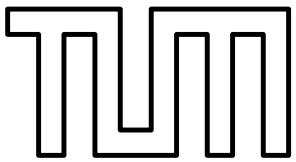


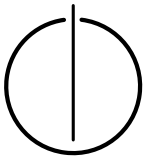
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

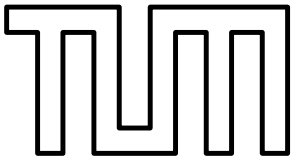
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Improving OpenMP Loop Scheduling in AutoPas

Mehdi Hachicha





SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

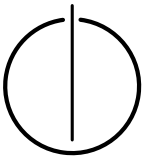
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Improving OpenMP Loop Scheduling in AutoPas

Verbesserung des OpenMP-Loop-Schedulings in AutoPas

Author:	Mehdi Hachicha
Supervisor:	Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome
Submission Date:	01.07.2024



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

Sfax, Tunisia, 01.07.2024

Mehdi Hachicha

Acknowledgments

With sincere gratitude, I thank Prof. Dr. Hans-Joachim Bungartz, and Samuel James Newcome, for granting me the privilege to work on the AutoPas project. I deeply appreciate Samuel's thorough advice, and valuable guidance throughout this thesis.

Special thanks to LB4OMP's author Jonas H. Müller Korndörfer for the availability, helpfulness, and crucial insights.

Kind regards to my family and friends for their support and company.

Abstract

Particle simulators commonly loop over particles and compute their interactions pairwise. Indeed, running the computations in parallel threads can significantly speed up the simulations. However, developers should look out for **load imbalance**: the uneven distribution of tasks between threads, such that some finish early and wait instead of taking up more work. For example, this may occur when tasks differ in complexity. To maintain optimal performance, simulators can thus benefit from load balancing techniques when scheduling their loop iterations to the threads. This thesis improves parallel loop scheduling in **AutoPas**: a particle simulation library. AutoPas employs **OpenMP**: a library that provides loop scheduling features via compiler directives. We integrate **Auto4OMP**, which is an OpenMP extension with advanced load balancing capabilities. To test the new setup, we ran different simulations with various parameters. In general, Auto4OMP performed close to the optimal standard OpenMP schedule, but the impact of its advanced scheduling techniques on non-homogeneous simulations was weaker than expected. That said, it showed promising results with homogeneous simulations.

Kurzfassung

Partikelsimulatoren durchlaufen Partikel häufig in einer Schleife und berechnen ihre Interaktionen paarweise. Das Ausführen der Berechnungen in parallelen Threads kann die Simulationen erheblich beschleunigen. Entwickler sollten jedoch auf **Lastverteilung** achten: sollten Lasten ungleichmäßig verteilt sein, sind einige Threads zu früh fertig, und warten, anstatt mehr Arbeit aufzunehmen. Dies kann beispielsweise auftreten, wenn sich die Aufgaben in ihrer Komplexität unterscheiden. Um eine optimale Leistung zu halten, können Simulatoren daher von Lastausgleichstechniken profitieren, wenn sie ihre Schleifeniterationen den Threads zuweisen. Diese Bachelorarbeit verbessert die parallele Schleifenplanung in **AutoPas**: einer Partikelsimulationsbibliothek. AutoPas nutzt **OpenMP**: eine Bibliothek, die Schleifenplanung über Compiler-Anweisungen bereitstellt. Wir integrieren **Auto4OMP**, eine OpenMP-Erweiterung mit fortgeschrittenen Lastausgleichsfunktionen. Um das neue Setup zu testen, haben wir verschiedene Simulationen mit unterschiedlichen Parametern ausgeführt. Generell lag die Leistung von Auto4OMP nahe am optimalen Standard-OpenMP-Schedule. Allerdings waren die Auswirkungen seiner fortschrittlichen Planungstechniken auf nicht homogene Simulationen geringer als erwartet. Es zeigte aber versprechende Ergebnisse bei homogenen Simulationen.

Contents

Acknowledgments iii

Abstract v

Kurzfassung vii

1. Introduction 1

2. Background 3
2.1. Parallelization . . . 3
2.2. OpenMP . . . 4
2.2.1. Chunk Size . . . 4
2.2.2. Scheduling Kinds . . . 4
2.2.3. Usage . . . 4
2.2.4. Limitations . . . 5
2.3. LB4OMP . . . 6
2.3.1. Scheduling Techniques . . . 6
2.3.2. Usage . . . 8
2.3.3. Limitations . . . 8
2.4. Auto4OMP . . . 9
2.4.1. DLS portfolio . . . 9
2.4.2. Expert Chunk Size . . . 9
2.4.3. Load Imbalance Metric . . . 10
2.4.4. Selection Methods . . . 10
2.4.5. Usage . . . 10
2.5. AutoPas . . . 11
2.5.1. Cutoff Radius . . . 11
2.5.2. Containers . . . 11
2.5.3. Traversals . . . 12
2.5.4. Auto-Tuning . . . 13
2.5.5. OpenMP in AutoPas . . . 13
2.5.6. MD-Flexible . . . 15

3. Implementation 17
3.1. Linking Auto4OMP to AutoPas . . . 17
3.2. Using Auto4OMP in AutoPas . . . 20

3.3. OpenMP Configurator 21
3.3.1. Features 21
3.3.2. Advantages 23
3.3.3. Alternatives 23
3.4. Testing 24

4. Results **25**
4.1. Homogeneous 25
4.2. Heterogeneous 27
4.3. Falling Drop and Exploding Liquid 29

5. Future Work **33**

6. Conclusion **35**

A. ExpertSel Fuzzy Logic **37**
A.1. Fuzzification 37
A.2. Expert Rules 39
A.3. Inference and Defuzzification 39

B. Tutorial: Testing AutoPas with Auto4OMP on LRZ's Linux Cluster **41**

List of Figures **45**

List of Tables **47**

Bibliography **49**

1. Introduction

From physics and engineering, to animation and game engines, countless disciplines leverage modern computing to run simulations: virtual experiments that replicate natural phenomena, e.g., fluid dynamics or particle physics. Such virtual sandboxes provide faithful results, without having to set up costly real-life experiments. They also enable scientists to explore otherwise inaccessible conditions in great detail, such as giant astronomical events or microscopic molecular dynamics. To handle larger scenarios, simulators may apply certain approximations or optimization techniques, to make the most use of available resources.

Although computers have been improving exponentially, careful scheduling is always needed to harness their full potential. To understand, consider that most CPUs employ multiple cores that work in parallel. On the extreme end of things, super-computers distribute their work over thousands of CPUs. A simulator must thus be able to deploy its work to multiple processing units, and address conflicts if said units access common memory. It must also distribute its tasks as evenly as possible, so that no unit runs out of work and remains idle, wasting valuable resources.

One very promising particle simulation library is **AutoPas**, which auto-tunes its traversal strategies at run-time [1]. It traverses its simulated space with **OpenMP** loops, which automatically dispatch their iterations to parallel threads.

This thesis attempts to improve parallel loops in AutoPas by incorporating **Auto4OMP**, an extended OpenMP with more sophisticated scheduling methods, tailored for load balance [2, 3]. Chapter 2 introduces the software at play: OpenMP, Auto4OMP and AutoPas. Chapter 3 walks through our implementation, and Chapter 4 presents our performance investigations and results.

2. Background

2.1. Parallelization

Resource intensive loops with numerous iterations can benefit from threads. These are processes that undertake subsets of iterations from the loops, and compute them in parallel. If the threads run on separate CPU cores for instance, performance is ideally improved. Moreover, the threads can work with a common memory (**Shared Memory Parallelization**). To avoid conflict when writing to a same address, they may require **synchronization**: temporarily locking write access to target regions of the memory. Parallel threads attempting to modify a synchronized datum must thus wait to acquire its lock. Indeed, parallelization is straight forward in ideal cases, where loop iterations have similar computational costs. In more complex scenarios however, parallel loops can perform sub-optimally without careful scheduling.

Should iterations vary significantly in execution times, **load imbalance** can occur: threads with easy tasks finish early and wait, instead of undertaking more load. For example, it may be beneficial to bundle short iterations into single chunks, while scheduling long iterations individually. Another approach would be for threads to dynamically request new tasks if they finish early. Although promising, such features require complex code. The next sections thus introduce **OpenMP**, a library that encapsulates thread and loop scheduling logic into simple compiler directives, and **LB4OMP**, an OpenMP extension with improved load balancing.

2.2. OpenMP

This chapter introduces standard OpenMP: a library bundled with LLVM/Clang [4], that handles the distribution of loop iterations over parallel threads.

2.2.1. Chunk Size

To reduce the overhead of scheduling the tiny iterations individually, loop iterations (tasks) are distributed to the threads in chunks [5]. The size of these chunks impacts performance, and can be tuned by the user.

2.2.2. Scheduling Kinds

Scheduling kinds are strategies to distribute the tasks. Standard OpenMP provides 5 scheduling kinds [5]:

- `static`: the task chunks are distributed to the threads in advance (round-robin).
- `dynamic`: threads start with one chunk each. When a thread completes its tasks, it requests a new chunk.
- `guided`: similar to dynamic, but the chunk size decreases over time. As shown in Equation 2.1, numerous iterations result in larger chunk sizes to reduce overhead. As the number of iterations closes in on the number of threads, they are schedule in smaller chunks to hopefully reduce load imbalance. The user-defined chunk size is the end-minimum.

$$\text{guidedChunk} \propto \frac{\text{Number of Remaining Iterations}}{\text{Number of Threads}} \tag{2.1}$$

- `auto`: an improved variant of guided scheduling.
- `runtime`: the scheduling kind and chunk size are read from the environment variable `OMP_SCHEDULE`. Users can also set them from application code with `omp_set_schedule(omp_sched_t kind, int chunk)`.

2.2.3. Usage

To use OpenMP, precede target loops with the following directive:
`#pragma omp parallel for schedule(kind,chunk)`
When the compiler encounters a loop with an OpenMP pragma, it transforms the code to start parallel threads and schedule the loop iterations to them. Listing 2.1 shows a simple OpenMP loop, and Listing 2.2 shows its transformation.

```
#include <omp.h>
#pragma omp parallel for schedule(dynamic,1)
    for (int i = 0; i < 10; i++)
        r += f();
```

Listing 2.1: OpenMP loop (original user code)

```
// Code transformed by the compiler:
__kmpc_begin();
__kmpc_fork_call(main_7_parallel_3, &r);
__kmpc_end();

// Function dispatched to the threads, written by the compiler:
void main_7_parallel_3(float *r) {
    // Initialize the scheduler.
    __kmpc_dispatch_init_4(
        schedule = kmp_sch_dynamic_chunked,
        from = 0, to = 9, step = 1, chunkSize = 1);

    // Keep requesting chunks and running them.
    while (__kmpc_dispatch_next_4(&lower, &upper))
        for (int i = lower; i <= upper; i++)
            r += f();
}
```

Listing 2.2: OpenMP loop transformed by the compiler [6] (Simplified code)

2.2.4. Limitations

Indeed, OpenMP provides a handy way to parallelize loops. However, it does not take into account the execution times of individual tasks, and can thus benefit from sophisticated scheduling techniques with load-balancing capabilities. The next subsection introduces one such extension: **LB4OMP**.

2.3. LB4OMP

LB4OMP is an extension of OpenMP. It provides a portfolio of scheduling techniques that aim to improve load balancing, and can outperform OpenMP's standard scheduling kinds under the right conditions [2].

