Combining Algorithms and Technologies to Speedup Computing Intensive Applications

Mauro Bisson

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the Degree of
Doctor of Philosophy in Computer Science
at the Sapienza University of Rome

December 15, 2010
Contents

Introduction 4

1 GPUs and CUDA overview 10
  1.1 The CUDA programming model ....................... 12
  1.2 Memory Hierarchy ............................... 16

2 Distributed dictionary attacks 19
  2.1 OpenPGP Standard ............................... 20
  2.2 Attack Strategy .................................. 21
    2.2.1 Dictionary Compilation Phase .................. 22
    2.2.2 Passphrase Generation Phase ................... 22
    2.2.3 Passphrase Verification Phase.................. 23
  2.3 Distributed Architecture ......................... 27
    2.3.1 General Requirements ......................... 27
    2.3.2 Overall Organization .......................... 27
    2.3.3 System Life-cycle ............................. 29
  2.4 Implementation ................................ 32
    2.4.1 MIDIC .................................. 33
    2.4.2 The dcrack Application ..................... 35
  2.5 Experimental results ............................. 37
  2.6 GPU Implementation .............................. 38
    2.6.1 Input .................................. 40
Introduction

In the present work we study two different, compute-intensive problems. The first is the execution of dictionary attacks to cryptographic systems. Cryptosystems are used everyday by millions of people around the world to lawfully enforce their privacy rights (e-commerce transactions, exchange of reserved data, etc...). While most of the security of cryptosystems lies in the algorithms they use, the growing computational power of commodity computers poses a serious threat since passwords and passphrases used by most of the users of cryptosystems are far from being ideal in terms of randomicity. With this in mind, we developed a distributed system to carry out large scale dictionary attacks against cryptosystems by tapping the computational power of computers spread over a geographic network, like the Internet. Our end goal was to study the impact that the coordinated use of commodity hardware can have on the security of digital communications.

The second problem that we face is the study of the hemodynamics. The original motivation for this work is the study of the genesis and evolution of atherosclerotic plaques. These plaques grow inside human coronary arteries and, with time, may block the blood flow bringing oxygen to the heart muscle leading to serious problems like heart attacks and strokes. A highly detailed comprehension of the dynamics of blood flows inside the cardiovascular system is necessary to improve our understanding of cardiovascular diseases, which are the most common cause of death in Western countries. Hemodynamics simulations require huge computational resources and today they can be performed only on state of the art supercomputers. With the advancement of computing technology we believe that
in the next few years our system could be implemented for a commodity hardware platform to help medical doctors in diagnosing cardiovascular diseases by running patient-specific simulations of hemodynamics.

For the implementation of the computational parts of both projects we focused on Graphics Processing Units (GPU), an emerging computing architecture evolved from processors dedicated to graphics rendering to highly parallel, many-core processors with exceptional computational power. In chapter 1 we present an overview of GPU architecture and programming model. This chapter is basically a summary of the document “CUDA Programming Guide” by NVIDIA [28].

In chapter 2 we present our system to perform distributed dictionary attacks to cryptosystems [1, 2]. A dictionary attack is a technique that, unlike brute force attacks, where the whole key space is searched systematically, only tries those passwords/passphrases that are most likely to succeed. The candidate passphrases are derived from a list of words, the dictionary, to which a set of generation rules are applied. The effectiveness of a dictionary attack is based on the tendency of users to choose passwords/passphrases that can be easily remembered, like sentences found in books, with simple variations on the characters and/or on individual words. As a target of our system we chose OpenPGP [3], a widely used solution for encryption and decryption of email messages and files. We attack its most famous implementation, the GnuPG cryptographic suite [4], at its security core: the secret keyring (secring). The attack target are the private keys stored in the secring. The knowledge of the keys allows the attacker to perform critical operations as the legitimate owner of the secring. The attack strategy consists of three phases, performed sequentially, each of which receives as input the output of the preceding step. The first phase is devoted to build the dictionary used during the second phase, in which there is the generation of passphrase candidates. The third and last phase consists of the test of every generated passphrase.

This work is divided in three parts. In the first part (section 2.2) we describe a
novel and simplified approach to quickly testing passphrases used to protect private keyrings of OpenPGP cryptosystems. This method allows to test passphrases in a fraction of the time required by the standard algorithm provided by GnuPG. The second part (section 2.3) presents a distributed architecture designed to deploy the three phases of the attack over networks of heterogeneous computers. In the last part (section 2.4) we detail the implementation, for Graphics Processing Units (GPU), based on the Compute Unified Device Architecture (CUDA), of the passphrases verification phase of the attack [5]. This combination of hardware, algorithm and implementation, reduces the time required to test a set of possible passphrases by three orders of magnitude if compared to an attack based on the standard algorithm provided by software packages like GnuPG and a ten-fold speed up if compared to our best CPU implementation. This result also shows that while in the last few years most of the works about GPUs focused on their floating point operation performance, GPUs should also be considered in strictly integer manipulation applications.

The second part of this dissertation describes a series of works on computational physics targeted to the study of the hemodynamics inside human arteries. Building a detailed, realistic model of hemodynamics is a formidable computational challenge. The simulation must combine the motion of the fluids, the intricate geometry of the blood vessels, continual changes in flow and pressure driven by the heartbeat, and finally the behavior of red and white blood cells and other suspended bodies, such as platelets and lipids. To this purpose, it is necessary to use different physical methods for fluid and particles dynamics in a multiscale [6] model of the system. The Lattice Boltzmann (LB) method [7] is used to simulate the fluid component of blood, the plasma, and Molecular Dynamics (MD) is used for suspended bodies like red and white blood cells.

1Cells are represented as non-bonded rigid bodies whose dynamics is usually called Particle Dynamics. However, since the code produced in this work can handle particle systems driven
In chapter 3 we briefly describe the physical multiscale model we adopt to simulate blood dynamics.

From the computational viewpoint these simulations are really demanding. Acceptable representations of human arterial geometry requires LB meshes composed of hundreds of millions of fluid nodes, at spatial resolutions of 20 microns, up to billions of nodes to reach resolutions comparable with that of red blood cells (10 microns). In addition to this, the translational and rotational motion of hundreds of millions particles need to be tracked down and the interaction between particles and fluid computed. This must be done at every simulation step. The only viable approach to fulfill such computational demand is to resort to parallel processing. However the parallelization of these simulations introduces additional challenges.

Since the spatial domain represents an anatomical system (artery vessels) it has a very irregular and sparse geometry. Because of the irregularity it is not possible to use a simple Cartesian partitioning of the domain (in cubes or slabs) without severely affecting the global load balancing among the processors. For this reason we resort to a graph partitioning based approach that allows to obtain a quasi-optimal partitioning from the load balancing point of view but that results in subdomain that are quite irregular. While it is well known how to parallelize the Lattice Boltzmann method inside irregular subdomains this is not true for Molecular Dynamics.

In chapter 4 we propose a novel, efficient and general method to parallelize MD inside irregular domains [8]. Our method relies on a decomposition of the subdomains that allows to handle arbitrarily complex geometries while still enabling to take advantage of known algorithms for particle-particle interactions. Moreover it allows to drastically limit the amount of particles data that need to be exchanged among processors at each simulation step.

In chapter 5 we present the complete MD implementation for multi-GPU plat-
forms [9] based on our MD parallelization method. This implementation has been
written from scratch in order to exploit at its best the computational power pro-
vided by modern GPUs, explicitely designed for high performance computing.
The code has been integrated into MUPHY [10], a concurrent multiscale code for
large-scale bio-fluidic simulations in anatomically realistic geometries. The basic
MUPHY scheme combines a hydrokinetic (Lattice Boltzmann) representation of
the blood plasma, with a Molecular Dynamics treatment of suspended biological
bodies, such as red blood cells. Performance tests show excellent results, with
a nearly linear parallel speed-up on up to 32 GPUs and a more than tenfold
GPU/CPU acceleration, all across the range of available GPUs.

To the best of our knowledge, this represents the first effort in the direction of
laying down general design principles for multiscale/physics parallel Molecular Dy-
amics applications in non-ideal geometries. This positions the present multi-GPU
version of MUPHY as one of the first examples of a high-performance parallel code
for multiscale/physics biofluidic applications in realistically complex geometries.

In order to roughly compare the performance of GPU clusters with those of
well known supercomputing platforms, and to further validate our MD method on
a radically different architecture, we implemented it also for the IBM BlueGene/P
in many aspects, the most important of which regard the computing units (nodes)
and the communication links connecting them. BlueGene systems are equipped
with a large number of small, low-power nodes (up to hundreds of thousands) con-
nected through specialized, high-speed networks. On the contrary, GPU clusters
available today are characterized by a relatively small number (up to hundreds)
of extremely powerful nodes, the GPUs, connected by a sub-optimal communica-
tion infrastructure. As a matter of fact, each data transfer among GPUs requires
a network transfer between source and destination host and two memory trans-
fers between host and device memory. The effectiveness of our MD method on
both multi-GPU clusters and the BlueGene/P system is an indication of its gen-

eral validity regardless of the specific platform. For what concerns the performance comparison, in our tests a single Tesla GPU performed as hundreds of BlueGene/P nodes.

Finally, in chapter 6, we present the largest simulation of the entire heart-circulation cardiovascular system ever done, performed with the BlueGene version of MUPHY that implements our MD parallelization technique. This work, detailed in [12], has been selected as finalist for the prestigious “Gordon Bell” award at SUPERCOMPUTING2010. We run the first large-scale simulation of blood flow in the coronary arteries and other vessels supplying blood to the heart muscle, with a realistic description of human arterial geometry at spatial resolutions from centimeters down to 10 microns (near the size of red blood cells). This multiscale simulation resolves the fluid into a billion volume units, embedded in a bounding space of 300 billion voxels, coupled with the concurrent motion of 300 million red blood cells, which interact with one another and with the surrounding fluid. The level of detail is sufficient to describe phenomena of potential physiological and clinical significance, such as the development of atherosclerotic plaques. This simulation was performed on the IBM BlueGene/P supercomputer of the Jülich supercomputing centre, ranked 5th in the Top 500 list of supercomputers as of June 2010. The code achieved excellent scalability on up to 294,912 computational cores and sustained performance of 64 Teraflops. The performance obtained by the code on this BlueGene/P system corresponds to simulating a full heartbeat at microsecond resolution in only a few hours on the whole BlueGene/P system.
Chapter 1

GPUs and CUDA overview

In the last 5 years, driven by the insatiable market demand for realtime, high-definition 3D graphics, Graphic Processor Units or GPUs have evolved into a highly parallel, multithreaded, manycore processors with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figure 1.1.

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 1.2.

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-
parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

In November 2006, NVIDIA introduced CUDA, a general purpose parallel computing architecture (with a new parallel programming model and instruction set architecture) that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

Figure 1.1. Floating-Point operations per second for the CPU and GPU.
CUDA comes with a software environment that allows developers to use C as a high-level programming language.

1.1 The CUDA programming model

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore’s law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores. The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C. At its core are three key abstractions, a hierarchy of thread groups, shared memories, and barrier synchronization, that are simply exposed to the programmer as a minimal set of language extensions. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved
A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

cooperatively in parallel by all threads within the block. This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Figure 1.3, and only the runtime system needs to know the physical processor count.

This scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of processors and memory partitions:
from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs.

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed $N$ times in parallel by $N$ different CUDA threads, as opposed to only once like regular C functions. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in $threadIdx$ variable.

For convenience, $threadIdx$ is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume. The index of a thread and its thread ID relate to each other in a straightforward way: for a one-dimensional block, they are the same; for a two-dimensional block of size $(Dx, Dy)$, the thread ID of a thread of index $(x, y)$ is $(x + yDx)$; for a three-dimensional block of size $(Dx, Dy, Dz)$, the thread ID of a thread of index $(x, y, z)$ is $(x + yDx + zDxDy)$.

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks. Blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by Figure 1.4. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

The number of threads per block and the number of blocks per grid are specified in the kernel call. Each block within the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in $blockIdx$ variable. The dimension of the thread block is accessible within the kernel through
the built-in `blockDim` variable.

Thread blocks are required to execute independently: it must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 1.3, enabling programmers to write code that scales with the number of cores. Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the a specific intrinsic function that acts as a barrier at which all threads in the block must wait before any is allowed to proceed. For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache).
1.2 Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 1.5. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages (see [28]). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see [28]). The global, constant, and texture memory spaces are persistent across kernel launches by the same application.
As illustrated by Figure 1.6, the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (see [28]). This includes device memory allocation and deallocation as well as data transfer between host and device memory.
Figure 1.6. Heterogeneous programming: serial code executes on the host while parallel code executes on the device.
Chapter 2

Distributed dictionary attacks

A dictionary attack is a technique for defeating a cryptographic system by searching its decryption key or password/passphrase in a pre-defined list of words or combinations of these words that, for any reason, are more likely than a random combination of characters. Although it is widely accepted that the main factor for the success of a dictionary attack is the choice of a suitable list of possible words, the efficiency and reliability of the platform used for the attack may become critical factors as well. In this chapter, we present a distributed architecture for performing dictionary attacks that can exploit resources available in local/wide area networks by hiding all details of the communication among participating nodes. As an example of possible cryptographic challenge for which the platform can be used, we selected the decryption of a private keyring generated by the GnuPG software package. From this viewpoint, the present work can be considered a replacement and an extension of pgpcrack (that is no longer available), an utility used for cracking PGP. Note that the structure of the GnuPG secring is much more complex with respect to the original PGP. To the best of our knowledge, no equivalent fast cracking system exists for GnuPG. Other scalable distributed cracking systems were proposed in [13] and [14]. We do not present here a detailed comparison, but we just mention that our solution pays much more attention to reliability and
The chapter is organized as follows: section 2.1 describes the features of OpenPGP, the standard to which GnuPG makes reference; section 2.2 describes our approach to the attack of the GnuPG keyring; section 2.3 introduces the architecture we propose for the distributed attack; section 2.4 gives some information about the current CPU implementation; in section 2.5 are discussed some experimental results and, finally, section 2.6 describes the GPU implementation of the attack.

2.1 OpenPGP Standard

OpenPGP is a widely used standard for encryption and authentication of email messages and files. It is defined by the OpenPGP Working Group in the Internet Engineering Task Force (IETF) Standard RFC 4880 [3]. OpenPGP derives from PGP (Pretty Good Privacy), a software package created by Phil Zimmermann in the beginning of nineties. GnuPG [4] is a well-known public domain software implementation of the OpenPGP standard. New commercial versions of PGP are also compliant to the OpenPGP standard.

The OpenPGP standard adopts a hybrid cryptographic scheme. For instance,
message encryption uses both symmetric and asymmetric key encryption algorithms. The sender uses the recipient’s public key to encrypt a shared key (i.e. a secret key) for a symmetric algorithm. That key is used to encrypt the plaintext of the message or file. The recipient of a PGP encrypted message decrypts it using the session key for a symmetric algorithm. The session key is included in the message in encrypted form and it is decrypted in turn by using the recipient’s private key. These keys are stored in two separate data structures, called “keyrings”: private keys in the private keyring, public keys in the public keyring. Every keyring is a list of records, each of which associated to a different key. In order to prevent disclosures, private keys are encrypted with a symmetric algorithm, by using a hash of a user-specified passphrase as secret key. For what concerns GnuPG, as shown in Figure 2.1, the asymmetric encryption algorithm is El Gamal [15], the hash algorithm is SHA1 [16] and the symmetric encryption is CAST5 [17], used in CFB mode [3].

### 2.2 Attack Strategy

One of the most critical issues regarding OpenPGP security is the secrecy of passphrases protecting private keys. The knowledge (by any means achieved) of the passphrase gives the chance to a malicious user to execute critical operations as signature and decryption of messages belonging to the legitimate owner of the private keyring. For this reason, the attack to the OpenPGP system aims at finding the passphrase associated to a private keyring stored according to the OpenPGP format. The attack is divided in three phases, each of which receives as input the output of the preceding step, as shown in Figure 2.2.

The first phase is devoted to build the dictionary used during the second phase in which there is the generation of the passphrases. The third phase consists of the test of every generated passphrase against the private key ciphertext found in the secring.
2.2.1 Dictionary Compilation Phase

In this phase, the basic dictionary is created starting from a set of text files. The procedure is quite simple: each different word is placed in the list that constitutes the dictionary. In order to increase chances of success, the content of these text files should contain information somehow related to the legitimate owner of the private keyring under attack. This process is depicted in Figure 2.3.

2.2.2 Passphrase Generation Phase

This second phase produces a list of passphrases by applying a set of generation rules to all words found in the dictionary. Every rule involves the current word and
a chosen number of subsequent words and allows the generation of passphrases, by performing permutations of the order of words and/or substitutions of single characters. In this way, the obtained passphrases are reasonably compliant with the basic rules of a natural language.

For instance, if we apply rules that involve a word and four subsequent words to generate passphrases with a length ranging from one to five words, for each word in the dictionary we obtain 39 possible passphrases:

- the current word as in the dictionary, then the same word with all lower case letters and all upper case letters (3 passphrases).
- the current word and the following one, taken in the original order and in the reverse order, with all lower case letters, all upper case letters and the unmodified case (6 passphrases).
- all possible permutations of the current word and the two subsequent words, with all lower case letters and all upper case letters (18 passphrases).
- the current word and the three subsequent words, taken with the order and the case in the dictionary and in reverse order, with all lower case letters and all upper case letters (6 passphrases).
- the current word and the four subsequent, taken with the order and the case in the dictionary and in reverse order, with all lower case letters and all upper case letters (6 passphrases).

Note that in the generation of passphrases with four and five words, some permutations are not considered, since they yield sequences unlikely for human memorization. The generation phase is depicted in Figure 2.4.

2.2.3 Passphrase Verification Phase

This phase is the core of the attack and the most expensive from the computational point of view. Each passphrase generated in the previous phase is checked
by following an incremental approach aimed at minimizing the cost of the controls required by the OpenPGP standard. For this reason, a symmetric key for the CAST5 algorithm is derived from every passphrase, by applying the SHA1 algorithm in iterated and salted mode. Such a key is used to try a decryption of encrypted components relating to the private key. This process is represented in Figure 2.5. The hashing procedure applied to the passphrase to generate the CAST5 decryption key is called string-to-key (S2K) procedure. Because SHA1 is used in iterated mode, this mechanism is computationally very expensive. By iterating and salting there is a significant increase in the time required to check a possible passphrase. This is a security precaution commonly adopted by many modern cryptosystems. Iterating many times increases the testing time of the single passphrase in a way that is unnoticeable for an authorized user but that, for an attacker, could mean orders of magnitude of additional work. Moreover, salting the passphrases prevents precomputed rainbow tables attacks [18].

In order to decide whether the passphrase under test is the right one, it is necessary to verify the plaintext obtained from the decryption procedure. This operation is performed taking into account how the OpenPGP standard represents
components relating to the private keys in keyrings.

As shown in Figure 2.6, a private key is represented by a Multi Precision Integer (MPI), followed by its SHA1 hash:

$$[b_l b_{l-1} | b_{l-2} ... b_{21}] [b_{20} b_{19} ... b_1]$$

The standard procedure employed by GnuPG to check the passphrase consists in hashing the MPI $b_l b_{l-1} ... b_{21}$ and comparing the result with the hash $b_{20} b_{19} ... b_1$ included in the plaintext. If they match, then it is verified the fulfillment of the correct algebraic relationship between the MPI and the corresponding public key.

In order to speedup the passphrase check, we introduce in the standard verification process, a preliminary control: we test whether the left part of the plaintext is a well-formed MPI (see Figure 2.6). An MPI consists of two parts: a two-byte scalar that is the length of the MPI in bits (starting from the most significant non-zero bit) followed by a string of bytes that contain the actual integer. For instance, the string $[00 09 01 FF]$ is a well-formed MPI with the value of 511 whereas the string $[00 0A 01 FF]$ is not a valid MPI. Our preliminary test consists in the verification that the two-byte scalar $b_l b_{l-1}$ represents the bit length of the string $b_{l-2} ... b_{21}$. Since the size $l$ of the plaintext is known (it equals the size of the

Figure 2.5. The third phase: verification of the passphrase.
plaintext

\[
\begin{array}{c|c}
\text{MPI } x & \text{HASH}(x) \\
\hline
\text{private key} & \text{hash SHA1} \\
\end{array}
\]

Figure 2.6. Validation test.

We can compute the size, in bits, of the integer field of the MPI as:

\[
l_{\text{MPI}} = (l - s - m) \times 8 - l_z(b_{l-2})
\]

where \(s\) is the size of the SHA1 hash (20 bytes), \(m\) the MPI length field (2 bytes), 8 is the number of bits in a byte and \(l_z(b_{l-2})\) is the number of leading zeroes of the most significant byte of the integer field. Henceforth, we will refer the check:

\[
\text{is } b_{lb_{l-1}} \text{ equal to } l_{\text{MPI}}? 
\]

as the fast test. If the fast test succeeds, we continue with the standard procedure described above.

Note that the fast test and the SHA1 hash check have a low computational cost but they may produce false positives. The last test, consisting in the verification of the algebraic relationship between public and private keys, is exact but it is computationally very expensive.

GnuPG does not carry out our first control that is already very selective: it discards around ninety-five percent of invalid passphrases. By following this multi-step procedure our validation test is much more cost-effective.
2.3 Distributed Architecture

The attack described in the previous section has been deployed over a loosely coupled distributed architecture. The three phases of the attack are scattered over the nodes of the network. There is a root node and two different groups of peers that share their computational resources.

2.3.1 General Requirements

Since our solution has been conceived to work with heterogeneous systems in a geographic context, the proposed architecture guarantees the following requirements:

**scalability:** the number of network nodes can be easily increased, augmenting the available computational power.

**load balancing:** the computational load must be distributed among the nodes according to their capabilities to prevent local starvation.

