HPC Benchmark Game: Comparing programming languages regarding energy-efficiency for applications from the HPC field

Max Lübke[1][0000−0008−9773−3038], Dorian Stoll[1], Bettina Schnor[1][0000−0001−7369−8057], and Stefan Petri[2][0000−0002−4379−4643]

[1] University of Potsdam, Institute of Computer Science, An der Bahn 2, 14476 Potsdam, Germany {max.luebke,dorian.stoll}@uni-potsdam.de, schnor@cs.uni-potsdam.de

[2] Potsdam Institute for Climate Impact Research (PIK), Member of the Leibniz Association, P.O. Box 6012 03, D-14412 Potsdam, Germany, stefan.petri@pik-potsdam.de

Abstract. This paper presents a benchmark suite for the HPC field, called the HPC Benchmark Game which allows comparing programming languages and compilers regarding their runtime performance and energy-efficiency. We started with 3 compiled languages (C, C++ and Fortran) and Julia which is a just-in-time compiled language. Julia has native support for threads, distributed computing, and GPU offloading, which makes it a promising candidate for HPC. For each language, we picked one benchmark as reference and re-implemented it for the other languages. This paper describes our guidelines for the re-implementation. Further, we demonstrate the benefit of the Benchmark Suite through measurements on a 128-core node. The results help an HPC programmer to decide which languages and compilers are recommended on a system for energy-efficiency. The presented results show that HPC developers still have to invest some effort to find an energy-efficient implementation of their algorithm on modern multicore architectures.

Keywords: Energy Efficiency · HPC · Programming Languages.

1 Introduction

To raise awareness for energy efficiency in High Performance Computing (HPC), the Green500 list [3] ranks installed hardware by Gigaflops per Watt.

Here, we focus on the software side and investigate the influence of the selection of a programming language on the energy efficiency of an implementation. Our study is inspired by the research of Pereira et al. [7], which investigated the correlation between runtime and energy efficiency for 10 benchmarks from the Computer Language Benchmarks Game (CLBG)[4]. The results of their study

[3] https://top500.org/lists/green500/

[4] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

demonstrated that the fastest programming language, C, unsurprisingly, was also the most energy-efficient implementation for the selected benchmarks [7]. Expansion of their research to the educational examples provided by the Rosetta Code [8][5] considered more than 20 programming languages. However, most of them are hardly relevant for high-performance computing (HPC), like for example Ada, Erlang, PHP, or Lua. Conversely, Julia is missing as a self-branded high-performance language with a growing number of users. With native support for threads, distributed computing, and GPU offloading, Julia appears as a promising candidate for HPC applications. However, it does just-in-time (JIT) compilation for a virtual machine, which can lead to additional overhead compared to languages which are pre-compiled into native machine code.

Furthermore, the majority of the CLBG benchmarks are not representative in terms of HPC. For instance, there are benchmarks that merely allocate memory or concatenate strings. The lack of compute-bound benchmarks hardly allows drawing conclusions about the energy efficiency of programming languages in the HPC context. To fill this gap, we initiate the HPC Benchmark Game with a focus on HPC-relevant languages and benchmarks. Since it is not the language syntax and semantics that are relevant for the runtime and energy demand of an application, but the language universe consisting of compilers, runtime environment, and available libraries for parallelization, we aim to investigate different research questions. Observing the rising popularity of Julia in the HPC community, we want to clarify how far an interpreter based language can compete with a natively-compiled language **(RQ 1)** in terms of energy efficiency. Further we investigate the impact of the utilized compiler **(RQ 2)**, and finally, facing modern NUMA manycores, also the impact of the hardware architecture and the impact of over-threading **(RQ 3)**. The parameter space is huge, therefore we restrict us to these RQs. Here, we do not consider the relation between memory and energy consumption. It was already shown that for compiled languages the energy consumption of the processor dominates by 89 % [8]. Furthermore, the impact of vector instruction widths and the impact of platform-specific OpenMP thread mappings are not considered. We also do not include the influence of different problem sizes on over-threading here.

