Performance Prediction for Distributed Virtual Reality Application Mappings on PC Clusters

Sylvain Jubertie, Emmanuel Melin
Université d'Orléans, LIFO

Rapport N^o **2007-03**
05/01/2007

Performance Prediction for Distributed Virtual Reality Application Mappings on PC Clusters

Sylvain Jubertie, Emmanuel Melin

Université d'Orléans — LIFO
BP 6759 — F-45067 Orléans cedex 2
{sylvain.jubertie | emmanuel.melin}@univ-orleans.fr

Abstract. Large and distributed Virtual Reality applications are often built from heterogeneous parallel codes. FlowVR offers a design model to easily compose a loosely coupled VR distributed application. FlowVR highly facilitates the development of this kind of applications making possible the description of their coupling and their mapping on the cluster nodes independently of their codes. Nevertheless, the choice of a good mapping is leaved to the developer skill. In an other hand, VR applications may include lots of different components and clusters may also be composed of numerous and heterogeneous nodes and networks. In this case it seems very difficult to map efficiently a distributed VR application onto a large cluster without tools to help the programmer to analyse his choices. The FlowVR design particularly does not propose any kind of performance model. In this paper we present an approach to determine performances of VR applications from a FlowVR network and from characteristics of a cluster. We also give some advices to the developer to design efficient FlowVR mappings. Since FlowVR model is very closed to underlying models of lots of VR codes, our approach can be useful for all designers of such applications.

1 Introduction

Today clusters are theoretically able to reach the performances needed by VR applications because they are extensible. This is the most interesting property because it doesn't limit the simulation complexity nor the number of data to consider. But clusters come with new programming problems : it is more complex to produce efficient applications on distributed memory architectures than on shared memory ones. Several communication libraries like MPI and PVM provide facilities, like point-to-point communications and synchronisations, to program this architectures efficiently. VR platforms were also ported to clusters to exploit their performances. For example the NetJuggler [9] environment permits to drive VR applications with parallel simulations and a distributed rendering. These approaches are very interesting but are limited to simple applications assumed to run on homogeneous clusters. The model behind NetJuggler is also too synchronous because the rendering rate is too dependant of the simulation rate [3].

Consequently we should add more asynchronism between the different application parts. For example the interaction codes and the simulations codes should be connected together but they should not be synchronized if we want the application to keep an interactive behaviour because the simulations have often lower frequencies than haptic devices. In this case we want the simulation to receive interaction data asynchronously even if some of them are lost. In the same way, we need asynchronism with the visualization. When the user wants to change its point of view, the visualization should change it interactively without waiting for the simulation. In both cases the application parts are able to communicate without waiting for the other. We say that they are linked with *greedy communications*.

Once we have described how to synchronize the different parts of the application then we could map these parts on the cluster processors. A lot of choices are possible depending of the underlying nature of the cluster which may be composed of heterogeneous nodes, peripherals and networks. This mapping is not straightforward and may have consequences on the application performance. For example mapping several parts on the same node may decrease latency between them by avoiding network communications but it introduces concurrency which could reduce computation times. In the case of a distributed simulation, increasing the number of processor used may decrease the computation time but it may increase the network load. Consequently we need a framework that eases mapping operations by catching the parameters of each application part and abstracting the architecture. This framework should also be associated to a performance model to tune efficient mappings on distributed architectures.

The FlowVR library[2][4] was created to ease the development of VR applications and permit greedy communications. The main idea is to abstract the application from a specific communication and synchronisation scheme. Consequently the FlowVR framework enables also the building of VR applications independently of the underlying architecture.

But this framework doesn't give a way to obtain the best application mapping on a given cluster nor any kind of performance information. Without such informations the developer should use its experiment and test several configurations to find a good mapping. But this task may become too complex for applications with lots of modules on heterogeneous cluster like for example the application presented in [4] which integrates 5000 different objects.

In this paper we propose several technics that could help the developer in his mapping choices. The first goal is to associate performance informations to a specific mapping. Then we will be able to compare several compatible mappings and to give the best one. We also need to warn the developer when he makes choices incompatible with the application requirements, for example associating a module to a node with insufficient memory, or a visualization module to a node not connected to the right display.

2 Performance models for parallel programming

We study the existing parallel models associated with cost models.

2.1 PRAM : Parallel Random Access Machine

The PRAM model defines a synchronous multi-processor machine accessing a shared memory. Each step in the computation is made synchronously by each processor which reads or writes data in the shared memory. A communication or computation step is assumed to take unit time.

On distributed memory architectures, such as clusters, this model is unrealistic because it assumes that all processors work synchronously and it doesn't take care of the communication cost, which is not negligible compared to the computation cost. Moreover, for distributed Virtual Reality applications we need heterogeneous and asynchronous computations on the different nodes of the cluster and the PRAM is not well adapted for it.

2.2 BSP : Bulk Synchronous Parallel

In the BSP model [13] a computation is defined by a sequence of supersteps : asynchronous computation followed by a global communication step and a synchronisation barrier. The cost of a BSP algorithm is defined by the input size and several architectural parameters. This model has interesting properties : it is architecture independent and the performance of a BSP program is predictable on a given architecture.

But this model imposes to write programs following a particular scheme : supersteps, which does not fit the heterogeneous nature of VR applications. All the supertsteps must wait for the longest one to complete before entering the global barrier. This leads to inefficiency for applications with not well balanced supersteps and more specifically for the VR applications.

The BSPWB model[12], a BSP Without Barrier, proposes a generalisation of the superstep to a Message step (M-step) : a local computation followed by a data emission and reception. Global barriers are removed because processors may be in different M-steps at the same time. But two M-steps could only communicate if they are adjacent which limits the possible asynchronism between M-steps. Moreover greedy communications are not possible in this model.

2.3 LogP

The logP model [6] was developed specifically for distributed memory architectures with point-to-point connections. The goal is to obtain a more realistic cost than the PRAM model by taking into account the communication cost. This model is asynchronous to reflect the intrinsic nature of distributed memory architectures and to obtain better performances than the BSP model without the need of expensive global synchronizations.

