Bulk Synchronous Visualization

Lars Ailo Bongo
Department of Computer Science,
University of Tromsø,
N-9037 Tromsø, Norway
larsab@cs.uit.no

ABSTRACT

Many visual 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. We describe the bulk synchronous visualization (BSV) model that makes it easier to write high resolution parallel visualizations. The dataset to be visualized is decomposed into thousands of tasks that are assigned to sequential processes. These are then run in parallel by the BSV system which provides efficient process and window management. BSV takes advantage of the large DRAM size and multiple cores of current computers, and the copy-on-write and low overhead fork mechanisms provided by current operating systems. We have implemented three BSV applications and used these to identify advantages and limitations of BSV on Windows, Linux and OS X. The results demonstrate that BSV makes it easy to implement visualization applications that utilize high resolution displays and multi-core processors.

Categories and Subject Descriptors D.1.3 [**Programming Techniques**]: Concurrent programming, Distributed Programming. D.2.13 [**Software Engineering**]: Reusable Software—Reusable libraries

General Terms Measurement, Performance, Design

Keywords Python visualizations; bulk synchronous parallelism; display walls; interactive performance; multi-core processors; window management.

1. INTRODUCTION

Many analytics applications require computationally expensive high resolution visualizations. One such example is bioinformatics, 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]. Other examples includes meteorological simulation [2], plasma physics experiment control systems [3], and tools for text analytics [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 typically provide the required resources for visualization applications.

However, writing a program that can utilize high resolution displays and multi-core processors is challenging. First, to utilize multiple CPU 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. Multi-threaded programming is especially challenging when combined with GUI libraries that often assume an event based programming model with a single thread doing all updates to the visualization. Second, to write a visualization program that performs well on a high resolution screen it may be necessary to use low-level graphics libraries such as Direct X [6], Open GL [7], or VTK [8] to achieve required performance, or to do manual 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 [9], genomics visualization tools [1][10], or scientific parallel visualization tools [11][12]. 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 or pylab [13]. But these visualizations often do not scale to high resolution displays.

It is also possible to reduce the amount of data to be visualized by using techniques such as clustering or other statistical analysis techniques. However, many users do not have the knowledge required to use these techniques or they may want to do some simple visualizations to quickly get an overview of the data [14].

We propose the bulk synchronous visualization (BSV) model for interactive parallel computation and visualizations. BSV is designed for MATLAB-type visualizations on high resolution displays including display walls. The most important requirements are therefore ease of use, short developer time, scalability and distributed execution. The BSV system provides efficient process and window management by taking advantage of the large DRAM size and multiple CPU cores of current computers, and the copy-on-write and low overhead fork mechanisms provided by modern operating systems.

The dataset to be visualized by BSV program is first decomposed into thousands of tasks that are assigned to plotter processes. Each plotter process has a separate address space and a separate window. The plotter processes are run in parallel by the BSV system. The program can be scaled with respect to processor cores and screen resolution by respectively adjusting the number of running and visible plotters. Window management is provided by

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PMAM'2013, February 23, 2013, Shenzhen [Guangdong, China]
Copyright © 2013 ACM 978-1-4503-1908-9/13/02... $15.00

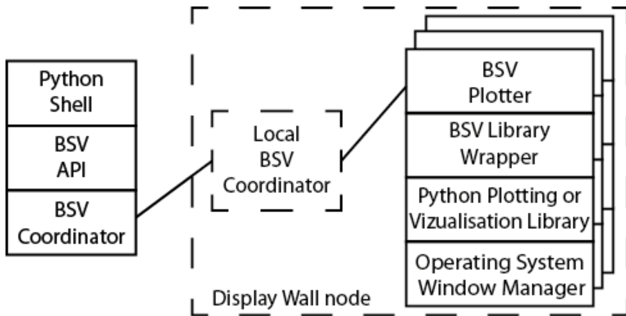

Figure 1. BSV architecture. A BSV coordinator orchestrates the visualizations of multiple BSV plotters.

interactive functions that filter visible plotters, or functions that show a set of plotters.

We have implemented three BSV applications and used these to evaluate the interactive performance of BSV on Windows, Linux and OS X. The results demonstrate that BSV makes it easy to program visualization applications that utilize high resolution displays and multi-core processors. We also identify advantages and limitations of the BSV model. Our conclusion is that BSV makes it easy to implement visualization applications that utilize high resolution displays and multi-core processors.

2. ARCHITECTURE

A BSV program is expressed as a series of visualization functions executed by many *plotter* processes. The data to be visualized is decomposed and assigned to plotter processes. Each plotter has a separate address space and a window. The visualization is orchestrated by a *coordinator* process that synchronizes the visualization shown on each plotter, and maps plotter processes to processors and plotter windows to screen space. Plotters may be mapped to CPU cores distributed on multiple computers (figure 1).

The coordinator is typically loaded into an interactive shell 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 process is started by cloning (forking) the coordinator process so each plotter has a replica of all the coordinator's data structures that can be read and modified without any communication or synchronization. A plotter visualizes or plots data by calling functions from visualization libraries. BSV wraps a few window management functions of the visualization library, but the plotter can also call all library functions directly. The high-level visualization library typically runs on top of the operating system's window manager.

2.1 BSV vs. BSP

BSV is inspired by the bulk synchronous programming (BSP) model [15] where a program is expressed as supersteps that typically comprise computation, point-to-point communication, and a globally synchronizing barrier. BSP programs are also typically overdecomposed such that many processes are mapped to one processor core.

In the initial design, BSV was intended to provide window management for visualizations implemented as BSP programs. But there are three main differences between a parallel computation and the parallel visualizations indented for BSV. First, the BSV plotter processes typically do not require point-to-point communication with other plotters. Second, the result of a BSV plotter is typically a visualization of which only a subset are viewed by the user. Third, a BSV program is often an order of magnitude more overdecomposed than a BSP program.

Based on these observations, the BSV design assumes that plotter processes do not have point-to-point communication and that the barrier is implicit. BSV therefore has *virtual* supersteps that only guarantee that all plotter processes will eventually execute all supersteps (typically during user think time). It is therefore not necessary to start all plotters for each superstep. However, BSV provides a barrier in the form of a gather-all operation which can be used to implement real supersteps that may include point-to-point communication.

