4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

http://www.univ-orleans.fr/lifo

Mapping and Performance Prediction for Distributed Applications on Heterogeneous Clusters

Sylvain Jubertie, Emmanuel Melin
Université d'Orléans, LIFO

Rapport N^o **2007-07**
05/02/2007

Mapping and Performance Prediction for Distributed Applications on Heterogeneous Clusters

Sylvain Jubertie, Emmanuel Melin

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

Abstract. Distributed applications running on clusters may be composed of several components with very different performance requirements. The FlowVR middleware allow the developer to deploy such applications and to define communication and synchronization schemes between components without modifying the code. Whereas it eases the creation of mappings, FlowVR does not come with a performance model. Consequently optimization of mappings is left to the developer skills. This task seems difficult to perform when the number of components and cluster nodes grow. Moreover the cluster may be composed of heterogeneous nodes making this task even more complex. In this paper we propose an approach to predict performances of FlowVR distributed applications given a mapping and a cluster. We also give some advices to the developer to create efficient mappings and avoid configurations which may lead to issues. Since the FlowVR model is very closed to underlying models of lots of distributed codes, our approach can be useful for all designers of such applications.

1 Introduction

The different parts of an heterogeneous distributed application are difficult to map efficiently on a cluster. FlowVR [2] is a middleware which eases the coupling and the mapping of distributed applications. It was designed with distributed Virtual Reality applications in mind but its approach could be generalize to non-interactive applications. FlowVR allows the developer to define communications, synchronizations and mappings of the application parts without code modification. For instance the developer can implement outside of the code operations like data filtering, data sampling, collective communications schemes like broadcasts, etc. This fine control over data handling enables to take advantage of both the specificity of the application and the underlying cluster architecture to optimize the latency and refresh rates. But these fine control capabilities does not come with a FlowVR performance model. Consequently the optimization of the application performance is left to the developer skills.

When building a mapping on a cluster, the developer has to choose among lots of possibilities and only few of them are able to take the best advantage of the clusters performance. To find an efficient mapping, the developer should take care of communications, synchronizations and concurrency between parts of the application. He must introduce asynchronism to increase performances while ensuring that the application communication and synchronization schemes are coherent. Then he should have a precise idea of the performances of the different modules of the application. Some of them may be distributed on several nodes to increase their performances, but distributing modules could increase communications. Moreover the developer should also consider the behaviour of the operating system scheduler

to predict performances of concurrent modules. The scheduling policy depends of the nature of the module which could be I/O-bound if the module performs I/O operations or CPU-bound if the module performs pure computations. In the case of a I/O-bound module the processor is not used at full load because the module waits for I/O-operations to complete. We have in this category modules performing out-of-core operations or accessing databases. In the second case we have for example simulation modules which are only limited by the CPU performance. Some modules may be less critical than others for the global performance of the application. In some cases concurrency could benefit to performances whereas in other cases it could dramatically decrease performances. Last, the cluster could be composed of heterogeneous nodes and networks, increasing the difficulty of the developers task. When the number of application parts grows it seems difficult for the developer to analyse all the possible mappings without an efficient performance model.

Classical performance models for parallel codes are not well suited for FlowVR applications. The PRAM model involves synchronous computations and doesn't take care of communications between processors so it is not well adapted to clusters. The BSP model [6] catches the parameters of distributed architectures but applications should be structured in supersteps which are too synchronous. The LogP model [7] was developed specifically for distributed architectures but it does not allow complex asynchronous communications like data sampling. The same remark is addressed to the Athapascan model [5]. Moreover the mapping of each task is done according to a scheduling policy and the optimization of this policy is left to the developer. However, sampling communication are very important for real-time application since we can take advantage of a data exchange model based on sampling to improve latency. This is possible in the FlowVR model since the producer updates data in a shared buffer asynchronously read by the consumer. Consequently we need to define a new performance model which should integrate sampling communications and efficient mapping of the application.

We present in this paper an approach to determine performances of FlowVR applications from a given mapping and cluster characteristics. With theses performance informations the developer is able to compare several mappings and choose the one which offers the best performances. We also provide to the developer recommendations to built coherent mappings.

2 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.

