Integrated frameworks for distributed computing

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea Politehnica Timişoara
în domeniul Calculatoare şi tehnologia informatiei
de către

ing. Răzvan-Mihai ACIU

Conducător științific:	prof.univ.dr.ing. Horia CIOCÂRLIE
Referenţi ştiinţifici:	prof.univ.dr.ing. Rodica POTOLEA
	prof.univ.dr.ing. Viorel NEGRU
	prof.univ.dr.ing. Vladimir-Ioan CREȚU

România, 300159 Timişoara, Bd. Republicii 9,
Tel./fax 0256 403823
e-mail: editura@edipol.upt.ro

Foreword

This PhD. thesis was developed during my activity at the Politehnica University of Timisoara, Faculty of Automatics and Computers. I consider teaching an important, noble work and I am deeply grateful for the opportunities I have here both for teaching and for researching.

Special thanks are due to my PhD. coordinator, Prof. Univ. Dr. Eng. Horia Ciocârlie for his guidance and important advices since the beginning of my work. He is a very good teacher and adviser and during the years I learned many things from his great life experience, both from a professional and a personal point of view.

My PhD. guiding commission composed of Prof. Univ. Dr. Eng. Ionel Jian, Lect. Dr. Eng. Dan Cosma and Lect. Dr. Eng. Ciprian-Bogdan Chirila was of an invaluable help. They offered me many advices, suggestions, corrections and ideas which were very important for the quality and consistency of my research.

I greatly appreciated the reviews and suggestions from my PhD. commission, composed of Prof. Univ. Dr. Eng. Rodica Potolea, Prof. Univ. Dr. Eng. Viorel Negru and Prof. Univ. Dr. Eng. Vladimir Crețu. They considerably helped me to shape the final form of the thesis.

I also wish to express thanks to my family, close friends and God, who helped and supported me during this important period of my life.

Table of contents

Table of figures

Table of tables

1. Introduction

The Distributed Computing is the Information Technology (IT) domain which researches and implements solutions applicable for distributed systems, as they are defined in [1]: "a distributed system is a collection of independent computers that appear to the users of the system as a single computer". Regarding independent computers, we try to use all their computing resources, such as their CPU cores and Graphic Processing Units (GPUs). In this way a greater computational power is used and it becomes possible to deal with large amounts of data.

In our days, as the computing requirements are more and more demanding, Distributed Computing started to be used more and more often. Associated with the fact that after a certain performance level the hardware becomes very expensive, the alternative to associate many computing resources to solve in parallel the same task is very attractive and economically feasible.

The actual microprocessors have an upper frequency range around 4-5 GHz and 8-16 cores. Top commodity PC motherboards support around 8 microprocessors. Above these values special technologies are needed, for example liquid cooling for CPU. From these values, if the computing requirements are much higher, the only solution is to use multiple computers or supercomputers formed by using many interconnected processing units.

Among the domains or applications types which require a large amount of processing power, we can enumerate the fallowing ones:

- **graphic rendering** – for complex scenes, with many objects and effects, for which it is required as much photorealism as possible, the computation involved can be very large. For example, the 3D animated movie "The Croods", required from its producer DreamWorks around 80 million compute hours to render and 250TB data storage capacity to make [2]. It can be easily seen that this amount of computation is well beyond the possibilities of a single computer. For such tasks, DreamWorks uses computing resources providers like Cerelink. In 2010, the cloud provided by Cerelink had a theoretical peak speed of 172 teraflops (peak theoretical speed) from its Altix ICE 8200 cluster, with 133 teraflops sustained operation. The ICE 8200 consists of 1,792 nodes (14,336 cores) of quad Xeon 3.0 GHz processors housed in 28 racks [3].

- **biochemical simulation** – when the simulation needs to achieve a great precision, it requires taking into account many chemical, biological and physical laws. To integrate all these, the computations are very expansive. Maybe the most well known project in this area is **folding@home**, which uses a volunteer network [4] to compute folding data for proteins. In September 2013, there were over 266,000 computers involved in this project [5]. An interesting aspect of this project is the fact that even if the computations are very demanding, the core algorithms used for them are quite simple. This allowed their implementation as Graphics Processing Units (GPU) kernels.

- **earth sciences, such as weather prediction and seismology** – these sciences use very complex systems with a lot of variables interacting each other according to computationally intensive formulas. To provide real-time predictions, a huge amount of processing power is needed. For example, the US National Center for Atmospheric Research (NCAR) uses several datacenters and one of them, located at Wyoming Supercomputing Center, houses the IBM Yellowstone supercomputer, with a processing power of 1.5 petaflop [6].

The **folding@home** project is only one of the many projects using the same model. There are also some other notable projects using the same model [7], as listed in Table 1.1.

Project	**Domain**	**Users**	**Hosts**
LHC@home SixTrack	Accelerator Physics based at CERN	115,937	301,786
SETI@home	Search for Extraterrestrial Intelligence	1,425,304	3,476,481
ABC@home	Finding abc-triples for the ABC conjecture	77,414	143,561
Climate Prediction	Climate simulation and prediction	269,829	555,184

Table 1.1 – Distributed computing projects

Regarding the access speed to its components and its management facilities, a distributed infrastructure can be classified into the following categories [8][9]:

- **massively parallel processing systems (MPP)** – tightly coupled computers. Most of the supercomputers are MPP. Because of the involved hardware and interconnections, in many cases these can be seen as a single multiprocessor computer.

- **clusters, clouds, computers/servers farms** – independent computers usually in the same space, sharing only a high-speed network connection, usually without other shared or bridging hardware which can be seen in MPP. The computers within a cloud can be remotely managed and they possibly have virtualization software [10][11].

- **grids** – independent computers located in different geographical areas, communicating through network [10][12]. These computers can have very different hardware, operating systems and software, usually without virtualization or remote management software. WAN networks and the Internet are part of this category.

When clouds are targeted, the fact that their installed operating system and software can be easily controlled and tailored to specific needs eases considerably the development of such applications. In the same time abstraction frameworks such as Apache Hadoop can be installed and configured to automate many distributed computing tasks.

The most complex task is to address computer grids, because of their heterogeneous nature and because the online time of the computers in most cases cannot be predicted [13]. If platforms like Java or .NET are not used to create an abstract level over the hardware, operating system and the installed software, the programmer will need to write application variants for specific operating systems and available hardware. In the same time, the network management and error handling must be more thoroughly implemented in order to take into account frequent situations such as computers joining or leaving the grid. Another issue which needs to be addressed when programming for grid is the fact that the available computers can have very different computation capabilities and the tasks scheduler must find a way to use all these computers in an optimal way.

In the context of the above classification, heterogeneous resources means different computing resources, such as CPU, remote computers and GPU, different operating systems and different hardware. Using Java frameworks the operating systems and hardware can be abstracted and also projects such as folding@home already use GPU for computations, but only in a limited way, for numerical intensive algorithms.

Another possible classification regarding the physical computing resources takes into account their computation capabilities. In this case, the targets can be:

- **systems with independent microprocessors, capable to run any algorithm** – this is the case of the computer systems. Any algorithm can be implemented for them and this kind of resources can be used to execute the most complex algorithms involving complex data structures, recurrence and dynamic memory allocation.

- **systems with dependent computing units, tailored for specific tasks** – this is the case of the **General Purpose Graphics Processing Units** (GPGPU or GPU in this work), which can have thousands of computing units but with more restrictions than the microprocessors. These systems can ensure a very high computation throughput, much higher than using microprocessors, but for now only some tasks are suitable for them [14]. Because of their computation power, the GPUs started to be used in many supercomputer architectures, such as the Titan supercomputer, made from 18,688 CPUs paired with an equal number of GPUs. It was able to achieve 17.59 petaflops, being the most powerful supercomputer in November 2012 [15].

The above examples are large and very large scale applications. There are many other cases, at a lower scale, where distributed computing can improve a process, especially by saving a lot of time. For example a firm with 10 computers mainly used for office work (so the CPUs are loaded under 5-10%), can create with them a distributed system for the cases when intensive computations are needed on one computer. In this case the workload can be deployed as background tasks with lower priority (in order not to interfere with the normal usage) on the other computers and it can be computed in a distributed manner. Depending on the required network traffic and the parallel structure of the application, its computation time can be greatly improved, without disturbing or putting on hold other activities. This leads to a much more efficient usage of the existent resources and in some cases can even allow undertaking tasks which would not be feasible without distributed processing.

Taking into account the above considerations, it is obvious that for computation or data intensive applications, distributed computing is a big advantage and it is very important for them to have such capabilities. Unfortunately, the actual tools (programming languages, libraries, protocols) are not easy to use by programmers to implement reliable and powerful distributed computing support in applications [16]. The mainstream programming languages have only a limited support for distributed computing and in most cases to write such an application involves using a lot of different libraries and configuration tools. The debugging process is also harder for a distributed application. The processes of an application can be implemented in a sequential or in a parallel way. Concurrent programs that employ message passing are called distributed programs, because the processes can be distributed across the processors of a distributed-memory architecture [17]. We consider the following cases:

- **sequential** – all the tasks are computed one after another. Even if in some cases an event driven model is employed, there are no two simultaneous computations. The sequential applications are the easiest to understand and code, they have the most extensive support from the actual programming tools and they tend to be very reliable. Their biggest drawback is the poor utilization of the available computing resources. Even if the compiler can make some optimizations such as automatic parallelization or vectorization, suitable for CPUs with multiple cores or with Single Instruction Multiple Data (SIMD) capabilities, these optimizations are made only in some very limited cases and most of the program execution is made in a strict sequential way, so mainly the application cannot use the advantages provided by the multicore or SIMD capable CPUs.

- **multithreaded** – some tasks are performed simultaneously, using the available CPU cores, or other local CPUs, in the cases of motherboards with multiple CPUs. The multithreaded applications succeed to fully use the local computer resources but they come with added complexity. New problems such as synchronization and simultaneous access to resources need to be solved and these are sometimes quite hard to model and implement. The debugging process is also much harder for multithreaded applications and in many cases to write a reliable multithreaded application can require several times the time used to write the same application in a sequential manner. The drawback of such applications is that they cannot use remote resources, such as other computers from the same local network.

- **distributed** – some tasks are performed simultaneously, both on the local machine and on the available remote computers. The distributed applications fully use both the local and the remote computing resources for the price of even more added complexity from the multithreaded case. New issues such as network management, serialization/deserialization, deployment, remote invocation and computing resources management need to be addressed. For distributed applications there is only little or inexistent support from the actual development environments. The testing of such applications needs to be done more thoroughly, because there are more interconnected modules, each one with its own requirements, which creates more possible failure points.

From the above considerations, developing distributed applications is a much harder task than developing sequential or even multithreaded applications. Most mainstream languages do not have support for distributed computing at the language level, but this support is only provided through third party libraries and tools. Taking into account that to write distributed applications is more and more necessary, new models, concepts, libraries and tools need to be researched and developed to address the distributed computing issues.

In this thesis, we will concentrate on proposing, implementing and testing new models, frameworks and tools in order to make the distributed applications easier to be developed. The proposed solutions need to fulfill important criteria, such as:

- **reliability** – the presented solution must not only work, but it needs to ensure data and computations integrity. If a computation or part of it cannot be done, it must be reported as so. The application needs to recover well from a wide range of unexpected conditions and errors and all these must be done as much as possible in a dynamic or even real-time manner [18], in order to address the continuous changing conditions existent in a distributed environment

- **abstraction** – all the computing resources, such as local CPUs, remote computers and GPUs must be handled in an uniform way, without explicitly resorting at the application level to specific functions calls or settings for each platform. Standard and well implemented distributed computing tasks, such as code and data serialization/deserialization, code deployment, remote invocation and others must be handled by the system and libraries themselves and not by the application programmer. The programmer must concentrate mostly on the implementation of the high level application logic [19].

- **simplicity** – the proposed solutions must be simple and familiar to many programmers. In order to achieve this, the models need to have only a few but powerful concepts and they need to express well the application logic.

This report is structured in three main modules:

- **motivation, objectives and main concepts** – in these sections we motivate the thesis research by outlining some insufficiently developed areas in the field, and we provide an overview of the distributed computing domain, with a discussion of the most used concepts.

- **the distributed computing model and a possible Java implementation** – in these sections we explain our model, and we highlight its advantages and use case. It is also presented a Java framework which implements the model as a library, making it available for any Java programs.

- algorithms for tasks distribution on GPU – in these sections we propose two algorithms which enable the GPU use as a computing resource. The first algorithm can be used when the application modules are not suitable to run entirely on GPU. The second algorithm can be used when an entire application module can be run on GPU. It automatically translates the Java bytecode to OpenCL and runs it on GPU

2. Thesis motivation, objectives and structure

2.1 Motivation

Taking into account the current status of the distributed computing domain, there are some insufficient researched or developed areas. Due to this fact, even if there is a growing need for the complex applications to distribute computation tasks on more computing resources, developing distributed applications is a complex task, with many pitfalls, the testing process is hard and even with all these, not all the computing resources are fully used. We highlight several areas which we identified as being insufficient researched and developed and our motivation is to enhance them, both from a theoretical and a practical point of view. In this way, we want to make them more powerful, easier to be used and capable to handle a much wider area of tasks, in an abstract way from the point of view of the distributed computing aspects.

Current models widely used in distributed computing are taken from other programming languages. For example, the **MapReduce** model is taken from the functional languages, such as the **map** and **reduce** primitives from Lisp [20]. It has a solid theoretical foundation and it can be used to express powerful abstractions, but the model current implementation for distributed computing has several drawbacks:

- **it is implemented only by using frameworks and does not have a language level support** – this makes its usage harder, due to the fact that many standard actions, which should be automatically solved from the compiling phase (if there were language level support), are left for the programmer to implement. For example framework calls for initializing the scheduler or for results synchronization must be explicitly made. Some languages, such as Scala, have a language level support for distributed computing, but these primitives model an Actors based computing model. These primitives can be used to implement a MapReduce model, but in this case in fact a library is developed and the programmer will use library calls and not the language primitives [21].

- **it needs to be extended with specific distributed computing concepts** - the functional languages from which this model was taken were mainly sequential languages. Several shortcomings of these primitives are present:

    - they do not have synchronization constructs – initially such languages did not need synchronization constructs for concurrent data access or for results ordering

- these primitives are not designed to work with asynchronous calls – they were meant to be blocking calls, due to the sequential character of the languages

- their strict implementation results in a lack of efficiency, which sometimes can be very serious – originally these primitives were designed to work with data stored in local memory, so the access speed to all data was the same. In a distributed computation there are most of the times at least two types of data: **global constant data**, which is the same for all computations and it does not change and **invocation specific data**, which is particular to each invocation. These primitives do not have constructs to differentiate between the two types of data, so the developer must manually specify how the application data must be handled (especially in order to decrease the network traffic), which adds more complexity to the application.

Due to the above shortcomings, new models and concepts need to be researched and developed, in order to make them more suitable for distributed computing. These models must be general, simple, they need to ensure a high computational performance and the concepts must be familiar to the application domain.

Due to the increasing need for complex computations or large data processing, many programming languages start to offer parallel or distributed execution features. These features are implemented at the core language level or as libraries. Examples are the concurrency APIs (Application Programming Interface) for Java (java.util.concurrent) or the Task Parallel Library for .NET Framework. These APIs can be extended with 3rd party libraries such as GridGain [22]. This thesis extends concepts and features from these APIs with new aspects such as the integration of the Graphic Processing Units (GPUs) and other elements which are made available by the proposed model.

GPUs are increasingly a valuable computing resource. For algorithms which are massively parallel, a GPU can offer appreciable speedups, in many cases reducing the execution time many times. In bioinformatics, with highly optimized libraries, GPU finely tuned algorithms can provide speedups of up to 1000x [23]. AMD Radeon Fury X [24] with 4096 cores, 8.6 TFLOPS FP32, 4GB HBM memory and NVIDIA GeForce GTX Titan X [25] with 3072 cores, 7 TFLOPS FP32 and 12GB memory are two 2015 top consumer GPUs. These are optimized especially for FP32 computing. Their FP64 performance is lower (GeForce GTX Titan X has 0.2 TFLOPS FP64). An Intel Xeon X7560 CPU is capable of 72.51 GFLOPS FP64 [26]. From this data, the GPUs are important resources. They can enhance greatly specific classes of applications. If the algorithm is massively parallel and FP32 operations are enough, one GPU may provide a performance comparable with several desktop CPUs.

Considering the above considerations, many researchers try to employ the GPU advantages and develop new algorithms and libraries capable to use the GPU computational power. Two main technologies are leaders: OpenCL and NVIDIA CUDA. We will develop on the OpenCL approaches, as it is vendor neutral, open standard and supported by many major vendors. Many discussed aspects also apply to CUDA, underlining that the structure of different GPUs has common elements, and OpenCL and CUDA are layers of abstraction over that physical structure.

In order to achieve their impressive number of cores, the GPUs must impose some limitations in other areas. Some of the GPUs tradeoffs are listed in the following paragraphs.

A first tradeoff is that a GPU core does not have its own instructions fetching and decoding unit but many cores are grouped in workgroups which run the same algorithm. In this respect a GPU core is more like an Arithmetic and Logic Unit (ALU) of a CPU core, than a full CPU core. The algorithms which require mostly arithmetic operations without execution branches are especially suitable to use a Single Instruction Multiple Data (SIMD) model [27], because in this case many ALU can work in parallel on the same instruction but on different data. This ensures a high computation throughput. Because such algorithms can be useful in multiple areas, many CPUs also include some forms of SIMD execution: on Intel/AMD the MMX, SSE, AVX instruction sets, on ARM the Neon extensions, etc. On GPU an entire workgroup execution is controlled by a single Computing Unit (CU), which implements the instructions fetching, decoding and other synchronization aspects. If a conditional branching makes different cores inside the workgroup to choose different execution paths, the CU will put some of them in a waiting state until the execution will resume at the same instruction, sometimes only after other cores finished their jobs. This phenomenon is named branch divergence [28]. GPU cores [29][30] are simpler than CPUs, which use optimizations such as out-of-order speculative execution. The speculative execution is mainly precluded because a single CU coordinates many GPU cores by using them to run the same instruction on different data, so it cannot simultaneously try different branching paths on all its cores. CPUs generally have a greater clock frequency than GPUs, so a CPU core has the same throughput as several GPU cores.

Another GPU tradeoff is that it does not have a stack. The OpenCL compiler is required to inline all the functions code in one kernel (the code to be run on a GPU core), so it eliminates all the functions calls. This limits the applicability of GPU use only to non-recursive algorithms. This is one of the reasons why many modern researches try to find optimized non-recursive variants [31][32] to recursive algorithms. A simple approach would be to eliminate the recursion by simulating the stack using suitable data structures, but this brings new questions regarding how to partition the limited GPU memory between many stacks and the global heap space. If we consider separate stacks for each core, in order to eliminate the synchronization overhead between cores, a GPU with 4096 cores would need 1 GB memory only to provide a 256 KB stack to each core. A simulated stack also eliminates the compiler possibility to make some optimizations, for example to eliminate some variables, because the programmer explicitly allocates space in the simulated stack for all the used variables.

Regarding the GPU access to the host computer operating system (OS) or devices, a tradeoff is that a program executed on GPU does not have access to them. It cannot make OS calls in order use the disks or the network. This aspect limits the use of GPUs only to certain segments (without I/O operations) of an application.

Until OpenCL 2.0 (which at this moment is implemented only for some GPUs) the GPUs had a different memory space than the CPU. This tradeoff has its roots in the history of GPUs, when they were only very specialized peripheral devices, without general computing abilities. Different memory spaces makes difficult to share complex data structures which involved pointers. These structures needed to be serialized with the pointers converted to other representations such as indexes or identifiers, transferred to GPU and deserialized. In some cases this process is repeated when the data is transferred back to CPU. These aspects complicate the algorithm and reduce the potential speedup obtained from the GPU execution. In the same time memory transfers between CPU and GPU must be made over an expansion bus (PCIe) and this process can be much slower than the access to regular memory. OpenCL does not have mechanisms for dynamic memory allocation (malloc/free) so these must be implemented by taking into account factors such as the high possible concurrency (thousands of cores) on accessing shared resources (the heap memory). A memory allocator which is not optimized for massive concurrency would become a performance bottleneck by keeping all the memory allocation requests on hold while it processes one request. Without dynamic memory allocation even simple algorithms such as working with variable sized data would need to consume much more memory, by preallocating suitable buffer spaces for all the possible cases.

Since the GPU general computing power vastly increased, it would be important to use this power even for more complex algorithms, not only for the mostly numerical ones. It can be seen from the above factors that to design algorithms suitable to run on GPUs is not an easy task. This usage is even impossible when I/O operations are required. In situations with complex code flows and data structures it is possible that the GPU execution does not bring significant speedups. It is therefore desirable to use a mixed approach CPU/GPU execution and to run the different parts of the application on the most suitable device. Our research improves on this direction with a new algorithm suitable for cooperative CPU/GPU computing even for complex cases such as GPU suitable sequences embedded in multithreaded algorithms run on CPU.

Due to the above limitations, it is not easy to integrate the GPU for general purpose programming tasks, especially when complex algorithms are needed. New ways to use the GPU in an application must be found. These new modalities must make available computation on GPU for complex algorithms and must ensure an optimal usage of the vast parallel processing power of the GPU. In the same time, the GPU or CPU cores selection and usage must be as much as possible hidden from the programmer, so he can access all computational resources in a uniform and abstract way.

Another research direction is to automate as much as possible the GPU use from a high level language such as Java, by automatically handling tasks such as OpenCL kernel code generation from Java bytecode, data serialization/deserialization and synchronization. These tasks can be formalized in a general way and implemented as algorithms. For example the translation from Java bytecode to OpenCL (which is a dialect of C) can be formalized by using the compilers theory. In section 7 we present some of the current approaches in these directions and our own contribution.

2.2 Objectives

The main objectives of this thesis are to research and develop a MapReduce inspired model and frameworks capable to distribute the computational tasks in an abstract manner over the main existing computation resources (CPUs, network computers, GPUs). In these frameworks we strive to automate many of the unique and complex requirements of the distributed computed domain. In this way many distributed programming tasks are handled automatically and the programmer can concentrate on the application logic.

First we introduce a new high level model capable to use in a uniform way the computing resources and in the same time automatically handle the distributed computing aspects such as network management, code deployment, serialization/deserialization, remote invocation, invocations scheduler. This model should satisfy the following conditions:

- it needs to be **general**, in order to be applicable to as many as possible types of applications. We target especially the computations which can be split in mainly independent processes, such as graphic rendering of multiple frames or data mining on large collections.
- it needs to be **simple** and use only a few concepts, so the developers could have a quick learning rate and in the same time they could master the tools they use
- it needs to **ensure a high computational performance**, due to the fact that the distributed computing is used to solve problems requiring vast amounts of computation power, so any optimization or any limitation can be magnified in good or in worse hundreds or thousands of times
- it needs to **be applicable to the application domain in a natural way**, so the developers will not need to use different stratagems in order to adapt the application algorithm to the proposed concepts and workflow

Second we create a framework which provides the means to use this model. This framework should address the following requirements:

- **all the low level tasks** such as resource discovery, remote invocation, serialization/deserialization, load balancing and error recovery **should be handled by the framework in an abstract manner**, so the developer can concentrate on the application logic. If the developer needs fine tuning of the above aspects, he should have access to their settings in order to tailor them to his specific needs.

- **the framework should use all the available computing resources**, such as local CPU cores, network computers and GPUs. The employment of these resources must be done **in a generic way**, so the programmer should not develop different implementations for different computing resources. The framework may provide a way to evaluate the performance of the available computing resources and use them accordingly, for example the local GPUs and CPU cores and if these are occupied, the remote resources.

- **the interface should be simple and it should use only a restricted set of concepts**. In this way the framework will be easy to be learned and applied by developers.

- **the framework should use when possible well established, industry standard libraries and tools**. Taking into account the fact that there are many low-level libraries, well developed and optimized over the years and which are open source, their integration in a high level framework would be a big advantage. This integration would lead to a significant decrease of the required development time, ensuring in the same time the usage of production ready solutions [33]. For example industry standard serialization or networking libraries can be employed.

