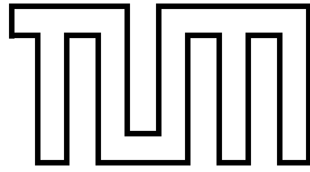


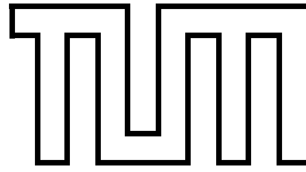
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Energy-Efficient Molecular Dynamics Simulations: Implementing Hardware-Agnostic Energy Measurement and Evaluating Runtime-Energy Tradeoffs

Maximilian Praus



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

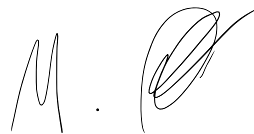
Bachelor's Thesis in Informatics

Energy-Efficient Molecular Dynamics Simulations: Implementing Hardware-Agnostic Energy Measurement and Evaluating Runtime-Energy Tradeoffs

Energieeffiziente Molekulardynamik-Simulationen: Implementierung hardwareunabhängiger Energiemessungen und Bewertung von Laufzeit-Energie-Tradeoffs

Author:	Maximilian Praus
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Manish Mishra, M.Sc.
Date:	12.10.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.



Munich, 12.10.2024 Maximilian Praus

Acknowledgements

First of all, I would like to thank my advisor, Manish Mishra, who immensely supported me throughout this thesis with his excellent knowledge in the field of molecular dynamic simulations. Talking with him about simulation results and upcoming problems helped me get a new viewing angle and possible solutions. I can recommend that everyone who is interested in the field of md-simulations and high performance computing write a thesis advised by him. Secondly, I would like to thank my wonderful girlfriend, who always motivated me, especially at the end of writing this thesis. Also, a huge thanks to Martin Rose, who was so kind as to run simulations on the AMD cluster to compare AMD and INTEL CPUs.

Abstract

The cost of energy is a substantial part of the overall costs of simulations in high-performance computing (HPC). Reducing energy consumption leads to reduced costs and correlates to lower greenhouse emissions. With climate change and increasing costs for energy as pressing problems right now, many HPC centers and companies strongly emphasize reducing energy consumption. Those energy optimizations are carried out at different levels, like switching to a sustainable energy source for computing and minimizing heat loss. This study focuses on node-level energy optimization by selecting the most energy-efficient algorithm. Achieving this requires integrating a portable energy measurement method into the molecular dynamics simulation framework. Therefore, we introduce the software library "PMT: Power measurement toolkit" and how it can be used in the node-level auto-tuned particle simulation library AutoPas to ensure an accurate way of measuring energy consumption on different hardware platforms. Additionally, we will show that in most cases, a reduced runtime comes hand in hand with a reduction in energy consumption. Finally, we present new energy metrics that can be used to compare performance across different hardware and highlight a case study between AMD and INTEL-based clusters.

Zusammenfassung

Die Energiekosten machen einen erheblichen Teil der Gesamtkosten von Simulationen im High-Performance Computing (HPC) aus. Eine Reduzierung des Energieverbrauchs führt zu einer Senkung der Kosten und korreliert zudem mit geringeren Treibhausgasemissionen. Angesichts des Klimawandels und steigender Energiekosten als drängende Probleme legen viele HPC-Zentren und Unternehmen großen Wert auf die Reduzierung des Energieverbrauchs. Diese Energieoptimierungen werden auf verschiedenen Ebenen durchgeführt, wie z.B. der Umstellung auf nachhaltige Energiequellen für das Computing oder der Minimierung von Wärmeverlusten. In dieser Studie konzentrieren wir uns auf die Optimierung des Energieverbrauchs auf Knotenebene, indem der energieeffizienteste Algorithmus ausgewählt wird. Dies erfordert die Integration einer portablen Methode zur Energieverbrauchsmessung in ein Molekulardynamik-Simulationsframework. Daher stellen wir die Softwarebibliothek "PMT: Power Measurement Toolkit" vor und erläutern, wie sie in die Knotenebene der auto-tuneden Partikelsimulationsbibliothek AutoPas integriert werden kann, um eine genaue Messung des Energieverbrauchs auf verschiedenen Hardwareplattformen zu gewährleisten. Zusätzlich werden wir zeigen, dass in den meisten Fällen eine Reduzierung der Laufzeit mit einer Verringerung des Energieverbrauchs einhergeht. Abschließend präsentieren wir neue Energiemetriken, die verwendet werden können, um die Leistung auf verschiedenen Hardwareplattformen zu vergleichen, und heben eine Fallstudie zwischen AMD- und INTEL-basierten Clustern hervor.

Contents

Acknowledgements **vii**

Abstract **ix**

Zusammenfassung **x**

1 Introduction **1**

2 AutoPas **3**
2.1 Containers 3
2.1.1 Linked Cells 4
2.1.2 Verlet Lists Cells 4
2.2 Data layouts 5
2.3 Newton's Third Law of Motion 5
2.4 Traversal 6
2.4.1 Base steps 6
2.4.2 Linked Cells Traversals 7
2.4.3 Verlet Lists Cells Traversals 8

3 PMT: Power measurement toolkit **9**
3.1 Structure 9
3.2 Implementation 10
3.3 RAPL power measurement backend 11
3.3.1 RAPL: Running Average Power Limit 11
3.3.2 Implementation in PMT 12
3.4 Overhead 13

4 Implementation **14**
4.1 Compiling PMT 14
4.2 Integrating PMT in AutoPas 14

5 Results **17**
5.1 Time to completion 17
5.1.1 Results before PMT modification 17
5.1.2 Adapting PMT source code 18
5.1.3 Results after PMT modification 19
5.2 Energy consumption measurement 20
5.3 Comparing tuning configurations 23
5.4 Recommendations 24

6 Benchmarks 25
6.1 Tuning configurations . . . 25
6.1.1 Comparing . . . 25
6.1.2 Metrics . . . 28
6.1.3 Energy Efficiency . . . 32
6.2 Tuning for energy efficiency . . . 34
6.3 Performance comparison across different architecture . . . 37

7 Conclusion 43
7.1 Future scope . . . 43

8 Appendix 45
8.1 Appendix A: Simulation file for exploding liquid . . . 45

Bibliography 50

1 Introduction

Molecular dynamics simulations are increasingly important in many scientific fields, including biology, materials science, and chemistry. In particular, for drug discovery and protein engineering applications, molecular dynamics simulations can "capture the behavior of proteins and other biomolecules in full atomic detail and at very fine temporal resolution" [7]. As these simulations typically have time steps in the region of femtoseconds and take place over nanoseconds or longer with millions of particles, this can become quite computationally demanding. As the computational power required for these simulations increases, so does energy consumption.
For instance, high-performance computing centers running these simulations consume vast energy. To illustrate this further, we calculate the energy costs for the simulations run in this study. In total, our simulations consumed approximately 151.17 MJ. As 3.6 MJ are equal to 1 kWh, we get:

$$\frac{151.17\,\text{MJ}}{3.6} \approx 41.99\,\text{kWh}$$

At the current electricity price in Munich of 0.3951 €/kWh, the total electricity costs amount to:

$$41.99\,\text{kWh} \times 0.3951\,\text{€/kWh} \approx 16.59\,\text{€}$$

Overall, our simulations consumed 464.33 core hours. This is only a fraction of the overall hours on the LRZ system, which had 452, 417, 767 in 2022 [13]. Setting our simulation time in relation gives us an idea of how much energy the HPC Center of LRZ consumes per year. To address the challenge of high energy consumption in molecular dynamics simulations, this thesis introduces "PMT: Power Measurement Toolkit", a C++ library used to measure energy consumption on various hardware. Integrating PMT in AutoPas provides a new way of measuring energy consumption without being hardware dependent, like the previously used implementation [3]. With PMT, molecular dynamics simulations in AutoPas can be tuned to use the most energy-efficient configuration, mitigating the increasing energy consumption of simulations [1].
Chapter 3 will provide a brief overview of PMT and its functionality. In Chapter 4, we will integrate PMT in AutoPas to measure energy consumption values and tune for energy efficiency. After that, in Chapter 5, we will compare the overhead and energy consumption values between PMT and the previously used implementation. We also introduce modifications to the PMT implementation for more accurate energy measurement, as this is critical for our application, where we need measurement at the iteration level, and any error can accumulate over time.

[1] Tuning can be quite energy expensive, as shown in [3]. For the present study, we consider tuning to be advantageous

Finally, in Chapter 6, we will compare different commonly used simulation configurations for runtime and energy consumption and calculate some essential energy and runtime metrics. With these results, we will establish and prove a thesis for the relationship between running time and energy efficiency. This will demonstrate how our approach contributes to more energy-efficient, sustainable, cost-effective molecular dynamics simulations.

2 AutoPas

AutoPas is an open-source C++ node-level performance library that aims to provide a base for arbitrary N-body simulations. It acts as a black-box data container, providing interfaces for accessing particles and applying short-range pairwise forces [5]. AutoPas employs dynamic auto-tuning to select the optimal algorithmic configuration to reduce the time to solution or energy consumption of molecular dynamic simulations. Therefore, the library has a broad portfolio of traversal algorithms, data layouts, and containers [4]. In the following subsections, we will provide a quick overview of crucial molecular dynamic algorithms that impact the energy consumption of simulations.

2.1 Containers

Containers are managing in which data structures particles are stored and how neighboring particles are identified. AutoPas provides several container options:

- **Direct Sum:** Calculating the distance between all particles. Forces are only applied for particles within a specific range (cutoff radius) [5].
- **Linked Cells:** Dividing domain into cells bigger or equal to the cutoff radius. Distance is only calculated for particles in the same or neighboring cells [5].
- **Verlet Lists:** Pre-compute distance relations and store all relevant neighboring particles in Verlet List. These lists must be rebuilt if a particle comes in range or leaves it [5].
- **Var Verlet Lists:** Generalization of Verlet Lists container. Provides interface to easily swap out the implementation of Verlet Lists and their generation [4].
- **Verlet Lists Cells:** Solving a problem that Verlet Lists are stored without any information about a spatial locality. This is done by associating the neighbor list with the cell where the particle is stored in [4].
- **Verlet Cluster Lists:** Building upon the Verlet Lists approach.A given number of M particles are combined into a cluster and thus reduces the number of lists by $\frac{1}{M}$. This is because neighboring particles will have very similar Verlet Lists [4].

The choice of used container can significantly impact the energy consumption of a molecular dynamics simulation. Based on the amount/distribution of particles and the size of the domain, some containers can achieve a faster runtime or reduce the amount of memory access, leading to lower energy consumption. In this thesis, we will take a closer look at the Linked Cells and Verlet Lists Cells container and their impact on energy consumption [4].

2.1.1 Linked Cells

The Linked Cells container stores uniform cells in a vector. All particles that are part of the cell are stored in a vector. Figure 2.1 visualizes how neighboring particles are identified within this structure. The forces currently applied to the red particle are calculated in the figure. The red circle illustrates the cutoff radius, defining the range within which particle interactions are considered. All particles in the same or neighboring cell are marked with grey. Distance is calculated for every particle connected by an arrow to the red one. However, force calculations are only performed for particles shown in blue, which are both in the same or neighboring cells and within the cutoff radius of the red particle.

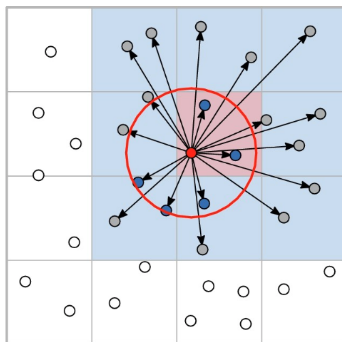


Figure 2.1: Linked Cells container force calculation [4]

Through this approach, neighboring particles that are close in space end up in the same cell and are also close in memory, which increases efficiency in loading neighboring particles. This allows for an efficient iteration of cells in the same region and vectorization of particle computations. A disadvantage is that the Linked Cells container represents the actual space, not particle relations. Moving particles need to be stored in the correct cell adding a overhead. Additionally, Linked Cells has a high number of superfluous distance calculations, caused by poor approximation of the cut-off radius [4].

2.1.2 Verlet Lists Cells

The Verlet Lists Cells approach enhances the traditional Verlet Lists method used for managing particle interactions in simulations. The traditional Verlet Lists approach generates a neighbor list for each particle, which contains a reference to all particles within a specified cutoff range. As building this neighbor list can be computationally expensive, the list should be rebuilt as infrequently as possible. That is why particles slightly outside the cutoff range are stored in the neighbor list as well. This additional range is referred to as verlet-skin. Figure 2.2 visualizes this approach. Currently, applying forces are calculated for the red-marked particle. The red circle illustrates the cutoff radius in which particle interactions are considered. The yellow circle marks the verlet-skin added to the cutoff radius. Distance is calculated for all particles connected by an arrow with the red one, but force calculations are only performed for the blue particles inside the cutoff radius. The neighbor list contains all particles within the verlet skin circle.

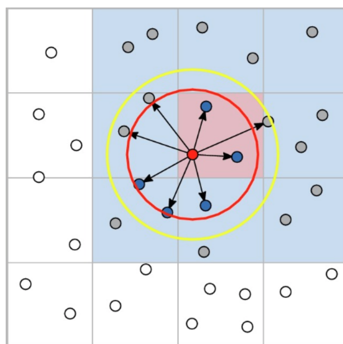


Figure 2.2: Verlet Lists container [4]

To efficiently build these neighbor lists, the particles are stored in an instance of Linked Cells. The neighbor list itself stores the particles in a vector of pointers. All neighbor lists are then collected in a map of particle pointers, each pointer mapping to a neighbor list. One disadvantage of the Verlet Lists approach is that no information about the spatial locality of a neighboring particle is stored. The Verlet Lists Cells tries to improve this by not saving all lists in a container-wide vector but associating them to the cell, where the particle is stored [4].

2.2 Data layouts

Data layout defines how particles are stored within a container during molecular dynamic simulations. There are two major ways to store those particles, which are both supported by AutoPas:

1. **Array of Structures (AoS):**
   In this data layout, particle information is stored within an encapsulated C++ object containing all important properties, e.g., velocity, positions, and forces. These objects are then stored within a `std::vector<Particle>`
2. **Structure of Arrays (SoA):**
   This data layout does use a `std::vector<double>` for each property like the y-position. Each entry represents one particle.

Each of the presented data layouts has different advantages. When accessing one particle, the AoS data layout is simpler, as it is only a random access on a vector. In contrast, for the SoA layout, each piece of information has to be gathered from each vector separately. In contrast, retrieving successive particle information can be done within one load for the SoA data layout [4].

2.3 Newton's Third Law of Motion

One important optimization that is used in molecular dynamics simulations is Newton 3 optimization. Newton's third law states "that for every force exerted on a body i by a body j, there must be a force of equal magnitude but opposing direction on body j" [15]. This is

of significant interest for molecular dynamics simulations as it holds $F_{ij} = -F_{ji}$, and we can reduce the amount of force and distance calculations by half.

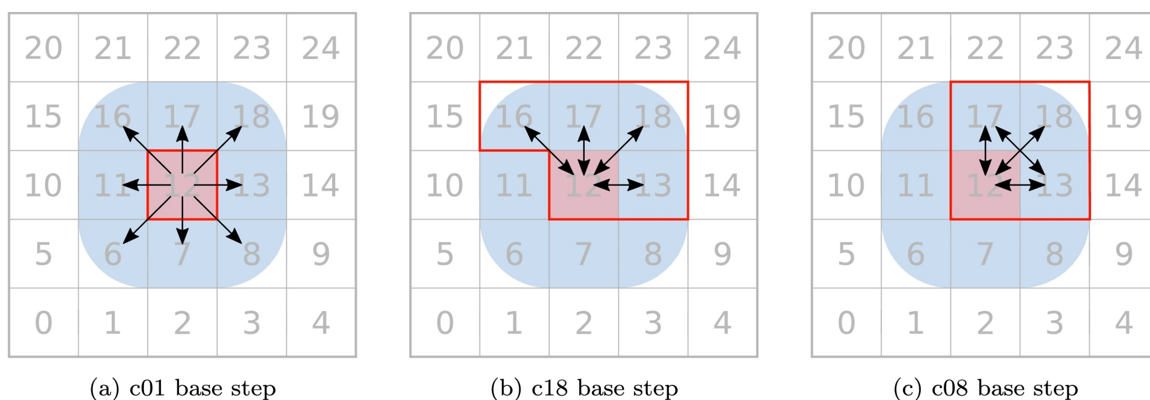
2.4 Traversal

The Linked Cells and Verlet Lists Cells containers both provide different algorithms for the traversal of the domain. To get a general understanding of how different algorithms work and what the benefits and disadvantages are, we need to introduce the base steps supported by AutoPas and that are used by most of the traversal algorithms.

