UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

MEMORY OPTIMIZATION AND PERFORMANCE STUDY OF PROTEIN/DNA-LIGAND INTERACTION SOFTWARE FOR HPC CLUSTERS

RICARD ZARCO BADIA

Thesis supervisor: VICTOR GUALLAR TASIES (Barcelona Supercomputing Center)

Tutor: DANIEL JIMENEZ GONZALEZ (Department of Computer Architecture)

Degree: Master Degree in Innovation and Research in Informatics

Specialisation: High Performance Computing

Thesis report

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

15/05/2023

Abstract

This master's thesis aims to improve the memory requirements and efficiency of the PELE software, developed by BSC, on High-Performance Computing (HPC) systems. The research focuses on analyzing the behavior of PELE on HPC systems to detect areas of improvement and optimize its efficiency. The primary objectives of this thesis are to significantly reduce the memory usage of PELE, replace legacy MPI communication models, and port the software to an ARM-based architecture.

The first objective of the thesis is to investigate the current memory usage of PELE and identify the areas where it can be reduced. This study will involve analyzing the code and profiling the performance of PELE on HPC systems to identify memory leaks and inefficient data structures. The results will be used to propose modifications to the software's code to reduce its memory footprint.

The second objective is to replace the legacy MPI communication models with more efficient communication models. This study will involve analyzing the communication patterns of PELE and identifying areas where the current models are not optimal. The proposed improvements will include implementing new communication models and optimizing the existing ones to reduce communication overhead and improve the software's scalability. Furthermore, we have made a proposal outside of local PELE changes to improve the global performance of the Adaptive PELE data-flow, which is a common workflow that uses PELE.

Finally, the thesis aims to port the PELE software to an ARM-based architecture. This study will involve analyzing the software's code and identifying any platform-specific dependencies. The proposed modifications will ensure that the software can run on ARM-based HPC systems efficiently.

In conclusion, this master's thesis aims to improve the efficiency of the PELE software on HPC systems by reducing its memory usage, replacing legacy MPI communication models, and porting it to an ARM-based architecture. The research will involve analyzing the software's code, profiling its performance on HPC systems, and proposing modifications to optimize its performance. The proposed improvements will make PELE more efficient and scalable, making it suitable for use in large-scale simulations and scientific research.

Contents

1 Introduction **8**

2 Computational resources **10**

2.1 MareNostrum 4 11

2.2 Nord 3 12

2.3 Huawei Cluster 13

3 Profiling and debugging resources **14**

3.1 Profiling and tracing tools 15

3.1.1 HPC Portal 15

3.1.2 Extrae and Paraver 16

3.1.3 Massif (Valgrind) 17

3.1.4 Callgrind (Valgrind) 18

3.2 Debugging software 21

3.2.1 GDB 21

3.2.2 DDT 21

4 PELE: Preliminary analysis **23**

4.1 General PELE behavior observed 24

4.1.1 Testing environment 24

4.1.2 Pure MPI execution with 5 ranks 24

4.1.3 Execution using multiple OpenMP threads per MPI rank 27

4.1.4 Conclusions 28

4.2 MPI execution flow and Adaptive PELE 30

4.2.1 MPI behavior of standalone PELE 30

4.2.2 Adaptive PELE 31

4.3 Preliminary memory profiling of PELE executions 34

4.3.1 Testing environment and inputs 34

4.3.2 Results 35

4.3.3 Conclusions 37

5 Optimization and parallelization proposals **38**

5.1 Improvements on memory usage 39

5.1.1 Intel MKL vs OpenBLAS 39

5.1.2 Code changes . . . 40
5.1.3 Results . . . 43
5.1.4 Conclusions . . . 45
5.2 Floating point precision changes . . . 46
5.2.1 Changes and problems . . . 46
5.2.2 Results . . . 49
5.2.3 Conclusions . . . 52
5.3 MPI design changes . . . 53
5.3.1 Eliminating unnecessary MPI communications and controllers . . . 53
5.3.2 Mitigating the effects of load imbalance . . . 55
5.3.3 Results . . . 57
5.3.4 Conclusions . . . 59
5.4 Relaxing Monte Carlo simulations . . . 60
5.4.1 Implementation . . . 60
5.4.2 Results . . . 61
5.4.3 Conclusions . . . 62

6 PELE variants analysis on an ARM-based platform 63
6.1 Deployment . . . 63
6.2 Performance analysis . . . 65
6.2.1 Memory and time scalability . . . 65
6.2.2 Single precision floating point . . . 67
6.2.3 Adaptive PELE . . . 68
6.3 Conclusions . . . 71

7 Final conclusions 72

Bibliography 74

A PELE configuration files 76
A.1 PELE's configuration file for small input . . . 76
A.2 PELE's configuration file for large input . . . 80
A.3 Base adaptive configuration file for Adaptive PELE execution . . . 83
A.4 Base PELE configuration file for Adaptive PELE execution . . . 84

B Complimentary plots of PELE's memory analysis 89
B.1 Massif-visualizer memory plots of baseline PELE binaries . . . 89

List of Acronyms

AWS Amazon Web Services

BLAS Basic Linear Algebra Subprograms

BSC Barcelona Supercomputing Center

EAPM Electronic and Atomic Protein Modeling (BSC's research group)

GDB GNU Debugger

GPFS (IBM's) General Parallel File System

HPC High Performance Computing

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

LAPACK Linear Algebra Package

LSF (IBM's) Load Sharing Facility

MC Monte Carlo

MKL (Intel's) Math Kernel Library

NAN Not A Number

PDB Protein Data Bank

PDF Portable Document Format

PELE Protein Energy Landscape Exploration

SLURM (SchedMD's) Simple Linux Utility for Resource Management

List of Figures

3.1 HPC Portal: CPU load of a job's nodes over time 15
3.2 HPC Portal: memory usage of a job's nodes over time 16
3.3 Visualization of a typical execution trace using Paraver 17
3.4 Visualization of the memory allocations of an application using massif-visualizer 18
3.5 Visualization of an application's function call graph using Kcahegrind . . . 19
3.6 Listing of applications contributing to the execution time using Kcahegrind 19
3.7 Example of a typical debugging session on DDT 22

4.1 Paraver visualization of PELE's useful computation and MPI calls 25
4.2 Paraver visualization of PELE's MPI calls with communication lines 25
4.3 MPI call profile of PELE's execution provided by Paraver 26
4.4 Zoomed-in view of the region between PELEsteps of the same trace 27
4.5 Paraver trace of a MPI + OpenMP execution of PELE 28
4.6 Flow diagram of a master MPI rank from a PELE execution 31
4.7 Flow diagram of a worker MPI rank from a PELE execution 31
4.8 Flow diagram of an Adaptive PELE execution, using two epochs and 8 MPI ranks 32
4.9 Baseline memory consumption metrics of serial PELE executions with different parameters 36

5.1 Memory consumption comparative of MKL vs OpenBLAS, small input, with binding energy 40
5.2 Massif snapshot of the peak heap memory usage of PELE, without binding energy 41
5.3 Massif snapshot of the peak heap memory usage of PELE, with binding energy 41
5.4 Memory usage comparison between the original PELE code and our modified version 44
5.5 Serial execution times of memory improved PELE against original binary, large input 44
5.6 Conceptual diagram of the infinite loop while creating a grid 48

5.7 Metrics comparison of single and double precision serial PELE, large input, no binding energy . . . 50
5.8 Callgrind traces of single (left) and double (right) precision serial PELE executions . . . 51
5.9 Execution times of single precision serial PELE binaries, large input, no binding energy . . . 51
5.10 Paraver visualization of useful duration and MPI calls of new PELE, 4 epochs, with barriers . . . 57
5.11 Paraver visualization of useful duration of new PELE, 4 epochs, no barriers 58
5.12 Execution times of Adaptive PELE and new PELE versions with different parameters . . . 59
5.13 Conceptual workflow diagram of the new Relaxed Adaptive PELE implementation . . . 61
5.14 Execution times of Adaptive PELE and new PELE version with different parameters . . . 62

6.1 Node's occupation of original and memory-changed PELE through multiple MPI ranks . . . 65
6.2 Execution time of original and memory-changed PELE through multiple MPI ranks . . . 66
6.3 Execution time of serial PELE with different floating point implementations 67
6.4 Execution times of Adaptive PELE and new PELE versions with different parameters . . . 68
6.5 Paraver visualizations of PELE with and without barriers: 8 MPI ranks, 2 pelesteps, 4 epochs . . . 69
6.6 Execution times of Adaptive PELE and new relaxed Adaptive PELE using multiple parameters . . . 69
6.7 Execution times of new relaxed Adaptive PELE on Nord 3 and Huawei, using multiple parameters . . . 70

B.1 Massif heap trace of a serial PELE execution, small input, without binding energy . . . 89
B.2 Massif total trace of a serial PELE execution, small input, without binding energy . . . 90
B.3 Massif heap trace of a serial PELE execution, small input, with binding energy . . . 90
B.4 Massif total trace of a serial PELE execution, small input, with binding energy . . . 91
B.5 Massif heap trace of a serial PELE execution, large input, without binding energy . . . 91

B.6 Massif total trace of a serial PELE execution, large input, without binding energy 92
B.7 Massif heap trace of a serial PELE execution, large input, with binding energy 92
B.8 Massif total trace of a serial PELE execution, large input, with binding energy 93

List of Tables

4.1 PELE's memory consumption profile of a single Nord3 node 35

5.1 PELE's memory consumption profile of a single Nord3 node after memory improvements 45

1 Introduction

In this master thesis, our work revolves around the analysis and proposed improvements for the PELE software. PELE (Protein Energy Landscape Exploration) is a software package used for molecular simulations of proteins [1]. It is currently maintained by the Electronic and Atomic Protein Modeling research group at BSC and Nostrum Biodiscovery. PELE uses a hybrid methodology that combines Monte Carlo (MC) simulations with molecular dynamics simulations to explore the conformational energy landscape of proteins. The software can be used to study a variety of biomolecular processes, such as protein-ligand binding, protein-protein interactions, and protein folding. In recent years, computational methods have become increasingly important in understanding the behavior of biomolecules at the atomic level, and PELE has emerged as a powerful tool in this area.

PELE works by simulating the movement of atoms and molecules in a protein using a series of algorithms. The software first generates an initial structure of the protein and then applies Monte Carlo simulations to explore the conformational space of the protein. One of the key features of PELE is its ability to simulate protein-ligand binding.

The software can automatically dock ligands to a protein and then simulate the binding process, allowing researchers to study the binding affinity and binding kinetics of different ligands. PELE also includes a range of tools for analyzing the results of simulations, such as calculating the energy of specific interactions between the protein and ligand, identifying key residues involved in binding, and generating conformational ensembles of the protein.

In addition to Monte Carlo simulations, PELE also can use molecular dynamics simulations to refine the structures obtained from the Monte Carlo simulations. Molecular dynamics simulations involve solving the equations of motion for each atom in the protein, allowing the simulation to explore the conformational space in a more detailed and accurate way. The molecular dynamics simulations are used to optimize the energy of the protein and ensure that the resulting structures are physically realistic.

There are several other software programs available for molecular simulations that are commonly used in the field of computational biophysics. Examples include GROMACS, NAMD, AMBER, and CHARMM, among others. These programs use molecular dynamics simulations as their primary approach for studying biomolecular systems. These programs are widely used and have been extensively validated, making them reliable tools for

studying biomolecular systems.

However, PELE stands out from these traditional molecular dynamics simulation programs due to its unique hybrid approach that combines Monte Carlo simulations with molecular dynamics simulations. Monte Carlo simulations are used in PELE to explore the conformational space of proteins in a different way compared to molecular dynamics simulations. Monte Carlo simulations involve randomly selecting new conformations for specific parts of the protein and calculating their energies, allowing for a more rapid exploration of conformational space. This approach can be particularly useful for studying large-scale conformational changes, such as protein folding or large-scale domain movements.

PELE is not the only biochemistry software using Monte Carlo simulations [2]. Other relevant programs for drug design that also use MC simulations are ProtoMS, BOSS, MCPRO and Macromodel, alongside others. Each software specializes in some form of protein or molecular modeling and simulations. PELE in particular is specifically designed for studying protein-ligand binding, which sets it apart from many other simulation programs. The software has an automated docking feature that allows for the efficient placement of ligands in the protein's active site, followed by the simulation of the binding process. PELE also provides various options for analyzing the binding energetics, such as calculating the energy of specific interactions and generating conformational ensembles of the protein-ligand complex.

Generally, PELE is a software that can be run either serially or using the MPI communication paradigm, the latter being normally intended for executions on HPC (High Performance Computing) systems. When executions are performed using MPI, it exploits parallelism by performing simultaneous independent explorations. Since access to HPC systems is usually limited and/or costly, optimizing the utilization of these resources becomes critical. In this work, we will explore some performance bottlenecks for this software and make some improvement proposals for them.

2 Computational resources

Running and profiling PELE in a real-world scenario requires, in general, access to HPC systems. Although PELE can be compiled and executed on a strictly sequential manner with just one exploration and no parallelism of any kind, the analysis of representative workloads may become unfeasible on regular workstations. In order to be able to compare and analyze MPI executions without incurring on memory limitations, we have used three clusters hosted at BSC:

- MareNostrum 4
- Nord 3
- Huawei Cluster

PELE's developer group already provides PELE binaries natively compiled for MareNostrum 4 and Nord 3, but not for Huawei's cluster (or any ARM-based system) at the moment of this work.

The main characteristics of each of the systems are presented in the next section. For more specific details outside the relevancy of this master thesis, the reader can refer to their user guides [3].

2.1 MareNostrum 4

MareNostrum 4 has been BSC's flagship supercomputer since 2017. It's main defining characteristics are the use of Lenovo's SD530 general purpose compute racks, based on Intel Xeon Platinum processors from the Skylake generation. It uses the x86-64 instruction set, which makes it very convenient for the deployment of software. All nodes are interconnected using an Intel Omni-Path high performance network with a full-fat tree topology.

Each rack is comprised of 72 compute nodes, for a grand total of 3456 nodes when operating under optimal conditions. Each node has the following characteristics:

- 2 sockets of Intel Xeon Platinum 8160 CPU with 24 cores each @ 2.10GHz. Total of 48 cores per node.
- L1D cache of 32K; L1I cache of 32K; L2 cache of 1024K; L3 cache of 33792K.
- 96 GB of main memory per node, with 1.880 GB/core. RAM configuration: 12 DIMM, 8GB/DIMMM, 2667Mhz clock frequency.
- 100 Gbit/s Intel Omni-Path network adapter.

The job scheduler used in this system is SchedMD's Slurm, which is widely used on the HPC ecosystem. The majority of this cluster's data storage is implemented using IBM's GPFS file-system, which is shared between all nodes. This file-system is also shared between different HPC systems's at BSC, including Nord 3 and the Huawei cluster.

This is the system that was originally used as the "baseline" for PELE executions and its performance metrics in this project. Due system incompatibilities with the memory profiling software that appeared during the early development of this project, this system was abandoned and replaced with Nord 3 as soon we started with the memory profiling analysis of PELE.

2.2 Nord 3

Nord 3 is the second most popular general purpose supercomputer deployed at BSC. At its core, it's just a scaled-down version of MareNostrum 3, using exactly the same nodes but at a lower scale. Excluding its scale, amount of CPU cores per node and available memory, this system is very similar to MareNostrum 4. This cluster is based on IBM's iDataPlex compute racks using Intel SandyBridge processors, so it also uses the x86-64 instruction set. All nodes are interconnected using Infiniband Mellanox in a full-fat tree topology.

Each rack is comprised of 84 IBM DX360 M4 compute nodes, for a grand total of 756 nodes when operating under optimal conditions. Each node has the following characteristics:

- 2 sockets of Intel E5-2670 SandyBridge-EP processors with 8 cores each @ 2.6GHz. Total of 16 cores per node.
- L1D cache of 32K; L1I cache of 32K; L2 cache of 256K; L3 cache of 20480K.
- 32 GB of main memory per node, with 2 GB/core.
- Infiniband Mellanox FDR10 adapter.

The job scheduler used in this system was originally LSF (IBM's Load Sharing Facility), but the cluster was updated and it switched its scheduler to Slurm (alongside its operative system) . This cluster's data storage is also implemented using the same GPFS file-system. Its basic software stack is very similar to MareNostrum 4's, which makes switching between cluster's fairly simple. Its only drawback is its reduced scale and lower CPU count per node, which makes it less desirable than MareNostrum 4 for sufficiently large workloads. However, it is still very competitive for sequential or embarrassingly parallel tasks that do not require high amounts of cores as long as these workloads don't rely heavily on vectorization instructions.

The majority of the work done in this project has been performed using this cluster.

2.3 Huawei Cluster

The Huawei cluster is another HPC system hosted by BSC. Although not relevant to this project's scope, one of the main characteristics of this cluster is the use of dedicated Huawei-made accelerators for AI training and inference workloads. The other defining characteristic of this cluster is the use of ARM-based processors (ARMv8.1 architecture), concretely Kunpeng 920 CPUs deployed in different node configurations. It uses a Mellanox high-performance network interconnect.

This is a heterogeneous cluster, meaning that its computing nodes are split between three different types with different characteristics. The basic specifications for each block are the following:

- General purpose computing, 16 nodes
  - 2 sockets of Kunpeng 920 CPU, providing 64 cores each @ 2.6 GHz. Total of 128 cores per node.
  - 256 GB of main memory per node, with 2 GB/core.
- AI training, 1 node
  - 4 sockets of Kunpeng 920 CPU, providing 48 cores each @ 2.6 GHz. Total of 192 cores per node.
  - 1 TB of main memory per node, with 5.2 GB/core.
  - 8 Ascend 910A (Huawei accelerators)
- AI inference, 1 node
  - 2 sockets of Kunpeng 920 CPU, providing 64 cores each @ 2.6 GHz. Total of 128 cores per node.
  - 256 GB of main memory per node, with 2 GB/core.
  - 5 Atlas 300C (Huawei accelerators, based on IA Ascend 310 processors)

This cluster also uses the same shared GPFS file-system. Its job scheduler is also Slurm. All work done with this cluster has been performed using the general purpose computing nodes since we only require to check PELE's behavior on an ARM-based system.

3 Profiling and debugging resources

The majority of the work done is based on the analysis of PELE's behavior and metrics. In order to get an insightful look into what is going on under PELE's hood, we have used an array of different tools to inspect PELE's executions at different levels (general resource usage, memory usage, MPI communication patterns...).

These tools can be divided into two different categories:

- **Profiling software:** any tool used to analyze aspects of another software's normal execution in order to profile and quantify different metrics. The specific metrics can be anything: time spent on specific functions, memory usage at differens points of the execution, amount of CPU instructions executed, number of cache misses, etc.
- **Debugging software:** any tool used to execute and analyze software in a controlled manner with the intent of detecting and understanding the causes of unexpected behaviors. These tools let the user know the exact state of the execution at any given point in time.

These tools are usually used in conjunction. Profilers are used to get a primary analysis of the software's behavior and metrics, which at the same time let us identify potential areas of improvement. After that, the analyzed software is modified to optimize those areas.

Due to code size and complexity, these modifications are prone to cause unexpected changes in the way the code runs, either by changing the expected output or by indefinitely hanging the execution. At that point, debugging tools are used to further inspect the causes of those behavioral changes. Once the causes are identified and corrected, another round of profiling can be performed to quantify the benefits and identify new possible improvements. This cycle can be performed as many times as necessary.

We will describe all tools used in the following sections.

3.1 Profiling and tracing tools

3.1.1 HPC Portal

While not exactly a profiling software, HPC Portal [4] is a service provided and developed by the support team at BSC. At its core, it is a job monitoring tool used to keep track of all jobs launched by all users in the Slurm-based clusters hosted by BSC. This tool is deployed as a web portal and is freely accessible to all BSC-hosted cluster's users.

The relevant aspect of HPC Portal for our intended purposes is the fact that it also tracks some historical statistics of the nodes allocated by our job executions. For example, on figures 3.1 and 3.2 we can see the CPU and memory usage of the nodes, respectively, involved in the execution of a specific job. This data is also retrievable as a CSV file containing a time series of these values.

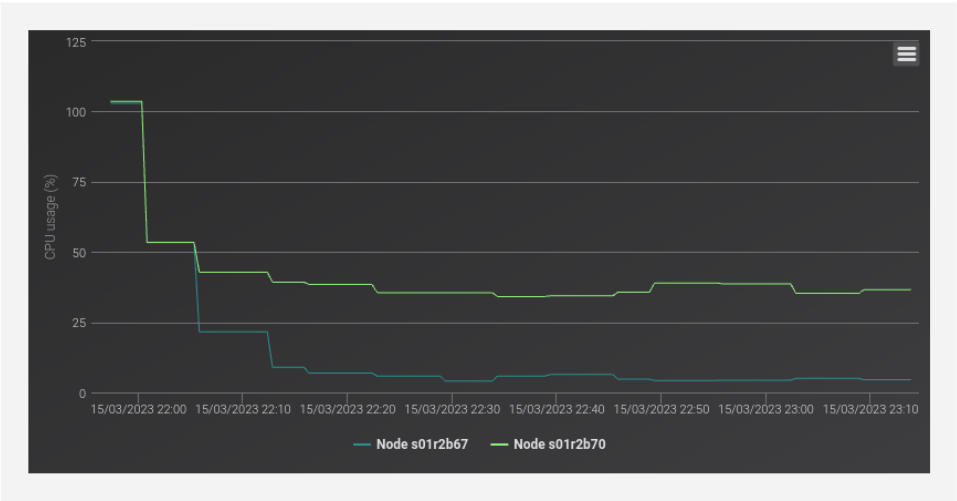


Figure 3.1: HPC Portal: CPU load of a job's nodes over time

The main drawback of HPC Portal is the limited amount of types of metrics, alongside the coarseness of the retrieved data. At this moment, the only metrics reported are CPU load, memory usage and power consumption. These metrics are polled every 30 seconds in a best-effort manner. Another important limitation is the fact that the data reported is at node level, not at job level. This means that it's less useful for sub-node executions that only use a handful of cores, since we can't assume that a node's behavior is the direct consequence of our monitored job. Other jobs could be coexisting at the same time in a given node, which could pollute our readings.

On the other hand, the principal benefit of this tool is its non-obtrusiveness and ease of use. HPC Portal does not require any type of setup or code modifications of any kind.

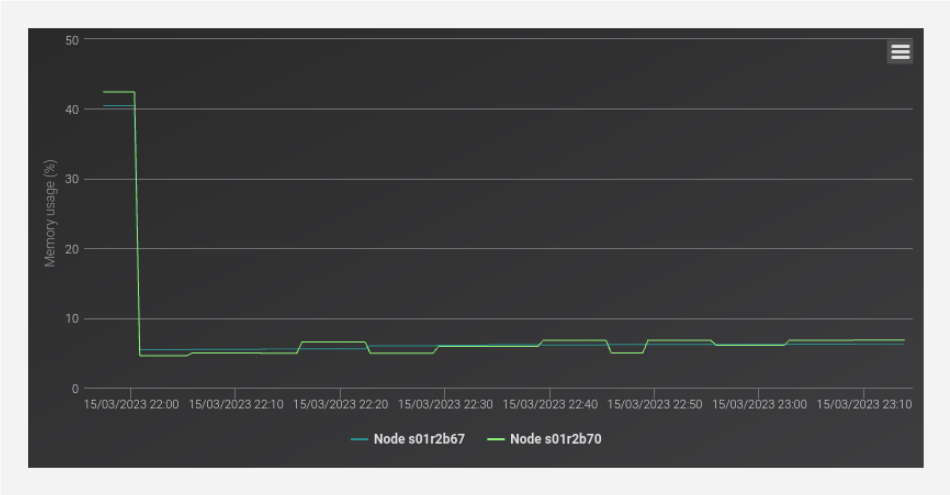


Figure 3.2: HPC Portal: memory usage of a job's nodes over time

It's an external service that is always running and is useful for getting a general idea of how a job execution performed, as long as the job fully allocates nodes.

3.1.2 Extrae and Paraver

Extrae and Paraver [5] are a subset of the performance analysis tools developed and maintained by the Tools team at BSC. These two tools work in tandem: Extrae (as the Spanish name implies) is used at execution time in order to extract metrics and events of a given job, and then Paraver is used to analyze those metrics in a visual manner. These tools are designed primarily for analysing and tracing parallel applications.

Extrae is a lightweight instrumentation tool that allows developers to capture detailed runtime information about an application, such as memory access patterns, MPI communication, and OpenMP parallelism. This information is stored in a trace file, which can be later analyzed using Paraver. Extrae supports a wide range of programming models, including MPI, OpenMP, CUDA, OmpSs, and OpenACC, and can be used on a variety of computing architectures, including multicore systems, clusters, and supercomputers.

Paraver is a visualization and analysis tool that allows users to explore the trace data generated by Extrae. It provides a user-friendly graphical interface that enables users to visualize the behavior of their application over time, identify performance bottlenecks, and understand the interactions between different parts of the application. Paraver includes a variety of built-in visualizations, including timelines, histograms, and scatter plots, and allows users to create custom visualizations using its scripting language. On figure 3.3 we can see an example of a typical Paraver visualization.

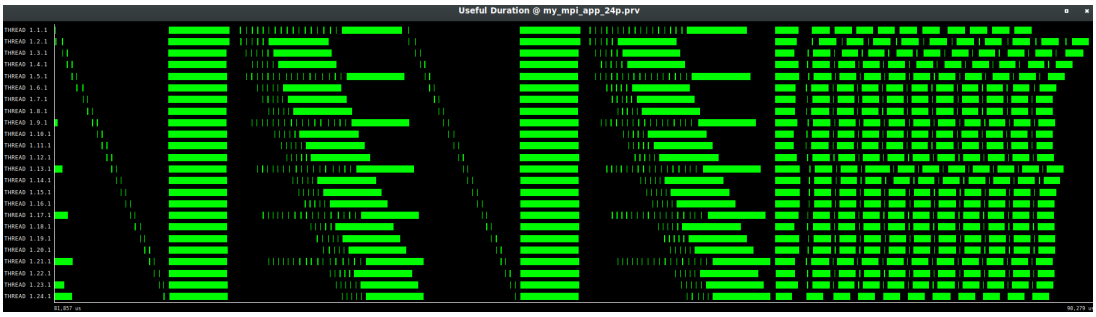


Figure 3.3: Visualization of a typical execution trace using Paraver

In general, a typical Paraver metric visualization (see figure 3.3) displays a timeline, where each row represents a processing element participating in an execution, and the display of zones in each row where the metric being measured shows activity. For example, in figure 3.3 we can see a visualization of an execution with 24 MPI ranks (rows), where each colored section represents the presence of activity for the measured metric at the given section in the timeline (in this case, useful duration). Furthermore, Paraver visualizations can use different color schemes to convey more information, with different interpretations depending on the type of metric displayed.