- **all the framework components must work in a decoupled manner,** in order to be able to use different settings on a specific level, without affecting other levels. For example, if the available network imposes restrictions on using TCP non-standard ports, the framework should be capable to use a different transport protocol, for example HTTP.

- **the framework should also provide a remote component (server/service provider)** which will be installed on the remote computers. This server will be the destination of the deployment of the application into network and through it the data flow, resources management and the remote invocations will take place. This server needs to be able to do the following actions:

    - respond to regular queries about its version or available resources
    - receive the code destined to run remotely
    - receive the global and invocation data
    - instantiate the code needed to run the invocation and use it to run the computation
    - return the results or the error codes to the main application

In this framework we integrate first the computation on local CPU cores and on the network computers. In a second step we address the integration of the GPUs when these are used as a general purpose computing resource. For each computation resource we present the status of the field and our original contributions. In some cases we developed multiple approaches, in order to address different applications requirements. Each approach was also implemented in a proposed framework and we tested it on different scenarios. The practical results are given and discussed.

2.3 Structure

In this thesis, after the introduction, motivation and objectives, section 3 (*Distributed computing concepts and requirements*) reviews some important concepts for the current research. Different aspects such as network management, code deployment, remote invocation, invocation scheduler and the remote server are discussed with some of their more important options.

Section 4 (*Application level execution model*) details the proposed computation model. The original MapReduce model is first analyzed in the context of a distributed computation and its shortcomings for this kind of computations are identified. After that we introduce our model by means of an example. The model is later formalized and theoretically analyzed. In the end we present an implementation of our model using a virtual machine developed for it and we discuss the practical results.

In section 5 (*Application components distributed computing framework*) we implemented our model as a Java framework (client library and remote server), so it can be used by regular Java applications, without requiring language level constructs. The library was designed to be flexible and generic, so it abstracts many aspects such as computing resources or data destinations. We discuss the implementation and analyze the practical results obtained by using the framework from a test application.

In section 6 (*Algorithm for hybrid execution on both CPU and GPU*) we propose a novel algorithm for the cases where the computation is too complex to be executed only on GPU, for example it has calls to the functions of the operating system. In this case, by using a combination of threads and fibers, we succeed to split the computation in parts suitable for CPU and for GPU and execute them in an efficient manner. The algorithm is especially efficient in the cases where there are multiple threads and each one can have GPU accelerated segments. We provide a C++ test application and we discuss the practical results.

In section 7 (*Java bytecode runtime translation to OpenCL and GPU execution*) we propose a new algorithm and its implementation as a Java library, capable to translate parts of the containing Java application bytecode to OpenCL and run the resulted code on GPU. Beside the use of the reference types, the algorithm is capable to translate some advanced Java constructs such as exceptions and memory allocation, for which there is no OpenCL support. The library implements our proposed model and it abstracts the computing resources, so the same code can run both and CPU and on GPU without any modification. We tested our library and we discuss the practical results. We also make a comparison with the Aparapi library, both as features and as performance.

In section 8 (*Conclusions*) we present the conclusions of our research, emphasizing our novel contributions.

3. Distributed computing concepts and requirements

In this section we discuss some distributed computing aspects, relevant to our research. These aspects can be found in most of the distributed computing applications. A direction of our research was to identify among them common patterns which in most of the cases can be abstracted, so the programmer can focus more on the application logic. In this respect, our proposed model and algorithms try to automate as much as possible the tasks involved by these concepts and requirements.

When compared with sequential or even with multithreaded applications, the distributed applications have several new requirements, such as network management or serialization. A distributed computing framework must address these requirements. In the following sections they will be explained specifically in correlation with their role in distributed computing. In our thesis we are concerned especially with the presentation and the application layer of the Open Systems Interconnection (OSI) layers [1].

3.1 Network management

The network management (services discovery, communication, security) is very important for the distributed applications because the network is the main mean to distribute the tasks and retrieve the results. There are several issues regarding the network, such as:

3.1.1 Servers or services providers discovery

A distributed application is formed from components which run on different hardware. These components need to cooperate so they need to connect in some way to other computers in order to use them. The components can be installed on the available hardware in several ways:

- on each computer the entire application package is installed

- a component which acts as a specific service provider is installed. It is used to handle requests, compute them and return the results

- a component which acts as a generic server that can be used to receive data and if necessary application components is installed. It instantiates these components, passes the data to them for computation and retrieves the results

In any of these cases, the application must have a way to detect the remote available computers (the ones which can act as servers or service providers). This detection is named **resource discovery** [34][35][36] and it can be done in several ways:

- **using broadcast messages** – the application sends query messages to all the computers from the registered networks to check which ones of them are available for computations. The networks to be searched can be obtained from a simple configuration file, which holds a list of hosts and networks. This is a truly distributed approach, but it can be quite slow to broadcast queries on large networks and it also creates more network traffic.

- **using resources index servers** – the remote computers register themselves on an index server [37]. When large or hierarchical networks are concerned, several machines can act as index servers in a hierarchical topology. The application will query the servers for the registered remote computers and it will get their list. This approach makes the resource discovery simple and quick, but it can create a centralized point of failure if something happens with the index server or with the communication with it. These shortcomings can be alleviated if redundant index servers are kept, better in different segments of the network.

For either method naming services (DNS, RMI registry) can be used to obtain the resources network address from the application specific resources description. After the available remote computers are found, subsequent queries are made in order to obtain their computing power, availability and current load. All these factors are important for the tasks scheduler, in order to make as good as possible decisions regarding where to send the computing tasks.

Especially for grid computing, the resource discovery must be a continuous process, due to the fact that in any moment a remote computer can join or can leave the network [38].

3.1.2 Data and code communication

To send data and code over the network [39][40], well defined communication protocols must be employed. These protocols must address some factors such as:

- **all data and code must have standard sizes and definitions**, for example the int type to have 4 bytes and the floating point numbers to be encoded in IEEE 754 format. If compression schemes such as variable-sized integers [40] are used, these schemes must also ensure platform neutrality.
- **when structures are sent, the members order (and padding or separators, if any) must be well defined**
- **before data is sent over the network, it must be encoded in an architecture neutral form**, for example in little endian or big endian formats. The process of encoding data in an architecture neutral form as called **serialization**. The process of decoding data from the architecture neutral form to a specific computer and application format is called **deserialization**. The process of serialization/deserialization is especially important for heterogeneous networks, in which CPUs with different endianness and data sizes can be found.

The above aspects are included in the data and code encoding formats [41]. When the required bytes themselves are sent, respecting the architecture neutral form, we have a **binary format**. Binary formats ensure fast encoding/decoding and do not introduce much overhead. For some applications, a disadvantage is that they are not human readable, so they cannot be easily inspected. In the same time the binary formats are very tightly connected with the application internal structures, which make these formats to change quite often when new application versions are released.

Sometimes the data and code are encoded in standardized, well-known, human readable formats, such as XML or JSON. These formats care named **text formats**. They require more time for encoding/decoding than the binary formats and in the same time they tend to be larger, which can be an issue when the data needs to be sent over the network. The main advantages of the text formats are that there are human readable, so they can be easily inspected, especially for debugging purposes and they can interoperate easier with other tools or services [42].

To reduce the network traffic, the serialized data can also be compressed. This is efficient especially for text formats and for communications over Wide Area Networks (WAN). When binary data is sent and this data is not well compressible (for example already compressed multimedia formats), or when the communication is made over fast local networks, the compression can slow the network communication.

There are several ways to send serialized data over the network:

- **using TCP/IP or other low-level network protocols** – this method ensures faster speeds because of a lower overhead and it also offers more customizations. Its drawback is that some firewalls or internet providers block the access to non-standard network ports so these computers will not be able to be used.

- **using high level protocols, such as http** – this method has a bigger overhead than the first one, but it has the advantage that it can use standard ports and accepted formats for data packets, so it can be used over many firewalls and with many internet providers.

3.1.3 Security

When a computer sends data over the network, or when it receives data from the network, some security aspects must be observed [43][44][45][46]. There are several issues and we can cite the following ones:

- **when sensitive data is sent over the network, it must be encrypted or Virtual Private Networks (VPN) must be employed**. Data encryption can be made at the application level, using custom encryption systems, or standard encrypted communication protocols such as SSL can be used.

- **some remote services or servers can be made available only for certain users or hosts**. In this case the communication between application and the remote resources must include authentication mechanisms or host identifying steps. These mechanisms can employ for example user/password authentication or electronic signatures. The authentication can also differentiate the user type, when different services are offered to different users categories [47].

- **logging for the communication history must be employed for all the cases when important data is used or sensitive computations are performed**. In case of later problems, divergences or disagreements, these logs can be used to assert the real history of the events and in the same time they can help to recover from some situations [48]

3.2 Code deployment

In most of the cases a distributed application needs to send components or all of it to the remote computers [49]. This process is called **deployment** and it can be done in several ways:

- **manual** – the network administrator or the application owner manually installs the remote parts of the application on the remote computers. For volunteer networks, the computers owners install themselves the remote parts of the application on their computers. This is the most tedious and time consuming way of deployment. Its main advantage is that for heterogeneous networks, a human can manually optimize the application settings for each computer [50].

- **made by the application itself** - the applications can have the ability to replicate themselves over the network, when suitable computers are found. The remote computers must have a receiver application which can receive and manage (install, run, update, uninstall) the distributed application. This deployment method is faster and easier, but it requires from the application developer himself to implement inside the application the code necessary for replication [51].

- **made by the distributed framework in which the application is run** – this method is suitable especially for cloud computing, where specialized frameworks are installed on all computers and all the network is carefully setup for distributed computing [52]. In this case the application developer must provide into the application some standard entry points for the framework used, or to provide some configuration files. The framework will connect to the provided entry points, or will load the configuration files and it will deploy the necessary parts of the application. For example a Java framework can specify that the distributed modules of an application must implement a certain interface. The framework will scan all the classes for this interface, load the ones which implement it and using this interface collect enough information to know what to deploy. This deployment method is very reliable, because it uses industry standard frameworks which operate in carefully set networks. It is also quite easy to use, because the developer must implement only some small interfaces and the deployment work will be handled by the framework. The drawback for this method is that in most of the cases the framework imposes some special software requirements for the remote computers, effectively making them dedicated for distributed computing. This approach cannot be used for volunteer networks, or when the remote computers are mainly used for other jobs, not for distributed computing.

In any of the above cases, some more aspects need to be addressed:

- **handling of the newer application versions** – when new versions of the application are available, these must be deployed to replace the old ones. Especially on grid networks it is possible that the update process will not affect all the computers with the old application version installed, for example because they are offline at the update time. In this case, some computers will run the new application version and others the old version. To solve such cases, all communication protocols must have a version identifier and a remote computer will take part into the computation only if it can handle the current version of the protocol. Else, it can request the main application to send updates [53].

- **code caching** – in order to decrease the network traffic and the remote startup time, the remote computers can maintain between sessions a cache of the applications sent to them. Every cached code must have a method to be compared with the actual code which needs to be run. For example, a hash can be computed for the cached code and when the new code is about to be sent, it will be transmitted only if on the remote computer is no cached code with the same hash [54].

- **the management of the deployed application** – besides deployment and remote storage, the remote computers must handle other tasks related to the deployed applications, such as their removal. This can be done manually (by the computer owner), at external requests or automatically, according with some setup, for example after a period of time or after a number of computations.

3.3 Remote invocation

The remote invocation is the process to invoke remote code, possibly passing input arguments and retrieving computation results [55]. In order to start the remote computations, the required data must be sent. This data can be categorized in two parts:

- **global data** – it is the same for all computations and it is constant until the end of their life time. This data can be sent only once to each remote computer.

- **invocation specific data** – it is specific for each invocation and must be sent separately for every computation.

For example, a graphic renderer can have some static, unchangeable data for the scenes background. As this data (which can be quite big) will never change, it can be sent only once to each remote computer. The characters and effects instead need to change through the frames, so their parameters will be different for each remote invocation and they need to be sent updated for each computation.

If we make an analogy with a sequential application, the distributed global data corresponds to global or dynamically allocated constant data and the invocation specific data corresponds to the parameters of the functions calls. The functions calls (the distributed remote invocations) can return a result, especially in an asynchronous way [56]. This result must be returned to the main application over the network. If an error or exception is generated on the remote execution, it must also be made known to the application.

In many cases, a remote invocation is performed using the following steps [51]:

- the call arguments and the code for the function needed to process them are encoded in an architecture neutral way and are packaged in an invocation data package [57]. In the case of remote methods calls, it is also encoded the remote object identifier to which belongs the computation.
- the invocation data package is sent to the remote computer
- the invocations server on the remote computer unpacks the invocation data and decodes it to its own platform format
- the requested function for computation is found using the function code and if necessary (for methods) also the computation object, using its identifier
- the data is passed to the function and it is computed
- the results are collected, encoded in an architecture neutral way and are sent back the main application. A special field is also appended to return the status of the computation, i.e. if it was successful or an exception occurred. In the last case, the returned data will contain the exception code or description.
- back at the main application, the returned data package is unpacked and decoded to the architecture own format and passed as result for the invocation. If an exception was sent as return, it is transformed in a native exception and thrown.

As a difference from the applications running only on the local host, in the case of the remote invocations, network errors can also appear. These must be handled and they are treated in a different manner than the returned erroneous computations results (such as exceptions), which belong to the application logic itself [58]. Mainly the network errors occur due to the following factors:

- **network communication errors** – the network link between the main application host and one or more of the remote computers is broken and the communication is cut off.
- **the remote computers leave the network** – for example they are shutdown or the remote server is stopped on them

In these cases, different approaches can be used [59], for example a two phase computation retry. First, the network errors must be detected, for example by using timeouts or ping signals to detect broken communication. Second, if an error is detected, the application can retry to send the invocation data package to it and if this also does not succeed, that host will be removed from the available computation resources and the invocation will be sent to another host. When the resource discovery is performed as a continuous process (important especially for grid networks), if the removed server is detected again online, it can be re-added to the available resources.

3.4 Invocations scheduler

In distributed computation, the main application has a central role of scheduler for all the other remote computations [60]. It sends the required global data, starts invocations and receives the results. In order to do that, the main application has to follow some principles.

To be able to act like a scheduler, all the invocations must be run asynchronously [50][61]. Else, the application will need to wait for their completion, one by one. In the asynchronous case, the invocation data is put in a waiting list and it is sent to the remote computers as they become available. This execution model is much like using worker threads in multitasking applications and in fact it requires worker threads for its own implementation.

When a remote resource becomes available [62][63], one invocation data is taken from the waiting list and is sent to it. After computation, the results are returned to the main application thread. In case of network failures, the scheduler can try to resend the invocation data to the same remote computer or to other available computers.

For grid or volunteer networks, two new problems arise:

- **the remote computers can join or leave the network any time** – this situation is much more frequent than in the cloud computing, where a computer is going off only in case of malfunctioning or revisions/upgrades. To solve this problem, the invocation scheduler must work closely with the resource discovery system, in order to have the updated network situation.

- **the remote computers have very different hardware and software installed** – this situation is very different from the cloud computing case, where the hardware is quite the same, the installed software in most cases is exactly the same for the entire cloud and all these aspects are tightly controlled by the network administrators. When dealing with different hardware and software, the main application must deploy correspondent variants for the existent remote components, for example it can have two different modules, one for computers running Linux and one for computers running MS Windows. In the same time, if one module is optimized for specific instruction sets (such as SSE or AVX), another module is needed to run on older hardware, which does not have these instruction sets. If the application is coded in Java or .Net, there will be no difference related to the remote hardware or software (besides the installed Java or .Net version) so only one module is needed for all situations.

In the situations when different hardware is used [64], or when the remote server is allowed to use only a percent of the host computing power, an advanced scheduler can evaluate the computational capabilities of each remote computer and assign different tasks to them, in such a way that they are used optimally, from the point of view of the total computation time [65][66][67].

When all the invocations are distributed to the remote computers and there are still unused remote computers, the scheduler can send the same invocation to more computers, with the hope that one of them will finish it sooner. This optimization can be very effective on heterogeneous networks, where different hardware can have very different computational capabilities. In this case, if another resource finished earlier, a special message can be send to the servers which are still computing the invocation, to tell them to cancel that particular computation.

3.5 The remote server

On the remote computers special software must be installed, in order to give access to the resources of that computer. In some cases the application itself can run in two modes, for performing the required computations or as a server, but in most of the cases the server is another piece of software.

The server has several tasks to do [4][51]:

- **respond to general queries** – in a distributed computing communication protocol there can be many queries, such as for retrieving the type and version of the server, its computing capabilities, security policies or its current load.

- **receive the deployed code of the remote applications** – this code can be stored or cached for future usage and it must be kept in such way that it can be uniquely addressed by the application which sent it

- **receive the global and invocations data packages** – this data must be unpacked and converted into the host native format

- **run the invocations** – to run an invocation, the server must load the code necessary for that invocation (the target function and all its dependencies) and run it with the invocation data. In the case of a method, an object must be instantiated or retrieved if the required instance already exists.

- **get the computations results and package them for sending over network** – when the computation is done, its result must be retrieved. If an exception occurs, it is also converted to a special data package

- **send the results back to the main application** – the results are sent back to the application. In case of network errors, the sending process can be repeated several times.

- **free the resources allocated for a specific application when that application ends** – the global data and code related to that application are freed from memory. The servers with caching facilities have specific options regarding the cache size, the disk and memory size used for cache. In this cache especially the application code can be stored, due to the fact that it changes only at relatively long periods of time and in most of the cases an application will be run many times.

In the case of grid or volunteer networks special usage policies must be provided [68]. In these cases the server needs to run according to the host owner preferences, such as:

- the remote applications will run only when the CPU or memory are free over some percent, situation defined as **host in idle state**.
- the remote applications will run as low priority threads and they cannot use resources (CPU, memory, disk space, network bandwidth) over a certain, user defined threshold
- the user can at any time stop, pause or resume the remote computations
- the remote applications must be isolated from the local data, such as the user personal files or settings, in order to maintain his privacy

All the above usage policies must be met in order not to disturb the user work and privacy. In this way the user will be in complete control of the usage of his computer. Due to this fact he can accept more easily to contribute with computing resources to the grid, because he can finely tune his contribution and in the same time he can be sure that his privacy will be ensured.

3.6 Conclusions

In this section we discussed some common distributed computing concepts and requirements. They can be found in most of the distributed computing applications, so it is important if they can be automated. Even if they are implemented as libraries (for example Java RMI), the use of these libraries add its own complexity to the coding effort and to the application maintenance.

As we will show in the next sections, our research proves that in many cases, especially when the application can be coded in a Divide et Impera manner, these requirements imposed by a distributed computation can be to a greater extent automated and abstracted from the programmer. An added benefit is a more uniform handling of different computing resources such as CPU, remote computers or GPU.

4. Application level execution model

In this section we present our novel distributed computing model. It is based on the well-known MapReduce model. First we analyze the current MapReduce model we and discuss it in the context of the distributed applications, in order to identify its shortcomings for this use case. Next we introduce our model by means of an example. After that we discuss in depth the concepts our model, their interactions and we present a theoretical analysis of its performance. We also provide an implementation of our model as a virtual machine and we discuss the practical results.

One of the most known computation models for distributed computing is **MapReduce**. It has a solid theoretical foundation and it was mainly used in functional programming languages such as Lisp [20]. With the advent of the first order functions and closures in many mainstream languages, primitives equivalent with **map** and **reduce** are now also available in standard libraries for Java or C#. We quote how this model handles a computation [20]:

"The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the ***MapReduce*** *library expresses the computation as two functions:* ***Map*** *and* ***Reduce****.*

Map*, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The* ***MapReduce*** *library groups together all intermediate values associated with the same intermediate key and passes them to the* ***Reduce*** *function.*

The ***Reduce*** *function, also written by the user, accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are*

Section based on the paper: Razvan-Mihai Aciu, Horia Ciocarlie, ”Application Level Execution Model for Transparent Distributed Computing”, New Results in Dependability and Computer Systems, Proceedings of the 8th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, September 9-13, 2013, Brunów, Poland, Springer, ISBN: 978-3-319-00944-5 *supplied to the user's **reduce** function via an iterator. This allows us to handle lists of values that are too large to fit in memory."*

For our purposes we propose a MapReduce model. In the **map** phase the computation is split in independent units and each unit is computed separately. If we consider the computation of a single unit as a function f:X->Y, where X,Y are any simple or compound types, we can write:

$$R=\{ f(x_i) \mid x_i \in J \}, J \subset X, R \subset Y$$

- J – the collection of jobs to be computed
- R – the collection of results, each element being the result of the same computation f applied to one element of J

If an ordering is required (the (key,value) pairs from the orginial description), the X and Y can also include the required keys besides values. Our model makes this original requirement optional, because not any MapReduce computation requires keys. For example, an application which only counts the total number of words on multiple texts does not need keys to identify the source text for that partial count.

With the above notations we can write: R=map(f,J). Traditionally a map phase is applied in a sequential way (in Lisp, OCaml, JavaScript, etc). In more recent libraries (such as Task Parallel for C#) the map phase can be applied in parallel.

In the **reduce** phase the independent results of the map phase are combined into a final result. If we consider g as a combining function and Z is the type of the final result, we can write:

$$g:Y^n \rightarrow Z, n=|R|$$

The above is the most generic form of the reduce function which can apply any combination from the result collection R. In practice simpler forms are used, for example by iterating R together with an initial value (an accumulator) and successively applying g to the current element of R and to the accumulator. The result will be the new accumulator (the fold family of functions from OCaml). In this case, the reduce function can take the form:

$$g:(Z,\{Y\}) \rightarrow Z$$

where the first argument of g is the accumulator of the partial results and the second argument is the list of the partial results computed in the **map** phase (the R collection). If g does not need a specific order to reduce the partial results, these can be passed directly to g, without putting them first in the R collection.

4.1 The original MapReduce shortcomings when a distributed computation is involved

The original implementations of MapReduce are found in functional languages such as Lisp. These languages were used mainly to express sequential computations and only later concurrent versions such as Multilisp were designed [69]. In a sequential computation some assumptions are (implicitly) made:

- **Assumption 1** – because of the sequential model, all the computations were done synchronously and the results are available immediately on computation function return. From this moment the results can be safely used.
- **Assumption 2** – in most of the cases all the involved data resides in the same memory so any part of it can be roughly accessed in the same way and requiring about the same access time (taking into account the cache memory and the disk swap).
- **Assumption 3** – aside of the computation errors and sometimes resources (memory or stack) exhaustion, there are no other sources of errors beside hardware failure, in which case in most of the times all the computation is aborted.
- **Assumption 4** – because of the sequential execution and also because of the predominant immutable nature of the data involved, there were no problems of concurrent access to resources (data racing) so in the map phase it was safe to apply the computation function without any data synchronization. In a sequential computation, a problem which may arise from a map function with side effects is when a specific order must be ensured on the map function calls. In this case, if the specific

order is not ensured, the side effects can change the computation result.

In a concurrent execution model and even more in a distributed computing model, the above assumptions are not necessary true. Because of them the original MapReduce model cannot be directly applied in distributed computing. We need to define new semantics and operations to be able to create a computation model for distributed computing applications, in the same time trying to keep as much as possible from the original MapReduce model, which was proved during time to be a successful computation model.

In the following sections we will discuss each assumption and we will analyze its requirements from a distributed computing point of view.

Assumption 1 – all the computations are done synchronously and the results are available immediately on computation function return. In a concurrent model, the computations are mainly done asynchronously and they can be available at any later time, during the program execution [70]. There are mainly two methods to deal with asynchronous operations:

- The main program or the operating system inspects from time to time the availability of the results. In this method some CPU cycles are lost due to necessary verifications.
- The main program enters in a wait state and the computation function notifies the program when the result is ready. In this method the CPU is better used because it does not consume cycles during the wait of the main program thread.

We use in our model the second method because it uses better the CPU. In this way, while waiting for a result from an asynchronous call, the CPU can be used for other tasks. The final step of the computation must be a synchronization step, in order to be sure that all results have arrived and they were processed.

