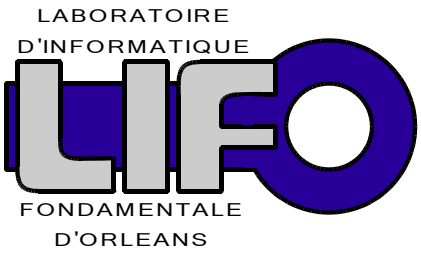

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

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

Task-to-processor allocation for distributed heterogeneous applications on SMP clusters

Sylvain Jubertie, Emmanuel Melin
Université d'Orléans, LIFO

Rapport N^o **2007-08**
05/04/2007

Task-to-processor allocation for distributed heterogeneous applications on SMP clusters

Sylvain Jubertie and Emmanuel Melin
{sylvain.jubertie | emmanuel.melin}@univ-orleans.fr

Laboratoire d'Informatique Fondamentale d'Orléans
Université d'Orléans

Abstract. Today, distributed architectures are based on multi core SMP nodes. Several middleware, like the FlowVR framework, exist to design interactive and heterogeneous distributed applications for these architectures. FlowVR defines an application as a set of codes mapped on cluster nodes and linked by a communication and synchronization network. But if we control the way modules are synchronized and mapped on nodes, we do not control the scheduling of concurrent modules on processors which is done by the Operating System scheduler. Since modules may be synchronized across different nodes, each local scheduling can affect the whole application performance. Moreover the OS scheduling is dynamic thus it is very difficult to predict and guarantee performance of such applications and especially of interactive ones because performance variations may totally break the immersion feeling. In this paper we propose to use a performance model to determine the processor load required by each distributed task. With this information we can define a task-to-processor allocation and tune the scheduler to respect it. Thus we can abstract this allocation from a particular OS, avoid effects of a dynamic scheduling, guarantee performance, optimize the processor use and the number of nodes used for our application.

1 Introduction

Today, distributed architectures have very different levels of parallelism. A cluster may be composed of heterogeneous SMP nodes connected with multiple heterogeneous networks. Common parallel programming libraries, like MPI or PVM, are not well suited to build heterogeneous distributed applications. Indeed the resulting application code contains the communication and synchronization schemes. Thus the developer has to modify the application code to change these schemes or to take advantage of different architectures. Consequently this approach is very dependant of the underlying architecture.

The FlowVR framework[1] allows to create applications for heterogeneous distributed architectures. It was developed with distributed Virtual Reality applications in mind but its model is general enough to consider non interactive distributed applications. The underlying model behind FlowVR can be compared to component approaches. A FlowVR application is a set of tasks, called modules in the FlowVR terminology. The communication and synchronization schemes are defined independently of the modules by a

[1] http://flowvr.sourceforge.net

FlowVR network. Since modules are independent of a particular communication or synchronization scheme it is possible to adapt an application to different clusters without modifying the application code. FlowVR can be used to build large distributed applications like the Virtual Reality applications presented in [3, 4] which can contain more than 5000 FlowVR objects. However, FlowVR does not provide its own performance model. Consequently the mapping of the application objects is leaved to the developer's skills but it is difficult and tedious to find an efficient mapping, especially for interactive application where performance is crucial. A performance model for FlowVR applications would bring to the user a mean to determine performance for its mapping without having a deep knowledge of the FlowVR model.

We propose in [8] a performance prediction model for FlowVR applications. This model integrates the synchronization and the communication schemes as well as the concurrency between modules to compute performance of the application. To determine the effect of concurrency between modules on performance the choice is made to model the OS scheduling policy. Obviously the OS scheduler does not take into account synchronization between modules and a performance drop for one module may be transmitted to modules on other nodes. Thus a local scheduling can affect the global performance of an application.

In this paper we propose to use our performance model to determine the processor load required by each FlowVR module for a given mapping. Then we define a module-to-processor allocation on each cluster node. We also point out nodes which are overloaded when such an allocation is not possible on them. In this case we can tell the developer to modify his mapping and to move some modules to other available nodes. Finally, the processor loads and the module-to-processor allocation allow the tuning of the scheduler in order to respect this information. We test our approach on a distributed application which contains more than 80 different objects to map on cluster nodes. Since our approach does not depend on a particular OS anymore we can completely abstract the performance prediction model from a specific OS scheduling policy. The FlowVR model is very close to underlying models of lots of distributed codes, for example component based applications, thus our approach can be useful for all designers of distributed applications.

