# 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. It is difficult to perform this task when the number of components and cluster nodes grow. Moreover, the cluster may be composed of heterogeneous nodes making this task even more complex. In this paper we propose an approach to predicting 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.

## I. Introduction

Today, clusters are theoretically able to reach the performances needed by large simulations because they are extensible. This is the most interesting property because 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 and PVM provide point-to-point communications and synchronisations to program these architectures efficiently. VR platforms were also ported to clusters to exploit their performances. For example the NetJuggler [10] 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 asynchronism between the different application parts. For example the interaction codes and the simulations codes should be connected but not synchronized if we want the application to keep an interactive behaviour because the simulations often have lower frequencies than haptic devices. In this case, we want the simulation to receive interaction data asynchronously even if some are lost. We also need asynchronism with the visualization. When the user wants to change his point of view, the visualization should change it interactively without waiting for the simulation. In both cases each part of the application is able to communicate without waiting for the other. We say that they are linked by *greedy communications*.

Once we have described how to synchronize the different parts of the application, then we can map these parts 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 may have effects on the application performance. For example, mapping several parts on the same node may decrease latency between them by avoiding network communications, but it introduces concurrency which may reduce computation times. In the case of a distributed simulation, increasing the number of processors used may decrease the computation time but it may increase the network load. Consequently, we need a framework that eases mapping operations by catching the parameters of each application part and abstracting the architecture. This framework should also be associated with a performance model to tune efficient mappings on distributed architectures.

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

But this framework doesn't offer a way to obtain the best application mapping on a given cluster nor any kind of performance information. Without such information 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 modules on heterogeneous clusters such as the application presented in [4] which integrates 5000 different objects.

We have shown in [11] a simple performance analysis adapted to multi network clusters. Indeed the network is the main limitating factor for performance on distributed architectures. Our experiment have shown that we also need to take into account synchronization and concurrency in this kind of applications and especially in interactive applications.

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

## II. Performance models for parallel programming

We examine the existing parallel models associated with cost models.

### *A. PRAM : Parallel Random Access Machine*

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

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

### *B. BSP : Bulk Synchronous Parallel*

In the BSP model [16] a computation is defined by a sequence of supersteps : asynchronous computation followed by a global communication step, and a synchronisation barrier. The cost of a BSP algorithm is defined by the input size and by several architectural parameters. This model has interesting properties : it is architecture independent and the performance of a BSP program is predictable for a given architecture.

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

The BSPWB model[14], a BSP Without Barrier, proposes a generalisation of the superstep to a Message step (M-step) : a local computation followed by a data emission and reception. Global barriers are removed because processors may be in different M-steps at the same time. But two M-steps can only communicate if they are adjacent which limits the possible asynchronism between M-steps. Moreover greedy communications are not possible in this model.

### *C. LogP*

The logP model [7] was developed specifically for distributed memory architectures with point-to-point connections. The goal is to obtain a more realistic cost than the PRAM model by taking into account the communication cost. This model is asynchronous to reflect the intrinsic nature of distributed memory architectures and to obtain better performances than the BSP model without the need of expensive global synchronizations.

The LogP model uses four parameters to catch the principal characteristics of distributed memory architectures : the communication delay, the communication overhead for the management of the network interface, the communication bandwidth and the number of processors.

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

This model allows to obtain optimal algorithms for simple problems [7] but it is not well adapted to more complex applications [12]. VR applications also require asynchronism even if a dependency (a path in the LogP graph) exists between two computations. For example, even if a simulation provides data at a low frequency, the rendering operations should not be tightly synchronized to it. In this case we need a greedy communication which is not available in LogP. This model also assumes that the cluster nodes are identical and does not account for heterogeneous configurations.

### *D. Athapascan*

Athapascan [6] is a C++ library designed for explicit parallel programming using threads. The parallelism is expressed in the code by explicit remote procedure calls to threads which are synchronized and communicate through a shared memory. The dependencies between threads are expressed by a graph which is built by following the sequentially structure of the code. Each node represents a different task, and each path a data dependency between two tasks. When a task is created then a node is added to the graph. If this task contains a reference to a variable shared by a previous task in the graph then it is linked to this task. As in the LogP model, the cost of an execution is defined by the length of the longest path in the dependency graph.