2.3.1. Scheduling Techniques

Beside OpenMP's standard kinds, LB4OMP provides the following scheduling techniques [2, 7]:

- Static Scheduling with Steal enabled (`static_steal`): similar to static, but threads steal tasks from each other if they finish early.
- Trapezoid Self Scheduling (`trapezoidal`): similar to `guided`, but the chunk size decreases linearly.
- Tapering (`tap`): similar to `guided`, but uses a more optimal model to decrease the chunk size based on prior profiling data.
- Profiling (`profiling`): employs a `dynamic,1` schedule, and tracks the execution times of loop iterations and their scheduling overhead. Some scheduling techniques then use this information in subsequent runs to calculate optimal chunk sizes; namely `fsc`, `tap`, `fac`, `faca`, and `bold`. Set `KMP_PROFILE_DATA` to a path for storing or providing the profiling data.
- Fixed Size Chunk (`fsc`): uses profiling data from a previous run to calculate an optimal chunk size, and fixes it for the new run.
- Modified Fixed Size Chunk (`mfsc`): calculates a suitable chunk size based on the number of threads and iterations [8], and fixes it during the loop.
- Fixed Increase Self Scheduling (`fiss`): periodically increases the chunk size by a constant bump [9].
- Variable Increase Self Scheduling (`viss`): periodically increases the chunk size by a variable bump. The bump starts as half the initial chunk size, and gets halved each period [10].
- Random (`rnd`): the chunk size varies randomly within a specific range [10].
- Factoring (`fac`): schedules the chunks in batches to the threads. Uses prior profiling info to calculate an optimal chunk size for each batch of chunks. The first thread of each batch handles the calculation of the chunk size.
- Practical Factoring (`fac2`): an implementation of `fac`. New batches pack half the remaining iterations. Instead of using prior profiling information, this technique

starts with half the initial chunk size of standard guided scheduling. Since batches start large, the technique can balance loops with disproportionately long first iterations.

- Improved Factoring (`faca`): similar to `fac`, but each thread calculates its chunk size independantly. This relaxes synchronization requirements compared to normal factoring, where the threads wait on the chunk size calculation of new batches. This technique outperforms normal factoring under certain scenarios.
- Improved Practical Factoring (`fac2a`): a variant of `fac2`, combined with the optimisations of `faca`.
- Bold (`bold`): similar to `fac`, but starts with larger chunks to reduce the scheduling overhead. Like with `tap`, chunk sizes decrease optimally until settling at the calculated optimal chunk sizes for the batches. The optimal models are again estimated from prior profiling info.
- Trapezoid Factoring Self Scheduling (`tfss`): similar to `fac`, but the chunk size decreases linearly over time to reduce overhead [9].
- Weighted Factoring (`wf`): similar to `fac`, but chunks in a batch vary in size according to the specs of the computational unit that hosts the concerned thread. The threads are assigned **processing weights** based on the capabilities of their host cores. This technique is thus only needed on machines with heterogeneous cores.
- Adaptive Factoring (`af`): similar to `fac`, but optimizes the chunk size using information from the current run (execution times of previous iterations) instead of preexisting profiling data.
- Improved Adaptive Factoring (`af_a`): similar to `af`, but also considers the scheduling overhead of previous iterations to optimize the chunk size.
- Adaptive Weighted Factoring (`awf`): similar to `wf`, but processing weights vary dynamically based on the performance of the concerned threads. The weights are updated at the end of an application's time-step.
- Adaptive Weighted Factoring Variant B (`awf_b`): similar to `awf`, but the processing weights are updated at the end of each batch.
- Adaptive Weighted Factoring Variant C (`awf_c`): similar to `awf`, but the processing weights are updated at the end of each chunk.
- Adaptive Weighted Factoring Variant D (`awf_d`): similar to `awf_c`, but scheduling overhead influences the processing weights.
- Adaptive Weighted Factoring Variant E (`awf_e`): similar to `awf_b`, but scheduling overhead influences the processing weights.

2.3.2. Usage

To leverage LB4OMP, use an omp directive with `schedule(runtime)`, set `OMP_SCHEDULE` to the technique's acronym (the bracketed abbreviations listed above), and set `KMP_CPU_SPEED` to the CPU's frequency in MHz (integer, queried with `lscpu`). Beware, the technique abbreviations in the papers and Git's README are outdated. The up-to-date abbreviations listed above are from LB4OMP's master branch source code [11].

OpenMP's schedule setter may also be used, but requires adjusting the library's code. Internally, the function `__kmp_set_schedule()` assigns the schedule to a given thread. It is wrapped by `ompc_set_schedule()`, accessible by application code via `kmp.h`. However, the needed scheduling techniques must be added to the standard enum `omp_sched_t` in `omp.h` [12], using the numerical values from the internal type `kmp_sched_t` [11]. This thesis provides a custom LB4OMP package with the necessary modifications [13].

2.3.3. Limitations

Although LB4OMP improves load balancing, its complex techniques exert extra computational efforts. Their performance thus varies loop-wise, making the optimal schedule contingent on the computed scenario. The next section presents **Auto4OMP**, which automates the selection of the scheduling techniques.

2.4. Auto4OMP

Auto4OMP is a new release of LB4OMP. It offers automated, dynamic selection of LB4OMP's scheduling techniques [3]. To achieve this, Auto4OMP cycles through the techniques during run-time and tracks their performance. On average, Auto4OMP was shown to perform 11% better than OpenMP's standard auto scheduling [3].

2.4.1. DLS portfolio

Although sophisticated, some of LB4OMP's techniques require a prior profiling run of the loop. Others use heavyweight synchronization mechanisms, or require multiple time-steps to reach optimal performance [3]. Such traits complicate the run-time evaluation of the techniques, making them unfit for automated selection.

Considering these criteria, Auto4OMP adopts a subset of LB4OMP's scheduling techniques, deemed suitable for dynamic tuning. They form the Dynamic Loop Scheduling (DLS) portfolio ordered by load balancing overhead, illustrated in Figure 2.1. Note, the DLS list in the Auto4OMP paper is slightly out of date. The the below list reflects master branch Auto4OMP [14].

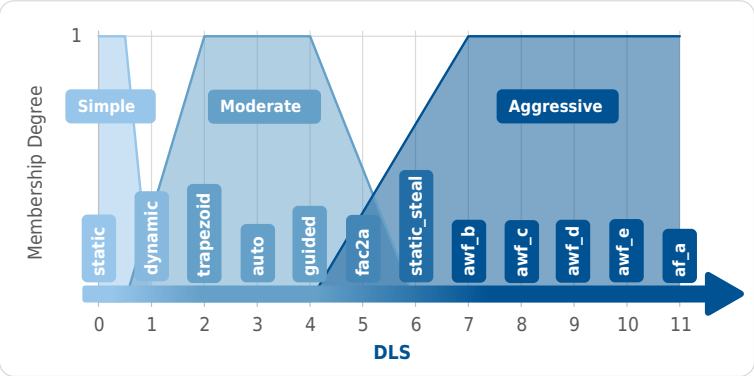


Figure 2.1.: Membership Functions of Auto4OMP's DLS Portfolio

2.4.2. Expert Chunk Size

By default, Auto4OMP calculates a suitable chunk size following Equation 2.2. The formula was shown to perform well in various experiments [3]. It incorporates the number of iterations N, the number of threads P, and the golden ratio $\phi \approx 1,618$. For the more advanced scheduling techniques, the chunk parameter serves as a minimal chunk size instead of OpenMP's standard minimum, which is 1.

$$\text{expertChunk} = \left\lfloor \frac{N}{2^{f+1}P} \right\rfloor, \quad \text{where } f = \left\lfloor \frac{1}{\phi} \log_2 \left(\frac{N}{P} \right) \right\rfloor \tag{2.2}$$

2.4.3. Load Imbalance Metric

The percent load imbalance metric, denoted LIB, measures the load imbalance within a group of threads following Equation 2.3. If one thread's execution time is significantly higher than average, LIB increases. Auto4OMP tracks the execution times of active threads, and keeps updating LIB during run-time. This measure serves to evaluate the performance of the scheduling techniques, and drives Auto4OMP's automated selection.

$$\text{LIB} = \left(1 - \frac{\text{Mean thread execution time}}{\text{Max thread execution time}}\right) \times 100 \tag{2.3}$$

2.4.4. Selection Methods

Auto4OMP provides various dynamic selection strategies [3, 15]. It cycles through the scheduling techniques in a time-stepping manner, where each time-step represents one execution instance of the entire target loop.

- RandomSel (`auto,2`): re-selects a random scheduling technique when thread loads become imbalanced. The probability P_j of re-selection at the end of a time-step increases proportional to LIB. If LIB crosses a given threshold, P_j surpasses 1 and re-selection is guaranteed.

$$P_j = \frac{\text{LIB}}{\text{threshold}} = \frac{\text{LIB}}{10} \tag{2.4}$$

- ExhaustiveSel (`auto,3`): tries all techniques consecutively, selects the best one, and uses it for a while before repeating. A high LIB triggers re-selection.
- Binary Search (`auto,4`): for each loop, this method cycles through the DLS techniques, ordered by their load balancing overhead. If LIB increases, the method steps to the right in the DLS list, to re-balance loads with a more aggressive technique. If LIB improves, the method steps to the left to reduce overhead. Steps start long, and get halved with each search trial. In other words, the method starts by jumping to the extremes of the DLS list, then gradually converges and ends up hovering around the optimal technique. The step length eventually reaches 0, concluding the search.
- ExpertSel (`auto,5`): selects the techniques with fuzzy logic. The method keeps track of the LIB, the execution times of loops, and their changes between time-steps. At the end of each time-step, it applies a fuzzy system with expert rules to select a scheduling technique for the next loop. Appendix A describes this method in detail.

2.4.5. Usage

To leverage Auto4OMP, use `schedule(runtime)` and set `OMP_SCHEDULE` to one of the bracketed schedules from the list above.

2.5. AutoPas

This chapter presents AutoPas: a sophisticated library that can be used by particle simulators. Users define custom particle classes and force equations, and interact with AutoPas as a black box [1]. During run-time, AutoPas iterates over the particles, performs force calculations pairwise, and updates the state of the simulation accordingly.

2.5.1. Cutoff Radius

Normally, full-precision pairwise calculations take quadratic time ($O(n^2)$). In many cases however, the influence of distant particles is small enough to neglect. AutoPas thus introduces a **cutoff radius**, set by users to represent the sphere of influence of the particles [1]. Indeed, this approximation removes the need to calculate forces outside a region of influence. However, pairwise distance calculations still need to be performed on all particle pairs to determine if they lie within the cutoff radius. AutoPas thus leverages different strategies to accelerate the lookup of a particle's neighbors.

2.5.2. Containers

To optimize the identification of neighbors, AutoPas stores particles in different types of containers [1]. For instance, **linked cells containers (LC)** split the simulated space into cubes, called linked cells, larger than the cutoff radius. All candidate neighbors of a target particle thus lie in adjacent cells, bringing the complexity down to $O(n)$. Figure 2.2 illustrates a 2D version of this container. In addition, AutoPas may maintain a list of neighbors, dubbed **Verlet list**, for each particle. Optionally, it associates the neighbor lists with their corresponding linked cells, in a **Verlet Lists Cells (VLC)** container. Other containers include **Verlet Cluster Lists (VCL)**, which clumps the particles into clusters with single Verlet lists. This container splits the simulated space into towers that sort their particles in arrays along the z-axis. With this vertical ordering, chains of consecutive particles in a tower form the clusters. The contents of the cells, towers, or Verlet lists, are updated regularly during the simulations.

