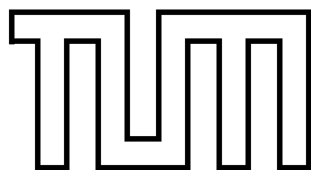


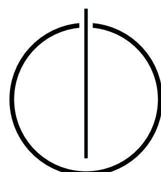
FAKULTÄT FÜR INFORMATIK

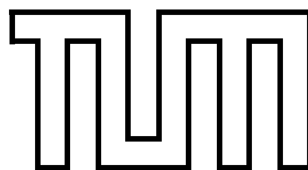
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computational Science and Engineering

Enabling Dynamic Load Balancing for MiniMD on OctoPOS

Radu Raicea





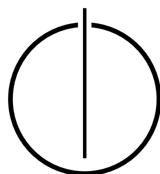
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Computational Science and Engineering

Enabling Dynamic Load Balancing for MiniMD on OctoPOS

Author:	Radu Raicea
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Santiago Narváez Rivas, M.Sc.
Date:	October 9, 2022



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

Munich, October 9, 2022 Radu Raicea

Acknowledgements

I would like to thank Santiago Narváez Rivas for being such a great advisor and for helping me throughout my thesis, as well as being so flexible with me.

I would also like to thank my wife for her support, especially during the more work intensive moments of the thesis.

Abstract

This thesis aims at adding dynamic load balancing to the miniMD (a simple parallel molecular dynamics simulation program) port in OctoPOS, an operating system that is meant to be run on exotic multiprocessor computers such as those designed specifically with the invasive computing paradigm in mind. Different load balancing schemes are implemented, and their performance are compared: a “naive” load balancing scheme, an inverted pressure scheme using a swipe to change all subdomains in one load balancing iteration, as well as A Loadbalancing Library’s tensor method.

Contents

Acknowledgements vii

Abstract ix

I. Introduction and Background 1

1. Introduction 2
1.1. A World of Systems . . . 2
1.2. The Supercomputer . . . 2
1.3. Molecular Dynamics Simulations . . . 3
1.4. The Imbalance Problem . . . 3
1.5. Dynamic Load Balancing . . . 4

2. Background 5
2.1. miniMD . . . 5
2.2. Lennard-Jones Potential . . . 5
2.3. Neighborhood Lists . . . 6
2.4. Invasive Computing and OctoPOS . . . 8

3. Related Works 10
3.1. Porting miniMD to OctoPOS . . . 10
3.2. ALL - A Loadbalancing Library . . . 11
3.3. miniMD's Structure . . . 12
3.3.1. Setting Up the Simulation . . . 12
3.3.2. Building the Neighborhood Lists . . . 13
3.3.3. Computing Forces and Moving the Particles . . . 13
3.3.4. Exchanging Atoms Between Ranks . . . 14

II. Thesis Development 15

4. Naive Dynamic Load Balancing 16
4.1. Detecting a Load Imbalance . . . 17
4.1.1. Defining Work . . . 17
4.1.2. Synchronizing Work Between Processing Units . . . 18
4.1.3. Computing the Imbalance . . . 18
4.2. Determining New Subdomains . . . 20
4.2.1. The Naive Heuristic . . . 20

4.2.2. Finding the Processing Unit With the Least Amount of Work 21
4.2.3. Determining Which Adjacent Processing Unit to Balance With . . . 21
4.2.4. Changing the Subdomain Boundaries 22
4.3. Transferring Atoms to their New Processing Units 24
4.3.1. The `comm_setup` function 25
4.3.2. The `exchange` function 25
4.3.3. The `borders` function 25
4.3.4. The `build` function 25

5. Improved Load Balancing Schemes 26
5.1. Swipe Load Balancing 26
5.1.1. Inverted Pressure Scheme 30
5.1.2. Tensor Method (ALL) Scheme 31

6. Polishing the Dynamic Load Balancing 33
6.1. Some Optimizations 33
6.2. Timing the Load Balancing 35

III. Results and Conclusion 37

7. Evaluation 38
7.1. Evaluation Test Scenarios 38
7.2. Evaluation Test Environment 38
7.3. Results 39

8. Conclusion 42
8.1. Goals Achieved 42
8.2. Future Work 42

IV. Appendix 44

Bibliography 47

Part I.

Introduction and Background

1. Introduction

1.1. A World of Systems

We live in a world made of systems working together. These systems can be either natural or man-made, however, at their core, they are still multiple parts interacting with each other to form a whole. Since we are driven by curiosity to understand these systems, as well as to build them, we require a way to gather information about how they behave in certain environments and how they react to actions performed on them. We could gather the information in the real world, but often times this comes with serious limitations, if even possible.

Instead, to study how these systems behave, we often need to perform computer simulations. Simulations provide a safe and controllable environment to understand the systems in various scenarios, often not necessarily possible in the real world. As well, given enough computational power, we are able to speed up the real world time of some systems to get much faster results than if we studied the system in the real world.

These computer simulations start off with a mathematical model, a simulation domain, boundary conditions, and the initial state of the system. By using a mathematical model, the computer is able to compute the new state that the system will be in after each time step. The simulation is also limited to the domain specified, and it knows what to do with elements of that system that reach the boundaries of that domain.

Since most interesting systems studied by scientists and engineers are gigantic with respect to the number of elements they contain, the computer running the simulation needs to have a great degree of computational power, as well as a large capacity to store the data of the system. This is why one single computer, even the most powerful computer in the world, would not be able to run interesting simulations in a reasonable amount of time. Instead, we need a network of computers working together to run those simulations; a supercomputer.

1.2. The Supercomputer

A large set of computers connected through a network, like SuperMUC-NG at the LRZ Lab in Garching [dBAdW99], are able to work in coordination to run large simulations in a reasonable amount of time.

Often, computer simulations meant to run on a supercomputer require us to divide the simulation domain into multiple smaller subdomains. The number of subdomains depends on the number of processing units allocated by the supercomputer for the simulation. Each processing unit performs computations on the element of its subdomain.

Afterwards, these processing units need to communicate with each other to exchange information about the elements that are located in their subdomain because they also interact with other elements from their neighboring subdomains. This communication between processing units is done through an interface like the Message Passing Interface (MPI)[Lab12].

1.3. Molecular Dynamics Simulations

One type of system scientists and engineers are interested in simulating is molecular systems. These systems contain a large number of atoms and molecules that interact with each other, changing their positions and velocities at each time step. In this thesis, we are interested in these kinds of systems.

A large and well known molecular dynamics (MD) program is the Large-scale Atomic/-Molecular Massively Parallel Simulator (LAMMPS) developed by the Sandia National Laboratories[APT22]. Another, much smaller, MD program is miniMD, also developed by the Sandia National Laboratories. This smaller MD program was made specifically to me more simple, lightweight, and easier to adapt to different architectures[Lab16]. We will be working on miniMD because modifying it will be a much easier task than modifying LAMMPS.

1.4. The Imbalance Problem

When dividing a simulation domain to run it on a supercomputer, we often make equally sized subdomains. If the elements of the simulated system are somewhat evenly distributed throughout the simulation, this approach can work well. This is because each processing unit performs a similar amount of work and holds a similar amount of data as the other processing units.

A problem can arise when the elements of the system do not stay somewhat evenly distributed throughout the simulation. In this case, the amount of work done by each processing unit is vastly different, resulting in a great loss of efficiency due to the idling of the processing units with little work to do. This can be seen in figure 1.1, where a water droplet is dropped on a copper block. Most of the molecules will stay at the bottom of the simulation domain, resulting in idleness in the processing units that cover the top part of the domain.

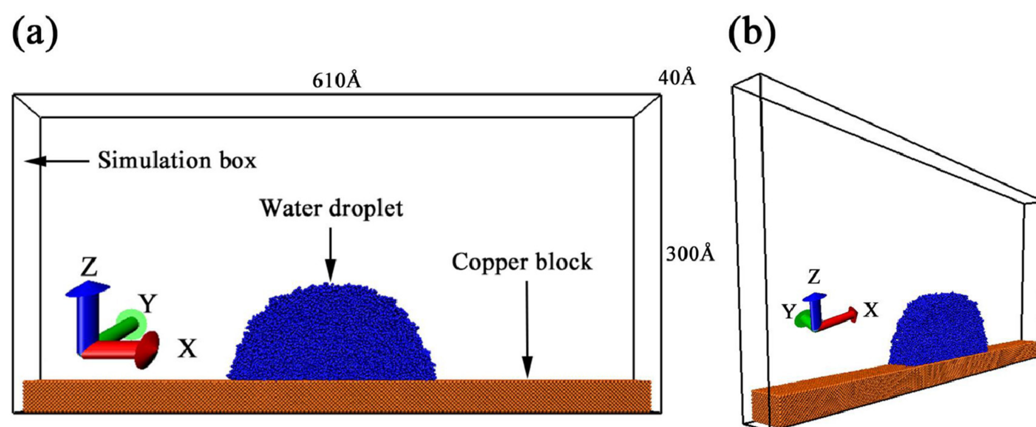


Figure 1.1.: Water droplet hitting a copper block.
Source: [YXL+21]

We could create unequally sized subdomains to assure a balance of work between processing units, but nothing guarantees that the elements will stay in the same subdomain. They might instead all move to another subdomain, creating idleness for some processing units while increasing the load of others.

This imbalance problem can be especially prevalent in molecular dynamics simulations because, often times, interesting simulations don't start with a domain that has evenly distributed molecules. Also, scenarios can involve collision of objects, which might drastically change the distribution of molecules throughout the domain.

1.5. Dynamic Load Balancing

What is needed is a way to detect these work load imbalances between processing units and a way to adjust the subdomains of the processing units to rebalance the work load. This would allow us to set the subdomains naively at the beginning of the simulation, and automatically change the subdomains whenever some processing units do much less work than others.

In the context of an MD simulation, this can be incredibly useful. If we simulate a collision between two small objects, most of the simulation domain is empty. If we have a high number of processing units, which we usually do, many of them might not have a single molecule, making them a complete waste of resources.

The motivation of this thesis is to add dynamic load balancing to the miniMD port in OctoPOS, an operating system described in section 2.4.

2. Background

2.1. miniMD

miniMD is a codebase that allows the creations and execution of parallel MD simulations. It was designed to be a much smaller version of LAMMPS by the Sandia National Laboratories and is part of the Mantevo mini-application suite [CTN+09]. It is mostly written in C++ and it is intended to be run on supercomputer architectures, as well as other experimental architectures because it can somewhat easily be updated to work on new architectures.

It performs simulations for MD systems using the Lennard-Jones or EAM models for intermolecular potential. In the scope of this thesis, only Lennard-Jones potential was used for the force calculations.

2.2. Lennard-Jones Potential

At each time step of the MD simulation, molecules interact with each other, creating a certain force on one another. This force makes them either attract or repel each other. The calculation for the force they exert on each other can be calculated using the Lennard-Jones model, which is defined as:

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right], \tag{2.1}$$

where r is the distance between the two molecules, sigma is the intermolecular distance when the potential is zero, and epsilon is the depth of the potential well[WRHDF20].

