A Hierarchical Component Model for Large Parallel Interactive Applications

Jean-Denis Lesage and Bruno Raffin

INRIA
Laboratoire d'Informatique de Grenoble (LIG)
Email: jean-denis.lesage@imag.fr and bruno.raffin@imag.fr

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). A data flow graph is a common approach to describe such applications. Edges represent data streams while vertices are nodes processing incoming data streams and producing new data streams. When applications become large this approach shows its limits in terms of maintainability and portability. In this paper, we propose to use the composite design pattern to extend this model for supporting hierarchies of components. The component hierarchy is traversed to instantiate the application and extract the data flow graph required for the execution. This approach has been implemented for the FlowVR middleware. It enables to define parametric composite components, commonly called skeletons, that can be reused in various applications. This approach proved to significantly leverage application modularity as presented in different case studies.

Keywords: Interactive Applications; Parallelism; Components; Composite Design Pattern

1 Introduction

An interactive application involves a program and a user interacting in an endless iterative process through input and output devices. It is often referred to 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, these applications must perform under strong performance constraints, often measured in terms of latency and refresh rate.

For example, the Hercules system couples an earthquake simulation and an on-line visualization using 2000 processors to reach the frequency of 2Hz on a

1200 billions elements simulation [1]. Other initiatives intend to design cross-continental interactive applications relying on the performance of optical networking [2]. A number of virtual reality applications are relying on parallel machines to provide the required I/O and computing resources. Blue-C [3] and Grimage [4] are good examples of high performance immersive platforms relying on parallel machines to process in real time data acquired through a network of cameras.

In this paper, we focus on two issues faced when designing such applications:

- 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.

Most iterative applications can be seen as an assembly of static tasks endlessly processing incoming data and forwarding results to other tasks. Many scientific visualization tools use this data flow graph model to specify the applications [5]. But the graph tends to quickly become complex as the application size grows, impairing the modularity.

In this paper, we propose to rely on the composite design pattern to extend the data flow graph model. Edges are components that can recursively contain other components. Vertices link sibling component ports or parent/child ports. 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. A traverse either leads to an error (missing data impairs the traverse completion) or a success. For instance a traverse is called to extract the data flow graph required for the execution from this hierarchical application description. This approach enables us to define highly generic composite components, enforcing the application maintainability and portability. In particular, we can define skeletons, i.e. parametric composite components, that encapsulate commonly used and optimized parallel processing patterns. This approach has been implemented for the FlowVR middleware [6].

Section 2 discusses related works. After an overview of FlowVR (section 3), we present the hierarchical component model in section 4. Section 5 presents a collection of skeletons built using our model. Section 6 focuses on 2 case studies to discus the benefits of our approach on real applications, before to conclude in section 7.

2 Related Work

The goal of scientific visualization is to process large data sets to compute images. Interactivity enables for instance users to change their point of view on the data set or the transfer function applied for volume rendering. Applications are developed with visualization environments like OpenDX [7], Iris Explorer [8] or VTK [9]. These environments are usually based on a data flow graph model where processing tasks receive data and generate new ones. Most of them 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 [10]. 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 visualization environements do not provide the necessary constructs [11].

Attempts to associate virtual reality, scientific visualization and simulations push forward the complexity of interactive applications. They involve various simulation codes that may generate large data sets, advanced I/O devices, like network of cameras, projector arrays, haptic devices. Pipeline must be used with care. It improves the application frequency, but also increases the latency. So to ensure a good trade-off between frequency and latency multiple forms of parallelism are associated, from pipelines or data parallelism to dynamic task parallelism.

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 [6], OpenMask [12] or COVISE [13] propose different approaches to support such features. However, the resulting application code tends to be difficult to be 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 from 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 Architecture Description Languages for distributed applications. SCIRun, an environment dedicated to scientific visualization, is based on the CCA model [14]. Some extensions intend to enforce the support of parallel components and the associated coupling patterns [15]. But these models suffer from the same limitations as the systems mentioned earlier (FlowVR, COVISE) regarding the modularity of parallel component coupling. Fractal [16] is a hierarchical component model. We are aware of one implementation of Fractal for parallel (grid) applications: ProActive [17]. 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 the application modularity.

The skeleton model proposes a pattern language for parallel programming [18, 19]. A program is written from the composition of predefined parallel patterns. Various environments rely on this model like ASSIST [20] for grid computing or Skipper [21] for vision applications. Skeletons have a clear semantics, can be associated to a cost model and hide their implementation details to the application developer. Given the target architecture, the application is compiled down to a specialized parallel code. Hierarchies of skeletons are supported by some environments like Skipper-D. ‘ With the emergence of multicore architectures and GPU programming, some programming environments propose to focus on a stream paradigm, like StreamIT [22], Brook [23] or Cg [24]. They target streaming applications like video, voice or DSP programming. A program is usually a set of iterative modules that communicate via FIFO data channels. Parallelism is expressed by the composition of a reduced number of skeletons. For example, in StreamIT, developers are allowed to use 3 kinds of skeletons: Pipeline, SplitJoin and FeedBack Loop. By limiting the available skeletons, it constrains the program to simple data parallel access patterns, enabling to write efficient compilers for the targeted architecture. It is however too restrictive to ensure efficient executions on a general purpose and potentially heterogeneous parallel machine.

3 FlowVR

We present in this section FlowVR [6]. Our component model relies on this middleware. FlowVR is dedicated to parallel interactive applications. It is based on the data flow model also used by other scientific visualization tools. A FlowVR application is a network of static iterative processes connected by data flow channels. The main target applications include virtual reality and scientific visualization.

FlowVR has been used for developing various large interactive applications [25, 26]. FlowVR is open source[1]. It is distributed with extensions like FlowVR-Render that enables distributed rendering or VTK-FlowVR that encapsulates VTK[9] applications into FlowVR Modules [27].

3.1 FlowVR Run-time

An application is composed of modules exchanging data through a FlowVR network. A module is an endless iterative code that defines input and output ports. At each iteration it reads incoming data from input ports, processes these data and writes the results on output ports. A module runs in its own independent process or thread, thus reducing the effort required to turn an existing code into a module. For instance an MPI program can be modified to define one module per process.

[1] FlowVR is available at http://flowvr.sf.net

The FlowVR network is handled at run-time by a FlowVR daemon running on each host of the target machine. Daemons act as brokers. They relay messages between modules. Modules are not aware of the existence of other modules. A module only exchanges data with the daemon that runs on the same host. If the destination module runs on the same host, the daemon gives this module a pointer to the data (messages are stored in shared memory segments). If the destination runs on a distant host, the module sends the data to the daemon of this host using TCP. At reception the daemon stores the message in a shared memory segment and handles a pointer to the destination module.

