Modelization for the Deployment of a Hierarchical Middleware on a Heterogeneous Platform

Eddy Caron, Benjamin Depardon, Frédéric Desprez

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

Modelization for the Deployment of a Hierarchical Middleware on a Heterogeneous Platform

Eddy Caron ,
Benjamin Depardon ,
Frédéric Desprez

University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon - INRIA - UCBL 5668. France.

June 2010

Research Report N° RRLIP2010-19

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`




Modelization for the Deployment of a Hierarchical Middleware on a Heterogeneous Platform

Eddy Caron , Benjamin Depardon , Frédéric Desprez

University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon - INRIA - UCBL 5668. France.

June 2010

Abstract

Accessing the power of distributed resources can nowadays easily be done using a *middleware* based on a client/server approach. Several architectures exist for those middlewares. The most scalable ones rely on a hierarchical design. Determining the best shape for the hierarchy, the one giving the best throughput of services, is not an easy task.

We first propose a computation and communication model for such hierarchical middleware. Our model takes into account the deployment of several services in the hierarchy. Then, based on this model, we propose algorithms for automatically constructing a hierarchy on two kind of heterogeneous platforms: communication homogeneous/computation heterogeneous platforms, and fully heterogeneous platforms. The proposed algorithm aim at offering the users the best obtained to requested throughput ratio, while providing fairness on this ratio for the different kind of services, and using as few resources as possible. For each kind of platforms, we compare our model with experimental results on a real middleware called DIET.

Keywords: Hierarchical middleware, Deployment, Modelization, Grid.

Résumé

De nos jours, l'accès à des ressources distribuées peut être réalisé aisément en utilisant un *intergiciel* se basant sur une approche client/serveur. Différentes architectures existent pour de tels intergiciels. Ceux passant le mieux à l'échelle utilisent une hiérarchie d'agents. Déterminer quelle est la meilleure hiérarchie, c'est à dire celle qui fournira le meilleur débit au niveau des services, n'est pas une tâche aisée.

Nous proposons tout d'abord un modèle de calcul et de communication pour de tels intergiciels hiérarchiques. Notre modèle prend en compte le déploiement de plusieurs services au sein de la hiérarchie. Puis, en nous basant sur le modèle, nous proposons des algorithmes pour construire automatiquement la hiérarchie sur différents types de plates-formes : des plates-formes avec des communications homogènes et des puissances de calcul hétérogènes, ou des plates-formes complètement hétérogènes. Les algorithmes visent à offrir aux utilisateurs le meilleur ratio entre le débit demandé, et le débit fourni, tout en utilisant le moins de ressources possible. Pour chaque type de plate-forme, nous comparons notre modèle à des résultats expérimentaux obtenus avec l'intergiciel de grille DIET.

Mots-clés: Intergiciel hiérarchique, Déploiement, Modélisation, Grille.

1 Introduction

Using distributed resources to solve large problems ranging from numerical simulations to life science is nowadays a common practice [3, 15]. Several approaches exist for porting these applications to a distributed environment; examples include classic message-passing, batch processing, web portals and GridRPC systems [18]. In this last approach, clients submit computation requests to a meta-scheduler (also called agent) that is in charge of finding suitable servers for executing the requests within the distributed resources. Scheduling is applied to balance the work among the servers. A list of available servers is sent back to the client; which is then able to send the data and the request to one of the suggested servers to solve its problem.

There exists several grid middlewares [6] to tackle the problem of finding services available on distributed resources, choosing a suitable server, then executing the requests, and managing the data. Several environments, called Network Enabled Servers (NES) environments, have been proposed. Most of them share a common characteristic which is that they are built with broadly three main components: *clients* which are applications that use the NES infrastructure, *agents* which are in charge of handling the clients' requests (scheduling them) and of finding suitable servers, and finally *computational servers* which provide computational power to solve the requests. Some of the middlewares only rely on basic hierarchies of elements, a star graph, such as Ninf-G [19] and NetSolve [2, 11, 21]. Others, in order to divide the load at the agents level, can have a more complicated hierarchy shape: WebCom-G [17] and DIET [1, 10]. In this latter case, a problem arises: what is the *best* shape for the hierarchy?

Modelization of middlewares behavior, and more specifically their needs in terms of computations and communications at the agents and servers levels can be of a great help when deploying the middleware on a computing platform. Indeed, the administrator needs to choose how many nodes must be allocated to the servers, and how many agents have to be present to support the load required by the clients. Using as many nodes as possible, may not be the best solution: firstly it may lead to using more resources than necessary; and secondly this can degrade the overall performances. The literature do not provide much papers on the modelization and evaluation of distributed middleware. In [20], Tanaka *et al.* present a performance evaluation of Ninf-G, however, no theoretical model is given. In [7, 13, 12] the authors present a model for hierarchical middlewares, and algorithms to deploy a hierarchy of schedulers on clusters and grid environments. They also compare the model with the DIET middleware. However, a severe limitation in these latter works is that only one kind of service could be deployed in the hierarchy. Such a constraint is of course not desirable, as nowadays many applications rely on workflows of services. Hence, the need to extend the previous models and algorithms to cope with hierarchies supporting several services.

In this paper, we will mainly focus on one particular hierarchical NES: DIET (**D**istributed **I**nteractive **E**ngineering **T**oolbox). The DIET component architecture is structured hierarchically as a tree to obtain an improved scalability. Such an architecture is flexible and can be adapted to diverse environments, including arbitrary heterogeneous computing platforms. DIET comprises several components. *Clients* that use DIET infrastructure to solve problems using a remote procedure call (RPC) approach. SED*s,* or server daemons, act as service providers, exporting functionalities via a standardized computational service interface; a single SED can offer any number of computational services. Finally, *agents* facilitate the service location and invocation interactions of clients and SEDs. Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single *Master Agent (MA)* (the root of the hierarchy) and several *Local Agents (LA)* (internal nodes).

Deploying applications on a distributed environment is a problem that has already been addressed. We can find in the literature a few deployment software: DeployWare [14], ADAGE [16], TUNe [4], and GODIET [8]. Their field of action ranges from single deployment to autonomic management of applications. However, none include intelligent deployment mapping algorithms. Either the mapping has to be done by the user, or the proposed algorithm is random or round-robin. Some algorithms have been proposed in [7, 13] to deploy a hierarchy of schedulers on clusters

and grid environments. However, a severe limitation in these works is that only one kind of service could be deployed in the hierarchy. Such a constraint is of course not desirable, as nowadays many applications rely on workflows of services. Hence, the need to extend the previous models and algorithms to cope with hierarchies supporting several services.

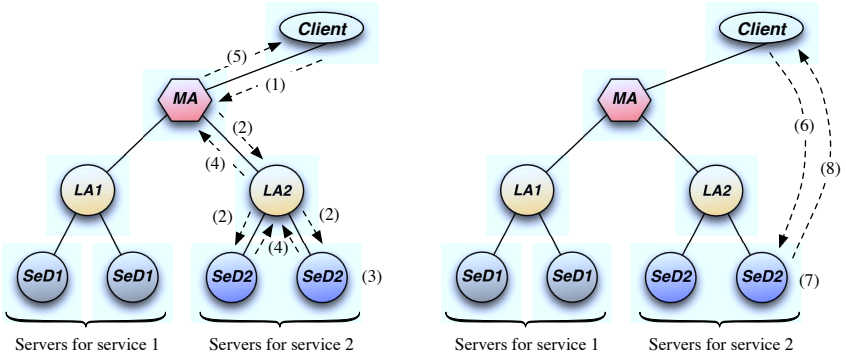
The contribution of this paper is twofold. We first present a model for predicting the performance of a hierarchical NES on a heterogeneous platforms. Secondly, we present algorithms for automatically determining the *best* shape for the hierarchy, *i.e.,* the number of servers for each services, and the shape of the hierarchy supporting these servers.

We first present in Section 2 the hypotheses for our model, then the model itself in Section 3 for both agents and servers. Then, we explain our approach to automatically build a suitable hierarchy on communication homogeneous/computation heterogeneous platforms in Section 4, and present experimental results. We then present, in Section 5 a genetic algorithm for automatically building a hierarchy on totally heterogeneous platforms, and also give some experimental results. Finally, we conclude this paper in Section 6.

2 Model Assumptions

2.1 Request Definition

Clients use a 2-phases process to interact with a deployed hierarchy. They submit a scheduling request to a Master Agent to find a suitable server in the hierarchy (scheduling phase), and then submit a service request (job) directly to the server (service phase). A completed request is one that has completed both the scheduling and service request phases and for which a response has been returned to the client. The throughput of a service is the number of completed requests per time unit the system can offer. We consider that a set $\mathcal{R}$ of services have to be available in the hierarchy. And that for each service $i \in \mathcal{R}$, the clients aim at attaining a throughput ρ_i^* of completed requests per second.



(a) Scheduling phase (b) Service phase

Figure 1: Scheduling and service phases.

