Software Environments For Cluster-based Display Systems

Yuqun Chen* Han Chen Douglas W. Clark Zhiyan Liu Grant Wallace Kai Li

Department of Computer Science, Princeton University, Princeton, NJ 08544

Abstract

An inexpensive way to construct a scalable display wall system is to use a cluster of PCs with commodity graphics accelerators to drive an array of projectors. A challenge is to bring off-the-shelf sequential applications to run on such a display wall efficiently without using expensive, high-performance interconnects.

This paper studies two execution models for a scalable display wall system: master-slave and synchronized execution models. We have designed and implemented four software tools, two for each execution model, including VDD (Virtual Display Driver), GLP (GL-DLL Replacement), SSE (System-level Synchronized Execution), and ASE (Application-level Synchronized Execution). In order to support the synchronized execution model, we have also designed a broadcast, speculative file cache to provide scalable I/O performance. The paper reports our experimental results with several 3D applications on the display wall to understand the performance implications and tradeoffs of these methods.

1 Introduction

A display is the most common device with which human beings visualize information inside a computer or a network. The scale and resolution of a display device define how much information a user can view at a time. While the cost-performance ratio for many key enabling technologies has been improving at or beyond the rate predicted by Moore's Law, display resolution – a key aspect of an effective information system – is lagging far behind. Monitors are still the dominant display technology through which people visualize information. Their resolutions have been increasing at a mere 5% annual rate for the last two decades. This widening gap is a major limitation on human-computer interaction.

One strategy for overcoming the resolution limitation is to tile multiple projection devices over a single display surface. Video products using this method are usually called video walls. A typical video wall uses a special piece of hardware, called a video processor, to scale lower-resolution video content, such as in NTSC, VGA, SVGA, and HDTV formats to fit a large display surface. It does not provide digital applications with higher intrinsic display resolution. To provide higher intrinsic resolution, some research prototypes and commercial products use a high-end graphics machine with multiple tightly coupled graphics pipelines to render images and drive the tiled display devices [12]. Such a system is very expensive, often costing millions of dollars.

The Scalable DisplayWall explores how to build and use a scalable display system for users to collaborate across space and time. We are particularly interested in building a scalable display system with low-cost, commodity components such as PCs, PC graphics accelerators, off-the-shelf network, consumer video and sound equipment, and portable presentation projectors. The advantages of this approach are low cost and technology tracking, as high-volume commodity components typically have better price/performance ratios and improve at faster rates than special-purpose hardware. In order to succeed, we must understand how to build a scalable display system that can render and drive hundreds of millions of pixels, support existing and new applications at interactive speed.

This paper focuses on the issue of developing software tools to bring off-the-shelf, sequential applications to a scalable display wall built with commodity components and run them efficiently at its intrinsic resolution. Many applications either play back multimedia content, for example, MPEG movies, HDTV streams, images, and VRML models, or allow users to interactively navigate content such as web pages and Macromedia Flash movies. The content usually conforms to publicly-documented standards. Therefore, it is feasible, though sometimes tedious, to write special playback software to parse and play back these kinds of content for a cluster environment.

Yet for many other applications, it becomes extremely difficult, if not impossible, to apply the play-back approach. One reason is that the interactive behaviors of these applications are often encoded in the software itself rather than in the raw data, as in a Macromedia movie. An example is a typical 3D game for personal computers. The scenes in the game are described using standard file formats. Manipulation of the scenes and interpretation of user inputs, on the other hand, is often a trade secret and hidden in millions of lines of machine instructions. Another obstacle is the sheer amount of work required to write the playback software that mimics the behavior of the original application. Consider Macromedia Photoshop, a high-quality image editing tool. Many of its functionalities and features are well known and described in textbooks and technical papers. However, to produce a new software package that has similar "look and feel" would require many months of work.

An important objective of the Scalable Display Wall Project is to design a a runtime environment for running and developing applications that we daily use on regular desktop computers, but with much higher intrinsic resolution. This paper focuses on the issue of developing software tools to bring off-the-shelf, sequential applications to a scalable display wall and run them efficiently at its intrinsic resolution.

Two models of program execution are studied: the *master-slave* model and the *synchronized program execution* model. With the master-slave model, a master node executes an application, intercepts all graphics outputs such as 2-D and 3-D primitives (polygons, lines, points, etc), and sends them to the nodes that drive the tiled

*Microsoft Research, Microsft Corporation, Redmond, WA 98052, {yuqunc}@microsoft.com

projectors for execution. With the synchronized execution model, an instance of the application runs on each of the nodes that drive the tiled projectors. Multiple instances are synchronized and coordinated so that they act as if they are a single application designed for the scalable-resolution display system. The synchronization (or coordination) can be done either at the application level via a programming API or by the runtime system transparently.

We have designed and implemented four software tools to support these two execution models. They are

- Virtual Display Driver (VDD), which supports 2-D Windows applications using the master-slave model,
- A Distributed GL tool that supports 3-D applications using the master-slave model,
- A runtime system that supports applications using the synchronized program execution model, and
- A system-level tool that supports applications using the synchronized program execution model.

In order to understand the performance implications and resource requirements of each method, we measured several 3-D applications with our software tools on the scalable display wall system. The results show that the master-slave model works reasonably well for applications that send few or no graphics primitives during each frame. The runtime support for the synchronized program execution model achieves similar or sometimes better performance than the master-slave model. The application-level synchronization tool, when source code modification is possible, can achieve much better performance and improve the application response time.

2 Previous Work

The common way to run digital applications on a video wall is to use the video content scaling hardware (video processor) to drive an application's output on a tiled display system. This approach does not allow applications to use the intrinsic resolution of the tiled displays.

