Distributed BioDynaMo

AUGUST 2018

AUTHOR:
Nam Nguyen

SUPERVISORS:

Lukas Breitwieser
Ahmad Hesam
Fons Rademakers

Project specification

The BioDynaMo project (http://biodynamo.web.cern.ch/) aims at creating a platform through which life scientists can easily create, run, and visualize three-dimensional biological simulations. Built on top of the latest computing technologies, the BioDynaMo platform will enable users to perform simulations of previously unachievable scale and complexity, making it possible to tackle challenging scientific research questions.

Current work concentrates on developing a distributed runtime which is the foundation to run simulations in the cloud. The summer student will be involved in these activities.

The student should have good knowledge of C++ programming language and software development tools (git, cmake). Prior knowledge in distributed systems and GPU programming is a plus.

The project is scheduled from July 2nd to August 31st, 2018.

Abstract

Computer simulations have become a very powerful tool for scientific research. In order to facilitate research in computational biology, the BioDynaMo project aims at a general platform for biological computer simulations, which should be executable on hybrid cloud computing systems [Breitwieser et al., 2016]. To that end, the project has achieved very good *vertical* scalability with almost linear speedup on one node [Breitwieser, 2017]. This report describes the technical work to scale BioDynaMo *horizontally* to more than one nodes while fully retaining current C++ simulation code. A general axis-aligned box partitioning scheme which is aware of bordering regions was implemented to divide the simulation space into smaller boxes, representing local tasks. From there, a Python driver built up a task dependency tree so that Ray distributed framework [Moritz et al., 2017] can manage task executions. Each task reassembles all the regions into a bigger one, runs a simulation step on that, and disassembles it back into different regions for the next step. The Ray framework was enhanced to take into account the local availability of these different regions in scheduling. This technique allowed BioDynaMo to run a minimally modified demo simulation in a distributed manner with the driver on the host and the workers in a docker container, opened up the possibility of actual *large-scale* simulation. However, much more work remains to be done to bring the distributed mode up to par with local mode.

Contents

Contents **iii**
1 Introduction . . . 1
a About BioDynaMo . . . 1
b About Ray . . . 1
c About the project . . . 1
2 Method . . . 1
a Work division . . . 2
b Connect BioDynaMo C++ and Ray Python . . . 3
i Making use of C++ from Python . . . 3
ii Building task dependency tree . . . 4
c Consider local availability of blobs . . . 5
3 Result . . . 5
4 Future work . . . 6
a Reconcile movement of simulation objects . . . 6
b Alleviate serialization and deserialization overhead . . . 6
c Ensure deterministic simulation . . . 6
d Relieve memory usage and support resumption . . . 6
e Distribute the initialization of the global space . . . 6
f Validate simulation results . . . 7
g Upstream enhancements to Ray . . . 7
5 Conclusion . . . 7
6 Postscript . . . 7

Bibliography **9**

1 Introduction

a About BioDynaMo

BioDynaMo is a platform for biological simulation. It was started at CERN openlab in 2015 [Breitwieser et al., 2015]. The project is released under Apache 2 open source license with source code made available on Github [BioDynaMo authors].

The BioDynaMo code is written in the C++ programming language and divided into two main parts: the runtime, and the actual simulation code. Life scientists define how simulation objects are initialized and their precise interaction with other objects in simulation code; the runtime executes the simulation in steps.

b About Ray

Ray distributed execution engine schedules and executes tasks on worker nodes. Tasks can be written in the Python or Java programming language. A driver Python script (or Java application) provides executable codes and contexts (e.g. global variables) for all the tasks and sets up the dependency among them.

In a Ray cluster, the head node runs a Redis in-memory key-value database [Redis authors]. This is the global control store whose purpose is to store information about the whole cluster (such as the cluster configuration, task specifications, where an object actually resides in) and act as a pub-sub broker. Workers subscribe to the head node for updates to tasks, and objects. Each worker node runs its own Plasma in-memory object store [Moritz and Nishihara, 2017].

It is important to note that when a task is scheduled in Ray, its "futures" (also known as promises, or deferred) are returned immediately, even though the task may not be executed yet.