Figure 1 presents the scheduling and service phases. The example shows the following steps:

1. the client sends a request of type i ($i = 2$ in Figure 1);
2. the request is forwarded down the hierarchy, but only to the sub-hierarchy that knows service i;
3. the SeDs perform performance predictions, and generate their response;
4. responses are forwarded up the hierarchy, and "sorted", *i.e.,* the best choice is selected at each agent level;

5. the scheduling response is sent back to the client (the response contains a reference to the selected server);
6. the client sends a service request directly to the selected server;
7. the server runs the application and generates the results;
8. the results are directly sent back to the client.

2.2 Resource Architecture

We consider the problem of deploying a middleware on a fully connected platform $G = (V, E, W, B)$: V is the set of nodes, E is the set of edges, $w_j \in W$ is the processing power in $Mflops/s$ of node j, and the link between nodes j and j' has a bandwidth $B_{j,j'} \in B$ in $Mbit/s$. We do not take into account contentions in the network. We also denote by $\mathcal{N} = |V|$ the number of nodes.

Remark 2.1 *We consider that whatever the node they are running on, services and agents require a constant amount of computation,* i.e., *we consider the case of* uniform machines *where the execution time of an application on a processor is equal to a constant only depending on the machine, multiplied by a constant only depending on the application.*

2.3 Deployment Assumptions

We consider that at the time of deployment we do not know the clients' locations or the characteristics of the clients' resources. Thus, clients are not considered in the deployment process and, in particular, we assume that the set of computational resources used by clients is disjoint from V, and that we will not consider client/servers direct communications in our model (phases 6 and 8 in Figure 1). This corresponds to the case where data required for the services is already present in the platform, and does not need to be sent by the clients. A valid deployment will always include at least the root-level agent and one server per service $i \in \mathcal{R}$. Each node $v \in V$ can be assigned either as a server for any kind of service $i \in \mathcal{R}$, or as an agent, or left idle. Thus with $|\mathcal{A}|$ agents, $|\mathcal{S}|$ servers, and $|V|$ total resources, $|\mathcal{A}| + |\mathcal{S}| \leq \mathcal{N}$.

2.4 Objective

We consider that we work in steady-state. The platform is loaded to its maximum. Thus, we do not take into account the phases when only a few clients are submitting requests to the system, but only the phase where the clients submit as many requests as the middleware is able to cope with.

As we have multiple services in the hierarchy, our goal cannot be to maximize the global throughput of completed requests regardless of the kind of services, this would lead to favor services requiring only small amount of computing power to schedule and to solve them, and with few communications. Hence, our goal is to obtain, for each service $i \in \mathcal{R}$, a throughput ρ_i such that all services receive almost the same obtained throughput to requested throughput ratio: $\frac{\rho_i}{\rho_i^*}$, of course we try to maximize this ratio, while having as few agents in the hierarchy as possible, so as not to use more resources than necessary.

3 Servers and Agents Models

3.1 "Global" Throughput

For each service $i \in \mathcal{R}$, we define ρ_{sched_i} to be the scheduling throughput for requests of type i offered by the platform, *i.e.,* the rate at which requests of type i are processed by the scheduling phase. We define as well ρ_{serv_i} to be the service throughput.

Lemma 3.1 *The completed request throughput ρ_i of type i of a deployment is given by the minimum of the scheduling and the service request throughput ρ_{sched_i} and ρ_{serv_i}.*

$$\rho_i = \min\{\rho_{sched_i}, \rho_{serv_i}\}$$

Proof: A completed request has, by definition, completed both the scheduling and the service request phases, whatever the kind of request $i \in \mathcal{R}$.

Case 1: $\rho_{sched_i} \geq \rho_{serv_i}$. In this case, requests are sent to the servers at least as fast as they can be processed by the servers, so the overall rate is limited by ρ_{serv_i}.

Case 2: $\rho_{sched_i} < \rho_{serv_i}$. In this case, the servers process the requests faster than they arrive. The overall throughput is thus limited by ρ_{sched_i}. □

Lemma 3.2 *The service request throughput ρ_{serv_i} for service i increases as the number of servers included in a deployment and allocated to service i increases.*

3.2 Hierarchy Elements Model

We now describe the model of each element of the hierarchy. We consider that a request of type i is sent down a branch of the hierarchy, if and only if service i is present in this branch, *i.e.,* if at least a server of type i is present in this branch of the hierarchy. Thus a server of type i will never receive a request of type $i' \neq i$. Agents won't receive a request i if no server of type i is present in its underlying hierarchy, nor will it receive any reply for such a type of request. This is the model used by DIET.

3.2.1 Server model

We define the following variables for the servers. w_{pre_i} is the amount of computation in $MFlops$ needed by a server of type i to predict its own performance when it receives a request of type i from its parent. Note that a server of type i will never have to predict its performance for a request of type $i' \neq i$ as it will never receive such a request. w_{app_i} is the amount of computation in $MFlops$ needed by a server to execute a service. m_{req_i} is the size in $Mbit$ of the messages forwarded down the agent hierarchy for a scheduling request, and m_{resp_i} the size of the messages replied by the servers and sent back up the hierarchy. Since we assume that only the best server is selected at each level of the hierarchy, the size of the reply messages does not change as they move up the tree. Figure 2 presents the server model.

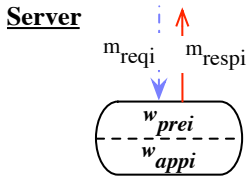


Figure 2: Server model parameters.

Server computation model We suppose that a deployment with a set of servers $\mathcal{S}_i$ completes N_i requests of type i in a given time frame. Then, each server $S_j \in \mathcal{S}_i$ will complete N_i^j such that:

$$\sum_{j \in \mathcal{S}_i} N_i^j = N_i \tag{1}$$

On average, each server S_j has to do a prediction for N_i requests, and complete N_i^j service requests during the time frame. Let $T_{comp_i}^{server_j}$ be the time taken by the server $S_j \in \mathcal{S}_i$ to compute

N_i^j requests and predict N_i requests. We have the following equation:

$$T_{comp_i}^{server_j} = \frac{w_{pre_i}.N_i + w_{app_i}.N_i^j}{w_j} \tag{2}$$

Now let's consider a time step T during which N_i requests are completed. On this time step, we have:

$$\forall i \in \mathcal{R},\ \forall j \in \mathcal{S}_i,\ T = \frac{w_{pre_i}.N_i + w_{app_i}.N_i^j}{w_j} \tag{3}$$

From (3), we can deduce the number of requests computed by each server $S_j \in \mathcal{S}_i$ for all type of requests $i \in \mathcal{R}$:

$$N_i^j = \frac{T.w_j - w_{pre_i}.N_i}{w_{app_i}} \tag{4}$$

From equations (1) and (4), we can rewrite the time taken by the $\mathcal{S}_i$ servers to process N_i requests of type i:

$$\begin{aligned}
\sum_{j\in\mathcal{S}_i} N_i^j &= \sum_{j\in\mathcal{S}_i} \frac{T.w_j - w_{pre_i}.N_i}{w_{app_i}} \\
N_i &= T \times \frac{\sum_{j\in\mathcal{S}_i} w_j}{w_{app_i}\left(1 + \sum_{j\in\mathcal{S}_i} \frac{w_{pre_i}}{w_{app_i}}\right)} \\
N_i &= T \times \frac{\sum_{j\in\mathcal{S}_i} w_j}{w_{app_i} + |\mathcal{S}_i|.w_{pre_i}} \\
T &= N_i \times \frac{w_{app_i} + |\mathcal{S}_i|.w_{pre_i}}{\sum_{j\in\mathcal{S}_i} w_j}
\end{aligned} \tag{5}$$

Hence the average computation time for one request of type i:

$$T_{comp_i}^{server} = \frac{w_{app_i} + |\mathcal{S}_i|.w_{pre_i}}{\sum_{j\in\mathcal{S}_i} w_j} \tag{6}$$

and the service throughput for service i:

$$\rho_{serv_i}^{comp} = \frac{\sum_{j\in\mathcal{S}_i} w_j}{w_{app_i} + |\mathcal{S}_i|.w_{pre_i}} \tag{7}$$

Server communication model A server of type i needs, for each request, to receive the request, and then to reply. Hence Equations (8) and (9) represent respectively the time to receive one request of type i, and the time to send the reply to its parent.

Communications time depends on the shape of the hierarchy, as the bandwidth is not the same over the platform. Thus, we denote by f_i^j the father of server $S_j \in \mathcal{S}_i$.

