A Hierarchical Programming Model for Large Parallel Interactive Applications

Jean-Denis Lesage and Bruno Raffin

INRIA, Grenoble Informatics Laboratory, France

Abstract. This paper focuses on parallel interactive applications ranging from scientific visualization, to virtual reality or computational steering. Interactivity makes them particular on three main aspects: they are endlessly iterative, use advanced I/O devices, and must perform under strong performance constraints (latency, refresh rate). In this paper, we propose an application description language based on a data flow and hierarchical component model to cope with the complexity of parallel interactive applications. It enables us to define highly generic components, enforcing the application maintainability and portability. An implementation on top of the FlowVR middleware is presented.

1 Introduction

An interactive application is an endless iterative process involving a user user interacting with a program through input and output devices. It is often referred to as a "human in the loop simulation". Today, an emerging class of interactive applications intends to associate virtual reality, scientific visualization, simulation and application steering. It leads to very complex applications coupling advanced I/O devices, large data sets, various parallel codes. To be interactive, they must perform under strong performance constraints, often measured in terms of latency and refresh rate. Examples of such applications are described in [1,2,3]. In this paper we focus on two issues faced when designing such application:

- Software engineering issues where multiple pieces of codes (simulation codes, graphics rendering codes, device drivers, etc.), developed by different persons, during different periods of time, have to be integrated in the same framework to properly work together.
- Hardware performance limitations bypassed by multiplying the units available (disks, CPUs, GPUs, cameras, video projectors, etc.), but introducing at the same time extra complexity. In particular it often requires to introduce parallel algorithms and data redistribution strategies, that should be generic enough to minimize human intervention when the target execution platform changes.

One challenge is to ensure the genericity and modularity of the application.

Scientific visualization applications are often developed with Modular Visualization Environments (MVE) like OpenDX [4], Iris Explorer [5] or VTK [6].

These environments are usually based on a data flow model where processing tasks receive data and generate new ones. Most of MVEs support parallel executions. An application is basically a list of filters applied to the data set before rendering. The first natural level of parallelism is to distribute the different steps of the filter pipeline on different machines. Because the data set is read only, the pipeline can easily be duplicated and executed in parallel on sub parts of the data set [7]. Advanced parallel rendering algorithms exist, based for instance on specific parallel data structures and dynamic work balancing schemes. In this case they are implemented on their own, usually using classical parallel programming languages, because MVEs do not provide the necessary constructs.

In virtual reality, to ensure an efficient data redistribution between parallel algorithms that may run at different and varying frequencies, complex coupling schemes associating data re-sampling and collective communications are required. Dedicated environments like FlowVR [8], OpenMask [9] or COVISE [10] propose different approaches to support such features. However, the resulting application code tends to be difficult to maintained when reaching a certain size. Connectivity between processing tasks (communication channels) are expressed by direct links between the corresponding elements: it requires the concerned elements be directly visible one for each other, preventing attempts to strongly structure the code by encapsulating patterns in methods or functions.

Component models, like CCA (Common Component Architecture) or CCM (Corba Component Model), provide ADLs for the description of distributed applications. SCIRun, an environment dedicated to scientific visualization, is based on the CCA model [11]. Some extensions intend to enforce the support of parallel components and the associated coupling patterns [12]. But these models suffer from the same limitations as the systems mentioned earlier (FlowVR, Covise) regarding the modularity of parallel component coupling. Fractal [13] is a truly hierarchical component model. We are aware of one implementation of Fractal for parallel (grid) applications: ProActive [14]. A ProActive composite component can be a parallel component. But redistribution patterns are coded into the ports of the parallel components. A pattern cannot be modified without modifying the component, limiting application modularity. In this paper we propose to encode coupling patterns as standalone fractal components with a connectivity model between primitive components (processing tasks) that does impair this modularity.

We propose an application description language, called architecture description language or ADL following the uses of the component community, based on a data flow and hierarchical component model. We focus on interactive applications, instead of a general purpose language, relying mainly on their iterative nature, to restrain the domain of the language.

To enforce the genericity of the described application, components defer introspection and auto-configuration processes to controllers. A controller is local to a given component, but it may get extra data consulting the state of the neighbor components or through external data repositories. These controllers, that can generate new components for instance, are called recursively and repeatedly in a traverse process until reaching a fixed point. Traverse either leads to an error (missing data impair the traverse completion) or a success. This approach enables to define highly generic components, enforcing the application maintainability and portability. In particular, we can define arbitrarily complex and adaptive data redistribution components, for instance mixing collective communications and re-sampling. This is an important feature for interactive applications where these coupling mechanisms play an important role to enforce interactivity.

