Parallel Computing Technologies 2025

October 6-10, 2025 Almaty, Kazakhstan

Al-Farabi Kazakh National University

A Plugin-based Approach for Parallel Programming Kit

Pavel Vasev, Sergey Porshnev

Institute of Mathematics and Mechanics named after. N.N. Krasovsky
Ural Branch of the Russian Academy of Sciences
Ural Federal University

Ekaterinburg, Russia

Application

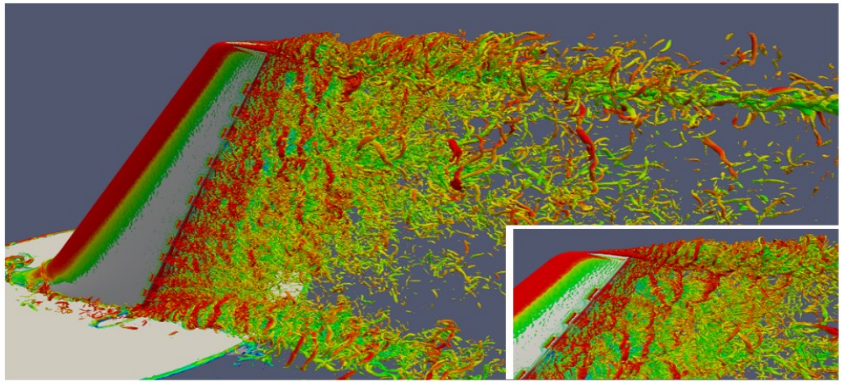
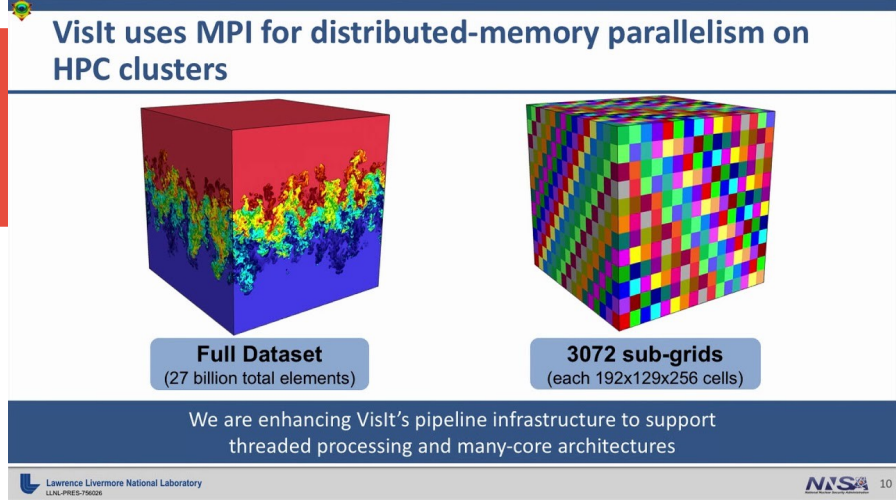


Figure 2: *Catalyst was utilized to contour two separate PHASTA quantities that are used for fluid flow computation and analysis, wall distance and Q-criterion, and generated Q-criterion contours colored by velocity magnitude. Inset shows a zoomed view of the wing tip. Image source: Rasquin & Jansen, UC Boulder*

Online visualization - is a visualization of parallel computation. It is performed during computation, so it is online.

Often it is interactive. It is considered to send control signals and other data to computation process.

Current problems



Computations become huge (exascale).

1. To visualize huge data, a parallel visualization is required.

2. There is actually no way to transfer data to specialized visualization nodes. The only possibility to visualize them online is insitu, e.g. on computing nodes, maybe using tight coupling. Or as closer as possible.

