University "Politehnica" of Bucharest

Faculty of Electronics, Telecommunications and Information Technology

N-Body Problem, Application, on a Map-Reduce Accelerator, to Molecular Dynamics

Dissertation Thesis

submitted in partial fulfilment of the requirements for the **Degree of Master of Science** in the domain of ***Electronics and telecommunications***, study program ***Advanced Microelectronics***

Thesis Advisor: Student:

Prof. univ. Dr. Ing. Gheorghe ȘTEFAN ***Ing. David MIHĂIȚĂ***

– 2016 –

Annex 2 bis

University "Politehnica" of Bucharest

Faculty of Electronics, Telecommunications and Information Technology

Department Electronic Devices, Circuits and Architectures

Master Program Director's Approval:

Prof. univ. Dr. Ing. Claudius DAN

DISSERTATION THESIS

of student MIHĂIȚĂ V. A. David, Advanced Microelectronics

1. Thesis title: *N-body problem, application, on a Map-Reduce accelerator, to molecular dynamics*

2. The student's original contribution will consist of:

- *adapting algorithms used in Gromacs simulation package to the used Map-Reduce architecture;*
- *updating and adapting the Map-Reduce architecture to accommodate used algorithms;*
- *writing and testing accelerated routines*

3. The Intellectual Property upon the project belongs to: *UPB*

4. The research is performed at the following location: *UPB*

5. The thesis project was issued at the date: *14 September 2015*

THESIS ADVISOR:

Prof. univ. Dr. Ing. Gheorghe ȘTEFAN

STUDENT:

David MIHĂIȚĂ

Statement of Academic Honesty

I hereby declare that the thesis "*N-body problem, application, on a Map-Reduce accelerator, to molecular dynamics*", submitted to the Faculty of Electronics, Telecommunications and Information Technology in partial fulfilment of the requirements for the degree of Engineer/Master of Science in the domain *of electronics and telecommunications*, study program *Advanced Microelectronics*, is written by myself and was never before submitted to any other faculty or higher learning institution in Romania or any other country.

I declare that all information sources I used, including the ones I found on the Internet, are properly cited in the thesis as bibliographical references. Text fragments cited "as is" or translated from other languages are written between quotes and are referenced to the source. Reformulation using different words of a certain text is also properly referenced. I understand plagiarism constitutes an offence punishable by law.

I declare that all the results I present as coming from simulations or measurements I performed, together with the procedures used to obtain them, are real and indeed come from the respective simulations or measurements. I understand that data faking is an offence punishable according to the University regulations.

Bucharest, *1st of June, 2016*

David MIHĂIȚĂ

(student's signature)

TABLE OF CONTENTS

Table of Contents 9

Table of figures 13

Tables 14

Terms and abbreviations 15

Introduction 17

1. Parallel computation and programming 1-23

1.1. Parallelism vs. Concurrency 1-25

1.2. Amdahl's law [15] 1-26

1.2.1. Definition 1-26

1.2.2. Examples 1-27

1.2.3. Speedup in a serial program [16] 1-28

1.3. Gustafson's law [17] 1-29

1.4. Amdahl's Law or Gustafson's Law [19] 1-32

1.4.1. Lemma 1 1-33

1.4.2. Lemma 2 1-33

1.4.3. What's does it imply 1-33

1.4.4. What does it not imply 1-34

1.4.5. Shortcoming of Gustafson's Law 1-34

1.4.6. What can programmers do? 1-34

1.4.7. What can architects do? 1-34

1.5. When Amdahl's law is inapplicable? [20] 1-35

1.5.1. Gene Amdahl's insight 1-35

1.5.2. The equation 1-36

1.5.3. Simplifying assumptions 1-36
1.5.4. Who can use Amdahl's law? 1-37
1.5.5. Who should avoid using the term "Amdahl's law"? 1-37
1.5.6. Concrete Example 1-38
1.6. Types of serial bottlenecks [21] 1-40
1.6.1. Some background 1-41
1.6.2. Who should this concern? 1-41
1.7. On Using Dependence Information [22] 1-43
1.7.1. Definition of tasks 1-43
1.7.2. What are task dependencies? 1-43
1.7.3. How does it apply to loops? 1-44
1.7.4. How it applies to parallel programming? 1-44
1.8. Understanding Critical Sections impact [23] 1-47
1.8.1. Update Critical Sections 1-47
1.8.2. Fine-grain critical sections 1-48
1.8.3. Reduction Critical Sections 1-49
1.8.4. Cache Locality: A second order effect 1-50
1.8.5. Looking forward 1-50
2. Software, hardware and competition 2-53
2.1. Gromacs 2-53
2.2. The global MD algorithm 2-55
2.3. Map-Reduce architecture 2-57
2.3.1. History 2-57
2.3.2. Connex architecture 2-57
2.4. Competition 2-60

2.4.1. Anton....2-60
2.4.2. Comparison between Anton and other machines....2-62
2.4.3. Anton computation time for researchers....2-62
3. Implementation....3-63
3.1. System variables....3-63
3.2. Periodic boundary conditions....3-63
3.3. Pseudocode implementation of the PBC....3-65
3.4. Neighbour searching....3-66
3.5. Pseudocode implementation of the neighbour search....3-67
3.6. Force computation....3-68
3.7. Pseudocode implementation of force computation....3-69
3.8. Temperature coupling....3-70
3.9. Pseudocode for temperature coupling....3-70
3.10. Coordinate and velocity updating....3-71
3.11. Pseudocode for coordinate and velocity updating....3-72
4. Results and conclusions....4-73
5. References....6-77
Annexe 1 – Code....6-79

TABLE OF FIGURES

Figure 0-1: a) DNA, b) Viral budding, c) Kinesin 17
Figure 0-2: a) Clathrin, b) tRNA, c) Vesicle (curtesy XVIVO) [1] 17
Figure 0-3: Bovine Pancreatic Trypsin Inhibitor 19
Figure 1-1: IBM Supercomputers 1-23
Figure 1-2: A cabinet from IBM's Blue Gene/L massively parallel supercomputer 1-24
Figure 1-3: SVG Graph illustrating Amdahl's law 1-27
Figure 1-4: Speedup under Amdahl's Law [17] 1-30
Figure 1-5: Fixed size model (Speedup = 1/(s + p/N)) 1-31
Figure 1-6: Scaled-Size Model (Speedup = s + Np) 1-32
Figure 2-1: GROMACS global MD algorithm [25] 2-56
Figure 2-2: Connex general architecture 2-58
Figure 2-3: Connex array module 2-59
Figure 2-4: Connex array execution unit 2-60
Figure 2-5 Anton supercomputer and insides 2-61
Figure 3-1: Schematic representation of the idea of periodic boundary conditions [29] 3-65
Figure 3-2: Neighbors for the current particle [30] 3-66
Figure 3-3: A graph of strength versus distance for the 12-6 Lennard-Jones potential. 3-69
Figure 3-4: GROMACS update algorithm [25] 3-72

TABLES

Table 1 Comparison of MD simulation speeds (all-atom, explicit solvent, standard DHFR benchmark) [28] 2-62
Table 2 Largest published molecular dynamics simulations (all-atom simulations of proteins in explicit solvent) 2-62
Table 3: Simulation cycles and percent time spent on algorithm ...5-75
Table 4: Performance for non-bonded interactions [33] [34] 5-75
Table 5: MRA cell usage 5-76

TERMS AND ABBREVIATIONS

ACS – Accelerating Critical Section

ASIC – Application-specific integrated circuit

BPTI – bovine pancreatic trypsin inhibitor

CPU – central processing unit

CS – Critical Section

DNA – deoxyribonucleic acid

EU – execution unit

FPGA – Field-Programmable Gate Array

I/O – input / output

L-J potential – Lennard-Jones potential

MRA – Map-Reduce architecture

MD – molecular dynamics

MPP – massive parallel processing

NIH – National Institutes of Health

NMR – nuclear magnetic resonance

NRBSC – National Resource for Biomedical Supercomputing

PBC – Periodic boundary conditions (PBCs)

SIMD – Single Instruction, Multiple Data

SOC – System-on-chip

TM – Transactional memory

VdW – Van der Waals

INTRODUCTION

One of the foremost tools in the theoretical study of biological molecules is the technique of molecular dynamics simulations (MD). This computational method calculates the time dependent behaviour of a molecular system. MD simulations have provided detailed information on the fluctuations and conformational changes of proteins and nucleic acids.

Figure 0-1: a) DNA, b) Viral budding, c) Kinesin

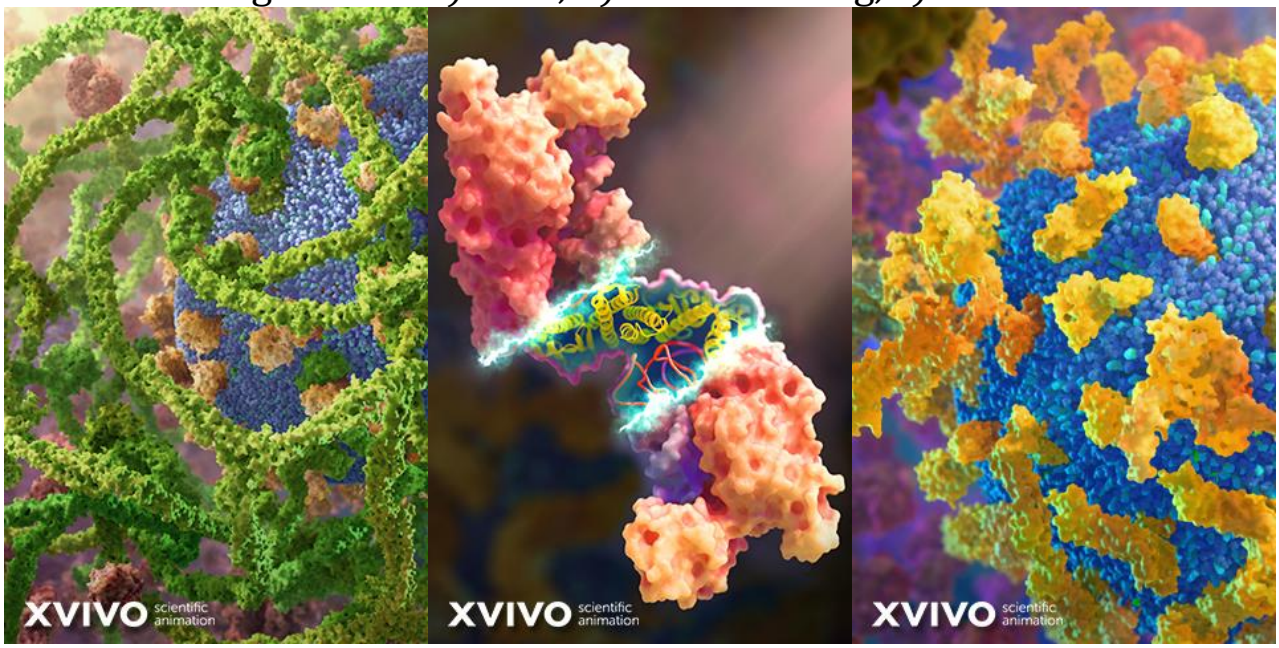


Figure 0-2: a) Clathrin, b) tRNA, c) Vesicle (curtesy XVIVO) [1]

These methods are now *routinely used to investigate the structure, dynamics and thermodynamics of biological molecules* and their

complexes. They are also used in the determination of structures from X-ray crystallography and from NMR experiments.

MD simulations are necessary because we don't know the structure of most proteins and we don't know how they work together, so, for elucidating structural dynamics of proteins there are two major approaches:

- Laboratory experiments ("wet lab") which are hard, since
    - Atoms are small
    - Difficult to get small pictures, much less movies
- Biophysical simulation ("dry lab")
    - Gold standard for protein-sized systems
    - MD simulations

Primary uses for MD:

- Determine structures by watching them form
- Understand dynamics by watching things move
- Transform messy wet stuff into nice dry data mining

Biological molecules exhibit a wide range of time scales over which specific processes occur; for example [2]:

1. Local Motions (0.01 to 5 Å, 10^{-15} s to 10^{-1} s)
    - Atomic fluctuations
    - Sidechain Motions
    - Loop Motions
2. Rigid Body Motions (1 to 10Å, 10^{-9} s to 1 s)
    - Helix Motions
    - Domain Motions (hinge bending)
    - Subunit motions
3. Large-Scale Motions (> 5Å, 10^{-7} s to 10^{4} s)
    - Helix coil transitions
    - Dissociation/Association
    - Folding and Unfolding

Many important biological phenomena occur on timescales between 10μs and 1ms:

- Major structural changes;
- Interactions of proteins with other proteins, nucleic acids and drug molecules;

- Folding of many proteins.

The molecular dynamics method was first introduced by Alder and Wainwright in the late 1950's [3] [4] to study the interactions of hard spheres. Many important insights concerning the behaviour of simple liquids emerged from their studies. The next major advance was in 1964, when Rahman carried out the first simulation using a realistic potential for liquid argon [5]. The first molecular dynamics simulation of a realistic system was done by Rahman and Stillinger in their simulation of liquid water in 1974 [6]. The first protein simulations appeared in 1977 with the simulation of the bovine pancreatic trypsin inhibitor (BPTI) [7]. Today in the literature, one routinely finds molecular dynamics simulations of solvated proteins, protein-DNA complexes as well as lipid systems addressing a variety of issues including the thermodynamics of ligand binding and the folding of small proteins. The number of simulation techniques has greatly expanded; there exist now many specialized techniques for particular problems, including mixed quantum mechanical - classical simulations, that are being employed to study enzymatic reactions in the context of the full protein. Molecular dynamics simulation techniques are widely used in experimental procedures such as X-ray crystallography and NMR structure determination.

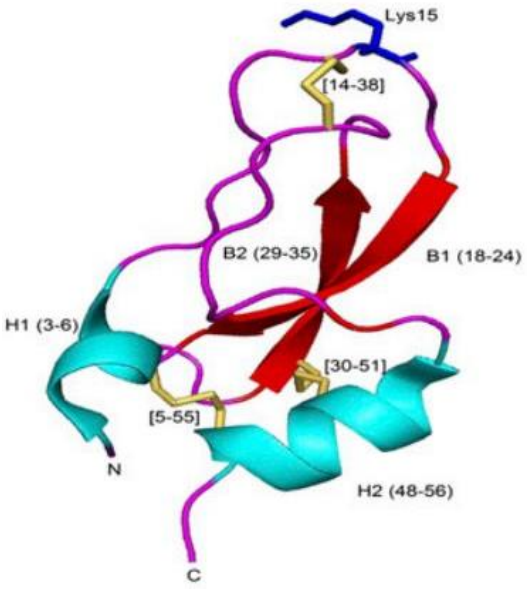


Figure 0-3: Bovine Pancreatic Trypsin Inhibitor

Molecular dynamics simulations generate information at the microscopic level, including atomic positions and velocities. The conversion of this microscopic information to macroscopic observables such as pressure, energy, heat capacities, etc., requires statistical mechanics. Statistical mechanics is fundamental to the study of biological systems by molecular dynamics simulation. For more detailed information, refer to the numerous excellent books available on the subject. [8]

In a molecular dynamics simulation, one often wishes to explore the macroscopic properties of a system through microscopic simulations, for example, to calculate changes in the binding free energy of a particular drug candidate, or to examine the energetics and mechanisms of conformational change. The connection between microscopic simulations and macroscopic properties is made via *statistical mechanics* which provides the rigorous mathematical expressions that relate macroscopic properties to the distribution and motion of the atoms and molecules of the N-body system; molecular dynamics simulations provide the means to solve the equation of motion of the particles and evaluate these mathematical formulas. With molecular dynamics simulations, one can study both thermodynamic properties and/or time dependent (kinetic) phenomenon.

