A computational model for interactive visualization of high-performance computations

Pavel Vasev[1][0000-0003-3854-0670]

[1] N.N. Krasovskii Institute of Mathematics and Mechanics of the Ural Branch of the Russian Academy of Sciences, Ekaterinburg, Russia
vasev@imm.uran.ru

Abstract. Interactive visualization of high-performance computations is important area in supercomputing. It assumes that visualization of results of computation is generated during computation process. However there is a problem: due to overwhelming size of data to visualize, a visualization program should be itself parallel and executed on supercomputer. Beside that, such program should allow to be changed dynamically, because visualization pipeline may change due to user steering of interactive visualization. Current mainstream frameworks for interaction with supercomputer programs assume usage of external parallel programming methods. In current paper, an original parallel programming model is suggested that have built-in capabilities for online visualization. At basic level, it is based on messages and reactions. At higher level it uses promises to simplify inter-operation of computation and visualization.

Keywords: Computational Model, Parallel Programming, Online Visualization, Insitu Visualization.

1 Introduction

Interactive visualization of high-performance computations is covered by online and insitu visualization areas. These are crucial areas in modern supercomputing. In some cases, it is impossible to achieve results of computation without them [1,3].

Online visualization is a process of interactive visualization of running computation [1]. **Insitu visualization** is a process of generating visual images of results during computation [2]. Whereas both terms are different, they are interconnected and have common aspect: use of supercomputer not only for computations, but also for visualization purposes.

Due to the fact that supercomputer power is be used, visualization [pipeline] algorithms have to be implemented in parallel form. Thus, to achieve online visualization of supercomputing, following tasks have to be solved:

1. Provide a way of interaction of visualization part and computation part.
2. Provide a way of parallel programming of visualization algorithms.

The first task is solved using various approaches, for example see [2]. A most common approach is to provide some data transmission service, and a library for interacting with it. Computing application is instrumented with calls for such library and thus data is offloaded from application into visualization processing. Example projects are:

ADIOS2, Ascent, Henson, Sensei, Damaris/Viz, Libsim, Paraview Catalyst, and others (see reviews [3,4]). However, all of these projects doesn't solve the second task – they don't provide any parallel programming models.

It seems that this is philosophically correct – e. g. a recognized paradigm is that "a given tool should solve single problem". They instead provide channels of data transmission and model of interaction of such channels. They moreover sometimes provide some kind of data endpoints compatible with visualization systems which are already parallelized. For example ADIOS2 may be interconnected with Paraview parallel visualizer using Paraview Catalyst / Fides technology.

By the way, this leads to very interesting configuration: a visualization pipeline in Paraview might be described as a set of interconnected actors both in Python code and in graphical user interface. This is on one hand is brilliant idea – because it is easily understandable by human. On other hand it leads to serious problems in computations balancing, due to design of Paraview's implementation.

However, there are not many such parallel visualization systems: Paraview, Visit, ScientificView, and it seems no more. In any case, even if there will exist enough quantity of them, the scientific progress should not be stopped and new ones should emerge.

In the current paper, the author suggest single solution that solves both stated tasks. The solution is proposed in a form of computational model. It may be used for interaction with HPC programs and for programming parallel algorithms of visualization.

The current paper is devoted to the main part of software technology – the model. It called main because other parts, e.g. implementation, depends on it. The suggested model, in turn, is not a ready-to-run software. It may be implemented using various programming technologies with some kind of model variations suitable for that technologies. But before turning a model into technology, we should be sure that the model is effective. So the global plan is to suggest the model, to play with it using experimental implementations, and finally create a technology.

It might be philosophically incorrect that single tool solves two problems, as in our case. But this fact might be corrected if a view on problem environment will drastically change in future. At least, it is not looking bad to have a parallel computational technology that may interoperate with other computational technologies.

The structure of this paper is as follows. In Section 2, the problem statement is defined. Sections 3, 4 and 5 propose a designed model for parallel programming. Section 6 highlights prototype implementation details. Section 7 suggest an experimental application of the model for parallel rendering task.