Various kinds of tiled display systems have been constructed for data visualization during the past few years. Examples include the Power Wall at the University of Minnesota, the Infinite Wall at the University of Illinois at Chicago [7], the Office of the Future at UNC [15], the Information Mural at Stanford [9], and various immersive products from several vendors. In most cases, an SGI Onyx2 or equivalent high-end machine with multiple graphics pipelines is used to drive multiple projectors. Since these systems are not using a PC-cluster architecture, they do not address the issue of software support for running sequential, off-the-shelf applications on a scalable resolution display wall.

The idea of executing graphics primitives remotely can be found in X windows [16]. The tools such as VDD and GRL described in this paper leverage and extend these ideas for remote graphics primitive executions for scalable resolution displays. VDD intercepts primitives in a device driver and executes them remotely at the user level whereas GLR substitutes primitives in a local OpenGL library. The mechanisms for remote procedure call [13, 2] and remote execution model [17] have been investigated in the context of programming languages.

The parallel graphics interface designed at Stanford [11] proposes a parallel API that allows parallel traversal of an explicitly ordered scene via a set of predefined synchronization primitives. The goal is to expose parallelism while retaining many of the desirable features of serial programming. The parallel API was not designed to execute multiple instances of a sequential program on a scalable display system built with a PC cluster.

3 Existing Desktop Applications

Our first approach to bringing desktop applications onto the display wall is to run the application on a *master* node, intercept the graphics primitives that the application generates, and send them over the network to the render nodes (or the *slaves*). Figure 1 shows a conceptual diagram of this master-slave approach. The primitives can be broadcast to all render nodes or, as an optimization, be sent only to the nodes with whose screen tiles they overlap. In either case, a slave node performs view-frustum clipping to render those primitives within its screen tile.

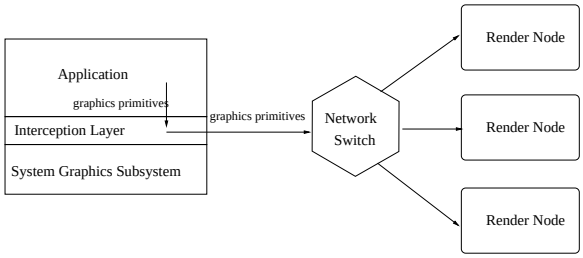


Figure 1: Conceptual view of the master-slave approach

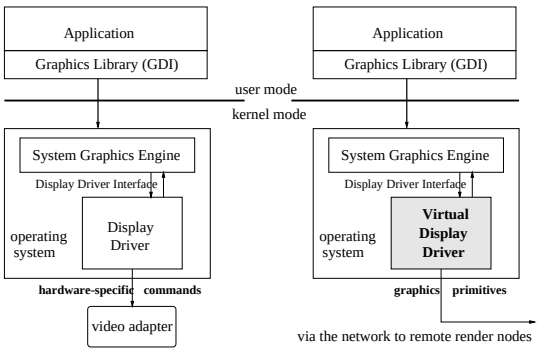
Clipping 2D primitives is straightforward. For 2D vector graphics, this means translation of the 2D coordinates in the global display space to the local coordinates on each render node. The translation is simply an addition of offsets in X and Y directions. The task is even simpler on the Windows platform. One can define a viewport transform to for the graphics device that does exactly this. Windows automatically applies the viewport transform to all subsequent drawings. For bitmaps, the render node selects a portion from the bitmap that falls within its screen tile. Once again, this can be trivially accomplished by specifying the appropriate rectangle to a `bitblt` operation.

To clip 3D primitives, the render node can simply specify a subvolume of the global viewing frustum as its viewing frustum. This sub-frustum can be trivially calculated using the screen tile's relative position in the global display space.

Ideally, primitive interception should be transparent to the applications, so that we can run any application binaries and have their graphics shown on the display wall without source modification or re-linking. This means that we inject the interception mechanism in the master node's graphics subsystem that sits beneath the running application. We found two ways to implement the transparent interception mechanism: via a *virtual display driver* and by replacing a dynamically-linked library (DLL).

3.1 Virtual display driver

In a typical PC operating system such as Microsoft Windows NT and Linux, the graphics subsystem is implemented by a display driver, which speaks a standard protocol to the application layer on one side, and translates the application's graphics commands into device-specific commands on the other side. The interface between the application layer and the display driver is standardized on a given operating system, so as to allow any graphics accelerator vendors to implement their own vendor-specific drivers. Therefore, we can implement a *virtual* display driver that acts as if it operated a real piece of graphics hardware, but actually takes the graphics commands (or primitives) from the application layer and sends



(1) A typical Display Driver architecture (2) The Virtual Display Driver architecture

Figure 2: Architectural diagrams of a typical display driver and a virtual display driver (VDD)

them over the network to the render nodes. Figure 2 illustrates the common block structure of a typical display driver and that of a virtual display driver.

An advantage of using such a virtual display driver is that it presents to the applications a large display with intrinsically high resolution. The applications and the operating system normally query the display driver to obtain its screen resolution and adjust the windows, menus, and drawing parameters accordingly. Unconstrained by the actual graphics hardware, the virtual display driver is free to fake a display with an arbitrarily large number of pixels, causing most applications to adapt their drawing resolutions to the high resolution. As a demonstration, we ran Microsoft PowerPoint on our 8-node Display Wall. In its "slide sorter view" mode, PowerPoint placed many slides on the display with clear definition for even the smallest fonts.

Due to the compact nature of the display driver protocol, implementing a virtual display driver is usually not a daunting engineering task. For example, on both the Windows NT 4.0 and Windows 2000 operating systems, the protocol between the 2D drawing layer in the application and the display driver, the *Display Driver Interface* (DDI), consists of no more than 25 commonly used graphics commands for drawing bitmaps, lines, polygons, etc. However, there are still several non-trivial issues in implementing a virtual display driver. The first issue is to translate the graphics objects that are meaningful in the master node's kernel environment to those that can be used by the render nodes. An example of this translation is the drawing surface object that is passed as an argument in most DDI commands to the virtual display driver. The surface object can either identify a bitmap stored internally in the kernel or the drawing surface.