The role of the daemon is not limited to data forwarding. It can load plugins to process data, duplicate, merge or split messages for instance. The user can define its own plugins if required. Notice that plugins have a less restricted access to the shared memory than modules, enabling to implement more efficient message handling actions.

Each FlowVR application is managed by a 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 launched, modules register to their local daemon that sends an acknowledgment to the controller. Then, the controller sends to each daemon the routing table and list of plugins to load to implement the FlowVR network.

3.2 Flat Data Flow Graph

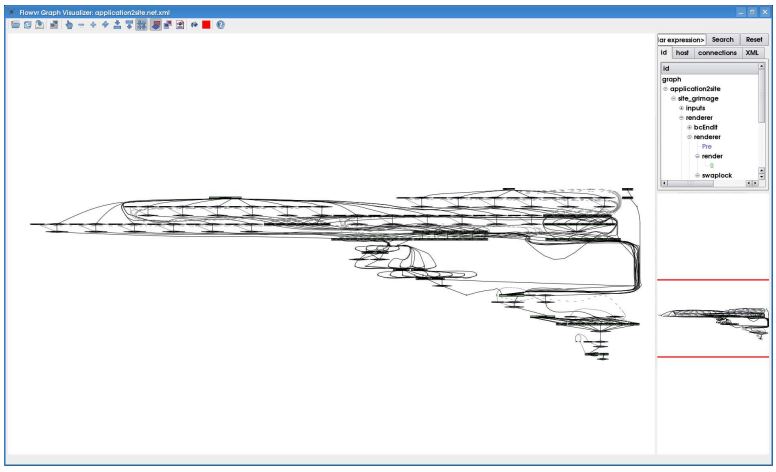


Fig. 1. The flat data flow graph of a large FlowVR application. Edges represent processing tasks and vertices data channels.

At low level a FlowVR application is modeled by a flat data flow graph composed of:

- Modules with input and output ports, each one is mapped on a given host,
- Filters that are daemon plugins. Like modules, filters have input and output ports, and are mapped on a given host.
- Connections that represent FIFO data channels. A connection connects one source input port to a destination output port.
- Routing nodes that have one input port and one or more output ports. They are assigned to a given host and model message routing actions.

In the first versions of FlowVR, the application developer had to specify its application describing this graph. He was assisted by a library of Perl functions that encapsulated some commonly used patterns. However large applications proved difficult to debug and maintain, motivating the adoption of a hierarchical approach to further enforce the application modularity (Fig. 1).

4 Component Model

We adopt a hierarchical component model to describe a FlowVR application. It is based on the composite design pattern [28].

4.1 Hierarchical Components

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

Primitive components. A primitive component is a base component that cannot contain an other component. Primitive components are modules, filters, routing nodes and connections.

Composite components. A composite component contains other components (composite or primitive). It has input and output ports. A port is visible from both, the outside and the inside of the component. It identifies the data that can cross a component boundary. Component encapsulation is strict. A component can not be directly contained into two parent components.

4.2 Links

A link connects two component ports. It cannot directly cross a component membrane. A link between 2 ports is allowed only for the 2 following cases:

- A descendant link connects a port of a parent composite component to a port of one of its child component. Such links must always connect an input/input or output/output pair of ports.
- A sibling link connects two ports of two components having the same parent component. Such a link must always connect an input/output pair of ports.

Port typing can be enforced if required, putting more constraints on the ports that can be linked. For instance, link could be restrained to connect only ports corresponding to the same data type.

4.3 Example

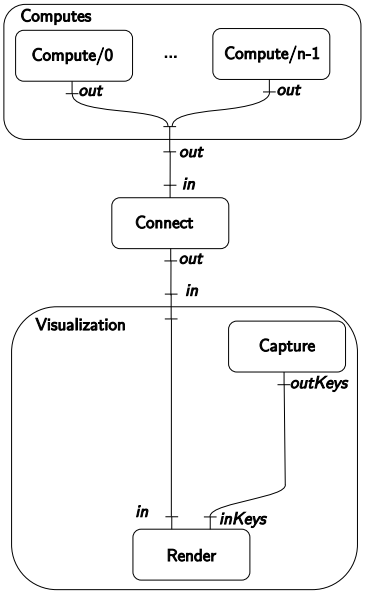


Fig. 2. Application example. *Computes* simulates the dynamics of a ball falling into a water tank. Results transit up to *Render* for rendering. *Capture* forwards mouse positions to *Render* that uses them to render the simulation scene with the point of view requested by the user.

Throughout this paper, we use a simple example (Fig. 2). It shows the classical structure of a basic interactive application. In this iterative simulation, component *Computes* publishes its state at each iteration. We can for instance consider that this simulation computes the dynamics of a ball falling into a water tank. Each simulation state is received by a *Render* component. For a given point of view, this component computes an image giving a view on the simulation scene. The user can control this point of view with a mouse. A *Capture* component is in charge of reading the mouse position and forwarding it to *Render*.

For sake of simplicity, we keep this application synchronous, i.e. the *Render* component can only start the next iteration if it receives data from *Computes* and *Capture*. Often real applications loose this synchronization by introducing data sampling components (a sampling pattern is presented in section 5).

Using the hierarchical component model, the example is structured as follows (Fig. 2):