2.2 BSV Programming Model

The coordinator first runs application specific code to initialize the data structures to be visualized. This is typically done by reading and parsing data from input files. It then runs application specific code to decompose the data and assign the parts to plotter processes. After starting plotter processes, it is assumed that the coordinator will not modify the application data structures; such that additional plotter processes started at a later time have identical state. A plotter process first initializes a window and then executes visualization functions received from the coordinator (a visualization function corresponds to a *superstep* in BSP). These functions read and write the data structures and draw in the window. To visualize different parts, the user 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 visualization functions, so it can kill a plotter process and later re-create it by sending it the list of functions to be executed in order to synchronize the visualization with the other currently visible visualizations.

In the program in figure 2, a matrix is read from a file and parsed in (1) using application specific code. The matrix is then split into multiple blocks that are visualized independently. We assume the application specific 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 and executes the code to visualize the data. 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). 30 random windows are shown (6), and then a filter function hides all windows in which the first column has a negative value (7 and 8). Finally the first 30 of the non-filtered 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(30)
```

Figure 2. A simplified BSV program.

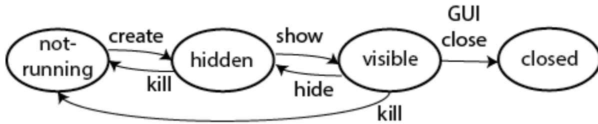

Figure 3. BSV plotter states.

2.3 Local and Distributed Execution

BSV can either be run on a single computer or distributed on a display wall cluster. On a single computer BSV visualizations are executed using multi-processing. There is one plotter process per task, and a single coordinator process. However, at a given time only a subset of the plotter processes are runnable, since the coordinator implements scheduling by killing and creating plotter processes. In addition the coordinator determines which windows are shown and hidden (figure 3). The user can close a plotter permanently using the application specific GUI (if any). The operating system scheduler decides which runnable plotters to run on the different CPU cores (or GPUs).

BSV can be run distributed on multiple computers, for example in order to use a display wall cluster. First, a local coordinator process is started on each node. The user then controls the visualization using a master coordinator that distributes visualization commands to local coordinator and implements global scheduling of visible windows. The local coordinators distribute the received commands to their plotters and implements scheduling of running plotter processes.

Distributed execution requires coordinating data management. For small datasets all local coordinators can maintain a replicated dataset. For large datasets, it may be necessary to partition the data. In BSV the master coordinator does the partitioning. The application can use libraries or infrastructure services for distributed data access or data streaming. BSV does not provide security features such as user authorization and auditing of code to be executed.

BSV programs can be run on a GPU if the visualization functions use a visualization library with for example openGL or openCL mappings. But the BSV coordinator will schedule these plotters similarly to plotters that only use CPU cores.

3. DESIGN AND IMPLEMENTATION

The BSV system is designed to provide efficient interactive exploration of large visualizations decomposed into hundreds of windows. The system design is motivated and based on four assumptions about 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 if these are mostly read only. Third, there are compute resources available for running computation on hidden windows. Fourth, the create process (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. Such prediction is easy to implement if the user views the visualizations in a predetermined order. The order can be based on for example indexes in a matrix, dataset names, or task properties such as size. Since BSV uses multi-processing it supports visualization libraries that are not thread-safe.

We have implemented the BSV system in Python. We use the multiprocessing module to fork plotter processes and use pipes for inter process communication. In Python, functions can be saved as objects and sent over a pipe or socket to another process. Visualization and filter functions can therefore be written by the user in and interactive shell such as iPython [16]. We use the marshal module to dump the functions func_code on the coordinator and to load the function into the global namespace of the plotter (or local coordinator).

We use pylab [13] for numerical computation and graph plotting, and for parallel computing on a display wall cluster. BSV implements wrappers for matplotlib [17] run on top of the wxPython [18] and TkAgg [19] GUI backends. We also implements wrappers for pyglet [20] and TkInter [19].

3.1 Fork and Copy-on-write

Copy-on-write is an optimization technique that is used by operating systems to implement address space cloning. For example in Linux the fork system call will create page tables that point to the same pages as the parent. These are both fast to create and have low memory overhead. The pages are shared until the child writes to a page. We assume most data structures used by BSV applications are read-only such that many pages can be shared.

The multi-processing module in Python 2.7.2 uses the fork system call that creates a new address space for the child using copy-on-write in Linux and OS X. In Windows 7, it uses the CreateProcess function in the Win32 API that does not implement copy-on-write. A plotter process will therefore create a new address space, start a new Python interpreter, and run the main module of the interpreter. The implications for interactive performance are evaluated in Section 6.

3.2 Show Prediction

The BSV system runs many visualization processes simultaneously, but only a few of these are visible at a given time. The coordinator predicts which visualizations are likely to be shown, and attempts to keep these plotters running in the background. The scheduler implementation assumes that the tasks are organized in a 1-dimensional array using an application specific order (figure 4), and that the user traverses the tasks in the order of the array. It therefore attempts to keep tasks adjacent to visible tasks running. The scheduler is run each time a window is hidden, and it replaces one plotter process per hidden window in each turn. The least recently visible plotters are selected for replacement.

3.3 Coordinator and Plotter Implementation

The coordinator is implemented as a Python class that exports an interface to be used from an interactive shell. For distributed execution the coordinator functionality is split between the master coordinator and the local coordinator. The master is then typically

