# Development of a Monte-Carlo Particle Transport Code in the Rust Programming Language





Internship Report

06/02/2023 – 04/08/2023

**Author**

Isaïe [redacted], *CEA*

**Supervision**

[redacted], *CEA*

[redacted], *Télécom Paris*

## Disclaimer from the Future

This report has been written during my 2023 internship at the French *Alternative Energies and Atomic Energy Commission* (CEA). Some claims may become outdated with time as **I decided not to tinker with the content of this report before making it public**. The only modifications made are to phrasing, word choices or grammar.

Please also note that the proxy-application we compare our implementation against, Quicksilver, is old and not written using modern C++ features (most notably, templates). This is important to consider when reading Section III.B.3.

A few modifications have been made to the Fastiron repository after this report was written, mostly some cleanup and automatic dependency upgrades. The "state" of the project described in this document might not be 100% accurate.

## Acknowledgments

I would like to thank, first and foremost, my supervisor [redacted] for providing support during the entirety of the internship through his explanations, suggestions and contributions to the project.

I thank my professor, [redacted], who allowed me to write this report entirely in English.

Finally, I extend my thanks to the staff of [redacted], as well as my colleagues for making the office a great place to work in.

# Contents

# I. Introduction – Context & Purpose

## I.A. Preamble

This report is a compilation of all the work done during my end-of-study internship at the *French Alternative Energies and Atomic Energy Commission* (CEA). Its content is roughly divided into three categories:

- **Context of the internship** and prerequisite knowledge, which corresponds to what I learned during the early research phase of the internship.
- **Core of the produced work**, which corresponds to both implementation and the reasoning behind it.
- **Results and their analysis**, a presentation of the performance of the developed program as well as observations related to initial objectives.

Additionally, appendices provide the reader with extra information and more detailed explanations on related (but non-essential) parts of the internship. Interesting code snippets will be directly commented in the report while those in appendices only serve as further illustrations, or to avoid a visit to GitHub. The glossary contains definitions of used acronyms as their meaning may not be given consistently.

The purpose of the internship is to evaluate Rust's capabilities and performance for a specific type of application: Monte-Carlo particle transport algorithm implementations. Due to the specificities of modern supercomputers' architectures, programs need to be refactored to maximize the usage of available computing power through parallelism/concurrency. Writing efficient programs can be strenuous, especially when debugging parallel code. The Rust programming language offers a number of tools and guarantees this kind of program could benefit from.

In order to assess the language potential, I was put in charge of the development of a program that mimics a full-scale Monte-Carlo particle transport simulation code, Fastiron [1]. I worked independently on this task, receiving assistance mainly from my supervisor, [redacted]. [He] provided me with pieces of information and documentation about the type of program that I was writing, as well as suggestions and contributions to the source code.

Fastiron was originally a port of the Quicksilver proxy-app [2], written in C++. This allows for clear comparisons between an existing Monte-Carlo particle transport code and a "Rustified" version making full use of the tools provided by the language. Using a C++ implementation as a starting point also allowed me to gain insights on the *C++-to-Rust* porting process.

Quicksilver being a proxy-app, Fastiron ended up being more polished in some places. Because this is partly due to the strict rules enforced by Rust and its more advanced tools for error handling, this is not something to ignore: Our evaluation of the language obviously includes performance, but also other aspects such as code robustness, program behavior or refactoring difficulty for example.

## I.B. Monte-Carlo Methods

### I.B.1. Definition

*Monte-Carlo methods* [3] is a broad term referring to algorithms using random number sampling to estimate a result. This method relies on the principle of *regression to the mean* and the *law of large numbers* to provide estimation with a certain confidence: With a large enough random sample of a given population, one can expect the sample to behave in the same way as the entire population.

In the context of computer science, Monte-Carlo methods refer to a class of algorithms using random sampling to provide an estimated result. These algorithms always finish, but the reliability and precision of the result depends directly on the sampling volume and quality. An increased sampling volume implies an increased amount of post-processing computations, hence the place of these methods in the HPC context.

As a side note, Las Vegas algorithms are defined as a dual to Monte-Carlo algorithms. They refer to randomized algorithms that can guarantee result correctness, but may not finish in a finite amount of time.

Methods of this kind usually follow the same three-step structure: Random sampling; Processing of the samples; Finalization of the results. First, samples are generated using a PRNG. The samples are then processed by a deterministic algorithm and the results are interpreted by the finalizing code.

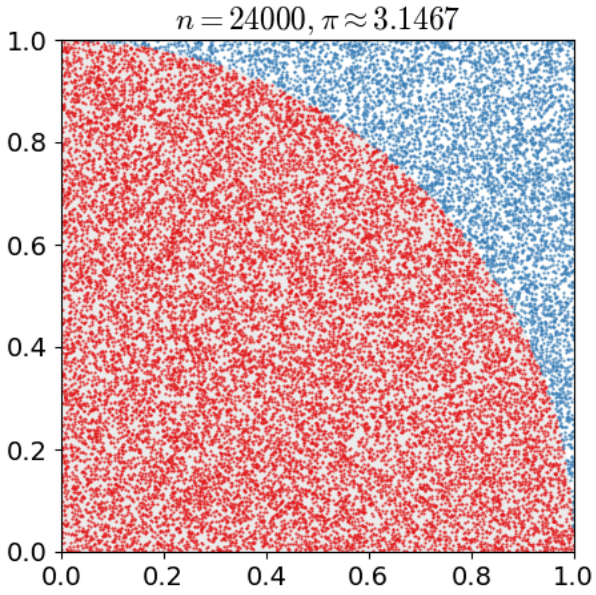



Figure 1: Estimating π with random sampling.

Monte-Carlo methods can be used in many contexts, ranging from applied physics to risk modeling to mathematics. A traditional example of such method is the estimation of π. Using a pseudo-random number generator, one can sample 2-dimension coordinates in $[0; 1]^2$. By considering the area of the quadrant inscribed in the unit square, one can deduce the value of π by counting the number of points inside the quadrant and the total number of point: The quadrant area is directly proportional to π, meaning that the ratio of points (inside the quadrant to total) also is.

### I.B.2. Parallel Strategies for Monte-Carlo Methods

Monte-Carlo methods are considered to be an embarrassingly parallel problem: The processing of a given sample is independent from the processing of another. If the processing algorithm has little to no branching, this allows for trivial SIMD parallelization or GPU targeting.

In the case of the computation of $\pi$, a naive implementation could look like this:

```c
// Sampling...

// Processing & finalizing at the same time
int count = 0;
for sample in sampled_list {
    compute norm of sample
    if (norm <= 1) count++;
}

// Returns approximately pi/4
return (double) count / n_sample
```

For a large amount of samples, this could be quite lengthy. Here is one way of rewriting this loop:

```c
// Sampling...

// Norm computation; Processing phase
double norms[n_sample];
for (int i = 0; i < n_sample; i++) {
    norms[i] = norm(sampled_list[i]);
}

// Count points in the quadrant; Finalizing phase
int count = 0;
for norm in norms {
    if (norm <= 1) count++;
}

// Returns approximately pi/4
return (double) count / n_sample
```

With this decomposition, the aforementioned three phases are clearly separated. The norm computation loop can easily be vectorized or pushed onto the GPU. The second loop can be parallelized by applying a reduction to the norms array. This is possible because all the samples are independent from each other.

### I.B.3. Particle Transport Application

In the context of this internship, the application considered for this method is particle transport. The OpenMC project [4] provides a detailed documentation about theory and methodology that was extremely useful in improving my understanding of Quicksilver's algorithm.

In the context of particle transport, two additional considerations appear for parallel execution:

- Simulation is done over a certain time frame, defined by a time step and a number of steps.

- The processing algorithm, while looking embarrassingly parallel, is not made of fully independent tasks and has a very branchy code path.

The first difference dictates the structure of the program: The classic Monte-Carlo workflow is contained within a temporal loop modeling simulation progress. Additionally, a synchronization phase is required in between processing phases in order to ensure simulation correctness.

The second difference is partly responsible for the parallelization strategies historically adopted for Monte-Carlo simulation. The complexity of the processing algorithm conflicts with the ideal SIMD model mentioned earlier. This led to the usage of a higher-level form of parallelism, notably through MPI, to have multiple physical processors execute the same program. This allows implementation to take advantage of the particles independence in the processing phase without being constrained by the heavy branching. The paradigm shifted away from instruction-level parallelism toward the more flexible SPMD model.

For these reasons, shared memory parallelism was not explored nearly as much as distributed memory parallelism and partitioning. With the advent of GPU-based supercomputers and the progress made in terms of GPU capabilities, shared memory parallelism is once again being considered as indicated in Quicksilver's paper. This is where Rust shows viability, as the different guarantees provided by the language seem to almost trivialize the implementation of shared memory parallelism.

## I.C. The Quicksilver Proxy-Application

### I.C.1. Purpose

Our Monte-Carlo particle transport code used Quicksilver as a starting point. Quicksilver is a proxy application for *Mercury* [2], both projects of the *Lawrence Livermore National Laboratory* (LLNL). The proxy application was developed in order to evaluate the effect of potential refactors of *Mercury* without having to manipulate too large amounts of code.

Using Quicksilver as a baseline for the Rust Monte-Carlo particle transport code allows for a better analysis of the language's capabilities by providing a reference to pit our code against. By comparing programs through benchmarking and source code analysis, a thorough evaluation is possible. It also ensures correctness of the computations of the Rust implementation.

### I.C.2. Structure of the Program

Quicksilver contains 13,000 lines of C++ code, 5,000 being from header files. According to the original associated paper, roughly 4,000 lines are dedicated to initialization code. Its functional core is quite simple: It follows the traditional structure of Monte-Carlo particle transport codes.

```c
/* Initialize data, problem */

/* Main loop */
for (int i = 0; i < n_steps; i++) {
    cycle_init();
    cycle_tracking();
    cycle_finalize();
}

/* Finalize results, free memory */
```

The core functions of the loop are defined as follows in the paper:

```c
cycle_init() {
    source in particles
    population control
}

cycle_tracking() {
    for all particles {
        // one iteration of this 'do' block corresponds to one segment
        do {
            compute distance to census
            compute distance to facet
            compute distance to reaction
            follow segment with shortest distance
            increment tallies
        } until census, absorbed, escaped
    }
}

cycle_finalize() {
    reduce all tallies
}
```

Out of these three sections, only one is subject to parallelization: `cycle_tracking`. There are two reasons for that. First, this section represents the most time consuming part of the program, making up more than 99% of the total execution time. Second, `cycle_init` and `cycle_finalize` require awareness of the full state of the problem for the code to be executed correctly. The former uses it to handle the particle spawning and regulation routines. The latter compiles results and prepares structures for the next time step, requiring the same as the former to ensure data correctness. This puts us in an ideal setup according to Amdahl's law (see Section V.A): The parallelizable part of the code is also the most time consuming.

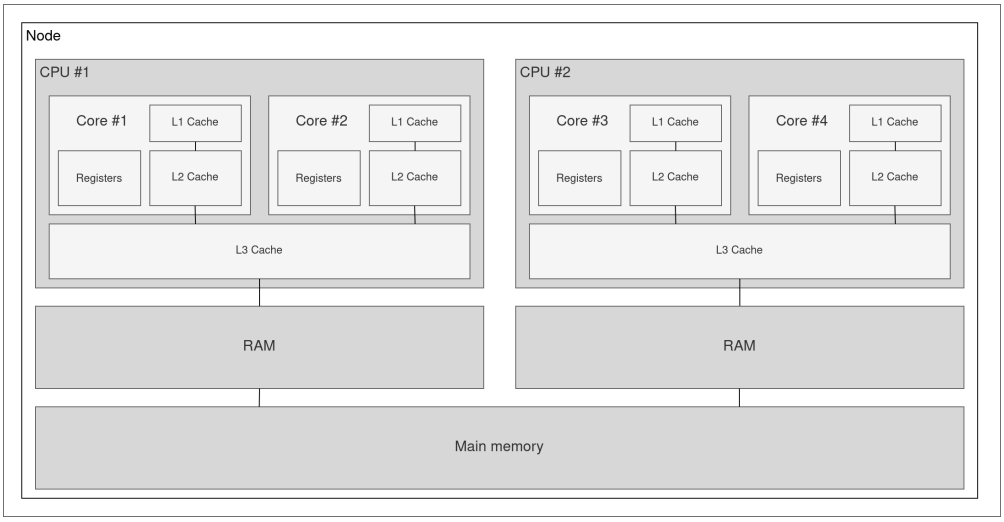



Figure 2: Example of a node's architecture. 2 CPUs per node, 2 cores per CPU.

For both of these reasons, the tracking loop is the focus of changes in implementations. Quicksilver implements parallelism using three technologies:

- OpenMP [5] – to target CPUs (shared memory)
- MPI [6] – to target CPUs across nodes (distributed memory) nodes or NUMA nodes (cf. Figure 2)
- CUDA – to target GPUs

Tweaking the `Makefile` allows us to customize the execution policy of Quicksilver, notably enabling or disabling MPI as well as disabling parallel execution completely. **Fastiron is designed to study shared memory parallelism, making the OpenMP-only Quicksilver our reference in terms of performances**. We also used a sequential version for the development of the first versions of Fastiron.

### I.C.3. Implementation Details

The goal of this part is not to go over all of Quicksilver's source code, but to highlight relevant parts. This will allow me to focus on explaining the changes and not mix in explanation about the original code in later parts.

#### I.C.3.1. Custom timers

Quicksilver uses its own timer structure to record time spent in predefined sections of the program. The sections correspond to the three core functions of the loop as well as subsections of the tracking routine. Additionally, the total run-time of the simulation is recorded. One key usage of these timers is the computation of the **Figure of Merit: The number of segments computed per second**. A segment corresponds to a sub-step a particle takes during processing; It is defined through its distance and associated event.

#### I.C.3.2. Custom allocator

An intermediate allocator is defined in Quicksilver's code as the class `BulkStorage`. The code can be found in the corresponding appendix Section V.B.1. The object is made for array allocation of generic items `T`. The interesting parts are the `setCapacity()` and `getBlock()` methods.

The first allows the object to reserve memory space of a given size for the object to later allocate memory from. The second method acts as the allocator: using the desired size of the array, the `BulkStorage` object internally updates its fields to represent the memory allocation and returns a pointer to the start of the dedicated block. The inner workings mimic the behavior of the stack, using a pointer that is incremented at each allocation.