2.4.1 Base steps

AutoPas currently supports three base steps, which are used for traversing neighboring cells.

1. **c01:** The c01 base step does calculate the applying forces for each neighboring cell. This can be seen in Figure 2.3 a. As we calculate the force for each cell individually and no race conditions have to be taken care of, this base step can be highly parallelized. One disadvantage is that the Newton 3 optimization, which could otherwise halve the number of particle interactions (see Figure 2.3 a)) is not utilized for this base step.
2. **c18:** This base step only computes the interaction with a cell index, which is higher than the currently iterated cell. As c18 makes use of Newton 3 optimization, as can be seen in Figure 2.3 and the box of cells increases, in which we update particles, we need to guard these cells against race conditions. However, as the number of cells that are needed for one base cell decreases, it is more likely that cells remain in the lower-level cache for the entire base step.
3. **c08:** The c08 base step improves the idea of the c18 base step further. This step replaces the forward diagonal interaction with the forward computation of the next cell. This idea can be seen in Figure 2.3 c. Here, the interaction between 16 and 12 is replaced by forward computation for 13 and 17. This increases the level of parallelism as the box size decreases compared to c08, and it also improves cache reuse efficiency.



(a) c01 base step (b) c18 base step (c) c08 base step

Figure 2.3: Base steps [4]

Depending on the various properties of the base steps, the energy consumption also varies. Important properties are, for example, parallelizability, use of Newton3 optimization, and box size.

2.4.2 Linked Cells Traversals

In the following subsection, we will introduce different traversal algorithms for the Linked Cells container.

1. **lc-c18:**
   The lc-c18 traversal algorithm is based on the c18 base step, described in 2.4.1. This algorithm takes Newton 3 optimization into account and reduces the interactions that we need to calculate by 50%.
2. **lc-c08:**
   The lc-c08 traversal algorithm makes use of the c08 base step, as the name suggests. As described in 2.4.1, this algorithm divides the domain into smaller pieces than lc-c18 and increases the level of parallelism.
3. **lc-sliced-balanced:**
   In contrast to the previously presented algorithms, sliced traversals reduce the scheduling overhead to a minimum. These traversals assign cells statically by dividing the entire domain into slices of equal size, as seen in Figure 2.4. To spread the computational load, each slice is assigned to one thread [5]. The sliced-balanced algorithm creates exactly one slice for each thread. It can use a heuristic to estimate the computational load of each slice and create slices with equal computational load. To ensure no data races occur, each thread locks the neighboring cells of the layer it is currently processing. [3].
4. **lc-sliced-c02:**
   The lc-sliced-c02 traversal does split up the domain just like lc-sliced-balanced. The main difference between the two algorithms is that lc-sliced-c02 creates as many slices as possible of the domain and uses dynamic scheduling to spread the computational load among all threads. Similar to the lc-sliced-balanced algorithm, each thread locks neighboring cells to prevent data races. [3].

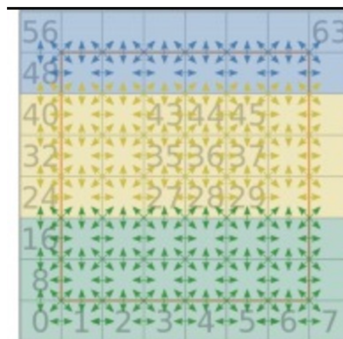


Figure 2.4: Sliced domain [4]

2.4.3 Verlet Lists Cells Traversals

In the following subsection, we will introduce different traversal algorithms for the Verlet Lists Cells container.

1. **vlc-c18:**
   This traversal algorithm processes all neighbor lists of a cell with the c18 base step. All neighbor lists of the currently iterated cell are processed. When creating the neighbor list, the vlc-c18 algorithm creates a list of forward-facing neighbors. When processing particle pair P_i and P_j with $i < j$, with different cells i and j, P_j is part of the neighbour list of P_i but not the other way round [4].
2. **vlc-sliced-balanced:**
   The vlc-sliced-balanced algorithm works the same way as lc-sliced-balanced, for each thread dividing the domain into equal slices. Neighbour lists are created just like in the vlc-c18 algorithm [4].
3. **vlc-sliced-c02:**
   This traversal algorithm also works the same way as lc-sliced-c02, creating as many slices of the domain as possible and dynamically scheduling them among all threads. Neighbour lists are made just like in the vlc-c18 algorithm [4].

3 PMT: Power measurement toolkit

After laying the foundation with some theoretical background in molecular dynamics simulations, we want to introduce the software library "PMT: Power Measurement Toolkit" and, in particular, the RAPL backend, which is the sole power measurement backend utilized in this thesis.
"PMT: Power Measurement Toolkit" is a high-level software library that collects power consumption measurements on various hardware. The library is written in C++ and is Linux-only. It provides a common interface to measure the energy consumption of CPUs and GPUs on different hardware [19].

3.1 Structure

PMT uses a base class to implement all available functions and attributes to get this common interface between the different power measurement backends. All additional subdirectory classes inherit this base class and implement hardware-specific functions. Additionally, there is a dummy subdirectory that is used as a fallback in case a sensor with an invalid or not compiled energy measurement backend is used. This ensures that PMT does not fail during runtime [18]. Figure 3.1 visualizes the structure of the PMT library. Important to note is that not all sub-directories are displayed.

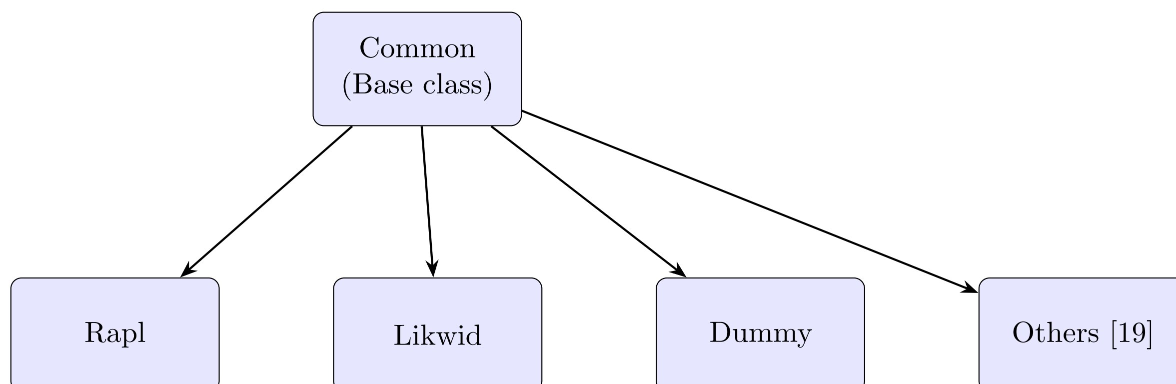


Figure 3.1: General structure of the "PMT: Power Measurement Toolkit" library

Each subdirectory provides functionality to measure the energy consumption on different hardware; for example, does the "Rapl" power measurement backend access energy consumption values on AMD and INTEL hardware. To decide which energy measurement backend should be included in the current build, PMT defines several CMake options for managing the build of each subdirectory.

3.2 Implementation

In contrast to the previously used RaplMeter implementation, PMT measures energy consumption asynchronously. When setting up a new energy sensor, e.g., with `std::unique ptr<pmt::PMT> sensor(pmt::Create("rapl"))` a new instance of `pmt::PMT` is created with RAPL power measurement backend in this case. To understand the difference between PMT and RaplMeter implementation, we will examine two methods `PMT::Read()` and `PMT::StartThread()` in the PMT base class `pmt::PMT`.

PMT::Read()

```
1  State PMT::Read() {
2    const int measurement_interval = GetMeasurementInterval();
3    if (!thread_started_) {
4      StartThread();
5      thread_started_ = true;
6      std::this_thread::sleep_for(std::chrono::milliseconds(
   measurement_interval));
7    }
8    while (seconds(state_previous_, state_latest_) == 0) {
9      std::this_thread::sleep_for(std::chrono::milliseconds(
   measurement_interval));
10   }
11   state_previous_ = state_latest_;
12   return state_latest_;
13 }
```

Listing 3.1: PMT::Read() method for retrieving latest energy consumption values

When calling `sensor.Read()` for the first time, a new thread is created that reads power consumption values at a specific interval asynchronously. If a thread was already started, it is tracked with the `thread_started_` boolean variable. The actual energy consumption values are returned within a custom object `State`, which can be further used to calculate the energy consumption between two timestamps. Note that calling `sensor.Read()` twice in a short time range will cause the thread currently executing the method to sleep for a specified time interval, based on the energy consumption backend used. This behavior will be relevant in Section 5.1.

PMT::StartThread()

```
1  void PMT::StartThread() {
2    thread_ = std::thread([&] {
3      const State start = GetState();
4      assert(start.nr_measurements_ > 0);
5      State previous = start;
6      state_latest_ = start;
7
8      if (dump_file_) {
9        DumpHeader(start);
10     }
11
12     const int measurement_interval =
```

```
13        GetMeasurementInterval();  // in milliseconds
14        assert(measurement_interval > 0);
15        const float dumpInterval = GetDumpInterval();  // in seconds
16        assert(dumpInterval > 0);
17
18        while (!thread_stop_) {
19          std::this_thread::sleep_for(
20          std::chrono::milliseconds(measurement_interval));
21          state_latest_ = GetState();
22
23          const float duration = seconds(previous, state_latest_);
24          if (dump_file_ && duration > dumpInterval) {
25            Dump(start, previous, state_latest_);
26            previous = state_latest_;
27          }
28        }
29      });
30    }
```

Listing 3.2: PMT::StartThread() method, spinning up a second thread for power measurement backend

The `PMT::StartThread()` method is used to start a new thread for a sensor, which reads the energy consumption values asynchronously at a specified time interval. This interval is defined by each power measurement backend. For RAPL, it is, e.g., 100 ms. PMT provides an additional `dump-mode`, which writes all measured values within a specified interval to a file. This part should be ignored because we will use measurement values within AutoPas internal log iteration. When starting a new thread, the initial state is obtained by `GetState()`, and we assure that the start state has at least one measurement. After that, the initial state is used to set `state_latest` and to create a new copy of the state named previous. In the next step, we obtain the measurement interval and enter a while loop to measure energy consumption values continuously. The thread sleeps for the specified time interval and updates the state.

3.3 RAPL power measurement backend

Since this thesis will exclusively use the RAPL power measurement backend, we will provide a brief overview of RAPL and its implementation in PMT.

3.3.1 RAPL: Running Average Power Limit

Intel's Running Average Power Limit (RAPL) was introduced in the Intel Sandy Bridge Architecture [12]. It allows for measuring energy consumption across different power planes with high precision and sampling rate [12]. PMT makes use of this interface in the similarly named power measurement backend. To obtain the consumed energy for different power planes, PMT accesses two files in the RAPL interface, `max_energy_range_uj` and `energy_uj`. These files were introduced as part of the `powercap` interface in the Linux kernel 3.13 release, providing a consistent way of reading energy consumption values. The files in `/sys/class/powercap/intel-rapl/` are organized to reflect the hierarchy of different

power planes. For each CPU socket, powercap contains a separate sub directory, e.g., `/sys/class/powercap/intel-rapl/0:intel-rapl` for the first CPU socket [22]. Figure 3.2 is an example of the powercap directory on CoolMUC-2, where all simulations for this study were conducted.

```
__ intel-rapl:0
|__ energy_uj
|__ intel-rapl:0:0
|__ max_energy_range_uj
__ intel-rapl:1
|__ energy_uj
|__ intel-rapl:1:0
|__ max_energy_range_uj

```

Figure 3.2: Content of powercap directory on CoolMuc-2

Each of those directories contains its own `energy_uj` and `max_energy_range_uj`, which is updated approximately every 1ms and reports the current energy consumption of the power plane and the maximum energy range that can be captured [2].

3.3.2 Implementation in PMT

PMT makes use of RAPL and the powercap interface by reading energy consumption values from `energy_uj` and `max_energy_range_uj`. When creating a new PMT instance, with RAPL as power measurement backend, the CPU topology is read with `GetActiveCPUs()` which returns an `std::vector<int>` containing all active CPU numbers. After that, the vector is used to gather the physical package ID for each CPU and iterate over the resulting `std::vector<int> package_ids` and the subdomains in each package.

```
// Read domain name
const std::string filename_domain = basename + "/name";
std::string domain_name;
bool valid = ReadFile(filename_domain, domain_name);
if (domain > 0) {
  domain_name = domain_name + "-" + std::to_string(
 package_id);
}
```

Listing 3.3: Retrieving domain names from powercap interface

For each package and domain, we read the name in the powercap `/name` file and save it in the local variable `domain_name`

```
// Read max energy
const std::string filename_max_energy = basename + "/
 max_energy_range_uj";
size_t max_energy_range_uj = 0;
if (valid) {
```

```
5     valid &= ReadFile(filename_max_energy, max_energy_range_uj
   );
6   }
```

Listing 3.4: Reading max energy, that can be stored for each domain

We save the max energy range as well by reading the `max_energy_range_uj` file and saving the content in a local variable.

```
1   // Read energy
2   const std::string filename_energy = basename + "/energy_uj";
3   if (valid) {
4     size_t energy_uj = 0;
5     valid &= ReadFile(filename_energy, energy_uj);
6   }
```

Listing 3.5: Reading energy consumption values for each domain

Additionally, we need to save the actual consumed energy by saving the content of `energy_uj` file.

```
1   if (valid) {
2     domain_names_.push_back(domain_name);
3     uj_max_.push_back(max_energy_range_uj);
4     file_names_.push_back(filename_energy);
5   }
6 }
7 }
```

Listing 3.6: Saving domain name, max energy and path to energy file in according vectors

If all these operations were successful, we save the domain name, max energy range, and path to the energy_uj file in the corresponding vector.
To obtain the actual energy consumption values, the RAPL backend does provide `State RaplImpl::GetState()`. The method uses the helper function `GetMeasurements()` to process all packages and domains. The helper functions iterate over all the previously saved file names and domain names, reading the energy values from each `energy_uj` file. The vector is then passed to the actual `GetState()` method, which calculates the duration and energy (in Joules) consumed. Lastly, those values are saved in a State and passed to the caller.
In general, we must note that the asynchronous measurement method occupies two threads. Especially for molecular dynamics simulation, where the main focus is to exploit all the available parallelism for computation, this is a disadvantage.

3.4 Overhead

According to "PMT: Power measurement toolkit"[19], PMT adds an overhead of 1 ms per iteration in C++. In Section 5.1, we will analyze the overhead that is added by PMT more accurately and see why PMT, without any code changes, might not be suitable for analyzing the power consumption of short code paths.

4 Implementation

Integrating PMT in AutoPas follows a similar approach to other already included libraries like spdlog and googletest. The implementation can be divided into two main stages: compiling PMT and integrating it into AutoPas.