The main uses of Extrae and Paraver are to identify performance bottlenecks in parallel applications, understand the interactions between different parts of the application, and optimize application performance. By capturing detailed runtime information about an application, Extrae and Paraver enable developers to pinpoint specific areas of the code that are causing performance issues and make informed decisions about how to optimize their application. These tools are widely used in the high-performance computing community and have been instrumental in improving the performance of many scientific and engineering applications.

3.1.3 Massif (Valgrind)

Massif [6] is a profiling tool integrated within Valgrind, which is designed for analyzing the memory usage of programs. It is an open-source, command-line tool that is widely used in software development for detecting memory leaks, identifying memory hotspots, and optimizing memory usage.

Massif works by dynamically instrumenting a program during its execution and tracking the memory usage over time. It can identify the exact amount of memory allocated and deallocated by a program, as well as the peak memory usage and the allocation stack trace. Massif is configured to only be a heap memory profiling tool by default, but it can also be configured to keep track of of the whole memory usage of a process by tracking memory pages instead.

Massif also provides a command line tool for getting detailed visualizations of historical memory usage in the form of snapshots, although 3rd party GUI visualizers are more usable. In this project, we have used KDE's massif-visualizer. We can see an example of a visualization on figure 3.4.

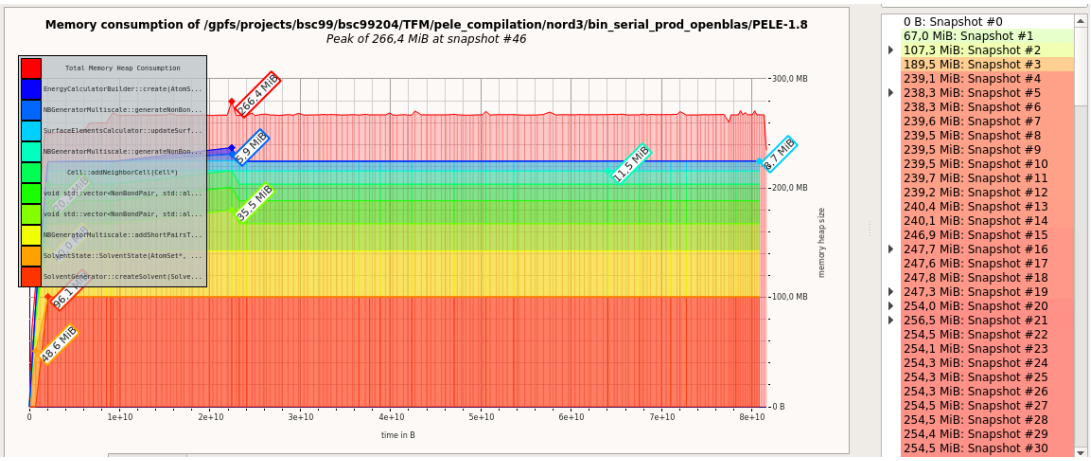


Figure 3.4: Visualization of the memory allocations of an application using massif-visualizer

The main uses of Massif include identifying memory leaks, optimizing memory usage, and identifying memory hotspots. Memory leaks occur when a program fails to free memory that is no longer needed, resulting in a gradual increase in memory usage over time. Massif can help identify the source of the memory leak, allowing developers to fix the problem and prevent it from happening in the future.

Memory hotspots are parts of the code that allocate and deallocate a large amount of memory frequently. These hotspots can be a bottleneck for performance and can also be a source of memory leaks. Massif can identify these hotspots and help developers optimize them to improve performance and prevent memory leaks.

In conclusion, massif is a powerful tool for analyzing the memory usage of programs. There aren't many alternatives for it and it has been the tool that has provided the best insight into the memory consumption patterns of PELE.

3.1.4 Callgrind (Valgrind)

Callgrind [7] is another profiling tool integrated within Valgrind designed for analyzing the performance of programs. It is an open-source, command-line tool that is for detecting performance bottlenecks, identifying hotspots, and optimizing code.

Callgrind works by dynamically instrumenting a program during its execution and recording a call-graph of function calls. It provides detailed information on the time spent in each function call, as well as the number of times each function is called. Callgrind can also provide a visualization of the call-graph in the form of a graph or a tree though an

application called Kcachegrind. In figures 3.5 and 3.6 we can see an example of a function call graph and a listing of different functions alongside their percentage of executed instructions, respectively.

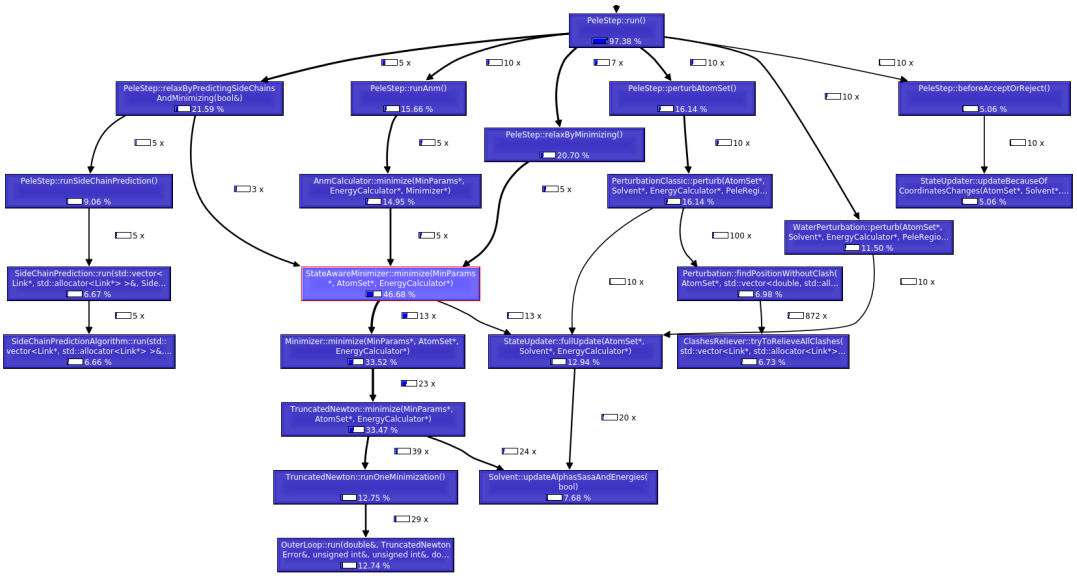


Figure 3.5: Visualization of an application's function call graph using Kcahegrind

Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x0000000000000fa0	ld-2.28.so
99.99	0.00	1	_start	PELE-1.8_serial
99.99	0.00	1	(below main)	libc-2.28.so
99.99	0.00	1	main	PELE-1.8_serial
99.99	0.00	1	runControlFile(std::__cxx11::basic_string<...	PELE-1.8_serial
98.73	0.00	1	Macro::run()	PELE-1.8_serial
98.73	0.00	1	PeleSimulation::run()	PELE-1.8_serial
98.73	0.00	1	PeleTask::run(bool&)	PELE-1.8_serial
97.38	0.00	10	PeleStep::run()	PELE-1.8_serial
58.46	0.00	138	SgbAlphaSasaUpdater::updateAlphasAndS...	PELE-1.8_serial
57.51	0.00	135	Solvent::updateAlphasSasaAndEnergies(b...	PELE-1.8_serial
46.68	0.00	13	StateAwareMinimizer::minimize(MinParam...	PELE-1.8_serial
33.73	0.00	26	Minimizer::minimize(MinParams*, AtomSe...	PELE-1.8_serial
33.73	0.00	418	AtomSetSurfaceElementsCalculator::comp...	PELE-1.8_serial
33.69	0.00	23	TruncatedNewton::minimize(MinParams*, ...	PELE-1.8_serial
31.67	0.99	933 827	SerialAtomSurfaceElementsCalculator::co...	PELE-1.8_serial
21.59	0.00	5	PeleStep::relaxByPredictingSideChainsAn...	PELE-1.8_serial
21.20	14.85	208 600 350	AtomSurfaceElementsCalculator::getSurfa...	PELE-1.8_serial
20.70	0.00	7	PeleStep::relaxByMinimizing()	PELE-1.8_serial

Figure 3.6: Listing of applications contributing to the execution time using Kcahegrind

The main uses of Callgrind include identifying performance bottlenecks, optimizing code, and identifying hotspots. Performance bottlenecks (in callgrind's context) occur when a program spends a significant amount of time in a particular function call or a sequence of function calls. Callgrind can help identify these bottlenecks, allowing developers to optimize the code and improve performance.

Hotspots are parts of the code that are frequently called or take a significant amount of time to execute. These hotspots can be a bottleneck for performance and can also be a source of inefficiency.

In conclusion, callgrind is a powerful tool for analyzing the performance and function call structure of a program's execution. On of the main issues with callgrind (compared to extrae and paraver) is that it is mainly a tool designed for single-threaded sequential programs. However, due to the nature of PELE, optimizations for sequential executions also translate really well to parallel executions.

3.2 Debugging software

3.2.1 GDB

GDB, or GNU Debugger [8], is a command-line tool used for debugging software programs written in languages like C, C++, and other programming languages. It is an open-source tool and is mainly used in software development for debugging complex programs.

GDB works by allowing developers to examine the internal state of a program while it is running or halted, such as the values of variables and the call stack of functions. Developers can set breakpoints in the code and step through the program, examining its execution and state at each step. GDB also provides a range of advanced features such as memory and thread debugging, remote debugging, and scripting capabilities.

The main uses of GDB include debugging code, diagnosing and fixing errors, and understanding the behavior of complex programs. Debugging code involves identifying and fixing issues in the program, such as segmentation faults, memory leaks, or other types of errors. GDB allows developers to identify the source of the error and step through the code to understand how it occurred.

Diagnosing and fixing errors involves finding and correcting problems in the program, such as incorrect program logic or unexpected behaviors. GDB can help identify the source of the error and provide insights into the program's state at the time of the error.

Understanding the behavior of complex programs involves examining how the program executes and interacts with the system. GDB allows developers to explore the program's behavior and execution in detail, helping to identify bottlenecks or areas for optimization.

GDB is probably the most widely used tool (or at least well known) for debugging programs. Its ability to examine the internal state of a program while it is running, set breakpoints, and step through code makes it an essential tool for software developers.

3.2.2 DDT

DDT, or the Distributed Debugging Tool [9], is a parallel and distributed application debugger developed by ARM for use in HPC systems. Unlike GDB, which is a command-line tool, DDT provides a graphical user interface that allows developers to visualize the execution of parallel applications and explore their internal state.

DDT is designed specifically for debugging complex parallel applications that run across multiple nodes or processors in an HPC system. It allows developers to examine the program's execution across multiple nodes or processors and provides advanced features for memory debugging, performance analysis, and message queue monitoring.

One of the key differences between DDT and GDB is the level of parallelism that each tool can handle. DDT is specifically designed to handle parallel and distributed computing systems and can debug applications that run across thousands of nodes or processors. In contrast, GDB is primarily designed for single-threaded applications. Although it can debug parallel applications by tracking each process individually, it is not designed for it.

Another difference is the level of abstraction provided by each tool. GDB works at the source code level, allowing developers to examine the execution of the code and its internal state. In contrast, DDT works at a higher level of abstraction, allowing developers to visualize the execution of the parallel application across multiple nodes or processors and to explore its internal state at a higher level. This becomes apparent in figure 3.7, which is a capture of a typical session. Monitoring and switching between threads and MPI ranks is a basic functionality of DDT.

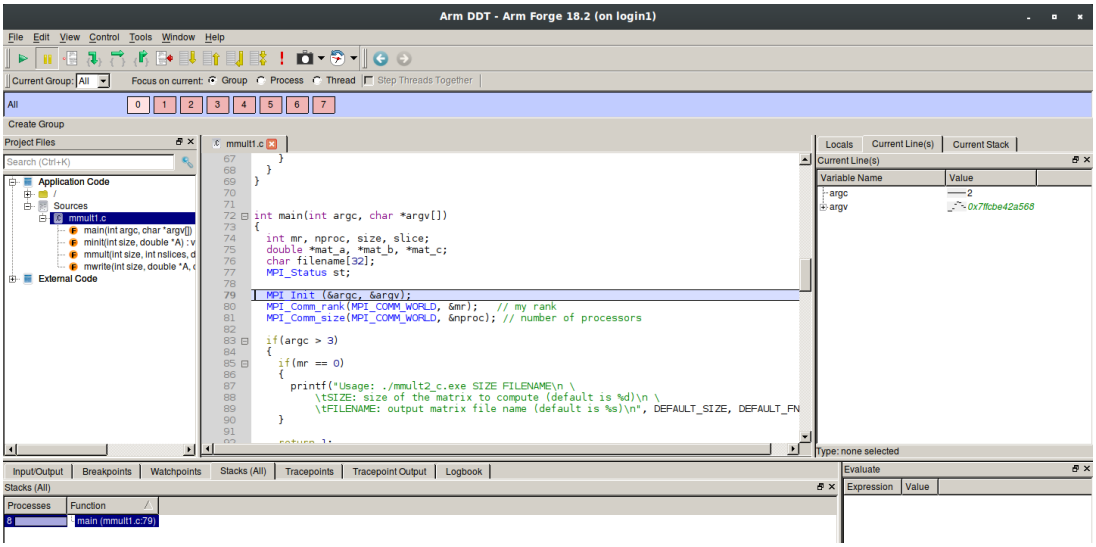


Figure 3.7: Example of a typical debugging session on DDT

Finally, DDT provides a range of advanced features that are not available in GDB. For example, DDT provides a message queue monitor that allows developers to track the communication between different nodes or processors in a parallel application. DDT also provides advanced performance analysis tools that allow developers to identify performance bottlenecks in the parallel application and optimize its performance.

In conclusion, ARM's DDT is a powerful tool for debugging and optimizing parallel and distributed computing systems. Its graphical user interface, advanced features for memory debugging and performance analysis, and its ability to handle parallel applications that run across multiple nodes or processors differentiate it from GDB, which is primarily designed for single-threaded applications.

4 PELE: Preliminary analysis

In order to gain a basic understanding of PELE, we have prepared and run a basic test provided by the Electronic and Atomic Protein Modeling research group at BSC [10] (which will be referred as EAPM from now on), using different compute resource configurations. After some runs, we have decided to trace and analyze some of the executions using Extrae and Paraver in order to see its general behavior and try to identify potential improvement areas.

While we know that memory optimization should be the priority in this study, we would also like to study the feasibility of changes in the code's behavior that could enable further scalability or speedup. We will detail all our observations in the following sections.

4.1 General PELE behavior observed

4.1.1 Testing environment

Unless otherwise specified, all runs and traces described in this section have been done under the following environment:

- **PELE binary**: production MPI + OpenMP binary provided by EAPM.
- **Machine**: Marenostrum 4.
- **Execution parameters**: Please see pele.conf fileA.1 for specific execution configuration. Defines 10 pelesteps.
- **Inputs used**: `1ZNK_complete.pdb`, which is a small input defining a system of 2500 atoms. In future sections it will be referred as "small input".
- **Resource configuration**: Mainly 5 rank pure MPI executions and hybrid MPI + OpenMP executions using 5 ranks with 2 threads/rank.

The PELE production binary provided by EAPM is the MPI + OpenMP version with optimization options enabled. PELE also provides configuration options in order to compile a serial version or enabling CUDA support for specific functions. It can also be compiled using MKL [11] and OpenBLAS [12] libraries interchangeably. By default, it uses MKL.

We also compiled another PELE executable ourselves, since we weren't able to get debugging information in the traces using the binary specified above. At that moment, the obtained executable also didn't provide useful debugging information when tracing it.

4.1.2 Pure MPI execution with 5 ranks

We have noted that the general paradigm of the software is to spawn a master thread and several worker threads. These worker threads seem to be independent copies of the same computation with slight variations. Increasing or decreasing the number of available MPI ranks doesn't increase or decrease the total execution time, it just spawns more worker threads of the same nature (weak scaling).

To illustrate the general behavior of this execution, figure 4.1 shows two different visualizations of the same execution trace: the visualization at the top represents the general useful computation done by each rank, and the one on the bottom illustrates the time that each rank spends on MPI calls.

For clarity purposes, we have displayed green flags in the trace, which indicate the presence of an "event" in the rank at the point in time where it has been placed. In this context, we

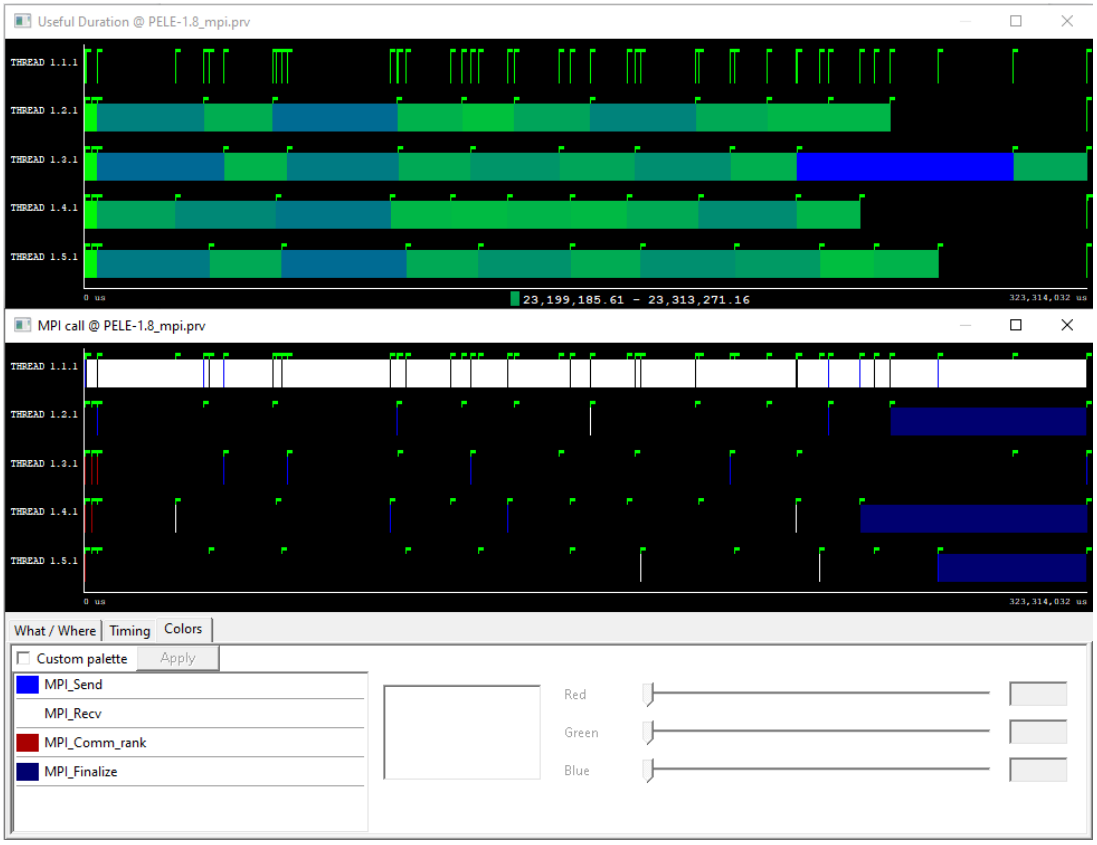


Figure 4.1: Paraver visualization of PELE's useful computation and MPI calls

can understand an event as the presence of an MPI call. In figure 4.2 we can also see the second visualization of the execution trace with the communication lines displayed, just to have a clearer view of the directionality of the MPI communications that are taking place. Finally, in figure 4.3 we can see the full MPI profile of the whole execution in table format. It can prove useful to see the general MPI utilization of the application.

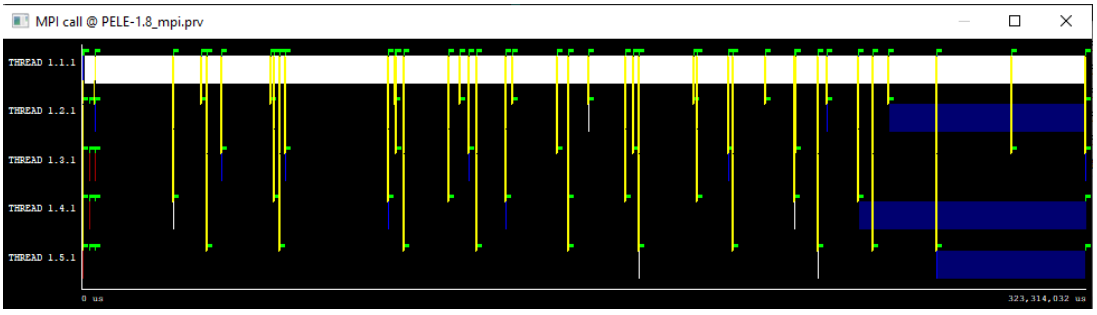


Figure 4.2: Paraver visualization of PELE's MPI calls with communication lines

The first observation that we can make from these traces is that the master rank (thread 1.1.1) does almost no useful computation for the duration of the whole execution. In the useful duration trace, we see that the first rank has barely any presence. Looking at the second trace and MPI profile table, we can see the actual reason: it spends almost all the execution on `MPI_Recv` calls (which are synchronous and blocking).

	Outside MPI	MPI_Send	MPI_Recv	MPI_Comm_rank	MPI_Comm_size	MPI_Init	MPI_Finalize	MPI_Probe
THREAD 1.1.1	0.12 %	0.00 %	99.86 %	0.00 %	0.00 %	0.00 %	0.02 %	0.00 %
THREAD 1.2.1	80.28 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	19.71 %	-
THREAD 1.3.1	100.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	-
THREAD 1.4.1	77.27 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	22.72 %	-
THREAD 1.5.1	85.17 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	14.83 %	-
Total	342.84 %	0.00 %	99.87 %	0.00 %	0.00 %	0.00 %	57.28 %	0.00 %
Average	68.57 %	0.00 %	19.97 %	0.00 %	0.00 %	0.00 %	11.46 %	0.00 %
Maximum	100.00 %	0.00 %	99.86 %	0.00 %	0.00 %	0.00 %	22.72 %	0.00 %
Minimum	0.12 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
StDev	35.10 %	0.00 %	39.94 %	0.00 %	0.00 %	0.00 %	9.68 %	0 %
Avg/Max	0.69	0.83	0.20	0.74	0.97	0.80	0.50	1

Figure 4.3: MPI call profile of PELE's execution provided by Paraver

In contrast, the worker threads spend a negligible amount of time on such calls. The discrepancy between the percentage of time spent outside MPI seen on different worker threads (as seen on figure 4.3) is not actually a consequence of different communication patterns and needs, it's just the product of load unbalance. Different worker threads see different execution times, which forces the faster threads to wait for the slower ones on `MPI_Finalize` calls.

The second observation is that these MPI calls are mostly done at the start and end of each execution step. These execution steps are defined as "PELEsteps" in this software's context. Without counting the initialization phase, there are exactly 10 different regions in each rank, which are the execution steps described in the "pele.conf" file. To see the behavior of MPI calls done between steps, we provide figure 4.4, which is an annotated zoomed-in view of the region between two PELEsteps in the MPI trace shown on figure 4.2.

The general observed behavior between steps is the following:

1. Master rank (MPI rank 0) waits for a worker rank using a `MPI_Recv` call.
2. Worker rank (MPI rank 2) initiates a `MPI_Send` against master rank.
3. After some compute time, the master rank enters another `MPI_Recv` call, expecting a connection from the same worker rank.
4. Worker rank initiates another `MPI_Send` against master rank and enters a `MPI_Recv` call afterwards, waiting for the master rank.
5. After some time, the master rank initiates a `MPI_send` call against the waiting worker thread.
6. MPI exchanges have finalized, master thread resumes its original waiting state by returning to a `MPI_Recv` state.

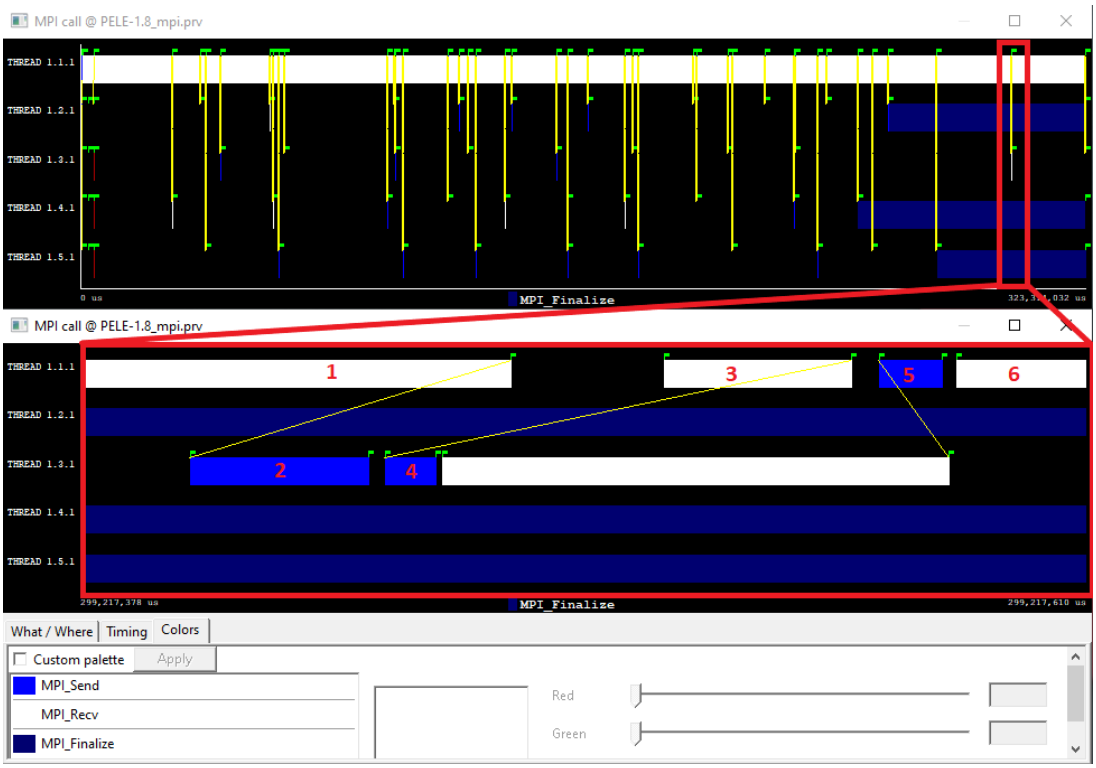


Figure 4.4: Zoomed-in view of the region between PELEsteps of the same trace

This behavior is not unique to that specific region, we have checked that it is consistent with other regions and pairs of master-worker threads. The time spent in these regions and the amount of data transferred are not very significant but contributes to the overall additional overhead and reduce efficiency along several iterations.