All-atom molecular dynamics simulations provide a vehicle for capturing the structures, motions, and interactions of biological macromolecules in full atomic detail.

Recent years have seen substantial advances in both the timescales accessible to molecular dynamics simulations and in the quality of the force fields used in such simulations. Together, these developments have led to dramatic improvements in the ability of molecular dynamics simulations to capture the structure and dynamics of proteins.

Access to longer timescales and the improved sampling of conformations has been enabled by progress in a number of areas.

A specialized computer for molecular dynamics simulations, called Anton, has allowed us to access long-timescale (up to 1 millisecond) dynamics in proteins using all-atom simulations with an explicit representation of solvent molecules.

The last five years have also seen substantial improvements in the force fields used in molecular dynamics simulations. In this area, NMR spectroscopy has played a central role by providing a wealth of experimental data reporting on a broad range of structural and dynamical properties of peptides and proteins; such data are ideally suited to validate molecular dynamics simulations.

Having access to long and accurate molecular dynamics simulations, it is in turn possible to provide new insight in to the dynamical properties of proteins, finding new drugs and treatments to major ailments.

1. PARALLEL COMPUTATION AND PROGRAMMING

Parallel computing is a type of computation in which many calculations are carried out simultaneously [9], operating on the principle that large problems can often be divided into smaller ones, which are then solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. [10] As power consumption (and consequently heat generation) by computers has become a concern in recent years, [11] parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core processors. [11]

Figure 1-1: IBM Supercomputers

Figure 1-2: A cabinet from IBM's Blue Gene/L massively parallel supercomputer

Parallel computing is closely related to concurrent computing—they are frequently used together, and often conflated, though the two are distinct: it is possible to have parallelism without concurrency (such as bit-level parallelism), and concurrency without parallelism (such as multitasking by time-sharing on a single-core CPU). [12] In parallel computing, a computational task is typically broken down in several, often many, very similar subtasks that can be processed independently and whose results are combined afterwards, upon completion. In contrast, in concurrent computing, the various processes often do not address related tasks; when they do, as is typical in distributed computing, the separate tasks may have a varied nature and often require some inter-process communication during execution.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

In some cases, parallelism is transparent to the programmer, such as in bit-level or instruction-level parallelism, but explicitly parallel algorithms, particularly those that use concurrency, are more difficult to write than sequential ones, [13] because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

A theoretical upper bound on the speed-up of a single program as a result of parallelization is given by Amdahl's law.

1.1. PARALLELISM VS. CONCURRENCY

The term *Parallelism* refers to techniques to make programs faster by performing several computations in parallel. This requires hardware with multiple processing units. In many cases the sub-computations are of the same structure, but this is not necessary. Graphic computations on a GPU are parallelism. Key problem of parallelism is to reduce data dependencies in order to be able to perform computations on independent computation units with minimal communication between them. To this end it can be even an advantage to do the same computation twice on different units. [14]

The term *Concurrency* refers to techniques that make program more usable. Concurrency can be implemented and is used a lot on single processing units, nonetheless it may benefit from multiple processing units with respect to speed. If an operating system is called a multi-tasking operating system, this is a synonym for supporting concurrency. If you can load multiple documents simultaneously in the tabs of your browser and you can still open menus and perform more actions, this is concurrency. [14]

If you run distributed-net computations in the background while working with interactive applications in the foreground, that is concurrency. On the other hand, dividing a task into packets that can be computed via distributed-net clients, this is parallelism. [14]

1.2. AMDAHL'S LAW [15]

1.2.1. Definition

In computer architecture, Amdahl's law gives the *theoretical speedup* in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved. It is named after computer scientist Gene Amdahl, and was presented at the AFIPS Spring Joint Computer Conference in 1967.

Amdahl's law states:

$$S = \frac{1}{1-p+\frac{p}{s}} \tag{1-1}$$

Where:

- S – theoretical speedup of the whole task;
- s – speedup in the latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- p – percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

With the added restriction that:

$$S \leq \frac{1}{1-p} \tag{1-2}$$

Meaning that the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless the magnitude of the improvement, *the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.*

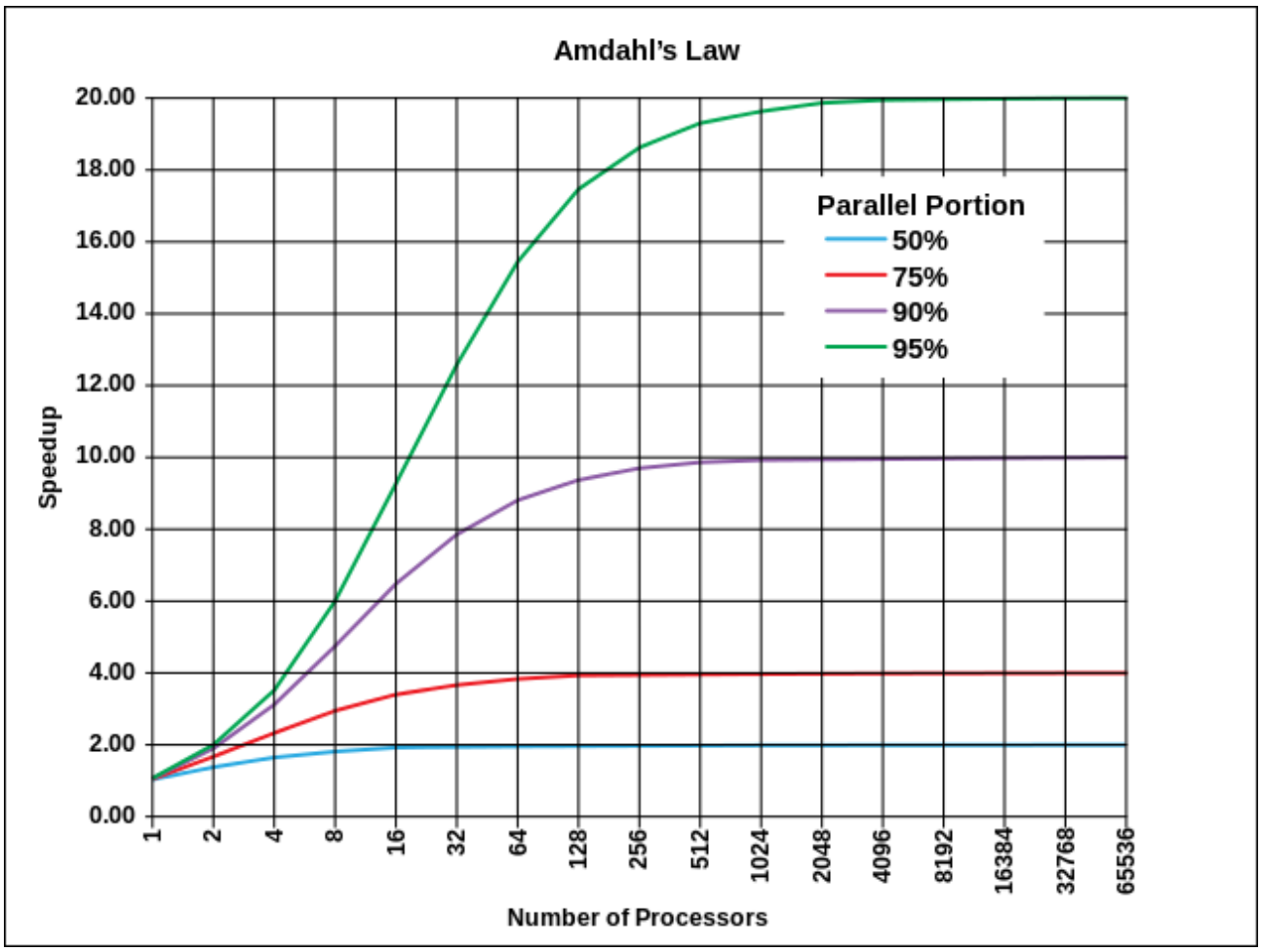


Figure 1-3: SVG Graph illustrating Amdahl's law

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors.

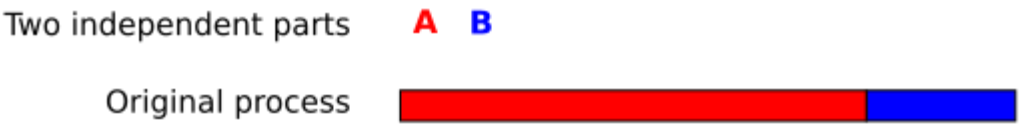
1.2.2. Examples

For example, if a program needs 20 hours using a single processor core, and a particular part of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours ($p = 0.95$) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence, the theoretical speedup is limited to at most 20 times ($(1 - p)^{-1} = 20$).

For this reason, parallel computing is relevant only for a low number of processors and very parallelizable programs.

1.2.3. Speedup in a serial program [16]

Assume that a task has two independent parts, *A* and *B*.



Part *B* takes roughly 25% of the time of the whole computation. By working very hard, one may be able to make this part 5 times faster, but this only reduces the time for the whole computation by a little.

In contrast, one may need to perform less work to make part *A* be twice as fast. This will make the computation much faster than by optimizing part *B*, even though part *B*'s speedup is greater by ratio, (5 times versus 2 times).

For example, with a serial program in two parts A and B for which T_A = 3s and T_B = 1s:

- if part B is made to run 5 times faster, that is s = 5 and p = $T_B/(T_A + T_B)$ = 0.25, then S=1.25;
- if part *A* is made to run 2 times faster, that is s = 2 and p = $T_A/(T_A + T_B)$ = 0.75, then S=1.60;

Therefore, making part *A* to run 2 times faster is better than making part *B* to run 5 times faster.

The percentage improvement in speed can be calculated as:

$$percentage\ improvement = 100\left(1 - \frac{1}{s}\right) \qquad (1\text{-}3)$$

Improving part A by a factor of 2 will increase overall program speed by a factor of 1.60, which makes it 37.5% faster than the original computation.

However, improving part B by a factor of 5, which presumably requires more effort, will only achieve an overall speedup factor of 1.25, which makes it 20% faster.

1.3. GUSTAFSON'S LAW [17]

Gustafson's law (or Gustafson–Barsis's law) gives the theoretical speedup in latency of the execution of a task at fixed execution time that can be expected of a system whose resources are improved. It is named after computer scientist John L. Gustafson and his colleague Edwin H. Barsis, and was presented in the article Reevaluating Amdahl's Law in 1988.

Gustafson's law can be formulated the following way:

$$S = 1 - p + sp \tag{1-4}$$

Where:

- S – theoretical speedup of the whole task;
- s – speedup in the latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- p – percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

Another approach:

If N is the number of processors, s is the amount of time spent (by a serial processor) on serial parts of a program and p is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by

$$S = \frac{s+p}{s+\frac{p}{N}} = \frac{1}{s+\frac{p}{N}} \tag{1-5}$$

where we have set total time s + p = 1 for algebraic simplicity. For N = 1024, this is an unforgivingly steep function of s near s = 0 (see Figure 1-4).

The steepness of the graph near s = 0 (approximately - N) implies that very few problems will experience even a 100-fold speedup.

Yet for three very practical applications (s = 0.4 - 0.8 percent) used, we have achieved the speedup factors on a 1024-processor hypercube

which we believe are unprecedented [18]: *1021* for beam stress analysis using conjugate gradients, *1020* for baffled surface wave simulation using explicit finite differences, and *1016* for unstable fluid flow using flux-corrected transport. How can this be, when Amdahl's argument would predict otherwise?

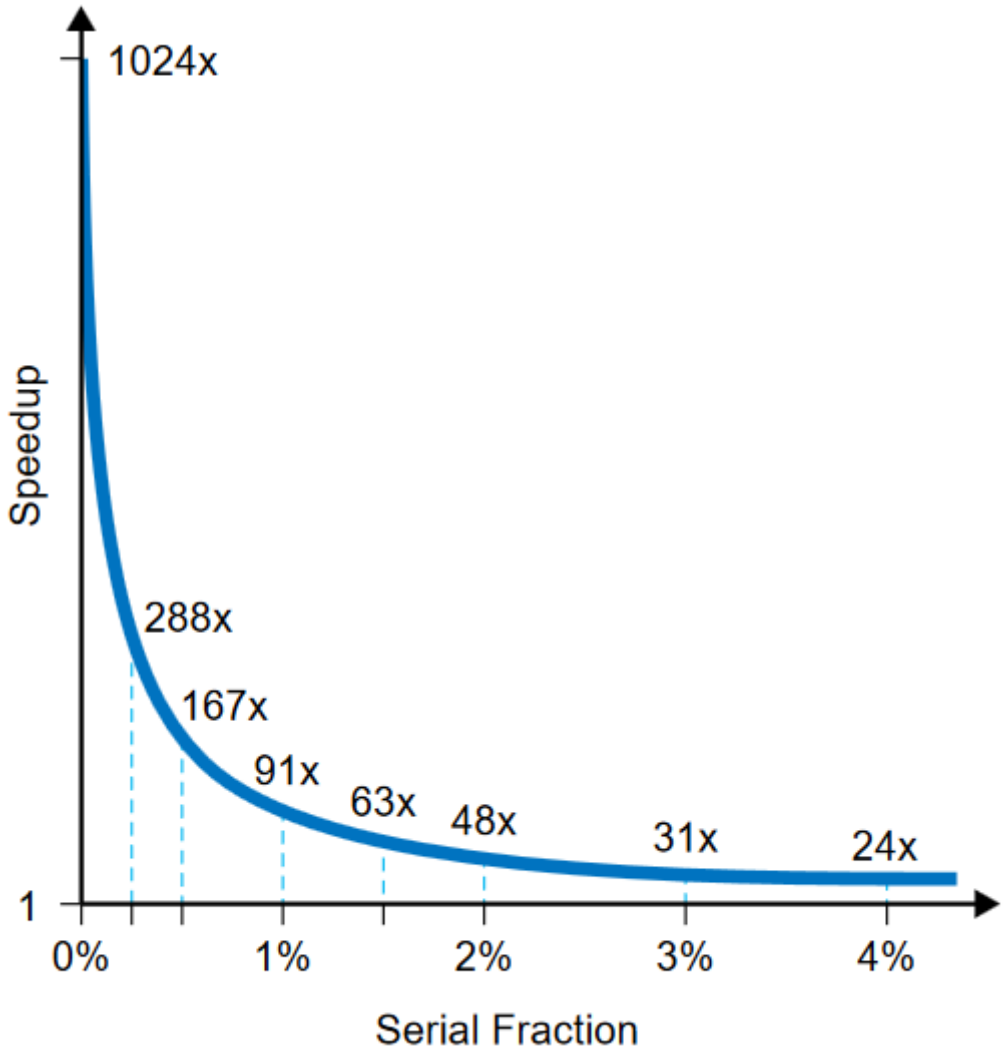


Figure 1-4: Speedup under Amdahl's Law [17]

The expression and graph both contain the implicit assumption that p is independent of *N, which is virtually never the case*. One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors*. When given a more powerful processor, the problem generally expands to make use of the increased facilities. Users have control over such things as grid resolution, number of time steps, difference operator complexity, and other parameters that are usually adjusted to allow the program to be

run in some desired amount of time. Hence, it may be most realistic to assume that *run time, not problem size, is constant*.