Server $S_j \in \mathcal{S}_i$ receive time for one request of type i:

$$T_{recv_i}^{server_j} = \frac{m_{req_i}}{B_{j,f_i^j}} \tag{8}$$

Server $S_j \in \mathcal{S}_i$ reply time for one request of type i:

$$T_{send_i}^{server_j} = \frac{m_{resp_i}}{B_{j,f_i^j}} \tag{9}$$

Service throughput Concerning the machines model, and their ability to compute and communicate, we consider the following models:

- Send or receive or compute, single port: a node cannot do anything simultaneously.

$$\rho_{serv_i} = \frac{1}{\max_{j\in\mathcal{S}_i}\left\{T_{recv_i}^{server_j} + T_{send_i}^{server_j} + T_{comp_i}^{server}\right\}}, \tag{10}$$

- Send or receive, and compute, single port: a node can simultaneously send or receive a message, and compute.

$$\rho_{serv_i} = \min \left\{ \frac{1}{\max_{j \in \mathcal{S}_i} \left\{ T_{recv_i}^{server_j} + T_{send_i}^{server_j} \right\}}, \frac{1}{T_{comp_i}^{server}} \right\} \tag{11}$$

- Send, receive, and compute, single port: a node can simultaneously send and receive a message, and compute.

$$\rho_{serv_i} = \min \left\{ \frac{1}{\max_{j \in \mathcal{S}_i} \left\{ \max \left\{ T_{recv_i}^{server_j}, T_{send_i}^{server_j} \right\} \right\}}, \frac{1}{T_{comp_i}^{server}} \right\} \tag{12}$$

3.2.2 Agent model

We define the following variables for the agents. w_{req_i} is the amount of computation in $MFlops$ needed by an agent to process an incoming request of type i. Let $\mathcal{A}$ be the set of agents. For a given agent $A_j \in \mathcal{A}$, let $Chld_i^j$ be the set of children of A_j having service i in their underlying hierarchy. Also, let δ_i^j be a Boolean variable equal to 1 if and only if A_j has at least one child which knows service i in its underlying hierarchy. $w_{resp_i}\left(\left|Chld_i^j\right|\right)$ is the amount of computation in $MFlops$ needed to merge the replies of type i from its $\left|Chld_i^j\right|$ children. We suppose that this amount grows linearly with the number of children. Figure 3 presents the server model.

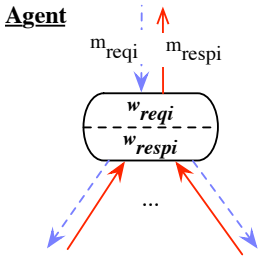


Figure 3: Agent model parameters.

Our agent model relies on the underlying servers throughput. Hence, in order to compute the computation and communication times taken by an agent A_j, we need to know both the servers throughput ρ_{serv_i} for each $i \in \mathcal{R}$, and the children of A_j.

Agent computation model. The time for an agent A_j to schedule all requests it receives and forwards,when the servers provide a throughput ρ_{serv_i} for each $i \in \mathcal{R}$, is given by Equation (13).

$$T_{comp}^{agent_j} = \frac{\sum_{i \in \mathcal{R}} \rho_{serv_i}.\delta_i^j.w_{req_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i}.w_{resp_i}\left(\left|Chld_i^j\right|\right)}{w_j} \tag{13}$$

Agent communication model. Agent A_j needs, for each request of type i, to receive the request and forward it to the relevant children, then to receive the replies and forward the aggregated result back up to its parent. We also need to take into account the variation in the bandwidth between the agent and its children. Let f^j be the father of agent A_j in the hierarchy. Equations (14) and (15) present the time to receive and send all messages when the servers provide a throughput ρ_{serv_i} for each $i \in \mathcal{R}$.

Agent $A_j \in \mathcal{A}$ receive time:

$$T_{recv}^{agent_j} = \sum_{i \in \mathcal{R}} \frac{\rho_{serv_i}.\delta_i^j.m_{req_i}}{B_{j,f^j}} + \sum_{i \in \mathcal{R}} \sum_{k \in Chld_i^j} \frac{\rho_{serv_i}.m_{resp_i}}{B_{j,k}} \tag{14}$$

Agent $A_j \in \mathcal{A}$ send time:

$$T_{send}^{agent_j} = \sum_{i \in \mathcal{R}} \frac{\rho_{serv_i}.\delta_i^j.m_{resp_i}}{B_{j,f^j}} + \sum_{i \in \mathcal{R}} \sum_{k \in Chld_i^j} \frac{\rho_{serv_i}.m_{req_i}}{B_{j,k}} \tag{15}$$

We combine (13), (14), and (15) according to the chosen communication / computation model (similarly to Equations (10), (11), and (12)).

Lemma 3.3 *The highest throughput a hierarchy of agents is able to serve is limited by the throughput an agent having only one child of each kind of service can support.*

Proof: The bottleneck of such a hierarchy is clearly its root. Whatever the shape of the hierarchy, at its top, the root will have to support *at least* one child of each type of service (all messages have to go through the root). As the time required for an agent grows linearly with the number of children (see (13), (14) and (15)), having only one child of each type of service is the configuration that induces the lowest load on an agent. □

4 Planning on Communication Homogeneous / Computation Heterogeneous Platforms

Given the models presented in the previous section, we propose a heuristic for automatic deployment planning on communication homogeneous/computation heterogeneous platforms. As we consider that we have homogeneous communications, we have $\forall j, j' \in V, B_{j,j'} = B$. The heuristic comprises two phases. The first step consists in dividing N nodes between the services, so as to support the servers. The second step consists in trying to build a hierarchy, with remaining nodes, which is able to support the throughput generated by the servers. In this section, we present our automatic planning algorithm in three parts. In Section 4.1 we present how the servers are allocated nodes, then in Section 4.2 we present a bottom-up approach to build a hierarchy of agents, and present in Section 4.3 the whole algorithm. Finally, we give experimental results in Section 4.4.

4.1 Servers repartition

Our goal is to obtain for all services $i \in \mathcal{R}$ the same ratio $\frac{\rho_{serv_i}}{\rho_i^*}$. Algorithm 1 presents a simple way of dividing the available nodes to the different services. We iteratively increase the number of assigned nodes per services, starting by giving nodes to the service with the lowest $\frac{\rho_{serv_i}}{\rho_i^*}$ ratio. We need to take into account the nodes' heterogeneity. Hence, we propose two heuristics: *min-first* which first give the less powerful nodes to the servers, and *max-first* which first give the more powerful nodes to the servers.

4.2 Agents hierarchy

Given the servers repartition, and thus, the services throughput ρ_{serv_i}, for all $i \in \mathcal{R}$, we need to build a hierarchy of agents that is able to support the throughput offered by the servers. Our approach is based on a bottom-up construction: we first distribute some nodes to the servers, then with the remaining nodes we iteratively build levels of agents. Each level of agents has to be able to support the load incurred by the underlying level. The construction stops when only one agent

Algorithm 1 Servers repartition

Require: L: list of available nodes
Ensure: L_a: list of nodes allocated to the servers
1: $S \leftarrow$ list of services in $\mathcal{R}$
2: $L_a \leftarrow \emptyset$
3: $N \leftarrow |L|$
4: $L \leftarrow L$ sorted according to *min-first* or *max-first* heuristic
5: **repeat**
6: $i \leftarrow$ first service in S
7: Assign one more node to i: n, following the order of L, and compute the new ρ_{serv_i}
8: $L_a \leftarrow L + \{n\}$
9: **if** $\rho_{serv_i} \geq \rho_i^*$ **then**
10: $\rho_{serv_i} \leftarrow \rho_i^*$
11: $S \leftarrow S - \{i\}$
12: $S \leftarrow$ Sort services by increasing $\frac{\rho_{serv_i}}{\rho_i^*}$
13: **until** $|L_a| = N$ **or** $S = \emptyset$
14: **return** L_a

is enough to support all the children of the previous level. In order to build each level, we make use of a mixed integer linear program (MILP): ($\mathcal{LP}_1$).

We first need to define a few more variables. Let k be the current level: $k = 0$ corresponds to the server level. For $i \in \mathcal{R}$ let $n_i(k)$ be the number of elements (servers or agents) obtained at step k, which know service i. For $k \geq 1$, we recursively define new sets of agents. We define by M_k the number of available resources at step k: $M_k = M_1 - \sum_{i \in \mathcal{R}} \sum_{l=2}^{k-1} n_i(l)$. For $1 \leq j \leq M_k$ we define $a_j(k) \in \{0,1\}$ to be a boolean variable stating whether or not node j is an agent in step k. $a_j(k) = 1$ if and only if node j is an agent in step k. For $1 \leq j \leq M_k, \forall i \in \mathcal{R}, \delta_i^j(k) \in \{0,1\}$ defines whether of not node j has service i in its underlying hierarchy in step k. For the servers, $k = 0$, $1 \leq j \leq M_0, \forall i \in \mathcal{R}, \delta_i^j(0) = 1$ if and only if server j is of type i, otherwise $\delta_i^j(0) = 0$. Hence, we have the following relation: $\forall i \in \mathcal{R}, n_i(k) = \sum_{j=1}^{M_k} \delta_i^j(k)$. For $1 \leq j \leq M_k, \forall i \in \mathcal{R}, \left|Chld_i^j(k)\right| \in \mathbb{N}$ is as previously the number of children of node j that know service i. Finally, for $1 \leq j \leq M_k, 1 \leq l \leq M_{k-1}$ let $c_l^j(k) \in \{0,1\}$ be a boolean variable stating that node l in step $k-1$ is a child of node j in step k. $c_l^j(k) = 1$ if and only if node l in step $k-1$ is a child of node j in step k.