Furthermore, another problem becomes apparent when looking at the traces. All MPI ranks finish their useful computation at different points in time. When the execution time difference between ranks is sufficiently large, we can say that an MPI execution shows load imbalance: not all ranks perform the same amount of computation, creating time gaps between ranks where the parallelism starts to decrease. As a result, we start to waste resources because faster ranks have to wait for slower ranks.

4.1.3 Execution using multiple OpenMP threads per MPI rank

During the configuration phase of PELE, we noted that this software has OpenMP capabilities. In order to test them and see their effectiveness, we recompiled PELE with MPI and OpenMP support. After that, a Paraver trace was generated using the same input. In figure 4.5 we can observe the obtained trace. This execution was configured to use 5 MPI ranks with 2 OpenMP threads each, which in total use 10 CPU cores.

The Paraver visualization on the top shows the useful computation done by each thread. Every MPI rank has two threads under it. For the main threads of each MPI rank, the

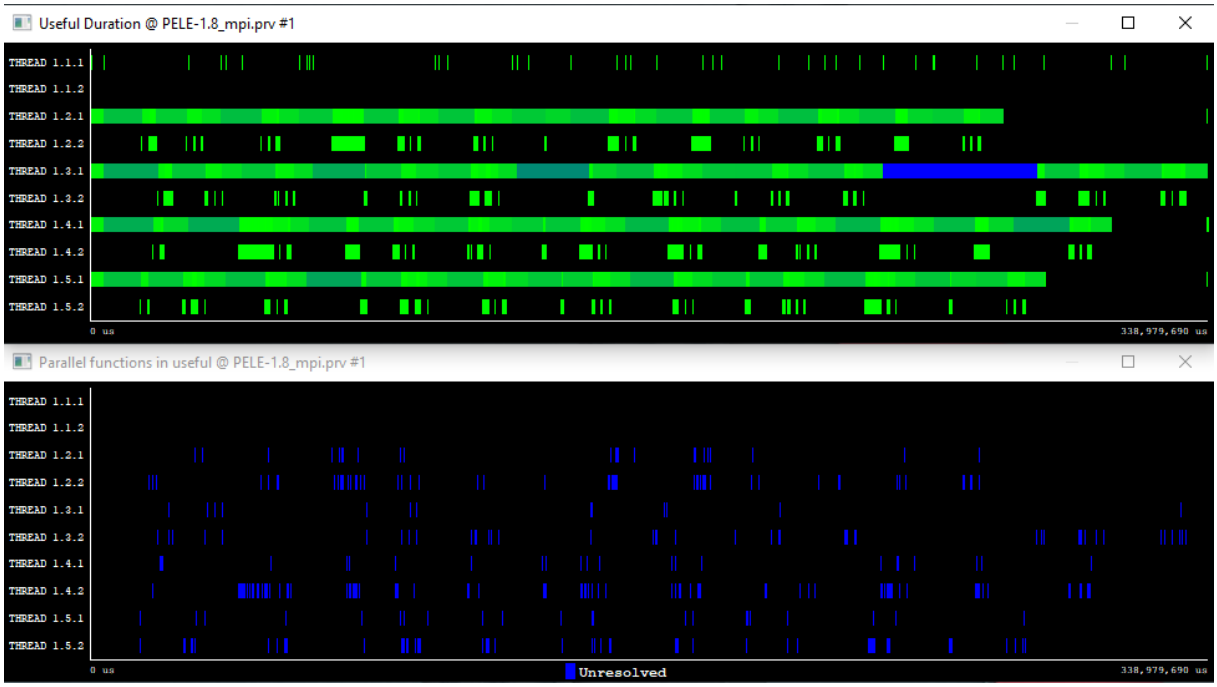


Figure 4.5: Paraver trace of a MPI + OpenMP execution of PELE

general behavior is similar to a pure MPI execution, but we can see that their secondary threads show little CPU usage. In the visualization of the execution trace on the bottom, we can observe the execution regions where parallel computation is present. Some parts of the execution appear to have parallelized functions that make use of the extra CPU of each of the worker MPI ranks. The master MPI rank sees no benefit from using the extra resources.

The general observation is that using that OpenMP implementation was not worth the resources. The extra OpenMP threads allocate one core each, and traces show that the general utilization of those cores is very low. Since those cores are reserved for our job when allocating the necessary resources to do the execution on compute nodes, those extra cores are almost wasted.

4.1.4 Conclusions

We have compiled a small list of conclusions and open questions that we derived from this first analysis. Most of these points were reported and discussed with EAPM's developers.

- Scaling the execution resources only seems to be useful for increasing the size of the problem while maintaining a similar execution time. We can't decrease the total execution time by providing more resources. In conclusion, **this application aims for weak scaling**.

- We see very little communication between MPI ranks. Since the amount of data transferred is almost negligible and the number of communications is very low, the necessity of these communications come into question. Avoiding communication altogether would be preferable, since executions seem to be independent. For a sufficiently large amount of MPI ranks, the master rank could become a bottleneck because it would have to manage point-to-point communications from all MPI ranks.
- The master rank does no real computation, so all resources spent on it are effectively wasted. This might be negligible for large executions, but it can be significant for small-scale runs.
- Do execution steps (PELEsteps) have to be sequential? If not, can they be done in parallel? Since the current OpenMP parallelization strategy doesn't seem to be effective, exploring other options could prove beneficial.
- With OpenMP enabled, why are the parallel sections of the code so small? Is it only present in some specific functions? Could it be used in other functions?
- There seems to be a bit of load imbalance between threads. What determines the execution time of each rank? Can we know in advance the amount of computation to be done in each rank?
- Since the parallelization strategy implemented at this moment is the replication of whole executions, focusing on improving a sequential version would translate its benefits to larger executions.

4.2 MPI execution flow and Adaptive PELE

In this section, we will analyze and describe the existing MPI implementation, alongside another software provided by EAPM that make use of PELE (Adaptive PELE) and will become relevant for this work.

4.2.1 MPI behavior of standalone PELE

When running with MPI, the PELE software assumes a master-slave paradigm. PELE operates by performing several instances of the same simulation using different random seeds, which are then distributed across multiple MPI ranks. Each rank executes its assigned independent simulation and communicates with the master rank (which only acts as controller) to exchange data and coordinate their actions. The results of each independent simulation are then reported at the end of the execution.

The current implementation of MPI in the PELE software has been effective in enabling parallel execution of simulations, since they rely on performing multiple simulations with randomized parameters using the same input. However, this parallelization strategy doesn't provide any execution time speed-up with higher MPI rank counts, it just broadens the exploration space for a PELE execution.

PELE implements two different types of MPI controllers, which act as the master processes that synchronize all worker processes. One is called *MpiBasicExplorationController*, which just waits for incoming communications from all workers. It is mostly used to keep track of the state of all ongoing simulations, but in the past it could also be used to update the coordinates of a worker's system in order to influence where explorations are taking place. However, we have been informed by EAPM that this functionality hasn't been in use for some time and has been superseded by Adaptive PELE (which will become relevant in future sections).

The second MPI controller is called *MpiRealTimeControlExplorationController*, which is a more complex controller that also enables the user to send commands to worker ranks through the controller rank. These commands can pause, modify and restart ongoing simulations in worker MPI ranks. Since we are interested in simplifying communications, **we have decided to work on MpiBasicExplorationController**. The general flow diagram of the master and worker ranks of a PELE execution has been simplified in figures 4.6 and 4.7.

After some examination it was concluded that, when using this MPI controller, there are only three instances where communications between worker and controller MPI ranks take place:

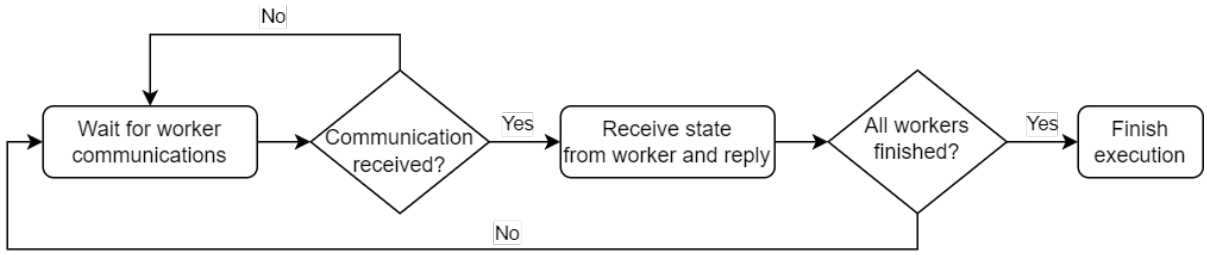


Figure 4.6: Flow diagram of a master MPI rank from a PELE execution

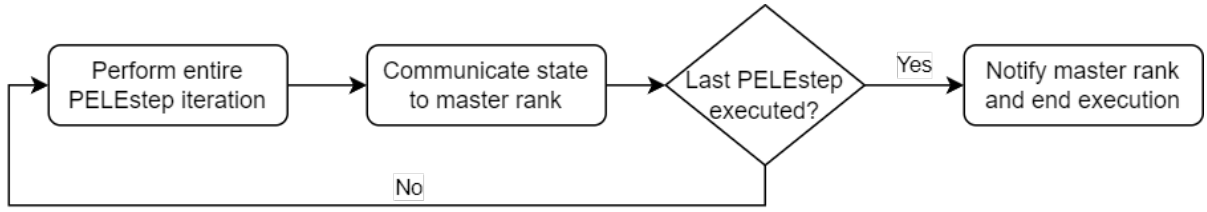


Figure 4.7: Flow diagram of a worker MPI rank from a PELE execution

- **During the initialization phase**: all worker ranks must communicate with the master rank in order to get initialization data. Also, all worker threads also communicate with rank 1 to get further initialization data.
- **After performing a pelestep**: all worker ranks communicate its state and progress to the controller rank. There is no communication during a pelestep.
- **Before terminating the execution**: all worker ranks communicate to the controller rank that they have finished their execution. If all worker ranks have reported the same, the controller does not wait for further communications and ends.

When an execution ends, it produces an output directory containing as many reports, log files and computed trajectories as worker ranks were used in the simulation. These trajectories usually use the same format as the input, so they represent the state of the system at the end of each accepted pelestep and can be used for further simulations. This will become relevant in future sections.

4.2.2 Adaptive PELE

The basic premise of PELE relies on the use of randomized parameters in order to perform multiple simulations over (usually) the same input file. Due to the randomized nature of these simulations, there is no guarantee that an optimal solution (in our case, an accurate trajectory) can be achieved in a timely manner, since there is no control of which exploration space is being exploited. Due to this limitation, Adaptive PELE [13] was developed as a tool to "guide" PELE simulations towards optimal trajectories. Adaptive

PELE is a python-based software that wraps multiple PELE executions, each execution being referred as an epoch. After each epoch, Adaptive PELE uses the produced results to perform a clustering and then tries to identify areas of the exploration space that have been undersampled. Finally, it seeds the necessary control files and input files for the next execution epoch. This process is repeated as many times as the user chooses. Using this method, PELE executions are progressively "guided" towards more desirable exploration spaces, which in turn reduces the time needed to reach optimal trajectories. A general flow diagram of its operation is shown in figure 4.8.

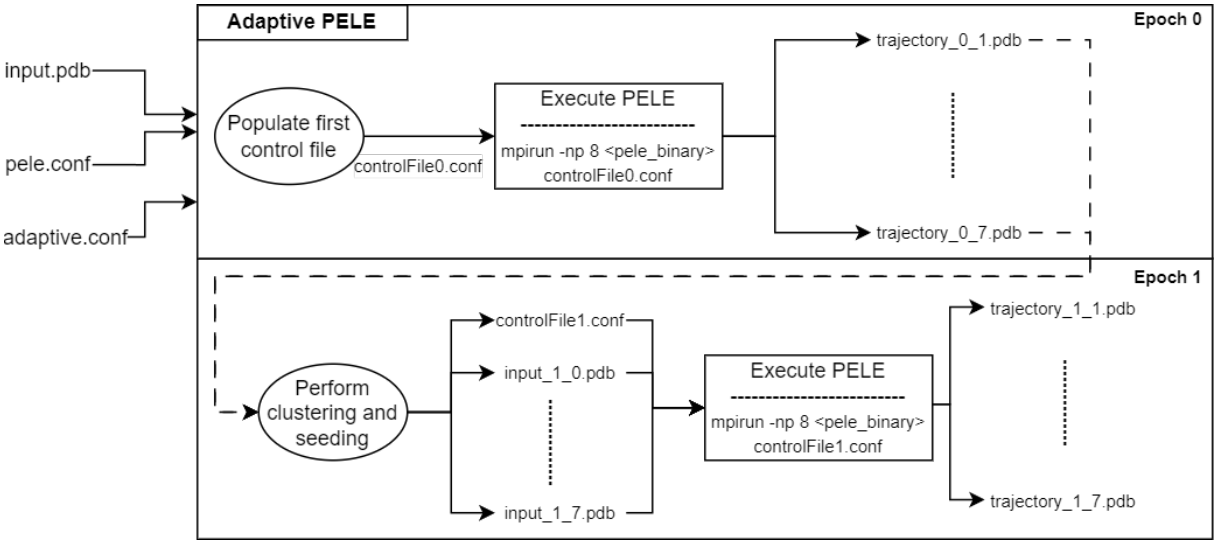


Figure 4.8: Flow diagram of an Adaptive PELE execution, using two epochs and 8 MPI ranks

Adaptive PELE generally needs three input files (we will use representative names):

- **Input.pdb**: Defines the initial state of a system. This is no different than inputs used for standalone PELE executions.
- **Pele.conf**: Same configuration file used for standalone PELE executions, but with some of its configuration parameters being undefined. Used as a template to generate a fully configured control file before a PELE execution takes place.
- **Adaptive.conf**: Configuration file that determines specific parameters for Adaptive PELE (number of epochs, for example) and also which values to be fed to the undefined parameters of *pele.conf*.

These input files are then used to prepare the first epoch of Adaptive PELE. An initial control file (called *controlFile0.conf* in our example) is generated, which is then used to perform the first PELE execution. All resulting trajectories of this PELE execution are then used to perform the previously explained clustering operation, and then a new control file (*controlFile1.conf*) is generated alongside new inputs for the next epoch. The only difference between the first epoch and all other epochs is that the first one uses the same

initial PDB input file provided to Adaptive PELE, while the remaining epochs all have a specific generated PDB input assigned to each MPI rank.

In this work, we will initially focus in standalone PELE executions. However, the general idea of Adaptive PELE will become relevant in future optimization proposals of the PELE code.

4.3 Preliminary memory profiling of PELE executions

One of the most important aspects of this project is evaluating the memory usage characteristics of PELE. It has been reported by the developers that PELE executions can become memory-bound under some circumstances, to the point where executions using all the available resources of a compute node become unfeasible.

On most HPC systems, the amount of memory that a job can use is determined by the amount of CPU cores requested. The reason for this is to guarantee that all CPU allocations in a node have memory to work with. In this context, memory-bound applications might be starved for memory when trying to compute using all their allocated compute resources. To compensate for that, the usual direct solution is to just allocate more CPU cores, even if the application doesn't make productive use of them.

Although this is a common solution that works, it's just a short term bypass of the actual issue. Some HPC clusters also offer limited nodes with more RAM installed, but this shouldn't be taken for granted. Since PELE aims to run in modest systems, memory requirements would have to be minimized in order to guarantee that compute resources are not wasted. To achieve that, in this section we will study how memory requirements increase based on the used models and the number of processors used (i.e. the number of output trajectories found), where the memory hotspots are, and which implementation changes can be done to alleviate them. Due to incompatibilities with Valgrind and the PELE binaries generated for MareNostrum 4, **all executions described in this section have been performed on Nord 3**.

4.3.1 Testing environment and inputs

In order to have a general idea of how much memory PELE needs for a simple execution, we performed some runs using two different inputs, which were provided by EAPM. These inputs are two different PDB files, which define three-dimensional structures of molecules. Each file describes a list of atoms (alongside their characteristics, including their coordinates) of a different molecule. From now on, these files will be referred as:

- **Small input**: molecule comprised of `~2500` atoms.
- **Large input**: molecule comprised of `~18000` atoms.

Furthermore, it has also been noted by the developers that there is a simulation parameter that greatly increases memory usage when enabled in an execution. This parameter is called *bindingEnergy*, which tells PELE that it must also compute the binding energy[1] of

[1] Just for context, the binding energy of a molecule is the minimal amount of energy required to separate its constituent atoms.

the resulting system obtained in a simulation.

The first step was to perform MPI executions allocating all the available CPU cores of a single Nord 3 compute node and then check the node's memory usage. Since each compute node has 16 CPU cores available, we have decided to perform these executions using up to 16 MPI ranks, with each rank being assigned to a different core. Two executions for each input and requested amount of MPI ranks were performed: one without the computation of binding energy and the other with the option enabled.

4.3.2 Results

The memory usage metrics obtained through HPC Portal for the different inputs and options can be summarized in table 4.1. Each cell represents the maximum percentage of memory used in a node. For reference, each node has 32GB of available memory. These computations have been performed using 4, 8 and 16 MPI ranks in order to have a general idea of how the memory usage increased. The executed PELE binary was provided by EAPM, which was compiled using MKL as its BLAS/LAPACK backend.

To determine if the memory usage metrics increases with the amount of time spent doing a simulation, we also experimented with different amounts of "pelesteps" with both inputs. In this context, a "pelestep" is the coarser iterative step the program performs in order to perform explorations. If memory consumption increased with the amount of "pelesteps" performed, that could indicate some sort of memory leaking. The same executions were repeated multiple times with different values of that particular parameter, but the memory consumption metrics remained the same.

	Without binding energy	**With binding energy**
Small input (4 ranks)	14%	15%
Small input (8 ranks)	17%	19%
Small input (16 ranks)	24%	29%
Large input (4 ranks)	39%	65%
Large input (8 ranks)	77%	Out of memory
Large input (16 ranks)	Out of memory	Out of memory

Table 4.1: PELE's memory consumption profile of a single Nord3 node

Although the memory consumption metrics shown in table 4.1 are coarse, they already provide a bit of insight on how PELE's memory requirements scale. Here we can already observe two things:

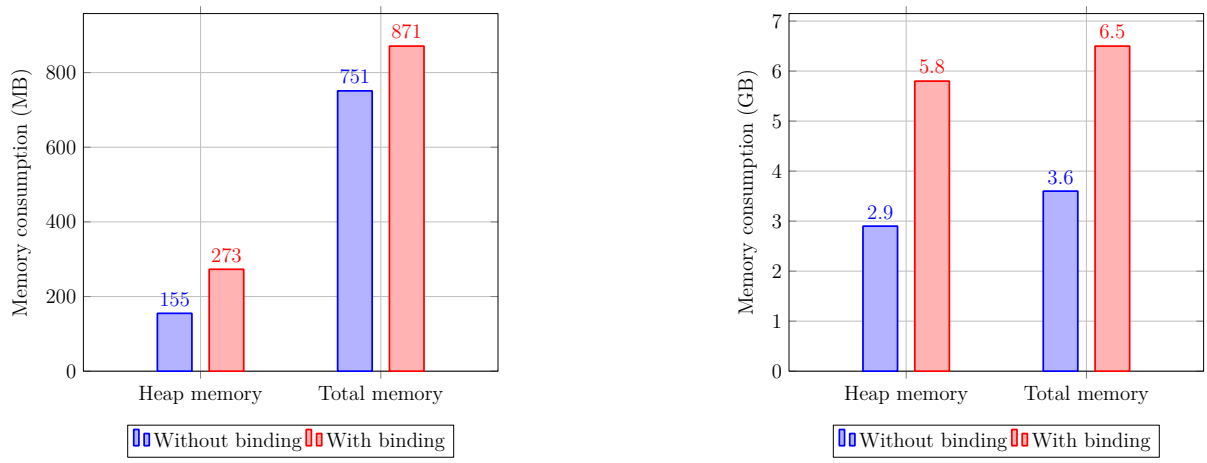
- Memory usage does not scale linearly with the amount of MPI ranks used. An increase in memory usage is expected, since PELE executes parallel independent explorations for each MPI rank.

- Enabling the computation of the system's binding energy increases memory usage. For smaller inputs the increase is not significant, but **it can almost double the memory consumption** for large inputs.

Knowing that each MPI rank performs an independent exploration, we decided to do further profiling using a serial compilation of PELE version 1.8b2 in order to facilitate the analysis. To obtain a more insightful look of how the memory usage evolves in a serial execution of PELE, the same executions were repeated and profiled using Valgrind's Massif tool. Massif profiling files were obtained for serial executions of the small and large inputs, with and without the computation of binding energy.

Furthermore, there is an important detail about Massif that must be taken into account. By default, the obtained memory profiling data **only tracks the heap memory** used by the program (dynamically allocated memory). To track the total amount of memory allocated to our process, Massif provides options that can change its behavior to a lower level measurement of allocated memory pages. Both options have its benefits and its drawbacks: direct heap profiling lets us know exactly where dynamic memory allocation takes place, but at the cost of not seeing the whole picture. On the other hand, memory page profiling fixes that issue but its readings become a lot more "polluted". Knowing this, we decided to use both types of profiling in conjunction to leverage its benefits.

After profiling serial PELE executions using different inputs, binding energy parameters and types of profiling, precise memory consumption metrics were obtained. The results have been summed up in figure 4.9. These metrics have been obtained from the plots generated by *massif-visualizer*, included in appendix B.1.



(a) Small input

(b) Large input

Figure 4.9: Baseline memory consumption metrics of serial PELE executions with different parameters

In both bar plots, we can see the memory usage metrics of serial executions using the small and large input. On each plot we have displayed the heap memory and total memory used

by the executions (heap memory is included in the total memory measurements), which are also color-coded to differentiate executions that do not compute binding energy and executions that have that option enabled. Please note that the scale of the first plot is in MB and measurements approach 1GB, while the scale for the second plot is in GB and measurements approach the 7GB mark.

In both plots, we can observe that the difference between heap memory and total memory for the same execution is always around 600MB, independently of the input used.

4.3.3 Conclusions

The results obtained for the serial PELE executions show a similar behavior to what we have seen with MPI executions on table 4.1, where using a larger input makes memory usage scale significantly faster. One interesting fact that we can observe is that for smaller inputs, dynamically allocated memory (heap) is just a small fraction of the total memory used for an execution. For larger inputs, it becomes the main contributor to memory consumption. In general, enabling the computation of binding energy almost doubles the amount of allocated heap memory.

With this initial study, we can define the values summarized in figure 4.9 as the baseline from which we will implement improvements in further chapters.

5 Optimization and parallelization proposals

In chapter 4, we conducted a preliminary analysis of PELE's performance and behavior, identifying several areas where improvements could be made. Specifically, we detected bottlenecks in the program's parallelization strategy and memory usage, the latter significantly limiting its effective resource usage. In this chapter, we present several optimizations and parallelization strategies that can address these issues and enhance the performance of the PELE program.

The proposed optimizations include the efficient use of the available hardware resources, a significant reduction in memory usage and change the real precision of the computational operations from double to single-precision. Additionally, we propose some modifications on the current parallelization strategy to leverage the capabilities of modern multicore processors and high-performance computing clusters.

By implementing these optimizations and parallelization strategies, we hope to improve currently feasible PELE executions and even enable PELE simulations that were originally resource-inefficient (or even unfeasible) on modest systems.

5.1 Improvements on memory usage

5.1.1 Intel MKL vs OpenBLAS

In section 4.3, and more specifically in figure 4.9, we noted that the difference between the heap memory usage and total memory usage values was particularly significant for executions that used the small input. Concretely, a gap of 600MB could be observed between the values obtained for the small input executions.

Since the difference observed was not trivial, we inspected the binaries and libraries used to check if we could reduce memory consumption with just some changes in the compilation of PELE's binary.

In listing 5.1 we can observe the reported disk size of the PELE binary alongside the MKL libraries that are linked to it. It should be noted that MKL is not the only major library linked to PELE (there are others like Boost, for example), but is one of the few ones that are interchangeable with other alternatives like OpenBLAS.

```
> ls -lah PELE-1.8_serial
-rwxr-xr-x 1 bsc72236 bsc72 39M mar 17  2022 PELE-1.8_serial

> ldd PELE-1.8_serial | grep mkl
libmkl_intel_lp64.so.1 => <path_to_libmkl_intel_lp64.so.1>
libmkl_intel_thread.so.1 => <path_to_libmkl_intel_thread.so.1>
libmkl_core.so.1 => <path_to_libmkl_core.so.1>

> ls -lah libmkl_intel_lp64.so.1 libmkl_intel_thread.so.1 libmkl_core.so.1
-rwxr-xr-x 3 bsc99002 bsc99 72M sep  4  2021 libmkl_core.so.1
-rwxr-xr-x 3 bsc99002 bsc99 13M sep  4  2021 libmkl_intel_lp64.so.1
-rwxr-xr-x 3 bsc99002 bsc99 62M sep  4  2021 libmkl_intel_thread.so.1
```

Listing 5.1: Disk sizes reported for PELE's serial binary and MKL libraries

Just counting linked MKL libraries alone, we can see that they can potentially take 147MB before any computation is even done. Just for reference, in listing 5.2 we can see that other available alternatives like OpenBLAS only take up about 18MB.

```
> ls -lah libopenblas_sandybridgep-r0.3.21.so
-rwxr-xr-x 1 bsc99204 bsc99 18M dic 15 23:15 libopenblas_sandybridgep-r0.3.21.so
```

Listing 5.2: Disk sizes reported for OpenBLAS library

Knowing that using MKL is not mandatory, a serial PELE binary was recompiled and linked against OpenBLAS to test if there were memory gains. The same executions were performed again, and we can see that it did in fact improve total memory consumption with little to no impact on heap memory used. An example of the obtained values for an execution using the small input with the computation of binding energy can be seen in figure 5.1.