2 Performance prediction for the FlowVR framework

In this section we briefly present the FlowVR framework and the performance prediction model proposed in [8].

2.1 FlowVR

The FlowVR framework is an open source middleware used to build distributed applications. More details on FlowVR can be found in [2]. A FlowVR application is a set of objects which communicate via messages through a data-flow network. Each message is associated with lightweight data called *stamps* which contain information used for routing operations.

Some objects, called modules, are endless iteration which encapsulate tasks. Each module waits until it receives one message on each of its input port. This task is performed by a call to the FlowVR *wait* function. Then messages are retrieved by the *get*

function and are processed by the module. Finally the module produces new messages and put them on its output ports with the *put* method.

The data-flow network describes the communication and synchronization schemes between module ports. Each communication is done with a point to point FIFO connection. Operations on messages like routing, broadcasting, merging or scattering are done with a special object called a *filter*. Synchronization and coupling policy are performed with another kind of object called a *synchronizer*. Both filters and synchronizers are placed on connections between modules. A *synchronizer* only receives *stamps* from filters or modules. Then it takes a decision according to its coupling policy and sends new *stamps* to destination objects. This decision is finally performed by the destination filters or modules. With the use of synchronizers it is possible to implement the *greedy* filter. When used between a source and a destination module, this filter allows them to respectively write and read a message asynchronously. Thus the destination module always uses the last available message while older messages are discarded. A FlowVR application can be viewed as a graph $G(V, E)$, called the *application graph*, where each vertex in V represents a FlowVR objects like a module, a filter or a synchronizer, and each directed edge in E represents a connection between two objects.

2.2 Performance prediction for heterogeneous distributed applications

In this section, we only present some aspects of our prediction model, described in [8], needed to understand our approach.

The goal of our performance model is to determine the effects of synchronization and concurrency between modules on performance. Our approach is based on the study of an application graph G enriched with mapping information (figure 1). A mapping is defined by two functions $node$ and $network$. For each object $o \in V$, $node(o)$ returns the node in $Nodes$, the set of the cluster nodes, where o is mapped. For each connection $c \in E$ between two objects $o_1, o_2 \in V$, $network(c)$ returns the network in $Networks$, the set of the cluster networks, used by c. If $network(c) = \emptyset$ then we have a local connection between objects on the same node. Each network $net \in Networks$ is associated to a bandwidth $BW(net)$ and a latency $L(net)$.

We now present the inputs of our performance prediction model. For each module m we need to know its execution time $T_{exec}(m, n)$ and its load $LD(m, n)$ on each node $n \in Nodes$. The execution time $T_{exec}(m, n)$ corresponds to the time needed to execute the module task i.e. the time between the exit of the *wait* function and its next call, when m has no concurrent modules on node n. The load $LD(m, n)$ represents the amount of $T_{exec}(m, n)$ used for computation and not for I/O operations on node n. We also need to know $vol(c)$, the amount of data sent on each connection $c \in E$. We assume that $T_{exec}(m, n)$ and $LD(m, n)$ for each module m on each node n, and $vol(c)$ for each connection, are given by the developer.

These inputs are then used by our model. Since the OS scheduler may interleave the execution of concurrent modules, the execution time of each concurrent module may increase depending of the scheduling policy. This new execution time is called the concurrent execution time $T_{conc}(m)$. If the module m is synchronized with other modules then, once it has completed its task, it must stay in the *wait* function until it receives messages from them. The iteration time $T_{it}(m)$ corresponds to the time between two

