Laboratoire de l'Informatique du Parallélisme

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

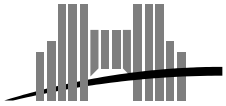
Automatic Middleware Deployment Planning on Clusters

Pushpinder Kaur Chouhan,
Holly Dail,
Eddy Caron,
Frédéric Vivien

Oct 2005

Research Report N° 2005-50

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



Automatic Middleware Deployment Planning on Clusters

Pushpinder Kaur Chouhan, Holly Dail, Eddy Caron, Frédéric Vivien

Oct 2005

Abstract

The use of many distributed, heterogeneous resources as a large collective resource offers great potential and has become an increasingly popular idea. A key issue for these Grid platforms is middleware scalability and how middleware services can best be mapped to the resource platform structure. Optimizing deployment is a difficult problem with no existing general solutions. In this paper we address a simpler subproblem: how to carry out an adapted deployment on a cluster with hundreds of nodes? Efficient use of clusters alone or as part of the Grid is an important issue.

In this paper we present an approach for automatically determining an optimal deployment for hierarchically distributed middleware services on clusters where the goal is to optimize steady-state request throughput. We prove that a complete spanning d-ary tree provides an optimal deployment and we present an algorithm to construct this optimal tree. We use a distributed Problem Solving Environment called DIET to test our approach. We define a performance model for each component of DIET and validate these models in a real-world cluster environment. Additional experiments demonstrate that our approach selects a deployment that performs better than other reasonable deployments.

Keywords: Deployment, Cluster, Cluster computing, Network Enabled Server, Steady-state scheduling.

Résumé

L'utilisation de ressources hétérogènes distribuées comme une grande ressource collective offre un grand potentiel et est devenue une idée de plus en plus populaire, connue sous le nom de Grille. Une question importante pour les plates-formes de type Grille concerne l'extensibilité des intergiciels et par extension comment les services fournis par ces derniers peuvent être distribués sur la plate-forme.

L'optimisation du déploiement est un problème difficile pour lequel il n'existe pas de solution dans le cas général. Dans cet article nous réduisons le cas à un sous-problème plus simple: "comment effectuer un déploiement à l'échelle d'une grappe avec des centaines de noeuds?".

L'utilisation efficace de la grappe, seule ou en tant que sous-ensemble de la Grille est une question importante. Dans cet article nous présentons une approche pour déterminer automatiquement un déploiement optimal pour les intergiciels hiérarchiquement distribués sur une grappe, où le but est d'optimiser le débit en régime permanent.

Nous montrons qu'un arbre couvrant n-aire complet est un déploiement optimal puis proposons un algorithme pour construire un tel arbre. Pour valider expérimentalement notre approche, nous employons un environnement de résolution de problème appelé DIET. Nous définissons un modèle pour évaluer chaque composant de DIET et validons ces modèles expérimentalement. Les expériences menées démontrent que notre approche choisit un déploiement efficace.

Mots-clés: Déploiement, Grappe, Calcul sur grappes, Serveurs de calcul distants, Ordonnancement en régime permanent.

1 Introduction

Due to the scale of Grid platforms, as well as the geographical localization of resources, middleware approaches should be distributed to provide scalability and adaptability. Much work has focused on the design and implementation of distributed middleware. To benefit most from such approaches, an appropriate mapping of middleware components to the distributed resource environment is needed. However, while middleware designers often note that this problem of deployment planning is important, only a few algorithms exist [4, 5, 14, 15] for efficient and automatic deployment. Questions such as "which resources should be used?", "how many resources should be used?", and "should the fastest and best-connected resource be used for middleware or as a computational resource?" remain difficult to answer.

Before deploying on the scale of the Grid, the first problem encountered by users is "how to manage an adapted deployment on a cluster with tens to hundreds of nodes?". While the homogeneous properties of such a platform simplify many aspects of the problem, this article will show that the task is not as simple as one would think.

The goal of this paper is to provide an automated approach to deployment planning that provides a good deployment for client-server middleware environments in homogeneous cluster environments. We consider that a *good* deployment is one that maximizes the throughput of the system in terms of satisfaction of client requests. We define *deployment planning* as the process of assigning resources to middleware components with the goal of optimizing throughput. In this paper the phrase deployment planning is often shortened to deployment. We focus on hierarchically distributed client-server middleware approaches. A hierarchical arrangement is a simple and effective distribution approach and has been chosen by a variety of middleware environments as their primary distribution approach [7, 9, 12, 20]. Given the popularity of this approach in middleware, automatic deployment for such middleware is an important problem.