3 Performance prediction

We turn now to present our approach to provide performance informations for a FlowVR application. This section is based on our preceding approach described in [1] and extends it to clusters with heterogeneous and SMP nodes and to clusters with multiple networks. Our goal is to give to the developer performance informations for a given application mapping on a given cluster. These informations are :

modules iteration times : it corresponds to the time between two consecutive calls to the *wait* function. The developer has to minimize iteration times to provide good performances to the application.

processor loads : this gives information on the efficiency of the mapping. If some processors are overloaded some modules may be mapped on different processors with lower loads.

communication volumes : we need to detect possible network bottlenecks.

latencies between modules : if we consider an interactive application then latency is critical.

We also provide a mean to detect problems which may occur in some particular configurations and may lead to buffer overflows. In these different cases we propose some solutions to solve these problems.

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 $LOAD(m)$ on the host processor. It corresponds to the percentage of the computation time used for the computation and not for waiting I/O operations.

Our approach is based on the traversal of the FlowVR application graph which is the extension of the data-flow graph of the application. We define the FlowVR application graph G_{appl} as a directed graph. Each vertex V in the graph represents a FlowVR object (a module, a filter or a synchronizer) mapped on a given node, and each directed edge E represents a connection from a FlowVR object to another on a given network.

From this graph we are able to give the developer informations on graph configurations which may lead to buffer overflows. Then we compute modules iteration times depending of synchronizations and concurrency with other modules. Finally, from modules iteration times and amount of data sent by each module we are able to compute volumes of data sent from each cluster node on the network. We also provide a mean to compute latencies between FlowVR objects.

3.1 Study of the application graph

For a module m we define its input modules $IM(m)$ 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 communications) and asynchronously (with greedy communications).

From the application graph we could detect some possible problems related to specific graph configurations. Indeed the FlowVR model does not restrict the communication and synchronization schemes and some particular configurations may lead to buffer overflow. A first study of the application graph could reveal such configurations. We note that this problem only happens with synchronous communications. Indeed a module m with only greedy communications connected to its input ports ($IM_s(m) = \emptyset$) should not wait for messages and their iteration time does not depend of the iteration time of their predecessors. Consequently greedy communications could not generate buffer overflows. From this remark we could remove the greedy communications from the graph G_{appl} and keep only synchronous communications to detect issues. The resulting graph is referred as G_{sync}. We note that G_{sync} may not be connected anymore and may be splitted into several components.

If a module m has two input modules $m_1 \in IMs(m)$ and $m_2 \in IMs(m)$ with respective iteration times $T_{it}(m_1)$, $T_{it}(m_2)$ then m must wait for the slowest one : $T_{it}(m) = max(T_{it}(m_1), T_{it}(m_2))$. Messages from the fastest predecessor module are accumulated in the receiving buffer leading to a buffer overflow. We could only guarantee that this situation never occurs if modules in $IM_s(m)$ are synchronized by a common predecessor module in G_{sync}. If this is not the case then we recommend the developer to add a synchronizer between modules in $IM_s(m)$. Consequently, for a module m, if $|IM_s(m)| > 1$ then modules in $IM_s(m)$ have a common predecessor module to synchronized them. In this case all modules in $IM_s(m)$ have the same iteration time $T_{it_{IM_s(m)}}$ and we have for the module m :

$$T_{it}(m) = T_{it_{IM_s(m)}} \tag{1}$$

The main consequence of this remark is that all modules in a connected component of G_{sync} must have at least a common predecessor. Indeed G_{sync} may contain cycles and if a common predecessor is part of a cycle then all modules in the cycle are also common predecessors. Consequently each component should begin with a single common predecessor or with a cycle. Moreover if we apply equation 1 for each module m in the component then we obtain that the iteration time should be the same for all the modules in the component and is equal to the iteration time of the predecessor pm :

$$T_{it}(m) = T_{it}(pm) \tag{2}$$

The next step is to compute the iteration time of the common predecessors but the iteration time also depends of concurrency between modules.

If a common predecessor pm has no concurrent module then we have :

$$T_{it}(pm) = T_{comp}(pm) \tag{3}$$

If several modules are mapped on the same processor then the scheduler may interleave their executions. Consequently concurrent modules could have a greater computation time which we called the concurrent computation time T_{cc}. In this case we need to compute T_{cc} for each predecessor which depends of modules mapped on the same processor.

