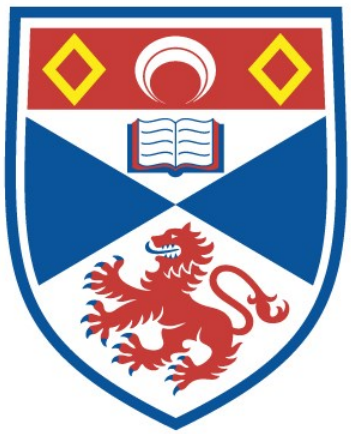
MODELLING ENERGY CONSUMPTION IN MULTI-CORE SYSTEMS USING META-HEURISTICS AND STATISTICAL MODELLING

Yasir Alguwaifli

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

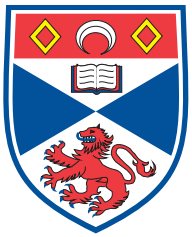


2023

Modelling Energy Consumption in Multi-Core Systems using Meta-Heuristics and Statistical Modelling

by

Yasir Alguwaifli



University
of
St Andrews

This thesis is submitted to the

University of St Andrews

in partial fulfilment for the degree of

Doctor of Philosophy

submitted on

2nd December 2022

Abstract

Controlling energy consumption has always been a necessity in many computing contexts as the resources that provide said energy is limited, be it a battery supplying power to an Single Board Computer (SBC)/System-on-a-Chip (SoC), an embedded system, a drone, a phone, or another low/limited energy device, or a large cluster of machines that process extensive computations requiring multiple resources, such as a Non-Uniform Memory Access (NUMA) system. The need to accurately predict the energy consumption of such devices is crucial in many fields. Furthermore, different types of languages, e.g. Haskell and C/C++, exhibit different behavioural properties, such as strict vs. lazy evaluation, garbage collection vs. manual memory management, and different parallel runtime behaviours. In addition most software developers do not write software with energy consumption as a goal, this is mostly due to the lack of generalised tooling to help them optimise and predict energy consumption of their software. Therefore, the need to predict energy consumption in a generalised way for different types of languages that do not rely on specific program properties is needed. We construct several statistical models based on parallel benchmarks using regression modelling such as Non-negative Least Squares (NNLS), Random Forests, and Lasso and Elastic-Net Regularized Generalized Linear Models (GLMNET) from two different programming paradigms, namely Haskell and C/C++. Furthermore, the assessment of the statistical models is made over a complete set of benchmarks that behave similarly in both Haskell and C/C++. In addition to assessing the statistical models, we develop meta-heuristic algorithms to predict the energy consumed in parallel benchmarks from Haskell's *Nofib* and C/C++'s Princeton Application Repository for Shared-Memory Computers (PARSEC) suites for a range of implementations in PThreads, OpenMP and Intel's Threading Building Blocks (TBB). The results show that benchmarks with high scalability and performance in parallel execution can have their energy consumption predicted and even optimised by selecting the best configuration for the desired results. We also observe that even in degraded performance benchmarks, high core count execution can still be predicted to the nearest configuration to produce the lowest energy sample. Additionally, the meta-heuristic technique can be employed using a language- and architecture-agnostic approach to energy consumption prediction rather than requiring hand-tuned models for specific architectures and/or benchmarks. Although meta-heuristic sampling provided acceptable levels of accuracy, the combination of the statistical model with the meta-heuristic algorithms proved to be challenging to optimise. Except for low

to medium accuracy levels for the Genetic algorithm, combining meta-heuristics demonstrated limited to poor accuracy.

Candidate's Declaration

I, Yasir Alguwaifli, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 40000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in January 2016.

I can confirm that no funding was received for this work.

Signature of Candidate:

Date: February 9, 2023

Supervisor's Declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Signature of Supervisor:

Date: February 9, 2023

Permission for Publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Yasir Alguwaifli, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

No embargo on print copy.

No embargo on electronic copy.

Signature of Candidate:

Date: February 9, 2023

Signature of Supervisor:

Date: February 9, 2023

Underpinning Research Data or Digital Outputs

I, Yasir Alguwaifli, understand that by declaring that I have original research data or digital outputs, I should make every effort in meeting the University's and research funders' requirements on the deposit and sharing of research data or research digital output.

Signature of Candidate:

Date: February 9, 2023

Permission for publication of underpinning research data or digital outputs

We understand that for any original research data or digital outputs which are deposited, we are giving permission for them to be made available for use in accordance with the requirements of the University and research funders, for the time being in force.
We also understand that the title and the description will be published, and that the underpinning research data or digital outputs will be electronically accessible for use in accordance with the license specified at the point of deposit, unless exempt by award of an embargo as requested below.

The following is an agreed request by candidate and supervisor regarding the publication of underpinning research data or digital outputs:

No embargo on underpinning research data or digital outputs.

Signature of Candidate:

Date: February 9, 2023

Signature of Supervisor:

Date: February 9, 2023

Acknowledgements

In the Name of Allah, the Beneficent, the Merciful

All praises to Allah for the aid and blessings in completing this thesis. I am truly grateful for all the experiences that I went through, leading to writing this thesis.

I would also like to extend my deepest gratitude to my wife, Mashael, who provided unparalleled support in some of the most challenging times.

I would also like to extend my deepest gratitude to my parents, Mohammed, Munira, brothers and sisters, Hana, Abdullah, Eman, and Yousef. You have always gone the extra mile to ensure my success in all life aspects. I would not be where I am today if it was not for your help. I am forever in debt.

I am deeply indebted to my supervisor Christopher Brown, who had the most instrumental role in making this thesis possible. Also special thanks to Adam Barwell, for the exceptional mentoring and guidance he provided through the years. Also I am grateful for the guidance and support of my second supervisor Edwin Brady

Also, I would like to thank my sponsor for the Saudi Arabian Ministry of Education represented by the Saudi Arabian Cultural Bureau and my employer, the University of Imam Abdulrahman Bin Faisal. The support I received from all parties in my journey finishing this thesis has been extraordinarily generous.

Yasir Alguwaifli
St Andrews
February 9, 2023

Contents

Acronyms

ACO Ant Colony Optimisation. 5, 91, 100, 107, 108, 117, 134, 135, 138, 139, 142, 143, 148, 150, 151, 155, 159, 165, 168, 171

API Application Programming Interface. 19

CPU Central Processing Unit. 2, 7, 42, 43, 59, 97, 184

D & C Divide and Conquer. 35

DSL Domain-specific Language. 2

DVFS Dynamic Voltage and Frequency Scaling. 4, 174, 178

GC Garbage Collector. 29, 30

GCC GNU Compiler Collection. 2

GHC Glasgow Haskell Compiler. xiv, 2, 7, 21–23, 27, 29, 30, 141, 177

GPU Graphics Processing Unit. 2, 178, 184

HEC Haskell Execution Context. 23, 27, 29, 30, 35

HJM Heath–Jarrow–Morton framework. 34

IPC Instructions Per Cycle. 92

ISA Instruction Set Architecture. 1, 2, 43, 59, 173, 182

MSR Model-specific Registers. 9, 22

NCG Native Code Generator. 21

NF Normal Form. 26, 37

NNLS Non-negative Least Squares. 4, 153–155, 157–160, 163–171, 182

NPU Neural Processing Unit. 2

NUMA Non-Uniform Memory Access. i, 43, 94, 178

OS Operating System. 29

PARSEC Princeton Application Repository for Shared-Memory Computers. i, 31, 33, 143

POSIX Portable Operating System Interface. xiii, 18, 33, 34, 68, 305

RAPL Running Average Power Limit. 9, 23, 41

RTS Runtime System. 7, 21–24, 184

SBC Single Board Computer. i, 9

SDE Software Development Emulator. 41–43, 61

SIMD Single Instruction Multiple Data. 31, 33, 59

SoC System-on-a-Chip. i, 182

SSE2 Streaming SIMD Extensions 2. 61

STM Software Transactional Memory. 21, 27, 28

TBB Intel Threading Building Blocks. 32–34

TDP Thermal Design Power. 7, 9

WHNF Weak-head Normal Form. 19, 20, 25, 26, 36

List of Theorems

Chapter 1

Introduction

This chapter introduces the challenges of energy consumption as it pertains to programming languages. Introductory Section 1.1 provides an overview and the rationale that makes energy consumption a relevant topic. Introductory Section 1.2 presents background on a few relevant programming language paradigms. Introductory Section 1.3 defines a set of questions that determines the scope of this thesis and lists the anticipated contributions.

When considering energy in the context of software, one must also consider various details affecting energy consumption. First, the long history of hardware evolution and advancements in processor manufacturing has affected software and many related hardware aspects. Then comes the lower layers of hardware development, such as the Instruction Set Architecture (ISA) and the numerous design decisions made by ISA developers to accommodate target platforms. Then, the operating system must be considered, as it manages the hardware and software resources. Atop these layers resides the application software.

Given the complexity of software–hardware architectures, predicting and monitoring energy consumption is challenging, especially as the multitude of developers responsible for the assemblage of software on a computer independently develop and optimise their codebases, often adding features that have different effects on the level of energy consumed.

1.1 Energy Consumption in Software

In software, we fundamentally program multiple ways of processing data at abstract levels using executable code running on specific types of processing hardware, such as Central Processing Unit (CPU)s, Graphics Processing Unit (GPU)s and more recently Neural Processing Unit (NPU)s. These types of processors have specific methods of computation, but the basic components remain similar across all models: they require an electrical source to operate. Every aspect of a microprocessor, such as logic and registers, require a direct low-voltage current to represent and transfer digital bits of information. Every gate switch consumes energy that can be measured in joules. Although that of a single gate operation is trivial, the billions typically found on modern units add up.

The next-higher aspect of the system includes the low-level ISA that runs on these processors. In essence, the ability to dynamically program a processor using specific instructions has given developers numerous methods with which to handle various computation scenarios via the ISA, which specifies a number of instructions that the processor understands and can execute to carry out computations, these are designed by CPU manufacturer like Intel, AMD, or ARM. ISAs come in different versions, but the most common ones are x86 and ARM, the former being used in Intel and AMD chips and the latter in ARM chips. In the past, developers used these instruction sets to write programs that are compiled at the binary level to be executed directly on the chips without an intermediary layer. Today, ISAs are mostly applied using high-level compilers, such as GNU Compiler Collection (GCC) [94] for C [49] and C++ [95] or GHC for Haskell [43]. Interpreters, such as Python [106], are also used. Virtual machines execute bytecode, e.g. Java [3] or Erlang [2], or instructions are simply translated from one language or Domain-specific Language (DSL) to another using compilers.

1.2 Imperative vs. Functional

When examining most programming languages' usage charts and statistics, they are found to share very similar properties as most languages are imperative in nature. The imperative programming paradigm is heavily used in industry; it was the first

to exist and was adopted from the outset to deploy some of the most iconic systems in the early days of the digital era. In the imperative style, developers use various concepts, such as global state, mutation, looping, branching and statements, and they apply different methods of describing the operational state as 'flavours'. Yet, the concepts remain the same; even if they employ a functional style, the overall implementation remains imperative.

A key feature adopted by imperative languages is strictness, which refers to specific ways of evaluating code before compiling it to remove potential errors that might arise during execution [20], e.g. divide-by-zero.

Functional paradigms apply different approaches, and the main concepts revolve around a function composition that enforces immutable rules, wherein a function cannot change any global state or carry out any input/output by itself, i.e. functional purity. This creates a strict rule-defined environment. Furthermore, if a function receives an argument, it should always return the same value to that argument, meaning that randomness must not occur. Some aspects of the imperative style exist in functional languages, but the main concepts of maintaining program flow differ. For example, the functional style relies on recursion instead of looping, which is important for the efficient execution of maths, such as functors, applicatives, monads etc. Most importantly, it differs in how a program is evaluated, there exist strictly evaluated languages in the functional space but the most notable languages use non-strict or lazy evaluation of program expressions.

1.3 Thesis Questions and Contributions

Modelling and analysing parallelism as it affects energy describes the heart of this thesis. To understand how the aims of this study can be approached, we must define a set of questions that, when answered, provides the outcome of this thesis.

Can parallel programs be statistically modelled according to energy use? An essential point of employing parallelism in programming is to achieve energy gains and benefit from extra hardware resources. To answer this research question, we must determine whether parallel programs can benefit statistical energy prediction.

Can the energy consumption of parallel computations be optimised or estimated, regardless of the implementation language (e.g. managed vs. unmanaged and lazy vs strict)? The goal is to examine parallel energy consumption from an execution-only perspective without considering language features. For example, a language like C++ has a strict evaluation method compared with Haskell, which is lazy by default. These concepts are further explained in the next chapter. A basic approach is to consider the chip design that executes the parallel code. However, when we look at the language features, e.g. scheduling and memory management, the answer to such a question might require more in-depth analysis.

What impacts do CPU clock frequencies have on program energy performance? The fact that a processor can run stably at the maximum clock frequency does not translate to having a requirement for such performance. Thus, chip manufacturers have create solutions, such as Dynamic Voltage and Frequency Scaling (DVFS), to manage varying computation requirements from a hardware design perspective.

1.3.1 Contributions

1. The development of a general optimisation technique to minimise energy consumption – (Chapter 4) Parallel programs can be demanding when optimising for performance, especially with a variety of modern multi-core hardware. Optimising energy consumption can be an order of magnitude more demanding because it also factors in performance. The thesis presents a technique based on regression models as well as a set of meta-heuristic algorithms that demonstrate stable accuracy when optimising energy across several programming languages, namely Haskell and C/C++.

2. Investigating energy modelling for parallel programs using multiple statistical models – (Chapter 3) The thesis delivers an analysis of multi-type regression models and evaluates the feasibility of using regression methods with language runtime features. The usability of statistical modelling is also examined for energy consumption prediction. This uses standard regression models such as Non-negative Least Squares (NNLS), Random Forests, and GLMNET and demonstrated on a variety of benchmarks for both Haskell and C/C++, showing its generality.

3. Development and formalisation of meta-heuristic algorithms to estimate and optimise energy – (Chapter 4) Optimising and estimating energy consumption can be complicated, even if sampling equipment is available. This thesis delivers three methods of energy sampling and probabilistic inferences of optimal program configurations for execution. This employs a number of standard meta-heuristic techniques such as Genetic algorithms, Ant Colony Optimisation (ACO), and Hill Climbing and demonstrated on a variety of benchmarks for both Haskell and C/C++ showing its generality.

4. Evaluation of energy consumption of multiple programming domains (functional and imperative) – (Chapters 5 to 7) The idea that languages perform similarly is not accurate, as languages like C might outperform many managed languages, such as Java. However, we desire to know whether their energy consumption profiles are similar. The thesis analyses energy consumption for two programming paradigms with different features via the extensive evaluation of standardised parallel benchmarks.

5. Set of Haskell/C++ energy profiles for PARSEC, Nofib, and the Computer Language Benchmarks Game – (Chapter 3) This thesis provides a complete profile set of parallel programs from three known benchmarking suites, Nofib, PARSEC, and the Computer Language Benchmarks Game, with energy consumption details at multiple frequencies and different workloads/inputs that highlight the energy patterns in each benchmark. In addition to evaluating several benchmarks that behave equally in Haskell and C/C++.

1.4 Thesis Structure

The structure of the thesis is as follows, Chapter 2 will explain details on the used benchmarks, types of parallelism, Haskell features, and tools used. Chapter 3 will provide an overview of the different benchmarks samples along with constructing statistical models and the application of given models on unseen benchmark datasets. Chapter 4 will present a group of meta-heuristic algorithms to optimise energy configurations in parallel programs. Chapters 5 to 7 will assess applying the meta-heuristic algorithms previously developed on various types of parallel programs, in addition to combining meta-heuristics with a statistical model to measure usability

of such method. Chapter 8 will give an overview of the contributions in the literature. Chapter 9 will iterate over the conclusions of this thesis.

1.5 Summary

In this chapter, we reviewed the different situations that make understanding energy consumption relevant to software and programming languages. Additionally, we identified hardware and software features and programming paradigms having distinctive effects on energy consumption.

Chapter 2

Background

This chapter discusses the various subjects and tools required to understand the details of energy consumption from a programming language perspective. Section 2.2 on page 9 explains basic parallelism hierarchy with existing implementations and examples. Section 2.4.3 on page 21 presents an overview of GHC, its internal runtime system and Runtime System (RTS) metrics. Section 2.4.5 on page 24 presents evaluation concepts for Haskell with examples, in addition to relevant parallelism concepts. Section 2.1 on page 7 described the tools used for sampling energy in Intel's x86. Section 2.5 on page 30 explains the target testbed and the sampling configuration.

2.1 Processor Energy and RAPL

In computer hardware design, power management is as essential, as it is for any other electric-based device. Minimising power and energy consumption is always paramount, as doing so improves portability and total operational costs. Notably, power and energy are different constructs. Power is measured in Watt, i.e. 1 watt per second, whereas energy is the total joules for a given device over time. In this case, CPUs have various Thermal Design Power (TDP)s. Although they indicate heat dissipation for a given chip, this can be used as a power measure for the chip. With the complexity of static and dynamic power usage in modern chips, it is not easy to define a specific range of power draw, as any chip's running state is dynamic.

Approaching energy consumption in software requires multiple components to be present to effectively and practically develop energy-aware software that can be reliably deployed. Holmbacka [41], demonstrated on a multicore ARM processor that the equation can represent CPU energy consumption over time: *Power = Static Power + Dynamic Power*, the static power being the power needed to run the CPU while idling, and the dynamic component refer to the frequency of the CPU cores when a computation is being executed on the processor. Although controlling schedulers at runtime can assist with applying a scheduling policy that aims to increase performance or reduce power draw as explained in [41], the entire program could have different stages of executions that behave differently, especially when benchmarking settings are changed, e.g. CPU architecture, Programming Language, the existence of a runtime component in the programming language.

Using instruction-level costs is essential to proper energy estimation, but it is sometimes overlooked. As the cost of energy is inherently affected by performance, i.e. the time is taken to execute a computation, as seen in Haskell's samples, modelling circuit switching may be the apparent answer; however, this is not the case. Morse *et al.* [74] demonstrated that attempting to model such operations at the bit level is an NP-hard problem, where the solution space is massively more significant than what machines are capable of searching. The authors first emphasised that energy does not directly correlate with time, despite how the energy equation is formulated. The power variable is a variant that changes over time. For example, an average power of `6.25` watts multiplied by `8` s would result in `50` J of total energy consumed. When examining the power sample of a processor running a computation, it would reflect a fixed amount equal to several bits being switched on and off; the beginning of the computation would have higher power consumption owing to transistors triggering circuit switches, i.e. bits being switched on and off. We can view program execution stages using the circuit-switching perspective. The authors discussed the possibility of finding the WCEC of a given program by analysing its switching instructions to find the combination of operands that trigger maximum energy consumption. This method is referred to as the *circuit-switching problem* (CSWP). Their approach also describes the *Hamming distance* effect, referring to the bits to be switched caused by an instruction being executed. For example, an `add` instruction used to operate on register, `R1 = 010`, would result in a Hamming distance of `1`. Finally, the authors demonstrated that the *CSWP* was like the *maximum satisfiability problem 2* (MAXSAT2),

which was proven to be an NP-hard problem [9].

Running Average Power Limit (RAPL) [45] is an Intel-specific tool that provides power management for Intel-based machines; it enables sampling from registers to calculate average energy consumption over a given interval. Its use is simple through the Model-specific Registers (MSR), which allows access to RAPL measurements. The MSR provides accuracy [87, 107] with minimal error, compared with electric wall sampling power readings for systems components, e.g. GPU and DIMMs. The same mechanism also allows energy consumption to be controlled/capped.

2.2 Parallelism Overview

Parallel execution gained attention early in computer science history. However, beforehand, chip manufacturers increased performance by increasing the number of transistors. Laws like Moore's law [73] stated that the number of transistors per chip area doubles in number every two years, which in turn reflects on the performance and the clock frequency of extant processors. Additionally, Dennard's Scaling [28] law which states that as transistor density increases, the power consumption for each transistor decreases. These laws would have remained valid [38] if the amount of energy required to power the increasing number transistors did not change significantly. However, power requirements turned out to be quadratic when doubling the number of transistors, meanings that the amount of heat dissipated always corresponds to power requirements. The emphasis on parallelism in mainstream machines was not always as crucial until CPU frequency and single-core performance hit a physical limit. That is, chip manufacturers were could not provide better clock performance. Hence, the industry applied different techniques, shifting towards multi-core and -thread architectures in which parallelism is introduced implicitly or explicitly. Implicit parallelism occurs without the user's knowledge; the machine runs specific computations in parallel. In the latter case, the user is required to explicitly manage threading. In all cases, the watts required to run the chips has grown. For example, the AMD Ryzen Threadripper 3990X [1] has a TDP of 280 watts, compared with the Intel Pentium Pro [46], which was a single-core CPU with a clock frequency of 200 MHz and a TDP of 44 watts. Even portable platforms, such as SBC and mobile phones, have moved to multiple cores with increasing energy requirements.

There are ways to solve specific problems when introducing parallelism [7, 12, 13, 47]; most notably, the automatic loop parallelisation in the Intel C++ compiler [72] uses techniques, such as static analysis. In its compilation phase, the compiler determines the sequential parts that are independent of each other, subsequently transforming them into multi-threaded code. However, there are challenges to this tactic that cannot be overcome without solving complicated problems. For example, not all computations processed in such code can be parallelised owing to data dependencies. Additionally, there are limitations to the how computations can be performed. High-level programming languages, for example, are complex in nature, making it difficult to match program properties to parallel execution opportunities. The data and program structure complexity can quickly limit what automatic parallelisation tools do. Thus, parallel libraries and tools have been deployed to help, e.g. OpenMP and TBB. These tools allow the introduction of parallelism in ways that allow libraries to parallelise specific parts of a program effectively without introducing unwanted behaviours.

2.2.1 Task Parallelism

Task or functional parallelism [85] has been introduced as a result of frequent observations of patterns of execution. Specific computations can run in parallel to compute streams of data independently from each other. In a task parallel execution, the developer implements an algorithm to identify functions that can run independently. The functions are implemented as streams executed in parallel alongside others, and the inner workings are abstracted. A basic structure of this type of parallelism is displayed in Figure 2.1. As shown, *parallel processing* encapsulating all tasks. The *Data/Input* is a series of values in a list or array that is fed to tasks in the next stage. Then, the functions compute their results independently and in parallel. The length of the arrows reflect the duration of each task; nevertheless, they all compute their values in parallel. The last phase, *computed output*, indicates that the parallel-processed data can be fed to another parallel stage or sequential step to save time and combine outputs. A sample implementation of the illustration in Figure 2.1 is found in Algorithm 1. Lines 1–4 define the arrays used to store the various computations; all arrays can store 101 elements. Line 7 defines a *Parallel Region* containing a group of parallel computations. Lines 8–19 define *Sections* with loops in each; the *Section* keyword is an indicator that the body of the section is to be executed in parallel alongside other *Sections*. The `for` loops in each *Section* carry out different

computations where the $\longleftarrow$ at the beginning of each loop designates the retrieval of an element at a specific position in an array, which is then stored in a temporary variable, `i`. The value stored in `i` is then modified according to the arithmetic operation defined, e.g. `i` $*$ 2 or `i` $*$ `i`, which is then stored at the same position as the retrieved value in the previous step. Each parallel *Section* is computed independently from other sections and without the need to wait to block executions at certain points. A more concrete implementation of Algorithm 1 using OpenMP is found in Appendix A.

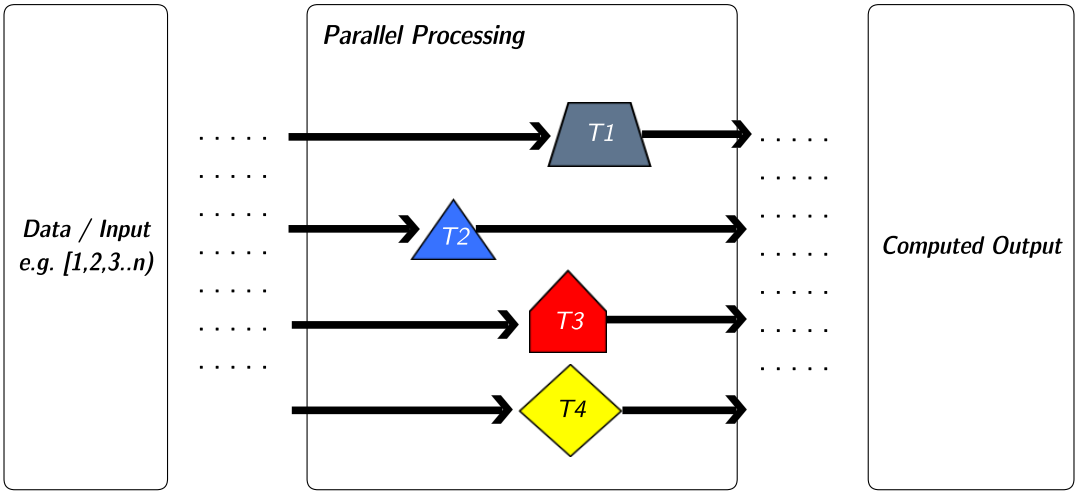


Figure 2.1: Diagram showing basic structure of task parallelism

2.2.2 Data Parallelism

Data parallelism is achieved by applying a function over a list of data, e.g. integers, strings, or other complex type). For example, to apply a function, f, over a sequence of data stored in an array of a given size, $[X_1, X_2, \ldots X_n]$, the parallel computation can be achieved independently by applying the function, $f(X_1)$, $f(X_2)$, $\ldots$, $f(X_n)$, in parallel. Considering Figure 2.2, the *Data/Input* is a sequence being processed by applying the function `f` over the elements of the sequence. The application of `f` is done in parallel by applying `f` over each element independently. Its simple implementation is demonstrated in Algorithm 2. Line 1 shows an assignment/initialisation of variable `length` with a value of 1M, which is used as the loop count. Lines 2 and 3 initialise two arrays of the same size, i.e. 1M. Finally, Line 5 defines a `Parallel For` indicator for the loop iteration, causing them to be computed in parallel. The `for` loop at Line 6 iterates from zero to the value of `length`, i.e. 1M. With each iteration, the value of `j`

Algorithm 1: Example of an algorithm computing tasks in parallel

```
1  numbers[101]
2  numbers2[101]
3  squares[101]
4  multBy2[101]
5  divisions[101]
6  for j ← 0 to 100 do
7  |   numbers[j] ⟵ j
8  Parallel Region:
9  Section:
10 for p ← 0 to 100 do
11 |   i ⟵ numbers[p]
12 |   squares[p] ⟵ i ∗ i
13 Section:
14 for p′ ← 0 to 100 do
15 |   i ⟵ numbers[p′]
16 |   multBy2[p′] ⟵ i ∗ 2
17 Section:
18 for p″ ← 0 to 100 do
19 |   i ⟵ numbers[p″]
20 |   numbers2[p″] ⟵ i/2
21 End Parallel Region
```

is assigned the value of the iteration. The body of the loop on Lines 7 and 8 begins by storing/assigning a computed value, `j + 3`, to the position, `array[j]`, followed by the retrieval of the `thread_id` value and the assignment of its value to the position, `thread_num[j]`, in the thread array to track the thread processing the computation. Furthermore, concrete implementation using the OpenMP library can be found in the listing in Appendix A.

Loop parallelism: Simple parallelism patterns improve upon low-level parallelism forms; in this case, the parallel execution of specific segments of the source code, such as parallel processing of loops, i.e. data parallelism. The management of this parallelisation is unlike low-level types, where the implementer must manage and address the details of parallel execution, such as sharing memory and other resources.

Algorithm 2: Example of Data Parallelism

```
1 length ⟵ 1000000
2 array[1000000]
3 thread_num[1000000]
4 Parallel For:
5 for j ← 0 to length do
6     array[j] ⟵ j + 3
7     thread_num[j] ⟵ thread_id
8 End Parallel For
```

This entails the primitives of parallelism that allow quick implementation without referring to tasks, such as scheduling, reserving and freeing memory. Parallel models, such as OpenMP and TBB, provide the building blocks to implement parallel segments. For example, sequential loop in Listing 1 on page 13 executes a for loop that runs for six iterations from zero to five (Line 5) and prints the value of the loop variable each time (line 5). It then exits the program with an integer value of zero. indicating that no errors occurred (Line 8). To implement a similar program in parallel using the TBB library, we use the code shown in Listing 2 on page 14, starting with the initialisation of the task scheduler that manages the execution and threading (Line 9). Then, instead of printing the value of the loop iterator, it instead prints the number of the thread that executed the given iteration (Line 14) using the function call, `this_tbb_thread::get_id()`, which is a TBB built-in function that retrieves the currently executing thread, and as the body of the loop executes in parallel, the print statement may print a different number each time. In this example, the scheduler decides that a single thread is faster in executing the loop as the number of iterations is small; however, if there were many loop iterations, more threads would be required.

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4     int i;
5     for(i = 0; i < 6; i++)
6         printf("Iteration = %d", i);
7     return 0;
8 }
```

Listing 1: Simple loop for printing numbers at each iteration

```
#include "tbb/tbb.h"
#include "tbb/parallel_for.h"
#include "tbb/task_scheduler_init.h"
#include <iostream>

using namespace tbb;

int main(int, char **) {
    task_scheduler_init init;
    parallel_for(
            blocked_range<size_t>(0, 6),
            [=](const blocked_range<size_t> &r) {
                for (size_t i = r.begin(); i < r.end(); ++i)
↪ {
                    printf("Thread Id: %d\n",
↪ this_tbb_thread::get_id());
                }
            }
    );
    return 0;
}
```

Listing 2: Example of a parallel_for that runs for a fixed number of iterations, printing the thread number from each iteration using TBB

2.2.3 Combining Parallel Skeletons

The various types of parallelism can be combined, managed and nested in many ways. Cole [21] introduced the concept of algorithmic skeletons, which are standard higher-order functions used to process different types of data. For example, a *map-reduce* makes use of two parallel skeletons to process a list-like structure. The *pipeline* skeleton typically processes task elements one after another. The primary approach to algorithmic skeletons is natural, as the definition of high-level implicit parallelism follows suit. Fragments of different types of parallelism can be combined to meet specific requirements.

Chis and Vélez [18] categorised skeletons into three types: data parallelism, task parallelism and resolution. The first two were discussed previously. The resolution category accommodates problem-specific parallelism, such as when computing the Fibonacci sequence in a recursive style, as shown in Figure 2.4. When invoking

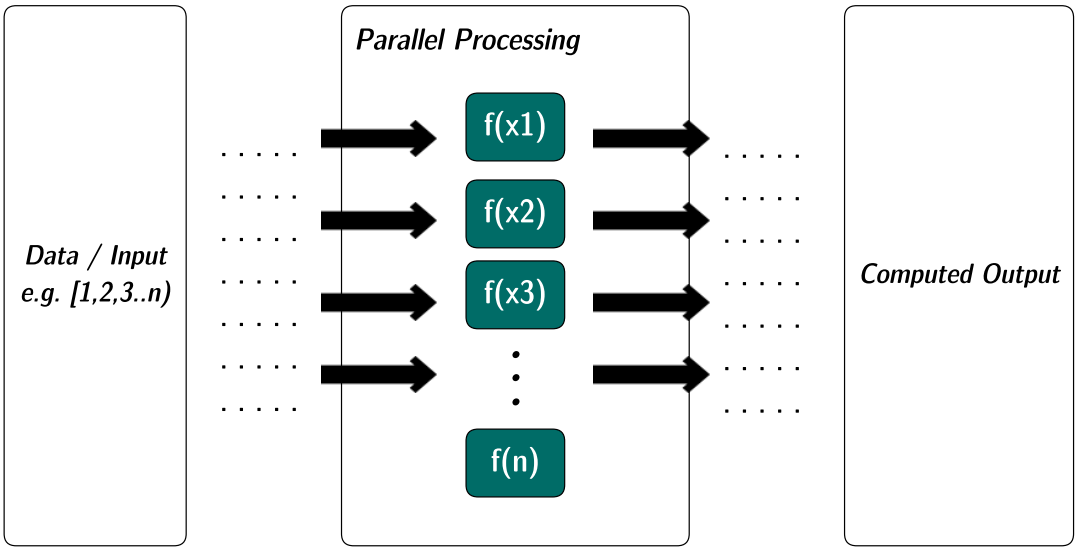


Figure 2.2: Diagram of basic data parallelism structure

the function, it begins by splitting the computation into two subsequent calls to the same Fibonacci function while adding the results from each recursive output. Applying a *divide and conquer* (DC) skeleton translates to a breakup or split stage. The computations are split into smaller chunks, and when a condition no longer holds, it merges or combines the results of each stage. *DC* skeletons can be used with searching or sorting algorithms, binary searches and merge-sorts.

There are limitless skeleton combinations. One example is shown in Figure 2.3, where a data-parallel stage checks whether a number is even, then a sequence of staged parallel tasks takes place one after the other. This final stage is referred to as a pipeline.

2.3 Parallelism Libraries: Pthreads, OpenMP and Intel TBB

To understand the reason why parallelism is a necessary and intricate part of modern programming and system design, we next review some parallelism libraries. In [83], the authors introduced a hierarchy of granularity for parallel computations, some of which are part of hardware-level parallelism; others are designed and used at a programming language level. The following sections describe the categories of

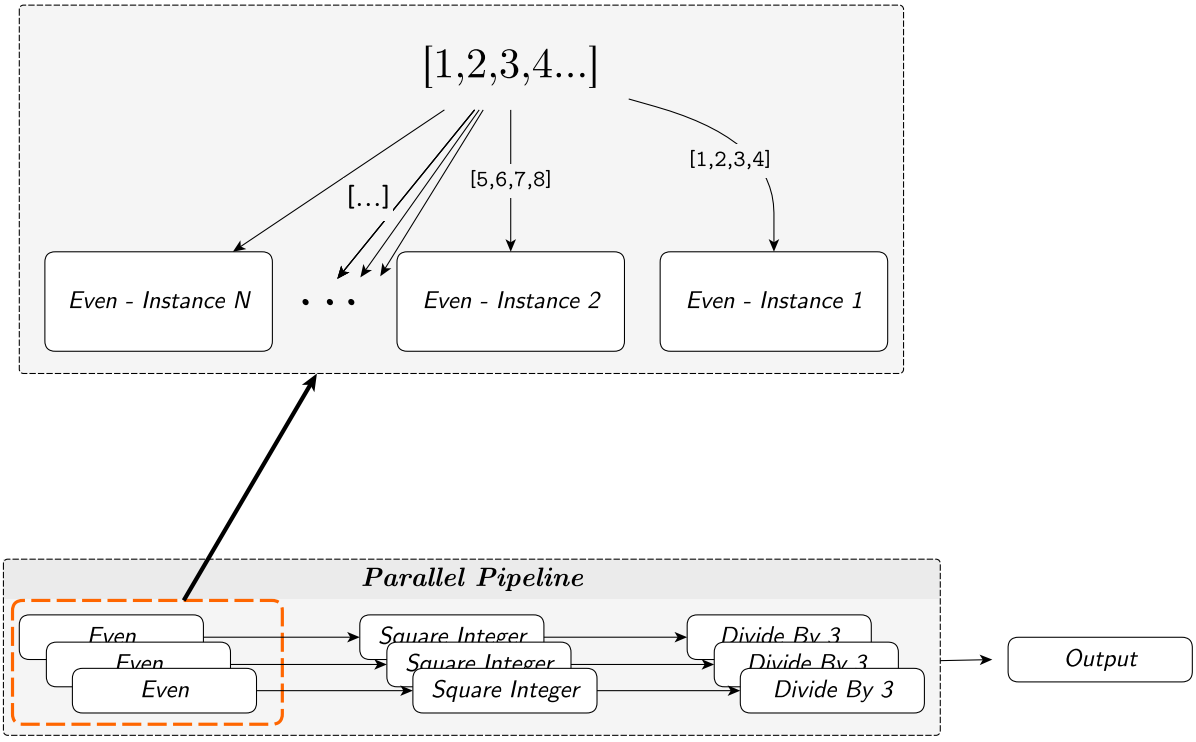


Figure 2.3: Example of combining types of parallelism

different parallel solutions and their granularity with examples of existing frameworks and libraries that implement the concepts.

2.3.1 TBB

Intel's TBB [58], released in 2006, provides a set of parallelism functions in a highly abstracted style, in which tasks are the central concept. For example, to run a parallel loop, a user must invoke a function call, `parallel_for`, with a set of parameters to specify the loop control variable that sets the beginning and end. Other examples of parallel task-based functions include `parallel_for_each`, `parallel_reduce` and `parallel_pipeline`. TBB allows the user to run the library regardless of the compiler used as the library is self-contained with scheduling, memory allocation and parallel functions. One of the disadvantages of TBB is its dependence on C++, unlike OpenMP, which supports multiple languages and has been implemented using many compilers.

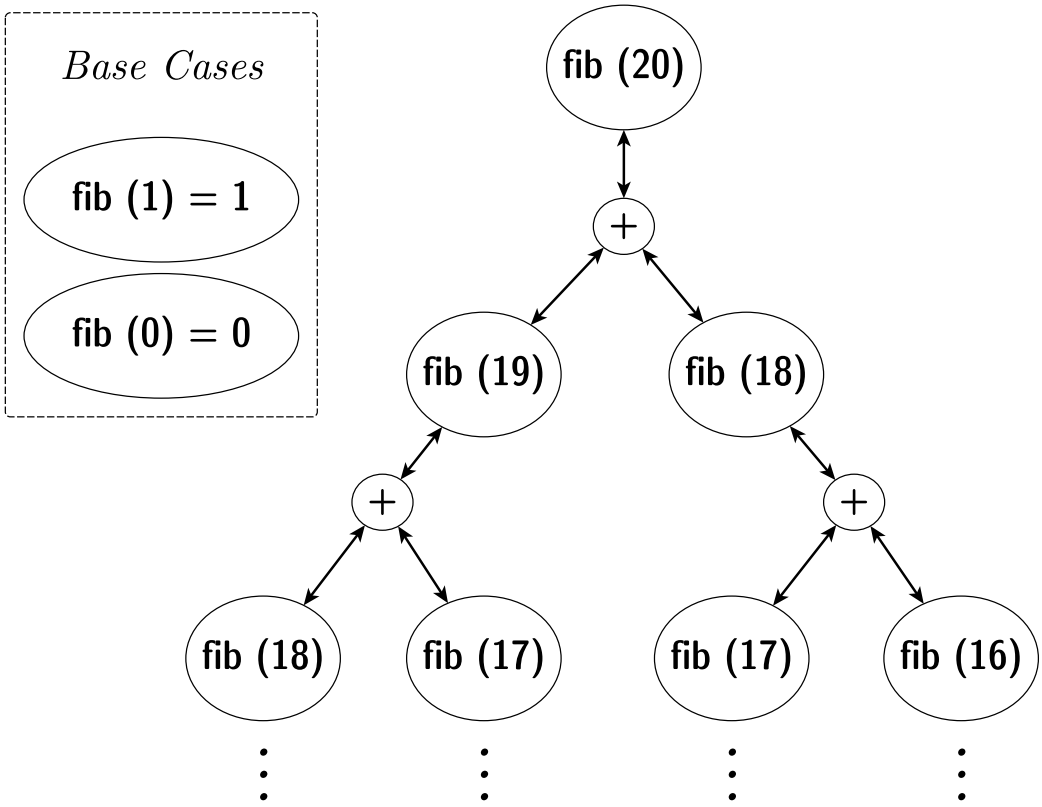


Figure 2.4: Example of recursive Fibonacci sequence-call tree

2.3.2 OpenMP

OpenMP [25] is a language extension which also can be considered a programming language in itself which was introduced by the *OpenMP ARB*, which released its first version for Fortran support in 1997, followed by C/C++ support the following year. Today, OpenMP has been adopted by many compilers, most notably the Gnu Compiler Collection (GCC) [101] and LLVM [59]. In C/C++, OpenMP allows parallel execution via a set of directives, functions and environment variables, which allow users of the language to implement parallel programs. However, complexity leads to buggy behaviour. Directives, i.e. `pragmas`, are used to introduce parallelism in OpenMP. A `pragma` has a set of pre-defined attributes, such as `parallel`, `for` and `sections` for *work-sharing*. It provides directives for controlling the accessibility of specific sections, such as `master`, `critical`, `atomic` and `ordered`. It also provides the ability to share variables and other program structures safely. For example, sharing a variable of a given type can be specified using the `shared(variable)` clause; likewise, the `private(variable)` clause provides the means of allocating a new copy of the vari-

able to each thread.

2.3.3 POSIX Threads

POSIX threads, i.e. Pthreads [75], are governed by the POSIX standard, which defines an interface-enabling portable application for different operating systems, e.g. Linux/Unix-based systems. It was initially established by IEEE in 1988, and another variant was released by ISO in 1990 under the name POSIX [44]. The ISO version is the most common implementation. Its interface enables hardware and software designers to develop standard solutions to allow portability. The initial POSIX release did not include threading, but later, Pthreads were introduced for use with C/C++ for parallelism. Pthreads allow the control of multiple execution contexts through a single process with data synchronisation. It also has its own set of *logical* registers, stack pointers, local variables and other features that make it uniquely powerful. Pthreads allow several parallelism implementations, some of which were applied internally to OpenMP and TBB. However, the POSIX standard does not ensure safe execution, bug-handling, or parallelism synchronisation; it relies on user design decisions. Therefore, when Pthreads are used directly, users must employ mechanisms, such as locking, execution control and synchronisation to manage safe execution. This is considered a low-level parallelism method, as the structure must be defined differently for each user requirement.

2.4 The Haskell Language

2.4.1 Introduction

Haskell is a functional language having characteristics different from common programming languages, such as C or C++. Lazy evaluation is one of its core features, allowing it to employ more programming languages concepts than most, such as *immutability*, which means that any variable/function created cannot be modified. Another important feature is the pure execution, wherein functions cannot modify global values or initiate input/output communications. The language is strict, rendered as such under the `main` function, which is the main entry for program execution.

In Haskell, a program may consist of multiple functions where each can be passed an argument or a set thereof. Consider the example in Listing 4 on page 22. The program defines three functions: `main`, `val` and `branch`. `main` is the main entry point where the program begins execution. `main` on Line 8 is defined as `putStrLn $ show $ branch 1` in the body. `purStrLn` is a built-in function for printing to standard output. `show` is another built-in function that takes a given type and returns a printable type. Finally, `branch 1` is a function call to the function, `branch`, with argument 1 being an integer type. The $ symbol denotes the sequencing operator to avoid rewriting functions in parenthetical format, e.g. `putStrLn (show (branch 1))`. Lines 1–3 define a function using *guarded statements/clauses*. `branch` has the signature type of `branch :: Int -> [Char]`, which describes the type of inputs and arguments that can be passed to the function and what it returns. In the case of `branch`, the function takes an integer type. In Haskell, a function always returns a value, which in the case of `branch`, is `[Char]`, which is a list type for characters, i.e. a string. Lines 2 and 3 are *guard clauses*, wherein the first one, `exitCode == 0 = "Success"` is equivalent to an if statement in imperative languages. The value of the input, `exitCode`, is compared to integer zero, and if true, the function returns the string, `"Success"`. The clause on Line 3 is a general one that compares any integer value to string `"Fail"`.

Finally, `val = "Hi"` returns a value only, meaning that it does not accept any input. Thus, when called, it will always return the value, `"Hi"`. Note that the type signature, `-- val :: [Char]` is commented-out by prefixing the signature with `--`. In Haskell, compilers can infer the type of the returning value from a function; thus, it is not required to defining functions as types, as errors will be thrown during compilation.

2.4.2 Evaluation Functions: `seq/pseq/rseq` and `par/rpar`

Haskell has mature and well-established parallelism Application Programming Interface (API)s. One essential API that is used to implement parallel constructs: the `Strategies` API. To understand how a computation is evaluated using `Strategies`, we must first understand its building blocks. `pseq` is defined as `pseq :: a → b → b`, which translates to the statement 'evaluate Weak-head Normal Form (WHNF), then return b', which controls the order of evaluation of `a` to WHNF. `b`, which is *strict* in `a`, ensures that the function will follow the sequential order of its arguments. There

is a subtle difference between `seq :: a → b → b` and `pseq :: a → b → b`. `seq` has the same type as `pseq`, but the way in which it evaluates its arguments can differ. In `seq`, the argument input is evaluated to WHNF, returning the second argument. However, if the first argument is already in WHNF, `seq` will skip evaluating the first one. On the other hand, `pseq` will compute using the same style, i.e. evaluating the first argument to WHNF, except that will always re-evaluate the first one, regardless of whether the first argument is in WHNF. Although `par` has the same signature as `pseq/seq`, it uses a completely different method of evaluating computations. `par` is an indicator of the runtime with which a given closure can be evaluated in parallel with another function, i.e. a closure. For example, `par` takes arguments `a` and `b` and allows `b` to be computed by the main program flow, meaning that the Spark (the computation to be evaluated) reflects the promise to carry out the computation as `b` is being computed, e.g. `a` is computed in parallel with `b`. The functions, `rpar` and `rseq`, are basic evaluation strategies from the `Strategies` library. `rpar` is another that displays similar behaviour as `par`, indicating that a computation can be evaluated in parallel. `rseq`, like `seq`, forces an argument to WHNF.

Another example is shown in Listing 3 on page 21. An implementation of the Fibonacci function on Line 1, `fib`, accompanies a guarded statement. The first guard clause, `n <= 1 = 1`, is a base case that returns a value of one if the argument of `n` has a value of one or less. The `otherwise` guard clause acts as the case for any `n` values greater than one. The part on Line 2, `f1 + (f2 ‘using‘ rpar)`, uses `rpar` to create a Spark of the recursive expression in `f2` on Line 5, which allows the evaluation of the computation in parallel with `f1`. The `where` clause contains two functions that recursively call `fib`. Figure-2.5 on page 21 visualises the evaluation tree. The top node of the tree `fib 8` is subsequently broken into two branches, `fib 6` and `fib 7`, and the recursive calls continue breaking down computations until the expression is evaluated by the guarded clause on Line 1, `n <= 1 = 1`. The ellipses circling some of the nodes, e.g. `fib 6`, indicates that some results can be shared as the functions computed in the same scope will have the same results. `fib 6` and `fib 5` will be computed once; therefore, the other computation is considered either *fizzled* or a *Dud*, as explained in 2.4.8 on page 26.

```
1 fib n | n <= 1    = 1
2       | otherwise = f1 + (f2 `using` rpar)
3                 where
4                  f1 = fib (n - 1)
5                  f2 = fib (n - 2)
6
7 main = putStrLn $ show $ fib 8
```

Listing 3: Basic parallel Fibonacci computation using the rpar evaluation strategy from the Strategies library

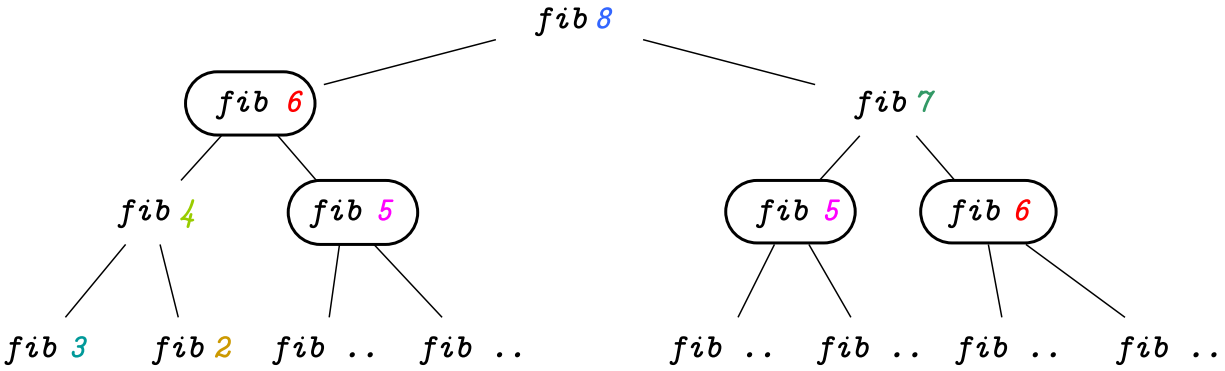


Figure 2.5: Partial visualisation of a recursive Fibonacci evaluation in tree form

2.4.3 GHC and RTS Feature Overview

The GHC [67] provides a vibrant ecosystem with some of the most advanced features in the field of programming. First, it provides several backends with which to compile. It uses Native Code Generator (NCG) but also can generate LLVM [59] and C [50] code. In addition to a sophisticated RTS, which contains a user-space implementation, a scheduling tool, a byte-code interpreter, a storage manager and Software Transactional Memory (STM), a profiling tool [37].

The runtime offers many metrics that provide insight into program execution. Section 2.4.4 below provides an example of the results obtained from running a Haskell program in parallel. The first five metrics correspond to initialisation, mutation, garbage collection, exiting and total time. Then, it shows initialisation and total execution time. Mutation reflects the code evaluation time. Then, it shows garbage collection time.

In sub-subection 2.4.4, we present the output from three processes: the executed Haskell program, RTS, and the C process that samples the overall energy consump-

tion. The first line is the Haskell process number, and the second is the process output. The RTS, which is part of GHC, spans Lines 3 to 28. The second part (Lines 29–37) is the output from the C program that samples energy using MSR and prints the details collected throughout. Sub-section 2.4.4 explains the various details of these metrics and how they are used for analysis.

```
1
2  branch :: Int -> [Char]
3  branch exitCode | exitCode == 0 = "Success"
4                  | otherwise     = "Fail"
5
6  -- val :: [Char]
7  val = "Hi"
8
9  main = putStrLn $ show $ branch 1
10
```

Listing 4: Basic parallel Fibonacci computation using the rpar evaluation strategy from the Strategies library

2.4.4 Runtime (RTS) Metrics

GHC's runtime system has many features for controlling and collecting information about overall execution. It can provide a complete report on the time and memory allocation required for program execution. However, as profiling performs additional computations on top of normal program execution, this may cause inaccurate energy readings when sampling programs. Therefore, it is excluded during data collection.

The remaining RTS results from sub-subection 2.4.4 can be summarised by the following points [65, 99, 102]:

- **Memory allocation** (*Lines 3–7*): This represents the results of memory allocated during process execution; it shows memory used for the heap, GC, the approximation of maximum memory during execution and maximum memory slop, i.e. the amount of memory wasted in the memory block because of GHC's memory allocation.

- **Generational GC** (*Lines 10–13*): Information provided here shows how many times garbage collection triggered and the number of parallel collections during each generation. For CPU time used and wall-clock time (elapsed), the lines show the parallel garbage collection balance, theoretically 100%, implying fully parallel GC time and 0% sequential. Based on the documentation of *work balance*, this is calculated using the number of threads used by the GC.

- **Tasks/Haskell Execution Context (HEC)s and Sparks** (*Lines 15–17*): In the context of GHC, tasks carry out the work of evaluating the Haskell code. As a simple analogy of bound tasks, consider a thread with a corresponding OS thread that ensures the OS executes foreign calls. A worker task is an internal task created by the `RTS`, which has a specific structure and a Haskell Execution Context (*HEC*) that provides a mechanism for executing and evaluating Sparks. Sparks in the context of Haskell are the *closures* to be evaluated. *Closures* in GHC are heap objects used to suggest to the RTS that a given computation can be evaluated in parallel (e.g. using `par/rpar`, as explained in Section 2.4.2). These are then made as separate sparks that are added to a *spark pool* having a specific size. The *HECs* then evaluates these, and the converted sparks means that these are evaluated all in parallel at runtime. Overflowed sparks mean that the generated sparks exceeded the spark pool size. *Duds* are sparks that have already been evaluated. *GC'd* sparks are those that have been created but were never evaluated during runtime, making them redundant. *Fizzled* sparks are those that were not evaluated when available in the sparking pool but were later evaluated during normal program execution.

- **Execution Summary** (*Lines 19–27*): The metrics summarised in these lines are divided into execution, runtime initialisation, mutation, i.e. the time taken to evaluate expressions, garbage collection, , exiting the process and finally total time. In each of these metrics the CPU time is stated, and the wall-clock time is shown between brackets.

- **Internal Counters** (*Lines 29 - 33*): Lines 29–32 are referred to in the documentation as *internal counters*, which are related to how the GC spinlocks operate and how *thunks/closures* are evaluated, these counters are subject to change in the GHC documentation and are unspecified in some cases.

- **RAPL [29] Power, Energy, Execution Time** (*Lines 33–37*): These lines are printed by the parent process that executes the binary to be sampled with arguments

passed to it. It then reports the total energy consumed during the execution. The average power is the average watts drawn during execution. Finally, the total time taken during execution is milliseconds longer than the RTS summary of elapsed time, given that the parent process exists after the child one.

```
cpu sig = 406f0
(-2574.1846) :+ 1081.3689
  5,997,364,096 bytes allocated in the heap
  1,901,167,120 bytes copied during GC
      3,016,448 bytes maximum residency (1245 sample(s))
        268,888 bytes maximum slop
              8 MB total memory in use (0 MB lost due to
                 fragmentation)

                                     Tot time (elapsed)  Avg pause  Max
                                        pause
  Gen  0       6483 colls,  6483 par    2.46s    1.23s     0.0002s
     0.0010s
  Gen  1       1245 colls,  1244 par    1.40s    0.70s     0.0006s
     0.0052s

  Parallel GC work balance: 40.45% (serial 0%, perfect 100%)

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

  SPARKS: 6900 (6899 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time   32.02s  ( 16.01s elapsed)
  GC      time    3.85s  (  1.93s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time   35.88s  ( 17.94s elapsed)

  Alloc rate    187,277,351 bytes per MUT second

  Productivity  89.3% of total user, 178.6% of total elapsed

gc_alloc_block_sync: 147188
whitehole_spin: 0
gen[0].sync: 8
gen[1].sync: 24880
PARENT: Child's exit code is: 0

        Total Energy:   613.847 J
        Average Power:  34.0557 W
        Time:   18.0248 sec
```

Output 1: Partial RTS results from executing Haskell

2.4.5 Evaluation and Parallelism in Haskell

In GHC, the runtime system comprises many parts: memory and storage management, scheduling, profiling and more. Most have had implementation changes that, over the years, matured and allowed for high-performance applications. The focus on this section is to provide an overview of how the internal threading model works in

GHC. It also explains how the evaluation and representation is performed at runtime.

2.4.6 Lazy By Default

Haskell's design makes heavy use of laziness. In laziness, no values are required to be evaluated unless they are *demanded*, and once evaluated, they can be shared among same-scope expressions where they are demanded. For example, in Figure 2.6 on page 26, a program defines two functions: `f` and `main`. Function `f` (Lines 1 and 2) takes three arguments: `x`, `y` and `z`. `main` is the main point of entry for Haskell programs. In `f`'s implementation, the function has a conditional statement, namely `if x`. When `x` is evaluated, it also evaluates `w * w`, where `w`'s definition is based on squaring the `y` argument. If the Boolean value is evaluated as false, `z` will be the result. In `main`, `f` is applied to values `True`, `2.0` and `(1.2/0)`, then it is applied `print` to `f`'s result. The evaluation of `f` will always include evaluation of expression `w * w` as its first argument, and `x` is always True. When evaluating `w * w`, we only need to evaluate `w` once, which is `y * y`, thus reducing the number of steps needed to fully evaluate `w * w`. Most importantly, the arithmetic expression, `(1.2/0)`, is never evaluated. In strict imperative languages, e.g. Java, C/C++, an expression containing a division by zero will cause an exception. An example is a Java-like array with elements `{1/2, 2/0, 21/3}`, although the length function does not require the evaluation of elements. The previous concept is referred to as *Call-by-Value*, which is implemented in most imperative-style languages. On the contrary, *Call-by-Need* is used in Haskell and does not cause errors or program halts, as the evaluation a computation like `2/0` would only cause errors if the program demanded the value be computed.

2.4.7 Normal Form and Weak-Head Normal Form

Haskell expressions may or may not be evaluated owing to the nature of lazy evaluation, based on whether program execution requires the values of those expressions. An unevaluated expression or one that can be reduced to a constant value, such as `(x : xs)`, is referred to as *WHNF*. Applying the Haskell operator, `Cons` uses (:) on `x`, where the collection of `xs` may have a polymorphic type, e.g. `[(a, String)]`. For this kind of evaluation, we only observe the top-level reducible expressions or concatenated

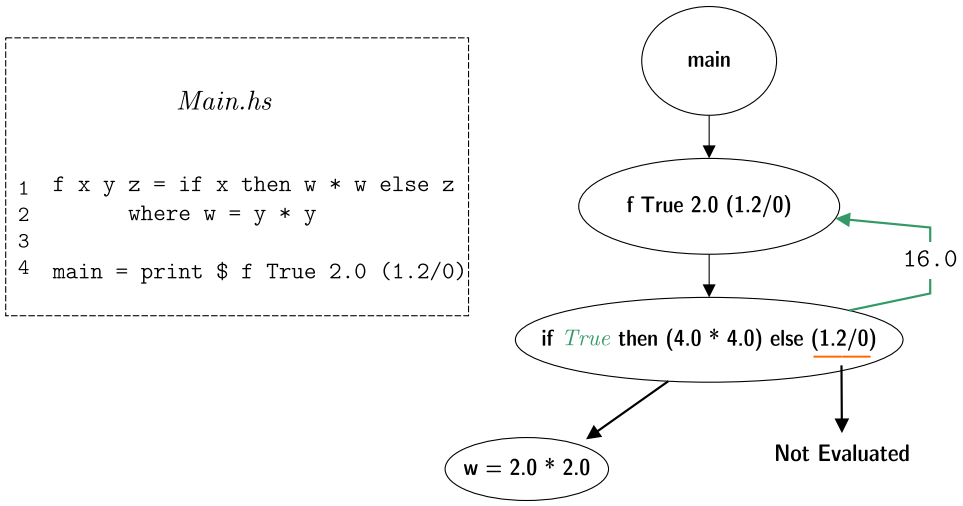


Figure 2.6: Lazy evaluation tree for minimal expression

redexes. In this case, the concatenation operation is reduced to [x, x1, x2, ...]. Note that none of the elements or expressions within were evaluated in any way; only the top-level expression was considered. x and the remaining list elements, xs, may have any expressions, but they are not evaluated until demanded. In this case, the current expression is WHNF. Normal Form (NF) is only reached when an expression cannot be further evaluated, i.e. there are no reducible expressions. Using the previous example, for NF, the expression may have the following form: [(4, "R1"), (2, "R2"), ...]. Note that the list does not contain any reducible expression. Thus, it is NF.

2.4.8 GHC Sparks, Spark Pools and Capabilities

The notion of self-managed execution and scheduling in programming languages is commonplace. Many apply philosophically driven implementations. Haskell implements threads and exposes them through the following components.

Sparks and closures are explained in Section 2.4.4 on page 22. They are basic component representing a computation [30, 65, 76, 102]. Whenever a spark is created, it represents a computation that will be evaluated at a later time. During execution of a Haskell process, sparks have a single state out of five. A converted Sparks means that its computations are evaluated. Other states include Overflowed, Dud, GC'd and Fizzled. A Spark cannot be added to the pool when the maximum capacity is reached; therefore, an Overflow occurs. If a Spark was already evaluated, garbage collected or GC'd before it could be evaluated is a Dud. Finally, if the current runtime

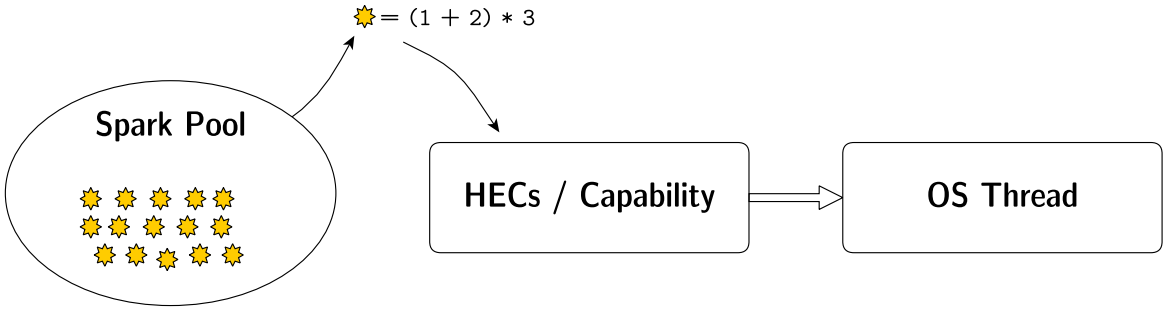


Figure 2.7: Example of a given spark pool containing a set of *closures/thunks* to be evaluated

was not the issuer of a Spark it is a fizzled. Each Spark is drawn from a Spark Pool containing several *Thunks* or computations under evaluation. These are then issued to Capabilities or HECs. This process can be envisioned as a corresponding thread interfacing an OS thread executing on a given processor. An example is seen in Figure 2.7.

2.4.9 Haskell's Strategies

Haskell's GHC provides a variety of APIs to implement parallelism. This includes a concurrency API containing STM with `forkIO` and mutable variables `MVar`, `TMVar` that can also run executions in parallel. The parallel monad, `monad-par`, exploits parallelism through a monadic implementation. Finally, the Strategies library [104] implements high-level parallel constructs of parallel patterns, providing the ability to execute various types of parallel processing with a simple, functional composition wherein the user chooses an evaluation strategy, e.g. `rpar/rdeepseq`, to process data in parallel. For example, a common parallel function used in Haskell is `parList :: Strategy a -> Strategy [a]`, which takes an evaluation strategy, e.g. `rpar/rseq`, and evaluates all elements of a list according to the evaluation chosen. Another common but higher-level Strategies function is `parMap :: Strategy b -> (a -> b) -> [a] -> [b]`, which in its internal implementation uses `parList` to apply a function over elements of a list using an evaluation strategy.

To understand how the Strategies library works, we consider Quicksort: an algorithm for sorting elements in a list of a given size. The sorting process traditionally uses a pivot value that acts as a marker for comparison with other elements beginning

with the leftmost. If a given element is smaller, the marker moves to the next index until it finds a value greater than or equal to the pivot value. It then switches the pivot with the greater value. Then, the rule is reversed, beginning from the rightmost value or the end of the list. The pivot is then switched with smaller values. The process continues until the pivot arrives at the middle, where smaller values are at its left, and larger values are at its right. Then, the same sorting process is applied to the left and right partitions of the list, continuing to subsequent partitions until the full list is sorted. Listing-5 is a parallel Haskell version of Quicksort, beginning with Lines 1–2. These are base cases of Quicksort wherein an empty list returns another empty list, as does a single element. The function begins sorting when a list has more than one element, e.g. Line 3. In this case, `losort` acts as the smaller or lower partition to the pivot, `hisort` as the greater or higher partition and the pivot is chosen to be `x`, which is the first element in the list (Line 4). Then, both list partitions are concatenated. The remaining part applies a strategy with the function, `using`, passing it a strategy named, `strategy`, in the next part, which is defined in the `where` clause. The `losort` uses a list comprehension to produce a list that compares all tail elements to the pivot, `x`, and applies `quicksortS` to that list (Line 6). The predicate occurs where `x` is always larger; otherwise, the element is filtered. The same holds for `hisort`, except that `x` must be either larger or equal. The parallelism part at Lines 8–11, where the strategy is defined, uses the initial recursive call to apply composed strategies of `rdeepseq` and `rpar`, sequentially using `dot`. `rdeepseq` allows forced evaluation of expressions until *normal form* is reached. Then, it sparks the subsequent recursive process. Finally, at Line 11, it fully evaluates `res` from the initial application of `using`. It is worth mentioning that the example described in Listing-5 is not related to the original Quicksort algorithm where Hoare [39] describes an in-place algorithm, which cannot be achieved in Haskell.

Notably, other parallel models are available in the Haskell ecosystem, e.g. STM and `Control.Concurrent` API. STM is one of several methods of handling concurrent/-parallel abstractions in Haskell. Through a *mutable reference*, primitive constructs, such as `TVar`, `Mvar` and other concurrency primitives, can achieve parallel/concurrent processing. `Control.Concurrent` provides more granular control of threaded workloads using functions like `forkIO` or `forkOS`, where the parallel structures of a program are more dependent on the implementation.

