Scientific Visualization, 2023, volume 15, number 4, pages 92 - 111, DOI: 10.26583/sv.15.4.08

3D Visualization of Asynchronous Many-Task Scheduling Algorithm

P.A. Vasev[1]

N N Krasovskii Institute of Mathematics and Mechanics of the Ural Branch of the Russian Academy of Sciences

[1] ORCID: 0000-0003-3854-0670 , vasev@imm.uran.ru

Abstract

The paper is devoted to the issue of visualization of the algorithm for scheduling parallel tasks. Task scheduling is a key part of the online visualization and parallel programming environment developed by the author. When programming a parallel version of one mathematical application, a suspicion arose that the scheduling algorithm does not optimally distribute the load between the workers. In this connection, it was decided to visualize its work in order to see the overall picture and possible problem areas of the algorithm. The constructed view successfully coped with the problem, and the scheduling algorithm was improved.

Keywords: software visualization, high-performance computing, asynchronous many-task systems.

1. Introduction

Visualization of the structure and operation of algorithms belongs to the field of software visualization [1]. It, along with scientific visualization and information visualization, is included in the science of computer visualization [2].

This work is devoted to the issue of visualizing the operation of the parallel task scheduling algorithm. Task scheduling is a key part of many high-performance computing systems, including the online visualization and parallel programming environment developed by the author [3].

The environment is built around the idea of representing a parallel computation process in the form of a set of dependent tasks [4]. Task dependency means that the input arguments to a task are determined by the computational results of other tasks.

In our environment, the stream of tasks and dependencies between them is determined by a user-specified algorithm, which can take the form of either a final computation or a process. Tasks stream is an input to the environment's scheduling process, which determines on which compute node a particular task should be executed. Then, as dependencies are resolved, tasks are executed. Naturally, the efficiency of the entire parallel computation depends on the quality of the scheduling algorithm used.

The correspondence of a task to a node will be called *an assignment*. We will call the set of such assignments and dependencies between tasks *an execution plan*. The execution plan, we note, is built over time, as tasks arrive and other aspects emerge.

The purpose of this work is to propose a view (visualization method) of the execution plan. This goal is determined by the need to understand how effectively the scheduling algorithm works and how it manifests itself in certain conditions. This need, in turn, turned out to be due to one practical problem, the solution of which raised suspicions that the used scheduling algorithm was not optimal.

The structure of the work is as follows. Section 2 introduces the parallel programming environment. Section 3 presents the applied problem and its parallel version. Section 4 presents the scheduling algorithm and requirements for its visualization. Section 5 presents the

constructed type of display and conclusions on the resulting visualization. Section 6 presents a modification of the scheduling algorithm and its visualization. In section 7 advantages and disadvantages of the view are discussed. Section 8 represents related works. In conclusion, future work ideas are highlighted.

2. Parallel programming environment

The author develops programming environment for solving problems of online visualization [3]. It is published at https://github.com/pavelvasev/ppk page.

Online visualization is the visualization of supercomputer calculations while they are running. Unlike post-computation visualization, online visualization requires additional tools to communicate with a running parallel program in order to receive current data from it and send control signals [5, 6].

Recently, the volume of calculated data grown, and Exascale calculations are being carried out. A situation has arisen that the process of visualizing the progress and results of calculations must be performed in parallel mode, as well as the calculations themselves.

Note that technically, visualization itself is a computation (with some quirks, of course). Therefore, the following hypothesis was formulated: it is possible to propose a certain programming method that will make it possible to effectively implement visualization algorithms, their connection with parallel computing programs, as well as these programs themselves, that is, arbitrary computing algorithms.

As a test of this hypothesis, the created environment offers the following method of parallel programming.

The method is based on the concept of promise. A ***promise***[1] (also used the term *future*) is an object that corresponds to data that will be calculated someday. A promise can be created in one process, fulfilled (also used term *resolved*, that is, binded with data) in another process, and reacted to fulfillment in a third processes.

The convenience of promises lies in the fact that they can be operated at any time, even before the results of the calculations of these promises are achieved. <u>This allows to describe parallel data processing algorithms using ordinary sequential codes</u>.

The proposed parallel programming method is expressed by the following model.

Operation "add-data". Loads data into the computing environment.

add-data: data→ promise

where *data* is the data (e.g. some block in operating memory of node or its device) to be loaded into the system. The result is the promise object, which is in the "fulfilled" state. Data *data* is copied to the computing environment.

Operation “add-task”. Adds a request to calculate a task.

add-task: action, args→ promise

where *action* is the action to perform, *args* are the action arguments. The result is a promise object that will be executed when the given task is evaluated. Thus, a task is defined by an action and the arguments of that action. Arguments are a set of (name, value) pairs.

It is important that promises can be specified among the argument values. The computing environment will perform the specified task 1) when all the promises specified in the arguments are fulfilled [2], and 2) as hardware capabilities become available to perform the task.

Operation "get-data". Loads data from the computing environment.

get-data : promise → data

---

1 The concept of a promise and the list of operations with promises are described, for example, in the Javascript language documentation, section Using Promises. Also Wikipedia article: Futures and promises.

2 In general, it is better when the argument in the form of a promise is the only one. This corresponds to mathematical models of computation such as category theory. This approach is used, for example, in the C++ stdexec library. However, the current implementation is more concise by specifying all expected promises in the task arguments (which is an implicit "when all" operation).

where *promise* is the promise object, data is the data obtained as a result of fulfilling the specified promise. Note that this operation does not return the data itself, but a local promise in terms of the programming language being used. When *the promise* is fulfilled, the local promise will also be fulfilled.

The model also implements additional operations for working with promises:

- *create-promise: void → promise* - creates a promise object.
- *resolve -promise: promise,data → void* - resolves the specified promise object, binding it to the specified data.
- *when-all: list → promise* - creates a promise object that will be fulfilled when all promises from the given list are fulfilled. The result of fulfilling such a promise is a list of the values of the fulfilled promises, in order corresponding to the list of promises.
- *when-any: list → promise* - creates a promise object that will be fulfilled when any one promise from a given list is fulfilled. The result of a promise being fulfilled will be the value of the first promise fulfilled.

It is assumed that using the above operations it is possible to implement a wide class of computational algorithms. A computational algorithm is believed to describe the computation process primarily by adding tasks to the environment.

If some tasks are independent of each other in terms of data, then their parallel execution is possible. Even in case of parallel execution, such computation is **determined**. Also note that the computational algorithm **remains sequental**.

Incoming tasks are passed on by the environment to processes called *runners* (in other systems, terms *worker* and *executor* are often used). When selecting a runner for the next task, that is, when scheduling a task, the environment evaluates the state of the runners and their workload. The overall architecture and the task scheduling algorithm are described in more detail in Section 4.

Additionally, to implement interactive processes, the environment offers interaction by sending and receiving messages according to the publish-subscribe model with filtering messages by topic. Computation participants subscribe to message topics. Other participants send messages to certain topics. When one sends a message to a topic, it is delivered to all those participants who have subscribed to that topic.

An interesting question is how applicable the presented method of parallel programming is. It is assumed that this method can solve both problems of parallel visualization (and in particular parallel rendering) and problems of a broader class.

3. Application and its parallel algorithm

To test the applicability of the proposed programming method, it was decided to create a parallel version of the algorithm for the application described in [7]. The application implements a method for approximating the dynamics of a nonlocal continuity equation with a nonlinear Markov chain.

The original (non-parallel) version of the application algorithm, was developed and provided by it's author Yu. V. Averboukh.

The solution is carried out in the one-dimensional case. A visual image of the solution is presented in Figure 1.

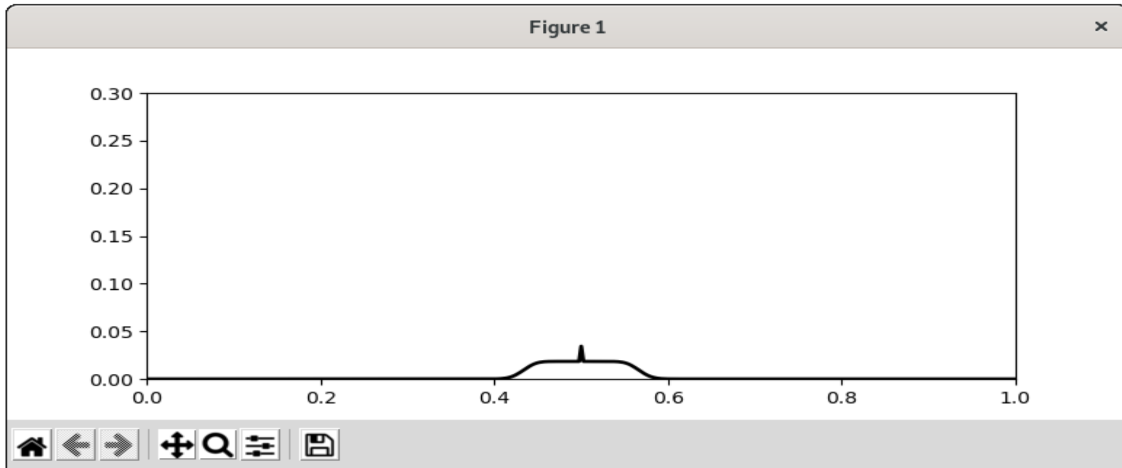


Figure 1 is a visual image of the solution computed by application with N=400, T=4000, see [7].

