Multiple Networks for Heterogeneous Distributed Applications PDPTA'07

***Index Terms*—multiple networks, distributed applications, performance prediction, FlowVR**

Sylvain Jubertie, Emmanuel Melin
Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
Université d'Orléans
BP 6759
F-45067 ORLEANS CEDEX 2
Phone : (+33)2 38 49 48 57
Fax : (+33)2 38 41 71 37
Email: {sylvain.jubertie | emmanuel.melin}@univ-orleans.fr
http://www.univ-orleans.fr/lifo

***Abstract*—We have experienced in our distributed applications that the network is the main limiting factor for performances on clusters. Indeed clusters are cheap and it is easier to add more nodes to extend the computing capacity than to switch to costly high performance networks. Consequently the developer should especially take care of communications and synchronizations in its application design. The FlowVR middleware offers a way to build distributed applications independently of a particular communication or synchronization scheme. This eases the design of distributed applications independently of their coupling and mapping on clusters. Moreover we propose a performance prediction model for FlowVR applications which is adapted to heterogeneous SMP clusters with multiple networks. In this paper we present an analysis of communication schemes based on our performance prediction model. We give some advices to the developer to optimize communications in its mappings. We also show how to use multiple networks on heterogeneous clusters to balance network load and decrease communication times. Since the FlowVR model is very close to underlying models of lots of distributed codes, our approach can be useful for all developers of such applications.**

I. Introduction

Heterogeneous distributed applications are difficult to map efficiently on clusters. Each module of the application could have very different requirements in term of processor load, memory, or storage. Some modules could be mapped on the same nodes and the developer should consider the scheduler policy to determine the impact of concurrency. The developer should also introduce asynchronism between modules to exploit the intrinsic asynchronism of clusters and at the same time ensure that the synchronization scheme is still coherent. Some modules may be distributed on several nodes like for example MPI-based applications. Consequently to take advantage of the cluster performance the developer should design an efficient mapping. When considering large applications this task becomes more complex. It is even more difficult if we consider clusters composed of heterogeneous nodes.

The FlowVR framework [1][2] allows to design such distributed applications. It was primarly design with distributed Virtual Reality applications in mind but its model is general enough to consider non-interactive distributed applications. The developer could create each of its modules independently and then define communications and synchronizations between them. Each module could then be mapped on the cluster nodes. To provide an efficient mapping we propose in [3] a performance prediction method for FlowVR applications. With this tool we could provide performance informations for each module of a given mapping on a given cluster.

In [3] we apply this method on a distributed application to determine an efficient mapping. We optimize computation performance of each module to take advantage of the computation capacity of SMP nodes of the clusters. Whereas we were able to design efficient mappings for computation performances, we were limited in several cases by network bottlenecks. But we have performance margins for network communications. It is possible to optimize communication schemes or to use multiple networks to reach even more efficient mappings.

We propose in this paper a more precise analysis of the impact of communication schemes on performances. We first present a brief description of the FlowVR framework and of our performance prediction method. Then we will study influence of several communication schemes on a distributed application. We also present how to use multiple networks to increase communication performances and consider larger problems.

II. The FlowVR framework

A. FlowVR

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. More details can be found in [1]. We turn now to present its main features.

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.

1) 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 result messages on its output ports. Messages are also associated with lightweight data called *stamps* that identify the message and allow routing operations. 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 the MPI library but they can be apprehended as one single logical module. Therefore parallel modules do not break the FlowVR model.

2) 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 which could induce buffer overflows. To prevent this behavior, interactive 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 a *filter*. Filters are placed between modules onto connection and have an entire access to incoming messages. They have the freedom to select, combine, create or discard messages.

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.

B. Performance prediction

Our complete performance prediction method for FlowVR applications is described in [3]. For the sake of clarity we only present in this paper the main principles of our method.

To determine performances we only require few informations. For each module we need to know :

- its computation time : the time needed to perform its computation when there is no concurrent modules on the processor.
- its processor load : the percentage of the computation time used for computation and not I/O operations.
- the amount of data sent on each output port.

If the cluster is composed of heterogeneous processors then we need the previous informations for each kind of processor. Because FlowVR modules are independant it is possible to test each one without having to run the whole application.