The following potential creates an intermolecular interaction where extremely close molecules repel each other, close molecules attract each other, and molecules that are not close to each other do not interact. This can be seen in figure 2.1, relating the potential with the intermolecular distance:

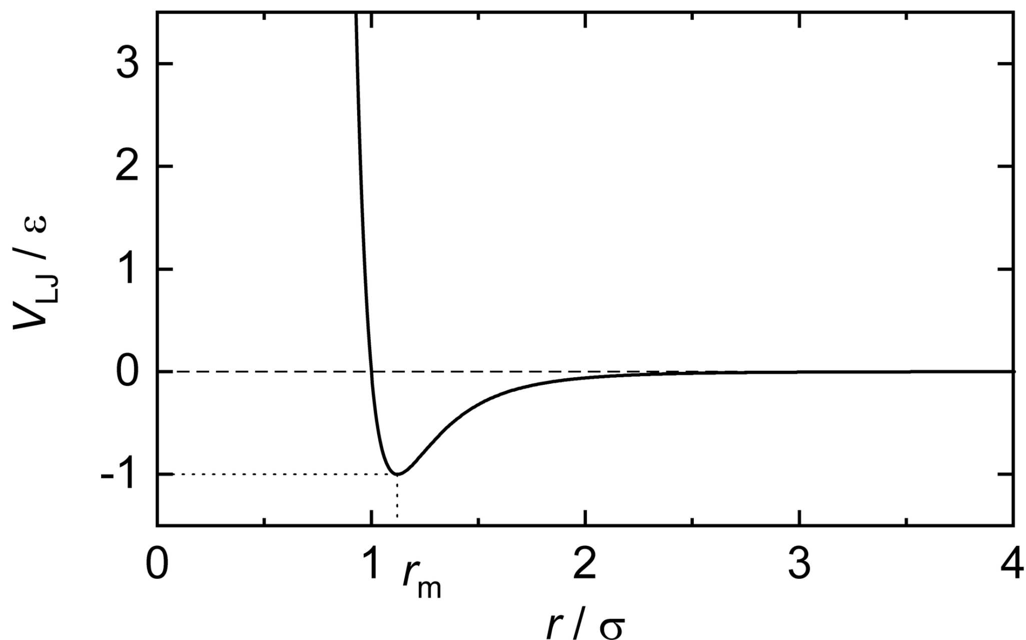


Figure 2.1.: Lennard-Jones Potential with respect to the intermolecular distance.
Source: [Wik22]

We can observe that when the intermolecular distance is equal to sigma, there is no interaction between the molecules, but when the distance gets smaller than sigma, there is a tremendous repulsive force. Another important thing to notice is when the intermolecular distance is three times sigma. There is virtually no attractive or repulsive force between the molecules.

The previous factor is very important in the scope of our simulation because calculating the Lennard-Jones potential for each pair of molecules in our simulation, on every time step, can become extremely computational intensive for not reason, since most potential will be virtually zero for far away molecules.

To counter these negligible potential calculations, miniMD uses a concept called neighborhood lists.

2.3. Neighborhood Lists

The purpose of neighborhood lists is to limit unnecessary force calculations between molecules that have a virtually no forces exerted between them.

This approach aims to determine a certain cut-off distance, after which pairwise forces become negligible. Then, for each molecule, a list of molecules that have an intermolecular distance less than the cut-off distance is kept. When calculating the force exerted on that molecule, it only uses the potentials of the molecules from its neighborhood list. This approach decreases the number of intermolecular potentials needed to be calculated, but adds the need to compute neighborhood lists for each molecule in the simulation, which in turn calculates the distance between each pair of molecules.

To optimize this, miniMD also uses the link-cell method [Koe21]. This method divides the domain into bins of the size of around the cut-off distance. This binning isn't very computationally expensive and only needs to be done once for the whole domain. The purpose of this binning is to limit the distance calculation when building neighborhood lists. In a 2D domain, building the neighborhood list for a molecule only requires distance calculations for pairs of molecules that are in 9 bins. An example of the reduced number of distances required to be computed is shown in the following figure.

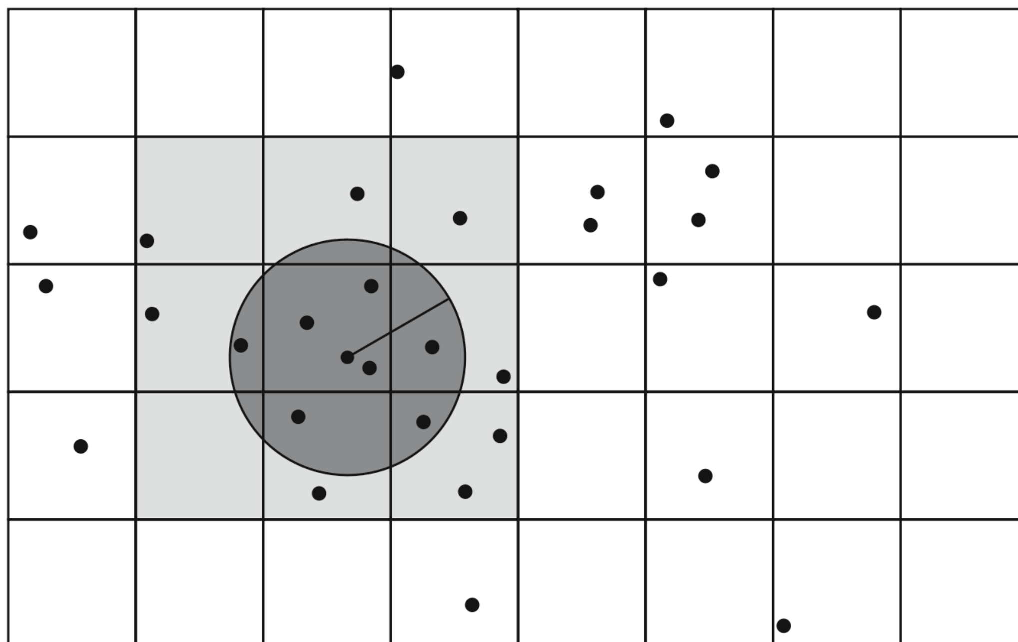


Figure 2.2.: The simulation domain decomposed in cells. The light gray area represents the cells in which we look for neighbors for the molecule in the center of the circle. The dark gray area represents the actual neighbors in that molecule's neighborhood list.
Source: [GZK07]

Building the neighborhood lists on each time step is still a computationally expensive endeavor. To counter this, we can build them after every few time steps. However, since molecules interact with each other and move in the domain, neighborhood lists might become outdated before they are rebuilt, introducing errors that can become major. This is why miniMD uses a cutoff distance slightly larger than needed, to compensate for the movement of molecules at every time step.

There is another problem when building a neighborhood lists for a simulation that is run on multiple processing units, each with their own subdomains: computing forces for molecules at the edge of the subdomains. This is especially true for subdomain edges between two processing units. This is a problem because molecules from one subdomain requires information about neighboring molecules from the other subdomain, but they reside on another processing unit with which communication is impossible in the middle of an iteration.

To solve this problem, whenever a division of subdomains is made, every subdomain needs to be expanded slightly. Then, molecules from neighboring subdomains need to be copied over on those expanded edges. These locations are called “ghost cells”. This makes every subdomain overlap with another, and the exchange can happen every few iterations.

Then, neighborhood lists for molecules at the edge of a subdomain can be built using the copied molecules from the neighboring subdomains.

2.4. Invasive Computing and OctoPOS

Invasive computing is a new paradigm for designing parallel computer systems and for writing applications to run on those. A computer designed under such paradigm allows applications to increase and reduce the number of processing units depending on its needs. One purpose of such an approach is to increase the efficiency of the processing units. Processes that temporarily require a great amount of parallel computing power can “invade” new processing units, and then “retreat” to release them and allow other processes to use them[THH+11].

OctoPOS is an operating system that is meant to be run on exotic multiprocessor computers such as those designed specifically with the invasive computing paradigm in mind[OSK+11]. miniMD was previously ported to work on a system running OctoPOS. The port is written in C instead of C++. OctoPOS allows a program to be run with varying allocated resources because it supports invasive computing. Its goal is to allow programs to free resources it does not need anymore, or to receive more resources if it needs them.

In the case of simulation programs like miniMD, which are meant to be run on supercomputers, this invasive computing paradigm would allow the resource manager of the supercomputer to increase or reduce the number of MPI ranks allocated to the process in which the simulation is running. This approach can increase the efficiency of a supercomputer by properly reallocating resources to the different processes currently running. This is a use case of invasive computing for a supercomputer. For OctoPOS, which is not necessarily meant to be run on a supercomputer, miniMD was ported on it to show how an MD simulation program can be run on it.

In order to change the number of MPI ranks allocated to a process while running, there needs to be a change in the interface of MPI. The new interface should allow a process to change its number of ranks during runtime. This extension of MPI's interface is called *i*MPI, for invasive MPI[HJB17]. *i*MPI's implementation in OctoPOS does not include all the functions, like `MPI_Gather`, known to be implemented in MPI. It has, however, the new functions that allow invasive programming features:

1.
```
int MPI_Init_adapt(int *argc, char **args,
int *local_status);

```

This is the initiation function for MPI, but which specifies that we are using the invasive programming model. It requires a `local_status` parameter to differentiate between new ranks being dynamically added by the resource manager and already existing ranks.

2.
```
int MPI_Probe_adapt(int *pending_adaptation,
int *local_status, MPI_info *info);

```

This function is used to check whether there are resources available to perform a change of the number of ranks. The `pending_adaptation` parameter defines whether a change of resources is waiting to happen, and the `local_status` parameter gives information about the rank's change in the adaptation.

3.
```
int MPI_Comm_adapt_begin(MPI_Comm *intercomm,
MPI_Comm *intracomm, int *stayingcount, int *leavingcount,
int *joiningcount);

```

This function starts the adaptation of ranks, and creates two communication groups. `intercomm` contains the ranks entering or leaving the simulation program, and `intracomm` contains either all the remaining ranks, in the case where the resource manager takes ranks away, or the previous ranks with the newly added ranks.

4.
```
int MPI_Comm_adapt_commit( );

```

This function finishes the change of ranks it replaces `MPI_COMM_WORLD` with the group of ranks from `intracomm` and the ranks that are no longer allocated to the process are killed.

3. Related Works

3.1. Porting miniMD to OctoPOS

In a previous master's thesis by Huaiwei Zhang, miniMD was ported to OctoPOS[Zha22]. In this port, only the Lennard-Jones potential can be used.

There are three test scenarios implemented:

1. EVENLY

   This scenario evenly distributes the molecules when setting up the simulation. It doesn't simulate gravity acting on the particles. It uses periodic boundary conditions.