The structure of non-parallel version of the algorithm is as follows.

1. The processed data is represented by an array *mu* of length *N* with elements of *double64 type*. The initial value of all elements *mu* is some constant.
2. A cycle of *T* iterations is launched, where *T=10*N*.
3. At each iteration, for each element *mu[i],* the next iteration's value is calculated as a function of the form

f(mu[i-1], mu[i], mu[i+1], expectation)

where *mu[i-1], mu[i+1]* are the "neighboring" elements *of mu[i]* , and *expectation* is a *double64 number*.

4. At the end of each iteration, a new *expectation* value is calculated as a function of the form *g(mu)*, that is, a function of the entire *mu* array.

A parallel version of the algorithm based on the parallel programming method presented in section 2 and is the following.

1. Data array *mu* of length *N* is split into *P* pieces of equal size (the set of these pieces is hereafter referred to as a "*P-partition*"). Each part is loaded into the environment through an "add-data" operation. As a result, a list of promises *p_mu* of length *P* is formed, and each such promise contains an associated *block value* - an array of elements of the corresponding part.
2. A cycle of *T* iterations is launched, where *T=10*N.*
3. At each iteration the following is performed:

3.1. For each *p_mu[j]* a computation task is started. It calculates elements in the same way as in the non-parallel case. The task takes the following arguments as input:

- *block* – the value of the part elements.
- *expectation* – calculation parameter.

The result of the calculation is written into the promise $p_mu_{unsynced}[j]$ and it contains:

- *block* – *new* value of the part elements.
- *left* – the value of the leftmost element of the part.
- *right* – the value of the rightmost element of the part.
- *average* – is a special value that will be used later when calculating the *expectation* of a given iteration.

3.2. For each $p_mu_{unsynced}[j]$, the task of synchronizing the boundary values of its part is started. This task takes the following arguments as input:

- *outer_left = p_mu[j-1].right* – the value of the rightmost element of the left part,
- *outer_right = p_mu[j+1].left* – the value of the leftmost element of the right part,

- *block = p_mu[j].block* – own data block

and inserts the value *outer_left* into the left boundary value of the data block, and *outer_right* into the right boundary value. As a result of launching all synchronization tasks, a set of *P* promises is formed, which is written to *p_mu* over the old values.

3.3. Additionally, an *expectation* calculation task is added, which takes as input
- *array = when_all(p_mu* $_{unsynced}$*)*

that is, a promise that is triggered when all tasks are solved *p_mu* $_{unsynced}$ in current iteration. The task reads the *average* value from all *P* results and using them calculates the final *expectation* value, which will be used at the next iteration.

Thus, the entire calculation is expressed through *T*(P*2+1)* tasks connected to each other by inputs and outputs. A non-parallel version of the algorithm was written in Python. The parallel version was implemented on the basis of the non-parallel one, also in Python.

The results of measuring the performance of the algorithms are shown in Table 1.

Table 1 – Application performance

Parallelization method	Millions of operations per second
Non-parallel version	**~370**
With automatic parallelization across threads (16) and shared memory numba.njit(parallel=True)	~986
Parallel version P=10 parts and W=5 runners	**~500**

By operation we mean the calculation of one value in the *mu* array. All measurements were carried out with the application parameter $N=4*10^7$ on a machine with a Ryzen 1700x processor. All versions of the programs are optimized using Numba technology with nopython, fastmath parameters and the SVML library.

Thus, the resulting parallel version proved to be faster than the non-parallel version. But it was inferior in performance to thread-based parallelization. The question arose as to what could have caused such a lag.

The following hypotheses were put forward:

When working in multi-threaded mode with shared memory, **it becomes easier to synchronize boundary values.** On shared memory, before starting a block calculation, one can read the boundary values directly from memory, and then use them as a calculation parameter. On distributed memory, boundary values must be transmitted to runners explicitly, via network or local interprocess communication protocols, which entails a corresponding **delay**.

Overhead of assigning tasks to runners. These costs are expressed in the **delay**, which passes from the moment the next task is actually ready until the moment the task begins to be executed on some runner. In the non-parallel version, such a delay is minimal, due to the fact that the “task generation algorithm” and the algorithm of the task itself are placed as close as possible. Specifically, in the non-parallel version, the task formation algorithm is a loop operation over *T*, and the task algorithm is the body of this loop. In the multi-threaded version, the picture is similar, although somewhat more complicated.

Such delays can be reduced. For example, in the parallel programming environment under discussion, the architecture has been changed. Calculation of task readiness used to work before the planning process (only ready tasks were assigned to runners). After the change, calculation of readiness began to work on runners (thus unready tasks were also assigned). In this way, delays from the scheduling algorithm and from data transfer to and from it were eliminated.

A faster method of transmitting information between the processes of the environment was also used: previously messages were transmitted via HTTP with keep-alive, in the new

version they began to be transmitted via TCP (the OpenUCX was also considered, but TCP currently is enough).

Suboptimal planning. After task start delays were minimized as much as possible, the question arose as to how well the tasks were assigned to runners. This issue is discussed in more detail in the next section.

4. The scheduling algorithm and requirements for its visualization

A scheduling algorithm is an algorithm that chooses which runner should perform a particular task. In general, the environment's operation scheme is as follows:

1. Launching a user program implies launching an environment and a set of runner processes (optionally on a local machine or on a computing cluster).
2. The user program communicates with the environment and adds tasks to it.
3. When added, a task is transferred to the scheduling algorithm.
4. The scheduling algorithm evaluates the workload and status of runners, and assigns tasks to them. Assignment entails the transfer of information about the task to the selected runner.
5. Each runner has a replenished list of assigned tasks and monitors their readiness. The readiness of a task is the readiness of all the promises specified in its arguments. When ready, the task may begin to execute on assigned runner.
6. The task is executed in two stages. The first stage is preparing arguments. Arguments can be constants, promise references, or functions. In all cases, except for constants, the results of preparing arguments are entered into the local cache of the runner. This is a very important feature of the environment because it allows to reuse duplicate arguments across different tasks, which has a significant impact on performance. In particular, at the stage of preparing arguments, data corresponding to promises is loaded into the runner's RAM, possibly from other nodes of the computer system.
7. The second stage is the execution is task's action (e.g. algorithm specified for the task). The result of task, e.g. result of execution of task's action, remains in the runner's memory in the cache. Such result often contains data blocks (large structures in RAM). References to these data blocks are placed in the promise associated with the task. The task's promise enters "fulfilled" state.
8. Upon completion of tasks and as a consequence of their promises, other tasks become ready, which is determined in step 5, and the computation flow continues.

It is important to note the following detail. If the result of a task contains large amounts of data, this data itself is left intact in the runner's RAM, as noted at step 7. When some other task needs this data, it is either a) accessed directly if the task is running on the same runner, or b) transferred to the memory of process of other task runner, for example, copied over network protocols.

This approach allows to minimize the number of actual data transfers between tasks, and allows to perform without such transfers altogether. On the other hand, it requires keeping track of available RAM (of the node or its GPU devices, if the data is located in them) and transferring data as memory is exhausted from runner processes to special storage processes.

However, sometimes such transfer can be avoided due to the following feature. In practice, there is often no need to duplicate data blocks. For example, in the application under consideration, described in the previous section, the results of calculating the next iteration part of *mu* can be written over of the data of the "old version" of this part.

To take this feature into account, a special task argument conversion function *reuse:promise→ promise* has been introduced into the environment, which transfers (but not copies) blocks of RAM data from the promise to the responsibility of the target task, and the promise itself is erased (deleted from the environment). Target task can use these data blocks

as both input and output. In the latter case, the task can produce its result on these same blocks. This eliminates duplication and unnecessary data transfers – RAM blocks are reused.

