CWI Quarterly Volume 7 (2) 1994, pp. 175 – 185

Shared Data Spaces for Distributed Computing and Parallelism in Scientific Visualization Systems

Michel Grave

ONERA/DI

29, Avenue de la Division Leclerc
F-92322 Châtillon Cedex, France

In a first part, this paper reviews the architecture principles of some visualization systems from the family of so-called application builders, focusing on their ability to be run in distributed environments. In a second part some points concerning the efficiency of distributed execution, and strategies for executing on parallel systems are discussed. Finally the main lines of a basic layer for handling data sharing and processes synchronisation, presently developed at ONERA, are presented.

1. Introduction

Scientific Visualization requires more and more complex processing, usually obtained by assembling several elementary operations. This complexity is often hidden when turnkey systems are used, but it is always present. Since requirements evolve quickly, a new generation of systems, called Application Builders has emerged during the last years, and Visualization applications in Scientific Computing are more and more often built today as interconnections of several cooperating processes. In addition, applications are more and more used in distributed environments, where different types of servers (for database handling, graphics, or cpu-intensive computing) are connected, and processes are running each on the best suited servers, or on the more available ones. This also applies for parallel computer architectures, and in that case, some of the processes can also be designed so that each of them can exploit parallelism. Finally, with the progresses in high-speed networking there is an increasing demand for Computer-Supported Cooperative Working (CSCW) applications,

in which several persons in different locations, could work at the same time on the same distributed application. All these factors have led several groups to study software architectures for such environments. In the first part of this paper, elements of the organisation of a few existing visualization systems are presented, mainly focusing on the communication and synchronisation features. The second part of this paper presents some requirements induced by the need to run visualization applications on parallel systems, and some of the problems to be solved for building CSCW applications. Finally, an approach for designing visualization systems is presented, and the basic kernel of a communication system currently being developed at ONERA and allowing the implementation of such new systems is presented.

2. Architecture of existing application builders

A general organisation of modular visualization systems based on the data-flow paradigm has been described by Haber and McNabb [9]. It shows clearly how different components are interconnected in a pipeline transforming step by step data into images, and where users interact with the system. This scheme is very close to what has been implemented in many systems. However, the data handling mechanism or the data transmission between modules, as well as the way the overall process is controlled does not clearly appear, and it is in fact in these features that real systems differ. The way these mechanisms are designed has a great influence on the ability of systems to handle distribution or parallelism.

AVS In AVS [12], a unique process, named Flow Executive, controls the flow of data and the execution of the different modules. It has a description of all of them, and decides when a module must be executed, based on the availability of data it needs. It also centralises user's interactions with the modules. In principle, each module is executed as a process, and communicates through classical mechanisms in UNIX environments. In reality, shared memory is used, when possible, between modules running on the same computer, in order to reduce data replication, and several modules are grouped in single programs in order to reduce the number of running processes, as well as the data communication overhead. In general, a module has a predefined number of typed input and output ports, and an input port can receive information from only one output port of an other module. This is however not the case for some specific modules, like for example the geometry viewer which can receive geometrical objects from many different sources and combine them into a unique scene, in order to produce a unique image. It is also possible to divide a pipeline at any stage, and replicate the output of a module, in order to feed several instances downstream.

Iris Explorer In Iris Explorer [11], there is also a centralised process, named Global Communications server (GC) that centralises user's interactions, and initialises the different modules to be executed, but it does not directly su-

pervise the execution of the modules. On each computer involved the GC communicates with a Local Communications server (LC), that will start modules, establish connections between them, or manage shared memory. When modules are started there is no other control on their execution. There is no decision of execution outside the modules themselves. Modules start processing when they have received enough data on their input ports. The fact that this decision is taken locally in each module is visible to the users, since they have the possibility to directly "fire" modules individually, which is not possible with AVS. In the version 2 of Iris Explorer, mechanisms for handling parallel data flows have been introduced, and fan-in of input data of modules is allowed. Several instances of a pipeline can then run in parallel, with a merging occurring not necessarily at the level of the display module. It is also possible to build modules sending out data as soon as subsets of them are ready. This allows a module to start processing only with a part of its input data, and not only at completion of it predecessors.

Khoros KHOROS [14], has many differences with AVS and Iris Explorer although it also has a visual programming interface (named Cantata). Modules are autonomous entities that can be utilised individually, as "shell" commands. Networks created with Cantata can be saved in files as "shell scripts", and run afterwards under no other environment than the operating system itself. Communication between modules can be implemented in different ways (regular files, shared memory or UNIX "sockets"), and different modules can be run on different computers in a network. In that case a communication managing process executes on each computer involved, which manages the transfers between modules. Cantata can also be requested to detect parallel branches of networks of modules, and arrange their parallel execution.