**flexibility:** since the availability of each node in the network is unpredictable, the architecture must be able to adapt itself to variations of available resources by changing the load distribution.

**fault tolerance:** possible failures of a node must not compromise the overall computation, thus the system must be able to re-assign any workload and to recover local computation.

2.3.2 Overall Organization

The proposed architecture consists of three levels, each of which implements a specific phase of the attack, as represented in Figure 2.7. Each level receives information from the upper level, elaborates them and then supplies the lower level.
The first level is constituted by a single “root” node, denoted as $r$, that is responsible for the compilation of the dictionary. The second level consists of a variable number of nodes, named “generators” and denoted as $g$, that form the “generation network” $G$. Such a network is devoted to the generation of passphrases starting from the dictionary compiled in the first phase by the “root” node. The third level consists of a variable number of nodes, named “verifiers” and denoted as $v$, that form the “verification network”. Such a network is in charge of verifying whether any of the generated passphrases decrypts the private key given in input. Node $r$ and the sets of nodes $G$ and $V$ form the network system $\sum = \langle r, G, V \rangle$.

System $\sum$ has a tree-like topology where generator nodes play the role of children of root node $r$. Verifier nodes $v$ are divided in groups, each of which is assigned to a generator node $g$, as depicted in Figure 2.7. Each node acts as client with respect to the parent node and as server with respect to any child node.

Every node performs a specific task:

- root node $r$ compiles the dictionary $D$, divides it in partitions $P_i(D)$ and
assigns the $i^{th}$ partition to the generator node $g_i$;

- each generator node $g_i$ extracts from $P_i(D)$ a list of passphrases $L$ and divides it in partitions $P_j(L)$. Every partition $P_j(L)$ is assigned to a verifier node $v_j^i$ (where the superscript $i$ indicates that $v_j$ is a child of $g_i$);

- each verifier node $v_j^i$ checks all passphrases in the assigned partition $P_j(L)$ with respect to the private key provided in input.

This model of interaction, represented in Figure 2.8, makes easier to achieve a reasonable load-balancing by assigning more work to groups with more verifier nodes. Every node of the network needs to know only the identifier of its parent node, of the “root” node and of all its child nodes (if any), in order to communicate with them. Moreover, for each of its child nodes, a parent node checks the status of available resources and stores the last messages sent to it. Information stored in a node are maintained until child nodes do not confirm the completion of operations assigned to them. Child nodes never communicate each other.

Communication occurs by means of messages that require an explicit receiver’s confirmation. A node accepts messages coming just from the parent, its children and, possibly, the root. Messages can be grouped as follows:

**task messages:** used to exchange information about the attack;

**maintenance messages:** used for handling asynchronous events related to the network;

**heart-beating messages:** aimed at detecting failures and sending information about available resources.

### 2.3.3 System Life-cycle

An instance of the system begins with just the root node. As new nodes join the network to participate in the attack (this is done by sending a message to
the root node), generation and verification networks are populated. A new node is assigned to the generation network if there are no generator nodes (this is the typical situation in the beginning), or if all existing generator nodes serve already the maximum number of verifier nodes (this maximum number can be tuned at run time). Otherwise, the new peer becomes a verifier node and it is assigned as child to the generator node having the lowest number of children. The expansion model of the system is shown in Figure 2.9.

An instance of the system ends when the correct passphrase for the given private key is found. The verifier node on which a candidate passphrase passes successfully the first two controls described in section 2.2.3 sends it to its parent generator node. This node performs further controls (the final test described in section 2.2.3) and, on success, finally forwards the passphrase to the root node that, as a consequence, stops the system. This process is depicted in Figure 2.10. For what concerns the single nodes, every peer can be in one of the following states:

- **running**: the node is performing its own task;
serving: the node is executing the assigned task and performing a maintenance operation that involves one or more child nodes;

stopped: the node is not executing a task because it is involved in a maintenance operation launched by its parent node or by itself;

The root node can be in either running or serving state, a generator node can be in running, serving or stopped state, a verifier node can be in either running or stopping state. State transitions occur when a message is received, or as a consequence of a local event.

Nodes execute maintenance operations when local events take place. The events must be compatible with the current state of the nodes. Usually, an event triggers a transition in a state where the corresponding maintenance operation is carried out. Three kinds of events are possible:

soft-quitting (SQ): produced when a node explicitly leaves out the system;

hard-quitting (HQ): generated when a node detects an unexpected quitting of a child, for example due to a child failure.

swapping (SW): event that occurs when a node exchanges its role with a child.

Each node manages its soft-quitting related operations and hard-quitting related operations of its children. Verifier nodes, since do not have children, do not
need to manage hard-quitting and swapping events. Moreover, the root node can not swap its role with a child.

If the root node suddenly quits the network, the entire instance of the system halts, unless the root node explicitly migrates all the information related to the instance to another system that becomes the new root. A possible future extension of the present architecture is to provide support for “multiple” roots that are automatically updated and may substitute, in a transparent way, the original root in case of failure.

When a failure (HQ operation) occurs in a generator node, the root appoints one of the orphan verifier nodes as new generator node for the remaining orphans nodes and assigns to it pending partitions left by the failed generator node. When a generator node wants to quit the system (SQ operation), it elects a substitute, choosing it among its verifier nodes, and supplies to it all the information required to complete the task. Finally, the outgoing node informs the root node and quits.

When a failure occurs in a verifier node (HQ operation), the parent generator node forwards to other child nodes the pending list of passphrases previously assigned to the broken node and informs the root. When a verifier node wants to quit the system (SQ operation), it informs its parent generator node about the number of checked passphrases in its pending list. The generator node then supplies residual passphrases to its other child verifier nodes and informs the root note. Finally, generator nodes, in case of variation of their own resources with respect to those available to child verifier nodes, may swap their role with one of the child verifier nodes (SW operation), in order to assign to the verification network the most performing nodes.

2.4 Implementation

The system has been implemented in a single application, named dcrack, that is able to perform all the three phases of the attack, i.e., dictionary compilation,
passphrase generation and passphrase verification. In such a way, the same application runs on every node of system. The code has been implemented in ANSI C taking into account the requirement of being usable in a multi-platform environment. To this purpose, the application relies only on portable components as shown in Figure 2.12. In particular, we use the Apache Portable Runtime (APR) [19], a set of Application Programming Interfaces (API) that guarantees software portability across heterogeneous platforms, through a replacement of functions that are not supported in the underlying operating system. For instance, the use of the APR environment allows to exploit synchronization mechanisms like the “condition-variables”, available in the Windows environment only with the latest versions. As to the networking issues, we resorted to the MIDIC middleware that we briefly describe in the following section.

2.4.1 MIDIC

MIDIC is a middleware [20] that allows to set up a computing infrastructure composed by the aggregation of resources belonging to independent institutions (e.g. Research Laboratories, Universities). Each institution provides all required resources (e.g., servers and network bandwidth) and services (e.g., job scheduling and user authentication) to support computing activities of its users. Institutions store their public data in Domain Name System (DNS) tables, thus allowing
other institutions to discover and interact with them. The infrastructure supports institutions interested in the execution of computing applications which require a significant amount of CPU time and operate in contests where: i) safety and privacy are not binding factors; ii) the low cost of the solution is a factor of outstanding importance, and iii) high technical proficiency for systems management and applications development is not necessary. The MIDIC infrastructure supports applications that can be organized according to the master-worker model. Systems belonging to the infrastructure can act as masters, by publishing jobs and tasks, or as workers by running tasks. Tasks published by a master can be assigned to workers of external institutions.

The infrastructure supports both master and workers for issues related to: i) application cycle management; ii) fair distribution of the resources among the applications (to prevent some applications from monopolizing donated resources), and iii) exchange of data and messages.

MIDIC includes two components: a server MiddlewareEndPoint and a user MiddlewareEndPoint. The server MiddlewareEndPoint is the server side component which runs on institutions servers and provides user MiddlewareEndPoints and other server MiddlewareEndPoints with required institution services. The user MiddlewareEndPoint is the client side of the middleware. It runs on users machines (usually personal computers) and provides applications with API to access middleware services. To satisfy the main goal to support a broad range of applications, and since it runs on users machines, it has been implemented so that it: i) is portable on major Operating Systems (e.g. Windows, Linux and MacOS), ii) supports applications written in different programming languages (e.g., C, C++, JAVA, Perl and Python), iii) allows applications to communicate across firewalls, and iv) reduces the impact of unpredictable user actions (e.g., sudden reboot or power-off). The portability requirement is satisfied by resorting to the APR library. Moreover, the API is implemented as Remote Procedure Calls (RPC) by using a combination of HTTP and XML. In this way, the API supports any lan-
guage that can handle TCP network communications and XML data. We adopt P2P communication protocols (JXTA) to enable cross-firewall communications. Finally, the Sqlite3 embedded database is used to store data managed by the user MiddlewareEndPoint. This allows the middleware to persist data on disk by reducing the risk of data loss in case of crash or power-off. A schematic description of the user MiddlewareEndPoint is shown in figure 2.11.

2.4.2 The dcrack Application

_Dcrack_ is subdivided in components, each of which implements a specific function in the node where it runs. The subdivision is made on the basis of a logical classification of activities common to all nodes:

**task execution:** each task is made of two components, the _worker_ that acquires and processes information about the attack and the _server_ that returns the results of required computations;

**maintenance operations:** such operations are managed by a _controller_ component, for what concerns quitting the system and failures, and by a _recruiter_ component for the entry of new nodes in the system.
heart-beating activity: this activity is carried out by a beater component for sending heart-beating messages to child nodes and by a heart component for receiving such messages.

Active components of the application for the three classes of nodes and the communication flow among them are shown in Figure 2.13. Task messages are sent from the Work component to the Server component of the application running on the parent node. Heart-beating messages are sent from the Beater component of the application running on the parent node to the Heart component running on the child nodes. Maintenance messages are exchanged between the controller component of the application running on a child node and the corresponding component in the parent node.

Each component runs in a thread whose implementation depends on the platform (but this is transparent to the application since it relies on the APR environment). Cooperation among threads follows the work-crew model. Moreover, threads in charge of components that may require simultaneous communications (i.e., the server, controller and recruiter components) generate a service thread to which the communication is demanded. Cooperation among component threads
and service threads follows the *boss-worker* model.

## 2.5 Experimental results

We measured the performances of the proposed architecture in a test-bed constituted by a 100baseT Ethernet LAN with 20 personal computers, equipped with a 2.8 Ghz Intel Pentium IV processor and 512Mb of RAM running the Linux operating system. As sample target of the attack, we selected the GnuPG cryptographic software with an ElGamal key having a length of 768 bits.

To generate the dictionary we started from the text of “Divina Commedia” and, as a consequence, generated passphrases are in Italian. In order to evaluate the throughput of the system we chose a passphrase that could not be found with this dictionary, forcing the system to generate and test all passphrases that could be derived from the input text and the defined passphrase generation rules.

Before starting the full experiment, we carried out some preliminary tests, in order to find out how many verifier nodes could be fed by a single generator node. Therefore, the following parameters have been evaluated: $k$, the number of passphrases that can be checked by a verifier node in a second; $t_g$, the time required to a generator node to generate $k$ passphrases; $t_s$, the time required to a generator node to compress and send $k$ passphrases to a verifier node.

Our tests showed that a verifier node is able to check about 1000 passphrases per second. A generator node requires 0.6 ms to generate 1000 passphrase and about 10ms to compress and send them. Thus, a generator node needs about 11ms to set up the workload that a verifier node carries out in one second. As a consequence, the adequate ratio $R$ between the number of generator nodes and verifier nodes is given by:

$$R = 1/(t_g + t_s) = (1/0.011) \sim 90$$

In other words, with these settings, each generator node could feed up to 90
verifier nodes.

In the test environment, we used a variable number of nodes but, since the time required to generate, compress and send passphrases is about two orders of magnitude smaller than the time used for verification, we used, in all tests, a single generator task that coexisted with the root task on a single node (same computer) of the network.

The results we obtained are very encouraging, since the throughput of the system (measured as the inverse of the time required to test all possible passphrases) increases in a linear way with respect to the number of verifier nodes.

2.6 GPU Implementation

In order to speedup the computational intensive phase of the attack we decided to implement on GPU the passphrase verification method.

To the best of our knowledge no cryptanalytic system targets directly the secret ring (secring) of GnuPG by using GPUs. Most of the attention in the use of these new architectures has been focused on the Advanced Encryption Standard (AES) [21, 22, 23]. In [23] the authors use NVIDIA cards through shader programming instead of exploiting the Compute Unified Device Architecture (CUDA). Other papers explored the use of CUDA for the Serpent [24] and RSA [25] algorithms. Public implementations of the Message Digest 5 (MD5)\(^1\) and the Data Encryption Standard (DES) [26] algorithm, are also available.

Implementing the whole standard verification procedure applied to passphrases by using the GPU technology would be overwhelming and ineffective because the vast majority of passphrases does not pass the fast test. For this reason our GPU implementation consists of the S2K procedure, the ciphertext decryption algorithm and the fast test. Thanks to the fast test it is possible to develop an efficient GPU implementation able to discard a large number of invalid passphrases.

\(^1\)http://majuric.org/software/cudamd5/
Our problem is embarrassingly data parallel. Each passphrase is independently valid or not. Moreover, we may expect to have a computational bottleneck rather than a bandwidth bottleneck because the number of operations performed per passphrase is very high.

The objective is to leverage our fast test to create a fast GPU filter that selects from the input buffer only those passphrases that pass our first check, obtaining a much smaller list of potential candidates.

Our solution follows the conventional execution flow of GPU applications (see Figure 2.14) and involves three steps: i) load the passphrases buffer into device memory; ii) process the data by using the GPUs; iii) copy the results back into host memory to be analyzed.

To take advantage of the GPU acceleration our implementation needs to collect a fair amount of possible passphrases, upload them into the GPU memory, process all of them at once and inspect the result.

Our CUDA version of the attack program has limited memory requirements and can run on any recent video card produced by NVIDIA. Device memory is almost exclusively allocated to store the passphrases with a negligible footprint.
for what concerns constant secring data and output structures. For example, in
128MB of device memory can be stored \(\sim 6.5M\) of passphrases. The GPU version
of the validation phase described here has not yet been implemented inside the
distributed architecture explained previously in this chapter. The integration of
the distributed architecture, containing the CPU version of the attack, with the
GPU implementation of the verification phase, will be part of our future activities.
Referring to the application scheme in Figure 2.13, the GPU implementation runs
to carry out part of the duties of the Worker thread within the verifier nodes.

A large increase in performance can be expected by using our distributed ver-
sion of the attack (described in 2.3 and [2]) on a even moderate network of simple
personal computers by exploiting both CPU and GPU resources.

2.6.1 Input

In order to use the GPU, the input data required by the verification phase must
be copied into device memory. They consist of the passphrase list and the secring.

For simplicity we use as input dictionary a simple text file with one word per
line. Supporting arbitrarily sized passphrases would complicate the logic of the
kernel and compromise its efficiency. For our experiments, we decided to limit the
maximum size of a passphrase to sixteen characters, a length that is fast to load
on a GPU device and that is usually over the average size of human-recallable
passphrases.

Passphrases are copied in device memory inside a linear buffer divided in el-
ments of 16 chars (see Figure 2.15). Each passphrase is stored in one element.
The trailing null delimiter is maintained only for strings having fewer than sixteen characters; this convention allows the storage of exactly sixteen characters instead of fifteen plus the null character.

The only negative impact of having a fixed max passphrase length is in the case of short passphrases (i.e. four chars). In this case many of the buffer elements would be almost empty. We sacrifice memory space for speed and logic simplicity.

![Figure 2.16. Bandwidth Test on a Tesla C1060.](image)

This buffer is then used as input by the kernels devoted to the S2K procedure. Because the number of passphrases in the input dictionary can be arbitrarily large, we split them into chunks to fill the passphrase buffer. The application continues to read the input dictionary in chunks until either a passphrase is found or the dictionary is depleted. We measured the bandwidth of the GPU by loading the
buffer in chunks of incrementing sizes (see Figure 2.16). From the results we can see that to achieve 99% of throughput we need, at least, a chunk of 4MB in size.

For what concerns the secring, the relevant data are copied in constant memory. The constant space is a special, read-only, memory that is directly referenceable by the kernel. Copied data include the salt and byte count for the iterated and salted mode of SHA1, the Initialization Vector (IV) used by CAST5 and finally the secret key ciphertext, the target of our attack.

2.6.2 Computing Kernels

The passphrases in the input buffer are processed by the S2K kernel to produce the corresponding CAST5 keys. These keys are in turn read by the CAST5 kernel that, by using the constant secring data, carries out the decryption of secret key ciphertext and the fast test on the resulting plaintext.

We initially developed two independent and complete kernels for the S2K and CAST5 steps. The arguments of each kernel are:

- **SHA1 kernel**: passphrase buffer, passphrase length, salt, byte count and output buffer for digests (CAST5 keys).

- **CAST5 kernel**: key buffer, Initialization Vector, secret key ciphertext and output buffer for the fast test outcomes.

Each thread processes the passphrase corresponding to its global ID:

\[ ID = \text{blockIdx} \times \text{blockDim} + \text{threadIdx} \]

where blockIdx is the ID of the thread block, blockDim is the kernel block size and threadIdx is the ID of the thread within the block.

Each thread processes only one item (passphrase or key) corresponding to its ID. If the thread ID is greater than the total number of items, then it terminates. This is a simple and effective choice, since it avoids complex logic and any kind of synchronization requirement.
Through a process of fine tuning and continuous testing, we subsequently merged the SHA1 and CAST5 kernels into a single and faster kernel.

**String-to-key Kernel**

The string-to-key (S2K) mechanism requires to generate the SHA1 digest of the passphrases in iterated and salted mode.

The procedure requires to apply SHA1 to a sequence of bytes by repeating the concatenation of the salt to the passphrase (see Figures 2.17 and 2.18). The salt and the length of the sequence are both specified inside the secring. Since that sequence is quite large ($2^{16}$ bytes), allocating the whole buffer for it (for every thread) could be overwhelming. The solution adopted is to store only enough space to handle the data required by a round of the SHA1 algorithm (64 bytes).

**CAST5 Kernel**

The CAST5 procedure is divided in two phases. In the first step the state of the cipher is initialized, by using a set of byte-transformation tables. Before launching the kernel we load these tables inside the constant memory of the device. Although the constant memory is limited, the tables are small enough to fit inside...
it. Each thread loads from device memory the key (produced by the S2K kernel) corresponding to its own global ID.

In the second step, an amount of encrypted data equal to the CAST5 block size (64 bit), is read from constant memory for the decryption. SHA1 digests are used as keys for the CAST5 block cipher. If the length of the ciphertext is greater than the cipher block size, multiple iterations are required to complete the decryption.

Once the first three bytes (i.e. the first part of the MPI number) are decrypted, we perform the fast test. If the test is successful, we store in global memory the thread ID value inside the corresponding result element, otherwise we store a negative value (-1).

**S2K+CAST5 Kernel**

While the registers and local memory are flushed away at each kernel invocation, the global memory of the device is not. Instead of launching the SHA1 kernel, storing its results in memory, launching the CAST5 kernel, fetching the results from SHA1 and computing the final result, we merged the two kernels into a single one, avoiding in such a way, $2 \times N$ accesses to global memory, where $N$ is the number of passphrases processed by a block of GPU threads.

The final kernel performs the following steps:

1. Loads the secring data from the constant memory.
2. Loads the passphrase from the global memory.
3. Computes the S2K.
4. Applies the CAST5 cipher to the ciphertext.
5. Carries out the fast test.
6. Saves in global memory the result of the test.
2.6.3 Output

After the kernel completes, the whole result buffer is copied back into host memory so that the CPU can scan it. The result data structure is simply an array with one element for each analyzed passphrase. The element is an integer equal to either -1, if the passphrase did not pass our fast test, or a non negative number corresponding to the index of the passphrase in the input array. When a valid entry is found, the CPU thread retrieves the corresponding passphrase and completes the last two tests of the standard procedure. Only the correct passphrase can pass the last test so, in case of success, it is communicated to the user and the attack ends. In the distributed version of the attack, the final result would be notified to the root node [2].

We investigated the chance of using a reduction mechanism, the classic parallel technique of stream compaction via prefix-sum [27] to select from the result structure only the positive values, compact them and download only the relevant part of it instead of the whole buffer. Due to the small footprint of data transmission, there are few benefits in using this technique so we dropped it in favor of a simpler code.

2.6.4 Code Optimizations

This section describes the various techniques we employed to improve the execution speed of our attack when running on GPU. Due to the novelty of the architecture and programming paradigm, we tried different approaches before reaching a satisfactory speed. Our main reference in this study has been the excellent CUDA Best Practices documentation provided by NVIDIA [29].

We started by measuring the time required to process a dictionary of one million passphrases of different length. The test was conducted on a Intel Xeon E5462 CPU (2.8 Ghz, 6 MB of cache) running a single thread by using randomly generated passphrases between five and sixteen characters long as input, with no
additional rules applied to them. The resulting reference time is 505 seconds.

Of all the three steps performed on the GPU, the string-to-key step is the main focus of our optimizations.

**String-to-Key Optimization**

To gain some more insight into the mechanism of the byte-shuffling operation performed by the S2K procedure and get some ideas on possible optimizations, we analyzed the machine code produced by the compiler for the GPU.

A large number of registers is spilled in the iterated and salted phase. This happens because of the continuous use of the local buffer, to store and fetch the state of the SHA1 algorithm and to populate and read the salted buffer.

The contents of the buffer are loaded into the registers, processed and then swapped back to local memory, resulting in a large number of data movement operations.

The main kernel optimizations implemented are the loop unwinding of the SHA1 round logic and the preprocessing of the data access pattern for the buffer.