At the same time, if the data blocks were placed on another runner, then they are transferred to the memory of the task's assigned runner, and removed from the memory of the original runner.

In connection with the above-mentioned features of the organization of the computing process, the requirements for the quality of work of the task scheduling algorithm become obvious. Such an algorithm should carry out an even load of tasks on runners, while trying to minimize the number of data transfers between runners, and at the same time ensure that the local runner caches are used as optimally as possible.

Planning algorithm. The algorithm process receives tasks as input. Each task is described by a tuple of the form *(action, args, resources),* where *action* is the action identifier, *args* are the arguments, *resources* are the requirements for the runner regarding the presence of devices and (their) RAM.

Among the arguments there may be some that should be "computed" and left in the local cache. For example, load data blocks corresponding to promises, or compile function code, etc. All such argument values are provided with a globally unique identifier, which allows to distinguish between values and place them in the local cache of the runner. The list of identifiers for such arguments is designated below as $needs_{task}$.

Also, the algorithm receives information from each runner:

- $queue_size_{runner}$ – amount of pending tasks of runner (e.g. size of the list of assigned and currently unresolved tasks),
- $needs_{runner}$ – the state of the local cache of runner, in the form of a list of identifiers.

Algorithm step.

1. Tasks are assigned to runners one by one and individually, as they are received.
2. For the next *task* for each runner, an estimate is made using the formula:

$$est = missing_needs(task,runner) + queue_size_{runner} * qcoef \quad (1)$$

where $missing_needs = | needs_{task} \setminus needs_{runner} |$ – the number of task arguments that are missing from the runner cache, and the second part takes into account the "load" of this runner. Coefficient value $qcoef = 0.1$.

3. The task is assigned to the runner with the lowest *est value* .
4. The $needs_{runner}$ variable is enriched with the values of the $needs_{task}$ and also the ID of the task itself. This means that in subsequent calculations the algorithm will consider these values as existing in the runner cache, although in fact they will only get into the cache when the task is completed.

The presented scheduling algorithm is therefore not an optimal assignment algorithm, like the Hungarian algorithm. Instead, it tries to schedule incoming tasks quickly, as they arrive, in some kind of a "smart" way.

It was decided to investigate how the presented algorithm schedules tasks to the application described in Section 3. It was decided to use visualization for the study. The following **visualization requirements** were put forward:

1. The main task of visualization is to provide an understanding of whether there are errors or suboptimality in the behavior of the algorithm, and ideas for improving the algorithm.
2. It is necessary to see the distribution of tasks among runners over time.
3. In this case, it is necessary to distinguish tasks by partitions (parts of *mu*) in order to understand exactly how the load is distributed between the runners.
4. It is necessary to separately see the moments of data copying between runners, since data movements are assessed as the most costly process, leading to delays in calculations.
5. It is necessary to see the dependencies of tasks on promises in order to check the correctness of the visualization itself.

Of course, this statement was not developed immediately; it began with the main task, and subsequent points were refined over time. The constructed visualization (or rather, the type of display) is presented in the next section.

5. The view

Based on the requirements stated above, the following view was constructed, an example of which is presented in Figure 2.

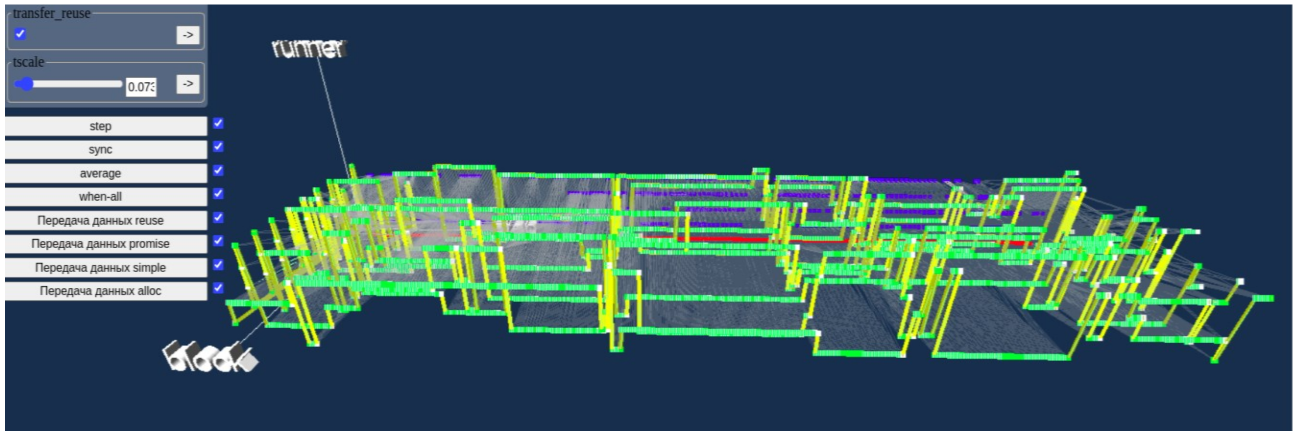


Figure 2 – visualization of the operation of the task scheduling algorithm. The user has the ability to move in 3D space, dive inside and study the details, change the scale of the time axis.

The view is built in three-dimensional space by the following rules:

- *X* axis (from left to right in Figure 2) corresponds to the logical time of the algorithm of the application, namely the iteration number.
- *Y* axis (from bottom to top in Figure 2 - runner) corresponds to a runner ID (e.g. serial number of a runner).
- *Z* axis (from far to near in Figure 2 - block) corresponds to the serial number of the part from the *P-partition*.
- Each task is represented by a dot, the color of which determines the type of task. White – compute *mu*. Green – synchronization of boundary values. Blue – calculation of *expectation*. Red – the "when all" synchronization operation used in calculating *expectation*.
- Dependencies between tasks are shown by lines (segments) going from one task to another. Since tasks are displayed in logical time, task dependency is interpreted on the logical time axis, a task later in time depends on a task earlier in time.
- Thin lines show dependencies with the transfer of “lightweight” data, for example, when the promise corresponds to data in the form of a small set of numeric constants.
- Bright, thick lines show dependencies with the transfer of "heavy" data, such as memory blocks from the *P-partition*.
- The lines going from the origin of coordinates are the loading of the initial data.
- The view can be scaled in time, for which a scale parameter has been introduced, which can be changed using the slider in the graphical interface.

The view shows an important characteristic of data transfers. Each bright thick line passing along the *Y* coordinate means data transfer between runners. The fewer such lines, the fewer data transfers occur during the calculation.

A visual analysis of the parallel task execution plans showed the following:

- It became obvious that the selected application can be implemented without transferring data blocks between runners at all. Such a transfer is justified if some runner works slower than others, but this information is not available at the planning stage in the current implementation of the scheduling algorithm.
- **Problem 1**. It turned out that at the initial phase all parts of the *P-partition* are planned onto single runner, then a series of part transfers to other runners occurs, and only then the balance is found, see Figure 3. This, although logical, is not entirely optimal behavior.

- **Problem 2**. It turned out that during work there are many unnecessary part transitions between runners. So, it can be seen that the runner solves a series of iterations of a certain part, and then this part is reassigned to another runner. What entails “imbalance” and mass transfer of other parts between runners, see Figure 4.
- In general, the conclusion was the following. The current scheduling algorithm is not adequate for the application; it obviously entails unnecessary data movements.

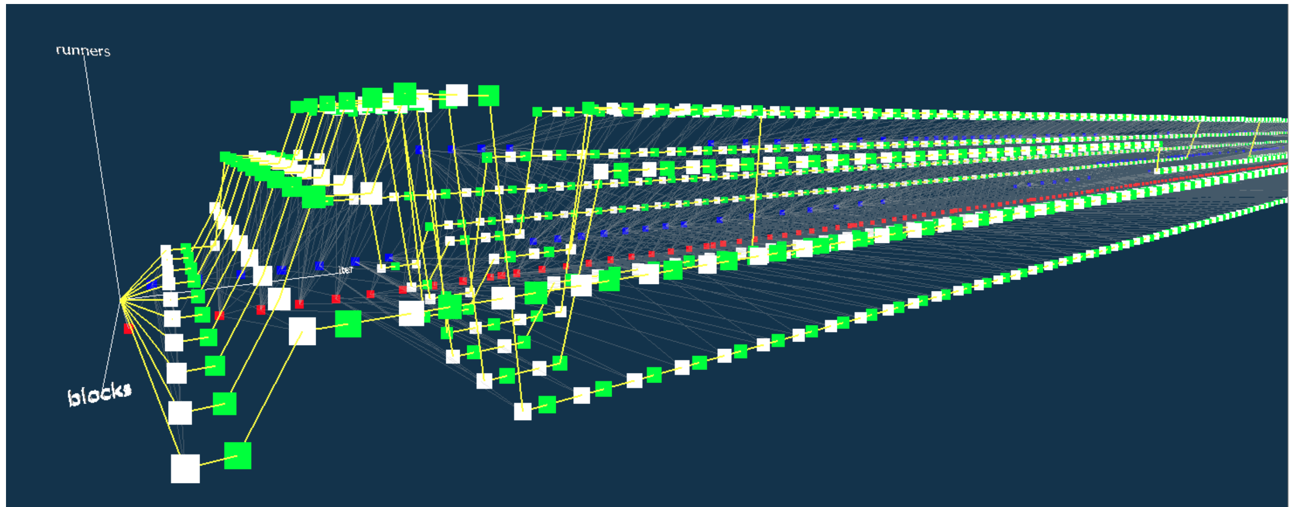


Figure 3 is a visual image of problem 1: the initial behavior of the scheduling algorithm at the start of calculations is shown, a strange rebalancing is visible - a “jump” of the entire load between runners (vertical axis) instead of the expected uniform load.

