A comparative study on parallel programming languages application

Artem Grukhal

Abstract—Currently the world of computing experiences the boom in computation, and newer methods of performing computations faster and more efficiently arise. This may include different approaches: new computation types (quantum computing, HPC etc.), new, more efficient programming languages, operation systems, improved hardware and parallel and so on. The primary focus of this report are parallel programming languages and their usage. It would be rational to discuss, how well adapted are they in the modern world, what would we want to see when using such languages, where to find application for them and most importantly: Do they actually make a difference?

1 Introduction

The ongoing research is still being made for developing new parallel programming languages and methodologies for parallelizing the environments. Latest trends are fuelled by both improved programming models and emerging hardware developments: better network connectivity, higher processing speed of CPU, usage of GPU for purposes other than computing the graphical elements etc.

When we talk about programming models, there are main ones, that have major popularity: programming with shared memory, distributed memory and programming on GPU. Originally, programming for all of these concepts was supported on rather old languages, like C/C++ (first version of C++ developed in 1986 [13]), Fortran (Created in 1954 [14]) [1]. Now we have flexibility to use languages, that are easier to write with, and easier to understand in general. Therefore, we have the ability to improve the performance of computing without losing the productivity of a developer. For this matter new parallel programming languages are developed, which are actually going to be the main scope of this paper.

This article will compare the ease of installation/adoption of a parallel programming languages, discuss if it makes a difference in the development speed and evaluate if the parallel programming languages are actually needed, or using the libraries from C/C++ and Fortran will be sufficient.

2 Programming languages

There are many languages to choose from, when deciding to evaluate the parallel features. A lot of major languages nowadays have the support for concurrency, so it would be good to compare both, languages that are developed specifically with the purpose of multi-threading, and languages that have multi-threading as part of the feature. For this, there are several languages which will be chosen to compare about:

Python - a general-purpose language, that can be used for MPI programming. Not multi-threaded by itself, however, it has libraries that allow it to be threaded.

C++ - the native language to most parallel programming implementations.

Harmony [2] - a model-checked programming language built specifically for parallel applications. This language will be used as one of the parallel languages for discussion, and will provide expertise from this perspective.

Cilk [3] - old general-purpose language designed for multi-threaded parallel computing. This language will be compared with the others, however it is worth noting, that as of now it is being deprecated and the only supported version is present in form of OpenCilk framework developed by the MIT.

These languages will be used to support thoughts, however, they will not include the technical demo, since the comparison between all of them would not be descriptive for the topic of discussion. The next sub-paragraphs will look at each one of them, and motivation for using exactly these languages.

2.1 Python

Python is widely used among scientists, and this is one of the main domains, where parallelism could be beneficial. Python is a general-purpose language, and it features a lot of external packages for usage in various fields of studies. It can help leverage the web development with minimum code required, and since it is widely used in scientific fields, it includes a lot of computation libraries, some of which include the using of parallel processing. The reason this language was chosen for the purpose of comparison in this report is because it is a very good beginner-level language, that may straighten the learning curve and help with support of the opinion of redundancy for development of newer programming languages, that specifically target the development with the help of parallel programming models like Cilk language, or Harmony language.

2.2 C++

This language is one of the only languages that can support most parallel programming models. There are numerous extensions, and the 3 main models discussed here are present as extensions as well. This is the oldest language on this list, and it either inspired most programming languages now existing, or was used to create such languages. It was

chosen to be overviewed, because it is still widely used among developers' community, and is often used to write the parallelization logic for newer languages, operating systems, etc. It supports most parallel programming models that exist, and is a tool that we may be certain to contain the functionality we need. Yet, it has its flaws with being too difficult to grasp, especially considering the scenario for writing parallel computation software. That is due to everything starting from memory manipulations has to be programmed manually - pointer logic should follow the parallel approach, the program has to adapt to specific hardware limitations that may mean the code will not be multi-platform and lastly, the amount of code written will at certain point become difficult to maintain.

2.3 Harmony

Harmony is developed by the Cornell University specifically for experimenting with distributed programs, and since it checks the model instead of running it, it is promised that all corner cases are explored. Basically, this is more of a testing language, to test the concepts of how well the program will behave, when developed with this language.

2.4 Cilk

Cilk is developed by Intel, and initially designed by MIT. The language's primary focus is development of application that use multi-threading, and it has lately received an open-source version OpenCilk [4].

