Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 323

A Study of Bottlenecks of parallel Simulations with ESPResSo

Tilak Waelde

Course of Study:	Softwaretechnik
Examiner:	Jun.Prof. Dr. rer. nat. Dirk Pflueger
Supervisor:	Steffen Hirschmann, M.Sc. Dipl.-Phys. Florian Weik
Commenced:	18. April 2016
Completed:	18. October 2016
CR-Classification:	I.6.3

Abstract

Particle based simulations are a very important part in the domain of Computational Science and Engineering and serve as the fundamentals of a diverse amount of simulations. The Extensible Simulation Package for Research on Soft Matter is a state of the art project in the field of Molecular Dynamics software. The project has existed for quite some time now and has over the time been subject to a vast amount of changes and additions to its code base. The purpose of this work is to measure the performance of some of the commonly used simulation scenarios and to identify bottlenecks in the projects code base. Furthermore during the course of this work a new Feature for ESPResSo is developed which enables ESPResSo to natively measure the performance of code sections.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The Extensible Simulation Package for Research on Soft matter (ESPResSo) is a program used for Molecular Dynamics (MD) simulations in the Field of High Performance Computing (HPC). It can be used for a lot for different simulation scenarios, from atoms, over agglomeration (simulations of e.g. soot particles) to simulations of DNA. The project originally started out as a C program and is today written in C++ with interfaces in Python and the Tool Command Language (TCL). The purpose of this work is to examine ESPResSo, take a closer look into the programs performance on four different simulation scenarios and to search for bottlenecks in the code base. In the course of this work there is also an new Feature developed for ESPResSo, which enables users to natively execute performance tests within ESPResSo.

This paper is structured as follows:

- Chapter 2 introduces Performance Analysis in HPC, then introduces some of the tools which are commonly used for this task and discusses their usefulness regarding this work.
- Chapter 3 introduces the four scenarios examined in this work, introduces ESPResSo and explains how simulations with ESPResSo work in detail.
- Chapter 4 introduces and explains the new ESPResSo feature: Timers
- Chapters 5 - 8 take a closer look into some of the more common simulation scenarios used with ESPResSo and present some data on ESPResSo's performamce
- Chapter 9 presents the bottlenecks found during this work
- Chapter 10 summarizes all that was done and gives a small outlook on future work

2 Performance Analysis

This chapter covers Performance Analysis. In the beginning the importance of Software Engineering in the High Performance Computing is adressed, then some performance profiling software is examined and in the end of this chapter the way ESPResSo is profiled in the course of this work is presented.

2.1 Software Engineering in High Performance Computing

Software Engineering (SE) has an interesting role in the field of High Performance Computing (HPC)[BCC+08]: A lot of the developers aren't trained software engineers, but scientists, which develop applications because they need them in order to conduct their research. These programs often start out as small set of scripts and then over time grow into projects with huge code bases. Often during this process of growth code from the very beginning of the project stays in the code base instead of being reworked at some point of time and creates projects, which would have probably gained much performance from one or another redesign process of their code[1]. On the other hand, such projects are written to be in the first place as performant as possible and when it comes to the way they are developed a lot of the rules and practices of SE, which apply to the development of other applications will contradict with the way that problems are solved in HPC[FDK+11]. As a result applications in HPC often aim to provide their users and developers with a combination of reasonable performance as well as a proper code base providing important SE Features like scalablilty.

2.2 Performance Analysis Tools

This sections covers some of the available performance analysis tools and shortly presents the way they work:

[1] The ESPResSo project for example started out as a project written in C and has evolved into a project currently in C++, with additional code in TCL and Python.

1. **Valgrind**

   Valgrind[NS03; NS07] is a framework intended to create profiling and analysis tools. It currently consists of six different tools. When analyzing a programm with Valgrind, the executable is run on a simulated CPU; this way Valgrind is able to obtain far more detailed Information the execution of an applications than other profilers. However, since Valgrind does not work directly on the hardwares CPU it is very slow compared to its alternatives. Valgrinds Callgrind tool generates an accurate callgraph containing information on the functions which were used during a programs execution and the amount of runtime the execution has taken up.

2. **Dprof**

   Dprof[PZM10] is a data profiler, which has a very unique and different approach to profiling than other tools; it aims to present a relation between cache performance of applications and the data types used in their source code. Sadly the profiler could not be installed.

3. **Gprof**

   Gprof[GKM82] is a tool which inserts code into all functions of the application which is being analyzed and then measures the execution time of these functions. To use gprof the program has to compiled with a special CFLAGS options. This profiler for some reason stopped working with the ESPResSo source code some time ago and could thus not be used in this work.

4. **Oprofile**

   Oprofile[LE04] is a sampling profiler, which uses hardware performance counters to gather its data on running applications. This offers several advantages[Era08] over tools like Gprof, e.g. the source code of the application does not have to be compiled with special options in order to use it.

5. **Google benchmark**

   Google benchmark is a small library aimed at profiling source code by injecting additional code into the functions which are about to be profiled. It can be applied to profile code in several different ways, however it works by running sections of the code in a loop, measuring their average time of execution. This concept seemed to be unfitting for use with ESPResSo, since most of what is examied in this work already runs inside of several loops.

6. **Perf**

   Perf or the 'Performance Counters for Linux' are another example for a sampling profiler like Oprofile. It can be used to create a profile of running applications utilizing for example hardware as well as software performance counters. It does not require the profiled applications to be executed with special compile options and is also well integrated into the linux operating system.

7. **LibSciBench**

   LibScibench[HB15] is a library intended for performance profiling of HPC applications. It has a very impressive set of Features, which are aimed to make sure of high quality and statistically correct results. However the project did at the time unfortunately not offer much documentation, which is why it was not selected to be used in this work.

2.3 Measuring performance in ESPResSo

After examining some of the available profilers as well as their feature sets the following way to approach ESPResSo was decided: As a first step and without the use of profilers, the source code is examined to get an detailed overview of what happens internally, when ESPResSo is executing simulations. Then Simulations are run by Valgrind as well as a sampling profiler and in a last step the actual runtime of functions during different simulation scenarios is measured.

Valgrind is chosen as profiling tool, because of all tools, which were examined it is the only one which gathers exact its during the execution by running the code on a simulated CPU. Valgrind does not suffer from the same errors as sampling profilers do, it has however the great disadvantage of being much slower due to the fact that it is simulating hardware.

As a second tool Perf is used: as a statistical and sampling profiler it regularly looks at the contents of the stack and from there on derives what happened during the execution of a program. Perf has the disadvantage of being somewhat prone to a certain amount of inaccuracy; sometimes sampling profilers like perf are kown to present the wrong funcions as execution hotspots. However the great advantage of a tool like perf is that the execution time of the application is only minimally affected.

Lastly it was planned to use Gprof or LibSciBench in order to get a profile which delivers actual execution times of functions instead of percental data relating to the global amount of execution time measured. However none of these tools could be used with ESPResSo and after some consideration it was decided to create the Timers Feature

which, natively brings such a functionality to ESPResSo and has potentially beneficial applications in the projects future [2]. The Feature is introduced and examined in Chapter 4.

[2] e.g. automated performance monitoring of different ESPResSo Versions or an easy way for users to compare the performance of different algorithms in their research

3 Analyzing ESPResSo

This chapter explains how the ESPResSo works. It introduces the program and some of its features which are used in this work, explains how the project is being developed and then explains how Molecular Dynamics simulations in general and ESPResSo in detail work.

3.1 ESPResSo

The Extensible Simulation Package for Research on Soft matter (ESPResSo) is an actively developed project which aims to provide its users with a wide variety of MD Simulation methods as well as an easy to use interface. In the following sections the way ESPResSo works internally will be described to some extent in order to provide all necessary information on what is examined in the following chapters. First ESPResSos User Interfaces as well as the features used in the scenarios in Chapters 5 - 8 are introduced, then ESPResSos Development Process is discussed and the rest of the chapter explains some Implementation details of ESPResSo.

3.1.1 User Interfaces - `TCL/Python`

ESPResSo has currently two different user interfaces, the old `TCL` interface, as well as the newer and not yet fully functional `Python` interface which will be introduced with ESPresSo's 4.0 Version along with some other important changes. Both interfaces, when started present the user with an interactive shell where simulations can be setup step by step or if provided with a file run it as simulation script. Special `TCL` respectively `Python` commands give the user control over the simulation. Apart from that all features provided by the interfaces language are available as well. In order to set up a simulation has to be defined in the script language and then communicated through the user interface to the ESPResSo core. This happenswith the help of special commands [1]. During

[1] for example in TCL the value of internal variable can be defined using the `setmd` TCL command

a simulation the ESPResSo core is controlled by the Script Interface and simulation data can be retrieved on the script level in between integration steps.

In this work the `Python` Interface is not used due to the fact that it has not yet been fully completed. Instead the `TCL` Interface is used to run all simulations. The Tool Command Language has the disadvantage that it is unfortunately not able to export into specific file formats like python is.Because of that the scripts used in this simulation rely on regular expressions in order to create the datasets in an ealsily parsable way. The script output is exported into JSON files which are then used to analyze the simulation data. The Python Interface would have been able to export data natively into other formats.

3.1.2 Features used in this work

In this work four scenarios have been chosen to be examined in order to investigate the performance of ESPResSo's different capabilities. These scenarios are:

Non-bonded Interaction

The interactions of non-bonded particles are very simple to calculate in a simulation. Particles are given new positions and velocities depending on their surrounding particles. ESPResSo supports various types of these non-bonded and short ranged interactions as well as different options to keep track of the particles internally. For this scenario the Lennard-Jones interaction was chosen. It will be discussed in chapter 5.

Bonded Interaction

Bonded short ranged interactions are in Molecular Dynamics simulations calculated in a different way than non-bonded interactions. In chapter 6 the FENE potential is examined.

Electrostatics - P3M

A scenario on Electrostatics serves as an example for long range interactions in ESPResSo. In chapter 7 the fourier based `Particle-Particle-Particle-Mesh` [Cou91] (P3M) method is introduced.

Agglomeration - Dynamic Bonds

Finally the Agglomeration scenario in chapter 8 presents ESPResSos approach on Dynamic Bonding simulations.

3.1.3 Development Process

ESPResSo is an open source softwareproject currently developed on github. The development of the project is very active and its development process has similiarities with the DevOps model [BWZ15]; since the project makes use of various Continous Integration services to provide its community with quality controlled and stable as well as bleeding edge releases of the software.

An iterative process is utilized to organize workloads and regularly audit new parts of ESPResSo's codebase. Continous Integration systems (TravisCI and CircleCI) provide automated testing for pull requests on github and another system (Jenkins) is used for additional testing and to create ESPResSo's release files.

Though ESPResSo is developed on and for Linux the project builds fine on all mature Linux distributions[2]. The project is not one of those open source projects which works only on specific Linux distributions (e.g. Ubuntu).

It is also worth mentioning that the development process is open to making use of new technologies, as long as they add a benefit to ESPResSo: In the upcoming 4.0 release the `TCL` interface will be dropped in favor of a new one `Python` and the build process which currently relies on `autotools` will be replaced to the much more userfriendly `cmake`.

3.2 Molecular Dynamics Simulations

Simulations in the field of Molecular Dynamics (MD) can be defined as "a technique for computing the equilibrium and transport properties of a classical many-body system. In this context the word classical means that the nuclear motion of the constituent particle obeys the laws of classical mechanics."[Smi01]: For a set of particles within a defined space (the simulation box) the interactions between these particles are calculated iteratively over a specific time. Each Particle in the simulation has specific properties, from which the way it interacts with the rest of the System is deduced. The interactions

[2] Due to the way the `vtk` package is being packaged on some distributions (e.g. archlinux) certain features of ESPResSo may not work.

between particles can be either local (short ranged), such that the particle only interacts with particles which are positioned within a specific distance or they can be global (long ranged) and thus affecting all particles in the system. Every iteration (integration step) of the simulation represents the passing of a time interval and calculates the new positions as well as the forces of every particle in the system.

3.2.1 General Approach to MD Simulations

Typically programs performing MD Simulations work in a similiar way:

1. `Setting up the Simulation`

   At the beginning the size of the simulation box is defined, a number of particles is added into the box by assigning positions as well as velocities to every particle. If needed additional constraints to the simulation can be defined. Usually also the simulation time or the amount of integration steps is defined, as well as the various interactions used in the simulation. Since the particles added to the system may initially be affected by unrealistic forces or impossible constraints this step results possibly results in an (highly) unstable system.

2. `Balancing the System (Warmup Phase)`

   Before the simulation can be started the system has to be brought to a state, where all predefined constraints are properly met and the particles behave within normal[3] parameters; this equilibration is usually reached by integrating the system until the properties of the system have reached an acceptable state.

3. `Running the actual Simulation for a number of timesteps (Integration)`

   At this point the simulation is all set and the integration can be started. In each step of the integration the new velocities and positions for every particle in the system are calculated:

   a) Force Calculation:

      Here all forces interacting with the particles in the simulation are calculated; for every particle in the system all forces affecting it have to be determined in this step.

[3] in this case "normal" refers to the ideal state of the system in which the simulation is intended to take place

b) Integration:

After having calculated the forces for all particles the system is integrated: the forces are applied to the particles resulting in new positions and velocities for every particle in the system

In between integration steps the state of the system can be put out or saved for later analysis

The integration of steps 2. and 3. basically work the same way, although the warmup integration doesn't necessarily need to calculate the particles positions and velocities according to the same rules as the actual integration; as long as the outcome of the Warmup Phase is a system in an acceptable state.

3.3 Implementation Details

In this section the way certain features in ESPResSo's core are implemented will be explained. Basically ESPResSo works similiar[4] to what was described in 3.2.1., however since ESPResSo is a very complex piece of software in order to understand the way it works some of its code is examined in this section.

3.3.1 Build Process

Currently there are two ways to build ESPResSo: using the older `autotools` and a provided build script or the newer and more user friendly `cmake` together with the optional interactive `ccmake` utility. In this work ESPResSo is build using `ccmake`, since it seemes to be the more convenient tool to work with. After downloading ESPResSo's source code, a build directory is created and then inside the build directory `ccmake ..` is run to configure the build. Before actually building ESPResSo it has to be configured by creating the `myconfig.hpp` file containing the appropriate preprocessor directives for the planned simulation. That way during the compilation only the features necessary are added to the executable.

Listing 3.1 The `Particle` struct is used by ESPResSo to hold the all necessary information on particles however it does contain information (like the bonded interaction list) which may not be needed in every simulation. The `ParticleList` struct is used by ESPResSo so the core process can keep track of all its particles in an efficient way.

```
/** Struct holding all information for one particle. */
typedef struct {
  ParticleProperties p;
  ParticlePosition r;
  ParticleMomentum m;
  ParticleForce f;
  ParticleLocal l;
  /** bonded interactions list. The format is pretty simple:
     Just the bond type, and then the particle ids. The number of particle ids can be
         determined
     easily from the bonded_ia_params entry for the type. */
  IntList bl;
} Particle;

/** List of particles. The particle array is resized using a sophisticated
   (we hope) algorithm to avoid unnecessary resizes.
   Access using \ref realloc_particlelist, \ref got_particle,...
*/
typedef struct {
  /** The particles payload */
  Particle *part;
  /** Number of particles contained */
  int n;
  /** Number of particles that fit in until a resize is needed */
  int max;
} ParticleList;
```

3.3.2 Particles

Particles are the most important ingredient to any Molecular Dynamics simulation and as such one of the most important data structures used in ESPResSo. The minimal set of properties for a particle in a Simulation are its coordinates inside the simulation box, the particles force and its Momentum in order to be able to determine the particles position in a coming integration step. In ESPResSo particles are stored as `structs` Listing 3.1 containing besides these already mentioned properties at the very least also the informa-

[4] other than the example in 3.2.1., ESPResSo uses the velocity verlet scheme[SABW82] in order to calculate integration steps

tion, whether the particle is local or not[5], Information on the particles properties as well as a bonded interaction list. Depending on whether additional features are enabled or not the particle struct can also contain more information.

Since every ESPResSo core process needs to keep track of its particles, there is another very important struct, the `ParticleList`: it contains the number of particles in the list, the number of particles which can currently be held in the list as well as a pointer to an array of particles — the actual particle list.

Since ESPResSo is using these particle structs to calculate the forces in its integration loop it is safe to conclude the following: From a HPC perspective the use of such structs for the calulation can be a bad choice, since with additional unused information on particles (e.g. an empty bonded interaction list in case of a simulation containing only non-bonded particles) during a simulation it is not possible to achieve optimal memory locality during the execution of a simulation. It would be far more performant to use data structures containing only the necessary information. Also it is well known that as an Array of Structs the Particle lists in ESPResSo are not necessarily the most performant data structure to use, see e.g. [Str11] and C++ as ESPResSo's implementation language supports zero overhead abstraction.

3.3.3 Short Range Interactions

Short ranged interactions are noteworthy at this point since ESPResSo's integration loop, the force calculation, separates between bonded and non-bonded interactions: As it is customary in Molecular Dynamics, in ESPResSo non-bonded interactions between particles are typically calculated pairwise, while the bonded interactions are calculated on a per particle level.

Bonded Interactions

Bonded interactions are calculated in ESPResSo by iterating over all particles and then calling the `add_bonded_force` function Listing 3.3 on every particle in the system [6]. When `add_bonded_force` is called on a particle it loops through all particles which are referenced in its bonded interaction list. `add_single_particle_force` is called once per particle per integration step.

[5] which is important for the way ESPResSo adresses parallelization

[6] The `add_bonded_force` function is directly called by the `add_single_particle_force` Listing 3.2 function

Listing 3.2 Part of ESPResSo's core loop where the bonded interacion is calculated by calling `add_single_particle_force`. From there on the `add_bonded_force` function is executed, which does the actual calculation of the bonded forces.

```
 for (c = 0; c < local_cells.n; c++) {
  cell = local_cells.cell[c];
  p1  = cell->part;
  np = cell->n;
  /* calculate bonded interactions (loop local particles) */
  for(i = 0; i < np; i++) {
   {
    add_single_particle_force(&p1[i]);
   }
  }
```

Listing 3.3 The begin of `add_bonded_force`. When the function is called on a particle, it loops through this particles bonded interactions list and then starts its calculation.

```
inline void add_bonded_force(Particle *p1)
{
 double dx[3] = { 0., 0., 0. };
 double force[3] = { 0., 0., 0. };
 double force2[3] = { 0., 0., 0. };
 double force3[3] = { 0., 0., 0. };
[...]
 Particle *p2 = NULL, *p3 = NULL, *p4 = NULL;
 Bonded_ia_parameters *iaparams;
 int i, j, type_num, type, n_partners, bond_broken;

 i = 0;
 while (i<p1->bl.n) {
  type_num = p1->bl.e[i++];
  iaparams = &bonded_ia_params[type_num];
  type = iaparams->type;
  n_partners = iaparams->num;

  /* fetch particle 2, which is always needed */
  p2 = local_particles[p1->bl.e[i++]];
  }
     [...]
}
```

Listing 3.4 Non-bonded forces are calculated by iterating once over the list containing all pairs and calling the `add_non_bonded_pair_force` function.

```
   for (n = 0; n < dd.cell_inter[c].n_neighbors; n++) {
    pairs = dd.cell_inter[c].nList[n].vList.pair;
    np  = dd.cell_inter[c].nList[n].vList.n;
    /* verlet list loop */
    for(i=0; i<2*np; i+=2) {
     p1 = pairs[i];           /* pointer to particle 1 */
     p2 = pairs[i+1];         /* pointer to particle 2 */
[...]
     {
      dist2 = distance2vec(p1->r.p, p2->r.p, vec21);
      add_non_bonded_pair_force(p1, p2, vec21, sqrt(dist2), dist2);
     }
    }
   }
```

Non-bonded Interactions

In ESPResSo non-bonded interactions are calculated pairwise; the naive approach to do so, would be to loop over all particles and then again loop over all other particles in the System to calculate a particles forces. However this is where ESPResSo's verlet lists feature (respectively ESPResSo's linked cells feature[7]) comes in. ESPResSo manages its particles through verlet lists, arrays which contain the particle pairs interacting with each other. To calculated all forces in every integration step ESPResSo iterates once over the pair lists and executes `add_non_bonded_pair_force`, which then executes — depending on ESPResSo's confiuration (at build time using myconfig.hpp file) — other functions, which calculate the pairs forces.

Integration

The integration is ESPResSo's core loop and when broken down to its essential parts reads straightforward: every call of the `integration` command on the script level results in a integration of `n_steps` actual integration steps by ESPResSo's core. For the integration the Velocity Verlet Scheme [SABW82] is used in order to caculate the systems new state:

First `propagate_vel_pos()` calculates partial velocities (`v(t+0.5*dt)`) for all particles in the system depending on the particles current forces `v(t) + 0.5*dt * f(t)`, then it

[7] At this point it suffices to show an example from the code of the verlet calculation.

