STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXIII**, Number 1, 2018
DOI: 10.24193/subbi.2018.1.05

TOWARDS GREEN COMPUTING IN ERLANG

ÁRON ATTILA MÉSZÁROS, GERGELY NAGY, ISTVÁN BOZÓ, AND MELINDA TÓTH

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. The 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 (such as higher order functions), parallelism, data structures and styles of programming on energy usage. Additionally we present a tool to measure and visualise the consumed energy.

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 a 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 been already presented on energy efficient computing (see Section 2), however most of them are focusing on mainstream languages. The goal of our research is to investigate the energy usage of Erlang [1] programs.

Received by the editors: March 31, 2018.

2010 *Mathematics Subject Classification.* 68N18.

1998 *CR Categories and Descriptors.* F.2.m [**ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY**]: Miscellaneous – *Energy usage*; F.3.22 [**SOFTWARE ENGINEERING**]: Miscellaneous – *Green computing* .

Key words and phrases. green computing, energy efficiency, Erlang, RAPL.

This paper was presented at the 12th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, June 14-–17, 2018.

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Erlang is a widely used functional programming language, designed for building concurrent/distributed soft-real time applications. Since Erlang is functional language and the main building blocks of the language are the functions, we have created a tool (see in Section 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 paper 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 Section 4). We demonstrate our finding 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 paper we are presenting our first findings on the energy usage on Erlang programs, and the latter mentioned goal is the target of our future work.

2. 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, for example in Java 6.2% energy savings have been obtained [6] and thread management constructs also have been studied [7]. There also have been studies regarding the power consumption of CMOS digital circuits [8], power analysis of embedded software [9], how code obfuscation affects energy usage [10] and finally the impact of commonly used refactorings have also been studied [11].

Large amount of researches were carried out for imperative languages, but green computing is just as important in the area of functional languages. Lima et al. [12] 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).

In the case of Erlang only a few research has been done regarding green computing. Ortiz [13] wrote her MSc thesis on the topic of green computing in Erlang and showed how some steps of refactoring and different data structures affect energy consumption in the case of Fibonacci and Karatsuba algorithms. Varjão [14] 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.

3. Measuring and visualising energy usage

In this section we present the tools used to measure the energy usage of Erlang functions. 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. It is available under Linux operating system, on CPUs that have at least Sandy Bridge or newer architecture. The onboard power meter have been introduced in the Sandy Bridge microarchitecture. This provides information on power meters and power limits of the CPU, and exports power information through a set of Model-specific Registers (MSRs).

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 performance counters and I/O models [2].

In RAPL, platforms are divided into domains, in order to get more detailed information on the energy consumption. The domains are:

- PKG - The entire package
  - PP0 - Only the cores
  - PP1 - The uncore part of the package
- DRAM - Main memory

Among the listed domains the following inequality always holds: PP0 + PP1 $\leq$ PKG and DRAM is independent of the other three [15].

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

- `sysfs` - Reads files from /sys/class/powercap/intel-rapl/intel-rapl:0 using the powercap interface. Requires at least Linux 3.13 with no special permissions.
- `perf_event` - Uses the perf_event interface. Requires at least Linux 3.14, and root access or the /proc/sys/kernel/perf_event_paranoid value to be less than 1.
- `msr` - Reads data directly from the MSRs under /dev/msr, and requires root privileges.

It is important to note that all methods provide readings for an entire CPU socket, there is no way to get readings for individual cores and processes this way.

We used a program written in C called rapl-read [3], that provides an interface for all three methods. We had to slightly modify it to support multiple sockets and to send and receive signals to and from our Erlang program. We also had to split all measuring functions to a pre and post versions, so that we can read the data before and after running an Erlang function, and thus obtaining the energy consumption.

3.3. **Erlang server.** 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. This is accomplished using ports in Erlang and using the `read()` and `write()` functions in C. The process of communication can be seen on Figure 1.