Another important aspect regarding the synchronization phase is the fact that not all algorithms require a full synchronization of all results of the map phase before the reduce phase. For example if each of the jobs provides data to be written in a separate file, these files can be written immediately on data arrival without needing to wait for other results (the reduce phase is not needed). In these cases standard synchronization functions such as "join" or "wait" must be carefully used, in order not to introduce not-needed waiting points.

Assumption 2 – the involved data resides in the same memory so any part of it can be roughly accessed in the same way and it requires about the same access time. In a distributed computing environment it is possible that parts of the application and associated data to run on different computers (which requires network traffic) or on different devices such as GPU (which requires traffic through the peripheral interfaces). Because any such traffic is much slower than the main memory access, it is very important to divide the data involved in computation in more categories, in order to optimize its usage.

In our model we consider two types of distributed data involved in computation:

- **global immutable (static) data** – this data is common to all distributed jobs and it remains the same during the computation. Because of this, if we need to run some jobs on a remote computer, it is enough to send this data only once to that computer and all jobs will use the same instance. This optimization can be quite important for the network traffic because there are applications in which this data is very large or there are many jobs and if we send this data with each job, the network traffic would be considerably increased. For example, let's consider an application which renders a movie. There are many components such as textures, background or non-modifiable elements which are all immutable data and its size can be quite big. In the same time there are a lot of jobs, for example to render a 2h movie at 30 FPS, 216000 jobs are needed, one for each frame.

- **job specific data** – this data is specific to each job and it needs to be sent separately for each job. For the above example, we can consider job specific data changeable positions, deformations or colors change, any aspects which change from frame to frame. Sometimes parts of this data can be further optimized by considering them immutable data if in more jobs these parts are not changed. In this case we can consider clusters of jobs organized in such way that the amount of immutable data is maximized inside such a cluster.

Assumption 3 – if the algorithm runs correctly, aside of memory/stack exhaustion there are no other errors. In a distributed computing environment even if the algorithm runs correctly (without errors such as accessing null or dangling pointers) there are a lot of new sources of errors. Most of these new errors are related to network communication or to the availability of the remote computers, especially on the case of heterogeneous networks. These errors are so frequent that they should be handled gracefully by the application, without aborting the entire computation. Assuming that the algorithm runs correctly and the results are valid, the above errors have mostly the effect of dropping the results of some computations. These errors can be handled by setting timeouts and/or sending query messages (pings) to the remote computers to test their status. If the timeout is reached the remote computation is considered lost. The main problem is the determination of the timeout time, because there are several factors involved:

- for heterogeneous networks, the computers can have very different computing capabilities and local workload factor. In this case even a timeout time is computed by testing the time needed for one machine to complete a job, this time may not suffice for other machines to end their computations.
- even in the case of homogeneous networks, it is possible that different tasks have very different computations requirements. For example to render a scene with a lot of complex objects and effects can take orders of magnitudes longer than to render an almost empty scene.

The recovery from such errors involves especially creating a new job with the arguments of the lost computation. If there are enough computing resources we can also assign a redundancy factor to each job and it will be run simultaneously on multiple computing resources. The first resource which sends back the results is "the winner" and the other computations are aborted. Creating redundant (backup) tasks is also useful to speedup computations [20], by creating a competition between different computing resources. For lost computations it is also important to send to the remote computers (if they are still reachable) cancellation requests, in order to be as sure as possible that their resources are not still used for a computation considered lost.

Assumption 4 – during the map phase there is no need for synchronization for data access (no data races). In a concurrent computing environment many algorithms have shared data so data access synchronization is needed. More than that, on a distributed computing environment, because different jobs can run on different machines, the shared (common) data will in fact be replicated on different places, so the data modifications will not be visible globally to all distributed jobs. The main solution to this problem is to keep the data on only one machine and when a job requires it, the necessary data will be locked, send to that specific job, used, send back and unlocked. This approach has several drawbacks, some of them quite important and because of these drawbacks many distributed computing models avoids using shared data and they use instead mechanisms such as pipes. Some of the drawbacks of using shared data are:

- if network traffic is involved, the access time to data can be very large, especially for Wide Area Networks (WAN).
- if many jobs needs the same data in the same time, its access will become a bottleneck and all these jobs will need to wait while only one job accesses the data
- if the job which locked the data is lost, all the other jobs will wait for data unlocking for a time equal with the timeout time, which can be quite big.

Because of these considerations we decided in our model to forbid completely shared mutable data (but it allows shared immutable data). In this way a programmer which uses our model will need to adopt a programming style similar to programming in functional languages, in which global variables are seen as highly undesirable. If this kind of data is still needed, a simple approach can be used: the shared data will be managed by a dedicated server/service which is also responsible for its synchronization. The accesses to this data server take place much like the accesses to an SQL server: data accesses are made mainly in small packages and are considered atomic (they can have rollbacks if necessary). The most important thing is to design the application from the beginning with the distributed environment in mind so the shared data will be as small as possible and its accesses needed only rarely. In this way the programmer is made aware that any access to shared mutable data can incur a high penalty (stopping other processes to access that data, network traffic) and he will design from the beginning the application as to minimize these bottlenecks.

From the above considerations, these assumptions which are true in a sequential model do not hold on a concurrent or distributed environment. Our model tries to define clear semantics for all the aspects involved and to create a generic algorithm so it can be safely and efficiently used in distributed computing applications.

4.2 A typical example of a distributed computing application

Ideally speaking, an abstract model at application level will hide from the programmer all tasks required for low level implementation. Also, it needs to fit into the existing frameworks and programming languages with minimal additions, so it can be implemented easily. The proposed model requires only three concepts and these are familiar to the OOP programming style. A programmer with an OOP background should feel comfortable in using them. We introduce the model with the help of an example.

To introduce our model, we implemented on a specific interval the Mandelbrot set, which is a well known algorithm. For this example we devised a C++ style language that highlights the specific code for the proposed model, as listed in Figure 4.1.

```
unit MandelbrotRow{
  double  minX,maxX;
  int     w,maxIters;
  MandelbrotRow(double minX,double maxX,int w,int maxIters)
  {
    this->minX=minX;
    his->maxX=maxX;
    this->w=w;
    this->maxIters=maxIters;
  }
  string     run(double yy)
  {
    int       pxl,iter;
    stringstream  res;
    double       xiter,yiter,x0,xtmp;
    for(pxl=0;pxl<w;pxl++){
         x0=minX+pxl*(maxX-minX)/w;
         xiter=yiter=0;
         iter=0;
         while(xiter*xiter+yiter*yiter<2*2 && iter<maxIters){
           xtmp=xiter*xiter-yiter*yiter+x0;
           yiter=2*xiter*yiter+yy;
           xiter=xtmp;
           iter++;
           }
         res<<iter%256<<" ";    //convert to grayscale
         }
       return res.str();
     }
}

#define PXWIDTH        1000
#define PXHEIGHT       1000
#define MAXITER  10000

int    main()
{
    int              idxLine;
    double         lineY;
    string          image[PXHEIGHT];
    double         minX=0.33072017, maxX=0.33925741;
    double         minY=0.04369091, maxY=0.0522281593;
    with( image ; MandelbrotRow(minX,maxX,PXWIDTH,MAXITER) ){
      for(idxLine=0;idxLine<PXHEIGHT;idxLine++){
        lineY=minY+idxLine*(maxY-minY)/PXHEIGHT;
        run[idxLine](lineY);
        }
      }
    ofstream     outFile("mdl.pgm");

outFile<<"P2"<<endl<<PXWIDTH<<","<<PXHEIGHT<<endl<<"255"
<<endl;
    for(idxLine=0;idxLine<PXHEIGHT;idxLine=idxLine+1)
      outFile<<image[idxLine]<<endl;
}
```

Fig 4.1 – A distributed computing application in a C++ style programming language

In the above code a grayscale Mandelbrot fractal is generated in a given interval and it is saved in a .pgm file. The result is given in Figure 4.2.

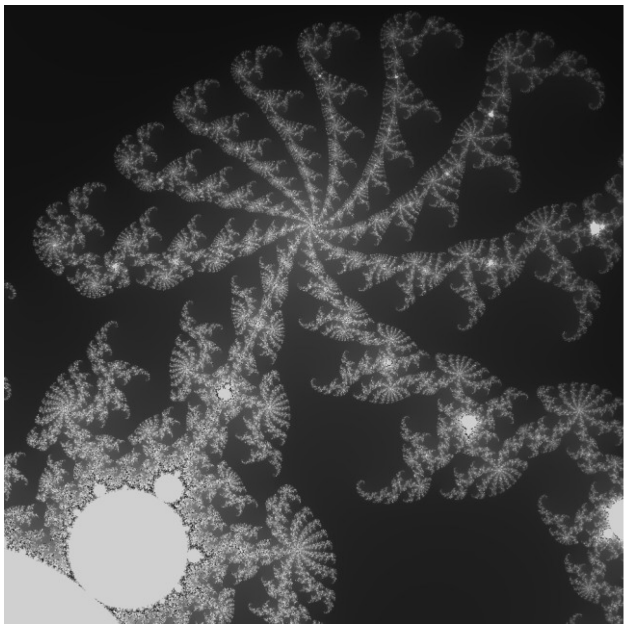

Fig 4.2 – The result generated by the code from Figure 4.1

We chose to compute each horizontal line as a separate job. There are three additions to the C++ language. These additions implement our model and they will be formalized later:

- **unit** – similar to a class definition. A unit encapsulates a job and it is deployed to the computing resources. This construct is analogous to the **map** application from the **MapReduce** model, but at a higher abstraction level. By using OOP class/instance semantics, many **unit** instances can coexist without interference, each one keeping its own private state. In a functional model it would have been harder to isolate individual **map** states in a multithreaded or distributed environment.

- **with** – creates a jobs scheduler and run all its associated tasks, such as jobs queuing, deployment, synchronization, error recovery, etc. It has two arguments: a destination and a unit constructor. Regarding the **MapReduce** model, **with** combines the phases of computation splitting (the creation of the individual **map** jobs) and combining (the **Reduce** phase).

- **run** – adds a new job to the scheduler. It provides both a unique id for the job result (the indexed parameter) and the job specific arguments. Because **run** is used inside a specific **with** construct, it has a better integration with it. In the original MapReduce model explicit result lists must be created and passed further to the **Reduce** function. **run** directly access the scheduler created by the **with** construct and in this way many **MapReduce** aspects such as the handling of the intermediate results can be automated.

Using the above concepts, we can write MapReduce algorithms as in Table 4.1.

Original MapReduce	**Our model**

// computes a single job
function doOneJob(job){…}
//add a new result to final
function add(final,result){…}
jobs = /*split goal into jobs*/;
// computes individual results
results=**Map**(jobs,doOneJob);
// combine all results
final=**Reduce**(results,add);
	// computes a single job, also providing initialization and encapsulation
	unit Worker{…}
	//combines all individual results
	class Destination{…}
	// handles all low-level tasks regarding network, error recovery, synchronisation
	with(Destination,Worker){
	  while((job=/*create new job*/)
	        !=**null**){
	    // run asynchronously the job
	    run[jobId](job);
	    }
	  }
	// if required, combine all results

Table 4.1 – The original MapReduce and our model

In our former example, in the main program the "img" vector is used as the destination of the jobs results. Every job returns a string which is the .pgm encoding of its rendered line. The **with** construct creates a jobs scheduler using its first parameter "img" as destination and it specifies the **unit** used for jobs as its second argument. In this second argument we pass the global constant data as parameters to the **unit** constructor. These data are the same for all computations, so they can be sent only once to each computing resource.

Inside the **with** construct a new job is created using the **run** construct. The **run** construct takes first an auxiliary parameter (in our example "lineIdx") which is used to specify how the job result will be put in destination and after that it takes as parameters all the job specific data, in our example the line vertical coordinate. The job is created asynchronously and it is added to the scheduler. The scheduler will run it on an available resource, by instantiating a **unit** and calling its run method. There is no predefined syntax about how to run jobs inside a **with** construct. The needed **run** constructs can be split in as many as desired **for**, **while**, **do...while**, etc statements, according with the algorithm which provides data for the jobs calls. This aspect gives us an added flexibility above the current approaches which use preprocessor pragmas to parallelize a specific iteration.

At the end of the **with** construct an automatic synchronization point is added. The synchronization point ensures that all jobs will be finished before the control flow reaches the next statement. During the jobs execution, when a new result arrives it is considered according to the destination argument which was given to the **run** construct. In this way the scheduler knows how to handle this result within destination, regardless the order of arrival of different results.

In all the above code there are no explicit distributed computing concepts, such as resource discovery, deployment, synchronization, remote invocation/retrieval, error recovery, etc. In the same time there are no different calls for different computing resources, for example to create GPU kernels for execution on GPU, etc. Our model succeeds to hide at the application level all these aspects. For fine tuning of different aspects, different settings can be provided, for example to search specific network segments for available remote computers or to prioritize specific jobs for GPUs or for remote computers.

4.3 The model functional description

This description is made to facilitate the model implementation in many programming languages. Because each programming language has a specific syntax, we cannot give a syntactical description of the model but a functional one. The main features and interactions are described and they can be implemented using a specific syntax for a programming language.

Our model is structured around three concepts. We designed these concepts by starting with the MapReduce model and we strived both to generalize this model to the distributed computing applications and also to hide as much as possible the low-level tasks of the distributed computing. In the same time our model provides an added flexibility by allowing starting the jobs in any sequence(s) and by defining implicit actions for most common processing needed in the reduce phase.

4.3.1 The unit concept

A **unit** represents a logical abstraction of a job, capable to run in a distributed environment. It has some common properties with the **class** concept from OOP, but with some important differences. Below are the most important aspects of the **unit** concept:

- it cannot have static attributes and it cannot access directly or indirectly static attributes of any class or global data – this requirement ensures that there is no common data shared among jobs. In this way every job will only have its own data (if jobs need to communicate, see the section 4.1, the assumption 4 in a distributed context).

- a **unit** can be deployed to different computing resources, such as remote computers. In his case all its code dependencies will be also deployed. Excepted from this deployment are dependencies of the application environment (for example standard Java API) which already exists on the remote computers. In order to avoid redundant network traffic, each **unit** has a **Global Unique Identifier** (**GUID**). When a **unit** is deployed, first the scheduler asks the remote computer if it already has cached a **unit** with the same GUID. If there is such a cached **unit**, it is used from the remote cache, without further need for redeployment. The GUID is automatically generated on each recompilation, so modified **units** will have different GUIDs. This also means that the new versions are automatically deployed, if the unit code changes.

- a unit is created for each job by using one of its constructors. The **unit** constructor receives the global constant data. This data is deployed only once to each computing resource and the **unit** is not allowed to change it, because it may be also used by other local instances (jobs) of that unit.

- the method **run** of a **unit** is the entry point of the computation. It is called with the specific data for each job. The result of the **run** method is the output of the computation and it is serialized back to the application. If the run method has mainly side effects such as writing to a database server, the returned value can inform about the actions performed or it can be discarded. If any exception occurs, it is also returned to the application.

4.3.2 The with concept

The **with** concept is used to encapsulate a jobs scheduler. It receives two parameters:

- a **destination**
- a **unit** constructor with its arguments, the same as a **new** construct

A **destination** is an abstract concept for any processing involving the jobs results. It can be simply a vector where all the results are stored, a function or an object which implements a standard interface which allows it to be called from the scheduler on each result arrival.

For example, if for each job result a file is created on the main application computer, a destination can be a function which is called with a job identifier (provided by the **run** concept, described later) and with the job results. This function will create a specific file for each result. In this way two goals are achieved:

- a job result is stored in memory only if needed, which can be significant for big results or for many jobs
- if possible the result will be directly processed, without needing an intermediate storage step

The **unit** constructor call used as the second argument of **with** has two functions:

- it specifies the specific job (the **unit** used to implement the computation)

- it defines the global constant data used by each job as parameters given to this constructor

At the beginning of the **with** construction a new scheduler is created. This scheduler has multiple functions:

- it performs computing resources discovery, if it is not done yet or if these are discovered in a dynamic manner, in order to respond to environment changes. These resources are: the cores of the local CPUs, network computers and GPUs with general processing capabilities
- it deploys to the network computers the code of the **unit** along with all its dependencies, if these are not in the local cache of these computers. For GPUs the code of the **unit** and all its dependencies are converted to a GPU kernel and it is compiled (if this compilation is not already done).
- it sends to the remote computers the global constant data.
- It provides a queue of jobs created by the **run** concept
- it distributes the jobs to the available computing resources and receive the results. In order to provide load balancing and redundancy advanced the resources can be selected using many factors, such as their computation power, load or the communication speed.
- if a result is received, it is sent to the destination for storage or for processing
- if an external error (for example a network timeout, not related to the application logic exceptions) occurs, it tries to compute again that job, possibly using other computing resource
- in the end it waits for all the jobs in the queue to be processed and their results retrieved

4.3.3 The run concept

The **run** concept creates a new job and puts it and the current scheduler queue. It has two sets of arguments:

- a unique identifier for each job which is used to identify each job inside a destination. This identifier can be an n-dimensional index for n-dimensional vector destinations, strings for associative arrays (maps, dictionaries), etc.
- specific data for each job

These arguments define a new job which is enqueued for execution. The **run** call is asynchronous and it returns immediately. It remains at the discretion of the scheduler the following aspects:

- when the job will run
- in what order the jobs are run
- where the execution takes place
- if the job runs once or multiple times (to recover from external errors)

The only constrains that the scheduler ensures are:

- all the jobs are run (if no application logic error occurs) and their results are passed to destination
- the destination receives all the jobs results before the execution flow leaves the **with** concept

A job can be created anywhere inside a **with** concept. This gives a great flexibility to the application logic, because the algorithm can use multiple statements, including nested/recursive function calls from where these jobs can be created. In general this level of flexibility is not achievable using decorations (pragmas, annotations) on specific statements.

The above concepts (**unit**, **with**, **run**) are enough to implement many distributed computing computations. Of course, because our model started with the MapReduce model, the jobs must not have mutable shared data. If this kind data is needed, it can be implemented as discussed in section 4.1 at assumption 4. Especially by using the scheduler many standard distributed computing tasks are hidden from the application logic.

4.4 Theoretical performance considerations

To evaluate the model theoretical performance, there is considered a distributed system with N_C computing resources and a total of N_P processing units (fully independent cores). A number of N_J jobs ($N_P \leq N_J$), will be executed, each job requiring a maximum time T_J to complete. We define T_S as the time to setup a computing resource (deploy a unit to that resource):

$$T_T = T_S + \lceil N_J / N_P \rceil * T_J, \qquad (1)$$

where T_T = total computation time; $\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}$ (2)

when $N_P \rightarrow N_J$: $$\lim_{N_P \to N_J} T_T = T_S + T_J \qquad (3)$$

Here for all jobs the total run time is the run time of the longest task (in the case of heterogeneous resources) added with the computing resources setup time. This time is a function of factors such as the size of global static data and the unit, and network usage.

Because T_J can be equated with $T_J = T_{JN} + T_{JC}$ (4)

where T_{JN} is the needed time for interface/network run (dispatching the parameters of "run" to the running units and getting the results from them) and T_{JC} is for one processing unit the effective computation time, forthe best situation

$$(N_P \geq N_J): T_T = T_S + T_{JN} + T_{JC} \qquad (5)$$

Because $T_S + T_{JN}$ is a function of only the interface/network performance, for best results it is optimal that its percentage from the total computing time to be lowered as much as possible. The best case is if the T_{JC} of the tasks is much bigger than $T_S + T_{JN}$. In this case the distributed system uses most of the time solving the computation (T_{JC})

than on interface/network traffic. In this situation, the distributed computation performance closes to the performance of the case when all the jobs are executed locally using a N_J cores machine.

The above results are in accord with Amdahl's Law for speedups on fixed workloads [71], because when the number of processing units N_P reaches the number of jobs N_J, the speedup is limited to the individual execution time on each processing unit T_{JC}. In later developments [71] of the Amdahl's Law by Gustafson (fixed time), and Sun and Ni (memory bounded), in many practical situations if a distributed system receives more hardware, its assigned workloads also tend to grow, so the tendency is to use the system to its full capacity. In this way, by adding or increasing workloads, the system tends to be used to its maximum capacity, even for many computing resources.

4.5 Study and implementation

Our model was implemented by creating a dedicated virtual machine (VM) with the associated runtime. The runtime is capable to use the CPU cores in order to run the VM. Each machine has abstracted its computing resources by using a server for receiving **units** and tasks (instances of **unit**) to be executed on it. A client application can make requests to the servers available if multiple threads are needed to run. The entire process uses the following steps:

- The available servers are checked by the client: A network hosts list is used and each resource is interrogated about its version of the server, the available cores number and protocol.
- The application is run on client: We created a strongly typed, register based VM with automatic memory management and high level abstractions such as classes and functions. A portable VM used as a layer of abstraction between the host available capabilities and the application allows to use a wide range of computers, enabling both software and hardware independence.
- A VM which is register based is also important when threads are run on GPU cores. For a portable implementation, the GPU (programmable using OpenCL) must be coded in a kernel

function. The OpenCL uses a C/C++ language subset. In this case, all **unit** functions must be translated from the specific VM opcodes to the OpenCL language. In the case of a register based VM the translation job is easier to be done.

- If a **with** statement is executed, a scheduler is created by the runtime: This statement receives the constructor for **unit** along with its parameters and also possible options. The parameters of the constructor are serialized in the beginning, only once, because they are constants.

- When needed a scheduler create worker threads. Every thread manages a computing resource connection (for example a CPU core) available through a server. Permanently open sockets are used for the communication between the server and worker threads. A new connection object is used for each worker on the server side. Each such object is run on a separate thread so a CPU core will be used for each connection. In the beginning, all the **unit** code along with all its dependencies is made available on the execution server. Code caching is used on the server. By doing so, a **unit** is sent only once and it will be used for all the required instances. The **unit** constructor arguments are sent for every worker.

- When a **run** statement is encountered, it is added a new job: The list of all the tasks is kept by the scheduler. The tasks are asynchronously added. When a task is added, the scheduler queries for available workers. When no available workers are found, the scheduler tries to create new workers. The list of the servers available is used. The maximum number of workers is equal on each server with the available cores number.

- The tasks from the list of invocations are handled by the free workers: On server is created a VM instance at the connection of a worker. The **unit** initialization data is already on server so its constructor is called in order to create a new instance. Method overloading is possible because the worker also sends methods signatures along with their parameters. On server it is executed the **run** method. The VM instance is used to isolate the execution context.

- The **run** results are serialized by the server and returned to the worker. In case of processing errors (such as network errors), the task is put back by the worker in the tasks list and it is tried to be reprocessed. In case of successful completion, the receiver given in the **with** statement receives back the results. With this method, tasks are taken from the list by workers, run remotely and the results will be made available to the receiver. The workers run the tasks until all invocations are successfully processed.

- The **with** statement ends with a synchronization point where the completion of all tasks is waited by the scheduler. The process end is given by the empty tasks list and all workers on idle state. When all the tasks are done, the workers and the scheduler are ended and their resources are disposed. The resources allocated on the server and the network connections are also disposed or closed.

4.6 Experimental results

The implementation scalability and performance was tested in a 10 computers network and on a processor's local cores. We used the test program from Code 5.1 with PXHEIGHT=2000, which gives a total of 2000 tasks. The implementation in our VM was done in a bytecode of about 1.1KB. Each trial was run using a clean server. In every run the tasks setup was complete (no caching), so on every test the full application code was sent to the servers.

On every test the speedup was measured from the base case with a single core or a single computer in order to test the distributed computing system scalability and also the workload for every core. This was measured as the amount (in percents) of distributed tasks run on a specific core, in order to determine the capacity of the implementation to distribute the tasks evenly on all the processing units which are available.

The theoretical top scalability is reached in the case when the speedup equals the added number of cores/computers, compared with the case of using only one core/computer for the computation. The best distribution of the workload would be when all the tasks are equally distributed on all the processing resources, in the case of homogeneous networks, when all the resources have the same capabilities.