4.1 Compiling PMT

To use PMT: Power Measurement Toolkit, we must compile and link PMT with the existing AutoPas codebase. This process is managed within the file `cmake/modules/autopas_pmt.cmake` and is very oriented towards other files like `cmake/modules/autopas_spdlog.cmake`, as they aim to include a third-party library as well. Generally speaking, our cmake file first looks for an already installed PMT version that can be reused. If no installed version is found, or if the pmt_ForceBundled option is enabled, we unpack the compressed PMT archive (libs/pmt_stable_1.0.zip) to make it available for our AutoPas build. The only difference compared to the other CMake files, including a library, is that we need to mark some CMake options as advanced predefined by the PMT library. These options control the build of different power backends, like `Xilinx` for Xilinx FPGAs architecture or `NVML` and `rocm-smi` for GPU measurements. Since AutoPas does not use the GPU for computation and focuses exclusively on CPU energy measurements, we only enable the RAPL and LIKWID backends, marking other options as advanced to prevent unnecessary configuration.
As PMT is capable of collecting power consumption measurements on different hardware, it is not necessary to control its compilation by a cmake option, and the previously used `AUTOPAS_ENABLE_ENERGY_MEASUREMENT` can be removed.

4.2 Integrating PMT in AutoPas

After compiling and linking PMT to the AutoPas code base, all the functionality that PMT provides can be used by including `pmt.h`.
Similar to previously used RaplMeter implementation [3], we abstract all the functionality into a newly created class `EnergySensor`. This has the advantage of running simulations without any overhead if the energy consumption should not or can not be measured. Therefore, the EnergySensor class offers several methods to measure the energy consumed between two states and start and stop energy measurements. An object of this class is stored in `AutoTuner.h` to be able to handle the energy consumption measurement before and after each iteration. To further elucidate the functionality of `EnergySensor` class, we will now examine three essential components.

1. **Constructor:**

```
1     EnergySensor::EnergySesnor(EnergySensorOption sensor)
    : _option(sensor) {
2       if (_option != EnergySensorOption::none) {
3         _sensor = pmt::Create(sensor.to_string());
4       } else {
5         AutoPasLog(WARN, "No sensor for energy consumption
      measurement specified.
6         Energy will not be measured);
7       }
8     }
9
```

Listing 4.1: Constructor of EnergySensor class

The constructor takes as only argument the parsed `EnergySensorOption` and stores it in the class variable `_option`. It is used in all EnergySensor functions to determine whether a valid energy measurement backend for PMT was specified. We initialize a new sensor with `pmt::Create()` if it is a valid option. If not, we will display a warning that no energy consumption measurement will take place as we did not parse a valid option.

2. **Starting measurements:**

```
1     bool EnergySensor::startMeasurement() {
2       if (_option != EnergySensorOption::none) {
3         _start = _sensor->Read();
4         return true;
5       }
6       return false;
7     }
8
```

Listing 4.2: EnergySensor::startMeasurement() method, called to mark the beginning of a new measurement

This method is used to start a new measurement. If a valid `_option` was provided in the constructor, it sets the class variable `pmt::State _start` to the return value of `_sensor->Read()`. The custom class `pmt:State` is defined in the pmt library. It represents the current energy consumption state of the system and is capable of holding multiple energy values in `std::vector<float> joules_`.
All methods in the `EnergySensor` class are checked for an valid sensor. If the option represents a valid backend, we call the encapsulated PMT function; otherwise, we just return a default value.

3. **Reading consumed joules:**

```
1     double EnergySensor::getJoules() {
2       if (_option != EnergySensorOption::none) {
3         return _sensor->joules(_start, _end);
4       }
```

```
5         return -1;
6     }
7
```

Listing 4.3: EnergySensor::getJoules() method, to get consumed Joules between two states

> Again, this method encapsulates `PMT::joules()`. The method uses this function to return the consumed joules between `_start` and `_end` state. The method additionally provides a way for handling an invalid or unspecified `_option`. Similarly, we encapsulate `PMT::seconds()` and `PMT::watts()` to get the time for one iteration and the consumed watts.

Besides the newly created `EnergySensor` class, we need to adapt some existing files. To pass the power measurement backend that should be used within PMT, we need to introduce a new option class `EnergySensorOption`. This class confirms the style of other already existing option classes and provides three options:

- EnergySensorOption::none: This option represents that no sensor should be used and no energy consumption measurement will take place.
- EnergySensorOption::rapl: This option initializes the rapl backend.
- EnergySensorOption::likwid: This option initializes the likwid backend.

Additionally, as we transition from the `RaplMeter` to `EnergySensor` class, it is necessary to update all method calls from the old implementation to the new class.
To select an energy sensor that should be used for a simulation, we extend the md-flexible code base as well as `src/AutoPas/AutoPasDecl.h` file to support a new yaml option `energy-sensor`. The YAML option should hold exactly one of the three previously defined options: `rapl`, `likwid`, or `none`. The parsed option is saved in `src/AutoPas/AutoTunerInfo.h` and has `EnergySensorOption::none` as a default value. That has the advantage that if energy measurement is enabled for a simulation but with no sensor option, the simulation can still succeed without throwing an error. The user notices the missing energy sensor by a warning logging message stating that no energy sensor was specified and the energy will not be measured.

5 Results

To ensure that PMT is a valid alternative to the existing RaplMeter implementation, we must inspect behaviors, regarding energy consumption measurement and time to completion. To verify that PMT and RaplMeter implementations produce the same results, we run several simulations with PMT and RaplMeter energy measurement enabled. As simulation, classic exploding liquid molecular dynamics scenario was used with different amount of threads. A copy of the YAML file used in the simulation can be found in 8.1. All simulations were conducted on the CoolMUC-2 Linux Cluster from the Leibniz-Rechenzentrum (LRZ). Hardware, Infrastructure, and Software characteristics for CoolMUC-2 can be found at [14].

5.1 Time to completion

First, we compare the time to completion for both energy consumption measurement implementations. This ensures that using pmt should not lead to degrading performance in terms of runtime.

5.1.1 Results before PMT modification

Table 5.1 compares the mean iteration time and total time to completion for PMT, using the rapl power measurement backend, and RaplMeter implementation, without any changes to the PMT source code.

Metric	PMT	RaplMeter
Iteration Time (s)	0.100009	0.002162
Total Time (s)	981.285641	25.946005

Table 5.1: Comparison of Iteration Time and Total Time for PMT and RapMeter implementation.

We can see that the introduced overhead by using PMT is approximately 0.098 seconds, which is more than stated in Section 3.4. The iteration time for PMT is almost exactly 100ms, which corresponds to the specified measurement interval in the RAPL power measurement backend. As explained in Section 3.2, PMT uses asynchronous measurements by starting a new thread responsible for reading energy values in a specified time interval. For the rapl power measurement backend, this interval is set to 100ms. The reason for this choice might be, that RAPL values are not updated exactly every 1ms but have some jitter. Additionally, high frequency sampling can add some overhead that PMT developers seemed to avoid [11]. AutoPas measures the energy before and after each iteration, as explained in Chapter 4, which makes measurement at 100ms infeasible as iteration runtime is often much smaller. Figure 5.1 visualizes this code flow.

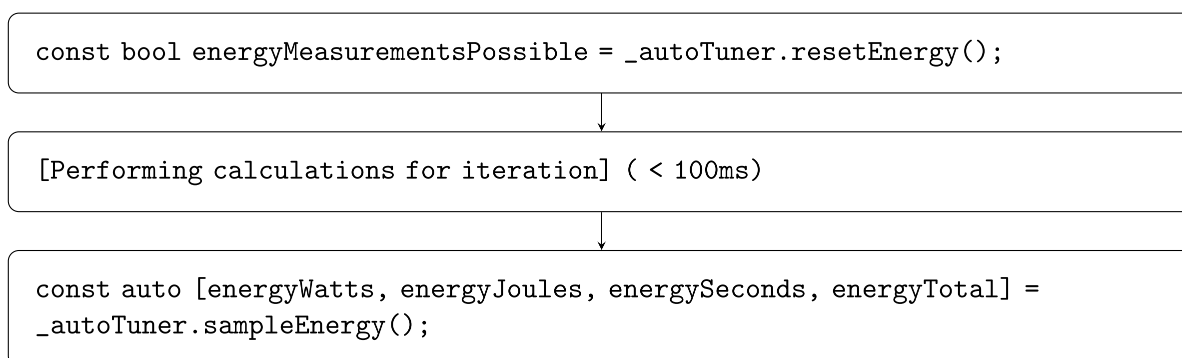


Figure 5.1: Energy measurement process in AutoPas, demonstrating the key steps for calculating energy usage during iterations.

We assume that the step "Performing calculations for iteration" takes less than 100ms. When reading the consumed energy with `_autoTuner.sampleEnergy()`, we call `PMT::Read()` a second time. Because of the measurement interval for the rapl backend, we do not have new energy consumption values, and the called thread wait for the remaining time until a new measurement is taken. This behavior can be seen in Listing 5.1, which is code, part of the `PMT::Read()` method and responsible for the large overhead when using PMT.

```
1  while (seconds(state_previous_, state_latest_) == 0) {
2    std::this_thread::sleep_for(
3    std::chrono::milliseconds(measurement_interval));
4  }
```

Listing 5.1: Sleep loop in PMT library, responsible for significant overhead

5.1.2 Adapting PMT source code

As a lower bound of 100ms per iteration for molecular dynamics simulations is not acceptable, the source code of PMT needs to be changed to reduce the introduced overhead as much as possible.
As described in 5.1.1, the overhead is introduced by the asynchronous measurement of energy consumption values and the time between two measurements. Our approach is to reduce the overhead by removing the asynchronous measurement logic and moving PMT to measure energy values "on call." This means that PMT only utilizes one thread, which directly calls the `GetState()` method, implemented in the power measurement backend, without spawning a new thread continuously reading energy values in a specified time interval. This also enables the code to optimally use all threads for computation and removes measurement intervals for power backends.

Code changes

To achieve this goal, the PMT source code was adapted and forked to the branch "pmt-stable" on github [16].
We completely removed previously used `PMT::StartThread()` and `PMT::StopThread()` methods as well as the two class variables `volatile bool thread_started_` and `volatile`

`bool thread_stop_` .
Additionally, we introduced a new class variable `volatile bool initialized_ = false;`, controlling if `state_previous` is initialized. The main changes were done to `PMT::Read()` method to call `GetState()` without using a second thread.

```
State PMT::Read() {
  if (!initialized_) {
    // Initialize the first measurement
    state_previous_ = GetState();
    initialized_ = true;
  }

  // Get the latest measurement
  state_latest_ = GetState();

  state_previous_ = state_latest_;

  return state_latest_;
}
```

Listing 5.2: New PMT::Read() method, aiming to reduce overhead

If we did not initialize state_previous so far, we set the variable to the current state of the used power backend, returned by `GetState()`. Afterward, we set `state_latest` to the returned value of `GetState()` as well and return the class variable.

5.1.3 Results after PMT modification

After modifying the PMT source code to use synchronous measurements, we conducted simulations again to compare the runtime of the PMT and RaplMeter implementation. In Figure 5.2, we compare the runtime of these simulations for PMT and RaplMeter implementation.

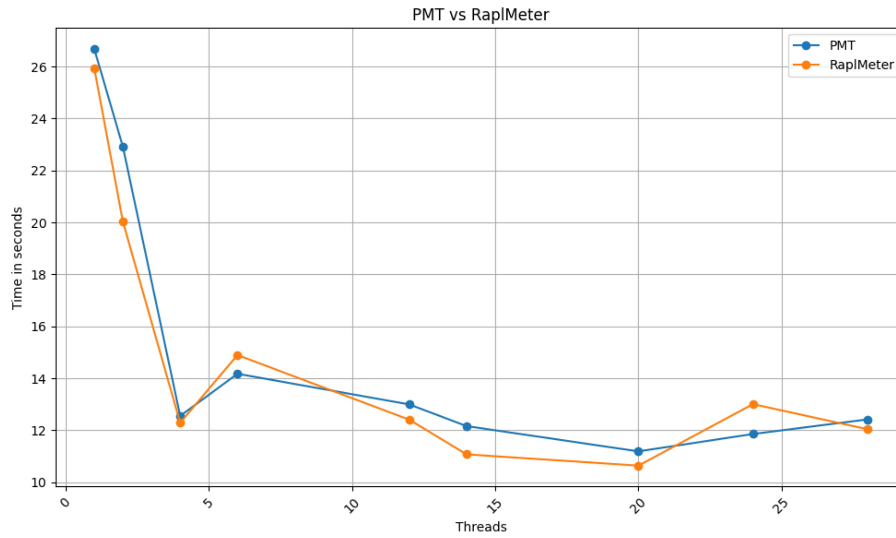


Figure 5.2: Time to completion, compared between PMT and RaplMeter

As we can clearly see in Figure 5.2, the time to completion is almost similar to the previously used RaplMeter implementation, and the updated version of the PMT library did reduce the introduced overhead.

5.2 Energy consumption measurement

Additionally, we compared the energy consumption between both implementations with the adapted PMT library. In the following plot, the total energy consumed by PMT and RaplMeter is compared for the exploding liquid simulation.

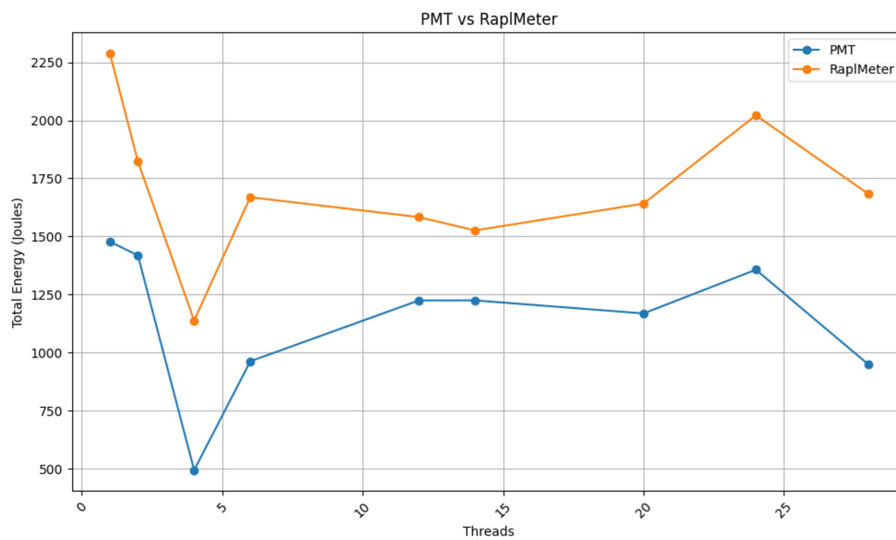


Figure 5.3: Energy consumption, compared between PMT and RaplMeter

We can clearly see in Figure 5.3 that PMT and RaplMeter energy consumption measurements follow the same trend, while RaplMeter constantly measures more energy consumed than PMT. As stated in Section 3.3.2 makes the rapl power measurement backend use of the powercap interface to read energy values. In contrast, the previously used RaplMeter implementation for energy consumption measurement relies on the perf_event interface [3]. It provides access to the following perf events [22]:

- **power/energy-psys:** Providing energy consumption values for whole system
- **power/energy-pkg:** Providing energy consumption values for whole cpu socket
- **power/energy-ram:** Providing energy consumption values for ram that is attached

Figure 5.4 visualizes the different power domains and which components are part of the domain. It is important to note that the Package domain is part of Psys.

