The IPSI BgD Transactions
on
Advanced Research

Multi-, Inter-, and Trans-disciplinary Issues in Computer Science and Engineering

A publication of

IPSI Bgd Internet Research Society
New York, Frankfurt, Tokyo, Belgrade
January 2014 Volume 10 Number 1 (ISSN 1820-4511)

Guest Editor: Filipovic Nenad

Table of Contents:

Optimization Procedures During Parallelization of Specialized Software for Fluid Flow Simulations
Filipovic, Nenad; Djukic, Tijana ................................................................. 1

Parallelization of Specialized Fluid Flow Simulator Based on Lattice Boltzmann Method on a Multi GPU System
Djukic, Tijana; Topalovic, Marko; Filipovic, Nenad ........................................ 8

Modeling of the Behavior of 220Rn Progeny In Diffusion Chamber Using CUDA
Stanojevic, Adam; Ivanovic, Milos; Stevanovic, Nenad; Filipovic, Nenad ................ 13

Parallel Algorithms for Statistic Modeling of Dam Behavior
Ivanovic, Milos; Nikolic, Srdjan; Stojanovic, Boban; Milivojevic, Nikola; Filipovic, Nenad .... 19

Modeling of Selfhealing Materials Using Dissipative Particle Dynamics Method and Parallelization at Architectures of Shared and Distributed Memory
Petrovic, D., Obradovic, M., Jovanovic, A., Jovanovic, S., Filipovic N .................. 23

Scientific Applications in Cloud Computing
Ogrizovic, Dario; Car, Zlatan; Kovacic, Bozidar .............................................. 27
The IPSI BgD Internet Research Society

The Internet Research Society is an association of people with professional interest in the field of the Internet. All members will receive this TRANSACTIONS upon payment of the annual Society membership fee of €100 plus an annual subscription fee of €1000 (air mail printed matters delivery).

Member copies of Transactions are for personal use only

IPSI BGD TRANSACTIONS ON ADVANCED RESEARCH
www.internetjournals.net

STAFF

<table>
<thead>
<tr>
<th>Veljko Milutinovic, Co-Editor-in-Chief</th>
<th>Jakob Salom, Co-Editor-in-Chief</th>
<th>Nemanja Trbovic, Journal Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department of Computer Engineering</td>
<td>Department of Computer Science</td>
<td>Department of Computer Science</td>
</tr>
<tr>
<td>ETF</td>
<td>Mathematical Institute of SANU</td>
<td>IPSI Internet Research Society</td>
</tr>
<tr>
<td>University of Belgrade</td>
<td>University of Belgrade</td>
<td>University of Belgrade</td>
</tr>
<tr>
<td>POB 35-54</td>
<td>POB 367</td>
<td>POB 35-54</td>
</tr>
<tr>
<td>Belgrade, Serbia</td>
<td>Belgrade, Serbia</td>
<td>Belgrade, Serbia</td>
</tr>
<tr>
<td>Tel: (381) 64-1389281</td>
<td>Tel: (381) 64-8183030</td>
<td>Tel: (381) 60-3191011</td>
</tr>
<tr>
<td><a href="mailto:vm@eft.rs">vm@eft.rs</a></td>
<td><a href="mailto:jakob.salom@yahoo.com">jakob.salom@yahoo.com</a></td>
<td><a href="mailto:ntrbovic@gmail.com">ntrbovic@gmail.com</a></td>
</tr>
</tbody>
</table>

EDITORIAL BOARD

<table>
<thead>
<tr>
<th>Lipkovski, Aleksandar</th>
<th>Gonzalez, Victor</th>
<th>Milligan, Charles</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Faculty of Mathematics, Belgrade, Serbia</td>
<td>University of Oviedo, Gijon, Spain</td>
<td>Sun Microsystems, Colorado</td>
</tr>
<tr>
<td>Blaisten-Barojas, Estela Janicic, Predrag Kovacevic, Milos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>George Mason University, Fairfax, Virginia</td>
<td>The Faculty of Mathematics, Belgrade</td>
<td></td>
</tr>
<tr>
<td>USA Serbia</td>
<td>Serbia USA</td>
<td></td>
</tr>
<tr>
<td>Crisp, Bob Jutla, Dawn Neuhold, Erich</td>
<td></td>
<td></td>
</tr>
<tr>
<td>University of Arkansas, Fayetteville, Arkansas</td>
<td>Sant Marry's University, Halifax Canada</td>
<td>Research Studios Austria, Vienna Austria</td>
</tr>
<tr>
<td>USA Canada</td>
<td>Austria USA</td>
<td></td>
</tr>
<tr>
<td>Domenici, Andrea Karabeg, Dino Piccardi, Massimo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>University of Pisa, Pisa Italy</td>
<td>Oslo University, Oslo Norway</td>
<td>Sydney University of Technology, Sydney Australia</td>
</tr>
<tr>
<td>USA Italy</td>
<td>Australia USA</td>
<td></td>
</tr>
<tr>
<td>Flynn, Michael Kiong, Tan Kok Radenkovic, Bozidar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stanford University, Palo Alto, California</td>
<td>National University of Singapore</td>
<td>Faculty of Organizational Sciences, Belgrade Serbia</td>
</tr>
<tr>
<td>USA</td>
<td>USA</td>
<td></td>
</tr>
<tr>
<td>Fujii, Hironori Kovacevic, Branko Rutledge, Chip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fujii Labs, M.I.T., Tokyo Japan</td>
<td>School of Electrical Engineering, Belgrade</td>
<td>Purdue Discovery Park, Indiana USA</td>
</tr>
<tr>
<td>Fujii, Hironori Covacevic, Branko Rutledge, Chip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Paris University, Paris France</td>
<td>ICTEK Worldwide L'Aquila</td>
<td>University of Szeged, Szeged Hungary</td>
</tr>
</tbody>
</table>
Optimization Procedures During Parallelization of Specialized Software for Fluid Flow Simulations

Filipović, Nenad; and Đukić, Tijana

Abstract — Modern fluid flow simulations can be extremely complex and computationally demanding. Using GPU devices (Graphics Processing Unit) they can execute up to several tens of times faster and simulations can be observed interactively. In this study the basic principles of GPU programming are applied to the implementation of lattice Boltzmann (LB) method. The software that was developed based on the basic LB equations is parallelized and a discussion is given about certain improvements made on the initial implementation. The developed software was tested on a Tesla GPU device and significant speed-up is obtained, when comparing to the traditional version of the software. Fluid flow simulations in the field of biomeedicine that needed up to a few hours to be performed, can now be finish in just a few minutes.

Index Terms — graphics processing unit, lattice Boltzmann method, CUDA architecture, execution time comparison, parallelization speed-up

1. INTRODUCTION

Scientific methods used to include only observation, hypothesis and experimentation, but in the past 20 years computer simulations have become a new scientific paradigm [1]. Modern methods of computer scientific simulation of various phenomena, such as fluid flow simulations, can be extremely complex. Computational demands of these applications are high and large computational resources are required in order to perform these simulations. GPU devices (Graphics Processing Unit) represent a good choice for applications with high requirements in terms of computational resources. These devices have drawn much attention since they could be used in a wide range of general-purpose applications [2,3]. GPU devices execute hundreds or even thousands of threads simultaneously and thus accelerate calculations. The greatest benefit is that GPU device works as a coprocessor of the main computer, and together they form an effective system. Also it is not necessary to introduce large modifications in existing computer programs, thanks to a specially designed CUDA architecture (Compute Unified Device Architecture). With the help of GPU devices, programs that used to execute for a few days or hours, can now execute up to several tens of times faster. This way, simulations of many different phenomena can be observed interactively. There are many examples of successful application of GPU devices in different areas [4-6].

The problem of fluid flow simulation can be numerically solved using a wide range of methods, such as a continuum based finite element method [7] or discrete methods like dissipative particle dynamics [8]. This paper considers fluid flow simulations using a discrete method called lattice Boltzmann (LB) method. This method observes the fluid as a set of fictitious particles and by studying the dynamics of these particles (the collisions between them) the fluid flow is modeled on the macroscopic level. The greatest advantages of LB method are the simplicity of implementation and the possibility of parallelization.

Nowadays, in order to obtain high-accuracy in simulations of fluid flow in complex geometries, it is necessary to define a fine mesh, which results into many thousands of nodes. And if it is expected to get results in a reasonable computing time, it is necessary to parallelize the software. Different CFD solvers were parallelized using GPU devices and the obtained speed-ups were published in literature, including more traditional methods [9,10], but also lattice Boltzmann method [11-13].

In this study the basic principles of GPU programming are applied to fluid flow simulations, namely to the implementation of lattice Boltzmann method. The first implementation is further optimized and the improvements in terms of obtained speed-up are discussed. The developed software is tested on a specialized Tesla GPU.
device and a speed-up of 21 times is obtained, when comparing to the traditional version of the software.

The paper is organized as follows. In Section 2 the theoretical background of lattice Boltzmann method and the final form of equations used in numerical implementation of LB method are presented. Some details of the implementation are the subject of Section 3. Parallelization of the software is explained in Section 4. Section 5 concludes the paper.

2. THEORETICAL BACKGROUND OF LATTICE BOLTZMANN METHOD

The lattice Boltzmann method belongs to the class of problems named Cellular Automata (CA) and observes the physical system in an idealized way, so that space and time are discretized, and the whole domain is made up of a large number of identical cells [14]. Single distribution function is defined that represents the probability for particles to be located within a certain space element. This function in a specific lattice cell depends on the state of neighboring cells and it has an identical form for all cells. The state of all cells is updated synchronously, through a series of iterations, in discrete time steps. The derivation procedure of the lattice Boltzmann method can be found in literature [15,16] and only the initial and final equations will be given in this paper.

In the presence of an external force field, one can write a distribution function balance equation – Boltzmann equation:

\[
\frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} + \frac{\partial f}{\partial \mathbf{n}} = \Omega
\]

where \(\Omega\) is the collision operator.

Since this operator is represented using a very complex expression, a simplified model is introduced, initially proposed by Bhatnagar, Gross and Krook [17]. This model is known as the single relaxation time approximation or the Bhatnagar-Gross-Krook (BGK) model. Operator \(\Omega\) is defined as follows:

\[
\Omega = \frac{1}{\tau}(f - f^{eq})
\]

where \(\tau\) is the relaxation time (the average time period between two collisions) and \(f^{eq}\) is the equilibrium distribution function, the so-called Maxwell-Boltzmann distribution function.

Finally, BGK model of the continuous Boltzmann equation is given by:

\[
\frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} + \frac{\partial f}{\partial \mathbf{n}} = -\frac{1}{\tau}(f - f^{eq})
\]

The original BGK Boltzmann equation is continuous in space domain and is related to continuous velocity field. This form is not suitable for numerical implementation. In order to develop a program that numerically solves this equation on a computer, the equations have to be previously discretized. After discretization, the macroscopic quantities are evaluated in terms of the distribution function, as weighted sums over a finite number of discrete velocities.

The discretized equation that is numerically solved in software based on LB method is given by:

\[
jf(x+v,t+\tau) - jf(x,t) = -\frac{1}{\tau}(jf(x,t) - jf^{eq}(\rho, u)) + \left(1 - \frac{1}{2\tau}\right)\mathcal{F}
\]

where \(\mathcal{F}\) is the modified relaxation time, introduced to provide more appropriate time discretization and better numerical stability of the solution.