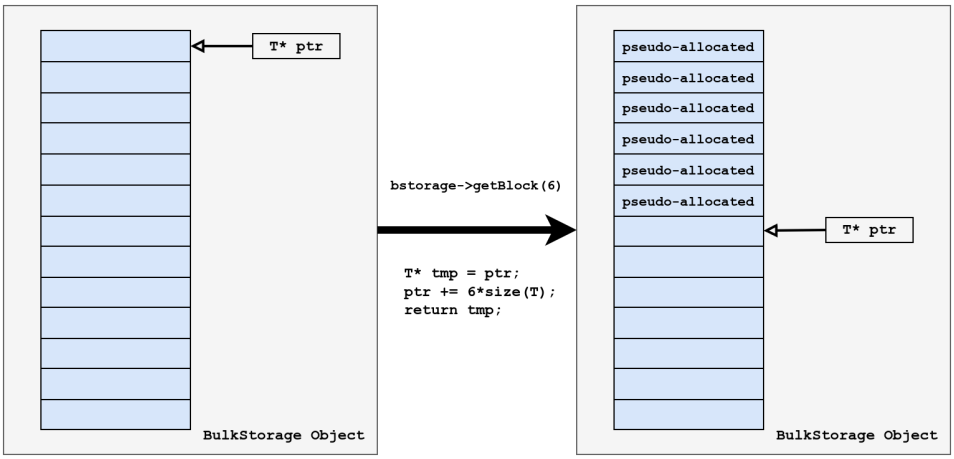



Figure 3: `BulkStorage` structure.

The purpose of this structure is to provide better *locality of reference* by allowing the processing unit to access memory with better *spatial locality*. By reserving a large memory range and making allocation in said range, Quicksilver guarantees that data (e.g. the mesh attributes, see corresponding appendix Section V.B.2) is not scattered randomly in the memory.

### I.C.3.3. Mesh structure

Quicksilver supports a single type of mesh. Its topology is static: The space is divided into hexahedral cells, with each face further subdivided in tetrahedron (cf. Figure 4). In a complete Monte-Carlo particle transport code, the face division would be useful to keep plane facets in the case of mesh deformation.

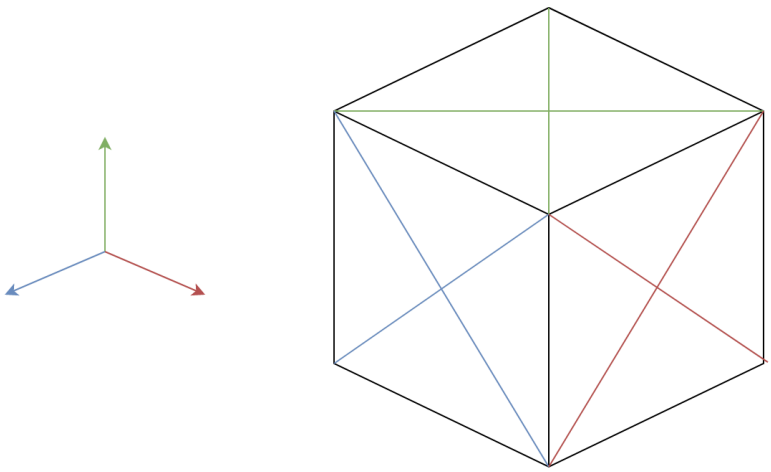


Figure 4: Base geometry of a cell.

While volume recomputation is implemented for cells, the mesh isn't actually modified at runtime, meaning it is not deformed during simulation. One can guess that Mercury implemented some degree of mesh deformation; The feature was probably left out of the proxy-app because of its conflicting nature with the original aim of the project.

### I.C.3.4. Mercury artifacts

The aforementioned “cut-content” from Quicksilver is only one example among others. Because of the exclusion of certain features, part of the code is redundant or completely unused. Among the most notable, we find:

- The `MCBaseParticle` class. One can guess that Mercury supported multiple particle species and used this as a stripped down structure to handle common data such as position, direction, etc.
- The species modeling `enum` as well as other fields of the particle classes. Some are untouched during the simulation while others are updated but never read.
- The `DirectionCosine` class. The model seems overly complicated for its usage, but this might be because of C++, not cut features.

#### I.C.3.5. Outputs & results

Quicksilver provides the user with up to four outputs. Two reports will be printed at run-time and, if file names are specified, two files will be created.

- Data of interest (tallied data) is printed at each time step.
- A timer report is printed at the end of the simulation. Note that the header of the timer report can be misleading.
- The final state of the energy spectrum and the values of the different cross sections are saved if names were specified for output files.

The energy spectrum is usually the main result of interest in Monte-Carlo particle transport code. For my work, the most useful outputs by far were the tallied data and the timer report. I used the former to check the results of my own implementation and ensure no breaking changes were made while the latter was the main data benchmarked along with the Figure of Merit.

# II. Fastiron – A Rust Monte-Carlo Particle Transport Proxy-App

## II.A. Porting to Rust, Naive Implementation

### II.A.1. The Porting Process

As hinted at in the introduction, the methodology I used is the following:

- Code a first one-to-one sequential implementation using Quicksilver as reference. Use it to check consistency and result correctness.
- "Rustify" the code, using results from the previous tested version to ensure no breaking changes are made in the process.
- Make a first batch of optimizations for sequential execution, benchmark, compare…
- Parallelize the Rust code.
- Make a second batch of optimizations, benchmark, compare…

Note that this methodology is close to what you may usually do when parallelizing an existing code or developing one using shared memory parallelism. This is not ideal in C++ because of the debugging phase following the addition of parallelism, however, Rust's guarantees are what make this workflow supposedly viable.

The first challenge was to get familiar with Quicksilver's code in order to avoid making too many algorithmic mistakes in the original rewrite. The OpenMC documentation and Quicksilver's paper were the main elements that allowed me to become familiar with Monte-Carlo methods for particle transport. Having a clear source of information along with the C++ source code was essential as I am not familiar with this domain by training.

Quicksilver's source code was quite easy to read aside from some specific parts, notably the `MCT.cc` file. The biggest issues came from the organization of the code and the unused sections rather than the code itself.

In order to be somewhat efficient when doing the first port, I processed the original code in an organized way. The very first step was to create a program capable of handling inputs and outputs, as this was the only way to test the simulation code as a whole. I then created a functional skeleton of the entire program, using the very useful `todo()!` macro. In retrospect, this was a really good decision as it allowed me to go through the entire source code one additional time before implementing it. From the skeleton, I implemented elements from the bottom up. There are two reasons for that:

1. Implementing from the bottom up allows to make full use of Rust's test system from the get-go, where unit tests are necessary or applicable.
2. Although dummy structures were written while doing the skeleton, having the full code is much more comfortable. Without it, some parts of the code would need to be partially written and completed later when other components were finished.

It was also decided early on to add the possibility to run the simulation using either single or double precision floating-point types. This was possible using Rust's *traits* and the `num` crate [7]. The former allowed me to write the entire code using a generic type implementing a specific trait provided by the latter.

Beside this, the first port is essentially an identical reproduction of Quicksilver in Rust, with minor refactors added to comply with the ownership rules. Those are enforced by the

borrow checker; A static analysis tool –part of the Rust compiler– that enforces borrowing rules. In Rust, each value has a single owner, and this ownership can be *transferred to* or *borrowed by* other parts of the code. The borrow checker ensures that borrows (i.e. *references*) do not outlive the data they point to. This system is what allows Rust to automatically deallocate memory with no need for garbage collection or runtime overhead. The borrow checker also ensures that if a single mutable borrow exists, no other borrows, mutable or not, exist. This second part is the key to Rust's claim of *fearless concurrency*.

The mutability of variables was the subject of many issues and modifications all the way to the end of the development. The problem was rather with its existence than restrictions it led to: Since mutability is the "default state" of variables in C++, I had to either take guesses on a variable's mutability, or rely on the diligence of the original programmers and hope any non-mutated variable was flagged as `const` in function signatures. As for actual constants, many parameters were hard-coded into the program as magic values. I progressively extracted their value and used proper alternatives during the early development.

The `BulkStorage` structure was directly cut out from the port for various reasons, one of them being that writing pointer arithmetic in Rust, while possible, is neither recommended nor coherent with our objectives. Instead, vectors were carefully initialized with the needed capacity. While this prevented memory re-allocation, this did not do any good in terms of locality of reference since allocations of nested vectors are not guaranteed to be contiguous.

### II.A.2. Performance Issues

The naive port, while having coherent results, had poor performance and terrible scaling. Its source code can be browsed in the release commit, tagged `v0.1.0`. Performance issues can be largely attributed to two things: Particle storage and inefficient looping.

The system initially utilized an array of fixed-size vaults for particle storage, expanding this array as needed during the simulation. Handling particles required iterating through these vaults and within each one. Initially, invalid particles were denoted by a magic value, which was later substituted with `Option` enums in Rust. However, this change meant that adding a new particle necessitated searching for a `None` to replace.

From a complexity standpoint, this nested search represents, at worst, $n_{\text{particles per vault}} * n_{\text{vaults}}$ operations. This search occurs each time a particle is processed, hence the worst case complexity:

$$C_{\text{worst}} = O((n_{\text{particles per vault}} * n_{\text{vaults}}) * n_{\text{particles}}) = O({n_{\text{particles}}}^2)$$

In practice, the index of the vault being currently filled can be saved between iterations; This gives the following complexity:

$$C = O((n_{\text{particles per vault}} * 1) * n_{\text{particles}}) = O({n_{\text{particles per vault}}}^2 * n_{\text{vaults}})$$

Our solution, a single-layered collection of particles (not `Option`-wrapped particles!) avoids entirely the search problem since new particles are added to the end of the collection, and invalid ones are filtered out.

One can guess from the introductory explanations that a Monte-Carlo particle transport code would be loop-heavy; They would be right. Besides the main temporal loop, there are many places where the program loops or iterates over data in order to either process it or choose a sample. The original code did this by simply using `for` statements on indices and magic values to break out of loops. Once again, this isn't the best choice in Rust.

Fundamentally, `for` loops and iterators do not yield very disparate performance in Rust. The reason these perform worse when using indices seems to be compile-time optimizations. Aside from having predefined, well optimized methods [8], iterators can bypass Rust's bounds check. When accessing slices or vectors using indexes, Rust inserts bounds check in order to prevent a potential buffer overflow. This is great from a security perspective but introduces runtime overhead in our program. One way of bypassing these –without introducing `unsafe` code– is to iterate over the objects directly instead of accessing them through indexing [9].

While these two problems were not necessarily the most interesting ones, they were the ones that, once corrected, brought Fastiron's sequential performance on par to those of Quicksilver.

## II.B. Profiling & Benchmarking Methodology

### II.B.1. Benchmarks

Fastiron was benchmarked using the same set of problems as Quicksilver. These problems are separated into two categories referred to as *CTS2 benchmarks* and *Coral2 benchmarks*. The CTS2 benchmark was designed in order to evaluate the performance of the code on existing hardware while the Coral2 problems were made for prospective purposes (e.g. testing existing code performance on new architectures).

Due to its size, the CTS2 benchmark was primarily used for sequential profiling. The other two were used alongside the first one for parallel profiling and benchmarking.

### II.B.2. Tools

Multiple methods were used during development in order to evaluate how Fastiron fared against Quicksilver, the project. The built-in timers were really helpful at the start since the two programs' structures were identical. As the project progressed, I had to learn to use full-fledged profiling tools to further analyze and optimize Fastiron. The most used were the following:

- `perf` [10], a Linux command line tool for profiling.
- Flame Graphs [11], a visual representation of a program's stack trace. Note that there is a Cargo subcommand implementation.
- Intel's tools Advisor and VTune from their OneAPI toolchain [12].
- `criterion` [13], a Rust library for benchmarking.
- `heaptrack` [14], a heap memory profiler for Linux

These tools, as well as the built-in timers, were all useful in different ways during the development. Using the timers, `perf` and flamegraphs, it is possible to have a quick overview of the code's hotspots and overall performance. This is what drove most of the optimizations made for sequential execution.

Once parallel computing is added to the equation, more refined tools were better suited. The main reason for that is the difference in the call stack. Multithreading coupled with Rust's closure system makes for a quite messy call stack. This is where Intel tools excelled as their capacity to compile and present the sampled data is quite good, though Advisor works much better on C++ code than Rust.

Criterion was probably underused, compared to its capabilities. It served as an easy way to compare implementations for specific functions or computing routines, notably to test the built-in PRNG of Quicksilver compared to what the `rand` crate can offer.

### II.B.3. Sampled Data & Interpretation

The main reference for performance comparison, for most of the development, was the time spent in the `CycleTracking` section and the *Figure of Merit* (FoM), i.e. the number of computed segments per second. The profiling tools provided insights on bottlenecks by using hardware performance counters. The tallied data also included an event counter. While it wasn't the output of interest, it was useful to assess the coherence of the results between the two programs, as well as to analyze their behavior.

For the latter, I created a small tool capable of providing insights on the characteristics of the program: `fastiron-stats`. Coupled with the `csv` output of the main program, it automatically generates data that can then be plotted for easy interpretation. It supports *correlation study*, *scaling study* as well as a simple *timer comparison*.

The first corresponds to the computation of correlation coefficients between event counters and section timers to evaluate the cost of each event. The second compiles timers data of multiple simulation into an easily plottable `csv` file. The final is a simple comparison between two timer reports. Note that all percentages were computed using the relative change definition [15], using the old value of the metric as reference value.