Section 2 presents our hierarchical model. Section 3 details our implementation on top of the FlowVR [15] middleware with a focus on the traverse process. Section 4 concludes the paper.

2 Programming Model

In this section, we describe our hierarchical component model inspired by Fractal[13] for large parallel interactive applications. Fractal is a component model based on a component hierarchy. This model enables to encapsulate components into high-level components. This encapsulation enforces reusability and modularity. We will also present another feature, named controllers, inspired by Fractal too. Theses objects enables dynamic reconfiguration and component introspection.

2.1 Components

A component has an interface defined by a set of ports. We distinguish two kinds of components:

Primitive components. A primitive component contains a loop. At each iteration, the component reads data from its input ports. It writes computation results on its output ports.

Composite components. A composite component contains other components (composite or primitive). We impose a strong encapsulation paradigm: a component cannot be directly contained into two parent components.

2.2 Port Typing

There are two types of ports: input and output ports. The input port receives data and output port sends data. We do not impose a strong typing. We simply require the input and output correspondence. Nevertheless, depending on the needs, the port typing can be extended. We plan a stronger typing based on the data type exchanged by the ports.

2.3 Example

Throughout this paper, we use a simple example (Fig. 1). It shows the classical structure of an interactive application. The goal of this application is to compute prime numbers and from these numbers compute a 3D image. The image is updated each time a new prime number is computed. A keyboard enables the user to change his point of view on the image.

The *Computes* composite component is a parallel component programed with MPI. It spawns n processes *Computes/0,...*, *Computes/n-1* seen as primitive components of *Computes.* Notice that n is only known once the application as been configured for an execution on a particular target machine.

The composite component *Renderer* is divided in two main parts (Fig. 1.b). The first one, *Visu* makes the rendering on a display. This display contains several screens. For each screen, a rendering process must be instanced. The *Visu* component contains all these rendering processes. The second one is the component *Capture.* It gets key events from user and sends them to the *Visu* component.

Two coupling components are dedicated to communication (Components *Connect* and *GreedyConnect*). The *Connect* component transmits data from *Computes* component to *Renderer* component. The *Connect* component contains a communication pattern. The *GreedyConnect* resamples messages from *Capture* for *Visu.*

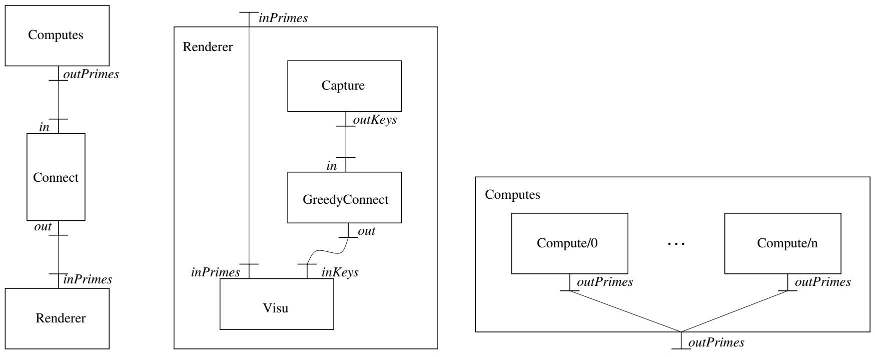

Fig. 1. a) The application : a composite component *Computes* generates primes numbers. They go through the *Connect* component to the renderer. **b)** The *Renderer* composite component contains two components *Render* and *Capture.* **c)** The *Computes* composite component is a MPI parallel component.

In this example, *Computes*, *Renderer* are examples of composite components. *Compute/0* and *Visu* are primitive objects. *outPrimes* from *Computes* is an output port. *inPrimes* from *Renderer* is an input port.

2.4 Links

Links are used to model data flows between ports. We distinguish two kinds of links. The parent link joins a port from a component to one of his children's port. The extremities of a parent link must have the same type. For example (Fig. 1.b), the *outPrimes* port on *computes* component has the same type as all its children (i.e. output port).