We prove in this paper that a complete d-ary spanning tree provides an optimal deployment for hierarchical middleware environments on homogeneous clusters. We use a distributed Problem Solving Environment (PSE) as a test case for our approach. This PSE, the Distributed Interactive Engineering Toolbox (DIET) [7], uses a hierarchical arrangement of agents and servers to provide scalable, high-throughput scheduling and application provision services. In [7], the authors showed that a centralized agent can become a bottleneck if a simple star deployment is used for DIET. To apply our approach to DIET, we first develop a performance model for DIET components. We present experiments performed in a real-world cluster environment to validate these performance models. We then present real-world experiments demonstrating that the deployment chosen by our approach performs well as compared to other reasonable deployments.

The primary contributions of this work are the definition of an optimal deployment for hierarchical middleware in cluster environments, the development of performance models for DIET, and the real-world experiments testing the maximum throughput of a variety of deployments using a real middleware package. However, we also believe that the results of this work are an important step towards deployment on Grids. In particular, we believe that the results of this work can be easily applied to Grids composed of a star or cluster of clusters.

The rest of this article is organized as follows. Section 2 presents related work on the subject of deployment and deployment planning. In Section 3 the architectural model and a proof of

optimal deployment, and an algorithm for optimal deployment construction are presented. Section 4 gives an overview of DIET and describes the performance models we developed for DIET. Section 5 presents experiments that validate this work. Finally, Section 6 concludes the paper and describes future work.

2 Related work

A deployment is the mapping of a platform and middleware across many resources. Deployment can be broadly divided in two categories: software deployment and system deployment. *Software deployment* maps and distributes a collection of software components on a set of resources. Software deployment includes activities such as releasing, configuring, installing, updating, adapting, de-installing, and even de-releasing a software system. The complexity of these tasks is increasing as more sophisticated architectural models, such as systems of systems and coordinated distributed systems, become more common. Many tools have been developed for software deployment; examples include ManageSoft [1], ootPrints Software Deployment [2], Software Dock [13], SmartFrog [10], and Ant [11].

System deployment involves two steps, physical and logical. In physical deployment all hardware is assembled (network, CPU, power supply, etc.), whereas logical deployment is organizing and naming whole cluster nodes as master, slave, etc. As deploying systems can be a time consuming and cumbersome task, tools such as Deployment Toolkit [1] and Kadeploy [17] have been developed to facilitate this process.

Although these toolkits can automate many of the tasks associated with deployment, they do not automate the decision process of finding an appropriate mapping of specialized middleware components to resources so that the best performance can be achieved from the system.

To the best of our knowledge, no deployment algorithm or model has been given for arranging the components of a PSE in such a way as to maximize the number of requests that can be treated in a time unit. In [16], software components based on the CORBA component model are automatically deployed on the computational Grid. The CORBA component model contains a deployment model that specifies how a particular component can be installed, configured and launched on a machine. The authors note a strong need for deployment planning algorithms, but to date they have focused on other aspects of the system. Our work is thus complementary.

In [5] we presented a heuristic approach for improving deployments of hierarchical NES systems in heterogeneous Grid environments. The approach is iterative; in each iteration, mathematical models are used to analyze the existing deployment, identify the primary bottleneck, and remove the bottleneck by adding resources in the appropriate area of the system. The techniques given in [5] are heuristic and iterative in nature and can only be used to improve the throughput of a deployment that has been defined by other means; the current work provides an optimal solution to a more limited case and does not require a predefined deployment as input.

Optimizing deployment is an evolving field. In [14], the authors propose an algorithm called Sikitei to address the Component Placement Problem (CPP). This work leverages existing AI

[1] http://www.managesoft.com/

[2] http://www.unipress.com/footprints/deploy.html

planning techniques and the specific characteristics of CPP. In [15] the Sikitei approach is extended to allow optimization of resource consumption and consideration of plan costs. The Sikitei approach focuses on satisfying component constraints for effective placement, but does not consider detailed but sometimes important performance issues such as the effect of the number of connections on a component's performance.

The Pegasus System [4] frames workflow planning for the Grid as a planning problem. The approach is interesting for overall planning when one can consider that individual elements can communicate with no performance impact. Our work is more narrowly focused on a specific style of assembly and interaction between components and has a correspondingly more accurate view of performance to guide the deployment decision process.

3 Platform deployment

Our objective is to generate a best platform from the available resources so as to maximize the throughput. Throughput is the maximum number of requests that are serviced for clients in a given time unit.

