Mapping Heterogeneous Distributed Applications on Clusters

Sylvain Jubertie, Emmanuel Melin, Jérémie Vautard, Arnaud Lallouet

Université d'Orléans, LIFO B.P. 6759
F-45067 ORLEANS Cedex 2
{firstname.name}@univ-orleans.fr

Abstract. Performance of distributed applications largely depends on the mapping of their components on the underlying architecture. On one side, component-based approaches provide an abstraction suitable for development, but on the other side, actual hardware becomes every day more complex and heterogeneous. Despite this increasing gap, mapping components to processors and networks is commonly done manually and is mainly a matter of expertise. Worse, the amount of efforts required for this task rarely allows to further consider optimal hardware use or sensitivity analysis of data scaling. In this paper, we rely on a formal and experimentally sound model of performance and propose a constraint programming based framework to find consistent and efficient mappings of an application onto an architecture. Experiments show that an optimal mapping for a medium-sized application can be found in a few seconds.

1 Introduction

Distributed architectures can have very different levels of parallelism. For example a cluster can be composed of an heterogeneous set of nodes where each node owns a set of (possibly multicore) processors sharing a common memory. These nodes can be linked to some others by multiple networks with different topologies and characteristics (bandwidth and latency). To hide the complexity and to foster portability, parallel programming models abstract these difficulties by providing a uniform notion of process. Then it remains to find a suitable placement, which is an assignment of each process to a processor.

Two common solutions are to use parallel libraries, like MPI[1] and PVM[2], or process migration techniques, like openMosix[3]. For some applications, these two solutions have drawbacks. The parallel libraries are not well suited for the creation of heterogeneous applications since the resulting code also contains communication and synchronization schemes. Thus the developer has to modify the application code each time he needs to optimize the application or to take advantage of a different cluster. On the other side, the load balancing provided by

[1] http://www.mpi-forum.org/
[2] http://www.csm.ornl.gov/pvm/pvm_home.html
[3] http://openmosix.sourceforge.net

process migration techniques moves both processes and data, which may result in poor performances for some applications.

The FlowVR framework[4] [1, 2] was created to address this problem and to ease the development of heterogeneous distributed applications (especially Virtual Reality ones). To build a FlowVR application, the developer first creates the components, called *modules*, which encapsulate codes, and then independently defines a communication and synchronization scheme between them. This is a convenient abstraction level for many applications since it provides a coarse granularity level for the code and an explicit representation of the data-flow.

Classically, the developer has to map modules on cluster nodes and connections on network links by taking care of hardware performance to ensure an efficient mapping. When hardware performance is far beyond the needs of the application, this task is fairly easy to perform. But for handling larger applications, or to be able to ensure a better interaction with the user, resources become quickly scarce and have to be used with care. Moreover, the number of possible mappings dramatically increases when we consider large applications and clusters. The result is that allocation made by human is strongly suboptimal and cannot obtain the best result. For the same reason, it does not seem reasonable to generate all the possible mappings and to test each one.

In this paper, we propose to automatically find a mapping from modules to processors and from communication requirements to network links using Constraint Programming (CP). The optimization capabilities of CP allow to answer questions users were asking, like:

– what is the largest data size allowed for this application on my cluster ?
– is it possible to ensure this frequency for this module on my cluster ? what is the maximum frequency ?
– is it possible to run this application on fewer processors while ensuring the same level of performance ?
– is is possible to deploy my application on cluster X ?

The search for a suitable allocation uses the performance model of FlowVR introduced in [8]. Some general constraints are defined from the FlowVR model and its performance model and are common to all FlowVR distributed applications. Some constraints are specific to the considered distributed application, like synchronization scheme. Other constraints are derived from the underlying architecture, like hardware limitations, number of processors, network bandwidths and latencies. Finally the developer can also add his own constraints. For example he may need a precise performance on a given application part. This can be useful to restrict mappings to those with expected performance. We propose an implementation in Gecode[5] [9] which takes advantage of the advanced features of the solver and the model (reified constraints, global constraints, symmetry breaking, user-defined heuristics, branch and bound). For the medium-sized application introduced in the experimental section, it yields a problem involving more than 2000 variables.