4.6.1 Tests using a computer network

A Wi-Fi network was used of 10+1 computers, each computer having a 2 cores microprocessor. We used all cores. The tasks were run using only on the remote machines. A computer was only used for the main application. In this setup all the threads were run in the same conditions. The test was started using a single computer. We added remote machines one by one on each step. The speedup results can be seen in Figure 4.3. The Figure 4.4 shows the workload for each core.

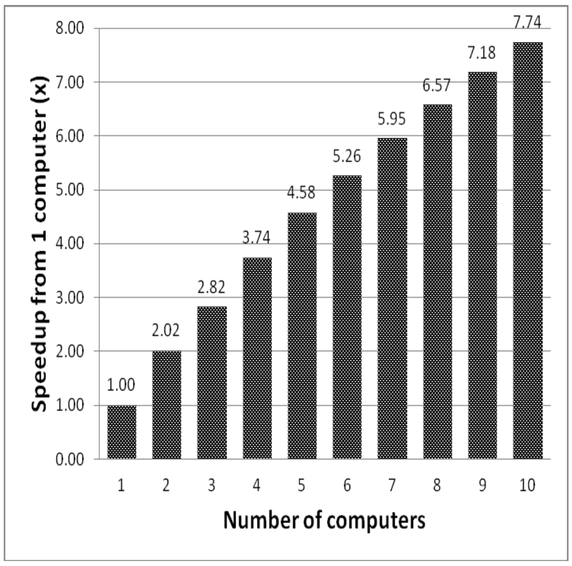

Fig 4.3 – Network speedup results

In the best case, the speedup is equal with the number of added computation resources, for homogeneous networks. In our tests, for a small number of computers, when the total computation amount on each machine is high, we obtained a speedup close to the best case when we added a new computer to the computation. For a higher number of computers, the traffic time and network setup, which are constants, start to contribute in a more significant percentage, so the overall speedup is lower. The theoretical model behavior presented in Section 4.4 is consistent with these results.

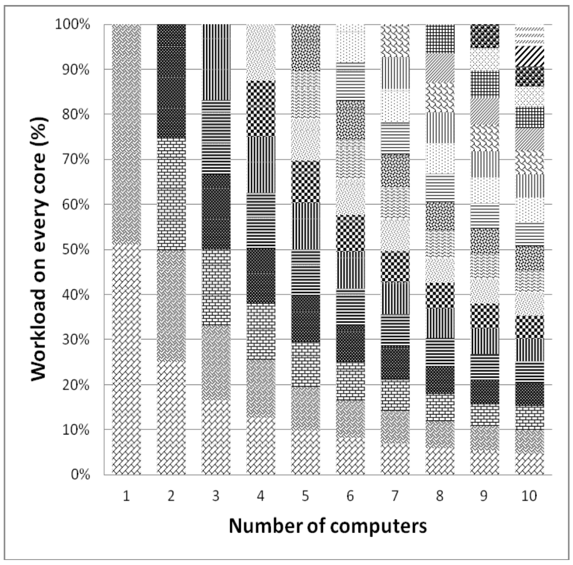

Fig 4.4 – Computer network workload results

A lower reliability Wi-Fi network was used, but even in this case the tasks allocation was very good and an approximately equal number of tasks were allocated on each core. The maximal difference in percents from the best distribution was 13.6%. For all tests the average difference in percents was of maximum 5%.

4.6.2 Computer cores tests

A 4 cores computer was used. The tests started using a single core. On each step we enabled another core. In Figure 4.5 are the results for speedup. The Figure 4.6 shows each core workload.

In these tests there was no network traffic involved (only the local microprocessor cores were used), so the theoretical model T_S+T_{IN} term is 0. There is required for threads synchronization only a small overhead. The speedup shows a linear growth for all the range of test cases and this growth is close to the best case.

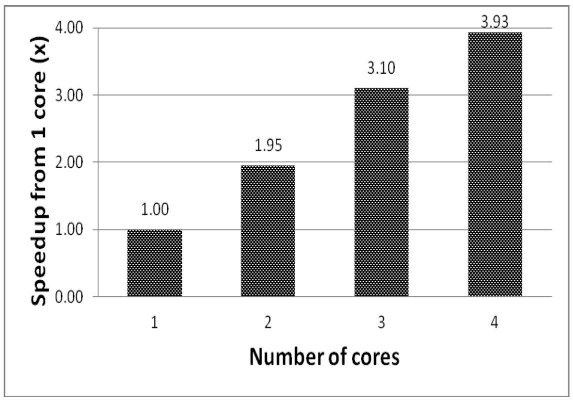

Fig 4.5 – Computer cores speedup results

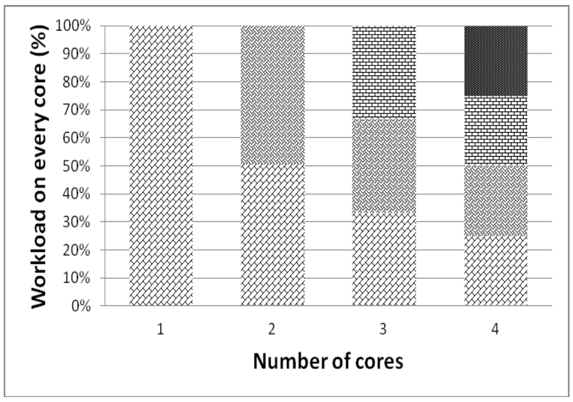

Fig 4.6 – Computer cores workload results

Even if the main application was also run on one core, the implementation distributed successfully the tasks on all cores in an almost optimal manner. The maximal difference in percents from the optimum in the test results was 0.6%. The average difference in percents from the maximum was 0.6% for all tests. Even if a core also needs to run the **with** scheduler, there was only a little difference from the best workload. This indicates that the scheduler most of the time waits for the completion of the threads, so only few resources are needed for itself.

4.7 Conclusions

We presented a model at the application level, suitable for dependable distributed computations. Our model requires only 3 concepts and it is a MapReduce derivative. The semantics of the model are similar to the OOP programming style and this allows an easy implementation of the model concepts in many of the mainstream programming languages.

We implemented the model using a specially developed VM. This implementation shows that using distributed thread pools for a scheduling system we can distribute the tasks on all the machines in a manner close to the optimal case. By doing so we can obtain a workload well balanced, both for network computers and for CPU cores. It was possible to abstract different resources (network computers or CPU cores) by using threads and client/server semantics.

The experiments show the scalability of the model, because it succeeded to make use of the added resources in a close to optimal way. The speedups obtained were close to the case when the computations were done by using multiple parallel programs.

In the case of network failures, the failed computations were run again so they are successfully completed, which makes the model dependable.

We consider further developing the research to include the usage of GPUs, to better enhance the reliability of the computation and the recovery from an extended class of possible errors. For the case of heterogeneous resources we also consider developing more advanced scheduling algorithms.

5. Application components distributed computing framework

In order to be of practical importance, our model should have an implementation as a language construct or as a framework or library. As a language construct an existing language should be extended with the necessary statements (unit, with, for). For languages such as Java, this means the completion of several steps such as: creating a Java Specification Request (JSR), JSR formal public review, final vote on JSR, creating a reference implementation, providing and testing on a Technology Compatibility Kit (TCK).

If the model is implemented as a framework, the above steps are not required since the language remains the same. In this case it is possible, due to the lack of the target language expressiveness or capacities, to have only a limited or harder to use framework, because the model specific constructs are implemented using only standard language features such as classes and methods calls.

We chose to implement our model in Java, due to several factors:

- Java is a mainstream language, widely used so the framework could be used by many programmers
- Java compiles to the Java Virtual Machine (JVM) which is independent of the operating system (OS) and CPU. In this way it becomes easier to implement the code deployment to remote computers, regardless their OS and CPU.
- Java offers strong reflection capabilities so a code can introspect itself and also create at runtime new code if necessary (by using bytecode generators). This introspection is very important when the framework computes all the dependencies of a unit which needs to be deployed and it also helps to check the consistency of the application code in regard to the model (for example to ensure that there is no accesses to outside data from a unit).

There are also some drawbacks which arise from the implementation of our model as a Java framework and we can mention the following:

- A framework needs to use existing language constructs to model new constructs or statements, especially through the means of classes/interfaces and methods calls. This can imply that the programmer need to adapt more

Section based on paper: Razvan-Mihai Aciu, Horia Ciocarlie, ”Framework for the Distributed Computing of the Application Components”, Advances in Intelligent and Soft Computing, Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, June 30-July 4, 2014, Brunów, Poland, Springer, ISBN: 978-3-319-07012-4 of the application domain to the semantic imposed by the language statements.

- In general the distributed applications are the ones which require every bit of computing resources, especially on the field of High Performance Computing (HPC). This is why for such applications are preferred languages which can optimize the code almost to the level of the assembly language, such as C/C++ or Fortran. For now, even with state of the art compilers, Java is behind these languages as execution speed. This means that an application written in Java will need more physical resources for the same level of computational throughput than a C/C++ or Fortran one.

We hope that the above mentioned drawbacks are not too big for many applications and our framework can be used in many of them. We view the proposed framework as a particular Java implementation of our model, which also demonstrates its validity and usefulness. There are other MapReduce Java frameworks, such as [22] and because the way of approaching computations (MapReduce) is similar, there are similarities between such frameworks. For example the automatic code deployment, intrinsic to our model, is also implemented. We regard our model as a higher order computation abstraction and we aim to add into our framework other features made possible by it, such as running distributed code on GPUs.

5.1 Framework overview

The framework consists from an application library and a server. The server is used on the remote computers as a mean to receive, run and return the results of the deployed code. The application library provides all the necessary code and data structures to implement the model in application. A pseudocode overview of how the framework is used is given in Figure 5.1.

```
(1)    var sched:Scheduler
(2)    sched=new Scheduler(computationClass, initialData, dest)
(3)    for job=every workload job
(4)        sched.addJob(destPosition,job)
(5)    sched.waitForAll()
(6)    combine_results()
```

Fig 5.1 – Pseudocode of the algorithm

A job is any task scheduled for run using a local or remote, abstracted computing resource. The *Scheduler* is a component of the framework, responsible for managing the low level aspects required for the distributed computing. The *Scheduler* at its initialization needs a class who provides the actual computation (*computationClass*), the constant global data who is required for all distributed computations initialization (*initialData*) and the results holder (*dest*).

The *addJob* method runs asynchronously. It adds a job into the jobs list. Each job consists of the needed data for its computation (*job*) and a destination abstract place (*destPosition*), used on results return. For the enqueued jobs the *Scheduler* creates worker threads, one thread connected to one computing resource.

A special server is provided by the framework for workers connections. Each computer used in computation has on it a running server. The jobs are taken from the queue by the worker threads and run on a machine. In this process the *Scheduler* will deploy first to the remote computing resources the required code (*computationClass* and the dependencies) and also *initialData*. The code and initial data are sent only once. They will be cached and used for all the jobs executed on that resource. The deployment of the code is made using a custom class loader. A serialization engine is used for data serialization. We used the Java standard serialization framework for our implementation.

After the scheduling of all the workload jobs, the method *waitForAll* waits for the completion of all the computations and the retrieval of all the results. The received results are combined as the application logic requires. These steps describe the entire algorithm. The aspects of a distributed application, such as code deployment, resource discovery, synchronizations and serialization are abstracted and they do not appear explicitly in the application logic. The computing resources are also treated in an abstract manner, so an application can make use automatically of any resource, like network computers or local microprocessor cores.

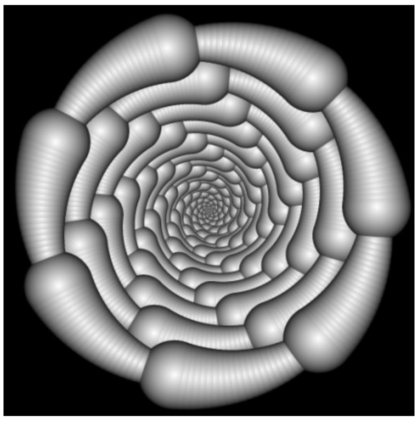

Fig 5.2 – An 3500 spheres image rendered by our test application

As an example, for the image from Figure 5.2 each line of the image is a job. We have 2000 jobs for an image of 2000 lines. The scene itself is the initial data (including view angles, observer position, output resolution). All this data is invariant. The jobs processing is done without a defined order, so the results (the image lines) are first stored and then combined in the final image in the right order. The *computationClass* is a derived class from a dedicated interface (framework provided), which handles the computation of one image line. This class including its dependencies is deployed to the remote computing resources. It will be computed employing a distributed manner. The *destPosition* indexes the job result into the vector of the lines of the image. The *job* is the data specific for each instance, in this example it is for every image line its vertical angle.

5.2 Framework detailed description

This description gives for each step of the algorithm an in-depth discussion of the available customizations and options, the support required by the framework and also implementation suggestions for specific platforms.

5.2.1 The setup of the network

Each distributed computing computer is running a dedicated server. These servers respond to queries about version and locally available resources. The maximum concurrent connections for each server are at most the same with its computing resources number. A worker thread is created by the scheduler only when a server with free connections is available. After a connection is made between the server and the worker, the connection is kept open until the completion of all tasks or until the occurrence of an exception. A worker is in this was assigned to a computer core. This way of allocating computing resources uses fully all the available resources and also minimizes the switching of the kernel threads. This algorithm is suitable for small or medium networks, up to around some thousands of simultaneous sockets. It also performs well for private networks, when there is no direct outside access.

In the case of larger networks can be used an alternate model, based on queries (pings) addressed to the servers with active computations, to assess the status and to retrieve the possible results. This algorithm has no bounds to the number of the open sockets, because the sockets exist only when queries are made.

To address volunteer, regulated or unsecure networks, we must observe additional requirements: communication encryption for server, application authentication, options to set bounds on the server resources usage, file system accessing security policies and a fine grained access for network or other sensitive functions or components.

5.2.2 The distributed application code

The distributed components of the application (the *computationClass* of Figure 5.1) need to implement the interface shown in Figure 5.3.

```
public interface Distributed<InitialData,TaskIdx,RunData,ReturnType>{
        boolean       dInit(final InitialData initialData);
        ReturnType   dRun(final TaskIdx taskIdx,RunData rData);
        }
```

Fig 5.3 – The interface Distributed

Java generics are used in the framework to enhance type safety. Each connection to a server creates a new instance of *computationClass,* so on a server can be a maximum number of *computationClass* instances equal with its cores number.

The *dInit* method is only once called, at the creation of the new instance. It returns true for a successful initialization. In this case the new instance can be used by the worker. For all tasks the *initialData* argument is the same. The instance created is used by the worker for all its computations. By doing so it is possible to keep the *computeClass* state information during multiple tasks, for example if partial computations caching is required. The number of tasks or their order is not specified for a specific instance.

For every job it is called the method *dRun*. In the destination the argument taskI*dx* is used to index/order a specific task. Some possible situations for taskIdx: a vector index or a map key. Every task must have a unique taskIdx. In our application the indexes of the image lines are used as taskI*dx*. For each job the argument *rData* is used to send specific task arguments. *dRun* returns on success an object newly created which encapsulates the computation result. An error is thrown on null return from *dRun*.

If a class implements the *Distributed* interface, all its data must be in instance attributes (it cannot have static variables), in order to enable the execution of the class on multiple hosts. At runtime the framework can enforce this requirement by analyzing (through reflection) the class members and all class dependencies.

5.2.3 The instantiation of the scheduler

The framework provides a *Scheduler* class. Figure 5.4 shows its signature.

```
(1)       public class Scheduler<InitialData,TaskIndex,RunData,ReturnType>
```

Fig 5.4 – The signature of the Scheduler class

InitialData, Task*Index*, *RunData* and *ReturnType* are generic parameters and they were detailed in section 5.2.2. There is both non-static constructor and a a static one for *Scheduler*. For automatic system initializations (like resource discovery) it is used the static constructor. These initializations are made only once, in the beginning of the application. It is possible to make subsequent checks for resources, because of the network dynamic nature, where computers can be removed or added at any time. These new checks can be triggered by the developer or performed automatically at specific time intervals. The non-static *Scheduler* constructor has the signature shown in Figure 5.5.

```
(1)        public Scheduler(final Class<?> distrClass,
(2)                         final InitialData initialData,
(3)                         Destination<TaskIndex,ReturnType> dest)
```

Fig 5.5 – The signature of the Scheduler class

The argument *distrClass* is the distributed class internal Java class representation. The distrClass code with its dependencies is serialized and sent to be run remotely on the available servers. The interface *Distributed* is implemented by this class, so it can be used in a standard Java way. If full reflection capabilities are available (languages such as C# or Java), the class description serialization and the serialization of the methods code is done by employing the standard reflection/introspection API for that language.

The argument *initialData* is the same and constant for all tasks and it will be used for each distributed worker initialization. It is sent only once to each server.

The argument *dest* is the distributed computation destination for the results. The *Destination* interface is shown in Figure 5.6. It abstracts a computation destination.

```
(1)        public interface Destination<TaskIndex,ReturnType>{
(2)                void    set(TaskIndex index,ReturnType retData);
(3)                }
```

Fig 5.6 – The interface Destination

When a *dRun* invocation ends, the result will be sent back. The method *set* of *dest* is called using the index of the destination and the result of the computation. The class used for destination can have a wide range of behaviors, as required by the application logic. When the results of the computation must be first available (like in our example), a wrapper over a collection class can be used as destination. When it is possible to use the computations results independently, the *set* method can encapsulate each result processing.

5.2.4 The workers and the jobs

A job is scheduled computation, waiting for its execution. The job is added to the list of invocations using the method *addJob* of the *Scheduler*, shown in Figure 5.7.

```
(1)        public void      addJob(TaskIndex index,RunData rData)
```

Fig 5.7 – The addJob method

The argument *index* is the index of the result in *dest*. The *rData* argument is the specific data required for a computation. The *addJob* method is asynchronously executed, so the application loop is not blocked during its execution. This method adds the job into the jobs list.

In the case when all the existing workers are occupied by other jobs and if more computing resources are available, *addJob* will create a new worker for the newly added job. The newly created worker is a *Scheduler* created thread that handles a specific resource (communication, serialization, etc). The jobs are not computed by workers, but a worker only sends the jobs to the computing resource receive the jobs results and send the results to the destination. With this algorithm only few resources are required by a worker thread and thousands of workers can be used.

When it is created, a worker locks a resource for itself. A *distrClass* instance is remotely created and the *initialData* is sent to its method *dInit*. During the worker's life that instance is kept alive. The worker will take jobs from the scheduled invocations and will send their data (*index* and *rData*) to the resource associated. On the computing resource, the *dRun* method of the *distrClass* instance is called with this data, the computation takes place and the result is sent back.

If the application logic causes errors by itself, exceptions are used to signal them back. If the network causes errors (or on other errors external to the application), first the worker tries to reconnect to its attached resource. If it fails, the worker signals the framework to verify the availability of the remote server. The remote servers which cannot be discovered anymore are removed from the list of the available servers. The workers which cannot reconnect will terminate themselves. In this situation the failed job remains into the jobs list and its reprocessing will be tried by other workers.

5.2.5 The distributed computations end

After the scheduling of all the jobs, there are two possibilities for the application to wait for the completion of the jobs. The straightforward way is to use the method *waitForAll* of the *Scheduler,* shown in Figure 5.8.

```
(1)          public void     waitForAll()
```

Fig 5.8 – The waitForAll method

The method waitForAll waits for all the jobs completion, including the jobs still in the scheduled list and the currently running ones. When *waitForAll* returns, all the jobs were computed and their results were sent to destination. Another way to wait for the jobs completion is to check manually for their completion. This can be done by using the *Scheduler* methods shown in Figure 5.9.

```
(1)          public int      getCompletedJobsNb()
(2)          public int       getAddedJobsNb()
```

Fig 5.9 – The methods getCompletedJobsNb and getAddedJobsNb

The *getAddedJobsNb* method returns the jobs number scheduled using the *addJob method*. The *getCompletedJobsNb* method returns the jobs number successfully computed. With these methods, the status of the scheduled jobs can be known by the application.

When the computation of all the jobs is done, all worker threads are stopped by the scheduler. The servers are signaled by the workers to free the resources and to end the instances created for these workers.

5.3 Practical tests results

The framework and the algorithm were tested on a quad core computer and on a computer network. Two metrics were evaluated. The first metric is the speedup obtained on the addition of new resources. This metric gives also a good estimation for a specific application if it is advantageous to employ more resources, also considering other factors such as the economic costs of these resources. The other metric was the distribution of the workload for every computing resource – by evaluating this metric we can assess the algorithm ability to fully use existing resources by distributing the workload on all of them, especially when heterogeneous networks are involved. We also tried in our tests different Java implementations and operating systems to evaluate the suitability of our framework. We used for all tests an application that renders the image shown Figure 5.2 using a 2000x2000 pixels resolution. Every image line is a job, so 2000 jobs were created. On each test a new server was used so there were no cached resources (such as remote classes) in order to achieve the same startup conditions.

5.3.1 Computer network tests

We used a 10+1 computers wired network, with CPUs Intel® Core™ 2 6600@2.40 GHz, with the 64 bits version of the Kubuntu 8.04 and Java HotSpot Server version 1.6.0_06. We used one computer only for the client application. The invocations were processed only on the network computers, so we can have a homogenous working environment for all the jobs. We began using one computer. On every step new computers were added, and we measured the speedup from the initial case of one computer. As we used dual core computers, we had in the end 20 cores on which the jobs were run. The Figure 5.10 shows the speedup. The Figure 5.11 shows the workload percent on each core.

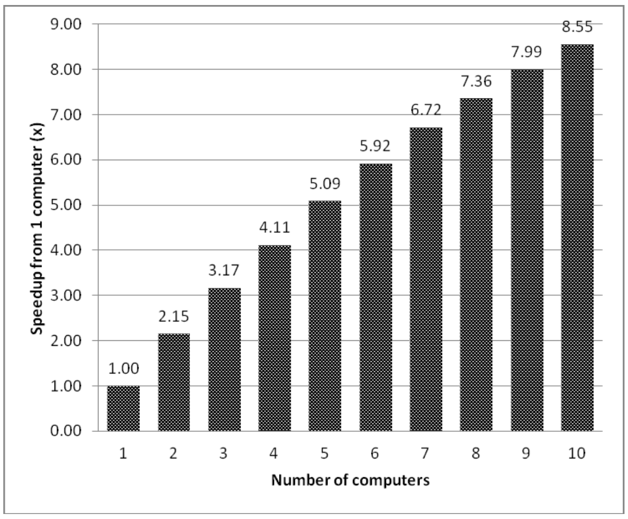

Fig 5.10 – Network speedup

We can see from Figure 5.10 that in the case of a low computers number, the obtained speedup is near the optimum. For 2-5 computers the experimental results indicate a performance which is a bit over the predicted model value. We consider responsible for these values external factors which can alter the small time intervals measurements, such as the network traffic variance.

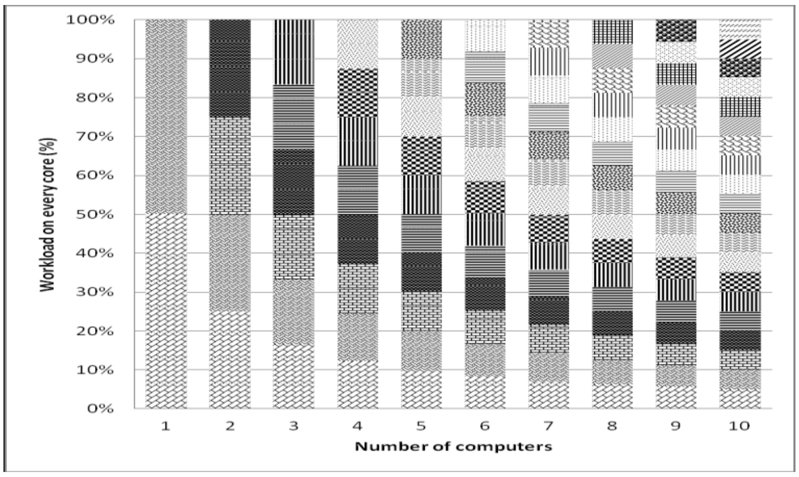

Fig 5.11 – Each core workload in network tests

If we increase the number of the computers, we obtain a lower speedup because the algorithm finishes quite quickly (around 1s). In this case factors such as threads and sockets management or resource discovery (the T_{JN} and T_S theoretical model components from section 4.4) accounts for a larger part of the execution time.