Loop unwinding (or loop unrolling) is a classic optimization technique [30]. The goal of this technique is to increase a program speed by reducing or eliminating the instructions controlling the loop, such as pointers arithmetic and end of loop tests, that occur on each iteration. Alternatively loops can be re-written as a repeated sequence of similar independent statements, thus eliminating this overhead and improving the data parallelism among register operations.

In our case we have a sequence of loops for each stage of the SHA1 algorithm. With the aid of a script program², we generate a CUDA-language source code of the unrolled version.

The resulting kernel is 20 times faster than the CPU code (see Table 2.1). To process our test case of one million passphrases, it takes only 26 seconds, compared to the former time of 505 seconds.

²written in the Python language, www.python.org
Table 2.1. GPU Version 1: Loop unroll.

Actually, the speedup is not only a consequence of the additional free registers but also of the intrinsic data parallelism that we introduced by unrolling and untangling the operations of the loop. Many of the SHA1 inner operations are independent from each other and this code change allows the threads to fill the hardware pipeline consistently.

The next optimization regards the access pattern to the buffer used to salt and iterate the passphrase. Our target hash is the result of SHA1 on the entire buffer created by repeating the concatenation of the salt to the passphrase. As already mentioned, we store only the intermediate block required by a single round (512 bit) and fill it iteratively with the salt and the passphrase.

However, the observation that the bytes used to feed the S2K procedure are always the same (i.e. the salt and the passphrase), suggested to create a reduced buffer containing only the salt and the passphrase. To simplify the operations we extended this buffer with four more bytes of the first part of the salt in order to simplify the lookup operations (more details below). The resulting, reduced buffer is arranged in such a way that is easy to extract the required sixteen values for one round of SHA1.

To fully comprehend this optimization, it is necessary to describe how the SHA1 round works. To start a SHA1 round, we need a buffer of exactly 512 bit (64 bytes). Sixteen 32 bit values are extracted and processed from this buffer (16 × 4 bytes). For each of these input values, we compute the position in the circular buffer of each of their four bytes.
for (int it = 0; it < 1024; it++) {
    int offset = (it * 64);

    // compute input value 1/16
    start = 0 * 4 + offset;
    start = start - shift * (start / shift);
    x00 = (buff + start)[0] << 24 |
          (buff + start)[1] << 16 |
          (buff + start)[2] << 8 |
          (buff + start)[3];

    // compute input value 2/16
    start = 1 * 4 + offset;
    start = start - shift * (start / shift);
    x01 = (buff + start)[0] << 24 |
          (buff + start)[1] << 16 |
          (buff + start)[2] << 8 |
          (buff + start)[3];

    ...  

    // compute input value 16/16
    start = 15 * 4 + offset;
    start = start - shift * (start / shift);
    x15 = (buff + start)[0] << 24 |
          (buff + start)[1] << 16 |
          (buff + start)[2] << 8 |
          (buff + start)[3];

    // execute one SHA1 round
    SHA1(x00, x01, x02, ..., x15);
}
<table>
<thead>
<tr>
<th>Version</th>
<th>Time (Seconds)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU single thread</td>
<td>505</td>
<td>1×</td>
</tr>
<tr>
<td>GPU v.1 loop unrolling</td>
<td>28</td>
<td>20×</td>
</tr>
<tr>
<td>GPU v.2 reduced buffer</td>
<td>24</td>
<td>22×</td>
</tr>
</tbody>
</table>

*Table 2.2.* GPU Version 2: improved access pattern.

Assuming that the sequence to be hashed is 65536 bytes long, we can find which bytes will form each of the sixteen values: for each passphrase there will be 1024 iterations (65536 / 64). From the circular buffer we extract sixteen values $x_00, x_01, \ldots, x_{15}$ and then execute a SHA1 round (see Code 1).

By following this approach, we achieved a $22\times$ speed up over the original implementation with a speed up of $\sim 1.2\times$ over the previous kernel (see Table 2.2). This change does not have a large impact on the overall performances but it plays an important role when combined with kernel specialization that we describe afterwards.

**Memory Access Optimizations**

One of the main issues to be addressed to use effectively a GPU is the limited data bandwidth available between the CPU main memory and the GPU global memory.

If the number of operations is not more than linear with respect to the size of the input data, then using the GPU may not offer any benefit to an application and actually can harm its performances. This is not our case.

We tested the different mechanisms of data transfer provided by the CUDA API: Simple Malloc Memory, PinnedMemory, WriteCombined and Mapped Write-Combined. The best performances have been achieved by allocating pinned memory and by using only the WriteCombined flag (see Figure 2.19). The usage of the Mapped flag to delegate the fetching of the memory to the driver, did not provide
benefit and occasionally resulted in lower performances.

Data stored in global memory need to be fetched by the threads. The way we load these data can affect performances significantly. The GPU schedules threads grouping them in *warps*, that are sequences of thirty-two threads. For memory requests the threads of a block are divided into *half-warps*, a sequence of sixteen threads. The GPU divides the global memory into segments of 128 bytes. If all the threads of a half-warp request a global memory address within the same memory segment, then the set of requests is said to be “coalesced” and is satisfied by issuing a single memory transaction. Otherwise, the set of requests is uncoalesced and needs to be serialized in different ways, depending on the access pattern, resulting in multiple transfers.

A trivial way to fetch the passphrases from global memory is by requesting one
Figure 2.20. Uncoalesced access of a half warp. The global memory is composed of elements of 16 bytes. Eight threads access $8 \times 16$ byte = 128 bytes. Sixteen threads (half warp) access $2 \times 128$ bytes or two segments.

character at time. However, given our data layout, with this access pattern only eight threads, out of sixteen, access the same segment (see Figure 2.20).

In order to measure the total number of memory transactions issued we profiled the code by using the CUDA profiler.

We built a small test code that runs on a grid composed by a single block of 32 threads. Each thread loops over all the 16 characters. On a device with compute capability 1.1, the profiler reports 512 uncoalesced accesses or, in other words, for each of the thirty-two threads, each kernel does a serial request each time ($32 \times 16 = 512$). A first way to lower this value is by fetching four characters at once instead of one. Although the memory access is coalesced, another important factor is the total number of requests made by each kernel. If we would request all the 16 characters at once, as a single vector of four unsigned integers, we could achieve a coalesced access and reduce the number of fetches to only four, resulting in an optimal memory transaction. We analyzed the resulting binary with decuda\(^3\), a third-party disassembler. From a CUDA compiled binary file, it produces the native instruction set instead of the abstract PTX [31]. We thus confirmed that the kernel actually issues a 128 bit ($16$ chars = $16 \times 8$ bit) request at once with a single instruction:

\[
\text{mov.b128 r0, global[r0]}
\]

This is the maximum size of a transaction that can be issued by a single instruction.

---

\(^3\)http://wiki.github.com/laanwj/decuda/

51
Table 2.3. GPU Version 3: coalesced global memory accesses.

transfers, we reached a speed up of sixty-three times over the original implementation and of $\sim 3\times$ over the previous kernel (see Table 2.3).

**Kernel Execution Optimizations**

The GPU threads can be organized in different layouts, allowing us to choose the best one according to the data parallelism of the problem. The layout is specified through a set of values that define the *grid*. Threads are organized in blocks. The number of threads per block is called *block size*, whereas the number of blocks defines the *grid size*. The total number of threads executing a kernel is equal to $\text{block size} \times \text{grid size}$. It is possible to shape both the block and the grid in different ways without changing the total number of running threads. So, a grid is fully specified by: block size, block shape, grid size and grid shape.

The *occupancy* value indicates the percentage, per multi-processor, of concurrently active threads with respect to the maximum possible number, that is 1536 on the latest generation NVIDIA cards.

The optimal size of the grid depends on both the number of multiprocessors available on the target device and the number of registers used by the kernel.

The NVIDIA documentation, suggests having at least 192 threads per block. Actually, we found, by running extensive tests, that for our kernels is better having 64 threads per block with a number of blocks equal to four times the number of multiprocessors available on the GPU.
Thread Branch Divergence

In the GPU Single Instruction Multiple Threads architecture, each thread has its own instruction address counter and register state. Therefore each thread is free to branch and execute independently.

The threads of a warp that execute different logic paths due to a data-dependent conditional branch, are said to be *divergent*. Branch divergence occurs only within a warp. In case of divergent threads, the warp executes each branch serially. The warp continues to execute concurrently only when all its threads converge back to the same execution path. Our kernel could be dramatically affected by branch divergence, for example when threads in the same warp process passphrases of different length. This could result in a different execution path for most of the threads, leading to a full serialization of all the threads inside the warp.

To investigate the impact of this phenomenon, we performed the following experiment. We generated five dictionaries: three with the same fixed passphrase length (8, 12, 16 characters) and two with passphrases of mixed length. Each dictionary contained one million passphrases. The mixed 1-16 dictionary contained a special passphrase arrangement that generated sixteen different branches on each warp: blocks of consecutive sixteen passphrases contained sixteen different passphrase lengths. We could not test for the full thirty-two different branches because of the (previously described) kernel restriction that limits the passphrase length to sixteen characters.

By varying the input data, the branch divergence metric changes (see second column of Table 2.4). As expected, the worst case is with the mixed 1-16 dictionary. Fixed length dictionaries allow a 11% performance increase.

With all the passphrases being of the same number of characters, all the threads in all the warps execute the same length-dependent code branch, resulting in no serialization. Regardless of the passphrase length, if it is fixed, the kernel speed will not be affected, whereas with passphrases of variable length the speed decreases (see last column of Table 2.4).
Table 2.4. Thread divergence with passphrases of different length and time required by our application for processing the corresponding dictionary layout.

<table>
<thead>
<tr>
<th>Passphrase length</th>
<th>Divergent branches</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed 8</td>
<td>14</td>
<td>3.49</td>
</tr>
<tr>
<td>Fixed 12</td>
<td>7</td>
<td>3.49</td>
</tr>
<tr>
<td>Fixed 16</td>
<td>10</td>
<td>3.49</td>
</tr>
<tr>
<td>Mixed 8,12,16</td>
<td>4473</td>
<td>3.60</td>
</tr>
<tr>
<td>Mixed 1-16</td>
<td>33485</td>
<td>3.93</td>
</tr>
</tbody>
</table>

Table 2.5. GPU Version 4: passphrases of fixed length.

<table>
<thead>
<tr>
<th>Version</th>
<th>Time (Seconds)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU single thread</td>
<td>505</td>
<td>1×</td>
</tr>
<tr>
<td>GPU v.1 loop unrolling</td>
<td>28</td>
<td>20×</td>
</tr>
<tr>
<td>GPU v.2 reduced buffer</td>
<td>24</td>
<td>22×</td>
</tr>
<tr>
<td>GPU v.3 coalesced accesses</td>
<td>8.15</td>
<td>63×</td>
</tr>
<tr>
<td>GPU v.4 fixed length input</td>
<td>7.53</td>
<td>67×</td>
</tr>
</tbody>
</table>
Code 2  Templatized Kernel Invocation.

// CUDA Code
template <int fixed_pass_len>
__global__ void gpu_kernel(...);

[...]

template <int fixed_pass_len>
switch(passphrase_length){
case 1:
  gpu_kernel<1><<<dimGrid, dimBlock>>>(args);
case 2:
  gpu_kernel<2><<<dimGrid, dimBlock>>>(args);
...
case 16:
  gpu_kernel<16><<<dimGrid, dimBlock>>>(args);
}

Fixed-length Password Optimizations

Kernels using dictionaries with fixed passphrase length produce a noticeable speedup (see Table 2.5). As a consequence, it looks reasonable to create and specialize kernels for each passphrase length. To easily create and maintain length-optimized kernels, we resorted to function templates, supported by CUDA to the full extent of the C++ standard [32]. Function templates are a form of meta-programming that allows writing a generic function in a data-type independent fashion.

The specialized kernel is called specifying the passphrase length in the template parameter (see Code 2).

Since in this implementation we fix the length of the input passphrase, it is unnecessary to store or compute the passphrase length. This eliminates the waste
of GPU memory caused by passphrases having different length. Moreover, with kernel templating the memory accesses can be optimized to fetch the exact amount of required data (Table 2.6).

Thanks to the meta-programming technique and by fetching a number of bytes exactly equal to the size of the passphrase, the code achieves an overall 83× speed up over the reference implementation (Table 2.7).

Finally, we may exploit the compile time knowledge of the total length of the SHA1 buffer (passphrase and salt). Since the string-to-key conversion phase behaves according to a fixed access pattern, there are cases where the access pattern can be predicted and pre-computed depending on the total buffer length. We further specialized the kernels for these cases by including the pre-computed access pattern optimization.

Furthermore, we considered the special situation when the total salt + passphrase length is sixteen bytes long. In that case the size of the circular buffer matches the total size of the 32-bit words processed for each SHA1 iteration. As a consequence, the indices of the sixteen input values read from the buffer never change, so instead of code 1 the more efficient code 3 can be used.

The access pattern pre-computation is quite effective. We reached a speedup of 100× over the reference CPU implementation (see Table 2.7).
2.6.5 Results

Through a process of continuous improvements and analysis we managed to dramatically improve our GPG attack. The most advanced GPU version performs 100 times better than the original reference implementation.

In the development process of the GPU kernels we found new ways to optimize parts of the attack. Since most of these optimizations were independent from the architecture, we back-ported them into the CPU code producing a new CPU version.

Running on a single Xeon E5462 (2.8 Ghz, 6 MB of cache) CPU, we can see from Figure 2.21, that the new version is much faster than the reference one. Most of this speed up is due to the SHA1 loop unroll and to the improved iterated and salted procedure that has a better access pattern to the cache memory.

From Figure 2.22 we can see how the application scales well compared to the number of multicores available on the GPU. By simply replacing the previous
<table>
<thead>
<tr>
<th>Version</th>
<th>Time (Seconds)</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU single thread</td>
<td>505</td>
<td>1×</td>
</tr>
<tr>
<td>GPU v.1 loop unrolling</td>
<td>28</td>
<td>20×</td>
</tr>
<tr>
<td>GPU v.2 reduced buffer</td>
<td>24</td>
<td>22×</td>
</tr>
<tr>
<td>GPU v.3 coalesced accesses</td>
<td>8.15</td>
<td>63×</td>
</tr>
<tr>
<td>GPU v.4 fixed length input</td>
<td>7.53</td>
<td>67×</td>
</tr>
<tr>
<td>GPU v.5 meta-programming</td>
<td>6.5</td>
<td>83×</td>
</tr>
<tr>
<td>GPU v.6 accesses pre-computation</td>
<td>5.0</td>
<td>101×</td>
</tr>
</tbody>
</table>

Table 2.7. GPU versions 5 (meta programming) and 6 (pre-computed access pattern).

generation device (Tesla C1060) with a new one (based on the Fermi architecture), we doubled the performances.

## 2.7 Discussion and future perspectives

In the first part of the chapter we presented an architecture to perform distributed dictionary attacks. The system has been tested on a private keyring of the GnuPG cryptosystem after a careful study of the features of the encryption system. In particular we devised a technique to quickly check candidate passphrases by limiting the execution of the most expensive control to a subset of the passphrases selected according to much less expensive controls. There are a number of possible directions for future activities. For instance it is possible to introduce new generation rules for the passphrases and increase their complexity. Moreover, it is possible to use the same approach to attack other cryptosystems.

We extended our work with the development of a GPU implementation of the computational core of our attack. In the last few years the majority of works on GPUs focused on their floating point performances. We showed that GPUs may represent a viable option for applications based on integer manipulation.
Graphic Processing Units are very effective in performing cryptographic attacks. By using our fast test GPU implementation, on a latest generation device (Fermi C2070), it is possible to attack the GnuPG secret ring at a rate of 500K passphrases per second.

Optimizing for the GPU can lead to optimizations that can be easily applied to CPUs as well.

To master this technology, the developer has to fully understand CUDA performance characteristics and be able to identify the hot spots of GPU kernels, like memory coalescing, divergent branches, memory bank conflicts and latency hiding. It also necessary to know how to identify which kind of bottleneck the application is suffering from (e.g., memory bandwidth, core computations, etc).

We can conclude that, comparing the time and effort with the speed up obtained, GPUs have a decisive advantage with respect to traditional CPUs.

This work furthermore discourages the use of simple dictionary passphrases, even in different languages or with simple transformations.

The next step regarding this work will be the integration of the GPU version of the attack with the distributed architecture code (implementing the CPU version).
Although we are still far from reaching the computational power required to execute a complete brute force attack, the new level of passphrases checked per second, with a single GPU device, is an alarm for the security community about the computational power that a network of relatively low budget systems can achieve.
Chapter 3

Multiscale hemodynamics model

Our approach to the study of the hemodynamics is based on efficient and accurate algorithms capable of handling the requirements of the diverse computational entities and associated scales. The numerical framework is handled by the software MUPHY developed by our group in recent years [10]. The approach is genuinely multiphysics, as it combines different levels of the description of matter, continuum hydrokinetic fluids for the dynamics of blood plasma and individual particles for the representation of red blood cells and other minority suspended species. The method is also multiscale, since fluid and particles are advanced concurrently and the exchange of information is computed on-the-fly. On general grounds, kinetic theory provides the conceptual framework to bridge micro and macroscales, and the Lattice Boltzmann (LB) method is extremely well suited for the numerical solution. LB is a minimal form of the kinetic Boltzmann equation, based on the collective dynamics of fictitious particles, representing a local ensemble of molecules moving on the nodes of a regular lattice [44]. The dynamics of such particles reproduces hydrodynamics in the continuum limit, when the molecular mean free path is much shorter than typical macroscopic scales. The LB method is a low-Mach, weakly-compressible fluid solver and presents several advantages for the practical implementation in complex geometries over Navier-Stokes solvers.
First, in the kinetic LB formalism, information always travels along straight
lines defined by a set of discrete speeds \( \{ c_p \} \). This represents a major advantage as
compared with the hydrodynamic representation, whereby any physical quantity,
including the flow field itself, is transported by the local flow velocity \( u(x,t) \), a
highly heterogeneous vector field, especially in the presence of complex real-world
geometries such as those involved in cardiovascular hemodynamics. The very same
feature also reflects into a much handier treatment of complex boundary condi-
tions (intersections with straight lines). This comes at the expense of numerical
accuracy, which usually degrades to first order, due to the staircase representation
of arbitrarily shaped boundaries. This weakness is, however, strongly mitigated by
two compensating effects: first, the shear stress is available \( \text{locally} \), as a linear com-
bination of the discrete populations sitting at each given cell. This relieves LB from
the burden of computing spatial derivatives of the velocity field at the boundaries,
which is an accuracy-threatenning procedure also in Navier-Stokes solvers. Second,
while it is true that staircase boundaries compromise second-order accuracy, it
is also true that a favorable prefactor (due again to straight trajectories) secures
significant error reduction by increasing resolution. In practice, wall shear stress
is found to converge to acceptable levels of accuracy at grid resolutions below 20
microns. Another favorable feature of LB is that, like the stress tensor, fluid pres-
sure is available locally, with no need of resorting to expensive Poisson solvers.
However, the most compelling asset of LB rests with its outstanding amenability
to parallel computing, \textit{even in complex geometries}. The work presented in chapter
6 represents a major testimonial to this asset: we prove excellent scalability on up
to \( \sim 300000 \) cores on a real-world complex geometry directly derived from medical
data.

In LB the basic quantity is \( f_p(x,t) \), representing the probability of finding, at
mesh location \( x \) and at time \( t \), a “fluid particle” traveling with discrete speed \( c_p \).
“Fluid particles” represent the collective motion of a group of physical particles
(often referred to as populations). We employ the common three-dimensional 19-
speed cubic lattice (D3Q19) with mesh spacing $\Delta x$, where the discrete velocities $c_p$ connect mesh points to first and second neighbors (Figure 3.1). The fluid populations are advanced in a timestep $\Delta t$ through the following evolution equation

$$f_p(x + c_p \Delta t, t + \Delta t) = f_p(x, t) - \omega \Delta t (f_p - f_p^{eq})(x, t) + \Delta f_p(x, t) \quad (3.1)$$

The right hand side of Eq. (3.1) represents the effect of fluid-fluid molecular collisions, through a relaxation towards a local equilibrium

$$f_p^{eq} = w_p \rho \left[ 1 + \frac{u \cdot c_p}{c_s^2} + \frac{uu : (c_p c_p - c_s^2 I)}{2c_s^4} \right]$$

that is a second-order expansion in the fluid velocity of a local Maxwellian with density $\rho$ and speed $u$. In addition, $c_s = 1/\sqrt{3}$ is the speed of sound, $w_p$ is a set of weights normalized to unity, and $I$ is the unit tensor in Cartesian space. The relaxation frequency $\omega$ controls the kinematic viscosity of the fluid, $\nu = c_s^2 \Delta t \left( \frac{1}{\omega} - \frac{1}{2} \right)$. The kinetic moments of the discrete populations $f_p$ provide the local mass density $\rho(x, t) = \sum_p f_p(x, t)$ and mass current $\rho u(x, t) = \sum_p c_p f_p(x, t)$. Finally, the last term in Eq. (3.1) represents the coupling between fluid and suspended bodies. This is given by

$$\Delta f_p(x, t) = -w_p \Delta t \left[ \frac{G \cdot c_p}{c_s^2} + \frac{(G \cdot c_p)(u \cdot c_p) - c_s^2 G \cdot u}{c_s^4} \right] \quad (3.2)$$
where $G$ is a forcing term containing the translational and rotational exchange of momentum induced by $N$ moving red blood cells at position $\{R\}$. The forcing term is smeared over a region made of 32 mesh points around each RBC having ellipsoidal shape. The drag force acting on particles is modelled as

$$F^D_i(R_i) = -\gamma_T(V_i - \tilde{u})$$  \hspace{1cm} (3.3)$$

and the torque is

$$T^D_i(R_i) = -\gamma_R(\Omega_i - \tilde{\Omega})$$  \hspace{1cm} (3.4)$$

with $\{V_i\}$ and $\{\Omega_i\}$ being the RBC velocities and angular velocities, and with $\tilde{u}$ and $\tilde{\Omega}$ the fluid velocity and vorticity fields, smeared over the same ellipsoidal region occupied by a RBC. This smearing is achieved through an envelope function similar to the one used in the Immersed Boundary method [45], which takes into account the finite extent of the particles by means of a smooth interaction. The constants $\gamma_T$ and $\gamma_R$ are translational and rotational coupling coefficients of RBCs represented as oblate ellipsoids in a hydrodynamic environment.