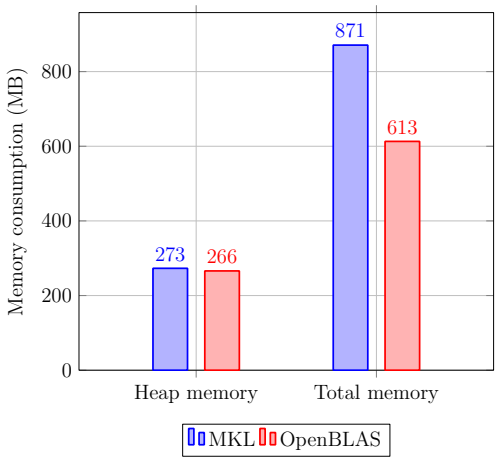


Figure 5.1: Memory consumption comparative of MKL vs OpenBLAS, small input, with binding energy

Although the memory benefits obtained using OpenBLAS might not be very significant for large systems and simulations, these are gains that come at practically no cost. If PELE is to be executed in modest systems and memory becomes a bottleneck, using OpenBLAS is one easy way to alleviate memory needs. The execution time cost of switching from MKL to OpenBLAS has also been measured by comparing executions using both libraries. At most, OpenBLAS only increases execution time by about 2.5% when compared against MKL.

5.1.2 Code changes

To obtain significant reductions in memory usage, a more detailed study of the executions and code was necessary. Concretely, we needed to know exactly when dynamic memory allocations were taking place and how much memory was used in each of them. After this, we would be able to get a general idea of which parts of the code were the main contributors to memory usage and where to focus our efforts.

To get this information, we have relied on a combination of Massif and GDB. In section 4.3 we explained that we could get the maximum memory metrics of each PELE execution thanks to Massif. Massif works by combining multiple snapshots of the execution at different points in time, with each snapshot containing a stack trace detailing how much memory has been allocated at every level of the stack.

Since the executions with the large inputs were the ones with more apparent memory needs, we decided to base our analysis on those. In figure 5.2 and 5.3 we can see the Massif snapshots obtained at peak memory consumption time, for an execution without the computation of binding energy and another with the option enabled, respectively. We have enabled a more detailed view of the stack trace of the functions with most memory requirements, with each line being the function where the allocation took place. In case

of function dependency, each function is a caller of its upper function and a callee of its downward function.

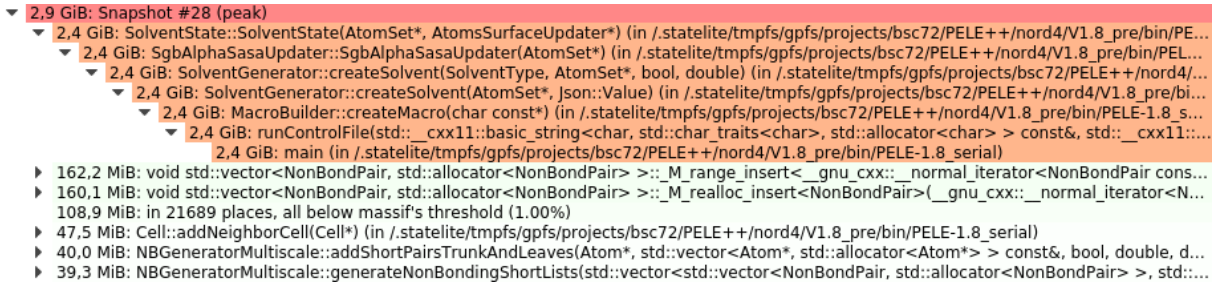


Figure 5.2: Massif snapshot of the peak heap memory usage of PELE, without binding energy

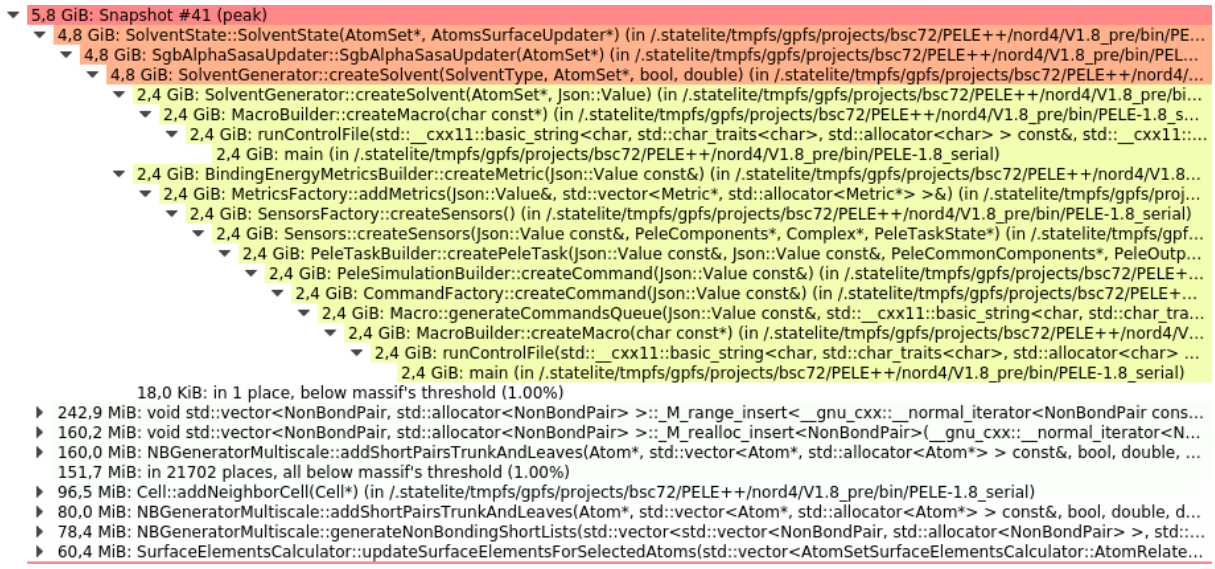


Figure 5.3: Massif snapshot of the peak heap memory usage of PELE, with binding energy

Inspecting both traces, we can see that the highest contribution to memory usage is located at the constructor function of the class "SolventState". For the execution without binding energy, we see that this function is only accessed from a specific execution flow, but that is not the case for the execution with binding energy enabled, where we can see that the same function has been called following two different paths.

Furthermore, we can also observe that the increased memory usage from the execution with binding energy comes from the addition of the memory used in the call to "BindingEnergyMetricsBuilder::createMetric", which is uses exactly the same amount of memory as "SolventGenerator::createSolvent". This fact indicates that this increase in memory when computing binding energy could be due to some sort of data replication.

Once the general location of the memory hotspots had been identified, we compiled a debug version of PELE and ran it with GDB, placing breakpoints at the start of those functions. At the same time, we also monitored the node's available memory to determine the exact line where the allocation happens. To make sure that the readings obtained

```
1  SolventState::SolventState(AtomSet * atomSet, AtomsSurfaceUpdater * atomsSurfaceUpdater)
2  {
3      this->atomSet = atomSet;
4      unsigned int numAtoms = atomSet->getNumberOfAllAtoms();
5      fill_n(back_inserter(hadFreeSasaNeighbors), numAtoms, false);
6
7      hadFreeLowResolutionNeighbors.resize(numAtoms);
8      hadFreeMediumResolutionNeighbors.resize(numAtoms);
9
10     frozenAtomsSurfaceContribution.resize(numAtoms);
11     for(unsigned i = 0; i < numAtoms; ++i)
12         frozenAtomsSurfaceContribution[i].resize(numAtoms,0);
13
14     atomsSurfaceIntegral.resize(numAtoms);
15     updateFrozenAndFreeList();
16     this->atomsSurfaceUpdater = atomsSurfaceUpdater;
17 }
```

Listing 5.3: Constructor method for SolventState class

were specifically from our testing, we reserved a whole node for the execution. In listing 5.3 we can see the code of the constructor method for the class "SolventState".

We detected that the memory consumption spike happens on lines 10-12 of listing 5.3. In these lines, we can see what that the program dynamically allocates a matrix called "frozenAtomsSurfaceContribution" of "numAtoms" x "numAtoms". For reference, "numAtoms" in this context refers to the amount of atoms that comprises the system defined in the input file. If we take into account all observations done until now, it becomes apparent that **the minimum memory complexity of PELE is at least $O(n^2)$**, where n is the amount of atoms (or input data) of the system. This explains the sharp increase in memory when using larger inputs.

Knowing this, the main objective should be decreasing the memory complexity of this "frozenAtomsSurfaceContribution" data structure. To do that, we should first determine the answer to the following questions:

- When and how is this structure written to or read from?
- Is this structure really needed in its current state?

After inspecting all related code that accesses this structure, we have determined that modifications to specific matrix positions are only done when initializing the whole structure, either for the first time or after a whole update of its contents is requested. More importantly, if we check all the read operations done on this structure, we see that the code where they are done doesn't only save one specific element, it accumulates all values into single variables. To be more exact, all read operations follow a similar pattern seen in listing 5.4.

These two facts combined answer our second question: this structure is not needed (unless there are future developments that rely on it), since we don't strictly need to read arbitrary

```
1  ...
2  for(unsigned i = 0; i < numAtoms; ++i){
3      Atom * atom = allAtoms[i];
4      double surface = 0.0;
5      if(atom->isFrozen()) {
6          for(unsigned j = 0; j < numAtoms; ++j) {
7              surface += frozenAtomsSurfaceContribution[atom->vectorId][j];
8          }
9      }
10 }
11 ...
```

Listing 5.4: Simplified code of read operations performed against "frozenAtomsSurfaceContribution"

positions of the matrix, we only need its values to compute their sum. Knowing this, the most straightforward approach would be to just discard the whole structure, compute the sum once and then only save that value. However, we can't use this solution because not all values of the structure are always used to compute its sum.

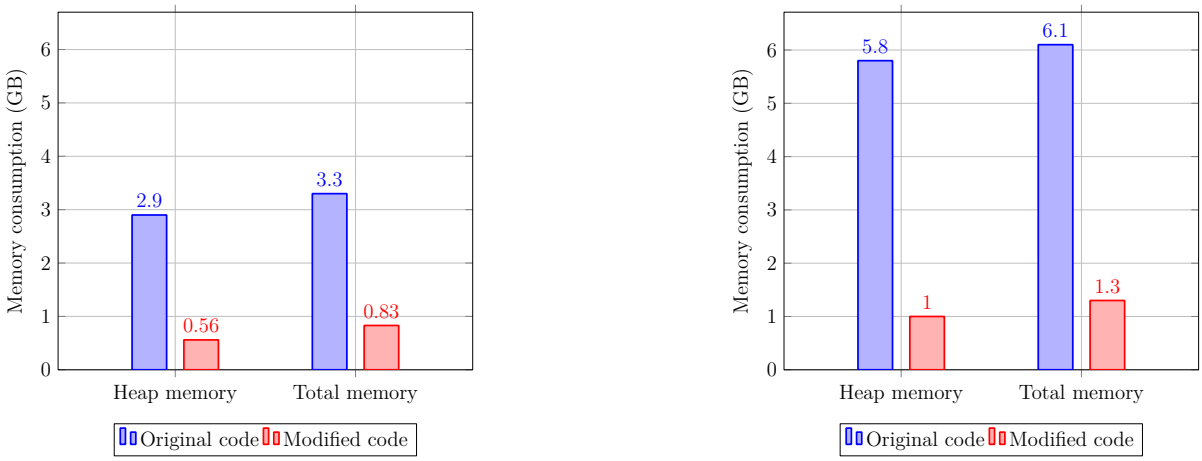
In lines 5-9 of listing 5.4 we can see that only some rows of the structure are used for the computation, with the condition being if an atom is frozen or not. If the atom is frozen, the whole row with that atom identifier is added to the sum. If not, it doesn't participate to the computation. While this means that we cannot simply save the sum value of all "frozenAtomsSurfaceContribution" elements, what we can do is to simplify the structure to a vector. Each position would contain the sum of a whole row, and then its initialization alongside all read and write operations should be adapted to accommodate the change. With this solution, **the memory complexity for this part of the code should drop to $O(n)$**.

5.1.3 Results

After applying these changes, a new serial binary was compiled (using OpenBLAS) and executed. Once the obtained outputs were compared against outputs from unmodified PELE executions and were confirmed to be valid, another memory profiling was performed using Massif. In figure 5.4 we can see the new memory consumption metrics compared against the unmodified version of PELE, using both versions the large input and OpenBLAS. Keep in mind that these values are for serial executions.

As we can see, for the large input we have managed to obtain significant reductions in memory usage. One important detail to point out is that the execution that computes binding energy has also benefited from this change, since both execution paths identified in figure 5.3 end up calling the same class constructor method. Most importantly, these memory improvements come at no cost: the execution time has not increased.

This can be seen in figure 5.5, where we have compared the execution times of a serial PELE run of our new version against the unmodified one. We have also used a different



(a) Large input, no binding energy

(b) Large input, with binding energy

Figure 5.4: Memory usage comparison between the original PELE code and our modified version

number of *pelesteps* to see how the execution time scales. Both versions use OpenBLAS and the large input.

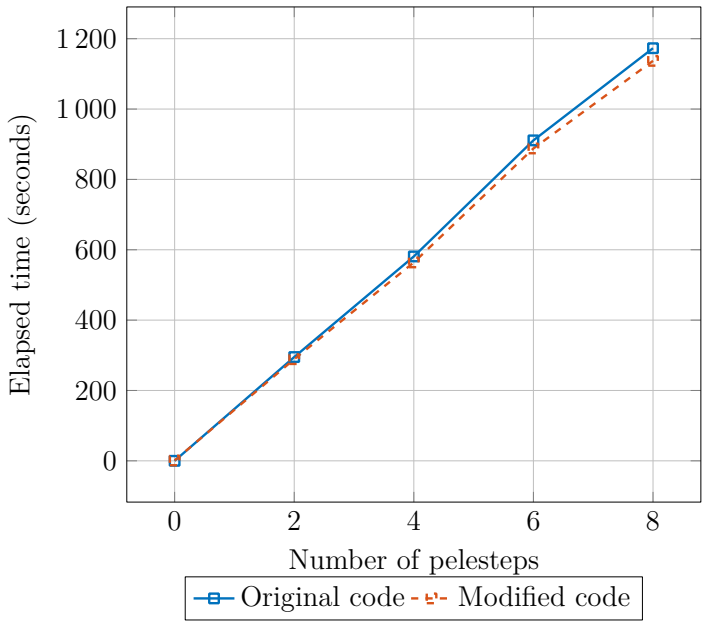


Figure 5.5: Serial execution times of memory improved PELE against original binary, large input

This modified version also presents slightly shorter execution times, which is to be expected since we now read vectors of *numAtoms* instead of matrices of *numAtoms* x *numAtoms*. It is important to note that our implemented change is not a memory-optimal solution. If the state of the atoms (frozen or not frozen) were known when *frozenAtomsSurfaceContribution* is defined, we could even discard all vector positions of non-frozen atoms, since they are not relevant for this structure. This would further reduce memory usage, but it would complicate reading values from the structure since it would dynamically change its number of elements and how atoms are indexed. Regardless, the results obtained with our current

solution were satisfactory enough and we decided to keep it for simplicity's sake.

With these changes applied, we tried to replicate the executions performed in table 4.1 to see how the improvements translate to MPI runs. These results can be seen on table 5.1, which contains the newly obtained memory metrics. Again, the percentages indicate the amount of used memory on a single Nord3 node of 32GB.

	Without binding energy	**With binding energy**
Small input (4 ranks)	13%	14%
Small input (8 ranks)	16%	17%
Small input (16 ranks)	20%	23%
Large input (4 ranks)	16.5%	20%
Large input (8 ranks)	24%	33%
Large input (16 ranks)	36%	57%

Table 5.1: PELE's memory consumption profile of a single Nord3 node after memory improvements

5.1.4 Conclusions

The improvements made to reduce the memory usage of the PELE software have been successful in significantly reducing the memory complexity from $O(n^2)$ to $O(n)$, while maintaining performance and without introducing any negative side effects. This has enabled the possibility of performing parallel simulations that were previously resource-inefficient or not possible due to memory constraints.

The reduction in memory complexity was achieved by implementing several changes to a specific data structure. The resulting improvements in memory usage have allowed the software to handle larger systems with greater resource efficiency. Furthermore, it has been shown that the use of OpenBLAS over MKL libraries can also reduce the amount of memory allocation needed.

Overall, the success of these improvements is probably the most beneficial change to PELE in this work, since it eliminates one of its most limiting issues: over-allocation of compute resources due to memory constraints.

5.2 Floating point precision changes

PELE is a software that exclusively performs double-precision floating point operations to compute the energy bindings. Although this is not uncommon, there are other applications (like GROMACS [14]) that allow the use of single-precision, or at least the use of mixed precision using only double-precision for sensitive variables. The benefits of using single-precision usually come as an improvement in execution times for codes that are compute intensive, alongside a decrease in memory consumption since single-precision variables take up only half the memory of their double-precision counterparts. After the examination of the inputs and outputs used and produced by PELE, we determined that the values used by the software should be representable using single-precision floating point variables. For example, in listing 5.5 we can see that the the range of the input values used is not large and that their precision is about 10^{-3}.

```
ATOM      1  N   MET A1001      12.703   6.654  -0.425  1.00 102.30        N
ATOM      2  CA  MET A1001      12.126   6.933  -1.750  1.00 106.02        C
ATOM      3  C   MET A1001      11.810   5.648  -2.563  1.00 100.52        C
ATOM      4  O   MET A1001      11.019   5.693  -3.521  1.00  93.34        O
ATOM      5  CB  MET A1001      13.051   7.871  -2.547  1.00 104.69        C
ATOM      6  CG  MET A1001      12.701   8.052  -4.045  1.00 109.14        C
ATOM      7  SD  MET A1001      10.958   8.432  -4.422  1.00 114.51        S
...
```

Listing 5.5: First lines of a PDB input file defining a molecule

For PELE's usual generated output files, it's true that their values require a bit more precision, but it still should be fairly accomplishable using single-precision variables. These output files generally contain a set of computed metrics of the system at different points of the simulation, an example can be seen in listing 5.6.

```
#Task  Step  numberOfAcceptedPeleSteps  currentEnergy  BindingEnergy  sasaLig  ligandRMSD
1      0     0                         -22636.4       122.449        0.185127 0
1      2     1                         -34859.5       -25.4017       0.183314 2.02767
```

Listing 5.6: Example report of a PELE simulation

5.2.1 Changes and problems

PELE's code contains a fair amount of hardcoded numeric values that directly feed into functions. At compilation time, these numeric values are implicitly treated as *double* data types, which at the same time produce a data type mismatch in these functions because they expect to be fed *float* parameters. To avoid this problem, all offending numerical values were explicitly cast to *float*.

Another important matter to consider when applying these changes is that the former precision or values of the constants used by PELE might be inadequate when using single-precision floating point operations. One critical example is the constant that defines

```
extern "C" {
// LAPACK routines
void dsptrd_(char*, int*, const double* const, double*, double*, double*, int*);
void dstevr_(char*, char*, int*, double*, double*, double*, double*, int*, int*, double*,
     int*, double*, double*, int*, int*, double*, int*, int*, int*, int*);
void dopmtr_(char*, char*, char*, int*, int*, const double* const, double*, double*, int
    *, double*, int*);
void dsyev_(char*, char*, int*, double*, int*, double*, double*, int*, int*);

// BLAS routines
double dnrm2_(int *n, const double * const x, int *incx);
double ddot_(int * n, const double * const x, int * incX, const double * const y, int *
    incY);
void daxpy_(int * n, double * alpha, const double * const x, int * incx, double * const y
    , int * incy);
void dscal_(int * n, double * scalar, double * x, int * incx);
void dcopy_(int *n, const double * const x, int * incx, double * y, int * incy);
}
```

Listing 5.7: Original BLAS and LAPACK routines declared in PELE's code

the threshold used to determine if two floating point values can be considered equal or not, or even the definition of the maximum value that can be considered zero. Some of these constants were too close to the representability limits of single-precision floats, and in some cases they were not representable. To alleviate the problem, the precision values of the worst cases were slightly relaxed.

Another unexpected issue was that PELE's code was using the Fortran versions of BLAS/LAPACK, as shown in listing 5.7. All Fortran versions are characterized the "_" suffix in the name, and having all the parameters as pointers. This is relevant, since it was found out that calling single-precision BLAS/LAPACK functions in this manner could produce unexpected values that would then silently propagate through the execution. The correctness of the new function declarations was checked multiple times, but the issue could only be resolved by switching the Fortran calls to their C-based interfaces provided by "cblas" and "LAPACKE".

A representative example of the previously defined behavior in PELE is the case of an infinite loop caused by silent *NaN* propagation. In PELE, there is a point where a grid-like structure of three dimensions is created. After that, all the atoms of the simulation are mapped to different positions of the grid, depending on their three-dimensional coordinates. To generate this grid, the program needs to ensure that the grid can properly map all atoms, and for that it needs to know which are the maximum and minimum values of each dimension. In conclusion, it needs to check the coordinates of all atoms beforehand and then determine the dimensions of the grid based on those values. This is what can be seen in listing 5.8.

As soon as the boundaries of the grid are determined, atoms can start to get mapped in it. However, what ends up happening is that coordinates of some atoms fall outside the boundaries of the generated grid. PELE detects this condition and decides to regenerate

```
void CellList::computeHighestAndLowestPositions() {
  Atom** atoms = complex->getAtoms();
  double * coords = complex->getCartesianCoordinates();
        // ...
  for(unsigned int i = 0; i < complex->getNumberOfAtoms(); i++)
    {
      // Find boundaries
      Atom* a = atoms[i];
      lowestX = min(coords[a->ix], lowestX);
      lowestY = min(coords[a->iy], lowestY);
      lowestZ = min(coords[a->iz], lowestZ);

      highestX = max(coords[a->ix], highestX);
      highestY = max(coords[a->iy], highestY);
      highestZ = max(coords[a->iz], highestZ);
    }
    //... function continues here, but code is not relevant
}
```

Listing 5.8: Code of function CellList::computeHighestAndLowestPoints

the grid, starting this cycle over. This goes on indefinitely, with the general loop diagram being shown in figure 5.6.

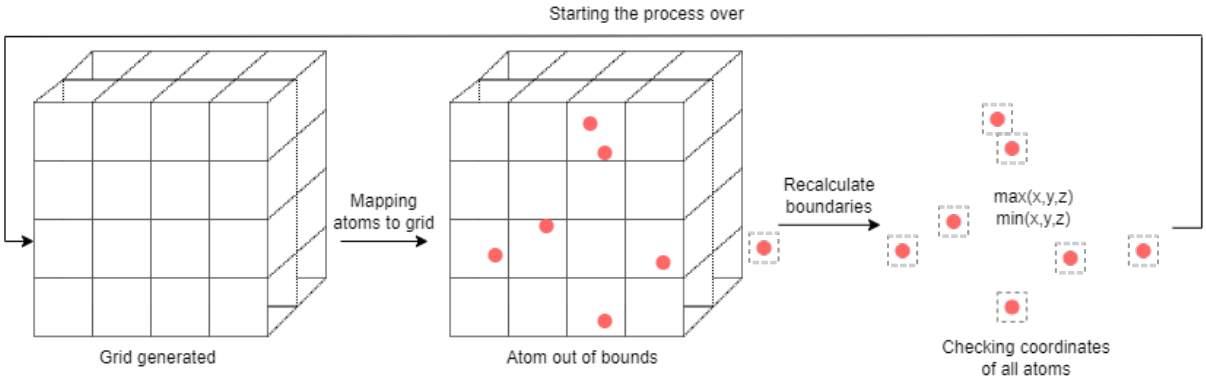


Figure 5.6: Conceptual diagram of the infinite loop while creating a grid

One contradiction is clear: a grid was being constructed using the coordinates of all atoms to determine its boundaries, but then those very same atoms ended up being mapped out of boundaries. This behavior was found to be caused by arithmetic operations using *NaN* as one of their operands. By leveraging one of the properties of the IEEE-754 floating point standard [15] that determines that a floating point *NaN* cannot be equal to itself, the precise producers of those values could be determined. These producers, at the same time, used values obtained through OpenBLAS/LAPACK functions, which later were found to be incorrect. There have been several cases of unexpected behaviors that had to be extensively studied with debugging tools, but we have left them out for the sake of brevity.

After applying all previous changes, a working version of PELE using single precision values was obtained, which was able to finish test executions without further issues.