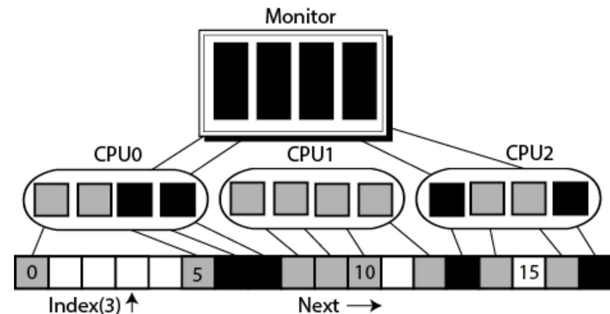

Figure 4. Of the 18 plotters, 4 are visible and mapped to an area on the screen (black), 8 are hidden and mapped to three processor cores (gray) and 6 are not running (white). To next function in BSV will switch visible plotters to 0, 8, 9, and 14. The index(3) function will show plotters 3—6.

Table 1. BSV coordinator API.

Functions	Arguments
[start, stop]Plotters	Number of plotters to start, list of plotter arguments, state object
visualize	Visualization function, list of plotter arguments
layoutGrid, layout	Grid layout, or list of window sizes and positions
show[Next, Previous, Random, First, Index, Indexes]	(function dependent)
filterVisible, pendingFilterVisible, filterOff	Filter function, list of plotter arguments
gather, readLogFiles	Gather function, list of plotter arguments
executeLocally	Function

run as an interactive shell, and the local coordinator as a Python program. Both have in addition to the shell thread, a network layer thread as described below.

The plotter is a Python program with an IPC thread and one or two visualization threads. For visualization libraries that support multi-threading one thread runs the GUI loop, while another receives and executes window management commands and visualization functions. Otherwise, a single thread must run both the GUI loop and execute received commands. It can therefore block while waiting for a new GUI event, or a command from the coordinator. Blocking in the GUI event loop will add additional latency to the window management and visualization commands received from the coordinator. Blocking on the coordinator socket will cause the GUI to be unresponsive for the user. Setting a short blocking time on both will use more CPU. We assume the user mostly controls the visualization through the coordinator, so the default blocking time for coordinator commands is 500ms, and 0ms in the GUI loop.

3.4 Inter-Process Communication

Inter-process communication (IPC) is implemented using pipes, queues and sockets. Pipes are used for coordinator-plotter communication. The coordinator has one pipe connected to each of the running plotters. Inter-node (master to local coordinator) communication is implemented using sockets. To avoid blocking the interactive coordinator, the messages are sent asynchronously by using a queue and a separate network thread. The plotter also has a dedicated thread for IPC.

3.5 Limitations

Ideally, BSV visualization functions would be similar to visualization code written without using BSV. However, there are three restrictions. First, the functions must take specific arguments such that BSV can do an upcall (these are references to a state object, a logger, and an arguments data structure). Second, the functions should not use global variables since these may not be supported by the underlying Python mechanisms (in particular multiprocessing on Windows 7). Instead the functions should store all global references in a state object that is passed as argument to all visualization functions. Third, all functions are in the global namespace with names assigned by BSV. It is therefore hard to split the visualization code over several functions.

Some visualization libraries do not work well with BSV. For example, the popular pygame library [21] does not provide programmatic control over window management. Another important limitation is that all of the coordinators data structures should fit in memory such that plotter process startup is fast.

BSV does not currently provide fault-tolerance. If a plotter process crashes the coordinator assumes that the plotter was closed by the user and hence the plotter is not later restarted (we have experienced a couple of crashes during testing and evaluation). However, it is straightforward to modify BSV such that crashed plotters can be restarted by the coordinator.

4. INTERFACES

BSV exports an API to the analyst that is used to control the visualization. The analyst can also interact with the visualization windows using regular GUI operations. There is also an interface for wrapping visualization libraries. Communication between the central coordinator and local coordinators, and local coordinator and plotters is over custom protocols.

4.1 Coordinator API

The API exported by the coordinator to the user (Table 1) consists of eight groups of functions. The first consists of functions to start and stop plotters. For each task there is an entry in a list sent as argument to the startPlotter, visualize, filter, and gather functions. Each per task entry in the list may for example contain the plotter's index in a replicated dataset. In addition, there is a stateObject that is used to store global variables. The reference to the state object is set in startPlotter, and then passed as argument in all plotter upcalls.

The filter, layout, and show function groups provide programmatic control over window management. But it is also possible to manually move and resize the windows.

BSV provides a function to gather values from all plotter by writing a gather-all function that is executed on all plotters, and a function to gather the log file content of all plotters.

For distributed execution executeLocally can be used to send functions to be executed only by local coordinators.

4.2 Library Wrappers

There are many GUI backend, plotting, and visualization libraries for Python each with its own API. BSV uses a wrapper approach to support each library. The wrapper interface consists of six window management functions that must be implemented for each new library (Table 2). For most libraries each function will only require a few lines of code and is straightforward to implement. The visualization functions that implement the application specific visualization are not covered by this interface.

4.3 User Interfaces

The user interacts with the BSV system either through a Python shell or by running a non-interactive Python script. In addition each plotter has a window that may contain a user interface that can be used to for example move or resize the window, interact with visualization, or save it to a file. Such interaction is done independently of BSV.

Table 2. Skeletons for visualization library wrapper functions.

Functions	Description
createWindow	Initialize window resources. Non-blocking.
guiLoop	Setup timer to call plotter run function. Enter GUI loop. Blocking.
closeWindow	Free window resources.
setWindowVisible	Show or hide a window.
getWindowSize	Get window size and position.
setWindowSize	Set window size and position.

5. APPLICATIONS

We have implemented three BSV applications that represent what we believe are typical explorative interactive visualization applications. These also demonstrate how BSV can be used with different visualization libraries. Since we expect BSV to be used mostly for prototyping the main goal is fast development time which evaluate by counting lines of code. Application performance is evaluated in section 6.

5.1 GeoOverlap