When LB method is implemented on a computer, this equation is most commonly solved in two steps – the collision step and the propagation step. Two values of the distribution function can be defined - \(f^\text{in}\) and \(f^\text{out}\), that represent the values of the discretized distribution function before and after the collision, respectively. The mentioned steps are expressed as follows:

Collision step:

\[
jf^\text{in}(x,t) =jf^\text{in}(x,t) - \frac{1}{\tau}(jf^\text{in}(x,t) - jf^{eq}(\rho, u)) + \left(1 - \frac{1}{2\tau}\right)\mathcal{F}
\]

Propagation step:

\[
jf^\text{out}(x+v,t+\tau) =jf^\text{out}(x,t)
\]

Each one of these steps must be applied to the whole system (to all nodes of the lattice mesh, i.e. to all particles) before the next step starts. The step that considers collisions is a completely local operation (the equations for every node are independent and not coupled), while the step that considers propagation takes into account only the currently considered node and a few closest neighboring nodes. This is why the described scheme and LB method are particularly convenient for parallelization.

3. IMPLEMENTATION OF LB METHOD

A specialized software based on lattice Boltzmann method is developed. The basic idea of the implementation of LB method is to divide the observed domain on a certain number of identical elements that are also called cells. For two-dimensional problems the type of mesh that is used is denoted by D2Q9, because it is a two-dimensional domain. For every node of this mesh 9 components of distribution function are defined. Figure 1 shows the domain in two-dimensional space and the lattice mesh, and also shows the directions of the discretized distribution function for one node of the mesh.

The program for the simulation of fluid flow based on LB method is written in programming language C++. There are functions that were specifically developed to manipulate data related to nodes individually, to the entire mesh (within the mesh the interactions between nodes are considered), as well as the entire simulation (that controls all other activities). All of these functions are implemented within several classes, such that there is particular class that defines a node, the mesh and the simulation. It is important to
emphasize that there must exist a loop of iterations within the program, since the problem is solved in a large number of time steps, until the steady state is reached.

4. Parallelization of the LB Solver

GPU devices are especially adapted to solve problems that can be described as processing a large number of arithmetical operations over a large data set. In other words, when the same series of operations needs to be performed on a large data set in parallel, with a small number of memory operations and a large number of arithmetical calculations – it is suitable to use GPU devices. A computer program that performs this type of operations would execute sequentially on a standard PC computer, while on the GPU device the data is transferred into the device memory and is divided on thousands of parallel-processing threads. Applications that operate with long arrays or matrices with large dimensions are appropriate for parallelization using this approach and in these cases a considerable speed-up of the calculations is achieved.

In this paper special software architecture called CUDA (abbreviated from Compute Unified Device Architecture) [19] was used for programming of GPU devices using upper level programming languages, like C and C++. CUDA is actually an extension of the programming language C, with a specific set of new keywords that are used for GPU commands. CUDA programming enables the source code to be executed on two different platforms, on a so-called host system (that represents the CPU) and a device system (that consists of one or more GPUs).

The task of programmer is to modify the existing program and to separate parts that perform a large number of calculations from the rest of the program and to transfer this data to the GPU device. The rest of the program is executed on the CPU. During the execution of the parallel version, the GPU works as the coprocessor of the CPU. Large data set on which the arithmetical calculations are performed is passed on to the GPU, while CPU interprets and transfers the necessary information.

In order to execute part of the program on a GPU device, the programmer must define specialized functions that are called kernels. Every kernel function is called from the main program and is executed on a grid of kernels. Within every kernel call it is necessary to define the number of threads and dimension of the grid and of course to pass the appropriate arguments.

4.1 NVIDIA Tesla

Most of modern graphics cards, that are part of the configuration of modern personal computers, can be used as a GPU device. However, NVIDIA has also offered specialized GPU devices – the Tesla series [20]. Tesla devices are optimized for execution of complex scientific simulations. The main difference between a typical graphics card and a Tesla device is that Tesla does not have the possibility to output images to a display. For parallelization of LB solver, implementation and testing of improvements and execution time measurements Tesla C2075 device was used. This device is part of the Curie supercomputer, owned by GENCI, located in France and it represents PRACE (Partnership for Advanced Computing in Europe) Research Infrastructure resource. We were able to use it since our project was rewarded with access to this cluster within one of preparatory project access calls.

4.2 Parallelization of the software based on lattice Boltzmann method

In the implementation of lattice Boltzmann method the collision and propagation step (defined in Equations (5) and (6)) are repeated in a predefined number of iterations (or until a condition is satisfied). As it was already stated in section 2, each of these steps must be applied on all lattice nodes, before the next step starts. Since the equations that simulate collisions between particles are mutually independent, this step is a completely local operation. In the propagation step only the currently observed node and its closest neighbors are considered. All this indicates that this large data set can be divided to smaller subsets and then a certain number of (mostly arithmetical) operations should be performed on these subsets in parallel. Therefore it is evident that LB method is very suitable for parallelization. Figure 2 shows the algorithm of execution of standard (sequential) LB solver, with marked part of the algorithm that can be parallelized. In the same Figure parts of the source code used to implement the two mentioned steps are stated.
Since it was concluded that the collision and propagation steps are local in nature and that they can be easily parallelized, it is necessary to develop two kernel functions that will be called instead of regular functions used in standard version of LB solver. Figure 3 shows the calls of these functions in sequential (regular) and parallel form.

When two kernels required for collision and propagation step are implemented, they are called instead of the equivalent regular (sequential) functions. However, there is another important aspect that needs to be considered when existing programs are parallelized - the memory manipulation. Namely, GPU device is not a completely independent device, since the CPU controls the execution process on the GPU. CUDA architecture is designed in such a manner that CPU (also called host) and GPU (also called device) have separate DRAM (abbreviated from Dynamic Random Access Memory), which in fact is the case, if it is observed from hardware point of view. They are called host and device memory, respectively. CUDA allows the program to access these memories independently, but there are certain limitations in this process. Part of the program that is executed on the CPU has access only to the host memory, while kernel functions have access only to the device memory. Therefore it is necessary to explicitly allocate and free memory resources on both systems, as well as transfer data between these two types of memory. Special CUDA functions are providing the transfer of data from the CPU to the GPU and vice versa.

The easiest way to parallelize LB solver would be to save all the necessary data about lattice nodes in host memory. Before every kernel call this data would be transferred to device memory and after the kernel execution the new data would be transferred back to the host memory. But when transferring memory it is very important to analyze the effects that communication between CPU and GPU will have on the performance of parallelized applications. Thus the described approach is completely wrong, because the transfer of data between these two memories is a very "expensive" operation. Maximum bandwidth between device memory and the GPU is 102.4 GBps (Gigabytes per second) for Tesla C1060, while the bandwidth between host and device memory is around 8 GBps [21]. This means that it is necessary to minimize the transfers between host and device memory whenever it is possible. One should always balance between speed-ups obtained using GPU and the cost of data transfers between host and device.

In this concrete case, that means that all data about lattice nodes (the components of the distribution function and everything else) have to be initialized on the host, then this data is transferred only once to the device memory and during program execution they stay in device memory. Kernel functions access data in device memory and make the necessary changes. In order to view the results (visualize velocity and pressure field), only the macroscopic quantities can be transferred back to host memory. These macroscopic quantities are calculated in a specialized kernel function. It should be noted that fluid velocity and pressure fields are not visualized after every iteration, but most commonly after 100, 1000 or even more iterations, depending on the problem that is being simulated. This means that the amount of data transferred from host to device is significantly reduced. There are some other aspects of memory manipulation that need to be considered when implementing kernel functions. There is a hierarchy in GPU device memory, i.e. threads can access data from memory at different levels. Every thread has its own private local memory and only that particular thread has access to it. Every block of threads has its own shared memory that is accessible to all threads within that block. And finally, all threads have access to global device memory. It does not take the same time for a thread to access data from local (register), shared or global memory. The least amount of time is needed to access register memory, more time is needed when accessing shared memory and the access to global memory requires the greatest amount of time [20; 22] (in some cases access to global memory is up to 150 times slower than access to register memory). Thus, access to global memory should be avoided whenever possible.

If all these facts are taken into consideration when performing parallelization of LB solver, instead of using two kernel functions for collision and propagation steps, it is more appropriate to implement one function that combines these two steps. That does not make any changes in the theoretical background, since the two steps are still performed separately. This is just a matter of implementation. Values of the distribution function
function $f_{\text{out}}(x,t)$ in equations (5) and (6) (at the end of collision step and at the beginning of propagation step) are simply not written into the global device memory and then read again, but are written in register memory of each thread individually. In order to ensure that the collision step is finished before the start of the propagation for all nodes (i.e. for all threads in the kernel function), a system-defined CUDA function is used to synchronize thread execution - `__syncthreads()`.

This function ensures that all threads are done with the execution of all previous operations before proceeding with kernel execution.

Figure 4 shows the algorithm of execution of parallelized LB solver.

5. RESULTS AND COMPARISON OF EXECUTION TIME OF REGULAR AND PARALLELIZED VERSION OF LB SOLVER

In the sequel the speed-up obtained with the parallelized version of LB solver will be presented. First the importance of proper memory manipulation in CUDA programs will be demonstrated. As a test example the simulation of straight stationary flow between two parallel walls is used. The execution time of the program mainly depends on the dimension of the domain (the total number of nodes in lattice mesh) and on the number of iterations. In this case the performance tests have been performed using the total number of 20,000 nodes and the number of iterations is set to 20,000. The number of iterations is chosen such that the simulations can be finished in a reasonable amount of time and the obtained results are relevant because the computational load remains the same for every iteration and the insight that can be obtained using these test simulations can be used to estimate the time needed for more complex simulations.

The execution time of regular LB solver was compared with execution time of three different versions of parallelized LB solver. The first version transfers data from device to host memory and vice versa after every iteration (after every kernel call). The second version is optimized from the aspect of memory transfers – an additional kernel function that calculates macroscopic quantities is introduced and only macroscopic quantities are transferred back to host memory, to visualize the results. In the third version the access to global device memory is minimized, i.e. the collision and propagation steps are combined in one kernel function, like it was already described in Section 4.2.2. The comparison of execution time (expressed in seconds) is shown in Figure 4.

From Figure 4 it is obvious that bad memory manipulation can have a great influence on the
execution time. The first version of parallelized LB solver is executed almost 10 times slower than the sequential version, which is of course unacceptable. Proper memory manipulation (like in versions 2 and 3) has brought a speed-up of almost 10 times compared to the regular version, which is an expected speed-up. If the execution time of second and third version is compared, it can be seen that the third version is executed faster, due to a more optimal approach to global memory manipulation, but the difference is not as extreme as with the first version.

\[ S = \frac{1}{(1 - P) + \frac{P}{N}} \]  

where \( P \) is the fraction of the program that can be parallelized (the easiest way to determine is by dividing the execution time of that particular portion with the overall execution time of the program), and is the number of processors.

In case of LB solver it can be considered that the number of processors is equal to the number of CUDA cores – 240. However, if the expression (7) is analyzed in detail, it becomes evident that as the number of processor increases, the ratio decreases and becomes a lower order member that can be neglected. Therefore with the increase of number of processors, this variable actually does not provide a greater performance benefit. It can be considered that this conclusion is valid in this particular case too.

Portion of the LB solver that is not parallelized includes the initialization of data and visualization of results. Therefore one can say that the portion of the program that is not parallelized is around 4.5% of the overall program, which means that .  

If the values for and are substituted in the initial expression, it is obtained:

\[ S \approx 21 \]  

Hence it is estimated that the maximum speed-up that can be achieved by parallelization of LB solver (and applying one Tesla C1060 GPU device) is 21.

When the execution time of the parallelized LB solver is compared with the regular solver, it is necessary to keep in mind the number of lattice nodes, i.e. the dimension of matrices on which the operations are performed. Figure 5 shows how the variation of speed-up of the parallelized LB solver is changing depending on the number of lattice nodes. As the number of nodes increases, the speed-up increases as well. Practically this means that the parallelized LB solver achieves greater speed-ups when simulation parameters are set such that the whole simulation is more computationally demanding. If the speed-up is determined based on the execution time of the entire program, the maximum value is 16.5 times (for 80.000 nodes). Further increase of the number of nodes does not affect the speed-up.