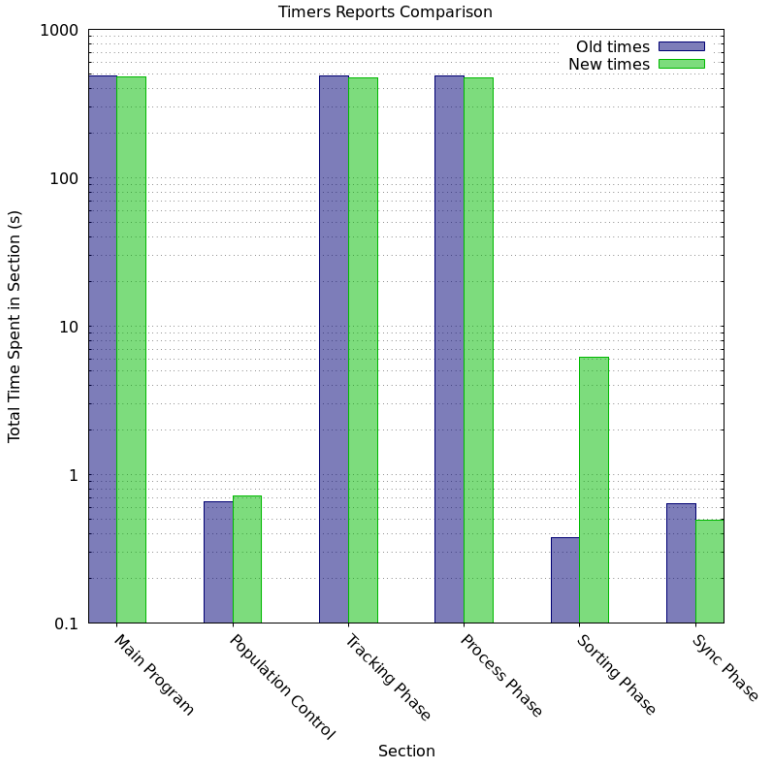



Figure 5: Example of timer comparison – `v1.2.0` to `v1.3.1`.

The correlation study was very useful to cross-reference indications from `perf` and flamegraphs. For example, the heavy refactoring of collision-related routines of the `v1.2.0` release was motivated by the high cost of the event, which was measured and detected by all three tools.

## II.C. Sequential Optimization

### II.C.1. Rustified Release

As mentioned before, the biggest issues of the naive port lay in the particle storage and inefficient looping routines. Those were the main focus of the first batch of optimizations. The file structure was also reorganized into themed modules for a better layout than Quicksilver's flat structure. It can be browsed in the release commit tagged `v1.1.0`.

Particles were accessed through copying by using this function:

```rust
/// Copies a single particle from the particle-vault data and returns it.
pub fn load_particle<T: CustomFloat>(
    particle_vault: &ParticleVault<T>,
    particle_idx: usize,
    ts: T,
) -> Option<MCParticle<T>> {
    if let Some(mut particle) = particle_vault.get_base_particle(particle_idx) {
        // update time to census
        if particle.time_to_census <= zero() {
            particle.time_to_census += ts;
        }

        // set age
        if particle.age < zero() {
            particle.age = zero();
        }

        return Some(MCParticle::new(&particle));
    }
    None
}
```

After being processed, the particle in the vault is overwritten with the processed version. This is highly inefficient:

- Ideally, we would want to operate on mutable references. The copy-overwrite system not only is useless, but creates memory and performance losses.
- Accessing elements of a `Vec` with indices in Rust implies bounds checks. This is easily avoided by iterating over objects instead or writing a few `assert!()` statements. Bounds checks also prevent vectorization; Although it isn't the case here, there are other places in the program that can benefit from it.
- The usage of `Option` leads to index searches and lots of time spent checking values, wrapping, and unwrapping data. It also introduced ambiguities in getter functions since those returned `Option<MCParticle>` objects. On top of that, particles were stored as `MCBaseParticle`, meaning there are two additional type conversions each time a particle is processed.

The `ParticleVault` and `ParticleVaultContainer` classes were scrapped in favor of a simpler, single-layered storage structure called `ParticleContainer`. It stores `MCBaseParticle` objects in a single contiguous vector to facilitate processing.

```rust
pub struct ParticleContainer<T: CustomFloat> {
    /// Container for particles that have yet to be processed.
```

```
3       pub processing_particles: Vec<MCBaseParticle<T>>,
4       /// Container for already processed particles.
5       pub processed_particles: Vec<MCBaseParticle<T>>,
6       /// Container for extra particles. This is used for fission-induced
7       /// particles and incoming off-processor particles.
8       pub extra_particles: Vec<MCBaseParticle<T>>,
9       /// Queue used to save particles and neighbor index for any particles
10      /// moving from a domain managed by a different processor than the current
11      /// one.
12      pub send_queue: SendQueue<T>,
13  }
```

The flat-layered, plain particle storage allows us to more easily add or pop particles from the structure. It also opens possibilities for smarter processing: instead of copying particles, processing them and saving them to the processed vaults, we can iterate on mutable references, and know when processing is done by adding a boolean return value to the right routine. Not only does this yields better performance, the code is also much cleaner (cf. Section V.B.3 appendix).

You can note that "off-processor" particles are mentioned in the documentation of the structure. This was related to the usage of MPI in the original code: The mesh was divided among MPI ranks and explicit communication was needed when a particle traveled to a cell handled by a different rank than the current one. **This was never fully implemented in Fastiron because other options were explored**.

Aside from the storage system, I focused on rewriting the code to fit the more idiomatic style seen in standard Rust programs. This mainly included transforming indexed loops into iterating loops as well as minimizing data cloning/copying. In a simple sequential context, idiomatic and straightforward code seems to yield the best performance in Rust.

The poor storage implementation led to a growth of the time needed to process a single segment. Here are the results of the first iterations of the CTS2 benchmark, without useless columns:

| cycle | num_seg | scalar_flux | cycleInit (s) | cycleTracking (s) | cycleFinalize (s) |
|---|---|---|---|---|---|
| 0 | 3392593 | 1.243407e8 | 8.47633e-1 | 1.4892728e1 | 4.33e-4 |
| 1 | 3865559 | 1.585038e8 | 6.47045e-1 | 2.0176099e1 | 3.42e-4 |
| 2 | 3648944 | 1.564038e8 | 6.25159e-1 | 2.0433194e1 | 3.91e-4 |
| 3 | 3779548 | 1.638133e8 | 5.93636e-1 | 2.5217456e1 | 3.67e-4 |
| 4 | 3782522 | 1.647084e8 | 6.07219e-1 | 2.8406273e1 | 3.65e-4 |
| 5 | 3746309 | 1.590727e8 | 6.08588e-1 | 3.0126254e1 | 3.5e-4 |

This behavior is nothing short of an aberration: The tracking times lengthen while the number of segments computed stays approximately the same. This means that the time needed to process a single segment was growing. Now here is the same data, using the Rustified release:

| cycle | num_seg | scalar_flux | cycleInit (s) | cycleTracking (s) | cycleFinalize (s) |
|---|---|---|---|---|---|
| 0 | 3392593 | 1.243407e8 | 1.336e-2 | 6.10601e0 | 7.100e-8 |
| 1 | 3865559 | 1.585038e8 | 9.636e-3 | 7.19680e0 | 7.000e-8 |
| 2 | 3648944 | 1.564038e8 | 7.172e-3 | 6.71715e0 | 7.100e-8 |
| 3 | 3779548 | 1.638133e8 | 7.898e-3 | 6.96330e0 | 7.200e-8 |
| 4 | 3782522 | 1.647084e8 | 8.088e-3 | 6.95451e0 | 7.000e-8 |
| 5 | 3746309 | 1.590727e8 | 7.586e-3 | 6.88376e0 | 7.400e-8 |

Not only is the tracking time now stable, but its average is twice as fast as the shortest one from the naive version. Note that the drastic change of values in `cycleFinalize` is because of a change in sampling. Quicksilver pauses this timer during printing. This is actually a good thing but the pause is done using the `start` and `stop` functions. The code that processes timers values sees this as two samples, meaning that the plotted value was the mean of the time spent in two different segments. I decided to change the sampling to have only one, at the cost of reducing the timer's relevance.

### II.C.2. RuSeq Release

The focus on this version was to chase performance in the sequential context. This incidentally led to further rustification of the code, as well as the development of a small analysis tool to identify costly routines and to more easily evaluate the influence of potential refactors. The secondary objective was to pave the way for the parallel implementation. The source code of both the main program and the tool can be browsed in the release commit tagged `v1.2.0`.

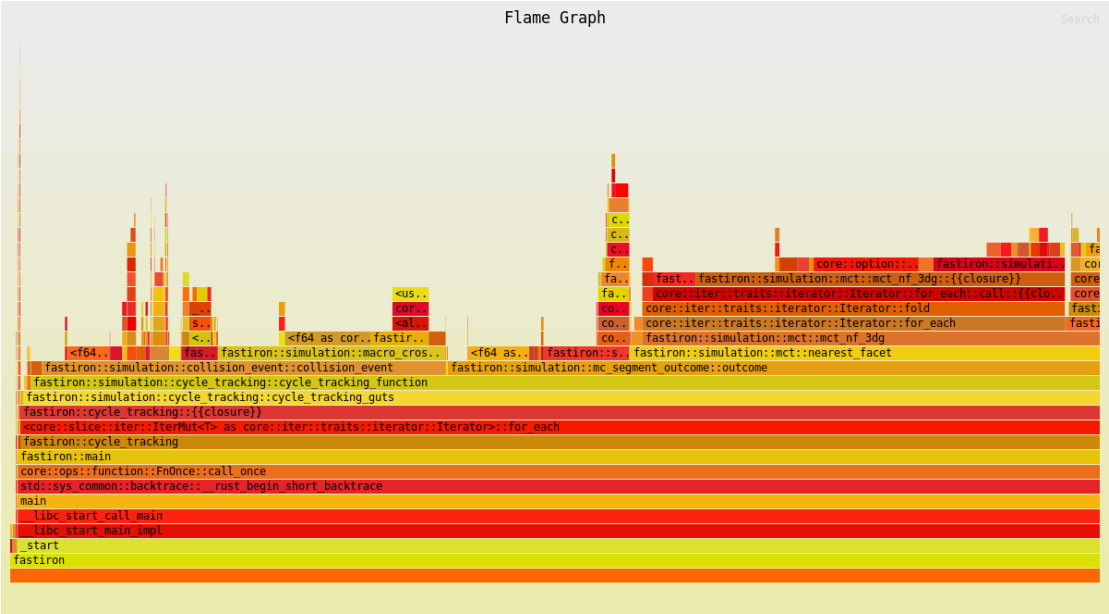



Figure 6: Fastiron `v1.1` flamegraph of a CTS2 benchmark run.

By going through the source code, I identified a number of possible refactors and proceeded with implementation. All changes led to observable improvements when using the tools mentioned before. Most of the mentioned changes here can also be observed on the flame graphs (cf. Figure 6), these flame graphs can be found as `svg` files in the project's repository.

The main performance oriented changes consisted of simplifying the code and avoid using magic values or `Option` by instead catching issues at earliest stage possible. The `MCBaseParticle` type was completely removed as only one type of particle was supported by the simulation. Other unused or never-read fields were deleted. This was an important change as it allowed me to update functions to "minimally borrow" data and sometimes to drop some mutability requirements. The deletion of the `MCBaseParticle` type also removed the need to convert back and forth the particles when processing them, meaning a reduced number of operations and temporary allocations.

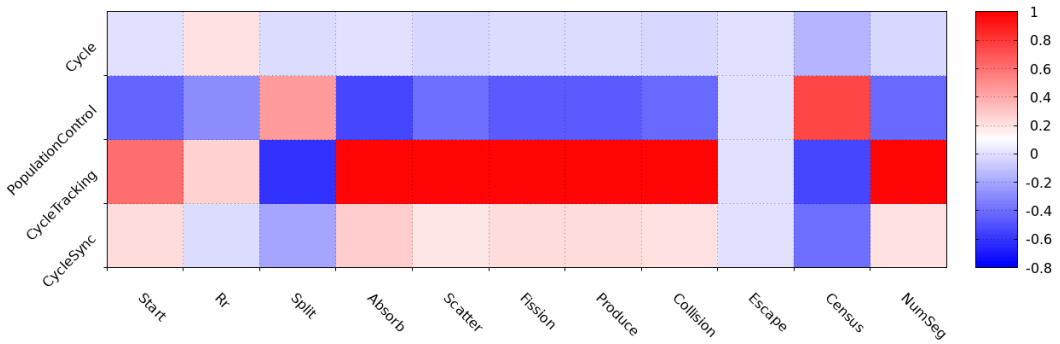



Figure 7: Fastrion `v1.2` correlation matrix of a CT2 benchmark run.

Because of the effect minimizing borrows had on performance, I reworked a lot of routines to be methods of structures instead of standalone functions. This decision was further supported by the results from the statistics tool, particularly from the correlation study. The computed coefficients indicated that the time spent in the tracking phase was very heavily correlated to the number of collision events (cf. Figure 7). The flamegraphs also indicated that a lot of time was spent in the routines called at collision. These routines not only were time-consuming but also had a significant memory footprint. This led me to rethink the collision routine as a method of `MCParticle`, moving away from its standalone form.

The result is a much more efficient function, although collisions are still quite costly compared to other events. Other routines were also reworked to be methods of the particle structure, both small ones and more complex ones. While these may not have significantly improved performance, they improved the code's readability and allowed me to rewrite some parts of the main algorithm more efficiently.

The concept of read-only and read-write data was central to preparations for the parallel implementation. Quicksilver used a single `MonteCarlo` super-structure that was passed around using either a pointer or a reference. This approach is suboptimal in Rust and often directly conflicts with the ownership system. In order to be more efficient, I identified which data was an invariant, and which was subject to alteration at any point of the simulation. This led to the breakdown of the `MonteCarlo` super-structure into a read-only one (`MonteCarloData`) and mutable units (`MonteCarloUnit`). Reducing the amount of nested structures was key to limit mutable borrows to the bare minimum.

All of this led to significant improvements in performance. Using the CTS2 benchmark, the *FoM* increased by 40.5%, going from *5.431e5* to *7.630e5* segments per second. This correlates with the decrease of the tracking time by 28.8%, going from a total of *688.2* seconds down to *489.8*. This particular set of changes showed that, in Rust, writing more idiomatic code tends to yield better performances.

### II.C.3. Sequential Behavior

#### II.C.3.1. Event Cost

The sequential tracking performance seem to be heavily linked to the segment computations and collision reactions. As seen in Figure 8, reaction-specific coefficients are generally consistent. This could indicate that the differences in cost of reaction-specific code are negligible compared to more significant variables.

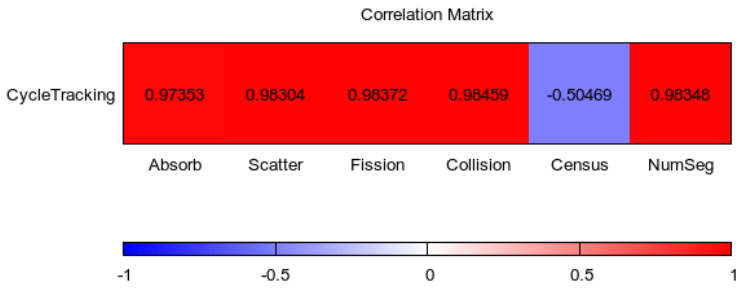



Figure 8: Events of interest.

