TDK dolgozat

Mészáros Áron Attila, Nagy Gergely

Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Towards green computing in Erlang

TDK thesis

Supervisors:
István Bozó, Melinda Tóth
Assistant lecturer

Authors:
Áron Attila Mészáros, Gergely Nagy
Computer Science BSc
2. year

Budapest

Date of closing: 10 May 2018

Abstract

Energy efficiency in computing was identified as low energy usage of the hardware for a while. However, nowadays, we can talk about energy efficiency in terms of software as well. Therefore, we have to investigate how the different design decisions and programming language constructs affect the energy consumption. Green computing is a relatively new research area, guidelines are required for the software developers in terms of energy efficiency. In our research we are focusing on the functional programming language Erlang. We have investigated the effect of different language constructs, data structures and styles of programming on energy usage. Additionally we present a tool to measure and visualise the consumed energy.

Contents

Chapter 1

Introduction

Environment friendly tools and devices are needed in every area of manufacturing, thus it is crucial to have computing devices with energy usage as low as possible. Therefore, we have to take into account the amount of energy used by certain devices (i.e. a PC) when running a software.

In the field of green computing we are investigating the energy usage of a software, this means the amount of energy used by the hardware when running the software.

Researches have already been presented on energy efficient computing (see Chapter 7), however most of them are focusing on mainstream languages. The goal of our research is to investigate the energy usage of Erlang [1] programs. Erlang is a widely used functional programming language, designed for building concurrent/distributed soft-real time applications. Since Erlang is a functional language and the main building blocks of the language are the functions, we have created a tool (see in Chapter 3) to measure and visualise the energy consumption of Erlang functions. The tool is based on the Intel provided *RAPL* [2] tool and the *rapl-read* [3] program.

In this thesis we present the measurements on some key elements of the language, such as the usage of lists, higher order functions and parallelisation as well (in Chapter 5). We demonstrate our findings on different algorithms.

The ultimate goal of our research is to extend the static source code analysis and transformation framework RefactorErl [4, 5]. We would like to define static analyses to find those source code fragments that are presumably more energy-intensive than other equivalent solutions. We also want to define a set of refactorings that can be applied, either automatically or semi-automatically, to reduce the energy used by the Erlang programs.

In this thesis we are presenting our first findings on the energy usage of Erlang programs, and the latter mentioned goal is the target of our future work.

1.1 Contributions

In this section we enumerate our contributions featured in this thesis.

We created a tool to measure the energy consumption of Erlang programs. This tool includes an Erlang module, so that measurements can be started from Erlang. The tool consists of this Erlang module and the *rapl-read* program to read RAPL register values. We also created a Python program to visualise and process the measured data. This makes it easier to find patterns that consume a lot of energy.

We measured many different implementations for the N-queens and sparse matrix multiplication problems. The different implementations include using different data structures (lists and arrays), eliminating higher order functions and different parallelisation techniques (such as using process pools).

We found that eliminating higher order functions decreases the energy consumption of an Erlang software, when the underlying data structure is a list.

We also found that in our cases using lists was more energy efficient than using arrays as an underlying data structure.

Finally, we found that the number of processes and the number of messages sent greatly influences the energy consumption of a parallel program, and by using process pools we can limit the number of processes and thus reduce energy consumption. We found that our program was the most efficient when the number of total processes was of a similar magnitude than the number of CPU cores.

In June 2018 we will present the findings of this thesis on the 12th Joint Conference on Mathematics and Computer Science (MACS'18) international conference in Cluj-Napoca [6]. A paper has also been submitted to the *Special Issue of Studia Universitatis Babes-Bolyai, series Mathematica, Informatica and Physica* and is currently under review [7].

1.2 Structure of the paper

In Chapter 2 we show the background for our work. We present the Erlang programming language and the concept of green computing. Then we list some of the possible methods to measure the energy consumption of software and give our reasons for choosing *rapl-read.*

In Chapter 3 we present the RAPL tool created by Intel. We show the *rapl-read* program used to read energy consumption values from the registers. At the end of the chapter, we also introduce the Erlang framework we created to make measuring energy consumption accurate and easy. The communication between our Erlang framework and the *rapl-read* program is also shown.

We created a Python program to visualise and process the measured data (in Chapter 4). In this chapter we also present our workflow of measuring and processing the data.

In Chapter 5 we write about our choice of algorithms (N-queens and sparse matrix multiplication) and explain our implementations of these algorithms in great detail. For every algorithm we give a summary of the measurements conducted, and explain our findings. In our measurements and implementations we paid attention to the advantage of eliminating higher order functions and also tried to investigate the effect of different data structures. We also measured different parallel versions of the matrix multiplication problem.

The results summarised for all algorithms and implementations are shown in Chapter 6. We present and explain our findings of measured values regarding the use of higher order functions, different data structures and parallelisation.

There have been many works regarding green computing, some of which are presented in Chapter 7. We present different works from other mainstream languages and even other functional languages. We also show and validate previous results on green computing in Erlang.

Lastly, in Chapter 8 we summarise our work and present the possibilities for future work regarding this subject.

Chapter 2

Background

In this chapter we give a brief summary of the subjects our work is based on, such as the Erlang language and the concept of green computing.

2.1 About Erlang

Erlang is a functional programming language, meaning that it uses a programming paradigm that originates from the theory of mathematical functions. This means that the computation is treated as the evaluation of mathematical functions and as such avoids changing state and mutable data. In theory the result of a function should only depend on the arguments given to it. All programs in functional languages are essentially just the evaluation of a function. This means that in order to measure the energy consumption of Erlang programs we have to be able to measure the energy consumption of computing the result of a function.

When running an Erlang program, it is run in a virtual machine called BEAM (Bogdan/Björn's Erlang Abstract Machine) [8]. Erlang is developed at the Ericsson Computer Science Laboratory. It is used to build massively scalable soft real-time systems with requirements on high availability [9]. Is is used by many telecommunications, banking and e-commerce companies.

Erlang provides lots of modules and functions to use. There are also some data structures built in that can be used, such as lists or arrays. Open Telecom Platform (OTP) is a collection of Erlang libraries and design principles that provides middle-ware to develop these systems. OTP includes its own distributed database, applications to interface towards other languages and many more useful tools [9, 10]. Since Erlang and OTP are closely connected they are often referred to as Erlang/OTP. Currently Erlang/OTP is maintained and managed by the Erlang/OTP unit at Ericsson.

```erlang
-module(hello_world).
-export([hello/1]).

hello(A) when A < 5 -> io:format("Hello World!~n");

hello(A) -> A.

```

Figure 2.1: An Erlang module called `hello_world`.

2.1.1 Examples

Erlang code is divided into modules. Modules consist of attributes and function declaration. Attributes include the name of the module, which functions to export (make visible for outside use) and other compile options. An example module called `hello_world` is shown in Figure 2.1, where the only exported function is `hello` which takes one argument and prints *"Hello World!"* if the argument is less than 5, otherwise returns its argument.

Functions are often identified by their names and number of arguments (for example `hello/1`). This clearly identifies a function within a module, while its name may not be enough to identify it. In this thesis we sometimes refer to functions by simply their names, but other times for the sake of clarity we may use the number of arguments in our notation as well.

As it can be seen in the example in Figure 2.1, Erlang uses a dynamic type system, meaning that the types of variables are not known at the time of compilation, only at runtime. In the example `hello` can be called with any type of variable. The $<$ operator is defined for when the values compared are not the same type. In this case for non numeric values the `hello` function will return its argument. Besides, Erlang is also strongly typed, which means that there is no implicit conversion between different types [11].

The main data structure used in Erlang is the list structure, but many commonly used data structures are also available, for example array, queue, set, record etc. Lists can be build by concatenating lists, enumerating its elements or using list comprehensions. There are also lots of functions working on these data structures. Some of these functions take functions as an argument as well. These are called higher order functions (HOFs). An example of this is the `lists:map/2` function, that applies the given function to all elements in a list. There are two main ways of passing functions as arguments. The first is passing the named function by its name, the other is using anonymous functions (*fun* expressions). Some examples for using higher order functions with lists can be seen on Figure 2.2. More examples using the most common build in higher order functions can be seen on Figure 2.3.

Since there are no loops in functional programming, iteration is made possible by recursion. An example for tail recursion can be seen on Figure 2.2. This example is a tail recursion, meaning that the result is accumulated into a 'variable', and passed further on in the recursion.

As Erlang was developed to build massively scalable soft real-time systems [9], it is important to be able to write concurrent programs easily. Concurrent programs can be written in Erlang by

```erlang
-module(adder).
-compile(export_all). %Export all of the functions

%Increment a number
add_one_element(Elem) -> Elem + 1.

%Incrementing every element in a list, using named function and map
add_one_named(L) -> lists:map(fun add_one_element/1,L).

%Incrementing every element in a list, using fun expression and map
add_one_fun(L) -> lists:map(fun(X) -> X+1 end,L).

%Incrementing every element in a list, using tail recursion
add_one_recursive(L) -> add_one_recursive(L,[]).
add_one_recursive([H|T],Acc) -> add_one_recursive(T,[H+1|Acc]);
add_one_recursive([],Acc) -> lists:reverse(Acc).

```

Figure 2.2: Example of using the higher order function `lists:map/2` with named and anonymous functions. The use of pattern matching and recursion is also shown.

```erlang
-module(example).
-compile(export_all).

%Summation using foldr
sum(L) -> lists:foldr(fun (A,B) -> A+B end, 0, L).

%Return only the positive elements of a list using filter
positives(L) -> lists:filter(fun (A) -> A>0 end, L).

%Decide if all elements are positive using any
all_positive(L) -> lists:all(fun (A) -> A>0 end, L).

%Generate a list and test the previous functions
test() ->
  Lista = [2*X + 3 || X <-lists:seq(-5,5)], %Using a list generatot to
    generate test data
  io:format("~p~n",[sum(Lista)]),
  io:format("~p~n",[positives(Lista)]),
  io:format("~p~n",[all_positive(Lista)]),
  ok.

```

Figure 2.3: Some commonly used higher order functions, and a list generator in to generate test data.

```
par_map(F, Xs) ->
  Me = self(),
  [spawn(fun() -> Me ! F(X) end) || X<-Xs],
  [receive Res -> Res end || _ <- Xs].

```

Figure 2.4: A parallel map implementation in Erlang.

spawning processes to calculate functions. Allocating CPU time and parallelising the processes is handled by the Erlang virtual machine. An example of a parallel map function can be seen on Figure 2.4. This example uses the `self()` command to store the process ID of the parent process and then using the `spawn` command in a list comprehension it spawns a process for each element in the list. These spawned processes calculate the function `F` for argument `X` and send the result back to the parent, who then uses another list comprehension to `receive` the data.

2.2 Green Computing

As global warming is already happening and it is fueled by greenhouse gases such as carbon dioxide, it is becoming more and more important to reduce harmful emissions [12]. Information and communications technology (ICT) systems are responsible for the same amount of CO_2 emissions as global air travel [13]. The ever rising need for computing power and the use of IT services in our lives is consuming lots of energy each year [14]. It seems obvious to try and reduce the energy consumed by these systems, thus reducing harmful greenhouse gas emissons as well.

The high energy usage of ICTs and our changing environment created the need for green computing. The core concept of green computing is to use computers and computing related systems in such a way that we minimise the environmental impact of computing.

Green computing is a broad concept, as it includes building and manufacturing hardware that consumes little energy, but also includes the use of low-energy consuming peripherals [15]. Green computing can also refer to recycling computer parts, using peripherals and parts that are more environmentally friendly to manufacture. For example it is important to consider that organic light-emitting diodes (OLED) are less harmful than regular displays. The size of a display is also an important factor in making a setup better for our world. A 17 inch display has about 40% more surface area than a 14 inch display and because of this it consumes almost 40% more energy [15].

In recent years big companies started to relocate their data centers inside the arctic circle [16] so that they have the natural cold air to cool down the immense heat generated by the servers. Data centers of big companies consume a lot of energy, most of which is wasted as heat and has to be dumped into the environment.

Another thing to consider is that greenness can not only be achieved on the hardware level by manufacturing energy efficient parts or phisically relocating the hardware, but also has to be considered when writing code [17, 18]. This is especially important when writing code that is going to be used on servers or on many devices, as these contribute the most to the carbon footprint of IT applications. As companies begin to see the importance of green computing and manufacturers begin to produce more and more environmentally friendly computer parts, it is software developers turn to utilise these improvements by designing and writing software that is environmentally friendly and has low energy consumption [19].

In order to write greener programs lots of research is needed, so that we know which language elements and language constructs are worth it in terms of energy consumption. These works have to be done on a language by language basis, because every programming language uses different concepts and implementations and what is efficient in one language may be inefficient in an other, simply because of the way it is implemented.

2.3 Tools to measure energy consumption

In this section we present the different tools that can be used to measure energy consumption. Many of these tool utilise RAPL (see Chapter 3), but provide different interfaces to interact with the registers provided by RAPL.

2.3.1 Intel Platform Power Estimation Tool (IPPET)

IPPET is a software tool created by Intel to monitor resource usage of different processes. It uses the machine specific registers provided by RAPL. This software requires Windos 7 or Windows 8+ operating system and because it uses RAPL it also needs Sandybridge or newer CPU. The software provides a graphical interface where under different tabs the user can see the energy consumption and resource usage of individual processes [20].

2.3.2 Intel Power Gadget

Intel Power Gadget is a software based power monitoring tool. It is enabled in Intel Core processors from 2$^{\text{nd}}$ generation up to 7$^{\text{th}}$ generation. It supports Windows and Mac OS X and the application provides a graphical interface to monitor power usage of the system [21].

2.3.3 Turbostat

Turbostat is a Linux tool that uses the RAPL registers to report data about the processor. It can report processor frequency, idle power-state statistics, temperature and power information on x86 processors. It can be used in two ways. The first is to supply a command as a parameter which is then forked and run. After the completion of the command the statistics are printed. The other method does not require a command, instead it displays the statistics every time a given interval has passed [22]. The source code for turbostat is open source.

2.3.4 Performance Application Programming Interface (PAPI)

PAPI provides a tool and interface to see, in near real time, the relation between software performance and processor events. It also provides acces to performance measurement opportunities [23]. PAPI can be difficult to get working, as it requires a kernel patch to work properly [24].

2.3.5 Linux Thermal Daemon

Linux Thermal Daemon is an open source tool created to monitor and control temperatures in tablets and laptops. By using the latest kernel drivers to get energy consumption information, Thermal Daemon is able to help monitor CPU temperatures [25]. Although this tool accesses RAPL registers, it is mainly a temperature controlling tool.

2.3.6 rapl-read

Rapl-read is a program written in C, that uses three different methods to read RAPL values. It obtains the values by reading the MSR values directly, as well as using other Linux specific interfaces to obtain these values [3]. The program measures the power consumption of the CPU for one second, but can be easily modified to measure whatever we want. An example output of the program can be seen on Figure 2.5.

We chose to use rapl-read to get power and energy consumption values, because we needed a lightweight program, that is open source and not too complex so that we can modify it to our specific needs. Tools with advanced graphical user interfaces were not suitable for our purposes. The Linux turbostat tool seemed suitable, but the code was too long to understand and confidently modify it. Thus we chose rapl-read, as it is both lightweight, provides multiple ways to obtain values, can be easily modified and it does everything we needed.

```
Listing paramaters for package #0
  Power units = 0.125W
  CPU Energy units = 0.00006104J
  DRAM Energy units = 0.00006104J
  Time units = 0.00097656s

Sleeping 1 second

Package 0:
  Package energy: 4.343872J
  PowerPlane0 (cores): 2.681274J
  PowerPlane1 (on-core GPU if avail): 0.023621 J
  DRAM: 1.200928J
  PSYS: 0.000000J

```

Figure 2.5: An example output of running `rapl-read` using `msr` mode

Chapter 3

Measuring energy consumption

In this chapter we present the tools, that we used to measure the energy usage of Erlang functions, in more detail. We provide an overview of the workflow of all our different program components working together in order to make measuring energy consumption more convenient.

3.1 RAPL

Running Average Power Limit (RAPL) is a tool created by Intel, as part of their power-capping interface.

3.1.1 Evolution of RAPL

Each computer has a thermal design power (TDP) that represents the maximum amount of heat the cooling system is required to dissipate. As heat takes some time to propagate it is possible for the processor to comsume more power than specified in the TDP, but only for a short amount of time.
Adding cores to a CPU means that in order to satisfy the TDP the CPU will use different maximum frequencies depending on the number of cores. These features were introduced with Intel Turbo boost technology [2]. The effect of Turbo boost can be seen on Figure 3.1.