The achieved speed-up of 16.5 times is slightly lower than the predicted value of 18.5 times. This can be explained with the fact that even though they are minimized, the transfers of data between host and device memory do exist and a certain amount of time is required for their execution, and this time is not directly spent on operations that are normally performed in the regular LB solver. Figure 6 shows the values of speed-up calculated based on the execution time measured only for the portion of the program where calculations are performed (the iterations loop shown in Fig. 5, without visualization of results). In this case, the obtained speed-up is up to 21 times, which is approximately equal to the predicted value.

6. CONCLUSION

It seems that scientific simulations demand more and more computational resources. Personal computers can no longer follow up these increased demands. NVIDIA has offered a good solution for solving this emerging problem with its series of Tesla GPU devices. Applying GPU devices a considerable speed-up can be achieved in existing complex and demanding programs. Thanks to a specialized technology
that was developed by NVIDIA, the so-called CUDA architecture, customizing the existing programs to work with GPU devices is significantly simplified. In this paper the developed software based on lattice Boltzmann method was parallelized using CUDA architecture and the execution time of regular and parallel version was compared. On a specialized Tesla device the speed-up of 21 times was achieved, which represents a great time saving. It was shown that the developed parallel LB solver achieves greater speed-ups when simulations are more computationally demanding. It was also shown that the cost of CPU-GPU communication in memory manipulation requires special consideration from programmer’s side, in order to obtain greater performance improvements.

REFERENCES


Abstract — Computational demands of fluid flow simulations are high, with large computational resources required to perform the calculations and these applications have recently been accelerated with the help of GPU devices (Graphical Processing Units). Fluid flow simulation using discrete method called lattice Boltzmann (LB) has also been parallelized using GPU. In this paper a single-node multi-GPU implementation of both two-dimensional and three-dimensional fluid flow LB simulation is presented. A configuration with 3 GPU devices connected to the same processor is used to test the developed multi-GPU software and to compare the obtained speed-up relative to the single-GPU implementation.

Index Terms — lattice Boltzmann method, GPU programming, CUDA architecture, OpenMP, multi-GPU application

1. INTRODUCTION

Modern computational methods are widely implemented in many scientific areas. Computational demands of such applications are high, with large computational resources required to perform the calculations. CPU technology has improved with years, but the applications require more and more computational power, that cannot be followed by the development of modern CPUs and desktop computers. High performance computing appears as an adequate solution. Many problems can now be solved interactively on a desktop computer paired with a GPU device (Graphical Processing Unit), that works as a coprocessor of the main computer. The specialized platform called CUDA (Compute Unified Device Architecture), developed by NVIDIA [1] enables the programmers to obtain significant speed-up of already developed applications with minor changes in source code.

There are many examples of successful implementation of GPU devices in existing programs. Some of the applications are inmolecular dynamics [2], processing and analysis of medical images [3], graph component labelling [4], block decomposition [5], DNA sequence alignment [6], bioinformatics pairwise sequence alignment [7], computation of shortest paths [8] and many others.

Fluid flow simulation using discrete method called lattice Boltzmann (LB) has also been parallelized using GPU devices [9-10]. However there are problems in biomedicine, like simulations of blood flow through human aorta or other arteries that are described with complex domain boundaries and require high accuracy. Therefore very fine mesh must be defined, with a large number of nodes. This type of simulation has large computational demands and even if it would be run on a single GPU device, it would not be good enough, neither from the aspect of execution time, nor from the aspect of memory requirements. Thus a multi-GPU application has to be developed. There are papers in literature reporting diverse implementations of LB codes on a multi-GPU cluster for simulation of flow of compressible fluid [11], thermal fluid flow [12], incompressible three-dimensional fluid flow [13], incompressible two-phase flow [14]. In this paper a single-node multi-GPU implementation of both two-dimensional and three-dimensional fluid flow will be presented. A configuration with 3 Tesla C1060 connected to the same processor is used to test the developed multi-GPU software. Also, during testing phase the Curie supercomputer was used. This supercomputer is owned by GENCI, located in France and is part of PRACE (Partnership for Advanced Computing in Europe) Research Infrastructure. Our project was rewarded with access to Curie cluster within one of preparatory project access calls, hence we were able to use GPU devices from the cluster.

The paper is organized as follows. In section 2 the basic equations of LB method are listed, as well as some implementation details, relevant for the parallelization procedure. In section 3 the parallelization procedure is discussed. Section 4 shows the results of speed-up obtained with multi-GPU system comparing to single-GPU
application. This section also shows the results of one high demanding simulation of human aorta. Section 5 concludes the paper.

2. LATTICE BOLTZMANN METHOD

Lattice Boltzmann method observes the fluid as a set of fictitious particles that are moving through the domain in a predefined set of directions. The set of directions for two cases considered in this paper (D2Q9 for two-dimensional domain and D3Q27 for three-dimensional domain) are shown in Figure 1. By studying the dynamics of these particles (the collisions between them and their further propagation) the fluid flow is modeled on the macroscopic level. The entire domain is divided on a predefined number of cells and the discretized domain is called lattice mesh. The greatest advantages of LB method are the simplicity of implementation and the suitability for parallelization.

The basic quantity that is used in LB simulations is the distribution function \( f \). This function is defined in all nodes of the mesh and it is updated synchronously in all nodes, through a series of iterations, in discrete time steps. The basic equation from which the whole method is derived is called Boltzmann equation and it is given by:

\[
\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f + \frac{g}{m} \frac{\partial f}{\partial \mathbf{v}} = \Omega \tag{1}
\]

where \( g \) is an external force field, and \( \Omega \) is the collision operator.

Collision operator represents the changes in the distribution function due to the inter-particle collisions. In this paper for the description of this operator a simplified model is used, proposed by Bhatnagar, Gross and Krook [15]. This model is known as the single relaxation time approximation or the Bhatnagar-Gross-Krook (BGK) model. Operator \( \Omega \) is defined as follows:

\[
\Omega = -\frac{1}{\tau} (f - f^{(0)}) \tag{2}
\]

where \( \tau \) is the relaxation time (the average time period between two collisions) and is the equilibrium distribution function, the so-called Maxwell-Boltzmann distribution function.

Since BGK Boltzmann equation is valid for continuum, it has to be discretized. The discretization procedure, as well as the derivation procedure of the whole LB method can be found in literature [16, 17]. After discretization, using the distribution function calculated in simulation, macroscopic quantities can be evaluated as weighted sums over a finite number of discrete velocities. The discretized equation that represents the LB numerical scheme and that is used when LB method is implemented and solved numerically is given by:

\[
f_i(x+v_i,t+1) - f_i(x,t) = -\frac{1}{\tau} (f_i(x,t) - f_i^{(0)}(\rho, \mathbf{u})) + \left(1 - \frac{1}{2\tau}\right) F_i \tag{3}
\]

where \( \tau \) is the modified relaxation time, introduced to ensure explicit time steps and better numerical stability of the solution, and \( F_i \) is the discretized external force term.

In numerical implementations, this equation is solved in two steps – collision and propagation step. Two values of the distribution function can be defined - \( f_i^{in} \) and \( f_i^{out} \), that represent the values of the discretized distribution function before and after the collision, respectively. The following equations represent mentioned steps:

Collision step:

\[
f_i^{in}(x,t) = f_i^{in}(x,t) - \frac{1}{\tau} (f_i^{in}(x,t) - f_i^{(0)}(\rho, \mathbf{u})) + \left(1 - \frac{1}{2\tau}\right) F_i \tag{4}
\]

Propagation step:

\[
f_i^{out}(x,v_i,t+1) = f_i^{out}(x,t) \tag{5}
\]

Both steps must be applied to all nodes of the lattice mesh synchronously. Equation that represents the collision step is independent for each node and therefore this is a completely local operation. Equation for the propagation step shows that here only the currently considered node and a few closest neighboring nodes have to be considered. It is evident from this analysis that LB method is suitable for parallelization.

2.1 Implementation details

The parallelization of the developed software based on lattice Boltzmann method is the topic of this paper. This program for the simulation of fluid flow is written in programming language C++. Several classes are implemented, for definition of appropriate function related to lattice node, lattice mesh and the entire simulation. The algorithm of the program is schematically shown in Figure 2.

![Figure 1 - Set of directions of the distribution function for two-dimensional D2Q9 lattice mesh (left) and three-dimensional D3Q27 lattice mesh (right)](image)

![Figure 2 - Two-dimensional mesh and discretized distribution function](image)
It is evident that there must exist a loop of iterations, because the problem must be solved in a large number of iterations, until the steady state is reached. Within this loop two steps defined with equations (4) and (5) are carried out.

3. **Parallelization of LB Solver**

The main task was to modify the existing program and to separate parts that perform a large number of calculations from the rest of the program. The data necessary for this calculation had to be transferred to memory residing on the GPU device. The rest of the program is run on the CPU and the CPU controls the execution of the entire program (calling GPU functions, gathering results etc.). The functions that have to be executed on the device are called kernels. The main CPU program manages calls of kernel functions and passes appropriate arguments, so that the kernel function can be executed in parallel on a grid of threads. Within these kernel calls it is necessary to define the parameters of this grid, i.e. the number of threads and dimension of the grid.

In the implementation of LB method the collision and propagation step are repeated in a predefined number of iterations. Two kernel functions were implemented and in the parallel version of the software they are called instead of functions used in sequential version of LB solver. Figure 3 shows the calls of these functions in sequential and parallel form. In the practical implementation these two functions are joined in one single function, to ensure better memory manipulation.

The single GPU version of LB solver was further improved, such that it can be run on a multi-GPU system, with several GPU devices on the same node. Here the concept of OpenMP was used [18]. Since several GPU devices will be used, kernel functions have to be called for every GPU individually. Also, data has to be allocated and initialized in memory of every GPU device individually. Therefore it was necessary to decompose the entire domain on N parts, where N is the number of GPU devices used. The decomposition is performed along x axis, like it is shown in Figure 4. This way every GPU device will be “responsible” for one part of the domain. Since the collision step is a completely local operation, no additional synchronization of devices is necessary. But, for the propagation step data from neighboring nodes is needed and thus the transfer of data from one GPU to its first neighbor is necessary. The data that has to be transferred consists of several components of the distribution function. Figure 5 shows the components that are transferred in a simulation of two-dimensional fluid flow, while Figure 6 shows the components that have to be transferred in a simulation of three-dimensional fluid flow.

4. **Results and Comparison of Speed-Up of Multi-GPU System Versus Single GPU System**

In this section the execution time of the parallelized LB solver running on a single GPU is compared with the execution time of the modified LB solver running on multiple GPU devices. It is necessary to consider the number of lattice nodes on which the operations are performed. As the number of nodes increases, the difference in measured execution time is bigger and the speed-up obtained using several GPU devices is greater.

As a test example the simulation of straight stationary flow between two parallel walls is used.
The total number of nodes varied, in order to show the mentioned dependence of speed-up from the number of nodes. The number of iterations is set to 20,000. This precise number of iterations is chosen because simulations can be executed in a reasonable amount of time. On the other hand, since the computational load remains the same for every iteration, the obtained results are relevant. Using these test simulations it is possible to gain insight of the execution time needed for more complex simulations.

Figure 7 shows the execution times of the sequential LB solver and parallelized LB solver running on one GPU. It is evident that the difference in time is increasing depending on the number of lattice nodes, the speed-up is greater for finer meshes. In this Figure the execution times were measured for a simulation of two-dimensional fluid flow. Figure 8 shows the same comparison of execution time of sequential and parallelized LB solver, but this time the simulation is performed in a three-dimensional fluid domain.

Figure 9 shows the measured execution time when LB solver was executed on one, two and three GPU devices. It is evident from the diagram that the speed-up of the application increases as the number of nodes increases. The final speed-up that was obtained when comparing LB solver running on one GPU and on two GPU devices is 1.9, while the program running on three GPU devices runs 2.9 times faster than the program running on one GPU. The scaling is not ideally linear due to the communication and transfer of data from memory residing on one GPU device to another.

