Performance Prediction for Mappings of Distributed Applications on PC Clusters

Sylvain Jubertie, Emmanuel Melin

Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
Université d'Orléans
Email: {sylvain.jubertie | emmanuel.melin}@univ-orleans.fr
http://www.univ-orleans.fr/lifo

Abstract. Distributed applications running on clusters may be composed of several components with very different performance requirements. The FlowVR middleware allows the developer to deploy such applications and to define communication and synchronization schemes between components without modifying the code. While it eases the creation of mappings, FlowVR does not come with a performance model. Consequently the optimization of mappings is left to the developer's skills. But this task becomes difficult as the number of components and cluster nodes grow and even more complex if the cluster is composed of heterogeneous nodes and networks. In this paper we propose an approach to predict performance of FlowVR distributed applications given a mapping and a cluster. We also give some advice to the developer to create efficient mappings and to avoid configurations which may lead to unexpected performance. Since the FlowVR model is very close to underlying models of lots of distributed codes, our approach can be useful for all designers of such applications.

1 Introduction

Today, clusters are theoretically able to reach the performances needed by large simulations because they are extensible. This is an interesting property since it does not limit the simulation complexity or the amount of data to consider. However clusters bring new programming problems : it is more complex to produce efficient applications on distributed memory architectures than on shared memory ones. Several communication libraries like MPI or PVM provide point-to-point communications and synchronisations to program clusters efficiently. VR platforms were also ported to clusters to exploit their performances. For example the NetJuggler [7] environment allows to drive interactive applications with parallel simulations and a distributed rendering. These approaches are very interesting but are limited to simple applications assumed to run on homogeneous clusters. For example the model behind NetJuggler is too synchronous because the rendering rate is too dependant of the simulation rate [3].

Consequently we should add more asynchrony between the application parts. For example an interaction and a simulation code should be connected but not

synchronized if we want to keep an interactive behaviour because the simulation often have lower frequencies than interaction devices. In this case, we want the simulation to receive interaction data asynchronously even if some are lost. We say that they are linked by *greedy communications*.

Once we have described how to synchronize the application parts, then we can map them on the cluster processors. Many choices are possible depending on the underlying nature of the cluster which may be composed of heterogeneous nodes, peripherals and networks. This mapping is not straightforward and affects the application performance. Consequently, we need a framework that eases mapping operations by catching the parameters of each application part and abstracting the architecture. This framework should also be associated with a performance model to tune efficient mappings.

The FlowVR library[2][4] was created to ease the development of distributed interactive applications and to permit greedy communication. But FlowVR does not offer a way to obtain the best application mapping on a given cluster nor any kind of performance information. Thus the developer should use his experiments and test several configurations to find a good mapping. But this task may become too complex for applications with many parts on heterogeneous clusters such as the application presented in [4] which integrates 5000 different objects.

We propose in this paper a unified approach to analyse at the same time synchronization, concurrency and network constraints. Thus the developer is able to associate performance information to his mappings. For example he can determine information like the frequency of each module, the load on each processor and the communication times for each connection.From these informations the developer can determine if its mapping is well suited and can run on the cluster. Otherwise our approach is able to detect and point out network bottlenecks and modules with low performance. Then the developer can detect parts of the application to optimize and adapt its mapping.

2 The FlowVR framework

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 modules 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.

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 network component called a *filter*. Synchronization and coupling policy are performed with another network component

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. This filter allows 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.

3 Performance prediction

We now present our approach to compute performance information for a FlowVR mapping on a cluster. Then the developer will be able to determine if his application runs as expected or to compare several mappings to find the best one.