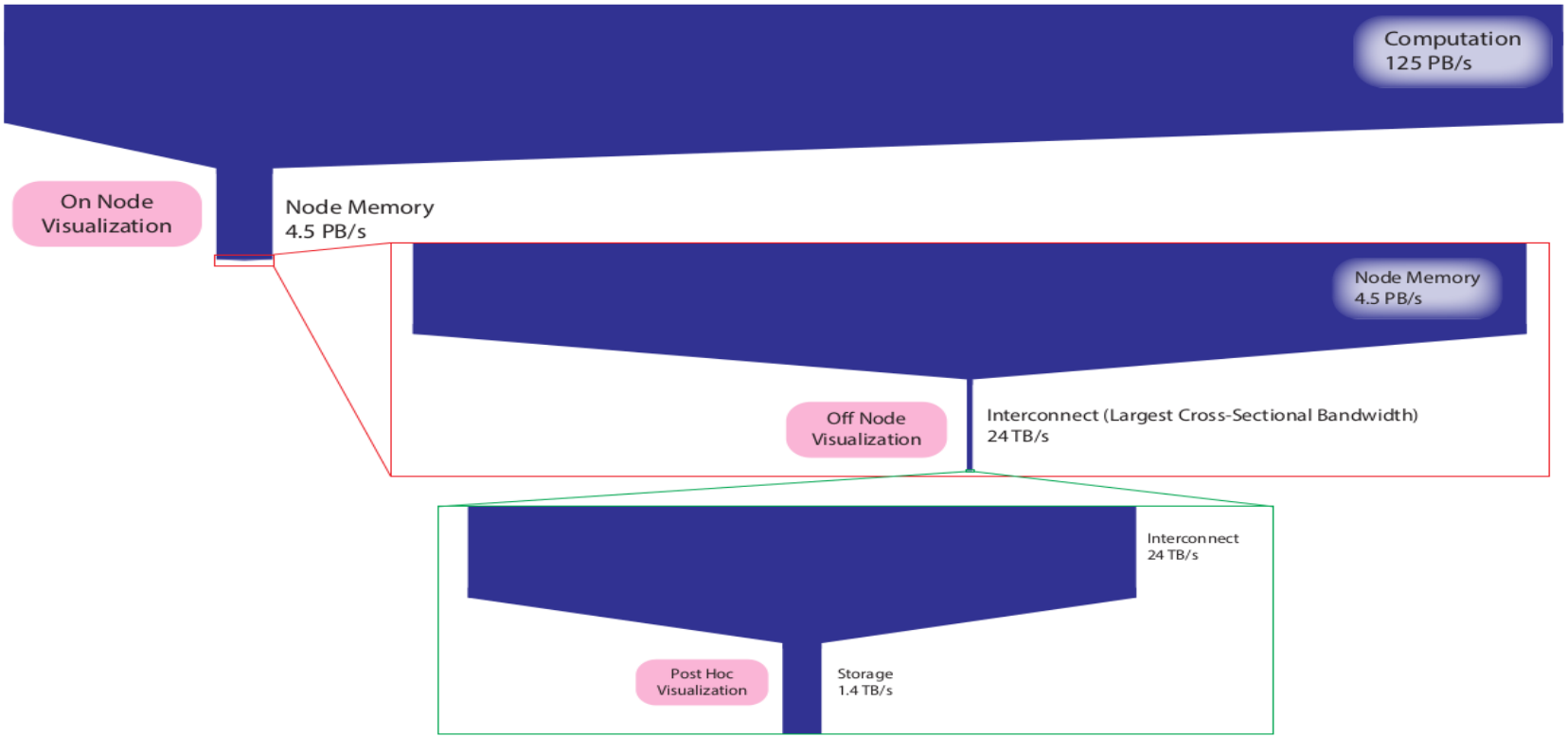
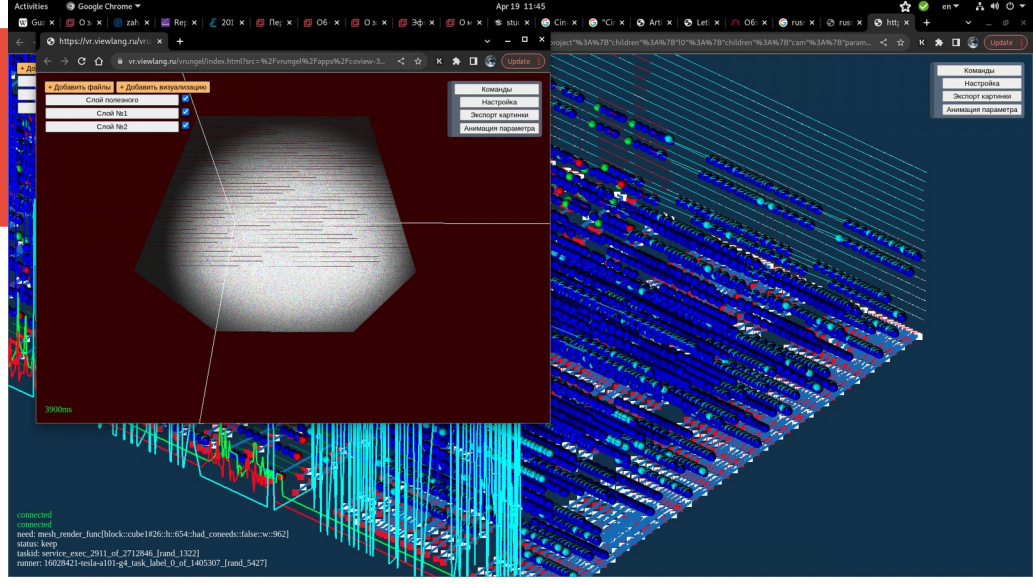