3 Programming models

3.1 Shared memory processing

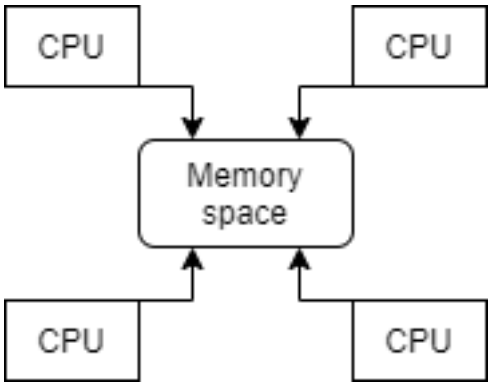

Fig. 1. The basic depiction of shared memory architecture

Shared memory model, depicted on figure 1, offers a way to parallelize an application without moving memory objects or pointers anywhere, but rather performing tasks on static memory, that are being operated from several processors. Shared memory's most obvious approach is fork-join paradigm, which essentially means that at some point all the processes that are being run in the program start running in parallel, with the help of different CPU cores, and after they are joined the program continues to run sequentially.

Shared memory gives advantage in speed, since we do not spend time to reassign the memory slots and move tasks around, however, it does offer limitations that may slow down the program.

One of such limitations is Read/Write problem [5]. The operations that are performed on a shared memory location need to have some sort of consistency mechanism, that will disallow the simultaneous Read and Write of different processes and alleviate the possibility of getting incorrect values in such memory cells.

As a result, even though the program is parallel, there still needs to be some sort of order, in which processes access memory space, and this is why a programmer has to first consider if the program will have many potential Read/Write overlaps or if the program relies much on intermediate execution results, and if the gained speed-up will be sufficient to use such model.

3.2 Message Passing

Message passing models assumes two main concepts [6]: synchronization of processes and access to memory of other processes. It is performed in a way where one process may send or receive bits of information to/from another process, and then operate with the information received.

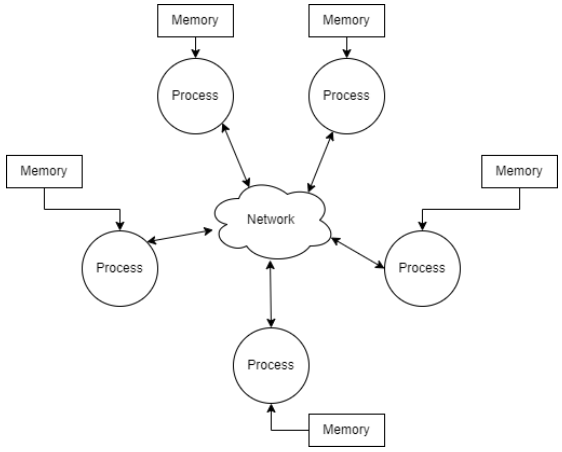

Fig. 2. The basic depiction of message passing architecture. We may see that there are N independent processes that may communicate with one another.

The layout of the message passing concept can be seen on figure 2. The biggest difficulty that arises during the development with such models is the growing complexity of a program, and overhead speed loss during the communication of different processes. Such issues may significantly slow down the program, and result in unnecessary deadlocks and/or flaws in program's logic.

The overhead losses may be compensated by estimating the size of data that is to be splitted between processes for further operating, and the computational power of the machine, hence, the number of processes that may be spawned. Some communication losses would still be present, but the power of the machine would be utilised better and therefore the program will operate faster.

As per growing complexity, there is no way to improve this aspect due to the problem lying in the core concepts of the Message Passing model - communication and synchronization. Large-scale problems should be avoided for such models, as further developments will increase the

complexity significantly, and development time will grow as well. Here the large-scale problems refer to complex logic regarding the communication in such systems, e.g. writing some simple algorithms is widely used, however, using many algorithms that overlap with one another is not preferred, as debugging will increase in complexity. Since the processes are tied between each other, the whole model runtime may be viewed as a graph of processes communicating, and it is best to either keep the graph as small as possible, or to keep it simple and iterative, without many overlaps.

3.3 GPU programming

GPU programming has been a rather new addition to parallel programming models, and only recently has gained the attention it deserves. According to a research by Andre R. Brodtkorb et al [7], the trend only began in 2000s and only now is being actively used and researched. This is mostly fueled by GPUs becoming more powerful, and the possibility to run non-graphical code externally. Some of the best property of a GPU over CPU is that GPU can process a large number of small operations much faster than CPU can. Generally a single CPU core itself is faster than a single GPU core, however hardware-wise the GPU usually has way more cores than a CPU, hence wins in processing of many small operations. It may also process rather big amount of data, and therefore makes it perfect for quick and easy computations, however there are flaws as well.

