Support for Collaboration, Visualization and Monitoring of Parallel Applications Using Shared Windows

Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus

Dept. of Computer Science, University of Tromsø, NO-9037 Tromsø, Norway
{daniels, jmb, otto}@cs.uit.no

Abstract. Results produced by a parallel application are typically collected and visualized on one display accessible to a single user. Collaboration between several researchers is usually achieved by sharing entire desktops. We have developed a system that shares windows, both from parallel applications and from desktop applications, with other users or to a wall-sized, high resolution display. Parallel applications can create several shared windows for each thread or process, enabling runtime visualization and monitoring. To aid collaboration, we provide multiple cursors for use on a display wall, allowing several researchers to interact simultaneously with windows shared by parallel and desktop applications. We measure the system's performance, and show that using shared windows for runtime visualization of the Mandelbrot computation increases the application's execution time by approximately 1.4%, while performance for sharing desktop application windows is halved as the number of users is doubled.

Keywords: Display wall, shared windows, multiple cursors.

1 Introduction

Current systems for runtime visualization of results from cluster applications are limited in their support for collaboration [1], as they rely on the X Window System [2] for window management and display. Visualization is typically done by a single user on a single display, making collaboration difficult. To collaborate, researchers have to share their entire display, which is often more than is necessary or desired. Finally, there are no standard desktop environments for wall-sized, high-resolution, tiled displays (display walls) that offer multiple cursors [3].

We have developed a system that can share windows from a parallel application and from desktop applications, and which provides support for multiple cursors on a display wall. Windows can be shared with other users and to a display wall, while the system's support for multiple cursors enables researchers to interact with shared parallel and desktop application windows on a display wall.

Figure 1 shows a user looking at a visualization of the Mandelbrot fractal, where each process of the parallel Mandelbrot computation draws into its own,

B. Kågström et al. (Eds.): PARA 2006, LNCS 4699, pp. 228–238, 2007.

shared window. On the display wall, the shared windows are displayed and placed next to each other to form a complete picture, whereas on the lower-resolution laptop display, there is only room for one window at a time.

Two scenarios further motivate the system presented in this paper. The first scenario concerns the use of shared windows as a means for runtime inspection of parallel applications. Each process of the parallel application creates a shared window, and uses it to visualize results or monitor the parallel application's performance. Figure 2 illustrates this scenario.

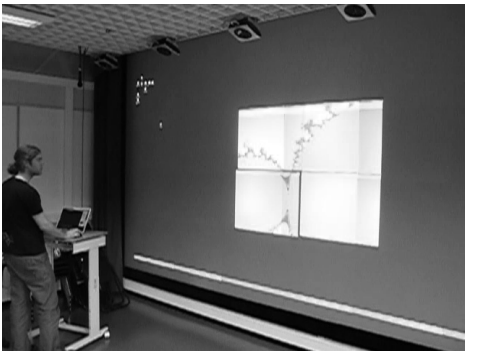

Fig. 1. A user looking at a visualization of the Mandelbrot fractal using shared windows on a display wall. The windows are placed next to each other, forming a complete picture.

In the second scenario, a group of researchers visualize a set of results on their desktop computers. To share data, they need to share desktop application windows with each other and a display wall. The other users can interact with the shared windows, modifying the shared view or change other settings as if the windows were local. On the display wall, several users can interact simultaneously using multiple cursors.

To meet the demands from these scenarios, our system should (i) support sharing of windows between different window systems and hardware platforms, and (ii) support the use of several cursors on a single, large desktop on a display wall.

We evaluate the performance of the parallel application window sharing subsystem by sharing windows containing the output from a parallel version of Mandelbrot, demonstrating that windows can be shared with less than 1.4% increase in the parallel application's execution time. This low impact on performance is due to a number of factors. First, the Mandelbrot application generates new content only about every five seconds, which means that the window sharing system only needs to provide updates to the shared windows at this rate. A higher rate of updates would likely increase the overhead from the window sharing system. Second, the benchmark was run without load balancing at either the application or system level, resulting in ample time for the window sharing system to run in on most nodes. Finally, since the window sharing system runs as a thread inside each process of the parallel application, it knows when the parallel application updates its shared windows and thus avoids sending unnecessary updates.