5.2.2 Results

Differences in results accuracy

Once we got a working version of this new serial single-precision PELE binary, two runs were performed using the large input: one with a single-precision binary and another one using the previous one. After that, we compared the values of both reports and obtained trajectories. For example, in listing 5.9 we can see a quick comparison of a few lines of the output PDB files for both versions. In figure 5.10, we can see an equivalent comparison with their report files. Please note that the single-precision version used for this execution only modified constants that were not representable with this new precision.

```
// DOUBLE PRECISION
ATOM      1  N   MET A1001      12.925   6.590  -0.491  1.00 102.30           N
ATOM      2  CA  MET A1001      12.407   6.970  -1.801  1.00 106.02           C
ATOM      3  C   MET A1001      12.031   5.729  -2.634  1.00 100.52           C
ATOM      4  O   MET A1001      11.483   5.844  -3.732  1.00  93.34           O
ATOM      5  CB  MET A1001      13.420   7.916  -2.490  1.00 104.69           C
...
********************************************************************************
// SINGLE PRECISION
ATOM      1  N   MET A1001      12.872   6.576  -0.425  1.00 102.30           N
ATOM      2  CA  MET A1001      12.391   7.024  -1.729  1.00 106.02           C
ATOM      3  C   MET A1001      11.851   5.845  -2.554  1.00 100.52           C
ATOM      4  O   MET A1001      11.184   6.042  -3.572  1.00  93.34           O
ATOM      5  CB  MET A1001      13.533   7.785  -2.442  1.00 104.69           C
...
```

Listing 5.9: Side by side comparison of output PDB files of double and single precision PELE executions

```
// DOUBLE PRECISION
#Task    Step    #OfAccPeleSteps    currentEnergy    sasaLig    ligandRMSD
1        0       0                  -22636.4         0.185127   0
1        2       1                  -35162.1         0.184356   1.99551
************************************************************************
// SINGLE PRECISION
#Task    Step    #OfAccPeleSteps    currentEnergy    sasaLig    ligandRMSD
1        0       0                  -22634.8         0.185127   0
1        2       1                  -35788.9         0.184496   1.90868
```

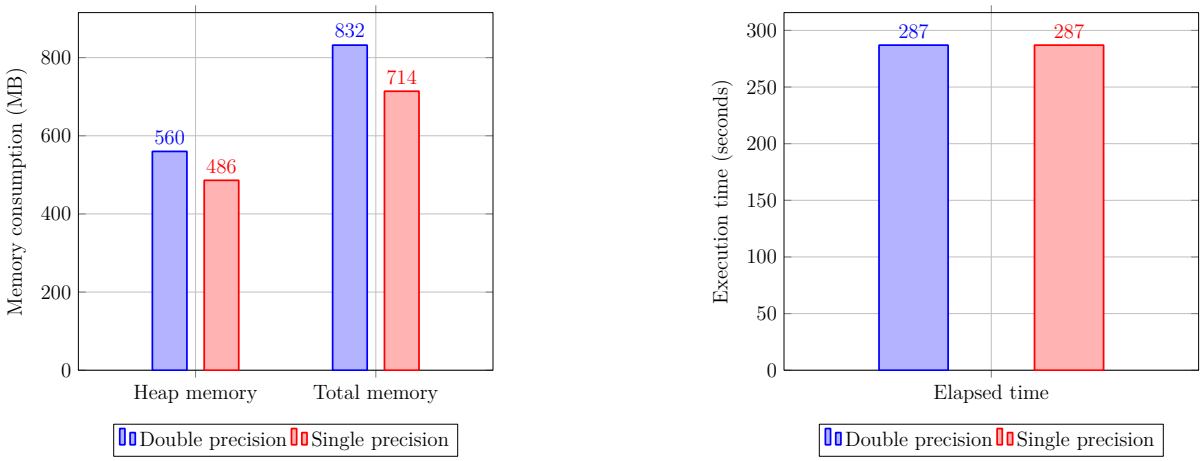
Listing 5.10: Side by side comparison of report files of double and single precision PELE executions

What can be seen in listing 5.9 is that the obtained coordinate values of the atoms (6th to 8th column) present non-negligible differences between versions. After systematically comparing all values from both PDB files, we determined that **the obtained values for this particular execution can present a fluctuation of ±0.8 units**. A similar issue can be seen in the *currentEnergy* column of both reports, where the reported values can have significant differences between them.

Since the evaluation of the quality of these reported results falls outside of our area of expertise, we provided these output files to PELE developers. The validity of the results is still pending to be evaluated by them.

Memory usage and execution time benefits

In this section, we will try to determine if these changes have a significant impact on execution time and memory usage for a PELE execution. Using serial PELE binaries, one using double-precision and another one using our modified version, we performed 4 runs for each binary using the large input (without the binding energy option). After tracing their average execution times and memory usage metrics, we obtained the results seen in figure 5.7.



(a) Memory usage comparison

(b) Average Execution time comparison

Figure 5.7: Metrics comparison of single and double precision serial PELE, large input, no binding energy

For this particular execution, we can see that **we have obtained a ~15% decrease in memory usage** when compared with our previous double precision version of PELE. However, execution time remains the same. This was an unexpected outcome, since arithmetic operations using single precision operands should be faster than their double precision counterparts. Inspecting the logs of the execution, there is a bit of profiling information for one of the algorithms used in the simulation. In listing 5.11 we can see the values reported for it, for the double and single precision executions.

```
// DOUBLE PRECISION
totalNewtonIterations: 3
totalInnerLoops: 214
totalOuterLoops: 29
************************
// SINGLE PRECISION
totalNewtonIterations: 3
totalInnerLoops: 309
totalOuterLoops: 26
```

Listing 5.11: Side by side comparison of report files of double and single precision PELE executions

It seems that the change to single precision did also increase the amount of iterations to perform for the truncated Newton algorithm to converge (which is used during the minimization phase of PELE). To confirm that this section of the execution is actually

taking more time, we traced this execution using callgrind. In figure 5.8 we can see the relevant sections of the trace.

Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000000fa0
100.00	0.00	1	_start
100.00	0.00	1	(below main)
100.00	0.00	1	main
100.00	0.00	1	runControlFile(std::__cxx11::b...
91.93	0.00	1	Macro::run()
91.93	0.00	1	PeleSimulation::run()
91.93	0.00	1	PeleTask::run(bool&)
91.58	0.00	2	PeleStep::run()
54.35	0.00	2	PeleStep::relaxByMinimizing()
53.72	0.00	1	StateAwareMinimizer::minimi...
50.63	0.00	13	SgbAlphaSasaUpdater::updat...
46.70	0.00	12	Solvent::updateAlphasSasaA...
45.59	0.00	1	Minimizer::minimize(MinPara...
45.59	0.00	1	TruncatedNewton::minimize(...
31.36	0.00	3	TruncatedNewton::runOneMi...
31.36	0.00	3	OuterLoop::run(float&, Trunc...

Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000000fa0
100.00	0.00	1	_start
100.00	0.00	1	(below main)
100.00	0.00	1	main
100.00	0.00	1	runControlFile(std::__cxx11::b...
90.94	0.00	1	Macro::run()
90.94	0.00	1	PeleSimulation::run()
90.94	0.00	1	PeleTask::run(bool&)
90.54	0.00	2	PeleStep::run()
56.04	0.00	13	SgbAlphaSasaUpdater::updat...
51.69	0.00	12	Solvent::updateAlphasSasaAn...
48.90	0.00	2	PeleStep::relaxByMinimizing()
48.17	0.00	1	StateAwareMinimizer::minimi...
39.17	0.00	1	Minimizer::minimize(MinPara...
39.17	0.00	1	TruncatedNewton::minimize(...
33.08	0.00	13	SolventState::updateAtomsSu...
33.08	6.06	13	AtomsSurfaceUpdaterSerial::...

Figure 5.8: Callgrind traces of single (left) and double (right) precision serial PELE executions

The highlighted functions in the callgrind traces are the ones where the iterations of listing 5.11 take place. As we can see, the single precision version takes a higher percentage of execution time in that function than the double precision version. The amount of iterations to perform in that function cannot be known beforehand, the end conditions for their loops are based of the convergence of values under a certain threshold.

As a proof of concept, we created two new single-precision PELE versions: one with single-precision awareness constants and another with a limited amount of iterations in the truncated Newton algorithm. The first binary is like the original single-precision implementation, but adjusting the single-precision requirements of some constants. The second version has been limited to perform, at most, the same amount of *totalInnerLoops* observed in the double precision output of listing 5.11. After performing another round of executions using these new single-precision versions, we obtained the timings observed in figure 5.9.

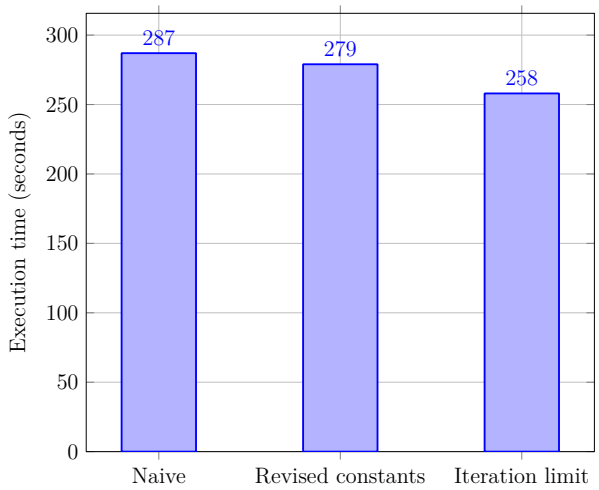


Figure 5.9: Execution times of single precision serial PELE binaries, large input, no binding energy

In the figure, *Naive* is the baseline single-precision version. *Revised constants* and *Iteration limit* are the single-precision range awareness and limited iteration versions, respectively. The *Revised constants* results show that execution time could actually decrease if the values for constants were revised in a controlled manner. Timings in *Iteration limit* indicate that modest execution time improvements could theoretically be achieved if PELE's single precision code was adapted to the point of requiring the same amount of truncated Newton algorithm iterations of its double precision counterpart.

5.2.3 Conclusions

Due to the complexity of the code, the reason for the differences in the amount of iterations needed to perform in some functions couldn't be fully determined. The code was later inspected and we found out that there is a significant amount of hardcoded numerical threshold values, some of those very close to the limit of single-precision representability. One option would be to relax the precision needed for those numerical thresholds, but without knowing a range of acceptable values, it could negatively affect the validity of the results.

Due to these circumstances, we informed PELE developers about this situation. If it were reasonable to reduce the precision requirements of all constants, reducing the execution time of the simulation would be feasible. However, we haven't been able to determine new valid values for those constants. Since the correctness of these changes haven't been fully validated yet, future sections will still use the double precision version of PELE as its baseline.

5.3 MPI design changes

In this section, we will focus on improving the MPI implementation of the PELE software. The current implementation of MPI in the PELE software has been effective in enabling parallel execution of simulations. However, we have identified several areas where performance can be improved. These include the use of unnecessary communications, the presence of a master rank that doesn't contribute to the computation, and the presence of load unbalance.

To address these issues, our proposals include the elimination of unnecessary MPI messages, the implementation of fully independent worker ranks that do not rely on controllers, and trying to decrease the impact of load unbalance for executions that follow the Adaptive PELE workflow. These changes will be performed in incremental steps. We will evaluate the effectiveness of these proposed improvements through testing and benchmarking on the Nord 3 system.

Ultimately, the goal of these improvements is to enhance the efficient use of resources of the PELE software.

5.3.1 Eliminating unnecessary MPI communications and controllers

On section 4.1.2 we observed the presence of MPI communications after each performed pelestep. These communications were exclusively performed between worker ranks and the controller rank, and never directly between worker ranks. These communications are just to report a worker's status to the controller, but this information is not relevant for the execution of a simulation.

Since it is known (as explained by EAPM) that the controller currently does not influence the execution of its worker ranks, all these communications can be omitted. Doing so, all worker ranks can freely run without having to stop to report to the controller, which will facilitate future changes. From the controller's side, the changes are simple because all communications are managed in the same file and they are very localized. In general, the code for the controller used for this can be summarized in listing 5.12.

For this specific part of the code, we can see that the controller waits for incoming communications in a synchronous *MPI_Recv* call, and once this communication happens, it executes different code depending on the event received. After that, it starts to wait again. The only thing that we need to do here is to eliminate all the helper functions that reply to the worker ranks once the controller receives an incoming communication from them.

From the worker point of view, we also need to eliminate all MPI calls targeting the

```
while(numExplorersFinished < numberOfExplorers){
  MPI_EVENTS event;
  MPI_Status status;
  MPI_Recv(&event, 1, MPI_INT, MPI_ANY_SOURCE, EVENT, MPI_COMM_WORLD, &status);
  int explorerId = status.MPI_SOURCE;
  switch(event){
    case EXPLORER_FINISHED:
      controllerEventsLogger->logExplorerFinishedEvent(explorerId);
      numExplorersFinished++;
      break;
    case EXPLORER_SENDS_STATE_TO_CONTROLLER:
      helper.respondToNewExplorerState(explorerId, jumpData, controllerEventsLogger);
      break;
    case EXPLORER_SENDS_PROGRESS_TO_CONTROLLER:
      helper.respondToNewExplorerProgress(explorerId, jumpData, controllerEventsLogger);
      break;
    case EXPLORER_REQUEST_COORDINATES:
      helper.sendJumpingCoordinates(explorerId, jumpData,
                EXPLORER_SENDS_PROGRESS_TO_CONTROLLER, controllerEventsLogger);
      break;
    default:
      throw PeleException(
          "Error: unexpected MPI event received by controller",
          "MpiBasicExplorationController::listenToExplorersEvents");
      break;
  }
}
```

Listing 5.12: Relevant code used by MPI controller to receive worker communications

controller, with the exception of the initialization and finalization communications. These changes were harder to implement, since MPI code was scattered throughout multiple files. Most of the worker MPI calls were tracked down by searching the MPI tags used by the controller. The remaining worker MPI calls that were not found using the previous method were discovered using the DDT parallel debugger.

With these changes, we obtained a PELE binary that could perform entire MPI executions without synchronization calls from its worker ranks, with the exception of its initialization and finalization calls. Once all the intermediate communications were omitted, we decided to focus on also eliminating the MPI controller rank and replacing it with another worker.

To achieve this, we eliminated the previous declaration of the MPI controller in the code and the distinction between MPI ranks when spawning PELE explorations. The finalization communication from worker ranks was also omitted, since each worker can produce its output by itself without the need to report it back to the controller. This produced a binary that could perform an additional simulation using the same amount of MPI ranks as before.

To be able to ascertain the validity of the results of this new version, some changes were needed to the distribution of the seeds used for the randomization of the computation parameters. Originally, the seed used by each MPI rank was computed in the following manner:

$$LocalSeed = OriginalSeed + MpiRankID - 1$$

Previously, MPI rank 0 was reserved for the controller MPI rank, with all following MPI ranks being workers. This is why the computation of the seed not only adds the value of the MPI rank, but also subtracts one unit to ensure that the original seed specified in the configuration file is mapped to the first worker rank. With the current version, the subtraction was omitted since now all MPI ranks are workers. This change enables the direct comparation of the produced reports and trajectories from both versions. After some PELE test runs using the original and our new modified version, it was determined that they produced equivalent results (and also an extra simulation trajectory and report).

5.3.2 Mitigating the effects of load imbalance

On section 4.1.2 we also saw the presence of load imbalance in the Paraver visualizations of our initial executions. Each pelestep could take an unpredictable amount of time to be performed, and they need to be executed sequentially. Since migrating pelestep executions to MPI workers that end their workload early would probably not be trivial and would require changing the whole execution model, it was determined that tackling this issue directly from PELE would not be the best choice.

PELE is usually launched using Adaptive PELE, which encapsulates multiple PELE executions. For each of those PELE executions, Adaptive PELE experiences periods of time where multiple MPI ranks are just waiting for all workers to end due to load imbalance. This happens for all epochs of an Adaptive PELE run. Ideally, MPI ranks shouldn't have to wait between epochs, but having to perform multiple PELE MPI executions enforces this waiting time.

However, if we performed all clustering and input generation for each epoch from PELE itself instead of relying on an external tool, we could theoretically assign workloads to MPI ranks that have already finished their assigned computation for an epoch. To measure the potential benefits of such a change, we decided to further modify PELE to implement a naive proof of concept.

Although it would be ideal to perform the clustering and seeding directly from PELE, we would need to study and port the existing Adaptive PELE python code to C++ and integrate it into the PELE software. This is beyond the scope of this work, so we decided to do the following to do this test:

- **Generate a pool of control files and inputs.** For this, we would use Adaptive PELE and save all generated intermediate input and control files for a specific execution.
- **Modify PELE code so it can use the previously generated files.** PELE would need to be able to perform multiple epochs by itself. To identify which input files to use each time, it would use its MPI rank ID and epoch number.
- **Ensure that the produced results are coherent**. We must confirm that the produced outputs from this new version of PELE are exactly the same from an equivalent Adaptive PELE execution.

For the first step, we requested EAPM to provide us with a test case for an Adaptive PELE execution, the configuration files can be seen in appendix A.3 and A.4. This test case was adapted and executed using 30 epochs, which provided sufficient input files for our purposes. These input files are conveniently named using an epoch identifier and MPI rank ID, which facilitates their use in our new PELE version.

For the second step, PELE was modified to accept a new input parameter to determine the amount of epochs to perform. Each worker rank performs its initial simulation as usual, but after finishing, it it now checks the value of this new input parameter (if present). If there are more epochs to be performed, it continues performing simulations using the previously provided input files that matches its current epoch and MPI rank. Following this strategy, we implemented two new versions of PELE: one implementing an MPI barrier between epochs, and the other omitting this barrier. The first version was meant to mimic the current Adaptive PELE execution model (without clustering time costs and the spawning of multiple MPI executions), while the second one represented an ideal version where each worker wouldn't need to wait at the end of each epoch.

For this last version, it should be noted that it doesn't consistently resolve the load imbalance issues of PELE. While we should be able to eliminate waiting times between epochs, there is no guarantee that there won't be load imbalance at the end of the last epoch (which would be the accumulation of imbalances from all the previous iterations). However, we can say that this implementation would be, at worst, as fast as the current implementation of Adaptive PELE. For that to happen, all the simulations with the highest execution time in each epoch would have to land consistently on the same MPI worker. Since the probability of this is very low, we should still have better execution times than Adaptive PELE on average.

Finally, in order to recreate a similar seed assignation algorithm used by Adaptive PELE and be able to compare the results, we used the following seed generation formula:

$$LocalSeed = BaseSeed + EpochID * (NumMpiRanks + 1) + MpiRankID$$

For the first epoch (epoch 0), the seed assigned to each worker is just the sum of the base seed provided by the configuration file and the MPI rank identifier of the worker. After that, the formula would also take into account the current epoch being executed. Since our version needs one less MPI rank to perform the same computation as the default Adaptive PELE (because it assumes that one of the ranks will act as a controller), we need to increment the number of MPI ranks by one in the algorithm.

With these changes, we were able to reproduce the same exact results as Adaptive PELE using our new PELE version.

5.3.3 Results

After compiling the last two new PELE versions (with and without MPI barriers), we generated Extrae traces of both using the intermediate files generated by Adaptive PELE. Since the original Adaptive PELE execution was done using 8 MPI ranks, we performed our executions using 7 ranks to account for our missing controller rank. These traces can be seen in figures 5.10 and 5.11. It is important to note that both executions are equivalent in terms of the computations being performed.

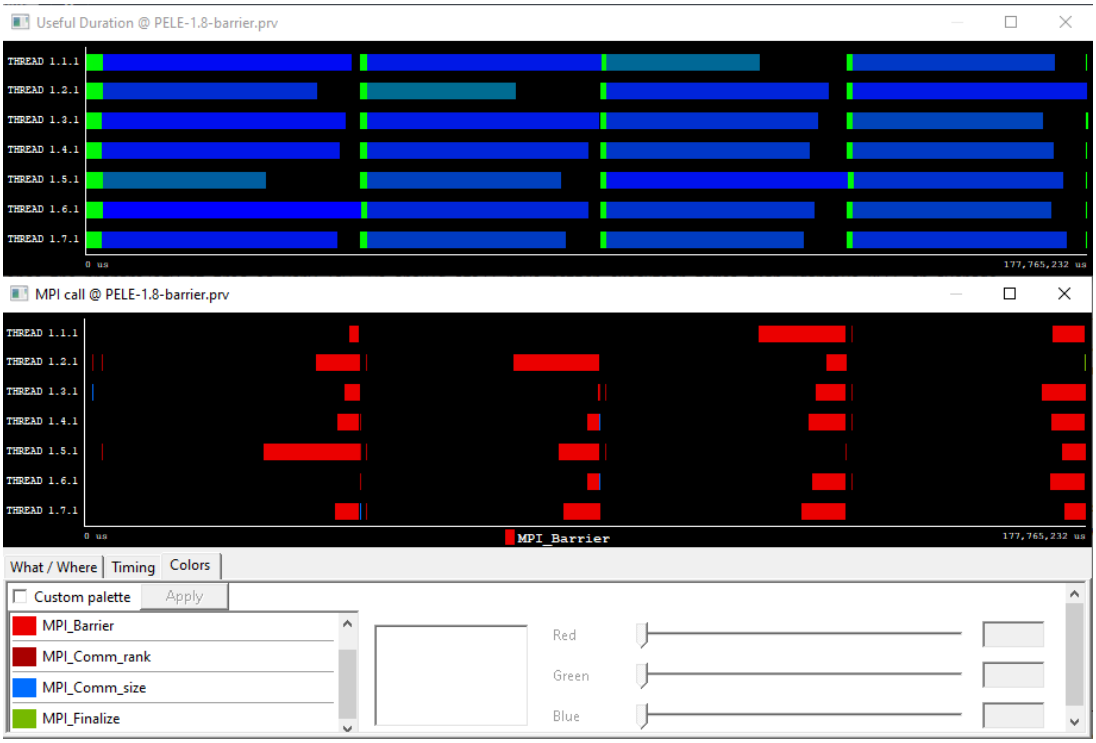


Figure 5.10: Paraver visualization of useful duration and MPI calls of new PELE, 4 epochs, with barriers

In the top visualization of figure 5.10, we can see four clear sections, each one being a full epoch. Each epoch shows a load imbalance equivalent to what has been seen on earlier standalone PELE executions, which has been forced by implementing MPI barriers between epochs as seen on the bottom visualization, where we are displaying all MPI calls taking place.

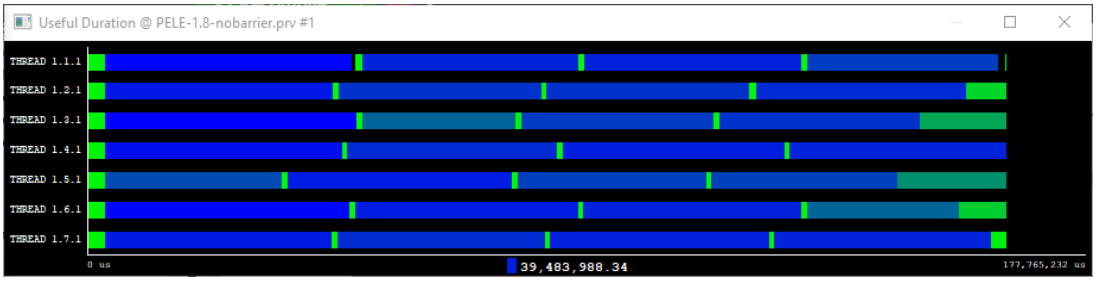


Figure 5.11: Paraver visualization of useful duration of new PELE, 4 epochs, no barriers

If we remove such barriers and allow each MPI rank to advance to its next epoch as soon as possible, we obtain a situation like the one seen in figure 5.11. This visualization has been scaled to use the same time duration as the previous ones.

We can see that the effects of load imbalance could be mitigated using this strategy, since we are performing exactly the same execution in less time. Since Adaptive PELE executions can be configured with different amounts of epochs and pelesteps/epoch, we decided to quantify the potential benefits of this strategy by comparing execution times between Adaptive PELE and our proof of concept using multiple configurations.

Three sets of Adaptive PELE intermediate files were generated, each using a different amount of pelesteps/epoch. The rest of the parameters were the same ones defined in configuration files A.3 and A.4, using an input PDB file defining a system of approximately 2600 atoms. The executions would be performed using 9 MPI ranks for Adaptive PELE, and 8 MPI ranks for our version. These intermediate files would later be used to feed our PELE executions and obtain multiple execution times. Figure 6.4 shows the execution time for 4, 8 and 16 epochs, and 2, 4 and 8 pelestep per epoch, for the original Adaptive PELE flow, the PELE barrier (precomputed epoch information and MPI barrier to wait for the end of each epoch) and the PELE no barrier (precomputed epoch information but no MPI barrier).

We can observe that Adaptive PELE executions are always slower than the proof-of-concept versions of the barrier and non-barrier PELE versions. Execution time improvements are more appreciable for higher epoch counts, which should be expected. While it is true that Adaptive PELE should be slightly slower due to the fact that it actually needs to perform a clustering operation in-between epochs to obtain new input files, the associated cost to this operation is very small: for this execution, each clustering took 0.25 seconds at most.

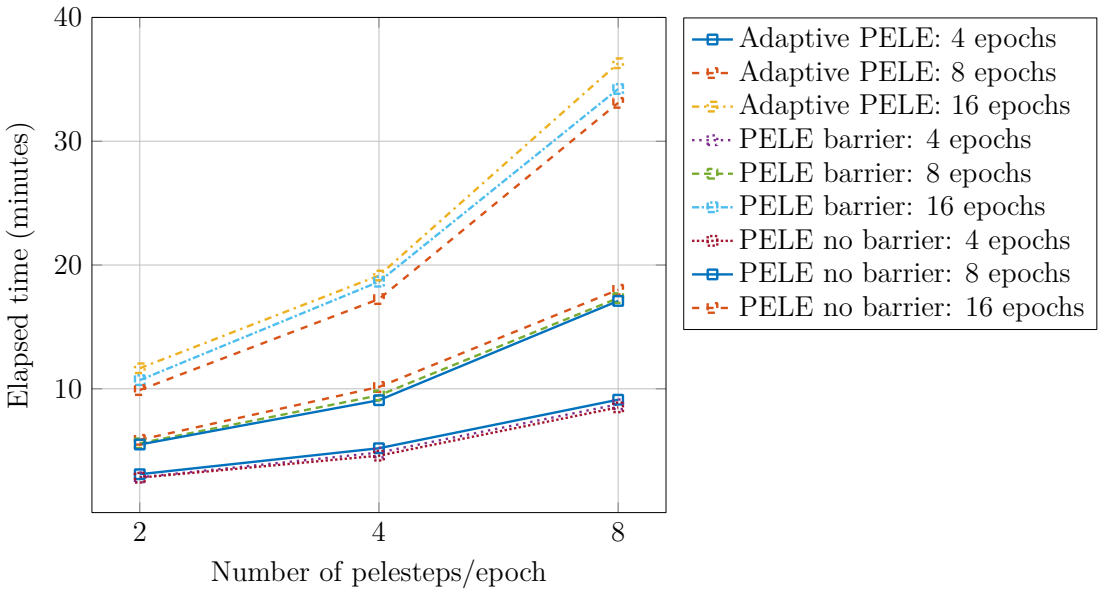


Figure 5.12: Execution times of Adaptive PELE and new PELE versions with different parameters

It is also important to note that the benefits of this new version are not only a slight improvement in execution time, but also in the amount of needed resources, since we need one less MPI rank. This benefit can be negligible for PELE executions using high MPI rank counts, but it can become significant for small-scale runs.