Using linear program ($\mathcal{LP}_1$), we can recursively define the hierarchy of agents, starting from the bottom of the hierarchy.

Let's have a closer look at ($\mathcal{LP}_1$). Lines (1), (2) and (3) only define the variables. Line (4) states that any element in level $k-1$ has to have exactly 1 parent in level k. Line (5) counts, for each element at level k, its number of children that know service i. Line (6) states that the number of children of j of type i cannot be greater than the number of elements in level $k-1$ that know service i, and has to be 0 if $\delta_i^j(k) = 0$. The following two lines, (7) and (8), enforce the state of node j: if a node has at least a child, then it has to be an agent (line (7) enforces $a_j(k) = 1$ in this case), and conversely, if it has no children, then it has to be unused (line (8) enforces $a_j(k) = 0$ in this case). Line (9) states that at least one agent has to be present in the hierarchy. Line (10) is the transposition of the agent model in the *send or receive or compute, single port* model. Note that the other models can easily replace this model in MILP ($\mathcal{LP}_1$). This line states that the time required to deal with all requests going through an agent has to be lower than or equal to one second.

Finally, our objective function is the minimization of the number of agents: the equal share of obtained throughput to requested throughput ratio has already been cared of when allocating the nodes to the servers, hence our second objective that is the minimization of the number of agents in the hierarchy has to be taken into account.

$$
\textbf{Minimize } \sum_{j=1}^{M_k} a_j(k)
$$

$$
\textbf{Subject to}
\begin{cases}
(1) & 1 \leq j \leq M_k & a_j(k) \in \{0,1\} \\
(2) & 1 \leq j \leq M_k, \forall i \in \mathcal{R} & \delta_i^j(k) \in \{0,1\} \text{ and } \left|Chld_i^j(k)\right| \in \mathbb{N} \\
(3) & 1 \leq j \leq M_k, & \\
 & 1 \leq l \leq M_{k-1} & c_l^j(k) \in \{0,1\} \\
(4) & 1 \leq l \leq M_{k-1} & \sum_{j=1}^{M_k} c_l^j(k) = 1 \\
(5) & 1 \leq j \leq M_k, \forall i \in \mathcal{R} & \left|Chld_i^j(k)\right| = \sum_{l=1}^{M_{k-1}} c_l^j(k).\delta_i^l(k-1) \\
(6) & 1 \leq j \leq M_k, \forall i \in \mathcal{R} & \left|Chld_i^j(k)\right| \leq \delta_i^j(k).n_i(k-1) \\
(7) & 1 \leq j \leq M_k, i \in \mathcal{R} & \delta_i^j(k) \leq a_j(k) \\
(8) & 1 \leq j \leq M_k & a_j(k) \leq \sum_{i \in \mathcal{R}} \delta_i^j(k) \\
(9) & & \sum_{j=1}^{M_k} a_j(k) \geq 1 \\
(10) & 1 \leq j \leq M_k & \sum_{i \in \mathcal{R}} \rho_{serv_i} \times \\
 & & \left(\frac{\delta_i^j(k).w_{req_i} + w_{resp_i}\left(\left|Chld_i^j(k)\right|\right)}{w_j} + \right. \\
 & & \frac{\delta_i^j(k).m_{req_i} + \left|Chld_i^j(k)\right|.m_{resp_i}}{B} + \\
 & & \left. \frac{\delta_i^j(k).m_{resp_i} + \left|Chld_i^j(k)\right|.m_{req_i}}{B} \right) \leq 1
\end{cases}
\qquad (\mathcal{LP}_1)
$$

4.3 Building the whole hierarchy

We do not detail the whole algorithm for building the hierarchy, as it is the same as the one presented in [9] for homogeneous platforms. So, we refer the reader to this research report.

4.4 Experiments

We ran experiments on DIET hierarchies containing two services: `dgemm` 100 and Fibonacci 30. We used two clusters on the Lyon site of Grid'5000: Sagittaire and Capricorne. Their respective computing power is $w_{sagittaire} = 3249MFlops$ and $w_{capricorne} = 2922MFlops$. We used two strategies to sort the nodes that are divided between the services: either we sorted them by increasing w_j (heuristic *min-first*) or by decreasing w_j (heuristic *max-first*).

4.4.1 Theoretical model / experimental results comparison

We validate our model against real executions with the DIET middleware. Figures 4 and 5 present the comparison between theoretical and experimental throughput for respectively experiments with *min-first* and *max-first* heuristics. Table 1 presents the relative error for those experiments. As can be seen, the *min-first* heuristic is the most interesting one when dealing with large platforms. This can easily be explained: using less powerful nodes for servers leaves more powerful nodes for the agents, hence the maximum attainable throughput due to agents limitation is higher. Whatever the heuristic, we remain within 18.3% of the theoretical model.

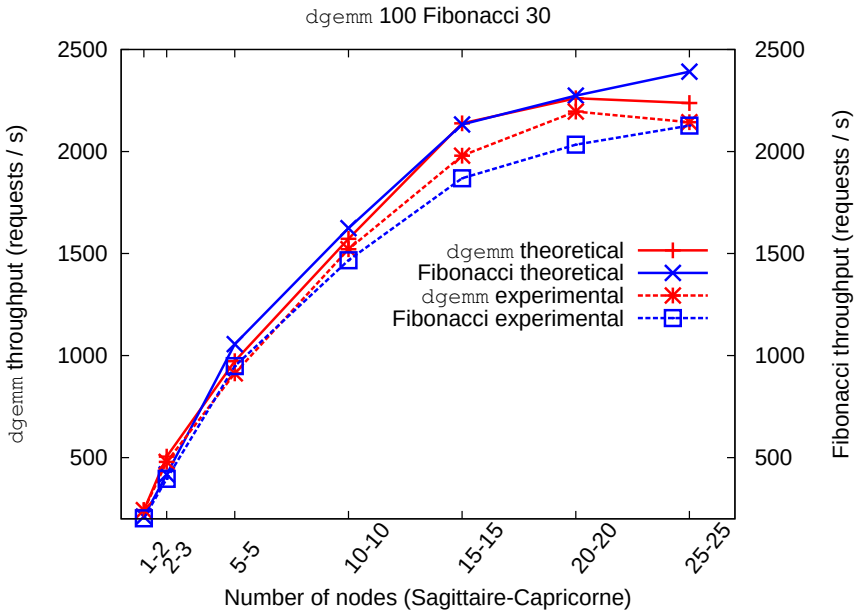


Figure 4: `dgemm` 100, Fibonacci 30 theoretical and experimental throughput, with *min-first* heuristic.

Experiment		Number of nodes						
		1-2	**2-3**	**5-5**	**10-10**	**15-15**	**20-20**	**25-25**
min-first	`dgemm`	0.3%	5.2%	6.1%	3.2%	3.9%	2.9%	4.20%
	Fibonacci	4.1%	5.1%	10.2%	9.7%	10.0%	10.6%	11.0%
max-first	`dgemm`	2.1%	0.1%	7.0%	7.7%	8.9%	2.3%	3.8%
	Fibonacci	4.6%	10.8%	12.7%	12.8%	18.3%	4.5%	3.0%

Table 1: Relative error, using *min-first* and *max-first* heuristics.

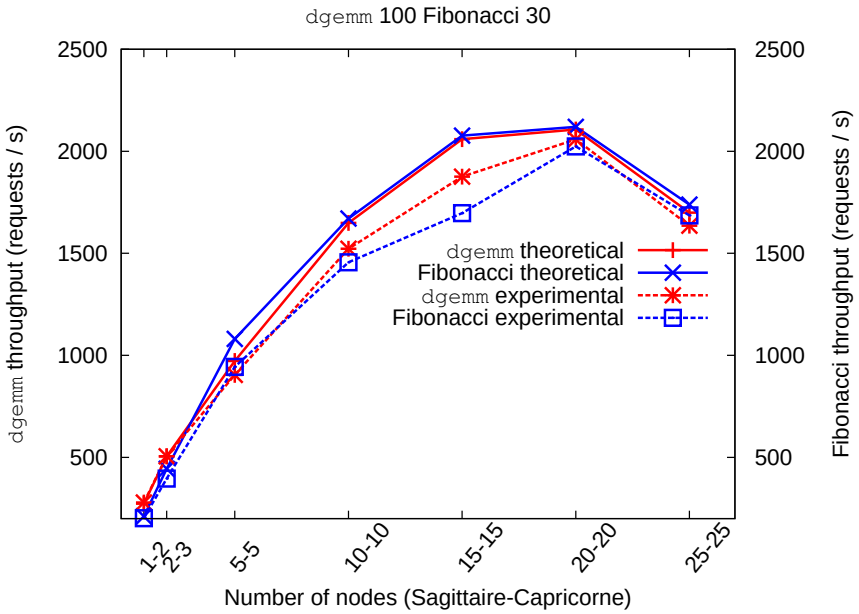


Figure 5: `dgemm` 100, Fibonacci 30 theoretical and experimental throughput, with *max-first* heuristic.

