Flowvr: a middleware for large scale virtual reality applications

Jérémie Allard, Valérie Gouranton, Loïc Lecointre, Sébastien Limet, Emmanuel Melin, Bruno Raffin, Sophie Robert

FlowVR: a Middleware for Large Scale Virtual Reality Applications

Jérémie Allard[1], Valérie Gouranton[2], Loïck Lecointre[1], Sébastien Limet[2], Emmanuel Melin[2], Bruno Raffin[1], and Sophie Robert[2]

[1] Laboratoire ID, CNRS/INPG/INRIA/UJF, Montbonnot, France
[2] LIFO, Université d'Orléans/CNRS, Orléans, France

Abstract. This paper introduces FlowVR, a middleware dedicated to virtual reality applications distributed on clusters or grid environments. FlowVR supports coupling of heterogeneous parallel codes and is component oriented to favor code reuse. While classical communication paradigms focus on either a synchronous approach (FIFO channels) or an asynchronous one (sampling), FlowVR enables a large range of intermediate policies to better balance the application performance between levels of details, latencies and refresh rates.

1 Introduction

Classically, a virtual reality (VR) application features a complex simulation using input and output devices to provide users with a sense of immersion in a synthetic world [7]. Most of today's VR applications only run on machines with a reduced number of processors, like visualization clusters or SGI Onyx. They do not take advantage of the computing power offered by large clusters and grid environments. One main limitation is the difficulty to assemble and distribute the different (potentially parallel) components and to maintain the overall application *coherent* while guaranteeing a good quality interaction with *low latency* and *high refresh rates*. We define the *coherency* as the fact that the information provided to the user senses at a given moment are related to the same simulated time.

To improve latency and refresh rates, VR applications can take advantage of a data exchange model based on sampling. The producer updates data in a shared buffer asynchronously read by the consumer. Some updates may be lost if the consumer is slower than the producer. While asynchronism leads to a performance improvement, the application coherency cannot be maintained. Depending on the context this may be acceptable. It is for example used when coupling haptic and visualization systems that run at very different frequencies (about 1000 Hz and 60 Hz respectively). Distributed virtual environments [9, 11] or VR middlewares like OpenMask [2] use such an approach, but parallel code coupling becomes difficult in this context as no coherency control is offered. The other approach classically used for parallel programming, parallel code coupling [8, 10], or distributed visualization environments [3–5], relies on a classical

FIFO synchronization semantics. It ensures proper application coherency, but it is difficult to efficiently implement a sampling approach.

In this paper, we propose a programming model that eases the implementation of a large range of synchronization policies, from FIFO to sampling. We present FlowVR [1], a middleware dedicated to VR and supporting coupling of heterogeneous parallel codes to build large scale applications. FlowVR reuses and extends the data flow paradigm commonly used for scientific visualization environments [3, 4]. A VR application is seen as a set of possibly distributed modules exchanging data. Each module endlessly iterates, consuming and producing data. From the FlowVR point of view, modules are not aware of the existence of other modules, the FlowVR engine taking care of moving data between producers and consumers. This leads to a simple application programming interface (API) that eases turning an existing code into a FlowVR module (or several modules in case of a parallel code). For data exchange between modules, FlowVR defines an abstract network featuring from simple routing operations to complex message handling operations. Each message is associated with a *list of stamps*, a lightweight data used to route or filter messages. This list can also be routed separately from its message to special network nodes in charge of synchronization policies. Besides predefined FlowVR stamps, others, like a time or a 3D bounding box for instance, may be added to extend the network routing, filtering or synchronization abilities. The FlowVR network enables to build complex collective communications, a desirable feature for efficient parallel code coupling. It is also possible to go beyond the classical synchronization barrier, designing synchronizations waiting for the resolution of complex constraints based on stamps (a data semantically richer than a signal). Different FlowVR networks can be designed without modification of the module codes.

2 The FlowVR Application Model

In this section we introduce the FlowVR application model.

