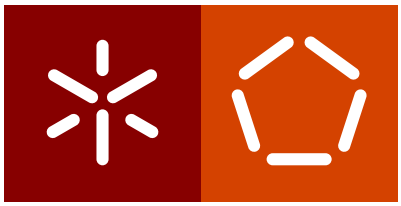
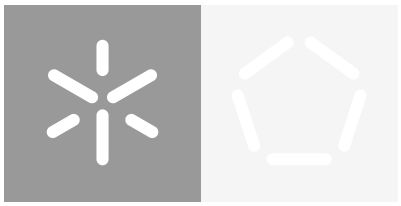


University of Minho
School of Engineering
Informatics Department

Sara Pereira

Improving the efficiency of the Energy-Split tool to compute the energy of very large molecular systems

August 2021



University of Minho
School of Engineering
Informatics Department

Sara Pereira

Improving the efficiency of the Energy-Split tool to compute the energy of very large molecular systems

Master dissertation
Master Degree in Integrated Masters in Computer Engineering

Dissertation supervised by
Alberto José Proença
André Pereira, Henrique Fernandes, Sérgio Sousa

August 2021

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

ACKNOWLEDGEMENTS

First, I must sincerely thank my supervisors Alberto José Proença and André Pereira, whose help and guidance were crucial to developing this thesis. Always very available and cooperative to assist me in overcoming the obstacles.

To Henrique Fernandes and Sérgio Sousa, I am grateful for their availability and patience to answer all my questions.

To my parents, which were very supportive and without whom this job would not be possible.

Also, a huge thanks to all my friends that were there for me, were patient to hear all my complaints and, above all, to support me and cheer with me.

A special acknowledgement to my dear friends Sofia and Ângela, the ones I can count on to everything and even without understanding my work, considerably helped me.

Last but not least, the person with the most patience, always willing to discuss my problems and pushing me further, I could not thank you enough, Luís Neto.

This work was supported by FCT (Fundação para a Ciência e Tecnologia) within project RDB-TS: Uma base de dados de reações químicas baseadas em informação de estados de transição derivados de cálculos quânticos (Refª BI2-2019_NORTE-01-0145-FEDER-031689_UMINHO), co-funded by the North Portugal Regional Operational Programme, through the European Regional Development Fund.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

The Energy-Split tool receives as input pieces of a very large molecular system and computes all intra and inter-molecular energies, separately calculating the energies of each fragment and then the total energy of the molecule. It takes into account the connectivity information among atoms in a molecule to compute (i) the energy of all terms involving atoms covalently bonded, namely bonds, angles, dihedral angles, and improper angles, and (ii) Coulomb and the Van der Waals energies, that are independent of the atom's connections, which have to be computed for every atom in the system. The required operations to obtain the total energy of a large molecule are computationally intensive, which require an efficient high-performance computing approach to obtain results in an acceptable time slot.

The original Energy-Split Tcl code was thoroughly analyzed to be ported to a parallel and more efficient C++ version. New data structures were defined with data locality features, to take advantage of the advanced features present in current laptop or server systems. These include the vector extensions to the scalar processors, an efficient on-chip memory hierarchy, and the inherent parallelism in multicore devices. To improve the Energy-Split's sequential variant a parallel version was developed using auxiliary libraries. Both implementations were tested on different multicore devices and optimized to take the most advantage of the features in high performance computing.

Significant results by applying professional performance engineering approaches, namely (i) by identifying the data values that can be represented as Boolean variables (such as variables used in auxiliar data structures on the traversal algorithm that computes the Euclidean distance between atoms), leading to significant performance improvements due to the reduced memory bottleneck (over 10 times faster), and (ii) using an adequate compress format (CSR) to represent and operate on sparse matrices (namely matrices with Euclidean distances between atoms pairs, since all distances further the cut-off distance (user defined) are considered as zero, and these are the majority of values).

After the first code optimizations, the performance of the sequential version was improved by around 100 times when compared to the original version on a dual-socket server. The parallel version improved up to 24 times, depending on the molecules tested, on the same server. The overall picture shows that the Energy-Split code is highly scalable, obtaining better results with larger molecule files, even when the atom's arrangement influences the algorithm's performance.

Keywords: Energy-Split, intramolecular energy, development framework, HPC.

RESUMO

A ferramenta Energy-Split recebe como ficheiro de input a descrição de fragmentos de um sistema molecular de grandes dimensões, de maneira a calcular os valores de energia intramolecular. Separadamente, também efetua o cálculo da energia de cada fragmento e a energia total de uma molécula. Ao mesmo tempo, tem em conta a informação das ligações entre átomos de uma molécula para calcular (i) a energia que envolve todos os átomos ligados covalentemente, nomeadamente bonds, angles, dihedral angles and improper angles, e (ii) energias de Coulomb e Vand der Waals, que são independentes das conexões dos átomos e têm de ser calculadas para cada átomo do sistema. Para cada átomo, o Energy-Split calcula a energia de interação com todos os outros átomos do sistema, considerando a partição da molécula em fragmentos, feita num programa *open source*, Visual Molecular Dynamics.

As operações para o cálculo destas energias podem levar a tarefas muito intensivas, computacionalmente, fazendo com que seja necessário utilizar uma abordagem que tire proveito de computação de alto desempenho de modo a desenvolver código mais eficiente. O código fornecido, em Tcl, foi profundamente analisado e convertido para uma versão paralela e, mais eficiente, em C++.

Ao mesmo tempo, foram definidas novas estruturas de dados, que aproveitam a boa localidade dos mesmos para tirar vantagem das extensões vetoriais presentes em qualquer computador e, também, para explorar o paralelismo inerente a máquinas multicore. Assim, foi implementada uma versão paralela do código convertido numa fase anterior com recurso ao uso de bibliotecas auxiliares. Ambas as versões foram testadas em diferentes ambientes multicore e otimizadas de maneira a ser possível tirar o máximo partido da computação de alto desempenho para obter os melhores resultados.

Após a aplicação de técnicas de engenharia de performance como (i) a identificação de dados que poderiam ser representados em formatos mais leves como variáveis booleanas (por exemplo, variáveis usadas em estruturas de dados auxiliares ao cálculo da distância Euclideana entre átomos, utilizadas no algoritmo de travessia da molécula), o que levou a melhorias significativas na performance (cerca de 10 vezes) devido à redução de sobrecarga da memória. (ii) a utilização de um formato adequado para a representação de matirzes esparsas (nomeadamente a de representação das mesmas distâncias Euclidianas do primeiro ponto, uma vez que todas as distâncias que ultrapassem a distância de cutoff (definida pelo utilizador) são consideradas como 0, representado a maioria dos valores).

Depois das otimizações à versão sequencial, esta apresentou uma melhoria de cerca de 100 vezes em relação à versão original. A versão paralela foi melhorada até 24 vezes, dependendo das moléculas em questão. No geral, o código é escalável, uma vez que apresenta melhores resultados consoante o aumento do tamanho das moléculas testadas, apesar de se concluir que a disposição dos átomos também influencia a perfomance do algoritmo.

Palavras-chave: Energy-Split, energia intramolecular, framework de desenvolvimento, HPC.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

The Energy-Split tool is a Tcl implementation of an approach developed by scientists to aid to compute the whole energy components of a very large molecule. In this case the computation is a heavy iterative process that requires to consider all particles. The tool follows a divide-and-conquer algorithm by splitting the system into several fragments, and then it separately computes the energy for each fragment, before computing the total energy of the molecule. The splitting part is performed by an external software program, VMD (Visual Molecular Dynamics[1]).

To better understand where the Energy-Split fits in the overall project structure, Figure 1.1 portraits a diagram of the workflow from the fragment choice until all energies are computed.

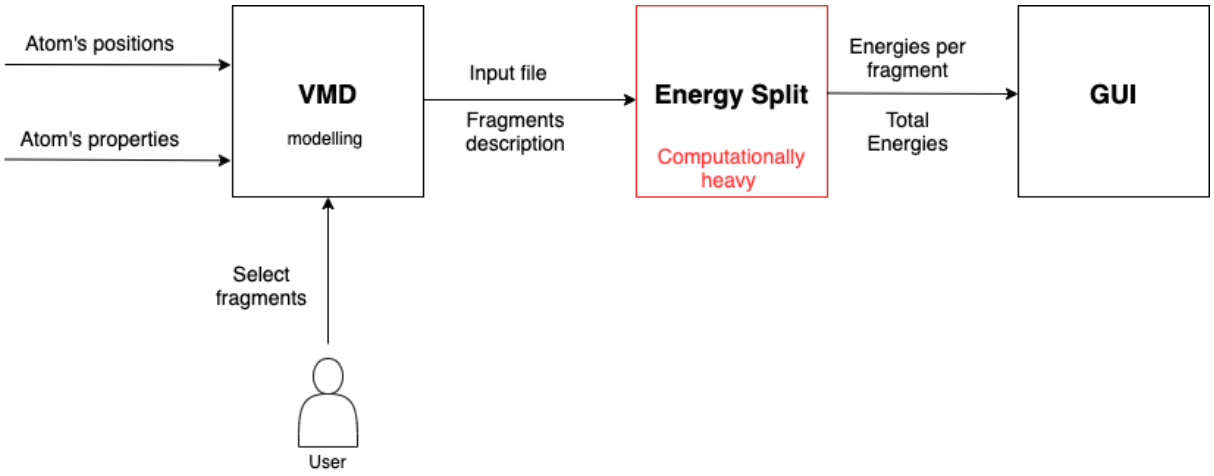


Figure 1.1: **Workflow to compute the energy of a fragmented molecule**

The Energy-Split computes the interaction energy among all atoms/molecules in a molecular split in user-defined fragments and then calculates the interaction energy for each fragment and between fragments. This computation takes into account the energy of the connection among atoms in a molecule (intramolecular). Since fragments can have multiple rearrangements, the approach is also able to compute the global energy difference between those different spatial rearrangements.

1.1 CHALLENGES AND GOALS

The Energy-Split purpose is to be applied in very large systems with several structures, becoming an useful tool to depict the source of energy fluctuations, with a panoply of applications. However, there is a performance issue with the original Tcl version of the Energy-Split: it takes more than 5 minutes to calculate the energy of a system containing about 100 000 atoms. Although the non-bonded terms are calculated using a parallel approach, the overall computation becomes very time-demanding when the focus is on comparing thousands of spatial rearrangements of a molecular system.

To complete the challenge of this work is worth to mention the required knowledge on molecular mechanics and on the several equations required in the energy calculations, also taking into account the connectivity information among atoms covalently bonded in a molecule (bonds, angles, dihedral angles, and improper angles). The connectivity specifies which atoms are linked (bonded) to each atom in the system. Additionally, non-bonded energy terms are computed for each pair of atoms in the system: the Coulomb energy and the Lennard-Jones energy. For each particle, the Energy-Split code calculates the interaction energy with all other atoms in the system.

After porting the code to a programming environment that offers better execution performance, it is necessary to characterize the application in terms of its computational tasks: if the code execution performance is compute-bound, memory-bound, or I/O bound. By profiling the code it is possible to identify where the key bottlenecks are and properly address and overcome these limitations. The execution times can then be improved and tested in different server architectures. The parallel version of the code can also be tested in a multithreaded environment.

To summarize, the main goals of this project were:

- to deploy an improved C++ parallel version from the Tcl Energy-Split code;
- to control in runtime the efficient use of the computing resources and to reduce execution times.

1.2 DISSERTATION OUTLINE

Chapter 2 presents the state of the art to compute intra-molecular energies and the molecular splitting into fragments, identifying the computing inefficiencies of the original software version. Chapter 3 details the Energy-Split tool and presents the approaches used to convert the Tcl version and to parallelize the code until reaching the final product. Chapter 4 presents the timeline of sequential and parallel results, the hardware used to achieve those results as

well as their discussion. The last chapter, Chapter 5, presents the conclusions of the research work and displays suggestions for future work.

2

COMPUTATION OF INTRA-MOLECULAR ENERGIES

The first step to the process of computing molecular energies is to prepare the input file with VMD (Visual Molecular Dynamics). This program also receives two files with information about a molecule, such as the atom's coordinates and the atom's properties. Then, it renders those files into a visual format of the molecule where the user can select the fragments whose energies will be computed by the Energy-Split code. As Figure 1.1 showed, information about fragments is passed to the Energy-Split through an input file generated by VMD. After all the preparation with VMD, it is time for the forces to start to be calculated by the Energy-Split. The code developed in Tcl takes into account several parameters to compute the values, like the distance between atoms, if they have bonded atoms or not, and even the angles between linked ones.

In molecular dynamics, a molecule is described as a series of charged points (atoms) linked by springs (bonds). To describe the time evolution of bond lengths, bond angles, also the non-bonding Van der Waals, and electrostatic interactions between atoms, one uses a force field. The force field is a collection of equations and associated constants designed to reproduce molecular geometry and selected properties of tested structures, these are designated by AMBER (Assisted Model Building with Energy Refinement) force fields[2]. AMBER is also the name for the molecular dynamics software package that simulates these force fields.

Six different molecular mechanics equations reproduce the behavior of the bounded and non-bounded atoms. When it comes to angles - dihedral (or torsion) angles and improper angles - there are aspects to take into account, like the planes where the atoms are placed and if there are three or more atoms on the chain. Figure 2.1 depicts the four examples of connections among particles.

Van der Waals involves some quantum chemistry calculations and uses the Lennard-Jones[3] potential to model the force, repulsive or not, among atoms. The Coulomb's potential (electrostatic force) is a well-known equation that reproduces the force when an object has its electric charge or its relative position to other electrically charged objects.

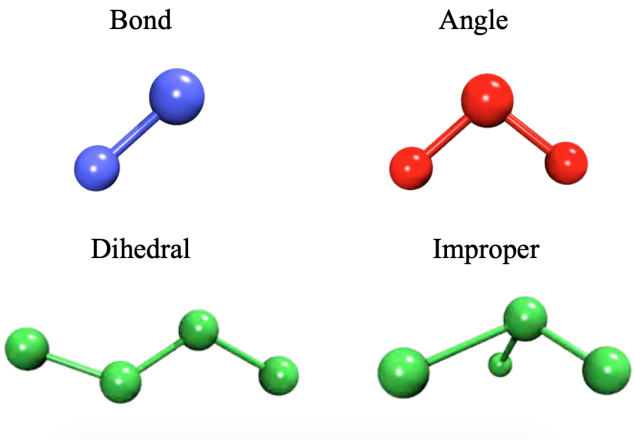


Figure 2.1: **States of the bonded atoms**

2.1 ENERGY SPLITTING APPROACH

The provided code is split into two major parts, the calculations for bounded terms and those for the non-bounded ones. The second part is the costly one, even though there are no dependencies between the counts. This happens because the estimations are made among all atoms of the molecule and, for now, it is where most CPU time is consumed.

2.1.1 *Energy computations*

As it was described before, for the energies computation to be possible, some molecular mechanics are used, named AMBER force fields. Those equations are calculated for bonded and non-bonded terms, since we know some atoms are linked to each other and some are not.

To compute the energies associated to covalently bonded atoms, the following expressions were used. Note that these computations involve parameters that come from AMBER and are described in the input file generated by VMD.