Before a task is executed on a worker node, the task's inputs are fetched (maybe from a remote object store) to the local object store. After a task completes, its outputs are, too, stored in the local object store. Task T_a is said to depend on task T_b if one of the inputs to task T_a is one of the outputs from task T_b.

c About the project

The project described in this report is about laying a stepping stone to distributed simulation in BioDynaMo. The goal of the project is to execute a simple, existing demo on more than one workers.

This is important to BioDynaMo because even though BioDynaMo scales up on one node very well, the physical resources on one node will put the hard limit on that scaling. In order for BioDynaMo to simulate a larger number of objects than it could currently, horizontal scaling with distributed computing is the best option.

Unlike Kanellis et al. [2017], in which the authors implemented a messaging middleware with an outlook to execution on the cloud, this project takes advantage of the distributed execution that Ray has already provided and focuses on how a BioDynaMo simulation can be fitted into that framework.

2 Method

As explained in About the project, the main challenge in this project is to fit a BioDynaMo's simulation in Ray's execution framework. This includes the follwing two main tasks and an optimization.

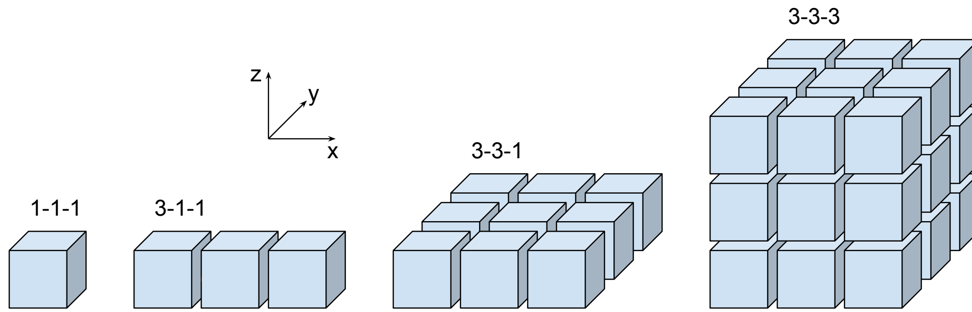

Figure 1: Illustrations of different 3D box partitioning schemes

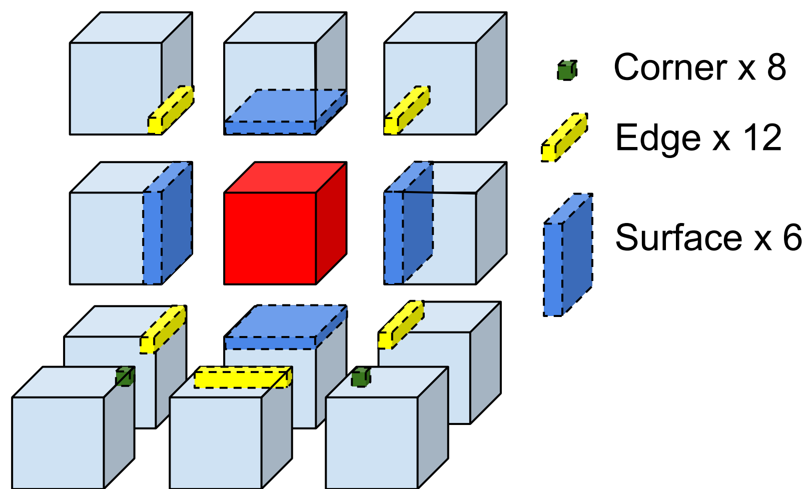

Figure 2: Halo regions bordering a box in red

a Work division

First of all, the work that is usually done on one node must now be divided into smaller tasks to be distributed. Since BioDynaMo is an agent-based simulation [Breitwieser and Hesam, 2017] where each cell only acts within its own sphere of influence, one way to divide the work is based on space, similarly to Hauri [2013].

The global 3D simulation space is partitioned into axis-aligned smaller spaces that I call boxes. At the moment, the user is responsible for the exact number of boxes along the x, y, and z axis. For example, when '3-1-1' is used, the user is telling the system that there are 3 boxes along the x axis, 1 box along the y axis (i.e. do not partition along the y axis), and 1 box on the z axis, giving a total of $3 \times 1 \times 1 = 3$ boxes, as shown in figure 1.