The second issue is performing *bitblt* across the render nodes. Unlike the bitblt operations on a single PC, the bitblt operations among render nodes transfer pixels across the network. Similar to the operations on a single PC, the order of operations is important. Another method is to refresh from the master node, which is inefficient.

The third issue is how to distribute the graphics primitives efficiently among the render nodes. This issue is common to both virtual display and DLL replacement approaches. We postpone its discussion until Section 3.3

One can use the virtual display driver to implement most drawing protocols on the Windows operating system: DDI and DirectDraw for 2D drawing, and Direct3D for 3D rendering. Similarly, we can also implement the X Windows Protocol [16] The X Windows server already embodies the concept of a virtual and networked display. In this case, the virtual X Windows server is situated on the master node, and speaks the X protocol to local applications. It takes and translates the application's X Windows commands and forwards them to the render nodes.

As a final note, the applications running on the master node still receive user inputs through conventional channels such as from a real keyboard and a real mouse. We are experimenting with virtualizing the user inputs as well, so that user interactions with the applications (and hence with the display wall) are no longer restricted to the keyboard and the mouse attached to the master node.

3.2 DLL replacement

On modern operating systems, many services no longer reside inside the OS kernel. Instead, they are provided as dynamically-linked libraries (DLL) and are linked into the applications by the OS runtime. An example of this is the `opengl32.dll` on Windows NT/2000 operating systems. This dynamically-linked library contains a default implementation for OpenGL renderings. OpenGL is an industrial standard for high-performance 3D rendering. It is based upon a stateful client-server interaction model. Major OpenGL commands include those that change and retrieve the states on the server side, for example, setting the model transformation matrix, and those that specify the actual rendering primitives, for example, 3D vertex and color specifications. The default OpenGL implementation in `opengl32.dll` performs most rendering stages on the CPU and then calls Windows' 2D drawing API, GDI, to carry out the final rasterization, i.e., converting a 2D line or 2D polygon to actual pixels on the screen. If the graphics accelerator vendor implements OpenGL in hardware, a vendor-supplied OpenGL DLL is also linked into the application; all OpenGL calls made by the application are forwarded to the vendor-supplied DLL by `opengl32.dll`.

To intercept the OpenGL commands from an application, we can either write a replacement DLL that has a compatible interface with the native `opengl32.dll` or one that is compliant with the OpenGL vendor DLL interface. This latter method is less of a hack than former, and can be well integrated with the virtual display driver.

An advantage of using DLLs to intercept graphics primitives is that communication between the master node and the render nodes occurs at the user level. Not only is it easier to debug this approach, it can also leverage several efficient user-level communication implementations, including the one developed by us [6, 18, 19].

The drawback of this approach is that the API exported by the DLL is typically large, and highly complex in some cases. This is why writing a virtual display driver for 2D Windows applications is a less painful job than writing a corresponding replacement DLL. One exception is the OpenGL API. It is a fairly regular and well thought-out interface. Even though there are between 200 and 300 calls in the API, we were quite successful at automatically generating the bulk of the replacement DLL using a simple parser.

3.3 Discussion

A remote display protocol is a general solution. It does, however, have a potential problem in efficient distribution of graphics primitives. An obvious solution is to broadcast the graphics primitives to all render nodes. System-area networks today typically implement point-to-point communication via a switch. Few networks except Ethernet implement broadcast or multicast in hardware. We currently employ a ring topology to broadcast graphics primitives in 64 KB chunks over Myrinet. This method may incur high latency as the number of projectors scales up. Another common broadcast topology is a binary balanced tree. Although its latency is

$O(log(N))$, this topology may suffer from poor bandwidth because each network endpoint has to send two copies of each packet out to its children. Efficient broadcast over a point-to-point network still remains an open problem.

Aside from lacking an efficient broadcast mechanism, the primitive distribution speed in the master-slave approach is essentially constrained by the speed of the outgoing network link on the master node. The network interface is typically connected to an I/O bus. Its bandwidth is often an order of magnitude smaller than that of the internal memory bus. For immediate-mode applications that generate 3D primitives for each frame, this means they may run slower on the display wall than on a single machine. The slow-down factor may not be an order of magnitude, because the primitive rendering task is now partitioned among many render nodes according to the screen tiles. But when these immediate-mode applications, typically 3D games, are written for a balanced system where the graphics accelerator's performance matches the bandwidth of the local graphics/memory bus, the relatively slow network link will definitely hamper the rendering rates of the display wall system.

Many retained-mode applications, however, will not suffer from the network bottleneck. These applications largely deal with static scenes whose rendering can be compiled into OpenGL display lists or the like. The master node only pays an up-front cost to send the rendering commands for each display list to the render nodes. It then can simply issue rendering commands by referencing the display lists. As a result, very little data, other than changes in the viewing position and the movements of the objects, must be sent over the network.

4 Synchronized Program Execution

The basic idea in synchronized program execution is to run multiple instances of a program on the display wall render nodes. The execution of these program instances are synchronized at some level, with respect to a synchronization boundary, so that within this boundary these instances assume identical behaviors. As a result of synchronization, the program instances generate identical scene descriptions, which can simply be identical OpenGL 3D primitives across all render nodes or some higher-level scene description that each node instantiates in a tile-specific fashion.