5. CONCLUSION

In the last few decades LB method has become very popular in simulations of fluid flow. The main advantages of this method are simple implementation and natural parallelism. The in-house developed LB solver has already been parallelized and executed on one GPU device. In this paper this software was further improved to enable the usage of several GPU devices to perform the calculations. In order for the program to run correctly, the synchronization of data between GPU devices was necessary and it was explained in this paper. Also, the obtained results and comparison of execution times for LB solver running on one, two or three GPU devices are presented. These results show that the improved version runs significantly faster on a multi GPU system and can be used to model fluid flow with complex domain boundaries, represented by a fine mesh with large number of nodes. Simulations of fluid flow that needed up to a few hours to be executed, can now be finished in just a few minutes which is extremely valuable.

REFERENCES


Modeling of the Behavior of $^{222}$Rn Progeny in Diffusion Chamber Using CUDA

Stanojević, Adam; Ivanović, Miloš; Stevanović, Nenad; and Filipović Nenad

Abstract — Parallel program has been developed for simulation of Radon progeny behavior in the diffusion chamber. The program executes on general purpose graphics processing unit based on CUDA platform. Algorithm of the sequential version based on Brownian motion and diffusion has been rewritten for parallel use. During development, special attention has been paid to instruction execution mechanism, usage of different memory types and overall resource consumption. For random number generating, auxiliary linear congruent mechanism has been used. Serial version has been significantly outperformed in execution speed without any accuracy loss. All that produced significant speed improvement in acquisition of statistical data for Radon progeny behavior in the diffusion chamber.

Index Terms — CUDA, diffusion chamber, general purpose graphics processing unit, GPU, parallelism, Radon progeny, simulation

1. INTRODUCTION

The diffusion chamber (Figure 1) is cylindrically shaped device made of permeable material with radioactive particles detectors inside. Radon decays in the diffusion chamber and new short-lived progeny atoms are formed inside. Activity equilibrium between radon and its short-lived progeny is established inside the chamber. However, the radon progeny may deposit onto the inner wall of the chamber. This process of deposition changes the irradiation geometry and affects the detector sensitivity. Some progeny atoms decay in air (hereafter referred to as the air fraction) and others decay after deposition onto the walls (hereafter referred to as the deposited fraction) [5].

Very small particles of gas or liquid are moving in random pattern. Changing of the direction of motion is caused by collisions with molecules of the fluid (in our case the fluid is air). Such random movement is known as the Brownian motion. The random motion of particles is characterized by the direction and magnitude of velocity as well as the path length between two subsequent collisions. All these variables are random in nature.

If many particles are generated at the time $t = 0$ at the point $(x_0; y_0; z_0)$, after some time $t$, the particles will be distributed according to the Gaussian distribution [7]:

$$f(x, y, z, t) = \frac{1}{\sqrt{4\pi Dt}} e^{-\frac{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2}{4Dt}}$$

Equation 1 — Particle distribution

$D$ is the diffusion coefficient for a given set of particles. The diffusion coefficient $D=0.054\text{cm}^2\text{s}^{-1}$ was taken; the same value was also used in [6] and [8].

![Figure 1 - Diffusion chamber](image)

Unfortunately, Equation 1 is strictly valid for the gaseous molecule movement in a gas and not for particles in a gas. They are even less valid when the particles with different size move in a certain gas. Consequently, the Equation 1 is only approximation. Therefore, we decided to take strict but computationally more expensive approach than in [2] in order to avoid these approximations and simulate Brownian motion of the particles without any statistical assumptions.

The scheme showing the behavior of radon progeny inside the diffusion chamber is shown in Figure 2. The air fraction is in the left box and the deposited fraction in the right box of Figure 2.

Progeny atoms will decay in the chamber, but some of them in a chamber volume and other as deposited onto the inner chamber walls. The fraction of progeny that decay in air depends on life time of atoms. Longer living atoms have larger probability for deposition onto internal chamber wall and air fraction is consequently smaller. The simulation of diffusion process will be described in the next paragraph.

Manuscript received May 8, 2013. The part of this research is supported by Ministry of Science in Serbia, Grants III41007 and OI14028, and FP7 ICT-2007-2-5.3 (224927) ARTreat project. Filipović Nenad is with the Faculty of Engineering, University of Kragujevac, Serbia (e-mail: fica@kg.ac.rs).
Step 1. A $^{222}\text{Rn}$ atom is generated randomly inside the cylindrical diffusion chamber ($x_0; y_0; z_0$). The well known formula $r_{\text{rand}} = R \sqrt{\gamma_1}$ was used to sample random radius. The coordinates $x_0$ and $y_0$ were calculated as $x_0 = r_{\text{rand}} \cos(\phi_{\text{rand}})$ and $y_0 = r_{\text{rand}} \sin(\phi_{\text{rand}})$, where $\phi_{\text{rand}} = 2\pi \gamma_2 \cdot \gamma_1$ and $\gamma_1$ and $\gamma_2$ are standard random numbers, uniformly taken from the interval $[0, 1)$. This point was taken as the starting point ($x_0; y_0; z_0$) in Equation 1 for the diffusion of the $^{218}\text{Po}$ atom.

![Figure 2 - Radon progeny](image)

**Step 2.** Step 2 consists of random sampling of the life of the $^{218}\text{Po}$ atom, $T$, according to the Equation 2.

$$T = -\tau \ln(1 - \gamma_3)$$

**Equation 2 – random sampling of the $^{218}\text{Po}$ life**

Here, $\tau$ is mean life time given as $\tau = T_{1/2} / \ln 2$ and $T_{1/2}$ is half life; $\gamma_3$ is another uniform random number between 0 and 1.

**Step 3.** The particle moves to its next position with velocity taken from the Maxwell distribution. If this point is inside the diffusion chamber, the atom of $^{218}\text{Po}$ would decay in air; otherwise, the atom would deposit onto the chamber wall. If the $^{218}\text{Po}$ atom decayed in air, the point $(x; y; z)$ was taken as the creation point for $^{214}\text{Pb}$ ($x_0; y_0; z_0$) and steps 2 and 3 were repeated using the half life of $^{214}\text{Pb}$. Then the history was continued with $^{214}\text{Bi}$ and so on. For the sake of simplicity, and GPU speedup benchmarking, we only considered the first step of the decay (generating $^{218}\text{Po}$).

**Step 4.** Calculations of the point of deposition were performed in following way. If the point $(x; y; z)$ was outside the diffusion chamber, the progeny would be deposited onto the chamber wall. The line that connected the points $(x_0; y_0; z_0)$ and $(x; y; z)$ intersected the chamber wall at some point $(x_i; y_i; z_i)$. This point was taken as the point of deposition.

As described above, we performed direct simulation of the random motion of particles in the air volume inside the diffusion chamber. However, this simulation is very time consuming because a particle experiences about 109 collisions per second and moves randomly for very short distances. Creation of one particle history takes more than several hours on Core-i7 computers. Furthermore, in order to calculate the air deposited fractions or the distribution of deposited progeny, thousands of particle histories are needed. The approach to overcome this issue was to develop parallel algorithm. The implementation executes on general purpose graphics processing unit based on CUDA platform, which turned to be appropriate for this kind of MC modeling.

CUDA platform device consist of large number of computing cores grouped into multiple streaming multiprocessors (SMP). Those cores are neither fast (compared to CPU speed) nor x86 paradigm capable. Their strength and usability comes from massive processing throughput they achieve with this large number of independently functioning threads on device cores. Each streaming multiprocessor is capable of running user developed code called kernel on each of his cores. Kernel defines SIMD paradigm computation logic written in C-like syntax executed within threads running on device cores. Several types of memory reside on devices. The largest and the slowest is global memory, available for all of the threads. Way faster but smaller is shared memory available only for threads executed on cores of one SMP. The smallest and the fastest is register memory available within one core. Several generations of CUDA devices emerged up to this point. Each of them made advanced steps in overcoming difficulties concerning development for highly parallel problem solving. Those difficulties include optimized memory access and manipulation, device cores utilization in means of instructions used and problem partitioning and sharing data between independent threads. Each of these difficulties are also addressed in process of development this simulation algorithm for $^{222}\text{Rn}$ progeny in diffusion chamber.

First, we give the details about sequential algorithm, and then we describe parallel approach including all specifics that must be taken into account regarding CUDA. Finally, we present speedup obtained and plans for future developments.

## 2. SERIAL IMPLEMENTATION

Serial implementation of the simulation is consisted of successive iterations modeling motion of each observed particle. In each time step, the motion is conducted with the following actions:

1) Mean path length between particles collision is calculated using appropriate distribution. According to determined path
length using random numbers, new possible position coordinates for particle are determined.

2) Position and parameters of particles are being checked:
   a. If particle is still in the air within chamber, Maxwell speed and life time of particle are being recalculated. If particle life time is longer than life time determined for that particle, it decays; otherwise it continues motion within chamber starting from its new position.
   b. If particle deposition occurred on the cylinder inner surface, the position is logged (bottom, top or wall). If this particle motion time exceeded its life time, instead of deposition, we consider it decayed in the air.

Above described steps for each particle are being conducted until maximum allowed time for simulation is reached or desired number of particles with determined final position in chamber is reached. The implemented algorithm is shown in 1.

3. PARALLEL IMPLEMENTATION

Parallel implementation is based on scheduling engaged CUDA threads in such manner that each thread is responsible for a single particle. Serial implementation for each particle uses CPU as shared computing resource. Thus for each serial implementation iteration, each particle waits for its turn to calculate new position and velocity from the Maxwell distribution. In parallel implementation, the motion calculation is independent within the context of CUDA threads.

Algorithm 2: Parallel algorithm implementation - CPU host code

```plaintext
LoadConfiguration(particlesToDo, wallTime, blocks, threads, chunk);
InitDataForParticles(halfPath, halfLife, lifetime, [x; y; z], status);
while true do
    RecordTime(elapsedTime);
    foreach particle in particles do
        if particlesDone == particlesToDo then break;
        if elapsedTime > wallTime then break;
        if status[particle] != INIT then continue;
        impactDistance = Distribution(halfPath);
        [ϕ; θ] = RandomPolarCoordinates();
        [x; y; z] = NewCoordinates([x; y; z], impactDistance, ϕ, θ);
        position = GetPositionInChamber();
        if position == AIR then
            particleSpeed = CalculateMaxwellSpeed();
            lifeTime[particle] += impactDistance / particleSpeed;
            if lifeTime[particle] >= halfLife[particle] then
                status[particle] = AIR;
                particlesDone += 1;
                continue;
            else
                [x; y; z] = [x; y; z];
                continue;
        end
        else
            lifeTime[particle] += impactDistance / particleSpeed;
            if lifeTime[particle] >= halfLife[particle] then
                status[particle] = AIR;
            else
                status[particle] = position;
            end
            particlesDone += 1;
        end
    end
    ReportStatus(status);
end
```

That enables absolute parallelism, although CUDA device cores used for threads execution do not have instruction execution speed comparable to CPU cores. Strength that these
devices bring to the table is gained by using massive work-force made of a large number of independent concurrent threads.

In order to use CUDA resources in the most efficient manner some general guidelines are defined and they can be found in [3] and [10]. Some of these rules were applied for construction of both host and device code for CUDA parallel implementation described with algorithms 2 and 3, respectively. They resulted in implementation details described below.

Each particle is associated to a single CUDA thread. As number of particles exceeds number of available device cores, more than one thread will be executed on each of the device cores. Execution plan like this requires efficient use of CUDA warp mechanisms and thread blocks organization according to the warp size. During process initialization. During execution, for each of the threads running, the number of iterations is defined for each particle associated with a thread. After these iterations (their number defined as chunk) are done, only status information is transferred to the host in order to discover how much work has been done. If possible, we adapt thread number to decreased number of particles and therefore demand for lesser resources. Inside kernel code, before planned chunk of iterations is done, all necessary data for a particle is copied to the local CUDA core registers. After that, all data operations conducted during chunk iterations are performed on local fast registers, avoiding slow global memory operations. Therefore, choice of chunk size is of great influence to overall performance, and is product of compromise between more iteration work to be done, and early discovery of particles that should not be observed any more.