The global application performance depends on the synchronization scheme and concurrency between modules. If a module is synchronized with its predecessor then it should wait until it receives a new message. We define the iteration time T_{it} of a module as the time between two consecutive calls to the *wait* function. If two modules are mapped on the same node then the operating system scheduler may interleave their executions and change their performances. We define the concurrent computation time T_{cc} as the time needed by a module to perform its computation when other modules are mapped on the same processor. Our goal is to provide for each FlowVR module its iteration time T_{it} and its concurrent computation time T_{cc}. With these informations we will be able to predict the amount of data sent and received on each network interface and to determine if network contentions may occur. We could also predict latency between FlowVR objects which is a usefull information for interactive applications like for example Virtual Reality ones.

1) Synchronizations: Performances are predicted from the application graph G_{appl} : a set of FlowVR objects mapped on cluster nodes and connected together with directed FlowVR connections. G_{appl} contains synchronizations between modules and module locations on the cluster. This allow to determine if some modules are synchronized and wait for messages, and if several modules are mapped on the same nodes. With these constraints we are able to predict the behavior of each module in the application.

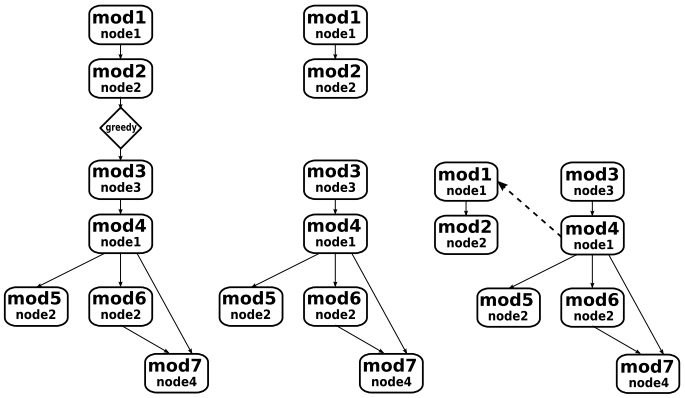

Fig. 1. Example of an application graph G_{appl} and its decomposition in G_{sync} and G_{dep}

We first determine if G_{appl} is correct. Indeed some graph configurations are known to generate buffer overflows or unexpected performances. We have shown in the previous section that greedy filters are used to prevent buffer overflows. Moreover modules connected through greedy filters are running asynchronously. This removes performance constraints between theses modules. Consequently we could remove greedy connections from G_{appl} in our study and only keep FIFO connections. The resulting graph G_{sync} may not stay connected. In this case we have several components. Figure 1 shows an example of an application graph and its corresponding G_{sync} which has two components. We turn now to show that modules in a component could have only one common predecessor module or predecessor modules arranged in a cycle. In this last case we have a predecessor cycle. If this condition is not satisfied then there is at least a module in the component with more than one independent predecessor. In this case these predecessors may have different computation times and consequently send messages to the module at different rates. Because the *wait* function imposes to have one message on all input ports, the module waits for the message of the slowest predecessor. Consequently the other messages are accumulated into the buffer until it generates a buffer overflow. Modules from the same component should have the same iteration time because they are synchronized by the component predecessor. If this constraint is not respected then it means that for a module m we have $T_{cc}(m) > T_{it}(m)$ i.e. m is slower than it predecessor and consequently messages from it are accumulated in the buffer which leads to a buffer overflow. This way we are able to warn the developer if his mapping may generate errors and to point out incorrect configurations.

2) Concurrency: To predict performances we also take into account the concurrency between modules mapped on the same cluster node because the scheduler may interleave their executions and change the time they need for computation. The way the processor load is distributed between modules depends of the scheduler policy. In our study we choose to model the Linux scheduler policy [4] which gives the priority to modules which wait the most.

To represent concurrency relations between modules we add edges into G_{sync} from concurrent modules to predecessor modules mapped on the same processor. We distinguish these edges related to concurrency from synchronization edges in G_{appl}. The resulting graph is called the dependency graph G_{dep}. Figure 1 shows an example of an application graph and its corresponding dependency graph G_{dep}. Then we could consider independently each component in G_{dep}. We first check for cycles to determine interdependencies between predecessors. In figure 1, if $mod2$ and $mod3$ were mapped on the same node then we would have a cycle. If several predecessors are in the same cycle then performance of each predecessor depends of the performance of the other ones and we are not able to directly determine T_{cc} of modules in the cycle. In this case we propose to consider the T_{comp} of one predecessor module as its T_{cc} and to propagate it to determine the T_{cc} of the other modules following the cycle order. We repeat this process along the cycle until we reach convergence of the concurrent computation time for each module. But in some cases concurrent computation times may oscillate because this process changes the iteration time of each module and could modify the priority between them. But we are able to determine for each module its minimal and maximal T_{cc} which is obtained respectively when the scheduler gives the highest and the lowest priority. However we should avoid this behavior for interactive applications or if we want a module to have the lowest possible T_{cc}. In this case the developer should tune the scheduler to change the priority of modules or bind modules on processors.