```
1  quicksortS []     = []
2  quicksortS [x]    = [x]
3  quicksortS (x:xs)  =
4    losort ++ (x:hisort) `using` strategy
5      where
6           losort = quicksortS [y | y <- xs, y < x]
7           hisort = quicksortS [y | y <- xs, y >= x]
8           strategy res = do
9                          (rpar `dot` rdeepseq) losort
10                         (rpar `dot` rdeepseq) hisort
11                         rdeepseq res
12
```

Listing 5: Example of QuickSort using Haskell's Strategies library

2.4.10 Glasgow Haskell Compiler's (GHC) Garbage Collector

GHC has multiple Garbage Collector (GC) implementations that behave differently. The default, which has been applied in all benchmarks used in the upcoming chapters, is the *parallel, generational, two-space, stop-the-world, garbage collector*, which uses several notions to address the memory management required by the language. The parallelism in the GC system in GHC evolved over several iterations. However, the basic concepts remain like the original parallel GC implementation. The work by Marlow *et al.* [66] and Sansom *et al.* [91] describe the approach of the parallel GC in detail. The parallel or multi-threaded GC operates based on execution configuration over several cores/processors, Haskell threads, *HECs* or *Capabilities*. These terms refer to the same component run by a pool of Operating System (OS) threads, where one runs a capability, then another picks the work where the previous one stopped. In essence, the thread concept differs from classical *POSIX* threads. Each Haskell thread has a *local heap* where it can manage memory without synchronisation with other threads, meaning it can allocate and perform garbage collection by itself. Then, there is a *global heap* that is accessible by all threads. However, garbage collection in such a space is only made when all threads synchronise. As Haskell uses immutability in almost all cases, object allocation is performed frequently for new computations as mutability increases the difficulty of maintaining details of given memory objects. The approach in which HECs handle the evaluation of *Sparks* as described by [68], is based on HEC's pool that queues the Sparks assigned to itself, there is also a shared pool/heap of Sparks that is shared among all HECs. When a HEC has more than one Spark, it will check other HECs in the runtime and if any are idling i.e. with

no Sparks in their local pool, the busy HEC will control the idling HEC to transfer some of its Sparks to initiate the Spark evaluation by the idling HEC and reduce the number of Sparks by its own local pool.

Another factor of GHC's GC is its generational part. The allocated *closures* [1] are immutable; they will not have any future updates after the computation completes. They also will not point to any future heap objects. Therefore, collecting such objects is easier and safer than traditionally mutable garbage collection techniques. This simplification can allow for generations of live and used heap objects to be promoted to an older generation or, in this case, to *gen 1*, resulting in all other objects in *gen 0* to be collected to maintain enough memory for new closures. This translates to a higher number of heap objects allocated. However, they are quickly collected without having to maintain and frequently lookup various pointer tables having have complex details about each object. The final part of this algorithm is *stopping the world*, meaning that there is set of rules for pausing program execution and garbage collection, e.g. *minor GC* and *major GC*.

2.5 Target Configuration and Platform

The experimental testbed machine was a 28-core shared-memory multicore system at the University of St Andrews, *corryvreckan*. The machine had two Xeon E5-2690 processors in separate *packages*, each with 14 physical cores (28 virtual), giving a total of 28 physical cores (56 virtual). It had 256 GB of physical DRAM shared among all cores. The machine ran *Red Hat 4.8.5–11* with kernel version *3.10.0-514.21.2.el7.x86_64*. To improve the stability of our measurement results, we restricted it so that it could be used only for our experimental purposes, plus standard operating system functions. GHC version 7.6.3 was used. This was the default version of the Haskell compiler installed on this system. All experiments were run with optimisations turned on (`-O2`) and using *eager black holing* (`-feager-blackholing`) to avoid accidental duplication of parallel work. GHC was run with the default runtime memory configuration. By default, the system used *dynamic voltage and frequency scaling* to limit energy usage. To obtain consistent and repeatable measurements that can be used to construct rational and predictable energy usage models, the system was configured to run at

[1] Here, the term *closure* is specific to the *Spineless Tagless G-machine* [82] (STG-machine) where all heap objects are referred to as *closures*

several stable clock frequencies (up to *2.60 GHz*) using the built-in Linux *performance governors*. Having made these changes, we minimised dynamic and unpredictable effects caused by thermal throttling, or by clock boosting drivers, such as Intel's P-State.

2.6 Benchmark Simulations

To provide an extensive dataset of energy profiles that simulate various types of workloads, we required multiple types of computations with different types of execution patterns, workloads and other algorithm-specific optimisations. A group of the *Nofib* [80], *PARSEC* [8], and *the Computer Language Benchmarks Game* [100] was used with Discrete Fourier Transformation (DFT) as an additional benchmark [79] to the Haskell group. Additionally, the complete code of the modified implementations used can be found in the online [2]. The following sections provide an overview of all benchmarks used in energy profiling; they are modelled in subsequent sections. The Table 2.1 provides a summary of all parallel benchmarks used in this thesis alongside the types of parallel constructs. In the table, the term, *Unstructured Parallelism*, refers to the implementations with more of a provisional or unstructured parallelism.

2.6.1 C/C++ Benchmarks

Spectral normalisation: The *spectral norm* benchmark algorithm finds the largest *eigenvalue* [42] of a given matrix. The algorithm comes from *the Computer Language Benchmarks Game* version 22.0 [100]. The algorithm finds the product of two matrices, one of a given size, and *transposes* it to find its eigenvalue by forming an equation of the product of the previously mentioned matrices. The parallel portion of the algorithm is based on data parallelism using OpenMP's loop `pragma`, which breaks down the matrix structure to multiply different vectors in nested loops. The C implementation uses the Single Instruction Multiple Data (SIMD) data type, which enables faster processing of computations via multiple chunks data, concurrently.

Binary trees: The binary tree benchmark is based on a garbage collection simulation inspired by Boehm's GC benchmark [10]. The algorithm has three parts, all of which

[2] github https://github.com/ymg/thesis

	TBB	Pthreads	OpenMP	GHC / Strategies	Task Parallel	Data Parallel	Unstructured Parallelism
SumEuler	✗	✗	✗	✓	✗	✓	✗
Nbody	✗	✗	✗	✗	✗	✗	✗
DFT	✗	✗	✗	✓	✗	✓	✗
N-Queens	✗	✗	✗	✓	✗	✓	✗
Ray	✗	✗	✗	✓	✗	✓	✗
MatMult	✗	✗	✗	✓	✗	✓	✗
PRSA	✗	✗	✗	✓	✗	✓	✗
Partree	✗	✗	✗	✓	✗	✓	✗
Parallel Quicksort	✗	✗	✗	✗	✗	✓	✗
Spectral-norm	✓	✓	✓	✓	✗	✓	✓
Prime Decomposition	✓	✓	✓	✓	✗	✓	✗
Blackscholes	✓	✓	✓	✗	✗	✓	✓
Bodytrack	✓	✓	✓	✗	✗	✓	✓
Binarytrees	✓	✓	✓	✓	✗	✓	✗
Fasta	✓	✓	✓	✓	✗	✓	✓
Ferret	✓	✓	✗	✗	✗	✗	✓
Swaptions	✓	✓	✗	✗	✗	✓	✗

Table 2.1: Benchmarks sampled and their corresponding library implementations

involve binary tree data structures. The C++ version uses the Boost library [92] with OpenMP. The benchmark starts by allocating a binary tree of a given size, followed by an operation that stretches or adds an extra node to the tree. Next, the workload creates a tree of the same size, which is kept in memory for the remainder of the benchmark life-cycle. For Haskell, The parallel portion of the benchmark is when the function `parMap` applies the function `rnf` which reduces its argument to *normal form*. In C/C++ version, the parallelism is made possible by setting the maximum and minimum depth of parallelism for which a thread will compute. For example, OpenMP and TBB versions both use a parallel loop that starts walking the nodes of the tree from minimum depth to maximum while processing each node of the tree for a given number of iterations. The Pthread version on the other hand, segments the computations between specific threads that are then executed by a single thread each.

Fast DNA sequencing (fasta): The *fasta* algorithm generates a set of random DNA sequences, implementing a *linear congruential generator* (LCG), which produces a pseudorandom float. The algorithm builds a set of these pseudorandom numbers

that are then used to construct parts of the DNA, i.e. nucleotides, based on a specific length, which in this case is 60 per block. The parallelism implementation exploits a *map* style of data parallelism that maps process chunks independently as the data do not have dependencies. Similarly, both TBB and POSIX thread versions process the computation, but with different internal mechanisms for handling parallelism, initialisation etc.

Prime decomposition: This benchmark finds the *largest minimal prime factor*, which requires prime decomposition computations made in parallel. The types of computations can be processed independently from each other, meaning that the execution can be split among available threads without defining a *reduction* step. The algorithm finds the largest prime factors of a given (non-prime) integer in a list, which requires finding primes having the product of given numbers followed by checking for their factors in the given list. In C/C++ versions, this requires a single step that allows variable sharing between threads, which then assesses the factors produced in a loop. If such factor are larger than the shared variable, it overwrites it with the new value.

Black Scholes model (blackscholes): The Black Scholes benchmark is based on a financial model that helps with option trading decision using approximated asset values. The blackscholes formula is solved by a series of floating point computations. The PARSEC implementation uses x86's SSE intrinsic functions for fast arithmetic. The computation is achieved via parallelism of data using SIMD, where the number of concurrent options processed is set to a given value, i.e. four. The implementation applies three forms of parallelism, OpenMP, TBB and POSIX, threads with equivalent implementations by allocating the number of threads and applying the blackscholes solver function in a parallel for-loop.

Bodytrack: Bodytrack simulates a computer vision workload wherein a human object walks in a scene. The simulation uses multiple angles to capture motion using *annealed particle filters*, where the edges of the human shape are tracked using multiple simulated annealing steps to reduce the number of particles required for tracking. The parallelism takes a similar approach in all three implementations: TBB uses a *pipeline* of parallel loops to implement the frame processing sequences; OpenMP uses a similar structure in which parallel loops are achieved through a set of functions calls with parallel `for` pragma is embedded in each function; the POSIX threading

example uses a different parallelism style wherein a specific implementation of a worker/thread pool is initialised and assigned work as required.

Ferret: The Ferret benchmark simulates a non-textual similarity search process based on image type where a set of images in a database are compared to a given one. Both TBB and POSIX threads are pipelined through different stages, from loading images in parallel to extracting properties, such as Hue, Saturation and Value (HSV). Segments are then constructed, followed by a vector similarity procedure and ranked result production.

Swaptions: The swaptions algorithm computes *swap options* that allow an investor to enter a swap contract. Because it is an option, this offers the investor the chance to trade without obligation until both parties agree on the swap. Therefore, an assessment is required by the investor offered the swap. The algorithm uses the Heath–Jarrow–Morton framework (HJM) simulation to compute future interest rates using various factors, e.g. volatility, then deciding whether a swap is good. The implementation of *TBB* and POSIX threading uses `for` loops that process the list of swaptions in parallel. *TBB* uses a loop with *grainsize* of one, referring to when the processor needs to split the load between available cores/processing elements, where the range will be processed from zero to the number of swaptions provided as input.

2.6.2 Haskell Benchmarks

Minimax: The minimax algorithm aims to solve a board game where n-players are competing to win. The turn-based game generates a fixed 4×4 *tic-tac-toe* board where every two players must form a line of three symbols to win. The algorithm finds best moves to form diagonal, horizontal, or vertical lines to win. Parallelism is introduced as a recursive `alternate` function that returns a sequence of moves as a list. The parallelism uses `parList`, wherein each element is a computation that identifies possible moves and chooses the best.

N-Queens: This benchmark solves the n-queens problem, where the requirement is to place an N number of queens on a chess board without any queen piece being able to attack another. Originally, the size of the board is 8×8. However, when increasing the number of queens by N, the board size also increases by $\mathrm{N} \times \mathrm{N}$. The `pargen` generates a list of *map* computations that are then evaluated in parallel

using Strategies' `parList` function. Each computation is a recursive call to `pargen` that applies a `safe` function over a list comprehension of `gen`, which generates the placement of a queen using a nested list.

Parallel Fibonacci (ParFib): In parallel Fibonacci, the algorithm works in a Divide and Conquer (D & C) pattern to recursively produce Fibonacci numbers. Parallelism is introduced by generating a set of *sparks* on one part of the computation to force a specific order for the remaining computations, e.g. a binary tree of *thunks* to be evaluated.

N-Body: N-Body is a simulation that demonstrates the gravitational effects of celestial bodies in a solar system. Each body has attributes that dictate object movements, such as gravitational force, velocity, mass and position. Such attributes affect how a given body interacts with others in the system. The N-Body algorithm simulates these effects with multiple steps, beginning with the creation of celestial bodies and ending with the assignments of specific attributes to the different components of the system. The parallelism in this benchmark uses a various techniques to avoid excessive allocations and makes use of *unboxed* values alongside explicit uses of arrays. It also uses a *fork/join* pattern to control the parallelism, wherein each chunk of elements in an array is evaluated by a specific HEC or *capability*. Following the creation of celestial bodies, a monadic for-loop is used to evaluate all necessary bodies and to map the *join* function to ensure that the evaluation is completed.

Parallel tree mapping (Partree): Partree handles the mapping of a function over a tree. The function applies a set of arithmetic operations where the final result is produced as a single integer. Parallelism is made possible by introducing a set of *sparks* recursively for both right and left leaves in the tree. *sparks* are created recursively using a basic computation over the leaves of the tree nodes.

Parallel QuickSort: This benchmark implements the well-known Quicksort algorithm to generate a list of randomised numbers that are then sorted by dividing the list into two parts: left and right. Then, the sorting procedure continues to divide the lists recursively down to a certain depth, i.e. eight elements per chunk. The parallel aspect of sorting takes place when the sparks are generated using `par`. The algorithm has a D & C skeleton, but it does not use an API.

Euler summation (SumEuler): In this benchmark, the computation relies on the *Euler totient function*, which uses a parallel strategy in which all chunks are split at given number, and each chunk is evaluated until WHNF is reached. For example, when using a chunking size of 100, and the size of the list is 1000, the evaluation runs 10 chunks in parallel, wherein each is evaluated by applying the Euler function to each number in the chunk, followed by summing all elements together.

Matrix multiplication (MatMult): This implementation of matrix multiplication uses a simple form of computing the product of two matrices. Strategies consist of *line*, *block* and *column* processing. Control is managed by program inputs. For example, in the column processing style, the program uses `parListChunk`, which split the input list to chunks of a given number, usually provided as an input to the program. It then applies a *transpose* transformation for the second matrix followed by list comprehension, which is a set of *vectors* multiplied according to the column-based strategy.

Ray tracing (Ray): Ray tracing works on stream of rays to render a scene containing a number of objects at a given resolution. In this algorithm, a scene is generated based on program input, followed by the application of a function for generating the required rays. The outputs of both are then used as input to a third function, `findImpacts`, which applies parallelism using `parBuffer`, which evaluates a list of elements using a specified buffer size. The buffer is used to control the lazy evaluation in parallel. Unlike a normal parallel list evaluation, `parBuffer` creates a set of *sparks* based on the buffer, which are evaluated and examined by another set. This method avoids consuming linear, as `parList` does.

Parallel RSA (PRSA): In the parallel RSA benchmark, the algorithm performs a long sequence of encryption computations. The `encrypt` function applies asymmetric encryption using public and private keys that remain the same throughout execution. The data include an encrypted, replicated string consisting of the letter 'x'. The goal is to run all encryption computations in parallel. The process begins by creating chunks based on the first key, followed by a parallel `parBuffer` application that maps a set of functions over the produced chunks.

Discrete Fourier transformation (DFT): The DFT benchmark is based on an algorithm that employs the Fourier transformation over a set of complex numbers,

given a stream of frequencies generated as a random sequence. The algorithm uses a *strategies* function, `parMap`, in two different steps. First, a `parMap` applies a lambda function, which applies another map over a list of numbers to generate *twiddle factors*. Using `rpar`, the `parMap` generates *sparks* and prepares them for evaluation. `parMap` uses `rdeepseq` to fully evaluate all computations to NF.

2.7 Summary

This chapter discussed the different tools and technologies that are essential to understanding and evaluating the problem of energy consumption in programming languages. The chapter also presented an overview of basic GHC and Haskell features and the way in which evaluation is possible using the language. The chapter also discussed the different hierarchies of parallelism and parallel libraries using examples from functional and imperative languages. Finally, the chapter described the testbed configuration alongside the tools and versions used.

Chapter 3

Energy Modelling and Prediction

In this chapter, we examine the benchmarks selected to evaluate Haskell and C/C++ energy consumption. Then, we present a statistical approach to predicting energy consumption using multiple models. Section 3.1 introduces the metadata collection setting. Section 3.2 presents the profiling method of the different datasets sampled. Section 3.3 presents the types of models used with the appropriate metrics needed to evaluate each model. Section 3.4 presents the evaluation of the predictability of each model on a new dataset with equivalent implementations in Haskell and C/C++. Section 3.5 highlights the energy effects using various CPU clock frequencies.

3.1 Modelling for Energy

When sampling any benchmark, it is essential to collect information about execution energy profiles and associated information. The more data collected, the greater the chance that it will be possible to understand energy consumption. Benchmark information can then be used to build regression models for predictive capability. To understand energy consumption based on languages used, we chose several benchmarks, as identified in Chapter 2 on page 7.

This chapter explains the sources of data collected and what they represent, followed by exploring candidate regression models and an overview of the results.

3.1.1 Data Collection

The benchmarks identified in Chapter 2 have multiple implementations, and some have equivalent implementations in both Haskell and C/C++. A crucial aspect of evaluating language models and benchmark predictive capabilities is to use programs with comparable behaviours. Generally, both Haskell and C++ compilers apply transformations to particular programs in similar and understood ways. Therefore, we expect to see differences in overall execution time and, consequently, power and energy. To compare Haskell and C/C++ performance, we must split the benchmarks introduced in Chapter 2 into three categories.

The first category is for C/C++, which comprises PARSEC benchmarks with the largest inputs/workloads available. PARSEC benchmarks take inputs of varying sizes. However, they may entail extensive datasets or complex scenes, as with Bodytrack. The types of input pose limitations to sampling as the benchmarks have small-to-medium input sizes that may not help with building an extensive energy profile or a single large input size. Using the largest input ensures that a program spends enough time executing so that we can identify computational patterns. The C/C++ benchmarks used for modelling include *blackscholes* (i.e. OpenMP, TBB, Pthreads), *Bodytrack* (i.e. OpenMP, TBB, Pthreads), *Ferret* (i.e. Pthreads, TBB) and *Swaptions* (i.e. Pthreads, TBB).

The second category includes Haskell benchmarks from the Nofib suite. Unlike PARSEC, Nofib benchmarks take simple inputs, e.g. a number, as an argument, which allows sampling at more frequent intervals and workloads. The benchmarks used for constructing the parallel Haskell models include *DFT, Nbody, Minimax, MatMult, Partree, Quicksort, Parfib, Sumeuler, Ray, NQueens* and *PRSA*.

The third category includes the modelling assessment benchmarks, consisting of all programs implemented in both Haskell and C/C++. These benchmarks also take simple inputs consisting of single or multiple numbers, which allow for sampling at more frequent intervals. These benchmarks include *Spectral-norm, Binarytrees, Fasta* and *Prime Decomposition*.

Program data were collected from different sources using multiple tools, and each source provides details that contribute to explaining program behaviour. Benchmark

sampling is treated equally in this study, meaning that all are sampled using the same tools multiple times to avoid anomalies that could affect outcomes. The energy sampling population is assumed to be infinite. In statistical sampling and choosing the correct sample size [27], the required sample size to have a margin error of $\pm 5\%$ is a minimum of 385, which in our case is achieved by increasing the number of samples to 1000 per input per benchmark. The reason for choosing an infinite population is that we assume that possible CPUs and configurations can affect how energy consumption happens for the benchmarks in this thesis. The data that we collect are acquired via the sampling benchmarks using the following tools:

- RAPL: a CPU-specific power draw interface that allows reading power values through special CPU registers. RAPL sampling is provided by [24].
- Perf: a Linux performance analysis tool that details process execution.
- Intel Software Development Emulator (SDE) [97]: a process emulation tool that emulates process execution with real input and provides breakdown information on instruction counts.

A more detailed RAPL explanation is provided in Chapter 2. In the following sections, we describe the types of data derived from Perf and Intel SDEs.

3.1.2 Perf

Perf [71] is a tool that enables collecting different metrics from a Linux operating system. For example, it can count system events, e.g. system calls, while providing ways to record and report such information. Perf also performs process and system-wide event collection. However, the counters we collect are program-specific, including collecting process-specific triggers lacking interference from events triggered by other processes.

Perf is used to sample the benchmarks identified in Chapter 2. The metrics collected will vary by process execution and computation type. For example, when using Perf to profile system calls of the *Fasta* benchmark on the exact core count and input size, the Haskell parallel program reports Linux *epoll* interface system calls, such as `sys_enter_epoll_create`. However, the same system calls will not exist in the equivalent C/C++ implementation. Language-specific implementations of memory management and, for Haskell, runtime components produce heterogenous

program metadata, which makes sense as the programming languages differ, and the benchmarks use instructions differently, although the behaviours are the same.

Perf provides insightful data from the operating system's user space and the kernel side of Linux-based operating systems. It also provides an overview of hardware events, such as cache misses and CPU cycle counts, while also providing details on tracepoint events, such as system calls and operating system scheduling. It is possible to mix data types for capture by specifying event names when executing Perf on a given process. We sample benchmarks for the following events:

- *syscalls*: a complete list of all *tracepoints* making a system call to the operating system.
- *cycles* or CPU cycles: a complete count of all CPU cycles used during program execution.
- *instructions*: total count of instructions executed for a given process.
- *cache-references*: total count of cache accesses.
- *cache-misses*: total count of cache misses resulting in main memory fetches.
- *bus-cycles*: total count of CPU bus cycles as they vary among CPU cycles.
- *branch-instructions*: total count of branching or program changes of flow events.

The above events are only the second metadata type to be collected for the benchmarks to be sampled. It is essential to understand that these events can use different counts, depending on the hardware used when running Perf. Additionally, the configuration used for such hardware, e.g. clock frequency boosting or having lower-/higher clock frequency, can affect the number of CPU cycles. Additionally, the CPU architecture affects the levels of cache, and their sizes affect cache miss ratios and references. Therefore, the profiling step is essential to understanding the advantages or disadvantages modelling energy consumption using such metadata.

3.1.3 Intel Software Development Emulator (SDE)

Intel SDE [97] is an emulation tool with many features used to examine and understand Intel CPU instruction sets. It is possible to execute a program on an Intel

Broadwell or Haswell CPU instruction set while the underlying CPU has a different Intel architecture. Emulating these environments can produce metadata to help debug programs without having physical access to specific CPU instruction sets. Owing to the power of Intel SDE, a complete count of instructions and extensions can be produced to understand program behaviours at the lowest levels. Intel SDE uses another tool for dynamic execution, Pin [4], which drives the execution of a given program. SDE's ability to provide a dynamic count of all instructions executed and a breakdown of all instructions for each function produces high granularity metadata regarding program execution. To obtain a complete mix of functions, we only need to run a binary of a given benchmark using the -mix flag, and Intel SDE provides a complete list of instructions counts and each instruction categories (emulated). If an ISA flag is not specified, Intel SDE defaults to the ISA emulation of the available CPU. In this case, it is the NUMA nodes of Xeon E5-2690.

The instructions reported after emulating a program may differ by architecture, and even the counts may differ by a small margin, e.g. an addition operation with carrying. An (ADC) instruction can be executed in the first sample 20.5M times globally, but in another sample, it may require 21M. Although the instruction count discrepancy between samples can be significant when examining numbers alone, the count varies considerably when running a program with different inputs or core counts. For example, ADC instructions for *Binarytrees* with inputs of tree depth 21 and 22 that are emulated at 22 cores. The count differences between the two samples are three executions for input size (depth) 22. however, the ADD instruction executions for *Binarytrees* for 21 and 22 inputs for the precise core count are 40 and 90 billion. Therefore, it is expected that we will notice relative changes and differences in instruction types whenever the core count changes or when program inputs vary.

As x86 assembly is backwards compatible, assembly instructions continue to grow in number. As Intel's SDE emulates instructions using x86 assembly language, the number of instructions available to report can be significant. It would be unfeasible to apply a fixed number of features or restrict a subset, as certain features may be incompatible with one architecture but compatible with another.

3.2 Benchmark Energy Profiling

3.2.1 Cache misses and Cache references

Cache miss and cache reference are two concepts that are tightly coupled in current software execution. If a given computation needed to operate on a given data of any shape, it will need to access that based on local-memory or cache, which is referred to as a cache reference operation. When the data demanded is not available already in the cache memory, the cache controller would need to fetch that specific data into the cache to allow the computation to continue, this operation is referred to as cache miss.

Some computations are heavily reliant on the type of data they operate on, which is why cache misses might increase in some software compared to others. When assessing the cache misses of a group of selected benchmarks from the modelling datasets like **MatMult**, we observe little to no effect of how the cache behaviour affects energy consumption. For example for MatMult with input 2000, the sequential or single core sample used energy of 2728.55 J with cache misses of 40477778, compared to the 27-core sample which consumed 394.069 J and had cache misses of 110202575. A similar case for MatMult 1500, where the single core sample had the lowest cache miss count of 20409460 had energy consumption of 1238.54 J, similarly we see increase in cache misses as we increase the number of cores used where the 27-core sample results in cache misses of 62302263 and energy consumption of 174.157 J. As it seems that PARSEC and Haskell's Nofib models indeed did not include any of the cache-miss/cache-reference in the final statistical models constructed which indicates the lesser importance of these features.

3.2.2 Haskell Benchmarks

Results from the baseline energy results are visualised in Figures 3.1 3.7. The graphs show *speedups* over the sequential execution times vs. *total energy consumed* in *joules*. The black lines in the middle of each figure show the 28^{th} core, and points after the lines indicate the use of virtual and physical cores.

Parfib For *Parfib* (Figure 3.1), we set *intervals* to range from *50* to *56*, providing sequential execution times from 61.8 to 2018.3 s. We obtained near-linear speedups when we used 28 physical cores (27x speedup for $n = 56$, and 26$\times$ speedup for n

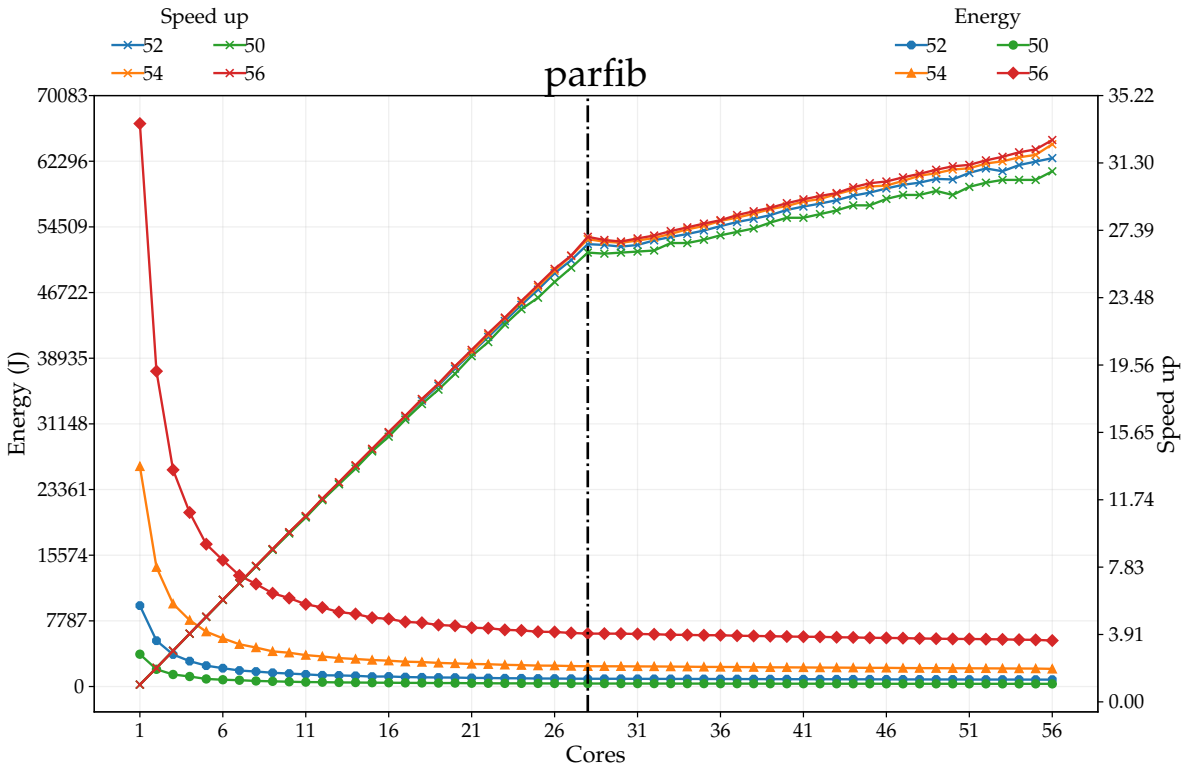


Figure 3.1: Parfib – *Energy vs. speedup* for Fibonacci numbers ranging from 50 to 56

= *50*). The energy usage was tightly correlated with execution time and indirectly with the speedup. When using virtual cores (beyond 28), although we still observed increasing speedups, they were no longer linear; however, energy usage continued to decline. When we reached 56 virtual cores, we observed maximum speedups of 32.64× for $n = 56$ and 30.8× for $n = 50$. As we increased core count, we clearly saw corresponding reductions in energy usage from a maximum of 66759.39 J on one core to a minimum of 5453.28 J on 56 cores at $n = 56$, and from a maximum of 3831.94 J on one core to a minimum of 315.99 J on 56 cores at $n = 50$. This represents just over a 12× reduction in energy usage when the complete machine was in use in all cases. We thus effectively improved *both* performance *and* energy usage. The reason for this counter-intuitive result is that energy usage is strongly correlated with time: each active package consumed energy, even when no cores were running. However, increasing the number of cores in use did increase the instantaneous power consumption, and the corresponding increase in energy usage was greater than that compensated for by the effect of the reduced execution time. Thus, the net result was lower energy usage.

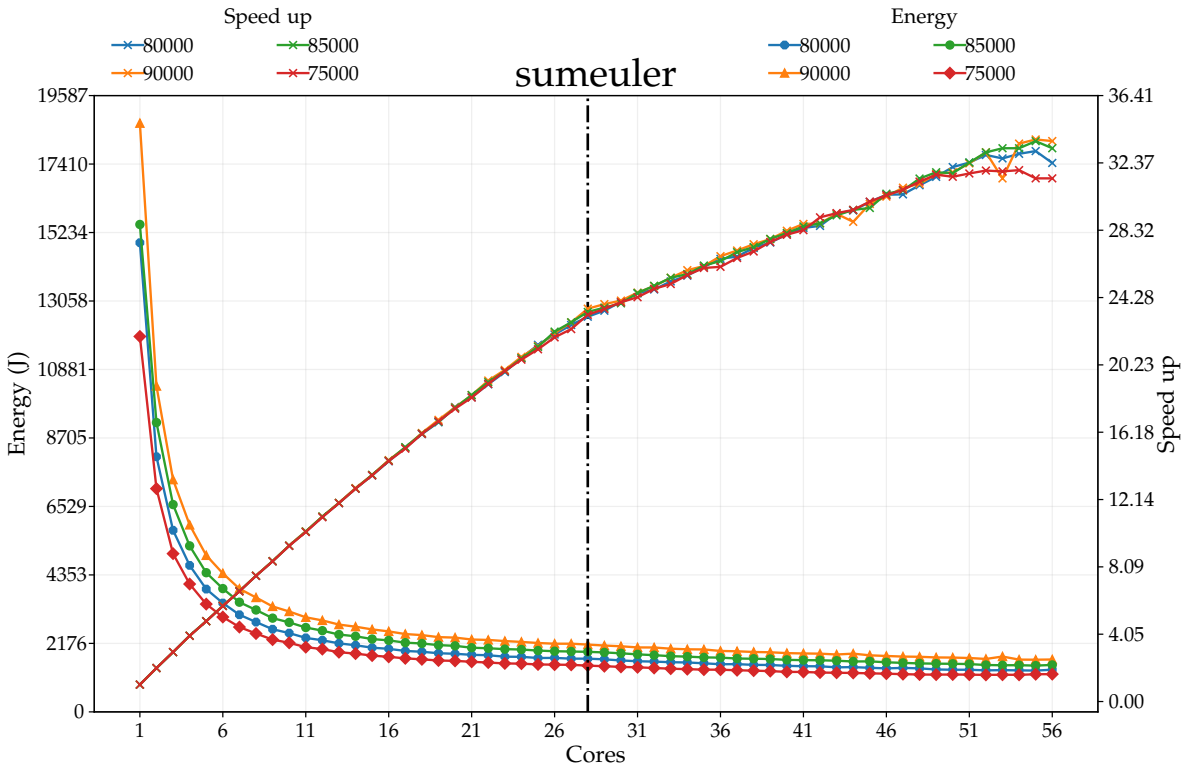


Figure 3.2: SumEuler – *Energy vs. speedup* for intervals from 75K to 90K; chunk size 400

Sumeuler For *Sumeuler* (Figure 3.2), we set the *intervals* to range from *75K* to *90K* with a chunk size of *400*, providing sequential execution times from 17.8 to 595 s. We once again obtained near linear speedups when we used the 28 physical cores (22.45× for $n = 90K$ and 22.01× for $n = 75K$), and we obtained speedup improvements for virtual cores. With 56 virtual cores, we observed maximum speedups of 32.46× for $n = 90K$ and 30.95× for $n = 75K$. As before, as we increased the core count, there were corresponding reductions in energy usage. Although the observed speedups became slightly erratic as we used larger numbers of cores, energy usage decreased consistently with the number of cores, from a maximum of 16084.77 J on one core to a minimum of 1656.38 J on 56 cores at $n = 90K$, and from a maximum of 10964.95 J on one core to a minimum of 1159.46 J on 56 cores at $n = 75K$. This represents an improvement in total energy usage of between 9.45× at $n = 75K$ and 9.71× at $n = 90K$. The differences between *Parfib* and *Sumeuler* suggest that energy usage depends not just on execution times but also on application characteristics. Although *Parfib* is essentially a pure integer calculation, *Sumeuler* also makes heavy use of data structures, which incurs memory traffic costs.

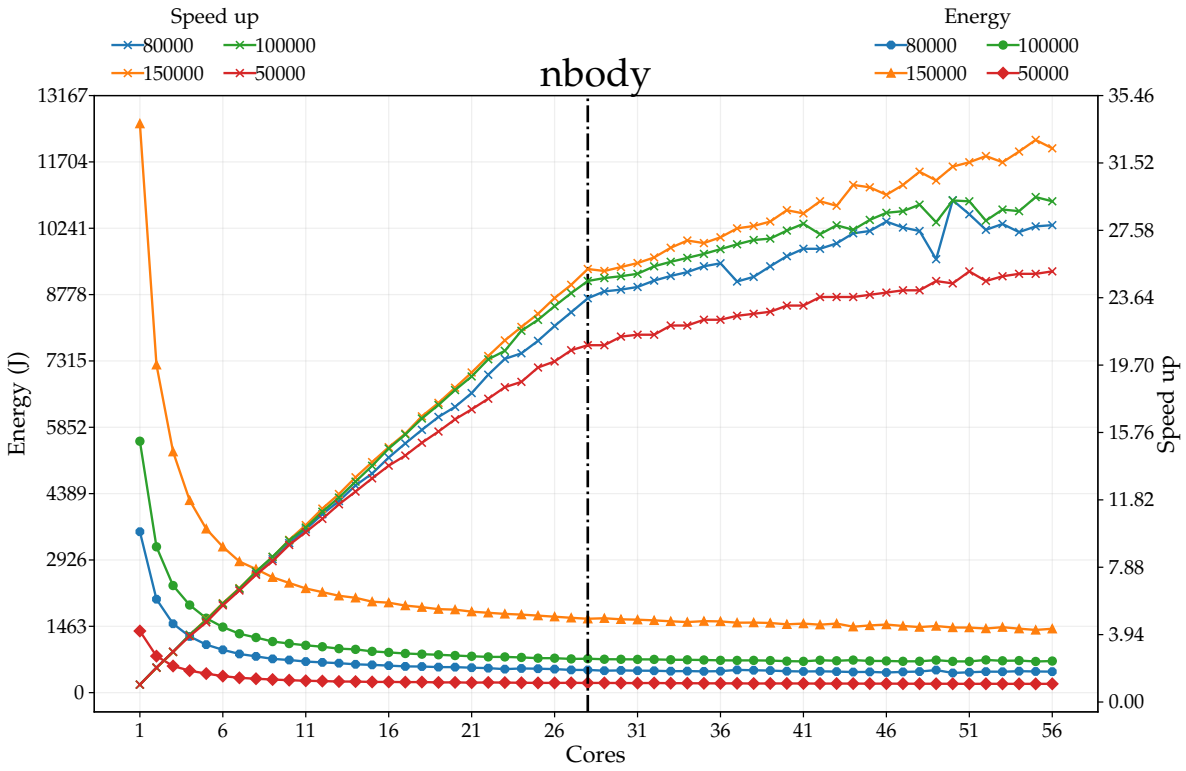


Figure 3.3: Nbody – *Energy vs. speedup* for numbers of bodies from 50K to 150K

Nbody For *Nbody* (Figure 3.3), we set the number of particles to range from *50K* to *150K*, providing sequential execution times from 12.15 to 389.4 s. We obtained good speedups when we used 28 physical cores, although speedup was lower for smaller problem sizes (22.39× for $n = 150K$ and 16.64× for $n = 50K$). Speedups continued to improve slightly when using virtual cores, but they were occasionally erratic. When we reached 56 virtual cores, we observed maximum speedups of 24.77× for $n = 150K$ and 14.98× for $n = 50K$. As we increased the core count, we also clearly saw corresponding reductions in energy usage, from a maximum of 11385.90 J on one core to a minimum of 1759.25 J on 56 cores at $n = 150K$ (6.47× improvement), and from a maximum of 1274.95 J on one core to a minimum of 272.72 J on 56 cores at $n = 50K$ (4.67× improvement). *Nbody* represents a class of problems that includes the processing of large data structures and complex arithmetic. It is both larger and far less regular than the previous two benchmarks. As before, energy usage reduced consistently with core count for a given problem size.

DFT For *DFT* (Figure 3.4), we set the *intervals* to range from *2K* to *6K* with a seed of *2*, providing sequential execution times from 652 to 35.30 s. We observed good

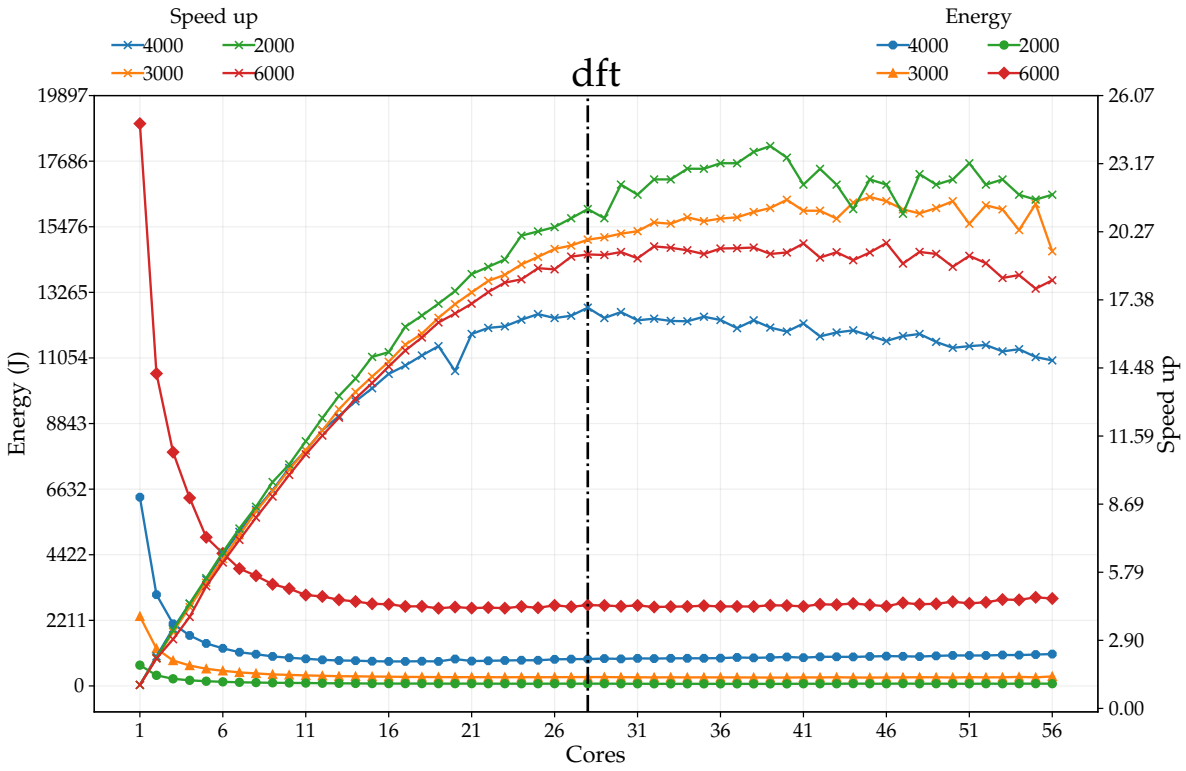


Figure 3.4: DFT – *Energy vs. speedup* for numbers of samples from 2K to 6K

speedups when we used 28 physical cores (19.31× for $n = 6K$ and 21.30× for $n = 2K$). In contrast to the previous examples, speedups were lower for some problem sizes, especially $n = 4K$, and we obtained our best speedups at $n = 2K$. We do not have a good explanation for the speedup anomaly on 20 cores when $n = 4K$. This is a problem for future study. Apart from the two smaller problem sizes, speedups degraded when using virtual cores, and even for the smallest sizes, we observed decreases in speedups when using the maximum number of cores. At 56 virtual cores, we observed maximum speedups of 18.30× for $n = 6K$ and 21.75× for $n = 2K$. Energy usage decreased from a maximum of 18593 J on one core to 2946 J on 56 cores at $n = 6K$ (6.31× improvement), and from a maximum of 700.38 J on one core to 76.16 J on 56 cores at $n = 2K$ (9.19× improvement). Although there was a good correlation with speedup for each problem size, in this case, energy usage flattened or even increased as we increased core count. This reflects the reduction in speedup that we observed.

NQueens For *NQueens* (Figure 3.5), we set the number of queens to range from *13* to *16*, providing sequential execution times from 573.1 to 55.5 s. Speedup followed a curve, reaching 10.07× for $n = 16$ and 4.61 for $n = 13$ on 28 physical cores. Beyond

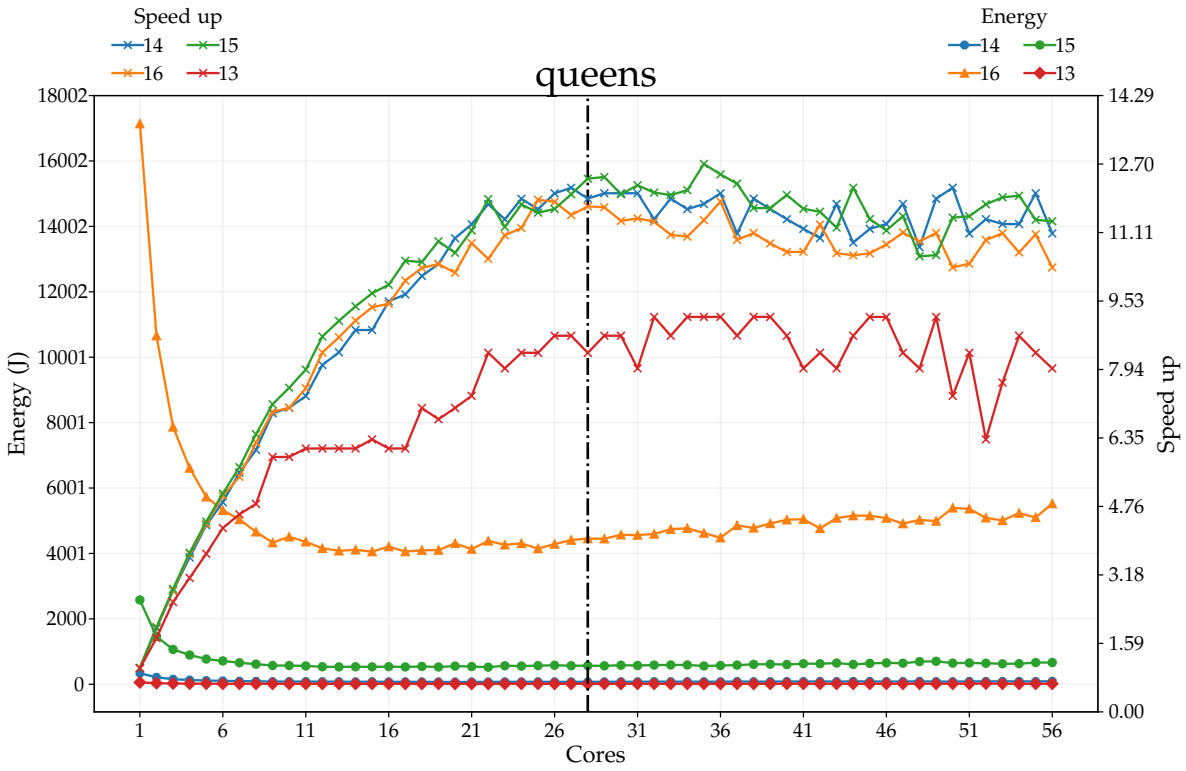


Figure 3.5: NQueens – *Energy vs. speedup* for numbers of queens from 13 to 16

28 cores, we observed only minor improvements, or even degradations in speedup (11.22× for $n = 16$ and 3.74× for $n = 13$ on 56 cores), and performance became erratic. Energy usage reduced from a maximum of 17318.85 J on one core to 4948.31 J on 56 cores for *16* queens, and from a maximum of 45.01 J on one core to 34.23 J on 56 cores for *13* queens. In the case of queens, energy usage increased with the use of virtual cores, reflecting the fact that we were using more cores. Thus, there was increased energy use; however, we did not obtain a performance advantage as execution time increases.

Matmult For *Matmult* (Figure 3.6), we set the *intervals* to range from *1500* to *3K* with block-based processing and 10 blocks per cycle, providing sequential execution times from 304.8 to 14.8 s. We obtained good speedups on the 28 physical cores (18.74× for $n = 3K$ and 12.9× for $n = 1500$). Speedup improvements did not directly correlate with increased problem size, however. We observed better speedups for $n = 2K$ than for $n = 2500$. Virtual cores gave some speedup improvements (20.5x for $n = 3K$ and 11.30× for $n = 1500$ on 56 cores), but there was a clear sawtooth effect, which may have been due to problem decomposition and mapping onto cores. As we

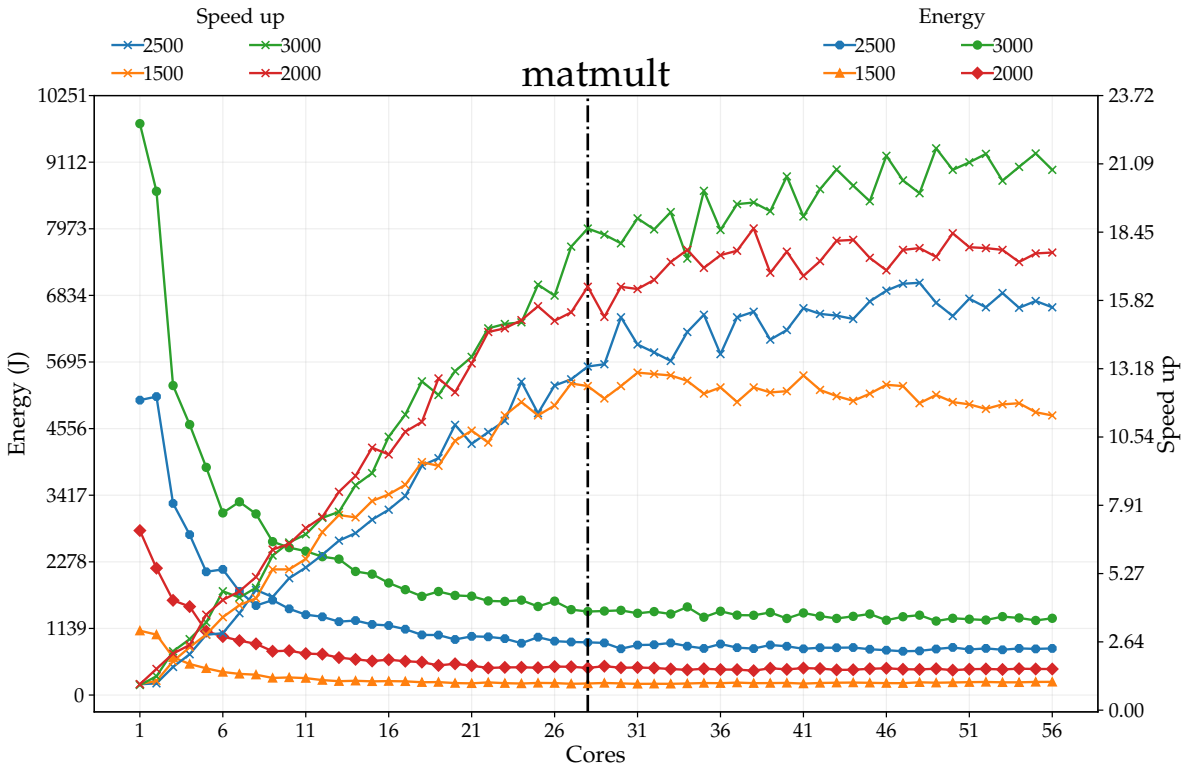


Figure 3.6: MatMult – *Energy vs. speedup* for matrix sizes from 1500 to 3000

increased the core count, we clearly saw corresponding reductions in energy usage, from a maximum of 9771.1 J on one core to 1312.5 J on 56 cores at $n = 3K$ (7.44× improvement) and from a maximum of 1100.5 J on one core to 225.5 J on 56 cores at $n = 1500$. (4.88× improvement). For this benchmark, energy usage was much more erratic than with the previous examples. Unlike *NQueens*, although energy usage improvements declined as the core count increases, there was no worsening in energy usage with larger core counts.

Minimax For *Minimax* (Figure 3.7), we set solution *intervals* to range from *6* to *12* with a depth size of *4*. This application quickly saturated the cores, reaching a peak 9.04× speedup on 16 cores for $n = 8$, levelling off to an 8.4× speedup for $n = 12$ and a 8.5× speedup for $n = 6$ on 28 physical cores. Then, it declined with increased numbers of virtual cores, apart from the very last results on 56 cores, where we saw an expected performance improvement. Energy usage reduced from a maximum of 24648.2 J on one core to a minimum of 5263.8 J on 15 cores for $n = 12$, and from a maximum of 223.4 J on one core to a minimum of 38.96 J on 14 cores (5.73× improvement) at $n = 6$, before increasing to about 54 cores and finally declining

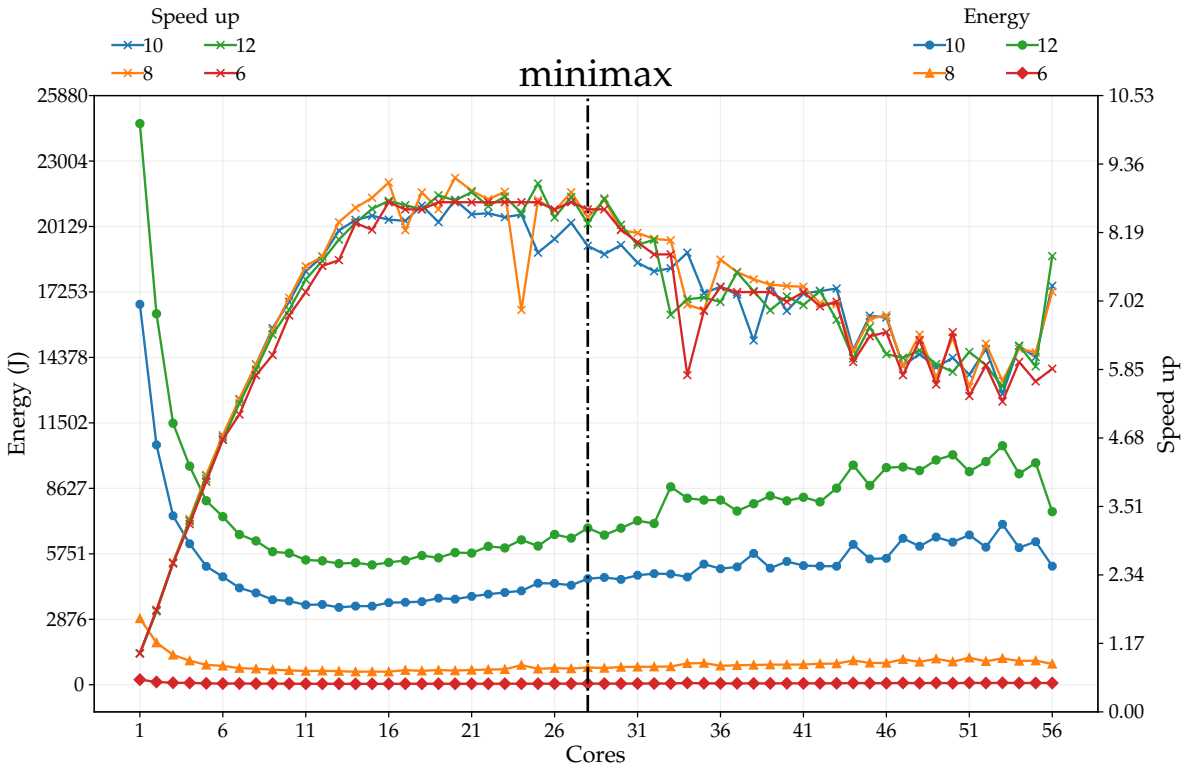


Figure 3.7: Minimax – *Energy vs. speedup* for numbers of solutions from 6 to 12 with depth of 4

slightly. There was once again a correlation with the speedup results obtained.

Ray The other two benchmarks, *Ray* (Figure 3.8) and *Partree* (Figure 3.9), showed different results. For *Ray* (Figure 3.8), we set the *detail/resolution* to range from *1300* to *1600*. Although we observed some speedups on 3–6 cores in all cases (up to 1.69× the sequential case), performance degraded thereafter to a real slowdown, and there was a curious reduction in performance when we used two cores; there was a discontinuity at 10 cores. As we increased the core count, we clearly saw the opposite effect of the former examples: energy usage increased rather than decreased as the number of cores increased. Ultimately, it reached a maximum of 9535.86 J for 1600 details on 55 cores. Investigation of our samples shows that this benchmark had significant levels of garbage collection. This behaviour reveals additional factors that may impact energy consumption.

In all cases measured, we obtained non-linear performance decreases when we used 28 physical cores (0.102 for $n = 1600$ and 0.108 for $n = 1300$). When using virtual cores, however, the speedups continued to decrease quickly as the number of cores

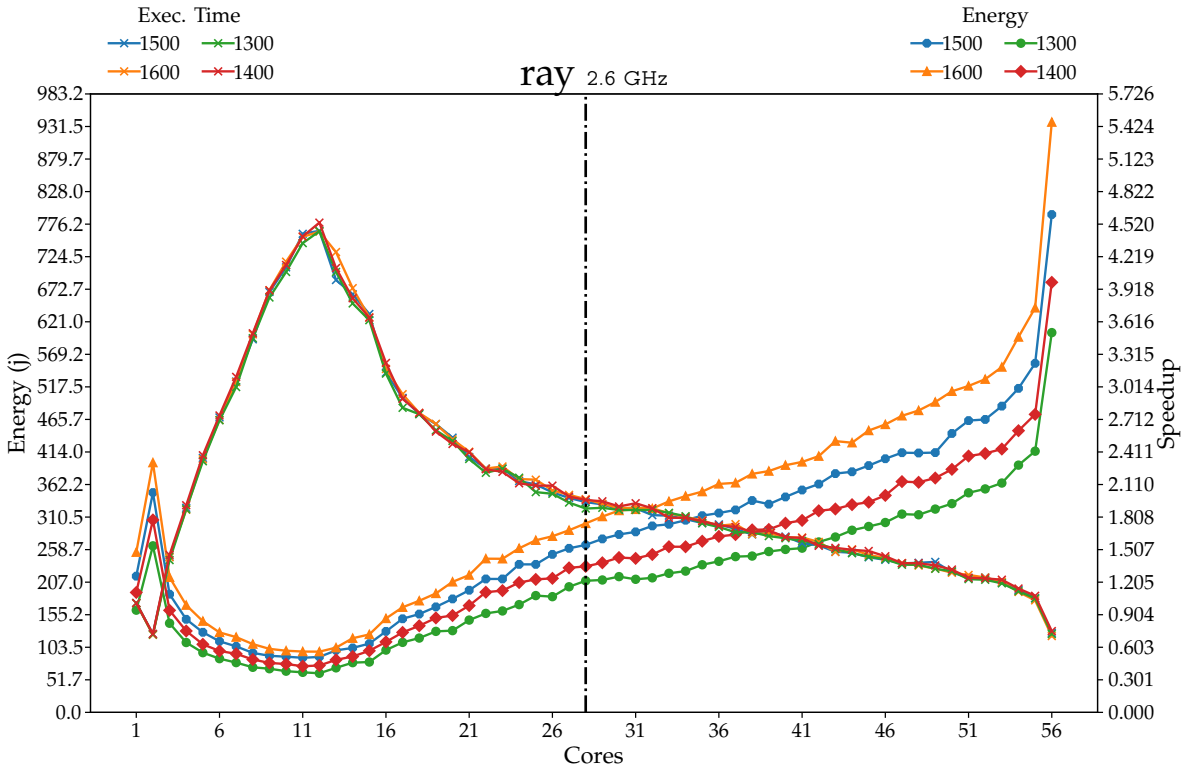


Figure 3.8: Ray – *Energy vs. speedup* for numbers of solutions from 1300 to 1600

increased. When we reached 56 virtual cores, we observed a maximum slowdown of 0.060 for $n = 1600$ and 0.062 for $n = 1300$). As we increased the core count, we also clearly saw the opposite effect of the former examples where energy usage increased with the number of cores.

Partree For *Partree* (Figure 3.9), we set the number of *tree nodes* to range from *600* to *800* with the number of *workers per node* set to 350. Although the performance continued to improve as we added physical cores (up to a maximum of 4.47× speedup for 28 cores for 650 tree nodes), the speedups decreased as more virtual cores were added. Although the energy usage initially decreased as we added cores (from 9285.73 J on one core for 800 tree nodes), beyond six cores, the energy usage plateaued before increasing beyond 12 cores. The energy usage was particularly unstable when virtual cores were used.

PRSA In *parallel RSA* (Figure 3.10), we saw a similar pattern to that which *Minimax* demonstrated. The inputs from $n = 2M$ to $n = 9M$ kept improving performance; consequently, energy consumption improved. From sequential execution forward, the

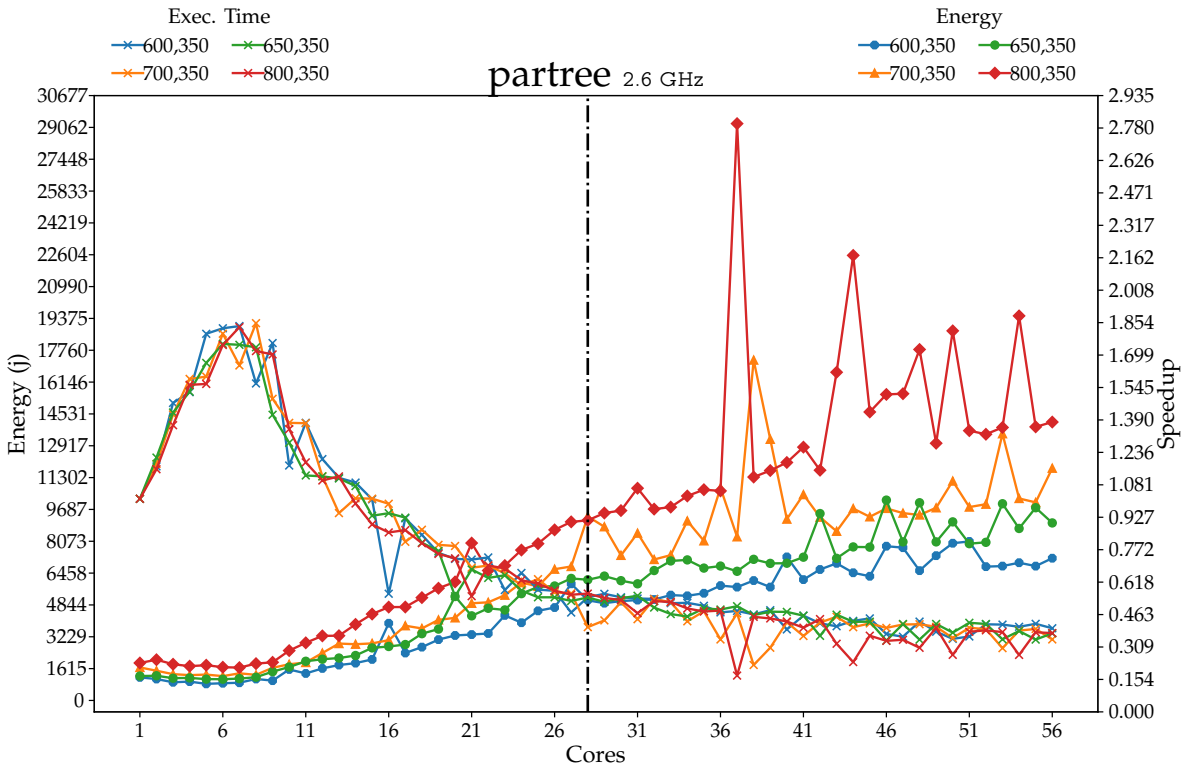


Figure 3.9: Minimax – *Energy vs. speedup* for numbers of solutions from 600 to 800 with *chunk* size of 350

benchmark improved energy consumption for all inputs with peak speedups ranging from 11.77× for $n = 2M$ up to 12.97× for $n = 9M$. The lowest energy consumption was achieved by $n = 2M$ at 82.91 J on 24 cores, which is understandable as other inputs were relatively more significant, such as $n = 6M$ at 244.22 J on 17 cores, $n = 8M$ at 325.10 J on 21 cores, and $n = 9M$ at 364.45 J on 18 cores. In the case of virtual cores, all input sizes underperformed after 28 cores; this effect was noticeable as the speedups declined after virtual cores were used.

QuickSort *Parallel Quicksort* (Figure 3.11) demonstrated a similar pattern to *Ray Tracing*. The performance gains were made early in the low core count at four-to-five cores for the inputs used. On average, the energy readings were lower for sequential execution at 28 cores, e.g. where $n = 500K$ was 48.88 J, $n = 1M$ at 123.10 J, $n = 3M$ at 457.9 J, and $n = 6M$ at 994 J. The energy readings improved as the speedup gains were made. The peak speedup was 1.47× for $n = 500K$ at five cores, followed by a 1.29×–1.41× speedup on four cores for the remaining inputs. Again a noticeable pattern

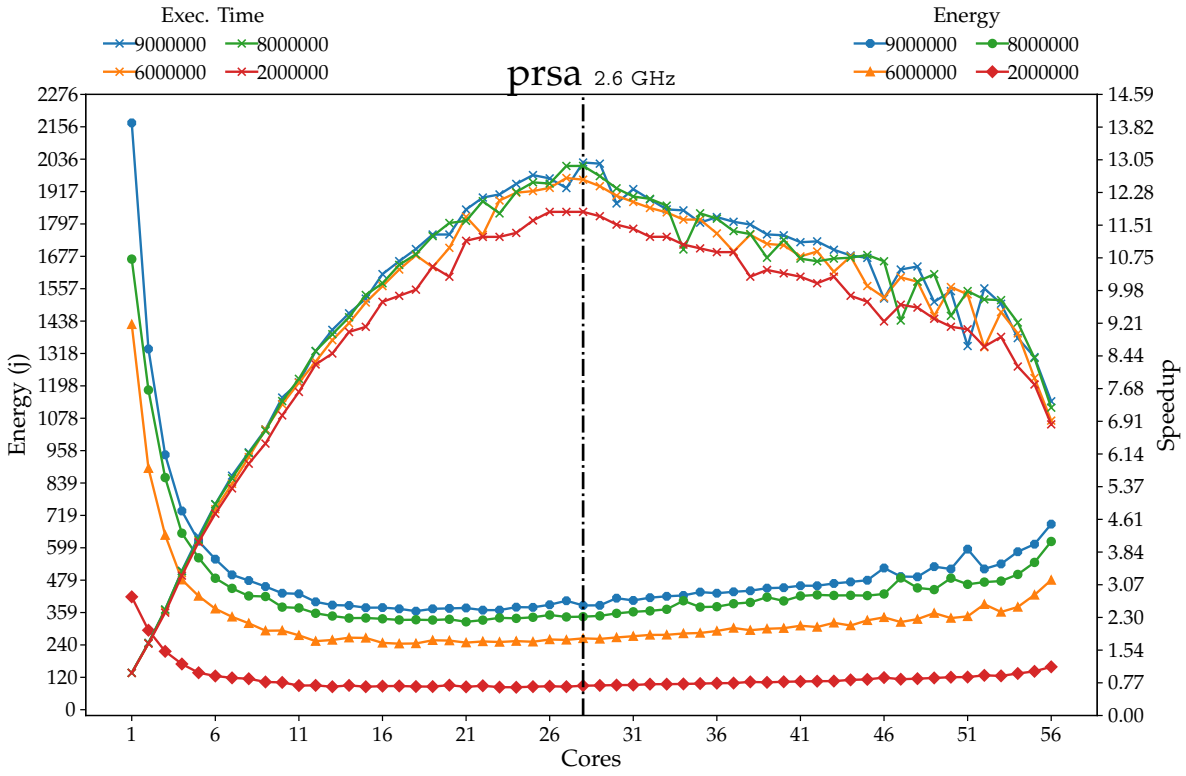


Figure 3.10: PRSA – *Energy vs. speedup* for numbers of Parallel RSA from 2 to 9M

was demonstrated after the number of cores grew such that the energy consumption only increased, peaking at 13659.09 J for $n = 6M$ on 49 cores.

3.2.3 C/C++ Benchmarks

Blackscholes *Blackscholes* (Figure 3.12) provides typical performance characteristics for a well-tuned and optimised parallel program. The different implementations of Pthreads, OpenMP and TBB improved energy consumption of 28 cores with minimal differences, with TBB at 1095 J, Pthreads at 1047 J and OpenMP at 1086.1 J. The same occurred at the lowest energy consumption point for all libraries at 55 cores. In all implementations, the speedups were consistent, where OpenMP achieved the best speedup at 6.34× on 56 cores, TBB achieved 6.14× on 56 cores and Pthreads achieved 6.30× on 55 cores.

Bodytrack The different implementations in *Bodytrack* (Figure 3.13) demonstrated improved performance with varying speedup gains. On 28 cores, TBB consumed

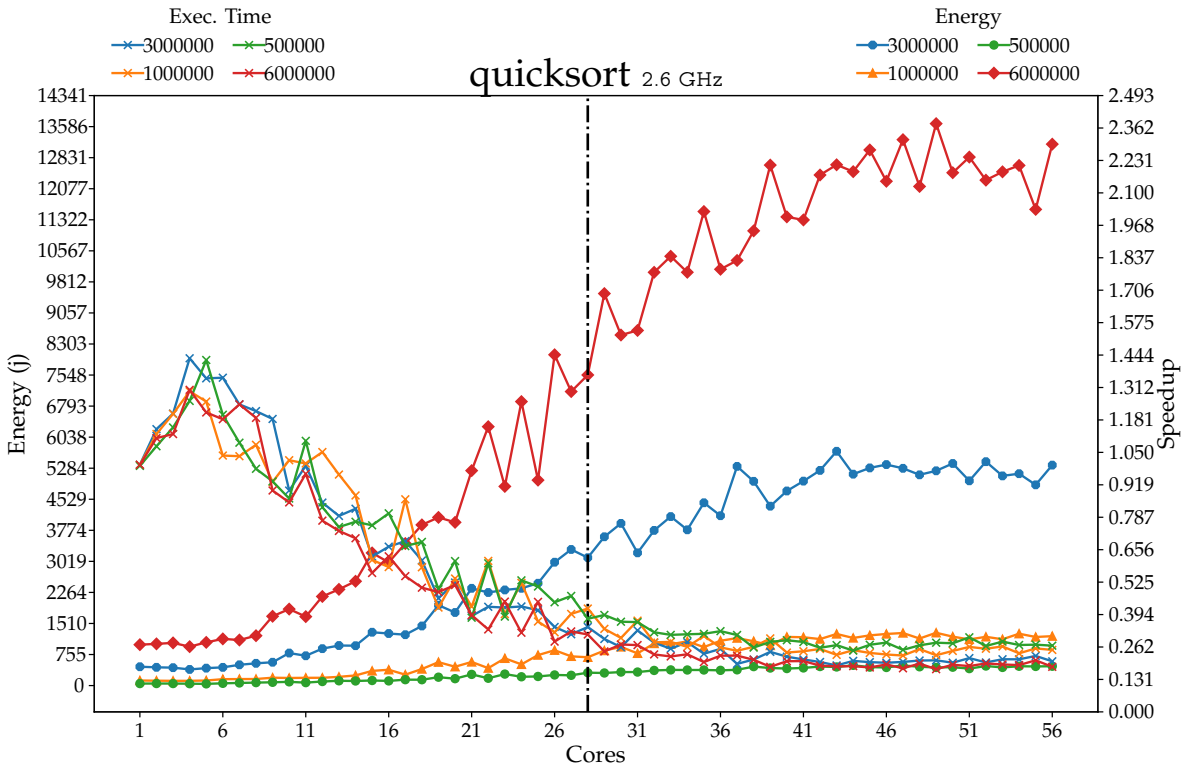


Figure 3.11: Quicksort – *Energy vs. speedup* for numbers of Quicksort from 500K to 6M

5555.54 J, Pthreads consumed 5770.04 J and OpenMP 5877.93 J. Until the 28th core, all parallel implementations had a near-linear improvement in speedup gains until virtual cores were employed. Then, both TBB and Pthreads had lower performance gains, with Pthreads having stagnant performance throughout, achieving 17.10× on 49 cores, which is the lowest energy consumption point at an average of 5711.71 J. TBB saw the best improvements on 54 cores with speedups peaking at 18.65× and energy consumption at 5361.30 J. Unusual behaviour was seen with OpenMP, where the speedups dropped sharply at 29 cores (28 physical, 1 virtual). The sharp drop was also reflected by a rise in energy consumption on core 29. Nevertheless, OpenMP continued to benefit from the virtual cores, peaking its speedup at 20.80× on 56 cores and consuming 5526.93 J.

Swaptions Although *Swaptions* (Figure 3.14) proved to be performer, its performance dropped at first when using the virtual cores. When considering speedups, the gains were made sequentially to a peak speedup at 26 cores at 16.23× with an energy consumption of 436.2 J using Pthreads. Its speedup was 16.22× at 27 cores with an

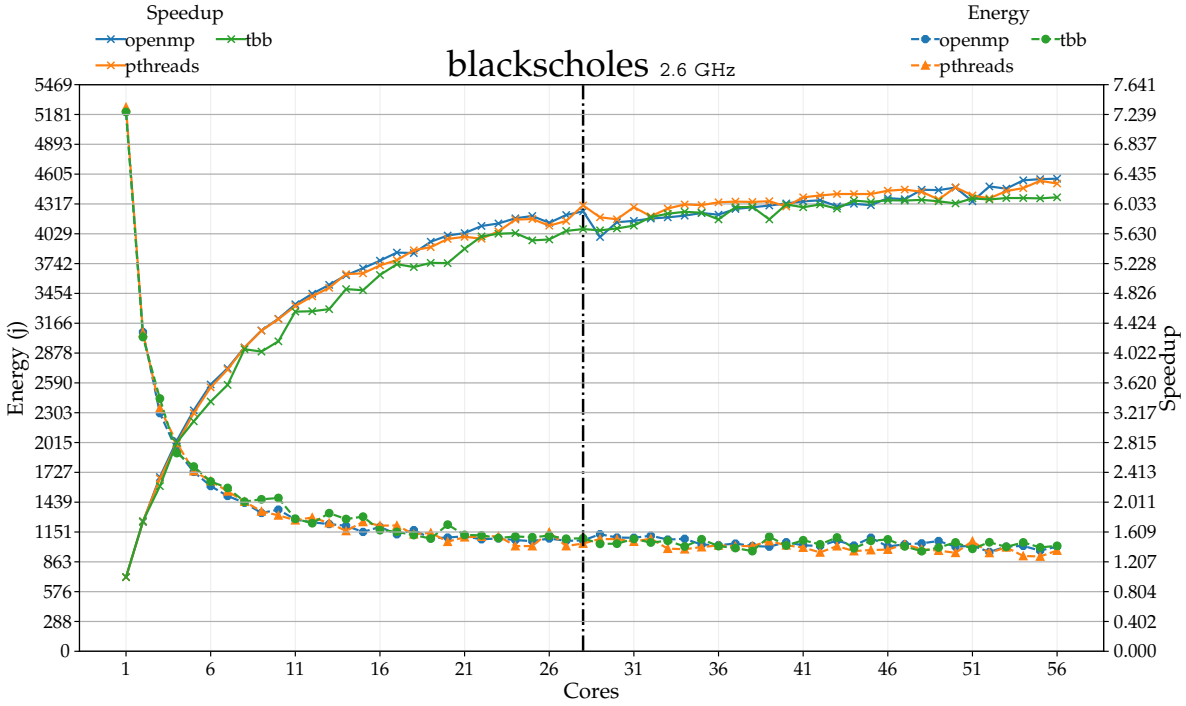


Figure 3.12: Blackscholes – *Energy vs. speedup* for Blackscholes for largest dataset on *Pthreads*, *OpenMP* and *TBB*

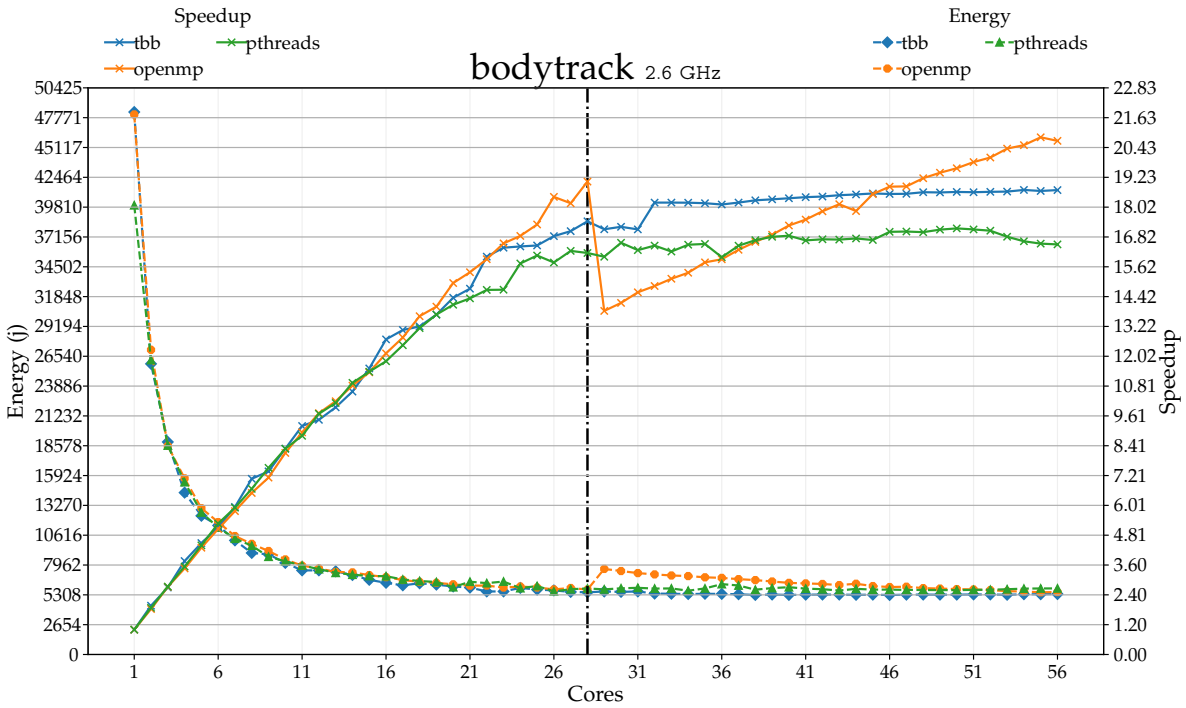


Figure 3.13: Bodytrack – *Energy vs. speedup* for Bodytrack for largest scene using *Pthreads*, *OpenMP* and *TBB*

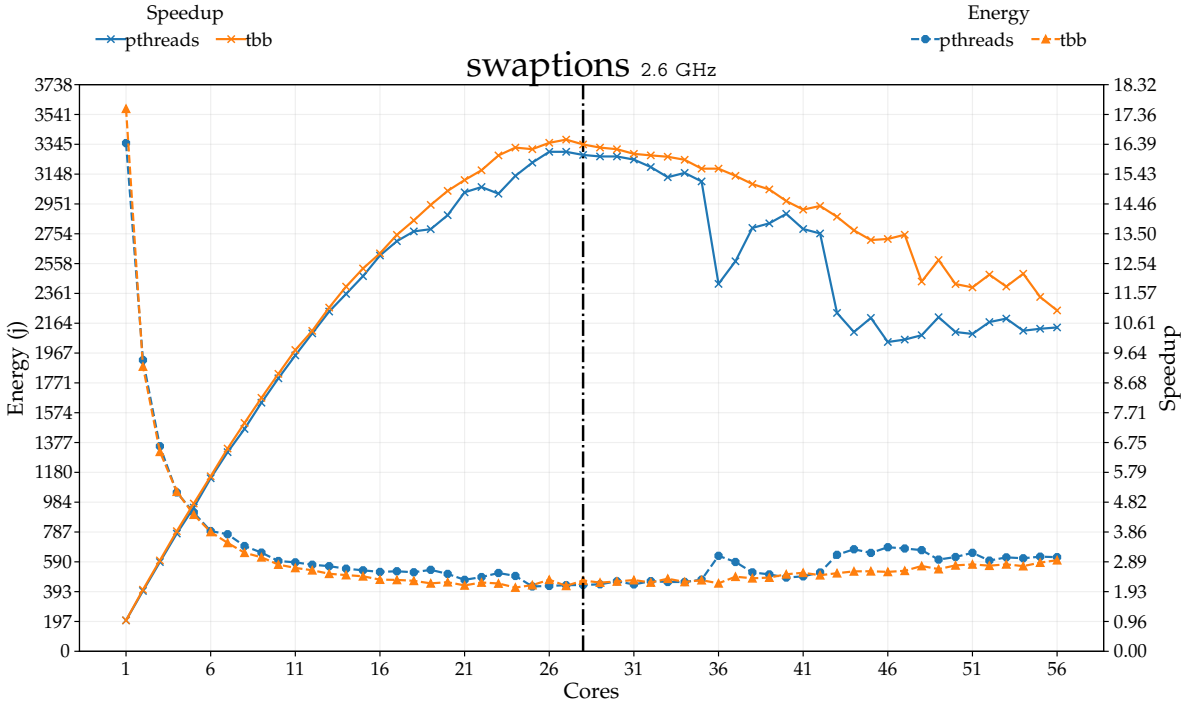


Figure 3.14: Swaptions – *Energy vs. speedup* for Swaptions for largest dataset using *Pthreads* and *TBB*

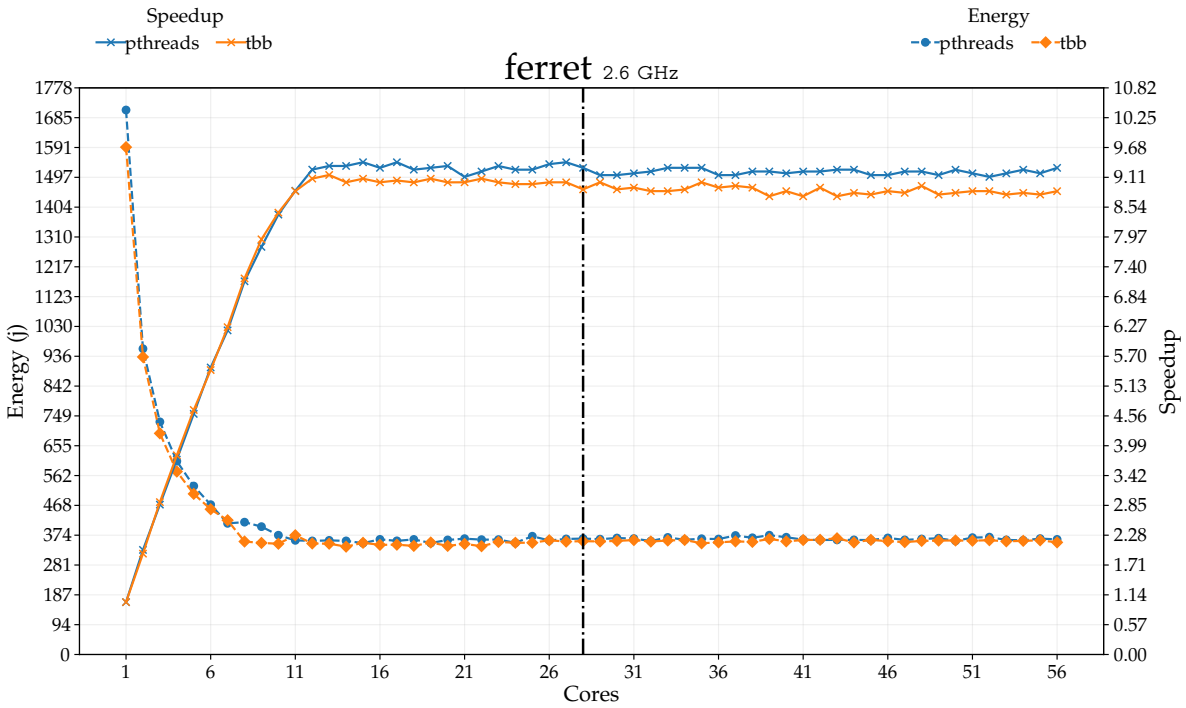


Figure 3.15: Ferret – *Energy vs. speedup* for Ferret for largest dataset using *Pthreads* and *TBB*

energy consumption of 430.13 J using TBB. The improvement in energy and speedups did not continue, though, as both parallel libraries had degraded performance after the introduction of virtual cores. Pthreads had unusually poor speedups with virtual cores, resulting in higher energy readings, e.g. at 56 cores, 619.04 J, speedup declining to 10x.

Ferret With *Ferret* (Figure 3.15), we observed a consistent and improved performance as the number of cores increased. Energy consumption and speedups both showed steady improvement. However, the improvements appeared to plateau at 19 and 18 cores for Pthreads and TBB, respectively. The lowest energy consumption varied between those two, where TBB achieved the least energy consumed at 13 cores with 345.06 J and 9.20× speedup, whereas Pthreads could not reach its lowest energy until 17 cores, consuming 356.94 J with 9.33× speedup. Virtual cores had minimal effect on speedups with minor drops, e.g. 56 cores consuming 365.75 J with a speedup of 9.08× for Pthreads.

Summary of results: Although all benchmarks showed reductions in energy with the use of increased core counts, there were some significant differences in behaviour. As discussed, it is clear that there was a tight connection between speedup/execution time and energy usage (i.e. energy $\propto$ speedup in all cases measured). Different applications may have characteristics that impact energy usage, but there is not necessarily a direct correlation with increasing problem sizes within applications.

3.2.4 Effect of Core Affinity

To study the effects of scheduling consistency, we tested the impact of using GHC's *core affinity* flag, `-qa`, on each benchmarks. This flag pins parallel threads to specific cores. We anticipated that using core affinity may improve energy usage by avoiding costly cache flushes and context switches. However, we observed speedups and energy consumptions that were broadly similar, with and without affinity. Moreover, both performance and energy usage exhibited irregular behaviours and were more erratic when core affinity was enabled. For example, although Nbody demonstrated lower energy on average when sampled with core affinity, the total speedup gains were much lower compared with the sample without core affinity. Apart from NQueens, the single-core/sequential execution had a lower energy consumption than the sample without core affinity, which may have been related to anomalies in the

sample causing the lower average. The complete plots for the core affinity samples can be found in Appendix C.

Although it seems counter-intuitive, we concluded that there was no overall energy advantage to using core affinity settings for the benchmarks. We conjecture that the negative effect on scheduling performance outweighs the benefits of reduced cache flushes and context switches. Therefore, the use of core affinity was avoided in upcoming experiments.

3.3 Energy Models Construction

In the benchmark sampling phase, the samples collected contained a relatively large number of features, and most were shared. However, several were dynamically generated and were benchmark-specific. For example, one benchmark triggered a set of Intel Advanced Vector Extensions 512-bit (*AVX-512*) instructions, whereas in another sample program, only one of the streaming SIMD extensions (*SSE*) instruction sets was used. Another example occurred when a program used a system call specific to an input/output task where other programs do not need such calls. The dynamic nature of program features made it challenging to understand shared features, especially when considering that certain features are specific to the ISA of a given CPU.

When identifying the essential features, we must consider the context in which we would plan to construct the model. For example, a model cannot have a negative energy prediction, as it is meaningless. Therefore, controlling model coefficients to retain positive weight values will ensure that new samples do not make negative predictions.

In classical multiple regression, the goal is to construct a model that minimises the error of fitted or predicted values, i.e. samples of new unseen programs. In minimising the error, we need a model that is as accurate as possible. Although multiple regression can provide the means of constructing a model with a large set of variables, the overall model is set to predict samples without bounds, i.e. the values can take any number from the range (*-infinity–infinity*). It is possible, however, to avoid unwanted effects with certain regression transformations, e.g. *logarithmic* functions. When multiple regression was applied to the dataset collected from the

Haskell and PARSEC modelling sets, the model failed to deliver acceptable accuracy. When examining the ability of a linear model to provide predictions, we construct a linear model using the complete set of features from the modelling sets of PARSEC and Haskell, even when using the complete set of features in the model. The results of all samples from the prediction set (TBB, OpenMP, Pthread) and Haskell have shown a substantial inability for such models to provide any helpful energy estimations, as can be seen in all the plots in Appendix F. With a few exceptions, such as GHC's Prime Decomposition and Spectralnorm 8.5k, the provided energy estimates were off by high margins and, in most cases, illogical, as in the case of negative energy consumption. We use several metrics to assess multiple regression models, but the most essential is R^2 (Equation 3.3). The *sum squared regression* (SSR) refers to the sum of squared difference between estimated or fitted values, $\hat{y}_i$ and the observed values of the response across the sample y_i. The *total sum of squares* (SST) is the sum of squared differences between the observed values of y_i and the mean or average across $\bar{y}$. When dividing both SSR and SST, we obtain a percentage of R^2, which helps explain the *goodness of fit* of a model, i.e. *how much of the response variation can be explained by the features*. After multiple regressions, we had R^2 = 14% in the C/C++ set and R^2 = 11% for the Haskell set, suggesting that multiple regression could not infer the relationship between the response, y, and the features or variables, x. Another metric to consider regression is RMSE, defined as the *root mean-squared error* ,and described by (Equation 3.1). RMSE is the square of all fitted or predicted values, $\hat{y}$, subtracted by observations, y, and divided by the total points in the sample, which are then summed and square-rooted. RMSE provides a good measure of prediction errors between the model and the observations (actual data points). In the multiple regression model, the RMSE value of the C/C++ dataset was 648570290, whereas that of the Haskell dataset was 5412.67. The total number of variables used in constructing the multiple regression model was 280 with 280 observations for the C/C++ dataset and 264 with 1092 observations for the Haskell dataset. The entire set of features used to construct the various models are available in the online appendix [1].

Other algorithms, e.g. non-negative least squares, can achieve beta coefficient control while minimising model errors. The following sections provide an overview of regression models that demonstrated a robust and consistent behaviour with multiple parallel programs from the benchmarks explored in Chapter 2. The models

[1] github https://github.com/ymg/thesis

will be constructed using the caret package [57] in the R language.

$$\text{RMSE} = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}} \tag{3.1}$$

$$R^2 = 1 - \frac{\text{sum squared regression (SSR)}}{\text{total sum of squares (SST)}}, \tag{3.2}$$

$$= 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}. \tag{3.3}$$

3.3.1 Non-Negative Least Squares (NNLS)

NNLS [60] is a regression model belonging to the family of least squares. The NNLS model minimises the error values of fitted samples while having only positive β coefficients in the regression model 3.4. X refers to a matrix of variables and observations, β is the coefficient value and $\hat{\beta}$ is a vector of coefficients with a constraint of $\hat{\beta} \geq 0$. y a vector of response values. $\|.\|_2$ is the L_2 or Euclidean norm of a matrix. We applied NNLS on the datasets collected from Haskell and C/C++ benchmarks. We used 280 variables and 280 observations for the C/C++ modelling dataset and 264 for 1092 observations for Haskell's modelling dataset. NNLS produced a sufficiently accurate model with an R^2 of 97.3% and an RMSE of 561.28 using the Haskell dataset. The C/C++ dataset had an NNLS R^2 of 96.3% and an RMSE of 942.15. The most significant values varied between datasets, but many of the provided variables did help improve the models. Therefore, the majority were set to zero by the NNLS algorithm. When considering the remaining variables for each dataset, we saw only a few influencing the models. In Table 3.1 on page 3.1, we see the results from the Haskell NNLS model. Most variables with any weight were from the Intel SDE. Therefore, the system calls did not affect the overall model construction. The most significant variable in the model was the execution time or `Total.Time`. This provides a clear correlation as the longer the program executes, the longer it consumes energy. Other variables, such as `MOVHPD` and `MOVLPD`, are instructions for moving *high/low packed double-precision floating-point values*, which are part of the Streaming SIMD Extensions 2 (SSE2) instruction extension; these were more important than other variables.

In contrast to Haskell, the C/C++ NNLS model (Table 3.2 on page 3.2) contained more variables that predicted the overall energy consumption. Along with Total.Time, the `MMAP` system call had a higher correlation with lower energy and other system calls, such as *Fast Userspace Mutex* (`futex`) and read. `MMAP` is a system call that maps files or devices to a memory address associated with an *Input/Output* (I/O) operation. `Futex` is one way to apply a user space locking mechanism, which is related to parallel programs accessing a single resource. `Read` is a system call that reads bytes from a file descriptor, which is a method of reading from a file in a Linux-based system. It is also worth mentioning that `SYSCALL` is an aggregate count of all system calls made by the program, unlike the specific count of Perf where system calls are broken down into specific types e.g. `syscalls.sys_enter_futex`. Since Perf and Intel SDE are two different tools, it may be difficult to cluster exact features from both tools without looking at specific sampling process, and even then each tool serves a specific purpose like Perf providing system calls during execution while Intel SDE is an emulation tool that provides x86 instruction counts.

$$\hat{\beta} = \text{argmin}\|y - X\beta\|_2^2 \tag{3.4}$$

3.3.2 Elastic-Net Regularised Generalised Linear Model (GLMNET)

The Elastic-Net [111] regression model is one that applies a penalty to model features that cannot help improving model accuracy. It combines two forms of regression models, the first being the least absolute shrinkage and selection operator (LASSO) [103], also referred to as L_1 regularisation. LASSO minimises the effect of coefficients that provide low contributions to a model (near zero or zero). LASSO does this by minimising the sum of the absolute values of the coefficients in a model. Ridge [40] (also known as L_2 regularisation) also performs regression with a shrinkage effect on the coefficients, which penalises the sum of squares caused by regression (SSR) (Equation 3.5) where $\hat{y}_i$ is the fitted/predicted value for a given sample and the mean of the response, $\bar{y}$. Elastic-Net balances LASSO and Ridge using λ (penalty factor) and an α (alpha factor) to control when coefficients are set to have no effect in a model while controlling the absolute value of the coefficients.

Haskell	
Variable	Weight
Total.Time	$3.103042 \cdot 10^{+01}$
X.iprel.write	$3.247744 \cdot 10^{-06}$
X.mem.write.16	$1.875508 \cdot 10^{-06}$
X.isa.ext.SSE	$1.103624 \cdot 10^{-09}$
X.isa.set.SSE42	$1.319413 \cdot 10^{-10}$
X.category.LOGICAL_FP	$1.144457 \cdot 10^{-08}$
X.nop.ilen.9	$2.875137 \cdot 10^{-06}$
X.nop.ilen.10	$2.477468 \cdot 10^{-07}$
X.elements_i8_16	$7.994750 \cdot 10^{-10}$
X.dataxfer_fp_double_1	$1.607577 \cdot 10^{-09}$
CMOVNS	$3.940728 \cdot 10^{-06}$
CPUID	$1.407745 \cdot 10^{-11}$
IMUL	$3.011265 \cdot 10^{-09}$
INC_LOCK	$4.148062 \cdot 10^{-09}$
JS	$1.915928 \cdot 10^{-09}$
MOVAPS	$6.849389 \cdot 10^{-10}$
MOVDQU	$6.865262 \cdot 10^{-10}$
MOVHPD	$1.662180 \cdot 10^{-05}$
MOVLPD	$7.002552 \cdot 10^{-05}$
MOVSD_XMM	$1.305229 \cdot 10^{-09}$
PCMPEQB	$6.943468 \cdot 10^{-05}$
PSUBB	$1.218494 \cdot 10^{-08}$
SETNLE	$1.884485 \cdot 10^{-06}$
XGETBV	$6.586630 \cdot 10^{-09}$
XSAVEC	$2.910658 \cdot 10^{-09}$

Table 3.1: Haskell variable weights using NNLS, 24 out of 264 variables

C/C++	
Variable	Weight
Total.Time	$3.110768 \cdot 10^{+01}$
syscalls.sys_enter_read	$2.885613 \cdot 10^{-04}$
syscalls.sys_enter_futex	$2.459005 \cdot 10^{-05}$
syscalls.sys_enter_rt_sigprocmask	$8.933374 \cdot 10^{-07}$
syscalls.sys_enter_mmap	$2.866033 \cdot 10^{-02}$
cycles	$5.791863 \cdot 10^{-11}$
cache.misses	$6.013455 \cdot 10^{-12}$
bus.cycles	$9.499066 \cdot 10^{-12}$
X.mem.write.56	$1.210414 \cdot 10^{+00}$
X.isa.ext.X87	$1.619357 \cdot 10^{-06}$
X.nop.ilen.6	$7.189445 \cdot 10^{-08}$
CQO	$1.285586 \cdot 10^{-09}$
INC	$4.910007 \cdot 10^{-08}$
JNB	$6.016429 \cdot 10^{-10}$
NEG	$8.978607 \cdot 10^{-09}$
RDTSC	$1.949810 \cdot 10^{-09}$
SBB	$5.984788 \cdot 10^{-09}$
SETNB	$2.954172 \cdot 10^{-08}$
SYSCALL	$1.618947 \cdot 10^{-09}$
VSUBSD	$1.995836 \cdot 10^{-09}$
XGETBV	$1.745517 \cdot 10^{-09}$

Table 3.2: C/C++ variable weights using NNLS, 20 variables out of 280

In Tables 3.3 and 3.4 on pages 65 and 66, the variable weights present a different story to NNLS. GLMNET shows fewer features being used for the C/C++ model, where no Perf metadata improve the model, i.e. the system calls were of no value, unlike NNLS. Haskell offered a different story as the model features were comparable to those of NNLS, where some instruction extensions were chosen in both: NNLS and GLMNET. When considering the model assessment metrics, i.e. RMSE and R^2, GLMNET provided a set of lambda and alpha factors to improve the model over several iterations. For C/C++, the best-tuned λ value was 394.29 with an α of 0.55, which provided an RMSE of 564.30 and an R^2 of 99.7%. The Haskell GLMNET model had a λ value of 68.31 and 0.55 for an α factor, resulting in an RMSE of 803.88 and an R^2 of 93.9%.

$$\sum_{i=1}^{n} \left(\hat{y}_i - \bar{y}\right)^2. \tag{3.5}$$

C/C++	
Variable	Weight
Total.Time	$2.893062 \cdot 10^{+01}$
X.mem.read.1	$2.572043 \cdot 10^{-09}$
X.mem.read.4	$1.271286 \cdot 10^{-10}$
X.mem.read.16	$2.667536 \cdot 10^{-08}$
X.mem.write.4	$1.292971 \cdot 10^{-09}$
X.mem.read	$1.367228 \cdot 10^{-10}$
X.mem.write	$8.471186 \cdot 10^{-10}$
X.mem	$1.367868 \cdot 10^{-10}$
X.isa.ext.BASE	$2.267633 \cdot 10^{-11}$
X.isa.ext.LONGMODE	$3.259578 \cdot 10^{-10}$
X.isa.set.FAT_NOP	$5.609084 \cdot 10^{-09}$
X.category.LOGICAL	$7.900707 \cdot 10^{-11}$
X.ilen.5	$1.426185 \cdot 10^{-10}$
X.one.memops	$2.407249 \cdot 10^{-11}$
X.scale_1	$8.105242 \cdot 10^{-11}$

Table 3.3: C/C++ variable weights using GLMNET, 14 variables out of 280

3.3.3 Parallel Random Forest (parRF)

Random forest [11] is a machine learning algorithm that can be applied to regression and classification problems. The algorithm handles high variation in response variables, where some samples may have unexpected estimates of y values given a set of input. The algorithm creates smaller subsets from all data points in the dataset, followed by constructing decision trees using the subset data points. For example, the variables/predictors form several trees consisting of nodes, with each representing a variable with a condition on the value of the given variable. The algorithm then generates a given number of trees, with each having an output corresponding to the response variable or y. When applied to regression, the output of all the trees is averaged. The algorithm has two parameters with which to optimise results: `mtry` and `ntree`. The `mtry` splits the number of variables in a dataset, `ntree` represents the number of trees generated to construct a model. The Haskell model had an R^2 of

Haskell	
Variable	Weight
Total.Time	$3.069262 \cdot 10^{+01}$
X.iprel.write	$3.202282 \cdot 10^{-06}$
X.mem.read.8	$8.816581 \cdot 10^{-11}$
X.mem.write.16	$6.910736 \cdot 10^{-06}$
X.mem.read	$4.858719 \cdot 10^{-10}$
X.isa.set.SSE	$9.329616 \cdot 10^{-10}$
X.category.LOGICAL	$3.242863 \cdot 10^{-10}$
X.category.POP	$1.371262 \cdot 10^{-10}$
X.ilen.6	$6.738369 \cdot 10^{-11}$
X.nop.ilen.5	$-6.963154 \cdot 10^{-07}$
X.nop.ilen.9	$3.263356 \cdot 10^{-06}$
X.scalar.simd	$3.873391 \cdot 10^{-12}$
X.legacy.prefixes.3	$1.734396 \cdot 10^{-10}$
X.disp_only	$1.584069 \cdot 10^{-10}$
X.dataxfer_fp_single_1	$8.987955 \cdot 10^{-13}$
X.dataxfer_fp_single_4	$-2.301403 \cdot 10^{-06}$
DIVSD	$-1.567484 \cdot 10^{-06}$
INC	$-3.445704 \cdot 10^{-08}$
JB	$-8.148723 \cdot 10^{-11}$
LEAVE	$-1.746571 \cdot 10^{-12}$
MOV	$-7.946631 \cdot 10^{-11}$
MOVLPD	$2.224406 \cdot 10^{-05}$
NEG	$-2.272617 \cdot 10^{-08}$
PSUBB	$1.853427 \cdot 10^{-08}$
RET_NEAR	$2.033985 \cdot 10^{-09}$
TEST	$-3.138661 \cdot 10^{-09}$
XGETBV	$3.132138 \cdot 10^{-09}$
XSAVEC	$3.394814 \cdot 10^{-10}$

Table 3.4: Haskell variable weights using GLMNET, 27 variables out of 264

87.7% with an RMSE of 1389.03 for a `mtry` 133. The C/C++ model had an R^2 of 96.0% and an RMSE of 1378.56 for the best tune, which was `mtry` 280.

Random forests provide an importance metric to help users understand the significance of predictors in a dataset. The Importance values of the predictors/variables for each dataset were far from similar. The C/C++ dataset showed that `Total.Time` was the highest among predictors, followed by `X.legacy.prefixes.1`, an instruction extension for prefixes for a group of instructions. For the Haskell model, `X.mem.read.1` was the most crucial predictor, whereas Total.Time came in second.

3.4 Model Predictions

During the prediction phase, we used benchmarks of the third category, which have an equivalent implementation in both languages, enabling the comparison of predictive abilities in Haskell and C/C++ model construction.

C/C++ sampling was performed for all parallel frameworks: TBB, OpenMP and Pthreads. Benchmark sampling was done in a single pass for energy, Intel SDE instruction count and Perf's system calls. During the sampling process, we sampled energy at different CPU frequency levels: 1.2, 1.4, 1.8, 2.1 and 2.6 GHz. In all samples, we predicted energy consumption for cores 1^{st}–28^{th}. Virtual cores theoretically cannot draw more power than the CPU design specification suggests, which translates to energy consumption. We used mean absolute percentage error (MAPE) to assess the prediction results and the plots showing the predictions for each input and benchmark. MAPE in (Equation 3.6) is the sum of absolute values of individual energy points minus y_i from the predicted/fitted energy values, $\hat{y}_i$, divided by actual observation, y_i, and converted to a percentage based on sample size. The complete energy heatmap profiles can be found in Appendix D.

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right|. \tag{3.6}$$

3.4.1 Haskell

In the Haskell sample, we observed a variation at all levels when applying the predictive models. In Table 3.5 on page 75, we see MAPE accuracy levels for NNLS at

the lowest point among all other models. Although some benchmarks showed irregularity in energy consumption, e.g. Binarytrees and Prime Decomposition, NNLS predictions were near the actual consumption for some benchmarks, such as Spectral norm. GLMNET and Parallel Random Forest showed similar prediction patterns to NNLS in some benchmarks, such as Prime Decomposition and Binarytrees. However, in Fasta, NNLS clearly provided more realistic predictions, although the energy consumption numbers were low. In Spectral-norm, GLMNET failed to provide any valuable predictions. The model produced negative energy consumption, which may have been caused by the negative coefficients having a higher model impact.

When we consider the prediction figures with MAPE values, especially those with values higher than 100%, we see that the difference is insignificant. For example, in all Fasta samples in Figures 3.20 to 3.23 pages 71 to 72, the predictions were off by several tens of joules, indicating a considerable error. Nonetheless, when we consider that the actual energy consumption was near 50 J, both NNLS and GLMNET were able to predict the energy consumption curve, suggesting it may be possible to improve the model. Another issue is related to the peculiar spikes, as in Binarytrees, with input size of 22 in Figure 3.18 Page 70, which showed inconsistent energy prediction at point 24. The sharp rise caused an error that may have contributed to the overall MAPE increase.

3.4.2 C/C++ (OpenMP, TBB and Pthreads)

The C/C++ sample demonstrated varying levels of accuracy across parallel libraries. In Table 3.8 Page 85, except for *Spectral norm*, the POSIX thread implementations demonstrated the most durable ability to minimise mean error. Pthread MAPE values, compared with TBB and OpenMP, had overall lower error averages (Tables 3.6 and 3.7 Pages 83 and 84). For example, random forest demonstrated high error values in all three implementations; however, Pthreads was lowest apart from Binarytrees in OpenMP. Notably, in all predictions, there was difficulty in predicting low-energy consumption cases. Other internal library implementations, e.g. OpenMP and TBB, may have scheduling and additional parallelism capabilities whose controls affect the total instruction counts, resulting in inaccurate predictions.

The Figures 3.38 to 3.41 pages 81 to 86 show various levels of accuracy. The

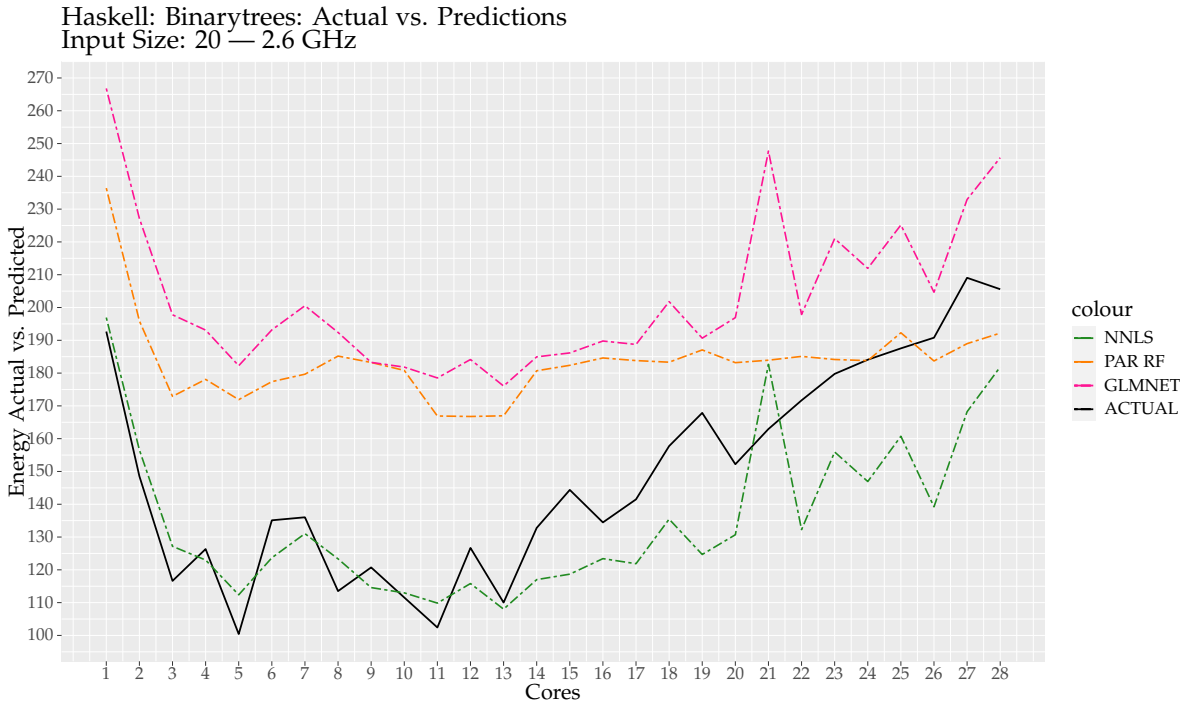


Figure 3.16: Binarytrees – *Energy consumption* for input 20 for models *NNLS*, *parRF* and *GLMNET*

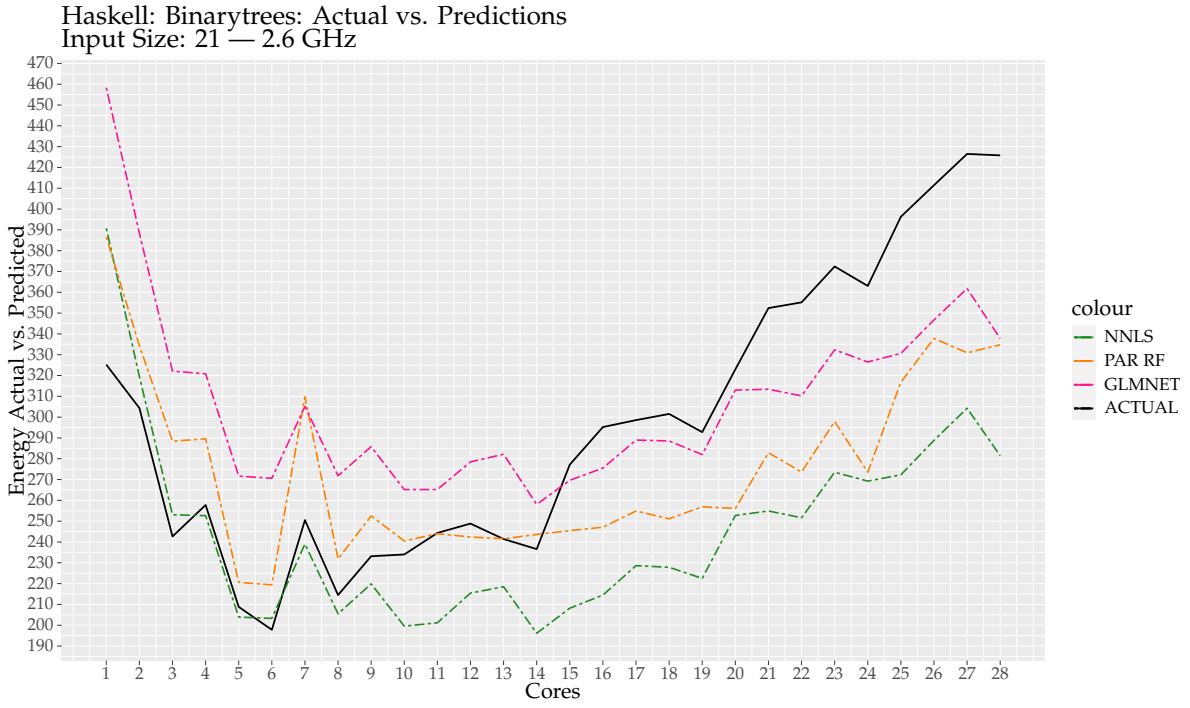


Figure 3.17: Binarytrees – *Energy consumption* for input 21 for models *NNLS*, *parRF* and *GLMNET*

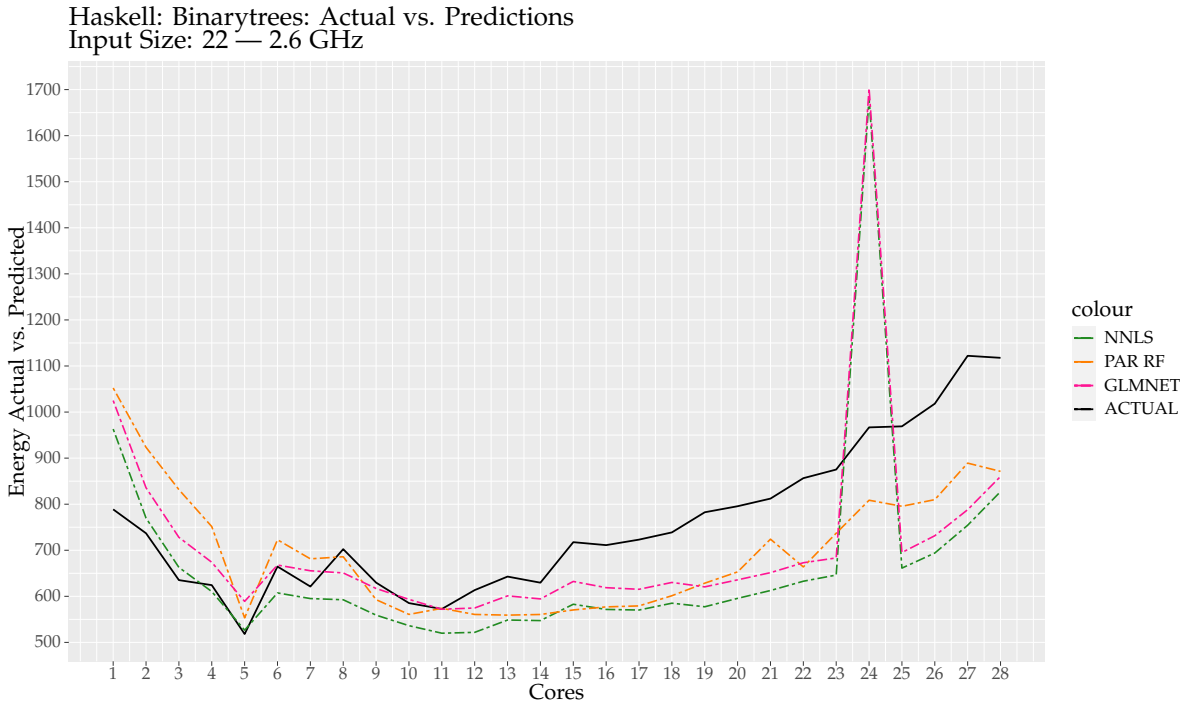


Figure 3.18: Binarytrees – *Energy consumption* for input 22 for models *NNLS*, *parRF* and *GLMNET*

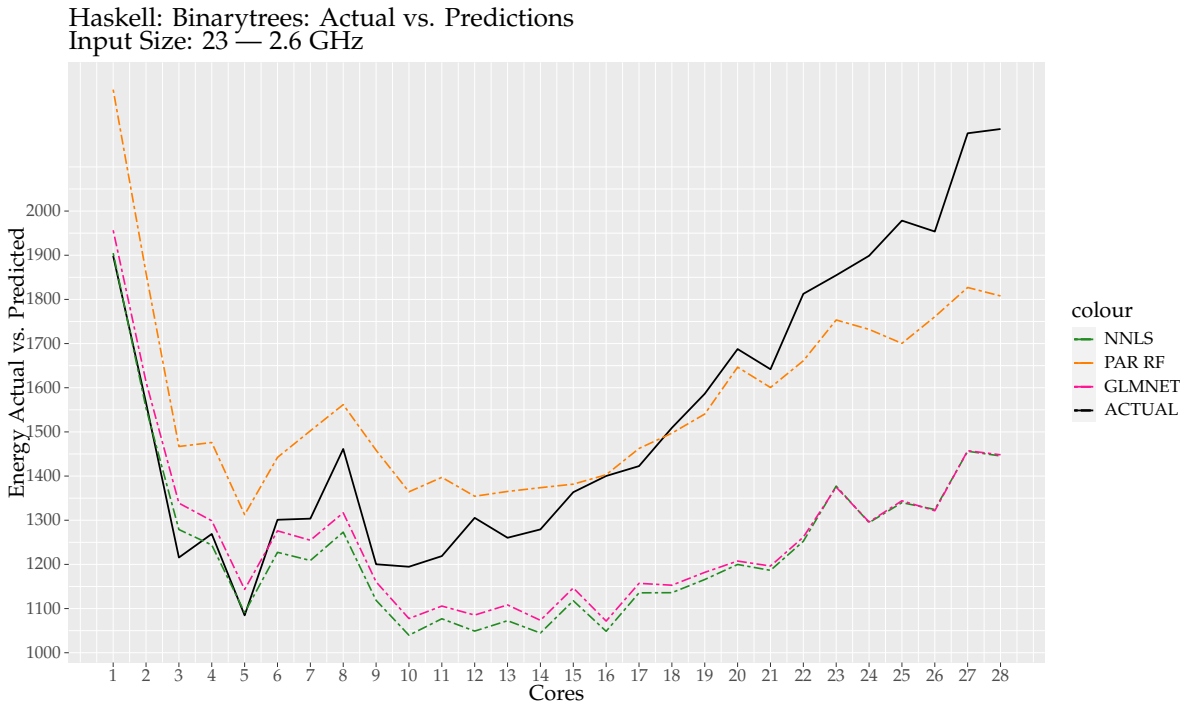


Figure 3.19: Binarytrees – *Energy consumption* for input 23 for models *NNLS*, *parRF* and *GLMNET*

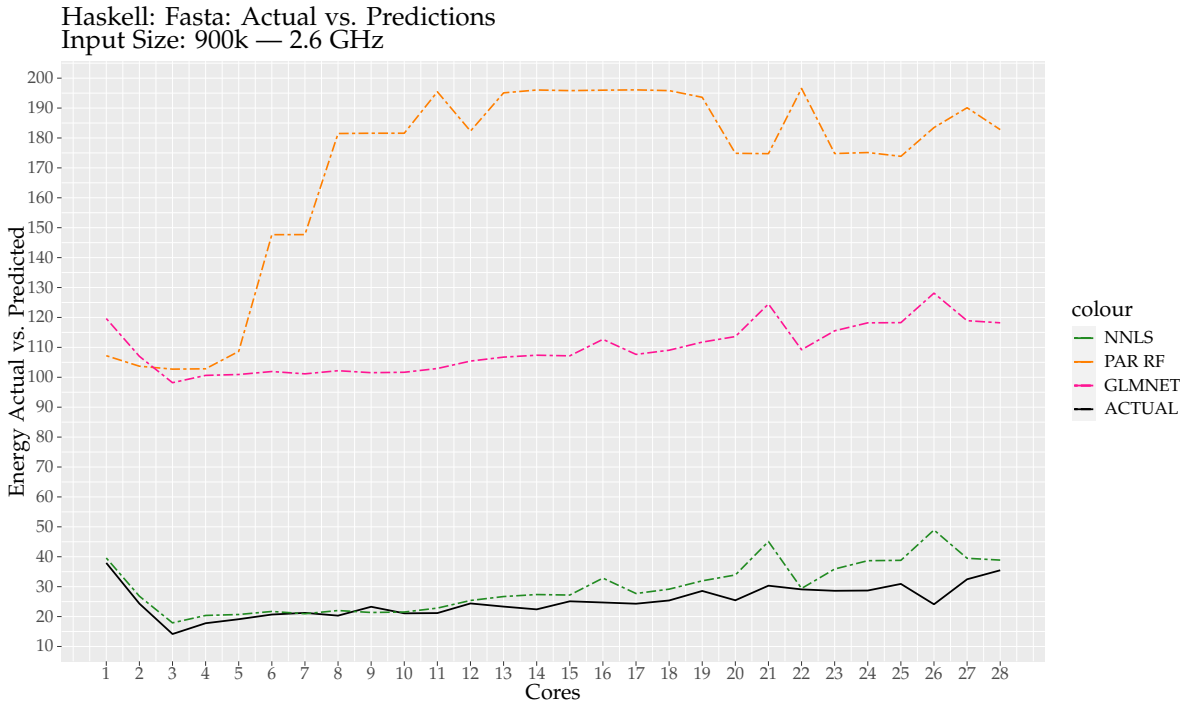


Figure 3.20: Fasta – *Energy consumption* for input 900K for models *NNLS*, *parRF* and *GLMNET*

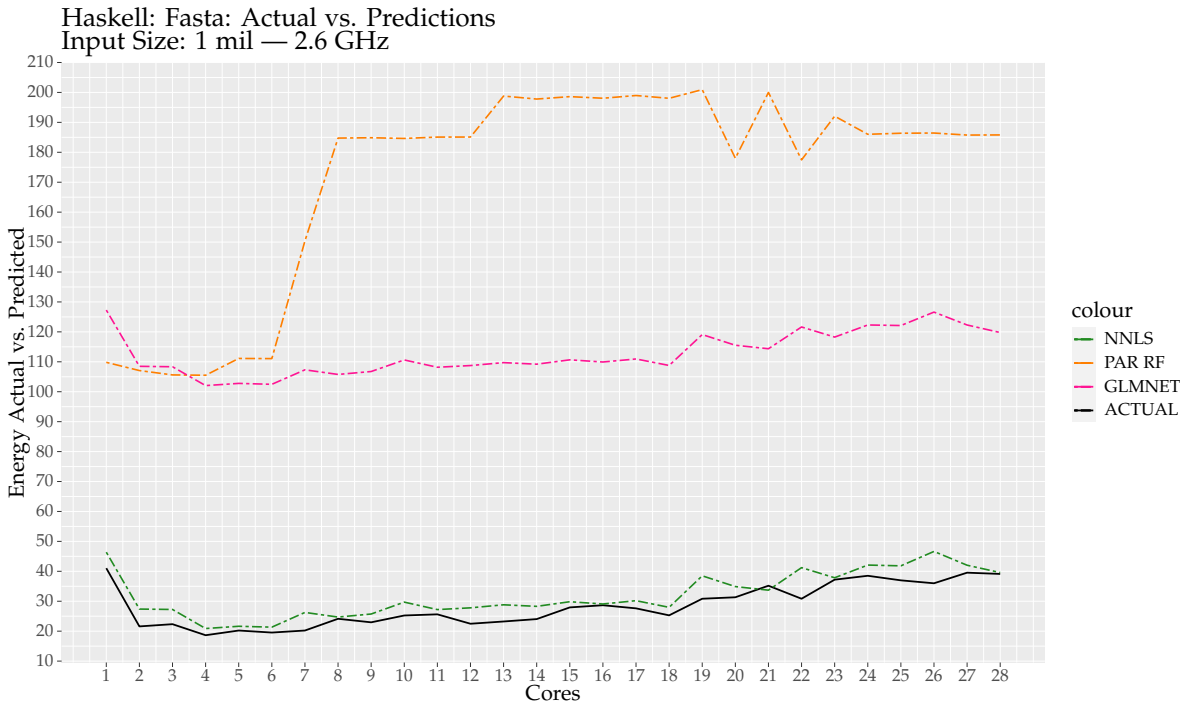


Figure 3.21: Fasta – *Energy consumption* for input 1M for models *NNLS*, *parRF* and *GLMNET*

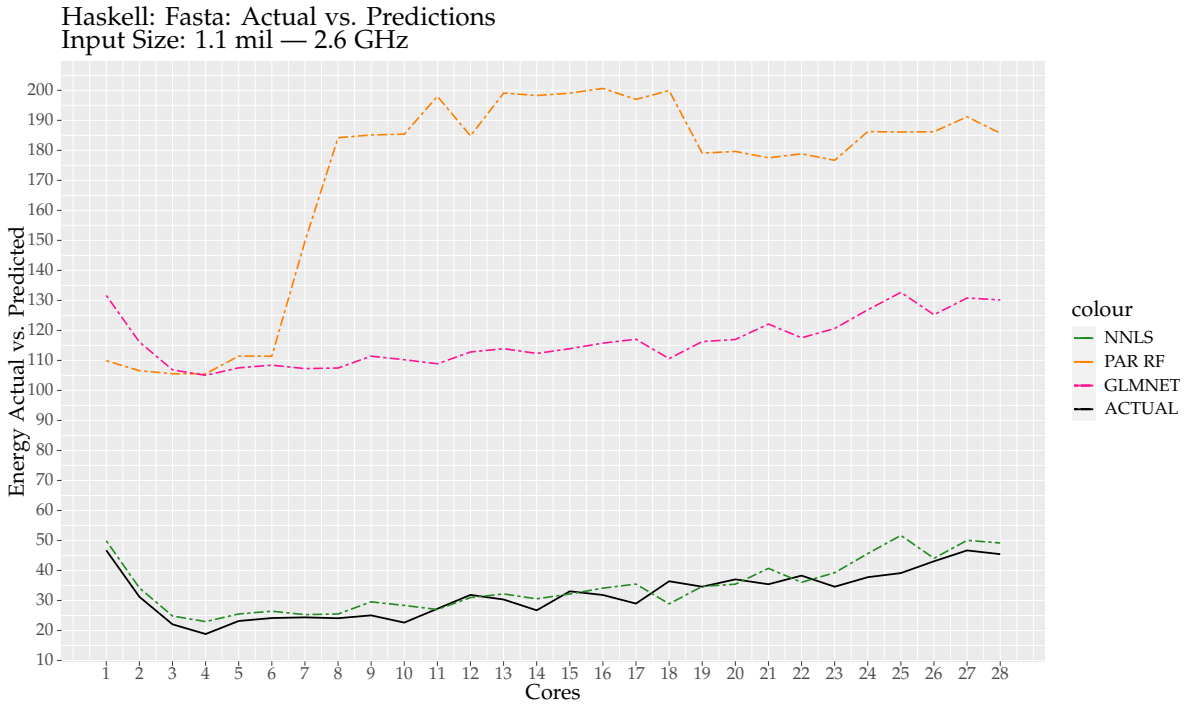


Figure 3.22: Fasta – *Energy consumption* for input 1.1M for models *NNLS*, *parRF* and *GLMNET*

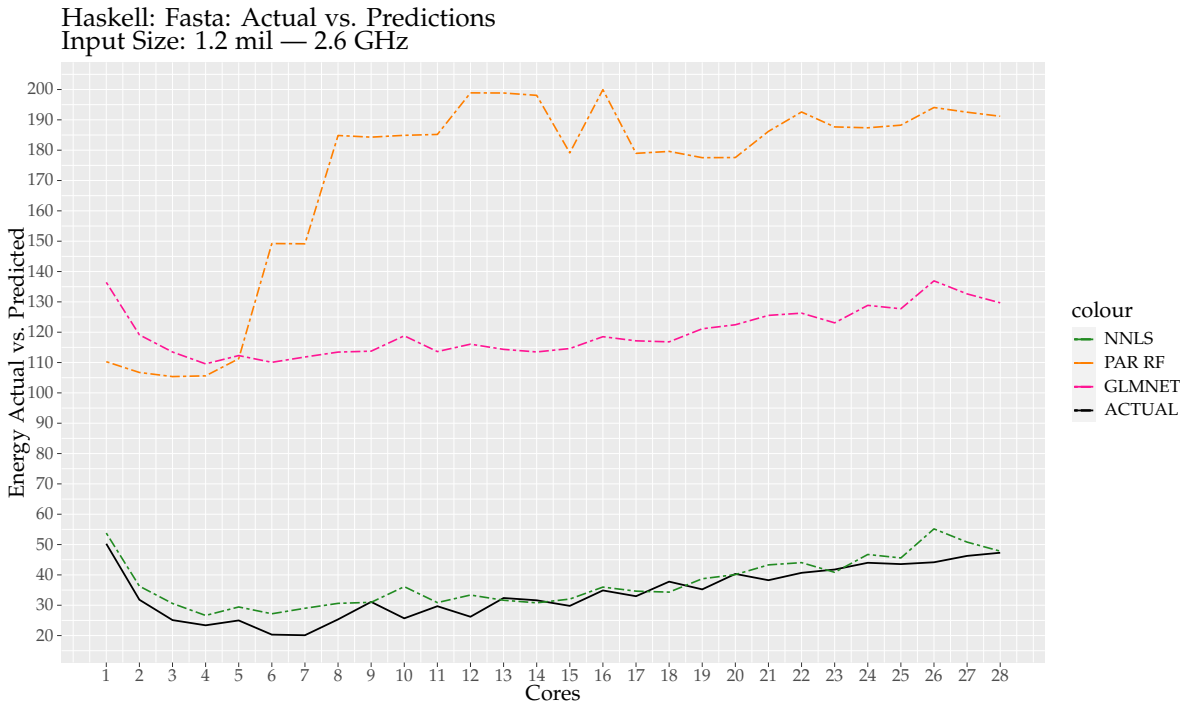


Figure 3.23: Fasta – *Energy consumption* for input 1.2M for models *NNLS*, *parRF* and *GLMNET*

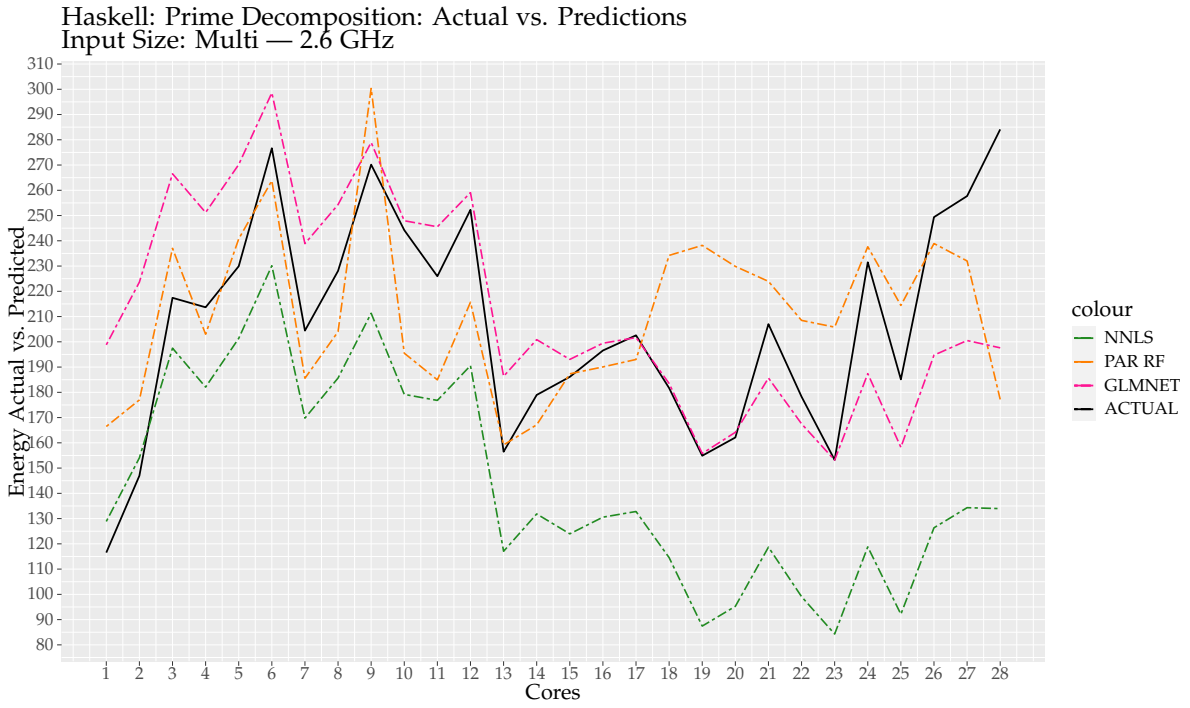


Figure 3.24: Prime Decomposition – *Energy consumption* for multiple inputs for models *NNLS*, *parRF* and *GLMNET*

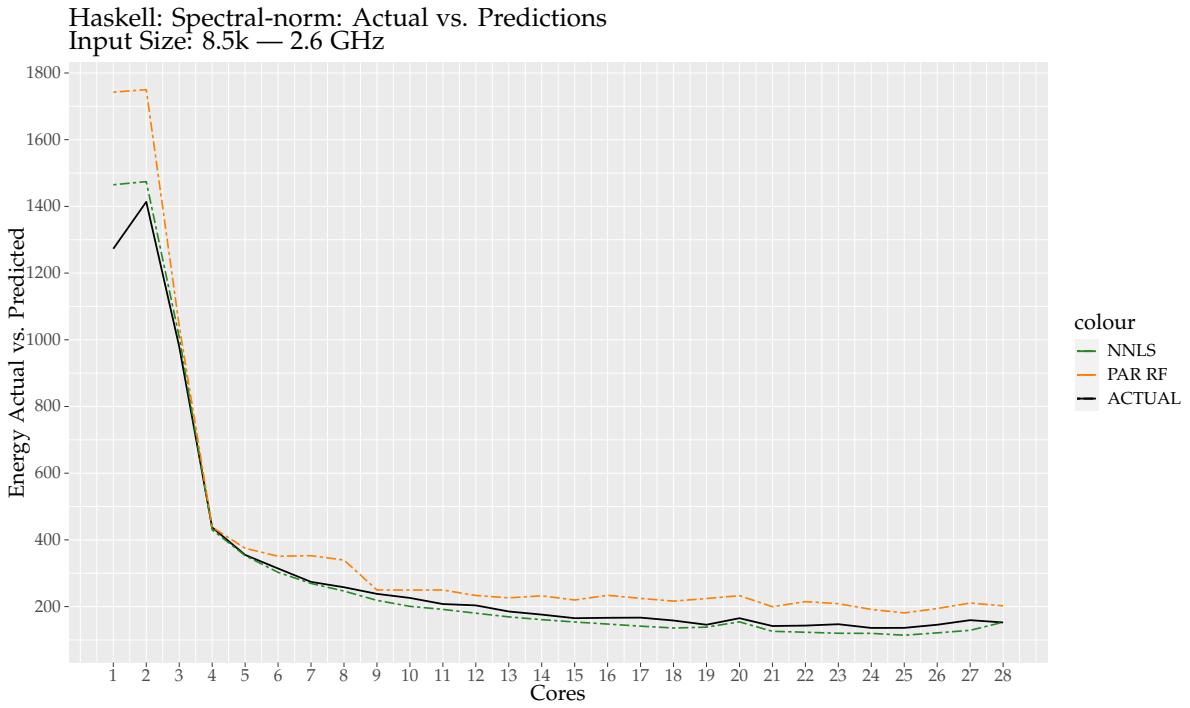


Figure 3.25: Spectral-norm – *Energy consumption* for input 8.5K for models *NNLS* and *parRF*

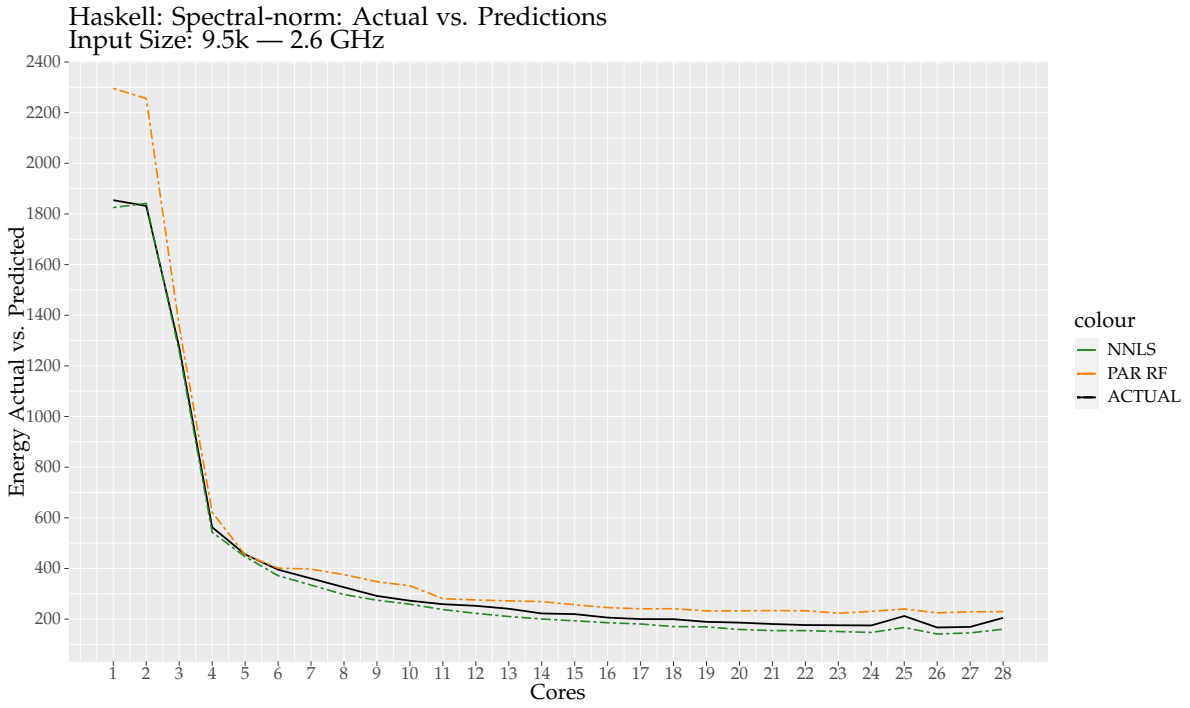


Figure 3.26: Spectral-norm – *Energy consumption* for input 9.5K for models *NNLS* and *parRF*

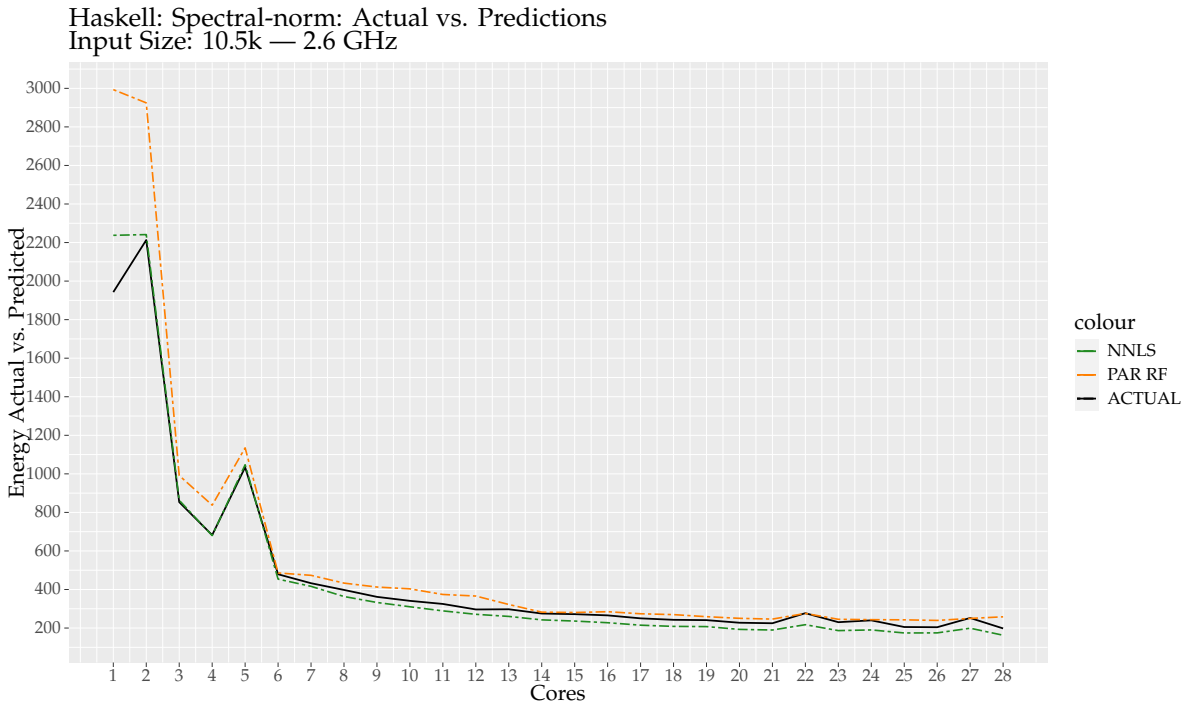


Figure 3.27: Spectral-norm – *Energy consumption* for input 10.5K for models *NNLS* and *parRF*

NNLS		
Benchmark	Input Size	MAPE
Binarytrees	20	11.52%
	21	17.93%
	22	19.27%
	23	18.22%
Fasta	900k	18.66%
	1M	13.83%
	1.1M	11.07%
	1.2M	12.73%
Prime Decomposition	Multi	30.52%
Spectral-norm	8.5k	9.20%
	9.5k	10.18%
	10.5k	11.71%
	11.5k	10.59%

GLMNET		
Benchmark	Input Size	MAPE
Binarytrees	20	40.25%
	21	16.42%
	22	16.38%
	23	17.46%
Fasta	900k	349.93%
	1M	314.36%
	1.1M	277.92%
	1.2M	269.81%
Prime Decomposition	Multi	14.18%
Spectral-norm	8.5k	2347.55%
	9.5k	2350.41%
	10.5k	2315.16%
	11.5k	2309.59%

Random Forest		
Benchmark	Input Size	MAPE
Binarytrees	20	29.46%
	21	13.79%
	22	15.80%
	23	10.44%
Fasta	900k	590.12%
	1M	522.37%
	1.1M	448.90%
	1.2M	424.20%
Prime Decomposition	Multi	15.76%
Spectral-norm	8.5k	28.35%
	9.5k	18.72%
	10.5k	13.20%
	11.5k	7.65%

Table 3.5: MAPE percentages for NNLS, GLMNET and parRF for Haskell sample

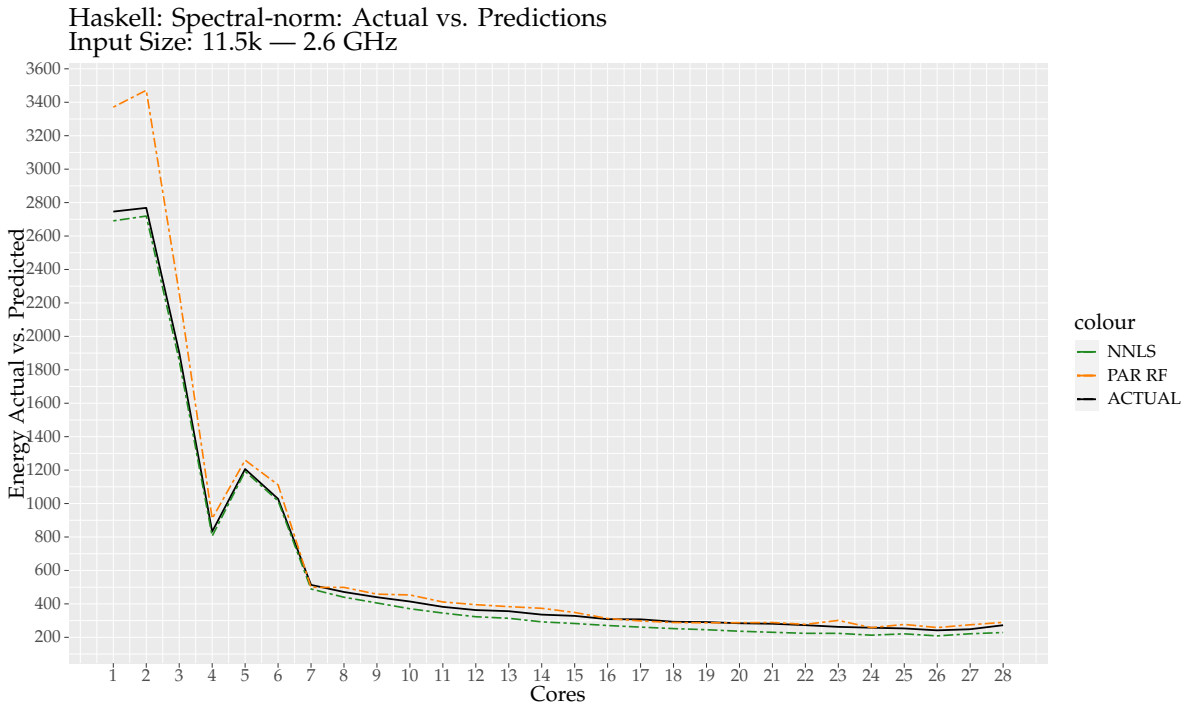


Figure 3.28: Spectral-norm – *Energy consumption* for input 11.5K for models *NNLS* and *parRF*

exceptionally accurate levels of NNLS exemplify the model's ability to accurately predict total energy consumed over a sequence of cores. However, this was an OpenMP-specific case, as both TBB and Pthreads had error margins larger than OpenMP. NNLS also showed close energy predictions in small energy consumption cases, e.g. OpenMP's version of Fasta, where the total energy consumed was around 1.2–3.9 J. Although random forest predicted the energy curve to some extent, the error margins were vast, and the energy consumption difference was up to 1400 J.

Apart from Binarytrees, the TBB versions of the benchmarks had small energy footprints, making energy predictions less accurate. In Binarytrees, we observed that most models underpredicted energy consumption, suggesting that the trained models may benefit from more extensive training that would include missing features, such as those used by TBB. Pthreads also showed similar cases to TBB, regarding low energy consumptions in Fasta, Prime Decomposition and Spectral norm. For Binarytrees, we observed that all inputs fluctuated in terms of energy consumption, which differs from other benchmarks, where energy consumption was reduced with an increasing number of cores. The remaining predictions and actual comparison is available in Appendix E.

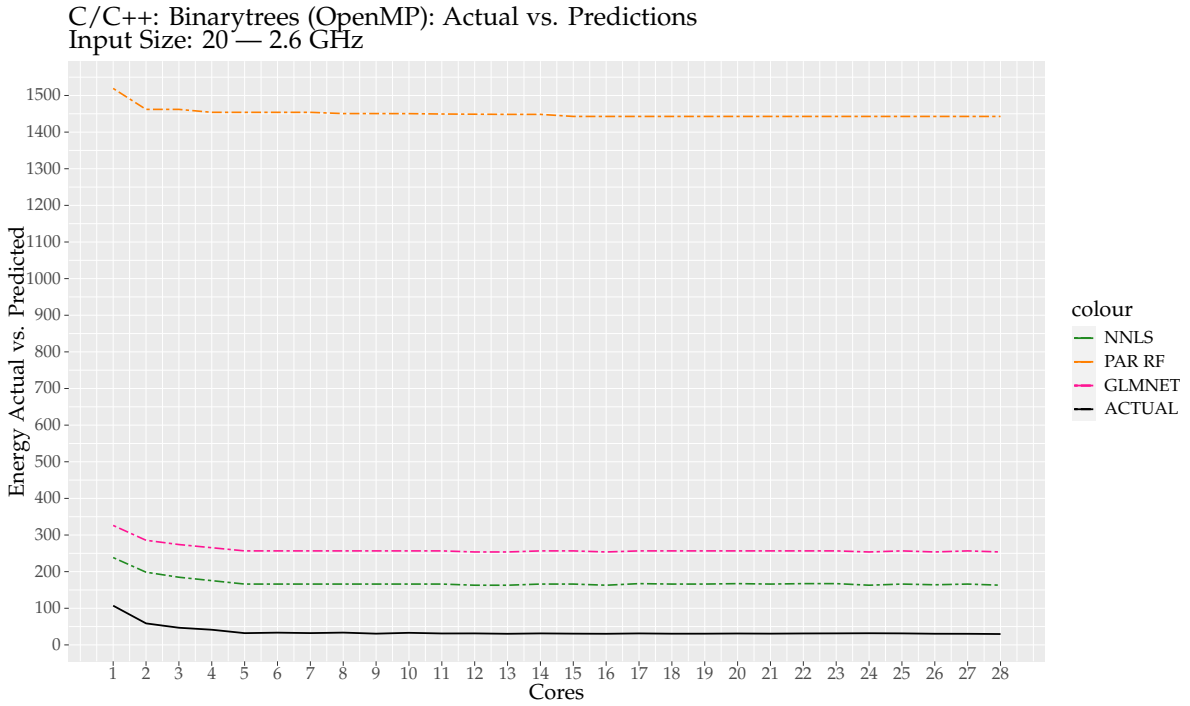


Figure 3.29: Binarytrees – *Energy consumption* for input 20 for models *NNLS*, *parRF* and *GLMNET*

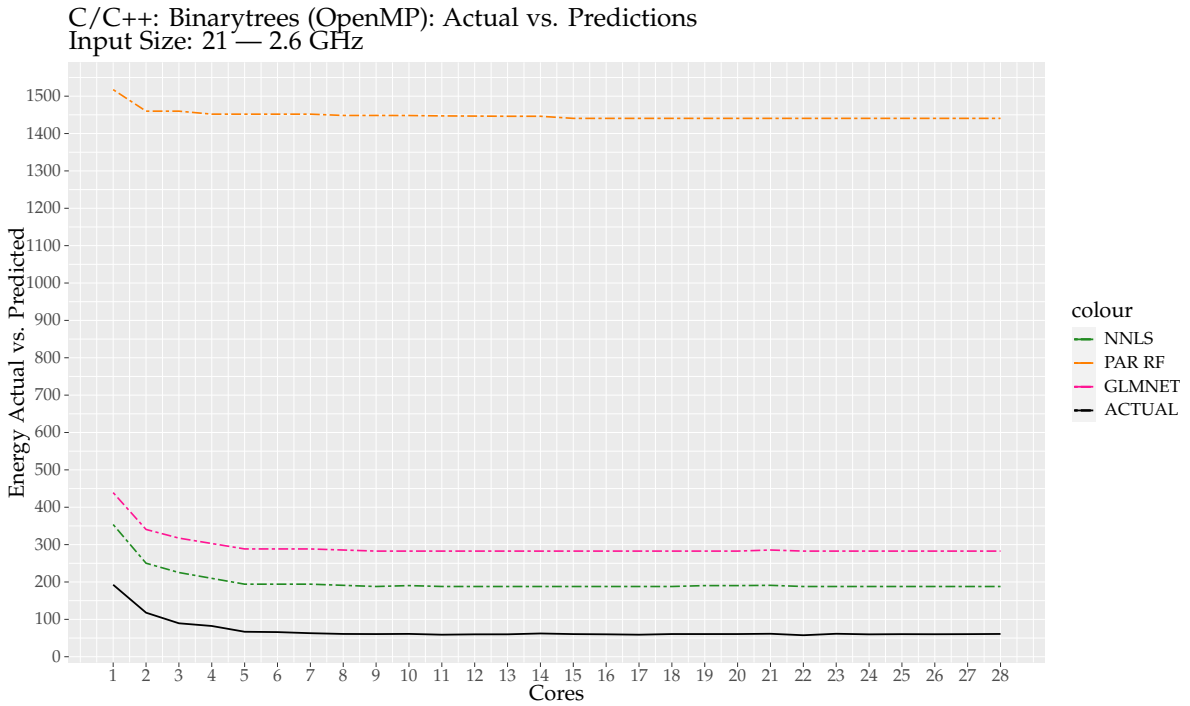


Figure 3.30: Binarytrees – *Energy consumption* for input 21 for models *NNLS*, *parRF* and *GLMNET*

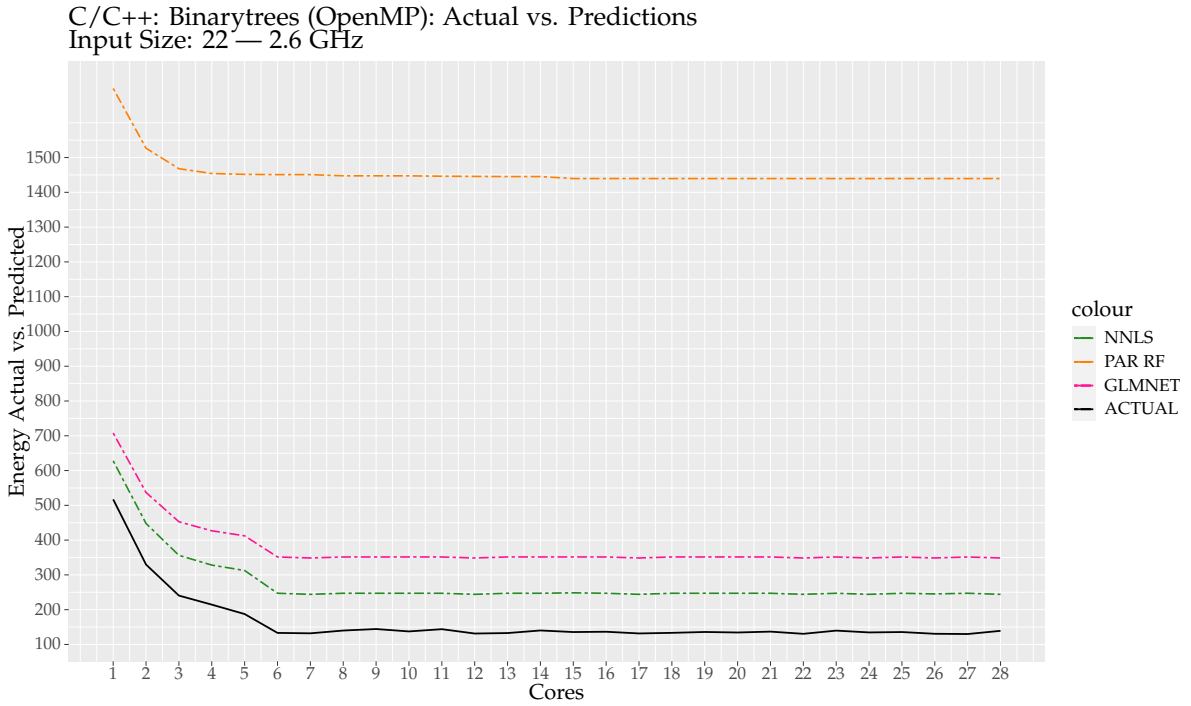


Figure 3.31: Binarytrees – *Energy consumption* for input 22 for models *NNLS*, *parRF* and *GLMNET*

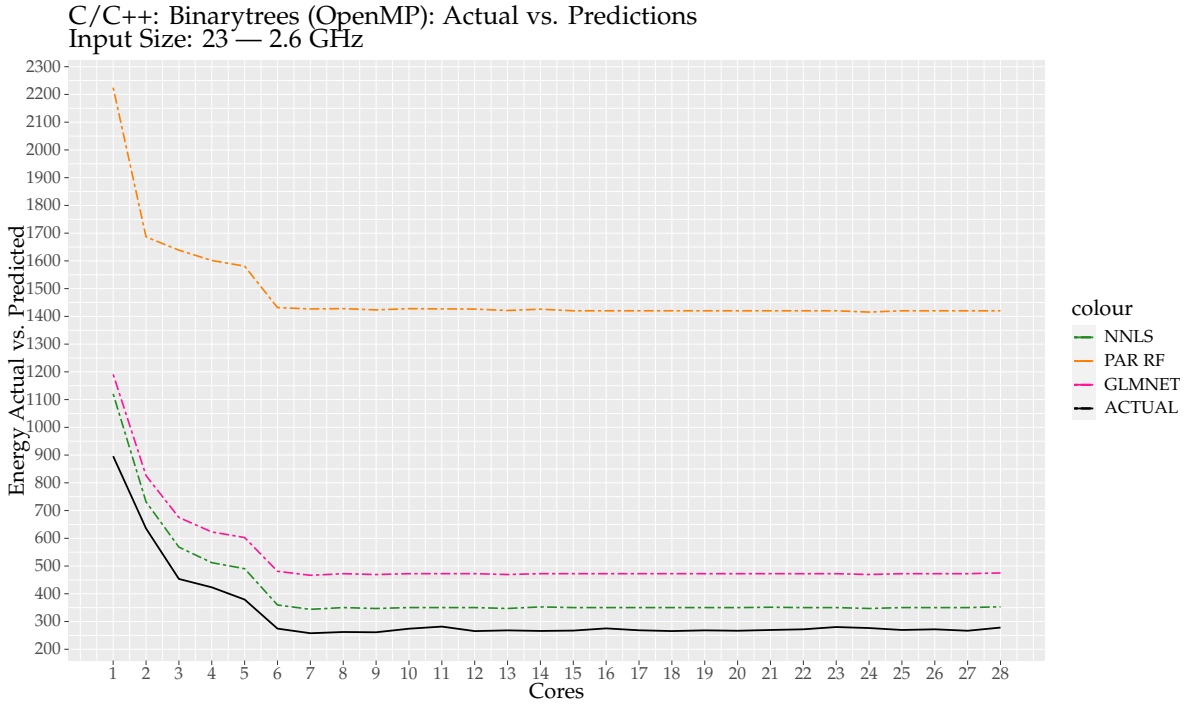


Figure 3.32: Binarytrees – *Energy consumption* for input 23 for models *NNLS*, *parRF* and *GLMNET*

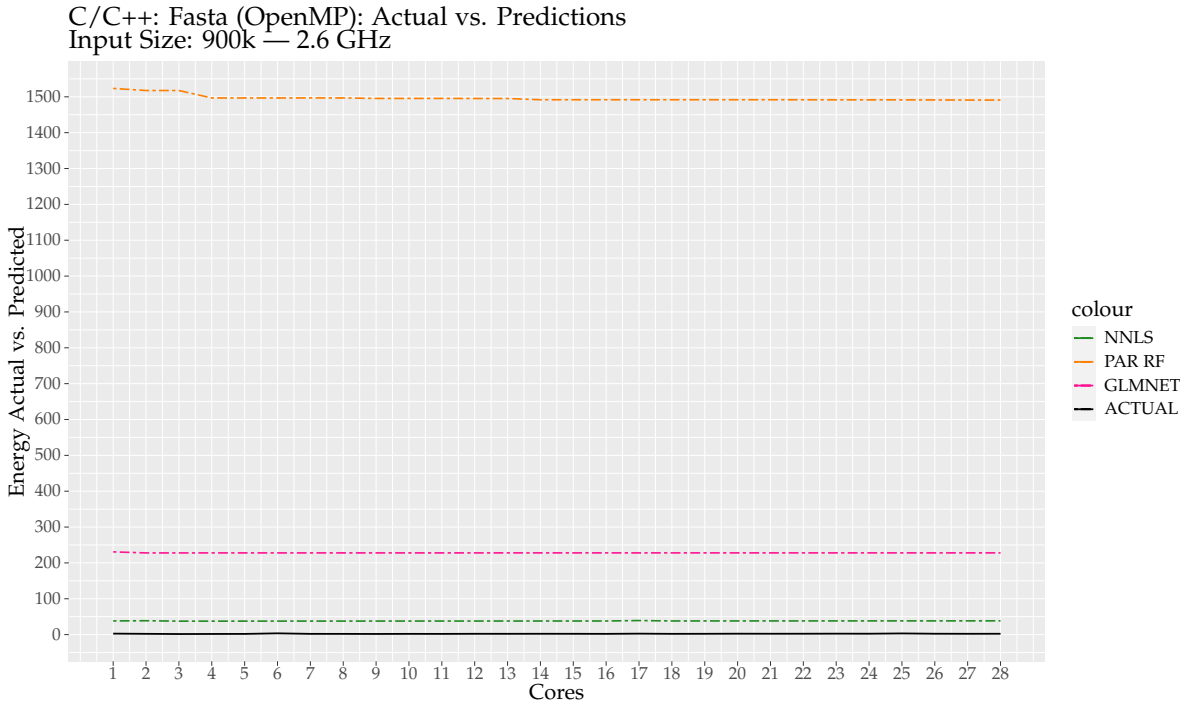


Figure 3.33: Fasta – *Energy consumption* for input 900K for models *NNLS*, *parRF* and *GLMNET*

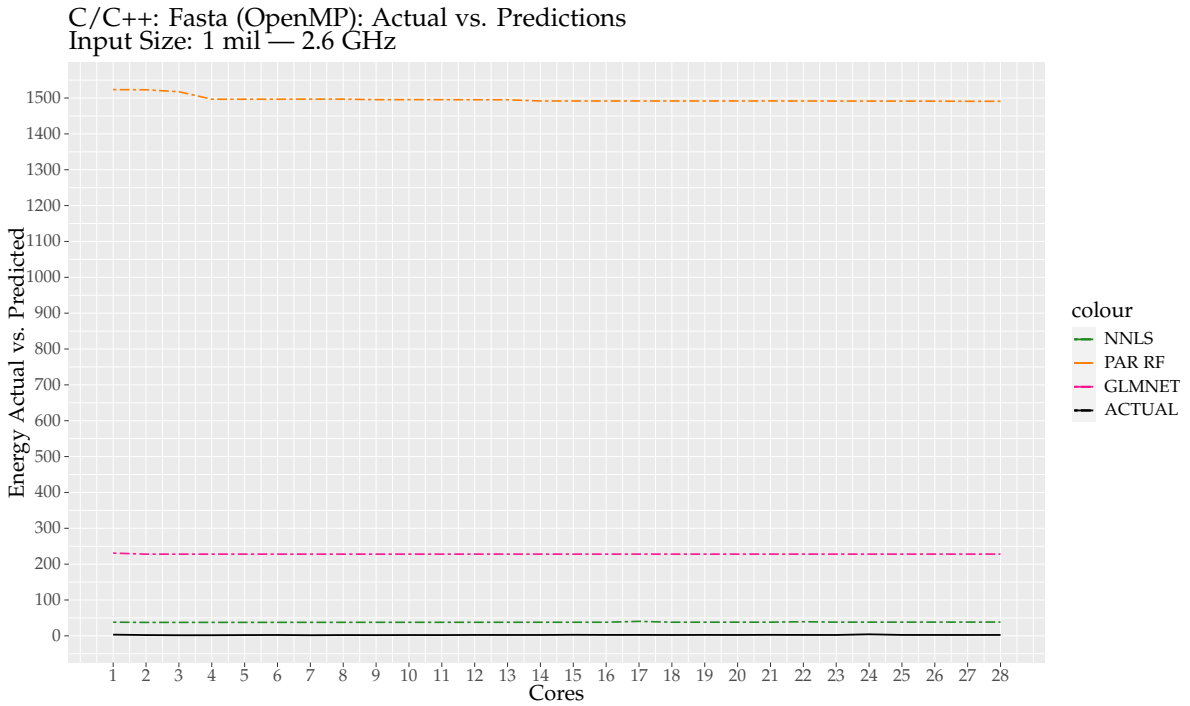


Figure 3.34: Fasta – *Energy consumption* for input 1M for models *NNLS*, *parRF* and *GLMNET*

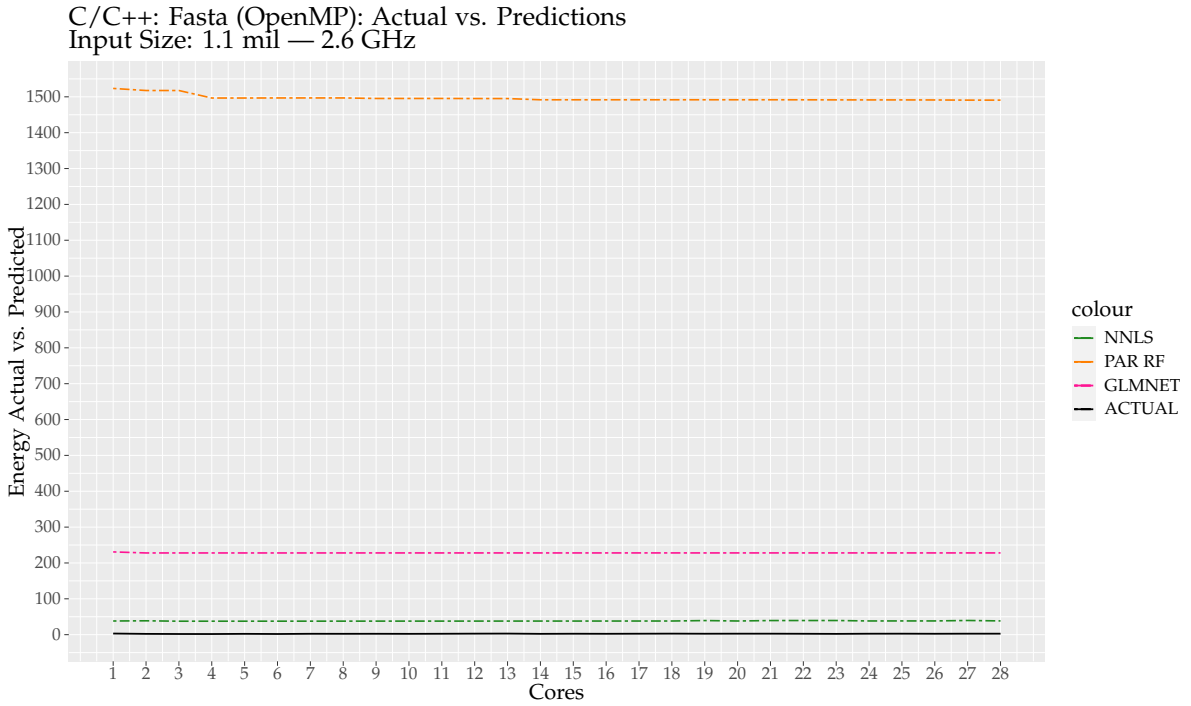


Figure 3.35: Fasta – *Energy consumption* for input 1.1M for models *NNLS*, *parRF* and *GLMNET*

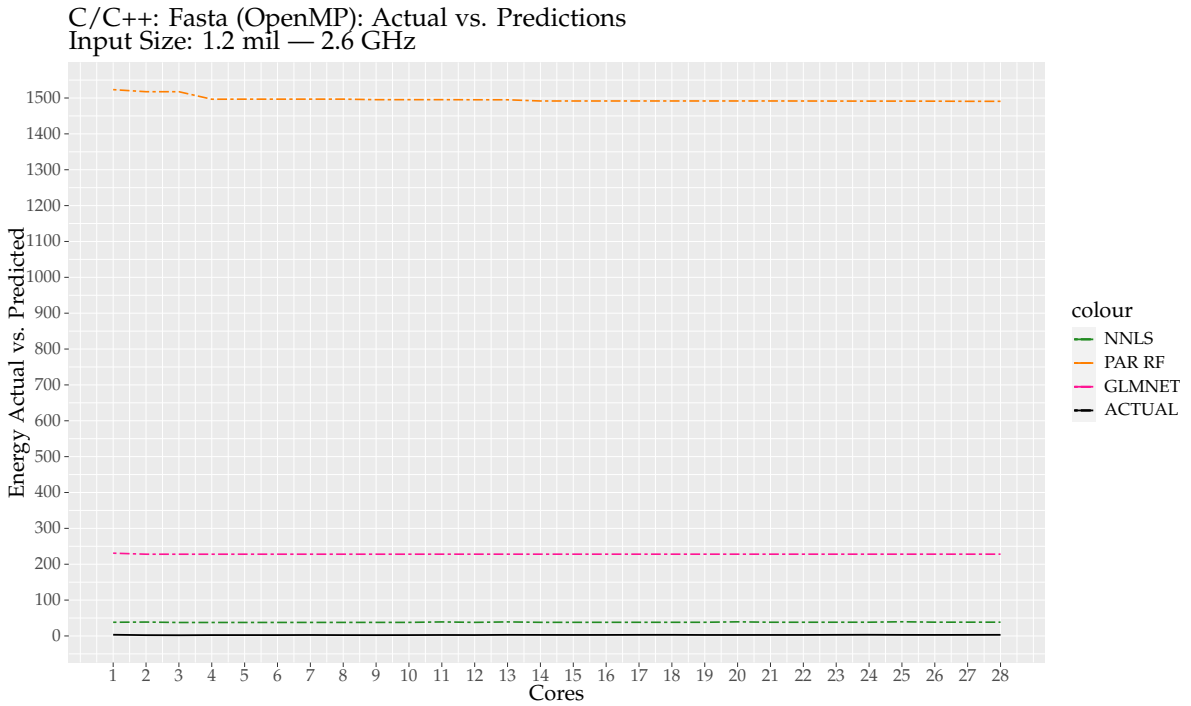


Figure 3.36: Fasta – *Energy consumption* for input 1.2M for models *NNLS*, *parRF* and *GLMNET*

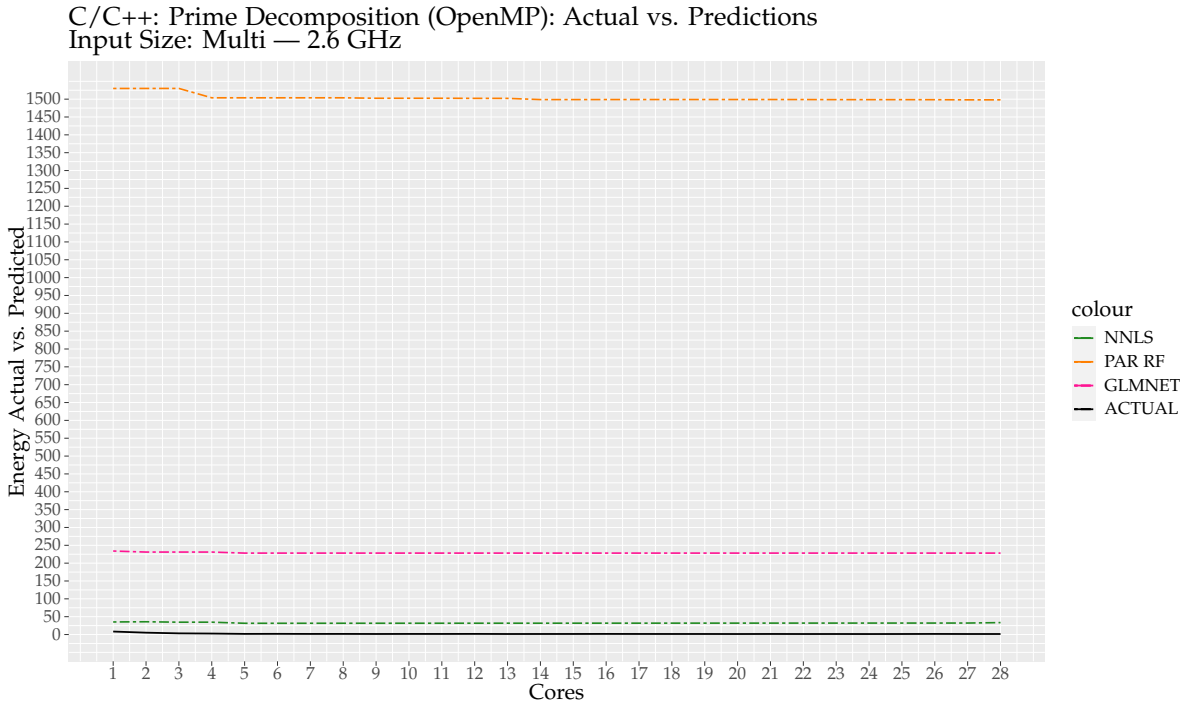


Figure 3.37: Prime Decomposition – *Energy consumption* for multiple inputs for models *NNLS*, *parRF* and *GLMNET*

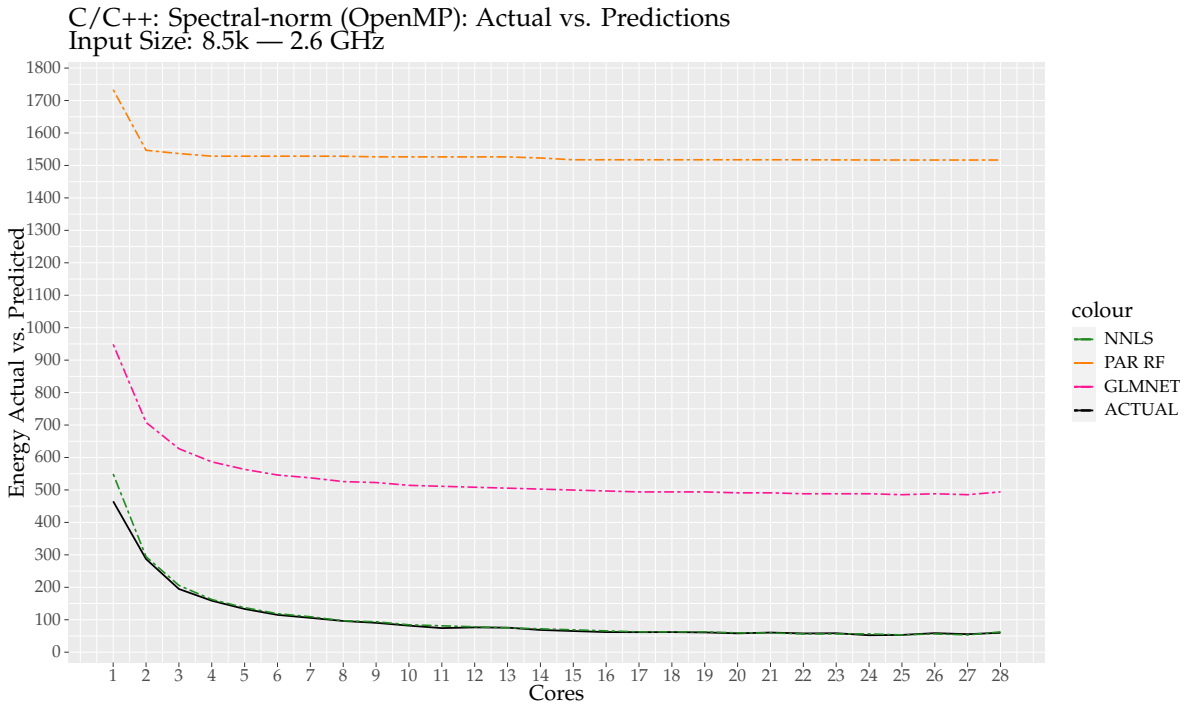


Figure 3.38: Spectral-norm – *Energy consumption* for input 8.5K for models *NNLS* and *parRF*

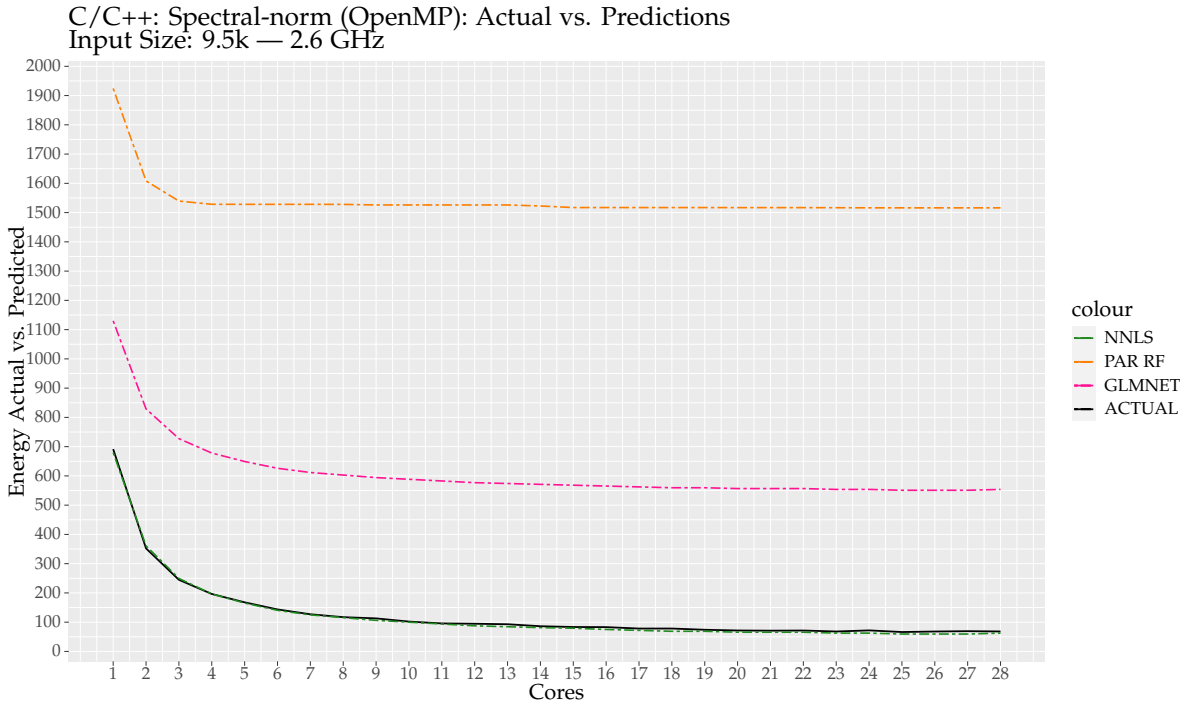


Figure 3.39: Spectral-norm – *Energy consumption* for input 9.5K for models *NNLS* and *parRF*

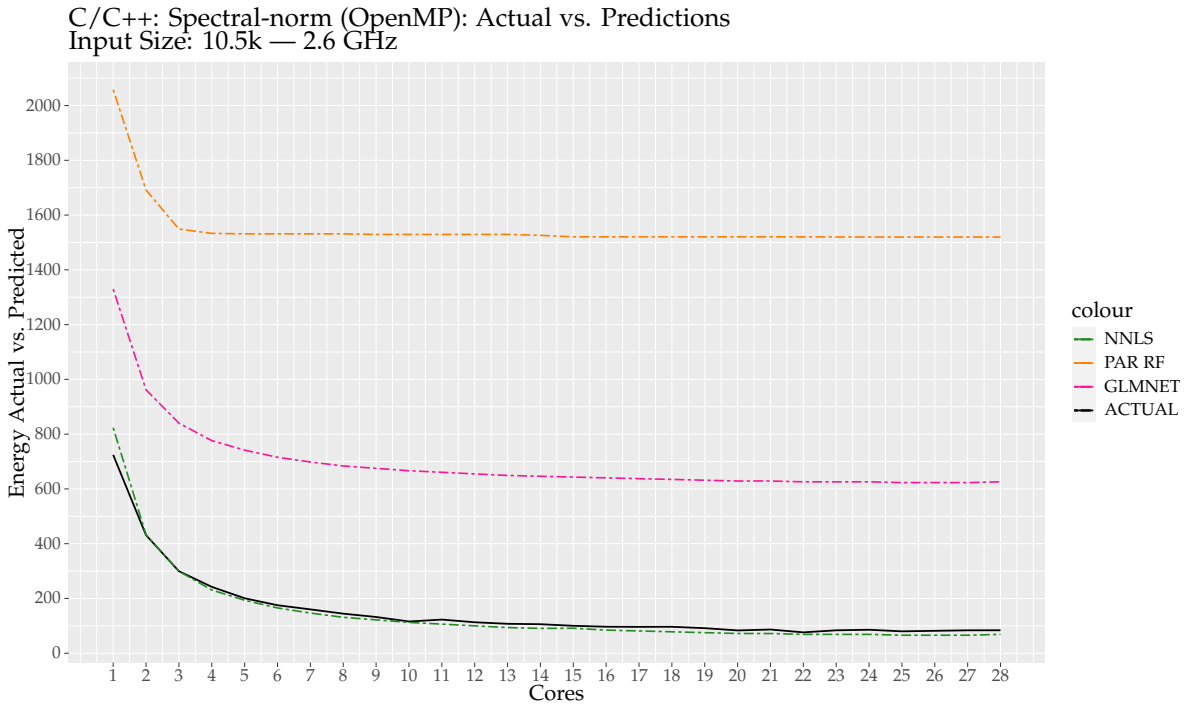


Figure 3.40: Spectral-norm – *Energy consumption* for input 10.5K for models *NNLS* and *parRF*

NNLS		
Benchmark	Input Size	MAPE
Binarytrees	20	402.90%
	21	198.53%
	22	75.20%
	23	28.65%
Fasta	900k	1589.53%
	1M	1483.55%
	1.1M	1443.98%
	1.2M	1315.13%
Prime Decomposition	Multi	1743.52%
Spectral-norm	8.5k	3.94%
	9.5k	6.30%
	10.5k	12.03%
	11.5k	17.59%

GLMNET		
Benchmark	Input Size	MAPE
Binarytrees	20	673.37%
	21	343.34%
	22	144.87%
	23	69.45%
Fasta	900k	10034.59%
	1M	9389.18%
	1.1M	9115.23%
	1.2M	8352.95%
Prime Decomposition	Multi	13008.28%
Spectral-norm	8.5k	569.39%
	9.5k	513.63%
	10.5k	480.09%
	11.5k	443.75%

Random Forest		
Benchmark	Input Size	MAPE
Binarytrees	20	4239.52%
	21	2125.67%
	22	876.16%
	23	392.83%
Fasta	900k	66416.37%
	1M	62184.39%
	1.1M	60369.89%
	1.2M	55366.48%
Prime Decomposition	Multi	86089.05%
Spectral-norm	8.5k	1904.51%
	9.5k	1517.65%
	10.5k	1252.97%
	11.5k	1023.13%

Table 3.6: MAPE percentages for NNLS, GLMNET and parRF for OpenMP sample

NNLS		
Benchmark	Input Size	MAPE
Binarytrees	20	20.18%
	21	28.01%
	22	30.09%
	23	29.02%
Fasta	900k	572.26%
	1M	505.67%
	1.1M	408.19%
	1.2M	366.74%
Prime Decomposition	Multi	4162.56%
Spectral-norm	8.5k	337.66%
	9.5k	335.87%
	10.5k	330.78%
	11.5k	339.06%

GLMNET		
Benchmark	Input Size	MAPE
Binarytrees	20	10.06%
	21	17.79%
	22	18.83%
	23	17.00%
Fasta	900k	771.73%
	1M	697.06%
	1.1M	590.09%
	1.2M	537.01%
Prime Decomposition	Multi	6674.32%
Spectral-norm	8.5k	363.68%
	9.5k	361.69%
	10.5k	357.66%
	11.5k	363.82%

Random Forest		
Benchmark	Input Size	MAPE
Binarytrees	20	128.13%
	21	36.57%
	22	12.51%
	23	13.61%
Fasta	900k	5268.52%
	1M	4780.16%
	1.1M	4094.90%
	1.2M	3747.51%
Prime Decomposition	Multi	44370.97%
Spectral-norm	8.5k	2491.51%
	9.5k	2480.45%
	10.5k	2455.45%
	11.5k	2492.14%

Table 3.7: MAPE percentages for NNLS, GLMNET and parRF for TBB sample

NNLS		
Benchmark	Input Size	MAPE
Binarytrees	20	21.49%
	21	24.63%
	22	28.86%
	23	29.11%
Fasta	900k	1431.25%
	1M	1368.91%
	1.1M	1276.31%
	1.2M	1173.77%
Prime Decomposition	Multi	1672.44%
Spectral-norm	8.5k	2164.73%
	9.5k	2176.45%
	10.5k	2158.48%
	11.5k	2125.39%

GLMNET		
Benchmark	Input Size	MAPE
Binarytrees	20	24.55%
	21	1.50%
	22	11.11%
	23	15.03%
Fasta	900k	10717.09%
	1M	10269.88%
	1.1M	9619.40%
	1.2M	8881.68%
Prime Decomposition	Multi	4325.13%
Spectral-norm	8.5k	4106.57%
	9.5k	4131.76%
	10.5k	4093.77%
	11.5k	4032.63%

Random Forest		
Benchmark	Input Size	MAPE
Binarytrees	20	150.84%
	21	166.56%
	22	76.93%
	23	5.97%
Fasta	900k	33975.54%
	1M	32604.81%
	1.1M	30425.34%
	1.2M	28106.69%
Prime Decomposition	Multi	28945.50%
Spectral-norm	8.5k	27405.06%
	9.5k	27579.39%
	10.5k	27330.07%
	11.5k	26920.53%

Table 3.8: MAPE percentages for NNLS, GLMNET and parRF for Pthreads sample

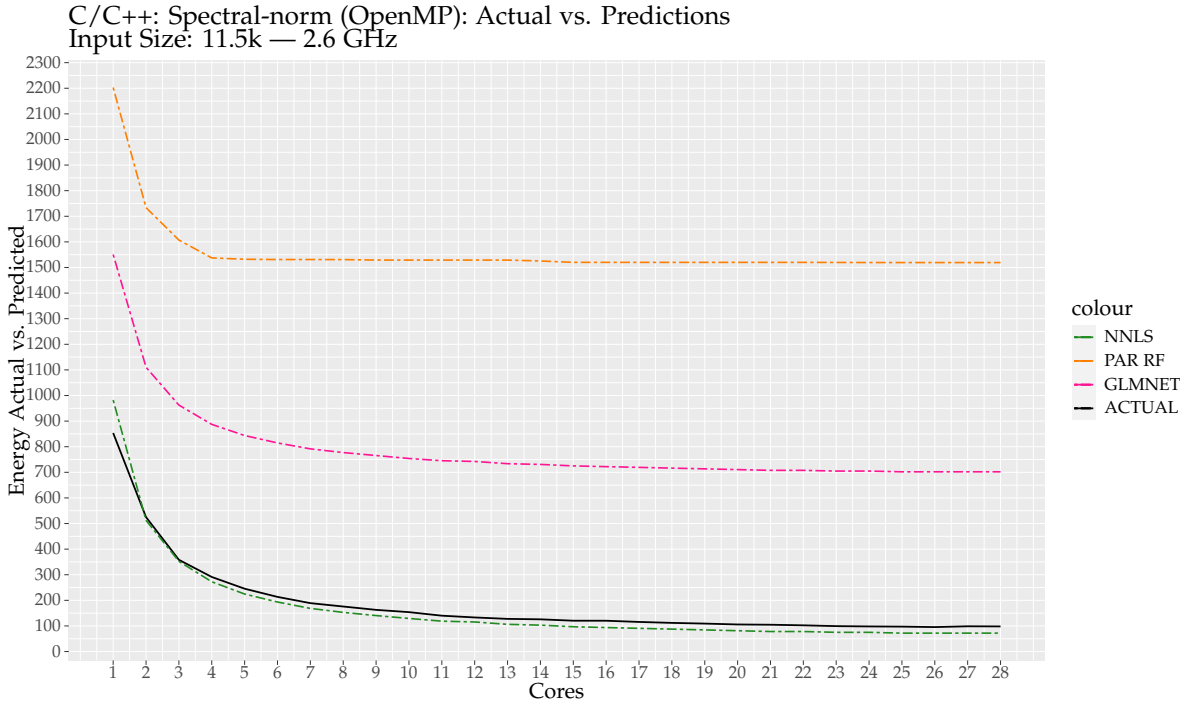


Figure 3.41: Spectral-norm – *Energy consumption* for input 11.5K for models *NNLS* and *parRF*

3.5 Effect of Reducing Clock Frequency on Energy Consumption

In the experiments performed thus far, we used the maximum clock frequency available to our system, a stable *2.6 GHz*, for Haskell and PARSEC modelling benchmarks. However, the power drawn by a CPU is proportional to the cube of its clock frequency; thus, we anticipate that we can further reduce energy usage for each benchmark by simply reducing the clock speed. Rather than purchasing additional testbed machines, we used the Linux *governor* to set the clock frequency for all cores to one of several fixed values: 1.2 (the minimum), 1.4, 1.8 and 2.1 GHz. The results obtained for the benchmarks from 1.8–2.6 GHz are provided in Appendix B. Notably, they provide a consistent picture. Each graph of lower clock frequency has a similar shape to that of the benchmarks at 2.6 GHz, showing remarkable consistency. However, we observed that the *energy usage increased when we used some lower clock frequencies*. For example, for **SumEuler**, the maximum energy usage was **18708** *J* at 2.6 GHz, but it was **22457** *J* at 1.2 GHz. The total energy usage was only lower at 2.1 GHz. A similar story holds for the other benchmarks. Although in some cases, e.g. **MatMult** and **NQueens**, we saw that the least energy was used at 2.6 GHz. As with our previous scalability experiments, we concluded that the effect of increasing the execution time outweighed the benefits of reducing clock frequency. For the architecture that is in focus, of the frequencies considered, 2.1 GHz is the best choice for most benchmarks, given its least total energy usage. We presume that the processor has been optimised to run at this or a similar frequency; higher clock frequencies may, for example, generate additional heat losses. We conclude that the clock frequency is a significant factor in determining energy usage, though not precisely in the way that we anticipated. Significant further experimentation is needed to determine the optimal clock frequency and to provide measurements that include it as part of our overall energy model. Nevertheless, the consistency of each graph suggests that the clock frequency is an essential scaling factor that does not affect model construction fundamentally. When adjusting the frequency of 2×NUMA nodes, a common energy consumption pattern appears in our test environment. Presumably, the lower the frequency, the more energy consumed by the application, as it executes for longer time. However, in one case, this does not hold. We still found the same pattern of higher energy consumption, beginning from the sequential execution onwards, except when running 56 cores at 2.1 GHz, where the general theme appeared to be

lower energy consumption by several thousand *joules*. Additionally, the fact that we saw scalability patterns maintained in all frequencies supports that performance improvements can enhance energy consumption, regardless of CPU frequency.

A similar case to Haskell's frequency effect was observed in C/C++ benchmarks. The multiple frequency values impacted the overall energy consumption with lower frequency values increasing the total time, thus increasing the energy consumed. The tradeoff effect between energy consumption and time was not always consistent, as we observed specific benchmarks providing larger energy reductions when the execution time was only slightly increased.

For example, Blackscholes in all implementations showed that parallel speedups were maintained between 5× and 6× when comparing 2.1 and 2.6 GHz. The energy and execution time effect was different, however. In Blackscholes, we observed a consistent 20 to 25% increase in time where all implementations showed execution times increasing by 24 to 30 s. When considering the energy differences of Blackscholes, the decreases were 21% at 2.1 GHz, where the energy values were reduced from an average of 983 to 771 J across all implementations. Another example is Bodytrack, with which the average execution time increased from 24 to 38%, and the execution time varied between 66 to 81 s for 2.6 GHz and 83 to 112 s for 2.1 GHz. The Bodytrack energy reductions ranged from 19 to 21%, where total energy consumed varied between 5361 and 5711 J for 2.6 GHz and 4268 to 4503 J. The frequency effect on both Ferret and Swaptions had a poor ratio of energy reductions to execution time increase, as both benchmarks had short execution times.

In some instances, the increase in execution time can be harmful when running code. However, in a scenario where energy control supersedes execution time, it may be essential to find minimal energy consumption at the cost of other factors, such as a minor increase in execution time.

3.6 Summary

This chapter investigated the energy usage for parallel Haskell and C/C++ benchmarks. We used three sets of benchmarks, with one having equivalent implementation in both Haskell and C/C++.

We found a strong correlation between energy usage and execution time for the benchmarks studied. We also found that using more processor cores can improve energy usage. In the Haskell dataset, power and CPU utilisation (GHC productivity) can significantly impact energy usage. Benchmarks with insignificant average power and productivity levels can have poor energy usage. Thus, increased performance may be beneficial for energy usage and execution time.

Based on our experimental results, we used NNLS, GLMNET and random forest to construct language-specific and generic energy models that can predict an application's energy usage on our experimental system for varying input sizes. Although we obtained good results with NNLS, some benchmarks challenged GLMNET and random forest models. Finally, we investigated the effects of core affinity on the Haskell dataset by pinning threads to specific cores and varying overall clock frequencies on energy usage. We determined that core affinity slightly affected our result consistency and have therefore avoided using it. Our investigation of clock frequencies showed that lowering the clock frequency for each core can increase or decrease total energy usage while also impacting execution times at varying levels. Moreover, improved time performance implies improved energy usage because there is a significant baseline energy cost incurred simply from using each processor package. However, it was determined that the maximum clock frequency did not yield the best overall energy usage, specifically when considering the context in which a small tradeoff in execution time can increase and energy consumption can be lowered, which may be more substantial than reducing execution time by a small percentage.

Chapter 4

Meta-Heuristic Analysis

In this chapter, we discuss the second approach to making energy consumption estimates. Section 4.1 explains the problem to be solved and provides examples from the experiments described in earlier chapters. Section 4.2 explains the meaning of *search space* optimisation. Section 4.3 addresses the different meta-heuristic algorithms and their strengths and weaknesses. Section 4.4 defines the probabilistic algorithms used. Sections 4.5, 4.6 and 4.7 then formally define the meta-heuristic algorithms for determining the energy consumption of Hill Climbing, ACO and Genetic Algorithm computations, respectively.

4.1 Introduction

In Chapter 3, we reviewed an approach to predicting energy consumption for Haskell programs. It used statistical modelling in which a model was constructed based on data collected from program execution and hardware power draw. Practically, however, the method is suboptimal because when input sizes differ, workload and total energy consumption differs, resulting in inconsistent predictions.

For parallel multi-core applications, there are limited ways to predict energy consumption. For example, we could attempt to estimate all possible energy outcomes based on the available methods available for a program execution, e.g. every core, input and frequency combination. However, this would be completely unreasonable.

There are also statistical methods, such as the ones explored in Chapter 3, where we collected an extensive sample from a specific machine using particular builds, i.e. GHC version 7 or 8 with known compiled code versions. The process then involved building a regression set that we tried specialising and generalising; however, we ran into significant limitations resulting in inadequate predictions over multiple cases.

As implied by this chapter title, there are also meta-heuristic methods by which we can draw solutions from a sizeable search space designed around a problem model. Doing so would allow us to approach the problem as one of optimising the output of a function in a given problem space using an objective function. Then, we determine whether the given output is an optimal solution to the problem at hand.

Supposing we wish to design wheel rims for cars, as depicted in Figure 4.1 on page 93. Instead of creating an infinite number of moulds to accommodate an infinite number of designs, then testing each production rim for all possible car requirements, we create a function that would account for the effects of each design feature. For example, a hypothetical function like $f(x) = \frac{(0.6x)^2}{2} + 2x$, where f is the function that we wish to optimise, and x is the feature value that, when optimised, would lead to the desired result. Meta-heuristically, an algorithm would maximise or minimise the function's output.

Energy consumption can be irregular. That is, some programs may have improved energy consumption when using more cores, meaning that energy efficiency improves with increased parallelism. Other programs may waste more energy with more cores, etc. This irregularity can be exacerbated by different architectures, whose instruction sets handle data and operations differently. This effect impacts the number of Instructions Per Cycle (IPC)s as some processors perform better than others. Moreover, the physical environment can impact hardware performance, e.g. temperature controls. When considering the vast number of variables affecting energy consumption, meta-heuristics may offer the only viable solution to reducing the amount of time needed to assess the impact of many feature variables. Weise [108] surveyed and classified most meta-heuristic algorithms at the time.

Section 4.2 explains the problem to be optimised from generic viewpoint, and Section 4.3 examines general meta-heuristic examples used to solve the complex

problem of this thesis. The complete implementations of the algorithms in this chapter are available in the online appendix [1].

Figure 4.1: Example of different rim designs

4.2 Optimisation Problem

A *Search Space* requires a function, e.g. $f(x) = 2x^2$, that we aim to optimise by substituting x with feature values. We desire to reduce the number of trade-offs as much as possible so that our solution has a high accuracy. Regarding energy consumption, we consider a sequential program, which is compiled using a given ISA, and it is then executed on a computer architecture. To estimate a program's energy consumption, we consider the following steps:

- Create an extensive dataset of program code feature values without modifying the source, i.e. one optimisation per specific configuration.
- Derive a function of possible problem sizes and inputs for the program that will allow precise direct or extrapolated energy measurements within the narrowest upper and lower bounds. For example, knowing that a program executes for a certain amount of time for a given input, it is essential to understand how increasing the number of cores would affect the execution, resulting in a function of *time* vs. *core count*.
- Provide an environment supporting consistent hardware performance, to the extent that the smallest changes are controlled.
- Avoid altering the given clock frequency to eliminate unpredictable spikes in power draw. For example, with Intel's P-State driver, the chip can boost clock frequency for limited periods for various benefits.

[1] github https://github.com/ymg/thesis

- Sample the program in question using multiple iterations to avoid inconsistent measurements caused by outlying factors, e.g. hardware, OS and processes.

Although these steps will help narrow results in terms of predicted behaviours, excluding unintended ones, it does not yet account for the complexities of multi-core execution. For example, a single NUMA system with multiple packages would use schedulers and kernel processes to manage threads in parallel. Hence, substantial layers of complexity would complicate determining the best configuration for executing a given process. Additionally, if a given configuration can be reasoned about for future versions of the same code executed in the same environment. Moreover, changing clock frequencies add additional dimensions to the problem. All these factors add up.

To understand the complexity of the search space, consider Equation 4.1 on page 95. In our tested examples in Chapter 3, we used *five* clock frequencies (F) in addition to sampling each input (I) on each set of cores (C), which in the case of the testbed used, was *56*. If were to exhaust all configurations in the search space, this process would take (C = *56*) $\times$ (F = *5*), resulting in ***280*** different combinations to explore. Figure 4.2 on page 102 provides a heatmap representing the comparison of a set of inputs, I = *4*. The **y-axis** shows the number of cores used [2–28], and the **x-axis** shows the problem sizes when sampling total energy for a Ray Tracing execution. The bar at the right of the figure shows the scale of total energy consumed in every case, and the colours correspond to the cells in the figure. We notice a pattern of low energy consumption with cores of 10, 11 and 12 for all problem sizes sampled. As the number of cores increased, energy consumption increases until the maximum 28 cores, translating to worst performance. Recall that our objective is to minimise total energy consumed. Thus, to optimise the process and to ensure the lowest energy consumption while ensuring the best performance, the use of parallelism is out of the question (see previous chapters). Consider the *Ray Tracing* heatmap in Figure 4.2. Then, Figure 4.5 on page 105 shows that *Ray Tracing* performance worsens before it improves at around 11 cores, where we achieve lowest energy consumption and lowest execution time. Then, it returns to poor performance onward until the maximum number of cores. This behaviour is also seen in the *Minimax* in Figure 4.4 on page 104, where the large problem sizes (12 and 10) have peak performances for energy between problem sizes 14 and 15; however, we also notice that execution time plateaus around 16 cores with slight improvements in the case of problem size 12

and an increase in the case of problem size 10.

For example, if we wanted to determine the optimal input for *Ray Tracing* to save energy, we would first multiply $56 \times 5 \times 4 = 1120$; this number represents all possible search space combinations for *Ray Tracing*. Hence, to exhaust all possible combinations, it would require $1120 \times$ *unit of time* (milliseconds, seconds, minutes and so on). Thus, if n-body took 1.5 min to finish, 28 h would be needed to search all points in the space. If we were to use a more capable machine with multiple cores at multiple clock frequencies, we could search a bigger space. Considering that modern CPU architectures can have more than 100 cores on a single die (Ampere's Altra Max 128-core), the exploration space quickly becomes massive.

Throughout the sampling process, the average power consumed by the benchmarks fell far short of the maximum power draw. From Figure 4.3 on page 103, a whisker plot of various frequencies and the base and max power draw of processors are shown. The averages do not vary significantly at the lower end. However, after testing for the highest power draw possible using workloads crafted specifically to induce high power consumption. For example, using a special set of instruction set extensions such as AVX512 where the use of such extensions would induce non-typical energy consumption. Using these workloads, we see notable variations between clock frequencies. This fact turns out to be an essential aspect in determining optimal solutions while maintaining parallelism.

$$C \times F \times I \tag{4.1}$$

4.3 Deterministic Meta-Heuristics

Weise [108] presented an extensive and detailed survey of the general algorithms used in meta-heuristic problem-solving. The primary classification described in his book breaks meta-heuristics into two major categories, one being the set of ***deterministic algorithms***, some of which are described as follows:

State-space search :

- This problem is represented as a graph, G, and the states are nodes/vertices in the graph. Edges between the vertices can have a specific direction with which

to transition from one vertex to another.

- Edges can have paths, e.g. single direction, as shown in the blue rectangle in Figure 4.6, where multiple vertices can be reached from different paths; however no state can be reversed, e.g. tic-tac-toe. Alternatively, a multi-directional move among multiple vertices is possible, as in the case of the graph shown in the red rectangle in Figure 4.6, e.g. Travel Salesman problem or chess-piece movement.
- Searches always start from a start node and continues until a goal state is reached using the lowest cost for the given problem.

In the previous algorithms, the nature of determinism does not always yield easy or useful results. For example, branch and bound routines could grow exponentially, wherein it would cease to provide answers in a suitable period. For such problems, deterministic algorithms lack the flexibility of ***probabilistic algorithms***, where a set of general algorithms can provide optimal solutions in polynomial time. The next section provides an overview of these algorithms.

4.4 Probabilistic Algorithms

There are many probabilistic algorithms with many implementations, i.e. generic and specific, for a given problem or a set of problems [98]. Interestingly, guaranteeing an optimal solution for a given problem is not the primary goal, unlike when exhausting a feature value search space. A tic-tac-toe simulation is a good example. After the first round, a probabilistic algorithm can tell us with a certain probability how to find the winning position, i.e., the optimal solution.

Probabilistic algorithms range from broad to specific in the way they help solve optimisation problems. A classical category is *Monte Carlo*. In its most basic version, the algorithm uses randomised choice to explore points in a search space, enabling the inferencing of a distribution of a given function. Sampling is performed several times, and the probability for a given input is output. There are several implementations of Monte Carlo, but the fundamental concepts remain the same. A generic Monte Carlo algorithm uses a tree search [14, 16]. The *Monte Carlo tree search* (MCTS) tree has n-depth, where a simulation walks a number of steps into its structure. Using a function, an optimal node is selected from the root, and this is repeated until a

non-terminal node is reached. Then, the search is expanded from the non-terminal node until the final node is reached. Then, the states of all nodes are back-propagated to the root, narrowing the search space. MCTS and variants simulate problems in which different steps help steer the result, but there should be sizeable depth to the problem to reap the benefits, such as with a chess problem, in which the tree operation with back-propagation changes after each move. When exploring the different feature values of CPUs, e.g. clock speed and number of cores, the problem space can be scaled, depending on the testbed. However, the depth of the problem space is not multidimensional; thus, it cannot be executed in a simulated step-by-step style. For example, if a given combination of cores and clock frequencies turns out to be energy-efficient, it does not imply that this could be fine-tuned by tweaking the clock frequency. Therefore, additional algorithms are required to complete the overview of optimisation solutions to measure and reduce energy consumption. To understand how to appropriately construct these algorithms, the following definitions are provided:

Definition 1 (Solution). *In a search space, a solution produced by an objective function solves a defined goal for a given problem. Thus, a search space contains a set of solutions,* $\mathbb{S} = \{s_1, s_2, \ldots, s_n\}$.

Definition 2 (Optimum/Optimised Solution). *In a search space, an optimised solution produced by an objective function maximises or minimises a defined goal for a given problem.*

Definition 3 (Solution Variables). *In each set of solutions,* $\mathbb{S}$*, each solution is characterised by the values of a number of variables where* $\mathbb{X} = \{x_1, x_2, \ldots, x_m\} \in s_i$*. Where a solution* $\mathbb{S}$ *is an* m *tuple that is two dimensional.*

Definition 4 (Objective Function). *An objective function,* F*, is used to evaluate solutions in an optimisation algorithm to maximise or minimise the output of a given function.*

4.5 Hill Climbing

Hill Climbing [90] is a well-known heuristic search algorithm that evaluates the fitness of set of solutions in a given search space of a given size. The generic Hill Climbing algorithm, 3, generates an initial solution S' using the function, `Select_Initial_State(S)`, the initial solution to be improved. The `Select_Initial_State(S)` as described in Algorithm 4 uses a pseudorandom number generator to choose from a set of frequencies

and cores. For Hill Climbing, the function behaves the same. Having chosen a starting point/solution, the iterations approach the best solution possible, and the process is terminated based on a given rule, τ, e.g. a fixed number of iterations or a threshold. The returned result is the *best solution*, $S^{\star}$.

Hill Climbing in its simplest form can solve many problems with excellent efficiency [32, 109, 110] and high accuracy within short periods. However, there are pitfalls that usually require refining the general implementation to meet specific requirements. Some of these were explored in [108]. For example, the Hill-Climbing algorithm can sometimes get stuck in the local maximum, not allowing it to proceed to additional points. Considering the function plot in Figure 4.7, the global maximum highlighted in green is not obtainable, owing to the gap between it and the global maximum. Some proposed solutions include a random restart during execution.

Definition 5 (Hill Climbing). *Given a set of solutions* $\mathbb{S}$ *in a problem space* P_{space}*, we can derive a solution using the following search properties with the algorithm defined in Algorithm 3*

Algorithm 3: Hill Climbing Algorithm

Input : Termination predicate, τ
Data : Solution set, $\mathbb{S}$
Data : Objective function, F
Output: Final best solution, $S^{\star}$

```
// select an initial state
S' ← Select_Initial_State(S)
while not τ do
    /* generate a new state using S' features (frequency and number of cores) */
    S'' ← Select_New_State(S')
    /* compare states based on one or set of variables – 𝕏 */
    if F(S'') > F(S') then
        S' ←S''
S* ←S'
return S*
```

To implement the simple Hill Climbing algorithm to search for the lowest energy consumption point, we apply the *perturbation function* described in Algorithm 5 and

Algorithm 4: Selecting Initial State in Hill Climbing

```
1 function Select_Initial_State:
  /* Full samples stored in a dictionary                                  */
2 samples ⟵ dict[benchmark][frequencies][number of cores]
  /* Using a pseudo-random generator function, Choice, we select an initial state */
3 return samples[benchmark][Choice(1.2,1.4,...)][Choice(2,3,4,...)]
```

Algorithm 5: Hill Climbing Perturbation Function