[4] http://flowvr.sourceforge.net
[5] http://www.gecode.org/

2 Performance of FlowVR applications

The FlowVR framework is an open source middleware used to build distributed applications. A FlowVR application consits of a set of objects which communicate via messages through a data-flow network. Each message is associated with lightweight data, called *stamps*, which contain information used for routing operations. A FlowVR application can be viewed as a multigraph $G(V, E)$, called the *application graph*, where each vertex in V represents a FlowVR object and each directed edge in E a point to point FIFO connection between two objects. Objects can be of three kinds: *modules*, *filters* and *synchronizers*.

Modules are endless iteration which encapsulate tasks. Each module owns a set of I/O ports. It waits until it receives one message on each of its input ports (thus providing an implicit synchronizations between connected modules), then it processes the messages, computes its task and produces new messages that are put on its output ports. Operations on messages like routing, broadcasting, merging or scattering are done by filters, while synchronization and coupling policy are performed by synchronizers. Both filters and synchronizers are placed on connections between modules. A synchronizer only receives stamps emitted by filters or modules, then takes a decision according to its coupling policy and sends new stamps to destination objects. This decision is then performed by the destination object. With the use of synchronizers, it is possible to implement a *greedy* filter, which allows its connected modules to communicate asynchronously. In this case, the destination module always uses the last available message while older messages are discarded. More details on FlowVR can be found in [1].

The performance of a FlowVR application can be described by a formal model, as introduced in [8]. This model takes several inputs :

- a FlowVR application graph $G(V, E)$
- a cluster configuration represented by a multigraph of SMP nodes connected to networks by network links. Each network link l has a maximum bandwidth $BWmax_l$
- a mapping description : the destination node of each FlowVR object and the network links used for each connection
- some information on each module m : its execution time T_m^{exec} and the load $Load_m$ it generates when m is mapped alone on its destination processor
- the amount of data $Vol_m^{o_i}$ sent by each module m through output port o_i

The goal of our model is to determine for each module its frequency F_m and the load $Load_m^c$ it generates on the destination processor. We also need to determine the required bandwidths bw_{nl} on each network link nl to detect possible network contentions. Note that we assume that synchronizers have negligible execution times and loads compared to module ones, and generate only stamps and we choose to ignore them in our study. We also assume that filters generate negligible loads on processors compared to modules since they only perform few memory operations. However the mapping of modules modifies the amount of data sent through network links. Thus we only consider the mapping of modules and filters on nodes and the mapping of connections on network links.

The performance of each module in the whole application depends on synchronization with other modules. We propose in [8] to study the application graph to determine implicit synchronizations between modules. Then, we compute the frequency F_m of modules according to their execution times, implicit synchronizations and explicit ones defined with synchronizers. Note that the processor load required for each module may vary depending on synchronization between modules, thus, we provide in [8] an algorithm to determine for each module the new value of its load, noted $Load^c_m$. At the end of each iteration, each module m sends messages on its output ports, thus we can compute the required bandwidths bw_l for each network link l from F_m and $Vol^{o_i}_m$. Our performance model allows to determine performance of modules for a given mapping but also to detect possible synchronization scheme misconfigurations or network contentions.

In order to be executed on a target architecture, a FlowVR application has to be mapped on available processors and network links. The technique we propose hereafter takes advantage of this formal performance model to find automatically sound and efficient mappings i.e. efficient homomorphisms from the application graph into the architecture multigraph. Thus, for each module m, the developer needs to provide the execution time $T^{exec}_{m,p}$ and the load $Load_{m,p}$ it generates when m is mapped alone on a processor p.

3 Modeling the problem using Constraint Programming

We first present the principle of Constraint Programming [3], then we show how to use it to solve our mapping problem.

3.1 Constraint Programming

A Constraint Satisfaction Problem (CSP) is a search problem established by giving a set of variables ranging over finite domains, and a conjunction of constraints i.e. logical relations, mentioning such variables. CSP formulations are naturally suited to model real-world problems, and countless applications exist indeed. For example, given $x \in \{1, 2, 3\}$, $y \in \{3, 4\}$, and $z \in \{4, 5, 6\}$, the following CSP :

$$x < y \ \wedge \ x + y = z \ \wedge \ z \neq 3x$$

is solved by selecting (if possible) one value for each variable in so as to satisfy the three constraints at once. For example, $x = 1$, $y = 4$, $z = 5$ is a valid solution, while $x = 2$, $y = 4$, $z = 6$ is not.