The LogP model uses four parameters to catch the main characteristics of distributed memory architectures : the communication delay (L), the communication overhead for the management of the network interface (o), the communication bandwidth (g) and the number of processors (P).

A distributed computation is represented in LogP by a directed acyclic graph, each node represents a local computation on a processor and each edge a point-to-point communication. Two local computations are asynchronous if there is no path between them. The execution follows the communication scheme of the dependency graph. Performances are obtained by computing the longest path in the graph.

This model permits to obtain optimal algorithms for simple problems [6] but is not well adapted to more complex applications [10]. VR applications need also asynchronism even if a dependency (a path in the LogP graph) exists between two computations. For example even if a simulation provides data at a low frequency, the rendering operations should not be tightly synchronized to it. In this case we need a greedy communication which is not available in LogP. This model also assumes that the cluster nodes are identical and doesn't take care of heterogeneous configurations.

2.4 Athapascan

Athapascan [5] is a C++ library designed for explicit parallel programming using threads. The parallelism is expressed in the code by explicit remote procedure calls to threads which are synchronized and communicate through a shared memory. The dependencies between threads are expressed by a graph which is built by following the sequentially structure of the code. Each node represent a different task to execute and each path a data dependency between two tasks. When a task creation is called then a node is added to the graph. If this task contains a reference to a variable shared by a previous task in the graph then it is linked to this task. Like in the LogP model, the cost of an execution is defined by the length of the longest path in the dependency graph.

In the general case, the graph is built at the beginning of the execution. But for VR applications which include infinite loops the graph has an infinite size and cannot be built. In this case it is possible to limit the graph construction at a certain size. For example [7] present a cloth simulation using the Athapscan library where each simulation step is associated to a new graph. The tasks are then spawn on the different processors by a scheduler following a mapping policy based on heuristics.

This model doesn't allow asynchronous communications between two threads because they are executed following the sequential structure of the code and synchronized by their access to a shared variable. If we consider the heterogeneous nature of a VR application and of the architecture then it seems difficult to find an efficient mapping policy.

2.5 Performance model for the SCP language

SCP [11] is a SPMD language based on structured dependencies directed by the syntax order. A SCP program could be viewed as a directed acyclic graph built by following the sequential instructions order. Each path in the graph represents a possible execution of the program. The cost of a SCP program corresponds to the longest path in the graph depending of an initial context.

In this model the syntax gives the synchronization scheme and the order between the different application parts while in VR applications we want to define the synchronization scheme independently of the syntax. Moreover a SCP application is seen as a single code while a VR application is built from heterogeneous codes.

3 The FlowVR model/design

FlowVR is an open source middleware dedicated to distributed interactive applications and currently ported on Linux and Mac OS X for the IA32, IA64, Opteron, and Power-PC platforms. The FlowVR library is written in C++ and provides tools to build and deploy distributed applications over a cluster. We turn now to present its main features. More details can be found in [2].

A FlowVR application is composed of two main parts, a set of modules and a data-flow network ensuring data exchange between modules. The user has to create modules, compose a network and map modules on clusters hosts.

Data exchange Each message sent on the FlowVR network is associated with a list of stamps. Stamps are lightweight data that identify the message. Some stamps are automatically set by FlowVR, others can be defined by users. Basic stamps are a simple ordering number or the module id of the message source. Stamps makes possible to perform computations or routing on messages without having to read the message content nor transmit it to avoid useless data transfers on the network.

Modules Modules encapsulate tasks and define a list of input and output ports. A module is an endless iteration reading input data from its input ports and writing new results on its output ports. A module uses three main methods:

– The *wait* function defines the beginning of a new iteration. It is a blocking call ensuring that each connected input port holds a new message.
– The *get* function obtains the message available on a port. This is a non-blocking call since the *wait* function guarantees that a new message is available on each module ports.
– The *put* function writes a message on an output port. Only one new message can be written per port and iteration. This is a non-blocking call, thus allowing to overlap computations and communications.

Note that a module does not explicitly address any other FlowVR component. The only way to gain an access to other modules are ports. This feature enforces possibility to reuse modules in other contexts since their execution does not induce side-effect. An exception is made for *parallel modules* (like MPI executables) which are deployed via duplicated modules. They exchange data outside FlowVR ports, for example via MPI message passing but they can be apprehended as one single logical module. Therefore parallel modules do not break the FlowVR model.

The FlowVR Network The *FlowVR network* is a data flow graph which specifies connections between modules ports. A connection is a FIFO channel with one source and one destination. This synchronous coupling scheme may introduce latency due to message bufferization between modules. This may induce buffer overflows. To prevent this behavior, VR applications classically use a *"greedy"* pattern where the consumer uses the most recent data produced, all older data being discarded. This is relevant for example when a program just needs to know the most recent mouse position. In this case older positions are usefullness and processing them just induces extra-latency. FlowVR enables to implement such complex message handling tasks without having to recompile modules. To perform these tasks FlowVR introduces a new network component called *filter*. Filters are placed between modules onto connection and has an entire access to incoming messages. They have the freedom to select, combine or discard messages. They can also create new messages. Although FlowVR model does not enforce the semantics of filters, the major pattern in VR applications is the *"greedy"* filter. loaded by FlowVR daemons. The goal is to favor the performance by limiting the required number of context switches. As a concequence the CPU load gererated by greedy filters can be considered as negligible in front of module load.

A special class of filters, called *synchronizers*, implements coupling policies. They only receive/handle/send stamps from other filters or modules to take a decision that will be executed by other filters. This detached components makes possible a centralised decision to be broadcasted to several filters with the aim to synchronize their policies. For example, a greedy filter is connected to a synchronizer which selects in its incoming buffer the newest stamp available and sends it to the greedy filter. This filter then forwards the message associated with this stamp to the downstream module.