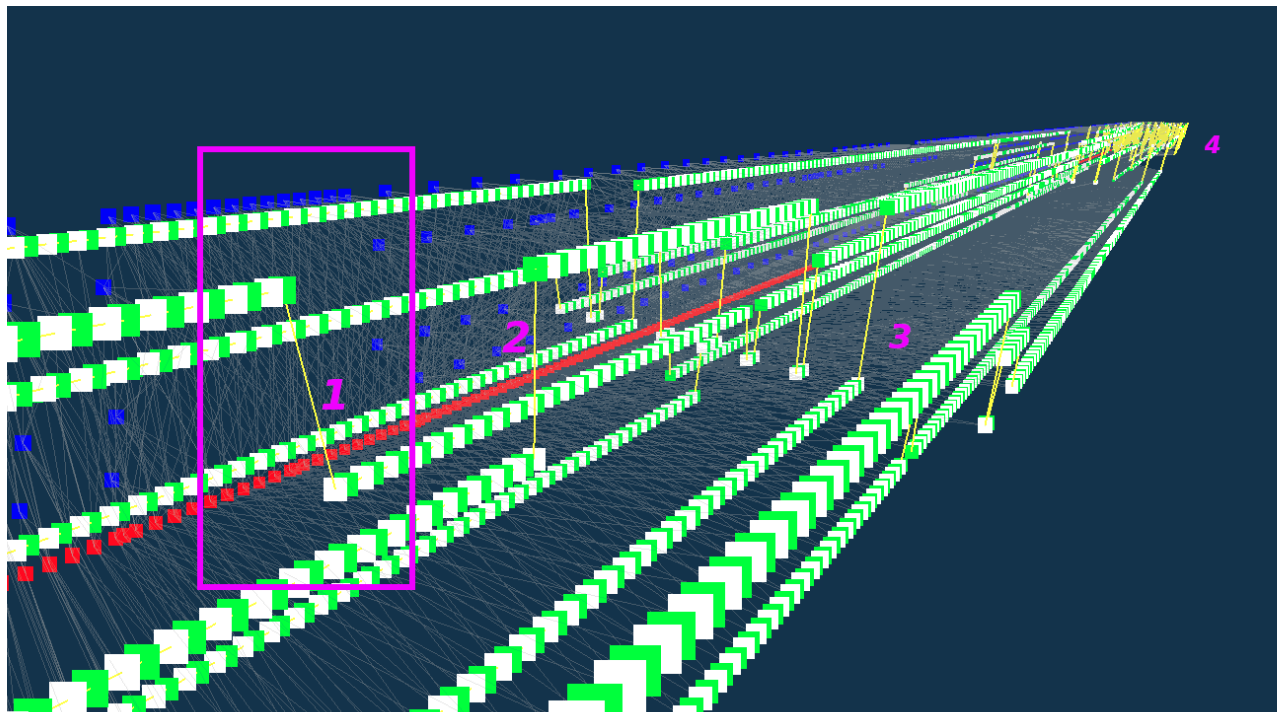


Figure 4 is a visual representation of problem 2: “unprovoked” load rebalancing. The purple rectangle (1) indicates the first point when the load on the part calculation was moved to another runner. What led to the subsequent “unloading” of this runner (2). After a few more iterations, this led to a complete rebalancing across all runners (3). A repetition of similar behavior of algorithm (4) is visible on the horizon.

6. Modification of the scheduling algorithm

A search was carried out for the causes of the identified problems identified using the constructed visualization. Among the assumptions, the following were confirmed.

Causes of the problem 1. As it turned out, the problematic behavior is caused by the fact that the scheduling algorithm, when assigning tasks, along with the presence of data blocks in runner's RAM, also takes into account the compilation of program codes. The task codes are supplied as a special argument, and the "cost" of preparing such an argument was the same as the "cost" of transmitting data for the task (see the *missing_needs formula*). Therefore, the runner who received the very first task assignment, received a super-advantage over other runner - after all, he had "already compiled" the code for the data processing function. This continued until the size of that runners's task queue exceeded a certain size according to the *qcoef* coefficient, after which the advantage was leveled out and tasks began to be assigned to the next runner, who also began to receive a temporary super-advantage.

Solution. In principle, the behavior of the algorithm is "logical", and the problem could not be solved. An ideal solution would be to introduce weights when calculating the cost of deploying task arguments. As an experiment, a weight of 0 was introduced for arguments for compiling function codes. The result is shown in Figure 5.

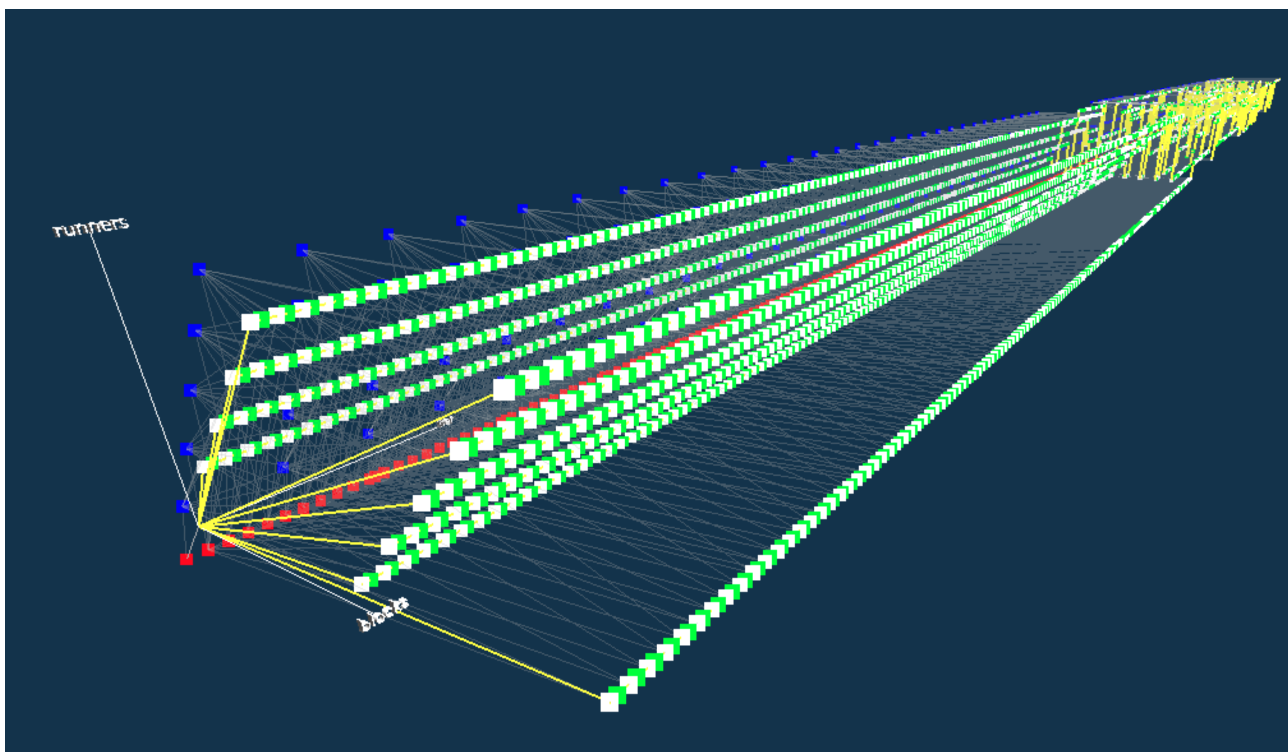


Figure 5 – the solution of problem 1. Tasks are distributed evenly among runners from the very beginning. However, in subsequent iterations, problem 2 with excessive rebalancing is still visible.