3.1 Model inputs

A mapping is a FlowVR network enriched with information on the location of modules in the cluster, and on networks used for communication. A cluster is defined as a set of nodes $Nodes$ and a set of networks $Networks$. To deal with SMP nodes, each node $n \in Nodes$ has a list of CPUs given by the function $CPUs(n)$. A node can also have several adapters connected to different networks. Thus each node n is associated to a list of networks $Nets(n) \subset Networks$. Each network $net \in Networks$ has a bandwidth $BW(net)$ and a latency $L(net)$. We assume networks with point-to-point connections in full-duplex handled by dedicated network adapters without CPU overload. We also assume that communication between objects mapped on the same node are costless since objects only exchange pointers to a shared memory. The FlowVR network is a graph G composed of a set of vertices V and a set of directed edges E. Each $v \in V$ represents a FlowVR object i.e. a module, a filter or a synchronizer. Each $e \in E$ represents a connection between a source objects $src(e)$ and a destination object $dest(e)$ with $src(e), dest(e) \in V$. To build a mapping the developer binds FlowVR objects and connections respectively to cluster nodes and networks. We denote the location of an object $v \in V$ by the function $node(m)$ which gives a node $n \in Nodes$. Note that the developer has to map modules on nodes but modules are then mapped on processors by the operating system scheduler. The network used by a connection e is given by the function $Net(e)$ which returns a network $net \in Networks$. If two connected objects are on the same nodes the connection is local : $Net(e) = \emptyset$. Otherwise the connection is associated to a network $net \in Networks$ such as $Net(e) = net$.

Our approach implies that the developer must give extra information on modules to compute performances. For each module $m \in V$ we need to know its execution time $T_{exec}(m)$ and its load $LD(m)$ on the host processor. The execution time $T_{exec}(m)$ is the time needed by a module m to perform one iteration

when m is not synchronized with other modules and have no concurrent modules. The load $LD(m)$ is the percentage of $T_{exec}(m)$ used for the computation. The rest of $T_{exec}(m)$ is used for I/O operations. For each edge $e \in E$ we need to know the volume of data $Vol(e)$ sent by $src(e)$ through e during one sole iteration. If $src(e)$ is a module then $Vol(e)$ is equal to the amount of data sent by v through the output port connected to e. If $src(e)$ is a filter then $Vol(e)$ depends on the filter characteristics. For example the *merge* filter sends only one message built from all messages it received. If $src(e)$ is a synchronizer then for the sake of simplicity we assume that $Vol(e) = 0$. Indeed messages sent and received by synchronizers contain only stamps. Consequently their message sizes are negligible compared to the amount of data sent by modules. We also assume that filters and synchronizers have a negligible load compared to module loads. Indeed they only perform some memory operations on messages. The value of $Vol(e)$ is independent of the hardware and is statically determined from the module characteristics. Values of $T_{exec}(m)$ and $LD(m)$ can be determined in different ways. For example the developer can measure them by running each module separately on the target node. On the other hand, FlowVR allows to reuse modules from other applications and $T_{exec}(m)$ and $LD(m)$ may be already available.

3.2 Determining performance

Performance of modules depend on synchronization and concurrency between them. Thus we need to determine for each module m its iteration time $T_{it}(m)$ and its concurrent execution time $T_{cexec}(m)$. We define $T_{it}(m)$ as the time between two consecutive calls to the FlowVR *wait* function. This definition characterizes the real frequency $F(m)$ of a module execution for a given mapping :

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

We define $T_{cexec}(m)$ as the execution time of m when several modules are running on the same node. Indeed, executions of concurrent modules are interleaved by the OS scheduler. Thus we always have $T_{cexec}(m) \geq T_{exec}(m)$. If m has no concurrent modules then :

$$T_{cexec}(m) = T_{exec}(m) \tag{2}$$

We determine $T_{cexec}(m)$ according to a scheduler policy. But this policy strongly depends on the time a module waits for I/O and is blocked in the FlowVR *wait* function. We first study the effects of synchronization on performances. Then we will evaluate how the concurrency between modules affects their performances.

Determining T_{it} from synchronization. In this section we examine how synchronization between modules affect their iteration time. For a module m we define its input modules $IM(m)$ as the set of modules with edges connected to m. We distinguish two subsets of $IM(m) : IM_s(m)$ and $IM_a(m)$ such as $IM_s(m) \cup IM_a(m) = IM(m)$ and $IM_s(m) \cap IM_a(m) = \emptyset$. The subsets $IM_s(m)$ and $IM_a(m)$ contain respectively the modules connected to m through FIFO connections and through *greedy* filters.