Brownian motion modeling is supported with linear congruent random number generator. It was both appropriate and necessary solution since it is convenient enough in manner of parallel implementation and does not affect implementation accuracy.

4. RESULTS AND DISCUSSION

Accuracy and performance for both serial and parallel CUDA implementation were tested on appropriate platforms. Serial version was executed on Intel Core2Quad Q6600 based server, while parallel versions are executed on same machine powered with GTX560 CUDA device and also on NVIDIA Tesla 2090 devices available from TGCC CURIE supercomputer. Execution included repetitive runs of simulation series. Each series consisted of several simulation runs, each for specific number of particles. In each run, and the execution in whole, we observe the same diffusion chamber with same parameters defining Radon progeny (mean life time, mean path length between collisions).
From all these simulation runs, we obtained data needed for Radon particles behavior analysis, and that data is consisted of:

1) Positions of particles - position in chamber particle achieve at the end of its life time.
2) Time when number of particles in characteristic positions of chamber is modified.
3) Overall execution time.

![Figure 4 - Results of two parallel versions for sample of 4096 particles](image)

The runs are executed until earlier defined number of particles \( \geq 95\% \) achieves any of the characteristic positions inside the diffusion chamber. This approach is used because for a very small number of particles, we need to wait for a significantly longer period of time before they deposit or decay. With this approach we have not compromised accuracy because number of neglected particles is very small, but performance gained is significant, since the execution time is shortened, in certain cases even for an order of magnitude.

In Figure 3, the execution times are compared between serial and two parallel versions (GTX560 and Tesla M2090 devices) for a sample of 1024 particles.

Simulations are executed for chamber with dimensions \( R=0.04m \) and \( H=0.08m \). Mean path length between collisions used in simulations is set to \( 6\times10^{-9}m \).

Simulation runs are executed until 1000 of particles (for a sample of 1024 that is approximately 97.6\%) are being deposited on the chamber surface or decayed in the air. In this scenario, the speedup observed is more than 6 times. Obviously, according to Amdahl's effect [4], the speedup increases when the number of the \(^{222}\text{Rn} \) particles increase. That holds not only compared to serial version, but also in comparing results for parallel devices with different computing capabilities as it is shown in Figure 4.

In order to validate accuracy of the implementation, it is necessary to analyze position in chamber where particles end up. Table 1 contains values representing relative amount of particles ending up in specific chamber wall positions. Comparing these results with those obtained in [1] and [2] showed that CUDA implementation is valid.

Beside obvious speedup of parallel compared to serial version, it is interesting to compare speedup results obtained from parallel variants with and without reconfiguration of engaged CUDA thread blocks during simulation run. If thread number executed in grid is not decreased for particles that already deposited or decayed, significant performance degradation is noted. This is due to unnecessary occupation and sharing of CUDA device cores for threads related to particles which do not move any more. Time spent on analysis and reconfiguration of execution thread blocks between two successive device runs seems justified, because it leads to the better device cores utilization. Figure 5 shows comparison of two parallel versions on the same CUDA device for 2048 particles, one with and the other without execution blocks reconfiguration.

We observe time needed for specific number of particles to achieve their final status and position in the chamber. Speedup obtained in version with reconfiguration is far from not being significant.

5. CONCLUSION

CUDA based simulation implementation improves serial implementation by running each \(^{222}\text{Rn} \) particle behavior simulation in parallel manner. That makes execution time significantly shorter and simulation highly scalable regarding to the number of particles within the model. The number of modeled particles can be increased with both execution configuration change and addition of GPGPU devices. Increased number of particles in faster simulation execution enables more efficient analysis and check of particles behavior in the diffusion chamber. Further research should provide more advanced simulation model and execution implementation.

![Figure 5 - Results for versions with and without blocks reconfiguration](image)

Simulation model needs further adjustments in order to fully adapt to GPGPU development paradigm. That could be accomplished by splitting diffusion chamber space into uniform segments of smaller volume. Using particle behavior data obtained from such small segment, it would be possible to approximate behavior of the particles in the entire chamber. Simulation implementation on CUDA could be made more efficient by introducing new NVIDIA KEPLER architecture devices and/or engaging hybrid parallel techniques such as combining inter-node MPI communication with CUDA.
6. ACKNOWLEDGMENT

We acknowledge that the results of these simulations have been achieved using the PRACE Research Infrastructure resource CURIE based in France at TGCC (Très Grand Centre de calcul du CEA).

REFERENCES


Stanojević Adam is a PhD candidate at the Department of Mathematics and Informatics, Faculty of Science, University of Kragujevac. email: adam.stanojevic@kg.ac.rs

Ivanović Miloš is Assistant Professor at the Department of Mathematics and Informatics, Faculty of Science, University of Kragujevac. email: mivanovic@kg.ac.rs

Stevanović Nenad is Assistant Professor at the Department of Physics, Faculty of Science, University of Kragujevac. email: nstevanovic@kg.ac.rs

Filipović Nenad is a Full Professor at the Department for Applied Mechanics And Automatic Control, Faculty of Engineering, University of Kragujevac. Email: fica@kg.ac.rs
Parallel Algorithms for Statistic Modeling of Dam Behavior

Ivanović, Miloš; Nikolić, Srđan; Stojanović, Boban; Milivojević, Nikola; and Filipović, Nenad

Abstract — It is well known that dams have strong interactions with environmental, hydraulic and other factors, such as air and water temperature, water level, pore pressure, rock deformability, etc. Each of them has significant influence on the structural behavior of the dam. In order to describe and predict structural behavior of dams during long periods of usage, over the last decades a number of deterministic, statistical and hybrid mathematical models have been developed. Statistical models have been applied to dam safety monitoring to find out the contribution of the external loads to the dam displacement. Despite the fact that more advanced methods exist, the Multiple Linear Regression (MLR) method is the most commonly used. Regardless the simplicity of the formulation, the computation of the derived variables with large data series is often computationally very intensive. This paper presents a parallel algorithm for averaging large time series used as input data for MLR. The solution is based on R statistical environment, with two distinct implementations. The first one uses Rmpi package, while the other one uses Multicore package. The speedup comparison between two implementations is given, showing similar performance, close to ideal.

Index Terms — parallel computing, dam, modeling, linear regression, monitoring

1. INTRODUCTION

Dams have strong interactions with environmental, hydraulic and geomechanical factors, such as air and water temperature, water level, pore pressure, rock deformability, etc. each of which influences the structural behavior of the dam [1]. In order to describe and predict structural behavior of dams, over the last decades a number of deterministic, statistical and hybrid mathematical models have been developed. For a long time now, statistical models have been applied to dam safety monitoring to find out the contribution of external loads to dam displacement [2]. A number of statistical models based on multiple linear regression (MLR) and their advanced forms such as hierarchical regression, stepwise multiple regression, robust regression, ridge regression, and partial least squares regression have been shown more or less successful in dam modeling [3]. The advantages of the statistical models consist in simplicity of formulation, execution speed and the availability of any kind of correlation between independent, derived and responses variables. On the opposite, deterministic models require complex differential equations solving, for which closed form solutions may be difficult or impossible to obtain [4].

While deterministic models based on finite differences or finite element methods are known for their high demand for computational resources, statistical models can also be highly computationally expensive, especially when they are based on the large series of measurements. Since all the measurements are performed by various automatic sensors placed on multiple locations within a dam itself or its immediate surroundings, with sampling period of every few minutes, the time series rapidly grow. On the other side, the raw measurements taken from the sensors are not always sufficient to construct high-quality MLR model. For instance, the temperature and precipitation data must be averaged in order to be of any benefit for the model. Immediate values of such variables make no sense on long multi-annual series. In a number of use cases, the delay operator should be performed as well, in order to statistically cover the effect of i.e. slow water transfer through the porous media.

While MLR is usually quite fast to perform on modern computers, deriving averaged and delayed series can be really slow. For instance, in case that time series consist of 5000 rows, averaging for the period of 30 days backwards takes more than a minute on 2.4GHz CPU. The main reason for the poor performance is that the averaging algorithm implies lookup table search for each single value. It is impossible to perform simple search based on index, since a number of rows is missing due to temporary malfunction of the corresponding sensors. In order to solve this issue, we took the path of performance improvement using parallelization techniques. In R statistical programming environment [5] which the entire system is based on, multiple library choices exist, for both explicit and implicit approach to parallelization. The package Rmpi is chosen to represent explicit, while multicore
package is chosen to represent implicit approach. Rmpi package is a universal choice due to the possibility to run on the distributed architecture such as multi-node cluster, but requires more dependencies (i.e. appropriate MPI implementation should be installed).

2. Methods

2.1 Statistical modeling

MLR is a method used to model linear relationship between a dependent variable (predictant) and one or more independent variables (predictors). The general form of MLR can be written as follows:

\[
y_i = \beta_0 + \beta_1 \cdot x_{i1} + \beta_2 \cdot x_{i2} + \cdots + \beta_j \cdot x_{ij} + \cdots + \epsilon_i, \quad i = 1, 2, \ldots, n \tag{1}
\]

where \( y_i \) is a response variable, \( x_{ij} \) are predictors \((j=1,2,\ldots,k)\), \( k \) is the number of significant predictors, the index \( i \) shows the sample number, and \( \epsilon_i \)'s are independent and normally distributed random variables with zero mean and variance \( \sigma^2 \). In majority of the applications of linear regression models, the functional forms of predictors (basis functions) are not clear in advance and are dependent on the nature of the modeled phenomenon \([6]\). Coefficients \( \beta_0, \ldots, \beta_k \) are unknown parameters of the model, estimated for a given set of data using the least square error (SSE):

\[
SSE = \sum_{i=1}^{n} \epsilon_i^2 = \sum_{i=1}^{n} (y_i - b_0 - \sum_{j=1}^{k} b_j \cdot x_{ij})^2, \tag{2}
\]

by taking derivatives of SSE with respect to \( b \)'s and setting them equal to zero. In Eq. (2), \( b \)'s are the estimated values of the model parameters.

Developed system for MLR modeling of the dam behavior is designed to be user friendly. The whole modeling process can be performed from the GUI, as shown in Fig. 1. On the other side, as already mentioned, the GUI part is only a frontend for the background engine written as a set of routines in R programming language. Throughout the entire preprocessing and modeling phases, the R session thread is present in the background, acting as a state machine. Modeling process starts with data input, then the user defines the regressors, and performs MLR algorithm in iterative manner. The application also has a capability to propose which regressors should be omitted in the next iteration, based on each regressor coefficient values and ANOVA \([7], [8]\). Characteristic dialog for defining average and delay regressors is shown in Fig. 2.

Figure 1. The main window of the application for dam behavior modeling

Figure 2. Dialog for setting up the averaged and delayed regressors

As mentioned above, the main objective was to speed-up the procedure of generating derived average and delay regressors during preprocessing phase. These regressors can be described by following equations:

\[
\hat{X}_i = X_{i-d}, i = d, d + 1, \ldots, n \tag{3}
\]

\[
\hat{X}_i = \frac{1}{d} \sum_{j=d+1}^{i} X_j, i = d, d + 1, \ldots, n, \tag{4}
\]

where \( \hat{X}_i \) denotes delayed regressor, \( X_i \) denotes averaged regressor, and \( d \) (in days) represents delaying and averaging period, respectively. As shown in (3) and (4), the parallel algorithm approach is as simple as assigning different tasks to different processors. As long as time needed to perform lookup for each \( X_i \) is constant, the tasks can be distributed in a static manner. There is no need even to consider any dynamic scheduling algorithm such as manager-worker, since the load is well balanced thanks to constant lookup time. Of course, the processors which participate in computation are considered uniform.