The second kind of links are called sibling links. They go from a component to an another. We assume that an object cannot share data with an another object without using a connection. So a sibling link must join an input port to an output port. Due to the strict encapsulation paradigm, a sibling link cannot directly connect two components that are not brothers (child of the same parent). A chain of sibling and parent links must be used to connect two non brother components.

The link between *outPrimes* port and *in* port of *Connect* in our example is a valid sibling link.

2.5 Parallel Components

A composite component can be a container for parallel application. For example, *Computes* is a parallel MPI code spawning when launched several processes, each one being a primite compenent. These primitive components are linked to the same parent port (Fig. 1.c). This kind of structure can express the data and task parallelism for instance. Notice that the number of processes spawned depends on the instanciation of the application for a given target architecture. The *Computes* component has a mandatory parameter that defines the number of MPI processes. It must be set to know the number of primitive components it contains. Such level of dynamicity is classical for parallel components.

A composite component can also encapsulate a pipeline. Each stage of the pipeline can be contained into a component. A sibling link from a component to another will make the transition from one stage to an other. Thanks to components reusability, we can also duplicate a pipeline by building a composite component containing various parallel pipelines.

2.6 Communications and Redistribution Patterns

Communication between parallel components have a huge impact on application performance. They need to be customisable and modular. A communication component is simply a component encapsulating a generic redistribution pattern. The simplest one is just a link transferring data from one output port of a primitive component to one input port of a primitive component.

In our example, a connection schema is implemented in the *MergeThenTree* component (Fig. 2). This component has a different implementation following the number of primitive components *Compute* and *Renderer* will spawn. Unlike the parallel components, user does not have to set the parameters of these dynamic components. These components get their mandatory parameters from their neighbors.

The simplest communication pattern is a simple connection. But it could be a merge tree and a broadcast tree with different arities. The order of merged messages could be customized. Communications may resample messages. Components can contain filters that operate on messages or enforce synchronizations between a set of components. Typically, filters are used to resample messages. Several filters can be synchronized to perform a coherent sampling, i.e. ensure they sample messages issued at the same logicial time.

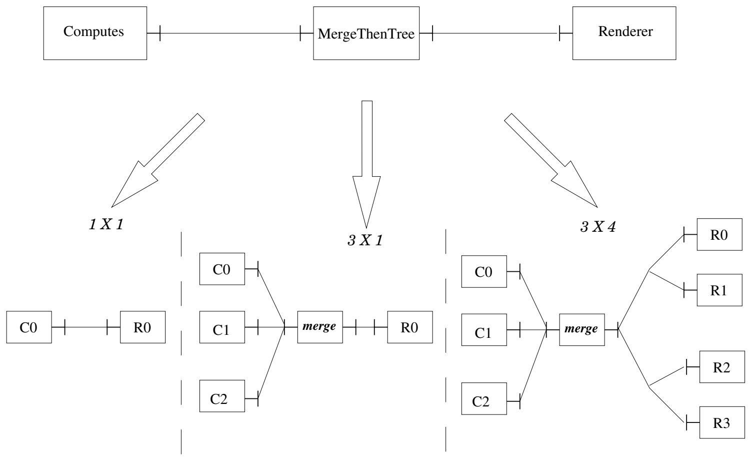

Fig. 2. Parallel compute component send data through a connection component to a parallel renderer. According to N and M parameters, a communication schemes is generated.

Some communication component parameters depends from the state of the neighbor components. In the example (Fig. 2), the shape of the communication pattern depends on the number of *Compute* and *Render* components connected at its extremities. For this reason, these components can create dependence relations between components.

Our model eases the development of generic communication patterns outside the context of an application. An implementation of this model can be associated for example with a library of $N \times M$ data redistribution components. Components provide modularity. A user is able to change a communication pattern for his application and see the impact on performance.

2.7 Controllers

Controllers are used for the configuration and the construction of dynamic components. Parallel and communication components are often dynamic. Parallel components can have a parameter to set the number of computational units (i.e. the degree of parallelism). Communication component parameters often depend on their neighbor states.

There are two types of controllers:

– Controllers getting data from a component (introspection)
– Controllers setting parameters (configuration)

A controller is associated to a component and a method. A main controller must be implemented for all new composite components. We named this controller *execute*. This controller creates children components. For example, in the *Computes* component, the *execute* controller creates all *Compute/i* primitive components and constructs the parent links.

The controller can lead to an exception if a mandatory parameter can not be set. For example, the communication pattern in *MergeThenTree* component can not be built if the number of *Compute* primitive components is not set (Sec. 2.6). In this case, the controller throws an exception.