A Power Control Unit firmware is responsible for making these decisions. It uses internal models and counters to estimate power consumption and before Sandy Bridge architecture these decisions were conservative and lacking exact accuracy. Since Sandy Bridge, CPUs have onboard power meter capability and thus better decisions can be made. Besides this, the CPU also exports power meters and power limits used through a set of Model Specific Registers (MSRs). This interface is the RAPL interface.
RAPL provides counters to get energy and power consumption information. RAPL is not an analog power meter, but an accurate software power model, that estimates usage by using hardware

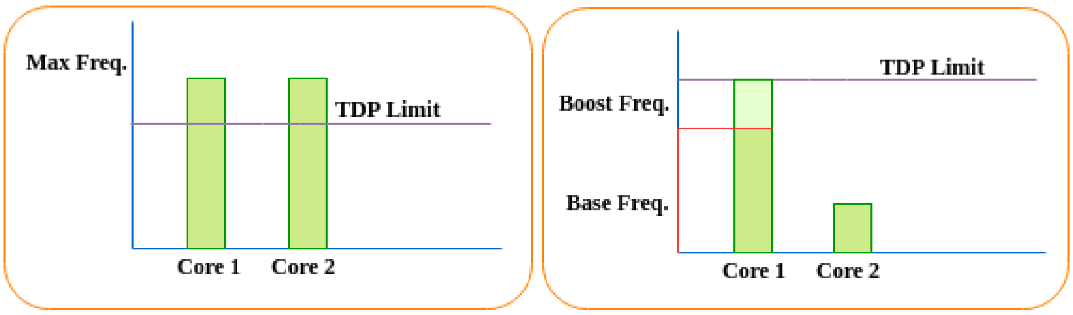


Figure 3.1: Double core CPU maximum frequencies without Turbo boost and with Turbo boost technology
From: https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl

performance counters and I/O models [2].

RAPL is available under Linux operating system, on CPUs that have at least Sandy Bridge or newer architecture, because older units do not have onboard power meter capability.

3.1.2 Data provided by RAPL

In RAPL, platforms are divided into four power planes or domains, in order to get more detailed information on the energy consumption. These domains are:

- PKG - The entire package, including the core and uncore part of the CPU
- PP0 - Only the cores of the CPU
- PP1 - The uncore part of the device, usually the GPU
- DRAM - Main memory

Between the listed domains the following inequality always holds: $\text{PP0} + \text{PP1} \leq \text{PKG}$ and DRAM is independent of the other three [26]. The availability of the PP1 domain is not guaranteed, it is platform specific. DRAM is supported only in the server platform. The RAPL domains on a typical machine are shown on Figure 3.2.
It is important to note that using RAPL there is no way to measure the energy consumption of individual cores, only the consumption of packages can be measured, which is further divided into the core and uncore part of the package. After that no further distinction can be made between different parts of the package.

The numbers read from the MSRs are given in the following units:

- power in watts
- time in seconds

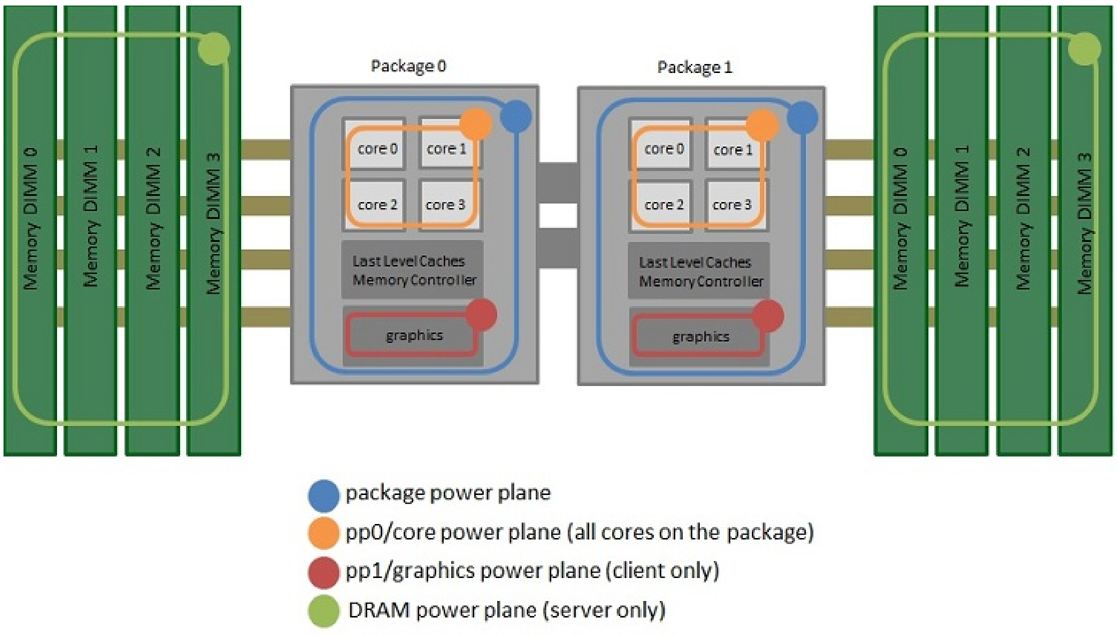


Figure 3.2: RAPL power planes shown on a typical machine
From: https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl

- energy in joules

Scaling factors are provided to each unit to make the information stored meaningful in a finite number of bits [27].

3.1.3 Accuracy of RAPL

Since on most systems the values measured by RAPL are estimated values generated by on-chip energy models, it is necessary to check if the estimated values match with those measured by other instruments. There have already been works to validate the results given by RAPL [28]. Desrochers et al. found that there is no significant difference between the estimated and the accurately measured values. Only slight offsets are possible, especially on idle systems, but in the case of heavy CPU, GPU and DRAM usage the estimates are quite accurate.

3.2 Rapl-read

In order to read the power usage values from the Model Specific Registers we used a program written in C by Weaver called `rapl-read.c` [3]. This program provides code for reading energy consumption values from the MSRs. It also detects the CPU version and reads all applicable domains of RAPL.

```c
static long long read_msr(int fd, int which) {
  uint64_t data;
  if ( pread(fd, &data, sizeof data, which) != sizeof data ) {
    perror("rdmsr:pread");
    exit(127);
  }
  return (long long) data;
}
```

Figure 3.3: C code for reading data from the MSRs

There are three methods to extract power and energy consumption data from RAPL.

- `sysfs` - Reading files from `/sys/class/powercap/intel-rapl/intel-rapl:0` using the sysfs powercap interface. This requires at least Linux 3.13 with no special permissions.
- `perf_event` - Using the `perf_event` interface. This requires at least Linux 3.14, and root access or the `/proc/sys/kernel/perf_event_paranoid` value to be less than 1.
- `msr` - Reading data directly from the MSRs under `/dev/msr`. This requires root privileges and may require running the `modprobe msr` command before being able to read the data.

`rapl-read.c` provides methods for reading data using all three available methods. These methods originally read the values of the registers in the beginning then waited some time and read the values after that time has passed. Finally the energy consumption was calculated by subtracting these values from each other.
In order to be able to measure the energy consumption of Erlang functions we had to slightly modify these methods. We split all three of them in two. The first one reads and stores the values of the registers before we run the Erlang function. The second part is called after the Erlang function finished and it reads the values of the registers again. After that from the before and after values we can calculate the total energy consumed by the Erlang function.

3.2.1 Overhead of measuring with Rapl-read

Since rapl-read measures the total energy consumption of all processes running on a machine, we only wanted to keep the necessary processes running during our measurements. We only had the operating system and the rapl-read process running whenever we measured, but this still provides an overhead compared to the energy consumed by the Erlang process.

To gather information on the magnitude of this overhead we ran one of our measurments and after that ran a process that simply slept for the same time as the other process was running. This way we were able to measure how much part of our measured values is the overhead of the measurement. These results can be seen on Figure 3.4. In most of these cases the overhead amounts to around 25-30% of the total energy consumed by the algorithm.

Even though the overhead is present and takes up some part of our measurements, we decided to keep all measured values as is, containing the overhead. This is because running the operating system in the background is necessary to run an Erlang program and thus we consider it as a necessary part of the energy consumed by the program.

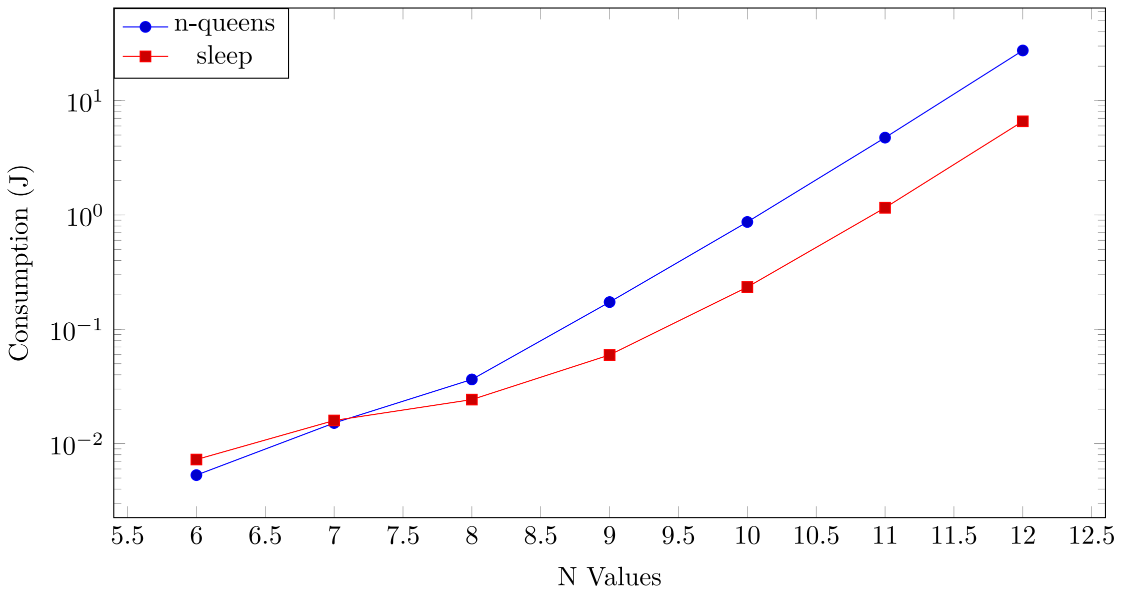


Figure 3.4: Energy consumption of an Erlang program solving the N-queens problem and another one sleeping for the same time. Note that the measured values are in logrithmic scale.

3.3 Erlang framework

We wanted to make it easy to measure energy consumption within Erlang, so we created an Erlang module that makes it possible to execute and measure functions. This module provides a function called `measure` (seen on Figure 3.5) that can be called with the name and path of the measuring program (`rapl-read.c`), a tuple describing the function and arguments we want to measure with an optional InputSize parameter, that describes the arguments. This optional parameter was needed for us to be able to easily graph the results according to the size of the inputs. Additionaly the measure function takes the number of measurements to make as parameter N, and the names of the output files.

```erlang
measure(Program, {Module, Function, Arguments, InputSize},
  N, ResultOutput, AvgOutput, LogFile)
  %InputSize is optional
```

Figure 3.5: The signature of the `measure/6` function

3.3.1 Communication

It is important for the accuracy of measurement to get the readings as close to the beginning and end of a function as possible. Because of this, we needed the Erlang program (`energy_consumption.erl`) and the `rapl-read.c` program to communicate. The code for communicating can be seen on Figure 3.6.

```erlang
%Opening ports
process_flag(trap_exit, true),
Port = open_port({spawn, Prog}, [{packet, 2}])

%Receiving and sending messages
receive
  {Port, {data, [1]}} -> %Receive expected data
    Port ! {self(), {command, [1]}}; %Send data
  X -> %Receive any other data
    io:format("~p~n", [X])
end.

```

(a) Communication on Erlang side

```c
read_exact(byte *buf, int len){ //read len bytes to buf
  int i, got=0;
  do {
  if ((i = read(0, buf+got, len-got)) <= 0)
    return(i);
  got += i;
  } while (got<len);
  return(len);
}

write_exact(byte *buf, int len){ //write len bytes from buf
  int i, wrote = 0;
  do {
  if ((i = write(1, buf+wrote, len-wrote)) <= 0)
    return (i);
  wrote += i;
  } while (wrote<len);
  return (len);
}

```

(b) Communication on C side

Figure 3.6: Code snippets that are used to communicate between Erlang and C

First, upon calling the `measure` function, the Erlang program spawns the program that was given in its first parameter. Spawning is done by opening a port using the `open_port` function of Erlang and then spawning the C program. After that as soon as `rapl-read.c` is ready it sends a signal back to Erlang. Now both programs are ready to start measuring. The Erlang code sends a signal to the C code to gather the before values. As soon as this is done, the C code sends a signal back indicating that it is done gathering the values. After that Erlang starts executing the function that needs to be measured. When its done it signals the C code to gather the after values and send them back. When it is done gathering the after values and calculating the energy consumed, the C program sends back the data to Erlang, where it is processed and stored. After the data is sent back the C program terminates. This communication process can be seen on Figure 3.7.

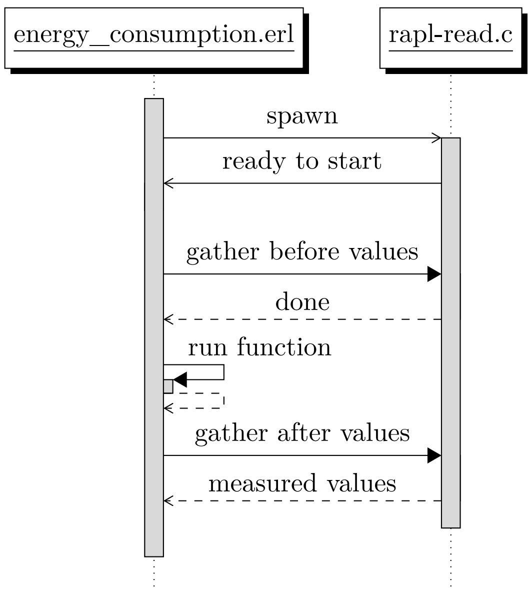


Figure 3.7: Sequence diagram of `energy_consumption.erl` and `rapl-read.c` communicating throughout one measurement

Sending the measured data to Erlang is done by first sending the number of methods used to measure (such as `msr`, `sysfs` or `perf_event`). Then the names of these methods are also sent. For each method the number and names of domains used are sent next. Then for each domain the measure value is sent as a floating point number.

3.3.2 Storing data

When the measured values have been received by the Erlang component from `rapl-read.c`, it saves it using a `dets` (disk-based term storage) table. Using `dets` makes it easy to use the data later in Erlang by simply reading it back (there is no need to parse text, because the terms themselves are stored). The `dets` file storing the measured data is opened in `duplicate_bag` mode. This means that elements with the same key can be in the `dets` multiple times, even with the same values. We used this mode, because we want to store multiple elements with the same key. Also, the measured values may be the same sometimes for the same key, so we need the storage to handle duplicates.

The results are stored as tuples in the following format:

$$\{\{\{Module, Function, InputSize\}, Method, Domain\}, Value\}$$

where $Value$ is the measured value, $Method$ is one of the three modes to access RAPL, and $Domain$ is one of the four available RAPL domains. $Module$ and $Function$ are the module and name of the measured function, while $InputSize$ is a value provided by the user describing the size of the test arguments. If no such value is provided we take the head of the argument list and use it as $InputSize$. This is useful when the argument is a single number. In this case the key of this tuple is $\{\{Module, Function, InputSize\}, Method, Domain\}$, while the value stored is $Value$.

3.3.3 Methodology

We used the following methodology to measure the energy consumption. Each time all three methods and all available domains are measured and sent to the Erlang program. In addition to this we also measure the runtime of the function and store it with the energy usage values. Runtime is measured using the `system_time` function of Erlang.

This process is repeated N times, where N is the parameter given by the user to the `measure` function. After this the Erlang program reads back all the data and for each method-domain pair calculates the average energy consumption, disregarding the lowest and highest values. This average is then inserted to the `dets` table and is also printed to a text file, so that we can easily plot and analize the data.