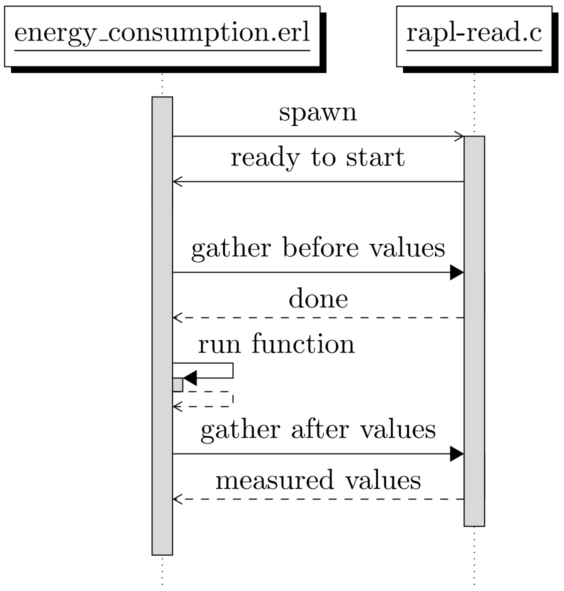


FIGURE 1. Sequence diagram of `energy_consumption.erll` and `rapl-read.c` communicating throughout one measurement

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.

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.

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 run-time of the function and store it with the energy usage values.

This process is repeated N times, where N is a parameter given by the user. 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 also printed to a text file, thus we can easily plot the data.

To make measurement processing easier we created a function that performs all of the above described functionalities. This function takes six arguments, that in addition to the ones mentioned above, are the executable file compiled from `rapl-read.c`, and the names of the output files:

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

3.4. **Visualisation.** As we have mentioned earlier, the measured results were exported to text files. These files contain one result in each line in the following structure:

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

Above the arguments the output contains the method used in 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 are irrelevant, thus to preserve the same structure, these fields are containing the measured time as well.

To visualise the measured data, it would had taken a lot of time to analyse manually these data files (for example in a spreadsheet). We decided to process the raw data with a Python script.

The script that processes the raw data has the following stages:

(1) Grouping the measured values by the functions. A function can be uniquely identified by the name of the module and the name of the function.
(2) Extracting the values separately by the different methods and domains.
(3) Draw the figures using *matplotlib* [16]: the Y-axis shows the energy consumption in Joule and the X-axis shows the different input size values.
(4) Optionally, there is a feature for export the required values to another data file and generate a latex file from it that contains only the diagram.

The steps are fully customisable to help analyse the data more precisely. We used command line arguments to give the exact specification of the diagrams. The following flags are available:

- –files: The script can take multiple files, all the contained data 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.

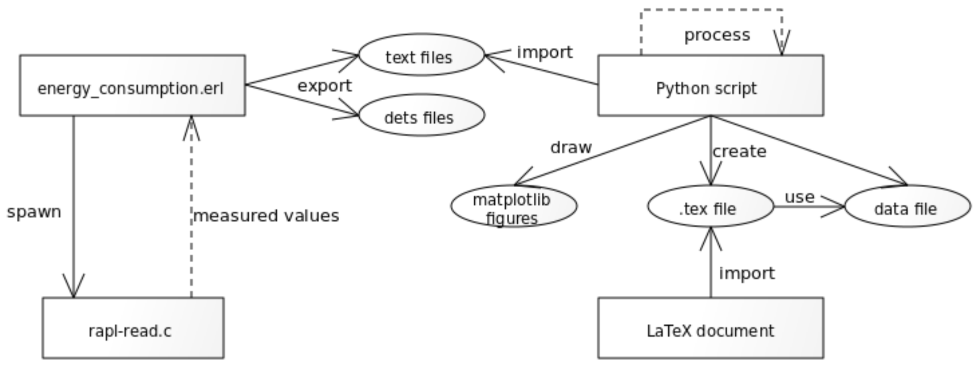


FIGURE 2. Process from measurement to visualisation

4. Energy usage in Erlang

We used two different problems and many implementations for them to gain information on the energy consumption. We used different language constructs and data structures in the implementations. The main aspects for our implementations were the following:

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

We also paid attention to the run-time of each implementation, in order to gain a better understanding of the effects of language constructs on energy consumption. The measurements 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 MSR method and PKG + DRAM domains.

