





A Plugin-Based Approach for Parallel Programming Kit

Pavel Vasev¹  and Sergey Porshnev² 

¹ Institute of Mathematics and Mechanics Named After. N.N. Krasovsky Ural Branch of the Russian Academy of Sciences, Ekaterinburg, Russia
vasev@imm.uran.ru

² Ural Federal University Named After the First President of Russia B.N. Yeltsin, Ekaterinburg, Russia

Abstract. Parallel programming is hard. To make it a little bit easier, various computation models and technologies are developed. However in turn they are complicated itself, making it harder to add useful features to them. One way to reduce software complexity is a modular approach, for example plugin-based. It allows to add features into software in a structured way. In this paper, a plugin-based approach is suggested to be used in the author-developed technology for parallel computing, the Parallel Programming Kit (PPK).

Keywords: parallel programming · online visualization · plugin system

1 Parallel Programming Kit

Nowadays online visualization (e.g. visualization of scientific simulation during it computation) requires special measures [1]:

1. additional large-scale computations, dedicated to visualization itself, e.g. to visualize computational datasets a separate high-performance computation is required;
2. insitu processing, e. g. to perform such visualization data should be processed almost near places where this data is computed.

According to last aspect, today such visualization is called “insitu visualization”.

One approach which may address these needs is a parallel programming technology and model, which is able to describe a parallel computation both for computing and visualization on one hand, and it's connection with user computer for other hand. The latter is required so user can view computation visualizations on it's computer.

Parallel Programming Kit¹ is a software technology for creating high-performance computing programs [2]. It was created firstly for online visualization of parallel programs, allowing visualization during computation for steering and control [3]. The overall implied schema for computation and visualization is the following:

¹ The software is available online at <https://github.com/pavelvasev/ppk>.

1. Simulation parallel algorithms compute that they desire;
2. Data produced by (1) is visualized online by parallel visualization algorithms, probably on same computing nodes as (1);
3. Visualization images are transferred to user station for online interpretation, or for image archiving, as in CinemaScience approach [4].
4. In case of online interaction at (3), control signals are transferred back to computation of simulation (1) or visualization (2) algorithms.

2 Communication Model

To implement such a scheme, a message-based approach was chosen. It was reinforced with publish-subscribe pattern², and in short the following communication model was achieved.

Entities:

A **computing environment** is a collection of computers connected by a network and processes running on them. Processes here are meant both in the sense of operating system (OS) processes and in the logical sense (Hoare processes [7]).

A **message** is a transmitted unit of information. Message contains a value which has form of binary data block.

A **channel** is a special entity for transmitting information. Messages are transmitted over the channel.

A **channel namespace** is a namespace common to the computing environment, in which channels are mapped to names. These names will be called channel identifiers.

Operations:

Open a channel for writing:

$$\text{open} : \text{id} \rightarrow \text{channel_obj}$$

where id is the channel identifier in channels namespace, channel_obj is a local object for interacting with channel.

Send a message to a channel:

$$\text{put} : \text{channel_obj}, \text{value} \rightarrow \text{nil}$$

where channel_obj is a local object for interacting with the channel; value is the value to send in the message.

Receive messages from a channel:

$$\text{react} : \text{id}, \text{func} \rightarrow \text{rhandle}$$

where id is the channel identifier in channel namespace; func is a local function that will be given control when messages arrive in the channel, with an argument — the message value; rhandle is a local object that allows to stop receiving messages.

Thus, the process that triggered the react operation becomes the “recipient” of messages from the specified channel. It should be noted that technically, the senders and

² The noted publish-subscribe pattern is described at https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

recipients of messages from one channel or another may be on different machines. There are no restrictions on how many writing or reading sides the channel has.

Create link between channels:

bind : source_id, target_id \rightarrow lhandle

where source_id is the source channel identifier; target_id is the receiver channel identifier; lhandle is a local object that allows you to delete the connection. The *bind* operation starts the process of copying messages from the source channel to the receiver channel.

It should be noted that the concept of link between channels is a first-class object, same as the concept of channel. Together, links and channels allows to describe internal communications of parallel programs, their visualizations, and interaction with the user.