In the general case, the graph is built at the beginning of the execution. But for VR applications which include infinite loops the graph has an infinite size and cannot be built. In this case it is possible to limit the graph construction to a certain size. For example [8] presents a cloth simulation using the Athapascan library in which each simulation step is associated to a new graph. The tasks are then mapped on the different processors by a scheduler following a mapping policy based on heuristics.

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

### E. Performance model for the SCP language

SCP [13] is an SPMD language based on structured dependencies directed by the syntax order. An SCP program can be viewed as a directed acyclic graph built by following the sequential instructions order. Each path in the graph represents a possible execution of the program. The cost of an SCP program corresponds to the longest path in the graph depending of an initial context.

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

## III. 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 [2]. We now 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 port.
- The *put* function writes a message on an output port. Only one new message can be written per port and per iteration. This is a non-blocking call, thus allowing the overlapping of computations and communications.

Note that a module does not explicitly address any other FlowVR component. The only way to gain 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 understood as a 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, 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 useless. 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 and have a total access to incoming messages. They have the freedom to select, combine, create or discard messages. For example the *gather* and *scatter* filters respectively combine and split messages, the *broadcast* filter duplicates message and the *greedy* filter selects the last message available.

A special class of filters, called *synchronizers*, implements coupling policies. They only receive, handle and send stamps from other filters or modules to take a decision that will be executed by other filters. These detached components make possible a centralised decision to be broadcasted to several filters in order 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 to 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 makes it possible 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 performance by limiting the required number of context switches. As a consequence, the CPU load generated by the FlowVR network management can be considered as negligible compared to module load.

## IV. Performance prediction

We now present our approach to compute performance informations for a FlowVR application mapping on a cluster.

Our goal is to provide information such as the frequency of the different modules, the CPU load on the different nodes or the volume of communications on each network link. Then the developer will be able to determine if his application runs as expected or to compare several mappings to find the best one.

## A. 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 network 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 $L(net)$. This allows communication schemes using multiple heterogeneous networks. We assume cluster networks with point-to-point connections in full-duplex, and communications handled by dedicated network controller without CPU overload. We also assume that communication between objects mapped on the same processor are costless. Indeed, in this case, messages are stored in the shared memory and a pointer to the message is given to the receiving object. The FlowVR network is a graph $G$ composed of a set of vertices $V$ and a set of directed edges $E$. A vertex $v \in V$ represents a FlowVR object which can be a module, a filter or a synchronizer. An edge $e \in E$ 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 objects and connections of the FlowVR network 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 $n \in Nodes$. The developer has to map modules on nodes. The mapping of modules on processors is done 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 : $Net(e) = local$. 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 the execution time the module $m$ really uses the CPU. The rest of the execution time 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 $e$. For example, if we consider a module computing a physical simulation we know that the size of its output depends on the size of the simulation domain. If $src(e)$ is a filter then $Vol(e)$ depends on the filter characteristics. For example a *greedy* or a *broadcast* filter sends the same amount of data it receives. Another example is the *merge* filter which sends only one message built from all messages it receives. 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 for the sake of simplicity that filters and synchronizers have a negligible load compared to module loads. Indeed they only perform memory operations on messages.

The value of $Vol(e)$ is independent on the hardware and is statically determined from the module characteristics. Values of $T_{exec}(m)$ and $LD(m)$ can be determined by different ways. For example the developer can measure them by running each module separately on the target node. On the other hand, since FlowVR allows to reuse modules from other applications, $T_{exec}(m)$ and $LD(m)$ may be already available.