BSV was motivated by the need to understand the behaviour of an algorithm we developed for removing overlapping *samples* from *series* downloaded from NCBI GEO [22]. 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 decomposes the data in our sample log file to 1636 tasks. Each task has one graph that is visualized using the NetworkX [23] Python package for drawing dot [24] graphs in pylab [13]. In the resulting visualizations, removed series are marked using different colours and styles, and the overlap is shown using edge labels (figure 6). The graphs are sorted descending based on the number of nodes in the graph.

We started by viewing 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 lines 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.

5.2 PyHidra

PyHidra is a simplified Python implementation of HIDRA [1] (figure 5). A large collection of DNA microarray data are visualized as heatmaps. The advantage of HIDRA compared to other similar tools is that it shows multiple integrated datasets at once. For example, if a user highlights a gene in one dataset, the same gene will be highlighted in all other datasets. PyHidra demonstrate the flexibility and power of BSV for displaying visualizations with a large number of pixels.

Our Python implementation implements the heatmaps, and supports gene highlighting from the command line. The heatmap is drawn in a canvas from the Tkinter GUI toolkit. PyHidra is about 240 LOC of which about 100 LOC is for drawing the heatmap. Most of the visualization code was based on code from the SPELL tool [25].

To select a gene the analyst runs a visualization function on all plotters that draw a rectangle surrounding the highlighted gene's row. PyHidra has a filter function for displaying only the dataset with at least N highlighted genes, and a visualization function for only showing the highlighted genes in a dataset. It is easy to write additional visualization functions to reorder the genes, or do other kinds of filtering.

5.3 Mandelbrot

Mandelbrot is a widely used embarrassingly parallel benchmark that calculates and displays the Mandelbrot set in a two-

Figure5. PyHidra heatmaps with some genes highlighted (1366x768 pixel resolution).

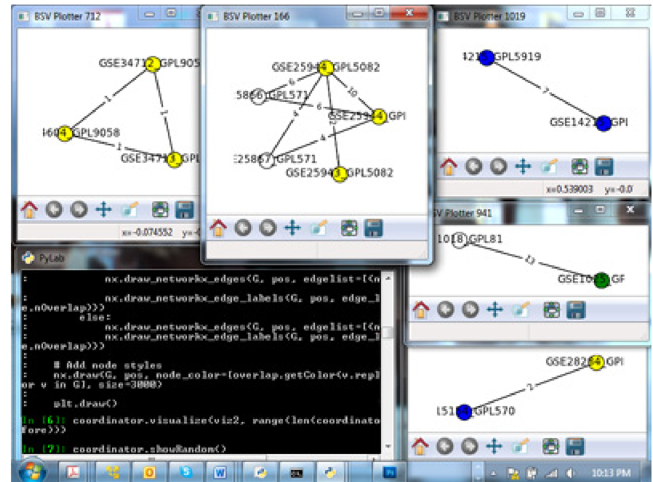

Figure 6. Screenshot of GeoOverlap. Screen resolution has been reduced to 800x600 for increased readability.

dimensional fractal shape. Our benchmark animates N zooms into a region of the image. This application demonstrates that BSV can be used to run computationally intensive applications.

We have two implementations of Mandelbrot; a version that uses the CPU for calculation and Tkinter for visualization, and a version that uses the GPU for calculation and pyglet for visualization.

6. EVALUATION

We evaluate the interactive performance of BSV. In particular we want to answer the following questions: (i) what is the latency of switching visible windows on a high resolution screen?, (ii) what is the latency of starting many plotter processes that each opens a new window?, (iii) how many plotter processes can be run on a single computer, and what limits the number of processes?, (iv) how does interactive performance and process parallelism differ among different operating systems?

In addition, we provided an informal software engineering evaluation in the previous section by counting lines of code for each application to estimate the developer effort.

6.1 Hardware and Software Platforms

We use two hardware platforms. The first is a dual-boot Dell Precision T3500 workstation that has two dual-core Intel Xeon 2.3 GHz processors, 6GB DRAM, and a Nvidia Quadro NVS 295 graphics card. It has two Seagate Barracuda ST31500341AS 7200RPM 1.5TB hard drives. One for Windows 7 (64-bit), and

one for Lubuntu 12.10 (64-bit). The platform has a HPZR30w display with a resolution of 2560x1600 pixels, a Dell 2001FP with 1600x1200pixels, and a Dell USB keyboard. Using this machine we can do direct comparisons between the performance of Linux and Windows.

We also use a Mac Mini, with a 2.5GHz Intel Corei5, 4GB DRAM, a ADM Radeon HD 6630M graphics card, and a 5400RPM 500GB hard drive. It is running Mac OS X 10.7.5 (Lion), and has a Dell2001FP 1600x1200 display and a Dell USB keyboard. We could not use the bigger HPZR30w with the mac mini due to compatibly issues.

We use the Entought Python distribution version 7.3.2-free on all platforms. This version has Python 2.7.3, matplotlib 1.1.0, networkx 1.6, ipython 0.12.1, pyglet 1.1.4, and wxPython 2.8.10.1. However, there is only a 64-bit version for Linux, so we use the 32-bit version on Windows and OS X.

6.2 Microbenchmarks

For the evaluation we use the applications described in section 5 and four microbenchmarks implemented in Python:

1. *startProcs*: start N processes using Pythons multi-processing module. The child processes sleeps 5 seconds before exiting.
2. *bigMem*: start N processes that each allocate a 100 megabytes list and then traverse the list.
3. *openWin*: start N processes that each opens a window and plot a simple graph using matplotlib.pyplot.
4. *switchWin*: start 2*N processes where N of these starts with a visible window, and the remaining have a hidden window. On a keyboard press all N windows will be hidden and N new windows will be shown In each window a barchart is plotted using matplotlib.pyplot. In Windows and OS X the default matplotlib backend is wxPython. In Linux we had to use TkAgg due to errors in wxPython when switching between multiple windows at a time.

In addition we use an image viewer on each platform as a micro-benchmark for native application window open time. We open a 400x300 PNG image (the image for Figure) in Windows Photo Viewer, GPicView in Linux, and Preview in OS X.

6.3 Methodology

To evaluate interactive performance we measure the latency between a user's input until the system responds by changing the display content. Frequently used response time limits are: 100ms for the user to feel that the response is instant, 1 second to avoid disrupting a user's flow of thought, and 10 seconds to avoid losing the users attention [26]. It is also recommended that some form of user feedback is used for latencies above 1 second to indicate that the system is working [27].

6.3.1 Resource usage

For the evaluation we need to measure CPU usage, DRAM usage, virtual memory size, process start latency, time to open new windows, and time to switch between visible windows.

CPU and memory usage is measured using Resource Monitor in Windows, procutils in Linux, and vm_stat in OS X. We report the maximum virtual memory size as reported by the above tools. Time is measured using the *time* function in the Python time module. We also use a high speed camera to measure window open and switch latency (as described below).

To measure DRAM usage we compare the amount of available (or free) memory when the application is running and the amount of available memory when all processes have exited. We assume that most of the memory in use by the processes is made available. We must make this assumption since there is no easy, and portable, way of finding the per process DRAM usage. For example in Linux, the reported per process DRAM usage (resident size) includes shared and copy-on-write memory, and hence the sum of reported DRAM usage of all processes can be many times higher than the amount of DRAM on the system.