2 Problem statement

To going further, we should define what we consider as a typical parallel computational program. It will give us a picture what kind of software in which environment we should operate with for online visualization.

2.1 Formalization of online visualization

Without loss of generality, we fix the scope of the developed online visualization model in the following formulation.

There is a set of information entities *{D}*, each divided into parts in the domain sense (so called domain decomposition), e.g. each $D = \{d_i\}$. For example, one may consider a structured grid D which is decomposed into parts $\{d_i\}$, as on fig. 1. These parts d_i are such that they fit into the memory of the computation process that calculates associated part.

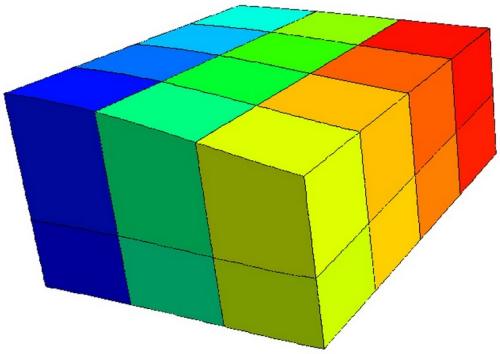

Fig. 1. A distribution of structured data on computation cores. Image courtesy of [5].

The scientific simulation is implemented in the form of a set of computation processes (processes of the operating system and processes on accelerators) located on a set of hardware nodes. These processes interact as necessary with each other and with external sources for the exchange of input, intermediate, edge, and output values. The set of computation processes and the structure of their interaction can change over time, as well as the set of computed entities *{D}*.

A significant feature of the content of entities *D* is that this content changes over time. Thus, entity *D* is a "living" informational "matter", its life (evolution) is a process of change of it's content in the course of the computational process.

The variability of the contents of D is also due to the limited memory of the hardware nodes. As a rule, only the contents of the previous and current iteration step of the computational process are stored in memory. Although, in general, computational processes can store a larger number of steps in node memory (for example also using disks). But this does not change the nature of the entities *{D}* - they evolve in time, and a limited trace of states or images of states from previous iterations follows them.

At the same time, in practice, the structure of the partition of *D* does not usually change during the calculation, although this is theoretically possible.

The **task of online visualization** is to build numerical and visual images of entities *{D}* and transfer them to the destination, build visual images of the composition and structure of computational processes, supply control signals to computational pro-

cesses, and possibly manage their composition and structure (that is, control the calculation process).

Using to this definition, **insitu visualization** may be considered as a specific part of online visualization. It doesn't need interactivity and concentrates on generating images using HPC power. Also, **parallel rendering** and **remote visualization** also may be considered as areas used by online visualization: they fit naturally in the phenomenon.

Now we are ready to provide problem statement: create a technology (a model, and it's implementation) that solves stated task of online visualization.

3 Calculation model. Basic level

A practical computational model is proposed, which can be programmed and which is expected to implement all the necessary features to solve tasks of online visualization. The model consists of three levels. This section deals with the first, basic level. Then in ongoing sections two additional levels will be described.

The model starts with the concept of a message. A **message** is a triple *(label,dictionary,payload)* , where

- *label* – is a message label,
- *dictionary* – is a key-value dictionary,
- *payload* – binary large objects associated with message.

The message label may be different, which will be discussed later. A dictionary is understood in the usual programmer sense, that is, a set {(key, value)} with the restriction that in each dictionary the key is unique (that is, exactly one value corresponds to the key in the dictionary). The payload is the additional binary information associated with the message. Its structure and meaning are determined by the interacting parties. The payload is taken out of the dictionary for technical purposes – so that the dictionary takes up relatively little memory; while a payload can be relatively large.

A **system** is a computation that performs certain actions according to a model. The interaction of different entities is implied by the system.

The message can be "sent" to the system (by any party, without restrictions). The system processes incoming messages using the so-called reactions. **Reaction** is the pair of *(criteria, action)* , where

- *criteria* – triggering criteria,
- *action* – action to execute when the reaction triggered,

Any party may **register** reactions within the system. When a new message is sent, actions of all reactions whose criteria matches a message are executed, in order as they registered.