4.4.2 Comparison with star graphs

On the tested platforms, our heuristics create hierarchies having up to three levels of agents. Is this really necessary, or would have a star graph, having the same SeD mapping, given the same or better results? Figure 6 presents the comparison between *min-first* and a star graphs having the same SeD distribution, and figure 7 present the comparison between *max-first* and star graphs having the same SeD distribution. As can be seen our approach clearly surpasses the simple star graph approach.

The fact that on Figure 7 the star graph throughput increases with 25-25 nodes, is easily explained by the number of SeDs this star graph has. As can be seen on Figure 11 (25-25 nodes) less SeDs are present than in Figure 10 (20-20 nodes). Thus, the MA is less loaded on the star graph obtained on 25-25 nodes, than on the star graph obtained on 20-20 nodes.

4.4.3 Hierarchy shape

Figures 8, 9, 10 and 11 give an example of the shape of the hierarchy generated by respectively *min-first* on a 20-20 and 25-25 platform, and *max-first* on a 20-20 and 25-25 nodes platform. Red nodes are on the Sagittaire cluster, white nodes on the Capricorne cluster. "D" stands for `dgemm`, and "F" stands for Fibonacci.

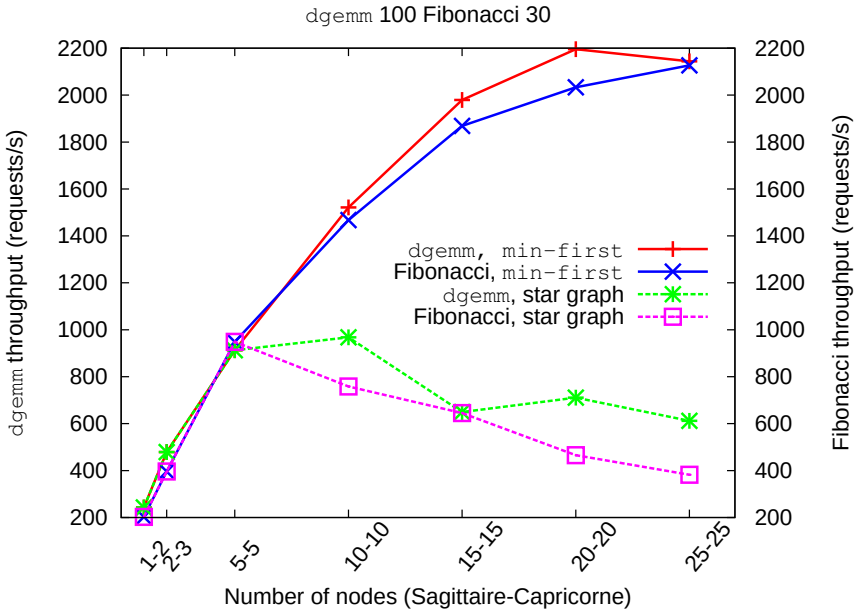


Figure 6: Comparison *min-first* with a star graph with the same SED distribution.

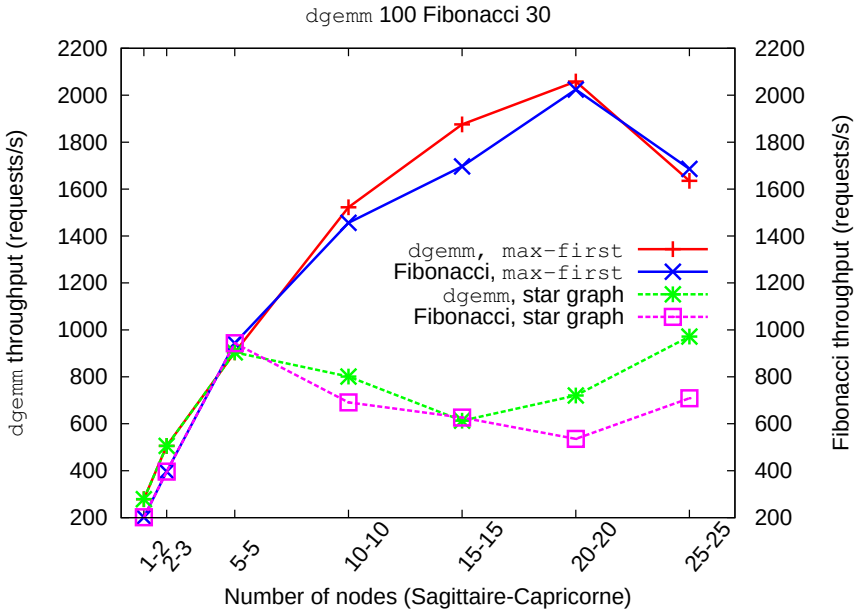


Figure 7: Comparison *max-first* with a star graph with the same SED distribution.

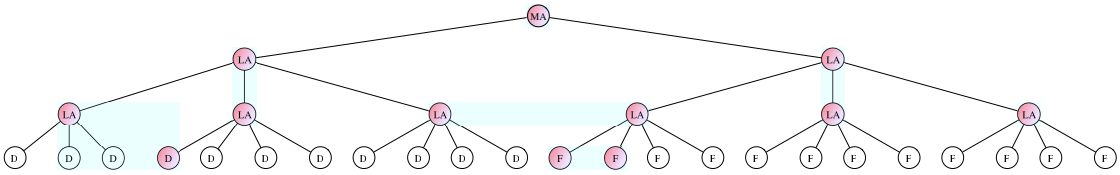


Figure 8: *min-first* : 20-20.

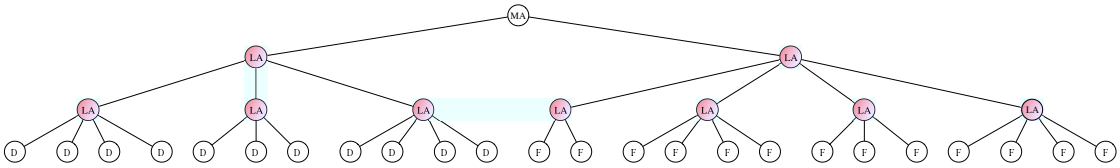


Figure 9: *min-first* : 25-25.

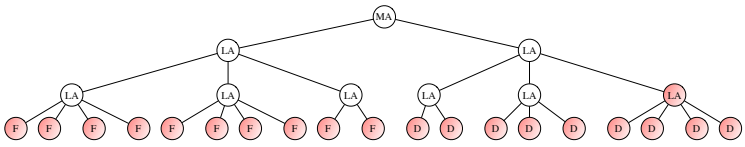


Figure 10: *max-first* : 20-20.

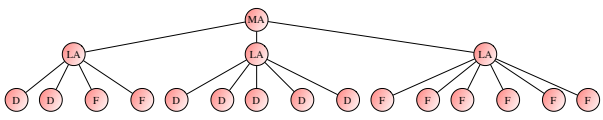


Figure 11: *max-first* : 25-25.

5 Fully Heterogeneous Platforms

In this section we finally deal with a fully heterogeneous platform: each node j has a computing power of its own w_j, and the communication links between any two nodes j, j' are possibly all different: $B_{j,j'}$. Hence, the agents and servers models follow the general model presented in Section 3.

The problem when dealing with totally heterogeneous platforms is that we need information about both the location of the parent, and location of the children. Hence, the bottom-up approach we used so far won't be applicable, nor would be a top-down approach: we won't be able to build a level if we do not know the position of the parent in a bottom-up approach, and conversely we cannot build a level without knowing the position of the children in a top-down approach.

5.1 A Genetic Algorithm Approach

As we cannot use an iterative approach to build a hierarchy without risking to have to test all possible solutions, we took a totally different approach: we rely on a *genetic algorithm* to generate a set of hierarchies, then evolve them, and finally select the best one among them.

In order to define our genetic algorithm, we need to describe a few notions: the objective function, crossover and mutations, and finally the evaluation strategy.

5.1.1 Objective function

The *objective function* has to encode all the goals we aim at optimizing in a hierarchy. It also needs to be subject to an order relation. Many genetic algorithm frameworks require that the objective function is encoded on an integer or floating point variable.

