A Computational Model for Interactive Visualization of High-Performance Computations

Pavel Vasev[0000-0003-3854-0670]

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. Interactivity 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 interactive visualization. At basic level, it is based on messages and reactions. At higher level it uses promises for inter-operation of computation and visualization parts.

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]. The difference is that online is considered to be interactive, steered by user or algorithms working on user's behalf. Insitu on other hand underlines the placement of visualization processes closer to a computation. Whereas 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 to 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 usually is solved using various approaches, for example see [2,3,4]. A most common approach is to provide some data transmission service, and a library for interacting with it. Computing application is instrumented with calls to such library and thus data is offloaded from application into visualization processing via such transmission service. However, existing approaches doesn't solve the second task – they don't provide any parallel programming models, in particular for implementing visualization algorithms.

On other hand, there are a plenty of technologies for parallel programming. However they are mostly not considering an interaction with existing parallel programs, which are built using other parallel technologies. They are more focused to be self-sufficient. Thus often a some kind of bridge is required.

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 any software technology – the model. It is 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.

We need to note the following. It might be philosophically incorrect that single tool solves two problems, as in our case. In current work, we join those problems into one: construct a way for high-performance online visualization. At least, it is not looking bad to have a parallel computational technology that may interoperate with other computational technologies well.

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. Section 8 express related works.

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 entity D 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 each part fits into the memory of the computation process that calculates this part (whereas maybe two parts will not fit in its memory).

The scientific simulation is implemented in the form of a set of computation processes (processes of the operating system and processes on accelerators like GPU) lo-

cated on a set of hardware nodes. These processes interact as necessary with each other and with external sources for the exchange of input, intermediate, boundary, 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}*.

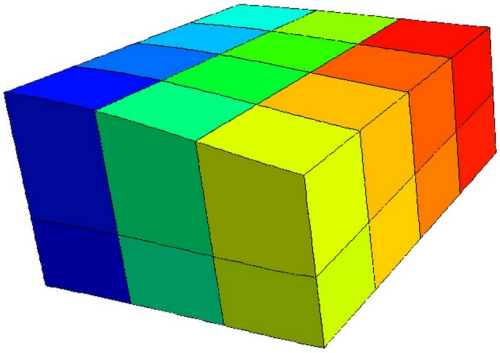

Fig. 1. An example distribution of structured data on computation cores, from [5].

A significant feature of entities *D* is that their content (e.g. data) 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 content of *D* is primarily due to the limited memory of the hardware nodes. Usually, only the contents of the previous and current iteration step of the computational process are stored in memory. Of course, in general, computation may store a larger number of steps in nodes 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 their states or images of that states from previous iterations follows them.

At the same time, in practice, the structure of the partition of *D* onto parts d_i does not usually change during the calculation, although this is sometimes used, for example in adaptive mesh refinements.

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 of computational processes, supply control signals to computational processes, and possibly manage their composition (that is, control the calculation process).

Using to this definition, **insitu visualization** may be considered as a specific case or part of online visualization. It doesn't need interactivity and concentrates on generating images using HPC power. Also, **parallel rendering** and **remote visualization** 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 the stated task of online visualization.

3 Computation Model. Basic Level

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 that 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 unique keys. The payload is the additional binary information associated with the message. Its structure and meaning are determined by the interacting parties. The payload is not placed in 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 the model. **Acting parties** interacts with each other via the system by the following.

The message can be "sent" to the system. 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 were registered.

Actions can do arbitrary processing and, in particular, a) test additional conditions (inexpressible in criteria), b) send new messages to the system, c) register additional reactions. Additionally, an action is able to cancel further processing of other actions.

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 actions without synchronization, in parallel for each arriving message.

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 consider logical OR. This design decision is made because it is ergonomic to have single reaction to match different kinds of messages.

4 Computation Model. Service Level

The basic level of the 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, e.g. some logical process.

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 globally unique identifier. This is due to the need to distinguish between reactions.

The action code of an action is assumed to be possible to execute in a programming environment that the system supports. It may contain for example source code in interpreted language like Python or IDs of methods to invoke for compiled languages.

4.2 Query Service

A **query** is a special kind of reaction, which differs in that the action of such reaction is executed on the client process that issued the query. As a consequence, an action may directly interact with a 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 register tasks to be executed.

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

4.3 Task Service

Task service is designed to execute arbitrary tasks using automatic balancing. Clients schedules tasks using messages. Tasks then distributed to dedicated runner nodes, which in turn execute these tasks and respond with results. This allows to program various algorithms using steps (e.g. a task is a such step) that are executed in parallel.

A task in our model is scheduled using message of the 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* defines 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 the operation. After execution of the 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 table is configured using messages of the following signature:

- label: “setenv”
- name: string
- value: definition of function to execute.