3.1 Platform architecture

This section defines our target platform architecture; Figure 1 provides a useful reference for these definitions.

Software system architecture - We consider a service-provider software system composed of three types of elements: A set of client nodes $\mathbb{C}$ that require computation, a set of server nodes $\mathbb{S}$ that are providers of computation, and a set of agent nodes $\mathbb{A}$ that provide coordination of client requests with service offerings via service localization, scheduling, and persistent data management. The arrangement of these elements is shown in Figure 1. We consider only hierarchical arrangements of agents composed of a single top-level root agent and any number of agents arranged in a tree below the root agent. Server nodes are leaves of the tree, but may be attached to any agent in the hierarchy, even if that agent also has children that are agents.

Since the use of multiple agents is designed to distribute the cost of services such as scheduling, there is no performance advantage to having an agent with a single child. The only exception to this policy is for the root-level agent with a single server child; this "chain" can not be reduced.

Thus a server $s \in \mathbb{S}$ has exactly one parent that is always an agent $a \in \mathbb{A}$, a root agent $a \in \mathbb{A}$ has no parent and one or more child agents and/or servers, and non-root agents $a \in \mathbb{A}$ have exactly one parent and two or more child agents and/or servers.

Request definition - We consider a system that processes *requests* as follows. A client $c \in \mathbb{C}$ first generates a *scheduling request* which contains information about the service required by the client and meta-information about any input data sets, but does not include the actual input data. The scheduling request is submitted to the root agent, which checks the scheduling request and forwards it on to its children. Other agents in the hierarchy perform the same operation until the scheduling request reaches the servers. We assume that the scheduling request is forwarded to all servers, though this is a worst case scenario as filtering may be done by the agents based

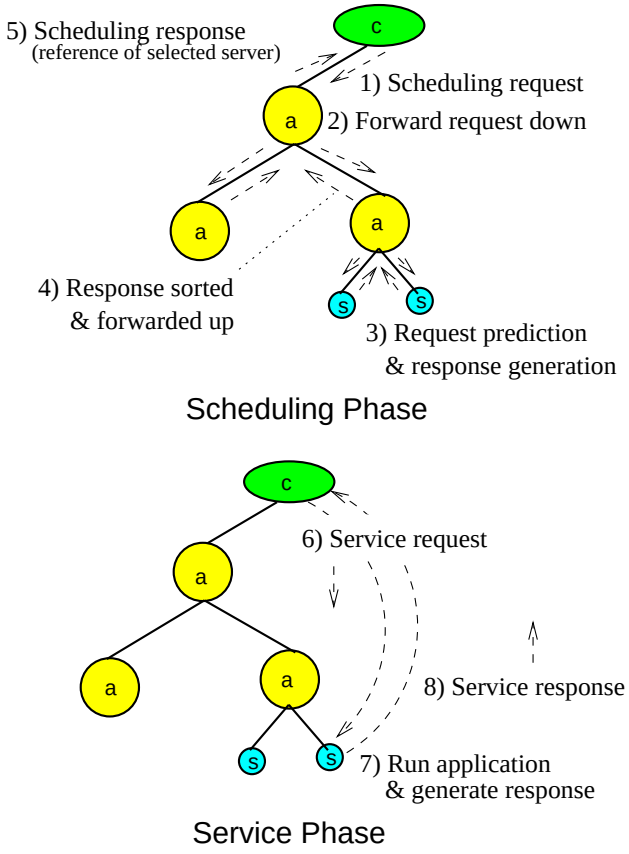


Figure 1: Platform deployment architecture and execution phases.

on request type. Servers may or may not make predictions about performance for satisfying the request, depending on the exact system.

Servers that can perform the service then generate a *scheduling response*. The scheduling response is forwarded back up the hierarchy and the agents sort and select amongst the various scheduling responses. It is assumed that the time required by an agent to select amongst scheduling responses increases with the number of children it has, but is independent of whether the children are servers or agents. Finally, the root agent forwards the chosen scheduling response (i.e., the selected server) to the client.

The client then generates a *service request* which is very similar to the scheduling request but includes the full input data set(s), if any are needed. The service request is submitted by the client to the chosen server. The server performs the requested service and generates a *service response*, which is then sent back to the client. 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.

Resource architecture - The target resource architectural framework is represented by a weighted graph $G = (\mathbb{V}, \mathbb{E}, w, B)$. Each vertex v in the set of vertices $\mathbb{V}$ represents a computing

resource with computing power w in MFlop/second. Each edge e in the set of edges $\mathbb{E}$ represents a resource link between two resources with edge cost B given by the bandwidth between the two nodes in Mb/second. To simplify our model we do not consider latency in data transfer costs.