The FlowVR network is implemented by a daemon running on each host. A module sends a message on the FlowVR network by allocating a buffer in a shared memory segment managed by the local daemon. If the message has to be forwarded to a module running on the same host, the daemon only forwards a pointer on the message to the destination module that can directly read the message. If the message has to be forwarded to a module running on a distant host, the daemon sends it to the daemon of the distant host. Using a shared memory enables to reduce data copies for improved performances. Moreover a filter does not run in its own process. It is a plugin loaded by FlowVR daemons. The goal is to favor the performance by limiting the required number of context switches. As a concequence the CPU load gererated by the FlowVR network management can be considered as negligible compared to module load.

FlowVR does not include performance informations to help the modules placement. This tasks is leaved to the developer.

4 Predicting/Determining performances

In this section we present our approach to determine performance of an application mapping on a given cluster. Our main goal is to provide an algorithm which takes as parameter a FlowVR network and the cluster caracteristics and return performance informations on the modules. This way we will be able to predict performance of different mappings without deploying the application on the cluster. We also want to warn the developer if its mapping is not compatible with the FlowVR model.

Before dealing with application performances we must answer the following questions : What means the "best mapping" for a VR application ? Is it the one which gives the best performances for all the application's modules ? What are the criteria to optimize ? In fact it is really difficult to answer this questions because optimizing part of an application may have unexpected consequences on the other parts. Some modules are more critical than others in the application. For example a visualization module should have a framerate around 20-30 fps to ensure interactivity, if it is lower then even if the other modules perform well the user will not have an interactive feeling. On the other side, if the framerate is too high some frames are not viewed by the user. In this case too much resources are provided to the module which may be usefull to other modules. Consequently, we cannot determine an absolute global performance for an application. But the user could specify his needs and his requirements on some modules. Thus we propose in our approach to give the user the means to choose if a mapping is compatible with its expectations.

To compute performances of a mapping we need the following informations for each module m :

- its nature : CPU or I/O-bound. The developer should identify his modules. If the modules is performing lots of I/O operations, like an interaction module, then it is I/O-bound else it is CPU-bound when it performs computations, like a simulation module. This helps to determine the scheduler policy in the case of concurrent modules.
- its computation time $T_{comp}(m)$ on the host processor. It is the time a module needs to perform its computation when there is no concurrent modules on the processor.
- its processor load $LD(m)$ on the host processor. It corresponds to the percentage of the computation time used for the computation and not for waiting I/O operations.

4.1 Computing iteration times

The *iteration time* T_{it} of a module is defined by the time between two consecutive calls to the wait function. This includes the computation time T_{comp} and the time T_{wait} the module is locked in the *wait* function.

When several modules are mapped on the same processor then their respective executions could be interleaved by the scheduler. Then a module could have a greater computation time. We called it the *concurrent computation time* T_{cc}. It is determined from the scheduler policy.

The developer can specify its own policy according to the behaviour of his operating system scheduler. In this paper we choose to modelize a Linux scheduler policy [1] which gives priority to modules depending of the time they wait for I/O operations. For the sake of clarity we restrict our model to single processor computers. We distinguish two cases depending of the concurrent module loads sum :

- if it is lower than 100% then the scheduler gives the same priority to each module. In this case the processor is able to handle the computation of each modules without interleaving their executions and for each module m we have :

$$T_{cc}(m) = T_{comp}(m) \tag{1}$$

Note that if there is only one module mapped on a processor then its concurrent computation time is also equal to its computation time.

– if it is greater than 100% then the priority is given to I/O-bound modules depending of the time they wait for I/O. In this case the execution of the CPU-bound modules are interleaved. First we compute the time $T_{I/O}$ each module waits for I/O according to their respective loads and we also add the time T_{wait} it blocks in the *wait* function :

$$T_{I/O}(m) = T_{comp}(m) \times (1 - LD(m)) + T_{wait} \tag{2}$$

We order the modules by the time they are waiting. We give the module $m1$ which waits the most a processor load $CPULD$ equals to its load : $CPULD(m1) = LD(m1)$. Then for the second module $m2$ we give it a processor load equals to $CPULD(m2) = (1 - CPULD(m1)) \times (LD(m2))$. We repeat this process for all the I/O modules according to the following equation :

$$CPULD(m) = (1 - \sum_{T_{I/O}(m_i) > T_{I/O}(m)} CPULD(m_i)) \times LD(m) \tag{3}$$

Then the rest of the processor load is shared by the n concurrent modules with $T_{I/O} = 0$:

$$CPULD(m) = \frac{1 - \sum_{nature(m_i)=I/O} CPULD(m_i)}{n} \tag{4}$$

According to the processor load of each module we could now compute their concurrent computation times :

$$T_{cc}(m) = T_{comp}(m) \times \frac{LD(m)}{CPULD(m)} \tag{5}$$

The iteration time depends also of synchronizations between the module and modules connected to its input ports. For each module m we need to define its input modules list $IM(m)$ as the set of modules connected to its input ports. We distinguish two subsets of $IM(m)$: $IM_s(m)$ and $IM_a(m)$, which contain respectively the modules connected to m synchronously with FIFO connections and asynchronously with greedy filters. If we consider a module m with $IM_s(m) \neq \emptyset$ then this module should wait until all the modules in $IM_s(m)$ have sent their messages. This means that it must wait for the slowest module in $IM_s(m)$ but only if it is slower than it. Consequently the iteration time T_{it} of a module m is given by the following formula :

$$T_{it}(m) = max(T_{cc}(m), max(T_{it}(m_{in}) | m_{in} \in IMs_m)) \tag{6}$$

If the module m has no connected input ports ($IM(m) = \emptyset$) or if it has only asynchronous inputs ($IM_s(m) = \emptyset$) then there is no constraint on the iteration time and we could simplify the formula 6. In this case the iteration time T_{it} is equal to the concurrent computation time :

$$T_{it}(m) = T_{cc}(m) \tag{7}$$

Else if $IM_s(m) \neq \emptyset$ and $IM_a(m) \neq \emptyset$ then asynchronous communications have no effect on the module iteration time and we could apply the equation 6. Consequently in the general case where $IM_s(m) \neq \emptyset$ if the receiver has a lower iteration time than the emitter then its iteration time is aligned on the emitter's one. Note that if $T_{cc}(m) > max(T_{it}(m_{in}) | m_{in} \in IMs_m)$ then m is slower than at least one of its input modules. In this case messages sent by these modules are accumulated in the daemon shared memory until it is full, leading to a buffer overflow. Our analysis allow to detect such cases.