The distribution of the workload was near to the optimum (considered to be the case of equal invocations distribution on each core). The average difference in percents on all used cores was 1.03% at maximum for all tests. The difference in percents from the optimum was at maximum 2.8% for any core workload.

5.3.2 Computer core tests

A computer with a CPU Intel® Core™ 2 QUAD Q6600@2.40 GHz was used, running the 32 bits version of the Windows Vista Business Service Pack 2. The Java environment was provided by the version 1.7.0_11-b21 of the HotSpot Client VM. This is a four cores computer. The base case was when the invocations were allowed to execute using a single core. In every iteration new cores were added. In Figure 5.12 it is shown the speedup and in Figure 5.13 it is shown the workload on each core.

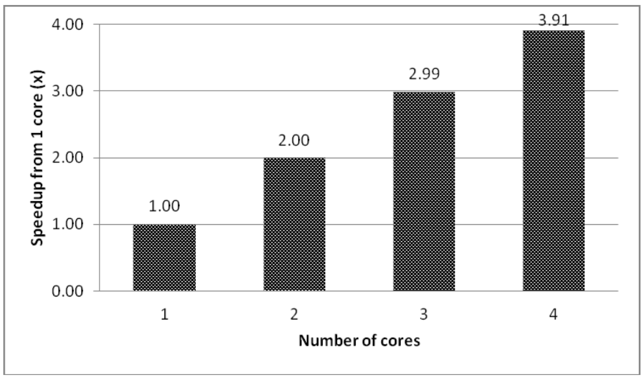

Fig 5.12 – Computer cores speedup

From Figure 5.12 it can be observed that the speedup when new cores are added is near the optimum. In the case of the 4th core we can see a bit larger difference (0.09%) from the optimum. That difference appears because that core needs also to execute the main application (including all the synchronization, serialization and scheduler code). That result indicates that all the threads (the workers) created by the scheduler and also other scheduler activities are using only a few resources. This is because a worker most of the time only creates jobs on other computing resources, waits for their completion and receive the results.

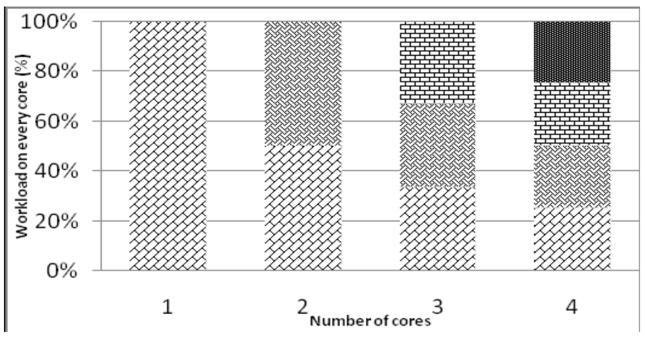

Fig 5.13 – Computer cores workload

The distribution of the workload among different cores has a difference in percents of maximum 0.6% from the optimum (especially because of the last core supplementary activities). The average difference of the workload in percents on all tests was 0.4% at maximum.

5.4 Conclusions

Our framework and the algorithm proposed enable the automatically usage of the application classes as components in a distributed computation. The resources like network computers or local CPU cores are abstracted by the framework, including the case of the heterogeneous networks, which allows the developer to employ them by using a uniform manner. Our framework can be used in a large variety of applications. The algorithm is well suitable for languages which employ virtual machines, like C# or Java. With certain restrictions the algorithm can be adapted for native code compiled languages which do not have advanced reflection capabilities, like C/C++.

The use of the framework is simple. Firstly the developer implements the interface *Distributed* on the class which is to be run distributedly. Secondly, the developer uses the scheduler to asynchronously add jobs. These are the only steps required for a distributed computation. All the required tasks like serialization, deployment, network management and synchronization are performed automatically by the framework.

We implemented our framework using the Java programming language. By analyzing the tests results it can be seen that when used both on network computers or on local cores our algorithm has a good scalability and it also achieves an adequate load-balancing. It does so by distributing uniformly the workload to each available computing resource. The framework run reliably and in all tests the results were provided even in situations such as the network errors occurrence.

The framework and algorithm open new directions of research. We consider further developing them to use GPUs, to increase the computation dependability (especially on external errors) and to better harmonize them with the standard API provided by the programming languages or by the industry standard libraries.

6. Algorithm for hybrid execution on both CPU and GPU

In this section we present a novel algorithm, suitable for the cases when some of its parts cannot run on GPU. Our algorithm allows an efficient split of the code segments between CPU and GPU. It collects the data for the GPU tasks across the CPU threads without stopping the CPU cores and it runs the collected tasks as a single package, in order to fully use the GPU. It is especially suitable for massive multithreaded applications with many threads, but where no individual thread can provide enough data to efficiently use the GPU.

As shown before, algorithms with complex code flows or data structures sometimes are not appropriate to be run on GPU, especially if they need I/O operations. Other algorithms, such as the sequential ones, do not benefit from a parallel execution on GPU, so they run optimally on the more powerful CPU cores. For such algorithms a mixed programming approach is more suitable to use. Some parts will be executed on CPU and others on GPU. If an algorithm part such as a matrix multiplication can have a parallel implementation, this part can be computed easily by a GPU. When the application works on multiple datasets, so it can benefit from a parallel execution but in itself each dataset handling is sequential, another approach should be used. In this case we propose our hybrid execution algorithm, which makes possible even for complex sequential algorithms to run some of their parts on GPU. By using our method, the suitable parts for GPU execution from the application algorithm are invoked in a special way so their calling data can be collected from all the application threads and run on GPU as a single batch of data. In this way any suitable part for GPU execution can benefit from the speedup offered by GPUs, even if this part is a component of an algorithm that cannot be run entirely on GPU.

A ray tracer for example has a high degree of parallelism and all primary rays can be processed each by using separate threads. A single ray computation is a serial algorithm, because for a given ray the next rays (reflected and refracted) can be found only after the given ray intersection is found. Due to the fact that many ray tracers employ data structures which are complex and also they in many cases need recursion [72], it is hard to use GPUs for full implementations of ray tracers which can be used in real world applications. In these situations it helps if it is possible to use the GPU only for some code parts, like the computations of the intersections, parts that are usually numerically intensive and which occur multiple

Section based on paper: Razvan-Mihai Aciu, Horia Ciocarlie, “Algorithm for Cooperative CPU-GPU Computing”, 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Romania, September 23-26, 2013, DOI 10.1109/SYNASC.2013.53

times during the process of rays tracing. With this approach, when a code designed to be run on GPU is reached by a thread, the thread will be put in a waiting state for other such threads to reach that code. For all these halted threads their data is collected and computed using GPUs. The threads receive back the computations results and their normal execution is resumed.

The fundamental aspect of the algorithm proposed is a method to run on GPU the code chosen by the programmer for this kind of execution, even if the invocations of this code are not executed in the same time and they are split among the application threads. In order to optimally use the CPU cores cycles, while a thread waits for others threads to reach the waiting point for GPU execution, each thread suitable for stopping needs to be executed using a single CPU core. In this case no thread context switching is necessary. In this respect we used cooperative threads (coroutines or fibers). Our algorithm also handles well the case when no other thread reaches a code designed for GPU execution, in the situation when there are no other threads to reach the same waiting point.

6.1 The proposed algorithm

The proposed algorithm runs on multithreaded applications by accumulating over CPU threads the invocations arguments of the functions which are to be run using GPUs, it runs the functions on GPUs and it resumes the CPU threads passing them the returned results. When the code reaches such an invocation, its data is collected and that thread is put on hold until the GPU computes the invocation. Meantime the application executes other threads, so the CPU is fully used. The programmer is responsible to choose the parts for GPU execution and to call them using a provided component, which will be detailed later. The process can be compared with a traversal using breadth-first order of a graph made from the active flows of code (threads). In Figure 6.1 are shown multiple CPU threads (T1…Tn), running the code ThreadFunction. A code part intended for GPU execution is reached (the F_GPU call), following possibly different flows of code. The T1 thread at F_GPU after executing the call to F1, and the T2 thread arrives at F_GPU after calling F1, F3 and F3. The functions F1…F4 are complex and they are executed using separate threads. The functions F1…F4 are better suitable for CPU execution (they may have recursion, I/O calls, complex lengthy code with multiple branches). Only the F_GPU function is intended to be executed on GPUs. In this case the programmer chooses GPU execution for F_GPU and he will invoke it in a special way.

To employ the computing power which a GPU can provide for massive parallel applications, we need to pause all the threads that reach F_GPU and collect the call data (the arguments) of the F_GPU invocations. After accumulating a suitable invocations number or when no other threads can reach the F_GPU call, the collected invocations are sent to GPUs that run a kernel which encodes the F_GPU function. All invocations are computed by the kernel, their results are retrieved to the threads paused and the threads execution is resumed.

When pausing CPU threads, it is advantageous to minimize the CPU cores context switching which appears when a thread is put on a holding state. In the case of preemptive multitasking, atomic operations and other mechanisms of synchronization must also be used to ensure the data integrity and to resume the execution of the paused CPU threads when the GPU execution ended. All the synchronization actions require CPU cycles which can be better used for the computation itself. In doing so, the synchronization overhead also reduces the applications types that may benefit from using the proposed algorithm.

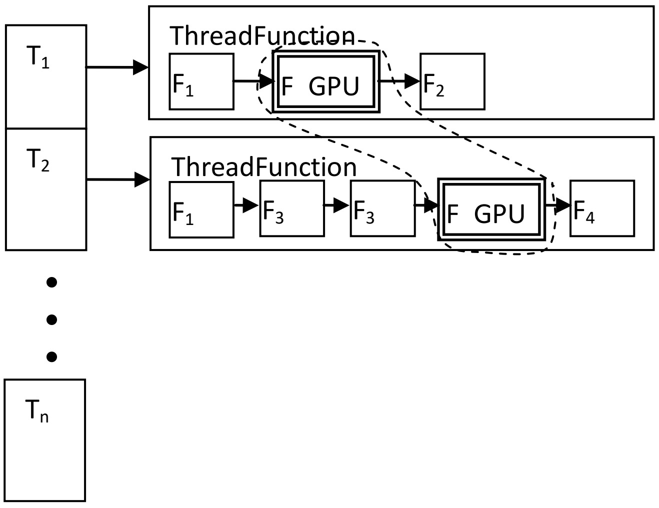

Fig 6.1 – Multithreaded flow of code for mixed CPU-GPU execution

To lower the synchronization and atomic operations incurred overhead [73], we employed cooperative multitasking. Its basics are implemented already by most OS like an API, for example the POSIX ucontext family of functions or on MS Windows such as fibers. Both these API allow the programmer to create a special kind of lightweight threads, which are cooperative (must receive explicit commands to switch between them) and incur a much lower overhead than the preemptive tasks. In the same time, by using explicit switching, many synchronization needs are eliminated, because the programmer knows at any time where it is the execution point and in a cooperative model only one thread is active at the time. The term "thread" will be used for preemptive threads; for cooperative threads we will use the term "fiber". The loop of the main algorithm is listed in Figure 6.2.

- For all GPU kernels initialize their schedulers
- For every CPU core create a new thread and add it to a threads list
- Iterate the list of threads in a circular manner and consider a current thread as given by this iteration

- For each parallel task (ThreadFunction):
    - Create a new fiber on the current thread and add it to the fibers pool of this thread
    - Advance the threads iterator
- When a call to a function designed to run on GPU is reached on a fiber, collect the call arguments, suspend the current fiber and continue the execution with the next fiber of that thread
- When a full circular iteration through all the threads is done, run all the collected call arguments on GPU using their specific kernels and resume the call fibers with the GPU provided results
- Repeat until there are no fibers left in any thread

Fig 6.2 – The loop of the main algorithm

The parallel running function is ThreadFunction. A thread is created for each CPU core, in order to fully use them. Each ThreadFunction is run in a new fiber. Each thread has a pool with its fibers. An iterator which iterates circularly through the list of threads assigns the new created fibers in order one fiber to every thread, to create a basic load-balancing of fibers in all threads. From the above considerations this main loop is almost the same as a main loop for multithreaded applications, with the only difference that we create fibers instead threads. This similarity with regular multithreaded applications makes easy to convert an application to hybrid CPU-GPU execution.

For any function (like F_GPU from the example) a special scheduler object is instantiated. Every scheduler is responsible to run an associated GPU kernel, which produces the same results as the original function. When a function designed to run on GPUs is called, instead of:

F_GPU(arg1…argn);

it will be performed through that function scheduler:

schedulerF_GPU.addCall(arg1…argn);

The scheduler which can compute the F_GPU function on GPU is *schedulerF_GPU*. Different from a regular function call, invoke does not execute the function but it will append the given arguments to the scheduler list of invocations. After that the scheduler yields the computation to the fiber which is next in the fibers pool of the current thread. By doing this the current fiber is paused at the function call, so the fiber can be resumed with the call result when it will be available. In the case when a fiber code (ThreadFunction) does not contain calls to functions to be run using GPUs, that code will run only on CPU until its end. After that that fiber will be removed from the pool of fibers and its fiber successor will be run. Figure 6.3 shows how all fibers are iterated

- While non-empty pool of fibers:
    - When reached a GPU designed call, add the call arguments to the scheduler of that function and yield the control for the next fiber
    - On a fiber end, delete it from the fibers pool
    - After each iteration of all the pools, run all the added calls using GPUs

Fig 6.3 – The handling of the calls to the functions designed to be run on GPU

After any iteration it is possible that some of the fibers will end. These are deleted from the fibers pool. The remaining fibers are the ones paused in waiting for GPU results. The schedulers with added calls will run them on GPUs, each scheduler running its own kernel. The results are returned to the fibers. The schedulers call lists are cleared so on the next iteration they will be empty. With this algorithm on any iteration all the GPU calls are collected and this ensures an efficient use of the GPU cores, by running simultaneously on them as many data sets as possible. The process is repeated until all the fibers are deleted from the pools. If a pool becomes empty, its thread is also ended.

In the proposed algorithm only the schedulers require synchronization, because more threads can access them concurrently. Inside one thread, its fibers do not need synchronization because they run cooperatively. With this algorithm, the only incurred overhead inside a single thread is the one needed by the fibers context switching.

From the above comments, the algorithm can be easily used in many situations with only a few changes in the application. This is especially true for code which is already multithreaded and a further GPU use is intended for it. Our algorithm can be applied to an already multithreaded application by following the next steps:

- Create schedulers for all the functions needed to run on GPU – mostly this only requires coding the kernels for these functions and passing the kernels source code to the schedulers in the initialization step

- The threads creation for the multithreaded code will be replaced with fibers creation. To optimally use the CPU cores, the application can use a threads pool and the fibers will be created circularly, one by one, in each thread, for load-balancing.

- All functions designed to run on GPU will be called through their specific scheduler, using the scheduler *addCall* method to accumulate the calls into the scheduler calls list

6.2 The performance of the algorithm

In this section we assess our algorithm performance and propose some situations for its optimal practical use. Let's consider an APP application that runs simultaneously on NC_{CPU} threads, with NC_{CPU} being the CPU cores number. Inside APP the function FN is desired to be run using GPUs. We wish to evaluate the speedup S_{APP} of the application if we use the algorithm to execute FN using NC_{GPU} GPU cores. The FN function needs to be called N times.

One FN call needs $T1_{CPU}$ time to be executed on CPU and $T1_{GPU}$ time to be executed on GPU. In the case of N FN instances, if we run them by using CPU cores:

$$TN_{CPU}=T1_{CPU}*\lceil N/NC_{CPU}\rceil \qquad (1)$$

with $\lceil x\rceil=\min\{n\in\mathbb{Z} \mid n \geq x\}$. The time needed to execute N FN instances by using GPU cores:

$$TN_{GPU}=T0_{GPU}+T1_{GPU}*\lceil N/NC_{GPU}\rceil \qquad (2)$$

$T0_{GPU}$ is a supplementary time required for GPU execution by the transfers between the CPU and GPU memory and by the translation of the data structures from the CPU format into the GPU format and back (especially when they contain pointers). The application requires a total time of:

$$T_{APP}=TN_{FN}+T_{OTHER} \qquad (3)$$

with TN_{FN} being the time needed to execute N FN instances and T_{OTHER} being the time needed for other components of the application. When FN is executed on GPU, the speedup of the application is:

$$S_{APP}=(TN_{CPU}+T_{OTHER\_CPU})/(TN_{GPU}+T_{OTHER\_GPU}) \qquad (4)$$

where $T_{OTHER\_GPU}>T_{OTHER\_CPU}$, because on the GPU execution case some more time is required by the APP for actions such as the GPU kernels loading and compiling. Taking (4) into account, we will analyze the involved factors, in connection with (1) and (2).

6.2.1 The GPU execution incurred overhead

If the CPU is used to run FN, its data and code are taken by the CPU from the memory of the computer (CMEM). After computation the results are put back into CMEM. If the GPU is used to run FN, in many cases its data and kernel need first to be transferred into the memory of the GPU (GMEM) from CMEM. When in GMEM these are taken and computed on the cores of the GPU and put back into the GPU memory (GMEM). In the end, the results are sent back to CMEM from GMEM. The transfers CMEM->GMEM->CMEM are not required for architectures with shared memory, such as integrated CPU+GPU.

Before the transfer CMEM->GMEM usually is required another step, for data fetching and translating from the application data structures into the GPU needed structures. An application may employ complex structures for its data and these structures must be transformed into a form adequate for execution on GPU. This requires iterating the structures, sometimes recursion and possible allocation for new memory to store the transformed structures.

After the application data was transformed into structures suitable for execution on GPU it is sent from CMEM to GMEM using busses like PCI Express (PCIe). The bus usage for data transmission can be a bottleneck with significant effects [74] for heavily multithreaded applications or when the data is transmitted in big amounts. After the GPU processing, the above steps are reversed: from GMEM data is retrieved to CMEM using busses and it is again iterated to transform it from the form in which the GPU produced it into a form suitable for the application logic. The transformation step may need recursion and memory allocation.

It can be seen that when a call is executed on CPU it needs only 2 transfers of memory (CMEM->CPU core and CPU core->CMEM). If the call is executed using a GPU, it needs 6 memory transfers (CMEM->CMEM (transformation step), CMEM->GMEM (over a bus), GMEM->GPU core, GPU core->GMEM, GMEM->CMEM (over a bus) CMEM->CMEM (transformation step)). The data transferred between different types of memory (CMEM->GMEM or GMEM->CMEM) also cannot be cached in the GPU or CPU cache memory and it makes its use even more time-consuming than the case of CMEM->CMEM transfers which can be sometimes cached in the cache memory of the CPU.

$T0_{GPU}$ is important when the called function (FN) needs just a few instructions to be executed. The overhead in this situation is similar with the needed CPU time to execute FN, particularly if the FN code and its data can fit on the cache memory of the CPU. For these small functions, their GPU execution will lower the performance of the application. It can be seen that it is a practical minimal instructions number (or cycles of the CPU) which can be considered as a lower limit for GPU execution of a function. Less than that number there is no gain resulted from an execution on GPU, because the possible speedup is lesser than the needed time for the transfers of the memory and for the supplementary code needed to transform the application data structures into a GPU suitable form. This minimum limit depends of aspects such as memory latency and speed, CPU cache size and speed. It may be experimentally evaluated. In this algorithm the developer is responsible to select the functions to be run on GPU. He can do this by taking into account many factors, especially the ones presented on section 1, when the differences between CPU and GPU were discussed. In the same time he can test the experimental results when a specific function is run on GPU.

6.2.2 The application threads number influence

Many consumer CPUs are built with fully autonomous 4-8 cores and capabilities such as SIMD, multistage pipelines, execution with out of order capabilities and also a CPU is better equipped with cache memory and has better clock rates than a GPU [75],[76]. Particularly if the function desired to run on GPU (FN) needs many branches, the core of a CPU can benefit greatly from capabilities like branch prediction. In the case of applications which use instruction sets like AVX or SSE when the CPUs are equipped with these instructions, the computing throughput of the application can be improved up to a 32x speedup (when using 256 bits AVX2 operands for bytes operations, like the instruction VPADDB that simultaneously adds 32 integers of 1 byte each). The main strength a GPU is its cores high number that ensures substantial parallel throughput especially for numeric computations, despite the fact that the GPU cores are organized in SIMD groups, so these cores are not entirely independent. If the function FN needs many branches, the cores of a GPU may by affected by branch divergence and this degrade further the performance.

From the above considerations, in many cases $T1_{GPU}>T1_{CPU}$, so for a smaller N, near of NC_{CPU} (and if SIMD instructions are used multiplied by their width) and considering the clock frequency difference between GPUs and CPUs, it is more useful to fully run FN using CPUs, not on GPUs. For example, when the application runs on a 8 cores CPU at 3 GHz, for float32 data and it makes use of an instruction set such as AVX with operands on 256 bits, the throughput of the computation is at the minimum the same with a GPU with a number of 192 cores clocked at 1 GHz frequency: (3 GHz/1 GHz) * 8 cores * (256 bits/32 bits) = 192 cores. That outcome is correct if there are not vector operations used on the GPU execution, so every core of the GPU executes 1 math operation/clock only. The smallest N for which the execution on GPU starts to be beneficial for an application is influenced by factors such as the code flow of FN, the ratio between the GPU and CPU clock rates, the relative complexities of their cores (aspects such as the cache memory size or hardware optimizations like out-of-order execution), instruction sets (availability of complex instructions such as cryptographic extensions) and their amount of cache memory. For example if the FN code has multiple branches, the code divergence on GPU will be increased, which will require some cores to wait until all the cores will reach again the same instruction, or until the running cores will end their tasks.

It can be seen from the above observations that in an optimal case for an application to run using GPUs, that application has a large threads number, it uses time-consuming functions (so the CPU execution time is bigger even if the overhead incurred by the execution on GPU is added) and the functions are arithmetically intensive. The definition of the arithmetic intensity is the performed operations number per memory transferred words.

6.3 The test application

We tested the algorithm with a C++ application in which we implemented the threads, scheduler and the fibers pools required by the associated framework. For the interface with the GPU (code compilation, data transfer and synchronization), OpenCL [77] was used, since it provides an open standard which is supported by the GPU producers. Our application uses a sphere to render on it a Mandelbrot fractal using a common illumination model, as described below. Figure 6.4 shows the resulted image. The selection of the code parts to run on GPU must be made by programmer. In this case, even if GPU can be used for all the rendering, we implement the process using 3 stages, so we can test the GPU-CPU collaboration:

- In stage 1 the CPU is used to cast ray traces for all pixels of the image. In the case of intersections, the point of the intersection is mapped into the surface coordinates (u,v) of the sphere
- In stage 2, computed on CPU and in the next test on GPU (to assess the differences between executions) the MandelbrotPoint function shown in Figure 6.5 is called for each (u,v) point to determine the color of that pixel
- In stage 3 the CPU is used to apply an illumination model for each pixel, so the incidence of the ray at the surface of the sphere is taken into account

These stages are particular to this application. In the general case, an application can have any number of stages and the CPU and GPU executions can be mixed in any order.

All pixels are concurrently computed. For CPU tests separate threads are used (optimized by using a threads pool) and the threads number is equal with the number of the CPU cores. For the GPU tests separate fibers are used. With this setup there is a one to one mapping between the threads/fibers number and the image pixels number. The sphere is placed in a way that all the visible area is covered. In this case the sphere is intersected by each ray trace, so all the above stages are executed for each pixel. That setup permits a more homogeneous result evaluation and it also needs a bigger time for computation, so the execution time differences can be shown better.