For the desktop application window sharing subsystem, the performance decreases by a factor of two when the number of window subscribers is doubled. This is caused by a combination of having to poll window contents in order to

discover updates, the request-based protocol between publisher and subscribers, and the publisher's implementation. The publisher batches requests, before a best-effort timer fires and makes the publisher process the requests, polling the window for updates at the same time.

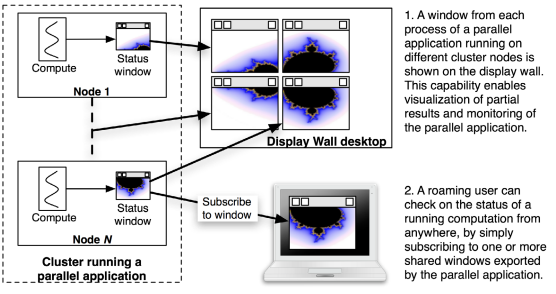

Fig. 2. Windows shared by a parallel application are accessed both for collaboration on a display wall and monitoring on a laptop

Parallel application window sharing performs better, since the window sharing system is integrated with the parallel application at the source code level. In contrast, sharing desktop application windows does not require modifications to the desktop application's source code, at the cost of lower performance.

Our main contribution with this paper is the integration of (i) shared windows as a means for runtime visualization of results and state directly from processes of a parallel application, (ii) the ability to share desktop application windows rather than a user's entire desktop, (iii) displaying shared windows from both parallel and desktop applications on a wall-sized, high-resolution tiled display, and (iv) support for multiple cursors on a display wall.

2 Related Work

VNC [4] and other remote desktop solutions [5,6] allows one to share an entire desktop. Although some VNC implementations can restrict the shared area to regions of the desktop, this does not amount to true window sharing, as any window brought within the shared region will be visible to others, whether intentional or not. SharedAppVNC [7] enables sharing of independent windows over VNC's Remote Framebuffer protocol on Mac OS X, Windows and Linux. The technique and code we developed for sharing windows on Mac OS X was shared with the developers of SharedAppVNC.

VNC is based on sharing the pixel representation of a remote display. We use the same approach in our window sharing system. In THINC [6], the authors demonstrate a solution that achieves better performance, in part due to their use of lower-level drawing operations to reduce communication. Their techniques are more complex to integrate into the window sharing system, as they rely on installing drivers into the X Server and intercept drawing operations to the framebuffer. The operations are then encoded and transmitted to clients. Due to the low level at which this is implemented, THINC has no concept of individual windows. In our opinion, this makes a window sharing implementation

utilizing ideas from THINC harder to realize. Other systems that use drawing operations to transfer display contents include the X Window System [2] and Microsoft Remote Desktop. We have chosen to share windows using their pixel representation, as parameters like coordinate systems, color spaces and line cap styles can be ignored.

Microsoft's Messenger and NetMeeting software [8] support application sharing under Microsoft Windows. Citrix' Presentation Server [9] supports application sharing across platforms. The drawback of application sharing is that it does not support sharing single windows. A shared application with multiple windows would make all those windows visible to other users, while window sharing would allow just a single window from the application to be shared. WinCuts [10] can support window sharing on Windows, but does not support interaction, and only updates windows once per second. For the X Window System, there are many application sharing solutions, including XTV [11] and Hewlett-Packard's commercial Shared X. Xmove [12] allows one to move applications between X servers, but does not support sharing the application with several users at the same time. MAST [13] is a tool that supports pixel-based application sharing for the Access Grid [14] on Microsoft Windows and Linux.

The MPI Parallel Environment, MPE [1], supports the creation of windows from each thread or process within a parallel application. Since MPE relies on the X Window System, the end-point for the visualization has to be fixed statically before starting the parallel application. That is, the X Server to use for display must be set prior to execution and can not be changed at runtime. Application sharing solutions like XTV or Xmove can alleviate this, but require that additional end-points also run the X Window System. Our window sharing system is more flexible, since the end-points are bound dynamically on-demand, and allows windows to be shared with computers running both Linux (X Window System) and Mac OS X. To visualize 3D data on display walls, software like Chromium [15] can be used. There is no concept of sharing visualizations in Chromium.