Figure 1: *A plot of the relative bandwidth of system components in the Titan supercomputer at the Oak Ridge Leadership Class Facility. The widths of the blue boxes are proportional to the bandwidth of the associated component. Multiple scales are shown to demonstrate the 5 orders of magnitude difference between the computational bandwidth and the storage bandwidth. Image source: Moreland.*

Our goal



Develop a technology for

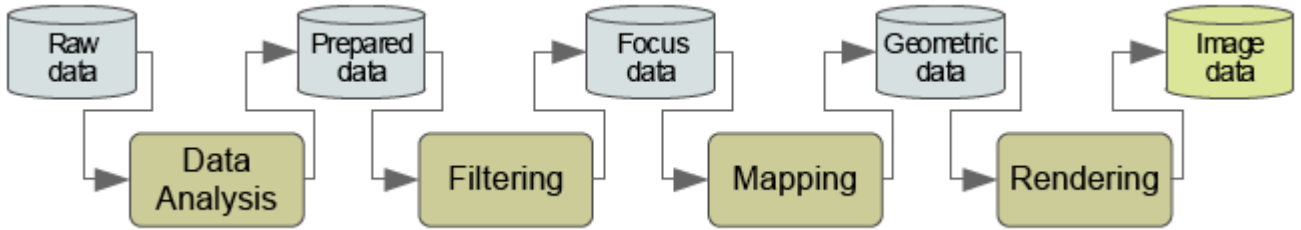
- online visualization,

- with capability of parallel visualization,

- with capability of insitu visualization.

Solution

Considering that visualization is just a computation itself,



and considering that we need tight code coupling for insitu mode,

we may just develop a universal parallel programming technology and thus accomplish our goal.

Approach

We love “Task graph” concept.

Task is a code sequence that:

- performed only once

- atomic in some sense

- have concrete inputs and an output.

Tasks are combined in graphs. This graph may be static or dynamic.