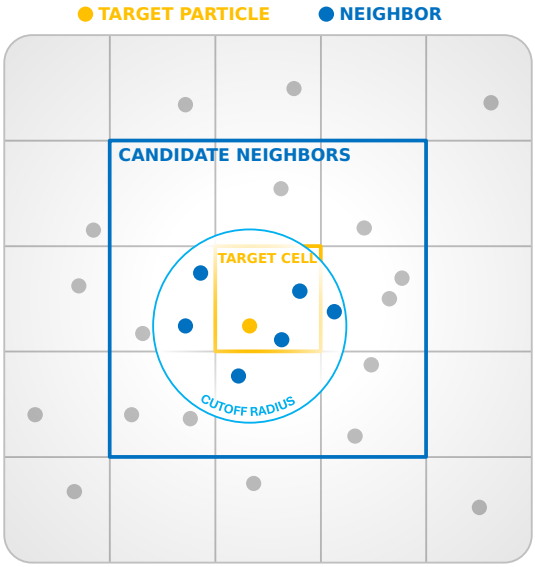
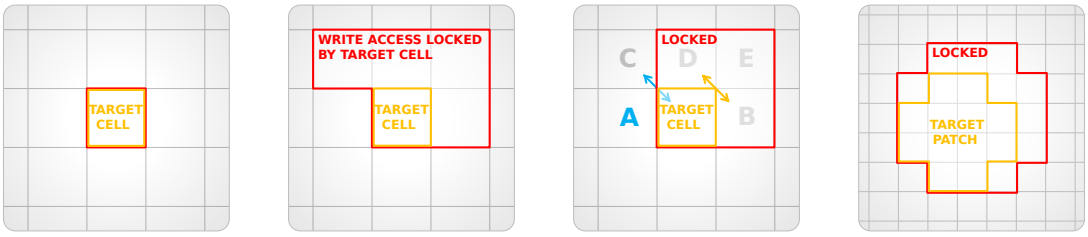


Figure 2.2.: Linked Cells Container

2.5.3. Traversals

To optimize the traversal of containers, AutoPas employs different strategies [1]. Different linked cells for instance are handled by parallel threads, synchronized via domain coloring (**color-based traversals**). To traverse the cells, AutoPas implements different strategies, each characterized by its **base-step**. The types of base steps are listed bellow, and illustrated in Figure 2.3:

- c01: each thread iterates over the particles of its assigned cell, and applies to them the forces exerted by their neighbors. Since candidate neighbors are not written to, threads do not need to lock writing access outside their target cells.
- c18: with symmetric forces, AutoPas can leverage the **Newton3** option: when a particle calculates the influence from its neighbor, it directly applies the calculated force to the neighbor too. Indeed, this cuts computational efforts in half. Threads can now disregard half the adjacent cells, influences from which will be added to the target cell by their corresponding threads. However, parallel writing to neighbor cells requires synchronisation, represented by the red cells in Figure 2.3b.
- c08: to further shrink the locked area, a thread can undertake interactions between two of its adjacent cells (B and D in Figure 2.3c). This allows threads to disregard interactions with one more adjacent cell (C), which will be handled by a different loop (A).
- c04: to further relax synchronization requirements, steps are assigned patches of cells instead of single target cells. Figure 2.3d illustrates an example patch.



(a) c01 (b) c18 (c) c08 (d) c04

Figure 2.3.: AutoPas Traversal Base Steps

To traverse Verlet cluster list containers, AutoPas may perform a **cluster iteration**, and assign the towers to parallel threads. All these traversals are hierarchized in Figure 2.4.

2.5.4. Auto-Tuning

Whether a traversal's advantages outweigh its overhead depends on the simulated scenario. AutoPas thus leverages **auto-tuning** to dynamically select optimal strategies [1]. It performs periodic tuning phases, in which it tests its different containers and traversals to select the fastest configuration. Although auto-tuning presents its own overhead, it can improve overall performance [1].

2.5.5. OpenMP in AutoPas

To traverse its containers, AutoPas uses OpenMP loops, as shown in Listing 2.3. An OpenMP wrapper is used: it replaces OpenMP's functions with stubs in case OpenMP is not linked, and contains a wrapper for the omp pragma (`AUTOPAS_OPENMP()`) Currently, it employs dynamic scheduling with chunk size 1, which is sub-optimal in many scenarios. This thesis investigates the effects of chunk size on different simulations, and leverages Auto4OMP to improve scheduling of the parallel loops.

```
1 #include "autopas/utils/WrapOpenMP.h"
2     AUTOPAS_OPENMP(parallel for schedule(dynamic,1) collapse(3))
3     for (unsigned long z = start_z; z < end_z; z += stride_z)
4       for (unsigned long y = start_y; y < end_y; y += stride_y)
5         for (unsigned long x = start_x; x < end_x; x += stride_x)
6           loopBody(x, y, z);
```

Listing 2.3: AutoPas OpenMP traversal loop (simplified) [16]

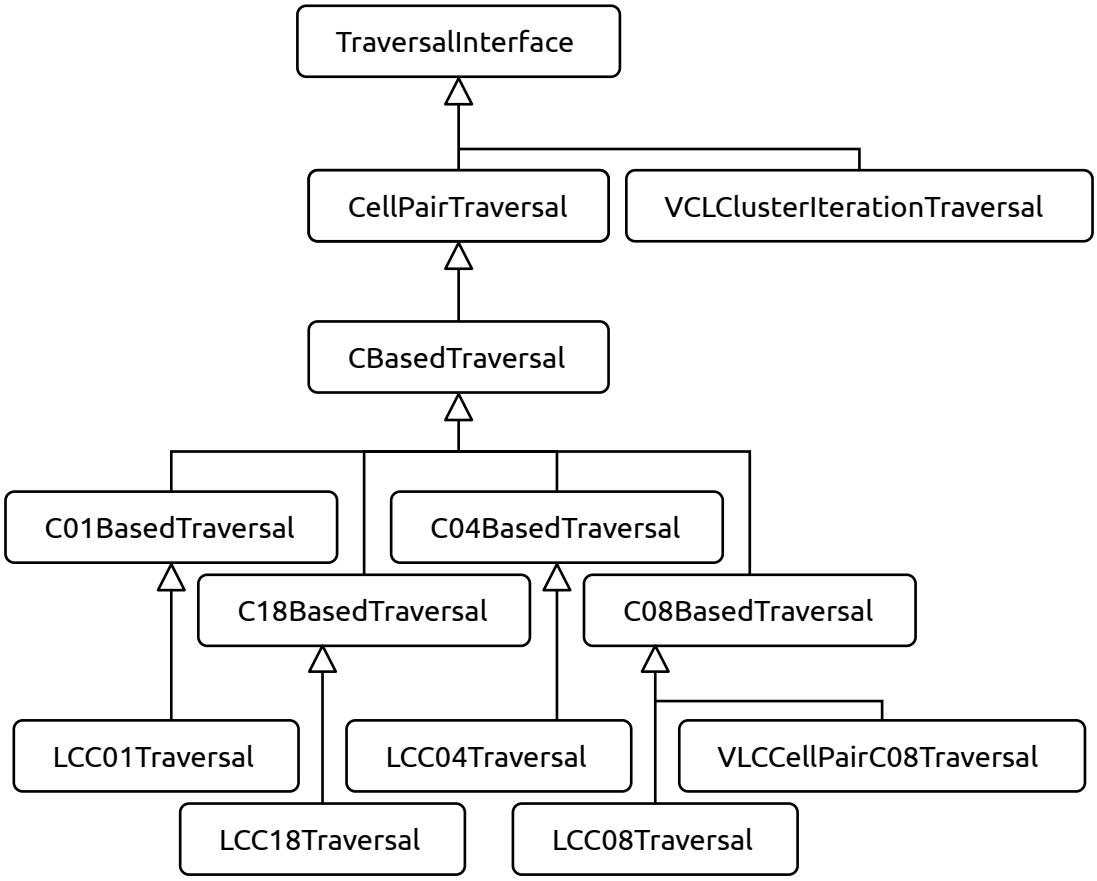


Figure 2.4.: AutoPas Traversal Class Hierarchy.
The concrete traversals are dubbed lc_c01, lc_c18, lc_c08, lc_c04, vlp_c08 (a.k.a. VLCCellPairC08Traversal) and vcl_clusterIteration.

2.5.6. MD-Flexible

MD-flexible is a molecular dynamics simulator that showcases AutoPas. It takes a yaml input defining the AutoPas settings and the initial state of the simulation, runs the simulation for a given number of iterations, and outputs its execution time.

3. Implementation

3.1. Linking Auto4OMP to AutoPas

Originally, AutoPas looks for standard OpenMP with CMake's FindOpenMP [17]. This module finds the standard libomp.so, bundled with Clang. AutoPas then links with the imported target `OpenMP::OpenMP_CXX`.

To link Auto4OMP, we first downloaded LB4OMP's master branch. Note that the original Auto4OMP release (LB4OMP v0.1) may result in build errors due to an outdated initializations of atomic ints. It is thus safer to use the master branch. From the download, we created 3 archives and bundled them with AutoPas:

- `libs/LB4OMP-master.zip`: untouched master branch Auto4OMP.
- `libs/LB4OMP-custom.zip`: available at [13], contains the following modifications:
  - `kmp_runtime.cpp`: corrects the index that maps the scheduling kind in `__kmp_set_schedule()`, and passes the chunk size if it selects an Auto4OMP method.
  - `kmp_settings.cpp`: moves certain LB4OMP functions into the OpenMP 5 condition. They are undefined for older versions, resulting in build errors with master branch LB4OMP.
  - `omp.h.var`: extends the standard enum `omp_sched_t` with LB4OMP's techniques, to provide them to the schedule setter `ompc_set_schedule()` defined in `kmp.h`.
  - Miscellaneous adjustments that address build warnings.
- `libs/LB4OMP-debug.zip`: based on LB4OMP-custom, but prints messages to confirm its run-time presence.

Users can choose one of the above archives, or a GIT download, with CMake options defined in a custom CMake module [18]. To set the options, simply add `-D<var>=<val>` to the AutoPas CMake command:

- By default, AutoPas chooses `LB4OMP-custom.zip`.
- `AUTOPAS_AUTO4OMP=ON|OFF` enables or disables Auto4OMP.
- `AUTOPAS_LB4OMP=ON|OFF` exposes LB4OMP's individual selection techniques.

- `AUTOPAS_AUTO4OMP_DEBUG=OFF|ON` loads `LB4OMP-debug.zip`.
- `AUTOPAS_AUTO4OMP_MASTER_FORCE=OFF|ON` loads `LB4OMP-master.zip`.
- `AUTOPAS_AUTO4OMP_GIT_FORCE=OFF|ON` downloads Auto4OMP from Git.
- `AUTOPAS_AUTO4OMP_GIT_TAG="master"` specifies the git branch or commit.
- `AUTOPAS_AUTO4OMP_INSTALL_DIR="/path/to/auto4omp-build/install"` specifies an install directory for the custom target auto4omp (not required, run explicitly with `make auto4omp`).

The custom module loads the chosen Auto4OMP package and imports its CMake targets. For a seamless user experience, it attempts automatic system checks to set the following variables, expected by the library:

- `LIBOMP_HAVE___BUILTIN_READCYCLECOUNTER`: indicates that the system provides a cycle counter register: a high resolution timer used for time measurements by LB4OMP. To find it, the module builds and runs `CycleCounterQuery.cpp`, which queries for the timer with `__builtin_readcyclecounter()`.
- `LIBOMP_HAVE___RDTSC`: also a high resolution timer, queried with the command `lscpu`. If no high res timer is found, AutoPas warns, and Auto4OMP's build fails.
- `LIBOMPTARGET_NVPTX_COMPUTE_CAPABILITIES`: If CUDA is installed, LB4OMP will automatically link it. However, CUDA dropped support for compute capability 3.5, set by default in LB4OMP. To avoid build errors, CMake must thus specify compute 5.3, supported by most modern Nvidia GPUs [19].
- `LIBOMPTARGET_NVPTX_ALTERNATE_HOST_COMPILER`: CUDA's NVCC requires gcc 12 or less. CMake attempts to find it, and warns if it fails. If CUDA's present, users should install `gcc-12` and pass its path to the CMake variable `AUTOPAS_NVCC_GNUC_PATH`.
- `LIBOMP_HAVE_X86INTRIN_H`: an Intel header that provides access to to the CPU's high resolution timers.
- `OPENMP_LLVM_LIT_EXECUTABLE` and `OPENMP_FILECHECK_EXECUTABLE`: To suppress warnings, the module attempts to find LLVM-lit and Filecheck. If the lookup fails, it instructs to pass their paths to `AUTOPAS_LLVM_LIT_EXECUTABLE` and `AUTOPAS_FILECHECK_EXECUTABLE`.

