IOSUD - Universitatea Politehnica Timişoara
Şcoala Doctorală de Studii Inginereşti

Integrated frameworks for distributed computing

Teză de doctorat – Rezumat

pentru obținerea titlului științific de doctor la
Universitatea Politehnica Timișoara
în domeniul de doctorat _CALCULATOARE ŞI TEHNOLOGIA INFORMATIEI_

autor ing. _Răzvan-Mihai ACIU_
conducător științific Prof.univ.dr.ing. _ Horia CIOCÂRLIE _

luna_09_ anul_2017_

1. **Introduction**

In this chapter we start by describing different application fields especially suitable for distributed computing. Examples are given for each field and it can be seen that the required processing power is far superior to what a common computer can offer, so a distributed system [1] must be used. For example, in the field of 3D rendering, the animated movie "The Croods" required around 80 million compute hours and 250TB data storage capacity. For this project, the producer DreamWorks employed a specialized computing resources provider, Cerelink, which provided a cloud with a theoretical peak speed in 2010 of 172 teraflops, with 133 teraflops sustained operation. This cloud used 1,792 computing nodes with a total of 14,336 cores of quad Xeon 3.0 GHz processors housed in 28 racks.

For projects of public interest a new method of obtaining computing power emerged: volunteer networks. In this case, people interested in a project offer their computing resources for project related computations, creating a computing grid. One of the best known projects of this type [2] is folding@home, which in September 2013 used a volunteer network of over 266,000 computers.

After these examples distributed infrastructure classification is made from the point of view of the access speed to its components and its management facilities. There are 3 cases [3]: **massively parallel processing systems** (strong physical interconnections, in many cases effectively making this system a supercomputer), **clusters, clouds, computers/servers farms** (independent computers, usually in the same place, sharing high speed connection network, that can be remotely managed also possibly with virtualization software), **grids** (independent computers in different geographical area, with very different hardware and operating systems, usually without virtualization or remote management). The most complex applications are the ones written for grids, because of their heterogeneous nature and their unpredictable online time.

Another classification of the computing resources takes into account their computational possibilities. We can have **systems with independent microprocessors** (such as the CPU cores, capable to process any algorithm and to access any part of the system) and there can be **systems with dependent computing units, tailored for specific tasks** (such as the GPUs, which can be very powerful on specialized applications but have more restrictions than CPUs).

We also considered the possible cases of implementing an application: **sequential** (all the tasks are computed one after another), **multithreaded** (some tasks are performed simultaneously, using the local CPUs), **distributed** (some tasks are performed simultaneously, both on the local machine and on remote computers) [4]. For each case was discussed its requirements from an implementation perspective. It can be seen that the distributed applications are the most complex, because they need to implements aspects such as resources discovery, serialization/deserialization, code deployment, remote invocation and computing resources management. Considering the complexity of the distributed applications, we will concentrate on proposing, implementing and testing new models, frameworks and tools in order to make the distributed applications easier to be developed. Some criteria such as **reliability**, **abstraction**, **simplicity** will need to be fulfilled.

2. Thesis motivation, objectives and structure

We discuss first the well known **MapReduce** model [5], taken from the functional languages such as Lisp, and we highlight some of its drawbacks when it is used for distributed applications. Many of these drawbacks originate in the fact that the original implementation of this computing model was a sequential one. For example, in general it does not have a language level support but it is implemented using libraries/frameworks, which makes harder for the compiler/runtime to automate or optimize some tasks, such as the scheduler initialization or results synchronization. MapReduce has to be extended with specific distributed computing concepts, such as concurrent data access management and asynchronous calls. It also needs to be enhanced in order to take into account the traffic time between the distributed resources, which in some cases can be very significant. 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.

In this thesis we propose a novel computing model, based on MapReduce, especially created for the distributed computing applications, which aims to fulfill as much as possible these requirements. In chapter 4 we detail our novel model.