Formally, a CSP is a 3-tuple (V, D, C) where V is a set of variables $v_1 \ldots v_n$, D a set of finite domains $d_1 \ldots d_n$ for the variables of V, and C a set of constraints $c_1 \ldots c_m$ over the variables of V. Many CSP solvers exists in the literature. The most common way of searching CSP solution is to combine a backtracking algorithm with a constraint propagation algorithm. The first will perform a tree-like search over the domains of all the variables of the CSP since the last will reduce these domains by reasoning on the constraints of the problem. Constraints are generally of limited arity. However, there exist some constraints with arbitrary large arity, called *global constraints*, that usually encapsulate a specific efficient algorithm or give access to a data-structure. For example we make use of the *Element* constraint [6] that links an index and its value in an array.

3.2 Problem modeling

Modeling the mapping problem into a CSP can be split in three parts : (1) the pure *module and filter placement* problem, (2) taking care of the *FlowVR connection* between objects, and (3) the *traffic* part, where the placement of connections between objects through the available links and networks is done, constrained by the links maximum bandwidth and network paths.

Let us describe this problem by illustrating it with a very simple example : we consider an application composed of three modules, the first sending data to the second, which in turn sends data to the third. We want to run it on a little cluster composed of two dual-processor nodes linked by a single network. Note that there are 4 processors in the cluster (2 per node).

Practically, a boolean variable is implemented as an integer variable ranging from 0 to 1 (0 for False, 1 for True). This will be assumed in this section.

Modules and filters placement. Basically, the problem we want to solve is a placement of modules on processors and filters on nodes since we assume that filters generate negligible loads compared to modules. So, we begin with creating variables $Module_i$ for each module in $G(V, E)$ (i going from 0 to the number of modules the application is composed of), which value will be the number of the processor on which the i-th is to be executed. Then we create filter variables $Filter_i$ for each filter in $G(V, E)$ (i going from 0 to the number of filters the application is composed of) which value will be the number of the node on which the i-th is to be executed. In our example, our processors are indexed from 0 to 3, we hence have three variables $Module_1$, $Module_2$ and $Module_3$ ranging on this interval.

The load $Load^c_{i,j}$ put by a module $Module_i$ on a processor p_j is expressed in percents and is determined by our performance model [8]. A first restriction is that a processor cannot be loaded beyond 100%. This restriction can be expressed using arithmetic constraints : first, a matrix M of auxiliary boolean variables is created, each $M_{i,j}$ being constrained to be true if module m_i executes on processor p_j, and false otherwise, using reified equality constraint $(Module_i = j) \leftrightarrow M_{i,j}$. Then, for each processor p_j, the constraint $\sum_i M_{i,j} * Load^c_{i,j} \leq 100$ is posted. The consistency between $Module_i$ variables and the M matrix is ensured by this last contraint, plus a set of *Element* constraints enforcing, for each module i, that $M_{i,Module_i} = 1$. In our example, the matrix M is created with a size of $3 * 4$ (three modules, four processors), as well as the $Load^c_{0,1}$ to $Load^c_{3,4}$ variables, ranging from 0 to 100. Then, we post the following constraints for the consistency of the model :

- $M_{0,0} + M_{0,1} + M_{0,2} + M_{0,3} = 1$
- $M_{1,0} + M_{1,1} + M_{1,2} + M_{1,3} = 1$
- $M_{2,0} + M_{2,1} + M_{2,2} + M_{2,3} = 1$

- $M_{0,Module_0} = 1$
- $M_{1,Module_1} = 1$
- $M_{2,Module_2} = 1$

and the following ones to ensure a processor will not be overloaded :