2.1 Running Example

All along this paper, we use a simple yet important example, an interactive VR application where the user can perturbate a fluid flow simulation with its hand. We distinguish three parts :

- A tracker that gives the user's hand position.
- A physical fluid simulation parallelized with MPI. The simulation is based on a 2D grid split in blocks amongst the different MPI processes. MPI communications take place at each iteration to exchange the values of the grid borders between neighbors. Each process should also receive the hand position, which acts as an obstacle for the fluid flow.
- A multi-projector visualization environment. Each projector, driven by its own PC, displays a tile of the entire scene. The distribution paradigm adopted

```
initialization
while not stop
    wait()
    get(position)
    computations
    put(grid)
```

Fig. 1. The algorithm of the simulation module.

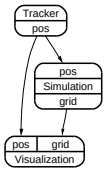


Fig. 2. Interactive fluid simulation with 3 modules.

is simple: all PCs run a copy of the visualization application, each one expecting the coordinates of the hand position and a density grid at each iteration. To ensure a strong coherency of the displayed images, these copies must receive the same input data at each iteration. Next, each copy computes its tile of the global image based on its own viewing frustum (viewing angle). All PCs must then display the new image synchronously, either using a hardware swaplock or a software barrier.

These codes can run independently at very different frequencies. The tracker is certainly the fastest one and the fluid simulation the slowest one. A sampling-based data exchange model will let the codes run independently at their highest frequency, but it may lead to incoherences. For instance, in a given image, the displayed hand position may not correspond to the one used to compute the displayed simulation state. On the opposite, a FIFO communication model will ensure the overall application coherency, but at the price of a lower performance. All codes will run at the same frequency, synchronized on the slowest one. The tracker will produce a new data as soon as room is available in the output channel buffer. The latency will increase by the time such data stay unused in this buffer, the time required by the fluid simulation to consume all data previously stored in this buffer. FlowVR has been designed to let the user specify these different policies and other *intermediate* solutions, without requiring any modification of the codes.

2.2 Modules

We first introduce the API used to program FlowVR *modules*. This API is kept as simple as possible to limit the effort required to convert an existing code into a FlowVR module. For that purpose we explicitly took advantage of the interactive nature of VR applications. A FlowVR module is a computation loop periodically reading input data and producing new results. To improve code reuse, a module cannot directly address another module. This way there is no explicit dependency between modules. Their only knowledge of the FlowVR environment is a list of input and output ports. The module API is based on three main methods:

– The *wait* defines the transition to a new iteration. It is a blocking call that ensures each connected input port holds a new message. Input ports not

connected to any other port will never receive any message. They are *deactivated*.
- The *get* function enables a module to retrieve the message available on a port.
- The *put* function enables a module to write a message on an output port. Only one new message can be written per port and iteration. Each output message is automatically stamped by FlowVR with the current iteration number.

In our example, we would define:

- One module for the tracker with one output port (a position data).
- Each MPI process of the fluid simulation will define a module with one input (a position data) and one output (its block of the fluid density grid) (Fig. 1). To be able to distinguish the different blocks, each process stamps its output messages with the coordinates of its block.
- One module for each visualization process, with two input ports each, one to retrieve the tracker position and the other one to retrieve the whole density grid.

Each module has two additional predefined ports. The *input activation port* is used to lock the module to an external event (fixed frequency trigger for the tracker for instance). The *output activation port* is used to signal other components that the module has started a new iteration (see section 2.5).

2.3 Connections

Once modules are defined, they are assembled connecting their input and output ports. The simplest primitive used to build a FlowVR network is a *connection*. A connection is a typed FIFO channel with one source and one destination. Messages in a connection are numbered. Each message is stamped with this number and the source id.

Let us consider our example. We can build a simple first application with one tracker module, one fluid simulation module and one visualization module (Fig. 2). We add one connection from the tracker to the visualization, another one from the tracker to the simulation and a last one from the simulation to the visualization. This simple application implements a classical communication scheme using FIFO channels. The FIFO connections ensure a strong coherency. At each iteration the visualization module will always retrieve a tracker position and a density grid corresponding to the same simulated time. Therefore the resulting application will be synchronized on the slowest module, presumably the fluid simulation. If the tracker module is faster than the simulation module, there will be a significant lag between between user interactions and their effects on the virtual world. Also notice that adding the connections does not require to modify the code of the modules.

However, having only point to point FIFO connections, it is difficult to loosen the synchronizations imposed by the FIFO model or to express collective communications.

2.4 Filters

To extend the capabilities of the FlowVR network we introduce a new component, called *filter*.