Based on the proposed model, we create a framework which provides the means to use it. Our framework needs to be address some requirements, such as: it should automate as much as possible the low level tasks (resource discovery, remote invocation, serialization/deserialization, load balancing and error recovery), all the available computing resources must be abstracted and used in an uniform way, the interface must be simple, when possible it should use industry standard libraries and tools and it should also provide the all required components, such as a remote server which handles the remote calls on the network resources. The proposed framework is described in chapter 5.

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 [6]. GPUs such as AMD Radeon Fury X or NVIDIA GeForce GTX Titan X are capable of 8.6 TFLOPS, respectively 7 TFLOPS FP32. An Intel Xeon X7560 CPU is capable of 72.51 GFLOPS FP64. From this data, if the algorithm is massively parallel and FP32 operations are enough, one GPU may provide a performance comparable with several desktop CPUs.

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 below:

- A GPU cannot access the operating system functions, such as the ones regarding the filesystem, network, etc.
- The GPUs do not have a stack. The OpenCL [7] 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 to recursive algorithms [8].

- Until OpenCL 2.0 the GPUs had a different memory space than the CPU. Different memory spaces makes difficult to share complex data structures which involves pointers. These structures need to be serialized with the pointers converted to other representations such as indexes or identifiers, transferred to GPU and deserialized.
- A GPU core does not have its own instructions fetching and decoding unit but many cores are grouped in workgroups which run using the same instruction pointer, but different data. If a conditional branching makes different cores inside a workgroup to choose different execution paths, some of them will be put in a waiting state until the execution will resume at a future common instruction, sometimes only after other cores finished their jobs. This phenomenon is named branch divergence.

Due to the above limitations, it is not easy to integrate the GPU for general purpose programming tasks, especially when complex algorithms are needed. A main research direction in this thesis is to integrate the GPUs on complex computations, if possible abstracting them in the same manner as for the CPUs, so the programmer will not need to handle them separately. In this respect, in chapters 6 and 7 we propose two novel algorithms for using the GPUs in different cases.

3. Distributed computing concepts and requirements

In this section we discuss some distributed computing aspects, relevant to our research. As these aspects are common to many distributed computing applications, a direction of our research was to identify among them common patterns which in most of the cases can be automated. In this respect, our proposed model and algorithms try to automate as much as possible the tasks involved by these concepts and requirements.

The **network management** handles aspects such as resource discovery [9], communication and security [10]. For resource discovery we discuss the use of broadcast messages and the use of resources index servers. Especially for grid networks, the resources discovery must be a continuous process, because in this case the remote computers can at any time join or leave the network. For communication a wide range of methods and formats can be used, starting with binary encodings over low-level, custom TCP/IP protocols to ensure a higher throughput and ending with generic text formats such as XML or JSON over standardized protocols like HTTP. Security must cover both the communication and the unauthorized resources access.

The **code deployment** can be done in several ways, starting with manually installing the application on all the available computing resources. As in our research we try to automate as many tasks as possible we aim to develop an automatic code deployment method. In order to do this we use a **remote server** [11] which receives the application code sent by different clients and install it on its computer. The remote server must also handle aspects such as uninstalling the code, caching and versioning, in order to reuses an already installed code, if the application resends it. In the case of faulty application code, the remote server must be able to end a possibly infinite computation. For volunteer networks, the remote server must also ensure a resource usage policy, in order not to disturb the regular usage of that computer.

The code installed on different resources is used through **remote invocations** [12]. These are sent by the application and each invocation requires sending its parameters to the computing resource, computing, receiving the results and possibly recovery from possible external errors such as the ones originating from the network usage. As an important optimization, we differentiate between **global data**, common to all invocations, so it needs to be sent only once, and **invocation specific data**, sent separately for each invocation.

The external errors can originate from the network (for example an infrastructure problem) or if a remote computer leave the network. In such cases it is important first to detect such errors, for example by using timeouts or pings, and secondly to recover from them, by resending the failed invocation to another computing resource.

On the application side, the **invocations scheduler** [13] is responsible with resources discovery, the collection of all the required invocations into a waiting list, asynchronously [14] sending them to the available resources, retrieving the results and also recovers from external errors. Each invocation result is then processed and if needed, assembled into the final result.