Deployment assumptions - We consider that at the time of deployment we do not know the client locations or the characteristics of the client 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 $\mathbb{V}$.

A valid deployment thus consists of a mapping of a hierarchical arrangement of agents and servers onto the set of resources $\mathbb{V}$. Any server or agent in a deployment must be connected to at least one other element; thus a deployment can have only connected nodes. A valid deployment will always include at least the root-level agent and one server. Each node $v \in \mathbb{V}$ can be assigned to either exactly one server s, exactly one agent a or the node can be left idle. Thus if the total number of agents is $|\mathbb{A}|$, the total number of servers is $|\mathbb{S}|$, and the total number of resources is $|\mathbb{V}|$, then $|\mathbb{A}| + |\mathbb{S}| \leqslant |\mathbb{V}|$.

Note that since the use of multiple agents is designed to distribute the cost of services such as scheduling, there is no performance advantage to having an agent with a single child. Thus, any chain can and should be reduced by moving the leaf child of the chain into the position of the first agent in the chain. The only exception to this policy is for the root-level agent with a single server child; this "chain" can not be reduced.

3.2 Optimal deployment

Our **objective** in this section is to *find an optimal deployment of agents and servers for a set of resources* $\mathbb{V}$. We consider an *optimal deployment* to be a deployment that provides the maximum throughput ρ of completed requests per second. When the maximum throughput can be achieved by multiple distinct deployments, the preferred deployment is the one using the least resources.

As described in Section 3.1, we assume that at the time of deployment we do not know the locations of clients or the rate at which they will send requests. Thus it is impossible to generate an optimized, complete schedule. Instead, we seek a deployment that maximizes the *steady-state throughput*. Our model is based on steady-state scheduling techniques [3] where startup and shutdown phases are not considered and the precise ordering and allocation of tasks and messages are not required. Instead, the main goal is to characterize the *average* activities and capacities of each resource during each time unit.

We define the scheduling request throughput in requests per second, ρ_{sched}, as the rate at which requests are processed by the scheduling phase (see Section 3.1). Likewise, we define the service throughput in requests per second, $\rho_{service}$, as the rate at which the servers produce the services required by the clients. The following lemmas lead to a proof of an optimal deployment.

Lemma 1. *The completed request throughput ρ of a deployment is given by the minimum of the scheduling request throughput ρ_{sched} and the service request throughput $\rho_{service}$.*

$$\rho = min(\rho_{sched}, \rho_{service})$$

Proof. A completed request has, by definition, completed both the scheduling request and the service request phases; we neglect the treatment of requests at the clients since we assume no details are available about clients at deployment time.

Case 1: $\rho_{sched} \geqslant \rho_{service}$. In this case requests are sent to the servers at least as fast as they can be serviced by the servers, so the overall rate is limited by $\rho_{service}$.

Case 2: $\rho_{sched} < \rho_{service}$. In this case the servers are left idle waiting for requests and new requests are processed by the servers faster than they arrive. The overall throughput is thus limited by ρ_{sched}.

□

The *degree* of an agent is the number of children directly attached to it, regardless of whether the children are servers or agents.

Lemma 2. *The scheduling throughput ρ_{sched} is limited by the throughput of the agent with the highest degree.*

Proof. As described in Section 3.1, we assume that the time required by an agent to manage a request increases with the number of children it has. Thus, agent throughput decreases with increasing agent degree and the agent with the highest degree will provide the lowest throughput. Since we assume that scheduling requests are forwarded to all agents and servers, a scheduling request is not finished until all agents have responded. Thus ρ_{sched} is limited by the agent providing the lowest throughput which is the agent with the highest degree. □

Lemma 3. *The service request throughput $\rho_{service}$ increases as the number of servers included in a deployment increases.*

Proof. The service request throughput is a measure of the rate at which servers in a deployment can generate responses to client service requests. Since agents do not participate in this process, $\rho_{service}$ is independent of the agent hierarchy. The computational power of the servers is used for both (1) generating responses to scheduling queries from the agents and (2) providing computational services for clients. For a given value of ρ_{sched} the work performed by a server for activity (1) is independent of the number of servers. The work performed by each server for activity (2) is thus also independent of the number of servers. Thus the work performed by the servers as a group for activity (2) increases as the number of servers in the deployment increases.