Chapter 4

Processing and visualising data

The measured values were also exported to text files so they could be easily processed. These files contain one result in each line in the following structure:

$Module \quad Function \quad InputSize \quad Method \quad Domain \quad Value$

Besides the module, function name and input size, the output contains the method used in the measurement, the referenced domain and the measured value.

We use the same framework for measuring the run-time of the functions. The output in this case includes the same fields, but when measuring the execution time, the method and domain names are irrelevant, thus to preserve the same structure, we assigned the method 'time' and domain 'time' to these lines.

To visualise the measured data, it would have taken a lot of time to analyse manually the data files (for example using a spreadsheet). We have decided to process the raw data with a Python script. Besides, this gives us more possibilities to make some calculations with the data and compare the functions according to different aspects. Firstly, we wanted to calculate the Pearson correlation coefficient between the time and energy usage of each function, which has been achieved with `scipy.stats.pearsonr` [29]. Secondly, we have calculated the power usage of the functions, which is defined as the ratio of energy consumption and elapsed time (Joule/sec). This was necessary because correlation does not provide any information about the relationship between different functions, but with power usage values we could compare them to each other. Power usage has been calculated separately for each input size to see if it is changing and was also calculated in total, aggregating the energy consumed and time used for all input sizes.

The visualising part of the script is managed by *matplotlib* [30].

Before this could happen, the script had to extract and collect the values by the different methods and domains. To make it easy, we use a class called `Measurement`, which has four arguments for the different domains (ram, core, uncore, package) and one for time. We use a dictionary to store `Measurement` objects, one key in this dictionary looks like the following: `(Module,Function,InputSize,Method)`. The script reads the file line by line, and constructs the key: if the same key already exists, it stores the read value in the correct attribute, otherwise creates a new item with the key and then the value could be stored likewise. By the end of the file, all attributes are specified.

It is not difficult to visualise the data from here. On one graph, we would like to draw values of different functions and do this with only one domain from a method. This can be specified with command line arguments and flags. For each specification we call a function which takes these conditions as arguments and creates the figure. In this function we use another dictionary for the functions which are going to be displayed. Because a function can be uniquely identified by the name of the module and the name of the function we use them as keys. The script iterates through the dictionary of `Measurement` objects, and selects items which meet the criteria. Then it creates the key, and gives (InputSize,Value) to it as a value. When this iteration is over, we create the figure: the Y-axis shows the energy consumption in Joules and the X-axis shows the different input size values. So the script iterates through the functions and gives the values to matplotlib in the right form. A figure drawn this way is shown on Figure 4.1. Matplotlib provides features such as zooming into certain parts of the graph, which is useful when the differences are hardly visible between functions. Besides, there is an option to save the plot as image. However, we did not use this feature, because although it is faster it has its limitations. To avoid these, the script provides an alternative saving option into `.tex` files. This happens very similarly to the former algorithm, but in the end it creates a data file containing the results, and a `.tex` file that imports these values. We can design the figures freely with modifing this `.tex` file and easily include the plots in other LaTeX documents.

To summarise, the script that processes the raw data has the following stages:

1. Reading the file: Extracting the values separately by the different methods and domains. Creating `Measurement` objects and storing them in a dictionary.
2. Calling the drawing function for each specification.
3. Selecting the right values and drawing the figures using matplotlib.
4. Optionally, making calculations from the measured values and exporting them into a file.
5. Optionally, exporting the required values to another data file and generating a LaTeX file from it that contains only the diagram.

As we have mentioned earlier, the figures are fully customisable to help analyse the data more precisely. We have used command line arguments to give the exact specifications of the diagrams. The following flags are available:

- --files: The script can take multiple files, all the data in these files is processed. This gives flexibility to visualise different functions. For example, we stored all the different functions in distinct files, so we could compare them as we wanted.
- --methods: Specify which methods to display. It can take the three methods and time, and also several methods at once. In this case all of them will be drawn.

- --domains: Specify which domain to display.
- --output: Optional: the name of the output data file and .tex file.
- --logscale: This flag changes the Y-axis to a logarithmic scale.

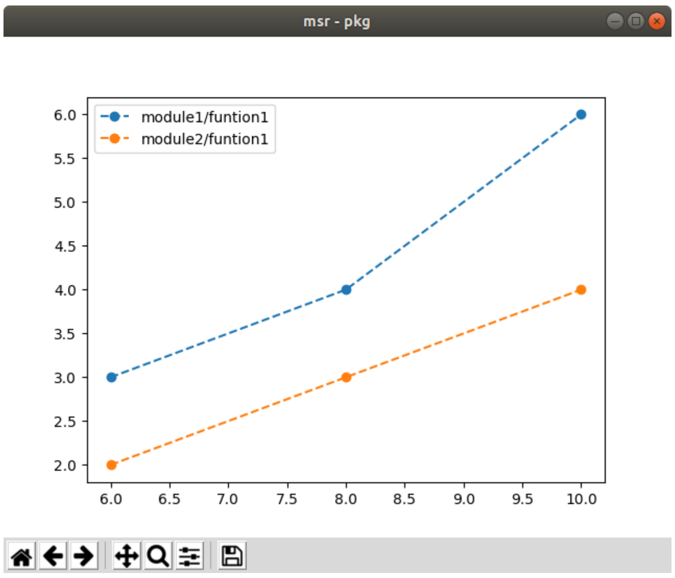


Figure 4.1: Figure drawn by matplotlib

The whole process from the measurement to the visualisation is shown on Figure 4.2.

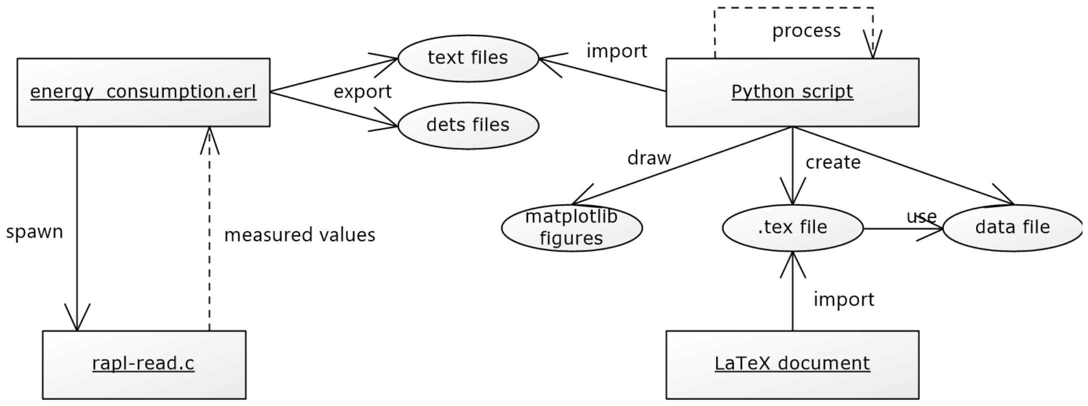


Figure 4.2: Process from measurement to visualisation

Chapter 5

Measured algorithms

For our measurements we wanted to select problems and algorithms on which most of the things we wanted to test could be measured easily.

We wanted to focus on three main aspects of the algorithms. These are the following:

- Using different data structures:
  - Lists
  - Extendible arrays
  - Fix-sized arrays
- Using or avoiding higher order functions (HOFs)
- Parallel or sequential implementations

Firstly, our algorithms of choice had to be implementable using multiple different data structures, mostly lists and arrays. We also wanted to measure the two types of arrays available in Erlang, the fix-sized array data structure and the extendible array.

The second main aspect was the use of higher order functions (HOFs). The measured algorithms had to have higher order functions, so that we can eliminate them, and measure the difference in energy consumption.

The third thing we wanted to inspect was the effect of parallelisation. We wanted to measure the difference between the sequential and parallel implementations, and also wanted to test different parallelisation techniques, for example using process pools or brute force parallelising our algorithms. Therefore our test problems had to be easily parallelisable.

After considering these things we chose two problems, that we thought were suitable to be our first test problems. The two problems were the following:

- N-queens
- Sparse matrix multiplication

More details about the problems and our implementations can be seen later on in this chapter (in Sections 5.1 and 5.2). We chose these problems, because they fit in the categories mentioned before, and the two problems are not really similar, so we can measure two fundamentally different problems.

We also paid attention to the runtime of each implementation, in order to gain a better understanding of the effects of language constructs on energy consumption. The measurements (where not otherwise specified) were made on a system with Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz and 8 GB of DDR3 RAM @ 1600MHz, using Ubuntu 16.04 LTS. All plotted data was measured using the MSR method of rapl-read and the sum of the PKG and DRAM domains is shown (where not otherwise specified).

5.1 N-queens

The task is to place N queens on an N×N chessboard, so that no two queens can attack each other. In other words, no two queens can be in the same row, column or diagonal. We need to find all possible ways to do this. All possible solutions for the 4×4 case are shown on Figure 5.1.

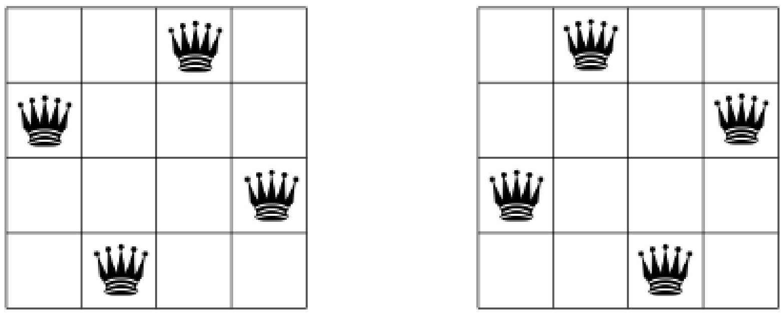


Figure 5.1: All possible solutions to the 4-queens problem
From: https://acrogenesis.com/or-tools/documentation/user_manual/manual/introduction/4queens.html

5.1.1 Solutions

For this problem we measured the following five implementations:

Lists with higher order functions (`queens_lists`): This was the first version we implemented, later versions are a modified variation of this implementation. To solve the problem, we used a simple recursive strategy, that starts enumerating all possibilities of placing the queens on the board. We have a list of all possible solutions. First we add an element to the solution

```erlang
attacks({RowA, ColA}, {RowB, ColB}) ->
  RowA == RowB orelse ColA == ColB
  orelse abs(RowA - RowB) == abs(ColA - ColB).

legal_list(Queen, Queens) ->
  lists:all(fun(Q) -> not (attacks(Queen, Q)) end, Queens).

solve_list(N, Row, Queens) when Row > N -> [Queens];
solve_list(N, Row, Queens) ->
  lists:flatmap(
  fun(Qs) -> solve_list(N,Row+1,Qs) end,
  [[{Col,Row}|Queens] || Col <- lists:seq(1,N),legal_list({Col,Row},Queens)]
  ).

queens_list(N) when N > 0 -> solve_list(N,1,[]).
```

Figure 5.2: Solution for the N-queens problem using lists and higher order functions

list for a queen on every possible position in row one. Then as we try to find all solutions, we add an element to the list of possible solutions for every possible placement of a new queen on the next row. This is done by recursively traversing the list and for each element of the list we find all possible location for a queen on the next row. If no such position exists, the element is removed from the list. This way when we get to the N$^{\text{th}}$ row, the solution list will contain all solutions to the problem. This strategy utilizes that in order to solve the problem we will need to place a queen in each row of the board.

Our Erlang implementation (see Figure 5.2) of this problem consists of five functions. Queens are represented by `{Row,Col}` tuples, describing their position on the board. The `attacks/2` function decides if two queens described by tuples `{RowA, ColA}` and `{RowB, ColB}` attack each other.
The `legal_list/2` function decides if a queen can be appended to an existing legal list of queens. This is decided using the higher order function `list:all` to find if for all queens in the list the new queen attacks them or not.
`solve_list/3` is the main function of this solution. It takes three arguments: the size of the board, the current row number (where we have to place the queens now), and the current list of possible solutions. When the current row number is greater than the board size, we are done and return the list of solutions. Otherwise, we use the higher order function `lists:flatmap` to recursively call `solve_list`. The mapping is done on a list consisting of lists of queens, where every element is the current solution list and a new legal queen appended to it. Since we use `flatmap` and not `map` the result will not be a many lists deep, but instead a simple list of lists. `flatmap` concatenates all lists that are the result of the mapping function.
Finally, the last function, `queens_list`, is simply an adapter to make it easier to call the solver. It takes the board size as an argument, and if this is a legal number (greater than 0) then it calls the solver to start from the first row and with an empty solution list.

Lists without higher order functions (`queens_nohof`): This implementation is almost the same as the previous one with higher order functions. The only difference is that instead of using the higher order functions `lists:flatmap` and `lists:all` we implemented our own version of

```erlang
all_nohof(_,[]) -> true;
all_nohof(Queen,[Q|Queens]) ->
  G = attacks(Queen,Q),
  if G -> false;
  true -> all_nohof(Queen,Queens)
end.

flatmap_nohof(Queens,N,Row) -> flatmap_nohof(Queens,[],N,Row).
flatmap_nohof([],R,_,_) -> R;
flatmap_nohof([H|T],R,N,Row) ->
  flatmap_nohof(T,solve_nohof(N,Row+1,H) ++ R,N,Row).

solve_nohof(N, Row, Queens) when Row > N -> [Queens];
solve_nohof(N, Row, Queens) ->
  flatmap_nohof(
    [ [{Col,Row} | Queens] || Col <- lists:seq(1,N),all_nohof({Col,Row},
   Queens) ],N,Row
  ).

queens_nohof(N) when N > 0 -> solve_nohof(N,1,[]).
```

Figure 5.3: Our own functions to replace the higher order functions used and the main function of this version

them, that take no functions as an argument. Instead, since we know exactly our use case of the functions, we implemented them specifically to behave as the higher order functions would behave given the functions of the previous version as arguments.

The Erlang code for the implementation can be seen on Figure 5.3. The `all_nohof/2` function uses the same `attack` function as the previous version. Calling `all_nohof(Queen, Queens)` does what the `lists:all(fun(Q) -> not (attacks(Queen, Q)) end, Queens)` function call would do in itself so we do not need to have the separate function `legal_list` anymore. Instead we can directly call `all_nohof` from `solve_nohof`.
The `flatmap_nohof/3` function makes it easier to call `flatmap_nohof/4` since when calling the 3 argument version we do not have to provide an empty list. The `flatmap_nohof/4` function does exactly what the higher order function `lists:flatmap` would do given the function argument in the previous version. This implementation is a tail recursive implementation that accumulates the resulting list in one of its arguments, and when there are no more elements to process returns it.
The `solve_nohof` function is almost identical to the `solve_list` in the previous version, but instead uses the non-higher oreder functions described above.

Extendible arrays (`queens_array`): In this version we use the same algorithm as in all the previous ones, but the underlying data structure is changed from lists to arrays. The arrays used are extendible in size, meaning that they are automatically resized by Erlang if needed. This is achieved by using `array:new()` to create new arrays.

Since there is no `flatmap` or `all` function implemented for arrays in Erlang, we wrote our own

legal_array/3 and flatmap_array/3 functions, that are identical to the previously shown non-higher order functions on lists, but this time they use arrays. To implement flatmap_array/3 we needed a way to concatenate arrays, since there is no operator to do that, we created the function concat_to_array to do it.
To access an element of an array we used the array:get function, and to set an element we used the array:set function.
Another thing we needed was to get the first element of an array that was not set. This is exactly what the array:size function does.
Since there are no list comprehensions that could be used on arrays, we needed a way to generate the array of all possible placements of queens. We needed an array of arrays to replace the list of lists in the previous versions. For this purpose we created the get_arrays/5 function, that generates an array of all possible solutions, given the current array of possible solutions, the size of the board, the current row and column numbers (the column should always be 1 when calling the function from outside), and an accumulator array, where the results are stored.
Some of the functions, that recursively iterate over an array store in one of their arguments the current index (IQ). This argument should always be set to 0 when calling the function from outside, since it only uses it during the recursion.
The Erlang code for this version is shown on Figure 5.4.

Fix-sized arrays (queens_array_fix): This version is almost the same as the one with extendible arrays, but instead we use fix-sized arrays where it is possible. This essentially means that in the beginning, when calling solve_array_fix from queens_array_fix, we create an array with size N, that we use in the map function, and copies of it are placed in the array of possible solutions.

To create fix-sized arrays the array:new(Size) function can be used, instead of array:new(), wich creates an extendible array.

Parallel version (queens_par): The final version we implemented was a basic parallel implementation of the version using lists and higher order functions. The algorithm is the same as before, but the representation has changed compared to other list versions. Queens are identified by their column number and their index in the list corresponds to the row they are in. This means that now we only have a list of integers, not a list of tuples.