The formula 7 shows that the iteration time of modules without synchronous communications on their input ports do not depend of the previous modules. Consequently we could remove greedy communications from the application graph to study the effect of synchronizations on modules iteration times. The resulting graph may stay connected or may not be a connected anymore. In this last case we obtain several set of modules called components.

The next step of our approach is to consider each component without taking care of the concurrency with modules from other components. Then we try to determine the iteration and computation times of each module in a given component . Finally we study the consequence of concurrency between each component.

The starting points of our study of each component are greedy modules and modules sending a first message before calling the *wait* function. This last case is necessary to start an application with a module in a cycle. We called this kind of modules *cycle starting modules.*

Greedy modules may have different iteration times. If we have several greedy modules in the same component then it means that it exists a module in the component which as more than one greedy module as a predecessor. This module should have the iteration time of the slowest greedy module. Because we could not guarantee that it greedy predecessors have the same iteration time, the module may accumulate messages from the fastest one, leading to a buffer overflow. Consequently we should recommend to only construct FlowVR applications with one greedy module by component.

Following the same way, if we consider a greedy module and a cycle starting module in the same component then we also have the same problem. Consequently the cycle must have the greedy module as predecessor.

Moreover if the component contains only cycle starting modules, only cycle of them should be the common predecessor of the others. So we could consider it as the only starting module because it imposes its iteration time to all modules.

Consequently for each component we must have only one starting point as predecessor of all modules : a greedy or a module within a cycle. If there is other cycle starting modules then we could ignore them because the iteration time of all modules in the component shoud be equal to the greedy module iteration time.

The iteration time is common to all the modules in a component and is equal to the iteration time of the starting module. We compute the iteration time of the starting module m_s according to its nature and to the nature of its concurrent modules. If m_s has no concurrent modules then its iteration time is equal to its computation time according to equation 7. If all its concurrent modules are in the same component then $T_{cc}(m_s)$ may increase. It involves that the iteration time of each concurrent module should also increased by the same amount. In this case the computation of iteration times for each modules is interdependent. We could compute the new $T_{cc}(m_s)$ from equation 5 with the current iteration time of the other modules. This means that we are giving less priority to m_s and we obtain the maximum $T_{cc}(m_s)$ which is equal to the maximum T_{it}. Then we apply the equation again but with the new T_{it}. In this case we have the lowest value for $T_{cc}(m_s)$. We could continue to perform this operation but it is not guarantee to converge. If m_s is CPU-bound then it always have the lowest priority and we could apply the preceeding process until convergence. Otherwise if m_s is a I/O-bound module its T_{cc} may oscillate because its priority could change in comparison with the other modules priority. Consequently we could have a variable iteration time of a starting point and all the modules in the same component. This could affect the interactivity of the application so we recommend the developer to avoid this configuration in its mappings.

In the general case starting points may have concurrent modules from other components. Then we need to compute the iteration time of starting points of the other components. This process may also be interdependent for example if we have two components with starting points mapped on the same processor that modules from the other component. To detect interdependencies we add edges from modules to starting modules mapped on the same processor. We called this graph the *interdependency graph*. If we have cycles in the resulting graph then we have interdependencies. Then we could also apply the process described above with the same consequences. If there is no cycles in the interdependency graph then we have at least one starting module which does not depend of the other components.

For the other non-starting modules of each component we could compute now their concurrent computation time with the equation 5. Once we have this information we could determine for each module m if $T_{cc} > T_{it}$. In this case we could detect a buffer overflow.

The study of this interdependency graph may help the developer to avoid problematical mappings. It also shows the consequences of concurrency in the whole application. For example if we want to improve performances of a particular component then we should avoid to map its starting module with modules from other components.

At this step we have the iteration times of all the modules in the application. This is important because the developer is able to see if the mapping corresponds to the performances he expected. For example he knows if a simulation module could run at the desired frequency because the module frequency $F(m)$ is equals to the inverse of the iteration time $T_{it}(m)$:

$$F(m) = \frac{1}{T_{it}(m)} \tag{8}$$

If this is not the case then he should consider another mapping. Else we should continue to study this mapping and consider communications between modules over the network.

4.2 Communications

We now study the communications defined by an application mapping between the different FlowVR objects.

We assume that synchronizer communications are negligible compared to the other communications. Indeed even if synchronizations occured at the frequency of the fastest module involved in the synchronizations, they required only few stamps informations compared to the message size sent by this module. We also assume that communications between objects mapped on the same processor are free. In this case the messages are stored in the shared memory and a pointer to the message is given to the receiving object.

Computing requested bandwidths We begin our study with a traversal of the application graph to determine for each filter f its frequency $F(f)$ and the volume of data on its output ports. We start our traversal with starting modules and follow the message flow in the graph. When we consider a filter f then we assign it a frequency $F(f)$ according to its behaviour. For example a greedy filter f_{greedy} sends a message only when the receiving module m_{dest} asks it for a new data. Thus we have $F(f_{greedy}) = F(m_{dest})$. A broadcast filter $f_{broadcast}$ processes messages at the same frequency of its input module m_{src}. In this case we have $F_{broadcast} = F_{m_{src}}$. For each filter we also compute the size of messages on its output port. For example a binary scatter filter takes as input a message of size s and splits it into two output messages of size $\frac{s}{2}$ on each output port.

We assume a cluster network with point-to-point connections in full duplex mode, with a bandwith B and a latency L. Communications are handled by a dedicated network controller without CPU overload.

The application graph G is defined by a set of vertices V for each FlowVR objects and a set of directed edges E representing communications between objects output ports and input ports. Then we add additional edges to modelize communications out of the FlowVR communication scheme, for example communications between several instances of a MPI module. For each instance we add output edges and input edges respectively to and from other MPI instances. We define for each edge e :

- a source object $src(e)$ which is the FlowVR object sending a message through e.
- a destination object $dest(e)$ which is the FlowVR object receiving message from $src(e)$.
- a volume $V(e)$ of data sent through it. It is equal to the size of the message sent by $src(e)$.