Here *name* corresponds to operation *code. Value* defines a function which will be called when operation is called. For example, it’s code in a programming language.

Additionally, it might be useful to consider different values for single operation, corresponding for different execution environments. For example, one may specify operation both for CPU and for GPU. The system then will be able to choose appropriate variant according to actual hardware 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 kernel for 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 some 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. Thus system user have to provide globally unique identifier with each *need*.

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 of a task are enlisted in *args* field of task signature (specified in *exec-request* message), together with other arguments. After a *need* is computed, it's value replaces corresponding argument. Thus applicative-order evaluation is performed.

A note on resources limits. Both *operations* and *needs* require computing resources to be available: for example, some amount of memory, hardware, so on. The actual amount of such resources on available 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). Due to the large amount of required memory, they are processed separately from message bodies. It is implemented by a special service that store payloads and present them as needed. This significantly "unloads" the main system. This idea was suggested earlier by M. O. Bakhterev [6].

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

1. Upload payloads to the payload service. The service generates unique URL for each stored payload. This URL might be used later by any other parties 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 the aspect of actually not moving data when payload is "uploaded" or "downloaded". Same should be implemented for memory on accelerators. Thus, any real movement of data should occur only when data is requested by party on remote node. This for example might be implemented by placing service parts directly into client processes.

However, extra "technical" movement of data may occur to offload payloads from RAM to persistent storage when it is still required but not accessed to offload RAM. Thus, service should act like a cache.

Implementation also should cleanup of payloads that are no longer required. It 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).

Additionally, payloads service might be better interconnected with task service, in aim to implement interleaving of data movements and task computations.

5 Computation Model. Promise Level

Usage of the model shown above is still not ergonomic for final applications. Additional primitives are required to make application code shorter and clearly to human mind.

One suitable known primitive is a *promise* (also called a *future*). It was developed by many researchers almost 50 years ago, see for example [7]. A **promise** is an object that corresponds to data that will be calculated sometime.

A promise can be created in one process, *resolved* in another process (also term *fulfilled* is used, e.g. bound with data), and perform reactions to promise resolution in some third processes. Promise objects may be freely copied between parties, for example using messages.

The convenience of promises lies in the fact that they can be operated on at any time, even before they resolved. This makes possible to represent parallel processing algorithms using sequential codes, like in *Example application* section.

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 employed in *Task service*, section 4.4.

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.

Sometimes it is required to pass promises to tasks by reference, without applying synchronization logic. Implementation should consider that case.

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,
- *resolve: p,data → void* – resolve 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.

Usage of promises. Explicit promises trivialize connection of the discussed computation 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 of *D* 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 of *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 the suggested model. It uses Javascript language and works within NodeJs and in browsers, and uses TCP, HTTP and Websocket protocols for inter-node communications (use of OpenUCX is considered). Following some ideas achieved during implementation of the model are highlighted.

Client library. It was found convenient to use client library 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 *label* field, maybe *payload* and other fields.
- **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 and then stop reacting to messages.
- **exec**(opcode, args) – schedule task defined by (opcode,args) to the task service and return task's promise.
- **setenv**(name, value) – define operation where name is operation code and value defines body of operation.
- **promise**(N) and **resolve**(p) – explicitly creates N promises and resolves given promise.

Thus all clients load the API 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. To get reactions, clients asks the central node, one time per criterion. Thus actions are executed concurrently, on the client processes. Central node role is to manage list of registered reactions, and to send parts of that list and following updates to active clients.

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 TCP server is started up inside that client's process. It provides endpoint *URL* which is ready to receive incoming messages asynchronously. Then query owner registers new reaction within the system. It's *criteria* is a *criteria* of query, and it's action is to send TCP request with found message to endpoint 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 *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.

Runners track ready state of tasks assigned to them (e.g. all promises in arguments in task are resolved), and execute them as they become ready.

Payload service is implemented as a part of client library and additional set of servers, which are started on each hardware node participating in computation. When client submits payload, a pointer to payload in RAM is stored in client state. Then a server is started inside client process. Then client library returns unique URL which may be used to access that payload from outside processes within the system.

Thus, when client sends message with payload, actual payload bytes are not moved. It might be transmitted over network later, if some other client would decide to download that payload.

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. Currently we implemented this by creating plugin system for runners, and created a special plugin to execute Python codes.