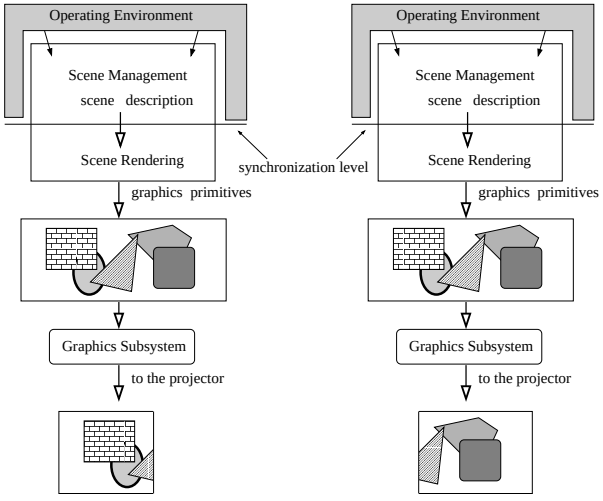


Figure 3: Full replication of multiple program instances

In the first scenario, as depicted in Figure 3, the graphics accelerator performs tile-specific culling and only renders those primitives that fall within its respective screen tile. This can be enabled by setting the appropriate view frustum matrix for each render node [8].

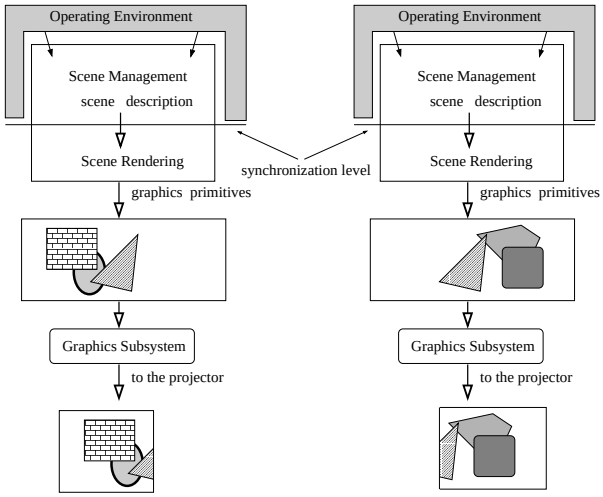


Figure 4: Tile-specific primitive generation in application-level synchronization

In the second scenario, as illustrated in Figure 4, the higher-level scene description can be fed into the view-dependent software layer that generates tile-specific primitives. An example of the second scenario is a scene graph render program that organizes the scene data in a hierarchy of objects. Given a tile-specific view frustum, the program can remove the objects that fall completely outside the frustum.

To better illustrate our idea, we partition a program conceptually into two components: scene management and scene rendering. The scene management component interacts with the program's operating environment, for example, reading files and getting the keyboard and the mouse inputs, and changes its internal behaviors accordingly. In other words, the scene management *responds* to the events in the environment; its behaviors are completely determined by its interaction with the environment. The scene rendering component takes from the scene management layer the scene description, from which it generates the graphics primitives. The picture is even clearer if we take a pipeline view of a program, as illustrated in Figure 5. The upper stage of the pipeline, the scene management, is synchronized across all render nodes. Beneath the synchronization level, the scene rendering layer is free to perform any tile-specific tasks, provided its actions *do not alter* the behavior of the scene management layer.

4.1 Synchronization infrastructure

Our program synchronization framework consists of a thin synchronization layer on each render node and a coordinator. The synchronization layer intercepts certain function calls made by the scene management component of the program to interact with the environment. For each intercepted function call, the results from one render node are picked by the coordinator and sent to the other nodes.

Each synchronization of a function call acts like a barrier synchronization. Since the result from only one node is sent back to other nodes, one can use an efficient broadcast topology such as a

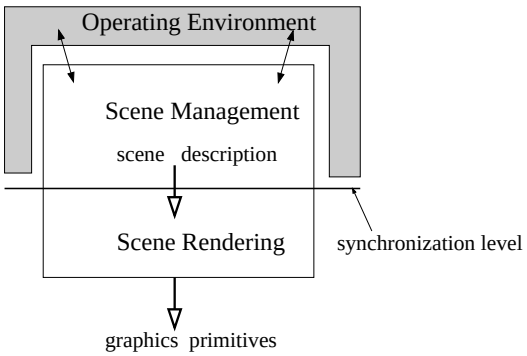


Figure 5: The pipeline view of the two program components

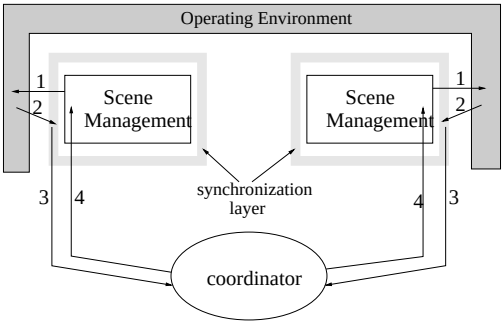


Figure 6: The synchronization framework

multi-cast tree to implement a call synchronization. Furthermore, only those function calls that can potentially alter the program behaviors need to be synchronized. They typically include calls to query keyboard and mouse inputs, calls to read the system timers, and file I/O operations. Most of these calls have results that are small in size. Hence synchronizing these calls require little communication bandwidth, though low communication latency is vital to keeping the synchronization overhead down. Large file reads can be synchronized with an efficient multicast mechanism such as the UDP-based broadcast mechanism that we describe later in the paper.

The actual placement and implementation of the synchronization layer depends on whether we employ system-level or program-level synchronization.