As a first approximation, we have found that it is the parallel or vector part of a program that scales with the problem size. Times for vector start-up, program loading, serial bottlenecks and I/O that make up the s component of the run do not grow with problem size. When we double the number of degrees of freedom in a physical simulation, we double the number of processors. But this means that, as a first approximation, the amount of work that can be done in parallel varies linearly with the number of processors. For the three applications mentioned above, we found that the parallel portion scaled by factors of 1023.9969, 1023.9965, and 1023.9965. If we use s' and p' to represent serial and parallel time spent on the parallel system, then a serial processor would require time s' + p' x N to perform the task. This reasoning gives an alternative to Amdahl's law suggested by E. Barsis at Sandia:

$$Scaled\ speedup = (s' + p'N) / (s' + p')$$

$$= s' + p'N$$

$$= N + (1 - N)s'$$

In contrast with Figure 1-4: Speedup under Amdahl's Law , this function is simply a line, and one with much more moderate slope: $1 - N$. It is thus much easier to achieve efficient parallel performance than is implied by Amdahl's paradigm. The two approaches, fixed-sized and scaled-sized, are contrasted and summarized in Figure 1-5 and Figure 1-6.

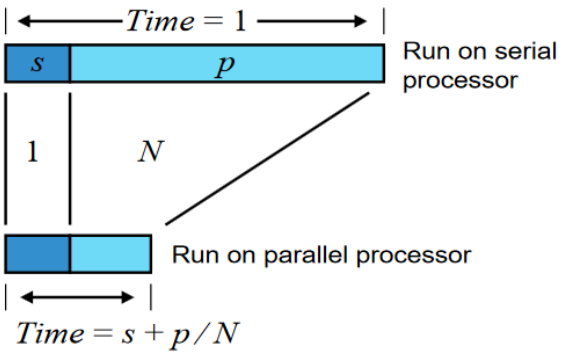


Figure 1-5: Fixed size model (Speedup = 1/(s + p/N))

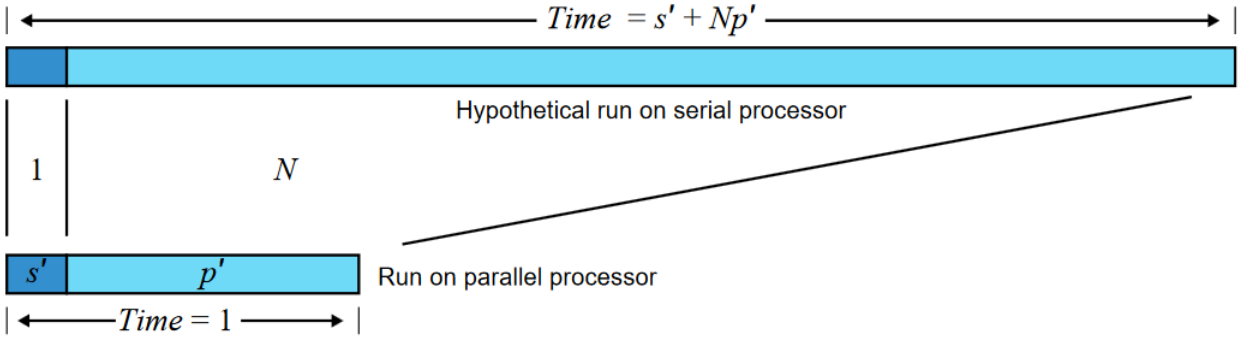


Figure 1-6: Scaled-Size Model (Speedup = s + Np)

The work to date shows that it is not an insurmountable task to extract very high efficiency from a massively-parallel ensemble, for the reasons presented here. We feel that it is important for the computing research community to overcome the "mental block" against massive parallelism imposed by a misuse of Amdahl's speedup formula; speedup should be measured by scaling the problem to the number of processors, not fixing problem size. We expect to extend our success to a broader range of applications and even larger values for N.

Gustafson's law addresses the *shortcomings of Amdahl's law*, which is based on the assumption of a fixed problem size, that is of an execution workload that does not change with respect to the improvement of the resources. Gustafson's law instead proposes that programmers tend to set the size of problems to fully exploit the computing power that becomes available as the resources improve. Therefore, if faster equipment is available, larger problems can be solved within the same time.

The impact of Gustafson's law was to shift research goals to select or reformulate problems so that solving a larger problem in the same amount of time would be possible. In a way the law redefines efficiency, due to the possibility that limitations imposed by the sequential part of a program may be countered by increasing the total amount of computation.

1.4. Amdahl's Law or Gustafson's Law [19]

There are two distinct pillars of Gustafson's law. I will describe them both in my own words.

1.4.1. Lemma 1

"*There are workloads that are gaseous in nature: when provided with more compute power, they expand to consume the newly provided power.*"

Such programs are more common than you think. I will give four real-life examples:

1. Bitcoin mining. If I give you more compute power, you will not finish sooner. Instead, you will just mine more coins.

2. Graphics. If I give you more compute power, you will just run your frames at a higher resolution or with more details.

3. Numerical analysis such as computing pi. If I give you more computes, you will just compute more digits of pi.

4. Weather Prediction. If I give you more computes, you will just run your software longer to get even more accurate predictions.

Side note: This property is also found in several non-parallel programs.

1.4.2. Lemma 2

"*When the problem size is increased, the parallel portion expands faster than the serial portion.*"

For example, Matrix-Matrix-Multiply (MMM). The setup of MMM, i.e. initializing the matrices increases linearly with the size of the matrix. however, the actual compute is $O(n^3)$.

1.4.3. What's does it imply

Gustafson's law only applies for workloads where both the above conditions are true. If a workload follows Gustafson's law, it is hurt.

1.4.4. What does it not imply

It does NOT imply that Amdahl's law is dead. It just implies Amdahl's law is less important in some workloads. I stress on the word less because the serial portion still exists and we know that it still hurts performance, but just with a lesser magnitude.

There are many workloads out there which aren't gaseous. For example, when sorting a list of numbers in Excel, I will not increase the size of my balance sheet if my computer gets faster. Similarly, when doing spell check, I will not write longer documents if my computer has become faster. Lastly, I will not make my database transactions lengthier if I can run them faster.

1.4.5. Shortcoming of Gustafson's Law

Just like Amdahl's law, Gustafson's law makes all the same assumptions about the world being infinitely parallel or completely serial. It also does not account for overhead associated with the creation/deletion of threads. It does not account for other type of serial portions such as critical sections. See this post for my list of assumptions. Thus, Gustafson's law only becomes applicable if: (a) the workload obeys all the assumptions of Amdahl's law, and (b) it obeys the above two Lemma's.

1.4.6. What can programmers do?

Still *try to eliminate the serial part*. If you can't eliminate the serial part, at least try to make it so that the serial part grows slower than the parallel part when the working set increases. Also try to make the time in serial part constant and independent of the number of threads (I realize this impossible in most cases).

1.4.7. What can architects do?

Keep Gustafson's law in mind and ensure that workloads that do obey Gustafson's law do not get penalized. This implies ensuring that a workload that does expand to leverage more cores does not become

limited due to the memory system. (I am just asking for a balanced design).

1.5. When Amdahl's law is inapplicable? [20]

A lot of industry and academic folks use the term Amdahl's law without understanding what it really means. Today I will discuss what Gene Amdahl said in 1967, what has become of it, and how it is often misused.

1.5.1. Gene Amdahl's insight

Amdahl's law was derived from Gene Amdahl's 1967 paper in AFIPS computer conference. On a side note, I am fascinated by the opening sentence of this paper:

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution.

Deja Vu — apparently, the death of single thread performance is not new.

This following sentence from this paper is the basis of the infamous Amdahl's law:

… the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

It is important to note *that the paper did not have an equation.* Neither did it talk about what types of serial bottlenecks exist, what parallel programming paradigm is he referring to, etc.

1.5.2. The equation

Somewhere along the road, we decided to convert Amdahl's insight into a law and characterize it using an equation. I could not find its origin but the equation goes as follows:

$$T_N = \alpha + \frac{1-\alpha}{N} \tag{1-6}$$

Where:

- N – number of cores
- T_N – time consumed with N cores
- α – fraction of instructions in serial code

1.5.3. Simplifying assumptions

This equation makes seven simplifying assumptions:

1. *The world is black and white*: the number of executing threads is either equal to 1 or N; there is nothing in between. This is often false, e.g., in Google Map Reduce, the number of threads in the Map phase may be N but the number of threads in the Reduce phase are often smaller than N but greater than 1.

2. The *parallel portion has perfect speedup*. This is not true because contention for shared data (critical sections) and shared resources (caches, memory bandwidth) often prohibit the program from reaching perfect linear speedup.

3. The *parallel portion has infinite scaling*, i.e., performance never saturates. This is incorrect because contention for shared data and shared resources increases with the number of threads. This contention can reach a point that adding more threads does not increase performance (or reduces performance).

4. There is *no thread creation/deletion overhead*. Allocating and de-allocating threads is expensive and this overhead increases linearly with the number of threads.

5. The *length of the serial, single-threaded portion is independent of the number of threads*. The single thread work often consists of splitting the work for the parallel portion. This work is often a function of how many threads will be spawned during the parallel work. Furthermore, more threads can lead to more inter-core communication, thereby extending the length of the serial portion.

6. *Serial and parallel code runs at the same rate*.

7. The *serial portion cannot be overlapped with the parallel threads*. Many workloads embed serial portions inside parallel portions in order to overlap their execution with parallel work, e.g., exploiting pipeline parallelism between serial and parallel portions.

The main distinction that reduces the scope of Amdahl's law is #7.

1.5.4. Who can use Amdahl's law?

Amdahl's law is only applicable in certain fork-join programming paradigms. Specifically, it is applicable to workloads where some code runs as a single thread followed by some embarrassingly parallel code, e.g., matrix-matrix-multiply or other HPC kernels.

1.5.5. Who should avoid using the term "Amdahl's law"?

As with any analytic model, Amdahl's law should only be used when a workload fits the programming model assumed by the model. There are many programs that do not fit this model. Fundamentally, Amdahl's law assumes that any code which cannot be parallelized is always on the critical program path. This is not the case in many modern programming paradigms as some non-parallel code sections can run in parallel with other independent code. For example:

1. *Critical sections*: Critical Sections are portions of code where only one thread can execute at a given time; other threads wanting to

execute the critical section must wait. These critical sections can serialize a variable number of threads depending on the contention for the critical section. This contention is sometimes zero and sometimes very high. Thus, Amdahl's law does not apply to critical section intensive workloads such as databases.

2. *Serial stages in a pipeline/task parallel workload*: The serial stage only leads to serialization if it is the critical path of the program. It may have zero or more threads waiting for it at any given time. For example, all graphics workloads have a thread which is producing work for the other threads. This serial thread does not become a bottleneck unless it cannot feed the other threads fast enough. However, if it becomes the bottleneck, then the parallel portion seizes to matter. Thus, the Amdahl's equation does not characterize the behavior of these kernels.

3. *Regions of limited parallelism*: These exist in programs due to contention for hardware resources or thread creation/deletion overhead. The Amdahl's equation does not apply to these as it assumes that the serial part is only one thread. For example, Google Map Reduce does not follow the Amdahl's model.

The above constructs change the first-order analytic models of the program completely and do not fit Amdahl's. Yet, we often use serial portions and Amdahl's law synonymously. I must admit that it is one of my pet peeves and I believe that we need to fix this problem as a community. It is best to use more specific terms for different types of serial code in order to avoid misunderstanding.

1.5.6. Concrete Example

Let's think about pipeline workloads. Let's use a simple example of a workload called Rank. Rank compares a set of input strings in a file to a given search string and returns the N most similar strings to the search string.

There are three stages in a particular implementation of Rank:

S1. Read input string — this is sequential

S2. Compute similarity score by comparing to search string — multiple strings can be compared concurrently

S3. Insert string in a heap sorted according to the similarity score. Drop the element with the lowest similarity score in case the heap has N+1 elements after the insertion. — this is a critical section in a naive implementation.

Now in the above code, there are three distinct loops that can be done one after the other in which case #1 and #3 will be your Amdahl's serial portions (where only a single thread exists). However, I can be smart and write this code as a pipeline where S1, S2, and S3 run concurrently and communicate via work-queues.

Let's suppose each iteration of S1 takes 1K cycles, S2 takes 10K cycles, and S3 takes 2K cycles. Then as I up the number of cores, eventually the throughput of this pipeline will become limited by S3 because even if I speed up S2 to 1000 cycles per iterations (by giving it 10 cores), S3 will not speed up. Thus, once I have say 5 cores assigned to S2, more cores will not help with performance.

Now, naively (and incorrectly), people call this Amdahl's law. No it is not Amdahl's law because the above cannot be characterized by the Amdahl's equation. If we use Amdahl's, the serial code is 3K cycles and parallel core is 10K cycles. Thus, with infinite cores:

Cycles_per_iteration_with_P_cores = 3K + 10K/P

Thus, the fastest speed will be 3K cycles per iteration. This is obviously wrong.

The equation which characterizes this pipeline case is as follows:

Cycles_per_iteration = MAX (Cycles_per_S1, Cycle_per_S2 / cores_assigned_to_S2, Cycle_per_S3);

This is my poster child example of showing why Amdahl's doesn't work for all non-parallel code.

1.6. TYPES OF SERIAL BOTTLENECKS [21]

In my definition, a serial bottleneck is code which can lead to thread serialization, i.e., it can cause threads to wait on each other. At a broader level, there are two types of serial bottlenecks:

Fully serial, always on Critical Path: These are code portions where only a single thread exists. These bottlenecks cause other threads to wait every time they execute. These include kernels which cannot be parallelized at all.

These single threaded regions are the classic Amdahl's bottlenecks. They have an important property that they always end up on the critical program path, i.e., if you take a single-threaded region and speed it up by a 100 cycles, the overall program execution time reduces by a 100 cycles. In addition to that, they have the following attributes.

- Easier to detect (program is in a fully serial portion if number of alive threads == 1)
- Do not impact the performance of parallel program portions (since they do not run concurrently with the parallel portion)
- Are less common since most people realize their disadvantages
- Shortening them always provides performance benefit (may be small, but its >0)

Partially serial: These include serial portions which are embedded in the parallel portions. I call them partial bottlenecks because (a) they block a variable number of threads (between 0 to N-1), and (b) they can be on or off the critical path thus shortening them may or may not benefit overall performance. These bottlenecks generally arise when threads communicate or contend for shared resources or shared data. The classic example of a partial bottleneck is a critical section. Only one thread can execute a critical section at a given time, all other threads wanting to execute the critical section must wait. However, threads not wanting to execute the critical section can continue to work. Thus, the impact on performance of the critical section depends on the number of waiting threads which in turn depends on the contention for the critical section. A highly contended critical section

stalls a lot of threads while a critical section with no contention is practically the same as parallel code.

The following distinguishes partial-bottlenecks from fully-serial bottlenecks:

- The existence and severity of these bottlenecks is highly dependent on the input set, machine configuration, the number of cores, communication latencies, cache sizes, etc.
- These bottlenecks are much harder to identify for the programmer since they only surface at run-time
- They impact only the parallel program portions
- Are very common in servers and sometimes in HPC kernels
- Shortening them is only beneficial if they are causing serialization
- They limit thread scalability, i.e., they can cause the performance to peak at a given number of threads such that more threads reduce performance

1.6.1. Some background

I feel that fully-serial bottlenecks are discussed more because of historical reasons. Classic parallel computers consist of loosely connected processors which require hundreds or thousands of cycles to communicate (sometimes via Ethernet). When writing code for such a machine, it is logical for the programmers to eliminate thread communication from the parallel portions and leave as fully-serial the kernels where it is impossible to remove (or minimize) thread communication. This trade-off has changed with the advent of multi-core: cores are tightly integrated and thread communication is a smaller overhead. Thus, we can expect more programs to contain partially-serial code with thread communication.

1.6.2. Who should this concern?

If you are a theoretical computer scientist, you should know that Amdahl's law (the equation that is known as the law) only applies to the fully serial portions and not to the partially serial portions. The partial bottlenecks have a very different behavior which has different

equations (See examples of critical sections, pipeline workloads, and task parallelism).