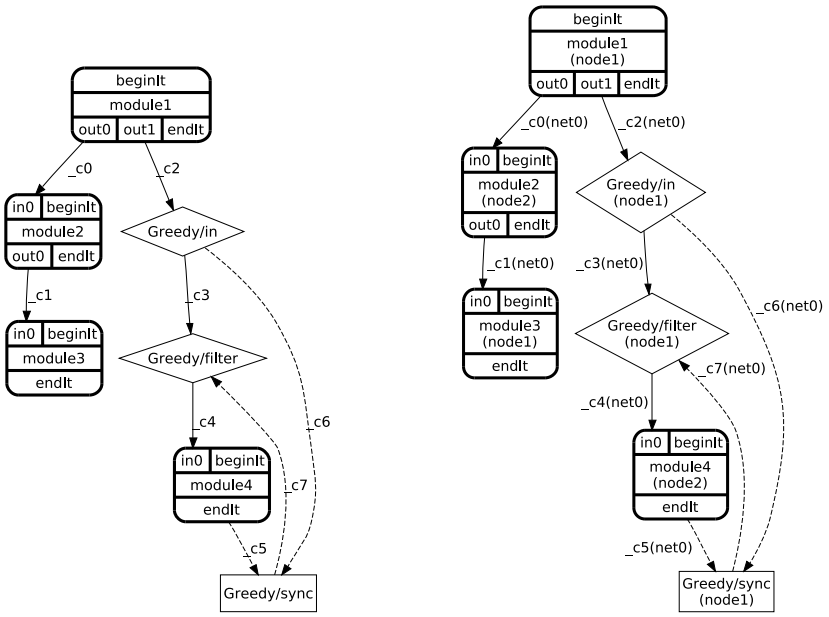

Fig. 1. An application graph G (left) enriched with mapping information (right)

consecutive calls to the *wait* function. The goal of the performance model is to determine, for a given mapping, $T_{conc}(m)$ and $T_{it}(m)$ for each module m. This is done by the study of two graphs, G_{sync} and G_{dep}, derived from G (figure 2). Note that we do not consider filters and synchronizers in our study. Indeed filters and synchronizers are waiting for messages from modules connected to them and have negligible loads compared to modules since filters only perform memory operations like merging or duplicating messages, and synchronizers perform simple operations on stamps. Thus we assume that filters and synchronizers have negligible loads and execution times.

The first graph G_{sync} is obtained from G by removing *greedy* filters. Thus it only contains synchronized modules since modules connected by *greedy* filters run asynchronously. This graph may be splitted into several connected synchronous components. A module m in G_{sync} must wait for messages from modules connected to its input ports. These modules are called input modules of m and are noted $IM_s(m)$. The iteration time of a module m in a component G_{sync} depends on its $T_{conc}(m)$ and on the iteration time of its inputs modules :

$$T_{it}(m) = max_{m_i \in IM_s(m)}(T_{conc}(m), T_{it}(m_i)) \tag{1}$$

The graph G_{sync} is useful to verify for each module m that $T_{conc}(m) \leq T_{it}(m_i), m_i \in IM_s(m)$. If this is not the case then m is slower than its input modules and messages are accumulated in the receiving buffer until it is full. Consequently we predict that a buffer overflow will occur when running the application. Since we want to avoid it, we must have $T_{it}(m) = T_{it}(m_i)$ for each $m_i \in IM_s(m)$. Thus the iteration time of

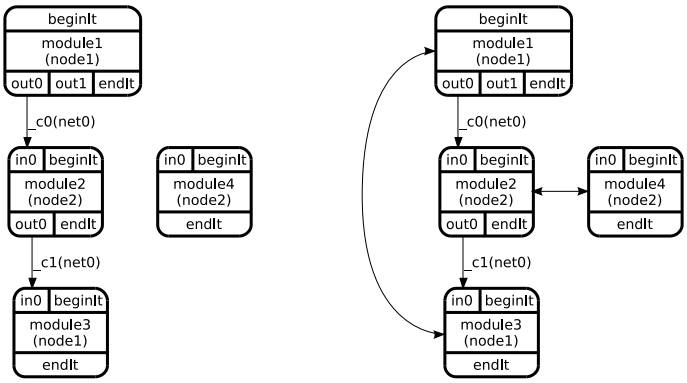

Fig. 2. G_{sync} (left) and G_{dep} (right) derived from G (figure 1)

each module in a component C_{sync} must be the same and we note it $T_{it}(C_{sync})$. We note that a module $m \in C_{sync}$ may have $IM_s(m) = \emptyset$. These modules are called *predecessor modules* of C_{sync} and we note them $preds(C_{sync})$. In this case we have $T_{it}(pm) = T_{conc}(pm), pm \in preds(C_{sync})$. Thus we define $T_{it}(C_{sync})$ as follow :

$$T_{it}(C_{sync}) = T_{conc}(pm), pm \in preds(C_{sync}) \tag{2}$$

This means that the performance of modules in a component C_{sync} only depends on the concurrent execution times of predecessor modules.

The dependency graph G_{dep} (figure 2) models concurrency between modules by adding bidirected edges in G_{sync} between concurrent modules. It is used to determine the effects of both concurrency and synchronization on module performance. If several modules are concurrent i.e. connected with bidirected edges in G_{dep}, then we determine their performance according to a scheduling policy. In [8], we choose to model the Linux scheduling policy [1, 5]. But we have experienced in our applications that it does not always provide the best performance since it does not take into account synchronization between modules. Moreover the OS scheduling is dynamic thus the priority of modules may change during execution. Since modules may be synchronized over several nodes a local scheduling may affect the scheduling on each node. This can, for example, lead to variable performance which breaks the interactive feeling in Virtual Reality applications. To prevent this variation we need to detect interdependencies between the effects of concurrency and synchronization. An interdependency occurs when we have a cycle with both directed and bidirected edges in G_{dep}. Thus we use G_{dep} to detect interdependencies between modules and to guide the developer to avoid them.