3. REQUIREMENTS FOR NEXT GENERATION OF SYSTEMS

3.1. Data Circulation

Todays computational environments include very powerful supercomputers, and the amounts of data produced by a single simulation can be very large. For example, in Computational Fluid Dynamics, meshes containing one million grid points are common, and in unsteady flows simulation, thousands of time steps need to be computed, each producing Megabytes of information. This means that the amount of data produced are measured in Gigabytes. It is then clear that if all that information circulates between nodes of parallel computers or on networks, communication problems can easily become the real bottleneck, diminishing seriously the benefit of using high performance computers. In addition, data replication, like for example when it has to circulate between processes can also lead to serious limitations in the use of visualization systems, if it makes them usable only on small size problems. Most of the systems address today this second point, mainly by using shared memory mechanisms, when possible, but since modules are interconnected and assigned to CPUs before execution, no optimisation of data circulation can be performed

at run time, depending on data location and modules usage. Low level data handling mechanisms have to be designed for that purpose.

3.2. Parallelism

As seen above, in many cases a visualization system can be considered as a pipeline of processes progressively transforming simulation or experimental data into images. The same idea also applies in the rendering stage itself, at least in systems based on the model where objects are projected from the application's space onto the image space [1], which is the case in most of those not using the ray-tracing method. A first obvious way of taking advantage of parallel architectures consists in assigning different stages of the pipeline to different processors, processing the data concurrently as they flow through them. This is usually referred to as the Procedural Parallelism scheme. In this case, parallelism can be achieved only if there is a rather continuous flow of data. This can happen when for example a time dependent phenomenon is processed, or when the input data can be split into parts that can be sent to the pipeline each after the other. A second method consists in using multiple copies of the visualization pipeline, each processing a part of the data. This is the scheme named Data Parallelism. In this second approach, a merging stage has to be included at a certain level, in order to produce a unique common image at the end. There are different criteria for splitting data into different subset and for distributing them between pipelines [13]. The choice of an approach depends very much on the context in which the visualization system is used, and on the type of processing it has to perform. Of course, Procedural and Data parallelism can be combined. Since processing can be distributed, an important issue is the design of a communication system between processes. Figure 1 illustrates different architectures. On the left part figure, the different pipelines are independent from each other, and merging appears only at the end, when the different partial images are combined [1]. On the right one, merging happens after each stage of the pipeline (merging need not necessarily happen physically at one place, and in some cases can be implemented just by broadcasting data).

All the systems analysed above can easily handle procedural parallelism, since the different steps of a pipeline can be distributed over different processors. However, except in the case of Iris Explorer 2 in which some flow partitioning functionalities are available, parallel execution of the different steps only occurs when there is a continuous flow of data feeding the pipeline, since each step waits for having all the data it has to receive from the previous one before starting to execute. When the data flow is not continuous, the data produced at each step should be split in smaller parts, and the modules receiving these data should be able to start to execute as soon as a part is available.

[1] An image can be +partial; in many different ways. For example, each of them can cover a limited portion of the screen, or each of them can represent only a subset of the initial data, and the merging techniques are then different, but their study is out of the scope of this paper

FIGURE 1. Examples of parallelisation strategies

For Data Parallelism, present systems are able to handle it by replicating parts of the pipeline, but only in the scheme presented on the left part of Figure 1. This means in particular that the numbers of modules assigned to each step will be identical, and that load balancing will be impossible between the different steps. In addition since connections are fixed, a module can only send data to a predefined following one, even if it is already running while an other one of that stage is idle. This is another difficulty for easily handling load balancing.

3.3. Cooperative working

For Cooperative Working there are two main needs. First, it must be possible to present at different locations an image of the same simulation, and in most cases, users would like to have the same image. This implies not only that the output of a visualization system must be replicable, but also that there exists a synchronisation mechanism, especially in the case of animated images, in order to ensure that the different users see the same thing at the same time. An other important feature needed is synchronisation of the user interfaces, including feedback of actions at one location on the interfaces of all others. In that case handling of distributed computing at the level of the visualization pipeline is not sufficient.

Current visualization systems can satisfy the first requirement, since rendering modules can either be replicated, or can at least generate images on different screens, and they have synchronisation mechanisms that can be explicitly added between modules. However, it is not possible for a new user to enter a session in which others are already working, without requiring a modification of the network of modules, performed by the person responsible of it. The second requirement is much more difficult to fulfil, and not implemented properly in those systems. This issue will also not be addressed in the rest of this paper.

FIGURE 2. The Shared Data Space approach

In this approach, a module has the possibility to produce shared data objects

without knowing if another module is ready to use them. Further, if a data object is not explicitly destroyed, it can persist even if no module is connected to it, until the data space is destroyed itself. In addition to this idea of Shared Data Space, it should be possible to partition date objects, which means that a module should be able to access only a part of an object, without knowing what other modules access other parts of it.