To represent dependencies between predecessors and other modules we add directed edges in G_{sync} from concurrent modules of a component to predecessor modules. We obtain the graph G_{dep}. In this graph different component of G_{sync} may be

joined if modules of different components are mapped on the same processor than predecessors.

If we detect cycles in G_{dep} with predecessors from different component in G_{sync} then we have an interdependency between these predecessors. In this case the iteration time of each predecessor depends of the iteration times of all the other predecessors in the cycle. Moreover if we consider SMP nodes then the scheduler may change the mapping and the priority of concurrent modules. Consequently in this case it seems difficult to provide accurate prediction. We could compute iteration time of a predecessor without taking care of iteration time of modules mapped on the same processor then we could propagate the iteration time through the cycle and repeat the process until it converges, but it also could oscillate. This information is useful for interactive applications because if the iteration time of the predecessors oscillate then it affects the whole component in G_{dep}. The developer should also consider that he needs to optimize the iteration time of the predecessor modules to increase performances of its applications. He should consequently avoid these configurations in its mappings. He has also the possibility to increase the priority of predecessor modules and to bind a module to a processor via scheduler settings.

If we have no cycle with several predecessors in G_{dep} but predecessors with input edges then we have a simple dependency between components and we should determine the concurrent computation time of the predecessors without input edges first.

For a predecessor pm without input edges we have :

$$T_{it}(mp) = T_{comp}(pm) \tag{4}$$

We have define the order between predecessors to evaluate their iteration times. We now study how to deal with concurrency.

3.2 Computing modules iteration times

We are able to compute iteration times of predecessors without dependencies. Then we should take care of dependencies and interdependencies between components to compute iteration times of the other predecessors. Once we have iteration time for each predecessor we compute the concurrent iteration time for the other modules.

We now show how to determine iteration time for concurrent modules following this evaluation order.

Concurrency : The behaviour of concurrent modules is determined by the scheduler of the operating system. Our approach is based on the Linux scheduler policy [4] which gives priority to a module over others according to the time each concurrent module waits for I/O operations. In this case the more a module waits, the higher priority it gets.

We define $T_{I/O}(m)$, the time a FlowVR module m waits, as follow :

$$T_{I/O}(m) = T_{it}(m) - T_{comp}(m) \times LOAD(m) \tag{5}$$

Sorting modules : Consequently to determine the priority order of the concurrent modules we order them from the one which waits the most to the one which waits the less according to their $T_{I/O}$.

A predecessor module pm is not synchronized with any other modules so we modify the equation 5 : $T_{I/O}(pm) = T_{comp}(pm) \times (1 - LOAD(pm))$.

If pm is mapped with modules from the same component then they have the same iteration time which is not yet determined. We are nonetheless able to compare

them because if we consider two modules m_1 and m_2 we have $T_{it}(m_1) = T_{it}(m_2) = T_{it}(pm)$, then :

$$T_{I/O}(m_1) - T_{I/O}(m_2) = T_{comp}(m_2) \times LOAD(m_2) - T_{comp}(m_1) \times LOAD(m_1)) \quad (6)$$

If pm is mapped with modules from other components then we have their iteration times according to the evaluation order we have defined. Consequently we are able to order them.

Determining CPULOAD : Once we have ordered concurrent modules on a processor we are able to determine the processor load the scheduler will give to each one. For two modules m_1 and m_2 if m_1 has a higher priority than m_2 we have $m_1 > m_2$.

On a SMP computer we set each module on the processor with the lowest load. The CPU load is allocated to a module m according to the following equation :

$$CPULOAD(m) = (1 - \sum_{m_i > m}^{cpu(m_i)=cpu(m)} CPULOAD(m_i)) \times LOAD(m) \quad (7)$$

On a SMP node, if several modules have the same $T_{I/O}$ then we are not able to order and mapped them if processors have different loads. Our tests show that processes may be moved from a processor to another. In this case the developer has the choice to take the worst case for each module or to change their priorities.