4.2 System-level program synchronization (SSE)

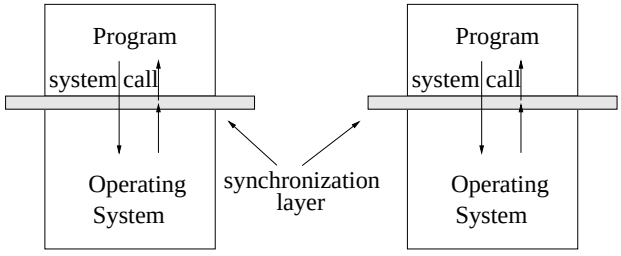


Figure 7: Program synchronization at the system-call level

The goal of system-level program synchronization (SSE) is to replicate a program on multiple nodes in a transparent fashion, i.e., without modifying and re-linking the program. We also require that SSE incur very low overhead, even as the number of nodes in the system scales. System-level program replication has been studied in the context of fault-tolerant computing. In Hypervisor, Bressoud et al proposed a method that treats an actual software system as running on a virtual machine [4, 5]. In their framework, two or more such systems can be replicated by synchronizing the runtime semantics of the virtual machine I/O instructions. However, the virtual machine defined in Hypervisor is too close to the actual microprocessor architecture. The fine-grain synchronization causes excessive overhead and slows down the program by as much as a factor of 2.

In order to avoid the high overhead incurred by the Hypervisor approach, we define a virtual machine as the operating system. The macro instruction set of this virtual machine is the system-call API defined on the OS, as shown in Figure 7. For a single-threaded program, one can prove that synchronizing at the system call level leads to synchronized program execution. The reason is that code execution is deterministic from the microprocessor architecture's point of view. A single-threaded program's execution path is only influenced by external events. The way these external events affect the program behavior is through the system-call interface and signal handling on UNIX. It follows that if we make the interaction between the program and the OS environment identical on all nodes, the program instances will all follow the same execution path and exhibit the same behavior, and as a result, produce the same graphics primitives.

We have made a few simplifying assumptions in our SSE approach. First, we ignore those programs that interact with the rest of the system via shared-memory segments, because in such cases we have to intercept all reads and writes to the shared-memory segments in order to achieve exact program replication. Second, we assume most programs access the CPU cycle counter via a well-defined API, such as the `QueryPerformanceCounter()` on the Windows platform, so that we can intercept these accesses as well. For programs that use assembly instructions to read the cycle counter, as allowed by the Intel architecture, we could certainly edit the program binary and replace offending instructions with calls to a special handler. In practice, however, this need has not arisen.

Our SSE mechanism does have one limitation: it is not guaranteed to work with arbitrary multi-threaded applications. The problem lies in the fact that it is hard to ensure identical interleaving of threads among multiple program instances. This problem becomes even harder for multi-processor nodes on which several program threads can be running at the same time, because in this case the threads can interact with each other via shared physical memory. To capture such interactions we would have to intercept the load and store instructions, basically going back to the Hypervisor approach.

Among numerous system calls a program makes, only a handful of them can alter program states. They include the calls to query Window messages and the system timer. Therefore, our system-level synchronization layer only performs synchronization for these selected system calls. Our system call synchronization technique works primarily for programs with a single thread. In theory, it should also work for those multi-threaded programs, in which a single thread interacts with the environment while the other threads simply compute the scenes without affecting the internal states of the program.

4.3 Application-level program synchronization (APE)

Program replication at the system call level can transparently synchronize multiple program instances. However, it is not guaranteed to work for multi-threaded programs. In addition, since the system-

level approach treats the entire program as a whole, it cannot separate the program into scene management and scene rendering components and let the latter perform tile-specific primitive generation and rendering. We can solve these two problems by moving the synchronization boundary into the application itself.

The same mechanism for system-level program synchronization can be used to synchronize program instances at the application level. Instead of system calls, we synchronize function calls within the application. We currently provide a simple API call, `SynchronizeResult()` to let all render nodes get consistent results. Figure 8 illustrates the semantics of a function call synchronization. One can also imagine building a sophisticated interface description language (IDL) that automates the synchronization of user functions.

synchronized fun $F(a_1, a_2, ...) \Longrightarrow$ result :
$tmp \Longleftarrow F(a_1, a_2, ...)$
$SynchronizeResult(tmp)$
$result \Longleftarrow tmp$

Figure 8: The semantics of a function call synchronization

To facilitate calculation of the tile-specific view frustum, we also provide two function calls, `QueryDisplayInfo()` and `GetLocalViewFrustum()`. The first call returns a render node's screen position and its node ID in the tuple `(GlobalScreenRectangle, MyScreenRectangle, MyNodeId)`. The second call takes a global view frustum, as defined in a single-node version of the program, and calculates the local view frustum based on the node's tile position in the display wall.

An application can take advantage of application-level synchronization by simply performing high-level object culling based on its local view frustum. Given N projectors in a display wall, each screen tile has only $1/N$ of the the global frustum. Generally this results in a small fraction of objects for each render node to render. Instead of generating all the primitives and letting the graphics accelerator throw out primitives that fall outside the local frustum, the scene rendering component of each program instance can reject a group of graphics primitives or avoid generating them, by comparing their bounding box with its view frustum. Such high-level culling needs much less computation time and consumes far less local bus bandwidth. And it is generally easy to implement. Note that this kind of high-level culling is not possible in the master-slave approach, because the master has to generate primitives for the entire global view frustum.

With more programming effort, one can further reduce the amount of computation required to generate graphics primitives. Many data visualization applications that deal with large data sets belong to this category. One example is an isosurface extraction program developed in our department. An isosurface of a 3D scalar field is made of points, the isopoints, that have the scalar value of a given constant. Extracting isosurfaces is a very useful technique to understand complex volume data that are generated by many medical applications like CT and scientific computations such as astrophysics simulations. These applications often generate scalar fields that are sampled at discrete points in 3D space. The isosurface in these cases is a smooth interpolation of discrete isopoints. Traditional isosurface extraction algorithms extract the whole isosurface. However, the size and complexity of datasets is constantly growing. Due to the high depth complexity and the large dimensions of the dataset, typically only part of the isosurface is visible. *Isoview* is an isosurface extraction program that tries to extract only the visible portion of the isosurface. The algorithm casts a certain number of rays from the eye through the screen into the dataset and calculates the intersection of the ray and the isosurface. From the intersection point, by exploiting isosurface's continuity property, part of the isosurface can be extracted efficiently.