Our goal is the same as previously: we aim at maximizing the minimum ρ_i/ρ^*, while minimizing the number of agents constituting the hierarchy. Hence our goal can be summarized with the following point of decreasing importance: (i) maximize $\min_{i \in \mathcal{R}} \{\rho_i/\rho^*\}$, (ii) then maximize the second ratio, the third..., (iii) minimize the number of agents to support the maximum throughput (*i.e.,* maximize $\mathcal{N}$ minus the number of agents). In order to encode all these points into a single value, we use the following encoding (presented Figure 12): if we are given a precision ϵ on each ρ_i/ρ_i^* ratio, then we can encode them on $1/\epsilon$ digits; moreover, as the maximum number of agents is $\mathcal{N} - 1$, we only need $\lceil \log_{10}(\mathcal{N}) \rceil$ digits to encode the number of agents. Hence, on the whole we need $\mathcal{R} \times 1/\epsilon + \lceil \log_{10}(\mathcal{N}) \rceil$ digits.

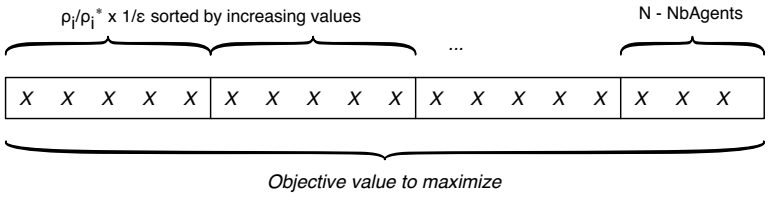


Figure 12: Objective value encoding.

Actually, in order to guide a bit more the genetic algorithm towards convergence, we added two more metrics that we wish to minimize at the end of the fitness value: the number of agents that do not have any children (in order to remove really useless elements) and the depth of the hierarchy (this shouldn't affect the throughput, but this impacts the response time of the hierarchy, and limits the formation of chains of agents).

5.1.2 Genotype

The *genotype* needs to encode the whole hierarchy: the parent/children relationship, and the type of each node (agent, server or unused). It can easily be encoded on two arrays of size $\mathcal{N}$: one for

encoding the type of the nodes, and one for encoding the parent/children relationship. The *alleles*, *i.e.,* each value in the arrays, can have the following values. The *type* of a node can either be 0 if the node is unused, 1 if it is an agent, or $i \in \{2 \dots 2 + \mathcal{R}\}$ if the node is a server of type $i - 2$. The *parent* of a node i can either be itself if the node is unused (*i.e., type*$[i] = 0$) or the MA, or the node number corresponding to the parent of i in the hierarchy. Genotypes are randomly generated when creating the first generation of individuals: nodes' types and relationship are randomly chosen in such a way that a valid hierarchy is created.

5.1.3 Crossover

We define a *crossover* between two hierarchies as follows. Crossovers are only made on the parent array. We randomly select two nodes (one on each hierarchy) and exchange the parent of both selected nodes. Figure 13 presents an example of crossover. Why not define a crossover which replaces a whole part of a hierarchy into another one? This approach works well for a small number of nodes, but has a far too big impact on the hierarchy shape on a large number of nodes. As an example, consider hierarchies H_1 and H_2 in Figure 13, suppose a crossover between node 6 in H_1 and node 1 in H_2. Transferring node 6 into H_2 in place of node 1 would remove seven nodes, as node 6 is a server. Conversely, we cannot transfer node 1 in place of node 6 into H_1, as node 1 is the root of the hierarchy, and thus a transfer would lead to a loop in the hierarchy.

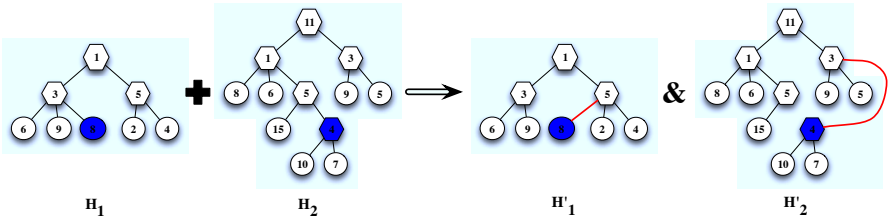


Figure 13: Crossover. Colored nodes are the one selected for crossover, within hierarchy elements represent the nodes' number.

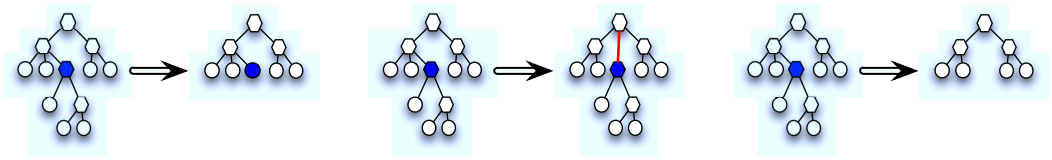
5.1.4 Mutation

Mutations on a hierarchy can occur at different levels in the hierarchy. We define the following mutations, also presented in Figure 14:

- Hierarchy modification:
  1. we randomly select a node to change its type. If the mutation changes the type from agent to unused or SeD, or from SeD to unused, then we remove the underlying hierarchy and modify the type of the node. If the type changes from unused to agent or SeD, we randomly choose a parent among the available agents.
  2. we randomly select a node that will choose a new parent among the available agents. We can end up with two hierarchies (if the new parent is the node itself), in this case we randomly select one of the two hierarchies, and delete the other.
- Pruning: a node is randomly selected, then its whole underlying hierarchy is deleted.

5.1.5 Hierarchy evaluation

In order to evaluate a hierarchy generated by our genetic algorithm, we first compute for all $i \in \mathcal{R}$ the throughput supported by the servers, we denote by $\rho_i^{\max}$ this throughput. Then, for each



(a) Type mutation (b) Parent mutation (c) Pruning

Figure 14: Mutations. Colored node has been selected for mutation. Hexagons are agents, and circles are servers.

agent in the hierarchy, we compute the maximum throughput ρ_i for each service supported by the agent, we use ($\mathcal{LP}_2$) to compute ρ_i.

$$
\begin{array}{l}
\textbf{Maximize } \dfrac{\rho_1}{\rho_1^{\max}} \\
\textbf{Subject to} \\
\left\{
\begin{array}{lll}
(1) & \forall i \in \mathcal{R} & 0 \leq \rho_i \leq \rho_i^{\max} \\
(2) & \forall i \in \mathcal{R}, i \neq 1 & \dfrac{\rho_1}{\rho_1^{\max}} = \dfrac{\rho_i}{\rho_i^{\max}} \\
(3) & & \displaystyle\sum_{i \in \mathcal{R}} \rho_i . \delta_i^j . \frac{w_{req_i}}{w_j} + \sum_{i \in \mathcal{R}} \rho_i . \left| Chld_i^j \right| . \frac{w_{resp_i}}{w_j} + \\
 & & \displaystyle\sum_{i \in \mathcal{R}} \frac{\rho_i . \delta_i^j . m_{req_i}}{B_{j,f^j}} + \sum_{i \in \mathcal{R}} \sum_{k \in Chld_i^j} \frac{\rho_i . m_{resp_i}}{B_{j,k}} + \\
 & & \displaystyle\sum_{i \in \mathcal{R}} \frac{\rho_i . \delta_i^j . m_{resp_i}}{B_{j,f^j}} + \sum_{i \in \mathcal{R}} \sum_{k \in Chld_i^j} \frac{\rho_i . m_{req_i}}{B_{j,k}} \leq 1
\end{array}
\right.
\end{array}
\qquad (\mathcal{LP}_2)
$$

5.1.6 Implementation and parameters

Genetic algorithms rely on quite a lot of different parameters. Each one of them can influence the quality of the result. Among them are the following:

- *Selection method*: when comparing x individuals, we need to choose which one should "survive." Different approaches exist: deterministic tournament, stochastic tournament, roulette or ranking. Our tests showed that the deterministic tournament was the one giving on average better results. Hence, we use this selection method in our experiments. When we force that the best parent replaces the weakest child if the child has a lower fitness, we talk about *weak elitism*. Weak elitism can possibly provide better solutions as it forces the algorithm to converge towards a *locally* good solution. However, it also reduces the population diversity, and thus, can lead to a local optimum. Our simulations tended to provide worst solutions when weak elitism was used, thus, in the following studies, we do not use it.

- *Size of the population*: with n machines, we varied the size of the population between $n/4$ and $2 \times n$. The worst results were always obtained for a size of $n/4$. The best results were obtained for populations having $5/4 \times n$ and $2 \times n$. On average, better results were obtained for populations of sizes between $5/4 \times n$ and $2 \times n$.

- *Probability of crossover and mutation*: crossover rate should generally be high, around 80%-90%, and mutation rate quite low, around 0.5%-1%. A very small mutation rate may lead to genetic drift, whereas a high one may lead to loss of good solutions. We chose set the mutation and crossover rates respectively to 1% and 80%.

We used the ParadisEO [5] framework to implement our genetic algorithm.