Instead of lists:flatmap now we use a parallel map implementation together with lists:append in order to concatenate all lists given by our parallel map.

The parallel map (par_map/2) stores its own process ID given by the self() built in function. Then for every element in the list given to it, it spawns a process that executes the function given to the map in its arguments and sends the result back to the parent. After that, using a list generator it receives the results sent back by the spawned processes. This list will be the result of the map.

Apart from the parallel map and slight change of data structure, this version is similar to the first version that used lists.
For the implementation of all of these functions see Figure 5.5.

```
queens_array(N) when N > 0 ->
solve_array(N,1,array:new()).

legal_array(Queen,IQ,Q) ->
Size_Q = array:size(Q),
if  Size_Q == IQ -> true;
true ->
  B = attacks(Queen,array:get(IQ,Q)),
  if B -> false;
    true -> legal_array(Queen,IQ+1,Q)
  end
end.

get_arrays(_,N,_,Col,R) when Col > N -> R;
get_arrays(Q,N,Row,Col,R) ->
B = legal_array({Col,Row},0,Q),
if B  -> get_arrays(Q,N,Row,Col+1,array:set(array:size(R),array:set(array:
    size(Q),{Col,Row},Q),R));
   true -> get_arrays(Q,N,Row,Col+1,R)
end.

solve_array(N, Row, Queens) when Row > N -> array:set(0,Queens,array:new());
solve_array(N, Row, Queens) ->
flatmap_array(
  get_arrays(Queens,N,Row,1,array:new()),N,Row
).

flatmap_array(Queens,N,Row) -> flatmap_array(0,Queens,array:new(),N,Row).
flatmap_array(IQ,Q,R,N,Row) ->
Size_Q = array:size(Q),
if  Size_Q == IQ -> R;
true ->
  flatmap_array(IQ+1,Q,concat_to_array(0, solve_array(N,Row+1,array:get(IQ,Q)
    ),R),N,Row)
end.

concat_to_array(IA,A,R) ->
Size_A = array:size(A),
if  Size_A == IA -> R;
true ->
  concat_to_array(IA+1, A, array:set(array:size(R),array:get(IA,A),R))
end.
```

Figure 5.4: Solution to the N-queens problem using extendible arrays

```
queens_par(N) ->
  solutions_par(N, N, []).

solutions_par(0, _Rows, Xs) ->
  [Xs];
solutions_par(N, Rows, Xs) ->
  Next = [[Row | Xs] || Row <- lists:seq(1, Rows), legal_par(Row, Xs)],
  lists:append(par_map(fun(Xs2) -> solutions_par(N - 1, Rows, Xs2) end, Next)
    ).

legal_par(Row, Queens) ->
  not lists:member(true, [attacks_par(Row, Queens, I) || I <- lists:seq(1,
    length(Queens))]).

attacks_par(Q, Queens, I) ->
  B = lists:nth(I, Queens),
  Q == B orelse abs(Q - B) == I.

par_map(F, Xs) ->
  Me = self(),
  [spawn(fun() -> Me ! F(X) end) || X<-Xs],
  [receive Res -> Res end || _ <- Xs].
```

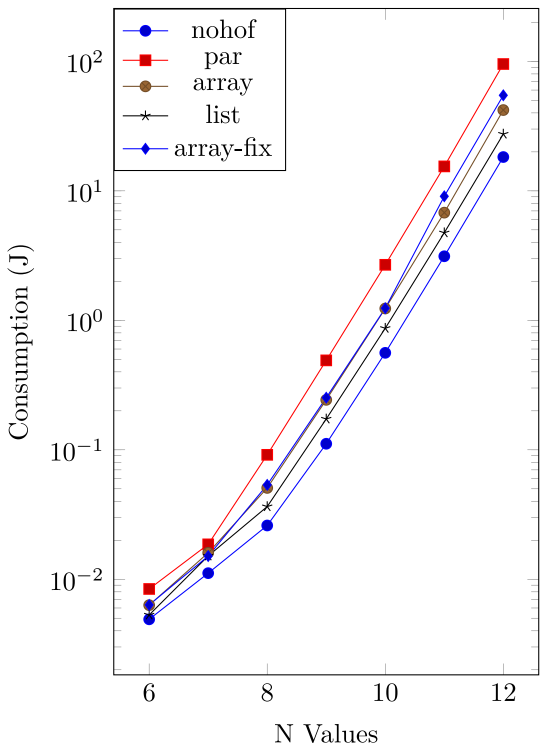
Figure 5.5: Parallel N-queens solution

5.1.2 Results

Each implementation was measured for N values from 6 through 12.

The energy consumption and runtime of these implementations can be seen on Figure 5.6.

For all non-parallel implementations we can see that the more time it takes for a program to produce a result, the more energy it consumes. This relationship between runtime an energy consumption is in line with our expectations. The correlation between the runtime and energy consumption of each implementation is over 0.999, furthermore even in the case of the parallel version, the correlation between time and energy consumption is over 0.999. These correlations do not provide us any useful information, because these only contain data for the energy usage and runtime of a single implementation, but nothing about the relationship between the different implementations.
To find a better way to compare the energy consumption and runtime of our functions, we investigated the energy consumed over a unit of time (the power used by a function). This power usage was calculated for each input separately, but also for each implementation we calculated the total power by dividing the sum of the energy consumed for all inputs by the sum of runtime for all inputs. The power usage of the sequential versions can be seen on Figure 5.7. From these values we can see that the version without higher order functions needs the least amount of power. It is interesting that the fixed array version uses less power than lists, but since it takes a long time to complete in the end it uses more energy. The power usage of the parallel version was 14.700 W, which is significantly more than that of any other implementation. This shows that using all cores of a CPU needs more power than simply running a sequential program.



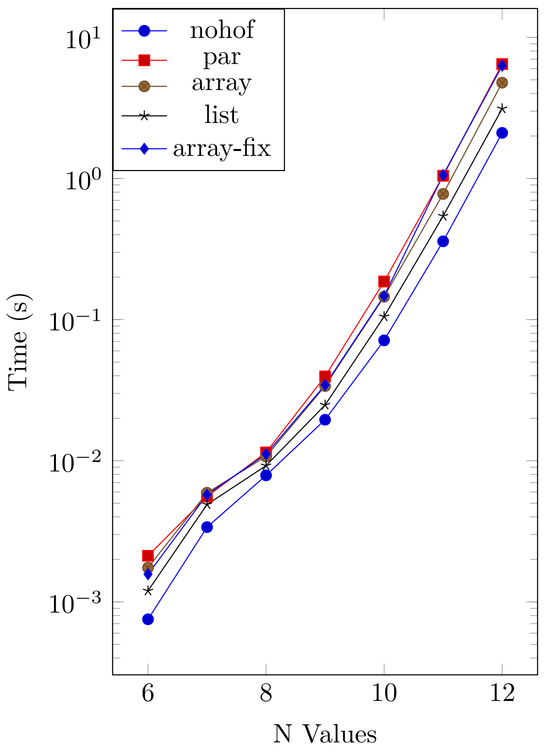


Figure 5.6: Energy consumption and run-time of all N-queens implementations. Note that the y-axis is in logarithmic scale, in order to make it easier to see the relation between different functions better.

In the parallel case, even though the parallel version is not always the slowest (the `array_fix` version is slower for some inputs), it clearly consumes the most energy for all test cases. The reason for this may be that in this parallel version we just replaced the sequential `map/2` with `par_map/2`. This results in lots of processes, since on each level of recursion many more possibilities arise. Our `par_map` spawns one process for each of these possibilities and later on the spawned process repeats this. This recursive spawning may result in more than a 1000 processes for the N = 9 case and even more than 8000 for the N = 10 case. The number of processes inceases drastically as the input increases. For N = 12 the total number of processes needed is more than 500 000. These numbers were determined by using the `+P` flag when starting the Erlang shell to limit the total number of processes. Then we determined the number of processes needed to be able to run the functions with the given arguments.

Spawning and scheduling this many processes may consume a lot of energy, upon inspecting RAM usage, we found that the parallel version uses significantly more RAM than the sequential versions. The DRAM energy usage of all versions can be seen on Figure 5.8. If we look at the quotient of the energy consumption of the parallel and the sequential (list) version, we can see that for the larger inputs the DRAM usage of the parallel version is much higher in proportion to the sequential version, while the total energy consumed is still higher for the parallel version, but it is not higher by as much as the DRAM usage is. For the smaller input sizes the difference is not quite that big, since there are not that many processes at that level in the parallel version.

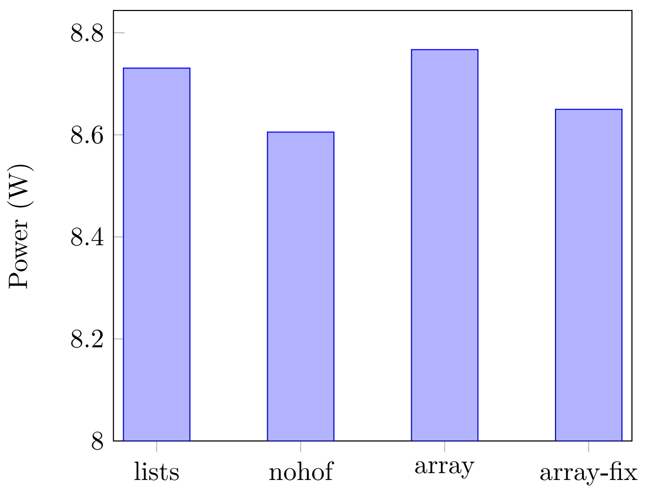


Figure 5.7: The average power consumption of all (except parallel) N-queens solutions. Note that the y axis begins at 8 watts.

For the data and quotients see Table 5.1.

The reason for this high DRAM energy usage may be that in the case of Erlang all data that is sent between processes is copied [31]. Because the items being mapped are lists (meaning that spawned processes use the lists in their arguments), lots of data is copied each time `par_map/2` is called, which is slow, usese lots of RAM and thus consumes lots of energy.

From the graphs on Figure 5.6 it is clear that both implementations using arrays were worse than the ones using lists. For smaller inputs both array versions performed about the same, but around the N = 10 input their graphs separate, making the fix-sized array worse than the extendible array for larger inputs. One reason for this difference may be that using fix-sized arrays may use more RAM than needed, because more memory has been allocated to it than for the one where we simply insert new elements and extend the size if needed.
It is possible that a better, more efficient solution to the N-queens problem exists using arrays in Erlang, but we wanted to measure as similar algorithms as possible, to truly see the effect of the data structure on energy consumption.

For our algorithms, the most efficient versions were the two implementations that use lists as their underlying data structure. Out of the two very similar versions the one where the higher order functions were eliminated performed consistently better than the one using `lists:flat_map` and `lists:all`.
There may be many reasons for this result. Higher order functions may perform worse, because of the extra function call when calling the function given in their argument. Another reason may be the cost of sending functions in an argument. One possibility is that these higher order functions performed worse simply because they are implemented in another module and not at the same place where they are called from.
It is clear that eliminating higher order functions and implementing them at our module as non-higher order functions was beneficial for the energy consumption of our program. Besides, because of the strong correlation between runtime and energy consumption, this modification

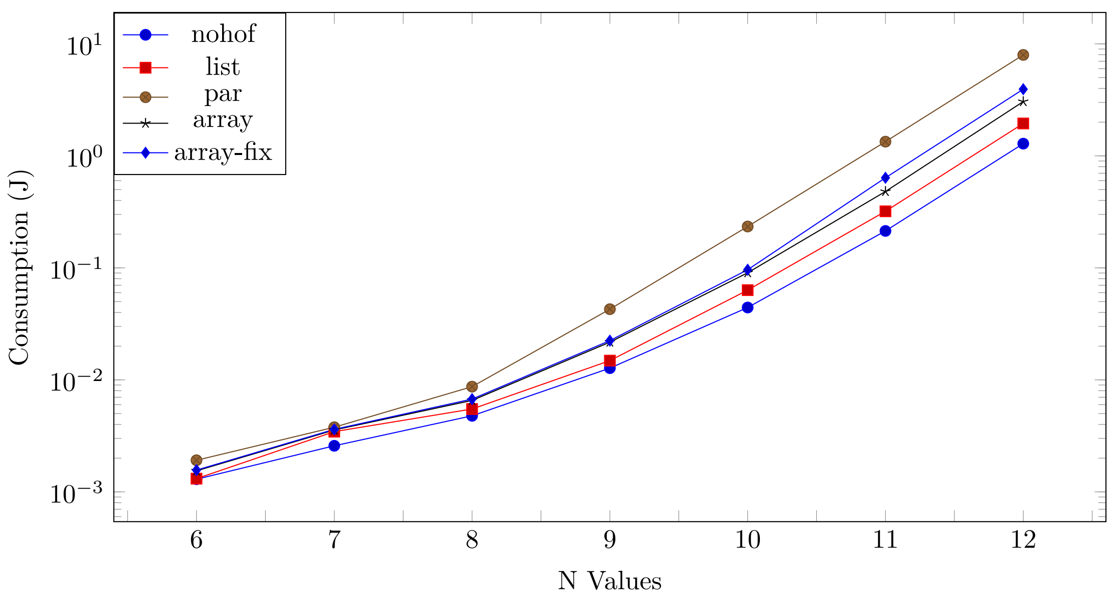


Figure 5.8: Energy consumption of DRAM in all N-queens versions

also made the program run faster. The `nohof` version was the most energy efficient and took the least amount of time to complete.

In conclusion to the N-queens problem and solutions, we saw that there is a strong correlation between the energy consumption and runtime, even in the parallel case.
Parallelising the program can be dangerous and we have to pay attention to the number of processes created, since it can significantly add to DRAM energy consumption and also runtime, thus making this version the worst in terms of energy consumed.
As for the underlying data structure, array seem to be less efficient than lists. The two types of arrays started out performing equally, but for larger inputs it became clear that extendible arrays consume less energy for this problem.
Finally, eliminating and reimplementing higher order functions as simple functions improved energy consumption, making it the best version in this case.

N values Function	6	7	8	9	10	11	12
`queens_par`	0.008	0.019	0.091	0.490	2.683	15.443	95.371
`queens_lists`	0.005	0.015	0.036	0.173	0.868	4.744	27.440
Quotient	1.587	1.229	2.510	2.835	3.091	3.255	3.476

Total energy consumption (J) and quotient of `queens_par/queens_lists`

N values Function	6	7	8	9	10	11	12
`queens_par`	0.002	0.004	0.009	0.043	0.234	1.339	7.968
`queens_lists`	0.001	0.003	0.005	0.015	0.063	0.312	1.947
Quotient	1.459	1.095	1.583	2.879	3.700	4.187	4.092

DRAM energy consumption (J) and quotient of `queens_par/queens_lists`

Table 5.1: DRAM and total energy consumption of the parallel and sequential (lists) N-queens solutions. The values are rounded, but the quotients are calculated using the exact values.

5.2 Sparse matrix multiplication

The task is to multiply two matrices, whose elements are mostly zeros. In general the problem means the multiplication of compatible rectangular matrices, but we only measured out implementations with square matrices. An example of two 4×4 sparse matrices multiplied can be seen on Figure 5.9

The density of a matrix means the ratio of the non-zero elements in it to the number of total elements in the matrix. For example a 10×10 matrix with 10% density has 100 elements in total, of which 10 are non-zero, the other 90 are zeros. There is no exact value of matrix density, under which matrices are considered sparse. We measured matrices with four different density values, which were 1%, 10%, 30% and lastly we also inspected how our algorithms behave given a non-sparse matrix with 100% density.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 5 & 3 & 0 & 7 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 15 & 0 & 16 & 0 \\ 15 & 9 & 0 & 21 \\ 0 & 0 & 12 & 0 \end{bmatrix}$$

Figure 5.9: 4×4 example of sparse matrix multiplication

5.2.1 Solutions

The basic algorithm that all our versions follow is based on the definition of matrix multiplication. We reduce the problem to a series of matrix-vector multiplications, where the first matrix is multiplied by each column of the other matrix. Then these matrix-vector multiplications are solved by further reducing them to simple row vector-column vector multiplications, which can be solved by recursively iterating over both vectors and summing the product of their elements. One such vector-vector multiplication yields one element of the resulting matrix.