Additionally, any action is able to cancel further processing of other actions. Actions are assumed to be computationally simple and limited in execution time. It is important to note that actions can, in particular, a) test additional conditions (inexpressible in criteria), b) send new messages to the system, c) register additional reactions.

The list of registered reactions can change dynamically over time.

Reactions are considered to have no shared state between each other. This design decision allows to execute them (e.g. their actions) without any synchronization, in parallel for each arriving message. The only dependence between reactions is when single message is processed, as actions are processed in order (see above).

A note about the reaction criteria. The criteria used by model might be different. The main demand for criteria is that is should allow to identify reactions matching incoming message with little computational complexity.

Without loss of generality, the current paper uses the following mechanism of criterion: the message label and reaction criterion are strings. If **message label equals to criterion**, then (and only then) we assume that criterion matches that message.

The reaction definition operates criteria, while here we denoted single criterion. It is assumed that criteria is constructed as a list of criterion. When message matches any of criterion from list in reaction, the message is considered matched to that reaction. Thus, we apply OR logical operation. This design decision is made by considering that it is ergonomic to have single reaction to match different kinds of messages.

4 Calculation model. Service level

The basic level of our model does not allow solving the entire range of tasks required to solve online visualization problems. However, this level is extensible, it allows to add additional features to it. It is suggested to add these new features using the following concept of services.

A **service** is a set of reactions registered within the system, and possibly additional software processes and other components. Together, they implement the functionality of a service.

Interaction with services is expected to be done primarily through messages introduced at basic level of the model. This design decision allows other parties to hook into such communications by placing additional reactions, which is considered to add flexibility to the computation.. But there is no restriction that interaction is allowed only through messages. One may implement custom API of any service if required.

The list of services can be updated as needed. To date, the practical need for the following services has been identified.

4.1 Service for managing reactions via messages.

It was found convenient to manage list of reactions using messages. This allows to use only message sending API to interact with the system. The service adds the reaction to the system that reacts to following message:

- label: “manage_reactions”
- cmd:"add" | "update" | set | "remove"
- reaction_id: string
- criterion: string, a criterion of controlled reaction

- action: string, the action code of the controlled reaction.

When message of such kind arrives, the service manages the list of reactions registered. The service assumes that each reaction must be associated with a unique identifier. This is due to the need to distinguish between reactions.

The action code of an action is assumed to be described in a programming language that the system supports. Since messages are supposed to be transmitted through various information transfer protocols, this code is assumed to be a string. In the future, if necessary, this restriction can be relaxed (for example via *setenv*, below).

4.2 Query service

A **query** is a special kind of reaction, which differs in that the action of such reaction is executed on client that issued query. As a consequence, action may directly interact with client program. Additionally, query may have a counter *N* which means that action should be executed no more than *N* times. Queries are useful for detecting messages of interest and implementing various logic. For example, queries are used by following runner service to load tasks to be executed.

Query service might be implemented using ordinal reaction, whose action send signals to client when message of interest is detected using some network protocol.

4.3 Task service

Task service is designed to execute arbitrary tasks using automatic balancing. Clients schedules tasks using messages. They are then distributed to dedicated runner nodes, which in turn execute these tasks and respond with results. These allows to describe arbitrary algorithm using steps that are executed in parallel, for example on nodes of supercomputer.

A task is scheduled using message of following signature:

- label: “exec-request”
- *code*: operation code
- *args*: a list of operands for operations
- *result-label*: the label for message with results of execution.

Here *operation code* is a code operation to be performed. *Args* is a list of operands that may contain constants, references to payloads (see payload service), and other values recognized by the system. They will be passed to operation. After execution of operation, it’s result is sent using message with label specified by *result-label*. This allows client to generate unique label, issue tasks, and catch results of those tasks.

Operation code might specify function in some programming language, or might specify a function defined in operations table. In latter case, such a table might be configured using messages of following signature:

- label: “setenv”
- name: string
- value: programming code.