5.2 Quality of the Approach

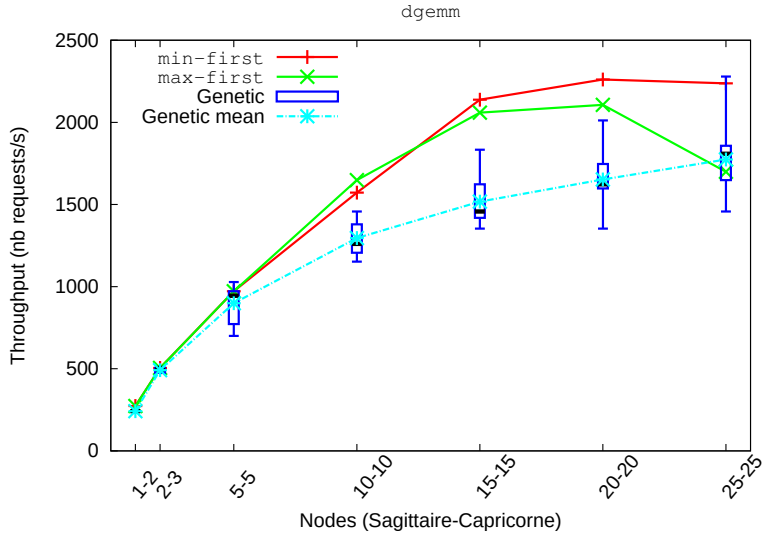
Genetic algorithms mainly depend on stochastic processes, thus we need to assess the quality of the results. As we do not have any other algorithm for fully heterogeneous platforms, we compare our genetic algorithm with the heuristics proposed in Section 4 for computation heterogeneous platforms.

5.2.1 Comparison with computation heterogeneous, communication homogeneous algorithm

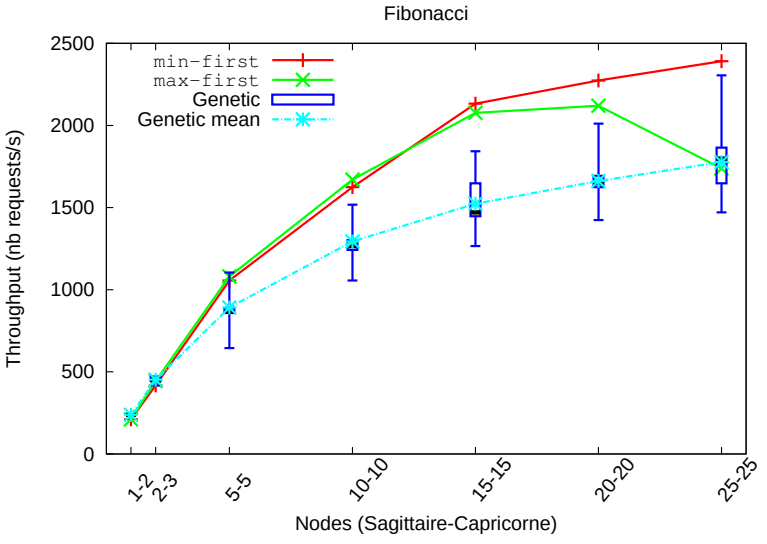
We compare our genetic algorithm (GA) with *min-first* and *max-first*. GA was run on 1000 generations, on a population having at least 50 individuals (or $2 \times |V|$ if this is higher than 50). We used the deterministic tournament method of selection, and we ran 100 executions (100 seeds). Figure 15a and 15b respectively present the results for `dgemm` and Fibonacci services. As can be seen, even if on the mean GA do not obtain results as good as *min-first* or *max-first*, the best GA results closely follows the *min-first* heuristic. Table 2 presents the relative gain/loss obtained with the best solution of the genetic algorithm, compared to the *min-first* and *max-first* heuristics. As one can see, the loss is no bigger than 15%, and it gives better results than *max-first* for the larger platform. This confirms that our approach can be effective: even if one run of GA is not sufficient to obtain the best result, taking the best hierarchy over a few runs of GA can give us good results. Note that this is often the case with genetic algorithms, as it is an exploratory method. We are quite confident that the performance loss obtained with the GA solutions can be reduced by fine tuning the GA parameters.

Experiment		**Number of nodes**						
		1-2	**2-3**	**5-5**	**10-10**	**15-15**	**20-20**	**25-25**
min-first	`dgemm`	-11.3%	0.0%	5.7%	-7.3%	-14.3%	-11.0%	1.8%
	Fibonacci	12.8%	12.6%	4.6%	-6.6%	-13.6%	-11.6%	-3.6%
max-first	`dgemm`	-11.1%	0.0%	5.7%	-11.6%	-11.0%	-4.5%	34.1%
	Fibonacci	12.8%	5.9%	2.3%	-9.2%	-11.3%	-5.1%	32.6%

Table 2: Genetic algorithm gains/loss compared to the *min-first* and *max-first* heuristics. A positive number denotes a gain, whereas a negative one denotes a loss.



(a) `dgemm` 100



(b) Fibonacci 30

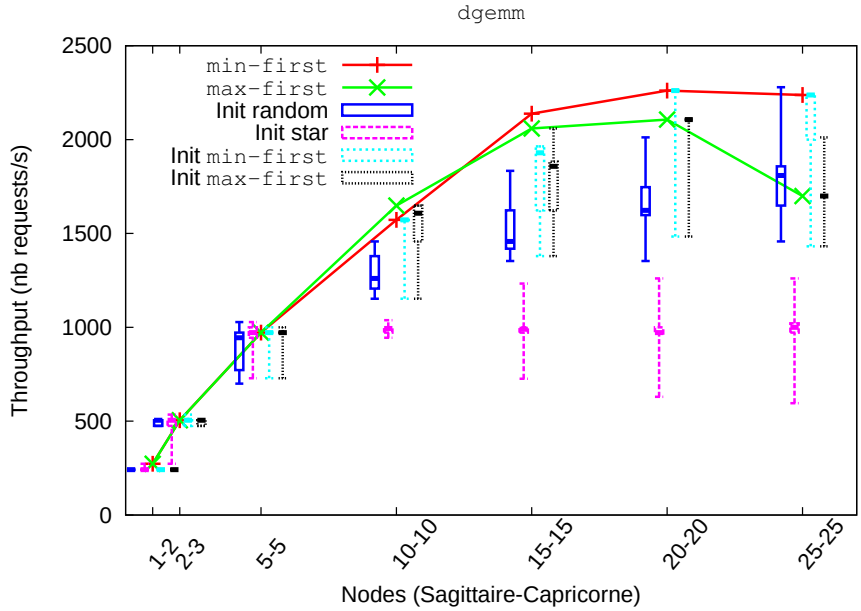
Figure 15: Comparison genetic algorithm results with *min-first* and *max-first* results.

5.2.2 Initialization difference

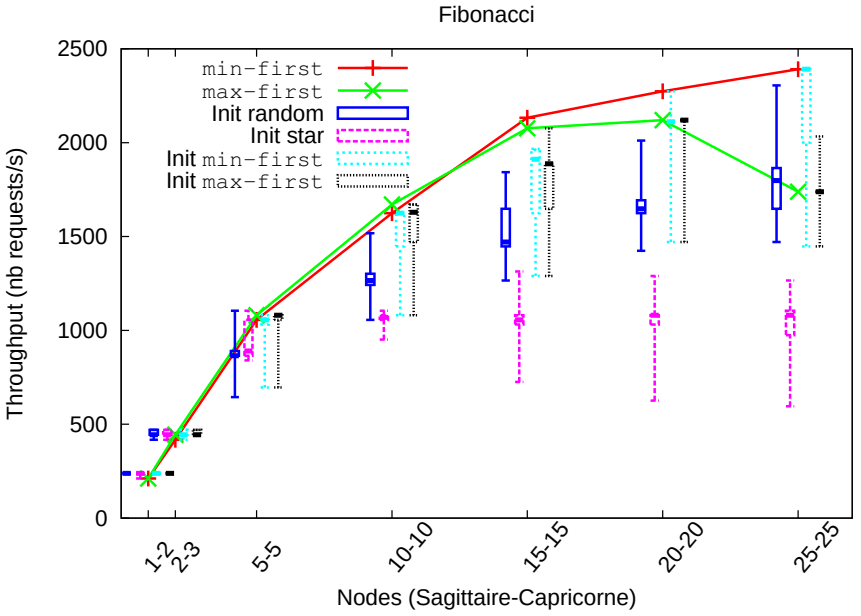
We also compared the results of the genetic algorithm for different initialization strategies. We compared four different strategies:

- *Random*: we first randomly choose the number of nodes for each service, they are mapped on random nodes. Then, we create a random number of agents, and finally we just connect everything randomly, but in a way that creates a valid hierarchy. This is the initialization method used in the previous section.
- *min-first* or *max-first*: we use the *random* initialization for all but one hierarchy. We initialize this latter with the hierarchy found by *min-first* or *max-first* (this of course is only possible for a communication homogeneous platform).
- *star-graph*: we randomly choose the type of each nodes, and ensure that only one can be an agent. Then elements are connected as a star graph under the agent.