The first work on multiple cursors was Engelbart and English' paper from 1968 [16], where the mouse was introduced as an input device. One user had a controlling mouse, while the remaining users had mice that could only be used for pointing, and not interacting. Time-sharing the system cursor is used in [17], where a multi-cursor window manager similar to our own is presented. Their implementation adds a cursor ID to unused bits in the X event structure, which limits the number of cursors to seven. Multi-cursor events are then handled by removing the cursor ID and re-sending the cursor event to the X server as a regular system cursor event. Our implementation does not limit the number of cursors and does not require events to pass through the X server more than once. The Multi-Pointer X Server, MPX [18], integrates support at the hardware layer for several cursors, driven by mice connected to the computer running the X server. Presently, MPX only supports a single keyboard.

3 Model, Design and Implementation

In VNC, clients pull a single desktop from a VNC server. Our window sharing system is based on the publish-subscribe model. A publisher shares one or more windows through a publishing service. Subscribers access shared windows by connecting to the service, from which they can select the windows they are interested in and display them to the user. The service notifies subscribers when new windows are published, or old windows removed.

The service is realized using one or more servers, with publishers and subscribers acting as clients. The servers support network discovery using multicast, allowing clients to discover them on a LAN. Clients connect to servers over TCP for publishing or subscribing to shared windows. Subscribing clients are realized as separate processes, and receive updates to windows they subscribe to after requesting an update from the publishing client. For parallel applications, the publishing client is realized as a thread inside the parallel application. For desktop applications, the publishing client is realized as a separate process. Windows are shared using their pixel representation.

To illustrate how the window sharing system works in a parallel application, we added it to a parallel solver for the Mandelbrot fractal set. The solver on each node originally worked by displaying its part of the solution when all the nodes were done. In our modified version, the solver begins by creating a shared window. The shared window is maintained by a separate thread, and instead of displaying the solution when all nodes are done, the thread reads pixel data from memory, and sends an update to the shared window.

We have implemented window sharing for desktop applications on Mac OS X, allowing Mac OS X windows to be published to subscribers running on Mac OS X and Linux desktops, including the Linux-based display wall desktop. No changes to desktop applications are required in order to share their windows. On Mac OS X, each window is backed by a memory buffer that contains the most up-to-date window contents. Sharing such a window amounts to transmitting the contents of that buffer to subscribers. We use a polling approach on the buffer, as the OS does not notify the publisher when there are changes to other applications' windows. The user can configure the publisher to either send everything or detect changes in the buffer. The decision of which to use will impact the publisher's CPU and bandwidth usage. For static windows, change detection will reduce the bandwidth required for keeping subscribers updated, whereas for a window that is frequently updated, the bandwidth savings will be very small. Change detection is a very costly operation, as it requires calculating a diff between the last contents sent to subscribers, and the current version of the window. The OS X window sharing implementation is further detailed in [19].

The multiple cursor model is based on a service that handles cursor and keyboard input from a number of different users. Users push input events to the service, and the service is responsible for forwarding them from users to applications running on the desktop the service adds multi-cursor support to.

The service is realized as a server. The server runs in a thread, which in turn resides in the same process as a window manager for the X Window System.

Input from users is sent to the server via clients that run on the users' desktops. For each client, the server creates a cursor that is visible on the multi-cursor enabled desktop.

To emulate support for multiple cursors in the single-cursor X Window System environment, the system cursor is time-shared. For instance, when a user clicks his mouse button, the system cursor is moved to the position of that user's virtual cursor, and a mouse click event is posted. For applications, this creates the illusion that a single user is working on the desktop, when in reality there are several. For users, the illusion of several cursors supported by the window system is created. This approach is similar to the one taken in [17].

We implemented the design by incorporating the server thread in the Window Maker[1] window manager. Each client connects to the server over TCP, and is assigned a "virtual cursor." The virtual cursor maintains state associated with the client (such as current focus window and TCP socket information), and provides the actual cursor visible on the desktop to the user. The virtual cursor is drawn by creating an X Window, and modifying the window's appearance to match that of a cursor using the XShape extension. Different cursors are assigned different colors, and input events are posted using the XTestExtension.