2.2 Rmpi implementation

Rmpi package acts as an interface to the MPI (Message Passing Interface), de-facto standard for the distributed memory parallelization (sometimes also called explicit parallelization) \([9]\). The main advantage of Rmpi is that the developer is able to invoke MPI communication routines directly from R code. The package is cross-platform and known to work with different MPI implementations such as MPICH2, OpenMPI, etc.

The implementation of the average/delay algorithm is straightforward. After successful
initialization (including series loading), the root process scatters equal portions of an appropriate column to the rest of the processes, including the root itself. Each process then executes average/delay operations on its own set of data. Finally, MPI gather collective operation is performed in order to prepare data for the disk output.

2.3 Multicore implementation

Multicore package’s purpose is to help developing shared memory parallel applications in R [10]. Recent global presence of multi-core processors made R developers include slightly modified copy of this library into the basic R installation. In shared memory approach, unlike MPI, all threads share the same address space, without any need to transfer input/output data among them. The implementation presented here uses library function mclapply, whose purpose is to assign different loop iterations to different threads. It takes the code for each iteration (can be a lambda expression) and the iteration index as parameters.

2.4 Results and discussion

The parallel average/delay parallel algorithms have been tested thoroughly using real-world data taken from the sensors deployed on the Bocac dam on the river of Vrbas in Republic of Srpska. The data set for precipitation data taken for the benchmark purposes consists of more than 5000 values, taken on daily basis for more than 10 years of monitoring. MLR algorithm and parallel average/delay algorithm are also capable of handling missing and meaningless values. All benchmarks are performed on the workstation equipped with two Intel Xeon E5504 quad-core processors, totaling to 8 CPU cores.

Time needed for varying numbers of participating CPUs are shown in Table 1 for Rmpi and Table 2 for Multicore implementation.

<table>
<thead>
<tr>
<th>Table 1: Computation time of Rmpi implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average(d)</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2: Computation time of Multicore implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average(d)</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

The speedup diagram shown is Fig. 3 shows satisfactory results, closed to ideal speedup. Rmpi and Multicore performance is almost equal. The process initialization overhead, which is an integral part of Rmpi implementation is not significant. This fact promises a good scalability on larger systems such as multi-node clusters.

![Figure 3. Obtained speedup for averaging period of d=30 days](image)

3. Conclusion

In this paper a parallel algorithm for averaging large time series used as input data for MLR is presented. Its main objective was to speed up modeling of dam behavior over long time periods. The solution is based on R statistical environment, with two distinct implementations. The first one uses distributed memory approach, while the other one uses shared memory approach. The performance of the system was benchmarked using Bocac dam test case, where radial displacement of a point inside the dam structure as a function of headwater, precipitations, concrete temperature and time has been modeled. The results turned out to be nearly ideal for both implementations. This is not only useful in speeding up the modeling, but also makes the GUI front-end of the system much more responsive.

The future work should be directed toward development of more robust adaptive parallel dam modeling and monitoring system, which will be able to handle sensor malfunction and other...
irregularities automatically, using various MLR models depending on the type of missing data. In such systems, the demand for parallelization increases rapidly.

ACKNOWLEDGMENT

We acknowledge that the results of these simulations have been achieved using the PRACE Research Infrastructure resource CURIE based in France at TGCC (Très Grand Centre de calcul du CEA). The part of this research is also supported by Ministry of Science in Serbia, Grants III41007 and TR37013, and FP7 ICT-2007-2-5.3 (224297) ARTreat project.

REFERENCES


Miloš Ivanović is Assistant Professor at the Department of Mathematics and Informatics, Faculty of Science, University of Kragujevac, Serbia. email: mivanovic@kg.ac.rs

Srđan Nikolić is Master Student at the Department of Mathematics and Informatics, Faculty of Science, University of Kragujevac, Serbia. email: srdjan.nikolic@fondmt.rs

Boban Stojanović is Assistant Professor at the Department of Mathematics and Informatics, Faculty of Science, University of Kragujevac, Serbia. email: boban.stojanovic@gmail.com

Nikola Milivojević is Lead Researcher at Jaroslav Černi Institute for the Development of Water Resources, Belgrade, Serbia. email: nikola.milivojevic@gmail.com

Nenad Filipović is Full Professor at the Department for Applied Mechanics and Automatic Control, Faculty of Engineering, University of Kragujevac, Serbia. email: fica@kg.ac.rs
Modeling of Selfhealing Materials Using Dissipative Particle Dynamics Method and Parallelization at Architectures of Shared and Distributed Memory

Petrović, D., Obradović, M., Jovanović, A., Jovanović, S., and Filipović, N.

Abstract — Degradation of materials and structures, by influence of corrosion, is one of important issues that lead to depreciation of investment goods. To protect material from corrosion destruction the coating systems are employed on a wide range of engineering structures, from cars to aircrafts. The “self-healing” or “inhibition” are a relatively new terms in material science which means a self-recovery of initial properties of the material after destructive actions of external environment.

The process of corrosion inhibition is simulated by Dissipative Particle Dynamics (DPD) method. Besides the standard repulsive, dissipative and random forces, used in the DPD method, it was introduced the additional polymerization force between two particles within nanocontainer. Parallelization of DPD software code is also implemented. With this technology we can optimize the number of nanocontainers and inhibitors during experiments and real protection procedures in the industry. We also can speed up a calculation process using parallel version of software.

Index Terms — nanocoating, parallelization, self-healing material process

1. INTRODUCTION

Corrosion degradation of materials and structures is one of important issues that lead to depreciation of investment goods. Two main approaches, an active and a passive one, are currently used for corrosion protection. The passive corrosion protection is achieved by deposition of a barrier layer preventing contact of the material with the corrosive environment [1], [2].

Small size defects can appear on a material surface. Such defects may have a substantial effect on the mechanical properties of material. To protect this material failure the coating systems are employed on a wide range of engineering structures, from cars to aircrafts, from chemical factories to household equipment. The “self-healing” or “inhibition” are a relatively new terms in material science which means a self-recovery of initial properties of the material after destructive actions of external environment.

It is an urgent demand for industrial applications to initiate development of an active healing mechanism for polymer coatings and adhesives [3], [4].

2. DPD MODEL

The coating layer with nanoscopic noach can be modeled using molecular dynamics [5]. Another approach to this problem is a mesoscoping modeling using the DPD method [6], [7]. Motion of each DPD particle (further called “particle”) is described by the following Newton law equation:

\[ m_i \ddot{r}_i = \sum_j \left( F_{ij}^C + F_{ij}^D + F_{ij}^R \right) + F_{ij}^{ext} \] (1)

where \( m_i \) is the mass of particle \( i \); \( \ddot{r}_i \) is the particle acceleration as the time derivative of velocity; \( F_{ij}^C \), \( F_{ij}^D \), and \( F_{ij}^R \) are the conservative (repulsive), dissipative and random (Brownian) interaction forces, that particle \( j \) exerts on particle \( i \); respectively, provided that particle \( j \) is within the radius of influence \( r_c \) of particle \( i \); and \( F_{ij}^{ext} \) is the external force exerted on particle \( i \), which usually represents gradient of pressure or gravity force as a driving force for the fluid domain [8]. The total interaction force \( F_{ij} \) (Fig. 1) between the two particles is

\[ F_{ij} = F_{ij}^C + F_{ij}^D + F_{ij}^R \] (2)

Figure 1. Interaction forces in the DPD method
The component forces can be expressed as [9]

\[
F^C_{ij} = \alpha_{ij} \left( 1 - \frac{r_{ij}}{R_{ij}} \right) r_{ij}^0
\]

\[
F^D_{ij} = -\gamma w_D (v_{ij} \cdot \hat{r}_{ij}) r_{ij}^0
\]

\[
F^R_{ij} = \sigma w_R \xi_{ij} r_{ij}^0
\]

(3)

In equation (3), \( \alpha_{ij} \) is the maximum repulsion force per unit mass, \( r_{ij} \) is the distance between particles \( i \) and \( j \), \( r_{ij}^0 \) is the unit vector pointing in direction from \( j \) to \( i \), \( \gamma \) stands for the friction coefficient, and \( \sigma \) is the amplitude of the random force. Also, \( w_D \) and \( w_R \) are the weight functions for dissipative and random forces, dependent on the distance \( r \) from the particle \( i \); and \( \xi_{ij} \) is a random number with zero mean and unit variance. The interaction force is equal to zero outside the domain of influence, \( r_c \) (cut radius), hence \( F^R_{ij} = 0 \) for \( r > r_c \).

Further, in order that a DPD fluid system possess a Gibbs–Boltzmann equilibrium state, the following relation between the amplitudes of the weight functions of dissipative and random forces, \( w_D \) and \( w_R \), must hold:

\[
w_D = w_R^2
\]

Also the amplitude of the random force \( \sigma \) is related to the absolute temperature \( T \),

\[
\sigma = (2k_B T \gamma)^{1/2}
\]

where \( k_B \) is the Boltzmann constant. The weight functions can be expressed in a form given as [8]

\[
w_D = \left( 1 - \frac{r_{ij}}{r_c} \right)^2, \quad w_R = 1 - \frac{r_{ij}}{r_c}
\]

(6)

The particles used in this study represent both inhibition agents and surrounding coating material with different material characteristics. This was achieved by taking into account different repulsion force coefficient \( \alpha_{ij} \). The additional interaction forces between particles of inhibition agents, which are placed in the primer layer and metal substrate particles, are added similarly as it was done in a model of thrombosis in [9]. These attractive forces are expressed as

\[
F_a = k_a \left( 1 - \frac{L_{af}}{L_{af}^{\max}} \right)
\]

(7)

where \( L_{af} \) is the distance of the inhibition particle from the substrate, \( k_a \) is the effective spring constant, and \( L_{af}^{\max} \) is the maximum length of inhibition particle attractive domain.

3. Simulation Model

We created here two models of protection. Self–healing model for smaller damage of material and inhibition model with nanocontainers which are placed in the primer layer. The initial process of nanocontainer breaking starts at the position where a crack occurred. The nanocontainer membrane is approximated by one porous layer of particles and particles inside the nanocontainers represent healing agents – inhibitors. We consider that nanocontainers are fixed in the coating layer (pretreatment or primer layer) so the membrane particles are fixed in the DPD space domain. Self–healing model is shown on Fig. 2 and Fig. 3 represent inhibition model.

![Figure 2. Initial position of particles at self-healing model](image)

Nanocontainers release the “self–healing” agent particles which are filling the space inside a crack in order to bond it and to protect it from further propagation. Additional spring force (equation (7)), acts between particles themselves and also with particles that represent the damaged metal surface and this force is present only at inhibition model.

![Figure 3. Initial position of particles at inhibition model](image)

4. Parallelization of DPD Software

Here we had two different ways of parallelization. One approach is using architecture of shared memory and another is using architecture of distributed memory.

At the shared model architecture we used the threads model where each process can have multiple, competitive paths of execution. Each thread has its own local data, but also has access to the resources of the central processor. Implementation of this model that we used here is OpenMP (Open Multi-Processing) [10].

Architecture of distributed memory has implementation using message model. At this model processes can be on same or on different computer. Processes communicate only exchanging messages and every message that one process send must have a process that will receive that message. In our implementation we used MPI standard (Message Passing Interface).
5. RESULTS AND DISCUSSION

5.1 Results of DPD simulation

Self-healing model contained 9600 particles where only white particles (water) have external force. Density of nanocontainers in this model was 12% (Fig. 4).

![Figure 4](image)

Figure 4. Position of particles on self-healing model at the end of simulation

Percent of healing agents filling crack is shown on Fig. 5.