We first consider the effect of *greedy* connections on performance. A module m receiving data through *greedy* filters does not wait for messages from modules in $IM_a(m)$. Indeed a *greedy* filter always provide a message which is the last

one available. This means that $T_{it}(m)$ does not depend on synchronizations with modules in $IM_a(m)$. Consequently m is like a module with only FIFO connections. Moreover if $IM_s(m) = \emptyset$ then $T_{it}(m)$ only depends on concurrency with other modules :

$$T_{it}(m) = T_{cexec}(m) \tag{3}$$

Thus to study the effect of synchronization on performance we can remove *greedy* filters from G. We obtain a new graph called G_{sync}. We note that G_{sync} may not be connected anymore and may be splitted into several synchronous components. Since components are not linked we can study each one independently.

We now consider each module m in a component $C \in G_{sync}$. If $IM_s(m) \neq \emptyset$ then m is synchronized with its input modules. To begin its iteration, m must receive messages from each module in $IM_s(m)$. If m is slower than its $IM_s(m)$ then $T_{it}(m) = T_{cexec}(m)$. Otherwise, it must wait for the slowest module in $IM_s(m)$ which determines its $T_{it}(m)$. Thus we have :

$$T_{it}(m) = max(max(T_{it}(i), \forall i \in IM_s(m)), T_{cexec}(m)) \tag{4}$$

If $IM_s(m) = \emptyset$ then m is not synchronized with other modules. We called these modules *predecessors* and we define $preds(C)$ as the set of *predecessors* in a component C. Their T_{it} is given by equation 3 since they are not synchronized. We can also have $preds(C) = \emptyset$. Indeed modules in C can be organized in synchronous cycles. In this case we have at least a predecessor cycle G_{pc} such as for each module m in G_{pc}, $IM_s(m) \in G_{pc}$. Note that we may have both predecessors and predecessor cycles in C. In the case of a predecessor cycle G_{pc}, each module $m \in G_{pc}$ waits only for other modules in G_{pc}. Consequently $T_{it}(m)$ depends on the T_{cexec} of other modules in G_{pc} and on the communication time between modules in G_{pc}. For each module $m_c \in G_{pc}$ we have :

$$T_{it}(m_c) = \sum_{m \in G_{pc}} T_{cexec}(m) + \sum_{\substack{e \in G_{pc} \\ Net(e) \neq \emptyset}} \left(\frac{Vol(e)}{BW(Net(e))} + L(Net(e))\right) \tag{5}$$

According to equations 3, 4 and 5 we need $T_{cexec}(m)$ for each m to obtain $T_{it}(m)$.

Determining T_{cexec} for concurrent modules. We turn to study consequences on concurrency on modules performances to compute their T_{cexec}. The behaviour of concurrent modules on a node n is determined by the OS scheduler. Our approach is based on the Linux scheduler policy [1][5] which gives priority to a module over others according to the time each concurrent module waits. In this case the more a module waits, the higher priority it gets. Therefore to determine $T_{cexec}(m)$ for each module m we first need the time spent for I/O operations and for the FlowVR *wait* function. A predecessor pm is not synchronized and only waits for I/O operations according to its $T_{exec}(m)$ and $LD(m)$. For each predecessor pm, we define $T_{I/O}(pm)$ as follow :

$$T_{I/O}(pm) = T_{exec}(pm) \times (1 - LD(pm)) \tag{6}$$

If $IM_s(m) \neq \emptyset$ then m If a module m is synchronized with its input modules then we define $T_{I/O}(m)$ as the time not used for the computation during an iteration :

$$T_{I/O}(m) = max(T_{exec}(m), T_{it}(i), \forall i \in IM_s(m)) - T_{exec}(m) \times LD(m) \tag{7}$$