4.1. Goals.

Our goal was to find any relation between language constructs, different data structures and energy usage. We intended to find out if extendible arrays, fix-sized arrays or lists are more efficient. We also wanted to, among other things, gain information on the effects of higher-order functions on energy consumption. Another thing we wanted to find out was the effect of parallelising on energy consumption. In the followings we are demonstrating different implementations of two well-known algorithms (placing queens on a chessboard and sparse matrix multiplication) and the energy used when evaluating the different implementations.

4.2. N-queens.

Problem. Place N queens on an N×N chessboard, so that no two queens attack each other.

Solutions. For this problem we measured the following five implementations:
Lists with HOFs (`queens_lists`): This version uses the higher-order functions `lists:flatmap/2` and `lists:all/2` to get the results.

```
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,[]).
```

Lists without HOFs (queens_nohof): The same as the previous one, but instead of lists:flatmap/2 and lists:all/2 it uses a custom implementation for these functionalities. These functions do not need function parameters, because we have hardcoded this information into them.

```
all_nohof(_,[]) -> true;
all_nohof(Queen,[Q|Queens]) ->
        G = attacks(Queen,Q),
        if G -> false;
        true -> all_nohof(Queen,Queens) end.
flatmap_nohof([],R,_,_) -> R;
flatmap_nohof([H|T],R,N,Row) ->
        L = solve_nohof(N,Row+1,H),
        P = concat_to(L,R),
        flatmap_nohof(T,P,N,Row).
concat_to([],R) -> R;
concat_to([H|T],R) -> concat_to(T,[H|R]).
```

Extendible arrays (queens_array): This version uses the same algorithm as the one using lists, but instead uses arrays created with array:new(). No HOFs are used in this implementation.
Fix-sized arrays (queens_array_fix): This is the same as the one with extendible arrays, but instead of array:new(), we create fix-sized arrays with array:new(Size).
Parallel version (queens_par): This version uses a parallel map (par_map/2) instead of map. The underlying data structure is a list.

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

Each implementation was measured for inputs from 6 through 12.
Results. The energy consumption and run-time of these implementations can be seen on Figure 3. We can see that in the case of the sequential versions, the slower program consumes more energy, as expected. In the parallel case, even though the parallel version is not always the slowest, it clearly consumes the most energy. The reason for this may be that this parallel version just replaced map/2 with par_map/2, not paying attention to the underlying data structure,

but in the case of Erlang all data sent between processes is copied [17]. Because the items mapped are lists, lots of data is copied each time `par_map/2` is called, which is slow, and thus consumes lots of energy.

Both implementations with arrays performed the same for smaller inputs, but for larger inputs the fix-sized array version consumed more energy.

It is clear that the most efficient versions were the ones using lists. It seems like eliminating the higher order functions from the implementation improved energy consumption, and also made the program faster.

In conclusion, we saw that eliminating HOFs, such as map can improve energy consumption. We also saw, that parallelising can be dangerous, we have to pay attention to what data is sent. Arrays seem to be performing worse than lists, but are still more efficient than our naive parallel algorithm.