Causes of problem 2. It turned out that the reason lies in the abrupt change in information about the sizes of queues of runners tasks *queue_size*. The implementation of the system is as follows: each runner sends information updates on its state (including the size of its *task queue*) periodically, several times per second. At the same time, the scheduling algorithm also models the size of this queue, increasing by 1 counter each time a task is assigned. But he never reduces it himself, but waits for information from the runners. It turns out that the next time we receive information, it "turns out" that this runner has already solved a number of problems, and his queue is significantly shorter than others. And a shorter queue of tasks for an runner, based on formula (1), gives an advantage to this runner when assigning tasks.

The size of these jumps turned out to be such that tasks were "removed" from the runner who even had already deployed blocks in RAM, and transferred to a "less busy" runner. Of course, after a short time there were updates from other runners, but the appointments had already been made. The actual queue length hopping numbers were of such an order that a runner queue was reduced from, for example, from 1614 to 1572 tasks. The difference 42 with *qcoef=0.1* is equal to 4.2 and "dragged" task assignments to this runner.

Solution. Among the solution options, the following was chosen. It was decided to "smooth out" jumps in information on queue sizes of runners using a logarithm, see formula (2):

$$est = missing_needs(task,runner) + \ln(queue_size_{runner}) * qcoef \quad (2)$$

where $missing_needs = | needs_{task} \setminus needs_{runner} \setminus needs_{compile} |$ this is the number of task arguments missing from *the runner's* cache and excluding code compilation tasks $needs_{compile}$, and the second part takes into account the "load" of this runner and smoothes out jumps in updating information in the memory of the scheduling algorithm. Parameter *qcoef=0.1*.

The result of solving the problem is shown in Figure 6.

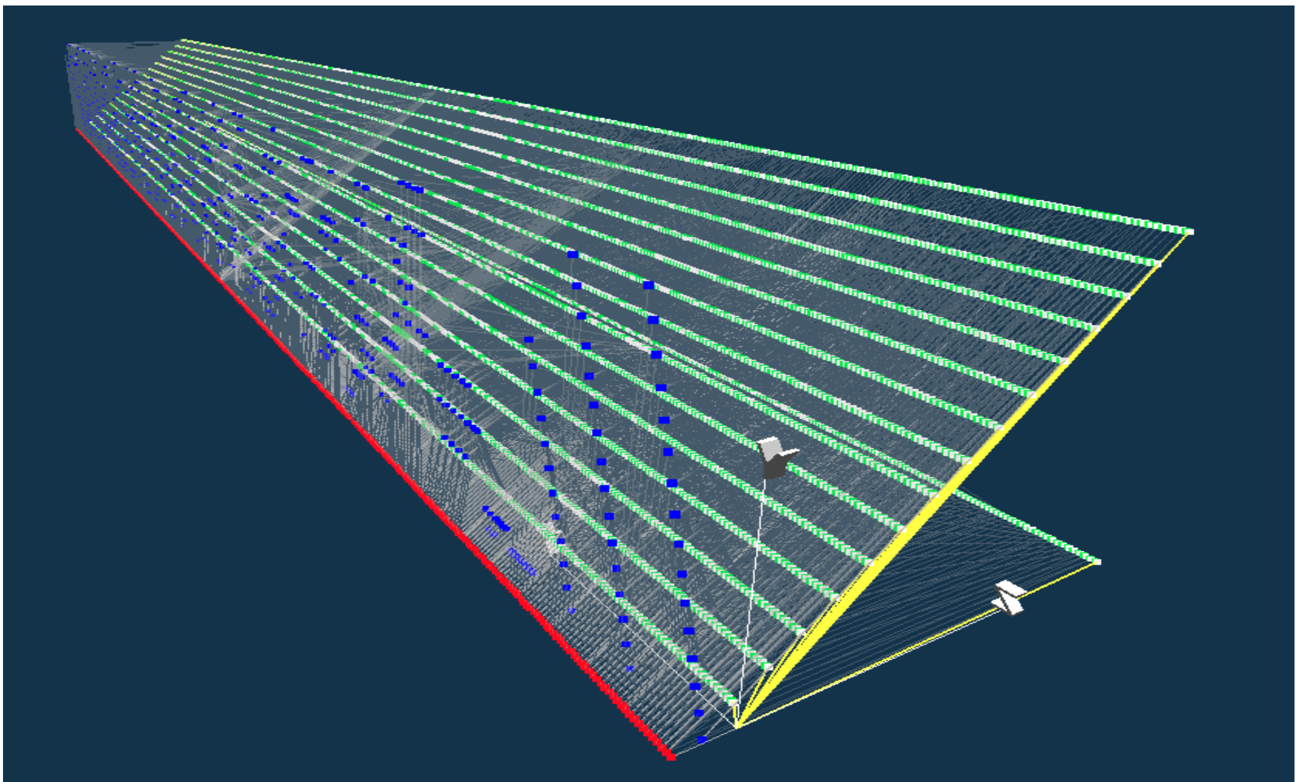


Figure 6 – the solution of problem 2**.** It shows 700 iterations with 16 parts and 16 runners. It is visually observed that tasks of computing *mu* parts are distributed among runners evenly throughout the whole calculation. The assignment of *expectation* calculation tasks (blue dots) is uneven, but this does not affect the calculation performance, since these tasks do not require the transfer of large data blocks.

The chosen solution to problem 2 is controversial and "frontal". Good results are initially observed, but the solution needs additional testing and reflection. For example, when trying to remove the coefficient *qcoef* from the formula under the assumption that the logarithm could completely smooth out the jumps, this led to the scheduling algorithm again starting to "play up", i.e. transfer responsibility over parts of the *P-partition* from runner to runner. It is also necessary to check how this formula behaves not only in iterative, but also in interactive tasks, for example, in online visualization tasks.

The updated performance information is as follows:

Table 2 – Application performance

Parallelization method	Millions of operations per second
Single threaded version	**~370**
With automatic parallelization across threads (16) and shared memory numba.njit(parallel=True)	~986
Parallel version P=10 parts and W=5 runners	~ 500
Parallel version with new algorithm P=10 parts and W=5 runners	**~ 670**
P=10 parts and W=10 runners	~740
P=16 parts and W=16 runners	~630

By operation we mean the calculation of one value in the *mu* array. All measurements were carried out with the application parameter $N=4*10^7$ on a machine with a Ryzen 1700x processor. The values $P=10$ and $P=16$ were chosen based on the number of processor cores and convenience: if N is divisible by P without a remainder, then the parts of P-partition are of equal size.

The new version of the algorithm (or rather, the new formula for estimating runners) showed an acceleration from 500 to 670 million operations per second. This was achieved by eliminating unnecessary data transfer between runners caused by suboptimal task assignments.

After the paper was finished, the performance indicators were increased to 1388 million operations per second (at P=4, W=4) due to the following modification of the application's parallel algorithm:

- Boundary values synchronization task is combined with the *mu* computation task (added to the beginning).
- The calculation of *average* has been embedded into loop of *mu* computation (before, it was a separate cycle).
- The task of calculating *expectation* based on the values of *average* values of parts is combined with the *mu* computation task (added to the beginning). Thus, calculating *expectation* is duplicated in each part, but the final performance indicators turned out to be better than waiting for *expectation* to be calculated as a separate task.

With a similar optimization (calculating *average* in a loop of calculating of *mu*) in a version with automatic parallelization across threads, its performance was 1250 million operations per second.

Currently, there is no more problems visible in the scheduling algorithm.

7. Discussion

We highlight advantages of the presented view as following:

- The view coped with its main task – to show the overall picture of task scheduling and help visually detect existing problems.
- These problems were eliminated, which was confirmed by performance estimations and visually using the view.
- The ability to scale the picture by logical time turned out to be convenient and made it possible to study the situation both in general and in detail.

Disadvantage of the presented view are following:

- Among the main costly elements of calculations are transactions of synchronization and data transfer between runners. The view successfully visually highlights data transfers – the lines when changing the Y coordinate are clearly visible. However, the length of such lines is perceived as a characteristic of "high cost", "distance" of information transmission, but in fact it is only equal to the difference in the ID number of runners.

• For view operation, it is required to enumerate data partition blocks and to assign these enumeration numbers to created tasks. This allows to place tasks among Z-axis. This is additional work for programmer.

