Parallelizing Pre-rendering Computations on a Net Juggler PC Cluster

Jérémie Allard Valérie Gouranton Emmanuel Melin
Bruno Raffin
Université d'Orléans
Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
F-45067 Orleans Cedex 2, France
gouranton@lifo.univ-orleans.fr, melin@lifo.univ-orleans.fr, raffin@lifo.univ-orleans.fr

Abstract

Net Juggler, an open source library developed at LIFO, turns a commodity component cluster running the VR Juggler platform on each node into a single VR Juggler image cluster. Net Juggler parallelizes rendering computations for immersive projection environments but pre-rendering computations are redundant. In this paper, we present how a classical parallelization of the pre-rendering computations can be deployed and controlled with Net Juggler. This approach is demonstrated with an interactive fluid flow simulation.

1 Introduction

Today, large computers built with PCs and a gigabit network are powerful enough to run high performance scientific applications. Such cluster architectures are now not unusual in the supercomputer top 500 [1]. Recent software developments ease the use of PC clusters equipped with graphics cards to power immersive projection environments where multiple video projectors form a high resolution and large surface display [12, 10, 15, 4]. It is then possible to consider a large PC cluster with most of the nodes dedicated to computations while the other nodes have graphics cards to power an immersive projection environment. Such architecture would offer scientists the possibility to visualize and control (in real-time) large-scale simulations.

Computations can be divided in two classes : pre-rendering computations and rendering computations. Rendering computations depend on the viewport while pre-rendering computations are independent from the viewport data. For example, in a fluid dynamics simulation the resolution of the Navier-Stokes equations is part of the pre-rendering computations while the image rasterization is part of the rendering computations. Libraries like Net Juggler [4], sizygy [18] or WireGL [15] provide automatic parallelization schemes for rendering computations. Pre-rendering computations go from "simple" scene graph management for walk-through applications to highly complex simulations of earth models for example. This diversity makes it difficult to develop a general approach for automatic parallelization of pre-rendering computations. Different tools are available, providing different levels of abstraction and different performances. Message passing libraries (MPI [14]) provide communication primitives for inter-process communications. Multi-thread programming requires appropriate mechanisms to hide the distributed memory of the PC cluster (OpenMP [11]). Distributed object computing with middlewares like Corba needs specific software architectures to allow remote method invocations. Higher-level tools are also available for special purposes, like parallel equation solvers (ScaLAPACK [9] or PEMCS [6]). For distributing, coupling and controlling pre-rendering and rendering parallel computations running on a PC cluster, the user faces a lack of adapted platforms. In this paper, we present how a classical parallelization approach for pre-rendering can be coupled and controlled with rendering using the Net Juggler platform.

Net Juggler [4, 2], an open source software developed at LIFO, turns a graphics cluster running the VR Juggler platform [8, 3] on each node into a single VR Juggler image cluster. In a transparent way for the user, Net Juggler duplicates the VR Juggler application on each node and broadcasts input events to guaranty data coherency between each copy. The required network bandwidth is limited due to the small amount of data communicated, ensuring high performance even for complex and fast changing real-time applications. VR Juggler applications are independent from the execution environment. When launching the application, the user provides configuration files containing data to instantiate the ap-