Listing 3.5 The essential parts of ESPResSo's integration loop: using the Velocity Verlet Scheme the system is integrated.

```
void integrate_vv(int n_steps, int reuse_forces)
{
[...]

 /* Integration loop */
 for (int step=0; step<n_steps; step++) {

  /* Integration Steps: Step 1 and 2 of Velocity Verlet scheme:
    v(t+0.5*dt) = v(t) + 0.5*dt * f(t)
    p(t + dt) = p(t) + dt * v(t+0.5*dt)
    [...]
  */
   propagate_vel_pos();
  }

  /* Integration Step: Step 3 of Velocity Verlet scheme:
    Calculate f(t+dt) as function of positions p(t+dt) ( and velocities v(t+0.5*dt) ) */
  force_calc();

  /* Integration Step: Step 4 of Velocity Verlet scheme:
    v(t+dt) = v(t+0.5*dt) + 0.5*dt * f(t+dt) */
  if(integ_switch != INTEG_METHOD_STEEPEST_DESCENT) {
   rescale_forces_propagate_vel();

    [...]
 }
}
```

updates the particles positions using the newly calculated velocities as a base. In a next step the new particle forces are calculated by `force_calc()`; this is also the most time consuming part of the integration step, since `force_calc()` executes beside the actual force calculation also EPSResSo's functions concerning the particle management. Finally `rescale_forces_propagate_vel()` calculates the velocities for all particles once again using the updated forces from `force_calc()`.

Parallelization

ESPResSo uses MPI [GLDS96] for its parallelization. When executed using `mpirun -np N ./Espresso ...`, `N-1` slave ESPResSo core processes are started, which are then controlled by a master ESPResSo core process, the master ESPResSo core is controlled through the user interface where the simulation is started.

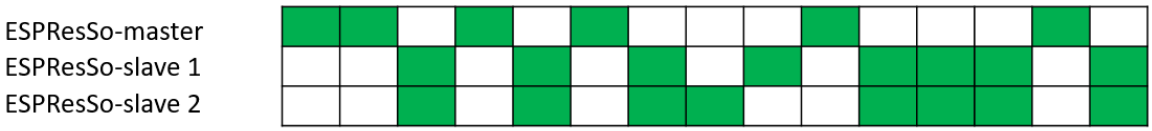


Figure 3.1: A schematic of the ESPResSo core processes master slave architecture: The script interface controls the simulation and starts integration steps which are calculated on the ESPResSo core process. When simulations are run using more than one core process, the core process started directly from the user interface controls the others as their master process. The slave processes get a part of the simulation box assigned and calculate the interactions for this partial system. Particles interacting in between multiple processes are communicated using ESPResSo's so called ghost communication. At the end of an integration step the slave processes communicate back to the master process so that the ESPResSo user interface can accesss the newly calculated information in between integration steps.

Domain Decomposition

At the beginning of every simulation the simulation box is divided into various smaller boxes, so called domains; these are then assigned to the ESPResSo core processes involved in the simulation. This process is called domain decomposition. The domains do not equal the cells[8], which are used to simplify the calculation of particle interactions, but every domain can contain multiple cells. The domain decomposition process is simply used to enable parallelisation of ESPResSo and is part of the so called ghost communication.

Apart from dividing the simulation box into domains another important part of the domain decomposition process is the communication of particles between cells which are not caculated from the same ESPResSo core process; this is done by the `dd_exchange_and_sort_particles` function. To save time during the simulation in ESPResSo this function is executed at the same time as the `build_verlet_lists_and_calc_verlet_ia` function of the verlet lists feature (see Verlet Lists).

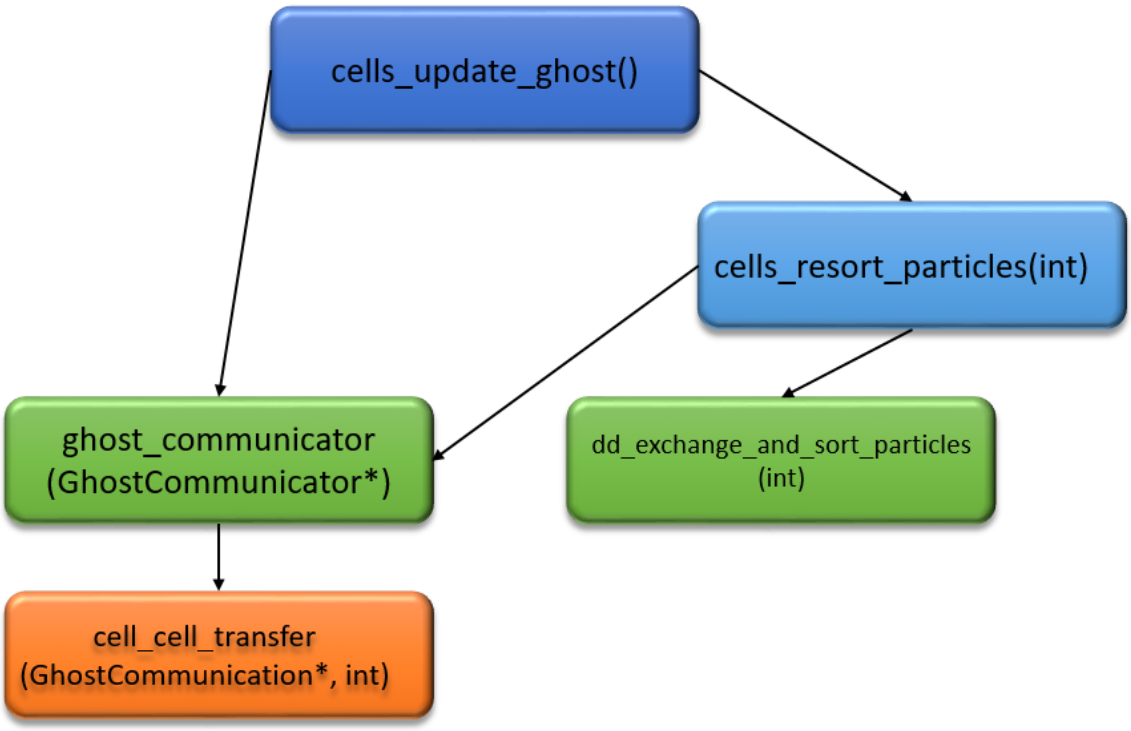


Figure 3.2: ESPResso's ghost communication is executed during the force calculation of the integration: the `cells_resort_particles` function checks whether a particles position has changed and it has to be moved into a ghost cell or a neighbouring cell; then depending on whether the particle is moved within the same domain or across domains it is communicated either locally using the local `cell_cell_transfer` method or to another ESPResSo core process using the `dd_exchange_and_sort_particles` function.

Ghost Communication

Every ESPResSo Core process manages a set of particles, the particles inside its domain and has no knowledge about other particles in the simulation system. The ghost communication is the process of managing all particles such, that they are always available to all cells, which need to know about them during a calculation.

Usually the integration is done cell by cell. Whenever a particles position has changed such, that the particle interacts with other particles in neighbouring cells the information has to be available on the other cell. If the particle remains in the same domain the problem is easily solved by just moving the particle to its new destination, since the

[8] see Linked Cells or Verlet Lists

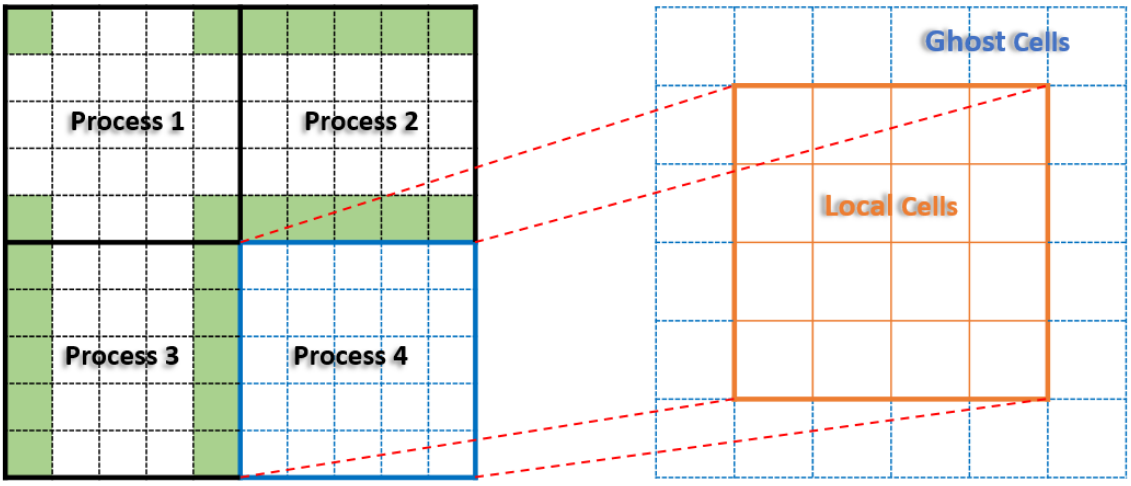
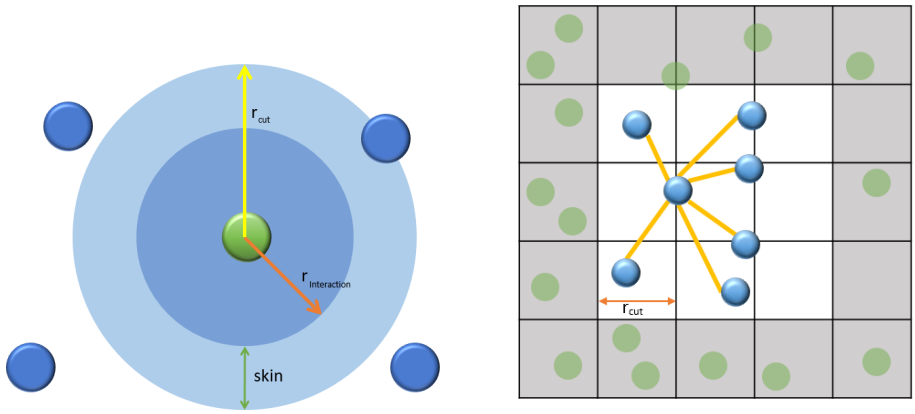


Figure 3.3: ESPResSo handles particle management by introducing the ghost layer: Cells on the border of domains are copied over to their neighbouring domains into ghost cells. That way every ESPResSo core process has at all times enough information to calculate an integration step without having to know about all particles in the system.

ESPResSo core process knows all about both cells. However when the particle interacts with particles in another domain one ESPResSo core process has to send over the particle, respectively a copy of the particle [9] to another core process. The way ESPResSo solves this issue is that it introduces another layer of particles, the ghost layer: This Layer consists of additional cells (ghost cells) which contain partial copies of all the particles which interact with the cells of the core processes. During the ghost communication (see Figure 3.2) particles are whenever necessary put into these ghost layers or retrieved into local cells, so that all information necessary for the integration is always available on every ESPResSo core process without having to copy around whole cells or domains.

During the force calculation the `cells_resort_particles` function checks whether particles have to be moved to other cells or interact with particles within other cells and if so these are communicated by either the `cell_cell_transfer` function (in case of local cells) or they have to be communicated using the `dd_exchange_and_sort_particles` function to another domain, which is executed simulataniously by all ESPResSo core processes using special MPI commands.

[9] if it only interacts with particles in the other domain but remains in its cell



(a) VerletLists **(b)** LinkedCells

Figure 3.4: In ESPResSo the cutoff radius is determined by adding a skin to the range of the shortrange interaction with the greatest interaction distance of the simulation (a)) This radius is then used to determine the cell sizes in the simulation box. This way a particle can interact only with its own or its neighbouring cells and ESPResSo has to provide fewer cells with information on particles. The skin is used to set the cell size and has no impact on the linked cells feature (not adding the skin to the cell size would result in particles interacting with the neighbours of their neighbourcells during the integration, which would be an unneccesary overhead). The skin is however used in the verlet pair lists Feature in order to minimize the amount of verlet list rebuilds.

Linked Cells

Internally the simulation box is split into smaller cells, and each of the ESPResSo core processes gets to control a number of cells. The size of these cells is determined by the interactions of the simulation Figure 3.4: the cellsize is determined by the so called cutoff radius, which is the range[10] of the greatest short range interaction in the simulation plus a small additional distance, the so called skin. Selecting the cellsize this way is useful since a particle can only interact with particles of its own cell and the cells direct neighbours. This has a huge impact on the time every integration takes. Integrating

[10] Of course the so called "short range" interactions do not just vanish after a specific range, but when simulating short range interactions, there is a distance from where the interaction only influences the system in a neglectible way. Thus in computer simulations these interactions are indeed only short ranged.

by using all particles in the own cell as well as all particles from the neighbours cells is called the `Linked Cells` Feature.

Verlet Lists

Other than using the `Linked Cells` during the force calculation ESPResSo can do even better by using so called verlet lists: For every single particle the amount of particles for which the pair forces are calculated is reduced even more by creating pair lists only including particles which are within the cutoff radius (which equals the interaction range plus the additional skin). These particles can be found by checking all particles in the particles own cell as well as the particles from its neighbor cells. Once created the verlet pair lists remain valid and can be reused in the following integration steps until a single particle moves out of or into another particles cutoff radius. Using verlet lists is far more effective as long as they are reused frequently enough.

3.3.4 Random Number Generators

ESPResSo makes use of Random Number Generators e.g. to initialize particle forces. Currently the Mersenne Twister Engine is used. However since there already exist studies on the performance of RNGs [Dem11] the impact of these will not be discussed in this work. It is obvious that the use of a faster Random Number Generator results in faster simulations. During this work the topic of the impact of Random Number Generators has been noticed and it is planned to add a Feature to ESPResSo which enables the user to choose between multiple different Generators.

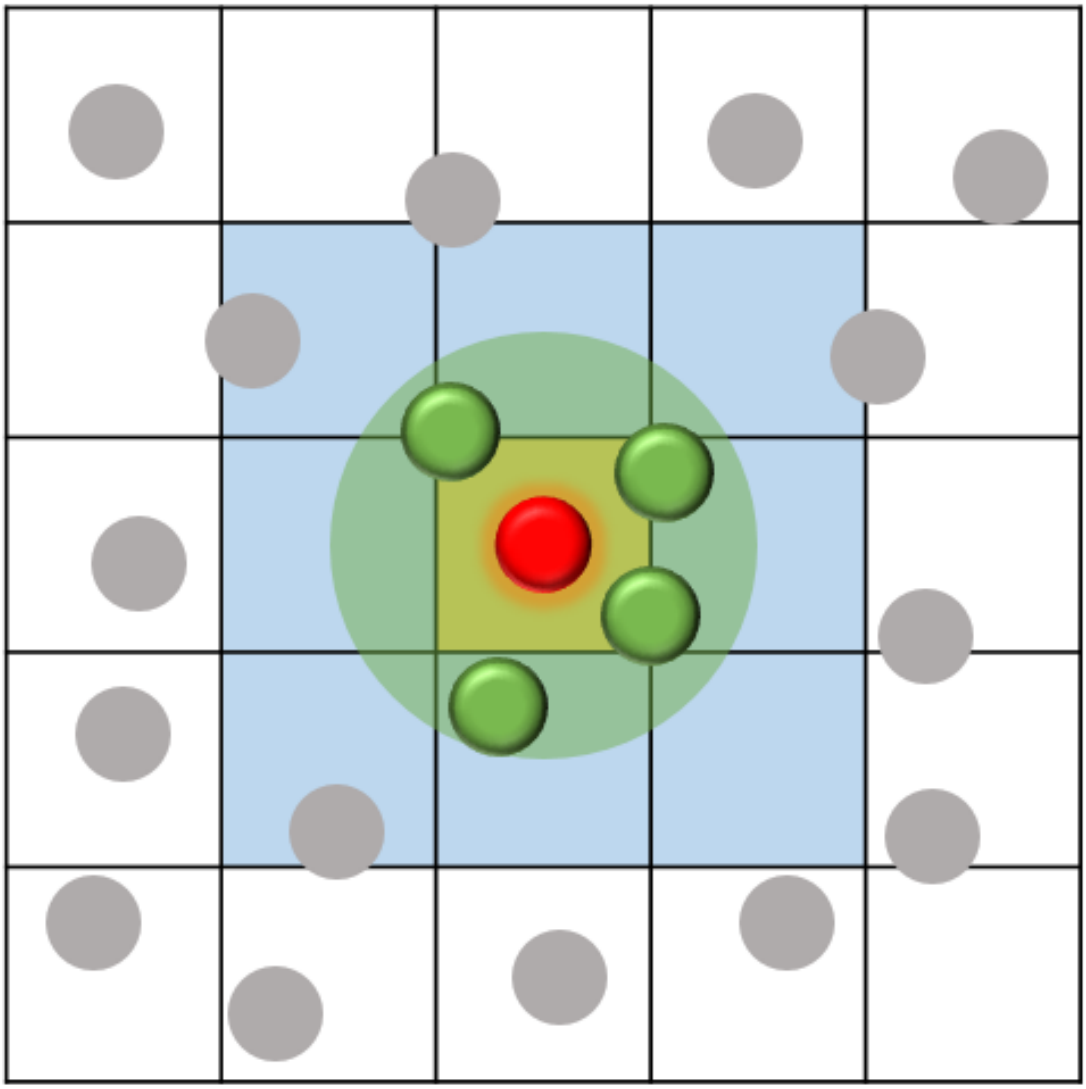


Figure 3.5: The Linked Cells Feature and the Verlet Lists Feature of ESPResSo: Instead of having to loop over all Particles in a particles neighbouring cells every time a non-bonded force is calculated ESPresSo can simply go over all the interaction pairs in a cells verlet pairs lists; these pair lists contain for every particle in the cell all interaction pairs which are in these particles cutoff radii. In this example the red particle only has to interact with particles, which are positioned within its cutoff radius (dark green). When using the linked cells feature, the particles would have to interact with all particles in its own cell (bright green area) as well as with all particles in the neighbouring cells(blue area), which is far less efficient.

4 ESPResSo Timers

This chapter covers the Timer Feature, an ESPResSo extension which has been developed in the course of this work and which is intended to be used in the future development process in order to help to test the code for performance. The Feature is introduced and then how it works is explained in detail.

4.1 The Insufficiency of using Profilers

In chapter 2 a few profiling tools are introduced and discussed; then from these tools Valgrind and the Linux performance Counters are choosen to examine ESPResSo (see chapters 5 through 8). However since one of the tasks in this work is to search ESPResSo's codebase for bottlenecks even when using profiling software there is still the need to be able to obtain significant and comparable data. Due to the way profiling tools work this is however not possible:

4.1.1 Perf

The Linux performance counters are using the sampling profiling method to obtain their data: This means, that during the execution of the program, which is to be profiled the state of the stack is saved iteratively creating a dataset from which then the structure of the program is deduced and presented in a humanly readable way. By doing so perf does essentially guess how the profiled program works. Moreover due to the way of obtaining the samples at essentially random intervalls (from the perspective of the executed code) it often occurs that the data obtained contain a certain degree of freedom when it comes to the validity of the samples: in case the samples are taken in such a way that a certain function is executed accidentially every time a sample is taken, the output of a sampling profiler will indicate this function as more time consuming than it is in reality. A way to reduce this inaccuracy is of course to increase the amount of samples taken during execution, resulting in an increase of runtime as well as an increase of data to analyze. Another disadvantage is that two datasets can't be compared properly (e.g. after a

change of a function in the executable) since they are both sample based instead of time based.

4.1.2 Valgrind

When creating a profile Valgrind is using a different approach, then sampling profilers like perf; instead of running the program and checking the stack on a regular basis, valgrind actually emulates the CPU which is running the program and is thus able to obtain more detailed information on how the program is executed. However this results in far longer runtimes than running the code on real CPUs. Also since the CPU is emulated the data measured during the execution of the program can't be necessarily applied to the execution on real hardware; code running smoothly inside of a simulation could run terribly in a real world environment. Also it is hard to compare multiple data sets, since valgrind only provides the percentage of runtime (global or local) a function used in stead of actual comparable runtimes.

4.1.3 The Need for Timers

While using profilers is a very useful and valid method of obtaining data on a programs performance or its bottlenecks, on the other hand profilers are not necessarily sufficient tools when it comes to examining how a programs internals perform. This is especially valid when it comes to a program as versatile as ESPResSo, which is used in a vast variety of way to create simulations using entirely different internal functions, which contain a huge number of different Algorithms.

The addition of a Timer Feature which provides ESPResSo with the ability to measure the time certain sections of code take during execution was decided out of the following reasons:

1. Profilers aren't sufficient

   they might lie or output incomparable data (as explained in the previous sections of this chapter)

2. Simplicity

   Timers measure the runtime of some code sections; they do not represent a tremendous overhead to the execution time, since only a few Integers and Floats are additionally stored during ESPResSo's execution and they can be used

anywhere in ESPResSo's core code base[1] Timers represent a minimal intrusive way of obtaining information.

3. Demand

   ESPResSo is used as the simulation environmant of choice in a lot of projects, which would greatly profit from a way of measuring the performance of specific parts of ESPResSo.

4. Quality Control

   ESPResSo uses Continous Integration systems to run automated tests and create its releases. Adding the Timers Feature to ESPResSo results in having the ability of automatically monitor ESPResSos performance over time.

4.2 Specification

Before going into the details of Implementation a few constraints to the Timer Feature are presented, which had to be considered; Timers had to be:

1. Minimal Intrusive

   The Timer Feature should be designed to enable the use of Timers without creating too much overhead in the process, resulting in near normal runtime of ESPResSo whenever the Feature is enabled.

2. Available throughout the code base of the ESPResSo core

   ESPResSo's source code consists out of three parts: the ESPResSo core as well as both, the `TCL` as well as the `Python` Interfaces. The since performance of either interfaces has nothing to do with the ESPResSo core performance, where the actual simulation happens it is only necessary to be able to measure times for functions within the core. This has almost been accomplished: with the exception of the internals of some Cuda code, the Timers can be used to measure every functions execution time.