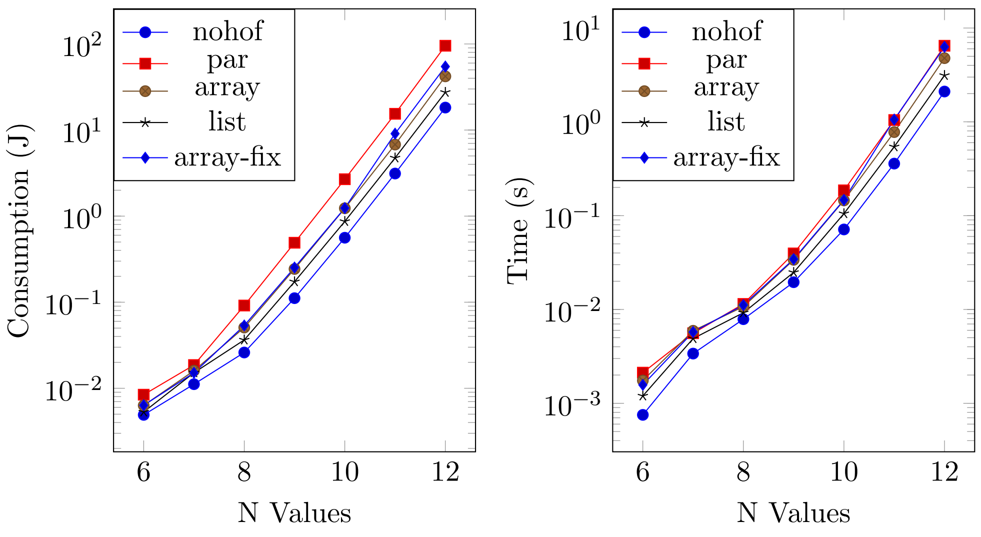


FIGURE 3. Energy consumption and run-time of all N-queens implementations.

4.3. **Sparse matrix multiplication.**

Problem. Multiply two matrices, whose elements are mostly zeros.

Solutions. All implementations solve the problem by reducing it to a series of matrix-vector multiplication, and then to vector-vector multiplication. For this problem we measured the following implementations:

Lists (`mxm_lists`): One of the matrices is represented as list of tuples, whose first element is the row number, the second element is also a list of tuples, whose first element is the column number and the second is the value of the matrix element: $[\{Row, [\{Col, Value\}]\}]$. The other matrix is represented the same way, but with row and column values swapped, in order to make multiplying them easy.

```
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).
```

Lists without HOFs (`mxm_nohof`): In this version we replaced all occurrences of the filter function with our own implementation, that does not need a function as its argument. For example the one in `mxv_list` was replaced by `filter_zeros`:

```
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]).
```

Parallel (`mxm_par`): The representation of the matrices is the same as in the version with lists, but the matrix-vector and vector-vector multiplications are executed in parallel, using spawn in the list comprehension. Receiving the data sent back by the processes is done in another list comprehension, with each process sending back its ID too, in order to find the proper place for the result in the matrix.

Parallel with process pools: We also implemented a parallel version with a parallel ordered map implementation, that uses process pools to limit the number of processes spawned. First, we replaced both instances of the list comprehensions in the parallel code, but this way if we limit the process pool to 20 processes, then because of the recursion in our code $20^2 = 400$ processes were spawned. An easy way to fix this was to only parallelise the outer matrix-vector multiplications and use the sequential program for solving vector-vector multiplications. These versions were named, respectively, `mxm_ppool` and `mxm_parseq`.

Array (`mxm_array`): In this version the matrices are represented as arrays of arrays, where for one matrix the first index determines the row and the second the column, and for the other matrix it is the other way around. Zero elements of the matrix are left undefined. We used `array:sparse_map/2` and `array:sparse_foldr/3` to replace the list comprehensions and recursion on lists, so for example the vector-vector multiplication became the following code:

```
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.
```

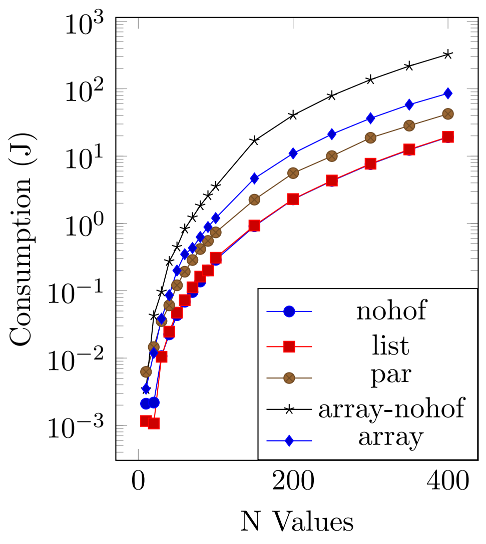
The implementation does not depend on whether we use extendible or fix-sized arrays, so no separate versions were made for these, but when measuring we used both types of arrays.
Array without HOFs (`mxm_array_nohof`): The same way we eliminated higher order functions from the list version, we also eliminated HOFs (`sparse_map` and `sparse_foldr`) in the version using arrays. The `sparse_map` of the previous `vxv_array` function became the following non higher order function:

```
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.
```

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%.