5.3.4 Conclusions

A new version of PELE could be created, which completely removes the need for MPI controller ranks and eliminates all unnecessary MPI communications between pelesteps. This effectively lets us perform an extra concurrent simulation using the same amount of resources needed for the original PELE version. Furthermore, we adapted PELE to be able to execute multiple epochs using intermediate input files generated by Adaptive PELE.

After some testing, this new version has shown that there would be potential benefits in execution time if MPI ranks were able to successively execute epochs without waiting for all other ranks, mitigating the effects of PELE's MPI load imbalance.

5.4 Relaxing Monte Carlo simulations

In section 5.3 we tested Adaptive PELE and introduced the idea of enabling worker MPI ranks to start executions without having to wait for all MPI workers still working in the same epoch. We tested a new standalone PELE version that simulated the current workflow of Adaptive PELE using previously generated input and configuration files, and finally an ideal version that showed that reductions in execution time were possible if all workers were able to perform epochs without interruption.

These proofs of concept, however, relied on having a pool of input and configuration files already available, since the clustering and intermediate file generation operations were not being performed. In this section, we present a final PELE version that actually performs these operations while also trying to mitigate the effects of load imbalance seen between execution epochs.

5.4.1 Implementation

Adaptive PELE is basically a python wrapper that launches multiple MPI executions of PELE, and once each execution finishes, it performs clustering and file spawning operations using all the results that each MPI rank provided at the end of its epoch. For the implementation of our new version of PELE, we have reversed the whole paradigm: we will have an MPI PELE version that is only executed once, but it will internally perform multiple epochs. Each MPI rank will invoke a modified version of the Adaptive PELE python wrapper to perform its related clustering and spawning operations at the end of each epoch.

The main benefit of this approach is that it lets us control when and how the clustering operations for each MPI rank are performed. Each MPI worker rank performs a clustering operation with a global pool of available trajectories at that specific moment, which generates the initial PDB input file to be used for the PELE execution of its current epoch. After that, the same MPI worker generates its needed PELE configuration file for its current epoch and performs a simulation. The results of this simulation are then added to the global pool, which will be used by other MPI ranks to perform their clustering operations. The general execution flow has been described in figure 5.13.

It is important to note that the first PELE epoch is used to generate the initial pool of input files. MPI ranks must wait for all their peers to finish the initial PELE simulation, and then one of the MPI ranks will perform a clustering step using all the obtained results, in a similar manner to what the original Adaptive PELE operated. After that initial clustering, all ranks can start to continue their execution independently.

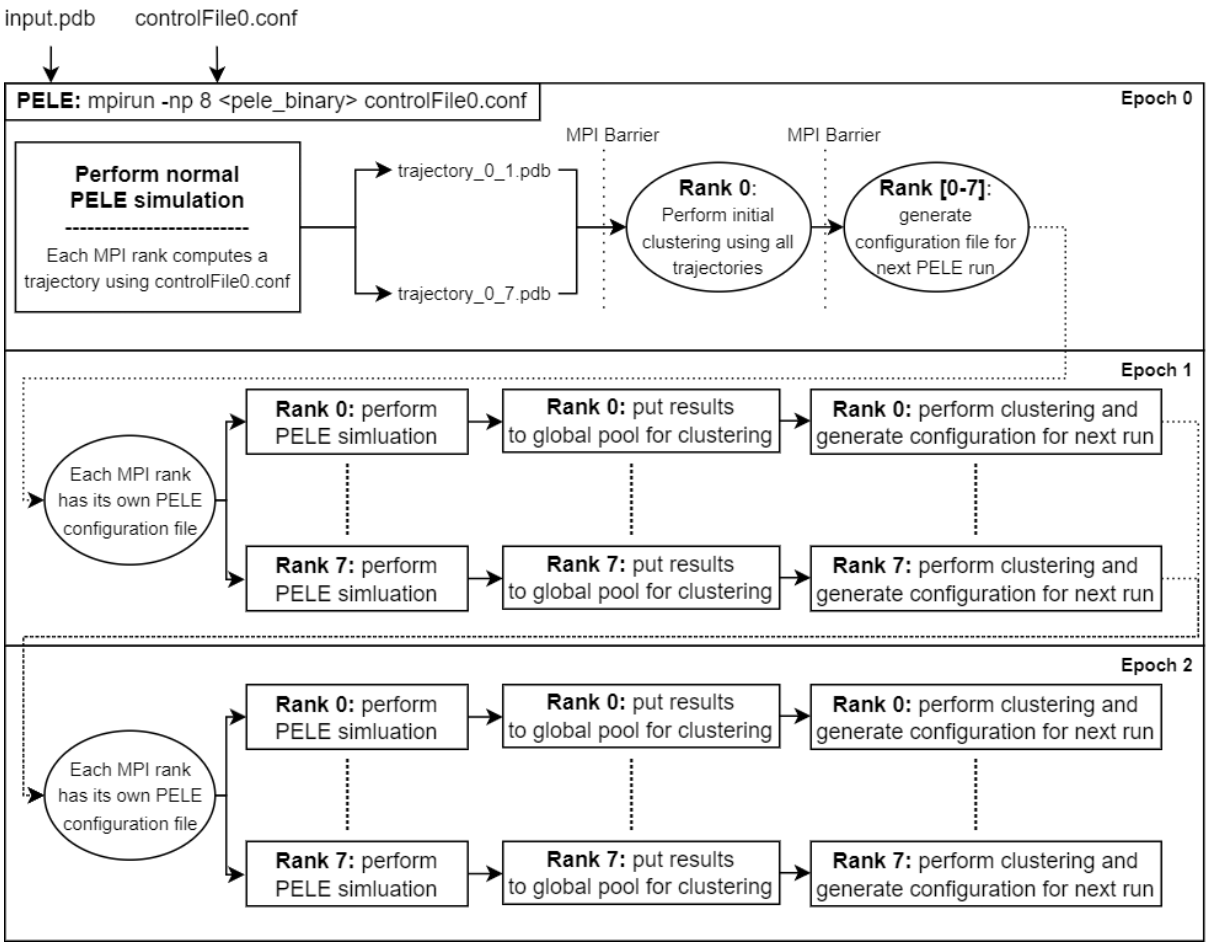


Figure 5.13: Conceptual workflow diagram of the new Relaxed Adaptive PELE implementation

To perform the clustering operations, the original Adaptive PELE python software had to be modified in order to disable the execution of PELE simulations. Furthermore, data transfer operations (like adding or getting data from the general file pool) had to be done under exclusivity to avoid operating with incomplete files that might be modified by other MPI ranks.

5.4.2 Results

After implementing this latest PELE version, we performed the same study done on section 5.3.3, using the same initial input files. This time, we will just compare the original Adaptive PELE against our new version. The obtained execution times have been summarized in figure 5.14.

In general, we can see lower execution times using this new version. However, it must also be noted that a direct comparison is difficult to achieve in this case, since both executions end up generating different intermediate trajectory files because the clustering operations performed do not use the same set of files. One important observation revealed by these

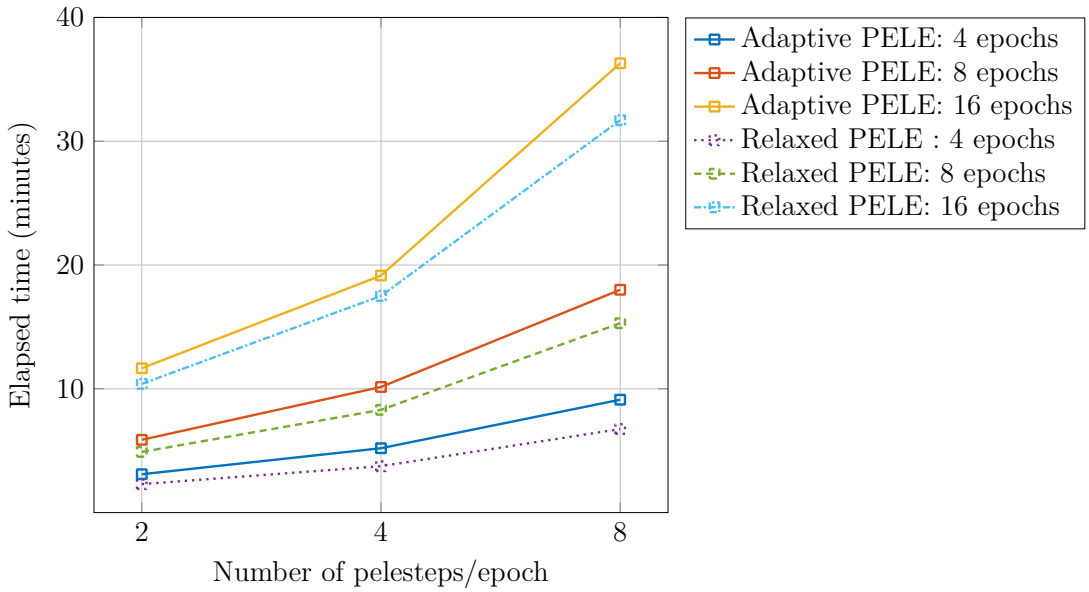


Figure 5.14: Execution times of Adaptive PELE and new PELE version with different parameters

results is that to perform the same number of pelesteps, it is faster to perform less epochs with more pelesteps performed on each one.

For example, if we compare the times needed to perform 32 pelesteps in total (2 pelesteps * 16 epochs, 4 pelesteps * 8 epochs and 8 pelesteps * 4 epochs), we will notice that execution time increases if we increment the number of epochs performed. This is true for both the original Adaptive PELE implementation and our relaxed version. However, we cannot evaluate which alternative produces qualitatively better results. This analysis should be performed by EAPM.

5.4.3 Conclusions

In conclusion, we have implemented a relaxed version of the Adaptive PELE workflow, which eliminates the need for a python wrapper and allows MPI ranks to perform epochs independently without having to wait for all other MPI ranks to finish their respective epochs. Furthermore, we have shown that this modification can be seamlessly integrated into the existing PELE software.

Our results show that our relaxed version of the Adaptive PELE workflow provides modest improvements in execution time compared to the original implementation. However, a qualitative analysis of the results obtained from this new version should be performed in order to compare the full benefits of this implementation against the current Adaptive PELE workflow.

6 PELE variants analysis on an ARM-based platform

In this chapter, we will discuss the deployment of the PELE software on an ARM-based HPC system. The motivation for this work stems from the fact that the developers of PELE have never tested its performance on an ARM-based platform before. Moreover, there is growing interest in deploying scientific software on cloud-based platforms, such as Amazon Web Services (AWS), which offers instances that are based on ARM processors [16]. Therefore, we decided to evaluate the performance of PELE on the Huawei cluster, which is an ARM-based HPC system hosted by BSC, in otder to determine its feasibility for deployment on such platforms.

In this chapter, we discuss the steps we took to deploy PELE on an ARM-based HPC system, and evaluate its performance on this platform using a baseline unmodified version and our PELE versions with the implemented optimization proposals we proposed in previous chapters. Finally, we also compare its performance against a x86-based system (Nord 3) to identify any significant differences.

6.1 Deployment

The Huawei cluster is mainly a research system, so it doesn't have too many dependency libraries already available for user applications. With the exception of the compiler, MPI implementation and python interpreter, the majority of the dependencies had to be installed. The general software stack used for PELE on ARM was the following:

- **GCC 11.2**: C/C++ compiler. The Intel compiler wasn't available.
- **OpenMPI 4.1.3**: MPI implementation. Intel MPI wasn't available.
- **OpenBLAS 0.3.21**: Linear algebra libraries. Intel MKL wasn't available, but there are other substitute libraries for ARM systems, like the ARM Performance Libraries [17]. However, OpenBLAS was chosen in order to be consistent with the experimentation results of Nord 3.
- **Boost 1.64.0**: General purpose libraries for C++.
- **Python 3.8.12**: Python interpreter. Needed for Adaptive PELE (which was also installed and modified).

Once all dependencies were compiled and installed, test runs were performed without further issue. After validating that the software worked, we proceeded to compile all the modified PELE versions explained in the previous chapter.

6.2 Performance analysis

6.2.1 Memory and time scalability

A similar memory scalability study to the one that was performed in Nord 3 has been repeated for this system. However, since each Huawei node has up to 128 cores, we have decided to perform the study until node saturation (assigning 1 MPI rank to each core). Please keep in mind that a Nord 3 node has 16 CPU cores and 32GB of memory, while a general purpose Huawei node has 128 CPU cores and 256GB of memory (notice that the memory/core relation is the same). The study has been done only for the large input described on section 4.3 and it computes 2 pelesteps. Figure 6.1 shows percentage of node memory usage for the original PELE and the improved PELE proposal, for Nord 3 and Huawei nodes.

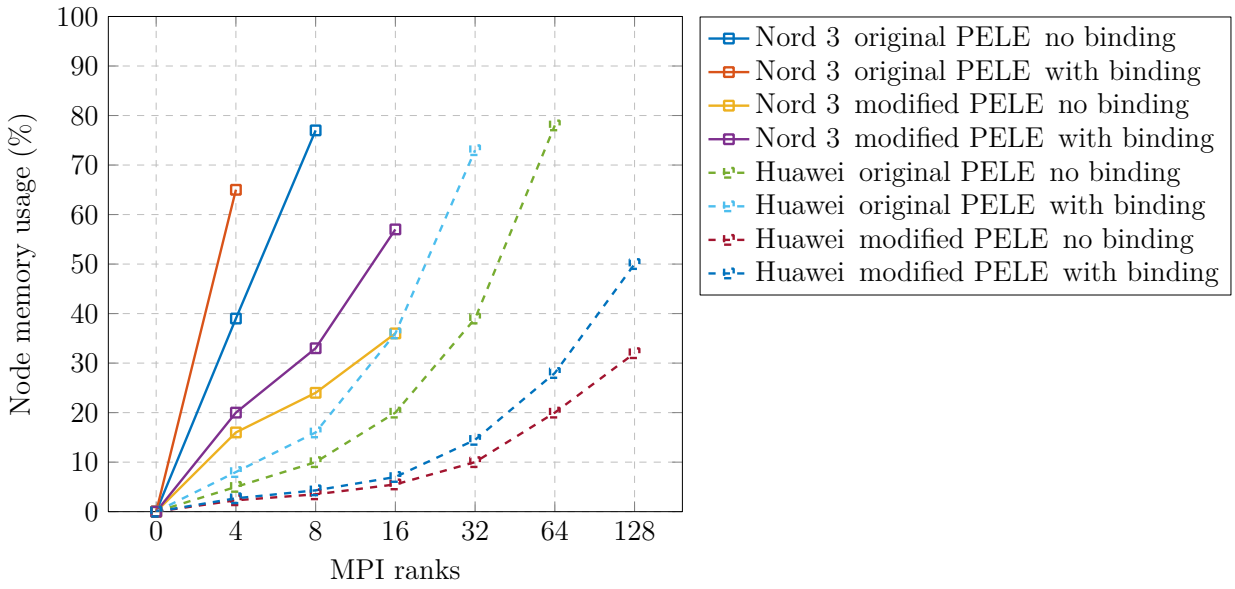


Figure 6.1: Node's occupation of original and memory-changed PELE through multiple MPI ranks

Missing data points are because the execution couldn't be performed using on node due to memory constraints (or in the case of Nord 3, because a node only has 16 CPU cores). On Huawei, we can see that the behavior is similar to Nord3: the original PELE version is not able to perform an execution using the whole node. Without computing binding energy, the original version of PELE is only able to perform an execution of up to 64 MPI ranks in a single node, effectively wasting half the node's CPU resources. Enabling the computation of binding energy makes things even worse, limiting the run to 32 MPI ranks and only using about 25% of the node's CPU cores.

When using the modified PELE binary with memory optimizations, the situation changes: single-node runs using 128 MPI ranks can now be performed without issues. If we are not computing binding energy, we only use about 30% of the node's total memory, and only 50% if we enable the computation of binding energy.

Therefore, the memory requirement to execute PELE with large PELE simulations are not a limitation for the data sets analyzed. This has been done with no additional cost on computational time. Figure 6.2 shows the execution time results for Huawei for the original PELE and the improved version.

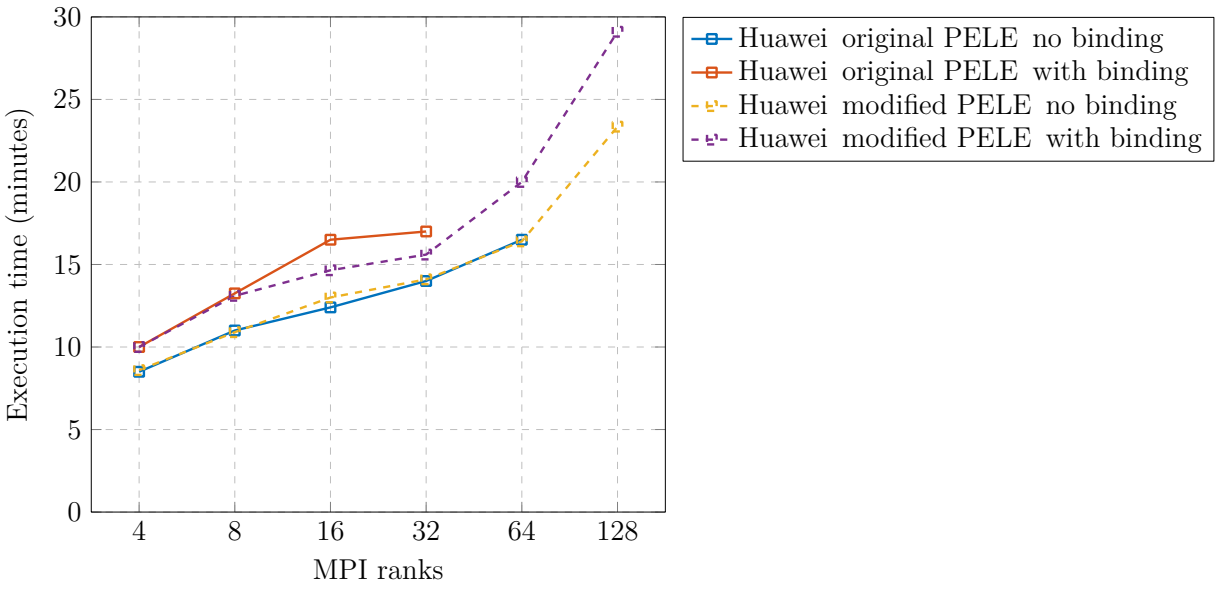


Figure 6.2: Execution time of original and memory-changed PELE through multiple MPI ranks

We can observe that runs with binding energy enabled also take more time than their counterparts without binding energy computation. Another interesting fact is that our modified PELE version also takes less time than the original version in all cases. However, execution time increases for all cases as long as we keep increasing the amount of MPI ranks. Since these times have been obtained with PELE versions that still rely on the master-slave paradigm, we decided to also time our new version with independent workers that do not use a master rank and do not require communications after each pelestep.

However, the differences in execution times were negligible. These time increases could come from multiple sources. Our suspected potential reasons are the following:

- **Resource conflicts**: the runs are performed in a single node. More MPI ranks must share the same set of resources.
- **Increased cost of initialization**: all MPI worker ranks still need to initialize its initial state using communication against the first rank. More MPI ranks may increase communication congestion.
- **Computationally costly simulations**: all MPI ranks perform randomized simulations, so there is the possibility that by increasing the amount of MPI ranks, we are introducing simulations that are more costly than the others.

Further analysis would be needed to either confirm or reject these hypotheses.

6.2.2 Single precision floating point

In this section, we test how our single-precision floating point versions of PELE performs on the Huawei cluster. For that, we have replicated the same executions performed on section 5.2.2, where we used the Nord 3 system.

Here we have decided to perform the same execution, where we will compare the original double-precision PELE version, our naive single-precision implementation, and a single-precision version with some of its internal constants being tuned to be aware of the single-precision range. These runs have been performed using the large input and without the computation of binding energy. We have also done the same runs with different amounts of pelesteps. All runs have been performed using serial binaries. The results can be seen in figure 6.3.

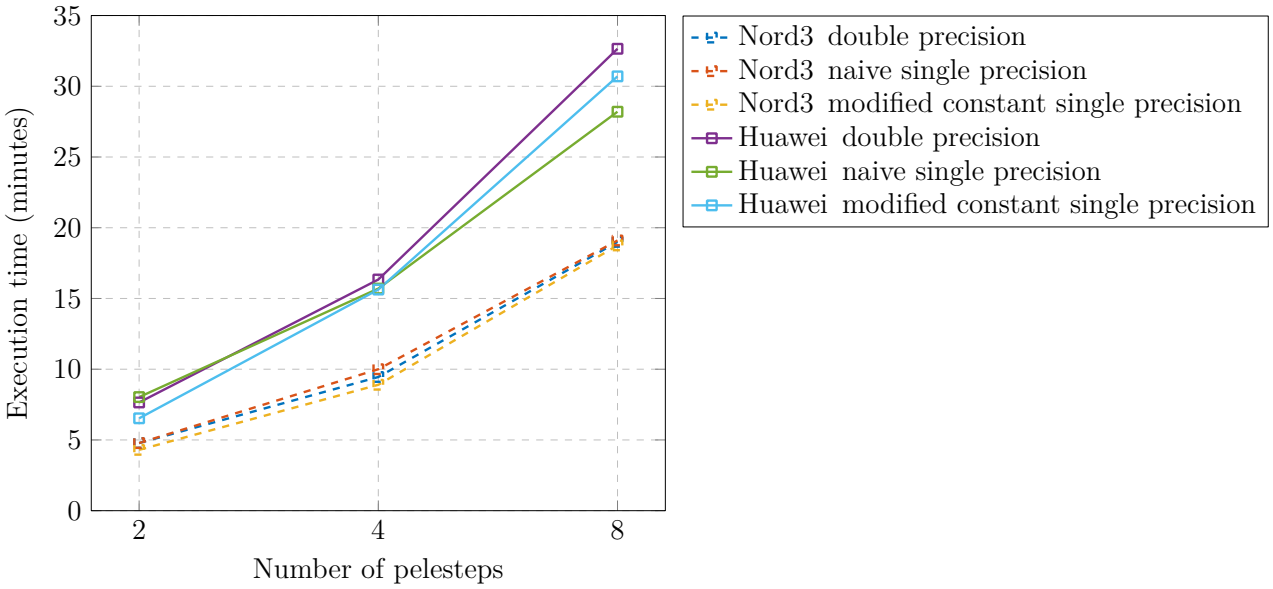


Figure 6.3: Execution time of serial PELE with different floating point implementations

Here we can see that the execution times of Huawei's runs are in general slower than the executions performed on Nord3. One interesting observation of these results is that the single-precision version of PELE with modified constants doesn't always provide better execution times than the naive single-precision implementation. For example, Huawei's execution time for 8 pelesteps using the naive single-precision implementation is faster than its modified constants counterpart.

We observe the same situation seen on Nord 3 (concretely, in listing 5.11). The number of iterations performed in the truncated Newton algorithm used by PELE can change from version to version, so the amount of computation performed for each run is not the same. This behavior, alongside the validity of the obtained simulation reports and trajectories, should be analyzed by PELE developers in the future.

6.2.3 Adaptive PELE

Finally, we have tested Adaptive PELE and all our standalone PELE versions that implement a similar workflow. We have repeated the executions performed in section 5.3.3 and 5.4.2, using the same input PDB files and control files (with the exception of the relaxed adaptive PELE implementation, which requires modified control files). The executions have been performed using 8 MPI ranks, with the exception of the original Adaptive PELE, which has been performed using 9 MPI ranks to compensate for the loss of a worker rank. For these executions, we obtained the following results:

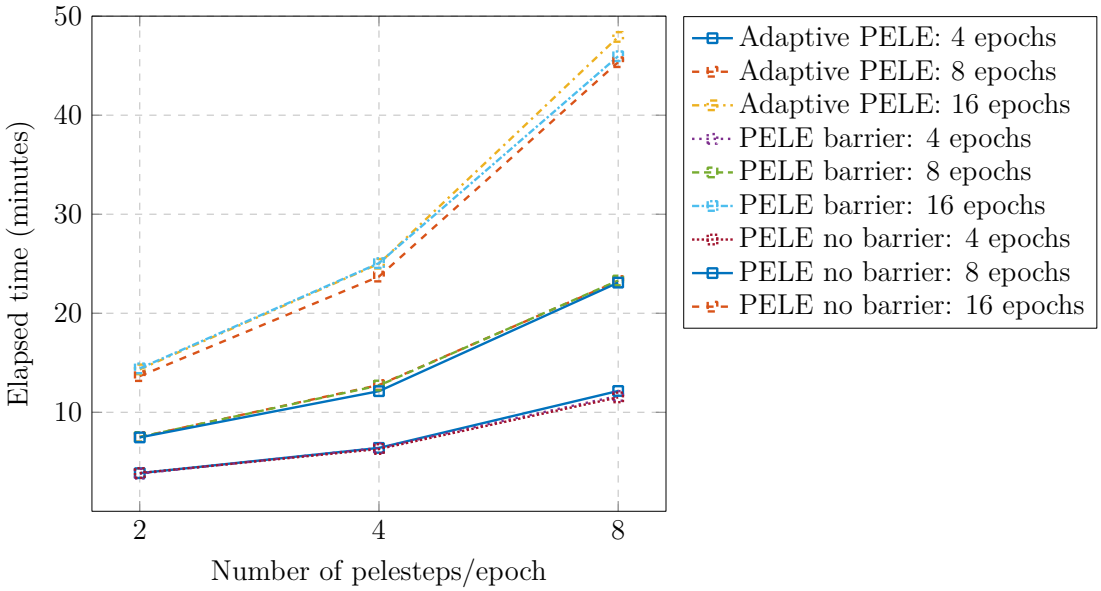


Figure 6.4: Execution times of Adaptive PELE and new PELE versions with different parameters

We can see that the relationship between the original Adaptive PELE and our versions with and without barriers is similar to what we saw in section 5.3.3, with the difference that in Huawei we are getting slower execution times. We can also see that the difference in execution times between the original Adaptive PELE and our barrier and non-barrier versions is very small, especially with the executions that perform 4 and 8 epochs.

To check why that is the case, we have obtained traces of the barrier and non-barrier executions performed with 2 pelesteps and 4 epochs, which can be seen in figure 6.5. The visualization at the top is for the version with barriers, and the visualization at the bottom is for the ideal version without barriers.