Here *name* corresponds to operation *code*, and value contains code of operation in some programming language. It is considered that this code defines a function which

will be called when operation is called. Additionally, it might be useful to consider different programming codes for single operation, corresponding for different execution environments. For example, one may specify operation code both for CPU and for GPU. The system then will be able to choose appropriate code according to actual environment.

A note on "needs". During experimentation it was noted that it is inefficient to execute some tasks from scratch. Sometimes, there are repetitive subtasks occurred required by various tasks. An example of such subtask is to load some programming library, configure a GPU, and so on. It was found efficient to cache results of such subtasks and reuse them between different tasks. Thus a concept of *needs* was appeared**.**

A **need** is state of memory and hardware that is required by tasks to perform. A same need might be required by different tasks, and might be reused. A runner, before running operation of a tasks, prepares all needs required by that tasks. If need is already prepared (e.g. its result is in cache), runner just touches it's access timestamp.

Needs should be identifiable, because caching algorithm should be able to distinguish them and associate with incoming tasks. It seems this might be done by some kind of a function from list of arguments of a *need* to a string.

As it noted, a *need* corresponds to some state of memory and hardware. This means that *need* is tied to runner, and different runners prepare their own copies of *needs*.

Needs required for a task is natural to enlist in *arg* field of task description, specified in *exec-request* message. It is then natural to pass results of needs to operation in it's arguments. Thus, a task in our model transforms from single operation into operation and *needs* required for that operation.

A note on resources limits. Both *operations* and *needs* require computing resources to be performed: for example, some amount of memory, hardware, so on. The actual amount of such resources on available supercomputer nodes is limited. So the implementation of the computational model should consider those limits and and correlate them with task's and need's requirements. This is also important for maintaining cache of prepared needs to keep it within available limits.

4.4 Payload service

Payloads are binary large byte objects (blobs). They are required so interacting parties may interchange with actual data of computed entities. Payloads, due to the large amount of required memory, are taken out of the main system components. This can be implemented by creating a special service that would store payloads and present them as needed. This significantly "unloads" the main system. The idea to work with payloads in a separate service was suggested earlier by M. O. Bakhterev [6].

If one want to send a message with payloads, it should go through following steps.

1. Upload payloads to the payload service. As a reply, the service generates unique URL for each stored payload. This URL might be used later by any other participants to download payload from the service.

2. Put the received URLs of payloads into payload field of the message dictionary, and then send the message to the system.

Implementation of payload service should consider following aspects:

- "Uploading" data to payload service without actual movement of data in memory. This might be done using shared memory concept, where payload process gains ownership of memory where payload is stored.
- "Uploading" data to payload located in GPU (or other accelerators) memory without actual movement of the data. That is, client should be able to transfer handlers of GPU buffers to the service.
- Same things should be done for downloads: client should be able to access payload bytes without data movements.
- Offload payloads from RAM to persistent storage when it is still required but not accessed.
- Cleanup of payloads that are no longer required.

The latter is sophisticated theme and might require additional actions from client to take care of some kind of payload usage counters. In ideal, specific cases, it probably might be done automatically, like some kind of garbage collection. Such automatic cleanup probably will be simplified by tracking promises (see below) somehow.

Additionally, payloads service might be better interconnected with task service. For example, a runner may download payloads required for task scheduled onto that runner, while executing other task. It also may upload payloads of task results while executing other task.

5 Calculation model. Promise level

Preliminary usage of the model shown that is is not ergonomic for final applications. It is possible to express a variety of computations using it (probably a full variety, in a sense of Turing machine). But it is not convenient to express them only in terms of *messages*, *reactions*, and *tasks*. It was determined that more high-level primitives are required to make application code shorter and clearly to human mind.

One known primitive is a *promise*. It was developed by many researchers almost 50 years ago, see for example [7]. Our practice had shown that promises bring some additional level of clarity and adds compactness to parallel application codes, see section "Experiment".

A **promise** is an object that corresponds to data that will be calculated sometime. A promise can be created in one process, fulfilled (that is, filled with data) in another, and respond to fulfillment in the third.