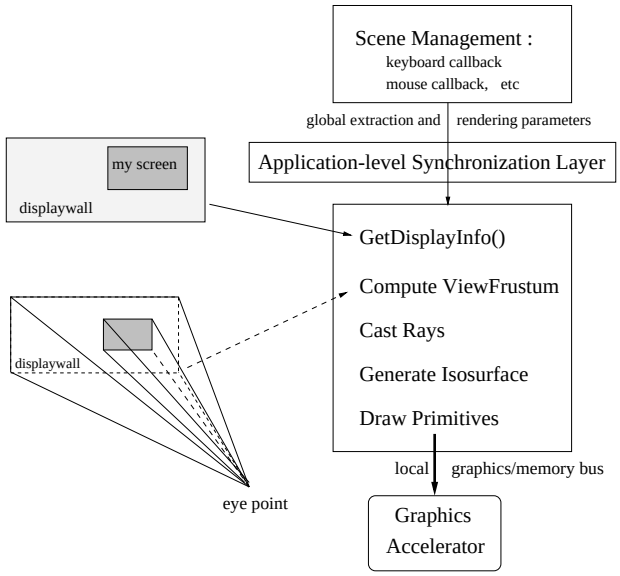


Figure 9: The flow diagram of the APE version of isoview

Running IsoView on the Display Wall can give the user more details and information about the data. Since this algorithm is screen-space based, it can be easily parallelized on the Display Wall. We partitioned the program into two components, the management and the extraction. The management component obtains user directives such as mouse and keyboard inputs. It sets the meta state of the program, which is synchronized by a function call that every program instance makes prior to performing the actual isosurface-extraction algorithm. The extraction component obtains the meta state and a node-specific screen tile position and calculates the perspective projection matrix that is used for both rendering and ray-casting. Data partitioning is automatically obtained through partitioned ray-casting. Figure 9 depicts the conceptual flow of the isoview program using application-level synchronization. Very little extra work is required to make the single-node isoview program into a display wall-aware version employing application-level synchronization. It took us only half an hour.

4.4 Past work on program replication

Synchronizing multiple instances of a same program has been studied by the fault-tolerant computing community. An early technique relied on a specially-built processor pair that executed instructions in lock steps. The memory accesses by the processors are checked by a special logic located between the processors and the memory subsystem. A third microprocessor was typically included in the mirror pair to provide what is known as triple modular redundancy (TMR). Some commercial vendors extended the hardware mirroring concept throughout the entire computer system to provide very high degree of availability in face of hardware component failures [1, 20]. Synchronizing processor pairs at the instruction level provides a transparent means to replicate program instances; the program need not be rewritten in order to run such a processor pair. However, this technique not only requires expensive engineering

efforts, but also faces a tremendous difficulty to keep up with the rapid increase of CPU clocks.

As an alternative, several commercial vendors of fault-tolerant computer systems provide special programming API to synchronize program instances in software. For example, on a Tandem computer system, an application uses messages to communicate with the operating system and other system services. Multiple instances of the application can be brought to be in sync as long as the messages they send and receive are synchronized, provided that between any two messaging events the application follows a deterministic execution path. A traditional architecture that only supports single-threaded execution guarantees this deterministic property. For a multi-processor multi-thread environment, program synchronization requires a special API at a level higher than messaging and application-specific programming.

Bressoud et al investigated a system-level approach to automatically and transparently replicate programs [4, 5]. Their key insight is that by virtualizing the hardware on which the program and the operating system runs, one can implement the traditional processor-pair approach as mirroring a virtual machine pair. The study by Bressoud et al further demonstrates that only a few I/O instructions in the virtual machine instruction set require synchronization. Their results show that two synchronized computer system using the virtual-machine-pair (or Hypervisor) approach runs at worse twice as slow as one system without synchronization. A significant benefit of the Hypervisor approach is that virtual machine synchronization (or replication) can be done entirely in software, thus can keep close track of the processor technology.

Unlike Hypervisor, our purpose is to synchronize a specific application program instead of the entire system. A Hypervisor-like virtual machine that mimics the actual processor is an overkill and incur too much synchronization overhead, as synchronization is performed on each I/O instruction. Instead, we lift the virtual machine abstraction even higher, and consider the interface between the application and the operating system as the virtual machine abstraction [1]. Synchronizing program instances at such a macro level reduces the synchronization frequency. In our case, it makes synchronization cost almost negligible.

5 Experimental Results

We have implemented all four approaches described in the previous sections: VDD (Virtual Display Driver), GLR (GL-DLL Replacement), SSE (System-level Synchronized Execution), and ASE (Application-level Synchronized Execution). These tools all run on the scalable display wall prototype system described in Section 2. In writing the SSE tool, we used the Detours package from Microsoft Research [10]. It is a binary re-write tool for intercepting any functions in a program or a DLL.

The display wall system has two communication mechanisms available to applications: the Microsoft Winsock over the 100 Mbit/sec Ethernet, and VMMC over the Myrinet. VDD, SSE and ASE all use the Winsock protocol while GLR uses VMMC. The main reason for the difference is that GLR requires a fast communication mechanism to run well. If a 3D application generates a large number of polygons at run time or so called immediate mode in GL, GLR needs high-performance communication paths to the render nodes to send polygons efficiently. The other three tools do not need to transfer polygons among nodes, so they work well with the Ethernet.

