric_pattern, 28);
29
30     metric_name[27] = 0;
31
32     // one digit
33     for (i = 0; i < 10; ++i) {
34         metric_name[26] = '0' + i;
35         //printf("%s\n", metric_name);
36         vt_plugin_cntr_metric_info *info =
               ↪ get_event_info(metric_name);
37         counterID[i] = add_counter(info->name);
38     }
39     // 2 digits
40     metric_name[26] = '1';
41     for (i = 10; i < 16; ++i) {
42         metric_name[27] = '0' + i - 10;
```

```
        //printf("%s\n", metric_name);
        vt_plugin_cntr_metric_info *info = get_event_info(metric_name);
        counterID[i] = add_counter(info->name);

    }

    free(metric_name);

    printf("START: %s \n", command);
    fflush(stdout);

    system(command);

    printf("FINISHED\n");
    printf("collecting power data...\n");
    for (i = 0; i < 16; ++i) {

        //the Result Values will be printed to a File so we do not need them here
        vt_plugin_cntr_timevalue *resValues = NULL;
        get_all_values(counterID[i], &resValues);
    }

    finalize();
    printf("done\n");

    return 0;
}
```

Listing B.1: Wrapper program to measure a not instrumented application.

The program takes a command as argument and will execute this command and measure its energy usage. The measured data will be written in the directory where the program was located. To use this program one has to link it with the pmlib library.

C. Table from Goekkus's thesis

Computational Efficiency	**1. Problem**: Insensitive choice of algorithms and data structures lead to significant energy wasting. **Solution**: Careful and targeted selection of algorithms and data structures.
	2. Problem: Careless programming of loops and overuse of spinning and polling loops lead to inefficiency. **Solution**: Designing the loops by following the rules provided in section 3.1.1.
	3. Problem: Single threaded applications are inefficient and waste energy. **Solution**:Multithreading.
	4. Problem: Use of Scalar C-code rather than Vectorizing lead to inefficient software that wastes energy. **Solution**: Vectorization and instruction sets.
	5. Problem: Choosing an inefficient programming language can lead to huge energy wasting. **Solution**:Careful and targeted selection of programming language.
	6. Problem: Not exploiting the well-proven energy efficient solutions can lead to inefficient software. **Solution**: Using energy efficient libraries and drivers.
Data Efficiency	**7. Problem**: Unnecessary data movement leads to energy wasting. **Solution**: Minimizing data movement.
	8. Problem: Unnecessary memory access leads to energy wasting. **Solution**: Minimizing memory access.

Table C.2.: Summary of the problems and solutions: Application software efficiency. Source: [Goe13]

Computational Efficiency	**1. Problem**: Insensitive choice of algorithms and data structures lead to significant [wasting of runtime]. **Solution**: Careful and targeted selection of algorithms and data structures.
	2. Problem: Careless programming of loops and overuse of spinning and polling loops lead to inefficiency. **Solution**: Designing the loops by following the rules provided in section 3.1.1.
	3. Problem: Single threaded applications are inefficient and waste [runtime]. **Solution**:Multithreading.
	4. Problem: Use of Scalar C-code rather than Vectorizing lead to inefficient software that wastes [time]. **Solution**: Vectorization and instruction sets.
	5. Problem: Choosing an inefficient programming language can lead to huge [wasting of time]. **Solution**:Careful and targeted selection of programming language.
	6. Problem: Not exploiting the well-proven [fast] solutions can lead to inefficient software. **Solution**: Using [fast and efficient] libraries and drivers.
Data Efficiency	**7. Problem**: Unnecessary data movement leads to [wasting of runtime]. **Solution**: Minimizing data movement.