The convenience of promises lies in the fact that they can be operated on at any time, even before the results of calculations are received. This makes it possible to describe parallel data processing algorithms using sequential codes, see example in section 6.

Promises can be created explicitly or implicitly. One of the convenient methods of implicitly creating and using promises is linking them with asynchronous task execution (which we employ in section *Task service*).

To do this, we extend our model with the following:

- Each task submission is associated with a promise object. Thus client scheduling a task gains a promise object of that task.
- Allow to specify promises in arguments of scheduled tasks.

In case if task have one or more promises in arguments, its calculation is started only when all such promises are fulfilled. Corresponding arguments are substituted by values of that promises. Thus task operation works as before, using arguments as values and don't boring that they were generated by other tasks.

This looks like Lambda calculus, with little difference that arguments for function application are calculated asynchronously and parallel. With promises, we implement applicative-order evaluation (and not normal-order, or lazy evaluation).

Explicit promises. Another way of creating promises is to create them explicitly. We add following operations into model for that:

- create_promises(n) → list of p – creates a list of *n* promises
- fulfill(p,data) – fulfills promise *p* with *data*.

Promises created this way might also be used in arguments of tasks, same as promises created implicitly. So system will wait their fulfillment before running tasks.

It is also might be useful to pass promises to tasks just as objects, without applying "wait and substitute" logic. Also, promises will be useful to send in messages, so their serialization of promise objects should be considered.

Usage of promises. Explicit promises trivialize connection of the model to scientific simulations. We consider the following scenario. As it stated before, each iteration of simulation computes some entity D that has domain decomposition $\{d_i\}$. Let each iteration of simulation have associated structure $S=\{p_i\}$ of promises corresponding to that domain decomposition. Each computational process of simulation fulfills promises which correspond to parts that this process computes. Simulation sends *S* to the system. Visualization algorithms get *S* and schedule tasks based on promises from S, required to achieve target visualizations.

This logic is modular. Each visualization algorithm may be expressed then as a sequential **function from *S* to *R***, where *S* is a structure of promises describing source entity and *R* is a structure of promises describing result of algorithm application.

Such algorithm implementation considered as following. It gets *S* in arguments, then schedules a set of tasks to *task service*, passing promises from *S* as arguments for that tasks. Because the model have feature to get promise for each scheduled task, algorithm may then pass such promises to additional tasks or sub-algorithms, so on. Finally, it achieve promises for *R* and returns it.

Above-mentioned visualization algorithms are functions, however online visualization is a process (because it visualizes ongoing computations). To create a process of visualization, we consider following: add a reaction for each new incoming *S*, issued by simulation, and pass execution to visualization function with that *S* as argument. Thus we will achieve that visualization will be built as simulation goes on.

6 Prototype implementation

The author develops implementation of suggested model. It uses Javascript language and works within NodeJs and browsers, and uses HTTP and Websocket protocols for inter-node communications. Following some ideas achieved during implementation of the model are highlighted.

Client library. It was found convenient to use client library for model clients to access the system API. The library provides entry points for all services described above:

- **msg**(m) – send message *m* to the system. The *m* is considered to be javascript object with at least *label* field.
- **reaction**(criteria, action) – register a reaction within the system, which will call action for every message that meets criteria. The *action* is encoded as a string with a function in Javascript language.
- **query**(criteria,N,callback) – put a query to the system which will call *callback* for at most *N* times.
- **exec**(opcode, args) – schedule task defined by (opcode,args) and return it's promise.
- **setenv**(name, value) – define operation body where name is operation code and value is body of operation on some language
- **promise**(N) and **fulfill**(p) – explicitly creates N promises and fulfills given promise. Currently not implemented yet.

Thus all clients load the library and interacts with the system using calls to it.