Here a compromise between physical fidelity and computational efficiency must be taken. Indeed, recent studies [46] indicate that the minimum number of degrees of freedom required for a quantitative description of RBC dynamics in a fluid flow, including deformability, is of the order of hundreds to thousands. This is far too much for a viable fluid-particle coupling at large-scales. As a result, an intermediate strategy has been developed, whereby RBCs are treated as rigid ellipsoidal bodies (six degrees of freedom) interacting with each other through custom potentials, and with the surrounding fluid (the blood plasma) via tensorial mobility coefficients, accounting for the anisotropic drag experienced by the RBC along and across the local fluid direction of motion. Such an intermediate strategy permits to capture the essential features of the complex behavior of the RBCs and their impact on the macroscopic blood rheology, at a very moderate computational cost. The represented behavior accounts not only for the translational and rotational motion of the RBCs, but also for their mutual interaction, reproducing aggregation.
patterns of RBCs and their impact on the overall behavior of the blood flow.

From an algorithmic point of view, this approach scales linearly with the number of RBCs, thanks to the fact that the solvent-mediated RBC-RBC interactions are entirely local and explicit. This stands in marked contrast with consolidated approaches, based on Brownian dynamics, which rely upon a non-local Green function representation of the Oseen tensor and consequently can only attain $N \log N$ scaling with the number of RBC’s by resorting to highly sophisticated procedures. The strategy described here makes therefore a particularly efficient use of the invested computational resources. In particular, as typical of LB applications, it provides a pretty simple algorithmic representation of fairly complex physical phenomena. In the present implementation, we neglect additional torques arising from coupling with the elongational component of the flow pattern and tank treading of the RBCs.

Mechanical hard core forces prevent contacts between RBCs. The RBC-RBC interactions are pairwise and modelled via the Gay-Berne (GB) potential [47], reading

$$u_{ij}^{GB}(q_{ij}) = 4\epsilon(q_{ij}) \times \left[ \left( \frac{\sigma_0}{R_{ij} - \sigma(q_{ij}) + \sigma_0} \right)^{12} - \left( \frac{\sigma_0}{R_{ij} - \sigma(q_{ij}) + \sigma_0} \right)^{6} \right]$$

where $q_{ij} \equiv (R_{ij}, \hat{u}_i, \hat{u}_j)$ and with $R_{ij}$ being the relative distance, $\hat{u}_i$ and $\hat{u}_j$ are the principal directions of the $i$-th and $j$-th ellipsoids, with $\epsilon(q_{ij})$ and $\sigma(q_{ij})$ being functions with lengthy expressions reported in ref. [47].

The potential $u_{ij}^{GB}$ is set to zero beyond a orientation-dependent cut-off given by the condition

$$\left( \frac{\sigma_0}{R_{ij} - \sigma(q_{ij}) + \sigma_0} \right)^{6} > 0$$

(3.6)

to retain the repulsive component of the potential only. The state of the suspended RBCs is advanced in time concurrently with the LB solver, that is, the same timestep $\Delta t$ is used for the LB fluid and the RBC dynamics. The rigid body dynamics of the suspended bodies is propagated in time via a second-order accurate
timestepping algorithm [48], properly modified to handle fluid-particle forces and torques.

In Appendix A are given details about the Computed Tomography Angiography (CTA) used to acquire patient data.
Chapter 4

Parallel Molecular Dynamics with Irregular Domain Decomposition

Molecular Dynamics (MD) is a very popular simulation method to study many-body systems by looking at the motion of the individual particles. In essence, MD tracks the motion of particles whose trajectories are the result of forces mutually exerted among them. The temporal propagation of the particles position obeys Newton’s equations of motion that can be solved by a time-discretization procedure of the differential equations followed by an integration in time [33, 34].

Since its inception, MD has benefited from several algorithmic advances that nowadays permit the simulation up to billions of particles with $O(N)$ complexity. Multiple techniques for Parallel Molecular Dynamics (PMD) have been put forward over the years. In particular, PMD has now reached a high degree of efficiency when dealing with regular geometries, that is, with bulk systems whose computational domain can be subdivided in terms of cubes, slabs, or other regular subdomains. Several implementations of PMD are freely available and run on both supercomputers, such as the IBM Blue Gene, and high-performance commodity hardware, such as clusters of Graphical Processing Units (GPUs). Two widely popular packages are NAMD[35] and LAMPPS[36] that are known to scale...
over thousands of processors. Moreover, we recall ACEMD, a production bio-
molecular dynamics software specially optimized to run on NVIDIA GPUs [37].
Finally, AMBER[38] is another molecular simulation program featuring NVIDIA
GPU acceleration support[39].

Recently, there has been growing interest in employing MD for multi-scale
simulations of particles suspended in a fluid. In the multi-scale framework, solute
particles are handled according to the conventional MD scheme whereas the solvent
is handled by means of conventional fluid dynamics solvers, such as the popular
Lattice Boltzmann (LB) method. The non-conventional aspect of the multi-scale
approach enters in the coupling between scales, an aspect that takes into account
the physical level of both the solute, the solvent and their mutual interaction. In
the case of the LB-MD multi-scale system, such design involves the kinetic level
to account for the microdynamics of the solvent. As a result, the LB-MD method
enjoys the same $O(N)$ complexity of stand-alone MD for systems with uniform
distribution of solute particles.

In the last few years, we have been devising and deploying multi-scale methods
to study the transport of molecular systems [10] and, more recently, the suspension
of red blood cells, an important topic in computational hemodynamics [40]. The
latter constitutes a strategic field since it allows to understand the physical behav-
ior of blood from a bottom-up standpoint, that is, by following the motion of red
blood cells and plasma. The wealth of information accessible from the multiscale
approach and the ensuing biomedical implications are beyond question.

When simulating large cardiovascular systems, the typical geometrical layout
consists of several interconnected blood vessels spreading in space with an irregular
pattern. Consequently, a parallel algorithm for both the LB and MD components
needs to account for the geometric sparsity of the vasculature. The optimal ap-
proach to parallelism is to decompose the computational space into subdomains
where the fluid and the particles are handled on the same footing. In this way,
the solution of the fluid-dynamic equations, the calculation of inter-particles and
fluid-particle interactions are mostly local on the processor responsible for the subdomain.

A possible approach to handle complex cardiovascular systems could be, by analogy with large scale stand-alone PMD in a simple regular box, to use a decomposition into box-shaped subdomains. Such approach is highly discouraged since it leads to poorly balanced subdomains, both in terms of number of active (from the fluid dynamics viewpoint) mesh points and average number of red blood cells. The end result would be a rapid degradation of performances on highly scalable architectures for both the LB and the MD components. The demand for load-balanced subdomains requires to decompose the highly irregular geometry by means of more sophisticated partitioning methods.

Several tools are available to achieve high-quality domain decompositions. In particular, we refer to tools that we customarily employ in our research, such as the METIS[42] and SCOTCH[41] packages. Both tools do not consider the geometrical shape of the domain, but rather employ the LB connectivity graph of the underlying mesh in order to construct a graph-based partitioning according to some heuristics (e.g., recursive multi-level bisection). The typical appearance of a domain partitioning is shown in Figure 4.1 for the case of an arterial system subdivided in a small number of domains (8 and 24).
In presence of highly-irregular domains, several critical issues arise related to the calculation of forces and migration of particles among subdomains. For instance, irregular subdomains imply irregular contact surfaces and, in principle, irregular communication patterns. The geometrical tests for particle ownership and exchange of particles among domains require strategic decisions that affect the efficiency of stand-alone MD as well as the LB-MD multi-scale method.

In this chapter, we describe a novel approach to PMD specially suitable for generic irregular subdomains that features the same $O(N)$ capability of PMD for regular decompositions. The proposed method relies on two basic notions, proximity and membership tests. These tests are used to discriminate particles according to their positions relative to the geometry of the domains. Proximity tests are used to select the particles that have out-of-domain interactions and are used to perform inter-domain forces computation. The membership tests regard the assignment of particles to domains and exploit a tracking method to associate particles position to the domains’ morphology. Using these tools, the different stages of the PMD algorithm can be derived. Even if our primary purpose is to define a computational strategy for hemodynamics, we underscore that the present treatment of irregular domains is usable for more microscopic systems. A typical scenario of sparsely distributed computational domain is the simulation of atomic or molecular systems confined in porous environments. Besides that, it is important to realize that our approach is completely independent from the partitioning strategy. As an example, other hemodynamic codes, as for instance HemeLB [43], even if adopting a different partitioning technique (a variant of the so-called Graph Grow Partitioning), could implement our technique to include the motion of red blood cells.

In this chapter we illustrate the general aspects of the PMD and the specifications of these aspects for our simulations of coupled LB-MD. We focus on simulations of particles without topological connectivity, as e.g., in the case of molecules. On the other hand, our treatment can be extended to molecules in
irregular domains without major modifications. In Chapter 5 the practical implementation of these notions for multi-CPU/multi-GPU clusters is described.

The chapter is organized as follows: section 4.1 describes the physical system abstraction that will be used as a reference in the discussion; section 4.2 describes the issues related to the parallelization of molecular dynamics and, finally, section 4.3 details our method to handle irregular domains.

### 4.1 Molecular dynamics

Molecular Dynamics is a general term that indicates numerical techniques for the simulation of a wide range of physical phenomena, from the atomic to the cellular scale, in which there are sets of interacting particles. On general grounds, we define a system of $N$ particles by using the set of particle positions $\{\vec{r}\}_{i=1,N}$ and forces $\{\vec{f}\}_{i=1,N}$ acting on them. Several physical parameters are associated to the particles, such as their masses, velocities and other quantities. For instance, in the case of red blood cells, particles are bodies having a moment of inertia, angular velocity, orientation that interact via mutual forces and torques.

The physical model is encoded by the interparticle pairwise forces that depend only on particle positions, such that the force acting on particle $i$ is given by $\vec{f}_i = \sum_{j \neq i} \vec{f}_{ij}$ where $\vec{f}_{ij} = \vec{f}_{ij}(r_{ij})$ and with $r_{ij}$ being the distance between particles $i$ and $j$. Hereafter we assume that forces are short-ranged, so there is a $r_{\text{max}}$, named cutoff, such that $\vec{f}_{ij} = 0$ for $r_{ij} > r_{\text{max}}$. Besides the interparticle interactions, an additional force avoids particles from crossing the arterial walls. This is achieved by treating the wall nodes as fictitious fixed particles, that act on the moving particles via a short-ranged repulsive interaction.

The simulation method consists in a step-by-step numerical solution of the classical equations of motion. At each step, forces acting on the particles are computed and the new state of the system is updated by integrating Newton’s law.
of motion. There are three main components in a MD program:

1. A model for the interaction among system constituents (atoms, molecules, etc.). It specifies the physical observables by which particles are represented and the force exercised between pairs.

2. An integrator, which propagates particle positions and velocities from time $t$ to $t + \delta t$. Usually, it is a finite difference scheme which moves trajectories discretely in time.

3. A statistical ensemble, where physical observables like pressure, temperature or the number of particles are controlled. This is used at each simulation step to verify that the system evolves in a correct way and, at times, to decide when the simulation ends (for example when a quantity reaches a given threshold).

These steps essentially define a MD simulation. From a computational viewpoint they specify the calculations executed at each step of the simulation to compute the forces acting on the particles, to advance their dynamic quantities in time and to decide when the simulation should end. Clearly, different phenomena may require a different implementation of these steps.

In the following, we discuss computational problems common to most MD implementations regardless of the details of the physics involved. We consider an abstract physical system that can represent the most common models used in MD simulations. The implementation of the forces and the integration step will be abstracted with generic function calls. Finally, the simulation is supposed to run as long as the system verifies a generic property $P$ which represents the statistical ensemble of finite size.

Algorithm 1 represents a generic MD serial pseudocode. This is a naive algorithm that does not contain any of the optimizations commonly used in MD codes but is useful in showing that the execution time is driven by the search of interacting particles pairs. In this trivial case where an all to all comparison
Algorithm 1 Naive MD Algorithm.

1: $R^t ← R^0$
2: $R^{t+1} ← ∅$
3: while not $P$ do
4:   for each $\vec{r}$ in $R^t$ do
5:     $\vec{F}_r ← 0$
6:     for each $\vec{s} \in R^t \setminus \{\vec{r}\}$ do
7:       if $d(\vec{r}, \vec{s}) ≤ r_{\text{max}}$ then
8:         $\vec{F}_r ← \vec{F}_r + \text{force}(\vec{r}, \vec{s})$
9:       end if
10:   end for
11:   $\vec{r} ← \text{newpos}(\vec{r}, \vec{F}_r, \Delta t)$
12:   $R^{t+1} ← R^{t+1} \cup \{\vec{r}\}$
13: end for
14: $R^t ← R^{t+1}$
15: $R^{t+1} ← ∅$
16: end while

is used, the execution time required is $O(N^2)$ making simulations of large scale systems prohibitively time consuming. Although algorithmic improvements are able to limit the time so that it scales, on the average case, as $O(N)$, with a high number of particles even highly optimized algorithms can be hardly used on single processor machines. In those cases the only viable approach is to resort to parallel processing.

4.2 Parallel Molecular Dynamics

The parallelization of Molecular Dynamics requires the partitioning of the simulation system into subdomains, each assigned to a different processor. Traditionally,
two different strategies are used, particle decomposition (PD) and domain decomposition (DD). In the first strategy, particles are assigned to processors according to an ordering index (given $N$ particles and $K$ processors, each processor receives $N/K$ particles, regardless of their position) whereas in the second strategy the assignment is based on the position of particles. In large scale systems DD provides a better solution since it ensures a high degree of local operations on each processor. The reason is apparent: particles assigned to a processor are spatially close to each other and thus most of pair interactions are intradomain. In the case of PD, on the other hand, spatial locality of particles is not guaranteed. In the case of DD, particle motion does not affect the average number of local interacting pairs while can have a strong impact in case of PD. Lastly, another advantage of DD is to keep local any operation between different physical methods coupled in the same simulation. This is the case of multi-scale simulations where particles interact with an underlying fluid (or solvent) whose dynamics is described, for instance, by means of the Lattice Boltzmann method. In the following we consider DD as the reference partitioning scheme and describe some critical issues that arise in presence of irregular subdomains along with our proposal to overcome them. The end result is that parallel MD simulations can run with high efficiency also in presence of generic partitioning.

The parallelization of a MD code, while allowing to scale the size of the system by using multiple processors, requires the solution of, at least, two problems related to interdomain pairs and particle migration.

Interdomain pairs are pairs of particles located in different subdomains (thus belonging to different processors) at distance smaller than the cutoff $r_{\text{max}}$. In order to compute interdomain forces, a processor requires to fetch information about all external particles that are close enough to interact with its own particles. Similarly, each processor must identify the set of particles in its subdomain that can interact with those belonging to neighboring processors, and exchange such set. We identify the sequence of operations required to handle interdomain pairs
as “frontier management”.

In a similar way, once particle positions are updated, particles that depart from a subdomain need to be sent to the processors in charge of the destination subdomains. This requires that processors identify their own departing particles and exchange them with neighbors. The sequence of operations that handle particle migration is identified as “migration management”.

We define the simulation system as $S = (D, R^0, P, r_{max})$ where $D$ is the spatial domain in which particles move, $R^0$ is the set of particles positions at time $t = 0$, $P = \{p_1, ..., p_n\}$ is a set of processors and $r_{max}$ is the cutoff distance for the forces. The parallel MD code can be represented as a couple of algorithms, $(A_{DEC}, A_{PMD})$. $A_{DEC}$ is a serial or parallel algorithm executed only once, before the simulation starts, to perform the domain decomposition and to identify the subdomains containing putative interacting particles. $A_{PMD}$ is a parallel algorithm executed by every processor to perform the MD simulation. $A_{DEC}$ takes as input the system $S$ and produces a decomposition:

$$\{(D_1, R^0_1, N_1), (D_2, R^0_2, N_2), ..., (D_n, R^0_n, N_n)\}$$

where $D_1, ..., D_n$ and $R^0_1, ..., R^0_n$ are partitions of the spatial domain $D$ and the initial positions $R^0$, respectively, and $N_1, ..., N_n$ is a subset of $P$. $D_i$ is the spatial subdomain assigned to processor $p_i$, $R^0_i$ is the set of particles with initial position inside $D_i$ and $N_i$ is the set of processors whose subdomains are at distance smaller than $r_{max}$ from any point of $D_i$.

In our LB-MD simulations the domain $D$ is identified by the underlying LB mesh used for the simulation of the solute surrounding the particles. The mesh has an irregular shape (embedded in a much larger bounding box) and typically represents multiple intersecting vessels in extended hemodynamics simulations.

Let us define a mesh by the set of cartesian points $\mathcal{M} \equiv \{n\} \subset \mathbb{N}^3$. The spatial domain $D$ in which particles move is the set of points at distance at most 0.5 (the mesh spacing step is conventionally chosen as 1) from a point in $\mathcal{M}$. What
Algorithm 2 Identification of neighboring subdomains within the cutoff distance.

Require: $r_{max}$

Require: $x_m, x_M, y_m, y_M, z_m, z_M$

1: $N_i \leftarrow \emptyset$
2: $\text{Gather}(x[], x_m - r_{max}, X[], x_M + r_{max})$
3: $\text{Gather}(y[], y_m - r_{max}, Y[], y_M + r_{max})$
4: $\text{Gather}(z[], z_m - r_{max}, Z[], z_M + r_{max})$
5: for $j = 1$ to $n$ do
6: if $j == i$ continue
7: if ($x_M < x[j]$) || ($x_m > X[j]$) continue
8: if ($y_M < y[j]$) || ($y_m > Y[j]$) continue
9: if ($z_M < z[j]$) || ($z_m > Z[j]$) continue
10: $N_i \leftarrow N_i \cup \{j\}$
11: end for

we formalized as $A_{DEC}$, is implemented as follows. In the first step, a graph partitioning tool (e.g., SCOTCH[41]) is run on the mesh $M$ in order to produce a partition $M_1, ..., M_n$. The set of particles is then divided by assigning to partition $i$ all particles in the subdomain identified by $M_i$ (more about that later in this section). As a final step, we need to identify the set $N_i$, a non-trivial task that in principle requires an awkward number of geometrical tests. The problem is that, with subdomains of arbitrary shape, the particles in two subdomains may be in interaction even if their surfaces are not in direct contact (for instance one can imagine two subdomains in 2D separated by a long thin stretch of a third subdomain, as shown in Figure 4.2). A possible solution, specific to our case, could be to leverage the granularity of each subdomain and compute the relative distance between cartesian mesh points belonging to all subdomains. These tests require to broadcast all mesh points among processors and perform operations with a computational cost of $O(n^2)$, with $n$ the number of mesh points.
However, we have devised a method to determine domains in interaction based on a reliable, yet simple criterion, that does not require such massive exchange of data among processors and can be performed in full generality. In Algorithm 2, we describe the operational procedure executed by processor \( p_i \) to construct the set \( N_i \) in the practical implementation of the method. We assume that every processor can compute the bounding box of its subdomain. This substep requires just a single scan of the mesh \( \mathcal{M}_i \).

At first, each processor initializes the set \( N_i \) and broadcasts to the other processors the bounding box of its domain augmented by the cutoff distance \( r_{\text{max}} \) (lines 1-4). As a result of the gathering function, the minimum and maximum coordinates of the boxes are stored in vectors indexed by the processor id, named with lowercase and uppercase letters, respectively. Now, processor \( p_i \) loops through the vectors to find the boxes that have a non-empty intersection with its own, non-augmented, box. The processor ids corresponding to the indices of the intersecting boxes are then added to the set \( N_i \). Although this procedure may generate a superset of the actual neighbors, we consider it an acceptable tradeoff given the limited computational effort it requires.

Once the initial stage of the domain decomposition and identification of interacting subdomains is completed, the parallel algorithm \( A_{PMD} \) is executed by every processor \( p_i \) on the set \( (D_i, R_i^0, N_i) \).
Algorithm 3 shows the pseudocode for $A_{PM}$ that implements the MD method on the subdomain $D_i$. The first half of the loop implements frontier management (lines 3-15). In the first part (lines 3-9) particles at a distance less than $r_{max}$ from the surface of $D_i$, $S(D_i)$, are selected and transmitted to the neighboring processors (line 9). The second part handles external particles received from neighbors (lines 10 to 15). For each received particle, its distance from $D_i$ is first checked (line 12). If it is greater than $r_{max}$ the particle does not interact with the subdomain and is discarded (line 13). After this check, set $E$ contains the received particles that putatively interact with the internal ones.
**Algorithm 3** Parallel MD algorithm, run by processor $p_i$.

