The Survey of Object-Oriented Software Programming Language from a Heterogeneous Cluster Programming Viewpoint

Tomas Brandejsky * and Vaclav Hrbek

Faculty of Electrical Engineering and Informatics, University of Pardubice,
Studentská 95, 53210 Pardubice, Czech Republic
tomas.brandejsky@upce.cz
http://www.upce.cz

Abstract. In this paper, the problem of programming language selection is presented from the position of large cluster with heterogeneous accelerators programming in the situations when it is need to apply object-oriented programming like in the case of heterogeneous multi-agent simulations or large data modelling using memetic algorithms. This work was inspired by experience with the Chapel language obtained during complicated conversion of hybrid evolutionary algorithm GPAes from a single node OpenMP C++ implementation onto HPC cluster with nodes equipped by both CPU and GPGPU.
The paper consists of discussion of many approaches to parallel programming including not only traditional ways such as OpenMP, MPI and Cuda and their combinations, but also modern extensions of C/C++ as OpenACC, Silk and CYCL. Emerging languages as Chapel and Julia are discussed too. The work concludes with an evaluation of the real state of parallel object-oriented programming on heterogeneous node HPC clusters.

Keywords: Object-Oriented Programming, C++, OpenMP, MPI, SICL, Chapel, Julia, Parallel Programming, HPC cluster, Heterogeneous System, Memetic Algorithm

1 Introduction

The total processor computing power continually increases, while the increase in single thread performance of the processor is slowing down. For many years, their development detract from power increase by increase of parallelism, because increase of processor frequency is coupled with quadratic increase of power consumption. In previous decades, many kinds of parallelism were established, especially

- pipelining
- superscalar (running more independent instruction in the same cycle, together with out-of-order execution)

* Corresponding author

- simd (e.g. vector instructions)
- etc.

With the incoming new hybrid architectures combining multi-thread many core CPUs with accelerators, it is more and more significant to parallelize existing code. While in some cases, this way is relatively easy (e.g. some kinds of numerical calculations), in others it can be difficult and hard to automate. Problems frequently occur on clusters, heterogeneous architectures, non uniform memory architecture systems, etc.

1.1 Background of the Publication

Within the frame of memetic algorithm research, the aim to implement GPAes [1] algorithm on heterogeneous computing cluster was formulated. On the one side, new computing accelerators like General-Purpose Graphic Processing Units (GPGPUs) from Intel, Nvidia and AMD), Field-Programmable Gate Arrays (FPGAs) and also vector processors as NEC Aurora are available now. On other side, memetic algorithms need a lot of computing power. E.g., following Flinn's taxonomy of processors [2], CPUs are now of Multiple Instruction Multiple Data (MIMD) architecture, but GPGPUs has Single Instruction Multiple Data (SIMD) one. Thus, a suitable programming language supporting many kinds of HW and allowing high programming productivity was searched.

Cluster or even HPC implementations of hybrid and memetic algorithms are essential for their computing complexity in the case of non-trivial training data size. The memetic algorithm GPAes consists of two parts - Genetic Programming Algorithm [3] and optimizer which can be represented e.g. by evolutionary strategy [4]. It is easier to process GPA on the CPU, while GPGPUs are better for optimization tasks. We can remember the pioneering works of Longdon. He used GPGPUs for fitness evaluation of structures produced by GPA and implemented two different approaches. The first was based on generating of source code for all evaluated individuals, its compilation and running on the GPGPU; the second was based on the interpret implementation [6]. The work [7] distinguishes 3 different approaches to implementation of GPA on GPGPUs from the viewpoint of computing kernel management. These discussions are oriented to standard GPA implementations on GPGPUs, not to memetic algorithms adding optimization of parameters (constants) of each individual in each population by multiple instances of nested optimizer. The implementation of many instances of nested optimizer increase the need for use of object-oriented programming (or difficult code and coding without it).