- Bonded terms

$$E_{bonds} = \sum k_b (l - l_0)^2$$

  k_b - Bond force constant ($kcal/mol/Å^2$)
  l_0 - Equilibrium bond length (Å)
  l - Bond length (Å)

- Angles

$$E_{angles} = \sum k_a (\theta - \theta_0)^2$$

k_a - Angle force constant ($kcal/mol/rad^2$)
θ - Angle amplitude (degrees)
θ_0 - Angle equilibrium (degrees)

- Dihedrals (Torsions)

$$E_{dihedrals} = \sum \sum_{i=1}^{4} \frac{k_i(1 + \cos(i.\theta - A_i))}{N}$$

k_i - Magnitudes for each atom i involved in the dihedral angle ($kcal/mol$)
A_i - Phase offsets (degrees)
N - Number of paths
θ - Dihedral angle (degrees)

- Impropers

$$E_{impropers} = \sum k_i(1 - \cos(P.\theta - A))$$

k_i - Magnitude ($kcal/mol$)
P – Period
A_i - Phase offsets (degrees)
θ - Improper angle (degrees)

For those terms which are non-bonded, the Energy-Split calculates for each atom the interaction's energy with all other atoms in the system, using two expressions: Van der Waals and Coulomb (or Electrostatic). Note that Coulomb's expression does not need any kind of parameters but atom's charges.[1]

- Van der Waals

$$E_{vdw} = \sum_{i<j} (\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}) \times ScaleFactor$$

$$A_{ij} = \varepsilon_{ij}.D_{ij}^{12}$$

$$B_{ij} = 2\varepsilon_{ij}.D_{ij}^{6}$$

$$\varepsilon_{ij} = \sqrt{\varepsilon_i.\varepsilon_j}$$

$$D_{ij} = R_i + R_j$$

ε_x - Energy well-depth ($kcal/mol$)
R_x– Atom Radius (Å)

1 There are no references to any energy expressions because they were provided by the REQUIMTE supervisor, Sérgio Sousa.

r_{ij} - Distance between atoms i and j (Å)
$ScaleFactor^1$ - applied to atoms separated by 1 to 3 bonds (usually 0 for 1-2 atoms; 0 for 1-3 atoms; and 0.5 for 1-4 atoms).

- Coulomb

$$E_{coulomb} = \sum_{izj} \frac{1}{4\pi\varepsilon_0} (\frac{q_i . q_j}{r_{ij}}) \times ScaleFactor$$

q_x - Atomic charge
ε_0– Vacuum permittivity
r_{ij} - Distance between atoms i and j (Å)
$ScaleFactor^1$ - applied to atoms separated by 1 to 3 bonds (usually 0 for 1-2 atoms; 0 for 1-3 atoms; and 1/1.2 for 1-4 atoms).

[1] - *The scale factor is a measure applied by the user, it can take any value*

2.1.2 *Fragments and frames*

All calculations are made considering the fragments that are chosen by the user. If the system is split into two fragments, the Energy-Split calculates the energy of fragments 1 and 2 and also the energy of interaction between both fragments like demonstrated on Figure 2.2. The interesting part is that we can have thousands of different spatial rearrangements of a certain molecular system, and the Energy-Split can depict which is the energy contribution of each part of the system for the global energy difference between those different spatial rearrangements.

One of the properties of having connections between atoms settles in the fact that they are in constant movement even though they remain connected. In other words, the position of an atom can vary over time and, consequently, the energy of that connection changes too. This is an important concept when we are before different fragments, since the computation of the total energy changes from frame to frame. A frame is a snapshot taken in a certain moment, it depicts the state of the molecule fragment at the time it was taken. Figure 2.3 shows the computations of the interactions when we have more than one frame.

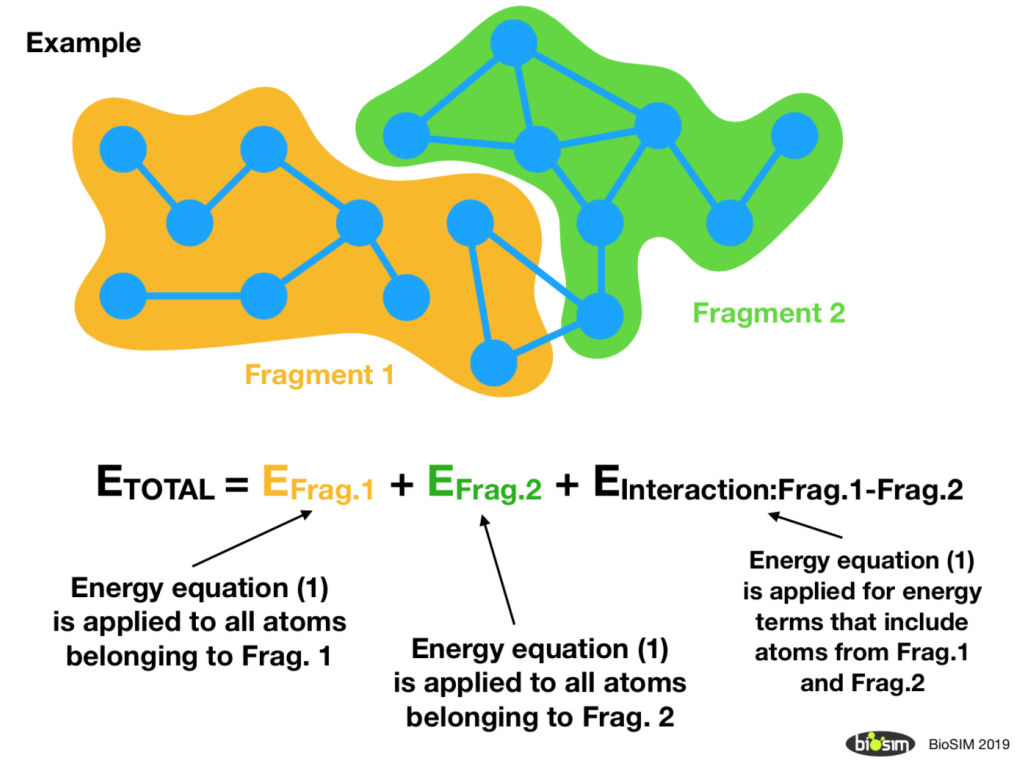


Figure 2.2: **Computation of energies by fragments**

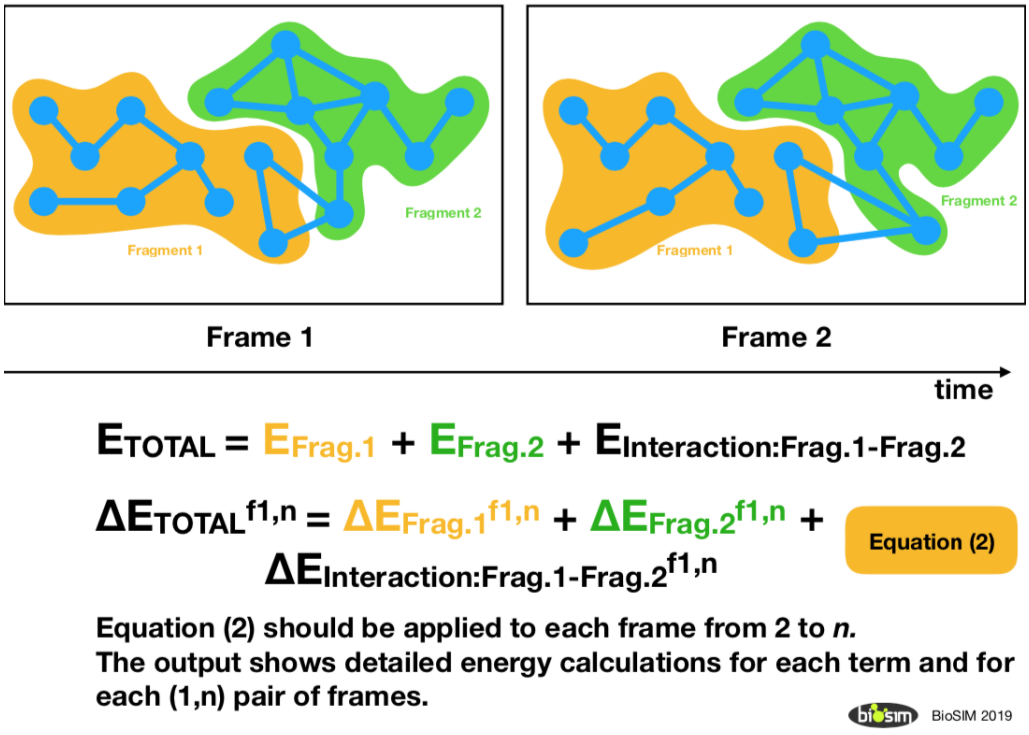


Figure 2.3: **Example of the energy computation with frames**

2.2 MOLECULAR MODELLING WITH VMD

VMD is developed as, essentially, a tool to view and analyze the results of molecular dynamics simulations. By providing an input file with molecule description and parameters, that is generated by a Tcl code developed by REQUIMTE with AMBER force fields. Those AMBER parameters include the atom's coordinates and properties like charges, types, and linked atoms. At the same time, the corresponding input file, which is in Tcl, is used by the Energy-Split extension to compute the energies calculations. Also, the user-defined fragments of the molecule are chosen on the VMD's interface and its description is included within the file. Figure 2.4 shows a simple input file into the Energy-Split code.

```
## Variables

set pi 3.14159265358979311599796346854
set numberAtoms 5
set types {N3 H H H CT}
set charges {0.07819998264312744 0.21999993920326233
             0.21999993920326233 0.21999993920326233 0.0291999913752079}
set xyz {
         {49.44599914550781 37.32699966430664 72.60800170898438}
         {50.266998291015625 36.740001678466872 58.100128173828}
         {48.64699935913086 36.78799819946289 72.9260025024414}
         {49.606998443603516 38.0890007019043 73.25800323486328}
         {49.16299819946289 37.87099838256836 71.26300048828125}
        }
set connectivity {{1 2} 3 {4 5}}
set parameters {
                {NonBon 3 1 0 0 0.0 0.0 0.5 0.0 0.0 -1.2}
                {VDW "C"   1.9080   0.0860}
                {HrmStr1 "CT" "HP" 340.00 1.0900}
                {HrmStr1 "N3" "HP" 340.00 1.0900}
                {HrmBnd1 "CT" "CT" "HC"  50.00 109.5000}
                {AmbTrs "C" "N" "CT" "H1"   0   0   0   0   0.000 0.000 0.000 0.000 1.0}
                {ImpTrs "C" "CT" "N" "H"  1.1 180.0 2.0}
               }
set fragList {{1 2 3 4 5} {6 7 8}}
```

Figure 2.4: **Example of a simple file with two fragments, generated by VMD**

After receiving this as an input file and making all computations, the energy values are printed to an output file, also in Tcl, as shown in Figure 2.5.

```
### Energy Split detected 8 cpu(s) available.

####################
### Fragments
####################
        Fragment 0: 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395
        Fragment 1: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76
87 88 89 90 91 92 93 94 95 96 97 98 99 100
####################
## Bonds
####################
        Energy Decomposition:
                Fragment 0 + Fragment X                         1.1175408337120441e-5   Hartree
                Fragment 0                                      0.001192631648569113    Hartree
                Fragment 1 + Fragment X                         3.1963997239640406e-5   Hartree
                Fragment 1                                      0.005877925197235168    Hartree
                Fragment X                                      1.3487114819893853      Hartree
        Total Energy: 1.3558251782407664 Hartree

####################
## Angles
####################
        Energy Decomposition:
                Fragment 1 + Fragment X                         0.003909277948827497    Hartree
                Fragment 0                                      0.005346621078139691    Hartree
                Fragment 0 + Fragment X                         0.0009422968362060424   Hartree
                Fragment 1                                      0.01587722526327147     Hartree
                Fragment X                                      2.840380163313124       Hartree
        Total Energy: 2.866455584439569 Hartree

####################
## Torsions
####################
        Energy Decomposition:
                Fragment 0 + Fragment X                         0.027413345672736608    Hartree
                Fragment 1 + Fragment X                         0.019303997109712725    Hartree
                Fragment 0                                      0.000960452740606765    Hartree
                Fragment 1                                      0.07027831060095976     Hartree
                Fragment X                                      11.779855376889085      Hartree
        Total Energy: 11.897811483013 Hartree
        Energy Decomposition:
                Fragment 0 + Fragment X                         7.384437533533025e-5    Hartree
                Fragment 1 + Fragment X                         1.843132775039692e-7    Hartree
                Fragment 0                                      0.0002228953732006277   Hartree
                Fragment 1                                      0.000930847728797576    Hartree
                Fragment X                                      0.14035946356321405     Hartree
        Total Energy: 0.14158723535382509 Hartree

####################
## VDW
####################
        Energy Decomposition:
                Fragment 0 + Fragment X                         -0.018663926510680447   Hartree
                Fragment 0                                      0.0005457981739828601   Hartree
                Fragment 1 + Fragment X                         -0.05217163842479787    Hartree
                Fragment 1 + Fragment 0                         -1.8613620853836156e-6  Hartree
                Fragment 1                                      0.0001458846864959111   Hartree
                Fragment X                                      -7.088191728714288      Hartree
        Total Energy: -7.1583374721513735 Hartree

####################
## Coloumb
####################
        Energy Decomposition:
                Fragment 0 + Fragment X                         0.08737279750499699     Hartree
                Fragment 0                                      0.05086567722001398     Hartree
                Fragment 1 + Fragment X                         -0.37673283820791265    Hartree
                Fragment 1 + Fragment 0                         0.0015886339333808883   Hartree
                Fragment 1                                      -0.14002814587392498    Hartree
                Fragment X                                      -44.06932366374086      Hartree
        Total Energy: -44.44625753916431 Hartree

####################
## Total Energy
####################
        Energy Decomposition:
                Fragment 0 + Fragment X                         0.09714953328693164     Hartree
                Fragment 1 + Fragment X                         -0.40565905326365315    Hartree
                Fragment 0                                      0.059134076234513035    Hartree
                Fragment 0 + Fragment 1                         0.0015867725712955048   Hartree
                Fragment 1                                      -0.04691795239716509    Hartree
                Fragment X                                      -35.04820890670034      Hartree
        Total Energy: -35.342915530268414 Hartree
```

Figure 2.5: **Example of the Energy-Split output file with the different energies**

2.3 COMPUTATIONAL EFFICIENCY

An initial analysis of the provided code is enough to reveal the origin of some performance deficits. The original code version was developed in Tcl, which is a high-level, general-purpose, interpreted, dynamic programming language. It was designed to be very simple but powerful, yet it is a C embedded language for scripted applications, GUI's and testing. The fact that it needs a runtime environment on which the code can be compiled upfront and make redundant calls to the interpreter makes the program slower to load and run. It is used by VMD for users to run their scripts because it includes a Tcl interpreter.

The first step to improve its performance is the translation of the original code, a Tcl extension on VMD, into a compiled language. The language chosen is C++, a system programming language, which has a good performance by offering ahead of time compilation, memory management, and an elegant way to support parallel programming and multithreading.

Another inefficiency comes when one computes the results for bounded terms, there is an iteration over each atom linked atoms, which are identified by its index on the list of all particles. In other words, accessing an atom's information depends on the indexes of their connections. Currently, it supports *stride-n* reference pattern of instructions, since an atom with an index of 1 can be connected to another with an index of 12, as portrayed in Figure 2.6. This problem is recurrent on all equations that involve bounded terms, especially with dihedral and improper angles that force to have four inter-connected atoms. These kinds of jumps from index to index increase the execution time due to the inadequate data locality in memory, as Figure 2.6 shows.

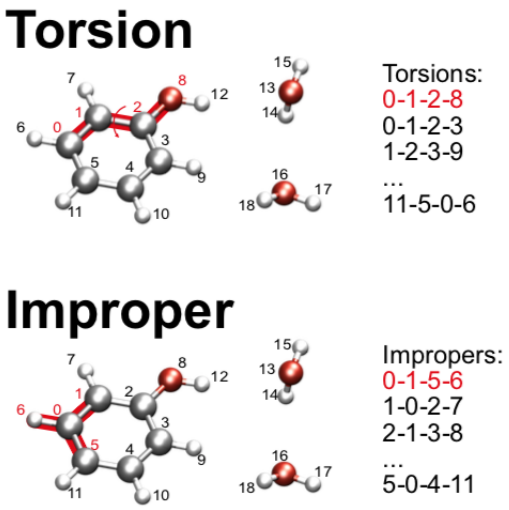


Figure 2.6: **Example of the stride-n accesses made in the original version**

As previously stated this issue is solved by redefining the data structures to take advantage of the properties of data locality and eventually vectorization. The new arrangement is an array of matrices that have relevant information about the atoms and their connections. Therefore, all the matrices have the same number of columns and vary in the number of rows. Columns represent atom properties like id, type, or coordinates, and rows represent atom neighbors that differ from atom to atom. Figure 2.7 portrays the structure described.