If you are designing hardware for running parallel programs, understanding the differences between these bottlenecks is pivotal. I will give an example from my personal experience. When I was first asked to architect a heterogeneous code chip for parallel programs, I considered only the fully-serial bottlenecks and hence the first architecture I designed had a single fast core and many slow cores. I wrote an OS scheduler which turned on the fast core only in the serial phase and turned on the many small cores only in the parallel region. It worked and I was done! However, as I learned more about parallel programs, and learned about these fine-grain bottlenecks (which are often a bigger issue), the landscape changed completely. I needed enough on-chip power to keep both slow and fast cores on at the same time. I had to design mechanisms to detect these fine-grain serial portions. I had to take thread migration overheads into account. I had to decide how small my small cores can be depending on how scalable my parallel portion is. Thus, I urge hardware designers to understand this distinction as designing multicores without knowing about these finer-grain bottlenecks can lead to very sub-optimal decisions. (Side note for developers: you will be surprised at how few chip architects know this).

If you do performance analysis, you should know that while you can characterize the fully-serial bottlenecks easily, you cannot expect that staring at the code or counting instructions will tell you much about the partially parallel portions since their severity is a function of contention and very dependent on run-time behavior. (Side note for programmers: this makes life very hard for us computer architects because deterministic performance simulations become next to impossible).

If you are an application programmer, you should already know these bottlenecks and everything about them. If not, you learn asap because you will find it useful.

1.7. On Using Dependence Information [22]

Writing parallel code is all about finding parallelism in an algorithm. What limits parallelism are the dependencies among different code portions. Understanding the dependencies in a program early on can help the programmers (a) determine the amount of available parallelism, and (b) chose the best parallel programming paradigm for the program. In this post, I try to layout a taxonomy of dependencies in loops and how it plays into parallel programming.

1.7.1. Definition of tasks

It is any piece of work that needs to be done and often takes more than one instructions. The instructions in the task are closely coupled such that it logically makes sense to group them together.

1.7.2. What are task dependencies?

There are two types of dependencies at the task granularity.

Data Dependency: A task K is said to be data-dependent on task J if K needs data generated by J. For example,

```
J: foo = bar + 3;
K: lama = foo + 3;
```

Note that this dependency is ordered: the dependent task J can neither run before K nor in parallel with K.

***Un-ordered Dependency*:** A task J and K are said to have an unordered dependency if they both read-modify-write the same data. For example,

```
J: foo++;
K: foo++;
```

Note that J and K can be processed in any order but they cannot be processed in parallel.

Thus, two tasks can run concurrently only if they are independent. The goal in parallel programming is to remove as

many dependencies as possible by re-factoring the code or the algorithm.

1.7.3. How does it apply to loops?

Let's use the following loop as an example:

```
for i = 1 to N:
A(i); B(i); C(i);
```

A single core will run the loop in this order: A0, B0, C0, A1, B1, C1, A2, … (A0 = task A in iteration 0, B0 = task B in iteration 0, etc.)

Loops have two types of dependencies:

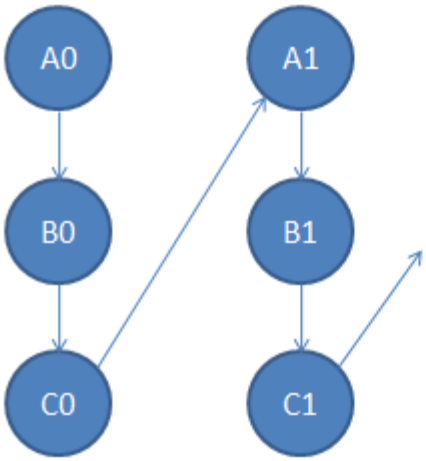
Intra-Iteration: Typically, tasks within an iteration of the loop are dependent on each other for data. For example, B0 depends on A0.

Inter-iteration: Sometimes different loop iterations share data or hardware resources which creates data or ordered dependencies among them. For example, A1 depends on A0.

1.7.4. How it applies to parallel programming?

I have seen four types of common loops.

1. No parallelism exists in a loop where all tasks are data-dependent on the previous tasks. Thus, programmers should re factor the code to remove those dependencies before writing any parallel code. If no dependencies can be removed, then the loop should be left untouched. For example,



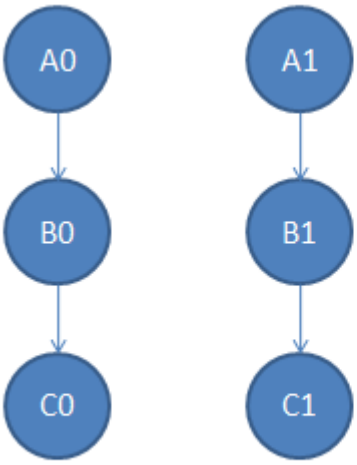

```
for I = 1 to N:
   X[i] = (X[i-1] + X[i])/2;
```

No parallelism exists in this loop

K is data-dependent on L

K has unordered dependency on L

2. Loops with independent iterations are easy to parallelize. In this case, programmers should consider using SIMD and watch out for false-sharing and off-chip bandwidth. For example,

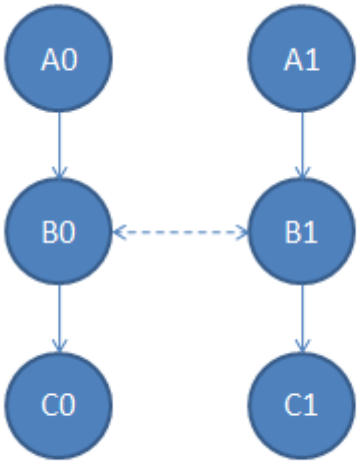



```
for I = 1 to N:
   X[i] = Y[i];
```

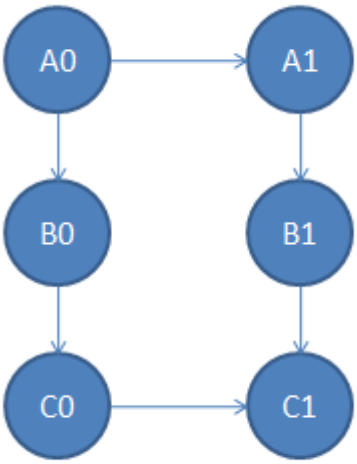
Loop with independent iterations can be parallelized by running iterations in parallel

3. In a typical loop, some tasks are independent of other iterations while other tasks have inter-iteration dependencies. In my experience, such dependencies are usually un-ordered which can be *enforced using*

critical sections. For example, task B should be put inside a critical section in the following loop.



4. If you cannot remove ordered dependencies, you should use a lesson from hardware designers, i.e., use pipeline parallelism. For example, the following loop is well-suited for pipeline parallelism where one thread can process all instances of "A" in-order, one thread can be in-charge of processing "B"s, and the last thread can be in-charge of processing "C"s. *Note that if B is substantially longer than A or C, it is possible to use multiple threads for the "B-stage" but that will make things out of order and C will need a re-ordering structure to put them back in order.*




```
for I = 1 to N:
   Read block from file
   Compress block
   Write block to file (in-order)
```

And there are many other types of loops out there.

1.8. Understanding Critical Sections impact [23]

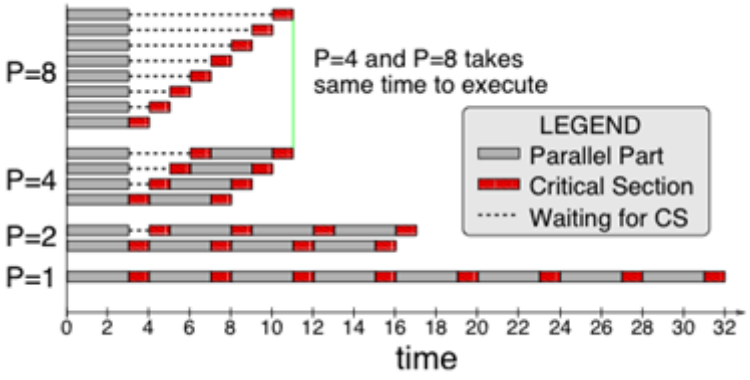
In shared memory systems, multiple threads are not allowed to update shared data concurrently, known as the mutual exclusion principle. Instead, accesses to shared data are encapsulated in regions of code guarded by synchronization primitives (e.g. locks). Such guarded regions of code are called critical sections. The semantics of a critical section dictate that only one thread can execute it at a given time. Any other thread that requires access to shared data must wait for the current thread to complete the critical section.

There are two types of critical sections in programs. I call them *update* critical sections and *reduction* critical sections.

1.8.1. Update Critical Sections

Update critical sections occur in the midst of the parallel kernels. They protect shared data which multiple threads try to read-modify-write *during* the kernel's execution, instead of waiting till the end of the kernel's execution. Their execution can be overlapped with the execution of non-critical-section code.

For simplicity, let's assume a kernel which has only one critical section. Each iteration of the loop spends one unit of time inside the critical section and three units of time outside the critical section. The following chart demonstrates the execution timeline of this critical section intensive application.



When a single thread executes, only 25% of execution time is spent executing the critical section. If the same loop is split across two threads, the execution time reduces by 2x. Similarly, increasing the

number of threads to four further reduces execution time. As the critical section is always busy, the system becomes critical section limited and further increasing the number of threads from four to eight does not reduce the execution time.

We can capture this using a simple equation. Suppose

Tcs =	Time inside critical section
Tnocs =	Time outside critical section
Tp =	Time with p cores
N =	Number of iterations

Then Tp can be computed as:

$$T_p = N \times MAX\left(\frac{T_{NoCS} + T_{CS}}{P}, T_{CS}\right)$$

We compute Pcs, i.e., the number of threads required to saturate the execution time, by solving the above equation for P:

$$P_{CS} \geq \frac{T_{NoCS} + T_{CS}}{T_{CS}}$$

1.8.2. Fine-grain critical sections

To reduce the contention for critical sections, many applications use different locks to protect disjoint data. Since these critical sections are protecting disjoint data, they can execute concurrently, thereby increasing throughput. In this software, the longest critical section, which has the highest contention, is the performance limiter. This can be captured using the following equation:

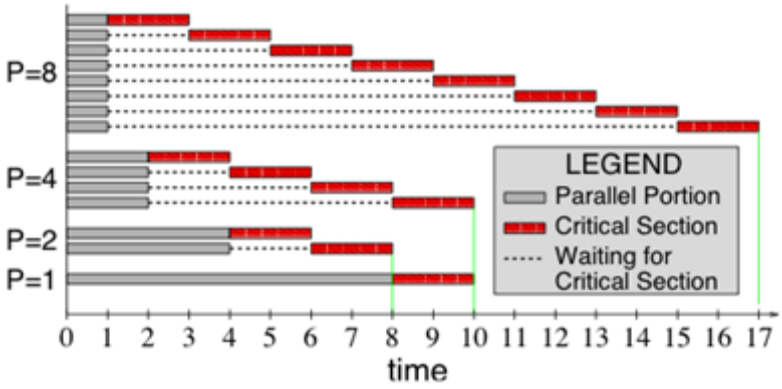
$$T_p = N \times MAX\left(\frac{T_{NoCS} + T_{CS_{all}}}{p}, T_{CS_{longest}}\right)$$

Note that this model is rather simplistic but it still conveys that long, frequently occurring critical sections can limit performance as the number of threads increases.

1.8.3. Reduction Critical Sections

Reduction critical sections occur at the end of kernels and are used to combine the intermediate results computed by individual threads. The key difference between update and reduction critical sections is that, unlike update critical sections, reduction critical sections occur at the *end* of a kernel and their execution cannot be overlapped with the execution of the non-critical-section code. Since every thread executes the critical section, the total time spent in executing the critical sections increases linearly with the number of threads. Furthermore, as the number of threads increase, the fraction of execution time spent in the parallelized portion of the code reduces. Thus, as the number of threads increase, the total time spent in the critical sections increases and the total time spent outside critical sections decreases. Consequently, critical sections begin to dominate the execution time and the overall execution time starts to increase.

For simplicity, let us assume that a kernel executes for 10 times units as a single thread and 2 of those 10 units are spent in reduction. The following figure shows its execution as the number of threads increases.



Notice how the parallel region shrinks with more threads but the critical sections begin to dominate the execution. We can capture this using a simple analytic model:

$$T_P = \frac{T_{NoCS}}{P} + P \cdot T_{CS}$$

We compute Pcs, i.e., the number of threads required to saturate the execution time, by solving the above equation for P:

$$\frac{d}{dP}T_P = -\frac{T_{NoCS}}{P^2} + T_{CS}$$

$$P_{CS} = \sqrt{\frac{T_{NoCS}}{T_{CS}}}$$

1.8.4. Cache Locality: A second order effect

The above analysis assumes that the execution time of each instance of the critical section is independent of the number of cores. This is not a true assumption. In fact, latency of the critical section increases with the number of cores for two reasons. First, critical sections must incur cache misses in fetching the shared data that is resident at another core. With more cores, shared data bounces around more frequently thereby increasing the probability of a cache miss. Second, the cache miss latency among cores increases with the number of cores due to longer wires and more interconnect hops. This increase in the number of misses and the cost of each miss further increases the overhead of critical sections on performance.

When there is contention for shared data, execution of threads gets serialized, which reduces performance. As the number of threads increases, the contention for critical sections also increases. Therefore, in applications that have significant data synchronization (e.g. Mozilla Firefox, MySQL database, and operating system kernels), critical sections limit both performance (at a given number of threads) and scalability.

1.8.5. Looking forward …

It is important to either shorten the critical sections or create fine-grain critical sections that do not suffer high contention. Parallel programming is already a daunting task and expecting programmers to shrink or eliminate critical sections is unreasonable. I believe that hardware can assist in this matter. Solutions like hardware

Transactional memory (TM) have been proposed to shorten the critical sections by letting them run concurrently, as long as it does not violate correctness. Another orthogonal solution is proposed in a paper titled “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures” [24] is to accelerate critical sections using a faster core on a chip with heterogeneous cores. A combination of TM and ACS can practically eliminate the overhead of critical sections, thereby relieving the programmer of this burden.

2. SOFTWARE, HARDWARE AND COMPETITION

2.1. GROMACS

GROMACS is an engine to perform molecular dynamics simulations and energy minimization. These are two of the many techniques that belong to the realm of computational chemistry and molecular modelling. Computational chemistry is just a name to indicate the use of computational techniques in chemistry, ranging from quantum mechanics of molecules to dynamics of large complex molecular aggregates. Molecular modelling indicates the general process of describing complex chemical systems in terms of a realistic atomic model, with the goal being to understand and predict macroscopic properties based on detailed knowledge on an atomic scale. Often, molecular modelling is used to design new materials, for which the accurate prediction of physical properties of realistic systems is required. [25]

Macroscopic physical properties can be distinguished by (a) static equilibrium properties, such as the binding constant of an inhibitor to an enzyme, the average potential energy of a system, or the radial distribution function of a liquid, and (b) dynamic or non-equilibrium properties, such as the viscosity of a liquid, diffusion processes in membranes, the dynamics of phase changes, reaction kinetics, or the dynamics of defects in crystals. The choice of technique depends on the question asked and on the feasibility of the method to yield reliable results at the present state of the art. Ideally, the (relativistic) time-dependent Schrodinger equation describes the properties of ¨molecular systems with high accuracy, but anything more complex than the equilibrium state of a few atoms cannot be handled at this ab initio level. Thus, approximations are necessary; the higher the complexity of a system and the longer the time span of the processes of interest is, the more severe the required approximations are. At a certain point (reached very much earlier than one would wish), the ab initio approach must be augmented or replaced by empirical parameterization of the model used. Where simulations based on physical principles of atomic interactions still fail due to the complexity of the system, molecular modelling is based entirely on a similarity analysis of known structural

and chemical data. The QSAR methods (Quantitative Structure Activity Relations) and many homology-based protein structure predictions belong to the latter category.