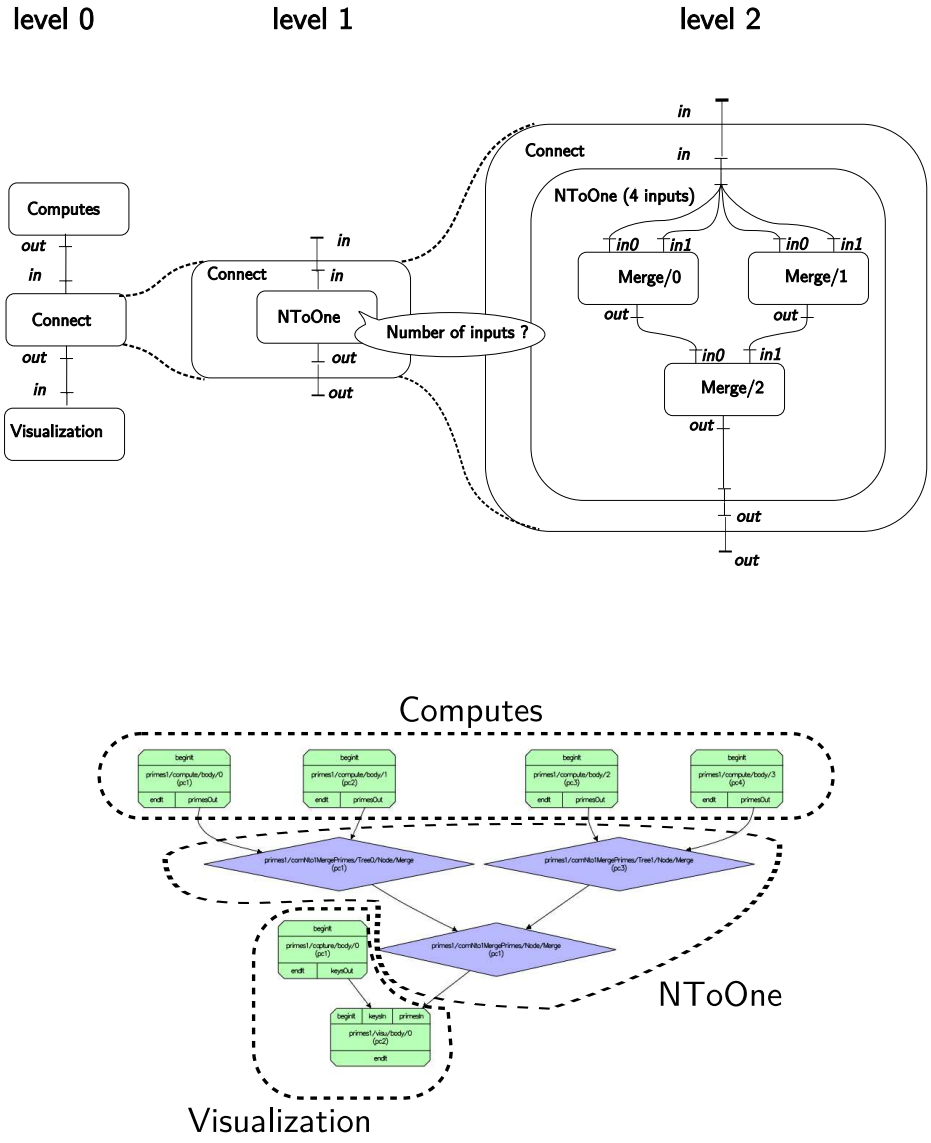


Fig. 3. a) Two levels of hierarchy for the *Connect* component. The skeleton defined by *NtoOne* is generated according to the number of *Computes* primitive components. b) The flat data flow graph for the application. Dashed sets show the composite components the graph elements are related to (connections are arrows, modules are in green and filters in blue).

– As the *Capture* and *Render* components are closely related, they are stored in a composite component called *Visualization*. This encapsulation is a commodity that enables to easily reuse this assembly having just to handle the *Visualization* component.
– The *Computes* component is actually a parallel application that spawns n processes. The goal is to be able to speed-up the simulation involving more processors if available. *Computes* is modeled as a composite component with one output port *out* to send its simulation state at each iteration. It contains n child components *Compute/0,...*, *Compute/n-1*. These are primitive components, each one having an output port *out* linked to the *out* port of *Computes*. The value n and where these n processes are mapped on a target architecture is unknown at the time of the application design. They will be instantiated later when traversing the application to call configuration controllers. Notice that communications can take place between the different parallel processes, but they are not modeled here. We consider that they are under the responsibility of the programming environment used to parallelized the application, MPI for instance.
– *Computes* being a parallel component, each process spawned computes one part of the simulation state. The *Visualization* component is not designed to received partial results. We could modify the *Visualization* component, but we actually prefer to manage this issue outside of this component. Application modularity is enforced by delegating data redistribution issues to specialized components. We use an extra component, called *Connect*, between *Computes* and *Visualization*. *Connect* is in charge of gathering the partial results from the various *Compute/i* processes to forward a single message containing a full simulation state to *Visualization*. *Connect* is a composite component (Fig. 3.a). It is built from the *NtoOne* component. This component encapsulates a generic tree pattern for data redistribution. *Connect* just set the parameters of *NtoOne*: the arity of the tree (2) and the type of the component used for the tree nodes (*Merge*). The actual content of *NtoOne* is only known once *Computes* is properly instantiated. Only at this point *NtoOne* knows how many pieces of data it has to gather to set the tree depth. The *NtoOne* configuration controller must be executed after the *Computes* configuration controller. We see here that the traverse algorithm in charge of executing the configuration controllers has to respect a given processing order. A possible traverse order is: *Computes, Visualization, Connect, NtoOne, Merge/0, Merge/1, Merge/2, Compute/0, Compute/1, Compute/2, Compute/3, Capture, Render*. *Merge* is a primitive component that builds one message sent on its *out* port from the 2 messages it reads on its *in/0* and *in/1* ports.

Notice that if the application is configured with only one component *Compute/0*, *Merge* becomes a simple point-to-point connection between *Compute/0* and *Render*.

The model we propose first target applications with static components, i.e. without components created while the application is running. Because of their

iterative nature, interactive applications tend to be mostly static. However, if required for some parts of the application, a component can dynamically create or kill threads or processes as long as it implements a proxy that hides this dynamic behavior. We are also working on extending the model to support some level of run-time reconfiguration.

4.4 Controllers

To improve the application genericity and thus its portability, instantiation of some component aspects are deferred to controllers. A controller is local to a component. It can only modify the state of its component. It can read the state of other components its owner is linked to (directly or not). A component can have several controllers. It usually enforces modularity to have multiple specialize controllers. We distinguish 2 types of controllers:

- An introspection controller just get data from its component. For instance an introspection controller can be dedicated to print its component name in a file.
- A configuration controller modifies its component state. In the example application, the child components of *Computes* are generated by such a controller.

Controllers are called during an application traverse. Usually one traverse just calls one controller per component. During a traverse, parameters can be exchanged between controllers. It enables for instance to exchange a file descriptor where each controller appends the name of its component. The final result of the traverse is a list of all application components. The result may of course depend on the execution order of the different traverses.

Our model imposes one configuration controller, called *execute*. This controller creates child components. For example, in the *Computes* component, the *execute* controller creates all *Compute/i* primitive components and links them to *Computes*. Data distribution components usually have *execute* controllers that need to get data from the neighbor components. For instance, the *execute* controller of *NtoOne* needs to get the number of *Compute/i* components to create the merging tree.

Developers can create controllers dedicated to a given aspect. A controller can be in charge of mapping primitive components to the target architecture processors. Implementing mapping in a controller enables to keep the application description independent of the mapping. In FlowVR, the application is first traversed to call the *execute* controller, then a mapping controller is called, and a third controller generates the flat data flow graph.