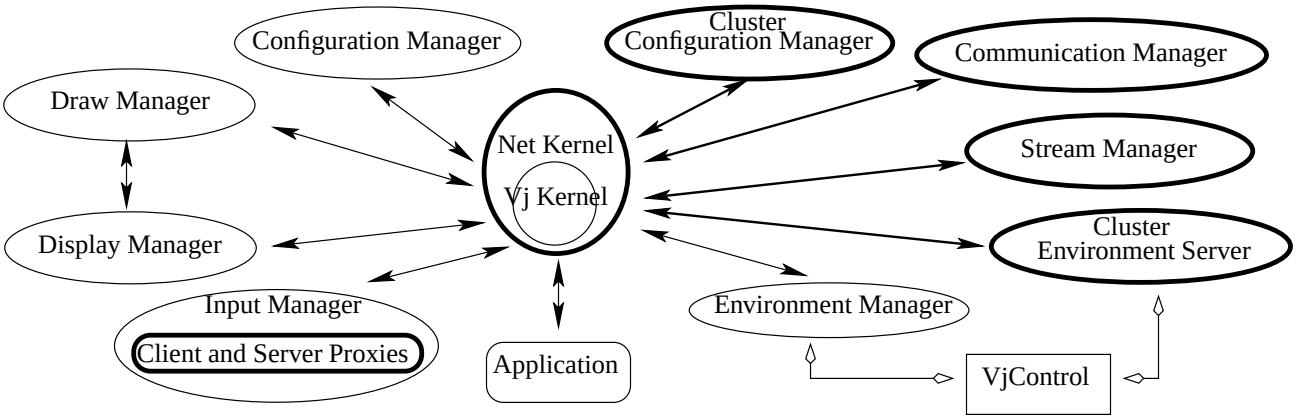

Figure 1. VR Juggler (thin lines) and Net Juggler (thick lines)

plication according to the target execution environment.

To broadcast input device data, Net Juggler maintains a MPI [14] environment between the different VR Juggler kernels running on each node. We reuse this MPI environment to run a MPI based parallelization of pre-rendering computations. The user is responsible for parallelizing the pre-rendering computations directly with MPI or any MPI based high-level parallel library. Once the pre-rendering computations are parallelized, Net Juggler provides a convenient way to take advantage of the computing power a cluster can offer. Extra nodes that do not have graphics cards can be used for pre-rendering computations. When launching the application, the user has to set the configuration files so that Net Juggler runs a copy of the application on these nodes only activating the pre-rendering computations. It is then possible to have pre-rendering computations executed on a large number of nodes, while rendering is run on the nodes driving the projectors of the immersive environment.

We first give a quick overview of Net Juggler and VR Juggler architectures in section 2. Section 3 exposes Net juggler contribution to pre-rendering and rendering parallel computation coupling. We present a parallel fluid flow interactive simulation and performance results in section 4 before to conclude.

2 VR Juggler and Net Juggler

We first present VR Juggler main concepts and architecture before to expose Net Juggler and how it allows VR Juggler to support clusters.

2.1 VR Juggler

The open source VR Juggler library [8] defines a execution platform for virtual reality applications. VR Juggler provides an abstraction of the underlying system, while giving direct access to various graphics API for maximum control over applications. The application is independent of the displays, the input and output devices. System components are configured with a set of files when launching the application. VR Juggler [8] is organized around a kernel and different components called managers (Fig. 1). Each manager handles a set of specific system details, while the kernel controls the run-time system and brokers communications between the different managers. Every input device is controlled by the input manager. When an application requests access to a device, it contacts a proxy. The proxy hides the actual device and tracks the most recent data received from the device. The draw manager gives direct access to the graphics API. The display manager takes care of the windows and displays. The configuration manager handles a database with configuration information, like window properties, proxy names and associated devices. The environment manager is the user's entry point to exchange data with VR Juggler. With the graphics utility called VjControl, the user can reconfigure the application at run-time or collect performance data.

2.2 Net Juggler

To add cluster support to VR Juggler requires new functionalities. Following VR Juggler microkernel organization we implemented new managers (Fig. 1). The Net Juggler kernel, deriving from the VR Juggler kernel, controls these managers.