This is further supported by the data shown on the flame graphs: More than 70% of the time spent in `cycle_tracking()` is spent computing the segment outcome (cf. Figure 9). On the other hand, the `Census` coefficient is negative, meaning that the time spent in the tracking section scales negatively with the number of particles reaching census. This makes sense because once a particle reaches census, no further segments are computed for it, thus reducing the remaining time to be spent in the tracking phase.


Figure 9: Fastiron `v1.2` flamegraph of a CTS2 benchmark run.

### II.C.3.2. Floating Type Precision

From version `1.2.0` and onward, Fastiron supports execution using the `f32` type through the usage of conditional compilation. The program can be compiled with the `single-precision` feature to produce a binary using the `f32` primitive type instead of `f64` for computations.

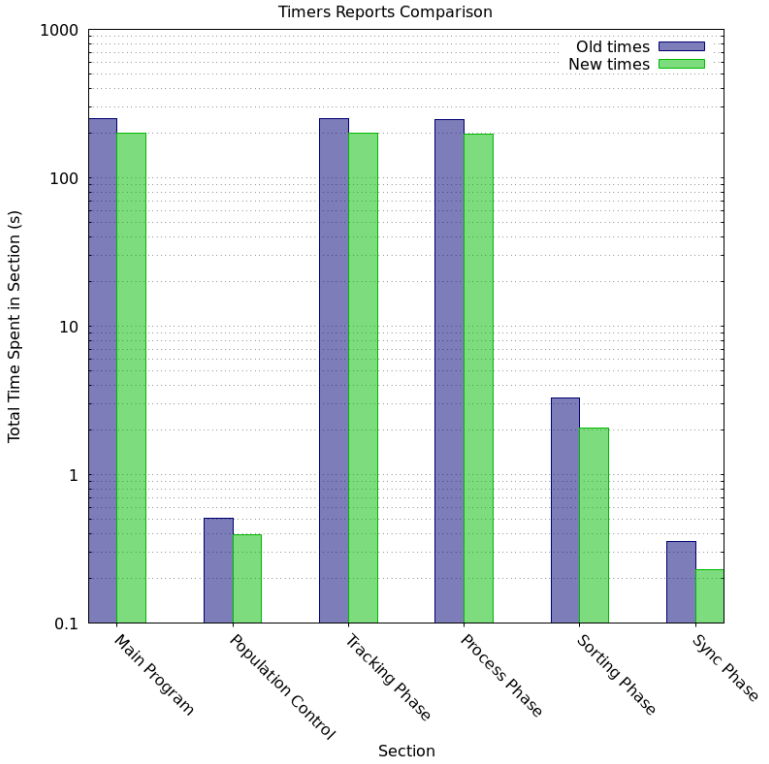



Figure 10: Measured total section times for `f64` (Blue) & `f32` (Green)

As shown in Figure 10, there are significant performance differences in both sorting and synchronization sections. The former does not contain any computations using floating-point types. This leads me to believe that the change observed is due to the overall size of the data the program is handling: The algorithm used by the `sort_by()` method is a modified merge sort, making temporary allocations to achieve its result. The allocation size could influence the time it takes for the method to execute: Smaller allocations might fit within an existing memory page, whereas larger allocations would necessitate a new page.

The synchronization phase is quite different: it is computation-heavy and uses plenty of decimal variables. The explanation here is much less convoluted: The algorithm is (very likely) memory-bound; using `f32` instead of `f64` slightly relaxes the bottleneck, yielding better performance in the section.

Precision of the results wasn't directly tested because, as a mini-app, there were no established standards to benchmark against. However, executing the program using double-precision type seems to accelerate the convergence rate. This can be verified with the built-in post-benchmark tests; These are defined in the `coral_benchmark_correctness` module. They check for simulation correctness by using tallied events as well as scalar flux homogeneity. The `f64` version of Fastiron tends to pass these tests at an earlier time step than the `f32` version, hence the faster convergence rate.

## II.D. Parallel Implementation

### II.D.1. The `rayon` Crate

Fastiron aims to study shared memory parallelism. While the standard library contains tools to implement parallelism explicitly, there is a crate that provides us with higher-level abstractions to introduce shared memory parallelism to existing code. The `rayon` crate [16] is

a data-parallelism library designed to be used on top of existing Rust code. Here is a simple example of its usage from the `rayon` documentation:

```rust
use rayon::prelude::*;

let v: Vec<_> = (0..100).collect();
assert_eq!(v.par_iter().sum::<i32>(), 100 * 99 / 2);

assert!(v.par_iter().zip(&v)
         .all(|(a, b)| std::ptr::eq(a, b)));
```

Parallel iterators can be created by calling the `par_iter()` method instead of the usual `iter()` method. While this requires the collection to implement the `IntoParallelRefIterator` or `IntoParallelIterator` traits, rayon also provides the `par_bridge()` method to convert any existing iterator into a parallel one. This is particularly useful when iterating over custom structures as the aforementioned traits most likely aren't implemented at first. This allows the developer to focus on making the parallelization work instead of writing boilerplate code.

### II.D.2. Introducing Parallelism to Existing Code

As stated before, `rayon` makes it quite easy to parallelize existing code. This partly justifies the methodology used for the development of Fastiron. However, what makes it really viable are the ownership system and the borrow checker, allowing `rayon` to guarantee data-race-free executions.

When an existing iterator is parallelized, potential race conditions will be identified at compile time, preventing the creation of an executable with undefined behavior. This is possible thanks to the strictly enforced borrowing rules; The existence of a single mutable borrows prevents the existence of other borrows, this allows the compiler to highlight potential synchronization hazards.

The data, or code, that does not respect borrowing rules is outlined by compiler errors, allowing us to make the necessary changes without having to search for the problem. Said changes usually consist of introducing synchronized types into the code, such as atomics or locks. In Rust:

- Atomics remove the need for mutability completely, as the function we use to access the data takes a non-mutable reference to `self`, even for editing functions.
- Mutexes are used through the `Arc<Mutex<&mut T>>` pattern. What is passed to each thread is a shared reference to a mutex containing the mutable reference. The shared reference itself is non-mutable and correct access to the mutable reference is granted by the mutex mechanism.

These were the first methods used in order to get the program to compile correctly. Thanks to Rust's guarantees, all results obtained in a sequential context were reproduced in parallel as soon as the program compiled, though performance was not necessarily great. **While this may sound trivial, this is where most time-consuming problems would start to occur if we did not have Rust's guarantees**.

### II.D.3. Optimizing Parallel Code

Using tools from the Intel toolchain, it was possible to evaluate the problems of my parallel implementation. Thanks to results from these tools as well as suggestions from [redacted], two main issues were identified:

- Resource contention on atomics and mutexes. This resulted in very poor scaling.
- Poorly designed memory access patterns. The effects were lesser than effects from contention, but these still added some latency to the processing phase.

Both of these issues were subtly linked to the usage of the `par_bridge` method, though not exclusively in the case of memory accesses. In order to do its work, `rayon` uses *work stealing*. Given a pool of worker threads and a certain amount of tasks, the tasks are first dispatched in queues associated to threads. If a thread reaches the end of its queue, it can steal work from the queue of another thread that is not finished: Essentially, the workload is dynamically managed at run-time to minimize idle time for each thread.

While this setup is great when it comes to load balancing, it creates a number of issues in our case. The way particles are split between threads is inconsistent, leading to random memory lookups with no coherence in memory access patterns; For example, two particles belonging to the same cell may be processed by different threads. Additionally, all threads try to access the same shared variables. While this is fine when using 2, 4 or 8 threads, it becomes the main bottleneck at a larger scale. This can be guessed just by looking at the source code:

```rust
fn cycle_tracking_function<T: CustomFloat>(
    // ...
) {
    let mut keep_tracking: bool;

    loop {
        // compute event for segment
        let segment_outcome = outcome(mcdata, mcunit, particle);
        // update # of segments
        mcunit.tallies.balance_cycle.num_segments += 1; // contention #2

        // ...
    }
}

pub fn outcome<T: CustomFloat>(
    // ...
) -> MCSegmentOutcome {
    // ...
    // contention #1
    let precomputed_cross_section = mcunit.domain[particle.domain]
                                    .cell_state[particle.cell]
                                    .total[particle.energy_group];
    let macroscopic_total_xsection = if precomputed_cross_section > zero() {
        // XS was already cached
        precomputed_cross_section
    } else {
        // compute XS
        let tmp = weighted_macroscopic_cross_section(
            // ...
        );
        // cache the XS
        mcunit.domain[particle.domain]
          .cell_state[particle.cell]
          .total[particle.energy_group] = tmp;
```

```
36
37          tmp
38      };
39  }
```

Right at the beginning of the tracking routine, two tasks require access to shared variables. These are done consistently as they are part of the common, non-branching path of the tracking routine. The first is the update of the number of segments computed, the second is the lazy computation of the cross section of the particle. There is a single segment counter and the precomputed cross sections are shared between threads, hence the contention. Because there is little to no branching before these computations, the issue is exacerbated to an extreme as all threads try to access these variables roughly at the same time.

Furthermore, because of the inconsistent work dispatch, there was no guarantee that two particle in the same cell, of identical energy level, would end up being processed by the same thread. Not only is it inefficient from a caching perspective, it actually creates even more contention by dispatching tasks using the same synchronized data on different threads.

All of this led to the re-implementation of `rayon`'s parallel iterators traits for a custom collection type, `ParticleCollection`. This allows us to create chunks that are dispatched onto worker threads, as opposed to individual particles being dispatched. By coupling chunks with a well-ordered particle list, it was possible to reach better performances in terms of memory accesses, and cache-related statistics. The usage of chunks also allowed me to use thread-local structures to limit contention on shared data. Here is the new, abridged processing routine:

```rust
pub fn process_particles(
        &mut self, // ParticleContainer
        mcdata: &MonteCarloData<T>,
        mcunit: &mut MonteCarloUnit<T>,
    ) {
        // ...
        // Create the reference to the shared object
        let extra = Arc::new(Mutex::new(&mut self.extra_particles));

        let res: Balance = self
            .processing_particles
            .par_iter_mut()
            .chunks(chunk_size)
            .map(|mut particles| {
                // Local event counter & extra collection
                let mut loc_balance = Balance::default();
                let mut loc_extra = ParticleCollection::with_capacity(chunk_size*5);
                // Process chunk sequentially
                particles.iter_mut().for_each(|particle| {
                    cycle_tracking_guts(
                        mcdata,
                        mcunit,
                        particle,
                        &mut loc_balance,
                        &mut loc_extra,
                    )
                });
                // Add local extras to shared collection
```

```
                extra.lock().unwrap().append(&mut loc_extra);
                // Return local balance for further processing
                loc_balance
            })
            .fold_with(Balance::default(), |a, b| a + b)
            .sum::<Balance>();
        mcunit.tallies.balance_cycle.add_to_self(&res);
        // ...
    }
```

By using a local event counter and extra storage:

- There is no need for synchronized types in the event counter as one is used per context. This comes at the cost of initializing the structure and summing all at the end. This is not really a problem as the counters are stored in a static array, making the folding operations very efficient.
- The mutex wrapping the extra particle storage is locked only once per chunk as opposed to once per fission event. One could imagine that it is locked for longer, so it "evens out". This might have been the case if the difference between the total number of locks was less drastic. Using a run of the CTS2 benchmark as reference, this change corresponds to dividing the total number of lock/unlock operations by roughly $10^5$. Even if the mutex is locked for a bitlonger, the cost of locking and unlocking the mechanism that many more times creates the observed performance gap.

The system for lazily-computed cross-section still operates using an organized vector of atomics as the explored options did not yield better performance. Among them were duplication of the table (similarly to the event counter) or usage of hashmaps for example. The only actual change was made in conjunction with another major one in the mesh data structure. The domain division was removed in order to limit the number of nested vectors in the data structure, i.e. the number of indirections.

Essentially, all nested vectors were replaced by single-layered ones, using implementations of the `Index` and `IndexMut` traits to compute indices on the fly. This led to better performances by improving locality of reference and reducing latency induced by indirections. This was coupled to a sorting routine that reorganized the particle collection in order to group particle of the same cell together. On smaller problems, this was beneficial as the sorting time was negligible before performance gains thanks to locality of reference and better cache efficiency.

# III. Results – Benchmarking & Analysis

## III.A. Performance Comparison

### III.A.1. Sequential Execution

We use the smallest problem size of the CTS2 benchmark to measure performance at sequential execution Simulation was run on a laptop, using specific settings to have more consistent numbers from one run to another. The following performances were recorded:

| Figure | Quicksilver | Fastiron v1.1 | Fastiron v1.2 | Fastiron v1.3.1 |
|---|---|---|---|---|
| *FoM* (segments per second, *HiB*) | *4.830e5* | *5.431e5* | *7.630e5* | *7.833e5* |
| Total tracking time (seconds, *LiB*) | *7.738e2* | *6.881e2* | *4.898e2* | *4.771e2* |

We can see that from version 1.1 and onward, performance has been consistently higher from one version to the next. Because the focus of each version was different, each refactor gave specific insights about the language:

- **Quicksilver – Fastiron v1.1**: At this point, Fastiron had little to no optimization besides the new particle storage. This means that the additional time spent coding in Rust –to fit stricter compiler rules– led to slightly better performance.
- **Fastiron v1.1 – Fastiron v1.2**: All the changes made for the `v1.2` release were focused on writing more idiomatic Rust code. This means using iterators for most loops, changing functions to use minimal borrows or splitting complex structures into lighter ones for example. Overall, making the code simpler, more straightforward. This resulted in a huge performance gap as the *FoM* increased by **40.5%**. In a sequential context, writing idiomatic Rust code led to better performance.
- **Fastiron v1.2 – Fastiron v1.3.1**: Whether it is `v1.3` or `v1.3.1`, the changes made were all focused on parallel execution; As a matter of fact, there were no recorded numbers on a laptop for this version prior to me writing this paragraph. It is interesting to see that the small changes made for better parallel performance also slightly affected sequential performance, though it is too late to have a precise breakdown of their effects.

As of `v1.3.1`, Fastiron achieves consistently twice the performance of OpenMP-only Quicksilver, in both sequential and parallel contexts: The execution time is halved, i.e. the number of segments computed per second doubles.