## B. Determining performance

Performance of the modules depend on synchronization and concurrency between them. Consequently we need to determine for each module $m$ its iteration time $T_{it}(m)$ and its concurrent execution time $T_{cexec}(m)$. We define the iteration time $T_{it}(m)$ 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 :

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

We define the concurrent execution time $T_{cexec}(m)$ as the time needed for the execution of one iteration of $m$ when several modules are running on the same node. Indeed, the execution of concurrent modules are interleaved by the OS scheduler and we have consequently $T_{cexec}(m) \geq T_{exec}(m)$. The concurrent execution time is determined according to a scheduler policy, but this policy strongly depends on the time a module $m$ waits for I/O operations and is blocked in the FlowVR *wait* function. If $m$ has no concurrent modules then we have :

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

We first study the effects of synchronization on performances. Then we will evaluate how the concurrency between modules affects their performances.

*1) 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 as the set of modules $IM(m)$ 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 $m$ synchronously through FIFO connections and asynchronously through *greedy* filters.

We first consider the influence 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 input modules in $IM_a(m)$. Consequently the module $m$ is like a module with only FIFO connections. Moreover if $IM_s(m) = \emptyset$ then its iteration time only depends on concurrency with other modules :

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

Thus to study the effect of synchronization on performance it is possible to remove *greedy* filters from the graph $G$. We obtain a new graph called $G_{sync}$. We note that $G_{sync}$ may not be connected anymore and may be split into several components. Each component consists of FlowVR objects connected synchronously with FIFO connections. 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 $IM_s(m)$. Thus $m$ should wait for the slowest module in $IM_s(m)$ and this module determines the iteration time of $m$ :

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

We note that if $m$ is slower than its input modules then $T_{it}(m) = T_{cexec}(m)$. 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 predecessor modules in a component $C$. Their iteration time is given by equation 3 because they are not synchronized. It is also possible to have $preds(C) = \emptyset$. Indeed modules in $C$ can be organized in a synchronous cycle. If there is no predecessor in $C$ then it means that we have a predecessor cycle $G_{pc}$, which is a synchronous cycle such as for each module $m$ in $G_{pc}$, $IM_s(m) \in G_{pc}$. Note that we may have both predecessor modules and predecessor cycles in $G_{pc}$. Modules in a synchronous cycle have the same iteration time. 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 concurrent execution time of other modules in the cycle. We should also consider the time needed for communications between modules in $G_{pc}$. For each module $m_c \in G_{pc}$ we have :

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

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

*2) Determining $T_{cexec}$ for concurrent modules:* We turn to study consequences on concurrency on modules performances to compute their concurrent computation time.

The behaviour of concurrent modules on a node $n$ is determined by the scheduler of the operating system. 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 for each module $m$ its $T_{cexec}(m)$ we first need to determine the waiting time. This time depends on the time a module $m$ waits for I/O operations and stays in the FlowVR *wait* function.

Predecessor modules are not synchronized. They only wait for I/O operations according to their execution times and their loads. For each predecessor module $pm$, we define $T_{I/O}(pm)$ as follow :

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

If a module $m$ is synchronized with its input modules then we can also define the time $m$ waits as the time not used for the computation during an iteration. Consequently we define $T_{I/O}(m)$ as follow :

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

With $T_{I/O}(m)$ we can sort modules on each node $n$ in a list $l(n)$ from the module with the highest value of $T_{I/O}(m)$ to the one with the lowest value. Then we allocate a CPU in $CPUs(n)$ and a load on this CPU to each module $m$ in $l(n)$. We consider modules in the list order. Each module is set to the most available CPU, that is the CPU with the lowest load. We define $CPULD(cpu)$ as the load of a CPU. It is equal to the sum of the module loads on the CPU. A module then receives a concurrent load $LD_c(m)$ on this CPU according to its load $LD(m)$. Then we use the ratio between $LD(m)$ and $LD_c(m)$ to evaluate $T_{cexec}(m)$. This process is implemented by the following algorithm :

```
for all $cpu \in CPUs(n)$ do
    $CPULD(cpu) = 0$
end for
while $l(n) \neq \emptyset$ do
    $m = head(l(n))$
    $l(n) = tail(l(n))$
    $load = 1$
    for all $cpu \in CPUs(n)$ do
        if $CPULD(cpu) < load$ then
            $p = cpu$
            $load = CPULD(cpu)$
        end if
    end for
    $LD_c(m) = (1 - CPULD(p))) \times LD(m)$
    $CPULD(p) = CPULD(p) + LD_c(m)$
    $T_{cexec}(m) = T_{exec}(m) \times \frac{LD(m)}{LD_c(m)}$
end while
```