The array implementations were tested with both extendible and fix-sized arrays. The measured energy consumption values and run-times are shown on

Figure 4. Fix-sized array versions are not shown as they were not different from extendible array values in any meaningful way.



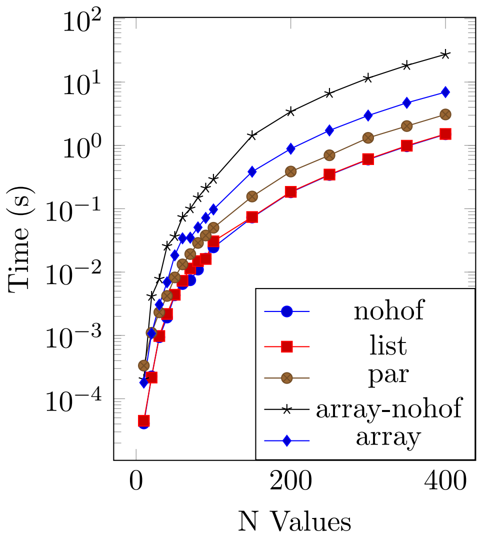


FIGURE 4. Energy consumption and run-time of sparse matrix multiplication implementations.

In all cases we see correlation between run-time and energy consumption, as expected. We can see that the versions using lists performed almost the same. The one without HOFs is in almost all cases slightly better than the one using filter as can be seen in Table 1. It can also be seen that all implementations using arrays performed worse than lists. One reason for that might be that in the case of arrays, the matrices were stored directly in the array, not as pairs of indices and values (as in the case of lists), thus resulting in lots of undefined elements in the arrays and increasing the size of stored data. Contrary to the versions using lists, in the case of arrays the one containing HOFs performed better than the one without HOFs. The reason for this might be that we do not know how arrays are implemented, and thus we cannot implement the non higher order versions of `sparse_map` and `sparse_foldr` as efficiently.

The parallel version performed worse than the sequential ones, probably because of the same reasons as mentioned before in the case of the N-queens problem. The effect of using process pools can be seen on Figure 5. We can see that spawning too many processes is really bad for power consumption. The basic parallel version and the process pool version with the number of processes limited to 4 and 20 processes (thus only creating 16 and 400 processes at a

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

time) perform almost the same. The reason for this might be that even though the process pool version uses fewer processes, it also sends more messages. The most effective parallel version was the half parallel, half sequential process pool version, limited to 20 processes. This shows that it can be worth it to limit the number of processes. These results require further analysis in order to find the true connection between the number of processes, messages and energy consumption.

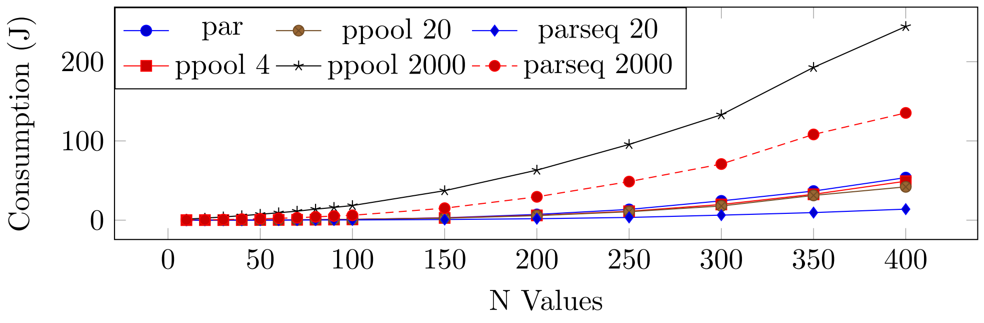


Figure 5. Energy consumption of different parallel implementations of sparse matrix multiplication. The numbers after the function names denote the number of processes in the process pool (note that in the case of pool, this number has to be squared to get the actual number of processes). These measurements were made using 12 cores on a system with Intel(R) Core(TM) i7-8700K CPU and 16 GB of DDR4 RAM, using Ubuntu 17.04.

