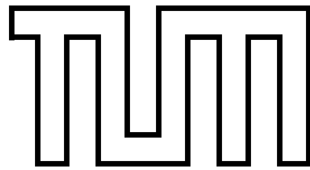


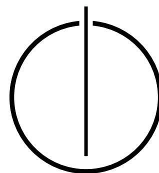
FAKULTÄT FÜR INFORMATIK

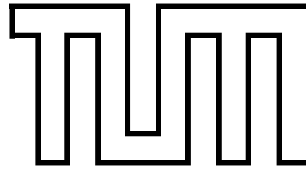
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Interdisciplinary Project Report

Measuring and Optimizing the Energy Efficiency of Molecular Dynamics Simulations

Vincent Fischer





FAKULTÄT FÜR INFORMATIK

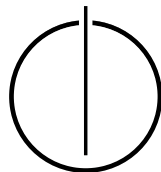
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Interdisciplinary Project Report

Measuring and Optimizing the Energy Efficiency of Molecular Dynamics Simulations

Messung und Optimierung der Energieeffizienz von Molekulardynamik Simulationen

Author:	Vincent Fischer
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	2023-05-24



Abstract

AutoPas is a particle simulation library that aims to reduce the time to solution of molecular dynamics simulations by automatically choosing the most appropriate strategy for a given scenario. With the rising cost of energy, however, it becomes more and more important to consider not only computation time but also the energy consumed in the process. Modern processors provide interfaces to monitor their energy consumption. In this project, we analyze the relationship between time to solution and energy consumption, first in a simple-to-understand problem, and then in a molecular dynamics simulation. We show that the fastest algorithm is not necessarily the most energy efficient. We also extend AutoPas by the ability to optimize for energy consumption instead of time to completion.

Zusammenfassung

AutoPas ist eine Partikelsimulationsbibliothek, welche dazu dient, die Berechnungszeit von Molekulardynamiksimulationen zu reduzieren, indem sie automatisch für jedes Szenario die am besten geeigntete Lösungsstrategie wählt. Aufgrung der steigenden Energiekosten wird es immer wichtiger, nicht nur die Berechnungszeit, sondern auch die wärhrend der Simulation verbrauchte Energie zu beachten. Moderne Prozessoren stellen Schnittstellen zur verfügung, über die der Energieverbrauch gemessen werden kann. In diesem Projekt analysieren wir, erst an einem einfach zu verstehenden Problem und dann an einer Molekulardynamiksimulation, den Zusammenhang zwischen Berechnungszeit und Energieverbrauch. Wir zeigen, dass der schnellste Algorithmus nicht zwingend auch der energieeffizienteste ist. Zudem erweitern wir AutoPas um die Möglichkeit auf Energieverbrauch zu optimieren, anstatt auf Berechnungszeit.

Contents

Abstract . . . v
Zusammenfassung . . . vii

1 Introduction and Background **1**
1.1 Introduction . . . 1
1.2 AutoPas . . . 1
1.2.1 Tuning . . . 1
1.2.2 Containers . . . 1
1.2.3 Traversals . . . 2
1.2.4 Data Layout . . . 3
1.3 Energy Measurement on Intel CPUs . . . 3

2 Preliminary Benchmarks **5**
2.1 Description of CSR Format . . . 5
2.2 Effect of Multithreading on Power Draw . . . 5
2.3 Effect of Computational Load on Per-Thread Power Draw . . . 6

3 Energy Measurement in AutoPas **9**
3.1 Implementation . . . 9
3.2 Effect of Configuration Options on Energy Usage . . . 9
3.2.1 Experiment Setup . . . 9
3.2.2 Results from Equilibration Step . . . 10
3.2.3 Results from Spinodal Decomposition . . . 11

4 Conclusion and Future Work **14**

Bibliography **15**

1 Introduction and Background

1.1 Introduction

Particle simulations are an important tool in many branches of science, such as computational chemistry. These simulations calculate the forces between particle pairs and use numerical integration methods to simulate their movement. The exact solution would require $\mathcal{O}\left(n^2\right)$ time due to the pairwise force calculation. Multiple strategies exist to improve this runtime by introducing a cutoff distance, such that particle pairs with larger distances from each other are no longer considered. Which of these strategies is most suited to a certain scenario is not always clear. Libraries such as AutoPas solve this problem by dynamically adjusting their behavior to the current state of the simulation. This approach is described in detail in [1]. Initially, AutoPas was designed to reduce the time to solution for simulations; however, the same approach can also be used for other metrics. In this project, we extend AutoPas by the ability to monitor energy consumption and to optimize for it instead of time to solution.

1.2 AutoPas

AutoPas is a node-level particle simulation library written in C++. It provides multiple algorithms for efficiently computing N-body simulations, and can automatically choose the configuration best suited to a certain scenario. A configuration consists of, among other things, a container, a shared memory parallelization algorithm, called traversal, for said container, and a data layout.