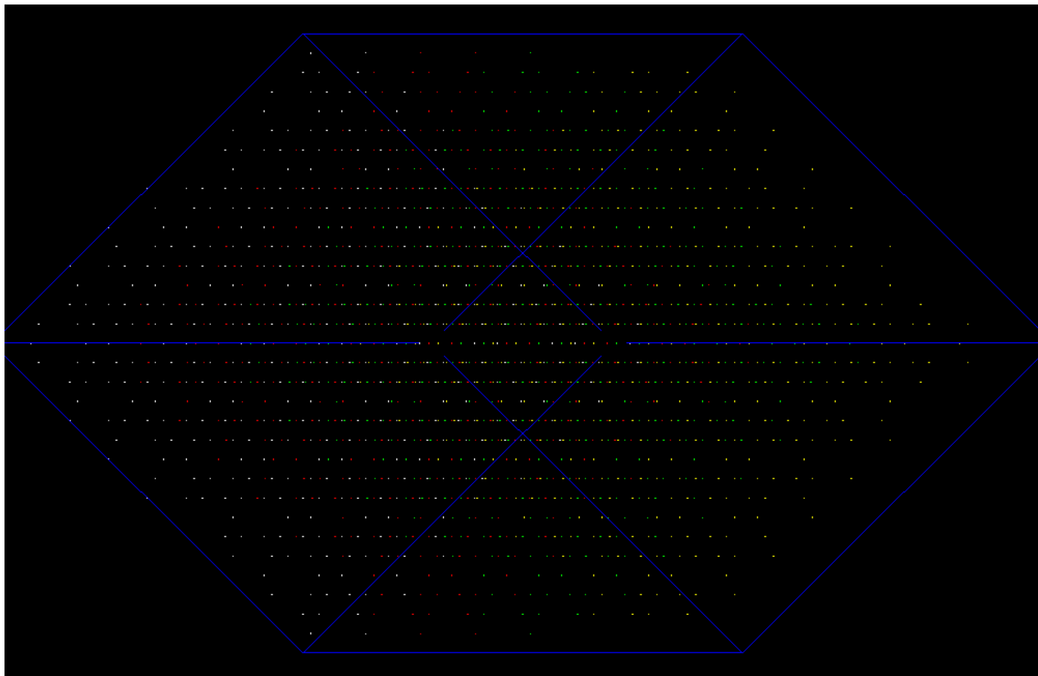


Figure 3.1.: Initial state of an EVENLY scenario.

2. RBCCOLLISION

   This scenario drops a circle of molecules on a slab of molecules. Gravity is simulated, and the boundary conditions are reflective.

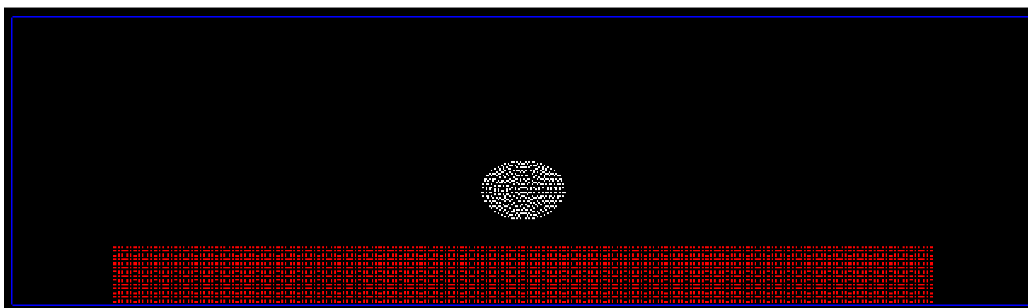


Figure 3.2.: Initial state of a RBCCOLLISION scenario.

3. OBCCOLLISION

   This scenario drops a rectangle of molecules on a slab o molecules. Gravity is simulated, and the boundary conditions are an outflow.

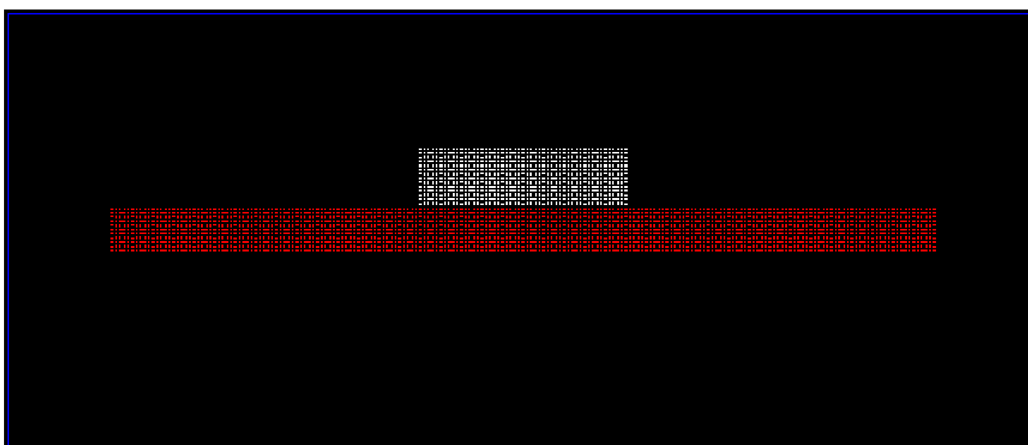


Figure 3.3.: Initial state of an OBCCOLLISION scenario.

These differing flows are useful to see how dynamic load balancing performs under different conditions.

3.2. ALL - A Loadbalancing Library

ALL is a load balancing library created by the Simulation Laboratory Molecular Systems of the Juelich Supercomputing Centre (JSC), Research Centre Juelich (Forschungszentrum Jülich GmbH) in Germany that provides multiple domain decomposition schemas to change the subdomains when load balancing is required[HSS99]. These include tensor product, staggered grid and histogram based staggered grid. The library also has multiple new domain decomposition schemes that are currently being built: topological mesh, Voronoi mesh and orthogonal recursive bisection.

Unfortunately, this library can not be used for the miniMD port on OctoPOS because it is written in C++ and requires a C++11 compiler, as well as full iMPI support of functions that are not currently supported in OctoPOS.

3.3. miniMD's Structure

In the following section, miniMD's port in OctoPOS will be presented to give a better understanding of how it functions. Multiple key features of miniMD will be presented as subsections.

3.3.1. Setting Up the Simulation

miniMD starts by setting up iMPI for inter-rank communication in `miniMD.c`. It does so by defining iMPI related variables and calling the initiation function for iMPI: `MPI_Init_adapt`. This can be seen in listing 3.1.

```
MPI_Info info;
MPI_Comm intercomm, new_comm_world;
int rank, old_size, new_size, local_status, tmp_status, pending_adapt,
    staying_count, leaving_count, joining_count;
MPI_Init_adapt(argc, argv, &local_status);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Listing 3.1: Initialization of iMPI.

Then, miniMD begins running the invasive logic given by iMPI to add or remove ranks to the main communicator. While this is a vital part of the miniMD port on OctoPOS, it is not directly relevant to the scope of this thesis.

After the communication is set up, the simulation logic begins. miniMD first initializes multiple containers to keep the data throughout the simulation by calling multiple helper functions. miniMD sets up the simulation using hard-coded parameters from the `input.c` module. This is also where the 3 scenarios mentioned in section 3.1 are defined, along with their parameters.

After the initialization of the containers, the relevant variables from the `in` variable are set on those containers, depending on the simulation scenario chosen.

After the initialization of the containers, the simulation box is created. In the `EVENLY` scenario, a 3D simulation box is created. In the other two scenarios, a 2D box is created.

Once the simulation box is created, the simulation domain is divided into multiple subdomains to distribute to the available iMPI ranks. This is done in the `comm.c` module. It computes and communicates multiple important values, such as which ranks are a current rank's neighbors. The grid of subdomains is divided on each dimension, which results in a regular grid.

After each rank has its defined subdomain, the subdomains are further split into bins, in order to apply the previously mentioned link-cell method when building the neighborhood lists. This is achieved inside the `neighborAndAtom.c` module.

One important aspect to note is that all the atoms are generated on rank 0, not on their corresponding subdomain to which the other ranks represent. This means that a transfer of atoms will be required so that each atom is moved to their corresponding rank. This transfer is done after by the `exchange` function of the `comm.c` module. This is a fairly large function that also applies global boundary conditions. It will be discussed in subsection 3.3.4.

Once the atoms are in their corresponding ranks, the simulation is almost ready to start. The last thing it needs to do is to transfer the ghost cells mentioned in section 2.3 with the use of the `borders` function from the `comm.c` module. Once this is completed, the first step of the simulation is to build the neighborhood lists.

3.3.2. Building the Neighborhood Lists

In miniMD, the neighborhood list of each atom is built inside the `build` function of the `neighborAndAtom.c` module. It creates a neighborhood list for each atom in a fairly efficient manner, because it makes use of the cells, or bins, previously created. This is the method mentioned previously in section 2.3.

When creating the neighborhood lists, one of the first things the `build` function does is to bin the atoms. This is the process of assigning each atom to its corresponding bin inside the rank's subdomain, according to its coordinates. Once the binning process is complete, the `build` function can efficiently go define, for each atom, its neighbors list from its own and neighboring bins.

This procedure of binning the atoms and building their corresponding neighborhood lists is, as mentioned in section 2.3, not performed before every time step of the simulation. Instead, it is performed after a set amount of time steps. This amount is defined in the `input.c` module and can be modified for each scenario.

3.3.3. Computing Forces and Moving the Particles

During each simulation iteration, miniMD computes the force exerted unto each atom by the atoms in its neighborhood list. This is done to move the atom to its new position. This force computation is done by the `force_lj.c` module. The miniMD port on OctoPOS supports full neighborhood lists for its force computations. The force calculation follows the Lennard-Jones potential mentioned in section 2.2.

Once the force exerted on each atom by its neighbors is computed, miniMD then moves the atom through the simulation domain. This process is done by the `integrate.c` module. The change in position and velocity of each atom follows the Velocity-Verlet method[GZK07], defined as:

$$x_{n+1} = x_n + v_n \Delta t + \frac{1}{2} a_n (\Delta t)^2 \tag{3.1}$$

$$v_{n+1} = v_n + \frac{1}{2}(a_{n+1} + a_n)\Delta t \tag{3.2}$$

3.3.4. Exchanging Atoms Between Ranks

miniMD implements an important function that does multiple things, but most importantly it applies the global boundary conditions, depending on the scenario, and it exchanges atoms between ranks, in the case where the atom's position is outside the boundaries of its current rank. The function in question is `exchange`, from the `comm.c` module.

This function first calls one of the boundary condition functions from the `atom.c` module. Then, `exchange` performs a seemingly complex set of operations to exchange atoms between ranks, and it does so one dimension at a time. In listing 3.2, a very small extract shows how, on one dimension, each atom in the rank is looped upon. It first verifies if the atom is outside the global boundaries (since the boundary conditions have already been applied at this point in the function). Then, it checks if the atom has left the subdomain's boundaries and notes its index, so that it can later be sent to a neighboring rank.