3 Important Implementation Details

For performant operation, it was necessary to implement operations stated in communication model with some special techniques.

Direct Message Sending. Messages are transmitted between senders and recipients directly, without intermediaries. For this purpose, a general registry agent is maintained. This agent collect information on added links and reactions in a form of global registry, and sections of this registry are transferred to senders. The sender has a complete list of message recipients and sends messages to them directly. When the registry is changed, the relevant updates are sent to the senders. No additional registry requests are required at the time of sending the messages. That is, to send a message, a client just sends it to a list of currently known recipients.³

Links Processing Directly on Sender. A channel can be a source in various links (see communication model). When performing the operation of sending messages to a channel, links are processed immediately. A sender of a channel has a complete list of recipients, e.g. linked channels (from the registry, see above) and calls the operation of writing a message to these channels. This allows you to set up complex message routing between channels.⁴

Direct Passing of Messages on Same Process (on same host). Then processing links directly on sender as stated above, this may lead to situation that messages may travel through many channels. But actually messages may even does not leave the sender's operational system (OS) process. For example consider if some code *A* is located at some OS process *X* and performs send operation. Consider another code *B* that is located at same process *X* and let *B* performed subscription. It is possible to send this message

³ It is possible that it can be that the registry has not yet sent updates to a sender but it starts to send messages to the previously known list of recipients. In current version of PPK, this is not resolved. However one may write a plugin that for example provides operation to wait for specific number of recipients on specified channel(s).

⁴ If there are several recipients that are on some node which is different from the sender's node, it is possible to optimize sending, e.g. to send a one message to the node. It is considered that such optimization might be performed at a plugin level, e.g. in a plugin responsible for message delivery via network.

directly, from *A* to *B*, by calling callback function at *Y* *directly*. E.g. without crossing user space / kernel space boundary nor network boundary.

Passing Pointers Instead of Data. If the sender and recipient are in the same OS process, the message values are transmitted via a pointer to memory (RAM). A data is not copied or moved anywhere. This feature turned out to be important, because it turned out to be convenient and efficient to run different logical processes and “disperse” their parts into the same OS processes. In this case, such processes interact by sending messages in channels at very high speed. The same speed as if they were implemented in code of single program and called each other’s functions directly.

4 Some Applications

Using suggested technology, various interesting results was achieved.

Task Graph. There was implemented a task-graph parallel programming model, details are described in [2]. For it’s implementation, special additional services and program roles were introduced:

1. Task broker service. It reads commands from “task” channel, collects incoming tasks, and schedules it to workers. A task was denoted as id of function to call, and it’s arguments⁵. Some arguments of functions might be global promises. In this case, a rule apply: a task may be executed only when all global promises noted in function arguments are resolved.
2. Promise service. It tracks global promises (e.g futures⁶). Each promise has a global id, and if promise is resolved, a corresponding global url-based pointer to data in data layer (implemented as a memory of workers).
3. Workers role. A worker is a process that is launched on supercomputer node and queries tasks from task broker service (1) for execution.
4. Task creator role. It is a user script that generates tasks and submits them to task broker service (1). Each task immediately produces a global promise, which might be used as an argument for other tasks, so on.

Using this model, for example, a parallel visualization of a voxel cube was implemented, see Figure 1. It’s architecture was the following: a web-server programs provides graphical interface to the user. User moves mouse within interface, and thus manipulates camera position. It is sent to web-server, which in turn generates tasks for voxel cube rendering, which immediately began to execute on a supercomputer. After image parts are rendered and merged, the result is displayed to a user.

The implemented task-graph model was also used to solve computational task of computing on some form of Markov processes [5], see Figure 2.

After that application, it was realized that suggested task-graph implementation is only capable for performing on large-grained tasks. On tiny-grained tasks, a scheduling overhead is significant. To solve this, various approaches was implemented. It’s main

⁵ It is assumed that worker have knowledge how to execute functions by given id. For example, these might be function names in specific library.

⁶ Promises are described at https://en.wikipedia.org/wiki/Futures_and_promises.