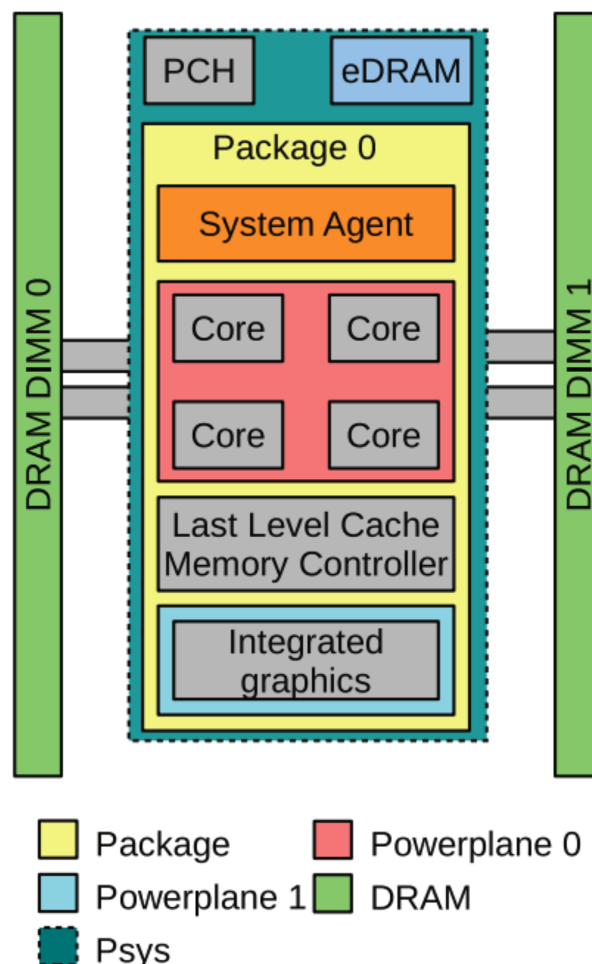


Figure 5.4: Structure of the rapl power domains [12]

RaplMeter relies mainly on the energy-psys perf event to get energy consumption for the whole system. If this perf event is not accessible, it uses the sum of energy-pkg and

energy-ram as a fallback value. The Linux CoolMuc-2 Cluster, on which all simulations were run, uses Haswell-based nodes, for which the energy-psys interface is not available, as can be seen in Figure 5.5. As we use the sum of energy-pkg and energy-ram as energy value for simulations on the CoolMuc2-Cluster, it could be, that the two perf events do not represent a valid alternative to the psys-energy value. To confirm this thesis, the same simulation was run on an HP EliteBook 850 G3 with Intel Core i7-6500U (Dual core, four threads) notebook based on the Skylake architecture, which supports the psys domain.

	Power domain supported?				
Model	**PKG**	**PP0**	**PP1**	**DRAM**	**PSys**
Sandy Bridge	yes	yes	yes	No	No
Sandy Bridge-EP	yes	yes	No	yes	No
Haswell	yes	yes	yes	yes	No
Haswell-EP [12]	yes	No	No	yes	No
Skylake	yes	yes	yes	yes	yes*

Figure 5.5: Comparison of RAPL power domains, supported by different Intel Processor Models [12]

Table 5.2 shows the results of exploding liquid simulation on Skylake architecture comparing the PMT energy values, Psys, and PKG domain. Because DRAM was not accessible for the simulation, energy per iteration and total is 0. We notice, that the reported PMT value is drastically higher than the PSys value per iteration and total. Another important thing that we notice is that the reported value for the PKG domain is greater than the one for PSys. This does indicate some error in the RaplMeter values as the PKG domain is part of the PSys domain, as seen in Figure 5.4, and it should always hold that energy-psys is greater than energy-pkg. To investigate further why PMT does report a lower energy

Metric	PMT	PSys	PKG	DRAM
Energy per Iteration [J]	0.03011	0.01363	0.02350	0
Total Energy [J]	361.324	163.583	282.091	0

Table 5.2: Comparison of energy values between PMT and RaplMeter on Intel Skylake architecture.

consumption than RaplMeter when using energy-pkg and energy-dram as total energy consumed, we did some extensive research about perf events and how the power domains for RAPL are built up. The final hint was found in a Firefox Source Docs [1]. Especially the fact that energy-pkg >= energy-cores + energy-gpu relationship holds is interesting for us, as it suggests that the energy-pkg domain does also include the GPU energy consumption and maybe even other uncore components. That would explain the difference between energy consumption values for PMT and RaplMeter, as PMT only measures the CPU energy consumption values [19], while RaplMeter also includes other hardware components like GPU. To confirm this thesis, we adapt the RaplMeter implementation to also measure the consumed energy for energy-cores and energy-gpu perf events and compare the values to energy-pkg. When running an exploding liquid simulation utilizing one thread on the same

EliteBook 850 G3, we get the values displayed in Table 5.3, which does confirm the thesis

Metric	energy-pkg	energy-cores	energy-gpu
Total Energy [J]	409.079	324.348	16.974

Table 5.3: Results for pkg, cores, and gpu domain for exploding liquid simulation on Intel Skylake architecture.

from [1], that energy-pkg >= energy-cores + energy-gpu does hold and the pkg domain includes GPU and other uncore components. As they are not utilized by PMT, they should be in an IDLE state, consuming a constant amount of energy, which would explain what seems to be a static value added to the RaplMeter energy consumption values. To further investigate this, the branch with the adapted RaplMeter version was used for simulations on the CoolMuc-2 cluster to compare the values for energy-pkg, energy-gpu, energy-cores, and PMT. After running the simulation, we noticed that the values for energy-cores and energ-gpu are reported as 0 Joules. The reason for this could be that the corresponding perf events are not available on CoolMuc-2, or access restricted. We can confirm that the perf events are available by running `perf list` and looking for `power/energy-cores` and `power/energy-gpu`. As they should exist, restricted access is probably the cause of the missing values. Since there is currently no way to confirm our thesis on the CoolMuc-2 cluster, we have to rely on the results obtained from the EliteBook. We suspect that the energy-pkg domain also includes the GPU and other uncore components, which leads to a higher report of energy consumption values. In this case, PMT seem to be a better solution for measuring the energy consumption, as the GPU and other uncore components are currently not utilized by AutoPas and including them in the consumption measurements might lead to a falsification of values.

5.3 Comparing tuning configurations

After comparing simulation time and energy consumption values between PMT and RaplMeter implementation, the last thing we need to check are the simulation configurations used, when tuning for energy. The exploding liquid scenario was used again as molecular dynamics simulation. As data we use the columns `Container`, `Traversal` and `Data Layout`. Together they represent the used configuration for one iteration step. As we got 12,000 rows in the csv file, each containing a simulation configuration, we are using a python script, analyzing the specified columns and only returning the rows with a different configuration for PMT and RaplMeter implementation. Table 5.4 does show these differences. We can see

Range	PMT Configuration	RaplMeter Configuration
1130 - 7129	VerletListsCells, vlc_sliced_balanced, AoS	VerletListsCells, vlc_sliced_c02, AoS

Table 5.4: Differences in tuning configurations

that the only difference between both configurations is from Iteration step 1130 to 7129, where PMT is using the vlc_sliced_balanced traversal algorithm, while RaplMeter is using

vlc_sliced_c02. This might be because both traversal algorithms are very close in terms of runtime and energy. The AutoTuner picks one over the other as there are slight differences in relative performance when using RaplMeter or PMT implementation. Additonally, we note that the container and data layout used by both implementations are the same throughout the simulation. This does indicate that PMT does report a lower energy consumption for the vlc_sliced_balanced traversal algorithm than RaplMeter implementation. A possible explanation is the difference in energy consumption values, analyzed in Section 5.2, and that they may have a very similar performance. In Subsection 6.1.1, we will compare different tuning configurations, with respect to runtime and energy consumed per iteration, and see that the AoS - sliced-balanced configuration for Verlet Lists Cells container is performing better than AoS - sliced-c02 and the tuning for sliced_balanced traversal algorithm is correct.

5.4 Recommendations

By removing the asynchronous measurement from the PMT library, explained in Subsection 5.1.2, we can significantly reduce the introduced overhead. It also follows that PMT does not wait for a new measurement to be taken before returning energy consumption values. This leads to the problem that for some iterations, PMT might report 0 Joules consumed. An example can be seen in Table 5.5, showing the iteration logs for exploding liquid simulation on CoolMUC-2 with tuning for energy enabled.

Iteration	iteratePairwiseTotal[ns]	energyJoules[J]	energySeconds[S]
1314	716742	0.10626	0.0007169
1315	726866	0.06372	0.0007259
1316	712427	0	0.0007119
1317	717689	0	0.0007181
1318	722565	0	0.0007221
1319	743505	0	0.0007438
1320	734244	0.05701	0.0007341
1321	690794	0.05133	0.0006912
1322	671123	0.11127	0.0006719
1323	680869	0.10400	0.0006809
1324	677865	0.05627	0.0006782
1325	681012	0	0.0006809
1326	703227	0	0.0007031

Table 5.5: Energy values for different iterations on the CoolMUC-2 cluster

We see in Table 5.5 that for several iterations, the reported consumed energy is 0 Joules. Especially for simulations with a low amount of particles and thus with an iteration time of $< 1ms$, PMT does not always obtain a new measurement. Thats why both `EnergySensor` class variables `_start` and `_end` contain the same state, resulting in no difference in energy consumed. Therefore we recommend to set the tuning phase to a number of M iterations that ensures $M \cdot T > 1ms$, where T is the iteration time. This adjustment will help guarantee that energy values are captured effectively during the tuning phase.

6 Benchmarks

After confirming that PMT produces valid energy consumption results and is an alternative to the previously used RaplMeter implementation, we conduct simulations using different data containers, layouts and traversal algorithm. We observe, what impact different tuning configurations, number of particles and threads have on the energy consumption of simulations.

6.1 Tuning configurations

As a simulation, we used the Spinodal Decomposition molecular dynamics example. This simulation can be divided into two steps. In the first step, a cuboid domain is filled with 4, 096, 000 particles and the simulation is run for 100, 000 iterations with a constant high temperature until an equilibrium state is reached [3]. In the second step, the result of the first simulation is used as a checkpoint, and the temperature drops for 30, 000 iterations. This causes the particles to condense into clusters. Figure 6.1 shows the domain before and after the second step.

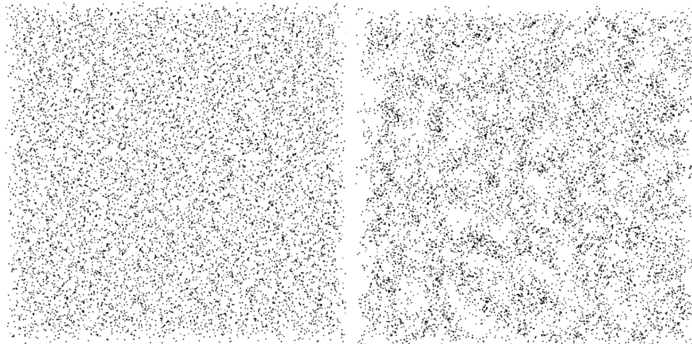


Figure 6.1: Particle domain left after equilibrium state reached and on the right after forming clusters [3].

6.1.1 Comparing

In the first benchmarks, we are using the first step of the Spinodal Decomposition simulation and comparing the energy consumption and time per iteration between:

- Container: Linked Cells, Verlet Lists Cells
- Data layout: Structure of Arrays (SoA), Array of Structures (AoS)
- Traversal: c08, c18, sliced-balanced, sliced-c02

Since the macroscopic state remains unchanged for the equilibrium state, we can use the data from this simulation to compare the different traversal algorithms and data layouts concerning energy consumption and time per iteration [3]. All simulations were run using 28 threads and one node on the CoolMUC-2 using the cm2_tiny cluster.

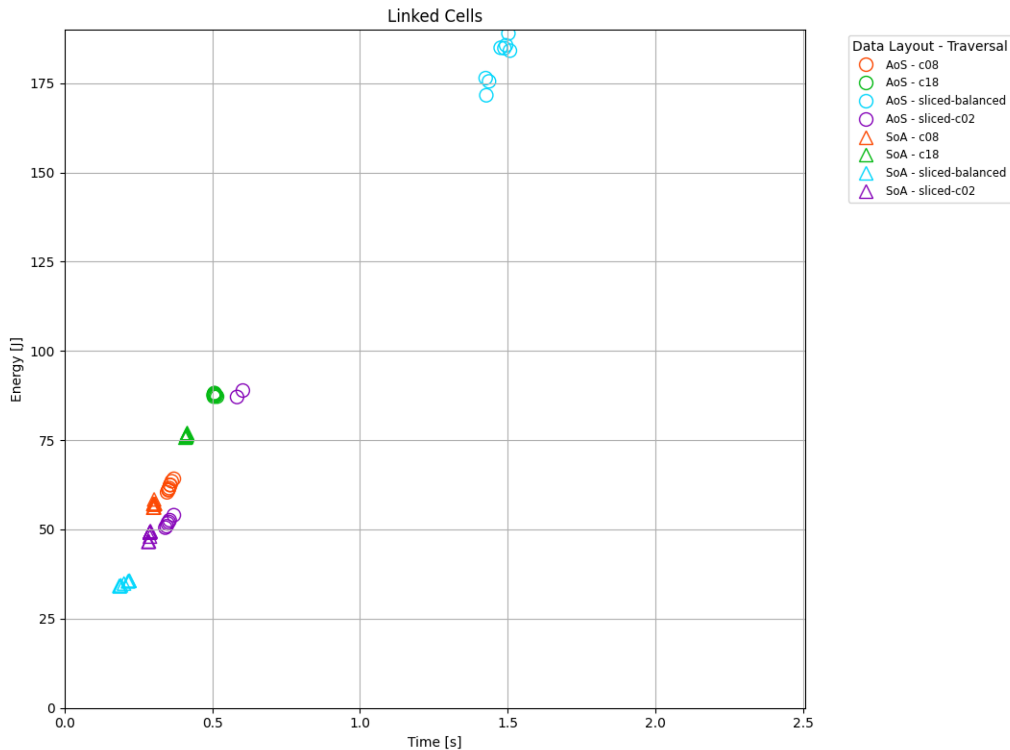


Figure 6.2: Comparing energy consumption and time per iteration for Spinodal Decomposition with different configurations for Linked Cells container

Figure 6.2 plots the mean value for energy usage and time to completion per iteration for ten iterations, as neighbor lists are rebuilt every ten iterations [3]. Iterations from 99000 to 99080 were used as the underlying data to determine the energy consumption during the equilibrium state. We can clearly see that the SoA data layout consumes less energy and is faster than the AoS data layout with the same traversal algorithms for the Linked Cells container. As stated in Section 2.2, retrieving successive particle information can be done with one load for the SoA data layout. In contrast, for AoS, the information must be gathered separately for each particle. Since the memory access is more efficient for the SoA data layout, the energy consumption and time for one iteration are lower than for AoS. Another thing we notice is that sliced traversals are faster and consume less energy than colored ones. One exception is the AoS - sliced-balanced configuration, which consumes the most time and energy. As stated in Subsection 2.4.2, the sliced-balanced traversals algorithms reduce the scheduling overhead to a minimum by statically dividing the domain into slices of equal size and assigning each slice to one thread. As the particles are distributed evenly over the domain, as seen in Figure 6.1, and the particles remain in a constant state, each thread has approximately the same computational load, and the static assignment of

cells reduces the overhead.