```
void exchange(Comm *comm, Atom *atom) {
    ...

    for (int i = 0; i < nlocal; i++) {
      if (x[i * PAD + idim] < lprd || x[i * PAD + idim] >= rprd)
        printf("DEBUG: wrong loc %f\n", x[i * PAD + idim]);
      if (x[i * PAD + idim] < lo || x[i * PAD + idim] >= hi) {
        if (nsend >= comm->maxsend_thread[tid]) {
          comm->maxsend_thread[tid] = nsend + 100;
          comm->exc_sendlist_thread[tid] = (int *)realloc(
              comm->exc_sendlist_thread[tid], (nsend + 100) * sizeof(int));
        }
        comm->exc_sendlist_thread[tid][nsend++] = i;
        comm->send_flag[i] = 0;
      } else {
        comm->send_flag[i] = 1;
      }
    }

    ...
}
```

Listing 3.2: Function applying the periodic boundary conditions.

Part II.

Thesis Development

4. Naive Dynamic Load Balancing

Performing dynamic load balancing on miniMD can be broken down into a few steps. First, there needs to be a way to detect that our processor perform an imbalanced amount of work. Second, a heuristic must be used to determine new subdomains for each processor, in hopes of balancing the workload between them. Third, atoms need to be transferred from one processor's memory to another, since changing the subdomain bounds of a processor might add or remove atoms from its subdomain.

For simplicity, the processor topology used for implementing dynamic load balancing is a split on one dimension, as can be seen in figure 4.1. All the load balancing methods can also be extended to support load balancing in two or three dimensions.

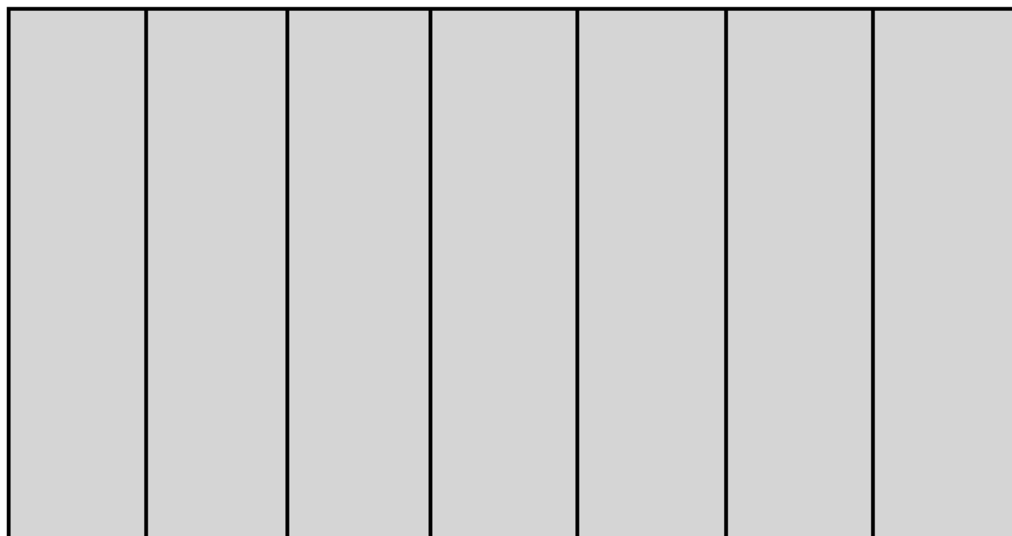


Figure 4.1.: Example of how the domain is divided for dynamic load balancing.

This division in one dimension is done inside the `comm_setup` function of the `comm.c` module. Initially, miniMD would determine the number of subdomains in each direction by its own heuristic. Instead, with dynamic load balancing enabled, the number of subdomains is only set on direction as the number of processing units given to the simulation, as seen in listing 4.1. `comm->procgrid` is an array holding the number of subdomains in each of the three dimensions, and `nprocs` is the number of processors assigned to the simulation.

```
1 // The processor layout when load balancing is enabled needs to be only on
    the x dimension.
2 void create_processor_layout_for_load_balancing(Comm *comm, int nprocs) {
```

```
3     comm->procgrid[0] = nprocs;
4     comm->procgrid[1] = 1;
5     comm->procgrid[2] = 1;
6 }
7
8 /* setup spatial-decomposition communication patterns */
9 int comm_setup(Comm *comm, MMD_float cutoff_distance, Atom *atom, In *in,
      int currently_load_balancing) {
10     ...
11
12     #ifndef LOAD_BALANCE
13         create_processor_layout(comm, prd, nprocs);
14     #else
15         create_processor_layout_for_load_balancing(comm, nprocs);
16     #endif
17
18     ...
19 }
```

Listing 4.1: Setting up the initial number of subdomains by dimension.

As seen in listing 4.1, `LOAD_BALANCE` is set by a macro defined in `types.h`, as seen in listing 4.2. This is also used in a few other places to execute dynamic load balancing code.

```
1 // LOAD_BALANCE: define for dynamic load balancing, do not define for no
      load balancing
2 #ifndef LOAD_BALANCE
3 #define LOAD_BALANCE
4 #endif
```

Listing 4.2: Defining the LOAD_BALANCE macro.

Multiple load balancing schemes were implemented for the purpose of this thesis in miniMD. The first one is a scheme that takes a very simple approach, possibly one of the most straightforward ones, hence its name: naive.

4.1. Detecting a Load Imbalance

The first step to implement dynamic load balancing is to have the ability to detect when there is a load imbalance between the processing units.

4.1.1. Defining Work

Each processing unit performs an amount of work during each simulation iteration. This work can be defined in different manners. It could take the form of the time elapsed to complete a simulation iteration, which would require time tracking of the processing units for each simulation iteration, as well as communication of it between the processing units. Instead, I chose to use a simpler value for the amount of work, which is already being held in each processing unit: the amount of atoms in the processing unit's subdomain.

This approach to evaluating the work functions well, because the number of atoms is directly proportional to the amount of time a processing unit takes to perform a simulation iteration.

This is because the more atoms a processing unit has, the more computation it needs to perform to build the neighborhood lists for each atom, as well as the to evaluate the force and move the atoms.

4.1.2. Synchronizing Work Between Processing Units

In order to synchronize the work done by each processing unit, at the beginning of each simulation iteration inside the `run` function of the `run.c` module, there is a loop broadcasting each processing unit's number of atoms to all the others, so that they can store the work of each processor inside an array, to later use for determining if there is a load imbalance. This can be seen in listing 4.3. It also synchronizes the subdomain size (in the x dimension) of each processing unit to be later used.

```
1  // Gather the number of atoms on each rank to check for load imbalance
2  float *number_of_atoms_per_rank = (float *) malloc(sizeof(float) * size);
3  float *subdomain_size_per_rank = (float *) malloc(sizeof(float) * size);
4
5  // Makes all ranks have the required data to make load balancing decisions.
6  for (int i = 0; i < size; i++) {
7    int current_count = atom->nlocal;
8    MMD_float subdomain_size = atom->box.xhi - atom->box.xlo;
9
10   MPI_Bcast(&current_count, 1, MPI_INT, i, MPI_COMM_WORLD);
11   MPI_Bcast(&subdomain_size, 1, MPI_FLOAT, i, MPI_COMM_WORLD);
12
13   number_of_atoms_per_rank[i] = (float) current_count;
14   subdomain_size_per_rank[i] = subdomain_size;
15 }
```

Listing 4.3: Synchronizing the number of atoms held in each processing unit.

The reason a broadcast inside a loop was used instead of `MPI_Gather` is because that function is not supported in iMPI's implementation on OctoPOS.

4.1.3. Computing the Imbalance

Once the work on each processing unit is synchronized between them, the load imbalance can be computed. This is done by computing the standard deviation of the work on each processing unit. Then, the standard deviation is compared with a previously defined tolerance. If it is higher than the tolerance, then load balancing is performed.

The tolerance is defined inside the `run.c` module, as can be seen in listing 4.4.

```
1 // The tolerance is calculated as a percentage of the quantity of interest
2 // (the quantity that represents the work on a rank), divided by the number
3 // of ranks.
4 //
5 // e.g. If we use the number of atoms as the quantity of interest, and we
6 //      have 2400 atoms split on 4 ranks, for a PERCENT_TOLERANCE of 0.15
7 //      (15%), we have a TOLERANCE of (2400 / 4) * 0.15 = 90.
8 const float PERCENT_TOLERANCE = 0.15;
```

```
9 const float TOLERANCE = atom->natoms / size * PERCENT_TOLERANCE;
```

Listing 4.4: Defining the tolerance.

The user defined value is the `PERCENT_TOLERANCE`, which represents the percentage of how far from the ideal load balance we can be to be considered "balanced". Then, the `TOLERANCE` is computed using the magnitude of the work on each processing unit, as can be seen in equation 4.1. Since the load imbalance is determined by comparing the standard deviation of each processing unit's number of atoms with the tolerance, the tolerance should be scaled up or down to that work value.

$$TOLERANCE = \frac{total_number_of_atoms}{number_of_processing_units} * PERCENT_TOLERANCE \quad (4.1)$$

Using this computed `TOLERANCE` and the work on each processing unit, miniMD can now determine whether there is a load imbalance. This is done in a newly create module called `load_balancer.c`, which contains a function named `load_balancing_needed`. This function computes the standard deviation of the array containing the work on each processing unit (called `quantities_of_interest` to allow for different definitions of work, other than the number of atoms). This function can be seen in listing 4.5.

```
1  /*
2  Function determining whether there is a significant load imbalance between
       the ranks.
3
4  Currently takes an array of values (one of each rank), computes their
       standard deviation,
5  and determines whether the standard deviation is larger than the given
       tolerance. If it
6  is, then there is a significant load imbalance.
7
8  Can be extended to support different heuristics for determining significant
       load imbalances.
9
10 Returns 1 if there is a significant load imbalance.
11 Returns 0 if there isn't a significant load imbalance.
12 */
13 int load_balancing_needed(int number_of_ranks, float tolerance, float*
       quantities_of_interest) {
14     int rank;
15     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16
17     float sum = 0.0;
18     float squared_sum = 0.0;
19
20     for (int i = 0; i < number_of_ranks; i++) {
21         sum += quantities_of_interest[i];
22     }
23
24     float mean = sum / (float) number_of_ranks;
25
26     for (int i = 0; i < number_of_ranks; i++) {
```

```
        float deviation = quantities_of_interest[i] - mean;
        squared_sum += deviation * deviation;
    }

    float standard_deviation = sqrt(squared_sum / (float) number_of_ranks);

    return standard_deviation <= tolerance ? 0 : 1;
}
```

Listing 4.5: Function determining whether there is a load imbalance.

4.2. Determining New Subdomains

Once it has been established that, in the current simulation iteration, there is a load imbalance, there can be a balancing to try to reduce the standard deviation of the work performed on each processing unit. The goal is to reach a point where the standard deviation is lower than the `TOLERANCE`.

To balance the work between processing units, which is measured in number of atoms for our implementation, we can change the area each subdomain owns. This increases or decreases the number of atoms owned by a processing unit. We can perform this subdomain change for all processing units in the same simulation iteration, or we can just change the subdomain of a pair of adjacent subdomains (by increasing one and decreasing the other).

4.2.1. The Naive Heuristic

For the naive load balancing scheme, the latter was chosen, to keep the heuristic as simple and straightforward as possible. The steps that the naive scheme needs to perform are the following:

1. Finding the processing unit that does the least amount of work.
2. Choosing one of the two adjacent neighboring subdomains to resize with.
3. Changing the size of the pair of subdomains.

Each of those steps have been separated into their own functions inside the `load_balancer.c` module. They are called from the `run` function inside the `run.c` module after determining that there is a load imbalance, as seen in listing 4.6.