Each of these boxes may border with at most 26 other boxes. These bordering boxes contribute a small portion of their spaces that are called halo regions to the main box. These halo regions allow for complete coverage of influence of any cell within the main box.

Each box has six faces. Each face yields a surface halo region. The intersection of two adjacent surfaces (e.g. front and top surfaces) yields an edge halo region. And the intersection of three adjacent surfaces (e.g. front-top-left) yields a corner halo region.

In figure 2, we see a cut out of some bordering boxes (in light blue) and halo regions surrounding the red box. The box directly below the bottom surface of the red box (call it box 1) would contribute its top surface halo region, the box to the right of box 1 (call it box 2) contributes its top-left edge halo region, and then the box in front of box 2 contributes its top-left-back corner

halo region.

It should be noted that a box is authoritative with respect to only the cells in its space. Because of that, it is also authoritative of the halo regions that it will be contributing to other boxes, but it is not authoritative in the halo regions that it pulls in.

b Connect BioDynaMo C++ and Ray Python

The next step is to construct a tree of tasks from the work division so that Ray can manage and schedule them. These boxes directly correspond to tasks. Each box is worked on in one task. The work essentially boils down to:

1. Obtain the halo regions for the box in consideration and assemble them together to form a complete simulation space.

2. Execute a simulation step in this new formed space.

3. After the simulation step, extract the box and its halo regions to be used in the next step.

Because BioDynaMo (and the work partitioning described above) is written in C++, and Ray uses Python as its driver, there are several technical issues in interoperability.

i Making use of C++ from Python

The first challenge is to make use of C++ code from the Python side so that Python tasks can, at the very least, run C++ simulation code. There are many options here and the selected solution was to use the built-in `ctypes` module as a foreign function interface (FFI) between Python and C++.

The C++ code is compiled into a shared library with all the needed functions exported as C symbols (with the keyword `extern`). From the Python side, `ctypes` will load with shared library as `CDLL` object, and invoke the functions as thought they were Python methods on that object. The C++ primitive returned values (such as `int`, `double`, etc.) are translated to Python types transparently. For bigger objects such as a struct, or an array, the Python side will allocate memory first, pass the pointer to that memory range, and the C++ side will fill in that buffer.

With this solution, we were able to share the partitioning and simulation logic between C++ and Python. For the partitioning logic, these functions were exported from the C++ side:

bdm_get_box_count Returns the total number of boxes in the chosen partitioning scheme.

bdm_get_neighbor_surfaces Returns the number of neighbors and their corresponding halo regions. The list of neighbor identification and corresponding halo region is also returned as an output argument.

The simulation in general made use of these functions:

bdm_setup_ray Passes Ray cluster information and simulation parameter (such as partitioning scheme) from Python to C++ side.

bdm_simulate_step Assembles halo regions into a box, runs one simulation step on the bigger space, and stores the box and halo regions from the box for next step as blobs in the Plasma object store. The serialization and deserialization are done by ROOT framework [Brun and Rademakers, 1996].

Tasks	Boxes
Inputs	Contributed halo regions
Outputs	Authoritative halo regions
Managed by Ray	Managed by BioDynaMo runtime

Table 1: Paralellism between tasks and boxes

ii Building task dependency tree

Another problem is to translate the requirements of halo regions into task dependencies.

Initially, the global simulation space is created with all simulation objects in it. This space is partitioned into boxes. Each box is authoritative for the halo regions that it will contribute to other boxes.

Then for each of the n simulation steps, each box will have to obtain the halo regions from its neighboring boxes to assemble a bigger space. These halo regions are the dependencies of a task, which simply calls into `bdm_simulate_step`. After the simulation step is done, the task produces 27 returned values, representing the main box, and its 26 halo regions, for use in the next simulation step that will be worked on by another task.

The parallelism between tasks and boxes is summarized in table 1.

As discussed in About Ray, Ray manages the inputs and outputs of a task transparently. There is no connection between an input or output of a task to a halo region or a box in the BioDynaMo runtime. But due to this parallelism, we can construct a simple scheme to map between task inputs and outputs to boxes and halo regions.