7 Example Application

We showcase the model with parallel interactive rendering of rectilinear grid of cells. It is not actually an online visualization, but it is very close for it's needs. The application is a simplified version of comparison test of ParaView and ScientificView visualization systems given in [5]. In our case, cells have no associated values, just geometry in form of voxels.

```
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 )
} // Todo: try converting to async queues as in [12].

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. Only codes important for showcasing main structure are provided. An external visualization frontend is considered: it allows user to control camera in 3D space, tracks it and sends 'render' requests, receives 'image' messages and displays it.

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 file load task for each block. As a result, an array of promises of loaded blocks is stored in the *blocks* variable.
2. The code queries messages with *render* label. It is assumed that visualization frontend issues such messages.
3. When render message detected, the query callback is called and it starts parallel rendering process by calling *__render* function for each block. 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 rendered images of blocks in the form *(color-buffer,z-buffer)*, e.g. having both color and depth data.
4. *Images* promises array is 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.

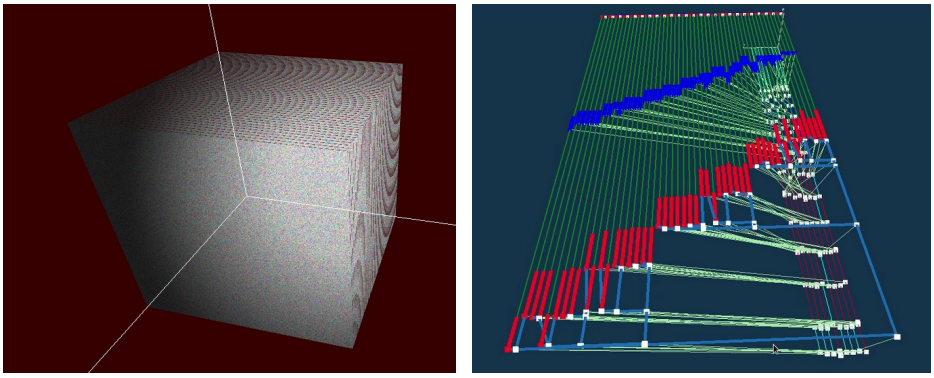

Fig. 3. Left: visual result of interactive parallel rendering of cells. A cube of size 1 is split into 50 parts having 10^9 cells in total. User may rotate view angle and thus change camera position using mouse. After user change camera position, code sends "render" message and the cube gets re-rendered. **Right**: visual debugging of used parallel rendering algorithm. Time goes from up to down. Blue lines are tasks of loading blocks, red lines are tasks of block rendering. White dots are tasks of images merge (actually they are lines too but perform too 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 on bottom plane, performing tasks. Animation is available: youtu.be/XnV3l8hw8QE.

To debug parallel applications implemented within the suggested model, a task visualization method was created. It queries 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. The developed debugging method is itself an online visualization of parallel algorithms working within the system.

8 Related Works

According to the aim of our project, we have to consider scientific advances in two areas: online visualization and parallel computation technologies. Former nowadays are mostly introduced by various institu systems. In the latter, our approach is near to task-based parallelism and asynchronous many-task systems (AMT). Also, as we consider visualization of scientific computations, we pay attention to existing parallel visualization areas, including parallel visualization-aimed processing and rendering.

Related AMT systems are: Template Task Graphs [8] and PaRSEC, HPX [11], Charm++, Uintah, Kokkos, Legion, C++ Sender Library, LuNA[13], Dask, and Flyte.

AMT languages: Linda, Pythagoras functional-streaming parallel language [12], Chapel and X10, Cilk, T++, Val, Caper and partially Lingua Franca.

Insitu visualization projects: ADIOS2, Ascent, Sensei, Henson, Damaris, Libsim (Visit institu component), Catalyst (Paraview insitu component).

Parallel visualization processing: Paraview's data-server and Python-based visualization pipeline programming, ScientificView parallel pre-post-processor [5].

Parallel rendering libraries: IceT (used by both Paraview and Visit for rendering), VTK-M (parallel but single node only).

Our project differs from projects stated above in various details. To illustrate this, let's look on two advanced projects – HPX for computations and ADIOS2 for institu.

HPX system has a well-described model named ParalleX (for simplification we refer it here HPX too). HPX is targeted both for single-node and multi-node (e.g. distributed) parallelism. Our model is primarily for multi-node. We rely on existing technologies for single-node parallelism (C++ standard parallelism, SYCL, etc).

Both HPX and our model work with multi-node parallelism by allowing to run *actions* (HPX term, same as *tasks* in our terms) remotely. In such environment, a question of load balancing occurs. Our implementation provides built-in load balancing. HPX in turn demands user to control that aspect, telling where to run an action – on a specific node (*locality* in HPX terms) or on some HPX component. User may receive performance counters from nodes and so select best ones for ongoing actions.

Sometimes for task to perform, it needs a state existing in memory of a node. In HPX, this accomplished by so called *components*. These are like C++ classes, and even allow migrations of component from node to node. To address components, *active global address space* (AGAS) concept is used. HPX suggests to create components on remote nodes, and call component's actions.

Our model reaches same purposes by means of *needs (*described in section 4.3). A need may be thought as a part of component; each task have a specification of required needs and arguments. This allows our model to distribute tasks easily, because it "migrate" needs automatically by recreating them on any desired nodes. However need's state doesn't considered to be changed, and work with "state" assumes explicit specification of input and output arguments of tasks.