Each experiment is run five times on an idle system and the averages are reported.

6.3.2 High speed camera

To measure the latency of window switch and open operations we use two portable approaches. First we add instrumentation to the Python code such that we can measure the time from a coordinator sent a command until the last plotter executed the show or open command. This approach does not include latencies and overhead of the visualization library, the operating system window manager, and hardware latencies.

To overcome this limitation we also use a commodity high speed camera (Casio EX-ZR200) that can film the monitor at 1000 frames per second (224x64 pixels). We watch the video in QuickTime Player version 7.2.2 frame-by-frame until we detect the key-press event and later the display-updated event. The main methodological challenge is to determine when these events occur.

We simplify the measurement by measuring the time between the user (i.e. us) *believes* the key was pressed until the user *believes* the display was updated. We can therefore assume the key pressed event occurs when the key is pressed all the way down. We have a custom made keyboard pressing device that consists of a small metal rod attached to a plank. The rod has a small flag, and the plank has a mark for where the flag position when the key is pressed all the way down. We can usually detect the frame where a key is pressed (the key is pressed very fast).

We assume the display update event is when the first part (typically a shadow) of the window becomes visible on the screen. The window does not appear immediately on the display, but starts as a shadow which gradually becomes less transparent. There may also be animation effects as in the Windows 7 Aero theme, and up to a hundred millisecond delay before the application content becomes visible. We argue that the first appearance is the most important, since the subsequent events are either not detectable for the human perception system, or they are designed to fool the user into believing that the interactive performance is better. When opening, or switching, between multiple windows we assume the display update event occurs when the last window becomes visible. This latency therefore includes user feedback in the form of windows being shown or updated.

6.4 Window Management Performance

Interactive performance is determined by user perceived latencies. For BSV window management these are the time to switch visible windows, and time to open new windows. We measure these using the *startProcs*, *switchWin*, *openPNG* and *openWin* micro-benchmarks. The results provide insight into the advantages of having many plotters running simultaneously compared to starting plotters on demand.

Table 3. Response time for single window experiments. All are in ms. Standard deviation is in parenthesis.

Platform	Start-1	Switch-1	Open-PNG	OpenWin-1
Windows	76 (8)	90 (10)	319 (31)	746 (43)
Linux	2 (0.03)	57 (6)	147 (2)	1427 (9)
OS X	7 (0.09)	97 (6)	520 (5)	1432 (14)

Table 4. Response times for full screen of windows experiments. All are in ms. Standard deviation is in parenthesis.

Platform	Start-28	Switch-28	OpenWin-28
Windows	453 (17)	1509 (53)	5884 (217)
Linux	11 (0.1)	578 (46)	12209 (529)
OS X	25 (0.4)	290 (16)	16420 (240)

Our results show as expected that the time to create new processes that open windows can be two orders of magnitude slower than switching between running processes (Tables 3 and 4).

The latency to start a single Python process is within 100ms on all platforms (Table 3). The Windows latency is an order of magnitude larger than Linux and OS X due to the lack of copy-on-write when forking a new Python process.

Switching between two windows is also within 100ms on all platforms (Table 3). Almost all of the time is spent in hardware and the operating system. The latency measured using the Python timestamps is less than 1ms.

The high speed camera results show that the latency for starting a new process that shows a pyplot window is twice as fast on Windows than Linux or OS X, even with the higher process create overhead of Windows (Table 3). Most of this latency is in the Python code on all platforms. The remaining 90-135ms are overhead in the keyboard/USB hardware, operating system, window manager, and display hardware. Compared to native applications the Python microbenchmarks are 2-10 times slower (Table 3), with the difference being time spent in the Python visualization library. It may be possible to optimize the Python library code, but it is easier to hide this overhead by starting the processes in the background such that these can be switched to when needed.

Increasing the number of windows to open and switch to 28, allows the workload to be run on multiple CPU cores, and hence the latencies only increase 7-15 times (Table 4). Also, we note that both microbenchmarks peak at 100% CPU utilization, so additional cores may have further reduced the added latency. In addition the user will get some feedback since the windows are often gradually updated.

These results demonstrate that the cores can be very efficiently utilized by BSV to improve interactive performance (parallel efficiency is up to 0.94).

6.5 Memory Usage

An important factor that limits the number of processors that can be started on each platform is the per-process DRAM usage. If all processes do not fit in memory, the resulting swapping will severely deprecate interactive performance. In this section we use the *startProcs*, *bigMem*, and *openWin* microbenchmarks to start 100 processes and then measure the per process memory usage.

The per process memory footprint for startProcs and bigMem are small for Linux and OS X, but much higher for Windows (Table 5). Again the problem is caused by the lack of copy-on-write during process creation. For Windows this will especially limit the number of processes that can be run if a large data structure is shared. However, the memory usage is similar for the openWin benchmark since it does not have many shared data structures.

Table 5. Per process memory usage (in MB).

Platform	Start-100	BigMem-100	OpenWin-100
Windows	3.05	132.3	47.3
Linux	0.84	11.8	46.6
OS X	0.70	12.7	38.7

Table 6. Total application memory usage (in MB).

Platform	GeoOverlap	PyHidra	Mandelbrot
Windows	3309	417	100
Linux	1121	856	73
OS X	1573	471	130

The openWin results also show that we can open up to 21 windows per gigabyte of DRAM, excluding application specific data structures.

6.6 Applications

In this section we use the applications described in section 5 to measure the window switch latency, memory usage, and identify the limitations to plotter parallelism.

We set the initial parameters based on how many plotter windows we can have comfortably visible on a 2560x1600 pixel monitor. For GeoOverlap we can have 20 windows open in a 4x5 grid, without the screen becoming cluttered or loss of information in a window. PyHidra has very tall visualizations so we can only fit 5 windows in a 1x5 grid. In Mandelbrot the visualization does not have any information content, so it is hard to determine a sufficient size for a window. However, we can assume that for CPU intensive visualization the number of processor cores will limit the number of open windows. We therefore only show four 500x500 pixel Mandelbrot windows.