Since there is no explicit connection between modules, any number of modules can access an object. As mentioned in the requirements of CSCW applications, a user can connect a new module to an object as soon as it gets its name, without asking anyone else to do the work. However, it is clear from Figure 2 that the SDS approach can handle synchronisation between operators only at the data level, and that synchronisation at the user level must be handled by other mechanisms.

For building a system based on such partitioned shared data objects a low level mechanism for handling communication and data sharing is needed, and for that, a basic library named DS2 has been designed at ONERA. Such a layer must hide the explicit sending of messages between tasks, even if this is used internally for communication, through a tool like PVM for example [4]. Even if other similar work has been done in other places, (see [7] for an analysis of some of them), a new development was started, mainly because the software has to be available on a network including workstations, a CRAY YMP, and an Intel PARAGON, without making any change inside the operating systems (unlike most of the other work we know). However, porting of the DS2 interface as it is presented below but also in further versions, on another software layer will be considered if an appropriate such basic layer is found. Upper layers of the system, including libraries of more applications specific objects are being developed on top of it.

5. The DS2 library

5.1. Definitions

In DS2, objects are collections of Pieces, themselves being arrays of homogeneous basic type elements (such as integers, characters). Objects are shared and acted upon by Modules, the unit of access to objects being the piece. Objects belong to a Shared Data Space (SDS), common to all the modules involved in a given application. Modules can access objects by sending requests to a Data Manager (DM). Of course, the object-oriented approach hides this communication to the user of DS2, who is not aware of the existence of a DM: all he manipulates are SDSes, objects and pieces. The physical space used by the SDS can be distributed over the different machines on which modules execute. In that case, the data manager itself is distributed, and consists of several cooperating Local Data Managers.

Figure 3 shows the general principles of DS2. Four processes (P1–P4) are shown distributed on 2 computers. P2 and P4 use different pieces of a common object (`Object_1`), and pointers to those pieces have been provided by the Local Data Managers, which handled requests from the processes, and coop-

FIGURE 3. General Principles of DS2

5.2. Mechanisms

Before gaining access to shared objects, a module has to enter an SDS, referenced by a name. When it no longer needs to access objects, it leaves the space. For that purpose, a module uses requests of the following type (C++ -like syntax has been used in the examples):

```
SharedDataSpace SDS1(data-space-name);
SDS1.EnterSpace();
-- reference objects belonging to SDS1 --
SDS1.LeaveSpace();
```

In order to access an object, a module has to open it first, possibly creating it, in which case a structure has to be specified in terms of pieces (number, sizes, basic types). Once processing an object is completed, a close operation has to be issued. The corresponding requests are the following:

```
Object SHOB1(SDS1, object-name [,pieces-structure]);
SHOB1.OpenObject();
-- access pieces of SHOB1 --
SHOB1.CloseObject(destruction);
```

Between OpenObject and CloseObject calls, the module can access pieces of the object. Pieces are accessed by only one module at a time (mutual exclusion of modules). The system takes care of the validity of this access. Therefore, a module first requests access to the piece, and finally sends another request once it has completed the operations it had to perform on it, so that other modules can access the piece again. When a piece has been acquired, the data it holds are available to the module by the use of a mere pointer. Therefore, it can perform any action on it, except re-sizing the area or moving the pointer. The following requests are used:

```
Piece PIEC1(SHOB1,piece-number);
BasicType *data = PIEC1.GetPiece(mode [,testflag][,tag-range]);
-- work on the data: data[xx] = yy; --
PIEC1.ReleasePiece(destruction [, newflag][, newtag]);
```

The system provides the user with multiple ways of selecting a piece from an object. First of all, it is possible to specify a mode of access, as blocking or non-blocking. This means that if the piece of the specified number is not immediately available (because it is held by another module or it does not satisfy the criteria described below), a call to GetPiece with a blocking mode will wait for the piece to become available, whereas a non-blocking mode will cause an unsuccessful return. Besides, two values are attached to a piece: a flag (group of bits) and a tag (numerical value). The piece will be made available to the module on a Get-Piece call only if its tag is in the tag-range and its flag verifies the formula

```
(testflag && flag)== TRUE
```

Note that the use of testflag and tag-range is optional.

Two types of flags are defined: free flags, which meaning is chosen by the user, and special flags, describing the status of a piece on a system point of view (for e.g. created).

The piece selection mechanism has been defined so that it allows the handling of time-dependent objects, for which tags can be used as computational time-step information. Besides, it is also possible to realise non-deterministic access to pieces, by iterative non blocking requests until finding a piece that meets a condition (flag). Complex requests can be expressed, like for example: I (module) want to access any piece of object O that is available for processing. By this means, automatic load balancing is achieved. Finally, the implementation of application synchronisation (triggering of modules) can be done without any external mechanism, by the use of special flags and blocking facilities.