With those parameters in mind, we laid a starting point with four HPC-relevant programming languages: C, C++, Fortran and Julia. We do not consider Python, since Pereira et al. have already shown the inefficiency of pure Python (rank 26 of 27 languages [8]). And while most Python programs depend on libraries written in C or C++, and despite its popularity, the Python interpreter produces significant overhead. For each language, we have chosen a representative benchmark written by a programmer with experiences in this language. These reference benchmarks were re-implemented in the other languages under consideration.

The contributions of this paper are: (1) Proposing a benchmark suite that enables the comparison of programming languages and compilers regarding runtime

[5] http://rosettacode.org/

and energy efficiency for applications from the HPC field[6]. (2) Demonstrating the usage of the benchmark suite by presenting runtime and energy consumption results on a 128-core compute node. This gives us the means to answer the research questions RQ 1 - RQ 3.

2 Related Work

Since energy is an important resource in our modern society, researchers started to also take the energy demand of applications into account [11,10,12,7,6,8,4]. Already a quarter of a century ago in 2001, Valluri and John [11] evaluated how the existing compiler optimizations -O1 to -O4 influence energy consumption.

An early work in the HPC field investigated the runtime behavior and energy demand for the OpenMP implementation of the data-intensive NAS benchmark Datacube (DC), written in C, on two hardware platforms, a 12-core and a 16-core machine [10]. They varied the number of OpenMP threads and also investigated the influence of compilers. Their most interesting result is the negative impact of *over-threading* on runtime and energy. The best runtime and energy efficiency is achieved always by *not* using all possible cores. The authors explain this behavior with the rising excessive data movements throughout the memory hierarchy which occur when the number of threads is increased. Hence, to benefit from parallelization, the degree of parallelism which is used has to fit to the given algorithm, architecture and problem size.

The Computer Language Benchmark Games is a collection of 13 synthetic benchmarks which are rarely representative for HPC applications (an exception may be the n-body problem), and the submitted implementations differ greatly in their quality and their optimization effort: *"Some of these programs are high-level and some are handwritten vector instructions."* (see CLBG webpage) This also applies regarding parallelization which is not done in a uniform manner: some use vector instructions, some use multithreading and others run only sequential. Therefore, we have selected algorithms for the HPC Benchmark Game which come from the HPC field, and which we parallelized using the same approach.

In 2017, a team of researchers published results taking 10 benchmarks from the CLBG suite to compare runtime, energy, and memory consumption [7] for 27 programming languages. They followed the CLBG instructions for compiler versions and options[7]. As expected, the compiled languages performed much better regarding runtime and energy efficiency than the interpreted languages. For example, on average, compiled languages consumed 120 J to execute the solutions, while for interpreted languages this value was 2365 J which means about 20 times higher. Later, the group extended their investigations to applications from the Rosetta Code repository [8] with similar results. While so far the energy was estimated in software using an energy model based on RAPL, in [4] the group selected 14 programming languages and compared their prior

[6] The benchmark suite is published as open-source [5].

[7] A discussion of the results can also be found on the CLBG webpage: https://benchmarksgame-team.pages.debian.net/benchmarksgame/energy-efficiency.html

results against energy measurements with a special hardware device. The authors conclude that no significant differences are obtained in the rankings based on hardware measurements and based on software estimations. In contrast, our HPC benchmark game reflects the special conditions from the HPC field by considering only languages important for HPC, different benchmarks, as well as different compiler versions installed on the HPC cluster, and also the degree of parallelism.

In 2013, Yuki and Rajopadhye presented a high-level model of power consumption. They derive that when idle power is comparable to the dynamic power consumption, using dynamic voltage and frequency scaling (DVFS) to trade speed with energy efficiency is not possible. Furthermore, optimizing for performance is also beneficial for the energy consumption [12]. While the work of Pereira et al. shows that on average there is a strong correlation between runtime and energy consumption (see Table 5 in [8]), some studies come to the result that faster programs are not always the ones that consume less energy [10,6]. There remains the influence of the chosen algorithm and the selected data structures within the implementation.