HPX have various advanced features, for example C++ standard-based parallelism implementation, and distributed data containers. Our model doesn't provide such means. It is indeed assumed that these may be implemented by plugins and aside projects, probably inter-operating in different programming languages.

Now we turn to compare with insitu framework ADIOS2. It is based on concept of streams. Streams are established (between producer and consumers) and then running. This provides maximum bandwidth with lowest latencies, but assumes that after establishing the connection a parties are running at fixed OS processes on fixed nodes.

In contrast, our model is designed to send “streams” of meta-data instead of data. The data follows metadata only if required by consumer, and is not sent by default. ADIOS2 users may implement the same by using 2 streams, one for meta-data and one for heavy data. But we found it ergonomic to make this feature present by design.

Our model is not tied to particular OS processes or nodes, because it is task-based (or even reaction-based, at first level). This allows automatic balancing the activities on respond on event occurrences (such as a simulation thread produced new portion of data). This in turn may introduce lags to processing.

But thanks to design of reactions implementation, our system allows to embed algorithms right into event occurrence place. For example, it allows to inject data processing codes right into simulation processes. The same effect is achieved with ADIOS2 operators and plugins. In both our and ADIOS2 cases, this looks like external management of algorithms, performed dynamically.

Despite we look optimistic on our design decisions and their contrasts with presented projects, we think only the practical tasks could highlight best combinations.

9 Conclusion and Future Work

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

The model is based on concept of messages and reactions. Then, a concept of tasks is added, which run concurrently. It was assumed that simulation software sends per-iteration updates on it’s data using messages, and reacts to control commands. Then visualization side reacts to data messages and processed it in aims of visualization.

However it was found inconvenient to express parallel algorithms of data processing using reactions on messages. To overcome this, the concept of promises is added.

To pass data from simulation to visualization using promises the following approach is suggested. On each computational iteration, simulation sends message with meta-information about data being computed, represented as structure of promises according to domain decomposition. Such structure or it parts or individual promises then may be easily processed by parallel visualization algorithms. Because with promises parallel algorithms are easily expressed by sequential algorithms.

Future work is currently seen as the following:

- Ability to specify actions of reactions and tasks for various programming platforms. It will allow to achieve inter-operation of system clients programmed in various languages, and easier running tasks on accelerators. We consider SYCL and Kokkos for the latter purpose.

- Suggested model is highly extensible. Provide some kind of plugin concept to involve other people into the project, letting them add more features.

The global aim of the project is to suggest parallel computations methods productive for reasoning, and to separate basics and higher-level. We are inspired for example by models like CSP [9], AST [10], A-system vision [14] and others (like [12]).

Finally, we are going to achieve human-computer interactive supercomputing as described by virtual test stands idea [15].

Author thanks colleagues and reviewers 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. M. Bakhterev, A. Kazantzev, P. Vasev, I. Albrekht, Dataflow-Based Distributed Computing System // Proceedings of the Euromicro PDP 2011 Work in Progress Session (Eds. E. Grosspietsch, K. Kloeckner), p.6-7, SEAA-Publications No. SEA-SR-29. Johannes Kepler University Linz (Austria), ISBN 978-3-902457-29-5
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
11. Kaiser et al., (2020). HPX - The C++ Standard Library for Parallelism and Concurrency. Journal of Open Source Software, 5(53), 2352, 10.21105/joss.02352

12. Legalov, A.I., Matkovskii, I.V., Ushakova, M.S. et al. Dynamically Changing Parallelism with Asynchronous Sequential Data Flows. Aut. Control Comp. Sci. 55, 636–646 (2021). DOI: 10.3103/S0146411621070105
13. Akhmed-Zaki et al. Automated Construction of High Performance Distributed Programs in LuNA System. 2019. DOI: 10.1007/978-3-030-25636-4_1.
14. Kotov V.E., Problems of parallel programming development, All-Union symposyum on “Problems of system and theoretical programming”, Novosibirsk, 1979, pp. 58-72. In Russian.
15. V. L. Averbukh, N. V. Averbukh, P. Vasev, I. Gajniyarov and I. Starodubtsev, The Tasks of Designing and Developing Virtual Test Stands, 2020 Global Smart Industry Conference (GloSIC), Chelyabinsk, Russia, 2020, pp. 49-54, DOI: 10.1109/GloSIC50886.2020.9267835.