Lists (`mxm_list`): In this implementation we represented the matrices using lists and tuples inside the lists. To make multiplication easier, the left and right matrices are represented in slightly different ways. The representation of the first (left) matrix uses a list, whose elements are tuples of the `Row` index and another list. This other list also contains tuples, whose first element is the `Column` index and the second is the value of the sparse matrix at the `(Row,Column)` position. The other matrix is also represented this way, but in this case the outer list contains the `Column` index and the inner list contains the `Row` index. This representation makes it easy to get the vectors that need to be multiplied during the matrix multiplication process. It is important, that only the non-zero elements are stored in these lists and the elements of a list are ordered by `Row` and `Column` numbers. The product of two martices is represented the same way as the second matrix, with the `Column` index being in the outside list. The two multiplied matrices and the result seen on Figure 5.9 are represented by the following lists:
`[{2,[{1,5},{2,8}]},{3,[{3,3}]},{4,[{2,6}]}]` and
`[{1,[{1,3},{3,5}]},{2,[{2,3},{4,1}]},{3,[{2,2}]},{4,[{3,7}]}]` and the result is
`[{1,[{2,15},{3,15}]},{2,[{2,24},{4,18}]},{3,[{2,16},{4,12}]},{4,[{3,21}]}]`

This version consists of 3 functions. The first function is `vxv_list`, which calls the tail recursive vector multiplication funtion `vxv_acc_list`. This recursive function iterates over a row and a column vector, always stepping in the list with the smaller index in its head element. When the two indices are equal we found elements to multiply and add the product of them to the accumulator.
Another function is `mxv_list`, that takes a matrix represented by its rows and multiplies it with a column vector resulting in a colum vector of the result matrix. This multiplication is done by calling the previous vector multiplier from a list comprehension over the rows of the matrix. After that, we need to use `lists:filter/2` to filter out the zero elements of our list.
The last function is `mxm_list` which takes the two matrices given in the previously described format and calculates the product matrix using a list comprehension and the `mxv_list` function. After the list comprehension is compete, we have to filter out any elements of the product matrix that consist entirely of zeros. This means to filter out the columns that are entirely zeros, in this case represented by an empty list. This is also done by using the higher order function `lists:filter/2`.

Lists without higher order functions (`mxm_nohof`): This version uses the same algorithm and representation as the previous (`mxm_list`) version, but both instances of the `lists:filter/2` function were replaces by non higher order function written specifically to remove the zero or empty list elements from a list. The one removing zeros is called `filter_zeros`, the one removing the empty lists is `filter_empty`. The tail recursive implementation of these function can be seen

```
vxv_list(Row,Col) -> vxv_acc_list(Row,Col,0).

vxv_acc_list([],_,Acc) -> Acc;
vxv_acc_list(_,[],Acc) -> Acc;
vxv_acc_list([{I,R}|Row],[{I,C}|Col],Acc) ->
  vxv_acc_list(Row,Col,Acc+R*C);
vxv_acc_list([{I,R}|Row],[{J,C}|Col],Acc) ->
  if I < J -> vxv_acc_list(Row,[{J,C}|Col],Acc);
    true  -> vxv_acc_list([{I,R}|Row],Col,Acc)
  end.

mxv_list( Rows, Col ) ->
  Product = [ {I,vxv_list(Row,Col)} || {I,Row} <- Rows ],
  filter( fun({_,V}) -> V /= 0 end, Product ).

mxm_list(Rows, Cols) ->
  Product = [{I,mxv_list(Rows,Col)} || {I,Col} <- Cols],
  filter( fun({_,V}) -> V /= [] end, Product).
```

Figure 5.10: The `mxm_list` solution to the sparse matrix multiplication problem.

on Figure 5.11. After all elements of the list have been processed it is important to reverse the result list, since the matrix depends on the order of elements in the list.

```
filter_zeros([],R) -> lists:reverse(R);
filter_zeros([{_,0}|P],R) -> filter_zeros(P,R);
filter_zeros([H|P],R) -> filter_zeros(P,[H|R]).

filter_empty([],R) -> lists:reverse(R);
filter_empty([{_,[]}|P],R) -> filter_empty(P,R);
filter_empty([H|P],R) -> filter_empty(P,[H|R]).
```

Figure 5.11: The non-higher order function that replace `lists:filter/2` in the `mxm_nohof` version.

Parallel (`mxm_par`): This version is a basic parallel version of the `mxm_list` version. It uses the same representation of the matrices as in both of the previous versions. The parallelisation is done on two levels in this implementation. The matrix-vector multiplications and also the vector-vector multiplications are calculated in parallel. This is achieved by using list comprehensions in both `mxm_par` and `mxv_par` to spawn the processes that calculate the result of `mxv_par` and `vxv_par` respectively. `vxv_par` uses the recursive vector multiplier used in the sequential version. Since the order of the elements in a list is important in this case, we store the process IDs of the spawned processes (in the order that the elements need to be received), and these spanwed processes return a tuple of their PID and the result. Receiving is done inside a list comprehension, where only the result of the process with the next ID is received. After receiving, filtering the result is done the same way as in the sequential version, using `lists:filter/2`. This implementation can be seen on Figure 5.12.

```
vxv_par(Pid, Row, Col) ->
  Pid ! {self(), vxv_list(Row, Col)}.

mxv_par(Parent, Rows, Col ) ->
  IPids = [{I, spawn(?MODULE, vxv_par, [self(), Row, Col])} || {I,Row} <-
    Rows ],
  Product = [receive
        {Pid, Res} -> {I, Res}
      end || {I, Pid} <- IPids],
  Parent ! {self(),filter( fun({_,V}) -> V /= 0 end, Product )}.

mxm_par(Rows, Cols) ->
  IPids = [{I, spawn(?MODULE, mxv_par, [self(), Rows, Col])} || {I,Col} <-
    Cols],
  Product = [receive
      {Pid, Res} -> {I, Res}
      end || {I, Pid} <- IPids],
  filter( fun({_,V}) -> V /= [] end, Product).
```

Figure 5.12: The parallel implementation of sparse matrix multiplication

Arrays (mxm_array): The version using arrays uses the same base algorithm of reducing the problem to matrix-vector and then vector-vector multiplications. The representation of the matrices is different from the prevous versions. We used arrays inside arrays to represent the matrices. The main difference between this representation and the representation using lists is that in this case the positions of the zero elements are left as `undefined` in the arrays, and the non-zero elements are inserted at their positions. The two multiplied matrices are represented differently in this version as well, in the case of the first matrix the outer array represents the rows and the inner arrays are the columns, while in the case of the other matrix the outer array represents the columns and the inner one the rows. This is done to make multiplication easier. The first matrix from the example on Figure 5.9 is represented by the array seen on Figure 5.13.

```
{array,4,10,undefined,
{undefined,{array,2,10,undefined,
          {5,8,undefined,...,undefined}},
       {array,3,10,undefined,
          {undefined,undefined,3,undefined,...,undefined}},
       {array,2,10,undefined,
          {undefined,6,undefined,...,undefined}},
       undefined,...,undefined}}
```

Figure 5.13: Example of a matrix represented by extendible arrays, for the sake of clarity in some cases `...` is used in place of `undefined` elements

In the implementation of the algorithm we used `array:sparse_map/2` to replace list comprehensions. `array:sparse_map/2` maps all not `undefined` elements of an array. This is useful, because we have many `undefined` elements in our arrays and this way we do not need to handle then specially. `vxv_array` is not implemented recursively as in the case of the previous versions, instead we used the `array:sparse_foldr` and `array:sparse_map` functions to multiply the elements and add them. At the end of `mxv_array` and `vxv_array` we do not use filter to throw away zero columns or elements, instead a simple condition checks if we need to return `undefined` or the actual value. The Erlang code for this implementation can be seen on Figure 5.14.

```
vxv_array(Row,Col) ->
  A = array:sparse_foldr(
      fun(_,Val,Acc)->
        Acc + Val end, 0,
      array:sparse_map(
        fun(Index,Elem) ->
          C = array:get(Index,Col),
          if  C == undefined -> undefined;
            true -> Elem*C
          end end, Row)),
  if  A == 0 -> undefined;
    true -> A end.

mxv_array(Rows,Col) ->
  A = array:sparse_map(
    fun(_,Row) ->
      if  Row == undefined -> undefined;
        true -> vxv_array(Row,Col)
      end end, Rows),
  S = array:sparse_size(A),
  if  S==0 -> undefined;
    true -> A end.

mxm_array(Rows,Cols) ->
  array:sparse_map(fun(_,Col) ->
      if  Col == undefined -> undefined;
        true -> mxv_array(Rows,Col) end end,Cols).
```

Figure 5.14: Sparse matrix multiplication using arrays

The implementation does not depend on whether we use extendible or fix-sized arrays, so no separate versions were made for these, but when measuring we used both types of arrays.

Arrays without higher order functions (`mxm_array_nohof`): Using the same principle that we used to eliminate higher order functions, we also created a variation of the array version where higher order functions are replaced by non-higher order functions written by us. The functions shown on Figure 5.15 were used to replace `array:sparse_foldr` and `array:sparse_map` in the vector multiplier function. Both of these replacement function recursively iterate over all elements of the array and if the element is undefined then do nothing, otherwise calculate according to the functions hardcoded into them.

This version is also not dependent on whether we use fix-sized or extendible arrays, so we did not create two versions, but used both types of array when measuring.

```
vxv_array_foldr(Index,Size,_,Acc) when Index == Size -> Acc;
vxv_array_foldr(Index,Size,Array,Acc) ->
  Elem = array:get(Index,Array),
  if  Elem == undefined -> vxv_array_foldr(Index+1,Size,Array,Acc);
    true -> vxv_array_foldr(Index+1,Size,Array,Acc + Elem)
  end.

vxv_array_map(Index,Size,_,Row) when Index == Size -> Row;
vxv_array_map(Index,Size,Col,Row) ->
  ElemR = array:get(Index,Row),
  ElemC = array:get(Index,Col),
  if  ElemR == undefined -> vxv_array_map(Index+1,Size,Col,Row);
    ElemC == undefined -> vxv_array_map(Index+1,Size,Col,array:set(Index,
    undefined, Row));
    true -> vxv_array_map(Index+1,Size,Col,array:set(Index, ElemC*ElemR, Row)
    )
  end.
```

Figure 5.15: Some of the functions used to eliminate higher order functions from the array version.

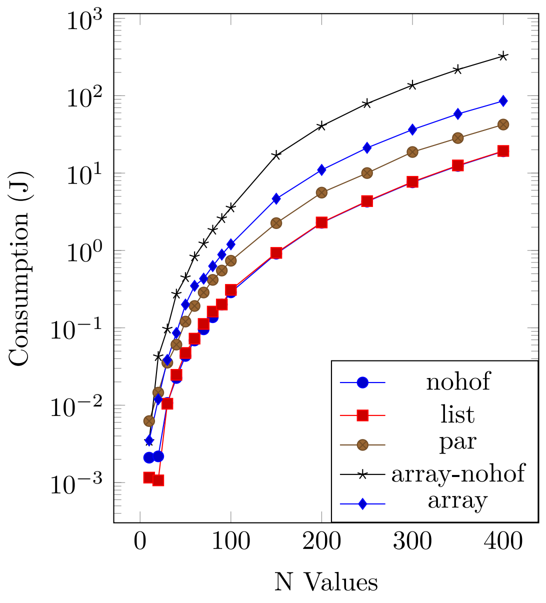
5.2.2 Results

Even though our implementations can handle non-square matrices, for ease of distinguishing between the sizes of test cases we only used square matrices. Test matrices were generated randomly, in sizes from 10×10 up to 400×400. For each size we generated three test cases with different ratio of non-zero elements. These ratios were 1%, 10% and 30% and even 100% to test the energy consumption of our algorithms using non-sparse matrices.

The array implementations were tested with both extendible and fix-sized arrays. The measured energy consumption values and runtimes for the 30% case are shown on Figure 5.16. Fix-sized array versions are not shown as they were not different from extendible array values in any meaningful way.

We calculated the correlation between time and energy consumption and for all functions the value of correlation is over 0.999 in all cases for all sizes and densities. From that we can see that the more time a program takes to run, the more energy it consumes. This correlation still does not give us any information about the relationship between the different implementations. For that, as in the case of the N-queens problem we calculated the power comsumption of all functions, calculated by dividing the sum of energy consumed for all test cases by the sum of runtimes. These power consumption values are shown on Figure 5.17. From these values we see that the parallel version consumes far more energy per second than all the others. Out of the sequential versions the arrays using no higher order functions consumed the least power, but their runtime was so big, that this may be the reason for that. The versions using lists consumed more energy per second than the others, but when inspecting total energy consumption, we will see that because they run faster, they consume less energy overall.

Upon inspecting total energy usage (Figure 5.16) we can see that both array versions consumed



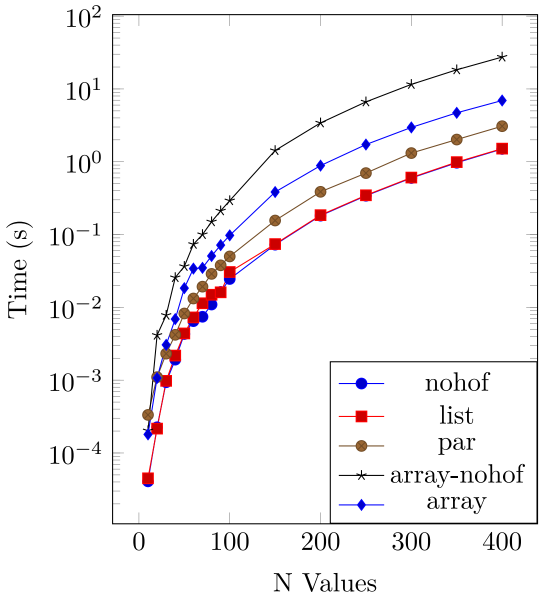


Figure 5.16: Energy consumption and run-time of sparse matrix multiplication implementations.

much more energy than the versions using lists. Out of the two array versions the one with higher order functions performed much better than the one without. The reason for this may be that `sparse_map` and `sparse_foldr` are specialized for sparse matrices, while our implementation of them had to test if an element is `undefined` or not.
The reason that arrays consumed more energy than lists may be that when representing matrices with arrays we left `undefined` elements in them, while when representing matrices with lists we used the lists as dictionaries to tell the position and the value of an element. To further inspect the difference between lists and arrays we measured our algorithms with non-sparse matrices, where arrays do not have any `undefined` elements. The results of this can be seen on Figure 5.18. On these graphs we can see that the array versions are closer to the list versions when using a non-sparse matrix, so part of the reason they performed so badly for sparse matrices may be because of the lots of `undefined` elements. On the other hand, they still perform worst of all implementations, even when matrices with 100% density are used, so other factors may contribute to making them consume more enegy than lists.

The parallel version performed worse than the sequential ones using lists, probably because of the same reasons as mentioned before in the case of the N-queens problem. There may be too many processes spawned in most cases, and spawning and scheduling that many processes may cost a lot of energy. Compared to the sequential versions, the DRAM usage of the parallelised version is also higher by a larger margin than the overall difference in energy consumption suggests, meaning that it used more RAM than the sequential versions. The reason for this may be that processes send the matrices and vectors between them, and that takes up space in RAM. Even though the parallel version was worse than the sequential version using lists, it still was better than both implementations using arrays. This may be because of the different representation of the matrices. Further investigation of the parallelisation of this problem can be seen in Section 5.3.