A filter has typed input and output ports and can perform complex operations on messages. Filters have all the freedom to discard, combine or even generate messages. They are not restricted to receive only one message per port and per iteration like modules. They have free access to incoming buffers. Filters usually handle messages based on the associated lists of stamps. For instance, a filter can discard all incoming messages, which 3D bounding box falls outside of a given volume. Amongst filters, we distinguish the *routing nodes* as the filters that only forward all incoming messages on one or several outputs.

Let extend our example by now using four modules for the simulation and two modules for the visualization (Fig. 3(a)). The tracker messages must be broadcasted to these modules. For that purpose we introduce in our network several routing nodes. To broadcast the data to modules we choose to implement a binary-tree broadcast. The data exchange between simulation and visualization is more complex as we have to ensure that all visualization modules receive the whole density grid while each simulation module sends only one fourth of it. For that purpose we use a filter that combines two blocks of density grids into a larger one. This example implements a network with non trivial collective communications. A strong coherency is still ensured as the filter we use here does not suppress or generate new data (FIFO network).

2.5 Synchronizers

We distinguish a special class of filters, called *synchronizers*, used to implement the resolution of non local constraints. A synchronizer works on stamps. Therefore all incoming and outgoing connections only carry message stamps. Generally a synchronizer activity is triggered by incoming stamps on some selected ports. As synchronizers do not receive the data part of the messages, their output ports are generally connected to filters. These filters typically have 2 input ports, one receiving full messages (the data and its list of stamps) from a module or a filter, and the other one receiving only stamps from a synchronizer. The filter processes the incoming full messages according to incoming stamps. For instance, such a filter can forward to its output only the full messages corresponding to the incoming stamps, discarding the other messages.

Classical synchronization schemes can often be expressed in term of signal handling. In this case the synchronizer only uses its inputs as signals. A sampling scheme is implemented by selecting the last received message each time an activation signal is received from the destination module (request for another input message). But synchronizers can implement more complex algorithms by taking advantage of the semantically rich information hold by stamps. For example, in VR environments some coherency constraints can be expressed in term of spatial relationships. A strong coherency is required for objects close to the user, while background or unseen parts of the scene require much less attention. A stamp

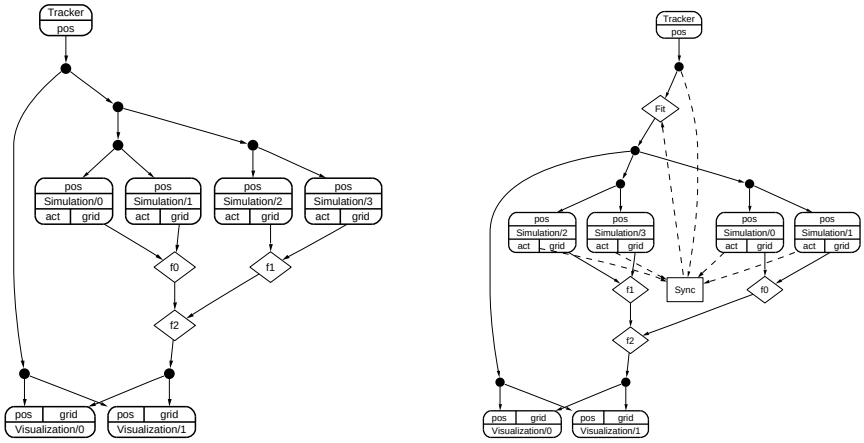


Fig. 3. (a) Fluid simulation with a FIFO network. Modules are represented as round-shaped squares, routing nodes as circles and filters as diamonds. (b) Fluid simulation with a coherent sampling network using one synchronizer (a square). Dashed lines correspond to connections carrying only stamps. The *act* port corresponds to the output activation port.

holding a bounding box information can be used to implement such a coherency policy.

In our example, because the simulation will probably be slower than the tracker, we introduce a synchronizer to keep pace with the tracker (Fig. 3(b)). This synchronizer takes as input the stamps from the position messages, and the stamps from the activation output ports of the fluid modules. When all fluid modules request a new data, the synchronizer selects the newest stamp available and sends it to the filter `Fit`. This filter only forwards on its output port the messages having the stamps selected by the synchronizer. A strong coherency is ensured as the visualization and simulation modules receive the same position messages. Similar ideas could be applied to implement a coherent sampling scheme to enable the visualization to run asynchronously from the simulation. Once again, building this network did not require any modification of the module codes.