3 HPC Benchmark Game

To compare programming languages, identical algorithms must be implemented in each language. An ideal benchmark would be a real-world application, however, the effort required to translate large, real-world applications into multiple languages is time-consuming and often impractical. Therefore, this study uses algorithms, like for example FFT, and computation patterns that are widely used in HPC applications. The selection criteria for the benchmarks were: (1) The benchmark must be based on a non-trivial mathematical algorithm (no string or memory manipulation only). (2) The benchmark must be parallelized with shared memory only. (3) The benchmark must be less than 2000 lines of code.

For each of the four programming languages, we selected a benchmark with a given implementation from an experienced programmer in that language. This benchmark then had to be ported to the other three languages.

The representative C benchmark is the Cellular Automaton (CA) which is a simple 9-point stencil application, based on Conway's Game of Life, implemented by a PhD student of our group[8].

For C++, we choose `TUG`[9] from the geoscience field, which solves diffusion problems on uniform grids with an alternating direction implicit method. The time step is divided into two half-steps. In each, a tridiagonal system is solved for each grid row and column using the Thomas algorithm (a form of Gaussian elimination). Due to its sequential nature (backward and forward substitution), SIMD operations and parallelization are not possible while solving the tridiagonal system. Instead, parallelization is achieved by using multiple threads to construct and solve these linear systems independently for each row and column.

[8] https://www.cs.uni-potsdam.de/bs/research/labsCa.html
[9] https://git.gfz-potsdam.de/naaice/tug

The NAS Parallel Benchmarks[10] were the source of the Fortran reference benchmark. We selected the Fast Fourier Transformation written in Fortran. We removed the internal validation algorithm. Instead, the result is written to stdout for offline validation.

Lastly, we selected the SRAD implementation from the JuliaParallel group as reference implementation in Julia[11]. SRAD belongs to the Rodinia Suite of the University of Virginia which consists of 23 HPC benchmarks originally written in C. SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations and is used to remove locally correlated noise. An overview of the selected benchmarks is given in Table 1.

Table 1. Overview of the selected benchmarks.

Benchmark	**Original Impl.**	**LOC**	**Data type**
Cellular Automaton	C99	297	Integer
TUG	C++17	1515	Floating-point
NAS FT	Fortran 90	694	Floating-Point
Rodinia SRAD	Julia	112	Floating-point

The reimplementation was conducted in an idiomatic manner, whereby language-specific features were utilized and no approach akin to "C with a C++ compiler" was employed. Additionally, several changes have been made to ensure consistent results across implementations, e.g. standardizing the random number generator to `rand` for integers resp. `drand48` for floats for all implementations [9].

For C++, the versatile and fast Eigen[12] library is used for matrix and vector operations. By default, Eigen and the Julia runtime have bounds checking enabled, which is disabled to achieve comparable performance to the C and Fortran implementations by compiling with `-DNDEBUG` or using `@inbounds` macros in Julia. For C, plain arrays are used.

To speed up the loading of the just-in-time compiled Julia code, we created a precompiled package for each benchmark. Additionally, the `PackageCompiler` package was used to create a bundled executable or "app". In the following, we refer to these types as `julia-pkg` and `julia-app` respectively.

Like the original reference implementations, for parallelization, we used OpenMP for C, C++, and Fortran. In case of Julia, we used macros from the `FLoops`[13] package, since the `Base.Threads` module does not provide complex operations such as shared variable reduction.

To validate the correctness of the implementations, all outputs of the benchmark implementations were compared to the output of the reference implementation.