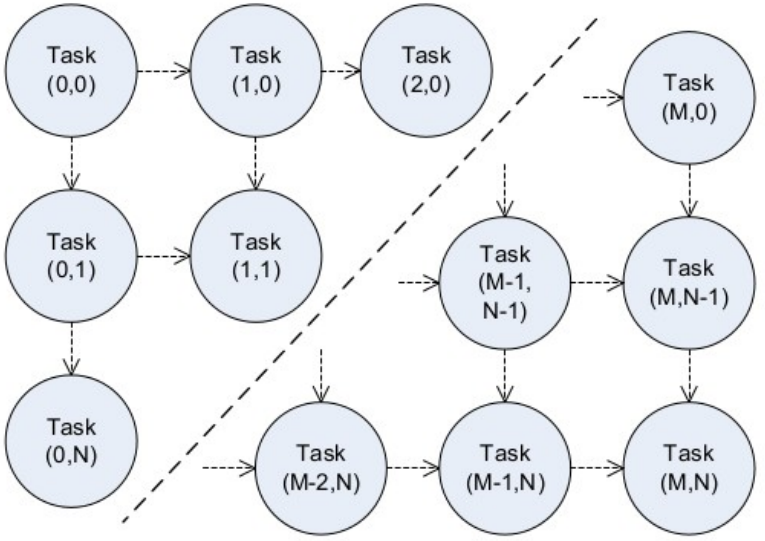
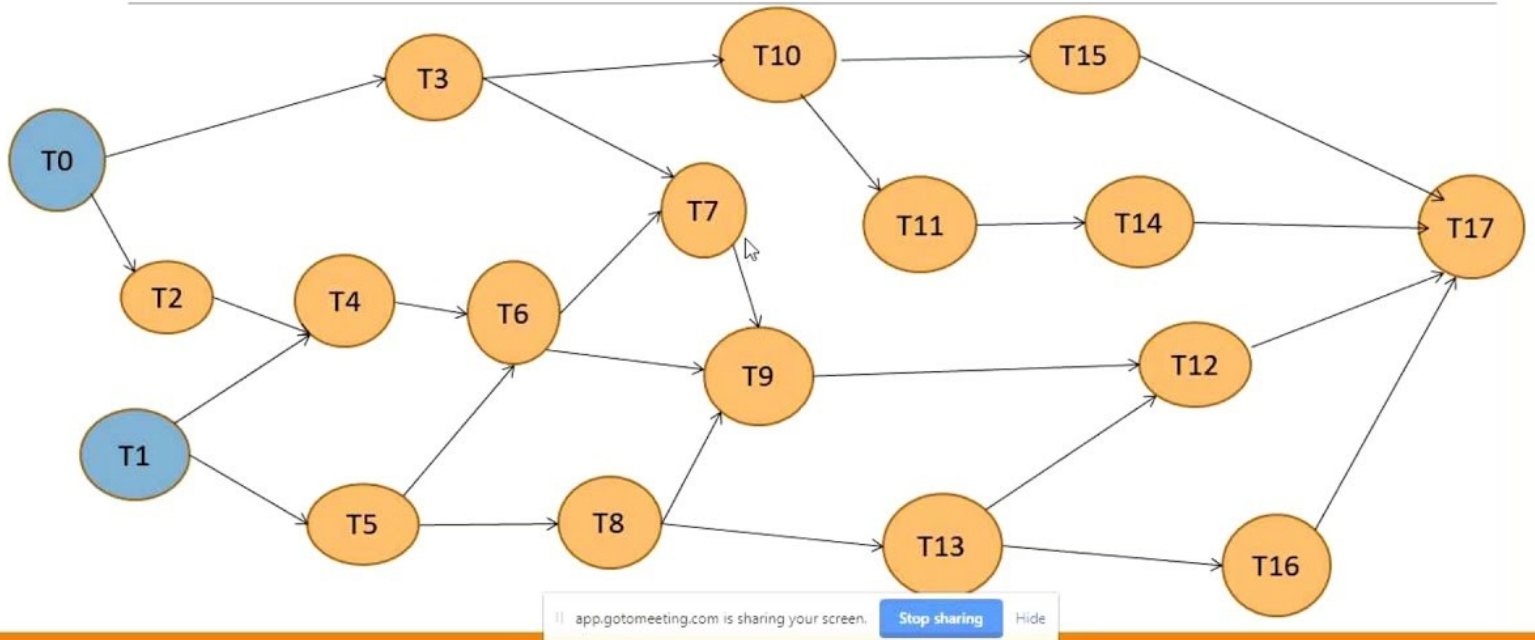


Figure 3: A dynamically defined task graph with a known dependency pattern.