```
1  function Select_New_State:
2  new_state ⟵ [ ]
   /* Randomly choosing a set of the values of Frequencies or Cores        */
3  feature ⟵ Choice(Frequencies, Cores)
   /* Selecting a point/state from the feature that was chosen in the previous line */
4  selection ⟵ Choice(feature)
   /* Selecting the next and previous values of the feature with regards to the incumbent */
5  points ⟵ Fetch_Previous_Or_Next_State(selection)
6  if Frequency was selected then
7      new_state ⟵ Choice(points)
8  if Core was selected then
9      new_state ⟵ Choice(points)
10 return new_state
```

Definition 5. The properties of the current best solution, S', are used as a reference point. There are random factors affecting the features used; however, the main goal of the perturbation function is to optimise either the frequency or the number of cores of the most recent solution. In Algorithm 5, Line 2 defines a new variable to hold the new state to be chosen, Line 3 is a pseudorandom choice function that randomly chooses between optimising frequencies or cores, Line 4 is another pseudorandom step that optimises the chosen feature from the previous step and Line 5 selects the next and previous states. For example, if the most recent solution, S', was at 2.1 GHz and 28 cores, the function on Line 5 would offer either 2.6 GHz and 1.8 GHz or 29 and 27 cores. Lines 6–9 determines whether cores or frequencies work better and

makes another random choice based on the available points provided by function `Fetch_Previous_And_Next_State`.

When applying this algorithm to the programs sampled in Chapter 3, we notice a pattern in how the fluctuations in energy/performance affect the overall accuracy of finding the optimal solution, i.e. lowest or near-lowest energy consumption. Table 4.1 show a segment of *Nbody* results from Hill Climbing, i.e. optimal solution accuracy, energy mean, lowest and highest energy values and average optimal solution convergence rate to within 5% of the lowest energy measured. These metrics were collected from 1000 Hill Climbing samples. In the case where the executed code may not have sufficient variations in energy consumption based on core count, we find that Hill Climbing has a slight disadvantage compared with methods. On the other hand, Hill Climbing performed exceptionally well when applied to programs that scaled well as the core count increased, e.g. *SumEuler* in Table 4.3 across all input sizes and with DFT in Table 4.2.

	Hill Climbing				
Metric	Highest Energy	Lowest Energy	Mean Energy	Accuracy	Convergence Rate Avg.
Sample and input					
nbody – 50k	144.77	126.69	139.32	12.80%	4.414
nbody – 80k	349.53	293.91	309.09	62.50%	20.13
nbody – 100k	521.87	483.00	489.40	98.10%	32.113
nbody – 150k	1064.54	996.21	1005.26	99.10%	31.151

Table 4.1: Table showing *Hill Climbing* results ***nbody*** for four different problem sizes using 30% search space or ***84*** points out of ***280***

4.6 Ant Colony Optimisation

The ACO algorithm was inspired by the natural activity of ants. When they search for food resources, they apply a multipath search. At first, their search follows no particular criteria. When a promising path is identified, the discoverer ant returns to the source releasing a pheromone along the way. The remaining ants quickly dispatch along that path, and the process repeats. Dorigo [31] developed an artificial form of

	Hill Climbing				
Metric	Highest Energy	Lowest Energy	Mean Energy	Accuracy	Convergence Rate Avg.
Sample and input					
DFT – 2k	121.34	67.13	68.40	91.70%	39.47
DFT – 3k	371.12	256.49	262.41	91.10%	39.61
DFT – 4k	861.02	816.90	824.78	99.70%	34.11
DFT – 6k	2704.66	2566.94	2591.08	94.20%	37.45

Table 4.2: Table showing *Hill Climbing* results ***DFT*** for four different problem sizes using 30% search space or ***84*** points out of ***280***

	Hill Climbing				
Metric	Highest Energy	Lowest Energy	Mean Energy	Accuracy	Convergence Rate Avg.
Sample and input					
sumeuler – 75k	887.23	815.72	820.27	99.90%	36.54
sumeuler – 80k	1021.12	909.23	916.87	99.60%	37.05
sumeuler – 85k	1126.34	1037.34	1043.65	99.60%	37.99
sumeuler – 90k	1281.82	1141.92	1147.49	99.00%	36.55

Table 4.3: Table showing *Hill Climbing* results ***SumEuler*** for four different problem sizes using 30% search space or ***84*** points out of ***280***

this algorithm, and it became a well-known swarm-based intelligent search.

Considering the previously described scenario, an initial population is used to set the number of solutions in the problem space, as shown in Figure 4.8 on page 109. The objective is then to locate the quickest path to the optimisation goal, see the two paths shown in the first step of Figure 4.8. As we are not aware which one is the shortest, we explore them equally paths (see step two of Figure 4.8) until the optimal solution is found. Hence, the 'pheromone' levels along that path lead to a new optimisation focus, i.e. Figure 4.9 on page 110. To formally define this algorithm, we use Definition 6 and Algorithm 6 on page 107:

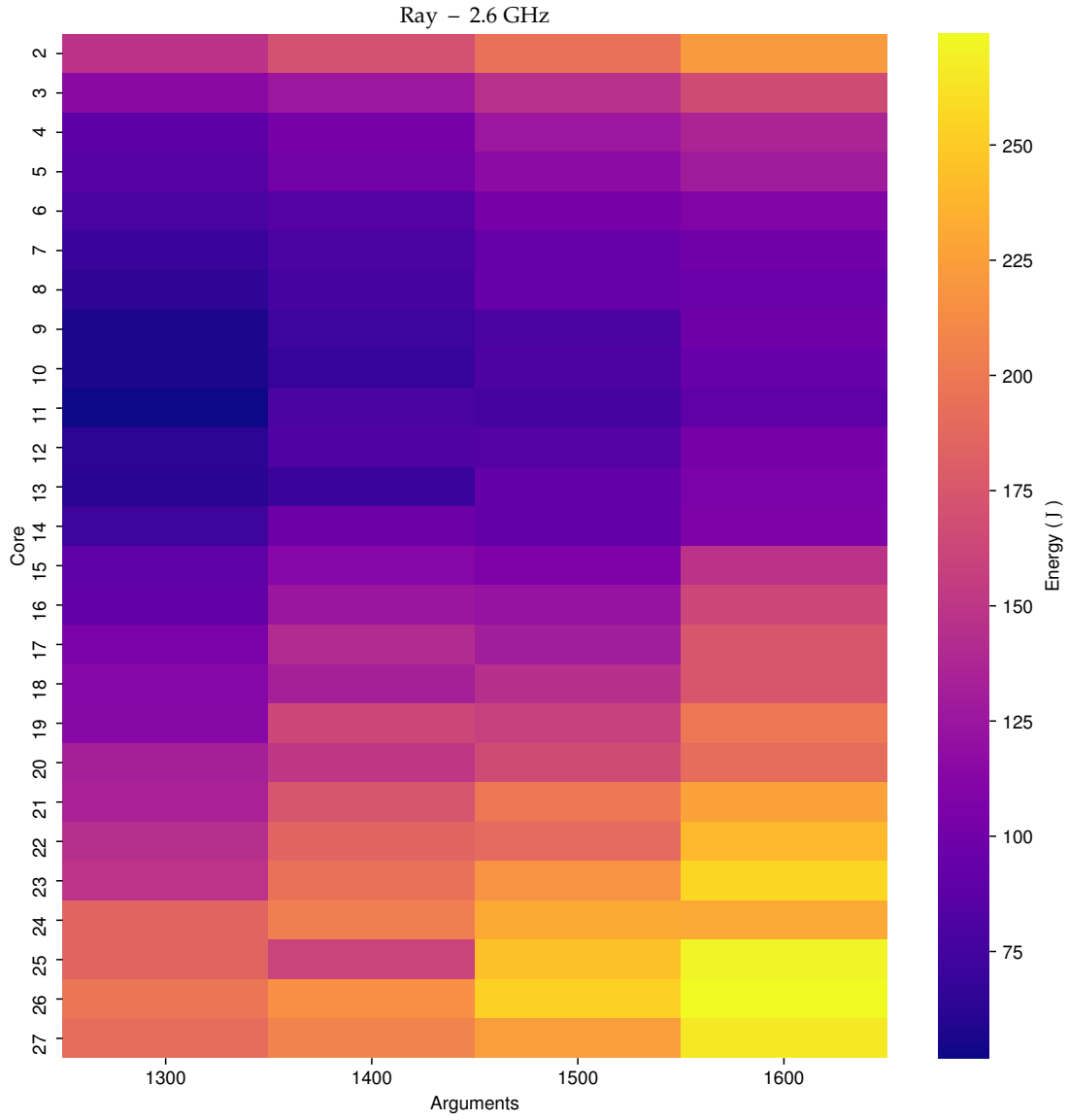


Figure 4.2: Heatmap demonstrating how low-core count impacts energy consumption in *Ray Tracing*

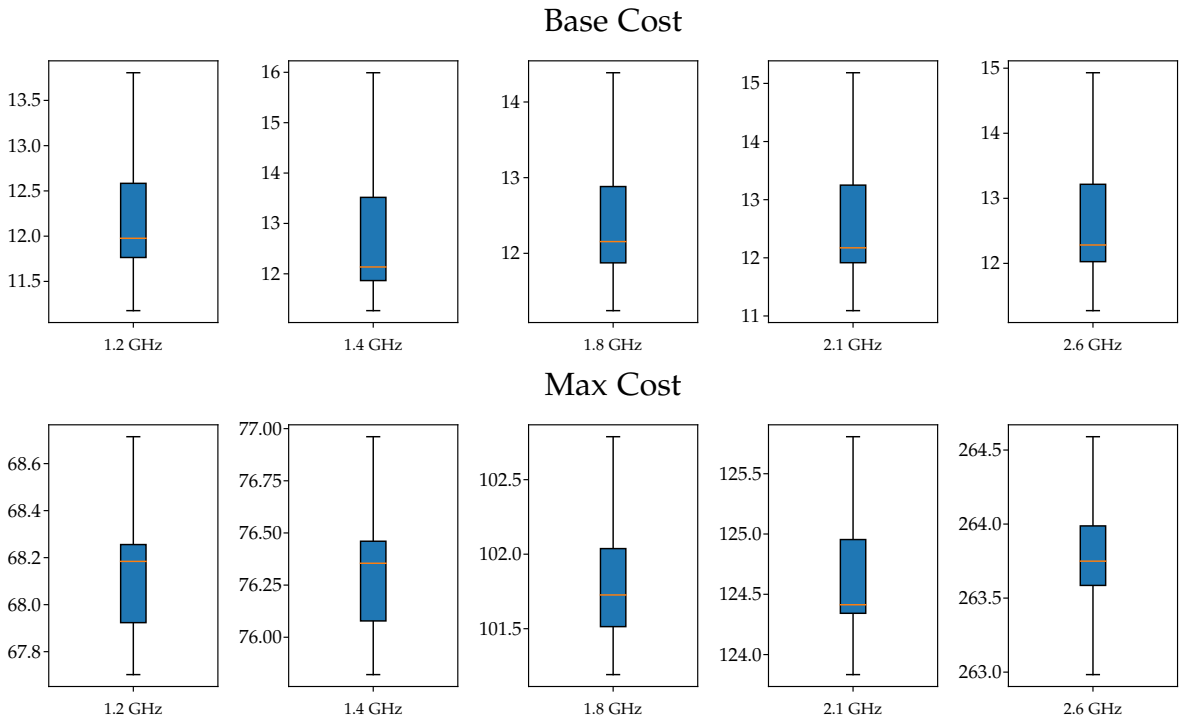


Figure 4.3: Box plot demonstrating average power, watt, for multiple CPU frequencies

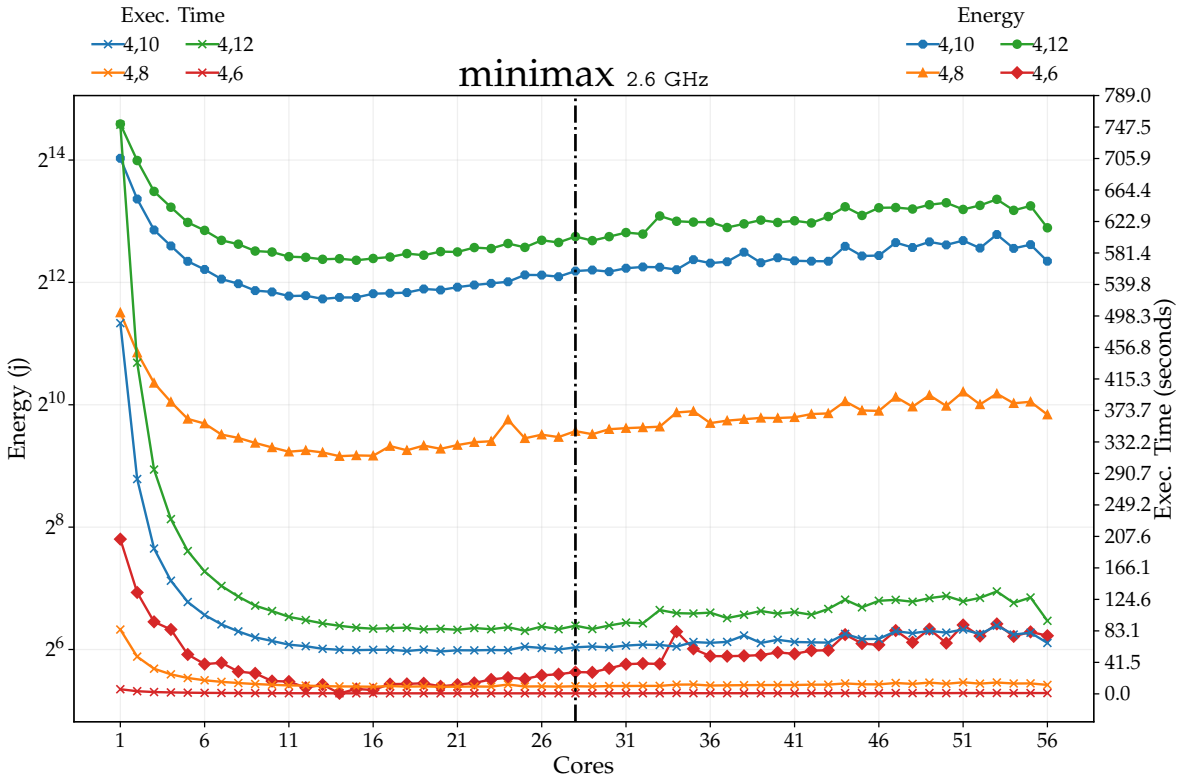


Figure 4.4: Graph showing energy and execution time against number of cores for *Minimax* at ***2.6 GHz***

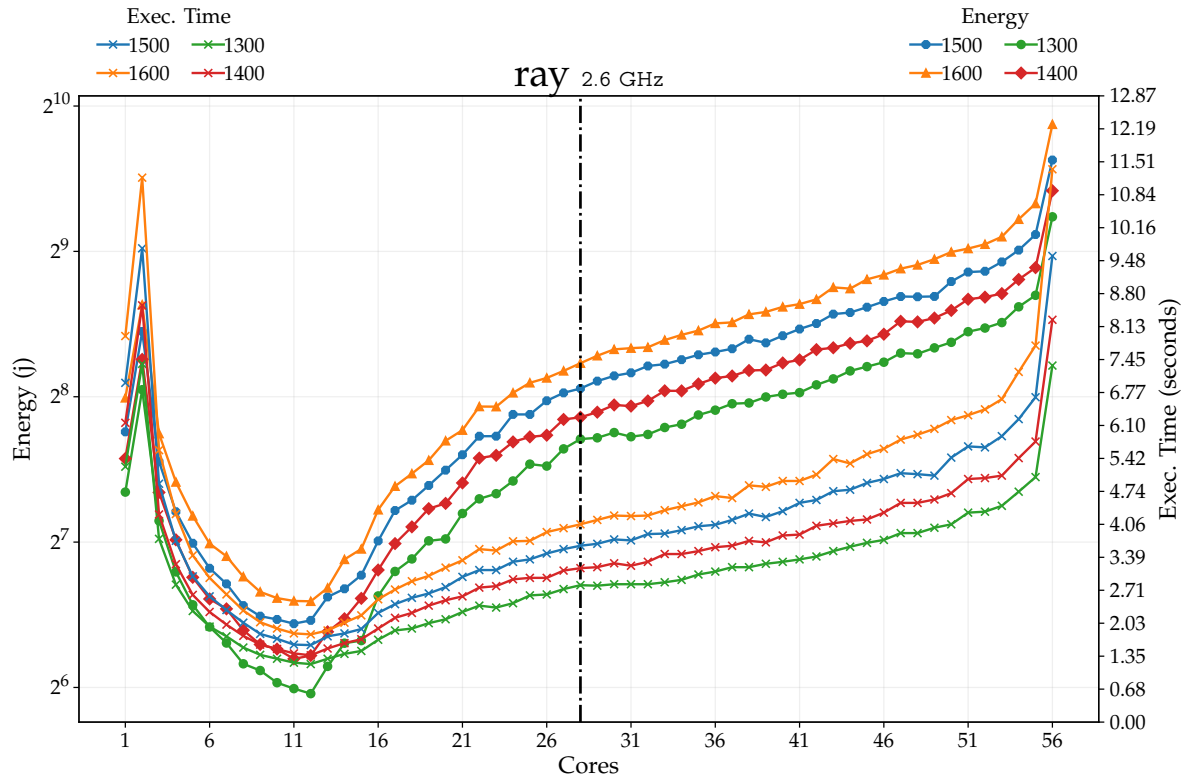


Figure 4.5: Graph showing energy and execution time against number of cores for *Ray Tracing* at ***2.6 GHz***

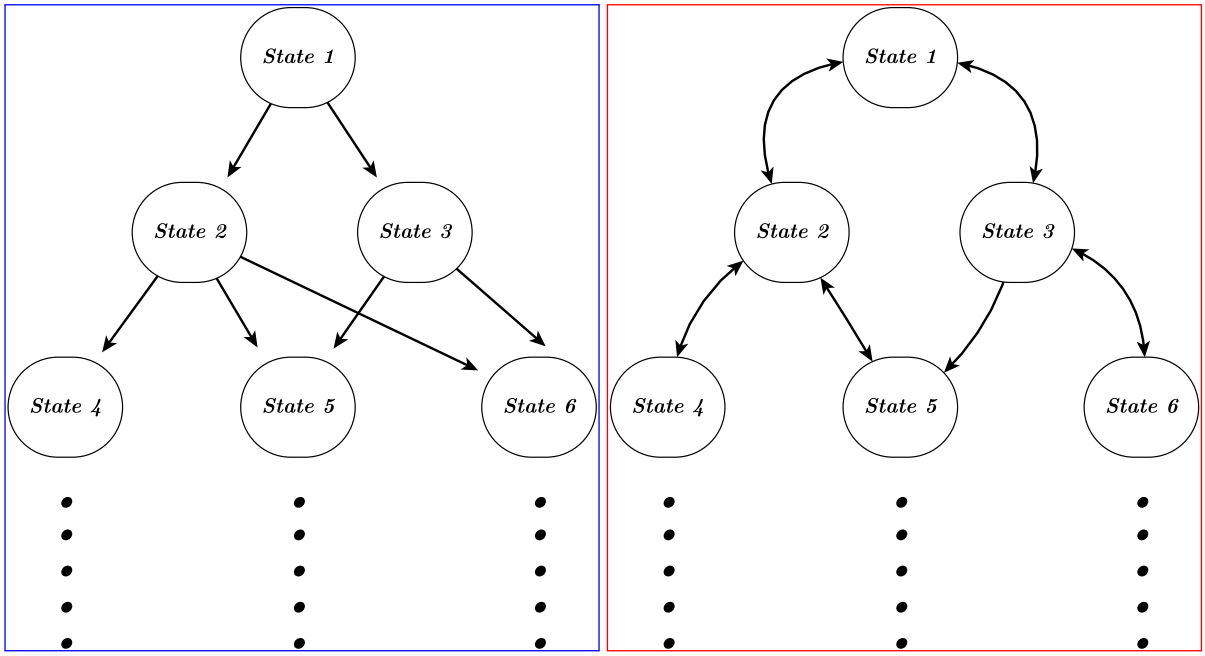


Figure 4.6: Example of graph containing edges and vertices with examples of directional edges

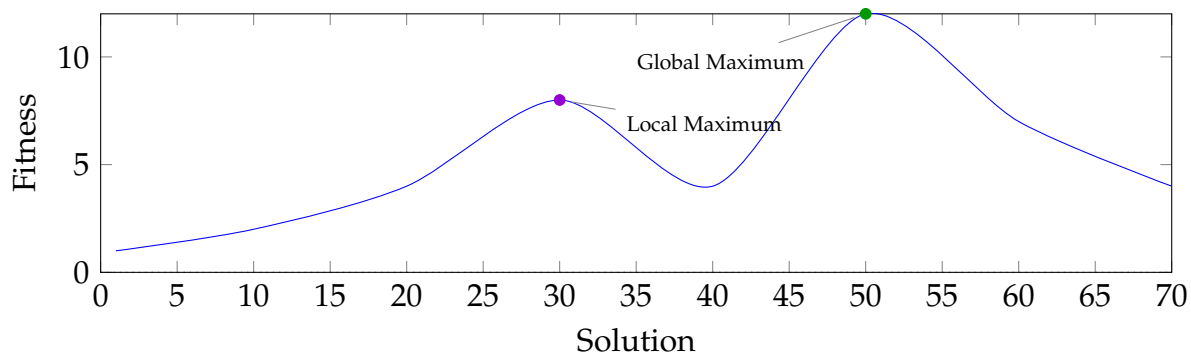


Figure 4.7: Plot for a function in some 2D space

Algorithm 6: Ant Colony Optimisation Algorithm

Data : $\{\mathbb{T}_{size} \mid \mathbb{T}_{size} > 1\}$
Input : Population, $\mathbb{P}$
Input : Bias initial value, Δ
Output: Solution, s

```
// number of nodes representing ant population
Popl ← Create_Population(ℙ)
// a value between 0 and 1
Delta ←Δ
// temporary variable pointing to current solution found
current_solution ⟵ Nil
foreach p ∈ Popl do
    /* function selecting a path based on Δ                    */
    foreach 1 to ℕ do
        new_solution ← Explore_Path(p,𝕋)
        new_delta ⟵ Random Between 0 and 1
        if new_solution resulted in better energy cost then
            current_solution ⟵ new_solution
            // lower delta value indicates better path
            Delta ⟵ Delta – new_delta
        else if Delta ≥ Acceptedthreshold then
            // accepting another state based on delta value
            current_solution ⟵ new_solution
            // Increase delta value provides better chance of finding new solutions
            Delta ⟵ Delta + new_delta
    end
end
s ⟵ current_solution
return s
```

ACO's algorithm for finding the lowest energy consumption point can be achieved as demonstrated in Algorithm 6. As previously shown in the pseudo-algorithm, ACO

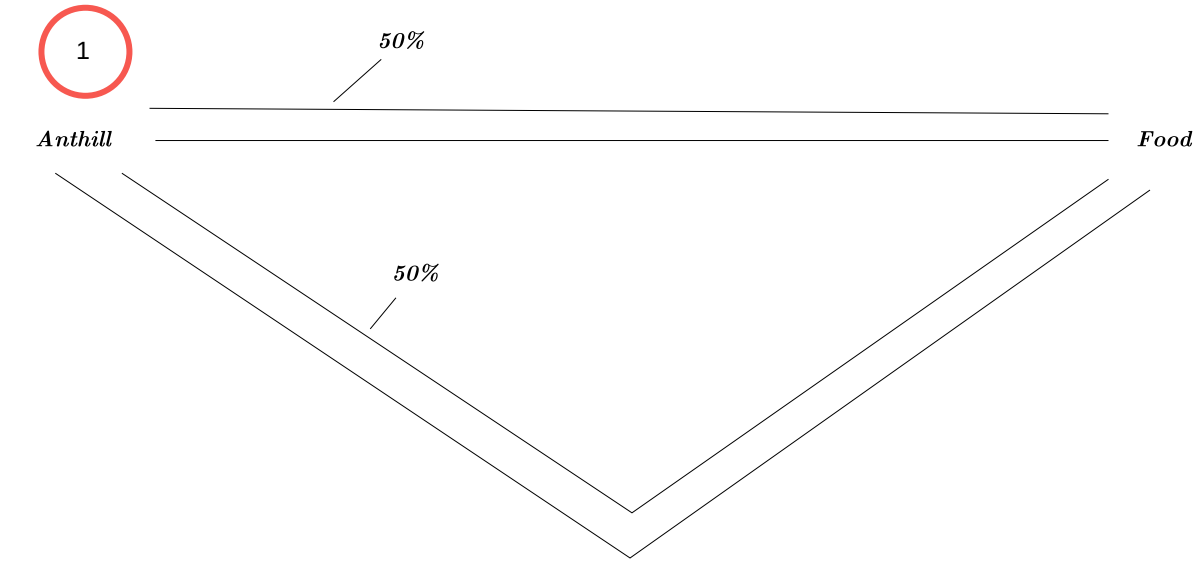
Algorithm 7: Explore Path Process in Ant Colony Optimisation

```
1 function Explore_Path:
2 random_core ⟵ Choice(core_list)
3 random_frequency ⟵ Choice(frequency_list)
4 next_solution ⟵ Fetch_State(random_core, random_frequency)
5 return next_solution
```

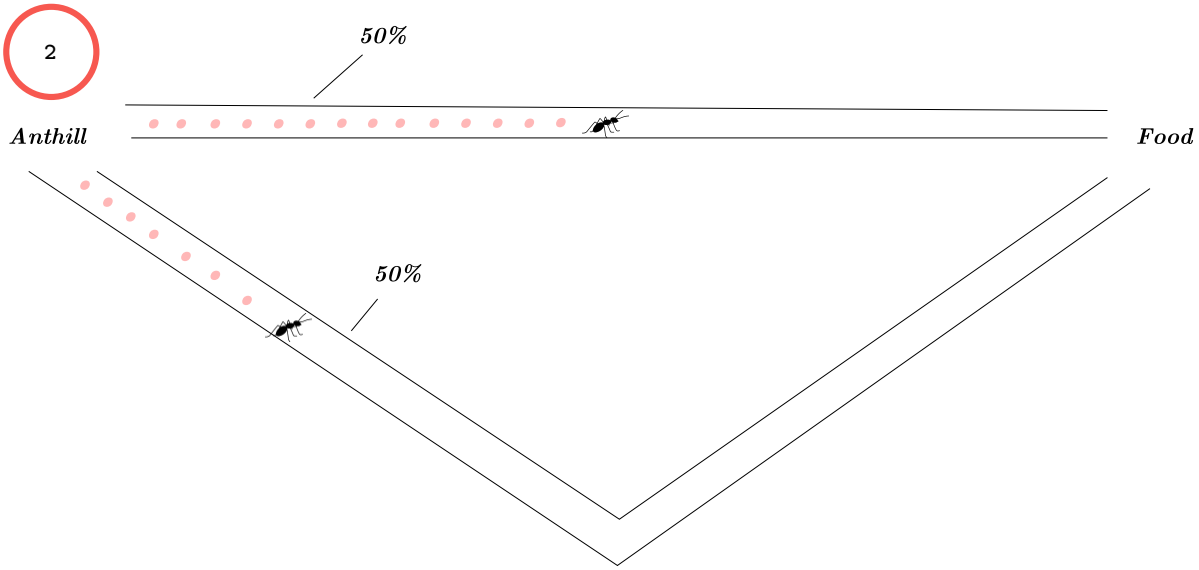
operates using a population of a given size; in this case, we assume that a population was created prior to entering the foreach loop at Line 4, followed by generating a random selection for a set of cores and clock frequencies, setting the initial δ to between zero and one. Then, at Line 3, a variable is used as a placeholder for the solutions found during the loop. In the foreach loop, we iterate over all members of the population, p, as shown in Line 4. At Line 5, another foreach loop is used to explore the various paths for a fixed number of iterations, $\mathbb{N}$, that acts as means of controlling how much of the solution space is searched. In the path exploration loop, we explore a path given a member of the population, p, followed by the generation of a new Δ, which is later used to assess whether we can accept a non-optimal solution to reach better states. The function, Explore_Path(p,$\mathbb{T}$), generates solutions using Algorithm 7, which selects a state randomly from a fixed set of cores and frequencies. In Lines 8–15 of Algorithm 6, the energy values of the newly selected state are assessed, as shown on Line 8. If true, the overall Δ would decrease to avoid accepting new unwanted states to set the new solution. The second case of the if statement accepts the new solutions only if the Δ factor allows it.

Making the initial ACO path selection can be challenging, as it affects the outcome and how quickly the optimal solution can be reached. Additionally, the pheromone trail from a randomly selected starting point could easily lead to a local optimum, where the search gets stuck. Multiple starting paths could be used to mitigate this issue, but it requires more energy consumption at first.

Definition 6 (Ant Colony Optimisation)**.** *Given a tree structure of* n*-depth* $\mathbb{T}_{\text{size}}$ *and a population,* $\mathbb{P}$*, of* n *size, such that* $\text{n} \in \mathbb{N}$ *and a bias variable* Δ*, we can derive a set of population of solution,* s*, where* $\text{s} \in \mathbb{T}_{\text{size}}$ *using Algorithm 6.*



1 Example multi-path problem that simulates ants searching for food

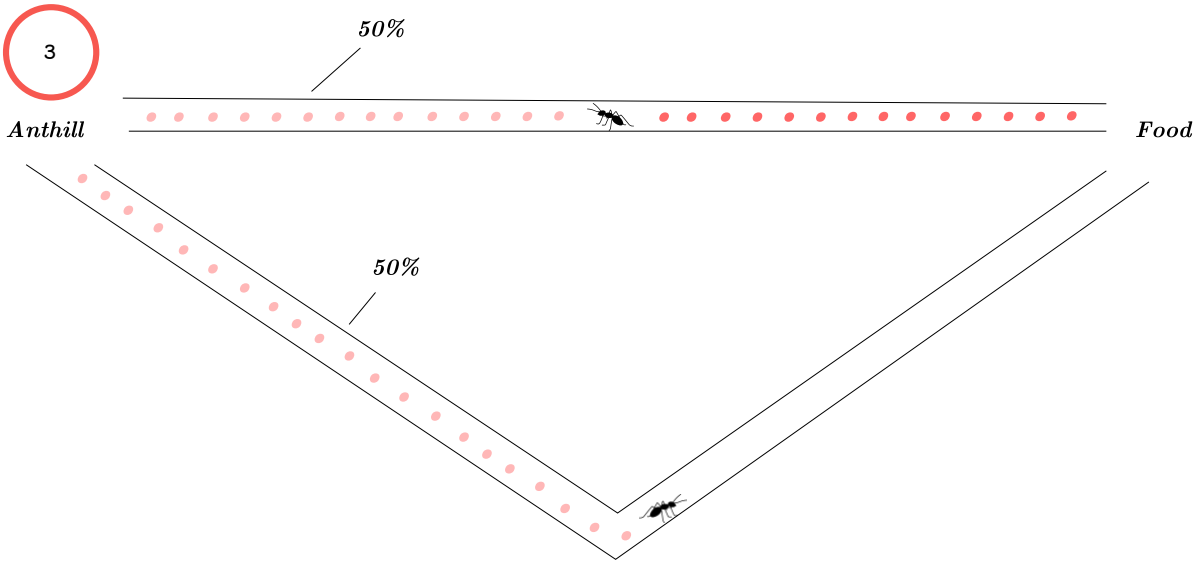


2 Different individuals from available population exploring different paths

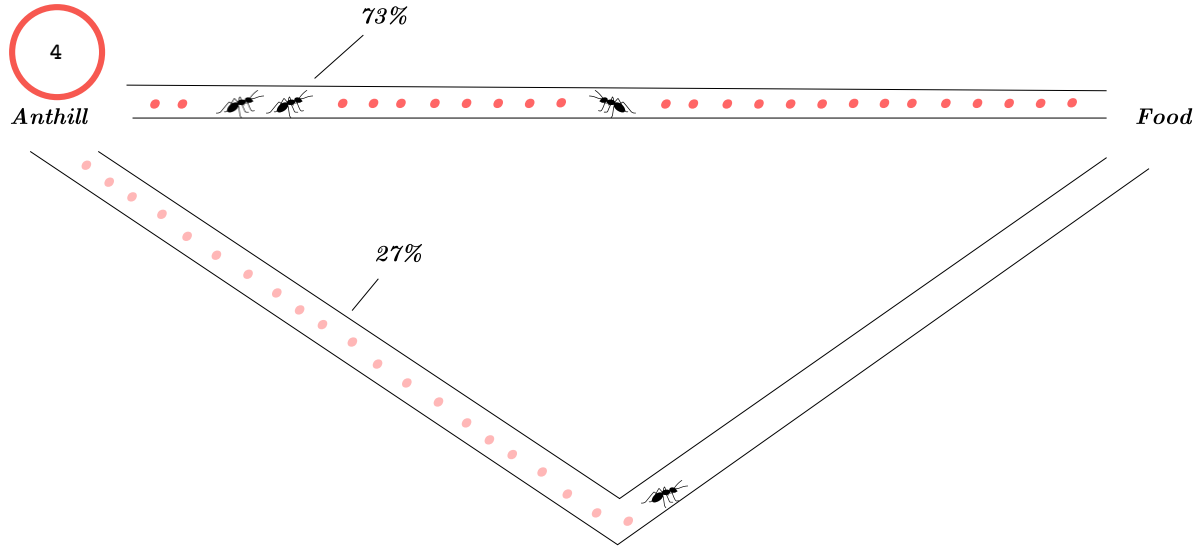
Figure 4.8: First two stages of Ant Colony Optimisation

4.7 Evolutionary Algorithm

Evolutionary algorithms have been applied to many disparate problems. Considering that they use population-based optimisation, the *Genetic Algorithm* is a famous exemplar. These algorithms build a population of a given size wherein individuals are generated based on features representing a seed function to solve a problem. Once



1 Ant successful exploration increases pheromone layers



2 Other individuals in the swarm (ants) following the preferred path based on increased pheromone

Figure 4.9: Continuous process in discovering paths in Ant Colony Optimisation

activated, the seeded population is used to generate more members that are then evaluated for fitness. Predicates are then applied to cull the unfit members. The process repeats for several generations before producing a final population that can solve the problem well. Variants apply more complex definitions and seeds, and the predicates used can be widely varied.

4.7.1 Genetic Mode

Algorithm 8 randomly generates a population representing frequency and number of cores. It is evaluated using a simulation wherein only the fittest may pass their genetic information to the next generation. The process produces edge cases that must be handled after each iteration, e.g. having most or all of the population pass the fitness test, etc. These scenarios are handled case-by-case as not all samples are the same. The process continues by generating a new generation using the previous subset of fit members. To reach completion, the final population indeed minimises energy consumption based on frequency and number-of-core fitness.

More specifically, in Algorithm 8, Line 1 is a partial application function assigned to a variable that is used in the next loop. Its first argument is assigned a value within the lowest 35% energy value. The choice for the 35% was made because of the prior experiments with the algorithm made the population deplete without a chance of finding new members for the population. The second argument, `Execute_sim`, is the solution to be generated. Line 2 sets an initial random population based on properties of frequency and number of cores. Line 5 creates the new offspring generation, as depicted in Algorithm 10. There, Line 2 chooses between cores or features to decide which parent's features are past, the `If` statement (Lines 3–9) handle the case when `Cores` a set of specific features are extracted from each parent and if `Frequency` is selected the operation is swapped. The process of Algorithm 9 creates edge cases, which are handled in a case-by-case basis. Line 6 reconstructs the population based on fitness thresholds. Algorithm 11 is used for reconstruction when needed. There, Line 2 measures the sizes of current and past populations to assess the number of removed and added members. In the Genetic algorithm, Line 3 checks to see if we have fewer than 10 members. Line 4 takes two arguments: prior population and number of low members (integer) selected. Lines 5–8 represent the case where no low members exist from the prior population; thus, we generate a new random population.

Definition 7 (Genetic Algorithm)**.** *Given a search space of* n *size and a fitness function,* f*, and using a population,* $\mathbb{P}$*, of* n *size such that* $n \in \mathbb{N}$*, we derive a population of* $n > 1$*.*

Algorithm 8: Genetic Algorithm Minimising Energy Consumption

```
/* prepare a curried function for GA simulation that accepts only the lowest 35% energy
   from the actual lowest                                                          */
1 Partial_Simulation ⟵ Execute_sim(lowest_35percent)

/* create a population of n-size consisting of randomly selected combinations of cores and
   frequencies from the full dataset                                               */
2 Popl ⟵ Create_Population(ℙ)

3 while not τ do
    /* apply the fitness function on all members of Popl                          */
4   Popl ⟵ Partial_Simulation(Popl)

    /* use the remaining fit members to generate new ones                         */
5   Popl ⟵ Generate_New_Members(Popl)

    /* rebuild the population size back to the defined size if some/all members were not
       fit to be part of the new population and/or if the remaining parents have not
       generated enough children                                                  */
6   Popl ⟵ Rebuild_popl(Popl)
7 end

  /* return final population after terminating                                    */
8 return Popl
```

Algorithm 9: GA Fitness Function / Partial_Simulation

Input : Single member from the population, $\mathbb{M}$
Input : Acceptance threshold, T_h

```
1  function Partial_Simulation:

   /* Randomly choosing a set of the values of Frequencies or Cores   */
2  feature ⟵ Choice(Frequencies,Cores)

   /* Selecting a point/state from the feature that was chosen in the previous line   */
3  selection ⟵ Choice(feature)

   /* Selecting the next and previous values of the feature with regards to the incumbent */
4  points ⟵ Fetch_Previous_And_Next_States(selection, 𝕄)

5  if Frequency was selected then
6  |  new_state ⟵ Choice(points)
7  end

8  if Core was selected then
9  |  new_state ⟵ Choice(points)
10 end

   /* the case where a member is not fit, accepted would be returned as Nil, i.e. removing
      inefficient solutions                                                    */
11 if new_state ≤T_h then
12 |  accepted ⟵ new_state
13 end

14 return accepted
```

Algorithm 10: Genetic Algorithm Generating New Members/Children

Input : Parent One, P_1
Input : Parent Two, P_2

```
new_child ⟵ Nil

/* Generate a value representing a choice between selecting cores from P1 and frequency
   from P2 or the opposite way                                                          */
feature_random ⟵ Choice(Frequency, Cores)

if feature_random = Cores then
    new_child.core ⟵ P1.core
    new_child.frequency ⟵ P2.frequency
else
    new_child.core ⟵ P2.core
    new_child.frequency ⟵ P1.frequency
end

/* retrieving the new state based on new_child selected features from P1 and P2        */
new_child ⟵ Fetch_State(new_child)

return new_child
```

Algorithm 11: GA Rebuild Population

Input : Old Population, $\mathbb{P}$
Input : New Children, C_p

```
function Rebuild_Population:

adjusted_population ⟵ length(ℙ) + length(Cp)

if adjusted_population < 10 then
    /* Attempt to seed Cp from the lowest members from P                                */
    Lowest_P_members ⟵ Get_Lowest_N_Energy(ℙ,1)
    if Lowest_P_members contains no members then
        new_gen ⟵ Generate_N_Random_Members(n=10)
        adjusted_population ⟵ new_gen
    end
end

return adjusted_population
```

4.8 Summary

This chapter introduced the scope of optimisation problem, including the search space of clock frequencies and number of cores. Four algorithms were formulised, covering four types of meta-heuristic processes. Finally, we described the optimisation goals of this thesis.

Chapter 5

Meta-Heuristic – Haskell Results

This chapter presents the Haskell benchmark analysis using the dataset from Chapter 3. Sections 6.1, 5.3, 5.4 and 5.5 respectively explain the sampling processes and metrics used to evaluate the Hill Climbing, ACO and Genetic algorithm sampling processes and discuss the results.

We have noted that statistical modelling has drawbacks when predicting energy consumption, especially considering programs with marginal consumption and input sizes that introduce high variations in energy consumed. We therefore turned to meta-heuristics to find other methods. Modelling circuit switching alone has been classified by some as an NP-hard problem as it will be explained in Section 8.2 of Chapter 8. Hence, predicting energy consumption with any guaranty of accuracy is going to be difficult. Thus, given that extensive datasets present pitfalls when used in conjunction with statistical modelling, we shall use a probabilistic sampling of code. In the next sections, we explain how meta-heuristics works in practice using the algorithms presented in Chapter 4.

5.1 Sampling and Algorithm Structure

Consistency is essential to ensuring that we can reasonably compare one approach to another. As we sampled Haskell parallel *nofib* programs in Section 3, we can use the same dataset to evaluate the new simulations. Because real-time execution varies in duration, we benefit from using extant execution configurations for the new scenarios.

Section 3 included a combined average of 10 samples per design, i.e. each point with specific input was sampled ten times. The mean of each set of 10 points was used in each run. Additionally, as the meta-heuristic algorithms simulates sampling and reporting results of different points in the search space, we collect 1000 results per algorithm to ensure reasonable accuracy and consistency, as they are probabilistic in nature. The meta-heuristic algorithms reviewed in Chapter 4 also vary in the way they report results. Some output single solutions; others output groups of solutions as a population. We report the results based on how they are output.

5.2 Tabu Search

In order to compare the performance of the meta-heuristic algorithms introduced in Chapter 4, we use Tabu search [55], a type of local-search algorithm with similar characteristics to Hill Climbing.

Tabu search can be described as an algorithm that improves solutions in a given search space by exploring neighbouring solutions. While Hill Climbing uses a similar approach, Tabu search has different traits that make it unique. The following points highlight Tabu search characteristics:

- **Tabu list**: The most crucial part of the Tabu search is using a Tabu list or memoisation of visited states/solutions. The Tabu list optimises all the iterations in the search algorithm to find unseen/unvisited solutions that could improve the overall process. In this process, the algorithm keeps a list of all the visited solutions so the algorithm can proceed to find unvisited solutions. There are different ways of using a Tabu list, like keeping separate sets of solutions based on quality, frequency, and partial solution properties. However, one typical approach is to have a Tabu list of all the visited solutions, which would have a given size.

- **Aspiration criteria**: The aspiration criteria control the number of solutions that have been visited/seen in the Tabu list, and when a specific criterion is met, a solution from the Tabu list is removed from the list to allow the algorithm to use it. The Aspiration criteria serve as means of avoiding stagnant states where no solutions can improve a state.

The Tabu search algorithm is one of many local-search algorithms that aim to search a solution space to find optimal or near-optimal solutions. The Tabu search algorithm used to sample the dataset collected from the previous chapters is slightly modified to handle the cases of optimising the frequency/cores search space, which is one approach to how energy costs can be managed and improved. The complete version of the modified code is part of the online appendix.

5.2.1 Tabu Search Result

Although the algorithms have close similarities, the difference in the overall results is unexpected. The first Tabu search sampling was made for 30% of the space, as seen in Table 5.4.

The benchmarks had varying results. However, it can be seen that most of the benchmarks were hindered from reaching an equivalent level of accuracy, similar to Hill Climbing. For example, the highest accuracy achieved was Ray's 1500, where the accuracy reached was 70.20% which is a significant improvement compared to Hill Climbing's version of 37.60%. Nevertheless, upon examining the rest of the benchmarks, we observe that most accuracy levels rarely reach anything above 30%. When a result like Ray's 70% is produced, the result is only achieved for a single input size, unlike Hill Climbing, where consistent patterns can be identified from one benchmark to another. It is unclear why the Tabu algorithm struggles to reach a similar accuracy to Hill Climbing.

5.3 Hill Climbing

In Hill Climbing, we sampled a dictionary of input sizes categorised by the number of cores and clock frequencies. In each cycle, we simulated a state selection containing metadata of pre-sampled benchmarks, and we used those describing energy consumption as the result of a single run. This approach used the following steps:

- For each Hill Climbing sample, we ran the algorithm for a specific percentage of a search space of 280 items so that we could obtain appropriate accuracy measurements. We sampled at 5, 10 and 30%, translating to 14, 28 and 84 steps out of 280, respectively.

- In each Hill Climbing run, we aimed to optimise solutions within 5% of the actual lowest value across all cores for a given problem size and within an upper bound of $\pm$5%.
- Hill Climbing simulates sampling program execution using the CSV dataset produced in Chapter 3. This facilitated selection comparisons of energy readings, as they did not differ much from one sample to another, i.e. from 5 to 10%.
- Hill Climbing results were sampled 1000 times per program/problem size to avoid anomalies.
- In each sample, we measured accuracy using the reference point defined earlier: within 5% of the lowest energy point. This provided metrics on the sampled dataset as CSV data for each program/problem size.

Analysis in this section is based on three *Hill Climbing* tables: Table 5.1 on page 127, Table 5.2 on page 129 and Table 5.3 on page 131. All data referenced in the text are presented in the tables.

5.3.1 First Category: Model Scalability

For this category, we grouped the benchmark results according to improved scalability with the increase of core count. We also examined raw values differently than that which was presented in Chapter 3, where energy was plotted against speedup. We instead looked at execution time and energy consumed.

Parfib: Parallel Fibonacci is an ideal scalable parallelism case. However, we provide a new perspective, as shown in Figure 5.1, where Parfib showed near-perfect improvements. The overall Hill Climbing performance was exceptional . At a 5% search space (14 steps), Parfib performed well for a problem size of 50, achieving 93.60% accuracy. However, the small search space was a limiting factor, scoring 34.50, 38.00 and 37.20% accuracies for problem sizes 52, 54 and 56 steps, respectively. The average rate of convergence showed improvements in problem sizes larger than 50 steps, capped at around 2.4 steps compared with 50, where the optimal solution was improved over an average of 5.1 steps. This limitation abated in the next two levels. At a 10% search space, the number of search steps increased to 28, and we witnessed significant improvements for the 50-step size, resulting in an accuracy of 96.90%. There were considerable gains for the remaining problem sizes of 52, 54 and 56 steps, which

achieved accuracy results in the 73rd percentile, as shown in Table 5.2. The final level for a 30% search space required 84 steps of 280, resulting in near-perfect results for all problem sizes. 50 steps resulted in 100% accuracy, meaning that we found the optimal solution within 5% of the actual, followed by 99.80% for 56 steps, 99.40% for 52 steps and 98% for 54 steps. An important optimising factor was observed in the convergence rate where most samples stopped improving beyond 38 steps. This may indicate that the percentage of the search space to consider can be reduced to about 14% of the search space, or a total of 38 steps.

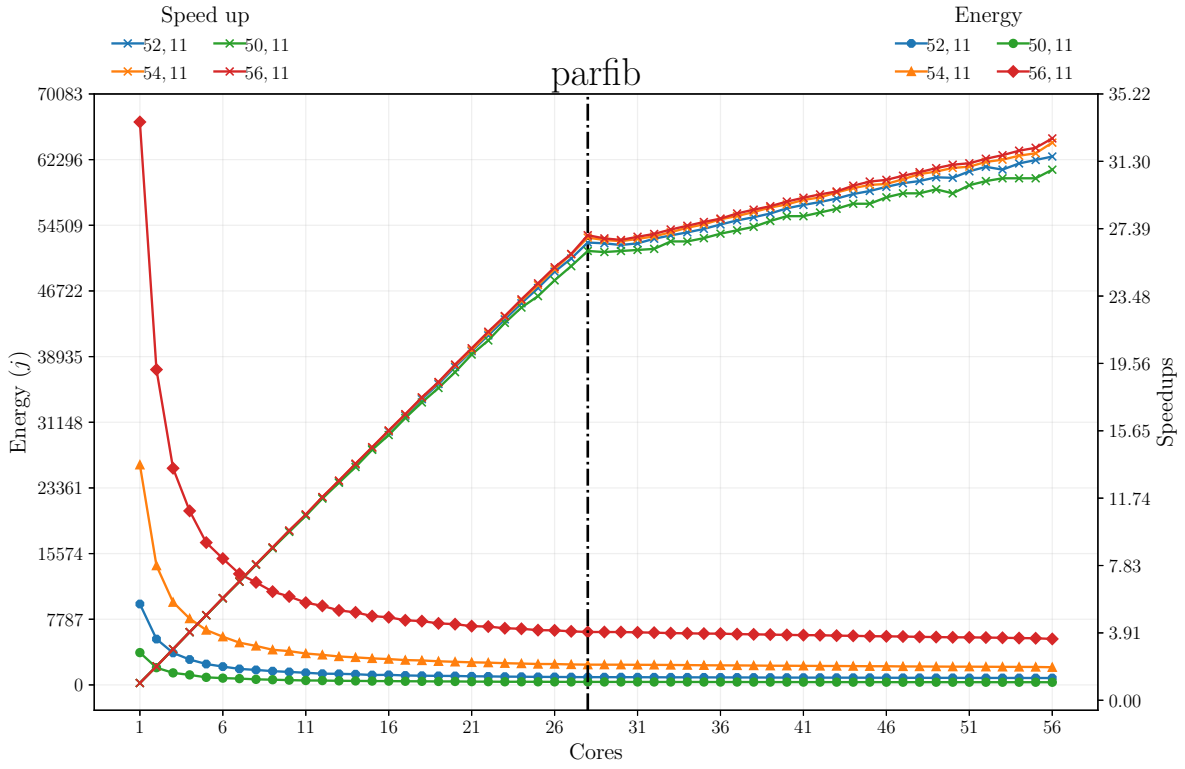


Figure 5.1: Plot showing energy and speedup vs. cores for ***Parfib*** with four different problem sizes sampled at 2.6 GHz

SumEuler: The Euler summation showed a similar pattern to that of Parfib. Starting with a 5% search space, we saw a near-duplicate pattern in which the smallest problem size had the highest frequency. 75K steps resulted in 77% accuracy , followed by 85K steps with a 42.90% accuracy. 80K steps had a 41.30% accuracy and 90K had 34.40%. The 10% search space continued to improve the overall accuracy, with 75K steps achieving the highest at 86.80%. The others achieved accuracies in the 70th percentile. Finally, the 30% search space showed a result similar to Parfib, in which all problem

sizes demonstrated accuracies within the 90th percentile. 75K steps achieved 99.90% accuracy, followed by 80K and 85K, both at 99.60%. Finally, for the largest problem size the accuracy metric reached 99.0%. In addition to the case of improved accuracy, we noticed that the average convergence rate was similar to that of Parfib, in that most cases achieved the optimal solution at around 36 or 37 steps.

DFT: With DFT, we observed a similar case of increasing steps enhancing accuracy. However, DFT started with a low accuracy compared with the two previous benchmarks. With 5% of the search space, we observed an unexplained similarity in accuracies, wherein the first 2K steps resulted in 41.70% accuracy. 4K steps achieved a 43.90% accuracy, 6K achieved 20.80% and 3K achieved 20.20%. The problem sizes were interestingly divided between accuracy percentiles: 2K and 4K steps produced similar results as the other two inputs. When extending the search space to 10%, we observed a different pattern, where 4K steps achieved an 83.70% accuracy, 6K achieved 57.40%, 3K achieved 41.60% and 2K achieved 41.30%. In this case, the operation at 4K steps well-outperformed the other problem sizes. This result may have been affected by having apparently used a smaller standard deviation. We also saw a similar standard deviation value for 2K steps. Nevertheless, accuracies lower than that achieved with 4K steps and energy values smaller than that achieved with 2K steps may explain this, especially at higher core counts where the mean energy of the complete dataset was not substantial enough for optimisation. When adjusting for the final 30% search space, we saw high accuracies for all programs. 4K steps again produced the best accuracy at 99.70%, followed by 6K at 94.20%, 2K at 91.70% and finally 3K at 91.10%. After examining the average convergence rates, we found that DFT pushed towards 14% of the search space, apart from 4K steps, where the average convergence rate was around 35.

Minimax: Minimax also started with low accuracy, ranging from 22.10 to 49.50% for the four problem sizes sampled. We saw moderate accuracies with a 5% search space. With a 10% search space, we saw significant improvements for all problem sizes apart from 6M steps, where the low energy readings may have affected progress. Finally, the results of the 30% search space demonstrated top accuracies for 8, 10 and 12 within the 90th percentile, whereas 6M steps remained in the mid-60th percentile.

PRSA: In the case of Parallel RSA, we witnessed a slightly different accuracy pattern. Starting with the first level at a 5% search space, the lowest accuracy was

achieved at 2M steps at 9.90%, followed by 9M at 26.70%, 8M at 76.60% and 6M at 77.60%. For the 2M-step problem size, we found that the convergence rate was already low at 0.699, as reflected by the low accuracy result. Next, with a 10% search space, we observed a substantial increase in accuracy, which was obvious with 6M and 8M steps, which produced 95.60 and 95.90% accuracies, respectively, followed by 9M steps at 57% and 2M at 27.20%. Finally, as seen with other results, PRSA reached the highest accuracy values at 30%, where 6M and 8M steps both achieved 100% accuracy, followed by 9M at 75.50% and 2M at 50.70%. For the 2M-step case, the results appeared to have been affected by small number energy readings, as the optimal solution was marked at 60.942 J relative to the lowest level at 58.04 J. As the mean of the algorithm dataset was 61.47 J, this value appears to have affected the result.

Nbody: with Nbody, at 5%, we observed accuracy values ranging from 5 to 63.70%. Larger problem sizes performed better, affected by low-energy readings and small standard deviations. The 10% level demonstrated an improvement over the previous one, where the two largest problem sizes achieved accuracies in the 90th percentile. 80K steps showed modest progress, followed by 50K, which had mediocre performance. Finally, 30% demonstrated the best performance for the largest two problem sizes of 150K and 100K steps. The rest followed the same pattern of 80K steps having modest gains and 50K steps lacking beneficial accuracy improvements.

5.3.2 Second Category: Irregular Patterns

In this category, we targeted programs having unpredictable or irregular results affected by overall execution performance and scaling, where energy readings and standard deviations may have affected finding the optimal solutions.

Queens: Although Parallel Queens has shown progress in scaling over an increasing core count, it remains limited by other factors, such as the unpredictable performance of some clock frequencies where energy readings alternate in some cores. With a 5% search space, all problem sizes failed to achieve accuracies greater than 50%. The highest was 21.10% for a problem size of 16 steps. Moving to the next search space level, we witnessed slight improvements across all problem sizes, but there were no changes in the pattern as 14 and 15 steps produced the lowest accuracies, and 13 and 16 steps had the highest. With a 30% search space, the same thing occurred, except

that 16 steps achieved an significant accuracy increase from 35 to 75%; the remaining problem sizes produced modest results.

Partree: Partree is considered inefficient, as it spends a considerable amount of time performing garbage collection. With a 5% search space, Partree produced low accuracy levels, but it performed better than matmult. For example, its RMSE values were among the highest of all datasets. At a 10% search space, slight improvements were seen apart from the 700 input, which nearly doubled its accuracy to 63%. The last level at a 30% search space achieved considerable progress with inputs of 700 and 800 steps, whereas those of 600 and 650 steps showed moderate improvements.

Matmult: Matrix multiplication showed increasing variations in energy readings across all clock frequencies and all problem sizes. However, we found a pattern of cores oscillating between high and low readings. At a 5% search space, no problem size achieved an accuracy close to 10%. At the 10% search-space level, we observed small gains in accuracy ranging between 14.90 and 32.70%. This increase was affected by the increase of the search space. However, there was still a problem with identifying appropriate accuracy levels. At a 30% search space, the pattern of irregular energy readings continued, possibly caused by the marginal execution time increasing or decreasing alongside other factors, e.g. standard deviation.

QuickSort: Quicksort is an amplified case of Partree, where the initial accuracy results showed that 500K was unable produce a single solution, whereas others ranged between 5 and 10% accuracy. The 10% search-space level was similar, with 500K steps remaining at 0% accuracy and others having small increases. Lastly, at a 30% search space, 500K were unable to produce solutions, as evidenced by the smallest value found in the dataset, 33.86 J, where the 5% margin was 30.55 J, which may explain why the solution remained stuck in the local optima. A feature that seems to correlate with problem sizes achieving moderate accuracy is their higher standard deviations.

Ray: The Ray Tracing samples produced a distinct pattern in which energy and execution time spiked at certain stages. Both improved when total energy consumed and execution time reduced before increasing again. This pattern reflects Ray's unique behaviour with regards to specific problem sizes. A 5% search space showed similar results as the other benchmarks, where two inputs had lower percentages,

and the other two had slightly higher ones. Specific problem sizes affected overall execution times, resulting in energy differences per input. With a 10% search space, the gap between 1300 and 1600 and 1400 and 1500 steps increased, and the highest two accuracies had rapid improvements, compared with the other two. A 30% search space behaved the same way.

Name	Raw Dataset			Hill Climbing Dataset						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Parfib 50	4164.59	236.05	502.39	0.088	3.71	252.49	236.05	242.87	93.60%	5.13
Parfib 52	10882.10	595.55	1307.79	0.367	12.61	649.00	595.55	625.23	34.50%	2.39
Parfib 54	27720.10	1540.42	3425.92	1.541	83.40	2112.48	1540.42	1647.09	38.00%	2.66
Parfib 56	72365.80	4014.32	8938.22	3.572	169.44	5696.40	4014.32	4278.47	37.20%	2.62
Queens 13	67.97	8.61	14.86	0.66	0.014	10.91	8.61	9.70	18.40%	1.02
Queens 14	440.15	50.92	87.37	4.26	0.096	79.33	50.92	58.20	6.00%	0.39
Queens 15	3163.03	354.66	628.09	0.579	17.79	462.93	354.66	402.29	6.30%	0.41
Queens 16	22008.80	2910.77	4855.65	3.543	162.58	3934.70	2910.77	3176.20	21.10%	1.13
Matmult 1.5k	1238.54	128.25	264.87	0.303	12.02	187.93	128.25	152.04	9.00%	0.43
Matmult 2k	3536.63	286.88	620.39	0.607	22.22	390.32	286.88	335.37	9.40%	0.53
Matmult 2.5k	5615.74	549.19	1197.87	0.899	42.30	699.77	549.19	615.88	4.10%	0.28
Matmult 3k	14183.10	932.49	2073.52	2.298	82.07	1504.32	932.49	1116.91	8.70%	0.51
Minimax 6	289.66	30.00	58.68	2.58	0.050	39.46	30.18	33.52	22.10%	1.07
Minimax 8	3388.07	443.91	834.34	0.723	47.59	653.18	443.91	485.94	45.30%	2.50
Minimax 10	18606.70	2673.08	4901.15	3.948	246.38	4316.63	2673.08	2917.24	43.50%	2.87
Minimax 12	27269.70	4094.10	7511.70	4.743	267.84	6410.61	4094.10	4413.27	49.50%	3.44
DFT 2k	1248.80	65.29	122.81	0.057	2.51	78.58	67.13	69.66	41.70%	2.17
DFT 3k	2626.52	256.49	421.19	1.098	58.25	468.31	256.49	333.42	20.20%	0.77
DFT 4k	6731.86	810.06	1135.40	0.791	39.10	975.20	816.90	867.50	43.90%	2.75
DFT 6k	21885.10	2514.16	3512.12	2.382	86.95	3021.50	2566.94	2704.50	20.80%	1.52
PRSA 2M	505.41	58.04	108.80	0.123	4.96	85.27	58.04	67.68	9.90%	0.69
PRSA 6M	1524.60	190.81	324.17	0.166	11.49	263.58	190.81	199.78	77.60%	4.42
PRSA 8M	2039.98	254.53	428.22	0.228	16.46	345.26	254.53	266.01	76.60%	4.40
PRSA 9M	2325.16	274.08	485.49	0.331	15.96	355.19	274.08	298.39	26.70%	1.82
Quicksort 500k	706.92	29.16	264.93	0.680	46.61	322.56	33.86	66.52	0.00%	0.0
Quicksort 1M	1882.37	73.52	699.90	1.854	135.89	630.14	73.52	163.39	5.20%	0.4
Quicksort 3M	8195.00	289.17	2945.35	6.687	491.49	3491.47	289.17	611.20	10.10%	0.67
Quicksort 6M	17104.40	655.49	7074.05	9.559	596.86	2771.06	655.49	1246.38	6.80%	0.37

	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Ray 1300	660.36	47.99	197.61	0.457	31.13	199.85	47.99	73.32	11.20%	0.83
Ray 1400	785.25	52.50	228.70	0.408	23.28	126.87	52.50	79.77	4.60%	0.37
Ray 1500	891.90	63.41	262.71	0.621	41.92	267.41	63.41	98.28	7.90%	0.54
Ray 1600	1125.44	73.62	298.49	0.683	48.30	322.78	73.62	109.28	16.20%	1.11
Nbody 50k	1808.52	126.69	254.78	0.209	6.88	210.72	126.69	143.67	5.00%	0.23
Nbody 80k	4629.35	293.91	629.65	0.556	24.92	511.09	293.91	335.95	16.70%	1.09
Nbody 100k	6711.05	483.00	952.77	0.456	28.46	724.76	483.00	511.20	53.00%	3.65
Nbody 150k	15903.00	996.21	2084.97	0.747	46.00	1225.62	996.21	1043.03	63.70%	4.61
Partree 600	23139.10	635.06	4137.45	11.829	850.23	4316.83	635.06	1232.93	11.10%	0.64
Partree 650	33083.30	791.78	5039.03	13.401	942.80	4502.98	791.78	1497.24	14.50%	0.96
Partree 700	57015.10	940.72	6244.34	16.876	1238.39	6689.99	940.72	1756.38	30.20%	1.96
Partree 800	92496.00	1218.31	8685.94	21.809	1549.06	6693.87	1218.31	2346.43	17.30%	1.30
SumEuler 75k	16477.30	815.72	1888.40	0.357	25.27	916.46	815.72	834.29	77.00%	3.62
SumEuler 80k	17794.00	909.23	2171.43	1.972	145.53	4238.26	909.23	1003.25	41.30%	3.04
SumEuler 85k	21015.70	1037.34	2442.17	1.712	117.98	2755.20	1037.34	1130.72	42.90%	2.99
SumEuler 90k	22457.00	1141.92	2766.56	1.005	47.97	1313.95	1141.92	1216.02	34.40%	2.38

Table 5.1: Results of sampling 5% of the search space using **Hill Climbing** over the raw benchmark dataset

Name	Raw Dataset			Hill Climbing Dataset						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Parfib 50	4164.59	236.05	502.39	0.062	4.01	267.84	236.05	239.71	96.90%	13.11
Parfib 52	10882.10	595.55	1307.79	0.274	13.74	703.68	595.55	615.32	77.00%	10.19
Parfib 54	27720.10	1540.42	3425.92	0.749	40.96	1780.52	1540.42	1591.92	73.50%	9.30
Parfib 56	72365.80	4014.32	8938.22	1.906	94.68	4589.64	4014.32	4152.41	76.30%	9.78
Queens 13	67.97	8.61	14.86	0.52	0.010	10.95	8.61	9.38	24.30%	2.87
Queens 14	440.15	50.92	87.37	0.058	2.38	65.09	50.92	55.40	13.30%	1.72
Queens 15	3163.03	354.66	628.09	0.442	17.51	433.54	354.66	389.29	21.20%	2.68
Queens 16	22008.80	2910.77	4855.65	2.572	119.17	3506.03	2910.77	3102.72	35.00%	4.20
Matmult 1.5k	1238.54	128.25	264.87	0.231	10.09	175.19	128.25	145.85	14.90%	1.67
Matmult 2k	3536.63	286.88	620.39	0.532	29.21	492.19	286.88	323.35	32.70%	3.79
Matmult 2.5k	5615.74	549.19	1197.87	0.954	43.37	768.07	549.19	620.86	21.30%	2.33
Matmult 3k	14183.10	932.49	2073.52	1.604	79.92	1279.05	932.49	1048.53	26.80%	3.07
Minimax 6	289.66	30.00	58.68	1.40	0.029	41.25	30.18	32.15	35.80%	4.74
Minimax 8	3388.07	443.91	834.34	0.264	17.43	594.05	443.91	459.26	75.60%	9.47
Minimax 10	18606.70	2673.08	4901.15	1.644	103.09	3414.09	2673.08	2774.25	76.20%	9.39
Minimax 12	27269.70	4094.10	7511.70	2.552	161.18	5411.73	4094.10	4249.98	84.80%	10.19
DFT 2k	1248.80	65.29	122.81	0.285	20.10	142.27	67.13	80.24	41.30%	5.74
DFT 3k	2626.52	256.49	421.19	0.658	44.61	453.22	256.49	293.32	41.60%	6.16
DFT 4k	6731.86	810.06	1135.40	0.366	18.34	941.51	816.90	836.43	83.70%	10.94
DFT 6k	21885.10	2514.16	3512.12	1.561	55.39	2955.19	2566.94	2639.60	57.40%	8.10
PRSA 2M	505.41	58.04	108.80	0.076	3.56	78.27	58.04	63.66	27.20%	3.26
PRSA 6M	1524.60	190.81	324.17	0.069	4.36	237.61	190.81	194.97	95.60%	10.79
PRSA 8M	2039.98	254.53	428.22	0.070	4.51	299.75	254.53	258.69	95.90%	11.43
PRSA 9M	2325.16	274.08	485.49	0.177	8.64	346.11	274.08	287.01	57.00%	7.07
Quicksort 500k	706.92	29.16	264.93	0.220	13.75	186.69	33.86	42.80	0.00%	0.0
Quicksort 1M	1882.37	73.52	699.90	0.549	39.38	531.42	73.52	101.41	14.60%	2.06
Quicksort 3M	8195.00	289.17	2945.35	1.867	140.06	2166.15	289.17	374.59	26.80%	3.28
Quicksort 6M	17104.40	655.49	7074.05	5.231	400.18	5037.45	655.49	881.54	16.70%	2.17

	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Ray 1300	660.36	47.99	197.61	0.152	10.42	122.99	47.99	56.39	33.30%	4.34
Ray 1400	785.25	52.50	228.70	0.223	14.17	169.06	52.50	65.98	12.80%	1.59
Ray 1500	891.90	63.41	262.71	0.209	14.07	189.12	63.41	75.20	18.50%	2.61
Ray 1600	1125.44	73.62	298.49	0.212	15.59	256.84	73.62	83.87	39.10%	5.22
Nbody 50k	1808.52	126.69	254.78	0.164	4.27	154.05	126.69	140.46	7.70%	0.89
Nbody 80k	4629.35	293.91	629.65	0.351	17.89	385.55	293.91	319.04	31.40%	4.10
Nbody 100k	6711.05	483.00	952.77	0.151	8.95	535.96	483.00	492.78	93.20%	12.66
Nbody 150k	15903.00	996.21	2084.97	0.278	19.32	1151.05	996.21	1011.11	94.30%	12.82
Partree 600	23139.10	635.06	4137.45	3.635	286.57	3147.22	635.06	776.10	27.80%	3.47
Partree 650	33083.30	791.78	5039.03	4.491	349.56	4385.69	791.78	974.99	35.00%	4.34
Partree 700	57015.10	940.72	6244.34	4.352	346.20	4515.58	940.72	1103.23	63.30%	7.90
Partree 800	92496.00	1218.31	8685.94	7.068	555.73	5938.55	1218.31	1495.59	47.70%	6.24
SumEuler 75k	16477.30	815.72	1888.40	0.407	26.42	974.64	815.72	839.76	86.80%	11.65
SumEuler 80k	17794.00	909.23	2171.43	0.528	34.69	1243.76	909.23	940.01	78.80%	10.78
SumEuler 85k	21015.70	1037.34	2442.17	0.543	37.51	1311.12	1037.34	1066.84	78.60%	10.58
SumEuler 90k	22457.00	1141.92	2766.56	0.745	51.79	1562.01	1141.92	1181.94	72.70%	9.70

Table 5.2: Results of sampling 10% of the search space using **Hill Climbing** over the raw benchmark dataset

Name	Raw Dataset			Hill Climbing Dataset						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Parfib 50	4164.59	236.05	502.39	0.019	1.25	243.78	236.05	237.11	100.00%	36.48
Parfib 52	10882.10	595.55	1307.79	0.124	7.31	629.99	595.55	603.61	99.40%	37.29
Parfib 54	27720.10	1540.42	3425.92	0.320	19.94	1663.14	1540.42	1560.27	98.00%	36.30
Parfib 56	72365.80	4014.32	8938.22	0.888	50.91	4297.16	4014.32	4073.41	99.80%	36.61
Queens 13	67.97	8.61	14.86	0.47	0.008	10.19	8.61	9.09	46.60%	15.04
Queens 14	440.15	50.92	87.37	0.041	1.83	62.01	50.92	54.06	24.10%	8.42
Queens 15	3163.03	354.66	628.09	0.293	17.04	406.22	354.66	373.96	52.70%	21.39
Queens 16	22008.80	2910.77	4855.65	1.508	94.76	3295.17	2910.77	3003.39	75.00%	26.38
Matmult 1.5k	1238.54	128.25	264.87	0.138	7.70	167.24	128.25	137.63	37.10%	12.75
Matmult 2k	3536.63	286.88	620.39	0.349	23.22	410.28	286.88	306.97	52.80%	17.50
Matmult 2.5k	5615.74	549.19	1197.87	0.714	37.26	702.64	549.19	599.69	37.10%	12.68
Matmult 3k	14183.10	932.49	2073.52	0.917	59.83	1189.54	932.49	986.48	64.00%	23.53
Minimax 6	289.66	30.00	58.68	0.019	1.10	35.87	30.18	31.23	65.00%	24.93
Minimax 8	3388.07	443.91	834.34	0.091	6.56	477.64	443.91	448.42	96.90%	34.28
Minimax 10	18606.70	2673.08	4901.15	0.722	44.56	2852.43	2673.08	2718.18	97.20%	28.66
Minimax 12	27269.70	4094.10	7511.70	1.113	77.26	4623.63	4094.10	4154.06	99.50%	30.92
DFT 2k	1248.80	65.29	122.81	0.083	6.56	121.34	67.13	68.40	91.70%	39.47
DFT 3k	2626.52	256.49	421.19	0.160	12.79	371.12	256.49	262.41	91.10%	39.61
DFT 4k	6731.86	810.06	1135.40	0.200	9.64	861.02	816.90	824.78	99.70%	34.11
DFT 6k	21885.10	2514.16	3512.12	0.948	31.96	2704.66	2566.94	2591.08	94.20%	37.45
PRSA 2M	505.41	58.04	108.80	0.051	2.84	74.77	58.04	61.47	50.70%	18.31
PRSA 6M	1524.60	190.81	324.17	0.030	1.89	200.35	190.81	192.61	100.00%	29.40
PRSA 8M	2039.98	254.53	428.22	0.027	1.77	265.64	254.53	256.03	100.00%	32.41
PRSA 9M	2325.16	274.08	485.49	0.111	6.17	298.55	274.08	281.63	75.50%	24.83
Quicksort 500k	706.92	29.16	264.93	0.090	3.21	59.75	33.86	36.37	0.00%	0.0
Quicksort 1M	1882.37	73.52	699.90	0.155	9.73	131.10	73.52	83.00	32.50%	10.34
Quicksort 3M	8195.00	289.17	2945.35	0.396	24.95	473.42	289.17	313.36	60.50%	19.62
Quicksort 6M	17104.40	655.49	7074.05	1.027	63.32	1070.19	655.49	719.79	41.30%	14.88

	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Ray 1300	660.36	47.99	197.61	0.040	3.04	72.58	47.99	49.75	74.20%	26.58
Ray 1400	785.25	52.50	228.70	0.085	4.73	80.61	52.50	58.29	32.90%	11.37
Ray 1500	891.90	63.41	262.71	0.060	3.71	95.65	63.41	67.18	37.60%	14.72
Ray 1600	1125.44	73.62	298.49	0.045	3.39	101.33	73.62	75.59	84.00%	26.78
Nbody 50k	1808.52	126.69	254.78	0.146	5.39	143.10	126.69	138.30	17.50%	6.34
Nbody 80k	4629.35	293.91	629.65	0.171	11.11	347.13	293.91	304.03	67.90%	23.09
Nbody 100k	6711.05	483.00	952.77	0.065	4.60	505.71	483.00	486.43	100.00%	41.54
Nbody 150k	15903.00	996.21	2084.97	0.069	5.38	1034.89	996.21	999.01	100.00%	34.57
Partree 600	23139.10	635.06	4137.45	3.635	286.57	3147.22	635.06	776.10	27.80%	18.59
Partree 650	33083.30	791.78	5039.03	0.716	47.88	1247.65	791.78	832.58	70.10%	21.59
Partree 700	57015.10	940.72	6244.34	0.600	47.88	1940.92	940.72	962.84	93.80%	30.23
Partree 800	92496.00	1218.31	8685.94	0.832	63.08	2368.14	1218.31	1255.20	88.00%	28.73
SumEuler 75k	16477.30	815.72	1888.40	0.131	10.57	887.23	815.72	820.27	99.90%	36.54
SumEuler 80k	17794.00	909.23	2171.43	0.132	8.75	1021.12	909.23	916.87	99.60%	37.05
SumEuler 85k	21015.70	1037.34	2442.17	0.129	9.44	1126.34	1037.34	1043.65	99.60%	37.99
SumEuler 90k	22457.00	1141.92	2766.56	0.148	11.80	1281.82	1141.92	1147.49	99.00%	36.55

Table 5.3: Results of sampling 30% of the search space using **Hill Climbing** over the raw benchmark dataset

Name	Raw Dataset			Tabu Search Dataset					
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*
Parfib 50	4164.59	236.05	502.39	2.447	184.60	1378.19	236.05	346.35	24.70%
Parfib 52	10882.10	595.55	1307.79	6.660	498.11	3679.15	595.55	902.80	10.20%
Parfib 54	27720.10	1540.42	3425.92	18.638	1373.84	9598.39	1540.42	2431.75	7.80%
Parfib 56	72365.80	4014.32	8938.22	44.092	3264.05	25433.00	4014.32	6119.04	10.10%
Queens 13	67.97	8.61	14.86	0.038	2.46	30.44	8.61	10.87	9.80%
Queens 14	440.15	50.92	87.37	0.251	16.89	170.75	50.92	65.04	10.40%
Queens 15	3163.03	354.66	628.09	1.531	103.87	1130.73	354.66	440.10	7.20%
Queens 16	22008.80	2910.77	4855.65	10.322	702.61	8702.32	2910.77	3484.25	15.70%
Matmult 1.5k	1238.54	128.25	264.87	0.936	61.91	421.95	128.25	182.45	4.80%
Matmult 2k	3536.63	286.88	620.39	2.713	182.72	1673.23	286.88	440.02	9.20%
Matmult 2.5k	5615.74	549.19	1197.87	6.370	458.65	3756.34	549.19	870.03	8.10%
Matmult 3k	14183.10	932.49	2073.52	10.788	762.89	6476.58	932.49	1495.04	6.30%
Minimax 6	289.66	30.00	58.68	0.127	8.64	82.83	30.18	37.10	22.40%
Minimax 8	3388.07	443.91	834.34	2.009	126.91	1249.67	443.91	566.55	23.50%
Minimax 10	18606.70	2673.08	4901.15	10.836	658.80	7208.76	2673.08	3360.35	21.60%
Minimax 12	27269.70	4094.10	7511.70	17.285	1173.74	11107.00	4094.10	5057.82	51.10%
DFT 2k	1248.80	65.29	122.81	0.413	30.01	319.24	66.99	85.72	4.30%
DFT 3k	2626.52	256.49	421.19	1.539	112.14	1024.84	257.82	332.01	2.10%
DFT 4k	6731.86	810.06	1135.40	2.454	166.65	2400.13	815.99	946.90	29.10%
DFT 6k	21885.10	2514.16	3512.12	9.613	753.49	8761.01	2562.81	2896.09	28.70%
PRSA 2M	505.41	58.04	108.80	0.263	18.12	203.49	58.04	72.42	18.60%
PRSA 6M	1524.60	190.81	324.17	0.834	65.85	633.11	190.81	222.93	58.20%
PRSA 8M	2039.98	254.53	428.22	1.019	78.76	819.60	254.53	297.17	56.70%
PRSA 9M	2325.16	274.08	485.49	1.319	100.06	1000.85	274.08	332.58	29.80%
Quicksort 500k	706.92	29.16	264.93	1.839	109.55	395.87	33.86	147.93	0.00%
Quicksort 1M	1882.37	73.52	699.90	5.594	316.77	1090.02	73.52	449.24	4.80%
Quicksort 3M	8195.00	289.17	2945.35	22.400	1284.34	4702.41	289.17	1780.29	5.80%
Quicksort 6M	17104.40	655.49	7074.05	52.509	3253.18	10672.00	655.49	3926.62	2.70%

	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*
Ray 1300	660.36	47.99	197.61	0.659	49.98	245.38	47.99	77.18	47.40%
Ray 1400	785.25	52.50	228.70	0.270	21.53	249.70	52.50	62.37	35.50%
Ray 1500	891.90	63.41	262.71	0.841	69.22	368.17	63.41	89.39	70.20%
Ray 1600	1125.44	73.62	298.49	1.009	75.47	267.82	73.62	120.23	61.70%
Nbody 50k	1808.52	126.69	254.78	1.095	77.72	637.26	126.69	183.37	0.70%
Nbody 80k	4629.35	293.91	629.65	2.735	198.74	1628.04	293.91	429.10	10.80%
Nbody 100k	6711.05	483.00	952.77	4.396	340.38	2524.85	483.00	665.73	32.90%
Nbody 150k	15903.00	996.21	2084.97	8.936	660.68	5711.90	996.21	1420.45	24.80%
Partree 600	23139.10	635.06	4137.45	26.689	1628.89	4980.37	635.06	2321.84	15.80%
Partree 650	33083.30	791.78	5039.03	24.517	1573.35	6041.69	791.78	2263.02	23.90%
Partree 700	57015.10	940.72	6244.34	32.432	2085.15	7413.56	940.72	2882.91	16.10%
Partree 800	92496.00	1218.31	8685.94	52.874	3175.62	10321.40	1218.31	4608.74	11.40%
SumEuler 75k	16477.30	815.72	1888.40	10.684	778.61	5579.21	815.72	1340.24	13.90%
SumEuler 80k	17794.00	909.23	2171.43	11.735	843.21	6434.43	909.23	1502.68	12.80%
SumEuler 85k	21015.70	1037.34	2442.17	12.568	896.27	7337.94	1037.34	1682.47	10.20%
SumEuler 90k	22457.00	1141.92	2766.56	15.988	1144.11	8198.63	1141.92	1957.04	9.50%

Table 5.4: Results of sampling 30% of the search space using **Tabu Search** over the raw benchmark dataset

5.4 Ant Colony

Swarm-based optimisation algorithms are in a different class than single-solution stochastic ones. They generate new solutions by incorporating a randomness factor, which can be a double-edged sword, wherein it can improve or worsen results. For example, different perturbation functions had different effects on Hill Climbing. Generally, in swarm-based optimisation, a population of a given size is used to simulate a particular process. The classical process consists of individual objects simulating pathfinding, whereas members are controlled using a delta value. As energy-finding and -generating solutions are not path-like problems with which we find the shortest path or optimal solution, we instead search for the solution that reduces energy the most by creating populations with specific program configurations that hold information about cores and clock frequencies.

When sampling ACO, we require a population of a given size, and each member uses the perturbation function to find their own solution. In this case, the search space is divided into two parts: population size and number of steps. To remain comparable with other algorithms, we use a factor pairing search-space percentage, i.e. 84 steps or a 30% search space in Hill Climbing equates to 21×4, 2×42 or 3×28 pairs. Factor pair selection means that if the simulation steps are low, we would not be able to improve the solution, and if the population is low, then no variety in the population exists. Additionally, these factors may have negative or positive effects.

We began with a population of 21 with 4 simulations, which immediately showed an interesting effect with regards to how accuracy progresses. After sampling with 28×3 and 21×4, we did not see any significant discrepancies in how the change in population or simulation steps affected the results. There were some slight gains and losses in accuracy, but the majority had remarkable similarities between versions. We compared the equivalent 30% search space of ACO to that of Hill Climbing. With ACO, we did not have an average convergence rate; instead, we had an optimised population upon algorithm completion.

In Table 5.5 on page 137, we notice the difference made by ACO. Most samples showed similar accuracy levels to other algorithms. However, the significant difference was the balance achieved by ACO. Beginning with Parfib, most results had accuracies compared to that of Hill Climbing. However, all ACO's accuracies were

above 90%. ACO really demonstrated its superior ability with Queens, as it seems to have balanced Hill Climbing's search space efficiency.

ACO also did well with Minimax, achieving accuracy levels close to that of Hill Climbing with six steps. It outperformed both with input sizes of 10 and 12. Accuracy levels similar to DFT were found, where most were closer to Hill Climbing. PRSA showed that ACO is advantageous for accuracy, but it had moderate gains in some problem sizes while maintaining high accuracy in others. Interestingly, Quicksort was challenging for ACO as an optimal solution could not be found for 500K inputs across all algorithms. Ray Tracing showed results similar to other cases, with some problem sizes, e.g. 1500 steps, having higher accuracy levels while other problem sizes presented an accuracy closer to Hill Climbing. ACO showed modest performance for N-body, apart from the 50K problem size, where ACO produced results within the 990-J range. Partree had moderate performance, where ACO showed balanced accuracy nearing Hill Climbing. With SumEuler, ACO produced results that were less accurate than those of Hill Climbing while maintaining higher levels than those of Tabu Search.

Name	Raw Dataset			Ant Colony Dataset					
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*
Parfib 50	4164.59	236.05	502.39	0.037	2.28	246.30	236.05	238.39	100.00%
Parfib 52	10882.10	595.55	1307.79	0.199	10.97	640.50	595.55	609.21	91.10%
Parfib 54	27720.10	1540.42	3425.92	0.521	29.76	1663.14	1540.42	1575.25	91.00%
Parfib 56	72365.80	4014.32	8938.22	1.342	72.60	4374.40	4014.32	4107.24	92.90%
Queens 13	67.97	8.61	14.86	0.005	0.34	9.96	8.61	8.93	58.90%
Queens 14	440.15	50.92	87.37	0.035	1.71	57.27	50.92	53.51	45.00%
Queens 15	3163.03	354.66	628.09	0.281	14.61	403.40	354.66	374.60	45.20%
Queens 16	22008.80	2910.77	4855.65	1.352	76.35	3183.93	2910.77	3001.76	69.20%
Matmult 1.5k	1238.54	128.25	264.87	0.109	6.46	152.63	128.25	135.36	45.80%
Matmult 2k	3536.63	286.88	620.39	0.206	14.47	355.00	286.88	297.70	76.80%
Matmult 2.5k	5615.74	549.19	1197.87	0.395	25.45	652.73	549.19	572.81	69.00%
Matmult 3k	14183.10	932.49	2073.52	0.721	45.52	1132.28	932.49	976.57	59.40%
Minimax 6	289.66	30.00	58.68	0.019	1.00	33.83	30.18	31.37	61.60%
Minimax 8	3388.07	443.91	834.34	0.084	6.01	476.75	443.91	448.20	98.40%
Minimax 10	18606.70	2673.08	4901.15	0.683	41.08	2871.58	2673.08	2716.84	98.00%
Minimax 12	27269.70	4094.10	7511.70	1.083	68.42	4440.69	4094.10	4160.21	99.30%
DFT 2k	1248.80	65.29	122.81	0.032	1.06	72.02	67.13	67.92	77.60%
DFT 3k	2626.52	256.49	421.19	0.112	6.13	283.89	256.49	264.15	78.00%
DFT 4k	6731.86	810.06	1135.40	0.184	7.61	872.82	816.90	824.32	99.90%
DFT 6k	21885.10	2514.16	3512.12	1.059	28.57	2685.95	2566.94	2602.66	89.40%
PRSA 2M	505.41	58.04	108.80	0.043	2.60	69.11	58.04	60.72	61.50%
PRSA 6M	1524.60	190.81	324.17	0.026	1.58	200.09	190.81	192.52	100.00%
PRSA 8M	2039.98	254.53	428.22	0.032	2.04	266.16	254.53	256.51	100.00%
PRSA 9M	2325.16	274.08	485.49	0.119	5.76	302.16	274.08	282.80	84.40%
Quicksort 500k	706.92	29.16	264.93	0.085	2.53	46.27	33.86	36.19	0.00%
Quicksort 1M	1882.37	73.52	699.90	0.138	7.91	119.18	73.52	82.65	27.20%
Quicksort 3M	8195.00	289.17	2945.35	0.358	20.27	403.67	289.17	313.17	45.50%
Quicksort 6M	17104.40	655.49	7074.05	0.963	49.01	928.14	655.49	724.46	26.10%

	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*
Ray 1300	660.36	47.99	197.61	0.033	2.21	63.23	47.99	49.89	61.70%
Ray 1400	785.25	52.50	228.70	0.068	3.69	70.07	52.50	57.25	25.80%
Ray 1500	891.90	63.41	262.71	0.059	3.75	80.20	63.41	66.96	46.50%
Ray 1600	1125.44	73.62	298.49	0.039	2.55	98.27	73.62	75.86	71.40%
Nbody 50k	0.152	1808.52	126.69	254.78	4.80	144.77	126.69	139.13	12.70%
Nbody 80k	4629.35	293.91	629.65	0.166	9.60	347.13	293.91	304.85	77.00%
Nbody 100k	6711.05	483.00	952.77	0.072	5.02	507.85	483.00	486.91	99.90%
Nbody 150k	15903.00	996.21	2084.97	0.083	6.28	1048.75	996.21	999.90	99.90%
Partree 600	23139.10	635.06	4137.45	0.468	26.15	831.29	635.06	666.78	47.20%
Partree 650	33083.30	791.78	5039.03	0.587	38.30	1030.27	791.78	826.31	61.60%
Partree 700	57015.10	940.72	6244.34	0.354	21.54	1163.40	940.72	963.15	91.00%
Partree 800	92496.00	1218.31	8685.94	0.786	50.04	1511.72	1218.31	1265.94	72.00%
SumEuler 75k	16477.30	815.72	1888.40	0.251	16.78	882.57	815.72	830.08	98.20%
SumEuler 80k	17794.00	909.23	2171.43	0.265	16.93	999.08	909.23	925.16	93.10%
SumEuler 85k	21015.70	1037.34	2442.17	0.246	14.77	1108.47	1037.34	1053.15	97.80%
SumEuler 90k	22457.00	1141.92	2766.56	0.334	23.01	1267.26	1141.92	1160.16	91.80%

Table 5.5: Results of sampling 30% (21 members and 4 steps) of the search space using **Ant Colony Optimisation** over the raw benchmark dataset

5.5 Genetic Algorithm

The Genetic algorithm presented in the previous chapter differs from ACO, which leverages a fixed population that cannot be dissolved or regenerated. We used a population of 10 solutions for the Genetic algorithm sample, each representing a solution of clock frequency and the number of cores for a given program input. To maintain a similar search space to the other algorithms presented in the other chapters, we set the number of generations (iterations) of the algorithm to 8, so the algorithm can explore a similar space size to 30%, i.e. 84 solutions in total. The population evolved using eight cycles to reach an optimised solution. The sampling is made using PyGAD [33], a generic library that provides an API for using different meta-heuristic algorithms. Although PyGAD is fundamentally the same as described in Chapter 4, it offers multiple configurations of how the genetic algorithm would behave, e.g. the API allows controlling the cross-over, the pairing of the parents and the mutation randomness. Several configurations might work better for some instances, but due to the scope of this thesis, it would not be possible to explore all combinations for such configurations. Therefore, we set the configuration to default values for all the sampling procedures.

The Genetic algorithm performed similarly to ACO and with a more balanced approach than single-solution stochastic algorithms. With Parfib, the Genetic algorithm achieved accuracies ranging from 80.10% up to 98.30% for an input size of 50. Regarding Parfib's 50 scores, the reason may be that input size 50 has 24 solutions in the space that are within the 5% margin of the absolute lowest energy consumption. In contrast, the other instances have between 8 and 9 solutions in the space within the 5% threshold. With Queens, it had moderate-to-low accuracies suggesting a similar issue to Parfib, where Queens had a small number of solutions ranging from 2 to 4 per input size within the accepted threshold. Matrix multiplication presented average performance, where the inputs 2K and 2.5K provided the best accuracy while 3k was not far behind 2.5k and 1.5k was the lowest.

In Minimax, the results observed were exceptionally better based on the number of solutions in the search space. Most problem sizes performed well, primarily as the solutions within the threshold increased from 4 to 12/16 for problem sizes 6 to 12, achieving results in the 80th-plus percentile. As for DFT, we notice varying accuracy levels across all problem sizes ranging from 61% up to 96.30%, where the

high accuracy levels in 6k and 4k may be attributed to the high number of solutions available within the threshold ranging from 91 to 92 solutions—followed by 84.10% accuracy for 2k despite having only 41 solutions available in the search space. Unlike DFT's 2k, the 3k input had difficulty scaling the accuracy levels similarly, even with the 31 solutions in the search space. With PRSA, we see that both 9mil and 2mil have low accuracy, which is reflected by the number of existing solutions lying in the search space where both 6 and 3 solutions, respectively, for the accuracy levels were different as the number of solutions in the search space exceeded 25 for 6mil and 8mil.

Quicksort showed accuracy levels like those of ACO with 500k again being at 0% since only one solution instance is within the threshold and the best solutions the algorithm found are only available within the 25% threshold. However, Hill Climbing had the best performance of all algorithms. Ray Tracing results resembled those of ACO, but they both had lower accuracy with specific inputs. With Nbody, it produced a wide range of accuracy levels from 19.70% up to 97.20% accuracy, which again may be attributed to the number of solutions available in each space, e.g. 80k having five solutions and 26 for 150k. Partree results were near those of ACO, displaying average accuracy, apart from a problem size of 700 inputs. SumEuler had above average and consistent performance, demonstrating accuracy levels of 80% and above; however, it remained lower than ACO.

Name	**Raw Dataset**			**Genetic Algorithm**
	Highest Energy	***Lowest Energy***	***Mean Energy***	***Accuracy***
Parfib 50	4164.59	236.05	502.39	98.30%
Parfib 52	10882.10	595.55	1307.79	81.90%
Parfib 54	27720.10	1540.42	3425.92	80.10%
Parfib 56	72365.80	4014.32	8938.22	86.10%
Queens 13	67.97	8.61	14.86	41.10%
Queens 14	440.15	50.92	87.37	29.30%
Queens 15	3163.03	354.66	628.09	35.70%
Queens 16	22008.80	2910.77	4855.65	51.60%
Matmult 1.5k	1238.54	128.25	264.87	31.40%
Matmult 2k	3536.63	286.88	620.39	58.60%
Matmult 2.5k	5615.74	549.19	1197.87	46.50%
Matmult 3k	14183.10	932.49	2073.52	44.60%
Minimax 6	289.66	30.00	58.68	52.70%
Minimax 8	3388.07	443.91	834.34	86.70%
Minimax 10	18606.70	2673.08	4901.15	92.20%
Minimax 12	27269.70	4094.10	7511.70	97.40%
DFT 2k	1248.80	65.29	122.81	84.10%
DFT 3k	2626.52	256.49	421.19	61.00%

DFT 4k	6731.86	810.06	1135.40	96.30%
DFT 6k	21885.10	2514.16	3512.12	94.90%
PRSA 2M	505.41	58.04	108.80	47.00%
PRSA 6M	1524.60	190.81	324.17	99.50%
PRSA 8M	2039.98	254.53	428.22	98.40%
PRSA 9M	2325.16	274.08	485.49	77.40%
Quicksort 500k	706.92	29.16	264.93	0.00%
Quicksort 1M	1882.37	73.52	699.90	29.10%

	Highest Energy	*Lowest Energy*	*Mean Energy*	*Accuracy*
Quicksort 3M	8195.00	289.17	2945.35	43.90%
Quicksort 6M	17104.40	655.49	7074.05	22.30%
Ray 1300	660.36	47.99	197.61	47.20%
Ray 1400	785.25	52.50	228.70	27.60%
Ray 1500	891.90	63.41	262.71	39.20%
Ray 1600	1125.44	73.62	298.49	61.20%
Nbody 50k	1808.52	126.69	254.78	19.70%
Nbody 80k	4629.35	293.91	629.65	58.60%
Nbody 100k	6711.05	483.00	952.77	92.40%
Nbody 150k	15903.00	996.21	2084.97	97.20%
Partree 600	23139.10	635.06	4137.45	48.10%
Partree 650	33083.30	791.78	5039.03	61.70%
Partree 700	57015.10	940.72	6244.34	84.20%
Partree 800	92496.00	1218.31	8685.94	71.20%
SumEuler 75k	16477.30	815.72	1888.40	95.60%
SumEuler 80k	17794.00	909.23	2171.43	85.60%
SumEuler 85k	21015.70	1037.34	2442.17	88.00%
SumEuler 90k	22457.00	1141.92	2766.56	85.40%

Table 5.6: Results of sampling ≈30% (10 members and 8 generations) of the search space using **Genetic Algorithm** over the raw benchmark dataset

5.6 Summary

When evaluating meta-heuristic algorithms, we observed varying performances based on the search space size and the algorithm used. Hill Climbing demonstrated good performance with Parfib, which had exceptional scalability. Hill Climbing's limitations were obvious when examining Queens and MatMult problems, wherein the general performance varied from one configuration to another. Each set of cores affected either the parallelism performance or the GHC runtime overall effect on the execution. Hill Climbing's accuracy formed a continuous pattern as the search space increased, until the final 30%, showing that other factors may affect accuracy. The standard deviation did not appear to affect this result. Notably, a result with the ±5% energy margin does not exist; therefore, the algorithm is hindered until it finds the available optimal state. With Tabu Search, lower accuracy existed across the search space, which failed to achieve reasonable accuracy levels compared to the other algorithms. However, its performance resembled that of Hill Climbing with scalable benchmarks, producing similar accuracy levels apart from some examples,

e.g. Partree. ACO displayed a new behaviour, providing more robust performance with regard to accuracy and demonstrating good optimisation in cases where Hill Climbing had difficulty. The genetic algorithm provided the final and most balanced results, which showed improved accuracy for some benchmarks at better levels than all previous algorithms. It also had moderately balanced accuracy levels, whereas some algorithms did well, e.g. Hill Climbing and Nbody. One non-trivial observation from all the sampling results is that some optimisations of specific benchmarks, e.g. Minimax and Quicksort. Several benchmarks showed improved accuracy levels as the workload increased, that might have resulted in more extended execution periods with which to complete computations.

Chapter 6

Meta-Heuristic – PARSEC Results

This chapter presents the analysis and details of simulating PARSEC benchmarks sampling a dataset collected using the same method used for the Haskell benchmarks in Chapter 3. Sections 6.2, 6.3 and 6.4 respectively discuss the Hill Climbing, ACO and Genetic algorithm results for the PARSEC's examples with equivalent implementations.

6.1 Sampling Structure

The PARSEC [8] benchmarking suite has parallel benchmarks with multiple equivalent implementations using different parallelism libraries, e.g. OpenMP and TBB. The suite offers parallelism workloads that assist with evaluating various multiprocessing and parallel problems. Its equivalent implementations of benchmarking programs highlight hardware performance changes with different workloads. As in the case of Haskell's results, the Tabu Search results are provided for comparative reasons only.

6.2 Hill Climbing

The present Hill Climbing sample follows the same method used with Haskell. The benchmarks were sampled for 1000 executions, and the final results represent the numbers over all those samples. The RMSE represents the error value between the selected solution and the absolute lowest, and the standard deviation shows the

spread of values across all sample. Values of high, low and mean represent values from both the raw dataset and the collected sample. The discussion refers to the data in Tables 6.1, 6.2 and 6.3.

Blackscholes: The sample from the three libraries collected from running the Hill Climbing algorithm on the raw CSV data showed similarities in performance, with slight advantages for Pthreads and TBB over OpenMP. TBB outperformed the three samples with accuracy results of 49.80%. Pthreads and OpenMP had accuracies of 38.40 and 23.10%, respectively. The standard deviation was similar for Pthreads and TBB, but it dropped slightly with OpenMP. In the 10% search space, Pthreads and TBB were similar in accuracy, outperforming OpenMP by a considerable margin. At a 30% search space, all three libraries did well, TBB at 98.50%, Pthreads at 91.20% and OpenMP at 83.00% accuracies. Moreover, the accuracy of OpenMP doubled, which may be related to the lack of sufficient steps for finding optimal solutions.

Bodytrack: Bodytrack performance was similar to that of Blackscholes, where OpenMP lagged behind other libraries. The other two libraries had high accuracy results. At a 5% search space, Pthreads and TBB had accuracies above the 90th percentile. OpenMP lagged behind at 47.70%. The highest and lowest energy consumption values between libraries were similar, with the only difference being the mean of OpenMP being above average. The 10% search space resulted in a 100% accuracy for Pthreads, 99.80% for TBB and 82.10% for OpenMP, whose standard deviation was again considerably higher those of the other libraries. A 30% search space resulted in Pthreads and TBB accuracies of 100%; OpenMP performed at 99.40%.

Ferret: with Ferret, we observed a similar contrast in performance for Pthreads and TBB. The 5% search space resulted in TBB having a 99.30% accuracy. Pthreads had 77%. At a 10% search space, TBB achieved 100% accuracy and Pthreads achieved 95.30%. The 30% search space allowed both libraries to find optimal solutions within +5% of the actual lowest energy consumption.

Swaptions: In Swaptions, we observed opposite results than those of Pthreads and TBB. At a 5% search space, Pthreads achieved 61.40% accuracy, whereas TBB achieved 51.90%. Notably, TBB's standard deviation was higher than that of Pthreads by several points. The 10% search space provided Pthreads with an accuracy of

84.10% and TBB at 80.10%. A similar increase in the overall standard deviation was seen with TBB. Finally, at a 30% search space, both libraries had high accuracy, but TBB slightly lower. TBB's increase in standard deviation still existed, but the larger search space narrowed the gap between standard deviations for both libraries.

Name	Raw Dataset			Hill Climbing Dataset						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Blackscholes – OpenMP	5253.32	742.42	1133.42	0.966	32.98	999.10	742.42	801.74	23.10%	1.30
Blackscholes – Pthreads	5287.08	739.52	1121.83	0.903	37.21	1014.97	739.52	790.90	38.40%	2.16
Blackscholes – TBB	5319.99	758.00	1152.05	0.862	37.60	1079.01	758.00	805.51	49.80%	2.80
Bodytrack – OpenMP	48829.46	4196.83	7119.01	4.605	210.95	5778.98	4196.83	4442.31	47.70%	2.945
Bodytrack – Pthreads	44933.52	4428.96	6800.69	1.228	74.71	5676.90	4428.96	4472.18	98.70%	5.45
Bodytrack – TBB	49043.34	4008.37	6591.74	1.349	79.68	5342.92	4008.37	4059.79	93.50%	5.84
Ferret – Pthreads	1708.75	269.59	352.54	0.171	5.22	349.84	269.59	280.38	77.00%	4.24
Ferret – TBB	1614.36	270.82	344.59	0.080	3.90	347.60	270.82	274.91	99.30%	5.62
Swaptions – Pthreads	3539.72	338.57	570.78	0.408	22.11	558.10	338.57	356.83	61.40%	3.51
Swaptions – TBB	3580.35	334.75	568.40	0.504	26.24	570.28	334.75	358.59	51.90%	2.81

Table 6.1: Results of sampling 5% of the search space using **Hill Climbing** over the raw benchmark dataset

Name	Raw Dataset			Hill Climbing Dataset						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Blackscholes – OpenMP	5253.32	742.42	1133.42	0.644	19.92	852.54	742.42	783.09	43.80%	5.04
Blackscholes – Pthreads	5287.08	739.52	1121.83	0.567	23.77	882.80	739.52	771.52	61.60%	7.63
Blackscholes – TBB	5319.99	758.00	1152.05	0.471	20.83	878.41	758.00	783.72	78.80%	10.07
Bodytrack – OpenMP	48829.46	4196.83	7119.01	2.225	114.84	4842.40	4196.83	4302.97	82.10%	10.99
Bodytrack – Pthreads	44933.52	4428.96	6800.69	0.382	14.38	4528.61	4428.96	4451.64	100.00%	12.00
Bodytrack – TBB	49043.34	4008.37	6591.74	0.408	21.94	4218.83	4008.37	4026.83	99.80%	13.56
Ferret – Pthreads	1708.75	269.59	352.54	0.126	3.91	288.89	269.59	277.50	95.30%	11.75
Ferret – TBB	1614.36	270.82	344.59	0.043	1.99	278.90	270.82	273.07	100.00%	12.68
Swaptions – Pthreads	3539.72	338.57	570.78	0.170	7.90	388.25	338.57	347.55	84.20%	10.54
Swaptions – TBB	3580.35	334.75	568.40	0.216	9.53	423.47	334.75	346.61	80.10%	9.30

Table 6.2: Results from sampling 10% of the search space using **Hill Climbing** over raw benchmark dataset

Name	Raw Dataset			Hill Climbing Dataset						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Converg. Rate*
Blackscholes – OpenMP	5253.32	742.42	1133.42	0.408	14.96	805.11	742.42	766.85	83.00%	26.79
Blackscholes – Pthreads	5287.08	739.52	1121.83	0.300	16.15	852.62	739.52	753.13	91.20%	31.69
Blackscholes – TBB	5319.99	758.00	1152.05	0.217	11.65	806.93	758.00	767.82	98.50%	35.23
Bodytrack – OpenMP	48829.46	4196.83	7119.01	0.667	42.00	4460.70	4196.83	4217.69	99.40%	36.03
Bodytrack – Pthreads	44933.52	4428.96	6800.69	0.233	9.21	4463.82	4428.96	4442.50	100.00%	30.42
Bodytrack – TBB	49043.34	4008.37	6591.74	0.124	7.44	4039.29	4008.37	4012.92	100.00%	37.75
Ferret – Pthreads	1708.75	269.59	352.54	0.089	3.91	281.11	269.59	274.51	100.00%	36.89
Ferret – TBB	1614.36	270.82	344.59	0.023	1.37	276.96	270.82	271.72	100.00%	36.06
Swaptions – Pthreads	3539.72	338.57	570.78	0.072	3.54	364.92	338.57	342.23	99.30%	34.07
Swaptions – TBB	3580.35	334.75	568.40	0.114	5.75	368.95	334.75	340.37	98.50%	31.97

Table 6.3: Results of sampling 30% of the search space using **Hill Climbing** over the raw benchmark dataset

Name	Raw Dataset			Tabu Search Dataset					
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*
Blackscholes – OpenMP	5253.32	742.42	1133.42	4.936	308.58	3107.32	742.42	1047.10	1.20%
Blackscholes – Pthreads	5287.08	739.52	1121.83	4.817	302.71	3093.37	739.52	1035.35	2.60%
Blackscholes – TBB	5319.99	758.00	1152.05	5.706	371.51	3130.15	758.00	1094.69	3.80%
Bodytrack – OpenMP	48829.46	4196.83	7119.01	46.565	3433.66	27112.42	4196.83	6550.57	5.60%
Bodytrack – Pthreads	44933.52	4428.96	6800.69	41.503	3285.29	26204.40	4428.96	6111.36	35.70%
Bodytrack – TBB	49043.34	4008.37	6591.74	46.664	3579.41	25864.04	4008.37	6068.53	18.80%
Ferret – Pthreads	1708.75	269.59	352.54	1.237	90.23	959.56	269.59	330.15	10.30%
Ferret – TBB	1614.36	270.82	344.59	1.048	78.80	933.72	270.82	318.50	37.80%
Swaptions – Pthreads	3539.72	338.57	570.78	3.351	222.04	1921.47	338.57	531.93	6.50%
Swaptions – TBB	3580.35	334.75	568.40	3.214	212.95	1879.46	334.75	520.18	3.80%

Table 6.4: Results of sampling 30% of the search space using **Tabu Search** over the raw benchmark dataset

6.3 Ant Colony

ACO sampling was performed using the same approach as Haskell's, where the sampling made at a 30% search space used a combination of population size and the number of steps for each individual in the population. The population was set to 21, and the number of simulations was set to four per member. The results shown in Table 6.5 reveal that ACO's accuracy performed similar to Hill Climbing. The overall results were highly accurate, apart from OpenMP in Blackscholes, where the performance was 68.80% accuracy. Initially, at a 5% search space, the results appeared similar to the previously reviewed algorithms; ACO had 70% accuracy, which is closer to an upper result in Hill Climbing. The remaining results differed little from the algorithms seen before; Ferret achieved maximum accuracy. The other benchmarks produced results very close to the other algorithms.

Name	Raw Dataset			Ant Colony Dataset					
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*
Blackscholes – OpenMP	5253.32	742.42	1133.42	0.450	17.93	829.61	742.42	768.46	70.00%
Blackscholes – Pthreads	5287.08	739.52	1121.83	0.312	15.62	807.52	739.52	754.87	92.70%
Blackscholes – TBB	5319.99	758.00	1152.05	0.262	13.15	813.65	758.00	770.84	97.40%
Bodytrack – OpenMP	48829.46	4196.83	7119.01	1.242	67.44	4473.79	4196.83	4252.28	98.30%
Bodytrack – Pthreads	44933.52	4428.96	6800.69	0.232	10.08	4475.11	4428.96	4441.76	100.00%
Bodytrack – TBB	49043.34	4008.37	6591.74	0.232	11.32	4073.08	4008.37	4020.08	100.00%
Ferret – Pthreads	1708.75	269.59	352.54	0.093	3.58	282.70	269.59	275.07	100.00%
Ferret – TBB	1614.36	270.82	344.59	0.024	1.25	276.25	270.82	271.92	100.00%
Swaptions – Pthreads	3539.72	338.57	570.78	0.080	3.52	360.25	338.57	342.93	99.50%
Swaptions – TBB	3580.35	334.75	568.40	0.122	5.73	358.78	334.75	341.12	97.00%

Table 6.5: Results of sampling 30% (21 members, 4 steps) of the search space using **Ant Colony Optimisation** over the raw benchmark dataset

6.4 Genetic Algorithm

The Genetic algorithm (**PyGAD**) yielded results like those of ACO and Hill Climbing. In Table 6.6, we see that the genetic algorithm struck a balance between all algorithms reviewed earlier. The first, the OpenMP version of Blackscholes, achieved an accuracy of 62.20%, not far behind ACO's 70%, following results for Pthreads and TBB resembled those of OpenMP, where the differences were the same in Pthreads and fractional in TBB. The Bodytrack benchmark produced results in the 90th percentile, with only TBB outperforming at 98.20%. Ferret and Swaptions performed decently. Nevertheless, the performance was not on par with ACO, for example. Ferret had perfect accuracy results achieved in ACO. In , the results lacked for Pthreads achieving only 93.60% and 99% for TBB. Swaptions achieved an accuracy lower than those of ACO and Hill Climbing, managing only to get 90th percentile results for both Pthreads and TBB.

Name	**Raw Dataset**			**Genetic Algorithm**
	Highest Energy	***Lowest Energy***	***Mean Energy***	***Accuracy***
Blackscholes – OpenMP	5253.32	742.42	1133.42	62.20%
Blackscholes – Pthreads	5287.08	739.52	1121.83	81.30%
Blackscholes – TBB	5319.99	758.00	1152.05	91.40%
Bodytrack – OpenMP	48829.46	4196.83	7119.01	91.90%
Bodytrack – Pthreads	44933.52	4428.96	6800.69	98.20%
Bodytrack – TBB	49043.34	4008.37	6591.74	92.80%
Ferret – Pthreads	1708.75	269.59	352.54	93.60%
Ferret – TBB	1614.36	270.82	344.59	99.90%
Swaptions – Pthreads	3539.72	338.57	570.78	90.10%
Swaptions – TBB	3580.35	334.75	568.40	90.20%

Table 6.6: Results of sampling ≈30% (10 solutions per population, 8 generations) of the search space using **PyGAD** genetic algorithm over the raw benchmark dataset

6.5 Tabu Search

For the PARSEC dataset, the results are more or less similar to those of Haskell's dataset Tabu search. The overall results do not appear to be improving nor presenting any reliable accuracy to assess. When comparing the Tabu and Hill Climbing results, we see that Tabu is significantly lower compared to Hill Climbing even though they

are both sampling 30% of the search space.

Blackscholes different implementations (*TBB*, *OpenMP*, *Pthreads*) have all presented results in the 80th to 91st percentile, but in Tabu, we notice that the results have stagnated at around the 2-3 percentile. Following the results from Bodytrack, Hill Climbing reached a perfect score in *TBB* and *Pthreads* implementations and a near-perfect result for *OpenMP*, nonetheless Tabu could not come anywhere near those numbers resulting in *OpenMP* scoring 3% while *TBB* and *Pthreads* having 16.50% and 35.40% respectively. Similarly, Ferret and Swaptions have presented higher accuracy levels compared to their Tabu counterpart.

6.6 Summary

PARSEC presented new ways to evaluate algorithm efficiency and applicability for energy optimisation programs. Considering Hill Climbing's performance, all benchmarks using multiple libraries inferred optimal solutions early, especially at the lowest search space level of 5%, specifically with Ferret and Bodytrack benchmarks. At the highest search space level, the results were mostly in the 90th percentile, indicating that different libraries and implementations were not affected the outcome. This is the case for Ferret and Bodytrack, where the results were very similar. The average convergence rate was relatively low owing to additional Hill Climbing iterations needed to improve the search space. Tabu Search delivered subpar results compared to Hill Climbing. The final search-space level (30%) showed similar performance for limited cases owing to the random nature of the perturbation function. Furthermore, both algorithms inferred optimal solutions at a near 16% search space, meaning that a large portion of the 30% search space was not required. ACO showed similar performance as most of PARSEC's benchmarks were scalable. This led to ACO showing similar accuracy levels to that of Hill Climbing, but with a lower accuracy than Blackscholes. The energy optimisation of the Genetic algorithm was better than that of ACO and close to Hill Climbing.

Chapter 7

Meta-Heuristic – Model Predictions

In this chapter we provide an overview of the results from applying the NNLS model on the prediction datasets of Haskell and C/C++. Section 7.3 provides a discussion on the results of applying meta-heuristics on the Haskell dataset. Section 7.4 provides a discussion on the results of applying meta-heuristics on the Haskell dataset.

7.1 Tabu Search

Since Tabu Search failed to achieve any practical results in the past meta-heuristic results, it would not be unexpected to observe a similar case with the statistical models used in this chapter.

As we see from all the Tables 7.1, 7.5, 7.9 and 7.13, the results struggled to deliver any results that can be considered accurate. With a few exceptions like Binarytrees in Table 7.13, the rest of the results provided low to none accuracy levels that would be practical for finding energy values.

7.2 Sampling Structure

The previous assessment where meta-heuristics were applied to the modelling datasets for both languages, C/C++ and Haskell, provided insight into using the different optimisation algorithms in various cases. However, to prove usability and provide

realistic use-cases, it would be essential to assess the energy search space using the statistical models introduced in Chapter 3.

Since the modelling datasets for both C/C++ and Haskell were used to construct the predictive models, we would not be able to apply the meta-heuristic prediction using those models on the same dataset. Therefore, we can use the statistical prediction on the prediction dataset, which contains multiple implementations for Haskell and C/C++ with various parallel C/C++ implementations: TBB, Pthreads, and OpenMP.

Also, to provide a comprehensive approach, the prediction sample will be sampled for the same frequencies: 1.2, 1.4, 1.8, 2.1, and 2.6 GHz to have a complete energy profile for all the benchmarks. Additionally, the different instruction counts (apart from certain anomalies and the number of cycles) remained consistent when assessing Intel SDE and Perf with different frequencies. For example, when we compare Intel SDE and Perf counts for 1.4 GHz against 2.6 GHz (except for a few anomalies such as Fasta) using the NNLS features in Chapter 3, we observe an average decrease of 0.41% to 0.097% in instruction counts for all benchmarks for SDE–suggesting that changing the frequency factor did not affect the overall emulation. Moreover, Perf also demonstrated a similar case to the Intel SDE sample, where the overall count of the metrics collected had an average change ranging from 0.6% to 1.84%. Perf change ratio suggests that the 1.4 GHz frequency increased the count of *syscalls* and other features; however, it did not change drastically.

Similar to the meta-heuristics results of C/C++ and Haskell, the Tabu Search results in the following sections will be used for comparison purposes to assess the algorithms sampled. In the sampling of this section, we use a smaller search space consisting of only 28 cores and five frequencies, the purpose of this configuration is to reduce the time taken to estimate points using the R models. In addition to the previously mentioned configuration, the sampling of the benchmarks will use NNLS as it had the lowest average of MAPE, and from observing the graph, the NNLS appeared to follow the actual sample pattern. Since we are assessing the capabilities of a predictive model, we will not need to cap the size of the search space or evaluate the search at different intervals. The search space will be capped at ≈60% for all algorithms using the full space of 140 points/solutions.

7.3 Haskell

The sections below will discuss the results for the Haskell prediction dataset. The results represent a sample of 1000 execution using each algorithm from Chapter 4. The sampling used NNLS estimates to optimise the parallel configuration for the benchmarks, meaning each state's energy values are derived from the NNLS model and used to drive the optimisation process. In all the instances, the accuracy referred is in the 95% range or how frequent the solutions found are within 5% of the lowest energy consumption. For the result discussion below, we will be referring to the Tables 7.1 to 7.4 pages 157 to 160. The *lowest instances* column provides the number of all the energy solutions from the raw dataset, which are within 5% of the energy consumption of the actual lowest.

Fasta The Fasta sample showed early signs of a continuous pattern for the rest of the dataset. The different input sizes had 0% accuracy meaning that none of the algorithms could optimise the state within 5% of the lowest energy value. The exception for Fasta is the 900k input achieving some accuracy levels, which could be attributed to the number of instances in the 5% range available. Also, when considering the Hill Climbing results, the 900k achieved 83%, suggesting that Hill Climbing could reach better accuracy by using NNLS estimated energy only. The Genetic algorithm managed to achieve only 20.90% accuracy for input size 900k but failed to improve any other inputs, this case can be seen in ACO and Hill Climbing. It is unclear what makes this accuracy discrepancy happens for a single input.

Binarytrees Binarytrees is another case of low accuracy levels. Since all Binarytrees inputs have one or two solutions within the 5% of the lowest energy consumption, finding a solution may be complex whether to use a predictive model or search by collecting actual samples. All algorithms find it challenging to find a single solution with appropriate energy consumption except for some low accuracy levels achieved by **PyGAD** where Binarytrees showed some accuracy between 8.70% and 13.00%.

Spectral-norm In the case of Spectral-norm, we observe another case of low accuracy even though more solutions are available than Binarytrees, for example. In all the sampled algorithms except for the Genetic algorithm sample, we observe low accuracy that does not change with more solutions available in the search space. The

genetic algorithm sample shows that the more solutions available with the different inputs, the more accurate the sample becomes.

Prime Decomposition Prime Decomposition demonstrated another case of low accuracy levels. The few solutions available in the search space proved to be challenging for all four algorithms. The inconsistency in finding solutions for Prime Decomposition may be because model estimates cannot drive the optimisation correctly.

Name	Raw Dataset			Tabu Search Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	37.93	13.59	22.13	0.108	4.84	35.47	13.59	21.70	2.50%	2.14% or 3
Fasta – 1 mil	41.04	13.91	24.31	0.134	5.89	39.54	13.91	24.05	1.00%	0.71% or 1
Fasta – 1.1 mil	46.71	15.54	27.14	0.144	6.34	46.67	15.54	26.50	0.80%	0.71% or 1
Fasta – 1.2 mil	50.26	16.98	29.48	0.160	7.12	47.31	16.98	29.05	1.10%	0.71% or 1
Binarytrees – 20	303.21	79.93	130.69	0.645	27.97	209.05	79.93	129.18	0.90%	0.71% or 1
Binarytrees – 21	474.86	158.44	261.79	1.363	61.56	426.50	158.44	261.15	1.70%	1.43% or 2
Binarytrees – 22	1122.13	393.14	656.68	3.337	146.33	1122.13	393.14	647.15	1.10%	0.71% or 1
Binarytrees – 23	2185.78	846.08	1320.17	6.202	281.75	2185.78	846.08	1312.37	1.70%	1.43% or 2
Spectral-norm – 8.5k	1548.33	104.56	293.92	3.564	270.45	1548.23	104.56	262.50	1.90%	2.14% or 3
Spectral-norm – 9.5k	1918.83	130.30	372.96	4.699	359.64	1918.83	130.30	333.10	5.20%	4.29% or 6
Spectral-norm – 10.5k	2370.19	159.84	463.99	5.418	406.44	2370.19	159.84	407.68	7.00%	6.43% or 9
Spectral-norm – 11.5k	2859.88	190.82	594.21	7.198	533.70	2859.88	190.82	530.15	4.00%	4.29% or 6
Prime Decomposition	284.12	95.05	181.92	1.113	39.74	284.12	105.74	184.42	0.00%	1.43% or 2

Table 7.1: Results from sampling 30% of the search space using **Tabu Search** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Hill Climbing Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	37.93	13.59	22.13	93.945	2.47	24.38	14.16	15.24	83.00%	2.14% or 3
Fasta – 1 mil	41.04	13.91	24.31	194.566	0.77	27.62	16.86	20.01	0.00%	0.71% or 1
Fasta – 1.1 mil	46.71	15.54	27.14	157.640	2.68	36.40	18.81	19.74	0.00%	0.71% or 1
Fasta – 1.2 mil	50.26	16.98	29.48	212.763	1.33	37.78	19.45	23.58	0.00%	0.71% or 1
Binarytrees – 20	303.21	79.93	130.69	941.409	10.50	144.38	79.93	107.79	1.40%	0.71% or 1

Binarytrees – 21	474.86	158.44	261.79	2481.170	9.31	295.23	158.44	236.35	0.20%	1.43% or 2
Binarytrees – 22	1122.13	393.14	656.68	6159.848	35.51	1018.17	415.83	584.67	0.00%	0.71% or 1
Binarytrees – 23	2185.78	846.08	1320.17	14632.686	65.54	1812.58	867.98	1304.14	0.60%	1.43% or 2
Spectral-norm – 8.5k	1548.33	104.56	293.92	1026.805	3.99	159.34	111.96	136.78	0.00%	2.14% or 3
Spectral-norm – 9.5k	1918.83	130.30	372.96	1186.069	2.89	209.70	147.42	167.69	0.00%	4.29% or 6
Spectral-norm – 10.5k	2370.19	159.84	463.99	1332.629	6.63	252.23	166.42	201.45	0.30%	6.43% or 9
Spectral-norm – 11.5k	2859.88	190.82	594.21	1738.535	7.91	272.43	196.79	245.22	0.40%	4.29% or 6
Prime Decomposition	284.12	95.05	181.92	1546.629	23.09	185.13	105.74	138.17	0.00%	1.43% or 2

Table 7.2: Results from sampling 30% of the search space using **Hill Climbing** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Ant-Colony Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	37.93	13.59	22.13	175.702	3.46	25.10	14.16	17.94	44.50%	2.14% or 3
Fasta – 1 mil	41.04	13.91	24.31	200.042	1.36	37.21	18.64	20.09	0.00%	0.71% or 1
Fasta – 1.1 mil	46.71	15.54	27.14	287.439	5.67	38.27	18.81	22.64	0.00%	0.71% or 1
Fasta – 1.2 mil	50.26	16.98	29.48	246.174	3.44	37.78	20.11	23.97	0.00%	0.71% or 1
Binarytrees – 20	303.21	79.93	130.69	1100.076	11.52	152.21	100.42	112.76	0.00%	0.71% or 1
Binarytrees – 21	474.86	158.44	261.79	2628.807	13.64	301.54	197.81	240.44	0.00%	1.43% or 2
Binarytrees – 22	1122.13	393.14	656.68	6688.392	35.79	875.21	518.16	601.59	0.00%	0.71% or 1
Binarytrees – 23	2185.78	846.08	1320.17	15132.441	62.51	1641.82	1194.72	1320.51	0.00%	1.43% or 2
Spectral-norm – 8.5k	1548.33	104.56	293.92	1110.293	5.16	159.34	136.02	139.29	0.00%	2.14% or 3
Spectral-norm – 9.5k	1918.83	130.30	372.96	1282.393	5.34	212.06	166.91	170.50	0.00%	4.29% or 6
Spectral-norm – 10.5k	2370.19	159.84	463.99	1493.561	11.89	277.15	198.03	205.55	0.00%	6.43% or 9
Spectral-norm – 11.5k	2859.88	190.82	594.21	1880.525	9.18	291.11	241.68	249.57	0.00%	4.29% or 6
Prime Decomposition	284.12	95.05	181.92	2029.480	16.22	231.48	105.74	157.14	0.00%	1.43% or 2

Table 7.3: Results from sampling 30% (21 members, 4 steps) of the search space using **ACO** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Genetic Algorithm Results	
	Highest Energy	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	37.93	13.59	22.13	20.90%	2.14% or 3
Fasta – 1 mil	41.04	13.91	24.31	0.00%	0.71% or 1
Fasta – 1.1 mil	46.71	15.54	27.14	0.00%	0.71% or 1
Fasta – 1.2 mil	50.26	16.98	29.48	0.00%	0.71% or 1
Binarytrees – 20	303.21	79.93	130.69	10.00%	0.71% or 1

Binarytrees – 21	474.86	158.44	261.79	13.00%	1.43% or 2
Binarytrees – 22	1122.13	393.14	656.68	4.20%	0.71% or 1
Binarytrees – 23	2185.78	846.08	1320.17	8.70%	1.43% or 2
Spectral-norm – 8.5k	1548.33	104.56	293.92	14.80%	2.14% or 3
Spectral-norm – 9.5k	1918.83	130.30	372.96	18.20%	4.29% or 6
Spectral-norm – 10.5k	2370.19	159.84	463.99	11.70%	6.43% or 9
Spectral-norm – 11.5k	2859.88	190.82	594.21	7.90%	4.29% or 6
Prime Decomposition	284.12	95.05	181.92	25.30%	1.43% or 2

Table 7.4: Results of sampling ≈30% (10 members, 8 generations) of the search space using **PyGAD** over the raw benchmark dataset using NNLS to derive the energy estimates

7.4 C/C++ (Pthreads, TBB, OpenMP)

The C/C++ samples were made using the equivalent benchmarks to those of Haskell. All parallel libraries have been sampled separately using each algorithm for a thousand samples to ensure more consistent results. The discussion in the following sections will evaluate the separate algorithms based on Tables 7.5 to 7.16 pages 163 to 171.

Hill Climbing The Hill Climbing performance varied between the different benchmarks and their inputs. Beginning with Fasta, the algorithm achieves varying accuracy levels ranging from 0% up to the 90^{th} percentile levels, as shown in the OpenMP case. It is difficult to attribute the low energy ranges to the low accuracy as Hill Climbing proved to achieve acceptable accuracy for the OpenMP sample. Binarytrees is another case of Hill Climbing's algorithm's inability to achieve accuracy within the 5% of the lowest energy range, even with the number of optimal instances growing with different inputs. Similar to Binarytrees, Spectral-norm proved challenging for Hill Climbing to optimise energy levels. All samples except for Pthreads demonstrated low to non-existent even though more solutions were available in the search space. The case of 8.5k and 11.5k from the Pthreads sample seem to achieve accuracy; however, it is unknown what may be contributing to the sudden increase in the accuracy levels. Although Prime Decomposition in some samples had 18 solutions in the search space, the Hill-Climbing algorithm struggled to achieve any accuracy in all the different benchmark implementations.

Ant-Colony Similar to Hill Climbing, ACO underperformed in all three parallel samples. The accuracy demonstrated by the algorithm appears to struggle to optimise particular benchmarks such as Binarytrees, Spectral-norm, and Prime Decomposition, which all had accuracy levels at zero or near zero except for a few cases where Spectral-norm's 11.5k achieved 12.90% for example. The previously mentioned benchmarks had more solutions than Fasta did not contribute to or help improve the accuracy levels. On the other hand, Fasta demonstrated somewhat below average accuracy for Pthreads and OpenMP; however, it failed to achieve any accuracy for the TBB sample. Even though Fasta had a low number of solutions available in the search space, the ACO algorithm optimised for the lowest energy solution at least in 2 samples.

Genetic Algorithm The first notable characteristic of this algorithm is the difficulty in finding any accurate solutions, which may be due to the need for more solutions in the space or even the model's inability to drive the accuracy levels correctly to match the actual energy measurements from the sample. Alternatively, both reasons can be present in some cases, contributing to a nearly impossible optimisation. In the case of the Genetic algorithm optimisation, the accuracy levels appear stagnant when trying to find solutions in the search space, even in the cases where the NNLS model has improved accuracy in estimating energy values. Even when the chosen statistical model might not have high error margins, it could not achieve accuracies above 10% across all parallel implementations.

Name	Raw Dataset			Tabu Search Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	4.66	1.60	2.52	0.012	0.50	3.97	1.60	2.49	2.10%	2.14% or 3
Fasta – 1 mil	4.29	1.75	2.65	0.011	0.44	4.29	1.75	2.61	3.00%	3.57% or 5
Fasta – 1.1 mil	4.21	1.77	2.86	0.013	0.47	4.11	1.77	2.82	1.40%	1.43% or 2
Fasta – 1.2 mil	5.91	1.99	3.09	0.013	0.42	3.98	1.99	3.07	1.50%	1.43% or 2
Binarytrees – 20	107.08	23.39	31.32	0.096	6.34	58.73	23.39	29.00	8.80%	7.86% or 11
Binarytrees – 21	210.42	44.19	60.82	0.199	12.29	117.86	44.19	56.61	2.80%	2.86% or 4
Binarytrees – 22	517.12	104.84	142.94	0.535	38.54	329.71	104.84	131.83	23.10%	20.71% or 29
Binarytrees – 23	999.99	203.85	283.81	1.130	79.46	635.29	203.85	263.42	8.50%	9.29% or 13
Spectral-norm – 8.5k	525.18	41.50	89.51	0.644	45.65	287.56	41.50	74.92	6.40%	5.71% or 8
Spectral-norm – 9.5k	711.42	51.16	113.81	0.791	55.36	359.32	51.16	93.20	5.90%	6.43% or 9
Spectral-norm – 10.5k	802.69	61.37	135.98	1.074	75.19	437.67	61.37	118.41	6.20%	6.43% or 9
Spectral-norm – 11.5k	1044.34	75.16	164.97	1.145	80.36	526.12	75.16	135.70	7.50%	8.57% or 12
Prime Decomposition	11.00	1.38	2.22	0.012	0.90	6.12	1.38	2.00	12.80%	12.86% or 18

Table 7.5: Results from sampling 30% of the search space using **Tabu Search** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Hill Climbing Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	4.66	1.60	2.52	13.184	0.27	3.55	1.60	1.92	45.30%	2.14% or 3
Fasta – 1 mil	4.29	1.75	2.65	5.221	0.11	2.42	1.75	1.87	43.90%	3.57% or 5
Fasta – 1.1 mil	4.21	1.77	2.86	6.231	0.08	2.28	1.81	1.94	3.20%	1.43% or 2
Fasta – 1.2 mil	5.91	1.99	3.09	5.794	0.12	2.45	1.99	2.13	54.40%	1.43% or 2
Binarytrees – 20	107.08	23.39	31.32	229.582	1.18	33.57	23.57	30.56	0.50%	7.86% or 11

Binarytrees – 21	210.42	44.19	60.82	542.236	1.81	62.21	47.16	61.24	0.00%	2.86% or 4
Binarytrees – 22	517.12	104.84	142.94	836.683	2.64	143.91	106.42	131.16	0.10%	20.71% or 29
Binarytrees – 23	999.99	203.85	283.81	1767.993	5.04	280.15	215.86	259.53	0.00%	9.29% or 13
Spectral-norm – 8.5k	525.18	41.50	89.51	435.618	1.35	59.85	43.16	55.21	0.40%	5.71% or 8
Spectral-norm – 9.5k	711.42	51.16	113.81	541.020	1.16	71.92	53.52	68.23	0.40%	6.43% or 9
Spectral-norm – 10.5k	802.69	61.37	135.98	692.546	1.95	85.71	65.66	83.19	0.00%	6.43% or 9
Spectral-norm – 11.5k	1044.34	75.16	164.97	703.053	2.61	102.42	77.55	97.24	1.40%	8.57% or 12
Prime Decomposition	11.00	1.38	2.22	14.906	0.08	1.96	1.56	1.85	0.00%	12.86% or 18

Table 7.6: Results from sampling 30% of the search space using **Hill Climbing** over raw benchmark dataset using NNLS to derive the energy estimates

Name	**Raw Dataset**			**Ant-Colony Results**						
	Highest Energy	***Lowest Energy***	***Mean Energy***	***RMSE***	***Standard Dev.***	***Highest Energy***	***Lowest Energy***	***Mean Energy***	***Accuracy***	***Low Instances***
Fasta – 900k	4.66	1.60	2.52	13.387	0.19	2.17	1.60	1.98	11.00%	2.14% or 3
Fasta – 1 mil	4.29	1.75	2.65	8.374	0.15	2.42	1.75	1.97	24.60%	3.57% or 5
Fasta – 1.1 mil	4.21	1.77	2.86	8.913	0.12	2.28	1.81	2.02	1.80%	1.43% or 2
Fasta – 1.2 mil	5.91	1.99	3.09	6.973	0.14	2.66	1.99	2.16	37.90%	1.43% or 2
Binarytrees – 20	107.08	23.39	31.32	227.485	0.93	33.57	29.70	30.53	0.00%	7.86% or 11
Binarytrees – 21	210.42	44.19	60.82	538.756	0.90	62.21	57.54	61.20	0.00%	2.86% or 4
Binarytrees – 22	517.12	104.84	142.94	881.011	3.15	144.34	129.99	132.52	0.00%	20.71% or 29
Binarytrees – 23	999.99	203.85	283.81	1876.491	6.59	280.15	257.82	262.83	0.00%	9.29% or 13
Spectral-norm – 8.5k	525.18	41.50	89.51	449.181	2.22	60.55	45.92	55.53	0.00%	5.71% or 8
Spectral-norm – 9.5k	711.42	51.16	113.81	537.921	1.87	71.92	53.52	68.07	0.70%	6.43% or 9
Spectral-norm – 10.5k	802.69	61.37	135.98	676.667	2.24	91.43	61.37	82.65	0.10%	6.43% or 9
Spectral-norm – 11.5k	1044.34	75.16	164.97	712.685	1.81	104.71	77.55	97.63	0.20%	8.57% or 12
Prime Decomposition	11.00	1.38	2.22	13.884	0.07	1.96	1.56	1.82	0.00%	12.86% or 18

Table 7.7: Results from sampling 30% (21 members, 4 steps) of the search space using **ACO** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Genetic Algorithm Results	
	Highest Energy	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	4.66	1.60	2.52	0.00%	2.14% or 3
Fasta – 1 mil	4.29	1.75	2.65	0.00%	3.57% or 5
Fasta – 1.1 mil	4.21	1.77	2.86	0.00%	1.43% or 2
Fasta – 1.2 mil	5.91	1.99	3.09	0.00%	1.43% or 2
Binarytrees – 20	107.08	23.39	31.32	0.00%	7.86% or 11
Binarytrees – 21	210.42	44.19	60.82	0.00%	2.86% or 4
Binarytrees – 22	517.12	104.84	142.94	0.00%	20.71% or 29
Binarytrees – 23	999.99	203.85	283.81	0.00%	9.29% or 13
Spectral-norm – 8.5k	525.18	41.50	89.51	58.80%	5.71% or 8
Spectral-norm – 9.5k	711.42	51.16	113.81	62.60%	6.43% or 9
Spectral-norm – 10.5k	802.69	61.37	135.98	43.50%	6.43% or 9
Spectral-norm – 11.5k	1044.34	75.16	164.97	49.50%	8.57% or 12
Prime Decomposition	11.00	1.38	2.22	0.00%	12.86% or 18

Table 7.8: Results of sampling ≈30% (10 members, 8 generations) of the search space using **PyGAD** over the raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Tabu Search Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	33.59	16.17	23.50	0.091	3.18	33.59	16.17	23.51	1.20%	1.43% or 2
Fasta – 1 mil	36.21	19.28	26.57	0.091	3.37	36.21	19.28	26.57	3.90%	2.86% or 4
Fasta – 1.1 mil	39.80	21.41	29.75	0.103	3.78	39.80	21.41	29.66	1.70%	1.43% or 2
Fasta – 1.2 mil	44.09	26.55	32.97	0.082	3.45	44.09	26.58	32.90	4.70%	5.71% or 8
Binarytrees – 20	5219.90	1139.89	1660.21	6.414	362.13	3351.88	1139.89	1571.69	1.50%	2.14% or 3
Binarytrees – 21	11213.10	2410.19	3510.48	14.359	861.05	7211.95	2410.19	3332.23	3.10%	3.57% or 5
Binarytrees – 22	25634.00	5047.91	7855.66	34.522	1929.21	16262.70	5047.91	7388.15	0.70%	1.43% or 2
Binarytrees – 23	56655.40	10955.10	16278.30	56.013	4091.99	35310.90	10955.10	15011.14	0.50%	0.71% or 1
Spectral-norm – 8.5k	63.33	32.66	50.22	0.216	6.32	63.33	37.09	50.59	0.00%	0.71% or 1
Spectral-norm – 9.5k	71.82	36.30	50.21	0.179	6.37	71.82	36.88	50.70	2.10%	3.57% or 5
Spectral-norm – 10.5k	69.49	32.61	50.49	0.220	6.50	69.49	36.81	50.78	0.00%	1.43% or 2
Spectral-norm – 11.5k	69.53	33.20	50.30	0.213	6.63	69.53	36.93	50.70	0.00%	0.71% or 1
Prime Decomposition	3.96	1.94	2.67	0.010	0.50	3.96	2.07	2.68	0.00%	1.43% or 2

Table 7.9: Results from sampling 30% of the search space using **Tabu Search** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Hill Climbing Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	33.59	16.17	23.50	194.765	0.74	31.53	16.17	22.28	0.10%	1.43% or 2
Fasta – 1 mil	36.21	19.28	26.57	179.464	2.03	32.51	19.28	24.58	0.60%	2.86% or 4
Fasta – 1.1 mil	39.80	21.41	29.75	271.821	2.11	36.12	21.41	29.74	0.80%	1.43% or 2
Fasta – 1.2 mil	44.09	26.55	32.97	387.429	3.44	42.11	27.97	38.31	0.00%	5.71% or 8
Binarytrees – 20	5219.90	1139.89	1660.21	24133.414	110.70	1939.20	1238.78	1894.99	0.00%	2.14% or 3

Binarytrees – 21	11213.10	2410.19	3510.48	46774.510	211.33	4045.87	2410.19	3874.17	0.90%	3.57% or 5
Binarytrees – 22	25634.00	5047.91	7855.66	121490.122	657.22	9115.47	5047.91	8833.19	1.10%	1.43% or 2
Binarytrees – 23	56655.40	10955.10	16278.30	226381.492	1230.10	18720.40	10955.10	18007.54	0.60%	0.71% or 1
Spectral-norm – 8.5k	63.33	32.66	50.22	907.362	2.25	63.11	44.82	61.26	0.00%	0.71% or 1
Spectral-norm – 9.5k	71.82	36.30	50.21	627.517	1.66	59.46	46.20	56.08	0.00%	3.57% or 5
Spectral-norm – 10.5k	69.49	32.61	50.49	826.932	2.23	62.54	46.43	58.67	0.00%	1.43% or 2
Spectral-norm – 11.5k	69.53	33.20	50.30	882.282	1.40	66.48	47.34	61.07	0.00%	0.71% or 1
Prime Decomposition	3.96	1.94	2.67	48.228	0.30	3.96	2.21	3.44	0.00%	1.43% or 2

Table 7.10: Results from sampling 30% of the search space using **Hill Climbing** over raw benchmark dataset using NNLS to derive the energy estimates

Name	**Raw Dataset**			**Ant-Colony Results**						
	Highest Energy	***Lowest Energy***	***Mean Energy***	***RMSE***	***Standard Dev.***	***Highest Energy***	***Lowest Energy***	***Mean Energy***	***Accuracy***	***Low Instances***
Fasta – 900k	33.59	16.17	23.50	208.744	1.10	31.53	19.44	22.68	0.00%	1.43% or 2
Fasta – 1 mil	36.21	19.28	26.57	226.029	3.14	32.51	22.57	25.70	0.00%	2.86% or 4
Fasta – 1.1 mil	39.80	21.41	29.75	285.827	1.81	38.08	27.42	30.26	0.00%	1.43% or 2
Fasta – 1.2 mil	44.09	26.55	32.97	386.238	3.25	41.64	34.27	38.32	0.00%	5.71% or 8
Binarytrees – 20	5219.90	1139.89	1660.21	22827.664	159.97	1939.20	1417.48	1843.83	0.00%	2.14% or 3
Binarytrees – 21	11213.10	2410.19	3510.48	45877.621	258.26	4206.08	3108.77	3837.82	0.00%	3.57% or 5
Binarytrees – 22	25634.00	5047.91	7855.66	117666.611	923.49	9275.25	5886.56	8652.55	0.00%	1.43% or 2
Binarytrees – 23	56655.40	10955.10	16278.30	203881.949	2175.51	19429.90	11921.20	17024.68	0.00%	0.71% or 1
Spectral-norm – 8.5k	63.33	32.66	50.22	877.662	1.71	63.11	55.83	60.36	0.00%	0.71% or 1
Spectral-norm – 9.5k	71.82	36.30	50.21	634.796	1.90	59.78	47.87	56.29	0.00%	3.57% or 5
Spectral-norm – 10.5k	69.49	32.61	50.49	845.947	2.92	62.54	52.32	59.20	0.00%	1.43% or 2
Spectral-norm – 11.5k	69.53	33.20	50.30	874.860	1.96	66.48	53.50	60.80	0.00%	0.71% or 1
Prime Decomposition	3.96	1.94	2.67	49.814	0.31	3.96	2.17	3.49	0.00%	1.43% or 2

Table 7.11: Results from sampling 30% (21 members, 4 steps) of the search space using **ACO** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Genetic Algorithm Results	
	Highest Energy	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	33.59	16.17	23.50	4.80%	1.43% or 2
Fasta – 1 mil	36.21	19.28	26.57	7.50%	2.86% or 4
Fasta – 1.1 mil	39.80	21.41	29.75	15.00%	1.43% or 2
Fasta – 1.2 mil	44.09	26.55	32.97	8.70%	5.71% or 8
Binarytrees – 20	5219.90	1139.89	1660.21	1.60%	2.14% or 3
Binarytrees – 21	11213.10	2410.19	3510.48	0.80%	3.57% or 5
Binarytrees – 22	25634.00	5047.91	7855.66	7.80%	1.43% or 2
Binarytrees – 23	56655.40	10955.10	16278.30	7.00%	0.71% or 1
Spectral-norm – 8.5k	63.33	32.66	50.22	0.00%	0.71% or 1
Spectral-norm – 9.5k	71.82	36.30	50.21	0.00%	3.57% or 5
Spectral-norm – 10.5k	69.49	32.61	50.49	0.00%	1.43% or 2
Spectral-norm – 11.5k	69.53	33.20	50.30	0.00%	0.71% or 1
Prime Decomposition	3.96	1.94	2.67	0.00%	1.43% or 2

Table 7.12: Results of sampling ≈30% (10 members, 8 generations) of the search space using **PyGAD** over the raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Tabu Search Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	8.43	1.16	2.48	0.020	1.13	8.43	1.16	2.53	0.80%	1.43% or 2
Fasta – 1 mil	11.24	1.23	2.60	0.020	1.09	11.24	1.23	2.57	2.10%	2.14% or 3
Fasta – 1.1 mil	7.16	1.37	2.84	0.021	1.13	7.16	1.37	2.83	1.80%	2.14% or 3
Fasta – 1.2 mil	5.96	1.40	2.94	0.021	1.04	5.96	1.40	2.89	1.10%	0.71% or 1
Binarytrees – 20	684.35	475.18	543.36	0.975	51.39	684.35	475.18	543.68	13.50%	12.86% or 18
Binarytrees – 21	1607.33	1138.27	1298.15	2.438	129.06	1607.33	1138.27	1309.24	6.60%	7.14% or 10
Binarytrees – 22	3844.68	2677.29	3012.87	5.238	309.01	3844.68	2677.29	3018.32	19.10%	21.43% or 30
Binarytrees – 23	7988.28	5726.73	6404.47	10.432	626.02	7988.28	5726.73	6396.12	27.30%	27.86% or 39
Spectral-norm – 8.5k	46.84	2.96	7.60	0.064	4.31	25.52	2.96	6.52	1.40%	1.43% or 2
Spectral-norm – 9.5k	46.74	2.94	7.58	0.063	4.28	25.31	2.94	6.42	0.80%	0.71% or 1
Spectral-norm – 10.5k	46.62	3.04	7.63	0.061	4.15	25.78	3.04	6.40	2.80%	2.86% or 4
Spectral-norm – 11.5k	46.67	3.02	7.71	0.060	4.06	24.08	3.02	6.42	2.90%	2.86% or 4
Prime Decomposition	10.51	1.49	4.49	0.043	2.27	10.51	1.49	4.56	7.70%	10.71% or 15

Table 7.13: Results from sampling 30% of the search space using **Tabu Search** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Hill Climbing Results						
	Highest Energy	*Lowest Energy*	*Mean Energy*	*RMSE*	*Standard Dev.*	*Highest Energy*	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	8.43	1.16	2.48	4.514	0.09	1.72	1.21	1.27	38.90%	1.43% or 2
Fasta – 1 mil	11.24	1.23	2.60	2.269	0.06	1.60	1.23	1.26	91.70%	2.14% or 3
Fasta – 1.1 mil	7.16	1.37	2.84	4.469	0.11	2.24	1.37	1.46	55.40%	2.14% or 3
Fasta – 1.2 mil	5.96	1.40	2.94	21.205	0.18	2.53	1.50	2.05	0.00%	0.71% or 1
Binarytrees – 20	684.35	475.18	543.36	4024.903	18.01	654.44	475.18	601.18	1.50%	12.86% or 18

Binarytrees – 21	1607.33	1138.27	1298.15	12555.916	60.40	1586.20	1182.49	1530.71	0.20%	7.14% or 10
Binarytrees – 22	3844.68	2677.29	3012.87	21157.148	121.35	3694.57	2703.54	3335.25	0.60%	21.43% or 30
Binarytrees – 23	7988.28	5726.73	6404.47	58779.203	144.17	7872.31	6207.58	7579.90	0.00%	27.86% or 39
Spectral-norm – 8.5k	46.84	2.96	7.60	24.234	0.36	4.40	2.96	3.64	13.40%	1.43% or 2
Spectral-norm – 9.5k	46.74	2.94	7.58	36.365	0.39	4.33	3.29	4.02	0.00%	0.71% or 1
Spectral-norm – 10.5k	46.62	3.04	7.63	28.910	0.41	4.53	3.11	3.86	0.90%	2.86% or 4
Spectral-norm – 11.5k	46.67	3.02	7.71	26.799	0.33	4.37	3.11	3.80	14.60%	2.86% or 4
Prime Decomposition	10.51	1.49	4.49	59.330	0.14	3.41	1.58	3.37	0.00%	10.71% or 15

Table 7.14: Results from sampling 30% of the search space using **Hill Climbing** over raw benchmark dataset using NNLS to derive the energy estimates

Name	**Raw Dataset**			**Ant-Colony Results**						
	Highest Energy	***Lowest Energy***	***Mean Energy***	***RMSE***	***Standard Dev.***	***Highest Energy***	***Lowest Energy***	***Mean Energy***	***Accuracy***	***Low Instances***
Fasta – 900k	8.43	1.16	2.48	14.935	0.31	2.15	1.16	1.52	2.10%	1.43% or 2
Fasta – 1 mil	11.24	1.23	2.60	9.272	0.24	1.95	1.23	1.40	54.10%	2.14% or 3
Fasta – 1.1 mil	7.16	1.37	2.84	11.268	0.28	2.13	1.37	1.60	42.30%	2.14% or 3
Fasta – 1.2 mil	5.96	1.40	2.94	18.218	0.31	2.21	1.50	1.89	0.00%	0.71% or 1
Binarytrees – 20	684.35	475.18	543.36	4193.280	10.29	644.65	597.41	607.38	0.00%	12.86% or 18
Binarytrees – 21	1607.33	1138.27	1298.15	12603.233	49.58	1586.20	1456.85	1533.73	0.00%	7.14% or 10
Binarytrees – 22	3844.68	2677.29	3012.87	24371.089	150.51	3694.57	3284.56	3433.15	0.00%	21.43% or 30
Binarytrees – 23	7988.28	5726.73	6404.47	58002.522	133.45	7728.87	7266.55	7556.07	0.00%	27.86% or 39
Spectral-norm – 8.5k	46.84	2.96	7.60	25.204	0.34	4.43	3.07	3.68	6.60%	1.43% or 2
Spectral-norm – 9.5k	46.74	2.94	7.58	37.369	0.31	4.38	2.94	4.08	3.10%	0.71% or 1
Spectral-norm – 10.5k	46.62	3.04	7.63	34.474	0.33	4.53	3.11	4.08	0.90%	2.86% or 4
Spectral-norm – 11.5k	46.67	3.02	7.71	25.436	0.30	4.37	3.11	3.76	12.90%	2.86% or 4
Prime Decomposition	10.51	1.49	4.49	59.506	0.06	3.41	1.61	3.38	0.00%	10.71% or 15

Table 7.15: Results from sampling 30% (21 members, 4 steps) of the search space using **ACO** over raw benchmark dataset using NNLS to derive the energy estimates

Name	Raw Dataset			Genetic Algorithm Results	
	Highest Energy	*Lowest Energy*	*Mean Energy*	*Accuracy*	*Low Instances*
Fasta – 900k	8.43	1.16	2.48	0.00%	1.43% or 2
Fasta – 1 mil	11.24	1.23	2.60	0.00%	2.14% or 3
Fasta – 1.1 mil	7.16	1.37	2.84	0.00%	2.14% or 3
Fasta – 1.2 mil	5.96	1.40	2.94	0.00%	0.71% or 1
Binarytrees – 20	684.35	475.18	543.36	1.70%	12.86% or 18
Binarytrees – 21	1607.33	1138.27	1298.15	3.20%	7.14% or 10
Binarytrees – 22	3844.68	2677.29	3012.87	2.90%	21.43% or 30
Binarytrees – 23	7988.28	5726.73	6404.47	1.90%	27.86% or 39
Spectral-norm – 8.5k	46.84	2.96	7.60	0.00%	1.43% or 2
Spectral-norm – 9.5k	46.74	2.94	7.58	0.00%	0.71% or 1
Spectral-norm – 10.5k	46.62	3.04	7.63	0.00%	2.86% or 4
Spectral-norm – 11.5k	46.67	3.02	7.71	0.00%	2.86% or 4
Prime Decomposition	10.51	1.49	4.49	0.00%	10.71% or 15

Table 7.16: Results of sampling ≈30% (10 members, 8 generations) of the search space using **PyGAD** over the raw benchmark dataset using NNLS to derive the energy estimates

7.5 Summary

Although the use of a statistical model in combination with meta-heuristics provided limited accuracy, the application of which in the Genetic algorithm showed that there are improvements possible in this area. The increase in available solutions in the search space shows that the Genetic algorithm was able to improve accuracy in certain cases even with tight energy ranges.

Chapter 8

Related Work

This chapter provides a review of the literature related to the topic. Section 8.1 focuses on efforts with embedded devices and chip design. Section 8.2 discusses low-level modelling of energy using compilers and compiler optimisations. Section 2.2.3 reviews structured parallelism.

Despite the increasing importance of energy usage in multicore and low-power settings, the energy analysis of software is relatively understudied, particularly with parallel software applications at the language level. Work to date has mostly focused on simple program-level measurements of energy using ISA analysis with CPU frequency sampling. Other techniques rely on compiler phases and instruction-level representations, often without considering high-level language features, e.g. looping, data dependencies and common micro-architectural features, e.g. pipelines.

8.1 Embedded Systems, Portable Devices and Chip Design

In [6], the hardware-directed analysis of energy/power costs and different methods of scaling and reducing energy was performed. For scheduling and core-task allocation, the authors used a pre-selected voltage and clock frequency. By providing a multi-objective *NSGA-II* [26] algorithm with two goals, i.e. energy consumption and execution time minimisation, they derived several optimal states that prioritise energy

and execution time. Their approach did not eliminate the use of DVFS, a system that manages power and clock frequency, but it supported the use of such features. The authors applied their approach to an XMOS board with a MIPS-based processor running at 400 MHz, obtaining an average 76% optimal allocation/scheduling at loose bounds of execution time and energy; they achieved 70% with stricter bounds.

At the design level, the work of Marowka [69] proposed a processing-based solution using an analytical model of each processing technique. The author reviewed data processing from a chip organisation perspective, that is, how the different processing elements can be organised to handle different computations in parallel. The categories discussed were symmetric, asymmetric and simultaneous asymmetric, and each design was evaluated for scalability and performance per watt. In symmetric processing, a general-purpose multicore processor based on *Gustafson–Barsis'* Law was used for scaled speedup, where performance per watt was moderate, compared with other chip designs. The asymmetric configuration consisted of a single-dye CPU–GPU. The overall performance was much-improved over symmetric processing, considering joule consumption. The final configuration was simultaneous asymmetric, wherein the same asymmetric design was utilised. However, GPU and CPU processing overlapped; the performance per joule for this configuration was not compared with the author's asymmetric configuration.

Although low-power devices, such as mobile phones and microcontrollers, have inherently low energy consumption by design, they still benefit greatly from lower energy consumption. The refactoring approach suggested in [23] analysed Android open-source codebases and identified several energy-intensive Android code-bases. The code was refactored to a more energy-efficient mode using a static analysis tool, i.e. *Leafactor*. Out of the 140 apps analysed, 45 improved energy consumption by up to 40%. Another noteworthy study [89] built a dataset of Android application energy consumption values using an experimental approach to collecting measurements from open-source projects. The applications tested were installed on an Android device with specific settings fixed at the lowest values, e.g. screen brightness, to ensure that no application energy readings interfered with measurements.

An OpenMP solution used to evaluate and minimise fork–join-based task parallelism was reviewed in [84]. The authors described how an energy model based on frequency scaling can be derived for this purpose. The method is based on OpenMP

scheduling, and optimisations mostly focused on performance without considering energy. The authors derived a model that describes the dynamic power consumed during switching in addition to leakage or base consumption. The dynamic power consumption is based on switching probability, load capacitance, the square of voltage ($\beta \times$ frequency) and a frequency result of the scale factor, s^{-1}, multiplied by the frequency. The authors assumed that static power consumption was a constant in the final equation. The scheduling algorithm handling execution considered energy minimisation and performance based on different scheduling cases, e.g. having sufficient processors and evaluating methods and optimal scalability for single tasks. The evaluation of the energy-aware scheduling algorithm corresponds to the energy model integrated into the algorithm.

8.2 Compilers, Scheduling and Systems

When considering software energy analyses of programming language code bases, approaching the problem from a compiler view might be viable. In [88], the authors presented a compiler-based solution for estimating energy consumed by code blocks, given that a CPU energy model is available. The solution used a performance estimate technique based on a C compiler with a *worst-case execution time-aware* (WCET-aware) extension, i.e. *WCC*, used to estimate and minimise the execution time cost of the compiled codebase. The *WCC* parses an extensible markup language (XML) file containing particular low-level instructions with energy costs based on predefined cases to handle energy estimations, given a code structure. The compiler then correlates the weights in the parsed XML tree containing the branching weights for energy.

Kerrison [51, 52] estimated energy usage based on the ISA of an *XMOS XS1-L xCORE* embedded processor with one core and eight threads. Energy consumption for imperative languages was estimated by considering instruction-level costs. The *XS1-L* processor, having a MIPS-based architecture, has features that differ from x86 and other high-performance processors. For example, hardware-managed context switching and thread management take place at the hardware level, and the processor switches threads at every cycle, allowing up to eight to execute at once. The model is based on the *XS1-L* instruction set, where the main components are derived from base, thread, instruction and inter-instruction power costs and execution statistics. The total

execution time, thread active time as a ratio of total time and number of operands used in each instruction sampled are also considered. The model equation factors all previous components and presents the total energy consumed by the program. The authors presented a solution and tested it using a set of known benchmarks, e.g. *matrix multiplication*, *SHA-2* hashing function and *audio mixing*.

A more advanced approach of energy analysis can be found in [78], where *worst-case energy consumption* (WCEC) was estimated by considering the operand bit values or the data-dependent instruction set. The authors used two candidate processors: 32-bit *XS1-L* and 8-bit *AVR*. The first runs on an XMOS board, and the second runs on an *Atmel* board. Both are embedded processors. Predicting worst-case energy costs for a given program structure involves careful attention to the data operated on by the instruction. The work discussed the fact that circuit switching individual bits in a processor's registers involves calculating a set of operands that trigger power draws like real data used in normal program execution. For example, the authors stated that the number of bits needed to explore a $20\times$ 20 matrix in an 8-bit AVR processor resulted in 3200 bits to understand energy consumed. Extending this to full program operations quickly becomes impractical. The authors proposed using three techniques to generate and collect energy consumption data that would be utilised to model energy consumption. The first method randomly generated data for operands, and the second method generated operand data using genetic algorithms. The third method manually generates operand data using a system to generate a distribution of possible energy values for an instruction sequence. The model uses a specific structure of sequenced instruction costs, but the cost is not represented by the base cost of the instruction being executed. Rather, it is represented by the cost of a sequence of instructions commonly used together, such as a `mov`, followed by an `add` or a `sub` instruction. Using the described model, the authors propose the use of an *implicit path enumeration technique* (IPET) [15] to calculate the individual cost of a given code block using the described model. They achieved predictive results; however, using a small subset of instructions did not simulate a real use case. Pallister [77] further expanded the work to highlight the compiler effects when optimising for performance vs. energy.

Georgiou *et al.* applied energy consumption static analysis (*ECSA*) to map the intermediate representations (*IR*s) of LLVM to a low-level assembly language [35]. Doing so allowed the analysis of software energy consumption on an embedded

system with a cache-less ARM processor, achieving accurate energy usage predictions. However, this method was not generic; it was tightly coupled to the LLVM compiler infrastructure and cannot link execution costs to high-level program constructs. For example, although the GHC Haskell compiler can generate code from the *Cmm* intermediate language via LLVM, this is not the default code generation option. Moreover, it is unsuitable for most Haskell applications or for general use. Moreover, the work in question was tied to sequential execution models and did not consider high-level parallelism or threading.

At the macro level, Marathe *et al.* [64] measured energy consumption on Intel CPU architectures. They studied CPU variations and *uncore functionality*[1] energy consumption over a set of benchmarks from NASA's *NAS Parallel Benchmarks* [5]. Their analysis revealed varying performance and energy consumption in newer architectures. They attributed their findings to several factors, such as the quality of chip production. However, they did not attempt to relate this to source-level characteristics.

Chowdury *et al.* [19] measured the overall energy consumption of several industrial software systems and constructed a system-level model. Their approach used *system calls* (syscalls) to model the energy usage for a large set of examples. However, the approach failed with applications that did not make regular system calls and those involving significant amounts of computation or memory access between calls.

Pereira *et al.* [81] evaluated execution time, DRAM energy and package energy usage for 28 programming languages using the *Computer Language Benchmarks Game* [22]. They concluded that overall execution time was necessarily directly correlated to energy consumption but that average power usage could introduce significant variations. Although languages, such as *C, Pascal, Fortran, Go and C++* demonstrated better overall energy efficiency, modern languages, such as *Haskell, OCaml, Racket and Python* are more consistent in terms of overall energy consumption. Georgiou *et al.* [36] similarly evaluated the energy consumption of 13 programming languages using different execution methods, i.e. compiled, interpreted and virtual-machine, considering several small sequential tasks in semantically equivalent programs from *Rosetta Code's* [86] repository. The authors suggested that compiled programs showed the most energy savings, among all others. However, their work did not consider

[1]Intel uses the term 'uncore' or 'system agent' to refer to functionality outside a microprocessor core. However, it is closely connected to the core to ensure good performance.

parallelism, nor did it attempt to relate energy usage to source programs.

Using a more theoretical approach, van Eekelen *et al.* presented an abstract high-level static analysis on energy-aware components (ECAlogic) with states [93], using a Hoare-like rule-based system that used energy-aware state components. They described a dependent-type system that uses finite-state machine models of a given single-processor system [34]. However, unlike our work, their work did not provide concrete energy predictions, nor did they consider parallel execution.

Very little of this work considers high-level programming languages or the relationship between source-level constructs and energy usage, especially for parallel computations. In a Haskell setting, Lima *et al.* [63] investigated energy consumption patterns over several data structures taken from the *Edison* library of purely functional data structures. They presented results showing how to minimise energy consumption when switching between alternative mutable primitives, i.e. *TMVar* and *MVar*, and reported non-trivial improvements to the energy consumption of particular data structures when specific operations are used. Melfe *et al.* [70] extended this work to consider common operations, such as *seeking*, *adding or removing* elements from data structures in the Edison library, deducing that overall energy consumption is also reduced by GHC's standard performance optimisations.

[96] reported additional details about energy consumption in processing components. Considering the increasing number of machine learning applications and the types of computations required in the learning process, increasing energy consumption is required. The authors provided an overview of the training of a natural language processor model using a NUMA system with two GPUs. The energy details were aggregated and applied to an equation that outputs the total power. The authors then demonstrated, using a comparative example, the amount of carbon emissions generated.

Kessler [53, 54] extended *crown* scheduling to manage task scheduling and task fusion as a single entity on a parallel system that is DVFS enabled. Kessler used *integer linear programming* (ILP) to resolve mapping cores and tasks while deciding on the appropriate DVFS configuration for each task. The authors tested the technique on an ARM big.LITTLE processor 2700× using a set of synthetic benchmarks. The results showed that using *task fusion* improves energy consumption when tuned for

specific goals.

Further work based on exploring performance and hardware in relation to varying GPU program implementations, the work of Li [61] proposed using an adaptive pruning algorithm in combination with a heuristic-based algorithm referred to as 'heuristic convexity assumption' to select particular samples for training. The authors show that it is possible to achieve up to 100% prediction accuracy using a small search space. Additionally, Li [62] demonstrated the use of *MeterPU*, a generic portable measurement framework for multicore CPU and multi-GPU systems, that has allowed the collection of time and energy related measurements which enabled the repurposing of algorithmeic skeleton optimisation tools for energy. Furthermore, Jia [48] also proposed a framework to explore GPU-based configurations using stepwise regression modelling for performance optimisation. The *STAtistical Regression-based GPU Architecture analyser* (Stargazer) system derives GPU design related parameters by sampling a small design space producing application-specific performance models that have accurate estimates of program runtime.

Chapter 9

Conclusions

The goals defined at the beginning of this thesis were approached using a variety of different methods, including regression and statistical modelling of parallel benchmarks, meta-heuristic optimisation algorithms and the evaluation approach, which tested the generalisability of the solutions. Here, we summarise the various aspects of the work presented and report the contributions with regard to the defined goals.

9.1 Contributions

The thesis aimed to answer several questions regarding parallel execution, energy prediction and the effects and behaviours of both on the functional language, Haskell. The work was extended to include an imperative programming language, i.e. C/C++, using well-known parallelism libraries. Next, we summarise the details and steps taken to answer the aims noted earlier.

1. **The development of a general optimisation technique to minimise energy consumption:** In the case of programs executing in different ways, i.e. smaller or larger reserved caches, with varying types of processors, different architectures and the availability of multiple clock frequencies, the situation quickly becomes complex. As seen in the modelling of statistical data samples in Chapter 3, the models offered some accuracy. However, when the inputs had different workload factors, the total energy consumed dropped or increased, and the energy prediction results were often inaccurate, sometimes by a large margin. As introduced in Chapter 4 and later evaluated in Chapters 5 and 6, the generic approach overcame the weaknesses of statically modelled energy consumption. Even in cases where energy consumption was low, the meta-heuristic

process successfully captured optimal states for programs with low energy consumption of the benchmarks. In other instances, programs with irregular performances, where energy spikes occurred with different sets of cores, the group of algorithms still produced reasonably accurate levels.

In the literature, efforts to create predictive models and provide safe bounds of energy consumption in programming languages were made. In some cases, the aim was the sequential execution of software. In others, the models were too specific for a unique processor or a complete system, such as building a model for a cacheless SoC with a graphics processor or other processing units. Otherwise, they required complicated steps that would be impossible to implement with an ISA. After examining the work in Chapters 3 and 4, it became apparent that meta-heuristic algorithms provide a preferable approach to prediction inferencing using a statistical model. The ability to derive optimal and near-actual lowest-energy consumption translates to better parallelism optimisation.

2. **Investigating energy modelling for parallel programs using multiple statistical models:** In Chapter 3, the statistical modelling was approached by building a dataset collected from a sample of benchmarks from different programming paradigms and different algorithms that were explicitly designed to run in environments that support parallelism. The sampling was performed with multiple processor clock frequencies and several inputs. The modelling was performed using three approaches, NNLS, parRF, and GLMNET, and the models had their advantages and disadvantages assessed against an unseen set of benchmarks. Certain models performed better than others such as NNLS being more robust with low error margins while others have shown some accuracy.

3. **Energy consumption patterns with regard to execution time and parallel speedup:** With the extensive sampling performed in Chapters 3, despite having scalable and high-performing parallel implementations, increasing parallelism did not yield additional speedup gains. However, it may have had a significant impact on the energy consumed. This was seen in both sets of benchmarks: *PARSEC* and *Nofib*. Another observation was made using the benchmarks from *Nofib*, when samples with different frequencies showed significant energy reductions, and performance had a marginal increase in execution time.

4. **Development and formalisation of meta-heuristic algorithms to estimate**

and optimise energy: In Chapter 4, several variations of meta-heuristic algorithms have been developed to further explore the generalisation of energy consumption optimisation. The results presented in Chapters 5 to 7 showed that optimising energy over small set of configurations can be achieved even with negligible accuracy loss. Applying statistical modelling in combination with meta-heuristic demonstrated limitations with regards to accuracy, however.

5. **Evaluation of energy consumption of multiple programming domains (functional and imperative):** The Chapter 3 presented a list of energy assessment for a range of benchmarks for Haskell and C/C++. The process of modelling datasets for both programming languages demonstrated different types of feature importance e.g. C/C++ giving weights to system calls instead of relying only on instruction counts.

6. **Set of Haskell/C++ energy profiles for PARSEC, Nofib, and the Computer Language Benchmarks Game:** The energy and program metadata such as instruction counts and system calls shown in Chapter 3 provided an overview of language/multi-core behaviour with regards to energy consumption.

9.2 Future Work

Although this thesis explored new domains of optimising and predicting energy, more experiments and ideas remain on tuning the methods discussed in this paper. Here, we review a few of these to help expand the field.

1. **Meta-heuristic Optimisation Algorithms for Multiple Objectives:** The experiments and data discussed in Chapters 5 and 6 provided a glimpse of the improvements possible when using specialised and generalised algorithms for energy analysis. Initial algorithm designs were based on optimising the execution setup, and this activity was worthy of deep examination. The next step should optimise all four algorithms to support more objectives, such as optimising energy consumption while maintaining a speedup of 4× or keeping memory usage below a particular size. There are multiple ways of implementing the base algorithms for multiple-objective goals in the literature [17, 56, 105].

2. **Energy-aware Runtime System:** CPUs, GPUs and memory can be equally demanding when it comes to power requirements. As languages, such as Haskell, are managed using a *runtime system*, understanding how each part of the language consumes energy would benefit the language as a whole. Evaluating runtime algorithms would improve the ability to control energy consumption and performance. Because RTS has many roles in managing program execution, memory management and scheduling, some features could benefit from modelling the general algorithms employed.

3. **Optimisation Extensibility:** We implemented optimisation algorithms that focused on classical `x86` architectures with symmetrical processing designs. It would be beneficial to broaden the scope to other processor designs and architectures. For example, *asymmetric multicore processors* (AMPs), such as those used in mobile phones, have a variety of multicore processing units, where the types of computations and energy consumption activities may have unique behaviours.

4. **Parallel Skeleton Energy Estimates:** The black-box energy consumption may need a better understanding of specific program structures, for example, addressing specific parallel skeletons concerning energy estimates without the need to sample the functions running in such parallel structures. Meaning that a framework of sampling simple sequential functions with inputs of some domain can help estimate if a given parallel skeleton would be advantageous compared to other skeletons when it comes to energy consumption.

Appendix A

Parallel Example Implementations

A.1 Data Parallel Implementation Example

A simple implementation of that can be demonstrated in the Listing–6, Line 4 defines a simple function to add the integer 3 to an argument `x` and return it to the callee, the Lines 7-8 define a set two constants used in configuring the parallelism, Lines 9-10 define two arrays one used to store the result from applying `add3` and another array `threadArr` to store the thread id that executed that particular iteration, the OpenMP directive definition on Line 12 defines that a `for` loop would run in `parallel` and that the scheduling is done dynamically where an internal queue would be used to distribute a set of chunks of the computation between threads. Lastly, the Lines 17-18 print a set of values of the index 400 to show the result and the number of thread that executed the iteration that produced it.