[10] https://www.nas.nasa.gov/software/npb.html
[11] https://github.com/JuliaParallel/rodinia
[12] https://eigen.tuxfamily.org/
[13] https://github.com/JuliaFolds2/FLoops.jl

4 Experiments and Results

This section presents the outcomes of our experiments on an HPC cluster node. The measured metrics are runtime and energy consumption. Furthermore, the influence of the different compilers available on the HPC system is examined.

4.1 Description of the Testbed

Each compute node has two AMD EPYC 9554 "Genoa" CPUs, i.e. 2*64 CPU cores per node, in "zen4" architecture, with 3.1 GHz base clock and up to 3.75 GHz turbo clock. There are 6 GB RAM per core, thus a total of 768 GB DDR5 RAM at 4800 MT/s per node. 8 cores share a 32 MB Level-3 cache, and 16 cores share 3 memory channels. The zen4 cores have two 256-bit vector units (AVX2). These can be combined and used as one 512-bit vector unit (AVX512). The operating system is "Red Hat Enterprise Linux 8.6 (Ootpa)" with kernel 4.18.0-372.32.1.el8_6.x86_64. Since we are interested in the impact of the software universe regarding energy efficiency, we did not investigate energy savings by frequency scaling [2] and used the default installed Linux governor.

We use four different compiler suites. Two generations of Intel compilers are most widely used on the PIK HPC cluster. Since the testbed has AMD processors installed, we also use the AMD Optimizing Compilers (AOCC). For comparison and as a baseline, we added the GNU compiler suite. For each compiler suite, the "highest" available platform-specific optimization settings are listed below:

1. Intel Compiler Classic oneAPI 2023.2.0: **icc**/**icpc**/**ifort** identify themselves as version 2021.10.0. These are based on Intel's proprietary technology. They can generate code only for AVX2 platforms, thus the common options are `-O3 -ipo -march=core-avx2 -mtune=core-avx2 -DNDEBUG -qopenmp`
2. Intel oneAPI 2024.2.1: **icx**/**icpx**/**ifx** identify themselves as version 2024.2.1 and are based on LLVM. Since the Fortran compiler ifx cannot optimize for the zen4 platform, the common options are `-O3 -flto -march=core-avx2 -mtune=core-avx2 -DNDEBUG -qopenmp`
3. AOCC version 4.2.0: **aocc-clang**/**clang**++/**flang** identify themselves as version 16.0.3, and are like the new Intel compiler suite based on LLVM; `-O3 -flto -march=znver4 -mtune=znver4 -DNDEBUG -fopenmp`
4. The GNU Compiler Collection version 14.1.0: **gcc**/**g**++/**gfortran**; compiling with `-O3 -flto -march=znver4 -mtune=znver4 -DNDEBUG -fopenmp`

Further we use Julia version 1.11.0 with option `-C znver4 -O3 -check-bounds="no"`.

Power consumption and runtime were measured using the EMA framework [1]. EMA reads RAPL counters and reports the power consumption in micro joules. On the AMD CPUs of the testbed, only the `core` and `package` domains are available, while there is no report for the `dram` domain. Thus, only the power consumption of the entire CPU is reported which is a lower estimate for the real energy consumption.

To keep the parameter space for this study manageable, we show only results of runs where all 128 cores of one compute node are used. At runtime, we specify

the number of threads and set the per-thread stack size to 1.5 GB, which is necessary for local array variables in Fortran procedures. Further, "spread" is specified as thread affinity, in an attempt to spread the threads evenly across the memory access channels of the hardware. Each experiment is repeated 5 times, with a cool-down interval of 60 seconds before each run.

4.2 Compiled and Julia implementations (RQ 1)

The first thing to notice in Figure 1 is that energy consumption depends roughly linear on run time for the selected HPC benchmarks. For the TUG benchmark, we observe a deviation where the gcc and g++ experiments show a longer runtime, but a similar energy consumption as the faster icpx and ifort runs. And while gcc, g++ and aocc-clang++ yield similar runtimes for TUG, the aocc-clang++ version consumes more energy.