The complex structure of memetic algorithms and concluding computing complexity are the main reason for search of the way how to implement them on heterogeneous clusters containing not only CPUs, but also computing accelerators like GPGPUs.

The memetic GPAes algorithm implementation uses the following data structures common for both GPA and ES part. In the case of heterogeneous implementations they must be present on both sides and synchronized. These structures are object arrays (representing array of optimizers or array of head nodes of trees

representing individuals); tree structures of objects representing individuals and standard arrays containing fitness values and magnitudes of individual parameters (constants). If the interpret it used, there is also a need to store programs representing individuals as their alternative representations are processed by the interpret.

2 Hybrid Cluster Programming State of the Art

Together with the increase of accessible computing power, the improvement of many computationally extremely expensive parts of science, and applications are evolving nowadays, especially multi-physics simulations, to which implementations are now available tools suitable. Models described by extremely large sets of Ordinary Differential Equations (ODE) find application in military, nuclear science and also in industry. But, it is the question, how many new computing paradigms are prepared for the implementation of models based on models other than on solving of large sets of differential equations. For example, these alternative models are represented by Artificial Intelligence (AI) models which are not based on Artificial Neural Networks (ANNs), agent-oriented simulation models, etc., as well as the area of data analytic (if it is not based on application of ANNs or statistic models).

Nowadays, we can observe intensive development of many different programming approaches and tools to simplify and thus speed up parallel program development for newly incoming heterogeneous computing systems, for increasing of programmers work efficiency in their programming. As heterogeneous computing systems are mainly labeled the computing clusters where standard compute nodes with CPU and local memory (thus Non-Uniform Memory Architecture systems) are extended by another computing accelerators, most frequently GPGPUs with separate local memory or FPGAs.

Data analytic and AI commonly may benefit from evolutionary algorithm application and especially from the memetic algorithm capability to provide fine symbolic regression. Also here, the problem is their computing expensiveness (the computing complexity of these algorithms) asking the need for implementation on large computing systems.

In the past, two basic architectures of parallel program design have expanded in the past. The Message Passing Interface (MPI) is especially suitable for clusters composed of nodes interconnected by a data network; but it is capable to manage parallel programs on multi-core and multi-processor nodes too. The other one, OpenMP (Open Multi Processing) is suitable for symmetric multiprocessor systems (and multi- or many- core processors) working with common shared memory. It allows one of the easiest programming but is constrained to a single node. In the past few years, clusters composed of hybrid nodes have begun to occur. These nodes do not contain only CPUs, but also computing accelerators specialized to some class of computing (typically vector ones), which contain it own separated memory like GPGPUs. These accelerators are highly

efficient in some classes of tasks, like numerical solving systems of equations, but their programming brings different concepts.

2.1 MPI

MPI is now in version 4.0. Originally it was based on (NUMA) distributed memory model; since the version 3 it introduces another approach to hybrid programming that uses the new MPI Shared Memory (SHM) model, see [12] and [13] introducing new basic principles.

For limited support of accelerator programming in the previous versions of OpenMP and lack of this support in MPI together with MPI programming difficulty and efficiency problems on multi-core processors, programmers still use complicated combinations MPI + CUDA, MPI + OpenMP, or even MPI + OpenMP + CUDA, MPI + OpenACC. The languages/techniques CUDA and OpenACC will be discussed below.

2.2 OpenMP

OpenMP is based on for-join model and parallelization of cycles, but the OpenMP standard 4.0 and later versions, have introduced pragmas for C, C++, and FORTRAN programming languages to offload work on general purpose GPUs. It is possible to find information about the usage of OpenMP and GPU programming in the OpenMP specifications [8]. The following papers [9], [10], [11] explain the usage of GPU offloading pragmas. Problem of this application is in it's strong similarity to CUDA and OpenAcc because accelerator programming in OpenMP is also based on defining of compute kernels, parallelization of cycles. This style is similar to the frequently used combination OpenMP+CUDA. Worse is the need to solve consistency of data in CPU and GPGPU memories manually.