```c
#include <stdio.h>
#include <omp.h>

int add3(int x) { return x + 3; }

int main() {
    const int length = 1000000;
    const int chunk = 1000;
    int arr[1000001];
    int threadArr[1000001];

#pragma omp parallel for schedule(dynamic, chunk)
    for (int i = 0; i < length; i++) {
        arr[i] = add3(i);
        threadArr[i] = omp_get_thread_num();
    }
    printf("N-th element: %d\n", arr[400]);
    printf("N-th element was generated by thread: %d",
    ↪ threadArr[400]);
    return 0;
}
```

Listing 6: Example showing an implemented version of the diagram in Figure–2.2 using OpenMP

A.2 Task Parallel Implementation Example

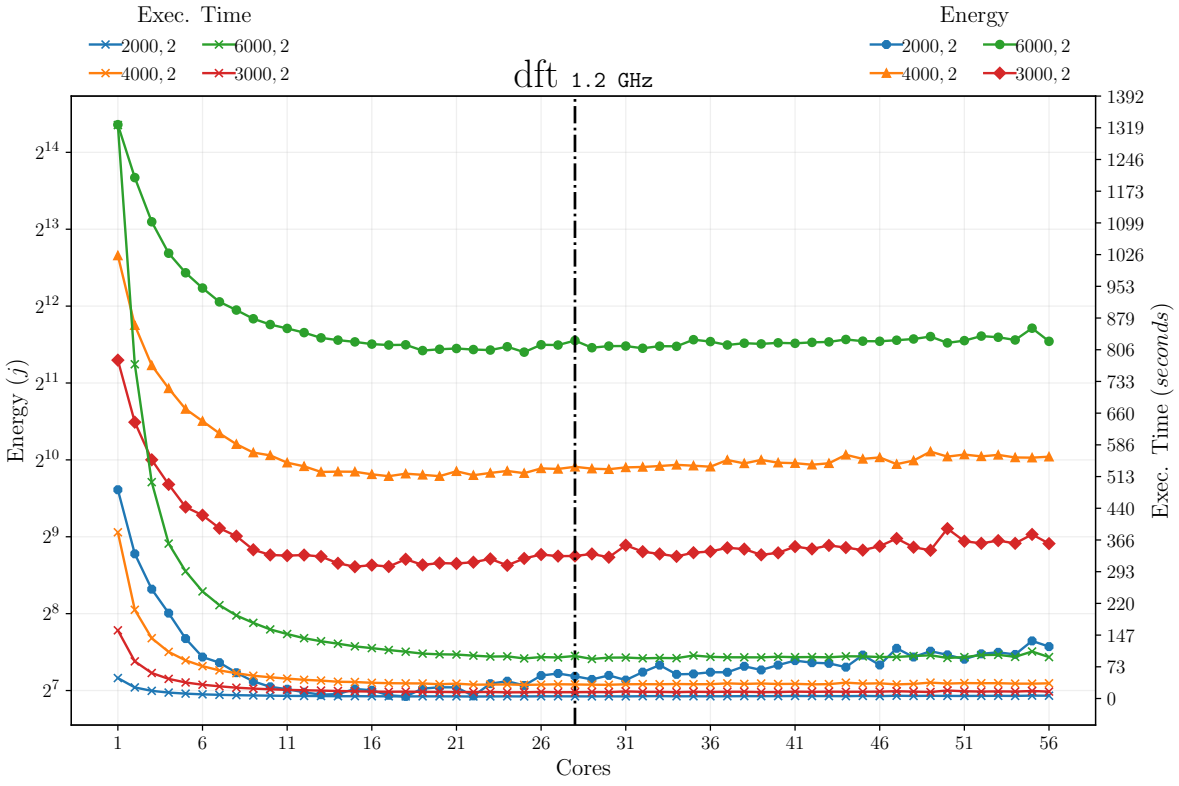
An example implementation of the last figure can seen in Listing–7, Lines 4-6 define functions that do basic computations over integers and floats, line 9 declare an `int` variable to be used later in a loop, Lines 11-14 declare a set of arrays of the same size for the purpose of the execution, Lines 18-19 uses a loop to populate the `numbers` array with a set of integers from 0 to 100, Line 23 is an OpenMP directive that defines a set of `parallel sections` each section would run in parallel to others, the `shared` clause states that arguments passed will be shared among threads executing in the `sections` region, the `num_threads` is a clause to specify the number of threads in a *thread team* to be used in scheduling work between the different sections. The `section` directive is a structured block that is defined to run in parallel, in the case of the current example, each section containing a loop with a fixed size to apply a single function over the elements of the `numbers` array then storing the result in a new array, in this case `squares`, the printing the thread id that performed the execution of that iteration, a similar set of operations happen in the other sections with a small difference of computations and arrays used to store the results, finally after the `omp` sections complete executing the set of print commands Lines 44-46 print random indexes from the arrays containing the results.

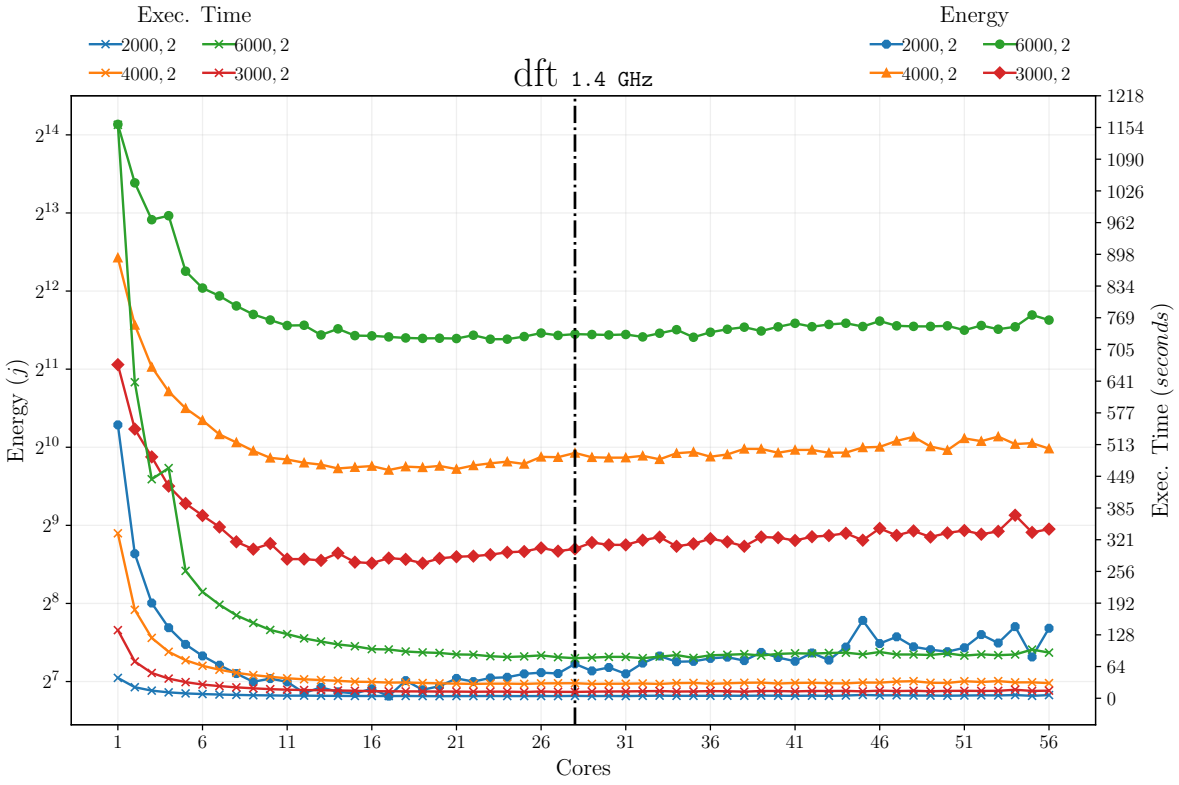
```c
#include <stdio.h>
#include <omp.h>

int square(int x) { return x * x; }
int multiply(int y) { return y * 2; }
float divide(float z) { return z / 2; }

int main() {
    int i;

    int numbers[101];
    int squares[101];
    int multipliers[101];
    float divisions[101];

    /* create a sequence of integers from 0 to 100 and save them in
     * array numbers */
    for (i = 0; i <= 100; i++) {
        numbers[i] = i;
    }


#pragma omp parallel sections shared(numbers,squares,multipliers,divisions)
↪  num_threads(6)
{
#pragma omp section
          for (int p = 0; p <= 100; p++) {
            squares[p] = square(numbers[p]);
            printf("Squares Thread id: %d\n", omp_get_thread_num());
          }

#pragma omp section
          for (int p_ = 0; p_ <= 100; p_++) {
            multipliers[p_] = multiply(numbers[p_]);
            printf("Mult Thread id: %d\n", omp_get_thread_num());
          }

#pragma omp section
          for (int p__ = 0; p__ <= 100; p__++) {
            divisions[p__] = divide((float) numbers[p__]);
            printf("Div Thread id: %d\n", omp_get_thread_num());
          }
}

    printf("Squares Array value 50: %d\n", squares[90]);
    printf("Mult Array value 50: %d\n", multipliers[92]);
    printf("Divs Array value 50: %.3f\n", divisions[100]);