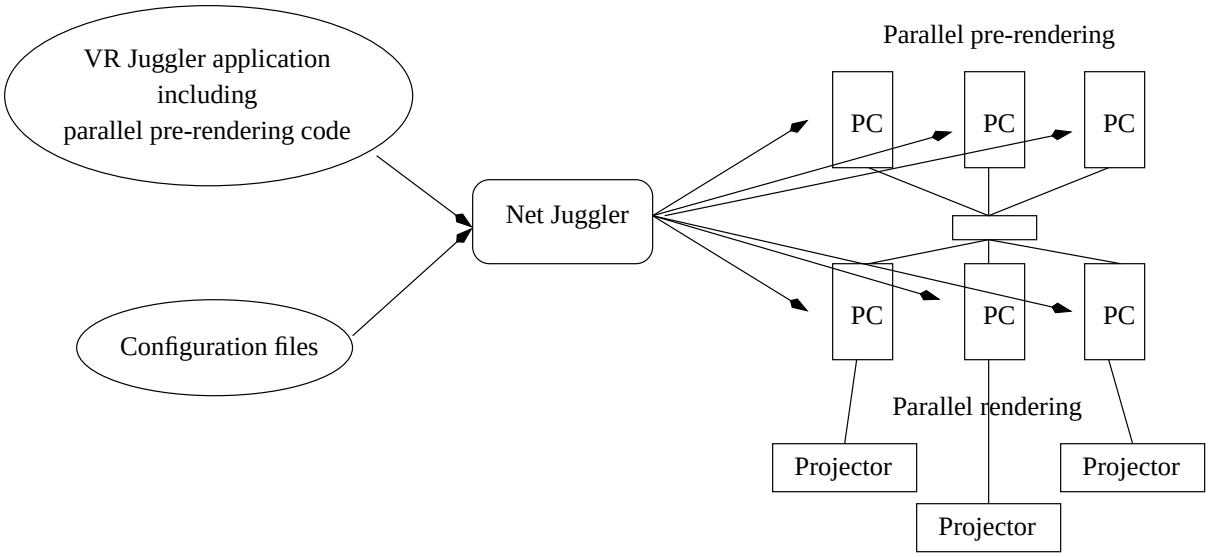

Figure 2. Pre-rendering and rendering parallel computations on a Net Juggler PC Cluster

Net Juggler has to collect data from each device and broadcast them to each node of the cluster. VR Juggler uses proxies to hides the actual devices. Net Juggler replaces these proxies by client and server proxies. The node where the device is connected runs a server proxy while the nodes requiring the data run a client proxy.

VR Juggler instantiates the application according to the execution environment with a set of configuration files. The user can also request reconfigurations at run-time with Vj-Control. Configuration data are organized in chunks, each chunk having information about a part of the system. The same functionalities are available with Net Juggler. For a Net Juggler cluster, chunks have to be modified to include a host parameter. The host indicates the node the chunk is related to. On each node, a cluster configuration manager stores the chunks in a database. The whole cluster configuration is then easily available from any node of the cluster. Each node selects the Net Juggler chunks it is concerned with and translates them into VR Juggler chunks. These chunks are then processed to instantiate the code running on that node.

The classical stream paradigm is used and extended to provide an abstraction of the actual data communications. There is one stream by server proxy and by cluster environment server. A stream is associated to a specific node source and can have several destination nodes. Each stream can be created, deleted or modified at run-time. Actual data communications take place only once per frame. The nodes also execute a synchronization barrier just before swapping their frame buffers to ensure the consistency of the displayed images. Communications are implemented with the MPI standard [14].

3 Pre-rendering Parallelization

In a transparent way for the user, Net Juggler parallelizes rendering computations. Because the application is duplicated on each node, pre-rendering computations are repeated on each node. Thus, the computation power available for pre-rendering is at most the computation power of one node. We expect this to be sufficient for a broad range of applications. For other applications, a parallelization is required to distribute the pre-rendering computations on different nodes and thus to obtain a computation power that is (in the best case) the sum of the power of the nodes.