During working with the view, additional question appeared: isn't it overcomplicated? Maybe there was enough to use simpler view, for example by showing tasks by colored points on 2D plane.

Such a view may be constructed by following:

• Each task is represented by a dot on a plane, where X axis used for logical time and Y axis for block ID or runner ID.

• The dot is colored using runner ID if Y axis used for block ID, and vice versa.

Additionally, we want to distinguish types of tasks (e.g. computation, synchronization, so on). To visualize this, for example, tasks types might be represented by different glyph types of dots. In common, as it is shown in work [8], to visualize data on dot-based representations at least following visual aspects might be used: position, color, opacity, shininess, size, glyph type, glyph orientation.

8. Related works

The task scheduling area presented in the paper considers high-performance computing. The presented environment might be related to area of task-based parallelism and asynchronous many-task systems (asynchronous many-task programming models). Neighboring areas are DAG scheduling algorithms, task graphs, dataflow scheduling.

Classical view of task graphs. Task graphs are directed acyclic graphs (DAG). For these graphs, there exist straightforward representation with nodes and directed edges between them. Figure 7 depicts widely used views of task graphs.

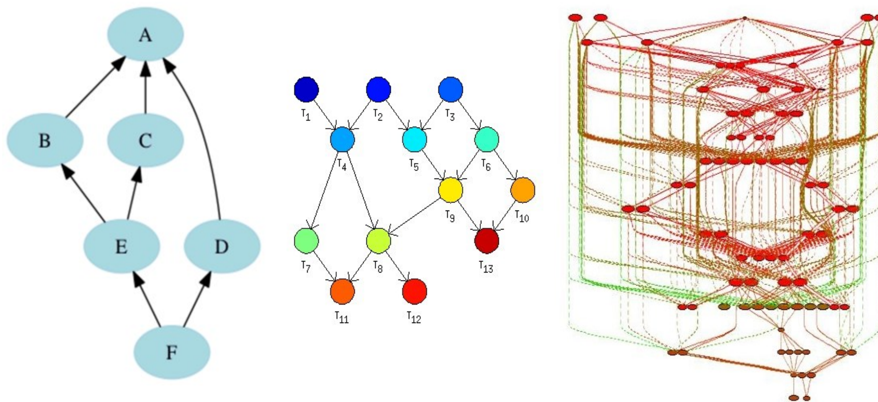


Figure 7 – classical views of task graphs. Each task is represented by a node. Dependencies between tasks represented by edges. **Left** – simplest graph [9]. **Middle** – color used to depict types of tasks [10]. **Right** – color used to depict types of dependencies [11].

The problem with classical view is that it is suitable for graphs of small size. Real task graphs may contain millions of tasks. The image shown in Figure 7, right, avoids this problem with the following trick: show only parts of task graph, suitable for investigation, which is noted in [11].

Task graphs without dependencies. In Figure 8, a task graphs are shown without representing dependencies. Tasks are grouped into 3 big classes, according to algorithm stage, and each class associated with specific color. This view seems better suits for showing large-scale task graphs. For example, in Figure 9 the same problem shown, but with much larger tasks count.

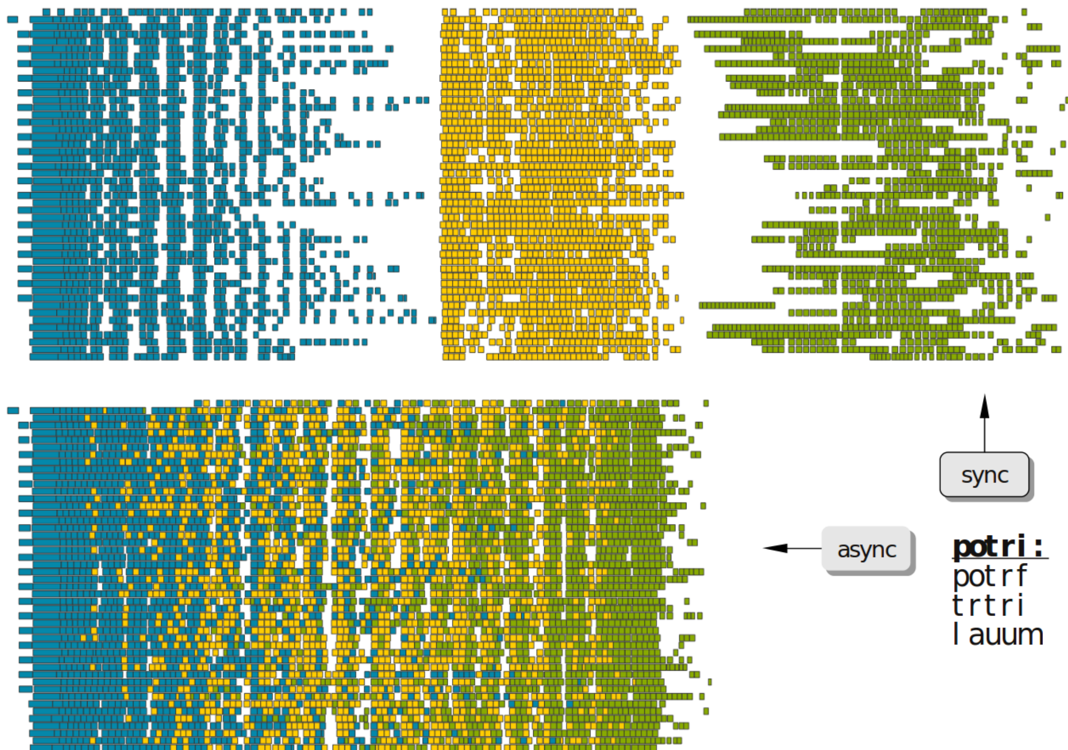
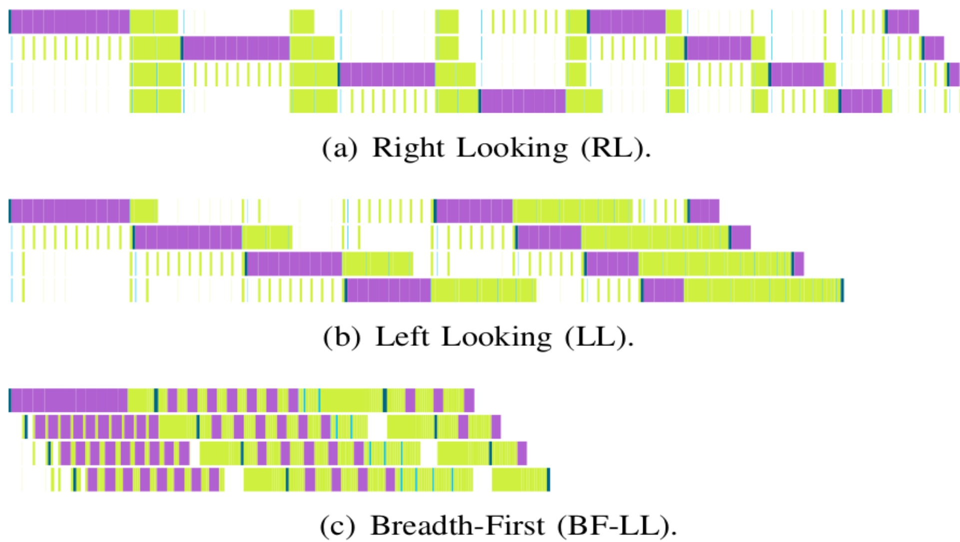


Figure 8 - POTRI (POTRF, TRTRI, LAUUM) algorithm with and without synchronization barriers, courtesey of the PLASMA team [12]. Also explained as bulk synchronous + node-level AMT (top) and holistic AMT (bottom) [7]. Same as with view presented in current paper, the view on this figure uses X axis for time (logical or physical). Also it requires some kind of enumeration to place tasks among Y axis (for example by runners or by data blocks).

Figure 9 – same problem shown as on Figure 8, but with much larger amount of tasks [13]. Tasks are shown by "pixels" and grouped by computing nodes.

Depicting time used by tasks. In Figure 10, tasks are shown with bars. The wider the task's bar, the longer time was occupied by task.



(a) Right Looking (RL).

(b) Left Looking (LL).

(c) Breadth-First (BF-LL).

Figure 10 - traces of three computations are shown, depicting various approaches for task distribution [14]. Tasks of different types have different colors. The “width” of the task denotes its duration in time.

Resource utilization. Figure 11 shows not tasks, but the utilization of hardware resources which executes these tasks. Thus a task graph is visualized from perspective on how it performs.