    return 0;
}
```

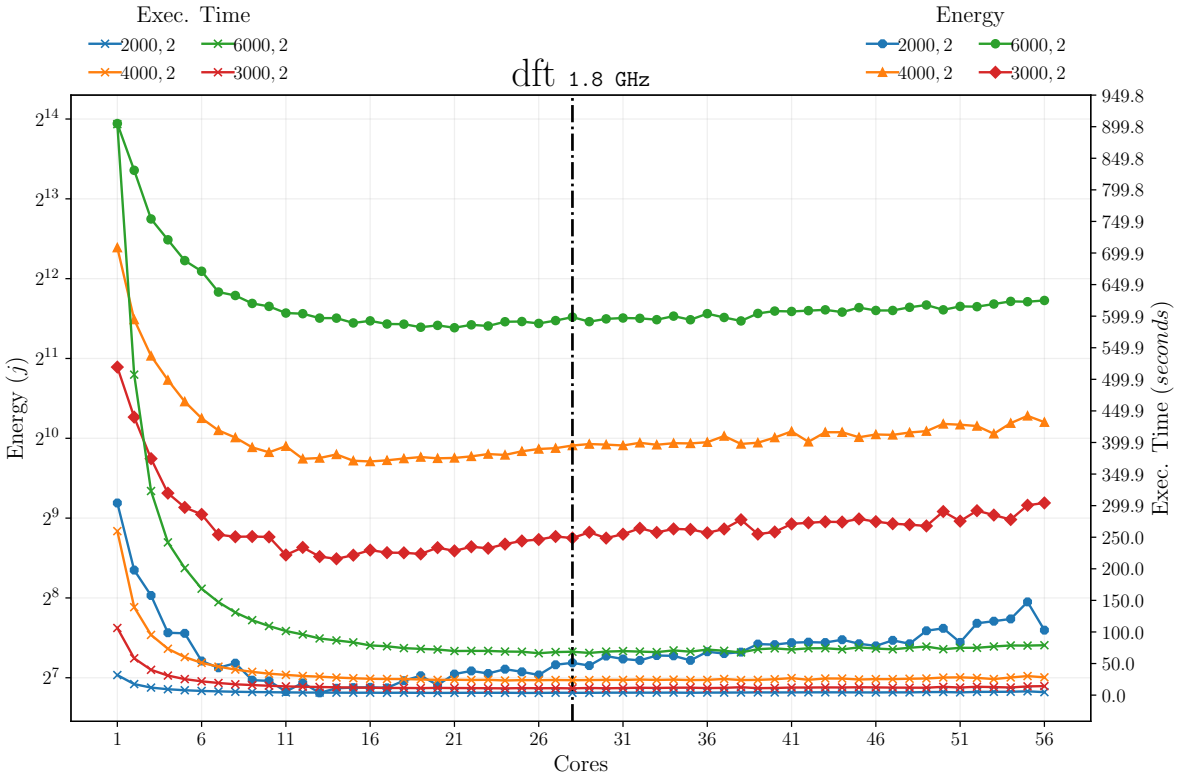
Listing 7: Example showing an implemented version of the diagram in Figure–2.1 using OpenMP

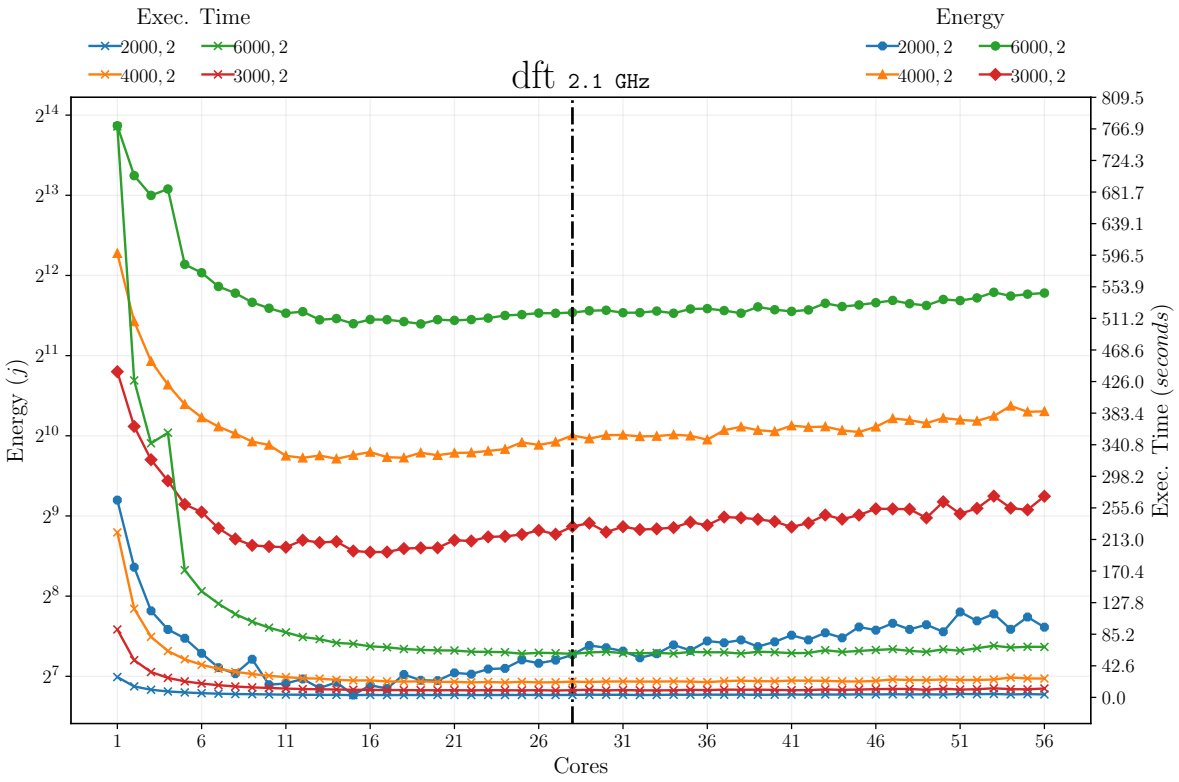
Appendix B

Multiple Frequency Samples




Exec. Time
2000, 2
6000, 2
4000, 2
3000, 2
dft 1.4 GHz
Energy
2000, 2
6000, 2
4000, 2
3000, 2
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
2000, 2
6000, 2
4000, 2
3000, 2
dft 1.8 GHz
Energy
2000, 2
6000, 2
4000, 2
3000, 2
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
2000, 2
6000, 2
4000, 2
3000, 2
dft 2.1 GHz
Energy
2000, 2
6000, 2
4000, 2
3000, 2
Energy (j)
Exec. Time (seconds)
Cores

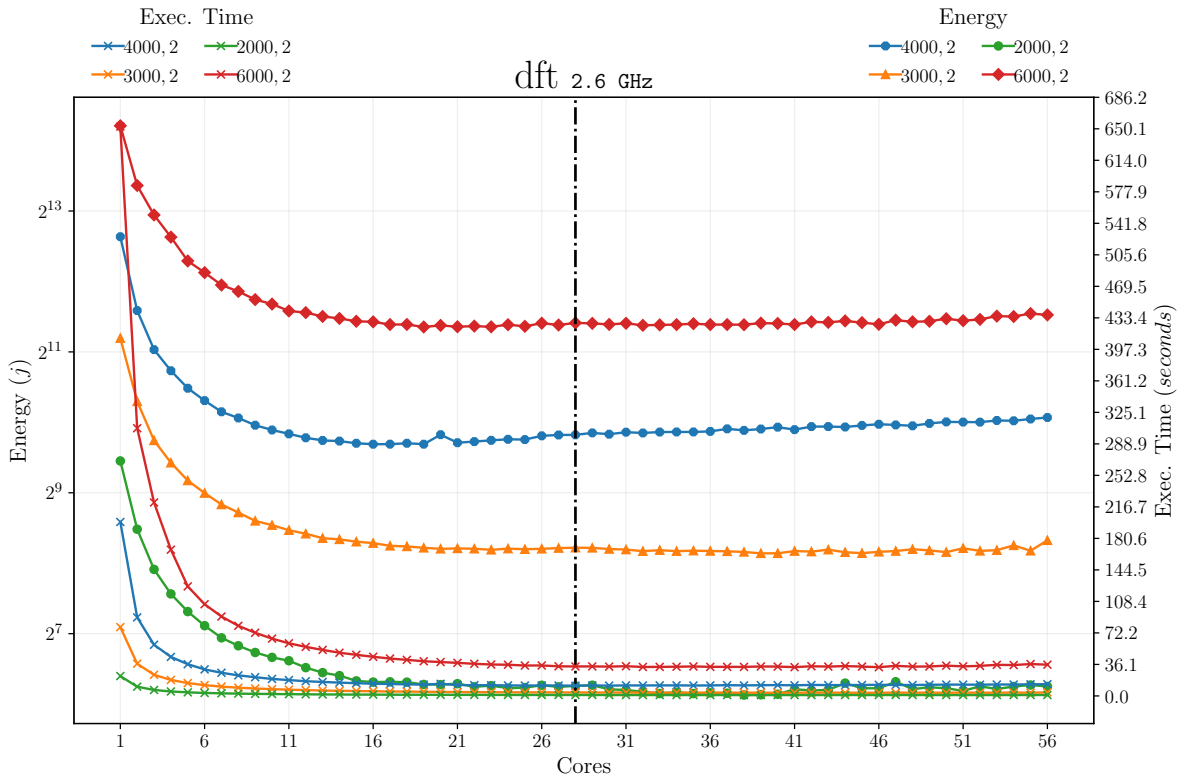
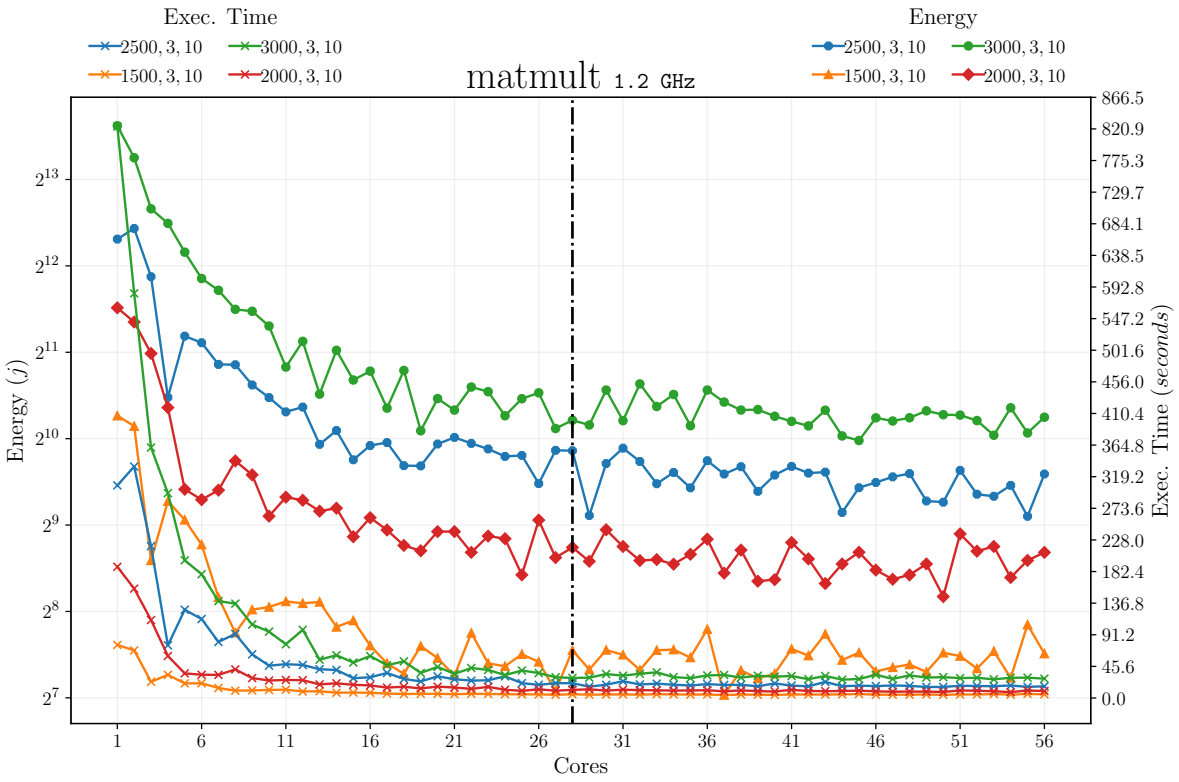
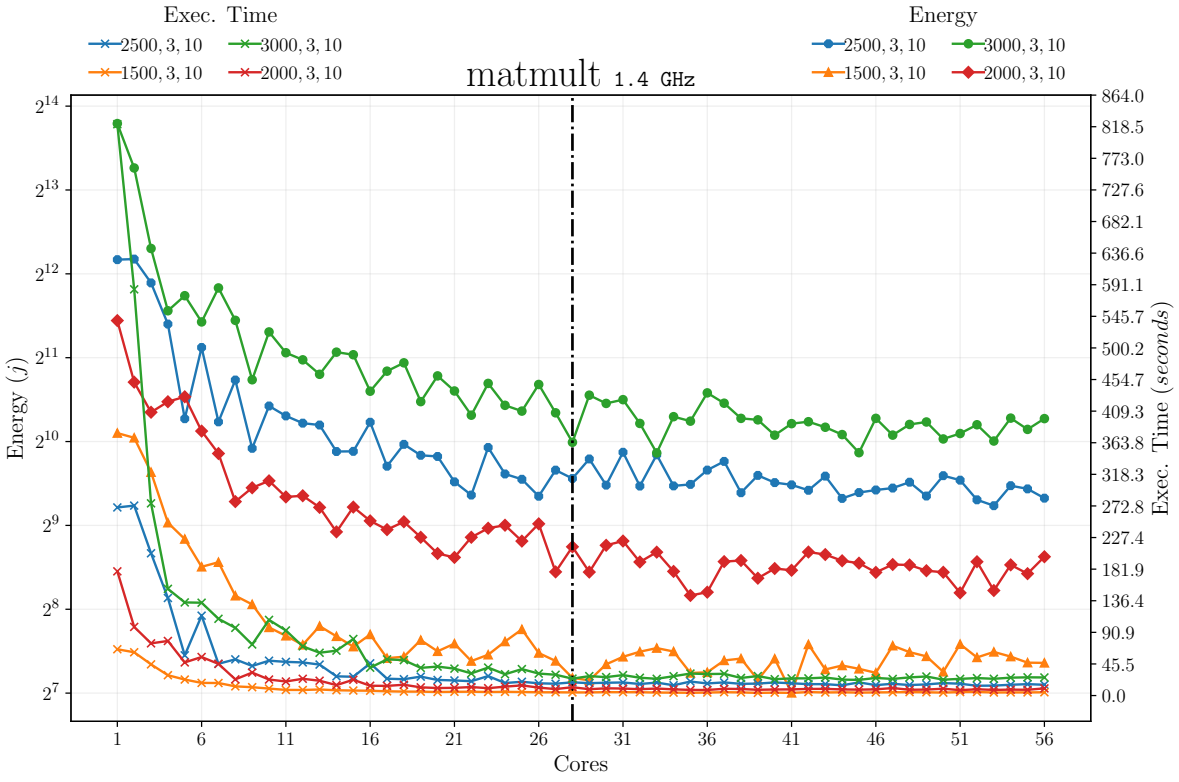
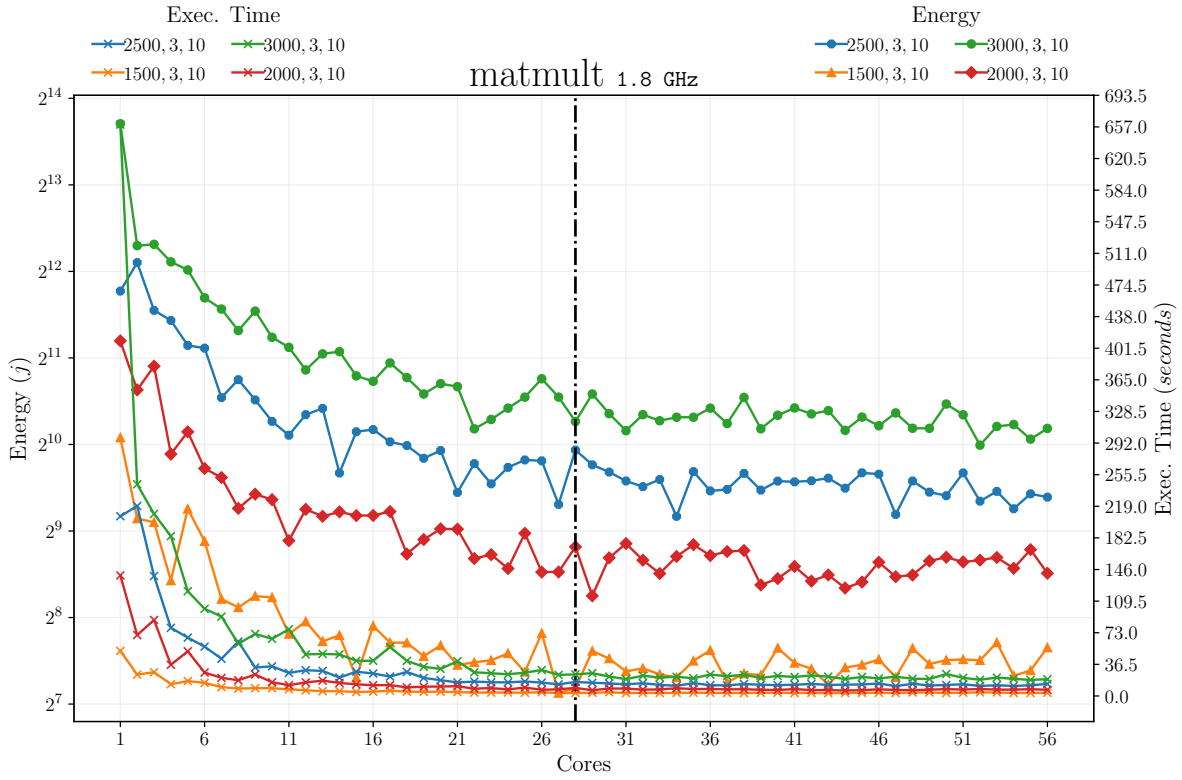
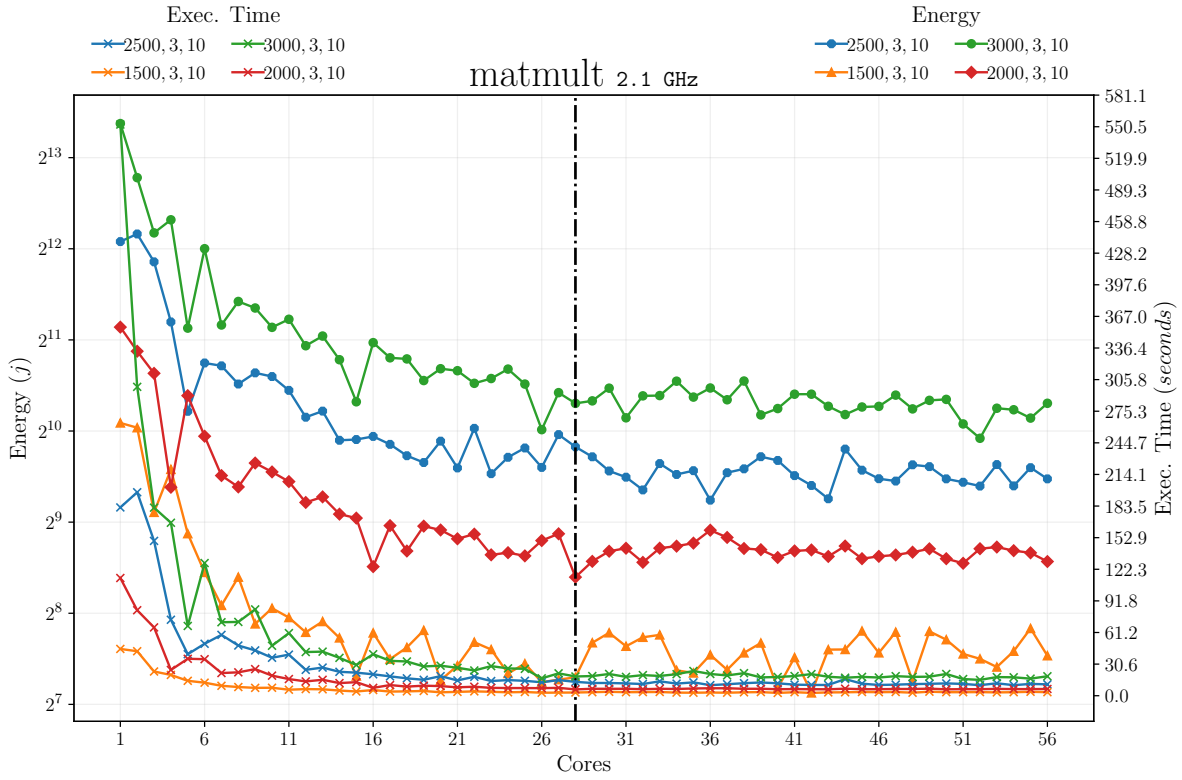


Figure B.1: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **DFT**


Exec. Time
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
matmult 1.2 GHz
Energy
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
matmult 1.4 GHz
Energy
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
Energy
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
matmult 1.8 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
Energy
2500, 3, 10
3000, 3, 10
1500, 3, 10
2000, 3, 10
matmult 2.1 GHz
Energy (j)
Exec. Time (seconds)
Cores

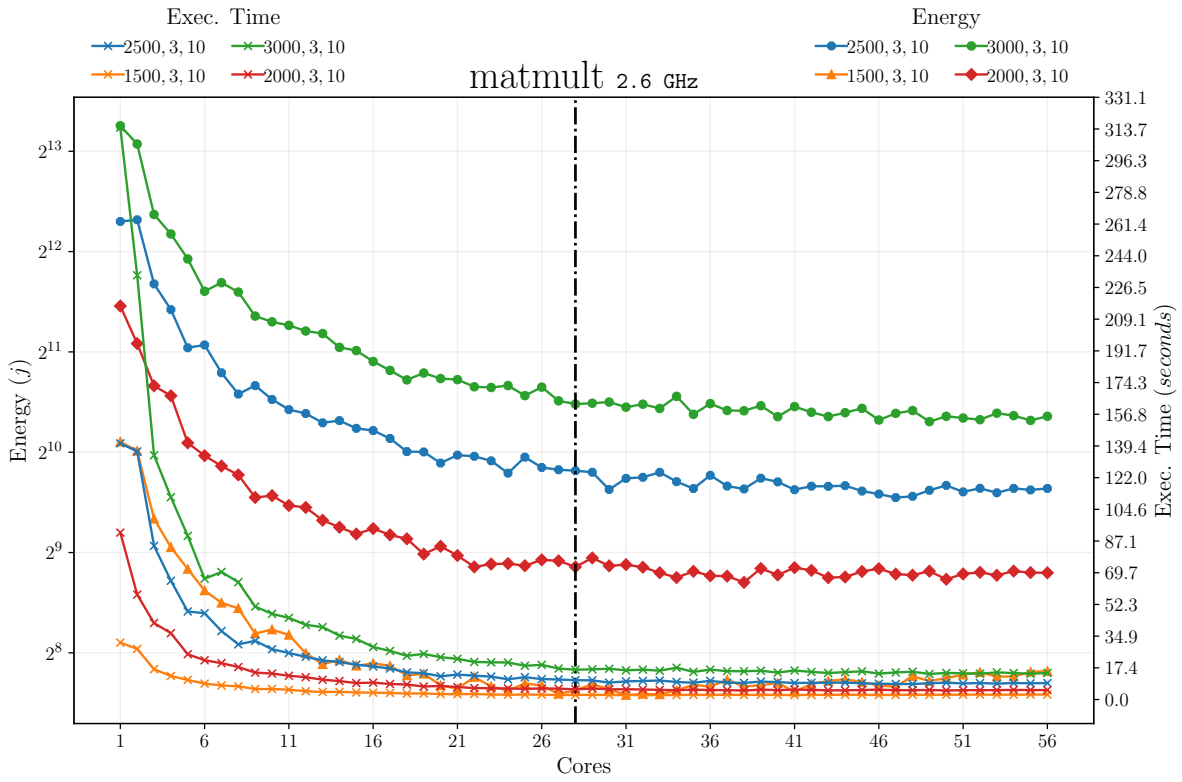
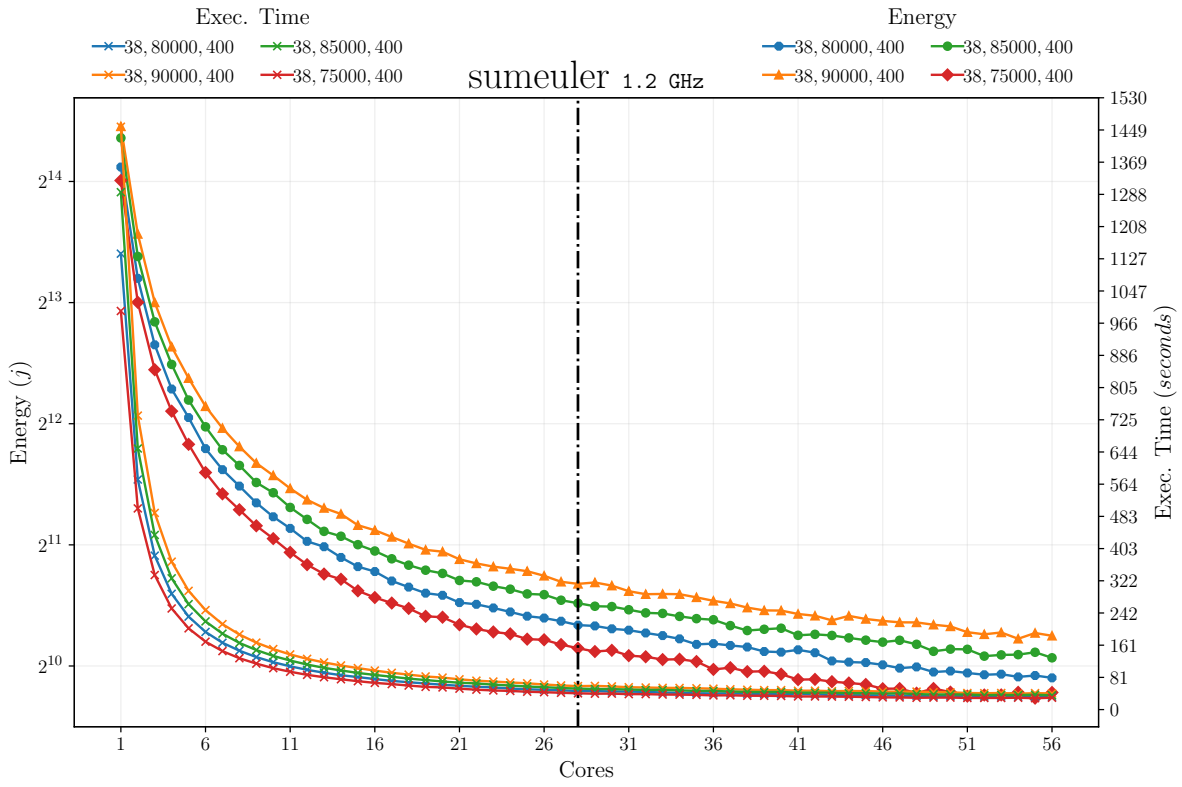
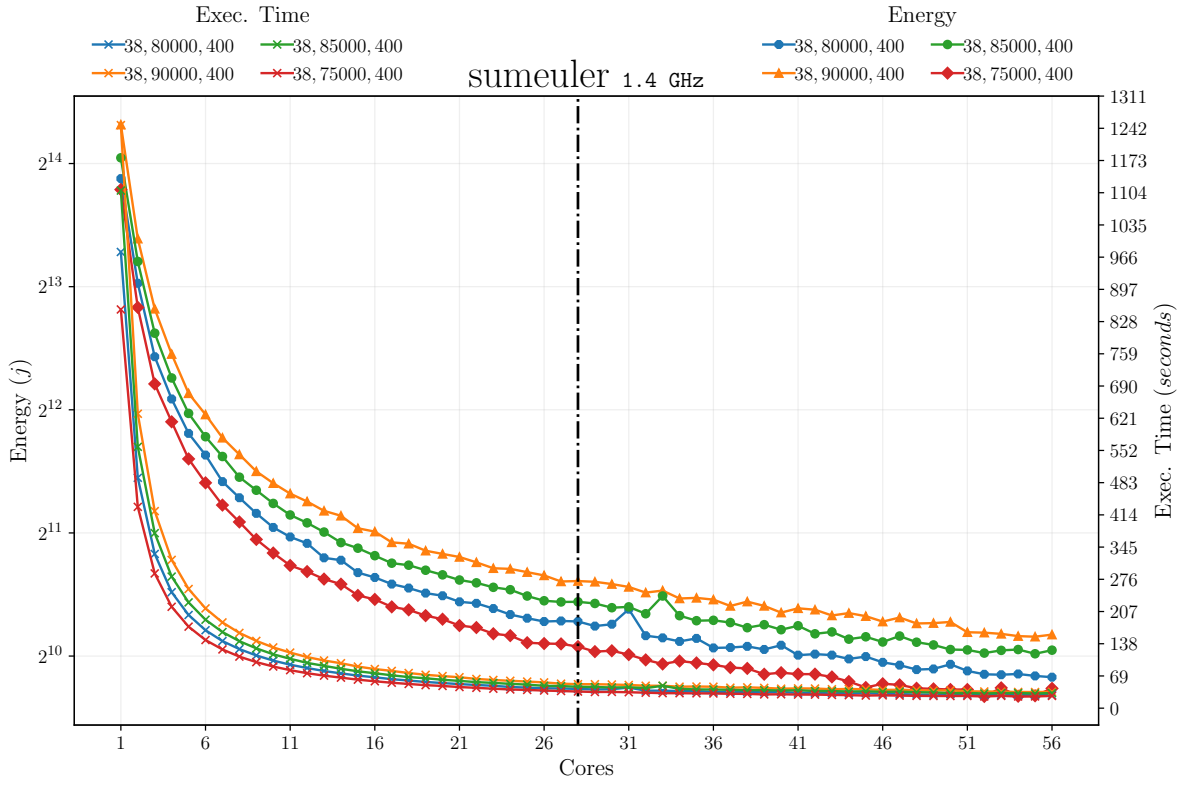
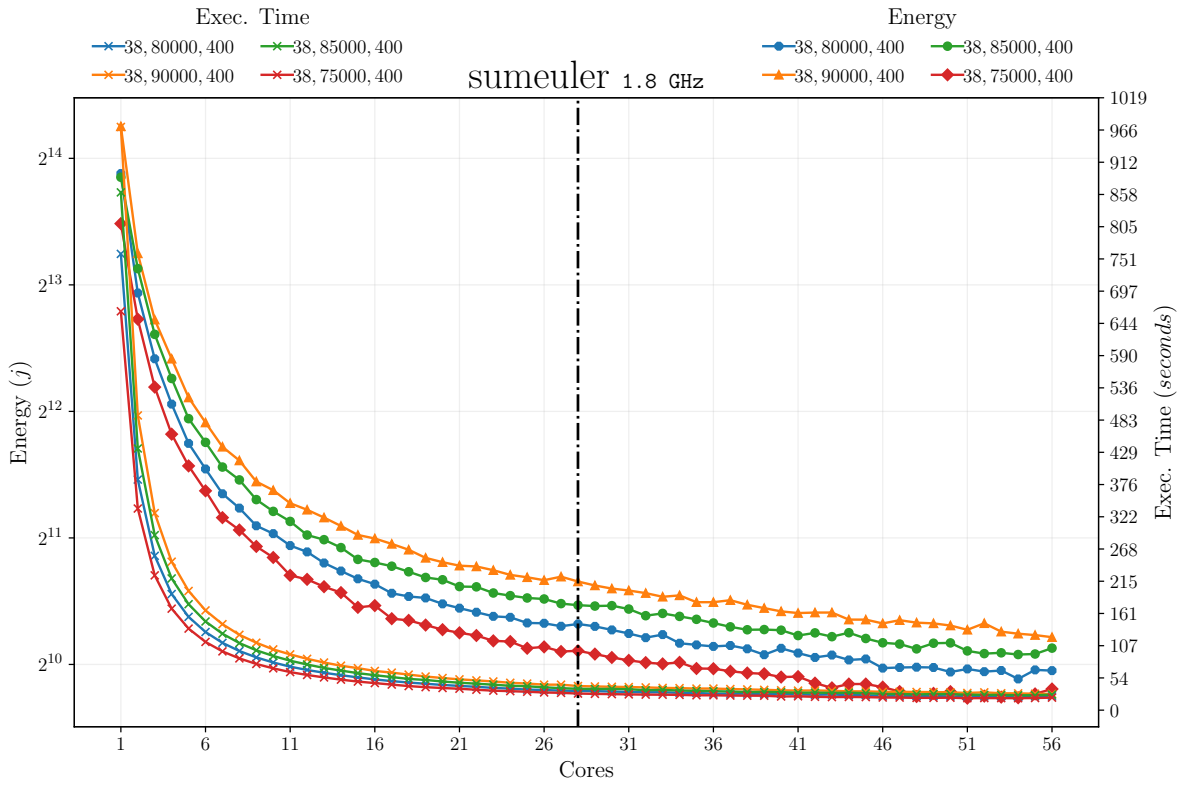
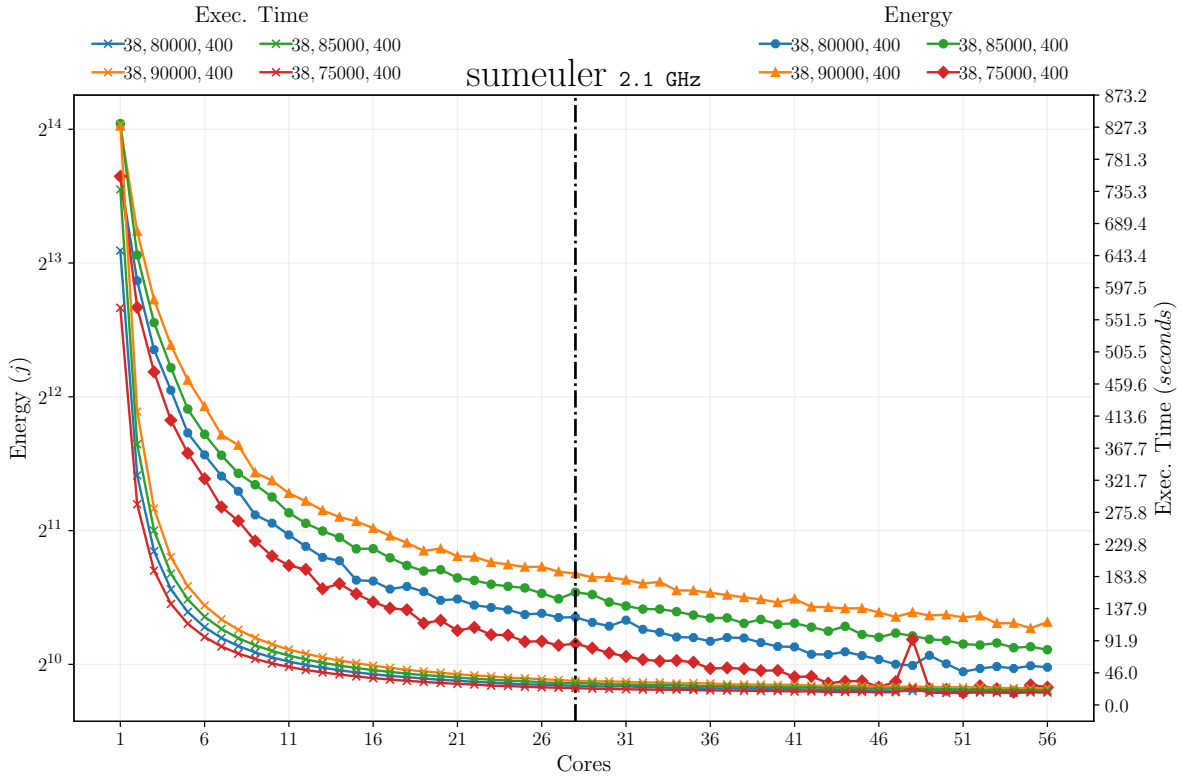


Figure B.2: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **MatMult**


Exec. Time
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
sumeuler 1.2 GHz
Energy
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
sumeuler 1.4 GHz
Energy
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
sumeuler 1.8 GHz
Energy
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
sumeuler 2.1 GHz
Energy
38, 80000, 400
38, 85000, 400
38, 90000, 400
38, 75000, 400
Energy (j)
Exec. Time (seconds)
Cores

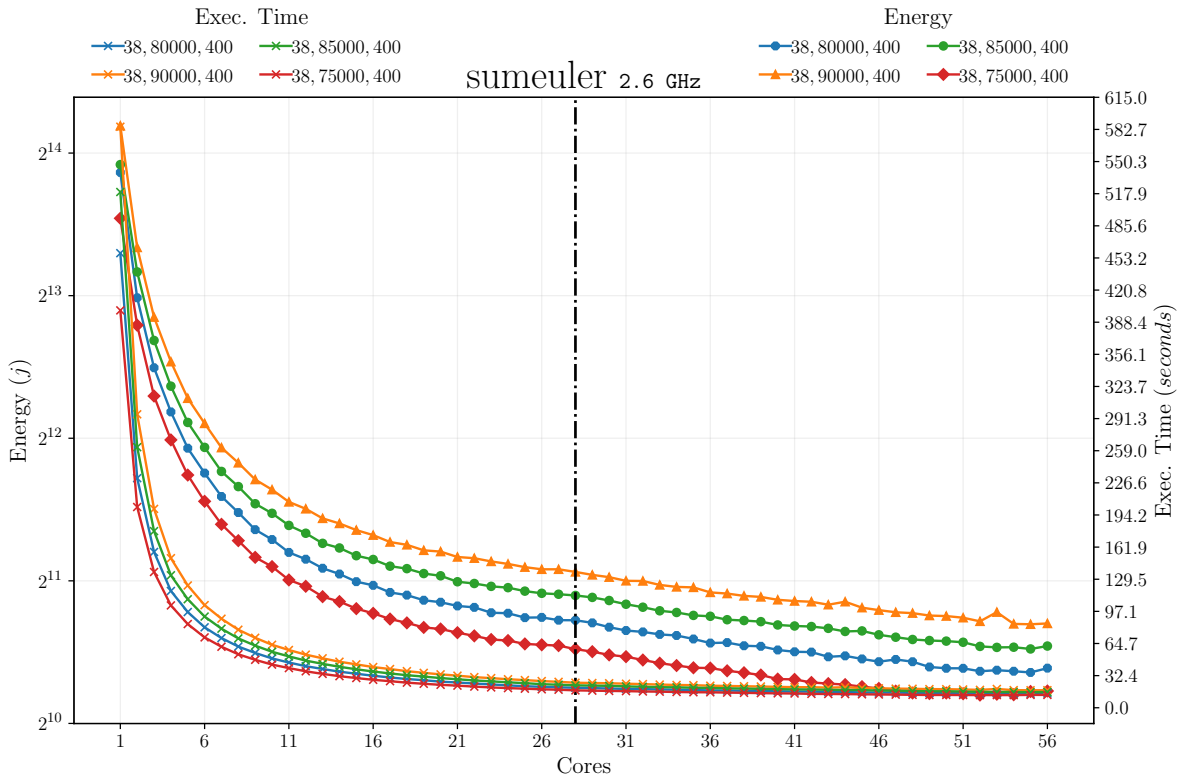
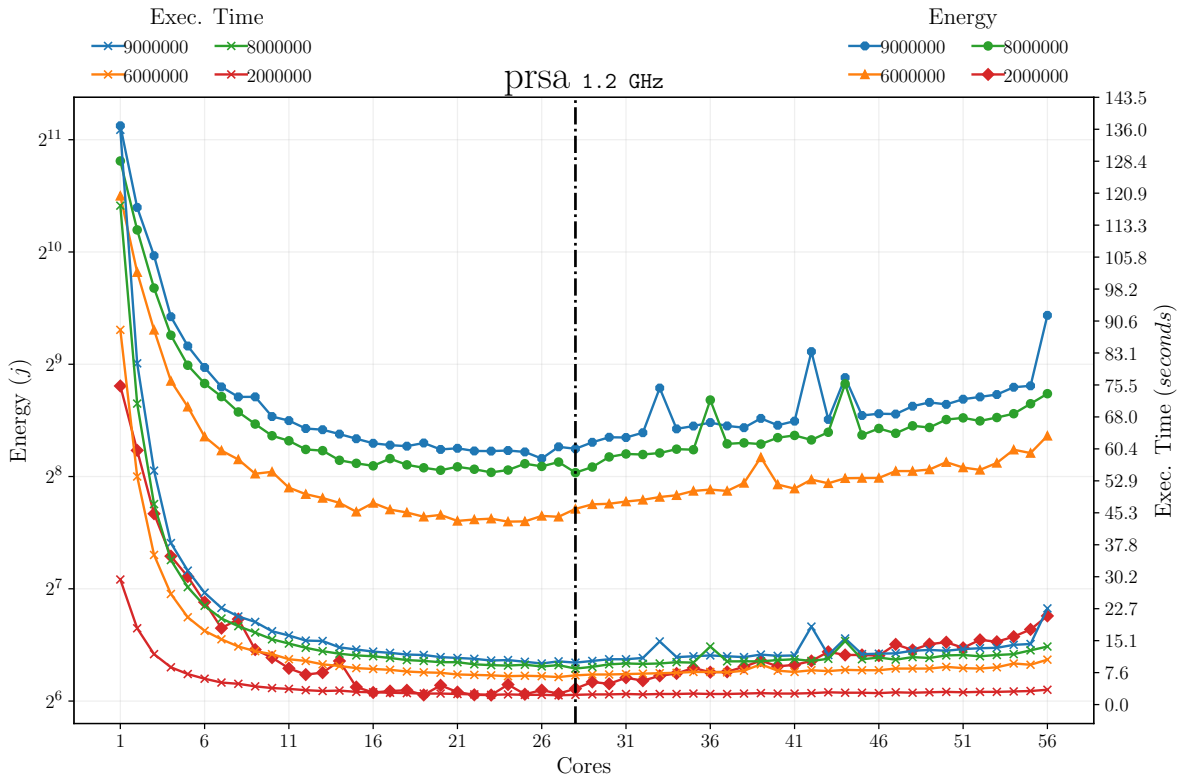
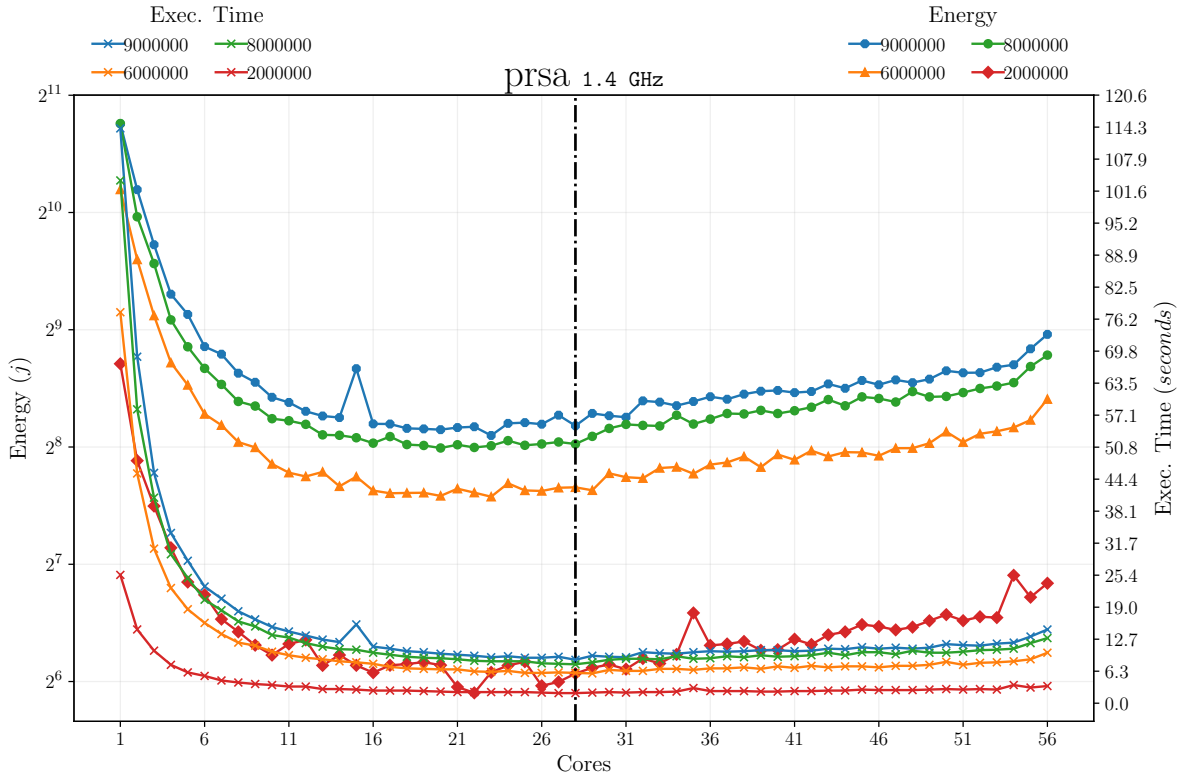
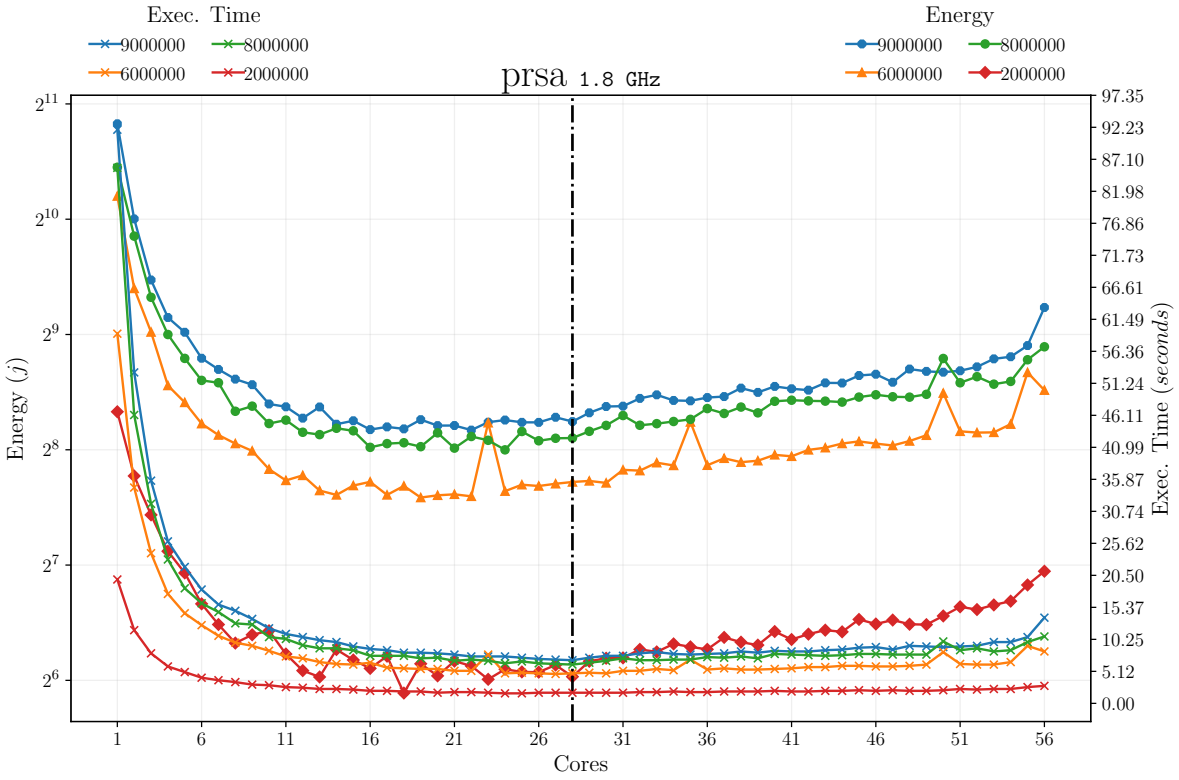
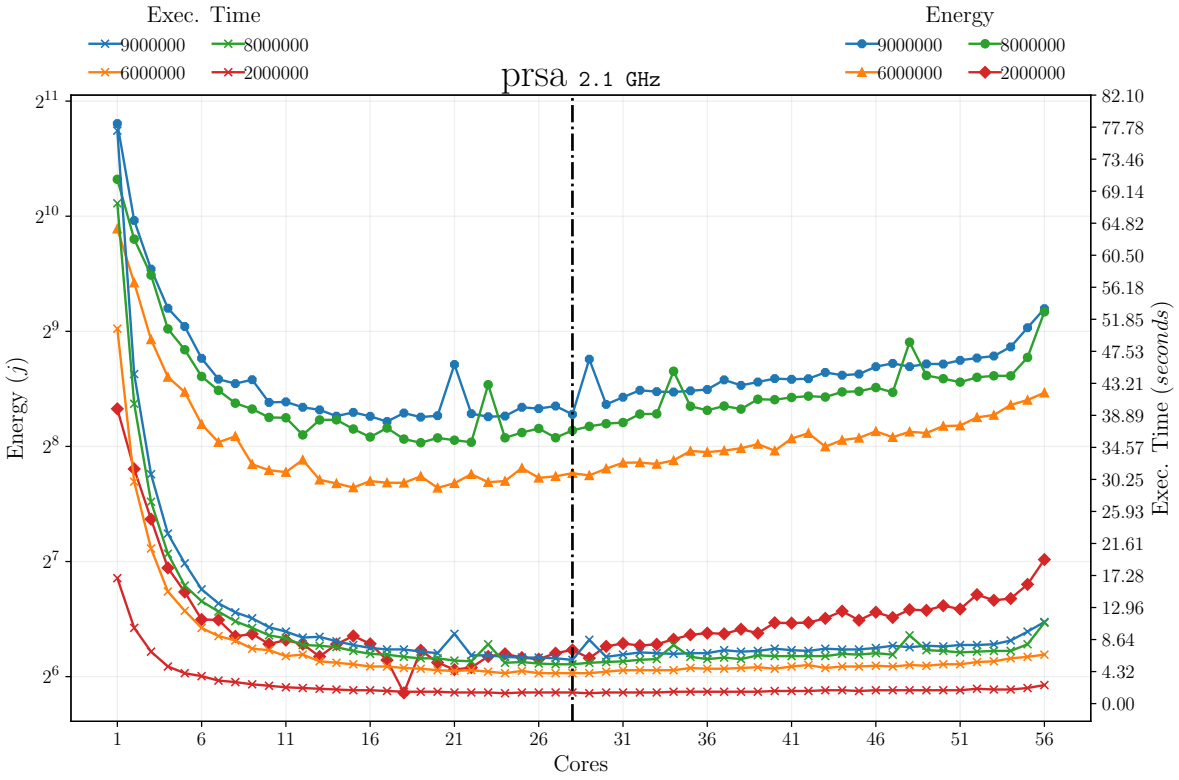


Figure B.3: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **SumEuler**


Exec. Time
9000000
8000000
6000000
2000000
Energy
9000000
8000000
6000000
2000000
prsa 1.2 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
9000000
8000000
6000000
2000000
Energy
9000000
8000000
6000000
2000000
prsa 1.4 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
9000000
8000000
6000000
2000000
Energy
9000000
8000000
6000000
2000000
prsa 1.8 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
9000000
8000000
6000000
2000000
Energy
9000000
8000000
6000000
2000000
prsa 2.1 GHz
Energy (j)
Exec. Time (seconds)
Cores

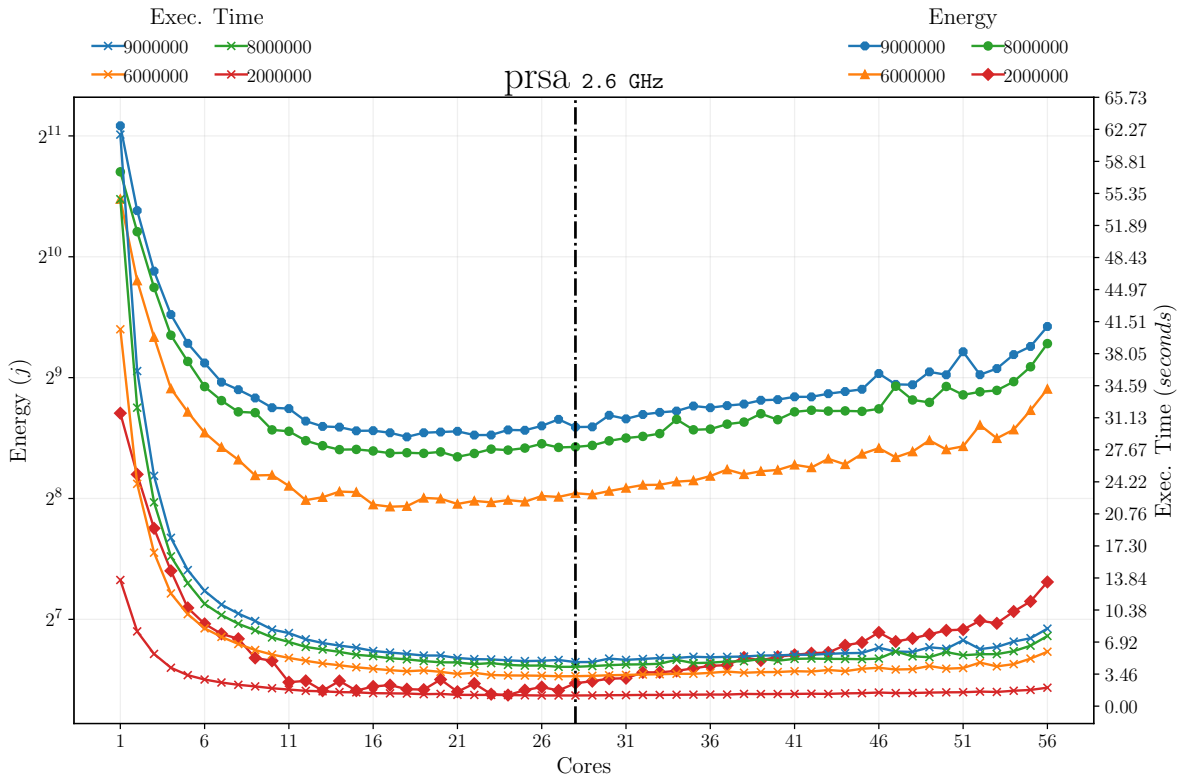
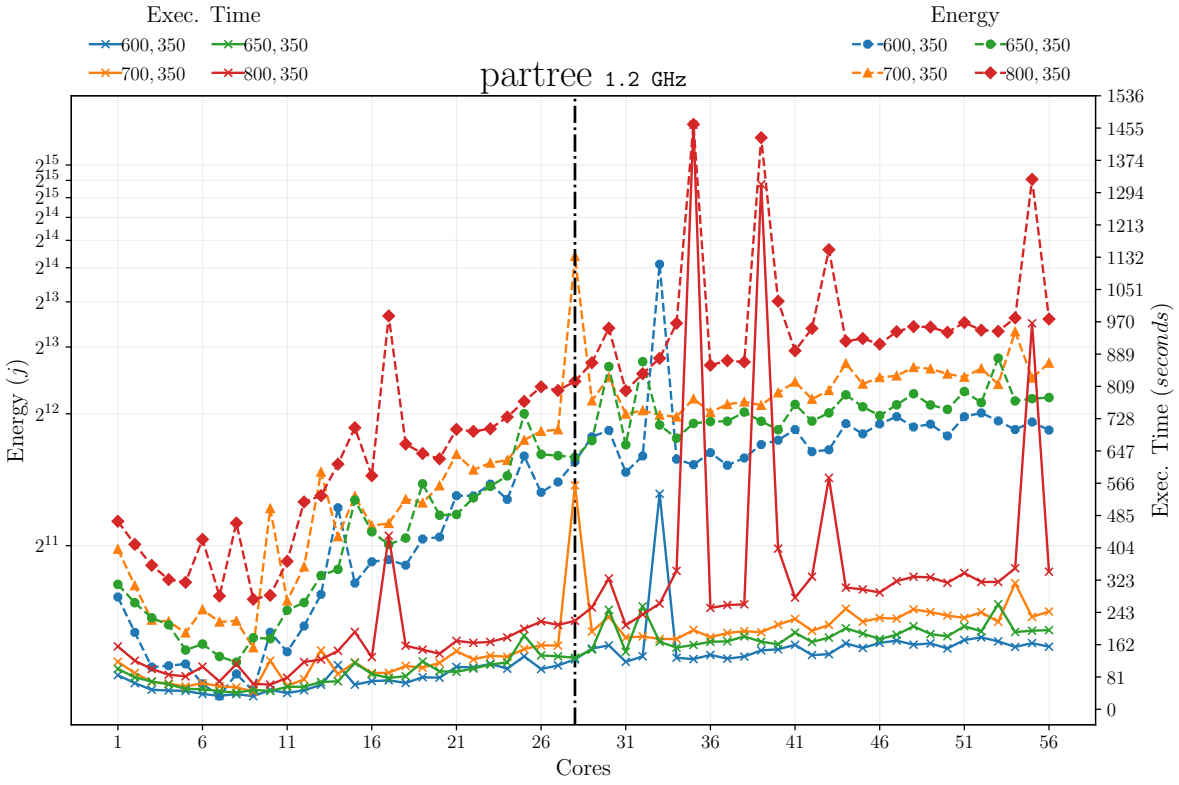
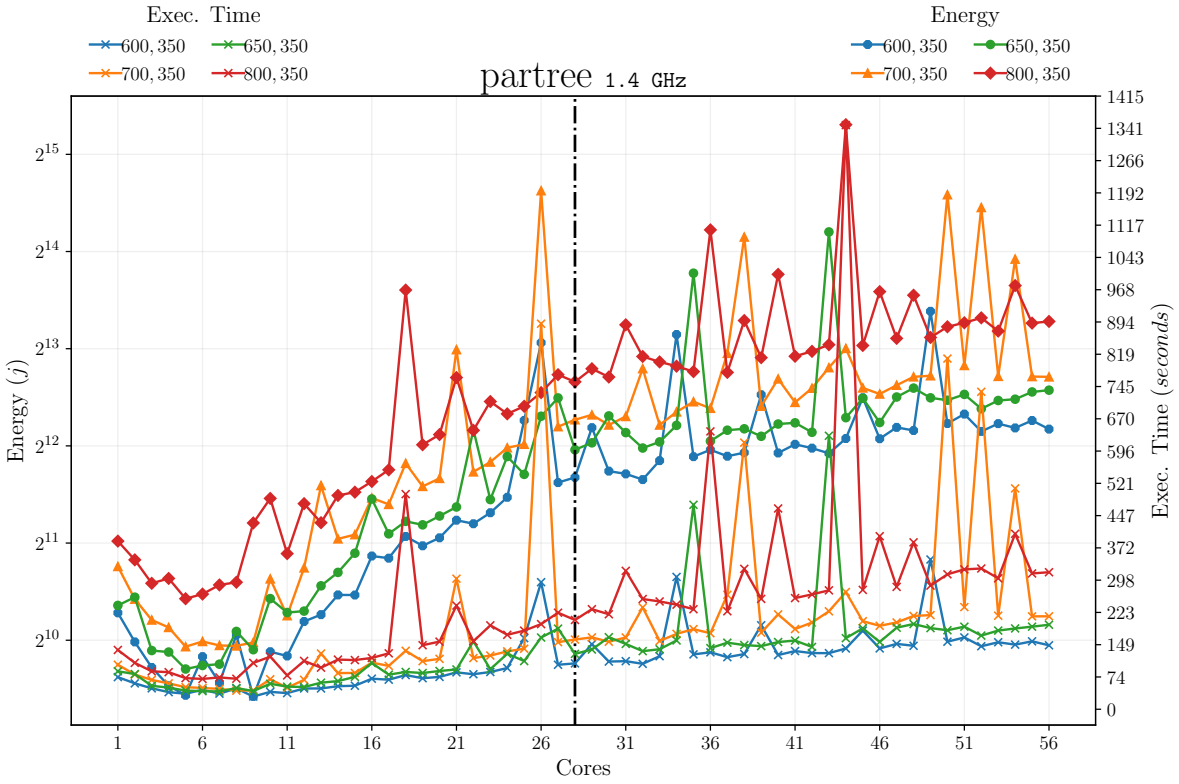
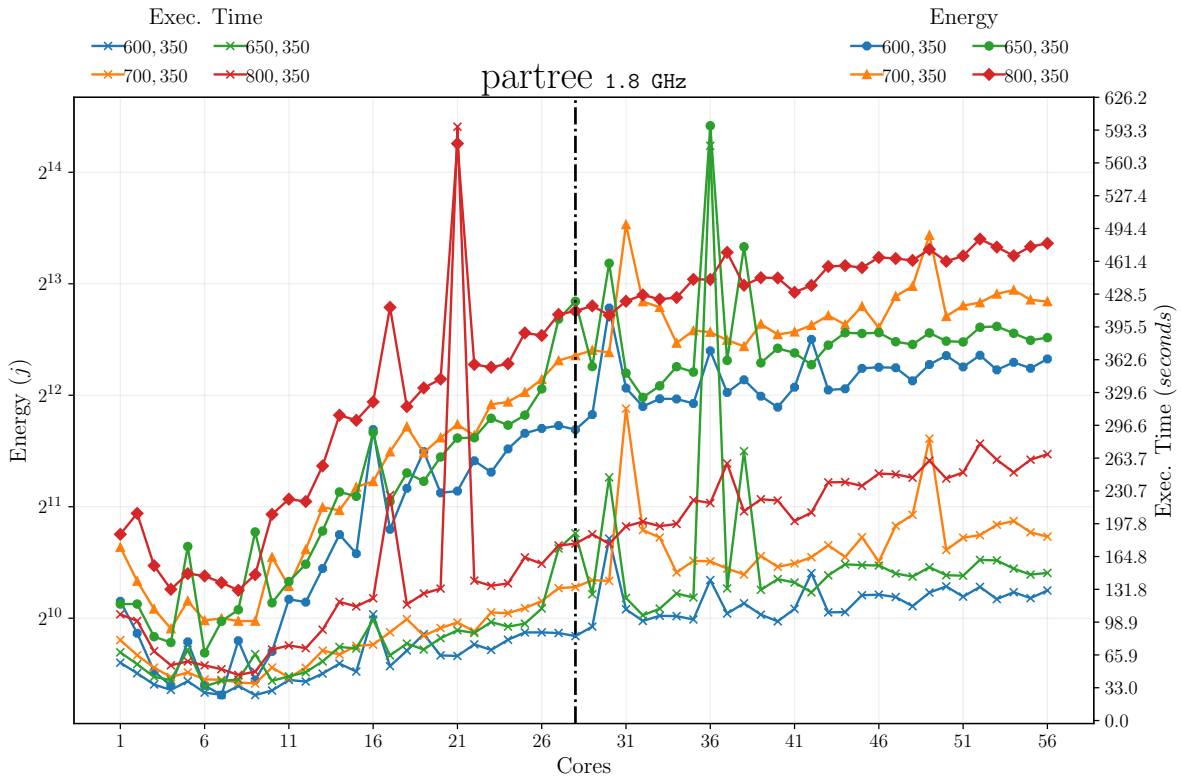
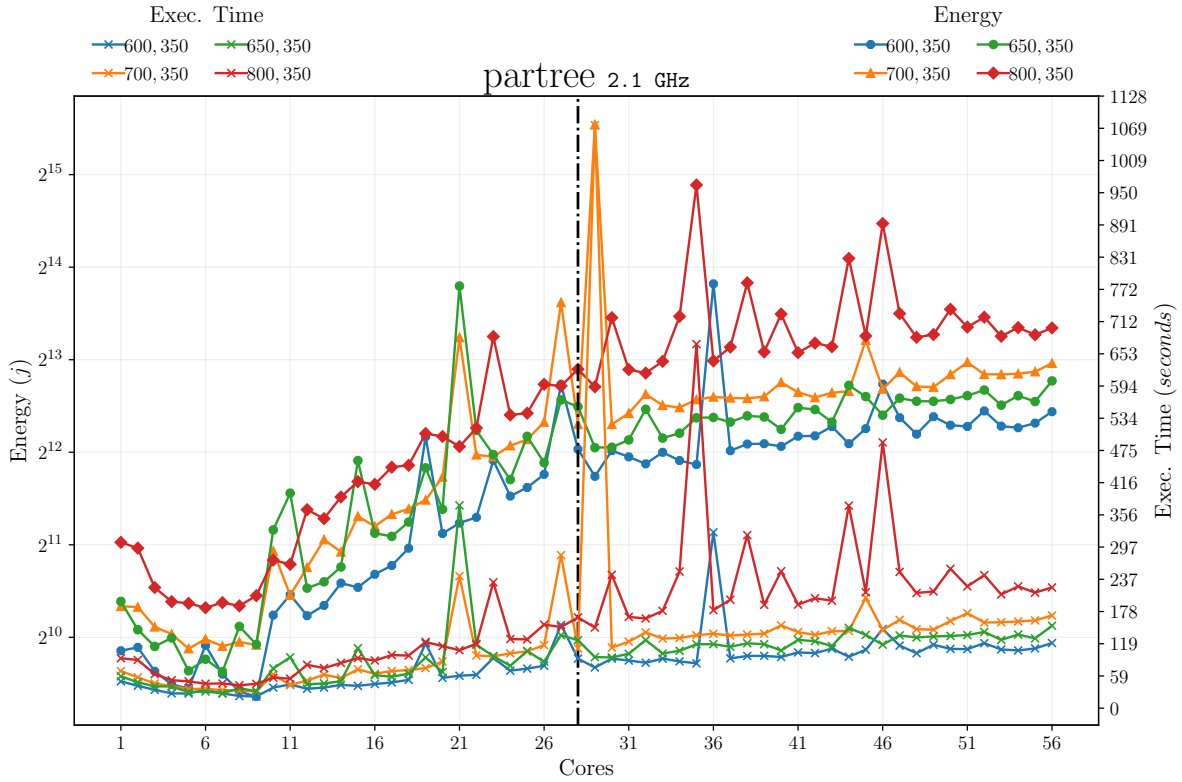


Figure B.4: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **PRSA**


Exec. Time
600, 350
650, 350
700, 350
800, 350
partree 1.2 GHz
Energy
600, 350
650, 350
700, 350
800, 350
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
600, 350
650, 350
700, 350
800, 350
partree 1.4 GHz
Energy
600, 350
650, 350
700, 350
800, 350
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
600, 350
650, 350
700, 350
800, 350
Energy
600, 350
650, 350
700, 350
800, 350
partree 1.8 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
600, 350
650, 350
700, 350
800, 350
Energy
600, 350
650, 350
700, 350
800, 350
partree 2.1 GHz
Energy (j)
Exec. Time (seconds)
Cores

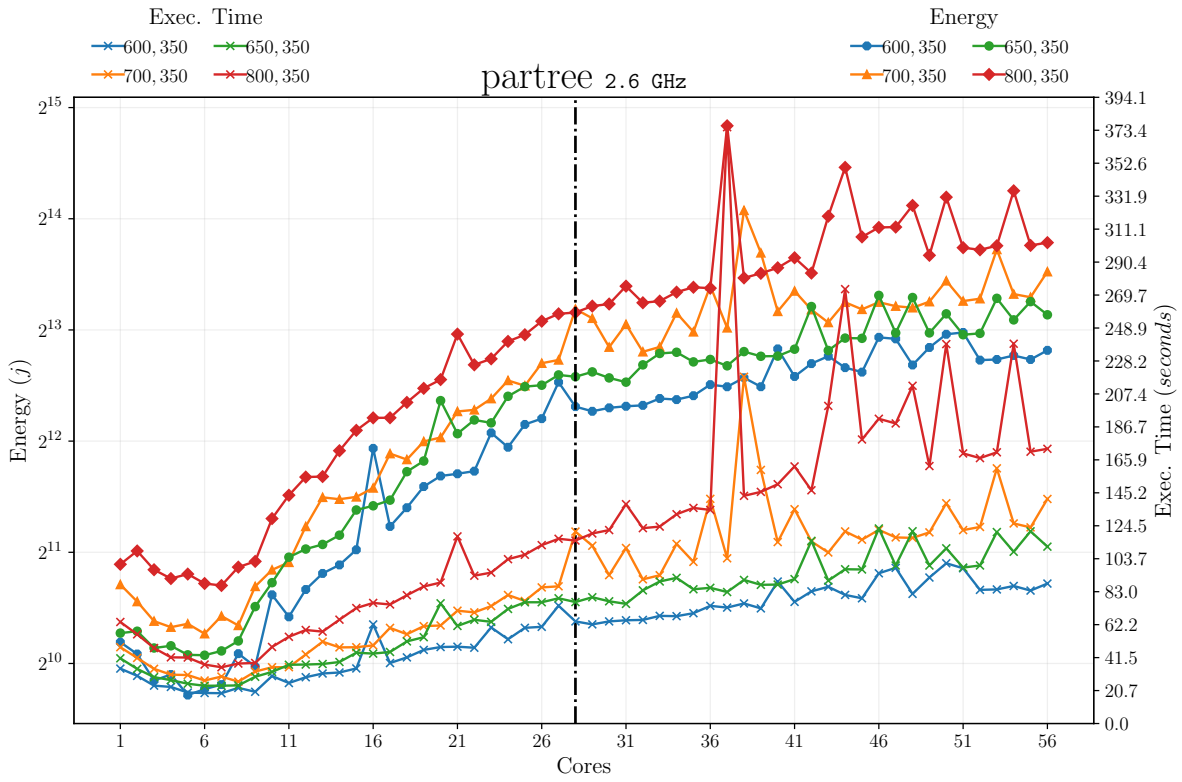
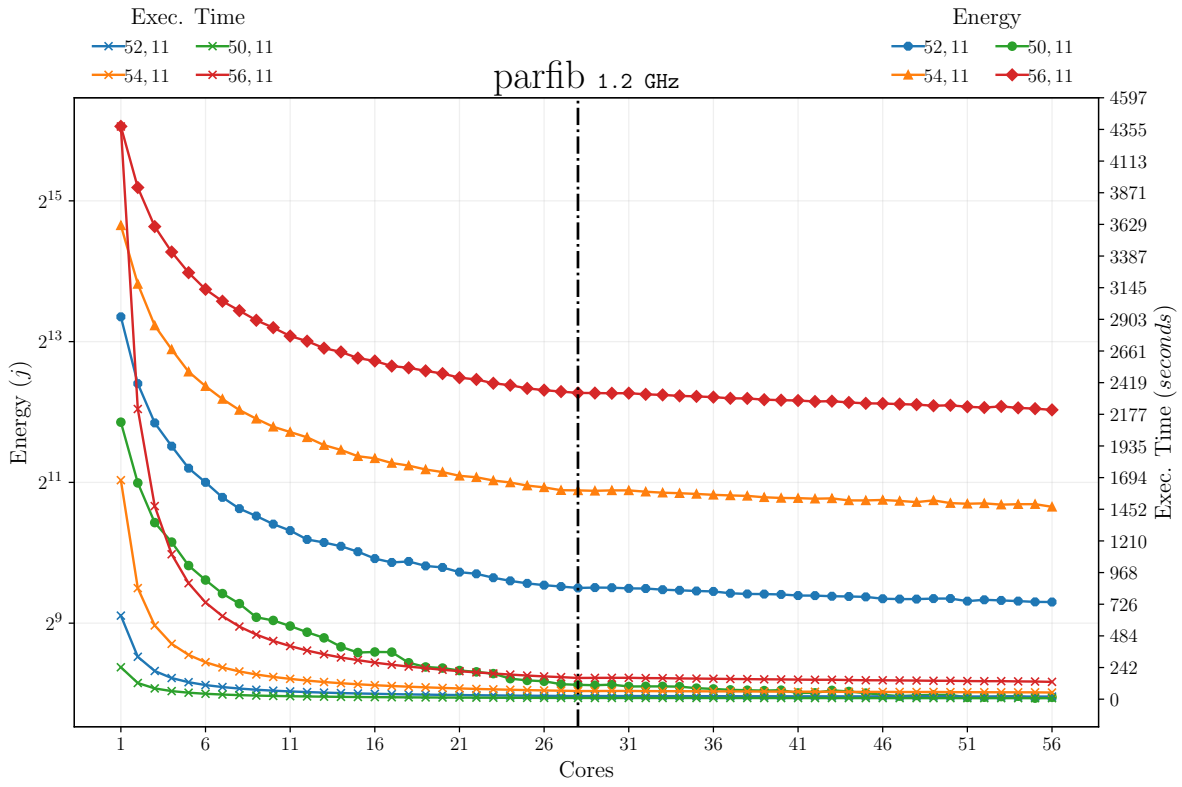
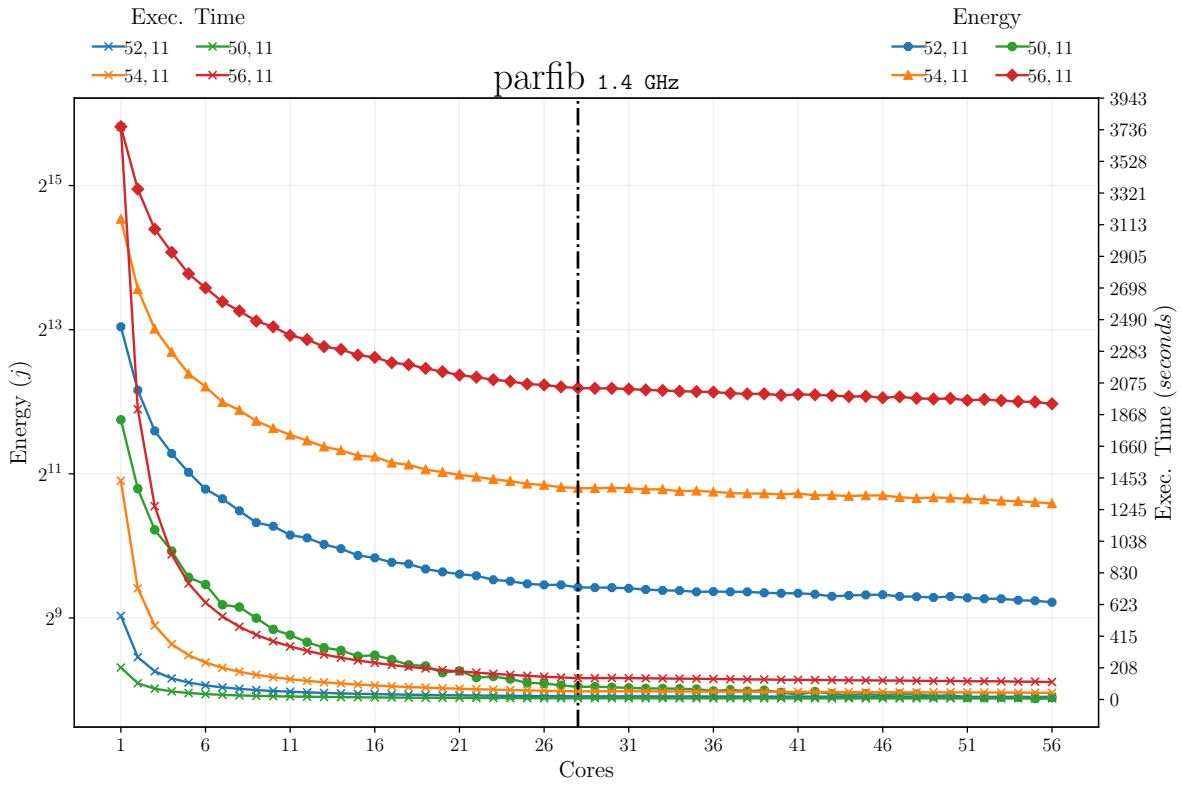
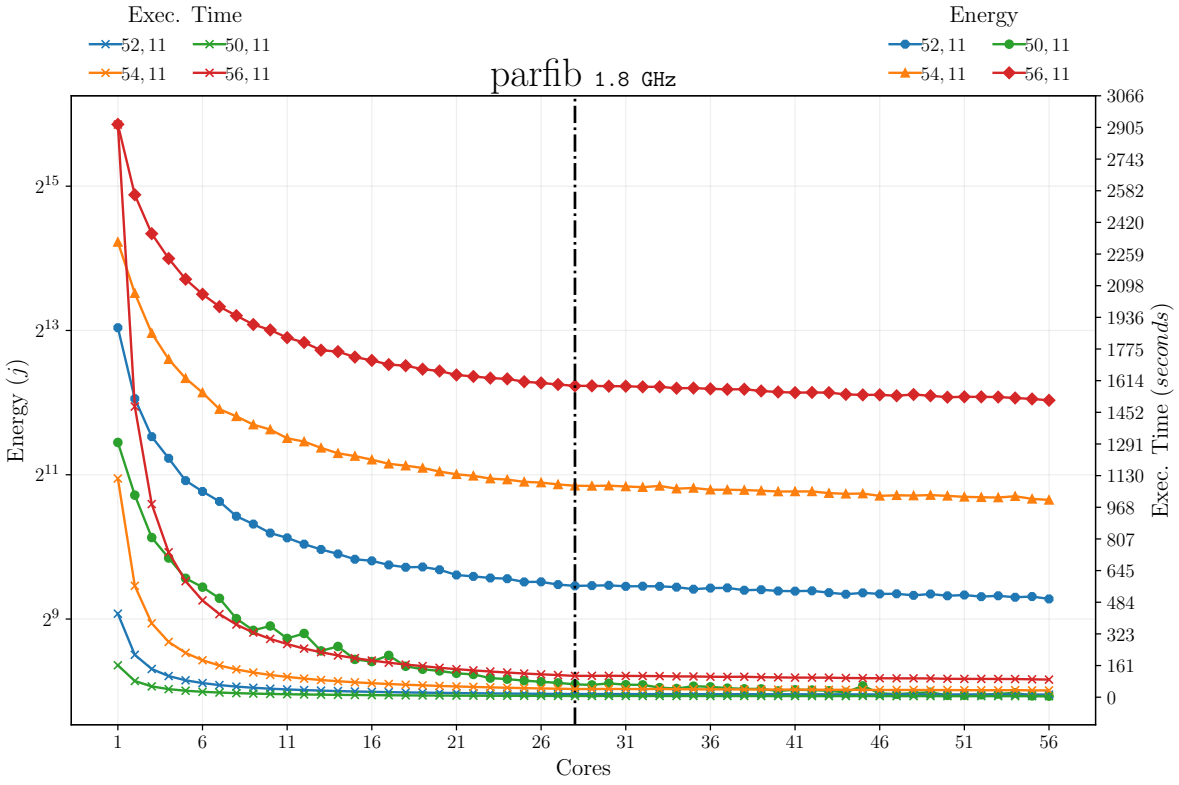
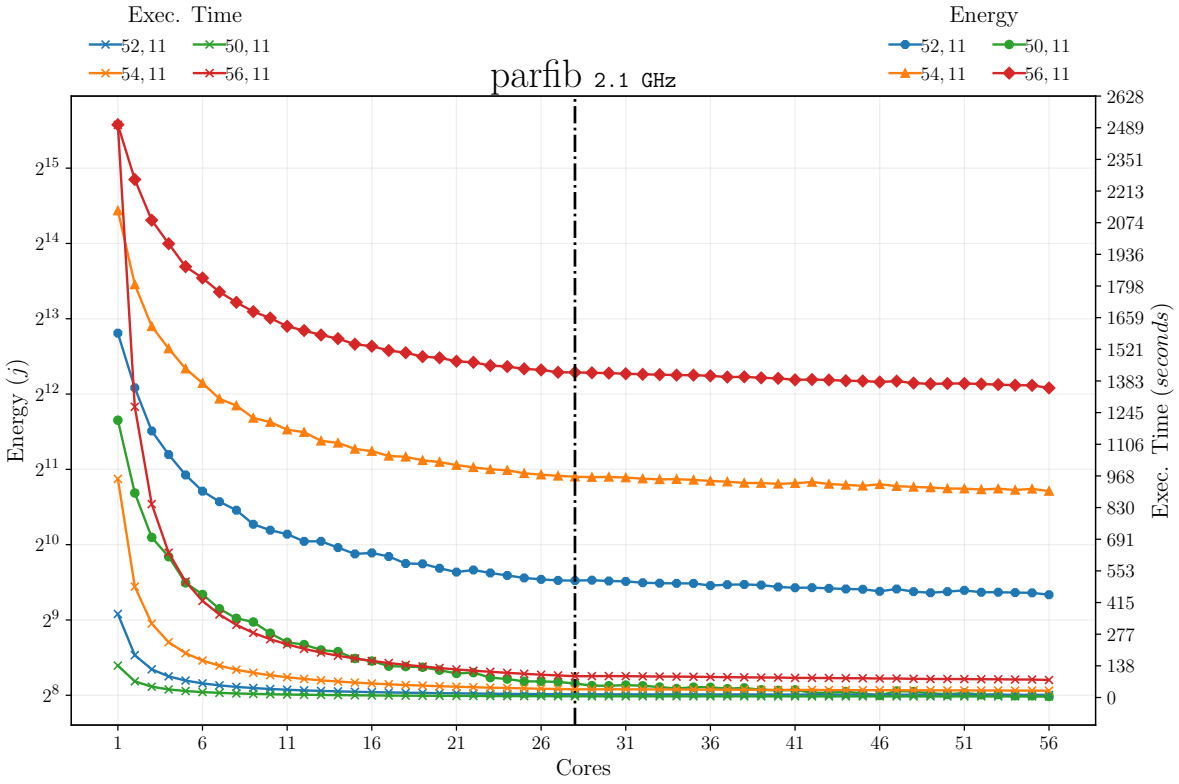


Figure B.5: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **Partree**


Exec. Time
52, 11
50, 11
54, 11
56, 11
Energy
52, 11
50, 11
54, 11
56, 11
parfib 1.2 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
52, 11
50, 11
54, 11
56, 11
Energy
52, 11
50, 11
54, 11
56, 11
parfib 1.4 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
52, 11
50, 11
54, 11
56, 11
parfib 1.8 GHz
Energy
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
52, 11
50, 11
54, 11
56, 11
parfib 2.1 GHz
Energy
Energy (j)
Exec. Time (seconds)
Cores

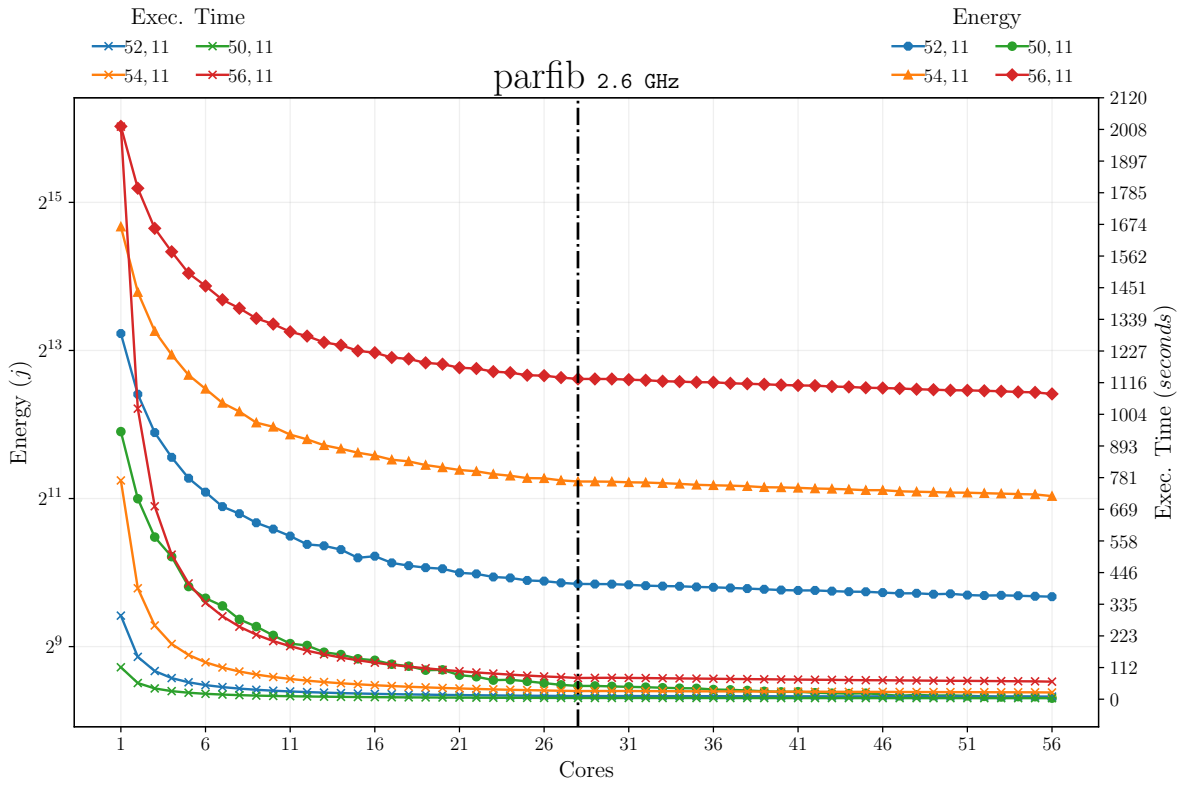
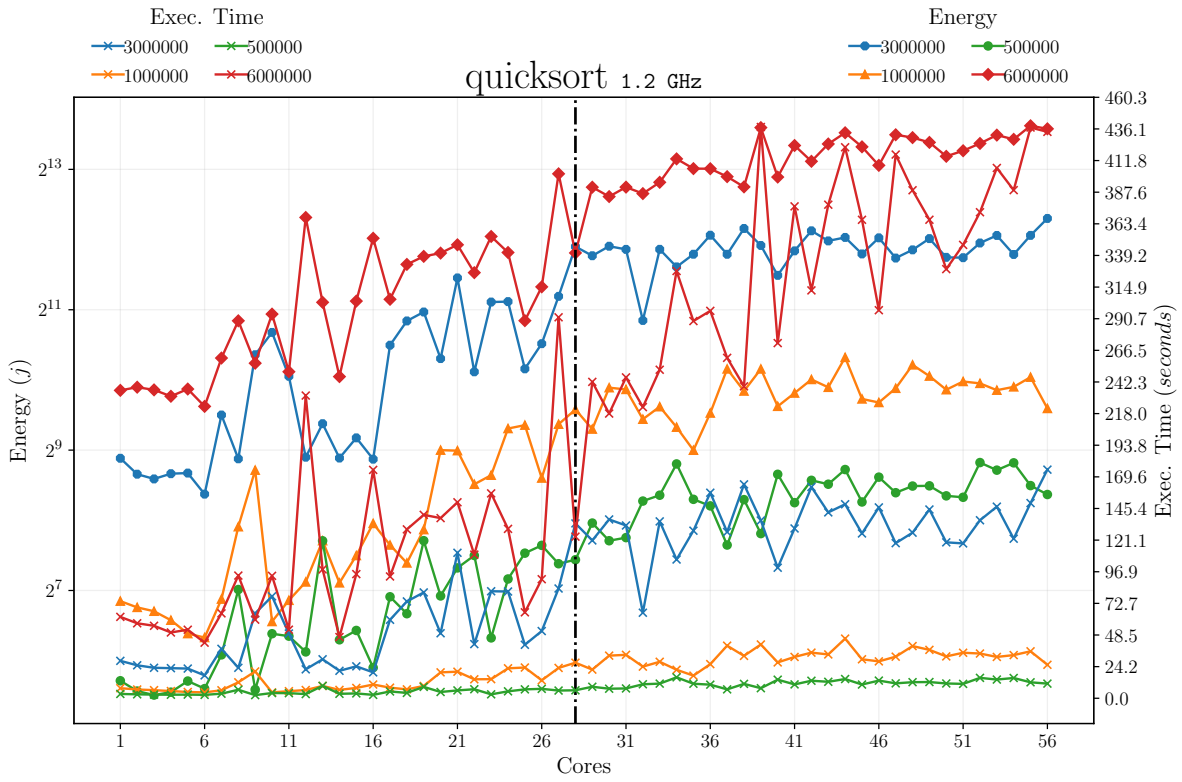
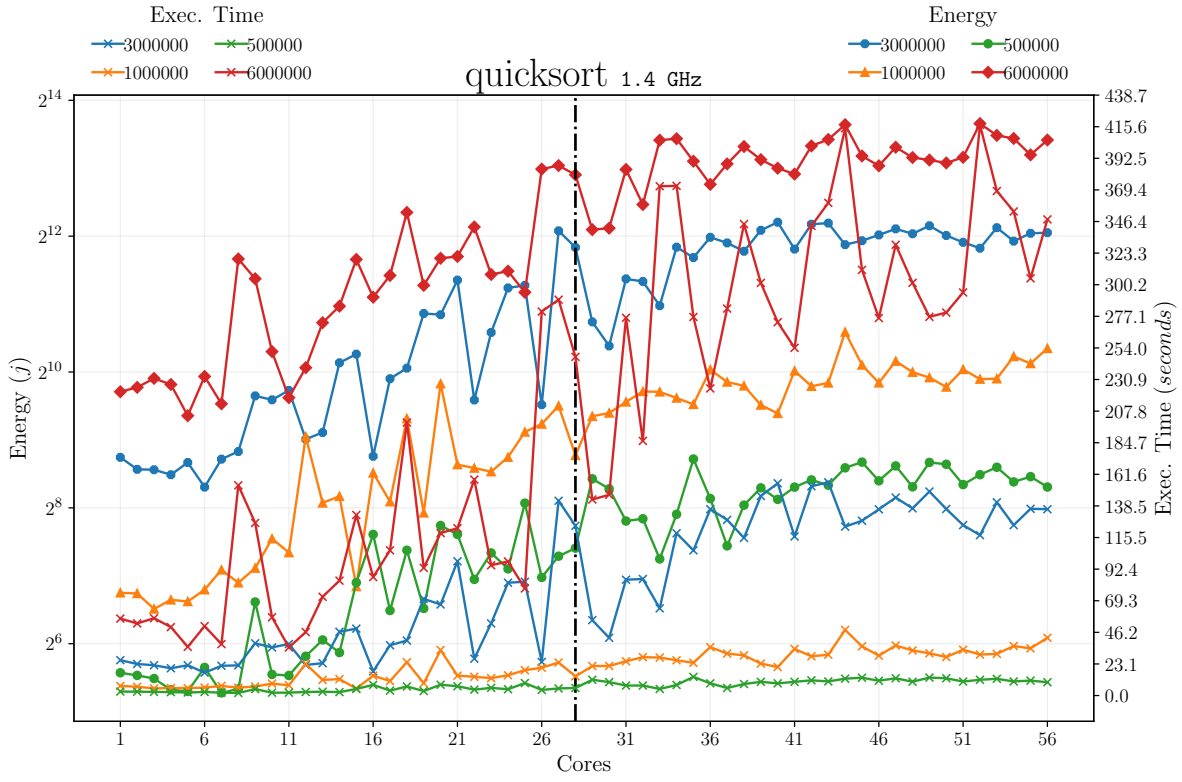
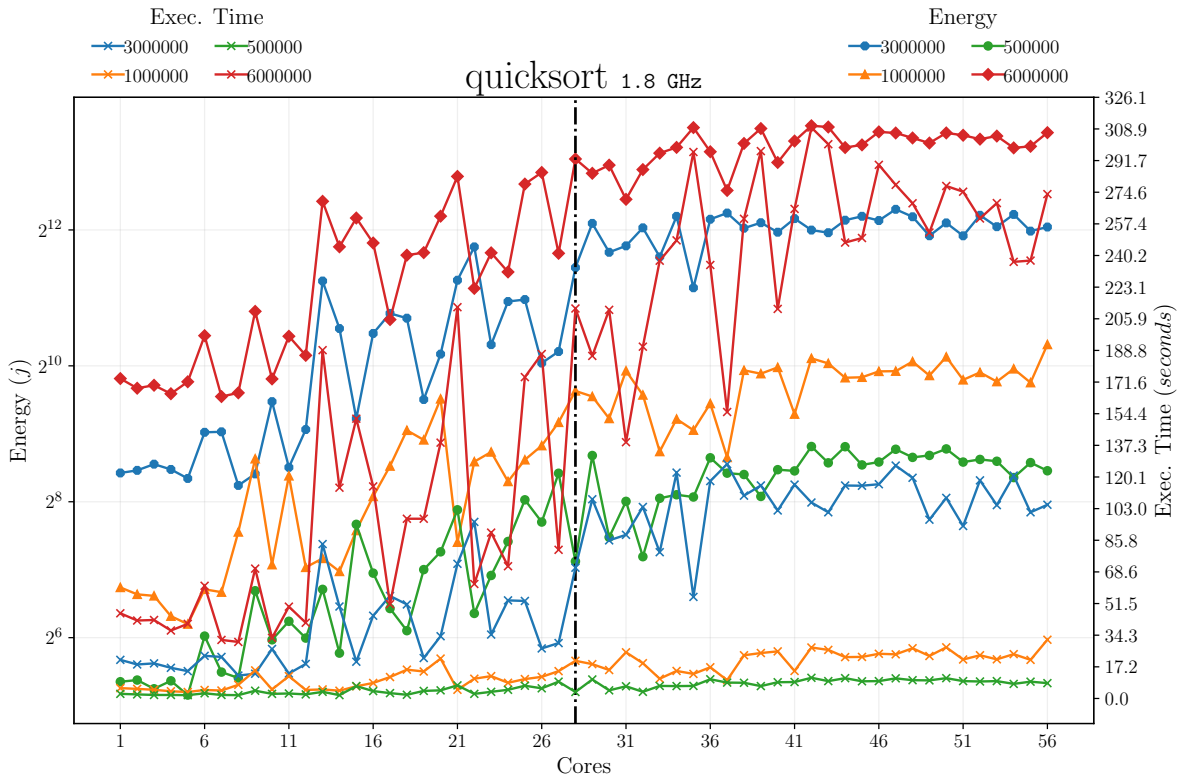
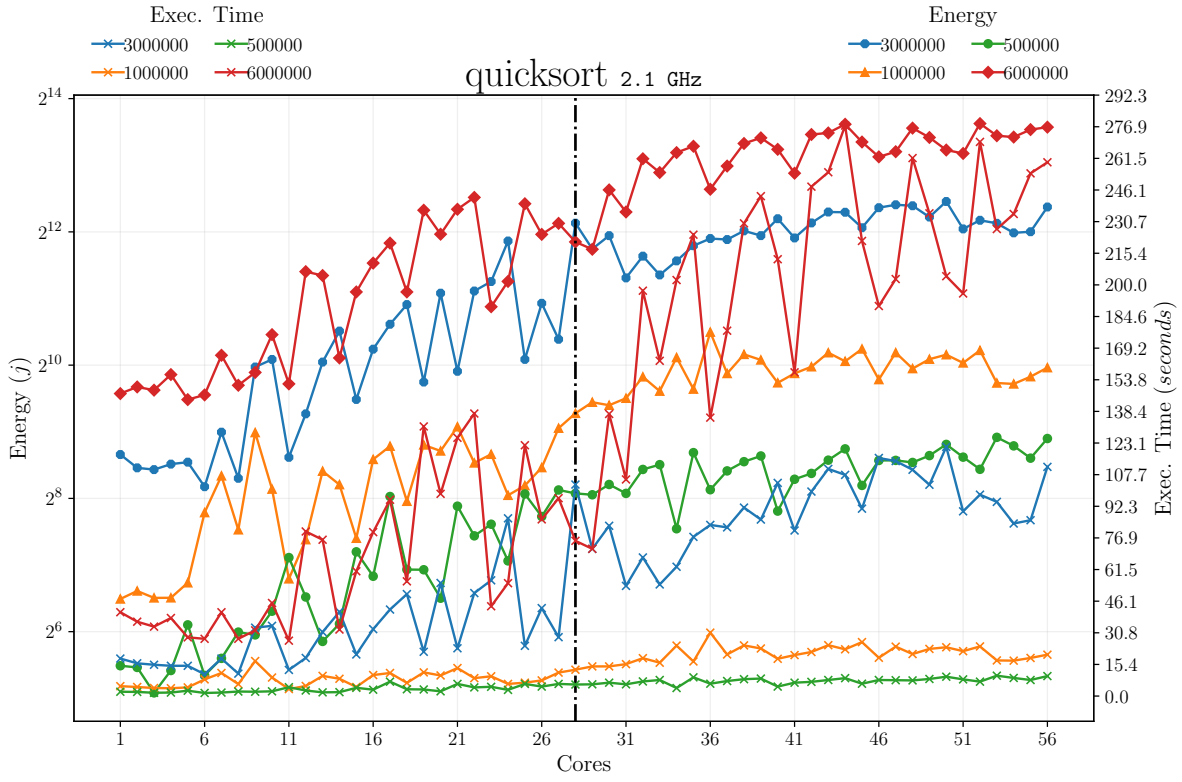


Figure B.6: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **Parfib**


Exec. Time
3000000
500000
1000000
6000000
Energy
3000000
500000
1000000
6000000
quicksort 1.2 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
3000000
500000
1000000
6000000
Energy
3000000
500000
1000000
6000000
quicksort 1.4 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
3000000
500000
1000000
6000000
quicksort 1.8 GHz
Energy
3000000
500000
1000000
6000000
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
3000000
500000
1000000
6000000
quicksort 2.1 GHz
Energy
3000000
500000
1000000
6000000
Energy (j)
Exec. Time (seconds)
Cores

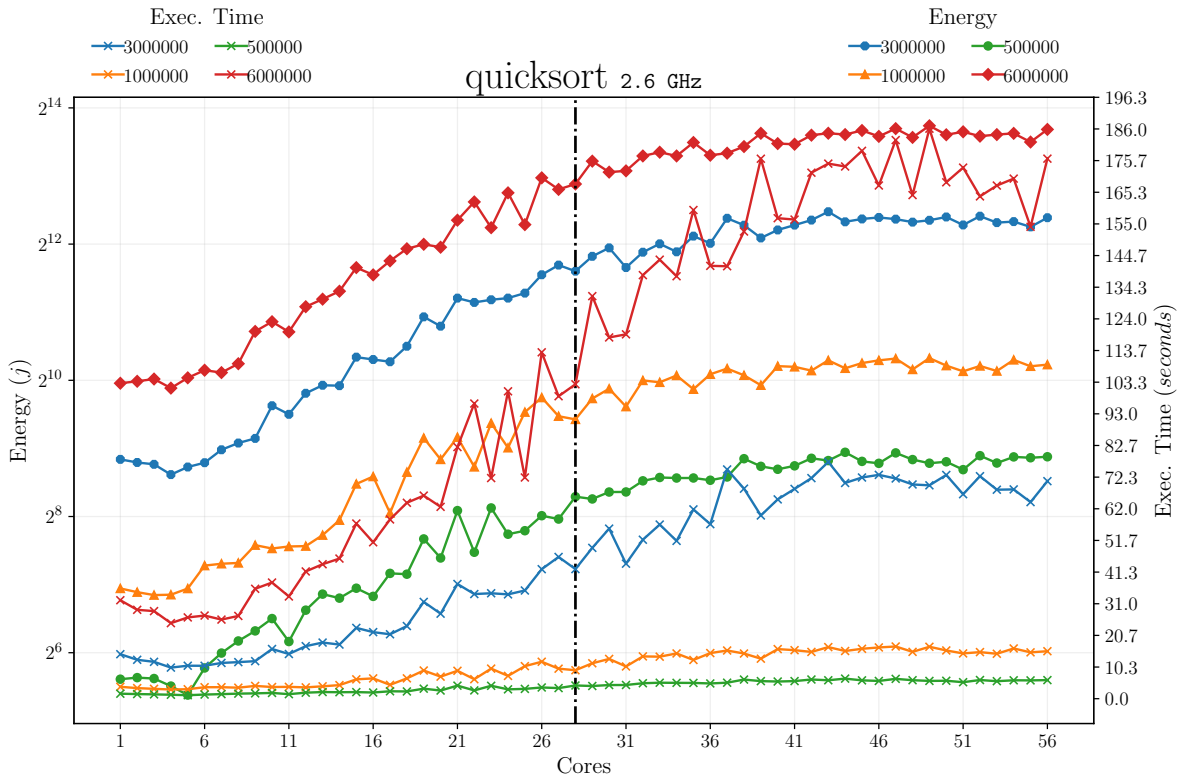
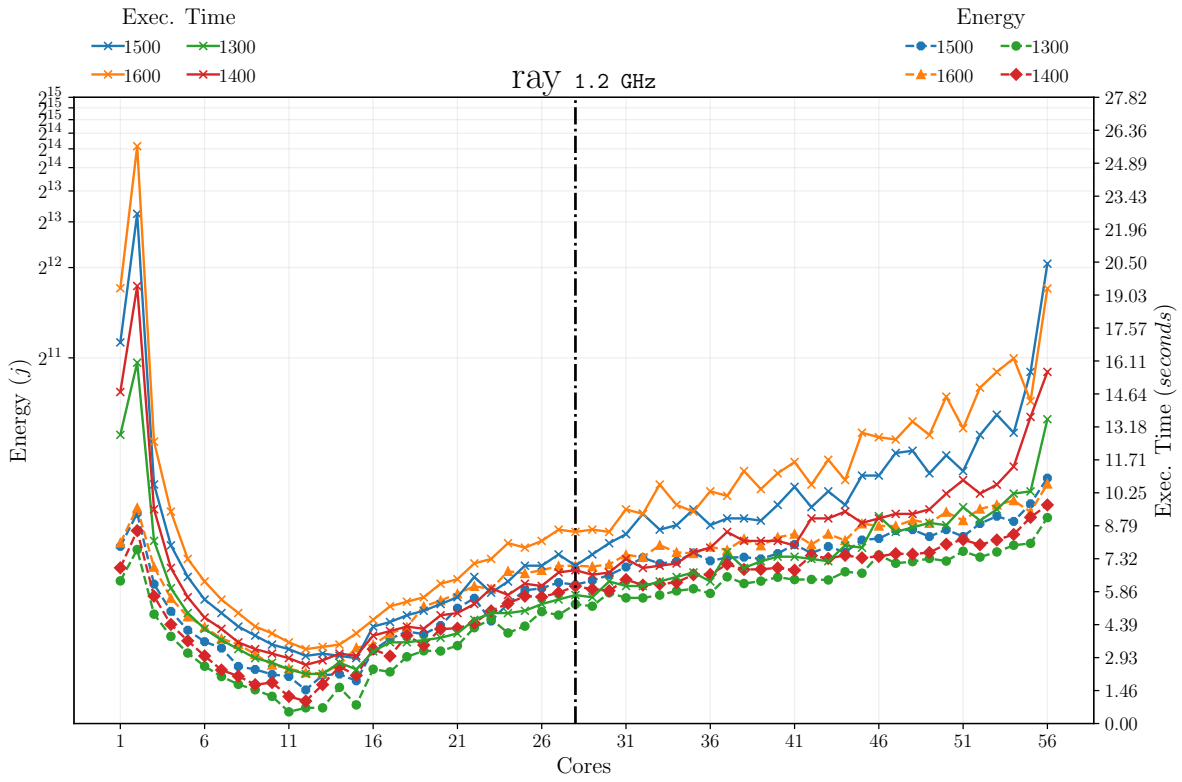
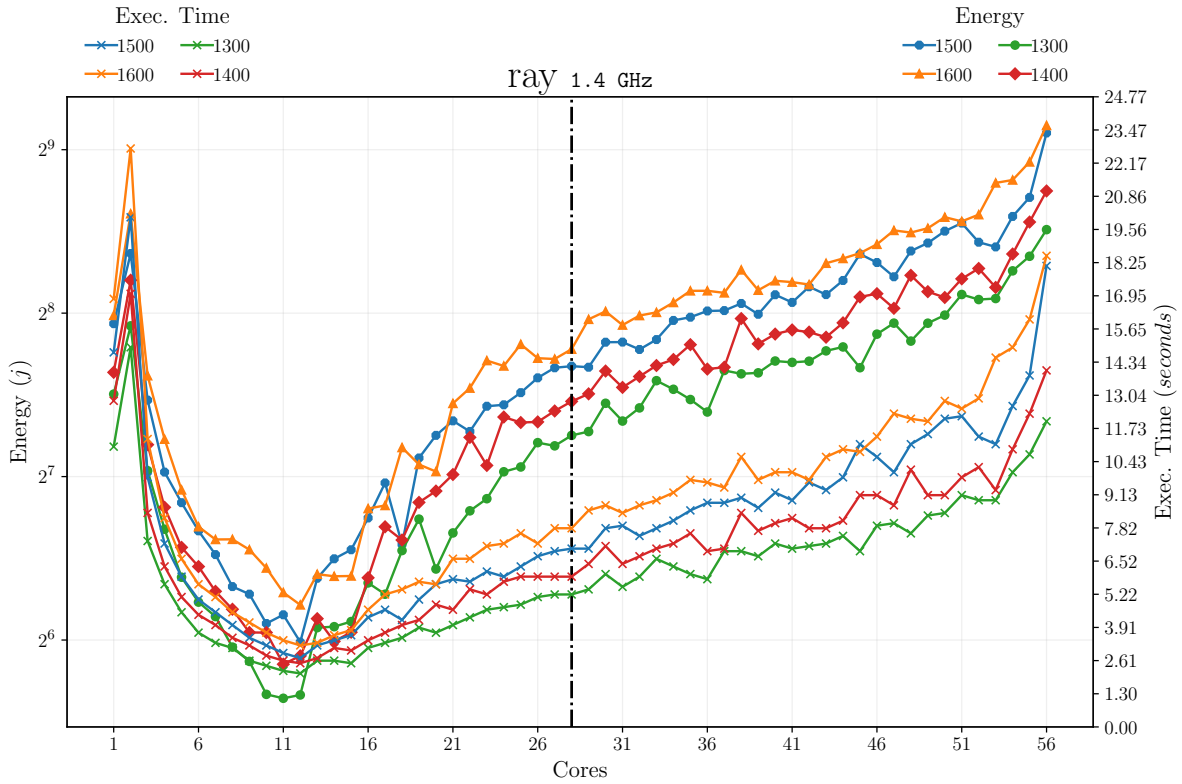
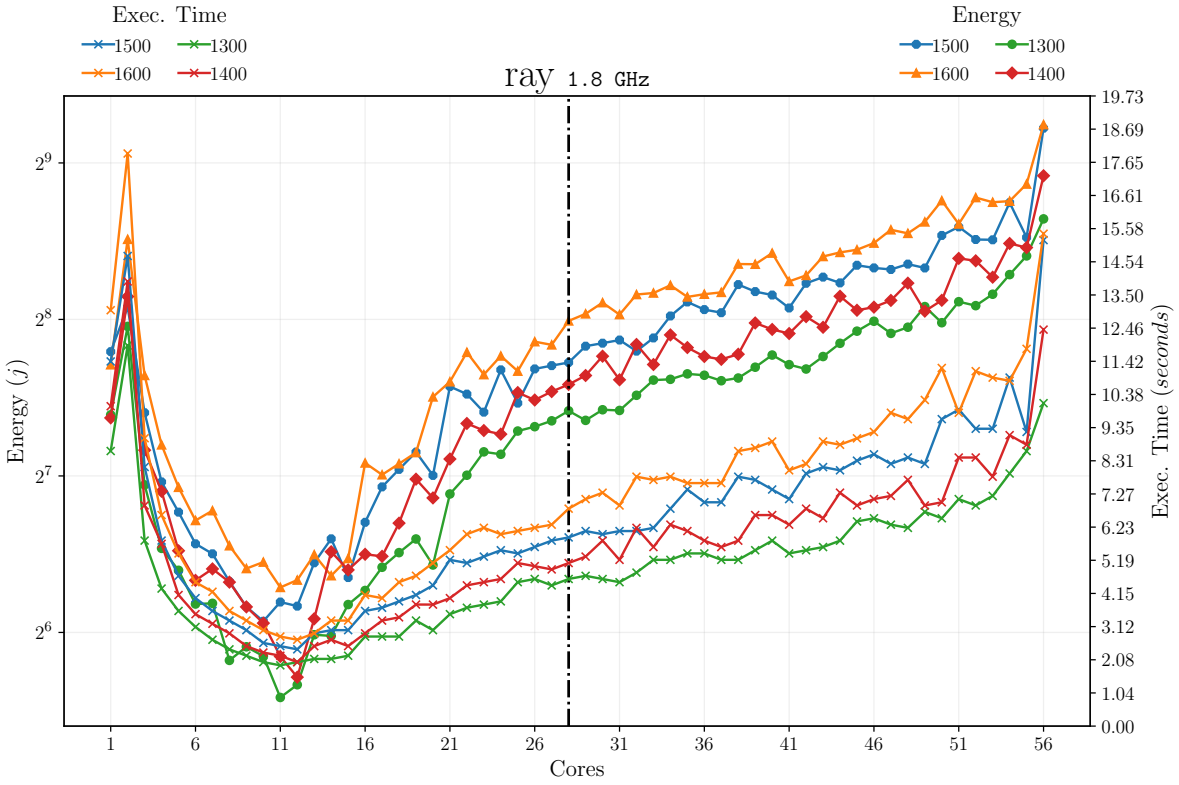
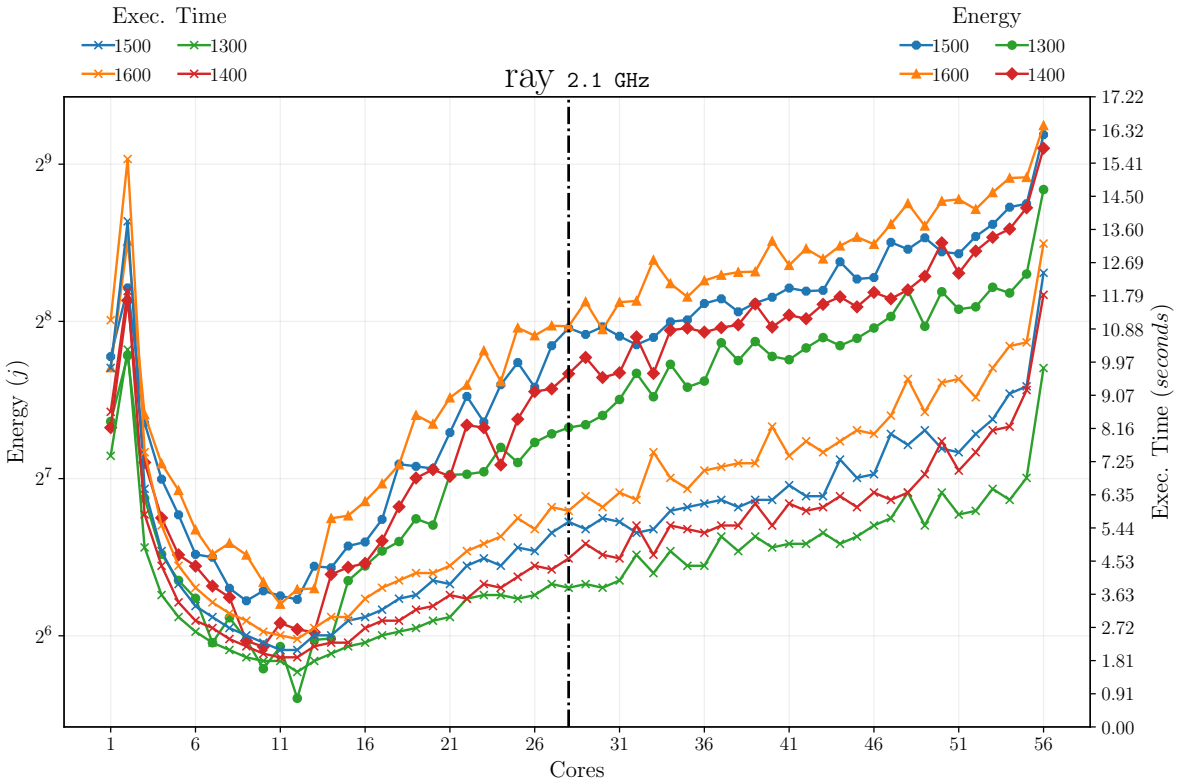


Figure B.7: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **Quicksort**


Exec. Time
1500
1300
1600
1400
Energy
1500
1300
1600
1400
ray 1.2 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
1500
1300
1600
1400
Energy
1500
1300
1600
1400
ray 1.4 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
1500
1300
1600
1400
Energy
1500
1300
1600
1400
ray 1.8 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
1500
1300
1600
1400
Energy
1500
1300
1600
1400
ray 2.1 GHz
Energy (j)
Exec. Time (seconds)
Cores

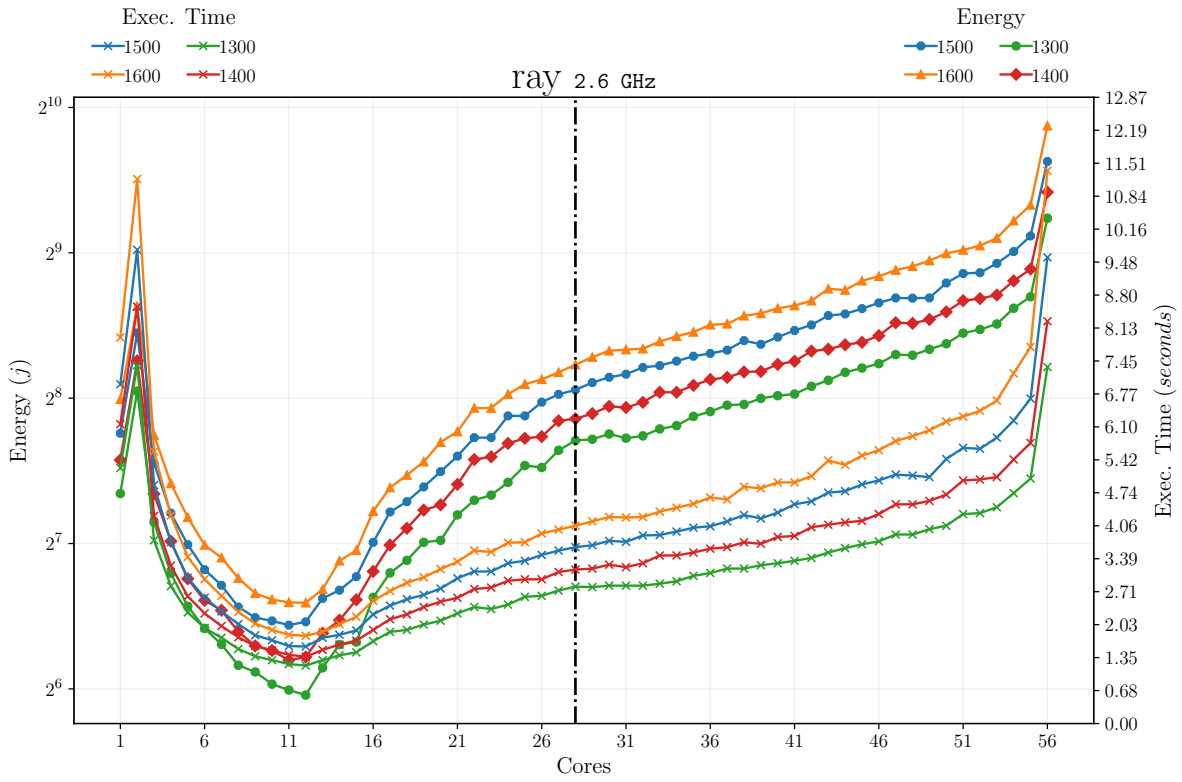
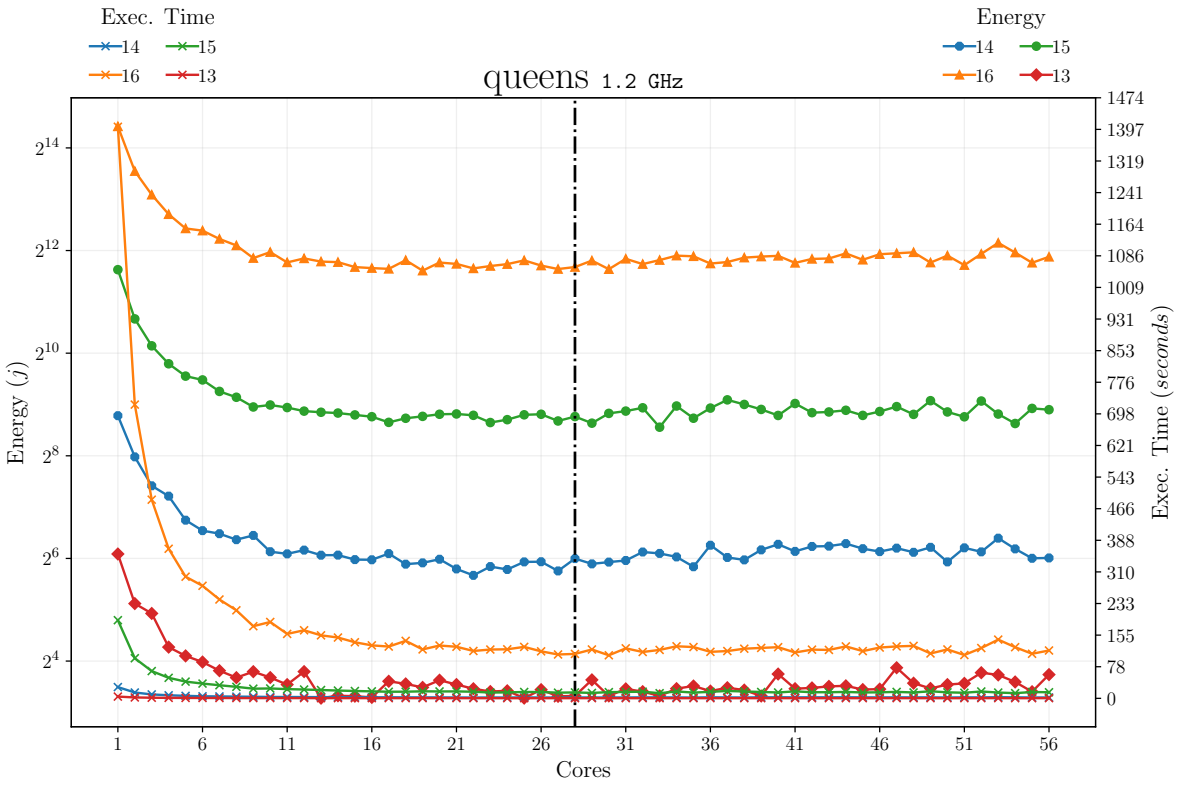
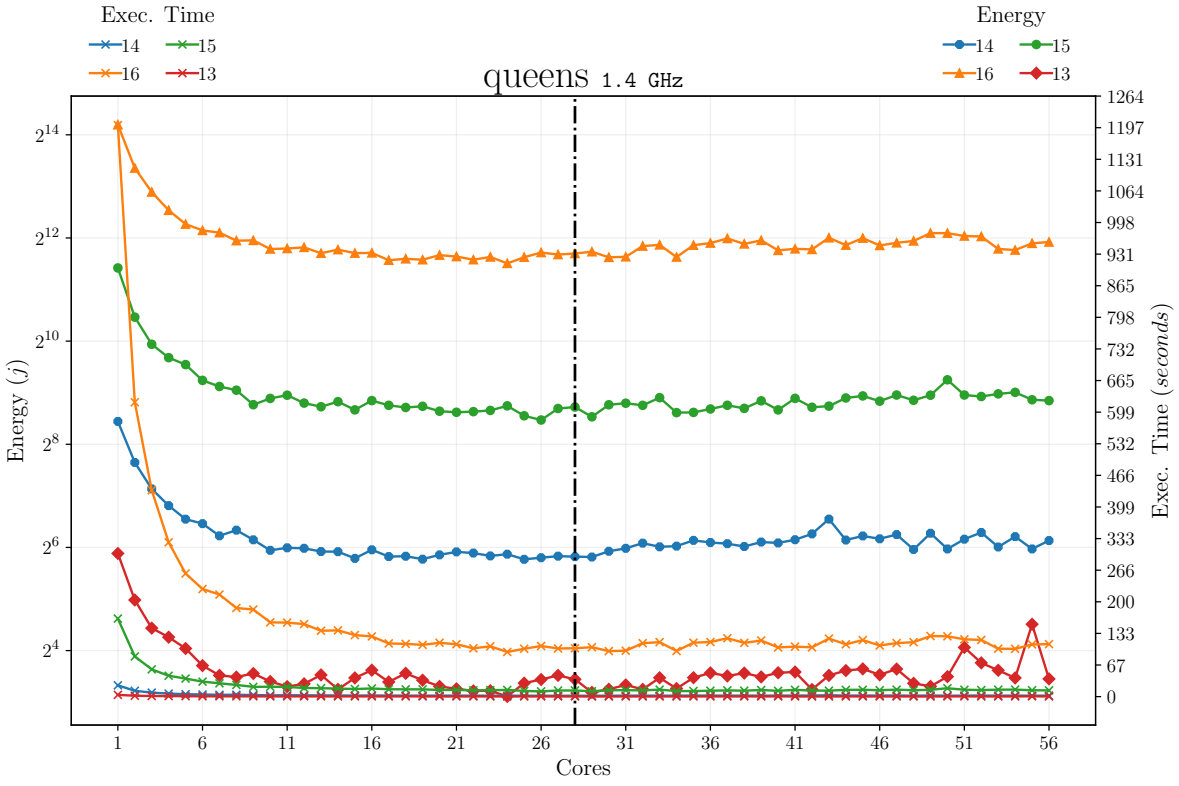
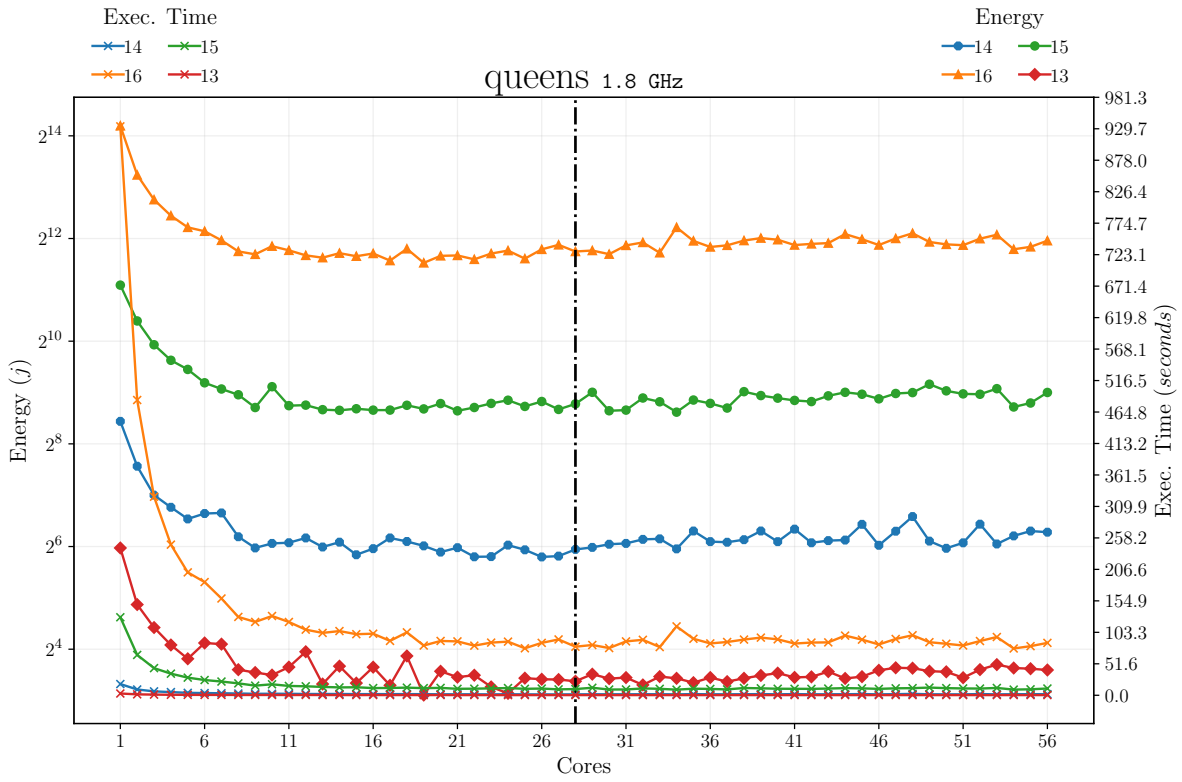
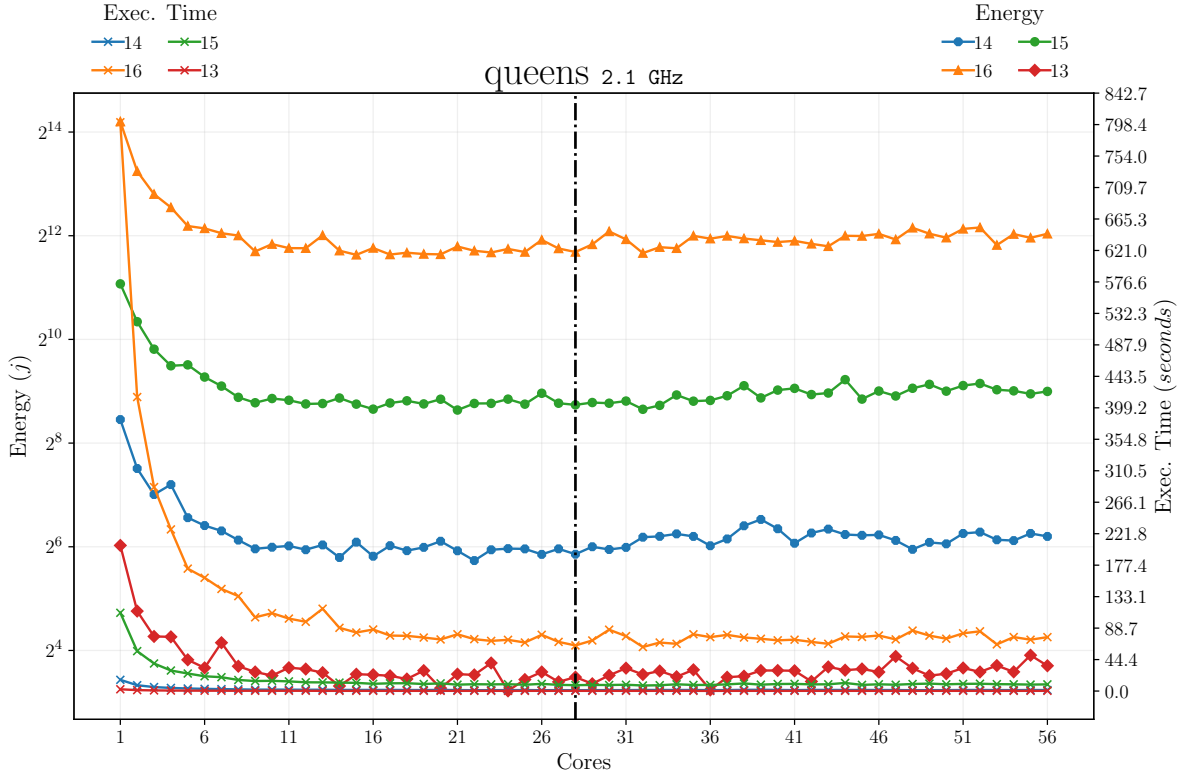


Figure B.8: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **Ray Tracing**


Exec. Time
14
15
16
13
Energy
14
15
16
13
queens 1.2 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
14
15
16
13
Energy
14
15
16
13
queens 1.4 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
14
15
16
13
Energy
14
15
16
13
queens 1.8 GHz
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
14
15
16
13
Energy
14
15
16
13
queens 2.1 GHz
Energy (j)
Exec. Time (seconds)
Cores

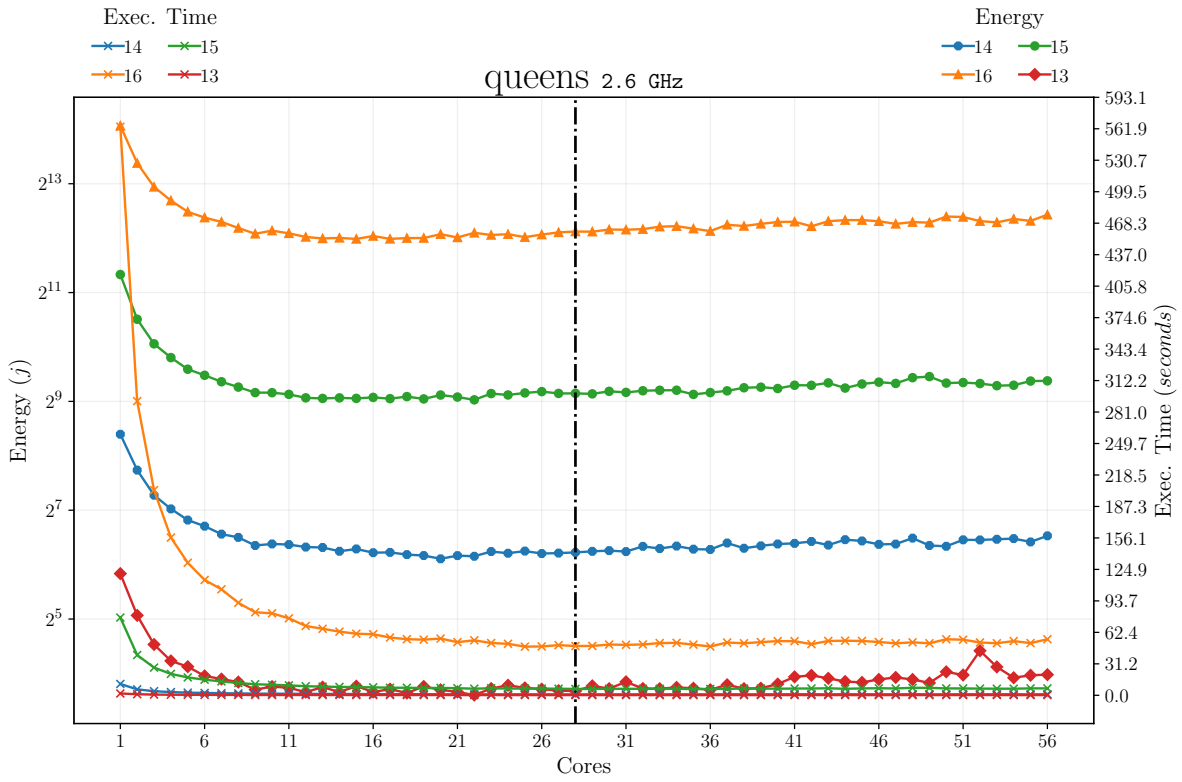
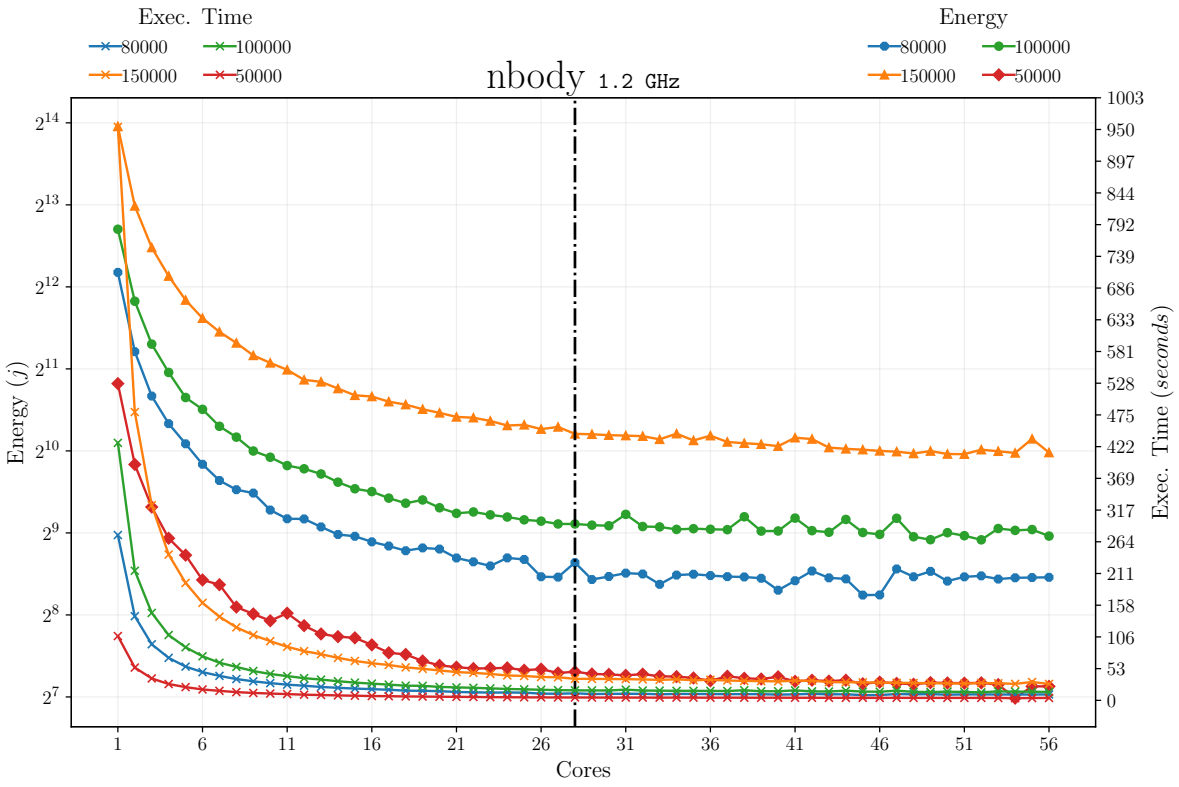
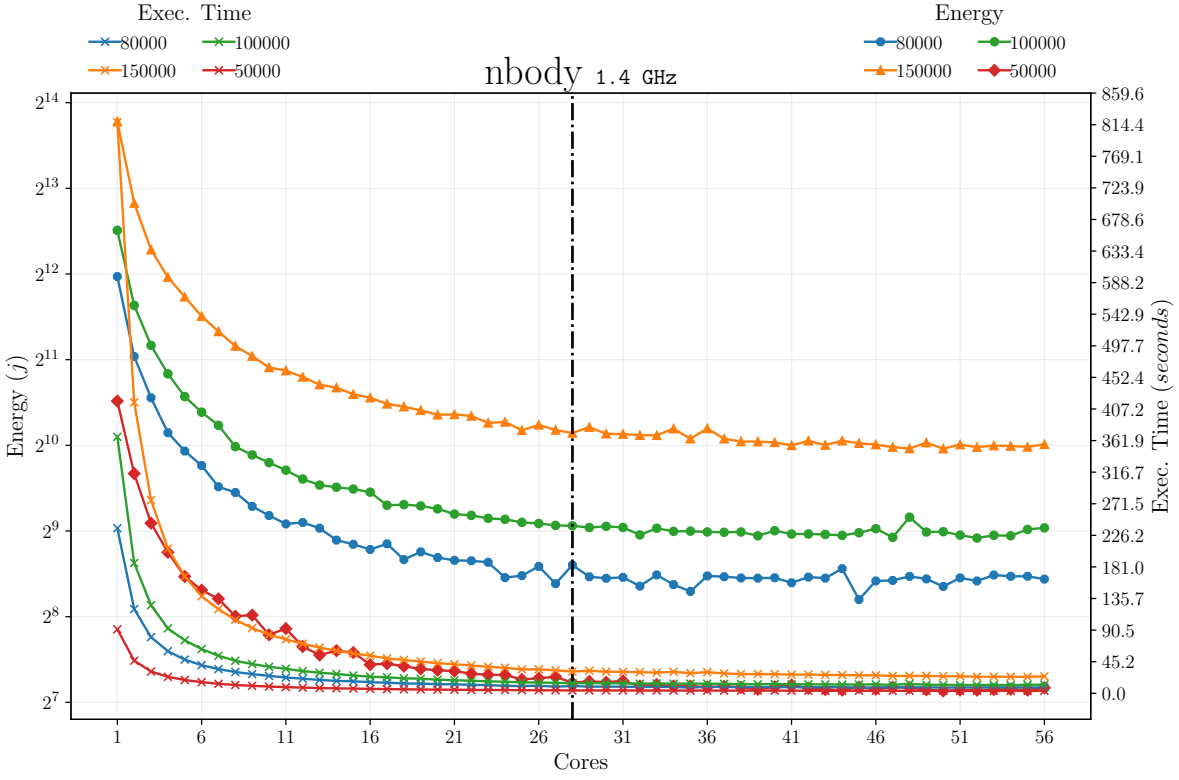
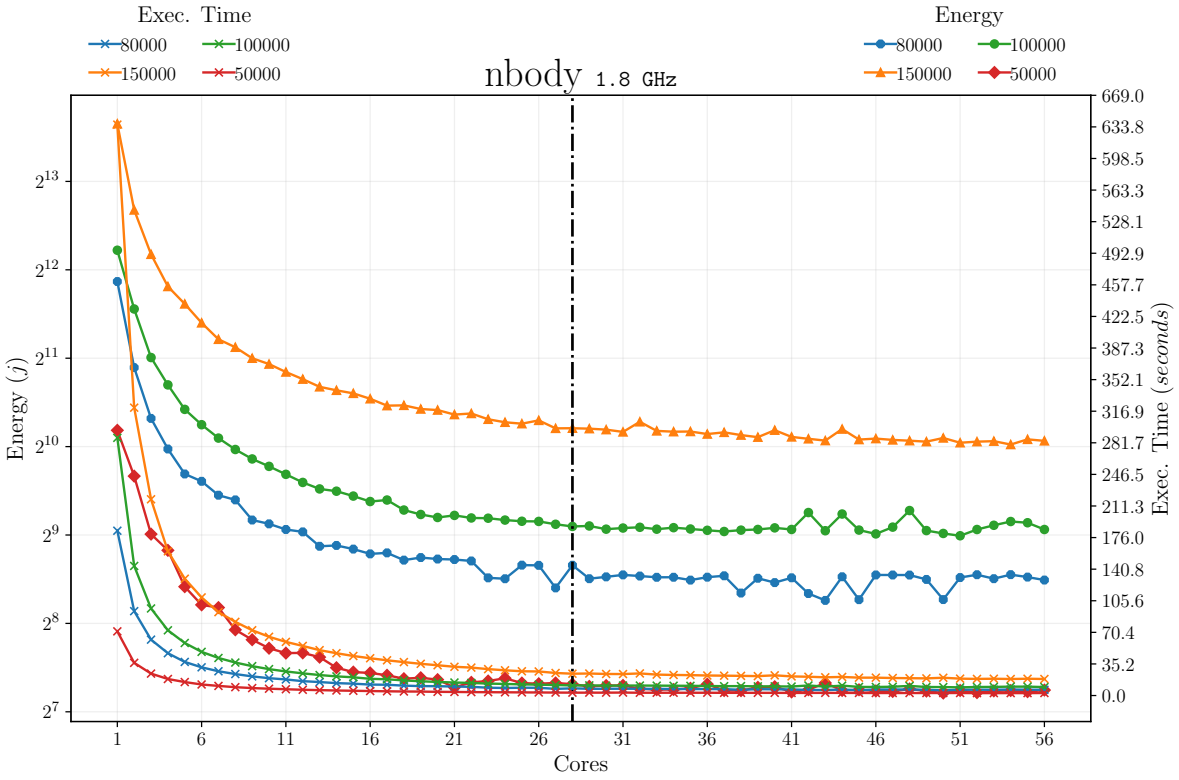
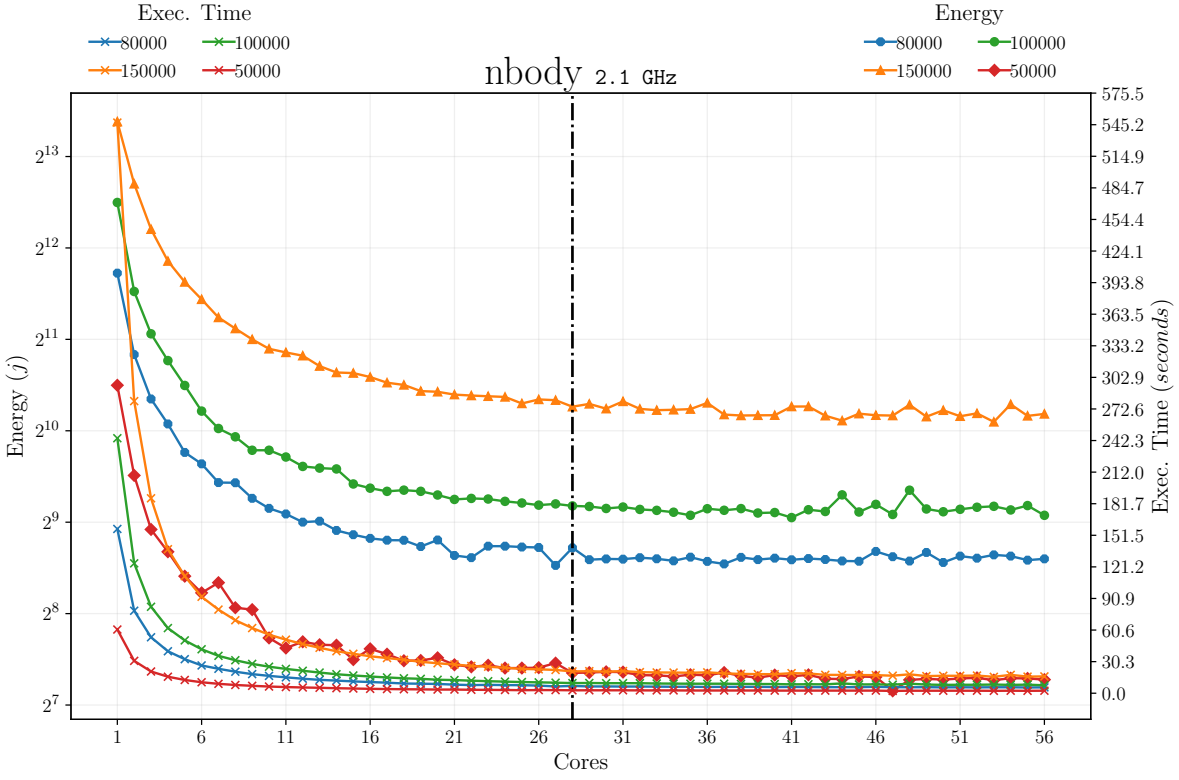


Figure B.9: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **Queens**


Exec. Time
80000
100000
150000
50000
nbody 1.2 GHz
Energy
80000
100000
150000
50000
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
80000
100000
150000
50000
nbody 1.4 GHz
Energy
80000
100000
150000
50000
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
80000
100000
150000
50000
nbody 1.8 GHz
Energy
80000
100000
150000
50000
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
80000
100000
150000
50000
nbody 2.1 GHz
Energy
80000
100000
150000
50000
Energy (j)
Exec. Time (seconds)
Cores

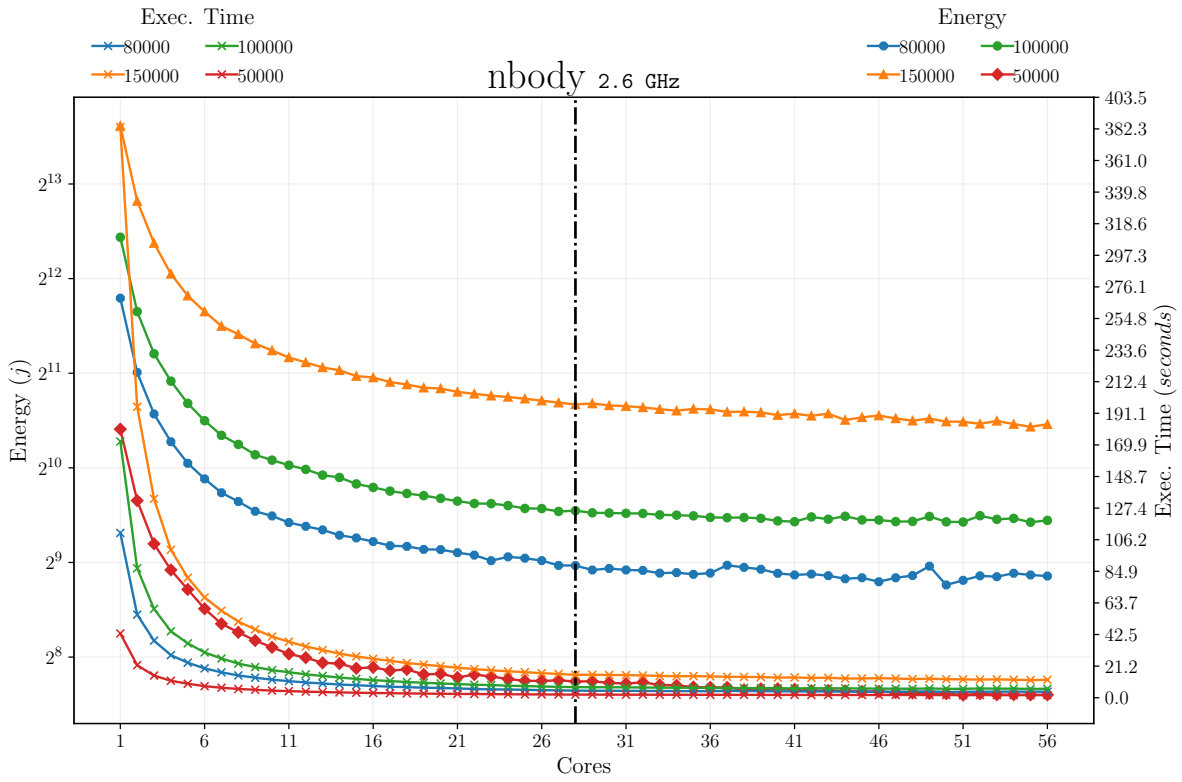
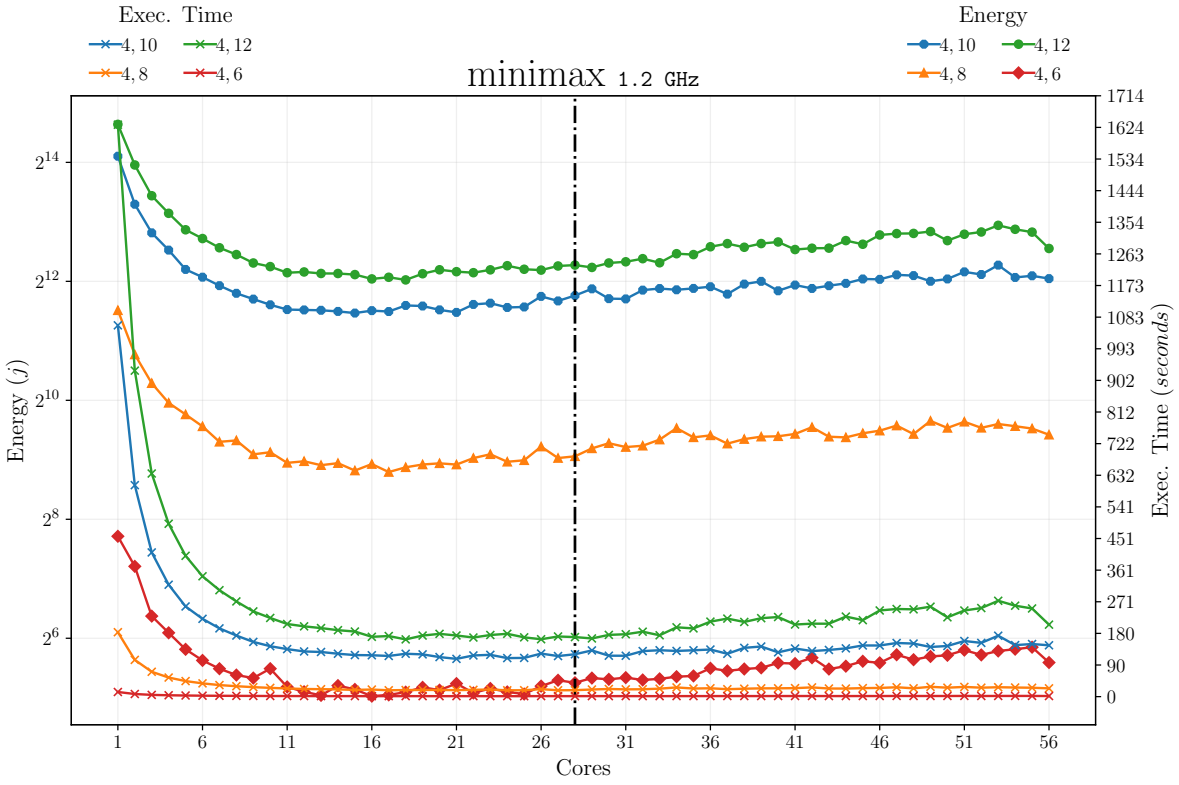
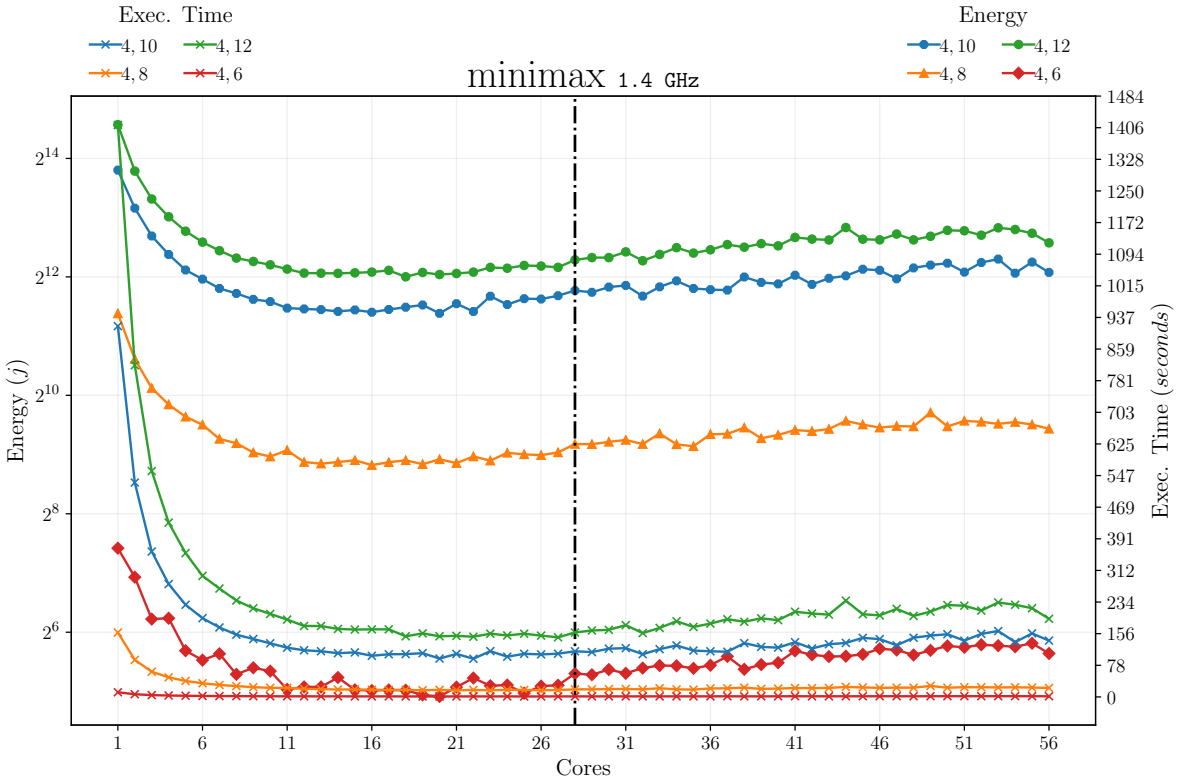
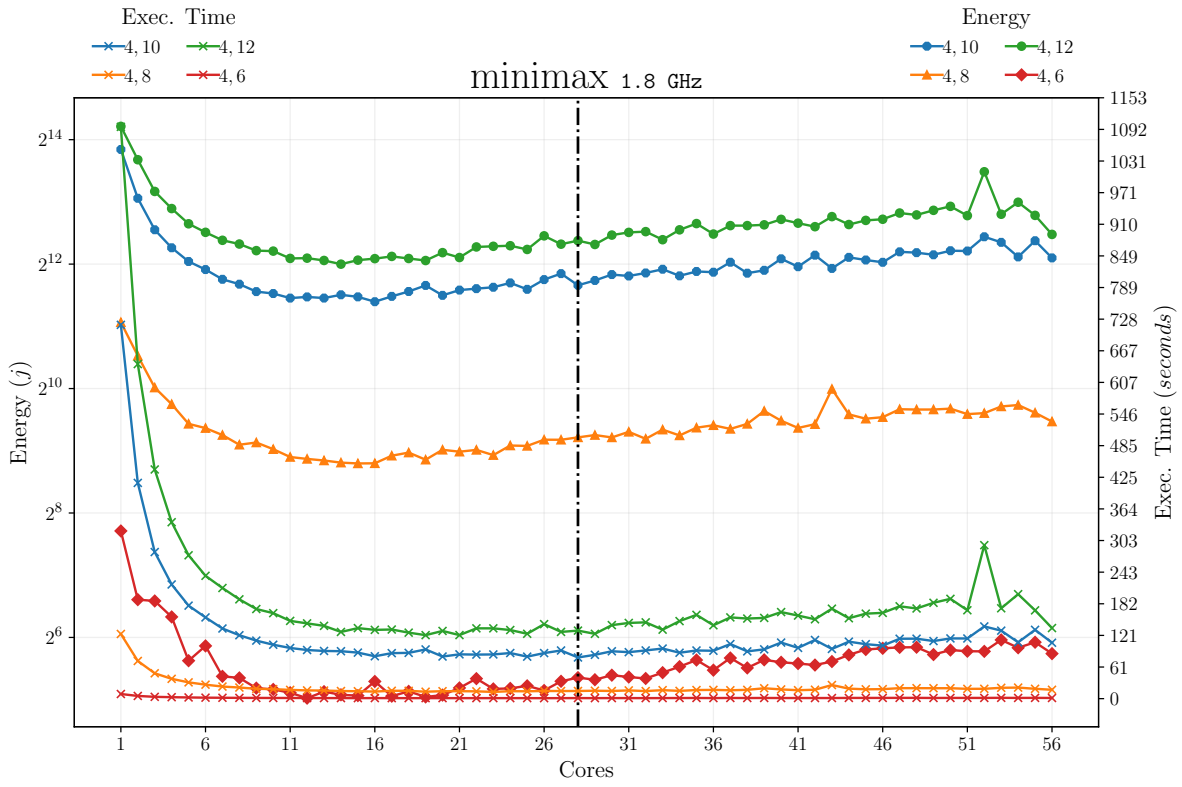
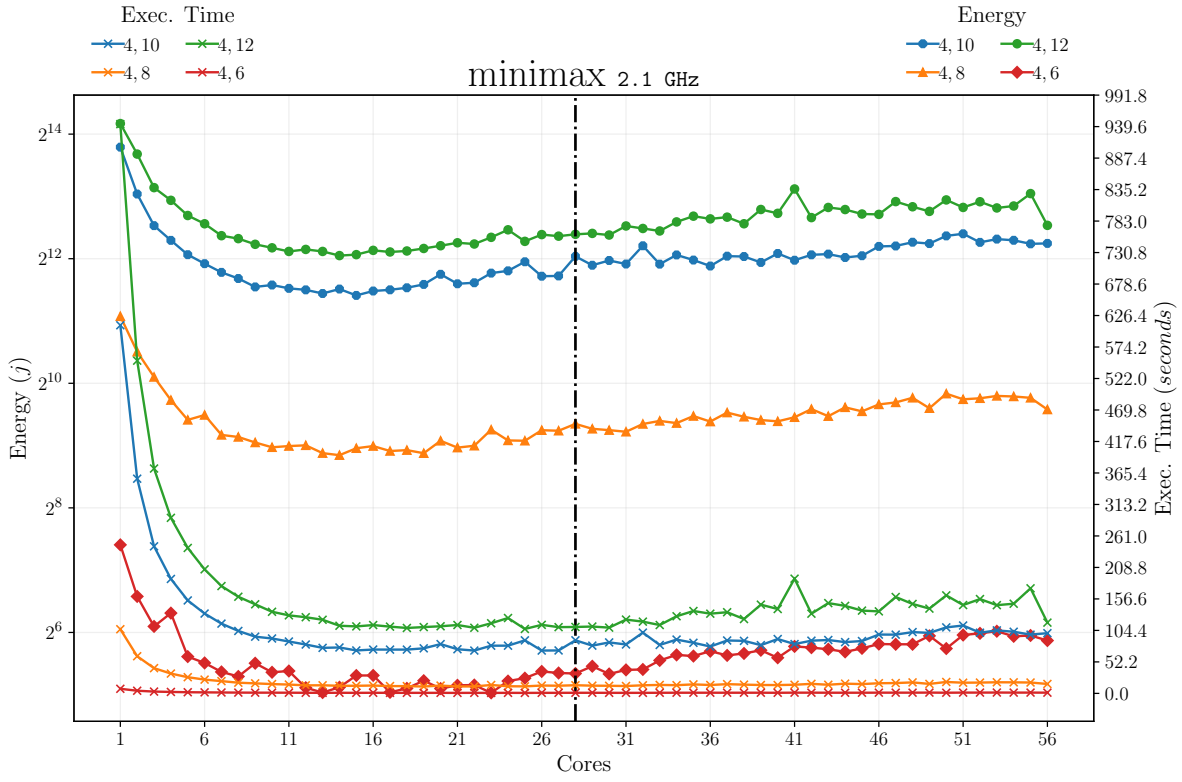


Figure B.10: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **Nbody**


Exec. Time
4, 10
4, 12
4, 8
4, 6
minimax 1.2 GHz
Energy
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
4, 10
4, 12
4, 8
4, 6
minimax 1.4 GHz
Energy
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
4, 10
4, 12
4, 8
4, 6
minimax 1.8 GHz
Energy
4, 10
4, 12
4, 8
4, 6
Energy (j)
Exec. Time (seconds)
Cores


Exec. Time
4, 10
4, 12
4, 8
4, 6
minimax 2.1 GHz
Energy
4, 10
4, 12
4, 8
4, 6
Energy (j)
Exec. Time (seconds)
Cores

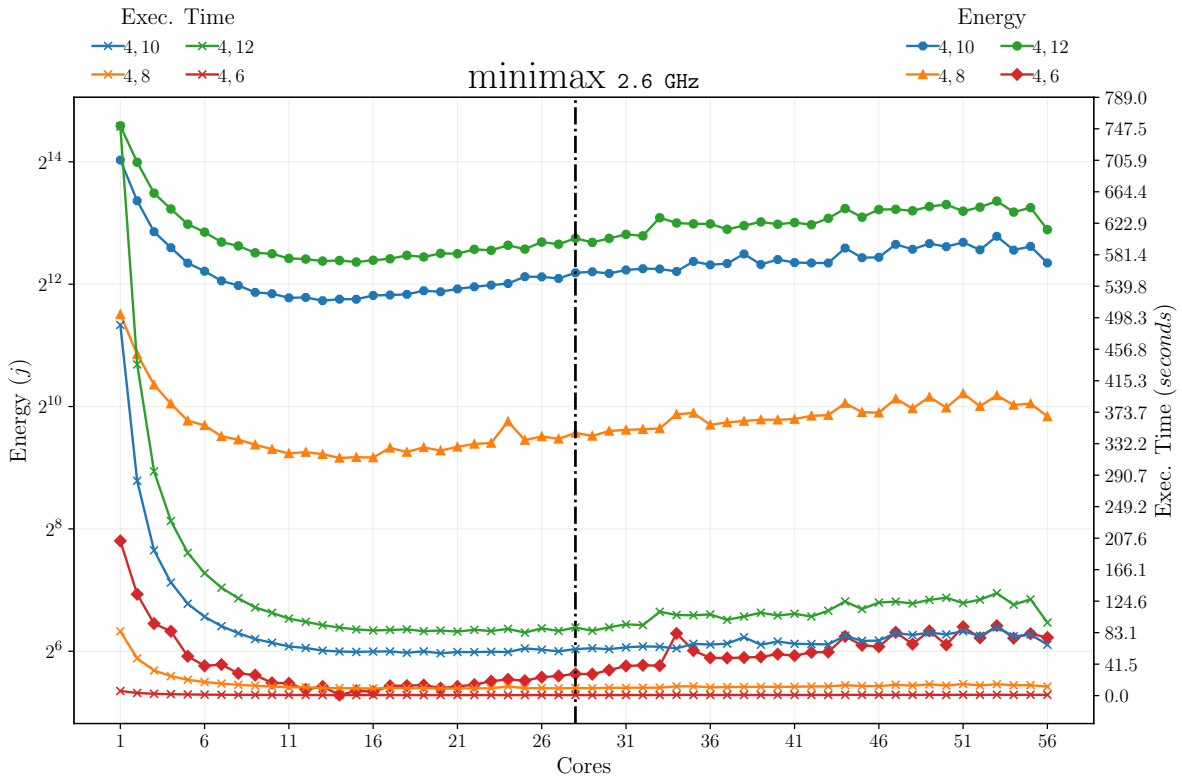
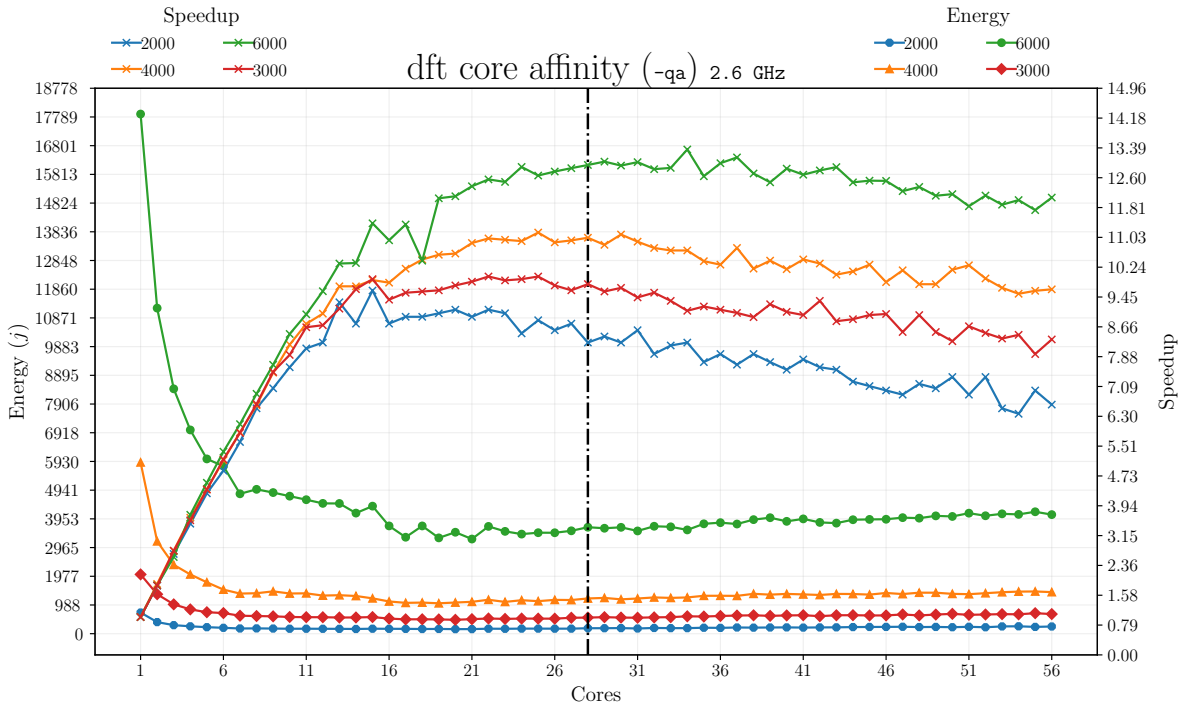
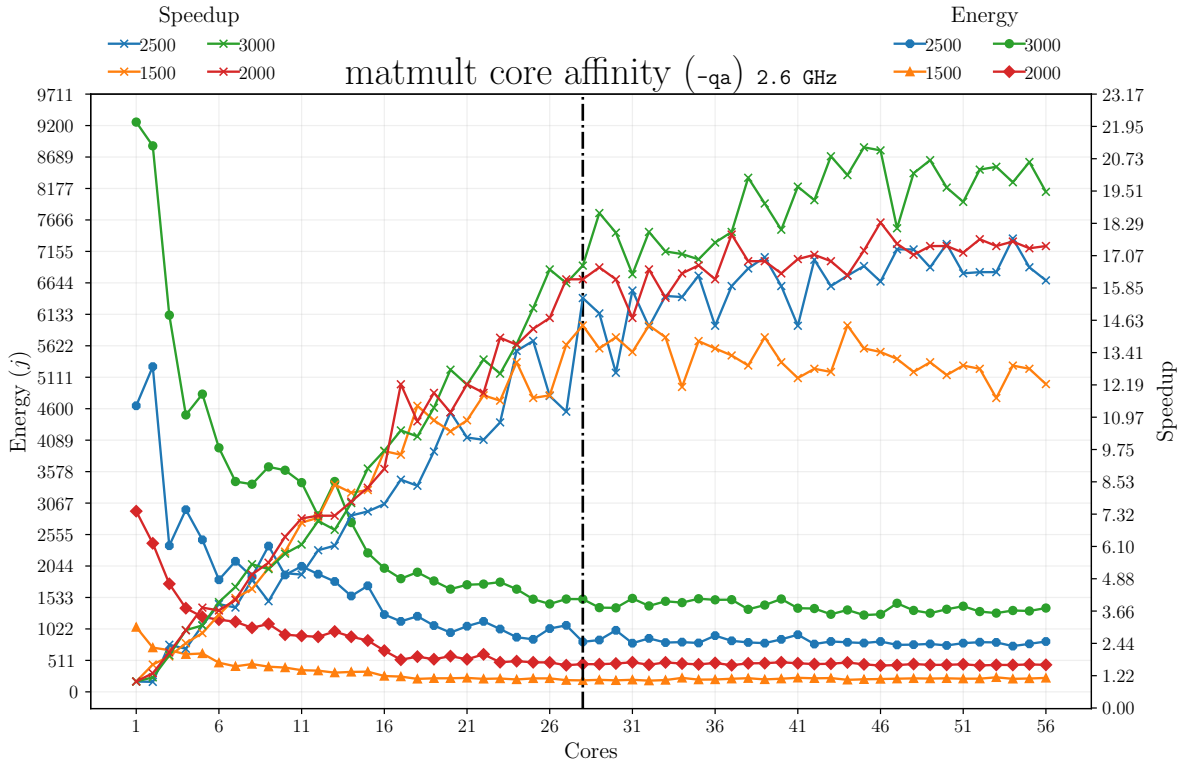


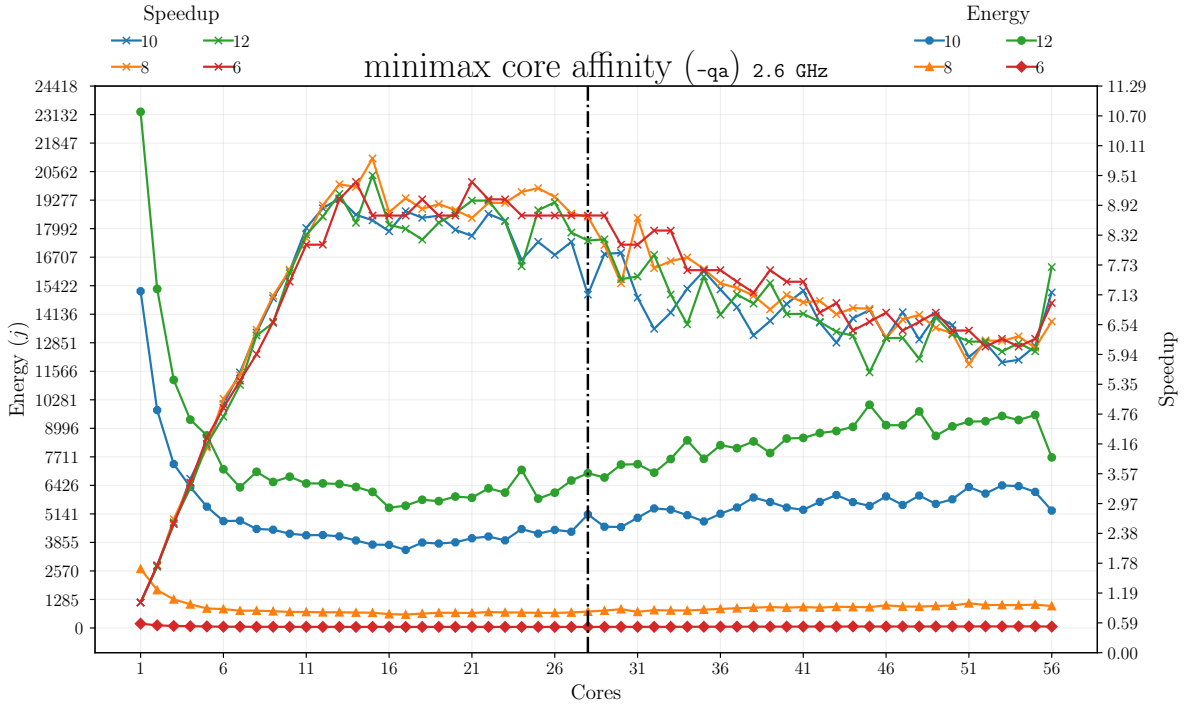
Figure B.11: Effect on energy usage of varying clock frequencies for: 1.2GHz, 1.4GHz, 1.8GHz, 2.1GHz, 2.6GHz for **Minimax**

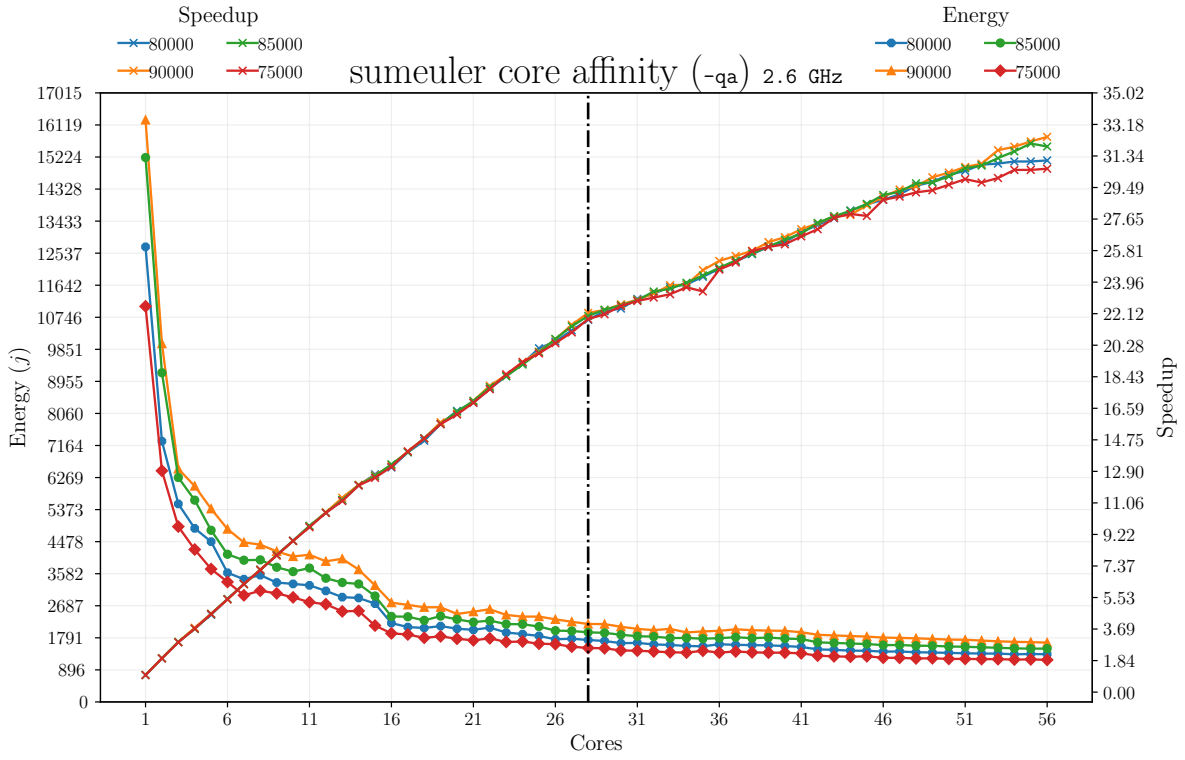
Appendix C

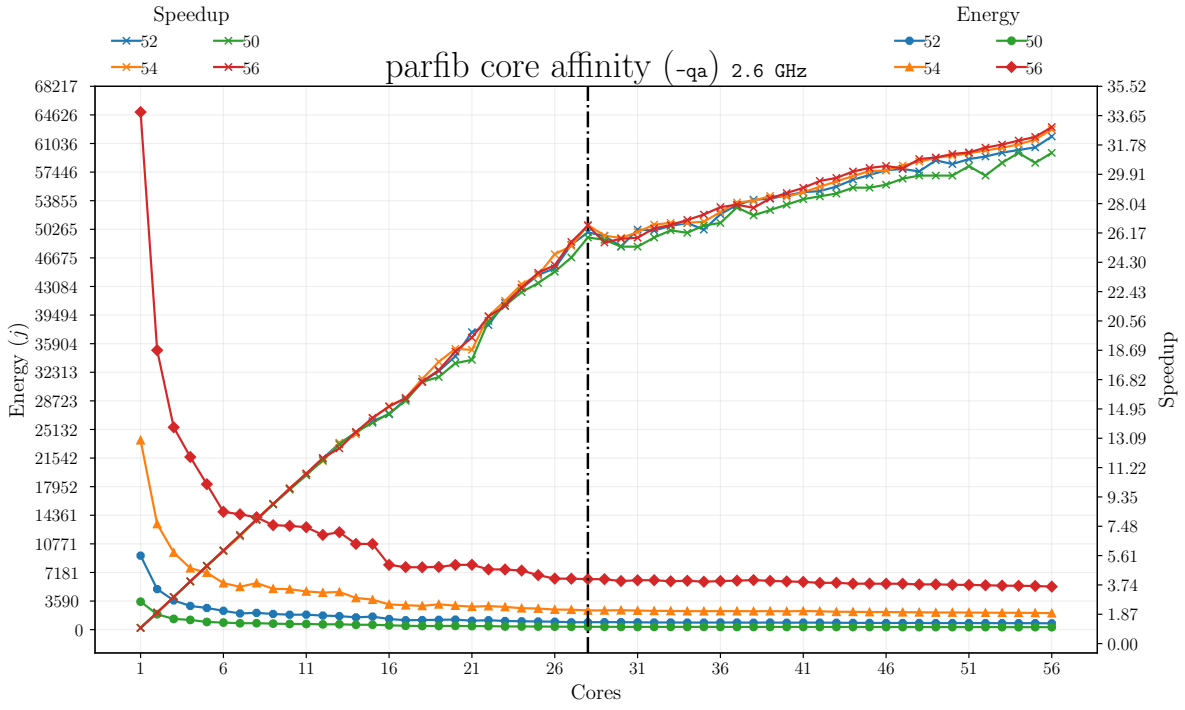
Core Affinity Plots

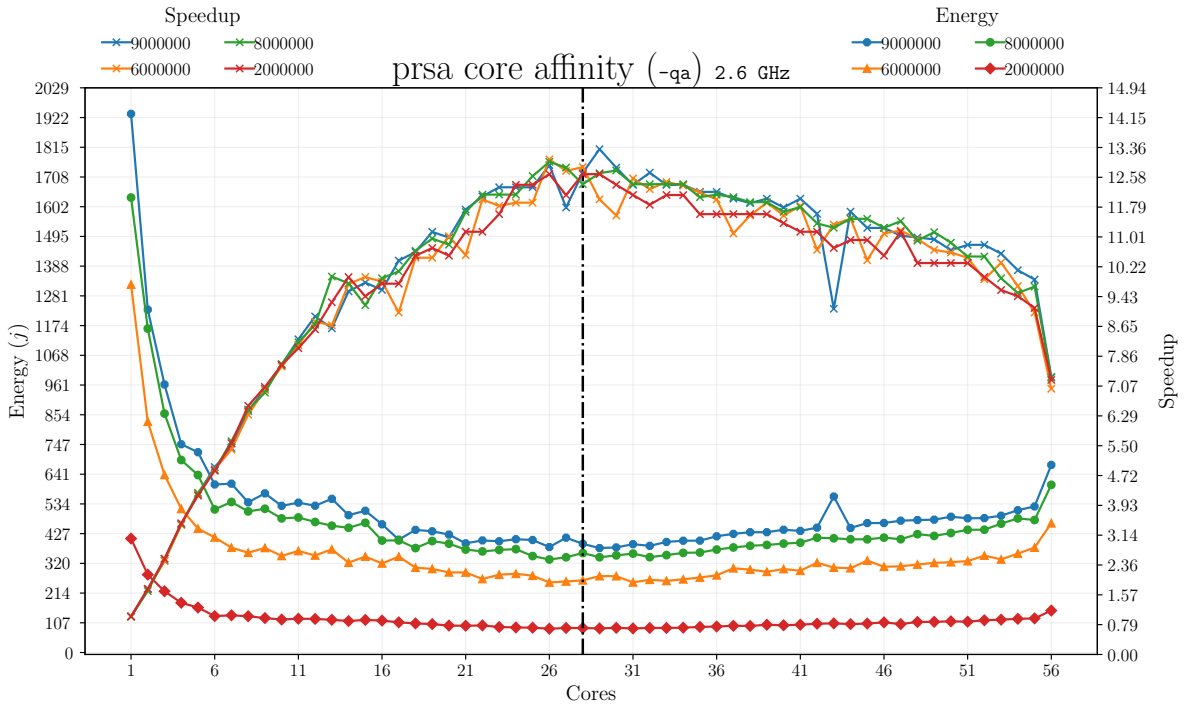


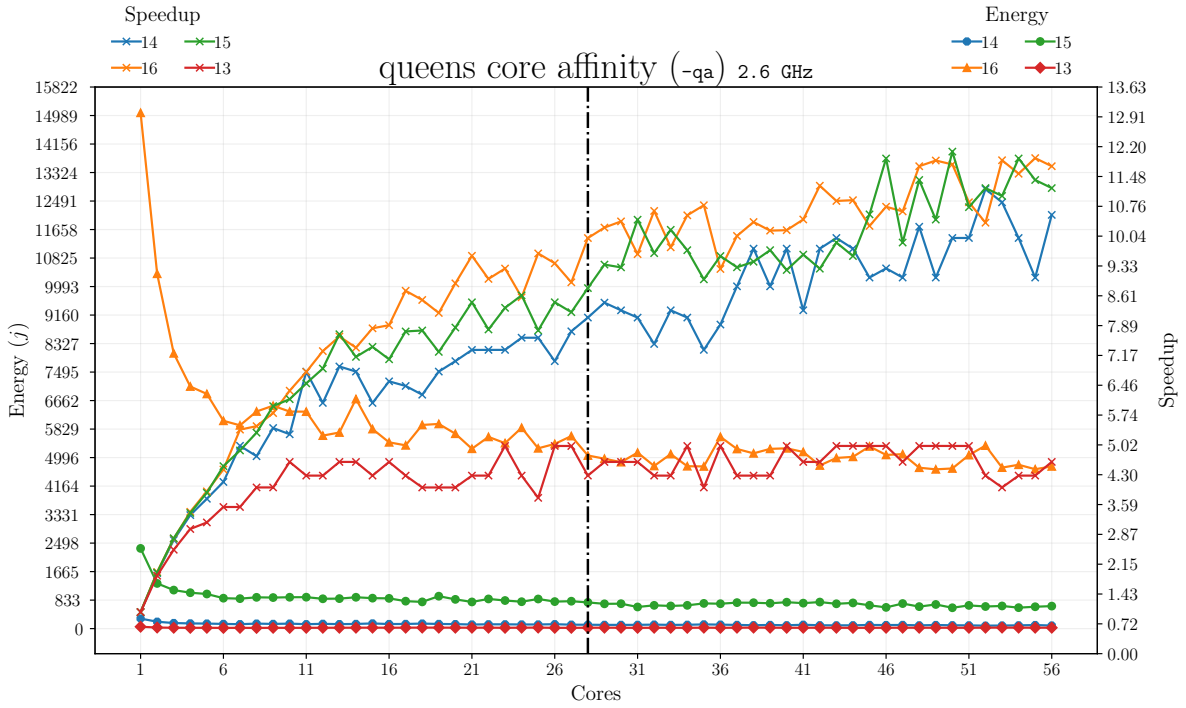

matmult core affinity (-qa) 2.6 GHz
Speedup
2500
3000
1500
2000
Energy
2500
3000
1500
2000
Energy (j)
Speedup
Cores
9711
9200
8689
8177
7666
7155
6644
6133
5622
5111
4600
4089
3578
3067
2555
2044
1533
1022
511
0
23.17
21.95
20.73
19.51
18.29
17.07
15.85
14.63
13.41
12.19
10.97
9.75
8.53
7.32
6.10
4.88
3.66
2.44
1.22
0.00
1
6
11
16
21
26
31
36
41
46
51
56

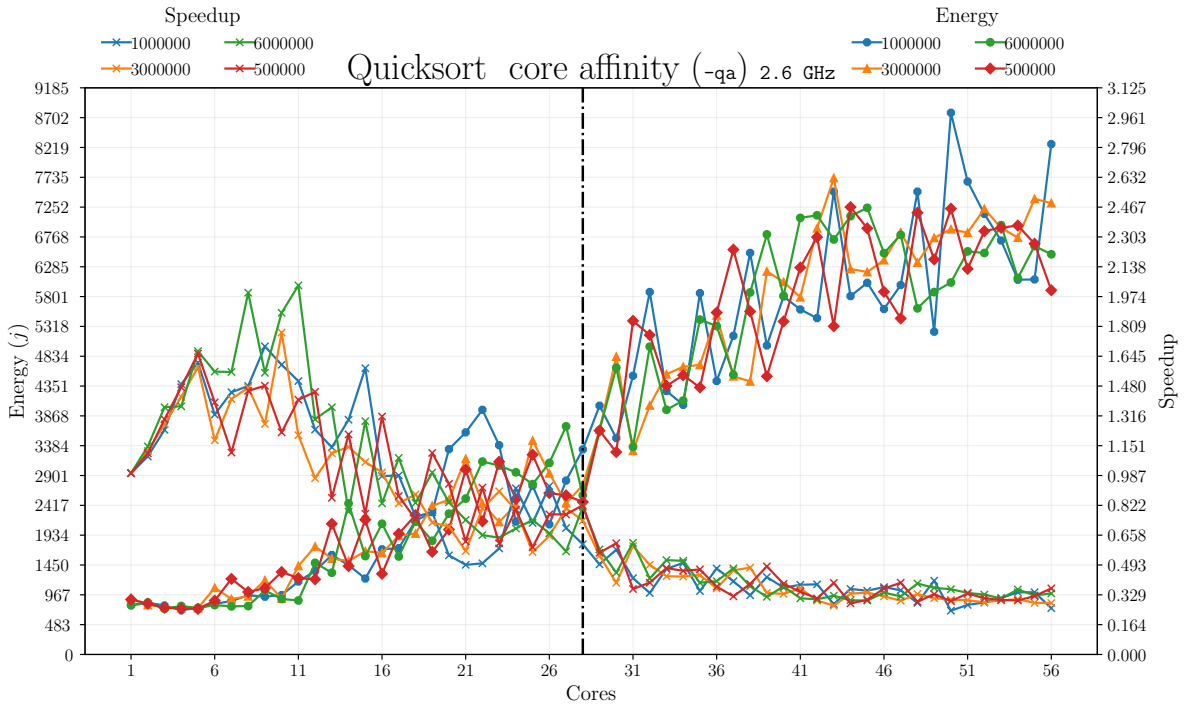

minimax core affinity (-qa) 2.6 GHz
Speedup
10
12
8
6
Energy
10
12
8
6
Energy (j)
Speedup
Cores
24418
23132
21847
20562
19277
17992
16707
15422
14136
12851
11566
10281
8996
7711
6426
5141
3855
2570
1285
0
11.29
10.70
10.11
9.51
8.92
8.32
7.73
7.13
6.54
5.94
5.35
4.76
4.16
3.57
2.97
2.38
1.78
1.19
0.59
0.00
1
6
11
16
21
26
31
36
41
46
51
56

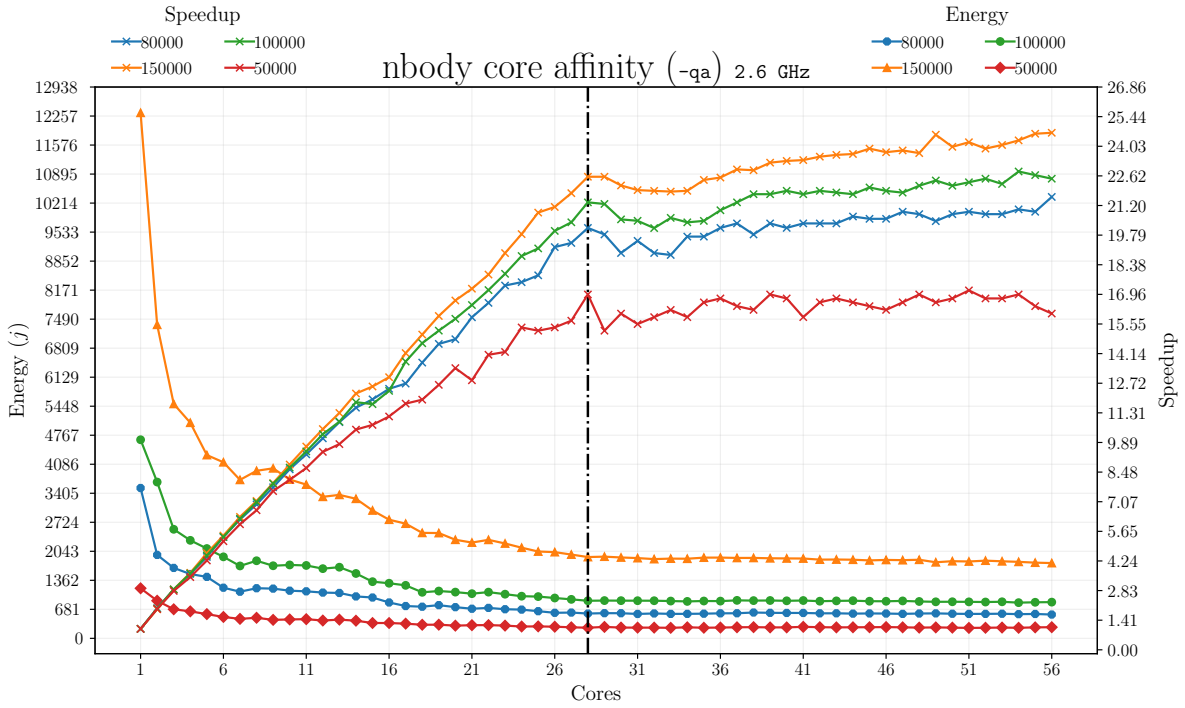

Speedup
80000
85000
90000
75000
Energy
80000
85000
90000
75000
sumeuler core affinity (-qa) 2.6 GHz
Energy (j)
Speedup
Cores
17015
16119
15224
14328
13433
12537
11642
10746
9851
8955
8060
7164
6269
5373
4478
3582
2687
1791
896
0
35.02
33.18
31.34
29.49
27.65
25.81
23.96
22.12
20.28
18.43
16.59
14.75
12.90
11.06
9.22
7.37
5.53
3.69
1.84
0.00
1
6
11
16
21
26
31
36
41
46
51
56

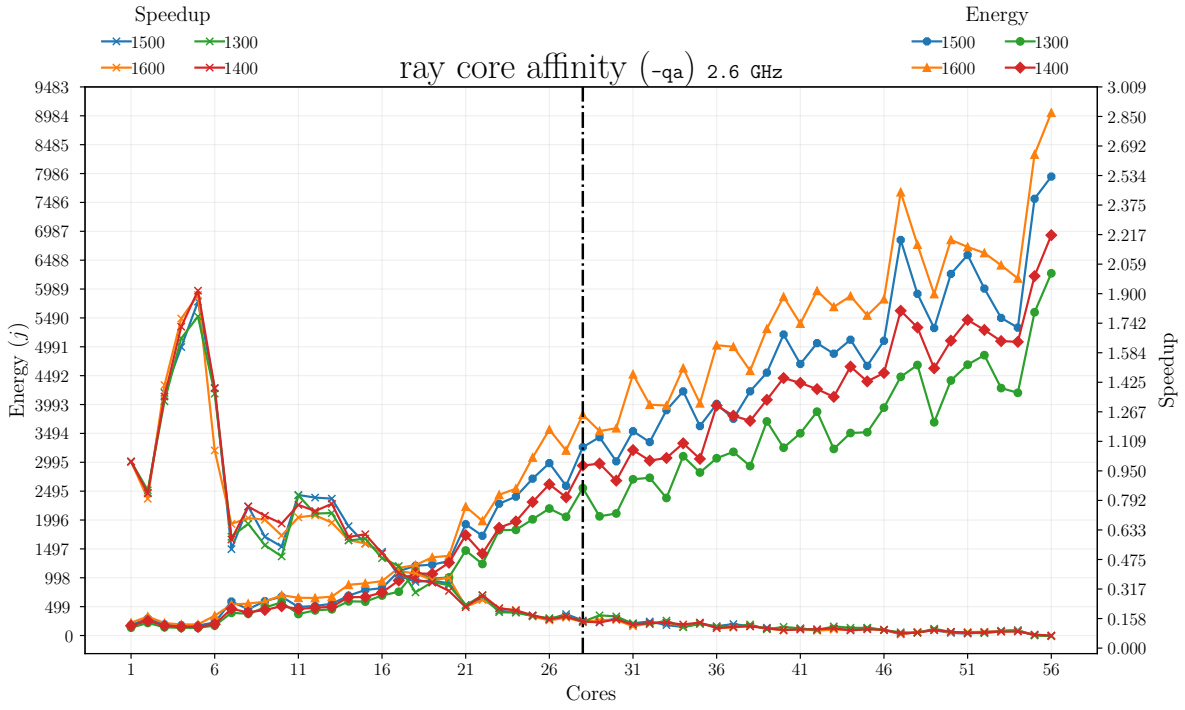

Speedup
52
50
54
56
Energy
52
50
54
56
parfib core affinity (-qa) 2.6 GHz
Energy (j)
Speedup
Cores
68217
64626
61036
57446
53855
50265
46675
43084
39494
35904
32313
28723
25132
21542
17952
14361
10771
7181
3590
0
35.52
33.65
31.78
29.91
28.04
26.17
24.30
22.43
20.56
18.69
16.82
14.95
13.09
11.22
9.35
7.48
5.61
3.74
1.87
0.00
1
6
11
16
21
26
31
36
41
46
51
56


prsa core affinity (-qa) 2.6 GHz
Speedup
9000000
8000000
6000000
2000000
Energy
9000000
8000000
6000000
2000000
Energy (j)
Speedup
Cores
2029
1922
1815
1708
1602
1495
1388
1281
1174
1068
961
854
747
641
534
427
320
214
107
0
14.94
14.15
13.36
12.58
11.79
11.01
10.22
9.43
8.65
7.86
7.07
6.29
5.50
4.72
3.93
3.14
2.36
1.57
0.79
0.00
1
6
11
16
21
26
31
36
41
46
51
56


queens core affinity (-qa) 2.6 GHz
Speedup
14
15
16
13
Energy
14
15
16
13
Energy (j)
Speedup
Cores
15822
14989
14156
13324
12491
11658
10825
9993
9160
8327
7495
6662
5829
4996
4164
3331
2498
1665
833
0
13.63
12.91
12.20
11.48
10.76
10.04
9.33
8.61
7.89
7.17
6.46
5.74
5.02
4.30
3.59
2.87
2.15
1.43
0.72
0.00
1
6
11
16
21
26
31
36
41
46
51
56

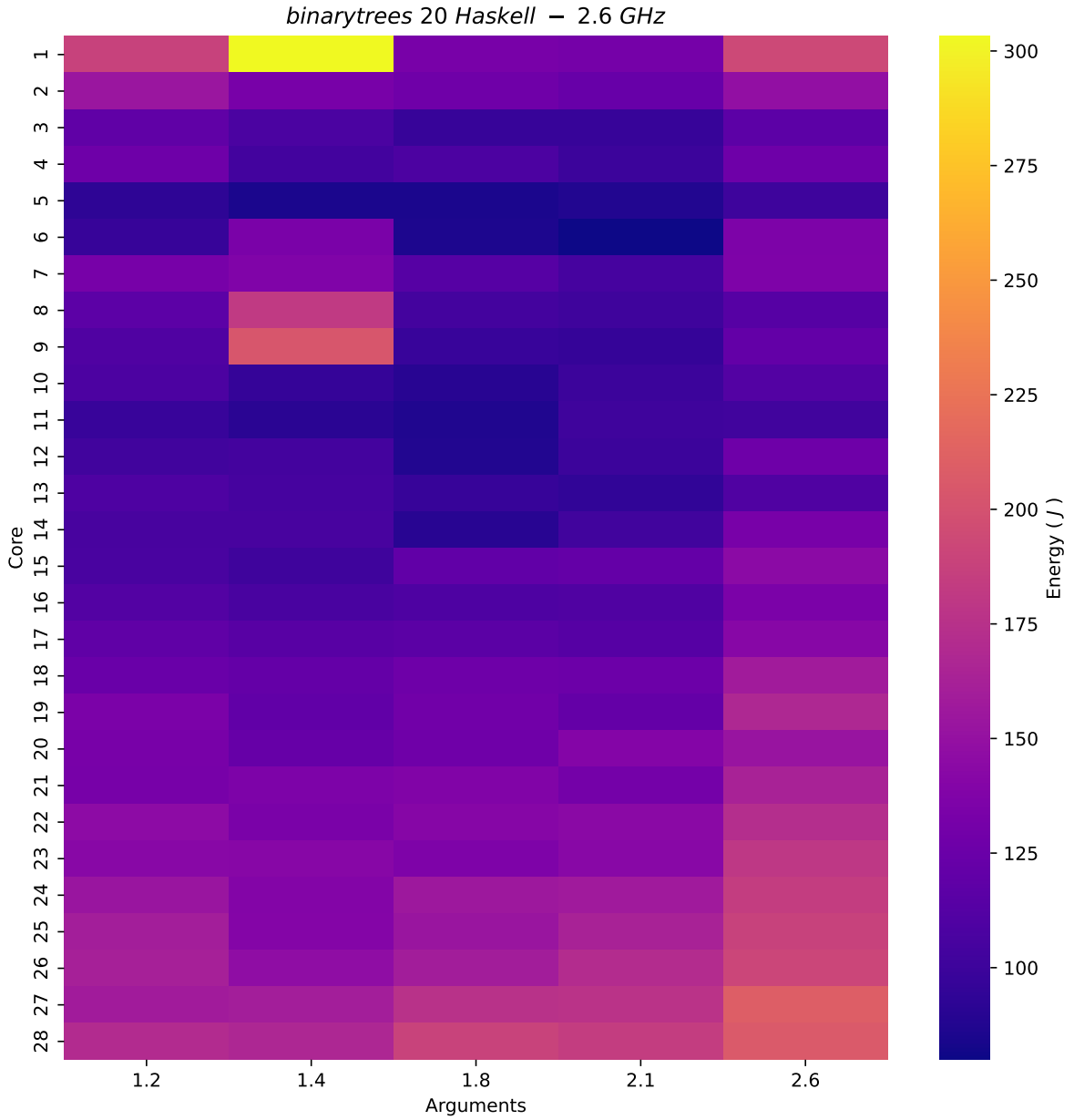

Speedup
1000000
6000000
3000000
500000
Quicksort core affinity (-qa) 2.6 GHz
Energy
1000000
6000000
3000000
500000
Energy (j)
Speedup
Cores

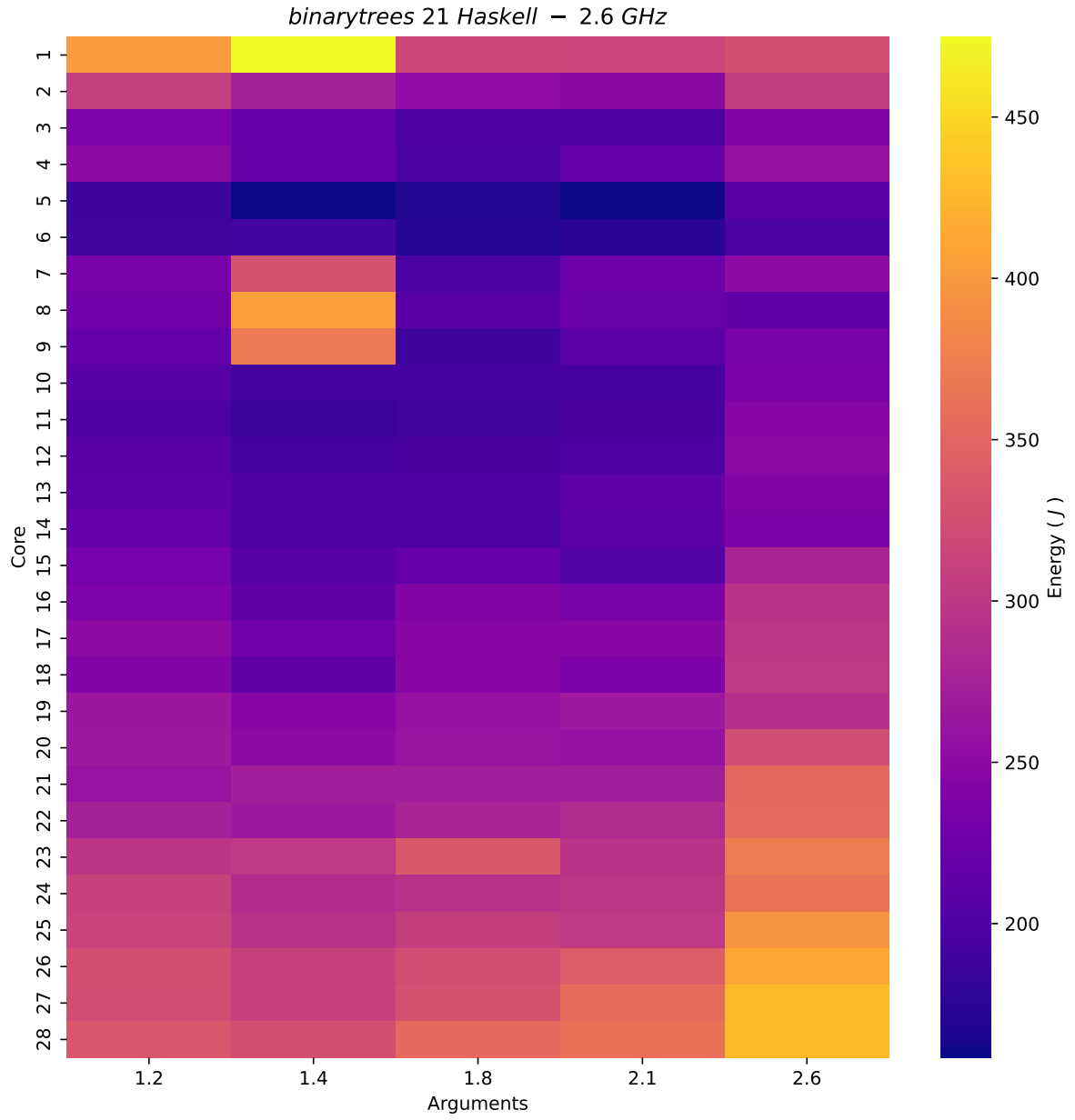

Speedup
80000
100000
150000
50000
nbody core affinity (-qa) 2.6 GHz
Energy
80000
100000
150000
50000
Energy (j)
Speedup
Cores

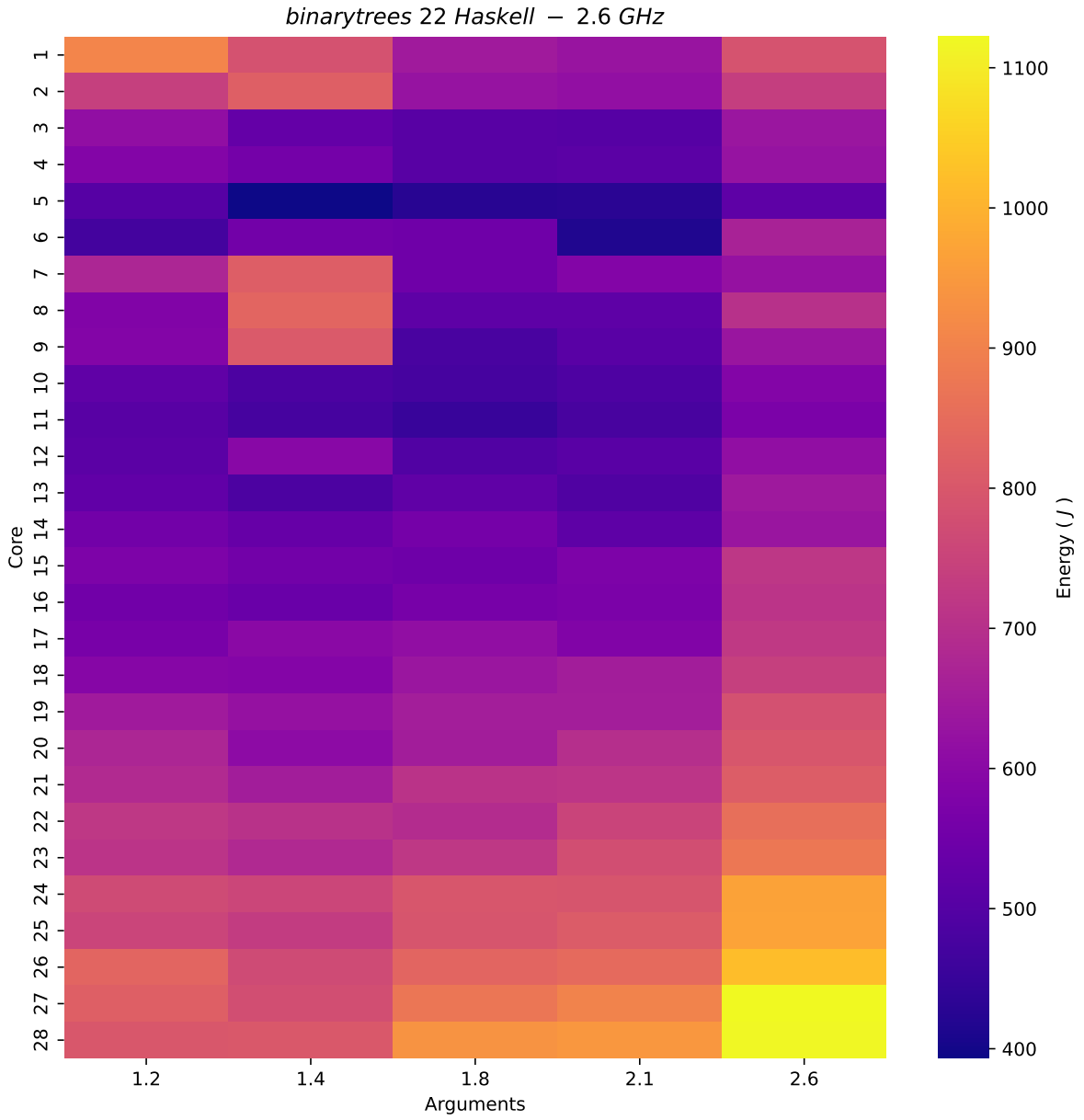

Speedup
1500
1300
1600
1400
ray core affinity (-qa) 2.6 GHz
Energy
1500
1300
1600
1400
Energy (j)
9483
8984
8485
7986
7486
6987
6488
5989
5490
4991
4492
3993
3494
2995
2495
1996
1497
998
499
0
Speedup
3.009
2.850
2.692
2.534
2.375
2.217
2.059
1.900
1.742
1.584
1.425
1.267
1.109
0.950
0.792
0.633
0.475
0.317
0.158
0.000
1
6
11
16
21
26
31
36
41
46
51
56
Cores

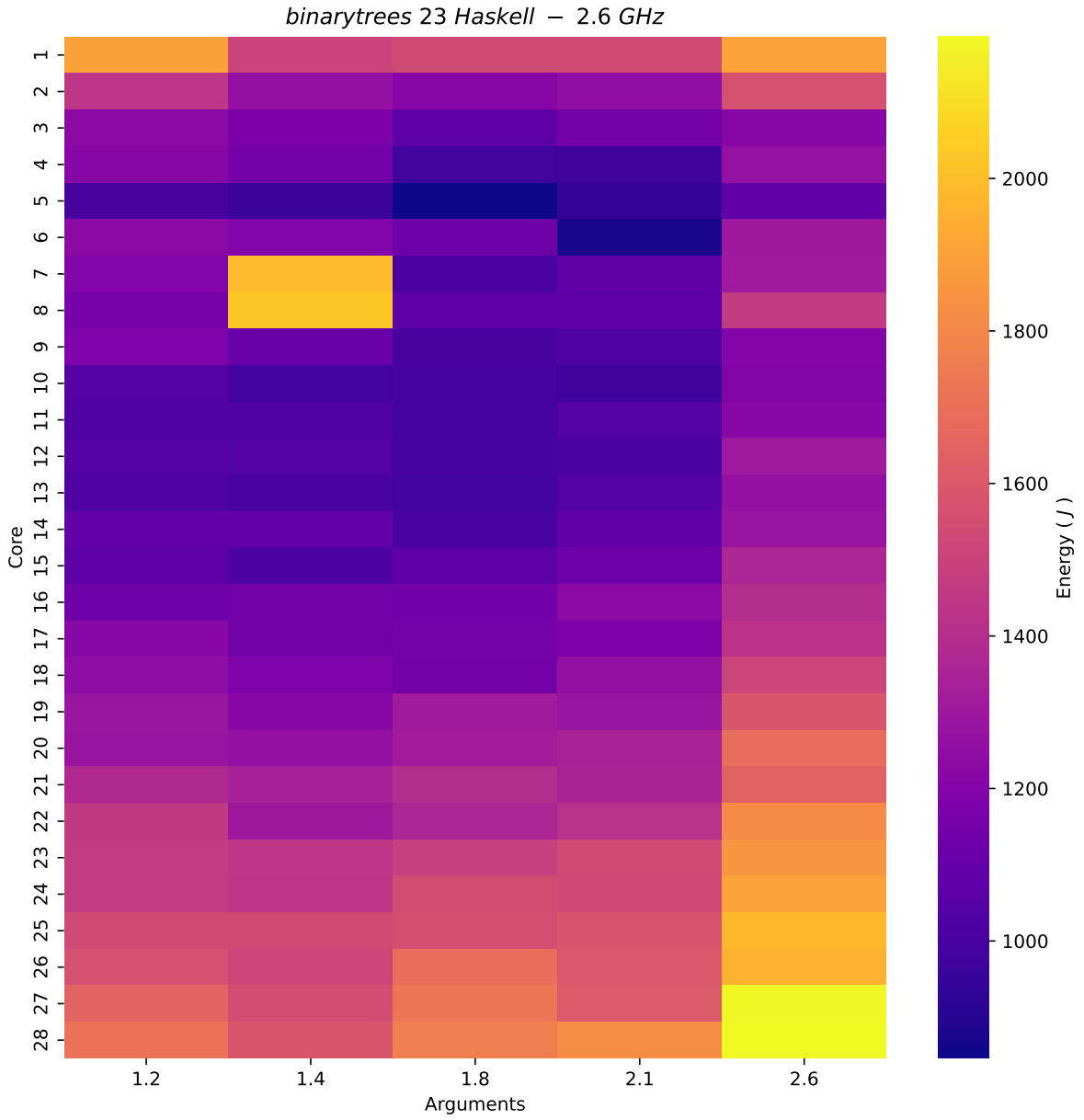
Appendix D

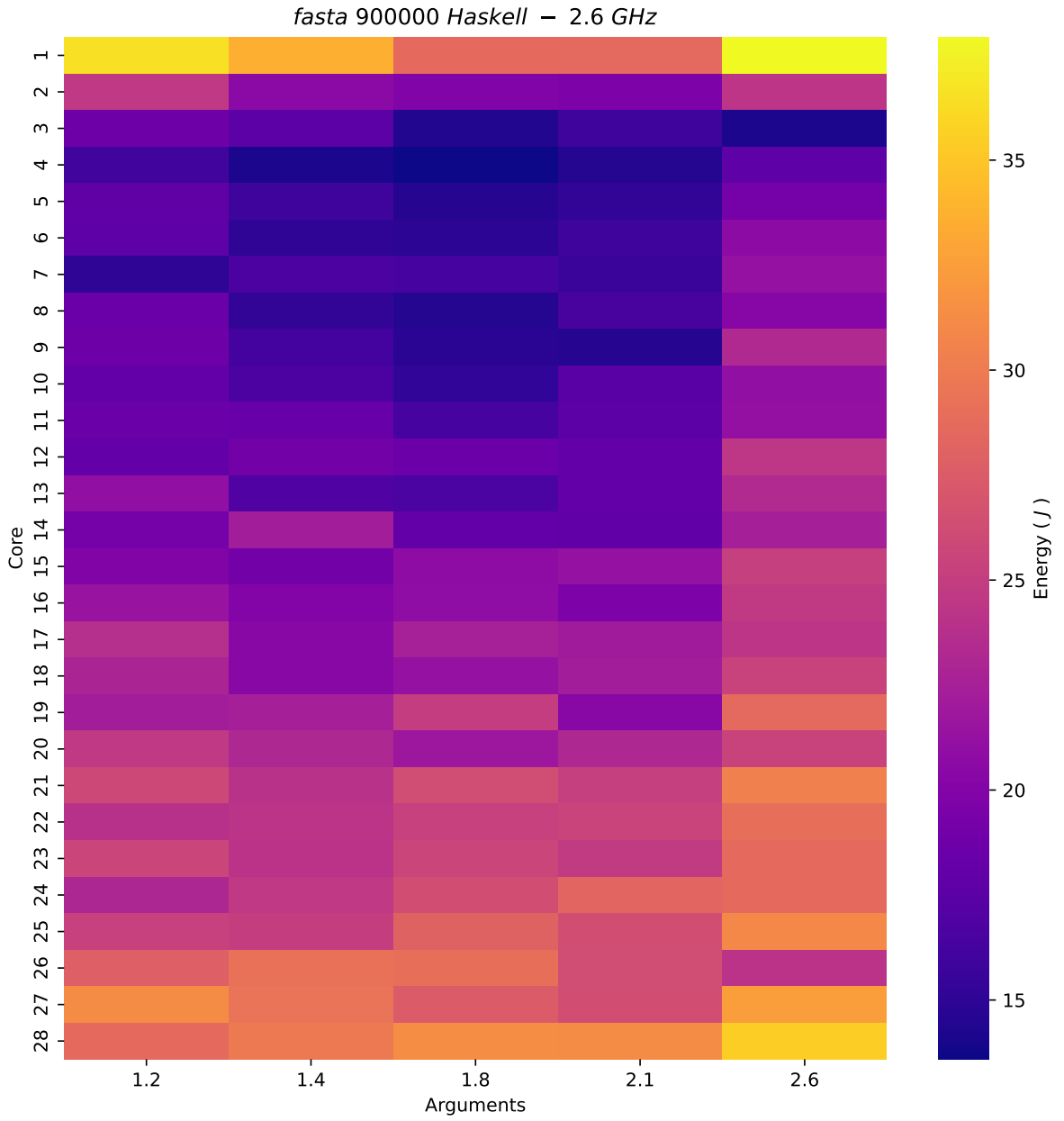
Heatmap Plots

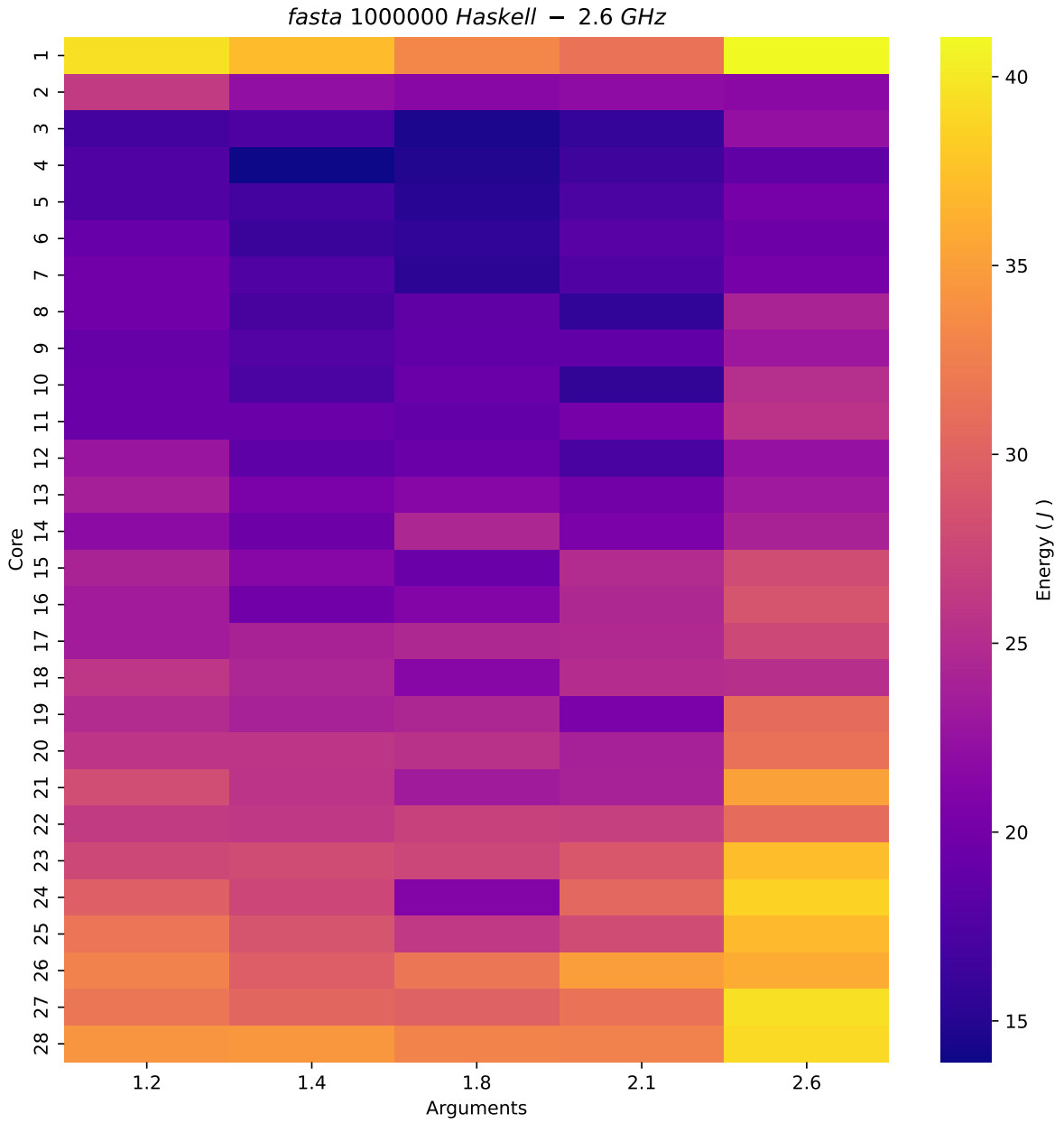


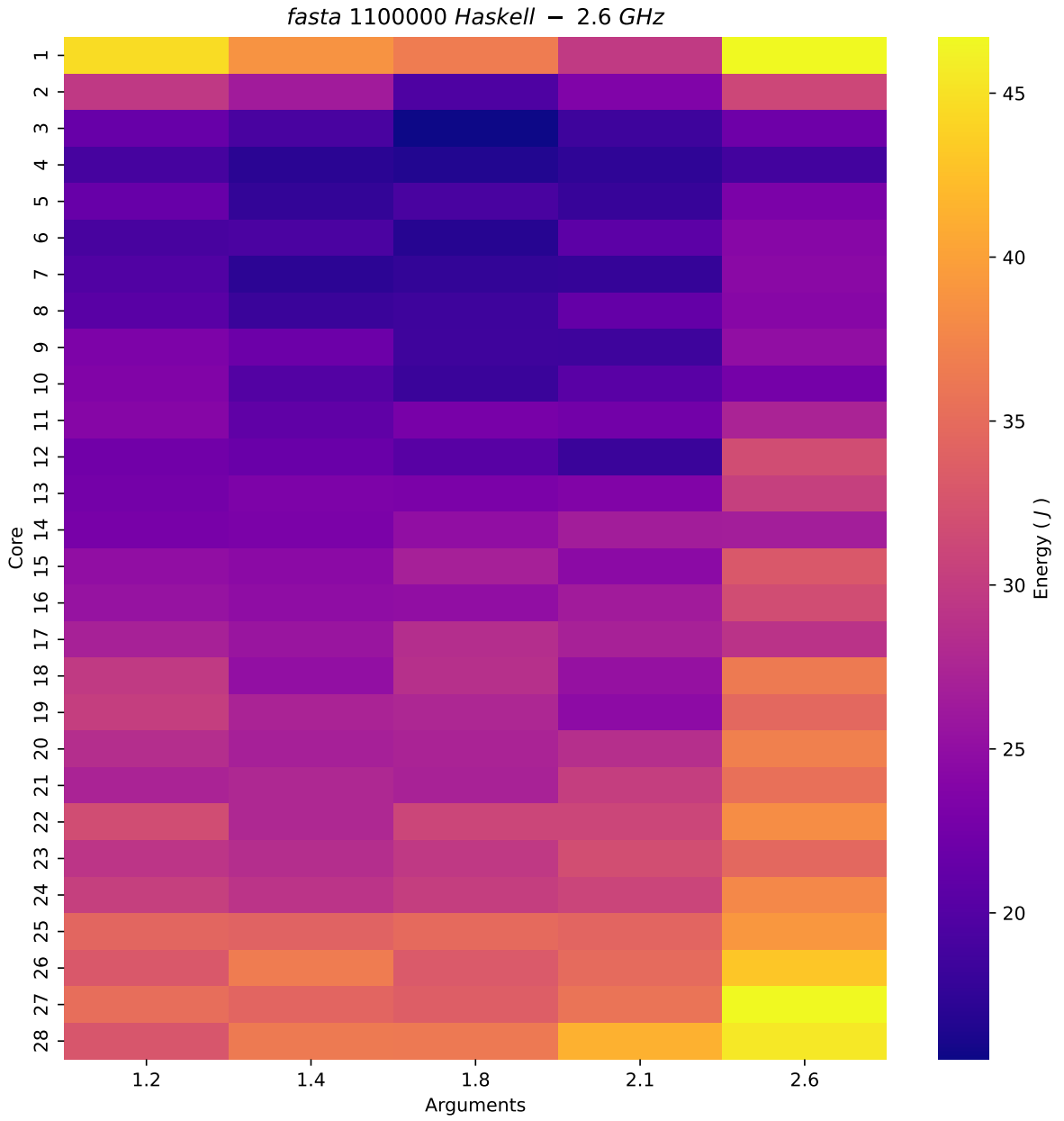

binarytrees 21 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
450
400
350
300
250
200

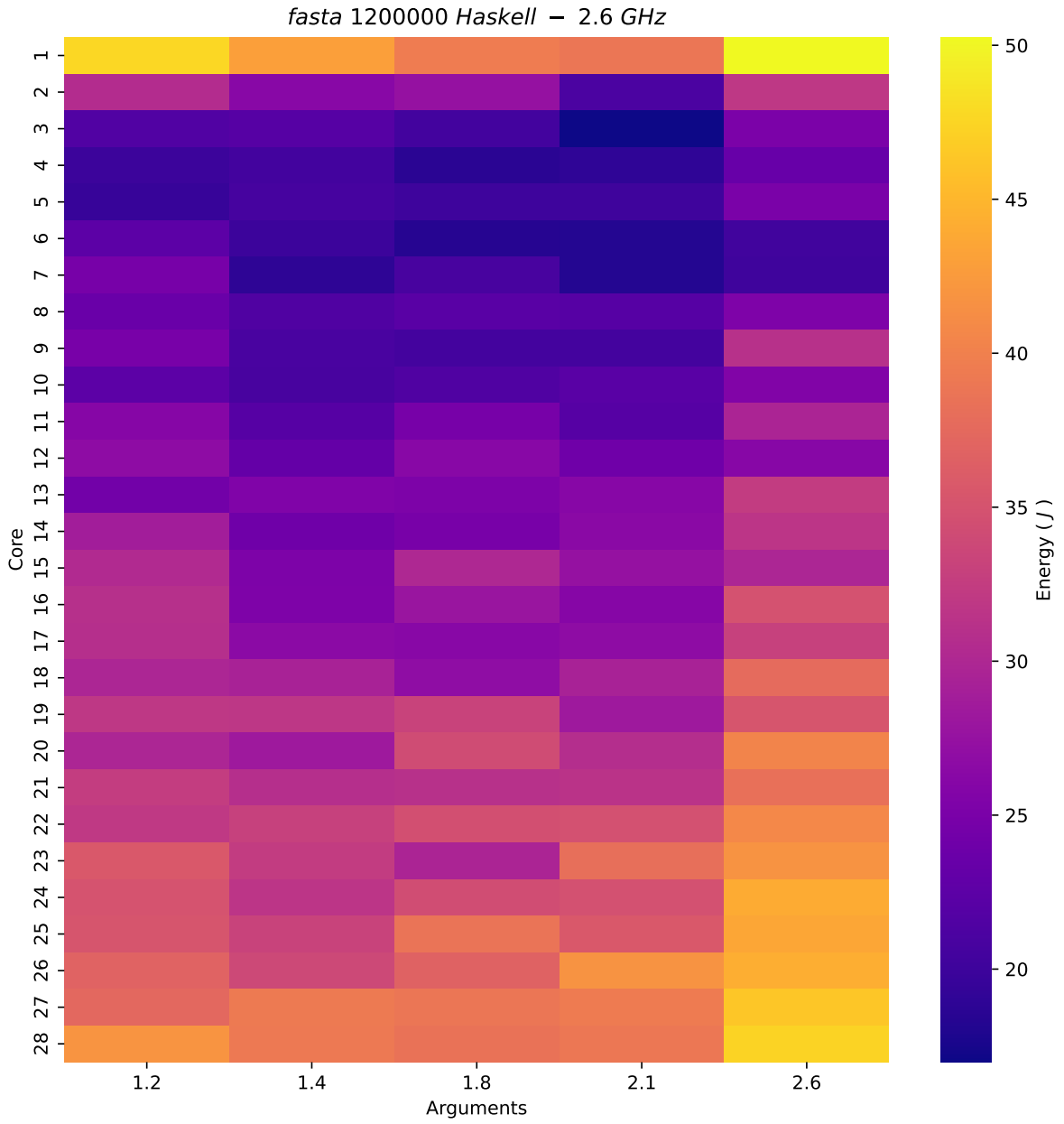

binarytrees 22 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1100
1000
900
800
700
600
500
400

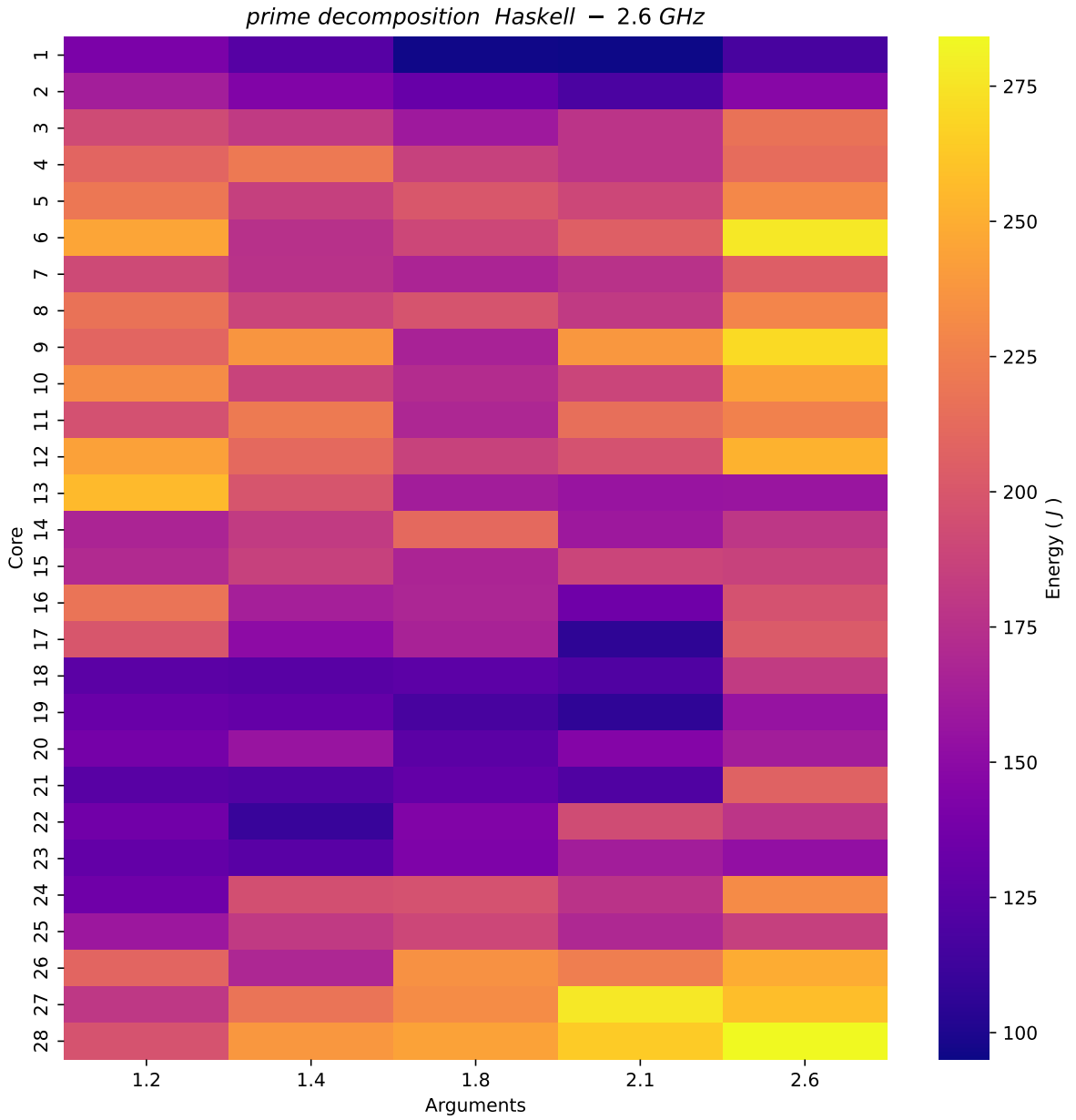

binarytrees 23 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
2000
1800
1600
1400
1200
1000
Energy (J)

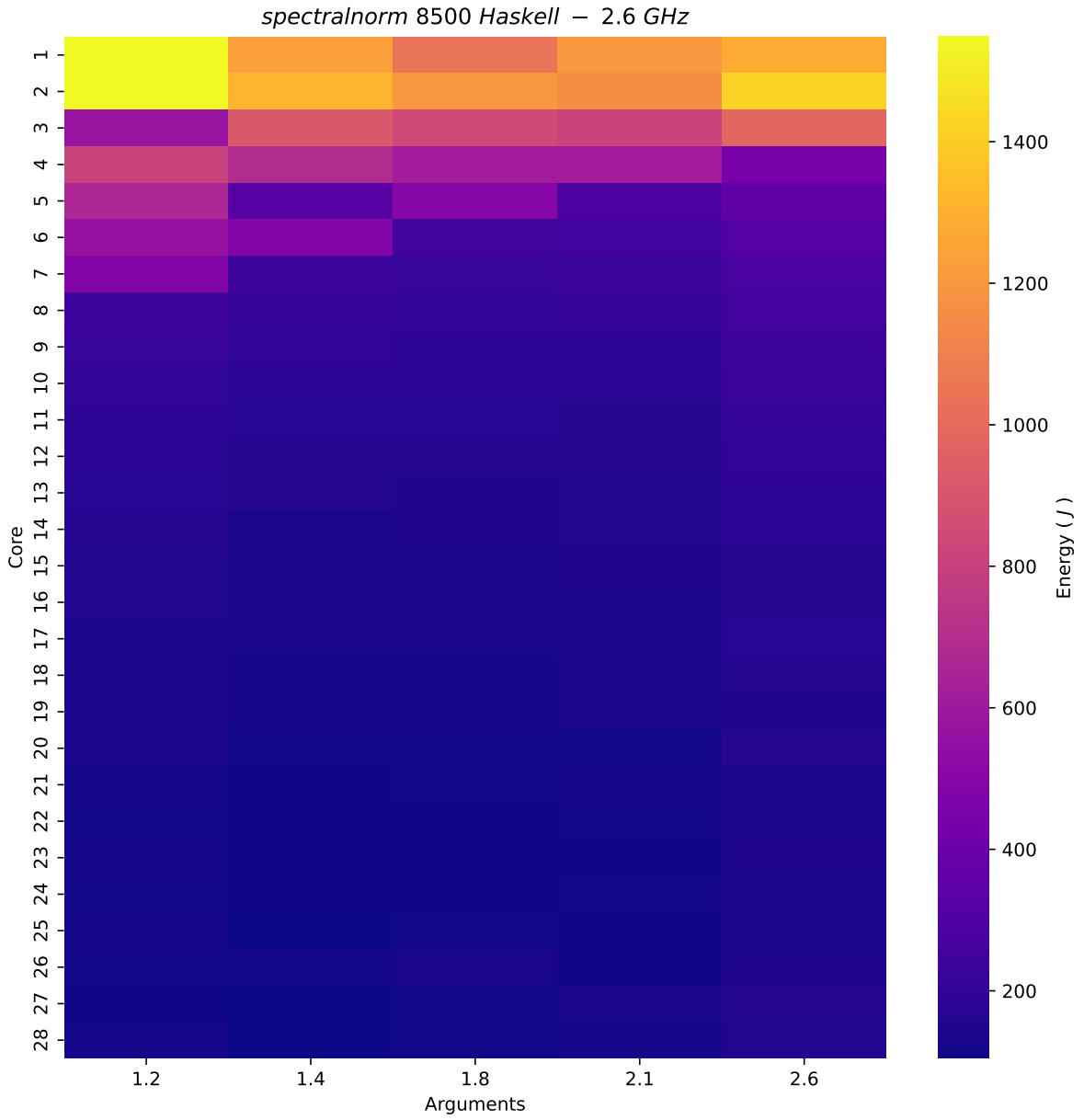

fasta 900000 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
35
30
25
20
15

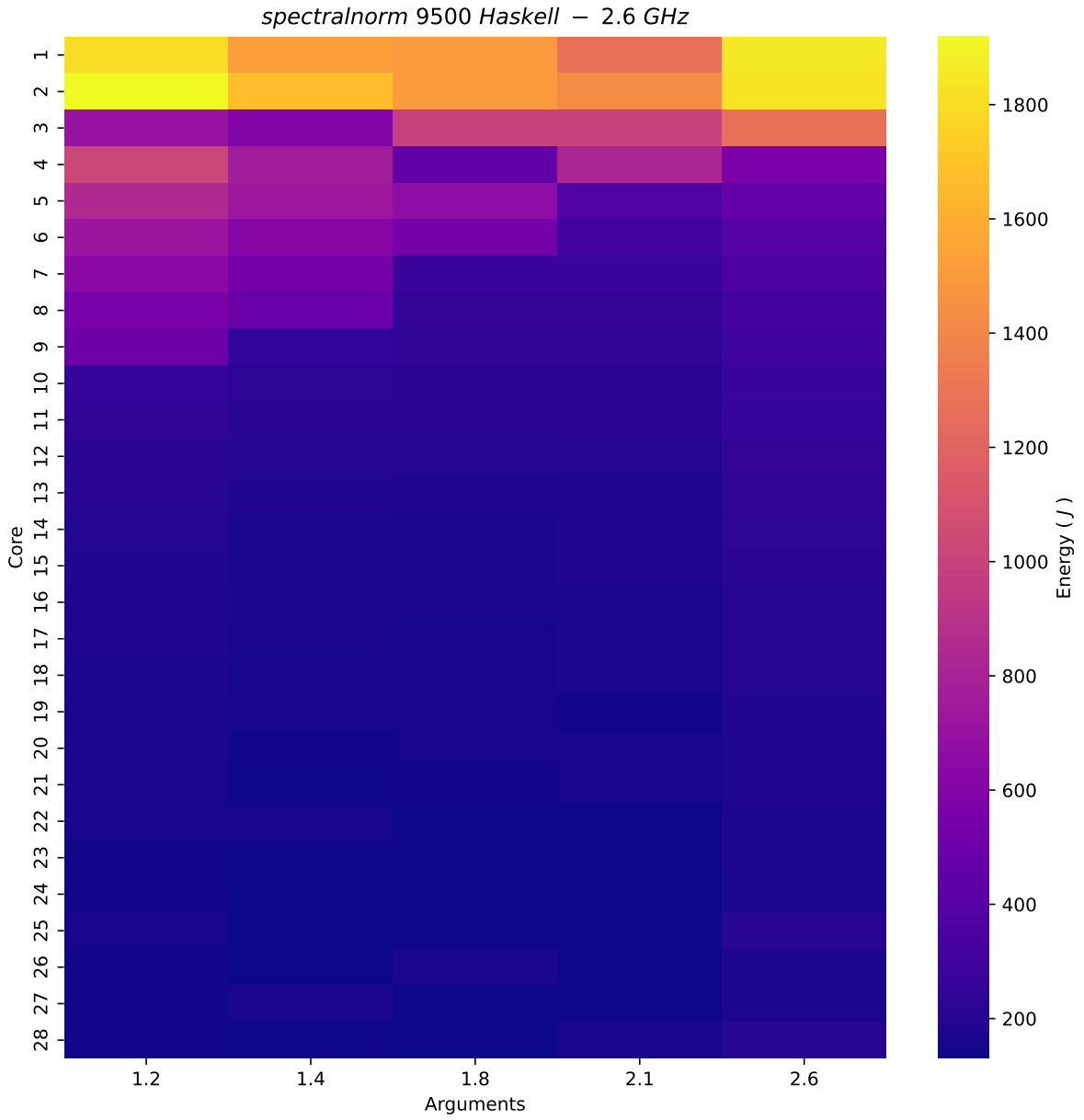

fasta 1000000 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
40
35
30
25
20
15
Energy (J)

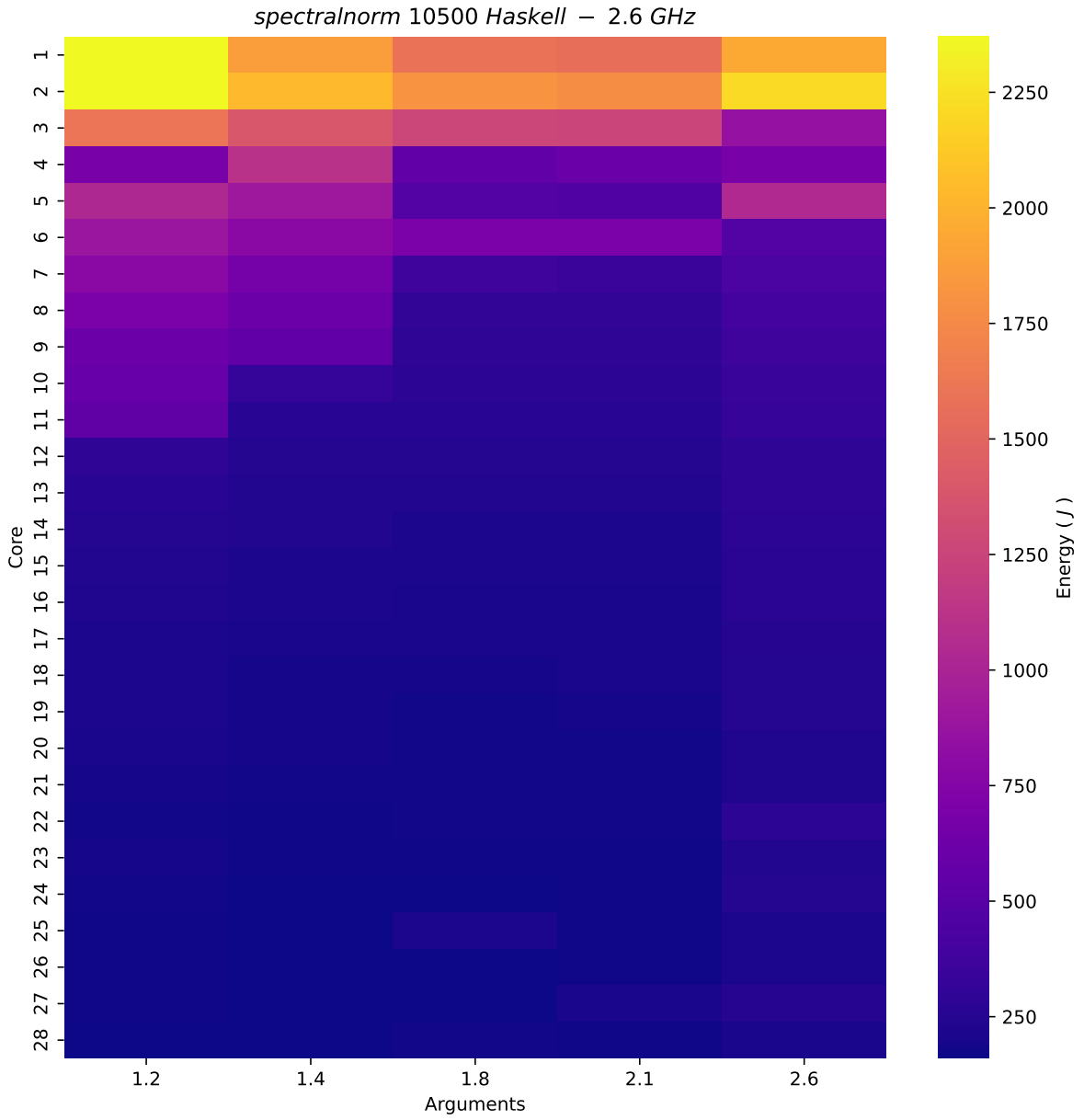

fasta 1100000 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
45
40
35
30
25
20

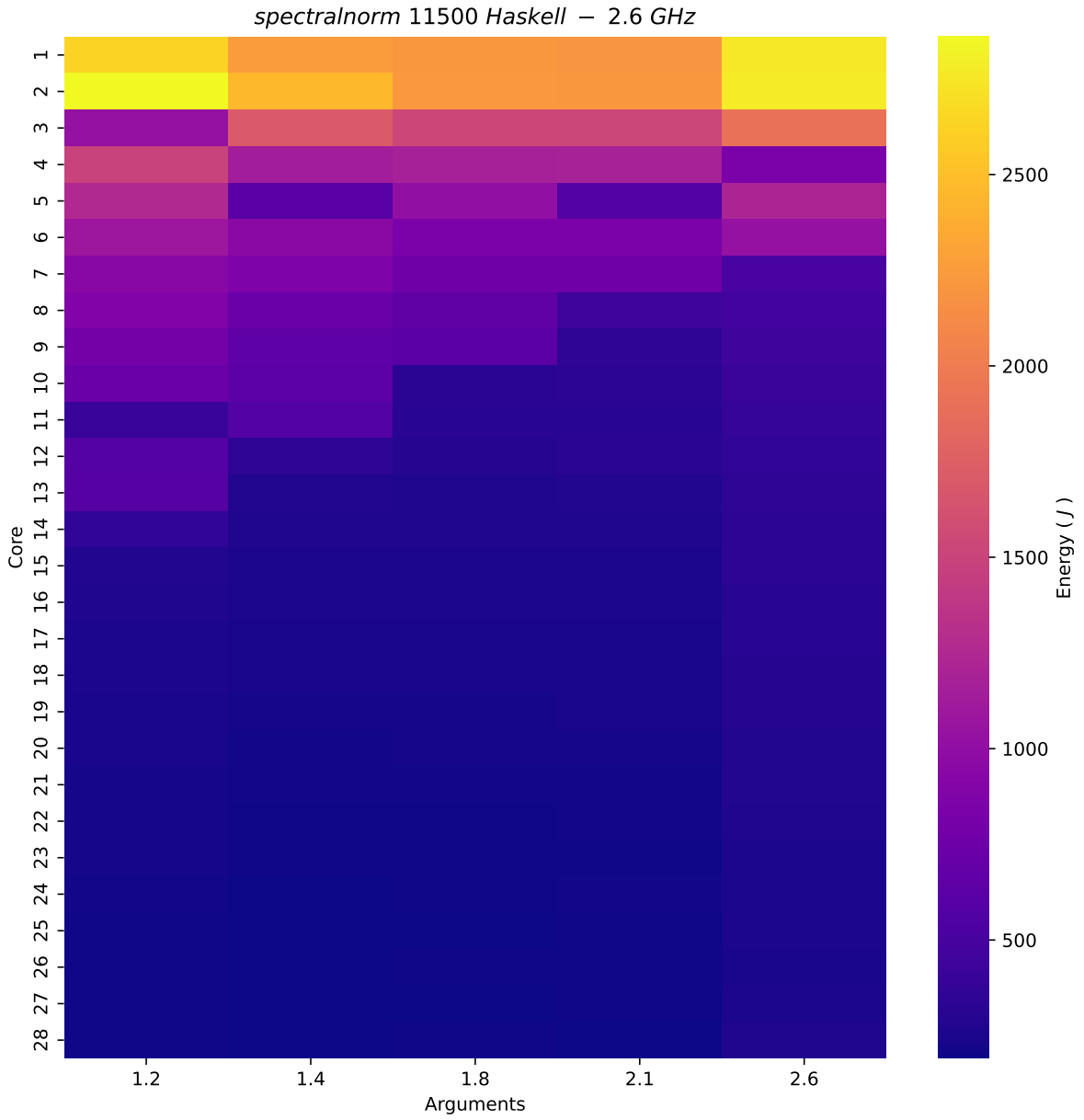

fasta 1200000 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
50
45
40
35
30
25
20

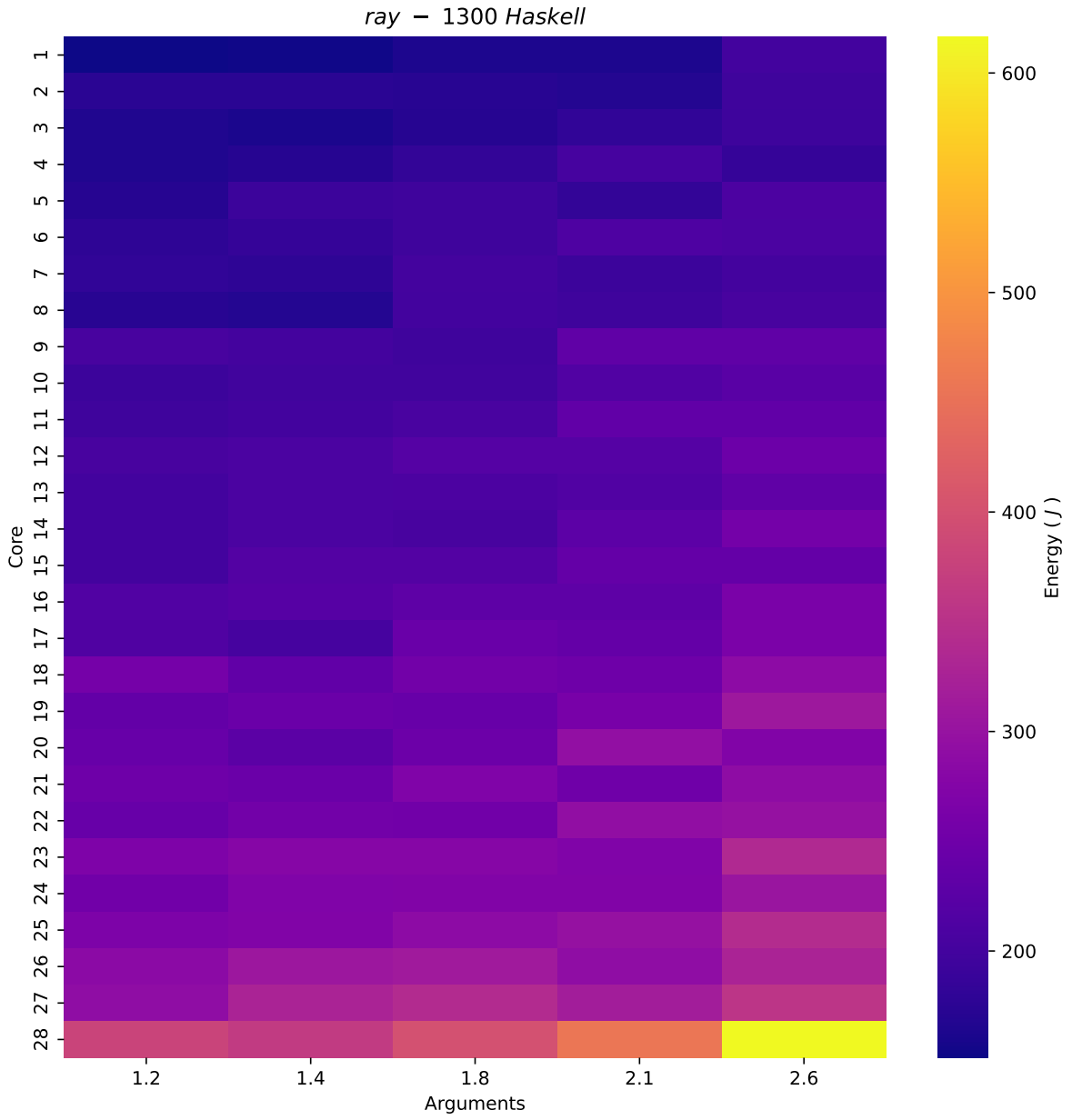

prime decomposition Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
275
250
225
200
175
150
125
100

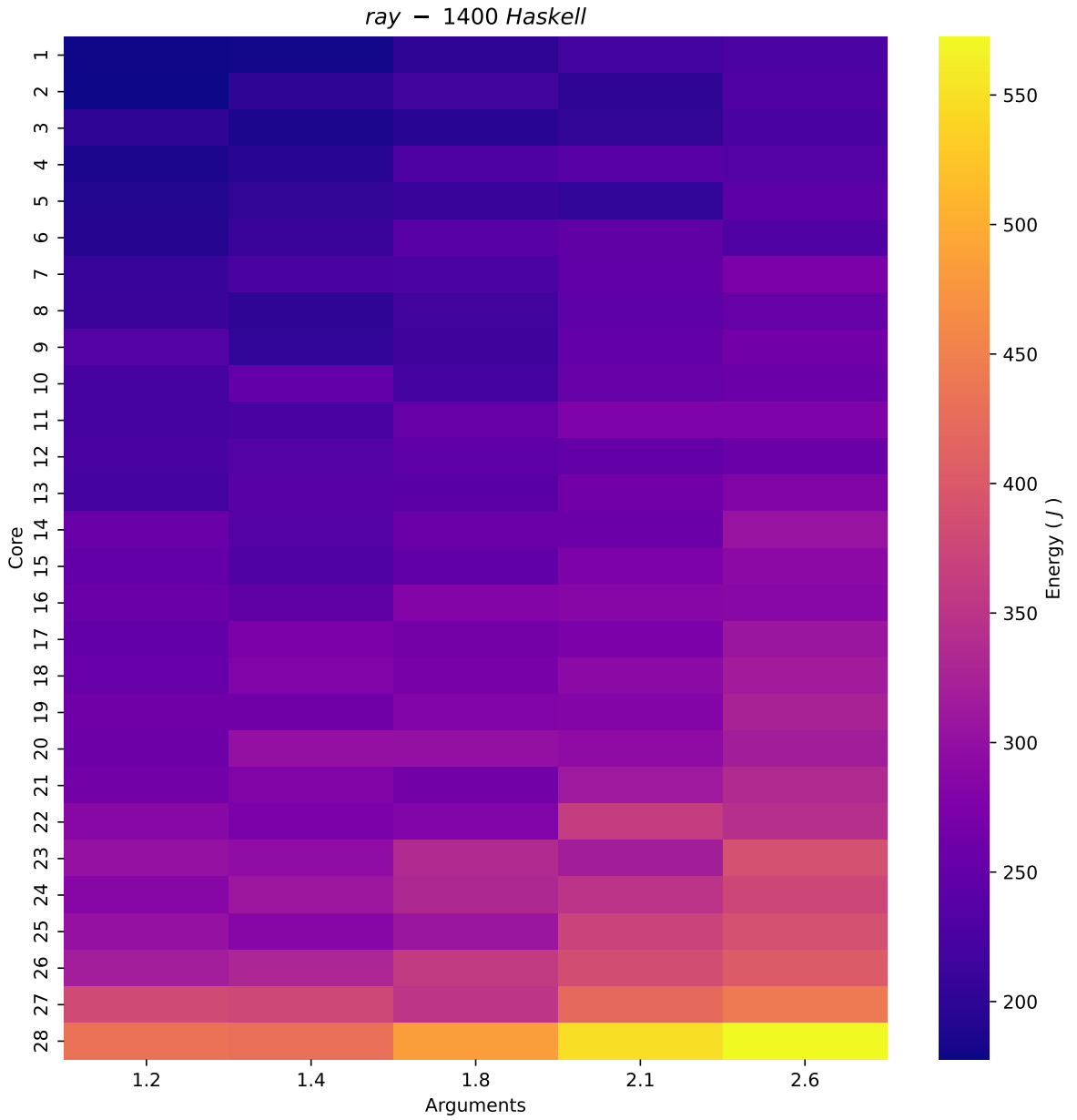

spectralnorm 8500 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
1400
1200
1000
800
600
400
200
Energy (J)

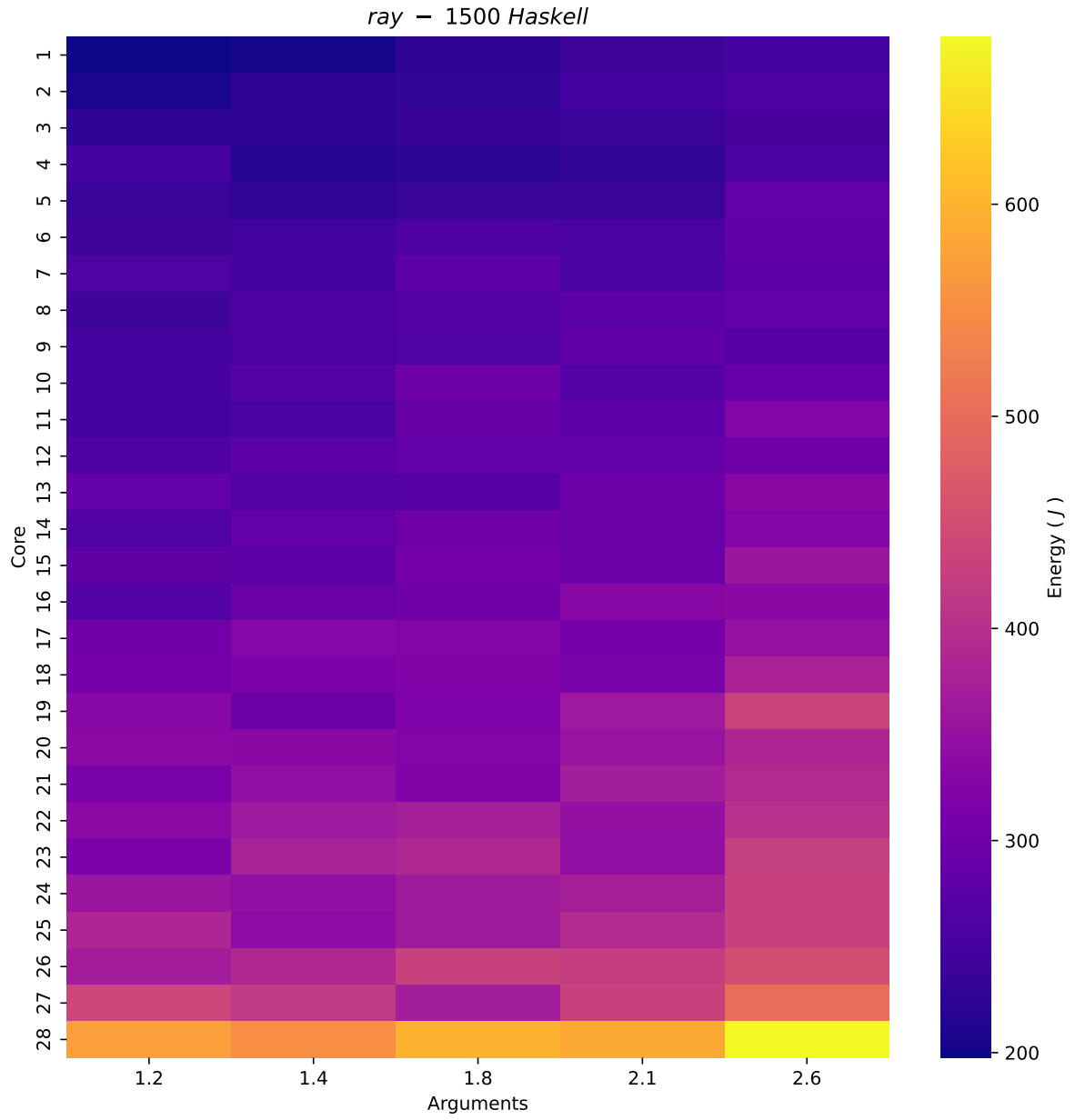

spectralnorm 9500 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
1800
1600
1400
1200
1000
800
600
400
200
Energy (J)

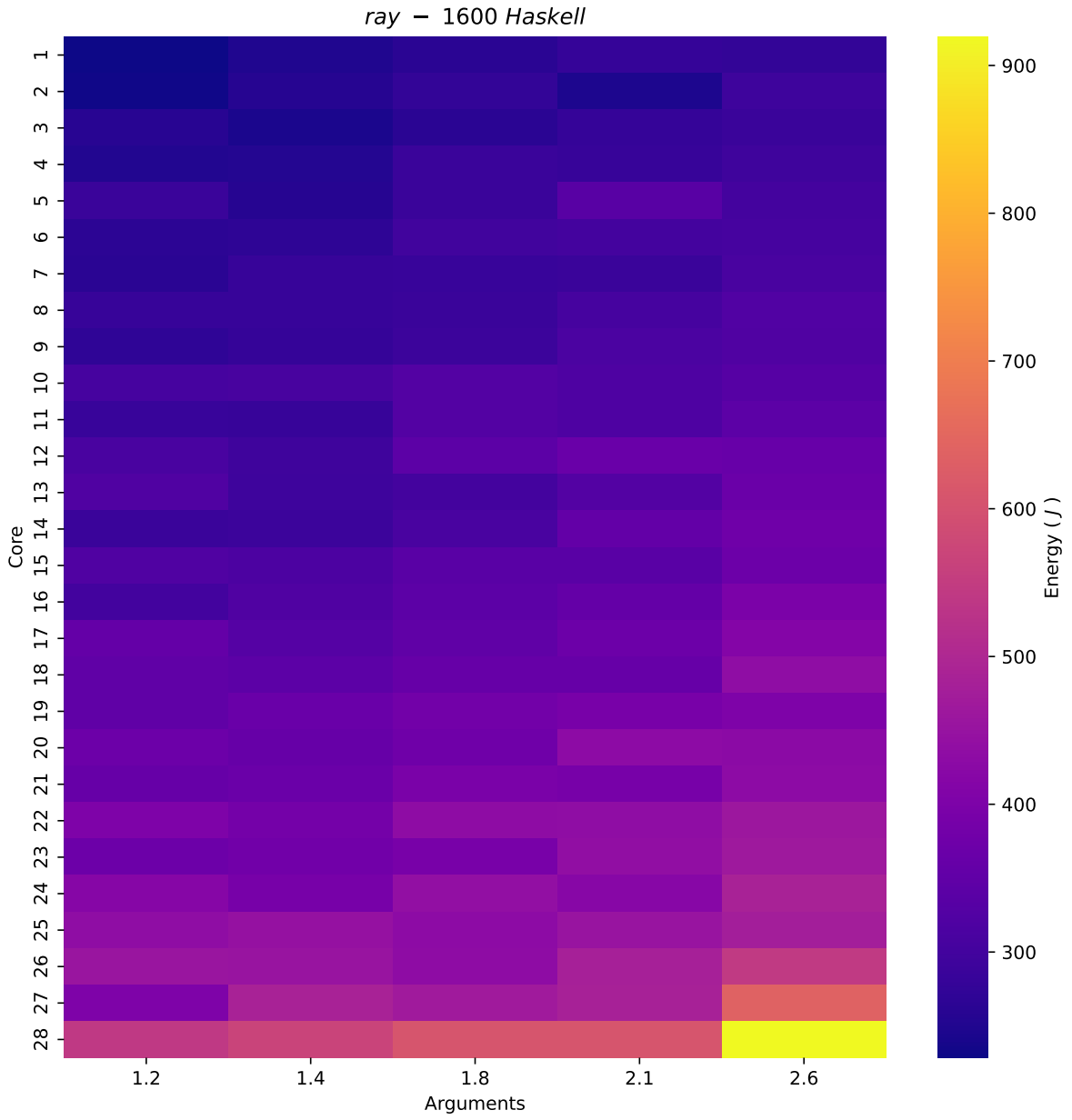

spectralnorm 10500 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
2250
2000
1750
1500
1250
1000
750
500
250
Energy (J)

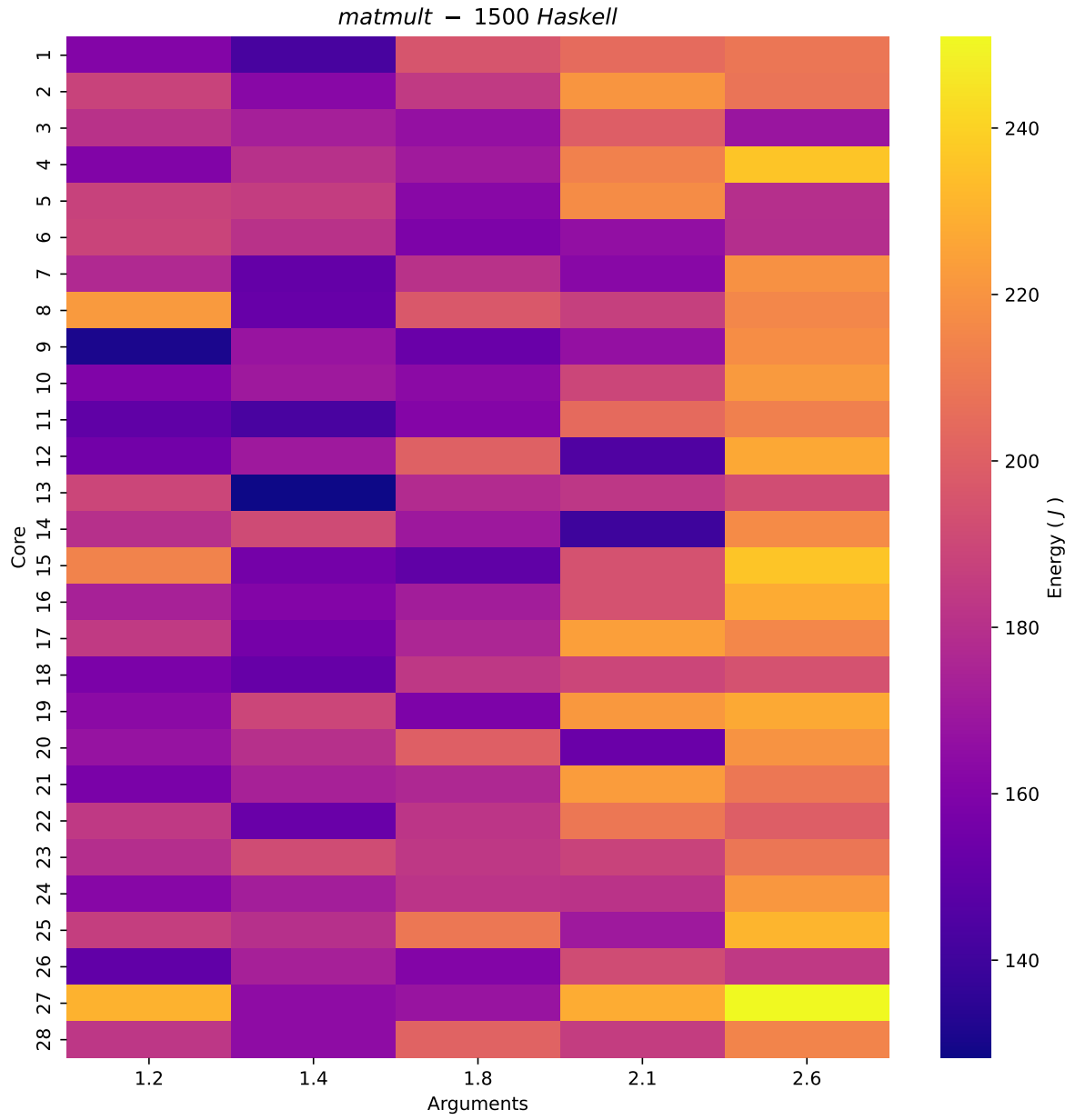

spectralnorm 11500 Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
2500
2000
1500
1000
500

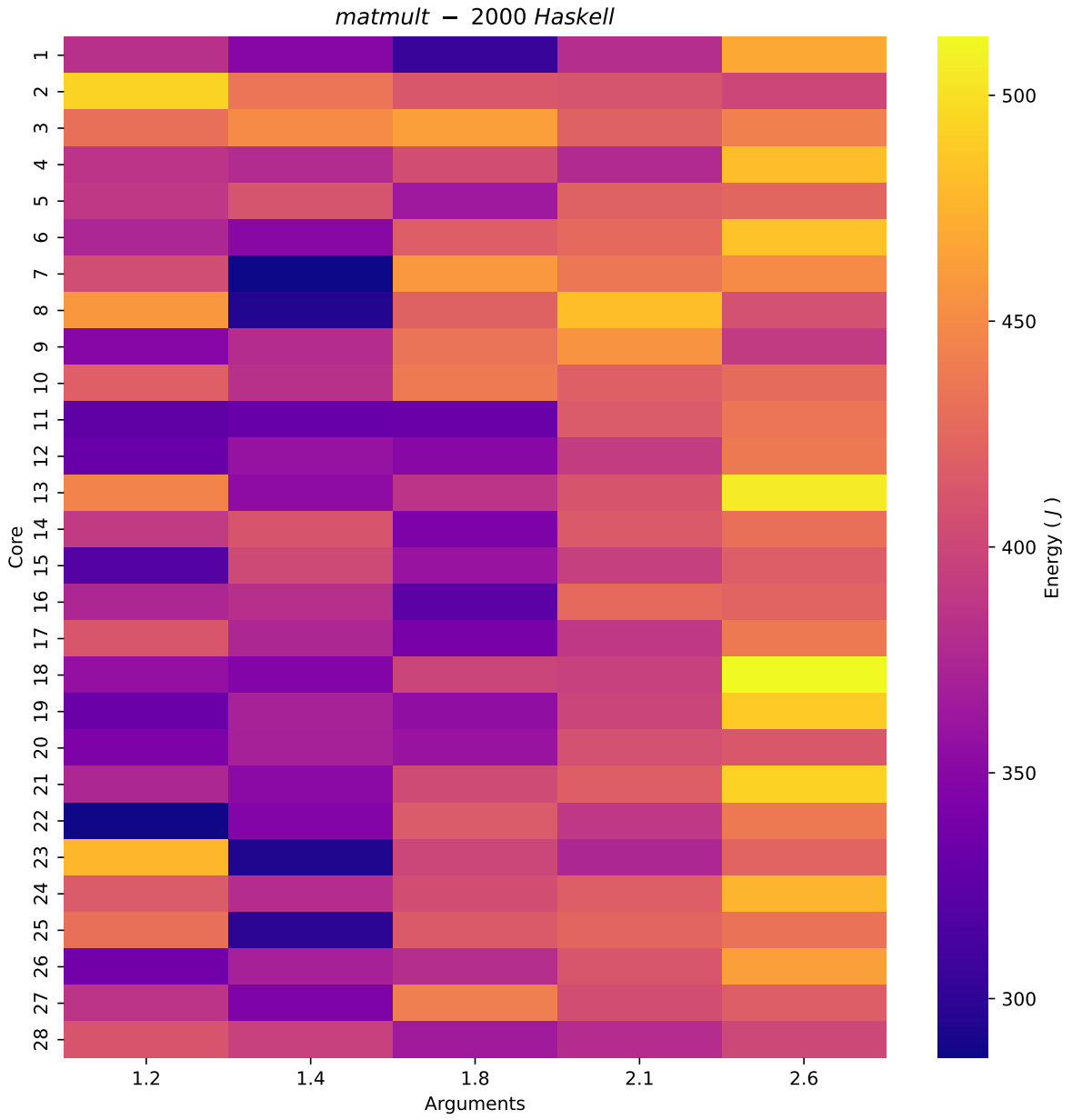