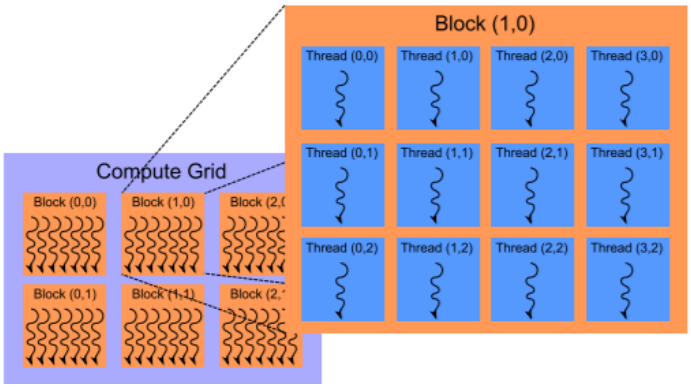

Fig. 3. The CUDA concept of a grid of blocks. Each block consists of a set of threads that can communicate and cooperate. Each thread uses its block index in combination with its thread index to identify its position in the global grid [7].

According to authors [7], GPU programming is very difficult to utilize at point, where resources of a processor are used in its most efficient way. It offers an architecture, in which there is a kernel, that will be put over a set of blocks, and each block will have a certain number of threads within itself as can be seen from figure 3. Making it difficult to switch from a regular CPU-parallelization, developers may take longer time to make parallel programs work on GPU correctly, since it will also require some level of expertise in underlying technology on the lower level.

The expertise that has to be gained, in order to efficiently manipulate the GPU threading is also usually vendor-dependant, since different GPU vendors still have different architectures, and as a result different interfaces and libraries to be used to make the program run. This brings to a second problem of code portability. The code may run rather good on one machine, however with the change of hardware the code may become useless, or require some additional steps and development being done, to tackle this problem.

Regardless of the downsides, once the program may run correctly over the scenario that was supplied to it, it will have significant speed-up as compare to a sequential CPU parallelization. Nowadays GPU programming is used a lot in simple scientific computations, and may be injected at different locations of code therefore making it unnecessary to base the whole code structure around this parallel programming concept.

4 Comparison of Languages

As it was mentioned above, languages can be evaluated by several parameters. The next subsections will refer to some most common parameters and discuss how well the language is adapted to current trends and to propose the reasons why it should, or should not be used for everyday development.

4.1 Setting up language environment

The idea of this evaluation is to find out which language is the easiest to go with initially. Here it is no doubt that Python seems to be the best option. The installation procedure is rather simple, and does not require much technical knowledge. It only takes around a couple of hours to setup the environment together with an IDE of preference.

Next in list comes Harmony, since it is more or less similar to Python installation. The language uses Python compiler to run the installation script, and gcc compiler for the features that are present in the language. Harmony does not have a standalone IDE, however it has online IDE and some extensions to other widely known IDEs, so after the installation, the usage should be rather straightforward. It was decided to go with the standalone Harmony compiler since the extensions sometimes need a connection to remote server that will run the Harmony code, and this is not stable as later results will show.

C++ is the next in terms of simplicity, since installation requires not only to install the compiler, but also additional libraries that will help to run parallel code. For instance, since C++ supports all the programming models, installing libraries takes time to figure out all the necessary installation steps.

Lastly, Cilk seems to be the most complicated in terms of installation procedures. It can only be run on Linux/MacOS operating systems, and for running on Windows machine, which the author has, there is a need to use Windows Subsystem for Linux (further WSL). After WSL is installed, the shell script has to be run.

4.2 Support of parallel programming models

4.2.1 Python

From the three models defined, with appropriate extensions Python supports all of them, however only two have their

separate packages and wide usage: GPU programming and message passing.

For GPU programming it is possible to utilize the CUDA package and numba, which is pretty analogous to widespread numpy, but runs its operations on GPU if such possibility exists. GPU programming in Python is already quite widespread. It is being used in numerous Machine Learning models, and in general for speeding up scientific computations.

As per message passing model, Python has an implementation of an MPI interface, which is one of the most popular interfaces for message passing model in general. It allows for utilizing defined number of cores of computer's CPU and distributing tasks among these cores as messages. There are various architectures of how messaging can be operated in Python as well, so the tool is flexible in terms of architecture one may come up with.