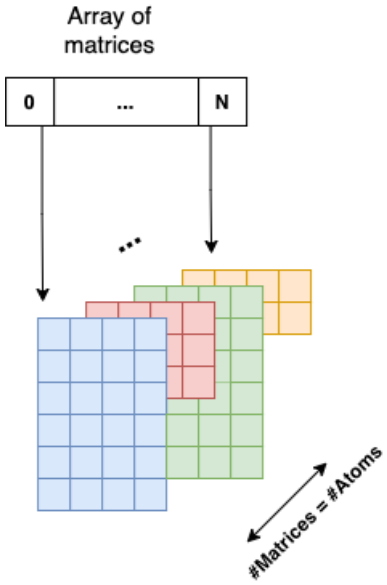


Figure 2.7: **Data structure used in the C++ version**

For the non-bounded terms, the problem stands on the computation of a variable, the scale factor referenced before as a measure applied to atoms separated by 1 to 3 bonds. This variable often takes 0 as its value, making much of the computations unnecessary and adding penalties on the execution time.

Beyond these issues we still have the frame challenge: multiple frames involve multiple repetitions of the energies among atoms and fragments computations. In a real case, this happens when an atom changes its location (coordinates) from frame to frame, which implies the recalculation of all terms. Even if there is a reuse of the energy values, it is still necessary to iterate over all connections to check which pair of atoms changed their positions and compute the energy of that connection again.

As molecules can have large numbers of particles, the computation will also depend on the memory latency to get the data. As was previously described, some considerations were taken in what concerns data locality. In other words, the data structures were carefully chosen to guarantee adequate access to memory and to prevent the penalties of the latency inherent to fetch the information from memory.

Another aspect to take into account is the possible dependencies between computations. As the code is divided into two major parts, they have to be analyzed separately to search for these dependencies.

To compute the energy due to the bounded terms, data of at least two atoms needs to be fetched to make the calculations.

As the system is not composed of connections of only two particles, there is more data necessary to compute each kind of energy. The equations provided involve three different types of angles that can be formed by the same set of atoms. It would be relevant to reuse some calculus over these computations. This recycling can come with the cost of creating dependencies between data and destabilize the parallelization approach. However, it may reduce the number of executed instructions and prevent repeated computations.

It is also necessary to characterize the application in terms of its computational tasks. To categorize the application as *I/O bound, memory-bound, or compute-bound*, the program needs to be profiled to identify its bottlenecks. If there are little computations and performance is limited by the I/O latency, this may lead to an I/O bound application. Compute-bound has large amounts of calculations to perform for each byte of memory accesses and the hardware specifications may limit performance. If the program has irregular accesses or *stride-n* reference patterns to memory with low computation tasks, this may lead to a memory-bound application.

Code parallelization for distributed memory environments may be considered since the dataset has no data dependencies among its elements. This version can be tested on a homogeneous server with an adequate workload distribution, namely with the aid of a software package that was developed for the high-energy physics researchers, the HEP-Frame [4] (detailed ahead), which takes the maximum advantage of the available computing resources in runtime, to reach optimum performance

After developing an approach that efficiently runs on multicore devices, the code can be tuned to execute on heterogeneous servers with accelerators, such as a GPU device.

The development of a graphical user interface (GUI) is also required to be integrated into the tool, so that users can visualize the results of the Energy-Split module. The GUI must be carefully designed to support its integration with the VMD package.

2.3.1 *Multicore devices*

The testing environment to use in this project is a multicore dual-socket homogeneous server, the most common platform in a cluster system. Each processing device has its own memory hierarchy [5] with three cache levels — two exclusive to each core (L1 and L2) and a shared L3 cache — a multi-channel access to random access memory (RAM) chips and a high bandwidth interface that connects both processing devices.

This memory hierarchy, where each processing device is directly connected to RAM modules, creates different memory access times for each core in a processing device: faster to access the RAM data on the chips connected to the processing device where the core is, and slower access times to the RAM on the neighbor processing device. This non-unified memory access is known as a NUMA architecture.

Since the small and faster cache memories are closer to the processor and using a different electronic structure (static *vs.* dynamic), a hit to a higher level of the hierarchy makes the data access faster. If the access misses, it goes to a lower level of the hierarchy, which is slower and larger. RAM accesses may also be penalized in a NUMA architecture, with different memory access latencies depending on which device the RAM modules are connected: data has to pass through their interconnection, increasing the application execution times.

Each processor device can run distinct instruction flows on separate cores, as Figure 2.8 depicts, increasing the overall speed for programs that support multithreading or other parallel computing techniques.

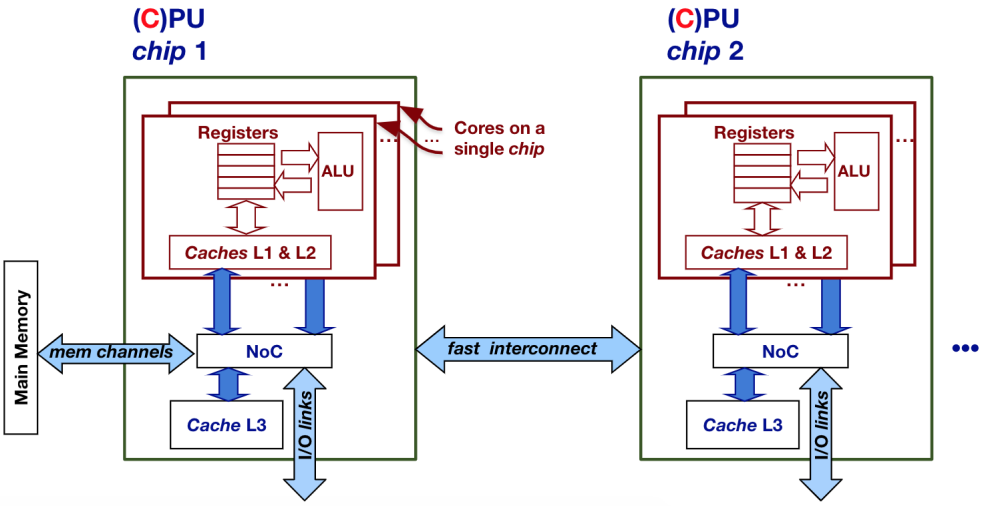


Figure 2.8: **Multicore device in a homogeneous server**

Due to the continuous hardware improvements over time, it is possible, for developers to spend less effort tuning code for better efficiency, as their programs run faster with these hardware improvements. One of the changes was the addition of more cores to a single chip, creating multicore devices, as portray in Figure 2.8. With this, the number of executed micro-instructions per clock cycle increases too; however one has to be more sensitive to write parallel code that exploits this optimization.

Allying a greater number of processing units (cores) with a hierarchical memory with multi-level caches to reduce the latency of data fetching from the main memory, each core also implements several mechanisms to improve performance, such as (i) ILP (Instruction Level Parallelism), the overlapping of instructions execution that would otherwise execute

sequentially, (ii) SMT (Simultaneous Multithreading), the hardware support to simultaneous execution of multiple threads in a single core, and (iii) instruction set extensions for vector computing, which will be detailed later.

Depending on the system's architecture and programming paradigm provided, two different versions of code can be written to take full advantage of the hardware: (i) a shared memory approach, which offers a similar throughput to all device cores involved both on a UMA architecture (Unified Memory Access) or on a NUMA architecture and (ii) a distributed memory approach, where independent processes communicate with others via message passing software modules.

2.3.2 *Multithreading and multiprocessing on multicore devices*

OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming on many platforms, instruction set architectures and operating systems. The core elements of OpenMP are the constructs for thread creation, workload distribution (work-sharing), data-environment management, thread synchronization, user-level runtime routines, and environment variables.

Its scheduler is based on a work-sharing strategy, where a master thread produces a set of other threads to simultaneously compute a task or different tasks on a shared data environment. This approach is efficient for irregular workloads.

TBB - Thread Building Blocks

Intel thread building blocks (TBB) is a widely used C++ library for shared-memory parallel programming and heterogeneous computing (intra-node distributed memory programming). If the code is written in C++, Intel TBB is likely the best fit. Intel TBB matches especially well with the code that is highly object-oriented and makes heavy use of C++ templates and user-defined types, even though it introduces a little coding overhead.

Unlike OpenMP, TBB does not require specific compiler support. It is a C++ template library to develop software that takes advantage of multi-core processors. The library contains various data structures and implemented algorithms that allow avoiding some of the complications of using lower-level APIs.

Intel TBB abstracts access to the multiple processors by allowing operations to be treated as tasks, which might be dynamically allocated to individual cores. An important factor here is the efficient use of the available cache hierarchy. A TBB program creates, synchronizes and destroys graphs of dependent tasks according to some implemented algorithms.

TBB has been designed to naturally support nested and recursive parallelism. A fixed number of threads are managed by the TBB task scheduler's task stealing technique. With this and the task scheduler's, dynamic load balancing algorithm, TBB makes it possible to keep all processor cores busy with useful work, without over-subscription (too many software threads means unnecessary overhead) and with minimal under-subscription (too few software threads means you are not taking full advantage of the available multiple cores).

Pthreads and Boost threads

Pthreads are defined in the C language and implemented with a header (pthread.h) and a library that, together, can potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead and require fewer system resources than managing processes. It includes four subroutines that are not included in the previous paradigms, like thread management, mutexes, condition variables, and synchronization.

The main advantage of using Pthreads, as compared to the previously presented solutions, is that all other solutions are, usually, internally build upon Pthreads. Since there are no advanced threading algorithms included, one has to split the work of for-loop manually.

On the other hand, the design of Boost.Threads were strongly influenced by Pthreads but styled as a C++ library instead of a C library. Boost.Thread enables the use of multiple threads of execution with shared data in portable C++ code. It provides classes and functions for managing the threads themselves, along with others for synchronizing data between the threads or providing separate copies of data specific to individual threads, as well as mechanisms like mutexes and conditional variables.

MPI and OpenMPI

MPI (Message Passing Interface) provides an API for processed based parallel programming in distributed memory environments, typical in computing clusters. Also, it is based on point to point messaging communication, where processes communicate with each other to share information, since it is not a shared memory environment. However, it can be used to handle work-sharing among servers and, in this context (OpenMPI), to ensure an efficient parallelization inside servers with physical shared memory. The programmer is responsible to guarantee scheduling strategies, create and manage an adequate amount of processes to be used in each server.

2.3.3 *Vector computing and data locality*

Vectorization is the process of transforming a scalar operation acting on a set of scalar data elements (SISD - Single Instruction stream, Single Data stream) into a vector operation, where a single instruction concurrently operates on multiple data elements (SIMD). Current multicore devices already support instruction set extensions to operate on vectors and their use is imperative to achieve full computational performance. Therefore, the programmer has to have basic care in what concerns developing code which makes vectorization possible for the compiler, namely resolving data dependencies, having unit-stride accesses and correct data alignment.

To take advantage of the compiler optimizations, such as vectorization, the data structures were carefully chosen to make sure there are no dependencies between sets of consecutive scalar data. This is an indispensable measure to ensure the compiler will generate vectorized code and it is particularly critical when defining structures with large quantities of data before all computations can be done. Since the data structures in this work were essentially matrices and vectors, we can take advantage of data locality, namely spatial and temporal locality: the first one takes advantage of the spatial proximity of the data, while the second one explores the reuse of recently accessed items.

A good data locality is essential as far as it allows information to be fetched in the same order as it is stored in memory. A program that enjoys one of temporal or spatial locality will be faster than one with poor locality. Thus, having data structures like matrices or vectors, which are stored contiguously and, consequently, makes the memory accesses contiguous too, provides a stride-1 reference pattern of read instructions, that gives the program a good spatial locality even if there is minor temporal locality, since the elements are accessed only once.

2.3.4 *Frameworks for efficient workload scheduling*

Legion

The design of Legion[6] focuses on a data-centric programming model that allows the user a better writing of parallel applications on heterogeneous servers. It permits an abstraction of data handling and makes explicit the definition of certain properties that, normally, are implicit to the programmer, like coherence and partitioning. For Legion to provide high-performance parallelization, it is necessary a configuration of a specific data structure. This structure must be adopted by the applications that experience this framework so it can correctly balance and distribute data among devices. With the use of logical regions, it is possible, for the developer, to implement dependencies among several tasks and, according to a history of the execution time of the jobs, it dynamically partitions and distributes the

data among devices, with an algorithm provided by the user. Though, unlike pipelined applications, it does not allow different tasks to execute simultaneously on the same data structure.

StarPU

StarPU[7] is a task programming library for hybrid architectures that allows the programmers to concentrate on algorithmic concerns instead of low-level issues through an abstraction of the system's architecture. It frees the developer of the workload scheduling and data consistency from the distributed memory environment of heterogeneous servers. Rather than rewriting the entire code, programmers can encapsulate existing functions within small procedures and it is possible to specify one function for each architecture (e.g. one function for CPUs and one function for CUDA). StarPU takes care of the task scheduling by automatically determining how much work each data will compute and in which device it will execute. This framework attempts to reduce and eliminate unnecessary memory transfers from multicore devices to accelerators in order to improve the performance of an application. Part of the scheduling strategy comes from using a queue sorted by the task's priority. However, it is the programmer's job to define each task's priority during the execution of the program. When it comes to memory consistency, the developer has almost no interaction since the framework ensures the asynchronous data transfer. However, the structure of the data has to be defined by the developer as well as the granularity of the task distribution. The major problem with StarPU is the workability to port the code, since it gives little flexibility when it comes to defining the task's organization and behavior.

HEP-Frame

This Highly-Efficient Pipeline Framework[4] was designed to aid the development of applications that use the available computational resources in homogeneous and heterogeneous servers, while guaranteeing a portable efficiency across different hardware generations. This happens because it was developed to present an interface that automatically generates code and automates the compilation process. Its multi-layer scheduler adapts at run-time the application to the compute server and processes the pipeline statements and various dataset elements in parallel, distributing them across the available computing resources. These features are what distinguish this framework from the previous ones since, unlike Legion, it allows simultaneous execution of tasks in different data structures and, unlike StarPU, it provides an easy port of the code and easy definition and distribution of the task's organization and behavior.

2.3.5 *Tools for profiling*

Code optimization is the way to achieve better performance of a developer's code. It is also one of the main goals of this dissertation, and it can not be accomplished without appealing to some tools. The following mechanisms collect data as memory used, the complexity of the program, or even a specific set of instructions, to diagnose many kinds of bottlenecks and understand the program's behavior, aiding the developer to adjust it to reach his objective.

Dtrace

Dtrace[8], short for Dynamic Tracing, is a framework developed for Unix systems. Currently, multiple operating systems like Solaris and Mac OS include it by default. By providing a language, D, it is possible to write scripts to dynamically trace multiple in-kernel overviews. With no support for Linux systems, Dtrace was used to profile the code on a personal laptop instead of the server to trace the function's execution time and build the program's call graph.

Gprof

Gprof[9], short for GNU GCC Profiling Tool, is a similar framework to Dtrace that runs on Linux environments. It also collects data and turns it into profiling statistics of each function's execution time and calls. It is possible to build the program's call graph with the data collected and analyze which methods are more time or memory consuming. This was the tool used on the server to trace the program's behavior and its call graph.