In this approach $T_{I/O}(m)$, and consequently $T_{cexec}(m)$, 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 module, $IM_s(i)) = \emptyset$, then $T_{it}(i)$ depends on $T_{cexec}(i)$ from equation 3. Then if $m$ and $i$ are mapped on the same node we can not compute $T_{cexec}(i)$ if we have not yet determined $T_{I/O}(m)$ and $T_{I/O}(i)$. 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 synchronizations (directed edges) or concurrency between modules (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 twice. For example two modules connected with a bidirected edge constitute a cycle.

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. For each module $m$ in a component $C_{dep} \subseteq G_{dep}$ we define $IM_{dep}(m)$ as the set of modules with directed or bidirected edges connected to $m$.

A component $C_{dep}$ can contain cycles of different nature and Directed Acyclic Graphs. We propose to remove cycles in $C_{dep}$. We obtain a set $D_{dep}$ of DAGs. We then study cycles and DAGs in $D_{dep}$ 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 $T_{cexec}(m) = T_{exec}$ for each module $m \in D_{dep}$. If $d$ contains a predecessor module $pm$ 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 to compute $T_{it}(m)$. We now consider cycles in $C_{dep}$.

Different kinds of cycles may be present in $C_{dep}$. We first consider a cycle $C_{cycle} \subset C_{dep}$ with only bidirected edges. In this case all modules in $C_{cycle}$ are mapped on the same node and are from different components. If $C_{cycle}$ contains only predecessor modules then we can determine $T_{I/O}(pm)$ for each $pm \in C_{cycle}$ according to equation 6. Otherwise if we have at least one module which is not a predecessor then we use equation 7. But we need to first study the 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 are able to determine $T_{it}(m)$ according to equations 2 and 5. Otherwise for each module $m$ in $C_{cycle}$ with input modules not in $C_{cycle}$ we first need to study parts of the graph which contain modules in $IM_s(m)$. Then we can apply equation 4 to modules in $C_{cycle}$.

We finally consider cycles with both directed and bidirected edges. If we detect a cycle $C_{cycle}$ of this kind then 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 can consider that modules in the same synchronous component $C$ have the same iteration time. Indeed if there is a module $m \in C$ such as $T_{cexec}(m) > T_{it}(i), i \in IM_s(m)$ then $m$ is slower than $i$. Consequently messages from $i$ are accumulated in the FlowVR shared memory until it is full. In this case we have a buffer overflow. Thus our hypothesis seems appropriate and desirable for the developer. But this single iteration time is not determined. We are nonetheless able to compare concurrent modules in the same component $C$. Indeed if we consider two modules $m_1, m_2$ 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 our hypothesis. Then if $m_1$ and $m_2$ are not predecessors of $C$ we have from equation 7 :

$$\begin{aligned} T_{I/O}(m_1) - T_{I/O}(m_2) = \\ T_{exec}(m_2) \times LD(m_2) - T_{exec}(m_1) \times LD(m_1)) \end{aligned} \tag{8}$$

Consequently it comes to compare the time each module effectively uses the CPU. For each module $m$ we already have its $T_{exec}(m)$ and its $LD(m)$. But if we have in $C$ a predecessor module $pm$, or a module $m$ from a different synchronous 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 $n$. Then according to our hypothesis we are able to compare them and to sort them. Thus, this solves the interdependency.

On the other hand if we have a predecessor module $pm$, 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 modifications in the order between modules. This behaviour does not correspond to the one we expect for performances and especially for interactive applications which performance has to be stable. Moreover this dynamic variation of performances due to the scheduling can be very difficult for the programmer 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 the $C_{dep}$ from these different parts. We first consider parts which are not dependent of others.