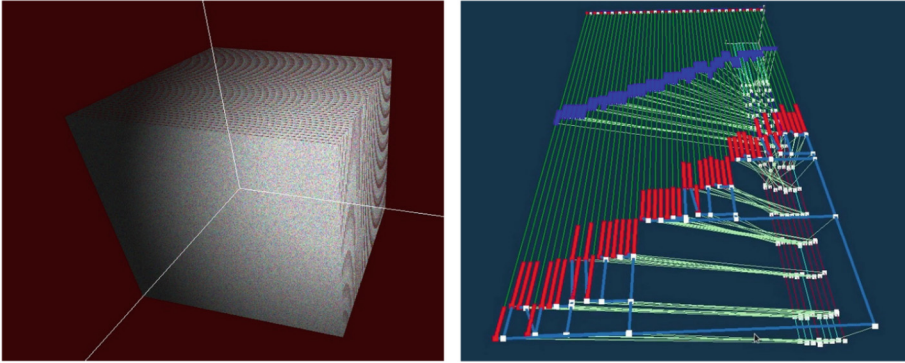


Fig. 1. Left: interactive parallel rendering of voxel cube. A voxel cube having 10^9 voxels in total is split into 50 parts. User may change camera position using mouse. After a change, a GUI code sends “render” message with new camera position, and the cube gets re-rendered. Right: visual debugging of used parallel rendering algorithm. Animation: youtu.be/XnV3l8hw8QE. See [2] for details.

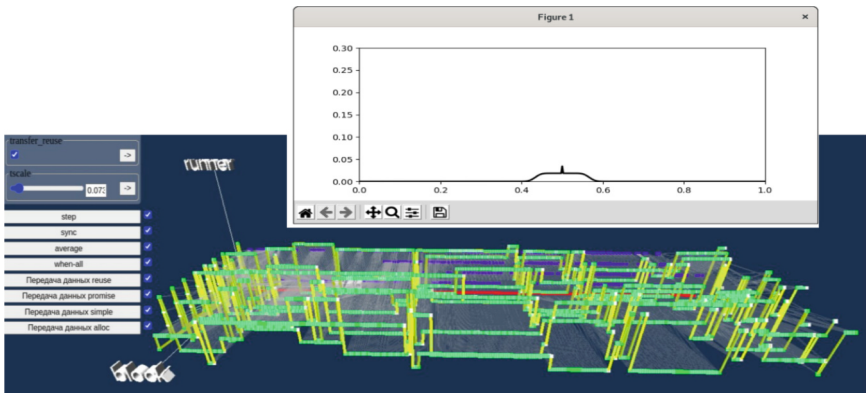


Fig. 2. Top: a result of parallel computation of 1d function. Bottom: visualization of task-graph that computes this function. Axes: X is time, Y is worker (runner), Z is grid data block number. Vertical change of horizontal green lines means rescheduling of computation of a grid data block to another worker (runner). Please see [5] for details (color figure online).

idea was to move task graph to virtual sphere. Tasks become virtual, and are generated in distributed fashion.

This means, that task graph idea is still considered by authors as effective for reasoning (e.g for human mind). However, it is considered ineffective for practical implementation. As it noted, for the reasons of overhead introduces by processing tasks. E.g. to execute some task, this task information should be generated, encoded, transmitted via network to task scheduler(s), decoded there and encoded again, transmitted to workers, so on. If task itself contains say 10K simple operations, the overhead of processing such small tasks became huge, relative to task processing itself.

To overcome this, simple idea is used: let workers generate tasks for themselves by themselves. E.g. let put the knowledge of which tasks should be performed, right to the place where these tasks should execute. For example if worker *A* should execute tasks T_1, \dots, T_n then let some algorithm located directly at worker *A* will produce the definitions of this tasks. In that case, all overhead related to tasks processing is decreased practically to zero.

Of course other overheads stay in place, e.g. for tracking distributed promises, network communication, so on. However it also can be reduced in some places. Last, some kind of balance is required, to make the model still reasonable for human mind and effective for execution. Please see [6] for details of author approach and related works on this topic.