This approach, precisely described in [8], allows to determine the effects of synchronization and concurrency on performance and to detect when the OS scheduling policy can lead to poor or variable performance. This approach is not well suited if we want to guarantee performance since it highly depends on the local OS scheduling policy of each node. On the other hand, we now show that is is possible to guarantee performance by choosing a suitable module-to-processor allocation and to harness the

scheduler to it. This approach also provide a mean to abstract our performance model from the OS scheduler.

3 Module-to-processor allocation

In this section we describe the principle of our module-to-processor allocation. Our approach is based on the study of a mapping given by the developer. In our previous approach modules were mapped on nodes by the developer and then on processors by the OS scheduler. We now propose to define a static module-to-processor allocation which ensures performance and avoid the OS scheduler drawbacks. The first step consists in determining the processor load required by each module. This is done by using our prediction performance model.

3.1 Determining module processor loads

The main idea of our approach is to take advantage of the time a module waits, which depends on its own load $LD(m)$ and on the time it stays in the FlowVR *wait* function, for the execution of other processes. For example if a module m is synchronized with an input module then from equation 1 we can add concurrent modules while we have $T_{conc}(m) \leq T_{it}(m_i), m_i \in IM_s(m)$ to avoid a buffer overflow. We assume that applications have a fixed number of processes and are running on dedicated nodes thus a static allocation seems realistic.

Component performance : The developer must choose for each predecessor module pm its $T_{conc}(pm)$ such as $T_{conc}(pm) \geq T_{exec}(pm, node(pm))$. Thus we also obtain $T_{it}(C_{sync})$ of each component $C_{sync} \in G_{sync}$ from equation 2. If he wants to slow down the component then he can choose for each predecessor pm the corresponding $T_{conc}(pm)$ such as $T_{conc}(pm) > T_{exec}(pm, node(pm))$. Then we obtain the load $LD_c(pm)$ required on the processor to ensure the correct $T_{conc}(pm)$ with the following equation :

$$LD_c(pm) = T_{exec}(pm, node(pm)) \times \frac{LD(pm, node(pm))}{T_{conc}(pm)}, pm \in preds(C_{sync}) \tag{3}$$

If the developer wants to run the predecessor module at its full speed then he chooses $T_{conc}(pm) = T_{exec}(pm, node(pm))$ and we obtain $LD_c(pm) = LD(pm, node(pm))$. Note that the developer must choose $T_{conc}(pm)$, consequently from equation 2 we have the $T_{it}(C_{sync})$ each module in a component C_{sync} must respect.

Module performance : We now determine for each other module $m \in C_{sync}$ its $LD_c(m)$. The lowest $LD_c(m)$, noted $LD_{cmin}(m)$, we can allocate to each module m is reached when $T_{conc}(m) = T_{it}(C_{sync})$. It depends on the part of $T_{it}(m)$ used for computation and not for I/O operations. Thus we define LD_{cmin} as follow :

$$LD_{cmin}(m) = \frac{T_{exec}(m, node(m)) \times LD(m, node(m))}{T_{it}(C_{sync})}, m \in C_{sync} \tag{4}$$

If $LD_{cmin}(m) < LD(m)$ the developer can choose for each module a higher processor load $LD_c(m)$ such as $LD_{cmin}(m) \leq LD_c(m) \leq LD(m)$. Thus we can decrease $T_{conc}(m)$ according to the following equation :

$$T_{conc}(m) = \frac{T_{exec}(m, node(m)) \times LD(m, node(m))}{LD_c(m)} \quad (5)$$

This can be useful to reduce latency between modules. If $LD_{cmin}(m) > LD(m, node(m))$ then it means that we can not allocate enough processor load to m and we can predict a buffer overflow. In this case the developer can increase $T_{it}(C_{sync})$ by slowing down predecessors or by mapping the module on a faster processor to decrease its T_{exec}.

An application mapping gives us the mapping of modules on nodes. We now give means to map each module m on a processor according to its $LD_c(m)$.