Indeed the graph contains such parts because if this is not the case then we have a cycle between parts and consequently we can consider them as a single part. 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". Thus we can compute $T_{cexec}(m)$ and $T_{it}(m)$ for each module $m$ in these parts. We repeat the process for the other parts until we have completely built the graph.

Once we have determined $T_{cexec}(m)$ and $T_{it}(m)$ for each module $m \in G_{sync}$ we verify that $T_{cexec}(m) \leq T_{it}(i), i \in IM_s m$. If this is not the case then $m$ is slower than its input modules and a buffer overflow will occur 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 are now able to determine module performances 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.

We now study communications between modules to determine network performances.

*3) Networking:* We now consider the communications defined by an application mapping between the different FlowVR objects.

We begin our study with a traversal of the application graph to determine the frequency $F(f)$ each filter $f$, and the volume of data on its output ports. We start our traversal with starting modules and follow the message flow in the graph. When we consider a filter $f$ then we assign it a frequency $F(f)$ according to its behaviour. For example a greedy filter $f_{greedy}$ sends a message only when the receiving module $m_{dest}$ asks it for a new data. Thus we have $F(f_{greedy}) = F(m_{dest})$. A broadcast filter $f_{broadcast}$ processes messages at the same frequency of its input module $m_{src}$. In this case we have $F_{broadcast} = F_{m_{src}}$.

Then we add additional edges to represent communication out of the FlowVR communication scheme, for example communication between several instances of a MPI module. For each iteration we add output edges and input edges respectively to and from other MPI instances.

Then we are able to compute the bandwidth $bw_s$ needed by a cluster node $n$ to send its data on a network $net$ :

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

This represents the total quantity of data send in a second by modules on a node $n$ through a network $net$. We can also determine the bandwidth $BW_r$ needed by a cluster node $n$ to receive its data :

$$bw_r(n, net) = \sum_{\substack{\forall e \in E, \\ Net(e)=net, \\ node(dest(e))=n}} Vol(e) \times F(src(e)) \qquad (10)$$

If for a node $n$ we have $bw_s(n, net) > BW(net)$ then messages are accumulated in the shared memory because the daemon is not able to send them all. Consequently we can detect a buffer overflow on the node $n$. If $bw_r(n, net) > BW(net)$ then there is too much data sent to the same node, leading to contention. In this case a buffer overflow occurs 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 for example 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 by modules and transported through the mapping. In VR applications the latency is critical between interaction and visualization modules : the consequence of a user input should be visualized within the shortest possible delay to keep an interactive feeling.

We determine the latency between two modules $m_1$ and $m_2$ from the path $P$ between them. 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 the iteration time $T_it(m)$ of each module $m$ and the network latency $L(Net(e))$ for each edge $e$ between two distinct nodes :

$$\begin{aligned} L(P) = & \sum_{m \in P} T_{it}(m) \\ & + \sum_{\substack{\forall e \in P \\ Net(e) \neq local}} \frac{V(e)}{BW(Net(e))} + L(Net(e)) \end{aligned} \qquad (11)$$

With this information the developer is able to detect whether the latency of a path corresponds to its requirements. For example he can verify that the latency between modules 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 communications latencies. It is also possible to create more instances of parallel modules to decrease their iteration times or to use a network with a higher bandwidth and a lower latency.

## V. 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 is composed of nodes with dual Pentium4 Xeon processors also linked with a Myrinet network. The second one is composed of nodes with two Opteron processors, each one with two cores, also networked with a second gigabit Ethernet network.

## A. Test application

We first verify each aspect of our model with simple FlowVR applications. These applications are based on a generic FlowVR module which can simulate different kinds of modules. We first determine for each module $m$ in an application $T_{exec}(m)$ by running independently each module on the destination host. Then we run the application on the cluster. Finally we compare predictions to results.

*1) Synchronizations:* We first consider a greedy synchronization between two modules $m_1, m_2$ mapped on different nodes.

| **Module** | **Nodes** | $LD$ | $T_{exec}$ | **Prediction** $T_{it}$ | **Measure** $T_{it}$ |
|---|---|---|---|---|---|
| $m_1$ | 1 | 1 | 37 | 37 | 37 |
| $m_2$ | 2 | 0.5 | 18 | 18 | 18 |

TABLE I
(TIMES ARE GIVEN IN MS)

Results are shown in table . This test confirms that greedy connections do not affect module performances.