Once we have determined T_{cc} for each predecessor module, we could then compute T_{cc} for the other modules. We should also verify that for all modules we have $T_{cc} \leq T_{it}$. If this is not the case then it means that a module receives messages at a higher rate than it computes. Messages are then accumulated in the buffer leading to a buffer overflow.

At this step for each module we have its T_{cc}, its T_{it} and we are able to point out a module which could generate buffer overflows. With these informations we are able to compare mappings and to choose the one which takes the best advantage of the cluster processors. However we cannot guarantee that the best mapping we obtain at this level is the best application mapping. Indeed if we optimize modules iteration times then we increase the amount of communications between modules. This could lead to network contentions. Consequently we should study the communication scheme of the mapping to verify if it is compatible with the module

performances.

3) Communications: In this section we determine the amount of data sent and received on each network interface from G_{appl}.

We assume that communications between objects mapped on the same node are free because messages are stored in a shared memory and local objects only exchange pointers to messages. In our study we consider networks with point-to-point connections in full-duplex mode. Each network Ni has a given bandwidth BWi and latency Li.

We also assume that synchronizer communications are negligible compared to other communications. Indeed even if synchronizations occure at the frequency of the fastest module involved in the synchronization, they require only few stamp information compared to the size of the message sent by this module.

Consequently we define a new graph G_{comm} which is obtained from G_{appl} by removing synchronizers. We also add additional edges in G_{comm} 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.

For the sake of clarity in the following sections we define the frequency of a module m as follow :

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

Indeed the volume of data sent by a module depends only of its frequency.

We begin our study by a traversal of G_{comm} to determine for each filter its frequency and the data volume it received and sent. The frequency of a filter depends of its nature. For a greedy filter f_{greedy} it is equal to the frequency of the destination module m_{dest} : $F(f_{greedy}) = F(m_{dest})$. For a broadcast filter it is equal to the frequency of the source object o_{src} : $F(f_{broadcast}) = F(o_{src})$. The data volume sent by a filter depends also of its nature. For example a broadcast filter sends the message it receives on its output ports without modifying its size whereas a merge filter sends only one message which contains messages it receives. For each edge in G_{comm} we are able to compute the data volume sent through it in a second by multiplying the frequency of its source object by the amount of data sent through it.

Then we could determine for each node the bandwidth BW_s required to send data on a given network. This is done by adding data volumes of each edge e connected from this node ($src(e)$) to another node ($dest(e)$) through this network :

$$BW_s(n) = \sum_{node(src(e))=n}^{node(src(e))\neq node(dest(e))} V(e) \times F(src(e)) \quad (2)$$

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

$$BW_r(n) = \sum_{node(dest(e))=n}^{node(src(e))\neq node(dest(e))} V(e) \times F(src(e)) \quad (3)$$

If the required bandwidth is greater than the available network bandwidth then messages are accumulated in the sending buffer until it is full and we have a buffer overflow error. This way we are able to point out network bottlenecks in application mappings.

4) Latency between FlowVR objects: The latency represents the time needed by a message to be propagated from a FlowVR object to another through the application graph. In interactive applications, like Virtual Reality ones, the latency is critical between interaction and visualization modules, the user should see the result of its interaction within the shortest possible delay to keep an interactive feeling.

We determine the latency between two objects from a path the message should follow between them. A path contains a set of FlowVR objects and edges between them. We assume that filters like *broadcast* or *merge* ones do not introduce latency. We compute the latency by adding iteration times of each module in the path and the network latency for each edges. This latency is reached in the best case when a module has no concurrent and consequently its messages are not buffered. If several modules are mapped on the same node then their messages are buffered and sent one by one. This process add extra latency and in the worst case messages from other modules mapped on the same node and sent on the same network are stored in the buffer. Consequently we could add the time each other message needs to be sent through the network to the best case latency. This way we obtain the latency in the worst case situation.