An other example of introspection controller used for FlowVR is the command line generator. The construction of command lines to launch modules is delegated to an introspection controller. This controller builds a command line using data related to the FlowVR network (hosts list, number of processes), configuration files (target architecture description) or user parameters (application

specific parameters). This specific controller is embedded into composite components called metamodules. A metamodule handles modules that are logically related, in particular when they are all started from a single command. This is for instance the case for a MPI code that uses `mpirun` to start all its processes.

Notice that a controller can be seen as an aspect (in the Aspect Oriented Programming way). Nevertheless, we do not have code weaving. Controllers are embedded in components by the programmer.

4.5 Traverse Algorithm

As seen for the example (Section 4.3), in a traverse the execution of controllers may need to obey a certain order to respect data dependencies. We propose a simple algorithm that guarantees to complete the traverse when possible or return the list of misprogrammed components if some data dependencies cannot be solved whatever the execution order is.

The traverse algorithm is a greedy process. The algorithm manages a queue of non-executed components, initialized with the top-level components of the application. For each component in this queue, the algorithm tries to execute the associated controller. If the controller is successfully executed, then all of its children are pushed in the queue. Otherwise, the algorithm restores the component initial state and push it at the end of the queue. The traverse ends successfully when the queue is empty. If no controller can be called on the rest of the components in the list, then the algorithm stops in a fail state. The controller of the remaining components cannot be executed either because at least one of these components is misconfigured (a parameter is not instantiated for instance), or because a cycle of dependencies has been introduced when assembling the components.

4.6 Traverse Proof

We prove the traverse algorithm always ends, with success if a solution exists, and that the number of controller calls, successful or not, is at most quadratic in the number of components.

Let C be the set of all components in an application and N_{comp} the size of C. The goal of the algorithm is to iterate on all components in C with a consistant order. We put in the non-executed queue a marker that denotes the starting point. Each time the marker comes back to the front of the queue, it is appended at the end of the queue. We count the number of times the marker has reached the front since the algorithm started. It denotes what we call in the following the *number of iterations*.

Let $NonExecuted_k = \{c \in C /$ the controller of c has not been executed at the iteration k $\}$ and $Executed_k = \{c \in C /$ the controller of c has been successfully executed during the iteration k $\}$. We call N the iteration that reaches a fixed point, i.e. the first iteration where $Executed_N = \emptyset$. In this case, the algorithm stops. The misconfigured components or dependency cycles are contained in $NonExecuted_N$.

Let $E_k = \bigcup_{i=1}^{k} (Executed_i)$ be the set of components successfully executed from the first to the k^{th} iteration. Let $\overline{E_k} = \bigcup_{i=k}^{\infty} (NonExecuted_i)$ be the set of components that have to be executed after the iteration k.

We call $\overline{E_\infty} = \bigcap_{i=1}^{\infty} \overline{E_i}$ the set of components that cannot be executed.

Thus we have:

- $\forall k, C = E_k \oplus \overline{E_k}$
- $\overline{E_0} = C$ and $E_0 = \emptyset$

We first prove the algorithm always ends.

Proposition 1. *The traverse algorithm reaches a fixed-point with $N \leq N_{comp}$ and $NonExecuted_N = \overline{E_\infty}$*

Proof. During execution of traverse, we are assured that $Executed_N \neq \emptyset$, so for all k we have $\overline{E_{k+1}} = \bigcup_{i=k+1}^{\infty} (NonExecuted_i) \subset \bigcup_{i=k}^{\infty} (NonExecuted_i) \subset \overline{E_k}$. So $\overline{E_N}$ decreases to $\overline{E_\infty}$. The algorithm reaches a fixed-point where $lim_{k\to\infty}\overline{E_k} = \overline{E_\infty}$.

As $C = E_k \oplus \overline{E_k}$, if at the iteration k we have $\overline{E_k} = \overline{E_{k+1}}$ then $E_k = E_{k+1}$. So the algorithm reaches the fixed-point at k.

Consequently $\overline{E_k}$ strictly decreases to $\overline{E_\infty} \Rightarrow N \leq |\overline{E_0}| = N_{comp}$.

We now focus on the complexity of the algorithm.

Proposition 2. *The traverse algorithm performs at most N_{comp}^2 calls to controllers.*

Proof. Let $Calls$ be the total of calls to controller. $Calls = \sum_{k \leq N} |NonExecuted_k|$. Previously, we proved:

- $N \leq N_{comp}$
- $\forall k, NonExecuted_k \subset C \Rightarrow |NonExecuted_k| \leq N_{comp}$

So $Calls \leq N_{comp}^2$

The overhead due to unsuccessful controller calls can be significant. But implementing an algorithm that solves all constraints to identify an acceptable execution order would be complex or it would require the application developer to encode extra information into its program to help that algorithm. Our solution is a good trade-off between scalability and complexity of the implementation. We experimented applications with 200 components. Traverse computation time is about one second only.

We now characterize $\overline{E_\infty}$, the set of remaining components. Let $Data = \{c \in C$ / c cannot be executed because a data is missing $\}$ and $Dep = \{c \in C$ / c cannot be executed because it depends on a component that has not been executed yet $\}$. No other reason can lead to a controller call failure. So we have $\overline{E_\infty} = Data \cup Dep$.

Proposition 3. *If $Data = \emptyset$ and $\overline{E_\infty} = Dep \neq \emptyset$ then there is at least one dependency cycle in $\overline{E_\infty}$*

Proof. Assume there is no dependence cycle in *Dep*. So there is a longest dependency path. Let c and d be the components at the extremities of one of the longest dependency paths.

But because c belongs to *Dep* and not to *Data* ($Data = \emptyset$), there exists e in *Dep* such as c depends on e. So the path from e to d is longer than the longest path from c to d. It contradicts the assumption: there is a dependence cycle in *Dep*.

This proposition shows the traverse algorithm can help debugging an application. If the traverse fails, the user should first fix the components with missing data. Usually such flaws are detected when the controller fails if error raising has been properly programmed. Next, if the algorithm still fails, the user should look at suppressing the cyclic dependencies. In our implementation we rely on exceptions to signal when controller fail.

4.7 The FlowVR Front-end

The hierarchical component model only affects the front-end of FlowVR (Fig. 4). The run-time engine is not modified. Components are written in C++ and compiled into shared libraries. An application is also a composite component compiled into a shared library. It can thus be reused in other applications without being recompiled. The FlowVR front-end loads the application and applies a sequence of several traverses to produce the list of commands to start the modules and the instructions to forward to the different daemons to implement the application network. The flat data flow graph is usually saved as it is useful for debugging purpose.