**Require:** $r_{\text{max}}$, $(D_i, R^0_i, N_i)$

1: $t \leftarrow 0$
2: **while** not $P$ **do**
3: \hspace{1em} $S \leftarrow \emptyset$
4: \hspace{1em} **for** each $\vec{r}$ in $R^t_i$ **do**
5: \hspace{2em} **if** $d(\vec{r}, S(D_i)) \leq r_{\text{max}}$ **then**
6: \hspace{3em} $S \leftarrow S \cup \{\vec{r}\}$
7: \hspace{2em} **end if**
8: \hspace{1em} **end for**
9: \hspace{1em} $S \rightarrow \text{mcast}(N_i)$
10: \hspace{1em} $E \leftarrow \text{recv}(N_i)$
11: \hspace{1em} **for** each $\vec{r}$ in $E$ **do**
12: \hspace{2em} **if** $d(\vec{r}, S(D_i)) > r_{\text{max}}$ **then**
13: \hspace{3em} $E \leftarrow E \setminus \{\vec{r}\}$
14: \hspace{2em} **end if**
15: \hspace{1em} **end for**
16: \hspace{1em} $R^\text{UP}_i \leftarrow \text{ForcesAndIntegrate}(R^t_i, E)$
17: \hspace{1em} $S \leftarrow \emptyset$
18: \hspace{1em} **for** each $\vec{r}$ in $R^\text{UP}_i$ **do**
19: \hspace{2em} **if** $\vec{r} \notin D_i$ **then**
20: \hspace{3em} $R^\text{UP}_i \leftarrow R^\text{UP}_i \setminus \{\vec{r}\}$
21: \hspace{3em} $S \leftarrow S \cup \{\vec{r}\}$
22: \hspace{2em} **end if**
23: \hspace{1em} **end for**
24: \hspace{1em} $S \rightarrow \text{mcast}(N_i)$
25: \hspace{1em} $E \leftarrow \text{recv}(N_i)$
26: \hspace{1em} **for** each $\vec{r}$ in $E$ **do**
27: \hspace{2em} **if** $\vec{r} \in D_i$ **then**
28: \hspace{3em} $R^\text{UP}_i \leftarrow R^\text{UP}_i \cup \{\vec{r}\}$
29: \hspace{2em} **end if**
30: \hspace{1em} **end for**
31: \hspace{1em} $R^{t+1}_i \leftarrow R^\text{UP}_i$
32: \hspace{1em} $t \leftarrow t + 1$
33: **end while**
Computation of forces and integration of Newton’s equation of motion are performed by the function call `ForcesAndIntegrate(R^i_t, E)` that returns the set of updated particle positions $R^i_{UP}$ (line 16). Although the actual implementation depends on the particular physical model under consideration, particles within set $E$ should only be considered for their influence over internal particles, i.e., interaction pairs should be searched in $R^i_t \times (R^i_t \cup E)$. After the position of internal particles has been updated, particle migration is handled in the second half of the main loop (lines 17 to 32). All particles in the updated set $R^i_{UP}$ that moved outside $D_i$ are removed from the set (line 20), collected together (line 21) and then transmitted to the neighboring processors for possible admission (line 24). Lastly, particles received from neighboring processors (line 25) are scanned to identify new entries and each particle that moved inside $D_i$ is added to the set $R^i_{UP}$ (line 28). Finally $R^i_{UP}$ is assigned to $R^i_{t+1}$ and a new iteration starts.

The efficiency of PMD is strongly influenced by the frontier and migration management (lines 4-15 and 18-30) that can be measured looking at the number of executions of two basic tests on particles:

- Proximity test: $d(\tau^i, S(D_i)) \leq \tau_{max}$
- Membership test: $\tau^i \in D_i$

These tests, in turn, depend heavily on the representation of the domain and usually the efficiency of their implementation is tied to the “degree” of regularity of the spatial domain. For regular subdomains, like cubes or parallelepipeds, these tests can be easily implemented as computations of distances between points and planes, or, in case of axes-alignment, as simple differences. However, when there is no regularity, these tests can become time consuming and an effort to limit the number of executions is highly desirable. For example, algorithm 3 is quite inefficient since it performs the tests on all particles at every iteration.

When the domain is represented by a Cartesian mesh $\mathcal{M}$, the membership test
can be written as:

$$\vec{r} \in D_i \iff \text{round}(\vec{r}) \in M_i$$

where $\text{round}(\vec{r})$ represents the vector whose components are obtained by rounding to the nearest integer the real components of vector $\vec{r}$. At first sight, due to the irregularity of the mesh, this operation requires scanning all mesh points in $M_i$ with a linear search method. Actually, an efficient search can be performed by using a binary search algorithm or by a hash table. In our case, we store the mesh as a compact, one-dimensional array and sort the mesh points in ascending order according to a 1D-index equal to $k \ast nx \ast ny + j \ast nx + i$ where $nx, ny, nz$ define the bounding box and $i, j, k$ are the coordinates of the mesh points. The binary search has logarithmic cost so that the membership test is much more affordable from the computational viewpoint. For what concerns the proximity test, it can be viewed as another membership test run on the subset of $M_i$ that covers the frontier region of $D_i$.

Another crucial issue that influences the efficiency is the communication scheme used to exchange particles among processors in both the frontier and migration management steps. For these steps, there are two possibilities:

1. Multicast transmissions;
2. Point-to-point transmissions.

In the multicast scheme (used in algorithm 3) each processor sends groups of particles to all neighbor processors whereas, in the point-to-point scheme, a processor sends to each neighbor only its particles that interact with those in the domain of the receiver. From the communication viewpoint, the latter has the clear advantage of imposing much less overhead on the network since processors receive only data strictly required. Nevertheless, to implement such a communication scheme it is necessary for the processors to associate frontier and migrated particles to external domains. This requires multiple executions of proximity and membership
tests for each neighboring subdomain which, in turn, requires knowledge of the geometry of the neighbors, typically consisting of a region of size \( r_{\text{max}} \). This results in an increase of memory requirements per processor and, more importantly, in a greater number of tests performed per particle.

In the multicast approach, as shown in algorithm 3, processors do not need to know the geometry of neighboring subdomains but received particles must be processed to identify those that are of interest for the receiver.

To summarize, the differences between the two approaches are:

1. multicast: higher network burden, only-local geometry required, simple test implementation;

2. point-to-point: lower network burden, non-local geometry handling, complex test implementation.

These are general considerations and there may be exceptions. For instance, in the most favorable case of a regular Cartesian decomposition, each processor easily identifies the subdomains to which a given particle needs to be sent by using only its position and the global decomposition rules. For the regular case it is possible to combine the advantages of both approaches, point-to-point transmissions by using only local geometry and efficient tests implementations.

Algorithms 4 and 5 implement the frontier and migration management sections of algorithm 3 for the point-to-point solution.

In the following section we introduce a domain decomposition scheme that, applied to the subdomains, allows to substantially reduce the number of proximity and membership tests performed at each iteration.

### 4.3 Cell Tiling

We present now a general technique to reduce the number of proximity and membership tests performed at each iteration, that does not depend on the representa-
Algorithm 4  Frontier management algorithm according to the point-to-point model.
1: \( S_j \leftarrow \emptyset, \forall j \in N_i \)
2: for each \( \overrightarrow{r} \) in \( R_i^t \) do
3:     for each \( j \) in \( N_i \) do
4:         if \( d(\overrightarrow{r}, S(D_j)) \leq r_{max} \) then
5:             \( S_j \leftarrow S_j \cup \{ \overrightarrow{r} \} \)
6:         end if
7:     end for
8: end for
9: for each \( j \) in \( N_i \) do
10: \( S_j \rightarrow send(j) \)
11: end for
12: for each \( j \) in \( N_i \) do
13: \( E_j \leftarrow recv(j) \)
14: end for
15: \( E \leftarrow \bigcup_{j \in N_i} E_j \)

As anticipated, our simulations make use of subdomains of arbitrarily complex geometry and a decomposition algorithm that is executed only once, before the simulation starts. It produces subdomains that require a new formulation of the proximity tests. Due to the lack of regularity in the subdomains, we employ the multicast scheme for the communications and design a more general scheme for the proximity and membership tests.

The basic idea is to approximate the critical regions around the contact surfaces of the subdomains in such a way that is computationally simple to find a
Algorithm 5  Migration management algorithm according to the point-to-point model.

1: $S_j \leftarrow \emptyset$, $\forall j \in N_i$
2: for each $\mathbf{r}^i$ in $R_{i}^{UP}$ do
3:      if $\mathbf{r}^i \notin D_i$ then
4:         for each $j$ in $N_i$ do
5:             if $\mathbf{r}^i \in D_j$ then
6:                 $S_j \leftarrow S_j \cup \{\mathbf{r}^i\}$
7:                 $R_{i}^{UP} \leftarrow R_{i}^{UP} \setminus \{\mathbf{r}^i\}$
8:             end if
9:         end for
10:    end if
11: end for
12: for each $j$ in $N_i$ do
13:    $S_j \rightarrow send(j)$
14: end for
15: for each $j$ in $N_i$ do
16:    $E_j \leftarrow recv(j)$
17: end for
18: $R_{i}^{t+1} \leftarrow R_{i}^{UP} \cup \bigcup_{j \in N_i} E_j$

superset of the particles located inside those regions and to apply the tests only to those particles. This is possible by covering each subdomain with identical box-shaped cells. The cells tile a larger region as compared to the original subdomain extension. The tiling is computed by the algorithm $A_{LDEC}$ (figure 4.3) that processor $p$ runs on its subdomain $D_i$ to produce the tiling $(C, I_c, F_c, E_c)$ where $C = \{C_1, ..., C_k\}$ is the associated set of cells, $I_c \subseteq [k]$, $F_c \subseteq [k]$ and $E_c \subset [k]$ represent, respectively, the set of internal, frontier and external cells. Sets $I_c$, $F_c$ and $E_c$ form a partitioning of $C$. 

84
Figure 4.3. Initially the domain decomposition algorithm \( A_{DEC} \) runs to partition the domain in subdomains. Each processor then executes the tiling algorithm \( A_{LDEC} \) on its subdomain and, finally, the iterative phase of the simulation starts by running the parallel MD algorithm \( A_{PMD} \).

The tiling verifies the following properties:

1. Every point of \( D_i \) is within either an internal or a frontier cell.

2. Internal cells contain only points of \( D_i \) at distance greater than \( r_{\max} \) from the domain boundary.

3. Frontier cells contain all points of \( D_i \) at distance less than or equal to \( r_{\max} \) from the domain boundary.

4. External cells contain only points outside \( D_i \).

5. All external points at distance less than or equal to \( r_{\max} \) from the domain boundary lie within either an external or a frontier cell.

The tiling is built in the following way. The bounding box of \( D_i \), having size \( n_x \times n_y \times n_z \), is initially divided in \( m_x \times m_y \times m_z \) identical, regular cells, such that

\[
m_x = \left\lfloor \frac{n_x}{r_{\max}} \right\rfloor, \quad m_y = \left\lfloor \frac{n_y}{r_{\max}} \right\rfloor, \quad m_z = \left\lfloor \frac{n_z}{r_{\max}} \right\rfloor
\]
As a consequence, all cells have the same size $c_x \times c_y \times c_z$, that verifies the following conditions

$$c_x = \frac{n_x}{m_x} \geq r_{max} \quad c_y = \frac{n_y}{m_y} \geq r_{max} \quad c_z = \frac{n_z}{m_z} \geq r_{max}$$

As a result, the bounding box is divided in the maximum number of cells whose size is greater or equal to $r_{max}$ in each direction. The pool of cells is then augmented by including a layer of cells external to the bounding box (figure 4.4a) and a unique identifier is assigned to each cell. This pool of cells forms the initial set $C$ from which sets $I_c, F_c$ and $E_c$ are subsequently built. Initially, all cells containing at least one point of $D_i$ are selected (Fig. 4.4b). From those cells, sets
Figure 4.5. Example of the tiling of a subdomain taking into account periodic boundary conditions. In this case the selection of external cells must follow the periodicity.

$F_c$ and $I_c$ are built by assigning to $F_c$ the cells that contain at least one point at distance less than or equal to $r_{max}$ from the frontier of $D_i$, and to $I_c$ the remaining set (Fig. 4.4c). Finally, the unselected cells of $C$ are scanned, assigning to $E_c$ the cells with a neighbor in $F_c$. The remaining cells are then discarded from $C$ (Fig. 4.4d). There are a number of issues in the process of building $E_c$ that are worth mentioning. Choosing external cells based on the proximity of frontier cells may lead to picking cells that contain only external points at distance greater than $r_{max}$ from $D_i$. Nevertheless, this construction guarantees that every frontier cell has all 8 neighbors (in 2D) and 26 (in 3D), in the $C$ set. Moreover, when building $E_c$, the case of periodic boundary conditions needs to be properly managed while searching for neighboring frontier cells. In case of a periodic domain, for each periodic dimension, if the subdomain fully extends along that direction, then the searched neighborhood must account for the periodicity. Figure 4.5 shows an example of tiling of a subdomain with periodicity along the horizontal axis.
Algorithm 6 Frontier management using the subdomain tiling.

Require: \( N_i \)

Require: \( C, I_c, F_c, E_c \)

1: \( S \leftarrow \bigcup_{c,d \in F_c} C_{cid} \)
2: \( S \rightarrow \text{mcast}(N_i) \)
3: \( E \leftarrow \text{recv}(N_i) \)
4: for each \( \vec{r} \) in \( E \) do
5: \( cid \leftarrow \text{coo2cell}(\vec{r}) \)
6: if \( cid \in F_c \cup E_c \) then
7: \( C_{cid} \leftarrow C_{cid} \cup \{ \vec{r} \} \)
8: end if
9: end for

Algorithms 6 and 7 show pseudocodes for frontier and migration management that rely on the subdomain tiling.

We assume that algorithm 6 receives the particles binned into the cells. In the first two lines of algorithm 6 frontier particles are sent (in multicast) to neighboring processors. This is done by simply transmitting all particles inside frontier cells. External particles received from neighbors must be checked for putative interdomain interacting pairs. For each received particle the index of the containing cell is computed (line 5). If it identifies an external or a frontier cell (line 6) then the particle is inserted into the cell (line 7), otherwise it is ignored.

The proximity tests required to spot frontier particles that need to be transmitted are replaced by the selection of particles in frontier cells. Property 3) of the tiling procedure ensures that all internal particles potentially involved in interdomain interacting pairs are transmitted. Since frontier cells cover a superset of the internal frontier, some internal particles that could never interact with neighboring domains are also transmitted, resulting in an communication overhead that is expected to be limited. On the receiving side, proximity tests are performed by
checking that received particles are located inside external and frontier cells. For property 4), all external particles potentially involved in interdomain interacting pairs are retained. As in the previous case, since external and frontier cells together represent a superset of the external frontier, some external particles that do not interact with the subdomain are kept. This may result also in a limited computation overhead when searching external cells for interacting pairs.

There may be cases in which either the communication or the computation overhead becomes too high. However, the communication overhead can be eliminated by performing the proximity test on frontier particles before the transmission. As per the computation overhead, it can be avoided by performing the proximity test on the subset of the received particles that lie within external or frontier cells. In either cases the test is performed only on a limited superset of the internal and external particles that can be involved in interdomain interacting pairs.

After frontier particles have been exchanged, forces can be computed and particles position updated. The tiling allows to perform these operations by using a link-list algorithm [33]. The update can be done by scanning all internal cells and, for each particle in each cell, by searching for all interacting particles inside the 27 neighboring cells. The procedure is similar for frontier cells, with the difference that interacting pairs must be searched only for elements belonging to the domain. This distinction usually does not require membership tests. For example, if cells are implemented as lists, then it is sufficient to maintain two distinct lists for internal and external particles in each frontier cell.

After positions have been updated, algorithm 7 is executed to handle particles migration. We assume that the update algorithm returns the cells containing only the updated internal particles without the external ones (that can be safely discarded after forces have been computed). In the first part of the algorithm, particles that departed from $D_i$ are searched (lines 1 to 10). From property 5), it follows that all particles that moved to external cells have left $D_i$, so that they can be added to set $S$ and removed from the cells (lines 1 and 2). Within the frontier
Table 4.1. Number of particles transferred vs total number of particles per processor, for frontier and migration management by using the cell tiling. The system is an irregular domain $342 \times 228 \times 600$ with 500000 particles evenly distributed partitioned in 8, 16 and 24 subdomains. Data are averaged on 1000 iterations.

<table>
<thead>
<tr>
<th>tasks</th>
<th>sent particles</th>
<th>total particles</th>
<th>sent %</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3434</td>
<td>62500</td>
<td>5.4%</td>
</tr>
<tr>
<td>16</td>
<td>2708</td>
<td>31250</td>
<td>8.6%</td>
</tr>
<tr>
<td>24</td>
<td>2300</td>
<td>20832</td>
<td>10.9%</td>
</tr>
</tbody>
</table>

cells, particles that actually departed from $D_i$ must be found. This step requires the execution of the membership test on all particles inside frontier cells (lines 3 to 10). Clearly, for property 2), internal cells can be ignored. At this point, particles that left $D_i$ are sent in multicast to neighboring processors (line 11). In the second part of the algorithm, particles that left neighboring subdomains are received (line 12) and searched for new entries (lines 13 to 20). For each received particle, the index of the containing cell is first computed (line 14). If it is an external cell, then the particle is ignored because it can not belong to $D_i$. If it is a frontier cell then the membership test is performed (line 16). If it enters $D_i$, it is added to the cell (line 17), otherwise it is discarded.

With this procedure, we limit the communication overhead in frontier and migration management by having processors exchange a limited superset of the particles involved in those operations. To assess the advantage of our solution, we run a parallel simulation on an irregular domain and applied the tiling to the resulting subdomains. Figure 4.1 shows the domain partitioned in 8 and 24 subdomains by using the SCOTCH package. We measured the amount of data transferred to exchange particles during both frontier and migration management. In Table 4.1 we report the average number of particles sent by processors compared to the total number of particles in their subdomains. With 8 subdomains, processors exchange only 5.4% of the their data, resulting in a bandwidth saving of
Figure 4.6. Total time required to exchange particles in frontier and migration management by PMD vs time required to exchange every particle. Data are collected by running 1000 simulation steps with the same testcase table 4.1 refers to, with a number of processors ranging from 2 to 28. The inset contains the saving percentage plot.

∼ 95%. In our tests, the percentage of transferred data increases with the number of processors up to ∼ 10%, with 24 processors. Figure 4.6 shows a comparison of the total time required to exchange particles for both frontier and migration management between our method and the naive approach, based on the exchange of every particle. The performance gain of ∼ 90% is in line with the bandwidth saving results.

From the computational viewpoint, frontier management requires only table lookups that are performed on particles received by neighboring processors. For what concerns migration management, it requires both table lookups and mem-
**Table 4.2.** Average number of internal, frontier and external cells, per processor, with an irregular domain 342 × 228 × 600 partitioned in 8, 16 and 24 subdomain. The upper part contains the number of cells obtained with $r_{max} = 0.81$, corresponding to a much finer cell tiling, while the lower part is obtained with $r_{max} = 1.62$.

<table>
<thead>
<tr>
<th>tasks</th>
<th>internal</th>
<th>frontier</th>
<th>external</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>891270 (77%)</td>
<td>170947 (15%)</td>
<td>91140 (8%)</td>
</tr>
<tr>
<td>16</td>
<td>429835 (72%)</td>
<td>107198 (18%)</td>
<td>58060 (10%)</td>
</tr>
<tr>
<td>24</td>
<td>277584 (69%)</td>
<td>82246 (20%)</td>
<td>45054 (11%)</td>
</tr>
<tr>
<td>8</td>
<td>95613 (60%)</td>
<td>41297 (26%)</td>
<td>23498 (14%)</td>
</tr>
<tr>
<td>16</td>
<td>44103 (52%)</td>
<td>25659 (30%)</td>
<td>15072 (18%)</td>
</tr>
<tr>
<td>24</td>
<td>27316 (46%)</td>
<td>19485 (34%)</td>
<td>11719 (20%)</td>
</tr>
</tbody>
</table>

Membership tests. Before data exchange, particles to be sent are found by applying a membership test for each particle in frontier cells. After the exchange, a table lookup is performed on each particle received and, only to those located in a frontier cell, is applied the membership test. Table 4.2 shows the average number of cells resulting from the tiling procedure. We used the same domain as in Table 4.1, partitioned in 8, 16 and 24 subdomains, tiled with two different cutoffs. In both cases, the percentage of frontier cells that contains particles to which tests are applied, is quite stable with any number of processors.
**Algorithm 7** Migration management using the subdomain tiling.

**Require:** $D_i, N_i$

**Require:** $C, I_c, F_c, E_c$

1: $S \leftarrow \bigcup_{cid \in E_c} C_{cid}$
2: $C_{cid} \leftarrow \emptyset$, $\forall cid \in E_c$
3: for each $cid$ in $F_c$ do
4: for each $\overrightarrow{r}$ in $C_{cid}$ do
5: if $\overrightarrow{r} \notin D_i$ then
6: $S \leftarrow S \cup \{\overrightarrow{r}\}$
7: $C_{cid} \leftarrow C_{cid} \setminus \{\overrightarrow{r}\}$
8: end if
9: end for
10: end for
11: $S \rightarrow mcast(N_i)$
12: $E \leftarrow recv(N_i)$
13: for each $\overrightarrow{r}$ in $E$ do
14: $cid \leftarrow coo2cell(\overrightarrow{r})$
15: if $cid \in F_c$ then
16: if $\overrightarrow{r} \in D_i$ then
17: $C_{cid} \leftarrow C_{cid} \cup \{\overrightarrow{r}\}$
18: end if
19: end if
20: end for
Chapter 5

Multiscale Hemodynamics using GPU clusters

The behavior of blood in both capillaries and large coronary arteries has deep implications on the genesis of cardiovascular diseases such as atherosclerosis. Computational hemodynamics aims at studying flows in complex geometries, like those of blood vessels under stationary and pulsatile flow conditions. In the last few years, the study of hemodynamics has experienced an upsurge of activity due to the rapid advancement of methodological approaches and the availability of a steadily growing computing power, as also provided by high-performance commodity hardware, such as Graphics Processing Units (GPU). Blood is a complex fluid, composed of more than 99% in volume by two components, plasma and Red Blood Cells (RBC). Plasma is the solvent carrying simple Newtonian rheology, whereas RBCs play the role of basic building blocks, which are responsible for shear-thinning and viscoelastic behavior. To capture the essence of blood dynamics, in particular close to the vessel walls and to morphological irregularities of the vessels, like the atherosclerotic plaques, it is imperative to look at the composite RBC-plasma system in its entirety by including the corpuscular nature of blood and evolve it concurrently with the continuum plasma component.
For this reason, we adopt a multi-scale simulation approach that follows the two components on equal footing and in a concurrent fashion [6].