1.2.1 Tuning

AutoPas selects the best algorithm by running a few iterations of the simulation with a variety of different configurations and measuring the time to completion for each. This process is called tuning and is repeated at regular intervals. In between, the configuration with the fastest average time to completion, as determined in the previous tuning session, is used.

1.2.2 Containers

Containers are data structures used to store the particles in the simulation. In this project, we mainly considered two available options: The linked cells container and the verlet lists cells container. These are used for inter-particle forces that drop off to zero quickly enough as the distance between both particles increases, that it suffices to consider only those particle pairs that are within a certain cutoff distance of each other.

Linked Cells

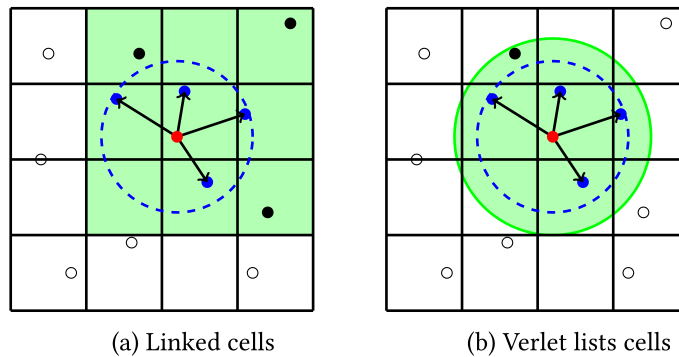
This container divides the domain into a grid of equally sized cuboid cells. If a particle a is in one cell, then it suffices to check each nth-degree neighbor of that cell for particles within the cutoff

distance to a, where the interaction length n is given by $\left\lceil \frac{\text{cutoff}}{\text{cell size}} \right\rceil$.

Verlet Lists Cells

This container builds on the linked cells container but additionally saves for each particle a a list of “neighboring” particles that are within a distance $d + \epsilon$ of a, where d is the cutoff distance and $\epsilon > 0$ is some additional distance which allows for some movement of the particles before the neighbor lists have to be updated. Rebuilding the neighbor lists is an expensive step, as it requires what is essentially a whole extra iteration. For this, it uses the traversal algorithm of the underlying linked cells container. But in between some time may be gained as only the particles in the neighbor list need to be checked.

?? visualizes the difference between these two containers.



(a) Linked cells (b) Verlet lists cells

Figure 1.1: Visualization of linked cells and verlet lists cells containers. Interactions between the red particle and all blue particles are calculated. For particles within the green area, the distance to the red particle has to be calculated. The dotted blue circle represents the cutoff radius.

1.2.3 Traversals

Both containers discussed offer different traversal algorithms. These mainly affect how the simulation is parallelized across multiple threads. The algorithms considered in this thesis can be classified into two categories: Colored traversals and sliced traversals.

Colored Traversals

Colored traversals assign a color to each cell of the container, in such a way that all cells with the same color can be processed in parallel while ensuring that no data races occur. This is done by applying a base step to each cell that defines which particle interactions get computed in each step. The combination of base step and coloring must ensure that there is no overlap of affected particles when applying the base step to two different cells of the same color. Then, for each color in turn, the base step can be applied in parallel to all cells of that color. **??** visualizes two options for such a base step.

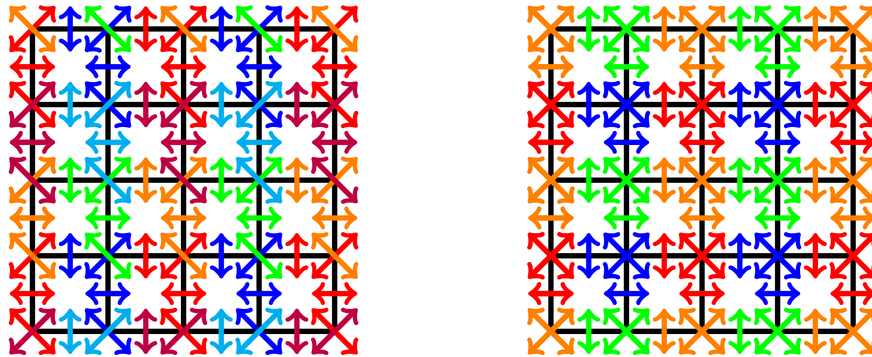


Figure 1.2: Comparison of c18 base step (left) and c08 base step (right). The colors mark the base cell to which the interaction is grouped.

Sliced Traversals

In this project, we only consider two variations of sliced traversals: sliced-balanced and sliced-c02. In both cases, the domain is divided into slices along the longest dimension. Sliced-balanced creates exactly one slice for each thread available. Optionally, it uses a heuristic that estimates the computational load of each cell to determine the thickness of each slice in such a way, that each slice receives equal load. We will not be using a heuristic in this project, however, and simply assign slices of equal thickness to each thread. Sliced-c02 instead creates as many slices as possible and uses dynamic scheduling to balance out the load among all threads. To prevent data races at the slice barriers, each thread places locks on the starting layers of the slice it is currently processing. This, of course, means that slices must have some minimum thickness, to ensure there are no interactions between particles in the first layer of one slice and those in the first layer of the next slice.