Perf

Perf[10], short for Performance counter for Linux, is also a tool for dynamic trace. Its approach is different from Dtrace since it works with system events and probes, making it more complete than Gprof since it allows statistics from the kernel and the userland. Perf monitoring gives knowledge about the program's performance and is useful to identify bottlenecks. In combination with Dtrace, it was used to profile the Energy-Split code on the cluster server.

Intel Advisor and VTune Profiler

Intel Advisor[11] and VTune[12] are two frameworks designed to improve software performance. Both provide a user-friendly interface that simplifies the analysis done to the code. That benchmarking includes analysis of efficient threading, vectorization, and memory usage. These two tools were mainly used to tune and test the efficiency of the code parallelization.

3

ENERGY-SPLIT

The first phase of this dissertation was the conversion of the Energy-Split Tcl code to C++. It started with a detailed analysis of the code and included the implementation of new and more efficient data structures for the C++ version.

To correctly convert the Tcl code to a C++ code, it was necessary an extensive study of the involved equations. As described in the previous chapter, six equations were applied to compute the energies. Four for bonded terms, bonds for two covalently bonded atoms, angles for three covalently bonded atoms that make an angle between them, dihedrals for four atoms which connection makes an angle. For last, there are the impropers, which are four bonded atoms with a different angle from dihedrals. Regarding the non-bonded terms, there are two more equations, the Van der Waals and the electrostatic. Following there is a detailed description of how each expression was implemented in Tcl, and then, follows the explanation of the code porting to C++.

The Energy-Split code computes in different procedures each of the relevant energies: bonds, angles, dihedral angles, improper angles, Van der Waals, and Coulomb. The outcome of each procedure is placed in a global variable, whose name is "<energy_type>TotalList". The supplied equations and associated parameters that came from VMD are described below. These parameters start with an identification of the type of connection between atoms. As only the bonded terms require these parameters, there are four different nomenclatures defined from the Amber force fields (described in the introduction of Chapter 2):

- **HrmStr1** - Harmonic String (bonds)
- **HrmBnd1** - Harmonic Bend (angles)
- **AmbTrs** - Amber Torsion (dihedrals)
- **ImpTrs** - Improper Torsion (impropers)

These expressions are computed in different procedures in the code, except angles and dihedrals that are in the same procedure, as well as Van der Waals, and Coulomb.

Bond

$$E_{bonds} = \sum k_b (l - l_0)^2$$

k_b - Bond force constant ($kcal/mol/Å^2$)
l_0 - Bond length equilibrium (Å)
l - Bond length (Å)

Parameters required by Energy-Split code:
Atom-type1 | Atom-type2 | k_b | l_0

Example of suplied parameters by VMD:
HrmStr1 "C" "CT" 340.00 119.34

Global variable where the result is stored:
bondTotalList

Procedure description:
The HrmStr1 is the description for harmonic string, which represents the bonded terms. In the image are examples of values for the parameters. This method computes the energy equation for bounded terms. For each atom, builds a list of its connections and iterates over the elements on that list to get the distance between atoms. After calculating the energy, it stores those values on a table, sums on the right fragment and prints it on the output file.

Angles and dihedrals

$$E_{angles} = \sum k_a (\theta - \theta_0)^2$$

k_a - Angle force constant ($kcal/mol/rad^2$)
θ - Angle amplitude (degrees)
θ_0 - Angle equilibrium (degrees)

Parameters required by Energy-Split code:
Atom-type1 | Atom-type2 | Atom-type3 | k_a | θ_0

Example of suplied parameters by VMD:
HrmBnd1 "C" "N" "H" 50.00 120.0001

Global variable where the result is stored:
angleTotalList

$$E_{dihedrals} = \sum \sum_{i=1}^{4} \frac{k_i(1 + \cos(i.\theta - A_i))}{N}$$

k_i - Magnitudes for each atom i involved in the dihedral angle (kcal/mol)
A_i - Phase offsets (degrees)
N - Number of paths
θ - Dihedral angle (degrees)

Parameters required by Energy-Split code:

Atom-type1 | Atom-type2 | Atom-type3 | Atom-type4 | A_1 | A_2 | A_3 | A_4 | k_1 | k_2 | k_3 | k_4 | N

Example of supplied parameters by VMD:

AmbTrs "O" "C" "N" "H" 0 180 0 0 2.000 2.500 0.000 0.000 1.0

Global variable where the result is stored:

dihedralTotalList

Procedure description:

The HrmBnd1 is the description for harmonic bend, which represents the angles between atoms. The AmbTrs is the description for Amber torsions and depicts the dihedral (torsion) between atoms. In the image are examples of values for the parameters. This method involves the computation of both angle and dihedral energy equations. First, it calculates energies for a group of 3 atoms with an angle on their disposition and associates each energy calculated with the right fragment. Secondly, it does the same thing with dihedrals, but with 4 atoms and their parameters.

Impropers

$$E_{impropers} = \sum k_i(1 - \cos(P.\theta - A))$$

k_i - Magnitude (kcal/mol)
P - Period
A_i - Phase offsets (degrees)
θ - Improper angle (degrees)

Parameters required by Energy-Split code:

Atom-type1 | Atom-type2 | k_b | l_0

Example of suplied parameters by VMD:

ImpTrs "CA" "CA" "CA" "HA" 1.1 180.0 2.0

Global variable where the result is stored:

improperTotalList

Procedure description:

The ImpTrs is the description for improper torsions, which represents the improper angles. In the image are examples of values for the parameters. This method checks the connections among atoms. If one atom is connected with 3 more atoms, the energy is calculated and associated with the correct fragment and printed on the output file.

Van der Waals and Coulomb

$$E_{vdw} = \sum_{i<j}\left(\frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}\right) \times ScaleFactor$$

ε_x - Energy well-depth (kcal/mol)

R_x– Atom radius (Å)

r_{ij} - Distance between atoms i and j (Å)

ScaleFactor - applied to atoms separated by 1 to 3 bonds (usually 0 for 1-2 atoms; 0 for 1-3 atoms; and 0.5 for 1-4 atoms).

Parameters required by Energy-Split code:

Atom-type1 | ε_x | R_x

Example of suplied parameters by VMD:

VDW "C" 1.9080 0.0860

Global variable where the result is stored:

vdwTotalList

$$E_{coulomb} = \sum_{i<j}\frac{1}{4\pi\varepsilon_0}\left(\frac{q_i.q_j}{r_{ij}}\right) \times ScaleFactor$$

q_x - Atomic charge

ε_0– Vacuum permittivity

r_{ij} - Distance between atoms i and j (Å)

$ScaleFactor^1$ - applied to atoms separated by 1 to 3 bonds (usually 0 for 1-2 atoms; 0 for 1-3 atoms; and 1/1.2 for 1-4 atoms).

Parameters required by Energy-Split code:

None

Global variable where the result is stored:

coulombTotalList

Procedure description:

This method uses threads to take advantage of parallelization, since the computations are independent of each other. However, this became the most computationally intensive procedure because it computes the interaction energy between one atom and all remaining atoms, for each particle of the molecule. This process happens twice, one for the Van der Waals expression and one for the Coulombs. The procedure is similar to the ones above, it stores the energies calculated on both vdwTotalList and coulombTotalList.

3.1 PORTING THE TCL TOOL INTO C++

The previous chapter described several inefficiencies of the Tcl code that need to be fixed, and this section gives special attention to the irregular memory accesses to fetch the atoms information. Currently, the accesses depend on each atoms connections list, making data referencing a stride-n access pattern. This kind of pattern causes a problem because the data is not accessed contiguously, and memory latency becomes a penalty in the execution time. To overcome this in the C++ version, a data structure array-of-matrices was defined. The array represents the total list of atoms and each position is a pointer to a matrix with all connections information. The number of lines is equal to the number of connected atoms, making the size of each matrix variable, as it can be seen in the Figure 3.1. The structure allows row-major accesses that bring blocks of information with the same size as the memory line and are essential to take advantage of vectorization instructions.

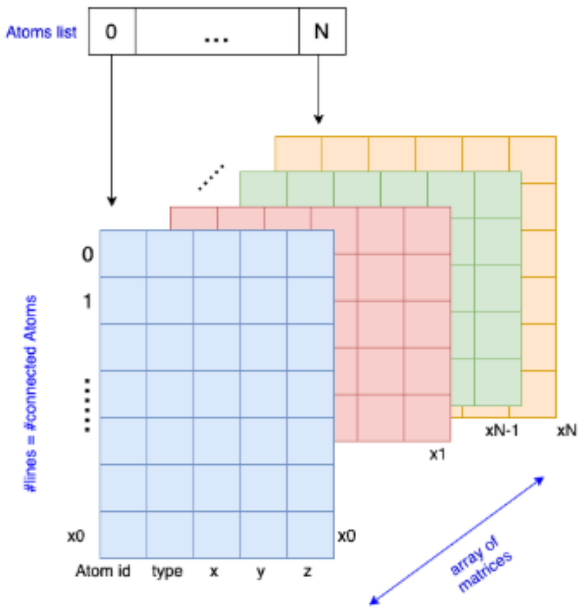


Figure 3.1: **Data structure for the atoms information**

As previously seen in Chapter 2, a typical VMD input file contains information on the atoms of a molecule. This data includes seven fields of interest: the total amount of atoms and their types, charges, coordinates, connections, and parameters. So, the port of the Tcl input file had several steps:

1. **Filter the molecule information from the output file generated by VMD**.

   The parsing is made line by line and requires the development of different functions that, for example, split a string by spaces or erase all occurrences of a specific character in a string to reach the relevant information, which is usually an incomplete string. After filtering, the data is distributed in the respective auxiliary data structures, at first vectors or vectors of vectors, depending on the data.

2. **Generate and initialize the data structure**.

   After parsing, all data is gathered in a more complex arrangement, as described before, the array of matrices. This structure needs to be initialized before being filled with data, to avoid memory issues, since the matrices may have different sizes. The size of each matrix is related to the number of connections of each atom.

   The following code snippet presents the matrix initialization.

```
float*** prepareMatrix(int nAtoms, vector<vector<float>> connections){
    float ***matrix = new float **[nAtoms];

    for (int i = 0; i < nAtoms; i++)
    {
        int sz = connections[i].size();
        matrix[i] = new float *[sz];
        for (int j = 0; j < sz; j++)
        {
            matrix[i][j] = new float[5]; // atomID, type, x,y,z
            for (int k = 0; k < 5; k++)
            {
                matrix[i][j][k] = 0;
            }
        }
    }
```

3. **Fill the structure with the information**

   After initialization, the data structure is ready to be filled with all information on the molecule atoms. The filling process is row-wise, with rows representing the fields of data and the columns represent the index of the connected particle. The computation of most equations requires this structure to keep contiguous accesses to the memory. However, the arrangement described has stride-1 accesses to fill it but not when it

comes to reading it, so when all data is in the right place, it is applied a transposition to each matrix.

4. **Building methods for each energy expression**

   The code porting to C++ relies on the Tcl code. Methods are very similar with few optimizations on the second version of the code because the process of computing the equations does not change between programming languages. After analyzing all Tcl equations, there is one method that seems to stand out, the one who involves Van der Waals and Coulomb's equations. Unlike the previous ones, all atoms have to be visited more than once to compute the energy involved.

3.1.1 *C++ code architecture*

The described four steps to port the Tcl code suggested to structure the code in different C++ files to keep code modularization. Figure 3.2 shows two files for each conversion step.

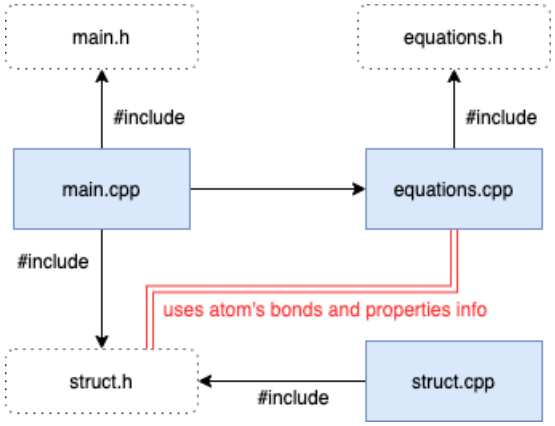


Figure 3.2: **File structure**

The struct files refer to the definition of all functions needed to deal with the main data structure, like generation and filling the array of matrices. The equation files are the ones who include all methods necessary to compute the expressions. For this to be possible, is needed some relation between the equations file and the struct file because all functions need to have access to the main data structure. Finally, the "main" files are the bridge between the other two, which includes functions responsible for parsing the input file, redirecting data to be filled in the structure, receiving and presenting the outputs from the equations in a clean output file.

3.2 THE SEQUENTIAL C++ TOOL

The main goal of this thesis is to optimize the sequential version and to implement an efficient vectorized and parallel code. The code is split into four main functions: bondEnergy, angleTorsionEnergy, impropersEnergy, and vdwCoulombEnergy. Some equations are implemented in the same method to reduce time and calculations, since those can be computed in just one traversal instead of two.

The first step was the passage to the graph concept instead of the molecule, where vertices represent atoms, and edges represent the connections between them. After creating appropriate structures to guarantee contiguous access to memory, now, the problem in the first three equations is to get an efficient way to traverse the graphs to compute the energies without calculating the same pairs of atoms twice. The problem with the vdwCoulombEnergy method is the thousands of calculations that emerge from the need to compare one atom with all others in the system.

3.2.1 *Bond energy*

Since this computation only involves two atoms, the graph traversal is simple: it only requires checking how many connections an atom has and compute the energy for each one. As the graph is undirected, there is a chance to repeat the energy computation of a pair twice, since the values of the connection 0-1 are the same as 1-0. To avoid repeating pairs of atoms, there is an auxiliary Set container to maintain a specific order of the elements, avoiding repetitions. After knowing which two atoms indexes to calculate, the algorithm gets their types to be able to get the Amber parameters necessary for the bond formula and the coordinates, and proceeds to compute the bond energy.

3.2.2 *Angle and torsion energies*

The reason these equations belong to the same method is that both involve at least three atoms. Torsions involve the same particles as angles plus an extra one, so the process is the same for both equations except for dihedrals that has one more step.

Consider Figure 3.3, one example of an angle chain would be 0-7-8, and one for torsions would be 0-7-8-9.

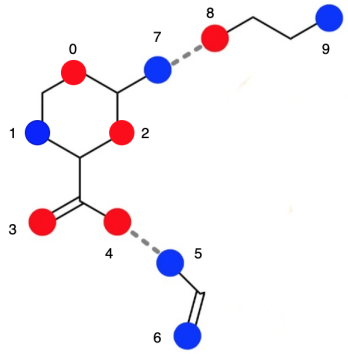


Figure 3.3: **Example of a molecule to show connections between atoms**

When it comes to angles and torsions, the energies should not be computed twice, since there are more atoms involved in both formulas, and there is no guarantee that the graph is acyclical. To keep track of the combinations already calculated is used a Set container to maintain a specific order of the elements inserted and to prevent computing 0-9-7-8 chain energy after computing 0-7-8-9.

After building a chain, if it was not computed, the algorithm proceeds to know the types and coordinates of the atoms involved. Two auxiliary functions use these parameters to estimate the cross product of two vectors and the angle between two vectors. The atom's types are necessary to find the Amber parameters, like constants used in the equations. After finding all variables, it can proceed to compute the energy.

3.2.3 *Impropers energy*

The impropers equation is simple to compute, since it is only necessary to verify for each atom if it has three connections. If it passes the condition, the algorithm takes the types and coordinates of the particles to compute cross product and the angle between vectors, as in subsection 3.2.2, and Amber parameters necessary to forecast the Impropers equation.