2.3 OpenACC

OpenACC is an alternative to CUDA. OpenACC was defined by a broader consortium than CUDA, but its influence is significant and also OpenACC is based on C/C++ language and computer kernels, but it is a little higher level and hides some implementation details. It is available on both NVidia and AMD GPGPUs. The application limits are similar to CUDA.

2.4 CUDA

CUDA (Compute Unified Device Architecture) means both HW architecture and programming style. It was developed by NVidia company. The original support for Fortran and C was extended to C++ and OpenACC. Now, many programming languages and specialized tools support programming of CUDA devices. The main problem of CUDA programming is given by its origin in graphic devices. GPGPUs are strongly parallel; their HW architecture is complicated. They

contain many different memories (local, global, texture ones etc.). And CUDA does not hide this HW structure to support highly efficient programming. CUDA also allows description of host code (code running on a host system, e.g. PC), but does not go far and does not allow to describe parallel computing on cluster level. It is possible to describe programs running on more graphic devices within the single node, but limitation to single node remains.

Simply talking, while a modern CPU is a multi-core/multi-thread processor, GPGPU is vector one - there are many threads, but they process the same instructions (in CUDA within so-called block). The basic programming concept in CUDA is kernel function. These functions are executed in parallel as block of threads. On CPU there is equivalent on some processors in the form of vector instructions. The use of vector (also called multimedia) instructions sometimes brings hardly predictable compute time on standard CPUs, as it will be discussed later.

CUDA as well as OpenMP, OpenACC, etc. offers concept of compute kernels to simplify pluralization of cycles. This fact limits its application in areas where pure multi-threading is required. Because GPGPU has its own memory, its use brings the need to solve data transfers between CPU's and GPGPU's RAM. Even if the newest versions of CUDA and NVidia's GPGPU allow direct access to host computer memory, this access is significantly slower than working with the device one. CUDA also allows to process complex structures, but it does not solve their synchronization between host and device memories implicitly. They must be organized by a programmer. This fact increases code complexity (and thus the probability of error occurrence). OOP on CUDA-like HW is possible since 2020, while on the host it was allowed since CUDA origin, see [14], [15].

The remaining problem of OOP code on both host and CUDA accelerator is the need of two instances of object - the first on the host and the second on the device not only for the existence of two different memory spaces (this problem can be solved by unified memory), but also for different addresses of virtual methods for host and device. Their dual existence is caused by different machine code for CPU and GPGPU. See also [16]. In the case of memetic algorithm implementation it is need to distribute not only a data with static structure, but also trees (graphs) of objects - instances of classes, which may on the different subsystems (e.g. CPUs versus GPGPUs) different code, thus different pointers from virtual method tables, etc.

As it is mentioned above, standard Intel and AMD processors also offer vector instructions. It is possible to compile code using them, e.g. the option -fopenmp-simd of the GNU C++ compiler. When this possibility was tested on GPAes code, the resulting code was significantly slower, but such a result was expectable. Compiler options influence the whole code, but GPAes consist of two parts. While for es optimizer the vector instructions can bring advance, for operations with tree-like representation of genes in GPA part, the vector processing brings decrease of execution speed. The use of GNU C++ with option -fopenmp-simd brings also compatibility problems that not all functionalities of OpenMP library

are in that moment available. For example, there is not available omp_get_wtime function to measure run time.

It is possible to conclude that "classic" methods of heterogeneous SW development represented by combinations of MPI, OpenMP, CUDA, and eventually OpenACC discussed above, are constrained by the need of non-homogeneous programming model (e.g., the use of combinations MPI+CUDA, OpenMP+MPI+Cuda etc.). This style of programming is difficult for a combination of diverse programming paradigms, but it is frequently used on large (and heterogeneous) clusters now.

3 Modern Languages and Derivatives of C/C++

In the last years these problems have caused formulations of new concepts of parallel software development occur and they stand at the origin of new extensions of standard languages (C++, Java) or even new languages (Chapel, Julia).