Determining concurrent computation time : With the CPU load information we could compute the concurrent computation time :

$$T_{cc}(m) = T_{comp}(m) \times \frac{LOAD(m)}{CPULOAD(m)} \quad (8)$$

For a module m it is interesting to compare $T_{cc}(m)$ and $T_{comp}(m)$ to measure the consequence of concurrency on it. If values are close then concurrency has few performance penalties, else the developer could change its mapping to reduce T_{cc}.

Determining iteration times : The iteration time of a module m with a predecessor pm is defined by the following equation :

$$T_{it}(m) = max(T_{cc}(m), T_{it}(pm)) \quad (9)$$

For a predecessor module pm we have $T_{it}(pm) = T_{cc}(m)$ because $IM_s(pm) = \emptyset$. If $T_{cc}(m) > T_{it}(pm)$ then m is slower than its input modules. Consequently messages sent from modules in $IM_s(m)$ are accumulated in the shared memory leading to a buffer overflow.

Remarks : At this step we are able to provide lots of informations to the developer. The concurrent computation time gives us an indication on the consequence of concurrency. From the CPU load we know if we use the processors efficiently. The iteration time gives us the module frequency :

$$F(m) = \frac{1}{T_{it}(m)} \quad (10)$$

We now study communications performances.

3.3 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 occurred 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.

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 bandwidth 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 represent communications out of the FlowVR communication scheme, for example communications between several instances of a MPI module. For each iteration 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 :

$$BW_s(n) = \sum_{\substack{node(src(e))=n \\ node(src(e))\neq node(dest(e))}} V(e) \times F(src(e)) \tag{11}$$

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

$$BW_r(n) = \sum_{\substack{node(dest(e))=n \\ node(src(e))\neq node(dest(e))}} V(e) \times F(src(e)) \tag{12}$$

If for a node n $BW_s(n) > B$ then messages are accumulated in the shared memory because the daemon 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 modules. It represents the time needed by a message to be propagated from a module to another through 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 modules m_1 and m_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 and the network latency L for each edge e between two distinct nodes :

$$L(m_1, m_2, P) = \sum_{m \in P} T_{it}(m) + \sum_{src(e) \neq dest(e)} \frac{V(e)}{B} + L \tag{13}$$

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 and by mapping several modules on the same node to decrease communications latencies.

Multiple networks : We assume that each network is based on point-to-point connections in full duplex mode. A network N_i has a given bandwidth B_{N_i} and a latency L_{N_i}.

In the application graph each communication between two FlowVR objects is associated to a given network. For each network we only consider the subgraph associated to it. Then we could apply the formula to compute for each module the required bandwidth to send and to receive data. This way we are able to detect network contentions and buffer overflows if required bandwidths are higher than physical bandwidths.

4 Test application

We illustrate our approach with our fluid-particles application. This application is composed of the following modules :

- the *flow simulation* module based on an MPI version of Stam's simulation [8] computes fluid forces. The fluid is discretized on a 500x500 grid.
- the *particle system* module adds forces from the simulation to a set of particles. We consider a set of 800x800 particles.
- the *viewer* module transforms particles into graphical primitives.
- the *renderer* module displays the scene on the screen.
- the *interaction* module which we could enabled if the simulation runs at an interactive frequency to interact with the fluid.

The cluster is composed of height nodes (*node*1 to *node*8) with two 64bits dual-core Opteron processors. Four nodes (*node*1 to *node*4) are connected to video-projectors to display the application on a 2x2 display wall. Nodes are connected with a gigabit network. We assume full-duplex connections between nodes handled by network adapter (no CPU overload).

The renderer modules are necessarly mapped on nodes connected to video-projectors so we study different mappings for the other modules. Our goal is to provide maximum performance to the simulation and to provide an interactive visualization.

4.1 Study of the application graph

Simulation modules are synchronized with the MPI library to ensure a global coherency of the simulation. Renderer modules are also synchronized with a swaplock synchronizer to provide coherency of the rendering on the display wall. We remove greedy connections from the application graph to obtain G_{sync}. We have two components in G_{sync}: the first one contains the simulation, particle and viewer modules and the second one contains renderer modules. Simulation modules are predecessor modules so we have to avoid concurrency to obtain their minimal iteration times. Renderer modules are also predecessor modules so we need to verify that concurrency with the other modules does not lead to a framerate under 15fps.