Macroscopic properties are always ensemble averages over a representative statistical ensemble (either equilibrium or non-equilibrium) of molecular systems. For molecular modelling, this has two important consequences:

• The knowledge of a single structure, even if it is the structure of the global energy minimum, is not sufficient. It is necessary to generate a representative ensemble at a given temperature, in order to compute macroscopic properties. But this is not enough to compute thermodynamic equilibrium properties that are based on free energies, such as phase equilibria, binding constants, solubility, relative stability of molecular conformations, etc. The computation of free energies and thermodynamic potentials requires special extensions of molecular simulation techniques.

• While molecular simulations, in principle, provide atomic details of the structures and motions, such details are often not relevant for the macroscopic properties of interest. This opens the way to simplify the description of interactions and average over irrelevant details. The science of statistical mechanics provides the theoretical framework for such simplifications. There is a hierarchy of methods ranging from considering groups of atoms as one unit, describing motion in a reduced number of collective coordinates, averaging over solvent molecules with potentials of mean force combined with stochastic dynamics, to mesoscopic dynamics describing densities rather than atoms and fluxes as response to thermodynamic gradients rather than velocities or accelerations as response to forces.

For the generation of a representative equilibrium ensemble two methods are available: (a) Monte Carlo simulations and (b) Molecular Dynamics simulations. For the generation of non-equilibrium ensembles and for the analysis of dynamic events, only the second method is appropriate. While Monte Carlo simulations are more simple than MD (they do not require the computation of forces), they do not yield significantly better statistics than MD in a given amount of

computer time. Therefore, MD is the more universal technique. If a starting configuration is very far from equilibrium, the forces may be excessively large and the MD simulation may fail. In those cases, a robust energy minimization is required. Another reason to perform an energy minimization is the removal of all kinetic energy from the system: if several "snapshots" from dynamic simulations must be compared, energy minimization reduces the thermal noise in the structures and potential energies so that they can be compared better.

2.2. The global MD algorithm

1. Input initial conditions

Potential interaction V as a function of atom positions, positions $\boldsymbol{r}$ of all atoms in the system, velocities $\boldsymbol{v}$ of all atoms in the system

Repeat 2,3,4 for the required number of steps:

2. Compute forces

The force on any atom

$$\boldsymbol{F}_i = -\frac{\partial V}{\partial \boldsymbol{r}_i} \tag{2-1}$$

is computed by calculating the force between non-bonded atom pairs:

$$\boldsymbol{F}_i = \sum_j \boldsymbol{F}_{ij} \tag{2-2}$$

plus bonded interactions forces, restraining and external forces.

Potential, kinetic energies and the pressure tensor may be computed.

3. Update configuration

The movement of the atoms is simulated by numerically solving Newton's equations of motion:

$$\frac{d^2\boldsymbol{r}_i}{dt^2} = \frac{\boldsymbol{F}_i}{m_i} \tag{2-3}$$

$$\frac{d\boldsymbol{r}_i}{dt} = v_i; \frac{dv_i}{dt} = \frac{\boldsymbol{F}_i}{m_i} \tag{2-4}$$

4. Output step (if required)

write positions, velocities, energies, temperature, pressure, etc.

THE GLOBAL MD ALGORITHM

1. Input initial conditions

Potential interaction V as a function of atom positions
Positions $\boldsymbol{r}$ of all atoms in the system
Velocities $\boldsymbol{v}$ of all atoms in the system

⇓

repeat 2,3,4 for the required number of steps:

2. Compute forces

The force on any atom

$$\boldsymbol{F}_i = -\frac{\partial V}{\partial \boldsymbol{r}_i}$$

is computed by calculating the force between non-bonded atom pairs:

$$\boldsymbol{F}_i = \sum_j \boldsymbol{F}_{ij}$$

plus the forces due to bonded interactions (which may depend on 1, 2, 3, or 4 atoms), plus restraining and/or external forces.
The potential and kinetic energies and the pressure tensor may be computed.

⇓

3. Update configuration

The movement of the atoms is simulated by numerically solving Newton's equations of motion

$$\frac{\mathrm{d}^2\boldsymbol{r}_i}{\mathrm{d}t^2} = \frac{\boldsymbol{F}_i}{m_i}$$

or

$$\frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t} = \boldsymbol{v}_i; \quad \frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} = \frac{\boldsymbol{F}_i}{m_i}$$

⇓

4. if required: **Output step**

write positions, velocities, energies, temperature, pressure, etc.

Figure 2-1: GROMACS global MD algorithm [25]

2.3. MAP-REDUCE ARCHITECTURE

2.3.1. History

Map-Reduce architecture is a machine developed on the model made public by Kleene. [26] Map-Reduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

MRA gave birth to the Connex system in the fall of 2001, based on an older Connex memory concept, developed by Gheorghe STEFAN and Dan TOMESCU.

Connex architecture is a general-purpose SIMD (Single Instruction, Multiple Data) computing architecture.

This architecture is a solution to solve computationally intensive problems. Unlike existing general purpose processors that are based on Turing machine model, this architecture is trying a different approach. The general idea is built from computing model of Kleene. In the Turing published his work, published Kleene partially recursive functions model. He defined calculation using a set of functions (zero increment, projection) and rules (composition, primitive recursion and minimization).

In 2003 the first chip was manufactured using this architecture, CA4096, the company BrightScale (formerly Connex Technology, Inc.) manufactured in 130nm technology at TSMC (Taiwan Semiconductor Manufacturing Company) followed in 2007 BA1024 manufacture in 95nm technology and in 2008 BA1024B in 65nm technology.

2.3.2. Connex architecture

Connex architecture consists of four main elements:

- Connex array - from 64 to 4096 32-bit execution units (EU);
- I/O Plan - 2D shift register of 64 to 4096 32-bit words;
- Interconnection Fabric - standard 128-bit interface with external memory;

- Controller - CPU controlling system, it sees Connex array and I/O Plan web as coprocessors.

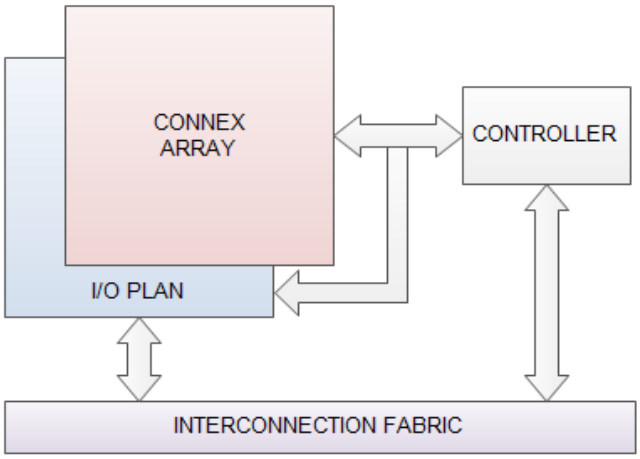


Figure 2-2: Connex general architecture

Connex architecture, exposed schematically in Figure 2-2, consists of multiple execution units (EU) (up to several thousand) selectively executing the same instruction in parallel. These cells are coordinated processing of a Central Unit of Execution (CONTROLLER) that can send instructions to the execution unit and can receive information from them through a reduction mode (NET REDUCTION) accessible to all cells. Selection of cells which process a particular statement is based on an individual's own cells bit determines if the cell is active or not.

Connex module array comprises the following:

- Execution unit, with 512 kB to 4096 kB of memory, receiving a 16-bit instruction each clock cycle;
- Reduction net - circuit that can do operations on all values of execution units (ex. the sum of all values).

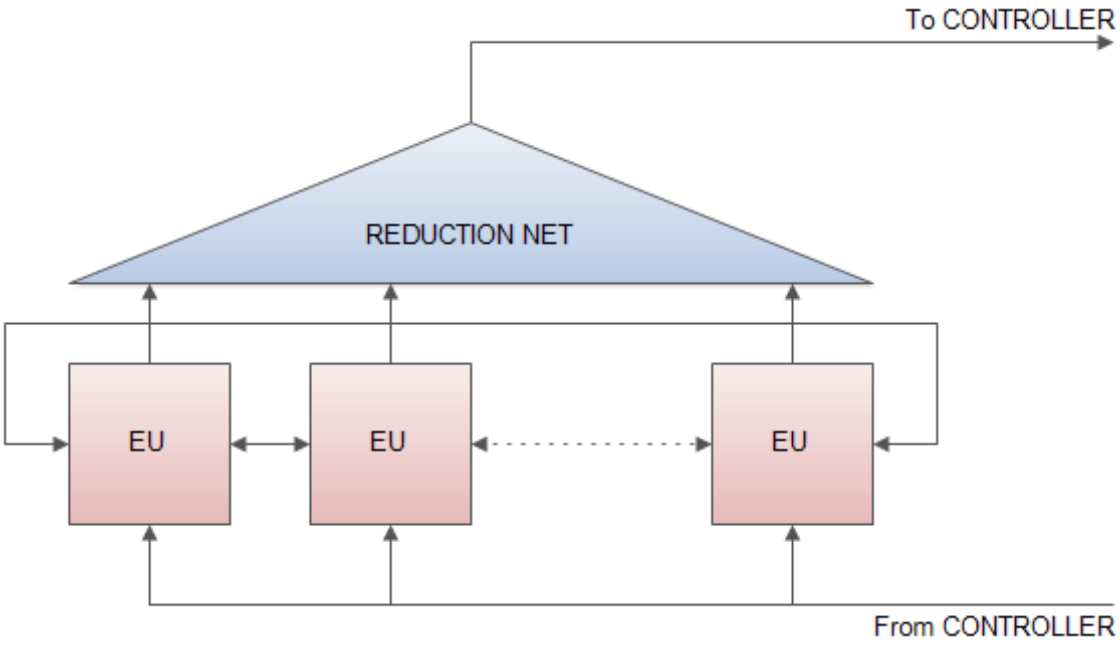


Figure 2-3: Connex array module

Access to external memory can be made directly by the Central Unit of Execution, or the row of cells via a System Input / Output controlled also by the Central Execution. Execution units are designated by the index of that cell. Further, the number of cells is P and these are numbered from 0 to P-1 from left to right as shown. These EU are interconnected through a left and right communication channel which can transmit information through a communication channel unidirectional linking with Module Input / Output and each access module Reducer that can submit information to the Central Unit of Execution.

The execution unit contains the following:

- Scalar unit - the arithmetic logic unit 16-bit special instructions to execute multi-cycle instructions and operations with fractional numbers;
- Boolean unit - the unit used to determine whether execution unit will be active or not;
- Data memory - 512-4096 KB data memory;
- Shift unit - manages connections Connex left-right module array and data exchange with I/O plane.

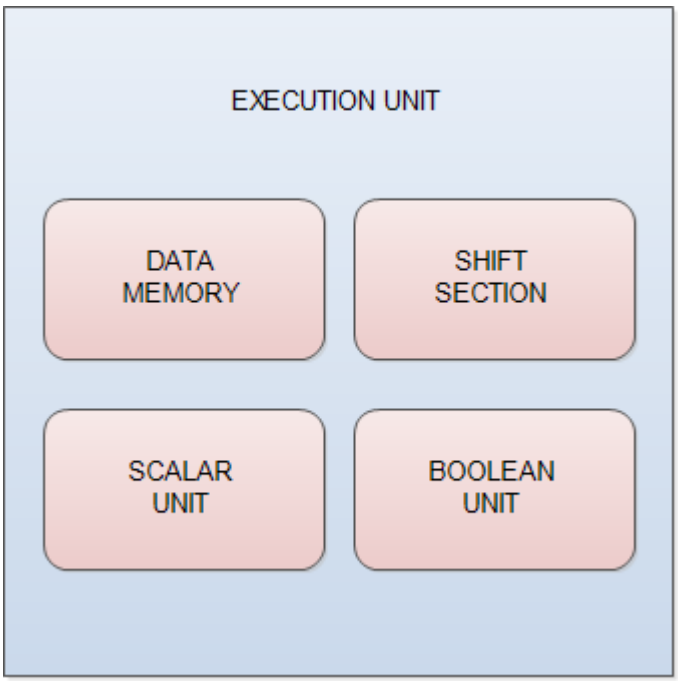


Figure 2-4: Connex array execution unit

Each unit has its own memory, knows its index and the boolean bit that determines whether it is active or not.

2.4. COMPETITION

Several other parallel computing platforms have been started, each bearing a higher or lower degree of flexibility (or specialization).

2.4.1. Anton

"*A Special-Purpose Machine That Achieves a Hundred-Fold Speedup in Bio-molecular Simulations*"

Molecular dynamics simulation has long been recognized as a potentially transformative tool for understanding the behaviour of proteins and other biological macromolecules, and for developing a new generation of precisely targeted drugs.

Many biologically important phenomena, however, occur over timescales that have previously fallen far outside the reach of MD technology.

Researchers have constructed a specialized, massively parallel machine, called Anton, that is capable of performing atomic-level simulations of proteins at a speed roughly two orders of magnitude

beyond that of the previous state of the art. The machine has now simulated the behaviour of a number of proteins for periods as long as a millisecond - approximately 100 times the length of the longest such simulation previously published - revealing aspects of protein dynamics that were previously inaccessible to both computational and experimental study. The speed at which Anton performs these simulations is in large part the result of a tightly coupled code sign process in which the machine architecture was developed in concert with novel algorithms, including an asymptotically optimal parallel algorithm with highly attractive constant factors for the range-limited N-body problem. [27]

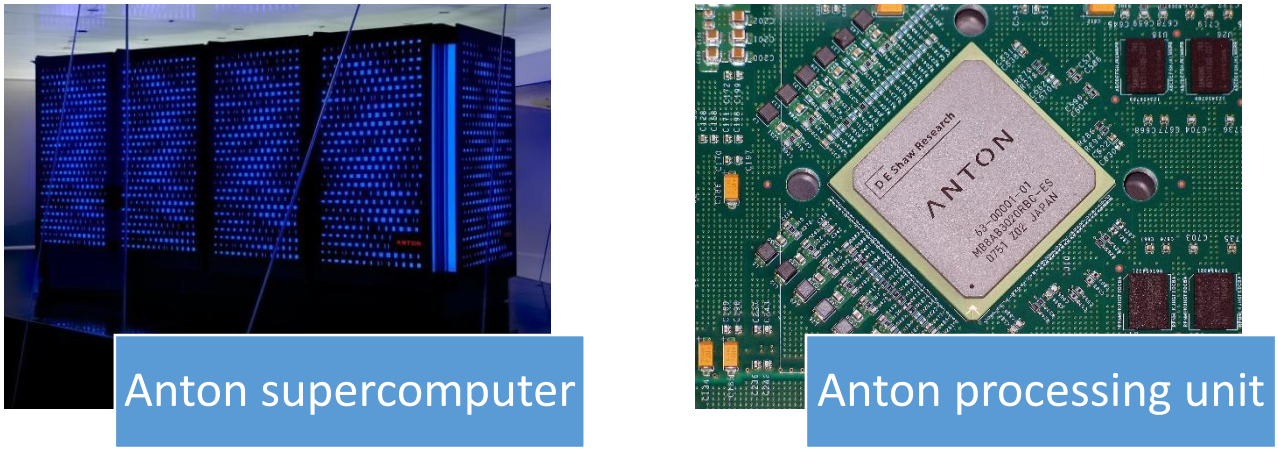


Figure 2-5 Anton supercomputer and insides

Anton is a special-purpose molecular dynamics machine, massively parallel, using custom-designed chips and designed together with a new algorithmic approach. It's dramatically faster for MD, but far less flexible for other purposes.

In 2012 there were 13 operational machines, each one capable of millisecond-scale simulations and much harder simulations than either very large number of very short simulations and/or very short simulations of very large molecules.