MIRABILIS design

Defining a Task Graph



A Precedence Task Graph

Task graphs

So computation is a parallel evaluation of tasks of some task graph.

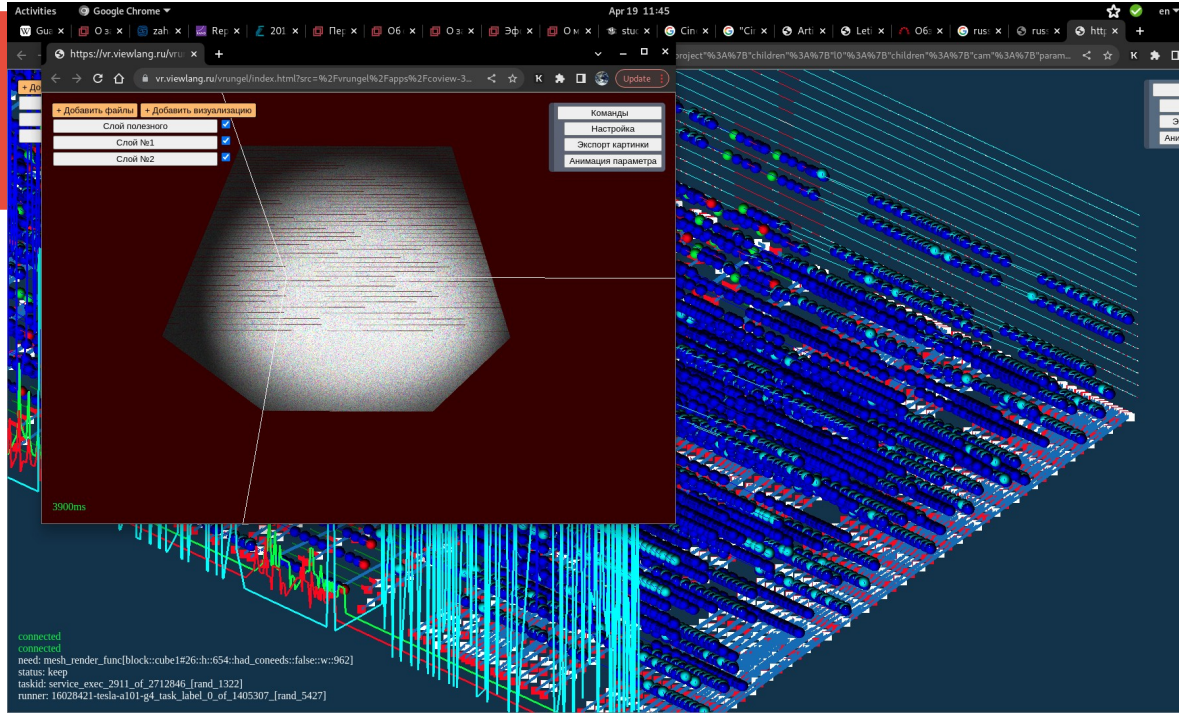
Task graphs are developed in early era of computing (for example Kotov, Narinyani, Asynchronous processes and the computer memory, 1966).

Considered at various scales: from scientific workflow systems to cpu (instruction level parallelism).

Task graphs are suitable for human mind.

Task graphs have good implementation capabilities. For example allows dynamic scheduling and resource allocation.

Task graphs are coupled good with Future/Promise concept.