The second thing is that the native-compiled implementations are generally much faster and also much more energy-efficient than the implementations in Julia. Although we tried to eliminate the JIT compilation overhead by creating an app, there was an increase in runtime and power consumption in three out of four benchmarks compared to Julia's packaged approach. Thus, combined with the long compilation times caused by building an app, there is no observable advantage to using the `julia-app` approach over `julia-pkg`.

Figure 2 zooms into the same results for the Cellular Automaton and the SRAD benchmarks without Julia to better show the differences between the native-compiled implementations. This closer look reveals that there is a gap between the fastest and most energy-efficient implementations and the midfield. While the fastest implementations are always also the most energy-efficient ones, there are several slower implementations with different energy behaviors, e.g. compare the results of the SRAD C++ implementations (g++ versus icpx).

4.3 Runtime Results for Compiled Languages

Figure 3 shows the median runtime of the four benchmarks for the native-compiled languages and the considered compilers.

For the Cellular Automaton, the fastest and most energy-efficient implementation is the original reference implementation in C compiled with the newer Intel compiler (icx). In case of NAS FT, the original reference implementation in Fortran is slower than the C implementation (with icc and icx) and the C++ implementation (with icpx). For SRAD, the situation is similar, where the older Intel compiler (icc) beats the new one with 2.2 seconds versus 2.8 seconds. Finally, the original C++ implementation of TUG performs best with the two Intel compilers (icpc, icpx), but the Fortran implementation is comparably fast (again with the two Intel compilers ifort and ifx). The fastest and most energy-efficient implementations for each of the four benchmarks are listed in Table 2 and show that C is dominant in terms of performance.

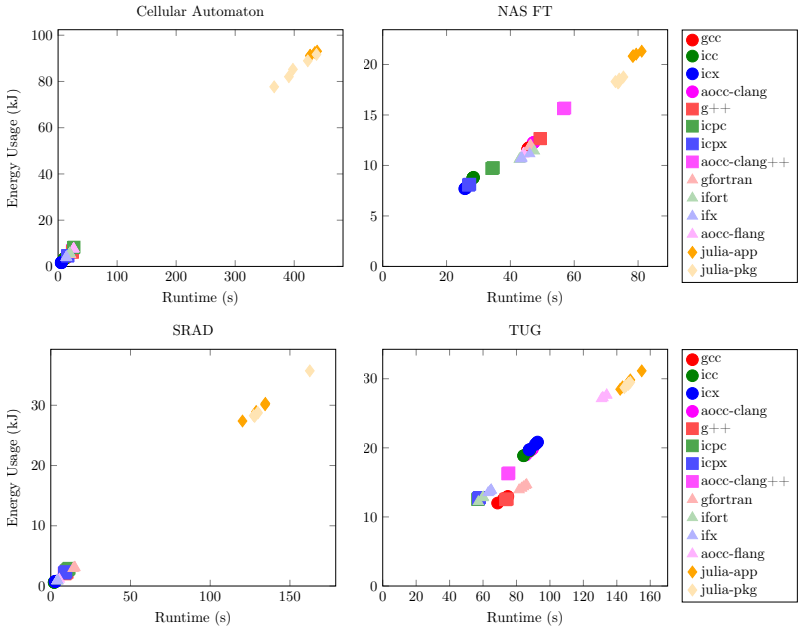

Fig. 1. Scatterplots of energy consumption vs. runtime. For each of the 4 benchmarks, results of 5 repeated measurements are shown.