1.2.4 Data Layout

The data within the containers can be arranged in two different ways. The simplest is an Array of Structs (AoS), where each struct defines one particle. The other option is a Structure of Arrays (SoA). The structure contains one array for each property of the particle type (x, y, and z coordinates, mass, etc.) the i-th particle is then defined by the i-th entry in each of these arrays. This data layout allows for more efficient vectorization of the force computations.

1.3 Energy Measurement on Intel CPUs

Modern Intel CPUs provide the ability to measure the processor's energy consumption through the running average power limit (RAPL) interfaces as described in [2, Chapter 15]. Multiple such interfaces exist for the energy consumed by different domains:

- Power planes PP0 and PP1 representing the core and uncore devices (for example an integrated GPU; most server CPUs do not have this domain),
- Package, which combines the previous two domains,
- DRAM

Each interface consists of several model-specific registers (MSRs), which can be read to monitor energy usage.

There are a few options available to Linux userland applications to use these interfaces:

- The MSRs can be read directly through `/dev/msr` device files. This requires root access.
- RAPL is exposed through the file system under `/sys/class/powercap/intel-rapl/` as documented in [3].
- The perf_event interface can read the energy counters. Documentation of this feature is only available through the `perf list` command and the commit message in the Linux kernel git repository [4].

The last two options can be used without root access, provided the machine is configured to allow it.

2 Preliminary Benchmarks

To get an understanding of the factors that affect the power draw of a system, it is useful to study a simpler program first. For this purpose, a simple program was written that performs a matrix-vector multiplication for a sparse matrix saved in compressed sparse row format (CSR) while utilizing OpenMP for parallelization.

2.1 Description of CSR Format

CSR is a format for saving an $n \times m$ matrix A, which avoids saving zero-entries. Assuming A has N non-zero entries, its CSR representation is given by the vectors

$(v_k)_{k<N}$, where v_k is the k-th non-zero entry of A (traversing row-wise),

$(c_k)_{k<N}$, where c_k is the column index belonging to v_k,

$(a_i)_{i<n}$, where a_i is the index into $(v_k)_k$ of the first non-zero entry in the i-th row of A,

$(b_i)_{i<n}$, where b_i is the index into $(v_k)_k$ of the last non-zero entry in the i-th row of A

Note that ordinarily, it suffices to save either a_i or b_i and setting $a_{i+1} = b_i + 1$; however, saving both will allow us to have two rows i and j be exactly the same by setting $a_i = a_j$ and $b_i = b_j$ This will be useful later.

The matrix-vector product $y \leftarrow A \cdot x$ where x is simply stored as an array can be computed by

$$y_i \leftarrow \sum_{j=a_i}^{b_j} v_j \cdot x_{c_j} \tag{2.1}$$

which is easily parallelizable over the rows of the matrix.

2.2 Effect of Multithreading on Power Draw

For the first benchmark, we compare the power draw of the system while performing the matrix-vector multiplication with a varying number of threads. We consider two different scenarios:

1. Only the first row of the matrix contains non-zero entries.
2. All rows of the matrix contain non-zero entries.

We use a matrix of size 28×10^7, where 28 is the maximum number of threads on the system without utilizing hyperthreading.

For the first case, it is expected that the time to completion does not depend on the number of threads n, as only the first thread will be doing all the work, while the other threads remain

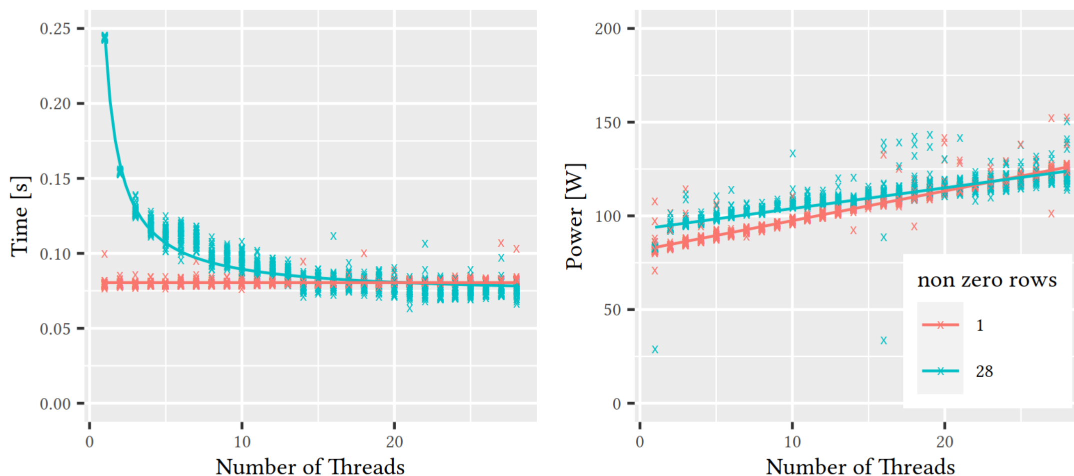


Figure 2.1: Time to completion is constant if only the first row of the matrix is non-zero and inversely proportional to the number of threads if all rows are non-zero (left). The power draw in both cases is linear in the number of threads (right).

idle. In the second case, we expect that the time to completion t is roughly proportional to $\frac{1}{n}$, as the work can be evenly divided among all of the threads. As more threads are performing work, we also expect the power draw to increase. We are especially interested to see if the decrease in computation time is enough to compensate for the higher power draw, to prevent an overall increase in energy usage.