4.2 Determining module computation times

First we need the computation time for each module. The computation time for a module could be given by the developer or it could be determined from a simple mapping. In this last case we use the mapping 1 (Figure 1) where modules are mapped on distinct nodes to obtain their respectives T_{comp}. We use the four processors of $node5$ for the simulation. The interaction module could be neglected in this study because it only performs I/O operations : it reads positions from the haptic device and sends it them to the simulation modules. Results are shown on figure 2. We note that, with four processors, $T_{comp}(simulation) = 330ms$.

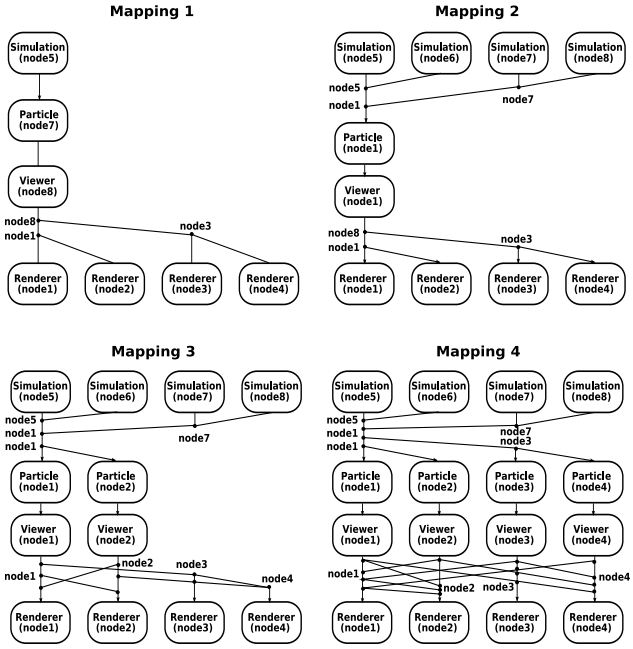

Fig. 1. Different mappings of the application

4.3 Mapping optimizations

We turn now to describe the mapping 2 (Figure 1). We note that, with the previous mapping, the particle system and the viewer module generate respective loads of only 20 and 15% on one cpu. Consequently we could map them on the same node to avoid network communications between them. Renderer modules use only one processor on the four vizualisation nodes so we could map the particle system, the viewer module on *node*1. In the mapping 2 we map the simulation on the four nodes available for a total of sixteen processors. The solver used in the simulation has a linear cost thus we could expect to divide the simulation computation time by four, but we need to had communications between simulation nodes to keep a global coherency. We estimate from the solver and the network characteristics a computation time around 100ms for mapping 2.

The viewer module has the simulation module as predecessor. From equations 2 and 3 we have $T_{it}(viewer) = T_{comp}(simulation) = 100ms$. The viewer produces primitive positions, for the 800x800 set of particles, which are then broadcasted to *node*2 and *node*3. Each position consists of two floats. If $T_{it}(viewer) = 100ms$ then we have $F(viewer) = 10Hz$. In this case $BW_s(node1)$ is equal to $102.4MB/s$ and is greater then the available bandwidth. We conclude that this mapping should generate a buffer overflow on *node*1. We have verified that the simulation has a computation time between 90 and 110ms and that a buffer overflow error is generated after some iterations of the application.

To solve this problem we propose in mapping 3 (Figure 1) to split the particle system on *node*1 and *node*2. This way the viewer module on *node*1 only broadcast 2.56MB to *node*2 and *node*3. But *node*1 also need to send the simulation result (2MB) to *node*2. In this case $BW_s(node1) = 71.2MB$ and is lower than the available bandwidth. Results of this mapping are shown in 2.

In mapping 4 (Figure 1) we map the particle system and the viewer modules on four nodes (*node*1 to *node*4). This way we reduce the computation time of these modules and we also decrease the latencie between the simulation and the renderer modules according to equation 13. Results of this mapping are shown in 2.