With $T_{I/O}(m)$ we can sort modules on each node n in a list $l(n)$ from the one with the highest $T_{I/O}(m)$ to the one with the lowest $T_{I/O}(m)$. Then we consider

modules in the list order. Each module m is mapped on the most available CPU i.e. the CPU with the lowest load, and receives a concurrent load $LD_c(m)$ on this CPU according to its load $LD(m)$. Finally, we use the ratio between $LD(m)$ and $LD_c(m)$ to evaluate $T_{cexec}(m)$. Algorithm 1 describes this process. Note that some modules may have the same $T_{I/O}$, in this case the order between them is arbitrary. Our tests show that the scheduler can choose one possible order but if we run the application several times the scheduler can choose another possible order. Thus we have no performance garantee but we are able to detect when this case occur.

Algorithm 1 Computation of Tcexec

```
for all cpu ∈ CPUs(n) do
  CPULD(cpu) = 0
end for
while l(n) ≠ ∅ do
  m = head(l(n))
  l(n) = tail(l(n))
  load = 1
  for all cpu ∈ CPUs(n) do
    if CPULD(cpu) < load then
      p = cpu
      load = CPULD(cpu)
    end if
  end for
  LD_c(m) = (1 − CPULD(p))) × LD(m)
  CPULD(p) = CPULD(p) + LD_c(m)
  T_cexec(m) = T_exec(m) × LD(m)/LD_c(m)
end while
```

In this approach $T_{I/O}(m)$ is determined from $T_{it}(i), i \in IM_s(m)$ from equation 7. But $T_{it}(i)$ may depend on $T_{cexec}(i)$ according to equations 3, 4 and 5. For example if i is a predecessor, $IM_s(i)) = \emptyset$, then $T_{it}(i)$ depends on $T_{cexec}(i)$ from equation 3. Then if m and i are mapped on the same node then we can not compute $T_{cexec}(i)$ since we have not yet determined $T_{I/O}(m)$ and $T_{I/O}(i)$ which depend on $T_{it}(i)$ from equation 7. Consequently, in this example we have an interdependency between equations 3 and 7. To detect interdependencies we first modify G_{sync} to represent concurrency between modules. Therefore we add bidirected edges between concurrent modules in G_{sync}. We obtain a new graph G_{dep} were each edge represents a dependency due to synchronization (directed edges) or concurrency (bidirected edges). If we detect a cycle in the graph then we can have an interdependency between modules in the cycle. We define a cycle as a path between a module and itself such as this path is not empty. Note that a cycle can contain the same bidirected edge twice but not the same directed edge.

We turn to present how to determine $T_{cexec}(m)$ and $T_{it}(m)$ for each module m in G_{dep}. Note that G_{dep} may not be connected, in this case G_{dep} has several components. Since there is no dependencies between components of G_{dep} we can study separately each one. A component C_{dep} can contain cycles of different nature and Directed Acyclic Graphs. We propose to extract cycles from C_{dep} to obtain a set D_{dep} of DAGs. Then we study cycles and DAGs independently.

If we consider a DAG d in D_{dep} then we have no concurrency between modules because we have no bidirected edges between them. Thus from equation 2 we have $T_{cexec}(m) = T_{exec}(m)$ for each module $m \in d$. If d contains a predecessor pm then from equations 2 and 3 we can determine $T_{it}(pm)$. Then we propagate this value to each module m such as $IM_s(m) = pm$ to determine $T_{it}(m)$ from equations 2 and 4. If, for a module m we have $IM_s(m) \not\subset d$ then it means that it is dependant of a module in a cycle. Consequently we must first study this cycle. Note that different kinds of cycles may be present in C_{dep}.