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 reduce communication latencies.

III. Case study

We illustrate our approach with our fluid-particles application. In [3] we have shown that we were able to optimize frequencies of the different modules. We now study optimizations of the communication schemes.

A. The cluster

Our cluster consists of two different sets of nodes. The first set (nodes 1 to 8 in figure 2) is built with dual Xeon processors interconnected with a gigabit Ethernet and a Myrinet network. The second one (nodes 11 to 18) with 8 dual-core dual Opteron processors networked with two gigabit Ethernet interfaces. This cluster is giving a total of 48 processors.

To visualize our applications we use a display wall of 4 projectors (2x2), each one connected to a different node.

B. The application

The fluid-particles application is composed of the following modules :

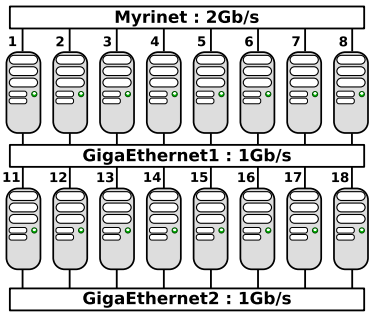

Fig. 2. The cluster and its networks

- the *flow simulation* module based on an MPI version of the Stam's simulation [5] 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 400x400 particles.
- the *viewer* module transforms particles into graphical primitives.
- the *renderer* module displays the scene on the screen. They must be mapped on nodes with projectors connected to it.
- the *joypad* module which we could enabled if the simulation runs at an interactive frequency to interact with the fluid.

The data flow of the application is described in figure 3.

The flow simulation may be distributed over several nodes to increase its frequency. In this case each node only consider a local subdomain of the simulation. After each iteration of the simulation module subdomains are sent to merge filters in order to create the global domain. Then we use a binary broadcast scheme to send the global domain to each instance of the particle system module. Particles are then transformed by the viewer module and broadcasted to each renderer module.

An example of an automatically generated graph with only two instances of each module is shown in figure 4. Even in this simple case the graph contains a total of 40 FlowVR objects. It illustrates the complexity for the developer to map efficiently each object of the application without a performance model. For the sake of clarity we only consider a simplified representation of the graph in the following sections.

C. Performance predictions and results

We study three different cluster configurations to show how to apply our approach in each case. First the application is mapped on an homogeneous cluster with only one network. Then we add a second network and we modify the communication scheme to show how to take advantage of the two networks. Finally we consider an heterogeneous cluster with three different networks. In each case we study several mappings to show how to optimize both module frequencies and communication schemes.

1) Homogeneous cluster connected with one network: We first consider a cluster with eight dual processor nodes (nodes 1 to 8 in our cluster on figure 2) connected with a gigabit Ethernet network.

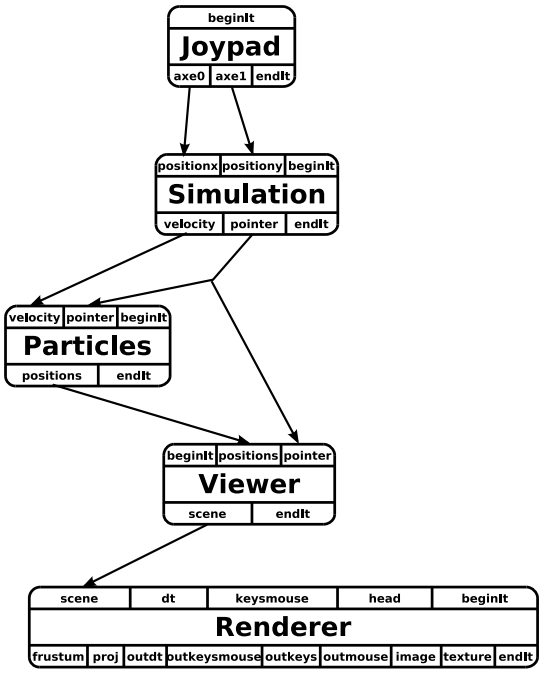

Fig. 3. The fluid-particle application simplified dataflow graph

Modules are mapped as described in table I. The simulation is mapped on four nodes with two instances by node to take advantage of the dual processors.