?? depicts the time to completion as well as the power draw for both cases. As can be seen, the time to completion stays constant in the single-row case and is inversely proportional to the thread count in the 28-row case, confirming our predictions. For the power draw, we see a linear dependency on the number of threads used, regardless of whether or not the additional threads are loaded or not. The increase in power draw per additional thread ((1.6 ± 0.2) W in the single row case and (1.10 ± 0.02) W in the 28 row case) is, however, relatively small compared to the baseline power draw of a single thread ((82.00 ± 0.17) W in the single row case and (93.00 ± 0.36) W in the 28 row case). Combined with the speedup gained by multithreading, this results in an overall decrease in accumulated energy usage, as can be seen in **??**. On the other hand, we also see a significant increase in energy usage when adding more idle threads for the case where only the first row contains any non-zero values.

2.3 Effect of Computational Load on Per-Thread Power Draw

In order to better understand this relationship, we perform a second benchmark. For this experiment, each multiplication is performed using all 28 threads but with a varying number of rows filled with non-zero values. The matrix is set up so that each non-zero row is exactly the same and uses the same memory ($a_i = 0$ and $b_i = N/k$ are constant for all k non-zero rows). This was done, because the number of cache misses otherwise increased with k, leading to wildly different execution times. This would have made any comparison much more difficult.

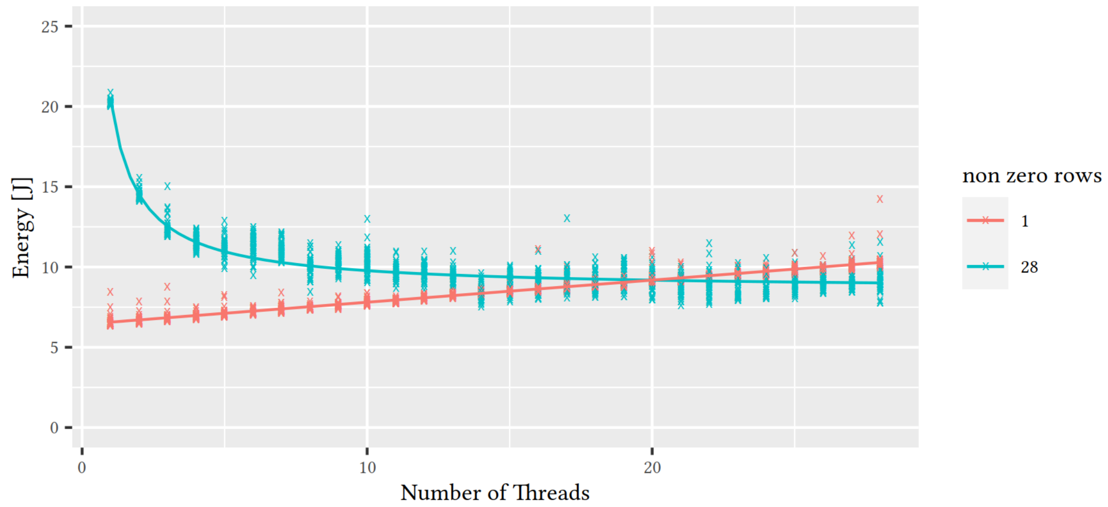


Figure 2.2: Energy consumption is linearly increasing in the number of threads for the single-row case and (apart from constant baseline) inversely proportional to the number of threads in the 28-row case.

As can be seen in **??**, the power draw increases with the number of non-zero rows, from which we can conclude that a thread under load does have a higher power draw than an idle thread. However, this increase of (0.77 ± 0.01) W compared to an idle thread is only about half as large as the increase resulting from adding the thread in the first place, which we saw in **??**. The key takeaway here should be that in order to reduce energy consumption in a multithreaded application, we should try to avoid as much idle time for threads as possible, as we pay at least two-thirds of the full price for each thread used with little regard to its computational load.