Python also provides support for shared memory model, however it is not as popular as C++ OpenMP, and it lacks some features, that OpenMP supported languages possess.

4.2.2 C++

C++ supports all of the models, and similarly to Python has most of them provided as additional libraries. C++ allows for writing own software that will implement such solutions as well, but in this report some well-known libraries will be used to help explaining how the language may support the models.

In GPU programming C++ has a variety of modules that may be used. However they all mostly rely on two libraries CUDA and OpenCL. Based off these libraries, modules were written to simplify the development process, and there is a wide variety to choose from. Main difference is the fact that CUDA only supports the NVIDIA GPUs, however OpenCL is more versatile, and may support not only GPU programming, but also help to parallelize applications on several CPUs.

The main player for message passing network in C++ is MPI. It has several implementations, however the main interface is intact and is a lot similar to Python implementation. There is a possibility to program one's own message passing model, since the C++ language itself provides a big variety of opportunities one may utilize, however, since it is mostly inefficient and due to the language complexity may take a lot of time, it is still preferable to use the MPI library where needed.

C++ also supports shared memory applications and has several libraries for performing so. One of the most popular libraries for shared memory programming in C++ is Boost library. It provides the possibility to create shared memory objects, which then may be manipulated through different threads. However, the readability of such code may be not of the best quality. Another popular alternative is OpenMP interface. Its implementation in C++ allows to create various code chunks with use of pragma-fields, to define different forks and joins, which then enable this code to be run in parallel using the same memory addresses.

4.2.3 Harmony

Harmony is built differently from the first two languages. Since the Harmony language is very young [2] and it is model-checked, we may not say that it supports either of these models, however there is a possibility to implement some of the models and test their correctness against Harmony's compiler. All of the implementations have to be done by the programmer, since there is practically no way of finding them online. The authors of Harmony language provided an extensive handbook [2] to Harmony's syntax and several examples to help getting started with implementing.

The language is compiled through the Harmony Virtual machine, that essentially will manipulate 3 main components:

- Code
- Shared memory address
- Threads

Even though Harmony operates over the shared memory space, there is a possibility to make threads store their private values, that may emulate message passing interface.

The only limitation in regard to the three programming models that were discussed earlier is the GPU programming, since the language is so young it doesn't have all essential basic data types to be able to develop in this direction. But as it is currently being developed, such support may appear in the future, once the language becomes more mature.

4.2.4 Cilk

Cilk is very limited in terms of parallel language support. It was originally developed to be a standalone language, but later became more of an extension to C/C++ language. Currently it only supports shared memory model, and may only provide features related to it.

Cilk is currently being deprecated, and the interest in it has faded with the recent years, however, there is an OpenCilk platform, which aims to be an alternative to OpenMP interface. Hence, being even more pushed into being an additional library to C++ language.

4.3 Variety of applications

4.3.1 C++

So far C++ is the top language in regard to ways one may use it. However, due to the complexity of developing software in C++ it is mostly used in the following industries:

- Operating systems
- Game development
- Embedded systems
- Distributed systems

These main industries use parallelization only to a certain extent, and all have different parallel models preferences.

Operating systems is the branch, where all of them can be applied together, but only from the perspective to allow programmers to leverage the power of multi-threading as they develop using these operating systems. This means, that there is more development to be done in C++ language, rather than using any ready-made libraries, as different operating system (later - OS) may further have some libraries for multi-threading built on top of their virtualization, but the core is usually simpler.

Game development concentrates on smooth experience when running applications, and here one of the important considerations is to achieve maximum machine utilization in order to provide smoothest experience. This industry may benefit from the parallel programming the most, since this may help to speed up the processes significantly, but there are also complexities tied to it. Since games sometimes are developed in sequential manner and have only one thread, increasing number of cores involved may increase the code complexity substantially.

After all, there has to be at least one main thread, which is responsible for rendering the images, and this is where the GPU programming may come into play. An interesting approach is also described in a paper by M. Joselli et al [8]. Here the author suggests that due to GPU processors being much more powerful in terms of computation, at certain point they can be used for leveraging the computational power for not only rendering of the image, but also using it for logical elements. Author believes that such approach may increase the performance of applications, and in this case GPU programming model will be helpful.

Embedded systems typically do not require high level of parallelism, since these systems are usually rather small and only consist of either one processor, or only several, making it hard to achieve any considerable speed up. However, with the recent developments there are some hints that having several threads in certain applications may benefit from having several threads in the system [10].