We provide a function $node(o)$ which returns the node hosting a given FlowVR object o.

Then we are able to compute the bandwidth BW_s needed by a cluster node n to send its data :

$$BW_s(n) = \sum_{node(src(e))=n}^{node(src(e))\neq node(dest(e))} V(e) \times F(src(e)) \tag{9}$$

We could also determine the bandwidth BW_r needed by a cluster node n to receive its data :

$$BW_r(n) = \sum_{node(dest(e))=n}^{node(src(e))\neq node(dest(e))} V(e) \times F(src(e)) \tag{10}$$

If for a node n $BW_s(n) > B$ then messages are accumulated in the shared memory because the deamon is not able to send them all. Consequently we could detect a buffer overflow. If $BW_r(n) > B$ then there is too much data sent to the same node, leading to contention.

These informations give the developer the ability to detect network bottlenecks in its mappings. Then he could solve them for example by reducing the number of modules on the same node and changing the communication scheme.

Computing latencies The *latency* is defined between two FlowVR objects. It represents the time needed by a message to be propagated from an object to another throught the communication network. In VR applications the latency is critical between interaction and visualization modules : the consequence of a user input should be visualize within the shortest possible delay to keep an interactive feeling.

We determine the latency between two objects o_1 and o_2 from the path P between them. A path contains a set of FlowVR objects and edges between them. The latency is obtained by adding the iteration time $T_it(m)$ of each module m in the path to communication time needed for each edge in the path :

$$L(m_1, m_2, P) = \sum_{m\in P} T_{it}(m) + \sum_{src(e)\neq dest(e)} \frac{V(e)}{B} + L \tag{11}$$

With this information the user is able to detect if the latency of a path is low enough for interactivity. If the latency is too high then the developer should minimize it by increasing frequencies of modules in the path or by mapping several modules on the same node to decrease communications latencies.

4.3 Test Application

Our test application is based on a generic parameterizable FlowVR module which could simulate lots of different kinds of modules.

We first verify that our scheduler policy is realistic. Tests are run on a Pentium IV Xeon 2.66Ghz with Linux as operating system. We use the revision 2.6.12 of the Linux kernel. We run three modules on the same processor and we compare their measured computation times with their concurrent computation times predicted with the equation 5. For each module we define its computation time T_{comp} and its waiting time $T_{I/O}$. Modules are not synchronized ($T_{wait} = 0$) so we could compute their loads by inverting the equation 2 :

$$LD(m) = \frac{T_{comp}(m) - T_{I/O}(m)}{T_{comp}(m)} \tag{12}$$

Results presented in figure 1 show for the three modules their respectives computation times T_{comp}, waiting times $T_{I/O}$, predicted concurrent computation times T_{cc} and their real computation times T_{real}.

In the first test we consider 3 I/O-bound modules. Then we compute their respective loads with equation 12. We predict 17% for module 1, 20% for module 2 and 25% for module3. The total load is equal to 62% and is lower than 100% so for each module we have $T_{cc} = T_{comp}$. For the other tests we have at least one CPU-bound module so the total load is higher than 100%. We order IO-bound modules according to the time $T_{I/O}$ from the module which wait the most to the module which wait the less. We compute their respectives concurrent loads and we split the remaining processor load available between the CPU-bound modules. Then we apply equation 5. Results show that our policy is correct so we could now apply our approach on a real application.

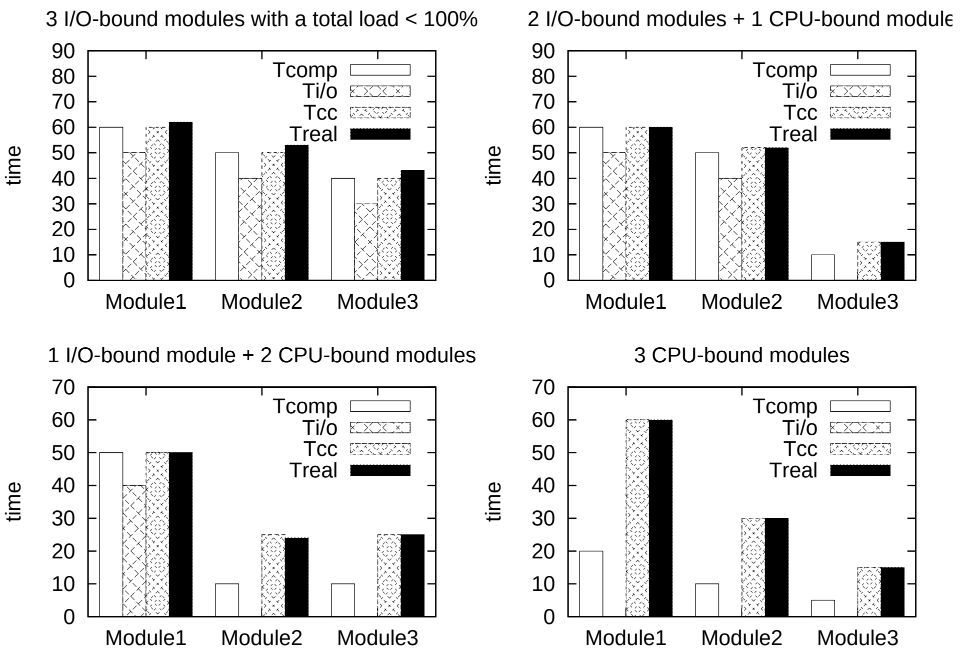

Fig. 1. Concurrent computation time predictions

4.4 Fluid-Particle Simulation

In this section we apply our approach on a mapping of a real FlowVR application on a cluster.