We would like to run 3 background plotters per visible plotter, so we start 20 plotter for PyHidra, However, the memory usage of GeoOverlap in Windows 7 limits the number of concurrent processes to 40, while the computational time of Mandelbrot limits the number of concurrent processes to 8. The resulting memory usage is larger on Windows due to the lack of copy-on-write for GeoOverlap, and larger on Linux since a 64-bit Python is used (Table 6).

We measured the time to switch all visible windows for each application (Table 7) and the time to start a background process for each hidden window (Table 8). For GeoOverlap all 20 windows can be switched in less than 1.5 seconds on all platforms, but it can take up to 48 seconds to start all background plotters in Windows. Again, it is due to the lack of copy-on-write which causes the started plotter process to re-initialize all off the coordinator's data structures by reading these from disk. On all plat-

Table 7. Response time for switching all visible windows. All are in ms. Standard deviation is in parenthesis.

Platform	GeoOverlap	PyHidra	Mandelbrot
Windows	1061 (37)	3097 (576)	1886 (1063)
Linux	1381 (67)	3420 (1216)	1887 (516)
OS X	1473 (259)	2371 (610)	1346 (299)

Table 8. Time to start all background process when switching all visible windows (one process is started per hidden window).

Platform	GeoOverlap	PyHidra	Mandelbrot
Windows	48sec	6.9sec	100ms
Linux	1.8sec	8.3sec	40ms
OS X	13.4sec	6.5sec	810ms

Table 9. Time to execute a filter and visualize function. All times are in ms. Standard deviation is in parenthesis.

Platform	Filter	Visualize
Windows	4108 (708)	694 (302)
Linux	6497 (1059)	907 (142)
OS X	4354 (525)	440 (391)

forms the switch is completed before many background processes are started so there is no competition for CPU.

A cost breakdown shows that the switch time overhead is mostly due IPC overhead for a message exchange between the coordinator and the plotter. This message is for the coordinator to get the size and location of the plotter window (the user may have moved it). The size and position are then inherited by the next shown visualization. This call is blocking and is done for one plotter at a time.

The time to switch 5 PyHidra windows is about three times longer than the time to switch 20 GeoOverlap windows. We believe that the higher latency is due to the visualization functions in PyHidra that update a canvas using thousands of draw operations. The canvas must be repainted when a window is made visible. Note that due to the smaller screen in the Mac mini the windows in OS X have a smaller canvas, which may cause the performance improvement over Linux and Windows.

PyHidra background process startup time is similar on all platforms since a plotter loads the data after begin created, hence there is no advantage of cloning the address space.

For Mandelbrot it takes almost the same time to switch among the four windows on both platforms, but the measured latencies have very high variation. Starting the four background processes is very fast, but the CPU utilization is not 100% before the switch is done. There are therefore no performance benefits by reserving a core for the switch by only displaying three windows at a time.

The above results demonstrate that BSV achieves our interactive latency goals. However, we also identified factors that limit the interactive performance and the number of processors that can be run on a machine. These differ among the applications and also for platforms. For *GeoOverlap* in Windows, the main limitation is the startup time for background plotters, and the large memory footprint, which limits the number of background processes that can be run per visible plotter. This will limit the rate at which the user can browse through the windows. Interactive performance in *PyHidra* is limited by the time to switch among windows. We have identified the problem to be due to IPC overhead causing serialization in BSV. Also, PyHidra would benefit from a higher resolution display. *Mandelbrot* is a benchmark designed to be CPU intensive and it is therefore limited by the compute resources in a computer.

We also note that the applications have much more similar cross-platforms performance than the microbenchmarks.

6.7 Filter and Visualize Function Latency

The time to execute filter and visualization functions is also important for the interactive performance of BSV. We measure these latencies using the PyHidra application (Table 9).

First we measured the time to execute a filter function on all of the 20 running plotter processes. The filter function requires the coordinator to wait for a response from each plotter, so it takes between 4-6 seconds to execute. Most of this is due to overhead in the coordinator-plotter inter process communication (IPC). However, by ensuring that the filter function is executed on the visible plotters first most of the latency can be hidden. Also, a plotter for which the plotter returns a false value can be hidden immediately. Therefore, the user may see the first window being hidden after a few tens of milliseconds.

The latency of a visualization function is much smaller since the coordinator does not need to receive any messages from the plotters, and hence it takes less than a second. Again most of the time is spent in IPC.

6.8 Discussion

We found the response time of BSV window management commands, visualization function, and filter functions, to be up to a few seconds. Although we believe these are low enough to make BSV useful there is still room for improvement.

Some of the application specific limitation can easily be fixed by using a bigger computer. For GeoOverlap more memory would allowed to run more process concurrently in Windows. An additional high resolution screen would allow more windows to be shown for PyHidra, and more cores would allow running more Mandelbrot processes in parallel. BSV system code also introduces serialization that unnecessary limits interactive performance.

However, we also found some limitation with regards to interactive performance mostly in the Python visualization function code. For example opening a native Linux window was 10x faster than a Python visualization. In addition, we observed differences in Window Manager behaviour, where Windows seemed to batch update events, while Linux and OS X updated the windows continuously.

Our microbenchmark results showed significant differences between Linux and Windows. However, for the application benchmarks the differences were much smaller. We found it in general hard to reason, and especially predict, the interactive performance of visual applications run on the different platforms.

Our results also show that although the Python visualization libraries we used were restricted to a single threaded event based programming model, BSV was able to utilize all CPU cores on the test systems by running each visualization in a separate process.