Today, no satisfactory solution exists to automatically and efficiently parallelize any sequential code. Pre-rendering covers a wide range of applications and unless we restrict ourselves to a well-defined class of pre-rendering computations, it seems unrealistic to develop a generic and high performance parallelization scheme. Therefore, we expect the user to provide the parallelization that best fits its pre-rendering computation requirements. Once pre-rendering computations are parallelized, difficulties remain for coupling pre-rendering and rendering parallelizations. We show in this paper how Net Juggler can conveniently help the user to achieve this goal and how VR/Net Juggler configuration files are used to distribute the computations on the cluster nodes.

Net Juggler communicates input events and configuration data between nodes using MPI [14]. MPI is a message

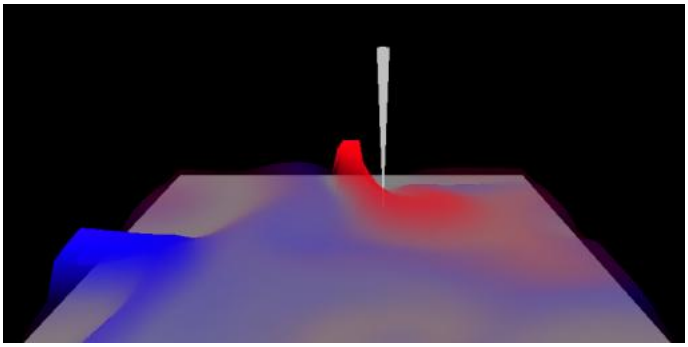

Figure 3. The NjFluid Application.

passing standard that has been ported to a wide range of platforms from Cray T3Es to Linux and Windows PC clusters. On top of user-level protocols [7, 16, 13] for gigabit networks like Myrinet, some MPI implementations reach performance levels that compete with proprietary supercomputers [17]. The message passing programming model does not generate an important uncontrolled network traffic, in opposite to virtually shared memory implementations for example. This is essential to achieve high performance on a PC cluster where nodes are loosely coupled. Because MPI has been used for several years, many programmers have MPI skills, many scientific codes are parallelized with MPI and several libraries on top of MPI implement highly optimized parallel algorithms (ScaLAPACK [9] or PEMCS [6] for example). In this context, MPI appears as a good candidate to parallelize pre-rendering computations on PC clusters.

Once pre-rendering computations are parallelized directly with MPI or any MPI based high-level parallel library, the code is inserted in the VR Juggler rendering loop like we would do with any sequential pre-rendering code. When the application is launched, each node should identify its role in the computation (pre-rendering, rendering or both). To include the name of the nodes and their role in the code would compromise scalability and portability. Net Juggler architecture provides an elegant solution to that problem. Each node runs a Net Juggler cluster configuration manager storing the whole cluster configuration. Thus, each node can access locally this database to know what it has to do. This database is also useful to identify the role of the other nodes. Because some data are distributed by the pre-rendering parallelization, pre-rendering nodes have to send some data to rendering nodes. The target rendering nodes are identified by the data stored in the cluster configuration manager.

The application is launched like any VR juggler application including only sequential pre-rendering code. The role of the different nodes is specified in the configuration files (Fig. 2).

VR Juggler tools are still available. VjControl can be used to modify the cluster configuration at run-time. VjControl can also be used to retrieve performance data from any given node.

4 The NjFluid Application

To test our approach we developed an interactive fluid flow simulation. When the application starts, no fluid is present in the simulation space. As time goes, a blue fluid and a red fluid are ejected from two different sources. These fluids spread, collide and mingle with each other. The user can interactively move a virtual stick to mix the fluids (Fig 3). To simulate fluids we implemented the Navier-Stokes equation solver proposed by Stam in [19, 20]. The primary goal of this solver is to exhibit all the visual characteristics of a real fluid, such as swirling flows around bodies. The speed of the simulator is crucial too, since we want a real time feedback in a virtual environment. This solver is a typical example of intensive pre-rendering computations. To reach the required performance level we implemented a parallel version of this solver that we integrated in a VR Juggler application. Executed with Net juggler on a PC cluster, the solver is parallelized on some nodes, while rendering computations are distributed on graphics nodes to power an immersive projection environment.