3.2 Mapping modules on processors

We now determine how to map modules on processors. For each node n we define $Procs(n)$ as the set of processors of n. For each module m on a node n we define $proc(m, n)$ as the processor where m is mapped. We can map several modules on a processor while the sum of their LD_c does not exceed 1. Thus to find a correct mapping we must verify the following inequality :

$$\forall n \in Nodes \sum_{\substack{p \in Procs(n), \\ proc(m,n)=p}} LD_c(m) \leq 1 \quad (6)$$

If this inequality is not verified on some nodes then we have not enough processors to ensure expected performance. Consequently the developer can slow down predecessors of modules mapped on n or he can consider a different mapping.

However, we can improve the mapping of modules on processors by determining if some modules can not run at the same time due to synchronization between them. In this case, these modules can be mapped on the same processor without cumulating their LD_c. This is for example the case of modules in the same synchronous cycle. This is also the case of modules in a synchronous path P_{sync} which verifies :

$$\sum_{m \in P_{sync}} T_{conc}(m) \leq T_{it}(C_{sync}), P_{sync} \subseteq C_{sync} \quad (7)$$

This means that the iteration of the last module in P_{sync} ends before the first module in P_{sync} begins a new iteration. We can now traverse the different components of G_{sync} to determine the paths which verify inequality 7. Moreover we only restrict our study to paths which start and end with modules mapped on the same node. Thus for each node n we obtain a set of synchronous path. Each path P_{sync} is defined between two modules m_1, m_2 such as $node(m_1) = node(m_2) = n$ and contains a set S of modules which can not run at the same time. Then we can remove from S modules which are not mapped on n. Thus we can map them on the same processor and the highest processor load required is equal to $max_{m \in S}(LD_c(m))$ and is noted $LD_c(S)$. Consequently if we map modules in S we replace the sum of their LD_c by $LD_c(S)$ in inequality 6.

If the inequality is still not verified on some nodes then the developer can slow down predecessors or he can modify its mapping. Since we compute the processor loads in our study, we can use this information to move some modules from nodes where the inequality is not verified to other available nodes.

3.3 Conclusion

With this information determined above on processor loads and on concurrent modules we are now able to determine a module-to-processor allocation for a given mapping. In usual applications, the number of modules mapped on nodes is generally comparable to the number of processors which is limited on current architectures. Thus we can easily map modules on processors since the number of possible allocations is reasonable. However chip makers are working on processors with even more cores. Consequently the allocation process will become even more complex. In this case it can be solved with the use of a solver based on allocation constraints obtained from equations 6 and 7.

Our method can also partially guide the developer in its mapping creation. Indeed, from allocation constraints, we can use a solver to generate mappings compatible with the performance expected without mapping information. This can be useful to increase performance, to minimize the number of nodes required for running an application or to run different applications on the same cluster. In [7] we have presented an approach to optimize communication schemes. In this case we can also define constraints on communication. The two approaches may be combined to optimize both module and communication performance. Even if we use the different constraints, a high number of mappings may still be possible. We can not give a general approach to optimize mappings since the mapping also depends on the performance expected by the developer. However, the developer can also add its own performance constraints to reduce the number of possible mappings. Thus we can further guide the developer in its mapping creation.

Since resulting allocations are static we can ensure performance to the developer. We are also able to evaluate the load of each processor and to detect if a node is overloaded. In this approach the allocation process does not depend on a particular OS scheduler. Consequently our performance prediction model is fully abstracted from the cluster and the operating system. We can now guarantee performance for heterogeneous distributed applications and interactivity in the case of Virtual Reality applications.

4 Experiments

4.1 Simple application

We first show the effects of the OS scheduling on a simple application. This application is composed of modules with different loads and execution times. Tests are performed on dual processor nodes. The operating system is based on the Linux kernel version 2.6.21. We use the `nice` command to set the priority of modules and the `taskset` command from the *Linux scheduler utilities*[2] to bind modules to processors.

[2] http://rlove.org/schedutils/

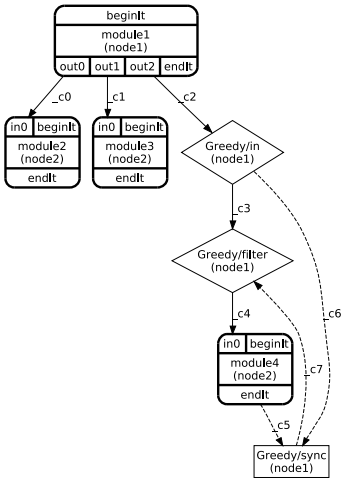

Fig. 3. First test application

We first consider an application composed of four modules on dual processor nodes as described in figure 3. Results are shown in table 1. We note that, on $node2$, the OS scheduler gives priority to modules $module2$ and $module3$ and maps them on different processors. Thus $module4$ is mapped with $module3$ which has the lowest load. However if we need better performance for $module4$ we have to define our own allocation. For example it is possible to map $module2$ and $module3$ on the same processor and $module4$ on the other processor. Thus, $module4$ can run at its full speed. Indeed, with this allocation we obtain better results for $module4$ as shown in table 1.

