Journal of the Serbian Society for Computational Mechanics / Special Edition / Vol. 10 / No. 1, 2016 / pp. 50-70

(UDC: 004.272/.275)

High Performance Computing in Multi-scale Modeling, Graph Science and Meta-heuristic Optimization

M. Ivanović[1*], B. Stojanović[1], V. Simić[1], A. Kaplarević Mališić[1], V. Ranković[2], B. Furtula[1], S. Mijailovich[3]

[1] Faculty of Science, University of Kragujevac, 12 Radoja Domanovića Street, Kragujevac, Serbia
e-mail: mivanovic@kg.ac.rs
[2] Faculty of Economics, University of Kragujevac, 3 Djure Pucara Starog Street, Kragujevac, Serbia
[3] Department of Chemistry and Chemical Biology, College of Science, Northeastern University, Boston, MA, USA
**corresponding author*

Abstract

One of the main activities within the Group for Scientific Computing at the Faculty of Science are methods for efficiently utilizing real parallel architectures, typically clusters of SMP nodes, shared-memory systems, and GPUs. Focus is on the design, development and implementation of parallel algorithms and data structures for fundamental scientific and engineering problems. Message Passing Interface (MPI) is an important paradigm that still poses interesting design and implementation problems, especially combined with other programming models, like CUDA. In addition to standard HPC (High Performance Computing) technology stack, the Group also utilizes computing stacks like Hadoop and Spark. In this paper we present a short review of the recent research of the Group, focused on large-scale applications in various research fields with references to original articles. The first part considers multi-scale muscle modeling in mixed MPI-CUDA environment. In our approach, a finite element macro model is coupled with the microscopic Huxley kinetics model. The original approach in scheduling tasks within multi-scale simulation ensures good load balance, leading to speed-up of over two orders of magnitude and high scalability. The second part considers application of HPC in graph science for the task of establishing the basic structural features of the minimum-ABC index trees. In order to analyze large amounts of data (all trees of certain order) we utilize grid computing services like storage and computing in order to reduce analysis time up to three orders of magnitude. The last part presents WoBinGO framework for solving optimization problems on HPC resources. It overcomes the shortcomings of earlier static pilot-job frameworks by providing elastic resource provisioning using adaptive allocation of jobs with limited lifetime. The obtained results show that despite WoBinGO's adaptive and frugal allocation of computing resources, it provides significant speed-up when dealing with problems with computationally expensive evaluations, as found in hydro-informatics and market risk management.

Keywords: High Performance Computing, big data, multi-scale, genetic algorithms, hydro-informatics, risk management

1. Introduction

High Performance Computing (HPC) is an indispensable tool for almost every research that includes numerical simulations of the real world problems. One of the main activities within the Group for Scientific Computing at the Faculty of Science are methods for efficiently utilizing real parallel architectures, typically clusters of SMP nodes, shared-memory systems, and GPUs. The Group deals with designing and implementing parallel algorithms and data structures aimed at solving fundamental scientific and engineering problems. Standard and robust Message Passing Interface (MPI) still presents an important paradigm, especially combined with other programming models like CUDA and OpenCL. In addition to standard HPC technology stack, the Group also utilize computing stacks like Hadoop and Spark, arisen in so called big data arena. On the other side, in case that flexibility and adaptivity are major requests, self-developed frameworks come as a natural solution. Having in mind the immense computing power offered by the computing grids, we devote proper attention to this area.

In this paper we present a short review of the recent research of the Group. The second section considers multi-scale muscle modeling in mixed MPI-CUDA environment with original load balancing approach. Section 3 considers application of HPC in graph science for the task of establishing the basic structural features of the minimum-ABC index trees. Section 4 represents framework for solving large scale optimization problems using HPC resources (including grids), together with case studies with computationally expensive evaluations.

2. Multi-scale muscle modeling in mixed MPI-CUDA environment

Investigating musculoskeletal disorders, as well as characterization and prognostication of neuromuscular diseases require deep understanding of structural and functional properties of muscles. The mechanical behavior of muscles is derived from the behavior of many individual components, such as cell membrane electrical conductivity and action potential, calcium dynamics, chemical reaction kinetics, and the actomyosin cycle, working together across spatial and temporal scales, following Hill (1938) and Zajac (1989). *In silico* analysis of detailed muscle models enable less expensive and time consuming hypothesis testing and evaluation, therefore providing valuable tool in the research activities. Most biophysical models evolve from the hypothesized cross-bridge kinetics concepts originally formulated by A.F. Huxley (1957). In Huxley-type models, following Razumova (1999), and Fernandez (2005), state transitions between myosin and actin states define force generation and relative velocity between sliding filaments. Mijailovich et al. (1996) reformulated A. F. Huxley's sliding filament theory by combining the partial differential equations (PDEs) approach to calculate the traction between actin and myosin filaments and the finite element method (FEM) to assess the deformation of extensible actin and myosin filaments. Simulations of these kinetics processes, in a context of whole muscle models are tremendously computationally intensive and require simplifications of geometry, composition, and activation Röhle et al. (2012). Even with these simplifications, whole muscle models are still computationally demanding. The most widely used phenomenological model, the Hill model, takes into account only the relationship between active stress and strain rate, and its use is therefore limited to isometric and steady state contractions.

In the last decade, significant efforts have been made in developing multi-scale muscle models. This enabled computation of the output force and stiffness by including description of the force-length relationship and the influence of the velocity on cross-bridge breakage at the microscopic scale, Heidlauf et al. (2013). Regarding the fact that multi-scale models are more informative and realistic, their practical implementation and usage in real-world applications is limited by their requirements for computational power, Röhle et al. (2012). Such model simulations are computationally demanding, also requiring significant memory consumption and use of HPC environments. Modern HPC environments are typically heterogeneous, including a variety of multiprocessing architectures such as fast multi-core processors, general purpose graphic processing units (GPUs) and clusters of highly interconnected CPUs. Efficient harnessing of available architectures requires fine-grained levels of parallelism in order to overcome differences in bandwidth and memory capacity available to each processing unit and adapt scheduling policies to processing speeds. To address this opportunity, we propose a methodology for distributed computation of multi-scale muscle models in heterogeneous computing environments, which can employ an arbitrary number of CPU and GPU units. We present extensible and scalable software solution, named *Mexie*, which is built on top of MPI and Compute Unified Device Architecture (CUDA) programming models and provides distributed and parallel execution of multi-scale muscle model simulations. The adapted algorithm for coupling FE macro model with Huxley micro models enables distributed calculations in a hybrid MPI-CUDA environment. Finite element calculations of the continuum macroscopic model run strictly on the CPUs, while Huxley micro models associated with the integration points execute on both CPUs and GPUs.

2.1 Two scale muscle model

From a mechanical point of view, a muscle can be considered as a mechanical system. The most common method for solving complex materially and/or geometrically nonlinear structural problems is the finite element method. In an incremental-iterative scheme, equilibrium configuration of a muscle can be calculated, considering the muscle as a structure composed of active fiber elements, able to contract under activation within the deformable connective tissue continuum, Kojic et al. (1998).