Stam's solver operates on a grid of cells discretizing the space where the fluid can flow. Each cell holds a fluid velocity vector and a scalar fluid density. These data characterize the fluid present in a given cell. Stam's solver updates the fluid velocity and next the density. These computations require simple matrix computations, a conjugate gradient and a Poisson solver. For more details refer to [19, 20].

To implement a parallel version of Stam's solver we used the PETSc [6, 5] parallel library. This library includes

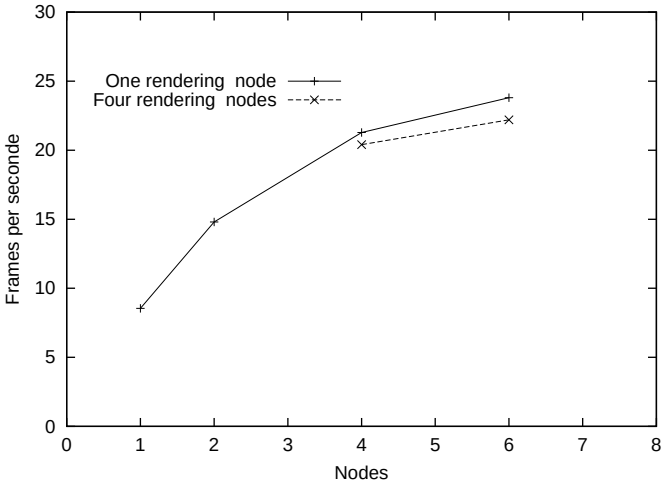

Figure 4. Performance results for the NjFluid application.

parallel linear and nonlinear equation solvers, support for distributed arrays, parallel matrix and vector assembly routines. PETSc is based on MPI, thus it naturally fits our MPI based approach. Using PETSc we avoid managing explicitly data distribution and data communications.

Our application uses one velocity field and two density fields, one for each fluid. A 2D grid discretizes the horizontal plane where the fluids flow. This 2D grid is divided in 2D blocks of cells that PETSc distributes on a 2D array of processes. Stam's solver requires each cell to know the data of its four neighbors. For the parallel version of the solver this implies communications between neighbors' blocks to exchange the data of their border cells. These communications are handled by PETCs. The only communication the user has to explicit is to send the density values computed on the pre-rendering nodes to the rendering nodes. The cells are then colored according to their density. Because rendering computations are limited and mainly executed on the graphics cards, we favored a more balanced CPU usage having rendering nodes to execute pre-rendering computations too.

NjFluid was tested with 6 dual Pentium III 800 MHz nodes connected with a 100 Mbit/s Fast Ethernet network. Four nodes were equipped with GeForce 2 GTS 64 MB DDR graphics cards. On each node we installed a Mandrake 7.2 linux distribution. Fluids flowed on a $128x128$ grid (Fig 4). NjFluid was first tested with one rendering node driving a single display. The frame rate increases with the number of pre-rendering nodes to reach 23 frames per second. In this case, there is no rendering parallelization (only one display). The performance improvement is only due to the pre-rendering parallelization. NjFluid was next tested with four rendering nodes driving four displays. Net Juggler synchronizes the four displays that show complementary parts of the scene. Because each rendering node also executes pre-rendering computations, the frame rate is close to 20 frames per seconds, only one frame less than with one rendering node and three pre-rendering nodes. With a sequential version of the solver where each rendering node would execute all the pre-rendering computations, the frame rate would be close to 8 frames per second instead of 20. Adding two pre-rendering nodes does not significantly increase the frame rate. At this point, the performance is close to the maximum available with this configuration. The communication overhead becomes too important.

From our point of view, the development of this application was not significantly more difficult than to develop a sequential version that would only run on a single PC. Most of the parallelism is hidden either by PETCs or by Net Juggler. The resulting application is scalable and portable. It leads to efficient executions on a PC cluster.