Cilk, Cilk ++ and Cilk Plus were developed as general purpose languages for concurrent/multi-threaded parallel programming on MIT Laboratory For Computer Science. They are based on standard structured (procedural) programming and the basic parallel concept is similarly like in OpenMP fork—join. They extend C/C ++ by three constructs cilk_spawn, cilk_sync and cilk_for (definition of parallel thread, result synchronization and parallel loop). But these extensions to C/C++ does not offer direct support of accelerators.

3.1 Sycl

C/C++ language-based extension (or rather abstraction layer) Sycl, whose standard is developed by Kronos group (now in the version Sycl2020) exists in many implementations including Intel oneAPI DPC++. It also uses compute kernel model (like CUDA), but it solves memory sharing using accessors objects. This model allows to write versatile code capable to run on the wide spectrum of HW, but also this code is not pure original C/C++. Sycl is not a part of GNU compiler collection, but on the pages [17] it is possible to find free accessible implementation of oneAPI DPC++ by Intel including Sycl. This implementation is based on LLVM free compiler architecture. Official support of NVidia and AMD GPGPUs is not, but it is possible using hipSYCL [18].

The Intel oneAPI DPC++ also contains OpenMP and especially MPI. This fact only concludes, that neither Sycl is capable to describe parallelism on the level of heterogeneous clusters; it is limited to single heterogeneous node programming.

Also, Sycl will require the similar effort like previously described combinations based on CUDA or OpenACC, because Sycl also requires manual solving of complex structure consistence.

After this review of Cycl there was remaining interesting group of new languages consisting of Chapel, X10, and Julia.

3.2 Chapel and X10

Because in this year the information about development of NVidia and AMD GPGPUs support in programming language Chapel [19] were publicized, this language become the main candidate for hybrid cluster implementation of GPAes code. At this moment, this support is still under construction, but it's hopeful for the future.

This language was formed within the above cited project supported by Darpa. In this project also the second language origins. X10 proposed by IBM. While Chapel was really a new language, X10 is Java-based. X10 is a programming language being developed by IBM at the Thomas J. Watson Research Center as part of above mentioned the Productive, Easy-to-use, Reliable Computing System (PERCS) project funded by DARPA's High Productivity Computing Systems (HPCS) program. X10 project in this moment seems to be stopped; the last upgrade was in 2019-01-07 [20]. Both languages (Chapel and X10) are from the so-called PGAS languages family. PGAS is an acronym for Partitioned Global Array Space [21]. Now there can be observed effort to include into this model also heterogeneous systems - clusters with nodes consisting of combination of CPUs and accelerators, as on the newest (and the most powerful) supercomputers.

3.3 Chapel - practical experience

With the work on the transformation of GPAes project into the Chapel language, strong problems were found. These problems are not in the support of cluster computing, but in the actual state of OOP implementation incompatible to C++ one. Even if the language supports OOP, there is no possibility to use generic objects (polymorphism), especially arrays of generic objects and structures with more complex relations between them; it means structures referencing generic class objects. This unexpected step was done for inconsistency in the automatic garbage collector work. Without this ability accessible in other OOP languages, it is not possible at this moment to implement effective work with array of genes in the GPA algorithm and thus no memetic algorithm with nested optimizer, as GPAes. This problem in the Chapel language at this moment also affects such data structure implementations, as the implementation of stack, or join list.

This current state of Chapel implementation, non-looking to promising implementation of GPGPU support (based on compute kernels as in CUDA, e.g.) does not allow to use Chapel for reasoned purpose.

3.4 Julia

The Julia language was reasoned for its ability to describe parallel programs using OOP and functional programming. Practical problem is given by fact, that Julia uses Java Virtual Machine (JVM) as e.g. X10 and thus at this moment cannot work on GPGPUs, because instruction set of GPGPUs does not allow its work.