Macroscopic model: The governing equilibrium equation of a FE structure in deformed configuration at a time step (t) and iteration (i) is formulated as:

$$\left({}^{t+\Delta t}K_{el} + {}^{t+\Delta t}K_{mol}\right)^{(i-1)} \delta U^{(i)} = {}^{t+\Delta t}F_{ext}^{(i-1)} + {}^{t+\Delta t}F_{int}^{(i-1)} + {}^{t+\Delta t}F_{active}^{(i-1)} \quad (1)$$

where ${}^{t+\Delta t}F_{ext}^{(i-1)}$, ${}^{t+\Delta t}F_{int}^{(i-1)}$, and ${}^{t+\Delta t}F_{active}^{(i-1)}$ are vectors of external physiological loads, internal (structural) nodal forces, and integrated active molecular forces lumped into FE nodal forces, respectively; ${}^{t+\Delta t}K_{el}^{(i-1)}$ and ${}^{t+\Delta t}K_{mol}^{(i-1)}$ are stiffness matrices of the passive components of constitutive FE and of cumulative stiffness of acto-myosin bonds, respectively; $\delta U^{(i)}$ are the increments of nodal displacements at iteration (i) and the left-upper index $t+\Delta t$ indicates that the equilibrium equations correspond to the end of the time step.

Evaluation of nodal internal and active forces in (2) demands determination of stresses in muscle fibers corresponding primarily to their stretch rates. This can be done by employing a phenomenological model or a more precise and complex physiological model such as Huxley's.

Microscopic model. Following Huxley, a muscle fiber constitutive unit is represented by interacting actin and myosin filaments following Huxley (1957). Elastic spring like connections between these filaments, formed via so-called cross-bridges, generate in aggregate the active muscle force and stiffness, Torelli (1997). Depending on boundary conditions, over time the filaments can slide relative to each other so cross-bridges can experience both tension and compression. Following these rules Huxley's sliding filament theory (McMahon 1984) is defined by the following partial differential equation defined over the domain Ω:

$$\frac{\partial n}{\partial t}(x,t) - v\frac{\partial n}{\partial x}(x,t) = \mathrm{N}\left(n(x,t),x\right), \qquad \forall x \in \Omega, \tag{2}$$

where *n* is fraction of attached cross-bridges displaced for *x* from its strain free position at time t; $v = -dx/dt$ is the shortening velocity of the thin filament with respect to the thick filament; $\mathrm{N}\left(n(x,t),x\right)$ is compounded transition flux between attached and detached state. For solving the first order hyperbolic equation (2) we used method of characteristics (Lister 1960). The total stress σ is expressed as the contribution of active muscle forces and the contribution of (passive) elasticity of collagenous connective tissue, cell membrane, and muscle non-contractile cytoskeleton in parallel to muscle cells: $\sigma = \sigma_m\phi + \sigma^E(1-\phi)$, where ϕ is the fraction of muscle fibers in the total muscle volume and σ^E is the stress in passive part of the muscle. More detailed information on two-scale model can be found in (Ivanovic 2015).

2.2 Programming model

We have developed a software platform for parallel and distributed execution of two-scale muscle model simulations in heterogeneous CPU/GPU environment. The starting point in the applied strategy was dividing each iteration, during incremental two-scale model simulation, into two distinct sets of algorithm steps. The first set considers finite element algorithm calculations like assembling the element balance equations (1) and solving the FE equilibrium equations for the entire muscle. The second set considers micro-scale models of muscle fibers for each integration point within FE mesh. Our solution applies parallelization only to the second set, i.e. to micro-scale model calculations, while FE algorithm executes in sequential manner. The justification for this approach comes from the performance analysis of the sequentially executed simulations. Micro model computations take more than 99.9% of total execution time.

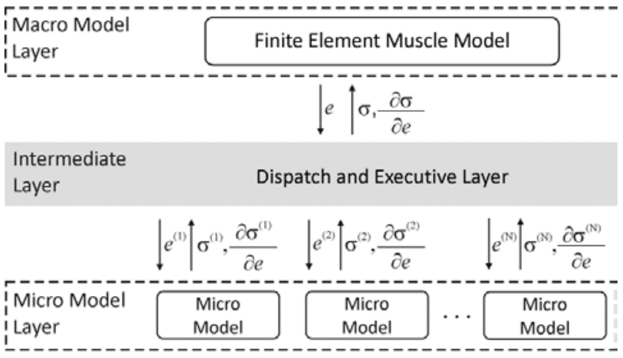
Macro Model Layer
Finite Element Muscle Model
e
$\sigma, \frac{\partial\sigma}{\partial e}$
Intermediate Layer
Dispatch and Executive Layer
$e^{(1)}$
$\sigma^{(1)}, \frac{\partial\sigma^{(1)}}{\partial e}$
$e^{(2)}$
$\sigma^{(2)}, \frac{\partial\sigma^{(2)}}{\partial e}$
$e^{(N)}$
$\sigma^{(N)}, \frac{\partial\sigma^{(N)}}{\partial e}$
Micro Model Layer
Micro Model
Micro Model
...
Micro Model

Fig. 1. Mexie architecture: High-level design. The proposed system embodies the interface between a macro-scale layer, including a finite element model, a micro-scale layer, including a material model that describes muscle behavior at the points of integration, and an intermediate layer, which acts to mediate the macro and micro-scale functions

The performance of parallel simulation runs in the *Mexie* solution assumes a set of MPI processes, with the fundamental idea being to divide the entire set of micro models into smaller collections called chunks. One of the processes, having a role of manager, executes FE calculations. When the simulation algorithm reaches the point where micro model calculations should begin, each participating process performs simulations over micro models from its domain of responsibility. Due to this domain decomposition, we reduce process intercommunication to only two operations: scatter of the deformations and gather of the stresses together with stress derivations. Any of the processes can be delegated to use the GPU device for calculations.

The *Mexie* system consists of three functional units: (1) the macroscopic model layer, (2) the intermediate model layer, responsible for coupling and distribution, and (3) the microscopic model layer (Figure 1). The macro model layer enables setting-up FE model and running the entire two-scale simulation. In each iteration, the data containing current deformations in all integration points of the FE model are sent to the intermediate layer. Upon successful execution of the micro model steps, intermediate layer returns stresses and their derivatives with respect to deformation.

The micro model layer includes material models used to describe muscle behavior within the *integration points* of the macro model. There is an independent micro model for each integration point, which implies establishing the algorithm that simulates the behavior of elementary building blocks like muscle fiber or sarcomere, and defining model parameters and model state variables. Micro model layer does not carry any information regarding macro model and communicates exclusively with the intermediate layer. At the request of the intermediate layer, for a given strain, micro model layer runs simulation over the corresponding micro model, in order to determine stress and its derivative with respect to strain. In order to enable utilization of various processing units (CPUs and GPUs), we have developed two distinct implementations of the Huxley micro model. The first one is dedicated to the execution on CPUs, and the second one employs GPUs and uses CUDA programming model.

The intermediate layer acts as a mediator between the macro-scale and the micro-scale layers, and its major role is to implement adopted strategy of parallelization. This layer is also responsible for binding data, like which micro model is associated to which FE integration point. Prior to the simulation run, we have to perform mapping of the integration points to the MPI processes, which remains unchanged during the entire simulation run. This static mapping in heterogeneous CPU/GPU environment is also a task that belongs to the intermediate layer.