5 Conclusion

Today, PCs are assembled to build low cost supercomputers. Such PC clusters can run intensive applications that the user can control in real-time through an immersive projection environment also powered by some nodes of the cluster. To favor application scalability and portability it is desirable to have software platforms that define an abstraction of the execution environment and that allow high performance executions.

To achieve this goal we propose to use VR Juggler, Net Juggler and MPI. In this paper, we presented our early experiences in using these libraries to combine high performance pre-rendering and rendering parallel executions on a PC cluster. For pre-rendering parallelization, the user can

elect any MPI based tool that fits his computation requirements. VR Juggler and Net Juggler ensure the parallelization of rendering computations, pre-rendering and rendering code coupling, and the control of the roles of the nodes. The resulting application is scalable and portable. We introduced an example where a parallel Navier-Stokes solver was implemented on top of the PETCs library. The application was easily and efficiently executed on different cluster configurations.

Because pre-rendering computations are triggered by the VR Juggler rendering loop, they are synchronized with the frame rate, i.e. a new frame cannot be rendered if a new pre-rendering computation step has not ended. Slow pre-rendering computations can limit the frame rate and impact the immersion illusion. Future research works will focus on extending Net Juggler to define a programming and execution model to handle asynchronism between pre-rendering and rendering computations and more generally between tasks executed on different nodes.

References

[1] Top 500 Supercomputers, www.top500.org.

[2] Net Juggler. netjuggler.sourceforge.net.

[3] VR Juggler. www.vrjuggler.org.

[4] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin. Net Juggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster. In *IEEE VR 2002*, Orlando, USA, March 2001.

[5] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc Web page. www.mcs.anl.gov/petsc.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc 2.0 Users Manual. Technical Report ANL-95/11 - Revision 2.0.29, Argonne National Laboratory, Nov. 2000.

[7] R. A. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[8] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *IEEE VR 2001*, Yokohama, Japan, March 2001.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[10] M. Bues, R. Blach, S. Stegmaier, U. Häfner, H. Hoffmann, and F. Haselberger. Towards a Scalable High Performance Application Platform for Immersive Virtual Environements. In J. D. B. Fröhlich and H.-J. Bullinger, editors, *Immersive Projection Technology and Virtual Environements 2001*, pages 165–174, Stuttgart, Germany, May 2001. Springer.

[11] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[12] Y. Chen, H. Chen, D. W. Clark, Z. Liu, G. Wallace, and K. Li. Software Environments for Cluster-based Display Systems. www.cs.princeton.edu/omnimedia/papers.html, 2001.

[13] P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP: High Performance Message Passing over a Cluster of Commodity SMPs. In *Proceedings of Super Computing 99*, Portland, USA, November 1999.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Scientific and Engeneering Computation Series. The MIT Press, 1994.

[15] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. WireGL: A Scalable Graphics System for Clusters. In *Proceedings of SIGGRAPH 2001*, 2001.

[16] M. Lauria and A. A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, 1997.

[17] G. R. Luecke, B. Raffin, and J. J. Coyle. Comparing the Communication Performance and Scalability of a Linux and a NT Cluster of PCs, a Cray Origin 2000, an IBM SP and a Cray T3E-600. In *Proceedings of the IEEE International Workshop on Cluster Computing (IWCC'99)*, pages 26–35, Melbourne, Australia, December 1999.

[18] B. Schaeffer. A Software System for Inexpensive VR via Graphics Clusters. www.isl.uiuc.edu/ClusteredVR/ClusteredVR.htm.

[19] J. Stam. Stable Fluids. In *SIGGRAPH 99 Conference Proceedings*, pages 121–128, August 1999.

[20] J. Stam. Interacting with smoke and fire in real time. *Communications of the ACM*, 43(7):76–83, 2000.