4 Experiments

The hardware used for the experiments was (i) a 28-node cluster (Intel P4 EM64T, 3.2 GHz, 2GB RAM, hyperthreading enabled) running Rocks 3.3[2], (ii) a PowerMac Dual-G5 (2.5 GHz, 4GB RAM) running Mac OS X 10.4.2, (iii) a stand-alone PC (identical hardware configuration as the cluster nodes) running RedHat Enterprise Linux 4, (iv) a display wall (28 tiles, 1024x768 resolution per tile) with a combined resolution of 7168x3072, and (v) a GigaBit Ethernet. The cluster nodes are connected to a switch, and the remaining computers are connected to a second switch, with a single link joining the two switches. The cluster nodes also drive the individual tiles of the display wall.

We evaluated the impact of the window sharing system on parallel application performance by sharing windows from a parallel solver for the Mandelbrot fractal set. We measured the execution time of the parallel application both with and without window sharing running on the 28-node cluster. When window sharing was enabled, each node running the computation shared one window each, and each window had one subscriber. All the subscribers ran on the RedHat box. The experiment was repeated five times without window sharing, and five times with. The execution time for the Mandelbrot computation when running without window sharing was between 64.63 and 64.78 seconds, while the execution time when running with window sharing was between 65.40 and 66.19 seconds - an average increase of 1.38%.

We measured the performance of desktop application window sharing by sharing a window on the PowerMac G5 sized at 508x519 pixels in 32-bit color. The

[1] http://www.windowmaker.org/
[2] http://www.rocksclusters.org/

window contained an animation that updated at 30 frames per second (fps). The PowerMac G5 shared the window with subscribers running on the 28-node cluster. We conducted experiments varying the number of subscribers from 1 to 28, running each subscriber on a separate cluster node. We also conducted an experiment with 56 subscribers, where each node ran two subscribers. The Mac OS X publisher was configured to update the shared window at 30 fps without change detection, ideally reaching 30 fps at each subscriber. We measured the publisher's CPU load, the publisher's bandwidth usage and the number of frames received per second by each subscriber.

Figure 3 relates the publisher's CPU load with the publisher's bandwidth usage, with an increasing number of subscribers to the shared window. There is a clear correlation between CPU load and bandwidth usage, both steadily increasing until leveling out at about ten subscribers. At this level and beyond, the network is saturated, while the publisher still has available processing resources to handle additional subscribers.

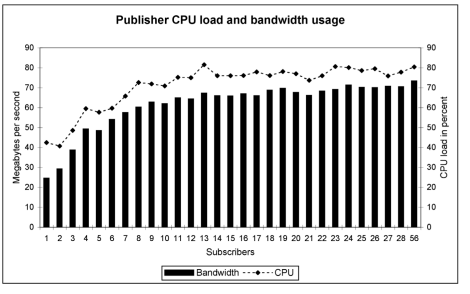

Fig. 3. The publisher's CPU load and bandwidth usage. For 1 to 28 subscribers, one subscriber runs on each node, while for 56 subscribers, two subscribers run on each node.

Figure 4 shows the average frame rate at the subscribers. With a single subscriber, about 28 fps is achieved, with 56 subscribers, the frame rate is 1.6. In general, doubling the number of subscribers cuts the frame rate approximately in half.

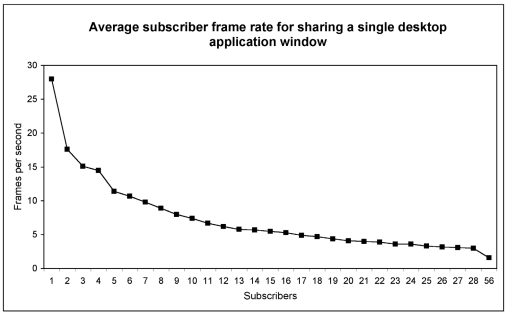

Fig. 4. Average frame rate as seen by each subscriber for sharing a single desktop application window

For the first few subscribers, Figure 3 shows that the publisher has available processing resources and network bandwidth but is unable to provide the subscribers with updates sufficiently fast to reach the target frame rate. There are two factors that contribute to this behaviour. First, subscribers must request updates from the publisher in order to receive them. If the subscribers do not do this sufficiently fast, the resulting frame rate will be lower. Second, the publisher accepts requests and processes them in batches. Each iteration is started

by a timer that fires in a best-effort manner. If an iteration exceeds the timer interval, the next iteration starts late, resulting in a lower frame rate.