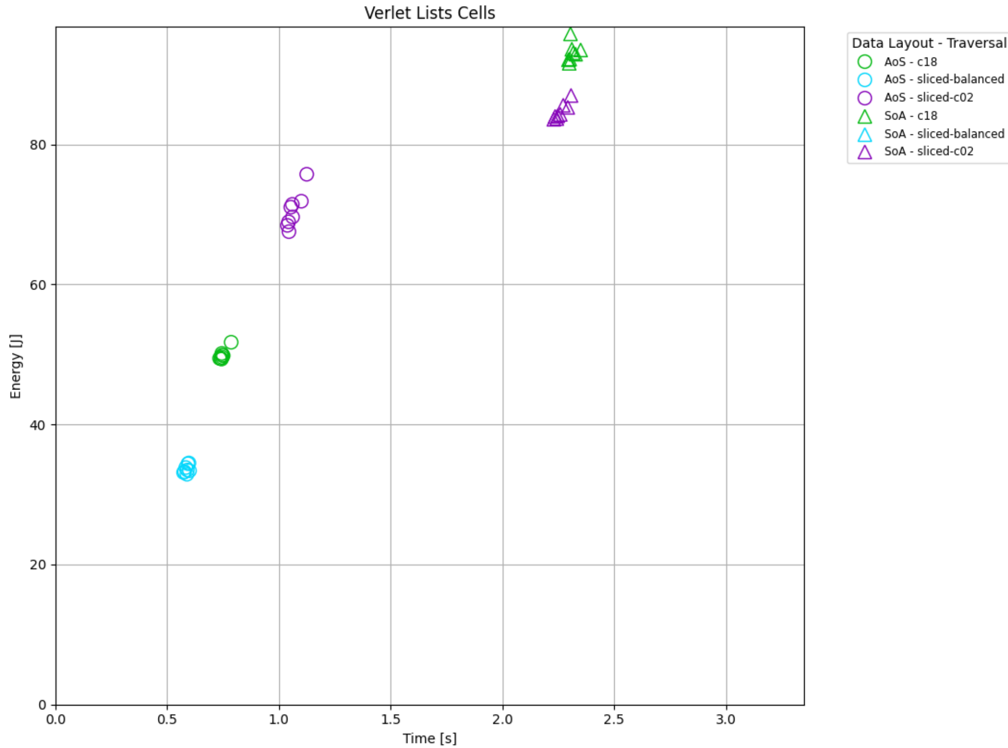


Figure 6.3: Comparing energy consumption and time per iteration for Spinodal Decomposition with different configurations for Verlet Lists Cells container

Figure 6.3 plots the same data for the Verlet Lists Cells container. The simulation for SoA - sliced-balanced configuration did fail on the CoolMuc2 cluster with a `std::bad_alloc` error, even after multiple tries, and is not part of the plot. We can see in Figure 6.3 that in contrast to the Linked Cells simulation, the AoS data layout performs better than SoA. This can be explained because Verlet Lists Cells rely on neighbor lists, and data can be better accessed for each particle with the AoS data layout than for the SoA one, where the entire vector for each cell has to be loaded. Moreover, memory access operations consume significantly more energy than arithmetic operations. According to recent studies [21] 10-100 times more. Although the SoA layout improves data locality and allows for faster memory access, the disadvantage is that it takes longer to rebuild the Verlet Lists. This rebuilding process is more time-consuming than the quicker memory access, leading to the SoA layout requiring more time and energy overall compared to AoS. When comparing the rebuild time between both layouts, SoA takes on average 16 s while for AoS the rebuild only takes 5 s. Additionally, we notice that the iteration time for Verlet Lists Cells is much longer for all data layouts and traversal algorithms than for Linked Cells. At the same time, the energy consumption between both is pretty close together. This can be explained as rebuilding neighbour lists for Verlet Lists Cells container each 10 iteration introduces a overhead and can take up to 16s.

6.1.2 Metrics

After comparing different tuning configurations for time and energy per iteration, we want to calculate performance metrics that were part of a study conducted by Thomas Rauber, Gudula Rünger, and Matthias Stachowski in 2018, focusing on improving energy efficiency in high-performance computing [17]. The study defines several performance metrics that we want to use to study the energy performance of MD simulation. We consider the Spinodal Decomposition equilibration case as problem for our computing of these metrics. The iterations were lowered to 1000 as we want to compare the metrics for different amounts of threads, and the time to completion would otherwise exceed the time limitation for single thread runs on the CoolMuc-2 cluster.

Speedup

As the first performance metric, we calculate the Speedup obtained using multiple threads. The metric is defined as

$$S(p) = \frac{T_{\text{seq}}}{T_{\text{par}}(p)}$$

where T_{seq} is the sequential execution time and $T_{\text{par}}(p)$ is the parallel execution time, when using p threads [17]. Figure 6.4 shows the Speedup for different configurations for the Linked Cells container when running the simulation for 1, 2, 4, 8, 12, 14, and 28 threads.

We can see in Figure 6.4 that the Speedup is increasing for more threads used. This is expected as the computational load is spread among more threads. It is interesting for us that the Speedup is not linear. This is probably due to increased scheduling overhead when assigning more threads to cells. Additionally, sequential parts of the code can not be parallelized, thus limiting the runtime to a lower bound. We note that the SoA - lc-sliced-balanced configuration performs best with a maximum speedup of 15.8 for 28 threads. This conforms with the behavior observed in Figure 6.2 as the configuration performed there the best. The AoS - lc-sliced-balanced configuration on the opposite is the worst performing. The maximum Speedup of 8.3 is reached for 12 threads utilized and decreases if we increase the number of threads used. This also conforms with the behavior observed in Figure 6.2.

Figure 6.5 shows the Speedup for the Verlet Lists Cells container. We can see that the Speedup obtained by the AoS data layout is significantly larger than for SoA, where we only achieve a speedup of about 2.3 for 28 threads. We note that the obtained Speedup for Verlel Lists Cells is much lower than that of the Linked Cells data container. The maximum Speedup we can achieve for Verlet Lists Cells is 5.8 for the AoS - c18 configuration, while for Linked Cells, the maximum is 15.8 for the SoA - sliced-balanced configuration. This can be explained by the fact that the Verlet Lists Cells container requires more careful management of neighboring particles. As the number of threads increases, managing these lists becomes more challenging, especially when updating and reading from shared memory.

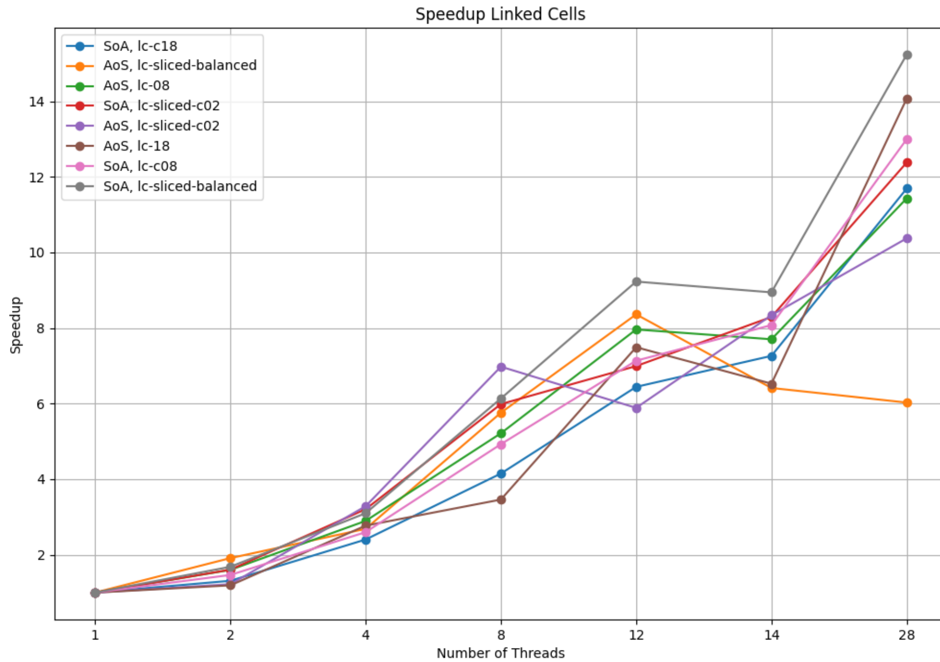


Figure 6.4: Speedup obtained for Linked Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with 1000 iterations

Energy speedup

The energy speedup is defined analogously to the Speedup.

$$ES(p) = \frac{E(1)}{E(p)}$$

where $E(p)$ is the consumed energy using p threads. It expresses the relative energy consumption difference that occurs using p threads compared to the energy consumed using one thread [17].
Figure 6.6 shows the energy speedup for different configurations, using again 1, 2, 4, 8, 12, 14, and 28 threads for the Linked Cells container.

We can see in Figure 6.6 that for all configurations, the energy speedup is greater than 1. This implies that by utilizing more threads, less energy is consumed for the whole simulation as it must hold $E(1) > E(p)$ for $ES(p) > 1$. This is quite interesting as we see that the reduced runtime for the simulation and the energy we save is greater than the additional energy we consume by utilizing more threads. Except for the AoS - sliced-balanced configuration, we note that utilizing more threads comes with a lower energy consumption for the whole simulation.
Like Figure 6.6, Figure 6.7 plots the exact data for the Verlet Lists Cells.

In Figure 6.7, we can see that the AoS data layout performs better than all configurations with the SoA data layout. Similar to the behavior we observed for Speedup in Figure 6.5.

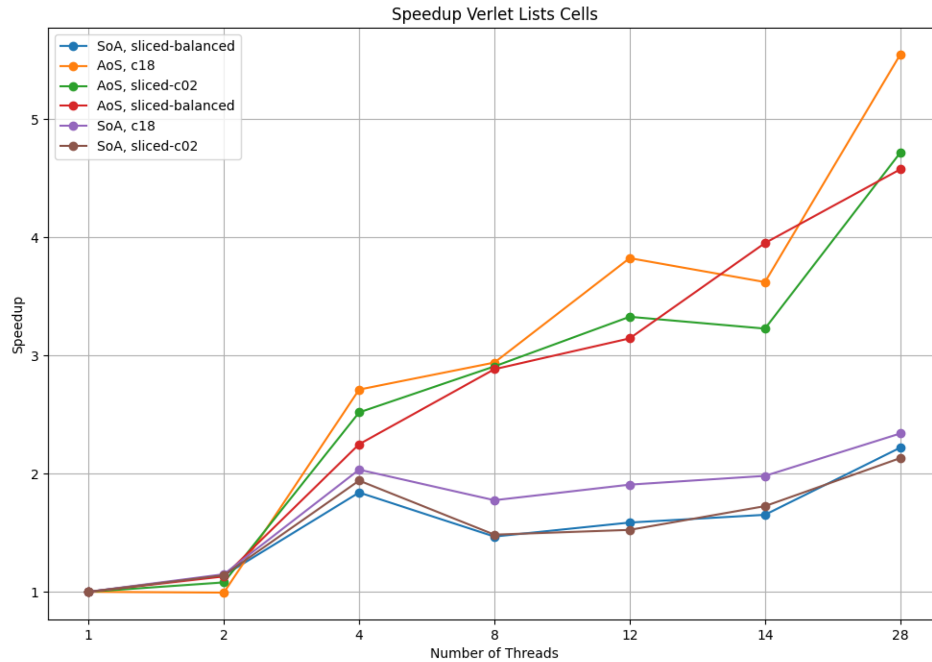


Figure 6.5: Speedup obtained for Verlet Lists Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with 1000 iterations

Interestingly, the energy speedup for the configurations using SoA data layout remains the same for 4 and 28 threads used and even drops for 8, 12, and 14. This implies that the energy consumption remains at least the same between 4 and 28 threads, and we do not increase energy efficiency by using more than four threads. The energy speedup for the AoS data layout also drops when using more than four threads. Still, it increases again for 28, where the maximum energy speedup is reached.
When comparing the Linked Cells and Verlet Lists Cells container, we notice that the maximum and minimum values for the energy speedup are very close to each other while the values for Speedup are very far apart. This is probably due to how Linked Cells and Verlet Lists Cells containers handle memory access and computational efficiency. As Verlet Lists Cells uses neighbor lists that store particles, expensive memory accesses are limited, costing up to 10-100 times more energy than arithmetic operations [21]. However, rebuilding these neighbor lists requires more computational overhead, reducing the Speedup for simulations using the Verlet Lists cells container. Linked Cell containers are not used on lists to identify neighboring particles, reducing the computational overhead and resulting in a higher speedup. However, each thread does more memory access, limiting the energy speedup for more threads used.

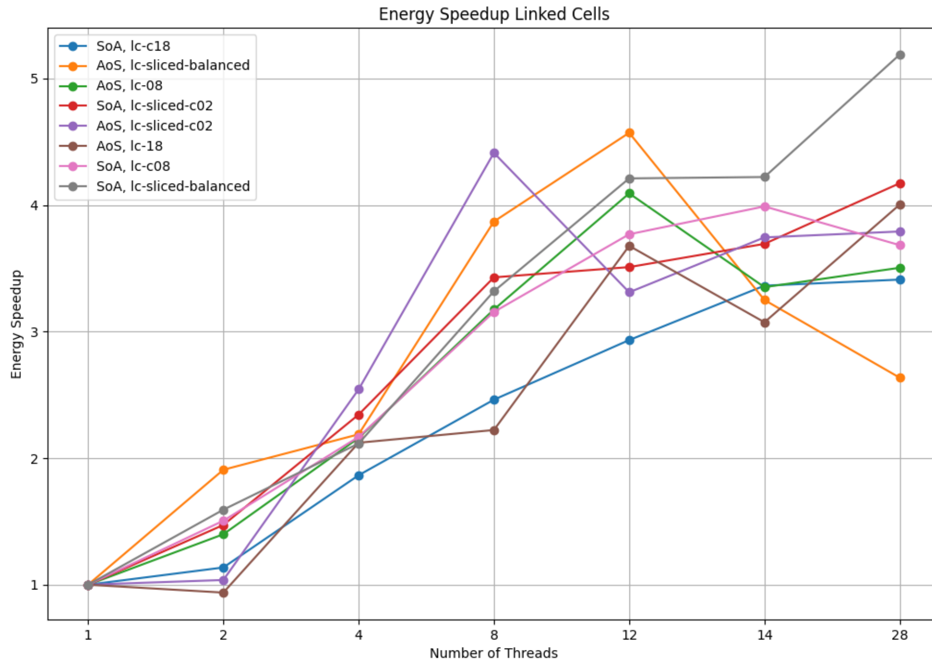


Figure 6.6: Energy Speedup obtained for Linked Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition Equilibration simulation with only 1000 iterations

Energy delay product

As the last metric, we calculate the energy-delay product for each configuration. It is defined as

$$EDP(p) = E(p) * T(p, 1)$$

. It combines the effects of execution time and energy consumption and captures the translation of energy into useful work. It is defined for a given number of threads and does not compare performance across them, like the energy speedup. For a simulation with a different number of threads used, our aim is to select the simulation with the minimum energy-delay product, as this is similar to minimizing both runtime and energy.
Figure 6.8 plots the energy-delay product (EDP) for the Linked Cells data container.

When analyzing Figure 6.8, we note that the SoA - lc-sliced-balanced configuration has the lowest EDP and the best energy efficiency while the AoS - lc-sliced-balanced has the highest EDP for 28 threads and hence the worst energy efficiency. This does conform with the results obtained by calculating (Energy) Speedup. As the EDP combines the execution time with energy consumption and SoA - lc-sliced-balanced configuration performed best in both metrics, the EDP is the lowest for this. AoS - lc-sliced-balanced performed worst in both metrics and has the highest EDP for 28 threads.
Additionally, Figure 6.9 plots the energy-delay product (EDP) for the Verlet Lists Cells container.

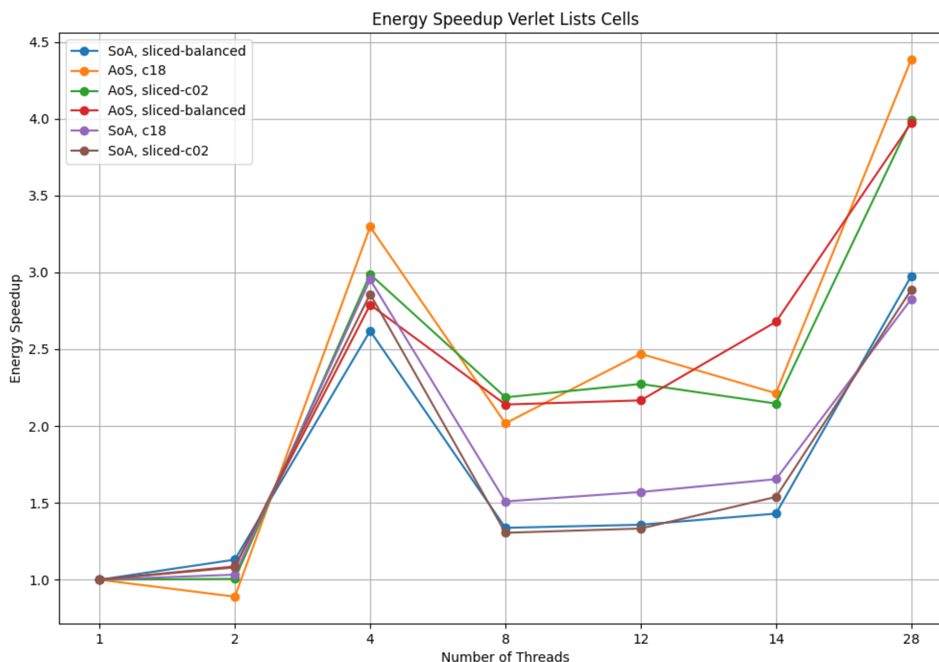


Figure 6.7: Energy Speedup obtained for Verlet Lists Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition Equilibration simulation with only 1000 iterations

As already noticed for the Energy Speedup for Verlet Lists Cells container, we can see in Figure 6.9 that the SoA data layout has a clearly higher EDP than the AoS one. The selection of data layout has a far greater impact on energy efficiency than the selection of a traversal algorithm. Another thing we note is when comparing Linked Cells and Verlet Lists Cells data containers, the EDP for both, in general, is very similar when only four or fewer threads are utilized. When using eight or more threads, the EDP for the Linked Cells container constantly decreases, while for Verlet Lists Cells, the EDP almost remains the same and only drops for 28 threads used.