5 Skeletons

We present four base parametric composite components, i.e. skeletons, that proved to be very useful for developing interactive applications. These skeletons provide users an easy way to handle parallel processing patterns or complex communication schemes. These skeletons fully take advantage of the component hierarchy and modularity provided by the controller based approach. They are templated to enforce their genericity. Their instantiation is deferred to their *execute* controller (Sect. 4.4):

Pipeline This is a very simple skeleton modeling a sequence of precessing steps. It is modeled by a composite component containing an arbitrary sequence of linked components (primitives or composite).

Parallel This skeleton creates N instances of a component passed as a template. The skeleton creates the same ports than the template component. Once the internal components created, their ports are linked to their equivalent skeleton ports. The *Computes* component in our example could have been alternatively designed by encapsulating a *Parallel* component pattern using *Compute* as template component. This skeleton can be used as a shell

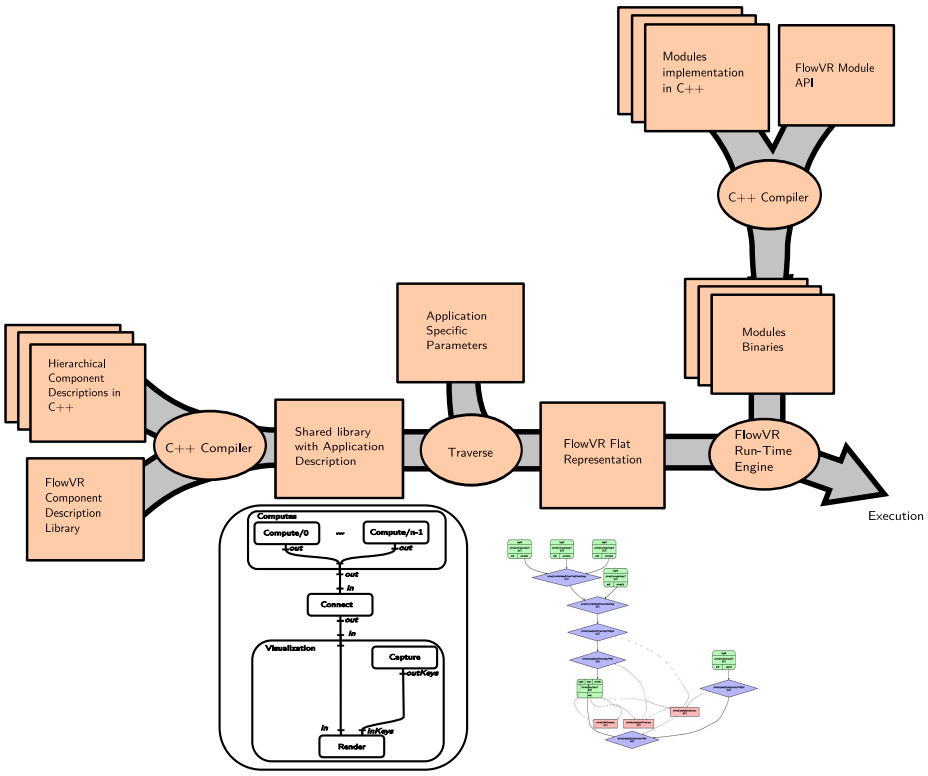


Fig. 4. The FlowVR front-end. Components (left to righ) are compiled, loaded and traversed to provide the module launching commands and the instructions for deamons. Once compiled, modules (top to bottom) are started as requested by the application.

for duplicating a given component. It can also be used to encapsulate a static parallel program. In this case communications due to parallelization are not visible from the component point of view. We consider the parallel programming environment used for the parallelization takes care of these communications.

Tree A tree skeletons has two ports, the root and the leaves. The number of leaves in the tree is defined a traverse time according to the number of neighbors connected to the leaves port. We distinguish 2 specializations of the tree depending on the data propagation direction, either from root to leaves (the *OnetoN* component) or from leaves to root (the *NtoOne* component). The arity of the tree is a parameter to be instantiated. The node type used to build the tree is a template pattern. Here are some examples of components pattern built from *Tree*:

Broadcast The simplest skeleton that can be built from the tree. It uses the *OnetoN* skeleton instantiated with a primitive component, a routing node, that forwards the messages it receives on its input to each of its outputs. The arity of the broadcast tree depends on the number of outputs of the template component.

Scatter Similar to the Broadcast except that the template component splits the input message into sub-messages forwarded on its outputs. For instance, a classical 3D rendering parallelization approach, called sort-first [29], consists in having a task responsible for one area of the screen (Fig. 5.a). Thus, a task only requires to execute the graphics primitives that will contribute to its screen area. To distribute the graphics primitive, we can use a *Scatter* with a *Culling* component that uses a fast method to test if a graphics primitive contributes to a given screen area [27].

Gather A *OneToN* tree that uses a message merging template pattern. Using a template component that sorts the integers it receives, it creates a distributed merge sort (Fig. 5.b). This skeleton is also used by the *Connect* component of our example (Fig. 3).

Sampling This skeleton is specific to interactive applications where tasks may run at different frequencies. For example, a physical simulation has to run at high frequency to be stable, while graphics rendering usually runs between 30 and 60 Hz. If the two tasks are directly connected with a FIFO connection, it will force both tasks to run a the frequency of the slowest one, the rendering task in this case. To avoid this issue a common approach is to sample the incoming signal. This sampling could be performed by the rendering task, making the rendering task less generic. To enforce the modularity, we design a special skeleton that samples data streams under the control of their destination tasks. With this approach neither the source neither the destination tasks need to be modified or even recompiled. The sampling skeleton is an assembly of 2 composite components (Fig. 5.c):

– The *Filter* composite component analyzes and samples the incoming data stream according to an external policy. It has four ports : *in* receives the

incoming data stream, *out* produces the sampled signal, *freq* sends the frequency of the incoming stream and *order* receives sampling orders.
- The *Sampler* composite component controls the sampling policy. It has two ports : *freq* receives the frequency of the incoming stream and *order* sends orders about the stream sampling. Using the incoming stream frequency, it decides the messages that have to be discarded and the ones to replay.

By changing the template components *Sampler* and *Filter* different sampling strategies can be implemented.