Distributing reactions. First implementations were sending messages to some central master node which role was to execute all registered reactions. It was occurred to be non effective. Then a new approach was developed with idea to distribute reactions to clients. When client "sends" a message to the system, it actually doesn't send it, but executes actions of registered reactions corresponding to that message. This approach seems to be much more effective, because actions are executed concurrently, on the client processes. Central node is still required but it's role is to manage list of registered reactions and send updates on that list to active clients. Probably such server might be replaced later with some peering mechanisms.

Query service. As reactions are executed on clients, query service was implemented by the following. When some client (query owner) issues a query, a local HTTP server is started up inside that client's process. It provides endpoint *URL* which is ready to receive incoming messages asynchronously. Then a new reaction is registered within the system. It's *criteria* is a *criteria* of query, and it's action is to send HTTP request with found message to endpoint URL of query owner. Thus when some party "sends" message to the system that is interesting to the query, it actually sends that message directly to the query owner.

Task service. Current implementation introduces concept of *runner processes* and single *runner-manager process*. The manager queries all upcoming scheduled tasks by placing query to messages with *exec-request* label. Runners advertise them to the manager using special messages with *runner-info* label. The manager continuously executes assignment algorithm to decide which tasks on which runners to perform. It then assigns tasks to runners. When runner achieves a task, it executes it and sends results to the manager and to the client of the task.

When assigning tasks to runners, the manager considers *needs* that already prepared on that runner, solving the assignment problem with some kind of heuristics.

Additionally, task service is used to track all promises within the system.

Payload service. It is implemented as a set of HTTP servers, which are started on each hardware node participating in computation. When client wants to submit payload, it communicates with payload service component located on the same node where client is located (e.g. localhost). The unique URL of each stored payload is considered to be generated using local counter on the node, and network address of the node. This assumes that all nodes have unique addresses and interconnected, but it seems to be common practice in supercomputing.

Thus, when client sends message with payload, actual payload bytes are kept on the node of the client. It might be transmitted over network later, if some other client would decide to download that payload.

In future, implementation of payload service is planned to consider shared memory and GPU buffers to avoid unnecessary transferring of data.

Connecting to other platforms. In spite of current implementation uses Javascript language, it might be used within other platforms. First of all the machine-code platform is considered (C++, Fortran, so on) because it is most often used in scientific computations. Two ideas are considered for connecting to other platforms:

- Middleware nodes.
- Specify reaction's actions in different languages.

Middleware node is a node that on one hand interacts with the system, and on other provides special API for it's clients on other platforms. For example, it might be interesting to provide API based on some kind of FUSE file systems to interact with the model. In that idea, writing to file of some specific path will issue the message, while reading some file leads to performing a query.

Another option is to allow specifying reaction's action in language other than Javascript. It might be done be leveraging same approach as used in task service, e.g. by forming a table of actions using *setenv* messages. This will allow client library to download and execute programming codes of actions appropriate to the platform of the client.

7 Example application

In this section test the created model and implementation with the task of parallel interactive rendering of 10^9 cells (in form of voxels). This task is a simplified version of comparison test of Paravew and ScientificView visualization systems given in [5]. The simplification is that cells here have no associated values (e.g no fields – just geometry). Cells are given by their coordinates *(x,y,z)* and all have same constant size *(dx,dy,dz)*. The code of solution is provided on fig. 2.

```
let K = 50
let filenames = ["1.dat","2.dat",…,K+".dat"]
let blocks = filenames.map( __load )

rapi.query( "render",(m) => {
 let images = blocks.map( b =>
 __render(b,m.camera_position,m.w,m.h) )
 let final_image = recursive_merge( images )
 rapi.msg( {label:"image", final_image } )
})

function __load( filepath ) {
  return rapi.exec( arg =>
      read_file_as_floats(arg.filepath),{filepath} )
}
function __render( block, camera_position, w, h ) {
  return rapi.exec( arg =>  arg.render_fn(arg.camera_position),
      {render_fn: {code: "cell_render_func", need: true,
                   arg: {block,w,h}}})
}

function recursive_merge( images ) {
  if (images.length <= 1) return images[0]
  let acc = []; for (let i=0; i<images.length; i+=2 )
     acc.push( __merge_2( images[i], images[i+1] ) )
  return recursive_merge( acc )
}

function __merge_2( image1, image2 ) {
  return rapi.exec( arg =>
    merge_2_zbuf( arg.image1, arg.image2 ), { image1, image2 })
}
```

Fig. 2. Source code of interactive parallel rendering of cells (javascript). The system's API is provided via *rapi* variable. The code of operation *cell_render_func* is big and omitted for clarity, also *merge_2_zbuf* is simple and omitted too.