In our work, we leverage two distinct methods to handle plasma and RBCs and combine them in such a way to achieve a simple, yet effective, Janus-like representation of blood.

Lattice Boltzmann (LB) is an efficient computational method to describe plasma as a fluid in the continuum within an Eulerian framework [7]. LB is a grid-based method, that uses a cartesian mesh and exchanges information related to the fluid among first and second mesh neighbors through the motion of fictitious molecules hopping and interacting on the sites of a regular lattice. LB shows an excellent scalability on high-end parallel computers that makes it very suitable for the simulation of large-scale systems, such as the complete coronary arterial system.

Molecular Dynamics (MD) is the method that handles the motion of suspended bodies in the Lagrangian (grid-free) framework. RBCs are represented as anisotropic particles that move, tumble and collide among themselves. RBCs are active scalars for the plasma, that is, they are responsible for a two-way exchange momentum with the solvent. The coupling is spatially local, rendering the concurrent evolution of plasma and RBCs an optimal choice for a bottom-up approach to the study of hemodynamics.

Coronary arteries constitute a system of interconnected vessels, presenting a non-trivial morphology (see Figure 5.1), that surround the heart and carry oxygen to the heart muscle. The vessels are irregularly distributed in space and their layout calls for a highly sparse mesh to manage the active nodes only [40].

To reduce the time required for the simulation of the whole set of coronary arteries, it is mandatory to resort to parallel processing.

We are using a two-level approach. A classical domain decomposition scheme is used to partition the original domain (fluid and RBCs) among processors. This first, coarse grained level of parallelism is handled by a conventional message-passing library, such as MPI [49]. The solution on each of the subdomains is
obtained with a second fine grained level of parallelism using GPUs, distributing tasks in threads. The highly irregular shapes of the partitioned domains are obtained by specialized software packages, such as METIS [42] or SCOTCH [41], that produce quasi-optimal partitionings, both from the load-balancing and communication view points. The migration and force calculation of RBC across multiple irregular domains requires the definition of ad hoc algorithms for the MD modules of the hemodynamic solver. In the previous chapter, we described the issues emerging from parallel MD within irregular domains and proposed a general approach entailing a set of solutions. Hereafter, we present the related implementation issues, along with their solutions and the results obtained in selected test cases of specific hemodynamic relevance.

As already mentioned, with respect to high-end multicore architectures, GPUs may attain speed-ups that range between one and two orders of magnitude [50]. Currently, only CPUs are able to manage communication at MPI level, and the hybrid CPU/GPU computational paradigm may represent an optimal solution in terms of simplicity, flexibility and efficiency.

The end result is a code that provides excellent performances on CPU/GPU clusters, achieving quasi-ideal scalability on up to 32 GPUs.
This chapter is organized as follows: section 5.1 details the GPU implementation of the code and section 5.2 discusses its performance.

5.1 Implementation

The numerical framework is implemented within the MUPHY software [10], a code that we recently developed to run multi-scale fluid simulations of different kinds. The original MUPHY (MUlti PHYsics/multiscale) code is written in Fortran 90 and uses MPI for the parallelization. MUPHY makes use of an indirect addressing scheme that has been described along the other main features of the code in ref. [10].

In the parallel processing, the lattice representing the arteries is decomposed into subdomains. As the simulation starts, all mesh points and particles are distributed among processors so that each processor receives the subset corresponding to a subdomain. To maintain a high degree of data locality within each processor, it is necessary to use the same decomposition for both the LB and MD components. This strategy ensures that each processor handles the particles interacting with the fluid associated to the LB mesh assigned to it. An optimal load-balancing for the LB component cannot be achieved by using simple cartesian decompositions in such complex domain. However, a very satisfactory load-balancing is obtained by employing graph-partitioning tools like SCOTCH [41] or METIS [42] on a graph representing the connectivity of the irregular lattice. These graph-decomposition tools work on a geometry-free representation of the lattice, i.e., a representation lacking any geometrical information and may provide a (highly) irregular decomposition. An important aspect introduced in ref. [8] is that of cell tiling. In order to efficiently manage RBC-RBC interactions and particles migration across subdomains, the irregular shapes are tiled by space-filling parallelepipeds. The tiling allows to carry out two basic functions: first to track and select particles that fall
in proximity of the subdomain surface and, second, to minimize the handling and transfer of information across processors.

MUPHY has been originally developed for the IBM BlueGene systems [11]. More recently, its LB computational core has been ported to clusters of Graphics Processing Units (GPU), using the CUDA software environment, showing excellent results [51]. The porting of the MD core to CUDA allows to run hemodynamics simulations entirely on GPU clusters thus reducing significantly data traffic between CPUs and GPUs. In this case, the CPUs are only used to perform the domain decomposition and assist data transfer among GPUs.

At the early stage of development, we decided to design a new GPU version of the MD module instead of porting the existing CPU implementation. In fact, an existing code would have simply posed too many restrictions on the underlying data structures, whereas the new MD code is designed to exploit at its best the GPU capabilities. We thus developed a modular software architecture in such a way that new features are easily incorporated into the existing code.

Traditionally, one of the main issues related to the achievement of good performances on GPUs has been the requirement of having properly aligned accesses to the global memory (coalesced accesses in the CUDA jargon). However, devices with capability 1.3 and 2.0 (the capability defines the specific architecture of the GPU) can combine memory accesses by threads in a half-warp (that is a group of 16 threads) into a single memory transaction. This weaker notion of coalescence is such that memory access remains efficient as long as data lay in the same segment (that is a block, properly aligned, of 128 bytes), regardless of the memory access pattern. This feature highly mitigates the performance drop of uncoalesced accesses with respect to older devices, where such accesses were always serialized on a per-thread basis. Since, at the time of this writing, the new devices are widely available whereas devices with capability < 1.3 are disappearing, we limited our attention to fulfill the memory alignment requirements of the newer devices.
5.1.1 Domain Structures

Given the irregular shape of the spatial domains, the mesh can not be stored in \textit{full matrix} mode since this would imply a huge waste of memory (the bounding box of the domain can be as large as $10^4 \times 10^4 \times 10^4$ with only 3-5\% of the nodes actually used). Instead, the LB module relies on an indirect addressing scheme [51], storing only active nodes (\textit{fluid, wall, inlet} and \textit{outlet}). Mesh nodes are stored in one-dimensional arrays, one for each LB discrete velocity. Nodes of the same type (\textit{fluid, wall, etc.}) are contiguous in the arrays. The arrays are complemented by a matrix that represents the connectivity of the lattice. For each node, the connectivity matrix contains the array indices of the neighboring nodes. The number of entries in the connectivity matrix depends on the LB model in use. Since we use the D3Q19 scheme, there are 18 populations connecting neighboring nodes, and, for each node, 18 neighbors are indexed (one population relates to fluid at rest and does not require any connectivity information).

The LB indirect addressing scheme does not allow to carry out efficiently one of the most important operations required by MD, that is finding the array index of the grid points (each one indicated by a triplet of integers $(i, j, k)$) covered by a particle at position $(x, y, z)$. This operation is fundamental to implement the membership test that decides whether a particle is located inside a subdomain or not. To overcome this limitation, a new set of data structures has been devised. These data structures are of general use, in that they help in performing all MD operations that take into account spatial relations: particle-particle interactions, \textit{frontier} and \textit{migration management}. These structures are defined at the beginning of the simulation on the CPU, on the basis of the partitioning assigned to each processor and are subsequently transferred to the GPU memory, where they remain unmodified until the end of the simulation.

The first data structure, that we name \textit{COO2CELL}, is the \textit{cell matrix} that we use to map points in space to cell indices. In order to avoid the full matrix representation, the cell matrix is stored by using the same indirect addressing
scheme used to store the mesh. It is represented as two vectors: \texttt{COO2CELL[0][]},
containing the 1D-coordinate of internal, frontier and external cells, in ascending
order, and \texttt{COO2CELL[1][]}, that contains the tiling indices of the cells. Given a
point \((x, y, z)\), the index \(cid\) of the containing cell is found by first computing the
1D-coordinate of the cell

\[
1Dcoo \leftarrow \left\lfloor \frac{z}{c_z} \right\rfloor \cdot m_x \cdot m_y + \left\lfloor \frac{y}{c_y} \right\rfloor \cdot m_x + \left\lfloor \frac{x}{c_x} \right\rfloor
\]

where \((c_x, c_y, c_z)\) and \((m_x, m_y, m_z)\) represent the size of the cells and their number
along \(x, y\) and \(z\) directions, respectively. The 1D coordinate is then searched in
the first array by using a binary search

\[
j \leftarrow \text{binsearch} (\texttt{COO2CELL[0][]}, 1Dcoo)
\]

and, finally, the tiling id of the cell is obtained by accessing the \(j\)-th location of
the second vector

\[
cid \leftarrow \texttt{COO2CELL[1][j]}
\]

The second data structure is the neighbor matrix, called \texttt{CNEIGH}. This is a
two-dimension matrix that contains for every \textit{internal} and \textit{frontier} cell the ids of
the 26 neighboring cells (that can be \textit{internal}, \textit{frontier} and \textit{external} cells). This
matrix is not strictly necessary, since it is a mere “repackaging” of information
already present in \texttt{COO2CELL}. However, its usage simplifies the neighborhood
scan when performing particle-particle interactions by avoiding the binary searches
(see section 5.1.4).

The third and last structure is the map matrix, called \texttt{IJKMAP}, that defines
the association between particles and mesh data. It is another bi-dimensional
matrix containing, for every \textit{internal} and \textit{frontier} cell, the indices, in the array of
LB nodes, of the mesh points inside that cell. Points inside the cells that are not
part of the intradomain mesh are stored as \(-1\). Indices are stored in lexicographic
order, so that, given a mesh node \((i, j, k)\), it is possible to easily compute the
corresponding column index (see Figure 5.2 for a 2D example). The index, in the
Figure 5.2. Entries in the IJKMAP matrix for the cell $cid$; $idx_i$ is the index of the $i$-th mesh element in the array of nodes.

LB array, of a mesh node at position $(i, j, k)$ is:

$$nid \leftarrow IJKMAP[cid][lid]$$

where $cid$ is the index of the cell containing the node and $lid$ is defined by the linearized coordinates of the node in the cell frame of reference

$$
\begin{align*}
i_{\text{cell}} &\leftarrow i - \left\lfloor \frac{i}{c_x} \right\rfloor \cdot c_x \\
j_{\text{cell}} &\leftarrow j - \left\lfloor \frac{j}{c_y} \right\rfloor \cdot c_y \\
k_{\text{cell}} &\leftarrow k - \left\lfloor \frac{k}{c_z} \right\rfloor \cdot c_z \\
lid &\leftarrow \left\lceil c_x \right\rceil \cdot \left\lceil c_y \right\rceil \cdot k_{\text{cell}} + \left\lceil c_x \right\rceil \cdot j_{\text{cell}} + i_{\text{cell}}
\end{align*}
$$

By using these data structures, the membership test is implemented as follows. Given a particle $p = (x, y, z)$, its coordinates are first rounded to the nearest integer, to find the nearest grid point $g$. Then, it is checked whether an index in the mesh vector for point $g$ exists. If this is the case, the particle $p$ is located inside the domain.
Figure 5.3. Layout of the grid matrix. Internal particles are represented with circles and external ones with squares. Internal, frontier and external cells are colored, respectively, in green, yellow and red.

5.1.2 Particles Structures

Each particle is characterized by a set of properties, such as position, velocity, angular velocity, etc. Particles are thus stored as a set of arrays, one for each of these properties, according to the structure-of-arrays layout. This approach allows consecutive threads to access particles at consecutive memory addresses and it is part of GPU programming best-practices. The arrays contain both particles internal to the domain and the external ones at distance less than or equal to the cutoff distance from the domain boundary. Internal and external particles are separated: internal particles are located in the first part of the arrays, followed by the external ones. This layout allows threads in charge of internal particles to access contiguous memory addresses. Particles position inside cells is stored in a grid matrix, called GRIDMAT. It is a 2D matrix that contains, for each cell, the indices of the particles located in the cell. The number of particles varies from cell to cell, so an auxiliary array NGRID stores the number of particles in each cell. Figure 5.3 shows a simple example with 8 particles in a 2D grid.
5.1.3 Frontier Management

We now present the GPU implementation of frontier and migration management, as outlined in our paper [8].

At the beginning of each iteration, the set of particles located within frontier cells is moved from GPU to CPU memory in order to be exchanged among processors by using MPI primitives. Since particle arrays are unsorted, frontier particles are first gathered in a GPU buffer and then transferred to CPU memory via the cudaMemcpy function. The gathering is done by using a map array $V_{idx}$ that contains at position $i$ the index of the frontier particle that is copied to the $i$-th location of the buffer.

The map is built by invoking a sequence of kernels (i.e., functions running on the GPU) starting from the data stored in the array of cell ids, $V_{cell}$. The first kernel fills up a binary array $V_{mask}$, such that $V_{mask}[i] = 1$ if $V_{cell}[i]$ is a frontier cell, or 0 otherwise. Then, a parallel reduction kernel scans $V_{mask}$ to compute the number $n$ of frontier particles. If $n$ is non-zero, the parallel prefix sum of $V_{mask}$ is computed into a temporary array $V_{ps}$. Subsequently, the map array is created by setting $V_{idx}[V_{ps}[i]] = i$ for the entries $i$ such that $V_{mask}[i] = 1$. Finally, a kernel with $n$ threads is launched to gather data about the particles indexed by the first $n$ locations of $V_{idx}$ into the buffer. Figure 5.4 illustrates an example of this process.

![Figure 5.4](image-url)
Once particles are copied to the CPU memory, they are exchanged among processors by using MPI point-to-point primitives. On the receiving end, particles are copied from CPU memory to a GPU buffer, their cell id is computed and only those particles located within external or frontier cells are moved into the corresponding data structures. This task is carried out by a kernel that runs one thread per received particle. Each thread calculates the cell id of its particle and, in case of either a frontier or an external cell $c$, uses the `atomicInc` primitive to compute the index where the particle is stored in the destination arrays. The grid matrix is also updated by using the CUDA `atomicInc` primitive to atomically increment the cell counter and store the index $j$ at the corresponding column in the row $c$ of $GRIDMAT$.

### 5.1.4 Particle-Particle Interactions

Every processor updates the particles data structures by running a kernel that implements the force calculation. A sequence of additional kernels propagates other variables of the particles related to linear velocities, angular velocities, torques, derivatives, rotation matrices, cell ids, etc.

Particle-particle interactions are computed by using a GPU implementation of the link-cell algorithm. For each internal particle $i$, cells $CNEIGH[cid_i][0...26]$ are searched to identify interacting pairs of particles. Particles located in the neighboring cells are accessed by looking up the corresponding rows of the $GRIDMAT$ matrix. There are at least two ways to map data onto GPU threads for this task. A first possibility is to process pair interactions on a per-cell basis, by using the grid matrix. In this case, each thread block is assigned to a cell and each thread of the block computes the force acting on a particle inside the cell. This approach allows threads of the same block to cooperate while scanning the neighborhood of the cell. More in detail, since threads of the same block are in charge of particles in the same cell, the shared memory of the GPU is used to cache memory accesses to neighboring particles. This is done by having that each thread copies in shared
memory a subset of the cells and then having all threads that scan synchronously the shared copy for interacting pairs. However, this approach may result in a huge waste of resources, since cells typically contain a number of particles much lower than the GPU warp size (32 threads, corresponding to the basic scheduling unit). As a consequence, many threads in a warp would be idle, as there would be an insufficient number of particles for all of them.

We thus followed a different approach, whereby interactions are processed on a per-particle basis. In this case, the grid of threads is directly mapped onto the arrays of particles. Threads are assigned to particles according to their global id and the search for interacting pairs proceeds in an independent fashion. Each thread scans the cell neighbors and, for each interacting pair, it computes the contribution to the total force. Cooperation is not easily achieved because the indices of the particles inside the arrays are not related to their positions in space, so that consecutive threads may have to handle particles located in different cells.
Algorithm 8  Search for interacting particle pairs and computation of pairwise forces.

Require: CNEIGH, NGRID, GRIDMAT

Require: \( V_r \) is the array of particles positions.

Require: \( V_{cell} \) is the array of cell indices.

Require: \( V_{force} \) is the array of forces.

Require: \( N_{int} \) is the number of internal particles.

1: \( n \leftarrow gridDim.x \cdot blockDim.x \)
2: \( tid \leftarrow blockIdx.x \cdot blockDim.x + threadIdx.x \)
3: while \((tid < N_{int})\) do
4: \( \vec{f}_{tot} \leftarrow \vec{0} \)
5: \( \vec{r}_{tid} \leftarrow tex1Dfetch(V_r[tid]) \)
6: \( cid \leftarrow tex1Dfetch(V_{cell}[tid]) \)
7: for \( c = 0 \) to 26 do
8: \( ncell \leftarrow CNEIGH[cid][c] \)
9: \( N_{ncell} \leftarrow NGRID[ncell] \)
10: for \( j = 0 \) to \( N_{ncell} - 1 \) do
11: \( idx_j \leftarrow GRIDMAT[ncell][j] \)
12: \( \vec{r}_j \leftarrow tex1Dfetch(V_r[idx_j]) \)
13: if \((\vec{r}_j == \vec{r}_{tid})\) then
14: continue
15: end if
16: \( k \leftarrow |\vec{r}_{tid} - \vec{r}_j| \leq r_{max} ? 1 : 0 \)
17: \( \vec{f}_{tot} \leftarrow \vec{f}_{tot} + k \cdot force(\vec{r}_{tid}, \vec{r}_j) \)
18: end for
19: end for
20: \( V_{force}[tid] \leftarrow \vec{f}_{tot} \)
21: \( tid \leftarrow tid + n \)
22: end while
Algorithm 8 shows the pseudocode for this implementation. Lines 1 and 2 compute the size of the grid of threads to be executed and define the indexing scheme for particles. Each thread handles the particle corresponding to its linear global index inside the grid. Then, each thread loops over the internal particles. Lines 4, 5 and 6 initialize the force acting on particle $tid$, read its position and the index $cid$ of the cell containing it, respectively. Line 7 starts the loop through the 27 cells representing the neighbors of cell $cid$. The index of the current cell is read from $CNEIGH$ (line 8) and the number of particles located inside this cell is read from the $NGRID$ array (line 9). The innermost loop (line 10) runs over the particles located in the current neighboring cell. The index of the $j$-th particle is first read from the $GRIDMAT$ matrix (line 11) and its data fetched from the array of particle positions (line 12). In this sample pseudocode, we only consider particle positions, but in real practice, much more data need to be fetched from memory (orientation matrices, universal ids, etc.). The test at line 13 prevents the evaluation of the force between a particle and itself. Lines 16 and 17 compute the force between the current pair and add it to the total. Here we do not exploit the action-reaction principle $f_{ij} = -f_{ji}$ to avoid accumulating forces belonging to different threads. Doing so would require atomic operations for floating-point numbers that are not available on the Tesla line of GPUs.

Finally, after all neighboring cells have been scanned, the force computed for particle $tid$ is saved into the corresponding array and the thread slides to the next internal particle to be processed.

Particles data are read from global memory by using texture fetches via the $tex1Dfetch$ function to take advantage of the caching capability of the texture. This results in a performance increase of $5 - 10\%$ with respect to direct memory fetches. We also tried to sort the arrays of particles by the cell id in order to limit the cache miss rate and improve the coherence of memory accesses. However,

\footnote{The latest generation CUDA architecture, named Fermi, implements atomic operations for floating point numbers.}
numerical tests showed that the cost of keeping particles in sorted order (periodic sorting via the radix sort implementation, provided by the cudpp library [52]) is greater than the performance gain provided by the slightly higher cache hit rate, so we ended to use texture lookups on unsorted arrays. Threads executing this kernel may diverge in the innermost loop, in case of cells containing a different number of particles. Kernel profiling, however, shows that divergent warps have a negligible impact in this kernel.

5.1.5 Particle Migration

Once the update is completed, particle arrays and the grid matrix need to be synchronized, the reason being that the particles inside the arrays may have moved outside the subdomain or to different cells. Consequently, the indices stored in the grid matrix are no longer valid due to the reallocation among cells.

The arrays and the grid matrix are set back in a coherent state in the migration management phase. In this phase, particles that moved outside the domain are identified and exchanged with neighboring processors and newcomers are inserted in the data structures. The search for departing particles is done in a similar way to frontier management. A new map vector $V_{idx}$ is built, in order to permute updated particles data in a second buffer of vectors. In such way, internal particles are placed at the beginning of the arrays, followed by those that moved outside.

The map is computed by invoking a sequence of kernels starting from the data stored in the array of particle positions $V_r$. A first kernel computes a mask vector $V_{mask}$. Each thread applies the membership test to a particle and sets the corresponding location of $V_{mask}$ to 1, if the particle remains inside the domain, otherwise sets it equal to 0. Then, a parallel reduction kernel scans the mask vector to compute the number $n$ of the particles that stay in the domain. As for the frontier management, a prefix sum vector $V_{ps1}$ is computed to assign destination indices to internal particles. However, in this case, a second prefix sum vector $V_{ps0}$ is built on a copy of $V_{mask}$ by swapping ones and zeroes. This vector is necessary
Figure 5.5. Steps performed to compute the map array used to separate internal and external particles inside the second buffer of arrays. Outgoing particles are marked in grey (subsequent steps are top to bottom and affect the arrays described in the text).

to compute destination addresses for external particles that follow the internal ones in the second buffer of arrays. The permutation vector $V_{idx}$ is finally built by setting:

$$V_{idx}[V_{ps1}[i]] = i \quad \text{if} \quad V_{mask}[i] = 1$$

$$V_{idx}[V_{ps0}[i] + n] = i \quad \text{if} \quad V_{mask}[i] = 0$$

Figure 5.5 illustrates an example of this process.