2.4.2. Comparison between Anton and other machines

Table 1 Comparison of MD simulation speeds (all-atom, explicit solvent, standard DHFR benchmark) [28]

Computational platform	Speed (ns/day)
Single-processor codes	~ 2
Parallel supercomputers	~ 200
Anton 1 (512-node machine)	17 400
Anton 2 (512-node machine)	85 000

Table 2 Largest published molecular dynamics simulations (all-atom simulations of proteins in explicit solvent)

Length (μs)	Hardware	Software	Protein
2	Single x86	GROMACS	villin HP-3
10	HPC cluster	NAMD	ww domain
1 119	Anton 1	[native]	ww domain
2 092	Anton 1	[native]	NTL9

2.4.3. Anton computation time for researchers

Anton was made available for use by researches, universities and other non-profit organisations / institutions without cost, at National Resource for Biomedical Supercomputing (NRBSC), where the funding for NRBSC's involvement was provided by NIH.

Time has been allocated to 45 research groups, all selected by National Academies.

3. IMPLEMENTATION

In this section, I'll explain how various MD algorithms have been implemented on the Map-Reduce accelerator.

3.1. SYSTEM VARIABLES

Every algorithm uses the following variables with the associated meaning:

Particle vectors

$vector\ real\ X, Y, Z - particle\ coordinates\ (should\ be\ in\ the\ box)[nm?]$

$vector\ real\ V_x, V_y, V_z - particle\ speeds\ [nm/ns]$

$vector\ real\ F_x, F_y, F_z - particle\ forces\ [N]$

MD parameters

$constant\ real\ dt - time\ between\ two\ particle\ coordinate\ updates\ (0.04)[ps]$

$constant\ integer\ p - number\ of\ updates\ before\ another\ neighbour\ search\ (10)[-]$

$constant\ real\ X_{Max}, Y_{Max}, Z_{Max} - box\ width, length, depth\ [nm]$

$integer\ N_{Max} - maximum\ number\ of\ neighbours$

$constant\ integer\ P_{Max} - number\ of\ particles\ in\ the\ system$

Particle parameters

$constant\ real\ m - particle\ mass\ (72)[kg/kmol]$

$constant\ real\ C_6, C_{12}$
$- constants\ used\ to\ multiply\ r^{-6}\ and\ r^{-12}, used\ for\ force\ computation\ [N/m]$

$constant\ real\ r_C - cutoff\ radius\ (1.2)[nm]$

Thermostat

$constant\ real\ \tau_t - Berendsen\ thermostat$ strength $(1)[ps]$

$real\ \lambda -\ Berendsen\ thermostat\ speed\ adjust\ value\ (\sim 1, 0.8 \ldots 1.25)[-]$

$real\ T - system\ temperature\ (\sim 315)[K]$

$constant\ real\ T_0 - reference\ temperature\ (315)[K]$

$constant\ real\ k_B - Boltzmann\ constant\ [J/K]$

Where: $real = float\ or\ double\ (float)$

3.2. PERIODIC BOUNDARY CONDITIONS

Periodic boundary conditions (PBCs) are a set of boundary conditions which are often chosen for approximating a large (infinite) system by using a small part called a unit cell. PBCs are often used in

computer simulations and mathematical models. The topology of two-dimensional PBC is equal to that of a world map of some video games; the geometry of the unit cell satisfies perfect two-dimensional tiling, and when an object passes through one side of the unit cell, it re-appears on the opposite side with the same velocity. In topological terms, the space made by two-dimensional PBCs can be thought of as being mapped onto a torus (compactification). The large systems approximated by PBCs consist of an infinite number of unit cells. In computer simulations, one of these is the original simulation box, and others are copies called images. During the simulation, only the properties of the original simulation box need to be recorded and propagated. The minimum-image convention is a common form of PBC particle bookkeeping in which each individual particle in the simulation interacts with the closest image of the remaining particles in the system.

In molecular dynamics simulation, PBC are usually applied to calculate bulk gasses, liquids, crystals or mixtures. A common application uses PBC to simulate solvated macromolecules in a bath of explicit solvent.

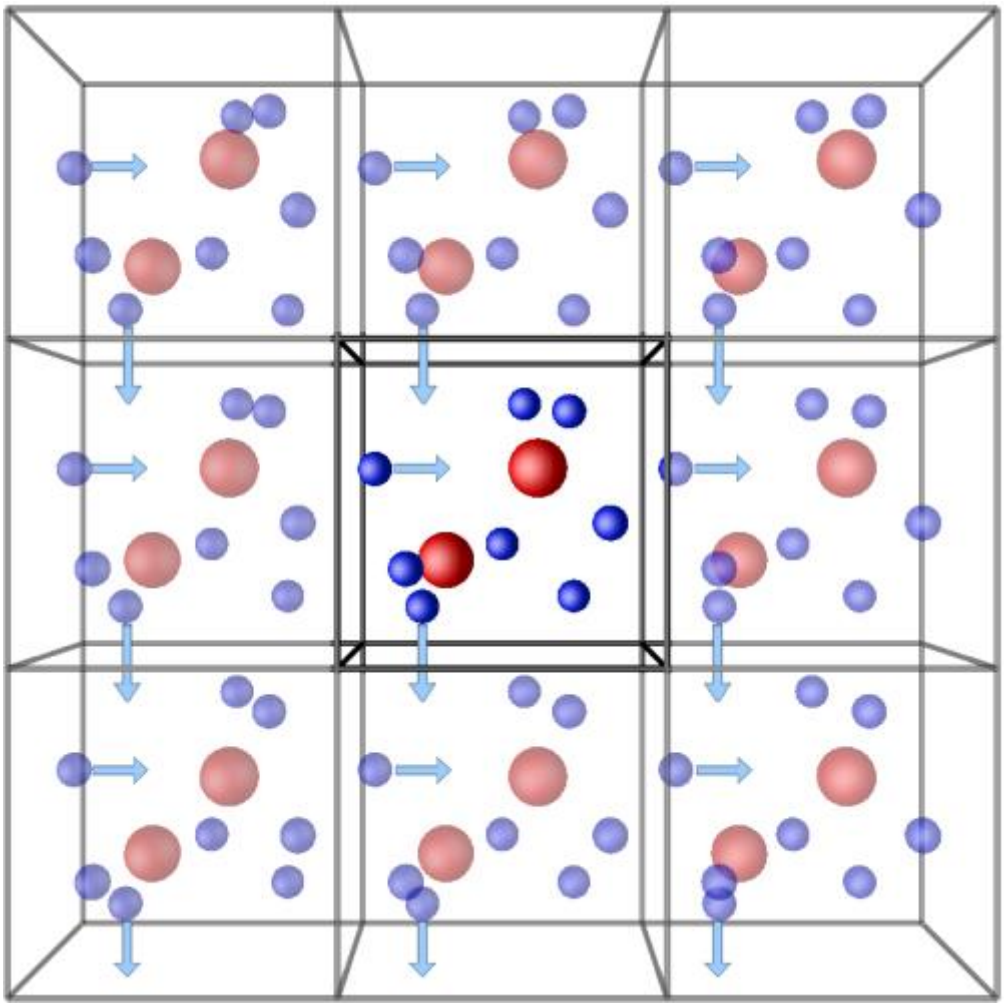


Figure 3-1: Schematic representation of the idea of periodic boundary conditions [29]

The following algorithm has been implemented:

1. Check if any particle is outside of the simulation box
2. If the particle is outside, create a new particle entering the simulation box (preserve speed) in the opposite direction from where it left the box
3. Delete the old particle

3.3. PSEUDOCODE IMPLEMENTATION OF THE PBC

// What's being computed

all particle DIM coordinates must be between 0 and DIM_Max, where DIM is X, Y or Z

// Pseudo-code

$select\ particles\ where\ X \geq X_{Max}$

$X = X - X_{Max}$

$select\ all\ particles$

$select\ particles\ where\ X < 0$

$X = X + X_{Max}$

$select\ all\ particles$

$select\ particles\ where\ Y \geq Y_{Max}$

$Y = Y - Y_{Max}$

$select\ all\ particles$

$select\ particles\ where\ Y < 0$

$Y = Y + Y_{Max}$

$select\ all\ particles$

$select\ particles\ where\ Z \geq Z_{Max}$

$Z = Z - Z_{Max}$

$select\ all\ particles$

$select\ particles\ where\ Z < 0$

$Z = Z + Z_{Max}$

$select\ all\ particles$

3.4. NEIGHBOUR SEARCHING

To reduce simulation time, the number of interactions must be reduced to the most important ones. Van der Waals interactions are only representative to up to a distance of $r_c = 1.2 \dots 1.3\,nm$. Anything else is considered as not interacting with the particle.

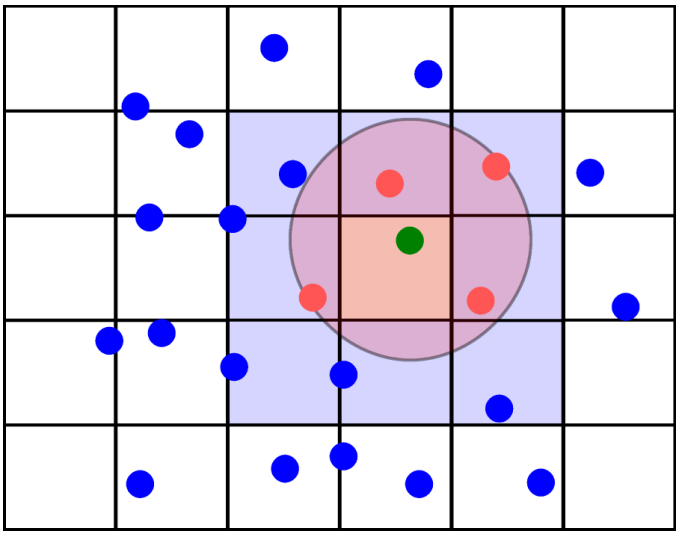


Figure 3-2: Neighbors for the current particle [30]

3.5. PSEUDOCODE IMPLEMENTATION OF THE NEIGHBOUR SEARCH

// What's being computed

$if\left(\sqrt{dx^2 + dy^2 + dz^2} < r_C\right) \rightarrow neighbour$

// Variables

$real\ vector\ t_x, t_y, t_z, t_r$

$integer\ i$

// Pseudo-code

$for\ each\ particle, i - 1 \ldots P_{Max}\{$

// Particle i is not a neighbour to itself

$select\ except\ particle\ i$

// Start calculating the distance between particle i and the rest of the particles

// Create new temp vectors for X, Y, Z dimensions and subtract i^{th} particle coordinates

$t_x = X - repeat(X[i], P_{Max})$

$t_y = Y - repeat(Y[i], P_{Max})$

$t_z = Z - repeat(Z[i], P_{Max})$

// Calculate the squares

$t_{x^2} = t_x * t_x$

$t_{y^2} = t_y * t_y$

$t_{z^2} = t_z * t_z$

// Add the squared vectors

$t_r = t_{x^2} + t_{y^2} + t_{z^2}$

// Make neighbour list

$select\ where\ t_r > 1/r_c^2$

$add\ particle\ i\ as\ neighbour\ to\ the\ currently\ selected\ particles$

$select\ all\ particles$

}

3.6. FORCE COMPUTATION

The interaction between the particles in the system is only a non-bonded interaction, a Van der Waals interaction, described by the Lennard-Jones potential.

The Lennard-Jones potential (also referred to as the L-J potential, 6-12 potential, or 12-6 potential) is a mathematically simple model that approximates the interaction between a pair of neutral atoms or molecules. A form of this interatomic potential was first proposed in 1924 by John Lennard-Jones. [31] The most common expressions of the L-J potential are:

$$V_{LJ} = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] = \varepsilon\left[\left(\frac{r_m}{r}\right)^{12} - 2\left(\frac{r_m}{r}\right)^{6}\right] = \frac{C_{12}}{r^{12}} - \frac{C_6}{r^6}$$

Where:

- ε is the depth of the potential well;
- σ is the finite distance at which the inter-particle potential is zero;
- r is the distance between the particles;
- r_m is the distance at which the potential reaches its minimum.

At r_m, the potential function has the value −ε. The distances are related as $r_m = 21/6\sigma \approx 1.122\sigma$. These parameters can be fitted to reproduce experimental data or accurate quantum chemistry calculations.

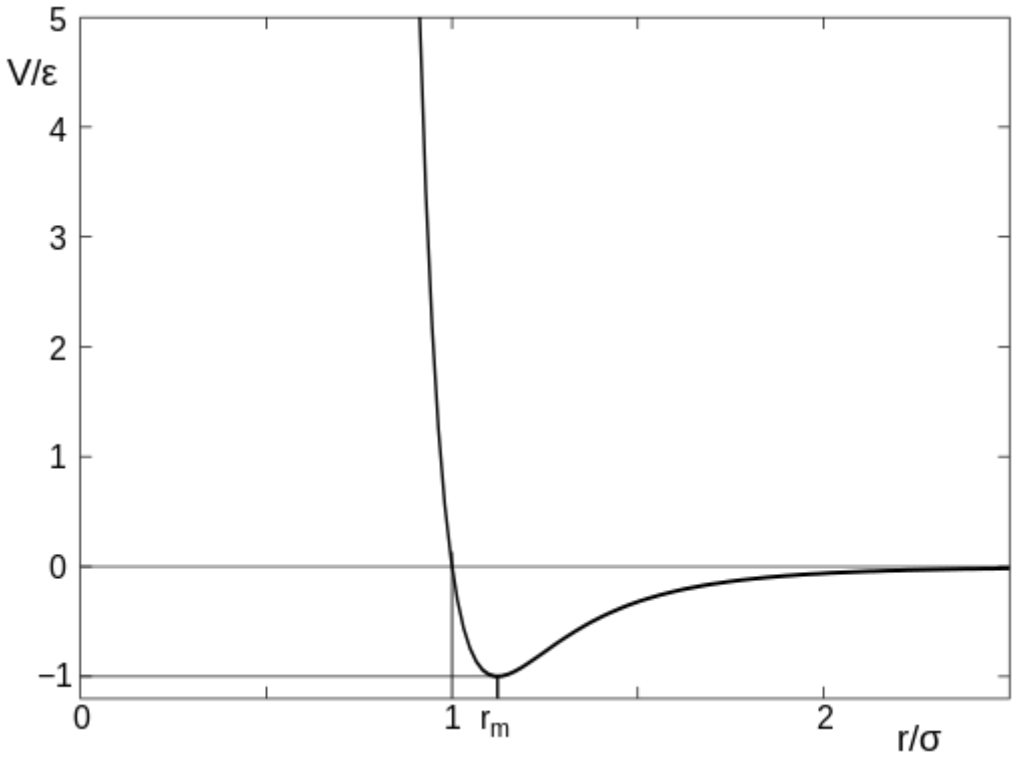


Figure 3-3: A graph of strength versus distance for the 12-6 Lennard-Jones potential.

Due to its computational simplicity, the Lennard-Jones potential is used extensively in computer simulations even though more accurate potentials exist.