Finally, the suggested communication technology was used for processing graphical user interfaces. A special technology named Grafix was developed [7]. It was noted that channel primitives are capable for describing communication with elements of graphical interfaces. A user programs submits commands to special graphical broker to create trees of graphical elements (buttons, checkboxes, rows, windows, 3d scenes, etc) which are considered as Hoare processes [8], and establishes communications with them via channels. When user interacts with graphical elements, they emit messages to channels. When program needs to update state of graphical elements, it sends messages to their incoming channels. The Grafix is now used in some custom software for web and mobile applications. It is suggested to be a basis for online visualization interfaces for parallel computations.

5 Plugin-Based Approach

To burst Parallel Programming Kit development, it was decided to convert it to a plugin-based architecture. It will allow to implement new features in a standard way.

By plugin-based architecture⁷ authors mean the following.

A **plugin** is a software module that implements a set of features.

A **feature** is a some valuable aspect, which brings new abilities into software.

A **system** is a special software module that loads user-defined set of plugins, and thus constructed software obtains desired features.

The main ingredient of plugin architecture is the following. The “load of a plugin” operation is meant to be calling it’s initialization function. When calling that function, a system passes system’s application programming interface (API) to it as argument.

After obtaining pointer to that API, a plugin may embed it’s codes into system behavior by calling that API. For example, for graphical program such API may contain a command for adding menu to a program, or a button to a toolbar. One plugin may call it to add some menu, and react to messages when this menu is clicked by user. Another plugin may add many menus. And third plugin may not add such menus, but instead add a set of buttons to a toolbar, so on. Thus, the final behavior of a system is distributed into context of it’s plugins.

⁷ Plugins are also described at [https://en.wikipedia.org/wiki/Plug-in_\(computing\)](https://en.wikipedia.org/wiki/Plug-in_(computing)).

This allows a system developer to create and document system API, and then invite other developers to create plugins and thus add features to a system. Thus it allows a way for communication with developers.

Another important feature of plugin-based approach is that it allows to separate feature codes into plugins. When some feature becomes non-important, its plugin is just removed from load list.

For Parallel Programming Kit, the following ideas of plugins are considered.

Message Passing Technology. Currently, messages are sent via TCP protocol. It is considered inefficient in HPC environments. There are more efficient variants specific to networking technology being used by a cluster, for example InfiniBand, Angara, RDMA hardware support, so on. To implement that, a system should provide *internal message* “send a message”, and some plugin should respond to that message. In other terms, a system should have an open (virtual) method of sending messages, and some messaging plugin have to fulfill (implement) that method.

Task-Graphs. As stated in applications section, some form of task graphs was implemented using the system. It was realized that the basis of communication is not tasks or promises, but messages. Tasks are abstractions above messages. However they are very robust in some applications, and it will be useful to have task-graph paradigm as a plugin. To implement that, a task-graph plugin should provide an API for a user to create tasks, global promises, so on. On other hand, the system should provide an API for starting services, so task-graph will be able to start task broker service, promise service, so on.

SYCL Integration. The SYCL technology allows to write a C++ code which will be able to execute on various hardware, including raw processors and GPU. To support SYCL in PPK, an integration plugin may provide a way for seamless integration of SYCL codes and channel messages. For example, it may provide a method of configuring reactions on messages in channels as SYCL codes; and to push results (partial or complete) of SYCL execution to channels.

Automatic Parallelization. For example, a user may write a “kernel” function in Python. Then software analyzes the code of the function and in some way generates parallel program that computes data using that kernel function. This looks like DVM and Fortran Coarray approach, but without additional hints, only due to analyze of data flow in a code of kernel function. It is well known that topic of automatic parallelization is developed for decades. Here we just denote the bridge between this topic and PPK technology.

Macro-operations. For example, in DASK programming environment a user may write a program using Numpy arrays API, and this program is mapped to parallel execution by DASK auto-parallelizing algorithms⁸. Thus a user writes an algorithm that operates with a high-level entity, which in turn gets executed on a cluster.

Active Knowledge. An active knowledge model and technology [9] is a great candidate for implementation in PPK as a plugin.

⁸ See DASK arrays, for example at <https://docs.dask.org/en/stable/10-minutes-to-dask.html>.