The FluidParticle application consists of a flow simulation giving a velocity field to a set of particles. Each particle's new position is then computed by adding the correspondent velocity vector in the field to its current position. The particles are then rendered on the screen using a point-sprite representation. The dataflow of the application is presented in figure 2. The visual result (figure 4) is a set of particles initially mapped regularly in a fluid which are then pushed by two flows. We could observe typical fluid phenomena like vortices.

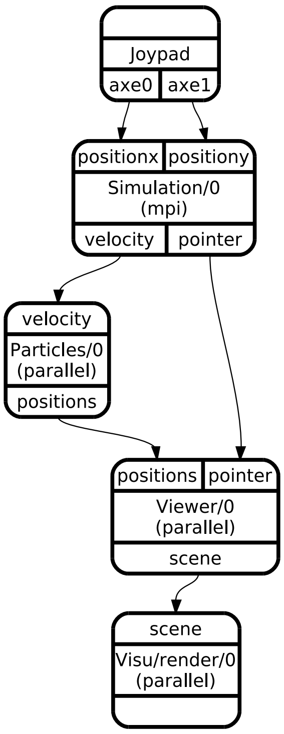

Fig. 2. Dataflow of the FluidParticle application

The cluster is composed of eight single processor nodes connected with a gigabit network.

Now we present each module of our application :

– the fluid simulation : this is a parallel version of the Stam's fluid simulation [14] based on the MPI library [8]. This module is CPU-bound and has a load of 97%. In our test we consider a global grid of 400x400 cells splitted in four 200x200 local grids across the nodes. For each module instance m^i_{sim} we have $T_{comp}(m^i_{sim}) = 200ms$. During each iteration each module send seven times its border with its neighboors.
– the particle system : this a parallel module. Each instance stores a set of particles and moves them according to forces provided by the fluid simulation. The particle set is then sent on the output port. This module is CPU-bound, has a load of 97%. We consider a gloabl set of 640.000 particles splitted up into two

local sets (one by module instance). For each module instances m^i_{part} we have $T_{comp}(m^i_{part}) = 20ms$.

- the viewer : it converts the particles positions received on its input port into graphical data which are then sent through its scene output port. It is a CPU-bound module with a load of 97%. It takes $T_{comp}(m_{view}) = 28ms$ to computes the 640.000 particles from the particle system.
- the renderer : it displays informations provided by the viewer modules. There is one renderer per screen. We want to visualize the particles on a display wall of four projectors connected to nodes 1, 2, 3 and 4. Renderer modules are synchronized together with a swaplock to provide a coherent result on the display. Each renderer module is CPU-bound with a load of 97%. It takes $T_{comp}(m_{rend}) = 57ms$ to display 640.000 particles.
- the joypad : it is the interaction module which allow the user to interact with the fluid by adding forces. It is mapped on node 6 where the joypad device is connected. This module is I/O-bound, has a load under 1% and a computation time $T_{comp} < 1ms$. Consequently according to our model it should not involve performance penalties on the other modules.

The figure 5 shows the mapping we have chosen. The fluid simulation is distributed on the four last nodes (5, 6, 7, 8) and receives interaction from the joypad module through a greedy communication. The result of the simulation is gathered using binary gather filters then it is broadcasted to the two particles modules on nodes 3 and 4. Then we have one viewer module on node 2 which transform the particle positions gathered with a gather filter into visual informations. Finally the visual informations are broadcasted to renderer modules mapped. Each renderer module displays a quarter of the full scene on a display wall.

We map the renderer modules on the nodes 1, 2, 3, 4 because We want to visualize the result of the application on the display wall connected on theses nodes. The joypad is connected to the node 6 so we must map the interaction module on this node. Then we are free to map the other modules on the cluster. We have choosen this mapping to avoid concurrency between the simulation and the other modules because we want to have the maximum frequency for the simulation. So the particles and the viewer modules should be placed on nodes 5, 6, 7 or 8.

We start to calculate iteration times of each modules. For the simulation modules we have $IM_s(m^i_{sim}) = \emptyset$ and they are not in concurrency with other modules so we have for each instance $T_{it}(m^i_{sim}) = T_{cc}(m^i_{sim}) = T_{comp}(m^i_{sim}) = 200ms$. The particles modules are synchronized with the simulation consequently we have : $T_{it}(m^i_{part}) = T_{it}(m^i_{sim}) = 200ms$. Moreover renderer modules 3 and 4 are mapped on the same processors and run asynchronously because $IM_s(m^i_{render}) = \emptyset$. In this case the priority is given to the particles modules which wait the most and we could compute their concurrent computation time : $T_{cc}(m^i_{part}) = T_{comp}(m^i_{part}) = 20ms$. Then we repeat the same process for the viewer module and we obtain $T_{cc}(m_{view}) = T_{comp}(m_{view}) = 28ms$. Finally we consider the renderer modules which are synchronized together with a swaplock and are distributed on four nodes so the global iteration time is given by the highest iteration time each module could have on the four processors. We compute the concurrent computation time for each module instance : on node 1 and 2 the particle module takes a $CPULD(m^{1,2}_{part}) = 0.10$ so we have $CPULD(m^{1,2}_{rend}) = 0.90$ then $T_{cc}(m^{1,2}_{part}) = 20ms$ and $T_{cc}(m^{1,2}_{rend}) = T_{comp}(m^{1,2}_{rend}) \times 1.10 = 63ms$. Following the same method on node 3 with the viewer module we obtain : $T_{cc}(m^1_{view}) = 28ms$ and $T_{cc}(m^3_{rend}) = T_{comp}(m^3_{rend}) \times 1.14 = 66ms$. Consequently we have $T_{it}(m^i_{rend}) = T_{cc}(m^{1,2}_{rend})$. Results are shown in figure 3 and we could compare the concurrent computation time predicted to the one we measured. We notice that our prediction is similar to the real result for all the modules.