We now replace the greedy connection between $m_1$ and $m_2$ by a FIFO connection.

| **Module** | **Nodes** | $LD$ | $T_{exec}$ | **Prediction** $T_{it}$ | **Measure** $T_{it}$ |
|---|---|---|---|---|---|
| $m_1$ | 1 | 1 | 37 | 37 | 37 |
| $m_2$ | 2 | 0.5 | 18 | 37 | 37 |

TABLE II
(TIMES ARE GIVEN IN MS)

We now invert the FIFO connection between $m_1$ and $m_2$. In this case we predict a buffer overflow because we have $T_{it}(m_1) = T_{exec}(m_1)$ and consequently $T_{exec}(m_2) > T_{it}(m_1)$. This test confirms that a buffer overflow occurs when $T_{exec}(m) > T_{it}(i), i \in IM_s(m)$. Indeed the shared memory is full and and the application exists with an error.

We turn to consider three modules organized in a synchronous cycle. Since each module in the cycle waits for the others we can not have two modules running at the same time. Thus we predict that the execution time does not change when modules are mapped on the same node. On the other hand we should see a variation of the iteration time when modules are not mapped on the same node since we have to take into account communications in equation 5.

We first map modules on different nodes. Each node sends a message of 5MB per iteration through a gigabit network with a bandwidth of 100MB/s. Thus we expect each communication will take around 50ms. We assume that the network latency is negligible compared to this communication time. We have three communications in the cycle so we add 150ms to the execution times in equation 5. Predictions and results are shown in table III.

Results in table III are really closed to our predictions even with a simple estimation of the network parameters.

| **Module** | **Nodes** | $LD$ | $T_{exec}$ | **Prediction** $T_{it}$ | **Measure** $T_{it}$ |
|---|---|---|---|---|---|
| $m_1$ | 1 | 1 | 37 | 234 | 240 |
| $m_2$ | 2 | 0.5 | 26 | 234 | 240 |
| $m_3$ | 3 | 0.5 | 21 | 234 | 240 |

TABLE III
(TIMES ARE GIVEN IN MS)

Then we map modules on the same node. In this case we only sum the execution times to obtain the iteration time of modules in the cycle from equation 5. The table IV shows our predictions and results.

| **Module** | **Nodes** | $LD$ | $T_{exec}$ | **Prediction** $T_{it}$ | **Measure** $T_{it}$ |
|---|---|---|---|---|---|
| $m_1$ | 1 | 1 | 37 | 84 | 84 |
| $m_2$ | 1 | 0.5 | 26 | 84 | 84 |
| $m_3$ | 1 | 0.5 | 21 | 84 | 84 |

TABLE IV
(TIMES ARE GIVEN IN MS)

Results show that the iteration time is equal to the sum of the execution times as predicted by our approach. Moreover we note that communication through the shared memory does not add extra latency. Thus our hypothesis is verified.

We turn to consider concurrency between modules.

*2) Concurrency:* To study concurrency we map several modules on the same SMP node with two processors. Then we apply our approach to determine for each concurrent module $m$ its $T_{cexec}$.

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 policy. We first run independently each module on the host to determine its load and its execution time. Then we determine their concurrent execution time according to our approach.

| **Module** | **Node** | $LD$ | $T_{ex}$ | **Prediction** $T_{cexec}$ | **Prediction** $LD_c$ | **Measure** $T_{cexec}$ | **Measure** $LD_c$ |
|---|---|---|---|---|---|---|---|
| $m_1$ | 1 | 1.00 | 20 | 38 | 0.42 | 36 | 0.55 |
| $m_2$ | 1 | 0.30 | 16 | 32 | 0.15 | 26 | 0.20 |
| $m_3$ | 1 | 0.50 | 10 | 10 | 0.50 | 16 | 0.44 |
| $m_4$ | 1 | 0.58 | 51 | 51 | 0.58 | 60 | 0.52 |

TABLE V
(TIMES ARE GIVEN IN MS)

Results in table V are close to our predictions. Nonetheless we note that the scheduler gives the maximum priority to modules $m_1$ and $m_2$ but does not give them the necessary load.