It is considered that cells that we have to render are distributed into *K* parts and stored in files named *k.dat*. The following is going on in the code:

1. The code starts with scheduling load task for each block. As a result, an array of promises of loaded blocks is stored in *blocks* variable.
2. The code issues a query on message with **render** label. It is assumed that visualization frontend issues such messages.
3. When render message issued, an query callback is called and it starts parallel rendering process by calling __render function for each part. The __render schedules render task to the system for parallel execution. As a result, an array of promises is achieved and stored into *images* local variable. These promises are considered to be fulfilled to images of rendered parts in the form *(color-buffer,z-buffer)*, e.g. having both color and depth data.
4. *Images* promises are passed to recursive_merge algorithm which in turn schedules tasks to join images using sort-last method.
5. Final image is sent with message labeled “final_image” which is queried and displayed by visualization frontend.

Fig. 3 (left) displays sample output of developed application.

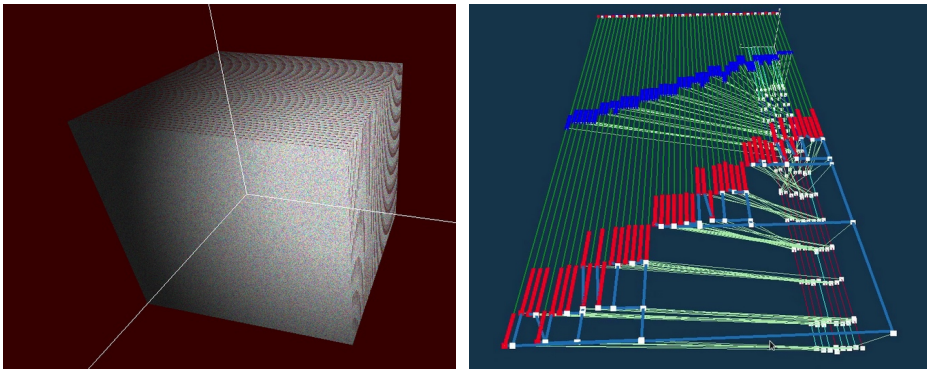

Fig. 2. Left: visual result of interactive parallel rendering of cells. A cube of size 1 is split into 50 parts having 10^9 cells in total. User may rotate view angle and change camera position using mouse. After camera position change the cube gets re-rendered. Currently it gets around 5..20 seconds to complete re-render using 8..50 runners. **Right**: visual debugging of cells parallel rendering algorithm. Time goes from up to down. Blue lines are tasks of loading blocks, red lines are block rendering tasks. White dots are image merge tasks (actually they are lines too but perform to fast so appear as dots). Task’s *x* location in a view corresponds it’s block number. Data dependencies between tasks is shown using cyan lines. There are 8 runners shown performing tasks, on bottom plane. Animation is available: youtu.be/XnV3l8hw8QE.

To debug parallel applications implemented within the suggested model, a visualization tool was created. It catches messages that schedule tasks to the system, and also messages when task is assigned to a runner and when task is completed. Then it visualizes ongoing processes using synthetic view in 3D space. On fig. 3 (right) an output of such view is presented. It may be considered as online visualization of a parallel algorithm structure.

8 Conclusion

In the paper a specialized model for parallel computations is suggested. It is specialized for online visualization tasks, which assumes that visualization of computation is made during that computation. These tasks demand an ability for inter-operation with working scientific simulation codes. Simulation codes need to pass data to visualization, and to receive commands from it.

To pass data from simulation to visualization, a message sending operation is suggested. To receive commands, a message query operation is suggested. Additionally a higher level mechanism of promises is suggested to simplify programming of inter-operation between simulation and visualization.