Finally, the imported CMake target `omp` (alias `auto4omp::omp`) is linked to the AutoPas target [20]. This target represents Auto4OMP's newly built libomp.so. The custom module also adds the target `auto4omp`, which runs a full `make install` in the Auto4OMP build directory. Listing 3.1 shows a very minimal version of the module.

Ideally, `cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ ..` works without extra options. Beware, Clang should be used if Auto4OMP is on; modern GCC produces build errors. If CMake needs manual values, it prints the corresponding instructions during its configuration. The CMake module `autopas_auto4omp.cmake` thoroughly documents the available options and the rationale behind its features [18].

```cmake
# cmake/modules/autopas_auto4omp.cmake: import Auto4OMP.
## Set the required CMake variables.
option(LIBOMP_HAVE___BUILTIN_READCYCLECOUNTER "" ON)
set(LIBOMPTARGET_NVPTX_COMPUTE_CAPABILITIES 53 CACHE INTERNAL "")
set(LIBOMP_HAVE_X86INTRIN_H ON CACHE INTERNAL "")
set(OPENMP_STANDALONE_BUILD ON)
## Etc.

## Load the bundled Auto4OMP.
include(FetchContent)
FetchContent_Declare(
    auto4omp
    URL ${AUTOPAS_SOURCE_DIR}/libs/LB4OMP-custom.zip
    URL_HASH MD5=c62773279f8c73e72a6d1bcb36334fef # md5sum
)

## Import Auto4OMP's targets (omp, omptarget, etc.)
FetchContent_MakeAvailable(auto4omp)
add_library(auto4omp::omp ALIAS omp) # Renames omp to auto4omp::omp.

# In cmake/modules/autopas_OpenMP.cmake, include the above module.
include(autopas_auto4omp)

# In src/autopas/CMakeLists.txt, link Auto4OMP to AutoPas.
target_link_libraries(autopas PUBLIC auto4omp::omp)
```

Listing 3.1: AutoPas CMake config for Auto4OMP (simplified)

With this setup, Auto4OMP automatically builds its own libomp.so and links it with AutoPas. Auto4OMP does not have to be built separately. After building a simulator however, it may be necessary to prepend Auto4OMP's runtime directory to the environment variable `LD_LIBRARY_PATH`. This tells the dynamic linker to prioritise the newly built libomp over the standard one shipped with LLVM. This is especially needed in environments such as Linux clusters, where the variable may already contain an explicit path to standard OpenMP. Simply run the following command before starting the simulator:

```
export LD_LIBRARY_PATH=/path/to/AutoPas/build/_deps/auto4omp-build/
    runtime/src:$LD_LIBRARY_PATH
```

To verify, run `ldd <executable>`, where <executable> is a path to the built simulator. `AutoPas/build/_deps/auto4omp-build/runtime/src/libomp.so` should be linked.

Beware, the standard CMake find module for OpenMP still runs. Skipping it causes Auto4OMP to fail to schedule parallel loops, and perform as if OpenMP is off. We suspect the module links extra libraries required for OpenMP, but not bundled with Auto4OMP. For the purposes of this thesis, we stopped the investigation here.

3.2. Using Auto4OMP in AutoPas

With the Auto4OMP library linked, we switched to `schedule(runtime)` in the OpenMP loop from Listing 2.3. To use Auto4OMP's selection methods or scheduling techniques, one either exports `OMP_SCHEDULE` in the terminal, or calls `omp_set_schedule()` from application code. For flexibility, we opted for the latter option, and added an AutoPas wrapper for OpenMP's schedule setter [21]:
`autopas_set_schedule(omp_sched_t kind, int chunk)`.

Indeed, this enables AutoPas to manipulate the schedule dynamically, and potentially auto-tune it. It also allows other loops to use separate runtime schedules, instead of the global `OMP_SCHEDULE`. For schedule customisation and ease of use, the next section introduces the **OpenMP configurator**.

3.3. OpenMP Configurator

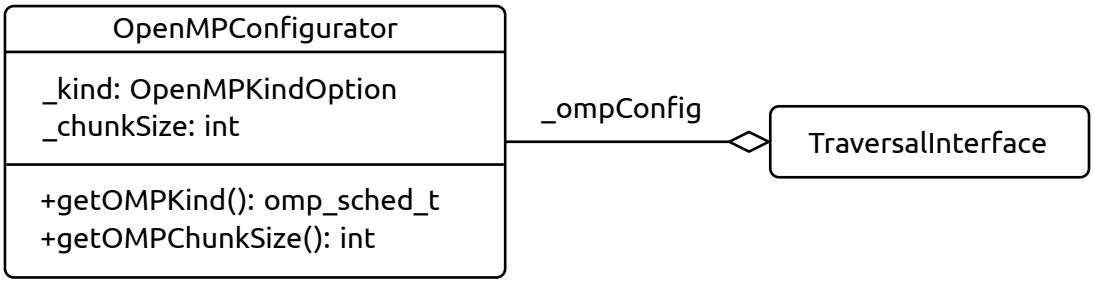


Figure 3.1.: The OpenMP configurator's UML class diagram

Each instance of an AutoPas traversal is assigned an OpenMP configurator. It contains the scheduling kind and chunk size for its host to use, and can be tuned by AutoPas. This section explains the configurator's interface and features.

3.3.1. Features

`OpenMPKindOption` : represents the different scheduling kinds, techniques and selection methods. Although LB4OMP's own `kmp_sched_t` can be used, an AutoPas wrapper enables extra customization. E.g., AutoPas may define its own scheduling technique in the future, or set custom names for the different kinds. The following options are enumerated:

- `omp_runtime` : reads from `OMP_SCHEDULE` , manually set by the user. If set, AutoPas ignores the configurator and prioritises the runtime scheduling variables provided to the thread.
- `omp_static` : standard static scheduling.
- `omp_dynamic` : standard dynamic scheduling.
- `omp_guided` : standard guided scheduling.
- `omp_auto` : standard auto scheduling.
- `auto4omp_randomsel` : Auto4OMP's RandomSel, i.e., `auto,2`.
- `auto4omp_exhaustivesel` : Auto4OMP's ExhaustiveSel, i.e., `auto,3`.
- `auto4omp_binarySearch` : Auto4OMP's Binary Search, i.e., `auto,4`.
- `auto4omp_expertsel` : Auto4OMP's ExpertSel, i.e., `auto,5`.

- `lb4omp_<tech>`: fixes an LB4OMP technique. Replace <tech> with one of the abbreviated names listed in Subsection 2.3.1. Only available if the CMake option `AUTOPAS_LB4OMP` is on.

`getOMPKind()` and `getOMPChunkSize()` convert the configurator's kind and chunk size to values passable to OpenMP's schedule setter. With `auto4omp_randomsel` for instance, the functions return `auto` and `2` respectively.

In addition, users may set a default kind and chunk size for configurators to initialize with, using the AutoPas container's `setOpenMPDefaultKind(OpenMPKindOption kind)` and `setOpenMPDefaultChunkSize(int chunk)`. The chunk size defaults to `0`, in which case Auto4OMP's expert chunk size is used (`1` is used if Auto4OM is off). If it has a positive value, the chunk size overrides the expert chunk. MD-Flexible's options `--openmp-kind` and `--openmp-chunk-size` set these defaults.

Finally, a custom inline schedule setter is provided. It takes an OpenMP configurator, and uses its current attributes to set OpenMP's runtime schedule, adopted by subsequent OpenMP loops with a `schedule(runtime)` clause:
`autopas_set_schedule(OpenMPConfigurator ompConfig)`

Listing 3.2 shows the resulting traversal loop, which sets OpenMP's runtime schedule from the configurator. The VCL cluster iteration traversal was adjusted separately, since it is not color-based. Note, c04 traversals did not adopt the OpenMP configurator's schedule, perhaps they override the concerned traversal loop at some point. For the purposes of this thesis, we stopped out investigation here.

```
1  #include "autopas/utils/WrapOpenMP.h"
2
3      // Sets OpenMP's runtime schedule from the OpenMP configurator.
4      autopas_set_schedule(TraversalInterface::_ompConfig);
5
6      // Use the configured runtime schedule in the OpenMP loop.
7      AUTOPAS_OPENMP(parallel for schedule(runtime) collapse(3))
8      for (unsigned long z = start_z; z < end_z; z += stride_z)
9        for (unsigned long y = start_y; y < end_y; y += stride_y)
10         for (unsigned long x = start_x; x < end_x; x += stride_x)
11           loopBody(x, y, z);
```

Listing 3.2: AutoPas OpenMP traversal loop with Auto4OMP (simplified) [22]

3.3.2. Advantages

The OpenMP configurator has the following uses:

- It encapsulates all OpenMP-related features and configurations in one object. In the future, the thread number and other OpenMP features can also be added to the configurator.
- Each class can aggregate an OpenMP configurator and tune its attributes locally, as opposed to one set of scheduling variables global to the AutoPas container. This also results in cleaner code compared to adding conditional statements before OpenMP loops. Internally, OpenMP's schedule setter assigns the scheduling variables to the calling thread [23].
- AutoPas can add conditions to the configurator's OMP getters. For instance, it can modify the getters to return a constant schedule during tuning phases, to avoid conflict with Auto4OMP's selection methods.
- AutoPas can wrap, rename, provide or hide the scheduling options. It may also define its own scheduling techniques in the future, for example one with an auto-tuned chunk size. Simply add an extra option after the existing enumerations in `OpenMPKindOption.h`. Beware not to use OpenMP's schedule setter with new options; either modify `autopas_set_schedule()`, or adjust the configurator's OMP getters.

3.3.3. Alternatives

Instead of the OpenMP configurator, the AutoPas container can define plain variables for the scheduling kind and chunk size. Although this setup seems simpler, it only provides the minimal feature of tuning global scheduling variables. Any additional features or customizations have to be added as raw code distributed in the AutoPas container, gradually complicating things. To maintain good readability for developers interested in OpenMP's features, we opted to encapsulate them in a single class.

3.4. Testing

We tested Auto4OMP's performance on CoolMUC-2 [24] with 28 threads and the following MD-Flexible inputs:

- `homogeneous`: a homogeneous cluster of $150 \times 150 \times 150$ particles. To isolate the effects of OpenMP's schedules, this simulation is static; it uses infinite time precision (delta_t = 0), such that its state does not evolve over the simulated 500 iterations.
- `heterogeneous`: a cluster of 30000 particles with varying densities. Also static, and simulates 500 iterations.
- `explodingLiquid` and `fallingDrop`: provided by MD-Flexible, illustrated in Figure 3.2 and Figure 3.3.

In addition, we added variants of the above inputs with single traversal types [25]. We ran each input on a dynamic schedule with different chunk sizes, ranging from 1 to 4096 on a logarithmic scale. In addition, we tested Auto4OMP's 4 selection methods, and fixed the optimal LB4OMP technique selected by them for some of the inputs. Appendix B provides steps to replicate the result of this thesis.