- $M_{0,0} * Load^c_{0,1} + M_{1,0} * Load^c_{1,1} + M_{2,0} * Load^c_{2,1} \leq 100$
- $M_{0,1} * Load^c_{0,2} + M_{1,1} * Load^c_{1,2} + M_{2,1} * Load^c_{2,2} \leq 100$
- $M_{0,2} * Load^c_{0,3} + M_{1,2} * Load^c_{1,3} + M_{2,2} * Load^c_{2,3} \leq 100$
- $M_{0,3} * Load^c_{0,4} + M_{1,3} * Load^c_{1,4} + M_{2,3} * Load^c_{2,4} \leq 100$

Connection mapping. Several issues occur when modeling communications : first, a connection between two objects must travel through two network links which share the same network. Of course, these links must belong to the nodes the modules are running on. Then, the total amount of communication going through one given link l_i must not exceed its bandwidth $BWmax_i$.

Each FlowVR connection $c_i \in E$ in the graph $G(V, E)$, is represented by two variables c_i^{in} and c_i^{out}, giving respectively the index of its input and output network link. Another variable cn_i indicates the network this connection is traveling through. To solve the first issue, we first build a static array LoN (Links on Network) such that LoN_i is equal to the network to which is connected the link l_i. Then, we post two *Element* constraints to ensure $LoN[c_i^{in}] = cn_i$ and $LoN[c_i^{out}] = cn_i$.

In the example, we have two connections (numbered 0 and 1), and therefore create the c_0^{in}, c_0^{out}, c_1^{in} and c_1^{out}, ranging over the two existing links (one for each node), numbered 0 and 1. Variables cn_0 and cn_1 can also be created, however their domain is already reduced to value 1, as there is only one network available. The array LoN in this case is equal to $[1, 1]$, and we then post the *Element* constraints $LoN[c_0^{in}] = cn_0$, $LoN[c_0^{out}] = cn_0$, $LoN[c_1^{in}] = cn_1$ and $LoN[c_1^{out}] = cn_1$.

Traffic. The traffic bw_i generated by a connection c_i is equal to the product of the frequency F_j of the emitter module m_j and the volume emitted by this module at each iteration Vol_{c_i}. This traffic really travels through a link iff the connection is not local (in this case, the connection is ignored). For each connection, Vol_{c_i} is a constant integer, and F_j a variable already defined, so the variable bw_i is created and the constraint $bw_i = F_j * Vol_{c_i}$ is posted. As we will need to know if a connection is local or not, we have to define for each connection c_i a boolean variable Loc_i that will be true iff c_i is local. This is enforced simply by checking that c_i^{in} and c_i^{out} are equal i.e. posting the reified constraint $(c_i^{in} = c_i^{out}) \leftrightarrow Loc_i$

In our example, we create variables bw_0, bw_1, Loc_0 and Loc_1 and simply post the constraints $bw_0 = F_0 * Vol_0$ and $bw_1 = F_1 * Vol_1$, $(c_0^{in} = c_0^{out}) \leftrightarrow Loc_0$ and $(c_1^{in} = c_1^{out}) \leftrightarrow Loc_1$.

The link bandwidth issue is solved like the processor maximal load one : we create two matrices of boolean variables Cin and $Cout$, $Cin_{i,j}$ (resp. $Cout_{i,j}$) being constrained to be true if and only if the connection c_i input (resp. output) is made via the link l_j. Then, for each link, we sum the traffic of the non-local connections passing through it by posting the constraints $\sum_i Cin_{i,j} * Loc_i * bw_i \leq BWmax_j$ and $\sum_i Cout_{i,j} * Loc_i * bw_i \leq BWmax_j$. Note that we have two distinct sums because the links work in full-duplex mode i.e. that they can emit and receive at their full bandwidth at the same time.

In the example, variables $Cin_{0,0}$ to $Cin_{2,2}$ and $Cout_{0,0}$ to $Cout_{2,2}$ are created, integers $BWmax_0$ and $BWmax_1$ are given, so we post the following constraints :

– $Cin_{0,0} * Loc_0 * bw_0 + Cin_{1,0} * Loc_1 * bw_1 \leq BWmax_0$
– $Cin_{0,1} * Loc_0 * bw_0 + Cin_{1,1} * Loc_1 * bw_1 \leq BWmax_1$
– $Cout_{0,0} * Loc_0 * bw_0 + Cout_{1,0} * Loc_1 * bw_1 \leq BWmax_0$
– $Cout_{0,1} * Loc_0 * bw_0 + Cout_{1,1} * Loc_1 * bw_1 \leq BWmax_1$

4 Experiments

We have implemented this framework using the constraint solver Gecode[6] [9]. Given descriptions of a distributed application and of a cluster architecture, it generates the CSP translation of the mapping problem using the techniques presented in previous section, plus possibly some user-defined constraints.