In the top execution trace, we can see 4 clear sections, each being an execution epoch. In this case, we can see that the similarities in execution times are because a certain MPI rank (second row in the top visualization) is always the last one to finish its computation on all epochs. Being that the case, the execution doesn't benefit from not having barriers, since all ranks will still end up still waiting at the end. This has been an example of the

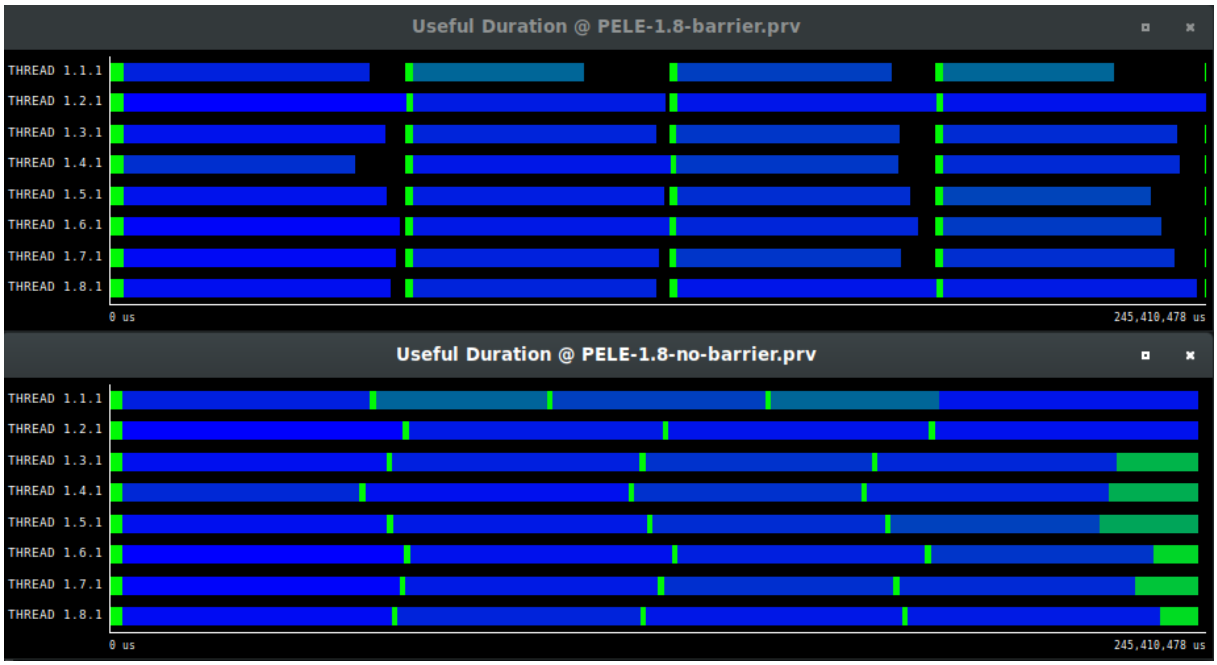


Figure 6.5: Paraver visualizations of PELE with and without barriers: 8 MPI ranks, 2 pelesteps, 4 epochs

worst case scenario, where an ideal implementation would still not see an improvement in execution time.

We have also compared the execution times of the original Adaptive PELE against our new relaxed implementation (see section 5.4). The results have been summarized in figure 6.6, and the execution parameters are the same ones used for the executions seen in figure 6.4.

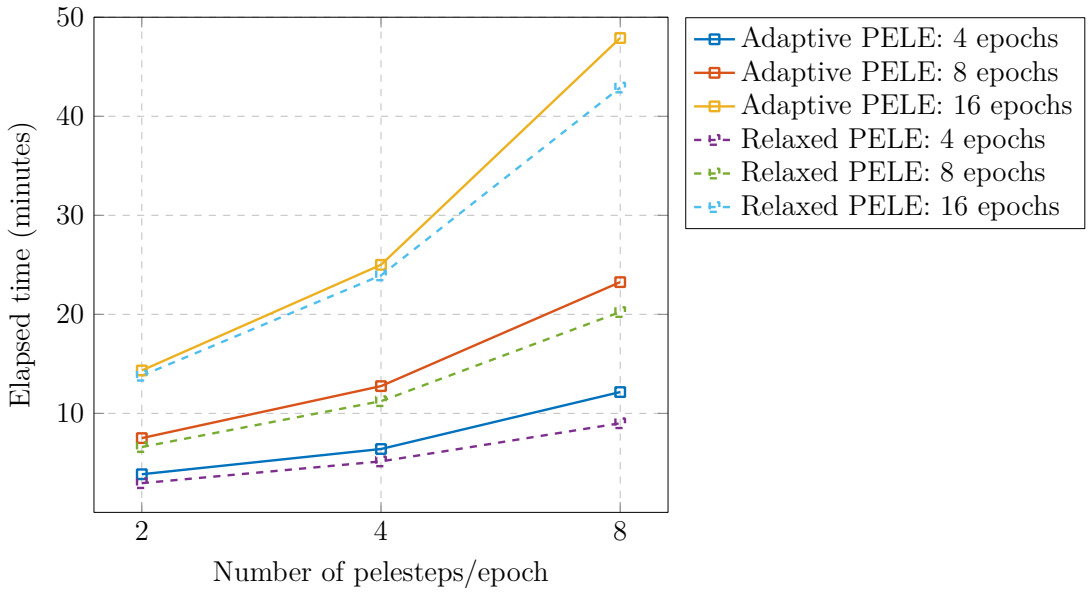


Figure 6.6: Execution times of Adaptive PELE and new relaxed Adaptive PELE using multiple parameters

In the figure, we can see that our new version is providing results with notably lower execution times in all cases. However, there is an important detail to take into account:

despite performing the same amount of epochs and pelesteps, both versions are not performing the exact same computation. Our new relaxed Adaptive PELE starts using the same input and configuration, but as its execution progresses, each MPI rank performs clustering operations with all obtained trajectories available at the time, independently of the execution state of its MPI worker peers.

Finally, we have also compared these execution time results against execution times obtained on Nord 3. In figure 6.7 we can see the obtained results.

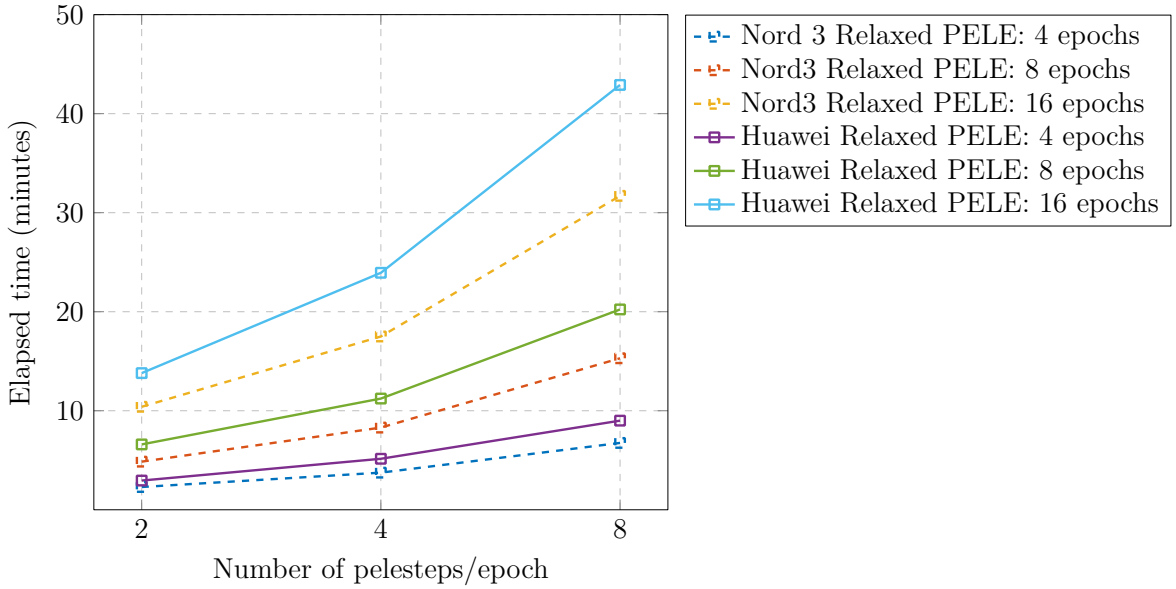


Figure 6.7: Execution times of new relaxed Adaptive PELE on Nord 3 and Huawei, using multiple parameters

Here we can see that if we compare executions using the same parameters, execution times are significantly slower when using Huawei. This was expected, since serial executions seen in figure 6.3 already showed that Nord3 executions are faster.

6.3 Conclusions

In conclusion, we have successfully deployed all developed versions of the PELE software on the Huawei cluster and tested their performance on this platform. Our results show that the Huawei cluster provides slower execution times compared to the Nord3 cluster, which was used as a reference for this study. However, we have demonstrated that PELE can be successfully deployed on an ARM-based HPC system, which is of great interest for future cloud-based deployments of the software.

Overall, our work provides valuable insights into the performance of the PELE software on ARM-based HPC systems and highlights potential areas for future research and development. We hope that our findings will contribute to the ongoing efforts to optimize the performance of PELE and enable its deployment on a wider range of computing platforms, including those based on ARM processors.

7 Final conclusions

This work has made significant contributions to the ongoing development of the PELE software, with a particular focus on identifying and addressing areas for improvement. Our analysis of the software enabled us to identify several key areas where optimization and parallelization strategies could be applied, and we have successfully implemented a number of proposals in these areas.

One of the achievements of this work has been the successful deployment of PELE on an ARM platform, specifically the Huawei cluster at the Barcelona Supercomputing Center. This demonstrates the software's versatility and adaptability, and opens up new possibilities for running simulations on a wider range of computing platforms.

The most important area of improvement has been the reduction of PELE's memory usage through the detection and correction of inefficient data structures. This optimization has enabled the execution of larger simulations that were previously not possible due to memory constraints, and also effectively removes the need of over-allocation of compute resources to meet memory requirements for feasible simulations.

We were also able to produce a version of PELE that uses single-precision floating point operations, which further reduces memory usage. However, this version didn't present benefits in execution time due to an increase in iterations needed for the convergence of some of PELE's algorithms. While this version needs further validation by PELE developers to ensure the accuracy and reliability of its results, it represents an important step forward in making the software even more efficient and flexible.

In addition to these improvements, we have also made significant changes to the MPI design of PELE. By moving away from the master-slave paradigm and enabling independent simulations between MPI ranks, we have enhanced the software's parallelization capabilities. These changes have also enabled the implementation of a proof of concept for reducing the effects of MPI load imbalance, which mimics a similar execution model of Adaptive PELE. This potentially opens up new avenues for optimizing PELE's performance on parallel computing systems. Finally, we have presented a proposal for a relaxed variant of Adaptive PELE, which has been implemented and tested.

Overall, this work has demonstrated the importance of ongoing research and development efforts to enhance the performance and capabilities of molecular dynamics simulation

software such as PELE. By identifying and addressing key areas for improvement, we have made strides towards making the software even more efficient and effective, and opened up new possibilities for running simulations on a wider range of computing platforms.

Bibliography

[1] Kenneth W. Borrelli, Andreas Vitalis, Raul Alcantara, and Victor Guallar. PELE: protein energy landscape exploration. a novel monte carlo based technique. *Journal of Chemical Theory and Computation*, 1(6):1304–1311, October 2005.

[2] Joan F. Gilabert, Daniel Lecina, Jorge Estrada, and Victor Guallar. Monte carlo techniques for drug design: The success case of PELE. In *Biomolecular Simulations in Structure-Based Drug Discovery*, pages 87–103. Wiley-VCH Verlag GmbH & Co. KGaA, December 2018.

[3] User guides for hpc systems hosted at bsc. `https://www.bsc.es/supportkc/`.

[4] General guide for bsc's hpc portal. `https://www.bsc.es/supportkc/docs-utilities/hpc_portal`.

[5] Main page from bsc's performance analysis tools. `https://tools.bsc.es`.

[6] Massif manual for memory profiling. `https://valgrind.org/docs/manual/ms-manual.html`.

[7] Callgrind manual for software calls profiling. `https://valgrind.org/docs/manual/cl-manual.html`.

[8] Gdb's documentation page. `http://gnu.ist.utl.pt/software/gdb/documentation/`.

[9] Ddt's documentation page. `https://developer.arm.com/documentation/101136/22-1-3/DDT`.

[10] Bsc's web page for electronic and atomic protein modeling research group. `https://www.bsc.es/discover-bsc/organisation/scientific-structure/electronic-and-atomic-protein-modeling-eapm`.

[11] Intel's reference page for oneapi's mkl. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-documentation.html`.

[12] Openblas reference page. `https://www.openblas.net/`.

[13] Daniel Lecina, Joan F. Gilabert, and Victor Guallar. Adaptive simulations, towards interactive protein-ligand modeling. *Scientific Reports*, 7(1), August 2017.

[14] Gromacs reference manual about single and mixed precision. `https://manual.gromacs.org/current/reference-manual/definitions.html#mixed-or-double-precision`.

[15] Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, 1985.

[16] Amazon's reference page detailing aws ec2 instances for cloud computing. `https://aws.amazon.com/es/ec2/instance-types/`.

[17] Arm's reference page for the arm performance libraries. `https://developer.arm.com/Tools%20and%20Software/Arm%20Performance%20Libraries`.

A PELE configuration files

A.1 PELE's configuration file for small input

```
{
  "licenseDirectoryPath" : "/gpfs/projects/bsc72/PELE++/license/",
  "simulationLogPath" : "out/logFile.txt",
  "Initialization" : {
     "allowMissingTerminals" :true,
     "ForceField" : "OPLS2005",
     "Complex": { "files" : [ {"path" : "1ZNK_complete.pdb" } ] },
     "Solvent" : { "ionicStrength" : 0.15, "solventType" : "VDGBNP", "useDebyeLength" :
    true }
   },
   "verboseMode" : false,
   "commands" : [
      {
         "commandType" : "peleSimulation",
         "RandomGenerator" : { "seed" : 1995 },
         "selectionToPerturb" : { "chains" : { "names" : [ "L" ] } },
         "PELE_Output" : {
            "savingFrequencyForAcceptedSteps" : 1,
            "savingMode" : "savingTrajectory",
            "reportPath" : "out/reports/report",
            "trajectoryPath" : "out/trajectories/trajectory.pdb"
         },
         "PELE_Parameters" : {
                "anmFrequency" : 2,
                "sideChainPredictionFrequency" : 2,
            "waterPerturbationFrequency": 1,
                "minimizationFrequency" : 1,
                "sideChainPredictionRegionRadius" : 4,
                "perturbationCOMConstraintConstant" : 5,
                "activateProximityDetection": true,
                "temperature": 1000,
                "numberOfPeleSteps": 10
         },
          "constraints":[
                       { "type": "constrainAtomToPosition", "springConstant": 2.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:1:_CA_" },
                       { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:10:_CA_" },
                       { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:20:_CA_" },
                       { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:30:_CA_" },
                       { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:40:_CA_" },
                       { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:50:_CA_" },
                       { "type": "constrainAtomToPosition", "springConstant": 0.5, "
```

```
    equilibriumDistance": 0.0, "constrainThisAtom": "A:60:_CA_" },
41                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:70:_CA_" },
42                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:80:_CA_" },
43                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:90:_CA_" },
44                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:100:_CA_" },
45                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:110:_CA_" },
46                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:120:_CA_" },
47                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:130:_CA_" },
48                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:140:_CA_" },
49                  { "type": "constrainAtomToPosition", "springConstant": 0.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:150:_CA_" },
50                  { "type": "constrainAtomToPosition", "springConstant": 2.5, "
    equilibriumDistance": 0.0, "constrainThisAtom": "A:157:_CA_" }
51                   ],
52        "Perturbation": {
53             "Box" : {
54                      "radius" : 6,
55                      "fixedCenter": [48.65, 24.8, 41.06],
56                      "type" : "sphericalBox"
57              },
58              "perturbationType":"naive",
59              "translationDirection": "steered",
60              "rotationAngles": "nonCoupled",
61              "parameters": {
62                  "numberOfStericTrials": 100,
63                  "steeringUpdateFrequency": 1,
64                  "overlapFactor": 0.65
65              }
66          },
67        "WaterPerturbation":
68        {
69            "Box" :
70            {
71                "radius" : 6,
72                "fixedCenter": [51, 23, 44],
73                "type" : "sphericalBox"
74            },
75            "watersToPerturb": { "links": { "ids": [ "W:1" ] } },
76            "parameters":
77            {
78                "temperature": 5000,
79                "overlapFactor": 0.78,
80                "numberOfStericTrials": 100000
81            }
82        },
83        "ANM" : {
84          "algorithm": "CARTESIANS", "nodes": { "atoms": { "names": [ "_CA_" ]} },
85          "linksToOmit" : { "links" : { "ids" : [ "W:1" ] }},
86          "ANMMinimizer" : {
87             "algorithm" : "TruncatedNewton",
88             "parameters" : {
```

```
                "MaximumMinimizationIterations" : 1,
                "MaximumNewtonIterations" : 25,
                "MinimumRMS" : 0.25,
                "alphaUpdated" : false,
                "nonBondingListUpdatedEachMinStep" : false
             }
          },
          "options" : {
             "directionGeneration" : "random",
             "modesMixingOption" : "mixMainModeWithOthersModes",
             "pickingCase" : "RANDOM_MODE"
          },
          "parameters" : {
             "displacementFactor" : 0.75,
             "eigenUpdateFrequency" : 1000000,
             "mainModeWeightForMixModes" : 0.75,
             "modesChangeFrequency" : 3,
             "numberOfModes": 6,
             "relaxationSpringConstant" : 0.5
          }
       },
       "SideChainPrediction" : {
          "algorithm" : "zhexin",
          "parameters" : { "discardHighEnergySolutions" : false, "resolution": 30, "
   randomize" : false, "numberOfIterations": 1 }
       },
       "Minimizer" : {
          "algorithm" : "TruncatedNewton",
          "parameters" : { "MinimumRMS" : 0.1, "alphaUpdated" : false, "
   nonBondingListUpdatedEachMinStep" : true }
       },

       "PeleTasks" : [
          {
             "metrics" : [
                      { "type": "bindingEnergy",
                         "boundPartSelection": { "chains": { "names": ["L"] } }
                      },
                      { "type": "sasa",
                         "tag": "sasaLig",
                         "selection": { "chains": { "names": ["L"] } }
                      },
                      {
                         "type":"com_distance",
                         "tag":"Lig-Phe38_distance",
                         "selection_group_1":{
                            "atoms": { "ids":["A:38: O  "]}
                         },
                         "selection_group_2":{
                            "atoms": { "ids":["L:1: H20"]}
                         }
                      },
                      {
                          "type" : "com_distance",
                          "tag" : "binding_site_distance",
                          "selection_group_1" :
                          {
                          "links" : { "ids" : [ "A:101", "A:38", "A:103", "A:40", "A
   :105", "A:45", "A:120", "A:116", "A:54", "A:90", "A:56", "A:92" ] }
```

```
                            },
                            "selection_group_2" :
                            {
                                "links" : { "ids" : [ "L:1" ] }
                            }
                        },
                        { "tag" : "rand", "type" : "random" },
                        { "tag" : "rand1", "type" : "random" },
                        { "tag" : "rand2", "type" : "random" },
                        { "tag" : "rand3", "type" : "random" }
                    ],
                    "parametersChanges" : [
                        { "ifAnyIsTrue": [ "rand >= 0.5" ],
                            "doThesechanges": { "Perturbation::parameters": { "
    rotationScalingFactor": 0.05 } },
                            "otherwise": { "Perturbation::parameters": { "
    rotationScalingFactor": 0.25 } }
                        },
                        {
                          "ifAnyIsTrue": [ "rand1 >= 0.5" ],
                            "doThesechanges": { "Perturbation::parameters": { "
    translationRange": 1.5 } },
                            "otherwise": { "Perturbation::parameters": { "
    translationRange": 0.75 } }
                        },
                        {
                          "ifAnyIsTrue": [ "rand2 >= 0.5" ],
                            "doThesechanges": { "Perturbation::parameters": { "
    steeringUpdateFrequency": 0 } },
                            "otherwise": { "Perturbation::parameters": { "
    steeringUpdateFrequency": 1 } }
                        },
                        {
                            "ifAnyIsTrue": [ "rand3 <= 0.4" ],
                            "doThesechanges": { "WaterPerturbation::parameters": { "
    translationRange": 2.0 } },
                            "otherwise": { "WaterPerturbation::parameters": { "
    translationRange": 4.0} }
                        },
                        {
                            "ifAnyIsTrue": [ "rand3 >= 0.85" ],
                            "doThesechanges": { "WaterPerturbation::parameters": { "
    translationRange": 5.0 } }
                        }
                    ]
            }
         ]
        }
        ]
}
```

Listing A.1: Contents of pele.conf file for small input

A.2 PELE's configuration file for large input

```
{
  "licenseDirectoryPath" : "/gpfs/projects/bsc72/PELE++/license/",
  "simulationLogPath" : "output/logFile.txt",
  "Initialization" : {
     "allowMissingTerminals": true,
     "ForceField" : "OPLS2005",
     "MultipleComplex": [
{ "files" : [ { "path" : "system_in_processed.pdb" } ] } ],
     "Solvent" : {
        "ionicStrength" : 0.15, "solventType" : "VDGBNP", "useDebyeLength" : true }
   },
   "verboseMode": false,
   "commands" : [
      {
         "commandType" : "peleSimulation",
         "RandomGenerator" : { "seed" : 1234567 },
         "selectionToPerturb" : { "chains" : { "names" : [ "L" ] } },
         "PELE_Output" : {
            "savingFrequencyForAcceptedSteps" : 1,
            "savingMode" : "savingTrajectory",
            "reportPath": "output/report",
            "trajectoryPath": "output/trajectory.pdb"

         },
         "PELE_Parameters" : {
            "anmFrequency" : 0,
            "sideChainPredictionFrequency" : 2,
            "minimizationFrequency" : 1,
            "waterPerturbationFrequency": 1,
            "perturbationCOMConstraintConstant" : 0,
            "sideChainPredictionRegionRadius" : 6,
            "activateProximityDetection": true,
            "temperature": 1500,
            "numberOfPeleSteps": 2
         },
          "constraints":[
{ "type": "constrainAtomToPosition", "springConstant": 5.0, "equilibriumDistance": 0.0, "
    constrainThisAtom": "A:1006:_CA_" },
... // Excluded constraints for simplicity's sake
{ "type": "constrainAtomToPosition", "springConstant": 5.0, "equilibriumDistance": 0.0, "
    constrainThisAtom": "D:1267:_CA_" }
],
          "Perturbation": {
                "Box": {
                    "type": "sphericalBox",
                    "radius": 10,
                    "fixedCenter": [-4.562, -44.69, -24.847]
                },
                "perturbationType":"naive",
                "translationDirection": "steered",
                "rotationAngles": "nonCoupled",
                "parameters": {
                    "numberOfStericTrials": 500,
                    "steeringUpdateFrequency": 0,
                    "overlapFactor": 0.65
                }
            },
         "ANM" : {
```

```
57            "algorithm": "CARTESIANS", "nodes": { "atoms": { "names": [ "_CA_" ]} },
58            "ANMMinimizer" : {
59               "algorithm" : "TruncatedNewton",
60               "parameters" : {
61                  "MaximumMinimizationIterations" : 1,
62                  "MaximumNewtonIterations" : 25,
63                  "MinimumRMS" : 0.2,
64                  "alphaUpdated" : false,
65                  "nonBondingListUpdatedEachMinStep" : false
66               }
67            },
68            "options" : {
69               "directionGeneration" : "random",
70               "modesMixingOption" : "mixMainModeWithOthersModes",
71               "pickingCase" : "RANDOM_MODE"
72            },
73            "parameters" : {
74               "displacementFactor" : 0.75,
75               "eigenUpdateFrequency" : 1000000,
76               "mainModeWeightForMixModes" : 0.75,
77               "modesChangeFrequency" : 4,
78               "numberOfModes": 6,
79               "relaxationSpringConstant" : 0.5
80            }
81         },
82         "SideChainPrediction" : {
83            "algorithm" : "zhexin",
84            "parameters" : { "discardHighEnergySolutions" : false, "resolution": 30, "
   randomize" : false, "numberOfIterations": 1 }
85         },
86         "Minimizer" : {
87            "algorithm" : "TruncatedNewton",
88            "parameters" : { "MinimumRMS" : 0.2, "alphaUpdated" : false, "
   nonBondingListUpdatedEachMinStep" : true }
89         },
90         "PeleTasks" : [
91            {
92               "metrics" : [
93                        { "type": "sasa",
94                           "tag": "sasaLig",
95                           "selection": { "chains": { "names": ["L"] } }
96                        },
97                                 {
98                                    "type": "rmsd",
99                                    "Native": {
100                                   "path":
101                                   "system_in_processed.pdb" },
102                                   "selection": { "chains": { "names": [ "L" ] } },
103                                   "includeHydrogens": false,
104                                   "doSuperposition": false,
105                                   "tag" : "ligandRMSD"
106                                },
107                  { "tag" : "rand", "type" : "random" },
108                  { "tag" : "rand4", "type" : "random" },
109                  { "tag" : "rand3", "type" : "random" },
110                  { "tag" : "rand2", "type" : "random" },
111                  { "tag" : "rand1", "type" : "random" }
112               ]
113          ,
```

```
                "parametersChanges" : [
                    { "ifAnyIsTrue": [ "rand >= .5" ],
                        "doThesechanges": { "Perturbation::parameters": { "
    rotationScalingFactor": 0.05 } },
                        "otherwise": { "Perturbation::parameters": { "
    rotationScalingFactor": 0.15 } }
                    },
                    {
                       "ifAnyIsTrue": [ "rand1 >= 0.40" ],
                        "doThesechanges": { "Perturbation::parameters": { "
    translationRange": 2.0 } },
                        "otherwise": { "Perturbation::parameters": { "translationRange":
     2.0 } }
                  },
                  {
                    "ifAnyIsTrue": [ "rand3 >= 0.10" ],
                      "doThesechanges": { "Perturbation::parameters": { "
    steeringUpdateFrequency": 1, "numberOfTrials" : 10 } },
                     "otherwise": { "Perturbation::parameters": { "
    steeringUpdateFrequency": 0, "numberOfTrials" : 25 } }
                  },
                  {
                    "ifAnyIsTrue": [ "sasaLig <= 0.15" ],
                       "doThesechanges": { "Pele::parameters": { "
    perturbationCOMConstraintConstant" : 0.25 }, "Perturbation::parameters": { "
    translationRange": 1.0 }},
                       "otherwise": { "Pele::parameters": { "
    perturbationCOMConstraintConstant" : 1.0 } }
                  },
                  {
                    "ifAnyIsTrue": [ "sasaLig >= 0.75" ],
                       "doThesechanges": { "Pele::parameters": { "
    perturbationCOMConstraintConstant" : 10.0 }, "Perturbation::parameters": { "
    steeringUpdateFrequency": 1, "numberOfTrials" : 4 }},
                       "otherwise": { }
                  }
                ]
            }
         ]
        }
        ]
}
```