We first consider a cycle $C_{cycle} \subset C_{dep}$ with only bidirected edges i.e. all modules in C_{cycle} are on the same node and but from distinct components. If C_{cycle} contains only predecessors then we determine $T_{I/O}(pm)$ for each $pm \in C_{cycle}$

with equation 6. Otherwise if we have at least one non predecessor module m then we use equation 7. But we need to first study parts of the graph which contain $IM_s(m)$. If C_{cycle} contains only directed edges then C_{cycle} is a synchronous cycle. Moreover each module m within C_{cycle} has no concurrent modules. Consequently we have $T_{cexec}(m) = T_{exec}(m)$ from 2. If C_{cycle} is a predecessor cycle then we use equations 2 and 5 to obtain $T_{it}(m)$. Otherwise, for each module m in C_{cycle} with $IM_s(m) \not\subset C_{cycle}$ we first need to study parts with modules in $IM_s(m)$. Then we apply equation 4 to modules in C_{cycle}. We finally consider cycles with both directed and bidirected edges. In this case we have an interdependency and we can not sort modules. To solve this problem we propose to choose an order between modules. For example, we consider that modules in the same synchronous component C have the same iteration time. Indeed if we have $m \in C$ such as $T_{cexec}(m) > T_{it}(i), i \in IM_s(m)$ then m is slower than i. In this case messages from i are accumulated and generate a buffer overflow. Thus our hypothesis seems appropriate and desirable for the developer. But this single iteration time is not yet determined. We are nonetheless able to compare concurrent modules in the same component C. Indeed if we consider $m_1, m_2 \in C$ and $\in C_{cycle}$ with $node(m_1) = node(m_2) = n$ we have $T_{it}(m_1) = T_{it}(m_2)$ according to our hypothesis. If m_1 and m_2 are not predecessors of C we have from equation 7 :

$$T_{I/O}(m_1) - T_{I/O}(m_2) = T_{exec}(m_2) \times LD(m_2) - T_{exec}(m_1) \times LD(m_1)) \quad (8)$$

Consequently it comes to compare the time each module effectively uses the CPU. Note that, if we have a predecessor $pm \in C$, or a module m from another component, then we are not able to compare them. Consequently we distinguish two possible configurations. In the first one we have only modules from the same component on a node n. According to our hypothesis we are able to sort them and we can solve the interdependency. On the other hand if we have a predecessor pm, or a module m from a different component in C_{cycle}, then our hypothesis does not allow to compare them. In this case we propose to set $T_{cexec}(m) = T_{exec}(m)$ for each $m \in C_{cycle}$, just to define an order. Then we are able to determine $T_{cexec}(m)$ for each module m and then $T_{it}(m)$. At this step we can verify the order. If the order has changed we repeat the process but we can not guarantee that this process always converge. In this case our tests show oscillations of the execution time due to variations in the module order. This behaviour does not correspond to the one expected for performance, especially for interactive applications which performance has to be stable. Moreover this dynamic variation of performance due to the scheduling can be very difficult to detect and to analyse. Our method makes possible to detect when this behavior may occur and to precisely point out modules in these configurations. With this information the developer can change its mapping or can tune the scheduler to sort modules statically.

We now construct C_{dep} from these different parts. We first consider cycles and DAGs which are not dependent of others. The graph contains such parts since we have extracted cycles from it. For each module m in these "predecessor parts" we have determined $T_{cexec}(m)$ and $T_{it}(m)$. Then we merge parts which depends on these "predecessor parts" and we can compute $T_{cexec}(m)$ and $T_{it}(m)$ for each module m in them. We repeat the process for the other parts until we complete the graph. Once we have determined $T_{cexec}(m)$ and $T_{it}(m)$ for each

$m \in G_{sync}$ we verify that $T_{cexec}(m) \leq T_{it}(i), i \in IM_s m$. If this is not the case for a module m then we predict a buffer overflow on $node(m)$. The developer can remove the buffer overflow in different ways. For example he can distribute m on several nodes to decrease $T_{exec}(m)$ and consequently $T_{cexec}(m)$. If is also possible to map concurrent modules of m on other nodes to decrease $T_{cexec}(m)$.

We can now determine performance for a given mapping. We also provide to the developer a way to detect incorrect mappings. In this case our analysis point out modules which generates errors and propose a mean to solve them.