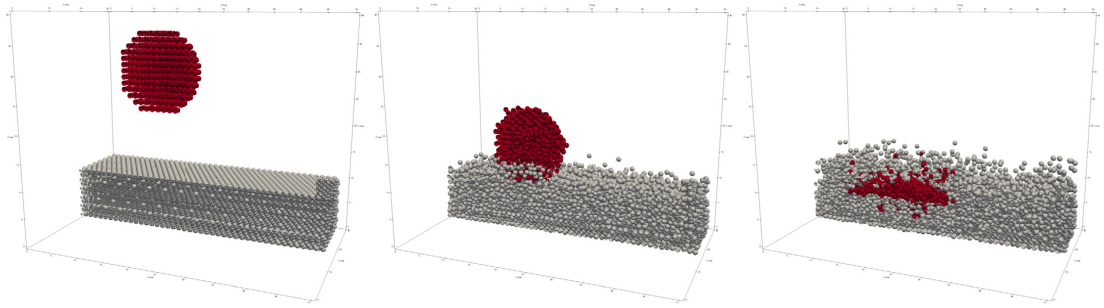


Figure 3.2.: MD-Flexible's fallingDrop simulation [1]

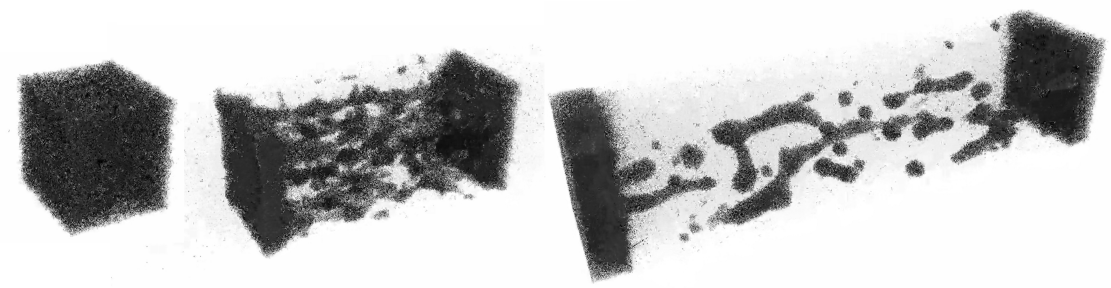


Figure 3.3.: MD-Flexible's explodingLiquid simulation [26]

4. Results

With non-homogeneous simulations, Auto4OMP at best matched the performance of a `dynamic` schedule with optimal chunk size. Indeed, the impact of its extended scheduling techniques on non-homogeneous scenarios was weaker than expected. That said, it did show promising results with homogeneous scenarios.

4.1. Homogeneous

As predicted, performance depended on chunk size. With linked cells c01, c18 and c08 traversals, the optimal `dynamic` chunk size was between 256 and 512, and performed around 10% faster than `dynamic,1`. Large chunk sizes performed significantly worse, and we expect them to eventually plateau at the run time of a sequential loop. Auto4OMP's `exhaustiveSel` outperformed the `dynamic` optimum by a further 12%, and internally maintained a `static_steal` schedule with chunk sizes 6 or 12 for the lc_c08 traversal. Fixing a `static_steal` schedule resulted in a 5% improvement compared to `exhaustiveSel`. Auto4OMP's other selection methods, as well as standard `auto`, performed about as fast as the optimal `dynamic` schedule. It is noteworthy that the optimal `dynamic` chunk size was not the Auto4OMP expert chunk. Figure 4.1 plots the above results for the linked cells homogeneous input with base step c08.

Note, fixing chunk size 512 for `static_steal` worsened its performance, the above graph represents the default chunk size 1. Interestingly, `exhaustiveSel` only performed a single selection round for lc_c01. This is likely because such a simple homogeneous scenario naturally results in balanced loads, such that re-selection is never triggered.

The `dynamic` curves look similar for the Verlet Lists Cells and Verlet Cluster Lists containers (respective traversals vlp_c08 and vcl_clusterIteration). This time however, Auto4OMP did not outperform the `dynamic` schedule. The `dynamic` optimum was also in the neighborhood of the expert chunk, as illustrated in Figure 4.2. Note, we have an old run of the vlp_c08 input with similar form, but higher overall times. It was likely due to an outdated commit or a mis-configuration, and was therefore excluded from the average.

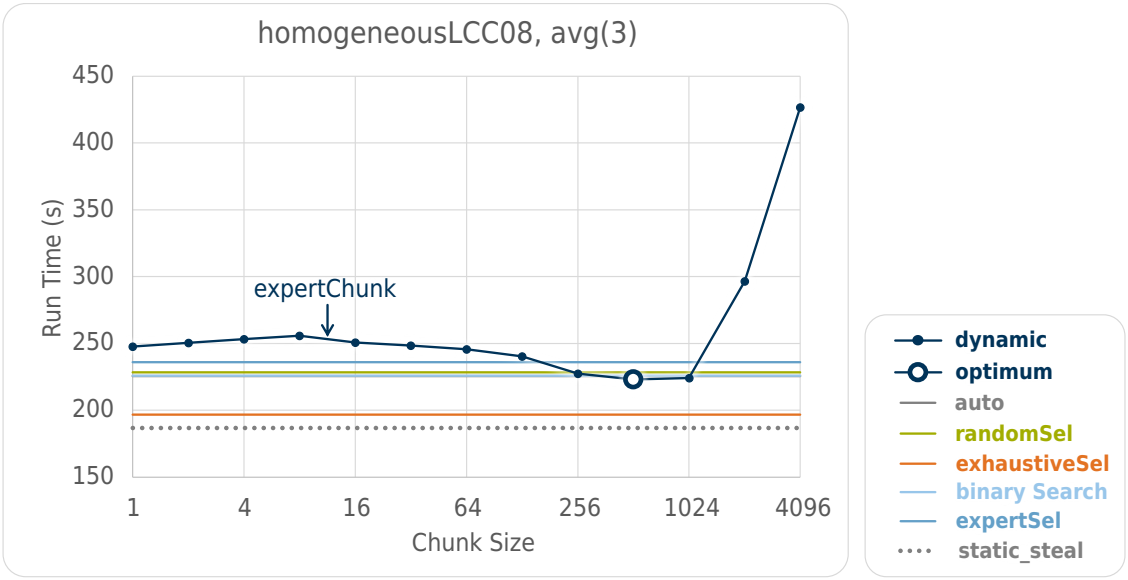


Figure 4.1.: MD-flexible performance with a homogeneous lc_c08 input.
Each data point is an average of 3 runs. The horizontal lines represent 3-run averages with the corresponding schedules from the legend.

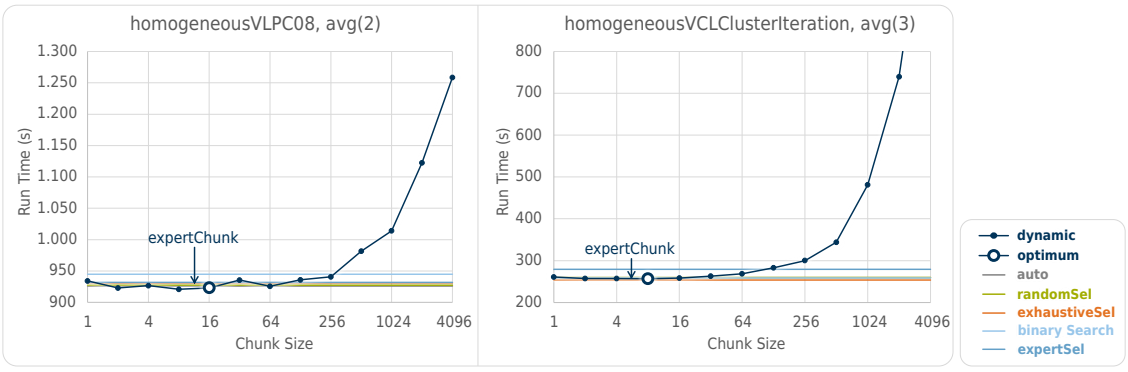


Figure 4.2.: MD-flexible performance with homogeneous vlp_c08 and vcl_clusterIteration.
Each data point represents an average of multiple runs, on a `dynamic` schedule.
The titular avg(i) means i runs were averaged per data point.

Furthermore, we tested small versions of the homogeneous lc_c08 input with various densities. Execution times differed as expected, but the optimal chunk size for the `dynamic` schedule was smaller for dense inputs. Figure 4.3 illustrates these results.

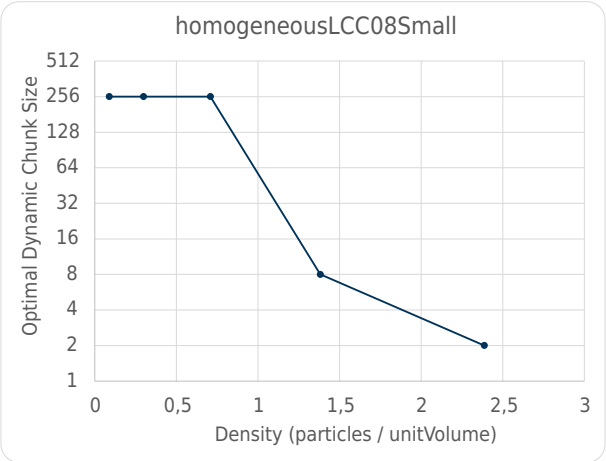


Figure 4.3.: MD-flexible performance with small homogeneous lc_c08 inputs.
Each data point represents the most performant chunk size of one input with a specific density, on a `dynamic` schedule.

4.2. Heterogeneous

The c18 and c08 heterogeneous inputs produced `dynamic` curves similar in shape to the homogeneous tests. However, Auto4OMP only performed as good as the `dynamic` optimum. Internally, `exhaustiveSel` tended to prefer a `trapezoidal` schedule. We thus ran extra simulations with `trapezoidal` scheduling. The c01 input again performed few selection rounds, fixing either `dynamic` or `af_a`. Since the expert chunk was not 1, `exhaustiveSel` did not find the `dynamic` optimum. Figure 4.4 graphs the results.

Indeed, the heterogeneous input seems to perform better in the neighborhood of the `dynamic` schedule on the DLS scale. This includes `dynamic` itself, as well as `trapezoidal` and standard `auto`, both of which outperformed Auto4OMP's methods. Auto4OMP wasted some time testing sub-optimal techniques, leading to performance overhead. Perhaps we did not observe this in the homogeneous scenario because it prefers `static_steal` (`awf_b` for lc_c01), which is relatively further from `dynamic` on the DLS scale. However, homogeneous vlp_c08 preferred `fac2a`, yet did not perform faster with Auto4OMP. Whether this negates our speculation, or whether it is due to the VLC container's inner workings, is to be determined.

The vlp_c08 and vcl_clusterIteration traversals yielded different results shown in Figure 4.5, with Auto4OMP's selection methods no longer being optimal.

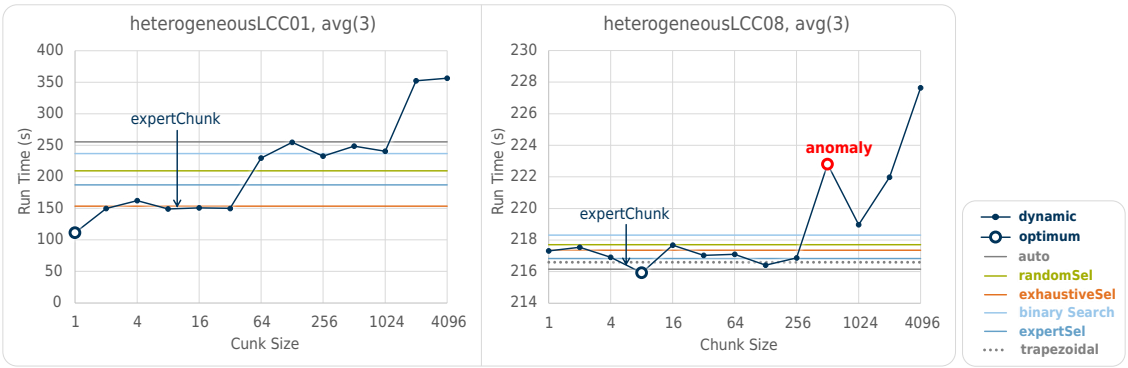


Figure 4.4.: MD-flexible performance with heterogeneous lc_c01 and lc_c08.
Each data point is an average of 3 runs. The horizontal lines represent 3-run averages with the corresponding schedules from the legend.
The anomaly deviated from the curve in only one of the 3 runs.