4 Conclusion

Presented paper points to the fact that if the solved problem tends to implementation of graph (or tree) of objects, forest of objects, array of graphs, etc., till now, there is no applicable progress in programming languages suitable to hybrid cluster programming. The situation still is not improving the standard approach based on combinations MPI+CUDA or MPI+OpenMP+CUDA used for many years with the known difficulties.

There are other approaches to solving the problem of hybrid single node programming as OpenACC or Sycl, but their use on clusters requires simultaneous application of MPI. On the opposite, there are languages optimized for problems where data are too big to fit to single node, to be processed on multiple nodes as Chapel or Julia, but there are another problems of accelerator programming (on heterogeneous nodes), and in the case of Chapel there are also still unsolved problems with OOP implementation.

Acknowledgments. The work was supported from ERDF/ESF " Cooperation in Applied Research between the University of Pardubice and companies, in the Field of Positioning, Detection and Simulation Technology for Transport Systems (PosiTrans)" (No. CZ.02.1.01/0.0/0.0/17_049/0008394).

5 The References Section

References

1. Brandejsky, T., Zelinka, I.: Specific Bahaviour of GPA-ES Evolutionary System Observed in Deterministic Chaos Regression. In: Zelinka, I., et al., (eds.) Nostradamus: Modern Methods of Prediction, Modeling and Analysis of Nonlinear Systems, pp. 73–81, Springer, Heidelberg (2013), pp. 73-81. Advances in Intelligent Systems and Computing. ISSN 2194-5357. ISBN 978-3-642-33226-5.
2. Flynn, M. J.: Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers. C-21 (9): pp. 948–960 (1972). doi:10.1109/TC.1972.5009071.
3. Koza, J. R.. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA, (1992)
4. Beyer, H.-G.,Schwefel, H.-P.: Evolution strategies - A comprehensive introduction, Natural Computing, (2002).
5. Langdon, W. B.: A Many Threaded CUDA Interpreter for Genetic Programming. EuroGP (2010).
6. Langdon, W. B.: Graphics processing units and genetic programming: an overview. Soft Computing 15, pp. 1657–1669, (2011).
7. Jinhan Kim, Junhwi Kim, Shin Yoo: GPGPGPU: Evaluation of Parallelisation of Genetic Programming using GPGPU. Korea Advanced Institute of Science and Technology, Republic of Korea
8. OpenMP compilers and tools, `https://www.openmp.org/resources/openmp-compilers-tools/`

9. Hayashi, A., Shirako J., Tiotto, E. Ho, R., Sarkar, V.: Performance evaluation of OpenMP's target construct on GPUs - exploring compiler optimisations. Int. J. High Perform. Comput. Netw., 13, pp. 54–69, (2019).
10. CUDA C Best Practices Guide, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
11. OpenMP on GPUs First experiences and best practices, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
12. Brinskiy, M., Lubin, M.: An-introduction-to-MPI, https://www.intel.com/content/dam/develop/external/us/en/documents/an-introduction-to-mpi-3-597891.pdf
13. Intro to Parallel Programming https://princetonuniversity.github.io/PUbootcamp/sessions/parallel-programming/Intro_PP_bootcamp_2018.pdf
14. CUDA C Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html\#c-cplusplus-language-support
15. Park, T.-J.: CUDA-based Object Oriented Programming Techniques for Efficient Parallel Visualization of 3D Content. June 2012Journal of Digital Contents Society 13(2). DOI: 10.9728/dcs.2012.13.2.169
16. How to use class in CUDA C, https://forums.developer.nvidia.com/t/how-to-use-class-in-cuda-c/61761/2
17. Intel Compilers, https://github.com/intel/llvm
18. HipSYCL, https://github.com/illuhad/hipSYCL
19. Chapel language, https://chapel-lang.org/
20. X10 language, http://x10-lang.org
21. Almasi, G.: PGAS (Partitioned Global Address Space) Languages. In: Padua, D. (ed.): Encyclopedia of Parallel Computing, Springer US, Boston, MA, 1539-1545, (2011). ISBN: 978-0-387-09766-4