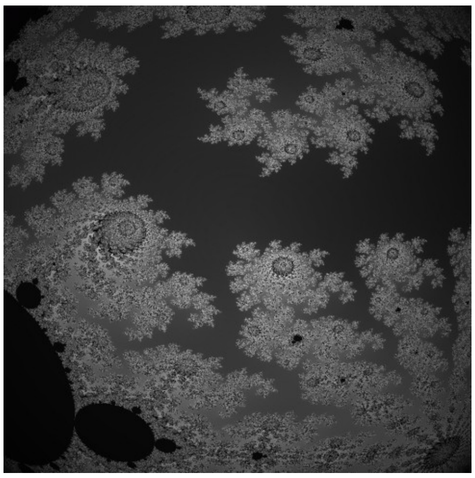
Fig 6.4 – The test application result

We used 3 different ways to compute stage 2: 1 CPU core, 2 CPU cores and GPU execution. Each way was tested on a separate run and we compared the results. The Mandelbrot fractal was chosen for the test because some reasons:

- Its GPU kernel implementation is almost identical with the CPU implementation, which allows for a more meaningful comparison
- MAXITER_MANDELBROT can be changed if we want to vary the maximum number of the possible computations done in each call

The kernel parameters are the point from the sphere surface for which the fractal will be calculated. The number of iterations in which the point convergence was computed is normalized to the interval 0...1 and it is used as a monochrome shade.

```
#define MAXITER_MANDELBROT        10000
float    MandelbrotPoint(float xp, float yp)
{
int      iter;
float    xiter, yiter, xtmp;
xiter = yiter = 0;
for(iter=0; xiter*xiter+yiter*yiter<2*2
                  &&iter<MAXITER_MANDELBROT;
                  iter++){
```

```
        xtmp = xiter*xiter-yiter*yiter+xp;
        yiter = 2*xiter*yiter+yp;
        xiter = xtmp;
        }
return (iter%256)/(float)255; //colors range normalization
```

Fig 6.5 – The function MandelbrotPoint

6.4 Experimental results

We used an E8200 Core™2 Duo Intel CPU at 2.00 GHz computer with a memory of 2 GB DDR2 and the 64 bit version of the Microsoft Windows 7 Service Pack 1. The GPU is a GTS450 Asus DirectCU Silent graphic card. This GPU has a memory of 1 GB DDR3 and uses the NVIDIA GF116 chip with 192 cores and clocked at 595 MHz. The GPU driver was NVIDIA WHQL v320.18 with OpenCL 1.1 support. For the tests on GPU, the compiling time for the kernel (~3 ms) is not included into the time measured, because this step is only once performed and it is relatively insignificant for most of the applications. The compiling time of the kernel can be significant for applications that run numerous times (such as web services), which use several large kernels without the facility to cache the already compiled kernels. In this case, the kernel compilation is repeated for each application run and it may require a significant amount of CPU computation.

To evaluate the relation between the computation time and the number of iterations performed, we computed an image of 200x200 pixels, varying the Mandelbrot iterations maximum number (the loop "for" of the Figure 6.5). The Figure 6.6 shows the results. We begun with 5000 iterations as the maximum and in each step we increased this maximum in increments of 5000, until we reached a maximum of 100000. The iterations numbers were chosen to allow a measurable increase in the computation requirements.

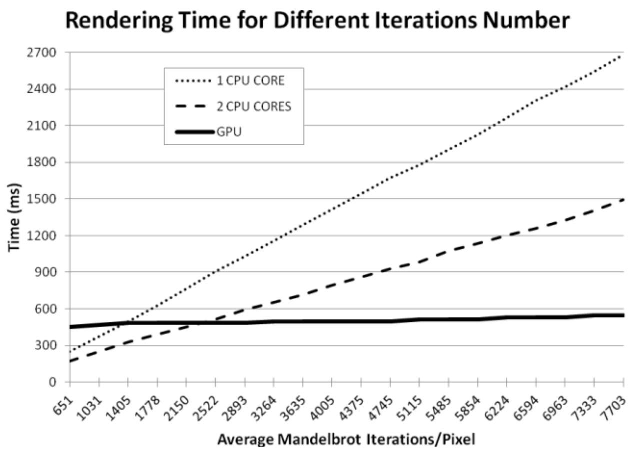

Fig 6.6 – Different maximum iterations number rendering time

The function MandelbrotPoint may need fewer iterations than MAXITER_MANDELBROT, so the total iterations effectively run was counted this was averaged for all the pixels of the image. In this way we obtained the average iterations/pixel effectively computed on average through the entire image. This metric allows a more objective view for the involved amount of computation. The relation between the MAXITER_MANDELBROT (maximum Mandelbrot iterations/pixel) and the iterations/pixel effectively computed is represented in Figure 6.7. It can be seen that for a limit of maximum 50000 iterations, the iterations/pixel effectively computed is 4005.

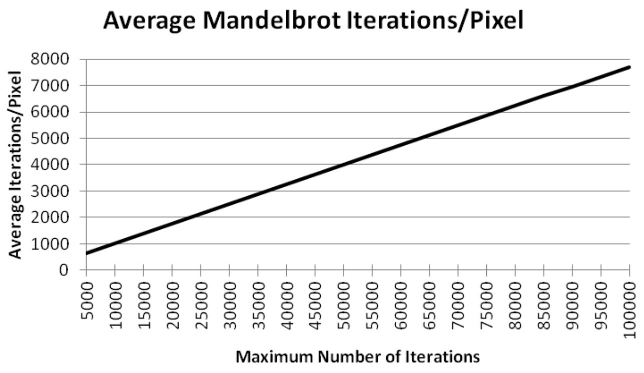

Fig 6.7 – Average Iterations/Pixel required by MandelbrotPoint

From Figure 6.6, it can be seen that when the number of iterations is lower (for 1 CPU core until around 1400 and for 2 CPU cores around 2300) it is more beneficial to execute the MandelbrotPoint function using the CPU. For higher numbers, the execution on GPU becomes more advantageous, since the fibers used (one fiber for each pixel) are spread to all the cores of the GPU (in our case 192 cores). That result confirms the Section 6.2.1 performance analysis, about the added overhead $T0_{GPU}$ impact for small values of $T1_{GPU}$. The added overhead impact is also represented in that figure by the time need for execution of the minimum workloads that is greater in the case of the GPU execution. Figure 6.6 is a good incentive to optimize the applications for execution on GPU, because across all the test range it can be seen that even when the total workload was bigger by ~11.8 times, the GPU execution time was increased by only ~20.1%, while in the same time the increase when the full CPU (both its cores) was used was ~774%. For the execution time of the application this is an important progress, achieved when a GPU was employed.

To assess the computation time dependency in regard to the total executed fibers, we computed images of various sizes. The maximum number of iterations was set to 70000. We begun with an image of 25 pixels width (resulting in 625 pixels rendered). The image width was increased with 25 pixels at each new step. There is a one to one dependence between the threads/fibers number and the pixels number. Figure 6.8 shows the results.

Considering the Section 6.2.2 performance analysis, in the case of a small fibers number, it is more beneficial to execute them using the CPU. The GPU becomes a more advantageous solution for an increased fibers number. When 2500 fibers were needed, the performance of the GPU (while using all its 192 cores) was better only by 2.02 times than the execution using the CPU (2 CPU cores). When 5625 fibers were needed, the difference between GPU and CPU increased to 2.19. Our analysis is consistent with these results and they demonstrate the computational performance difference between the GPU cores and the CPU cores, when the measurements include the additional overhead $T0_{GPU}$.

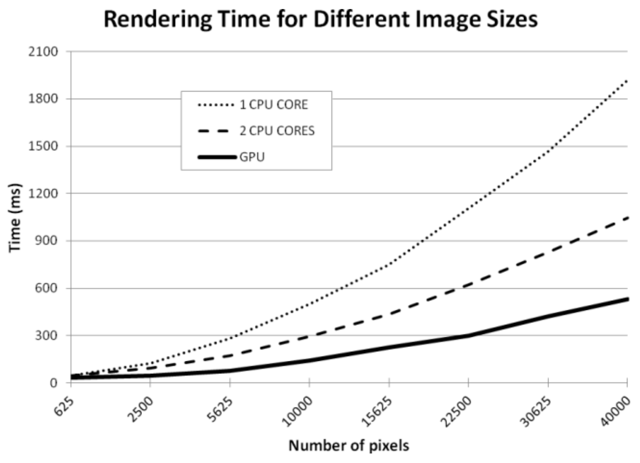

Fig 6.8 – Different image sizes rendering time

6.5 Conclusions

We presented in this section a new algorithm that enables the programmer to select for GPU execution only desired parts from a multithreaded code. That algorithm allows a cooperative GPU-CPU model of computation, which is helpful for several application classes. With this model of cooperative execution, even complex algorithms can be split in parts suited for GPU or CPU execution, so the programmer can choose for them the most advantageous computing resource.

The existent multithreaded applications can easily employ our algorithm, without demanding more restrictions or any other further limitations. In that respect the algorithm is suitable for the optimization of the existent applications for execution on GPU. The related runtime uses only standard APIs (for example fibers) existent in all the most used operating systems. This allows good application portability. The concepts used by the framework can be implemented by many programming languages.

The performance analysis of the algorithm highlights the most important factors implicated in the decisions concerning the selection of the most suitable computing resource both for functions and for applications. A programmer can use that analysis to accurately assess the impact of the GPU or CPU execution on the application performance.

The related framework is implemented in C++. The test application was run to evaluate different cases. From our results, it can be seen that an application can greatly profit from a GPU-CPU cooperative model of execution. The relative weight of the involved factors was also highlighted in the use of each of the two computing resources.

Our research can be further developed in directions like multiple GPUs usage, accurate algorithms and metrics to measure and to evaluate a function performance when it is executed on different resources, further enhancement and optimization of the related framework.

7. Java bytecode runtime translation to OpenCL and GPU execution

In this section we present a novel algorithm and library, which are capable to automatically translate at runtime the host application bytecode to OpenCL and to execute the resulted code on GPU. The algorithm is suitable for the cases when certain application modules can be run entirely on GPU. All the steps involved in using the GPU are automated: the Java bytecode translation, host data structures serialization into a GPU suitable format, GPU management and communication, results retrieval and their conversion back into the host data format. The library follows our proposed model and it abstracts the usage of the CPU and GPU. Both CPUs and GPUs are automatically used when available, without any specific settings in the source code.

Up to OpenCL 2.1 [78], which provides a high level language (subset of C++14) and also a standard intermediate representation (SPIR-V), the programs in OpenCL were defined using a subset of C99. To maximize performance, the OpenCL API is provided in C. The development of an OpenCL application involves the following tasks:

- implementation of the OpenCL code to be run on GPU
- conversion of the data to a format which can be used by GPU
- communication and synchronization between application and GPU
- GPU data retrieval followed by its conversion back to the format of the application

When we analyze the direct translations to OpenCL from the application bytecode, including data structures, we can notice that the process is a standard procedure. It can be addressed by special purpose tools and libraries. The translation of the Java bytecode into OpenCL is a problem of compiler theory and it can be solved by code generators which generate output for higher level language. Also the remaining tasks can be solved by specialized algorithms. The output code have the same behavior as the input, because it is only a new representation of the original code. It is not addressed the problem of creating advanced optimizations for the GPU code, but certain situations are still optimized. An optimization can employ intrinsics, translated directly to OpenCL constructions in certain situations. We

Section based on the paper: Razvan-Mihai Aciu, Horia Ciocarlie, ”Runtime Translation of the Java Bytecode to OpenCL and GPU Execution of the Resulted Code”, accepted at Journal of Applied Sciences “Acta Polytechnica Hungarica”, ISSN1785-8860, 2016 propose a library and an algorithm, which automatically handles the tasks discussed and which also simplify much the interoperation between application and GPU. This library implements when possible the above optimization so it generates OpenCL native instructions.

Code generation from application into OpenCL is advantageous because it is easy to employ and also it hides many of the GPU specific aspects. The developer does not need to implement special data structures. He also does not need to implement serialization, deserialization and synchronization to interoperate with the GPU. Custom kernels or libraries can be used if needed, and they can replace later the automatic generated code.

7.1 OpenCL code generation related work

We discuss here different libraries that convert to OpenCL fragments of their own code. Different algorithms can be employed for this process. For every method benefits and drawbacks are highlighted.

One method uses the original source code to generate OpenCL, by using preprocessor instructions. That method is employed for example by the Bolt library (C++) [79]. The Bolt library provides macros like BOLT_FUNCTOR that encapsulate segments of the original code, transforming it to text representations. The texts are glued to constitute the OpenCL output. No further processing is provided for the captured strings. These are used in the original form in which they are provided. The strings are combined using glue code. After that they are written in a suitable order. That procedure is straightforwardly applicable to languages with the same lexical and syntactic structure as the OpenCL, for example C and C++. We can note certain advantages, such as the fact that data structures are in the same layout in the application and in OpenCL. The exact layout of data can be enforced using alignment specifiers. This method considerably simplifies the interoperability between host and kernel. If the data uses a compact structure (for example without pointers to distinctly allocated structures), the data can be sent to GPU in its native form so there is no overhead incurred by serialization/deserialization. The code is also the same on the application and on kernel, which greatly simplifies debugging and also CPU fallback computation if no GPU is available. As some weak points of this method, we can mention the necessity that all the GPU code involved and its dependencies to conform to the OpenCL subset. On simple kernels this requirement is simple to accomplish. In cases with external dependencies, these dependencies need to be encoded in C/C++ restricted form, compatible with OpenCL. For the original code, all the dependencies which are not textually included in project need to be enclosed manually using provided macros. Other shortcoming is that the code provided at runtime (plugins, formulas, snippets) cannot be processed.

Another method implies explicitly building at runtime the computation needed using suitable representations, followed by OpenCL generation. Suitable representations can be different forms of Abstract Syntax Trees (AST). Employing standard procedures, OpenCL can be produced using the AST. Libraries such as ArrayFire [80] use that approach. This method provides to the application the possibility to run on GPU as a single kernel combined library functions, and so it eliminates the transfers between host and GPU, such when these functions are run individually. The AST leafs represent the data involved, interfaced by proxy adapters. AST nodes are operators and functions. Using operators overloading, the AST coding for expressions can be in many cases syntactically abstracted as expressions using infix common operators, also including their parenthesis and precedence rules. This method is very flexible. If the AST structure allow (when nodes can encode loops and declarations), any construction may be generated. Expressions provided at runtime or snippets of code can be compiled easily by parsing them and generating the associated AST. This method has some drawbacks such as the need to write manually the AST. In the case of small kernels, having only simple expression and combining only predefined functions, that task is easy. In the case of big kernels needing variables, loops and decisions, the AST encoding may be tedious and complex. As a different disadvantage, the AST representation is separated from the source code and it is not directly available for inspection. Because of this, the debugging is difficult and also the interfacing kernel/application is complex, even if predefined helpers are used.

Other method uses the programming languages reflection features. This allows the application to access its bytecode. That method is applicable in languages having sufficiently powerful reflection features, for example C# or Java. With this method, segments of the application code are decompiled and translated to OpenCL. A translation begins at the algorithm main method (provided by an interface exposing the main method of the algorithm to be executed on GPU). If this method needs other dependencies, these are also decompiled and added to the generated code recursively. The required structures are also translated and the live data is retrieved and encoded to the output format. That method is used in well known libraries, for example Aparapi [81]. The Aparapi library receives an instance of a class derived from the predefined Kernel class. The bytecode of this instance is translated to OpenCL. Aparapi uses the method *run* of a class inheriting Kernel such as the starting point. The required data is also serialized and deserialized. That method can be applied on many situations and it is possible to automatically handle different OpenCL related tasks. Also, the resulted kernel is a direct translation of the application code which adds certain benefits, for example ease of debugging (the application can be debugged using standard Java tools), also the possibility to employ CPU fallback computation if no GPU is available. Other benefit is the capacity to translate specific constructs to OpenCL optimized forms. The Java Math class has many methods with corresponding OpenCL primitives. Other commonly used primitives, for example dot and cross products may be added. That method also allows plugins with GPU execution to be loaded at runtime. A shortcoming of these methods is given by the increased time needed for kernel generation due to the overhead produced by the code disassembling. This can be alleviated caching the generated kernel for later reuse. In the Aparapi particular case, for now it is implemented as a thin layer on top of OpenCL, so it uses specific functions like *getGlobalId*. Even more important, Aparapi does not process data structures which have objects (it supports only primitive types, single dimension arrays), so complex data structures must be handled manually by implementing proxy code. The support for reference types was planned on certain architectures, for example Heterogeneous System Architecture (HSA).

A more integrated approach is the Sumatra OpenJDK [82] project. Its primary goal is to enable the applications written in Java to use GPUs and other devices from this class. The project Sumatra tries to translate specific APIs like Java 8 Stream into HSAIL (HSA Intermediate Language). HSAIL will be next converted for concrete architectures, (CPUs and GPUs). An important specific aspect when compared with the previous methods is that in this case different OpenJDK components are employed (custom compiler and virtual machine) for the GPU interconnection tasks, making the GPUs first class citizens of the targeted architectures. As of this writing, Sumatra project is mainly in proof of concept stages. It also depends by the adoption of new technologies (HSA). If it will enter production, this project may be an important step forward on the heterogeneous execution of the Java applications.

7.2 Our proposed algorithm and library

The algorithm uses runtime reflection in order to access the application bytecode, followed by disassembly, analysis, code generation, in order to generate OpenCL from the relevant code. In the execution phase required data is automatically serialized and transferred to GPU. Our algorithm also handles GPU synchronization and results retrieving, followed by a conversion to the data structures of the application. As the main goal, the GPU execution is abstracted as much as possible. A single code could be executed on CPU and GPU, and this allows easy debugging and it also provides execution fallback on CPU if no available GPU. The library generates OpenCL code for cases of medium complexity, such as structures with reference types (class instances), exceptions and dynamic management for memory. With these features we create new possibilities to run code on GPU, when compared with Aparapi like libraries. These libraries are mostly thin layers on top of OpenCL, with OpenCL API dependencies which need to be explicitly called by the programmer. Our abstraction layer may cause some loss of performance. In the cases where better optimization is needed, the developer can code key algorithms using a more optimized form for execution on GPU.

The algorithm starts with a tasks scheduler that enqueues the user provided tasks, run the tasks using the GPU and receives their results. We enhance [83] the Java thread pools API with an MapReduce model [20] semantics, asynchronous and event driven handling on the receiving side, in the same way as Node.js [84] non-blocking operations. Our library makes available distributed execution using different resources like GPU, CPU or network computers. When results are sent back, they are processed by the overridden set method of an object that implements the interface *Destination*. To this method are sent the results and a identifier unique for each task, like an index into an array. A *Destination* may abstract relatively simple collections like maps and arrays. A Destination can also implement advanced processing, by using the data immediately. With this approach, when it is allowed by the application algorithms, the retrieved results may be directly processed on arrival without needing to store them first. The scheduler begins running the jobs when they are added. After the addition of all the jobs, it is called a synchronization method to ensure that all the processings are ended and the results are available.

One task is represented like an object that implements the interface *Task*. The only method of this interface is *run*. A task encapsulates all the required data. The task is created in the host (application) side and enqueued asynchronously for execution on GPU, using the *add* method of the scheduler. That method associates also a unique identifier (UID) for each task. The UID will be used when the computation result will be processed. Task data is serialized, then sent to GPU. After the result is computed and returned from the *run* method, it is converted to the format of the application and sent to the scheduler destination. One generic use is shown in Figure 7.1.

```
// the class Result is defined by programmer and it encapsulates the result of a task
class ResultHandler implements Destination<Integer, Result>{
        // arguments: task UID, data received from computing resources
        @Override public void set(Integer id, Result ret){
                // received result processing
        }
}
class TaskHandler implements Task<Integer, Result>{
        public TaskHandler(/*in task specific parameters*/){
                // the initialization of one task on the host side
        }
        @Override public Result run() throws Exception{
                // runs the task using the GPU; returns the Result to the application
        }
}
// ... library start...
// instantiation of the Scheduler
Scheduler <Integer, Result> scheduler=new Scheduler<>(new ResultHandler());
```

```
// the creation of tasks and their addition to the scheduler queue
for(/*all input data*/){
        TaskHandler task=new TaskHandler(/*data specific for each task*/);
        scheduler.add(id, task);   // add asynchronously the task to queue
}
scheduler.waitForAll();            // wait until all tasks are completed
```

Fig 7.1 – One generic use of our library

In simple cases like the processing of the Java standard collections, implementations for the Destination class are provided. From the above considerations and example, the processing resources are fully abstracted. On the *Task* implementation and on scheduler no OpenCL specific instructions are used, but only general concepts, like the task UID. That allows a better modeling of the domain of the application; it better abstracts the computing resources and an easier porting to other architectures. Other specific calls are provided only for special cases, for example when multiple resources exist and we need to select a specific one.

When new jobs are enqueued, the scheduler first checks the case when it already has generated the OpenCL code of that task. When the OpenCL code was already generated it will be used, otherwise the host code for the task is loaded via reflection. The bytecode is disassembled and the OpenCL corresponding code is produced. The instances data of the tasks are serialized. To accomplish that, their fields are checked by reflection. Their content is copied in a buffer. This will act as the global heap of the OpenCL code. The serialization and disassembly for both code and data are treated in a recursive manner. They continue running until all dependencies are processed. When the GPU ends running the code, the data from the (modified) heap is sent back to host and deserialized, updating the application data. In this way the modifications made by the running kernel are propagated back to the application. It also makes available to the application the structures allocated by the kernel.

7.2.1 Serialization and retrieval of the data

The space needed for the global heap, serialized data, and dynamically allocated memory is provided in one buffer named the OpenCL heap or global memory. The references to data are translated into offsets into heap. A kernel uses heap relative indexing to accesses data, in the same way as array accesses. That method has an impact on performance when the GPU native instruction set does not have indexed access. If this is the case, an instruction will need to add a particular data offset at the heap base pointer. Solutions such as employed for Sumatra project may utilize shared virtual memory (SVM – introduced in OpenCL 2). In the Sumatra context this can be done because the JVM implementation is known (OpenJDK), so all its specific capabilities and structure can be employed. On a general case, a memory layout is not specified by JVM for many data structures, so the SVM cannot be directly used.

Our serialization algorithm directly copies the primitive JVM data types. One aspect is when the GPU has a different endianness from the host. In this case a conversion must be made. The values handled by reference are handled according to two situations: arrays and class instances. The first member is in both cases a UID for that array type or class. In the cases of the class instances, their references and primitive values are added on the reflection order. All classes are implemented using OpenCL structures. The class instances are accessed and created through their corresponding class structures. The array length for arrays follows after UID and after it the array elements. The primitive type arrays are separate types. The reference types arrays are implemented using only one array type (*Object)* and employing type erasure. To be able to know the heap offsets for all instance members and array elements at the serialization of the enclosing instance, the members are serialized first, in a recursive way. The static members of the classes are also stored on heap and they are accessed using the global context.

7.2.2 Code generation

There are specific JVM features that do not exist in OpenCL, for example exceptions throwing/catching, dynamic allocation, recursion, virtual method calls (non-final methods). These features must be made available as a layer over the OpenCL supported functions. Features like host functions calling are not available for now in OpenCL and it is no way to do this apart of ending the kernel execution, calling the function from the host and running again the kernel from the last ending point. This prohibits the usage of I/O functions, effectively blocking the use of the APIs like network or the file system.

In the first step, it is simulated linearly (without branching or looping) the JVM bytecode for each method, using a symbolic stack. In this stack a cell represents an AST node. The bytecodes which operates on stack combine the operand nodes into new, result nodes, which will also be pushed on stack, simulating the action of the operator. For example the bytecode for *constant push* creates an AST leaf for one constant value or a bytecode for *addition* creates the AST addition node, combining the two top AST nodes from the stack. Different nodes are created and operated on by instructions that do not update the stack, such as the opcode *goto*. When this step is ended, there is a distinct full AST constructed for any method. By traversing these ASTs we generate the OpenCL code.