3. Usable within the ESPResSo User Interface

   Timers should provide the option to retrieve their values from within the simulation. Adding a Timer command to the `TCL` User interface satisfies this behavior, since ESPResSo steers its simulations using this Interface.

[1] with the exception of the internals of ESPResSo's cuda code.

4. Timers must implement a reset function

   This Feature was actually not included in the specification, however it is essential since the way ESPResSo sets up its simulations always features a warmup phase. If during that phase the Timers are measuring, the results are distorted. This came to our attention during a later point and was then fixed, by adding the "timer reset" command to the tcl Interface, which simply clears all timers.

4.3 Implementation

It was decided to implement timers as follows:

1. every measured function has its own Timer
2. every ESPResSo core process has its own set of Timers.

1. is obvious since if measuring multiple different Timers these must have some sort of distinction, in this case every timer has a unique identifier (a string).

2. is due to the fact that ESPResSo itself does not implement any sort of shared memory between multiple core processes.

As a result the Timer Feature returns one Timer per core process used in the simulation per function tracked by Timers.

4.3.1 Timers

First the Timer class Listing 4.1 is added to ESPResSo so that internal Timer objects can be created and used. Timers in ESPResSo are managed using an unordered map of strings. That way Timers can later be defined in ESPResSo's source code be declaring a new Timer Object which is identified by its name. Every Timer contains several values, which are later updated every time the marked code section is executed. These Values are:

- `n` : the amount of times the code section has been executed
- `t` : the time, which was measured when the code section has been executed the last time
- `avg`: the average time of execution so far
- `sig`: the standard deviation

Listing 4.1 Timer objects are internally stored in an unordered map of strings, identified by the name given to them when they are initialized in the source code. Timers currently support `start()`, which starts the measurement, `stop()`, which updates the Timer using the `add_sample` function from the RunningAverage Class, `stats()`, which retrieves the current values of the Timer as well as `reset()`, which deletes all values from the timer.

```
class Timer {
public:
  struct Stats {
  public:
    Stats() {}
    Stats(double avg, double sig, double var, double min, double max, double t, int n)
       : m_avg(avg), m_sig(sig), m_var(var), m_min(min), m_max(max), m_t(t), m_n(n) {}
    double avg() const { return m_avg; }
    double sig() const { return m_sig; }
    double var() const { return m_var; }
    double min() const { return m_min; }
    double max() const { return m_max; }
    double t() const { return m_t; }
    int n() const { return m_n; }

  private:
    [...]

  void start() { m_mark = MPI_Wtime(); }

  double stop() {
    const double time = MPI_Wtime() - m_mark;
    if (time > 0.0)
      m_running_average.add_sample(time);
    return time;
  }

  Stats stats() const {
    return Stats(m_running_average.avg(), m_running_average.sig(),
                 m_running_average.var(), m_running_average.min(),
                 m_running_average.max(), m_running_average.t(),
                 m_running_average.n());
  }

  void reset() { m_running_average.clear(); }

  static void reset_all() {
    for (auto &t : m_timers) {
      t.second.reset();
    }
  }

[...]

  private:
  static std::unordered_map<std::string, Timer> m_timers;
  Statistics::RunningAverage<double> m_running_average;
  double m_mark;
};
}
}
```

- `var`: the variance
- `min`: the shortest time measured so far
- `max`: the longest time measured so far

After declaring Timers in the ESPResSo source code, whenever a section surrounded by a Timer is executed during the simulation the Timer is either created or has already been used and its values are simply updated. Since ESPResSo has no shared memory this happens in parallel on all the core processes involved in the simulation.

Timers support the following functions:

start()

After declaring a Timer Object with a unique string as identifier, the `start()` function can be used to mark the beginning of the code section, which is to be measured. This function saves the current value of `MPI_Wtime()` for later use in the `stop()` function.

stop()

The `stop()` function uses the previously defined timestamp and calculates the time the code spent since `start()` was executed. Then it executes the `add_sample` function from the `RunningAverage` classListing 4.3, which either initializes the Timer (on the first execution of the code section) or updates the Timers values with the newly obtained sample.

stats()

The `stats()` function returns all the values of a Timer. This function is used by the TCL interface `timer` command to retrieve the measured data from the core processes.

Listing 4.2 ESPResSo is using MPI for its inter process communication, so the newly created Timers have to be integrated into the ghost communication.

```
void mpi_timers_slave(int action, int) {
  switch (action) {
  case 0: {
    map<string, Utils::Timing::Timer::Stats> my_stats =
       Utils::Timing::Timer::get_stats();
    comm_cart.send(0, SOME_TAG, my_stats);
    break;
  }
  case 1:
    Utils::Timing::Timer::reset_all();
    break;
  }
}

void mpi_reset_timers() {
  Utils::Timing::Timer::reset_all();
  mpi_call(mpi_timers_slave, 1, 0);
}

vector<map<string, Utils::Timing::Timer::Stats>> mpi_gather_timers() {
  vector<map<string, Utils::Timing::Timer::Stats>> ret(comm_cart.size());
  ret[0] = Utils::Timing::Timer::get_stats();
  mpi_call(mpi_timers_slave, 0, 0);
  for (int i = 1; i < ret.size(); ++i) {
    comm_cart.recv(i, SOME_TAG, ret[i]);
  }
  return ret;
}
```

reset()

The `reset()` function was added later to the timers so that the Timers can be reset after starting the simulation. This is very useful whenever parts of simulations[2] would falsify the results.

4.4 Adding MPI functions

Since ESPResSo is using MPI to communicate data between its core processes some additional functions had to be added to ESPResSos ghost communication Listing 4.2:

mpi_gather_timers

This function is used by the TCL interface to obtain a list of all Timers from all the processes involved. The list is stored as a vector, mapping the string wich uniquely identifies the Timers to their data.

mpi_reset_timers

This function triggers a reset of all timers on all the ESPResSo core processes.

mpi_timers_slave

This function listens to the ESPResSo master process during the simulation and if necessary executes either the reset or returns the Timers and their Values to the master process.

4.4.1 RunningAverage

The initialization of a Timers values as well as their iterative calculation are done within the RunningAverage Class Listing 4.3. Besides `add_sample` this class also provides functions to update every value of the Timer as well as a clear function, which is triggered when the Timer is being reset.

[2] e.g. the simulations warmup phase, where the system does not necessarily follow the correct physical laws since it just has to be balanced.

Listing 4.3 In RunningAverage.hpp the values for every Timer are calculated every time the Timers `.stop()` function is executed. Every timer includes the amount of times the function has been executed (`n`), the time of the last measurement taken (`t`), the average time of the Timer (`avg`), the variance (`var`), the standard deviation (`sig`) as well as the minimal and maximal value, the Timer has reached yet (`min` respectively `max`). RunningAverage also includes a clear function which is used to reset Timers when invoking the `timer reset` command in the TCL interface.

```
void add_sample(Scalar s) {
  m_n++;
  m_t = s;
  if (m_n == 1) {
    m_old_avg = m_new_avg = s;
    m_old_var = 0.0;
    m_min = m_max = s;
  } else {
    m_new_avg = m_old_avg + (s - m_old_avg) / m_n;
    m_new_var = m_old_var + (s - m_old_avg) * (s - m_new_avg);
    m_old_avg = m_new_avg;
    m_old_var = m_new_var;
    m_min = std::min(m_min, s);
    m_max = std::max(m_max, s);
  }
}

void clear() {
  m_n = 0;
  m_min = std::numeric_limits<Scalar>::infinity();
  m_max = -std::numeric_limits<Scalar>::infinity();
}

int n() const { return m_n; }

Scalar t() const { return m_t; }

Scalar avg() const {
  if (m_n > 0)
    return m_new_avg;
  else
    return 0.0;
}

Scalar var() const {
  if (m_n > 1)
    return m_new_var / m_n;
  else
    return 0.0;
}

Scalar sig() const { return std::sqrt(var()); }

Scalar min() const { return m_min; }

Scalar max() const { return m_max; }
```

Listing 4.4 The Timer command for the TCL interface is implemented. It supports the retrieval of all Timers in the simulation as well as the reset of all timers.

```
namespace {
string format_stats(const Utils::Timing::Timer::Stats &stats) {
 ostringstream ss;
 ss << stats.avg() << " " << stats.sig() << " " << stats.var() << " "
   << stats.min() << " " << stats.max() << " " << stats.n() << " " << stats.t();
 return ss.str();
}
}

vector<map<string, Utils::Timing::Timer::Stats>> mpi_gather_timers();
void mpi_reset_timers();

int tclcommand_timer(ClientData data, Tcl_Interp *interp, int argc,
               char *argv[]) {
 if (argc == 1) {
  auto timers = mpi_gather_timers();
  char buf[8];
  for (int i = 0; i < timers.size(); ++i) {
   snprintf(buf, 8, "%d", i);
   for (auto &it : timers[i]) {
    Tcl_AppendResult(interp, "{ ", buf, " ", it.first.c_str(), " ",
                 format_stats(it.second).c_str(), " } ", nullptr);
   }
  }
 } else if ((argc == 2) && (string(argv[1]) == "reset")) {
  mpi_reset_timers();
[...]
```

Listing 4.5 The newly implemented Timer command is registered so that it can be recognized by the TCL user interface.

```
static void tcl_register_commands(Tcl_Interp* interp) {
 [...]
 REGISTER_COMMAND("timer", tclcommand_timer);
}
```

4.4.2 Tcl Interface

Finally the Timers have to be added to the TCL interface. This is done by first adding a `timer` command to the interface, which enables the user to use the timer objects inside the interface and then by adding this command to ESPResSo's TCL Interpreter, so that the command is actually recognized by the user interface.

To add the Timer Feature to the TCL interface a `timer` command is defined Listing 4.4. This command first checkst for command line arguments; if the `reset` argument is

Listing 4.6

```
#ifdef WITH_INTRUSIVE_TIMINGS
  auto &t_some_timer_name = Utils::Timing::Timer::get_timer("some_timer_name");
  t_some_timer_name.start();
#endif

  [CODE SECTION TO BE MEASURED]

#ifdef WITH_INTRUSIVE_TIMINGS
  t_some_timer_name.stop();
#endif
```

provided the `mpi_reset_timers()` function is executed, setting all timers back to their initial values. If on the other hand no argument is provided, the Timers are collected from all the ESPResSo core processes and then returned to the TCL interface using the TCL commands `format_stats` function.

Finally so that the TCL Interface is able to recognize the newly created `timer` command, the interface has to be made aware of it. This happens by adding the command to the `tcl_register_commands` functionListing 4.5, which is executed when the TCL interpreter is started.

4.5 Using Timers

To be able to use a Timer it has to be first declared in the ESPResSo Source CodeListing 4.6; there it can mark a section of code, which is to be measured. Timers are identified by the string naming them. [3] To be able to toggle the Timers Feature in an easy way a precompiler definition has been added to ESPResSo (`WITH_INTRUSIVE_TIMINGS`). When using it, the Timer Feature can easily be turned on and of, when configuring up ESPResSo with tools like `ccmake`.

In the TCL interface Timers can be either reset (using the command `timer reset`) or their values can be extracted from the core processes by iterating over the `timer` variable[4].

[3] If multiple timers with the same identifying string exist, they will be considered the same Timer.

[4] This can be done like with any other list in TCL, e.g. by iterating over all timers using a `for each` loop.

5 The Lennard-Jones Interaction

This chapter covers the first simulation scenario. The Lennard-Jones Interaction is introduced and then examined, first by using Perf and Valgrind and then by making use of the newly implemented Timers Feature.

5.1 The Lennard-Jones Potential

The Lennard-Jones Potential is a relatively simple model, which can be used to approximate forces between multiple particles in space depending on their distance. It can be calulated very easy and is thus used very often in simulations.

In Lennard-Jones InteractionsFigure 5.1 particles which are in very close proximity to each other repel each other, the closer particles get to each other, the higher becomes the energy repelling them. At greater distances Particles attract each other due to various forces (e.g. the Van der Waals force). At a certain distance (σ) the potential becomes zero.

5.2 Lennard-Jones implementation in ESPResSo

To use the Lennard-Jones interaction in a simulation the feature has to be enabled by adding the `#define LENNARD_JONES` option has to be used in ESPResSo's configuration file (`myconfig.hpp`).

In ESPResSo the Lennard-Jones potential is probably the simplest of particle interactions possible: In the beginning of a typical Lennard-Jones simulation a number of particles is put into the simulation box - usually using randomized coordinates. Depending on these coordinates the particles are thrown into one of the multiple cells by the ESPResSo master process. From then on the master process has no knowledge of particles in cells other than his own. The various cells do however have the knowledge about which cells are their neighbours and during the integration neighbouring cells communicate with each other using MPI.

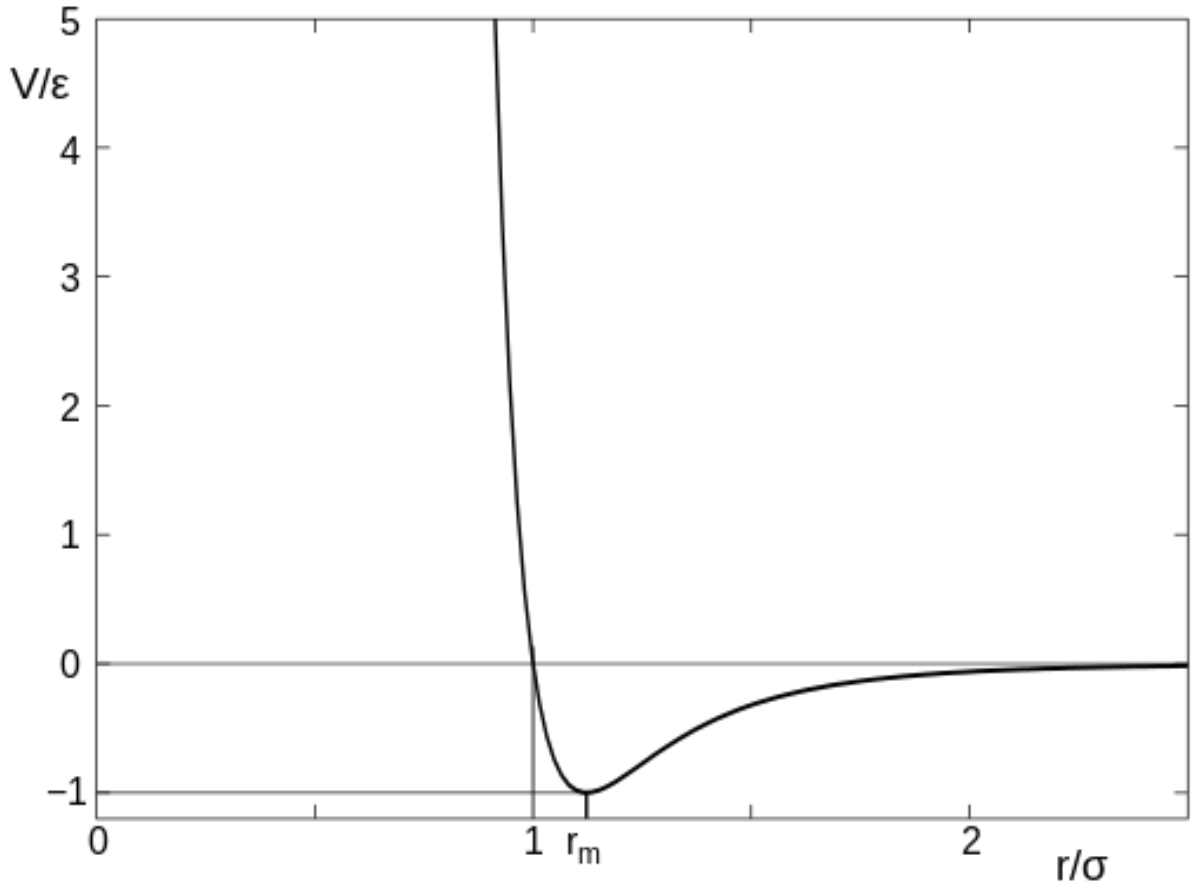


Figure 5.1: Typical example of a Lennard-Jones potential. This graph visualizes how the strength of the particle force counteracts the distance between particles

Later during the integration steps and depending on whether the verlet list feature has been turned on, the cells communicate particles which have impact on their Neighbours. If verlet lists are enabled, the ESPResSo will use these lists to calculate the new position of a particle by calculating the impact of the Lennard-Jones Potential on a specific particle from all particles which are part of its verlet list.

Verlet lists for a specific particle contain all particles surrounding it up to the maximal distance of Lennard-Jones impact plus an additional small distance, the so called skin. Whenever a single particle leaves its verlet list all the verlet lists have to be rebuild. The additional skin is used in order to minimize the amount of rebuilds in the lists. If verlet lists are disabled the linked cells feature is used to calculate the potential: All particles in the neighbouring cells are used to calculate the particles new position, which is of course much less performant, but still way better than to calculate forces by iterating over all particles in the simulation box.

After creating all particles a typical simulation defines interactions which are going to be simulated.

Then during a warmup phase all particles are being integrated for a number of timesteps, but not according to realistic physical rules. Instead a forcecap is used on the system to prevent unrealistic particle forces from occuring. This step is repeated over and over while every time the forcecap is increased until the system is stable enough.

Then the actual simulation begins: The system is integrated in a loop and in between the integration steps data can be retrieved from the simulation by using several of ESPResSos builtin features.

The Lennard-Jones simulation used in this work was one of the sample scripts and can be found in `samples/tcl/lj_liquid.tcl` .

5.3 Analyzing a Lennard-Jones Simulation

5.3.1 Using `perf report` to get a rough overview

To get an overview of what was done internally by ESPResSo when executing the `lj_liquid.tcl` script perf was used to generate a report. The four hot zones identified by the report were (see Figure 5.2):

- `29.54%` calculate_verlet_ia
- `17.85%` build_verlet_lists_and_calc_verlet_ia
- `6.87%` std::generate_canonical<double, 53ul, std::mersenne_twister_engine<...
- `5.69%` init_forces

The functions `calculate_verlet_ia` as well as `build_verlet_lists_and_calc_verlet_ia` are both part of ESPResSo's core loops: If the verlet lists can simply be used to calculate all particle forces `calculate_verlet_ia` is executed. Whenever a single particle violates the verlet criterium all the lists have to be rebuild before calculating the particle forces for the integration step and `build_verlet_lists_and_calc_verlet_ia` is executed.

The `std::generate_canonical<double, 53ul, std::mersenne_twister_engine<` refers actually the Mersenne Twister Engine[MN98], the random number generator which is currently being used by ESPResSo.

Lastly the `init_forces` function "initialize real particle forces with thermostat forces and ghost particle forces with zero."[dev].