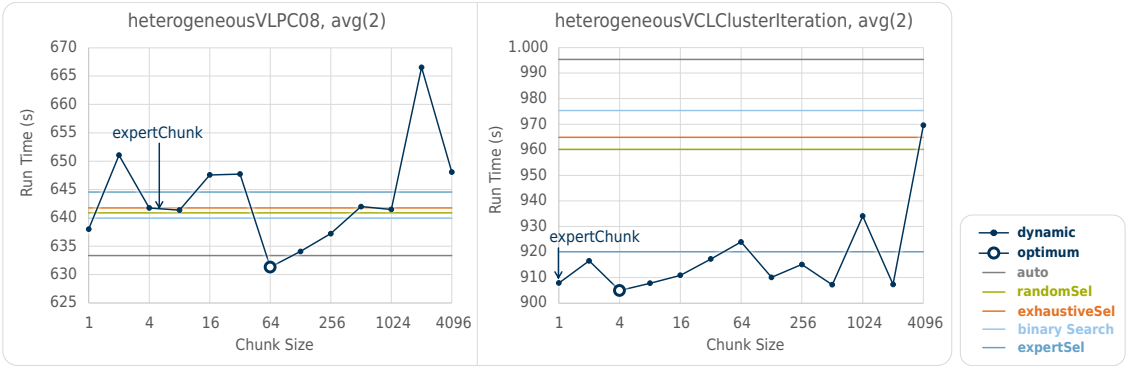


Figure 4.5.: MD-flexible performance
with heterogeneous vlp_c08 and vcl_clusterIteration.
Each data point represents an average of multiple runs, on a `dynamic` schedule.
The titular avg(i) means i runs were averaged per data point.

4.3. Falling Drop and Exploding Liquid

The Falling Drop and Exploding Liquid inputs yielded similar results to the heterogeneous linked cell traversals, but plateaued earlier. This time, the vlp_c08 graphs resembled their linked cells counterparts. Figure 4.6 illustrates the graphs for the c18 traversals. Fixing the most selected scheduling technique for Falling Drop (`fac2a`) did not improve performance. This is likely due to the simulation evolving over time, changing the preferred technique along with it. Note, explodingLiquid fails on the Git branch Dynamic-VL-merge, on which this thesis was based. Therefore, we maintained a parallel branch based on AutoPas master, on which the explodingLiquid tests were run [27].

It is noteworthy that the optimal chunk size is now 1, consistent throughout all traversal types. Combined with the previous runs where the dynamic optimum was over 200 for homogeneous, and under 100 for heterogeneous, these results hint that the optimal chunk size approaches 1 when the simulation's complexity increases.

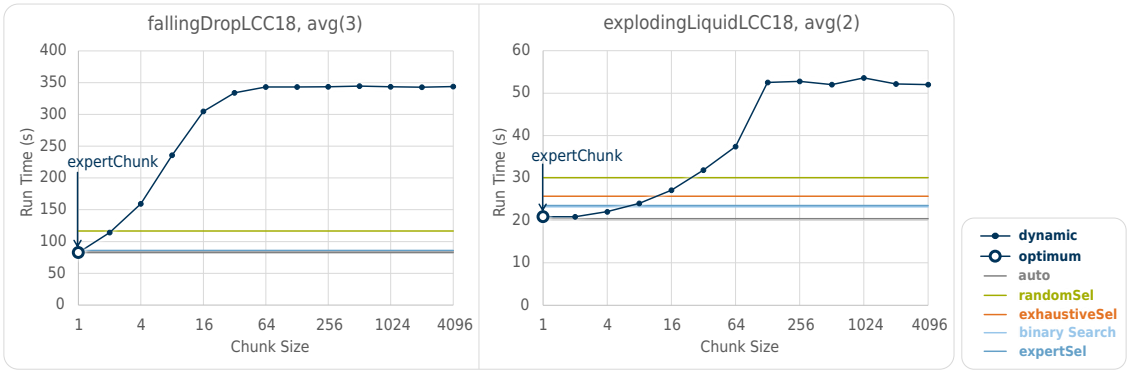


Figure 4.6.: MD-flexible performance with fallingDrop lc_c18 and explodingLiquid lc_c18. Each data point represents an average of multiple runs, on a `dynamic` schedule. The titular avg(i) means i runs were averaged per data point.

Finally, we investigated the Auto4OMP's effect during AutoPas tuning phases. We ran the standard fallingDrop input with logging enabled, and compare the iteration performance with and without Auto4OMP. As illustrated in Figure 4.7, Auto4OMP caused occasional jitters that worsen performance, compared to the standard `dynamic` schedule. AutoPas traversals tend to perform a bit worse towards their start, which sometimes extends to the end of their trial as AutoPas swaps in the next traversal. In particular, Figure 4.7 shows that lc_c01 performed a bit worse under Auto4OMP, leading AutoPas to select the next best traversal, which was vcl_c01_balanced in this case. This effect is likely due to Auto4OMP's re-selection triggering during the tuning-phase, and taking multiple loops before settling at the optimal schedule.

We suspect that this interaction influences both the AutoPas and the Auto4OMP tuning decisions, possibly resulting in sub-optimal selections. We repeated the `dynamic` schedule simulation 3 times, and AutoPas left this tuning phase with lc_c01 in all 3 runs. We thus suggest modifying the OpenMP configurator getters, such that they always return `dynamic,1` during AutoPas tuning phases, regardless of the configurator's internal attributes. This ensures that the AutoPas does not get fooled by the changing OpenMP schedule and assume the corresponding traversal is slow. After the auto-tuning phase concludes, the OpenMP configurator can then expose its internal schedule once more, allowing Auto4OMP to resume.

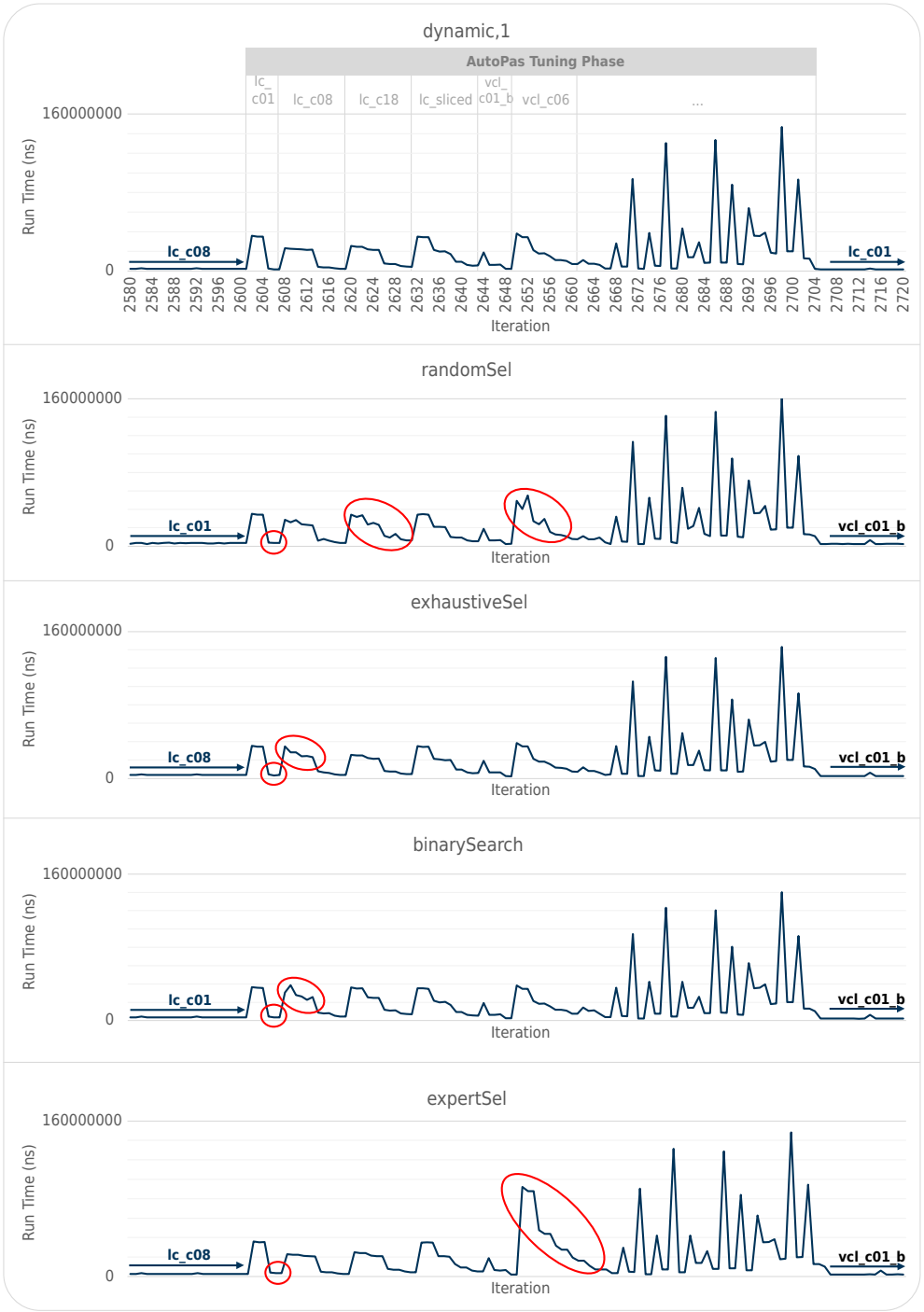


Figure 4.7.: MD-flexible auto-tuned fallingDrop iteration performance.
Each graph corresponds to the same tuning phase for all 5 OpenMP schedules.

5. Future Work

Although Auto4OMP showed promising results for homogeneous scenarios, it did not outperform the standard dynamic schedule in more complex simulations. This may stem from the fact that distinct regions in the complex simulation prefer different scheduling techniques, resulting in average performance on larger scales.

In particular, we suspect that the denser regions should be scheduled in smaller chunks. To understand, consider a space with sparsely scattered particles, and a dense clump of them in the middle. Since each particle in the clump has numerous neighbors, their computation takes much longer. Should one thread receive a chunk of these complex iterations, it would outlive other threads and essentially perform most of the simulation's load sequentially. One should thus avoid chunking dense iterations together. Indeed, work stealing or advanced scheduling techniques may mitigate this issue, provided they allow breaking up chunks after they are scheduled. However, Auto4OMP evaluates the execution times of iterations post-hoc, with no direct knowledge on the content of the simulation.

To maximize performance, AutoPas should thus consider implementing a custom scheduling method. For example, it could use standard dynamic scheduling, but linked cells with dense neighborhoods saturate the chunks that host them. Tracking the number of particles in a linked cell or neighbor list unlocks prior knowledge on the complexity of iterations, which has the potential to yield highly optimal load balancing techniques. Another promising, albeit more complex idea is to reorder the iterations in the traversal loop according to density, such that dense iterations are re-distributed evenly.

With the homogeneous results, it was shown that sophisticated scheduling techniques can outperform the dynamic optimum, and that the optimal chunk size depends on density. We therefore believe that setting custom OpenMP schedules for different regions of the simulation, which approach homogeneity on small enough scales, can potentially unlock the improvement seen with the simple input. To implement its custom techniques, AutoPas can leverage the already linked LB4OMP, and access its thread dispatching methods. One alternative may be for AutoPas to auto-tune a global chunk size or scheduling kind, but we suspect that this would perform about as good as Auto4OMP.

6. Conclusion

This thesis successfully integrated Auto4OMP into the AutoPas project. We adjusted LB4OMP's internal features and linked them with AutoPas, and thoroughly investigated the performance of this new setup. Although it showed promising results for homogeneous scenarios, Auto4OMP did not outperform the optimal dynamic schedule in more complex simulations. That said, it mostly maintained a similar performance to the dynamic optimum, which should be equivalent to auto-tuning the chunk size.

Furthermore, we found that the best chunk size for the MD-Flexible Falling Drop and Exploding Liquid inputs was close to one, hinting that the original standard dynamic schedule was already close to optimal. That said, further investigation with much larger and more varied simulations is warranted. As a next step, we suggest using the linked LB4OMP and the OpenMP configurator to implement a custom scheduling method, which leverages prior AutoPas knowledge on the state of the simulation to predict an optimal schedule.

A. ExpertSel Fuzzy Logic

This chapter explains the fuzzy system used in Auto4OMP's ExpertSel [3], implemented in `kmp_dispatch.cpp` [28].