On all figures for this problem the inputs with 30% non-zero elements are plotted.

5. Evaluation

In both cases we have found that naively parallelising made the algorithm consume more energy. This is probably because we did not use any special strategy to parallelise the algorithms, we simply replaced map with parallel map and list comprehension with list comprehensions that spawn processes. Even though spawning processes has a relatively low cost, since we spawned so many of them, in the case of N-queens sometimes more than 500 000, it may have increased energy consumption and run-time. Another problem may have been, that in Erlang data sent to a process is copied [17], so our program made a copy of large lists each time a process was spawned.

We have seen that using process pools to reduce the number of processes can be beneficial, but we have to choose the maximum number of processes wisely, because it greatly affected energy consumption.

We also observed, that in most cases eliminating higher-order functions improves energy consumption. This is most visible in the case of the N-queens problem. It can also be seen in the case of the sparse matrix multiplication problem, but to a lesser extent. In that algorithm the HOF filter does not take up that much part of the whole solution, so it contributes less to overall energy consumption. While in the N-queens algorithm map is used as the base of the algorithm, so it contributes much more to the energy consumed. An outlier to this rule is the array version of sparse matrix multiplication, where eliminating HOFs made the algorithm much worse. This might be because in the version using HOFs we did not use the traditional `map` and `foldr`, but instead the sparse version of them, which may be implemented much more efficiently than our own implementations.

The third thing we noticed was that in both cases arrays performed worse than lists. That may be because in the case of N-queens our algorithm was first developed for lists and then adapted to arrays. In the case of the matrix multiplication the reason may be that we stored data in a completely different way in arrays than in lists. From the results it seems like the array version is not efficient in storing sparse matrices. Even though our observation was that arrays are not as efficient as lists, there are cases, for example calculating Fibonacci numbers, where they may perform better [13].

6. Conclusion and future work

We wanted to measure the energy consumption of Erlang programs and discover patterns and relations between language constructs and power consumption. We used RAPL to measure energy consumption, and created an framework to measure and store energy consumption values.

After measuring several implementations of the N-queens problem and the sparse matrix multiplication we found that eliminating higher order functions may make the program more efficient. In our cases we also found that using arrays instead of lists was not a good idea. Parallelising the solutions can make energy usage worse, because it spawns lots of processes, but this could be solved using process pools.

In the future we 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. Additionally, we would like to confirm our current findings using different algorithms, such as the N-body problem.

Finally, we would like to create a tool as part of RefactorErl, that automates the process of finding patterns that could be refactored into more energy efficient version and then helps transform the code into a more energy aware version.

References

[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. [Accessed: 03.10.2018.].

[3] Vincent M. Weaver. Reading RAPL energy measurements from Linux. http://web.eece.maine.edu/~vweaver/projects/rapl/. [Accessed: 03.10.2018.].

[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 *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] 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`.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Jessica Tatiana Carrasco Ortiz. Green computing in Erlang, 2017.

[14] Filipe Varjão. Measuring Erlang energy consumption, and why this matters. http://www.erlang-factory.com/sfbay2017/filipe-varjao.html. [Accessed: 03.10.2018.].

[15] Wander Lairson Costa. Power profiling overview. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Power_profiling_overview, [Accessed: 03.10.2018.].

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

[17] Ericsson. Erlang Efficiency Guide, Processes. http://erlang.org/doc/efficiency_guide/processes.html. [Accessed: 03.10.2018.].

ELTE, Eötvös Loránd University, Pázmány Péter sétany 1/C, Budapest, Hungary, 1117

Email address: {archy, nagygeri97, bozoistvan, tothmelinda}@caesar.elte.hu