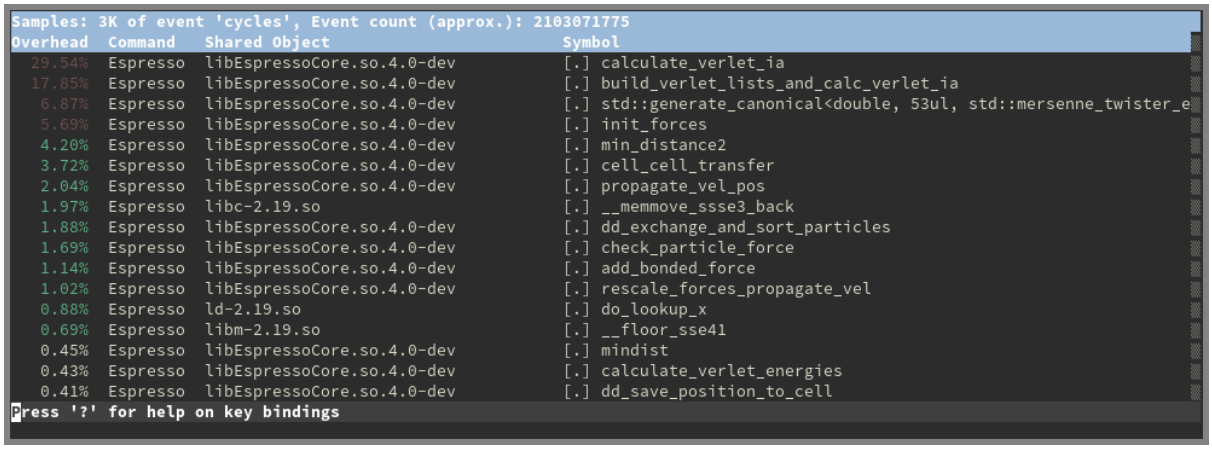
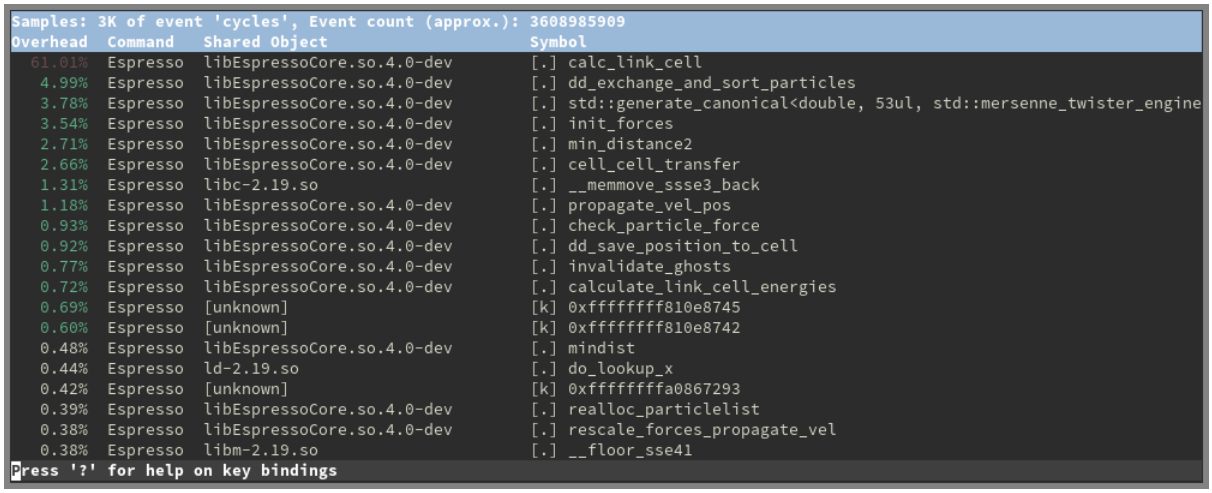


Figure 5.2: Perf report on the **lj_liquid.tcl** simulation script showing the hot zones during execution

To cover all possible scenarios at this point the same simulation was rerun with a small change to the simulation script such that verlet lists were turned off and in their stead the linked cells feature was activated. The `cellsystem domain_decomposition -no_verlet_list` option was added to the scripts system setup section. As could be expected the `calculate_verlet_ia` and the `build_verlet_lists_and_calc_verlet_ia` were not executed in this run. The perf report of this simulation run (see Figure 5.3) revealed a different hot zone:

- `61.01%` calc_linked_cell
- `4.99%` dd_exchange_and_sort_particles
- `3.78%` std::generate_canonical<double, 53ul, std::mersenne_twister_engine<...
- `3.54%` init_forces

Instead of utilizing the verlet lists now during every integration step ESPResSo has to look at all particles in the adjacent cells to calculate a particles force. As expected `calc_linked_cell` uses up slightly greater part of the runtime than the `calculate_verlet_ia` and the `build_verlet_lists_and_calc_verlet_ia` functions combined. Additionally the amount of ghost communication between the cells has risen, which can for example be observed by the function `dd_exchange_and_sort_particles` playing a slightly greater role (4.99% as opposed to 1.88% in the previous simulation) than in the simulation in which verlet lists were used.



```
Samples: 3K of event 'cycles', Event count (approx.): 3608985909
Overhead  Command   Shared Object                    Symbol
  61.01%  Espresso  libEspressoCore.so.4.0-dev       [.] calc_link_cell
   4.99%  Espresso  libEspressoCore.so.4.0-dev       [.] dd_exchange_and_sort_particles
   3.78%  Espresso  libEspressoCore.so.4.0-dev       [.] std::generate_canonical<double, 53ul, std::mersenne_twister_engine
   3.54%  Espresso  libEspressoCore.so.4.0-dev       [.] init_forces
   2.71%  Espresso  libEspressoCore.so.4.0-dev       [.] min_distance2
   2.66%  Espresso  libEspressoCore.so.4.0-dev       [.] cell_cell_transfer
   1.31%  Espresso  libc-2.19.so                     [.] __memmove_ssse3_back
   1.18%  Espresso  libEspressoCore.so.4.0-dev       [.] propagate_vel_pos
   0.93%  Espresso  libEspressoCore.so.4.0-dev       [.] check_particle_force
   0.92%  Espresso  libEspressoCore.so.4.0-dev       [.] dd_save_position_to_cell
   0.77%  Espresso  libEspressoCore.so.4.0-dev       [.] invalidate_ghosts
   0.72%  Espresso  libEspressoCore.so.4.0-dev       [.] calculate_link_cell_energies
   0.69%  Espresso  [unknown]                        [k] 0xffffffff810e8745
   0.60%  Espresso  [unknown]                        [k] 0xffffffff810e8742
   0.48%  Espresso  libEspressoCore.so.4.0-dev       [.] mindist
   0.44%  Espresso  ld-2.19.so                       [.] do_lookup_x
   0.42%  Espresso  [unknown]                        [k] 0xffffffffa0867293
   0.39%  Espresso  libEspressoCore.so.4.0-dev       [.] realloc_particlelist
   0.38%  Espresso  libEspressoCore.so.4.0-dev       [.] rescale_forces_propagate_vel
   0.38%  Espresso  libm-2.19.so                     [.] __floor_sse41
Press '?' for help on key bindings
```

Figure 5.3: Perf report on the same **lj_liquid.tcl** simulation with verlet lists turned off

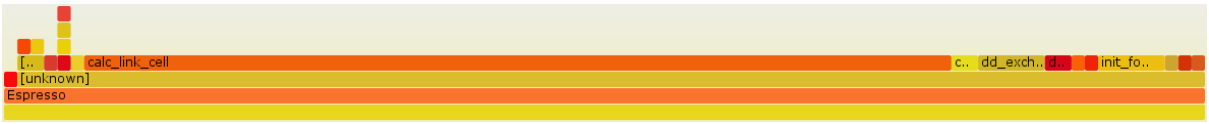
Listing 5.1 Creating a Flamegraph featuring ESPResSo's Lennard-Jones sample script using `perf` as well as the Perls cripts from Brendan Greggs github Repository:
At first a `perf` record containing all the necessary information is generated using specific commandline options. Then in a second step the information in `perfs` output script is parsed into an easy to visualize format and finally this file is being processed into an interactive and searchable svg image. The `egrep` command is used in the last step to ensure only information on ESPResSo is displayed in the Flamegraph

```
sudo perf record -F 99 -a -g -- ./Espresso lj_liquid.tcl
perf script | ~/git/FlameGraph/stackcollapse-perf.pl out-lj.perf-folded
egrep 'Espresso' out-lj-no-verlet.perf-folded | ./flamegraph.pl > lj-no-verlet.svg
```

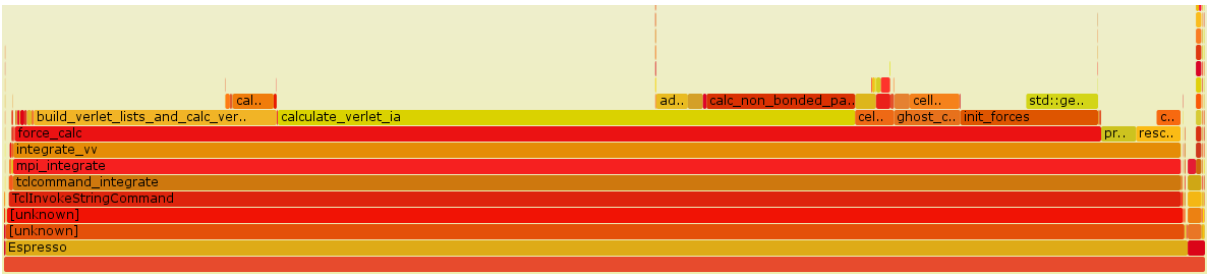
5.3.2 `perf` Flamegraphs

Next `perf` was run to create Flamegraphs[Gre16] to get a visualiztion of the time each of ESPResSos components needed when beeing executing this particular simulation script. When recording the execution of a programm perf looks constantly on the stack and saves its contents into the `perf.data` outputfile. Using a set of Perl scripts written by Brendan Gregg it is then possible to filter all this collected data and to create Flamegraphs:

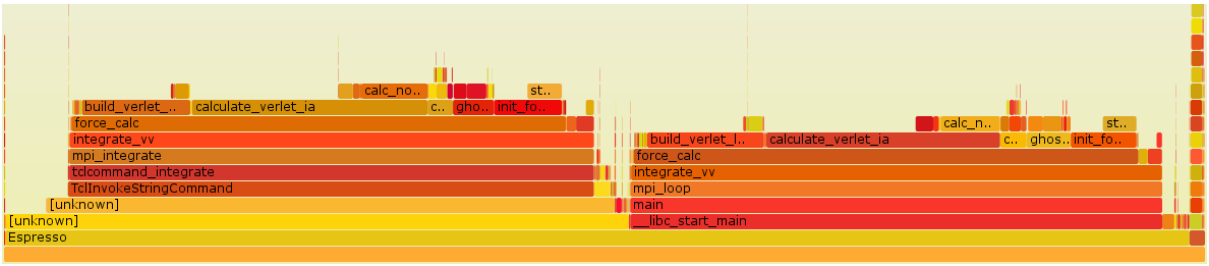
In a Flamegraph the x-axis represents a collection of the stack traces and is usually ordered alphabetically. It does however not in any way show any information regarding time other than the relativ amount of time spent in a function. The y-axis features the depth of the stack: a function is always the child of the function below it. The great advantage of Flamegraphs over the usual perf records is the fact that a function executed by different parents is displayed in different places.



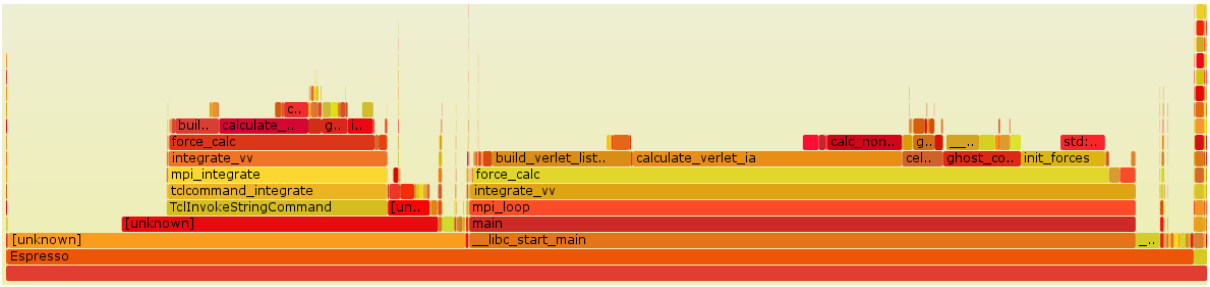
(a) Lennard-Jones Simulation Script with the linked lists feature enabled run by 1 processor



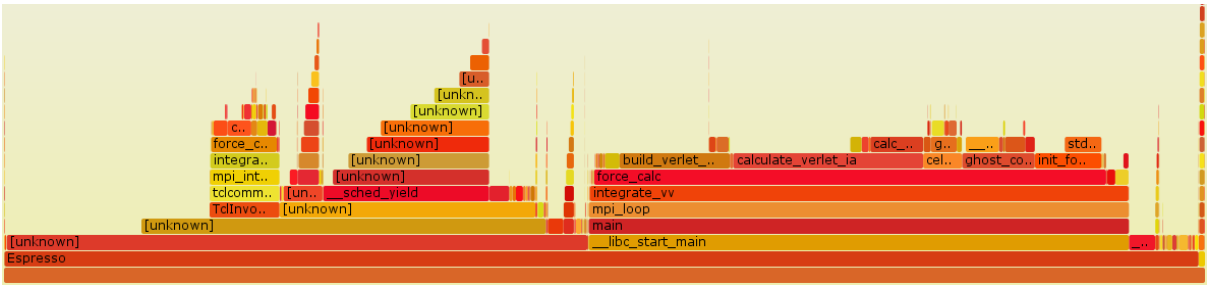
(b) Lennard-Jones Simulation Script with the verlet lists feature enabled run by 1 processor



(c) Lennard-Jones Simulation Script with the verlet lists feature enabled run by 2 processors



(d) Lennard-Jones Simulation Script with the verlet lists feature enabled run by 4 processors



(e) Lennard-Jones Simulation Script with the verlet lists feature enabled run by 8 processors

Figure 5.4: Various perf Flamegraphs

From the Flamegraphs it could be deduced, that the main difference between executing the Lennard-Jones Script with the verlet lists disabledFigure 5.4a and running the Script while using the featureFigure 5.4a was indeed the use of the `calc_link_cell` function instead of both the `calculate_verlet_ia` and the `build_verlet_lists_and_calc_verlet_ia` function as well as the increased use of the `dd_exchange_and_sort_particles` function.

Some fields in the flamegraph read `[unknown]` as the Executable monitored by perf wasn't compiled with debugging options turned on. In case of the `perf` records visualized here these turned out to be mostly `TCL` or `MPI` related. Both, `TCL` as well as `MPI` were compiled from source with debuuging options turned on, however the corresponding Flamegraphs did not provide any additional information which was useful at this point.

In a next step Flamegraphs were generated for a run of the script using the verlet list feature and 2,4 as well as 8 cores for the simulation(Figure 5.4). This highlighted the way `MPI` works when executing an ESPResSo script via the `mpirun -np X`[1].

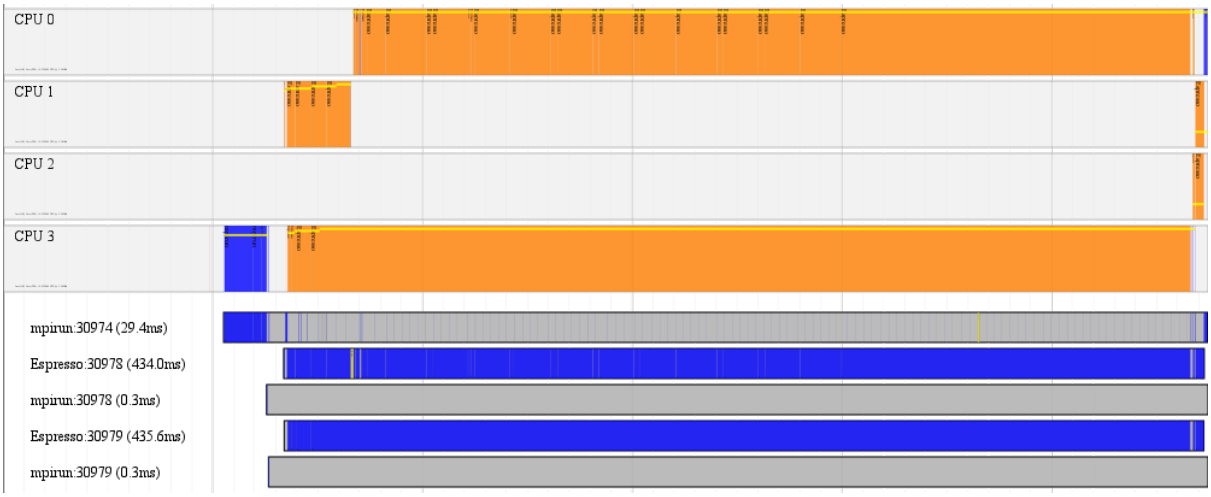
When looking at the Flamegraph of the run with a single processor the following of ESPResSo's behavoir could be deduced: There is a ESPResSo Process which is invoked by ESPResSo's `TCL` shell when the script is started. When the `integrate` command inside of the Script is run the `tclcommand_integrate` function is started, which directly executes the `mpi_integrate` function in which then the main integration loop is started via the `integrate_vv` function. From there on `force_calc` is called, as well as the rest of the functions participating in the actual calculation of the integration.

Taking a closer look at the runs where more than just one processor were involved showed that there is indeed one ESPResSo Master process, which is invoked through ESPResSos TCL shell and behaves as explained above. However there is also a number of additional ESPResSo Slave processes which directly start up ESPResSo's `mpi_loop` function and run the main integration loop (`integrate_vv`) within. It could further be deduced that running the simulation with more processes increased the amount of time spent in Slave processes relative to the time spent in the masters process. Unfortunately - as `perf` Flamegraphs do not have the feature of visualizing per CPU so at this point the observations came to an end.

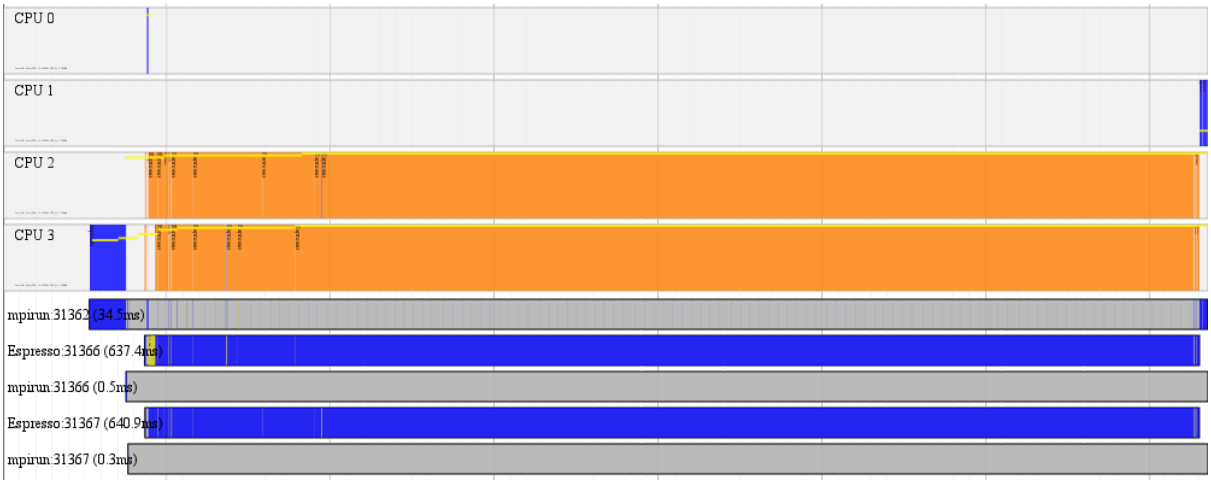
5.3.3 `perf` Timecharts - taking a closer look onto how MPI works

After using the `perf` to create Flamegraphs another of its features was used to take a closer look of MPI's behavoir regarding to ESPResSo. A `perf timechart` was created

[1] Where X denotes the amount of parallel processes.



(a) Lennard-Jones simulation script started without specifying the cores to be used (`mpirun -np 2`)



(b) Lennard-Jones simulation script specifying the two processors to be used (`mpirun -np 2 --slot-list 0:1`)

Figure 5.5: The Lennard-Jones script run with and without specifying to MPI on which cores to run.
In the upper Part of the pictures the `ESPResSo` process is orange, while the `mpirun` process is blue.
The lower part of the diagram shows which processes were active during the simulation, here blue correspondes to whether a process is running.

and from ESPResSo running the Lennard-Jones script using two processors. Afterwards another run was taken pinning the ESPResSo processes to specific cores.

As can be seen in **??** 5.5 when not directly specified the ESPResSo process can jump around between all the available cores. Unlike in the run, where the cores were directly

Listing 5.2 Using `callgrind`, one of six tools provided by the `Valgrind` instrumentation framework the callgraph of the Lennard-Jones interaction is created

```
valgrind --tool=callgrind --collect-jumps=yes --collect-systime=yes --trace-children=yes
    --dump-instr=yes ../build/Espresso lj_liquid.tcl
```

specified there are also gaps when ESPResSo changes the core in which no ESPResSo process is active. While this is not going to have any effect on the calculations it can definitely have effects on the ghost communication, where particles are communicated between various cells and thus this definitely has on the overall performance of ESPResSo.

Although this could obviously not read from the `perf timechart` there is actually a slight difference in between the performance of single processors within a NUMA domain and the performance of these processes run losely within the same NUMA domain[JWH14],[ZYS10]. Further research also revealed that it is advised to optimize the performance of MPI manually rather than to let the operating system take care of this, when setting up a high performance computing environment[HW10].

5.4 Using `Valgrind` to reveal some of ESPResSo's internals

After having used `perf` to identify the hot zones some more insight into ESPResSos internal way of functioning was needed. The Lennard-Jones simulation script was run using Valgrind to create a callgraph (see Listing 5.2). This callgraph was then viewed using a tool called `kcachegrind` to view and interact with the found data.

Since the CPU on which the Lennard-Jones script was run was simulated by `Valgrind`, the data obtained is in one way more reliable than the data `perf` provided: `Valgrind` actually knows what happens within the CPU, while `perf` merely deduces coherences of functions and objects by looking regularly at the content of the stack and then piecing all the information together. On the other hand `perf` does, since the CPU is real as opposed to `Valgrind`s simulated one represent the more realistic behavoir of the profiled program.