First, the initialization of the simulation will be a task which will return $b \times 27$ matrix of dummy values where b is the total number of boxes. Each row in this matrix comprises of a main box and its 26 halo regions. Let's call this matrix $laststep$.

The first simulation step of a box, then, will depend on the dummy object in $laststep$ that represents the box under consideration, as well as all other dummy objects in $laststep$ that correspond to the halo regions that this box should pull in. The outputs from this task will be an array of 27 dummy values, slotted in a matrix called $thisstep$ at the row corresponding to the box in question. When all boxes in the first step are done, we swap the two matrices $laststep$ and $thisstep$.

Then we repeat until we've completed all n simulation steps.

Finally, when all steps are done, we schedule one final task that depends on all b main boxes in $laststep$ to signal the end of the simulation.

Note that this process does not actually execute any task but merely schedules all the tasks with proper dependencies.

Figure 3 depicts the task tree for a simulation following the 2-1-1 partitioning scheme.

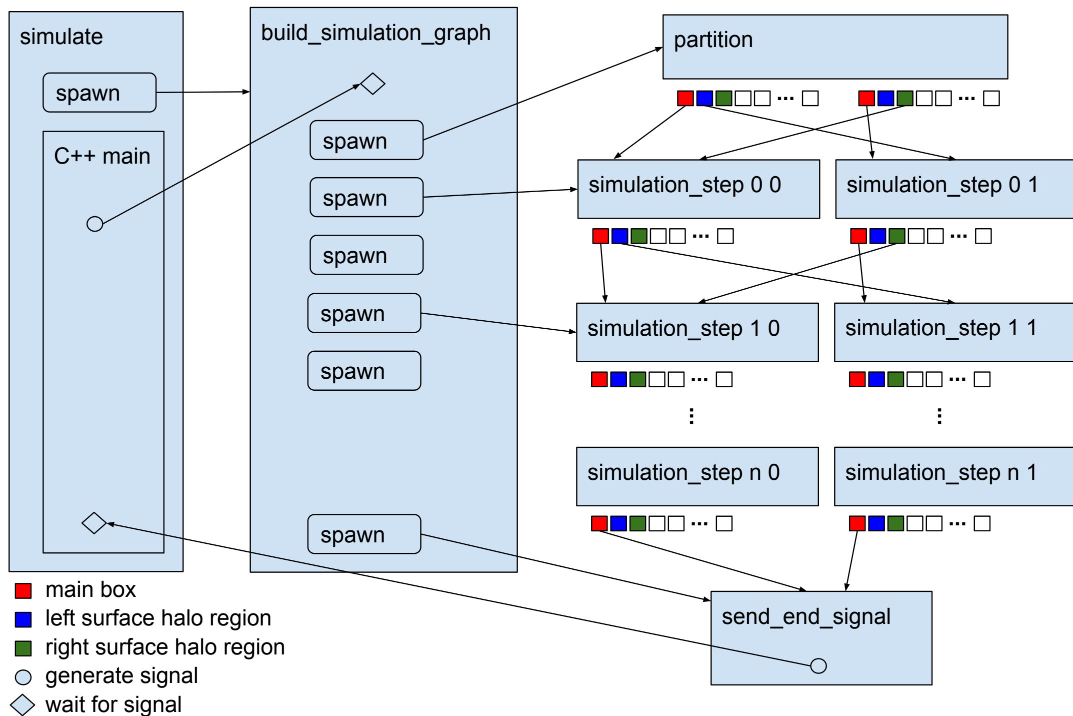

Figure 3: Task tree of a 2-1-1 partitioning scheme

c Consider local availability of blobs

This is an optimization to help with executing tasks on the node that has the most data locally available to it.

In Building task dependency tree, we used the returned futures to represent the main box and required halo regions. Ray generates deterministic identifiers for these futures when a task is scheduled and thus the C++ code cannot make use of these IDs. Because these identifiers are (certainly) different from the identifiers of the blobs, as generated by the C++ at execution time, the schedulers do not know about the sizes of the actual blobs.

However, since the IDs of the blobs are deterministically generated, we could force Ray to use these IDs for the returned futures, and hence when the actual blobs are made available later, their sizes will be taken into account by the schedulers.