We have not evaluated switch prediction algorithms since we do not know what realistic switch patterns will be. We assume that they will be mostly left-to-right traversal combined with filtering. An unanswered question is how many background plotter processes are needed for these traversal patterns.

7. RELATED WORK

Related work can be divided into visualization libraries and tools, parallel programming libraries and languages, and interactive performance evaluation studies.

We discussed limitations for commercial visualization tools, parallel visualization tools, and domain specific visualization tools in the introduction. The BSV system is for the Python scripting language. There are many alternatives to Python such as Python, Perl, Ruby, R, MATLAB, PHP, or JavaScript. In addition there are domain specific programming languages for visualization such as Processing [28] (including the high resolution tiled-display wall extension implemented in [29]) . Since BSV is based on multiprocessing, we believe a similar approach can be used to improve the screen resolution and processor utilization of these languages.

BSV was inspired and has its name from the Bulk Synchronous Programming (BSP) parallel programming model [15]. The general purpose BSP libraries [30][31][33][34] do not provide visualization libraries (nor do they seem to be actively maintained).

Alternative approaches for parallelization includes multithreading, a compiler that supports automatics parallelization, OpenMP directives in the code, MPI [34], MapReduce [35], or Pregel [36] (another BSP inspired special purpose systems). Most of these are intended for long running computationally or data

intensive computations, and are therefore not designed to provide low latency operations required for interactive visualization tools.

BSV show prediction can be considered as a special case of software based speculative parallelization as done in for example the BOP system [37]. BOP uses the copy-on-write for data protection, while BSV use it to reduce DRAM usage. BSV data sharing conflicts therefore only affect resource usage and not the correctness of the program.

Previous work on interactive system level performance analysis includes [38–41][42]. These typically focus on the variation of application response time, and are typically measured using software timestamps that miss the hardware and operating system overhead. Our high speed camera measurements include hardware and operating system delays. Similar high speed camera measurements are described in a forum post [43]. We are not aware of other recent performance evaluations of window manager, and scripting language visualization applications.

An alternative to software and system focused performance evaluation are user studies [44]. User studies could answer important questions not investigated in this paper such as the upper bounds for the number of visualization a human can interpret in a reasonable amount of time, and the perceived performance of big changes on high resolution screens such as switching tens of windows.

It is a commonly accepted that desktop systems have low CPU utilization. This has been confirmed by studies on thread-level parallelism in desktop applications [45][46], and there have recently been some suggestion about how to improve parallelism in interactive applications [47][48]. Although a multi-processing system can improve interactive performance, the application structure [47] often limits the number of cores that can be utilized [45][46]. Our experience implementing the BSV system confirms this, since we found the multi-threading support of the Python visualization libraries to be limited, and in addition differed among the three operating systems we used.

8. CONCLUSION

The contribution of this paper is a model for parallel visualization, and a description of the models design, implementation, and interactive performance and parallelism.

We have demonstrated that the model is useful and usable by implementing three explorative visualization applications that can be run on the three common workstation platforms Windows, Linux, and OS X. Based on our experiences developing and using these applications we have developed an API with the functionality required to programmatically control an interactive visualization with thousands of windows. Our experimental evaluation demonstrated that the model can utilize both high resolution displays and multi-core processors. We also identified interactive performance limitations, and parallelism limitations.

The source code is freely available, is well documented, has test cases, and example applications. It can therefore be further developed and extended. All are available at: www.cs.uit.no/~larsab/bsv/

9. FUTURE WORK

Two main areas for improvements to the BSV system are better visible window show prediction and easier window management. The algorithm currently used for show prediction assumes the user traverser the windows in 1-dimension. This works well if there is a natural sorting of the tasks. However, there may be cases when it is more natural to navigate through the windows in two (or more dimensions). For such navigation the prediction algorithm should take these additional dimensions into account. Implementing a BSV application that uses scatterplots to visualize multi-dimensional data would be a good case study.

Currently, all interaction with BSV is through function calls. However, it is easy to write a small GUI for the window management functions, or to integrate BSV with for example a touch/ gesture display wall interface [49].

GPU based visualization systems are capable of driving a high resolution display, and current many-core GPUs have the processing power required for many visualization applications. However, we believe the development time is too high with the current Python bindings. But it may be possible to implement very efficient sub-figure management using GPUs.

Another area of interesting future work is additional interactive performance evaluation of visualization applications that run on multiple platforms. Especially to understand the performance issues and dependencies of visualization libraries, window managers, and hardware.

We also intend to further investigate how the programmatic control over visualization can be used in interactive exploration of large collections of genomics data, and we plan to do a performance evaluation of BSV on a high resolution display wall to measure the overhead added by two levels of coordinators.

10. ACKNOWLEDGMENTS

I would like to thank Lars Tiede, Edvard Pedersen, and the anonymous reviewers for their comments to this paper.

11. REFERENCES

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

[2] B. Fjukstad, O. Anshus, and J. M. Bjørndalen, High resolution numerical models on a Display Wall, in *The 7th Annual Meeting of the European Meteorological Society (EMS) and the 8th European Conference on Applications of Meteorology*, 2007.