```
int rank_with_smallest_load;
int exchange_with_rank;

rank_with_smallest_load = min_quantity_of_interest_rank(size,
    number_of_atoms_per_rank, subdomain_size_per_rank);

exchange_with_rank = neighbor_to_exchange_with(size, rank_with_smallest_load
    , number_of_atoms_per_rank);

resized = change_sub_domains_naive(rank_with_smallest_load,
    exchange_with_rank, atom, minimum_subdomain_size);
```

Listing 4.6: Calling the relevant functions for the naive load balancing implementation.

The logic of choosing to take the processing unit with the least amount of work done and to take more work from an adjacent subdomain is done slowly balance the work between processors.

4.2.2. Finding the Processing Unit With the Least Amount of Work

To find the processing unit with the least amount of work performed, a function named `min_quantity_of_interest_rank` from the `load_balancer.c` module is called. This function, which can be seen in listing 4.7, loops through each subdomain sequentially, keeping track of the processing unit with the lowest amount of work done.

There might be a case where multiple processing units have the same amount of work (especially when there are multiple subdomains with zero atoms in them). When this happens, `min_quantity_of_interest_rank` chooses the processing unit with the smallest subdomain, because the larger one already has more chances of an atom landing in it. In any case, if there is still a tie on the next simulation iteration, and the previously changed subdomain is larger the one that was larger in the last iteration, it will get increased as well.

```c
/*
Helper function to find the minimum quantity of interest's rank.
If it's a tie, choose the rank with the smallest subdomain.
*/
int min_quantity_of_interest_rank(int number_of_ranks, float*
    quantities_of_interest, float* subdomain_size_per_rank) {
    int minimum_rank = 0;
    float current_minimum = quantities_of_interest[0];
    float current_minimum_rank_subdomain_size = subdomain_size_per_rank[0];

    for (int i = 1; i < number_of_ranks; i++) {
        if (quantities_of_interest[i] < current_minimum || (
    quantities_of_interest[i] == current_minimum && subdomain_size_per_rank[
    i] < current_minimum_rank_subdomain_size)) {
            current_minimum = quantities_of_interest[i];
            current_minimum_rank_subdomain_size = subdomain_size_per_rank[i
    ];
            minimum_rank = i;
        }
    }

    return minimum_rank;
}
```

Listing 4.7: Function finding the rank with the minimum work done.

4.2.3. Determining Which Adjacent Processing Unit to Balance With

Once the rank with the smallest workload has been determined, the adjacent rank to change the subdomain's boundaries with needs to be chosen. We can recall in figure 4.1 that the domain is divided only on one dimension, meaning that a processing unit can only have at most two adjacent processing units: one to the left and one to the right. Of course, if the

processing unit with the least amount of work is at an edge, it can only have one adjacent processing unit.

To determine which one of the adjacent processing units to alter subdomains with, the `run` function calls the `neighbor_to_exchange_with` function from the `load_balancer.c` module, which can be seen in listing 4.8.

The function first checks if the processing unit with the lowest workload is at an edge, and returns the only possible adjacent rank. If it isn't on the edge of the domain, then it chooses the adjacent rank with the highest workload, to reduce the load imbalance as much as possible.

```
/*
Helper function to find which neighbor to exchange with.
*/
int neighbor_to_exchange_with(int number_of_ranks, int lowest_rank, float*
    quantities_of_interest) {
    if (lowest_rank == number_of_ranks - 1 && lowest_rank != 0) {
        return lowest_rank - 1;
    }

    if (lowest_rank == 0 && lowest_rank != number_of_ranks - 1) {
        return lowest_rank + 1;
    }

    return quantities_of_interest[lowest_rank - 1] > quantities_of_interest[
    lowest_rank + 1] ? lowest_rank - 1 : lowest_rank + 1;
}
```

Listing 4.8: Function determining which adjacent processing unit to alter subdomains with.

4.2.4. Changing the Subdomain Boundaries

After the pair of processing units has been chosen, their subdomain alterations can be done. This is implemented in the `change_sub_domains_naive` function of the `load_balancer.c` module. Since this function is fairly large, an explanation of each part of it is described below.

First, we need to determine the amount of the subdomain with the higher workload that we want to transfer to the subdomain with the lowest workload. This is done in listing 4.9 by computing the length of the subdomain with the higher workload in the x dimension, and then taking 10% of it.

After the interval in the x dimension that we want to add to the subdomain with the lowest workload has been computed, it is sent and received through a pair of `MPI_Send` and `MPI_Recv` calls.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int resized = 1;

MMD_float interval_to_add_to_smallest;

```

```
8  if (rank == exchange_with_rank) {
9      interval_to_add_to_smallest = (atom->box.xhi - atom->box.xlo) / 10;
10     MPI_Send(&interval_to_add_to_smallest, 1, MPI_MMDFLOAT,
       rank_with_smallest_load, 0, MPI_COMM_WORLD);
11 }
12
13 if (rank == rank_with_smallest_load) {
14     MPI_Status status;
15     MPI_Recv(&interval_to_add_to_smallest, 1, MPI_MMDFLOAT,
       exchange_with_rank, 0, MPI_COMM_WORLD, &status);
16 }
```

Listing 4.9: Determining how much of the subdomain to transfer to the rank with the lowest workload.

Once the interval is synchronized between the pair of processing units, there are two branches in which they can proceed. This depends on whether the processing unit with the higher workload is on the right or the left of the processing unit with the lowest workload.

If it is on the right side, then the branch in listing 4.10 is executed.

```
1  if (rank_with_smallest_load < exchange_with_rank) {
2      if (rank == exchange_with_rank) {
3          if (!can_make_smaller_from_the_left(interval_to_add_to_smallest,
       atom->box.xlo, atom->box.xhi, minimum_subdomain_size)) {
4              resized = 0;
5          }
6
7          MPI_Send(&resized, 1, MPI_INT, rank_with_smallest_load, 0,
       MPI_COMM_WORLD);
8
9          if (resized) {
10             atom->box.xlo = atom->box.xlo + interval_to_add_to_smallest;
11         }
12     }
13
14     if (rank == rank_with_smallest_load) {
15         MPI_Status status;
16         MPI_Recv(&resized, 1, MPI_INT, exchange_with_rank, 0, MPI_COMM_WORLD
       , &status);
17
18         if (resized) {
19             atom->box.xhi = atom->box.xhi + interval_to_add_to_smallest;
20         }
21     }
22 }
```

Listing 4.10: Changing the subdomains when the subdomain with the bigger workload is on the right.

This branch first checks, for the subdomain with the higher workload, if it can be made smaller, through the call to `can_make_smaller_from_the_left`. This function verifies whether the subdomain's length in the x dimension will reach a length that is smaller than a defined minimum subdomain size.

This minimum subdomain size is set inside the `run.c` module to be 5% of the simulation domain's length in the x dimension. This minimum size was implemented to avoid having subdomains that reach an extremely small size, as that created some abnormal behavior where atoms were moved in unrealistic locations between simulation iterations.

If the resizing of the subdomains fails this minimum subdomain size condition, the `resized` variable, which was initially set to `1`, gets set to `0`. This is then synchronized between the pair of processing units to decide whether to change the subdomain coordinates through a pair of `MPI_Send` and `MPI_Recv` calls.

If the `resized` variable is still `1`, then the coordinates of the subdomains edges in the x dimension are changed for both processing units, using the previously calculate interval to add or remove.

In the opposing case, where the processing unit with the higher workload is on the left side, the branch is almost identical, with its difference being its logic on opposing sides.

4.3. Transferring Atoms to their New Processing Units

After changing the boundaries of the subdomains in the x dimension, some atoms need to be moved from one processing unit to another. This is what also changes the workload of the processing units.

Initially, I had planned to implement this exchange, possibly by synchronizing all the atoms to one processing unit, and then distributing them to their corresponding subdomains. However, miniMD already implements a function that moves atoms between processing units, when their position change in a simulation iteration makes them jump far enough to reach another subdomain.

Actually, miniMD implements multiple useful functions to move atoms between processing units and to set up certain variables before or after this exchange is done. After changing the subdomains, during a load balancing iteration, calls to those functions are made, sequentially, as seen in listing 4.11.

```
1  // This part calls a few functions that already exist from miniMD to
       transfer atoms to their new
2  // respective ranks, as well as set the ghost cells and recreate the
       neighborhood list.
3  MPI_Barrier(MPI_COMM_WORLD);
4  comm_setup(comm, neighbor->cutoff_distance, atom, &in, 1);
5
6  MPI_Barrier(MPI_COMM_WORLD);
7  exchange(comm, atom, 0);
8
9  MPI_Barrier(MPI_COMM_WORLD);
10 borders(comm, atom);
11
12 MPI_Barrier(MPI_COMM_WORLD);
13 build(neighbor, atom);
```

Listing 4.11: Calling miniMD's functions to move atoms between processing units.

These four functions are also called by miniMD when setting up the simulation, as well as at the end of each set amount of simulation iterations (except `comm_setup`, which only gets called when setting up the simulation).

4.3.1. The `comm_setup` function

The `comm_setup` function sets up the boundaries of each subdomain, and initializes important variables and containers. I modified its signature to take a flag to signify that it is being called from a load balancing iteration and not during the setup. This flag can be seen in listing 4.11, as its last parameter. Using the flag, it skips on a few console prints, as well as does not overwrite the subdomain boundaries (which were previously changed during the load balancing iteration).

4.3.2. The `exchange` function

The `exchange` function is in charge of moving atoms to their correct processing unit. It is used during the simulation to move atoms between subdomains when the simulation iteration made their position change outside the boundaries of their current subdomain. It is also called at the beginning of a simulation because atoms are initialized in one processing unit, and then moved using this `exchange` function. I also added a flag to the signature of this function, to enable or disable the application of boundary conditions, as seen in the last parameter of the function call in listing 4.11. This flag disables boundary conditions during load balancing iterations, because otherwise there is a bug produced. At the end of the simulation iteration, the boundary conditions are still applied, so it does not affect the results of the simulation.

4.3.3. The `borders` function

The `borders` function is responsible for communicating atoms forming ghost cells between processing units. This function is important to call after changing the boundaries of subdomains because entirely new atoms form the ghost cells between subdomains.

4.3.4. The `build` function

The `build` function, implemented in the `neighborAndAtom.c` module as opposed to the `comm.c` module in which the previously mentioned functions are implemented, creates neighborhood lists for each atom. These neighborhood lists, mentioned in section 2.3, are important to avoid useless force computations between pairs of atoms that are far away from each other. This function is also important to call after performing a load balancing iteration because the atoms that were moved between processing units will need to create new neighborhood lists with the atoms in their new subdomain.