We used GLR, SSE, and ASE to run 3D applications in our experiments. The goal here is to understand the tradeoffs of these tools in terms of speed, memory requirements, communication requirements, I/O requirements, and software development efforts. The rest of this section reports our experimental results.

[1] Although we came up with this technique independently, we did later find a U.S. patent about exactly the same technique [3].

5.1 3D Applications

We have selected 4 representative applications: Cars, GLQuake, Atlantis, and Isoview. We chose these applications to cover a wide range of 3D animation characteristics.

- **Cars**: This is a demo program of the WorldUp Toolkit from EAI, Inc. It renders a virtual-reality scene that includes two highly sophisticated car models in an exhibition hall. There are a lot of texture details and fancy lighting in the scene. The cars themselves are made up of 200,000+ polygons each. This makes rendering time the dominant cost. All the objects in the scene are made into OpenGL display lists. The GLR tool needs to send the display lists once at the beginning of the program execution.

- **GLQuake**: This is an OpenGL program to view Quake scenes, downloaded from the Internet. The Quake scene description is in BSP-tree format. By parsing the BSP tree, the program generates polygons in real time. The scene that we use is fairly simple, containing only 12,722 polygons.

- **Atlantis**: This is a demo program from Silicon Graphics, Inc. It simulates a pool of swimming sharks, whales, and a dolphin. The body poses for each object are computed in real time, so it is not possible to use OpenGL display lists to avoid generating polygons for each frame. In our experiment, we increased the number of sharks to 300 to simulate a lively scene with many fish. This results in a large number of graphics primitives for each frame. Furthermore, in the application-level synchronization version of the program, we perform object culling based a coarse bounding sphere for each shark.

- **Isoview**: This is a program to visualize the isosurfaces of volumetric datasets. The program extracts a 3D surface from the data where $F(x, y, z) = v$ for a given threshold v. The program uses a combination of ray casting and isosurface propagation techniques to extract only the visible portions of the isosurface. Since the program allows users to change the view of the isosurface dynamically, it has to generate polygons for each frame at run time. Our experiment uses Isoview to view $256 \times 256 \times 209$, a down-sampled version of the dataset of the head of a visible woman [14]. The visualization of the data typically generates about hundreds of thousands of polygons per frame. With the ASE tool, this program performs high-level object culling.

The following table summarizes the features of the four 3D applications:

Applications	GL features	# Polygons/frame	Source
Cars	display list	200,000	no
GLQuake	immediate	12,722	no
Atlantis	immediate	94,549	yes
Isoview	immediate	645,237	yes

Since the source codes of Cars and GLQuake are not available, we are not able to run them with the ASE tool.

5.2 Results

Table 5.1 shows the results we gathered from running the four applications with GLR, SSE and ASE on the display wall system.

Performance Metrics	Applications/Methods											
	Cars			GLQuake			Atlantis			Isoview		
	GLR	SSE	ASE	GLR	SSE	ASE	GLR	SSE	ASE	GLR	SSE	ASE
frame time (ms)	370	325	-	80.6	65.7	-	147.6	86.2	57.9	20163	16825	4798
sync cost/frame (ms)	-	4.9	-	-	20.0	-	-	2.1	1.8	-	2.0	3.6
sync msgs/frame (count)	-	4	-	-	8	-	-	2	2	-	2	3
data size/frame (KB)	99.2	0.47	-	174	0.82	-	3300	0.24	0.19	46500	0.24	0.48

Table 1: The performances of three methods: GL-DLL Replacement (GLR), System-level Synchronized Execution (SSE), and Application-level Synchronized Execution (ASE)

The performance metrics include the average time for each frame, the number of synchronization messages and synchronization overhead (when appropriate), the amount of data sent over the network, and the maximum amount of memory consumed on each render node.

The frame times for each application running with the three tools are also plotted in Figure 10.

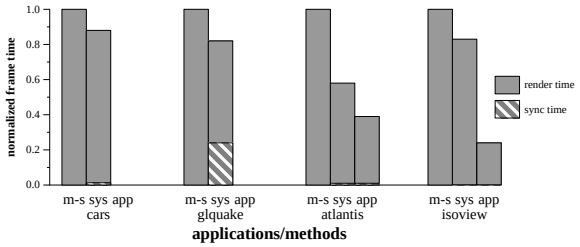


Figure 10: Frame time comparison of three methods

Our results show that the application-level synchronized execution approach using the ASE tool is the most efficient approach. For *Atlantis*, ASE is 2.5 and 1.48 times faster than GLR and SSE respectively. For *Isoview*, ASE is 4.2 and 3.5 times faster than GLR and SSE respectively. The second fastest is the system-level synchronized execution approach with the SSE tool. The master-slave approach with the GLR tool is the least efficient among the three.

There are two main reasons why ASE is the most efficient. The first is that its communication requirements are very small, less than 500 bytes per frame for either application. The second is that ASE provides each render node with opportunities to avoid doing duplicated work for other render nodes. To take advantage of this approach, one needs to modify the application to perform high-level object culling according to the knowledge about the tiled screen space. Our experience with both *Atlantis* and *Isoview* demonstrates that with a convenient mechanism such as application-level synchronization, one can easily modify the application codes. In the *Atlantis* case, the resulting reduction of shark pose computation and graphics primitives averages 50% for each render node. On the other hand, this approach requires accessing application source codes.

The system-level synchronized execution approach with the SSE tool works quite well without having to access application source codes. Similar to the application-level synchronized execution approach, it imposes very little communication overhead in all applications.