[3] G. Wallace, O. J. Anshus, D. Clark, P. Cook, A. Finkelstein, T. Funkhouser, A. Gupta, M. Hibbs, R. Samanta, R. Sukthankar, and O. Troyanskaya, Tools and Applications for Large-Scale Display Walls, *IEEE Computer Graphics and Applications*, vol. 25, no. 4, pp. 24–33, Jul. 2005.

[4] A. Singh, L. Bradel, A. Endert, R. Kincaid, C. Andrews, and C. North, Supporting the cyber analytic process using visual history on large displays, in *Proceedings of the 8th International Symposium on Visualization for Cyber Security - VizSec '11*, 2011, pp. 1–8.

[5] 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, and J. Zheng, Building and using a scalable display wall system, *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 29–37, 2000.

[6] DirectX Graphics and Gaming (Windows), http://msdn.microsoft.com/en-us/library/windows/desktop/ee663274.

[7] OpenGL, http://www.opengl.org/.

[8] VTK - The Visualization Toolkit, http://www.vtk.org/.

[9] L. Zhang, A. Stoffel, M. Behrisch, and S. Mittelstädt, Visual Analytics for the Big Data Era–A Comparative Review of State-of-the-Art Commercial Systems, in *Proceedings of IEEE Symposium on Visual Analytics Science and Technology*, 2012, pp. 173–182.

[10] N. Gehlenborg, S. I. O'Donoghue, N. S. Baliga, A. Goesmann, M. A. Hibbs, H. Kitano, O. Kohlbacher, H. Neuweger, R. Schneider, D. Tenenbaum, and A.-C. Gavin, Visualization of omics data for systems biology., *Nature methods*, vol. 7, no. 3 Suppl, pp. S56–68, Mar. 2010.
[11] VisIt Visualization Tool. https://wci.llnl.gov/codes/visit/.
[12] ParaView - Open Source Scientific Visualization, http://www.paraview.org/.
[13] PyLab, http://www.scipy.org/PyLab.
[14] B. Shneiderman, The eyes have it: a task by data type taxonomy for information visualizations, in *Proceedings 1996 IEEE Symposium on Visual Languages*, 1996, pp. 336–343.
[15] L. G. Valiant, A bridging model for parallel computation, *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.
[16] IPython, http://ipython.org/.
[17] pyplot — Matplotlib documentation, http://matplotlib.org/api/pyplot_api.html..
[18] wxPython, http://www.wxpython.org/.
[19] TkInter, http://wiki.python.org/moin/TkInter.
[20] pyglet, http://www.pyglet.org/.
[21] pygame - python game development, 2012. http://www.pygame.org/
[22] T. Barrett, D. B. Troup, S. E. Wilhite, P. Ledoux, C. Evangelista, I. F. Kim, M. Tomashevsky, K. A. Marshall, K. H. Phillippy, P. M. Sherman, R. N. Muertter, M. Holko, O. Ayanbule, A. Yefanov, and A. Soboleva, NCBI GEO: archive for functional genomics data sets--10 years on., *Nucleic acids research*, vol. 39, no. Database issue, pp. D1005–10, Nov. 2010.
[23] NetworkX 1.7 documentation, http://networkx.lanl.gov/.
[24] E. R. Gansner and S. C. North, An open graph visualization system and its applications to software engineering, *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, Sep. 2000.
[25] M. A. Hibbs, D. C. Hess, C. L. Myers, C. Huttenhower, K. Li, and O. G. Troyanskaya, Exploring the functional landscape of gene expression: directed search of large microarray compendia., *Bioinformatics (Oxford, England)*, vol. 23, no. 20, pp. 2692–9, Oct. 2007.
[26] R. B. Miller, Response time in man-computer conversational transactions, in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS '68 (Fall, part I)*, 1968, p. 267.
[27] B. A. Myers, The importance of percent-done progress indicators for computer-human interfaces, in *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '85*, 1985, vol. 16, no. 4, pp. 11–17.
[28] C. Reas and B. Fry, *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2007, p. 736.
[29] B. Westing, MostPixelsEverCE, https://github.com/bmwesting/MostPixelsEverCE.
[30] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, The Paderborn University BSP (PUB) library, *Parallel Computing*, vol. 29, no. 2, pp. 187–207, Feb. 2003.
[31] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, BSPlib: The BSP programming library, *Parallel Computing*, vol. 24, no. 14, pp. 1947–1980, Dec. 1998.
[32] M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas, Portable and efficient parallel computing using the BSP model, *IEEE Transactions on Computers*, vol. 48, no. 7, pp. 670–689, Jul. 1999.
[33] R. Miller, A Library for Bulk-Synchronous Parallel Programming, in *Proc. British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, 1993.
[34] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. Version 3.0, 2012.
[35] J. Dean and S. Ghemawat, MapReduce: a flexible data processing tool, *Communications of the ACM*, vol. 53, no. 1, p. 72, Jan. 2010.
[36] G. Malewicz, M. H. Austern, A. J. . Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, Pregel: a system for large-scale graph processing, in *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, 2010, p. 135.
[37] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, Software behavior oriented parallelization, *ACM SIGPLAN Notices*, vol. 42, no. 6, p. 223, Jun. 2007.
[38] B. K. Schmidt, M. S. Lam, and J. D. Northcutt, The interactive performance of SLIM, in *Proceedings of the seventeenth ACM symposium on Operating systems principles - SOSP '99*, 1999, vol. 33, no. 5, pp. 32–47.
[39] J. Nieh, S. J. Yang, and N. Novik, Measuring thin-client performance using slow-motion benchmarking, *ACM Transactions on Computer Systems*, vol. 21, no. 1, pp. 87–115, Feb. 2003.
[40] M. Jovic, A. Adamoli, and M. Hauswirth, Catch me if you can, in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications - OOPSLA '11*, 2011, vol. 46, no. 10, p. 155.
[41] N. Zeldovich and R. Chandra, Interactive performance measurement with VNCplay, in *ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005, p. 54.
[42] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer, Using latency to evaluate interactive system performance, *ACM SIGOPS Operating Systems Review*, vol. 30, no. SI, pp. 185–199, Oct. 1996.
[43] J. Carmack, Transatlantic ping faster than sending a pixel to the screen?, http://superuser.com/questions/419070/transatlantic-ping-faster-than-sending-a-pixel-to-the-screen.
[44] P. A. Dinda, G. Memik, R. P. Dick, B. Lin, A. Mallik, A. Gupta, and S. Rossoff, The user in experimental computer systems research, in *Proceedings of the 2007 workshop on Experimental computer science - ExpCS '07*, 2007, p. 10–es.
[45] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, Thread-level parallelism and interactive performance of desktop applications, *ACM SIGOPS Operating Systems Review*, vol. 34, no. 5, pp. 129–138, Dec. 2000.
[46] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, Evolution of thread-level parallelism in desktop applications, *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, p. 302, Jun. 2010.
[47] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser, Using threads in interactive systems, in *Proceedings of the fourteenth ACM symposium on Operating systems principles - SOSP '93*, 1993, vol. 27, no. 5, pp. 94–105.
[48] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík, Parallelizing the web browser, p. 7, Mar. 2009.
[49] D. Stødle, T.-M. S. Hagen, J. M. Bjørndalen, and O. J. Anshus, Gesture Based, Touch Free Multi User Gaming on WallSized, High Resolution Tiled Displays, *Journal of Virtual Reality and Broadcasting*, vol. 5, 2008.