Figures 16a and 16b present the results. The star-graph gives the worst results, as new levels of hierarchy can be created only through mutations, and they do not occur often. Without much surprise, the *min-first* and *max-first* initialization give the best results as they are guided by the result of the bottom-up algorithm. Finally, the random initialization give results that are in between *min-first* and *max-first* on the mean, but which best result is almost as good as *min-first* method. These results also assess the effectiveness of our approach, as starting from totally random hierarchies, we obtain results as good as with the *min-first* heuristic.



(a) `dgemm` 100



(b) Fibonacci 30

Figure 16: Different methods of initialization.

5.3 Experiments

We ran experiments on DIET hierarchies, with the same services as before: `dgemm` 100 and Fibonacci 30. We used three clusters present on three different sites of Grid'5000: Sagittaire in Lyon, Chti in Lille and Paradent in Rennes. Their respective computing power is $w_{sagittaire} = 3249MFlops$, $w_{Chti} = 3784MFlops$ and $w_{Paradent} = 4378MFlops$. Figure 17 present the comparison between theoretical and experimental throughput, and Table 3 presents the relative error. As can be seen, the experiments follow the model. Even though the error can be quite big for small platforms (less than 10 nodes), the performance prediction becomes more accurate for larger platforms, as the relative error remains lower than 16%. The higher errors obtained for small platforms can be explained by the fact that the platform benchmarks have been done by stressing the network and machines. This stress is more easily attained when many nodes are involved in the experiments.

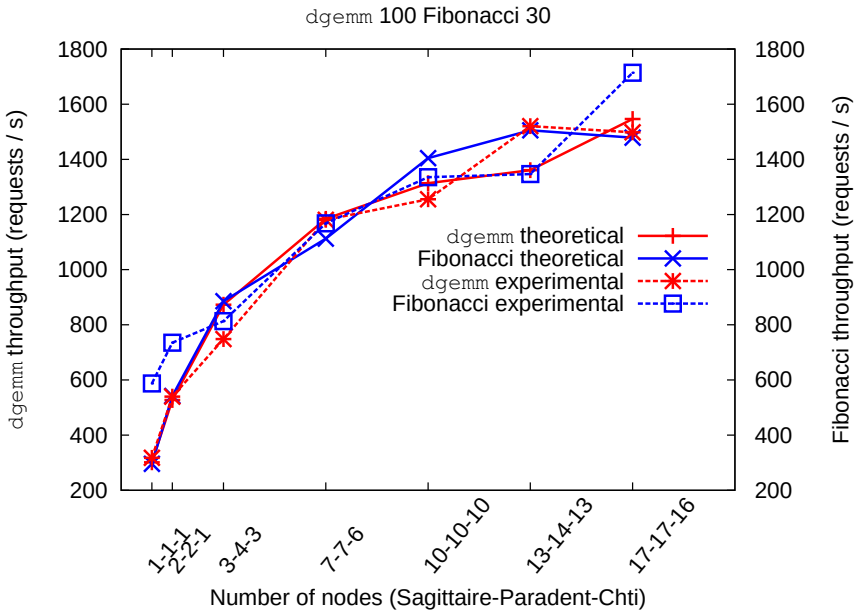


Figure 17: `dgemm` 100, Fibonacci 30 theoretical and experimental throughput, with genetic algorithm approach.

Client	Sagittaire-Paradent-Chti						
	1-1-1	**2-2-1**	**3-4-3**	**7-7-6**	**10-10-10**	**13-14-13**	**17-17-16**
`dgemm`	5.42%	2.29%	14.26%	0.33%	4.44%	11.73%	3.12%
Fibonacci	98.63%	35.70%	8.16%	4.96%	4.91%	10.53%	15.95%

Table 3: Relative Error, using genetic algorithm.

Figure 18 presents the shape of the hierarchy obtained for a 17-17-16 platform. As can be seen, not all the available nodes are used: as previously, adding more nodes does not necessarily provide better performances as it tends to overload the agents. Blue nodes are on the Chti cluster, red ones on Paradent, and the white ones on Sagittaire.

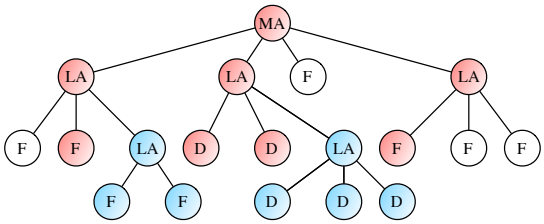


Figure 18: Hierarchy shape for a 17-17-16 platform.

6 Conclusion

In this paper we presented a computation and communication model for hierarchical middleware, when several services are available in the middleware. We proposed algorithms to find a hierarchy that gives the best obtained throughput to requested throughput ratio for all services on two different kinds of platforms: communication homogeneous/computation heterogeneous platforms, and fully heterogeneous platforms. The algorithm for communication homogeneous platforms uses a bottom-up approach, and is based on a linear program to successively determine levels of the hierarchy. The algorithm for totally heterogeneous platforms relies on a genetic algorithm. Our experiments on a real middleware, DIET, show that the obtained throughput closely follows what our model predicts and that both approaches (bottom-up algorithm, and genetic algorithm) provide excellent performances. We clearly showed that they add new levels of agents whenever required, and that it outperforms the classical approach of deploying the middleware as a balanced star graph.

7 Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] Abelkader Amar, Raphaël Bolze, Yves Caniou, Eddy Caron, Benjamin Depardon, Jean-Sébastien Gay, Gaël Le Mahec, and David Loureiro. Tunable scheduling in a GridRPC framework. *Concurrency and Computation: Practice and Experience*, 20(9):1051–1069, 2008.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[3] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[4] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate. Autonomic management policy specification in tune. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663, New York, NY, USA, 2008. ACM.

[5] S. Cahon, N. Melab, and E. G. Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 05 2004.

[6] Yves Caniou, Eddy Caron, Frédéric Desprez, Hidemoto Nakada, Keith Seymour, and Yoshio Tanaka. *Recent Developments in Grid Technology and Applications*, chapter High performance GridRPC middleware. Nova Science Publishers, April 2009. To appear.

[7] E. Caron, P.K. Chouhan, and A. Legrand. Automatic deployment for hierarchical network enabled servers. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 109–, April 2004.

[8] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. GoDiet : A deployment tool for distributed middleware on grid'5000. In IEEE, editor, *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15.*, pages 1–8, Paris, France, June 19th 2006.

[9] Eddy Caron, Benjamin Depardon, and Frédéric Desprez. Modelization for the Deployment of a Hierarchical Middleware on a Homogeneous Platform. Research Report RR-7201, INRIA, 02 2010. New experimental results compared to version 1 Comparisons with balanced star deployement since version 2.

[10] Eddy Caron and Frédéric Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[11] Henri Casanova and Jack Dongarra. Netsolve: a network server for solving computational science problems. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 40, Washington, DC, USA, 1996. IEEE Computer Society.

[12] P.K. Chouhan. *Automatic Deployment for Application Service Provider Environments*. PhD thesis, PhD thesis, Ecole Normale Supérieure de Lyon, 2006.

[13] Pushpinder Kaur Chouhan, Holly Dail, Eddy Caron, and Frédéric Vivien. Automatic middleware deployment planning on clusters. *Int. J. High Perform. Comput. Appl.*, 20(4):517–530, 2006.

[14] Areski Flissi and Philippe Merle. A generic deployment framework for grid computing and distributed applications. In *Proceedings of the 2nd International OTM Symposium on Grid computing, high-performAnce and Distributed Applications (GADA'06)*, volume 4279 of *Lecture Notes in Computer Science*, pages 1402–1411, Montpellier, France, nov 2006. Springer-Verlag.

[15] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[16] Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*, Seattle, WA, USA, November 2005. Springer-Verlag.

[17] J. P. Morrison, B. Clayton, D. A. Power, and A. Patil. Webcom-G: grid enabled metacomputing. *Neural, Parallel Sci. Comput.*, 12(3):419–438, 2004.

[18] Keith Seymour, Craig Lee, Frédéric Desprez, Hidemoto Nakada, and Yoshio Tanaka. The end-user and middleware apis for GridRPC. In *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12, Brussels, Belgium*, 2004.

[19] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-g: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing*, 1(1):41–51, 03 2003.

[20] Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada, and Satoshi Sekiguchi. Design, implementation and performance evaluation of gridrpc programming middleware for a large-scale computational grid. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 298–305, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Asim YarKhan, Jack Dongarra, and Keith Seymour. Gridsolve: The evolution of a network enabled solver. *Grid-Based Problem Solving Environments*, pages 215–224, 2007.