![Figure 5](image)

Figure 5. Percent of healing agents inside crack at self-healing model

Model of inhibition that we created has 28320 particles and nanocontainers are randomly distributed in the model. Density of nanocontainers in primer layer is 15%. Results of corrosion inhibition of metal substrate at the end of simulation is shown on Fig. 6.

![Figure 6](image)

Figure 6. Position of particles at the end of simulation

At the Fig. 7 is shown percent of inhibitors on metal substrate during time. We have more than 100% protection because healing agents created more than one layer of particles.

![Figure 7](image)

Figure 7. Percent of inhibitors on metal substrate during time

5.2 Results of parallelization

Testing of example is done on the server computer that is running Scientific Linux 5.5 (Boron) operating system for both architectures mentioned here. Characteristics of computer was: Intel(R) Xeon(R) Processor E5504 (4M Cache, 2.00 GHz, 4.80 GT/s Intel(R) QPI); number of cores – 4; number of threads – 4; 8GB of RAM memory.

Duration of analysis, at self-healing model, depending on number of processes is shown on Fig. 8 for both parallelization techniques.

![Figure 8](image)

Figure 8. Time of analysis at self-healing model

Big difference between MPI and OpenMP architecture is in fact that we used static scheduling for each particle where one thread calculates values of interests for one particle and one loop iteration.

On Fig. 9 and Fig. 10 are shown results that we get using parallel version of software for inhibition model of corrosion protection.
6. CONCLUSION

In this study we used DPD computer modeling methods to investigate coating by the healing agents of substrates that contain nanoscale defects. The results that we get using our models have significant overlaps with experimental results. For both investigated models it was sufficient to have 15%-20% inhibitors (healing agents) to protect the damaged material.

We also developed parallel version of software to speed up the process of calculating. With parallel DPD software we get satisfying results as concerns of speed up of calculation time but there is enough place for algorithm optimization in both shared and distributed memory architectures.

ACKNOWLEDGMENT

This research was supported by Ministry of Science in Serbia, ON174028, III41007 and FP7 MUST project NMP3-LA-2008-214261.

REFERENCES

Scientific Applications in Cloud Computing

Ogrizović, Dario; Car, Zlatan; and Kovačić, Božidar

Abstract — Scientific applications over the years have been executed on traditional high performance computing (HPC) systems like supercomputers or clusters and high throughput computing (HTC) systems like Grids. With large amounts of computing resources traditional HPC has been used by many organizations to help resolve a variety of problems. Although these systems were usually designed to address a specific problem, more and more SMEs and even university departments started to take advantage of general purpose HPC systems. With Cloud Computing, as new emerging technology, scientists, engineers, system administrators and developers have been considering HPC Cloud environment in order to exploit of what Cloud Computing has to offer them. We will present the economic and technical benefits of running scientific applications in Cloud Computing environment and some key challenges especially for communication intensive tightly coupled scientific applications.

Index Terms — HPC applications, HPC Cloud, HPC Cloud benefits, HPC Cloud challenges, tightly coupled scientific applications, loosely coupled scientific applications

1. INTRODUCTION

With popularity of Cloud Computing running complex scientific applications is more accessible to the research community by accessing on-demand compute resources in minutes instead of spending waiting times for their compute jobs in queues, experiencing peak demand bottlenecks. Cloud Computing offers great potential for scientific applications, but Clouds have been designed for running business and web applications, whose resource requirements are different from communication intensive tightly coupled scientific applications which typically require low latency and high bandwidth interconnections and parallel file systems to achieve best performance. Most commercial Clouds use commodity networking and storage devices which are suitable to effectively host loosely coupled scientific applications which frequently require large amounts of computation with modest data requirements and infrequent communication among tasks. Several studies have shown that Cloud Computing is viable platform for running loosely coupled scientific applications and workflow applications composed of loosely coupled parallel applications consisting of a set of computational tasks linked via data and control dependencies [1-8]. New capabilities, challenges and performance behavior have not been conclusively answered and need to be addressed for each type of scientific application in Cloud Computing environment.

The rest of the paper is organized as follows. Section 2 presents Cloud Computing paradigm, deployment models relevant to scientific applications, and HPC Cloud solutions. Section 3 proposes scientific computing applications. In Section 4, we present Cloud Computing benefits and challenges for scientific applications. Section 5 discusses future trends of running scientific applications in Cloud Computing environment. We conclude the paper with a summary in Section 6.

2. CLOUD DEPLOYMENT MODELS

Cloud Computing has been defined as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction” [9]. Clouds are classified in three layers or service models: a) Infrastructure as a Service (IaaS), b) Platform as a Service (PaaS) and c) Software as a Service (SaaS). In the first layer, physical resources are delivered as a service, usually through machine virtualization. This layer is dominant for running HPC Cloud environment. In the second layer, a software development platform is delivered as a service to deploy and maintain applications in an integrated environment. In the third layer, a software application is delivered as a service and instead of purchasing license to be installed on premise; users subscribe to use the service of a specific application.

User willing to deploy scientific applications, based on their particular requirements, has several options: either to run it entirely in a private or public Cloud, or to combine these two approaches in a hybrid Cloud model; two or more private Clouds could also interact for common goals and thereby form a federated Cloud.

Tomic et al. [10] defined three tiers based on HPC Cloud solutions available on the market.
First tier HPC Cloud solutions are complete solutions from Hewlett-Packard [11], IBM [12], Microsoft [13], and VCE [14], for private, public and hybrid HPC Clouds with a support for both physical and virtual resources. Second tier HPC Cloud solutions are complete solutions as well, but either not supporting all classes of Clouds, or not supporting both physical and virtual resources. Second tier includes HPC Cloud solutions from Amazon [15], Google [16], SGI [17], Rackspace [18], GoGrid [19], Penguin [20], and R Systems [21], Nimix [22], and Univa [23]. Third tier HPC Cloud solutions are components only, but of the key importance for first and second tier HPC Cloud solutions. These HPC Cloud solutions are from Adaptive Computing [24], IBM [25], VMWARE [26], and Bright Computing [27].

3. Scientific Computing Applications

Scientific applications in Cloud Computing environment, based on their resource requirements, can be classified into tightly and loosely coupled scientific applications.

3.1 Tightly coupled applications

Supercomputers and clusters have traditionally been executing tightly coupled applications within a particular machine over low latency interconnects, which makes it possible to share data very rapidly between a large numbers of processors working on the same problem, and with message passing interface (MPI) [28] to achieve inter process communication. These systems are typically optimized to maximize the number of operations per seconds. Tightly coupled applications are common classes of scientific HPC applications which require low latency network of high bandwidth because frequent communication is necessary. Examples include domain decomposition solvers, linear algebra, FFTs, N-body systems, etc.

3.2 Loosely coupled applications

Grids have been the preferred platform for more loosely coupled applications that are managed and executed through workflow systems. In contrast to HPC (tightly coupled applications), the loosely coupled applications are known to make up high throughput computing (HTC). HTC is a computing paradigm that focuses on the efficient execution of a large number of loosely-coupled tasks. Tasks can be executed on clusters or using grid technologies because low parallel communication requirements. HTC systems are optimized to maximize the throughput over a long period of time (jobs per month or year) [29]. These applications are also called embarrassingly or pleasingly parallel applications which can be parallelized with minimal effort and with the MapReduce [30] based frameworks are good candidates for running in Cloud Computing environment with commodity interconnects. There are many scientific applications that fall in to this category. Few examples would be Monte Carlo simulations as they involve large numbers of compute cycles with a relatively small data set, BLAST searches, many image processing applications such as ray tracing, parametric studies, etc.

4. Cloud Computing Benefits and Challenges for Scientific Applications

4.1. Cloud Computing benefits

The Cloud Computing offers scientific applications a range of benefits, including cost advantages for some type of scientific applications, ability to rapidly provision new clusters and instantly access them, elasticity for instant adding and removing resources, configurability with root access, and sharing and collaboration of data, results, methods, and resources between partners.

Cost advantages

When evaluating options for Cloud based scientific applications, costs are often a major consideration. With Cloud Computing users can eliminate the cost and complexity of procuring, configuring, operating, managing and maintaining their own cluster infrastructure with low, pay-per-use pricing for actual resource usage.

Gupta et al. [31] indicated that small and medium scale scientific applications with modest communication and data requirements could be cost effective on Cloud resources. Large scale scientific tightly coupled applications running on virtualized Clouds using commodity networks can be more cost effective on dedicated optimized clusters than Cloud showed Magellan report [32]. Works such as [33, 34] have studied the cost or benefit of using Cloud technologies versus the cost of owning a datacenter infrastructure. Paper [35] performs a detailed comparison between physical and virtual HPC clusters from the point of view of the TCO, considering energetic, management, and infrastructural issues.

With advantage of spot instances users can optimize and keep Cloud Computing cost as low as possible. Cloud providers like Amazon start to establish spot markets on which they sell excess capacity even for scientific computing [36]. Spot Instances enable users to bid for unused Amazon capacity. Instances are charged the Spot Price, which is set by Amazon and fluctuates periodically depending on the supply of and demand for Spot Instance capacity. To use Spot Instances, users place a Spot Instance request, specifying the instance type, the region desired, the number of Spot Instances they want to run, and the maximum price they are willing to pay per instance hour. However, there is no guarantee for continuous operation because a virtual machine is stopped if the market price exceeds the maximum bid.

Beside Amazon spot instances there are
specific market services like SpotCloud. SpotCloud [37] is an easy to use, structured Cloud capacity marketplace where service providers can sell their excess computing capacity to a wide array of buyers and resellers. SpotCloud has implemented an environment to buy and sell computing capacity globally based on price, location, and quality on a fast and secure platform. SpotCloud platform provides an easy method to maximize revenue for Cloud providers, datacenters, etc. for their unused capacity. For users it provides an easy way to discover and access targeted premium or commodity compute capacity. Users can choose the most efficient Cloud Computing resources suited for their application and budget.

**Instant access**

Ability to rapidly provision new clusters and access compute resources, and configure them, in minutes instead of spending hours or days waiting in queues, and in case of the initial procurement of a new cluster waiting for months. Thereby, time to get the job done improves even if the performance of a Cloud Computing cluster is lower than that of a traditional dedicated cluster. These resources can be released when they are no more needed and they are offered within the context of a Service Level Agreement (SLA), which ensure the Quality of Service (QoS). To achieve maximum efficiency and scalability users can use resources by using simple APIs or management tools and automate workflows.

**Elasticity**

In traditional cluster infrastructure it is difficult to adequately size a system so resources will be idle or inadequate for application requirements. With Cloud Computing elasticity users can instantly add and remove resources to meet their application requirements. Similar option is Cloud bursting where private Cloud users could burst into a public Cloud to get more resources or during peak load requirements of local dedicated cluster to accelerate results. Bright Computing [27] has introduced a Cloud bursting component to its Bright Cluster Manager platform to easily create new clusters in the Cloud, or add Cloud based resources on-the-fly to existing private infrastructure running HPC computations.

**Configurability**

On Clouds in contrast to traditional dedicated clusters users have root access (IaaS) and they can customize their cluster instances with specific libraries, compilers, applications, running the operating system of choice, even different parallel file systems and disk configurations according to the needs of scientific application, etc. or can use pre-built environment.

**Sharing and collaboration**

In Cloud users can create a common space to share data, results, and methods and even to extend HPC resources to community partners without their own HPC infrastructure. Amazon provides a centralized repository of public data sets that can be seamlessly integrated into Cloud based applications. AWS is hosting the public data sets at no charge for the community, users pay only for the compute and storage they use for their own applications. [38]

### 4.2. Cloud Computing challenges

Despite the many benefits of the Cloud Computing, broader use of scientific applications in Cloud Computing also presents some key challenges to overcome. Primary among them is the lack of high bandwidth, low latency interconnections for running tightly coupled scientific applications, security issues which prevent wider Cloud Computing adoption, operating systems and network noise with its roots in traditional HPC environment and have hardly been assessed in Cloud Computing studies, data transfer as main bottleneck for large datasets, procuring Cloud services, concerns about vendor lock-in, portability and interoperability, knowledge and expertise to implement Cloud Computing solutions, and traditional licensing models which does not adapt well to a Cloud Computing.