These four functions, which were minimally modified to allow to be called in a load balancing iteration, allowed me to reuse a lot of very useful code from miniMD implementation. They provide an easy way to move atoms between processing units after changing the boundaries of subdomains, as seen in listing 4.10.

5. Improved Load Balancing Schemes

The naive dynamic load balancing iteration shown in chapter 4 gave positive results in execution time. However, intuitively, it seemed inefficient because it only updated the subdomains of one pair of adjacent processing units per load balancing iteration.

Ideally, the subdomains of all processing units would be changed in each load balancing iteration, to reduce the number of load balancing iterations. This can potentially be an improvement in the time taken to reach a load imbalance within the defined tolerance.

5.1. Swipe Load Balancing

As the name suggests, swipe load balancing performs a swipe over the x dimension, changing the subdomains of each pair of adjacent processing units. This swiping approach was inspired by the Tensor Method in the A Loadbalancing Library [HSS99].

This swipe load balancing uses the same logic as the naive load balancing to determine whether there is a load imbalance or not, as well as to transfer atoms between processing units. The differing code is the function that changes the subdomain edges. Instead of calling `min_quantity_of_interest_rank`, `neighbor_to_exchange_with`, and `change_sub_domains_naive` , it makes a single call to `change_sub_domains_swipe` from the `load_balancer.c` module, as seen in listing 5.1.

```
1  switch (load_balancing_scheme) {
2  case NAIVE:
3    int rank_with_smallest_load;
4    int exchange_with_rank;
5
6    rank_with_smallest_load = min_quantity_of_interest_rank(size,
       number_of_atoms_per_rank, subdomain_size_per_rank);
7
8    exchange_with_rank = neighbor_to_exchange_with(size,
       rank_with_smallest_load, number_of_atoms_per_rank);
9
10   resized = change_sub_domains_naive(rank_with_smallest_load,
       exchange_with_rank, atom, minimum_subdomain_size);
11
12   break;
13 case SWIPE_INVERTED:
14 case SWIPE_ALL:
15   resized = change_sub_domains_swipe(size, number_of_atoms_per_rank, atom,
       minimum_subdomain_size, load_balancing_scheme);
16
17   break;
```

```
18 }
```

Listing 5.1: Branching between the naive and the swipe load balancing schemes.

`change_sub_domains_swipe` iterates through pairs of processing units from the left to the right of the x dimension. For each pair, it compares the work performed by those processing units, and decides a new position for the edge separating them. Then, it changes both subdomains and continues iterating through the pairs.

In the end, each subdomain gets resized twice, once with its left adjacent subdomain, and another time with its right adjacent subdomain. This, of course, does not apply for the two subdomains at the edges of the domain.

One challenge with this approach is calculating the new workload on each processing unit after changing the subdomain edges during the swipe. This is because, during the swipe through all adjacent pairs of subdomains, atoms are never exchanged between processing units (otherwise this would the overhead of a full load balancing iteration for each pair of subdomains, beating the purpose of changing them all at once).

The approach chosen to counter this challenge was to estimate the new workloads. This estimation is done proportional to the subdomain size increase or decrease. The assumption used for this approach is that atoms are evenly distributed in a subdomain. Even though they are almost never evenly distributed, it does give a somewhat good estimate for the purpose of applying the load balancing scheme to decide how much and in which direction to move the edge between subdomains.

The `change_sub_domains_swipe` function can be seen in listing 5.2. It keeps a pointer to a left and right processing unit, and loops through each adjacent pair from the left to the right of the x dimension. It performs some logic to change the subdomains on the left and right processing units, and then synchronizes the new estimated workload of the right processing unit using a `MPI_Bcast`.

```
1  int change_sub_domains_swipe(int size, float* quantities_of_interest, Atom *
       atom, MMD_float minimum_subdomain_size, enum LoadBalancingScheme scheme)
        {
2      int rank;
3      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4
5      int resized = 0;
6
7      int left_rank = 0;
8      int right_rank;
9
10     for (int i = 1; i < size; i++) {
11         right_rank = i;
12
13         MMD_float new_edge_position;
14         float new_estimated_quantity_of_interest_right_rank;
15
16         if (rank == left_rank) {
17             ...
18         }
19
```

```
20          if (rank == right_rank) {
21              ...
22          }
23
24          left_rank = right_rank;
25
26          MPI_Bcast(&new_estimated_quantity_of_interest_right_rank, 1,
    MPI_MMDFLOAT, right_rank, MPI_COMM_WORLD);
27
28          quantities_of_interest[right_rank] =
    new_estimated_quantity_of_interest_right_rank;
29      }
30
31      return resized;
32  }
```

Listing 5.2: Base of the function that changes subdomains in the swipe manner.

Of course, since the swipe happens from left to right, after a processing unit is resized when it's the left one of the pair, its workload is not relevant for the remaining calculations of the swipe, so its new workload is not estimated.

When a processing unit is assigned as the left one of the pair, it must send the position of its left subdomain edge, as well as receive the position of the right subdomain edge of the right processing unit of the pair. These two values are synchronized using a pair of `MPI_Send` and `MPI_Recv`, as seen in listing 5.3.

Then, the new edge position for the edge between the subdomain is calculated using the chosen load balancing scheme. Once the new position has been determined, a check is performed to verify if change the subdomains would result in a subdomain smaller than the minimum subdomain size.

Once the check is done, the subdomain of the left processing unit is changed.

```
1   if (rank == left_rank) {
2       MMD_float left_rank_left_edge = atom->box.xlo;
3       MMD_float right_rank_right_edge;
4
5       MPI_Send(&left_rank_left_edge, 1, MPI_MMDFLOAT, right_rank, 0,
    MPI_COMM_WORLD);
6
7       MPI_Status status;
8       MPI_Recv(&right_rank_right_edge, 1, MPI_MMDFLOAT, right_rank, 0,
    MPI_COMM_WORLD, &status);
9
10      switch (scheme) {
11          case SWIPE_INVERTED:
12              new_edge_position = get_new_edge_using_inverted_pressure(
13                  atom->box.xhi,
14                  quantities_of_interest[left_rank],
15                  quantities_of_interest[right_rank],
16                  left_rank_left_edge,
17                  right_rank_right_edge
18              );
19
```

```
20              break;
21          case SWIPE_ALL:
22              new_edge_position = get_new_edge_using_all(
23                  atom->box.xhi,
24                  quantities_of_interest[left_rank],
25                  quantities_of_interest[right_rank],
26                  left_rank_left_edge,
27                  atom->box.yhi - atom->box.yylo
28                  lo
29              );
30      }
31
32      if (new_edge_position - left_rank_left_edge >= minimum_subdomain_size &&
        right_rank_right_edge - new_edge_position >= minimum_subdomain_size) {
33          atom->box.xhi = new_edge_position;
34
35          resized = 1;
36      }
37  }
```

Listing 5.3: Logic performed by the left processing unit of the pair in a swipe iteration.

The logic performed on the right processing unit of the pair is very similar to the one shown in listing 5.3 for the left processing unit.

The key difference is the computation of the estimated new workload of the right processing unit. This can be seen in listing 5.4.

```
1  if (new_edge_position - left_rank_left_edge >= minimum_subdomain_size &&
       right_rank_right_edge - new_edge_position >= minimum_subdomain_size) {
2      // This value assume an even distribution of atoms in the domain.
3      // This is estimated in order to avoid having to calculate the
4      // actual new quantity of interest during load balancing.
5      new_estimated_quantity_of_interest_right_rank = quantities_of_interest[
       right_rank] + quantities_of_interest[right_rank] * (atom->box.xlo -
       new_edge_position) / (atom->box.xhi - atom->box.xlo);
6
7      atom->box.xlo = new_edge_position;
8
9      resized = 1;
10 }
```

Listing 5.4: Changing the subdomain of the right processing unit of the pair for swipe load balancing.

The last piece missing from the `change_sub_domains_swipe` function is the computation of the new position of the edge between subdomains. For the purpose of this thesis, I chose to introduce two schemes:

1. Inverted Pressure Scheme
2. Tensor Method Scheme

Both load balancing scheme implementations in miniMD have been modified to fit the unique needs of its implementation.

An illustration of what these two schemes aim to solve can be seen in figure 5.1. When the swiping algorithm balances each pair of adjacent subdomains, it attempts to find a corresponding new position for x_1 using the weights of both subdomain A and B.

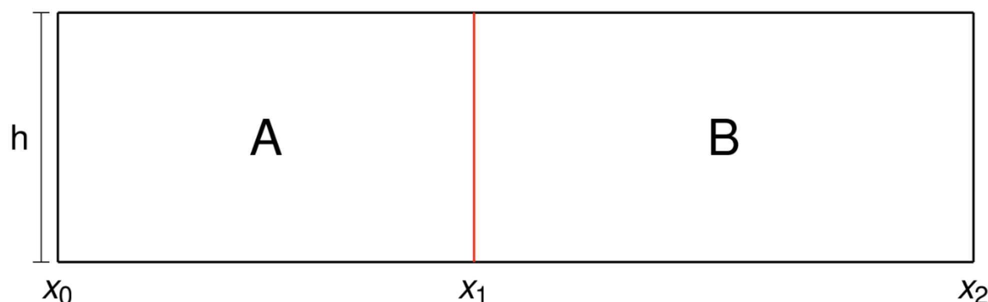


Figure 5.1.: The problem that the two load balancing schemes aim to solve: finding an appropriate position for the common edge.
Source: [Koe21]

5.1.1. Inverted Pressure Scheme

The first of the two schemes is the Inverted Pressure Scheme, inspired by the implementation from the AutoPas Demonstrator MD-Flexible thesis [Koe21].

This scheme requires the position of the left boundary of the left subdomain (A), the right boundary of the right subdomain (B), and the weight for each processing unit in the pair.

In order to define the new position of x_1, equation 5.1 is applied [Koe21]. W represents the workload of the corresponding subdomain in its subscript.

$$x_1' = \frac{W_B * x_2 + W_A * x_0}{W_A + W_B} \tag{5.1}$$

This method of defining the new edge position has an edge case for miniMD, where one of the workloads can be zero. This happens when a subdomain does not have any atoms (something quite common if the simulation scenario only has a concentration of atoms in a certain area of the domain). When this is the case, the new edge position will make it equal to either x_0 or x_2, resulting in one subdomain to cease to exist.

In order to counter this problem, the Inverted Pressure Scheme was modified for miniMD. Its new version can be seen in equation 5.2.

$$x_1' = \frac{W_B * x_2 + (W_B + W_A) * x_1 + W_A * x_0}{2 * (W_A + W_B)} \tag{5.2}$$

This new version puts the same amount of weight in the current position of the edge between the two subdomains as the amount put on the outer edges. This fixes the subdomain vanishing problem for processing units with a workload of zero.

The downside of this adjustment is that it will also update the common edge position by a smaller amount on each load balancing iteration. This, however, could also be a benefit, since it might trigger the minimum subdomain size condition seen in listing 5.3 more rarely.