6.1.3 Energy Efficiency

As we outlined in Subsection 6.1.2, a faster iteration time is closely related to less energy consumed. However, when comparing AoS - sliced-c02 and AoS - c18 configurations with respect to time and energy per iteration, we notice that this thesis does not always hold. Figure 6.10 compares time and energy per iteration, averaged over Spinodal Decomposition Equilibration simulation, for the two configurations.

We can clearly see that even if the iteration time for the sliced-c02 is greater than for the c18 configuration, it consumes less energy. We can conclude from this that the power, defined by $P(Watts) = \frac{Energy(J)}{Time(s)}$, has to be lower for the sliced-c02 configuration than

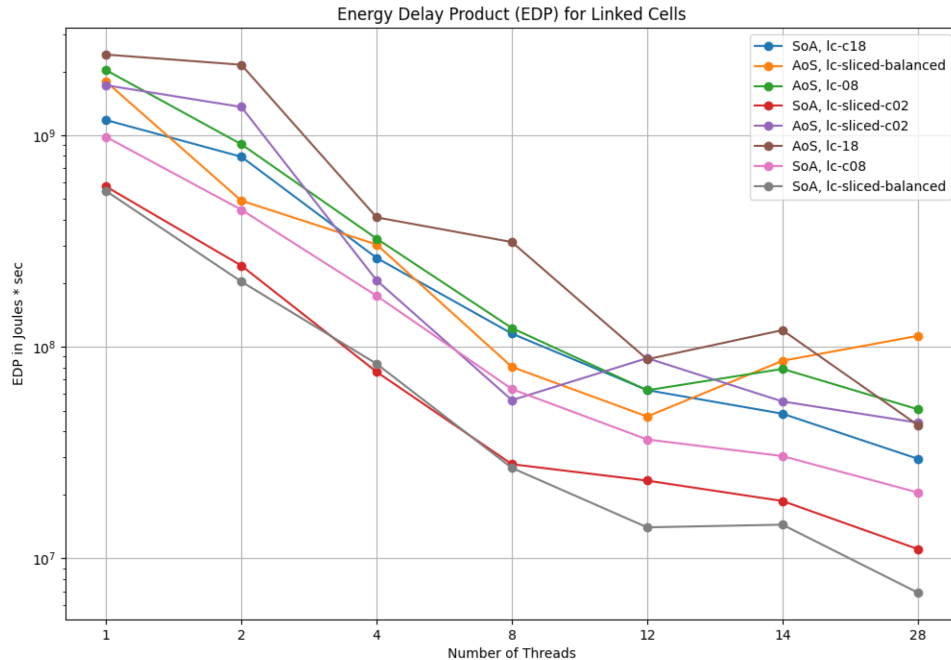


Figure 6.8: Energy delay product (EDP) obtained for Linked Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with only 1000 iterations

for the c18 one. To investigate the reason for the higher power draw, we ran a second simulation with CMAKE option `AUTOPAS_LOG_FLOPS` enabled to compare the computational power of both implementations. Figure 6.11 plots the power and FLOPs compared between both configurations using Spinodal Decomposition equilibration simulation with 100,000 iterations.

As expected, we can see that the power for the AoS - sliced-c02 configuration, with a mean value of 144.073 Watts, is constantly lower than for the AoS - lc-18 one, with 172.083 Watts. The FLOPs per iteration are also higher for the AoS - lc-18 configuration, with a mean value of 1784.768 MFLOPs per iteration, while the other configuration has only a mean value of 1588.241 MFLOPs per iteration. A possible explanation for this is that sliced-c02 needs more memory transfer, which limits the computation and results in a lower FLOP count. However, as sliced-c02 can do the computations over a larger time, it draws less power, as the frequency is scaled lower than for the lc-c18 configuration automatically, resulting in lower overall energy. This frequency scaling is done automatically by the Intel® Xeon® Processor E5-2697 v3 CPU, which is used as the computational CPU on the CoolMuc-2 Linux cluster. The Intel Turbo Boost technology can scale the CPU between a frequency from 2.60 GHz to 3.60 GHz, based on the current computational load [10] [9]. By lowering the frequency, the CPU draws less power and consumes less energy, explaining why AoS-sliced-c02 consumes less energy in more time than AoS-c18 in Figure 6.10.

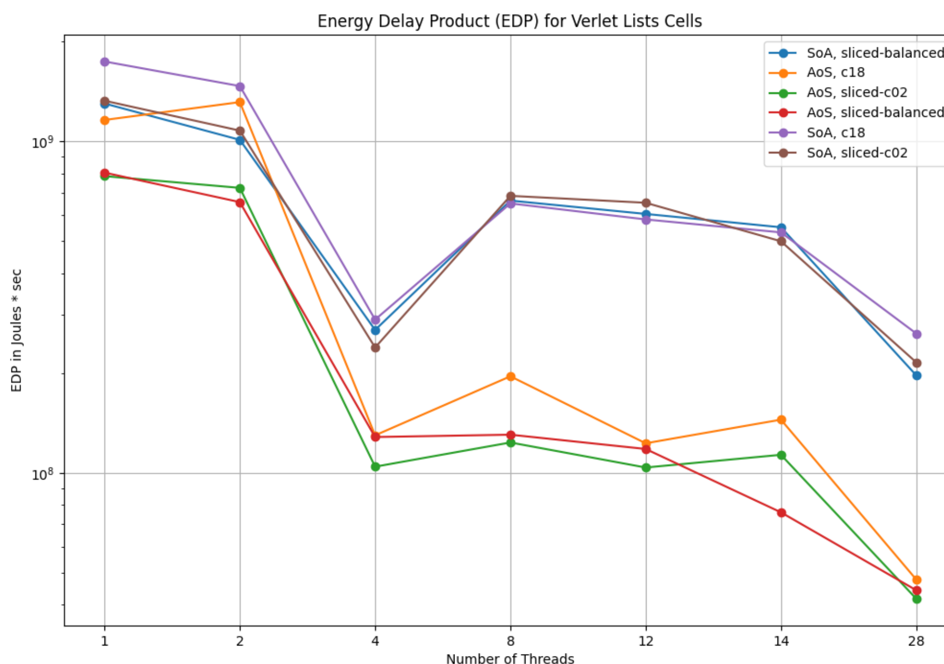


Figure 6.9: Energy delay product (EDP) obtained for Verlet Lists Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with only 1000 iterations

6.2 Tuning for energy efficiency

To further investigate the relationship between runtime and energy efficiency, we compare the time to completion and energy consumed between two simulations with the same configurations: one tuned for energy and the other tuned for time. We use the classic molecular dynamics case exploding liquid, with all available containers, data layouts, and traversal algorithms as configurations. With auto-tuning enabled, AutoPas selects the best suitable simulation configuration during runtime from all available options. We are running the simulation on 1, 2, 4, 6, 12, 14, 20, 24, and 28 threads for each tuning metric to monitor any difference in time or energy consumed.
Figure 6.12 shows the time to completion for exploding liquid simulation tuned for energy and time. For the simulations using six threads, the time to completion of the energy tuned simulation is lower than the one tuned for time, which might be due to performance irregularities on the CoolMuc-2 Linux cluster. Overall, we observe that the time to completion results for both the configurations closely follow each other. This may suggest that even when tuning for energy, the fastest configuration gets selected by the AutoTuner, as this seems to be also the most energy-efficient one. Figure 6.13 shows the energy consumed for both simulations. When comparing the plot with Figure 6.12, we observe that the time to completion is related to energy consumed. If we look, e.g., at the simulations using

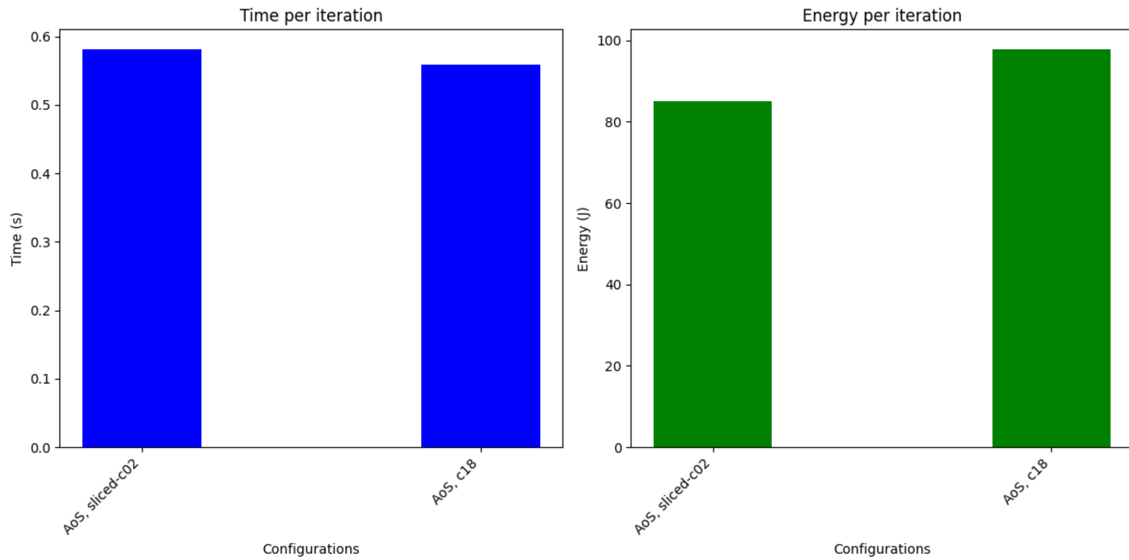


Figure 6.10: Comparing time and energy per iteration for AoS - sliced-c02 and AoS - c18, using Spinodal Decomposition equilibration

one thread where the time to completion is nearly identical, the energy consumed is also very close together, while for six threads used, the time to completion for the energy-tuned simulation is remarkably lower than for the one tuning for time. This also results in lower energy consumption. Another thing we notice is that in some cases, the energy consumed by simulations tuned for time is lower than that of the ones tuned for energy. In Figure 6.13, we can see that this is the case for 2, 12, and 14 threads used, where the runtime for energy-tuned simulation is higher as well. To further support our thesis, we compare the used configurations for each iteration between both simulations, similar to the evaluation in Section 5.3. Given the complexity of comparing configurations across all nine simulations, each with 12,000 iterations, we focus on two specific cases: simulations with one thread, where time to completion is nearly identical, and simulations with six threads, where there is a notable difference in time to completion. Table 6.1 shows the range of rows where the simulation configurations selected by the AutoTuner are different when tuning for energy and time when using one thread. Table 6.2 shows the same data for six threads used.

Row Range	Energy	Time
1130 - 7129	VerletListsCells, vlc_sliced_balanced, AoS	VerletListsCells, vlc_sliced_c02, AoS

Table 6.1: Differences in Configuration Between tuning for energy and time using one thread

In both cases, we can see that the container and data layout are the same throughout the simulation for energy and time tuning. The only difference is the traversal algorithm used from iteration 1130 to 7129. For the simulation using one thread and tuned for energy, the `vlc_sliced_balanced` algorithm is used, while the `vlc_sliced_c02` is selected when tuning for time. The time to completion compared between both traversal alogirthms is

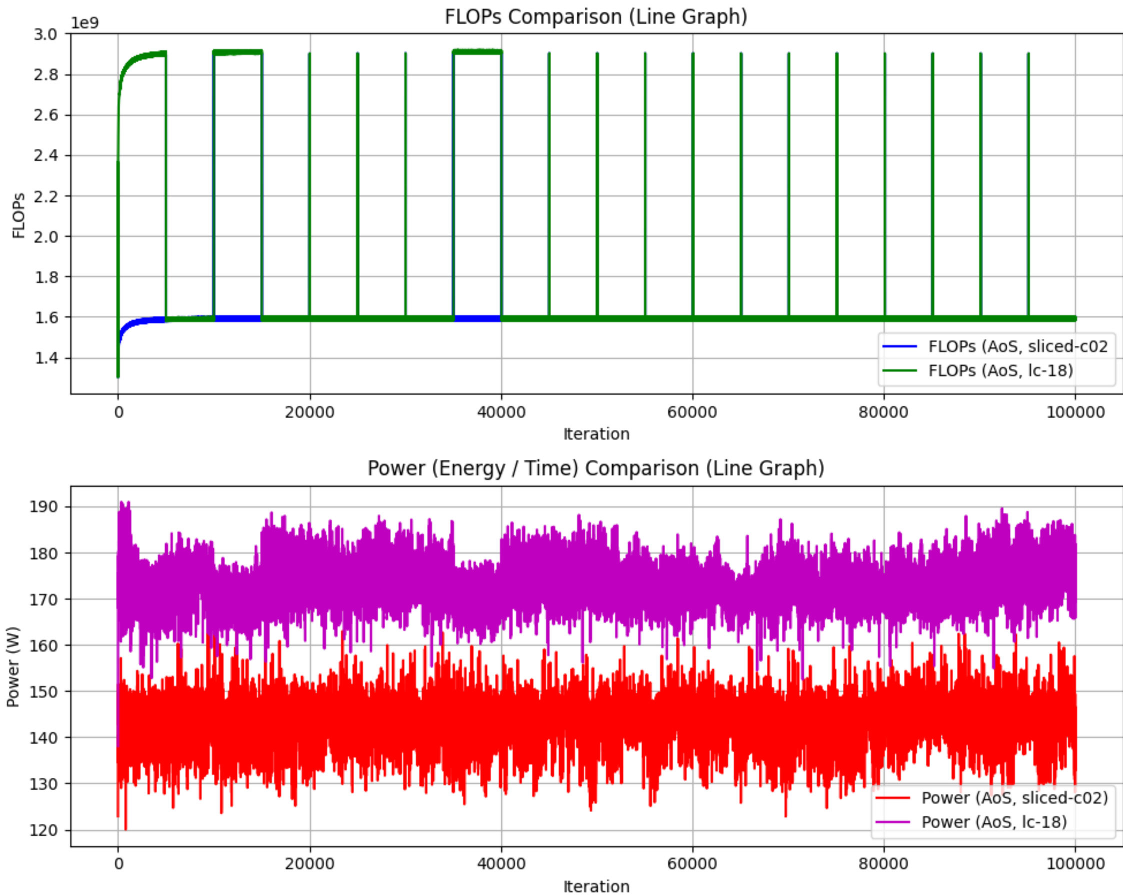


Figure 6.11: Comparing Power and FLOPs for AoS - sliced-c02 and AoS - c18, using Spinodal Decomposition equilibration

0.5% lower for `vlc_sliced_c02` while `vlc_sliced_c02` consumes 1.6% more energy. From this experiment, we can conclude that using the `vlc_sliced_balanced` traversal alogrithm might be the better choice as time to completion remains nearly identical. At the same time, energy consumption is reduced compared to the `vlc_sliced_c02` algorithm. For the simulation using six threads, we can see a remarkable difference for energy consumed in Figure 6.13 as well as for time to completion in Figure 6.12. This suggests that `lc_c04_HCP` is faster and more energy efficient than `lc_c01_combined_SoA`.
In this section, we further supported our thesis that a lower runtime strongly correlates with reduced energy consumption. In most cases, the configuration with the lowest runtime also resulted in lower energy consumption, suggesting that optimizing for speed often leads to energy efficiency. This could be observed in Figure 6.12, where even when tuning for energy, the AutoTuner selected runtime-wise configurations close to the simulations tuned for time.