6 Implementation Details

An important aspect of PPK is its support for many programming languages. Currently it is implemented for Python, Javascript, and C++. For that reason, if one implement plugin, it should be considered to be multi-language. Actually this means that many versions of plugin should be implemented, one for each language that plugin author going to support.

In Python, earlier versions of PPK used plugins in following manner. There is a primary object provided for user, usually named *rapi*, which provides user API for interaction with PPK. Plugins are loaded from lists defined in context of PPK itself, and in context of user application. Plugins that were loaded in context of PPK, just defined new methods in *rapi* interface.

For example, to define a method to subscribe to messages, *ppk_query.py* plugin was introduced. It injects “query” method into *rapi* interface⁹. Internally, it implements it by 1) starting a server for receiving messages in caller’s process and obtaining it’s ip address and port, 2) deploying reaction onto message senders via general registry agent. The deployed reaction is parametrized with (ip address, ip port) information obtained at step (1). When senders performs send operation, the reaction code specified by (2) sends messages using TCP protocol to the server (1).

If one want to implement another protocol, it should provide another plugin with “query” method and another transport implementation, for example using UCX¹⁰.

In C++, such approach is not applicable, because C++ does not allow to add methods dynamically into object interfaces as in Python or Javascript. In C++ case, API-wide methods should be predefined and then connected to implementations located in plugins using some kind of internal routing (e.g. internal messages).

Other plugins, which introduce some specific non-API-wide features, in turn, should be implemented as a software libraries, e.g. using orthogonal method of software coupling, which connects to PPK externally and manually by a user program.

Currently concrete step-by-step examples are under development.

7 Related Works

One case of plugin-based approach in HPC may be observed in ADIOS2 programming environment [10]. It suggests a concept of engines, which in turn a method for communication which may be implemented in different ways. Another option for plugins in ADIOS2 is a selection of methods of compression and decompression of transferred data. That is, one can see that here plugins become specific, and one may consider different types for plugins: for data transmission and for compression.

Another case may be observed in the HDF5 Virtual Object Layer (VOL). It is an abstraction layer within the HDF5 library that enables the use of different storage backends and data management approaches while maintaining the HDF5 data model and API for applications. Essentially, it allows HDF5 applications to interact with various

⁹ https://github.com/pavelvasev/ppk/blob/main/client-api/python/ppk_query.py#L47.

¹⁰ <https://openucx.org/>.

underlying storage systems or data layouts without requiring significant code changes. The VOL sits between the HDF5 public API and the actual storage mechanism. HDF5 API calls are intercepted by the VOL and forwarded to a specific “object driver” or “connector” plugin.

Various plugins are developed for VOL. One example is the HDF5 Asynchronous I/O VOL Connector¹¹ [11] which purpose is, as it names implies, an asynchronous data operations using HDF5 API. Another example is the LowFive plugin [12] which allows to interconnect different parallel tasks by network using HDF5 API. On top of LowFive, the Wilkins [13] insitu visualization and workflow systems is built.

Finally, an interesting approach is implemented in Logos platform [14]. As opposed to other systems like ADIOS or PPK, Logos API doesn’t provide “put” and “get” operations for data. Instead, it provide “transfer of control” operation. During that operation, data may be both exported or imported to/from environment of computing module, according to external (workflow) configuration. Additionally, Logos allows to specify so called “user functions” and “processing functions”. These are additional algorithms applied on data that is being exported or imported. A computation consist of a set of parallel computing tasks. Data connections between tasks and data processing functions are specified in a workflow. A sets of processing functions, in terminology of current article, may be considered as plugins which are attached to workflow. A user attaches plugin and activates it’s operation on specific data links, for example for compressing data during transfer on that link, or use specific protocol, so on.

8 Conclusion

The transition of the Parallel Programming Kit to a plugin-based architecture represents a significant advancement in its development trajectory. This architectural shift offers a flexible and extensible framework that facilitates the implementation of new features in a standardized manner.

Concrete plugins and system joint points (e.g. system’s API) for plugins are under development now. An overview of our current ideas are provided in “Plugin-based approach” section. As it shown in “Related works” section, some system already benefit from plugins, for example applied to data transport and data processing means.