The code of the function being called inside `change_sub_domains_swipe` can be seen in listing 5.5.

```
MMD_float get_new_edge_using_inverted_pressure(MMD_float current_edge_position, float left_weight, float right_weight, MMD_float left_rank_left_edge, MMD_float right_rank_right_edge) {

    float total_weight = left_weight + right_weight;

    return (left_weight * left_rank_left_edge + right_weight * right_rank_right_edge + total_weight * current_edge_position) / (2 * total_weight);
}
```

Listing 5.5: Determining the new edge using the Inverted Pressure Scheme.

5.1.2. Tensor Method (ALL) Scheme

The second of the two schemes is the Tensor Method Scheme from the ALL library, applied to one dimension [HSS99].

This scheme requires the left boundary of the left subdomain (A), the position of the edge between the two subdomains, the height of the subdomains (h), and the weight for each processing unit in the pair.

The update equation for the common edge is defined in the AutoPas Demonstrator MD-Flexible thesis [Koe21] in the same way as in equation 5.3.

$$x'_1 = x_1 + \frac{W_B - W_A}{2 * \gamma * (W_A + W_B) * ((x_1 - x_0) * h + W_B)} \tag{5.3}$$

The gamma (γ) in the equation is dynamically set in the ALL library for each iteration [HSS99]. It is calculated using equation 5.4.

$$\gamma = MAX(4.1, 2 * (1 + \frac{MAX(left_subdomain_size, right_subdomain_size)}{MIN(left_subdomain_size, right_subdomain_size)})) \tag{5.4}$$

The gamma, as defined by the ALL library, will always wield an extremely small change for x'_1. This usually makes miniMD load balance on every single iteration, never reach a load balance within the tolerance.

To counter this problem, the gamma used for this load balancing scheme in miniMD has a hard-coded gamma of 0.0001. This value was defined empirically because it reached a load balance within the tolerance in a reasonable amount of time, without losing the clear

advantage of the Tensor Method to slowly optimize the subdomains until the workload is balanced.

The code of the function being called inside `change_sub_domains_swipe` can be seen in listing 5.6.

```
MMD_float get_new_edge_using_all(MMD_float current_edge_position, float
    left_weight, float right_weight, MMD_float left_rank_left_edge,
    MMD_float height) {

    float total_weight = left_weight + right_weight;
    float weight_difference = right_weight - left_weight;

    float gamma = 0.0001;

    return current_edge_position + weight_difference / (2 * gamma *
    total_weight * ((current_edge_position - left_rank_left_edge) * height +
     right_weight));
}
```

Listing 5.6: Determining the new edge using the Inverted Pressure Scheme.

6. Polishing the Dynamic Load Balancing

After successfully implementing all three load balancing schemes, there were a few obvious inefficiencies with regard to when a load balancing iteration should start, as well as a need to differentiate the time spent on load balancing code, to understand where exactly time improvements are made when using one of the load balancing schemes.

6.1. Some Optimizations

There were two cases when a load balancing iteration would not be needed or would not be possible in the next few simulation iterations. These are:

1. If the load imbalance is within the tolerance.
2. If a load balancing iteration started, but was unable to change the subdomain because it would create a subdomain smaller than the minimum subdomain size.

This is why a concept of load balancing iteration skips was introduced to miniMD.

A variable named `load_balancing_skips` is introduced and set to `0`. Then, on each simulation iteration, the value of the variable is reduced by `1` if miniMD does not enter a load balancing iteration. If it does enter one, then the `load_balancing_skips` variable can either:

- Remain `0`, if the load balancing iteration was successful and subdomains were changed.
- Be set to `LOAD_BALANCING_NOT_NEEDED_SKIPS` if the load imbalance is within the tolerance.
- Be set to `LOAD_BALANCING_NOT_POSSIBLE_SKIPS` if there was a failed attempt to change subdomains because it would have created a subdomain smaller than the minimum subdomain size.

These three behaviors can be seen in listing 6.1.

```
const int LOAD_BALANCING_NOT_NEEDED_SKIPS = 10;
const int LOAD_BALANCING_NOT_POSSIBLE_SKIPS = 100;

int load_balancing_skips = 0;

// Simulation iteration loop
for (n = 0; n < integrate->ntimes; n++) {

  ...

  if (load_balancing_skips == 0) {

    ...
```

```

    if (load_balancing_needed(size, TOLERANCE, number_of_atoms_per_rank)) {

      // Change the subdomains.
      ...

      for (int i = 0; i < size; i++) {
        MPI_Bcast(&resized, 1, MPI_INT, i, MPI_COMM_WORLD);

        if (resized) break;
      }

      if (resized == 0) {
        load_balancing_skips = LOAD_BALANCING_NOT_POSSIBLE_SKIPS;
      }

      // Move atoms to their new processing units.
      ...

    } else {
      load_balancing_skips = LOAD_BALANCING_NOT_NEEDED_SKIPS;
    }
  } else {
    load_balancing_skips = load_balancing_skips - 1;
  }

  // Continue simulation iteration.
  ...

}
```

Listing 6.1: High level overview of the concept of load balancing skips.

A challenge was how to determine whether subdomains were changed or not (due to the minimum subdomain size condition), to know whether `load_balancing_skips` should be set to `LOAD_BALANCING_NOT_POSSIBLE_SKIPS` or not.

The solution to that problem was to return a flag on each of the two subdomain change functions (`change_sub_domains_naive` and `change_sub_domains_swipe`). This flag, which is initially set to `0`, only gets returned as `1` from those functions if that particular processing unit had its subdomain changed.

This is why there needs to be a loop with a `MPI_Bcast` to sync the local `resized` values until there is a value of `1` or until all processing units were looped upon.

The values for `LOAD_BALANCING_NOT_NEEDED_SKIPS` and `LOAD_BALANCING_NOT_POSSIBLE_SKIPS` were determined empirically and with the following assumptions:

- If there is no load imbalance, atoms could move enough in a few iterations to cause an imbalance greater than the tolerance. This is why the number of skips for this case is small.
- If there is a load imbalance, but the subdomains can not be changed due to failing the minimum subdomain size condition, then it is likely that approximately the same

subdomain change is going to be attempted in the near future, unless the atoms have significantly moved. This is why the number of skips for this case is large.

These values can, of course, be viewed as simulation parameters and be changed according to one's needs.

6.2. Timing the Load Balancing

To be able to understand and assess the impact of dynamic load balancing on miniMD's simulations, it is important to time the amount of time spent during load balancing iterations.

Other important steps of the simulation, such as the force calculation time, neighborhood list computation time and communication time, are already timed, as seen in figure 6.1.

```
##################################################################################################################################
#MPI_PROC    #OMP_THREAD  nsteps       final_natoms TOTALTIME    OTHERTIME    INVASIVETIME FORCETIME    COMMTIME     NEIGHTIME      IOTIME
4            1            30000        9270         794.892016   58.909839    0.000000     439.082317   236.987196   59.901975      0.000000
##################################################################################################################################
```

Figure 6.1.: Performance overview at the end of a simulation.

The time spent on load balancing iterations is included in `OTHERTIME`, which is calculated as the `TOTALTIME` minus all the other, more specific, times.

To calculate the time spent on load balancing iterations, a new time type of time called `LOADBALANCETIME` was added, and the appropriate functions from the `timer.c` module were added before and after the load balancing code in the `run.c` module.

These functions calls can be seen in listing 6.2.

```
#if METRICS == 1
  if (rank == 0) time_mark(timer);
#endif

#ifdef LOAD_BALANCE
  if (load_balancing_skips == 0) {

    ...

  } else {

    ...

  }
#endif

#if METRICS == 1
  if (rank == 0) {
    time_mark(timer);
    add_time(timer, LOADBALANCETIME);
  }
#endif
```

Listing 6.2: Functions calls to capture the time spent in a load balancing iteration.

The `METRICS` flag defined in the `types.h` module enables all the timing to occur, in the same way as `LOAD_BALANCE` enables or disables dynamic load balancing on miniMD.

The result of this addition is a new column being printed at the end of simulation, as can be seen in figure 6.2.

```
################################################################################################################################
#MPI_PROC   #OMP_THREAD  nsteps    final_natoms TOTALTIME   OTHERTIME   INVASIVETIME FORCETIME   COMMTIME    NEIGHTIME   LOADBALANCETIME   IOTIME
4           1            30000     9270         639.645532  97.430073   0.000000     373.705975  92.313287   51.356213     24.839984       0.000000
################################################################################################################################
```

Figure 6.2.: Performance overview at the end of a simulation with load balancing timed.

Part III.

Results and Conclusion

7. Evaluation

7.1. Evaluation Test Scenarios

All three load balancing schemes will be evaluated on two test scenarios: a smaller one and a larger one.

Both test scenarios are similar to the `RBCCOLLISION` scenario, which uses reflective boundary collision of a water drop simulation, like in figure 3.2.

In the larger test scenario, miniMD will run 30,000 simulation iterations, so that the water droplet hits the slab and enough of the post-impact behavior is simulated. It will simulate a total of 9270 atoms on a domain of size $250 \times 100 \times 1$. The minimum subdomain size is set to 5% of the domain size.

The smaller test scenario is almost identical to the larger one, with the sole difference being the number of atoms in the simulation, which is 4361 atoms.

Each simulation configuration will be run five times and the times will be averaged to get a representative performance result. The simulation configurations are the following:

1. No load balancing
2. Load balancing using the naive scheme and a `PERCENT_TOLERANCE` of `0.15`
3. Load balancing using the inverted pressure swipe scheme and a `PERCENT_TOLERANCE` of `0.22`
4. Load balancing using the tensor method swipe scheme and a `PERCENT_TOLERANCE` of `0.25`

The `PERCENT_TOLERANCE` values were chosen empirically such that the method eventually load balances the workloads and does not attempt to load balance during the whole simulation.

7.2. Evaluation Test Environment

The simulations were run on a Lenovo Ideapad 320-15AST computer running OctoPOS on the QEMU emulator, with the following specifications:

- AMD A6-9220 Radeon r4, 5 compute cores 2c+3g $\times$ 2 Processor
- 6.7 GB RAM
- AMD Stoney Graphics

The simulation was run with 4 processing units that were initially unbalanced. For the larger test scenario, the processing units had the following number of atoms: `1120, 3492, 3538, 1120`. The load imbalance, or standard deviation, at the beginning of the simulation was `1197.61`. For the smaller test scenario, the atom distribution was `760, 1408, 1433, 760`, and the load imbalance was `330.37` at the beginning of the simulation.

7.3. Results

The averaged results of the larger and smaller simulations can be seen in tables 7.1 and 7.2 respectively.

Scheme	Total	Other	Force	Comm.	Neigh. Lists	Load Balancing
No Load Balancing	770.02	57.88	421.55	232.00	58.58	0.01
Naive	613.90	78.20	383.10	82.47	53.18	16.94
Inverted Pressure	831.54	73.34	400.13	80.64	55.04	222.39
Tensor (ALL)	642.82	97.11	377.90	90.11	53.07	24.63

Table 7.1.: Time spent on each simulation part by the different load balancing schemes for the larger test scenario.