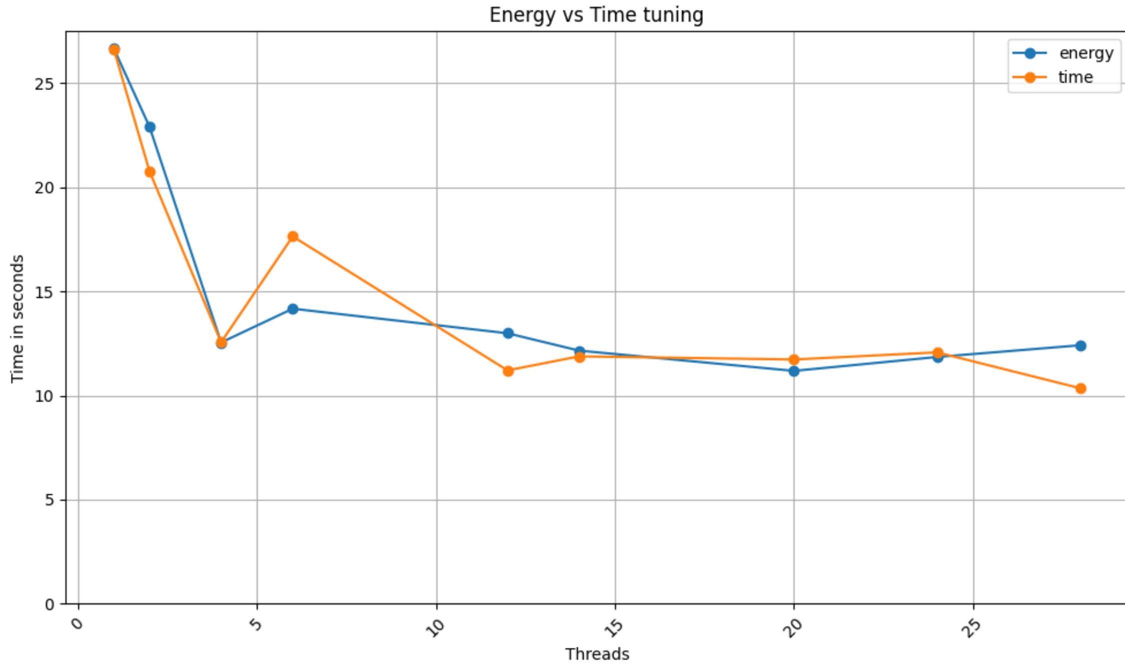


Figure 6.12: Comparing time to completion between tuning for energy and time

Row Range	Energy	Time
1130 - 7129	LinkedCells, lc_c04_HCP, SoA	LinkedCells, lc_c01_combined_SoA, SoA

Table 6.2: Differences in Configuration Between tuning for energy and time using six thread

6.3 Performance comparison across different architecture

As the last part of this thesis, we want to compare AMD and INTEL CPU architecture and what difference can be observed between both, regarding energy consumption and runtime. The Spinodal decomposition equilibration scenario was used as a simulation, reduced to 1000 iterations for 1, 4, 8, and 16 threads. For the INTEL architecture, we used the CoolMuc-2 Linux cluster, which is based on an Intel Xeon E5-2697 v3 14-core Haswell CPU, each having 28 cores and a frequency of 2.6 GHz [8]. For the AMD architecture, the simulation was run on the HPE Apollo cluster of High-Performance Computing Center Stuttgart. It uses the CPU type AMD EPYC 7742 with 64 cores per CPU and a frequency of 2.25 GHz [6]. We calculate energy - /runtime-speedup to compare both CPU architectures, introduced in Subsection 6.1.2. Figure 6.14 plots the speedup we obtain by utilizing multiple threads using the Linked Cells container. As configurations, the AoS - sliced-balanced, plotted on the left of Figure 6.14 and SoA - sliced-balance, on the right, were used, as they were the configurations that performed worst and best in Subsection 6.1.1.

For the AoS - sliced-balanced configuration, AMD is constantly performing better with a higher speedup than INTEL. This is because the AMD EPYC 7742 CPU has eight memory channels compared to 4 for the INTEL CPU. This allows AMD CPU to process more data

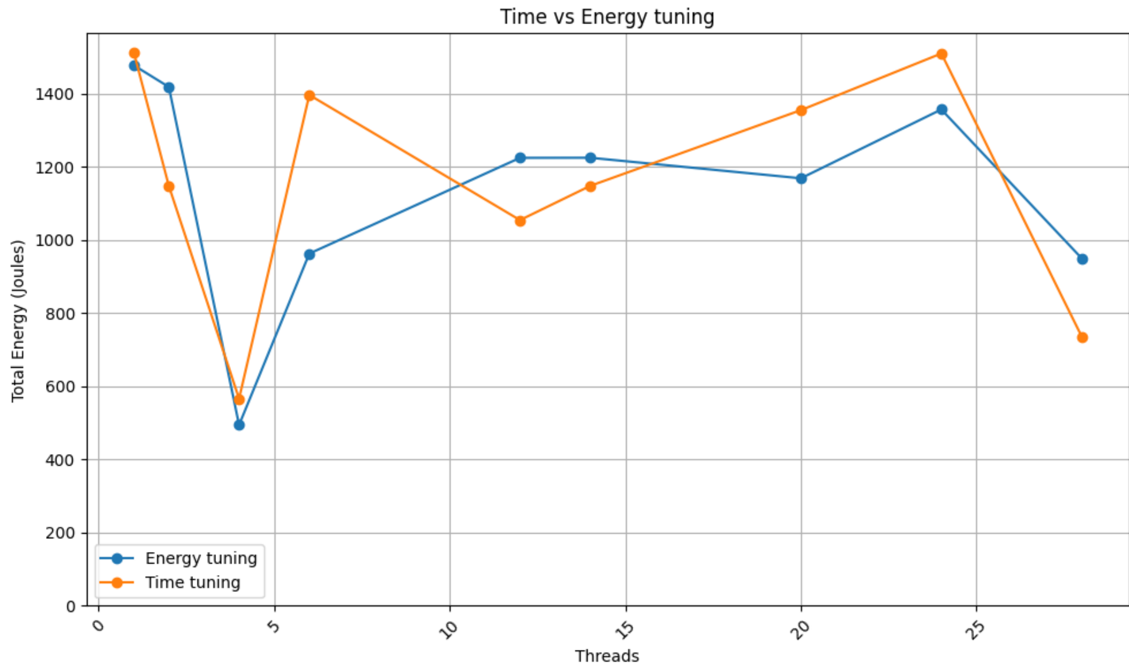


Figure 6.13: Comparing total energy consumed between tuning for energy and time

in parallel, which is particularly important for the Array of Structure data layouts, where multiple structures must be accessed concurrently. Conversely, for the SoA - sliced-balanced configuration, we can see that AMD is performing better for 1 and 4 threads, but for 8 and 16, the INTEL architecture is achieving a higher speedup. The AMD CPU achieves a higher speedup for 1 and 4 threads, as it can leverage the larger L3 cache with 256 MB compared to 35 MB for the INTEL CPU. Especially for the Structure of Arrays layout, the arrays, storing particle data, can be directly loaded from the L3 cache. For 8 and 16 threads, the INTEL CPU achieves a higher speedup, as the frequency of 2.6 GHz is notably higher than for AMD CPU with 2.35 GHz. Especially when utilizing more threads, the several differences in hardware, like higher frequency and larger L3 cache, might lead to those observations. Generally, we need to note that for both architectures, the SoA, sliced-balanced configuration, performs worse with a maximum speedup of 7.2 than the other configuration with a maximum speedup of 12.7 for 16 threads utilized.
Figure 6.15 compares the speedup of AMD and INTEL for AoS - sliced-balanced and SoA - c18 configuration using the Verlet Lists Cells container.

For both configurations, AMD achieves a higher speedup for 8 and 16 threads than INTEL. The AMD CPU has eight memory channels per CPU that can be used for data access. Especially with increasing thread count, shifting the bottleneck from computational load per thread to memory access, the increased amount of memory channels and the larger L3 cache provide faster memory access for each thread than the INTEL CPU with only four memory channels and 35 MB of L3 cache. Generally, we have to note that compared to the calculated speedup in Figure 6.14, the Linked Cells container is better with a higher speedup, similar to the results from INTEL simulations. As stated in Subsection 6.1.1 is this

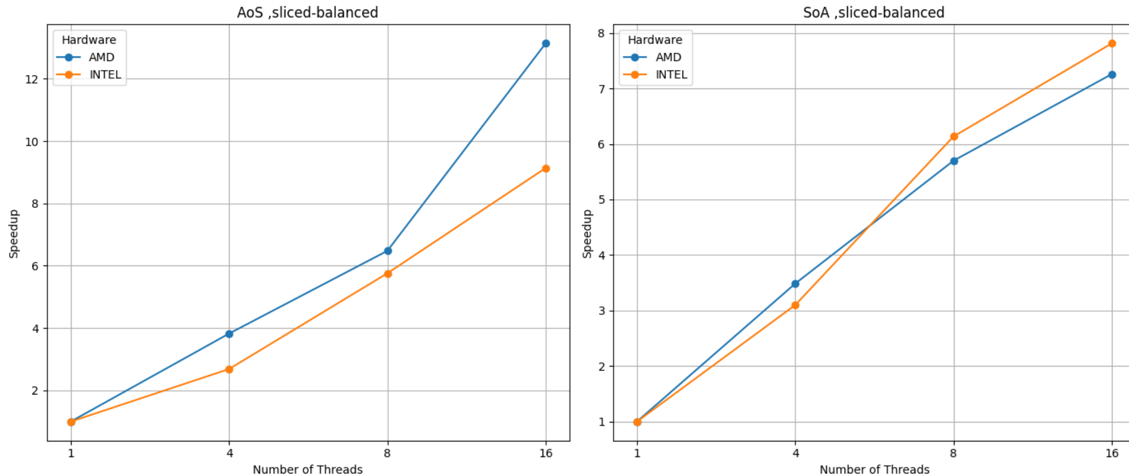


Figure 6.14: Comparing speedup for Linked Cells container between AMD and INTEL

due to the rebuilding of neighbour lists in every 10th iteration step, consuming remarkably more time and energy.
Next, we compare the energy speedup metric, introduced in Section 6.1.2, between AMD and INTEL. Figure 6.16 compares the energy speedup for the Linked Cells container, using again SoA - sliced-balanced and AoS - sliced-balanced configurations.

The key observation from Figure 6.16 is that AMD does have a higher energy speedup for both configurations. Even for the SoA - sliced-balanced simulation with 8 or 16 threads, where the INTEL runtime speedup is higher than the one for AMD, as can be seen in Figure 6.14, the energy speedup for AMD is higher. This suggests that the AMD CPU improves energy efficiency for more threads used than the INTEL one for the Linked Cells container. This could be due to several differences in the design of the AMD EPYC 7742. The CPU is based on a chiplet design, meaning that the CPU is build up from multiple modular chiplets, designed for a special purpose. This allows the AMD CPU to allocate energy resources for each chiplet dynamically. This ensures, that each chiplet operates at the optimal power for its current workload. The advantages of such a design architecture are optimized power, performance, and lower manufacturing costs [20]
Figure 6.17 does plot the energy speedup for the Verlet Lists Cells container. In difference to Figure 6.17, the INTEL CPU has a higher energy speedup for 1 and 4 threads used, which is because of the higher frequency range of the INTEL CPU from 2.6 GHz to 3.6 GHz, allowing to do more arithmetic operations in the same time than the AMD one to rebuild the neighbor lists. For 8 and 16, the AMD CPU has a higher energy speedup again and is constantly increasing for more threads used. Instead, the INTEL CPU has a drop in energy speedup for eight threads and slowly increases again for 16. This is because the AMD CPU has a larger L3 cache with 256 MB, leading to fewer cache misses and expensive memory accesses than when using the INTEL CPU. This gets important, especially when utilizing more threads, as the main energy is not spent on arithmetic operations but memory accesses. Important to note here is that even if the AMD EPYC 7742 has better performance and energy metrics than the INTEL Xeon E5-2697 v3 14-core Haswell, this does not mean that simulations on the AMD CPU are using less energy overall than simulations on the Intel CPU. They

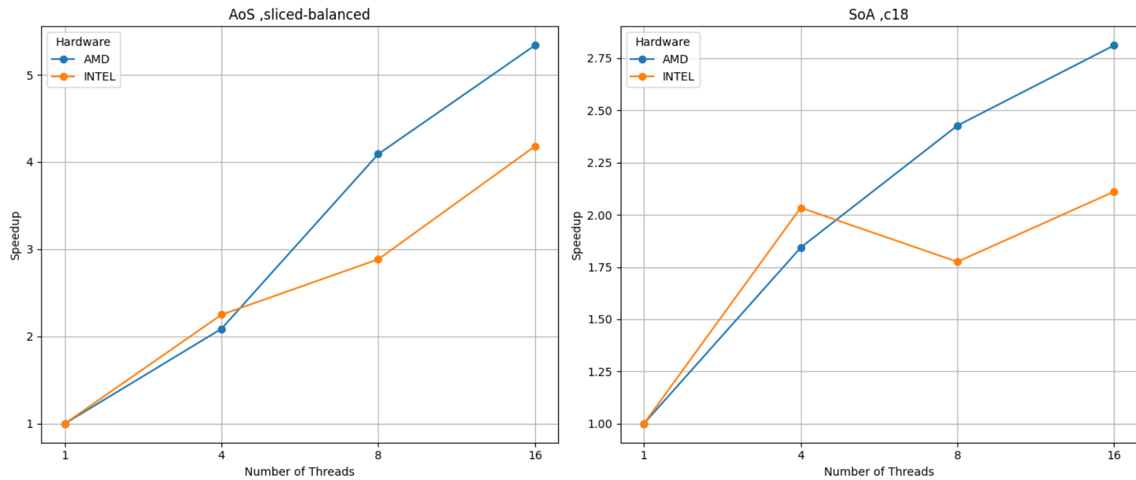


Figure 6.15: Comparing speedup for Verlet Lists Cells container between AMD and INTEL

only display the performance and energy improvements compared to the simulation using one thread. Figure 6.18 does plot the total energy consumed by the simulations on AMD and INTEL CPU when using the Linked Cells container. We can see that the AMD CPU consumes not less energy overall than the INTEL one. This can be explained by the AMD CPU having a higher thermal design power (TDP) of 225 Watts than the INTEL CPU with 145 Watts. Thermal design power represents, in this case, the average power the processor consumes when operating at the base frequency with all active cores. Only for a increasing number of threads, the AMD CPU does consume similar or less energy, due to several hardware advantages like the larger L3 cache, increased memory channels and chiplet design.