### III.A.2. Parallel Execution

#### III.A.2.1. Quicksilver Execution Modes

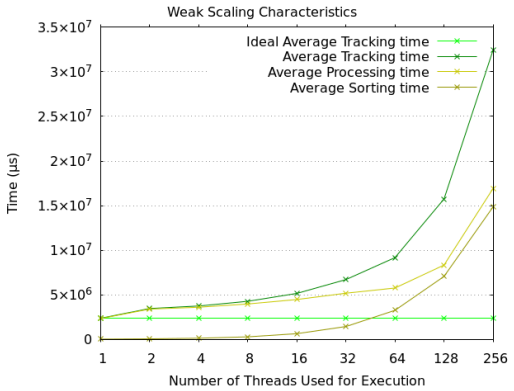
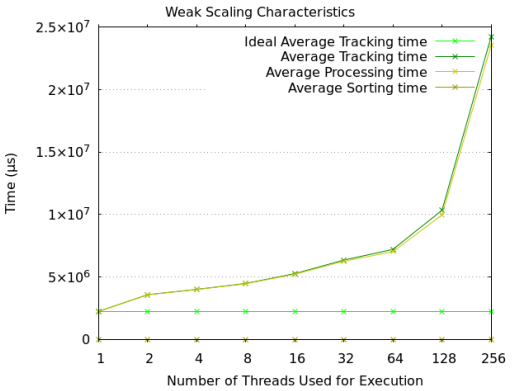
As stated in the introduction, Quicksilver uses multiple parallelism tools in its full implementation. It mixes OpenMP with MPI to achieve a hybrid execution mode. We can compare the performance of Fastiron to two versions of Quicksilver: The hybrid one, and the OpenMP-only one.

Note that the most comparable implementation is the OpenMP-only one, which is achieved by running the program on a single MPI rank, not by using the "dummy" MPI implementation Quicksilver provides. This is due to the way the mesh is handled in the dummy implementation: The mesh is split into four domains whereas all cells are in a single domain

if the program runs on a single MPI rank. In the latter case, there are fewer nested vectors in complex data structures, which is closer to how Fastiron operates. Unless stated otherwise, OpenMP-only Quicksilver refers to the version executed on a single MPI rank.

### III.A.2.2. Particle Sort Tradeoff

In a shared memory context, Fastiron consistently outperforms Quicksilver. We can however observe on the first of the next two figures that the time spent sorting particles grows rapidly starting from the 32-thread-sized problem.

| With particle sort | Without particle sort |
|---|---|
|  |  |



This led me to run all benchmarks once more, this time without particle reordering. In larger problems, the cost of the sorting routine largely overshadows the time gained from reduced memory latency, the initial problem it helped mitigate. With no particle reordering, the performance gap between Quicksilver and Fastiron widened (cf. Figure ).

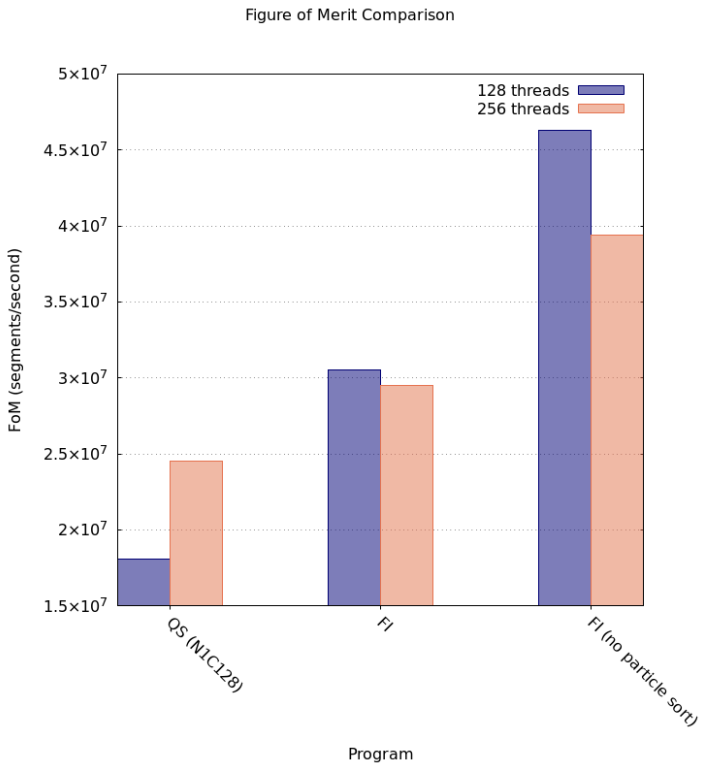



Figure 11: Figure of Merit for the CTS2 benchmark (shared memory).

Figure 11 and Figure 12 show a very interesting difference in behavior between the two programs. The simulations from which the data was measured were executed on a system with 128 physical cores, meaning the 256 threads runs actually involves hyperthreading, a technique based on the execution of two virtual cores –or threads– on a single physical core. The figures show that Quicksilver highly benefits from hyperthreading while Fastiron does not at all; The latter actually performs worse.

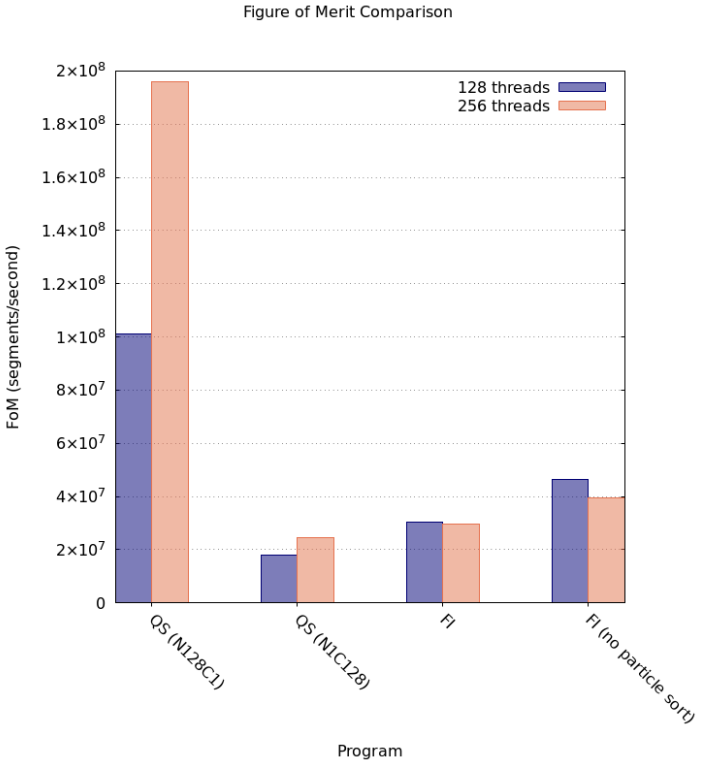



Figure 12: Figure of Merit for the CTS2 benchmark (all).

Whether a program benefits or not from hyperthreading and how much it does both are complex questions. Better performances are achieved through hyperthreading by executing instructions from one thread while the other is stalling, or vice versa. While each thread's individual execution time is lengthen, the overall time is shortened thanks to the overlapping of inactive and active phases of each.

### III.A.2.3. Scaling

In the context of HPC, it is common to analyze the behavior of a program through scaling studies. There are two types of scaling defined: *strong* and *weak*.

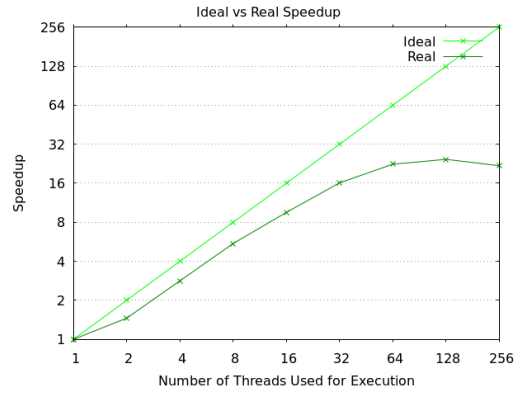
For both scaling type, the number of threads increases while the problem size:

- remains constant in the case of strong scaling, effectively reducing the workload per unit.
- scales accordingly in the case of weak scaling, resulting in a constant workload per processing unit.

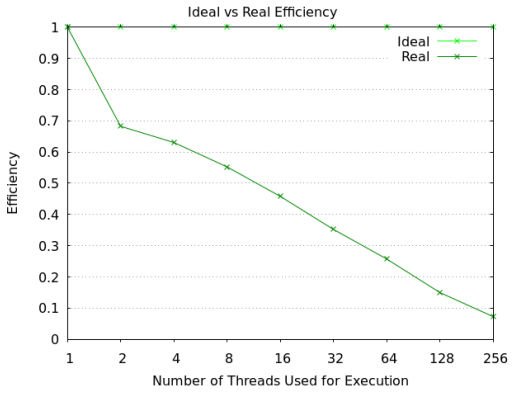
We define respectively to these the *speedup* $S$ and the *efficiency* $E$ as $\frac{t(1)}{t(N)}$, with $t(1)$ and $t(N)$ the processing time using 1 and $N$ units respectively. The difference in nomenclature is linked to the ideal case of each scaling type. In the case of strong scaling, Amdahl's Law [17] tells us that the ideal speedup of a highly parallelizable code like ours should be proportional to the number of processors used (cf. Section V.A appendix). On the other hand, the efficiency defined for weak scaling studies is expected to stay close to one in ideal conditions since the workload per processing unit is constant.

What follows are the measured speedups and efficiencies of the cycle tracking section for Fastiron `v1.3.1`. Note that the ideal case has been chosen to fit the embarrassingly parallel nature of the initial algorithm, not determined through a code analysis.

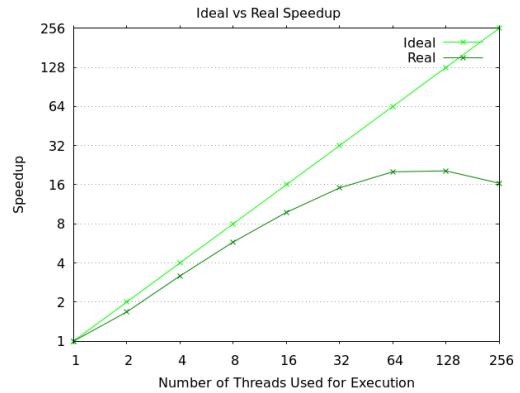
**CTS2 Speedup**



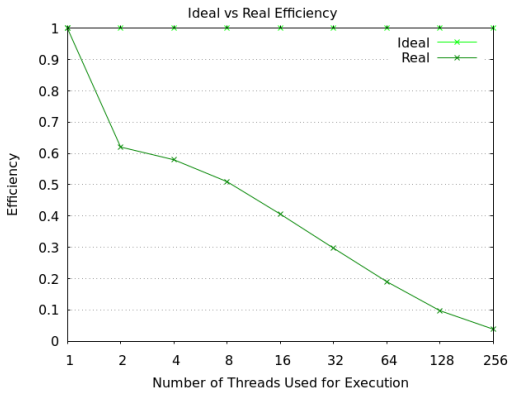

**CTS2 Efficiency**



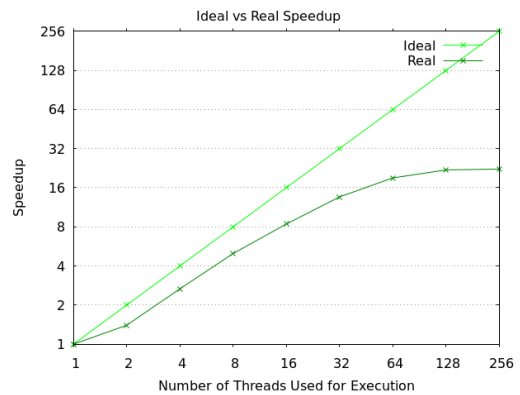

**Coral2P1 Speedup**



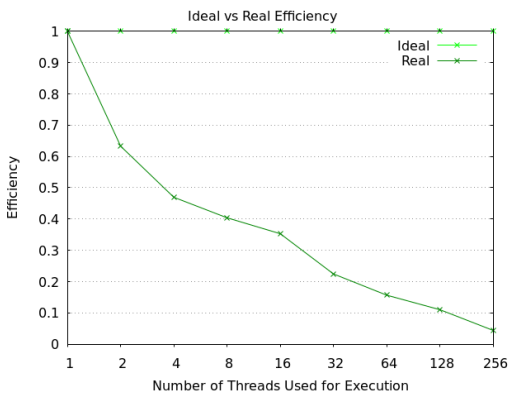

**Coral2P1 Efficiency**




**Coral2P2 Speedup**




**Coral2P2 Efficiency**




First, one can notice that the behavior of the program does not change drastically between each benchmark type. As for the results, we can use reports provided by the LLNL [18], [19] to try and explain the measured performances. Quicksilver is described as follows in both documents:

*"Performance of Quicksilver is likely to be dominated by latency bound table look-ups, a highly branchy/divergent code path, and poor vectorization potential."*

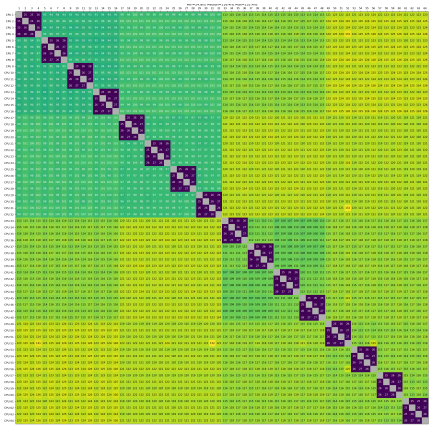
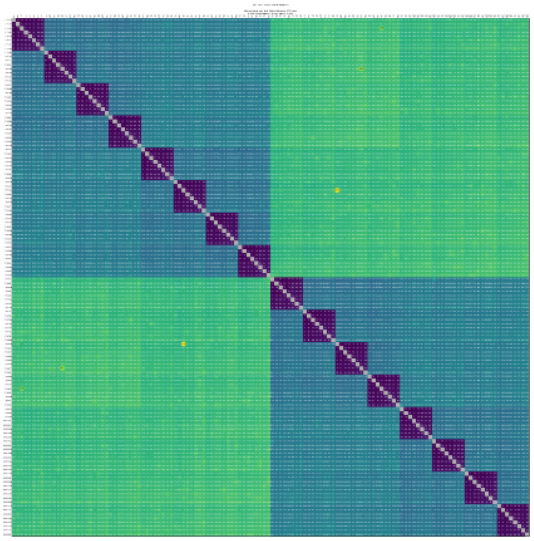
The weak scaling performances seem particularly poor, which is coherent with the fact that the program stresses memory latency and data look-ups: Shared memory parallelism, while helping for computation heavy tasks, does not affect memory access behavior. It also introduces resource contention in our case.