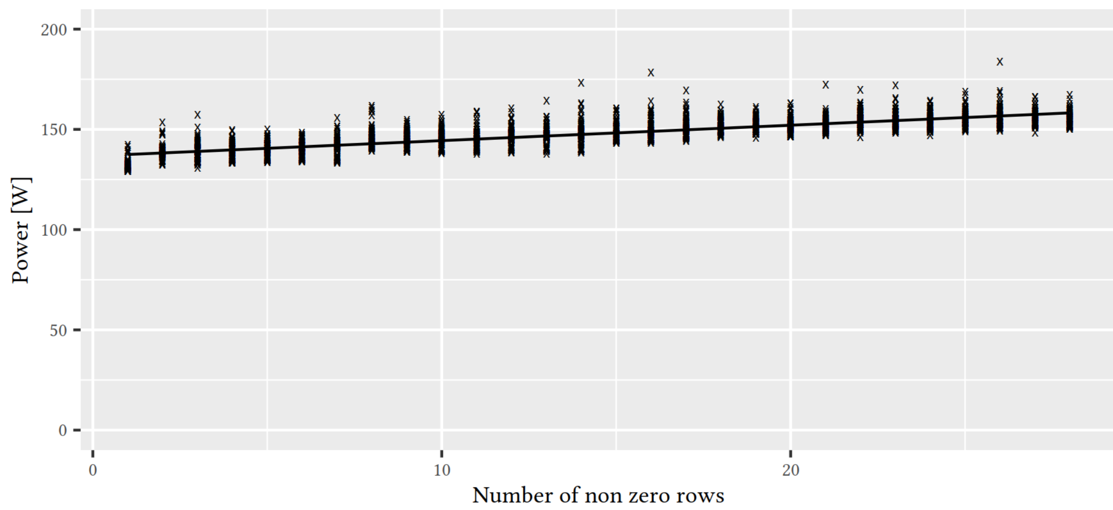


Figure 2.3: The power draw of matrix-vector multiplication using 28 threads is linearly increasing in the number of non-zero rows.

3 Energy Measurement in AutoPas

3.1 Implementation

To enable energy monitoring and tuning in AutoPas the new utility class `RaplMeter`, which performs the actual measurement, was first implemented. On initialization, the class determines the available RAPL domains and reads their configuration. A file descriptor for each RAPL domain is opened through the perf event interface. When `RaplMeter::sample()` is called, these file descriptors are read, which returns the energy consumed in the corresponding RAPL domain. The energy in joules for each RAPL domain can then be accessed using getter functions. Additionally, the function `get_total_energy()` returns the best possible estimate for the total energy consumption of the system using all available domains.

This new class is instantiated in `AutoTuner` and the energy consumption during each iteration is measured. The result is logged using `IterationLogger`. Additionally, the new configuration option `TuningMetric` was added, which allows the choices of time or energy. If time is chosen, then `AutoTuner` functions as before and selects the configuration with the least time to completion. If energy is chosen, it instead selects the configuration with the least energy consumption.

3.2 Effect of Configuration Options on Energy Usage

We measure the per iteration energy usage as well as the time to completion during the simulation of the classic molecular dynamics scenario “Spinodal Decomposition”.

3.2.1 Experiment Setup

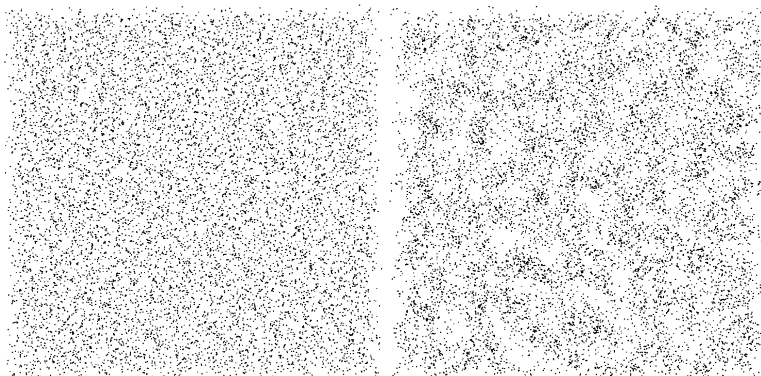


Figure 3.1: The particles are distributed uniformly at the beginning of the spinodal decomposition simulation (left) and have condensed into clusters after 30k iterations (right).

A cuboid domain is filled with a $640 \times 80 \times 80$ grid of identical particles. In the first step, the simulation is run for an extended time at a constant high temperature until an equilibrium state is reached. Then, the temperature is dropped over the course of several iterations. This causes the particles to condense into clusters, resulting in an inhomogeneous domain. **??** shows a slice through the domain before and after this step.