ray − 1300 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
600
500
400
300
200
Energy (J)

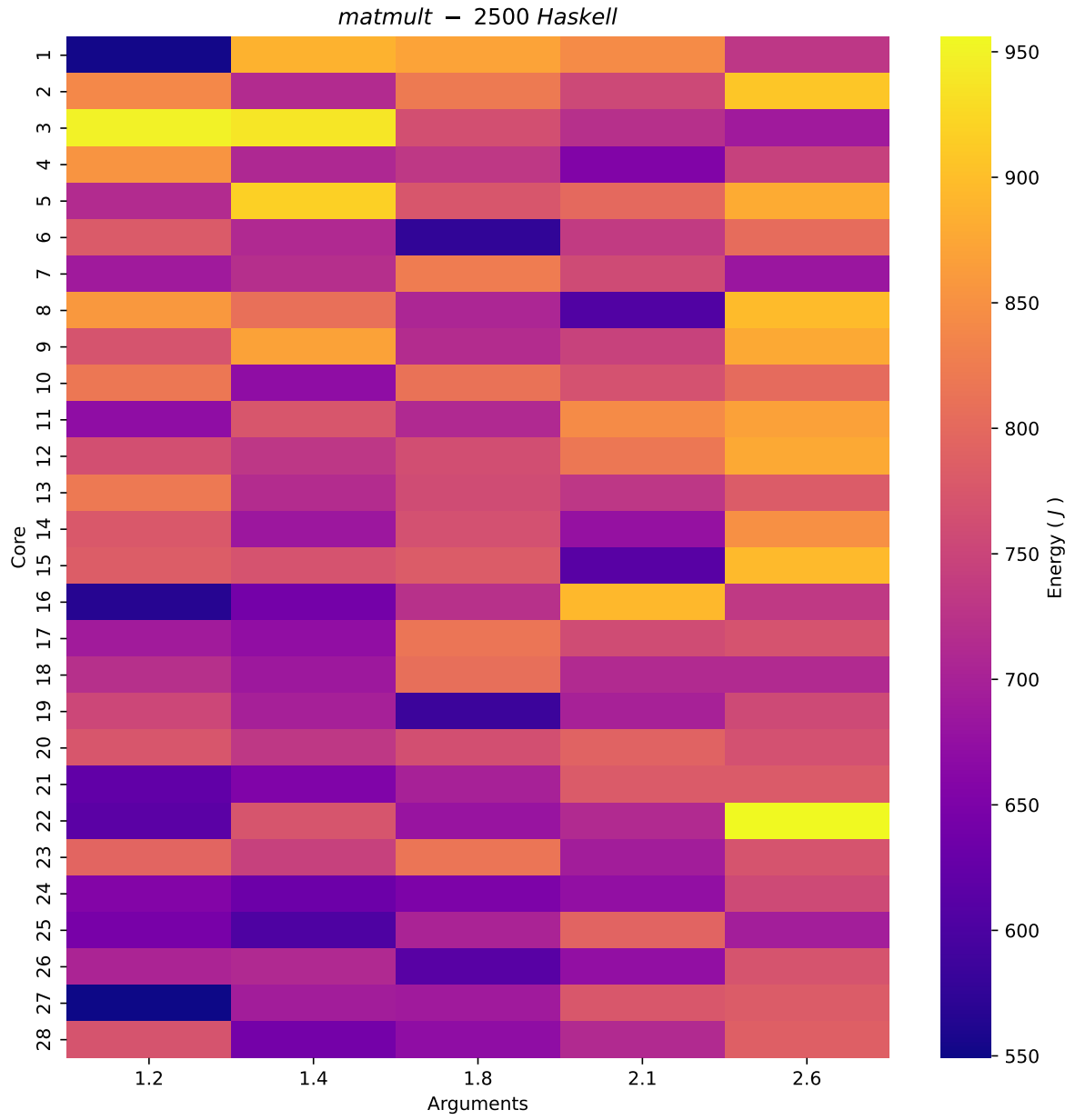

ray − 1400 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
550
500
450
400
350
300
250
200

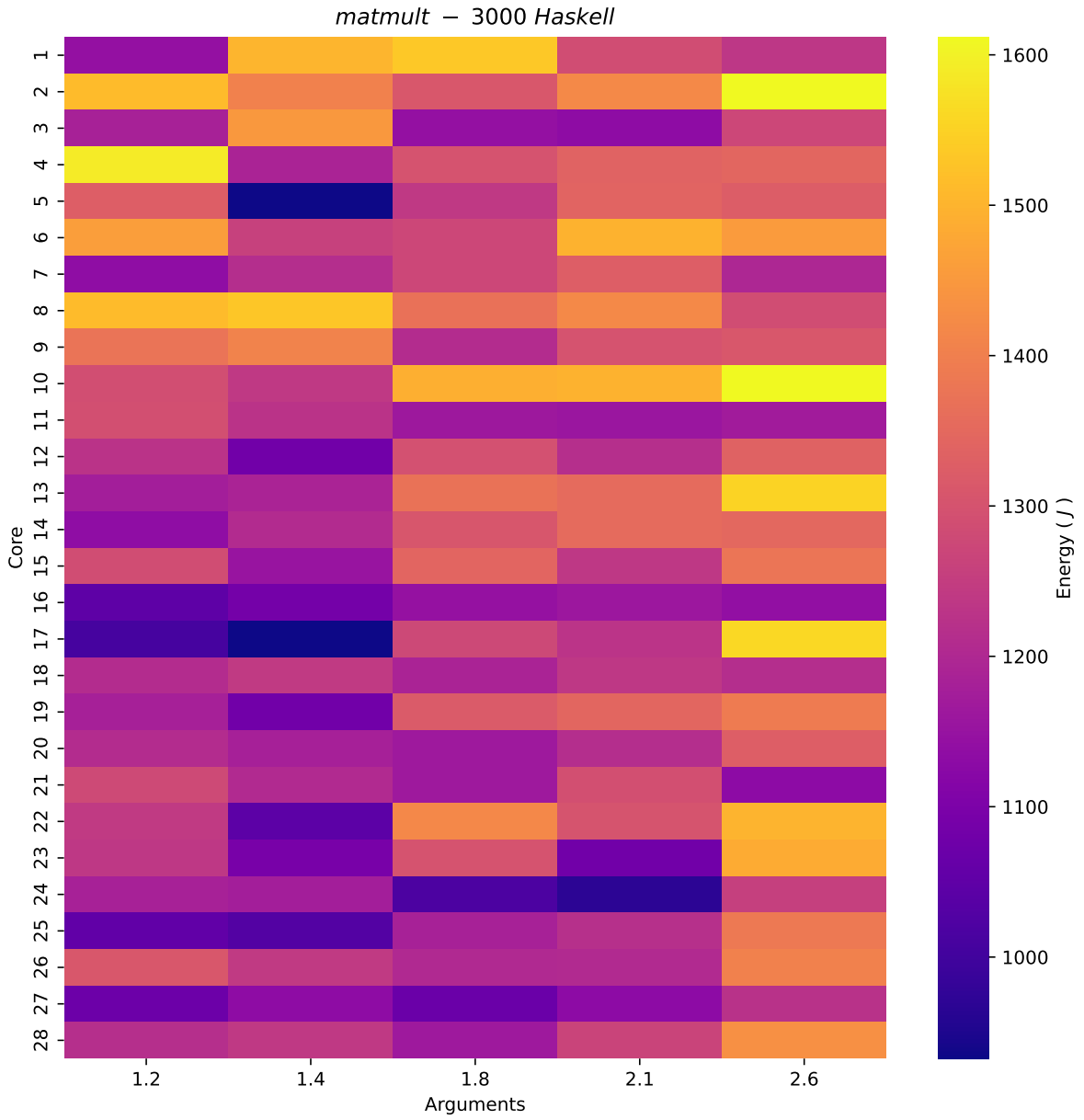

ray − 1500 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
600
500
400
300
200

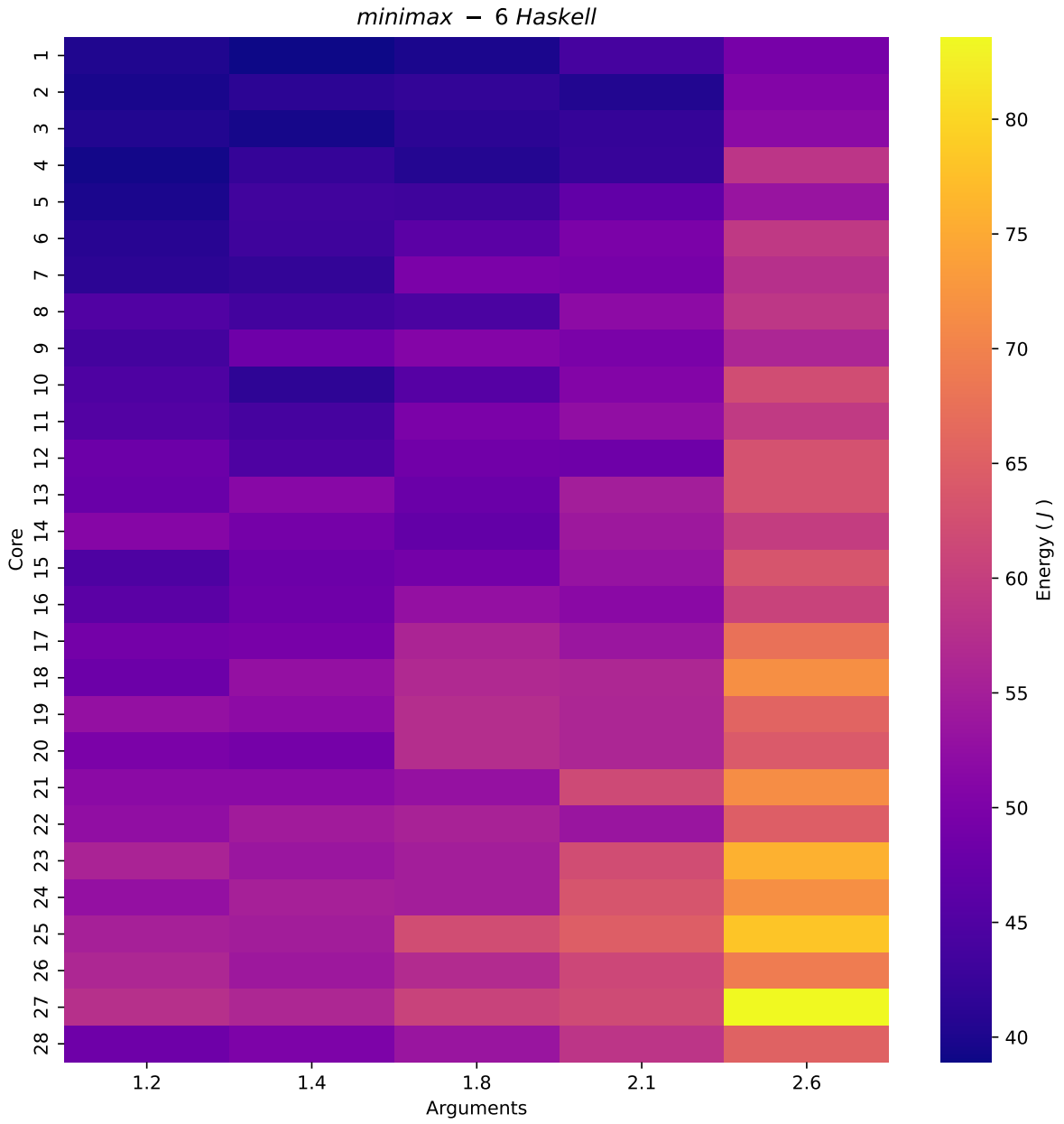

ray − 1600 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
900
800
700
600
500
400
300
Energy (J)

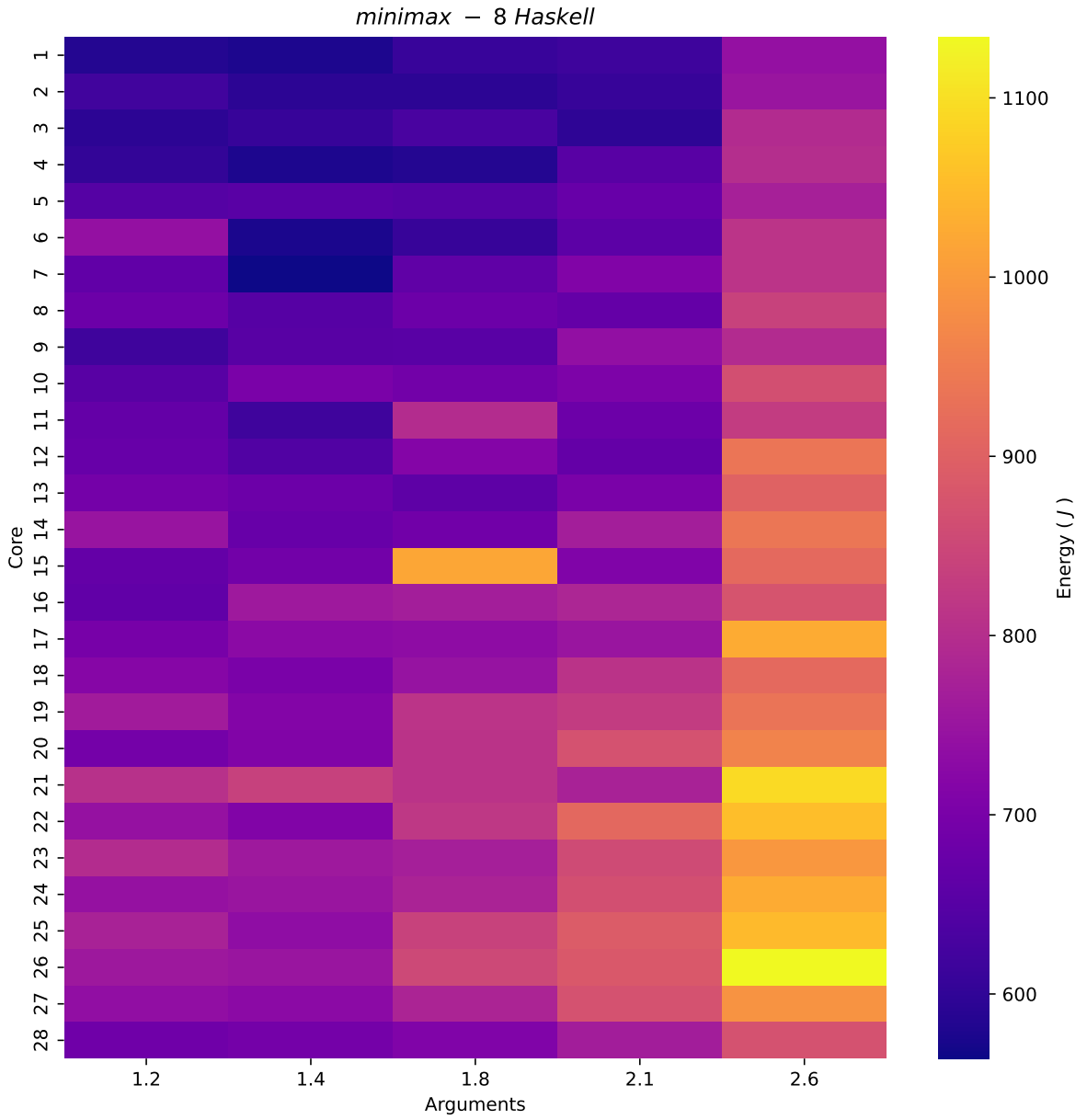

matmult − 1500 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
240
220
200
180
160
140
Energy (J)

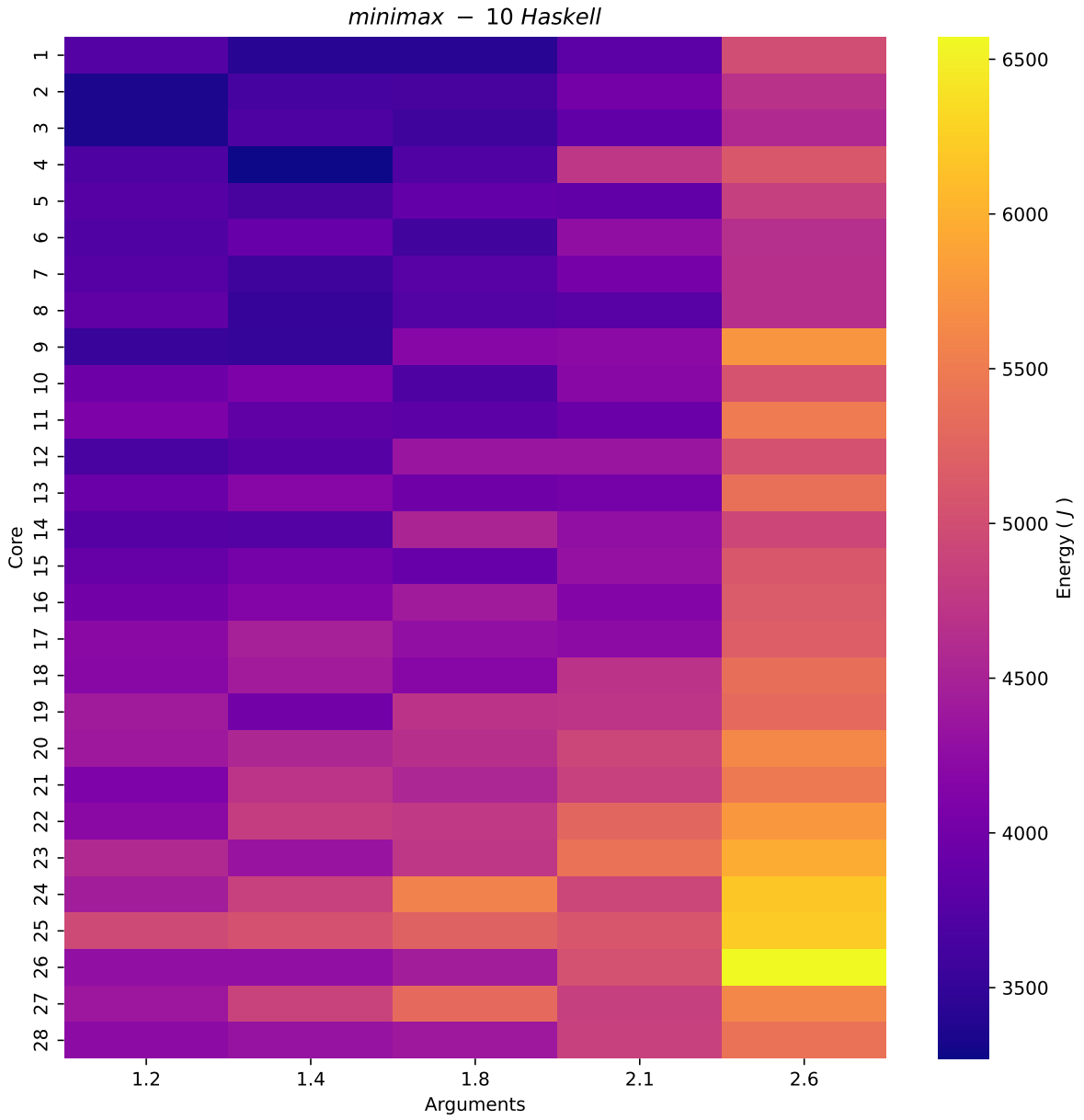

matmult – 2000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
500
450
400
350
300

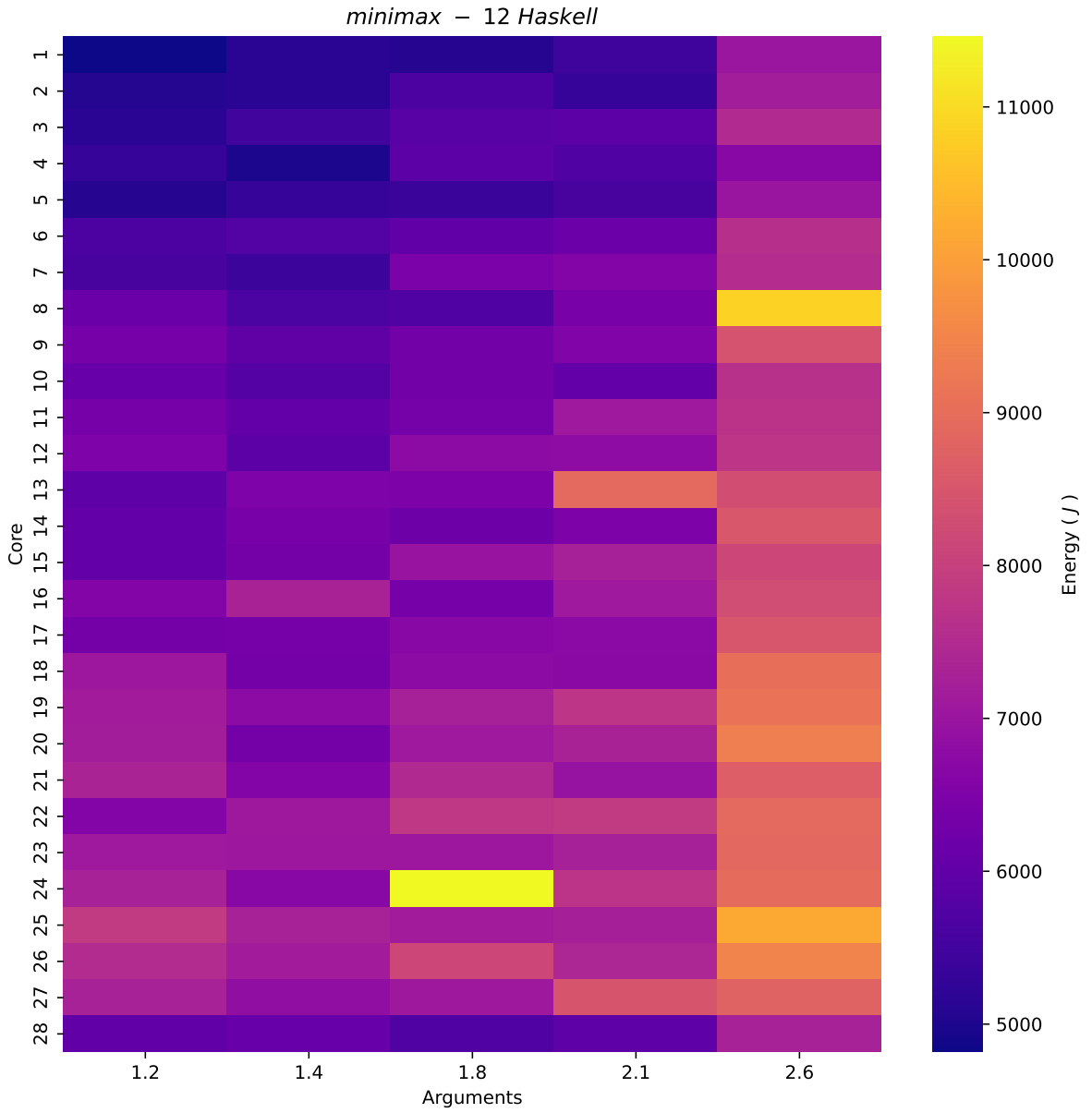

matmult − 2500 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
950
900
850
800
750
700
650
600
550

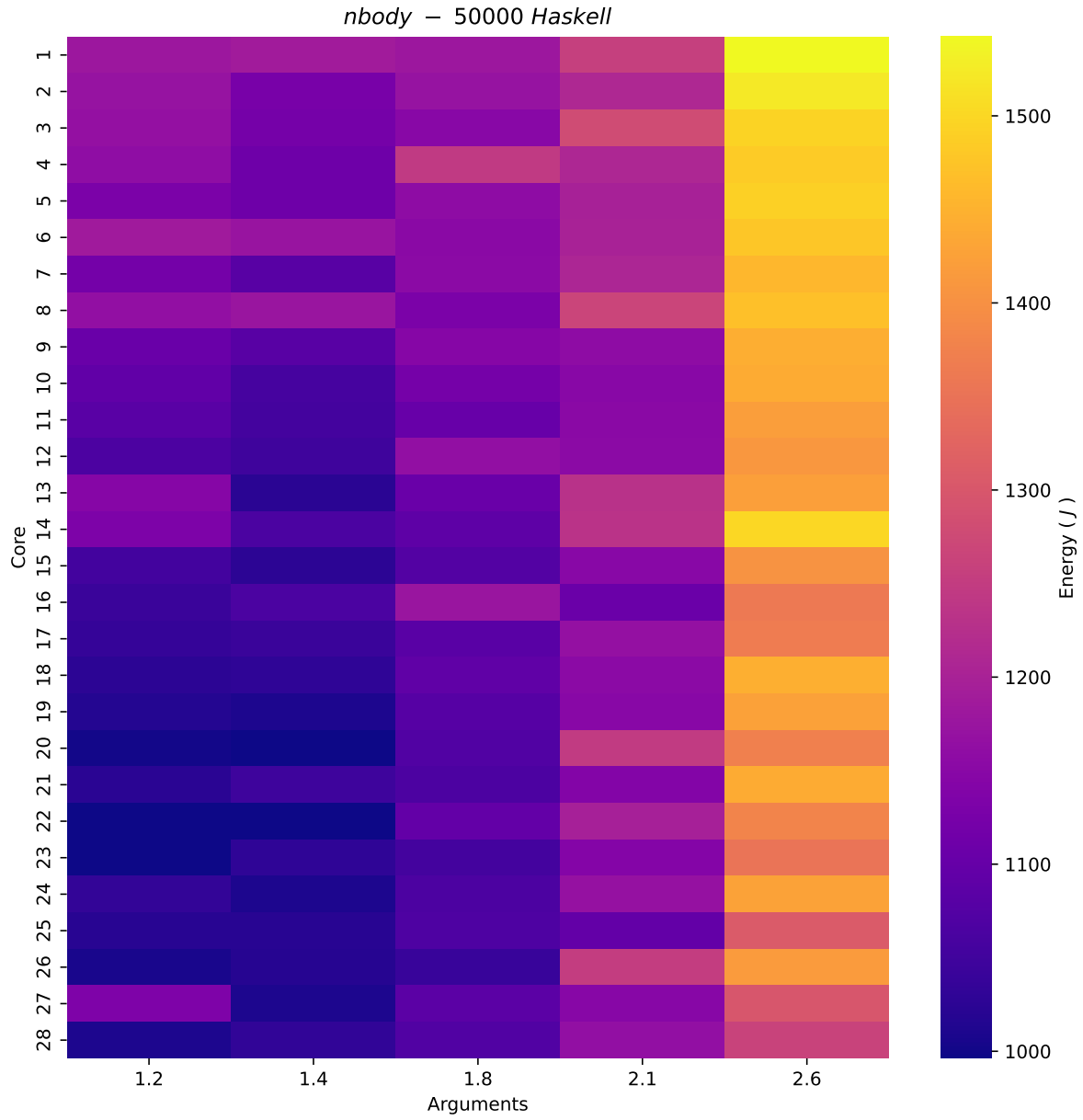

matmult − 3000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1600
1500
1400
1300
1200
1100
1000

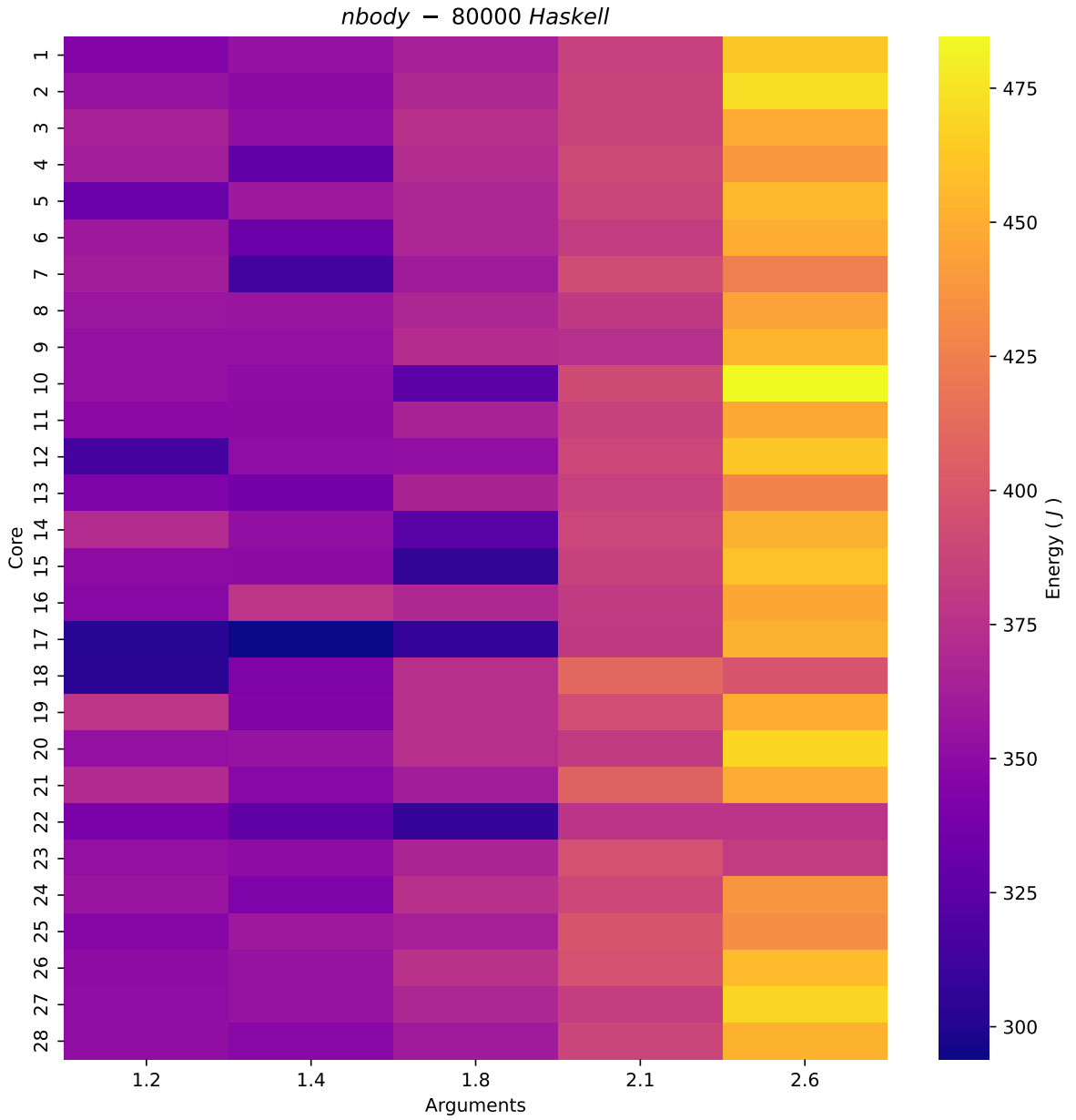

minimax − 6 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
80
75
70
65
60
55
50
45
40

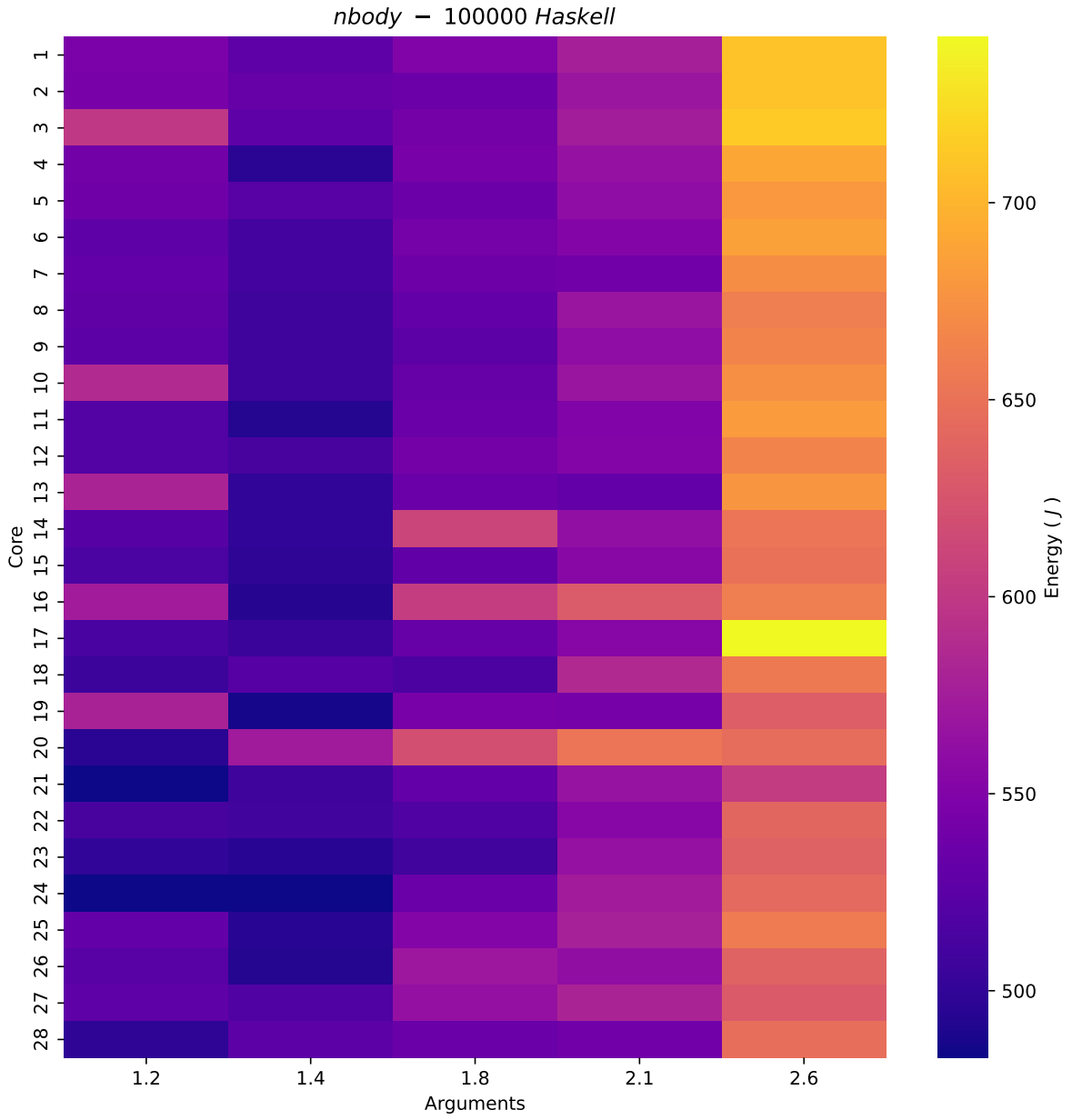

minimax − 8 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1100
1000
900
800
700
600

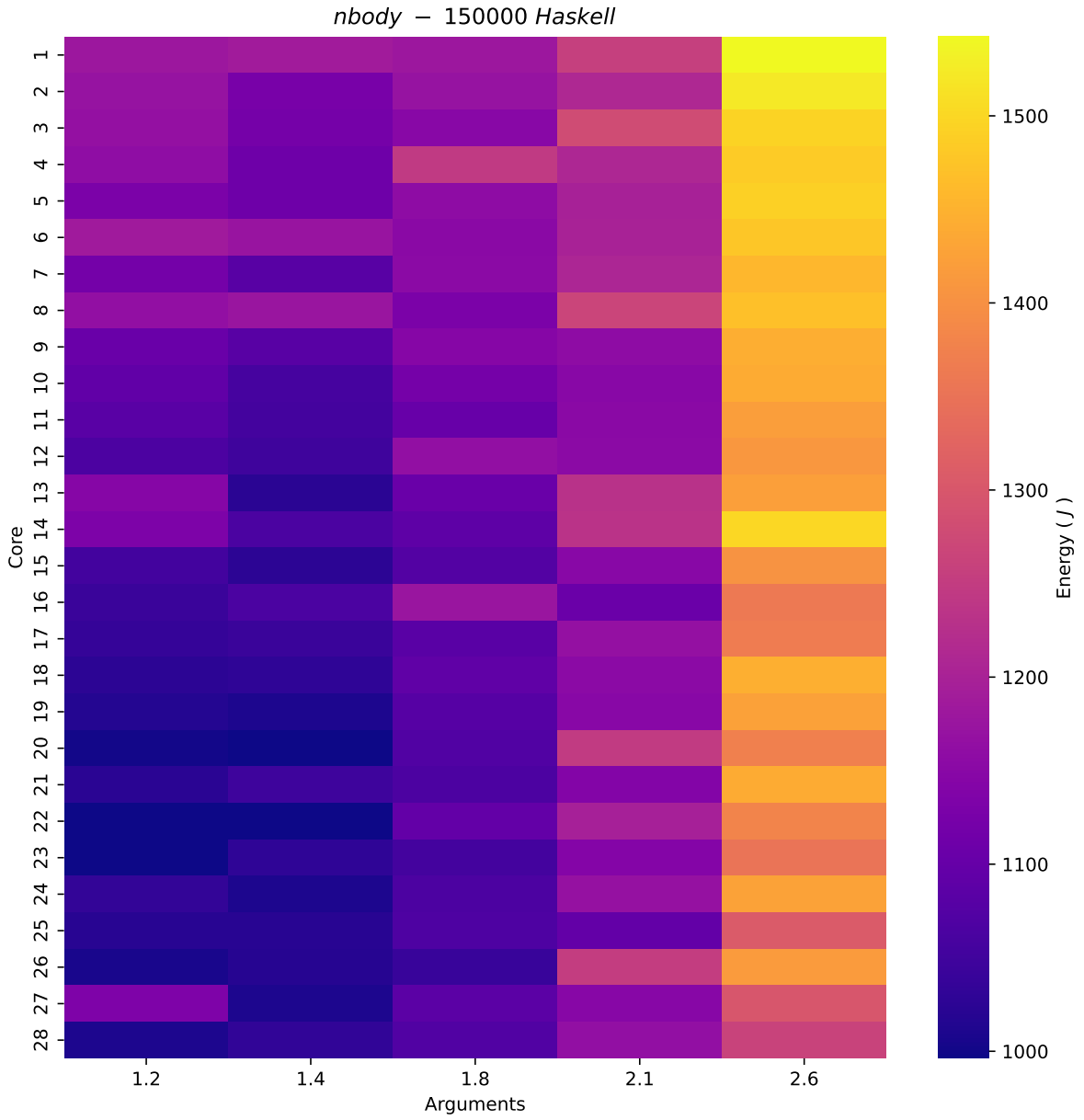

minimax − 10 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
6500
6000
5500
5000
4500
4000
3500

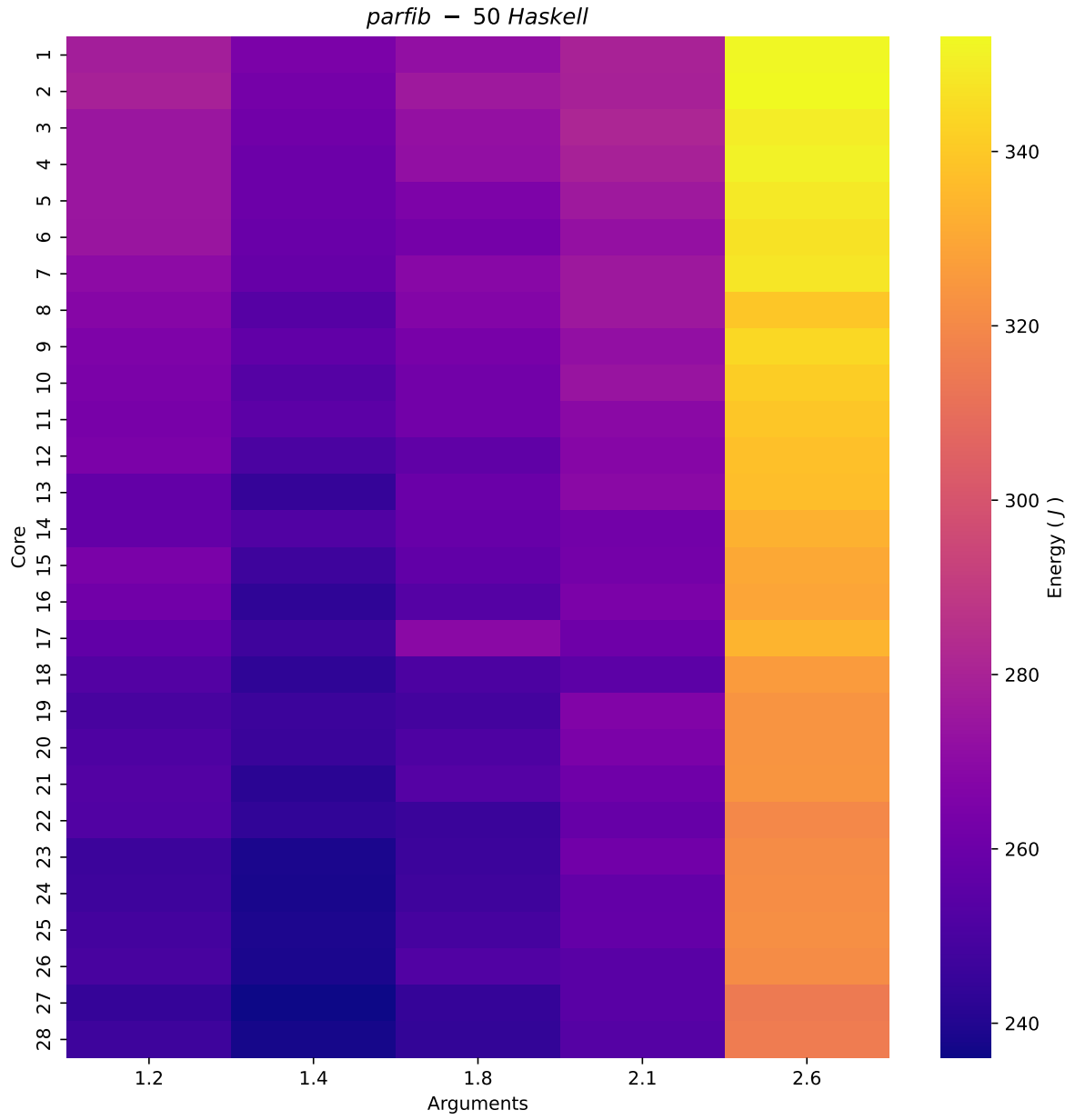

minimax − 12 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
11000
10000
9000
8000
7000
6000
5000
Energy (J)

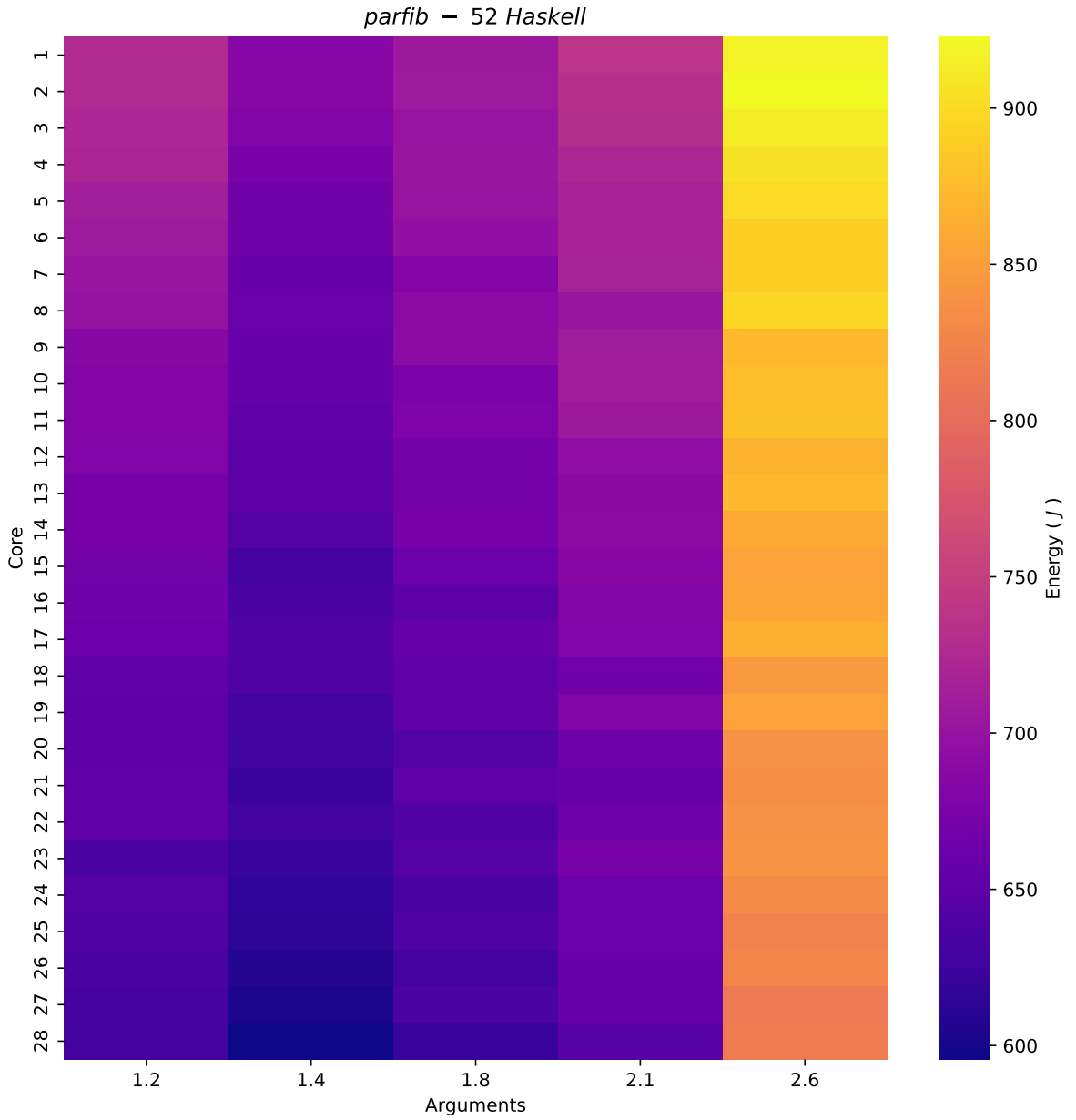

nbody − 50000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
1500
1400
1300
1200
1100
1000
Energy (J)

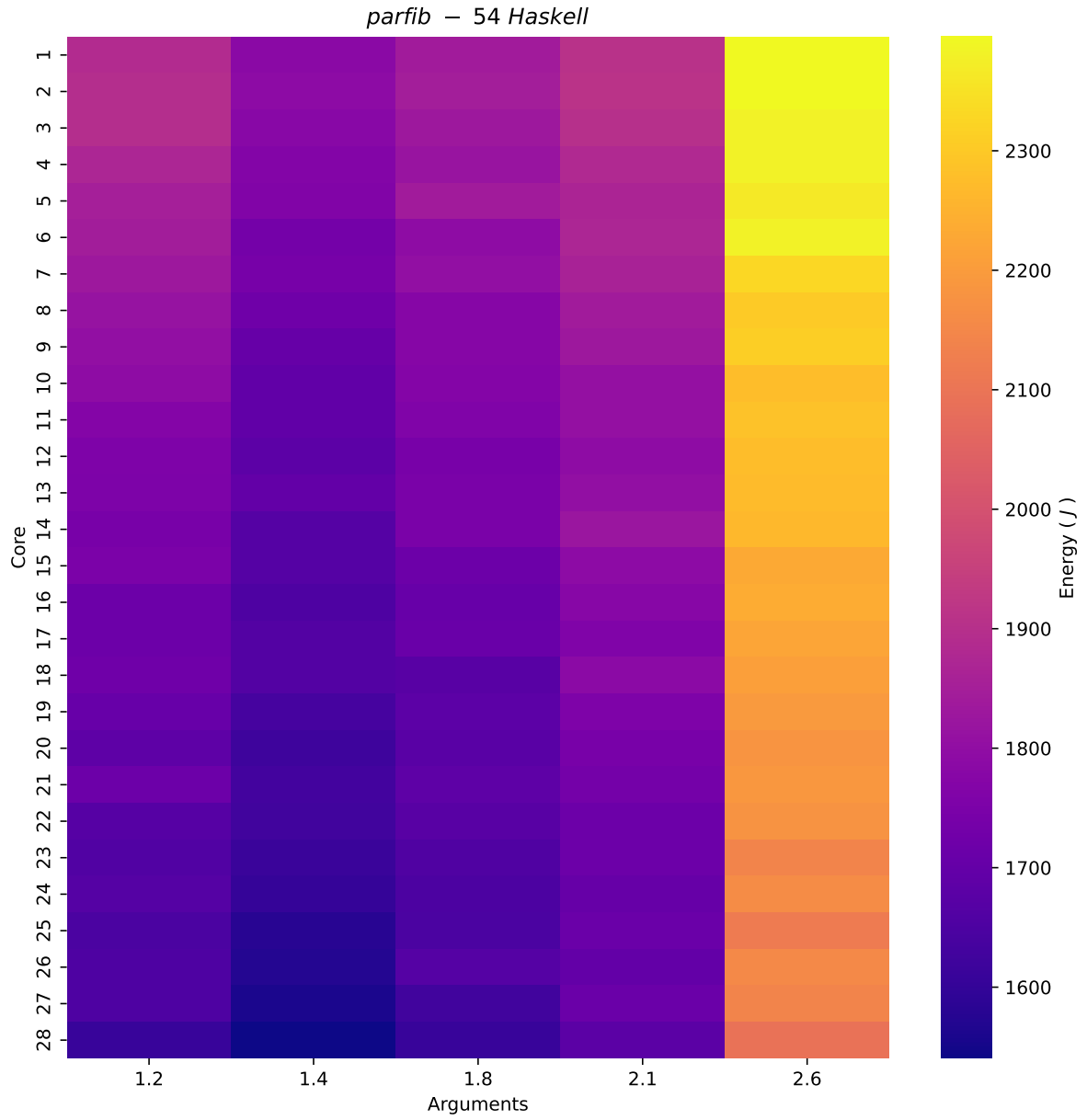

nbody – 80000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
475
450
425
400
375
350
325
300

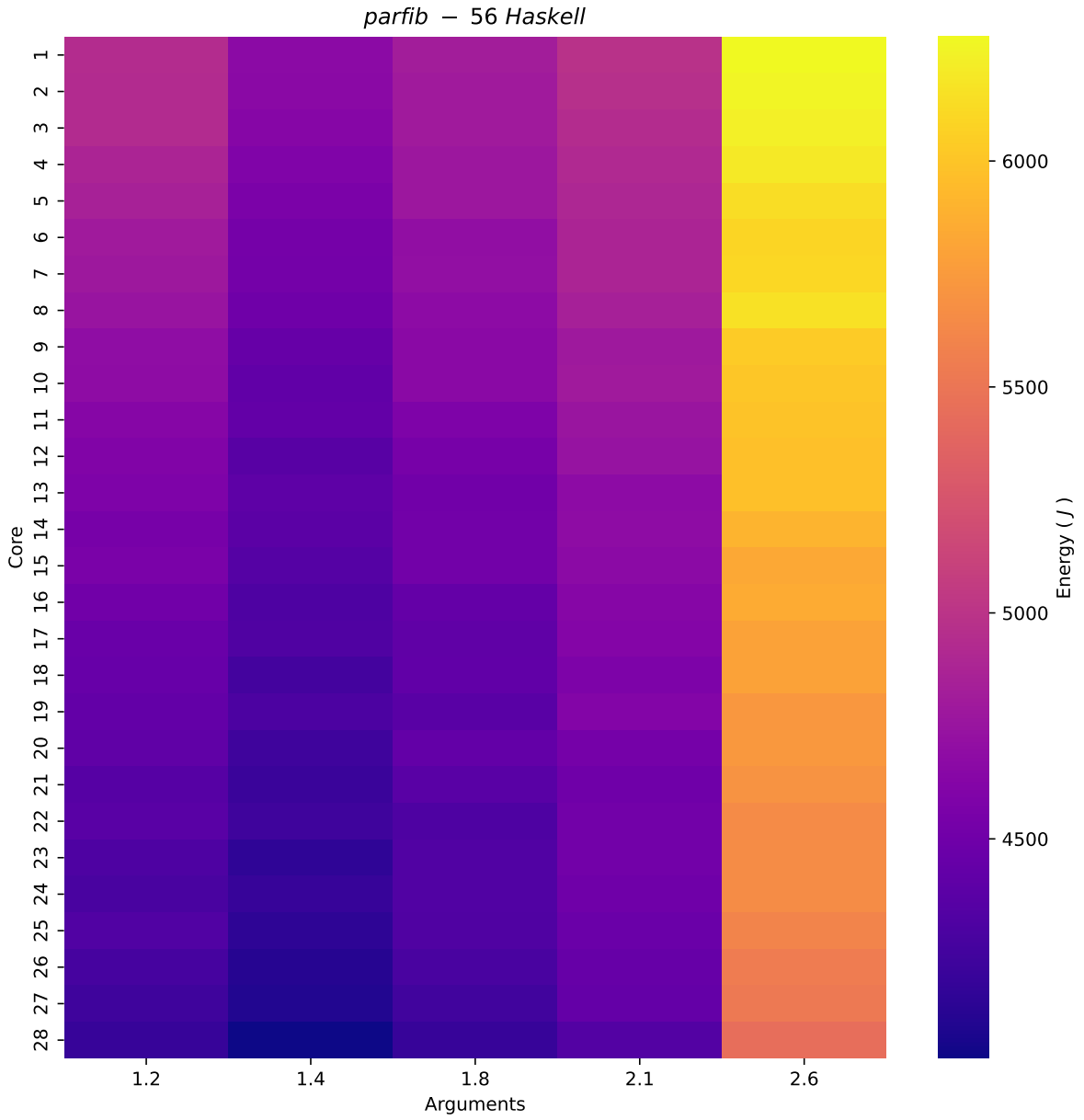

nbody – 100000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
700
650
600
550
500

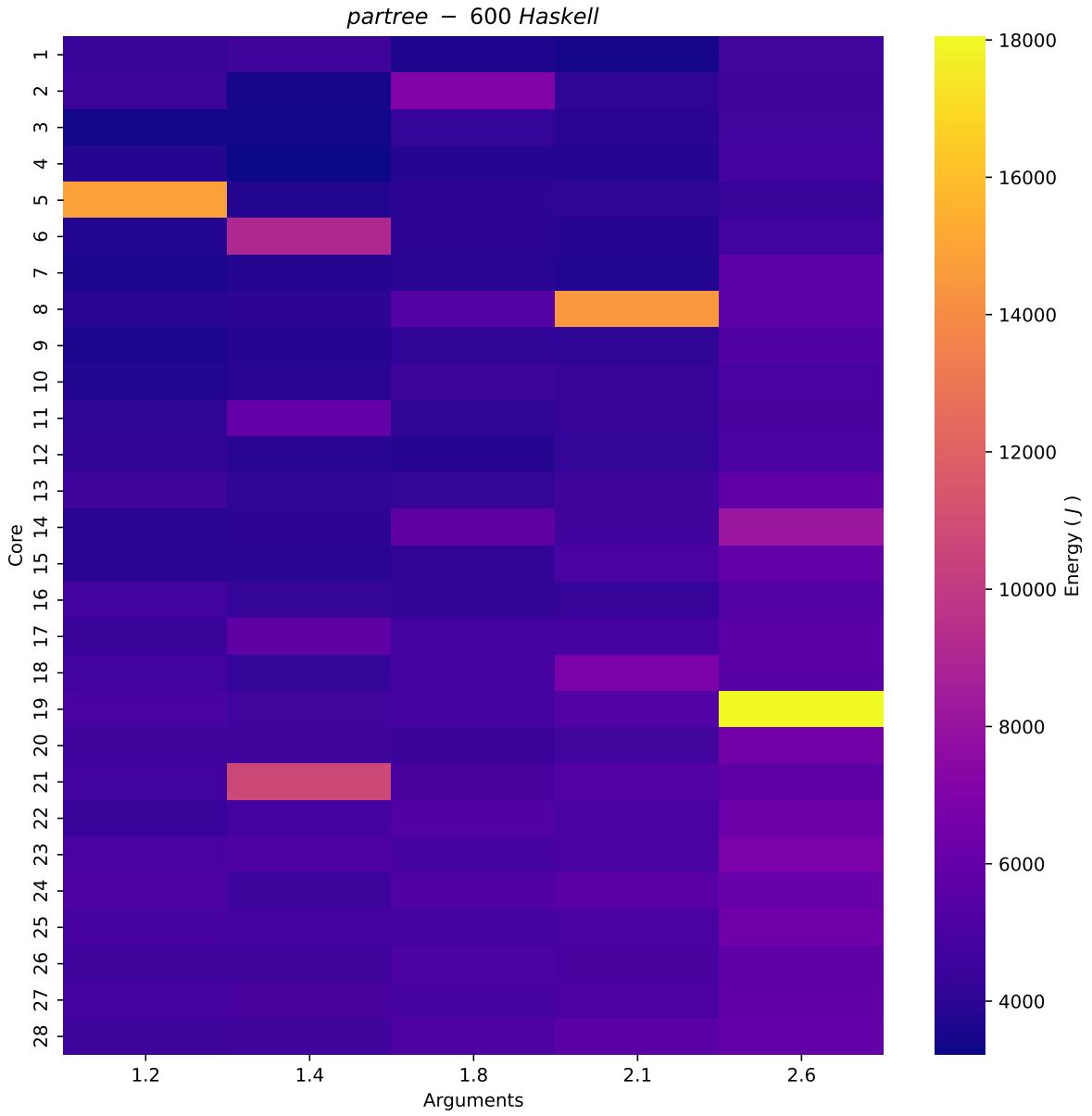

nbody − 150000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1500
1400
1300
1200
1100
1000

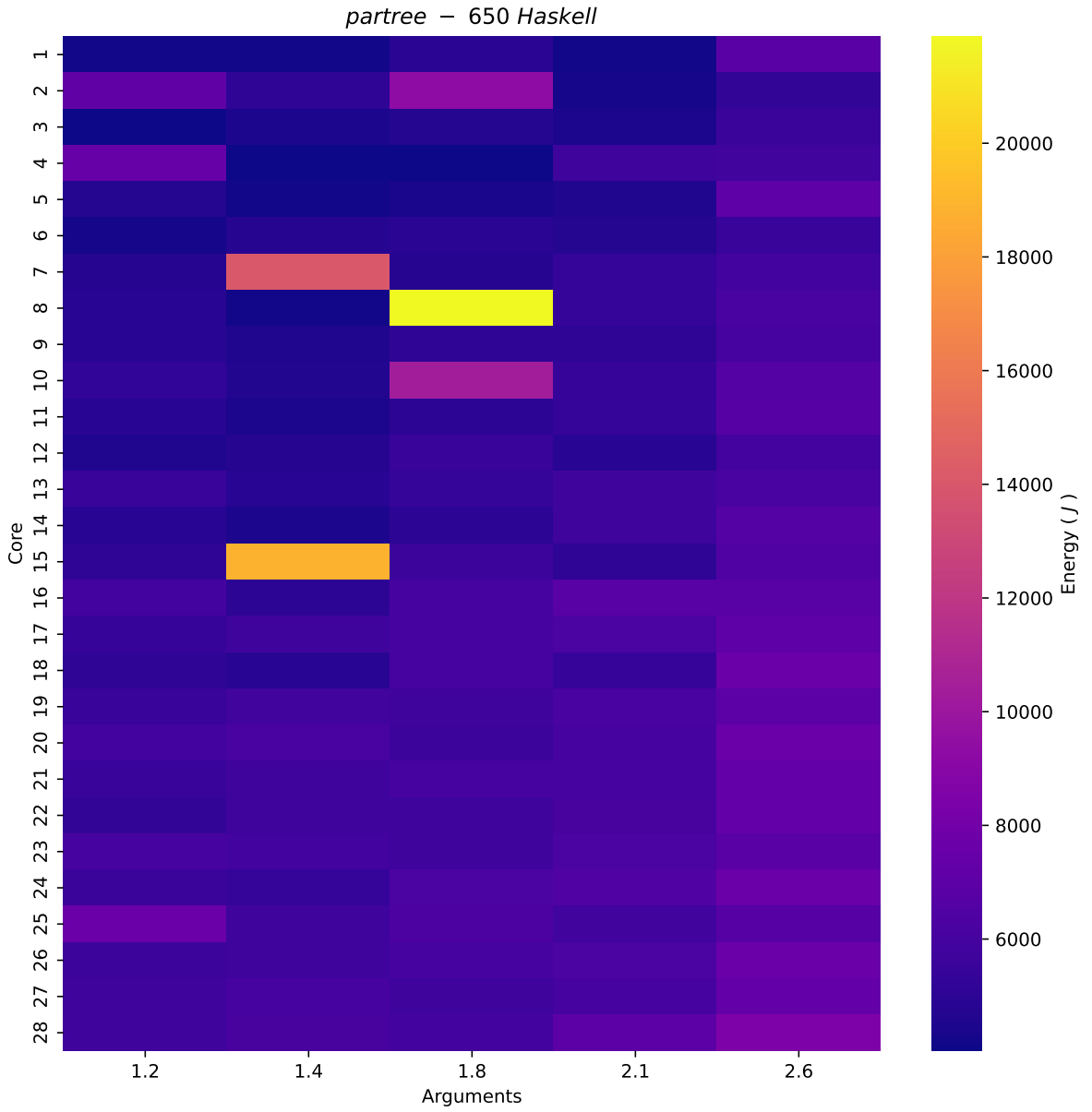

parfib − 50 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
340
320
300
280
260
240

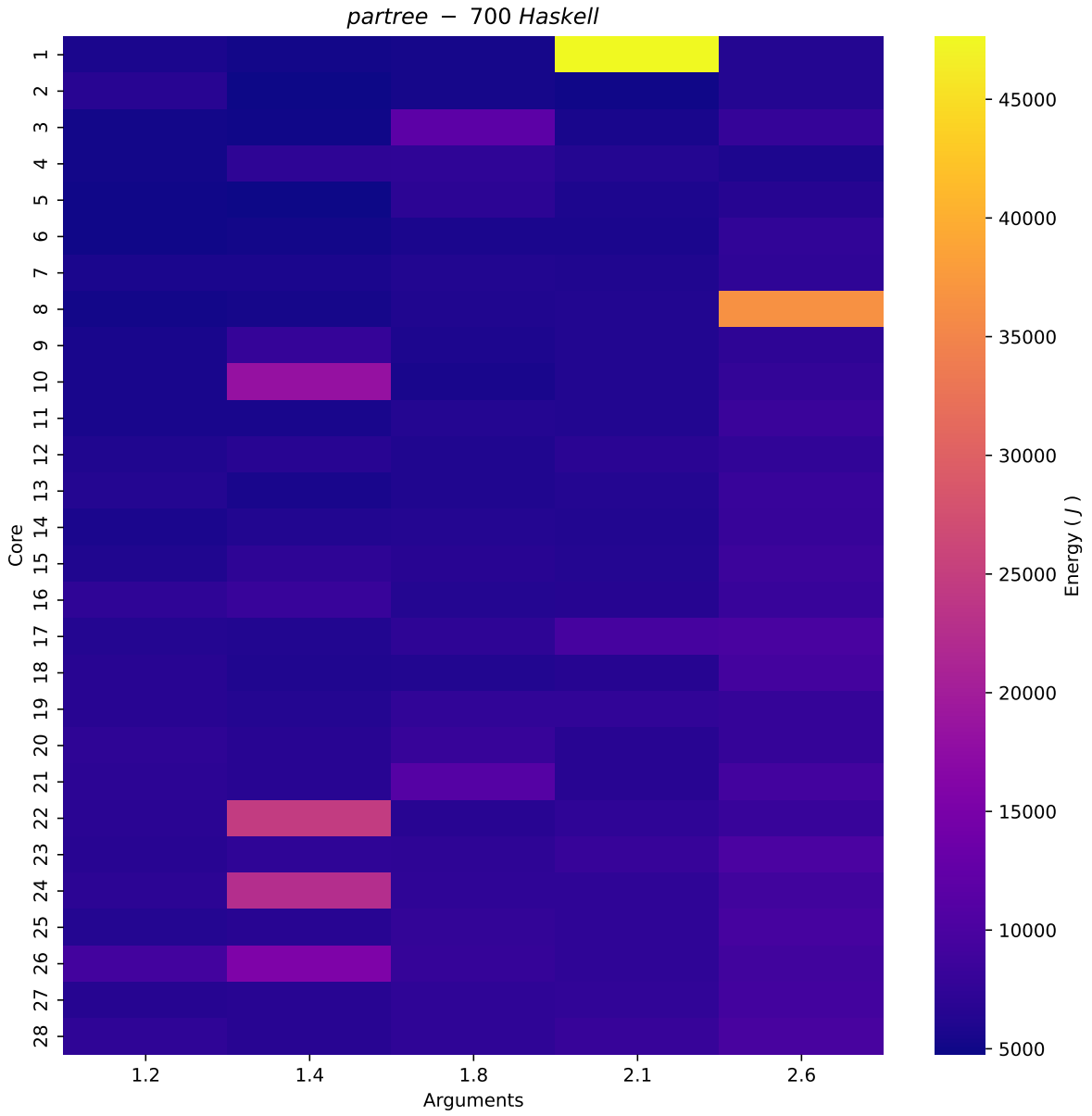

parfib − 52 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
900
850
800
750
700
650
600
Energy (J)

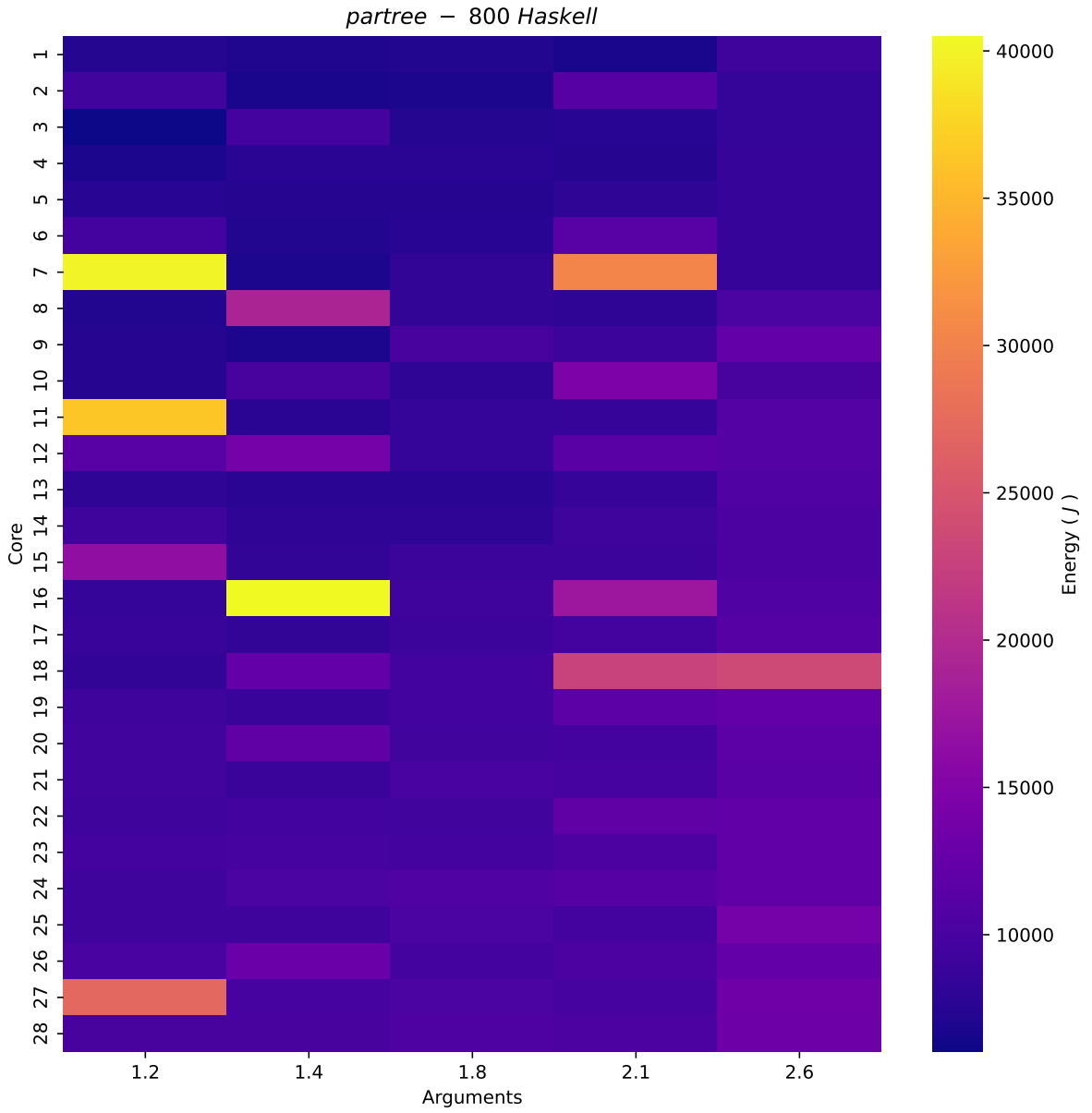

parfib − 54 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
2300
2200
2100
2000
1900
1800
1700
1600

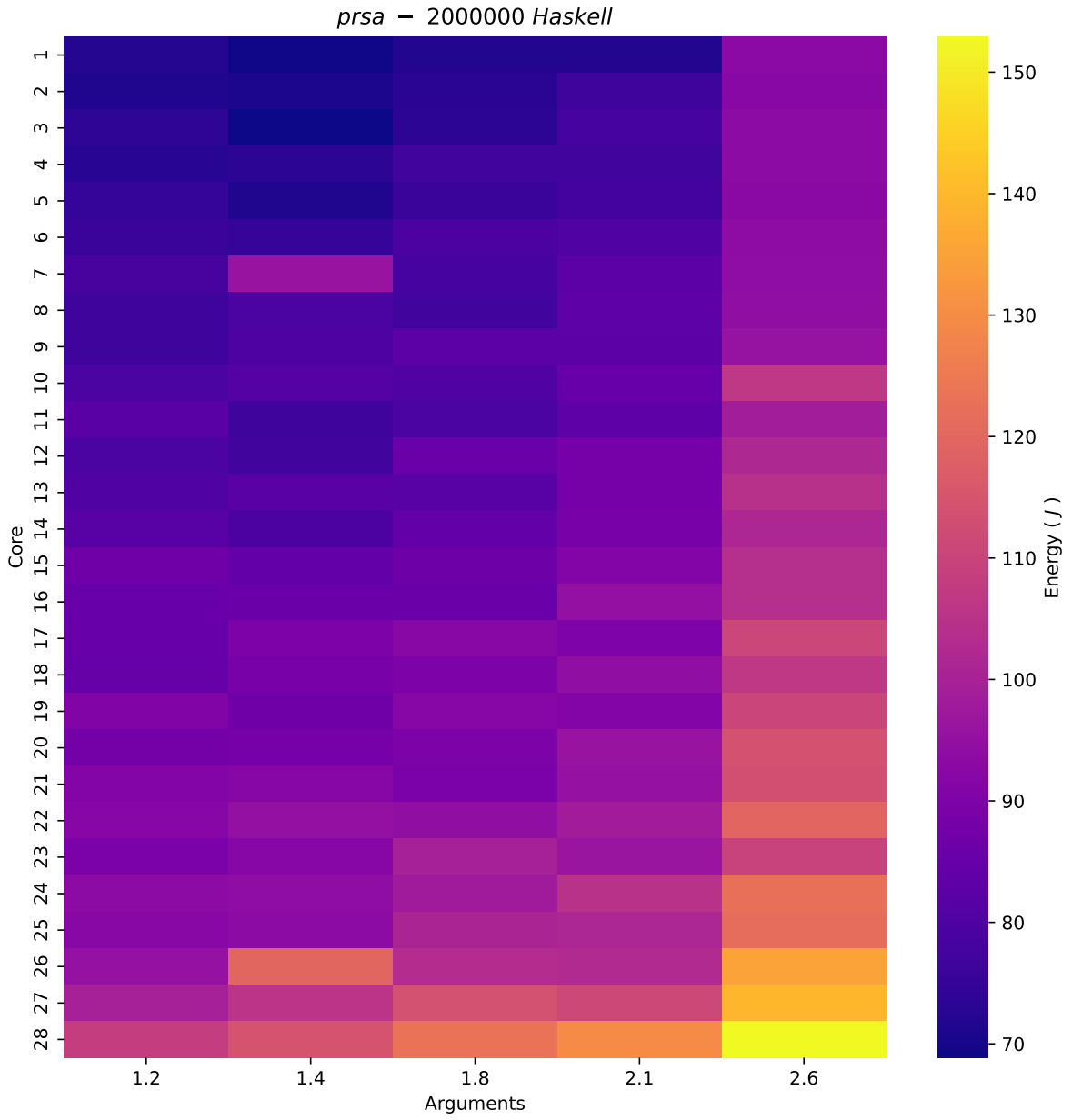

parfib − 56 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
6000
5500
5000
4500
Energy (J)

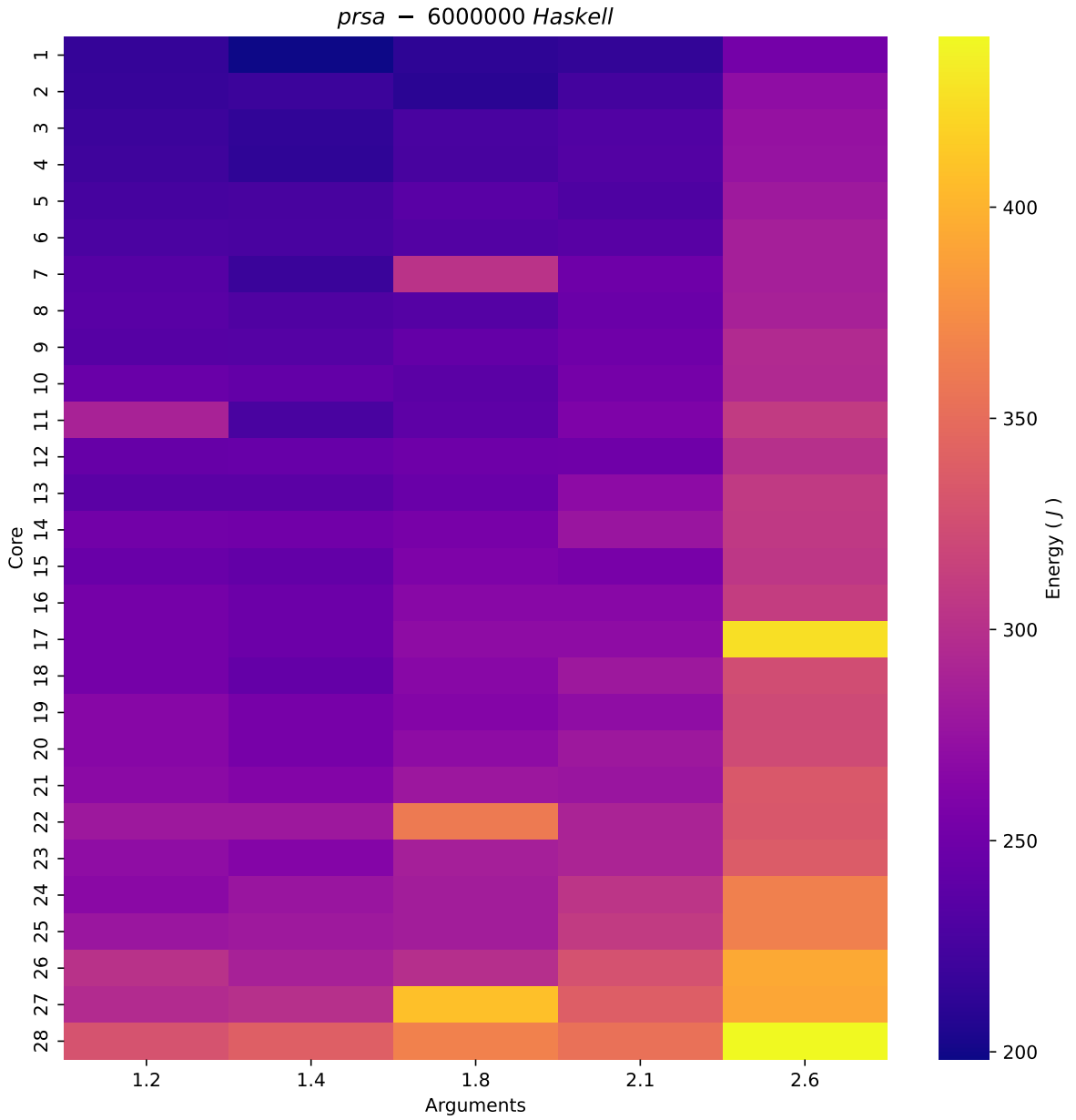

partree − 600 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
18000
16000
14000
12000
10000
8000
6000
4000
Energy (J)

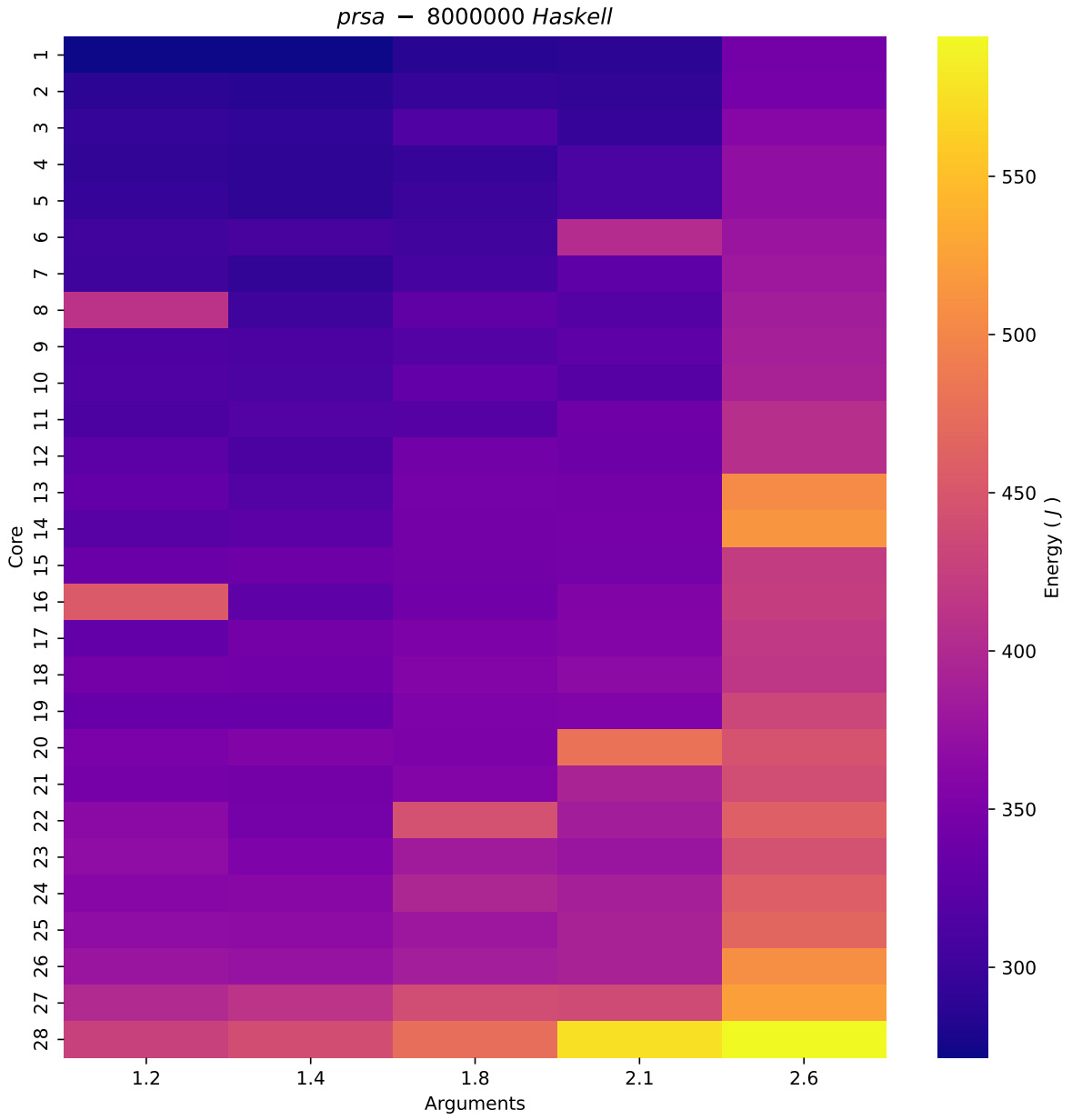

partree − 650 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
20000
18000
16000
14000
12000
10000
8000
6000

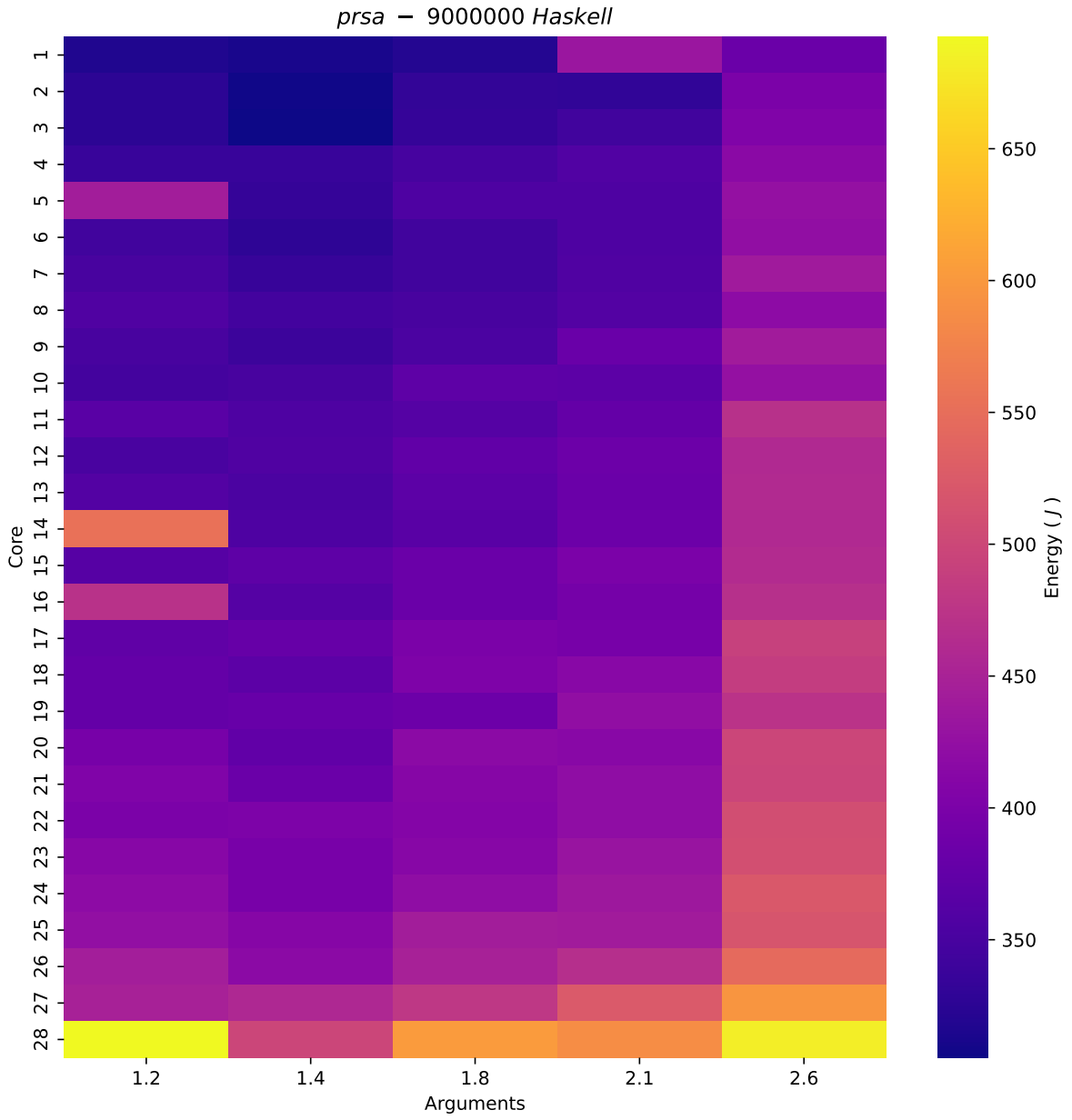

partree − 700 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
45000
40000
35000
30000
25000
20000
15000
10000
5000
Energy (J)

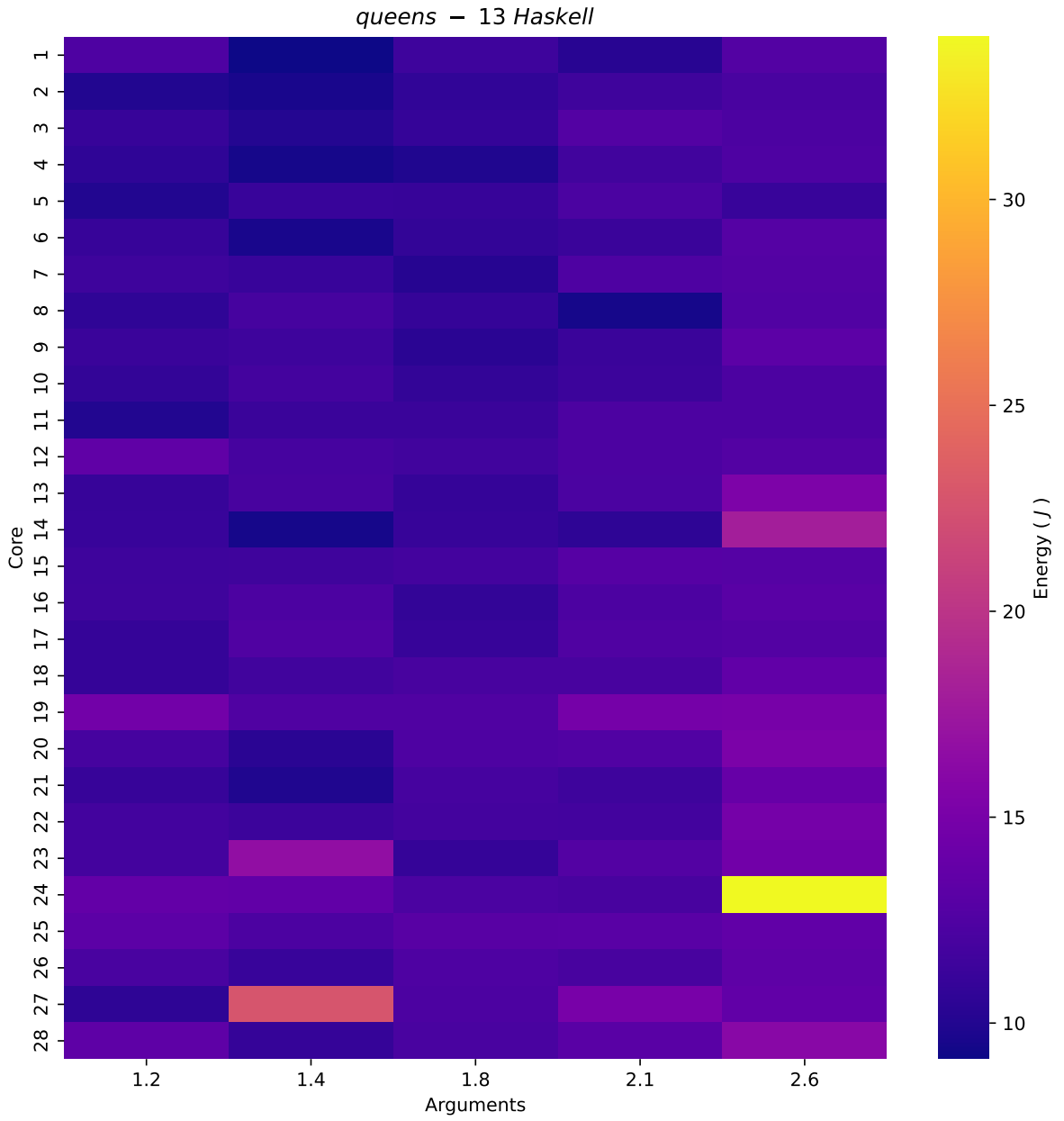

partree − 800 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
40000
35000
30000
25000
20000
15000
10000
Energy (J)

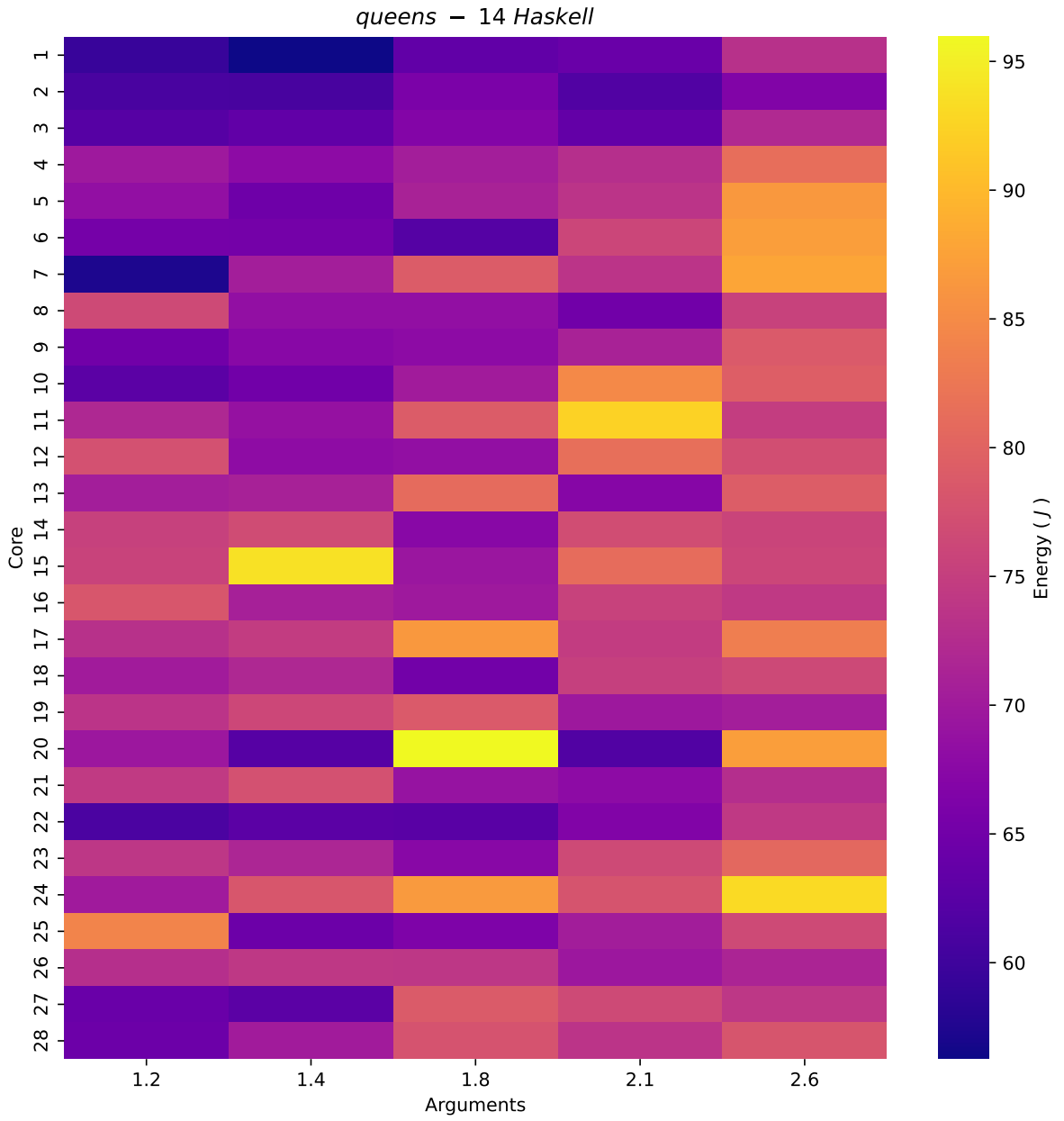

prsa − 2000000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
150
140
130
120
110
100
90
80
70

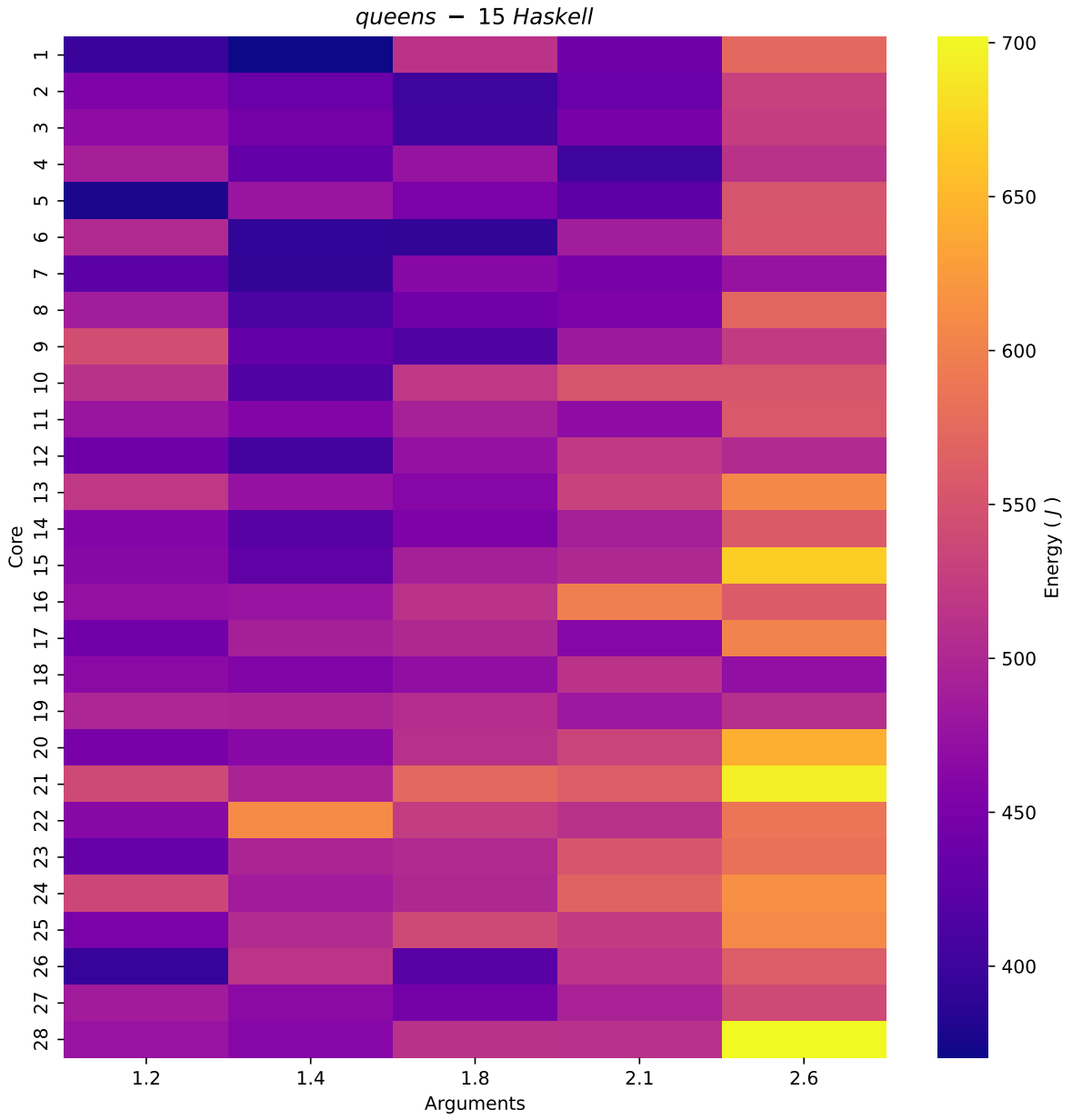

prsa – 6000000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
400
350
300
250
200

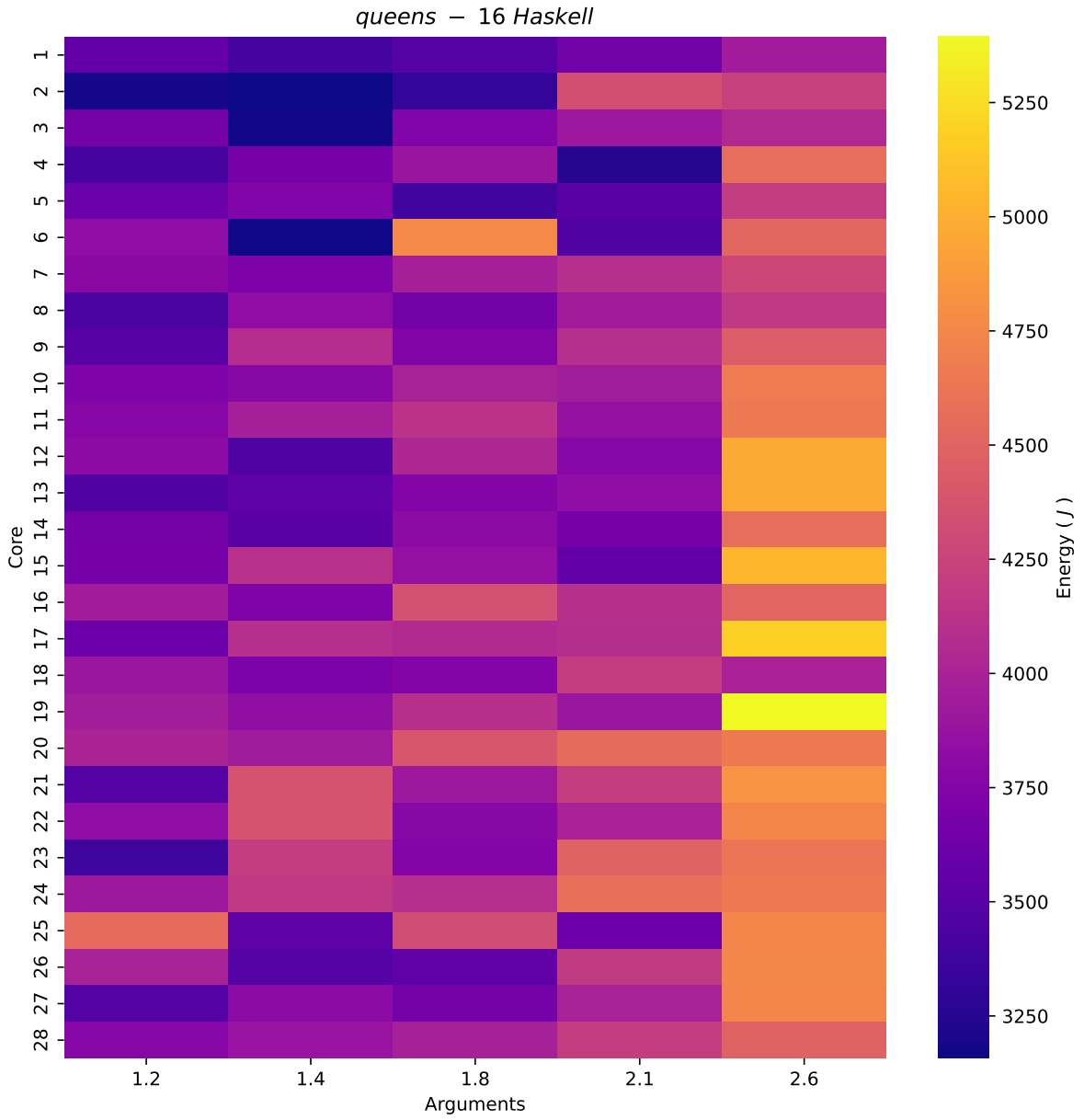

prsa – 8000000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
550
500
450
400
350
300

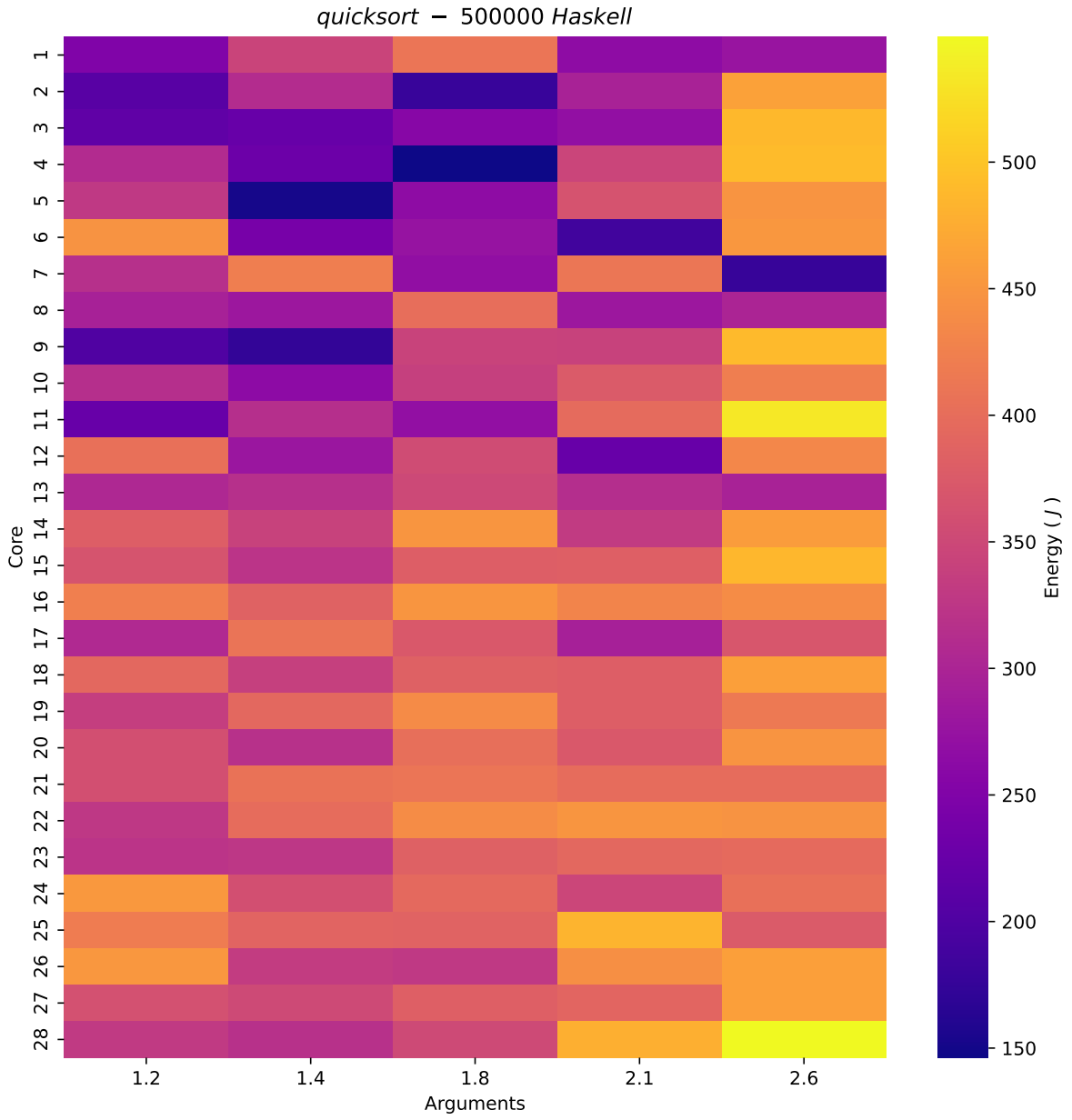

prsa – 9000000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
650
600
550
500
450
400
350
Energy (J)

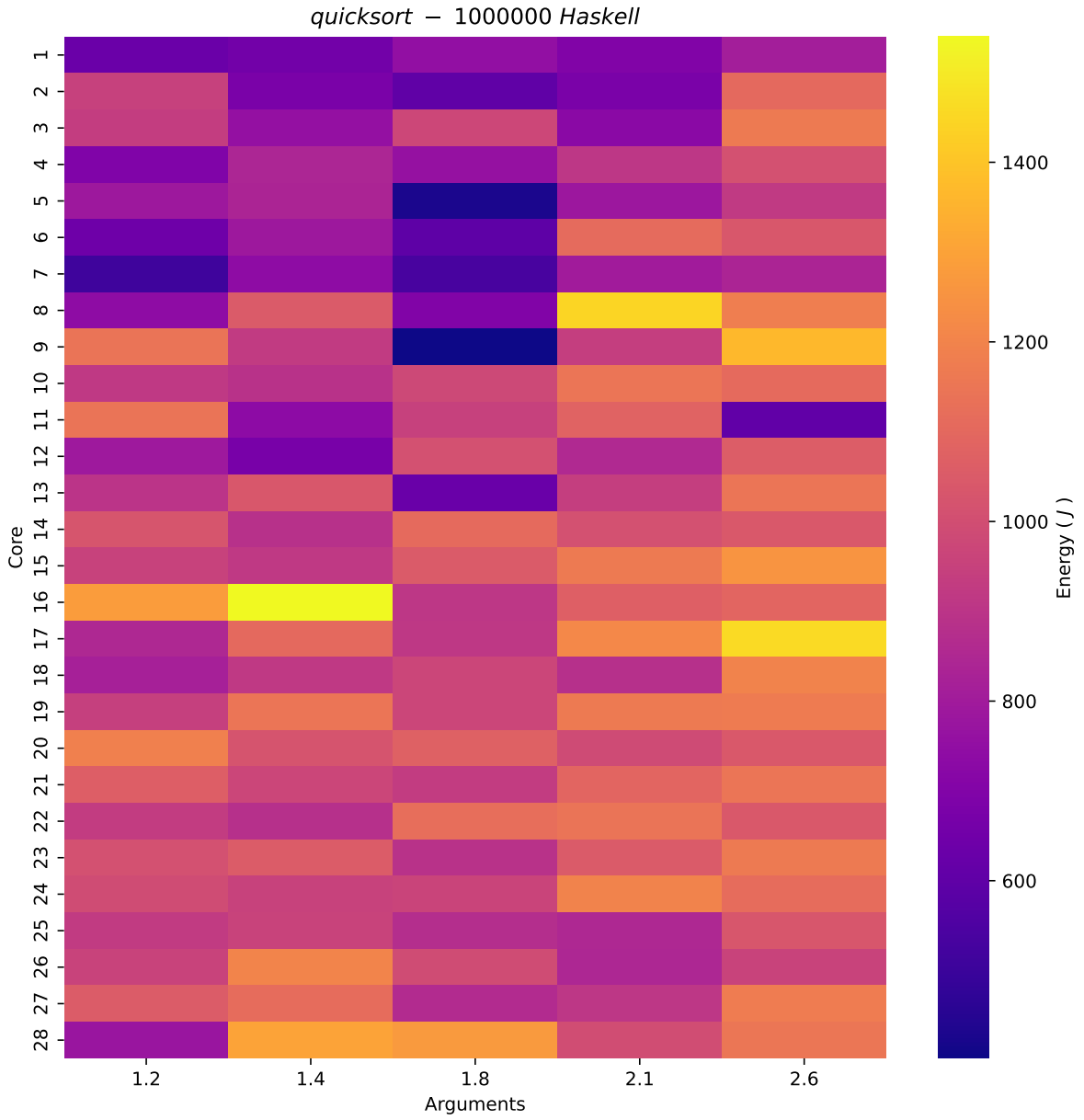

queens − 13 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
10
15
20
25
30

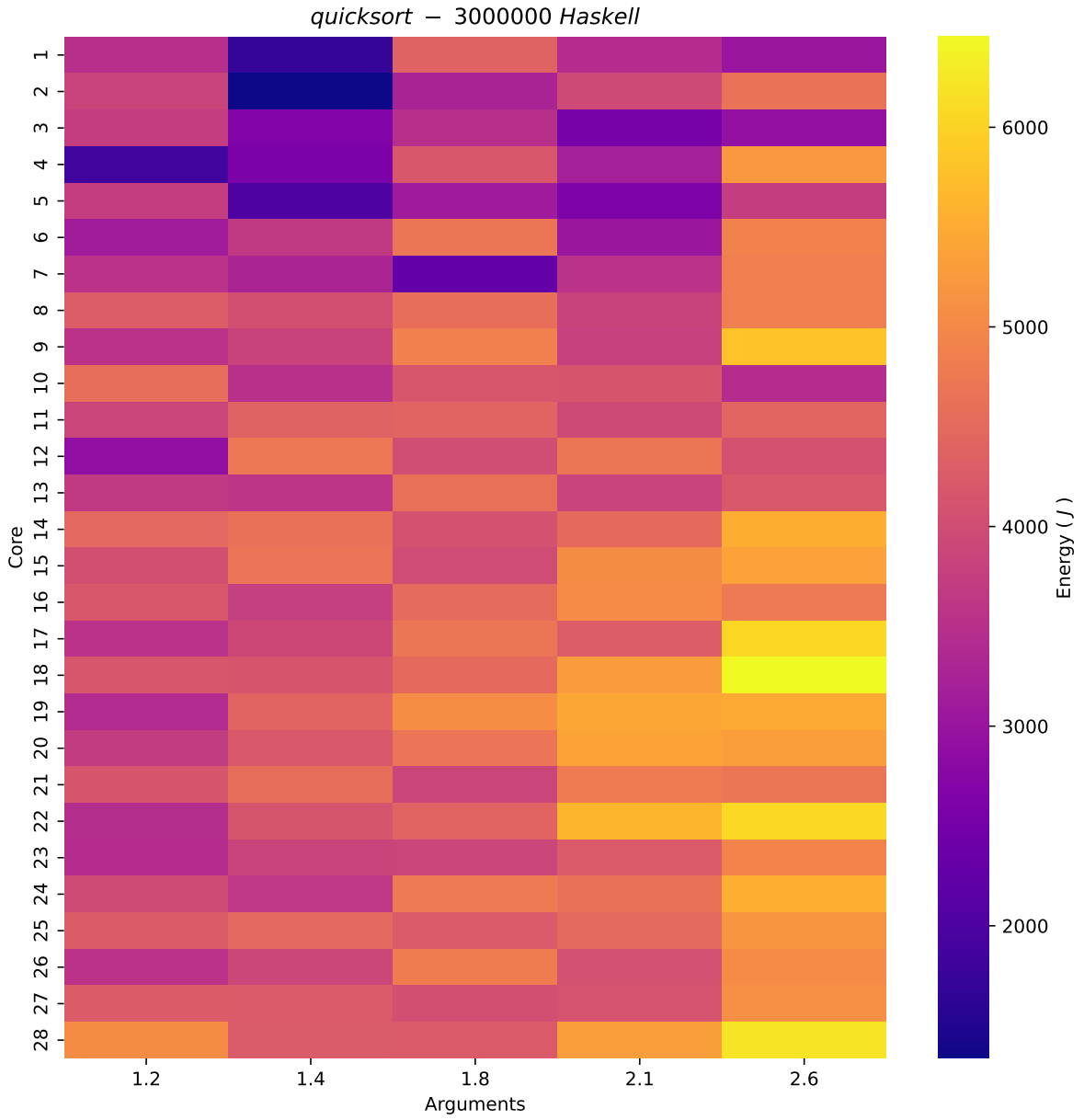

queens − 14 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
95
90
85
80
75
70
65
60
Energy (J)

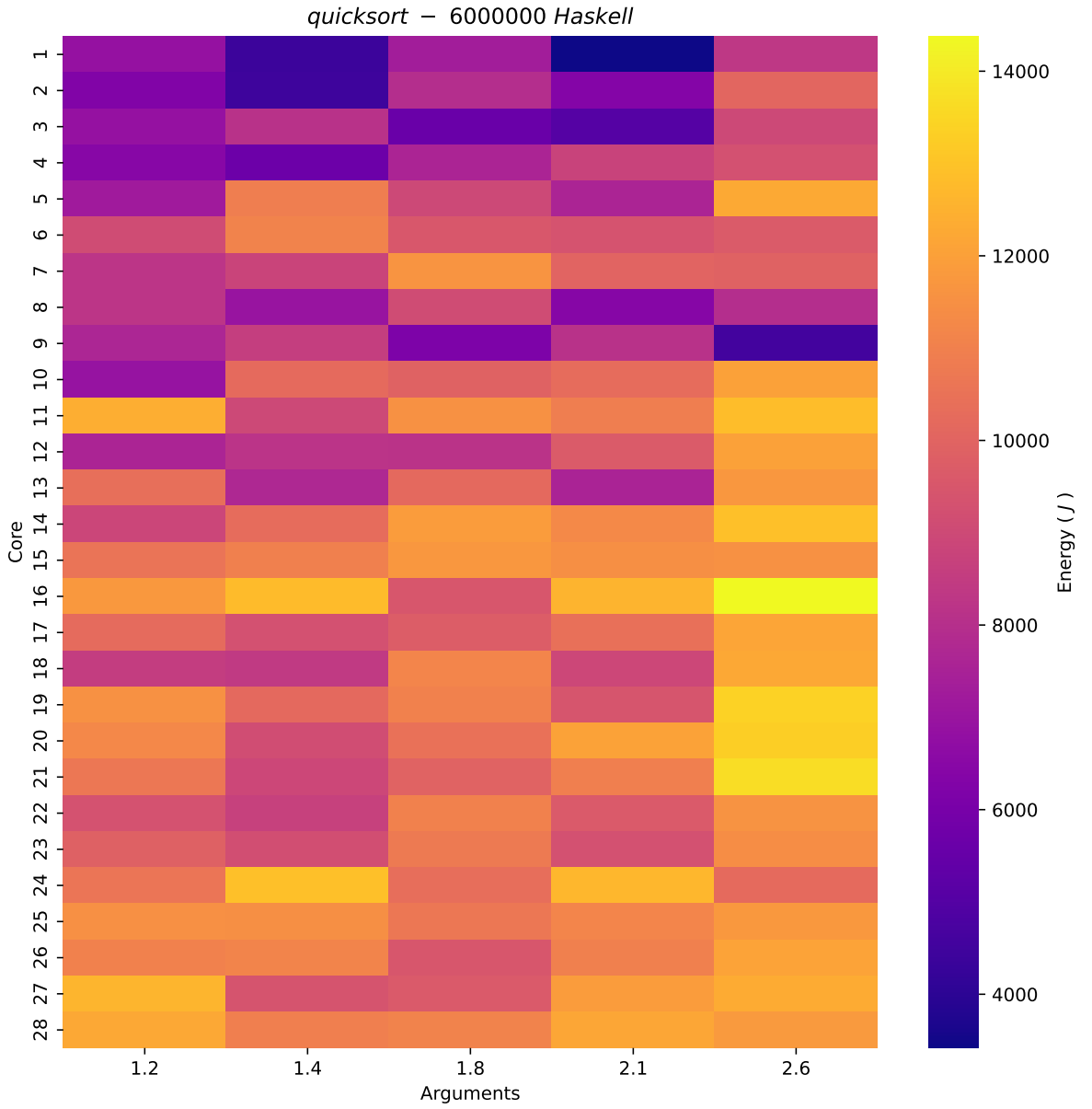

queens − 15 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
700
650
600
550
500
450
400
Energy (J)

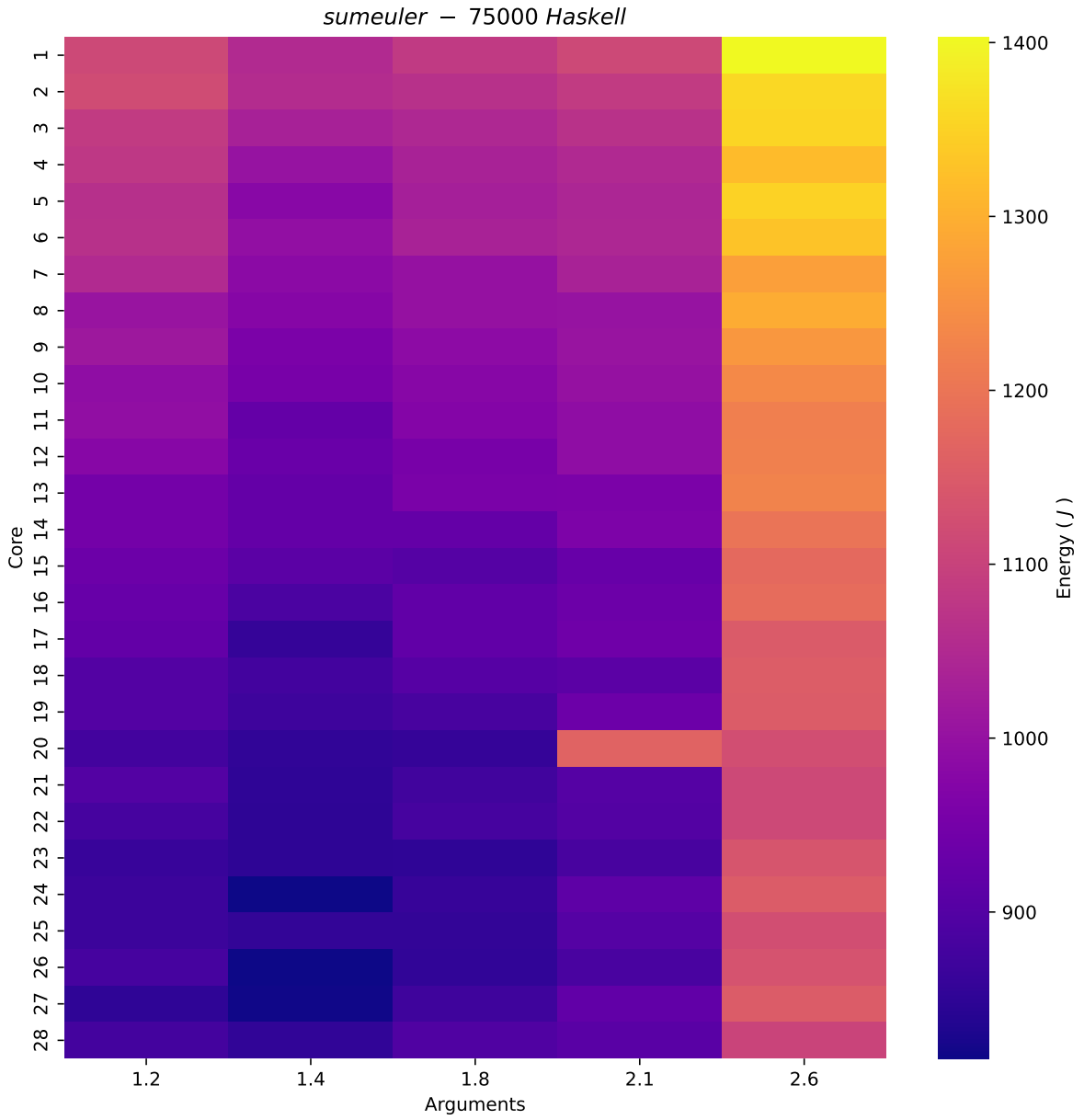

queens − 16 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
5250
5000
4750
4500
4250
4000
3750
3500
3250

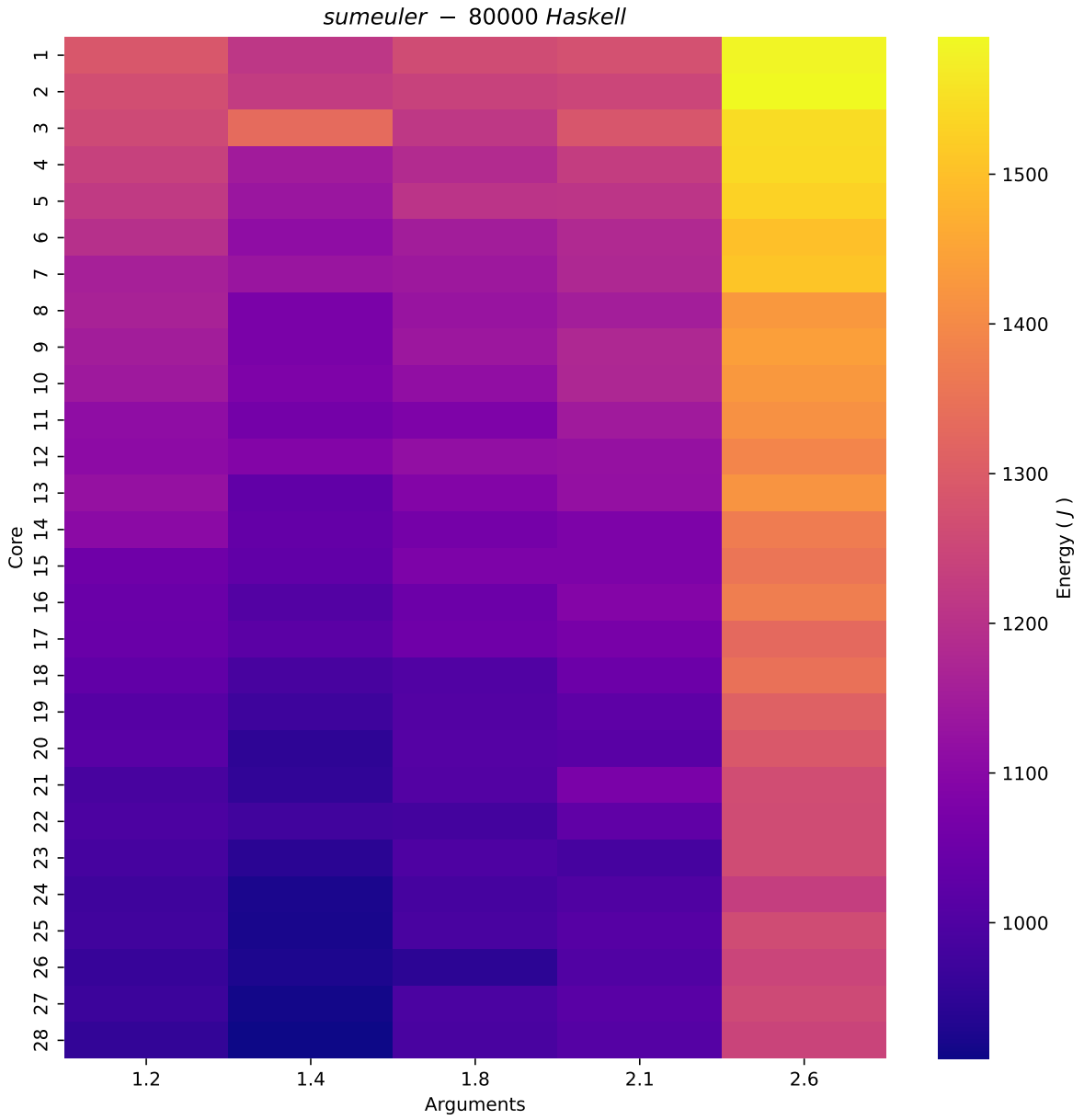

quicksort − 500000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
500
450
400
350
300
250
200
150

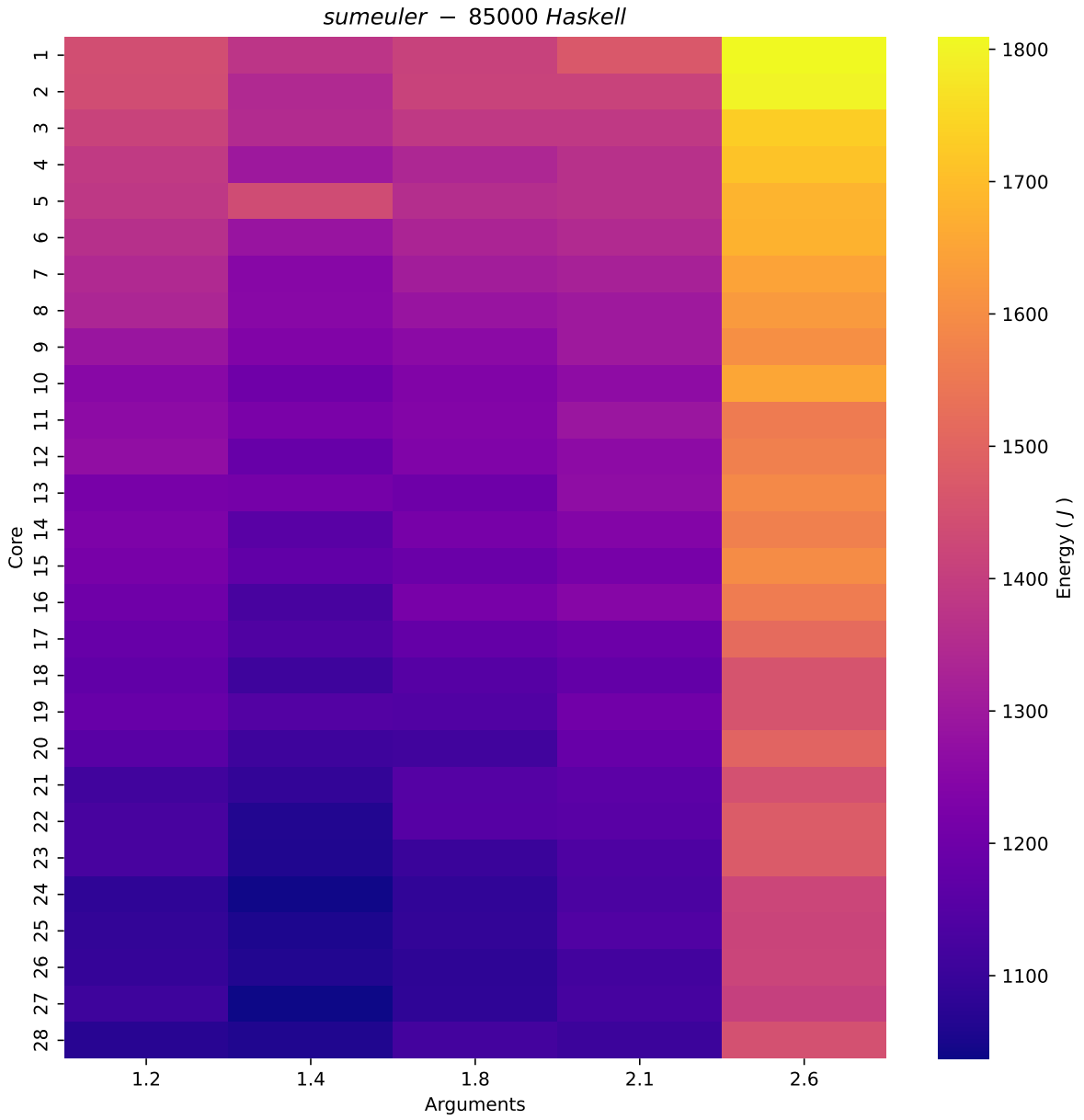

quicksort − 1000000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
1400
1200
1000
800
600
Energy (J)

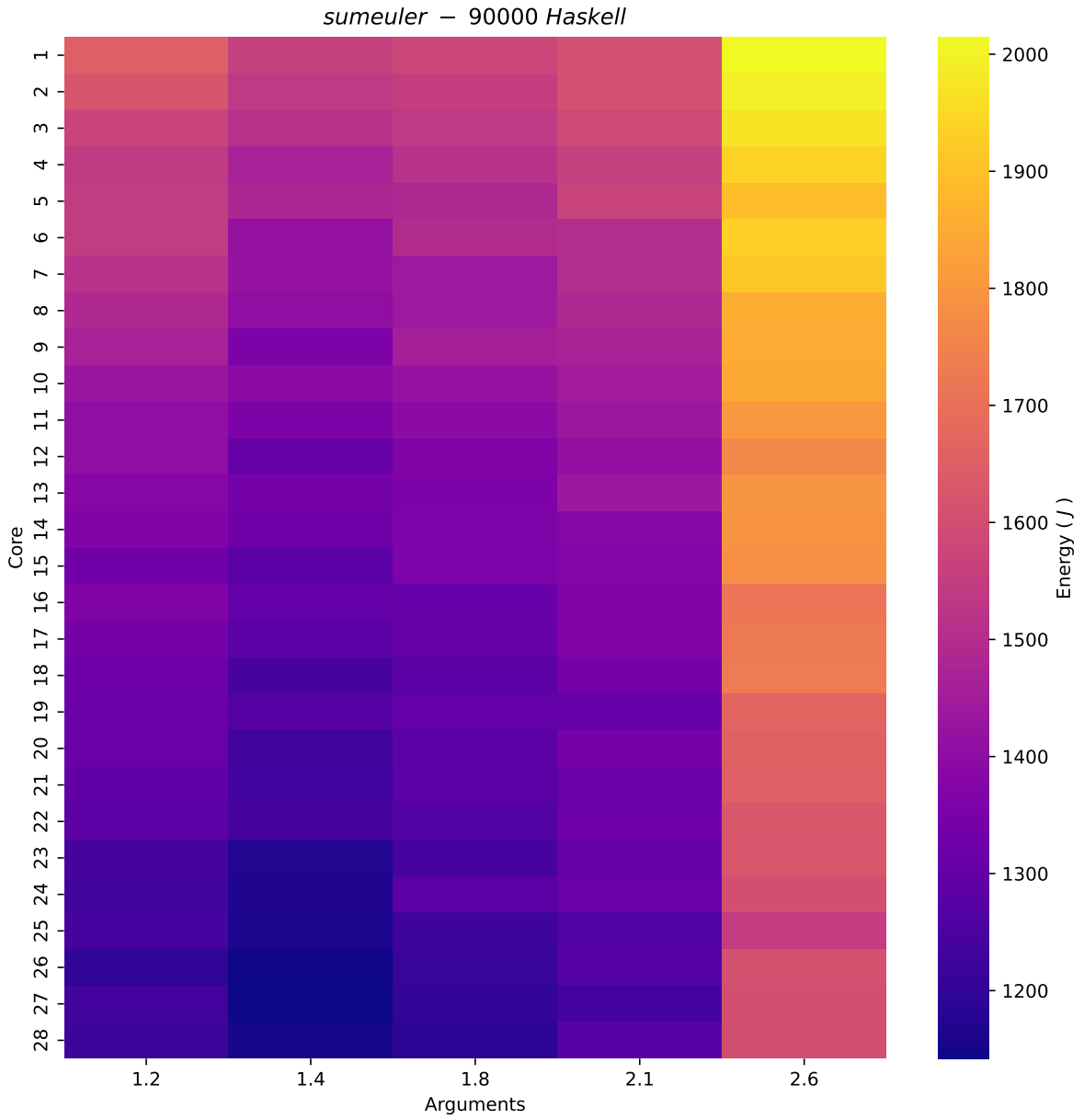

quicksort − 3000000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
6000
5000
4000
3000
2000
Energy (J)

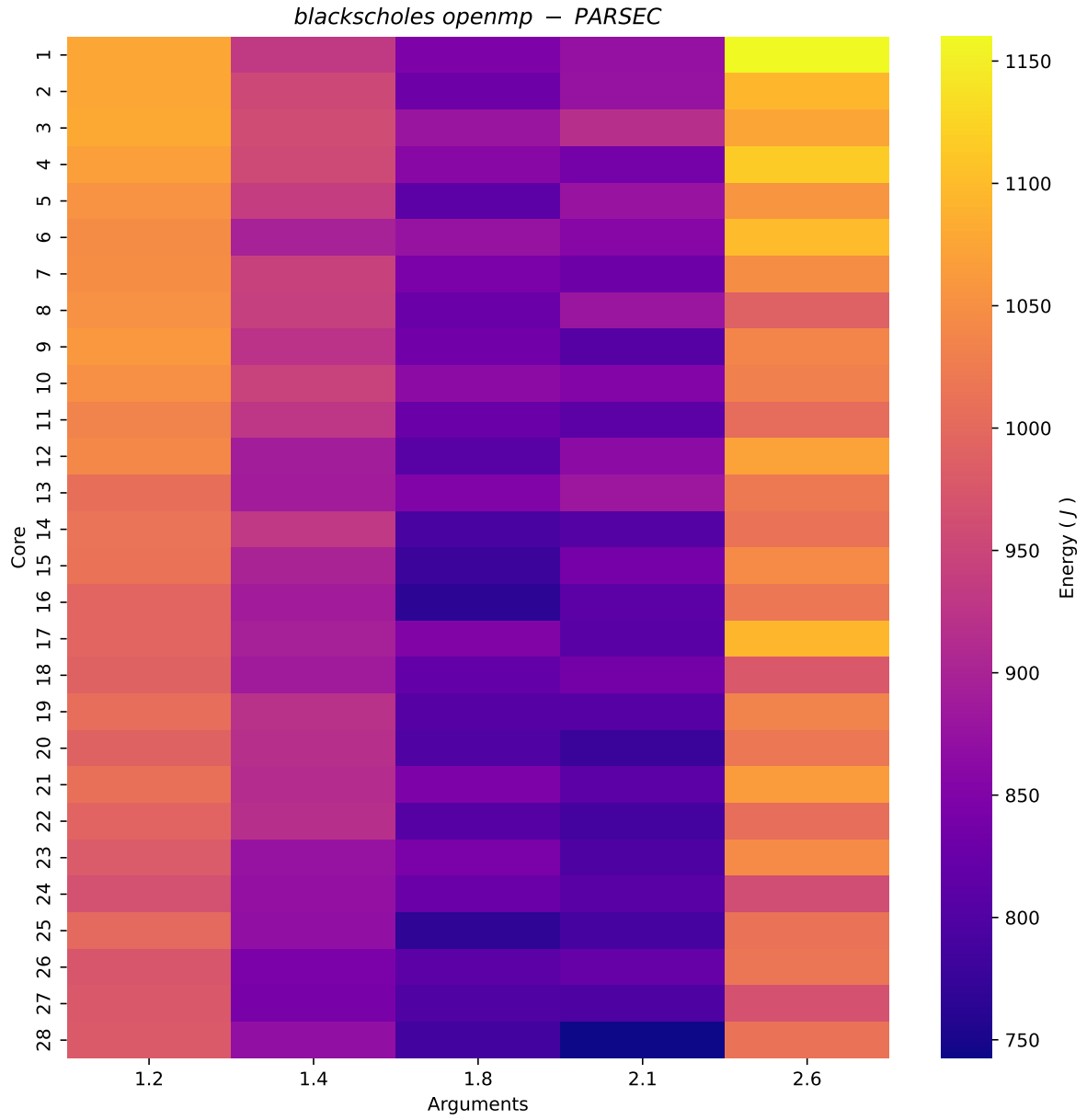

quicksort − 6000000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
14000
12000
10000
8000
6000
4000
Energy (J)

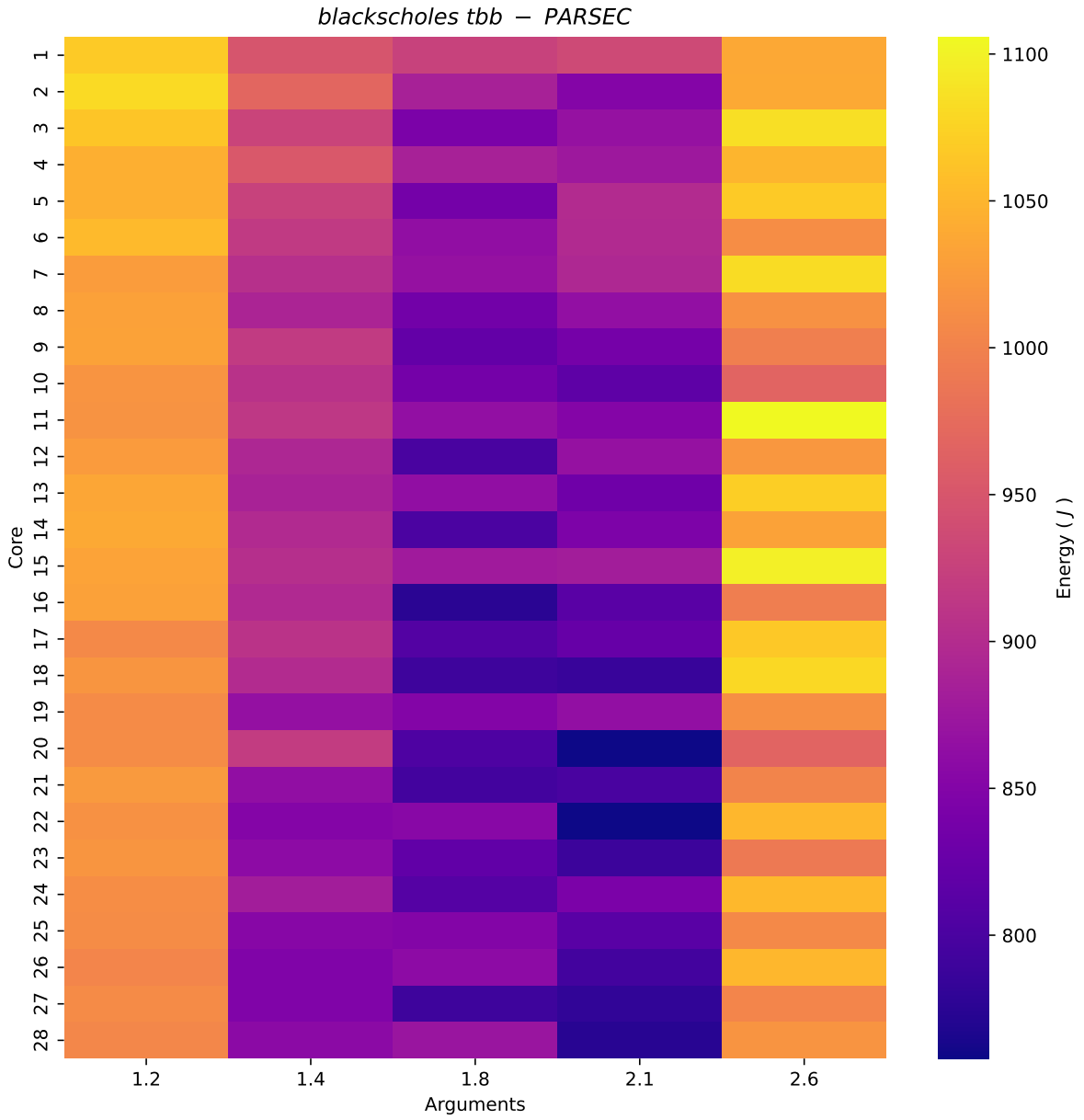

sumeuler − 75000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
1400
1300
1200
1100
1000
900
Energy (J)

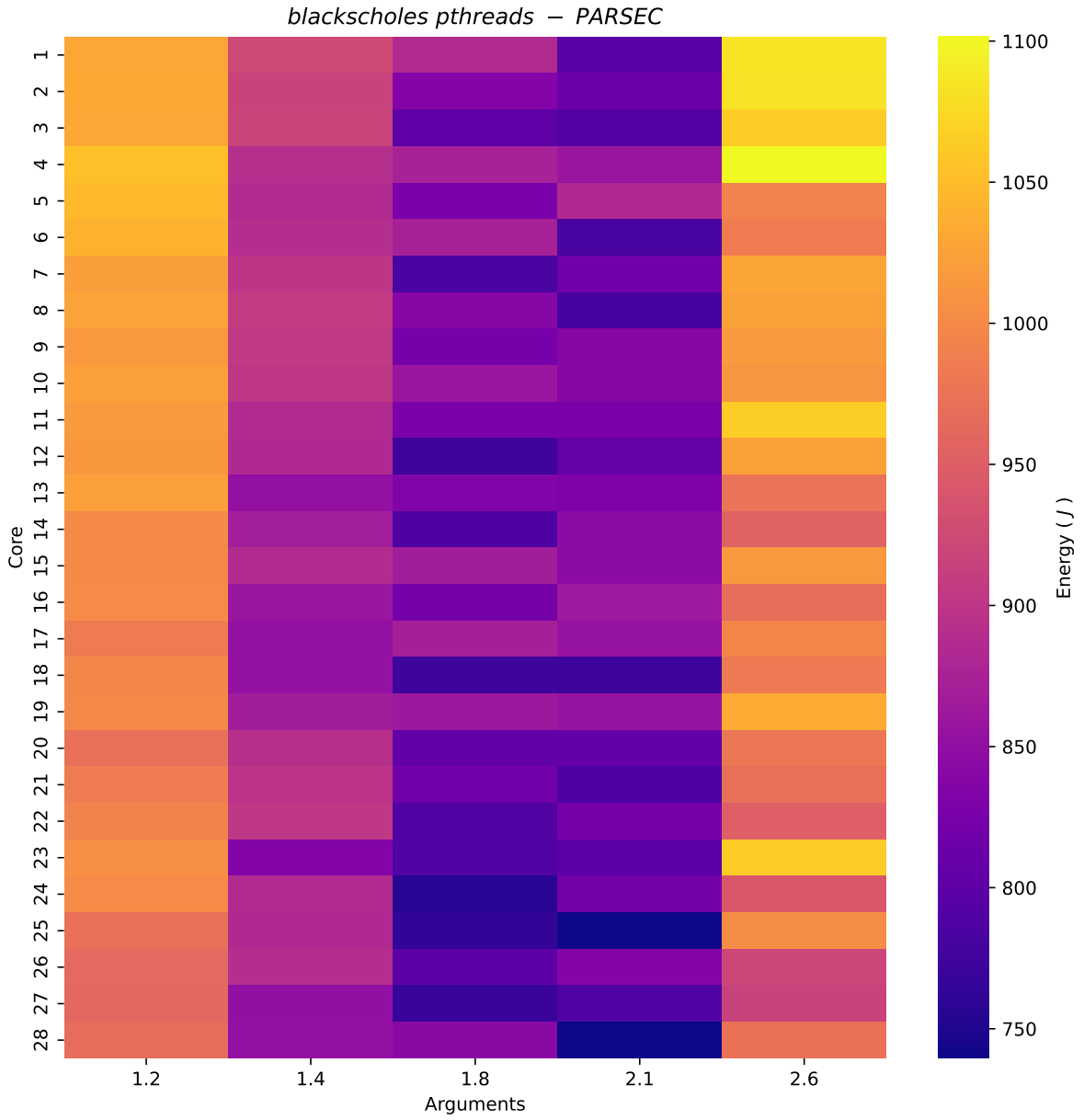

sumeuler − 80000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1500
1400
1300
1200
1100
1000

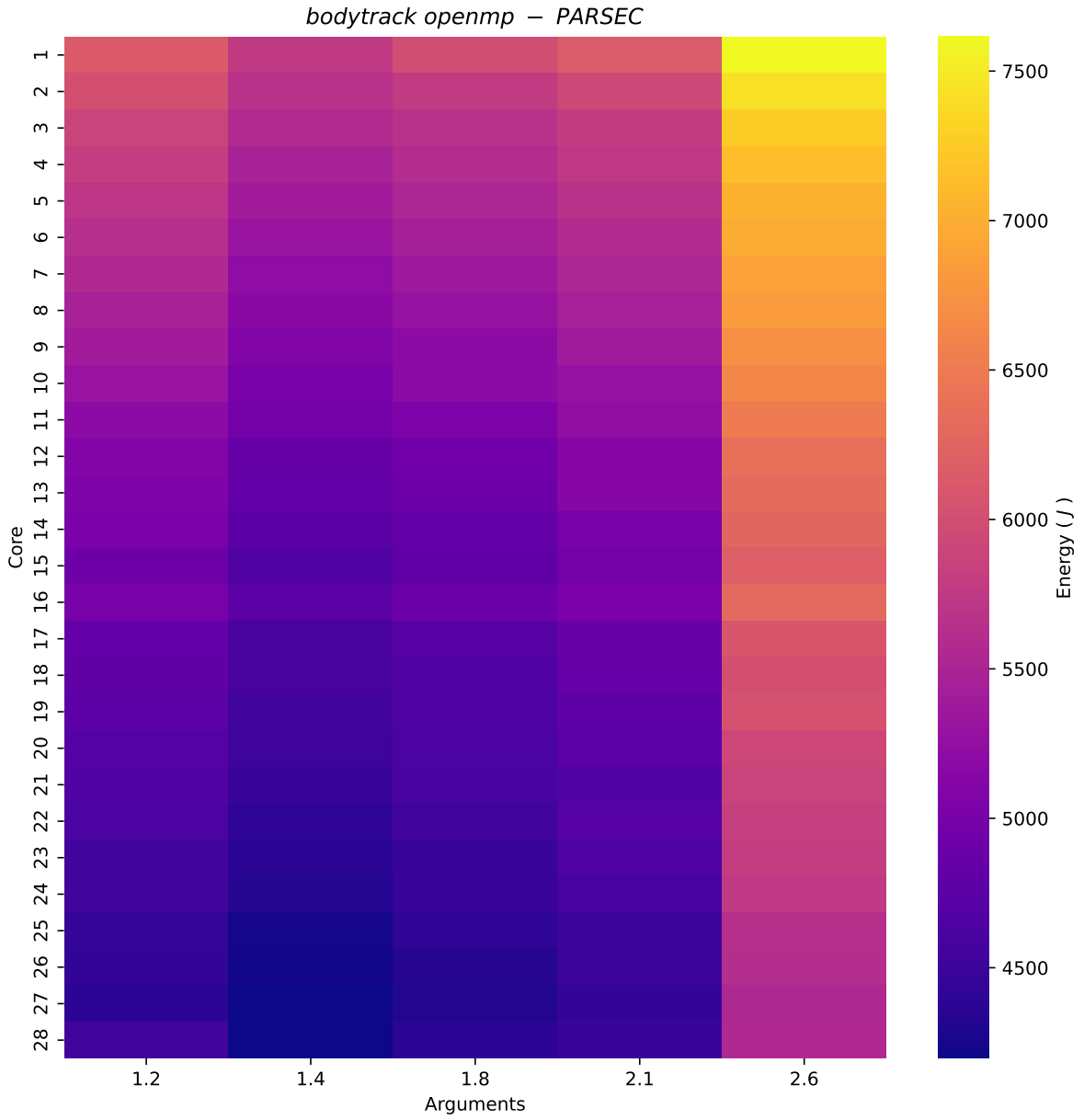

sumeuler − 85000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1800
1700
1600
1500
1400
1300
1200
1100

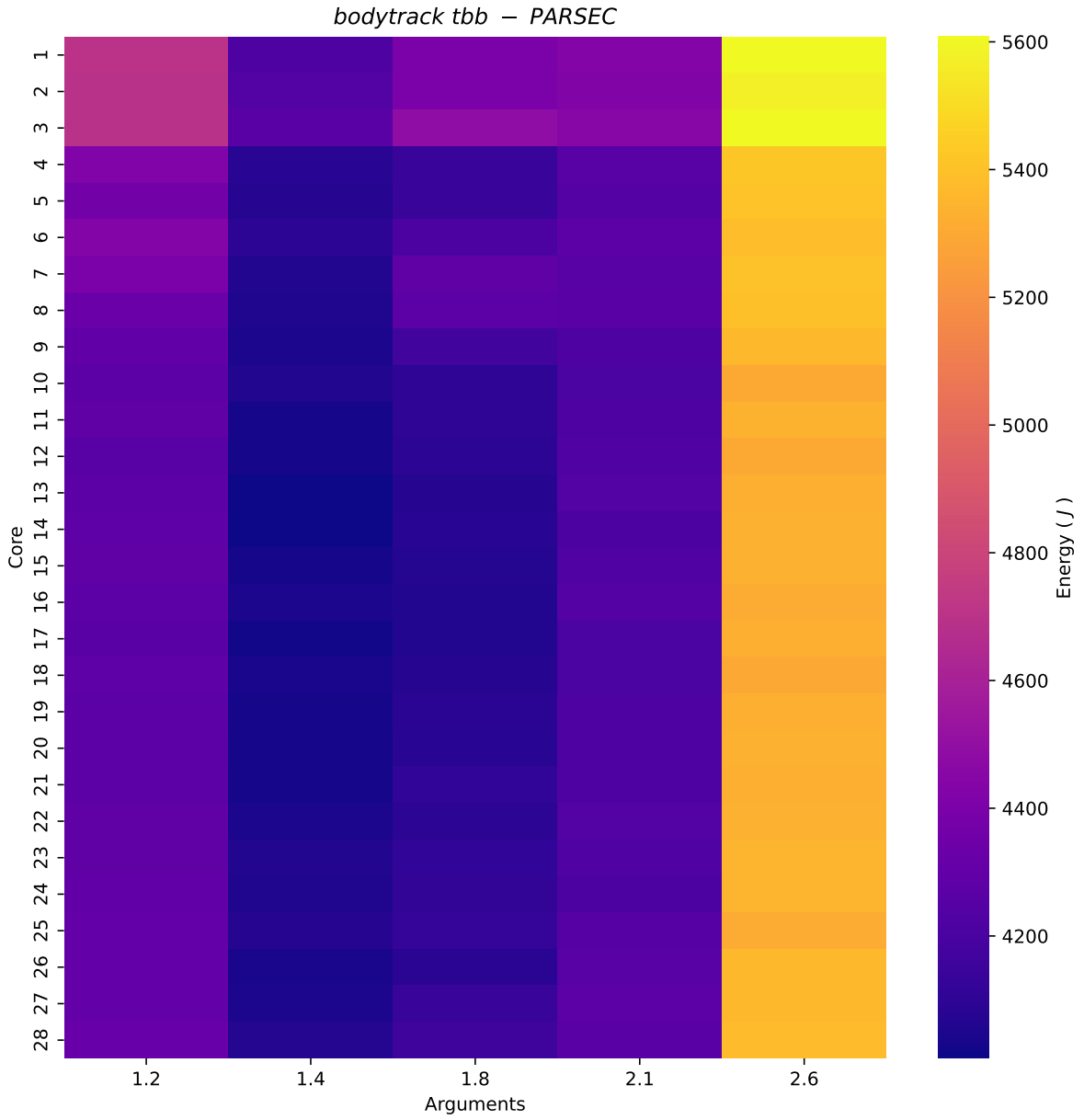

sumeuler − 90000 Haskell
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
2000
1900
1800
1700
1600
1500
1400
1300
1200

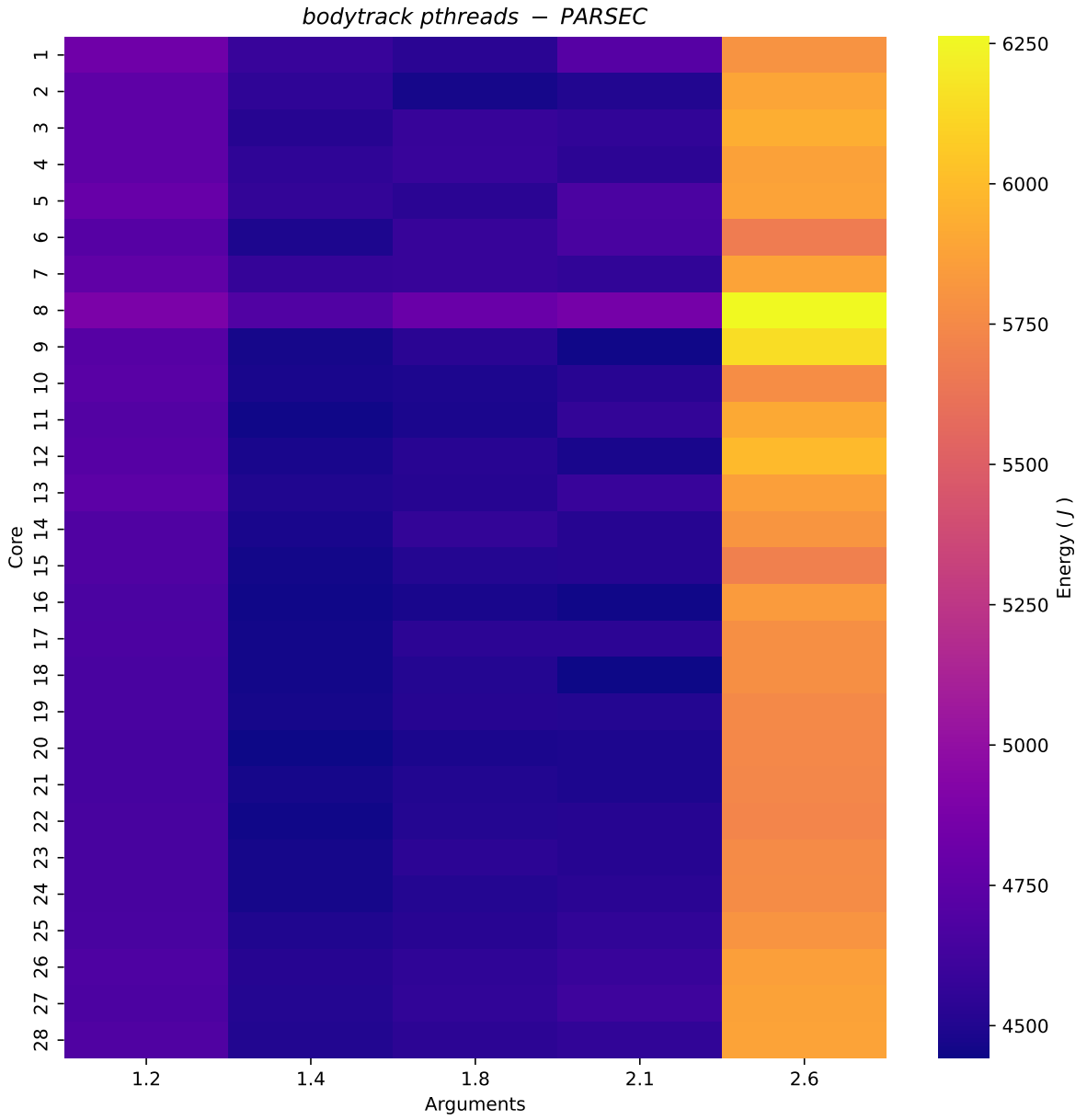

blackscholes openmp — PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
1150
1100
1050
1000
950
900
850
800
750
Energy (J)

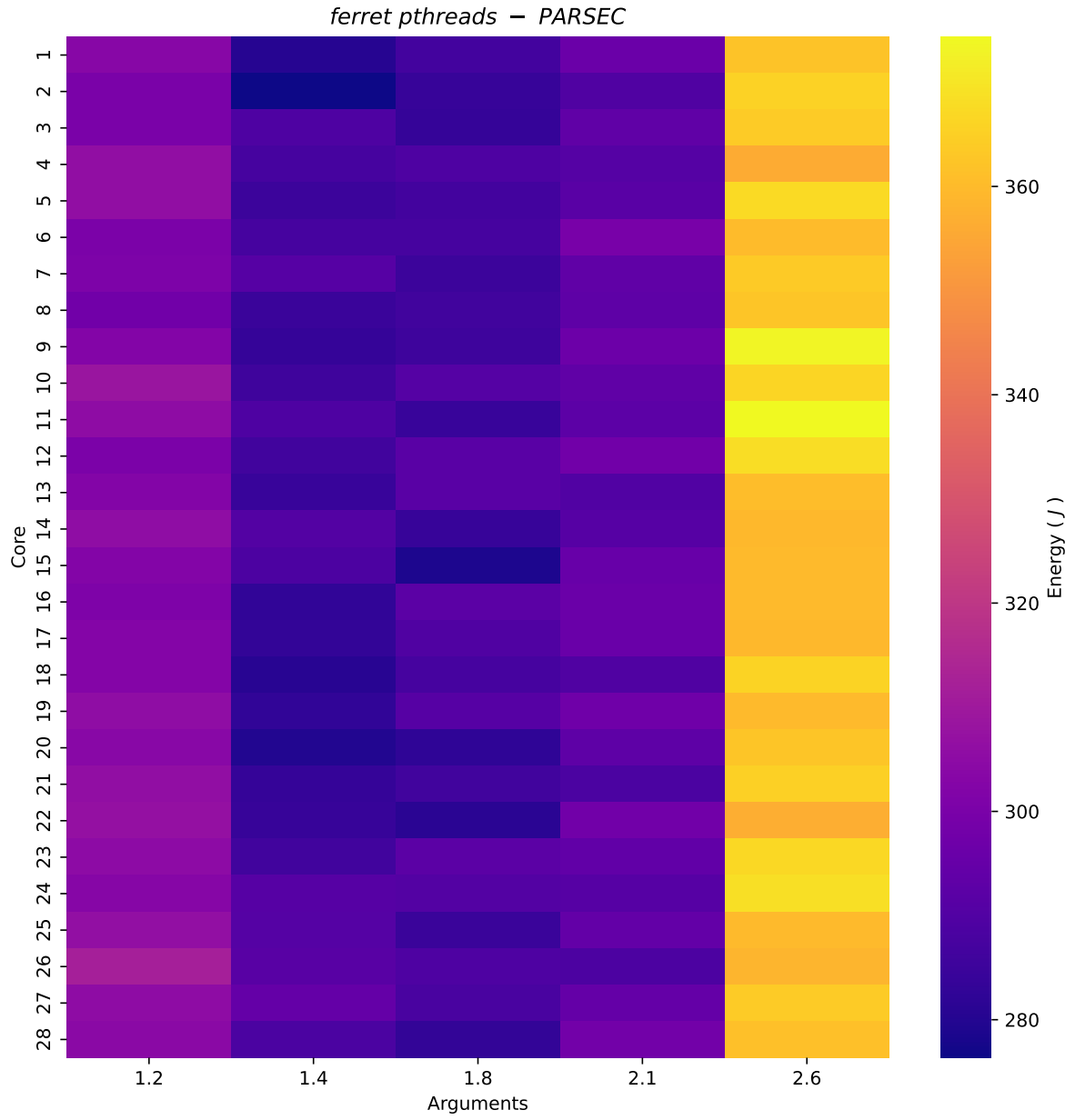

blackscholes tbb — PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1100
1050
1000
950
900
850
800

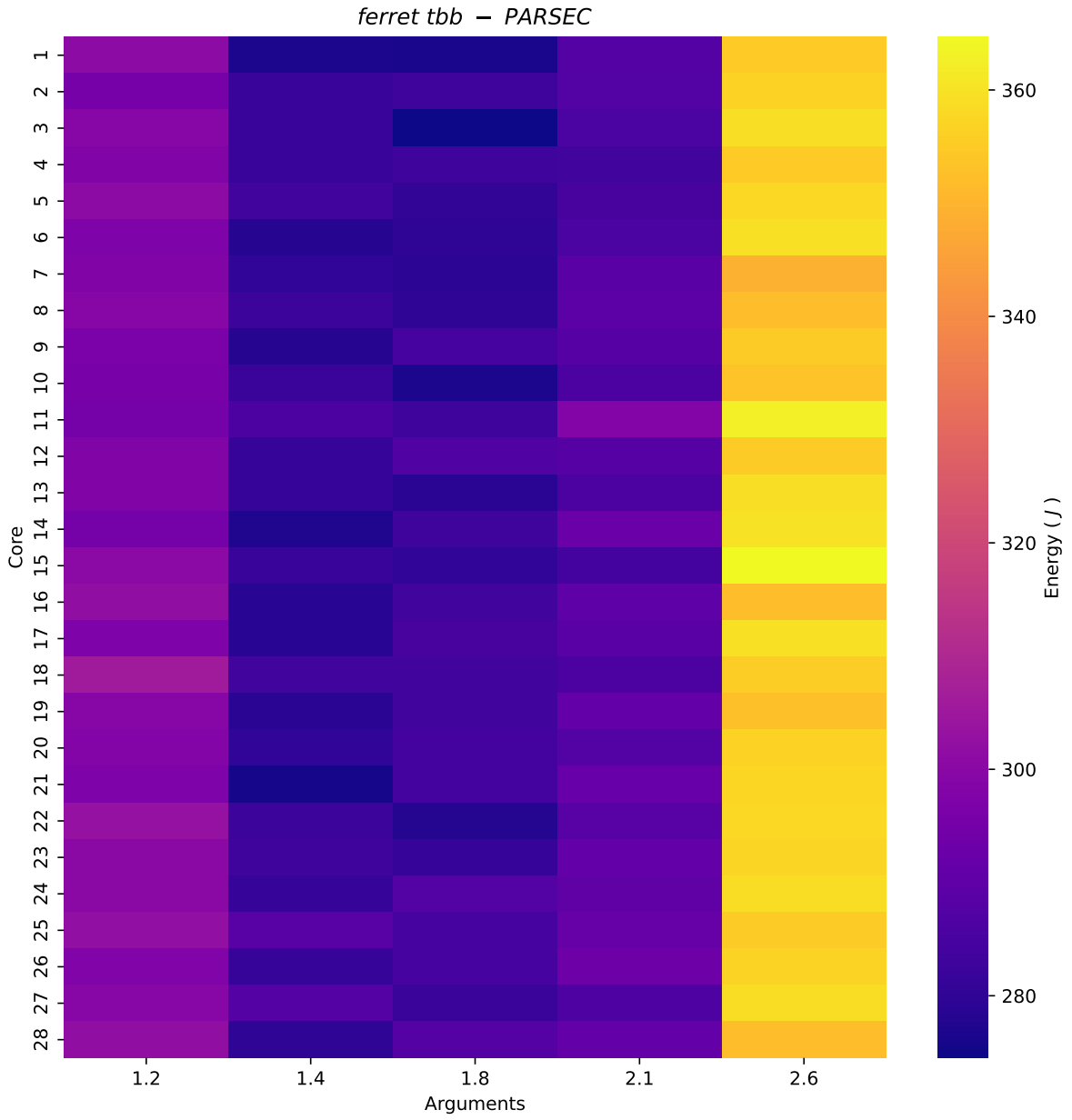

blackscholes pthreads — PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
1100
1050
1000
950
900
850
800
750

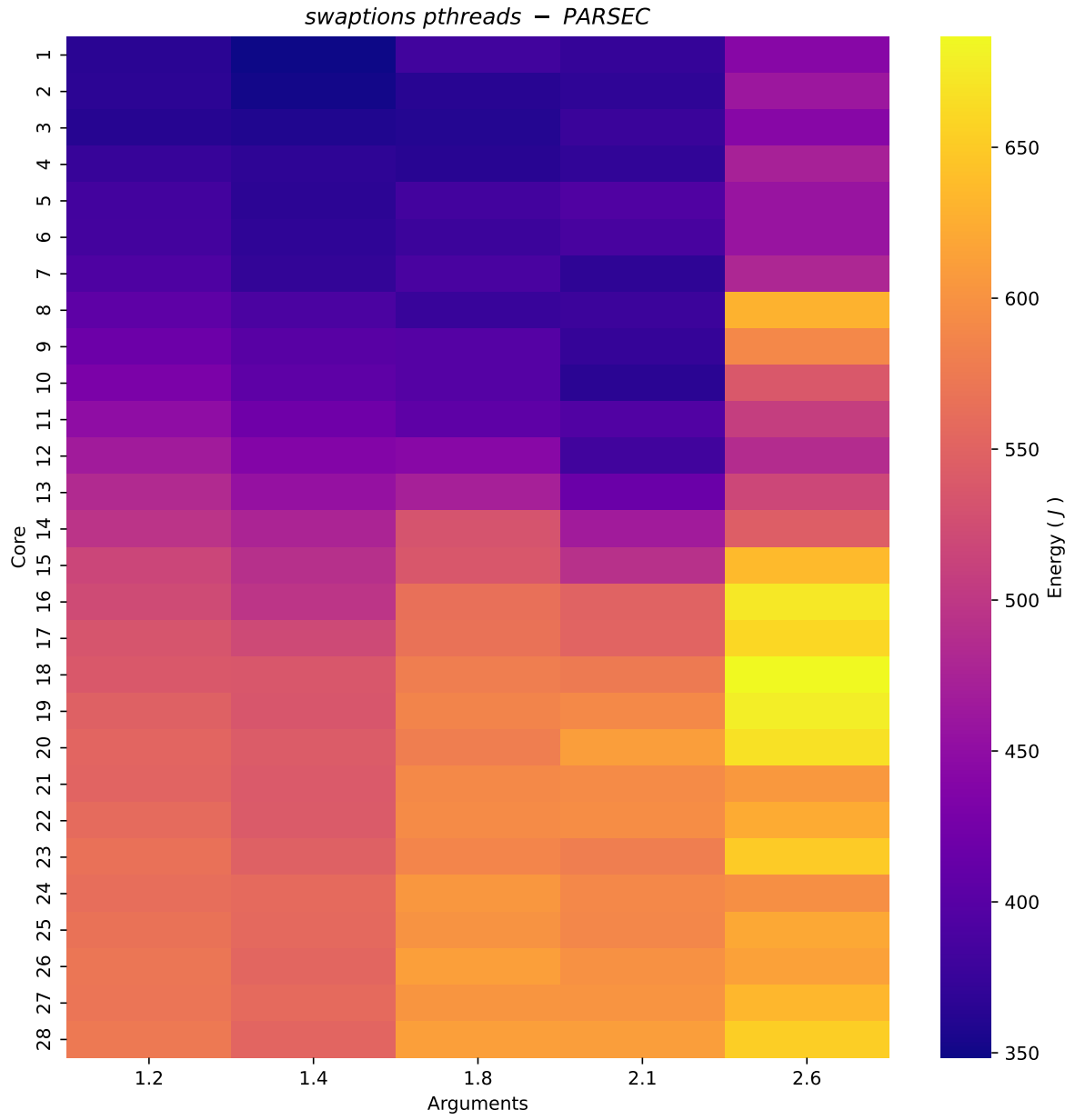

bodytrack openmp – PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
7500
7000
6500
6000
5500
5000
4500
Energy (J)

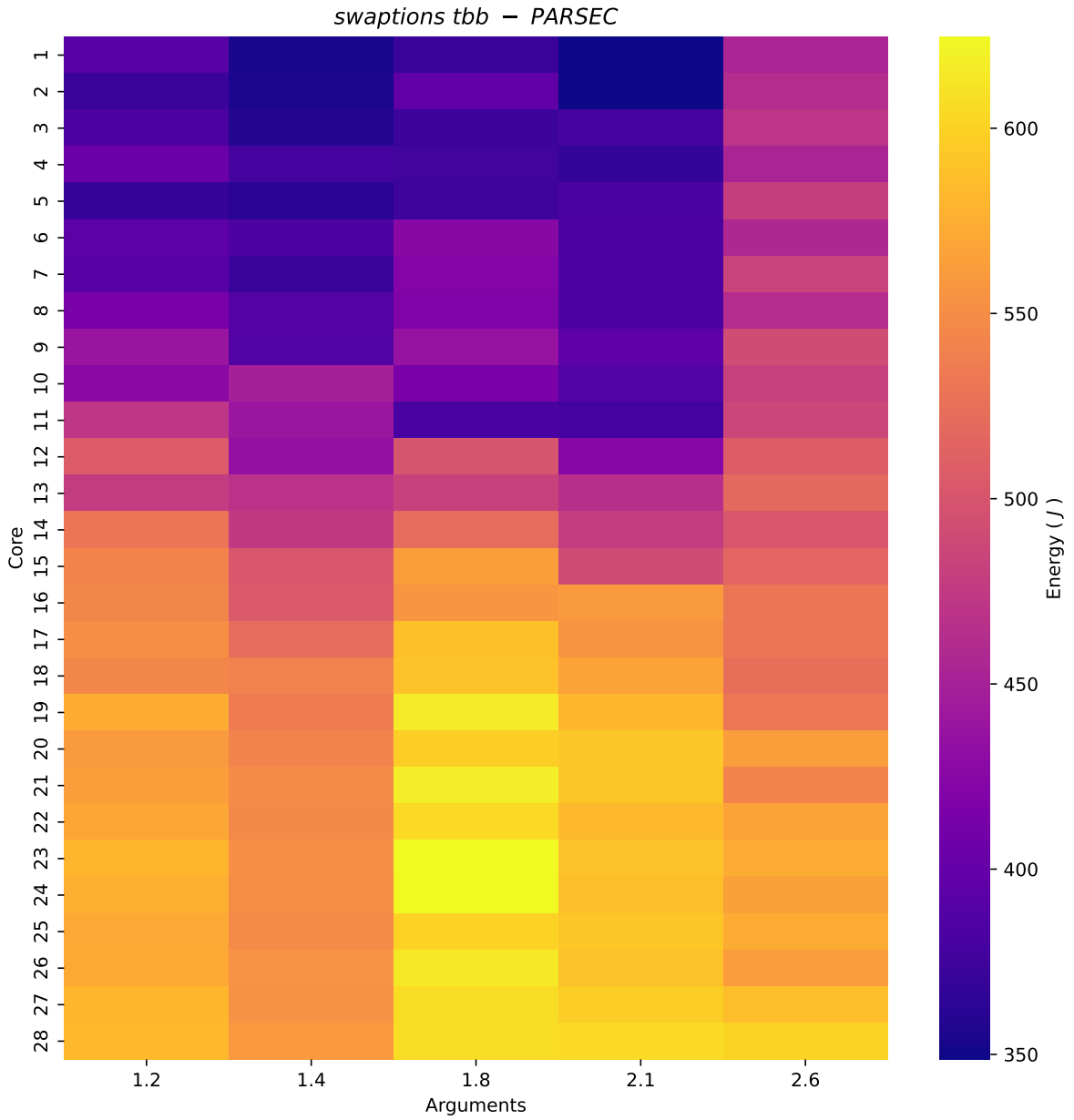

bodytrack tbb – PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
5600
5400
5200
5000
4800
4600
4400
4200
Energy (J)

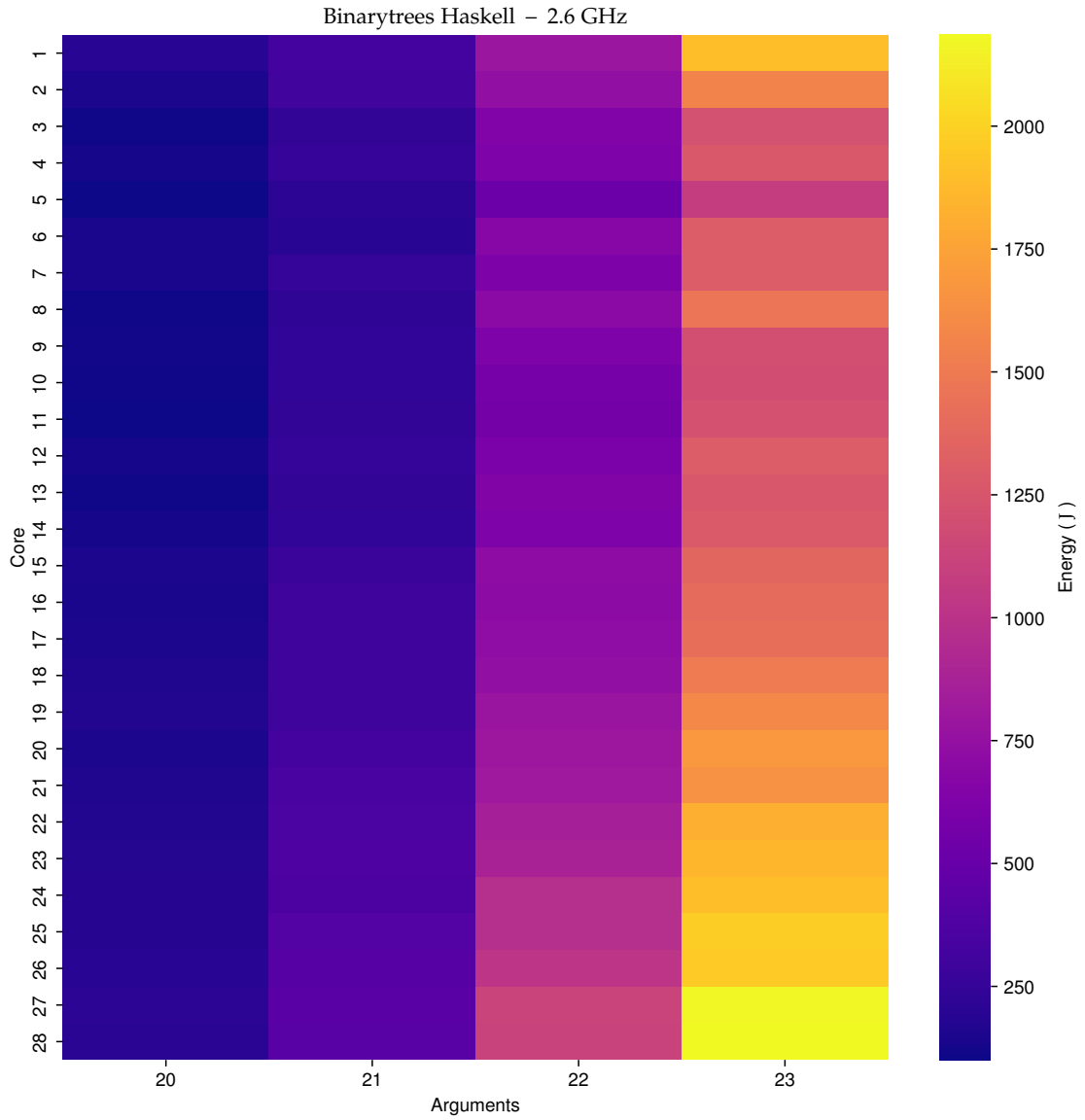

bodytrack pthreads — PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
6250
6000
5750
5500
5250
5000
4750
4500

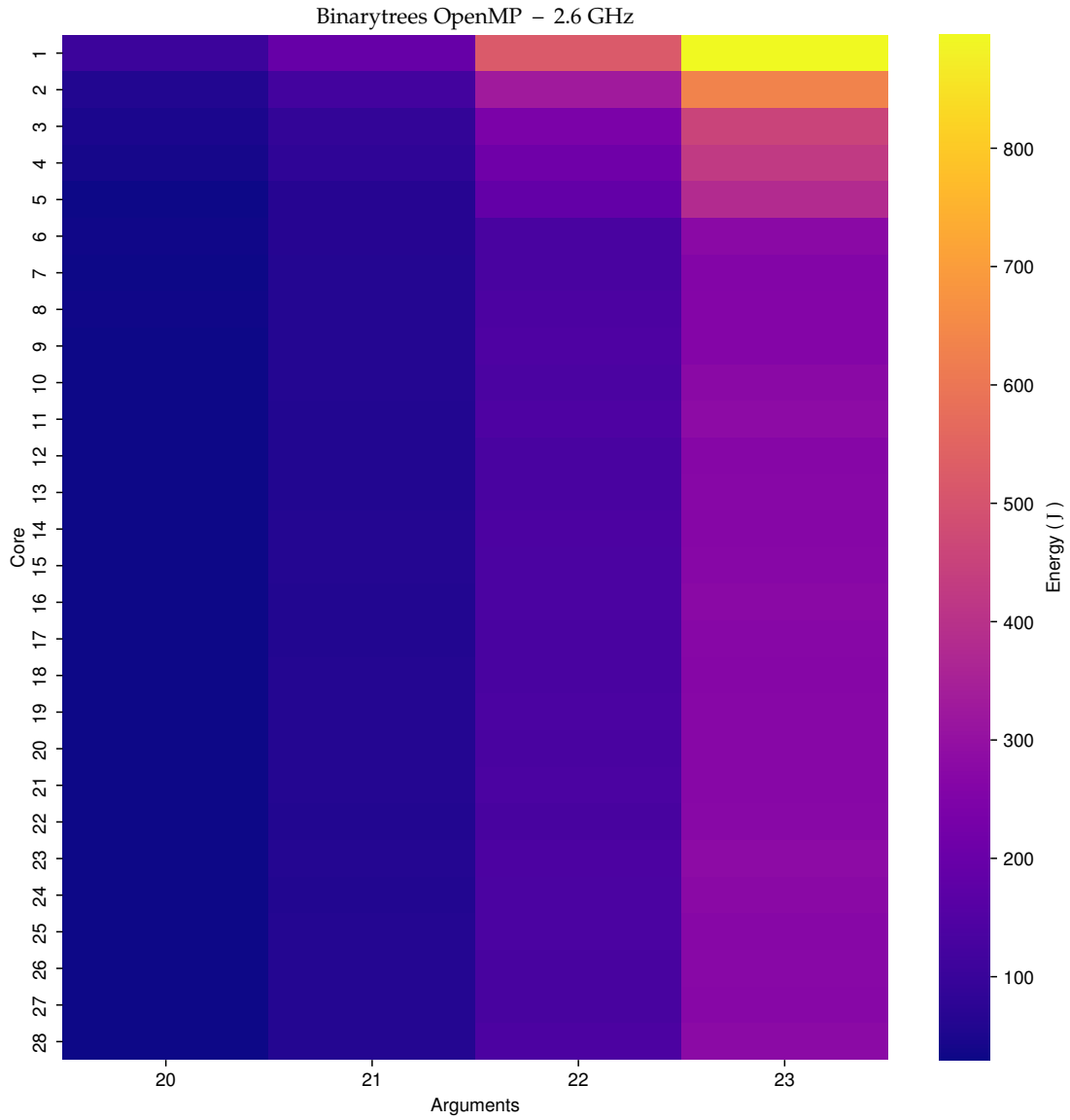

ferret pthreads – PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
360
340
320
300
280

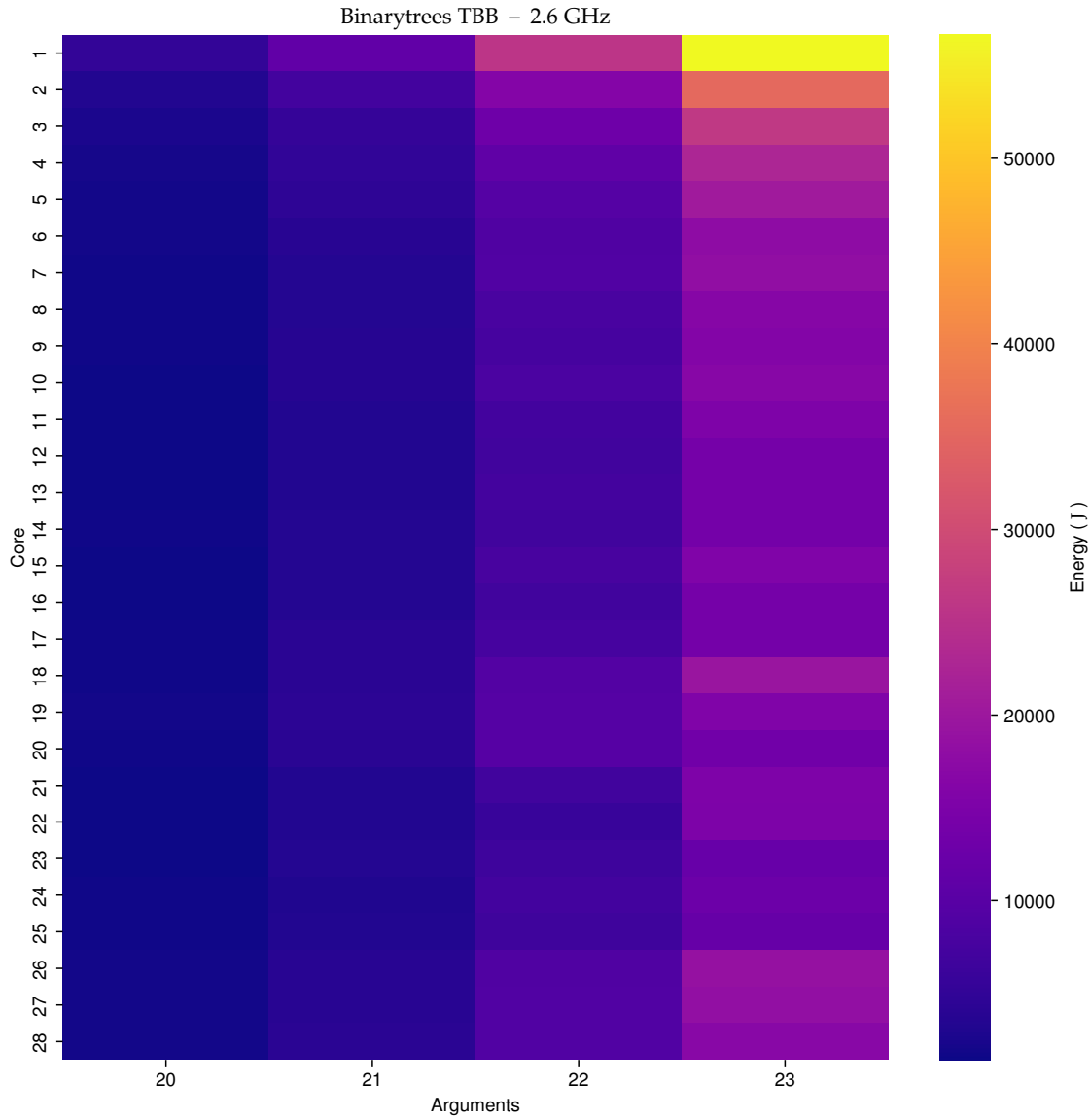

ferret tbb – PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
360
340
320
300
280
Energy (J)

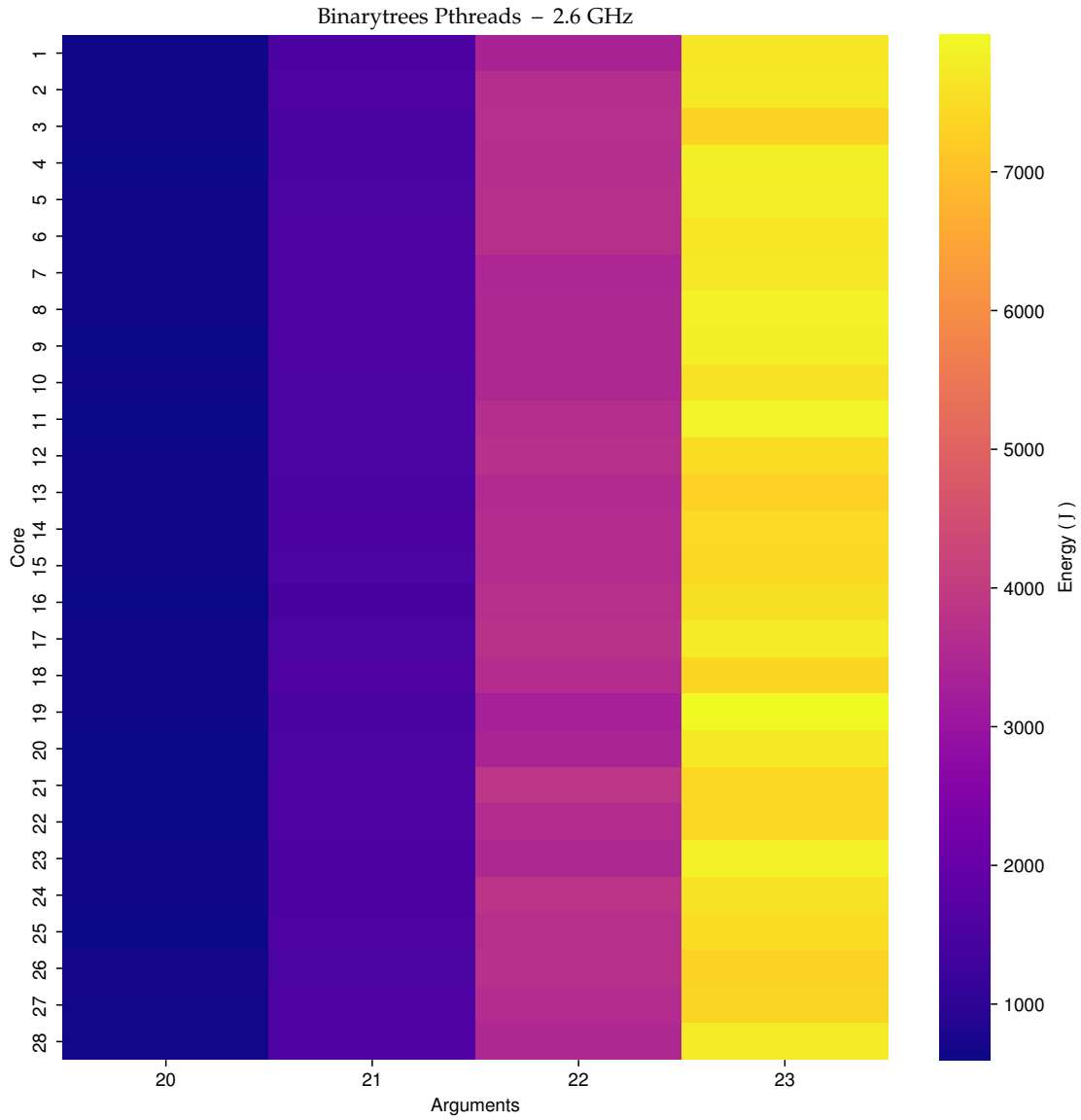

swaptions pthreads – PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
650
600
550
500
450
400
350

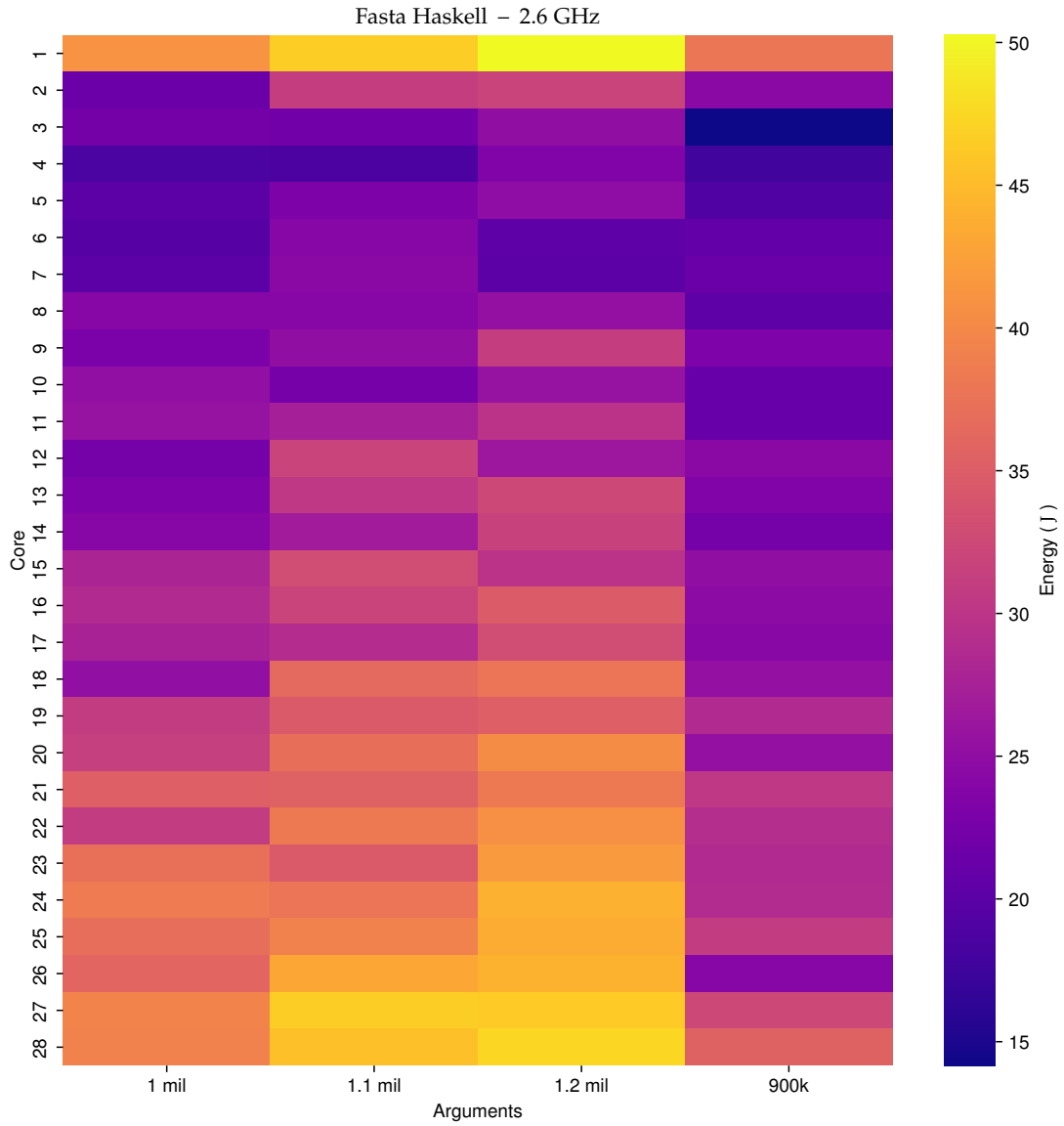

swaptions tbb – PARSEC
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1.2
1.4
1.8
2.1
2.6
Arguments
Energy (J)
600
550
500
450
400
350

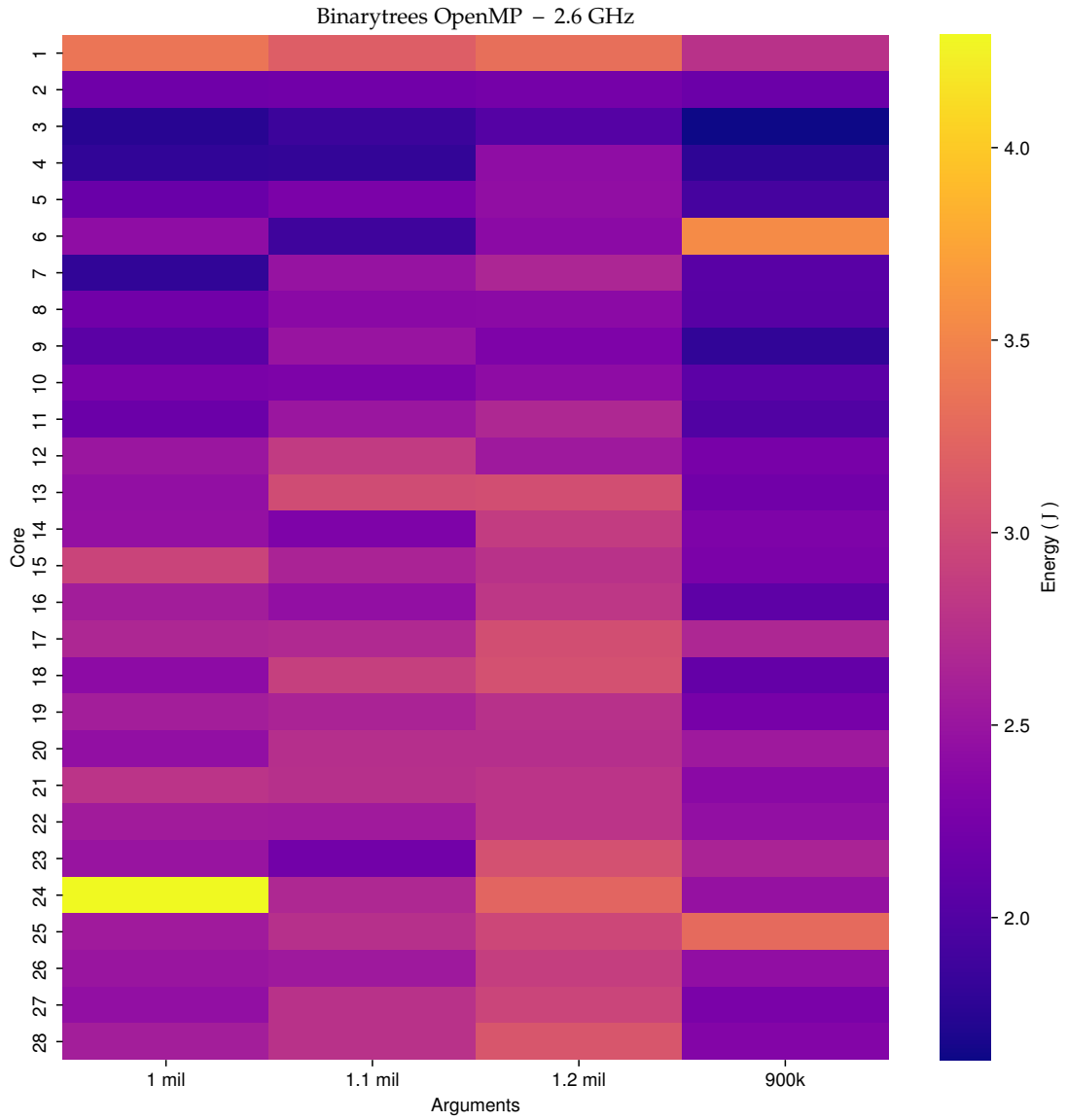

Binarytrees Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
20
21
22
23
Arguments
Energy (J)
2000
1750
1500
1250
1000
750
500
250

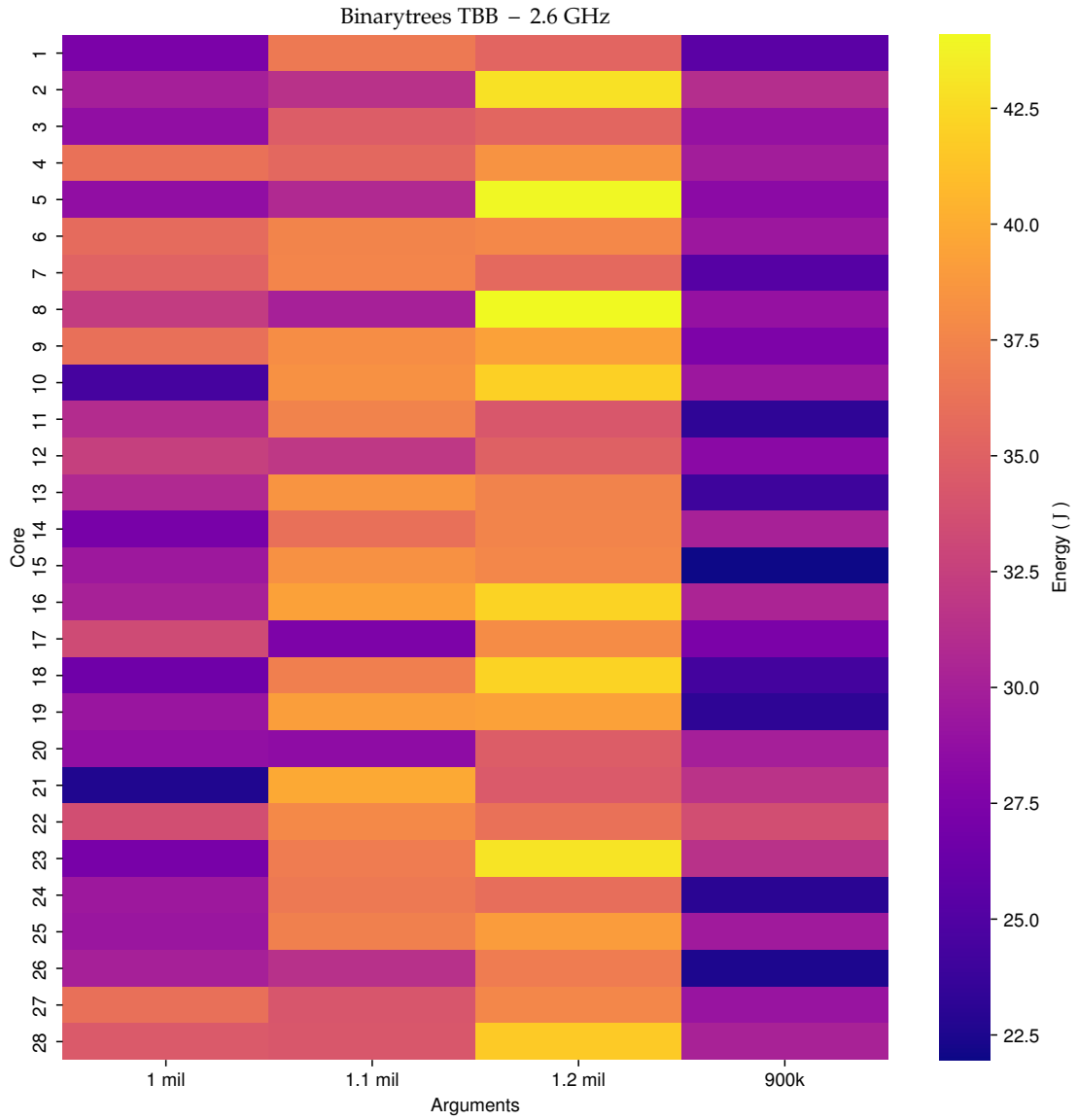

Binarytrees OpenMP – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
20
21
22
23
Arguments
Energy (J)
800
700
600
500
400
300
200
100

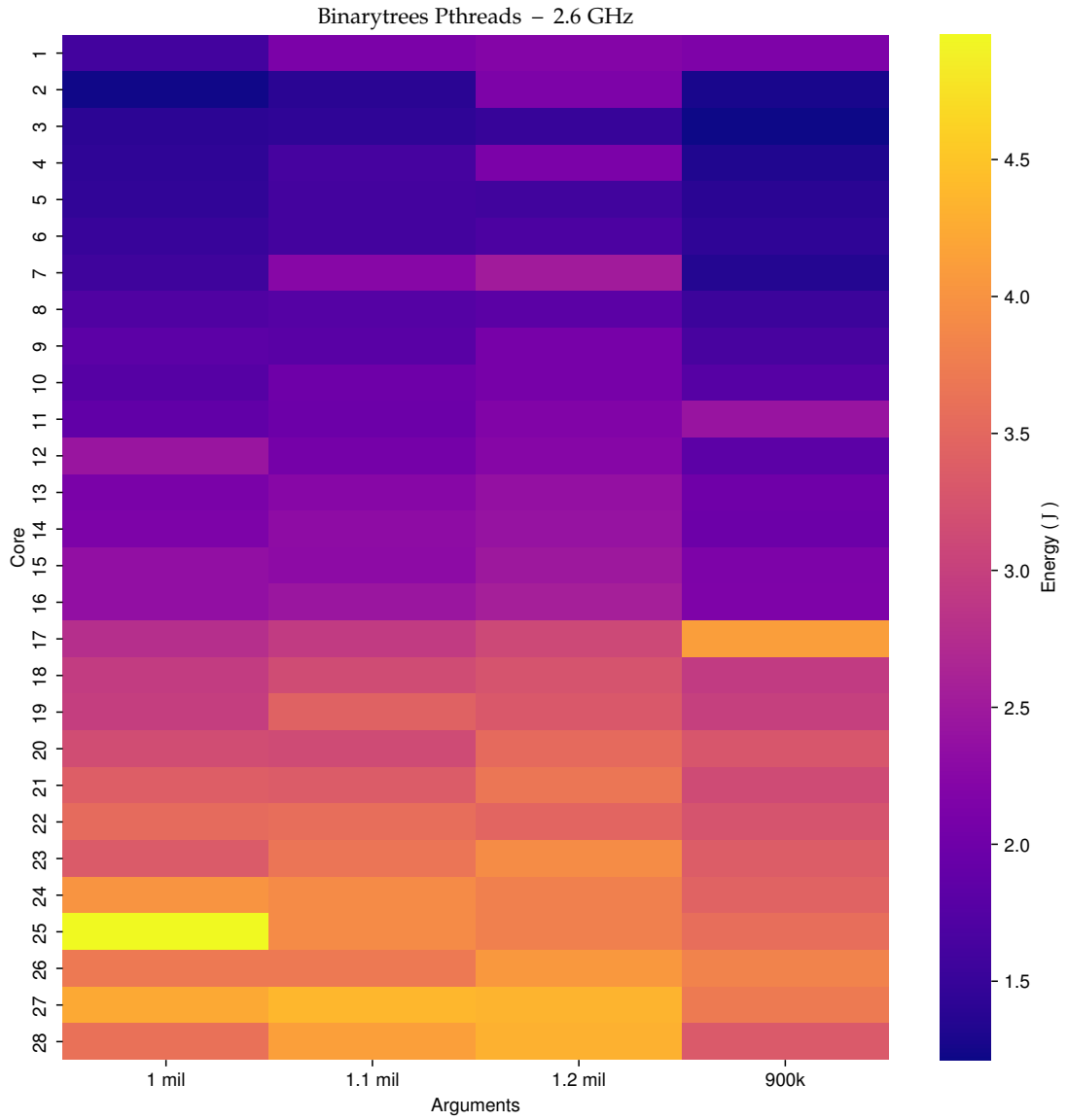

Binarytrees TBB – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
20
21
22
23
Arguments
50000
40000
30000
20000
10000
Energy (J)

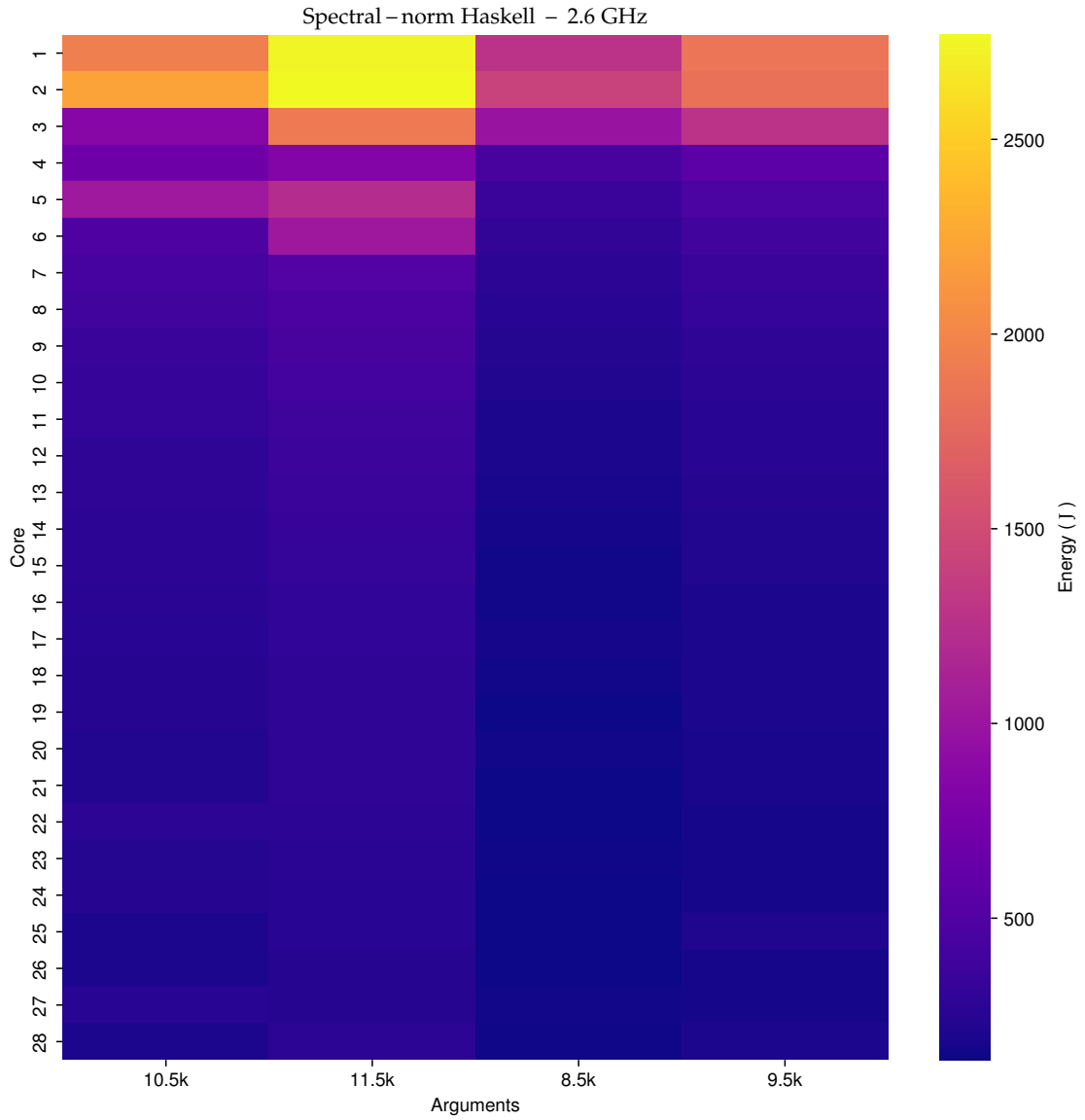

Binarytrees Pthreads – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
20
21
22
23
Arguments
Energy (J)
7000
6000
5000
4000
3000
2000
1000

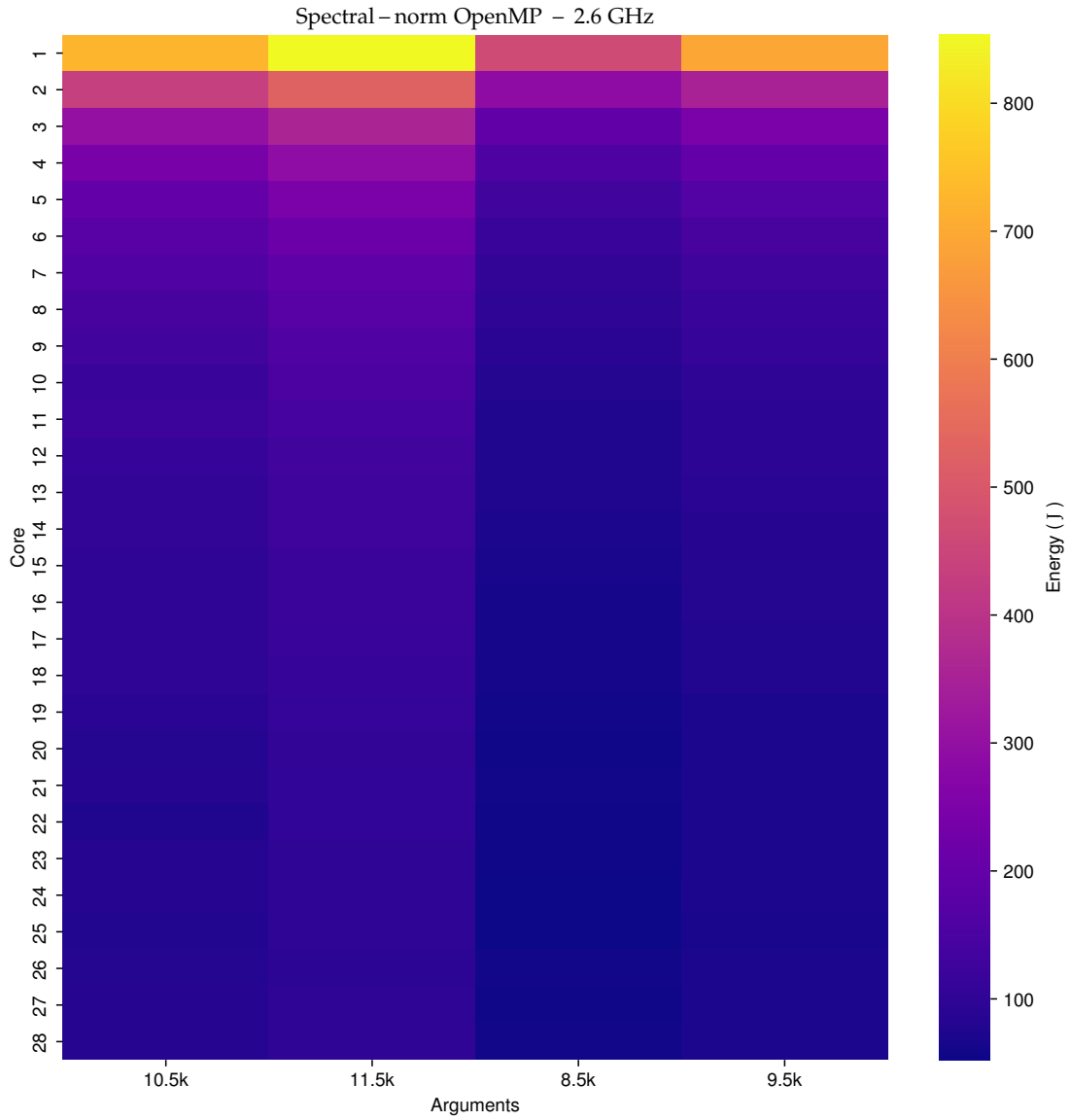

Fasta Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1 mil
1.1 mil
1.2 mil
900k
Arguments
Energy (J)
50
45
40
35
30
25
20
15

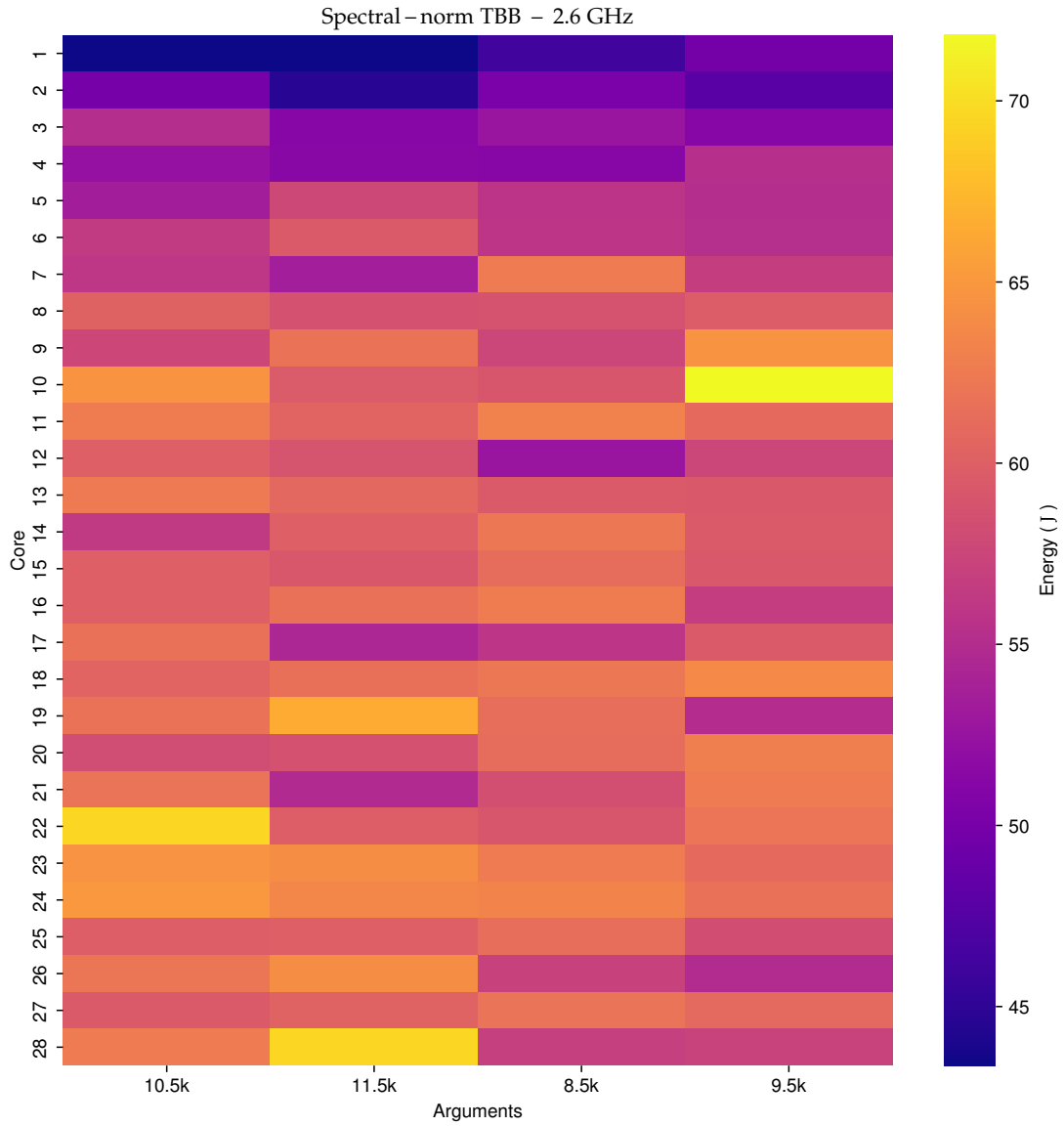

Binarytrees OpenMP – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1 mil
1.1 mil
1.2 mil
900k
Arguments
Energy (J)
4.0
3.5
3.0
2.5
2.0

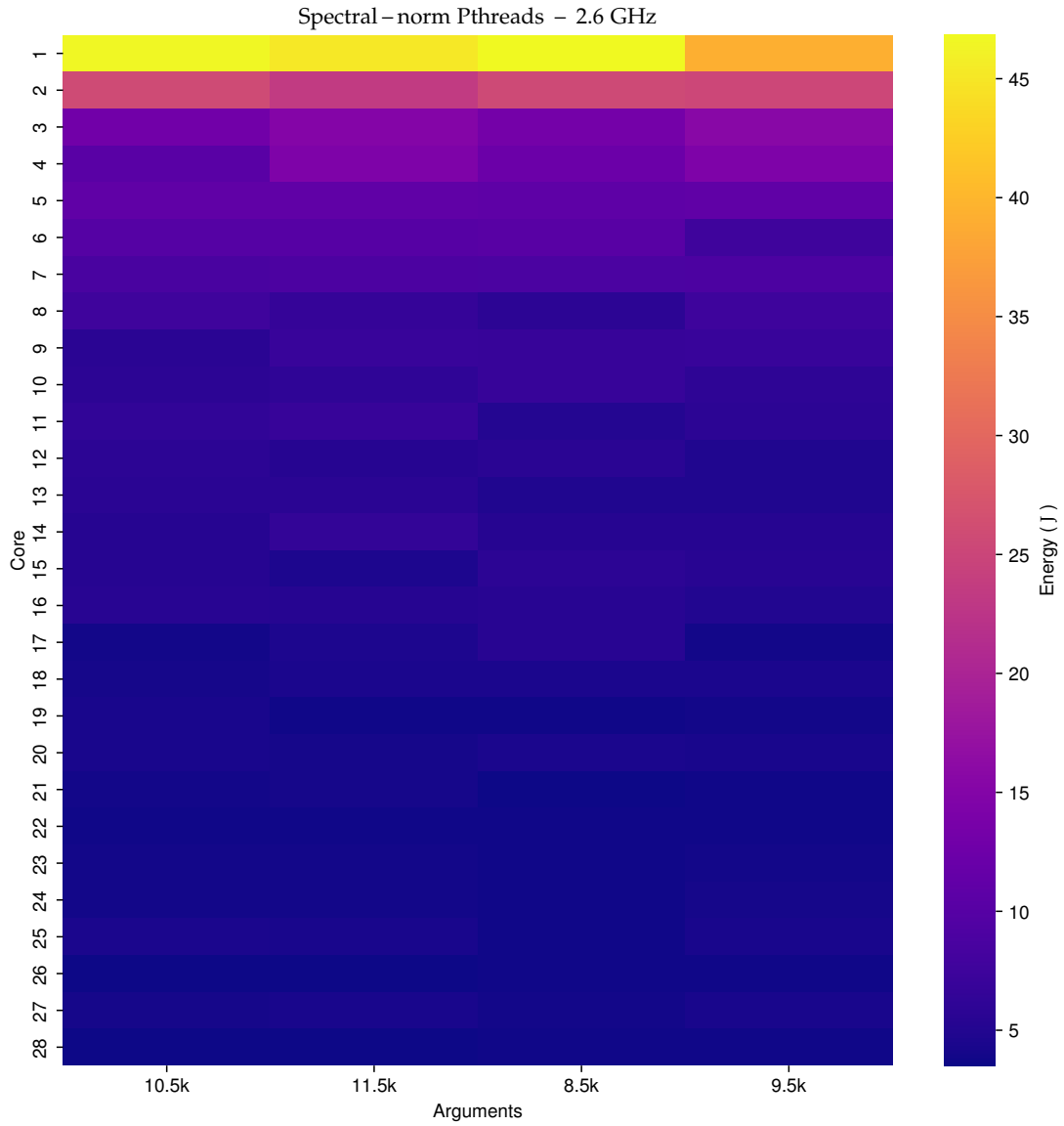

Binarytrees TBB – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1 mil
1.1 mil
1.2 mil
900k
Arguments
Energy (J)
42.5
40.0
37.5
35.0
32.5
30.0
27.5
25.0
22.5


Binarytrees Pthreads – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
1 mil
1.1 mil
1.2 mil
900k
Arguments
Energy (J)
4.5
4.0
3.5
3.0
2.5
2.0
1.5


Spectral – norm Haskell – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
10.5k
11.5k
8.5k
9.5k
Arguments
2500
2000
1500
1000
500
Energy (J)


Spectral – norm OpenMP – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
10.5k
11.5k
8.5k
9.5k
Arguments
Energy (J)
800
700
600
500
400
300
200
100


Spectral – norm TBB – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
10.5k
11.5k
8.5k
9.5k
Arguments
Energy (J)
70
65
60
55
50
45

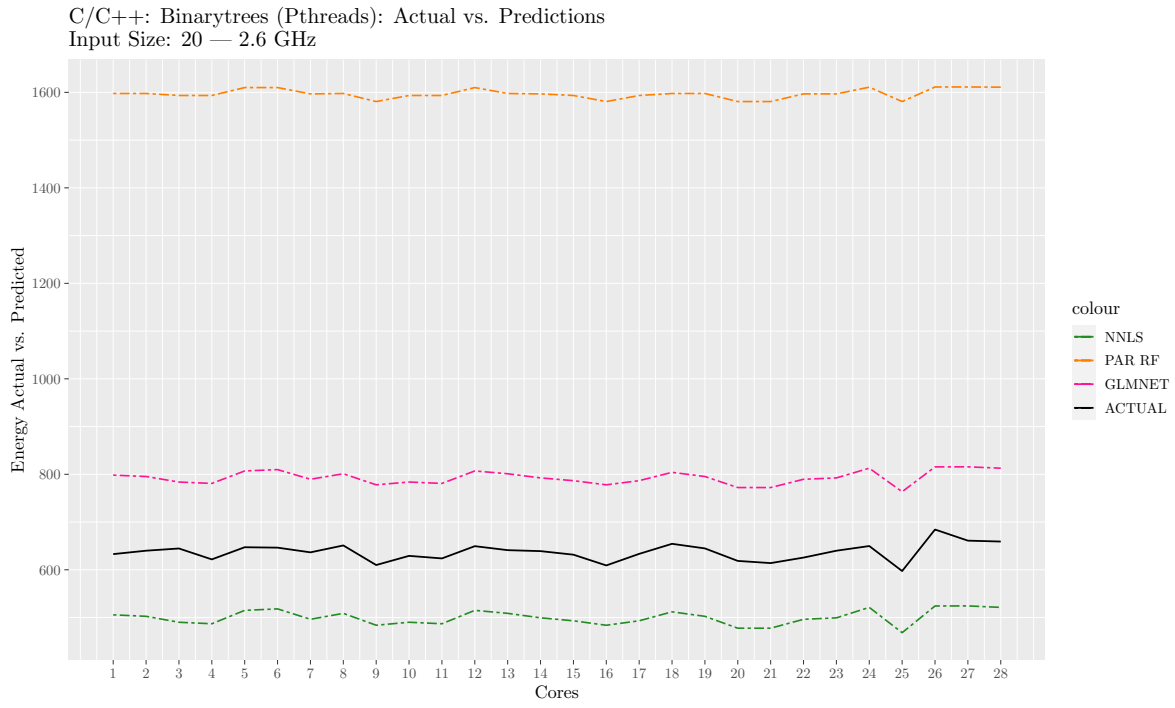

Spectral – norm Pthreads – 2.6 GHz
Core
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
10.5k
11.5k
8.5k
9.5k
Arguments
Energy (J)
45
40
35
30
25
20
15
10
5

Appendix E

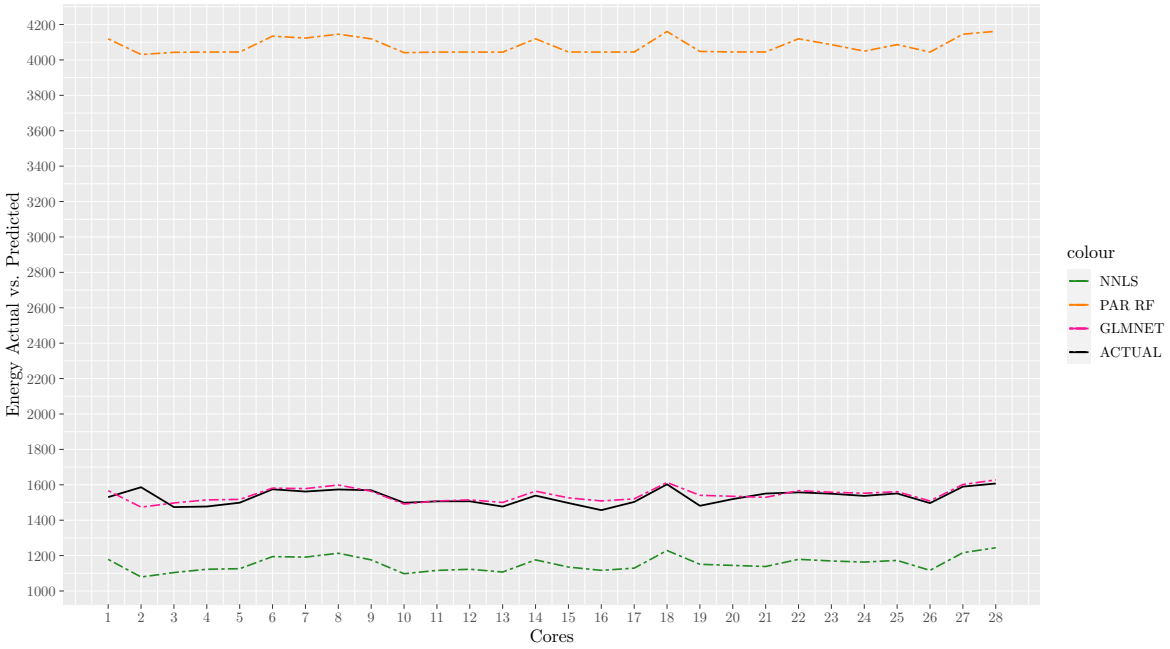
TBB and Pthreads Model Plots

E.1 Pthread Fitted vs. Actual Energy Consumption

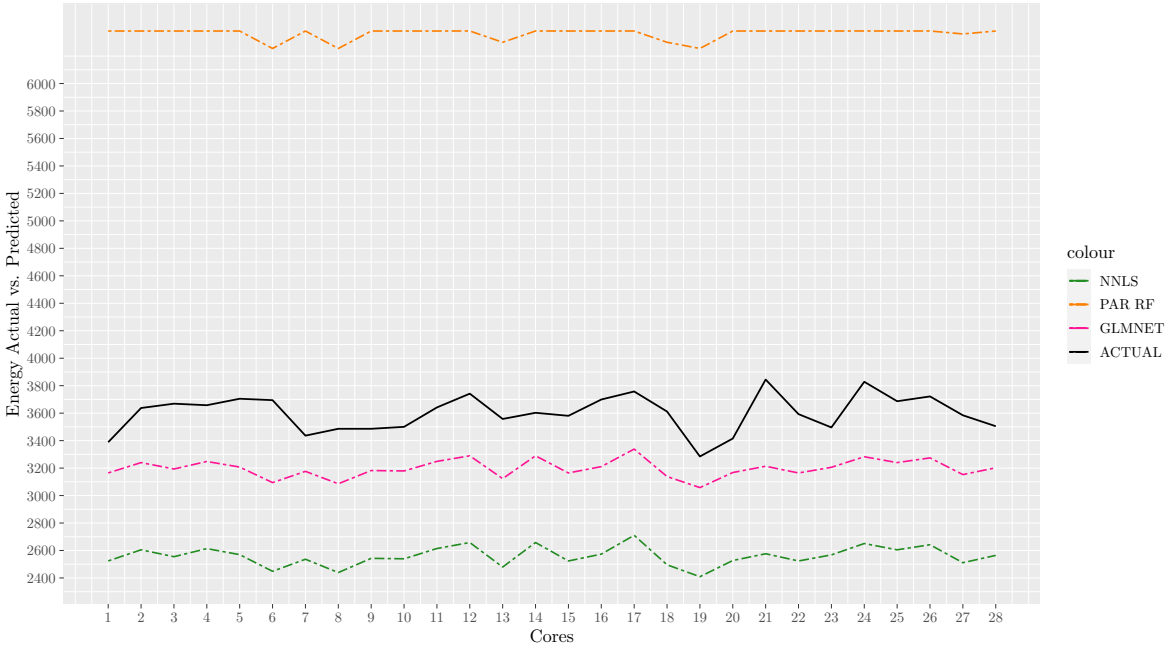
Figures below provide predictions for POSIX threads sample of the prediction dataset. The figures show predictions for **NNLS**, **parRF**, and **GLMNET** against the actual sample energy consumption at highest frequency.



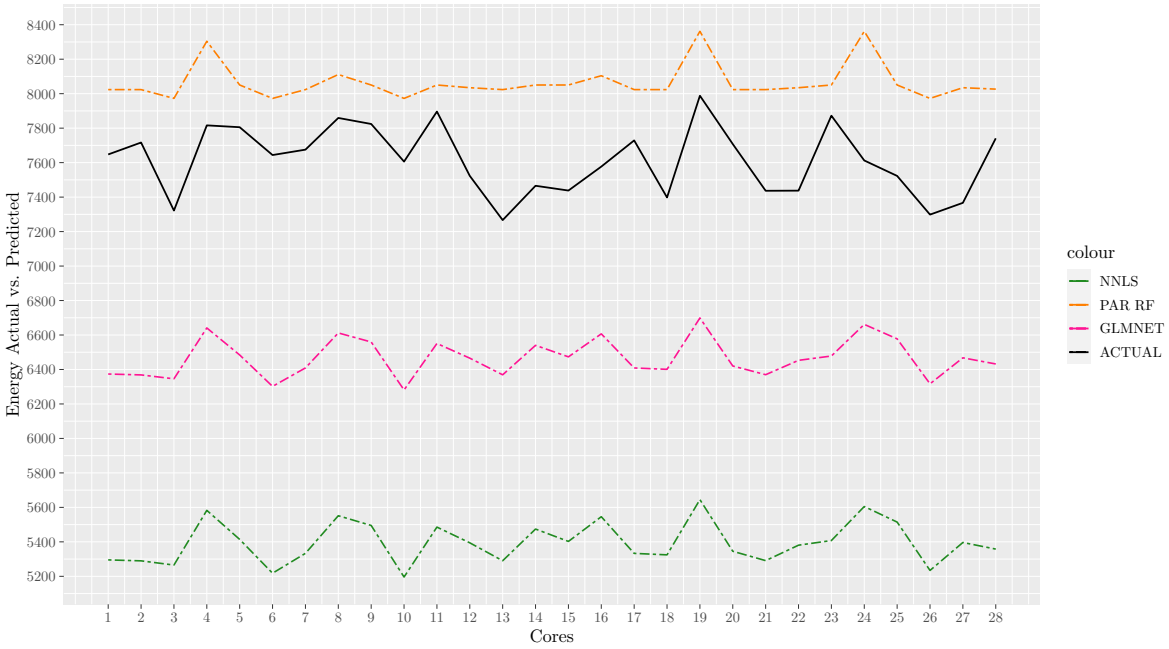
C/C++: Binarytrees (Pthreads): Actual vs. Predictions
Input Size: 21 — 2.6 GHz



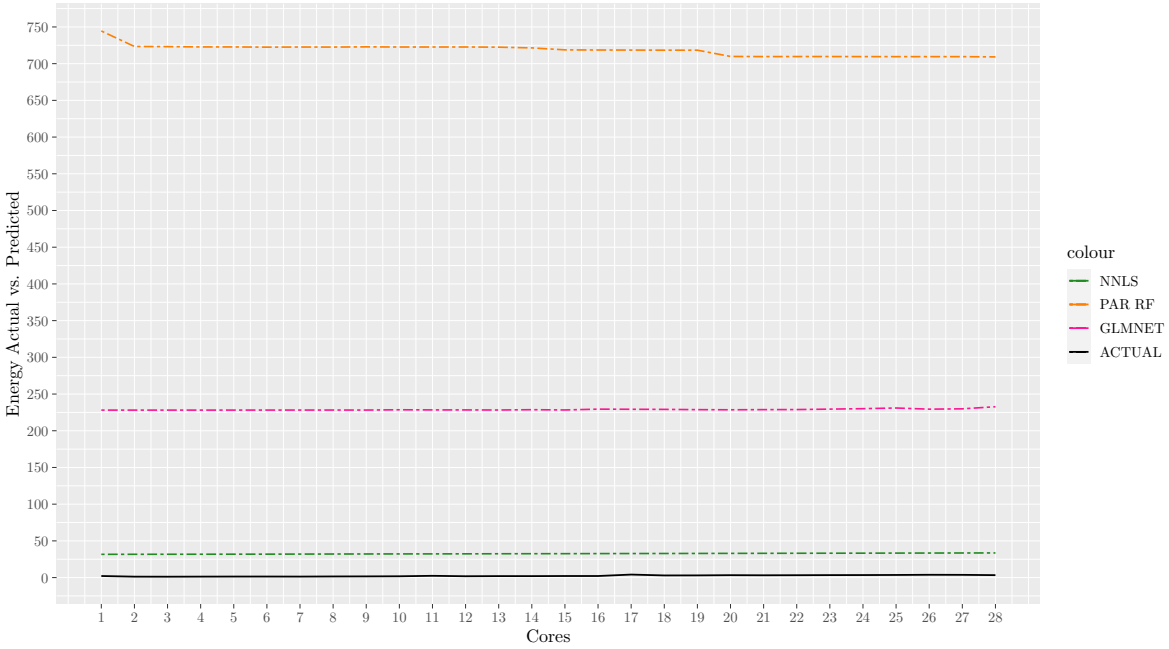
C/C++: Binarytrees (Pthreads): Actual vs. Predictions
Input Size: 22 — 2.6 GHz



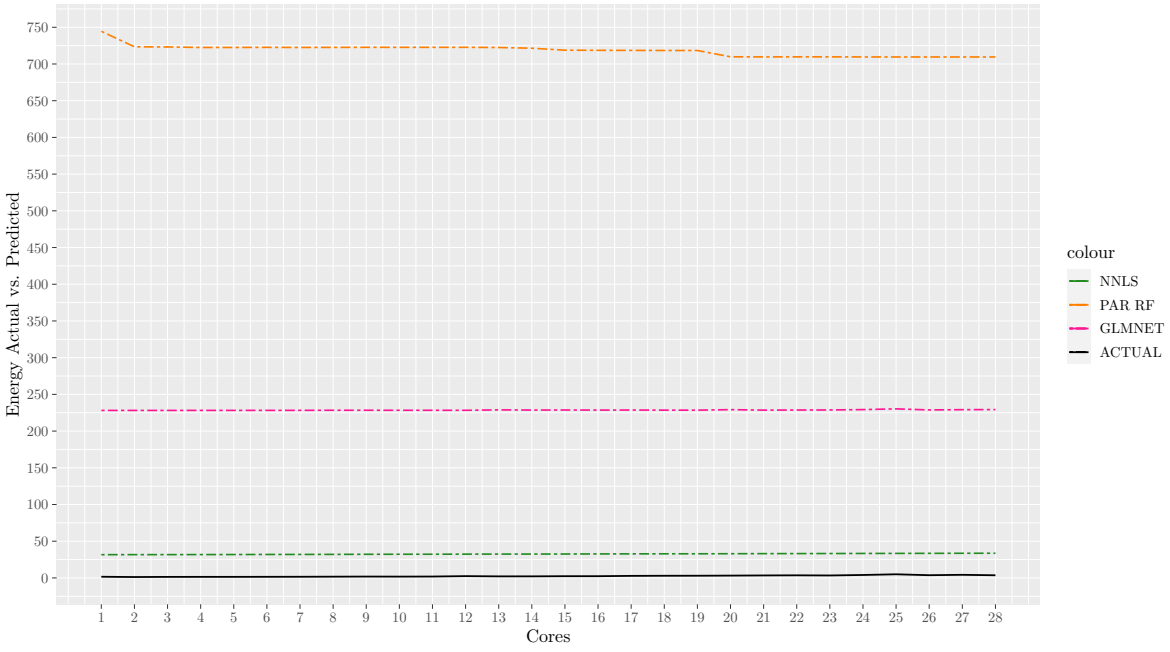
C/C++: Binarytrees (Pthreads): Actual vs. Predictions
Input Size: 23 — 2.6 GHz



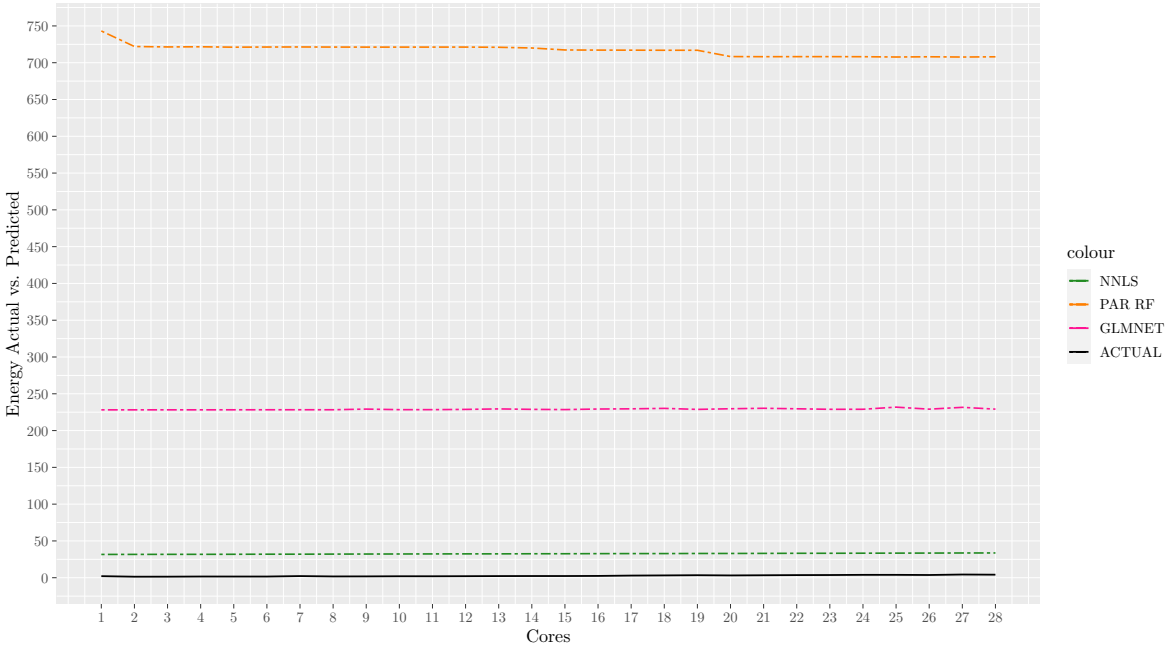
C/C++: Fasta (Pthreads): Actual vs. Predictions
Input Size: 900k — 2.6 GHz



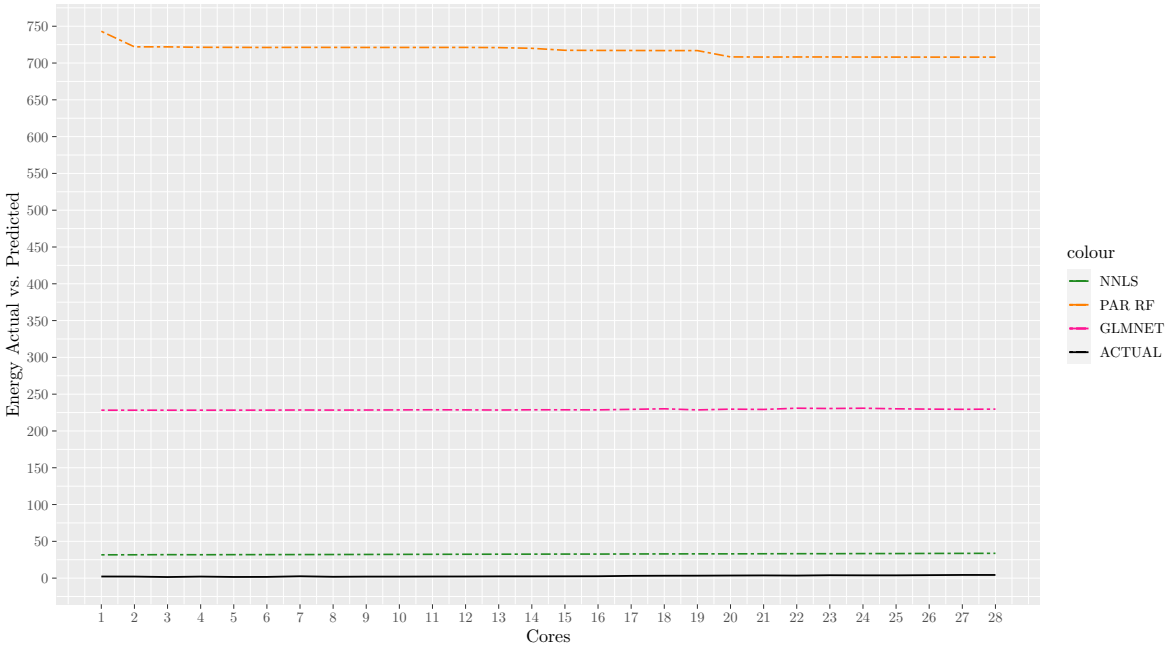
C/C++: Fasta (Pthreads): Actual vs. Predictions
Input Size: 1 mil — 2.6 GHz



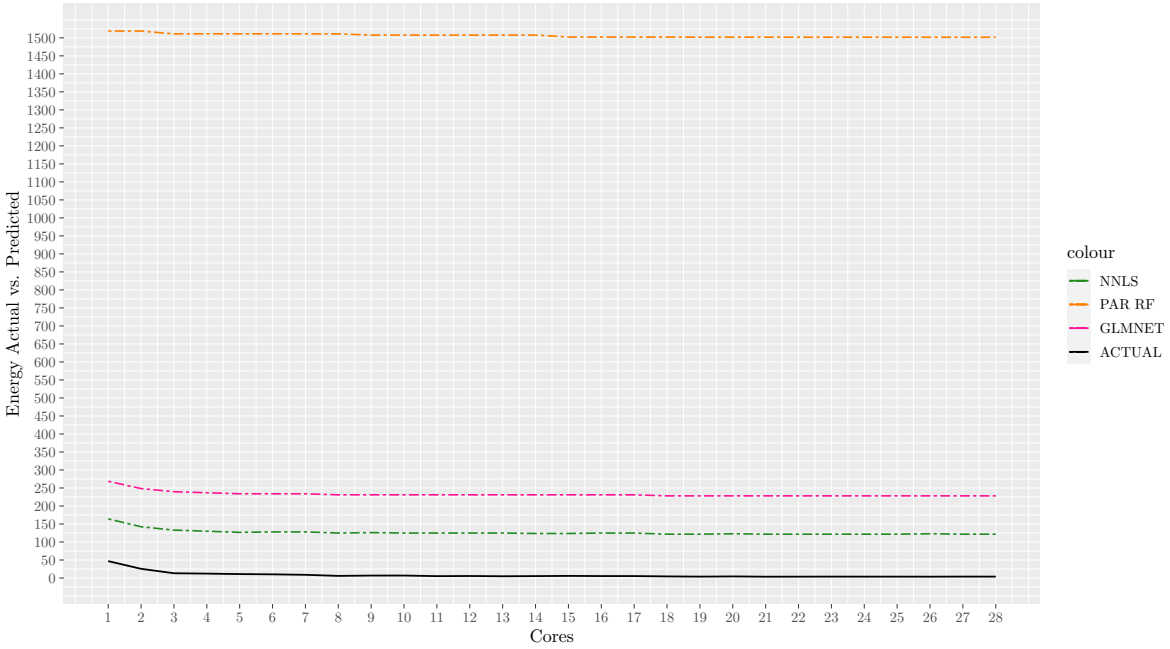
C/C++: Fasta (Pthreads): Actual vs. Predictions
Input Size: 1.1 mil — 2.6 GHz



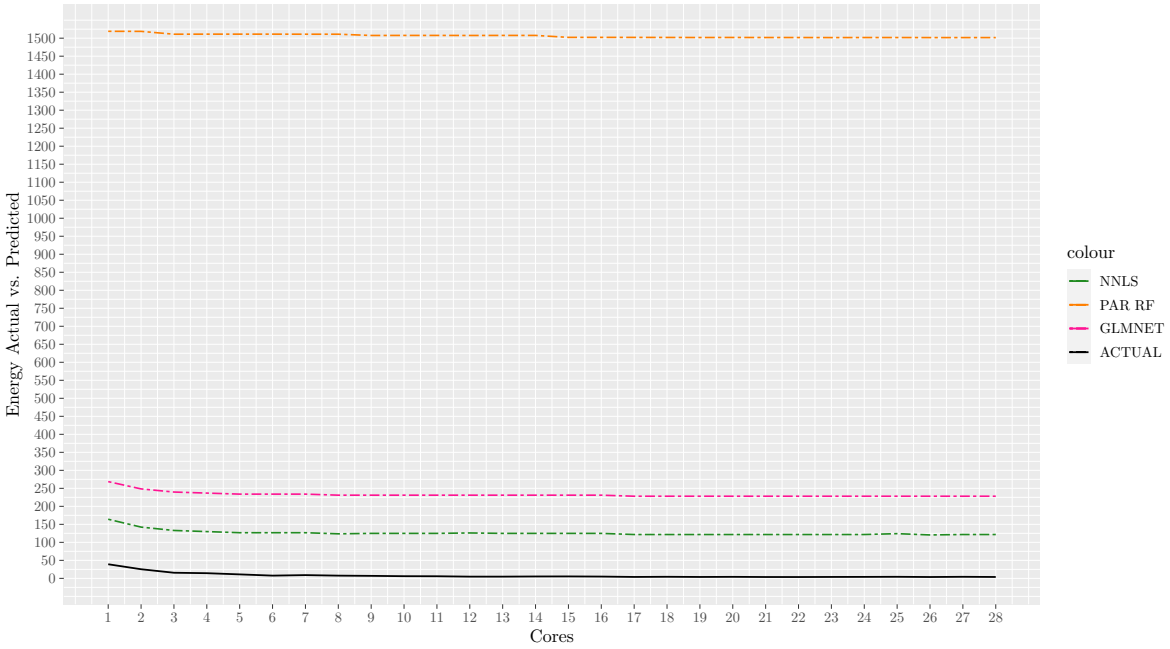
C/C++: Fasta (Pthreads): Actual vs. Predictions
Input Size: 1.2 mil — 2.6 GHz



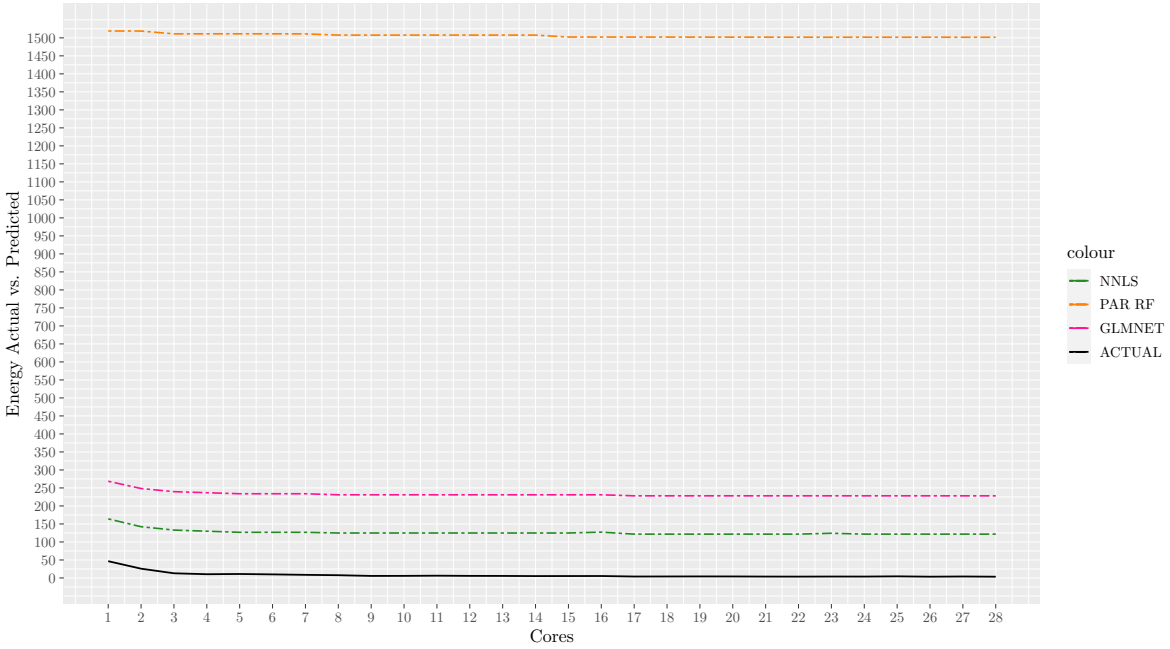
C/C++: Spectral-norm (Pthreads): Actual vs. Predictions
Input Size: 8.5k — 2.6 GHz



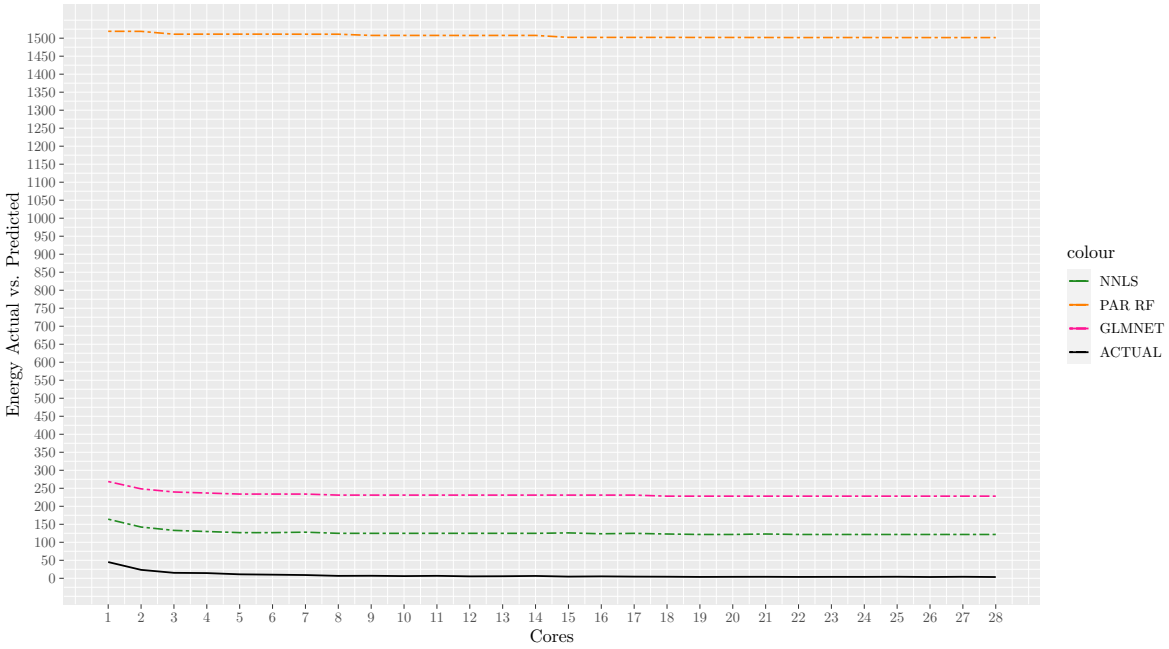
C/C++: Spectral-norm (Pthreads): Actual vs. Predictions
Input Size: 9.5k — 2.6 GHz



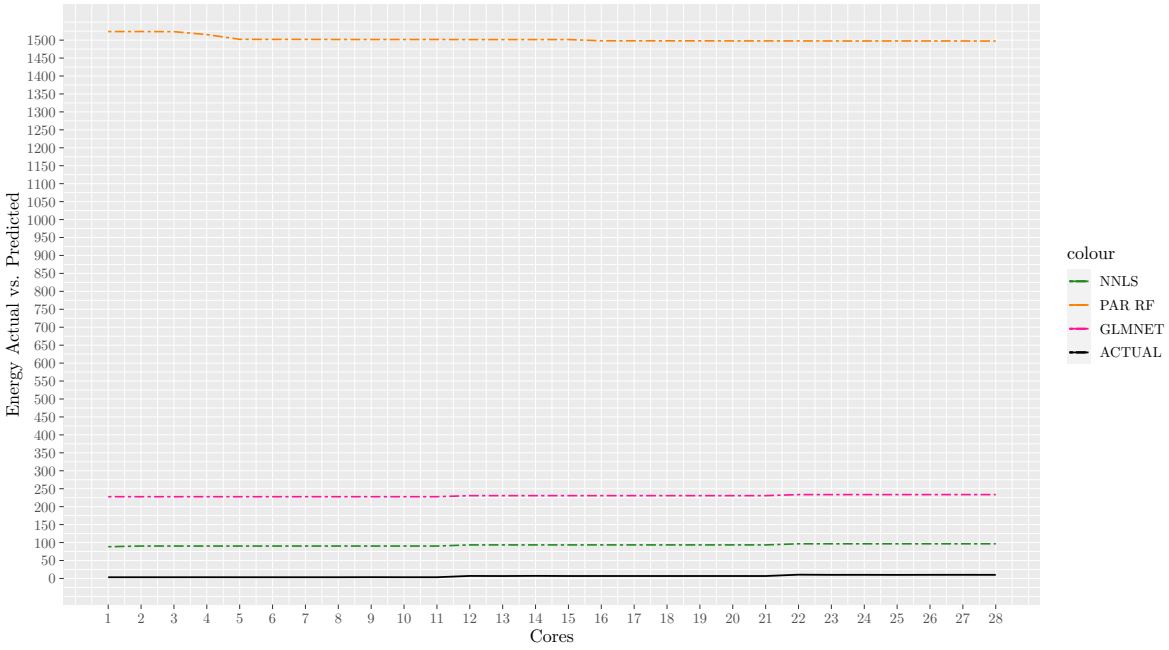
C/C++: Spectral-norm (Pthreads): Actual vs. Predictions
Input Size: 10.5k — 2.6 GHz



C/C++: Spectral-norm (Pthreads): Actual vs. Predictions
Input Size: 11.5k — 2.6 GHz

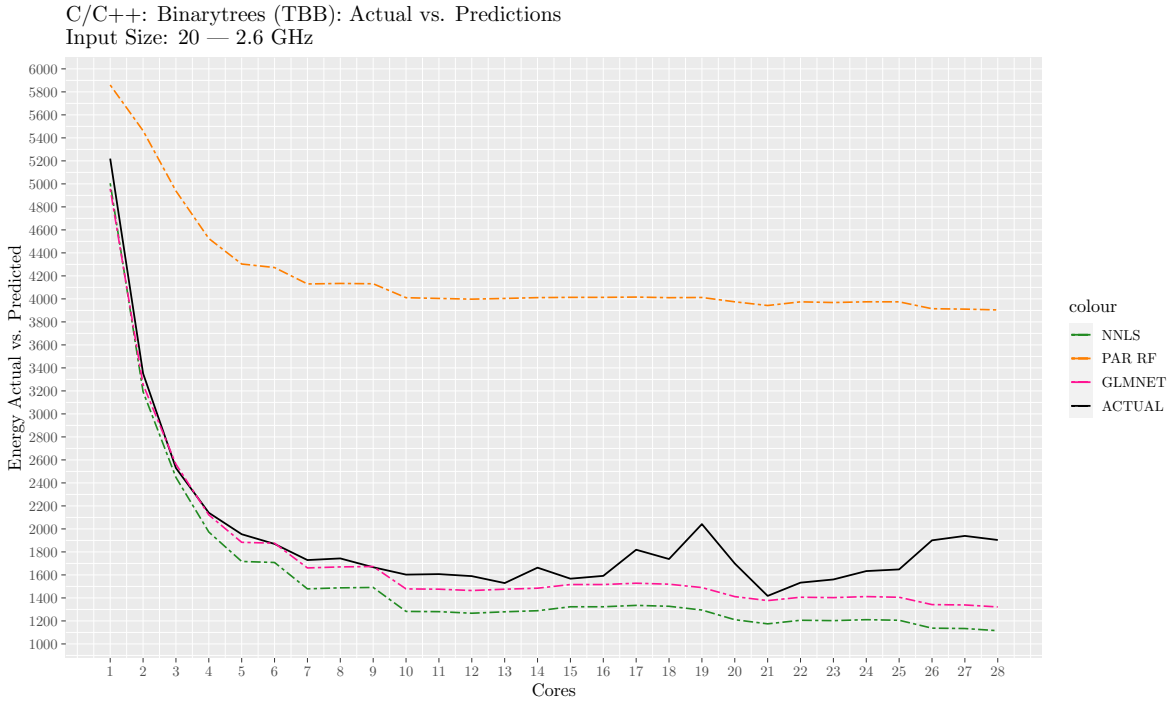


C/C++: Prime Decomposition (Pthreads): Actual vs. Predictions
Input Size: Multi — 2.6 GHz

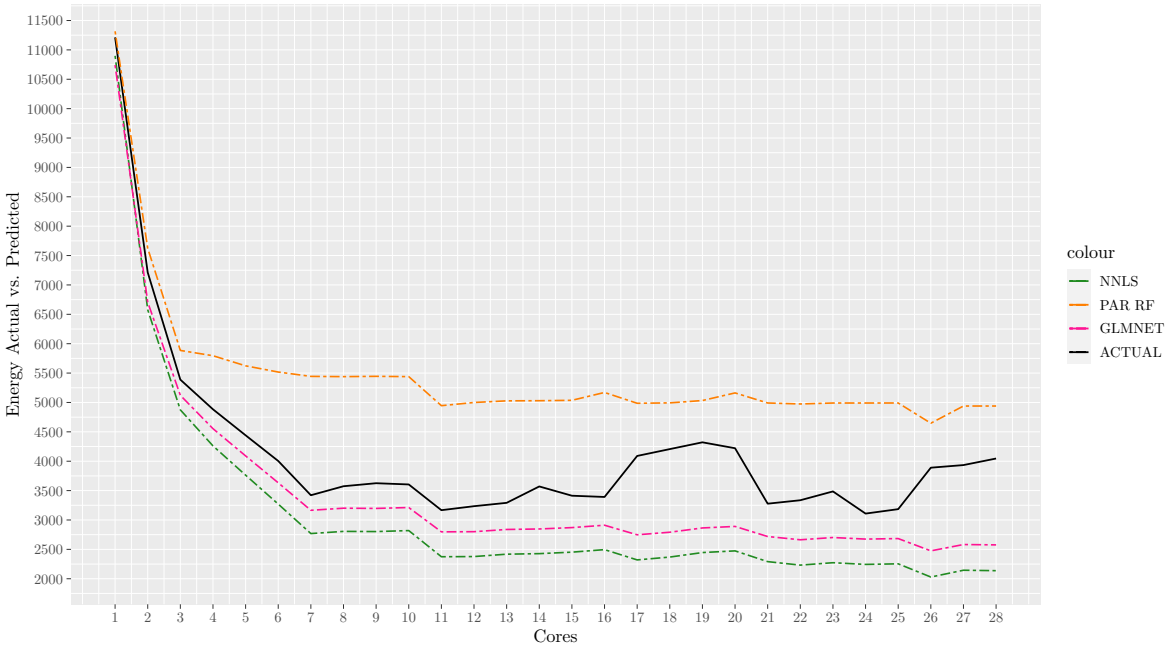


E.2 TBB Fitted vs. Actual Energy Consumption

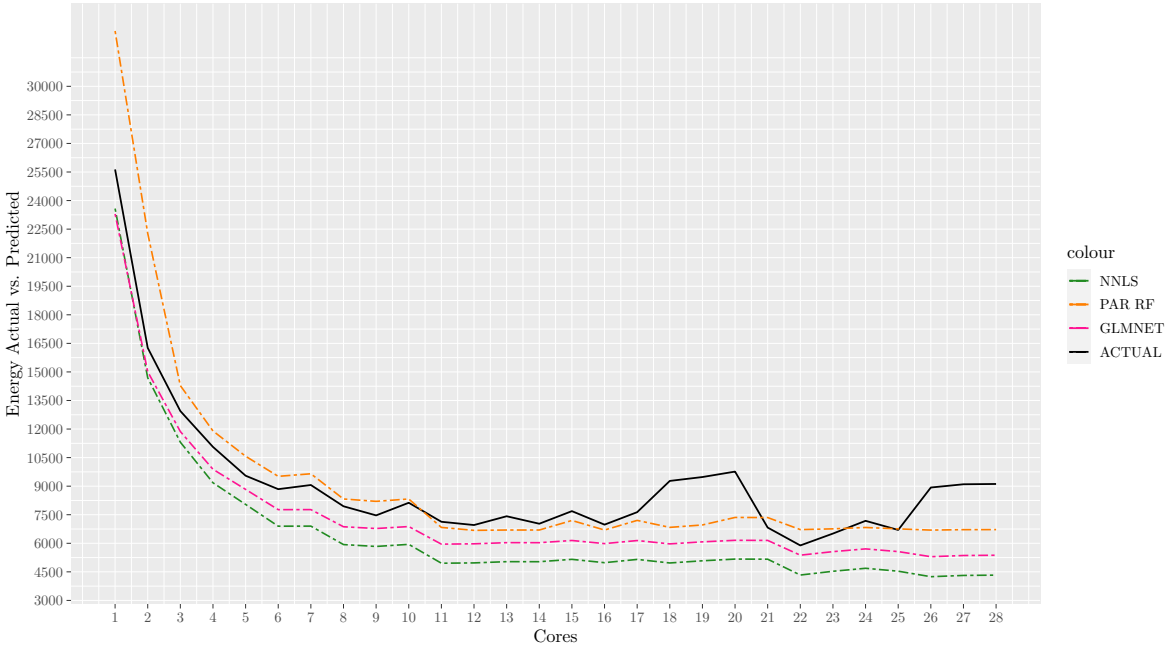
Figures below provide predictions for TBB sample of the prediction dataset. The figures show predictions for **NNLS**, **parRF**, and **GLMNET** against the actual sample energy consumption at highest frequency.



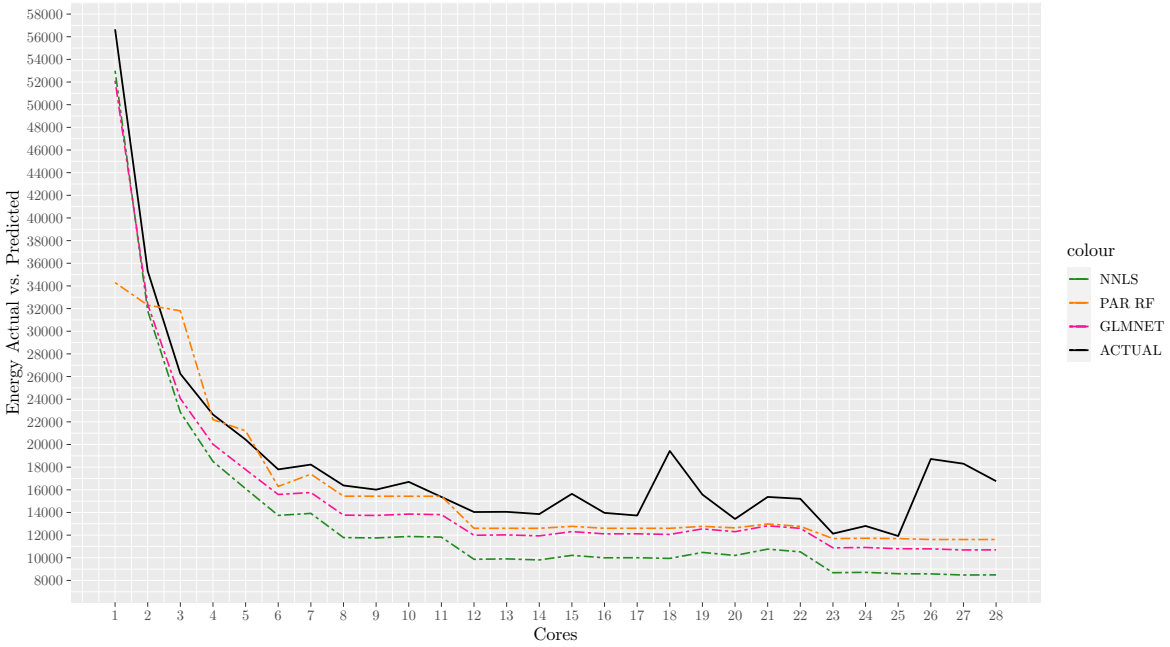
C/C++: Binarytrees (TBB): Actual vs. Predictions
Input Size: 21 — 2.6 GHz



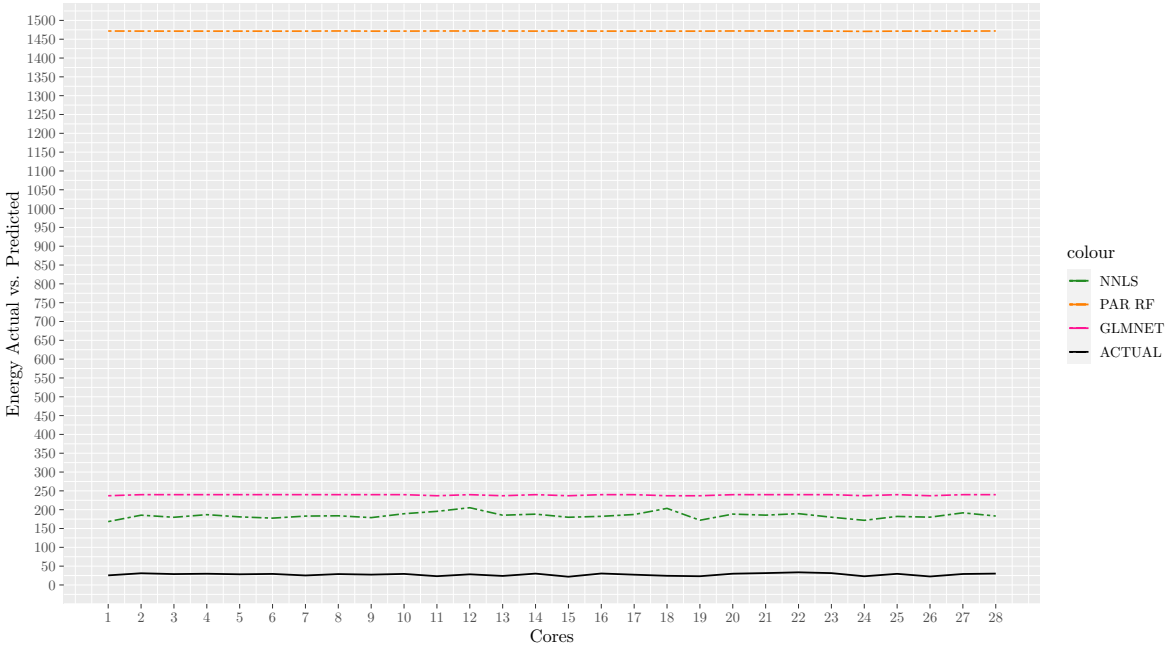
C/C++: Binarytrees (TBB): Actual vs. Predictions
Input Size: 22 — 2.6 GHz



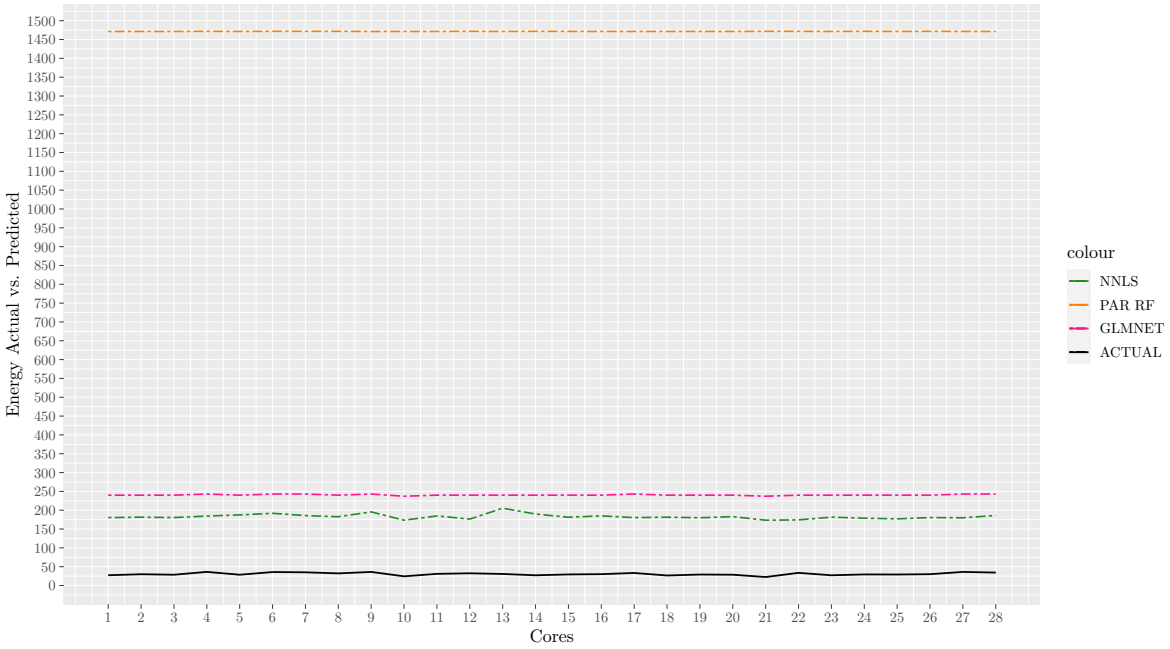
C/C++: Binarytrees (TBB): Actual vs. Predictions
Input Size: 23 — 2.6 GHz



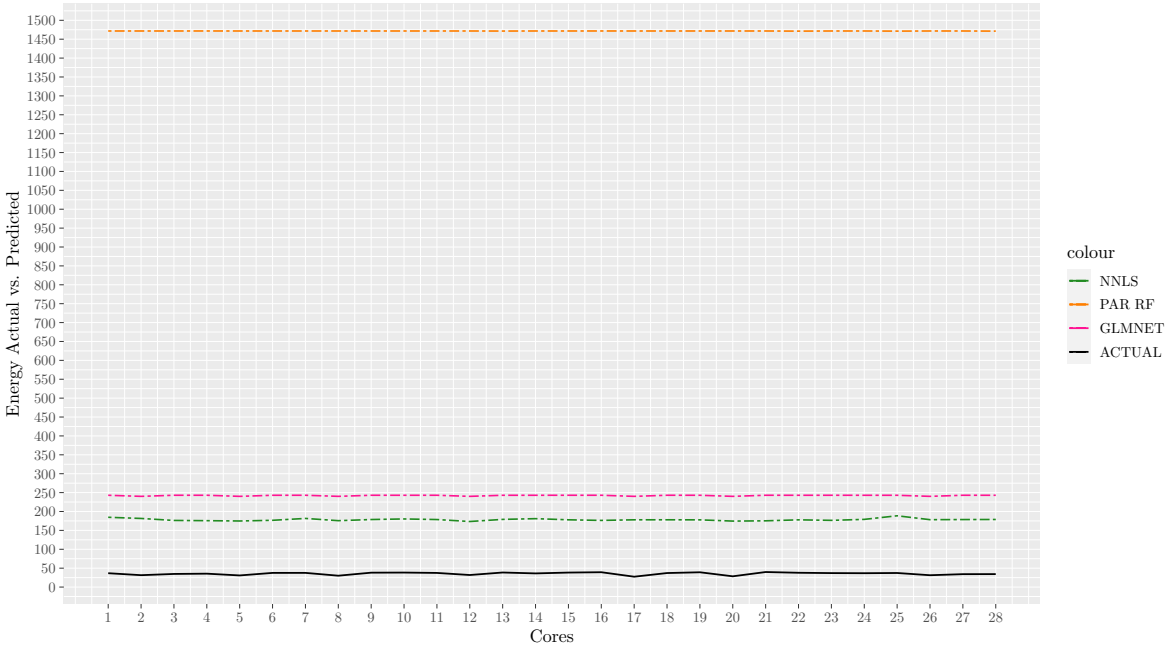
C/C++: Fasta (TBB): Actual vs. Predictions
Input Size: 900k — 2.6 GHz



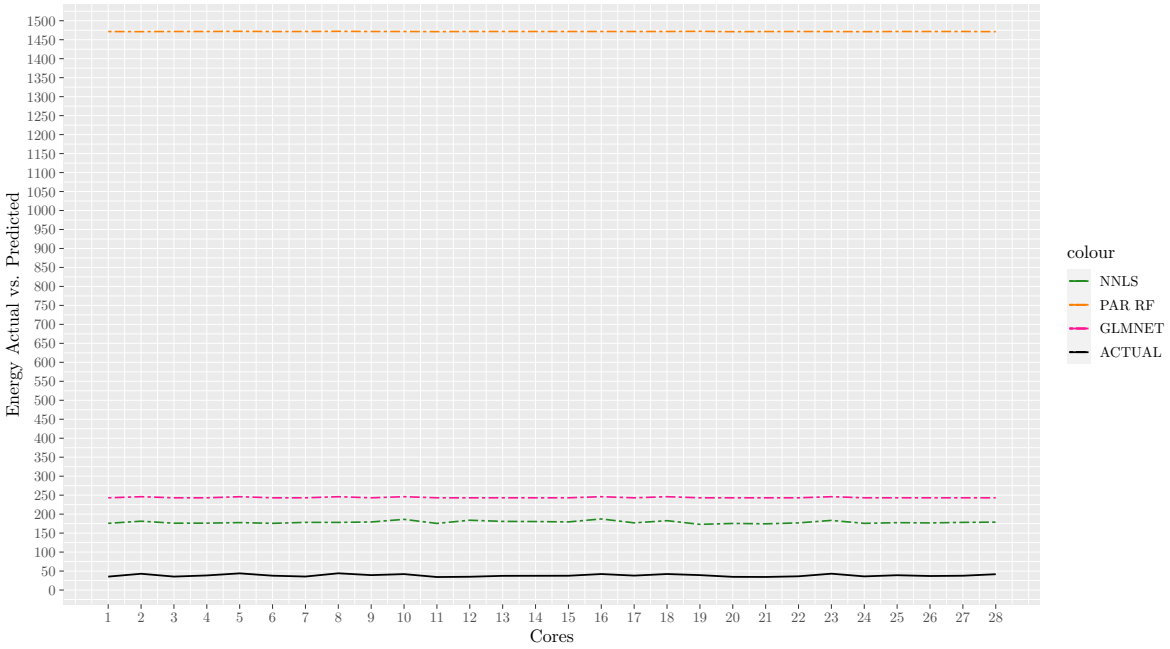
C/C++: Fasta (TBB): Actual vs. Predictions
Input Size: 1 mil — 2.6 GHz



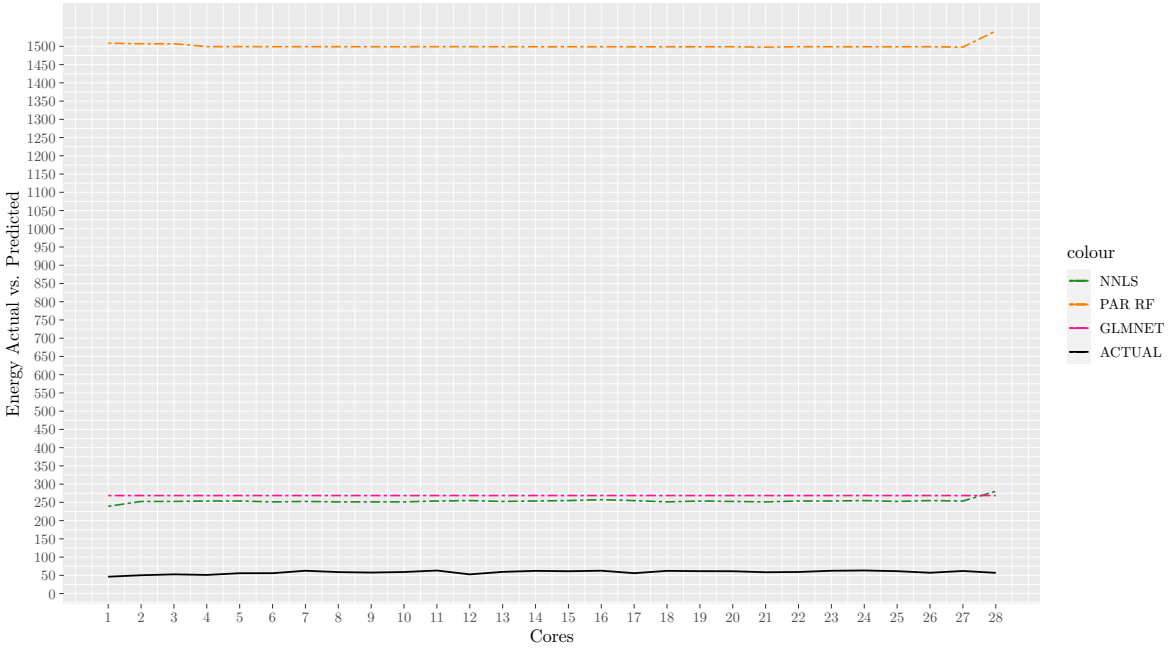
C/C++: Fasta (TBB): Actual vs. Predictions
Input Size: 1.1 mil — 2.6 GHz



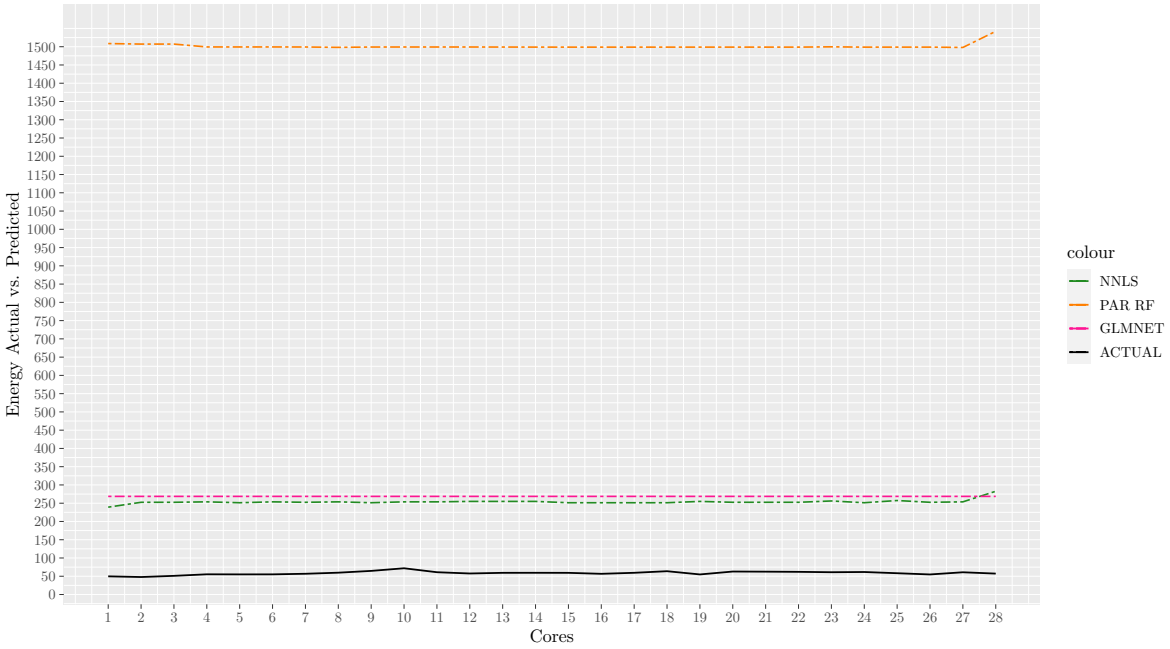
C/C++: Fasta (TBB): Actual vs. Predictions
Input Size: 1.2 mil — 2.6 GHz



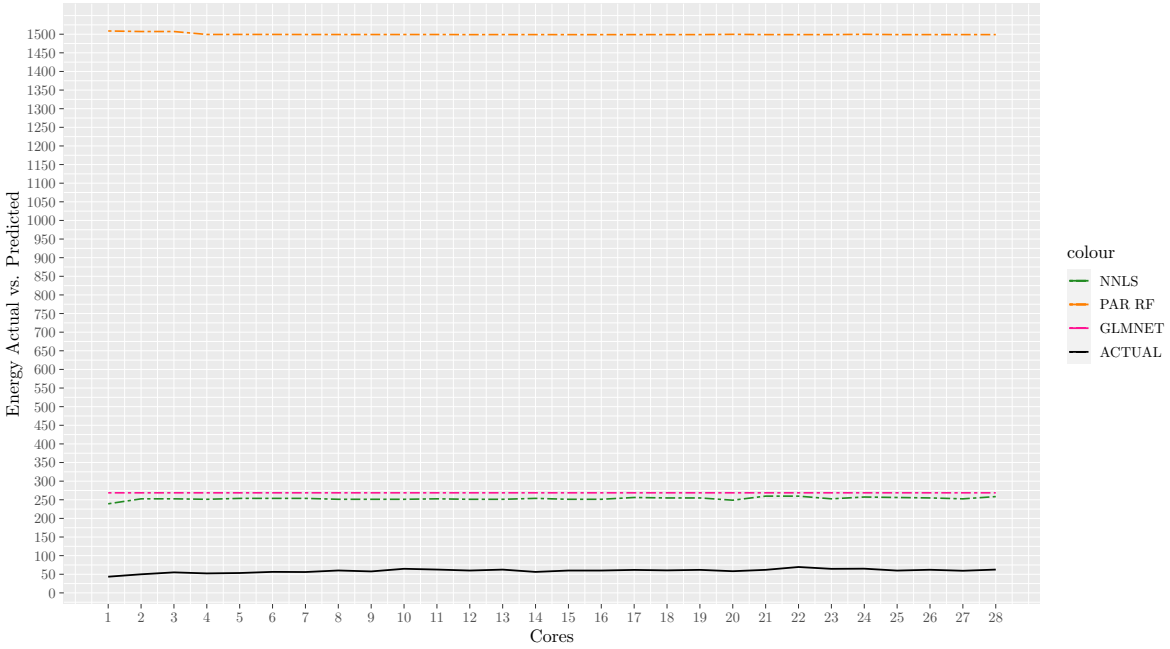
C/C++: Spectral-norm (TBB): Actual vs. Predictions
Input Size: 8.5k — 2.6 GHz



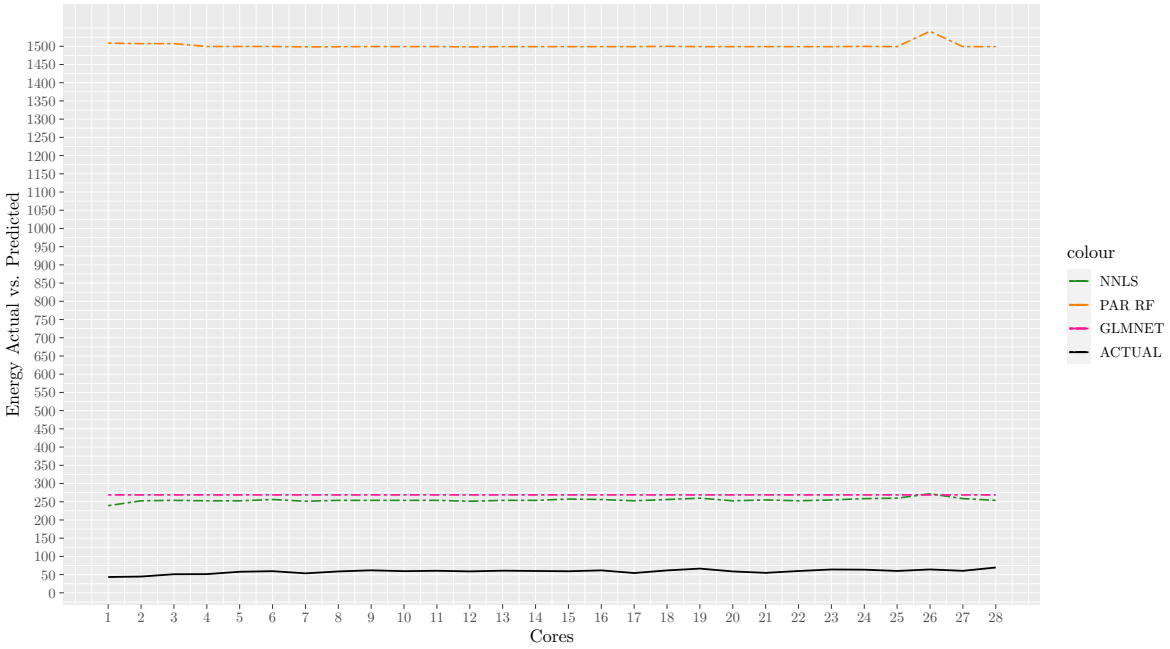
C/C++: Spectral-norm (TBB): Actual vs. Predictions
Input Size: 9.5k — 2.6 GHz



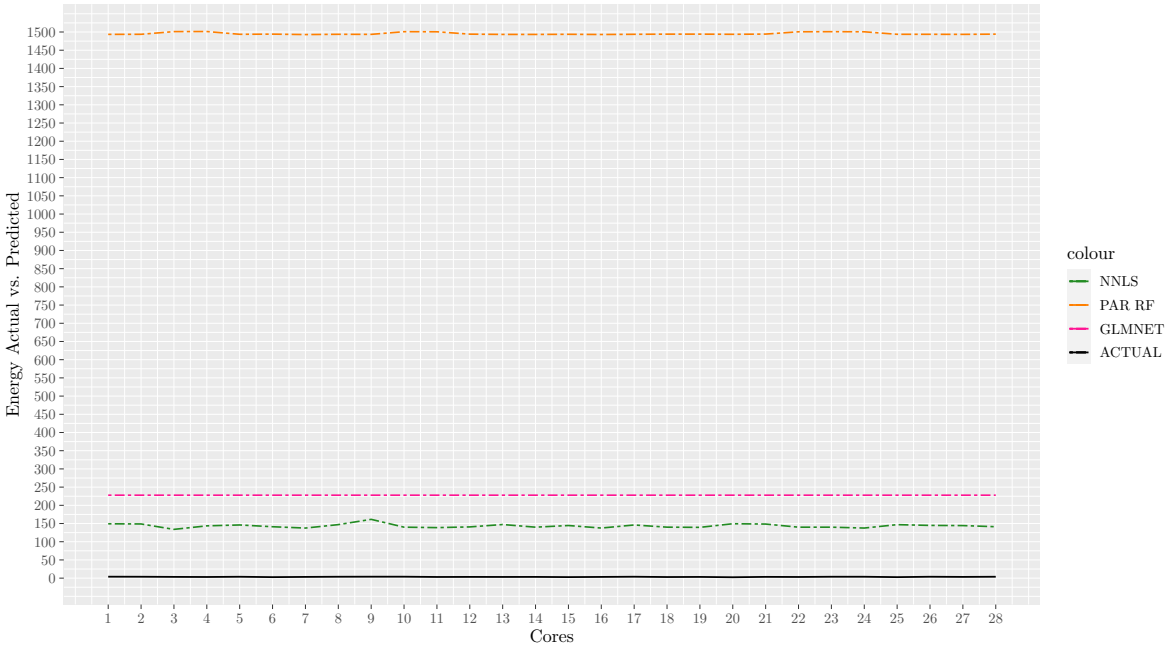
C/C++: Spectral-norm (TBB): Actual vs. Predictions
Input Size: 10.5k — 2.6 GHz



C/C++: Spectral-norm (TBB): Actual vs. Predictions
Input Size: 11.5k — 2.6 GHz



C/C++: Prime Decomposition (TBB): Actual vs. Predictions
Input Size: Multi — 2.6 GHz

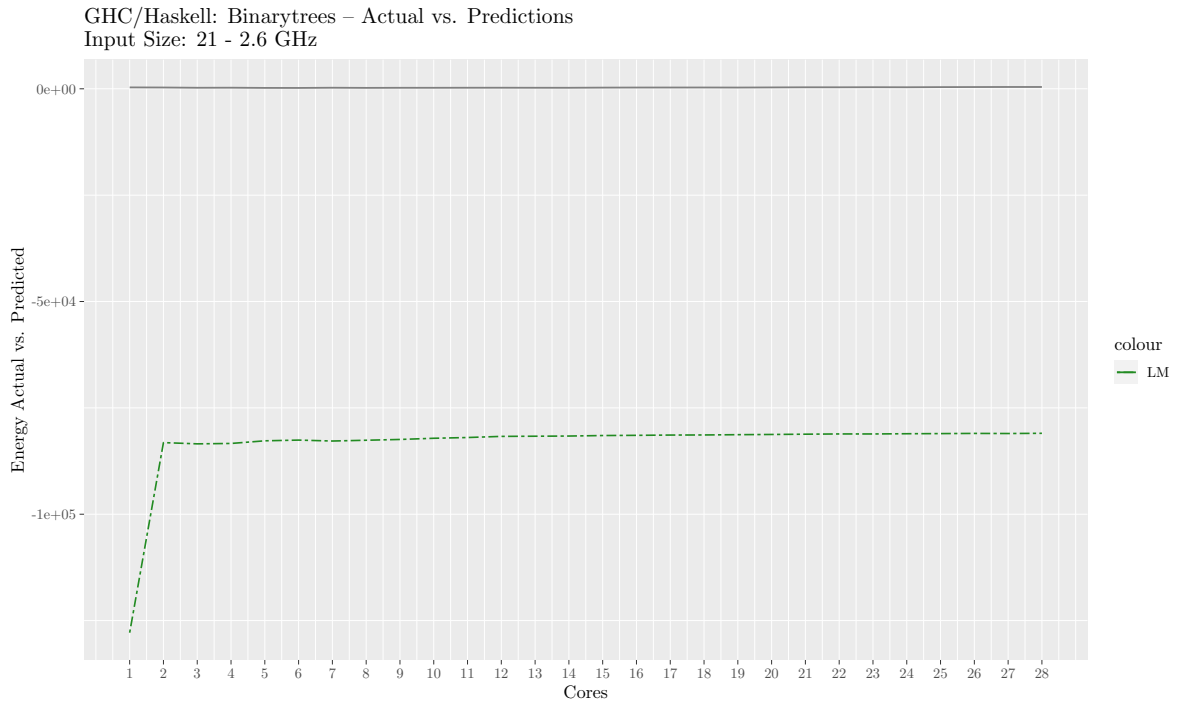


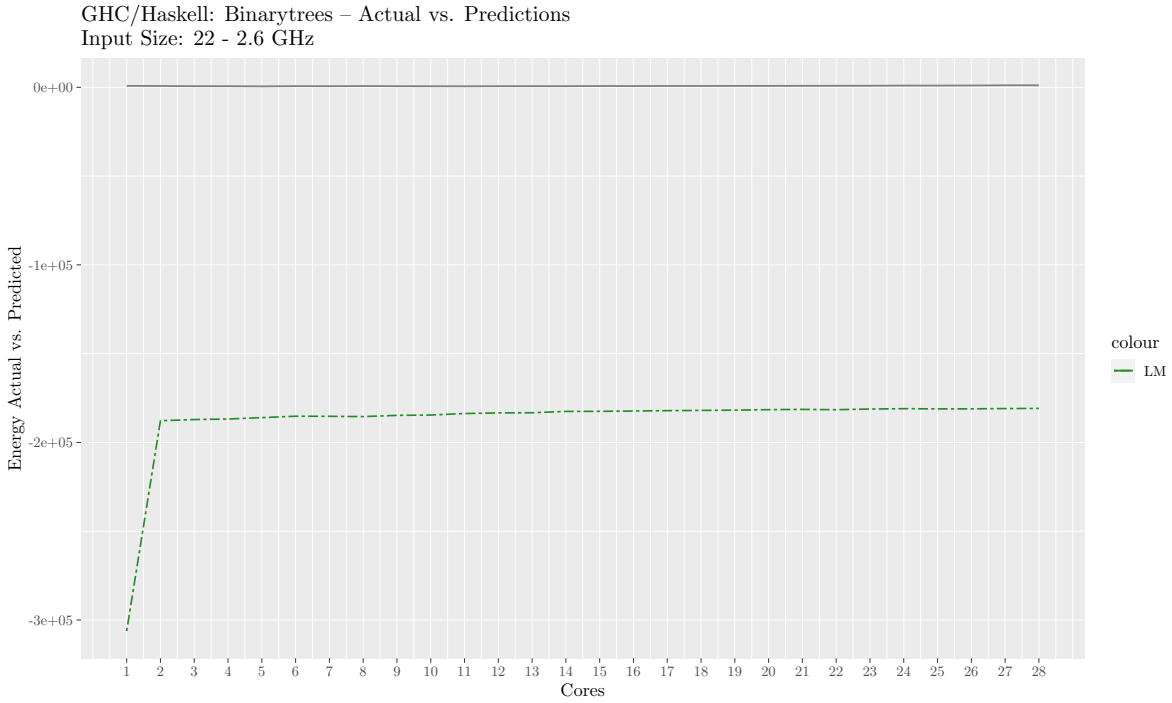
Appendix F

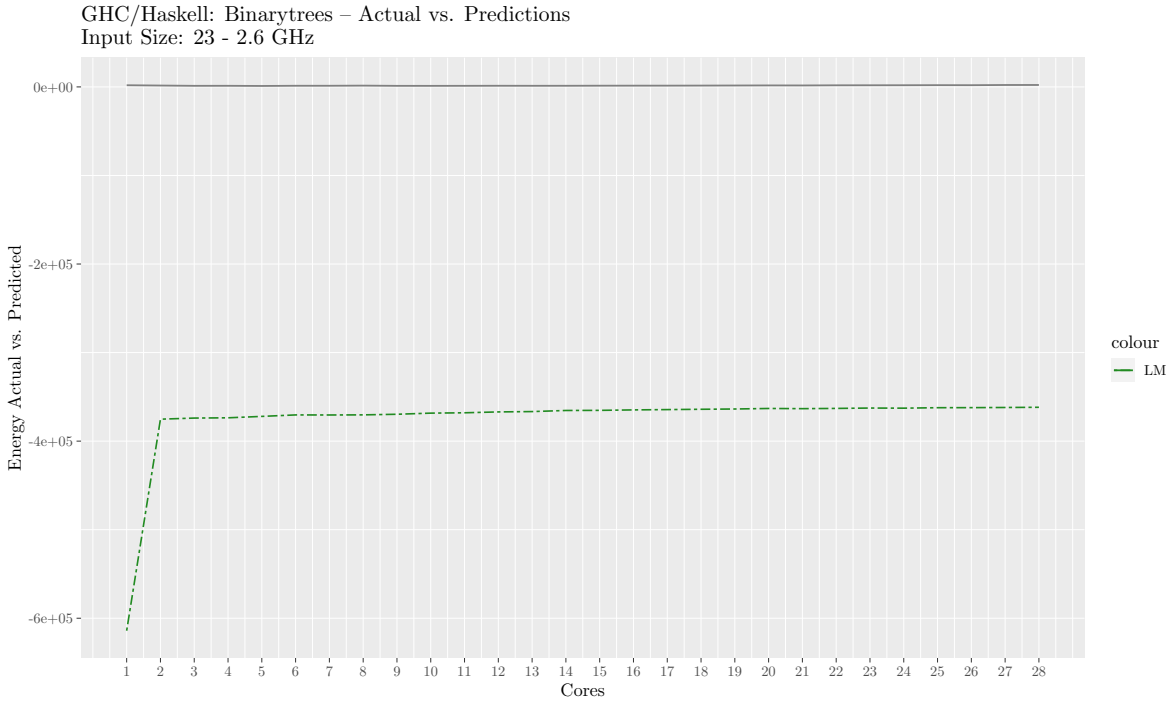
Examples: OpenMP and GHC Linear Model Prediction

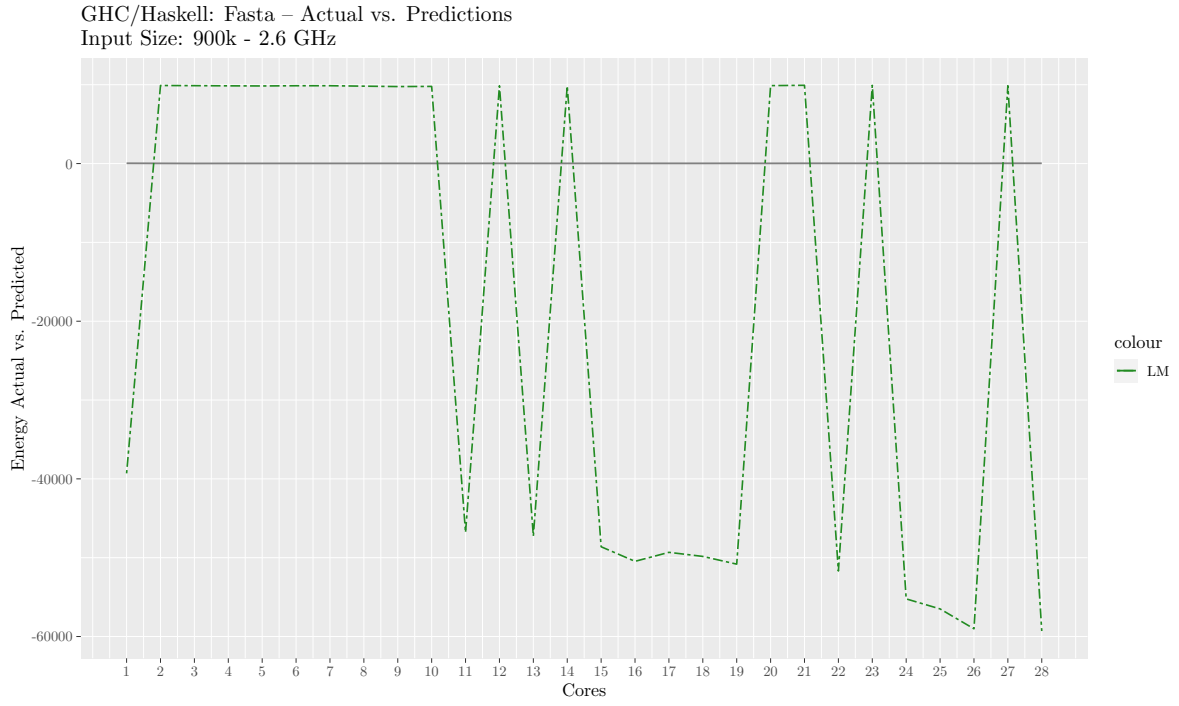
F.1 GHC Prediction Dataset – Actual vs. Linear Model Energy

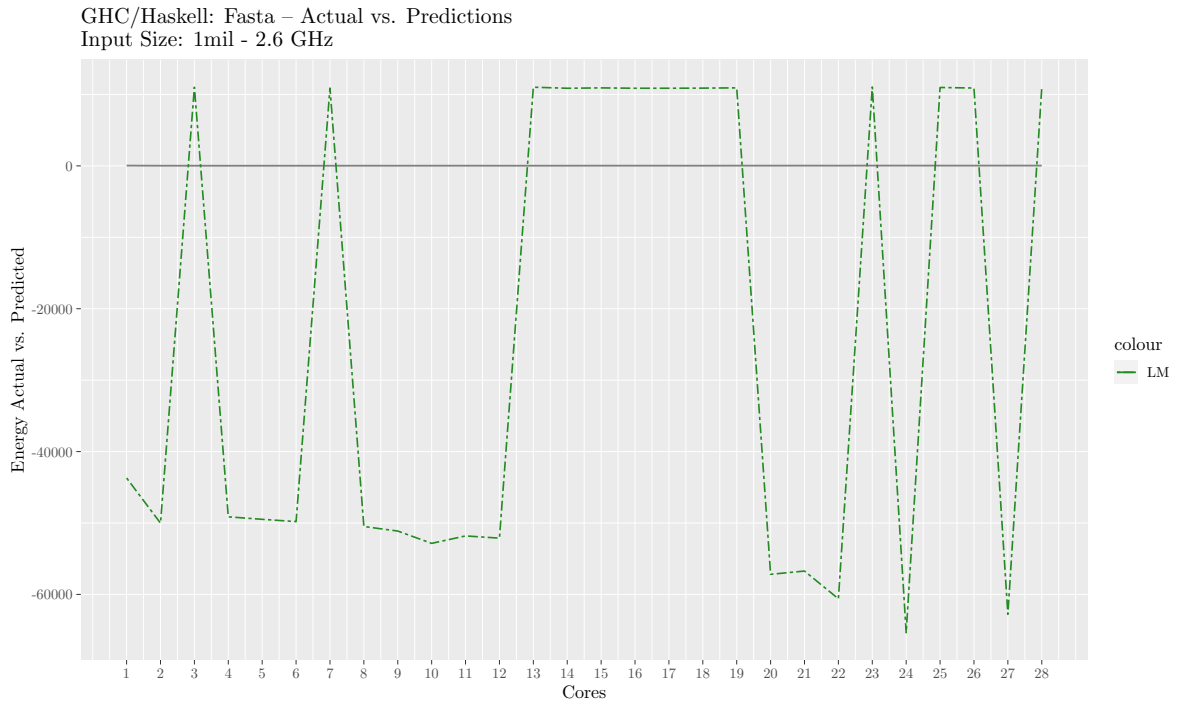
The following plots present the predicted values of benchmarks from Haskell's prediction dataset against a linear model constructed using the entire dataset's features.

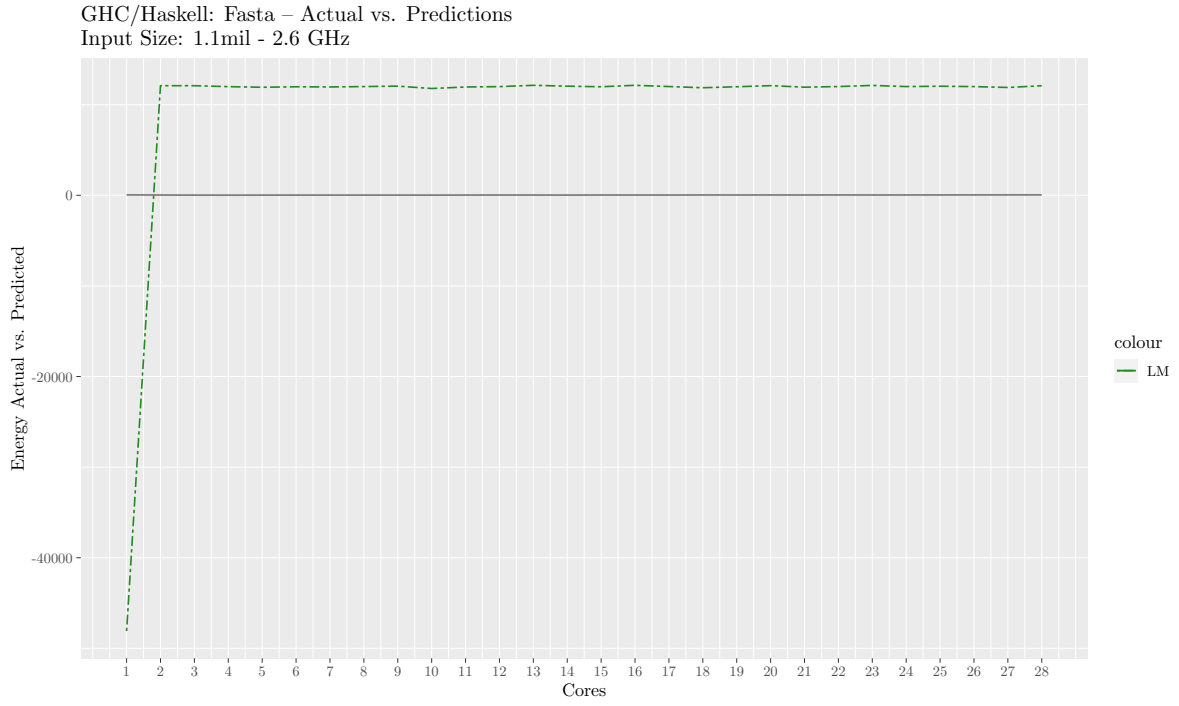


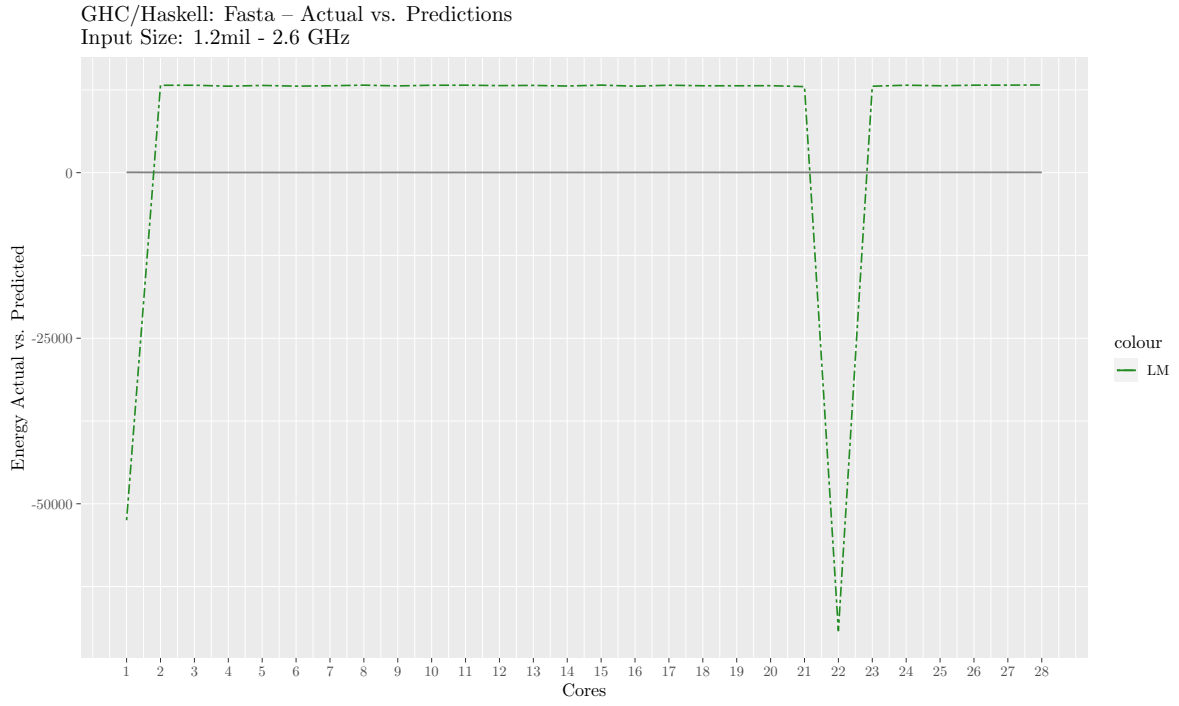

GHC/Haskell: Binarytrees – Actual vs. Predictions
Input Size: 22 - 2.6 GHz
Energy Actual vs. Predicted
0e+00
-1e+05
-2e+05
-3e+05
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM


GHC/Haskell: Binarytrees – Actual vs. Predictions
Input Size: 23 - 2.6 GHz
Energy Actual vs. Predicted
0e+00
-2e+05
-4e+05
-6e+05
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM

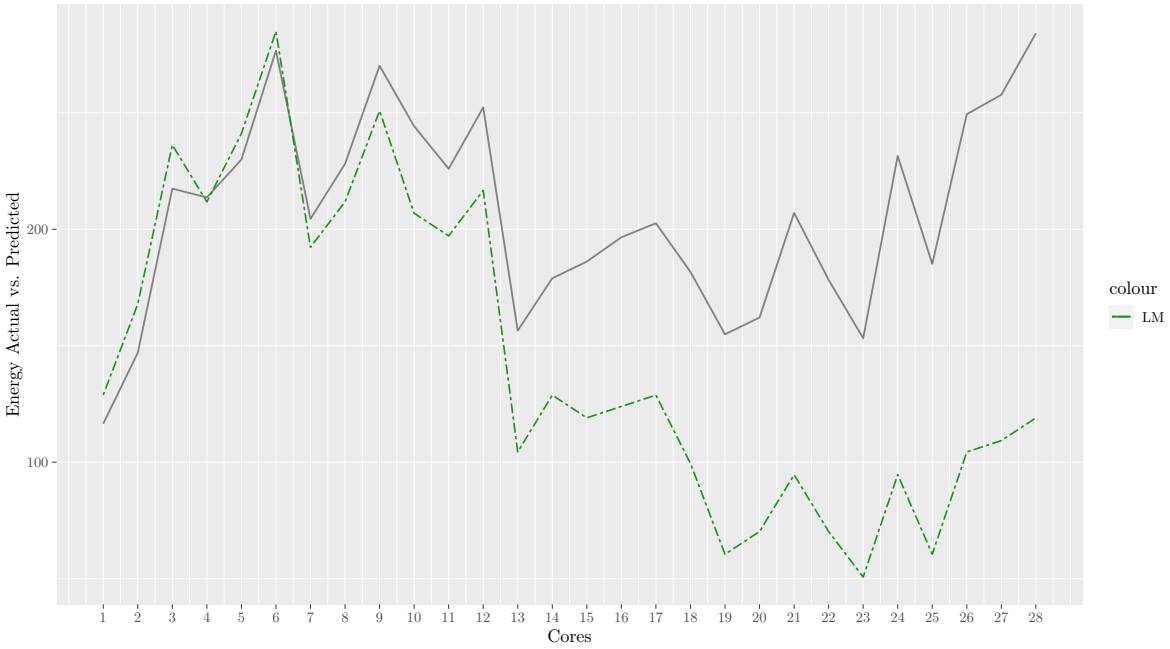

GHC/Haskell: Fasta – Actual vs. Predictions
Input Size: 900k - 2.6 GHz
Energy Actual vs. Predicted
0
-20000
-40000
-60000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM


GHC/Haskell: Fasta – Actual vs. Predictions
Input Size: 1mil - 2.6 GHz
Energy Actual vs. Predicted
0
-20000
-40000
-60000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM

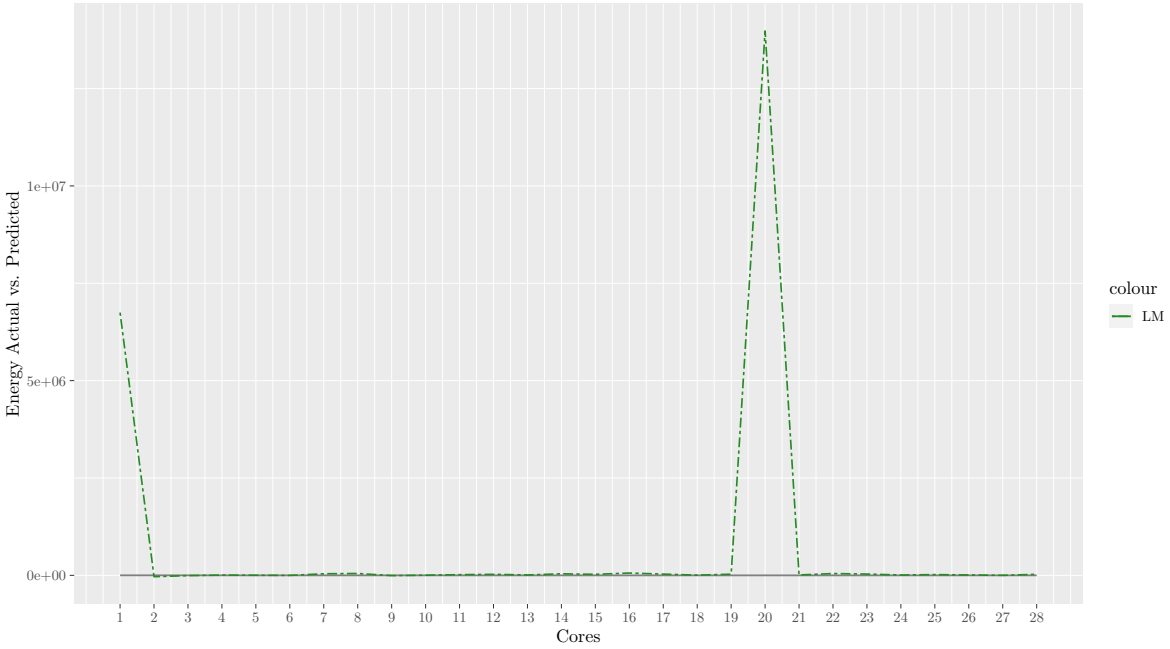

GHC/Haskell: Fasta – Actual vs. Predictions
Input Size: 1.1mil - 2.6 GHz
Energy Actual vs. Predicted
0
-20000
-40000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM

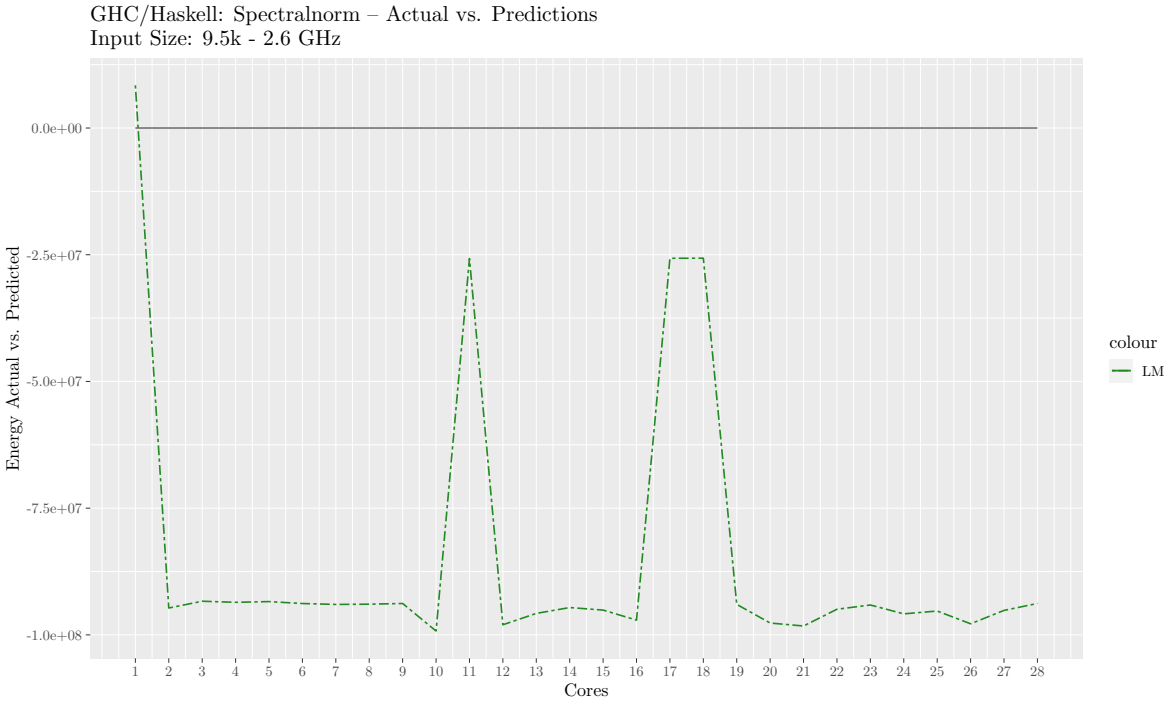

GHC/Haskell: Fasta – Actual vs. Predictions
Input Size: 1.2mil - 2.6 GHz
Energy Actual vs. Predicted
0
-25000
-50000
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM

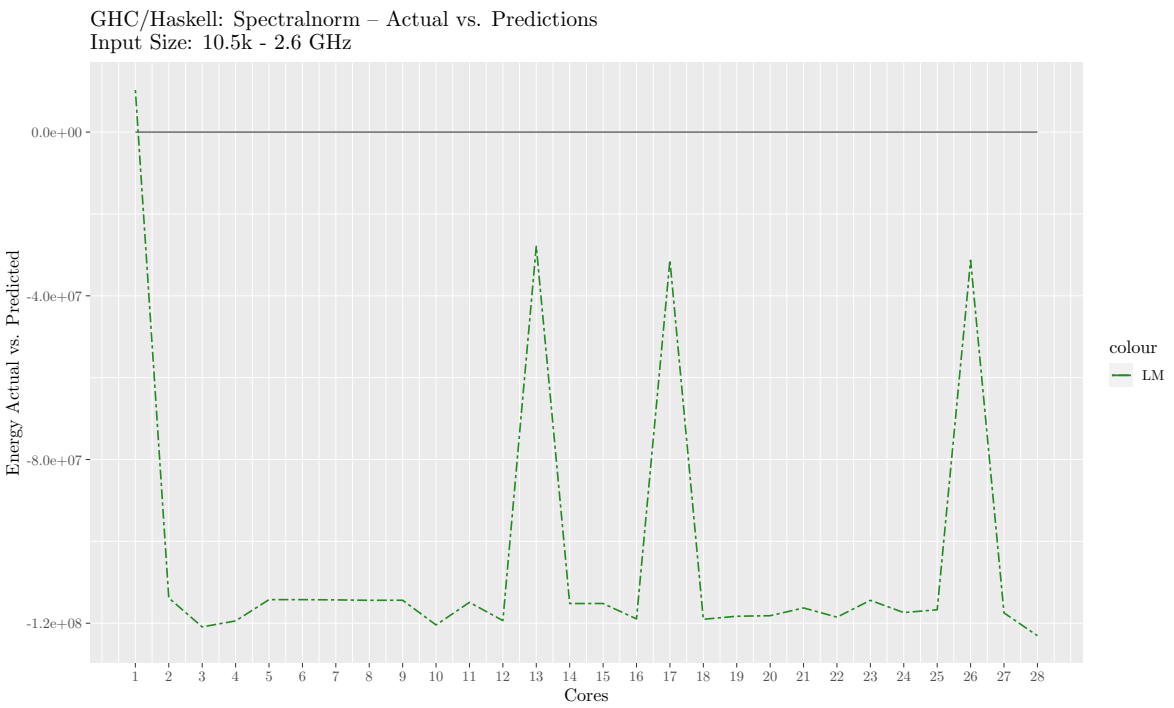
GHC/Haskell: Prime Decomposition – Actual vs. Predictions
Input Size: default - 2.6 GHz

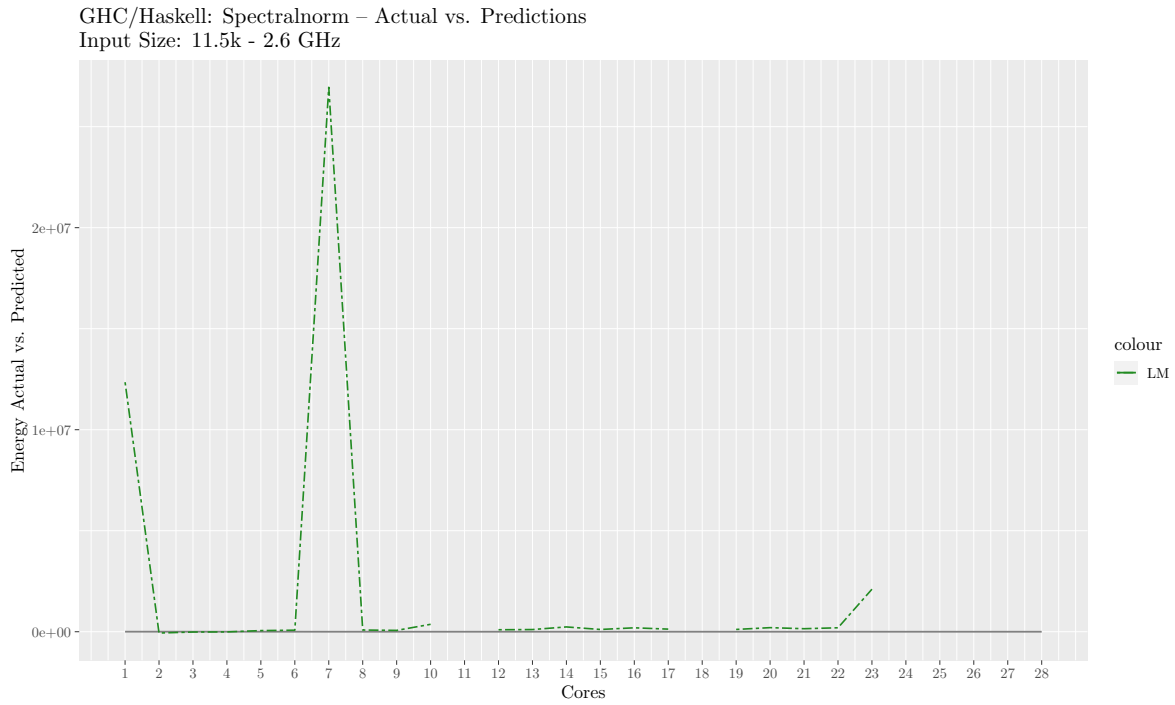


GHC/Haskell: Spectralnorm – Actual vs. Predictions
Input Size: 8.5k - 2.6 GHz



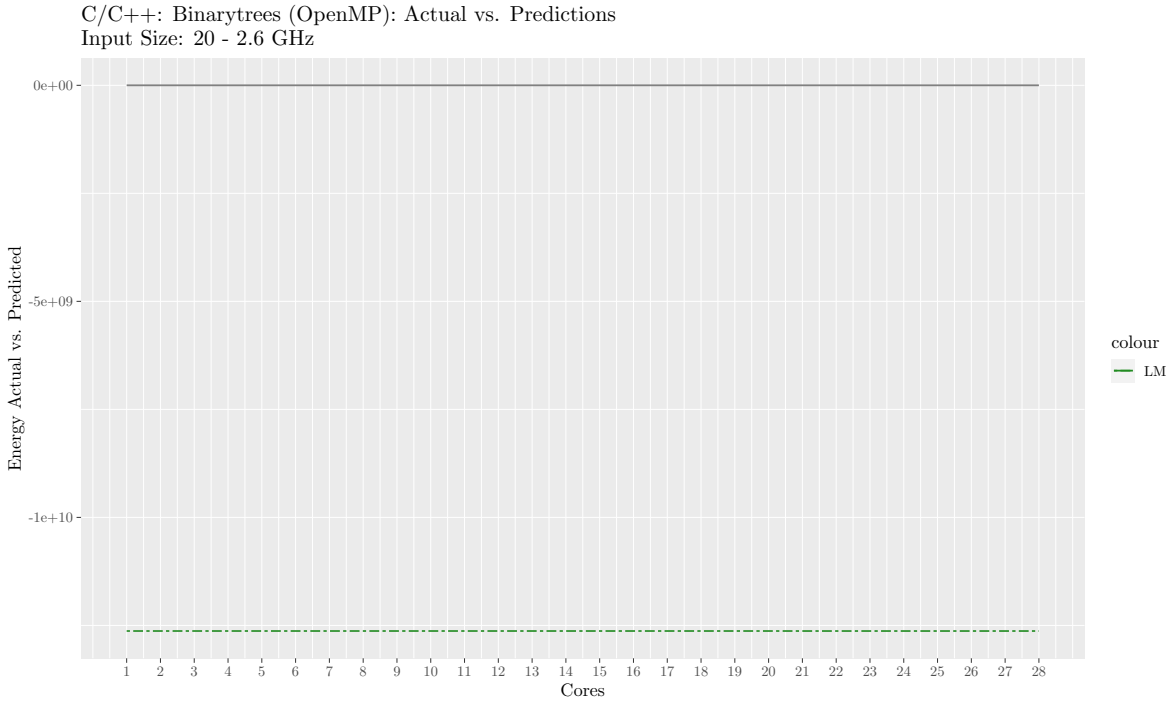

GHC/Haskell: Spectralnorm – Actual vs. Predictions
Input Size: 9.5k - 2.6 GHz
Energy Actual vs. Predicted
0.0e+00
-2.5e+07
-5.0e+07
-7.5e+07
-1.0e+08
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM

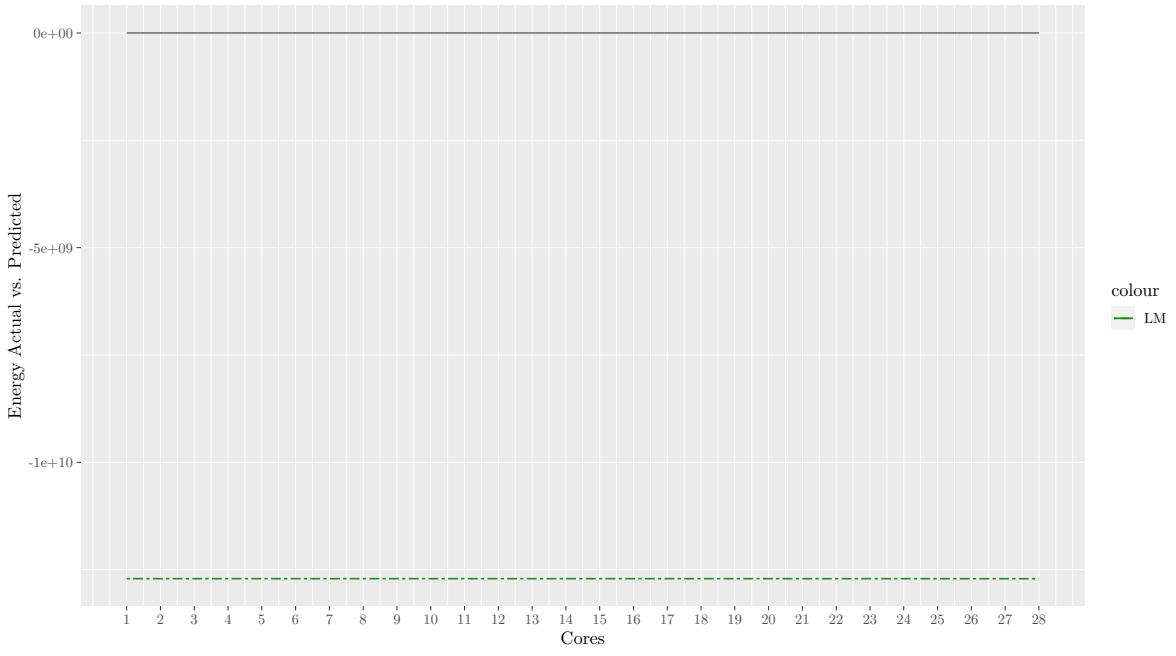

GHC/Haskell: Spectralnorm – Actual vs. Predictions
Input Size: 10.5k - 2.6 GHz
Energy Actual vs. Predicted
0.0e+00
-4.0e+07
-8.0e+07
-1.2e+08
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM



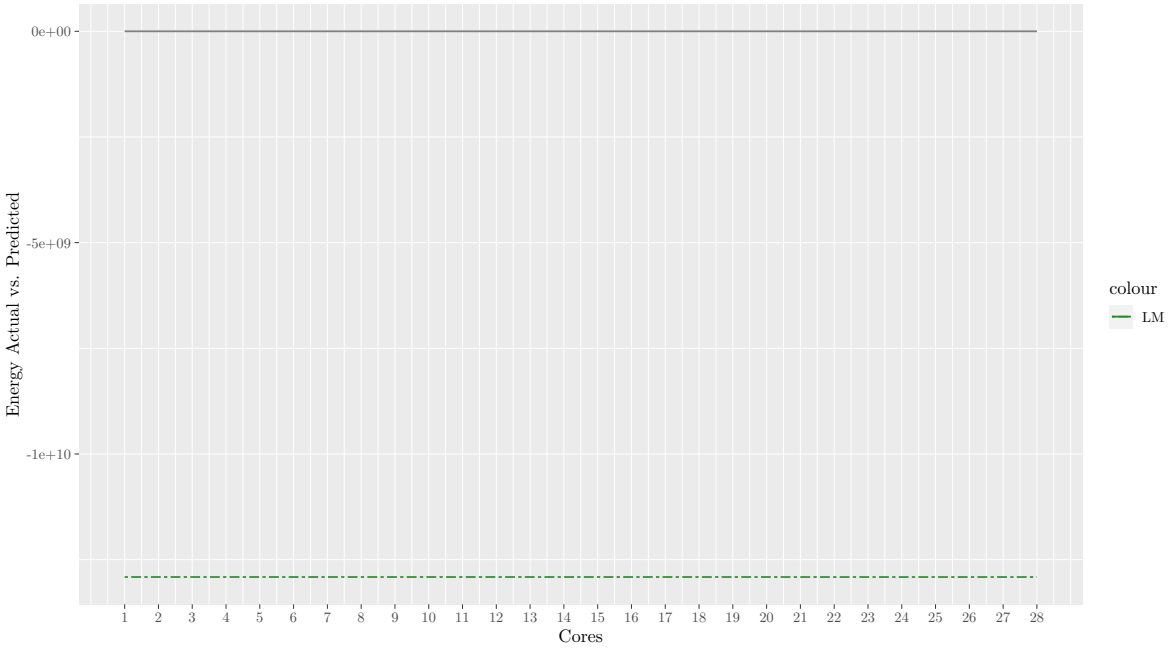
F.2 C/C++ Prediction Dataset – Actual vs. Linear Model Energy

The following plots present the predicted values of benchmarks from C/C++'s prediction dataset against a linear model constructed using the entire dataset's features.

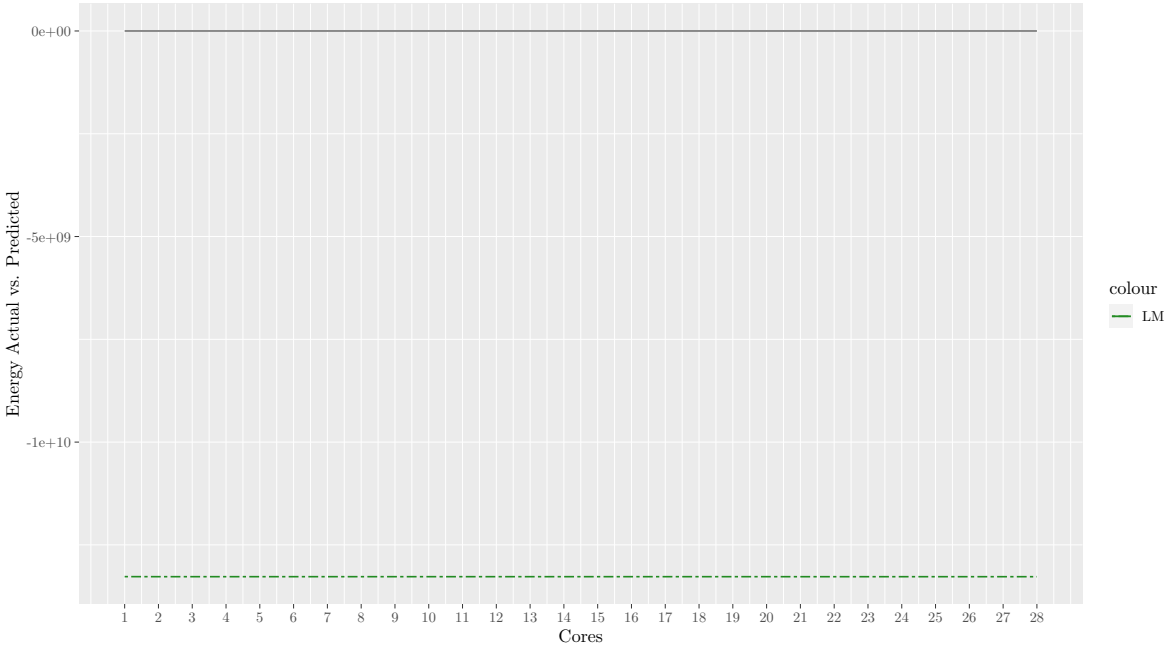

C/C++: Binarytrees (OpenMP): Actual vs. Predictions
Input Size: 20 - 2.6 GHz
Energy Actual vs. Predicted
0e+00
-5e+09
-1e+10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM


C/C++: Binarytrees (OpenMP): Actual vs. Predictions
Input Size: 21 - 2.6 GHz
Energy Actual vs. Predicted
0e+00
-5e+09
-1e+10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM

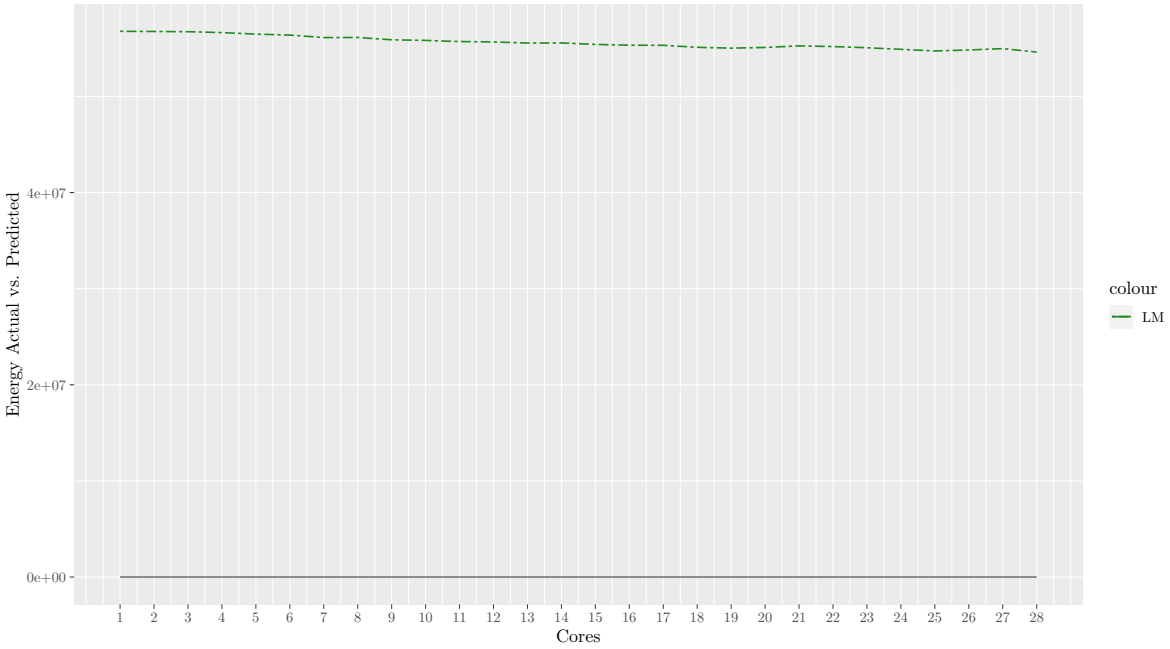
C/C++: Binarytrees (OpenMP): Actual vs. Predictions
Input Size: 22 - 2.6 GHz



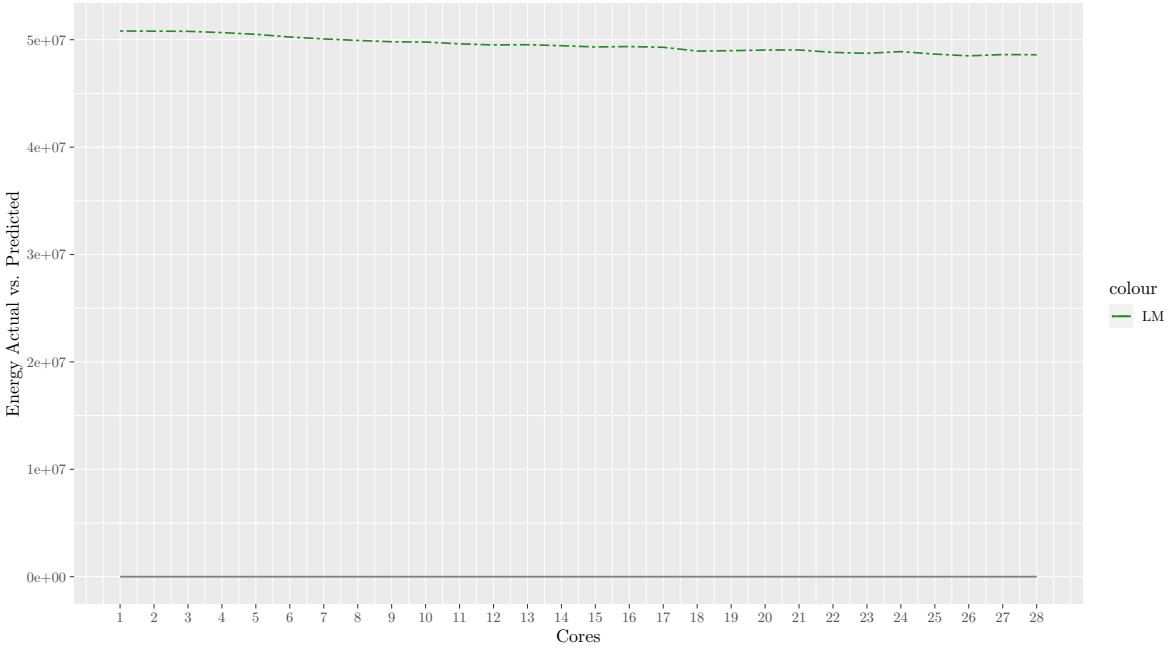
C/C++: Binarytrees (OpenMP): Actual vs. Predictions
Input Size: 23 - 2.6 GHz



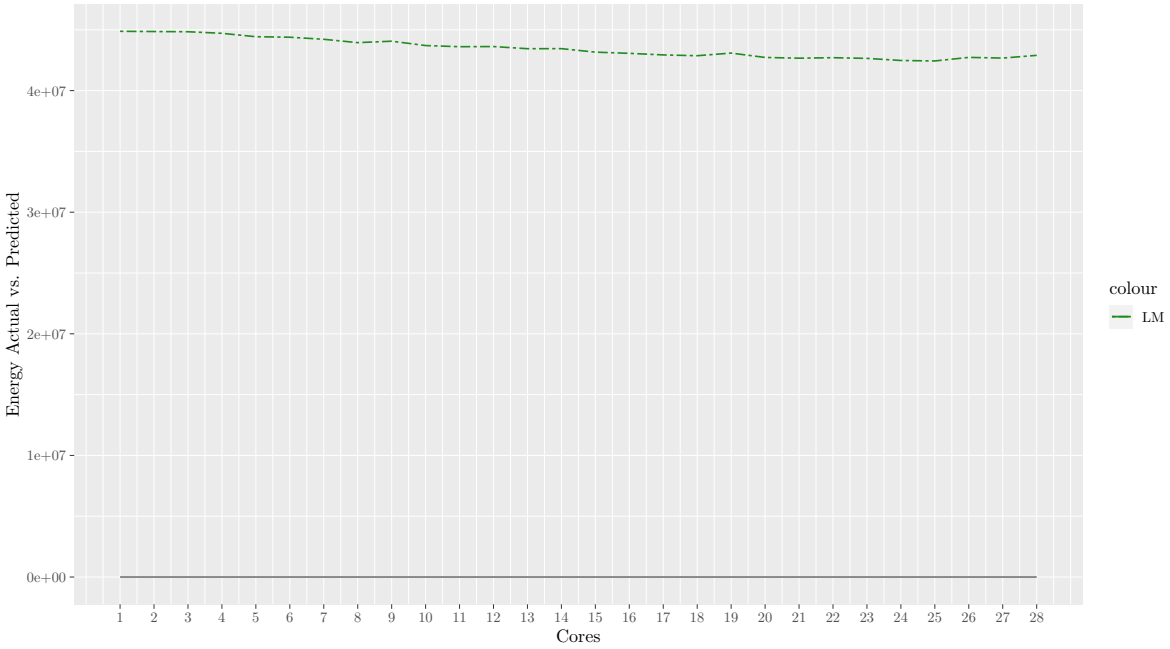
C/C++: Fasta (OpenMP): Actual vs. Predictions
Input Size: 900k - 2.6 GHz



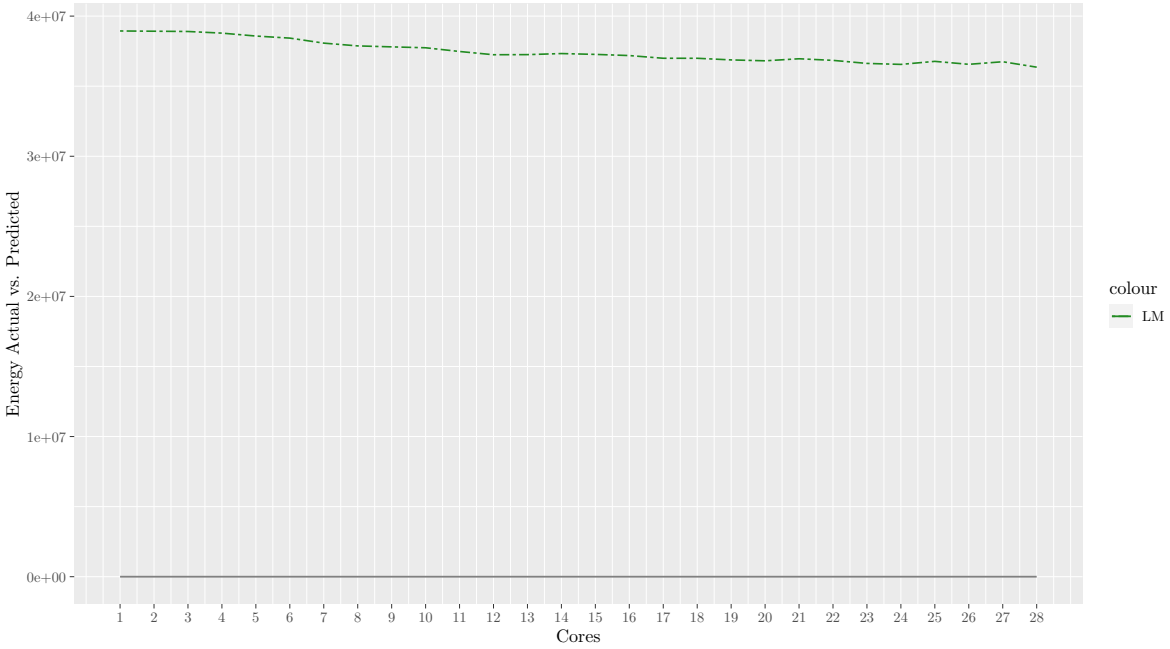
C/C++: Fasta (OpenMP): Actual vs. Predictions
Input Size: 1mil - 2.6 GHz



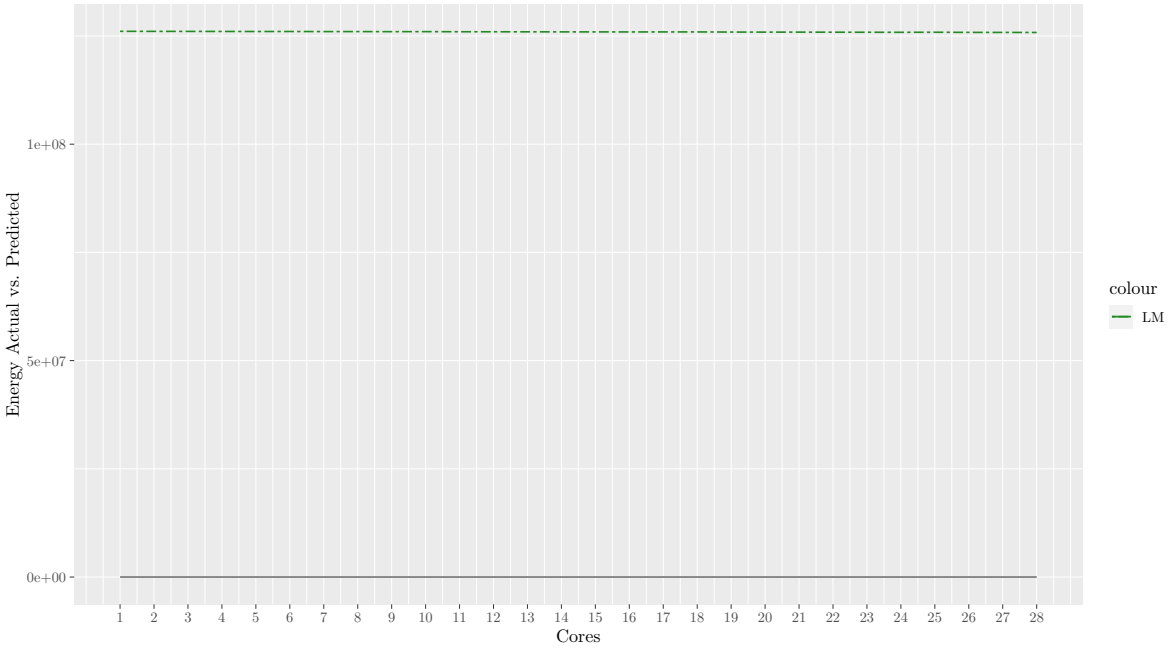
C/C++: Fasta (OpenMP): Actual vs. Predictions
Input Size: 1.1mil - 2.6 GHz



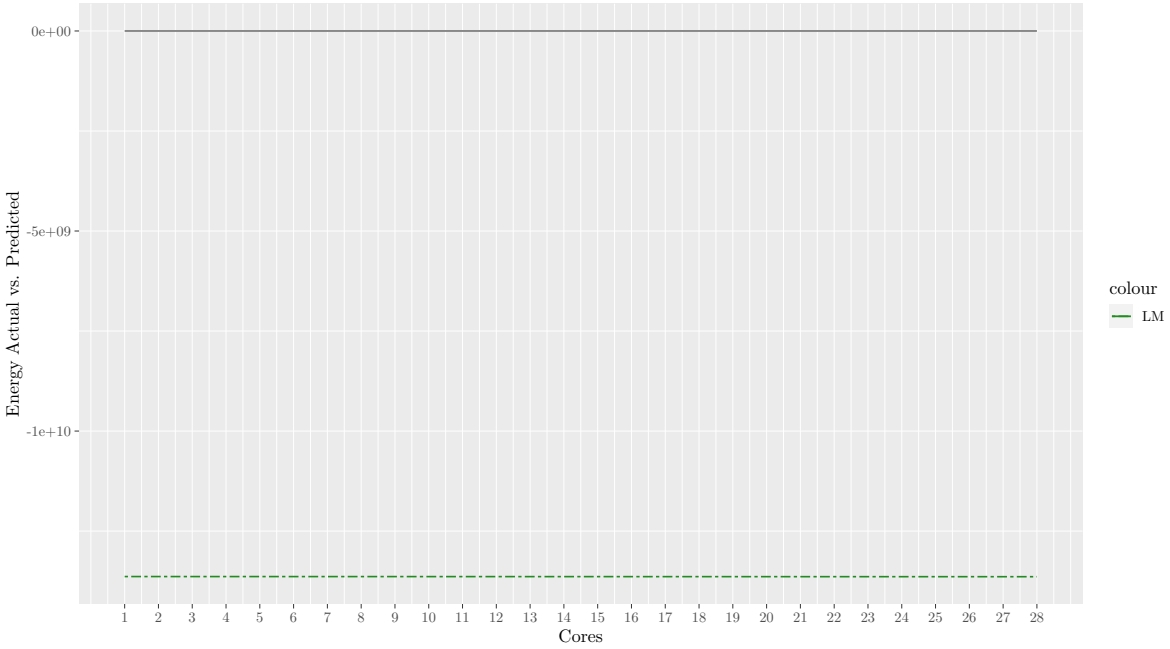
C/C++: Fasta (OpenMP): Actual vs. Predictions
Input Size: 1.2mil - 2.6 GHz



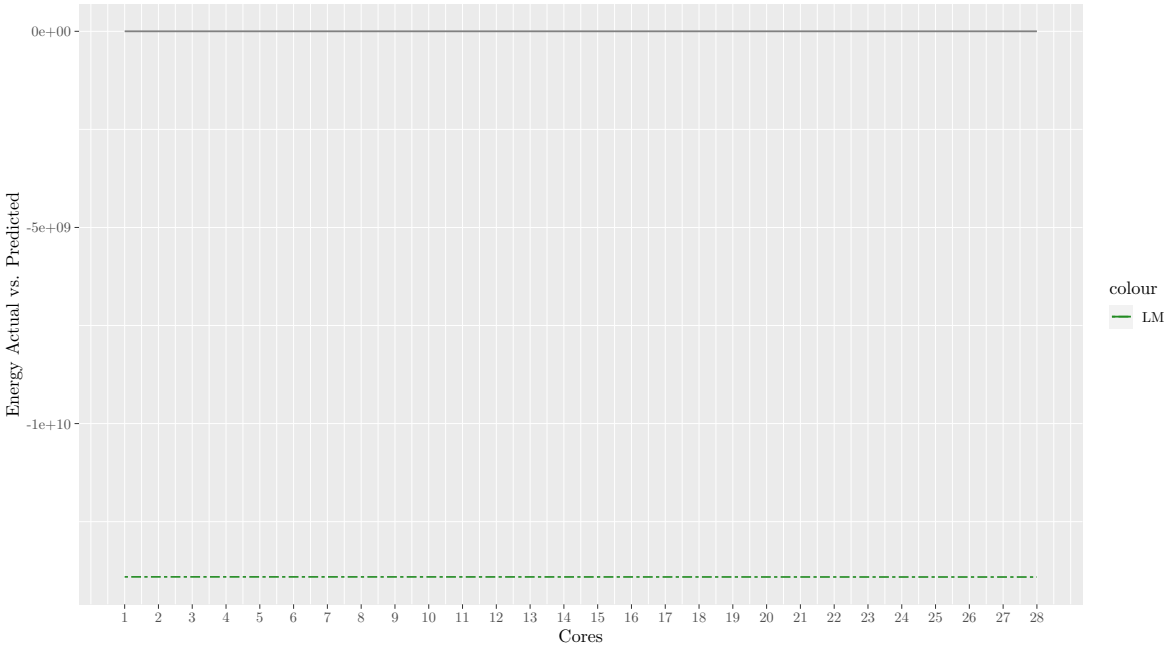
C/C++: Prime Decomposition (OpenMP): Actual vs. Predictions
Input Size: default - 2.6 GHz



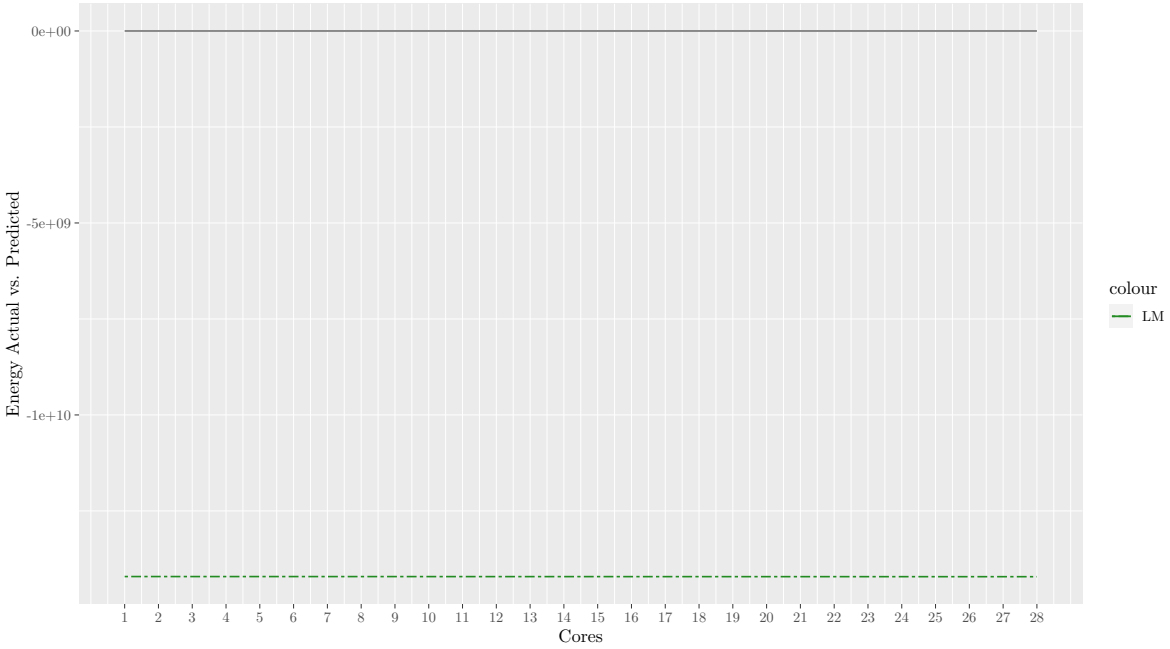
C/C++: Spectralnorm (OpenMP): Actual vs. Predictions
Input Size: 8.5k - 2.6 GHz

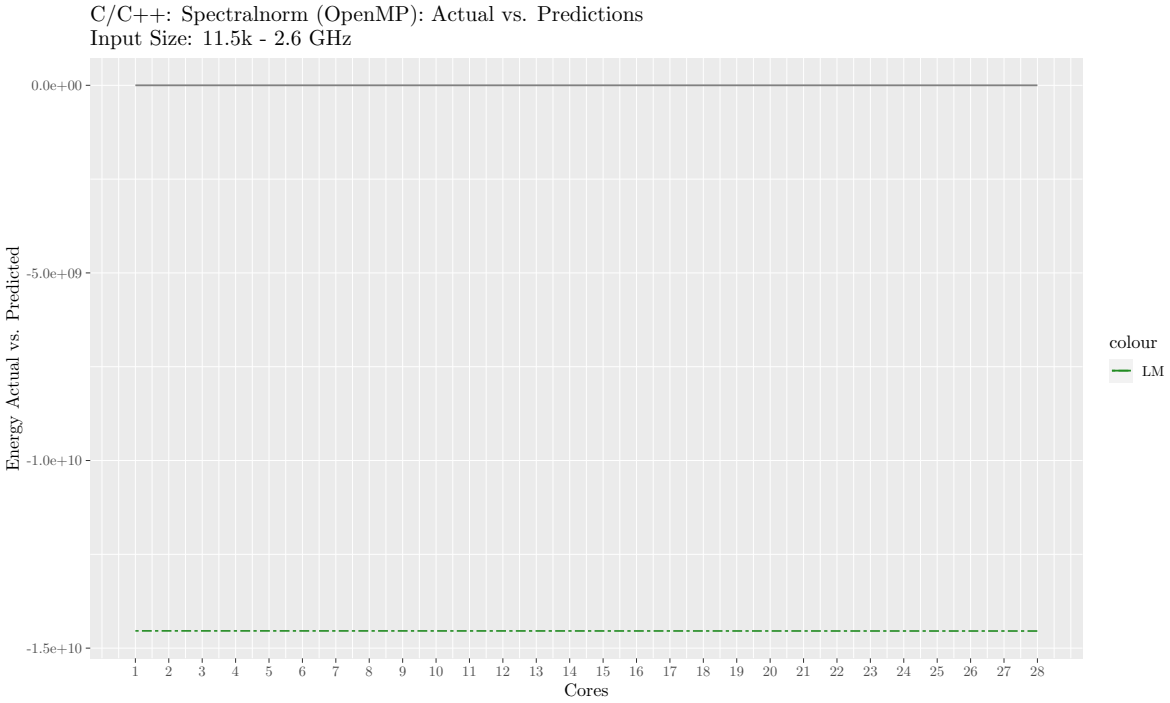


C/C++: Spectralnorm (OpenMP): Actual vs. Predictions
Input Size: 9.5k - 2.6 GHz



C/C++: Spectralnorm (OpenMP): Actual vs. Predictions
Input Size: 10.5k - 2.6 GHz




C/C++: Spectralnorm (OpenMP): Actual vs. Predictions
Input Size: 11.5k - 2.6 GHz
Energy Actual vs. Predicted
0.0e+00
-5.0e+09
-1.0e+10
-1.5e+10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Cores
colour
LM

Bibliography

[1] *AMD Ryzen™ Threadripper™ 3990X Processor*. en. 2018. URL: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x (visited on 10/10/2020).

[2] Joe Armstrong. 'The development of Erlang'. In: *Proceedings of the second ACM SIGPLAN international conference on Functional programming*. 1997, pp. 196–203.

[3] Ken Arnold, James Gosling et al. *The Java programming language*. Vol. 2. Addison-wesley Reading, 2000.

[4] Moshe Bach, Mark Charney et al. 'Analyzing parallel programs with pin'. In: *Computer* 43.3 (2010), pp. 34–41.

[5] D.H. Bailey, E. Barszcz et al. 'The Nas Parallel Benchmarks'. In: *Int. J. High Perform. Comput. Appl.* 5.3 (Sept. 1991), pp. 63–73. ISSN: 1094-3420. DOI: 10.1177/109434209100500306https://doi.org/10.1177/109434209100500306. URL: https://doi.org/10.1177/109434209100500306.

[6] Zorana Banković and Pedro López-Garcıa. 'Energy Efficient Allocation and Scheduling for DVFS-enabled Multicore Environments Using a Multiobjective Evolutionary Algorithm'. In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. Genetic and Evolutionary Computation Conference (GECCO) 2015. Madrid, Spain: ACM, 2015, pp. 1353–1354. ISBN: 978-1-4503-3488-4. DOI: 10.1145/2739482.2764645https://doi.org/10.1145/2739482.2764645. URL: http://doi.acm.org/10.1145/2739482.2764645.

[7] Adam D Barwell, Christopher Brown and Kevin Hammond. 'Finding parallel functional pearls: Automatic parallel recursion scheme detection in Haskell functions via anti-unification'. In: *Future Generation Computer Systems* 79 (2018), pp. 669–686.

[8] Christian Bienia, Sanjeev Kumar et al. 'The PARSEC Benchmark Suite: Characterization and Architectural Implications'. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Oct. 2008.

[9] Armin Biere, Marijn Heule and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[10] Hans-J Boehm. 'The space cost of lazy reference counting'. In: *ACM SIGPLAN Notices* 39.1 (2004), pp. 210–219.

[11] Leo Breiman. 'Random forests'. In: *Machine learning* 45.1 (2001), pp. 5–32.

[12] Christopher Brown, Kevin Hammond et al. 'A Language-Independent Parallel Refactoring Framework'. In: *Proceedings of the Fifth Workshop on Refactoring Tools*. WRT '12. Rapperswil, Switzerland: Association for Computing Machinery, 2012, pp. 54–58. ISBN: 9781450315005. DOI: 10.1145/2328876.2328884https://doi.org/10.1145/2328876.2328884. URL: https://doi.org/10.1145/2328876.2328884.

[13] Christopher Brown, Vladimir Janjic et al. 'Refactoring GrPPI: generic refactoring for generic parallelism in C++'. In: *International Journal of Parallel Programming* 48.4 (2020), pp. 603–625.

[14] Cameron B Browne, Edward Powley et al. 'A survey of monte carlo tree search methods'. In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[15] Claire Burguière and Christine Rochange. 'History-based schemes and implicit path enumeration'. In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2006.

[16] Guillaume Chaslot, Sander Bakkes et al. 'Monte-Carlo Tree Search: A New Framework for Game AI.' In: *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI, 2008.

[17] Jixang Cheng, Gexiang Zhang et al. 'Multi-objective ant colony optimization based on decomposition for bi-objective traveling salesman problems'. In: *Soft Computing* 16.4 (2012), pp. 597–614.

[18] Adriana E. Chis and Horacio González–Vélez. 'Design Patterns and Algorithmic Skeletons: A Brief Concordance'. In: *Modeling and Simulation in HPC and Cloud Systems*. Ed. by Joanna Kołodziej, Florin Pop and Ciprian Dobre.

Cham: Springer International Publishing, 2018, pp. 45–56. ISBN: 978-3-319-73767-6. DOI: 10.1007/978-3-319-73767-6_3https://doi.org/10.1007/978-3-319-73767-6_3. URL: https://doi.org/10.1007/978-3-319-73767-6%5C%5F3.

[19] Shaiful Alam Chowdhury and Abram Hindle. 'GreenOracle: Estimating Software Energy Consumption with Energy Measurement Corpora'. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. The Mining Software Repositories (MSR) 2016. Austin, Texas: ACM, 2016, pp. 49–60. ISBN: 978-1-4503-4186-8. DOI: 10.1145/2901739.2901763https://doi.org/10.1145/2901739.2901763. URL: http://doi.acm.org/10.1145/2901739.2901763.

[20] Chris Clack and Simon L Peyton Jones. 'Strictness analysis—a practical approach'. In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1985, pp. 35–49.

[21] Murray I. Cole. 'Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation'. AAID-85022. PhD thesis. 1988.

[22] *Computer Language Benchmarks Game*. [Online; accessed 2-January-2021]. 2021. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

[23] Luis Cruz, Rui Abreu and Jean-Noël Rouvignac. 'Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring'. In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2017, pp. 205–206. DOI: 10.1109/MOBILESoft.2017.21https://doi.org/10.1109/MOBILESoft.2017.21.

[24] Kent Czechowski. *rapl-tools*. https://github.com/kentcz/rapl-tools. 2014.

[25] Leonardo Dagum and Ramesh Menon. 'OpenMP: an industry standard API for shared-memory programming'. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[26] K. Deb, A. Pratap et al. 'A fast and elitist multiobjective genetic algorithm: NSGA-II'. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017https://doi.org/10.1109/4235.996017.

[27] Mm Rodrıguez Del Águila and AR González-Ramırez. 'Sample size calculation'. In: *Allergologia et immunopathologia* 42.5 (2014), pp. 485–492.

[28] Robert H Dennard, Fritz H Gaensslen et al. 'Design of ion-implanted MOSFET's with very small physical dimensions'. In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 38–50.

[29] Paul Diefenbaugh and Lilly Huang. *Enhancing power delivery with transient running average power limits*. US Patent No. 7484108. 2009. URL: https://patents.google.com/patent/US7484108.

[30] Stephen Diehl. *What I Wish I Knew When Learning Haskell*. 2020.

[31] M. Dorigo, M. Birattari and T. Stutzle. 'Ant colony optimization'. In: *IEEE Computational Intelligence Magazine* 1.4 (2006), pp. 28–39. DOI: 10.1109/MCI.2006.329691https://doi.org/10.1109/MCI.2006.329691.

[32] M. Fattah, M. Daneshtalab et al. 'Smart hill climbing for agile dynamic mapping in many-core systems'. In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–6. DOI: 10.1145/2463209.2488782https://doi.org/10.1145/2463209.2488782.

[33] Ahmed Fawzy Gad. *PyGAD: An Intuitive Genetic Algorithm Python Library*. 2021. DOI: 10.48550/ARXIV.2106.06158https://doi.org/10.48550/ARXIV.2106.06158. URL: https://arxiv.org/abs/2106.06158.

[34] Bernard Van Gastel, Rody Kersten and Marko Van Eekelen. 'Using Dependent Types to Define Energy Augmented Semantics of Programs'. In: *Foundational and Practical Aspects of Resource Analysis*, ed. by Marko Eekelen and Ugo Dal Lago. Cham: Springer International Publishing, 2016, pp. 20–39. ISBN: 978-3-319-46559-3.

[35] Kyriakos Georgiou, Steve Kerrison and Kerstin Eder. 'On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs'. In: *CoRR* abs/1510.07095 (2015). arXiv: 1510.07095https://arxiv.org/abs/1510.07095. URL: http://arxiv.org/abs/1510.07095.

[36] Stefanos Georgiou, Maria Kechagia and Diomidis Spinellis. 'Analyzing Programming Languages' Energy Consumption: An Empirical Study'. In: *Proceedings of the 21st Pan-Hellenic Conference on Informatics*. Pan-Hellenic Conference on Informatics (PCI) 2017. Larissa, Greece: ACM, 2017, 42:1–42:6. ISBN: 978-1-4503-5355-7. DOI: 10.1145/3139367.3139418https://doi.org/10.1145/3139367.3139418. URL: http://doi.acm.org/10.1145/3139367.3139418.

[37] HaskellWiki. *Software transactional memory — HaskellWiki*. [Online; accessed 17-October-2020]. 2013. URL: %5Curl%7Bhttps://wiki.haskell.org/index.php?title=Software%5C%5Ftransactional%5C%5Fmemory%5C&oldid=55691%7D.

[38] John L Hennessy and David A Patterson. 'A new golden age for computer architecture'. In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[39] C. A. R. Hoare. 'Quicksort'. In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10https://doi.org/10.1093/comjnl/5.1.10. eprint: https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf. URL: https://doi.org/10.1093/comjnl/5.1.10.

[40] Arthur E Hoerl and Robert W Kennard. 'Ridge regression: Biased estimation for nonorthogonal problems'. In: *Technometrics* 12.1 (1970), pp. 55–67.

[41] Simon Holmbacka and Jörg Keller. 'Workload Type-Aware Scheduling on big.LITTLE Platforms'. In: *Algorithms and Architectures for Parallel Processing*. Ed. by Shadi Ibrahim, Kim-Kwang Raymond Choo et al. Cham: Springer International Publishing, 2017, pp. 3–17. ISBN: 978-3-319-65482-9.

[42] Roger A Horn, Roger A Horn and Charles R Johnson. *Topics in matrix analysis*. Cambridge university press, 1994.

[43] Paul Hudak, Simon Peyton Jones et al. 'Report on the programming language Haskell: a non-strict, purely functional language version 1.2'. In: *ACM SigPlan notices* 27.5 (1992), pp. 1–164.

[44] American National Standards Institute. *IEEE standard for information technology: Portable Operating System Interface (POSIX) : part 2, shell and utilities*. Two volumes. IEEE Std 1003.2-1992 (includes IEEE Std 1003.2a-1992). Approved September 17, 1992, IEEE Standards Board. Approved April 5, 1993, American National Standards Institute. The primary purpose of this standard is to define a standard interface and environment for application programs that require the services of a 'shell' command language interpreter and a set of common utility programs. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press, Sept. 1993, pp. xvii + 1195. ISBN: 1-55937-255-9.

[45] Intel-Corporation. *Intel 64 and IA-32 Architectures Software Developer Manual: Vol 3*. Vol. 3. Intel. Dec. 2017, pp. 10–31. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html.

[46] *Intel® Pentium® Pro Processor 200 MHz, 1M Cache, 66 MHz FSB Product Specifications*. en. URL: https://ark.intel.com/content/www/us/en/ark/products/49951/intel-pentium-pro-processor-200-mhz-1m-cache-66-mhz-fsb.html (visited on 10/10/2020).