3.2.4 *Van der Waals and Coulomb energies*

Since the beginning it was known that those two equations would be the most computationally intensive: they have to compare the distances of each atom with all others in the molecule. The number of comparisons depends on the number of atoms in the molecule and can be inferred from the triangular number sequence: $\#pairs\_of\_atoms = n(n+1)/2$. As the number of atoms increases, the number of comparisons also increases, quadratically. However, this is not the only obstacle to compute these equations because there are two distances to take into account: the euclidean distance and how many connections apart there is between the atoms in consideration. The energy computation only happens if the

particles are apart by three or more bonds because the ones with less than three bonds were calculated in the equations previously described. At the same time, there is a measure designated Scale Factor that takes different values when applied to atoms three bonds apart and more than three bonds.

An efficient graph traversal is required to balance the waste when two atoms are less than three bonds apart. The approach used was a modification of the Breadth-First Search, where each node begins as the root, and all neighbor nodes are explored before moving to the next level. When the algorithm faces a disconnected graph, it does not assume all vertices are reachable from the starting node and takes extra steps to reach all detached atoms.

When a vertex is three levels distant from the root, the energy between them is calculated, as presented in Figure 3.4 where 0 is the root, and the red dots are the ones three levels distant. If there is no path between the origin and a node, the bond distance between them is more than three bonds. If there is a cycle, two paths may exist between the starting node and another node, with three alternatives:

- in both paths: if the second node is less than three levels distant, both are disregarded;
- in one path the second node is three levels apart: the algorithm chooses this one;
- in both paths: if the node is more than three levels apart, the shortest one is adopted.

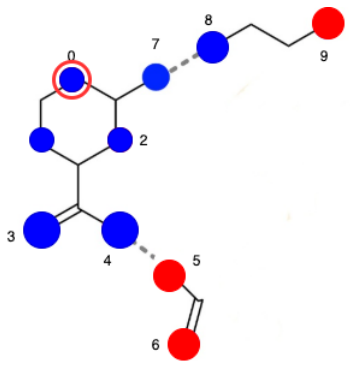


Figure 3.4: **Example of a molecule: red dots represent atoms 3 bounds away from atom 0**

Since the energy between two atoms is inversely proportional to the distance between those particles, it can be disregarded if the euclidean distance is longer than a user defined cutoff distance.This distance is usually between 10 and 12 Å.

Figure 3.5 shows an example of a range of atoms under a cutoff distance to the atom surrounded by the red circle. Atoms inside the dashed line make pairs with the central atom and their energy is computed.

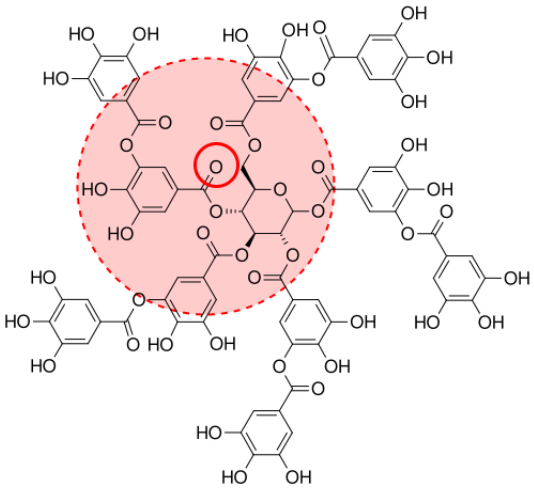


Figure 3.5: **Representation cutoff distance to the atom surrounded by the red line**

3.3 PROFILLING

Following finishing the conversion and development of the sequential version of the C++ code is necessary to study the application bottlenecks. This process is called profiling, and the tools used to do a more profound analysis were Dtrace, perf, and VTune. Dtrace runs on a personal laptop, perf, and VTune run on a multicore environment, the SeARCH cluster.

After several optimizations that will be described in the next chapter, the profiling results show that there are two functions where the program spends most of its time and resources, the angle and torsion, and the Van der Waals and Coulomb methods. Yet, the second function spent much more time than the first, initially spending 96% of the time (Figure 3.6), and after some optimizations, spending about 59% (Figure 3.7, green square). These values are five times more than the angle and torsion, which run in about 12% of the time. Thus it made sense to parallelize the second one.

Grouping: Source Function Stack

Source Function Stack	CPU Time: Total ▼	CPU Time: Self
Total	100.0%	0s
_start	100.0%	0s
__libc_start_main	100.0%	0s
main	100.0%	0s
vdwCoulombEnergy	96.1%	1132.262s
parseInfo	3.6%	0s
angleTorsionEnergy	0.3%	0.460s
bondEnergy	0.1%	0.250s
impropersEnergy	0.0%	0s

Figure 3.6: **VTune capture of the sequential version of Energy-Split code**

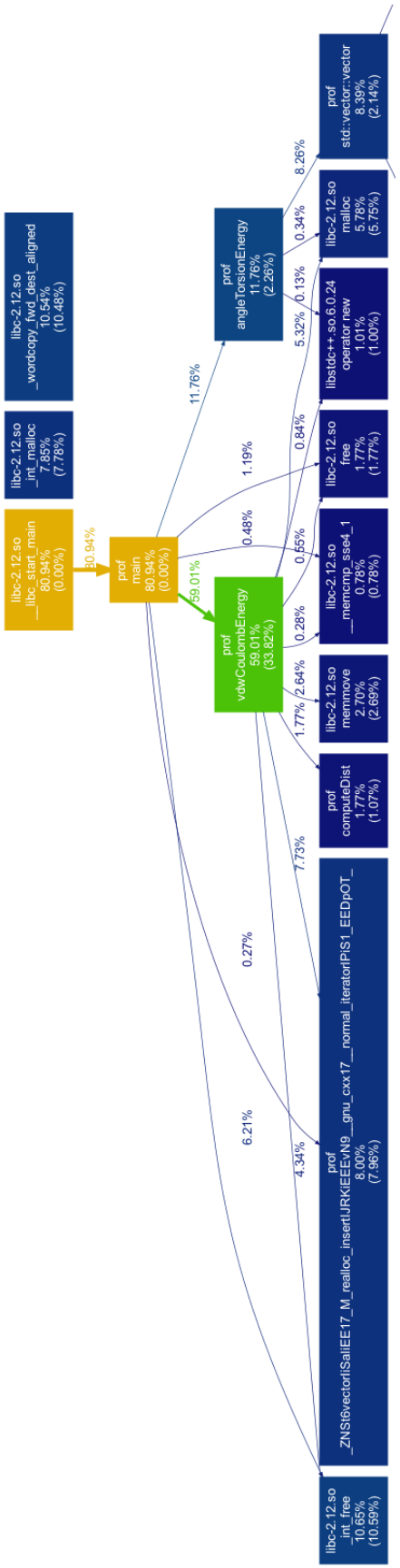


Figure 3.7: **Perf capture of the sequential version of the Energy-Split code**

3.4 THE PARALLEL C++ TOOL

After optimizing the sequential version to its best, it is time to develop a parallel one. To parallelize the Energy-Split code, first, it is necessary to analyze data dependencies, which does not exist because the energy computation of a pair of atoms is independent of the other. The only request is to do not repeat calculations. Thus, two slightly different versions were developed, a basic and a fragmented one, which is portrayed in Figure 3.8. To control concurrency and manage threads was used C++ and Boost mechanisms. Boost libraries are a complement to Standard Template Libraries (STL) from C++. Thus, they have great portability to other platforms and provide structures that are not present on STL, such as the thread pool and low level primitives like mutexes and conditional variables.

Boost libraries create a thread pool where Van Der Waals and Coulomb function run on one of a fixed number of threads. Mutex and conditional variables control synchronization and critical sections.

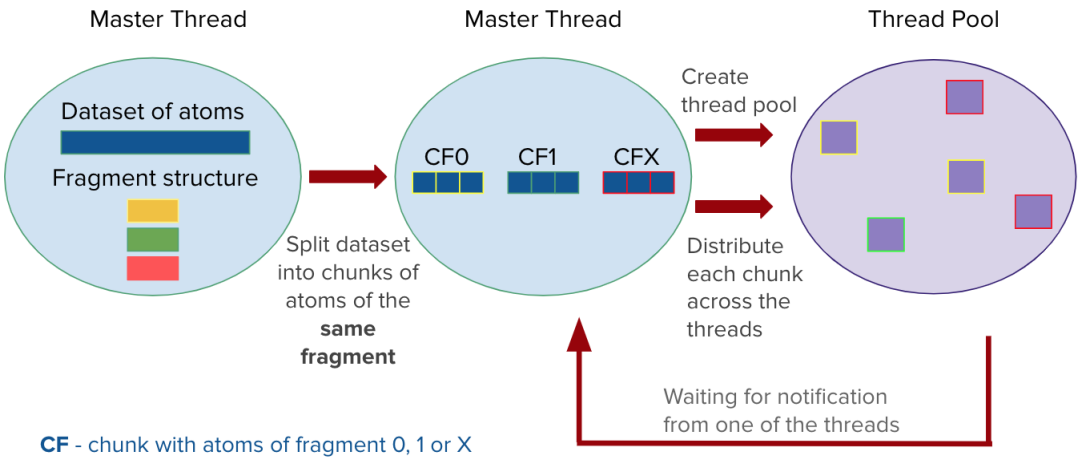


Figure 3.8: **Fragmented approach of the Energy-Split code**

3.4.1 *Partition, communication and mapping*

As was previously described, there are no data dependencies between the energy computations of the pairs of atoms. Thus, it made sense to implement a master-worker algorithm because the cost of communicating between master and workers is very lightweight.

So, the partition is the first phase of building a parallel application, and as we are facing a master-minion algorithm, this step is relevant to decide which chores are managed by the master thread. This thread is responsible for creating a pool of threads and all the scheduling between them. At first, it generates the same number of threads as the number of cores defined by the user. Then, the master creates vectors with atom's ids to pass to

workers so they can process them with the Van der Waals and Coulomb function. The communication was thought to avoid busy waiting, the master sleeps after distributing work, and it wakes when a thread has processed 80% of the atoms. Then, it attributes more work to the threads and sleeps again. This procedure is more efficient than creating a worker whenever one finishes working because it avoids the overhead inherent to thread creation. This cycle repeats until there are no more atoms to process, and when all work is completed, the master collects the results and presents them to the user.

3.4.2 *Basic version*

This version is very similar to the one described in Figure 3.8, with one difference: the data distribution. Instead of attribute data from a fragment to the threads and distribute threads according to the size of fragments, it is attributed 200 indexes in ascendant order. The bottleneck present in this approach is work distribution because workers that process lower indexes have much more work than the others. Assuming that the molecule has 1000 atoms, the worker that process atom 0 has to compare it with 999 other particles. In contrast, the thread that processes index 800 only has to make 200 computations.

3.4.3 *Fragmented version*

The fragmented parallel approach is portrayed in Figure 3.8. This idea has two advantages when comparing with the basic version: data distribution and memory coherence. The distribution of data is more even in this version because there is the same probability of any thread getting lower indexes, thus there is a balanced work distribution. Also, in the basic approach, there is no guarantee that false aliasing can not happen, as all threads in a process share the same heap. False aliasing or false sharing is a degrading performance pattern that exists when a party needs to reload a whole cache line even though there is no logical necessity to do it. That may happen when data shares a cache block with other data that is periodically modified.

So, this version guarantees that there is no false aliasing because threads do not share any variables or memory spaces. Each one writes the energy results in a private variable, and in the end, the master performs a reduction to sum up all results.

The work is distributed according to fragments and their sizes. Picture a dataset with 20000 atoms, and 3 fragments where Fragment 0 has 15 elements, Fragment 1 has 100, and Fragment X (remaining particles in the system except the ones from fragments 0 and 1) has 19885. With this scheme, and with four threads, one processes Fragment 0, the second processes Fragment 1, and the other two process Fragment X. When the first two finish

working in the smaller fragments, the master attributes them atoms from the remaining chunk. The last fragment has, then, four workers computing its indexes.

4

PERFORMANCE RESULTS AND DISCUSSION

This chapter shows the results of the sequential version as well as the parallel version. It starts by presenting the testbed environments that were used to validate and evaluate the Energy Split code versions. It then describes the required steps to reach the final sequential product and measures the execution time of each step. The parallel results come next with plots of execution times and speedups that depict the performance improvements, followed by a discussion of the overall results.

4.1 TESTBED ENVIRONMENT

This section presents the specifications of the environments used to run the Energy Split code, from resources to input datasets, and how measurements were taken. Table below describes the 3 different systems, one laptop and two servers in the SeARCH cluster. The code was compiled with the GNU C++ compiler on both machines: version 14 on the laptop and 11 on the server. Compiler flags used: -O3 and -stdlib=libc++.

The input dataset in the sequential runs had 13865 atoms, which is considered a small molecule. For the parallel version, the datasets had 8332, 13865, 33857, 195001, and 1802243 particles. The display of the measured execution times used the K-best approach out of 8 different runs: to average the K best measured times in milliseconds (in this case, K = 5).

	Laptop	**Server 1**	**Server 2**
Operating System	MacOS 10.14	CentOS 6.3	CentOS 7.4
CPU Device	Intel Haswell	Dual Intel Ivy Bridge	Dual Intel Skylake
CPU Model	Core i5-4278U	Xeon E5-2695	Xeon Gold 6130
#Sockets / #Cores per socket	1 / 2	2 / 12	2 / 16
Clock Frequency	2.6 GHz	2.4 GHz	2.1 GHZ
Vector Extensions	up to AVX2	up to AVX	up to AVX-512
Peak FP Performance	83.2 GFlops	460.8 GFlops	2150.4 GFlops
Peak Memory Bandwidth	30 GHz/s	47 GHz/s	119 GHz/s
#Memory Channels / socket	2	4	6
L1 Cache per core	2x32 KiB	2x32 KiB	2x32 KiB
L2 Cache per core	512 KiB	256 KiB	1 MiB
L3 Cache (shared)	3 MiB	30 MiB	22 MiB (non inclusive)
RAM Memory	8 GiB	2x32 GiB (NUMA)	2x48 GiB (NUMA)

Table 1: **Specifications of laptop and dual socket server**

4.2 Sequential results and optimizations

The development of the Energy Split code had five stages until reaching the final product, for the Van der Walls and Coulomb equations, as depicted in Figure 4.1. The expressions for bounded terms were portrayed to C++, and the optimizations made were simple, like keeping variables in registers, eliminating loop inefficiencies, simplify the formulas by converting all divisions into multiplications, and keeping the code vectorizable by the compiler to take advantage of its optimizations.

All tables below present results for a dataset with 13865 atoms, which is considered a small molecule.