Networking. We now consider communication between FlowVR objects. We begin our study with a traversal of the application graph $G(V, E)$ to determine the frequency $F(f)$ of each filter f, and $Vol(e)$ on its output ports. 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 and we have $F(f_{greedy}) = F(m_{dest})$. We also determine $Vol(e)$ from the frequency and the behaviour of objects. Note that we can add additional edges to represent communication out of the FlowVR communication scheme, for example MPI connections. Then we can compute the bandwidth bw_s needed by a node n to send its data on a network net:

$$bw_s(n, net) = \sum_{\substack{\forall e \in E, \\ Net(e)=net, \\ node(src(e))=n}} Vol(e) \times F(src(e)) \tag{9}$$

If, for a node n, we have $bw_s(n, net) > BW(net)$ then messages can not be sent through the network thus we can predict a buffer overflow on n. We can also determine the bandwidth bw_r needed by n to receive its data by replacing $node(src(e)) = n$ by $node(dest(e)) = n$ in equation 9. If $bw_r(n, net) > BW(net)$ then too much data are sent to the same node, leading to a buffer overflow on nodes sending data to node n through network net. Our method gives the developer the ability to point out network bottlenecks in his mappings. Then it is possible to remove them by reducing the number of modules on the same node, by modifying the communication scheme, or by using other networks.

We now study the *latency* between modules. It represents the time an information needs to be processed and transported through the mapping. In VR applications the latency is critical for user interaction and visualization. We determine the latency between two modules m_1 and m_2 from the path P between them. The path P is provided by the developer and contains a set of FlowVR objects and edges between them. The latency is obtained by adding concurrent execution times of modules in P and communication times.

$$L(P) = \sum_{m \in P} T_{cexec}(m) + \sum_{\substack{\forall e \in P \\ Net(e) \neq \emptyset}} \frac{V(e)}{BW(Net(e))} + L(Net(e)) \tag{10}$$

The developer can detect whether the latency corresponds to its requirements, for example if it is low enough for interactivity. If the latency is too high, the developer can minimize it by mapping several modules on the same node to decrease communication latencies. He can also create more instances of parallel modules to decrease execution times or to use a faster network.

4 Tests

In this section we present several tests to validate our performance prediction model on simple FlowVR applications. Then we apply our method to a real application. Tests are performed on a cluster composed of two sets of eight nodes linked with a gigabit Ethernet network. The first set (nodes 1 to 8) is composed of nodes with two Opteron processors, each one with two cores. The second one (nodes 11 to 18) is composed of nodes with dual Pentium4 Xeon processors.

4.1 Test application

We first verify our model on simple FlowVR applications.We first determine for each module m its $T_{exec}(m)$ by running independently each module on the destination host. Then we run the applications to compare predictions to results.

Synchronizations. We first consider a greedy connection between two modules m_1, m_2 mapped on different nodes. Results are shown in table 1 and confirm that greedy connections do not affect module performance. Then we replace the greedy connection by a FIFO connection. Results are shown in table 2. As expected $T_{it}(m_2) = T_{it}(m_1)$. Finally we invert the FIFO connection between m_1 and m_2. In this case we predict a buffer overflow since $T_{exec}(m_2) > T_{it}(m_1)$. Our tests confirm that the application exists with a buffer overflow error.

We turn to consider three modules organized in a synchronous cycle. Since each module waits for the others, two modules can not run at the same time. Thus we predict that $T_{cexec} = T_{exec}$ for each module if they are mapped on the same node. But we should have a higher T_{it} if modules are mapped on distinct nodes since we have network communications. We first map modules on distinct nodes. Each module sends 5MB per iteration through a gigabit network (BW=100MB/s) thus we expect that each communication will take around 50ms. We assume that the network latency is negligible compared to this communication time. We have three connections in the cycle so we add 150ms to the execution times in equation 5. Results shown in table 3 are close to predictions even with a simple estimation of the network parameters. If we map

Module	*Nodes*	*LD*	T_{exec}	T_{it} pred.	T_{it} real
m_1	1	1	37	37	37
m_2	2	0.5	18	18	18

Table 1.