This is further supported by performances measured using the different versions of Quicksilver: The hybrid implementation achieves much better performance by splitting the mesh –i.e. the looked-up data– on several nodes (cf. Figure 12). It allows the different processing units to work with a more restricted, coherent data set. This translates directly to a more efficient cache usage, hence much better performance in the case of a code *dominated by latency bound table look-ups*.

Additionally, distributed parallelism programming techniques allow the developer to better accommodate Non Uniform Memory Access (NUMA) designs. In NUMA systems, each processor socket has its own local memory, and accessing local memory is faster than accessing remote memory on another socket. This leads to non-uniform memory access times and potential bottlenecks when non-local memory accesses multiply. Distributed parallelism allows the developer to have better data locality and implement NUMA-aware scheduling policies, therefore reducing memory-induced latencies.

In order to explain our scaling performance, we can also look at the hardware used for execution. In our case, the program was benchmarked on a single node of a computer cluster. This node holds two sockets, each with an AMD EPYC 7H12. This model of processor possesses 64 CPU cores and 128 threads. For massively parallel units such as the one we used, there is an additional factor to take into account when evaluating strong scalability: core-to-core latency.

We can evaluate it at the scale of a single node by using `core-to-core-latency` [20]:

| **Single CPU latency** | **Node latency (two CPUs)** |
|---|---|
|  |  |
| Lowest: *24.8ns* | Lowest: *24.4ns* |
| Average: *116.4ns* | Average: *196.3ns* |
| Highest: *132.4ns* | Highest: *212.9ns* |

The program produces a `csv` output that can be plotted as a heatmap using provided python code. The heatmaps can also be visualized in Section V.C. Columns and lines correspond

to cores ID. Coefficients are measured latencies. Dark blue corresponds to lowest latencies, yellow to highest.

We can observe that cores work really well in groups of 4 subsequent units, and latency is significantly higher between cores of different groups. Some groups of a larger scale can also be observed. On a single CPU, there is a significant difference between latencies inside the first 32 cores chunk and the rest. The second half of the cores seems slower to synchronize, even in between those. This could explain why the gap between ideal and measured speedup starts to widen at the 64 threads mark. This is further supported by the results of the latency sampled on the entire node: delays are significantly higher when cores are not on the same socket (i.e. CPU). This is reflected in the results as the gap between ideal and measured speedup significantly widens past the 64 threads threshold. The same behavior is observable on the results yielded by OpenMP-only Quicksilver.

Note that the latency measures done on a full node using `core-to-core-latency` yields incorrect results; the two nodes do not behave symmetrically. Our figure was obtained using a separate program.

### III.A.3. Memory Footprint

Using the same setup as for the sequential performance benchmarking, we can pass Fastiron and Quicksilver through `heaptrack` and `perf` in order to evaluate the memory footprint of both programs. The former gives us a nice overview of allocations and memory leaks while the latter can be used to fetch cache-related statistics and hardware counters.

Here is the summarized data from `heaptrack`:

| Program | Quicksilver | Fastiron | Fastiron (no particle sort) |
|---|---|---|---|
| Allocated Memory Peak (overall) | 39.3 Mo | 72.4 Mo | 68.6 Mo |
| Allocated Memory Peak (main loop) | 1.15 Mo | 3.8 Mo | 40 o |
| Leaked Memory | 1 Ko | 660 o | 660 o |
| Allocations | 187,582 | 15,244 | 10,124 |
| Allocations per second | 235 | 31 | 21 |
| Allocations (tmp) | 147,324 | 10,976 | 8,416 |
| Allocations (tmp, %) | 78.54% | 72% | 83.13% |

Overall memory usage differs mostly during initialization. Quicksilver is able to do it more efficiently thanks to the flexibility of C++; In Fastiron, some values need to be duplicated because of the limitations on mutable borrows. The main loop uses very little additional memory, for both programs. In the case of Fastiron, most of the memory used comes from the particle sort: the algorithm used by the `sort_by()` method is a modified merge sort, a method that uses a divide-and-conquer approach –hence the allocations–.

Fastiron makes approximately 10 times fewer calls to the memory allocator. This could be explained by the carefully chosen capacities Fastiron uses to initialize its vectors and others dynamically allocated structures. While Quicksilver allocates and deallocates particle vaults during the simulation, Fastiron does a single allocation at the beginning, using a value such that no reallocation may occur later. Quicksilver also allocates memory when handling fission reactions. This is not the case in Fastiron thanks to the data types provided by the `tinyvec` crate.

Note that executing both programs in parallel leads to more allocations. In the case of Fastiron, it is because of the chunk split and local storages. This growth scales logarithmically.

Using `perf`, one can use hardware counters to evaluate cache efficiency of both programs. Here are the results of both programs, using the same setup as before:

```shell

 Performance counter stats for
   './target/release/fastiron -i input_files/QS_originals/CTS2_Benchmark/CTS2_1.inp':

   791 958 021 205      cycles
   322 317 007 268      cpu/event=0x0e,umask=0x01,inv,cmask=0x01/ # stalls
 1 295 722 789 397      instructions                              # 1,64 insn per cycle
       261 144 359      L1-icache-load-misses
   280 510 551 624      L1-dcache-loads
    38 795 050 549      L1-dcache-load-misses        # 13,83% of all L1-dcache accesses
     6 789 551 440      LLC-loads
     1 413 617 231      LLC-load-misses               # 20,82% of all LL-cache accesses

     497,771336852 seconds time elapsed


 Performance counter stats for
   './qs -i ../Examples/CTS2_Benchmark/CTS2_1.inp':

 1 242 677 262 453      cycles
   434 517 931 523      cpu/event=0x0e,umask=0x01,inv,cmask=0x01/ # stalls
 2 705 248 900 164      instructions                              # 2,18 insn per cycle
       248 305 385      L1-icache-load-misses
   828 776 324 811      L1-dcache-loads
    33 176 119 251      L1-dcache-load-misses         # 4,00% of all L1-dcache accesses
     5 214 117 327      LLC-loads
       808 586 242      LLC-load-misses               # 15,51% of all LL-cache accesses

     780,418622286 seconds time elapsed
```

We can see from the results that Quicksilver has a significantly lower miss rate than Fastiron. This high miss rate might have been a very important issue if Fastiron had a similar total number of instructions and memory accesses as Quicksilver, but that is not the case. Fastiron completes execution with fewer than half the instructions and loads of Quicksilver.

Considering both observations, I would say that Fastiron was well optimized in terms of computations and control flow, but could benefit from better memory management, granted finer control is possible.

## III.B. The Rust Language

### III.B.1. Added Value

The purpose of the internship was to evaluate Rust's capabilities for a given type of program: Monte-Carlo particle transport codes. Because Fastiron was developed from scratch using a C++ code as reference, I was able to evaluate the language on multiple aspects.

Unlike C++ or other less restrictive languages, Rust enforces a set of strict rules –stricter than what `-Wpedantic -Wall -Werror` provides for example–. These rules come under the form of the ownership system, the borrow checker and the compiler's strong static type system. This means that the developer spends more time getting his first implementation up and running; However, it is also why the language comes with many of its guarantees.

The strict rules notably allow developers to refactor their code much more easily, by catching potential issues at compile-time. This is further supported by tools of the ecosystem such as `rustfmt` or `clippy`, allowing users to create cleaner code.

Despite the seemingly more verbose syntax, the Rust port ended up being much shorter than the original C++ program. This is extremely interesting in the context of proxy-application development as the purpose of such programs is to allow developers to manipulate less code when profiling and measuring performance.

The ownership model also ensures thread safety by preventing data races and other concurrency issues at compile-time. As for actual tools, Rust's standard library provides a number of implementation for explicit parallelization such as standard threads or communication channels. It also provides synchronization primitives such as atomics, mutexes or read-write locks. In our case, we used `rayon`, a crate implementing high-level abstractions for data parallelism. This high-level API coupled with thread safety allowed me to work first on a sequential version to ensure result correctness before introducing parallelism to the code.

This type of workflow (cf. Figure 13) is not ideal for a regular C++/OpenMP implementation such as Quicksilver. Without the borrow checker to spot potential data races at compile time, it is possible to have the program compile and not work as intended, without any warnings. In the best case, this result in an aberrant output, but the issue can be more subtle. This is where Rust shines by catching errors at compile time and pointing them out very clearly to the developers. Instead of spending time searching for the mistake, they can focus on fixing the code and do not have to worry about incoherence induced by synchronization hazards.

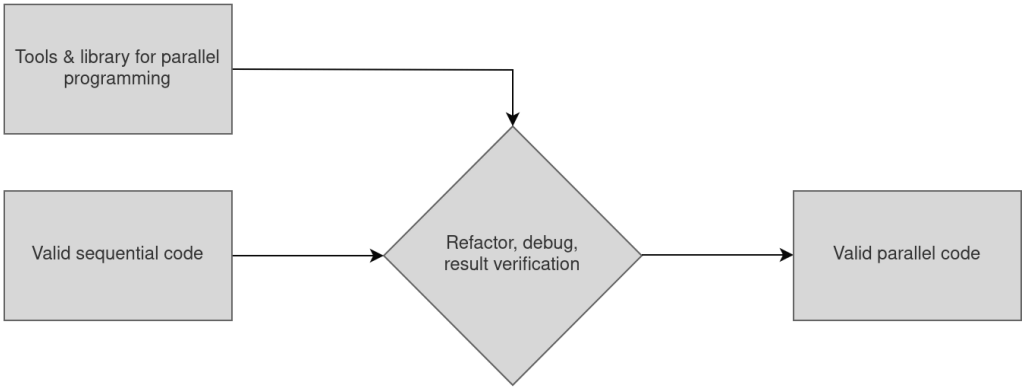



Figure 13: Development workflow.

Note that searching for the mistake in the HPC context usually means having to work with programs using up to 64, 128 or even 256 threads. Step-by-step debugging is not really an option, so the process usually consists of adding and removing locks and atomics randomly on variables. This, coupled with the inconsistency between runs of a faulty parallel program, makes debugging a really tedious task that can take surprisingly large amounts of time,

hence the value of what Rust is providing. As an indication, **obtaining a working parallel implementation from the sequential one took less than three days**.

Checking for result correctness or various properties of our code is also made easier by the Rust ecosystem. The language has a strong focus on testing and provides a built-in framework to this end. Using `cargo test`, developers can write unit and integration tests. A collection of macros are available to use in both tests and code. These macros are not only useful for debugging, but also allow the compiler to optimize away some computations (e.g. bounds check [9]).

Another element contributing to the robustness of a Rust code is the error handling system. The `Result` and `Option` types allow us to catch errors early and gives us fine control over their handling: One could either adapt and proceed differently, crash the program or even propagate the error to the top level caller. These types are defined using `enum` and can be used instead of magic values to discriminate correct return values from incorrect ones. This, coupled with strong pattern matching, allows the developer to produce flexible code where adding or removing cases is trivial as missing logic is detected at compile time.

All of this is complemented nicely by the documentation system. Rust encourages good documentation practices by providing built-in support for generating documentation from code comments. The standard documentation tool, `rustdoc`, allows developers to write documentation comments using Markdown and generate HTML or other formats for their libraries and applications. In our case, the documentation was automatically generated and published on Github pages using continuous integration. While the documentation written is not as detailed as one of a library, it allowed me to keep a clear overview of the entire code during the development process.

### III.B.2. Limits

While Rust provides a lot of very interesting tools and features in the context of a Monte-Carlo particle transport code, I was also confronted with a number of limits. Some from the language itself, others from used libraries.

The `rayon` library, while providing very convenient abstractions, gives the developer very limited control over the parallelization process. Rayon operates by recursively splitting work into smaller tasks that can be executed in parallel. The workload is dynamically evaluated and tasks are scheduled accordingly. While this leads to good work balancing, assuming the tasks' costs aren't too irregular, it leaves a lot to be desired on other aspects of data parallelism. First, the dynamic scheduling means that the tasks dispatching is inconsistent from one run to another. It also excludes from the get-go clever optimizations such as grouping tasks that require the same data to reduce contention.

`rayon` operates very well on immutable data. On the other hand, operating on mutable data really accentuates the limit of its model. The latency introduced by the usage of synchronization primitive represented the prime contention issue during the development of Fastiron `v1.3`. What makes this really problematic is the fact that latency increases as the problem scales, since more threads try to access the same synchronized data set. Quicksilver stressed memory latency and data lookup in the first place and `rayon` is not really providing any help with that, especially with the very limited control over task granularity.

As of today, Quicksilver still has much better locality of reference than Fastiron. While I did not write the first implementation with that problem in mind, reducing the number of indirections and grouping particles per cell still wasn't enough to reach Quicksilver's

performance. My guess is that this is the work of the `BulkStorage` system: It provides a single contiguous memory space for all the data used. In Rust, such control of memory is impossible without `unsafe` code blocks. While I could have done this by using the unsafe API of Rust's vectors, its usage goes against the purpose of this internship as many of Rust's guarantees are lost when introducing `unsafe` code. It is easy to write short, non-dangerous `unsafe` blocks, but incorporating a bunch in the main code would make the usage of Rust pointless to some extent.

Another memory-related shortcoming of `rayon` is the lack of control over thread-local variables. In Fastiron, we iterate over particle chunks, using local arrays and tables to avoid locking mutexes and accessing atomics too often. These objects are initialized each time a chunk is processed by a thread, i.e. each time work is dispatched to a given thread. There is currently no simple, straightforward way to have each thread use a pre-initialized set of variables, consistent across dispatches.

### III.B.3. Porting C++

Thanks to the methodology used to develop Fastiron, I was able to gain insights on the portability of C++ code towards Rust. Most of what can be done in C++ can be done in Rust without using `unsafe` features. What will be the cornerstone of the porting process is how familiar the developer is with the reference code. Assuming that the port is done by someone else than the original program, a fair assumption in my opinion, this means that the tediousness of the porting process is directly linked to the quality of the reference code.

Because C++ code can be ported to Rust in a methodical way, the only obstacle to it would be the understanding of the reference code, which is mainly linked to its clarity, documentation and organization.