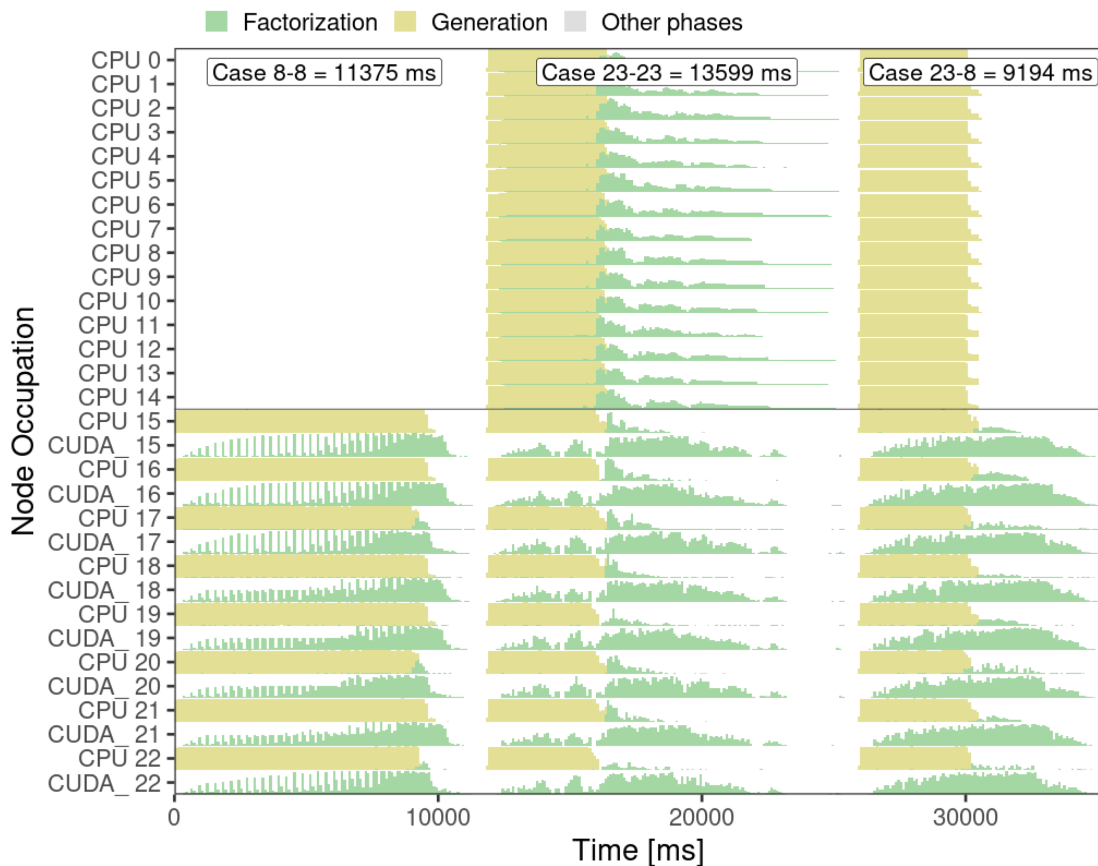


Figure 11 - depicts three iterations of ExaGeoStat package where the x-axis is the time, and the y-axis has the aggregated resource type utilization per node. The different colors correspond to different phases: the yellow ones are the generation, while the green ones are the tasks the factorization, a small number of tasks in gray correspond to the other three phases [15].

Computation state. Same as in previous case, in Figure 12 one may see not tasks, but their aggregated layout on nodes and more interesting, the evolution of the picture of a computation state.

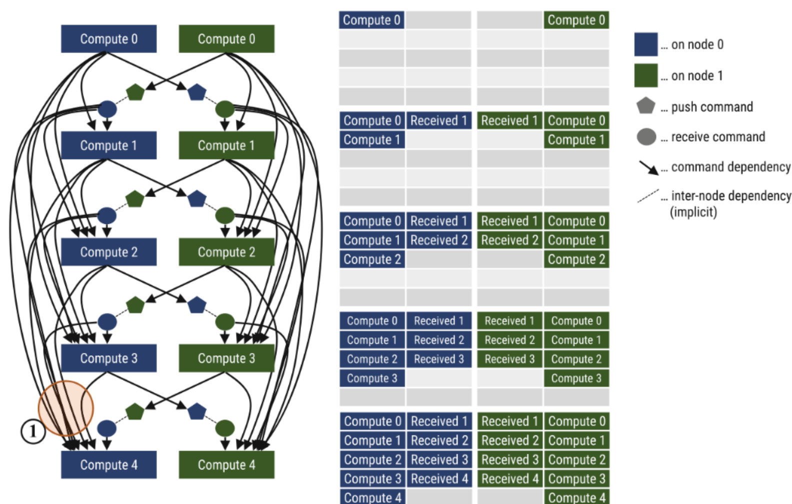


Figure 12 illustrates a simplified view of the command graph generated for the first five iterations of a computation scheduled on two nodes/GPUs. It includes compute commands, as well as data push and receive commands. As each row of the involved data buffer is generated by subsequent time steps, the number of dependencies in the command graph scales with the iteration count [16].

Computing nodes layout. In classical HPC computing, a set of nodes (or their parts) are assigned to a job, and this job might use them freely on it's behalf. Nodes are connected to each other using network links of some topology. This means that distance between nodes (measured for example in hop counts) vary. Thus actual selection of nodes for a job is crucial for it's performance. In Figure 13 the view uses spatial coordinates for representing supercomputer network structure. Actual nodes assigned for a job visible in the prism of their mutual connection distances.

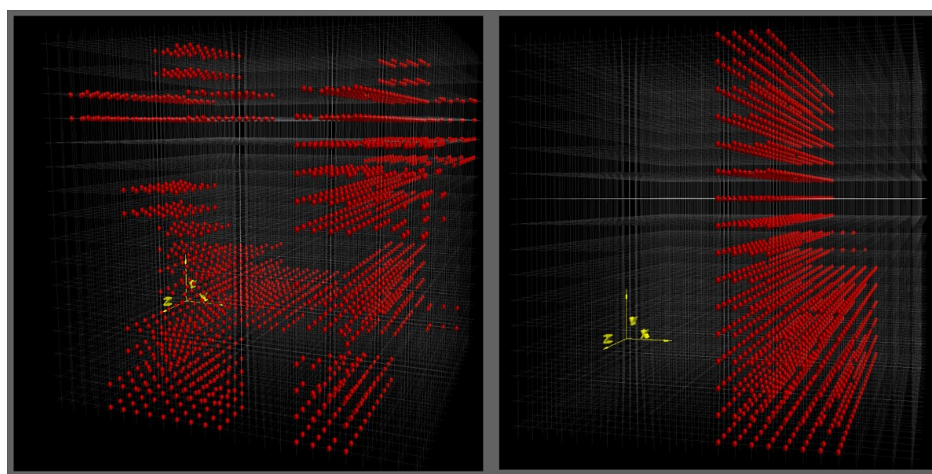


Figure 13 – visualization of runners layout on computing nodes of Titan supercomputer [17]. Red dot denotes assignment of runner to a node. Node position, highly likely, is determined by supercomputer network interconnect structure. Thus a set of red dots depicts layout of nodes allocated for some job on the supercomputer. **Left** – layout before optimization. **Right** – layout after optimization. It is visible that allocated nodes are closer together than on left variant. The closer the nodes are, the less network hops required for their communication.

9. Conclusion and future work

As it was noted in Discussion section, the presented view was succeeded to solve it's task. It was able to show problems in scheduling algorithm under consideration. During algorithm repairing, the view was used to control its development.

Prospects for the development of the view are as follows:

View scalability. The view has been tested in practice on a small number of "data blocks" and runners (up to 100). It would be interesting to test it on a larger number of entities. As it shown in related works sections, views showing a lot of tasks do exist.

See simultaneously both the task plan and its actual execution. The view of the constructed task plan shows the work of the scheduling algorithm. The actual execution shows how results of this algorithm are applied in reality. In Figure 14, the example image of such view is presented (it is preliminary, and the view is still under construction).

Actual execution of tasks may differ significantly from the planned one due to non-obvious but significant reasons. It seems interesting to view them simultaneously, for example side by side. However there is a caveat: the presented view of tasks plan uses logical time axis (e.g. iteration number), and actual execution view uses physical time axis. Solving this caveat and bringing two views together may provide an interesting insights in performance of parallel programs created in asynchronous many-task paradigm.