3 Runtime Engine

FlowVR is open source and currently ported on Linux for IA32, IA64 and Opteron.

The FlowVR runtime engine relies on daemons, one per participating node. Daemons are in charge of FlowVR networks. They act as brokers and relay messages between modules. Filters, including synchronizers, are implemented as

dynamically loaded classes (*plugins*) within the daemon. Communications local to a node use a shared memory area. Care is taken to avoid unnecessary data copies and memory allocations by exchanging pointers and reusing allocated buffers. The current implementation of inter-node communications relies on TCP. Networks of heterogeneous nodes are easily exploited, as connections are dynamically created and each daemon can be launched independently. Several applications can safely run concurrently using the same daemons.

Each FlowVR application is managed by one special module called a *controller*, automatically loaded at starting time. The controller first starts the application's modules using their own launching command, `ssh` or `mpirun` for instance. Once the modules launched, they register themselves to their local daemon that sends an acknowledgment to the controller. Then, the controller sends to each daemon the list of plugins to load to implement the FlowVR network.

FlowVR integrates tools to generate the module launching commands and the list of plugins to load. It uses as input an XML description of the syntax of the launching command associated with each module code, as well as an XML description of the FlowVR network with an explicit placement of all components on target nodes. Ongoing work focuses on developing automatic and semi-automatic FlowVR network generation tools.

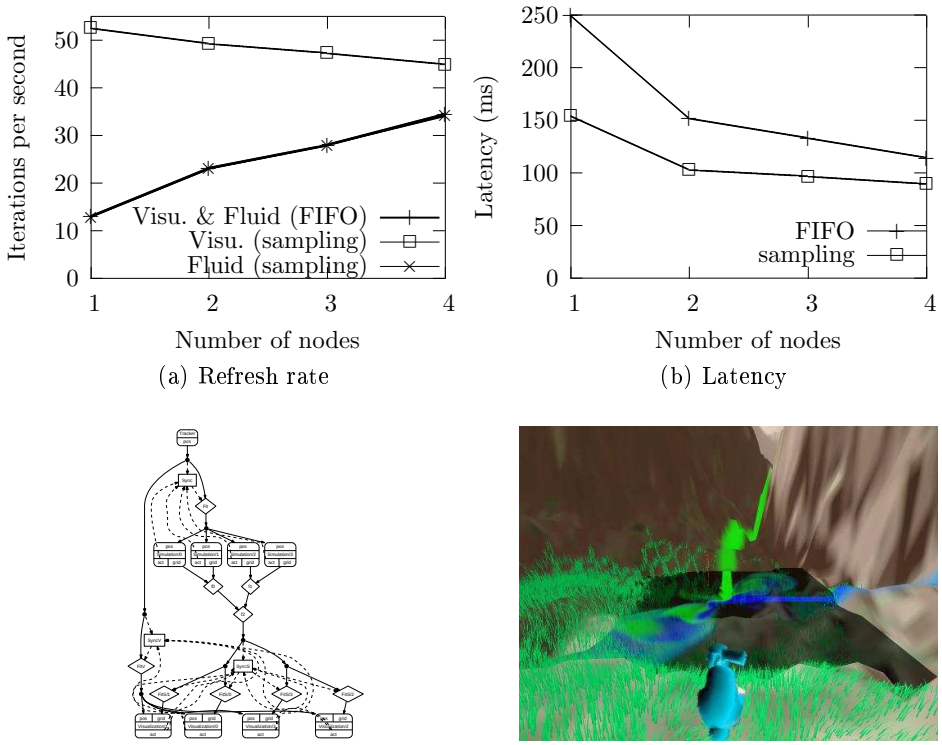
3.1 Experimental Results

We implemented the running example porting an existing fluid simulation code. The fluid simulation is parallelized with MPI, while the multi-projector visualization is handled by Net Juggler (also based on MPI) [6]. From the FlowVR point of view, each MPI fluid process and each Net Juggler process is seen as a module. Note that all fluid modules (respectively visualization modules) are synchronized through MPI communication calls FlowVR is not aware of. All results presented here run a fluid simulation based on a 2D 512×512 grid. The visualization modules integrate the fluid into a rich virtual environment (See Fig. 4(d)).