3.7. PSEUDOCODE IMPLEMENTATION OF FORCE COMPUTATION

// What's being computed

$$F(r) = \left(\frac{C_{12}}{r^{12}} - \frac{C_6}{r^6}\right)\frac{1}{r}$$

// Variables

$real\ vector\ F_{resc_X}, F_{resc_Y}, F_{resc_Z}, F_{vdw}$

$real\ vector\ t_{r_{inv}1}, t_{r_{inv}2}, t_{r_{inv}4}, t_{r_{inv}7}$

$real\ C_6, C_{12}$

// Pseudo-code

$F_{resc_X}, F_{resc_Y}, F_{resc_Z} = repeat(0, P_{Max})$

//dx, dy and dz have been already calculated at step 3.5

$for\ each\ neighbour, j - 1 \ldots N_{Max}\{$

$t_{r_{inv}1} = 1/t_r$

$$t_{r_{inv}2} = t_{r_{inv}1} * t_{r_{inv}1}$$
$$t_{r_{inv}4} = t_{r_{inv}2} * t_{r_{inv}2}$$
$$t_{r_{inv}7} = t_{r_{inv}4} * t_{r_{inv}2} * t_{r_{inv}1}$$

$$F_{vdw} = C_{12} * t_{r_{inv}7} - C_6 * t_{r_{inv}4}$$

$$F_{resc_X} += F_{vdw} * t_x$$
$$F_{resc_Y} += F_{vdw} * t_y$$
$$F_{resc_Z} += F_{vdw} * t_z$$

}

3.8. TEMPERATURE COUPLING

In any system, we must take the necessary steps to prevent or treat the generation of free energy. One of these steps is the temperature coupling. I've chosen to implement the Berendsen thermostat [32], which is an algorithm to re-scale the velocities of particles in molecular dynamics simulations to control the simulation temperature.

3.9. PSEUDOCODE FOR TEMPERATURE COUPLING

// What's being computed

$$\lambda = \sqrt{1 + \frac{dt}{\tau_t} \cdot \frac{T_0}{T - 1}};\ T = \frac{\sum mv^2}{k_B \cdot nrdf}$$

// Variables

$real\ vector\ V_x^2, V_y^2, V_z^2, V^2$

$real\ constant\ k_B = 8.3144621455e - 03$

$real\ constant\ m = 72$

$integer\ constant\ nrdf = 3P_{Max} - 3$

$real\ temp$

$real\ \lambda$

// Pseudo-code

$select\ all\ particles$

$V_x^2 = V_x * V_x$

$V_y^2 = V_y * V_y$

$V_z^2 = V_z * V_z$

$V^2 = V_x^2 + V_y^2 + V_z^2$

$temp = m * sum\_reduce(V^2) * {k_B}^{-1} * nrdf^{-1}$

// First two terms from the series expansion for λ should be enough

$$\lambda \approx 1 + \frac{1}{2} \cdot \frac{T_0 \cdot dt}{\tau_t} \cdot \frac{1}{T-1} - \frac{1}{8} \cdot \left(\frac{T_0 \cdot dt}{\tau_t} \cdot \frac{1}{T-1}\right)^2 + \frac{1}{16} \cdot \left(\frac{T_0 \cdot dt}{\tau_t} \cdot \frac{1}{T-1}\right)^3 + \cdots$$

$if\ \lambda < 0.8\ then\ \lambda = 0.8\ else$

$if\ \lambda > 1.25\ then\ \lambda = 1.25$

3.10. COORDINATE AND VELOCITY UPDATING

After all the previous steps have been completed, the new coordinates and the new speed of the particles can be computed.

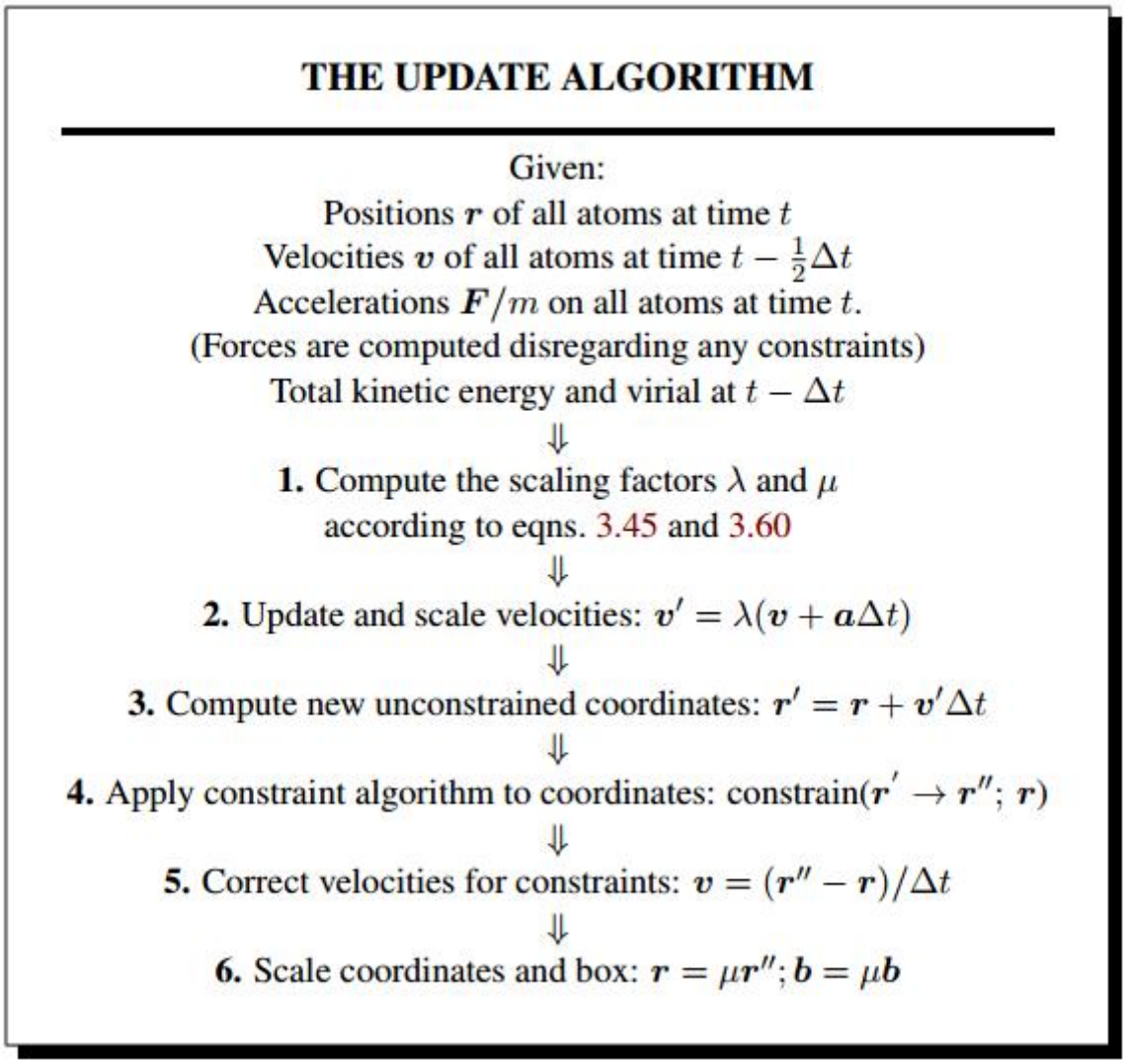


Figure 3-4: GROMACS update algorithm [25]

3.11. PSEUDOCODE FOR COORDINATE AND VELOCITY UPDATING

// Pseudo-code

$$V_x = \lambda * V_x + F_{resc_x} * \frac{dt}{m}$$

$$V_y = \lambda * V_y + F_{resc_y} * \frac{dt}{m}$$

$$V_z = \lambda * V_z + F_{resc_z} * \frac{dt}{m}$$

$$X = X + V_x * dt$$

$$Y = Y + V_y * dt$$

$$Z = Z + V_z * dt$$

4. FUTURE WORK

The MD simulation domain is evolving [33], and more simulation power is required to be able to simulate interesting phenomena in reasonable amounts of time. For this, either performance, power consumption or scalability costs need to be significantly reduced.

And additional development time must be spent to make MRA able to perform protein folding simulations:

- Non-bonded interactions (Coulomb interaction etc.)
- Bonded interactions (Bond stretching: Morse potential, Cubic potential, FENE potential, Angle potential: Harmonic, Cosine etc.)
- Parallelization (dynamic load balancing, domain decomposition)
- Quantum molecular potentials
- Integrators (Leap-frog, Verlet)
- Cut-off schemes (group, Verlet, twin-range)
- Temperature coupling (V-rescale, Andersen)
- Pressure coupling (Berendsen, Parrinello-Rahman)
- Outputting steps (trajectory files)

But, before all that, it is my personal opinion that MRA needs a viable compiler.

5. RESULTS AND CONCLUSIONS

Running code present in Annexe 1 – Code on the Map-Reduce architecture simulator, we get the following results:

Full simulation (simulation box periodicity, neighbour search, VdW force computation, Berendsen thermostat and update): 33040 cycles, simulation box periodicity: 80 cycles, neighbour searching: 26410 cycles, force computation: 6409 cycles, Berendsen thermostat: 91 cycles and coordinate / speed update: 50 cycles.

Table 3: Simulation cycles and percent time spent on algorithm

Simulation part	Cycles	Percent of algorithm
Full simulation	33040	100.00%
Box periodicity	80	0.24%
Neighbour search	26410	79.93%
Force computation	6409	19.40%
Thermostat	91	0.28%
Update	50	0.15%

Table 4: Performance for non-bonded interactions [33] [34]

Machine	Cores	NS:F	Freq. [GHz]	Price [USD]	Perf. [µs/day]	Power [W]	Perf./Watt [Wh/µs]
Intel i5	1	1:10	2.7 GHz	$200	5.84	65	267.1
Intel i5 (SSE)	1	1:10	2.7 GHz	$200	9.78	70	171.8
Intel i5	4	1:10	2.7 GHz	$200	18.94	90	114.0
Intel i5 (SSE)	4	1:10	2.7 GHz	$200	31.48	95	72.4
MRA (FPGA)	512	1:1	0.5 GHz	$1000	52.42	35	16.0
MRA (FPGA)	512	1:10	0.5 GHz	$1000	187.01	35	4.5
MRA (ASIC)	512	1:10	1.0 GHz	$10	374.02	3	0.2
Anton	1	–	0.4 GHz	-	572.32	75	3.1
Anton	512	–	0.4 GHz	$10Mil	293027.84	116500	9.5

* expecting Anton 2 in fall 2016

** Intel i5 750 processor was used for comparison

Table 5: MRA cell usage

Simulation part	Active cells	Controller
Full simulation	75.6%	13.8%
Box periodicity	66.8%	0.0%
Neighbour search	79.2%	15.30%
Force computation	60.4%	7.60%
Thermostat	51.0%	62.70%
Update	100.0%	17%

From Table 3 and Table 4 we can see that there are some very interesting and promising results but additional funding is required to take the next design step (FPGA -> ASIC).

Chip development costs are extremely high, complex system-on-chip (SOC) platforms like Ax family (Apple) or the IBM Cell very likely surpassed $1B in total development costs, involving thousands of engineers in total. On the other hand, relatively modest SOCs like the Epiphany family of chips (Adapteva) were designed for less than $3M over the period of several years. Even simpler ASICs like Bitcoin mining chips can be designed for budgets under $1M.

But usually, the costs of hardware development are between $300k - $200M (already done), software development $0 - $800M (partially completed, several tools available but more are needed), chip tape-out ($100k - $3M) and testing $5k - $1M, not involving additional per wafer or per chip costs.

Of course, it's always advantageous to make chips, especially promising ones, where all the benefits (performance, energy consumption) outweigh the development costs.

6. REFERENCES

[1] [Online]. Available: http://www.xvivo.net/.

[2] "Theory of Molecular Dynamics Simulations," [Online]. Available: http://www.ch.embnet.org/MD_tutorial/pages/MD.Part1.html. [Accessed June 2016].

[3] B. J. Alder and T. E. J. Wainwright, *The Journal of Chemical Physics,* vol. 27, p. 1208, 1957.

[4] B. J. Alder and T. E. J. Wainwright, *Chemical Physics,* vol. 31, p. 459, 1959.

[5] A. Rahman, *Physical Review A,* vol. 136, p. 405, 1964.

[6] F. H. Stillinger and A. J. Rahman, *Chemical Physics,* vol. 60, p. 1545, 1974.

[7] J. A. McCammon, B. R. Gelin and M. Karplus, *Nature (Lond.),* vol. 267, p. 585, 1977.

[8] D. McQuarrie, Statistical Mechanics, New York: Harper & Row, 1976.

[9] A. Gottlieb and G. S. Almasi, Highly parallel computing, Redwood City, California: Benjamin/Cummings, 1989.

[10] S. Adve, "Parallel Computing Research at Illinois: The UPCRC Agenda," University of Illinois at Urbana-Champaign, Illinois, November 2008.

[11] K. Asanovíc, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams and K. Yelick, "The Landscape of Parallel Computing Research: A View from Berkley," Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

[12] R. Pike, "Concurrency is not Parallelism," in *Waza conference*, 2012.

[13] J. L. Hennessy, D. A. Patterson and J. R. Larus, Computer organization and design: the hardware/software interface, San Francisco: Kaufmann, 1999.

[14] "https://wiki.haskell.org/Parallelism_vs._Concurrency," [Online]. [Accessed July 2016].

[15] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS spring joint computer conference*, IBM Sunnyvale, California, 1967.

[16] [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law#Speedup_in_a_serial_program.

[17] E. H. B. John L. Gustafson, "Reevaluating Amdahl's Law," in *Communications of the ACM*, 1988.

[18] R. G. J. M. G. Benner, "Development and analysis of scientific application programs on a 1024-processor hypercube," Sandia National Laboratories, Feb. 1988.

[19] [Online]. Available: http://www.futurechips.org/thoughts-for-researchers/parallel-programming-amdahls-law-gustafsons-law.html. [Accessed June 2016].

[20] [Online]. Available: http://www.futurechips.org/thoughts-for-researchers/parallel-programming-gene-amdahl-said.html. [Accessed June 2016].

[21] [Online]. Available: http://www.futurechips.org/software-for-hardware-guys/types-serial-bottlenecks.html. [Accessed June 2016].

[22] [Online]. Available: http://www.futurechips.org/chip-design-for-all/parallel-programming-dependences-loop-iterations.html. [Accessed June 2016].

[23] [Online]. Available: http://www.futurechips.org/tips-for-power-coders/parallel-programming-understanding-impact-critical-sections.html. [Accessed June 2016].

[24] M. A. Suleman, O. Mutlu, M. K. Qureshi and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS'09, 2009.

[25] [Online]. Available: ftp://ftp.gromacs.org/pub/manual/manual-5.1.2.pdf.

[26] S. C. Kleene, "Origins of recursive function theory," *Foundations of Computer Science,* pp. 371-382, 1979.

[27] D. E. Shaw, "Anton, a Special-Purpose Machine for Molecular Dynamics Simulation," D. E. Shaw Research, LLC, New York, 2007.

[28] D. E. Shaw, "Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Piscataway, NJ, USA, 2014.

[29] [Online]. Available: http://isaacs.sourceforge.net/phys/pbc.html. [Accessed June 2016].

[30] [Online]. Available: https://www.computer.org/csdl/trans/td/2014/01/ttd2014010043-abs.html. [Accessed June 2016].

[31] J. E. Lennard-Jones, "On the Determination of Molecular Fields," Proc. R. Soc. , London, 1924.

[32] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola and J. R. Haak, "Molecular-Dynamics with Coupling to an External Bath," Journal of Chemical Physics 81 (8): 3684–3690, 1984.

[33] D. E. Shaw, "Anton: A Specialized ASIC for Molecular Dynamics," D E Shaw Research, LLC, 2008.

[34] [Online]. Available: http://ark.intel.com/products/42915/Intel-Core-i5-750-Processor-8M-Cache-2_66-GHz.

[35] H. Zeiger, "Simulation studies reveal the role disulfide bonds play in protein folding," [Online]. Available: http://phys.org/news/2015-08-simulation-reveal-role-disulfide-bonds.html.