A complex topic is the memory allocation. It is important particularly in the case of automatic memory management, and especially when there is no value semantic for classes, which made possible only dynamically allocated class instances (if the compiler does not optimize the allocation). On environments with a large number of threads like in the GPUs case, where several thousand tasks may run simultaneously, memory allocators specially designed are very important, otherwise they will constitute performance bottlenecks [85]. For this implementation we created a lightweight and fast memory allocator, on the same principles as [86], which for now is capable to only allocate data. By using this approach we obtained a high throughput on allocations (a single atomic operation required for heap access serialization) and there is no memory overhead associated with the blocks allocated. For now the allocator does not free the non-referenced memory, so a developer needs to be careful on the allocations number. When many memory allocations and reclaiming cycles are needed, this may be a serious problem and the allocator should be completed with garbage reclaiming. From the programmer point of view, considering that the situations when execution on GPU is required are mostly situations when high performance is needed, simple methods may be employed to reduce memory allocations. These methods also optimize for CPU and GPU by lowering the allocator pressure. In our OpenCL interface library, we reduce the allocations number by reusing already existent objects. To be able to do this, we made the operations on objects (like vectors addition) to use for result the first argument of the addition function. In this way we avoid creating new result vectors. Because OpenCL does not have global variables, every function needs to access the global variables (in this case use the base of the heap address) and this is accessed through a supplementary argument for each function. There is no overhead associated with the function call (the OpenCL compiler inlines all the function calls so no code is required for parameters passing).

Even in newer versions such as 2.1 (based on a C++ subset), there is no exceptions handling in OpenCL. In Java, even if the developer does not use exceptions in the GPU executed code, the exceptions may originate in the memory allocator on out of memory cases. Because facilities like stack unwinding are not supported, the exceptions were implemented using the functions returned values. Each function returns an integer that is a heap offset. For the 0 value (associated with Java null pointers), there is no generated exception. If exceptions are thrown, the exception objects are allocated and the index of the allocated exception is returned. All function calls are guarded against non-zero returned values. When this case takes place, the enclosing functions exits immediately, further propagating the exception received. When there is no catching block for the exception on any level, the kernel will end with the exception object returned to application like that particular task result. In order to have a valid exception object for out of memory cases, an *OutOfMemoryError* exception object is preallocated. Because the functions returns propagate exceptions, if a function has a non-void return type, the returned value is stored using a supplementary parameter transferred by reference that is added by the compiler to all non-void functions. That parameter points to the variable that will receive the value returned. The *return* instruction uses this parameter to store its expression at the referred address. We studied the GPU machine code generated from the produced OpenCL kernel to assess the impact of this implementation decision. As the OpenCL compiler inlines all its functions, the pointer indirection required to store the returned value was simply replaced with direct stores, so this implementation did not added supplementary processing. Similarly, the checks for non-zero returns and early exit of the enclosing functions were propagated up to the original exception point (the out of memory check inside the memory allocator), followed by an immediate exit from the kernel when the user code does not use *try…catch* statements. In conclusion the compiler optimized away this checking.

Some Java core classes like *Object*, *Math* and *Integer* are handled as intrinsics. That allows a more optimized code generation, which uses the predefined functions in OpenCL. Several *Math* functions can be directly translated into native functions, because they are defined already in OpenCL. Some available OpenCL functions, like cross and dot products were implemented in an auxiliary intrinsic library. The calls to these functions are treated as intrinsics. When the tasks are not run using a GPU, this library uses automatically a pure Java implementation, so it will run on any JVM. In Figure 7.2 there is a code generation example. We show the Java method we used to compute a Mandelbrot fractal and in Figure 7.3 (enriched with some comments and also formatted to reduce the lines number) we show its OpenCL generated code.

```
int mandelbrot(float xp,float yp){
        final int ITERATIONS=256;
        float xiter=0,yiter=0;
        float xtmp;
        int iter;
        xp=translate(xp, 1.2501276f, 3.030971f, 0.31972017f, 0.34425741f);
        yp=translate(yp, -2.9956186f, 1.8466532f, 0.03119091f, 0.0572281593f);
        for(iter=0;xiter*xiter+yiter*yiter<2*2 && iter< ITERATIONS;iter++){
                xtmp=xiter*xiter-yiter*yiter+xp;
                yiter=2*xiter*yiter+yp;
                xiter=xtmp;
                }
        return iter;
}
```

Fig 7.2 – The Mandelbrot method – original Java version

For overloaded functions, to generate multiple C names starting from the same Java name, we created a name mangling algorithm, because the original JVM name mangling has characters that cannot appear in valid C identifiers. This mangling algorithm was also needed to make the difference between similarly named functions in different classes. In our system it is appended to a function mangled name the context (class name and package) and the signature (the arguments types).

```
// _g - pointer to heap
// _l - pointer where the return will be stored
// return value: 0 on no exceptions, a heap index for exceptions
// idxtype - integer type used heap indexing (unsigned int)

idxtype   tests_D_T3Work_D_mandelbrot_LP_FF_RP_I
                              (_G _g, idxtype _this, float xp, float yp, int *_1){
int      ITERATIONS, iter, _TV8;              // temporary variables: _TV*
float    yiter, xiter, xtmp, _TV5, _TV6, _TV7;
idxtype _0;              // for exceptions testing and propagation
ITERATIONS =256;
xiter=0.0; yiter=0.0;
// all function calls are checked for exceptions occurrence
if((_0=tests_D_T3Work_D_translate_LP_FFFFF_RP_F(_g, _this, xp,
        1.2501276,3.030971, 0.31972017,0.34425741, &_TV5))!=0)return _0;
xp=_TV5;
if((_0=tests_D_T3Work_D_translate_LP_FFFFF_RP_F(_g, _this, yp, -2.9956186,
        1.8466532, 0.03119091, 0.0572281593, &_TV5))!=0)return _0;
yp=_TV5;
iter=0;
goto _TMP172;
_TMP173:;
_TV5=xiter*xiter;_TV6=yiter*yiter;_TV7=_TV5-_TV6;_TV5=_TV7+xp;xtmp=_TV5;
_TV5=2.0*xiter;_TV6=_TV5*yiter;_TV5=_TV6+yp;yiter=_TV5;
xiter=xtmp;_TV8=iter+1;iter=_TV8;
_TMP172:;
_TV5=xiter*xiter;_TV6=yiter*yiter;_TV7=_TV5+_TV6;
// to implement the JVM FCMPG opcode, FCMPG macro is used
_TV8=FCMPG((float)_TV7,(float)4.0);
_TV5=_TV8>=0;
if(_TV5)goto _TMP177;
_TV8=iter<256;
if(_TV8)goto _TMP173;
_TMP177:;
*_1=iter;              // set the return
return 0;              // no exceptions
}
```

Fig 7.3 – The Mandelbrot method – OpenCL generated code

In OpenCL there are no functions pointers or any means for indirect calls. In this situation, the virtual (non-final) functions (implemented commonly with virtual tables and pointers to dispatch functions) must be implemented differently. In this implementation the first member of each class structure (the unique class identifier), can be used in this purpose. That identifier is used as an index to a vector maintained on host containing the structures for the generated classes. That field for each kernel maintains its value on any actual heap serialized data. By using that id a virtual function may be implemented checking the id of the current object using a *switch* statement with dispatches for every situation possible (for all classes in a specific hierarchy that provide an implementation for that function). Similarly can be generated dynamic dispatch for all implemented interfaces. Our generator for now does not implement fully dynamic dispatch (virtual methods) and we can only use the Java final classes, in which the compiler can infer the specific method called.

Because the GPU do not has an execution stack, recursive calls are not implicitly supported [87]. A programmer must use an iterative version of an algorithm or the recursion should be implemented with an explicit stack. Our library in this version does not implement code generation for recursive calls. We consider that a limitation that can be solved so we are trying to implement recursive calls in the next versions.

7.3 Practical results

To test the algorithm and library, a Java application was created which renders an image using ray casting. It sends primary rays, without reflections or refractions. We used a test scene made from 1301 spheres. Every sphere has the Mandelbrot fractal drawn as a procedural texture. In this case every pixel is computed on-demand, without using precomputed bitmaps. Along with the procedural texture, on each point an illumination model is computed. This model considers the intersection angles between the rays and the normals of the spheres on the intersection points. The horizontal lines are treated such as separate tasks (work-items). Figure 7.4 shows the final result. Java features like classes, static members, members of reference types were used. All calculations are done using FP32. The Java Math library has especially FP64 operations, and it was needed to write a wrapper for the Java provided function, to have a FP32 version. Because of this, different operations such as cos*()* or sin*()* are executed on CPU as FP64 and on GPU as FP32. It was assumed that in the real world scenarios with FP32 data use, the developer does not convert it to FP64 for cos*()* or sin*()*, but the developer will use FP32 if possible. In the OpenCL code these operations become intrinsic functions. Specific OpenCL or CPU features were not used. The application ran without modifications on GPU and CPU. Our library uses in this version for manipulating Java bytecode the ASM v5.0.3 library [88]. Standard OpenCL bindings are provided by JOCL v0.2.0-RC [89].

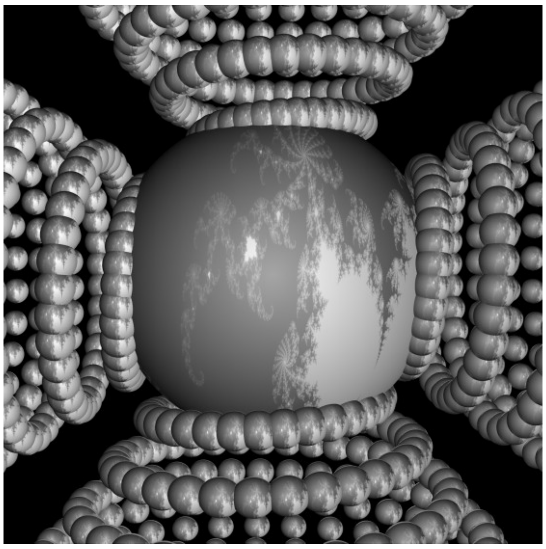

Fig 7.4 – The test program result

We used the following configuration for tests:

- a computer with CPU Intel® Core™ i5-3470 at 3.20 GHz, 8GB RAM, Java SE 8u45 and Windows 7 64 bits Home Premium SP1. The CPU has 4 cores.
- GPU AMD Radeon™ R9 390X, 1060 MHz, 8GB GDDR5 RAM. The GPU has 2816 streaming cores with 44 compute units.

We observed three main aspects: the GPU execution compared with the CPU execution, the GPU handling of different workloads and our library compared with the library Aparapi. For the comparison between CPU and GPU we used square images. We linearly increased the number of pixels to determine the most suitable method of execution for different sizes. All tests were run 5 times. The average value was considered. The GPU time sums all the implied times: the kernel generation and compilation, the serialization/deserialization and execution time. The execution time includes data transfer between GPU and CPU. This GPU total time is required only in the worst case, if the computation runs once. If the task code runs multiple times, its kernel may be reused (by caching it). In this case the times for generation and kernel compilation are insignificant. These times were measured over the entire test range. In Table 7.1 are given the results.

	Image of 10 KPixels			Image of 43000 KPixels		
	Heap data (KB)	Heap total (KB)	Time (ms)	Heap data (KB)	Heap total (KB)	Time (ms)
Kernel generation			22			23.3
Kernel compilation	62.8	111	224	44442	45490	228.5
Serialization			11.3			55.4
Deserialization			0.65			100.5

Table 7.1 – GPU - Data sizes and setup times

The Table 7.1 shows that the kernel generation and also compilation are invariant with the workload due to the fact that the processed bytecode is identical. The times of serialization and deserialization increase when more data has to be processed. On small workloads we have the result in Figure 7.5. The measurements were made in steps of 10 KPixels (KP).

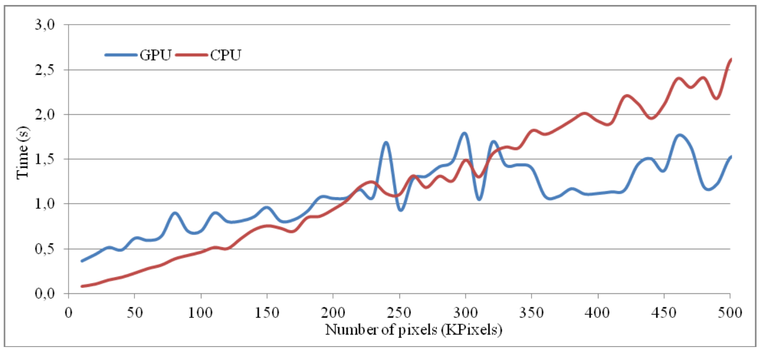

Fig 7.5 – CPU vs GPU execution on small workloads

The execution on GPU begins to be more efficient around 250 KP. When the times needed for generation and compilation are subtracted, this threshold is near 100 KP. In our case, the worst case for GPU is the lower number of pixels, because a square 100 KP image has around 316 pixels height. In this case at best maximum 316 streaming cores on GPU are used from the total of 2816 streaming cores.

Maybe a good thing in this situation is that the compute units run fewer work-items simultaneously, which improves the divergence.

We compared next the CPU vs GPU execution over the entire test range. On the test system a GPU run is limited at around 30 seconds. When this period passes, the operating system considers the driver unresponsive and it reinitializes the graphic driver. On different systems that timeout may be lower, and provisions must be made to restrict the execution time for long running kernels. In Figure 7.6 are shown the results. The measurements were made in steps of 1 MPixel.

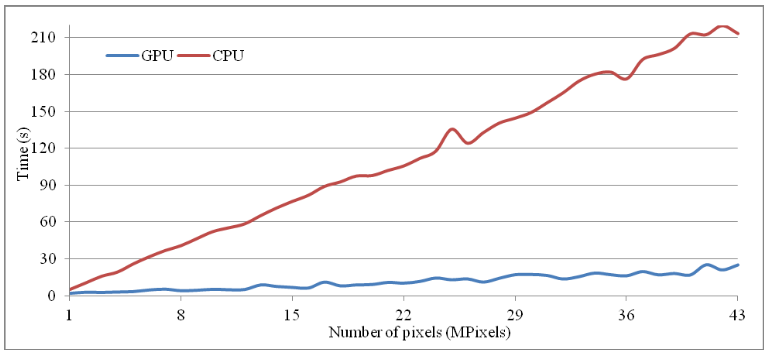

Fig 7.6 – CPU vs GPU execution across all the test domain

If the work quantity needed for a task (in this case the width of the image), and also the tasks number (the height of the image) grow, the CPU is outrun by GPU with a linear progression. Because we incremented linearly the pixels number, the work quantity also has a linear increase. In this test the numeric difference between the streaming cores of GPU (2816) and the cores of the CPU (4) becomes apparent, because on each case the incremented workload is sent to all the processing elements available, so each CPU core receives a greater quantity of work. Even if both times changes in mostly a linear way, the increase of the CPU time is stepper than the time increase on GPU. The GPU maximum speedup compared with CPU was 12.65x.

We also tested how the GPU computes different types of workloads. In this test we kept constant the task size (the image width) and we varied the tasks number (the image height). We executed this test on GPU only. We measured the times of 3 different task sizes. In Figure 7.7 are are shown the results. Data was collected in steps of 64 tasks each. We explain that graph by the effects of two aspects. The general shape of every line is given by the way in which the workload occupies the GPU cache memory. In the case of smaller workloads (of approximately 1000 width), the growth is almost linear across the test domain. If the workload is increased and more memory is required, the cache misses grows. This increase is more apparent at the graph in the right side, because it influences strongly the increase in time. The next aspect that influences the graph is the tasks allocation scheme on GPU. When tasks are added and their total number is lower than the streaming cores, the increase of time for each task added is very small, due to the fact that a single batch of works is used by the GPU to run all these tasks. If the tasks number is greater than the streaming cores number, we can be in two extreme case: a batch of jobs end about in the same moment and in that case we are in a local minimum; if we increase with a small number of tasks, the new tasks need another batch for them only and we will be in a local maximum. More complexity is added because the execution divergence and also because not all tasks require the same quantity of work.

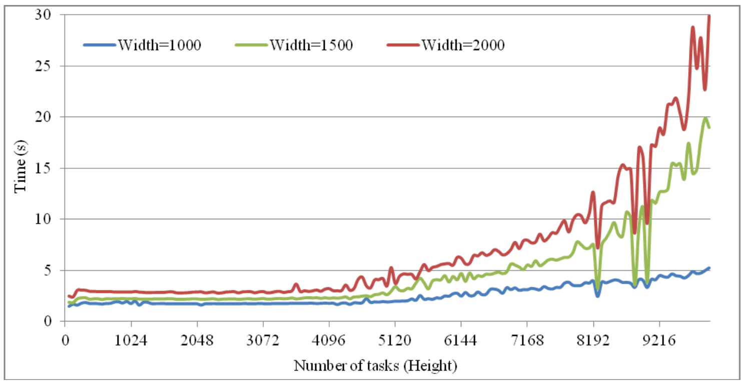

Fig 7.7 – Different number of tasks run on GPU

A result of that analysis is that the longer tasks are better to be reorganized so the amount of a task required memory is smaller (or they have common data). The execution time in this way is optimized by lowering the cache misses; this avoids also the timeout of the operating system when the code is executed on GPU. The tasks number for one GPU execution may be modified using the scheduler settings.

When we compared Aparapi with our library, we needed to write the test code using a representation that can be run with Aparapi. It was required to replace some high level representations of data structures with lower level ones, like:

- There is no support for reference types in Aparapi (aside of primitive vectors), so each used classes (like Point, Line, …) are replaced using float vectors. A Point for example is represented like a 3 floats vector and a Line like a 6 floats vector.
- Aparapi uses the same global context for each thread. The differentiation between each thread data is commonly implemented using OpenCL functions like getGlobalId(). It was needed to employ combined vectors for every thread (task) data. The access of the thread specific data was made with the thread global id.

- There is no dynamic memory management (not even allocations) in Aparapi so it was needed to employ global data structures to hold the values returned by functions in the case of non primitive types. To differentiate between the threads data it was used the thread global id.

To illustrate some of the required changes in Figure 7.8 we show how parts of the application are implemented with our library; in Figure 7.9 we show how these are implemented with the Aparapi library.

```
public class Point{
        public float xp,yp,zp;
        public Point(float xp,float yp,float zp)
                {this.xp=xp;this.yp=yp;this.zp=zp;}
        public Point(){}
        public void set(float xp,float yp,float zp)
                {this.xp=xp;this.yp=yp;this.zp=zp;}
        public void set(Point pt){xp=pt.xp;yp=pt.yp;zp=pt.zp;}
        public float len(){return Math3D.length(xp,yp,zp);}
        public void vectorFromPoints(Point orig,Point dst)      {
                xp=dst.xp-orig.xp;
                yp=dst.yp-orig.yp;
                zp=dst.zp-orig.zp;
        }
…
}
…
float   propagateRay(Line ray){…}
Point centerToInters=new Point();
Point intersToRayOrigin=new Point();
```

Fig 7.8 – An implementation with our library for a part of the application

Because the Aparapi implementation cannot use Java basic idioms such as classes, the algorithms written using this library require additional proxy code to be integrated with the application: the classes are translated to primitive vectors and back, individual tasks data is merged into the same vector, etc.

```
final float pointLen(float []pt){
        int offset = 3*getGlobalId();
        return length(pt[offset],pt[offset+1],pt[offset+2]);
}
final void vectorFromPoints(float []dest,float []orig,float []destination){
        int offset = 3*getGlobalId();
        dest[offset]=destination[offset]-orig[offset];
        dest[offset+1]=destination[offset+1]-orig[offset+1];
        dest[offset+2]=destination[offset+2]-orig[offset+2];
}
…
final float []intersToRayOrigin;
final float      propagateRay(float []r){…}
final float []centerToInters;
```

Fig 7.9 – An implementation with Aparapy for a part of the application

To compare the execution of our library with Aparapi, we varied the pixels number and the amount of needed work for each pixel. The work per pixel measurement was required to evaluate different workloads keeping constant the used memory amount and the threads number. We recomputed every pixel more times (n), starting the normal case (n=1). This process does not requires memory allocations. In Figure 7.10 we showed the results using Aparapi and in Figure 7.11 the results using our library. For all figures the data was collected in steps of 1 MPixel.

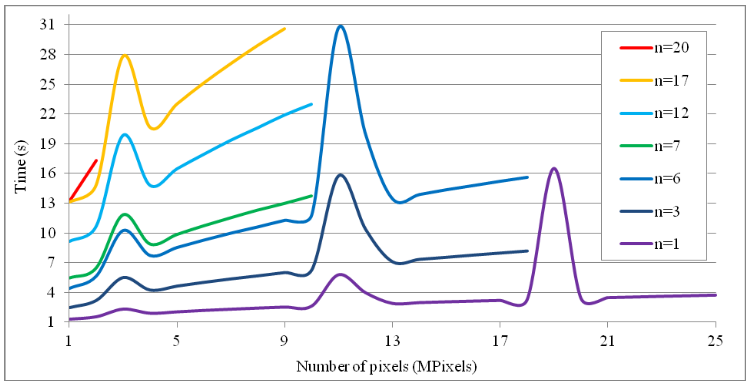

Fig 7.10 – Aparapi execution for different numbers of tasks and recomputations

For the Aparapi version, the operating system began at around 25 MPixels to reset the graphic card driver. Our library was capable to produce results until 46 MPixels. Both libraries have approximately linear progressions, with a number of prominent peaks. There is a better time for Aparapi and a smaller angle of growing. Our library has more irregularities from the linear progression and, as previously discussed, we consider that a combined effect of the GPU cache and data layout. Aparapi has smaller irregularities due to the fact that in its case data is already ordered by tasks and vectorized, which enhances the data locality. The maximum running time for both implementations, after which the OS starts to reset the graphic card driver was of around 31 seconds. If the peak time exceed 31 seconds, then the application crashes.

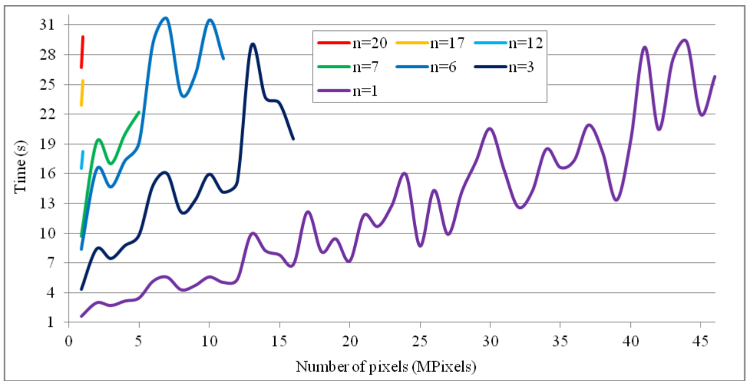

Fig 7.11 – Our library execution on different number of recomputations and tasks

7.4 Conclusions

We proposed in this section an algorithm and library that enable Java execution on GPU. We used reflection to access the code of the application and OpenCL code generation in order to create the kernels which run on GPU. Our approach is suitable also in the case of code processed at runtime like plugins. The library handles automatically common distributed computing tasks like serialization and deserialization, synchronization and communication with the GPU. The system of data serialization is capable to handle complex data structures and this makes possible to make available for execution GPU classes, static methods, fields of reference types and all types of arrays.

The library provides a thin compatibility layer for Java on top of the OpenCL. That layer makes possible dynamic memory management and exceptions handling. We strive in the future to extend the compatibility layer with recursive calls and also with calls of virtual methods (dynamic dispatch for non-final methods). When possible, the functions native to OpenCL are employed over the standard Java libraries. A library for OpenCL primitives that are not in the Java standard libraries is also provided. This library is portable, and it can be used on CPU and on GPU.

Our algorithm uses a model based on MapReduce to manage the simultaneous tasks. The return values of tasks are sent directly to a handler. In certain cases this allows the processing of results on arrival, without the need to store them. The creation and management of tasks is abstracted in regard with the computing resources, and the code can be run both on CPU and on GPU without any modifications. That simplifies the maintenance of the code, allows an easy debugging and the CPU can act as a fallback resource if no suitable GPU is present.