3.2.2 Results from Equilibration Step

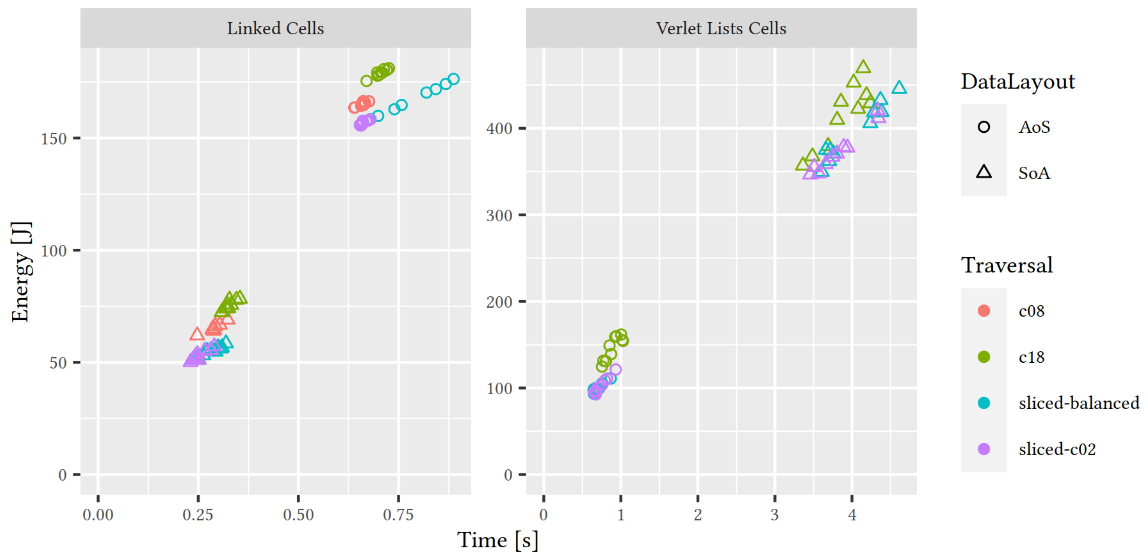


Figure 3.2: Energy consumption and time to completion per iteration during the equilibration step. The data points clearly cluster into groups according to the data layout used. Each data point represents an average of 10 iterations of the same configuration, to eliminate variation due to rebuilding neighbor lists.

As the macroscopic state of the domain stays constant for the majority of the equilibration step, we can use the measurements from this simulation to get a general overview of how different configurations compare in terms of energy usage and time to completion. **??** plots the per iteration energy usage against its time to completion. Each data point represents an average of 10 iterations. This is done as the neighbor lists for the verlet lists cell container are rebuilt in every 10th iteration, causing those iterations to take significantly more time and use more energy. For both containers, the choice of data layout has a far greater effect on energy usage and time than the choice of traversal algorithm. We can also see that the colored traversals (c08, c18) lie on lines with a steeper slope in the diagram compared to the sliced traversals. This corresponds to a higher power draw. In **??** we identified two factors that increase power draw: the number of threads, and the computational load of each thread. As the number of threads is identical for each traversal algorithm, the likely explanation is that the colored traversals have a lower amount of idle thread time (ie. greater load) than the sliced traversals. This makes sense, as the colored traversals dynamically schedule a large number of small regions among the threads, which allows the load to be distributed very evenly. On the other hand, sliced-balanced has a fixed region assigned to each thread, which may not all have equal load. In this case, some slices were 8 cells thick, while others were only 7 cells thick,

resulting in a slight imbalance even for a homogeneous domain. While sliced-c02 does in fact use dynamic scheduling, the regions are much larger than in the colored traversals, so the potential load difference is also greater.