2.8 Traverse Algorithm

A controller always acts locally on a component, but some actions must be executed globally on the entire application. For example, building a view of an application, a graph for instance, requires to call a view builder controller on each component. Data dependences may impose a given execution order on controllers. For instance some controllers, like *execute*, dynamically create new components. Connection components often have to be constructed after their neighbors. Most controllers have to be executed at most once by component to obtain the correct result. Consequently, the iteration algorithm is an important issue in our model. We named this algorithm the traverse algorithm. This algorithm must respect following constraints :

– Top-down iteration : a controller must be applied on the parent compoenent before to be applied to its children.
– A controller must be applied on a component at most once.
– Constraints on the execution order must be respected.
– The traverse algorithm stops if the controller cannot be called on any remaining component.

In the implementation section (Sec. 3), we will present an implementation of the traverse algorithm and some controllers.

2.9 Interactions with Traverse Algorithm

Due to traverse properties, when a traverse fails, the controller leads to an exception on the remaining components. Most programming languages enables exception catching. If exceptions provide enough details, user can know why controller cannot execute on these components. Often, a parameter is missing. In order to finish the traverse, the simplest solution is to ask the user to correctly set this parameter.

Indeed, the exception raised by component can be printed. User can give an appropriate answer to the algorithm. In case of an application with thousand components, we have made the interaction simpler with the use of a comma-separated-value file. This file can be read by a spreadsheet program. User can fill an automatic generated file with all parameters to be set with his favorite spreadsheet program.

Traverse algorithm can also interact with an other program. For example, for mapping issues, the choice of machines where a process must be mapped is a complex problem for a human. Mapping has a huge impact on performance like refresh rate or latency. A mapping program using a hardware description file could calculate a mapping solution efficiently.

This implementation could give the possibility to make dynamic reconfigurations. During execution, the entire application could be stopped. The system will proceed to a new instantiation of the application. The traverse algorithm can now use the log file to resolve exception raised during the traverse algorithm. This traverse algorithm could be done in parallel with the execution. A mapping algorithm could adapt the application to resource capacities at execution-time.

3 Implementation

3.1 Greedy Traverse Algorithm

The main issue in the model implementation is the traverse algorithm. This algorithm must iterate on components and respect several constraints. (Sect. 2.8). This algorithm must find a consistant order considering all constraints for the iteration through the components.

We make the traverse via a greedy algorithm. This algorithm manages a queue of non-executed components. For each components in this queue, the algorithm tries to execute the associated controller. If the controller was successfully executed, then all of its children are pushed in the queue. Otherwise, the algorithm makes a rollback operation on the component and push it at the end of the queue.

The traverse is done when the queue becomes empty. If the algorithm can not change the queue state, then a fixed point is reached. No new evolution can be performed to component states. To respect traverse properties, the algorithm must stop and signal its fail.

With this implementation of the traverse algorithm, there is no need to express constraints on components. But, this implementation may lead to unnecessary controller calls. We provide bounds on the number of controller calls for this algorithm:

Proposition 1. *Let N_{comp} the maximum number of composite components in an application. The maximum (resp. minimum) number of call of controllers performed by greedy traverse algorithm is $O(N_{comp}^2)$ (resp. $O(N_{comp})$).*

For sake of conciseness, the proof is omitted. The proof outline is to show that a controller can be called at most N_{comp} times by component.

The complexity of our algorithm is upper bounded by $O(N_{comp}^2)$ but we do not have to compute an order of iterations between components considering all constraints. The greedy traverse algorithm tries to iterate on components until it finds an acceptable order. Theses tries can lead to extra costs but computation of an acceptable order may involve complex algorithms. Our solution is a good tradeoff between scalability and complexity of the implementation.

3.2 Implementation on the Top of the FlowVR Middleware

We have built our model on the top of FlowVR [15,8]. This middleware is used to construct large parallel interactive applications. It eases the development of virtual reality applications that associates scientific visualization and simultations. For instance we developed applications involving a real time 3D modeling algorithm using data from a camera network, parallel simulations and multi-projector visualization with FlowVR.

FlowVR is based on four types of primitive components [8]:

Modules. User defined components. They make all computational issues in the application.
Connections. They transmit data from an output port to an input port.
Filters. They make treatments on messages. They are involved in communication schemes.
Synchronizers. They implement synchronization policies between components.