Module	**Nodes**	**Prediction**(ms)		**Results**(ms)	
		T_{it}	T_{cc}	T_{it}	T_{cc}
Simulation	{5, 6, 7, 8}	80	80	81	81
Particles	{1, 2, 3, 4}	80	20	81	20
Viewer	{1, 2, 3, 4}	80	15	81	15
Renderer	{1, 2, 3, 4}	40	40	30-40	30-40

TABLE I

The particles, viewer and renderer modules are mapped on the same nodes and we should take care of the concurrency between them to determine their concurrent computation time. The particles and the viewer modules have a highest priority over the renderer module because they are waiting more. Thus the scheduler should map the particles and the viewer modules on different nodes and the renderer module do not run at full speed. We note that $T_{cc}(particles) + T_{cc}(viewer) < T_{cc}(simulation)$. This means that the particles and the viewer modules are not executed at the same time because when a message is sent by the simulation it is processed by the particles module then by the viewer module and the simulation has not yet sent a new message. The consequence of this remark is that we could tell the scheduler to bind the particles and the viewer modules to the same processor. In this last case the renderer module is bind to the other processor to take the best advantage of it. This is what we have done for this mapping.

We first verify for each module that the concurrent computation time is not greater than the iteration time. Then we compute the amount of data sent and received by each node. The simulation grid of 500×500 is distributed on four nodes, each node with a 250×250 (62500 cells) local grid. Each cell in the grid contains a vector of two floats (8 bytes) representing

a force. On node 5 $1.5MB$ are gathered per iteration from the other simulation nodes to build the full grid of $2MB$ which is then broadcasted to nodes 1 and 3. The iteration time of the simulation is equal to $80ms$, consequently we have $BW_r(node5) = 18.75MB/s$. and $BW_s(node5) = 50MB/s$. We apply the same reasoning to node 1 which receives simulation data from node 5 and viewer data from nodes 2, 3 and 4. It also sends viewer data to renderer modules on nodes 2, 3 and 4. We obtain $BW_r(node1) = 37MB/s$ and $BW_s(node1) = 12MB/s$. The required bandwidth doesn't exceed the physical bandwidth of the gigabit Ethernet network. Consequently we could predict that this mapping will work.

Results of the mapping are shown in table I. The application runs as expected. We note that the concurrent computation time of the renderer varies from 25ms to 35ms because it depends on the scene view point and the number of visible particles.

We now try the same mapping on the eight other nodes with the same network (table II). For each simulation node we have four instances of the simulation to take advantage of the four processors.

We expect the simulation to be twice as fast as in the previous mapping.

Module	**Nodes**	**Prediction**(ms)		**Results**(ms)	
		T_{it}	T_{cc}	T_{it}	T_{cc}
Simulation	{15, 16, 17, 18}	40	40	42	42
Particles	{11, 12, 13, 14}	40	20	42	18
Viewer	{11, 12, 13, 14}	40	15	42	14
Renderer	{11, 12, 13, 14}	10	10	7-12	7-12

TABLE II

We could predict a buffer overflow because the broadcast filter on node 15 will send messages of 2MB per iteration to node 11 and 13. Consequently we have $BW_s(node15) = 100MB/s$ which is greater than the available bandwidth of the gigabit Ethernet network.

Indeed the application failed few iterations after we launched it. Results of theses iterations are shown in table II. We note that the renderer module is four times faster on theses nodes. This is due to the different graphic cards and especially to the larger bus between the memory and the graphic card.

We now propose to add a second network to solve this problem.

2) Homogeneous cluster connected with two networks: With a second network we could decrease the communication latency by splitting each message over the two networks. This requires some work but it is very facilitated by the FlowVR library. We need to add scatter filters after each output port and merge filters before each input port. Another solution is to bind communications between modules to a given network.

To remove the bottleneck of the previous mapping we choose this last solution. We use the two available networks to perform the broadcast from node 15 to node 11 and 13. Each message is sent through a different network. With this configuration only 50MB/s are emitted on each network.

Node 11 receives 50MB/s from the simulation and also 24MB/s from the viewer modules on nodes 12, 13 and 14. Consequently we do not exceed the network bandwidth.

Results are the same as the previous mapping (table II) with a single network but this time the application runs without buffer overflows.

3) Heterogeneous cluster and networks: We now consider our application on the full cluster with 16 nodes (figure 2). Our goal is to reduce the computation time of the simulation to have the best performances, to maximize the number of particles to catch fine details like vortices in the flow, and to have an interactive visualization with at least 20 frames per second.