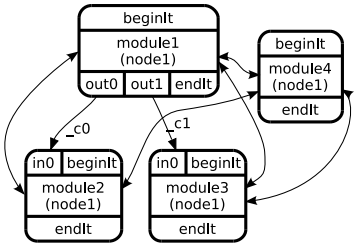

Fig. 4. The graph G_{dep} with bidirected edges between concurrent modules

We now study another application with modules mapped on the same dual processor node. The application graph is the same as in figure 3 but modules have different execution times and loads. This time we detect some interdependencies since we have cycles in G_{dep} (figure 4). Indeed, results in table 2 show that concurrent execution times are variable. In this case we propose a custom allocation to solve this problem. For example we can slow down $module1$ by allocating it a load of only 0.5. Then we determine

Module	**Node**	T_{exec}	LD	T_{conc}	
				OS scheduling	custom allocation
$module1$	1	22	0.5	22	22
$module2$	2	12	0.4	12	14
$module3$	2	9	0.3	9	12
$module4$	2	5	1	8	5

Table 1. (Times are given in ms)

Module	**Node**	T_{exec}	LD	OS scheduling		custom allocation		
				LD_c	T_{conc}	LD_c	T_{conc} pred.	T_{conc} real
$module1$	1	8	0.60	0.35-0.50	10-15	0.50	10	11
$module2$	1	6	0.65	0.45-0.55	7-9	0.40	10	9
$module3$	1	6	0.75	0.55-0.75	6-8	0.45	10	10
$module4$	1	5	1	0.60-1	5-8	0.60	8	9

Table 2. (Times are given in ms)

$LD_{cmin}(module2)$ and $LD_{cmin}(module3)$ from equation 4 and we choose to set their LD_c to these values. This allocation is described in table 2. Finally we map $module1$ and $module3$ on the first processor to maximize its use since the sum of the modules LD_c reaches 0.95. Consequently $module2$, with $LD_c(module2) = 0.4$, is mapped on the second processor and a load of 0.6 remains for $module4$. Results in table 2 show that predicted concurrent execution times are really close from measured ones.

4.2 The FluidParticle application

To verify the efficiency of our approach we now test it on a real FlowVR application. Our tests are performed on a cluster composed of two sets of eight nodes linked with a gigabit Ethernet network. The first set is composed of nodes with two dual-core Opteron processors while the second set is powered with dual Pentium4 Xeon processors.

Module	**Nodes**		**OS scheduling**		**Improved scheduling**				
		T_{exec}	T_{conc}	T_{it}	cpu	LD_{cmin}	LD_c	T_{conc}	T_{it}
$fluid$	1, 2, 3, 4	40	45-50	45-50	1, 2	1	1	40	40
$particles$	1, 2, 3, 4	9	9	45-50	4	0.225	1	9	40
$viewer$	1, 2, 3, 4	10	10	45-50	4	0.25	1	10	40
$renderer$	1, 2, 3, 4	10	10-20	10-20	3	1	1	10	10

Table 3. (Times are given in ms)

Module	**Nodes**		**Improved scheduling**				
		T_{exec}	cpu	LD_{cmin}	LD_c	T_{conc}	T_{it}
$fluid$	1, 2	40	1, 2, 3, 4	1	1	40	40
$particles$	11, ..., 14	13	1	0.325	1	13	40
$viewer$	11, ..., 14	15	1	0.375	1	15	40
$renderer$	11, ..., 14	20	2	1	1	20	20

Table 4. (Times are given in ms)

The FluidParticle application is composed of the following modules :

- *fluid* : this is an MPI version [6] of the Stam's fluid simulation [9].
- *particles* : this is a parallel module which stores a set of particles and moves them according to a force field.
- *viewer* : it converts the particles positions into graphical data.
- *renderer* : it displays informations provided by the viewer modules. There is one renderer per screen. In our study we use a display wall with four projectors thus we use four renderer modules on four nodes.
- *joypad* : it is the interaction module which converts user interaction into forces.