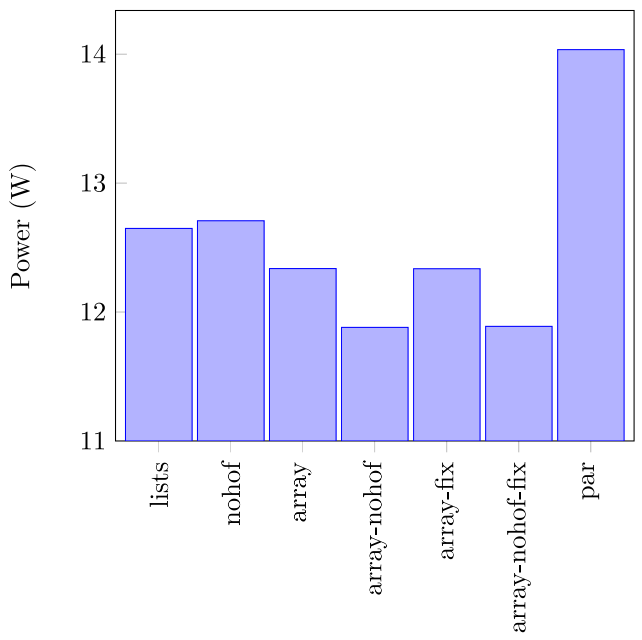


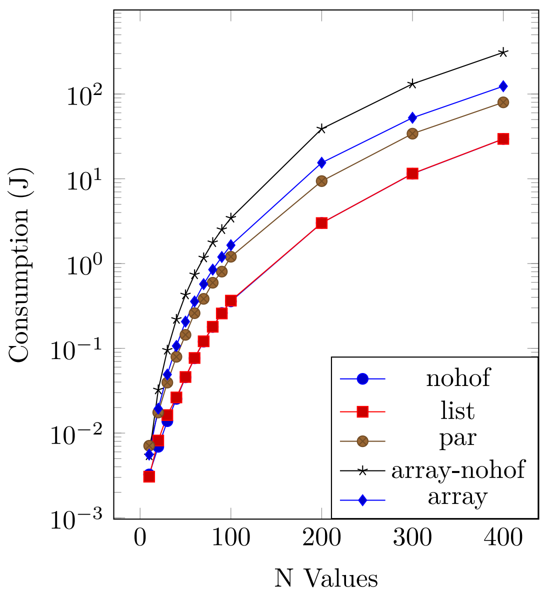
Figure 5.17: The average power consumption of all sparse matrix multiplication solutions for the 30% inputs. Note that the y axis begins at 11 watts.

Finally, on Figure 5.16 we see that the implementations using lists (`mxm_nohof` and `mxm_list`) performed almost the same, but upon inspecting the data in Table 5.2 we see that for all input sizes the one using no higher order functions consumed slightly less energy. Even though the energy saved by eliminating higher order functions is minimal, the amount of energy consumed still decreased. In the case of our matrix multiplication algorithms, higher order functions were not the major part of the code, but a small energy saving can be achieved even in this case. The only part of the original `mxm_list` implementation that used higher order functions was where we filtered the resulting lists. The most energy and time comsuming part of the algorithm is not this part of the code, but despite that it was beneficial to eliminate it.

	50	100	150	200	250	300	350	400
mxm_nohof	0.043	0.287	0.909	2.277	4.278	7.599	12.374	19.120
mxm_list	0.047	0.309	0.930	2.300	4.340	7.711	12.556	19.315

Table 5.2: Energy consumption of the given functions for different input sizes, measured in Joules, with 30% non-zero elements

In conclusion, the strong correlation between runtime and energy consumption stands for this problem as well.
We saw that choosing the correct representation and underlying data structure for a problem can have massive impact on the energy consumption of the program. In this case, arrays performed much worse than lists, and eliminating sparse higher order function made the array versions even worse.
The parallel version was not the worst, but it performed worse than the sequential versions using



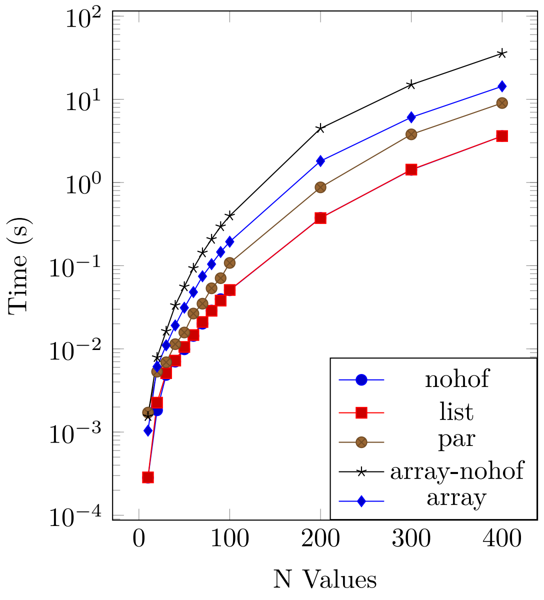


Figure 5.18: Energy consumption and run-time of sparse matrix multiplication implementations using non-sparse matrices.

lists. The reason for this may still be the number of processes spawned and also the increased DRAM usage.
Another important thing we observed was that eliminating higher order functions made the program consume less energy, even if by just a small amount.

5.3 Parallelisation of sparse matrix multiplication

In order to better investigate the effects of parallelisation on energy consumption, we implemented and measured different methods to parallelise the sparse matrix multiplication problem shown in the previous section. In this section we show the different ways we parallelised the problem and the energy consumption values of these parallel implementations.

The measurements in this section were made using a system with 12 cores available, using Intel(R) Core(TM) i7-8700K CPU and 16 GB of DDR4 RAM, using Ubuntu 17.04.

5.3.1 Solutions

Using parallel map (`mxm_parmap`): In this version, the original list version (Figure 5.10) was modified so that instead of list comprehensions the `lists:map/2` function is used. Then we replaced this map with our parallel map implementation, that also preserves the order of elements (`ord_par_map`). The preservation of the order of elements is achieved by storing the process IDs of spawned processes and then only receiving the result of the next in the list. In terms of complexity, this solution is the same as the `mxm_par` version shown in the previous section. We needed to use this method of introducing parallel map in order to later be able to introduce other methods of parallelisation, that depend on the use of a parallel mapping function. The `mxm_parmap` implementation is shown on Figure 5.19

```
vxv_parmap(Pid, Row, Col) ->
  Pid ! vxv_list(Row, Col).

mxv_parmap(Parent, Rows, Col ) ->
  Parent ! filter( fun({_,V}) -> V /= 0 end, ord_par_map(fun({I,Row}) -> {I,
    vxv_parmap(self(),Row,Col)} end, Rows) ).

mxm_parmap(Rows, Cols) ->
  filter( fun({_,V}) -> V /= [] end, ord_par_map(fun({I,Col}) -> {I,
    mxv_parmap(self(),Rows,Col)} end, Cols)).

ord_par_map(F, Xs) ->
  Me = self(),
  Pids = [spawn(fun() -> Me ! {self(), F(X)} end) || X<-Xs],
  [receive {Pid, Res} -> Res end || Pid <- Pids].

```

Figure 5.19: Parallel sparse matrix multiplication using parallel map.

Using process pools (`mxm_ppool`): Since the problem in the previous section seemed to be the number of processes spawned, we wanted to limit the number of possible processes, in order to improve the energy consumption. Limiting the number of processes was done using process pools. This means that we have a *dispatcher*, a *collector* and some *worker* processes. The dispatcher distributes the tasks to the workers, while the collector collects the results the workers produce. All a worker does is compute a function, send the result to the collector and signal to the dispatcher that it needs work again. This makes it possible to use a parallel map in our program without the number of processes getting out of hand. The process pool implementation we used can be seen on Figure 5.20. The parameters of `ordmap/3` are the F function that needs to be called with all elements of L list, and the N number of workers to spawn. This way our process pool spawns N + 2 processes.

To use this process pool with our sparse matrix multiplier, we simply replaced both instances of the parallel map in the previous (`mxm_parmap`) version with our new `ordmap` that uses process pools.

Limiting the use of process pools (`mxm_parseq`): Our goal was to limit the number of processes with the use of process pools, but in the `mxm_ppool` version we replaced both maps with process pools. That meant that in some cases even more processes were spawned than originally, because if we tell a process pool to spawn only N processes, but then the function that the process pool uses creates another process pool that spawns N processes, then we have spawned more than N^2 processes (including the dispatchers and collctors). We tried to solve this

```
ordmap(F, L, N) ->
  Main = self(), Index = lists:seq(1,length(L)),
  DPid = spawn(fun() -> dispatcher(lists:zip(Index, L)) end),
  CPid = spawn(fun() -> collector(0, length(L), [], Main) end),
  Workers = [spawn(fun() -> worker(F, DPid, CPid) end) || _<- lists:seq(1, N)
    ],
  [W ! init || W <- Workers],
  receive
    {value, CPid, Data} -> Data
  end,
  [W ! stop || W <- Workers],
  Data.

worker(F, DPid, CPid)->
  receive
    {data, {I, D}} ->
      CPid ! {ready, {I, F(D)}},
      DPid ! {done, self()},
      worker(F, DPid, CPid);
    init ->
      DPid ! {done, self()},
      worker(F, DPid, CPid);
    stop ->
      ok
  end.

collector(N, N, Acc, Main) ->
  Main ! {value, self(), lists:reverse(Acc)};
collector(C, N, Acc, Main) ->
  receive
    {ready, {I, Data}} when I == C+1 ->
      collector(C+1, N, [Data | Acc], Main)
  end.

dispatcher([]) ->
  ok;
dispatcher([H|T]) ->
  receive
    {done, W} ->
      W ! {data, H},
      dispatcher(T)
  end.
```

Figure 5.20: Process pool used to limit the number of spawned processes.

problem by only parallelising the outer map (inside the matrix-matrix multiplication) and then using the sequential `mxm_list` version to solve the rest of the problem. This way if we limit the number of workers of a process pool to N they will not spawn other processes.

5.3.2 Results

All of these parallel versions were measured on 2, 6 and 12 cores, but the number of cores had no effect on the relationships between the different functions. The only difference the cores made was in the exact amount of energy consumed. As we expected, with the increase of cores, the energy consumed by the program decreased, as it took less time to finish using multiple cores.

The `ppool` version was measured for N values of 4, 20 and 2000, so the number of spawned processes were around 16, 400 and 4 000 000 respectively. With the `parseq` version we only measured the program for N values of 20 and 2000 which in that case is approximately the same as the number of processes spawned. For these two cases the number of processes is the same regardless of the input size, but for the `parmap` version the input sizes influence the number of processes, which is proportional to the number of elements in the result matrix, for the 400×400, 30% case around 48 000.

The correlation between energy consumption and runtime was in all cases over 0.999, which shows that even in the case of parallel programs this correlation exists and is strong.

On Figure 5.21 the energy consumption of all measured parallel functions is shown for the 30% case, using 12 cores. From these energy consumption values it is clear, that the version with the most processes performed worse, probably because there were too many processes to handle. The second worst is the `parseq` version for N = 2000, even though it spawned less processes than the `parmap` version, which performed better. The reason for this may be that not only the number of processes, but the number of messages sent between the processes influence energy consumption. The use of process pools means the number of messages is much more than the number of messages without process pools, since the workers receive the data to process, sent it to the collector and then message the dispathcher again.

The `parmap`, `ppool 4` and `ppool 20` values are really close to each other. There may be many reasons for this, but we think that the around 400 processes for `ppool 20` may be slightly too many to handle efficiently, while the 16 processes of `ppool 4` may be too few to efficiently utilise the multiple cores, especially since most of these processes are continously killed and respawned as the outer function iterates over the elements. These `ppool` versions may be almost as bad as the `parmap` version because of the lots of messages and not ideal number of processes.

Finally, the best energy consumption values were achieved by the `parseq` version using N = 20. This means that using around the same number of processes as cores seems the most efficient way of limiting process numbers. This version is better than `ppool` for N = 4, because this does not kill and respawn its processes as it is solving the problem, instead spawns 20 processes in the beginning and assigns larger tasks (matrix-vector multiplication) to them. These larger tasks mean that the number of messages is also reduced, since workers do not need to send the results as often and also do not need to message the dispatcher as often.

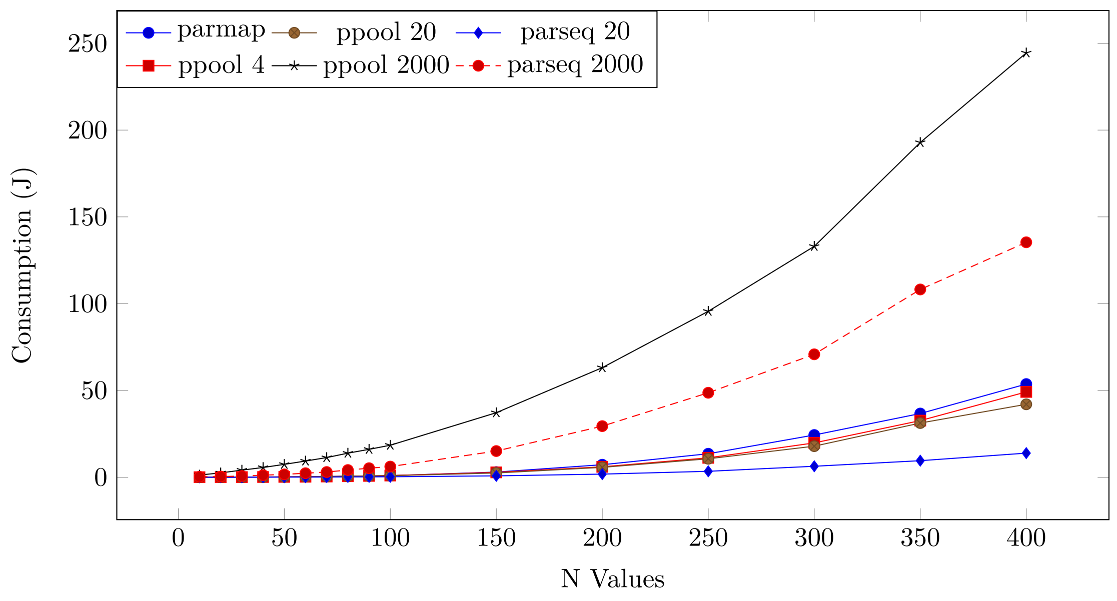


Figure 5.21: Energy consumption of different parallel implementations of sparse matrix multiplication. The numbers after the function names denote the N value givent to the process pools.

5.4 Token ring

The token ring algorithm is used to achieve mutual exclusion in a distributed system. The processes construct a logical ring, within which they have a well specified position. This can be achieved if all processes know which one is the following process in the ring. In the first step, the starting process is given a token. This token will circulate around the ring in messages. When a process acquires the token, it does its work with it and when finished, passes the token to its neighbor.

This algorithm is good for measuring the energy cost of spawned processes and sent messages, since the numbers of both are well defined, and no other significant computation takes place.

5.4.1 Solution

We have measured two very similar algorithms. Both of them get an integer `N` and a list of numbers `L` as inputs. They spawn `N` processes, and circulate every item of the list through the ring once.

They use the same solution to construct the ring and to pass forward the token. One algorithm increases every integer in the list by `N` and the other one calculates the square root of each element of the list taking `N` steps of Newton's method. So the only difference is the work that the processes do with the token.

The construction of the ring is shown on Figure 5.22.

The spawning of the processes happens with the function `lists:foldl`. `self()` is used as the base item and in every step, the function spawns a new process with the fucntion `Stage` which gets the PID of the last spawned process as the next element in the ring. Note that this method builds up the ring from behind, so the last spawned process will be the starting element in the ring. The PID of this process is stored in `Pid1`, so we can initiate the circulating process by sending messages to this first element. This happens in the next command, every list item is sent to `Pid1` with a forward tag. The only difference between the two algorithms is in this step, because in Newton's method we have to take the initial step and send the initial item as well. So this line changes to the following:

```
[Pid1 ! {forward, {H/2, H}} || H <- L]
```

After the sending of all items of the list is finished, one closing message is sent which signals the processes to stop. Finally, `self()` receives as many items as sent, when they come back.

The function of a process is accompished by the function `Stage` shown on Figure 5.23. It receives messages until a `finished` message is got. When this happens, the process terminates. Otherwise, the process calculates `NewData` using the function `process`. When it is ready, the process passes the new data to the next ring item and recursively waits for another message.