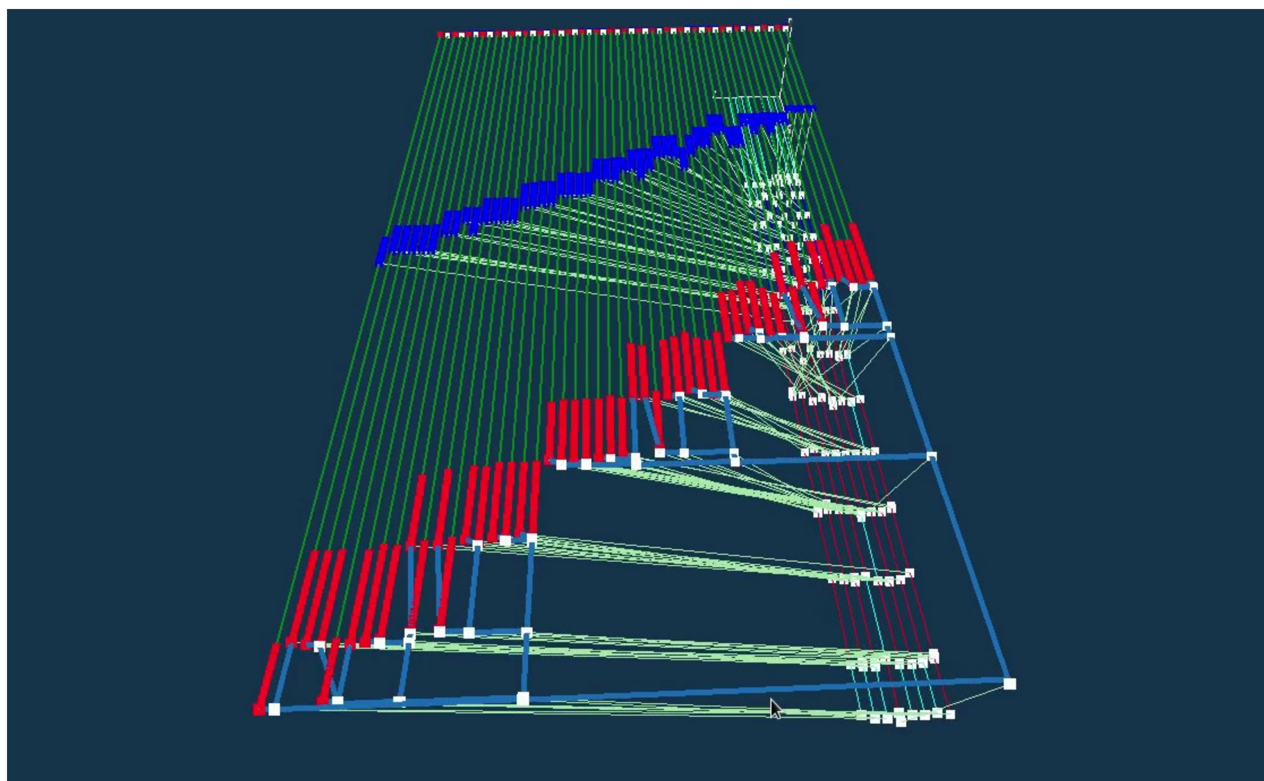


Figure 14 – the view of actual execution of tasks. Time goes from up to down. Lines are tasks, and their length depicts actual task duration. Some tasks look like dots because they are short. Color depicts task type. X-axis in a view corresponds to task's block number. Data dependencies between tasks is shown cyan lines. Green lines depicts wait time of task, from time when task was added up to time it was solved. Additionally, there are 8 runners shown on bottom plane, they are connected with with tasks that they perform by light-green lines. Animation is available: youtu.be/XnV3l8hw8QE.

10. Acknowledgments

The author is grateful to his colleagues from the Krasovskii Institute of the Russian Academy of Sciences and Ural Federal University: Yuri Vladimirovich Averboukh for providing the application; Sergei Vladimirovich Porshnev, Ilya Sergeevich Starodubtsev, Mikhail Olegovich

Bakhterev, Sergei Vladimirovich Sharf, Dmitry Valeryanovich Manakov for participation and support of the project.

11. References

[1] Averbukh V. L. et al, Metaphors for Software Visualization Systems Based on Virtual Reality // De Paolis L., Bourdot P. (eds) Augmented Reality, Virtual Reality, and Computer Graphics. AVR 2019. Lecture Notes in Computer Science, vol 11613. Springer, Cham. Pp 60-70. URL: https://www.cv.imm.uran.ru/e/3241731

[2] Averbukh Vladimir L., Semiotic Analysis of Computer Visualization // Chapter 6. Interdisciplinary Approaches to Semiotics. InTech. Rijeka, Croatia. Pp. 97-133. URL: https://www.cv.imm.uran.ru/e/3241679

[3] Vasev P.A., Model of an online visualization system // Parallel Computing Technologies (PaVT'2023): Short articles and descriptions of posters. St. Petersburg, March 28–30, 2023. – Chelyabinsk: SUSU Publishing Center, 2023. – P. 236. – EDN IJALIW. In Russian. URL: https://www.cv.imm.uran.ru/e/3241872

[4] Kotov V. E., Problems of development of parallel programming // Proceedings of the All-Union Symposium “Prospects for System and Theoretical Programming”. Novosibirsk, 1979. P. 58-72. In Russian.

[5] Ryabinin, K.V., Development of the adaptive multiplatform science visualization module for high-performance computing systems // Scientific Visualization, 2012, Quart. 4, Vol. 4, Num. 4, P. 17-29. In Russian. URL: http://sv-journal.org/2012-4/03.php?lang=en

[6] Vasev P.A., Analyzing an Ideas Used in Modern HPC Computation Steering // 2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology, Yekaterinburg, Russia, 2020, pp. 1-4, DOI: 10.1109/USBEREIT48449.2020.9117685.

[7] Averboukh, Y. Lattice approximations of the first-order mean field type differential games. Nonlinear Differ. Equ. Appl. 28, 65 (2021). DOI: 10.1007/s00030-021-00727-2

[8] Mikhailov, I., Averbukh, V.: Design and development methods for visualization multidimensional discrete data. // In Proceedings of National Supercomputer Forum (NSCF). The Program Systems Institute of RAS, Pereslavl-Zalesskiy, Russia (2017), NSCF-2017. Presentation in Russian. URL: https://www.cv.imm.uran.ru/e/3241829

[9] Jeremiah Wilke et al, Asynchronous Many-Task Programming Models for Next Generation Platforms, SAND2015-5106PE, 2015. URL: https://www.osti.gov/servlets/purl/1261059

[10] John K. Holmen, Damodar Sahasrabudhe, and Martin Berzins. Porting Uintah to heterogeneous systems. In Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '22). ACM, article 11, 1–10. https://doi.org/10.1145/3539781.3539794

[11] Jeremy Thornock Todd Harman John Schmidt, Uintah User Interface Tutorial, 2014. URL: https://www.slideserve.com/kris/uintah-user-interface-tutorial

[12] Emmanuel Agullo, PI, et al. Chameleon: A dense linear algebra software for heterogeneous architectures. URL: https://solverstack.gitlabpages.inria.fr/chameleon/

[13] T. Herault et al, "Composition of Algorithmic Building Blocks in Template Task Graphs," 2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM), Dallas, TX, USA, 2022, pp. 26-38, doi: 10.1109/PAW-ATM56565.2022.00008.

[14] Agullo, Emmanuel and Augonnet, Cédric and Dongarra, Jack and Faverge, Mathieu and Langou, Julien and Ltaief, Hatem and Tomov, Stanimire, LU Factorization for Accelerator-based Systems, 9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11). URL: https://hal.inria.fr/hal-00654193

[15] Lucas Nesi, Lucas Mello Schnorr, Arnaud Legrand. Multi-Phase Task-Based HPC Applications: Quickly Learning how to Run Fast. IPDPS 2022 - 36th IEEE International Parallel & Distributed Processing Symposium, May 2022, Lyon, France. pp.1-11. ffhal-03608579f. URL: https://inria.hal.science/hal-03608579/document

[16] Thoman, P., Salzmann, P., Command Horizons: Coalescing Data Dependencies While Maintaining Asynchronicity. In: Diehl, P., Thoman, P., Kaiser, H., Kale, L. (eds) Asynchronous Many-Task Systems and Applications. WAMTA 2023. Lecture Notes in Computer Science, vol 13861. Springer, Cham. https://doi.org/10.1007/978-3-031-32316-4_2

[17] Jonathan Hines, Moab scheduling tweak tightens Titan's workload, 2015. URL: https://www.olcf.ornl.gov/2015/10/13/moab-scheduling-tweak-tightens-titans-workload/