We also conducted an experiment comparing the publisher's two update modes. The publisher can either send everything for each iteration, or calculate a diff between the current window contents, and the window contents most recently sent to subscribers (change detection). We measured the publisher's CPU load, bandwidth usage and frame rate at each subscriber.

Performing change detection was very costly. With one subscriber, the frame rate was 7.13, and the publisher's CPU load at 76.2%, transmitting 5.8 MB/s. With 20 subscribers, the frame rate was only 3.7, and publisher CPU load was at 100.1%[3] with bandwidth usage at 50 MB/s. In comparison, sending everything to a single subscriber gave a frame rate of 28, with publisher CPU load at 42.5% and bandwidth usage at 24.5 MB/s. To 20 subscribers, the publisher CPU load was 77%, transmitting 67.6 MB/s, and the frame rate was 4.1. We have performed informal practical experiments with the multi-cursor implementation, testing it with up to 8 simultaneous cursors.

5 Discussion

The experiments indicate that the impact on performance from adding shared windows to a parallel application is low. The benchmark was run without load balancing, neither on the system level nor the application level. For the Mandelbrot computation, this results in a very uneven load distribution, which on many nodes result in ample time for the window sharing system to execute in. This may contribute to hiding additional overhead from the window sharing system.

The system shares windows by sharing their pixel representation. This is the simplest way of sharing windows between hardware platforms and different window systems, and is the same approach as that taken by VNC [4]. The alternative to sharing pixels is to share drawing operations, like "fill rectangle" or "draw line." This approach is more challenging to make platform independent, compared to the simple operation of copying a block of pixels and sending them across the wire. As an example, drawing operations can save bandwidth by transmitting the raw text rather than the pixels making up the text on a display.

The publisher running on Mac OS X can use two different strategies when sending updates to windows. Since it doesn't know when or which regions of a window is updated, it can decide to either always send everything to everyone, or compute a diff between what the subscriber already has, and the current contents of the window. The trade-off is between publisher CPU load and publisher bandwidth. Subscribers will also potentially have fewer updates to draw, resulting in lower subscriber CPU load. Sending everything consumes more bandwidth, but incurs a lower CPU load on the computer running the publisher, while performing change detection is very costly. In contrast, for the Mandelbrot application, window contents are only sent when there are actual updates to the window. This

[3] The publisher ran on the PowerMac, which has two CPUs.

is possible since the window sharing code has been built directly into the Mandelbrot application, allowing it to send updates only when the shared window is actually updated.

Integrating the window sharing system with parallel applications requires that the source code for the parallel application is available. This is not a major problem though, since the window sharing system only simplifies the task of publishing the windows. The application itself is responsible for filling that window with meaningful content - be it a runtime visualization or performance monitoring data. This will require further modifications to the application's code.

6 Conclusion

This paper presents a system that shares windows created by parallel and desktop applications between two or more users. To further enhance collaboration, a system supporting multiple cursors on a wall-sized, high-resolution display is used to allow many users to manipulate shared windows simultaneously.

Parallel applications require modifications to their source code in order to share windows and discover updates in them. For desktop applications, modifying their source code is usually neither practical nor possible. Because of this, sharing desktop application windows is more costly, both in terms of CPU and network load, as the system has to poll window contents in order to discover updates.

We have integrated the system with a parallel implementation of a solver for the Mandelbrot fractal, and measured its impact on the application's execution time. We measured the performance of desktop application window sharing by sharing a single window containing an animation with a varying number of users. The windows were displayed on the tiled display wall. The multi-cursor system was used with eight cursors.