The permutation vector is used to separate particles inside the second buffer, by copying at position $i$ the particle at position $V_{idx}[i]$ in the first buffer. External particles are then moved to CPU memory and the grid matrix is rebuilt with the data of the new $n$ particles. External particles are exchanged among processors via MPI calls and the received particles are moved to GPU memory to identify newcomers. The procedure is basically the same as for the frontier management. The membership test is applied to each particle and only those particles that, actually, moved inside the domain are inserted into the corresponding arrays and into the grid matrix.
5.2 Performance Tests

To evaluate the performances of our implementation, we ran a set of tests on a cluster equipped with 32 NVIDIA GPUs. The cluster has 16 computing nodes connected by Infiniband. Each node has an Intel Xeon Quad-Core E5520, equipped with 24 GB of RAM and connected to a pair of NVIDIA Tesla C1060 GPUs (capability 1.3), each one having 4 GB of global memory.

The testcase is part of the system (shown in Fig. 5.1) representing an artery bifurcation derived from real-life tomographic data for which we simulate fluid and particle dynamics. The ensemble of RBC consists of $5 \times 10^5$ particles, immersed in the LB solvent and with a mesh made of $\sim 6$ million active fluid nodes.
At first, we compared the GPU and CPU implementations of the code and measured the strong scaling of the system, by running the same testcase on an increasing number of GPUs, ranging from 1 to 32.

The speedup, measured on the total running time of the simulation (LB, MD and LB-MD coupling), obtained by the GPUs with respect to the CPUs, is shown in Figure 5.6. With any number of processors the GPU implementation is more than 14 times faster than the CPU reference code. The resulting curve is not smooth, due to the domain decomposition. Given the irregularity of the domain, the SCOTCH graph-partitioning tool produces partitionings with a balancing whose quality slightly varies with the number of subdomains.

Figure 5.7 shows a plot of the parallel efficiency of the complete simulation (in-
Figure 5.8. Time required to perform the main operations of MD and the coupling with the LB method vs number of GPUs. The inset contains a detail of the timings for 16, 24 and 32 GPUs.

including LB, MD and LB-MD coupling) for both GPU and CPU implementations. Both versions achieve an efficiency above 84% with any number of processors. However, while the CPU version almost immediately reaches the minimal efficiency, the GPU code runs with an efficiency greater than 96% up to 14 GPUs from where it begins to decrease down to 84%, with 32 GPUs. This effect is due to the relative small size of the test case. On the other hand significantly larger test cases would not fit in memory with few GPUs.

Figure 5.8 reports the timings of the simulation components including the coupling between LB and MD, and the breakdown in terms of computation of pairwise forces, frontier and migration management. To measure the kernels execution
times without the overhead of MPI transfers, frontier and migration management timings have been split into \textit{i}) the time required to gather and move data from GPU to CPU memory and \textit{ii}) the time required to exchange data among processors. It is apparent that the coupling between the LB and MD methods incurs the largest cost, regardless of the number of processors, whereas the computation of pairwise forces takes at most one third of the total time. The cost of frontier and migration management (both data gathering and MPI transfer) is negligible up to 8 GPUs.

The high cost of the LB-MD coupling is due to the large number of GPU global memory writes that cannot be combined together. As a matter of fact, each particle exerts a feedback on eighteen LB populations belonging to 32 nodes.
covered by the particle. Since the subdomains assigned to each GPU are rather irregular and the number of particles is much smaller than the number of mesh points, it is highly unlikely that the populations updated by consecutive particles lay within the same memory segment. For such reason almost every memory write is translated into a single memory transaction.

The timings breakdown of the CPU implementation is similar to that of the GPU code. It is worth noticing that in spite of the above mentioned issue for the LB-MD coupling, the GPU is almost one order of magnitude faster than the CPU in executing the coupling part of the code.

Regarding the performance of the MD component alone (without the LB-MD coupling), most of the execution time is taken by the computation of forces. With one processor, the GPU implementation is more than 20 times faster than the CPU one. The speed-up increases with the number of processors up to a factor of 34, achieved with 32 processors.

The time required by the gathering and transfer of data in frontier and migration management is negligible up to 16 processors. Starting with 24 GPUs, the time required by frontier and migration management becomes comparable with that of forces computation and coupling, given the reduced workload on each GPU.

Figure 5.9 shows a plot of the speed-up of the complete MD component with the LB-MD coupling activated. The speed-up is super linear with any number of GPUs.

Finally, we tested the code on a testcase representing the entire heart cardiovascular system depicted in Figure 5.1. The LB mesh represents the arteries with a resolution of 20 microns and consists of ~ 220M fluid nodes. For the MD component we used 10M particles. Given the memory requirements of this testcase, ~ 64GB for LB data and ~ 40GB for MD data, this run required the entire cluster at our disposal (32 GPUs). In Figure 5.10 it is shown the partitioning of the LB mesh produced by SCOTCH. We confronted the running times of the simulation
Figure 5.10. Partitioning of the coronary artery system produced by SCOTCH for 32 GPUs.

performed on the GPU cluster with those obtained by running the same testcase on 1024 processors of the BlueGene/P [11] architecture. For this test the LB mesh has been repartitioned in 1024 subdomains and the tiling procedure detailed in chapter 4 has been implemented in our BlueGene version of the MUPHY simulator. To complete 1000 simulation steps, 1024 nodes of the BlueGene/P take $9797.4 \text{s}$ whereas the Tesla cluster completes the same simulation in $559.2 \text{s}$. This means that, on this application, one GPU performs as $\sim 560$ nodes.
Chapter 6

Multiscale simulation of full heart-circulation system at near red-blood cell resolution

Accurate and reliable modeling of blood flows in the human cardiovascular system has the potential to improve understanding of cardiovascular diseases. But building a detailed, realistic model of hemodynamics is a formidable computational challenge. The simulation must combine the motion of the fluids, the intricate geometry of the blood vessels, continual changes in flow and pressure driven by the heartbeat, and finally the behavior of red and white blood cells and other suspended bodies, such as platelets and lipids.

Large-scale hemodynamic simulations have made substantial progress in recent years [53, 54, 55, 56, 57], but until now the coupling of fluid dynamics with the motion of blood cells and other suspended bodies in vessels with realistic shapes and sizes has remained beyond reach. Typically, coupled fluid-particle simulations are confined to microscale vessels, such as capillaries and venules, bearing no notion of the global geometry of the problem [58, 59, 60]. Yet the global geometry has significant effects on local circulation patterns, most notably the shear stress at
arterial walls. Wall shear stress is a recognized trigger for the complex biomechanical events that can lead to atherosclerotic pathologies. Currently it is not possible to measure this shear stress in vivo. The simulations we are going to present are a necessary precursor to the development of accurate and reliable hemodynamic simulations that can predict this critical parameter in the progression of cardiovascular disease.

In this chapter we present the first multiscale simulation of cardiovascular flows in realistic human arterial geometries derived from Computed Tomography Angiography (CTA) data. The simulation covers the entire heart circulation system, the network of arteries and arterioles that supply blood to the heart muscle, with a spatial resolution extending from 5 cm down to 10 µm.

The simulations involve up to a billion fluid nodes, embedded in a bounding space of about a three hundred billion voxels, with 10-300 million suspended bodies. They are performed with the multiphysics code MUPHY (MUlti PHYsics/multiscale), which couples Lattice Boltzmann methods for the fluid flow and a Molecular Dynamics treatment of the suspended bodies [10, 61]. The simulation achieves an aggregate performance in excess of 60 TeraFlops, with a parallel efficiency of more than 60 percent on a full BlueGene/P configuration with 294,912 processors.

Our work presents a number of unique features, both at the level of high-performance computing technology and in terms of physical/computational modeling. The extremely complicated conditions that are implicit to irregular geometries require that the workload be evenly distributed across the pool of as many as 294,912 computational nodes of the BlueGene/P supercomputer. The formidable graph-partitioning problem, even at the mere level of the fluid computation, cannot be overestimated. On top of this, our multiphysics/scale application adds the further constraint of keeping a good workload balance also across the Molecular Dynamics (MD) sector of the simulation. To the best of our knowledge, the latter issue has never been tackled before in any MD simulation. Indeed, even top-ranking (Gordon-Bell winning) multi-billion MD simulations are invariably
performed in idealized geometries, cubes or regular boxes [62]. Similarly, multi-billion node simulations of, say, biofluid turbulence are indeed available, but only in the same ideal geometries mentioned above. Complex and large-scale geometries, such as the one considered here, are rare in the literature [63]. By leveraging large-scale parallel architectures, this work demonstrates a simulation of cardiovascular flow of unprecedented scale in a geometry from real patient data and with blood flow at physiological hematocrit values.

In this chapter, we introduce our treatment of red blood cells as extended structure (i.e. not as point sources), our handling of highly irregular geometries via topology driven graph partitioning, and an efficient MD load balancing scheme.

6.1 Code Features

The original MUPHY (MUlti PHYsics/multiscale) code is written in Fortran 90 and uses MPI for the parallelization. To handle in a flexible and efficient way any complex geometry, MUPHY makes use of an indirect addressing scheme that has been described along with other main features of the code in [10]. In the same paper we showed that the penalty introduced by the indirect addressing scheme for the cases of regular geometries is very limited (∼5% of the execution time) and, as a matter of fact, we used the code also for problems, such as biopolymer translocation in which the geometry is trivial (a regular box). Originally developed for the IBM BlueGene/L system [11], MUPHY has been recently ported to heterogeneous clusters of CPUs and Graphics Processing Units (GPU), using the CUDA software environment, showing excellent results [51]. For the present work we employed the latest generation of the IBM Bluegene system (i.e., BG/P) whose main features may be summarized as follows:

- quad SMP processor chip per node with 2GB of memory per node.
- system-on-a-chip design with superscalar 850 MHz PowerPC 440 cores;
• a large number of cores (scalable up to at least 294,912);

• three-dimensional torus interconnect with auxiliary networks for global communications, I/O, and management;

• lightweight, Unix-like OS per node for minimum system overhead.

The first versions of the LB component of the code used the “fusion” of the collision and streaming steps in a single loop. This technique, by now standard in all high-performance LB codes, significantly reduces data traffic between main memory and cache/registers of the processor, since there is only one read and one store of all LB populations at each time step. However most implementations of the fused update resort to a “double buffer” to store the LB populations. The double buffer avoids the mixing of old and new data during the non-local streaming step that would be a source of inconsistency but, obviously, doubles the memory required for the LB populations. A recent enhancement to MUPHY is the implementation of the “single buffer” mechanism for the Lattice Boltzmann update through an adaptation of the so-called swap algorithm [64]. In this algorithm, the particle populations are rearranged after collision. This results in a memory layout in which, as a given population is copied to a neighboring cell during the streaming step, it simply exchanges its memory location with a neighbor-node population. Thus, the copy operations in the streaming step are replaced by exchange operations, or variable swaps, as suggested by the name of the algorithm. Given that no information is lost during the swap operation, the algorithm handles the streaming phase without the need for temporary buffers. This point becomes clear by looking at Figure 6.1, where four nodes of a one-dimensional Lattice Boltzmann mesh, with just two populations per node, perform a single global collision-streaming cycle, carrying them from a discrete time step \( t \) to \( t + \Delta t \). Right after the mesh populations have collided, the algorithm rearranges the data locally by exchanging the two populations (in a higher-dimensional case, all local populations are exchanged with the population corresponding to the opposite direction), as indicated by the
Figure 6.1. Schematic representation of a single collision-streaming cycle on four cells of a one-dimensional Lattice Boltzmann simulation with two populations per cell (detailed explanations are found in the text). The two populations on each cell are distinguished by the use of a solid line for the first and a dashed line for the second. The numbers next to the populations label the cell on which the populations were located at the initial time step $t$. Red denotes a population that has reached the post-collisional state.

keyword “swap” on the figure. Then, as soon as the neighboring node also reaches its post-collision state, a non-local exchange operation, indicated by the keyword “exchange”, is performed in lieu of the streaming step.

In a typical execution of a MUPHY program, memory is used mainly for the storage of the particle populations and the connectivity list. Thus the swap algorithm reduces the overall memory needs by one-third by avoiding a duplication of memory for the particle populations. Furthermore, this approach leads to a sensible performance improvement, because the program becomes more cache efficient as it holds the populations and connectivity matrix in a smaller memory space.

The geometry gathered from the CTA data that was used in the runs reported in section 6.2 is highly irregular, as shown in Figure 6.4, and its partitioning among the available processors represents a major challenge in itself. Several domain decomposition strategies for irregular lattices already exist, as described in [43]. In particular, state-of-the-art techniques like those represented by multilevel $k$-way
partitioning schemes can be used for irregular geometries. However, when either
the size of the mesh or the number of partitions increases to critical values (in our
case, the figures are $\sim 1$ billion nodes for the mesh and $\sim 300,000$ partitions) most
of the widely used tools simply fail, meaning that they are unable to manage the
problem (that is much worse, of course, than producing a sub-optimal solution).
For instance, the well-known tool for graph-partitioning $METIS$ [42], even in its
parallel version, requires the allocation on each processor of a block of memory
equal in size to the square of the number of partitions. On the other hand, pre-
liminary tests showed that a naive partitioning based only on a balanced number
of mesh nodes on each processor produced a very poor load balancing.

The graph-partitioning strategy is entirely topology-driven, i.e. it proceeds
based on the input provided by the local connectivity supplied by the Lattice
Boltzmann grid (18 neighbors, uniformly across the entire computational domain)
with no information on the global geometry of the problem.

Finally, we found a very effective solution by using $PT-SCOTCH$, the parallel
version of the $SCOTCH$ graph/mesh partitioning tool [41]. One of the interesting
features of $SCOTCH$ is that its running time is linear in the number of edges of the
source graph, and logarithmic in the number of vertices of the target graph for map-
ing computations. Moreover, a test carried out on a smaller case ($\sim 20,000,000$
mesh nodes partitioned among 1,024 processors) showed that $SCOTCH$ produces
a partitioning scheme superior to $METIS$, that is, with a better load balancing
taking into account both the number of mesh nodes per processor and the total
communication among the processors. Unfortunately, $PT-SCOTCH$ runs out of
memory on our real test case that produces a graph with almost one billion vertices
and $\sim 18$ billion edges. Our solution has been to use a pruned graph which repre-
sents the connectivity along the six main directions only ($+x, -x, +y, -y, +z, -z$).
This reduces the number of edges in the graph by 66% (by eliminating the 12 edges:
$+x + y, +x - y$, etc). We confirm that, in a smaller test case, the resulting partition
is, for all practical purposes, very similar to the partition produced for the whole

121
Figure 6.2. The distribution of the 294,912 tasks with respect to the number of tasks with which they are required to communicate.

It is interesting to look at the distribution of the tasks with respect to the number of other tasks with which they are required to exchange data, following the partitioning scheme produced by SCOTCH. The result is reported in Figure 6.2, which shows that, on average, each task exchanges data with other 15 tasks.

This workload distribution indicated that despite the highly complex geometry, the final workload partitioning ends up relatively close to the initial topological input, which we expect indeed as a heuristic measure of good balance. Visual inspection of the computational domains, shows that this is realized through a fairly sophisticated and highly varied morphology of the computational domains, often taking highly counterintuitive shapes in the vicinity of geometrical complexities. This “morphological richness”, which stands in stark contrast with elementary partitionings in idealized geometries (cubes, slabs and similar), conveys an intuitive flavor for the complexity of the partitioning task and also hints at some form of homeo-morphism between dynamics and geometry which surely deserves a separate investigation for the future.
We create the communication pattern by means of the following “run-time” pre-processing procedure. Mesh nodes are assigned to tasks according to the partition created as described above. Each mesh node is also labeled, in the input file, with a tag that identifies it as belonging to a specific subregion of the computational domain (e.g., fluid, wall, inlet, outlet). After the assignment of the nodes to the tasks, the pre-processing phase begins. Basically, each task asks which tasks own the nodes to be accessed during the subsequent phases of simulation, for instance for the streaming part of the LB algorithm and for the Molecular Dynamics. Such information is exchanged by using MPI collective communication primitives, so that each task knows the neighboring peers for send/receive operations. Information about the size of data to be sent/received is exchanged as well.

All point-to-point communication operations make use of the same scheme: the receive operations are always posted in advance by using corresponding non-blocking MPI primitives, then the send operations are carried out using either blocking or non-blocking primitives, depending on the parallel platform in use (unfortunately, as it is well known, few platforms allow real overlapping between communication and computation). Then, each task waits for the completion of its receive operations, using the MPI wait primitives. The latter operation, in the case of non-blocking send operations, is to wait for their completion. The choice between blocking and non-blocking send can be done at run time. The evaluation of global quantities (e.g., the momentum along the x, y, z directions) is carried out by using MPI collective reduction primitives.

Molecular Dynamics with a highly irregular domain decomposition is a major challenge in itself. In most parallel Molecular Dynamics applications the geometry of the spatial domain is a regular bounding box with Cartesian decompositions defined in such a way that each task has (approximately) the same number of particles and minimal communicating regions. In our case, this strategy would generate two separate domain decompositions: one for the LB (defined by the
graph-based partitioning method previously described) and another for the MD part of the simulation. As a consequence, the exchange of momentum between particles and fluid would become a non-local operation with a very high cost due to the long-range point-to-point communications imposed on the underlying hardware/software platform. For the IBM BlueGene such communications are explicitly discouraged. We decided to resort to a domain decomposition strategy where the MD parallel domains coincide with the decomposition of the LB mesh. In this way, each computational task performs both the LB and MD calculations and the interactions of the particles with the fluid are quasi-local.

The underlying LB mesh serves the purpose of identifying particles that belong to the domain via a test of membership: a particle with position \( \mathbf{R} \) belongs to the domain if the vector of nearest integers \( \text{round}(R_x), \text{round}(R_y), \text{round}(R_z) \) coincides with a mesh point of the domain. In addition, we exploit the load balancing of the mesh partitioning into the MD component, given that an even number of RBCs is expected to populate the domains. For the MD part of the code, a novel parallelization strategy suitable for the irregular geometry of the LB domains has been developed (see chapter 4). Our solution relies on the notion of cells, parallelepipeds with linear sizes greater or equal to the interaction cutoff, that cover the whole irregular domain. This representation allows the processors to perform an efficient search of both interdomain and intradomain pairs of particles and to reduce data transfers by exchanging a limited superset of the particles actually involved in interdomain pairs and particles moving across domains. The cells are grouped into three sets (internal, frontier and external cells) that verify the following properties:

1. Every point of the domain is within either an internal or a frontier cell;

2. Internal cells contain only points of the domain at distance greater than the cutoff distance from the domain boundary;

3. Frontier cells contain all the points of the domain at distance less than or
Figure 6.3. Decomposition of a 2D domain in external cells (red), frontier cells (yellow) and internal cells (green). The dashed line represents the region within a cutoff distance from the domain (solid line). The domain frontier has a staircase shape, but in this figure it is shown as a smooth curve for the sake of simplicity.

1. Equal to the cutoff distance from the domain boundary;

4. External cells contain only points outside the domain;

5. All external points at distance less than or equal to the cutoff distance from the domain boundary lie within either an external or a frontier cell.

Figure 6.3 shows an example of such decomposition applied to a simplified two-dimension domain.

The decomposition into cells helps in handling MD for irregular domains in the following way. At the beginning of each iteration, each processor searches for the particles inside its domain that could interact with particles located inside neighboring domains. Property 3 guarantees that the particles are only contained inside frontier cells. All particles located in the frontier cells are exchanged with neighboring processors so that only a limited superset of the particles that could interact with the outer region is transferred. On the receiving side, only the particles that could interact with the inner region are considered. Given property
Figure 6.4. The geometry of the 12.5µm resolution test case, derived from a CTA scan of human coronary arteries. The inset shows a detail of the geometry with red blood cells visible. Note: the red color in the inset is meant simply to highlight the presence of RBCs and is not an indicator of ESS. The Endothelial Shear Stress (ESS) is the field derived from the simulations that encodes the atherosclerotic risk map and is represented as a color map on the arterial walls.

5, the received particles to be retained lie inside either external or frontier cells. After particles involved in interdomain pairs are exchanged, forces can be computed and particles position updated.

Next, RBC migration among processors is handled. All particles are binned inside the cells they moved into, and those that move outside the domain are exchanged with neighboring processors. Departing particles are found by selecting those that moved to external cells (property 4) and frontier cells. To discriminate RBCs inside the frontier cells that moved to other domains, we make use of the underlying mesh by means of the membership test previously described. In this way, each processor sends exactly the particles that left its domain to all neighboring domains. On the receiving side, incoming particles are selected among the pool of all transferred ones.
The final component of our multiscale methodology involves the fluid-particle coupling. Each suspended RBC experiences hydrodynamic forces and torques arising from the fluid macroscopic velocity and vorticity, smeared over a domain made of $4 \times 4 \times 4$ mesh points. This non-local operation requires a communication step such that each processor owning a given particle exchanges the hydrodynamic quantities with the surrounding processors. The same type of information is exchanged to build the forces acting on the fluid and arising from the suspended RBCs.