*3) Communications:* We turn to study communications between modules.

We verify that if a module is slower than its input module then it generates a buffer overflow. Therefore we use two modules $m_1$ and $m_2$ with a FIFO connection between $m_1$ and $m_2$. We set $T_{it}(m_2)$ such as $T_{it}(m_2) > T_{it}(m_1)$. Tests show that the buffer overflow error occurs few iterations after we launch the application.

A complete and more complex example of performance prediction and optimization using multiple networks can be found in our previous paper [11].

## *B. The FluidParticle application*

We now apply our approach on a real FlowVR application. The FluidParticle application consists of a flow simulation which produces a velocity field. This field is then used to advect particles. Particles are displayed on a display wall of four projectors using a point sprite representation. This application is used to observe typical fluid phenomena like vortices. Our goal is to provide the highest performance for the simulation and to provide an interactive visualization. In this section we focus our study on effects of synchronization and concurrency between modules since a complete example of performance analysis focused on networks can be found in [11].

The application is composed of the following modules :

- *fluid* : this is a parallel version of the Stam's fluid simulation [15] based on the MPI library [9].
- *particles* : this is a parallel module. Each instance stores a set of particles and moves them according to forces provided by the fluid simulation. The particle set is then sent on the output port.
- *viewer* : it converts the particles positions received on its input port into graphical data which are then sent through its scene output port.
- *renderer* : it displays informations provided by the viewer modules. There is one renderer per screen. We want to visualize the particles on a display wall of four projectors. Renderer modules are synchronized together with a swaplock to provide a coherent result on the display.
- *joypad* : it is the interaction module which allow the user to interact with the fluid by adding forces.

We determine for each module its execution time and its load. Results are shown in table VI. We note that the *joypad* module has load under 1%. It is an interaction module and it is always connected to other modules through *greedy* filters. Indeed this allows to interact asynchronously with the simulation. Consequently it can not involve penalties and we choose to ignore it.

We now describe the communication and synchronization scheme between modules. The *fluid* module is connected synchronously with the *particles* module. Then the *particles* module is also connected with the *viewer* module synchronously. 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.

The graph can be splitted into two synchronous components if we remove greedy connections between the *viewer* and the *renderer* modules. The first component contains the *fluid*, the *particles* and the *renderer* modules while the second one only contains the *renderer* modules. We note that we have no synchronous cycle in our application.

We turn to study synchronization and concurrency between modules for three possible mappings. We first propose a mapping with the simulation, the particle system and the renderer modules on the same dual processor nodes. In this case we detect an interdependency between the three modules. Indeed we have a cycle which contains a directed edge between the *fluid* and the *particles*, and a bidirected edge between *renderer* and *fluid*. 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 module. 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 may 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 processor dynamically. Indeed our tests show that their concurrent execution time vary. We now propose a different mapping.

If we map the *particles*, the *viewer* and the *renderer* modules on the same node with two processors then we also detect a cycle with both directed and bidirected edges. Thus we also have an interdependency between modules and we can not sort them. Indeed we have two modules from the same synchronous component and a predecessor module from a different component. In this case we suppose that the iteration time is the same for each module in the component. Then we can compare the *particles* and the *viewer* modules from equation 8. We obtain that the *particles* module has a higher priority than *viewer* module. But we could not compare them with the *renderer* module. To solve the interdependency we propose to set $T_{cexec}(m) = T_{exec}(m)$ for each module $m$ to try to set an order between these modules. In this case the module with the highest priority is *particles*, then *viewer* and then *renderer*. Consequently the first two modules are mapped on different processors, then the *renderer* module is mapped with the *particles*. If we apply this process one more time we obtain the same order and the same iteration times and concurrent execution times for each module. Consequently we converge. Thus we can predict that the *renderer* module may be mapped with the *particles* module on the same processor and that it will be allocated a load of 78%. Our tests show that the frame rate of the *renderer* module is around 23% slower which drops from 18 to around 13 frames per second. Consequently this mapping does not offer an interactive visualization.