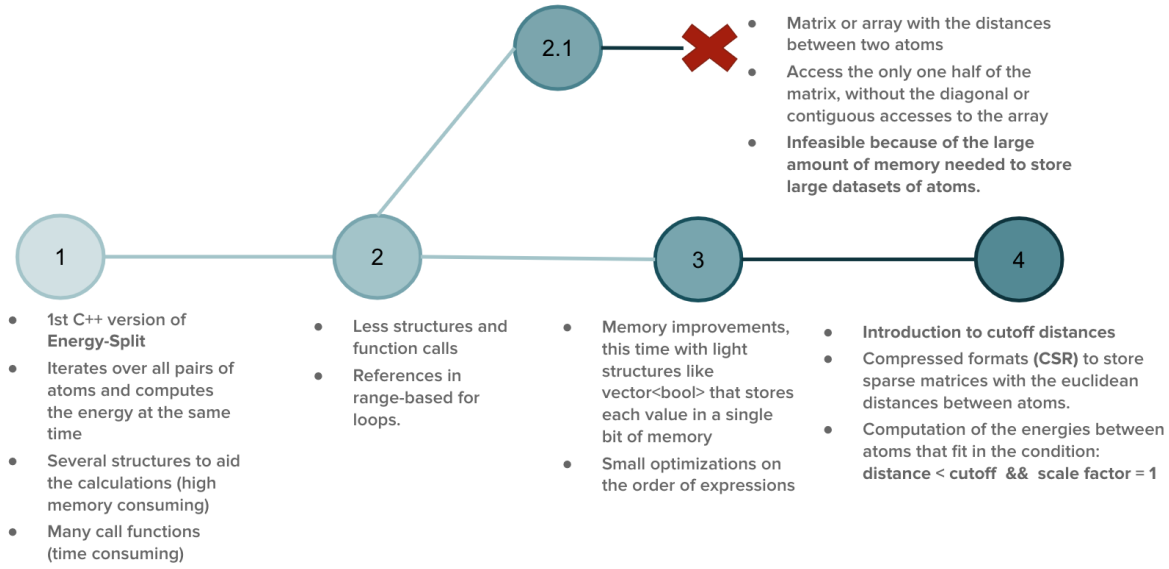


Figure 4.1: **Development steps of C++ sequential version**

Step 1

The TCL code was translated to C++ code, but many unnecessary function calls and massive auxiliary structures consume much memory. The first step was the simple conversion of the Tcl code, with the same order of operations. Table 2 shows that this implementation in C++ needs a lot of improvements and optimizations since it makes no sense that an interpreted language runs faster than a compiled one.

	Exec. Time (s)	Exec. Time (min)
Laptop Tcl	1,363.05	22.7
Laptop C++	8,722.47	145.37
Server 1 C++	1,978.99	57.69

Table 2: **Step 1, sequential execution time of a dataset with 13865 atoms**

Step 2

In this second step is when the process of optimizing begins. The function calls were eliminated instead of inlined to ensure there is no overhead because inlining is only a compiler request, and it may not perform that request in certain circumstances. The structures were changed to lighter ones; for example, whenever possible, unordered_map containers substitute maps, since these in C++ are internally built as a BST (Binary Search Tree), while unordered_maps are built as hash tables. This kind of structure is recommended when one wants a faster search, and the order can be arbitrary, which is the case when it comes to storing the Amber parameters of an equation, since they are arranged according to the type of the atom. Also, the for loops were adjusted to reference-based ones to avoid the penalty of making a copy of an element in each iteration.

Before moving to the next stages, it can be interesting to analyze the differences or improvements of these changes brought to the code, presented in Table 3. Execution times are also presented in minutes to understand better the magnitude of the time and how the optimizations are essential to the overall program performance.

	Exec. Time (s)	Exec. Time (min)
Laptop Tcl	1,363.05	22.7
Laptop C++	416.16	6.94
Server 1 C++	297.85	4.96

Table 3: **Step 2, sequential execution time of a dataset with 13865 atoms**

After some optimizations, it is noticeable the variation between the results. Second phase outcomes show the code runs 3.3 times faster than the Tcl version and 21 times faster than the first C++ version in a laptop. As for SeARCH server results, they are 12 times faster than the first measurements.

Step 2.1

A matrix with the distances between the atoms was created. The idea was to make the search efficient and simplify the computations. It would be a square matrix with NxN size (N is the total number of the particles in the system), where the position (0,15) represents the distance between atom 0 and 15. Thus, only the upper diagonal of the matrix was filled because the length between 0-15 is the same as 15-0. Also, the pairs of atoms that are connected by fewer bonds than three do not have the energy presented in the matrix because it was, previously, computed in other equations. Later, this approach was converted to a single array, joining the rows of the matrix to keep the accesses to the distances with a stride-1 access pattern, which reduces the latency of the memory accesses. Figure 4.2 depicts both approaches and conversion of the matrix to the array.

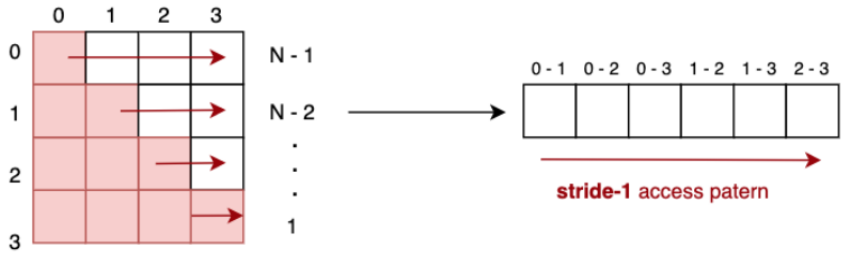


Figure 4.2: **Matrix and array approaches**

However, these approaches are feasible only for small datasets like ones with a maximum of near 130 000 atoms. For molecules with 200 000 atoms, it would be necessary around 149 GiB of memory for the matrix approach and 18,6 GiB for the array. The values were obtained by computing $matrix\_size \times matrix\_size \times 4bytes/1024/1024/1024$. Furthermore, the process of building the matrix was very inefficient due to the stride-N accesses to the matrix. Even so, for the small datasets, the performance increases compared to the first sequential version.

	Exec. Time (s)	Exec. Time (min)
Laptop Tcl	1,363.05	22.7
Laptop C++	266.78	4.45
Server 1 C++	239.09	3.98

Table 4: **Step 2.1, sequential execution time of a dataset with 13865 atoms**

Table 4 shows a 5 times speedup of the C++ code in the personal laptop, when compared with the Tcl code. And a speedup of 1.6 of version 2.1 over version 2. As for the dual socket server, this approach shows almost no difference from stage 2 with, a speedup of 1.2.

Step 3

So, upon invalidating the matrix approach, it is necessary to optimize even more the first code, which leads us to the third phase of the Energy Split program. Some structures that aid the traversal of the molecule occupied too much memory due to the increasing size of the datasets. So, they were changed to lighter ones like a vector of Booleans. In C++, vectors of Booleans store elements in a single byte of memory instead of the size of a Boolean variable (4 bytes). These changes favor memory optimization over processing, which was the first need in this case. This stage shows significant improvements in the results when compared with the ones before, as Table 5 show.

	Exec. Time (s)	Exec. Time (min)
Laptop Tcl	1,363.05	22.7
Laptop C++	35.67	0.59
Server 1 C++	23.97	0.4

Table 5: **Step 3, sequential execution time of a dataset with 13865 atoms**

The magnitude of the execution time switched to seconds instead of minutes. The code runs on a laptop 38 times faster than the same version in Tcl and 12 times faster than the previous C++ version. As for the server results, they were 10 times faster.

Step 4

In this stage, it was introduced a new feature. As the formulas show, there is an inverse correlation between the energy and the distance between atoms. So, the lengths considered to the energy computation are the ones typically less than 10 Å or 12 Å (this value is user defined). This feature reduces significantly the number of distances to compute. With fewer data makes sense to go back to a matrix format, because it is an efficient way to store data, since it takes advantage of the data locality. There was a search for a compressed format for sparse matrices that takes less space than to build a NxN matrix.

The compressed format that better fits the problem is CSR (Compressed Sparse Row). This format uses three one dimensional arrays to represent the sparse matrix, containing, respectively, the extents of rows (rowptr array), non-zero values (values array), and column indices (colind array), as shown in Figure 4.3. Rowptr array has the same size as the number of atoms + 1, and the remaining arrays have the same length, which is the number of non-zero values of the matrix. Thus, instead of occupying $N \times N \times 4\_bytes$ of space, it fills $(N + 1 + 2 \times NZ(non - zero)) \times 4\_bytes$. The number of non-zero values is always smaller than N.

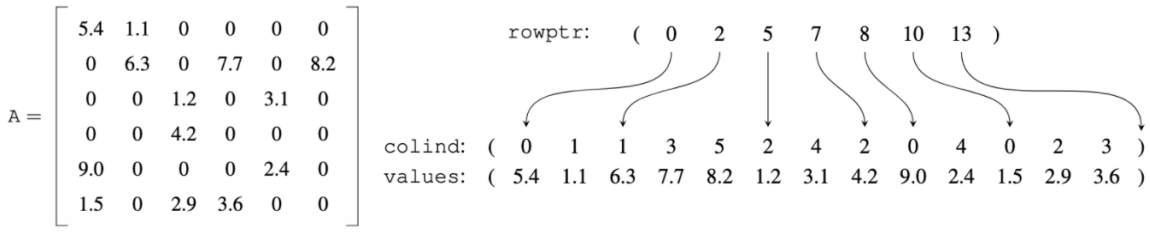


Figure 4.3: **Compressed Sparse Row format**

The values stored in the matrix have to follow two conditions:

- Scale factor - takes different values (configurable by the user) according to the bond distance between atoms:
  - if 1-2 or 1-3, default value: 0
  - if 1-4, default values: 0.5 for Van der Waals and 1/1.2 for Electrostatic
  - if +1-4, default value: 1
- Cutoff distance between two atoms:
  - Euclidean distance > 10 Å or 12 Å (user defined values)

The algorithm for CSR format is a traversal algorithm (for non-bonded energies) that follows a breadth-first search approach to compute and store energy for atoms with 1-2 and 1-3 (if the scale factor is different than 0) and 1-4 bonds. It fills colind, values, and rowptr arrays for the distance sparse matrix to compute Van der Waals and Electrostatic energies by iterating over the rowptr array to compute the energies with the distances on values array. Table 6 presents the results of this new version of the code.

	Exec. Time (s)	Exec. Time (min)
Laptop Tcl	1363.05	22.7
Laptop C++	19.46	0.32
Server 1 C++	13.23	0.22

Table 6: **Step 4, sequential execution time of a dataset with 13865 atoms**

This stage was the one where the code suffered more changes, it was inserted the concept of cutoff distance as explained before. As the previous version was already very optimized, the speedups were smaller than in phase 3. Nonetheless, the results were very similar on laptop and server, 1.8 times faster than the previous C++ version. Comparing to the Tcl version, the code is 71 times faster on laptop and 103 times faster on server.

4.2.1 *Summary*

To summarize, the optimizations made along with the different steps have always improved the algorithm's performance. Table 7 shows a compilation of the execution times, while Tables 8 and 9 depict the speedups according to Tcl code and the previous C++ code. Step 2.1 does not appear in the tables, since it was disregarded.

	Tcl	Step 1	Step 2	Step 3	Step 4
Laptop	22.7	145.37	6.94	0.59	0.32
Server 1	-	57.69	4.96	0.4	0.22

Table 7: **Compilation of the key code improvements (minutes)**

According to Table 7, server 1 has a better performance than the laptop, although an Ivy Bridge architecture precedes the Haswell. This algorithm executes more memory accesses as the size of the input file increases, so the memory happens to be the bottleneck of this algorithm, which makes the program memory-bound. As Table 1 shows, Server 1 has more channels per socket, more memory bandwidth than the laptop thus, it reads and stores faster from/into memory, which increases the program's performance. At this point, there are almost no optimizations, so the difference is higher than in the following steps. As far as the optimizations go, the lesser is the difference between laptop and server. However, it will always exist because of the different specifications of the hardware.

	Step 1	Step 2	Step 3	Step 4
Speedup Tcl	0.16	3.27	38.47	70.94
Speedup C++	-	20.95	11.76	1.84

Table 8: **Compilation of the laptop speedups**

	Step 1	Step 2	Step 3	Step 4
Speedup Tcl	0.39	4.58	56.75	103.18
Speedup C++	-	11.63	12.40	1.82

Table 9: **Compilation of the server 1 speedups**

Tables 8 and 9 show what was concluded by Table 7, the performance was better on Server 1, so as the speedups. The first step was the simple conversion of the code that brought no gains in what concerns execution time. Optimizations on step 3 were the more significant ones, and the bottleneck had to do with memory, so these enhancements brought the best results. From step 3 to 4, there was almost no improvement, showing that the code was already very optimized.

4.3 PARALLEL RESULTS

For the fragmented version of the parallel approach, some changes were necessary in the CSR format since the indexes of the atoms that were passed to threads were not continuous: for example, one worker could receive index 0-200 and, next time, 500-700. In the initial format, the rowptr vector keeps the number of elements of each row but can not keep track of the row number, which is essential to know the atom index. The solution was to create a new vector (row vector) to know which atoms belong to each row. Figure 4.4 depicts this new format.

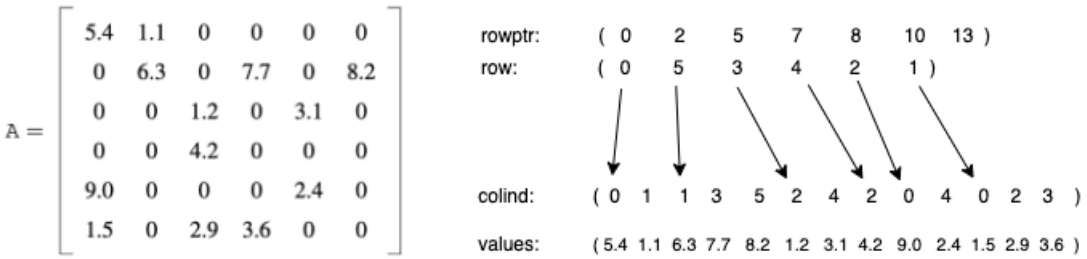


Figure 4.4: **New Compressed Sparse Row Format**

The parallel version was tested with five different datasets with 8332, 13865, 33857, 195001, and 1802243 atoms in the computer systems described in subsection 4.1. In the next section, these values are rounded, thus, they will appear as 9k, 14k, 34k, 200k, and 1,8M.

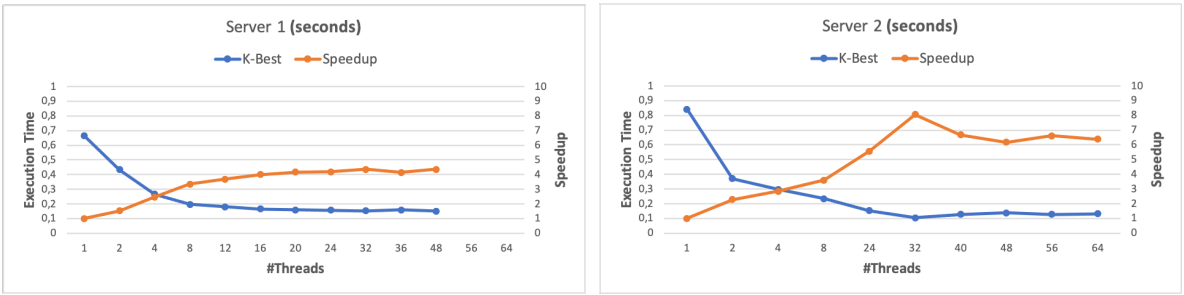


Figure 4.5: **Execution time and speedup of a 9k atoms input dataset**

Figure 4.5 shows that this small dataset has different behaviors in the servers. First chart shows a more intense growth until one full chip is used (12 threads). Using two chips or Simultaneous Multi-Threading do not show improvements due to the small size of the input file. Server 2 shows peak performance when using two full chips (32 threads). Simultaneous Multi-Threading does not seem to bring improvements for the same reason, small dataset.

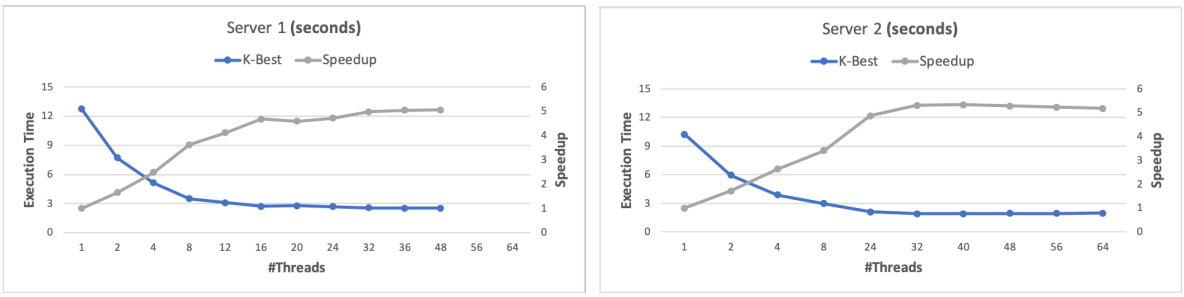


Figure 4.6: **Execution time and speedup of a 14k atoms input dataset**

Server 1 on Figure 4.6 shows peak performance at 16 cores, after, the performance gains stabilize. Server 2 present a similar behavior as before, a significant performance gain until reaching 32 threads, after it stabilizes.