A proof-of-concept implementation of the suggested model is presented. Some practical ideas of implementation are highlighted. An example application that uses the suggested model is presented – a parallel rendering of cells. It is not actually an online visualization application, but instead an interactive visualization tool. Despite of that it served as a very productive basis for understanding the needs of the model and it's implementation.

A new computational model is suggested despite that there are already exist a lot. Starting for example from MPI and passing to Template Task Graph technology [8] (overview video). Author neglects to use MPI because it has fixed number of participating processes, due to it is effective for online visualization to change number of computing nodes at runtime according to current demands. In contrast, task-based parallel programming technologies provides this and other benefits, but their model is often non-clear.

Author tries to keep the model as much as close to computation basics, keeping out of non-substantial entities. Such basics are seems achieved for example by C.A.R. Hoare's CSP [9] and Yuri Gurevich's AST [10] models.

In current paper, nothing novel of primitives is added:

- Sending and reacting to messages is something from basis of computing.
- Promises are developed in 1970th.
- Tasks looks like asynchronous remote-procedure calls to a worker pool.

What is new is the implementation nuances and details of composition.

- Sending messages to a common bus instead of sending them directly to target peer. This allows any other peer to hook into "communications" and add additional logic. It looks like system-wide mixins used to influence processing logic on multiple nodes at runtime.
- A method of communication between simulation and visualization consisting of transferring structures of promises. On each computational iteration, simulation sends meta-information about data being computed, represented as structure of promises according to domain decomposition. This structure or it parts or individual promises then may be easily distributed to various tasks of visualization. This differs from existing practice of establishing communication channels which are then tied to "reader" processes (as in ADIOS2).

The author thanks colleagues for discussions on the presented work.

References

1. Bennett et al, Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. International Conference for High Performance Computing, Networking, Storage and Analysis, 2012, SC. 49:1-49:9. DOI: 10.1109/sc.2012.31
2. Childs H, Ahern SD, Ahrens J, et al. A terminology for in situ visualization and analysis systems . The International Journal of High Performance Computing Applications. 2020;34(6):676-691. DOI: 10.1177/1094342020935991
3. Kenneth Moreland, Andrew C. Bauer, Berk Geveci, Patrick O'Leary and Brad Whitlock. " Leveraging Production Visualization Tools In Situ ." In *In Situ Visualization for Computational Science* . Springer, 2022. DOI: 10.1007/978-3-030-81627-8_10
4. Pavel Vasev, Analyzing an Ideas Used in Modern HPC Computation Steering // 2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT), Yekaterinburg, Russia, 2020, pp. 1-4. DOI: 10.1109/usbereit48449.2020.9117685
5. Potekhin, A. L., On one way of organizing data structures in scientific visualization systems // Issues of atomic science and technology. Series: Mathematical modeling of physical processes. – 2022. – No. 4. – pp. 64-71. – DOI: 10.53403/24140171_2022_4_64 In Russian.
6. Bakhterev MO, On the application of the data flow formalism in the development of parallel and distributed programs // Parallel Computing Technologies / Proceedings of the Scientific Conference. Chelyabinsk. Publishing house of SUSU, 2007. Volume 2. Pp. 201-211. In Russian.
7. Friedman, Daniel P. and David S. Wise. “Aspects of Applicative Programming for Parallel Processing.” IEEE Transactions on Computers C-27 (1978): 289-296. DOI: 10.1109/TC.1978.1675100
8. J. Schuchart, et al., "Generalized Flow-Graph Programming Using Template Task-Graphs: Initial Implementation and Assessment" in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 2022 pp. 839-849. DOI: 10.1109/ipdps53621.2022.00086
9. C.A.R. Hoare, Communicating Sequential Processes , December 4, 2022.
10. Blass, Andreas & Gurevich, Yuri. (2008). Abstract state machines capture parallel algorithms: Correction and extension. ACM Trans. Comput. Log.. 9. DOI: 10.1145/1352582.1352587