6.2 Results

The performance of the Lattice Boltzmann component of MUPHY on a single core is in line with other LB kernels highly tuned for the Bluegene/P platform [65]. From this viewpoint, it is important to remember that: \(i\) the algorithm for the update of the LB populations has an unfavorable ratio between number of floating point operations and number of memory accesses; \(ii\) unlike other applications which can heavily draw upon consolidated computational kernels (e.g., matrix operations or FFTs), no optimized libraries are available to perform the basic LB operations; and \(iii\) it is not possible to exploit the SIMD-like operations of the PowerPC 440 processor, since those operations require stride-one access whereas the LB method has a “scattered” data access pattern due to the streaming phase.

We focus on the total runtime for the simulation, as well as on the breakdown between computation and communication. To this end, we ran a simulation at 12.5 $\mu$m resolution, corresponding to about 1 billion lattice sites for the fluid flow. The measurements were performed on the Jülich Bluegene/P with 294,912 cores, 144 TB of total memory and a theoretical peak performance of about 1 Petaflops. All runs were made in Virtual Node mode\(^1\).

\(^1\) In Virtual Node mode, the four cores of a Compute Node act as different processes. In this mode, an MPI application can use all of the cores in a node by quadrupling the number
With a mesh having 1 billion fluid nodes within a bounding box having a total of almost 300 billion nodes, our reference hematocrit level corresponds to 10 million RBCs, as we run on the 72 racks system. More recently, we obtained results utilizing the same mesh but with 100 and 300 million RBCs on the 40 rack Bluegene/P system (163840 core) at Argonne National Laboratory. These results provide a fundamental check of the reliability of the code at physiologic levels of hematocrit.

The successful completion of the simulations at each number of RBCs proves the feasibility and robustness of the method up to physiological levels. The joint usage of linkcell algorithms to compute pairwise interactions together with the linear method to access the indirect addresses of the mesh for the RBC-fluid exchange of hydrodynamic forces proved the linearity of the multiscale methodology with the problem size on the 40 racks configuration. Through the combination of the fine mesh and inclusion of red blood cells, we were able to observe the ESS in the real patient over the course of several hundred time steps as shown in Figure 6.4.

6.2.1 Strong Scaling

This scaling analysis is performed by increasing the number of processors at a fixed problem size, in an effort to analyze the impact of the number of computational cores on the total simulation time. In Table 6.2.1 and Figure 6.5, we show the elapsed time per time-step, as well as the breakup for the LB and MD components separately. A few comments are in order. First, the elapsed time decreases significantly with the number of cores, with a speed-up of 43.5 between the 4,096-versus 294,912-core configurations (see Figure 6.6), corresponding to a parallel efficiency in excess of 60%. This result is particularly significant given that the average number of mesh points per computational core becomes pretty low (i.e., of MPI tasks. The distinct MPI tasks that run on each of the four cores of a Compute Node communicate with each other transparently through direct memory access (DMA) on the node.
Figure 6.5. Log-log plot of the elapsed time for the LB component (circles), the MD component (squares) and for the full simulation (diamonds) versus the number of cores, for the system composed by 1 billion fluid nodes and 10 million RBCs.

Figure 6.6. Semilog plot of the speed-up for the LB (circles) and MD (squares) components, for the full simulation (diamonds), and for the ideal regime (dashed line) versus the number of cores. Data are for the system of 1 billion fluid nodes and 10 million RBCs.

∼ 3,300) on the full configuration of 294,912 cores. Particularly efficient are the data concerning the LB component, showing a speed-up of 54.1 and efficiency of 75%. Second, we notice that up to 147,456 cores the MD and LB sections remain in a fairly satisfactory balance with each other across the whole range of cores, thereby highlighting the excellent quality of the workload partitioning. Third, the MD component shows saturation above 147,456 cores. This is not unexpected, since at this number of cores and with 10 million RBCs each domain contains an average number of 60 RBCs, a critically small number regarding the calculations of Gay-Berne forces, the time-consuming MD component. Below the threshold of 147,456 cores, the constant ratio between the LB and MD workloads underscores the good response of the MD component in dealing with a handful of particles per domain. It is likely that, with a significant increase in the number of RBCs, the MD component would show a further speed-up up to 294,912 cores.
To further analyze the parallel performance of our simulation, we next inspected the breakdown of the communication time across the pool of cores. Details of the communication performance were obtained by using the MPI Profiler [66]. The communication times decrease significantly as the number of cores increases, roughly by 24% when going from 16,384 to 294,912 cores. The time for communication by the master core remains basically the same, with just a minor 5% increase. Table 6.2.1 reports the MPI function communication summary, where the Send/Irecv calls represent by and large the most time-consuming communication routines. The bandwidth for the (blocking) MPI_send corresponds to roughly 27 MB/sec and is satisfactory in view of the highly non-trivial communication pattern.

<table>
<thead>
<tr>
<th>Cores</th>
<th>LB</th>
<th>MD</th>
<th>LB+MD</th>
</tr>
</thead>
<tbody>
<tr>
<td>4,096</td>
<td>0.4761</td>
<td>0.04633</td>
<td>0.5224</td>
</tr>
<tr>
<td>16,384</td>
<td>0.1191</td>
<td>0.01610</td>
<td>0.1352</td>
</tr>
<tr>
<td>147,456</td>
<td>0.0151</td>
<td>0.00419</td>
<td>0.0193</td>
</tr>
<tr>
<td>294,912</td>
<td>0.0088</td>
<td>0.00419</td>
<td>0.0130</td>
</tr>
</tbody>
</table>

Table 6.1. Breakdown of the elapsed runtime

<table>
<thead>
<tr>
<th>MPI routine</th>
<th>Calls</th>
<th>Avg. bytes</th>
<th>Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>10251</td>
<td>1187.7</td>
<td>0.452</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>10302</td>
<td>17148.0</td>
<td>0.016</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>603</td>
<td>0.0</td>
<td>0.222</td>
</tr>
</tbody>
</table>

Table 6.2. Communication Breakdown for the run with 294,912 cores.
6.2.2 Hardware Performance Monitoring

Finally, we conducted a performance analysis by using the hardware performance monitoring library (HPM) on BlueGene/P. HPM tracks 256 performance counters that measure events ranging from integer and floating-point operations to cache and memory accesses. The hardware counters can be set to measure the performance for either cores 0 and 1 or cores 2 and 3 during a single execution. The tool reports Flops as a weighted sum of various floating-point operations. More information is given in [66]. For 72 racks of BlueGene/P in VN mode (294,912 cores), we measured 64 TeraFlops, as shown in Figure 6.7. In view of the intrinsic Flop-limitations of the Lattice Boltzmann algorithm discussed previously, and taking into account the coupling between LB and MD components, this appears to be a fairly satisfactory overall performance. Just to convey the flavor of the practical impact of this application, the above performance corresponds to simulating a full heartbeat at microsecond resolution in only a few hours time on the 72-rack BlueGene/P system.

Figure 6.7. Aggregate performance (Floating Point Operations Per Second) as a function of the number of cores.
Chapter 7

Conclusions and future directions

In the first part of this dissertation we presented a distributed architecture to perform dictionary attacks to cryptosystems and the GPGPU implementation of the attack. The system has applied to the decryption of the GnuPG secret keyring for which we devised an ad-hoc technique to limit the computational workload required to reject wrong passphrases.

Afterwards, in order to exploit the data parallelism of the passphrase verification phase, we implemented the core of the attack on Graphics Processing Units whose architecture is perfectly suited to such tasks. The resulting, highly optimized and tuned GPU code can verify 500000 passphrases per second on a single, latest generation, GPU. This result shows that modern GPUs, widely recognized as excellent floating point accelerators, are also competitive computing platforms for applications that need to execute integer arithmetics operations.

The next step regarding this work will be the integration of the GPU version of the attack in the existing distributed architecture. We aim at a hybrid computing scheme where both CPU and GPU resources spread across wide area networks, like the Internet, could be exploited for the attack. If the GPU technology will continue to scale according to the Moore’s Law, and its recent evolution path seems to confirm this trend, in less than five years we can expect graphic processors at
least four times faster than the current ones. This roughly means that with a relatively modest cluster of a hundred GPUs it will be possible to test \textit{trillions} of passwords/passphrases within the time span of hours.

In the second part of this dissertation we presented our work on hemodynamics. We described a novel method to perform Parallel Molecular Dynamics in arbitrarily complex geometries. To the best of our knowledge, this is the first effort to define a general method to perform parallel Molecular Dynamics in presence of irregular domains. The end result is a scheme that features \(O(N)\) complexity together with a general way to handle particle migration and computation of forces among irregular subdomains. This method represents a fundamental tool to study those biological and physical systems for which the boundary of the spatial domain confining the phenomena directly influences the outcome of the simulation. We employed our novel PMD method in the GPU implementation of a Molecular Dynamics module for the multiscale simulator MUPHY, one of the first examples of high performance parallel code for the simulation of multi-physics/scale bio-fluidic phenomena in realistically complex geometries. This multi-GPU module has been written from scratch using CUDA for GPU programming and the OpenMPI library for the Multi-GPU parallelization. Coupled with the pre-existing CUDA module for Lattice Boltzmann simulations, the resulting code represents the first effort to design and implement a general method to concurrently perform GPU-accelerated Lattice Boltzmann and Molecular Dynamics simulations inside irregular domains.

The code is totally independent from the specific geometry in use for what concerns both computation and memory usage. This means that in case of irregular spatial domains, the distribution of the computational workload among the processors can always be performed according to the most suitable strategy without geometric constraints. Furthermore, the code accomplishes an extremely efficient use of memory by storing only the “active” regions of the domains.

The resulting performances are very encouraging. The flexibility and robust-
ness of the system allows to parallelize the simulations on any number of processors with close to linear performance scaling. Compared to the CPU implementation, the GPU code delivers a speedup in excess of one order of magnitude, in line with the speedup of other non-trivial codes ported to GPUs. This result is even more important considering that, because of the different physical methods in use, certain aspects of the code could not be optimized. This emphasizes the effectiveness of hardware optimizations introduced in the latest generations of NVIDIA GPUs with respect to previous generation cards, like the relaxation of the conditions required to coalesce memory accesses, introduced with the Tesla line of processors, or the cache hierarchies present in the Fermi architecture.

Finally, our PMD method has been implemented on the BlueGene version of MUPHY. With this code we run the first large scale simulation ever of the entire heart-circulation cardiovascular system, with a realistic representation of the complex human arterial geometry at the spatial resolution of red-blood cells: from centimeters all the way down to microns in a single multiscale simulation. This simulation, involving one-billion fluid nodes, embedded in a bounding space of three hundred billion voxels and coupled with the concurrent motion of ten million red-blood cells, achieves over 60 Teraflops performance on the full 294,912 Blue-Gene/P processor configuration, with a parallel efficiency in excess of 60 percent, performing about 100 billion lattice updates per seconds. Using the same arterial system and simulation parameters it has been possible to elevate the hematocrit level to physiological levels of 300 million RBCs.

The above achievement results from the development of several unique features, in terms of both high-performance computing technology and of physical/computational modeling, namely i) the solution of the formidable graph-partitioning problem prompted by the need of evenly distributing the workload associated with the complex arterial geometry, across as many as 294,912 Blue-Gene/P cores; ii) the innovative communication techniques required to secure a balanced workload between fluid-dynamics and Molecular Dynamics in geometries
of real-life complexity; and iii) the innovative modeling techniques required to manage the self-consistent fluid-particle interactions in complex geometries. As to (ii), it represents the major contribute of the present thesis work and we are not aware of any previous solution dealing with non-ideal geometries.

Currently, no combination of computational models, however sophisticated, can provide a comprehensive and all-embracing description of all complex phenomena which underlie the dynamical behaviour of the entire human cardiovascular system, including a realistic description of the arterial tissues, wall compliance, RBCs deformability, to name but a few. However, this does not prevent the possibility to gain completely new insights on specific cardiovascular phenomena of major clinical relevance. For instance, as far as long-term atherogenesis is concerned, neither wall compliance, nor RBCs deformability, are credited for playing a lead role. On the other hand, even in large arteries, the finite extent of the RBCs, is likely to exert a major effect on the near-wall circulation patterns, hence the local wall shear stress distribution. This is the kind of effect that the present simulations are expected to shed new light on, once appropriate hardware resources are available.

This work represents a major progress in the predictive capabilities of computer simulations for real-life cardiovascular applications. The scientific and societal impact of the extensions of such activity cannot be underestimated.

This result is a further evidence of the reliability of our PMD method, also on architectures completely different from GPUs. The performances obtained on both multi-GPU platforms and the BlueGene architecture show that the method imposes a very limited overhead on both the computing nodes and the communication network.

In the immediate future we aim at performing simulations of the entire cardiovascular system on GPU powered clusters. With the GPU code presented in this dissertation we were able to simulate the full coronary system with a resolution of 20 microns and 10^7 red blood cells on a cluster equipped with 32 GPUs. To
match the level of detail of the 10-microns/300M-particles testcase we estimate \( \sim 128 \) Tesla GPUs are needed.

To conclude, it’s just a matter of resources availability.

The works described in this dissertation have been published in the following papers:


M. Bisson, M. Bernaschi and S. Melchionna, “Parallel Molecular Dynamics with Irregular Domain Decomposition”, accepted for Comm. in Comp. Phys.


Appendix A

Geometry preparation and mesh generation

We next outline the procedure to build the LB mesh starting from the MDCT raw data for a typical coronary artery system, as illustrated in Figure A.1. Briefly, the procedure for the acquisition of the MDCT data is the following. Patients are imaged axially using a 320 × 0.5 mm detector CT scanner (Toshiba AquilionOne Dynamic Volume CT, Tochigi-ken, Japan) with 350 millisecond gantry rotation time. The geometry for a single vessel, reconstructed from the MDCT acquisition, is initially evaluated using a research build post-processing workstation (Vitrea 4.0, Vital Images, Minnetonka, MN, USA), undergoes low-pass filtering, and is formatted as stacked bi-dimensional contours (DICOM slices), with a nominal resolution of 0.1 mm. The DICOM slices are over-sampled along the axial direction up to a slice-slice distance of 0.02 mm to allow further post-processing.

The contour path of each slice is irregular and shaped as a collection of 256 points. The slices are quasi-parallel and mostly transverse to the path connecting different centerlines. Moreover, the sequence of contour points is mostly aligned along the stacking sequence, that is, the contour index does not present any major twisting when moving between contiguous centerlines. Data relative to each vessel
in a multi-branched geometry have the same format. Finally, the geometrical data
do not carry any topological information about branching, but bifurcations occur
where the contours and centerlines of different vessels overlap in space.

The MDCT data formatting is rather general but presents some complications
at the stage of the LB mesh creation for a ramified system of vessels. The com-
plications arise from the limited resolution of the MDCT scans that often exhibits
mismatches at the vessel attachment and bifurcation regions. Therefore, the pre-
processing phase must be considered with care and aided by three-dimensional
visualization tools. Starting from the raw MDCT data, the mesh generation pro-
ceeds through three distinct stages:

i) Raw MDCT data present a mild level of geometric irregularities that can
affect the quality of the LB simulations. We regularize the initial geometry by
slightly over-sampling the slices by a factor 2 – 3. Subsequently, the contour
points are smoothed by using a linear filter along the axial direction (see Figure
A.2).

ii) Each slice is scanned to locate the set of mesh points that fall in its proximity
and enclosed within the vessel surface. For this purpose, for the \( s \)-th slice we
consider the triangle having for vertices the slice centerline, the \( c \) and \( (c + 1) \)
points of the contour. Given the corresponding triangle lying on the \( (s + 1) \)-th
slice and vertices coinciding with the slice centerline and \( c \) and \( (c + 1) \) contour
points, we consider the prism enclosed between the two triangles and locate the
mesh point enclosed within the prism (see Figure A.2). The enclosed mesh points
are located by subdividing the prism into three adjacent tetrahedra and finding
the mesh points falling inside each of the three tetrahedra. Each tetrahedron has
vertices \( p_1, p_2, p_3 \) and \( p_4 \) and, for each putative mesh point \( m \), we consider the
**Figure A.1.** Left coronary system imaged with 320-detector MDCT, where automatically segmented coronary arteries and branches with lumen diameter greater than 1 mm are illustrated. The smaller pictures illustrate two slices corresponding to the sections indicated by the green and blue dashed lines.

determinants of the following $4 \times 4$ matrices

\[
\begin{align*}
    d_0 &= \det(\tilde{p}_1 \tilde{p}_2 \tilde{p}_3 \tilde{p}_4) \\
    d_1 &= \det(\tilde{m} \tilde{p}_2 \tilde{p}_3 \tilde{p}_4) \\
    d_2 &= \det(\tilde{p}_1 \tilde{m} \tilde{p}_3 \tilde{p}_4) \\
    d_3 &= \det(\tilde{p}_1 \tilde{p}_2 \tilde{m} \tilde{p}_4) \\
    d_4 &= \det(\tilde{p}_1 \tilde{p}_2 \tilde{p}_3 \tilde{m})
\end{align*}
\]

where $\tilde{p}_\alpha = (p_{\alpha,x}, p_{\alpha,y}, p_{\alpha,z}, 1)$ is a four-component vector constructed by the Cartesian component of a tetrahedron vertex complemented by unity, with $\alpha = 1, \ldots, 4$, and $\tilde{m} = (m_x, m_y, m_z, 1)$. The geometric criterion for $m$ to belong to a tetrahedron is that $d_0$, $d_1$, $d_2$, $d_3$ and $d_4$ have the same sign. In this case, $m$ is labelled as a fluid node if it falls between two generic slices, and is labelled as an inlet or outlet node if it falls between the initial or final slices of a vessel (and for slices not belonging to a bifurcation point).
iii) The 18 mesh points surrounding each fluid/inlet/outlet mesh node according to the LB connectivity are inspected in order to locate wall nodes. If any of the 18 surrounding points is yet unlabelled we tag it as a wall node. This phase needs to exclude multiple counting of wall nodes, since several fluid/inlet/outlet points can be connected to the same wall node. In our indirect addressing scheme, this is achieved by considering an ordered sequence of connectivity.

Each of these stages is designed to create a workflow with minimal bottlenecks and high efficiency. The mesh search algorithm is the most time consuming part and we take advantage of the fact that the elementary regions to be scanned are tetrahedra. Thus, for each tetrahedron the search algorithm runs over the mesh points caged by the simplex, filtering out a large number of un-needed operations. Overall, the search algorithm scales linearly with the number of slices. Moreover, the algorithm is parallelized over the number of slices, by distributing the slices in a balanced manner across processors. As a result, this pre-processing stage can be completed in a few minutes on commodity parallel hardware.

An important remark regards the way we choose the sampling points that will be subsequently used to compute the ESS, that is, the points $\vec{x}_w$ in close proximity to the mesh wall nodes. The sampling points are chosen to coincide with the smoothed, off-lattice points forming the slice contours. Each contour point is surrounded by a small number of fluid nodes and a linear extrapolation scheme is used to evaluate the ESS. Clearly, the set of $\vec{x}_w$ points falls between the external fluid and wall mesh nodes and, as the mesh resolution increases, the set converges towards the exact no-slip hydrodynamic surface.

Finally, the volumetric quantities, such as the volumetric flow rates, are computed by using triangular quadratures over the slices. The triangles lie on a slice plane and have vertices $\vec{v}_1$, $\vec{v}_2$ and $\vec{v}_3$ corresponding to the slice centerline $o$ and the $c$ and $(c+1)$ contour points, respectively. We consider four-point quadratures, with the triangular integrals approximated as $\sum_{k=1,4} w_k F(\vec{r}_k)$, where $F(\vec{r}_k)$ are hydrodynamic quantities tri-linearly interpolated from the caging 8 mesh nodes, and
Figure A.2. Three-dimensional stack of slices over a segment of an artery from MDCT data (left panel) and after the regularization procedure (middle panel). The insets show the geometry of three consecutive slices. The right panel shows the prism enclosed by two contiguous slices (solid lines), with vertices \((s, o)\) (centerline), \((s, c)\) and \((s, c + 1)\) (contour points) on slice \(s\), and vertices \((s + 1, o)\), \((s + 1, c)\) and \((s + 1, c + 1)\) on slice \(s + 1\). The prism is further subdivided into three tetrahedra enclosed between solid and dashed lines and obtained by considering all possible combinations of four vertices, for the mesh point searching method.

the quadrature nodes are given by the combinations \(\vec{r}_k = \sum_{i=1}^{3} t_k^i \vec{v}^i\). The quadrature coefficients and weights are \(t_1^i = \frac{1}{3}(1,1,1)\) and \(w_1 = 0.28125\), \(t_2^i = \frac{1}{15}(11,2,2)\), \(t_3^i = \frac{1}{15}(2,11,2)\), \(t_4^i = \frac{1}{15}(2,2,11)\) and \(w_2 = w_3 = w_4 = 0.26041666\), with \(i = 1, 2, 3\).

Tests have shown that four-point quadratures are accurate enough as compared to more involved seven-point or higher-order quadratures.
Bibliography


Medussa: http://www.bastard.net/kos/medussa/medussa.html


Apache portable runtime (apr). http://apr.apache.org/


NIST, Advanced encryption standard (aes), Federal Information Processing Standards publication, (2001)


[63] LB simulations with coupled bodies, even deformable ones, are available in the literature, see for instance M. Dupin et al, Phys. Rev. E, 75, 6, (2007), but we are not aware of any massively parallel implementation of this method to tackle full heart-circulation problems. Parallel LBM applications to complex geometries have been performed by L. Axner et al, J. Comp. Phys., 227, 4895, (2008), but on much smaller sizes (order of ten million cells) and with no suspended particles. Computational fluid dynamics with multi-body dynamics are also available, see J. Gotz et al, Parallel Computing, 36, 142, (2010), but on a much smaller number of nodes (8192). Larger Lattice Boltzmann simulations, with up to 150
billions cells and 260 million suspended particles, have been recently presented (J. Gotz et al, Supercomputing 2010). However, these simulations only deal with idealized cartesian geometries.