Static load balancer. We utilize GPUs for the purpose of micro models computations. The CUDA implementation assumes the use of GPU in executing numerically intensive parts of the Huxley model simulation. The simulation contains massive vector operations, which are inherently suitable for SIMD parallelization model. Due to the heterogeneous nature of the computing environment employed, task scheduling has a significant impact on *Mexie* performance. We developed a static scheduler which objective is to perform initial partitioning of the set of all material models into chunks and assign those chunks to the participating MPI processes. Scheduling policy takes into account the speed of the processes and memory capacity of the devices on which the processes take place. The algorithm (shown in Fig. 2) assumes the following input parameters:

- **Calculation speeds** of the participating MPI processes, identified as $V_i, i = \overline{1,k}$, where k is the total number of MPI processes (including ones which purpose is restricted to GPU kernels invocation). We define the calculation speed as the average number of Huxley micro model iterations that the process executes in a unit of time.
- **Memory capacities** of the devices associated to the MPI processes, identified as $M_i, i = \overline{1,k}$. We define the memory capacity as the maximum number of micro models which state variables can be stored during simulation run. In case that the process is associated to a GPU, the size of the “global” memory dictates the capacity

value. When it comes to processes that are executed solely on the CPUs, their memory capacity is considered "unlimited".

The first step in proposed scheduling scheme is obtaining the value of time *T*, required to perform simulations over total of *N* instances of micro-models, if memory is taken as unlimited:

$$T = \frac{N}{\sum_{i=1}^{k} V_i} \tag{3}$$

For each process, we determine the chunk size (number of micro models), that the process with speed V_i is capable to complete in time *T:*

$$C_i = T \cdot V_i \,. \tag{4}$$

However, in case that for *i*-th process rank, the projected number of integration points exceeds memory capacity $C_i > M_i$, we have to adjust the speed of this process to $V_i = \frac{M_i}{T}$. Upon rewriting speeds, we have to recalculate synchronization time *T* and consequently the size of the chunks. The adjustments of V_i and C_i must be performed in iterative manner, until all memory capacity requirements are satisfied. As the scheduler tends to distribute more models to faster processes (in this case the ones associated with the GPUs), it is more likely to reach their memory capacity, then reaching memory capacity of the slow processes associated with the CPUs.

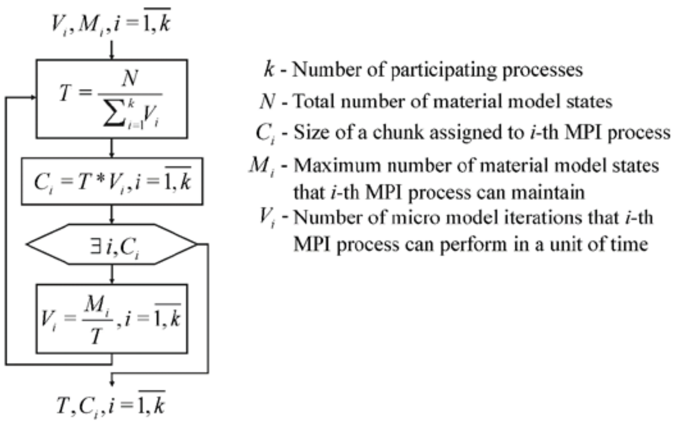
$V_i, M_i, i = \overline{1,k}$
$T = \frac{N}{\sum_{i=1}^{k} V_i}$
$C_i = T * V_i, i = \overline{1,k}$
$\exists\, i, C_i$
$V_i = \frac{M_i}{T}, i = \overline{1,k}$
$T, C_i, i = \overline{1,k}$
k - Number of participating processes
N - Total number of material model states
C_i - Size of a chunk assigned to i-th MPI process
M_i - Maximum number of material model states that i-th MPI process can maintain
V_i - Number of micro model iterations that i-th MPI process can perform in a unit of time

Fig. 2. The algorithm for task scheduling which aim is to divide the entire micro-scale domain into chunks and assign their integration to slave processes, taking into account their speed and memory capacity

2.3 Speed-up results

Figure 2 shows speed-ups for the larger benchmark model, which consists of 1000 quadrilateral finite elements, totaling to 4000 Huxley micro models. As stated earlier, GPU unit is capable to keep only limited number of micro models' states throughout the simulation process. This limitation influences distribution of duties among the processes, and consequently achieved speed-up value. Depending on the size of models and applied configurations, the impact of the GPU units on the total speed-up can be less than assumed (~22 CPUs). Our Tesla M2090 units with 6GB have a capacity of approximately 1000 Huxley micro models. In accordance to the optimistic assumption that a single GPU unit delivers processing speed equal to that of 22 CPUs, the configurations denoted as 0/66 and 2/22 should give approximately equal speed-up. However, in case of larger model (4000 Huxley micro-models), the GPU is limited to 1000 points, which is approximately 11 times the number of points allocated by CPU units. This means that the influence of a single GPU is approximately equivalent to the influence of the 11 CPUs (not idealistic 22). In fact, in terms of achieved speed-up, the configuration denoted as 2/22 is approximately equal to the configuration 0/44 (both of them allocate 1000 micro models per GPU), Fig. 3.

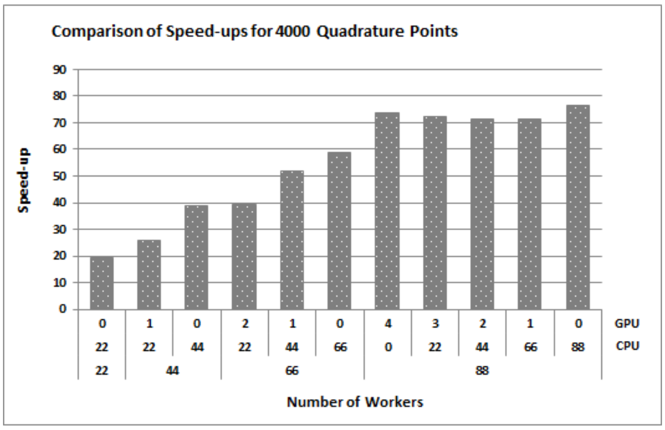

Fig. 3. Speed-ups obtained for the benchmark model with 1000 FEs, obtained with: 1/0, 0/22, 2/0,1/22, 0/44, 3/0, 2/22, 1/44, 0/66, 4/0, 3/22, 2/44, 1/66 and 0/88 GPUs/CPUs

To summarize, the key features of *Mexie* software solution, giving it advantage over other related solutions are following. (1) The ability to employ GPU accelerators for very efficient parallel execution of the multi-scale muscle models. (2) *Mexie* shows strong scaling, thanks to the hybrid programming model and optimized static task scheduler; (3) the significant improvements in performance by using GPUs provide very large performance-per-watt ratio and therefore achieves the same speedup at significantly lower cost and power consumption comparing to traditional multicore CPUs; (4) the two orders of magnitude speed-up provides opportunity for development of complex multi-scale models and their usage in everyday medical practice.

3. Computer search for trees with minimal ABC index

In this section we introduce another value of HPC approach to problem solving, now in the field of graph science. The ABC index is a degree-based molecular structure descriptor that found chemical applications. Finding the connected graph(s) of a given order which ABC index is minimal is a hitherto unsolved problem, but it is known that these must be trees. By combining mathematical arguments and computer-based modeling including employment of HPC resources, we establish the basic structural features of the minimum-ABC trees.