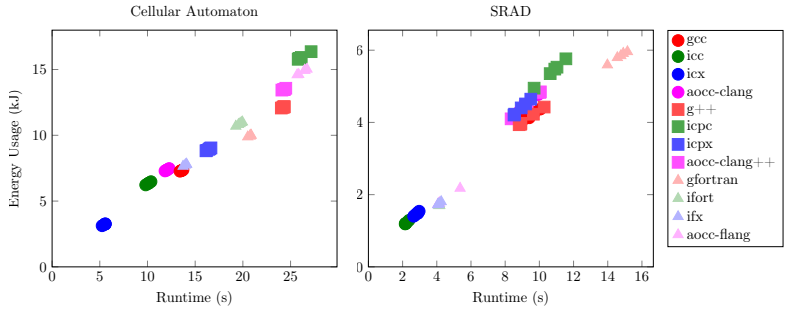

Fig. 2. Scatterplots of energy consumption vs. runtime for Cellular Automaton and SRAD, for the native-compiled implementations only.

4.4 The Impact of the Compiler (RQ 2)

The impact of the compiler is highly application dependent. Figure 4 shows the runtimes for each benchmark, normalized to the fastest combination of language and compiler.

We see different trends for SRAD and CA versus NAS-FT and TUG. While some SRAD and CA implementations are more than four times slower, the variance of the NAS-FT and TUG measurements is much smaller, and the slowdown

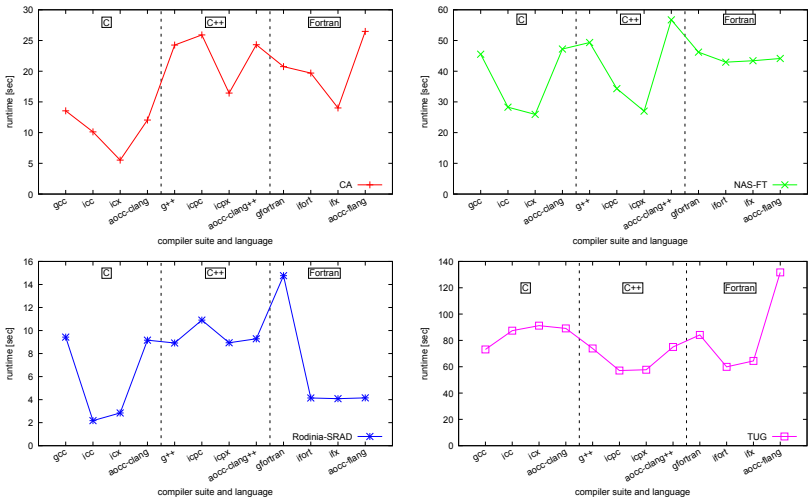

Fig. 3. Runtimes of the four benchmarks for the native-compiled implementations.

Table 2. Fastest and most energy-efficient implementations.

Benchmark	**Winner**
CA	C/icx
NAS FT	C/icx, C/icc, C++/icpx
SRAD	C/icc, C/icx
TUG	C++/icpc, C++/icpx, Fortran/ifort, Fortran/ifx

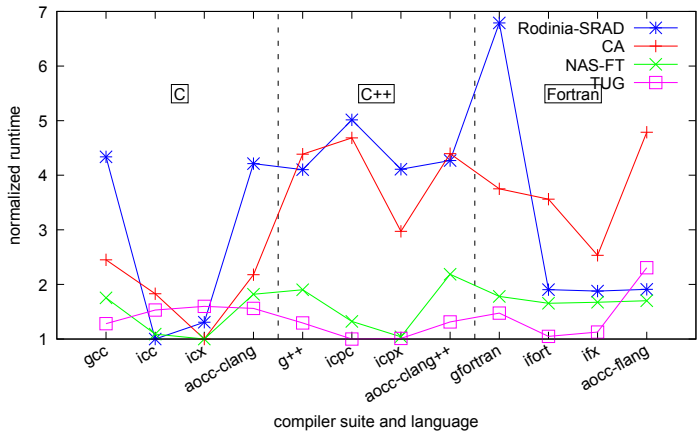

Fig. 4. Runtimes for the native-compiled implementations of the four benchmarks, each benchmark normalized to its **fastest** combination of language and compiler.

is almost always less than two (exceptions are the aocc-clang++ NAS-FT run and the aocc-flang TUG run).