Because there is no discontinuity from primitive components to high-level composite ones, the developer can freely choose to combine, extend, specialize or simply ignore these skeletons. In a sense the approach we propose is very close to the one of the C++ Standard Template Library. This skeletons can also be seen as a derivative of Cools skeletons [18] for a specific application domain. One of the main difference is the absence of cost model.

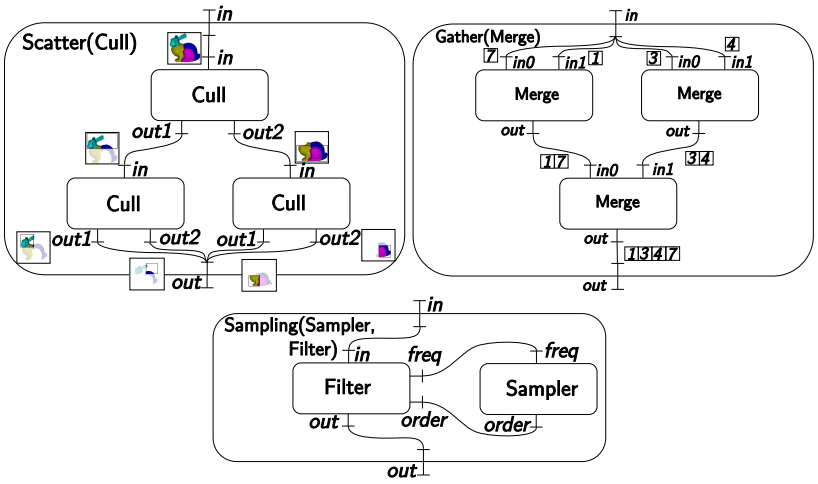


Fig. 5. a) Sort-first scatter pattern. The Culling components route the graphics primitive for rendering the bunny according to the screen area they project onto. b) Integer merge-sort scatter pattern. c) Sampling pattern. *Filter* is the operative part of the communication: it processes sampling on incoming messages flow. *Sampler* is the control part: it decides the sampling policy.

6 Case Studies

In this section, we present two case studies taking advantage of the component hierarchy and the skeletons presented in previous sections. The first application

shows an example of coupling MPI and FlowVR. The second example is an interactive 3D modeling application using a camera network.

6.1 Case Study 1: MPI Fluid Simulation

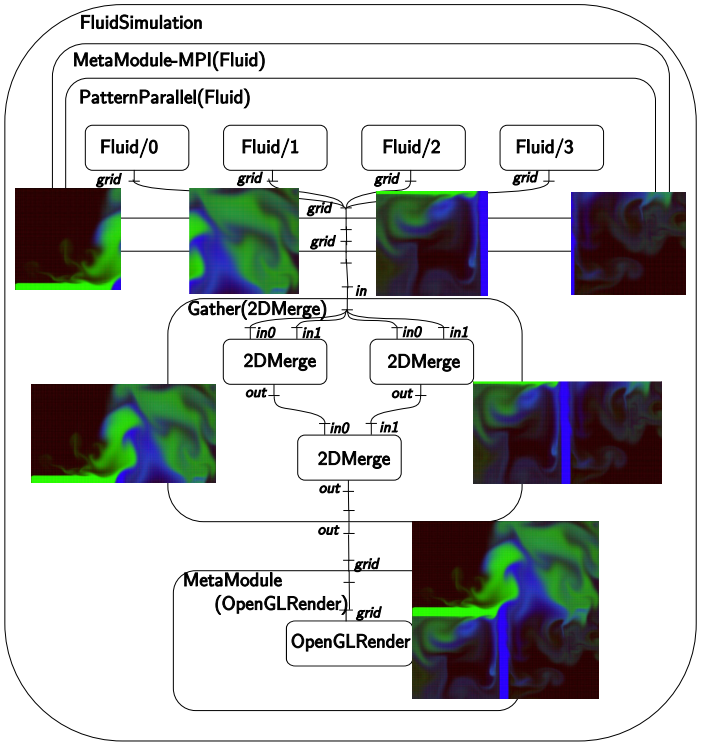
We implemented a fluid simulation algorithm [30] using MPI (Fig. 6.a). This application shows how to integrate a MPI code. The fluid simulation is based on a 2D grid of cells. At each new iteration, a new state is computed for each grid cell. This state depends on the state at the previous iteration of the considered cell and its four neighbors. The simulation is parallelized by splitting the grid cell into blocks distributed amongst the different MPI processes. Data exchange between blocks are MPI communications, transparent to FlowVR. The MPI code is modified so that each process is a FlowVR module with one output port to send the result of each iteration. These modules are called *Fluid/0 .... Fluid/N*, the number being assigned based on the rank provided by MPI. Beside the actual MPI code of the modules, a *Fluid* primitive component is written. A metamodule *Metamodule-MPI* implements the controller to generate the launching command using *mpirun*. It also contain a parallel skeleton that creates the correct number of instances of the *Fluid* module. It is important here that the ranking be the same as the one assigned by MPI. The *Metamodule-MPI* component is linked to a Gather skeleton (Fig. 5.b) using a *2DMerge* filter as template. The goal here is to gather the results of each MPI process into one full 2D grid forwarded to an OpenGL renderer. For more implementation details refer to the fluid example provided with the FlowVR source code.

A possible traverse order to generate the flat data flow graph and the launching commands is:

1. *FluidSimulation* instantiates the 3 components: *MetaModule-MPI(Fluid)*, *MetaModule(OpenGLRender)* and *Gather(2DMerge)*.
2. *MetaModule-MPI(Fluid)* creates the *PatternParallel(Fluid)* component.
3. *PatternParallel(Fluid)* instantiates the 4 *Fluid* modules and set their ranks.
4. *Gather(2DMerge)* detects the 4 *Fluid* and creates the gather tree with 3 *2DMerge* filters. Each *Fluid* is connected to one leave of the gather tree according to their rank.
5. *MetaModule(OpenGLRender)* creates the *OpenGLRender* module.
6. *MetaModule-MPI(Fluid)* generates the MPI command line with the appropriate list of hosts and ranks.
7. *MetaModule(OpenGLRender)* generates the UNIX command line to launch the rendering in the appropriate X-server.