The physico-chemical applicability of the ABC index is based on the fact that there exists an excellent (linear) correlation between ABC and the experimental heats of formation of alkanes (Estrada 1998, Gutman 2012). In the paper, Estrada (2008) established the physical basis for this correlation. In fact, a new mathematical model was put forward, capable of rationalizing (both qualitatively and quantitatively) the experimentally established regularities for the stability of linear and branched alkanes, as well as for the strain energy of cycloalkanes. In Estrada's model, the term (d_i+d_j-2)/(d_i d_j) in Eq. (5) is interpreted as the count of the 1,2-, 1,3-, and 1,4-interactions in the carbon-atom skeleton. These interactions play a fundamental role in the energetics of alkane molecules. If G is a graph of order n, and if d_i is the degree (=number of first neighbors) of its i-th vertex, i=1, 2,…, n, then

$$ABC = ABC(G) = \sum_{ij} \sqrt{\frac{d_i + d_j - 2}{d_i d_j}} \,, \tag{5}$$

with the summation going over all pairs of adjacent vertices. When a new graph invariant is studied, the first question is to determine its minimal and maximal value for graphs of a given order and to characterize the respective external species. Whereas the finding of the n-vertex graph with maximal ABC index is an easy task, the structure of the connected n-vertex graph(s) with minimal ABC appeared to be a significantly more difficult problem, Gutman et al. (2012a). In Furtula et al. (2012), we perform a computer search and use computer-aided models, aimed at bringing us closer to the solution of the problem. Of the numerous already established properties of the ABC index we point out the inequality ABC(G)>ABC(G-e) which holds for all edges e of any graph G according to Chen et al. (2011), and Das et al. (2011). Its immediate consequences are that

(1) among all graphs of order n, the complete graph K_n has the greatest ABC index, and

(2) among all connected graphs of order n, the smallest ABC index is attained by one or more trees.

Recall that, trivially, the n-vertex graph with minimal ABC index (equal to zero) is the edgeless K_n. In addition, the n-vertex tree with maximal ABC index is the star (Furtula et al. 2009), which also is an easy result.

3.1 Computational search

In order to get some idea about the structure of the n-vertex tree(s) with minimal ABC index, we decided to check all trees of order n, up to n as large as possible, and to single out the tree(s) with smallest ABC. It was hoped that after some value of n the form of the minimum-ABC tree will emerge, enabling us to formulate a sound conjecture on its general structure. Whereas the evaluating of the ABC index for any particular tree is an easy computational task, the true problem is that with increasing value of n the number of trees rapidly increases and becomes prohibitively large. Table 1 shows how many trees would be needed to examine in order to perform our naive direct approach. In view of the rapidly growing number of n-vertex trees, it turns out that the task of generating trees and identifying the minimum-ABC species using a single PC is feasible up to n=25. For larger values of n a special strategy had to be adopted.

Consider, for example, the case of n=29, when 5,469,566,585 distinct trees need to be generated and their ABC-values computed. Using Python script on a modern 2.4 GHz CPU to fulfill this task, it turns out that average speed on a single core is about 200,000 trees/min. Therefore, the entire process of calculating the ABC indices would take about 19 days on a single CPU core. Our algorithm for identifying the minimum-ABC tree(s) consists of two successive steps:

(1) Generating the trees using a recursive scheme.

(2) Computing the ABC index for each generated tree in order to find its minimum value.

According to the theoretical discussion found in Quinn (2004), phase (1) is not parallelizable at all, due to the recursive nature of the algorithm used. Even with all modern distributed computing technologies and frameworks, some algorithms have inherent sequentiality and cannot be efficiently decomposed into smaller semi-independent parts having the ability to be executed concurrently. The situation is opposite regarding phase (2), since searching inside graph spaces is an easily parallelizable task, with single point of synchronization – reduction at the very end. In the simplest scenario, the work can be shared equally among the available processors statically, before the calculation takes place. After each CPU finds the minimum ABC index inside its own tree subspace, the result is reported to the master which is then responsible to compute the grand total. In a more complicated scenario, the work could be distributed dynamically, but in this paper, the first (static load balancing) scenario was employed, since it is more suitable for the grid computing platform.

n	**No. of trees**	***n***	**No. of trees**
15	7741	25	104636890
16	19320	26	279793450
17	48629	27	751065460
18	123867	28	2023443032
19	317955	29	5469566585
20	823065	30	14830871802
21	2144505	31	40330829030
22	5623756	32	109972410221
23	14828074	33	300628862480
24	39299897	34	823779631721

Table 1. Number of trees with *n* vertices

There is another issue with splitting the entire process into phases (1) and (2) as we did. That means that as soon as phase (1) finishes, the trees have to be stored in order to be distributed to the assigned CPUs. Taking the same example of *n*=29, using standard ASCII format, more than 400 GB of disk space would be necessary to store all trees of the mentioned order. Having in mind that such large files have to be transferred over the network to reach the appropriate nodes in the distributed computing scheme, it is obvious that it would be a big waste of time and network resources. The solution capable of reducing storage space up to 27 times has been found in the ZIP streaming; instead of storing trees in pure ASCII sequence, the phase (1) algorithm directly stores trees into compressed archives, one archive per each CPU in the distributed framework.

This way, only 16 GB is needed to store all trees of order 29, instead of 400 GB in pure ASCII. More specifically, the algorithms have been developed for the grid distributed computing platform, based on middleware developed under EGI (European Grid Initiative), but can be seamlessly ported to any other similar distributed platform. The computational resources employed are part of the Serbian national grid infrastructure AEGIS. The real implemented solution consisted of the following steps:

1. Generation of all trees of the given order n and storing them into multiple compressed archives on-the-fly. For example, all trees of order 31 were contained into 758 zipped files of about 150 MB size each, placed on the grid UI (User Interface).
2. Transferring the generated files to the grid SEs (Storage Elements). It is recommended that the files be finally deployed at the SEs nearby the computing resources in order to reduce time needed to copy them to the nodes, where ABC index calculation would eventually take place. Each grid site was equipped with one or more SEs.
3. Calculating the ABC index for each tree on the distributed WNs. Triggered by the submitted command on UI, the distributed calculation on multiple WNs takes place. The start script first transfers file from the SE assigned to it, then decompresses it on the local scratch space. Each job then starts calculating the ABC index for each tree, leading to the local minimum.
4. Gathering local minimum ABC index data and finding the global minimum. As soon as all submitted jobs finish, the output data can be retrieved on the UI and the global minimum of ABC index obtained, including the structure of the tree(s) assigned to it.

By means of our computational method we were able to do the calculations until *n*=31. In this latter case, about 400 CPUs of the grid infrastructure were employed at the same time, thus reducing the computation time from (theoretical) 140 days on a single machine to only about 14 hours, including file transfes. One must pay attention that these times do not take the first step of tree generation into account. This recursive sequential algorithm can take considerable amount of time to execute, i.e. almost 7 days for *n*=31 on a modern CPU.

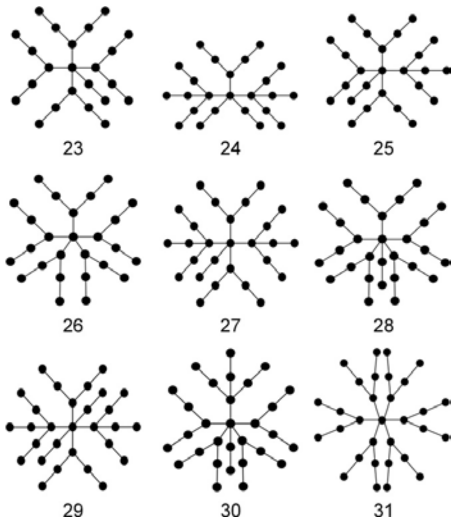

Fig. 4. Trees with *n*=23, 24,..., 31 vertices with minimal value of the atom–bond connectivity (ABC) index. These were determined by calculating ABC of all *n*-vertex trees

The trees with minimal ABC index for *n*=23, 24,…, 31 are depicted in Fig. 4. The analogous results for smaller values of *n* are found in the earlier paper (Gutman et al. 2012a). As established earlier in Gutman et al. (2012), for *n*=16 there are two minimum-ABC trees. According to the presented calculations, for 17<*n*<31 the minimum-ABC tree is unique. By inspecting Fig. 4, we can envisage some (possible) structural features of the minimum-ABC trees. These we state as:

Proposition 1. (a) The minimum-ABC tree possesses a single central vertex of high degree. (b) To this central vertex only branches of the type B1, B2, and B3 are attached, depicted in Fig. 5. (c) For *n*=16, the minimum-ABC tree with *n* vertices is unique.