To obtain an interactive visualization we should map the *renderer* module on a dedicated node. We also need to avoid concurrency with the *fluid* module to obtain the fastest simulation. Thus we propose to map modules as described in table VI. In this mapping we use node 11 to 18 for the

| **Module** | **Node** |  |  | **Prediction** |  |  | **Measure** |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  |  | $LD$ | $T_{exec}$ | $T_{cexec}$ | $T_{it}$ | $LD_c$ | $T_{cexec}$ | $T_{it}$ | $LD_c$ |
| $fluid$ | 11, 12, ..., 18 | 0.97 | 70 | 70 | 70 | 0.97 | 73 | 73 | 0.97 |
| $particles$ | 5, 6, 7, 8 | 0.97 | 20 | 20 | 70 | 0.28 | 21 | 73 | 0.30 |
| $viewer$ | 5, 6, 7, 8 | 0.97 | 28 | 28 | 70 | 0.40 | 28 | 73 | 0.38 |
| $renderer$ | 1, 2, 3, 4 | 0.97 | 57 | 57 | 57 | 0.97 | 60 | 60 | 0.97 |
| $joypad$ | 1 | $<$ 0.01 | $<1$ | 0 | 0 | 0 | 0 | 0 | 0 |

TABLE VI
(TIMES ARE GIVEN IN MS)

simulation and we distribute four modules on each one 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 each one on these four 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 module mapped with them. Consequently we are able to determine their iteration time. Results of this mapping are shown in table VI. We note that it confirms the performance predicted by our approach. We also note that we avoid concurrency between the *particles* and the *viewer* module. Indeed 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, since the *fluid* and the *renderer* modules have no concurrent modules we have for each module $m$ in the application $T_{cexec}(m) = T_{exec}(m)$. Thus we have a mapping which optimizes the execution times.

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

## VI. CONCLUSION

We have shown in this paper that our approach is able to predict performances for distributed FlowVR applications. For each module we provide its iteration time which characterizes its frequency. 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 generates more communications on the network. Nevertheless our method allow to determine consequences of such choices. Moreover we also provide a way to detect errors mappings. 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 our approach 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 performances. Indeed the default policy of the scheduler does not guarantee optimal performances for our applications. We also plan to provide automated tools based on our model to assist the developer in his mapping creation and optimization.

## 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] G. Cavalheiro, F. Galilee, and J.-L. Roch. Athapascan-1: Parallel programming with asynchronous tasks. In *Proceedings of the Yale Multithreaded Programming Workshop*, Yale, June 1998.

[7] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1993. ACM Press.

[8] J.M.Vincent F. Zara, F. Faure. Physical cloth simulation on a pc cluster. In *Eurographics Workshop on Parallel Graphics and Visualization*, 2002.

[9] R. Gaugne, S. Jubertie, and S. Robert. Distributed multigrid algorithms for interactive scientific simulations on clusters. In *ICAT*, 2003.

[10] E. Melin J. Allard, V. Gouranton and B. Raffin. Parallelizing pre-rendering computations on a Net Juggler PC cluster. In *IPTS 2002*, 2002.

[11] S. Jubertie and E. Melin. Multiple networks for heterogeneous distributed applications. In *Proceedings of PDPTA'07*, Las Vegas, 2007. *To appear*.

[12] G. Loh. A critical assessment of logp: Towards a realistic model of parallel computation. http://citeseer.ist.psu.edu/330639.html.

[13] X. Rebeuf. *Un modèle de coût symbolique pour les programmes parallèles asynchrones à dépendances structurées*. Thèse de Doctorat d'Université, Université d'Orléans, décembre 2000.

[14] José L. Roda, Casiano Rodríguez, Daniel González-Morales, and Francisco Almeida. Predicting the execution time of message passing models. *Concurrency - Practice and Experience*, 11(9):461–477, 1999.

[15] J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003.

[16] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.