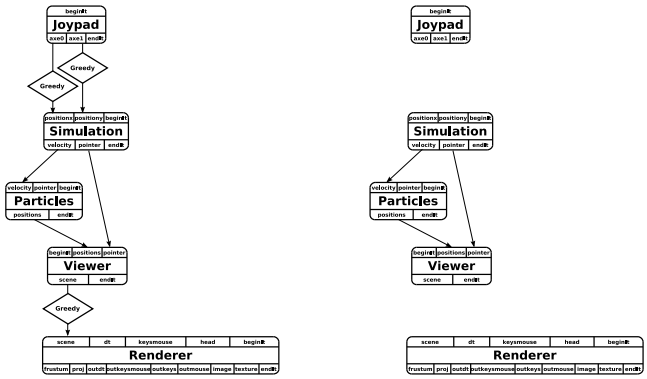

Fig. 5. A simplified graph of the application (left) and the corresponding G_{sync} (right)

A simplified application graph is described in figure 5. The *joypad* module is connected to the *fluid* module through a *greedy* filter since we want to interact with the simulation asynchronously. Then the *simulation* is connected with a FIFO connection to the *particle* module. The *simulation* also sends the position of the pointer to the *viewer* module. The *particle* module is connected to the *viewer* with a FIFO connection. Finally the *viewer* module sends messages to the *renderer* through a *greedy* filter.

We build G_{sync}, figure 5, to determine the different synchronous components. We obtain three different components. The first one only contains the *joypad* module. Since the *joypad* module only reads a position from the interaction device it has a very low execution time and load. Moreover it is the single module in the component and consequently it has no effect on the iteration time of other modules. Thus the *joypad* module is negligible for our study and we choose to ignore it. The second component contains the *fluid* modules which are predecessors, the *particles* modules and the *viewer* modules. The third component only contains the *renderer* modules which are predecessors.

We first test our application with the OS scheduling on four SMP nodes with two dual-core processors. We map one *renderer* per node since each node is connected to a projector. The *fluid* simulation is parallel and requires eight processors to run at an interactive frequency thus we map two *fluid* modules per node to take advantage of the SMP

nodes. We also map one *particle* and *viewer* module per node to reduce communication cost between modules. For the sake of clarity, a simplified version of the application graph with only two instances of each module is described in figure 6. Results are detailed in table 3. We note that the iteration times of both components are variable. For example the value of $T_{conc}(fluid)$ varies from 40 to 50ms with an average value close to 50ms. Indeed the scheduler can map several modules on the same processor and change their scheduling dynamically. Since *fluid* modules are synchronized, this affects the whole simulation. Thus we propose to use our approach to avoid variable results and to take a better advantage of the processors for the simulation. We choose to map the *fluid* modules on two dedicated processors since we need the best performance for the simulation. We choose to set $LD_c(particle) = 1$ and $LD_c(viewer) = 1$ since in this case we have $T_{conc}(particle) + T_{conc}(viewer) \leq T_{it}(fluid)$. This means that *particle* and *viewer* are never running at the same time. Thus we map them on one processor without performance penalty. The *renderer* module is consequently mapped on the fourth processor. The results of our improved scheduling are shown in table 3.