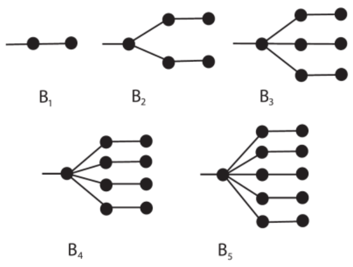

Fig. 5. *Proposition 1* claims that to the central vertex of the minimum-ABC trees only branches of the type B1, B2, and B3 are attached. We nevertheless considered the possibility that also greater branches of the same type, such as B4 and B5, occur at higher values of *n*

After proposing plausible structural model, testing its properties and refining proposed model considering large number of vertices, the section can be concluded with:

Proposition 2. If n is sufficiently large (as specified above), and assuming that the minimum-ABC tree possesses a single central vertex of high degree, then the following holds. (a) To the central vertex only branches of the type B2, B3 and B4 are attached, depicted in Fig. 5. (b) In a minimum-ABC tree there can be at most two B2 branches, and at most three B4 branches, but never both B2 and B4 branches. All the remaining (numerous) branches are of B3-type. (c) If $n \equiv 2 \pmod 7$, then the minimum-ABC tree possesses an external path of size 2. (d) For *n*>168, the minimum-ABC tree with *n* vertices is unique.

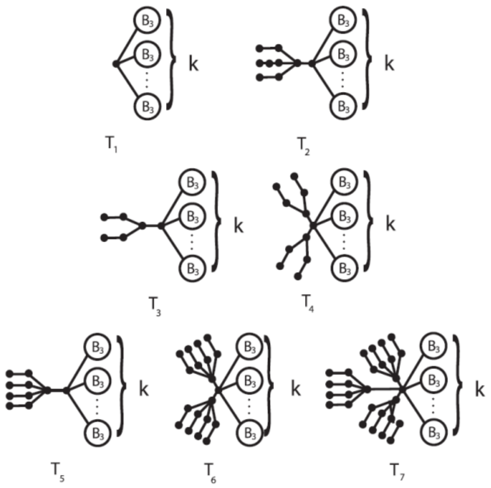

Fig. 6. The seven types of minimum-ABC trees (form of the branch B3 is shown in Fig. 5)

4. WoBinGO: A parallel framework for genetic algorithm based optimization

One of the most frequently encountered challenges in applied science and engineering, is optimization. Genetic algorithms (GAs) (Goldberg 1989) have proven themselves as robust and powerful mechanisms when it comes to solving complex real-world optimization problems. GA is characterized by a large number of function evaluations. Due to the time-consuming fitness evaluation functions found in real-world problems, it may take days or months for the GA to find an acceptable solution. Speeding up the optimization process is achieved by parallelization of GA according to Munawar et al. (2008), Alba et al. (2002), which reduces the resolution times to reasonable levels. Grid computing environments provide the infrastructure for implementing parallel meta-heuristics. There, researchers face new difficulties associated with developing and deploying a Grid based application. The fact that Grid resources are distributed, heterogeneous and non-dedicated, makes writing parallel Grid-aware applications very challenging, according to Foster (1995). In this context, tools for simplifying Grid application development, by hiding the complexity of Grid computing from the researchers, can significantly enhance Grids harnessing by scientific and engineering applications.

We present the WoBinGO framework for solving optimization problems over heterogeneous resources, including HPC clusters and Globus-based grids. Although it is possible to utilize diverse computing resources for solving optimization problems in parallel (multiple university clusters, grids), having in mind the immense computing power offered by grids, we will restrict our discussion in this paper only to this kind of deployment. The framework was designed to meet the following goals: (1) speeding up the optimization process by parallelization of GA over the Grid; (2) relieving the researcher burden of obtaining Grid resources and dealing with various Grid middlewares; (3) enabling fast allocation of Grid jobs to avoid waiting until requests for computing resources are processed by Grid middleware; (4) providing flexible allocation of worker jobs in accordance with the dynamics of the users' requests, thus avoiding the unnecessary reservation of computing resources.

The framework is dedicated for parallel execution of single and multi-objective optimization using GA on the grid. It uses a master–slave parallelization model and allows both: parallel evaluation of a population in GA and parallel execution of multiple instances of the parallel GA. As a novelty, this framework incorporates the Work Binder (WB), Marovic et al. (2001), which provides almost instant access to grid resources and interactivity for client applications. Integration of WB into the framework enables the programmer to focus solely on the optimization problem without having to worry about specific details of Grid computing. Additionally, WB increases the utilization of the Grid infrastructure by offering automated elasticity in its occupancy, based on present and recent client behavior. Furthermore, a single WB service is capable of serving multiple users with multiple GA instances, where for each instance of GA a population evaluation is also parallelized. Due to the multi-tier design, it is easily possible to replace master-slave parallelization model with hierarchical parallel GA with master-slave demes (Cantu-Paz et al. 2000) or with PEGA (parallel cellular GA) (Dorronsoro et al. 2007), keeping all the other components intact. The framework also adheres to the standard Globus security mechanisms, including GSI and MyProxy.

Much work has been done on parallelizing GAs and developing parallel metaheuristics frameworks which employ grid resources. On the other hand, pilot-job systems have been developed to improve the utilization of production grids, greatly reducing middleware overheads, by decoupling workload submission from resource assignment. However, to our knowledge, there is no framework for parallel meta-heuristics that employs a pilot-job system dynamically using elastic resource provisioning. We argue that elastic resource provisioning can be as efficient as static pilot-job infrastructures used in previous frameworks, but with the key advantage of avoiding unnecessary occupation of Grid resources.

The key feature that WB provides, as a part of the WoBinGO framework, is its inherent frugality in resource consumption, provided by two distinct aspects: (1) the system adapts the number of placeholder jobs to the current workload, and (2) a defined maximum job lifetime prevents endless waiting of other users' jobs in the batching queues. This section presents the key features of the WoBinGO framework, giving its advantages over other related solutions:

- Optimization using parallel GA can be performed over diverse computing resources, including Grid and HPC clusters.
- The complexity of the underlying Grid infrastructure is hidden from the user.
- Automatic adaptive allocation of jobs with limited lifetime which provides friendliness towards other batching queue users.
- A single WB service is capable of serving multiple users with multiple GA instances, sharing the same pool of ready jobs.
- Quick client-worker binding is enabled by elastic maintenance of the pool of ready jobs.
- Development of objective functions and optimization algorithms are independent, and can be performed by different developers.
- Objective function can be written in any compiled or script language supported by an underlying OS/runtime environment.

4.1 Architecture

The basic structure of the framework is illustrated in Fig. 7. The framework consists of the optimization master and the distributed evaluation system based on WB. The distributed evaluation system is composed of the evaluation pool and the WB subsystem. The master executes the main evolutionary loop and the distributed evaluation system takes care of the Grid execution of the evaluation processes. It should be noted that the evaluation pool is NOT the same entity as the pool of ready jobs created and maintained by WB itself.

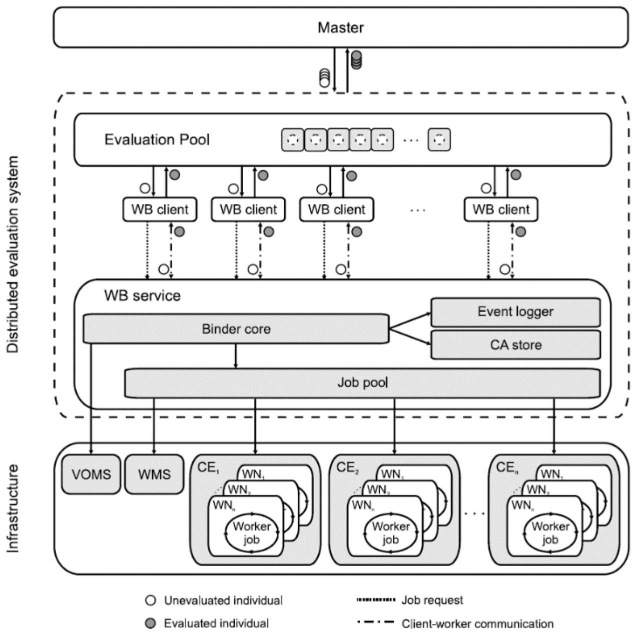

Fig. 7. WoBinGO architecture

The master performs the main evolutionary loop to the point when a generation has to be evaluated. At that moment, the master sends all individuals in a generation to the evaluation pool. After all individuals from a generation have been evaluated, and assigned with an objective function value, the master proceeds with the rest of the evolutionary algorithm until the stopping criteria is reached. The evaluation pool acts as an intermediate layer between the master and the WB service. It provides asynchronous parallel evaluation of individuals from a generation. Each time a generation has to be evaluated, the evaluation pool receives individuals from the master and enqueues them. The evaluation pool invokes the WB client for each of the queued individuals. When an individual is evaluated, the evaluation pool receives the result back from a WB client and then assigns objective function value to the corresponding individual. The WB environment consists of software components distributed in three tiers: the client, the worker, and WB service. The purpose of the WB service is to maintain the pool of ready worker jobs on the grid and to bind them with clients that request evaluation. It submits jobs to CEs in order to load enough worker jobs in the pool for incoming requests. The client establishes a connection to the WB service, and requests a worker. As soon as the client and worker are successfully coupled, the WB service acts as a proxy that relays traffic between them. The client sends an individual to a worker which computes fitness value and sends it back through WB proxy. After completing the evaluation, the worker reconnects to the WB service asking for more work within job time limits determined by WB configuration. The job lifetime cannot exceed the limit specified by the local grid site administrator, obtained using MDS.

4.2 Theoretical speed-up analysis

With the WoBinGO framework, the goals of near-optimal usage of Grid resources and high availability of ready jobs are achieved by implementing an adaptive job submission and pool management policy. These are, however, fulfilled without undue sacrifice of Grid resources, unlike previous solutions involving static pilot job infrastructures.

According to the WB's elastic resource provisioning, the number of workers changes over time. If $\overline{p}$ is the average number of processors that perform evaluation of *n* individuals, *λ(n)* serial part, $O(n,\ \overline{p}\,)$ parallel overhead and t_{eval} is the average time needed to evaluate single individual, then speed-up *S* can be expressed as follows:

$$S = \frac{\lambda(n) + n{\cdot}t_{eval}}{\lambda(n) + \dfrac{n{\cdot}t_{eval}}{\overline{p}} + O(n, \overline{p})} \tag{6}$$

In order to achieve any speed-up, following Ivanovic et al. (2015) evaluation time should be kept above:

$$t_{eval} > \frac{\overline{p}}{n(\overline{p}-1)}{\cdot}O(n, \overline{p}) \approx \frac{1}{n}{\cdot}O(n, \overline{p})\,. \tag{7}$$

4.2 Empirical speed-up analysis

We present an empirical study of the WoBinGO framework. The first aim was to determine its performance in terms of infrastructure utilization, achieved speed-up and inherent overhead, while varying t_{eval}. The second aim was to obtain the framework's speed-up when t_{eval} is constant. The third aim was to estimate to which extent WoBinGO's characteristic to limit pool jobs lifetime assists other batching queue users. Using a simple GA with real-encoded chromosomes we solved the artificial test problem. The objective function was a dummy function that does nothing except receive individuals, sleep for t_{eval} seconds, and return random fitness value to the master. Because we have aimed to determine communication overhead, it did not matter whether the workers were performing calculations or simply sleeping, it only mattered how long it took them to return their responses.

The benchmark was carried out using a single grid site, in order to put aside all internetworking effects and explore theoretical limits of the WoBinGO framework. The site AEGIS04-KG consists of 6 nodes, each equipped with 2 AMD 16-core CPUs and 96 GB RAM, totalling 192 processors. As AEGIS04-KG is a part of EGI infrastructure, the jobs were not submitted by using the batching system directly, but by employing EMI/UMD services, including WMS (Workload Management System), which is not collocated with the site. The population size during all experiments was set to 500 individuals. The estimated average time required for the sequential part of the algorithm (denoted *λ(n)* in Eq. (6)) is 200 ms.

t_{eval}	$\overline{p}$	T_{ser}	T_{par}	T_{ser}/T_{par}	$O(n, \overline{p})$	$O(n, \overline{p})/T_{par}$ (%)
0.5	25	2.5E+03	4.13E+02	6.05	3.13E+02	76
1	49	5.0E+03	4.09E+02	12.23	3.07E+02	75
2	48	1.0E+04	4.37E+02	22.86	2.29E+02	52
5	74	2.5E+04	5.17E+02	48.4	1.78E+02	35
10	80	5.0E+04	7.40E+02	67.52	1.15E+02	16
30	91	1.5E+05	1.76E+03	85.21	1.12E+02	6
60	95	3.0E+05	3.25E+03	92.22	9.51E+01	3
120	96	6.0E+05	6.35E+03	94.55	1.17E+02	2

Table 2. Empirical results obtained by running an artificial problem over WoBinGO using a simple GA (time is shown in seconds)

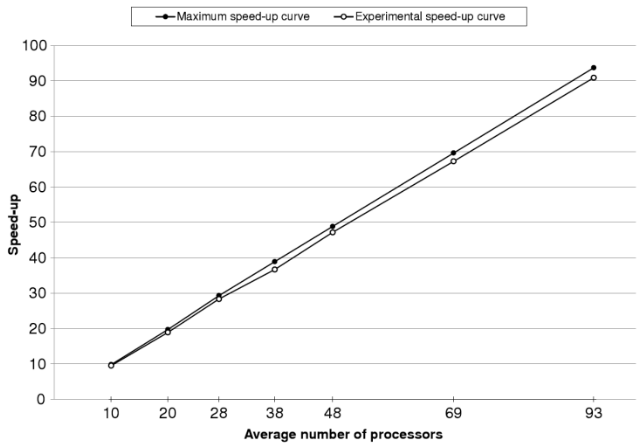

Fig. 8. Speed-up curve obtained by running an artificial problem over WoBinGO

The average number of processors $\overline{p}$ that have been used to evaluate individuals was determined experimentally. The WoBinGO framework was configured so that the maximum allowed number of worker jobs (p_{max}) is 100. However, $\overline{p}$ varies in accordance with time t_{eval} taken by a single fitness value calculation. With short evaluations, the workers quickly become reusable so there is rarely the need to submit new jobs. On the other hand, with longer evaluations, a significant amount of time passes until the workers become reusable. Therefore, new jobs must be submitted more often. Consequently, $\overline{p}$ has larger values for longer fitness evaluations. T_{ser} is the time needed to evolve a population of individuals on a single processor and T_{par} is the time needed to evolve the same population in parallel, using $\overline{p}$ processors.

T_{ser}/T_{par} is the speed-up achieved by the WoBinGO framework, and $O(n,\overline{p})$ is the communication overhead. The value of $O(n,\overline{p})$ was determined by substituting T_{par}, $\lambda(n)$, t_{eval} and $\overline{p}$ into denominator of the Eq. (6).

As can be observed from Table 2, the overheads are quite significant for smaller problems. This is due to the fact that for problems with short evaluations the frequency of client and worker requests addressed to the WB service is greater than with longer evaluations. Higher frequency implies longer waiting for the requests' fulfilment.

For the problems with the more time-consuming fitness evaluation considerable speed-ups can be achieved, as the overhead becomes almost constant for t_{eval}>5s. Fig. 8 contains two speed-up curves: the maximum speed-up curve and the experimental speed-up curve. The maximum speed-up curve is obtained from Eq. (6), assuming $O(n,\overline{p})=0$. The following parameter values are used: n=5000(=500 ·10), t_{eval} = 60s, $\lambda(n)$=200 ms, while $\overline{p}$ was determined experimentally. As can be seen from Fig. 8, the theoretically predicted speed-up fits nicely with the observed experimental speed-up. The difference between these speed-ups comes from the communication overhead, which was excluded. Speed-up curve shows that the framework is suitable for problems with computationally expensive evaluations which are common in real-world applications. Furthermore, considering the condition (7), we have experimentally obtained that there is no point in using the WoBinGO framework for t_{eval}<0.1 s.

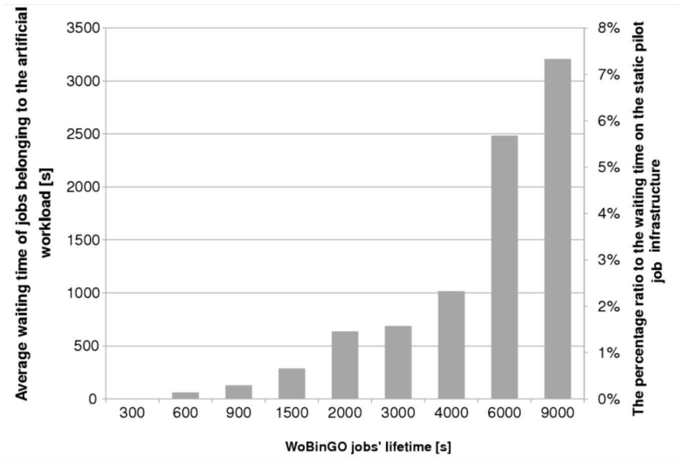

Fig. 9. The average waiting time of a job belonging to the artificial load. The ratio to the waiting time on the static pilot job infrastructure is shown on secondary ordinate axis

The final goal was to benchmark the feature that distinguishes WoBinGO framework from the previous static pilot job solutions - limited pool job lifetime. This specific feature enables other batching queue jobs to reach running state with less delay. In order to estimate that delay, the artificial test problem was used, as previously. t_{eval} was set to 60s, population size was 500 and the maximum number of processors was 150. The artificial load of 18 jobs/h was created, with job submission occurrence that fits the Poisson distribution. Duration of these jobs respects *Gamma* distribution (Ivanovic et al. 2015).

For the sake of reproducibility, all these values were taken as an approximation of the real AEGIS04-KG load for the period of two years in production. Additionally, AEGIS04-KG scheduler was set to keep the sum of processors held by WoBinGO and the processors held by the artificial job submitter to 150. We measured how the average waiting time of a job belonging to the artificial load depends on WoBinGO jobs' lifetime. Each experimental point resulted from 12h run. The obtained results are shown in Fig. 9.

It can be noticed that for shorter lifetimes of WoBinGO jobs, the jobs belonging to the artificial load quickly reach running state. The average waiting time increases for longer lifetimes of WoBinGO jobs, but does not exceed 1h, even with WoBinGO jobs' lifetime of two and a half hours. Considering the fact that with the static pilot job infrastructure, the jobs belonging to the artificial load would certainly have to wait in the batching queue till the end of the optimization run (12h), shown results are quite remarkable. The percentage ratio to the waiting time on the static pilot job infrastructure is shown on the secondary ordinate axis on the right. Even for very long WoBinGO jobs' lifetimes, the waiting time of the artificially loaded jobs is below 8% of the time they would spend in the batching queue if a static pilot job infrastructure was used instead of WoBinGO framework.

We employed WoBinGO in a number of real world optimization tasks and reported obtained results. Multi-objective calibration of a leakage model at the Visegrad power plant (Republic of Srpska, Bosnia and Herzegovina) was reported in Ivanovic et al. (2015). Another use case, from the field of finance, dealing with Mean-Capital Requirement portfolio optimization, was reported in Drenovak et al. (2016).

The framework provides a novel, elastic approach in utilizing computing resources in accordance with the dynamics of the users' requests. Unnecessary reservation of computing resources is avoided through flexible allocation and limited lifetime worker jobs.

5. Conclusion

We have presented three distinct use cases of high performance computing utilization. Definitely, HPC involvement made these research ventures feasible. In case of multi-scale muscle modeling we achieved speed-ups of over two orders of magnitude with even higher scalability promise if using more CUDA capable devices. Future research in this area will target at incorporating more realistic models at micro level, i.e. Duke (1999) and Smith et al. (2007).

Within the area of graph science, we presented relatively narrow use case of brute force search, aimed at finding trees with minimal ABC index. Presented framework could also be employed in a much broader context. Involving Hadoop and Spark frameworks would considerably simplify these calculations, but issue of HPC/Hadoop peaceful coexistence on the same hardware remains to be solved.

Our WoBinGO framework efficiently performs elastic resource provisioning within large scale GA optimization tasks. We presented the results undertaken in computing cluster and grid environments, demonstrating near ideal speed-up for time consuming evaluations. In near future, we plan to check how this elastic and frugal behavior could impact instantiation of instances within public cloud infrastructures.

Acknowledgements. Part of this research was supported by The Ministry of Science in Serbia, Grants III41007 and OI174028.

Извод

Рачунарство високих перформанси у вишескалном моделовању, теорији графова и оптимизацији заснованој на метахеуристикама

М. Ивановић[1]*, Б. Стојановић[1], В. Симић[1], А. Капларевић Малишић[1], В. Ранковић[2], Б. Фуртула[1], С. Мијаиловић[3]

[1] Природно-математички факултет, Универзитет у Крагујевцу, Радоја Домановића 12, Крагујевац, Србија
имејл: mivanovic@kg.ac.rs
[2] Економски факултет, Универзитет у Крагујевцу, Ђуре Пуцара Старог 3, Крагујевац, Србија
[3] Department of Chemistry and Chemical Biology, College of Science, Northeastern University, Бостон, МА, САД
**главни аутор*

Резиме

Једна од главних активности Групе за научне прорачуне на ПМФ-у у Крагујевцу су методе ефикасне употребе паралелних рачунарских архитектура, као што су кластери SMP чворова, системи са дељеном меморијом и графички процесори. Примарни циљеви су дизајн, развој и имплементација паралелних алгоритама и структура података приликом решавања фундаменталних проблема науке и технике. MPI је важна програмска парадигма која још увек поставља различите изазове, нарочито када се комбинује са другим програмским моделима, као што је CUDA. Поред стандардне HPC (High Performance Computing) батерије алата, унутар Групе је присутна и експертиза за алате као што су Hadoop и Spark. У овом раду ће бити дат преглед резултата Групе који се односе на конкретне апликације на рачунарским системима великих размера, заједно са референцама у виду оригиналних чланака.

Први део се бави вишескалним моделирањем мишића у хибридном MPI-CUDA окружењу. Наш приступ комбинује коначне елементе на макро плану и Хакслијев кинетички модел мишића на микро плану. Оригинални приступ распоређивања задатака на процесоре и графичке процесоре осигурава добар баланс оптерећења, што доводи до убрзања од преко два реда величине и високе скалабилности.

У другом делу се разматра употребна вредност HPC-а у оквиру теорије графова, а у сврху постављања основних структурних особина стабала са минималним ABC индексом. У сврху анализе огромне количине података, користе се складишни и рачунски ресурси на гриду, чиме се постиже убрзање од три реда величине.

Последњи део се бави WoBinGo софтверским оквиром за решавање захтевних оптимизационих проблема на HPC ресурсима. Оквир превазилази ограничења ранијих статичких инфраструктура заснованих на пилот пословима тако што омогућава еластичну употребу ресурса помоћу адаптивне алокације послова ограниченог животног века. Резултати указују да, упркос штедљивој употреби ресурса, добијамо значајно убрзање када се ради о оптимизацијама са скупим евалуацијама, као што је случај у области хидро-информатике и управљању тржишним ризиком.

Кључне речи: паралелно рачунарство, моделовање мишића, грид, рачунарство у облаку

References

Alba E, Tomassini M (2002). Parallelism and evolutionary algorithms, *IEEE Trans. Evol. Comput.* 6 pp. 443–462.

Cantú-Paz E, Goldberg DE (2000). Efficient parallel genetic algorithms: theory and practice, *Comput. Methods Appl. Mech. Engrg.* 186 pp. 221–238.

Chen J, Guo X (2011). Extreme atom-bond connectivity index of graphs, *MATCH Commun. Math. Comput. Chem.* 65 p. 713–722.

Das KC, Gutman I, Furtula B (2011). On atom-bond connectivity index, *Chem. Phys. Lett.* 511 p. 452–454.

Dorronsoro B, Arias D, Luna F, Nebro AJ, Alba E (2007). A grid-based hybrid cellular genetic algorithm for very large scale instances of the CVRP, in: *High Performance Computing & Simulation Conference*, HPCS, pp. 759–765.

Drenovak M, Ranković V, Ivanovic M, Urošević B, Jelic R (2016). Market Risk Management in a Post-Basel II Regulatory Environment, *European Journal of Operational Research*, Volume 257, Issue 3, 16 March 2017, pp. 1030–1044.

Duke T (1999). Molecular model of muscle contraction. *Proc. Natl. Acad. Sci. USA* 96:2770–2775.

Estrada E (2008). Atom-bond connectivity and the energetic of branched alkanes, *Chem. Phys. Lett.* 463 p. 422–425.

Estrada E, Torres L, Rodríguez L, Gutman I (1998). An atom-bond connectivity index: modelling the enthalpy of formation of alkanes, *Indian J. Chem.* 37A, p. 849–855.

Fernandez JW, Buist ML, Nickerson DP, and Hunter PJ (2005). Modelling the passive and nerve activated response of the rectus femoris muscle to a flexion loading: a finite element framework. *Med. Eng. Phys.*, vol. 27, no. 10, pp. 862–70.

Foster I (1995). Designing and Building Parallel Programs, *Addison-Wesley Reading*

Furtula B, Gutman I, Ivanović M, Vukičević D (2012). Computer search for trees with minimal ABC index, *Applied Mathematics and Computation* 219(2): 767-772

Goldberg DE (1989). Genetic Algorithms in Search, Optimization, and Machine Learning, *Addison Wesley*

Gutman I, Furtula B, Ivanovic M (2012a). Notes on trees with minimal atom-bond connectivity index, *MATCH Commun. Math. Comput. Chem.* 67 p. 467–482.

Gutman I, Tošovic J, Radenkovic S, Markovic S (2012). On atom-bond connectivity index and its chemical applicability, *Indian J. Chem.* 51A p. 690–694

Heidlauf T and Röhrle O (2013). Modeling the chemoelectromechanical behavior of skeletal muscle using the parallel open-source software library OpenCMISS, *Comput. Math. Methods Med.*, vol. 2013, p. 517287.

Hill AV (1938). The Heat of Shortening and the Dynamic Constants of Muscle, *Proc. R. Soc. B Biol. Sci.*, vol. 126, no. B, pp. 136–195.

Huxley AF (1957). Muscle structure and theories of contraction, *Prog. Biophys. Biophys. Chem.*, vol. 7, pp. 255–318.

Ivanovic M, Simic V, Stojanovic B, Kaplarevic-Malisic A, Marovic B (2015). Elastic grid resource provisioning with WoBinGO: A parallel framework for genetic algorithm based optimization, *Future Generation Computer Systems*, Vol. 42, 44–54.

Ivanovic M. Stojanovic B, Kaplarevic-Malisic A, Gilbert, Mijailovich S (2015). Distributed multi-scale muscle simulation in a hybrid MPI-CUDA computational environment, *SIMULATION: Transactions of The Society for Modeling and Simulation International*

Kojic M, Mijailovic S, and Zdravkovic N (1998). Modelling of muscle behaviour by the finite element method using Hill’s three-element model, *Int. J. Numer. Methods Eng.*, vol. 43, no. 5, pp. 941–953

Lister M (1960). The numerical solution of hyperbolic partial differential equations by the method of characteristics,”*Math. methods Digit. Comput*. vol. 1, pp. 165–179

Marović B, Potočnik M, Čukanović B (2001). Multi-application bag of jobs for interactive and on-demand computing, *Scalable Comput. Pract. Exp.* 10

McMahon TA (1984). *Muscles, reflexes and locomotion*. Princeton University Press, New Jersey.

Mijailovich SM, Fredberg JJ, and Butler JP (1996). On the theory of muscle contraction: filament extensibility and the development of isometric force and stiffness, *Biophys. J.*, vol. 71, no. 3, pp. 1475–84.

Munawar A, Wahib M, Munetomo M, Akama K (2008). A survey: genetic algorithms and the fast evolving world of parallel computing, in: 10th IEEE International *Conference on High Performance Computing and Communications*, HPCC’08, pp. 897–902.

O. Röhrle, J. B. Davidson, and A. J. Pullan (2012). “A physiologically based, multi-scale model of skeletal muscle structure and function”, *Front. Physiol.*, vol. 3, no. September, p. 358

Quinn MJ (2004). Parallel Programming in C with MPI and OpenMP, *McGraw-Hill*, New York

Razumova MV, Bukatina AE, and Campbell KB (1999). Stiffness-distortion sarcomere model for muscle simulation, *J. Appl. Physiol.*, vol. 87, no. 5, pp. 1861–76

Smith DA, Geeves MA, J. Sleep and Mijailovich SM (2007). Towards a unified theory of muscle contraction. I: Foundations. *Ann Biomed Eng.* Oct;36(10):1624-40.

Torelli A (1997). Study of a mathematical model for muscle contraction with deformable elements, *Rend. Sem. Mat. Univ. Politec. Torino*, vol. 55, pp. 241–271.

Zajac FE (1989). Muscle and tendon: properties, models, scaling, and application to biomechanics and motor control, *Critical reviews in biomedical engineering*, vol. 17, no. 4. pp. 359–411.