5.4.1 The Lennard-Jones callgraph

The Lennard-Jones callgraph obtained using `Valgrind` (see Figure 5.6) revealed how the integration phase of the Lennard-Jones script is being processed when executing ESPResSo. As could be expected the callgraph shows that most

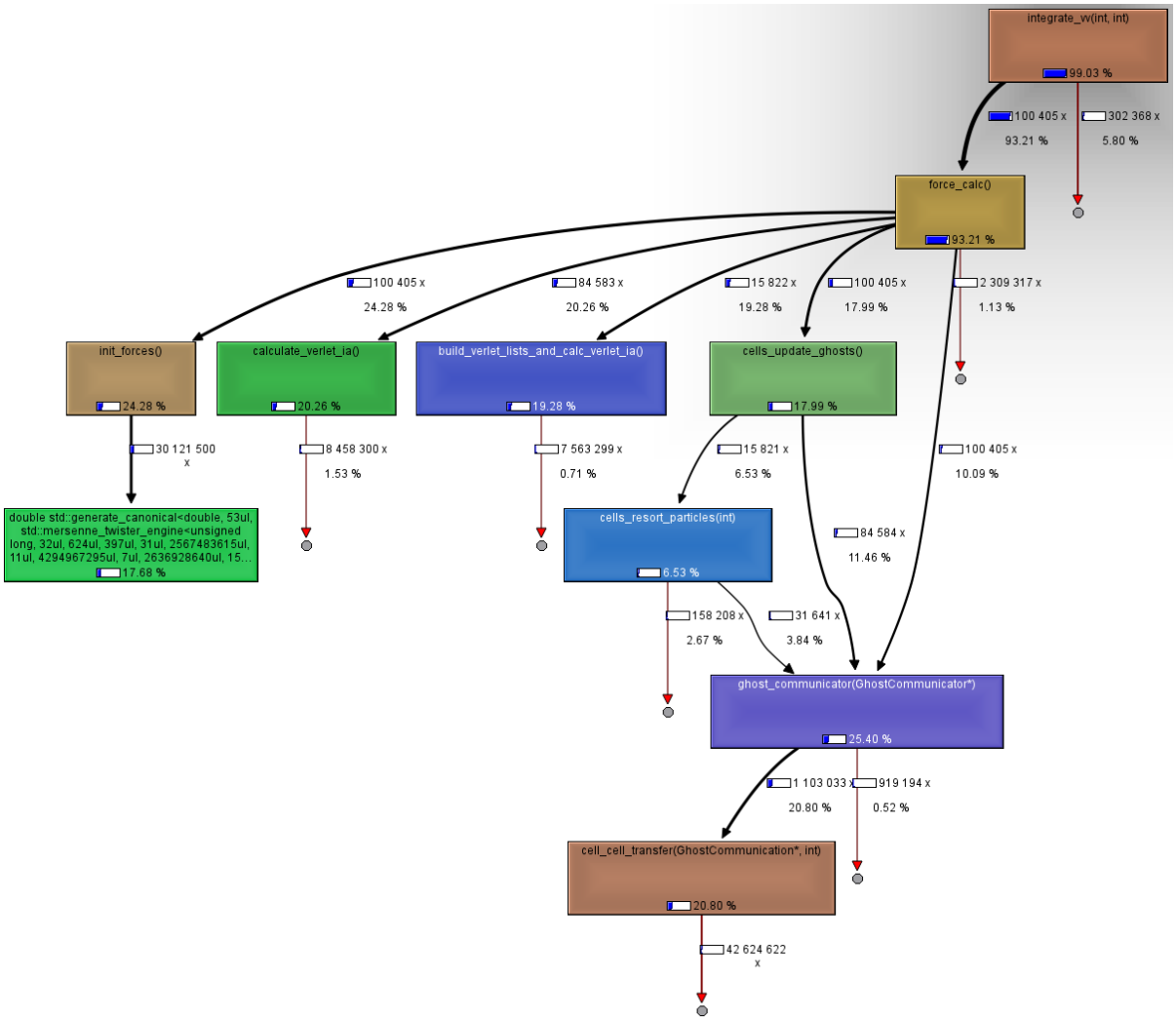


Figure 5.6: Callgraph of the Lennard-Jones Simulation created by `Valgrind` and viewed using `kcachegrind`. The parts of the callgraph leading to the `integrate_vv` function have been skipped for better readabillity, since they do not contain any additional information which is relevant to this thesis.

of the time is spend within both of the core loops (`calculate_verlet_ia` and `build_verlet_lists_and_calc_verlet_ia`) as well as in the `init_forces`) function and the a few other functions, which seem to have much less impact on the runtime of ESPResSo. These functions turned out to be the functions, which are responsible for the ghost communication.

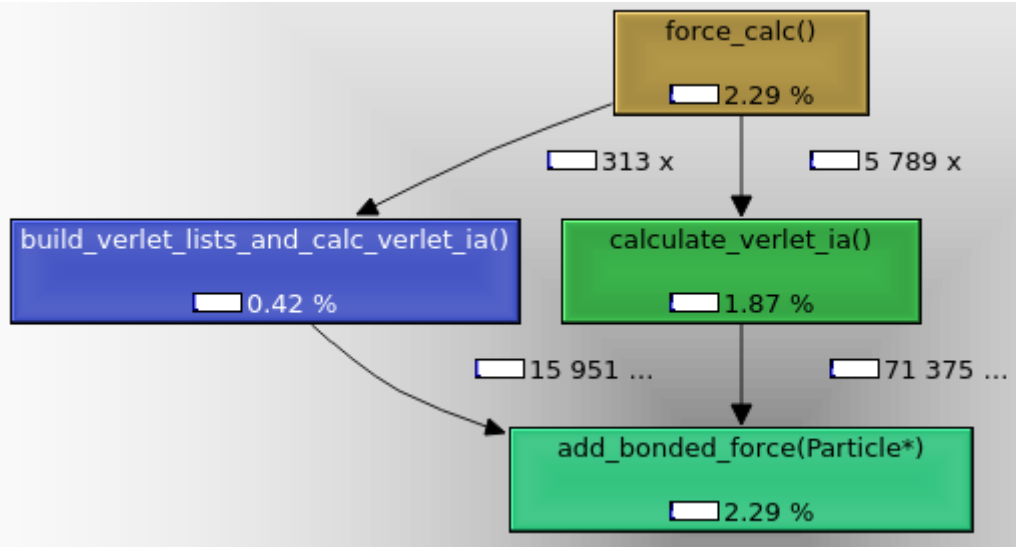
Listing 5.3 The `add_bonded_force` function instantiated various variables before checking whether the simulation was actually using bonds

```
inline void add_bonded_force(Particle *p1)
{
  double dx[3] = { 0., 0., 0. };
  double force[3] = { 0., 0., 0. };
  double force2[3] = { 0., 0., 0. };
  double force3[3] = { 0., 0., 0. };
#if defined(HYDROGEN_BOND) || defined(TWIST_STACK) || defined(OIF_LOCAL_FORCES)
  double force4[3] = { 0., 0., 0. };
#endif
#ifdef TWIST_STACK
  double force5[3] = { 0., 0., 0. };
  double force6[3] = { 0., 0., 0. };
  double force7[3] = { 0., 0., 0. };
  double force8[3] = { 0., 0., 0. };

  Particle *p5 = NULL,*p6 = NULL,*p7 = NULL,*p8 = NULL;
#endif
#ifdef ROTATION
  double torque1[3] = { 0., 0., 0. };
  double torque2[3] = { 0., 0., 0. };
#endif
  Particle *p2 = NULL, *p3 = NULL, *p4 = NULL;
  Bonded_ia_parameters *iaparams;
  int i, j, type_num, type, n_partners, bond_broken;

  i = 0;

  while (i<p1->bl.n) {

    [...]
```

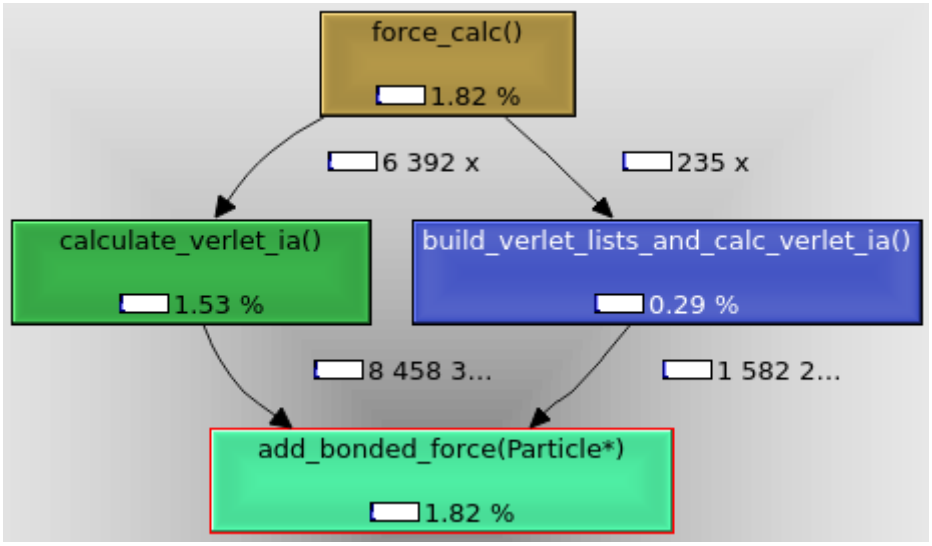
5.5 add_bonded_forces()

A first bottleneck was found when examining the callgraph for the Lennard-Jones script: a function called `add_bonded_force` took up more than 2 percent of the runtime and, although at this point there were no reference points on how much of the runtime specific functions within ESPResSo were supposed to take up it seemed a bit odd that a function related to bonded interactions would take up this much of runtime, when simulating a system without bonded particles.

The reason why the function took as much time was that the `add_bonded_force` function is called by the core loops and thus executed in every integrationstep. This function



(a) add_bonded_force function before moving the check outside the loop



(b) add_bonded_force function after moving the check outside the loop

Figure 5.7

Listing 5.4 Before initializing any variables and starting the while loop the particle is now checked for bonds and only executed when bonds are present.

```
inline void add_bonded_force(Particle *p1)
{

 type_num = p1->bl.e[i++];
 iaparams = &bonded_ia_params[type_num];
 type = iaparams->type;
 n_partners = iaparams->num;

 if (n_partners > 0) {

  double dx[3] = { 0., 0., 0. };
  double force[3] = { 0., 0., 0. };
  double force2[3] = { 0., 0., 0. };
  double force3[3] = { 0., 0., 0. };

  [...]

  while (i<p1->bl.n) {

    [...]
```

initialized various variables and then checked for bonds within its calculation loop, checking whether another particle in the current bond existed, but not checking if a bond existed in the first place (see Listing 5.3). This was fixed by adding a small check to the beginning of the function (see Listing 5.4) and resulted in a huge change of the functions runtime (see Figure 5.7)

5.6 Adding Timers to the Lennard-Jones script

At this point the Lennard-Jones interaction had been examined by both, `perf` as well as `Valgrind`. Using `perf` the hot zones of the Simulation had been identified, also the way MPI is utilized in ESPResSo had been visualized using Flamegraphs and Timecharts. Through running `Valgrind` the internal structure of ESPResSos functions was revealed. Both tools did a great job by revealing some of ESPResSos structure, however both tools were also not able to provide with absolute data, `perf` because of it's non-invasive approach of only checking the stack and `Valgrind` due to the fact that the CPU used to profile the Simulation was simulated. To get actual comparable data various additional tools were examined and compared (see chapter 2). After examining these tools it was however decided to not use any of them to gather the needed information; On

the contratry it was decided to add a Timer Feature to EPResSo (see chapter 4): since ESPResSo has over the time evolved into having a very successfull development model (DevOps[BWZ15][Htt12]) and automated performance tests could greatly influence as well as enhance the quality of the software over time.

These Timers were then added to ESPResSo's source code, the code was compiled using the `#define WITH_INTRUSIVE_TIMINGS` preprocessor directive, under which all the timers were included into ESPResSos source. From then on it was possible to access the timers in the Lennard-Jones Script using the `timer` object.

5.6.1 Adding timers to the Lennard-Jones Simulation

The Lennard-Jones tcl script was altered to work with timers: In a first attempt the a simple loop was added in the end of the script printing out the measured timers to the command line. Since this was a rahter poor solution, especially for analyzing multiple simulations also since this work exclusively used ESPResSos `tcl` interface and there was at the time not yet an integration for the Timers into ESPResSo's `python` interface the script was then altered to create JSON-like output files (see Listing 5.5), which later could be concatinated into a JSON file and corrected by using a few regular expressions. From then on `python` functions were used to vizualize the data.

5.7 Timers

All simulations using the Timer Feature were run on the BEE Cluster of the Istitute for Computational Physics (ICP) at the University Stuttgart. The Cluster consists of 768 compute cores interconnected via 40 Gbps Infiniband. Three different types of nodes are available:

- 42 standard Nodes

  These nodes each consist of a Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 1x Nvidia Titan Black (6GB GDDR5)

- 2 big nodes

  These nodes consist of Intel Quad Xeon E5-4640v2, 4x10 cores at 2.2 GHz, 512 GB RAM

Listing 5.5 Timers are written into an outputfile to be processed later into correct JSON syntax.

```
inter forcecap 0
set i 0

while {$i < $int_n_times} {

   integrate $int_steps
   set energies [analyze energy]

   # write timers into backup_file every $backup_timer steps:
   if {$i%$backup_timer==0} {
      puts $backup_file "\"$i\": \["
      set n 1
      foreach t [timer] {
         puts $backup_file " \["
         puts $backup_file " \"[lindex $t 0]\","
         puts $backup_file " \"[lindex $t 1]\","
         puts $backup_file " \"[lindex $t 2]\","
         puts $backup_file " \"[lindex $t 3]\","
         puts $backup_file " \"[lindex $t 4]\","
         puts $backup_file " \"[lindex $t 5]\","
         puts $backup_file " \"[lindex $t 6]\","
         puts $backup_file " \"[lindex $t 7]\""
         if {$n%[expr {$number_of_timers * $sim_procs}]} {
            puts $backup_file " \],"
         } else {
            puts $backup_file " \]"
         }
         incr n
      }
      puts $backup_file " \],"
      flush $backup_file
   }
   incr i
}

[...]
```

- 1 multi GPU node

  This node consists of Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 4x Nvidia Tesla K20C (5GB GDDR5)

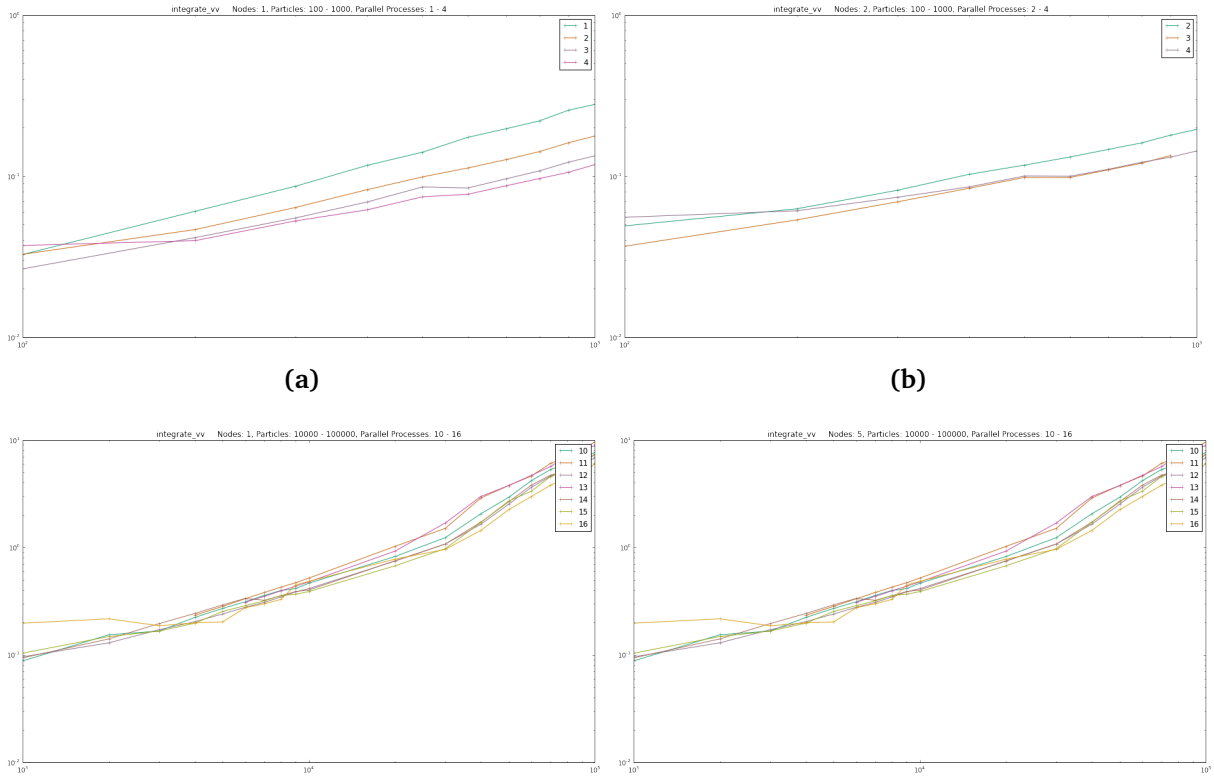
The ESPResSo Version used was ESPResSo3.2.03836g4088074git, openmpi Version 1.10.1 is used and the Operating System on the BEE Cluster is CentOS release 6.7, running the 2.6 Linux Kernel. All simulations use systems with a density of 0.7 and box sizes according to the amount of particles used in the simulations.

At first multiple Lennard-Jones simulations were started: The simulations ran for 1000 MD Timesteps each and the particles in the simulated Systems ranged from 100 to 1000 in steps of 100, from 1000 to 10000 in steps of 1000 and from 10000 to 100000 in steps of 10000. The simulations were run on multiple nodes with various numbers of ESPResSo core processes. It can be observed that during these simulations ESPResSo behaves as follows:

In a small system simulations that have smaller numbers of particles the integration steps are computed faster in simulations with fewer core processes involved. When however the system contains more particles, at somt point. This due to the MPI communication being the slowing factor in simulations where only a few particles are distributed over multiple processes. When however the amount of particles increases, the communication steps carry less weight compared to the rest of the integration and multiple processes can calculate the forces for the next integration step much faster than a single one can. This Behavoir can for example be observed in Figure 5.10: The simulations with 4 respective 16 processes are significantly slower than the processes using fewer processes, when the system is small and contains fewer particles. When however the amount of particles increases, these simulations surpass the rest of the simulations, when it comes to speed.

A similiar run of simulations was started on 5 nodes with much larger systems and the processes ranging from 32 to 80, however this simulation run was abandoned in the process, since it behaved exactly as the smaller simulations and time on the simulation cluster is shared between researchers. Figure 5.9 shows the partial results of this run of simulation: Again the simulations with a lot of processes start out slower than the rest and then surpass the rest of the simulations when it comes to speed as the systems get bigger.

Additionally the Lennard-Jones system was simulated over a number of timesteps with Timers gathered for every timestep of the simulation, so that the behavoir of ESPResSo during the simulations can be examined. This was run using only a single ESPResSo core process and 100, 1000, 10000 and 100000 particles in the system. Additionally



(a) **(b)**

(c) **(d)**

Figure 5.8: These Graphs represent the speed of the simulations (time on the y-axis) integrationsteps depending on the amount of particles (on the x-axis) in the system and the amount of processes used in the computation process: a) and b) are run on 1 respective 2 nodes and feature up to 4 processes and systems containing 100 - 1000 particles , c) and d) are run on 1, respectively 5 nodes and feature each 10 to 16 processes and systems containing 100 - 1000 particles. It can be observed, that the simulations, which feature more core processes are slower in small systems, but surpass the other simulations, as the size of the systems grows. In smaller systems the MPI communication seems to create an overhead for multi process simulations, while in larger systems the speed gained from the additional computing power is greater than the time lost communicating particles.

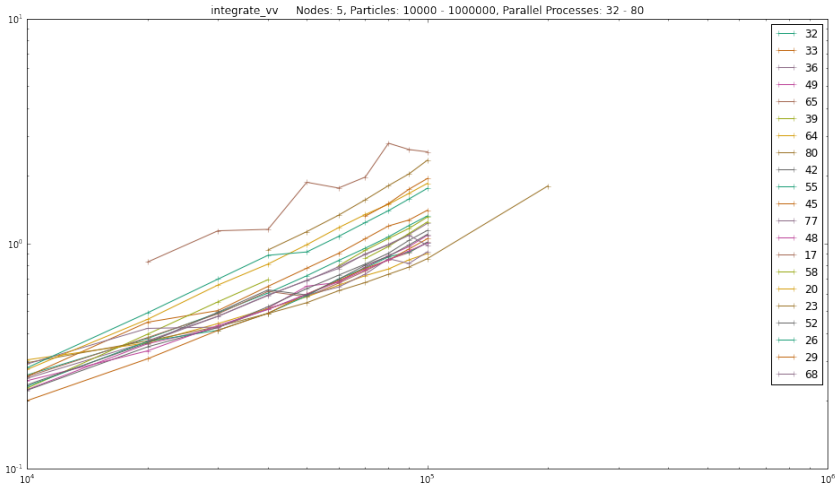


Figure 5.9: This Graph represent the speed of the simulations (time on the y-axis) integration steps depending on the amount of particles (on the x-axis) in the system and the amount of processes used in the computation process. 5 Nodes were used for each simulation, featuring between 32 and 80 processes each and between 10000 and 100000 particles. Since however the partial results showed exactly the same behavoir as the smaller simulation runs and the time on the simulation cluster is shared between all researchers this run was ended before all simulations were done.

the standard error was measured every integrationstep and the simulations were run up until the error was smaller than 5 percent.

This revealed that with increasing amount of particles in the system somehow the calculation of the verlet lists (`calculate_verlet_ia`) seems to become slower over time**??**. At first it seemed as if the function `dd_exchange_and_resort_particles` was responsible, however the simulation run with 100000 particles in the system, which is where the integration loop becomes notably smaller this functions runtime did not correlate with the runtime of the main integration loop.

Unfortunately due to a bug in ESPResSo, particularly the ghost communication process, simulations using the Timers Feature were crashed in some environments, since at some point in the simulations Timer data was lost due to a section of code which some times replaced the object containg the timer and could not be communicated. The BEE cluster of the ICP was such an environment and a lot of the simulations concerning ESPResSo's behavoir on simulations with multiple Nodes and multiple core processes were affected. By the time the bug was fixed unfortunately in this work there was no time left to rerun these simulations.