**Interconnection network**

There have been several studies of running scientific applications in Cloud Computing, whereas many early studies, since 2008, [39-43] have shown that on commercial public Clouds lack of a high bandwidth, low latency interconnections can limit performance, particularly for communication intensive applications. As an offer to the HPC Cloud requirements Amazon introduced Cluster Compute Quadruple Extra Large instances (CC1) and Cluster Compute Eight Extra Large instances (CC2). In contrast to regular EC2 instances, cluster compute VMs are assigned to dedicated nodes with a full-bisection high bandwidth 10Gbit/s Ethernet, which is the differential characteristic of these resources, with performance comparable to that of HPC clusters [44]. Zhai et al. [45] showed a significant performance increase of MPI applications on CC1 instances compared to previous evaluations on EC2 instances but noted that 10 Gigabit Ethernet remains the chief problem in scaling MPI programs. Ramakrishnan et al. [46] stated that virtualized network is the main performance bottleneck, including 10 Gigabit Ethernet. Interconnection network issue can be resolved by providing virtual machines with direct access to a high bandwidth, low latency interconnect like, common cluster interconnect, InfiniBand [47]. Mehrrotra et al. [48] did different benchmarks and applications tests on dedicated cluster, Amazon Cluster Compute (CC1), and SGI Cyclone [17] and concluded that SGI Cyclone performance is close to dedicated cluster with overhead of initially booting nodes with the requested image and that
Amazon CC1 performance lags dedicated cluster for HPC applications due to network technology and virtualization overhead. Mauch et al. [49] presented an approach to use high speed cluster interconnects like InfiniBand in a high performance Cloud Computing environment. Exposito et al. [50] analyzed the main performance bottlenecks in HPC application scalability on the Amazon EC2 Cluster Compute platform and proposed direct access of the virtual machine to the network for reducing the impact of the virtualization overhead in the scalability of communication - intensive HPC applications and the combination of message - passing with multithreading as the most scalable and cost - effective option.

There are developments that will include functionality to meet the needs of the HPC Cloud, such as low latency remote direct memory access (RDMA) over Converged Ethernet (RoCE) [51], and various network virtualization technologies, such as VMware’s Virtual Extensible LAN (VXLAN) or VMware’s network and security virtualization platform (NSX) [52]. Single Root I/O Virtualization (SRIOV) [53] that will allow multiple VMs to share PCI device, and hardware-assisted virtualization that will accelerate context switching, speed up memory address translation, and enable guest VMs direct access to I/O devices which will allow that some virtualized environment can achieve near native CPU and I/O performance [54]. Vienne et al. [55] did a comprehensive performance evaluation of four possible modes of communication InfiniBand FDR / RoCE 40 GigE over InfiniBand QDR and RoCE 10 GigE interconnects with conclusion that InfiniBand FDR interconnect gives the best performance.

Generally, Cloud Computing network infrastructure has improved over the years, but it is not a replacement for traditional dedicated, InfiniBand connected clusters yet.

Security

Centralization of security, redundancy and high availability, data and process segmentation are Cloud Computing security benefits, but a number of security issues are also introduced. [56-58]. One of the biggest fears is that all data and applications, which represent intellectual property, stored on the Cloud are more easily to obtain, use or destroy illegally by third persons, even by the Cloud providers. Several Cloud providers have started to offer virtual private Clouds that are hosted by the provider but fully under control of a customer [18-23, 59]. Some providers offer secure physical dedicated “nothing shared” environment (compute, network, storage), prepared to support performance and regulatory compliance needs of customers to be exclusively used by their critical applications [60]. The private Cloud is a good option for those scientific applications with prohibitive security concerns.

Despite significant Cloud security research, there is a still lack of convincing, clear and transparent model of security and trust for better Cloud Computing adoption.

Noise

Parallel applications running on large number of processors suffer degradation in performance caused by unexpected delays in the processing time due to operating system noise. Operating system noise is interference caused by daemon processes and asynchronous kernel events such as different interrupt types, timers, process preemption, paging activity, loggers, etc. In a typical HPC environment, each computing node executes user’s application and its own operating system which are competing for the node computing resources, and because this activity is unsynchronized on large number of nodes there is always some node being delayed influencing overall application performance.

De et al. [61] presented the implementation of a tool to identify sources of operating system noise in the Linux operating system. They reported that timer interrupt is causing 63% delays, and the rest came from different operating system daemons and other hardware interrupts.

Ferreira et al. [62] described the effect of different kinds of noise on application performance at scale.

Large scale HPC systems (supercomputers), such as Blue Gene and Cray XT series, use a specialized lightweight operating systems and microkernels to decrease the noise, other HPC systems use techniques as OS and hardware tuning, synchronization of noise and commodity operating system enhancements. In virtual environment, noise can be decreased with use of Palacios [63] embeddable virtual machine monitor (hypervisor) which provides a lightweight environment (no kernel modifications) [64]. Palacios selectively takes over resources (CPUs, memory, devices) and manage them internally. Virtualization of a large scale supercomputer using modified Palacios hypervisor together with Kitten HPC OS [65] was studied by Lange et al. [66]. Kudryavtsev et al. [67] explored KVM/QEMU [68] and Palacios hypervisors. Hypervisors test results, with NUMA emulation enabled, were similar, with KVM providing more stable and predictable results and Palacios being much better on fine-grained tests. With amount of noise generated, Kitten behaved better than Linux resulting in better scaling for tests running inside Palacios’ virtual machines. Another type of noise in virtual environment is multi-tenancy noise [69].

A related problem to operating system noise is network noise, which comes from shared use of an interconnection network by parallel processes. Hoeffer et al. [70] conducted a series of simulations and experiments, using an own developed benchmark, to examine the influence of network noise on parallel applications. Their results showed decrease in the communication performance when running large-scale
applications and that influence of network noise grows with the system size.

Moving data to, from and between Clouds

Moving data to (local to Cloud environment), from (Cloud to local environment) and between remote Cloud infrastructures is another HPC Cloud challenge which can be costly and very slow because it requires sending data over the WAN, with high bandwidth costs and network latency, especially when transferring large amounts of data. To reduce the time to move large data between datacenters users and providers are upgrading network connectivity and using high-speed data transport acceleration like Aspera [71], Signiant [72], FileCatalyst [73], CloudOpt [74], etc. They use a combination of UDP (User Datagram Protocol) and custom written application layer software that compensates for network latency and loss in IP networks thus allowing large data to be transmitted to, from and between Clouds more quickly.

Another way of moving data to and from the Cloud is via courier services. Amazon Web Services Import/Export service [75] uses portable storage devices for transport and high-speed internal network to transfer data directly onto and off of Amazon Web Services storage. To decide if it is financially viable to ship data via courier services or upload over the WAN Amazon provides an online calculator [76]. Number of terabytes, devices, average files size, etc. can be entered to find out how much the data transfer would cost. Downside of this approach is that portable storage devices can get delayed, damaged or lost.

Procurement

Procuring services on a pay-as-you-go model is still evolving from the traditional computing centers approach of grants and quotas. In Cloud scientists lease resources and outsource their operation for a short time and pay small amounts of money. Effective account management and procurement options needs to be made in order to use Cloud services with specific quantities and fluctuating costs.

Cloud vendor lock-in, portability and interoperability

Vendor lock-in is a situation in which a customer using a product or service cannot easily transit to another vendor as the result of proprietary technologies that are incompatible with those of competitors. The fear of vendor lock-in is often on the top ten list of Cloud concerns and a major impediment to Cloud service adoption according to a survey [77]. Adapting standards like DMTF’s Open Virtualization Format (OVF 2.0) [78], which is designed to address the portability and deployment of virtual appliances, will allow easy translation between the proprietary virtual machine formats. Satzger et al. [79] present meta Cloud that could solve the vendor lock-in problems and promise transparent use of Cloud Computing services. But no matter what kind of software environment scientists are using, proprietary or FLOSS (Free/Libre and Open Source Software), there are always some kinds of lock-in to the specific technology, though it is easier to replicate FLOSS environment.

Preserving and ensuring data and service portability and interoperability is still challenging task in Cloud Computing because of a lack of thoroughly defined and adopted standards between Cloud vendors.

Acquiring knowledge and expertise

Research teams may not have the necessary tools or resources, such as expertise among staff with the Cloud Computing skills, to implement Cloud solutions. Delivering Cloud services without direct knowledge of the technologies and teaching staff an entirely new set of processes and tools has been a challenge. As a result, some research teams were reluctant to move to a Cloud environment. Proper Cloud Computing training and guidance are necessary to ensure that research teams harness the power of Cloud Computing and remain competitive.

Because of lack of the experts lots of vendors like Cisco, IBM, and Microsoft launched training courses for Cloud Computing, but promotion of non-vendor specific certifications is essential to secure open standards and platforms. Rackspace launched the Open Cloud Academy [80], a training program to teach skills in various open source areas, such as OpenStack, Hadoop and Linux, with a specific focus on Cloud Computing.

Licensing

Traditional licensing models from ISVs (independent software vendors) does not adapt well to a Cloud Computing so ISVs have started to create new and adjust their licensing models to Cloud Computing.

5. Future Trends

Compared to traditional HPC systems HPC scientific application running on Cloud are rare and are mainly present at public Clouds. With proliferation of a high bandwidth, low latency interconnections, which are the keys to scaling performance for HPC applications, Cloud Computing will provide a completely satisfactory HPC environment. Development and deployment of data security, data encryption technology and maturing of Cloud standards will continue to improve Cloud adoption. Adding more network bandwidth and data transport acceleration, by users and Cloud vendors, to reduce the time for transferring and storing large data sets between compute resources will be another future trend. Also wide collaboration between Cloud providers, HPC application vendors and HPC community is needed to accelerate HPC Cloud. Good example is the UberCloud HPC Experiment [81] which is open to the entire community to identify, test and
document potential solutions to the known roadblocks in HPC as a service. HPC private Cloud solutions will be dominant for organizations needed a more collaborative work on the same datasets, and HPC public solutions as a worldwide working environment for a smaller research entities not willing or be able to invest in a HPC clusters. Another aspect of future HPC Cloud would be GPU virtualization on Cloud Computing environment without increased virtualization overhead as described in [82]. Finally, science community will make a future trend how computational science is performed because is no longer limited by fixed size compute and data capacity. Cloud Computing is changing constantly, Napper et al. [41] posed the question “Can cloud computing reach the top500?” with answer “No” and one year later Amazon was ranked 233rd in the Top 500 list [83]. Another year later, in the November 2011 Amazon virtual HPC moved up to 42nd place in the Top 500 list [84]. Present and future HPC Cloud Computing are not the same.

6. CONCLUSION

Cloud Computing is a viable environment for low and medium scale scientific applications that are loosely coupled which can scale even on commodity interconnects common to most of the public Clouds. But for broad adoption of large scale, communication intensive, tightly coupled scientific applications a few major challenges needs to be addressed. In order to more effectively engage and serve scientific applications Cloud Computing need to adopt some features of dedicated HPC systems like high bandwidth, low latency interconnections with properly virtualized interfaces, noise-free operating systems, and properly address security issues, then it will have the potential to grow and eventually go from perspective trend to mainstream in the near future.

REFERENCES


[78] “DMTF’s Open Virtualization Format (OVF)”, http://dmtf.org/standards/ovf
Welcome to IPSI BgD Conferences and Journals!

http://www.internetconferences.net

http://www.internetjournals.net

CIP – Katalogizacija u publikaciji
Narodna biblioteka Srbije, Beograd
ISSN 1820 – 4511
The IPSI BGD Transactions on Internet Research
COBISS.SR - ID 119128844