Lastly, distributed systems indeed use a big share of parallelization techniques. However, since this is a rather big domain, that typically uses a big number of different servers, the standalone machine itself is imagined as a process. Here all the models except for GPU programming may apply, and since the nature of this branch itself is hinting on using techniques from parallel programming, most algorithms implemented already have supporting software to use and manipulate the distributed systems architecture. Moreover, since in this case we may define a thread as a standalone machine, it provides for greater variety of distributed systems architectures to be developed.

4.3.2 Python

Python is mostly appreciated for its simplicity, and hence the variety of applications grows with each year. Since Python is a scripting language it's syntax is mostly simplistic, and all the external features rely on modules that are built for Python. Such modules are usually built using lower level languages like C/C++. Some of the main application fields that are currently booming in Python community are:

- Scientific computations
- Machine learning
- Web development
- Data science

Scientific computations is among the largest use cases of Python programming language. This includes writing applications that utilize supercomputers' resources. This is particularly interesting, since the number of cores in such systems allows for flawless parallelization, where the computing speed can be increased significantly with the help of either parallel programming model discussed in this report. As all of the models can be used, this is a trending topic at the moment, and shows great perspectives for possible application.

Machine learning is also a big domain, that partly includes data science, yet, here it is mentioned in context of the algorithms for machine learning models. Parallel programming may be used to improve the parts of algorithms, where the one thread's processing does not depend on another thread. Particularly, some layers of neural networks may benefit from such approaches.

In fact, GPU programming is already widely used to speed up the calculations, yet, there is still room for improvement with incorporating more hybrid models for parallel processing, possibly distributing parts to different machines that have their own GPU cores.

Modern web systems are also built in a way that a certain level of multi-threading is present, allowing for different requests to be run on different cores, however, there is almost no possibility to incorporate GPU programming because of its redundancy. Web requests typically should not contain massive calculations, therefore it is hard to provide real speed up in this sphere.

Data science may benefit from parallel processing in case there is a need to pre-process data, or analyze very big chunks which may be analyzed independently. In theory all of the models can be utilized, but this would be specific in relation to the task that stands for a programmer.

4.3.3 Harmony

The main application of Harmony language will be mostly testing different parallel computing paradigms and constructing algorithms for parallel computing. This will particularly be interesting in the following fields:

- Research
- Quality assurance
- Education

For the first option, the possible usage may include the development of different architectures for parallel programming, since the language provides support for finding common issues that are related to parallel processing:

- Safety violation
- Non-terminating state
- Active busy waiting
- Data race

Since the research directed at new algorithms usually includes proving the correctness of algorithm, and making sure that all the corner cases are covered, languages like Harmony seem to be a very useful tool in finding such corner cases.

For quality assurance purposes these languages may be used by regular programmers, for verifying that the solution works as expected. Even though the usage of parallel concepts is not as popular in industrial projects, the mature version of languages like Harmony may be extended so the testing of already existing solutions that use multi-threading is easier and provides not only fault-related information, but also some statistical information regarding the code run.

Lastly, for educational purposes this language has a great potential to be used in teaching the parallel programming

concepts. It would be easy to showcase some of the components that make up a distributed application due to the language creators providing extensive documentation with some very good examples.

4.3.4 Cilk

Since this language is becoming obsolete, and moved to a status of an extension to C and C++ languages, its application scope is similar to what C++ has. The main limitation is that Cilk may only provide parallelization based on shared memory model.

The only branch that might not benefit from Cilk may be the distributed systems sphere, since Cilk is designed to work correctly within the same machine applications, whereas in distributed systems often we would expect different different machines passing messages to one another.

Cilk essentially makes C/C++ languages simpler in terms of parallelization, so theoretically we may assume that it will improve the code readability and speed up the development time, when compared to plain C/C++ languages. With such an approach, the system will also be more bug-free since Cilk handles scheduling in its core, reducing the amount of bugs a human developer may produce.

4.4 Community support

One of the main features that a language should have is a strong community support. It can describe how many programmers there are to take advice from, and how well-discussed the problems of a specific language are, which make development easier.

The leaders for community support would be C++ and Python [11], since both are oldest languages from the list (Python first version developed in 1991 [12]), and given that both are used proactively even today, the support for such languages is not expected to decline any time soon. There are various forums, dedicated not only for the language features, but also for specific libraries provided by such features. The documentation is extensive as well, and gives possibility to quickly find a solution should a problem occur.