All these kinds of components have been implemented using our model. The second step of the implementation was to construct controllers dedicated to the middleware. The main controller specially developed for FlowVR builds a XML description of the application. When launching an application, FlowVR distributes order to FLowVR dameons running on the nodes of the target machines to load plugins, configure communications schemes, etc. These orders are describeexgtracted from an XML desctiption of the application. For each primitive component, we have created the controller that builds this XML description. Composite components just recusrively link children description into the XML tree.

All examples from the FlowVR suite have been redeveloped with the hierarchical model introduced in this paper. The example used in this paper (Fig. 1) was inspired from one of these applications. Mocing to the hierarchical model improved application modularity. For instance, an application can now be imported as a composite component in larger applications.

4 Conclusion

We presented an ADL for interactive applications based on the fractal component model. Our main goal was to ensure a high level of modularity for large applications involving parallel components and advanced coupling schemes. Configuration of components is deferred to controllers. It enables us to separate some aspects of a component from its core functional nature. An application is then configured by calling the controllers in a traverse process. This ADL has been implemented and validated on top of the FlowVR middleware. We expect to integrate it in the FlowVR distribution soon.

References

1. Tu, T., Yu, H., Ramirez-Guzman, L., Bielak, J., Ghattas, O., Ma, K.-L., O'Hallaron, D.R.: From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In: Super Computing (2006)
2. Strehlke, K., Moere, A.V., Gross, O.S.M., Wuermlin, S., Naef, M., Lamboray, E., Spagno, C., Kunz, A., Koller-Meier, E., Svoboda, T., Gool, L.V., Lang, K.S.S., Moere, A.V., Staadt, O.: Blue-C: A Spatially Immersive Display and 3D Video Portal for Telepresence. In: Proceedings of ACM SIGGRAPH 03, San Diego (2003)
3. GrImage: website, http://www.inrialpes.fr/grimage/
4. Lucas, B., Abram, G.D., Collins, N.S., Epstein, D.A., Gresh, D.L., McAuliffe, K.P.: An architecture for a scientific visualization system. In: VIS '92: Proceedings of the 3rd conference on Visualization '92, Los Alamitos, CA, USA, pp. 107–114. IEEE Computer Society Press, Los Alamitos (1992)
5. Foulser, D.: IRIS Explorer: a framework for investigation. SIGGRAPH Comput. Graph. 29(2), 13–16 (1995)
6. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd edn. Kitware, Inc. (2003)
7. Ahrens, J., Law, C., Schroeder, W., Martin, K., Papka, M.: A parallel approach for efficiently visualizing extremely large, Time-varying Datasets. Technical report, Los Alamos National Laboratory (2000)
8. Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., Robert, S.: FlowVR: a Middleware for Large Scale Virtual Reality Applications. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, Springer, Heidelberg (2004)
9. Margery, D., Arnaldi, B., Chauffaut, A., Donikian, S., Duval, T.: OpenMASK: Multi-Threaded or Modular Animation and Simulation Kernel or Kit: a General Introduction. In: Richir, S., Richard, P., Taravel, B. (eds.) VRIC 2002 Proceedings, pp. 101–110 (2002)
10. Wierse, A., Lang, U., Rhle, R.: Architectures of Distributed Visualization Systems and their Enhancements. In: Eurographics Workshop on Visualization in Scientific Computing, Abingdon (1993)
11. Zhang, K., Damevski, V., Venkatachalapathy, S.G., Parker, S.G.: SCIRun2: A CCA Framework for High Performance Computing. hips 00, 72–79 (2004)
12. Denis, A., Pérez, C., Priol, T.: PadicoTM: an open integration framework for communication middleware and runtimes. Future Generation Comp. Syst. 19(4), 575–585 (2003)
13. Bruneton, E., Coupaye, T.: Stefani, J.: The Fractal Component Model. Technical report, ObjectWeb Consortium (February 2004)
14. Baude, F., Caromel, D., Morel, M.: From distributed objects to hierarchical grid components. In: Meersman, R., Tari, Z., Schmidt, D.C. (eds.) CoopIS/DOA/ODBASE 2003. LNCS, vol. 2888, pp. 1226–1242. Springer, Heidelberg (2003)
15. Arcila, T., Allard, J., Ménier, C., Boyer, E., Raffin, B.: FlowVR: A Framework For Distributed Virtual Reality Applications. In: AFRV, Rocquencourt (November 2006)