Consequently we could consider using the eight nodes with four processors for a total of 32 processors for the simulation. If we use all these processors we could predict that the simulation computation time will decrease to 20ms because the simulation complexity is linear. However this means that the simulation will produce 200MB/s and our networks are not able to transmit this amount of data. Moreover in the previous mapping we have a concurrent computation time equal to 40ms which corresponds to a frequency of 25Hz. This frequency is sufficient if we consider that the visualization should run at least at 20Hz to be interactive. Thus we only need 16 processors for the simulation i.e. four dual-core dual processor nodes or eight dual processor nodes.

The next step is to increase the number of particles. We are limited in this case by the network bandwidth but also by the graphic card capacities. We have also a constraint on the number of processor but it is not the most restrictive one. In the first mapping (table II) the renderer modules on nodes 1 to 4 run at 25Hz ($T_{cc}(renderer) = 40ms$). If we increase the particle system then we decrease the renderer module frequency which will not be interactive anymore. Consequently we could not take advantage of the Myrinet network to consider more particles. We could use more powerfull graphic cards on nodes 11 to 14 to remove this bottleneck. If we map particles and viewer modules on nodes 11 to 14 we are in the same case described in the previous section : an homogeneous cluster with two networks. Thus we are limited by node 11 which receives a total of 74MB/s and we are too close to the network bandwidth to consider more particles.

The last possibility is to map particles and viewer modules on nodes 1 to 8. We run two instances of each one on each node to take advantage of the two processors and to reduce the latency of the application. In this case node 1 receives the global grid from the simulation and then broadcast it through the Myrinet network. Then each viewer module send its data to each renderer module through the gigabit network which could send and receive data at around 80MB/s. If the simulation runs at 25Hz we could transmit a maximum of 3.2MB of data per iteration which corresponds to a set of around 600x600 particles.

Regarding all these parameters we propose the mapping described in table III. Results are really close to our prediction. We also could show that only four nodes are sufficient to perform the computation on particles. Indeed with four nodes and two instance of the particles and viewer modules on each one we could predict that $T_{cc}(particles) = 10ms$ and $T_{cc}(viewer) = 9ms$ and we still have $T_{cc}(particles) + T_{cc}(viewer) < T_{cc}(simulation)$. This information could be useful on large clusters running several applications to

optimize the number of nodes for each one.

Module	**Nodes**	**Prediction**(ms)		**Results**(ms)	
		T_{it}	T_{cc}	T_{it}	T_{cc}
Simulation	{15, 16, 17, 18},	40	40	44	44
Particles	{1, 2, 3,..., 8}	40	5	44	5
Viewer	{1, 2, 3,..., 8}	40	4	44	4
Renderer	{11, 12, 13, 14}	20	20	25-35	25-35

TABLE III

IV. Conclusion

The FlowVR library highly facilitates the coupling of distributed applications. The developer could map each part of the application and synchronize them without adding modifications to the code. Facilities provided by FlowVR are necessary to abstract a distributed application from the underlying cluster. But this is not sufficient to provide good performance to a distributed application because FlowVR does not provide its own performance model. Consequently without performance model we are not able to choose a mapping with expected performances.

Our performance prediction model gives to the developer a mean to determine its mapping performances. This way it is possible to compare several mappings and to find the most efficient of them. Once again it is a useful tool but it doesn't give informations on how to obtain the best mapping for a distributed application on a given cluster.

In this paper we have shown that it is possible to use our performance prediction model to optimize communication schemes in our applications. Indeed it is often the main performance limitation for applications distributed on clusters and adding more nodes in a cluster does not solve this problem. However this constraint limits the number of mappings to study and gives to the developer a mean to obtain good mappings without having to consider all the possible parameters. For example it could limit the number of instances per module or the choice of the network to use. Consequently we are able to determine if only a subdomain of the cluster is sufficient. Then the remaining available nodes could be used for other applications. Our approach could also determine if an application could run on a given cluster with the expected performances. If this is not the case we are able to point out the limiting factor and to determine the least cluster configuration able to run the application. Thus we avoid expensive and non efficient investments.

The next step in our approach is to provide automated tools which integrates our model to assist the developer in his mapping creation and optimization. We also plan to provide a solver for automatic optimization of mappings based on cluster constraints, like network bandwidth, and constraints defined by the developer, for example a minimum framerate for a visualization module.

References

[1] 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.

[2] J. Allard, C. Ménier, 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.