Module	*Nodes*	*LD*	T_{exec}	T_{it} pred.	T_{it} real
m_1	1	1	37	37	37
m_2	2	0.5	18	37	37

Table 2.

Module	*Nodes*	*LD*	T_{exec}	T_{it} pred.	T_{it} real
m_1	1	1	37	234	240
m_2	2	0.5	26	234	240
m_3	3	0.5	21	234	240

Table 3.

Module	*Nodes*	*LD*	T_{exec}	T_{it} pred.	T_{it} real
m_1	1	1	37	84	84
m_2	1	0.5	26	84	84
m_3	1	0.5	21	84	84

Table 4.

				Prediction			*Measure*	
Mod.	*Node*	T_{exec}	*LD*	$T_{I/O}$	T_{cexec}	LD_c	T_{cexec}	LD_c
m_1	1	20	1.00	0	48	0.42	36	0.55
m_2	1	16	0.30	11	16	0.30	19	0.25
m_3	1	10	0.50	5	14	0.35	17	0.44
m_4	1	51	0.58	20	51	0.58	56	0.52

Table 5. (Times are given in ms)

modules on the same node then we only sum the execution times to obtain the T_{it} of modules in the cycle from equation 5. Results in table 4 show that T_{it} is correctly predicted by our approach. We note that communication through the shared memory does not add extra latency.

Concurrency. In this test we consider four different modules m_1, m_2, m_3 and m_4 mapped on a dual processor node. These modules are not synchronized to avoid interdependencies since we want to validate our scheduling model. We apply our approach to determine $T_{cexec}(m)$ for each module m. Results in table 5 are close to our predictions. Nonetheless we note that the scheduler gives the higher priority to m_2 and m_4 but does not give them the necessary load.

4.2 The FluidParticle application

We now apply our approach on our FluidParticle application which is used to observe typical fluid phenomena like vortices. It contains 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 primitives.
- *renderer* : it displays informations provided by the viewer modules. In our study we use a display wall with four projectors thus we use four renderer modules on four distinct nodes.
- *joypad* : it is the interaction module which allows the user to interact with the fluid by adding forces.

Our goal is to obtain an interactive application. We focus our study on synchronization and concurrency effects on performance. A complete example of network performance analysis can be found in [8]. We first determine $T_{exec}(m)$ and $LD(m)$ for each module m (table 6). We note that the *joypad* module has load under 1% and is connected to simulation modules through *greedy* filters to allow an asynchronous interaction. Consequently it can not involve performance penalties and we choose to ignore it.

We now describe communication and synchronization between modules. The *fluid* module is connected synchronously with the *particles* module. The *particles* module is also connected synchronously with the *viewer* module. Finally the *viewer* and the *renderer* modules are connected through a *greedy filter*. This allows to change the user point of view and to update data from the *viewer* module asynchronously. If we remove greedy connections then the graph is splitted into

Module	*Nodes*			*Prediction*			*Measure*		
		LD	T_{exec}	T_{cexec}	T_{it}	LD_c	T_{cexec}	T_{it}	LD_c
fluid	1, ..., 8	0.97	70	70	70	0.97	73	73	0.97
particles	15, ..., 18	0.97	20	20	70	0.28	21	73	0.30
viewer	15, ..., 18	0.97	28	28	70	0.40	28	73	0.38
renderer	11, ..., 14	0.97	57	57	57	0.97	60	60	0.97
joypad	1	<0.01	<1	0	0	0	0	0	0

Table 6. (Times are given in ms)

two components. The first one contains the *fluid*, the *particles* and the *renderer* modules while the second one contains the *renderer* modules.