In the code generated by the GNU and AOCC compilers, we find both AVX2 and AVX512 instructions. In our measurements, the AVX2 optimization implemented in the Intel compilers for both C and C++ yields better performance than the "zen4" optimization provided by GNU and AMD compilers.

Across all benchmarks, we also observe that the choice of compiler can have a more significant impact on performance than the choice of the programming language itself. For example, in the Cellular Automaton benchmark, while C implementations consistently had the fastest execution times compared to their C++ and Fortran counterparts, the slowest C implementation (gcc, 13.6 seconds) had a runtime comparable to the fastest Fortran execution (ifx, 14 seconds). For NAS-FT, the runtimes achieved by the Fortran compilers appeared to be consistently stable. Given NAS-FT's popularity as a Fortran benchmark, it is plausible that compiler developers prioritize it for Fortran compiler optimizations. For C and C++, however, the performance results are highly dependent on the specific compiler used. While the fastest C and Fortran compilers yielded comparable results for SRAD, switching to the GNU compiler resulted in a significant runtime slowdown, by a factor of 4.3 for C and 6.8 for Fortran. Similarly, using the AMD Fortran compiler for TUG resulted in 2.3 times longer runs.

4.5 The Impact of Hardware / Over-Threading (RQ 3)

All results presented so far used all 128 cores of the node. Since over-threading is a known problem [10], we also investigated this problem for the four benchmarks on the given architecture. All implementations are parallelized by adding simple OpenMP pragmas before the loop that shall be parallelized. We did not provide any compiler hints for load distribution.

Table 3. Scaling results for TUG for 1 up to 128 Threads showing medians.

# **Threads**	1	2	4	8	16	32	64	128
Runtime [s]	174	122	87	70	62	59.1	**58.9**	59.6
Energy [kJ]	56	40	29	23	21	21	22	25

While we observed runtime improvements for CA, SRAD and NAS-FT, this was not the case for TUG (see Table 3). The runtime decreases up to 64 threads, but then even increases again. This over-threading effect goes along with a higher energy consumption (25 kJ for 128 threads versus 22 kJ for 64 threads). When only 32 threads instead of 128 are used, a comparable runtime is achieved, but more than 17 % energy can be saved. That illustrates that the effort for finding an optimal mapping of the application also pays out in energy efficiency.

4.6 Ranking of Languages across all Benchmarks

Inspired by Table 4 in [8], we compute a global ranking over all 4 benchmarks. First, we normalize the runtimes for each benchmark against the runtime of the fastest language-compiler combination (as for Fig. 4). Then we use the geometric

mean to average the normalized runtimes over all benchmarks [3] In this way, every benchmark has the same weight in the global result. We proceed in the same way with energy.

The global results are shown in Table 4. Across all benchmarks, the analysis shows that no single programming language can be declared the overall winner. In fact, it is worth noting that the top five and bottom two language-compiler combinations are identical, regardless of whether runtime or energy consumption is considered. Table 4 confirms our observation that the compiler may have a more significant impact on performance results than the choice of programming language itself. While C is the winner with the icx compiler, it is beaten by Fortran and C++ when the GNU or AMD C compiler is used. Investigation whether platform-specific OpenMP thread mappings contribute to the observed performance characteristics is beyond the scope of this paper.

Table 4. Ranking of global results.