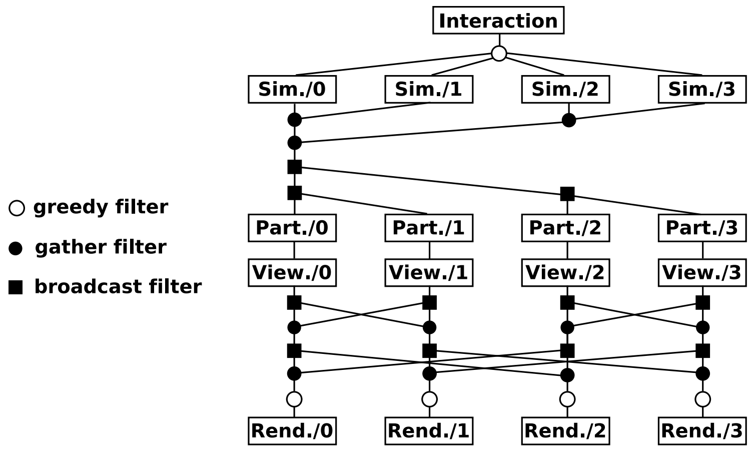

Fig. 1. FluidParticle application graph

In the following, we use the medium-sized FlowVR application *FluidParticle* to benchmark the different mappings found by the solver. We have been able to leverage the hardware power by scaling up the application to a size that was impossible before. The application is composed of the following parts :

– *simulation* : it is a parallel version [4] of the Stam's fluid simulation [10].

[6] http://www.gecode.org

– *particles* : it stores a set of particles and moves them according to a force field.
– *viewer* : it converts particle positions into graphic primitives.
– *renderer* : it displays on the screen information provided by the viewer modules.
– *interaction* : it is an interaction module, it converts user interaction into forces.

The graph (without synchroniszers) of our *FluidParticle* application is shown in figure 1. Note that we define multiple instances of some modules to decrease their execution time.

Our tests are performed on our VR platform, described in figure 2. We now show the different possibilities offered by our approach by answering questions of increasing difficulty.

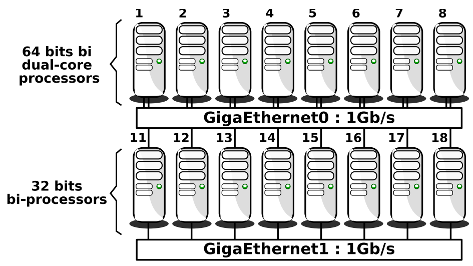

Fig. 2. Our cluster platform

4.1 Validating mappings

Since our approach integrates our performance model, we can use it to validate mappings. In this case we set the mapping of objects and connections by restricting each variable domain to a given value. Then, we run the solver which only verifies the mapping constraints.

We have validated mappings of our application [7, 8] previously obtained after a long trial and error process.

4.2 Generating mappings

The main interest of our approach is to automatically generate mappings for a given application on a given architecture. Depending on the application, some constraints can be added to the problem. For example, some modules, like the interaction or visualization ones, are hardware dependent. Consequently they must be mapped on a given node or only on a subset of the cluster and we can restrict the domain of the corresponding variables. In our example, we need to map *renderer* modules on nodes 1 to 4 which are connected to our video projectors, thus we add the following constraints :

$$renderer_0 = 1, renderer_1 = 2, renderer_2 = 3, renderer_3 = 4$$

Then we can use the solver to automatically generate pertinent mappings.

When running the solver, we can have two possible results : it produces mappings as it is exploring the search domain, or it ends without giving any mapping. We have tried out different configurations of our application and of our architecture. For example, if we only use a single network link to connect nodes to the gigaEthernet network, then the solver does not find a solution. If we use the three available networks, then the solver provides thousands of solutions. Note that the execution time of our solver may vary from seconds to hours depending on the cluster configuration.