So what this token ring implementation does is dependent mostly on the `process` function. When we want to increase all elements by `N`, every process needs to increase the given item by 1. This function is visible on Figure 5.24. In the other case, the process needed to make an iterative step could be described by the following formula:

$$x_{k+1} = \frac{1}{2} * (x_k + \frac{n}{x_k})$$

where square root of n is to be determined. This formula needs the number n so we have to pass it as well. We use tuples to do this. This is shown on Figure 5.25.

```
ring(N, L) ->
  Pid1 = lists:foldl(fun(_, Pid) -> spawn(fun() -> stage(Pid) end) end,
      self(),
      lists:seq(1,N)),
  [Pid1 ! {forward, H} || H <- L],
  Pid1 ! finished,
  [receive
  {forward, D} ->
    D
  end || _ <- L].
```

Figure 5.22: Construction of the ring

```
stage(NextPid)->
  receive
  {forward, Data} ->
    NewData = process(Data),
    NextPid ! {forward, NewData},
    stage(NextPid);
  finished ->
    NextPid ! finished
  end.
```

Figure 5.23: One stage of the ring

```
process(Data) ->
  Data + 1.
```

Figure 5.24: Process function of the increasing token ring algorithm

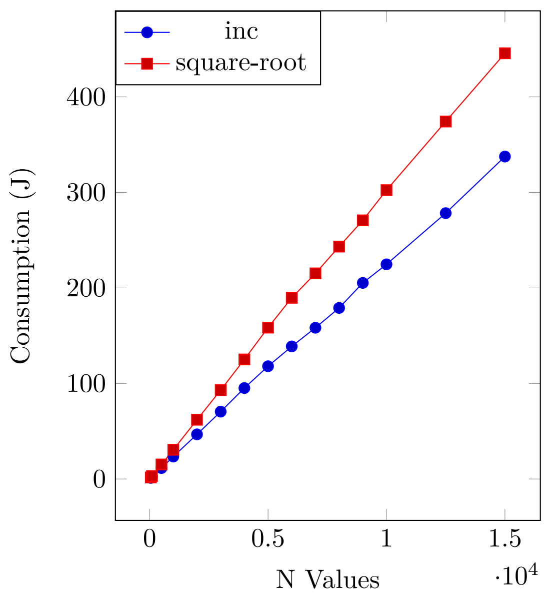
5.4.2 Results

As we have previously mentioned, this algorithm is effective to measure the cost of spawned processes and sent messages. We measured both algorithm with the same list: `lists:seq(1,10000)` but with increasing `N` values (from 50 up to 150000). We expected that if we raised the `N` values, energy consumption will increase following a linear scale since the amount of spawned processes and sent messages also grows linearly.

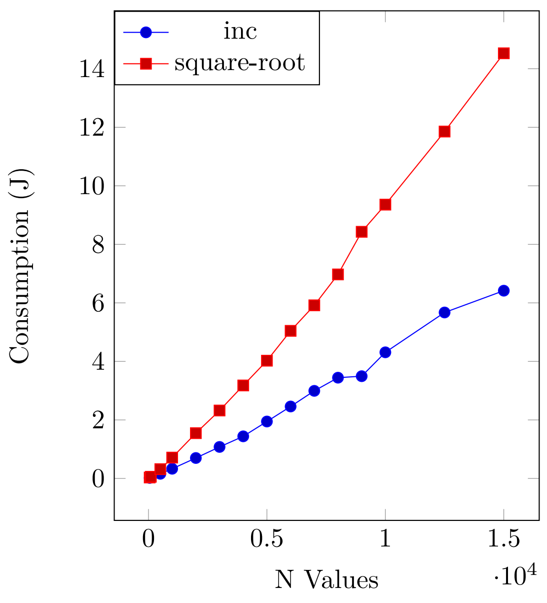
Our results are shown on Figure 5.26a. It is clear that the values increase almost in a linear scale as we have expected. The difference between the complexity of the two `process` functions is visible as well. It could be caused by the fact, that computing an iterative step is more complex than simply increasing a number, but there is another very important difference between the algorithms. While the incrementing version sends around only the values, the square root calculating version needs to send the initial number as well. So it is worth checking to see if this is visible in the energy consumption of the main memory. The energy usage of the main memory can be seen on Figure 5.26b. The consumption is also growing almost in a linear scale, but in this case the square root version has a steeper slope. This means that sending messages with more data will cost much more energy so it is important to send only the necessary information.

```erlang
process({Prev, I}) ->
  {(Prev + I/Prev) / 2, I}.
```

Figure 5.25: Process function of the square root calculate token ring algorithm



(a) The whole energy consumption.



(b) Energy consumption of main memory.

Figure 5.26: Energy consumption values of token ring algorithms.

Chapter 6

Results

In this chapter we summarise the results found when measuring the energy consumption of the algorithms and implementations shown in Chapter 5.

6.1 Data structures

During our measurements we inspected two data structures, lists and arrays. In all cases arrays performed worse than lists, the reason for this may be that in the case of N-queens the algorithm was first developed for lists and then adapted to arrays. In the case of sparse matrix multiplication the reason may be that we used a completely different representation when using lists than when using arrays. We saw that arrays perform worst when there are lots of `undefined` elements, so arrays are not the most efficient in storing sparse matrices. From comparing lists and arrays we learned that when refactoring our code to use arrays instead of lists, we have to be careful, because the most efficient representation for lists may not be as good for arrays.
From our observation it seems that arrays are always worse than lists, but there are cases, for example calculating Fibonacci numbers, where they may perform better [32].

We also used two types of arrays: extendible and fix-sized. There was not much difference in the energy consumption between the two types of arrays. The only difference was that in the case of the N-queens problem fix-sized arrays consumed slightly more energy, probably because in that implementation most of the time not all elements of the array were set, so it may have benefited from using extendible arrays.
In the case of sparse matrix multiplication the elements of the arrays were given in advance, and in the result array each element was calculated so there was no advantage in using either type of array. The energy consumed by both types of arrays was almost the same.

6.2 Higher order functions

We also wanted to find out whether eliminating higher order functions from our code was beneficial to energy consumption. We measured three cases where we eliminated higher order functions. In two of these case the energy consumption improved, both of these cases were where we used lists. The biggest improvement was achieved in the case of N-queens problem, where we replaced `lists:flatmap` and `lists:all`. The reason for such a big improvement may be that in this case most of the program consisted of using these higher order functions, so by eliminating them we influenced the energy consumption of all parts of the algorithm. In the case of matrix multiplication with lists, we saw that eliminating `lists:filter` slightly improved energy consumption for all test cases. The improvement was not as big as in the N-queens case, the reason for this is probably that using filter was not a major part of the matrix multiplication algorithm, so improving it had only a small effect on overall energy consumption. Still eliminating the higher order function was beneficial.
Lastly, when eliminating `array:sparse_foldr` and `array:sparse_map` from the matrix multiplication with arrays, we saw that the energy consumption became much worse. This may be because these higher order functions are specialised to sparse arrays, while our implementation of them had to check for `undefined` elements.
Apart from the case with arrays, we can be confident that eliminating higher order functions improves energy consumption. The amount of improvement largely depends on how much part of the program is dependent on these higher order functions.

6.3 Parallelisation

In the case of both problems we found that parallelising the algorithm made it consume more energy. The reason for this may be that both of these parallelisations were naive, and made the program spawn lots of processes, in some cases more than 500 000. Another problem may be that our programs sent lots of costly messages between the processes. In Erlang data sent to a process is copied [31], so our program made a copy of large lists each time a process was spawned. This can be seen by the increased energy usage of DRAM in these cases. From these result we can see that we have to be careful when parallelising an algorithm, because the number of processes and messages has a huge effect on energy consumption.

To find a more energy efficient way of parallelising our solutions, we used process pools to limit the number of processes. These implementations sent more messages for each value computed, but most of these messages were small signals to communicate between each other. We tried limiting the number of processes to different values and found that the most efficient solution is to have the number of processes be of the same magnitude as the number of CPU cores. We also found that besides the number of processes it is better to spawn the processes in the beginning and send them the tasks, than to kill and respawn these processes as needed.

Chapter 7

Related work

In recent years there have been lots of studies on the topic of energy-efficiency. Many of these works studied the energy consumption of imperative languages or other mainstream languages.

In Java 6.2% energy savings have been obtained [33] by using different Java Collection Frameworks (JFC) and optimizing based on calls to JFC methods. Thread management constructs also have been studied [34].

There also have been studies regarding the power consumption of CMOS digital circuits [35], where a technical method was given on how to trade silicon area on a chip against power consumption. Power analysis of embedded software [36] ahowed that power reduction up to 40% can be achieved by rewriting code using information provided by the instruction level power model.

Upon inspectiong the energy usage of Android applications, code obfuscations were found to increase energy usage [37] in most cases, but these increases were not deemed meaningful enough to impact users.

The impact of commonly used refactorings have also been studied [38]. The findings of this study show that refactorings can impact energy consumption in both a positive and negative way. It was also shown that commonly used metrics, that are believed to correlate with energy usage may fail to predict the effect of applying a refactoring.

Large amount of research was carried out for imperative languages, but green computing is just as important in the area of functional languages. Lima et al. [39] analysed the energy behaviour of Haskell. They presented tools for testing the energy footprint of a program and also showed that some constructs can be beneficial in some situations, while in others they may not be a good choice. They collected energy consumption data using Running Average Power Limit (RAPL [2]) and accessing the data through model-specific registers (MSRs).

7.1 Previous works in Erlang

In the case of Erlang only a few studies have been done regarding green computing. Varjão [40] gave a talk on the Erlang Factory SF conference in March 2017, where he presented a tool for measuring the energy consumed by Erlang functions. His tool is based on RAPL, but it has not been made publicly available yet, therefore we could not use it.

There has already been an MSc thesis [32] written by Ortiz, which also studied energy consumption in Erlang. She measured several algorithms (such as Fibonacci, sum list, Karatshuba) and implemented different refactoring steps to compare their power usage. We wanted to validate these results, so we re-measured some of the functions. She found that introducing new variables does not have considerable effect on energy usage, on the contrary, using higher order functions has a negative impact, which can be extremely huge. She also found that tail recursive functions consume less energy, than primitive recursion and reducing the data structures also helps to reduce energy consumption. Finally, using parallel versions reduces the amount of energy consumption, even when higher order functions are used. Our values certified her statements in these cases.

7.1.1 Fibonacci algorithm

The Fibonacci numbers are the numbers in the following integer sequence, called the Fibonacci sequence, which is given recursively: every number after the first two is the sum of the two preceding ones:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...$$

The Fibonacci algorithm takes N as input and gives the N$^{\text{th}}$ Fibonacci number. Ortiz measured a basic algorithm and its modifications after each refactoring step and compared them.

She measured these functions with the input number 40. She found that when we consider the primitive recursive solution as the starting point, higher order functions have a huge negative effect on energy consumption, although introducing new variables does not affect power usage in either of the cases. However, with parallelisation we can achive favorable results in energy consumption. We have evaluated the functions with several inputs, but we have found the same conclusion.

The following algorithms were implemented and measured:

Primitive recursion version (`fib_rec`): The base function uses a primitive recursion to get the result.

```
    fib_rec(0) ->    0;
    fib_rec(1) ->    1;
    fib_rec(N) ->
        fib_rec(N-1) + fib_rec(N-2).
```

Primitive recursion version with varialbes (`fib_recvar`): The first refactoring step to introduce the variables A and B.

```
fib_recvar(1) -> 1;
fib_recvar(0) -> 0;
fib_recvar(N) ->
    A = fib_recvar(N - 1),
    B = fib_recvar(N - 2),
    A + B.
```

Primitive recursion version with binding to list refactoring (`fib_bindinglist`): The next algorithm applies binding to list refactoring.

```
fib_bindinglist(1) -> 1;
fib_bindinglist(0) -> 0;
fib_bindinglist(N) ->
    [A, B] = [fib_bindinglist(N - 1), fib_bindinglist(N - 2)],
    A + B.
```

Primitive recursion version using HOF (`fib_hof`): This version uses the higher order function map to get the result.

```
fib_hof(1) -> 1;
fib_hof(0) -> 0;
fib_hof(N) ->
    [A, B] = lists:map(fun fib_hof/1, [N - 1, N - 2]),
    A + B.
```

Primitive recursion version using HOF and variable SubPr (`fib_hofvar1`): Based on the previous algorithm, introduce the variable SubPr.

```
fib_hofvar1(1) -> 1;
fib_hofvar1(0) -> 0;
fib_hofvar1(N) ->
    SubPr = [N - 1, N - 2],
    [A, B] = lists:map(fun fib_hofvar1/1, SubPr),
    A + B.
```

Primitive recursion version using HOF and variables SubPr and SubSols (`fib_hofvar2`): Based on the previous algorithm, introduce the variable SubSols.

```
    fib_hofvar2(1) -> 1;
    fib_hofvar2(0) -> 0;
    fib_hofvar2(N) ->
        SubPr = [N - 1, N - 2],
        SubSols = lists:map(fun fib_hofvar2/1, SubPr),
        [A, B] = SubSols,
        A + B.
```

Parallel version based on the higher order function version(`fib_parallel`): Parallelisation based on the higher order function version, use the function `pmap`. It should be mentioned that this algorithm uses a sequential solution when N is less than 15.

```
    fib_parallel(N) when N < 15, N >= 0 ->
        fib_p(N);
    fib_parallel(N) when N > 1->
        SubProb = [N-1, N-2],
        SubSol = pmap(fun fib_p/1, SubProb),
        lists:sum(SubSol).

    fib_p(0) ->
        0;
    fib_p(1) ->
        1;
    fib_p(N) when N > 1->
        fib_p(N-1) + fib_p(N-2).
    pmap(F, L) ->
        MyPid = self(),
        [spawn(fun() -> MyPid ! F(H)  end) || H <- L],
        [receive
            Res -> Res
        end || _ <- L].
```

On the graphs on Figure 7.1 it is visible that the three versions which are using higher order functions consumed much more energy than the other implementations, but amongst these there is not much difference. Similarly, introducing variables did not change the energy consumption of the primitive recursive version and binding to list refactoring had no effect on power usage either. However, with parallelisation we could save a little energy, as it is clear on the graph. It is worth to mention that for low N numbers the energy consumption of the functions is very mutable, but when N is above 20 the order becomes consistent. It could have been caused by the fast running times and very small energy consumption values.

7.1.2 Sum list

The sum list algorithm gets a list of integers as input and gives an integer as output: the sum of the numbers in the list.

In the MSc thesis four different algorithms were tested: a primitive recursive function, a tail

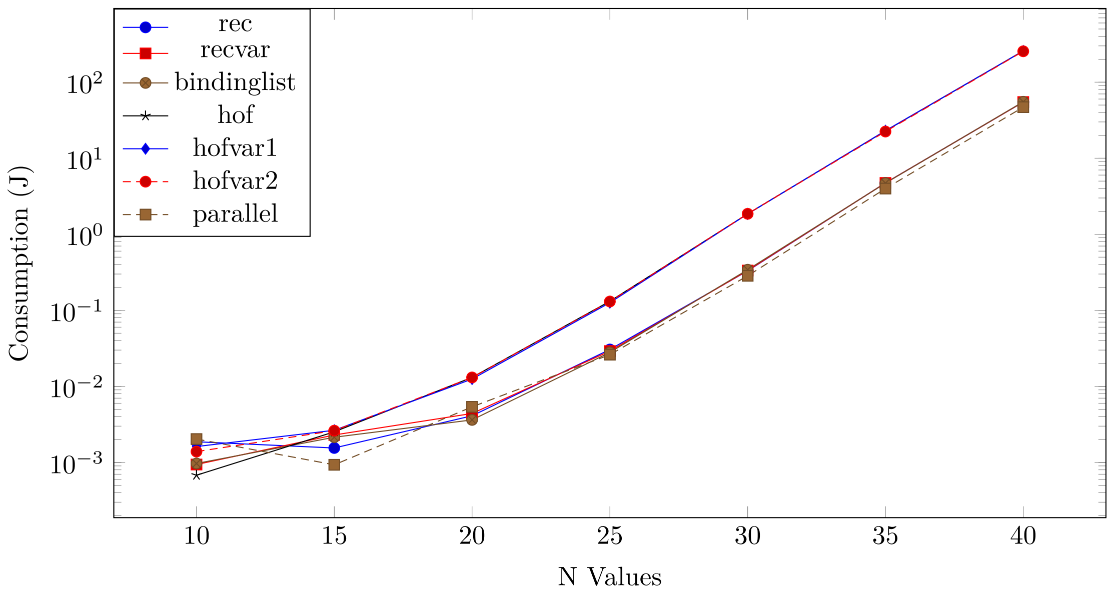


Figure 7.1: Energy consumption of all Fibonacci algorithm implementations. Note that the y-axis is in logarithmic scale, in order to make it easier to see the relation between different functions better.

recursive version, a tail recursive version with variables and finally one that uses the higher order function `foldl`. The conclusion of the thesis was very similar to the previous problem. Tail recursive function was the most efficient, introducing new variables did not change its power consumption. The primitive recursion was more costly, but again, the HOF version had the worst energy efficiency. We were able to validate these results as well.