Task graphs are suitable for online visualization. E.g. to see some point of view, just submit some subgraph.

But. In case of small tasks, a special attention is required because overhead of task transfer between nodes become relatively huge, and computation become ineffective.

This is solved by providing distributed task graph generation algorithms.

Which in turn, moves us to other concepts, for example concept of parallel distributed processes..

So, task graphs is not the basis.

Seems messages concept is the basis. But in messages, in our opinion, the most important part is message routing concept.

Thus lets forget for some time on task graphs and turn to messages contept..

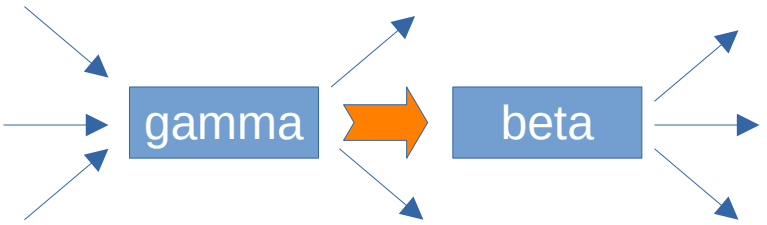
Finally together with tasks graph concept we will find them very strong.

Basic model. Entities:

- **Channel** = a message transfer method
- **Message** = atomic entity transferred in channel.
- **Environment** = global space of channels and their names

Operations:

- **Open channel**: name, read|write → ch
- **Send message**: ch, msg → none
- **Subscribe**: ch, code → undo_handler
- **Bind**: name1, name2 → undo_handler
  (e.g. create link)

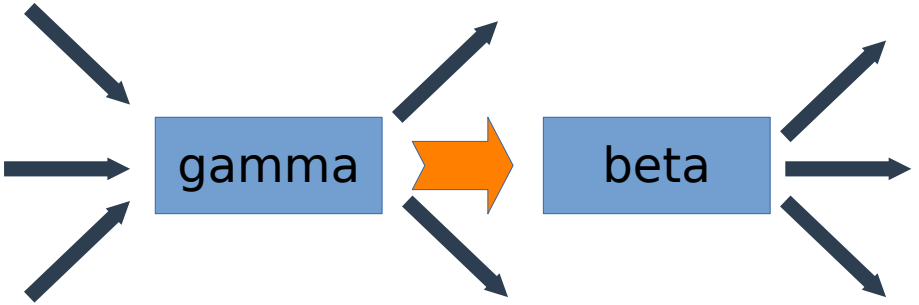


Important details

Direct message passing without brokers (using meta-broker).

Message routing via links right on sender. Thus global links does not involve networking if not needed.

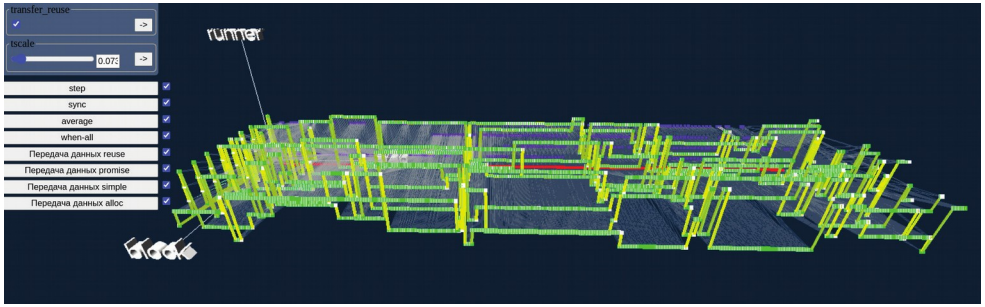
Sending of pointers instead of data in local passing. For example if sender and receiver are in same process, no need to copy data.