We turn to study synchronization and concurrency between modules for two mappings. We first propose a mapping with the *fluid*, *particles* and *renderer* modules on the same dual processor nodes. In this case we detect an interdependency since we have a cycle with a directed edge between the *fluid* and the *particles* modules, and a bidirected edge between the *renderer* and the *fluid* modules. The *particules* module is synchronized with the *fluid* module and have a lower execution time. The *renderer* module is the single module in its component and is consequently a predecessor. Since both the *renderer* and the *fluid* modules have a load of 97%, the *particles* module always have the highest priority. Consequently *renderer* or the *fluid* module will be mapped with the *particles* module on the same processor. But we can not order these two modules. Thus the scheduler may change their mapping on the two processors dynamically. Indeed our tests confirm that their T_{cexec} vary. We now propose a different mapping.

To obtain an interactive visualization we should map the *renderer* modules on dedicated processors. We also need to avoid concurrency for the *fluid* module to obtain the fastest simulation. Thus we propose to map modules as described in table 6. In this mapping we use nodes 1 to 8 for the simulation and we distribute four modules on each node to take advantage of the four processors. Then we map the *renderer* module on four nodes connected to four projectors to visualize the simulation on our display wall. Four nodes, with two processor on each one, are still available for the *particles* and *viewer* modules. Consequently we distribute them on these nodes to reduce their execution time. In this last case we have a cycle with only modules from the same synchronous component on each one of these nodes. Moreover we do not have a predecessor mapped with them. Thus we are able to determine their T_{it}. Results of this mapping are shown in table 6. We note that it confirms the predicted performance. However in this mapping each module is mapped on a dedicated processor to avoid interdependencies. We note that, if we want to optimize the use of the cluster, we can bind modules on processor to avoid interdependencies. Moreover, we have $T_{exec}(particles) + T_{exec}(viewer) < T_{it}(fluid)$. This means that each message from the *fluid* module is processed by the *particles* module which then sends a message to the *viewer* and waits for a new message. Then the message is processed by the *viewer* module which then waits for a new message from the *particles* module. But a new message is not yet available from the simulation. Consequently the next *particles* iteration can not start before the end of the *viewer* iteration. Thus the *particles* and the *viewer* modules are never concurrent and we can bind them to the same processor. We propose to modify the previous mapping by moving the *particles* and *viewer* modules to nodes 1 to 4 and to bind them on the second processor to avoid an interdependency with the *renderer* module. Our tests confirm that we obtain the same performance with this mapping.

We have applied successfully our approach on our interactive simulation. In each case we take into account synchronization and concurrency to determine performances of modules. We also detect mappings with poor performance.

5 Conclusion

We have shown in this paper that our approach is able to predict performances for distributed FlowVR applications. Thus the developer can determine if its mapping offers for each module the frequency he expected. He can also compare the execution time of a module to the concurrent execution time and then observe the effects of concurrency between modules. For each node we are able to compute the load of each processor. If the developer needs more performances our approach allows to point out modules which could be optimized. Then he can choose to map modules on nodes with lower processor loads or to distribute a module on several nodes. But this can generate more communications on the network. Nevertheless our method allows to determine consequences of such choices. We can point out modules which generates buffer overflow due to synchronizations. We can also locate bottlenecks on network links.

This approach brings to the FlowVR model a way to abstract the performance prediction from the code. Nevertheless it is not limited to FlowVR applications and is sufficiently general to consider applications developed with other distributed middleware. The next step in our approach is to enhance the scheduling of concurrent modules to improve performance. We also plan to provide automated tools based on our model to assist the developer in his mapping creation and optimization.

Acknowledgment

This work is supported by the Region Centre.

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, V. Gouranton, E. Melin, and B. Raffin. Parallelizing pre-rendering computations on a net juggler pc cluster. In *Proceedings of the IPT 2002*, Orlando, Florida, USA, March 2002.
4. J. Allard, C. 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.
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 *ICAT*, 2003.
7. E. Melin J. Allard, V. Gouranton and B. Raffin. Parallelizing pre-rendering computations on a Net Juggler PC cluster. In *IPTS 2002*, 2002.
8. S. Jubertie and E. Melin. Multiple networks for heterogeneous distributed applications. In *Proceedings of PDPTA'07*, Las Vegas, 2007. *To appear.*
9. J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003.