We found that the addition of shared windows to the Mandelbrot application only added about 1.4% to the application's execution time. For this benchmark, no load balancing was used, resulting in an uneven distribution of work between the different processes. This gives the window sharing system CPU time to execute in that would otherwise remain unused, which combined with the Mandelbrot application's infrequent updates, explains the window sharing system's low impact on the application's performance. A higher update frequency would likely increase the window sharing system's impact on execution time.

For desktop application windows, the number of updates received per second by each subscriber went from 28 with one subscriber, to 1.6 with 56 subscribers. In general, when the number of subscribers is doubled, the update frequency seen by each subscriber is halved. For less than ten subscribers, CPU and network are not the limiting resources. Instead, the scaling behaviour is caused by the iteration- and timer-based approach used by the publisher. With many subscribers, the limiting resource is the network.

Sharing windows and support for multiple cursors are promising for increasing the flexibility of runtime visualization and monitoring of parallel applications, and for collaboration using desktop applications. More work remains to

determine the window sharing system's impact on these issues, and to better characterize the system's performance.

Acknowledgements. Thanks to Lars Ailo Bongo, Vera Göbel, Espen Skjelnes Johnsen, Tore Larsen, Kai Li and Grant Wallace for their discussions, suggestions, help and support. This work has been supported by the NFR funded project No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices.

References

1. Chan, A., Gropp, W., Lusk, E.: MPE: MPI Parallel Environment, `http://www-unix.mcs.anl.gov/perfvis/software/MPE/index.htm`
2. Scheifler, R.W., Gettys, J.: The X Window System. ACM Trans. Graph. 5(2), 79–109 (1986)
3. Faith, R.E., Martin, K.E.: Xdmx: Distributed, multi-head X, `http://dmx.sourceforge.net/`
4. Richardson, T., Stafford-Fraser, Q., Wood, K.R., Hopper, A.: Virtual Network Computing. IEEE Internet Computing 2(1) (January 1998)
5. NoMachine. NX server and client, `http://www.nomachine.com/`
6. Baratto, R.A., Kim, L.N., Nieh, J.: Thinc: a virtual display architecture for thin-client computing. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, pp. 277–290. ACM Press, New York (2005)
7. Wallace, G.: SharedAppVNC, `http://shared-app-vnc.sourceforge.net/`
8. Microsoft Corporation: Netmeeting, `http://www.microsoft.com/windows/netmeeting/`
9. Citrix: Citrix Presentation Server, `http://www.citrix.com/`
10. Tan, D.S., Meyers, B., Czerwinski, M.: WinCuts: Manipulating arbitrary window regions for more effective use of screen space. In: CHI '04 extended abstracts on Human factors in computing systems, pp. 1525–1528. ACM Press, New York (2004)
11. Abdel-Wahab, H., Feit, M.: XTV: A Framework for Sharing X Window Clients in Remote Synchrounous Collaboration. IEEE Tricomm (April 1991)
12. Solomita, E., Kempf, J., Duchamp, D.: XMove: A Pseudoserver for X Window Movement. The X Resource 11(1), 143–170 (1994)
13. Lewis, G.J., Hasan, S.M., Alexandrov, V.N., Dove, M.T., Calleja, M.: Multicast application sharing tool - Facilitating the eMinerals virtual organisation. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2005. LNCS, vol. 3516, pp. 359–366. Springer, Heidelberg (2005)
14. The AccessGrid Project website, `http://www.accessgrid.org/`
15. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: A stream-processing framework for interactive rendering on clusters. In: SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, pp. 693–702. ACM Press, New York (2002)
16. Engelbart, D.C., English, W.K.: A research center for augmenting human intellect. In: AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference, pp. 395–410 (December 1968)

17. Wallace, G., Bi, P., Li, K., Anshus, O.: A MultiCursor X Window Manager Supporting Control Room Collaboration. Technical Report TR-707-04, Princeton University, Computer Science (July 2004)
18. Hutterer, P.: The Multi-Pointer X Server, MPX, `http://wearables.unisa.edu.au/mpx/`
19. Stødle, D., Bjørndalen, J.M., Anshus, O.J.: Collaborative sharing of windows between Mac OS X, the X Window System and Windows. In: Norsk informatikkonferanse 2004, NIK 04 (2004)