Normalized Runtime		Normalized Energy	
C/icx	1.20	C/icx	1.20
C/icc	1.32	C/icc	1.37
Fortran/ifx	1.73	Fortran/ifx	1.51
Fortran/ifort	1.85	Fortran/ifort	1.60
C++/icpx	1.89	C++/icpx	1.81
C++/g++	2.11	C/gcc	1.85
C/gcc	2.21	C++/g++	2.11
C/aocc-clang	2.26	C/aocc-clang	2.14
C++/aocc-clang++	2.26	C++/aocc-clang++	2.14
C++/icpc	2.36	Fortran/aocc-flang	2.15
Fortran/aocc-flang	2.45	Fortran/gfortran	2.26
Fortran/gfortran	2.86	C++/icpc	2.30
Julia/pkg	13.32	Julia/pkg	10.66
Julia/app	13.74	Julia/app	11.23

5 Conclusion

We presented the HPC Benchmark Game to compare the popular programming languages C, C++, Fortran and the more recent Julia regarding energy efficiency. The majority of the benchmarks are related to the authors' working fields, but this set can be extended. The presented results on a 128-core node confirm a positive correlation between runtime and energy consumption. Hence, optimizing for speed also reduces the energy consumption. Further, Julia cannot compete with the native-compiled languages for the HPC benchmarks. But the world of applications and compilers remains complex and colorful: First, the impact of the compiler significantly depends on the application and on the target platform, and further the influence of the compiler may be much higher than the influence of the chosen language. We find no single "winner" compiler, and thus want

to make users aware that it pays out to try different combinations for their production environment. Further, we show the negative effect of over-threading on energy-efficiency.

Acknowledgments. The authors gratefully acknowledge the Ministry of Research, Science and Culture (MWFK) of Land Brandenburg for supporting this project by providing resources on the high performance computer system at the Potsdam Institute for Climate Impact Research. Furthermore, the authors thank Klemens Kittan for his technical support.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Christgau, S., et al.: On the Usability and Energy Efficiency of High-Level Synthesis for FPGA-based Network-Attached Accelerators. In: 2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 886–895. Milano, Italy (2025). https://doi.org/10.1109/IPDPSW66978.2025.00139
2. Fan, K., et al.: SYnergy: Fine-grained Energy-Efficient Heterogeneous Computing for Scalable Energy Saving. In: SC '23. ACM (2023). https://doi.org/10.1145/3581784.3607055
3. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: the correct way to summarize benchmark results. Commun. ACM **29**(3), 218–221 (1986). https://doi.org/10.1145/5666.5673
4. Gordillo, A., et al.: Programming languages ranking based on energy measurements. Software Quality Journal **32**(4) (2024). https://doi.org/10.1007/s11219-024-09690-4
5. Lübke, M., et al.: HPC Benchmark Game Source Code and Results (PECS 2025). [code] Zenodo (2025). https://doi.org/10.5281/zenodo.15785010
6. Oliveira, W., et al.: A study on the energy consumption of Android app development approaches. In: IEEE/ACM 14th International Conference on Mining Software Repositories. pp. 42–52 (2017). https://doi.org/10.1109/MSR.2017.66
7. Pereira, R., et al.: Energy efficiency across programming languages: How do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. pp. 256–267 (2017). https://doi.org/10.1145/3136014.3136031
8. Pereira, R., et al.: Ranking programming languages by energy efficiency. Science of Computer Programming **205** (2021). https://doi.org/10.1016/j.scico.2021.102609
9. Stoll, D.: Performance und Energiebedarf von HPC-Anwendungen in Abhängigkeit von der gewählten Programmiersprache. Project Report, University of Potsdam (2024), https://www.cs.uni-potsdam.de/bs/teaching/docs/lectures/2024/stoll.pdf
10. Trefethen, A.E., Thiyagalingam, J.: Energy-aware software: Challenges, opportunities and strategies. Journal of Computational Science **4**(6), 444–449 (2013). https://doi.org/10.1016/j.jocs.2013.01.005
11. Valluri, M., John, L.K.: Is Compiling for Performance — Compiling for Power?, pp. 101–115 (2001). https://doi.org/10.1007/978-1-4757-3337-2_6
12. Yuki, T., Rajopadhye, S.: Folklore Confirmed: Compiling for Speed = Compiling for Energy. In: Languages and Compilers for Parallel Computing. pp. 169–184 (2014). https://doi.org/10.1007/978-3-319-09967-5_10