6.2 Case 2: Real-Time 3D Modeling

We ported a parallel real-time 3D modeling application. It consists in computing in real-time a 3D model of a scene from the various 2D video-streams acquired by multiple video cameras surrounding the scene [31] (Fig. 7.a). Real-time 3D




```
void FluidSimulation::execute()
{
 // Components instantiation
        Component* fluid = addObject(
                                    MetaModule-MPI<Fluid>());
        Component* gather = addObject(
                                    Gather<2DMerge>());
        Component* render = addObject(
                                    MetaModule<OpenGLRender>());

 // Link components
        link(fluid->getPort("grid"), gather->getPort("in"));
        link(gather->getPort("out"), render->getPort("grid"));
}
```

Fig. 6. a) Fluid application. Four MPI processes compute a fluid simulation on a 2D grid. A gather skeleton merges results from MPI processes and sends the full grid to an OpenGL renderer. b) C++ description of the FluidSimulation component that encapsulates the application.

modeling enables full body interactions into a virtual environment [4]. 3D modeling is both I/O and computation intensive. We typically use between 6 and

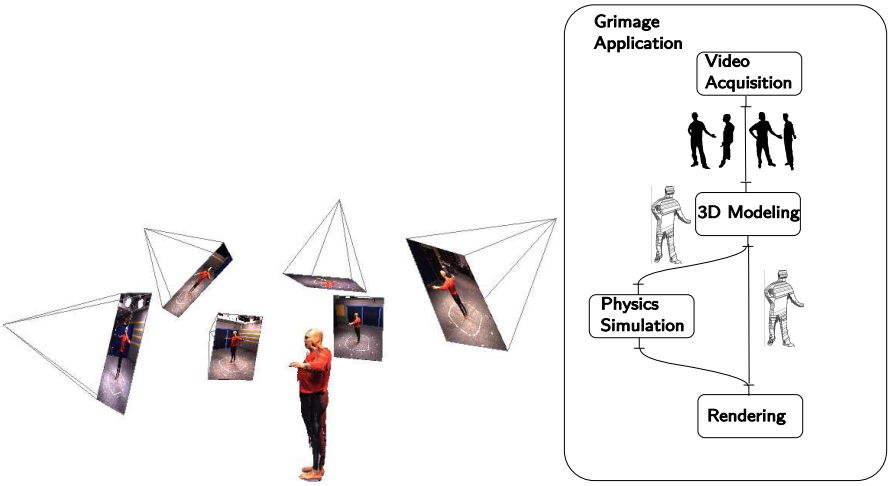


Fig. 7. a) A 3D model of a person computed from 6 cameras. b) Description of the application. This composite contains 4 composite components.

15 cameras, each one acquiring 30 images per seconds. The computation of 3D models must match the camera frequency and run in less than 100 ms per 3D model to keep the overall latency small enough to enable interactions. A parallelization is thus required. Parallelization is based on several steps. For sake of conciseness we just give an overview of the parallel algorithm here. Refer to [32] for details. First, the video stream of each camera is acquired and filtered to subtract the background and compute the image silhouette. This pipeline is executed in parallel on each machine a camera is connected to. Next silhouettes have to be redistributed for computing the 3D model. 3D modeling is implemented with 3 parallel processing steps separated by data redistributions.

This application contains 4 composite components (Fig. 7.b): a video acquisition component, a 3D modeling component, a physical component and a rendering one. The video acquisition component is a parallel pattern containing a pipeline of the different steps from acquisition to silhouette extraction. The 3D modeling component is another hierarchy of components making an intensive use of various skeletons. Physical simulations are computed by SOFA [33], an external framework. This framework computes collisions between virtual objects and user 3D model. The rendering component renders all meshes (virtual objects and user 3D model) and the virtual environment. The application designed is independent from the number of processors available on the target machine, and from the number of cameras and their mapping on the machines.

This application represents a significant development effort involving several developers over several years. Developments started in 2002 using MPI. It was quickly abandoned as MPI proved not to offer a sufficient level of modularity for this type of interactive application. A computer vision specialist should be able to work on the acquisition pipeline without having to worry about the MPI code

or the overall coherency of the communication schemes. We switched to FlowVR that better separates the code of the tasks (the modules) from the task coordination issues. But as the applications grew, for instance texturing of the 3D model started in 2006 and SOFA was only added in 2007, the FlowVR network became very complex and bugs difficult to track and solve. Switching to the hierarchical component approach increased significantly the application maintainability, scalability and portability. It did not directly modify the flat data flow graph and so the performance. But because the modularity improved, performance enhancements proved easier to implement. Several videos are available at http://flowvr.sf.net showing the evolution of the application and the level of performance reached.

Let now focus on the acquisition component. The full pipeline from camera to image silhouette is implemented in a composite component. Using the parallel skeleton, we are able to instantiate this pipeline for all cameras (Fig. 8). These pipelines can be driven from a user interface for on-line tuning of some parameters. To implement this new feature we used 3 parallel skeletons and 1 sampling skeleton (Fig. 9).

Controllers ease extensions of this basic implementation of the acquisition component. For instance, we developed a controller that adds a supervision interface to control these pipelines. This supervision consists in a graphic user interface to set some parameters for the different modules in the pipeline. For example, the user can set the acquisition rate. This interface also displays the outputs from several stages of the pipeline. We use this interface to control and debug the acquisition algorithms. We implemented this controller using a new component that encapsulates the graphics interface. This controller also adds several asynchronous communications that send parameters to pipeline components (Fig. 9). These communications use the sampling skeleton. This implementation enables to separate the main implementation of the pipeline from this supervision aspect. It improves the modularity and provides a simple solution to extend the application.

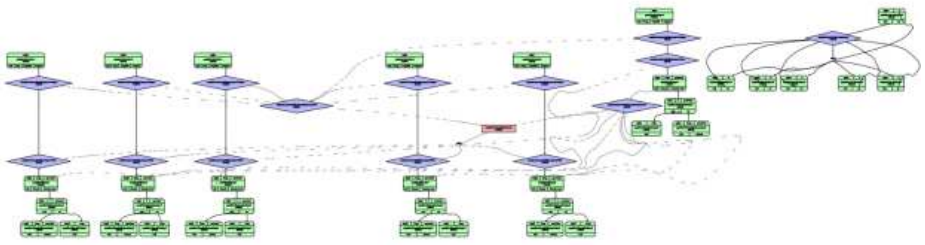


Fig. 8. Flat data flow graph of the acquisition component for 6 cameras (50 nodes and 68 edges).

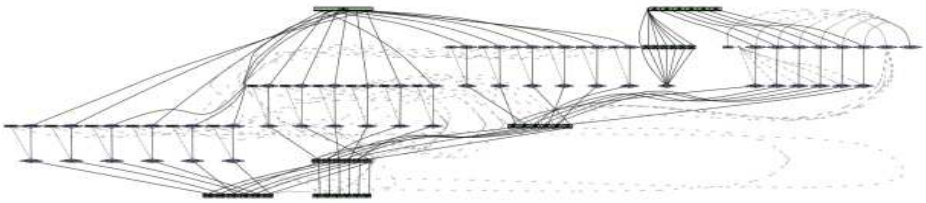


Fig. 9. Flat data flow graph of the acquisition component for 6 cameras and a supervision interface (105 nodes and 176 edges).

