Bulk Synchronous Visualization

Lars Ailo Bongo*

Department of Computer Science
University of Tromsø
Norway

Abstract

Many biological applications require computationally expensive high resolution visualizations. One such example is visual analysis of multiple gene expression datasets. Large desktop displays and display walls may provide the required resolution, and current multi- and many-core processors often have the required computational resources. However, it is still challenging to write programs that can utilize high resolution displays and multi-core processors. This extended abstract describes the *bulk synchronous visualization* (BSV) model for interactive parallel computation and visualizations. The data to be visualized is split into multiple visualizations, each shown in a separate window that is rendered by a separate plotter. The user controls the plotters through an interactive Python shell by calling functions to visualize, filter, and mange plotter windows. The system is designed to efficiently explore hundreds of small visualizations. It provides efficient window management, and it makes it easier to write parallel visualizations since process management, multi-threading, and distribution is hidden from the user.

Keywords: Python, bulk synchronous parallelism, biological visualization, display walls

Index terms: D.1.3 [Programming Techniques]: Concurrent Programming Distributed programming; D.2.13 [Software Engineering]: Reusable Software--Reusable libraries

1 Introduction

Many biological applications require computationally expensive high resolution visualizations. One such example is visual analysis of multiple gene expression datasets, where simultaneous integrated display of many datasets can provide novel biological insights that are not apparent when displaying only one dataset at a time [1].

High resolution displays are readily available either as large format monitors, multiple monitors connected to a computer, or as tiled display walls where multiple computers with one or more monitors or projectors are coordinated to provide one high resolution display [2]. In addition, current computers have very powerful multi-, or many-core processors. These therefore typically provide the required resources for biological visualization applications.

However, writing a program that can utilize high resolution displays and multi-core processors is challenging. First, to utilize multiple processor cores requires writing either a multi-threaded program to be run on a shared memory computer, or a distributed program to be run on a distributed memory computer cluster. Second, to write a visualization program that performs well on a high resolution screen it may be necessary to use low-level graphics primitives to achieve required performance, or to do window management if there are multiple sub-visualizations. All of the above requires either advanced programming knowledge or many days of developer time, which often leads to underutilization of the available resolution and computational resources.

The developer time is justified for visualization tools with many users such as business intelligence tools [3-5], omics visualization tools [1,6], or scientific parallel visualization tools [7]. But there are many cases where a single user needs to quickly visualize some data using an easy to use visualization environment such as MATLAB [8] or pyplot [9].

It is also possible to reduce the amount of data to be visualized by using techniques such as clustering or statistical analysis. However, many users do not have the knowledge required to use these techniques or they may want to do some simple visualizations to quickly look at the non-reduced data before using these advanced techniques.

We propose the *bulk synchronous visualization* (BSV) model for interactive parallel computation and visualizations (inspired by the bulk synchronous programming model [10]). It assumes that the data to be visualized can be split into multiple parts that can be computed and visualized independently. Each part is displayed in a separate window and run in a separate process. The user controls the visualization through an interactive Python shell by calling functions that filter visible tasks, or show next set of tasks. The model provides several advantages:

- The program is sequential. It is therefore not necessary to implement data synchronization and protection as would be necessary for multi-threaded programs.
- The number of displays can easily be scaled. It is therefore easy to quickly write and test a function for visualizing one part of the data, and then viewing the rest of the data in bulk by running the function in parallel for all parts.
- The program can be run on a distributed memory cluster without writing message passing or other synchronization code.
- Low level process management is hidden from the user.
- The system provides efficient window management and filtering such that an analyst can quickly iterate over hundreds of sub-graphs shown in hundreds of windows.

The system is work in progress. The rest of the extended abstract describes the programming model, the design and implementation of the BSV system, and a use case.

2 Programming Model

A BSV program consists of a *coordinator* process that orchestrates computation and visualizations executed by multiple *plotter* processes that may run on multiple computers. The *coordinator* is an interactive Python script where the user can call the BSV API functions. The user can also write at runtime a visualization function to be executed by the *plotters*.

The *coordinator* first initializes the data structures to be visualized, typically by reading and parsing data from input files.

* larsab@cs.uit.no

It then splits the data into multiple parts and assigns each part to a *plotter*. A *plotter* process is created by forking the coordinator process, so each *plotter* contains a replica of the coordinator data structures in a separate address space. A modification by one *plotter* process is therefore not visible to other *plotters*. A *plotter* process first initializes a window and then executes visualization functions received from the *coordinator*. All running *plotters* are synchronized with respect to the visualization commands executed. To visualize different parts, the programmer provides a list with arguments to be sent to each *plotter*. A *plotter* may also receive a show or hide window command, a command to move or resize the window, or a kill command. The hide and kill commands may be sent in the form of a filter function that is evaluated to determine whether to hide the window of a plotter.