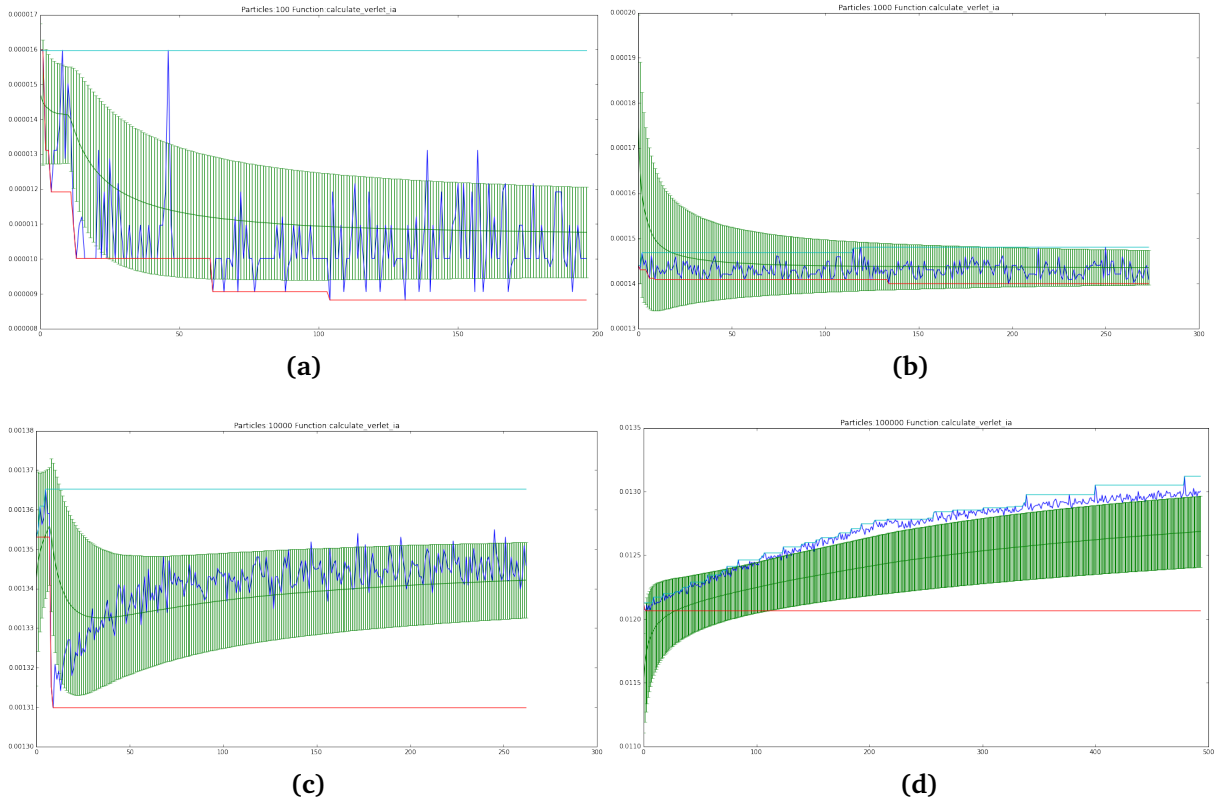


Figure 5.10: These graphs represent the time spent in the `calculate_verlet_ia` function (time on the y-axis) over the course of MD time steps (on the x-axis). The time spent calculating the verlet pair lists seems to increase over time. a) is a simulation with 100 particles b) is a simulation with 1000 particles c) is a simulation with 10000 particles and d) is a simulation with 100000 particles.

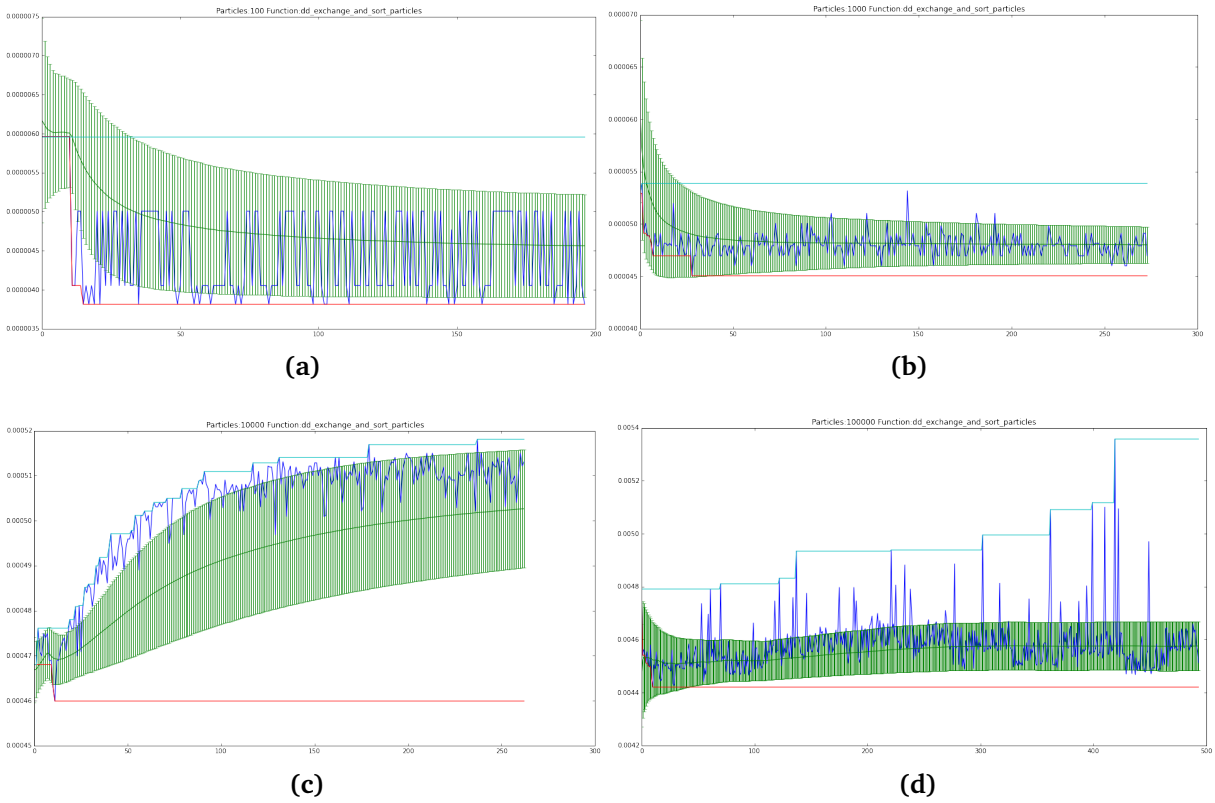


Figure 5.11: These graphs represent the time spent in the `dd_exchange_and_resort_particles` function (time on the y-axis) over the course of MD time steps (on the x-axis). This function which is part of the ghost communication seemed to be the reason why the time of the verlet list calculation increased**??**, however this could not be confirmed, since a simulation with an even bigger system (d)) showed that the time spent `dd_exchange_and_resort_particles` indeed does not increase enough over time to match the strange behavoir of `dd_exchange_and_resort_particles`; a) is a simulation with 100 particles b) is a simulation with 1000 particles c) is a simulation with 10000 particles and d) is a simulation with 100000 particles.

6 Electrostatics

This chapter covers the second scenario, electrostatic interactions between particles; first the P3M Interaction is introduced, then a P3M simulation is examined using Valgrind. After that the Timers Feature is used to take a closer look at the performance of electrostatic interactions in ESPResSo.

6.1 P3M

The second scenario deals with one of ESPResSo's methods to calulate Electrostatic interactions: the Particle Particle Particle Mesh (P3M) [BLD98; LG96]. These interactions between particles are generated by calculating the Particles short range interactions as well as their long range interactions by calculating the fast Fourier Transform (FFT)[BB74; PG96], which is more performant than e.g. the Ewald method[PG96].

6.2 P3M Implementation in ESPResSo

In ESPResSo's implementation of the P3M is split into two steps:

1. The calculation of the short range interaction

   This is done similiar to the calculation of the Lennard-Jones potential, using the `add_non_bonded_force` function.

2. The calculation of the long range interaction

   In ESPResSo's code the function `calc_long_range_forces()` initiates this process,starting two other functions, `p3m_calc_kspace_forces`, which is where the FFT is calculated by a set of special functions as well as `p3m_charge_assign`, which is a template function, used to assign charges into the grid.

6.3 Analyzing a P3M Simulation

In ESPResSo long range interactions of the P3M Method are calculated as follows: The system is represented as grid, every ESPResSo core process calculates the interaction for a part of this grid. Every core process interpolates the particles on the grid, then the information is communicated using MPI, after that the FFT is calculated, communicated back using MPI and then the new information from the grid is applied back to the particles in the system.

6.3.1 P3M vs Lennard-Jones

Other than the Lennard-Jones potential, using P3M introduces a new set of functions into the integration loops force calculation. These long range interactions are, as well as the short ranged interactions calulated on every integration step, since they are grid based FFT calculations it is to be expected that these functions take up most of the time of the interaction process.

6.3.2 The P3M Callgraph

Valgrind is used to create a callgraph of a typical P3M simulation script. It showes that indeed the long range interactions take up most of the execution time. Other than that the callgraph looks identical to the one of the Lennard-Jones interaction in chapter 5. It can be noted, that indeed the calculation of the long range interactions (`calc_long_range_forces()`) Figure 6.1 takes up most oof the simulation time, with the FFT calculation `p3m_calc_kspace_forces` Figure 6.2 being the most time consuming part of this process.

6.4 Timers

All simulations using the Timer Feature were run on the BEE Cluster of the Istitute for Computational Physics (ICP) at the University Stuttgart. The Cluster consists of 768 compute cores interconnected via 40 Gbps Infiniband. Three different types of nodes are available:

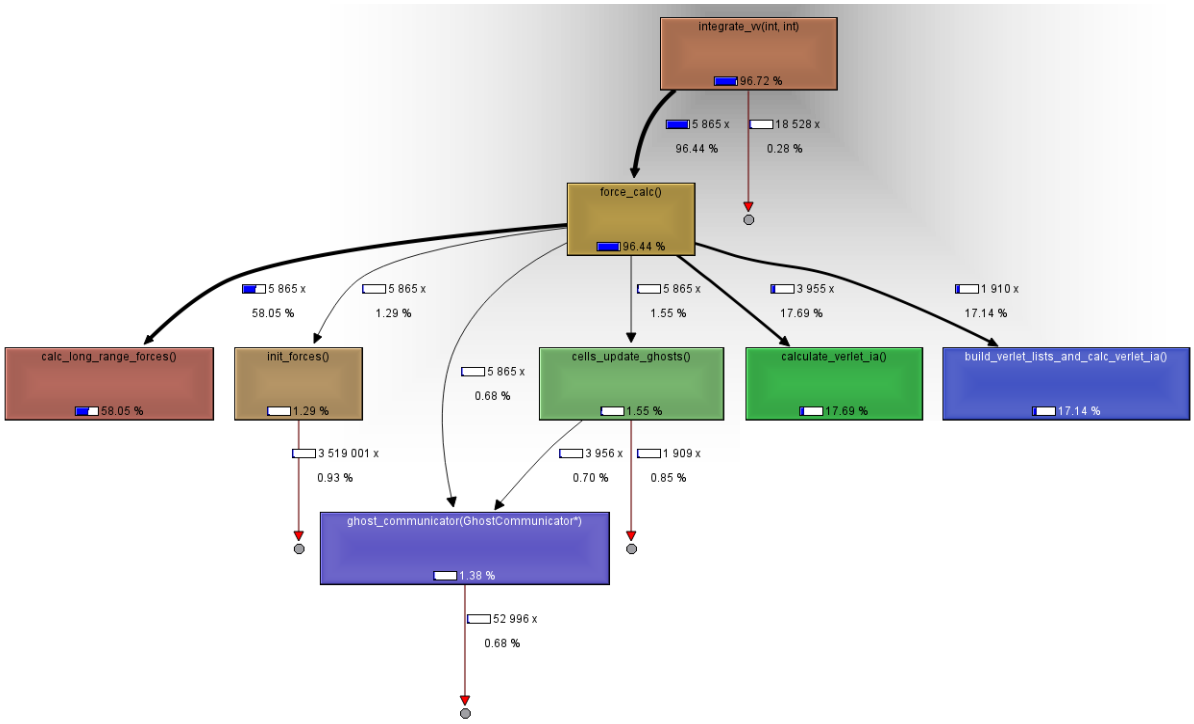


Figure 6.1: The callgraph of an Electrostatics simulation using P3M looks different that the callgraphs of the rest of the simulations: It features the additional execution of the `calc_long_range_forces()`: This function calculates the longrange interactions of the P3M scenario and makes use of computationally cheap FFT calculations. It can be observed that during the P3M simulation most of the time is spent in the long range calculation.

- 42 standard Nodes

  These nodes each consist of a Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 1x Nvidia Titan Black (6GB GDDR5)

- 2 big nodes

  These nodes consist of Intel Quad Xeon E5-4640v2, 4x10 cores at 2.2 GHz, 512 GB RAM

- 1 multi GPU node

  This node consists of Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 4x Nvidia Tesla K20C (5GB GDDR5)

The ESPResSo Version used was ESPResSo3.2.03836g4088074git, openmpi Version 1.10.1 is used and the Operating System on the BEE Cluster is CentOS release 6.7,

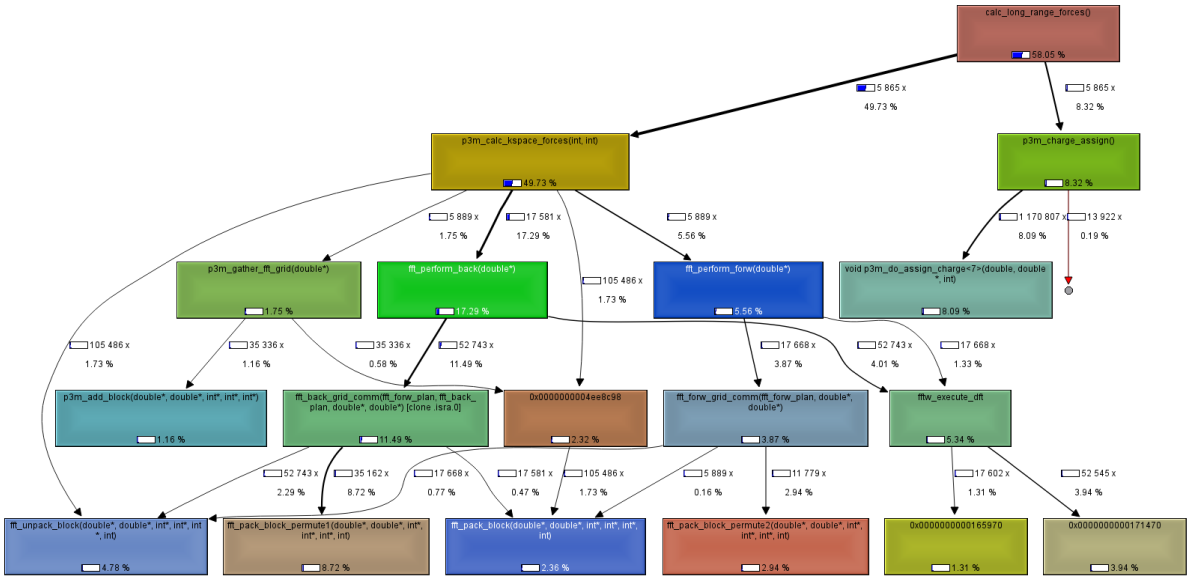


Figure 6.2: A closer look at the `calc_long_range_forces()` function reveals, that this function makes use of FFT calculations, which take up most of the simulations time.

calc_long_range_forces		
Processes:	1000 particles	10000 particles
1	0.00144456 s	0.00533132 s
2	0.00115211 s	—
4	0.000838543 s	—
8	0.000382556 s	—

Table 6.1: This Table shows the mean runtime of the `calc_long_range_forces` function from a set of simulations run with 1000 and 10000 particles in the system and 1,2,4 and 8 parallel processes. It can be observed, that adding more processes to the simulations decreases the mean runtime on every process, since a smaller part of the grid has to be calculated per process. Unfortunately due to a bug Electrostatics simulations with more than one process on the BEE Cluster currently fail.

running the 2.6 Linux Kernel. All simulations use systems with a density of 0.7 and box sizes according to the amount of particles used in the simulations.

A set of simulations is run on the BEE Cluster: The simulations are all run on one Node of the cluster and simulate an Electrostatics scenario with 1000 or 10000 particles using 1, 2, 4 and 8 parallel ESPResSo core processes. Table 6.1 shows the results of these

simulations. It can be observed, that by adding more processes to the simulation the time spent in the `calc_long_range_forces` function is significantly decreased. Due to a bug in the current Timers Feature all simulations on the BEE Cluster using 10000 particles have failed, whenever they used more than a single ESPResSo core process to compute. This bug has unfortunately not yet been fixed.

7 Bonded Interaction

This chapter covers the third scenario, bonded interactions, The FENE Interaction is introduced and then examined using Valgrind and the Timers Feature.

7.1 The FENE Potential

The third simulation scenario looks at a basic method of calculating bonded particle interactions: The Finitely Extensible Non-linear Elastic Model (FENE) [BAH87; WOB91] works by connecting beads using nonlinear springs, which can extend up to a maximal distance, after which these springs break.

7.2 Analyzing a FENE Simulation

In ESPResSo FENE Polymers are calculated by the `add_bonded_force()` function of the force calculation. Other than the calculation of this additional Interaction, the simulation is executed identical to a simulation of a Lennard-Jones system, which is discussed in chapter 5.

Since the only difference to the Lennard-Jones Scenario is that the `add_bonded_force()` function is actually used it is to be expected that the callgraph of this simulation looks identical to the callgraph of the Lennard-Jones simulation, with the exception of the amount of runtime which is used executing `add_bonded_force()`. Indeed Valgrind confirms this Theory, instead of taking up less than 2 percent (Figure 5.6) a bonded simulation takes up slightly over 6 percent of the global simulations runtime (Figure 8.1).

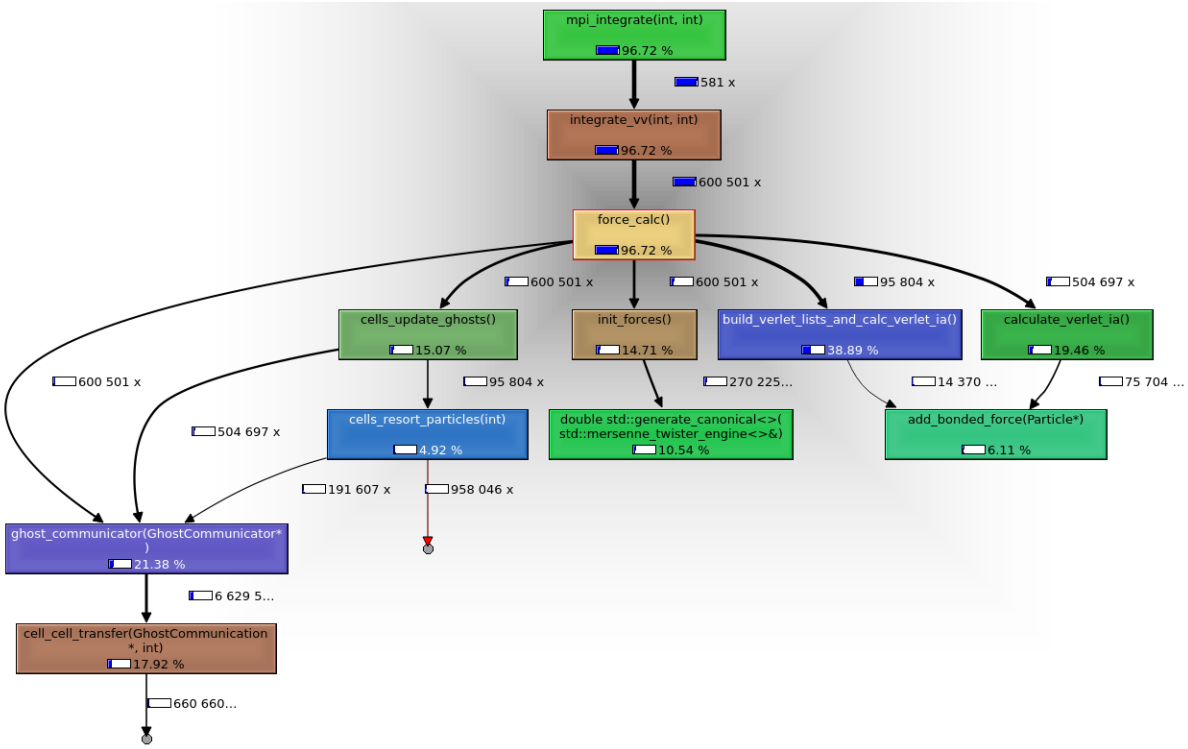


Figure 7.1: The callgraph of a FENE simulation looks exactly as the callgraph of the Lennard-Jones simulation, however by simulating the FENE interaction the `add_bonded_force()` function is actually used, which results in more time being spent by its execution.