This optimization requires an enhancement in Ray so that its remote functions will also take in a list of identifiers to be used for the returned futures, and no deserialization will be performed when these blobs are fetched. This also means that the Python and C++ side must agree on the method to generate these IDs.

3 Result

With the method discussed, we have achieved the following encouraging results.

1. For the first time, a BioDynaMo simulation was run in a multi-process, multi-node setup. We launched a Ray cluster in 2 docker containers each having 2 worker processes, and ran the driver on yet another docker container, all in a 4-core machine.

2. Virtually no change to the current simulation code. The only minor change (to switch to `RaySimulation`) could be integrated into the runtime portion of BioDynaMo later.

3. The driver is also able to run the simulation in a local setup, in the same way these simulations have always been run before.

4 Future work

As positive as the results could be, there are many outstanding issues that have to be addressed for the distributed mode of execution to be useful.

a Reconcile movement of simulation objects

The simulation objects are currently assumed to not move from one region to another. This is always a false assumption and must be relaxed.

On top of that, reconciliation of the movement is also required. For example, if a cell from one box moves across the border to its neighbor box, the neighbor must be informed of that. In this case, there is a change in authoritative source of the cell. Conversely, to save compute time, we might skip the cells in the halo regions that a task pulls in, since those cells are not in the space that this task is authoritative for.

b Alleviate serialization and deserialization overhead

We are seeing around 90% of execution time is spent on reassembling and disassembling of the halo regions. This time is attributed mostly to the serialization and deserialization of the blobs from Plasma store. Even though these blobs are in a shared memory among all processes on the same node, they cannot be directly treated as regular simulation objects. If the BioDynaMo runtime could devise a mechanism to store objects in a position independent manner, that would alleviate this major overhead.

c Ensure deterministic simulation

Random number generators are used extensively in simulations. In a local execution, there is only one generator used by all threads. In distributed mode, a task could run on a random node, which means that the source of random number is not deterministic. This affects reproducibility of the simulation. There needs to be a more deterministic way to seed the random number generator at the start of each task.

d Relieve memory usage and support resumption

Since the simulation happens in locked steps, it is possible to remove blobs from at least 2 generations ago. This will require enhancements to Plasma object store because it currently does not offer APIs to do that.

In the same vein, it should be possible to resume the simulation from any step in case of sudden failures of any node, or inavailability of any removed blob.

e Distribute the initialization of the global space

The initialization of the global space is done in one node at the moment. Because we partition the global space into cubes, this work could also be distributed across the whole cluster. Care must be taken to make this step resumable, too.

f Validate simulation results

We have validated that the simulation can run in a distributed manner but no effort has been spent on validating the correctness of the simulation results.

g Upstream enhancements to Ray

It could be helpful to work out a more comprehensive solution and upstream the modifications we have made to Ray (see Consider local availability of blobs).

5 Conclusion

Due to off-loading of execution to Ray framework, we were able to make BioDynaMo run its first distributed demo within two months with virtually no change to existing simulation logic. The results are encouraging but still far from being useful. Much more work needs to be done to carry out a reproducible and realistic simulation, reduce resource consumption both in storage and compute, and validate the simulation results.

6 Postscript

This report was written about 2 weeks before the end of the project. Since then, some of the recommended future works were also designed and implemented, the most important one being mentioned in Reconcile movement of simulation objects. This section discusses more about that work.

Originally, the halo regions were *internal* to a box. This ensured that all the simulation objects contained in the halo regions were under the authority of the box. This work expanded the halo regions *outside* the box so that it would be able to catch the simulations objects that had moved from inside to outside the box.

There were three main issues to tackle in this design. The first was to keep track of the simulation objects that were originally in a main box so that the simulation step only applied to them, and their movements were fully captured. The second was to limit simulation to a limited portion of a resource manager (an container of all simulation objects). The third was to accept a manually specified global bounding box that all simulation objects were assumed lost if they moved out.

The first issue was solved by arranging the simulation objects within the bounds of a main box to the front of the resource manager. We only needed, then, to keep track of how many of them there were. When iterating through the resource manager again, if the object was among these very first objects, we knew that this box was authoritative for it.