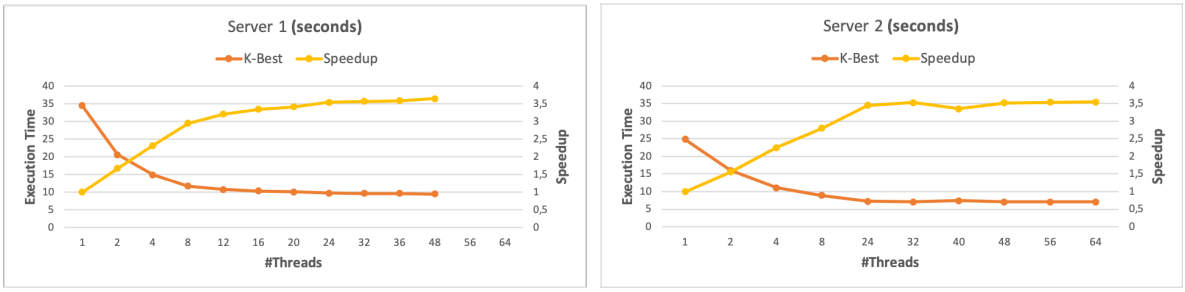


Figure 4.7: **Execution time and speedup of a 34k atoms input dataset**

For a medium size molecule, the behavior on both charts on Figure 4.7 is similar to the previous ones. Although, the growth is more intense until using 8 threads (Server 1) and 24 threads (Server 2). Both devices stabilizes performance from the moment when begin using 24 threads.

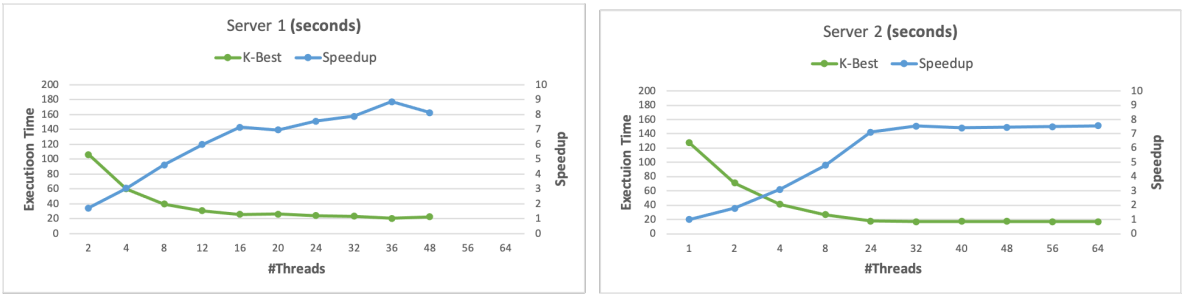


Figure 4.8: **Execution time and speedup of a 200k atoms input dataset**

Figure 4.8, on Server 1, the peak performance is reached with 36 threads, and the performance grows fast until using 16 threads, and reaching a speedup of 7. From 16 to 48 threads, the use of Simultaneous Multi-Threading boosts the speedups, but in a smaller ratio. In the second server, the behavior is much like the first server, but the gains are more significant until using 24 threads, then the speedup stabilizes between 7 and 8.

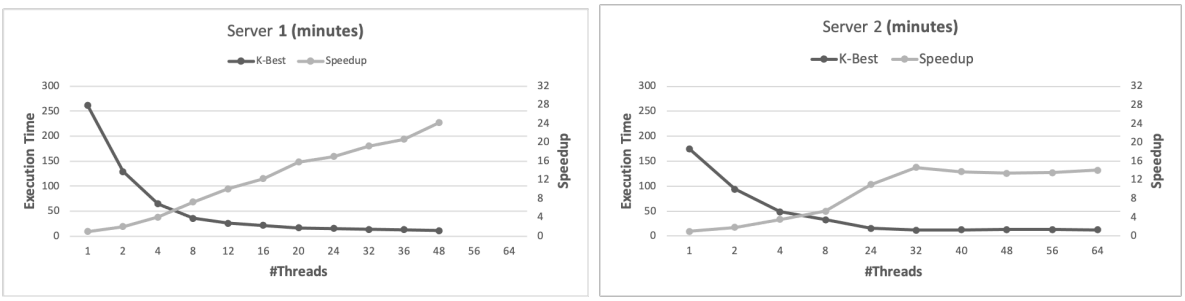


Figure 4.9: **Execution time and speedup of a 1,8M atoms input dataset**

The last figure stands from the others because the performance curve, on Server 1, has an almost linear growth. this behavior does not happen on Server 2, which maintains the previous pattern.

At this point, VTune was used to check if there was any evident bottleneck of this approach. Figure 4.10 was the first shot taken after developing the parallel version.

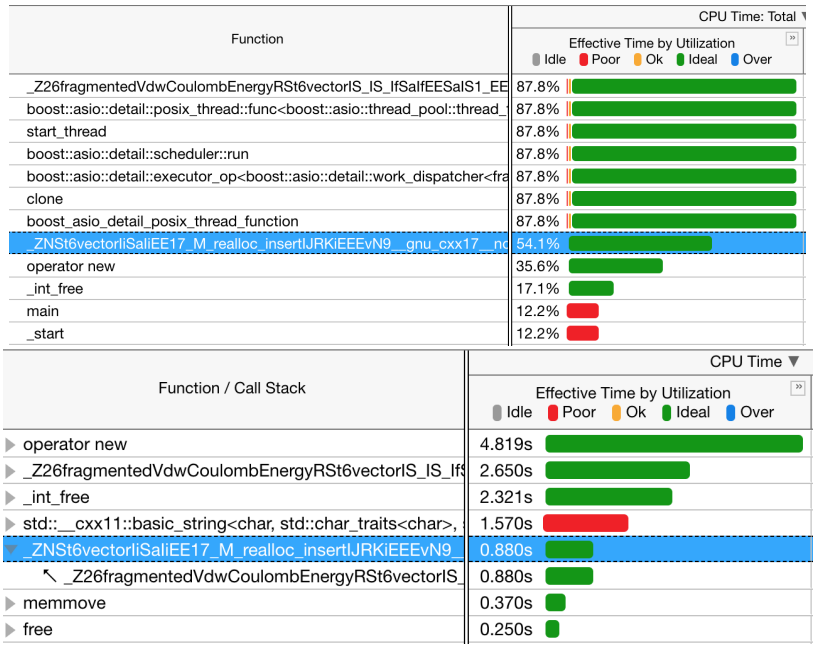


Figure 4.10: **VTune shots after development of the fragmented parallel version of the Energy-Split**

The line selected in blue shows a function where the program is spending too much time. Although the bar appears in green, the function is effectively using the CPU time. The right side of the figure shows which function performs that many reallocations and insertions: fragmentedVdwCoulombEnergy. This is the method where the fragmented parallel approach is implemented (described in Section 3.4.3). The reallocations and insertions happen when a thread processes one atom and removes it from a vector. That vector is the one passed to workers with all the atom's indexes to be processed by each thread. Removing an element from a vector with the pre-existing erase method causes the vector to reallocate all the posterior elements to their new positions. Thus, instead of removing an index after being processed, the components are removed when 80% of them are computed. Figure 4.11 shows how the percentage of reallocations and insertions are significantly lowered; however, this had almost no impact on the reduction of the overall program's execution time.

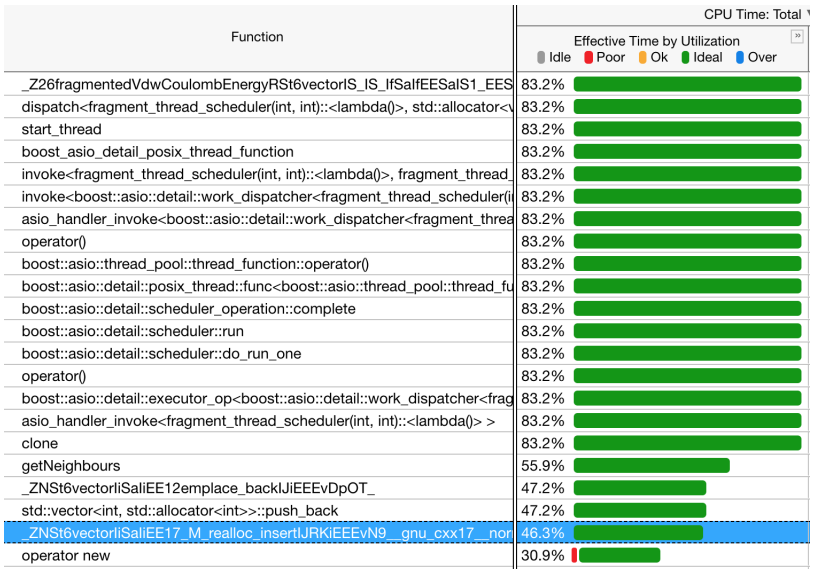


Figure 4.11: **VTune shot after reducing reallocations and insertions**

Figure 4.12 shows that after reducing the time spent on reallocating elements, another function appears as poorly using the CPU time, the skipChar function, which has never emerged, because other methods overlapped this one.

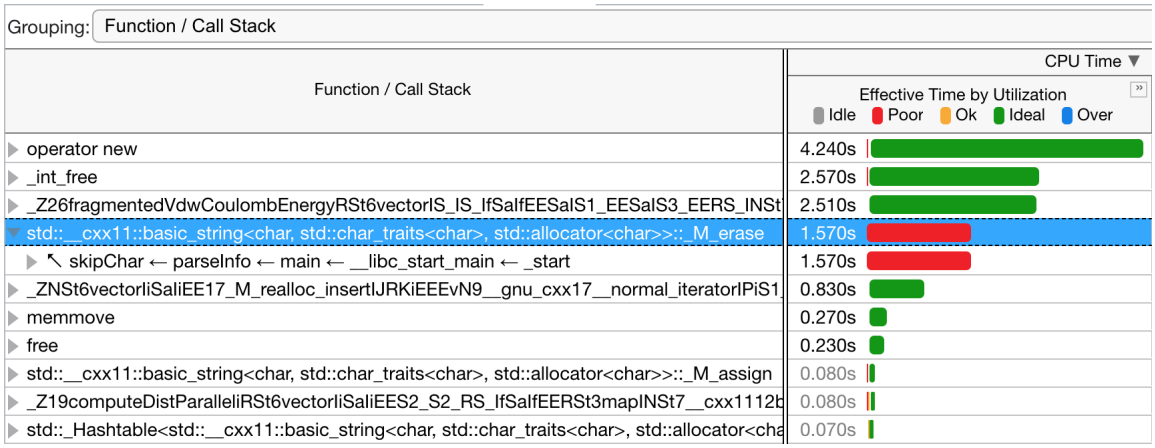


Figure 4.12: **VTune shot that shows new bottleneck on skipChar function**

This method is responsible for removing all the brackets from the input file. Its implementation is similar to fragmentedVdwCoulombEnergy when it comes to deleting elements, using the erase function from the std namespace of C++. So, picturing an input file with one million atoms, there have to exist space coordinates for all of them, each one with three components: x, y, and z. These coordinates are written as {xo yo zo}, and removing the first bracket causes the succeeding four elements to reallocate their positions. Thus, there are four reallocations for each existing coordinate, which is a million. Four million reallocations

just for one parameter of the input file bring an enormous overhead to the program that comes from memory latency.

The solution was using an algorithm from the C++ Standard Library that combines both erase and remove methods. The remove method uses iterators to move all the elements that do not fit the removal criteria to the front of the range. With just the erase method, if one vector has ten elements, and the first is removed, there are ten reallocations. If the remove method is used, instead of ten reallocations there is just one.

Table 10 shows the impact on execution times when using the erase-remove idiom.

	Before erase-remove (s)	After erase-remove (s)	Difference (s)
9k	0.63	0.15	-0.48
14k	3.75	2.52	-1.23
34k	17.08	9.46	-7.66
200k	264.59	22.49	-242.1
1,8M	37,260.70	650.50	-36,610.2 (∼10h)

Table 10: **Execution time, with 48 threads, before and after implementing the erase-remove idiom**

Thus, Table 10 clearly presents the need to pay attention when it comes to memory issues. By reducing the number of times one element is moved from one address to another in memory, it was possible to reduce the inherent latency, and drastically increase performance for larger input files.

4.4 DISCUSSION

The performance analysis of Energy Split targets two different experimental environments, a laptop that aims the execution on personal equipment, and two compute servers that test the efficiency of the solution on a single server in an HPC environment. At the same time, using different datasets allows testing the program's scalability as well as the influence of the connection's complexity between atoms. The performance results analysis focused on the speedup obtained when using multiple threads, as portrayed in Figure 4.13.

Figures from the previous section present similar behaviors between the two servers. The performance increases more until beginning to use the logic cores (with Simultaneous Multi-Threading, SMT, that Intel calls in this case 2-way Hyper-Threading), then the increasing rate is lower because the execution units of each core are shared between both threads. Also, the speedups with 48 threads on both machines are very similar, which shows good performance portability between servers.

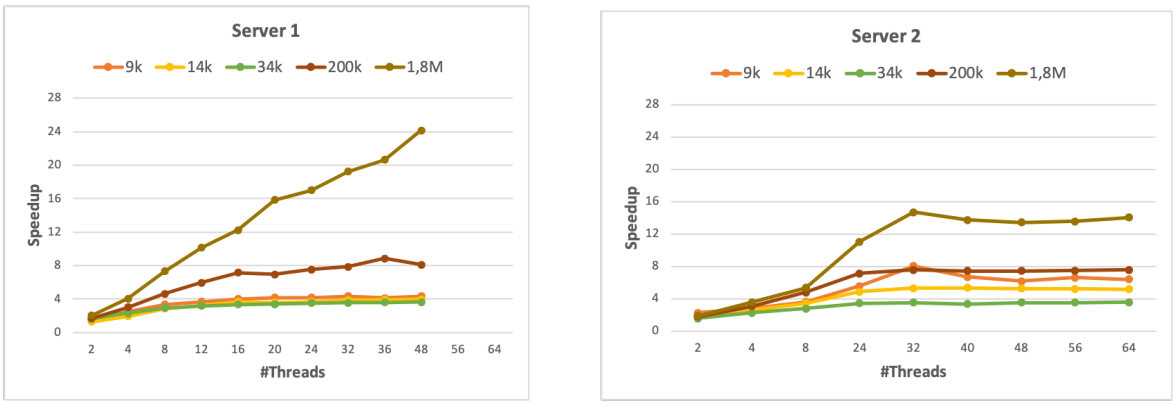


Figure 4.13: **Speedups of all datasets**

Figure 4.13 shows the scalability of the program, since the speedups are more significant with larger datasets. The behavior is different from Server 1 to Server 2. Server 1 chart presents growth even using two chips and SMT, in contrast with Server 2, which has its peak when using both chips without SMT (32 threads in 2x16 cores). When using the logic cores, the values stabilize. The speedup is higher on Server 1 since the sequential times are also higher in this machine due to the older architecture, lower memory bandwidth and floating point performance. Also, since the program has a memory bound pattern, it is expected that the sequential version should have better performance on the second server, due to its higher memory bandwidth. The fact that the program is memory bound explains the continuous growth of the Server 1 chart, since having a lower memory bandwidth is compensated by increasing the number of threads.