Implementation is simple

- Environment broker process, unique to computation.
- When someone subscribes to channel A, it sends information to broker: I need messages from A, my endpoint is U.
- When someone send message to channel A, it communicates with broker and gets the list of endpoints {U} and sends the message to all of them. Such communication with broker is performed only once.
- If someone later subscribes to A, broker sends updates of list {U} to active senders. So any actual message passing is done without broker.
- Same for links.

Have we just reinvented MPI?



Channel name = MPI tag + comm …

Send = MPI send …

But:

Receive is always <u>asynchronous</u> (via subscribe). Moreover, we see that subscribe callback code is EQUAL to task code.

This is very important theoretical result! Each callback code defines a task in a task graph sense. So we know what is blocking “MPI receive” now: this is just syntax sugar for tasks! Same for “await” in modern languages ;-)

<u>Links</u>. In MPI, as we know there is no concept of links, e.g one cannot send message to one communicator and receive it in another one. So no hooks, no external management..

Applications

1. We created a message sending API and framework, in C/Python/Js, and found it very hardware effective.

2. But we found it not usable for programming for human mind, and created task graph API and framework on top of it.

3. And then we found such task graphs hardware ineffective, and developed distributed processes that generate task graphs.

4. And then we see that all is vanity. And we need some stronger concepts. One of such concept is changeability (according to Darwin).

Plugin approach

A system is a space of local channels (hooks points, joint points).

A plugin is an arbitrary code that loads and subscribes to some of these channels (installs hooks). It also may provide new channels.

It then and receives local messages, and perform actions in response to these messages.

An action may even 1) change message for ongoing plugins, 2) stop message propagation, 3) send message to other channels.

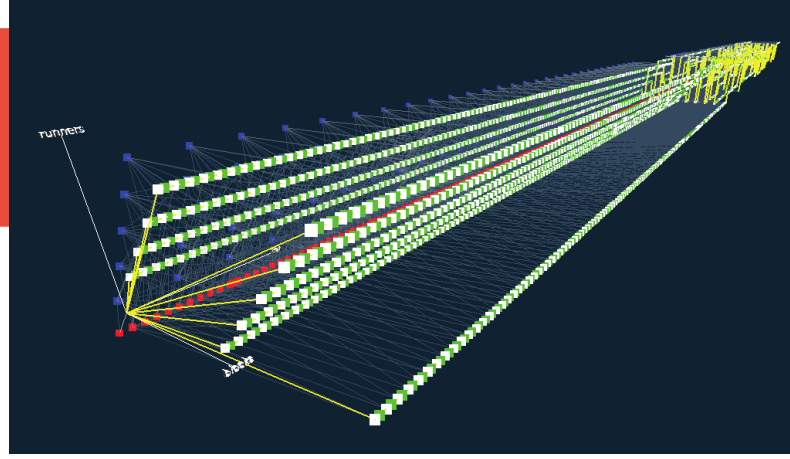
This allows to extend systems with new features with very modular way. It seems the best way to do it.

Example: Infiniband plugin

Our API have subscribe command. It is then implemented as
1) creating a server in subscriber process,
2) generating endpoint (URL),
3) sending it to broker,
4) client code connecting to subscribers endpoints and sending messages to them.

So to switch from TCP (current implementation), one have to implement hooks for 1,2 and 4 steps.

Example: Task graphs plugin



1. Create a task scheduler process,
2. Create promises tracking process,
3. Create standard worker process,
4. Provide a “submit task” and “wait promise” API functions.

The above looks like not a plugin, but like a library on top of API.

This is the point where we are now, trying to understand which things are suitable for plugins and which are not. And how to provision them to the final user.

Parallel Computing Technologies 2025

October 6-10, 2025 Almaty, Kazakhstan

Al-Farabi Kazakh National University

Thank you for your attention!

Pavel Vasev, Sergey Porshnev // **A Plugin-based Approach for Parallel Programming Kit**

Institute of Mathematics and Mechanics named after. N.N. Krasovsky
Ural Branch of the Russian Academy of Sciences
Ural Federal University

Ekaterinburg, Russia