A.1. Fuzzification

At the end of each time-step, the fuzzy system receives 4 inputs:

- LIB: load imbalance in the current time-step.
  Fuzzy sets: `Low`, `Moderate`, `High`.
- T_{par}: loop execution time in the current time step.
  Fuzzy sets: `Short`, `Medium`, `Long`.
- ΔLIB: change of LIB compared to the previous time-step.
  Fuzzy sets: `Improved`, `NoChange`, `Degraded`.
- ΔT_{par}: change of T_{par} compared to the previous time-step.
  Fuzzy sets: `Improved`, `NoChange`, `Degraded`.

The membership degrees $\mu_{\text{input,set}}$ of the above crisp inputs in their corresponding fuzzy sets are calculated, using the membership functions illustrated in Figure A.1.

In addition, Figure A.2 shows the fuzzy sets for the feedback variable ΔDLS, which indicates the scheduling technique to use in the next time-step. DLS $\in \{0, ..., 11\}$ represents the rank of an Auto4OMP scheduling technique, ordered by load balancing overhead. For example, if `awf_c` (DLS_8) was used in the current time-step, and $\Delta DLS = -2$, `fac2a` (DLS_6) will be used in the next time-step.

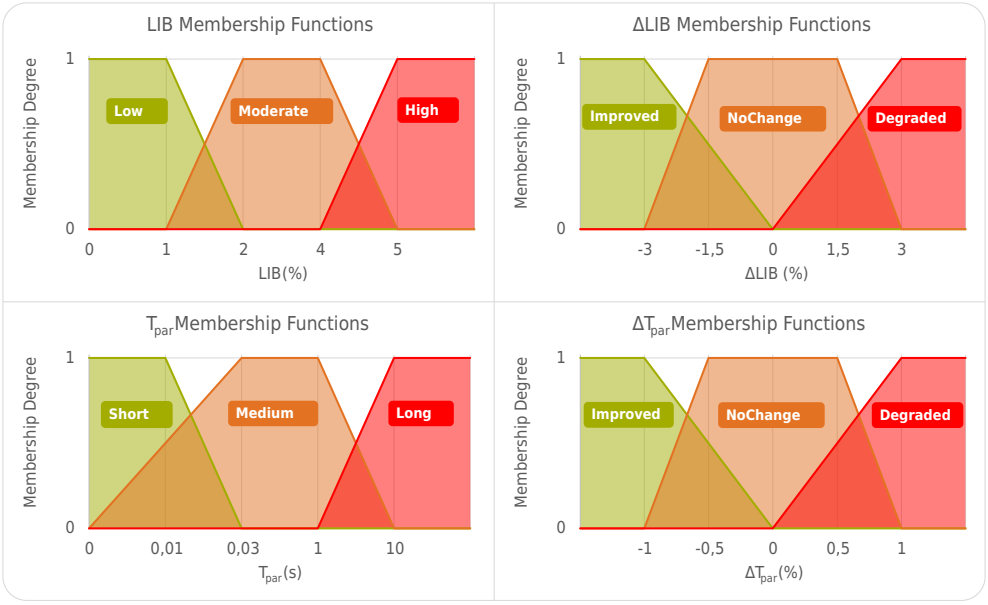


Figure A.1.: ExpertSel input membership functions

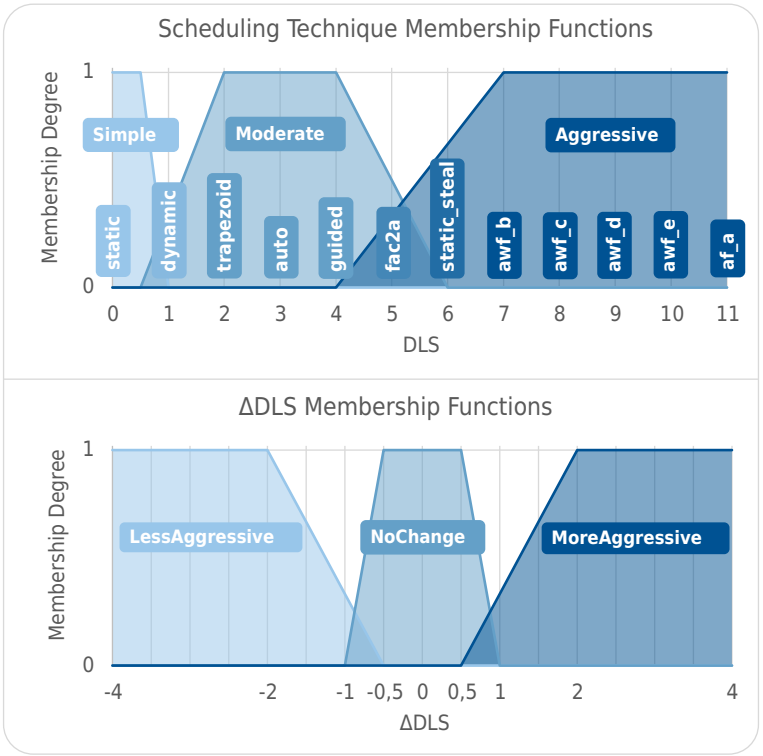


Figure A.2.: ExpertSel output membership functions

A.2. Expert Rules

The following rule base applies:

0. General form: (Elementary Condition 1) ∧ ... ⇒ Triggered Action

1. $(\Delta T_{par} =$ **Degraded**$) \wedge (\Delta LIB =$ **Improved**$) \Rightarrow \Delta DLS =$ **LessAggressive**
2. $(\Delta T_{par} =$ **Degraded**$) \wedge (\Delta LIB =$ **NoChange**$) \Rightarrow \Delta DLS =$ **LessAggressive**
3. $(\Delta T_{par} =$ **NoChange**$) \wedge (\Delta LIB =$ **Improved**$) \Rightarrow \Delta DLS =$ **LessAggressive**
4. $(T_{par} =$ **Short**$) \Rightarrow \Delta DLS =$ **LessAggressive**
5. $(\Delta T_{par} =$ **Degraded**$) \wedge (LIB =$ **low**$) \Rightarrow \Delta DLS =$ **LessAggressive**
6. $(\Delta T_{par} =$ **NoChange**$) \wedge (\Delta LIB =$ **NoChange**$) \Rightarrow \Delta DLS =$ **NoChange**
7. $(LIB =$ **Low**$) \Rightarrow \Delta DLS =$ **NoChange**
8. $(\Delta T_{par} =$ **Improved**$) \Rightarrow \Delta DLS =$ **NoChange**
9. $(LIB =$ **High**$) \wedge (T_{par} =$ **Long**$) \Rightarrow \Delta DLS =$ **MoreAggressive**
10. $(\Delta T_{par} =$ **Degraded**$) \wedge (\Delta LIB =$ **Degraded**$) \Rightarrow \Delta DLS =$ **MoreAggressive**
11. $(\Delta T_{par} =$ **NoChange**$) \wedge (\Delta LIB =$ **Degraded**$) \Rightarrow \Delta DLS =$ **MoreAggressive**

A.3. Inference and Defuzzification

Using the input membership degrees, get the minimal membership degree μ_i corresponding to the elementary conditions of each rule i. Rule 1 for example, gives $\mu_1 = \min\{\mu_{\Delta T_{par},Degraded}, \mu_{\Delta LIB,Improved}\}$. From these minima, get the maximal membership degrees per ΔDLS class. For instance, $\mu_{LessAggressive} = \max\{\mu_1, ..., \mu_5\}$.

Next, the dispatcher caps each ΔDLS fuzzy set j with the corresponding membership degree μ_j, and calculates the area under each resulting cut membership function. Figure A.3 illustrates an example. Finally, ΔDLS is selected based on the contribution of each cut set to the total area. In Figure A.3 for instance, the less aggressive fuzzy set has a large relative area, and will thus pull the output crisp ΔDLS towards it.

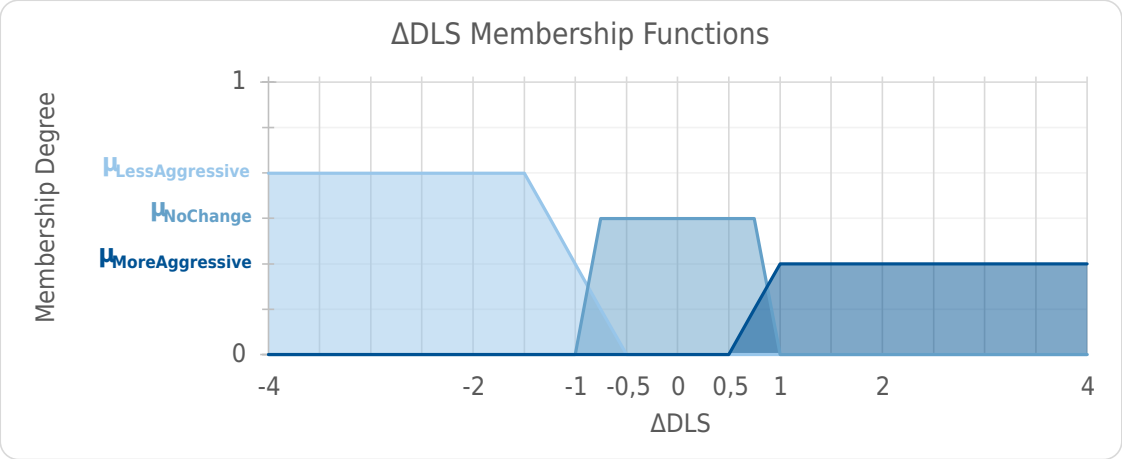


Figure A.3.: ExpertSel capped membership functions

Note, the very first time-step does not have prior LIB and T_{par} values to measure the changes against. Auto4OMP thus uses static scheduling for this first loop, and applies a one-time fuzzy system which does not use the Δ values. It takes the LIB and T_{par} from the first time-step, applies the rules from Table A.1 and the DLS membership functions from Figure A.2, and outputs a scheduling technique to use in the second time-step.

T_{par} \ LIB	Low	Moderate	High
Short	Simple	Simple	Simple
Medium	Simple	Moderate	Moderate
Long	Simple	Moderate	Aggressive

Table A.1.: DLS expert rules for the first time-step

B. Tutorial: Testing AutoPas with Auto4OMP on LRZ's Linux Cluster

This tutorial replicates the results of my thesis on CoolMUC. The Slurm job scripts submitted to CoolMUC are available at [29]. Each job tests one MD-Flexible input with different OpenMP chunk sizes and Auto4OMP selection methods. To log Auto4OMP's selection decisions, pass a file path to `KMP_TIME_LOOPS`. Beware, this option worsens performance.