7.3 Timers

All simulations using the Timer Feature were run on the BEE Cluster of the Istitute for Computational Physics (ICP) at the University Stuttgart. The Cluster consists of 768 compute cores interconnected via 40 Gbps Infiniband. Three different types of nodes are available:

- 42 standard Nodes

  These nodes each consist of a Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 1x Nvidia Titan Black (6GB GDDR5)

- 2 big nodes

  These nodes consist of Intel Quad Xeon E5-4640v2, 4x10 cores at 2.2 GHz, 512 GB RAM

add_bonded_force	
Processes:	100 Polymers
1	3.00159e-06 s
2	3.00147e-06 s
4	3.00091e-06 s
8	3.000751e-06 s
16	3.000898e-06 s

Table 7.1: This table shows the mean runtime measured for the function `add_bonded_force` in simulations with 100 polymers which have 15 beads each and 1,2,4,8 or 16 parallel ESPResSo core processes involved. It can be observed, that the mean runtime of this functionslightly decreases when simulations use more parallel processes. Otherwise the simulations seem to behave like the simulations from the Lennard-Jones scenario.

- 1 multi GPU node

  This node consists of Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 4x Nvidia Tesla K20C (5GB GDDR5)

The ESPResSo Version used was ESPResSo3.2.03836g4088074git, openmpi Version 1.10.1 is used and the Operating System on the BEE Cluster is CentOS release 6.7, running the 2.6 Linux Kernel.

A set of simulations is run on the BEE Cluster: The simulations are all run on one Node of the cluster and simulate an Bonded interaction scenario using FENE bonds with 100 polymers which have 15 beads each and using 1, 2, 4, 8 and 16 parallel ESPResSo core processes. Table 7.1 shows the results of these simulations. It can be observed, that by adding more processes to the simulation the mean runtime spent in the `add_bonded_force` function is slightly decreased, except for the simulation with 16 processes involved. This might be due to the fact, that ESPResSo's bonded interactions are calculated by iterating over each particles bonds lists. Apart from that the simulations behave similiar to the Lennard-Jones simulations.

8 Agglomeration

This chapter covers the last scenario, dynamic bonding interactions in agglomeration scenarios; dynamic bonding is introduced and explained. Then using Valgrind this interaction is examined further and finally the Timers Feature is used to analyze the performance of an agglomeration simulation.

8.1 Dynamic Bonding

The last scenario examined is a Agglomeration [IAKW14] scenario, which makes use of ESPResSo's dynamic bonding[ALK+13] implementation: During a simulation, whenever the distance between two particles bacomes closer than a defined bonding distance these particles become bonded. To be able to determine whether or not this applies to any two particles in the system, ESPResSo makes use of its collision detection Feature.

8.2 Analyzing an Agglomeration Simulation

In ESPResSo the collision detection is executed during the verlet lists calculation. It is part of the calculation functions which are started by the `add_non_bonded_pair_force` function. Whenever two particles collide, bonded interactions between particles are created. ESPResSo also allows the use of virtual sites: at the coordinates of the collision between particles two new virtual sites are created which are connected by a bond with a length of zero; this bond can not be removed during the simulation and the virtual sites are also immutably connected to their corresponding particle. As a result the newly bonded particles seem to stick together allowing for realistic and very detailed simulations of e.g. soot particle agglomeration.

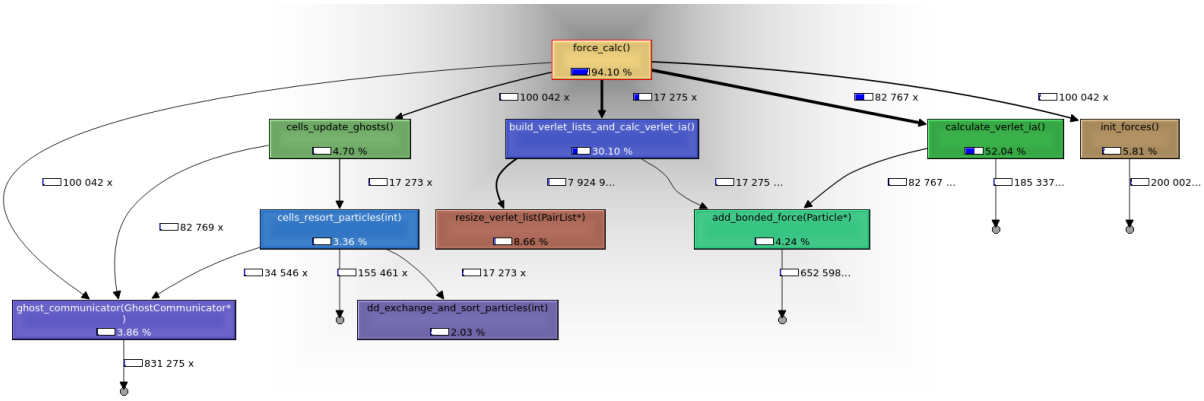


Figure 8.1: The callgraph of a Agglomeration simulation looks similiar to the the callgraph of the Lennard-Jones simulation, however due to simulating the bonds the `add_bonded_force()` function is used notably. Also the `calculate_verlet_ia` and `build_verlet_lists_and_calc_verlet_ia` take up over 80 percent of simulation runtime, since the Dynamic Bonding Feature is enabled and since the Collision Detection is run during these functions. The `resize_verlet_list` function seems to take up quite a lot of time and could indicate a possible bottleneck in ESPResSo.

8.2.1 Dynamic Bonding vs Bonded Interaction

The dynamic bonding Scenario similiar to the bonded interaction simulation from chapter 7, however the main differences are, that simulations of an agglomeration scenario do not necessarily start out with existing bonds between particles, but are added dynamically. Also the bonds added on the virtual sites are not using the FENE interaction, but are immutable.

8.2.2 The Agglomeration Callgraph

Compared to a Lennard-Jones simulation the same functions of ESPResSo are active: The collision detection process is part of the calculation of the non-bonded particle interactions, the bonds of virtual site particles are immutable and the bonds between particles which have collided in the simulation are calulated by the `add_bonded_force` function as they would be in a simulation of a bonded interaction.

Thus it can be expected that, since the collision detection is calculated in the verlet loops, that the functions `calculate_verlet_ia` as well as `build_verlet_lists_and_calc_verlet_ia` take up a greater part of the runtime, than in the Lennard-Jones simulation. It can also be expected, that as in the bonded interaction simulations the

`add_non_bonded_pair_force()` function takes up more runtime than in the Lennard-Jones simulation.

A callgraph generated by Valgrind confirms these Theories, the `calculate_verlet_ia` and `build_verlet_lists_and_calc_verlet_ia` take up over 80 percent of the total simulation runtime, as opposed to the Lennard-Jones simulation (Figure 5.6), where well under 40 percent of the runtime was used by both of these functions. With around 4 percent of the runtime the `add_bonded_force` function uses notably more runtime as in the Lennard-Jones simulation.

8.2.3 Verlet List Management in Agglomeration Simulations

The callgraph also shows, that there is yet another function, `resize_verlet_list` which takes up almost 9 percent of the runtime. Since during the simulation slowly all the particles lump together the sizes of the verlet pair lists increase drastically. Spending around 10 percent of the simulations time just resizing verlet lists however seems unnecessarily much time spent in a function which is only used to help the rest of ESPResSo to run simulations in a more performant way by using the verlet feature.

Two possible solutions could help speed up the simulation time in future work, however due to limitations in time these could not be explored further in this work:

1. Using the Linked Cell Feature instead of the Verlet Lists Feature

   Agglomeration scenarios, especially those using immutable Dynamic Bonds start with particles randomly placed in the simulation box and usually end up with these particles lumped together if the simulation is run for a period of enough MD time steps. When using the verlet pair list Feature normal ESPResSo simulations are faster, since the particle interactions only have to be calculated for particles within the cutoff radius instead of with all particles in the own cell as well as the neighbour cells. At some point however all particles of the system will have lumped together into a single agglomerate. It is uncertain whether the overhead of creating and maintaining verlet pair lists is higher than the overhead of iterating over all particles in the surrounding cells on every time step. This could be examined in future work.

2. Modifying the resize process for the verlet pair lists

   Using precompiler definitions ESPResSo could be altered to use a different resize process when using the dynamic bonding Feature: Instead of resizing the verlet pair lists the usual way, the lists size could be increased by a greater amount of new elements, which would probably result in fewer executions of

the `resize_verlet_list` function. The amount of particles from the own and the neighbour cells could be used to anticipate the amount of elements to be added to the verlet pair lists. To prevent the allocation of unnecessary memory, the resize could be based on the total amount of particles in the system. This way ESPResSo would maximally allocate enough Resources to fit in an agglomerate of all particles in the system, which is what would happen anyway, given enough MD timesteps were taken in the simulation. The impact of a modified version of such a resize function could also be examined in future work.

8.3 Timers

All simulations using the Timer Feature were run on the BEE Cluster of the Istitute for Computational Physics (ICP) at the University Stuttgart. The Cluster consists of 768 compute cores interconnected via 40 Gbps Infiniband. Three different types of nodes are available:

- 42 standard Nodes

  These nodes each consist of a Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 1x Nvidia Titan Black (6GB GDDR5)

- 2 big nodes

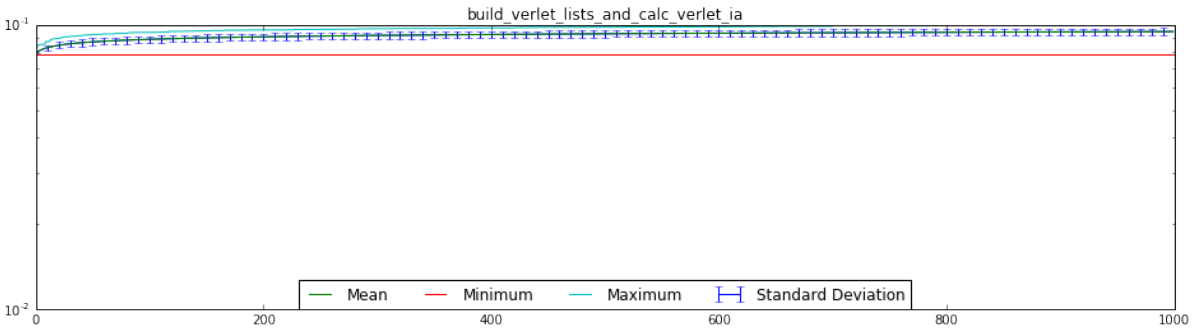
  These nodes consist of Intel Quad Xeon E5-4640v2, 4x10 cores at 2.2 GHz, 512 GB RAM

- 1 multi GPU node

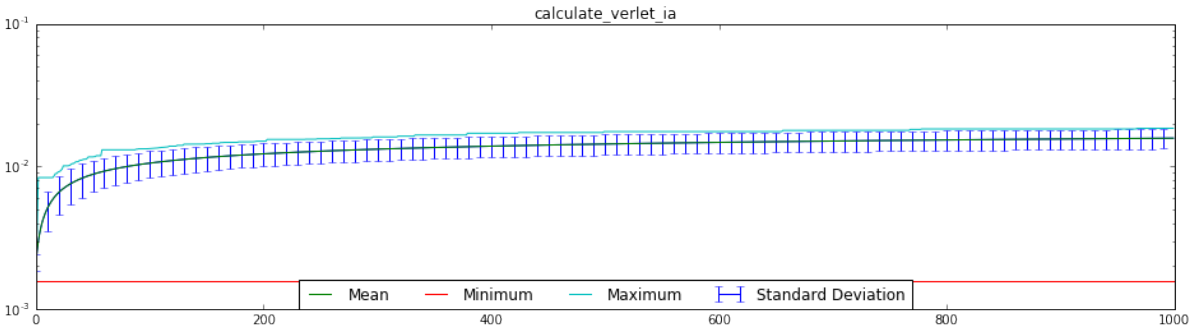
  This node consists of Intel Dual Xeon E5-2630v3, 2x8 cores at 2.4 GHz, 128 GB RAM, 4x Nvidia Tesla K20C (5GB GDDR5)

The ESPResSo Version used was ESPResSo3.2.03836g4088074git, openmpi Version 1.10.1 is used and the Operating System on the BEE Cluster is CentOS release 6.7, running the 2.6 Linux Kernel. All simulations use systems with a density of 0.7 and box sizes according to the amount of particles used in the simulations.

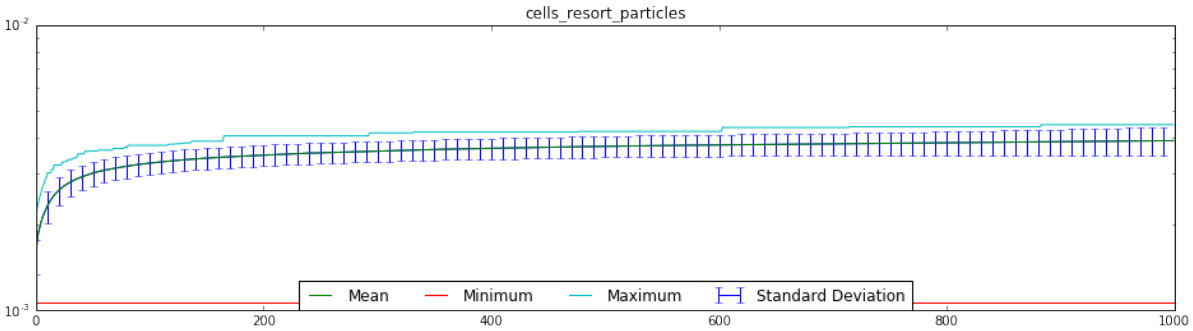
For the Agglomeration scenario a simulation is run on one core of the BEE cluster featuring 1000 particles. The simulation is run for 1000 MD time steps and every time step is measured. It can be observed that the verlet list calculation (`calculate_verlet_ia` and `build_verlet_lists_and_calc_verlet_ia`) takes up increasingly much time and then stabilizes as the MD timesteps pass. The rebuild of the verlet pair lists



(a)



(b)



(c)

Figure 8.2: These graphs represent the time spent in various functions (time on the y-axis) over the course of MD time steps (on the x-axis): a) represents `build_verlet_lists_and_calc_verlet_ia` which is used to generate new verlet pair lists. Then mere calculation of these lists (`calculate_verlet_ia` b)) takes up less time. It can be noted, that the tim spent in the ghost communication (`cells_resort_particles`) is also definitely increasing as the MD timesteps pass along.

calculate_verlet_ia		
Processes:	1000 particles	10000 particles
1	0.0020507 s	0.0157722 s
2	0.00151643 s	0.00829174 s
4	0.00198069 s	0.00542875 s
8	0.00651882 s	0.00490315 s
16	0.00291336 s	0.00545421 s

Table 8.1: This table shows the mean runtime measured for the function `calculate_verlet_ia` in simulations with 1000 or 10000 particles and 1,2,4,8 or 16 parallel ESPResSo core processes involved. It can be observed, that in the 1000 particle simulations the mean runtime of the function first decreases for simulations, where 4 parallel processes are in use and then increases again for 8 or 16 process simulations. For the 10000 particle simulations the time constantly decreases, with exeption to the 16 process simulation. This indicates that the improvement in speed per process is counteracted by the ghost communication , when there are too many processes involved.

(`build_verlet_lists_and_calc_verlet_ia`) takes up much more time as their calculation (`calculate_verlet_ia`). Also the `cells_resort_particles` function, takes up more time throughout the whole of the simulation (which is indicated by the increase of the maximum); as the particles in the simulation lump together, more particles have to be communicated to other cells.

Another set of simulations is run on the BEE cluster: The simulations are all run on one Node of the cluster and simulate an Agglomeration scenario with 1000 or 10000 particles using 1, 2, 4, 8 and 16 ESPResSo core processes. Tables Table 8.1, Table 8.2 as well as Table 8.3 show the results of these simulations.

`calculate_verlet_ia` Table 8.1 with 1000 particles decreases in time up when adding a second or a fourth process to the simulation, however the function spikes, when being executed in a simulation which is using 8 processors. Using 16 processes uses up still more time than simulating with 1,2 or 4 processes (about half of the time spent when using 8 processes). For simulations with 10000 particles the same behavoir can be observed. Here however the simulation using 16 processes takes up about as much time as the best simulation using 4 processes. The same behavoir can be observed for the `build_verlet_lists_and_calc_verlet_ia` Table 8.2, with the difference, that the simulations using 16 processes actually take more even time than the simulations using 8 processes. This could be due to the fact, that it takes more resources to build verlet pair lists when there are more domains between which the particle informa-

build_verlet_lists_and_calc_verlet_ia		
Processes:	1000 particles	10000 particles
1	0.0777099 s	0.0945724 s
2	0.0753636 s	0.083682 s
4	0.0764665 s	0.0792694 s
8	0.0931963 s	0.081157 s
16	0.10022 s	0.103319 s

Table 8.2: This table shows the mean runtime measured for the function `build_verlet_lists_and_calc_verlet_ia` in simulations with 1000 or 10000 particles and 1,2,4,8 or 16 parallel ESPResSo core processes involved. It can be observed, that the mean runtime of the function increases (other than Table 8.1). This indicates, that the rebuild of the verlet pair lists is influenced greater than just their calculation through the use of additional processes.

resize_verlet_list		
Processes:	1000 particles	10000 particles
1	1.00021e-06 s	1.00022e-06 s
2	1.00009e-06 s	1.00054e-06 s
4	1.00325e-06 s	1.00082e-06 s
8	1.00221e-06 s	1.00112e-06 s
16	1.00075e-06 s	1.00207e-06 s

Table 8.3: This table shows the mean runtime measured for the function `resize_verlet_list` in simulations with 1000 or 10000 particles and 1,2,4,8 or 16 parallel ESPResSo core processes involved. It can be observed, that the mean runtime of this function constantly increases in 10000 particle simulations which use more parallel processes. This might indicate that the resize process of verlet lists becomes more complicated depending on the amount of processes used.

tion have to be communicated. Apart from building the verlet lists, the functions `build_verlet_lists_and_calc_verlet_ia` and `calculate_verlet_ia` have the same functionality. The `resize_verlet_list` function Table 8.3 increases also in simulations, with more processes involved. This could indicate that the resize process is more complicated when there are potentially more particles to be communicated to other domains.

9 Bottlenecks in ESPResSo

In this chapter the bottlenecks found during this work are presented and discussed.

9.1 Search for bottlenecks

In the last chapters ESPResSo has been examined concerning its performance: first Valgrind as well as Perf are used to examine the execution of simulation scripts, Before that, some more important parts of the ESPResSo cores code have been explained in detail in chapter 3. Also the Timer Feature has been added to ESPResSo; this is explained further in chapter 4. However another focus of this work is not only to understand ESPResSo and examine the way certain commonly used simulation scenarios perform, but also to search for, to identify and discuss and possibly fix bottlenecks in ESPResSo.

Although a lot of what was done in this work has been related to the measurement of performance within ESPResSo most of the bottlenecks uncovered were actually not found by using the Timer Feature, which was added to ESPResSo, the same goes for the use of the profiling tools used. On the contrary, looking at how ESPResSo works as well as discussing this with the developers been more successfull. This is not meant as a try to revoke the usefullness of the Timers Feature or the outstanding profilers, but merely represents a common theme in software engineering: no matter how good the tools at hand are, when trying to increase the performance of a program it is often that small changes of the sort, which a profiler can't detect have a far greater impact on the programs performance than the ones which can be searched for with such a tool. A lot of time could have been spent analyzing and tweaking the code (e.g. every single `for`-loop in the code) to create optimal cache lines using already existing profiling tools, but since this work was not written from the perspective of an already active ESPResSo contributor it was decided to go for another more abstract approach: To first understand how ESPResSo works[1] and then to find a way to help improve the projects performance in the future. A part of this work was the addition of the Timer feature. What was not

[1] ESPResSo is a project which has been around for several years and thus has grown into a relatively huge and in some parts ancient code base

done however is to analyze greater sections of the code to improve their performance, this is a task that — if undertaken — should be done be more experienced contributors, since a lot of the code sections in ESPResSo have rather complex relations to other parts of the projects source.

9.2 Impact of Timers

As already stated in the previous section the Timers Feature was added to ESPResSo not with the intention to find bottlenecks, but also to help improve future performance of the project: one of the main advantages of Timers in the codebase as opposed to using profilers is that the Timers — same as e.g. sampling profilers — run without having too much of an impact on the executions runtime, but — other than sampling profilers — have the advantage of delivering more accurate and comparable data. This enables users of this feature to test different versions of their code for performance by simply running a simulation which executes the code in question. Timers also relate directly to the real world applications of ESPResSo since they are run on the same Hardware, contrary to profilers which execute the code on a simulated CPU, which not necessarily has to behave the same way as their real counterpart.

9.3 Bottlenecks found

This section lists as well as presents the bottlenecks which were found during this work. It then discusses additionally why these bottlenecks occur and whether these bottlenecks can be approached or prevented in the future. It then discusses, how such approaches may be possible.

9.3.1 The Lack of Software Engineering Techniques

The `add_bonded_force` function is used in ESPResSo to calculate the forces of bonded particles. During the examination of the Lennard-Jones scenario in chapter 5 it is discovered that this function when executed takes around 2 percent of global runtime. This is altough the scenario does not include any kind of bounded particle interaction.

Further examination shows, that this function is called in every single integration step[2].