6. Conclusions and further developments

The new visualization tools under development have been designed for allowing scientists to analyse results produced on distributed memory parallel computers. A classical way of parallelising a code in such environments consist in

decomposing the computational domain into pieces with data exchanges between the processes working on subdomains, in order to manage consistency at borders, and to propagate data if necessary. With post-processing tools like iso-surface computation or cutting planes, Data parallelism is easy to implement (since they can also work independently on subdomains) either by linking them directly to the simulation in the same module, or by assigning them to a group of processors. With the SDS approach, there is no need to allocate as many processors to the post-processing as there are for the simulation, and this is interesting, especially when the simulation is more cpu intensive than the post-processing. The same applies to other stages of visualization, like rendering.

First experiments, with iso-surface computation and parallel rendering with pixel merging [15] have been performed, and showed a good behaviour of the program, with a rather fair load balancing between processors. However, more important analysis has still to be done, with more CPU intensive computations, in order to limit the influence on the data transfer tools on the results.

Since modules are not explicitly interconnected, each of them can take parts of the data that are available, independently of their origin. When advanced access mechanisms like "give me the first piece available" will be available, the Local Data Manager will be able to decide to give locally available pieces in priority, in order to reduce data transfers. At present, such advanced access mechanisms are not implemented inside DS2, and have to be programmed on top of it. This causes some overhead of communication between the application program and the data manager, and in addition, this implies useless computing. In a next version, a selection of mechanisms will be directly implemented a the lower level.

A large part of the work presented here has been made possible by ECC Funding, through RACE 2031 Project. This is especially true for the DS2 system, which development has been in large parts realised at ONERA by Sylvain Causse and Frederic Juaneda. All other members of the Graphics Group also contribute to different parts of the experiments, or to some related developments. Finally, Robert Van Liere from CWI provided valuable remarks, and very useful additional information during the writing of this paper.

References

1. Akeley K. (1989). The Silicon Graphics 4D/240GTX superworkstation, *IEEE Computer Graphics and Applications*, **9**(4).
2. Grave M. (1993). Distributed Visualization in Flow Simulations, *Computers and Graphics*, Vol **17** (1).
3. Lucas B., Abram G.D., Collins N.S., Epstein D.A., Gresh D.L., McAuliffe K.P. (1992). An Architecture for a Scientific Visualization System, *Proceedings IEEE Visualization'92 Conference*, Boston, Ma.
4. Beguelin, A., Dongarra, J., Geist, G.A., Manchek, R., Sunderam, V.S. (1992). *A User's Guide to PVM Parallel Virtual Machine*,

Oak Ridge National Laboratory Internal Report.
5. Butler, D.M., Pendley M.H. (1989). The Visualization Management System Approach to Visualization in Scientific Computing, *Computers in Physics*, pp. 40-44.
6. Causse, S., Hollard, M., Grave, M. (1992). *Experimenting Distributed Visualization with a PVM Based System*, ONERA Internal Report.
7. Causse, S., Juaneda, F., Grave M. (1994). Partitioned Objects Sharing for Visualization in Distributed Environments, In *Frontiers in Data Visualization*, Rosenblum et al. (eds.) Academic Press.
8. Duclos, A.M., Grave M. (1993). Reference Models and Formal Specification for Scientific Visualization, In *Scientific Visualization: Advanced Software Techniques*: P. Palamidese (ed.) Hellis Horwood.
9. Haber, R.B., McNabb, D.A. (1990). Visualization Idioms: A Conceptual Model for Scientific Visualization Systems, In *Visualization in Scientific Computing:* Nielson et al. (eds.) IEEE Computer Society Press.
10. Schroeder, W.J., Lorensen, W.E., Montanaro, G.D., Volpe, C.R. (1992). VISAGE: An Object-Oriented Scientific Visualization System, *Proceedings IEEE Visualization'92 Conference*, Boston, Ma.
11. Iris Explorer (1991). Silicon Graphics Technical report.
12. Upson, C., Faulbaher, T., Kamins, D, Laidlaw D., Schlegel, D., Vroom, J., Gurwitz, R., van Dam, A. (1989). The Application Visualization System: A Computational Environment fo Scientific Visualization, *IEEE Computer Graphics and Applications*, Vol **9**(4).
13. Whitman S. (1992). *Multiprocessor Methods for Computer Graphics*, Jones and Bartlett, Boston.
14. The Khoros Group (1992). *Khoros Manual*, Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque.
15. Cox, M., Hanrahan, P. (1993). Pixel Merging for Object-Parallel Rendering: A Distributed Snooping Algorithm, *Proceedings of IEEE 1993 Parallel Rendering Symposium*, San Jose 1993, Published by ACM Siggraph.