In the thesis the functions were evaluated with the input `lists:seq(10,50000000)`. We have measured them with the same method, but with different sized lists. For example the smallest was `lists:seq(1,10000000)` and the longest was `lists:seq(1,70000000)`. The following functions were measured:

Primitive recursive version (`sum_rec`): This version recursively adds the sum of the tail list to the head item.

```
    sum_rec([]) -> 0;
    sum_rec([H|T]) ->
        H+sum_rec(T).
```

Tail recursive version (`sum_acc`): This version uses the `Acc` variable like an accumulator. In every recursive step this variable is increased by the value of the head item and the function call itself with the tail list.

```
    sum_acc([H|T]) ->
        sum_acc_help([H|T],0).

    sum_acc_help([H|T],Acc)->
        sum_acc_help(T,Acc+H);

    sum_acc_help([],Acc) ->
        Acc.
```

Tail recursive version with variable `X` (`sum_accvar`): Very similar to the previous function, but when incrementing the `Acc` variable a new variable is used.

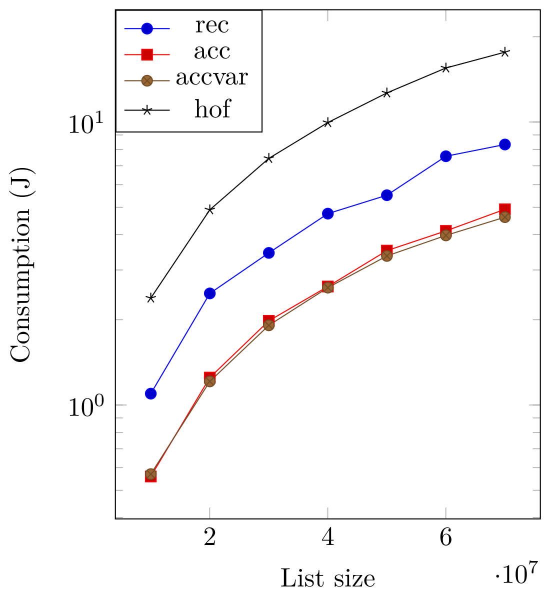
```
    sum_accvar([H|T]) ->
        sum_accvar_help([H|T],0).

    sum_accvar_help([H|T],Acc)->
        X=Acc+H,
        sum_accvar_help(T,X);

    sum_accvar_help([],Acc) ->
        Acc.
```

Higher order function version (`sum_hof`): This version uses the higher order function `lists:foldl` to get the result.

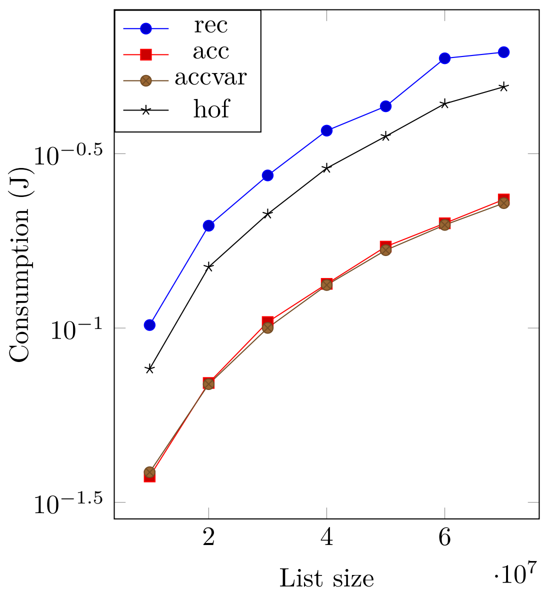
```
    sum_hof(L) ->
     lists:foldl(fun(X, Sum) -> X + Sum end, 0, L).
```

Our results are visible on Figure 7.2a. It is very clear that the version which is using the higher order function `foldl` has consumed the most energy by far, the primitive recursive version has cost a bit less and the two version of the tail recursive solution have performed the best. Between them there has not been much difference, but surprisingly sometimes the normal version has consumed a bit more energy than the version using a plus variable. We wanted to investigate the cause, so we have watched the avarage power consumption of the functions, which is showed on Figure 7.3. We could observe very similar facts on it, namely the version using the higher order function has used the most Joules per second and primitive recursion has used the second most. However, the consumption of the tail recursive versions turn around: the normal version performed better than the other, even if it is just a little difference. It means that the variable version has had better runtimes but have cost almost the same amount of Joules.

Another interesting topic comes out when we look at the energy consumption of the main memory on Figure 7.2b. Again, the two tail recursive version have performed the best, but we can notice that in case of main memory the primitive recursive version has cost more energy than the higher order function version.



(a) Whole energy consumption



(b) Energy consumption of main memory

Figure 7.2: Energy consumption values of sum list functions. a shows the whole consumption and b shows only main memory values. Note that the y-axis is in logarithmic scale.

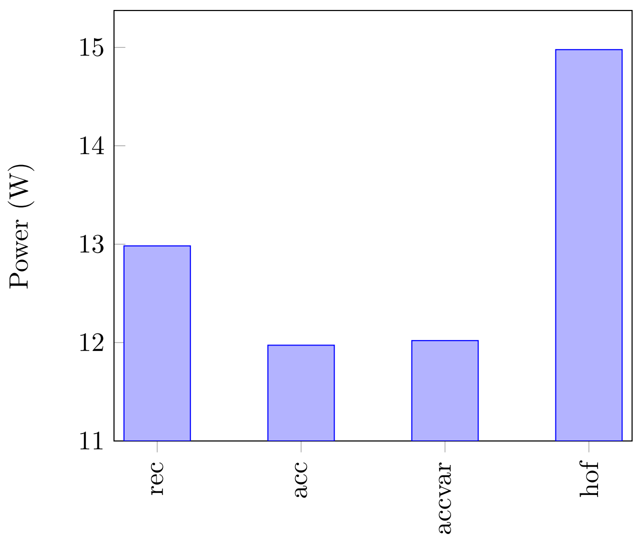


Figure 7.3: The average power consumption of all sum list solutions. Note that the y axis begins at 11 watts.

Chapter 8

Conclusions and future work

In this final chapter we summarise our work and findings and also give possibilities for future work regarding the subject.

8.1 Conclusion

Our goal was to measure the energy consumption of Erlang programs and discover patterns and relations between language constructs and power consumption. The RAPL [2] tool created by Intel provided the energy consumption values through model specific registers, that we read using the `rapl-read.c` [3] program. We also created an Erlang module that can be used to easily measure and store energy consumption values for different functions. Our module communicates with the C program that sends the measured values back to Erlang. Besides energy consumption we also measured and stored runtime values. Our Erlang module makes it easy to measure and then analyse the energy consumption of an Erlang function.

We also created a Python program that helps us visualize energy consumption and further process the data, such as calculating correlation between runtime and energy consumption and calculating average power consumption.

We had three main aspects that we took into consideration when chosing the algorithms to measure. These were the following:

- Using different data structures
- Use of higher order functions
- Parallelisation

We wanted to measure algorithms that make it possible to inspect the effect of all three previously mentioned aspects. Our choice was the N-queens problem and sparse matrix multiplication. We measured many different implementations of these problems, and also created multiple parallel versions of the matrix multiplication algorithm.

After measuring these implementations we found that eliminating higher order functions (HOFs) makes the program more efficient, especially when using lists. The method used to eliminate these higher order functions was to create a non-higher order function, that is written specifically to do the same thing as the HOF would have done, but the function argument of the HOF is hardcoded into the function.

In all of our implementations lists were more energy efficient than arrays, so it may be beneficial for a program to use lists instead of arrays. We found no significant difference between extendible and fix-sized arrays in terms of energy consumption.

The last thing we found is that parallelising our solutions initially made them consume more energy. The reason for this was that too many processes were spawned and too many messages were sent. When using process pools to limit the number of processes spawned the energy consumption improves significantly. When limiting the number of processes, the ideal number seems to be around the same magnitude as the number of cores that the program is run on.

In June 2018 we will present the findings of this thesis on the 12th Joint Conference on Mathematics and Computer Science (MACS'18) international conference in Cluj-Napoca [6]. A paper has also been submitted to the *Special Issue of Studia Universitatis Babes-Bolyai, series Mathematica, Informatica and Physica* and is currently under review [7].

8.2 Future work

We would like to measure algorithms using other data structures, such as gb_trees, digraphs, queues, dictionaries etc. We will need to find algorithms to use these data structures and measure their energy consumption.

We also want to further investigate the effects of eliminating higher order functions, as well as measuring and weighing the amount of energy that can be saved by eliminating different higher order functions.

We would like to confirm our current findings using other algorithms, such as the N-body problem, but we would also like to measure some language constructs individually, not inside a more complex problem, in order to confirm what we have found so far.

In the future we also would like to investigate different parallelisation techniques and we would like to further examine the effect of limiting the number of processes and messages sent on energy consumption. We also would like to measure the effect of the number of cores used when running the parallel program.

To better analyse the energy consumption of parallel programs, we want to establish a cost model, that helps us weigh different patterns. The basis for this may be the ParaPhrase Refactoring Tool for Erlang (PaRTE) [41], that is a tool that uses pattern discovery, cost modeling and other metrics to help discover candidates for parallelisation.

Our long term goal is to create a tool as part of RefactorErl [42], that helps automate the process of finding patterns that could be refactored into more energy efficient versions. RefactorErl is currently a static code analyser tool, but this refactoring and pattern detecting to improve energy consumption could also include deciding dinamically which version is the most efficient, by measuring the energy consumption of the program and making a decision based on the measured values.

Bibliography

[1] Joe Armstrong. *Programming Erlang.* The Pragmatic Bookshelf, 2nd edition, October 2013. ISBN 978-1-93778-553-6.

[2] Srinivas Pandruvada. Running average power limit - rapl. https://01.org/blogs/2014/running-average-power-limit-–-rapl, 2014. [Accessed: 2018.03.10.].

[3] Vincent M. Weaver. Reading rapl energy measurements from linux. http://web.eece.maine.edu/~vweaver/projects/rapl/, 2015. [Accessed: 2018.03.10.].

[4] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[5] Zoltán Horváth, László Lövei, Tamás Kozsik, Róbert Kitlei, Anikó Nagyné Víg, Tamás Nagy, Melinda Tóth, and Roland Király. Modeling Semantic Knowledge in Erlang for Refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, July 2009.

[6] Áron Attila Mészáros and Gergely Nagy. Towards green computing in Erlang, Abstract accepted to the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-17, 2018.

[7] Áron Attila Mészáros and Gergely Nagy. Towards green computing in Erlang, Paper submitted to the Special Issue of Studia Universitatis Babes-Bolyai, series Mathematica, Informatica and Physica, MACS'18, 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-17, 2018.

[8] Ericsson. Implementations and ports of erlang. http://erlang.org/faq/implementations.html, 2018. [Accessed: 2018.04.17.].

[9] Ericsson. Erlang. http://erlang.org, 2018. [Accessed: 2018.04.17.].

[10] Ericsson. Erlang/otp 20.3. http://erlang.org/doc/, 2016. [Accessed: 2018.04.20.].

[11] Fred Hébert. *Learn You Some Erlang for Great Good.* No Starch Press, 1st edition, January 2013. ISBN 978-1-59327-435-1.

[12] Umair Shahzad. Global warming: Causes, effects and solutions. *Durreesamin Journal*, Vol.1 Issue 4, 08 2015. ISSN 2204–9827.

[13] Gerhard Fettweis and Ernesto Zimmermann. Ict energy consumption – trends and challenges. In *The 11th International Symposium on Wireless Personal Multimedia Communications*, 2008. URL `https://mns.ifn.et.tu-dresden.de/Lists/nPublications/Attachments/559/Fettweis_G_WPMC_08.pdf`.

[14] Mark P. Mills. The cloud begins with coal - an overview of the electricity used by the global digital ecosystem. http://www.tech-pundit.com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf, 2013.

[15] Swasti Saxena. Green computing: Need of the hour. *International Journal of Current Engineering and Technology*, Vol.5, No.1 (Feb 2015), 2015. ISSN 2347–5161. URL `http://inpressco.com/wp-content/uploads/2015/02/Paper59333-335.pdf`.

[16] Mark Gregory. Inside facebook's green and clean arctic data centre. http://www.bbc.com/news/business-22879160, 2013. [Accessed: 2018.04.21.].

[17] Ayse Basar Bener, Maurizio Morisio, Andriy Miranskyy, and Sedef Akinli Kocak. Green it and green software. https://www.computer.org/web/computingnow/archive/october2014, 2014. [Accessed: 2018.04.17.].

[18] Techopedia. Green computing. https://www.techopedia.com/definition/14753/green-computing, 2018. [Accessed: 2018.04.17.].

[19] Vimal P.Parmar, Apurva K. Pandya, and Dr. CK Kumbharana. Optimization of energy usage for computer systems by effective implementation of green computing. *International Journal of Advanced Networking Applications (IJANA)*, 2011. ISSN 0975-0290. URL `http://www.ijana.in/Special%20Issue/16.pdf`.

[20] Patrick Fay. Intel platform power estimation tool (ippet) available. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/518161, 2014. [Accessed: 2018.04.21.].

[21] Seung-Woo Kim, Karthik Krishnan, Vardhan Dugar, Joseph Jin-Sung, Jun De Vega, Mike Yi, Joe Olivas, Patrick Konsor, Martin Dimitrov, and Carl Strickland. Intel power gadget. https://software.intel.com/en-us/articles/intel-power-gadget-20, 2014. [Accessed: 2018.04.21.].

[22] Ubuntu. turbostat - report processor frequency and idle statistics. http://manpages.ubuntu.com/manpages/xenial/man8/turbostat.8.html, 2010. [Accessed: 2018.04.21.].

[23] The University of Tennessee Innovative Computing Laboratory. Performance application programming interface. http://icl.utk.edu/papi/, 2017. [Accessed: 2018.04.21.].

[24] Timothy Chagnon. Papi (3.6.1) on ubuntu (8.04). https://www.cs.drexel.edu/ tc365/papi361u804.html, 2008. [Accessed: 2018.04.21.].

[25] Linux Foundation. Linux thermal daemon monitors and controls temperature in tablets, laptops. https://www.linuxfoundation.org/blog/linux-thermal-daemon-monitors-and-controls-temperature-in-tablets-laptops/, 2013. [Accessed: 2018.04.21.].

[26] Wander Lairson Costa. Power profiling overview. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Power_profiling_overview, 2015. [Accessed: 2018.03.10.].

[27] Intel. Intel 64 and ia-32 architectures software developer's manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2018. [Accessed: 2018.04.18.].

[28] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 455–470, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4305-3. doi: 10.1145/2989081.2989088. URL `http://doi.acm.org/10.1145/2989081.2989088`.

[29] The Scipy community. scipy.stats.pearsonr. `https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.pearsonr.html`, 2014. [Accessed: 2017.04.29].

[30] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[31] Ericsson. Erlang efficiency guide, processes. http://erlang.org/doc/efficiency_guide/processes.html, 2018. [Accessed: 2018.03.15.].

[32] Jessica Tatiana Carrasco Ortiz. Green computing in erlang, 2017.

[33] Rui Pereira, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. The influence of the java collection framework on overall energy consumption. *CoRR*, abs/1602.00984, 2016. URL `http://arxiv.org/abs/1602.00984`.

[34] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. *SIGPLAN Not.*, 49(10):345–360, October 2014. ISSN 0362-1340. doi: 10.1145/2714064.2660235. URL `http://doi.acm.org/10.1145/2714064.2660235`.

[35] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, Apr 1992. ISSN 0018-9200. doi: 10.1109/4.126534.

[36] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, Dec 1994. ISSN 1063-8210. doi: 10.1109/92.335012.

[37] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, and J. Clause. How does code obfuscation impact energy usage? In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 131–140, Sept 2014. doi: 10.1109/ICSME.2014.35.

[38] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2774-9. doi: 10.1145/2652524.2652538.

[39] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 517–528, March 2016. doi: 10.1109/SANER.2016.85.

[40] Filipe Varjão. Measuring erlang energy consumption, and why this matters. http://www.erlang-factory.com/sfbay2017/filipe-varjao.html, 2017. [Accessed: 2018.03.25.].

[41] István Bozó, Viktoria Fordós, Zoltán Horvath, Melinda Tóth, Dániel Horpácsi, Tamás Kozsik, Judit Köszegi, Adam Barwell, Christopher Brown, and Kevin Hammond. Discovering parallel pattern candidates in erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pages 13–23, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3038-1. doi: 10.1145/2633448.2633453. URL `http://doi.acm.org/10.1145/2633448.2633453`.

[42] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Köszegi, Tejfel. M., and M Tóth. Refactorerl - source code analysis and refactoring in erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2*, pages 138–148, Tallin, Estonia, October 2011.