Two versions of the network were tested, a FIFO network (similar to Fig. 3(a)), and a coherent sampling network enabling the tracker, the fluid simulation and the visualization to run asynchronously. It extends the network presented in Fig. 3(b) by adding an extra synchronizer between the tracker and the visualization, and another one between the simulation and the visualization (Fig. 4(c)). Tests were performed on a PC cluster with dual Xeon PCs (2.66 GHz) connected through a Gigabit Ethernet network. Each machine was equipped with a GeForce FX 5600 graphics card.

The number of visualization and fluid modules vary from 1 to 4. Each module runs on its own PC. For instance when 8 nodes are used, 4 of them execute a fluid module, while each of the 4 other PCs run a visualization module. Each of these 4 PCs drives a video projector to display the result of its visualization module (1/4 of the global image).

We measured the refresh rate, i.e. number of iterations per second, for the visualization and the fluid simulation (see Fig. 4(a)). The FIFO networks im-



(a) Refresh rate (b) Latency

(c) Coherent sampling network (d) Screenshot of the visualization

Fig. 4. Experimental results with a coherent sampling network and a FIFO network.

pose the same refresh rate for the visualization and the fluid modules. For the coherent sampling network, the visualization and the fluid run asynchronously. It enables the visualization to run significantly faster than the simulation. The fluid simulation keeps the same performance as in the FIFO case. It shows that the communications induced by synchronizers do not significantly affect the performance. As the number of nodes allocated to the fluid simulation increases, the fluid performance increases too. For the sampling approach this decreases the refresh rate of the visualization modules as they must upload to the graphics card new data from the fluid modules more frequently.

We also measured the overall latency, i.e. the time lag between the time a new tracker position is available and the end of the iteration of the visualization modules using this tracker position (see Fig. 4(b)). Allocating more nodes to the simulation also improves latency. Sampling leads to a better latency than FIFO, because sampling uses the more recent data available while FIFO uses the older one. Note that the FIFO was executed with intermediate buffers of size 2.

The synchronizers used for the sampling approach can be extended to enable a finer control over dependencies between modules. For instance, the synchro-

nizer between the fluid modules and the visualization modules could take into account a user position data to know for each visualization module if the fluid is visible or not. If not, it could block the transmission of fluid grid to the visualization module, to let the visualization and network resources fully available for objects that are in the user field of view.

4 Conclusion

We introduced FlowVR, a middleware dedicated to distributed interactive applications. FlowVR distinguishes two main parts in an application, the modules and the network. Modules are endless loops reading and writing data on input and output ports. Modules are assembled in a network with advanced features for message handling. It enables parallel code coupling and the design of complex communication and synchronization schemes. First experiences show that FlowVR eases the development and deployment of interactive distributed applications, while leading to high performance executions.

Acknowledgment

This work is partly funded by the RNTL project Geobench.

References

1. FlowVR. http://flowvr.sf.net.
2. OpenMASK. http://www.irisa.fr/siames/OpenMASK.
3. Scirun: A scientific computing problem solving environment. http://software.sci.utah.edu/scirun.html.
4. *Covise Programming Guide*, 2001. http://www.hlrs.de/organization/vis/covise.
5. J. Ahrens, C. Law, W. Schroeder, K. Martin, and Michael Papka. A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. http://www.acl.lanl.gov/Viz/papers/pvtk/pvtkpreprint/.
6. J. Allard, V. Gouranton, E. Melin, and B. Raffin. Parallelizing pre-rendering computations on a net juggler PC cluster. In *Immersive Projection Technology Symposium*, Orlando, USA, March 2002.
7. C. Cruz-Neira, D. J. Sandin, T. A. DeFanti, R. V. Kenyon, and J. C. Hart. The Cave Audio VIsual Experience Automatic Virtual Environement. *Communication of the ACM*, 35(6):64–72, 1992.
8. A. Denis, C. Pérez, and T. Priol. Padicotm: An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 2003.
9. E. Frécon and M. Stenius. Dive: A scalable network architecture for distributed virtual environments. *Distributed Systems Engineering Journa*, 5:91–100, 1998.
10. N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.
11. K. Watsen and M. Zyda. Bamboo - a protable system for dynamically extensible, real-time, networked, virtual environments. In *IEEE Virtual Reality Annual Internationnal Symposium*, Georgia, USA, 1998.