Cilk, on the other hand, has very small community. There are no official public forums for this language, and there are no intentions to have one, since as it was mentioned earlier the language is being deprecated.

Harmony is a promising language, and if it keeps being developed, it has a good potential to receive support from other developers. There is no official forums or hot discussions about the language yet, but given the language is only developed by a team of scientists and is too young to be useful in everyday applications this may not happen in the nearest future. Although, the language has a very good documentation [2], and it is described how exactly it works from the Harmony Virtual Machine perspective, which will surely make development and debugging the code easier.

5 Conclusion

This paper tries to shine a light on some recent developments and advancements in the sphere of parallel computing using three main models, that are commonly used. In general it can be seen that utilizing GPU cores for more efficient computation seems to be valuable research focus and languages that will support this vector would be preferable in the nearest future. It has also been discussed how exactly different languages provide the support for parallelization of software, and which are the ways to use libraries or modules provided by some bits of software.

It can be seen, that the trend is such that developers prefer to extend the existing languages with functionality through libraries, instead of creating new standalone parallel languages and possibly porting some features from other languages, thus adapting such libraries to the chosen one.

Another note we may learn from this report, is that developers try to find the application for parallelization not only for general-purpose programming languages, but also for some specific languages like Harmony, which essentially is a language that will help with testing the parallel models created by developers. With the necessary attention such languages may be used to alleviate various actions that are still performed manually like algorithm proofing and software testing. Ideally, there may exist other concepts of languages built with the parallel model under the hood, that may give a way to other actions in software development cycle or a research that are currently too complex to handle programatically.

References

[1] Pryadko S.A., Troshin A.Y., Kozlov V.D., Ivanov A.E. Parallel programming technologies on computer complexes. Radio industry (Russia), 2020, vol. 30, no. 3, pp. 28–33. (In Russian). DOI: 10.21778/2413-9599-2019-30-3-28-33

[2] van Renesse, R., 2020. Concurrent Programming in Harmony. Link: http://harmony.cs.cornell.edu [Last accessed: 24.11.2021]

[3] Thoman, P., Dichev, K., Heller, T. et al. A taxonomy of task-based parallel programming technologies for high-performance computing. J Supercomput 74, 1422–1434 (2018). https://doi.org/10.1007/s11227-018-2238-4

[4] Tao B. Schardl, I-Ting Angelina Lee, and Charles E. Leiserson. 2018. Brief Announcement: Open Cilk. In Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18). Association for Computing Machinery, New York, NY, USA, 351–353. DOI:https://doi.org/10.1145/3210377.3210658

[5] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," in Computer, vol. 29, no. 12, pp. 66-76, Dec. 1996, doi: 10.1109/2.546611.

[6] Fan Chan, Jiannong Cao, Yudong Sun, High-level abstractions for message-passing parallel programming, Parallel Computing, Volume 29, Issues 11–12, 2003, Pages 1589-1621, ISSN 0167-8191, https://doi.org/10.1016/j.parco.2003.05.008.

[7] André R. Brodtkorb, Trond R. Hagen, Martin L. Sætra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, Journal of Parallel and Distributed Computing, Volume 73, Issue 1, 2013, Pages 4-13, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2012.04.003.

[8] M. Joselli and E. Clua, "GpuWars: Design and Implementation of a GPGPU Game," 2009 VIII Brazilian Symposium on Games and Digital Entertainment, 2009, pp. 132-140, doi: 10.1109/SBGAMES.2009.23.

[9] Lisandro Dalcín, Rodrigo Paz, Mario Storti, MPI for Python, Journal of Parallel and Distributed Computing, Volume 65, Issue 9, 2005, Pages 1108-1115, ISSN 0743-7315, https://doi.org/10.1016/j.jpdc.2005.03.010

[10] S. Aldegheri, S. Manzato and N. Bombieri, "Enhancing Performance of Computer Vision Applications on Low-Power Embedded Systems Through Heterogeneous Parallel Programming," 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), 2018, pp. 119-124, doi: 10.1109/VLSI-SoC.2018.8644937.

[11] Programming languages popularity, Link: https://www.tiobe.com/tiobe-index/ [Last accessed: 20.12.2021]

[12] Python history of versions, Link: http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html [Last accessed: 20.12.2021]

[13] Stroustrup B., The C++ programming language, 1986, Addison-Wesley Publishing Company, Inc.
[14] Fortran history by IBM, Link: https://www.ibm.com/ibm/history/ibm100/us/en/icons/fortran/ [Last accessed: 20.12.2021]