To improve the efficiency of our solver we can add some constraints. In our example, we observe that some mappings are symmetric and provide the same performance. For example, modules $simulation_0$ and $simulation_1$ have the same performance and send the same amount of data. Thus, if we swap them we obtain the same performance. To remove symmetries, we can add lexicographic order constraints [5] on the corresponding modules variables, for example : $simulation_0 < simulation_1$. We can also add constraints to restrict the mapping of filters. Indeed, if a filter is not mapped with one of its source or destination objects, then its input and output messages are sent through a network. Thus, if we map a filter on the same node as one of its source or destination objects, then we avoid a distant communication. To restrict the mapping of filters, we use a disjunctive constraint. For example, to restrict the mapping of the *greedy* filter between *interaction* and *renderer* modules, we define :

$$\begin{aligned} greedy = interaction || greedy = simulation_0 || greedy = simulation_1 \\ || greedy = simulation_2 || greedy = simulation_3 \end{aligned}$$

4.3 Testing application and hardware limits

We can also use our solver to find the limits of our application. For example, we can increase the number of particles in our application while the solver finds a least one solution. In our default application, we define a set of 400×400 particles. If we now consider a set of 500×500 particles, our solver still gives some solutions. But when we reach 600×600 particles, then the solver does not provide solutions anymore. Tests confirm these results. Indeed, if we try to run the application with a set of 600×600 particles, then it produces a buffer overflow due to network contention. Thus we have reached the limit of our application on our architecture.

If we want to run our application with a set of 600×600 particles then we need a more powerful cluster. Thus, we modify the current description of our architecture by virtually adding nodes and networks. This way we can determine which choices are to be made to run our application. If we add more nodes then our solver does not provide solutions, but if we virtually replace the common gigaEthernet network with a Myrinet one then we obtain some solutions.

4.4 Optimization of the cluster use

Clusters are often used by several users at the same time. However, interactive applications, like the FluidParticle one, require dedicated nodes to ensure performance. We propose to use our solver to find mappings that minimize the number of nodes. Thus we run our solver with different cluster configurations. Results show that it is possible to find mappings using only six nodes with two dual-core processors on our current architecture, for example nodes 1 to 6. It is even possible to use only four nodes if we add a Myrinet network between nodes 1 to 4.

5 Conclusion and future work

The approach presented in this paper brings to developers a very useful tool to create and optimize mappings for heterogeneous distributed applications. Its implementation allows to validate mappings and enables to automatically generate mappings which respects constraints from our performance model and those defined by the developer. It is also possible to determine the cluster configuration required to run a given application. Moreover, we have shown that optimization of mappings are possible. For example it is possible to reduce the number of nodes required by distributed applications. This answers to a very important problem since clusters are often shared by several users.

References

1. J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. FlowVR: a Middleware for Large Scale Virtual Reality Applications. In *Proceedings of Euro-par 2004*, Pisa, Italy, August 2004.
2. J. Allard, C. Ménier, E. Boyer, and B. Raffin. Running large VR applications on a PC cluster: the FlowVR experience. In *Proceedings of EGVE/IPT 05*, Denmark, October 2005.
3. K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.
4. R. Gaugne, S. Jubertie, and S. Robert. Distributed multigrid algorithms for interactive scientific simulations on clusters. In *Online Proceeding of the 13th International Conference on Artificial Reality and Telexistence, ICAT*, december 2003.
5. I.P. Gent, K.E. Petrie, and J.-F. Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 10. Elsevier, 2006.
6. Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with ai and or techniques. In *AAAI*, pages 660–664, 1988.
7. S. Jubertie and E. Melin. Multiple networks for heterogeneous distributed applications. In Hamid R. Arabnia, editor, *Proceedings of PDPTA'07*, pages 415–424, Las Vegas, june 2007. CSREA Press.
8. S. Jubertie and E. Melin. Performance prediction for mappings of distributed applications on PC clusters. In *Proceedings of IFIP International Conference on Network and Parallel Computing, NPC'07*, Dalian, China, september 2007.
9. Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer-Verlag, 2006.
10. J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003.