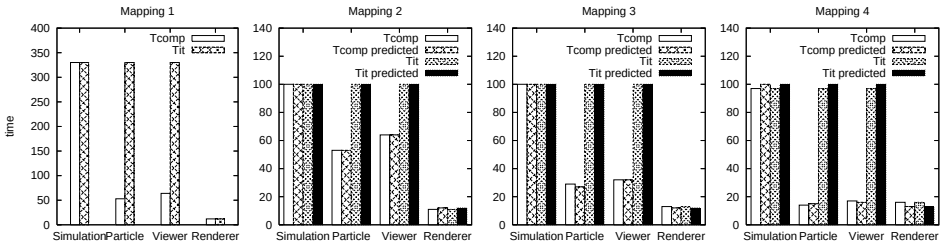

Fig. 2. Performance predictions of the different mappings

In mapping 4 we have on *node*1 to *node*4 only three modules mapped on four processors. We note that $T_{it}(particle) + T_{it}(viewer) < T_{it}(simulation)$. This means that a message from the simulation is processed by the particle and the viewer modules before the production of a new simulation message. In this case the particle and the viewer modules are never concurrent. We would like to map them on the same processor and to add two simulation modules by node on *node*1 to *node*4 for a total of 24 processors dedicated to the simulation. But we know that the scheduler gives the priority to modules which wait the most, here the particle and the viewer modules, and tends to map them on different processors. Because the irregular activity of these modules, the scheduler often remaps CPU-bound modules, here the

simulation and the renderer modules, and we could expect a variable concurrent computation time of these last modules. In the worst case a simulation module could be mapped with another CPU-bound module with the consequence of multiplying by two the computation time of the whole application because simulation modules are synchronized. Consequently in such cases the developer should tune the scheduler to bind modules to processors and to increase the priority of the critical modules to reach optimal performances. In our application with 24 processors dedicated to the simulation we predict $T_{it}(simulation) = 70ms$. With this information we compute the bandwidth needed by $node1$ to send messages. We obtain $BW_s(node1) = 98.4MB/s$ which is greater than the real bandwidth available. Tests show that this mapping lead effectively to a buffer overflow on $node1$.

We also could keep the mapping 4 (Figure 1) and use the available processors to increase the amount of particles in the application. In this case we are limited by the bandwidth of $node1$. We could have at most a number of particles with generates $BW_s(node1) = 80MB/s$. In this case we determine that we could consider a global field of 1000×1000 particles. We also could propose to map two particles and two viewer modules by node to decrease the latency between the simulation and the renderer.

5 Conclusion

We have shown in this paper that we are able to predict performances for distributed applications. We also guide the developer in its mapping optimizations by giving him means to detect networks contentions and to measure consequences of concurrency between modules.

This approach brings to the FlowVR model a mean to abstract the performance prediction from the code. Nevertheless our approach is not limited to FlowVR applications and is sufficiently general to consider applications developed with other distributed middleware. The developer only needs sufficient informations on modules he wants to integrate in his applications. 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.

We have shown that the developer could optimize a lots of parameters but that some local optimizations does not always lead to more efficient mappings. The next step in our approach is to provide automated tools based on our model to assist the developer in his mapping creation and optimization. We also plan to provide automatic optimizations of mapping from constraints defined by the developer.

References

[1] S. Jubertie and E. Melin : Performance Prediction for Distributed Virtual Reality Applications Mappings on PC Clusters. Technical Report RR-2007-03, LIFO, Orlans, France, December 2006

[2] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, S. Robert : FlowVR: a Middleware for Large Scale Virtual Reality Applications. Proceedings of Europar 2004

[3] J. Allard, C. Mnier, E. Boyer, B. Raffin : Running Large VR Applications on a PC Cluster: the FlowVR Experience. Proceedings of EGVE/IPT 05, Denmark, October 2005

[4] J. Aas : Understanding the Linux 2.6.8.1 CPU Scheduler

[5] G. Cavalheiro, F. Galilee and J.-L. Roch : Athapascan-1: Parallel Programming with Asynchronous Tasks. Proceedings of the Yale Multithreaded Programming Workshop, Yale, June 1998

[6] D.B. Skillicorn : Predictable Parallel Performance: The BSP Model

[7] D. Culler and R. Karp and D. Patterson and A. Sahay and K. E. Schauser and E. Santos and R. Subramonian and T. von Eicker : LogP: Towards a Realistic Model of Parallel Computation
[8] J. Stam : Real-time Fluid dynamics for games. In Proceedings of the Game Developer Conference, March 2003