[47] Vladimir Janjic, Christopher Brown et al. 'Refactoring for introducing and tuning parallelism for heterogeneous multicore machines in Erlang'. In: *Concurrency and Computation: Practice and Experience* (2019), e5420.

[48] Wenhao Jia, Kelly A. Shaw and Margaret Martonosi. 'Stargazer: Automated regression-based GPU design space exploration'. In: *2012 IEEE International Symposium on Performance Analysis of Systems Software*. 2012, pp. 2–13. DOI: 10.1109/ISPASS.2012.6189201https://doi.org/10.1109/ISPASS.2012.6189201.

[49] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.

[50] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[51] Steve Kerrison and Kerstin Eder. 'Energy Model - Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor'. In: *ACM Transactions on Embedded Computing Systems* 14.3 (2015), 56:1–56:25. ISSN: 1539-9087. DOI: 10.1145/2700104https://doi.org/10.1145/2700104. URL: http://doi.acm.org/10.1145/2700104.

[52] Steven P Kerrison. 'Energy modelling of multi-threaded, multi-core software for embedded systems'. PhD thesis. University of Bristol, 2015.

[53] C. Kessler, S. Litzinger and J. Keller. 'Robustness and Energy-elasticity of Crown Schedules for Sets of Parallelizable Tasks on Many-core Systems with DVFS'. In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2020, pp. 136–143. DOI: 10.1109/PDP50117.2020.00027https://doi.org/10.1109/PDP50117.2020.00027.

[54] C. Kessler, S. Litzinger and J. Keller. 'Static Scheduling of Moldable Streaming Tasks With Task Fusion for Parallel Systems With DVFS'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 4166–4178. DOI: 10.1109/TCAD.2020.3013054https://doi.org/10.1109/TCAD.2020.3013054.

[55] Alaa Khamis and Yinan Wang. *AI Search Algorithms for Smart Mobility*. 2022.

[56] Abdullah Konak, David W Coit and Alice E Smith. 'Multi-objective optimization using genetic algorithms: A tutorial'. In: *Reliability Engineering & System Safety* 91.9 (2006), pp. 992–1007.

[57] Max Kuhn. 'Building Predictive Models in R Using the caret Package'. In: *Journal of Statistical Software, Articles* 28.5 (2008), pp. 1–26. ISSN: 1548-7660. DOI: 10.18637/jss.v028.i05https://doi.org/10.18637/jss.v028.i05. URL: https://www.jstatsoft.org/v028/i05.

[58] Alexey Kukanov and Michael J Voss. 'The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks.' In: *Intel Technology Journal* 11.4 (2007).

[59] Chris Lattner and Vikram Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation'. In: *CGO*. San Jose, CA, USA, Mar. 2004, pp. 75–88.

[60] Charles L Lawson and Richard J Hanson. *Solving least squares problems*. SIAM, 1995.

[61] Lu Li, Usman Dastgeer and Christoph Kessler. 'Pruning Strategies in Adaptive Off-Line Tuning for Optimized Composition of Components on Heterogeneous Systems'. In: *2014 43rd International Conference on Parallel Processing Workshops*. 2014, pp. 255–264. DOI: 10.1109/ICPPW.2014.42https://doi.org/10.1109/ICPPW.2014.42.

[62] Lu Li and Christoph Kessler. 'MeterPU: a generic measurement abstraction API'. In: *The Journal of Supercomputing* 74.11 (2018), pp. 5643–5658.

[63] L. G. Lima, F. Soares-Neto et al. '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)*. Vol. 1. Suita, Japan: IEEE, Mar. 2016, pp. 517–528. DOI: 10.1109/SANER.2016.85https://doi.org/10.1109/SANER.2016.85.

[64] Aniruddha Marathe, Yijia Zhang et al. 'An Empirical Survey of Performance and Energy Efficiency Variation on Intel Processors'. In: *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*. E2SC'17. Denver, CO, USA: ACM, 2017, 9:1–9:8. ISBN: 978-1-4503-5132-4. DOI: 10.1145/3149412.3149421https://doi.org/10.1145/3149412.3149421. URL: http://doi.acm.org/10.1145/3149412.3149421.

[65] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. Farnham, Surrey, UK: O'Reilly Media, Inc., August 2013. ISBN: 9781449335946.

[66] Simon Marlow and Simon Peyton Jones. 'Multicore garbage collection with local heaps'. In: *ACM SIGPLAN Notices* 46.11 (2011), pp. 21–32.

[67] Simon Marlow and Simon Peyton Jones. 'The Glasgow Haskell Compiler'. In: *The Architecture of Open Source Applications, Volume 2*. The Architecture of Open Source Applications, Volume 2. Lulu, Jan. 2012. URL: https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler/.

[68] Simon Marlow, Simon Peyton Jones and Satnam Singh. 'Runtime Support for Multicore Haskell'. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: Association for Computing Machinery, 2009, pp. 65–78. ISBN: 9781605583327. DOI: 10.1145/1596550.1596563https://doi.org/10.1145/1596550.1596563. URL: https://doi.org/10.1145/1596550.1596563.

[69] Ami Marowka. 'Energy-Aware Modeling of Scaled Heterogeneous Systems'. In: *International Journal of Parallel Programming* 45.5 (Oct. 2017), pp. 1026–1045. ISSN: 0885-7458. DOI: 10.1007/s10766-016-0453-2https://doi.org/10.1007/s10766-016-0453-2. URL: https://doi.org/10.1007/s10766-016-0453-2.

[70] Gilberto Melfe, Alcides Fonseca and João Paulo Fernandes. 'Helping Developers Write Energy Efficient Haskell Through a Data-structure Evaluation'. In: *Proceedings of the 6th International Workshop on Green and Sustainable Software*. Proceedings of the 6th International Workshop on Green and Sustainable Software (GREENS) 2018. Gothenburg, Sweden: ACM, 2018, pp. 9–15. ISBN: 978-1-4503-5732-6. DOI: 10.1145/3194078.3194080https://doi.org/10.1145/3194078.3194080. URL: http://doi.acm.org/10.1145/3194078.3194080.

[71] Arnaldo Carvalho de Melo. 'Performance counters on Linux'. In: *Linux Plumbers Conference*. Vol. 118. 2009.

[72] Olga V Moldovanova and Mikhail G Kurnosov. 'Auto-vectorization of loops on Intel 64 and Intel Xeon Phi: Analysis and evaluation'. In: *International Conference on Parallel Computing Technologies*. Springer. 2017, pp. 143–150.

[73] G. E. Moore. 'Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.' In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35.

[74] Jeremy Morse, Steve Kerrison and Kerstin Eder. 'On the Limitations of Analyzing Worst-Case Dynamic Energy of Processing'. In: *ACM Trans. Embed. Comput. Syst.* 17.3 (Feb. 2018), 59:1–59:22. ISSN: 1539-9087. DOI: 10.1145/3173042https://doi.org/10.1145/3173042. URL: http://doi.acm.org/10.1145/3173042.

[75] Frank Mueller. *Pthreads Library Interface*. Tech. rep. 1999.

[76] Stefania Loredana Nita and Marius Mihailescu. *Practical Concurrent Haskell: With Big Data Applications*. Apress, 2017.

[77] James Pallister. 'Exploring the fundamental differences between compiler optimisations for energy and for performance'. PhD thesis. University of Bristol, 2016.

[78] James Pallister, Steve Kerrison et al. 'Data Dependent Energy Modeling for Worst Case Energy Consumption Analysis'. In: roceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES) 2017 (2017), pp. 51–59. DOI: 10.1145/3078659.3078666https://doi.org/10.1145/3078659.3078666. URL: http://doi.acm.org/10.1145/3078659.3078666.

[79] *Parallel Functional Programming*. URL: %5Curl%7Bhttps://www.cse.chalmers.se/edu/year/2015/course/DAT280%5C%5FParallel%5C%5FFunctional%5C%5FProgramming/given.hs%7D (visited on 28/03/2022).

[80] Will Partain. 'The Nofib Benchmark Suite of Haskell Programs'. In: *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. London, UK, UK: Springer-Verlag, 1993, pp. 195–202. ISBN: 3-540-19820-2. URL: http://dl.acm.org/citation.cfm?id=647557.729913.

[81] Rui Pereira, Marco Couto 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*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. ISBN: 9781450355254. DOI: 10.1145/3136014.3136031https://doi.org/10.1145/3136014.3136031. URL: https://doi.org/10.1145/3136014.3136031.

[82] Simon L Peyton Jones and Jon Salkild. 'The spineless tagless G-machine'. In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, pp. 184–201.

[83] Gudula Rauber Thomasand Rünger. *Parallel Programming Models*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 105–167. ISBN: 978-3-642-37801-0. DOI: 10.1007/978-3-642-37801-0_3https://doi.org/10.1007/978-3-642-37801-0_3. URL: https://doi.org/10.1007/978-3-642-37801-0%5C%5F3.

[84] T. Rauber and G. Rünger. 'Energy-Aware Execution of Fork-Join-Based Task Parallelism'. In: *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2012, pp. 231–240. DOI: 10.1109/MASCOTS.2012.35https://doi.org/10.1109/MASCOTS.2012.35.

[85] Thomas Rauber and Gudula Rünger. *Parallel programming*. Springer, 2013.

[86] *Rosetta Code*. [Online; accessed 1-January-2021]. 2021. URL: https://rosettacode.org/wiki/Rosetta%5C%5FCode.

[87] E. Rotem, A. Naveh et al. 'Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge'. In: *IEEE Micro* 32.2 (Mar. 2012), pp. 20–27. ISSN: 0272-1732. DOI: 10.1109/MM.2012.12https://doi.org/10.1109/MM.2012.12.

[88] Mikko Roth, Arno Luppold and Heiko Falk. 'Measuring and Modeling Energy Consumption of Embedded Systems for Optimizing Compilers'. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. SCOPES '18. Sankt Goar, Germany: Association for Computing Machinery, 2018, pp. 86–89. ISBN: 9781450357807. DOI: 10.1145/3207719.3207729https://doi.org/10.1145/3207719.3207729. URL: https://doi.org/10.1145/3207719.3207729.

[89] Rui Rua, Marco Couto and João Saraiva. 'GreenSource: A Large-Scale Collection of Android Code, Tests and Energy Metrics'. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 176–180. DOI: 10.1109/MSR.2019.00035https://doi.org/10.1109/MSR.2019.00035. URL: https://doi.org/10.1109/MSR.2019.00035.

[90] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597.

[91] Patrick M. Sansom and Simon L. Peyton Jones. 'Generational Garbage Collection for Haskell'. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, 1993, pp. 106–116. ISBN: 089791595X. DOI: 10.1145/165180.165195https://doi.org/10.1145/165180.165195. URL: https://doi.org/10.1145/165180.165195.

[92] Boris Schäling. *The boost C++ libraries*. XML Press, 2014. ISBN: 9781937434366 1937434362.

[93] Marc Schoolderman, Jascha Neutelings et al. 'ECAlogic: Hardware-parametric Energy-consumption Analysis of Algorithms'. In: *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages*. Foundations of Aspect-Oriented Languages (FOAL) 2014. Lugano, Switzerland: ACM, 2014, pp. 19–22. ISBN: 978-1-4503-2798-5. DOI: 10.1145/2588548.2588553https://doi.org/10.1145/2588548.2588553. URL: http://doi.acm.org/10.1145/2588548.2588553.

[94] Richard M Stallman et al. *Using and porting the GNU compiler collection*. Vol. 86. Free Software Foundation, 1999.

[95] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

[96] Emma Strubell, Ananya Ganesh and Andrew McCallum. 'Energy and policy considerations for deep learning in NLP'. In: *arXiv preprint arXiv:1906.02243* (2019).

[97] Ady Tal. *Intel Software Development Emulator*. en. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/software-development-emulator.html (visited on 01/04/2022).

[98] El-Ghazali Talbi. *Metaheuristics: from design to implementation*. Vol. 74. John Wiley & Sons, 2009.

[99] GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.3*. 2002 - 2007. URL: https://downloads.haskell.org/~ghc/7.6.3/docs/html/users%5C%5Fguide/index.html (visited on 30/05/2020).

[100] *The Computer Language Benchmarks Game / benchmarksgame — salsa.debian.org*. https://salsa.debian.org/benchmarksgame-team/benchmarksgame. [Accessed 15-Jan-2022].

[101] *The Gnu Compiler Collection- gnu project - free software foundation (fsf) 2021*. 2021. URL: https://gcc.gnu.org.

[102] Samuli Thomasson. *Haskell High Performance Programming*. Birmingham, UK: Packt, 2016.

[103] Robert Tibshirani. 'Regression shrinkage and selection via the lasso'. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 58.1 (1996), pp. 267–288.

[104] Philip W. Trinder, Kevin Hammond et al. 'Algorithms+ strategy= parallelism'. In: *Journal of functional programming* 8.1 (1998), pp. 23–60.

[105] Immanuel Trummer and Christoph Koch. 'A Fast Randomized Algorithm for Multi-Objective Query Optimization'. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1737–1752. ISBN: 9781450335317. DOI: 10.1145/2882903.2882927https://doi.org/10.1145/2882903.2882927. URL: https://doi.org/10.1145/2882903.2882927.

[106] Guido Van Rossum et al. 'Python programming language.' In: *USENIX annual technical conference*. Vol. 41. 2007, p. 36.

[107] Vincent M. Weaver, Matt Johnson et al. 'Measuring Energy and Power with PAPI'. In: *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*. International Conference on Parallel Processing Workshops (ICPPW) 2012. Washington, DC, USA: IEEE Computer Society, 2012, pp. 262–268. ISBN: 978-0-7695-4795-4. DOI: 10.1109/ICPPW.2012.39https://doi.org/10.1109/ICPPW.2012.39. URL: http://dx.doi.org/10.1109/ICPPW.2012.39.

[108] Thomas Weise. *Global optimization algorithms-theory and application*. 2009.

[109] Bowei Xi, Zhen Liu et al. 'A Smart Hill-Climbing Algorithm for Application Server Configuration'. In: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 287–296. ISBN: 158113844X. DOI: 10.1145/988672.988711https://doi.org/10.1145/988672.988711. URL: https://doi.org/10.1145/988672.988711.

[110] Ali Rıza Yıldız. 'An effective hybrid immune-hill climbing optimization approach for solving design and manufacturing optimization problems in industry'. In: *Journal of Materials Processing Technology* 209.6 (2009), pp. 2773–2780. ISSN: 0924-0136. DOI: https://doi.org/10.1016/j.jmatprotec.2008.06.028https:

//doi.org/https://doi.org/10.1016/j.jmatprotec.2008.06.028. URL: http://www.sciencedirect.com/science/article/pii/S0924013608005268.

[111] Hui Zou and Trevor Hastie. 'Regularization and variable selection via the elastic net'. In: *Journal of the royal statistical society: series B (statistical methodology)* 67.2 (2005), pp. 301–320.