3.2.3 Results from Spinodal Decomposition

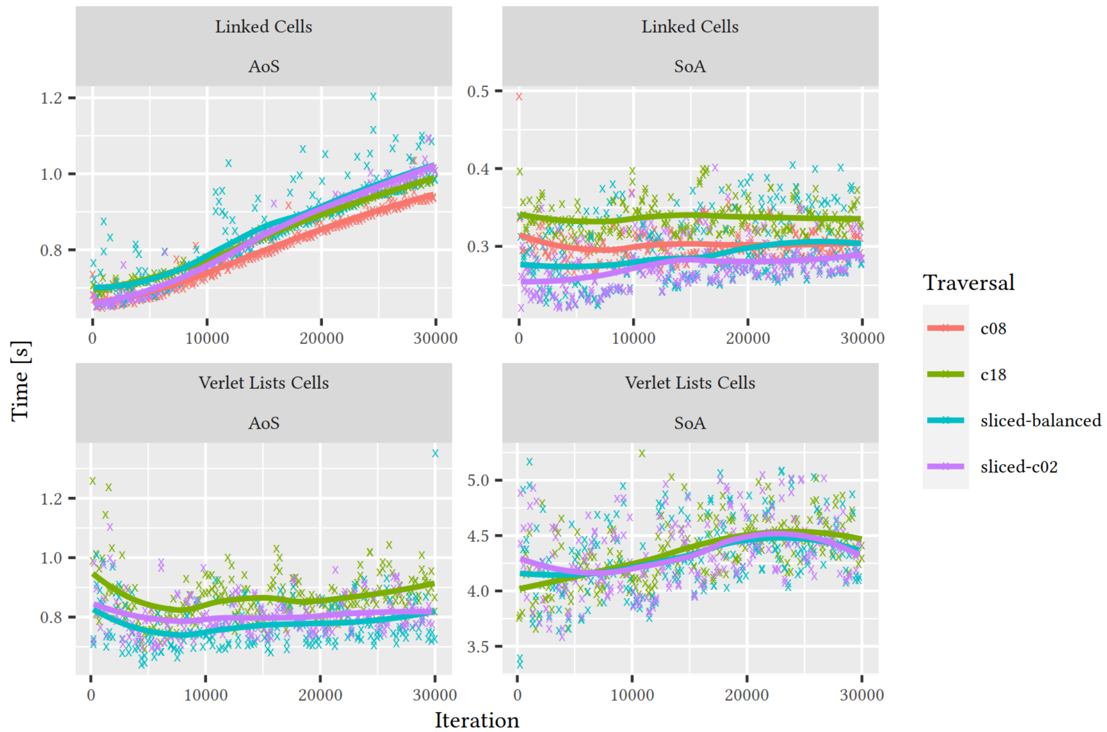


Figure 3.3: Time to completion per iteration during spinodal decomposition simulation. The time increases or at least stays constant during the course of the simulation for most configurations. Each data point represents an average of 10 iterations using the same configuration. The lines were fitted using the LOESS local regression method.

?? depicts the time to completion for each iteration of the spinodal decomposition. The first thing to note is that, generally, this time increases or at the very least stays constant for most configurations. This is to be expected, as when the particles condense to form clusters, the forces between more particle pairs need to be calculated, as their distance drops below the cutoff.

Taking a closer look at the linked cells container in the AoS case, we see that in the beginning, the difference between the various traversals is quite small, but as the simulation progresses the two colored traversals, especially c08, gain a slight advantage over the sliced traversals. A possible explanation for this is that the dynamic scheduling of many small regions (the cells covered by one base step) is better able to balance the computational load among all threads in an inhomogeneous domain than the static scheduling used with sliced-balanced and even the dynamic scheduling of larger regions as in sliced-c02. This effect has been described in [1]. The smaller number of colors of c08 (and thus smaller base step) gives it a slight advantage over c18 for the same reason. In the SoA case, we can observe a similar effect. While the colored traversals are still slower than the

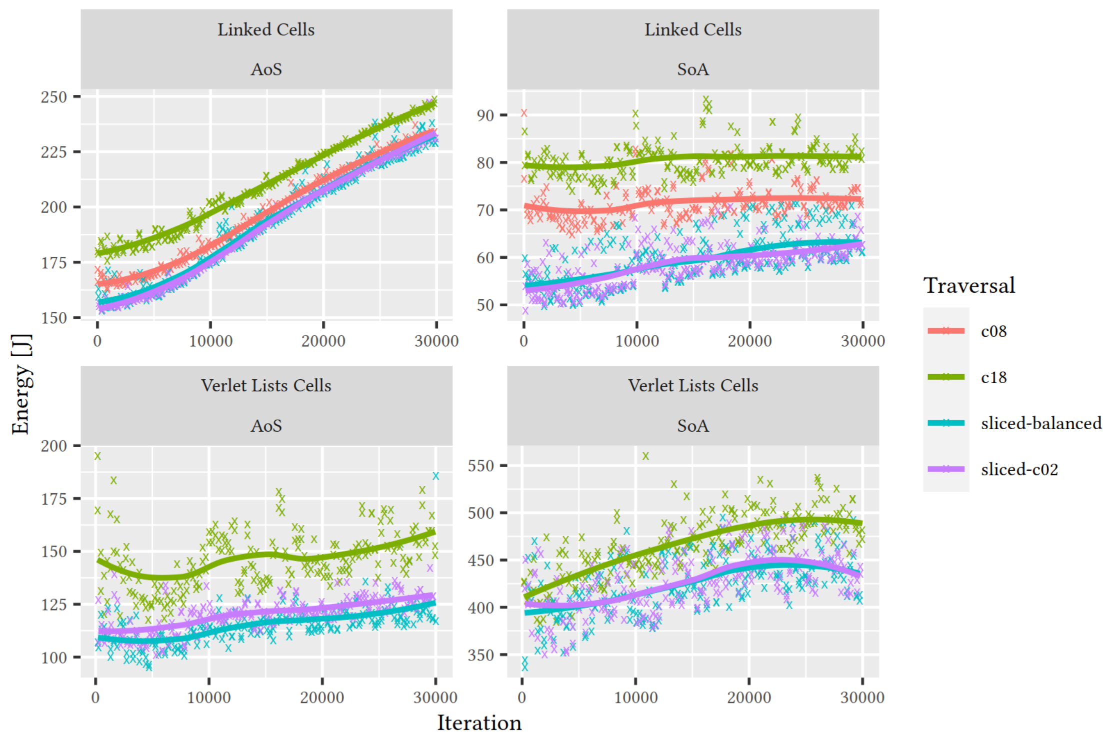


Figure 3.4: Energy usage per iteration during the spinodal decomposition simulation. Colored traversals consistently use more energy than sliced traversals for both containers and both data layouts. Each data point represents an average of 10 iterations using the same configuration. The lines were fitted using the LOESS local regression method.

sliced traversals by the end of the simulation, the gap is smaller than at the beginning. The sliced traversals may have a larger advantage over the colored traversals when compared to the AoS case, as the order in which cells are processed in sliced traversals allows for more efficient memory access patterns. In the colored traversals, each thread gets a new region scheduled after each base step, which is very bad for cache locality.

