BSV - 1000 Window Visualization System

Lars Ailo Bongo
Department of Computer Science
University of Tromsø
larsab@cs.uit.no

Abstract

Many analytics applications require computationally expensive high resolution visualizations. 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* 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 process. All processes are initialized with the same replicated state, and they receive visualization functions to execute from a coordinator process. The coordinator is an interactive Python shell where the user writes visualization functions at runtime, and does window management. The system is designed to efficiently explore hundreds of graphs. It provides efficient window management, and it makes it easier to write parallel visualizations since process management and distribution is hidden from the user, and there is no need for multi-threaded programming.

1 Introduction

Many analytics applications require computationally expensive high resolution visualizations. One such example is analysis of biological data, where the simultaneous integrated display of many datasets can provide novel biological insights that are not apparent when displaying only one dataset at a time [1]. Another example is visual analytics for business intelligence [2-4].

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 [5]. In addition, current computers have very powerful multi-, or many-core processors. These therefore provide the required resources for visual analytics 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 the underutilization of the available resolution and computational resources.

The developer time is justified for visualization tools with many users such as business intelligence tools [2-4], genomics visualization tools [1,6], or scientific parallel visualization tools [7]. But there are many cases where a user needs to quickly visualize 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 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 using operations such as 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 a subset of the data, and then viewing the rest of the data in a bulk by running the function in parallel for the remaining parts.
- The program can be run on a distributed memory cluster without writing message passing or other synchronization code.
- The low level process management is hidden from the user.
- The system provides efficient window management 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 system, and initial evaluation results.

2 Architecture and 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 write visualization code to be executed on the plotters. In addition to visualization code execution the API also provides a simple interface for window management. The coordinator communicates with either a BSV local coordinator in case of distributed execution, or directly with a BSV plotter if run on a single computer.

The plotters receive commands to be executed from a coordinator. A plotter is started by cloning (forking) the coordinator so each plotter has a replica of all the coordinators data structures that can be read and modified without any communication or synchronization. A plotter visualizes or plots data by calling functions from a Python library such as *matplotlib*, or *mayavi*. BSV wraps a few window management functions of the library, but the plotter can also call all library functions directly. The high-level Python library typically runs on top of the operating systems window manager.

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

Figure 1 shows an example BSV program. A matrix is read from a file and parsed in (1). The matrix is then split into multiple blocks that are visualized independently. We assume the user provided split function returns a start and end row of each block that is saved in the blocks variable (2). A visualization function is in (3). This function receives the start and end row index of its block by the system. The BSV coordinator is started in (4), and the coordinator receives a visualization function to be executed on each visible visualization processes (5). There will be one visualization process for each block, but only 60 blocks are visible at a time as specified by the argument in (4). The user wants to see 30 new random windows (6), and then writes a function that hides all windows in which the first column has a negative value (7 and 8). Finally the first 60 of the non-hidden windows are shown (9).

```
[1]: matrix = readAndParseData()
[2]: blocks = split(matrix) # (start, end)
[3]: def viz1(startRow, endRow)
...:   plot(matrix[startRow:endRow])
[4]: coordinator = bsv.Coordinator(60)
[5]: coordinator.visualize(viz1, blocks)
[6]: coordinator.showRandom(30)
[7]: def filter1(startRow, endRow):
...:   for i in range(startRow, EndRow):
...:     if matrix[i][0] < 0:
...:       return False # hide window
[8]: coordinator.filter(filter1, blocks)
[9]: coordinator.showFirst(60)
```

Figure 1 A simplified BSV program.

3 Design and Implementation

The BSV system is designed to provide efficient interactive exploration of large visualizations split into thousands 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 a predetermined order. Such an order can be based on for example indexes in a matrix, dataset names, or visualization 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.

5 Initial Evaluation and Conclusions

The initial questions we want to evaluate are: (i) how well the system scales with regards to the number of visualization windows and processes, (ii) what is the latency of starting a new visualization process, and (iii) what is the latency of switching between multiple visualization windows at once? Combined these will give an indication about the interactive performance of the system, and the number of visualization processes that can be running at once.

BSV was motivated by the need to understand the behavior of an algorithm that removed overlapping of samples in the GEO series [13]. This was implemented by using dot [14] to visualize graphs of datasets with overlapping samples, and then displaying the 1636 resulting graphs on a high resolution display.

Our initial evaluation of the BSV system running the overlapping samples analysis on a Windows 7 laptop, with 8GB DRAM and 4 CPU cores, shows that 100 visualization processes use about 5.4GB of DRAM. The time to hide 10 window and then show 10 windows from running processes is less than a second, and it takes about 2.7 s from a new process is started until it has displayed its visualization.

The initial results demonstrate that the system scales to hundreds of windows, and it provides the low window management latency required for interactive visual analysis. Finally, we found it easy to quickly implement visualizations that allowed us to understand the behavior of the above described overlap removal algorithm.

Acknowledgments

Thanks to Lars Tiede for his many suggestion for improvements to this paper.

References

[1] Viewing the Larger Context of Genomic Data through Horizontal Integration Matthew Hibbs, Grant Wallace, Maitreya Dunham, Kai Li, Olga Troyanskaya (2007). *Proceedings of 11th International Conference Information Visualization (IV '07)*. p. 326-334

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

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

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

[5] Building and using a scalable display wall system. 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 (2000). *IEEE Computer Graphics and Applications* 20 (4) p. 29-37

[6] Visualization of omics data for systems biology. Nils Gehlenborg, Seán I O'Donoghue, Nitin S Baliga, Alexander Goesmann, Matthew A Hibbs, Hiroaki Kitano, Oliver Kohlbacher, Heiko Neuweger, Reinhard Schneider, Dan Tenenbaum, Anne-Claude Gavin (2010*). Nature methods* 7 (3 Suppl) p. S56-68.

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

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

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

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

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

[12] http://ipython.org/

[13] Edgar R, Domrachev M, Lash AE.Gene Expression Omnibus: NCBI gene expression and hybridization array data repository. *Nucleic Acids Res*. 2002 Jan 1;30(1):207-10

[14] Emden R. Gansne, Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 1999. 00(S1), 1–5