The *coordinator* keeps a list of all executed commands, so it can kill a *plotter* process and later restart it by sending it the list of commands to be executed in order to synchronize the visualization with the other currently visible visualizations.

```
[1]: matrix = readMatrixRows()
[2]: rowIndxs = range(len(matrix))
[3]: def viz1(indx):
       plot(matrix[indx])
[4]: coordinator = bsv.Coordinator(nWindows)
[5]: coordinator.visualize(viz1, rowIndxs)
[6]: coordinator.showRandom(nWindows)
[7]: def filter1(indx):
       for i in range(len(matrix[indx])):
         if matrix[r][indx] < 0:
           return False # hide window
       return True
[8]: coordinator.filter(filter1, rowIndxs)
[9]: coordinator.showFirst(nWindows)
```

Figure 1: A simplified BSV program for plotting rows of a matrix in separate windows, showing a random subset of plots, and then hiding all plots with negative values.

3 Design and Implementation

The BSV system is designed to provide efficient interactive exploration of large visualizations split into hundreds of windows. We take advantage of four properties of current computers and operating systems. First, there is enough DRAM to keep many visualization processes in memory at once. Second, if the operating system implements fork using copy-on-write the resident set size of the child processes will be small even for processes with large data structures that are mostly read only. Third, there are compute resources available for running computation on hidden windows. Fourth, the fork system call has low overhead.

The BSV system therefore runs many visualization processes simultaneously, but only a few of these are visible at a given time. In addition the system predicts which visualizations are likely to be shown in the near future, and if needed starts these process in the background such that the visualizations are ready when requested by the user. The prediction is easy to implement if the user views the visualizations in an order specified at the time the data was split into multiple parts. Such an order can be based on for example indexes in a matrix, dataset names, or graph properties such as size.

We have implemented the BSV system in Python. We use the multiprocessing module to fork child processes and use pipes for coordinator-child process communication.

In Python, functions can be saved as objects and sent over a pipe or socket to another process. Simple functions can therefore be written by the user at run time. We use pylab [11] for numerical computation and graph plotting, and iPython [12] as the interactive shell and for parallel computing on a display wall cluster.

4 Use Case

BSV was motivated by the need to understand the behavior of an algorithm we developed for removing overlapping *samples* from *series* downloaded from NCBI GEO. Overlap is removed by creating graphs with edges between all *series* that have one or more overlapping *samples* and then iteratively removing samples until the overlap between two *series* is less than a maximum specified as a parameter. The program outputs a log file with the graphs and lists of removed *series* and *samples*.

The BSV *coordinator* reads in the log file and assigns each of the 1636 graphs to *plotters* that visualize these using the NetworkX Python package for drawing dot [13] graphs in pylab [11]. In the resulting visualizations, removed *series* are marked using different colors and styles, and the overlap is shown using edge labels.

We can fit about 30 windows on a 2560x1440 pixel screen, and about 100 windows on our 6144x2304 pixel display wall. We first viewed the first tens of graphs, and then a few tens of randomly selected graphs to get a rough idea about how the algorithm removes overlap. We then studied significant details by writing filter functions to only show graphs with certain properties such as graphs that contains superset or duplicate *series*, graphs with at least N overlapping *samples* between a pair of *series*, or graphs that contains a *series* X or a *sample* Y.

To parse the log file and create the graph data structures we wrote about 200 line of code (LOC). The final visualization function was about 60 LOC, mostly for specifying the node and edge styles to use in the graph. The filter functions were less than 10 LOC.

The use case demonstrates that BSV is useful to write and scale up simple visualizations that use the many visualization libraries available for Python.

References

[1] M. Hibbs, G. Wallace, M. Dunham, K. Li, O. Troyanskaya. “Viewing the Larger Context of Genomic Data through Horizontal Integration”. *Proceedings of 11th International Conference Information Visualization (IV '07)*. 2007. pp. 326-334

[2] K. Li, H. Chen, Y. Chen, D.W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, J.P. Singh, B. Shedd, J. Pal, G. Tzanetakis, J. “Zheng. Building and using a scalable display wall system”. *IEEE Computer Graphics and Applications* 20 (4). 2000. pp. 29-37

[3] http://www.qlikview.com/

[4] http://spotfire.tibco.com/

[5] http://www.tableausoftware.com/products/desktop

[6] N. Gehlenborg, S. I O'Donoghue, N.S. Baliga, Alexander Goesmann, M. Hibbs, H. Kitano, O. Kohlbacher, H. Neuweger, R. Schneider, D. Tenenbaum, A-C. Gavin. “Visualization of omics data for systems biology”. *Nature methods* 7 (3 Suppl), 2010. pp. S56-68.

[7] http://www.paraview.org/

[8] www.mathworks.com/products/matlab/

[9] http://matplotlib.sourceforge.net/api/pyplot_api.html

[10] L. G. Valiant. “A Bridging Model for Parallel Computation”. *Communications of the ACM* 33(8), 1990, pp. 103-111.

[11] http://www.scipy.org/PyLab

[12] http://ipython.org/

[13] E. Gansner, E. Koutsofios, S. North. “Drawing graphs with dot”. http://www.graphviz.org/Documentation/dotguide.pdf. 2006