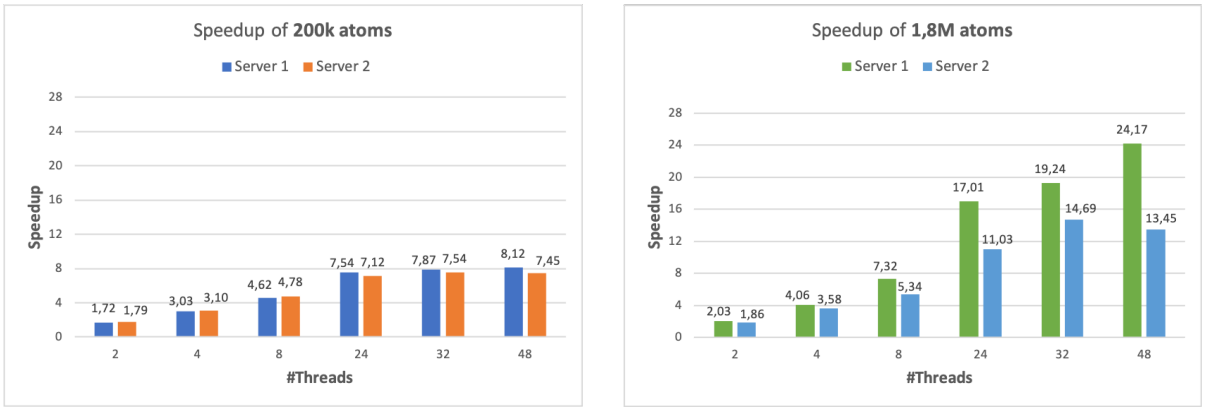


Figure 4.14: **Scalability of speedups of the two bigger datasets**

Figure 4.14 portrays the scalability of the program. Scalability is a measure of a parallel algorithm's capacity to efficiently use an increasing number of processors. The previous chart presented this combination of factors as the number of threads increased, where the speedup also increased. This occurrence is more visible with the second dataset, which is

nine times larger than the input file with 200k atoms. Figure 4.13 also shows scalability, since the speedups are more significant as the dataset sizes get larger and the number of threads increases, one can conclude the Energy-Split code is scalable. Ahmdal's law was used to validate this conclusion and to see if the obtained speedups are similar to the predicted. Ahmdal's law estimates the algorithm's maximum theoretical speedup when in a multiprocessor environment. Figure 4.15 shows the comparison between theoretical and obtained speedups with 200k and 1.8M atoms datasets.

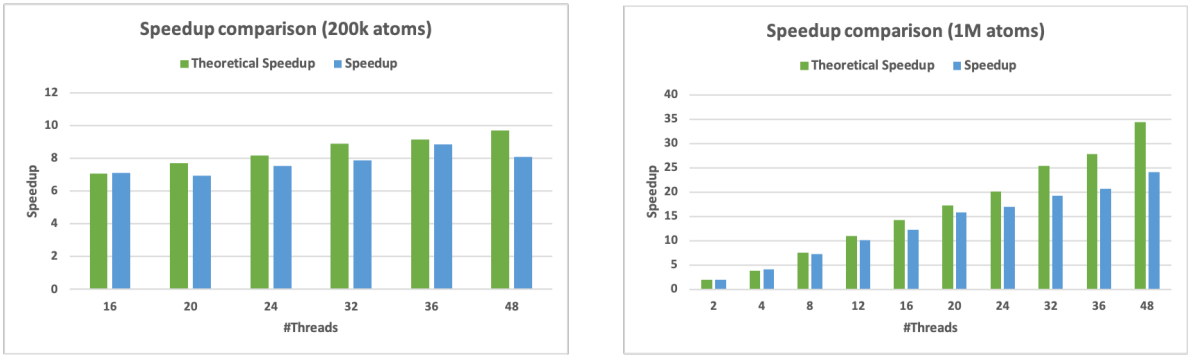


Figure 4.15: **Comparison between theoretical and obtained speedups on Server 1**

Previous speedup plots support the scalability of the program. The fact that the increasing number of threads increases performance shows that parallel processing is well suited to the problem. The atoms throughput, which evaluates the number of atoms processed per second, is also a metric to validate the parallel algorithm, as shown in Figures 4.16 and 4.17.

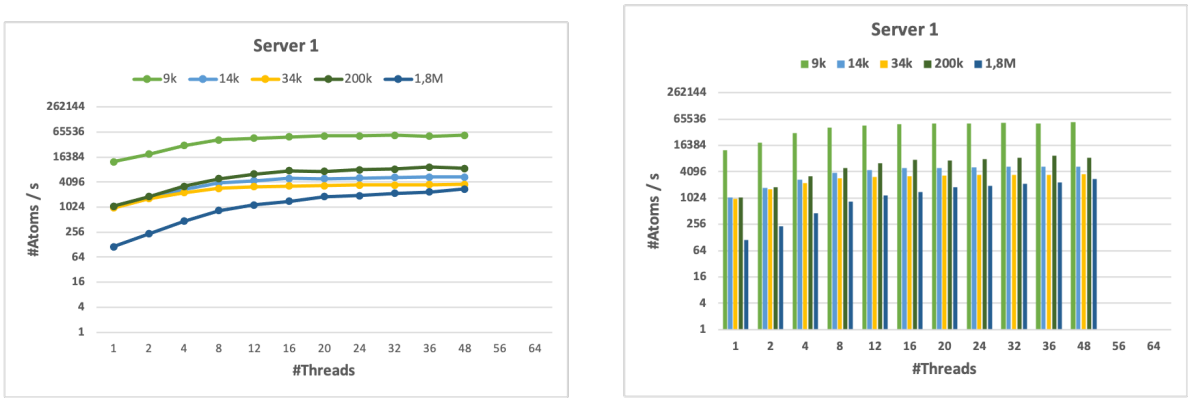


Figure 4.16: **Scalability of all datasets**

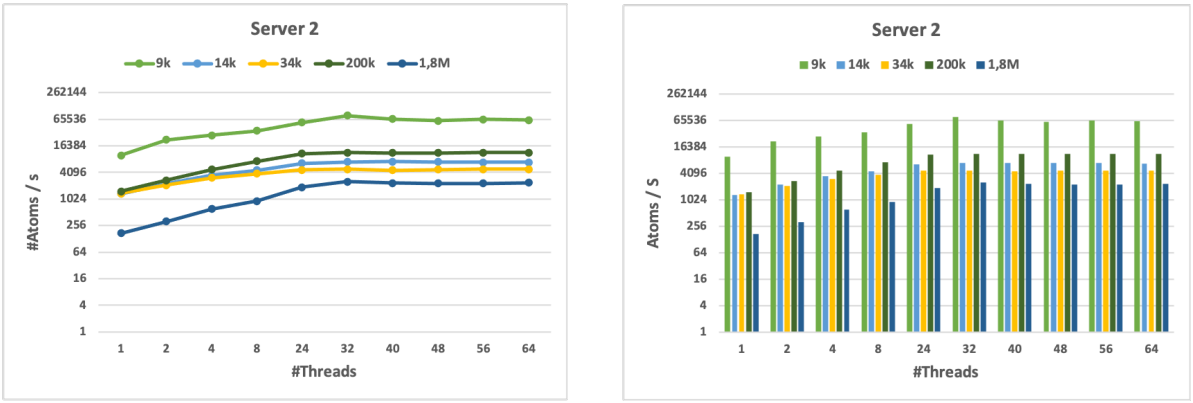


Figure 4.17: **Scalability of all datasets**

The previous four plots show that the algorithm has the same pattern on both servers. The smallest dataset's plot has fewer improvements in execution times because there is no compensation for the parallelization's overhead, yet there still are performance gains. The plot for the larger dataset has the same behavior as the others. In the beginning, this amount of atoms takes longer to process, but with an increase in the number of threads, there are good speedups, even though it takes longer to process this enormous molecule. The curious part takes place in the curves of the medium input dataset. Following the same logic as before, the larger the dataset, the longer the execution time per atom, so the lower the number of particles processed per second. However, the 14k and 34k atoms datasets have a longer execution time per particle than the input file with 200k atoms. It may be due to the complexity of the connection between the particles of the input file. Having a more complex network of particles takes longer to process than a more simple one. At the same time, it shows the algorithm's capability to take advantage of the available resources and the crescent atom throughput.

4.4.1 *Parallelization analysis with VTune*

VTune is a performance profiler that aids the developer in finding algorithm's bottlenecks and identifying where to focus on tuning his/her parallel application. After developing and already identifying the bottlenecks of the Energy-Split algorithm, it is essential to validate the tunings performed to the code.

The first parameter to analyze is the workload distribution, which can be identified by the amount of time each thread spends working. Figure 4.18 presents a shot from VTune, measured with one master thread and four workers, with the smallest dataset (9k atoms). As it shows, the last bar is the master thread because it runs sequentially until creating a thread pool with four workers. Around 2.5 seconds after the program initializes its execution, four workers begin to process the atoms, and after almost 3 seconds, at 5.6 seconds, all

the threads finish their work. This fact shows that the parallel algorithm has a balanced workload distribution.

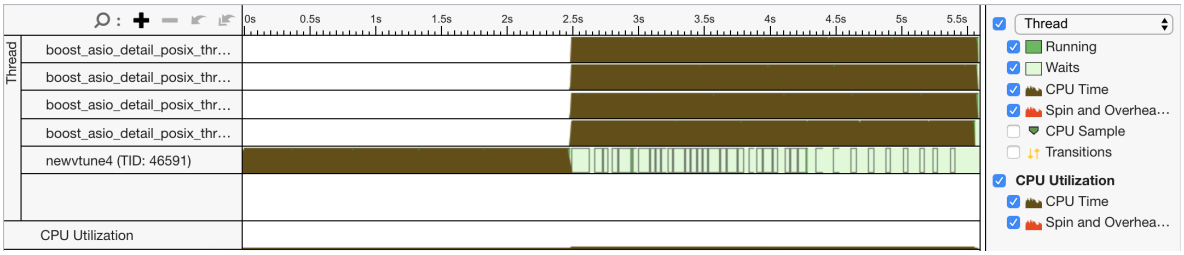


Figure 4.18: **Energy-Split's workload distribution of a dataset with 9k atoms using 4 threads, measured with VTune**

The brown bars represent the CPU time, and the green parts represent the time that the master thread spent waiting for the workers to ask for indexes to process. As it was described before, and Figure 4.19 confirms, there are no busy waitings because when the master is waiting, it is occupying no CPU time.

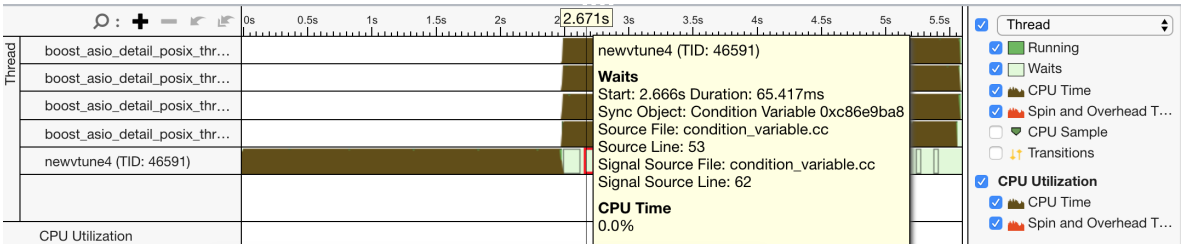


Figure 4.19: **CPU occupation by the master thread in Energy-Split parallel approach**

Concerning the scheduler, VTune also presents metrics to calculate its overhead, and Figure 4.19 can aid, visually, to realize the amount of CPU time spent by the scheduler, which is minimum, since it only uses the CPU to attribute atoms indexes to threads. So, after computing the results, it is concluded the scheduler overhead is 0.09%. This small value shows that the parallel portion of the Energy-Split code is correctly developed and, if some overhead is decreasing the performance, it does not come from the thread scheduling.

5

CONCLUSION

This document describes all the steps, changes, optimizations to reach the final product, a combination of chemistry and high-performance computing: the Energy-Split tool. The Energy-Split is a tool that receives a file with parameters that define a molecular system, which can take enormous proportions, and computes its intramolecular energy. The system can be split into several fragments. So the Energy-Split is capable of computing, separately, each fragment's energy and the total energy of the system with an execution time significantly lower than its original Tcl version. The biggest challenges were the computation of thousands or millions of calculations, as well as efficiently managing the memory. When developing the algorithm, it became apparent that it fits in the memory-bound pattern, which explains the results obtained.

The tool is to run on laptops although, it was tested and developed on multicore environments too. The testbed environment involved three different devices, one laptop, and two multicore servers, as well as five input files of distinct molecule sizes. The development of the Energy-Split involved two versions, one sequential and one parallel. The results were validated with the Intel VTune Profiler. The performance of the sequential version was improved by around 100 times when compared to the original version on a dual-socket server. The results showed good scalability and the portability of the efficiency between architectures. The atom throughput increased drastically from the Tcl version to the C++ version. Initially, for a dataset with 8332 atoms, the Energy Split could compute 23 particles per second, and now it calculates the energy of 21811 in one second.

5.1 FUTURE WORK

Initially, one of the objectives of this thesis was to evaluate the resulting efficient implementations with computing accelerators (f.e, Graphics Processing Unit devices). Although, in the development phase, it was clear that it would be a misfit for GPUs since the code has very irregular patterns to access memory, namely stride-n accesses. Also, it does not take advantage of SIMD (Single Instruction Multiple Data) instructions, which makes it unsuitable for GPU architecture, and it will not have a better performance than in the CPU.

However, there is some future work that can be implemented, as the extension of the Energy-Split code to run with multiple frames, as described in chapter 2, section 2.1.2. This new feature will allow the computation of the delta energy through time. Also, this implementation could be suitable to test on heterogeneous environments with a combination of tools like StarPU, Legion and HEP-frame, since the amount of parallelism would be increased by adding frames. The development of an adequate GUI (Graphical User Interface) also would be a handy feature to add to present the results of the Energy-Split computations.

BIBLIOGRAPHY

[1] Jen Hsin, Anton Arkhipov, Ying Yin, John E. Stone, and Klaus Schulten. Using vmd: An introductory tutorial. *Current Protocols in Bioinformatics*, 24(1):5.7.1–5.7.48, 2008.

[2] Junmei Wang, Romain Wolf, James Caldwell, Peter Kollman, and David Case. Development and testing of a general amber force field. *Journal of computational chemistry*, 25:1157–74, 07 2004.

[3] J E Lennard-Jones. Cohesion. *Proceedings of the Physical Society*, 43(5):461–482, sep 1931.

[4] André Pereira and Alberto Proença. Hep-frame: Improving the efficiency of pipelined data transformation & filtering for scientific analyses. *Computer Physics Communications*, 263:107844, 2021.

[5] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, 3rd edition, 2015.

[6] Michael Edward Bauer. Legion: Programming distributed heterogeneous architectures with logical regions. 2014.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23, 08 2009.

[8] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall, 2011.

[9] Gprof, gnu gcc profiling tool. https://en.wikipedia.org/wiki/Gprof.

[10] Gabriel Pénega. Perf, a performance monitoring and analysis tool for linux. https://www.tecmint.com/perf-performance-monitoring-and-analysis-tool-for-linux/.

[11] Intel Corporation. Intel advisor. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

[12] Intel Corporation. Intel vtune amplifier. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

This work was supported by FCT (Fundação para a Ciência e Tecnologia) within project RDB-TS: Uma base de dados de reações químicas baseadas em informação de estados de transição derivados de cálculos quânticos (Refª BI2-2019_NORTE-01-0145-FEDER-031689_UMINHO), co-funded by the North Portugal Regional Operational Programme, through the European Regional Development Fund.