We tested our library with a test application in standard Java code, which does not have any OpenCL specific constructs. With our test configuration we had significant speedups up to 12.65x for GPU over the execution on CPU. We consider an important conclusion of our research that parts of Java standard applications which use classes, exceptions handling and dynamic memory allocation (but for now without recursion and virtual calls) can be translated automatically into OpenCL and be run on GPU, which can bring certain advantages. The proposed library and algorithm provides the capability to implement Java code that can be integrated easily with complex data structures and this code does not need specific platform calls. This simplifies greatly the goal to run more complex applications on computing resources such as GPU or CPU. This is an important step forward over existing libraries like Aparapi, which were originally designed as thin layers on top of the OpenCL and the developer must use specific OpenCL idioms. Aparapi in our tests obtained a better time compared with our library but it was able to process only a limited data domain. It also needed coding the test application using a way that is not Java specific (without classes). The Aparapi implementation also required proxy code in order to serialize and deserialize the data of the application to and from an OpenCL suitable representation. On future research we will investigate ways to increase the range of applications that can run on GPU, research better optimizations and also obtain an enhanced reliability for GPU execution.

8. Conclusions

The distributed computing field is a very dynamic one and there are many research and development directions of great importance. As many computing tasks become more resource intensive, both from the point of view of the computation power required and regarding the vast amount of data involved in these computations, more and more applications try to offer distributed computing implementations, in order to be able to solve these problems. With the current tools and frameworks the distributed applications require a considerable coding effort and to ease and optimize this effort was one of our main research motivations in this thesis. Our research was concentrated on three main directions. First we elaborated a distributed computing model suitable for using heterogeneous resources such as CPUs, GPUs and computer networks in a uniform manner and which can also automate many distributed computing tasks. Secondly, we implemented our model as a Java library, so it can be used in regular Java applications. Third, in order to include the GPUs as a computing resource, we developed two algorithms and their implementations. First algorithm is suitable for the cases when the computation is too complex to be run solely on GPU. The second algorithm can be used in the cases when the GPU is capable to handle the entire computation.

There are many concepts specific to distributed computing, such as remote function invocation or code deployment. These concepts need to be better integrated into the existent frameworks, development tools, programming languages and high level models. Many of the tools and frameworks use powerful computation models (such as the MapReduce model) taken for example from the functional languages theory, but these models were initially developed for sequential applications and they are not sufficiently enough adapted to a highly multithreaded and distributed environment. We shown that research and development work needs to be done in order to adapt the sequential models to the challenges imposed by distributed computing, such as scheduled asynchronous invocations or synchronization.

A comprehensive high level distributed computing framework must make available to the application all the existent computing resources, like local CPU cores, remote computers and GPUs. Each one of these has very different interfaces, for example multithreading libraries for CPU cores, networking and remote invocation libraries for remote computers and OpenCL libraries for GPUs. All these interfaces must be made available to the application in a uniform and abstract manner, in order to isolate the low level details from the application logic. If this aspect is accomplished, the framework will be able to run a given computation on the most suitable resource, without any additional code required from the developer.

In this thesis we propose an original model for distributed computing that builds on the well known MapReduce model and extends it with new concepts necessary for a multithreaded and distributed environment. Our model uses only three concepts and it is simple to learn, especially for programmers with an OOP background. The model abstracts in a uniform manner different computing resources such as CPUs, GPUs and computer networks, so the application can use heterogeneous architectures without the need to write specific code for each computing resource type. In our model many distributed computing tasks are handled automatically by default, so aspects like resources discovery, code deployment, data serialization, tasks distribution, remote invocations, recovery from errors and synchronization are automatically handled with good results in most of the cases. The model also includes specific optimizations, for example it treats immutable data as a special case and this data is sent only once to the computing resources. Especially for network computers or slow buses this optimization can bring considerable improvements on computing time. We implemented our model as a virtual machine and the test results shown that it is scalable and in the same time it is capable to distribute evenly the workload on the available computing resources.

We implemented the model as a Java framework so it can be used in regular Java applications, without specific language level support. We are able to implement most of its semantics and constraints only by means of data structures and function calls, without resorting to other compilation steps, such as a preprocessor phase. We tested our framework and it was capable to abstract both the remote computers and the CPU cores as computing resources, as well as low-level distributed computing tasks such as resource discovery, code deployment and remote invocations. The test results for our framework shown that it is scalable both in terms of network computers and CPU cores and the scheduler algorithm succeeded to distribute the workload evenly on all these resources.

In this thesis a considerable research is dedicated to use the GPUs as a computing resource. With the development of the GPUs with general-purpose computing capabilities, this new and powerful computing resource can be employed to solve different tasks, especially the numerical intensive ones. Due to the fact that the GPUs have several shortcomings such as their incapacity to call operating system functions or their stackless execution model, it is not an easy task to adapt complex code flows for execution on GPU.

In order to distribute computing tasks on GPU, we propose two approaches on which we made advancements with our research. If the tasks are complex and not suitable for execution only on GPU, for example they contain I/O calls, we developed a cooperative CPU-GPU execution algorithm. It collects the computation parts intended for GPU execution across multiple CPU threads and runs them on GPU in a single batch. For optimal performance this algorithm uses a combination of threads and fibers, which reduces the switching time between threads and also enables the creation of thousands of tasks, implemented as fibers, which allows us to fully use all the GPU cores. The algorithm was implemented as a C++ library and it is easily applicable to the existent threaded applications. The practical results obtained with our library show a significant speedup over the CPU only execution.

If the tasks intended to be distributed on GPU are suitable for this kind of execution, we developed a new algorithm and library, capable to translate from Java bytecode to OpenCL code and to run the resulted code on GPU. The library also automatically handles tasks such as data serialization/deserialization, and GPU communication and synchronization. The algorithm can translate code which uses classes, reference types, dynamic memory allocation and exception handling (but for now without virtual calls and recursion), which is a significant improvement over the existing approaches. We integrated the library with our model and we provided an abstraction layer over OpenCL, which allows the use of either the CPU or GPU in an abstract manner, without any change in the source code. The practical results show improvements of over ten times speedup on GPU execution.

We intend to further research the thesis subjects in directions such as further refinements of our model to allow more use cases, better integration of all the three computing resources discussed (CPU, network computers, GPU), load balancing, and increased reliability and recovery from errors. We also intend to add more features to the OpenCL generation algorithm, so we can increase the range of applications which can be run on GPU.

Published papers

Razvan-Mihai Aciu, Horia Ciocarlie, ”Runtime Translation of the Java Bytecode to OpenCL and GPU Execution of the Resulted Code”, Journal of Applied Sciences “Acta Polytechnica Hungarica”, ISSN1785-8860, Volume 13, Issue 3, 2016. **ISI indexed**

Razvan-Mihai Aciu, Horia Ciocarlie, ”Framework for the Distributed Computing of the Application Components”, Advances in Intelligent and Soft Computing, Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, June 30-July 4, 2014, Brunów, Poland, Springer, ISBN: 978-3-319-07012-4. **ISI indexed**

Razvan-Mihai Aciu, Horia Ciocarlie, ”Application Level Execution Model for Transparent Distributed Computing”, New Results in Dependability and Computer Systems, Proceedings of the 8th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, September 9-13, 2013, Brunów, Poland, Springer, ISBN: 978-3-319-00944-5. **ISI indexed**

Codruta-Mihaela Istin, Horia Ciocarlie, **Razvan-Mihai Aciu**, ”Optimization Algorithm for the Preservation of Sensor Coverage”, New Results in Dependability and Computer Systems, Proceedings of the 8th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, September 9-13, 2013, Brunów, Poland, Springer, ISBN: 978-3-319-00944-5. **ISI indexed**

Razvan-Mihai Aciu, Horia Ciocarlie, “Algorithm for Cooperative CPU-GPU Computing”, 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Romania, September 23-26, 2013, DOI 10.1109/SYNASC.2013.53. **ISI indexed**

Razvan-Mihai Aciu, Horia Ciocarlie, “G-Code Optimization Algorithm and its application on Printed Circuit Board Drilling”, 9th International Symposium on Applied Computational Intelligence and Informatics (SACI), Romania, May 15-17, 2014, DOI 10.1109/SACI.2014.6840096. **ISI indexed**

Razvan-Mihai Aciu, Horia Ciocarlie, “Cooperative Task Scheduler For Entry Level Microcontrollers”, 6th International Workshop On Soft Computing Applications (SOFA), Romania, July 24-26, 2014, Springer, ISBN 978-3-319-18416-6

Codruta-Mihaela Istin, **Razvan-Mihai Aciu**, Horia Ciocarlie, “Traffic Behaviour Simulator SIMULO with Sensor Coverage Computation”, 6th International Workshop

On Soft Computing Applications (SOFA), Romania, July 24-26, 2014, Springer, ISBN 978-3-319-18416-6

Acknowledgement

This work was partially supported by the strategic grant POSDRU/159/1.5/S/137070 (2014) of the Ministry of National Education, Romania, co-financed by the European Social Fund – Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

Bibliography

[1] A. S. Tanenbaum, M. Van Steen, “Distributed Systems – Principles And Paradigms”, 2002

[2] L. Mearian, “DreamWorks tops compute-cycle record with 'The Croods'“, High Performance Computing, March 2013

[3] C. Mellor, “DreamWorks signs cloud computing deal, The register”, July, 2010

[4] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, V. S. Pande, "Folding@home: Lessons From Eight Years of Volunteer Distributed Computing", IEEE International Symposium on Parallel & Distributed Processing, 2009

[5] Pande Lab, Stanford University, Folding@home project, 2013, http://folding.stanford.edu/

[6] The NCAR-Wyoming Supercomputing Center, 2013, http://nwsc.ucar.edu/

[7] Berkeley Open Infrastructure for Network Computing (BOINC) Project, Berkeley University, http://boinc.berkeley.edu/

[8] J. Dongarra, T. Sterling, H. Simon, E. Strohmaier, "High performance computing: Clusters, constellations, MPPs, and future directions", “Lawrence Berkeley National Laboratory”, 2003

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing", Technical Report No. UCB/EECS-2009-28, 2009

[10] A.Iosup, D. Epema, “Grid Computing Workloads: Bags of Tasks, Workflows, Pilots, and Others”, Internet Computing, IEEE Volume:15 , Issue: 2, pp 19,26, 2010

[11] J. Peng, X. Zhang, Z. Lei, B. Zhang, W. Zhang, Q. Li, “Comparison of Several Cloud Computing Platforms”, Second International Symposium on Information Science and Engineering (ISISE), Shanghai, 2009

[12] S. Zhang, H. Polytech, X. Chen, S. Zhang, X. Huo, “The comparison between cloud computing and grid computing”, International Conference on Computer Application and System Modeling (ICCASM), Taiyuan 2010

[13] F. Xhafa, A. Abraham, "Computational models and heuristic methods for Grid scheduling problems", Future Generation Computer Systems, Volume 26, Issue 4, 2010

[14] B. Neelima, NITK-Surathkal, P.S. Raghavendra, "Recent trends in software and hardware for GPGPU computing: A comprehensive survey", International Conference on Industrial and Information Systems, Mangalore, 2010

[15] Introducing TITAN, Advancing the Era of Accelerated Computing, 2013, http://www.olcf.ornl.gov/titan/

[16] S. Tucker Taft, J. Bloch, R. Bocchino, S. Burckhardt, H. Chafi, R. Cox, B. Gaster, G. Steele, D. Ungar. "Multicore, manycore, and cloud computing: is a new programming language paradigm required?" In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, SPLASH 2011

[17] G. R. Andrews, "Foundations of multithreaded, parallel, and distributed programming", Pearson, 1999

[18] H. Kopetz, "Real-Time Systems: Design Principles for Distributed Embedded Applications", Springer 2011

[19] C. Weinhardt, W. A. Anandasivam, B. Blau, N. Borissov, T. Meinl, W. W. Michalk, J. Stößer, "Cloud Computing – A Classification, Business Models, and Research Directions, Business & Information Systems Engineering", 2009, Volume 1, Issue 5, pp 391-399

[20] J. Dean, S. Ghemawat, "MapReduce: simplified data processing on large clusters", Commun. ACM 51, 2008

[21] P. Haller, F. Sommers, "Actors in Scala", artima, 2011

[22] GridGain, "GridGain In-Memory Computing Platform 6.0", http://atlassian.gridgain.com/wiki/display/GG60/Home, access time: 07.06.2015

[23] L. Dematté, D. Prandi, "GPU computing for systems biology", Briefings in Bioinformatics, Vol. 11, No. 3, pp. 323-333, 2010

[24] AMD: A New Era in PC Gaming, E3, Los Angeles, California, U.S., 2015

[25] Jen-H. Huang, "Opening Keynote, GPU Technology Conference", San Jose, California, U.S., 2015

[26] P. Gepner, D. L. Fraser, M. F. Kowalik, K. Waćkowski, "Evaluating New Architectural Features of the Intel(R) Xeon(R) 7500 Processor for HPC Workloads, Computer Science", Vol 12, 2011

[27] J.M. Vahid, D. Venkata, "Large-Scale Transient Stability Simulation on Graphics Processing Units", IEEE Power & Energy Society General Meeting, 2009

[28] M. Roman, M. Avi, K. Avinoam, B. Evgeny, "Exploring the Limits of GPGPU Scheduling In Control Flow Bound Applications", ACM Transactions on Architecture and Code Optimization - TACO , pp. 1-22, 2012

[29] N.G. Peter, "NVIDIA's Fermi: The First Complete GPU Computing Architecture", September 2009

[30] X. Ping, Y. Yi, M. Mike, R. Norm and Z. Huiyang, "Many-Thread Aware Instruction-Level Parallelism: Architecting Shader Cores for GPU Computing", PACT'12, September 19–23, 2012

[31] X. Kai, Z.C. Danny, X.H. Sharon, Z. Bo, "Shell: A Spatial Decomposition Data Structure for 3D Curve Traversal on Many-core Architectures", 21st European Symposium on Algorithms ESA 2013, September 02-04, 2013

[32] G. Pengcheng, T. Yubo, L. Hai, "Parallel Shooting And Bouncing RayMethod On Gpu Clusters For Analysis Of Electro-Magnetic Scattering", Progress In Electromagnetics Research, Vol. 137, 87-99, 2013

[33] C. Dabrowski, "Reliability in grid computing systems", Concurrency and Computation: Practice and Experience, Volume 21, Issue 8, pages 927–959, 2009

[34] B.S. Murugan, D. Lopez, "A Survey of Resource Discovery Approaches in Distributed Computing Environment", International Journal of Computer Applications Volume 22, No. 9, 2011

[35] S. K. Kwan, J.K. Muppala, "Resource Discovery and Scheduling in Unstructured Peer-to-Peer Desktop Grids", Conference on Parallel Processing Workshops (ICPPW), 2010 39th International, Canada 2010

[36] M. Nazir, "Cost-effective resource management for distributed computing", Doctoral thesis, UCL London 2011

[37] D. Cokuslu, A. Hameurlain, K. Erciyes, "Grid Resource Discovery Based on Centralized and Hierarchical Architectures" International Journal for Infonomics (IJI), Volume 3, Issue 1, 2010

[38] M. M. Motalebi, R. Maghami, A. S. Ismail, A. A. Ahmed, "Reliable Resource Discovery Approaches for Grid Environments", International Journal of Computer Communications and Networks, 2011

[39] M. I. Andreica, "Techniques for the Optimization of Communication Flows in Distributed Systems", 2010

[40] K. Jander, W. Lamersdorf, "Compact and Efficient Agent Messaging", Eleventh International Conference on Autonomous Agents and Multiagent Systems, Valencia 2012

[41] D.E. Comer, "Computer Networks and Internets", Addison-Wesley, 2008

[42] A. Sumaray, S. K. Makki, "A comparison of data serialization formats for optimal efficiency on a mobile platform", Conference on Ubiquitous Information Management and Communication, NY 2012

[43] K. Popović, Z. Hocenski, "Cloud computing security issues and challenges", Proceedings of the 33rd International Convention, Croatia 2010

[44] S. Subashini, V. Kavitha, "A survey on security issues in service delivery models of cloud computing", Journal of Network and Computer Applications, 2010

[45] D. Zissis, D. Lekkas, "Addressing cloud computing security issues", Future Generation Computer Systems, Volume 28, Issue 3, Pages 583–592, 2012

[46] M. Zhou, R. Zhang, W. Xie, W. Qian, A. Zhou, "Security and Privacy in Cloud Computing: A Survey", Sixth International Conference on Semantics Knowledge and Grid (SKG), Beijing, 2010

[47] P. Mell, T. Grance, "Effectively and Securely Using the Cloud Computing Paradigm", NIST Cloud Research Team, 2009

[48] N. Hammoud, "Decentralized log event correlation architecture", Proceedings of the International Conference on Management of Emergent Digital EcoSystems, SUA 2009

[49] V. Saraswat, G. Almasi, G. Bikshandi, Calin Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, O. Tardieu, "The Asynchronous Partitioned Global Address Space Model", First Workshop on Advances in Message Passing, Canada 2010

[50] N. Rodriguez, S. Rossetto, "Integrating Remote Invocations With Asynchronism And Cooperative Multitasking", Parallel Process. Lett. 18, 2008

[51] P. Krzyzanowski, "Remote Procedure Calls", Rutgers. University 2012

[52] G. Juve, E. Deelman, "Automating Application Deployment in Infrastructure Clouds", IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom), Greece, 2011

[53] M. J. Park, D. K. Kim, W. T. Kim, S. M. Park, "Dynamic Software Updates in Cyber-Physical Systems", Conference on Information and Communication Technology Convergence (ICTC), 2010 International, Jeju 2010

[54] J. Ravi, Z. Yu, W. Shi, "A survey on dynamic Web content generation and delivery techniques", Journal of Network and Computer Applications, Vol 32, 2009

[55] V. Narvaez, "Distributed Objects and Remote Invocation", Addison Wesley 2010

[56] B. Silvestre, S. Rossetto, N. Rodriguez, J.-P. Briot, "Flexibility and coordination in event-based, loosely coupled, distributed systems", Computer Languages, Systems & Structures, vol 36, 2010

[57] E. Tilevich, S. Gopal, "Expressive and Extensible Parameter Passing for Distributed Object Systems", ACM Trans. Softw. Eng. Methodol 2011

[58] M. D. Preda, M. Gabbrielli, I. Lanese, J. Mauro, G. Zavattaro, "Graceful Interruption of Request-Response Service Interactions", Service-Oriented Computing Lecture Notes in Computer Science Volume 7084, pp 590-600, 2011

[59] E. Sindrilaru, A. Costan, V. Cristea, "Fault Tolerance and Recovery in Grid Workflow Management Systems", Conference on Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International, Krakow 2010

[60] J. Celaya, U. Arronategui, "Distributed Scheduler of Workflows with Deadlines in a P2P Desktop Grid", Conference on Parallel, Distributed and Network-Based Processing (PDP), Pisa 2010

[61] M. Malawski, T. Bartyński, M. Bubak, "Invocation of operations from script-based Grid applications", Future Generation Computer Systems, Volume 26, Issue 1, Pages 138–146, 2010

[62] L. Watkins, W.H. Robinson, R.A. Beyah, "A Passive Solution to the CPU Resource Discovery Problem in Cluster Grid Networks", Parallel and Distributed Systems, IEEE Transactions on (Volume:22, Issue: 12), 2011

[63] V.Vinothina, R. Sridaran, P. Ganapathi, "A Survey on Resource Allocation Strategies in Cloud Computing", International Journal of Advanced Computer Science and Applications Vol. 3, No.6, 2012

[64] H. Izakian, A. Abraham, V. Snášel, "Comparison of Heuristics for Scheduling Independent Tasks on Heterogeneous Distributed Environments", International Joint Conference on Computational Sciences and Optimization, CSO 2009.

[65] Y.-F. Yu, P.-J. Huang, K.-C. Lai, C.-T. Yang, K.-C. Li, "On the Design of a Performance-Aware Load Balancing Mechanism for P2P Grid Systems", Advances in Grid and Pervasive Computing Lecture Notes in Computer Science Volume 5529, pp 269-280, 2009

[66] M. Camelo, Y. Donoso, H. Castro, “MAGS – An Approach Using Multi - Objective Evolutionary Algorithms for Grid Task Scheduling”, International Journal Of Applied Mathematics And Informatics, vol. 5, 2011

[67] H.-H. You, C.-C. Yang, J.-L. Huang, “A load-aware scheduler for MapReduce framework in heterogeneous cloud environments”, In Proceedings ACM Symposium on Applied Computing, 2011

[68] A. Lazouski, F. Martinelli, P. Mori, “Survey: Usage control in computer security: A survey” Comput. Sci. Rev. 4, 2010

[69] R. H. Halstead Jr, "Multilisp: A Language for Concurrent Symbolic Computation", ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, October 1985

[70] E. B. Johnsen, O. Owe, "An Asynchronous Communication Model for Distributed Concurrent Objects", Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM 2004), IEEE press, Sept. 2004

[71] M. A. N. Al-hayanni, A. Rafiev, R. Shafik, F. Xia, A. Yakovlev, "Extended Power and Energy Normalized Performance Models for Many-Core Systems", Technical Report Series, NCL-EEE-MICRO-TR-2016-198, January 2016

[72] A. Timo, K. Tero, “Architecture considerations for tracing incoherent rays”, Advances in Computer Graphics Hardware, pp. 113-122, 2010

[73] T. Usui, R. Behrends, J. Evans, Y. Smaragdakis, "Adaptive locks: Combining transactions and locks for efficient concurrency", Journal of Parallel and Distributed Computing, February 2010

[74] N. Nimalan, “Use of Multi-GPU Systems for Larger Than Device FFTs: With Applications in Ultrasound Simulations”, Thesis submitted for the degree of Master of Philosophy of the Australian National University, February 2013

[75] K. P. Raphael, “Multi-core architectures and their software landscape”, Computing Science Handbook, Vol. 1, Chap. 35, 2013

[76] C. S. Angela, M. Jacob, D. Arash, M. Kevin, E. Bryan, “Parallelism via Multithreaded and Multicore CPUs”, IEEE Computer - COMPUTER , vol. 43, no. 3, pp. 24-32, 2010

[77] E.S. John, G. David, S. Guochun, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”, Computing in Science and Engineering - C in S&E, vol. 12, no. 3, pp. 66-73, 2010

[78] Khronos Group, “The open standard for parallel programming of heterogeneous systems”, https://www.khronos.org/opencl/, access time: 14.07.2015

[79] AMD, "Bolt”, https://github.com/HSA-Libraries/Bolt, access time: 16.07.2015

[80] K. Spafford, “ArrayFire: A Productive GPU Software Library for Defense and Intelligence Applications”, GPU Technology Conference, San Jose, California, U.S., 2013

[81] AMD, “Aparapi”, https://github.com/aparapi/aparapi, access time: 16.07.2015

[82] E. Caspole, “OpenJDK Sumatra Project: Bringing the GPU to Java”, AMD Developer Summit (APU13), 2013

[83] R. M. Aciu, H. Ciocarlie, “Framework for the Distributed Computing of the Application Components”, Proceedings of the Ninth International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, 2014, Brunów, Poland, Springer, ISBN: 978-3-319-07012-4

[84] M. Cantelon, M. Harter, T.J. Holowaychuk, N. Rajlich, “Node.js in Action", Manning, 2014, ISBN 9781617290572

[85] M. Steinberger, M. Kenzel, B. Kainz, D. Schmalstieg, “ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU, Innovative Parallel Computing” (InPar), San Jose, California, U.S., 2012

[86] C. Hong, D. Chen, W. Chen, W. Zheng, H. Lin, “MapCG: Writing Parallel Program Portable between CPU and GPU”, Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, Vienna, Austria, 2010

[87] Y. Ke, H. Bingsheng, L. Qiong, V.S. Pedro, S. Jiaoying, “Stack-based parallel recursion on graphics processors”, 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2009

[88] E. Kuleshov, “Using the ASM framework to implement common java bytecode transformation patterns”, Sixth International Conference on Aspect-Oriented Software Development, Vancouver, British Columbia, Canada, 2007

[89] Java bindings for OpenCL, http://www.jocl.org/, access time: 19.07.2015