4. Application level execution model

In this chapter we present our novel distributed computing model. As it is based on the MapReduce model, first we analyze this model from a distributed computing perspective. As MapReduce was first used in sequential computations, some assumptions were (implicitly) made. We list below these assumptions and how they change in a distributed computing case:

1. All the invocations are synchronous so the results are available when the function returns - in distributed computing the scheduler asynchronously add the invocation to a waiting list and returns. The invocation's results may be available only much later after the scheduler call returned [15].
2. All data resides in the same memory (taking into account the cache memory and disk swapping) so it can be accessed in roughly the same time - a distributed computing application will need to send/receive data over network, a much slower process than a memory access and serialization may also be needed.
3. Besides hardware failure, there are no other sources of external errors - in a distributed computation, even for legitimate invocations and good code network errors or remote resources status changes are likely to appear and all these can result in a failed computation.
4. There is no concurrent access to resources (data racing) - on a distributed computing environment there is no deterministic order in which the results are made available, so it must exists a method to order/identify the result of each invocation, so it can be properly handled.

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.

To introduce our model, we implemented the computation of a specific interval of the Mandelbrot set, which is a well known algorithm, very suitable for parallel processing. For this example we devised a C++ style language with some language constructs which implements our model. We discuss the example and after that we give a functional description of the model's concepts.

The first concept of our model is named **unit**. It is somewhat similar to a **class** construct from an OOP language, but with some important differences:

- It cannot have static attributes and it cannot access directly or indirectly static attributes of any class or global data - this effectively isolates a job from other jobs, eliminating the need of synchronization between jobs.
- Each **unit** has an automatically assigned **Global Unique Identifier (GUID)**, used to check if it already exists on the remote computer.
- The **unit** constructor receives the global constant data, which is sent only once for

each remote computer, regardless the jobs number run on that computer.

- 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.

The second concept of our model is named **with** and it implements the invocations scheduler. It receives two parameters, a **destination** and 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. The **unit** constructor call used as the second argument of **with** has two functions: it specifies the specific job and it defines the global constant data used by each job.

At the beginning of the **with** construction a new scheduler is created. The scheduler has many functions, such as: resources discovery, code deployment, adds the invocations to the waiting list, load balancing, remote invocations, error recovery and results retrieval.

The third concept of our model is named **run** and it is used to create a new job and add it to the waiting list. It has two sets of arguments: a unique identifier for each job which is used to identify each job inside a destination and the specific data for each job. The **run** call is asynchronous and it returns immediately. The **run** calls occurs inside a **with** body (including the functions called from it) and they can appear multiple times.

At the end of the **run** body a synchronization point is automatically inserted. At this point is waited for the computation of all the enqueued jobs and their processing into the destination. In this way, after the **run** body ends, all the **run** calls are processed.

We analyzed our model from a theoretical point of view and highlighted its best use cases. We also implemented our model as a virtual machine, so we can implement the required language level features, and tested it both on a computer network and on the CPU cores of a single computer. In the case of a computer network we started with one computer and added new computers, one by one, measuring the speedup. 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. For a higher number of computers, the traffic time and network setup, which are constants, started to contribute in a more significant percentage, so the overall speedup is lower. This is consistent with the behavior predicted by our theoretical model. On the tests using the cores of a single CPU, the speedup has shown a growth close to the best case. Regarding the workload distribution, in this situation the maximal difference in percents from the optimum in the test results was 0.6%.

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). If the model is implemented as a framework, this is 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:

- It is as mainstream language, so the framework can be used by many programmers.
- Java bytecode is OS and CPU independent, so the deployment on heterogeneous architectures is easier.
- Java offers strong reflection capabilities so a code can introspect itself. This introspection is used 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 static data from a unit).

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. Each distributed computing computer is running a dedicated server. 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.