Then we study the application network to determine possible bottlenecks. For each filter we compute its frequency and the volume of data on its output ports. Then we apply the formula 9 and 10. The MPI simulation module is mapped on nodes 5, 6, 7 and 8. Each instance of this module sends its border to his neighboors seven times per iteration for a total of $22.4kB$ per iteration. So we add edges in the communication graph according to this MPI communication scheme. We show that the fluid simulation produces 6.4MB/s which are then broadcasted to the particles modules. The broadcast filter on node 3 needs to send a total of 12.8MB/s. The particle modules send 25.6MB/s from node 1 to the viewer module on node 2. Then the same amount of data is sent to each renderer module through a broadcast tree. The viewer should only send 76.8MB/s to the renderer modules on nodes 1, 3 and 4 because we have the viewer module and one renderer module on the same node. In all the previous cases the gigabit network is able to handle this communications. We should not overload the network. Therefor we are also able to predict the behaviour of each application module and to determine possible network contentions.

We have studied different mappings of this application. For example we have tried to map the viewer module on node 3 with a particle module instance and a renderer module instance. According to our prediction, we observe in this case a decrease of the renderer modules frequency from 15 to 9fps. In this case the viewer module has a higher concurrent computation time but its frequency stays at 5Hz due to the constant iteration time. With this mapping we keep the same interactive feeling because the renderer modules frequency is higher than the viewer module frequency. We could also predict that 50% of the processor load is not used on node 2, 3, and 4. This information might be usefull if we plan to add other modules to the application. We have also considered only two instances of the renderer module on nodes 1 and 2, and the particles and viewer modules on node 3 and 4. In this case we have predicted a network overload because the viewer is not able to send more than 100MB/s through the network to the renderer modules. We have tried this mapping and we have obtained a buffer overflow on the emitter node. Note that our analysis permits to detect buffer overflows related to network overloads as well as related to differences of frequency between synchronous modules.

We could map the viewer module on node 1 to solve this problem. Another solution is to instanciate one more viewer module on node 4 to split the particles set in two. Then we connect each particles module instance to the viewer module on the same processor.

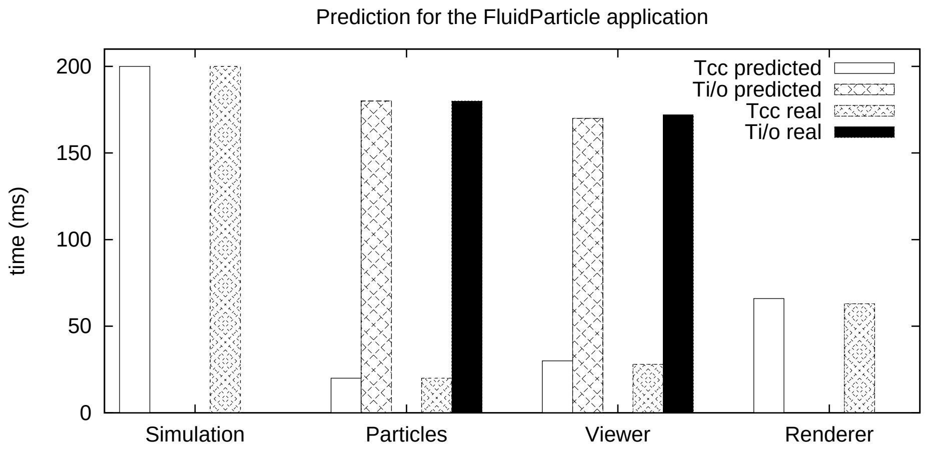

Fig. 3. Performance prediction on the FluidParticle application

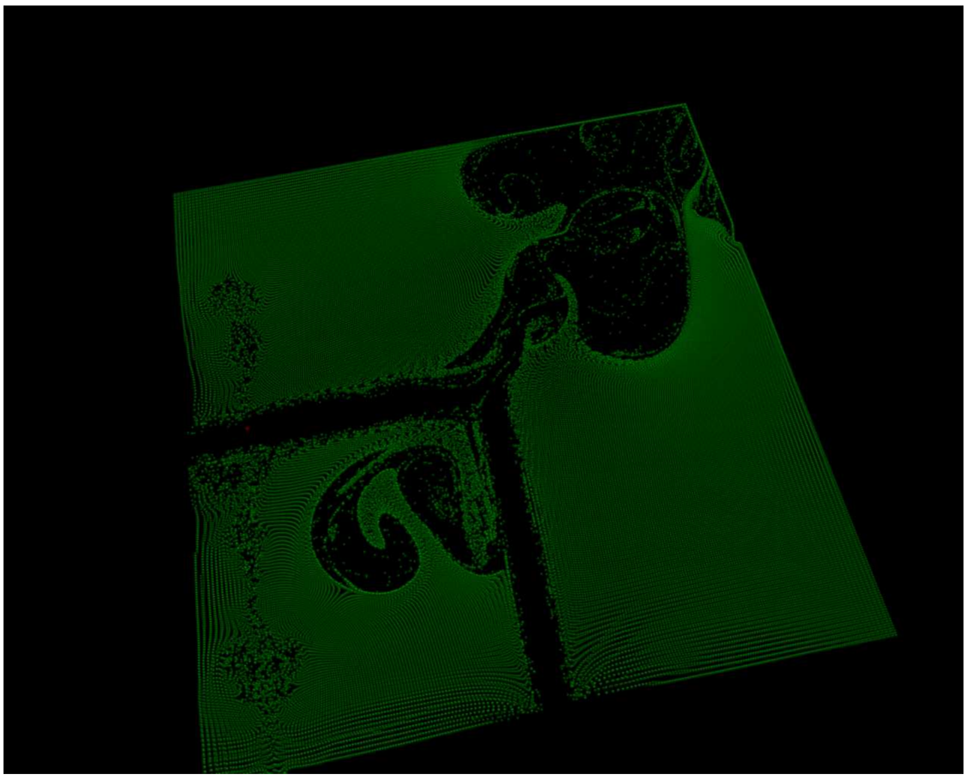

Fig. 4. Visualization of particles

5 Conclusion

We have shown that our approach can predict performances of applications from their mappings. We are able to determine the frequencies of each module by taking care of synchronizations and concurrency between modules. Moreover we could also detect network bottlenecks and mappings not compatible with the user expectations. We also provide advices for the developer to build a correct mapping and avoid lots of issues.