Scheme	Total	Other	Force	Comm.	Neigh. Lists	Load Balancing
No Load Balancing	339.13	36.19	126.43	154.23	22.27	0.01
Naive	267.29	48.25	116.22	67.09	21.26	14.46
Inverted Pressure	260.75	53.34	116.65	54.71	21.47	14.58
Tensor (ALL)	277.10	46.86	122.91	69.62	21.34	16.37

Table 7.2.: Time spent on each simulation part by the different load balancing schemes for the smaller test scenario.

We can observe that, for both test scenarios, the total simulation time was reduced by around 20-25% for the best performing load balancing scheme, compared to the run without any load balancing.

Both the naive and the tensor method (ALL) performed consistently better than the base case in both test scenarios, with the naive method performing slightly better. However, the inverted pressure method had a great variability between the two test scenarios. This can be better seen in figures 7.1 and 7.2.

In the larger test scenario, seen in figure 7.1, the inverted pressure method spent a fairly large amount of time in the load balancing portion of the simulation. This was because it did not reach a load imbalance within the tolerance for a while.

The reason why finding an acceptable load imbalance took so long for this method, compared to the others, is that the subdomains in the inverted pressure method change by a higher amount on each simulation iteration than for the other methods. This could lead to a slightly

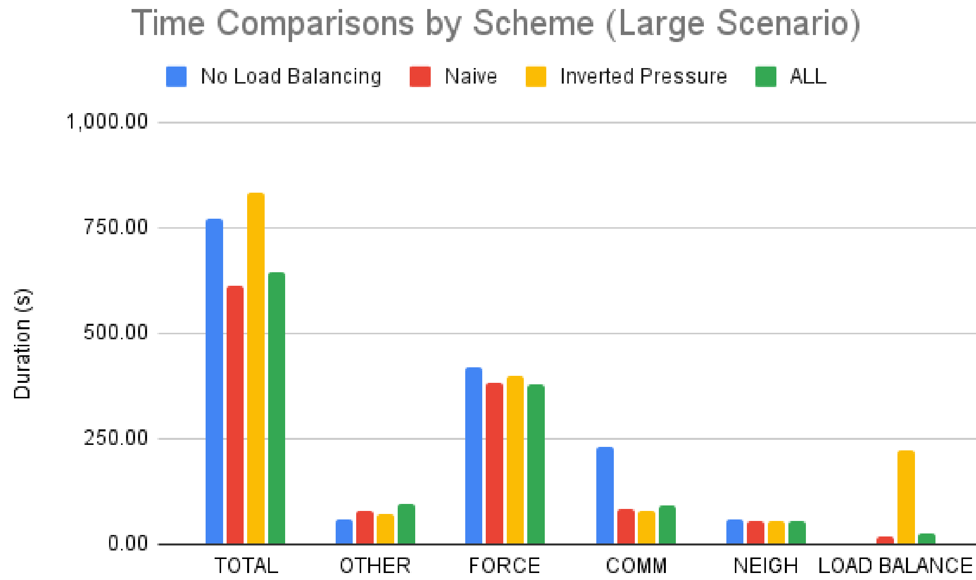


Figure 7.1.: Time comparisons by load balancing schema for the scenario with 9270 atoms.

smaller amount to reach a load balance, or to a vastly larger amount of time, like in this case. For this particular scenario, some subdomain edge positions probably oscillated for a while due to the large changes applied on them, until they finally reached a load balance.

On the other hand, for the smaller test scenario shown in figure 7.2, the inverted pressure method outperformed the other load balancing methods. This is because of a combination of quickly reaching a load imbalance within the tolerance, as well as having a more balanced work distribution between the processing units.

This high variability in results makes the inverted pressure method's performance more dependent on the user defined load balancing parameters like the `PERCENT_TOLERANCE` or the minimum subdomain size than the naive and tensor method, but it can outperform them given the right parameters for the test scenario.

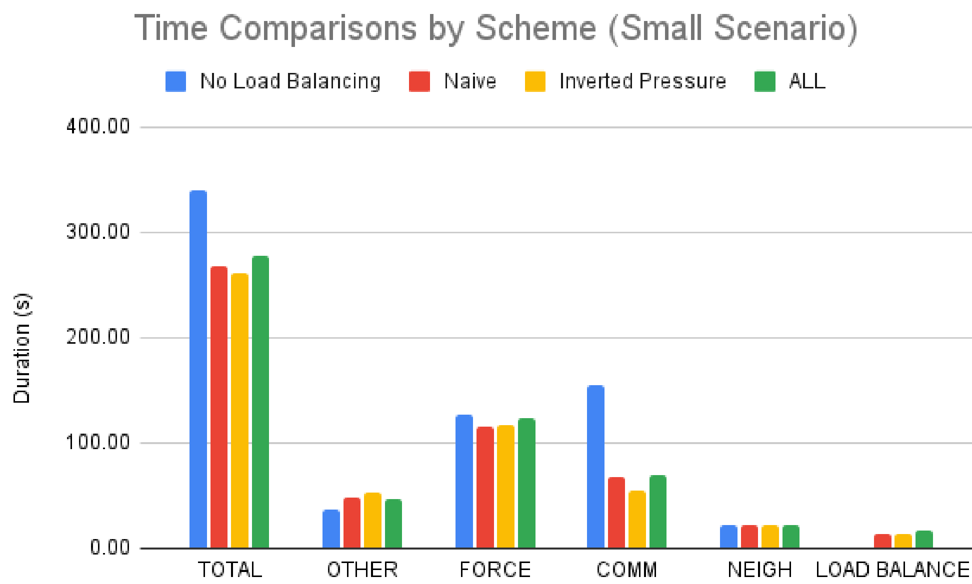


Figure 7.2.: Time comparisons by load balancing schema for the scenario with 4361 atoms.

8. Conclusion

8.1. Goals Achieved

During this master thesis, I was able to successfully add dynamic load balancing to miniMD's port in OctoPOS.

This addition included some refactoring work to make certain parts of miniMD more modular, in order to implement multiple load balancing schemes in such a way that they can be easily extended and modified.

Three different schemes were implemented:

1. A naive scheme that updates a single pair of adjacent processing units on every iteration.
2. A modified version of the inverted pressure method, using a swipe to modify all subdomains in a single iteration.
3. The tensor method from the ALL library applied in a single dimension, with a static gamma that keeps its advantage of a slower change on each load balancing iteration, while reaching a load balance in a reasonable amount of time, also modifying all subdomains on each iteration using a swipe.

Multiple parameters were implemented in such a way that they can be customized by the user to optimize the load balancing of the simulation according to their scenario's needs.

The performance of the schemes is very promising, but depend on the amount of initial imbalance of the test scenario and initial subdomain configuration.

8.2. Future Work

In future work, the load balancing schemes that I implemented could be extended to support 2D or 3D domain decompositions. Since I implemented the load balancing module in a modular way, this can be done without much difficulty.

The value used for work, which is currently the number of atoms in the current processing unit, could be changed. Another potential value to represent the workload of a processing unit could be the time spent performing the previous simulation iteration.

More schemas can also be added, in addition to the inverted pressure and the tensor methods, to possibly improve the new subdomain edges during a swipe iteration.

While the dynamic load balancing worked in a virtual machine running OctoPOS, and this was tested on multiple different computers, the dynamic load balancing did not work on the real machine running OctoPOS due to a bug. The bug is probably due to differences in memory allocations between the virtual machine and the real machine.

Part IV.

Appendix

List of Figures

1.1. Water droplet hitting a copper block. 4

2.1. Lennard-Jones Potential with respect to the intermolecular distance. 6
2.2. The simulation domain decomposed in cells. The light gray area represents the cells in which we look for neighbors for the molecule in the center of the circle. The dark gray area represents the actual neighbors in that molecule's neighborhood list. 7

3.1. Initial state of an EVENLY scenario. 10
3.2. Initial state of a RBCCOLLISION scenario. 11
3.3. Initial state of an OBCCOLLISION scenario. 11

4.1. Example of how the domain is divided for dynamic load balancing. 16

5.1. The problem that the two load balancing schemes aim to solve: finding an appropriate position for the common edge. 30

6.1. Performance overview at the end of a simulation. 35
6.2. Performance overview at the end of a simulation with load balancing timed. 36

7.1. Time comparisons by load balancing schema for the scenario with 9270 atoms. 40
7.2. Time comparisons by load balancing schema for the scenario with 4361 atoms. 41

List of Tables

7.1. Time spent on each simulation part by the different load balancing schemes for the larger test scenario. 39

7.2. Time spent on each simulation part by the different load balancing schemes for the smaller test scenario. 39

Bibliography

[APT22] R. Berger D. S. Bolintineanu W. M. Brown P. S. Crozier P. J. in 't Veld A. Kohlmeyer S. G. Moore T. D. Nguyen R. Shan M. J. Stevens J. Tranchida C. Trott S. J. Plimpton A. P. Thompson, H. M. Aktulga. Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales, 2022.

[CTN+09] Paul Crozier, Heidi Thornquist, Robert Numrich, Alan Williams, H. Edwards, Eric Keiter, Mahesh Rajan, James Willenbring, Douglas Doerfler, and Michael Heroux. Improving performance via mini-applications. 01 2009.

[dBAdW99] LRZ: Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. High performance computing, 1999.

[GZK07] Michael Griebel, Gerhard Zumbusch, and Stephan Knapek. *Numerical Simulation in Molecular Dynamics Numerics, Algorithms, Parallelization, Applications: Numerics, Algorithms, Parallelization, Applications.* 01 2007.

[HJB17] M. Bader I. Comprés A. Hollmann A. Mo-Hellenbrand M. Schreiber J. Weidendorfer H.-J. Bungartz, M. Gerndt. D3: Invasion for high-performance computing, 2017.

[HSS99] R. Halver, S. Schulz, and G. Sutmann. All - a loadbalancing library, 1999.

[Koe21] J. Koerner. Enabling massive parallelism for the autopas demonstrator md-flexible using adaptive domain decomposition master thesis. Master's thesis, Technische Universität München, 2021.

[Lab12] Argonne National Laboratory. Mpich, 2012.

[Lab16] Sandia National Laboratories. minimd, 2016.

[OSK+11] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Octopos: A parallel operating system for invasive computing. 04 2011.

[THH+11] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. *Invasive Computing: An Overview*, pages 241–268. Springer New York, New York, NY, 2011.

[Wik22] Wikipedia contributors. Lennard-jones potential — Wikipedia, the free encyclopedia, 2022.

[WRHDF20] Xipeng Wang, Simón Ramírez-Hinestrosa, Jure Dobnikar, and Daan Frenkel.

The lennard-jones potential: when (not) to use it. *Phys. Chem. Chem. Phys.*, 22:10624–10633, 2020.

[YXL+21] Yinhao Yu, Xiongwen Xu, Jinping Liu, Yuehui Liu, Wenhao Cai, and Jianxun Chen. The study of water wettability on solid surfaces by molecular dynamics simulation. *Surface Science*, 714:121916, 2021.

[Zha22] H. Zhang. Implementing a molecular dynamics simulation for the invasive run-time support system. Master's thesis, Technische Universität München, 2022.