On the application side, the framework consists on some classes and interfaces, which implements our model. The most important are:

- *public interface **Distributed**<InitialData,TaskIdx,RunData,ReturnType>* - implements the **unit** concept. The client code which implements this interface and all its dependencies are deployed by the scheduler on the remote computers. **Distributed** has two methods:
    - *boolean **dInit**(final InitialData initialData)* - it receives the global data and it is called only once at the creation of the new instance.
    - *ReturnType **dRun**(final TaskIdx taskIdx,RunData rData)* - it is called for each task with the task GUID and its specific data.
- *public interface **Destination**<TaskIndex,ReturnType>* - defines a destination for the job's results. It has a single method:
    - *void **set**(TaskIndex index,ReturnType retData)* - when a new result arrives, it is sent by the scheduler togheter with its GUID to this method. Its implementation can for example put the result in a vector, write it to a file, etc.
- *public class **Scheduler**<InitialData,TaskIndex,RunData,ReturnType>* - implements the **with** concept. For automatic system initializations (such as resources discovery), **Scheduler** also has a static constructor. Some of its methods are:
    - *public **Scheduler**(final Class<?> distrClass, final InitialData initialData, Destination<TaskIndex,ReturnType> dest)* - a constructor which receive a handle to an implementation of the **Distributed** interface, which is the **unit** to be deployed, the global data and a destination which will receive the results of the computations.
    - *public void **addJob**(TaskIndex index,RunData rData)* - asynchronously adds a new job to the waiting list. The job is defined by its GUID and instance data.
    - *public void **waitForAll**()* - implements the synchronization point at the end of the scheduler, after all the jobs were added. This method waits for the computation of all the results and their processing the destination.

We tested our framework in the same way we tested the virtual machine for our model, in order to also check the model implementation consistency. There were two types of tests, one in a computer network and one using the cores of a single CPU. For the network tests, in the case of a low computers number, the obtained speedup is near the optimum. If we increase the number of the computers, we obtain a lower speedup because the algorithm finishes quite quickly. In this case invariant factors such as threads and sockets management or resource discovery accounts for a larger part of the execution time. For the tests using the cores of a single CPU, the speedup when new cores are added is near the optimum, with a maximum difference of 0.09% from the optimum. The distribution of the workload among different cores had a difference in percents of maximum 0.6% from the optimum.

6. Algorithm for hybrid execution on both CPU and GPU

This novel 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. Due to this efficient CPU/GPU split, any suitable code for GPU execution can benefit from the speedup offered by GPUs, even if this code is a component of an algorithm that cannot be run entirely on GPU.

The proposed algorithm runs on multithreaded applications by accumulating over multiple CPU threads the invocations arguments of the functions which are to be run using GPUs, it runs the functions on GPUs as a single package 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 CPU cores are used to execute other threads, so the CPU is fully used.

In order to reduce the context switch overhead [16] when a CPU core is used for multiple threads (in preemptive multitasking), we kept the threads at a lower number and inside each thread we used fibers/coroutines (cooperative multitasking). In this way we are capable to run massively parallel applications, with a much higher simultaneous parallelism than if we were using only threads, and also at a much lower switching overhead. In the same time, by using explicit switching, many synchronization needs are eliminated, because the programmer knows at any time where the execution point is and in a cooperative model only one fiber is active at the time. The term "thread" will be used for preemptive threads; for cooperative threads we will use the term "fiber".

Each GPU kernel has its own scheduler. This scheduler is responsible to accumulate the invocations over multiple fibers, run them as a single package on GPU and resume the original fibers using the computing results. The main body of the algorithm is listed below:

- For all GPU kernels initialize their schedulers
- For every CPU core create a new thread and add it to a threads list
- Use a circular iterator to iterate the list of threads in a circular manner and consider a current thread as given by this iteration
- For each parallel task:
    - 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 inside 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

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. After an iteration of the threads, the 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.

We analyzed the performance of the algorithm and also the factors which contributes to the overhead incurred by the GPU execution, such as memory transfers, the differences between GPU and CPU cores [17] and the number of the threads. To test our algorithm we implemented it in C++ and we made a series of tests in which we modified different factors such as the computation volume required by each thread. We also modified the number of the threads. The test results shown that when the total workload was bigger by ~11.8 times, the application execution time when we used the GPU was increased by only ~20.1%, while in the same time when only the CPU was used the increase was ~774%. For the execution time of the application this is an important progress, achieved when a GPU was employed.