Our mainstream idea is that final computation could be described as a workflow in some simple language like XML, YAML or even Python. Such approach is used, for example, in Wilkins [13] and Logos [14] systems. Beside describing parallel tasks and their interconnections, a user should be able to specify a list of plugins to load. Then, she could be able activate them for specific points in workflow.

Authors suggest that plugin-based approach will bring desired benefits, and make Parallel Programming Kit technology usable for various computational needs, extendable by users for users.

The authors would like to thank the reviewers for their detailed reviews of this work.

¹¹ <https://hdf5-vol-async.readthedocs.io/en/latest/index.html>.

References

1. Moreland, K.: The tensions of in situ visualization. In: IEEE Computer Graphics & Applications, pp. 5–9 (2016). <https://doi.org/10.1109/MCG.2016.35>
2. Vasev, P.: A Computational Model for Interactive Visualization of High-Performance Computations. In: Voevodin, V., Sobolev, S., Yakobovskiy, M., Shagaliev, R. (eds.) Supercomputing. RuSCDays 2023. Lecture Notes in Computer Science, vol 14389. Springer, Cham. https://doi.org/10.1007/978-3-031-49435-2_9
3. Vasev, P.: Analyzing an Ideas Used in Modern HPC Computation Steering. 2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBREIT), pp. 1–4. Yekaterinburg, Russia (2020). <https://doi.org/10.1109/USBREIT48449.2020.9117685>
4. Ahrens, J., et al.: An image-based approach to extreme scale in situ visualization and analysis. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14), pp. 424–434. IEEE Press, Piscataway, NJ, USA (2014). <https://doi.org/10.1109/SC.2014.40>
5. Vasev, P.A.: 3D Visualization of Asynchronous Many-Task Scheduling Algorithm. Scientific Visualization J. **15**(4), 92–111 (2023). <https://doi.org/10.26583/sv.15.4.08>
6. Vasev, P.A., Porshnev, S.V.: The scheme method for efficient execution of task graphs. Parallel Computing Technologies – 19th Proceedings of the International Scientific Conference, PCT'2025, Moscow, Russia, April 8–10, 2025. Publishing of the South Ural State University, Chelyabinsk (2025). <https://doi.org/10.14529/pct2025>. (in Russian)
7. Vasev, P.A.: Grafix – a Networked Visualization Technology. Scientific Visualization **17**(1), 97–113. <https://doi.org/10.26583/sv.17.1.08>
8. Hoare, C.A.R.: Communicating sequential processes. Prentice Hall. Updated edition edited by Jim Davies (2022)
9. Buyko, K., et al.: On some technological issues of LuNA active knowledge system implementation. Bull. Novosibirsk Comp. Center. Ser. Computer Science. Iss. 48, pp. 13–32. NCC Publisher, Novosibirsk (2024)
10. Klasky, S., et al.: ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management, SoftwareX **12** (2020). ISSN 2352-7110. <https://doi.org/10.1016/j.softx.2020.100561>
11. Tang, H., Koziol, Q., Byna, S., Ravi, J.: Transparent Asynchronous Parallel I/O using Background Threads. IEEE Trans. Parallel Distrib. Syst. **33**(4), 891–902 (2021). <https://doi.org/10.1109/TPDS.2021.3090322>
12. Peterka, T., et al.: IEEE International Parallel and Distributed Processing Symposium (IPDPS), St. Petersburg, FL, USA **2023**, 985–995 (2023). <https://doi.org/10.1109/IPDPS54959.2023.00102>
13. Yildiz, O., Morozov, D., Nigmatov, A., Nicolae, B., Peterka, T.: Wilkins: HPC in situ workflows made easy. Front. High Perform. Comput. **2**, 1472719 (2024). <https://doi.org/10.3389/fhpcp.2024.1472719>
14. Naduev, A.G., Cherevan, A.D., Lebedeva, A.S.: Architecture of the logos platform software module. Problems of atomic science and technology. Series: Mathematical modeling of physical processes. **4**, 55–63 (2022). https://doi.org/10.53403/24140171_2022_4_55. (in Russian)