The master-slave approach using the GLR tool works reasonably well with applications that use display lists, because polygon data need to be transferred only at the beginning of the program and very little data during each frame. For example, the master node running the *Cars* program sends roughly 99 KBytes worth of data to the render nodes during each frame. Its performance is comparable to that of the system-level synchronization which requires very little communication. We do notice the frame time difference of 45 milliseconds between the master-slave approach and the SSE approach for the Cars program. Its cause is still unclear to us. We are investigating this issue using detailed performance monitor.

The master-slave approach is sensitive to the amount of immediate-mode graphics primitives. With the *GLQuake* program, the GRL tool has about 15ms overhead. This is because the program creates only a relatively small number of polygons per frame which translates to about 174 Kbytes per frame. With *Atlantis* and *Isoview*, the overheads are 61.4ms and 3338.7ms respectively because these two programs require the master node to transfer 3.3Mbytes and 46.5Mbytes per frame respectively. However, the percentage difference in the case of *isoview* is relatively small, because the computation time dominates.

Therefore, our conclusion is that for immediate-mode applications with low computation time per primitive, synchronizing multiple instances at the system level has a clear advantage over the master-slave approach. When per-primitive computation is high, communication time becomes either less significant or completely negligible due to its overlapping with computation.

On the other hand, the master-slave approach is the best in terms of memory requirements. The master node runs a single copy of the application on the master node, whereas SSE and ASE runs an instance of the application on every render node. The application-level synchronized execution approach can have smaller working sets than the system-level synchronized execution approach when it performs high-level object culling.

6 Conclusions

In this paper we described four software tools for bringing off-the-shelf, sequential applications onto a scalable display wall system. We have used these tools to experiment with several 2D and 3D applications on our display wall system to study the scalability issue. What we have learned is that these methods have different trade-offs in terms of speed, memory requirements and communication requirements.

The master-slave approach does not scale well in performance for applications that generate a large number of primitives dynamically, but it scales well in terms of memory requirements. Our experience with the GLR tool, for example, shows that the approach works reasonably well with 3D applications that use display lists whereas it performs poorly with applications using immediate-mode 3D primitives. Despite of its performance drawbacks, it is quite convenient to use.

The synchronized execution approaches trade memory space for less communication requirements. The system-level synchronized execution approach scales well in performance because its communication requirements are minimal. Our experiment with the SSE tool, for example, shows that it performs better than the master-slave approach with all applications. Like the master-slave

approach, it requires no source code modifications to applications. Unlike the master-slave approach, the SSE tool works only with applications that use a single thread to communicate with the external environment.

The application-level synchronized execution approach is the most efficient method when one performs high-level object culling. For *Atlantis*, ASE is 2.5 and 1.48 times faster than GLR and SSE respectively. For *Isoview*, ASE is 4.2 and 3.5 times faster than GLR and SSE respectively. By default, this approach takes the same amount of memory as the system-level synchronized execution approach, but by high-level object culling, it can reduce memory requirements. Unlike the system-level synchronized execution approach, this method works with all applications though it does require application source code modifications.

The software tools development described in this paper is our first step towards understanding how to build scalable display wall software systems. Our studies have several limitations. First, our GLR tool was implemented using the VMMC on Myrinet. Although the comparisons are conservative from the point of view of the synchronized execution model, it would be better if we can port the implementation to Winsock on the same Ethernet.

Acknowledgements

This project is part of the Princeton Scalable Display Wall project which is supported in part by Department of Energy under grant ANI-9906704 and grant DE-FC02-99ER25387, by an Intel Research Council and Intel Technology 2000 equipment grant.

We also would like thank the reviewers who provided much useful feedback on improving this paper.

References

[1] J. F. Bartlett. A nonstop operating system. In *Proceedings of the 11th Hawaii International Conference on System Sciences*, volume 3, 1978.

[2] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[3] T.C. Bressoud, J.E. Abhern, K.P. Birman, R.B. Cooper, B.B. Glade, F.B. Schneider, and J.D. Service. Transparent fault tolerant computer system. United States Patent 5,968,185, October 1999.

[4] Thomas C. Bressoud and Bruce B. Schneider. Hypervisor-based Fault-tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (ASPLOS V)*, pages 1–11, Copper Mountain Resort, Colorado, December 1995. ACM.

[5] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, February 1996.

[6] Yuqun Chen, Angelos Bilas, Stefanos N. Damianakis, Czarek Dubnicki, and Kai Li. Utlb: A mechanism for translations on network interface. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 193–204, October 1998.

[7] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 135–142, August 1993.

[8] J.D. Foley, A. van Dam, S. K. Feiner, and J.F. Hughs. *Simulated Annealing Methods*, chapter 0, pages ???–??? Addison-Wesley, 2 edition, 1996.

[9] Greg Humphrey and Pat Hanrahan. A distributed graphics system for large tiled displays. In *IEEE Visualization '99*, 1999.

[10] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *The 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, Washington, July 1999.

[11] Homan Igehy, Gordon Stoll, and Pat Hanrahan. The design of a parallel graphics interface. In *ACM 1998 SIGGRAPH*, 1998.

[12] Theo Mayer. New options and considerations for creating enhanced viewing experiences. In *Computer Graphics*, pages 32–34, May 1997.

[13] Bruce J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, May 1981.

[14] National Institute of Health. Electronic imaging: Report of the board of regents, u.s. department of health and human services, public health, national institutes of health. NIH Publication 90-2197, 1990.

[15] Ramesh Raskar, Greg Welch, Matt Cutts, Adam Lake, Lev Stesin, and Henry Fuchs. The office of the future: A unified approach to image-based modeling. In *SIGGRAPH 98*, pages 179–188, July 1998.

[16] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[17] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):260–273, April 1982.

[18] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual Symposium on Operating System Principles*, pages 40–53, December 1995.

[19] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[20] S. Webber and J. Beirne. The stratus architecture. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 79–85, 1991.