ANNEXE 1 – CODE

```
// ------------------------------------------------------------
// Vectors
`define   X          0   // particle's X coordinates
`define   Y          1   // particle's Y coordinates
`define   Z          2   // particle's Z coordinates
`define   Vx         3   // particle's speed on X axis
`define   Vy         4   // particle's speed on Y axis
`define   Vz         5   // particle's speed on Z axis
`define   dX         6   // delta X between two particles
`define   dY         7   // delta Y between two particles
`define   dZ         8   // delta Z between two particles
`define   DIMXMAX    9   // simulation box width (constant)
`define   DIMYMAX   10   // simulation box length (constant)
`define   DIMZMAX   11   // simulation box height (constant)
`define   SRX       12   // shift register X temp
`define   SRY       13   // shift register Y temp
`define   SRZ       14   // shift register Z temp
`define   Rinv2     15   // 1/r^2 vector
`define   Rinv7     16   // 1/r^7 vector
`define   Rinv13    17   // 1/r^13 vector
`define   NEG       18   // change sign mask
`define   TMP       19   // temporary
`define   RSVD      20   // all vectors below are reserved

// Vector reuse
`define FRX         12   // X decomposition of final force
`define FRY         13   // Y decomposition of final force
`define FRZ         14   // Z decomposition of final force
`define Fvdw        15   // final Van der Waals force
`define V           15   // final speed
```

```
// ------------------------------------------------------------
// Memory locations
`define   P          8   // number of particles in the system
`define   DIMXMAX    9   // already defined above
`define   DIMYMAX   10   // already defined above
`define   DIMZMAX   11   // already defined above
`define   Rc2       12   // squared cut-off radius (rc^2)
`define   One       13   // the value 1 in floating-point
`define   C6        14   // VdW C6 constant
`define   C12       15   // VdW C12 constant
`define   NaN       16   // not-a-number floating-point flag
`define   Ptmp      17   // current particle number
`define   Idx       18   // index
`define   V2T       19   // squared speed to temperature
`define   DTT       20   // reference temperature
`define   Lambda    21   // temperature coupling output
`define   dt_m      22   // dt / particle mass
`define   dt        23   // simulation time step
// ------------------------------------------------------------
// Labels
`define   RSH        0
`define   NS         1
`define   F          2
`define   S          3

// ------------------------------------------------------------
// Cooperand values
`define   Radd       0
`define   Rmin       1
`define   Rmax       2
`define   Rflg       3
`define   SR0        4
```

```
// ------------------------------------------------------------
// Main
// Initialization
                cNOP;                   ACTIVATE;
                cVLOAD(`RSVD);          NOP;
                cNOP;                   CADDRLD;
                cVLOAD(3);              VLOAD(128);
LB(`RSH);       cBRNZDEC(`RSH);         VMULT(64);
                cNOP;                   STORE(`NEG);

                cNOP;                   IXLOAD;
                cSEND(`P);              CCOMPARE;
                cNOP;                   WHERECARRY;

LB(`S);         cSEND(`DIMXMAX);        CLOAD;
                cNOP;                   XOR(`NEG);
                cNOP;                   STORE(`DIMXMAX);
                cSEND(`DIMYMAX);        CLOAD;
                cNOP;                   XOR(`NEG);
                cNOP;                   STORE(`DIMYMAX);
                cSEND(`DIMZMAX);        CLOAD;
                cNOP;                   XOR(`NEG);
                cNOP;                   STORE(`DIMZMAX);

// Particle periodicity
     // X coordinates
                cNOP;                   LOAD(`X);
                cNOP;                   XOR(`NEG);
                cNOP;                   COMPARE(`DIMXMAX);
                cNOP;                   NOP;
                cNOP;                   WHERECARRY;
                cNOP;                   XOR(`NEG);
                cNOP;                   FADD(`DIMXMAX);
                cNOP;                   MADD;
```

```
            cNOP;                   APACK;
            cNOP;                   ENDWHERE;
            cNOP;                   XOR(`NEG);
            cNOP;                   COMPARE(`NEG);
            cNOP;                   NOP;
            cNOP;                   WHERECARRY;
            cNOP;                   FADD(`DIMXMAX);
            cNOP;                   MADD;
            cNOP;                   APACK;
            cNOP;                   XOR(`NEG);
            cNOP;                   ENDWHERE;
            cNOP;                   STORE(`X);
// Y coordinates
            cNOP;                   LOAD(`Y);
            cNOP;                   XOR(`NEG);
            cNOP;                   COMPARE(`DIMYMAX);
            cNOP;                   NOP;
            cNOP;                   WHERECARRY;
            cNOP;                   XOR(`NEG);
            cNOP;                   FADD(`DIMYMAX);
            cNOP;                   MADD;
            cNOP;                   APACK;
            cNOP;                   ENDWHERE;
            cNOP;                   XOR(`NEG);
            cNOP;                   COMPARE(`NEG);
            cNOP;                   NOP;
            cNOP;                   WHERECARRY;
            cNOP;                   FADD(`DIMYMAX);
            cNOP;                   MADD;
            cNOP;                   APACK;
            cNOP;                   XOR(`NEG);
            cNOP;                   ENDWHERE;
            cNOP;                   STORE(`Y);
// Z coordinates
```

```
                cNOP;                   LOAD(`Z);
                cNOP;                   XOR(`NEG);
                cNOP;                   COMPARE(`DIMZMAX);
                cNOP;                   NOP;
                cNOP;                   WHERECARRY;
                cNOP;                   XOR(`NEG);
                cNOP;                   FADD(`DIMZMAX);
                cNOP;                   MADD;
                cNOP;                   APACK;
                cNOP;                   ENDWHERE;
                cNOP;                   XOR(`NEG);
                cNOP;                   COMPARE(`NEG);
                cNOP;                   NOP;
                cNOP;                   WHERECARRY;
                cNOP;                   FADD(`DIMZMAX);
                cNOP;                   MADD;
                cNOP;                   APACK;
                cNOP;                   XOR(`NEG);
                cNOP;                   ENDWHERE;
                cNOP;                   STORE(`Z);

// Particle neighbour search
          // Preparation
                cVLOAD(0);              NOP;
                cSTORE(`Idx);           NOP;
                cLOAD(`P);              LOAD(`X);
                cVSUB(1);               STORE(`SRX);
                cNOP;                   LOAD(`Y);
                cNOP;                   STORE(`SRY);
                cNOP;                   LOAD(`Z);
                cNOP;                   STORE(`SRZ);
          // X cords
LB(`NS);        cNOP;                   LOAD(`SRX);
                cNOP;                   SRSTORE;
```

```
        cCSEND(`SR0);           CLOAD;
        cNOP;                   XOR(`NEG);
        cNOP;                   FADD(`X);
        cNOP;                   MADD;
        cNOP;                   APACK;
        cNOP;                   STORE(`dX);
        cNOP;                   FMULT(`dX);
        cNOP;                   MPACK;
        cNOP;                   STORE(`dX);
        cNOP;                   SRSHLEFT;
        cNOP;                   SRLOAD;
        cNOP;                   STORE(`SRX);
    // Y coords
        cNOP;                   LOAD(`SRY);
        cNOP;                   SRSTORE;
        cCSEND(`SR0);           CLOAD;
        cNOP;                   XOR(`NEG);
        cNOP;                   FADD(`Y);
        cNOP;                   MADD;
        cNOP;                   APACK;
        cNOP;                   STORE(`dY);
        cNOP;                   FMULT(`dY);
        cNOP;                   MPACK;
        cNOP;                   STORE(`dY);
        cNOP;                   SRSHLEFT;
        cNOP;                   SRLOAD;
        cNOP;                   STORE(`SRY);
    // Z coords
        cNOP;                   LOAD(`SRZ);
        cNOP;                   SRSTORE;
        cCSEND(`SR0);           CLOAD;
        cNOP;                   XOR(`NEG);
        cNOP;                   FADD(`Z);
        cNOP;                   MADD;
```

```
            cNOP;                   APACK;
            cNOP;                   STORE(`dZ);
            cNOP;                   FMULT(`dZ);
            cNOP;                   MPACK;
            cNOP;                   STORE(`dZ);
            cNOP;                   SRSHLEFT;
            cNOP;                   SRLOAD;
            cNOP;                   STORE(`SRZ);
        // Calculate squared distance between selected particle
and the rest
            cNOP;                   LOAD(`dX);
            cNOP;                   FADD(`dY);
            cNOP;                   MADD;
            cNOP;                   APACK;
            cNOP;                   FADD(`dZ);
            cNOP;                   MADD;
            cNOP;                   APACK;
        // Select all particles that are "close"
            cSEND(`Rc2);            CCOMPARE;
            cNOP;                   WHERECARRY;
            cNOP;                   STORE(`TMP);
            cSTORE(`Ptmp);          IXLOAD;
            cSEND(`Idx);            WHEREOP;
            cNOP;                   NOP;
            cLOAD(`Idx);            ELSEWHERE;
            cVADD(1);               LOAD(`TMP);
        // Store distance between particles
            cSTORE(`Idx);           RISTORE(1);
            cLOAD(`Ptmp);           LOAD(`dX);
            cNOP;                   RISTORE(1);
            cNOP;                   LOAD(`dY);
            cNOP;                   RISTORE(1);
            cNOP;                   LOAD(`dZ);
            cNOP;                   RISTORE(1);
```

```
                cNOP;                   ENDWHERE;
                cBRNZDEC(`NS);          ENDWHERE;
                cSEND(`NaN);            CLOAD;
                cNOP;                   RISTORE(1);
// Force calculation
                cVLOAD(`RSVD);          NOP;
                cNOP;                   CADDRLD;
                cVLOAD(0);              NOP;
                cNOP;                   CLOAD;
                cNOP;                   STORE(`FRX);
                cNOP;                   STORE(`FRY);
                cNOP;                   STORE(`FRZ);
LB(`F);         cNOP;                   RILOAD(1);
                cSEND(`NaN);            WHEREOP;
                cNOP;                   ELSEWHERE;
                cNOP;                   STORE(`TMP);
                cSEND(`One);            CLOAD;
                cNOP;                   FDIV(`TMP);
                cNOP;                   MDIV;
                cNOP;                   DPACK(`TMP);
                cNOP;                   STORE(`Rinv2);
                cNOP;                   FSQRT;
                cNOP;                   NOP;
                cNOP;                   FMULT(`Rinv2);
                cNOP;                   MPACK;
                cNOP;                   FMULT(`Rinv2);
                cNOP;                   MPACK;
                cNOP;                   FMULT(`Rinv2);
                cNOP;                   MPACK;
                cNOP;                   STORE(`Rinv7);
                cNOP;                   FMULT(`Rinv2);
                cNOP;                   MPACK;
                cNOP;                   FMULT(`Rinv2);
                cNOP;                   MPACK;
```

```
                cNOP;                   FMULT(`Rinv2);
                cNOP;                   MPACK;
                cNOP;                   STORE(`Rinv13);
                cSEND(`C6);             CLOAD;
                cNOP;                   FMULT(`Rinv7);
                cNOP;                   MPACK;
                cNOP;                   XOR(`NEG);
                cNOP;                   STORE(`Fvdw);
                cSEND(`C12);            CLOAD;
                cNOP;                   FMULT(`Rinv13);
                cNOP;                   MPACK;
                cNOP;                   FADD(`Fvdw);
                cNOP;                   MADD;
                cNOP;                   STORE(`Fvdw);
                cNOP;                   RILOAD(1);
                cNOP;                   FMULT(`Fvdw);
                cNOP;                   MPACK;
                cNOP;                   FADD(`FRX);
                cNOP;                   MADD;
                cNOP;                   STORE(`FRX);
                cNOP;                   RILOAD(1);
                cNOP;                   FMULT(`Fvdw);
                cNOP;                   MPACK;
                cNOP;                   FADD(`FRY);
                cNOP;                   MADD;
                cNOP;                   STORE(`FRY);
                cNOP;                   RILOAD(1);
                cNOP;                   FMULT(`Fvdw);
                cNOP;                   MPACK;
                cNOP;                   FADD(`FRZ);
                cNOP;                   MADD;
                cBRAACT(`F);            STORE(`FRZ);
// Temperature coupling
                cNOP;                   ACTIVATE;
```

```
      cNOP;                    IXLOAD;
      cSEND(`P);               CCOMPARE;
      cNOP;                    WHERECARRY;

      cNOP;                    LOAD(`Vx);
      cNOP;                    FMULT(`Vx);
      cNOP;                    MPACK;
      cNOP;                    STORE(`V);

      cNOP;                    LOAD(`Vy);
      cNOP;                    FMULT(`Vy);
      cNOP;                    MPACK;
      cNOP;                    FADD(`V);
      cNOP;                    MADD;
      cNOP;                    STORE(`V);
// Compute system temperature, Lambda
      cNOP;                    LOAD(`Vz);
      cNOP;                    FMULT(`Vz);
      cNOP;                    MPACK;
      cNOP;                    FADD(`V);
      cNOP;                    MADD;
      cNOP;                    STORE(`V);
      cNOP;                    NOP;
      cNOP;                    NOP;
      cNOP;                    NOP;
      cNOP;                    NOP;
      cCLOAD(5);               NOP;
      cFMULT(`V2T);            NOP;
      cMPACK;                  NOP;
      cXOR(`NEG);              NOP;
      cFADD(`One);             NOP;
      cMADD;                   NOP;
      cAPACK;                  NOP;
      cXOR(`NEG);              NOP;
```

```
                cSTORE(`Ptmp);          NOP;
                cLOAD(`One);            NOP;
                cFDIV(`Ptmp);           NOP;
                cMDIV;                  NOP;
                cDPACK(`Ptmp);          NOP;
                cFMULT(`DTT);           NOP;
                cMPACK;                 NOP;
                cFADD(`One);            NOP;
                cMADD;                  NOP;
                cAPACK;                 NOP;
                cFSQRT;                 NOP;
                cSTORE(`Lambda);        NOP;
// Update speed vector
                cCSEND(`Lambda);        CLOAD;
                cNOP;                   FMULT(`Vx);
                cNOP;                   MPACK;
                cNOP;                   STORE(`Ptmp);
                cCSEND(`dt_m);          CLOAD;
                cNOP;                   FMULT(`FRX);
                cNOP;                   MPACK;
                cNOP;                   FADD(`Ptmp);
                cNOP;                   MADD;
                cNOP;                   STORE(`Vx);

                cCSEND(`Lambda);        CLOAD;
                cNOP;                   FMULT(`Vy);
                cNOP;                   MPACK;
                cNOP;                   STORE(`Ptmp);
                cCSEND(`dt_m);          CLOAD;
                cNOP;                   FMULT(`FRY);
                cNOP;                   MPACK;
                cNOP;                   FADD(`Ptmp);
                cNOP;                   MADD;
                cNOP;                   STORE(`Vy);
```

```
                cCSEND(`Lambda);        CLOAD;
                cNOP;                   FMULT(`Vz);
                cNOP;                   MPACK;
                cNOP;                   STORE(`Ptmp);
                cCSEND(`dt_m);          CLOAD;
                cNOP;                   FMULT(`FRZ);
                cNOP;                   MPACK;
                cNOP;                   FADD(`Ptmp);
                cNOP;                   MADD;
                cNOP;                   STORE(`Vz);
// Update coordinates
                cCSEND(`dt);            CLOAD;
                cNOP;                   FMULT(`Vx);
                cNOP;                   MPACK;
                cNOP;                   FADD(`X);
                cNOP;                   MADD;
                cNOP;                   STORE(`X);

                cCSEND(`dt);            CLOAD;
                cNOP;                   FMULT(`Vy);
                cNOP;                   MPACK;
                cNOP;                   FADD(`Y);
                cNOP;                   MADD;
                cNOP;                   STORE(`Y);

                cCSEND(`dt);            CLOAD;
                cNOP;                   FMULT(`Vz);
                cNOP;                   MPACK;
                cNOP;                   FADD(`Z);
                cNOP;                   MADD;
                cJMP(`S);               STORE(`Z);

                cHALT;                  NOP;
```