If we compare these results with the energy usage per iteration as depicted in **??**, we see a stark contrast. Here, the sliced traversals consistently outperform the colored traversals. This is consistent with the higher power draw we saw for colored traversals in **??**. Only c08 manages to match the performance during the end of the simulation in the AoS case, likely simply due to the faster time to completion. In fact, for each configuration, the power draw varies very little over the course of the simulation as can be seen in **??**. The maximum relative standard deviation of 7.3% occurs for the configuration (linked cells, sliced-balanced, SoA).

This same contrast also exists for the verlet lists cells container. C18 takes on average 7% more time compared to sliced-c02 in the AoS case but 21% more energy (12% more time and 27% more energy compared to sliced-balanced). In the SoA case, the difference in time to completion is insignificant compared to the variation for each configuration itself, but the energy usage for c18 is 9% higher than for sliced-c02 (10% higher than sliced-balanced).

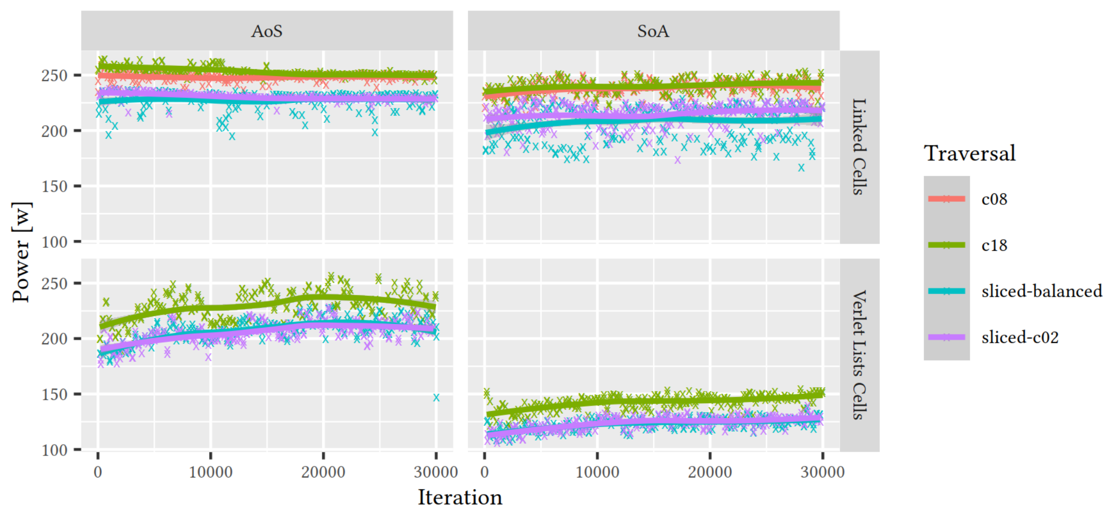


Figure 3.5: Power draw per iteration stays constant during the course of the spinodal decomposition. Each data point represents an average of 10 iterations using the same configuration. The lines were fitted using the LOESS local regression method.

4 Conclusion and Future Work

The results from **??** clearly show that the power draw while running a molecular dynamics simulation is dependant on the specific algorithm used, with data layout and parallelization strategy playing an important role. This difference in power draw means that faster traversal algorithms do not necessarily use less energy than slower ones. When getting results as fast as possible is not a priority, it therefore makes sense to optimize for energy usage instead. The new tuning metric option implemented in AutoPas makes this easy to do.

The results from **??** also suggest some options to optimize existing traversals for energy efficiency. For example, in the sliced-balanced traversals, the number of threads could be reduced if the layers can not be evenly distributed among them. As the time to solution only depends on the slowest thread, this should not drastically increase the time, but it would reduce the power draw.

The current implementation is specific to Intel CPUs. AMD provides a similar interface that could also be supported in the future.

Bibliography

[1] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann, "Autopas: Auto-tuning for particle simulations," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2019, pp. 748–757.

[2] *Intel® 64 and ia-32 architectures software developer's manual volume 3*, Intel Corporation, 2023-03. [Online]. Available: `https://cdrdv2.intel.com/v1/dl/getContent/671427` (visited on 2023-03-29).

[3] "Power capping framework." (2020), [Online]. Available: `https://www.kernel.org/doc/html/latest/power/powercap/powercap.html` (visited on 2023-03-29).

[4] S. Eranian. "[patch v7 0/4] perf/x86: Add intel rapl pmu support." (2013), [Online]. Available: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4788e5b4b2338f85fa42a712a182d8afd65d7c58` (visited on 2023-03-29).