Listing A.2: Contents of pele.conf file for large input

A.3 Base adaptive configuration file for Adaptive PELE execution

```
{
   "generalParams": {
      "restart": true,
      "outputPath": "output",
      "initialStructures": [
         "complex_2.pdb"
      ]
   },
   "spawning": {
      "type": "inverselyProportional",
      "params": {
         "reportFilename": "report"
      }
   },
   "simulation": {
      "type": "pele",
      "params": {
         "iterations": 4,
         "peleSteps": 2,
         "processors": 8,
         "seed": 24083,
         "executable": "/gpfs/projects/bsc99/bsc99204/TFM/pele_compilation/nord3/
    bin_mpi_prod_newfrozen_newmpi_fix/PELE-1.8",
         "data": "/gpfs/projects/bsc72/PELE++/nord4/V1.8_pre/Data",
         "documents": "/gpfs/projects/bsc72/PELE++/nord4/V1.8_pre/Documents",
         "useSrun": false,
         "controlFile": "pele.conf"
      }
   },
   "clustering": {
      "type": "rmsd",
      "params": {
         "ligandResname": "LIG",
         "alternativeStructure": true
      },
      "thresholdCalculator": {
         "type": "heaviside",
         "params": {
            "values": [
               1.75,
               2.5,
               4,
               6
            ],
            "conditions": [
               1,
               0.6,
               0.4,
               0.0
            ]
         }
      }
   }
}
```

Listing A.3: Provided adaptive.conf for Adaptive PELE executions

A.4 Base PELE configuration file for Adaptive PELE execution

```
{
   "licenseDirectoryPath": "/gpfs/projects/bsc72/PELE++/license/",
   "Initialization": {
      "allowMissingTerminals": true,
      "ForceField": "OPLS2005",
      "Solvent": {
         "ionicStrength": 0.15,
         "solventType": "VDGBNP",
         "useDebyeLength": true
      },
      "MultipleComplex": [
         $COMPLEXES
      ]
   },
   "commands": [
      {
         "commandType": "peleSimulation",
         "randomGenerator": {
            "seed": $SEED
         },
         "selectionToPerturb": {
            "chains": {
               "names": [
                  "L"
               ]
            }
         },
         "PELE_Output": {
            "savingFrequencyForAcceptedSteps": 1,
            "savingMode": "savingTrajectory",
            "reportPath": "$OUTPUT_PATH/report",
            "trajectoryPath": "$OUTPUT_PATH/trajectory.xtc"
         },
         "PELE_Parameters": {
            "anmFrequency": 0,
            "sideChainPredictionFrequency": 2,
            "minimizationFrequency": 1,
            "PerturbationCOMConstraintConstant": 1.0,
            "sideChainPredictionRegionRadius": 6.0,
            "activateProximityDetection": true,
            "temperature": 1500.0,
            "numberOfPeleSteps": 2
         },
         "Perturbation": {
            "Box": {
               "type": "sphericalBox",
               "radius": 5.0,
               "fixedCenter": [
                  26.72313008770274,
                  6.051754793307583,
                  4.722083771208937
               ]
            },
            "perturbationType": "naive",
            "translationDirection": "steered",
            "rotationAngles": "nonCoupled",
            "parameters": {
               "overlapFactor": 0.65,
```

```
            "influenceRange": 3.0
         }
      },
      "ANM": {
         "algorithm": "CARTESIANS",
         "nodes": {
            "atoms": {
               "names": [
                  "_CA_"
               ]
            }
         },
         "ANMMinimizer": {
            "algorithm": "TruncatedNewton",
            "parameters": {
               "MinimumRMS": 0.2,
               "alphaUpdated": false,
               "nonBondingListUpdatedEachMinStep": false,
               "MaximumMinimizationIterations": 1,
               "MaximumNewtonIterations": 25
            }
         },
         "options": {
            "directionGeneration": "random",
            "modesMixingOption": "mixMainModeWithOthersModes",
            "pickingCase": "RANDOM_MODE"
         },
         "parameters": {
            "displacementFactor": 0.75,
            "eigenUpdateFrequency": 1000000,
            "mainModeWeightForMixModes": 0.75,
            "modesChangeFrequency": 4,
            "numberOfModes": 6,
            "relaxationSpringConstant": 0.5
         }
      },
      "SideChainPrediction": {
         "algorithm": "zhexin",
         "parameters": {
            "discardHighEnergySolutions": false,
            "resolution": 10,
            "randomize": false,
            "numberOfIterations": 1
         }
      },
      "Minimizer": {
         "algorithm": "TruncatedNewton",
         "parameters": {
            "MinimumRMS": 1.0,
            "alphaUpdated": false,
            "nonBondingListUpdatedEachMinStep": true,
            "energyDifference": 1.0,
            "MaximumMinimizationIterations": 1
         }
      },
      "PeleTasks": [
         {
            "metrics": [
               {
```

```
                    "boundPartSelection": {
                        "chains": {
                            "names": [
                                "L"
                            ]
                        }
                    },
                    "type": "bindingEnergy",
                    "tag": "bindingEnergy"
                },
                {
                    "selection": {
                        "chains": {
                            "names": [
                                "L"
                            ]
                        }
                    },
                    "type": "sasa",
                    "tag": "ligandSASA"
                },
                {
                    "Native": {
                        "path": "/home/bsc99/bsc99204/projects/bsc99204/TFM/adaptive_pele
/ricard_examples/adaptive_run/complex_2.pdb"
                    },
                    "includeHydrogens": false,
                    "doSuperposition": false,
                    "selection": {
                        "chains": {
                            "names": [
                                "L"
                            ]
                        }
                    },
                    "type": "rmsd",
                    "tag": "ligandRMSD"
                },
                {
                    "type": "random",
                    "tag": "rand1"
                },
                {
                    "type": "random",
                    "tag": "rand2"
                },
                {
                    "type": "random",
                    "tag": "rand3"
                },
                {
                    "type": "random",
                    "tag": "rand4"
                },
                {
                    "type": "random",
                    "tag": "rand5"
                }
            ],
```

```
176             "parametersChanges": [
177                {
178                   "ifAnyIsTrue": [
179                      "rand1 >= 0.0",
180                      "rand1 <= 0.5"
181                   ],
182                   "doThesechanges": {
183                      "Perturbation::parameters": {
184                         "rotationScalingFactor": 0.05
185                      }
186                   }
187                },
188                {
189                   "ifAnyIsTrue": [
190                      "rand1 >= 0.5",
191                      "rand1 <= 1.0"
192                   ],
193                   "doThesechanges": {
194                      "Perturbation::parameters": {
195                         "rotationScalingFactor": 0.2
196                      }
197                   }
198                },
199                {
200                   "ifAnyIsTrue": [
201                      "rand2 >= 0.0",
202                      "rand2 <= 0.5"
203                   ],
204                   "doThesechanges": {
205                      "Perturbation::parameters": {
206                         "translationRange": 0.5
207                      }
208                   }
209                },
210                {
211                   "ifAnyIsTrue": [
212                      "rand2 >= 0.5",
213                      "rand2 <= 1.0"
214                   ],
215                   "doThesechanges": {
216                      "Perturbation::parameters": {
217                         "translationRange": 1.0
218                      }
219                   }
220                },
221                {
222                   "ifAnyIsTrue": [
223                      "rand3 >= 0.0",
224                      "rand3 <= 0.5"
225                   ],
226                   "doThesechanges": {
227                      "Perturbation::parameters": {
228                         "steeringUpdateFrequency": 0
229                      }
230                   }
231                },
232                {
233                   "ifAnyIsTrue": [
234                      "rand3 >= 0.5",
```

```
                        "rand3 <= 1.0"
                    ],
                    "doThesechanges": {
                        "Perturbation::parameters": {
                            "steeringUpdateFrequency": 1
                        }
                    }
                },
                {
                    "ifAnyIsTrue": [
                        "rand3 >= 0.0",
                        "rand3 <= 0.5"
                    ],
                    "doThesechanges": {
                        "Perturbation::parameters": {
                            "numberOfTrials": 30
                        }
                    }
                },
                {
                    "ifAnyIsTrue": [
                        "rand3 >= 0.5",
                        "rand3 <= 1.0"
                    ],
                    "doThesechanges": {
                        "Perturbation::parameters": {
                            "numberOfTrials": 10
                        }
                    }
                }
            ]
        }
    ],
    "constraints": [
        {
            "type": "constrainAtomToPosition",
            "springConstant": 5,
            "equilibriumDistance": 0.0,
            "constrainThisAtom": "A:1:_CA_"
        },
        ... // Remaining constraint omitted for brevity
    ]
  }
 ],
 "verboseMode": true,
 "simulationLogPath": "$OUTPUT_PATH/logFile.txt"
}
```

Listing A.4: Provided pele.conf for Adaptive PELE executions

B Complimentary plots of PELE's memory analysis

B.1 Massif-visualizer memory plots of baseline PELE binaries

In figures B.1 and B.2 we can observe memory profile graphs (obtained from *massif-visualizer*) of a serial execution of a unmodified PELE binary using the small input without the computation of binding energy. They account for heap memory and total memory used by the process, respectively. In figures B.3 and B.4 we observe equivalent graphs, but this time from executions that also compute binding energy. All measurements have been performed on the Nord 3 system.

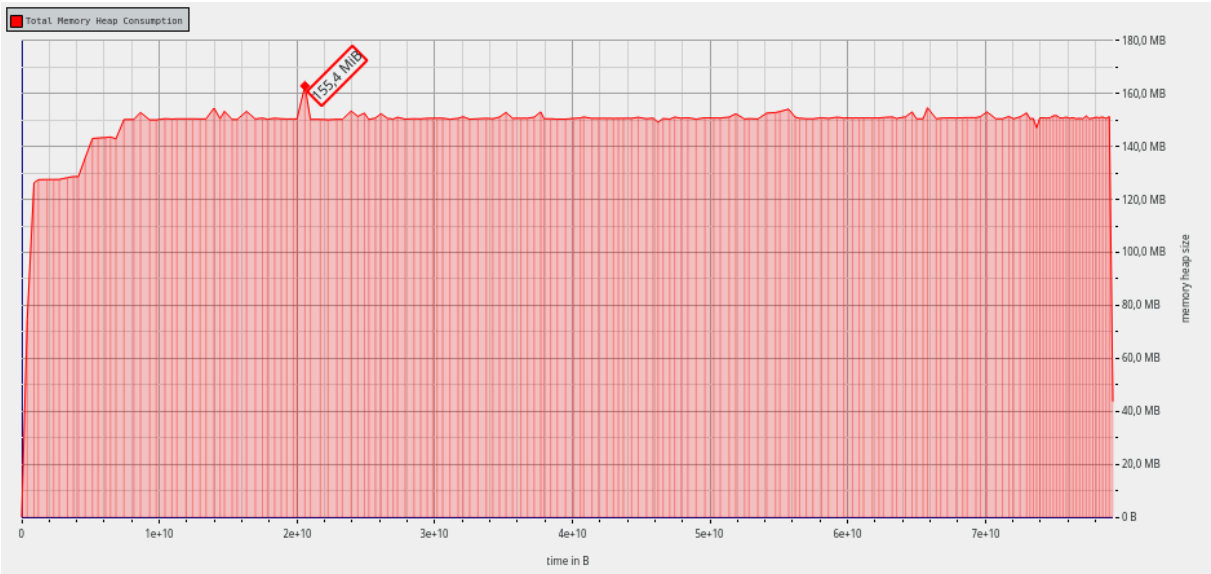


Figure B.1: Massif heap trace of a serial PELE execution, small input, without binding energy

An equivalent set of plots have been obtained for the executions that used the large input, which can be seen in figures B.5 - B.8.

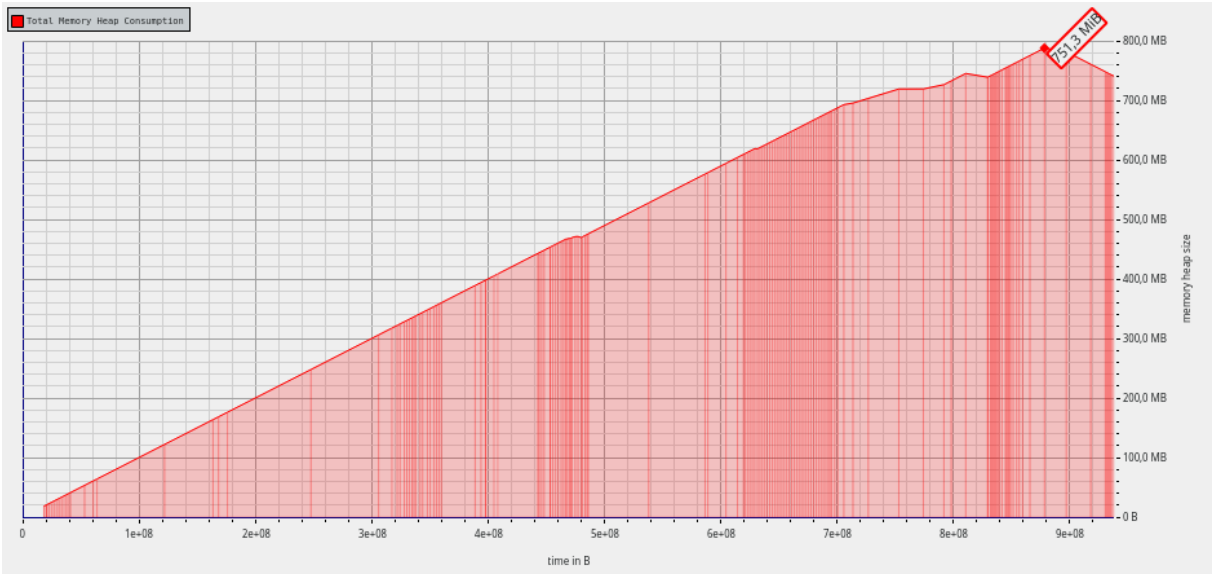


Figure B.2: Massif total trace of a serial PELE execution, small input, without binding energy

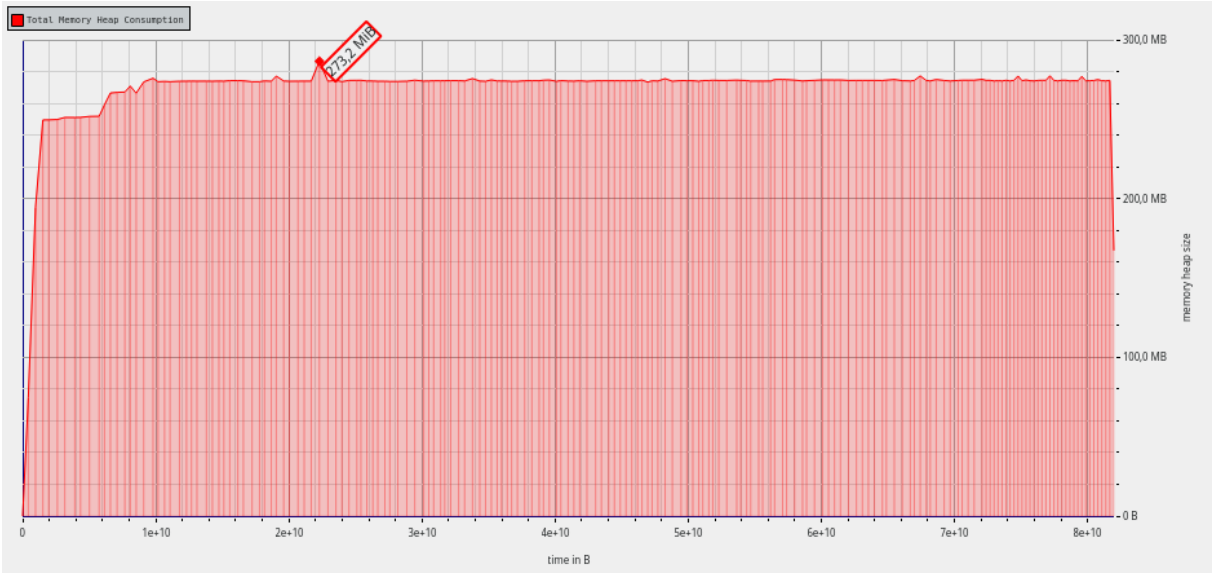


Figure B.3: Massif heap trace of a serial PELE execution, small input, with binding energy

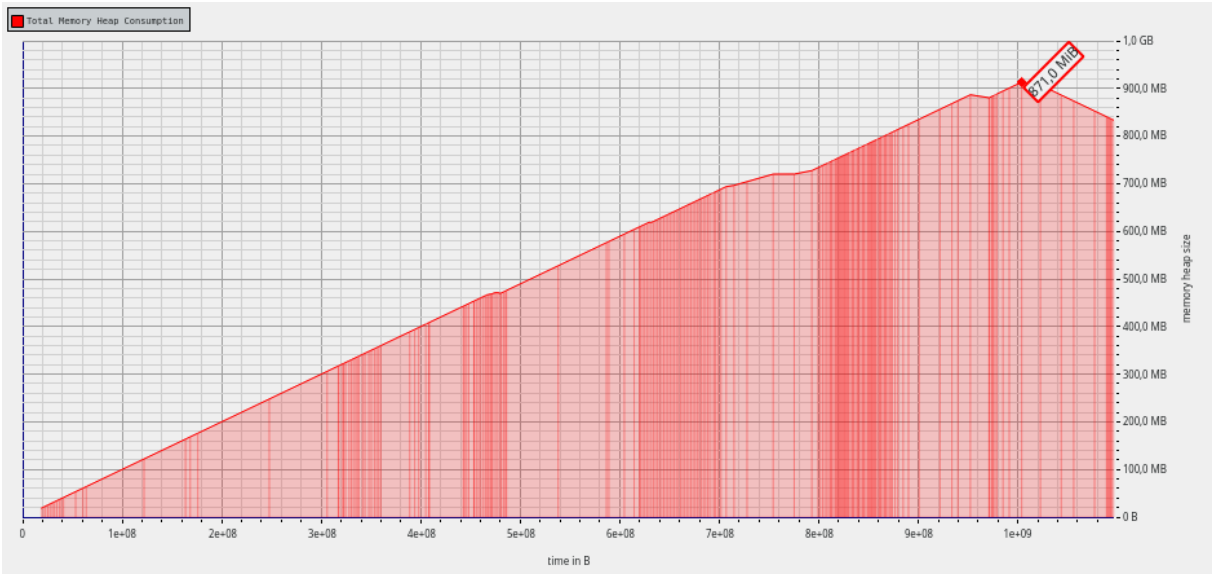


Figure B.4: Massif total trace of a serial PELE execution, small input, with binding energy

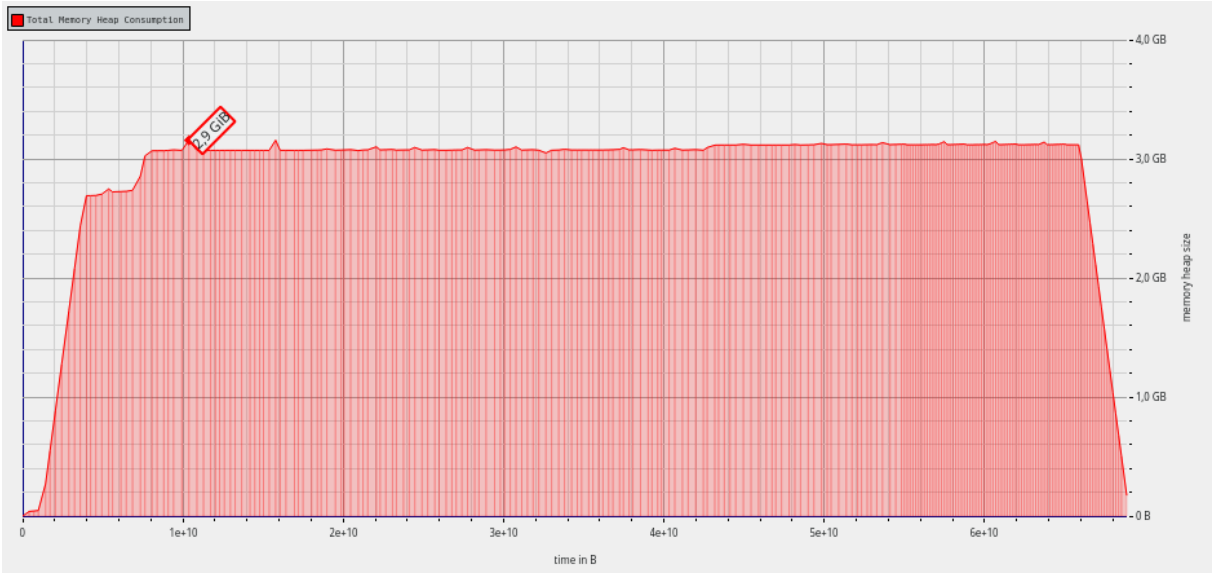


Figure B.5: Massif heap trace of a serial PELE execution, large input, without binding energy

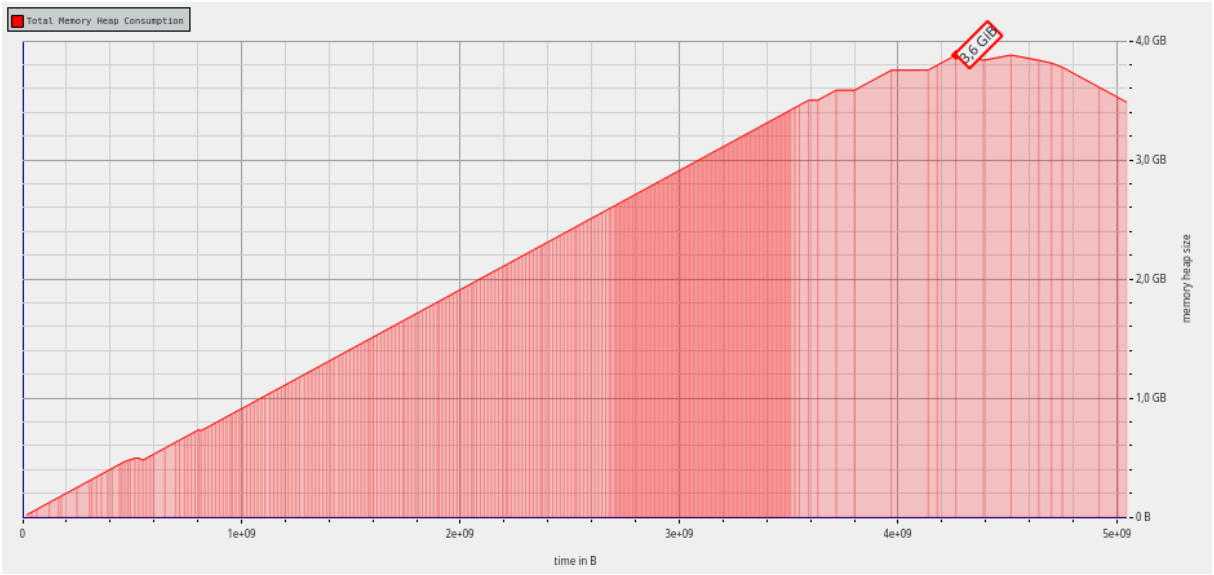


Figure B.6: Massif total trace of a serial PELE execution, large input, without binding energy

Figure B.7: Massif heap trace of a serial PELE execution, large input, with binding energy

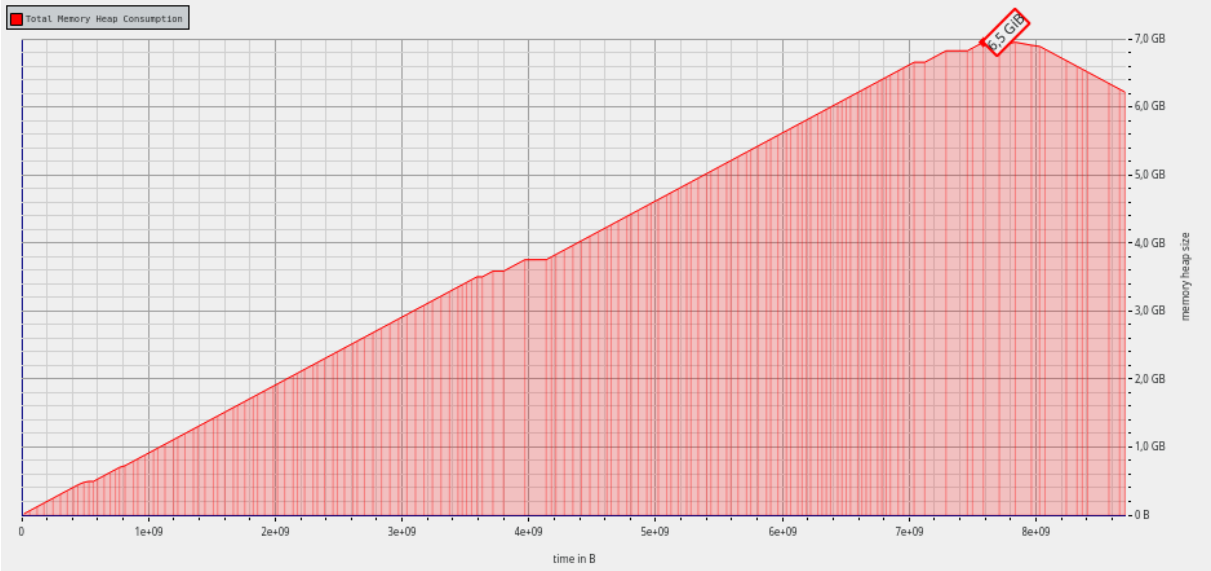


Figure B.8: Massif total trace of a serial PELE execution, large input, with binding energy