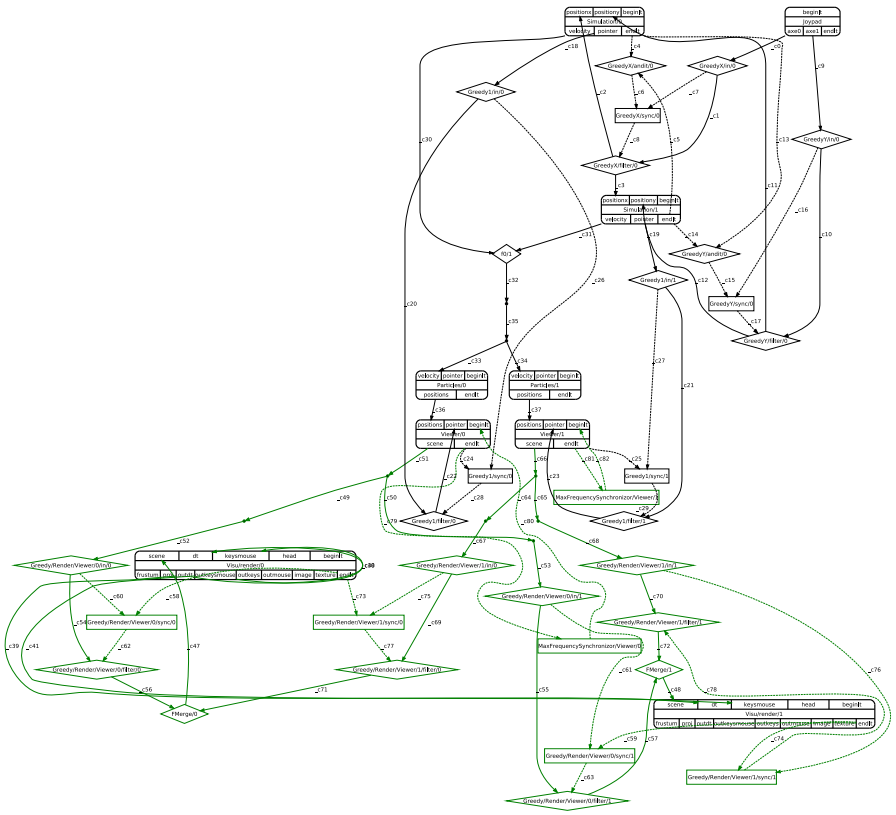

Fig. 6. A simplified graph of the FluidParticle application with only two instances of each module

We can also propose another mapping which requires more nodes but less computational power. Indeed the *renderer* module can be mapped on a node of the second cluster set. In this case the processor and the graphic card are less powerful but the frame rate is still interactive since we have $T_{exec}(renderer, n) = 20ms$ on a node n of the second set of nodes. We now consider mapping the *particle* and the *viewer* modules on the second set of processors. If we choose to allocate to each module a load of 1 then we still have $T_{conc}(particle) + T_{conc}(viewer) \leq T_{it}(fluid)$. Thus we can also map them on a single processor. Consequently we can use four nodes from the second set to host the *renderer*, the *particle* and the *viewer* modules. Then we have not enough processors available in the second set to run the simulation at 40ms per iteration. Thus we use eight processors from two nodes from the first set. Results are shown in table 4. Note that the application runs at an interactive frame rate but this improved mapping increases the latency between the *joypad* module and the *renderer* module. Indeed we increase T_{exec} for the *particle*, *viewer* and *renderer* modules, thus we increase the time needed between the interaction and the visualization of its effect. However the difference is negligible and is not perceptible by the user.

5 Conclusion

In this paper we have shown that the OS scheduling can lead to poor and variable performance for distributed applications. Thus we have presented a module-to-processor allocation process based on our performance model. The resulting allocation solves the drawbacks of the OS scheduling. Indeed it ensures performance and stability of FlowVR distributed applications. This allocation depends on our performance model and also of the performance expected by the developer. Consequently it is abstracted from a particular operating system and only driven by performance information. Thus we can consider cluster with heterogeneous operating systems. Our approach can also be applied to several applications running on the same cluster since we can consider them as a single application with more synchronous components. This can be useful to optimize both the performance of applications and the use of the cluster.

We also show that the mapping creation can be guided by different kind of constraints. These constraints are given by our model, by the hardware or software required by modules and by the user to respect its performance expectation. For example a module must have the same iteration time as its input modules in our model, an interaction module must be mapped on the node connected to the interaction device or a module must respect an iteration time given by the developer. Thus we plan to develop a solver to solve these different constraints. The goal is to assist the developer in its mapping creation or to provide an automatic generation of mappings. It will also enables to consider large heterogeneous applications with even more modules and connections on large clusters with heterogeneous nodes connected with complex network topologies.

References

1. J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. http://citeseer.ist.psu.edu/aas05understanding.html.

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, 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.
4. J. Allard and B. Raffin. Distributed physical based simulations for large vr applications. In *IEEE Virtual Reality Conference*, Alexandria, USA, March 2006.
5. D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*, chapter 7. Oreilly, 2005.
6. R. Gaugne, S. Jubertie, and S. Robert. Distributed multigrid algorithms for interactive scientific simulations on clusters. In *Online Proceeding of the 13th International Conference on Artificial Reality and Telexistence, ICAT*, december 2003.
7. S. Jubertie and E. Melin. Multiple networks for heterogeneous distributed applications. In *Proceedings of PDPTA'07*, Las Vegas, june 2007.
8. S. Jubertie and E. Melin. Performance prediction for mappings of distributed applications on pc clusters. In *Proceedings of IFIP International Conference on Network and Parallel Computing, NPC*, Dalian, China, september 2007. *To appear*.
9. J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003.