The reason why this function has this much impact on the programs runtime is due to the way its control flow is handled: The function is called and immediately starts calculating the bonded interactions for every particle in the system by first creating a number of temporary variables and then entering a loop running through all particles in the bonds list of the current particle for which the bonded interaction is being calculated. It does not check whether the particle actually has any bonded interactions before allocating any resources. Adding a simple check to the start of this function which prevents the functions execution greatly reduced the percentage of global runtime the function used up.

In HPC this way of programming is not uncommon, most of the time the code is written with maximizing performance in mind (for example instead of calculating the square root in a lot of cases the calculation of the sqsuare is sufficient enough to solve the problem). The problem with this way of writing code is that instead of creating a solution which works for all use cases — what is how programming is handled often, when problems are not performance relevant — most of the time programmers in HPC approach every single use case on its own.

In this situation the code was probably written with scenarios in mind where bonded interactions are an inherent part of the simulation. From this perspective the use of a check for the need of such an interaction would actually cost performance and is thus not considered useful. Simulations, which do not use bonded interactions were probably simply not considered when the code was written.

Dealing with a complicated code base of a HPC program introduces a new set of problems. Since having a complex code base is essential, some of the usual rules of software engineering (such as e.g. the KISS principle[Mis04]) do not apply at the same time a development process defined by such rules (e.g. Analysis Modeling[Pre05]) can help prevent these kinds of mistakes.

ESPResSo's code base probably consists of some sections of code which have been written with some missing constraints or unconsidered implications to the rest of the program. This could be prevented more precise analysis of the problem at hand and a more formal way to highlight all the constraints a section of code has on the rest of the code base. ESPResSo itself is, due to the way it is configured using precompiler definitions, definitely able make use of multiple implementations of the same code section, since it is compiled explicitly for every simulation scenario anyway.

[2] This is becase ESPResSo supports simulation scenarios where bonded interactions may during simulation time and not from the simulations start.

9.3.2 MPI

ESPResSo uses MPI for parallelisation. As briefly pointed out in chapter 5 the way MPI is configured can have a great influence on how performant ESPResSo works. This is not necessarily an ESPResSo bottleneck, however it is worth mentioning, since the MPI Configuration can greatly improve or worsen the performance of simulations. Usually it should suffice to configure MPI once on the simulation environment[3], however according to literature on the subject ideally every simulation is run in an environment where MPI is configured especially to suit the simulations needs [JWH14; ZYS10].

A lot of ESPResSo's users are not experts on the subject of system administration and don't have deep knowledge when it comes to configuring MPI. A way to improve this would be to extend the projects documentation on the subject of how an environment for running ESPResSo can be set up in a performant way.

9.3.3 Random Number Generators

During this work the hot zones of ESPResSo have been identified and examined. One of the time consuming parts of any ESPResSo simulation is the use of Random Number Generators during the Force Calculation. Currently the Mersenne Twister Engine is used in ESPResSo. The time spent using a Random Number Generator can of course not be drastically reduced by changing ESPResSo code, however not all simulations require the high quality, which the Mersenne Twister currently delivers and there exist a great number of Random Number Generators which perform a lot faster but deliver randomness of lesser quality.

To enable users to speed up their simulations if these don't require the use of the Mersenne Twister it is planned to extend the amount of available Generators and to add options to configure ESPResSo to use either of them. This way a lot of users will be able to significantly reduce the runtimes of their simulations.

9.3.4 The Use of ESPResSo's Interface

ESPResSo has a very broad user base, which includes people from a lot of different areas such as physics, computer science or from the various topics of engineering. While improvements of ESPResSos internal performance affect all of these groups, there is another part of the project which can devastatingly decrease the performance of

[3] e.g. such that multiple processes of the simulation are if possible running on the same NUMA Domain

simulations, the ESPResSo User Interface: Since simulations are set up using a scripting language (TCL or Python) there are two kinds of issues a user might run into. The first one is the lack of programming experience, which may cause users (e.g. from engineering backgrounds) to set up simulations in such a way that they simply run slower. This does not really have much to do with the ESPResSo project itself. The second issue however is the lack of knowledge regarding ESPResSo's way of functioning as well as its set of options for tuning simulations. Without knowing that ESPResSo uses verlet lists to caculate forces one might not think of deleting or excluding particles from simulations whenever these become unnecessary, which has a notable influence on simulation times. And in a similiar without knowing ESPResSos set of options, the use of a fixed value for the verlet lists skin might not improve the simulations runtime the same way the automated `tune skin` option would.

To improve both of these issues, the potential lack of programming experience as well as the lack of knowledge on ESPResSo, it would probably be beneficial to the project to add some examples to the projects documentation, which deal with the topic of setting up simulation scripts to be more performant and which explain how ESPResSo's works internally and how this relates to the use of the simulations script interface.

9.3.5 Pinned Particles

ESPResSo's simulations calculate paticles which have various constraints in a simulation box over time. chapter 3 explains how ESPResSo works internally. If the verlet Feature is turned on during a simulation, verlet pair lists are created for every particle in the System. These Lists contain the particle pairs for which interactions have to be calculated during the integration step. ESPResSo supports the feature of pinning particles in the simulation box. Whenever a particle is pinned in a simulation, its position in the box can not change, but other than that the particle still interacts with the rest of the system. Particle pinning can for example be used to generate fixed structures like walls using a number of particles in the simulation box. During the integration ESPResSo generates the verlet pair lists which contain the interactions of all particle pairs in the system. These lists are then used to calculate the new particle forces and applied to all particles. Pinned particles are currently not considered a special case by ESPResSo, which leads to potentially huge verlet lists (e.g. every particle of a wall consisting of particles interacts with all particles within the cutoff radius) and to a lot of unnecessary interactions being calculated. This can be fixed by implementing a check for whether a particle is pinned before creating the verlet pair lists, skipping the inclusion of unnecessary particle pairs.

9.3.6 Agglomeration

Dynamic bonding is explained in chapter 8 and itself represents a challenge to the performance of simulations. ESPResSo works by assigning a domain, each containing multiple cells to every process involved at the beginning of the simulation. When simulating an Agglomeration scenario, over time the particles lump together resulting in cells containing the majority of particles and other cells containing barely any particles. This results in a great part of the simulations being calculated on only a few processes while having other processes at hand which barely have to do anything. To solve these issues currently concepts for load balancing the cells containing the majority of the particles is developed.

9.3.7 Data structures in ESPResSo

Basic data structures of ESPResSo, particles as well as particle lists are stored as structs. In case of the particle lists these structs contain arrays of particles, resulting in ESPResSo using arrays of structs. Such data structures are not necessarily the most performant way to process these information during simulations. ESPResSo was originally written in C before it was rewritten as a C++ program and since the code was rewritten over time rather than redesigned from scratch with the new base language in mind some of the features of C++ didn't make it into the code base. Unfortionately zero overhead abstraction[Str12], which is one of the ideals of C++ was not included in the rewritten ESPResSo code base. Changing the way basic data structures work would result in a performance gain, however it would also result in a lot of ESPResSo's code having to be rewritten, which is at this point in time rather unlikely.

9.3.8 Warmup phase

During the warmup phase of simulations the system has to be equilibrized. This currently works by using a force cap, which is taken into consideration every single time a interaction is calculated. Instead it would be cheaper to caculate all interactions and at the end check whether the particle forces are greater than the force cap, in which case they would be set to the maximal allowed value.

9.3.9 Verlet list creation

Verlet pair lists currently consist of pointers to particles. If instead the necessary data from the particles would be copied into the verlet lists, this would lead to better memory locality and could be used for a better vectorization.

9.3.10 Optimizing ESPResSos code base

ESPResSo makes use of a lot of different algorithms, which have not been optimized for performance. Auditing these already implemented Features and redesigning them would however mean to spend a lot of resources on already working features instead of adding new ones to the code base and extending the feature set of the software. In theory reworking the code is of course possible, but currently there are no plans to do so in the near future.

9.3.11 The lack of metrics

Lastly over the course of ESPResSo's development there has not yet been any way of assigning any kind of value to the performance of ESPResSo's features. Thus in the development process it was sufficient to create correctly working code. With the addition of the Timer feature ESPResSo simulations can now be natively and without the use of much overhead be monitored for their performance. It is planned to include automated performance tests using the Timers into the continous integration process so that users as well as developers are able to examine and compare the performance of various features of ESPResSo over time.

10 Summary and Outlook

In thesis performance analysis is briefly introduced, and some of the commonly used tools are examined for the use in this work. Then the approach of profiling ESPResSo in this work is introduced. ESPResSo as project has been introduced and it is explained how simulations in ESPResSo work. A a new ESPResSo Feature, ESPResSo Timers, has been developed during this work in order to be able to create comparable performance data of simulations natively within ESPResSo and without the help of any other tools. Four simulation scenarios which are commonly used with the program have been examined: Lennard-Jones interaction, Bonded interaction, Electrostatic interaction and Agglomeration using dynamic bonding. Finally the bottlenecks uncovered during this work have been introduced and discussed.

10.1 Summary

Performance Analysis

The unique role of Software Engineering in High Performance Computing has been pointed out. Also several tools commonly used for performance profiling have been discussed with regarding their use with ESPResSo; Dprof, Gprof, Oprofile, Google Benchmark as well as LibSciBench have been dismissed for various valid reasons. Valgrind and Perf have been chosen to be used during the rest of this work in order to further examine ESPResSo simulations. Additionally it was decided to develop the ESPResSo Timers Feature, in order to be able to measure runtimes of internal ESPResSo functions.

Analyzing ESPResSo

The ESPResSo Project has been introduced. The User Interfaces have been explained, also the four common simulation scenarios have been addressed. ESPResSo's Development Process has been explained and then the basics of Molecular Simulation programs have been presented. Then ESPResSo's implementation has been illuminated, from its configuration, over the way it stores particles and particle lists as structs respectively

arrays of structs up to further implementation details such as the way linked cells and verlet lists are used up to how the ghost communication works. Also the impact of Random Number Generators on the programs runtime has been adressed.

The Timers Feature

The way profilers work as well as the necessity of implementing Timers due to other profilers not being able to provide comparable data has been adressed. Then a specification for the Timers Feature has been presented, followed by an illustration on how the Timers Feature works; The new RunningAverage Class in the code calculates Timers iteratively and is controlled by the Timer class. This Class can be used by the TCL Interface either to return information from the Timers measured or to reset all timers used in the simulation.

Lennard-Jones Interaction

After explaining the the Lennard-Jones Potential, ESPResSo's implementation of the Lennard-Jones Interaction has been examined using Valgrind to generate callgraphs and Perf to generate Reports, Flamegraphs and Timecharts. A designflaw in the `add_bonded_force` function, which caused it to allocate Resources even in simulations which don't use bonded Forces has been discovered and fixed in the code. Then The Timers Feature is used to further examine the Lennard-Jones Simulation by looking closer at the way ESPResSo performs when simulating multiple processes. It has been found that ESPResSo simulations using too much parallel processes might slow down in comparison to others using fewer processes. This probably depends both on the size of the simulated system as well as on how the inter-process communication slows down the simulation process. In another set of simulations it has been observed, that the function, which is calculating the verlet pair lists interactions slows down over time and a potential reason for this behavoir might have been a part of ESPResSo's ghost communication.

Electrostatic Interaction

The P3M interaction is explained and the way it has been implemented in ESPResSo has been discussed shortly. Then this Interaction has been compared to the already familiar Lennard-Jones interaction and afterwards analyzed using Valgrind. A set of simulations has been analyzed using the Timers Feature and it was observed, that adding more processes to the simulations resulsts in a noticable decrease in simulation time, which

is due to the fact that the P3M Grid operations are calculated much faster by multiple processes.

Bonded Interaction

The FENE interaction was shortly introduced and using Valgrind a callgraph for a bonded interaction simulation has been created. It was observed, that this callgraph was identical to the callgraph of the Lennard-Jones interaction, with the exeption of the amount of runtime the simulation spent in the `add_bonded_force` function. Timers were used to analyze the performance of bonded interactions and it was found that performance-wise bonded simulations behave similiar to simulations of Lennard-Jones Interactions.

Dynamic Bonding

Agglomeration using dynamic bonds was introduced and compared to the way Bonded interaction Simulations work in ESPResSo. A Agglomeration Callgraph was created using Valgrind and it was observed that during Agglomeration simulations using dynamic bonding both of the functions which calculate the verlet interactions take up most of the runtime. It was also noted that a huge part of the runtime is spent resizing the verlet pair lists. This could eventually be a bottleneck which could be solved by reszing the verlet pair lists in greater steps, resulting in fewer resze operations or through the use of the Linked Cells Feature during simulations. Simulations using the Timers Feature showed, that three of ESPResSo's functions showed an increase in their mean runtime over the course of MD Timesteps. This might be caused by the slower rebuild of the verlet lists as well as by the ghost communication which over time might have to communicate a great amount of data between processes.

Bottlenecks

The way bottlenecks were searched and found during this work has been explained and the impact of the Timers feature on this work as well as the development process of ESPResSo has been explained. Then the bottlenecks found in this work have been presented. The lack of Software Engineering techniques as well as the way that ESPResSo is configured using preprocessor definitions leads to the problem that every single use case has to be considered when writing code for ESPResSo. MPI has to be configured in a proper way for ESPResSo to work at its peak performance, the projects documentation however does not provide the user with extensive information on how to accomplish

this task. Currently only one Random Number Generator is used in the code base, in the future however it is planned to provide the users with the option to choose between different Random Number Generators, which perform their tasks at different speeds and qualities. The ESPResSo user interfaces are a possible source for bottlenecks; since simulations are controlled by the interface, the user is able to create simulations which perform really bad, when it comes to their efficiency. ESPResSo could provide a best practices guide to its userbase, where common ways to set up simulations in a performant way are explained. When Particles in ESPResSo are pinned to form structures, they can not move any more but these particles still are included into verlet pair lists and thus the interactions of these particles is calculated in every time step of the simulations. This could be fixed by a simple check in the code. Agglomeration, when run for enough MD time steps leads to slow unresponsive simulations, because all particles are lumped together and in the worst case calculated on only one of many processes. ESPResSo does not make use of C++ zero overhead abstractions on its very basic datastructures. The Force capping during the warmup phase is calculated into every single calculation instead of calculating all of the force and than capping it. During the verlet list generation pointers in stead of the actual variables are copied into the lists. This could increase memory locality and potentially benefit the applications runtime. A lot of ESPResSo's Algorithms is not optimized for performance and could be audited. Lastly up until now there was no Timers feature which enabled the users, the developers or the projects Continous Integration to monitor ESPResSo's performance.

10.2 Outlook

The addition of the Timers Feature, as well as the measurements in this work and the bottlenecks found in the code open up quite a few opportunities for future work. For example ESPResSo's performance could be tested for larger scale simulations and the results could then be used to optimize the simulation environments or assign simulations with a specific amount of parallel processes to optimize the simulations runtime. The bottlenecks presented could be examined further and fixed. The development process of ESPResSo could be reworked to implement more formalities in order to prevent sloppy mistakes.

A follow up project to this work is the implementation of automated performance tests into the Continuos Integration system of ESPResSo so that the performance of the software over time can be monitored.

Bibliography

[ALK+13] A. Arnold, O. Lenz, S. Kesselheim, R. Weeber, F. Fahrenberger, D. Roehm, P. Košovan, C. Holm. "Espresso 3.1: Molecular dynamics software for coarse-grained models." In: *Meshfree methods for partial differential equations VI*. Springer, 2013, pp. 1–23 (cit. on p. 77).

[BAH87] R. B. Bird, R. C. Armstrong, O. Hassager. "Dynamics of polymeric liquids. Vol. 1: Fluid mechanics." In: (1987) (cit. on p. 73).

[BB74] E. O. Brigham, E. O. Brigham. *The fast Fourier transform*. Vol. 7. Prentice-Hall Englewood Cliffs, NJ, 1974 (cit. on p. 67).

[BCC+08] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, M. V. Zelkowitz. "Understanding the high-performance-computing community: A software engineer's perspective." In: *IEEE software* 25.4 (2008), p. 29 (cit. on p. 15).

[BLD98] J. Beckers, C. Lowe, S. De Leeuw. "An iterative PPPM method for simulating Coulombic systems on distributed memory parallel computers." In: *Molecular Simulation* 20.6 (1998), pp. 369–383 (cit. on p. 67).

[BWZ15] L. Bass, I. Weber, L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015 (cit. on pp. 21, 60).

[Cou91] H. Couchman. "Mesh-refined P3M-A fast adaptive N-body algorithm." In: *The Astrophysical Journal* 368 (1991), pp. L23–L26 (cit. on p. 20).

[Dem11] V. Demchik. "Pseudo-random number generators for Monte Carlo simulations on ATI Graphics Processing Units." In: *Computer Physics Communications* 182.3 (2011), pp. 692–705 (cit. on p. 33).

[dev] E. developers. *autogenerated doxygen documentation* (cit. on p. 49).

[Era08] S. Eranian. "What can performance counters do for memory subsystem analysis?" In: *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. ACM. 2008, pp. 26–30 (cit. on p. 16).

[FDK+11] C. Feichtinger, S. Donath, H. Köstler, J. Götz, U. Rüde. "WaLBerla: HPC software design for computational engineering simulations." In: *Journal of Computational Science* 2.2 (2011), pp. 105–112 (cit. on p. 15).

[GKM82] S. L. Graham, P. B. Kessler, M. K. Mckusick. "Gprof: A call graph execution profiler." In: *ACM Sigplan Notices*. Vol. 17. 6. ACM. 1982, pp. 120–126 (cit. on p. 16).

[GLDS96] W. Gropp, E. Lusk, N. Doss, A. Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard." In: *Parallel computing* 22.6 (1996), pp. 789–828 (cit. on p. 28).

[Gre16] B. Gregg. "The flame graph." In: *Communications of the ACM* 59.6 (2016), pp. 48–57 (cit. on p. 51).

[HB15] T. Hoefler, R. Belli. "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. 2015, p. 73 (cit. on p. 17).

[Htt12] M. Httermann. *DevOps for developers*. Apress, 2012 (cit. on p. 60).

[HW10] G. Hager, G. Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010 (cit. on p. 55).

[IAKW14] G. Inci, A. Arnold, A. Kronenburg, R. Weeber. "Modeling Nanoparticle Agglomeration using Local Interactions." In: *Aerosol Science and Technology* 48.8 (2014), pp. 842–852. DOI: 10.1080/02786826.2014.932942. eprint: http://dx.doi.org/10.1080/02786826.2014.932942. URL: http://dx.doi.org/10.1080/02786826.2014.932942 (cit. on p. 77).

[JWH14] S. Jarvis, S. Wright, S. D. Hammond. *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation: 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers*. Vol. 8551. Springer, 2014 (cit. on pp. 55, 88).

[LE04] J. Levon, P. Elie. *Oprofile: A system profiler for linux*. 2004 (cit. on p. 16).

[LG96] B. A. Luty, W. F. van Gunsteren. "Calculating electrostatic interactions using the particle-particle particle-mesh method with nonperiodic long-range interactions." In: *The Journal of Physical Chemistry* 100.7 (1996), pp. 2581–2587 (cit. on p. 67).

[Mis04] R. B. Misra. "Global IT outsourcing: Metrics for success of all parties." In: *Journal of Information Technology Case and Application Research* 6.3 (2004), pp. 21–34 (cit. on p. 87).

[MN98] M. Matsumoto, T. Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.” In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30 (cit. on p. 49).

[NS03] N. Nethercote, J. Seward. “Valgrind: A program supervision framework.” In: *Electronic notes in theoretical computer science* 89.2 (2003), pp. 44–66 (cit. on p. 16).

[NS07] N. Nethercote, J. Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation.” In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100 (cit. on p. 16).

[PG96] E. Pollock, J. Glosli. “Comments on P 3 M, FMM, and the Ewald method for large periodic Coulombic systems.” In: *Computer Physics Communications* 95.2 (1996), pp. 93–110 (cit. on p. 67).

[Pre05] R. S. Pressman. *Software engineering: a practitioner’s approach*. Palgrave Macmillan, 2005 (cit. on p. 87).

[PZM10] A. Pesterev, N. Zeldovich, R. T. Morris. “Locating cache performance bottlenecks using data profiling.” In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 335–348 (cit. on p. 16).

[SABW82] W. C. Swope, H. C. Andersen, P. H. Berens, K. R. Wilson. “A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters.” In: *The Journal of Chemical Physics* 76.1 (1982), pp. 637–649 (cit. on pp. 24, 27).

[Smi01] B. Smit. *Understanding molecular simulation*. Academic Press, 2001 (cit. on p. 21).

[Str11] R. Strzodka. “Abstraction for AoS and SoA Layout in C+.” In: *GPU computing gems Jade edition* (2011), p. 429 (cit. on p. 25).

[Str12] B. Stroustrup. “Foundations of C++.” In: *European Symposium on Programming*. Springer. 2012, pp. 1–25 (cit. on p. 90).

[WOB91] L. E. Wedgewood, D. N. Ostrov, R. B. Bird. “A finitely extensible bead-spring chain model for dilute polymer solutions.” In: *Journal of non-newtonian fluid mechanics* 40.1 (1991), pp. 119–139 (cit. on p. 73).

[ZYS10] C. Zhang, X. Yuan, A. Srinivasan. “Processor affinity and MPI performance on SMP-CMP clusters.” In: *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–8 (cit. on pp. 55, 88).

All links were last followed on October 17, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature