The 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications
21-23 September, 2017, Bucharest, Romania

Simulation of Large Neuronal Networks in Cloud and Grid with Graphics Processing Units

Oleksandr O. Sudakov [1,2], Andrii I. Cherederchuk [1], Volodymyr L. Maistrenko [1]
[1] Scientific Center for Medical and Biotechnical Research, NAS of Ukraine, Volodymyrska Str. 54, 01030 Kyiv, Ukraine, sudakov.oleksandr@gmail.com, http://nll.biomed.kiev.ua/
[2] Taras Shevchenko National University of Kyiv, Volodymyrska Str. 60, 01030 Kyiv, Ukraine, saa@univ.kiev.ua, http://rex.knu.ua/

Abstract—**Software for simulation of large networks of coupled non-linear oscillators in clusters, grids and clouds using graphical processing units (GPU) was designed, developed, tested and applied for scientific simulations. The software provides easy integration of new oscillators' models support, dynamic load distribution between hosts' central processing units (CPU) and several GPU devices. Different GPU devices provide speed-up of 12-50 compared to single core Intel Xeon, 2.4 GHz depending on GPU and job types. The software was efficiently applied for simulations of 3D networks with 10^7-10^8 oscillators described by Kuramoto-Sakaguchi model and new types of "chimera states" were discovered in such simulations.**

Keywords— graphics processing unit; CUDA; non-linear network dynamics; neuroscience; cluster; grid, cloud

I. INTRODUCTION

Research in neuroscience, various fields of physics, biology and other areas requires computing of dynamic processes in large networks of coupled oscillators [1, 2]. Integration of systems with large number of non-linear and non-locally coupled differential equations is required for such simulations. These computations are very memory and CPU resource consuming and also generate a large amount of data. High performance computing approaches like clusters and grids [3] were traditionally used for such computations [1, 2]. Conventional high performance computing approaches with batch job processing and data stage in/out are considered as not user friendly for most scientists. Thus many efforts were taken for development of web and other interfaces for clusters and grids to simplify usage of different computing applications [1, 4-6]. There are also trends to utilize clusters' and grids' resources as data [6, 7] and computing [8] back-ends for interactive desktop applications. Recently utilization of cloud resources for scientific computing [9] becomes very popular. Such utilization include running of virtual machines with users' applications as jobs in clusters and grids [10], utilization of containers for construction of specific scientific jobs' environment [11] and even creation of virtual high performance (HPC) clusters with preinstalled software [12]. Additional virtualization layers often reduce performance of conventional HPC applications and such applications should be ported to new hardware and software environments to achieve the best performance. Today several computing acceleration hardware technologies are available. Such technologies include graphics processing units (GPU) computing accelerators (General Purpose Graphic Processing Units, GPUGU). Computing accelerators can provide very large performance for cloud and even for a desktop environment. They are very attractive for end users because of low price. Thus porting of existing software for such hardware computing accelerators is an actual task. This task is not trivial because architectures of such accelerators significantly differ from each other and from conversional message passing distributed memory or multi-core shared memory approaches.

In present work we describe porting of the software for simulations in neuroscience for compatibility with GPU computing accelerators. Original software was developed by authors [1, 2]. The GPU-powered software can simultaneously use all GPUs and CPUs available at computing nodes, work with different CPU instructions sets in desktops, clusters, grids and clouds. Almost all features of original software except differential equations with delay are now supported on GPUs and significant performance increase was achieved. Described software was applied in Ukrainian National Grid[1] (UNG) [13] for investigations of chimera states (spatiotemporal patterns of coexisting coherence and incoherence) in three dimensional networks with 10^7-10^8 coupled oscillators described by Kuramoto-Sakaguchi model [14, 15].

II. COMPUTING SOFTWARE ARCHITECTURE

A. Software components

The described computing software is the extension of the original authors' software for simulation of large neuronal networks [1, 2]. It consists of three main parts:

[1] The work was supported within the Program for Scientific research, National Academy of Sciences of Ukraine "Grid Infrastructure and Grid Technologies for Science and Applications", 2014-2018.

initial data generation tools, parallel differential equations integrator and trajectory analysis tools.

Initial data generation tools are used for generation of all dynamical variables initial conditions for all network elements and adjacency matrix of links topology between network elements. These data files scale in size as $O(N)$ and $O(N^2)$ appropriately, where N is a number of network elements (neurons, oscillators, etc). For a large number of network elements (for instance 10^7) these data structures especially links adjacency matrix become very large and data compression is applied. Utilities xz for LZMA compression are used for initial conditions files. Special algorithm for links adjacency matrix compression was developed.

Integrator implements all supported models of network elements and provides integration of differential equations on parallel procession hardware based on initial data and configuration files. Integrator outputs trajectory of network dynamics that contains the state of all dynamic variable of network elements in different time moments. Trajectory has the same format as initial conditions file and is compressed with parallel xz utilities.

Data analysis tools read trajectory file and output some analysis results like filtering, statistics, plots, animations, visualizations, etc.

B. Integrator

Integrator performs integration of non-stiff, moderately stiff and stiff differential equations by three different numerical integration methods for supported models of oscillators' networks in the form:

$$\frac{d\vec{x}}{dt} = \vec{f}\left(\vec{x}\left(t - t_i\right), t, \vec{a}\right), \qquad (1)$$

where $\vec{x}$ - vector of oscillators' dynamical variables, t – time, $\vec{f}$ - right part of differential equation vector function that describes model and links topology, t_i - possible delays in time, $\vec{a}$ - vector of model's parameters.

All supported models of oscillators are hard-coded into integrator for maximum performance. Models' parameters, links topologies and mapping of models to neurons may be changed at runtime via configuration files. New model may be easily added into the software by redefinition of the base models' C++ class. Model's class should implement virtual functions for computing of function $\vec{f}$ for given dynamic variables $\vec{x}$, time moment t and links topology, reading of configuration parameters and optionally input and output of integration results. All other job is done by the integrator itself.

At startup integrator reads configuration file with specification which models to use. For each model the default parameters' values are specified. Then several groups of neurons may be defined. Each group corresponds to the model and may redefine the models default parameters. Matrix of links parameters between groups should be specified. Then a total neurons number is given and an array for correspondence of neurons to groups is specified. Also configuration file contains parameters of numerical integration algorithm, additional data output, etc. When network elements configuration is known the links topology adjacency matrix and initial state are read. To read the initial state the input read virtual functions are called for each neuron. Finally the specified numerical integration procedure is called. Within this procedure the appropriate functions $\vec{f}$ are computed for each neuron. Integrator outputs the results calling the output virtual functions.

C. Links topology adjacency matrix compression

Coupling of oscillators in the network is specified via the adjacency matrix, A_{ij}.

$$A_{ij} = \begin{cases} 1, & \text{neuron } i \text{ influences neuron } j \\ 0, & \text{neuron } i \text{ not influences neuron } j \end{cases}$$

The size of this matrix scales as $O(N^2)$. For large networks (10^5-10^8 elements) the size of this matrix may reach terabytes. This matrix is required to be stored in memory while computing the network dynamics. Thus compression of this matrix is necessary. Compression and decompression algorithms are required to have high speed and compression ratio. General purpose algorithms do not meet the requirements and a special compression algorithm was suggested.

This algorithm utilizes the special structure of adjacency matrix for most network types and belongs to entropy type algorithms. For each neuron the indices of neurons connected to it are stored in an array. This array of the first neuron is called the *common vector*. The common vector is stored uncompressed. If network is regular this vector for other neurons should have the same number of elements and all elements would be shifted by the neuron's index. To perform compression of links to some neuron all the indices of neurons connected to it are decremented by the neuron index and the XOR operation with common vector is performed. For regular network all such differences would be zero. If the network is slightly irregular the differences would be non-zero but many differences would be equal. All unique differences are stored in *differences table*. For each neuron the pointer to its difference is stored in a *pointer array*. Decompression is performed by computing XOR of difference and common array and adding of neuron's index.

Proposed algorithm provides compression ratios more then 10^3 and reduces adjacency matrix size in memory from terabytes to several megabytes. Decompression may be performed in stream using iterator concept to get the index of the next connected neuron on demand.

D. Optimizations and parallelizing

Several optimizations were suggested to achieve the maximum scalability and performance of the software.

The most resources consuming tool is integrator and the optimizations are almost related to it.

Depending on the network model each neuron in each model is defined to have some number of local, exchange and cache dynamic variables. The memory used for neurons' dynamic data is divided into three continuous regions: local variables, exchange variables and cache variables of all neurons. Exchange and cache variables are shared between neurons and local variables are not. Cache variables are not used for continuous dynamics output while exchange variables are used. Division of memory into such areas simplifies parallelizing.

The most resource consuming part of differential equations (1) integration is computing of right hand function, $\vec{f}$. This computation requires $O(NM)$ operations, where N is a number of neurons (oscillators), and M is a number of each neuron's links. Numerical integration and output require $O(N)$ operations. Thus parallelizing is the most applicable for computations of $\vec{f}$. Depending on parallel computer model parallelizing should be performed in different ways. Thus different drivers for computing of $\vec{f}$ on different parallel computer models are defined. These drivers are called by numerical integrator.

In shared memory system all memory regions are shared so each neuron may be computed in parallel to others. On symmetric multiprocessors (SMP) procedure level parallelism is applied using OpenMP directives. SMP parallelizing is very efficient and provides speedup close to the number of CPU cores in all tested environments. The maximum number of OpenMP cores tested is 24.

On strongly coupled distributed memory systems message passing interface (MPI) parallelizing is applied. All computing nodes contain all copies of neurons and topology compression structures. Each node computes function $\vec{f}$ for its part of neurons. At first all nodes perform MPI_Allgather call for cache and exchange memory regions. Then each node computes function $\vec{f}$ in parallel and performs results exchange. Output of results is performed by the root process which gathers all data. For MPI parallelizing all neurons are distributed uniformly between nodes. The efficiency of MPI parallelizing is significantly lower then the OpenMP one and depends on the implementation of the MPI_Allgather call. The speed-up values of MPI version is about 4 [2].

For computing clusters and geographically distributed grid systems the OpenMP version is the most efficient. Multiple copies of application with different initial data files are submitted for single program multiple data (SPMD) execution in such systems.

III. Porting to Graphic Processing Units

A. GPU Architecture

In present work NVIDIA computing GPU accelerators are used. These accelerators contain few multiprocessors. Each multiprocessor executes large number of parallel threads. Several threads (warp) execute the same machine instruction with different data elements. Each thread can access relatively slow shared global memory, have registers set and slow local memory area. Threads' block consists of several warps. Threads of the same block can use fast shared memory area. Shared memory of the block can be divided between threads to increase the memory per thread. There are also several memory caches available depending on GPU's computing capability. GPU devices communicate with the host via PCI bus. Host and GPUs forms the distributed memory system. Host applications and GPU kernels execute different machine instructions incompatible with each other. Memory regions of GPU and host are also different. Data may be transferred between host's and device's memories via PCI bus by the Application Programmer Interface (API). In present work we use NVIDIA Compute Unified Device Architecture (CUDA) API. Depending on the GPU device computing capability, version of GPU driver and CUDA API different communication interfaces available.

Performance of GPU significantly depends on the task's parallelism, optimizations, communication overhead, etc. In many cases GPU significantly outperforms host CPU. In desktop and cloud environments usually only GPUs are used for computations while host's CPUs perform only control function. Nevertheless in computing clusters, grids and HPC clouds computing power of nodes with a large number of CPUs can be comparable with GPU performance. Thus scheduling of computations and communications for CPUs and GPUs is required to achieve the best performance.

B. Software architecture changes

Host's part of software architecture is almost unchanged. Device's part contains copies of some host's data structures and some host's code. The data structures are needed to be copied to GPU include system configuration, list of supported neuron's models, mapping of neurons to groups, links topology pointer array, differences table and common vector. Code that performs computing of function $\vec{f}$ (1) and links topology adjacency matrix decompression are needed to be ported to GPU. The new subsystem added to host's part is scheduler that performs load balancing between CPUs and GPUs.

A large part of host's code is not necessary on GPU. It includes reading of configuration files, numerical integrators and output functions.

C. Code

All computing code of the original software was written in C++. CUDA C++ compiler supports almost all C++99 features required for software porting. Thus porting of the most code was trivial. The difficulties arise when some host's library functions need to be called by GPU device code. In some cases CUDA provides library calls similar to the host's ones. Unfortunately many library functions are unsupported on GPU device. These calls include Standard Template Library (STL) containers and algorithms used in original code. Additional template wrappers were developed to enable support of std::vector class on device.

The only code was not ported is the code for support of models with delay. This code requires large data structures to be concurrently accessed by host and device and is related to numerical integration libraries. Support of models with delays is in TODO list.

D. Data structures

The most difficult thing related to GPU computations is copying of complex data structures with pointers to GPU. The problem is even more challengeable if objects with virtual methods created in runtime should be copied from host to device.

CUDA provides several APIs to copy data structures from host to device and vice versa: managed memory, zero-copy pinned memory, and memory allocation and copying functions. Managed memory is the most convenient API for CPU/GPU data copying. It provides mapping of host's memory addresses to the same memory addresses in GPU memory. Unfortunately on conventional NVIDIA GPUs this approach is slow, mapped memory cannot be concurrently accessed by the host and device and amount of allocated mapped memory is limited. GPU architecture GP100 supported by CUDA-8 significantly extends the capabilities of managed memory. It is faster, can be concurrently accessed by GPUs and CPUs and its size limit is significantly extended. Nevertheless shared GPUs/CPUs managed memory with transparent copying is slower then the dedicated host or GPU memory and require expensive synchronization for concurrent access. For compatibility and performance reasons managed memory usage is limited it this software. Zero-copy pinned memory approach provide mapping of host's memory to device, but memory addresses on host and device are different. This API is fast. Memory allocation and copying API provides allocation of memory on host/device and data coping to/from host/device while called on host.

All these APIs were used for porting to GPU. Managed memory is used only for creation of temporary structures during initial data copying. Most data structures were copied by memory allocation and copying API. Special template functions were designed for coping of std::vector structures used for neurons array, groups structures, common vector, differences table and pointer array. Arrays of numbers are copied as is while arrays of pointers are copied by allocation of data structures in device memory, returning of pointers for filling of temporary data structures and copying these structures again.

Coping of supported models that are configured in runtime is performed as follows. Initially the array of all supported modes objects is created on device. The memory region with runtime models in mapped to device using zero-copy memory. All host models have unique numbers. Device model is selected from the array using host's model number as index. For this model the virtual cloning method is called on device passing the host's model region as parameter. This virtual method is aware about the model type and all host's data structures are being copied to device correctly. The cloned model's virtual methods for computing of functions $\vec{f}$ (1) are also setup automatically by the compiler.

E. Performance optimizations

Scheduler was added to the original software architecture for load distribution between host and device. The scheduler divides all neurons into the portions for computations on the host and each GPU device found in the system. Computations of functions $\vec{f}$ (1) are performed on CPUs and GPUs in parallel. Host's portion is computed in parallel on all available CPUs by OpenMP parallelizing. Directives for cycles paralleling are used. The first several OpenMP threads start GPUs' kernels and continue procession of the host's neurons. Kernels for all available GPUs are started in parallel.

Execution on GPU requires all initial data to be sent from host to device. These data in provided by the numerical integration procedure. Results are also need to be copied back. Communication between the host and device significantly affects performance of the software. Overlapping between computations and data transfer is used to increase performance. All GPU's neurons are divided into chunks. At first all exchange and cache data is sent to GPU. After that local variable for the chunks are sent to GPU the kernel for these chunks are started and the coping of results for the chunks are scheduled in parallel. CUDA streams API is used for asynchronous data coping and kernels execution. CUDA events API is used for synchronization. If the number of neurons exceeds the number of concurrent threads on GPU (usually 10^2-10^4) such scheduling overlaps data transfer and kernels execution.

When all host's neurons are computed the scheduler checks if all GPUs are finished their part and updates the neurons portions for host and GPUs to make computation time equal on all devices.

F. Portability

In grids and clouds different hardware and software architectures may be available and the computing

software should support all of them. The portability is achieved by providing the binaries compiled for different hardware architectures. All necessary libraries were linked statically as much as possible to reduce the number of necessary extern libraries. On startup application tries to determine the CPU's and GPU's hardware and executes the binary compatible with current hardware. If CUDA is unsupported the CPU only version is executed.

IV. Testing and Applications

Testing of the software was performed to obtain on GPU and in mixed CPUs/GPUs environments the same results we got previously on CPUs. Then the performance optimization was performed for different CPU and GPU architectures we can access and a production application of the software was performed.

The massive computations were performed in Ukrainian grid infrastructure [13] on clusters of Scientific Center for Medical and Biotechnical Research, NAS of Ukraine [chimera.biomed.kiev.ua] and Information and computer center National Taras Shevchenko University of Kyiv [cluster.univ.kiev.ua]. The first cluster is heterogeneous. It contains 3 nodes with 16 hyper-threading (HT) CPU cores Intel Xeon E5620, frequency 2.4 GHz; 3 nodes with 24 HT CPU cores Intel Xeon E2620, frequency 2,6 GHz; 3 nodes with 12 HT CPU cores Intel Xeon E5620, frequency 2.4 GHz and 1 node with 12 HT cores E5-2603, frequency 1.7 GHz. The second cluster is homogeneous. It has 6 nodes with 24 HT CPU cores Intel Xeon E2620, frequency 2,6 GHz. Cluster chimera biomed.kiev.ua has NVIDIA Tesla K40 GPU installed at E5-2603 node and GPU NVIDIA GeForce GT640 at E5620 node. The simulation software was compiled with gcc-4.9.2 compilers using CUDA-7.5 libraries with optimization to all CPU types mentioned above.

The massive computations for performance testing were performed using batch jobs execution in grid. About 3000 trajectories were computed in grid using OpenMP, GPUs and mixed CPUs/GPUs environments on the clusters described above. Several runs of the software were performed in interactive mode at the cluster node of Institute for Cybernetics, NAS of Ukraine that has two NVIDIA Tesla K20 GPUs installed and in EGI Federated Cloud at the virtual machine with 2 NVIDIA Tesla K40 GPUs installed. Testing at desktop and cloud was performed to check compatibility and multiple GPUs support and was not used for performance measurement.

The testing computing task was simulation of large 3D networks described by Kuramoto-Sakaguchi model for 100x100x100, 200x200x200 and 400x400x400 oscillators. The model is described by the equation:

$$\frac{d\varphi_{i,j,k}}{dt} = \frac{3}{4\pi R^3}\sum_{\Omega}\sin\left(\varphi_{i',j',k'} - \varphi_{i,j,k} - \alpha\right),$$

where φ - phase of the oscillator; i, i', j, j', k, k' - oscillators' numbers on 3D network; region Ω is given by the equation $(i-i')^2 + (j-j')^2 + (k-k')^2 < R^2$; R - coupling radius; α - phase shift; $r = \frac{R}{N}$ - relative coupling radius; N - number of oscillators in each dimension.

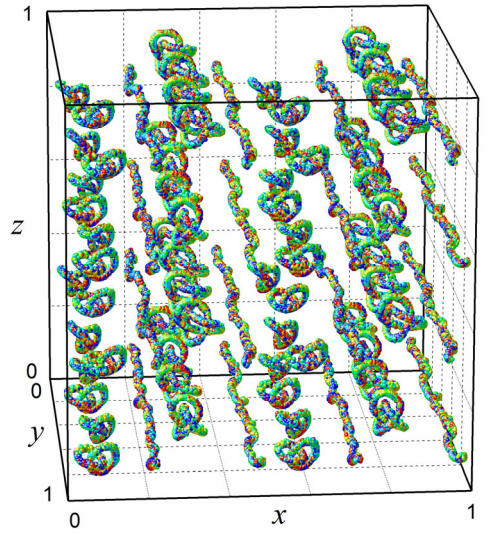

Figure 1. Hybrid chimera state. *N = 400. x = i/N, y = j/N, z = k/N.*

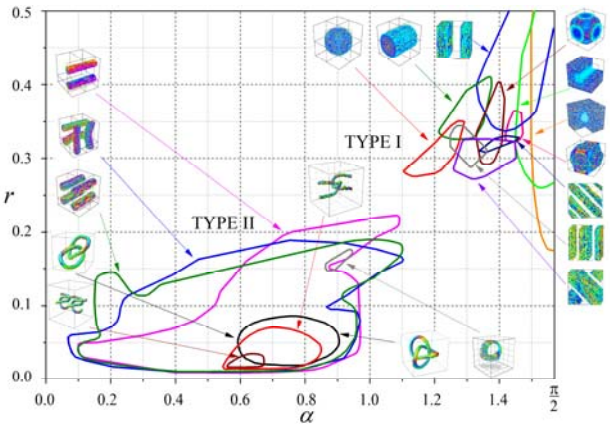

Figure 2. Regions of different chimera states. Snapshots of the chimera states are shown in the inserts. *N = 100.*

The main goal was to find new types of stationary states with coherency and incoherency in dynamics described by this model. The example of a new discovered hybrid chimera state is presented in Fig. 1. Each pixel corresponds to the single neuron. Frequencies of the coherent oscillators are displayed as transparent. Incoherent oscillators are displayed in color. The total number of coupled oscillators and differential equations were changed from 10^6 to about 10^8 for different runs. The whole experiment provided the possibility to find the regions where different chimera states exist in parameters space (α, r). The results are shown in Fig. 2 [14, 15]. Thus the performance measurement results include cases for

different parameters' values and oscillators' number as well as computing times on GPUs, CPUs and in mixed CPUs/GPU environment.

The aggregated performance measurement results are presented in Fig. 3. Performance of described software on GPU in CPU cores, P, characterizes how many baseline CPU cores are required to achieve the same performance as GPU provides:

$$P = \frac{T_{CPUs} * n_{CPUs}}{T_{GPU}} \frac{T_{CPU}}{T_{BL}} = \frac{N_{GPU}}{N_{CPUs}} \frac{T_{CPU}}{T_{BL}},$$

where T_{GPU} - computing time on GPU, T_{CPUs} - computing time on n_{CPUs} CPUs, T_{CPU} / T_{BL} - relative computing time on single CPU to baseline (BL) CPU. For mixed GPU/CPUs environment the value N_{GPU} / N_{CPUs} is the ratio of oscillators distributed to GPU and CPUs. The baseline for this measurement is a single HT core of Intel Xeon E5620 with frequency 2,4 GHz. Original software computes such jobs about a week on 16 baseline cores. It is easy to see from Fig. 3 that performance on GPU depends mostly on GPU type but not on the number of neurons, parameters' values and CPUs number because data points are located in two narrow clusters. It also means that GPU efficiently adds its performance to CPUs.

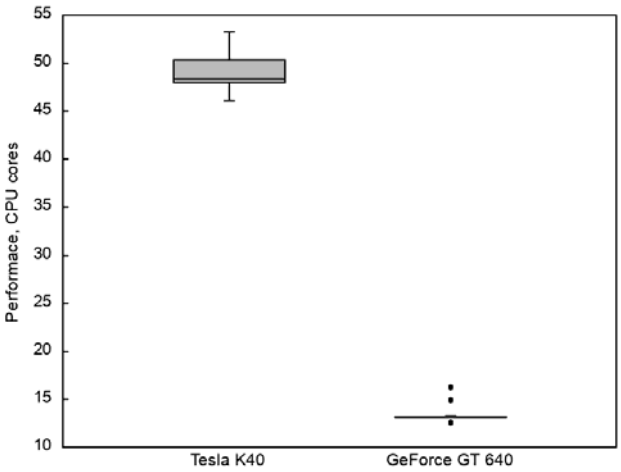

Figure 3. Performance of GPU versus CPU.

V. Conclusions

The software for computing of nonlinear dynamics on networks powered by GPU provides the main functionally of the original software and proved to be efficient on desktop, clusters, grids and clouds in GPU, CPU and mixed environment.

The performance of the software on GPU powered computing node with 16 cores is 2-4 times higher then its' performance on same node's CPUs only.

GPU outperforms general purpose single core CPU in 12-50 times depending on CPU's and GPU's type for computations with proposed software.

Acknowledgment

We thank Ukrainian Grid Infrastructure for providing the computing cluster resources and the parallel and distributed software.

References

[1] Andrii Salnikov, Roman Levchenko, Oleksandr Sudakov Integrated Grid Environment for Massive Distributed Computing in Neuroscience // Proc. 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications 15-17 September 2011, Prague, Czech Republic . - p. 198- 202

[2] Levchenko, R. I.; Sudakov, O. O.; Maistrenko, Yu L. Parallel software for modeling complex dynamics of large neuronal networks. In: Proc. 17th International Workshop on Nonlinear Dynamics of Electronic Systems, Rapperswil, Switzerland. 2009. p. 34-37.

[3] I. Foster and C.Kesselman, The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1999

[4] Yuriy O. Koval, Hlib O. Mendrul , Andrii O. Salnikov, Ievgen A. Sliusar, Olexandr O. Sudakov // Interactive Dynamical Visualization of Big Data Arrays in Grid. Proc. 8-th IEEE International Conference IDAACS 2015, 24-26 September 2015, Warsaw, Poland. , p. 153-156

[5] Salnikov, A.O., Sliusar, I.A., Sudakov, O.O., Savytskyi, O.V., Kornelyuk, A.I. MolDynGrid virtual laboratory as a part of Ukrainian Academic Grid infrastructure // Proc. 5th IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS'2009., pp. 237-240

[6] Savytskyi, O.V., Sliusar, I.A., Yesylevskyy, S.O., Stirenko, S.G., Kornelyuk, A.I. Integrated tools for molecular dynamics simulation data analysis in the MolDynGrid virtual laboratory // Proc. 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS'2011, pp. 208-211

[7] Sudakov, O., Kononov, M., Sliusar, I., Salnikov, A. User clients for working with medical images in Ukrainian Grid infrastructure // Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems, IDAACS 2013, 2, 6663016, pp. 705-709

[8] Cyrille Rossant. Learning IPython for Interactive Computing and Data Visualization. Packt Publishing. – 2015 – 175 p.

[9] WANG, Lizhe, et al. Scientific cloud computing: Early definition and experience. In: High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on. Ieee, 2008. p. 825-830.

[10] Boretskyi, O., Salnikov, A., Sliusar, I., Sudakov, O., Boyko, Y. Rainbow framework: Running virtual machines on demand as a grid jobs. // Proc. IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, IDAACS 2015, pp. 972-976

[11] Walker, E., Gardner, J. P., Litvin, V., & Turner, E. L. (2007). Personal adaptive clusters as containers for scientific jobs. Cluster Computing, 10(3), 339-350.

[12] Amazon Web Services https://aws.amazon.com/

[13] Zynovyev, M., Svistunov, S., Sudakov, O., & Boyko, Y. (2007, September). Ukrainian Grid infrastructure: practical experience. // *4-th International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications.* Dortmund, Germany, September 6-8, 2007 pp. 237-240

[14] Maistrenko, Y., Sudakov, O., Osiv, O., Maistrenko, V. Chimera states in three dimensions // New Journal of Physics, 17 (7), 073037.

[15] Volodymyr Maistrenko, Oleksandr Sudakov, Oleksiy Osiv, Yuri Maistrenko. Multiple scroll wave chimera states https://arxiv.org/abs/1702.00161