The main steps of the method I established are as follows:

1. Familiarize yourself with the reference code.
2. If the program has non-trivial inputs and outputs, start by implementing those.
3. Write a skeleton of the code from the top down.
4. Fill the skeleton from the bottom up.
5. Check result correctness with the first full implementation.
6. Proceed with refactors, reorganization, or parallelism.

Note that some trivial refactors can be done at the same time the first implementation is written, but others are much easier to introduce later on. I will detail each step in order to clarify the process.

The first step is to get familiar with the C++ code. This is especially important if the program includes specific elements that will not be ported, whether by choice or by constraints. This will help us later when we navigate the source code while writing the first pieces of code as we will have a rough idea of the layout of the program.

Inputs and outputs are the first elements to implement. Rust provides a number of tools and libraries that make handling I/O extremely easy. I would advise not to rely on the C++ code for this part, but instead look at the behavior of the compiled program. The C++ code can and must be used to ensure correct I/O processing though (e.g. parameter structures initialization in our case). The reason for that is that Rust has plenty of high-level abstractions for I/O handling and using those will be much easier than trying to replicate how the C++ code operates. It will also contribute to code robustness.

Now that we have an empty program that can read input, we can build a skeleton of the application, from the top down. This allows us to get familiar with the program's execution flow and ensure that we do not forget any calls or operations. It is at this step that we can start modifying the code to have an easier time later:

- Classes become structures, all attributes are qualified as public to remove the need for getters and setters.
- Abstract classes and interfaces can be either made into traits or ignored entirely depending on their importance.
- Pointers and references become borrows. Try to spot any `const` keyword to help you specify mutability.

From the skeleton, we can implement the code from the bottom up. This allows us to write unit tests at the same time, ensuring that each new implementation relies on tested code. If there are tests, in the C++ code, they can be taken verbatim. It is possible to do additional modifications:

- Update function signatures:
  - Adjust the mutability of borrows if needed.
  - Return the result directly instead of fetching it through pointers.
- Replace simple `for` indexed loops by iterators. Other loops can be tweaked after the first implementation for simplicity's sake.
- Replace magic values by `Option` or `Result` types. There are specific cases where it leads to bad performance, but it can be corrected easily later.

By following the first four steps, there should be only a limited amount of debugging from there to get a fully working implementation. Most of the work will probably be making the program compile. That first implementation should be used to cross-reference the results of our program with the results of the C++ code. By first making sure that the results and their behavior are the same, we can then work on additional refactors without uncertainties if a problem occurs.

The additional changes include but are not limited to:

- Introducing matching patterns and enums to replace repetitive `if {} else {}`.
- Rework the more complex loops to use iterators. One of the specific cases where this is impossible is a loop using an element and modifying another; This will have to stay indexed. Otherwise, iterators and `for` loops seem to yield the best performance. `while` and `loop` blocks seem to produce convoluted assembly.
- Minimize the borrows; Take only what is needed.
- Use common traits such as `Default`, `Index` or `From` to shorten the code.

While I did not follow this exact workflow, I operated mostly according to it. This allowed me to notice what was efficient, and what could have been more efficient. This methodology I suggest may not be the indisputable best, but it is a very solid starting point to port C++ code to Rust.

If the developer wishes to introduce parallelism, it should be considered while writing the first refactors following the naive implementation. There are a few guidelines to smooth the process:

- Prior to adding parallelism, separate mutable data from immutable data. This will make it easier to respect the borrowing rules and introduce synchronization.

- Focus on having a first working “parallel” implementation with no performance consideration. It is easier in Rust to refactor later than to try to optimize everything from the get-go.

In order to get the first implementation up and running, it is possible to add synchronized types at the highest level possible. This will result in a pseudo-parallel implementation where threads lock entire structures to run their routines, effectively running one at a time and yielding similar performance to the sequential implementation. From there, the synchronization can be adjusted down to the finest or most optimal level.

# IV. Conclusion

## IV.A. State of the Project

Currently, Fastiron can run all problems defined by Quicksilver. Core features of the command line interface are replicated; Additionally, Fastiron:

- implements control over the parallel execution through a command line argument.
- can execute using `f32` or `f64`, depending on a CLI argument
- can save tallied data and timers' report as a `csv` file for easy post-processing.
- is fully documented along with its companion tool, `fastiron-stats`.

The program implements roughly the same algorithm as Quicksilver, using `rayon` to achieve data-parallelism. It totals approximately 6,000 lines of Rust code, while the repository totals 9,500. On the other hand, Quicksilver totals 13,000 lines of C++ code.

A larger rework was theorized to obtain a dataflow-oriented implementation. This would imply fundamentally changing the followed algorithm and may not be fully implemented at the time this report is submitted. The main issue we identified in such an algorithm would stem from the population control and particle generation implementations.

## IV.B. Rust for Monte-Carlo Particle Transport Proxy-Application

Overall, I would say that the internship is a success. By allowing the developer to write robust and concise code, the Rust programming language solidifies its viability for proxy-application development.

The main code is only 6,000 lines long, which is less than half the original. This is extremely interesting in the context of proxy-applications as it allows the developer to work on a very limited amount of code. This, coupled with the strict rules enforced at compilation, greatly facilitates refactoring. These two elements alone make Rust a prime choice for proxy-application development.

Thanks to the guarantees it brings, notably the *fearless concurrency*, Rust also shows viability for the development of massively parallel programs such as Monte-Carlo particle transport codes. Because debugging a parallel program is extremely tedious, detecting issues at compile time is a very valuable feature. This is due to two fundamental aspects of parallel programming: inconsistencies between runs and the unintuitive nature of parallel execution. These two factors make searching for errors in the source code significantly harder than in a sequential context. By detecting errors at compile time, parallel specific issues can be avoided from the start.

The functional approach of the language also contributes, in my opinion, to a more intuitive representation of parallelism. While indices naturally imply ordering, iterators, closures and mapping functions are much more neutral.

# V. Appendices

## V.A. Amdahl's Law

Amdahl's Law [17] refers to a formula giving the theoretical speedup of a system with scaling resource:

$S = \frac{1}{(1-p)+\frac{p}{N}}$, – $p$ the fraction of code that is parallelizable, $N$ the computing resource

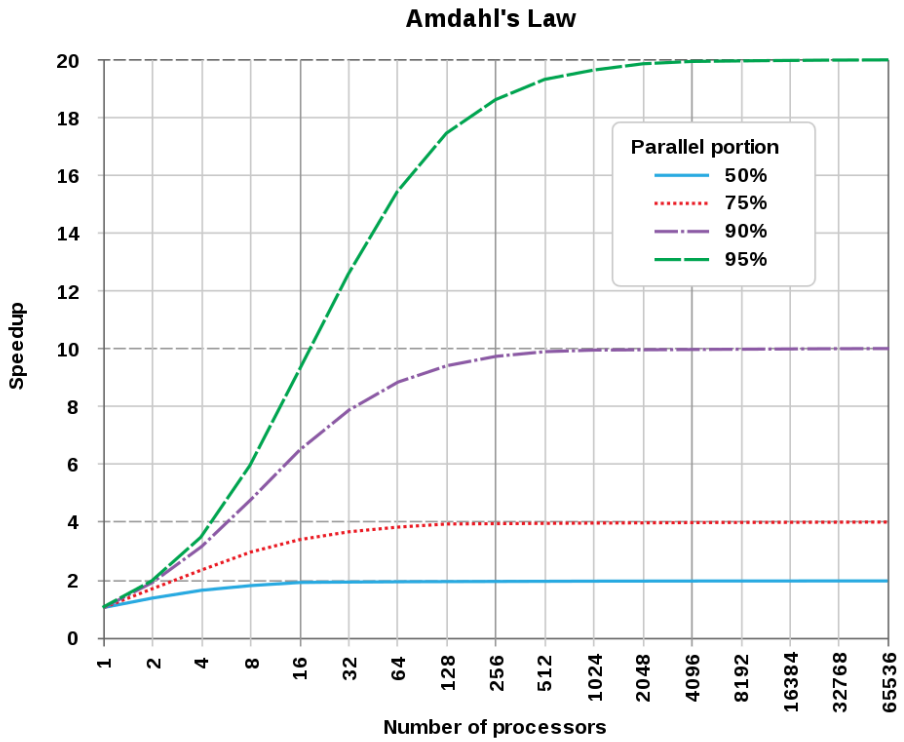



Figure 14: Amdahl's Law predictions for different parallelizability.

In our context, the scaling resource is the number of cores used for data parallelism. **I considered the parallel fraction $p$ to be equal to 1**. This was possible because the tracking section of the algorithm has a specific timer, and that section is (in theory) embarrassingly parallel. The speedup computations are done using this timer's values, allowing me to make this assumption for $p$.

## V.B. Code Snippets

The snippets from Quicksilver are taken verbatim, only indentation may slightly differ for readability purposes.

### V.B.1. Quicksilver – `BulkStorage` Class

```cpp
#ifndef BULK_STORAGE_HH
#define BULK_STORAGE_HH

#include "MemoryControl.hh"

```

```
6    template <typename T>
7    class BulkStorage
8    {
9     public:
10     BulkStorage()
11     : _bulkStorage(0),
12       _refCount(0),
13       _size(0),
14       _capacity(0),
15       _memPolicy(MemoryControl::AllocationPolicy::UNDEFINED_POLICY)
16     {
17        _refCount = new int;
18        *_refCount = 1;
19     }
20
21     BulkStorage(const BulkStorage& aa)
22     : _bulkStorage(aa._bulkStorage),
23       _refCount(aa._refCount),
24       _size(aa._size),
25       _capacity(aa._capacity),
26       _memPolicy(aa._memPolicy)
27     {
28        ++(*_refCount);
29     }
30
31     ~BulkStorage()
32     {
33        --(*_refCount);
34        if (*_refCount > 0)
35           return;
36
37        delete _refCount;
38
39        // Catch the case that the storage was never allocated.  This
40        // happens when setCapacity is never called on this instance.
41        if (_bulkStorage != 0)
42           MemoryControl::deallocate(_bulkStorage, _capacity, _memPolicy);
43     }
44
45     /// Needed for copy-swap idiom
46     void swap(BulkStorage<T>& other)
47     {
48        std::swap(_bulkStorage, other._bulkStorage);
49        std::swap(_refCount,    other._refCount);
50        std::swap(_size,        other._size);
51        std::swap(_capacity,    other._capacity);
52        std::swap(_memPolicy,   other._memPolicy);
53     }
54
55     /// Implement assignment using copy-swap idiom
56     BulkStorage& operator=(const BulkStorage& aa)
57     {
58        if (&aa != this)
```

```
59          {
60             BulkStorage<T> temp(aa);
61             this->swap(temp);
62          }
63          return *this;
64       }
65
66       void setCapacity(int capacity, MemoryControl::AllocationPolicy policy)
67       {
68          qs_assert(_bulkStorage == 0);
69          _bulkStorage = MemoryControl::allocate<T>(capacity, policy);
70          _capacity = capacity;
71          _memPolicy = policy;
72       }
73
74       T* getBlock(int nItems)
75       {
76          T* blockStart = _bulkStorage + _size;
77          _size += nItems;
78          qs_assert(_size <= _capacity);
79          return blockStart;
80       }
81
82
83     private:
84
85       // This class doesn't have well defined copy semantics.  However,
86       // just disabling copy operations breaks the build since we haven't
87       // been consistent about dealing with copy semantics in classes like
88       // MC_Mesh_Domain.
89
90
91
92       T* _bulkStorage;
93       int* _refCount;
94       int _size;
95       int _capacity;
96       MemoryControl::AllocationPolicy _memPolicy;
97
98    };
99
100
101 #endif
```

### V.B.2. Quicksilver – Mesh Data Classes

```cpp
1  #ifndef MC_DOMAIN_INCLUDE
2  #define MC_DOMAIN_INCLUDE
3
4
5  #include "QS_Vector.hh"
6  #include "MC_Facet_Adjacency.hh"
7  #include "MC_Vector.hh"
```

```
8  #include "MC_Cell_State.hh"
9  #include "MC_Facet_Geometry.hh"
10 #include "BulkStorage.hh"
11
12 class Parameters;
13 class MeshPartition;
14 class GlobalFccGrid;
15 class DecompositionObject;
16 class MaterialDatabase;
17
18
19 //--------------------------------------------------------
20 // class that manages data set on a mesh like geometry
21 //--------------------------------------------------------
22
23 class MC_Mesh_Domain
24 {
25  public:
26
27    int _domainGid; //dfr: Might be able to delete this later.
28
29    qs_vector<int> _nbrDomainGid;
30    qs_vector<int> _nbrRank;
31
32    qs_vector<MC_Vector> _node;
33    qs_vector<MC_Facet_Adjacency_Cell> _cellConnectivity;
34
35    qs_vector<MC_Facet_Geometry_Cell> _cellGeometry;
36
37
38
39    BulkStorage<MC_Facet_Adjacency> _connectivityFacetStorage;
40    BulkStorage<int> _connectivityPointStorage;
41    BulkStorage<MC_General_Plane> _geomFacetStorage;
42
43     // -------------------------- public interface
44    MC_Mesh_Domain(){};
45    MC_Mesh_Domain(const MeshPartition& meshPartition,
46                   const GlobalFccGrid& grid,
47                   const DecompositionObject& ddc,
48                  const qs_vector<MC_Subfacet_Adjacency_Event::Enum>& boundaryCondition);
49
50 };
51
52
53
54 //-----------------------------------------
55 // class that manages a region on a domain.
56 //-----------------------------------------
57
58 class MC_Domain
59 {
60 public:
```

```
    int domainIndex;  // This appears to be unused.
    int global_domain;

    qs_vector<MC_Cell_State> cell_state;

    BulkStorage<double> _cachedCrossSectionStorage;

     // hold mesh information
     MC_Mesh_Domain mesh;

    // -------------------------- public interface
     MC_Domain(){};
     MC_Domain(const MeshPartition& meshPartition, const GlobalFccGrid& grid,
               const DecompositionObject& ddc, const Parameters& params,
               const MaterialDatabase& materialDatabase, int numEnergyGroups);


    void clearCrossSectionCache(int numEnergyGroups);
};

#endif
```