To start, connect to the Linux cluster with `ssh -Y lxlogin1.lrz.de -l <username>`, and enter the following commands:

```
# Load the needed modules:
module load \
git gcc llvm cmake python doxygen graphviz automake sqlite hwloc cuda

# Clone AutoPas and the Slurm job scripts:
cd $HOME && mkdir -p ba && cd ba \
&& git clone https://github.com/AutoPas/AutoPas-CoolMUC-Jobs.git \
&& git clone https://github.com/AutoPas/AutoPas.git \
&& cd AutoPas && git checkout feature/improving-openmp-loop-scheduling \
&& mkdir build && cd build

# Build MD-Flexible:
CC=`which clang` CXX=`which clang++` cmake -DMD_FLEXIBLE_USE_MPI=OFF \
-DAUTOPAS_LOG_ITERATIONS=ON .. && make md-flexible && cd ..

# Build master-branch-based MD-Flexible for explodingLiquid:
git checkout \
feature/improving-openmp-loop-scheduling-dynamic-vl-unloaded \
&& mkdir build-dvl-off && cd build-dvl-off
CC=`which clang` CXX=`which clang++` cmake -DMD_FLEXIBLE_USE_MPI=OFF \
-DAUTOPAS_LOG_ITERATIONS=ON .. && make md-flexible && cd ..
```

```
# Option 1: individual tests with salloc.
## Allocate a CoolMUC-2 job:
cd build/examples/md-flexible \
&& salloc --partition=cm2_inter --time=01:00:00

## Prioritize Auto4OMP's libomp.so:
export LD_LIBRARY_PATH=\
$HOME/ba/AutoPas/build/_deps/auto4omp-build/runtime/src\
:$LD_LIBRARY_PATH

## Make sure Auto4OMP's libomp.so is linked:
ldd md-flexible

## Configure OpenMP to use 28 parallel threads:
export OMP_NUM_THREADS=28

# Get the GPU frequency and pass it to Auto4OMP:
export KMP_CPU_SPEED=$(lscpu | grep "CPU max MHz" | \
tr -d ' ' | cut -d ":" -f2 | cut -d "." -f1)

# Optional: log 'Auto4OMPs selection decisions.
## Beware, this worsens performance!
export KMP_TIME_LOOPS=./auto4omp.log

## Execute.
### The possible inputs are at AutoPas/examples/md-flexible/input.
### Pass the input file name without a path, as all inputs
### are copied directly under AutoPas/build/examples/md-flexible.
### This example tests the homogeneous lc_c08 small input
### with Auto4OMP's ExhaustiveSel.
srun ./md-flexible --openmp-kind exhaustiveSel \
--yaml-filename homogeneousLCC08Small.yaml

exit
```

```
# Option 2: submit a job script for a full test.
## This takes a few hours per job, depending on the input.
## Beware, the jobs assume AutoPas is at ~/ba/AutoPas.
## For custom directories, the scripts have to be adjusted.
## This example tests the homogeneous lc_c08 small input.
## Move to the job's subdirectory so the logs are written there:
cd $HOME/ba/AutoPas-CoolMUC-Jobs/auto4omp/homogeneousLCC08Small \
&& sbatch homogeneousLCC08Small.sh && cd $HOME

## To track a submitted job:
squeue -M cm2_tiny

## To cancel a submitted job:
scancel -M cm2_tiny <id>

## To summarize job outputs (assuming a single output exists per input):
INPUT_DIR=$HOME/ba/AutoPas-CoolMUC-Jobs/auto4omp/homogeneousLCC08Small \
&& grep $INPUT_DIR/$(ls $INPUT_DIR | grep ".out" -m 1) \
-e == -e LCC -e VCL -e VLP -e Simulate

## To summarize Auto4OMP's log (e.g., exhaustiveSel):
### (prints the number of times each scheduling technique was used.)
KMP_LOG=$INPUT_DIR/auto-3.log \
&& echo "static:" && grep -o " STATIC " $KMP_LOG | wc -l \
&& echo "dynamic:" && grep -o " SS " $KMP_LOG | wc -l \
&& echo "trapezoidal:" && grep -o " TSS " $KMP_LOG | wc -l \
&& echo "auto:" && grep -o "LLVM" $KMP_LOG | wc -l \
&& echo "guided:" && grep -o " GSS " $KMP_LOG | wc -l \
&& echo "steal:" && grep -o " Steal " $KMP_LOG | wc -l \
&& echo "fac2a:" && grep -o " mFac2 " $KMP_LOG | wc -l \
&& echo "awf_b:" && grep -o " AWF-B " $KMP_LOG | wc -l \
&& echo "awf_c:" && grep -o " AWF-C " $KMP_LOG | wc -l \
&& echo "awf_d:" && grep -o " AWF-D " $KMP_LOG | wc -l \
&& echo "awf_e:" && grep -o " AWF-E " $KMP_LOG | wc -l \
&& echo "af_a:" && grep -o " mAF " $KMP_LOG | wc -l
```

Listing B.1: Bash: Testing AutoPas with Auto4OMP on CoolMUC

List of Figures

2.1. Membership Functions of Auto4OMP's DLS Portfolio 9
2.2. Linked Cells Container 12
2.3. AutoPas Traversal Base Steps 13
2.4. AutoPas Traversal Class Hierarchy. The concrete traversals are dubbed lc_c01, lc_c18, lc_c08, lc_c04, vlp_c08 (a.k.a. VLCCellPairC08Traversal) and vcl_clusterIteration. 14

3.1. The OpenMP configurator's UML class diagram 21
3.2. MD-Flexible's fallingDrop simulation [1] 24
3.3. MD-Flexible's explodingLiquid simulation [26] 24

4.1. MD-flexible performance with a homogeneous lc_c08 input. Each data point is an average of 3 runs. The horizontal lines represent 3-run averages with the corresponding schedules from the legend. 26
4.2. MD-flexible performance with homogeneous vlp_c08 and vcl_clusterIteration. Each data point represents an average of multiple runs, on a `dynamic` schedule. The titular avg(i) means i runs were averaged per data point. . 26
4.3. MD-flexible performance with small homogeneous lc_c08 inputs. Each data point represents the most performant chunk size of one input with a specific density, on a `dynamic` schedule. 27
4.4. MD-flexible performance with heterogeneous lc_c01 and lc_c08. Each data point is an average of 3 runs. The horizontal lines represent 3-run averages with the corresponding schedules from the legend. The anomaly deviated from the curve in only one of the 3 runs. 28
4.5. MD-flexible performance with heterogeneous vlp_c08 and vcl_clusterIteration. Each data point represents an average of multiple runs, on a `dynamic` schedule. The titular avg(i) means i runs were averaged per data point. . 28
4.6. MD-flexible performance with fallingDrop lc_c18 and explodingLiquid lc_c18. Each data point represents an average of multiple runs, on a `dynamic` schedule. The titular avg(i) means i runs were averaged per data point. 29
4.7. MD-flexible auto-tuned fallingDrop iteration performance. Each graph corresponds to the same tuning phase for all 5 OpenMP schedules. 31

A.1. ExpertSel input membership functions 38
A.2. ExpertSel output membership functions 38

A.3. ExpertSel capped membership functions 40

List of Tables

A.1. DLS expert rules for the first time-step 40

Bibliography

[1] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann. "N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas." In: *Computer Physics Communications* 273 (Apr. 2022), p. 108262. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2021.108262.

[2] J. Korndörfer, A. Eleliemy, A. Mohammed, and F. M. Ciorba. "LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications." In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (Apr. 2022), pp. 830–841. ISSN: 2161-9883. DOI: 10.1109/tpds.2021.3107775.

[3] A. Mohammed, J. H. M. Korndorfer, A. Eleliemy, and F. M. Ciorba. "Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP." In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (Dec. 2022), pp. 4383–4394. ISSN: 2161-9883. DOI: 10.1109/tpds.2022.3189270.

[4] The Clang Team. *Clang Compiler User's Manual*. Comp. software. Version 19.0.0git. June 14, 2024. Chap. OpenMP Features. URL: clang.llvm.org/docs/UsersManual.html#openmp-features.

[5] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Comp. software. Version 5.2. Nov. 2021. Chap. 11.5.3 schedule Clause, pp. 252–254. URL: openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[6] The LLVM Project. *LLVM OpenMP Runtime Library*. Comp. software. Sept. 23, 2015. Chap. 1.7.1 Work Sharing Example, pp. 4–5. URL: github.com/llvm/llvm-project/blob/main/openmp/runtime/doc/Reference.pdf.

[7] J. Korndörfer, A. Eleliemy, A. Mohammed, and F. M. Ciorba. *LB4OMP*. Comp. software. Version master. Sept. 14, 2022. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/README.rst.

[8] OpenMP Runtime Team, Intel Corporation and HPC Group University of Basel. *kmp_dispatch.cpp*. Nov. 18, 2021. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/runtime/src/kmp_dispatch.cpp#L2380-L2390.

[9] A. Chronopoulos, R. Andonie, M. Benche, and D. Grosu. "A class of loop self-scheduling for heterogeneous clusters." In: *Proceedings 2001 IEEE International Conference on Cluster Computing*. IEEE, 2001. DOI: 10.1109/clustr.2001.959989.

[10] A. Eleliemy and F. M. Ciorba. "A distributed chunk calculation approach for self-scheduling of parallel applications on distributed-memory systems." In: *Journal of Computational Science* 51 (Apr. 2021), p. 101284. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2020.101284.

[11] OpenMP Runtime Team, Intel Corporation and HPC Group University of Basel. *kmp.h*. Nov. 19, 2020. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/runtime/src/kmp.h#L324-L364.

[12] OpenMP Runtime Team, Intel Corporation and HPC Group University of Basel. *omp.h.var*. Apr. 27, 2020. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/runtime/src/include/50/omp.h.var#L45-L51.

[13] HPC Group University of Basel, Intel Corporation, and M. Hachicha. *LB4OMP-custom*. June 26, 2024. URL: github.com/MehdiHachicha/LB4OMP-custom.

[14] OpenMP Runtime Team, Intel Corporation and HPC Group University of Basel. *kmp_dispatch.cpp*. Nov. 18, 2021. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/runtime/src/kmp_dispatch.cpp#L112-L135.

[15] OpenMP Runtime Team, Intel Corporation and HPC Group University of Basel. *kmp_dispatch.cpp*. Nov. 18, 2021. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/runtime/src/kmp_dispatch.cpp#L1067-L1135.

[16] C. Menges. *CBasedTraversal.h (original)*. Commit bfc5a4e. AutoPas. Feb. 16, 2024. URL: github.com/AutoPas/AutoPas/blob/master/src/autopas/containers/cellPairTraversals/CBasedTraversal.h#L159.

[17] Kitware, Inc. and Contributors. *FindOpenMP*. Version 3.30. 2024. URL: cmake.org/cmake/help/latest/module/FindOpenMP.html.

[18] M. Hachicha. *autopas_auto4omp.cmake*. Apr. 15, 2024. URL: github.com/AutoPas/AutoPas/blob/feature/improving-openmp-loop-scheduling/cmake/modules/autopas_auto4omp.cmake.

[19] NVIDIA Corporation. *Your GPU Compute Capability*. 2024. URL: developer.nvidia.com/cuda-gpus.

[20] AutoPas. *CMakeLists.txt*. June 15, 2024. URL: github.com/AutoPas/AutoPas/blob/feature/improving-openmp-loop-scheduling/src/autopas/CMakeLists.txt#L23.

[21] F. Gratl. *WrapOpenMP.h*. June 3, 2024. URL: github.com/AutoPas/AutoPas/blob/feature/improving-openmp-loop-scheduling/src/autopas/utils/WrapOpenMP.h#L77.

[22] C. Menges. *CBasedTraversal.h (new)*. June 15, 2024. URL: github.com/AutoPas/AutoPas/blob/feature/improving-openmp-loop-scheduling/src/autopas/containers/cellPairTraversals/CBasedTraversal.h#L141.

[23] OpenMP Runtime Team, Intel Corporation and HPC Group University of Basel. *kmp_runtime.cpp*. Nov. 19, 2020. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/runtime/src/kmp_runtime.cpp#L2798-L2847.

[24] Leibniz-Rechenzentrum. *CoolMUC-2*. URL: doku.lrz.de/display/PUBLIC/CoolMUC-2.

[25] M. Hachicha. *auto4omp-tests*. June 7, 2024. URL: github.com/AutoPas/AutoPas/tree/feature/improving-openmp-loop-scheduling/examples/md-flexible/input/auto4omp-tests.

[26] S. Seckler, F. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann. *Exploding liquid scenario*. Dec. 10, 2019. URL: youtu.be/u7TE5KiSQ08.

[27] F. A. Gratl, S. Seckler, H.-J. Bungartz, P. Neumann, and M. Hachicha. *AutoPas branch feature/improving-openmp-loop-scheduling-dynamic-vl-unloaded*. June 24, 2024. URL: github.com/AutoPas/AutoPas/tree/feature/improving-openmp-loop-scheduling-dynamic-vl-unloaded.

[28] OpenMP Runtime Team, Intel Corporation and HPC Group University of Basel. *kmp_dispatch.cpp*. Nov. 18, 2021. URL: github.com/unibas-dmi-hpc/LB4OMP/blob/master/runtime/src/kmp_dispatch.cpp#L513-L1062.

[29] M. Hachicha. *AutoPas-CoolMUC-Jobs*. June 22, 2024. URL: github.com/AutoPas/AutoPas-CoolMUC-Jobs.