[3] S. Jubertie and E. Melin, "Mapping and performance prediction for distributed applications on heterogeneous clusters," LIFO, Tech. Rep., 2007.

[4] J. Aas, "Understanding the linux 2.6.8.1 cpu scheduler," 2005.

[5] J. Stam, "Real-time fluid dynamics for games," in *Proceedings of the Game Developer Conference*, March 2003. [Online]. Available: citeseer.comp.nus.edu.sg/stam03realtime.html

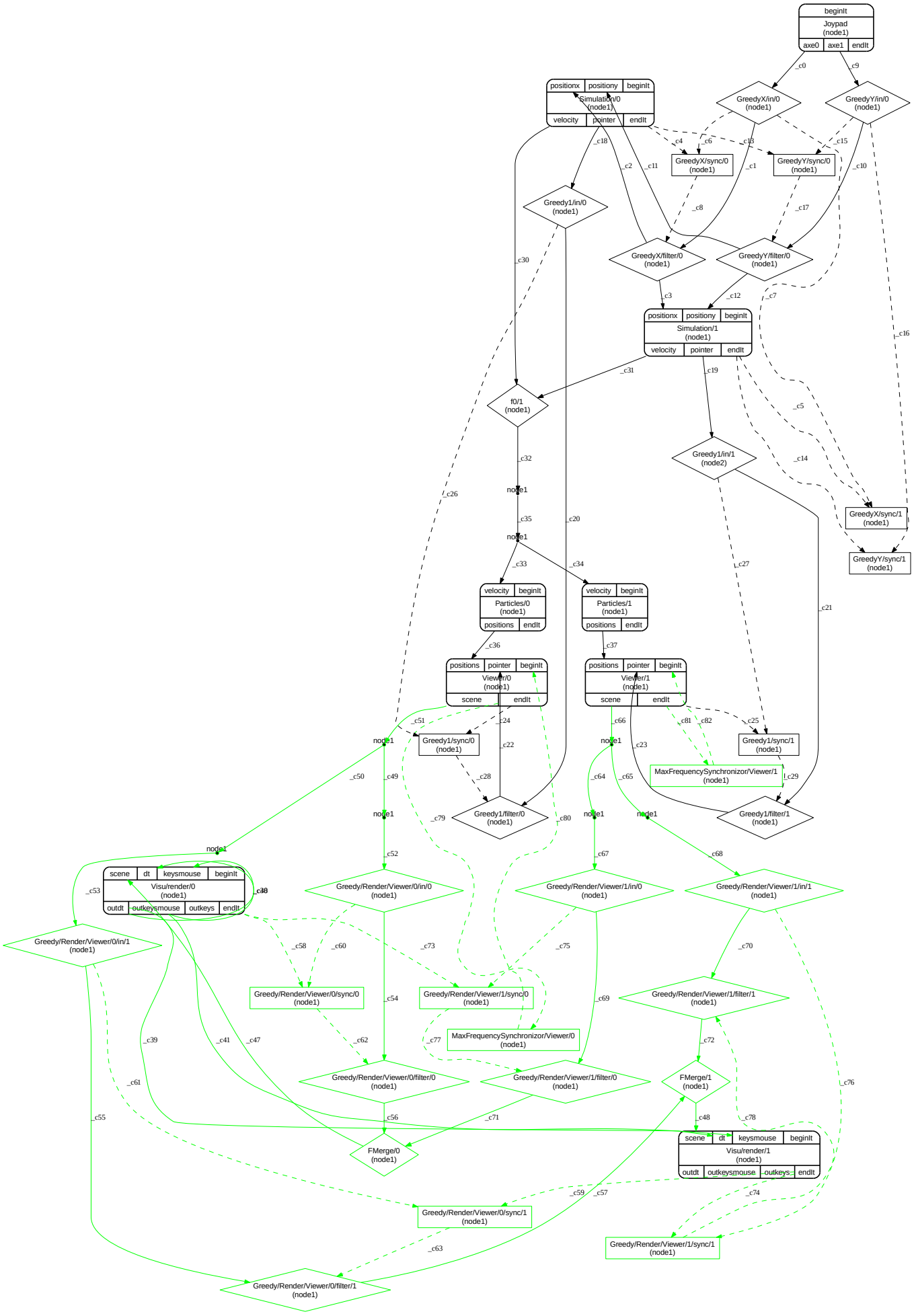

Fig. 4. An automatically generated application graph