### V.B.3. Particle Processing Workflow

#### V.B.3.1. Original Workflow

```rust
// fn main() {
// ...
    let mut done = false;
    loop {
        while !done {
            let mut fill_vault: usize = 0;
            for processing_vault_idx
              in 0..mcco.particle_vault_container.processing_vaults.len() {
                // Computing block
                mc_fast_timer::start(mcco, Section::CycleTrackingKernel);

                // number of VALID particles in current vault
                let num_particles =
                    mcco.particle_vault_container.processing_vaults[processing_vault_idx].si

                if num_particles != 0 {
                    let mut particle_idx: usize = 0;
                    while particle_idx < mcco.particle_vault_container.vault_size {
                        cycle_tracking_guts(mcco, particle_idx, processing_vault_idx);
                        particle_idx += 1;
                    }
                }

                mc_fast_timer::stop(mcco, Section::CycleTrackingKernel);

                // Inter-domain communication block
                mc_fast_timer::start(mcco, Section::CycleTrackingComm);
```

```
28
29                  let send_q = &mut mcco.particle_vault_container.send_queue;
30
31                  for idx in 0..send_q.size() {
32                      let send_q_t = send_q.data[idx].clone();
33                      let mcb_particle = mcco.particle_vault_container.processing_vaults
34                          [processing_vault_idx]
35                          .get_base_particle(idx);
36
37                      mcco.particle_buffer
38                          .buffer_particle(mcb_particle.unwrap(), send_q_t.neighbor);
39                  }
40
41                  send_q.clear();
42
43                  mcco.particle_vault_container.clean_extra_vaults();
44                  mcco.read_buffers(&mut fill_vault);
45
46                  mc_fast_timer::stop(mcco, Section::CycleTrackingComm);
47              }
48
49              mc_fast_timer::start(mcco, Section::CycleTrackingComm);
50
51              mcco.particle_vault_container.collapse_processing();
52              mcco.particle_vault_container.collapse_processed();
53              done = mcco.particle_buffer.test_done_new(mcco);
54
55              mc_fast_timer::stop(mcco, Section::CycleTrackingComm);
56          }
57
58          done = mcco.particle_buffer.test_done_new(mcco);
59
60          if done {
61              break;
62          }
63      }
64  // ...
65  // }
```

### V.B.3.2. Rustified Workflow

```
1   // fn main() {                                                    Rust
2   // ...
3   loop {
4       while !done {
5           mc_fast_timer::start(mcco, Section::CycleTrackingKernel);
6
7           // track particles
8           container
9               .processing_particles
10              .iter_mut()
11              .for_each(|base_particle| {
12                  cycle_tracking_guts(
```

```
                    mcco,
                    base_particle,
                    &mut container.extra_particles,
                    &mut container.send_queue,
                );
                if base_particle.species != Species::Unknown {
                    container.processed_particles.push(base_particle.clone());
                }
            });
        container.processing_particles.clear();

        mc_fast_timer::stop(mcco, Section::CycleTrackingKernel);
        mc_fast_timer::start(mcco, Section::CycleTrackingComm);

        // clean extra here
        container.process_sq();
        container.clean_extra_vaults();

        done = container.test_done_new();

        mc_fast_timer::stop(mcco, Section::CycleTrackingComm);
    }
    done = container.test_done_new();

    if done {
        break;
    }
}
// ...
// }
```

## V.C. Core-to-Core Latency

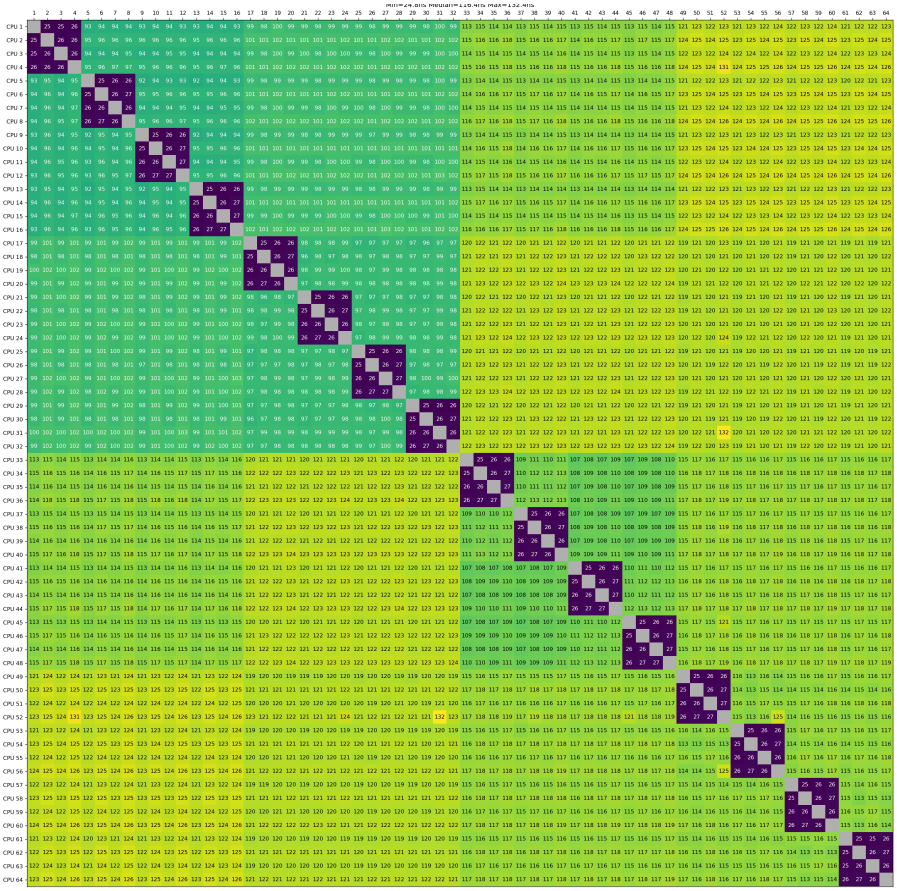


Figure 15: Single socket latency (one CPU).

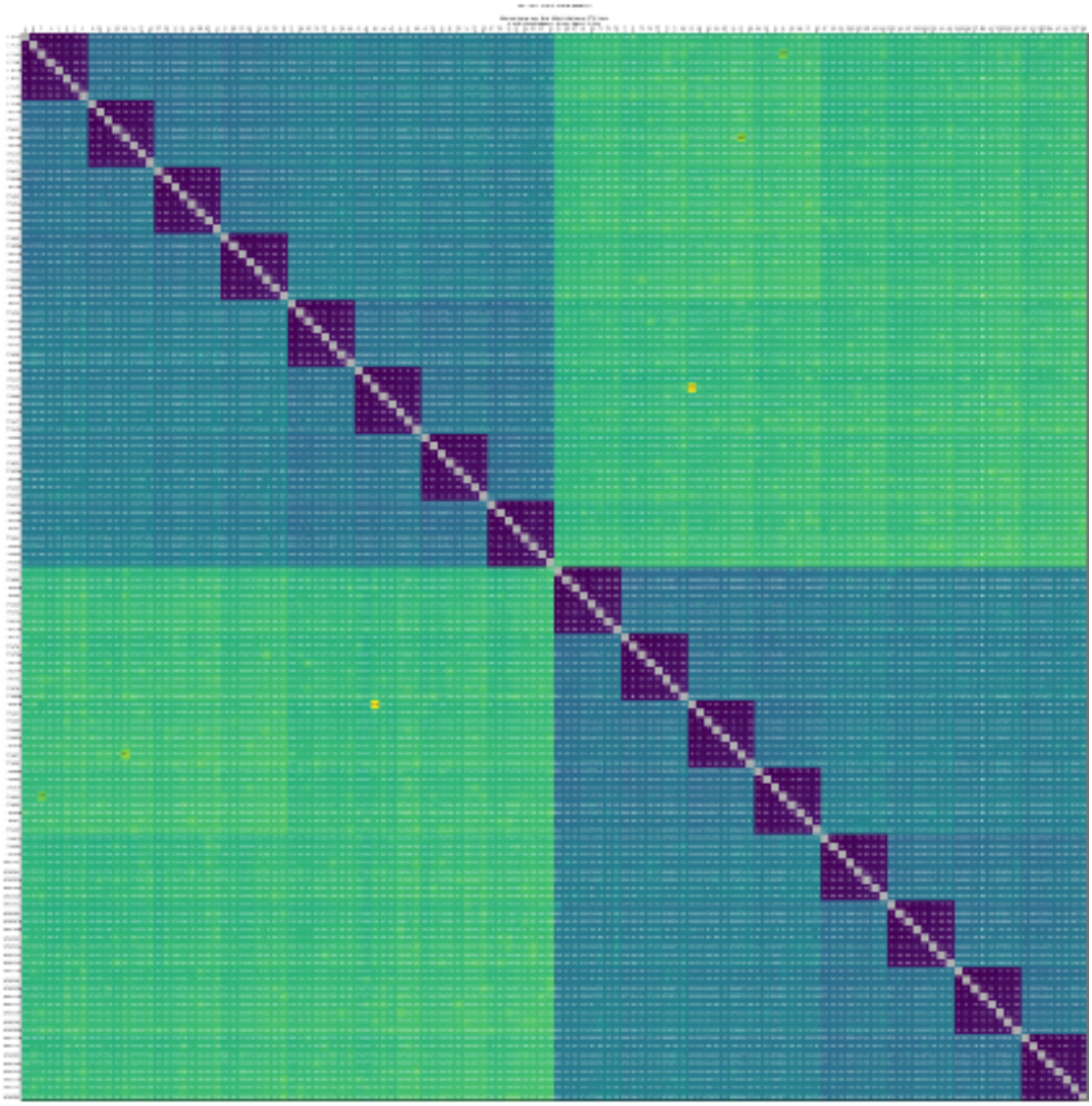


Figure 16: Two socket latency (full node).

# Bibliography


[1] "Fastiron - A Rust Code for Monte-Carlo Particle Transport Simulations." Accessed: Jul. 25, 2023. [Online]. Available: https://cea-hpc.github.io/fastiron/

[2] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, and M. J. O'Brien, "Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 866–873. doi: 10.1109/CLUSTER.2017.121.

[3] J. Guttag, "Lecture 6: Monte Carlo Simulation | Introduction to Computational Thinking and Data Science | Electrical Engineering and Computer Science." Accessed: Jul. 25, 2023. [Online]. Available: https://ocw.mit.edu/courses/6-0002-introduction-to-computational-thinking-and-data-science-fall-2016/resources/lecture-6-monte-carlo-simulation/

[4] P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith, "OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development," *Annals of Nuclear Energy*, vol. 82, pp. 90–97, Aug. 2015, doi: 10.1016/j.anucene.2014.07.048.

[5] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 5.2." Accessed: Jul. 25, 2023. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf

[6] E. Gabriel *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Sep. 2004, pp. 97–104.

[7] "Num - Rust." Accessed: Jul. 25, 2023. [Online]. Available: https://docs.rs/num/latest/num/

[8] N. Nethercorte, "Iterators - The Rust Performance Book." Accessed: Jul. 31, 2023. [Online]. Available: https://nnethercote.github.io/perf-book/iterators.html

[9] N. Nethercorte, "Bounds Checks - The Rust Performance Book." Accessed: Jul. 25, 2023. [Online]. Available: https://nnethercote.github.io/perf-book/bounds-checks.html

[10] "Perf Wiki." Accessed: Jul. 25, 2023. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[11] B. Gregg, "Flame Graphs." Accessed: Jul. 25, 2023. [Online]. Available: https://www.brendangregg.com/flamegraphs.html

[12] "Intel oneAPI." Accessed: Jul. 25, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

[13] B. Heisler, "Criterion.Rs." Accessed: Jul. 25, 2023. [Online]. Available: https://github.com/bheisler/criterion.rs

[14] "Heaptrack - a Heap Memory Profiler for Linux." Accessed: Jul. 25, 2023. [Online]. Available: https://github.com/KDE/heaptrack

[15] "Relative Change and Difference." Accessed: Jul. 25, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Relative_change_and_difference&oldid=1164822526#Definition

[16] "Rayon - Rust." Accessed: Jul. 25, 2023. [Online]. Available: https://docs.rs/rayon/latest/rayon/index.html

[17] “Amdahl's Law.” Accessed: Jul. 26, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=1154304432

[18] “CTS2 Report.” Accessed: Jul. 25, 2023. [Online]. Available: https://hpc.llnl.gov/sites/default/files/Quicksilver_CTS.pdf

[19] “Coral2 Report.” Accessed: Jul. 25, 2023. [Online]. Available: https://asc.llnl.gov/sites/asc/files/2020-09/quicksilver_coral2_v2.pdf

[20] N. Viennot, “Measuring CPU Core-to-Core Latency.” Accessed: Jul. 25, 2023. [Online]. Available: https://github.com/nviennot/core-to-core-latency

# Glossary

### Non-Technical

- **CEA**: Commissariat à l'énergie atomique et aux énergies alternatives
- **LLNL**: Lawrence Livermore National Laboratory

### High Performance Computing

- **HiB**, **LiB**: Higher is Better, Lower is Better; Used to qualify a metric.
- **MPI**: Message Passing Interface; MPI is an API-defining standard designed for distributed memory parallelism. For simplicity's sake, *MPI* can refer to either the standard or one of its implementation.
- **OpenMP**: Open Multi-Processing; OpenMP is an API designed for shared-memory parallelism and can be used in C, C++, or Fortran codes.
- **Proxy-app**: Proxy applications are small, simplified codes that allows developers to evaluate effects of hardware changes, refactors, etc. without having to manipulate too large amounts of code.
- **SIMD**: Single Instruction, Multiple Data; It is a type of parallel processing defined in Flynn's taxonomy.
- **SPMD**: Single Program, Multiple Data; It is a subcategory of MIMD (Multiple IMD), a type of parallel processing defined in Flynn's taxonomy.

### Program Specific

- **Collision**: In our context, collision refers to a reaction of the particle with its environment. There are three reaction types implemented: absorption, fission, and scattering.
- **Cross-section**: Measure of the probability that a given event will take place, in our case, a collision.
- **FoM**: Figure of Merit; Main performance metric of Quicksilver and Fastiron. It corresponds to the number of segments computed per second. A segment is an iteration of the internal particle tracking loop.
- **PRNG**: Pseudo Random Number Generator. Note that while the study of PRNG is essential to Monte-Carlo methods, it falls outside the scope of the internship.
- **Tallies**: Structure used to record data of interest in the simulation, e.g. energy spectrum. The original program also included an event counter and an unused decimal-storing structure.