	8. Problem: Unnecessary memory access leads to [wasting of runtime]. **Solution**: Minimizing memory access.

Table C.4.: Table C.2 with runtime instead of energy.

D. The ArduPower device installed

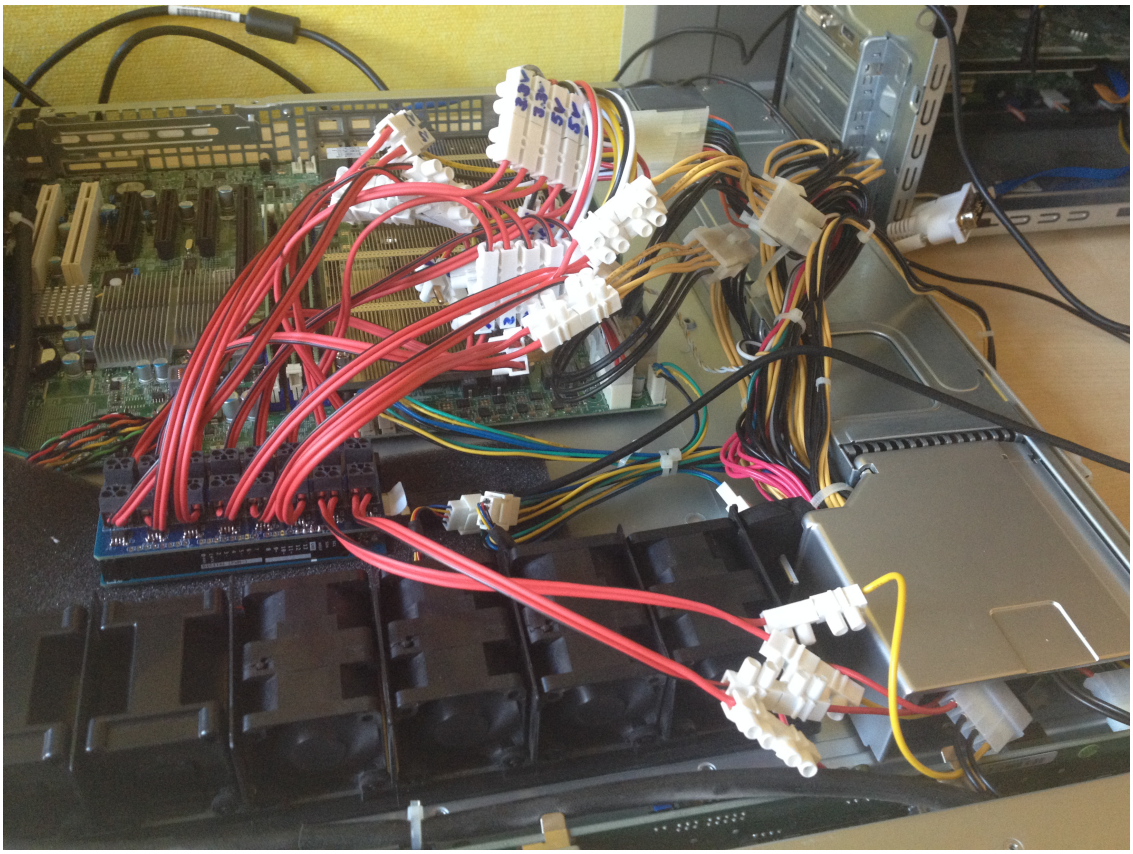


Figure D.1.: ARDUPOWER wattmeter installed. Source [BH15]

List of Figures

1.1. Intel CPU Trends. Source [Sut09] 5
1.2. Power density of CPU. X-axis denote the transistor size. Source [Mar13] 6

2.1. Illustration of the grid used by the ECOHAM model. 11
2.2. Illustration on how pmlib works. Source [BBC+13] 13
2.3. Components of the ARDUPOWER wattmeter. Source [DHK+15] 15

3.1. Illustration of the timing error. The black boxes illustrate one sample. The red dots show when exactly the measured value was read. 19
3.2. Energy usage data summed up by order of points (blue) and using linear interpolation (yellow). 19

4.1. Energy used to simulate different months. 22
4.2. Overview of the whole trace. Detailed description of the chart in Section 4.2.1. 23
4.3. Vampir showing the initialize section of the application. Detailed description of the chart in Section 4.2.1. 24
4.4. Vampir showing the finialization section of the application. Detailed description of the chart in Section 4.2.1. 25
4.5. Vampir showing a full iteration surrounded by the section where the output is written. Detailed description of the chart in Section 4.2.1. . . . 26
4.6. Vampir showing the output phase alongside with the start of next iteration. Detailed description of the chart in Section 4.2.1. 27
4.7. Vampir showing two full iteration, surrounded by the section where the output is written, for the adjusted program. Detailed description of the chart in Section 4.2.1. 29
4.8. Vampir showing the output phase, alongside with the start of next iteration, for the adjusted program. Detailed description of the chart in Section 4.2.1. 30
4.9. Power usage of the test program (Listing 4.2) with busy waiting (green) and sleep between polling (blue). 35
4.10. Plot of $f(x) = \frac{0.5+\sqrt{2x+0.25}}{x}$ which describes the proportion of the additional time the processor have to wait in the worst case depending on the time that was waited before. 36
4.11. Plot of the waited time. Busy wait $u(x)$ (bluish green), wait between polling $h(x)$ (purple). 37
4.12. Plot of the waited time on a larger scale. Busy wait $u(x)$ (bluish green), wait between polling $h(x)$ (purple). 38

4.13. Vampir showing the program running on 4 different nodes. The power chart is only the power usage of one node. Detailed description of the chart in Section 4.2.1. 39
4.14. Vampir showing the program running on 4 different nodes. The power chart is only the power usage of one node. Detailed description of the chart in Section 4.2.1. 40
4.15. Vampir showing the program running on 4 different nodes. The power chart is only the power usage of one node. Detailed description of the chart in Section 4.2.1. 41

D.1. ARDUPOWER wattmeter installed. Source [BH15] 54

List of Listings

2.1. Usage of the pmlib Plugin for VampirTrace. 14

4.1. MPI implementation to wait for the request to complete (from MPICH implementation). 31
4.2. Test program to show the busy waiting. 32
4.3. The busy function used in Listing 4.2. 33
4.4. Changes to the `MPID_nem_mpich_blocking_recv function` (Lines 1, 4 and 5 are new). 33

B.1. Wrapper program to measure a not instrumented application. 50

List of Tables

A.1. Measured energy consumption. Watts are the average that the system needed during the runtime. 49
A.2. Measured energy consumption with the adjustment presented in Section 4.4. Watts are the average that the system needed during the runtime. 49

C.2. Summary of the problems and solutions: Application software efficiency. Source: [Goe13] 52
C.4. Table C.2 with runtime instead of energy. 53

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang B.Sc. Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

Ort, Datum	Unterschrift