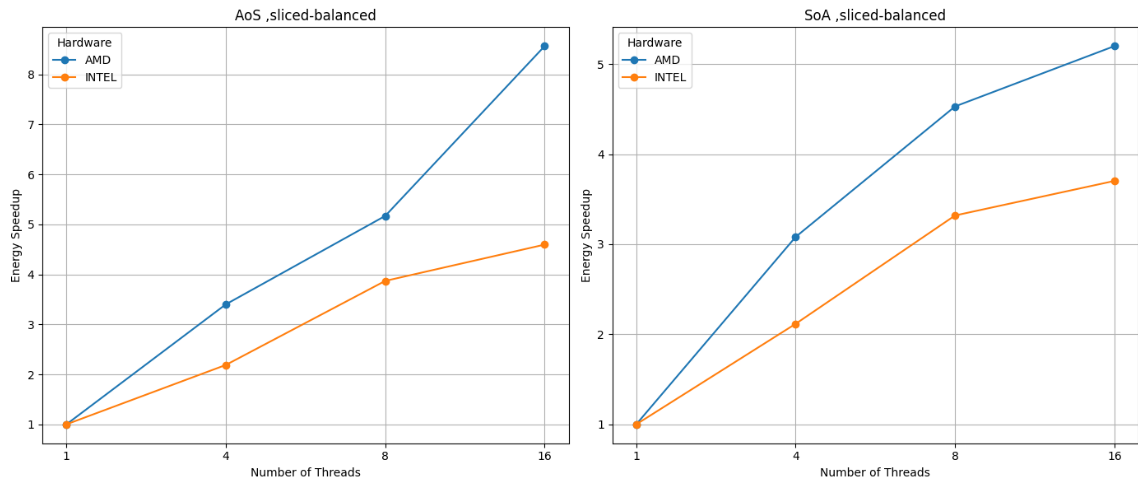


Figure 6.16: Comparing energy speedup for Linked Cells container between AMD and INTEL

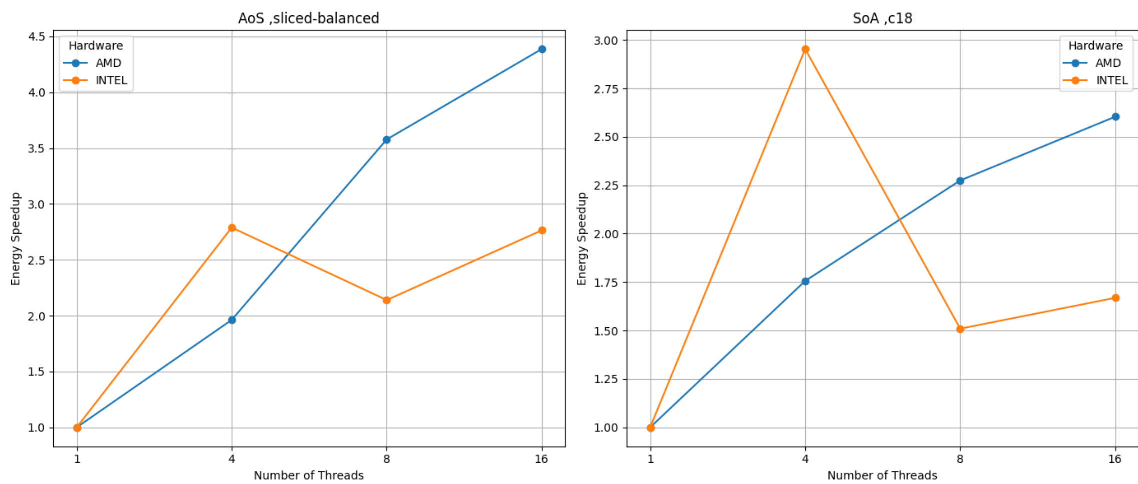


Figure 6.17: Comparing energy speedup for Verlet Lists Cells container between AMD and INTEL

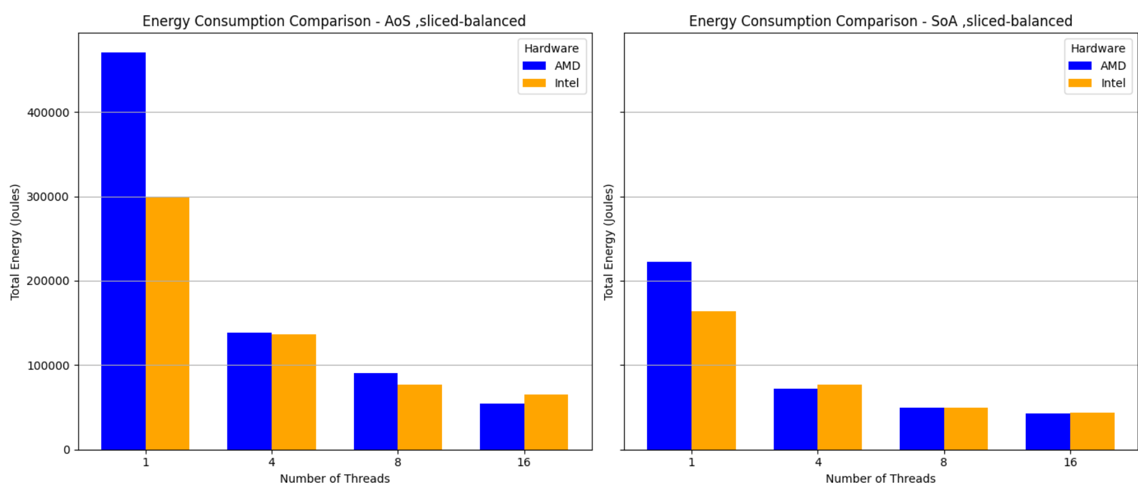


Figure 6.18: Comparing total energy consumed for Linked Cells container between AMD and INTEL

7 Conclusion

In this thesis, we presented the high-level software library "PMT: Power measurement toolkit", how it is integrated into AutoPas to be able to measure energy consumption on various hardware. Our objective was to compare different tuning configurations to get a relation between runtime and energy efficiency. Theoretical foundations for molecular dynamics simulations, including containers, data layouts and traversal algorithms, were outlined to provide some context for the work. The software library PMT was introduced and integrated into AutoPas. We compared runtime overhead, energy values and tuning results between PMT and the previously used RaplMeter implementation and observed differences in energy measurements between both implementations. It was determined to be due to the energy-pkg domain capturing the energy consumption values of gpu and other uncore components, which PMT filters out. This thesis could only be verified on a local machine but not on CoolMuc-2 Linux cluster, which lacked energy-gpu data, necessary for verification. This limitation should be considered when interpreting the results. Despite this, simulations on the CoolMuc-2 cluster allowed us to compare different simulation configurations in respect of time and energy consumption per iteration. We concluded that some data layout, containers and traversal algorithms are performing better speaking of energy consumption than others and noticed that the simulation configurations with a better runtime also consume less energy. This led us to the thesis, that tuning for energy efficiency is almost the same as tuning for time. To confirm this thesis we ran multiple simulations with tuning for energy and time and compared the results in respect of energy consumed, tuning configurations used and time to completion.

7.1 Future scope

Looking ahead, the bachelor thesis opens new possibilities for future research. When comparing different simulation configurations we noticed, that simulations using the Verlet Lists Cells container are consuming more energy and time than Linked Cells simulations. When calculating time and energy metrics, we also noticed that Linked Cells container can achieve higher speedups for energy and time for more threads used, than the Verlet Lists Cells one. This is probably caused by the rebuilding of neighbour lists which can take up to several seconds for our simulations. With this insight, future researches could focus on improving the algorithms for rebuilding neighbour lists, to reduce the energy consumption and runtime for the Verlet Lists Cells container. Future work could also focus on extending the power measurement backends for PMT. With the modular approach, explained in Section 3.1, PMT can easily be adapted to support new components by implementing new power measurement backends. One example is to include a subdirectory, for ARM Performance Monitoring Unit, to measure ARM CPUs' energy consumption. Another potential direction for future research could study the effect of frequency, briefly considered in Subsection 6.1.3, and how frequency scaling could be used for energy efficiency. Last, future research could

explore new metrics, that can be used for energy tuning like the Energy Delay Product introduced in Subsection 6.1.2 or some energy perfromance metric e.g. $\frac{Energy}{Flops}$.

8 Appendix

8.1 Appendix A: Simulation file for exploding liquid

```
# This yaml file is for single-site molecular simulation. Uncomment the Molecules option
# md-flexible compiled for multi-site molecules.
# container                       :  [LinkedCells, VarVerletListsAsBuild, VerletClusterL
# traversal                       :  [lc_sliced, lc_sliced_balanced, lc_sliced_c02, lc_c
# data-layout                     :  [AoS, SoA]
# newton3                         :  [disabled, enabled]
verlet-rebuild-frequency        :  10
verlet-skin-radius-per-timestep :  0.02
verlet-cluster-size             :  4
selector-strategy               :  Fastest-Mean-Value
tuning-strategies               :  [predictive-tuning]
tuning-interval                 :  6000
tuning-samples                  :  10
functor                         :  Lennard-Jones-AVX2
cutoff                          :  2
tuning-metric                   :  energy
energy-sensor                   :  rapl
box-min                         :  [0, 0, 0]
box-max                         :  [15, 60, 15]
cell-size                       :  [1]
deltaT                          :  0.00182367
iterations                      :  12000
boundary-type                   : [periodic, periodic, periodic]
Sites:
0:
epsilon                     :  1.
sigma                       :  1.
mass                        :  1.
# Uncomment below to run a multi-site simulation.
#Molecules:
#  0:
#    site-types                  :  [ 0 ]
#    relative-site-positions     :  [ [0, 0, 0 ] ]
#    moment-of-inertia           :  [ 1., 1., 1. ]
Objects:
CubeClosestPacked:
0:
```

```
particle-type-id           :  0
box-length                 :  [15, 6, 15]
bottomLeftCorner           :  [0, 27, 0]
particle-spacing           :  1.
velocity                   :  [0, 0, 0]
no-end-config                    :  true
no-progress-bar                  :  false
vtk-filename                     :  explodingLiquid
vtk-write-frequency              :  100
```

List of Figures

2.1 Linked Cells container force calculation [4] 4
2.2 Verlet Lists container [4] 5
2.3 Base steps [4] 6
2.4 Sliced domain [4] 7

3.1 General structure of the "PMT: Power Measurement Toolkit" library 9
3.2 Content of powercap directory on CoolMuc-2 12

5.1 Energy measurement process in AutoPas, demonstrating the key steps for calculating energy usage during iterations. 18
5.2 Time to completion, compared between PMT and RaplMeter 20
5.3 Energy consumption, compared between PMT and RaplMeter 20
5.4 Structure of the rapl power domains [12] 21
5.5 Comparison of RAPL power domains, supported by different Intel Processor Models [12] 22

6.1 Particle domain left after equilibrium state reached and on the right after forming clusters [3]. 25
6.2 Comparing energy consumption and time per iteration for Spinodal Decomposition with different configurations for Linked Cells container 26
6.3 Comparing energy consumption and time per iteration for Spinodal Decomposition with different configurations for Verlet Lists Cells container 27
6.4 Speedup obtained for Linked Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with 1000 iterations 29
6.5 Speedup obtained for Verlet Lists Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with 1000 iterations . . 30
6.6 Energy Speedup obtained for Linked Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition Equilibration simulation with only 1000 iterations 31
6.7 Energy Speedup obtained for Verlet Lists Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition Equilibration simulation with only 1000 iterations 32
6.8 Energy delay product (EDP) obtained for Linked Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with only 1000 iterations 33
6.9 Energy delay product (EDP) obtained for Verlet Lists Cells by using 1, 2, 4, 8, 12, 14, 28 threads for Spinodal Decomposition equilibration simulation with only 1000 iterations 34

6.10 Comparing time and energy per iteration for AoS - sliced-c02 and AoS - c18, using Spinodal Decomposition equilibration 35
6.11 Comparing Power and FLOPs for AoS - sliced-c02 and AoS - c18, using Spinodal Decomposition equilibration 36
6.12 Comparing time to completion between tuning for energy and time 37
6.13 Comparing total energy consumed between tuning for energy and time . . . 38
6.14 Comparing speedup for Linked Cells container between AMD and INTEL . 39
6.15 Comparing speedup for Verlet Lists Cells container between AMD and INTEL 40
6.16 Comparing energy speedup for Linked Cells container between AMD and INTEL 41
6.17 Comparing energy speedup for Verlet Lists Cells container between AMD and INTEL 41
6.18 Comparing total energy consumed for Linked Cells container between AMD and INTEL 42

List of Tables

5.1 Comparison of Iteration Time and Total Time for PMT and RapMeter implementation. 17
5.2 Comparison of energy values between PMT and RaplMeter on Intel Skylake architecture. 22
5.3 Results for pkg, cores, and gpu domain for exploding liquid simulation on Intel Skylake architecture. 23
5.4 Differences in tuning configurations 23
5.5 Energy values for different iterations on the CoolMUC-2 cluster 24

6.1 Differences in Configuration Between tuning for energy and time using one thread 35
6.2 Differences in Configuration Between tuning for energy and time using six thread 37

Bibliography

[1] Mozilla Contributors. Performance optimization. https://firefox-source-docs.mozilla.org/performance/perf.html, 2024. Accessed: 2024-09-19.

[2] Valerie Luna Dickson and Péter Tamás Sebok. Quantifying the power consumption of processes on linux using intel rapl. 2023.

[3] Vincent Fischer. Measuring and optimizing the energy efficiency of molecular dynamics simulations. Master's thesis, Technical University of Munich, May 2023.

[4] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2022.

[5] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 748–757, 2019.

[6] Höchstleistungsrechenzentrum Stuttgart (HLRS). Hpe apollo (hawk). https://www.hlrs.de/de/loesungen/systeme/hpe-apollo-hawk, 2024. Accessed: 2024-09-26.

[7] Scott A. Hollingsworth and Ron O. Dror. Molecular dynamics simulations for all. *Neuron*, 100(2):375–390.e8, 2018.

[8] Herbert Huber, Marco Sonnekalb, Rüdiger Geyer, Gerhard Baumeister, and Arndt Bode. CooLMUC-2: A supercomputing cluster with heat recovery for adsorption cooling. https://www.researchgate.net/publication/316352834_CooLMUC-2_A_supercomputing_cluster_with_heat_recovery_for_adsorption_cooling, 2017. Accessed: 2024-09-26.

[9] Intel Corporation. Intel xeon processor e5-2697 v3. https://www.intel.com/content/www/us/en/products/sku/81059/intel-xeon-processor-e52697-v3-35m-cache-2-60-ghz/specifications.html, 2024. Accessed: 2024-09-26.

[10] Intel Corporation. Was ist intel® turbo-boost-technik? https://www.intel.de/content/www/de/de/gaming/resources/turbo-boost.html, 2024. Accessed: 2024-09-26.

[11] Kashif Khan, Mikael Hirki, Tapio Niemi, Jukka Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3, 01 2018.

[12] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action : Experiences in using rapl for power measurements. 2018. Accessed: 05.09.2024.

[13] LRZ. Linuxstatistik2022. `https://doku.lrz.de/files/11484407/36865276/1/1689090180770/LinuxStatistik2022.pdf`, 2022. Accessed: 2024-09-26.

[14] Leibniz-Rechenzentrum (LRZ). Coolmuc-2. `https://doku.lrz.de/coolmuc-2-11484376.html/`, 2023. Accessed: 2024-10-03.

[15] Isaac Newton. *Philosophiae Naturalis Principia Mathematica*. Prostat apud plures bibliopolas, 1687.

[16] Maximilian Praus. `https://github.com/MaxPraus23/pmt-stable`, 2024. Accessed: 12.08.2024.

[17] Thomas Rauber, Gudula Rünger, and Matthias Stachowski. Performance and energy metrics for multi-threaded applications on dvfs processors. *Sustainable Computing: Informatics and Systems*, 17:55–68, 2018.

[18] Emma Tolley Stefano Corda, Bram Veenboer. `https://git.astron.nl/RD/pmt`, 2022. Accessed: 01.08.2024.

[19] Emma Tolley Stefano Corda, Bram Veenboer. PMT: Power Measurement Toolkit. `https://doi.org/10.48550/arXiv.2210.03724`, 2022. Accessed: 31.7.2024.

[20] Keysight Technologies. What is a chiplet and why should you care? `https://www.keysight.com/blogs/en/tech/sim-des/2024/2/8/what-is-a-chiplet-and-why-should-you-care#:~:text=A%20chiplet%20is%20a%20small,or%20a%20signal%20processing%20unit.`, February 2024. Accessed: 2024-09-26.

[21] Amirreza Yousefzadeh, Jan Stuijt, Martijn Hijdra, Hsiao-Hsuan Liu, Anteneh Gebregiorgis, Abhairaj Singh, Said Hamdioui, and Francky Catthoor. Energy-efficient in-memory address calculation. *ACM Trans. Archit. Code Optim.*, 19(4), sep 2022.

[22] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. Red alert for power leakage: Exploiting intel rapl-induced side channels. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 162–175, New York, NY, USA, 2021. Association for Computing Machinery.