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.

The algorithm uses runtime reflection in order to access the application bytecode, followed by disassembly, analysis and code generation, in order to generate OpenCL from the relevant code. In the execution phase required data is automatically serialized and transferred to GPU. On results retrieval, our algorithm converts the received data back to the data structures of the application. The GPU execution is abstracted as much as possible. The same 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 bytecode, 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 [18] 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. As a disadvantage in our case, because the programmer does not need to program in an OpenCL coding style, the translation to OpenCL may cause some loss of performance.

The tasks scheduler enqueues the user provided tasks, run the tasks using the GPU and receives their results. When new tasks 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.

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. Our serialization algorithm directly copies the primitive JVM data types. The reference types are handled according to two situations: arrays and class instances. The first member is in both cases a unique identifier 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 the unique identifier and after it follows the array elements. The arrays of primitive types are separate types.

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.

For code generation, in the first step the bytecode execution is simulated linearly (without branching or looping), 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. 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. 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. 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.

To implement memory allocation, which in a massively parallel system can create bottlenecks if not properly designed [19], we created a lightweight and fast memory allocator, on the same principles as [20], which for now is capable to only allocate data. 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. Because the returned values were used for exceptions handling, the values returned by the Java non-void methods were returned using a supplementary parameter, passed by address. 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. We also implemented a Java library with OpenCL functions such as cross and dot products. All calls to this library are directly translated to the native OpenCL correspondents.

We tested our algorithm on different metrics, such as performance and the range of

JVM code which can be translated. The maximum speedup when the GPU was used was 12.65x. We also compared our results with the results obtained using the Aparapi library on a similar test program. The Aparapi version 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) and required proxy code in order to serialize and deserialize the data of the application to and from an OpenCL suitable representation.

8. Conclusions

We consider the following aspects as the main contributions which we researched and developed in this PhD thesis.

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.

A Java framework which implements our model, so it can be used in regular Java applications. We are able to implement most of its semantics and constraints only by means of data structures and method 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.

A cooperative CPU-GPU execution algorithm, suitable for complex tasks that are not suitable for GPU only execution, for example if they contain I/O calls. Our algorithm 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.

An algorithm and library which translates Java bytecode to OpenCL and execute it on GPU. The library also handles automatically tasks such as data serialization/deserialization, 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.

Bibliography

[1] A. S. Tanenbaum, M. Van Steen, “Distributed Systems – Principles And Paradigms”, 2002
[2] Berkeley Open Infrastructure for Network Computing (BOINC) Project, Berkeley University, http://boinc.berkeley.edu/
[3] J. Dongarra, T. Sterling, H. Simon, E. Strohmaier, "High performance computing: Clusters, constellations, MPPs, and future directions", “Lawrence Berkeley National Laboratory”, 2003
[4] G. R. Andrews, "Foundations of multithreaded, parallel, and distributed programming", Pearson, 1999
[5] J. Dean, S. Ghemawat, “MapReduce: simplified data processing on large clusters”, Commun. ACM 51, 2008
[6] L. Dematté, D. Prandi, “GPU computing for systems biology”, Briefings in Bioinformatics, Vol. 11, No. 3, pp. 323-333, 2010
[7] Khronos Group, “The open standard for parallel programming of heterogeneous systems”, https://www.khronos.org/opencl/, access time: 14.07.2015
[8] 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
[9] 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
[10] K. Popović, Z. Hocenski, “Cloud computing security issues and challenges”, Proceedings of the 33rd International Convention, Croatia 2010
[11] 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
[12] V. Narvaez, “Distributed Objects and Remote Invocation”, Addison Wesley 2010
[13] 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
[14] 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
[15] 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
[16] 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
[17] K. P. Raphael, “Multi-core architectures and their software landscape”, Computing Science Handbook, Vol. 1, Chap. 35, 2013
[18] AMD, “Aparapi”, https://github.com/aparapi/aparapi, access time: 16.07.2015
[19] 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
[20] 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