- At re-assembling time, all simulation objects in the main box were automatically copied to the resulting resource manager because all simulation objects here should have been within the bounds. Then we traversed through all 26 halo regions to append inside-the-box simulation objects to the resource manager. A current count of all simulation objects was taken at this step. These objects would be the *managed* simulation objects. Finally, another traversal pass was done to copy remaining simulation objects to the resource manager. These are *unmanaged* objects.

- At dis-assembling time, only managed objects are considered. Each managed simulation object was copied to its corresponding main and halo regions. For example, if the managed object had moved from one inside corner to outside the box, it would be copied to all the halo regions overlapping where it was right now, and it would not be copied to the main box for the next step.

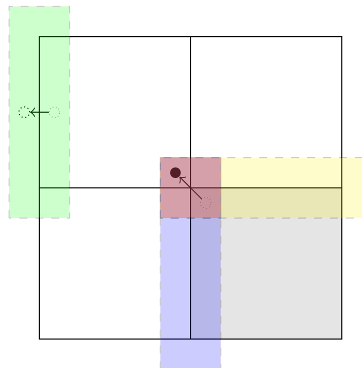

Figure 4: 2D illustration of halo regions involved when one simulation object moves

In 2D, figure 4 illustrates all participating halo regions when a simulation object moved from inside a corner to outside the box. The object is removed from the main region (gray square), copied to the halo regions (yellow, blue, and red rectangles).

Figure 4 also depicts another problem. When an object in the upper left box moved out of the global bounds, since no tasks would pull that green halo region in, that object would be lost. Therefore, the users should appropriately size this global bounding box to the scale of the simulation. An alternative approach here is to adjust the global bounding box dynamically after each step. But this approach introduces a synchronization point and much more complexity because repartitioning might be necessary due to the size adjustment.

Bibliography

BioDynaMo authors. BioDynaMo source respository. URL https://github.com/BioDynaMo/biodynamo.

Lukas Breitwieser. BioDynaMo: Large-scale simulation for biological research. Technical report, CERN openlab, CERN, Geneva, Switzerland, January 2017. URL https://indico.cern.ch/event/609597/contributions/2457806/attachments/1403405/2143673/MarcoManca-biodynamo.pdf.

Lukas Breitwieser and Ahmad Hesam. BioDynaMo – Biology dynamics modeller. Technical report, CERN openlab, December 2017. URL https://indico.cern.ch/event/655459/contributions/2669818/attachments/1569898/2475927/20171201_biodynamo_it_technical_forum.pdf.

Lukas Breitwieser, Roman Bauer, Marco Manca, and Fons Rademakers. Porting a Java-based brain simulation software to C++. Technical report, CERN openlab, November 2015. URL https://zenodo.org/record/46842#.W3Zri3UzZqv.

Lukas Breitwieser, Roman Bauer, Alberto Di Meglio, Leonard Johard, Marcus Kaiser, Marco Manca, Manuel Mazzara, Fons Rademakers, and Max Talanov. The BioDynaMo project: Creating a platform for large-scale reproducible biological simulations. In *Proceedings of the Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4)*, volume 1686 of *CEUR Workshop Proceedings*, University of Manchester, Manchester, UK, September 2016. CEUR-WS.org. URL http://ceur-ws.org/Vol-1686/WSSSPE4_paper_14.pdf.

Rene Brun and Fons Rademakers. ROOT - An object oriented data analysis framework. In *AIHENP'96 Workshop, Lausane*, volume 389, pages 81–86, September 1996. URL https://root.cern.ch.

Andreas Hauri. *Self-construction in the context of cortical growth from one cell to a cortex to a programming paradigm for self-constructing systems*. PhD thesis, ETH Zurich, 2013. URL https://doi.org/10.3929/ethz-a-009997273.

Konstantinos Kanellis, Fons Rademakers, and Lukas Breitwieser. Scaling a biological simulator platform to the cloud. Technical report, CERN openlab, August 2017. URL https://zenodo.org/record/1204348#.W3ZrhnUzZqv.

Philipp Moritz and Robert Nishihara. Plasma in-memory object store, August 2017. URL https://arrow.apache.org/blog/2017/08/08/plasma-in-memory-object-store/.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, December 2017. URL http://arxiv.org/abs/1712.05889.

Redis authors. Redis. URL https://redis.io.