This method also brings to the FlowVR model a mean to abstract the performance prediction from the code. The developer needs only sufficient informations on modules he wants to integrate in his application to be able to predict performances. Moreover it enforces the possibility to reuse modules on other applications. For example a developer could replace his simulation and adapt his mapping without having a deep knowledge of implementation details.

This approach is limited for the moment to homogenous single processor clusters connected with a single network. We plan to extend it to consider heterogeneous clusters, concurrency on SMP nodes, and multiple networks. The next step is to study assisted or automatic optimizations of mappings to help the developer. We plan to define constraint systems on a VR application and to process them using a solver.

References

1. J. Aas. Understanding the linux 2.6.8.1 cpu scheduler. 2005.

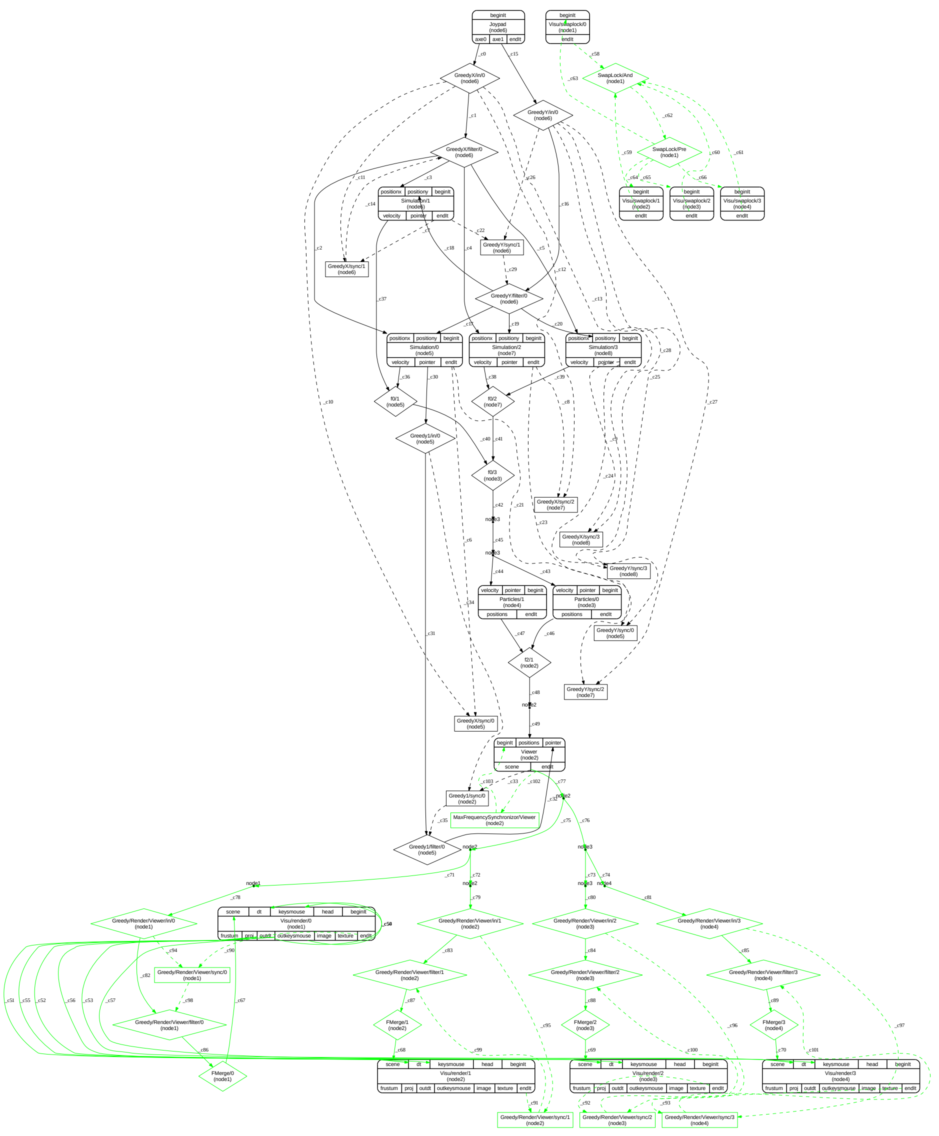

Fig. 5. The FluidParticle application graph

2. J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr: a middleware for large scale virtual reality applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.
3. J. Allard, V. Gouranton, E. Melin, and B. Raffin. Parallelizing pre-rendering computations on a net juggler pc cluster. In *Proceedings of the IPT 2002*, Orlando, Florida, USA, March 2002.
4. J. Allard, C. Mnier, E. Boyer, and B. Raffin. Running large vr applications on a pc cluster: the flowvr experience. In *Proceedings of EGVE/IPT 05*, Denmark, October 2005.
5. G. Cavalheiro, F. Galilee, and J.-L. Roch. Athapascan-1: Parallel programming with asynchronous tasks. In *Proceedings of the Yale Multithreaded Programming Workshop*, Yale, June 1998.
6. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicker. Logp: Towards a realistic model of parallel computation.
7. J.M.Vincent F. Zara, F. Faure. Physical cloth simulation on a pc cluster. In *Eurographics Workshop on Parallel Graphics and Visualization*, 2002.

8. R. Gaugne, S. Jubertie, and S. Robert. Distributed multigrid algorithms for interactive scientific simulations on clusters. In *ICAT*, 2003.
9. E. Melin J. Allard, V. Gouranton and B. Raffin. Parallelizing pre-rendering computations on a net juggler pc cluster. In *IPTS 2002*, 2005.
10. G. Loh. A critical assesment of logp: Towards a realistic model of parallel computation.
11. X. Rebeuf. *Un modle de cot symbolique pour les programmes parallles asynchrones dpendances structures.* PhD thesis, Universit d'Orlans, 2000.
12. J. L. Roda, C. Rodriguez, D. G. Morales, and F. Almeida. *Concurrency: Practice en Experience*, chapter Predicting the execution time of message passing models. 1999.
13. D.B. Skillicorn. Predictable parallel performance: The bsp model.
14. J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003.