7 Conclusion

We presented a framework to use a hierarchical component model for interactive applications. 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. Controllers are called in a traverse process. We presented a traverse algorithm that calls the controllers in an appropriate order or produce an error if completion is not possible due to cycles or missing data. This approach was implemented for the FlowVR middleware and proved effective to leverage the modularity of applications.

Acknowledgment

This work was partly funded by Agence Nationale de la Recherche contract ANR-06-MDCA-003.

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. Smarr, L.L., Chien, A.A., DeFanti, T., Leigh, J., Papadopoulos, P.M.: The OptIPuter. Communication of the ACM **46**(11) (2003) 58–67
3. Gross, M., Wuermlin, S., Naef, M., Lamboray, E., Spagno, C., Kunz, A., Koller-Meier, E., Svoboda, T., Gool, L.V., S. Lang, K.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)
4. Allard, J., Ménier, C., Raffin, B., Boyer, E., Faure, F.: Grimage: Markerless 3D Interactions. In: Proceedings of ACM SIGGRAPH 07, San Diego, USA (August 2007) Emerging Technology.
5. Brodlie, K., Duce, D.A., Gallop, J.R., Walton, J.P.R.B., Wood, J.: Distributed and Collaborative Visualization. Computer Graphics Forum **23**(2) (2004) 223–251

6. Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., Robert, S.: FlowVR: a Middleware for Large Scale Virtual Reality Applications. In: Proceedings of Euro-par 2004, Pisa, Italia (August 2004)
7. 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: IEEE Visualization Conference, Los Alamitos, CA, USA, IEEE Computer Society Press (1992) 107–114
8. Foulser, D.: IRIS Explorer: a Framework for Investigation. Journal of ACM SIGGRAPH 95 **29**(2) (1995) 13–16
9. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit An Object-Oriented Approach To 3D Graphics, 3rd Edition. Kitware, Inc. (2003)
10. 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, Los Alamos National Laboratory (2000)
11. Wang, C., Gao, J., Shen, H.W.: Parallel Multiresolution Volume Rendering of Large Data Sets with Error-Guided Load Balancing. Parallel Computing **31**(2) (February 2005) 185–204
12. D.Margery, B.Arnaldi, A.Chauffaut, S.Donikian, T.Duval: 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. (2002) 101–110
13. A.Wierse, U.Lang, Rhle, R.: Architectures of Distributed Visualization Systems and their Enhancements. In: Eurographics Workshop on Visualization in Scientific Computing, Abingdon (1993)
14. Keming Zhang and Kostadin Damevski and Venkatanand Venkatachalapathy and Steven G. Parker: SCIRun2: A CCA Framework for High Performance Computing. In: Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS. Volume 00., Los Alamitos, CA, USA, IEEE Computer Society (2004) 72–79
15. Denis, A., Pérez, C., Priol, T.: PadicoTM: an Open Integration Framework for Communication Middleware and Runtimes. Future Generation Computer Systems **19**(4) (2003) 575–585
16. Eric Bruneton and Thierry Coupaye and Matthieu Leclercq and Vivien Quéma and Jean-Bernard Stefani: The FRACTAL Component Model and its Support in Java: Experiences with Auto-Adaptive and Reconfigurable Systems. Software Practice & Experience **36**(11-12) (2006) 1257–1284
17. Baude, F., Caromel, D., Morel, M.: From Distributed Objects to Hierarchical Grid Components. In: CoopIS/DOA/ODBASE. (2003) 1226–1242
18. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press (1989)
19. Mattson, T.G., Sanders, B.A., Massingill, B.L.: A Pattern Language for Parallel Programming. Addison Wesley (2004)
20. Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., Zoccolo, C.: ASSIST as a research framework for high-performance Grid programming environments. In Cunha, J.C., Rana, O.F., eds.: Grid Computing: Software environments and Tools. Springer (January 2006)
21. Serot, J., Ginhac, D.: Skeletons for parallel image processing: an overview of the skipper project. Parallel Computing **28**(12) (December 2002) 1685–1708
22. William Thies and Michal Karczmarek and Saman P. Amarasinghe: StreamIt: A Language for Streaming Applications. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002) 179–196

23. Ian Buck and Tim Foley and Daniel Horn and Jeremy Sugerman and Kayvon Fatahalian and Mike Houston and Pat Hanrahan: Brook for GPUs: Stream Computing on Graphics Hardware. ACM Transaction on Graphics **23**(3) (2004) 777–786
24. William R. Mark and R. Steven Glanville and Kurt Akeley and Mark J. Kilgard: Cg: a System for Programming Graphics Hardware in a C-like Language. In: Proceedings of ACM SIGGRAPH 03, New York, NY, USA, ACM Press (2003) 896–907
25. Arcila, T., Allard, J., Ménier, C., Boyer, E., Raffin, B.: Flowvr: A framework for distributed virtual reality applications. In: 1ière journées de l'Association Française de Réalité Virtuelle, Augmentée, Mixte et d'Interaction 3D, Rocquencourt, France (November 2006)
26. Allard, J., Raffin, B.: Distributed Physical Based Simulations for Large VR Applications. In: IEEE Virtual Reality Conference, Alexandria, USA (March 2006)
27. Allard, J., Raffin, B.: A Shader-Based Parallel Rendering Framework. In: IEEE Visualization Conference, Minneapolis, USA (October 2005)
28. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
29. Molmar, S., Cox, M., Ellsworth, D., Fuchs, H.: A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications **14**(4) (1994) 23–32
30. Jos Stam: Stable fluids. In: Proceedings of ACM SIGGRAPH 99, New York, NY, USA, ACM Press (1999) 121–128
31. Franco, J., Boyer, E.: Exact Polyhedral Visual Hulls. In: Proceedings of BMVC2003. (2003)
32. Allard, J., Boyer, E., Franco, J.S., Ménier, C., Raffin, B.: Marker-less Real Time 3D Modeling for Virtual Reality. In: Proceedings of the Immersive Projection Technology Workshop, Ames, Iowa (May 2004)
33. Allard, J., Cotin, S., Faure, F., Bensoussan, P.J., Poyer, F., Duriez, C., Delingette, H., Grisoni, L.: SOFA: an Open Source Framework for Medical Simulation. In: Medicine Meets Virtual Reality (MMVR). (2007)