□

For the rest of this section, we need some precise definitions. A *complete d-ary tree* is a tree for which all internal nodes, except perhaps one, have exactly d children. If n is the depth of the tree, the remaining internal node is at depth $n-1$ and may have any degree from 1 to $d-1$; for our purposes, a degree of 1 can only exist for the root node. Leaf nodes are at level n or $n-1$. A *spanning tree* is a connected, acyclic subgraph containing all the vertices of a graph. We introduce the following definition to aid later discussions.

Definition 1. *A **complete spanning d-ary tree (CSD tree)** is a tree that is both a complete d-ary tree and a spanning tree.*

For deployment, leaves are servers and all other nodes are agents. A degree d of one is useful only for a deployment of a single root agent and a single server. Note that for a set of resources $\mathbb{V}$ and degree d, a large number of CSD trees can be constructed. However, since we consider only homogeneous resources, all such CSD trees are equivalent in that they provide exactly the same performance. Thus, we consider that for any given $\mathbb{V}$ and d, exactly one distinct CSD tree can be constructed.

Definition 2. *A **dMax set** is the set of all trees for which the maximum degree is equal to dMax.*

Figure 2 shows some examples of trees from the dMax 4 and dMax 6 sets.

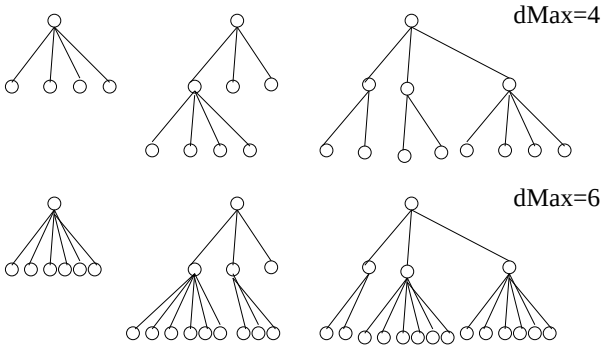


Figure 2: Deployment trees of $dMax$ set.

Theorem 1. *In a dMax set, for all deployment tree with optimal throughput, a corresponding dMax CSD tree exists.*

Proof. By Lemma 1, the throughput ρ of the CSD tree of degree $dMax$ will be limited by either schedule request throughput ρ_{sched} and/or service request throughput $\rho_{service}$.

By Lemma 2, scheduling throughput is limited by the throughput of the agent with the highest degree. Therefore, the scheduling request throughput ρ_{sched} of all trees in a dMax set will be the same. Thus, to show that the CSD tree is an optimal solution we must show that the service throughput $\rho_{service}$ of the CSD tree is at least as large as the $\rho_{service}$ of all other trees in the dMax set.

By Lemma 3, we know that service throughput $\rho_{service}$ increases with the number of servers included in a deployment. Given the limited resource set size $|\mathbb{V}|$, the number of servers is largest for deployments with the smallest number of agents.

There is atleast one or more trees in dMax set that has optimal $\rho_{service}$. This tree can be a CSD or an non CSD tree. If the tree is a non CSD tree with height h then it has nodes with degree less than dMax at depth h', where $h' < h - 1$.

Move the required number of nodes from depth h to the node that have degree less than dMax at level h' and due to this, may be height of the tree is reduced, then change h by $h - 1$. Repeat the node movement procedure till all the nodes at level h' have dMax children and all the leave nodes are at level h'', where $h'' >= h - 1$. The node movement procedure may increase the

total number of leave nodes in the tree. According to CSD tree definition, a tree constructed with limited resource set size $|\mathbb{V}|$, has leave nodes at level h'' and at most one internal node with degree less than dMax at level $h-1$, is a CSD tree. So, any tree can be converted to a CSD tree without decreasing the number of leave nodes.

Thus, the dMax CSD tree provides a $\rho_{service}$ that is at least as large as that provided by all other trees in the dMax set, and the dMax CSD tree is therefore an optimal solution in the dMax set.

□

Theorem 2. *A complete spanning d-ary tree with degree $d \in [1, |\mathbb{V}|-1]$ that maximizes the minimum of the scheduling request and service request throughputs is an optimal deployment.*

Proof. This theorem is fundamentally a corollary of Theorem 1. The optimal degree is not known a priori; it suffices to test all possible degrees $d \in [1, |\mathbb{V}|-1]$ and to select the degree that provides the maximum completed request throughput. □

3.3 Deployment construction

Given Theorem 2, it is straightforward to find the optimal degree CSD tree; see Algorithm 1 for details.

$best\_d = 1$
$best\_\rho = 0$
for all $d \in [1, |\mathbb{V}|-1]$ **do**
  Calculate ρ_{sched}
  Calculate $\rho_{service}$
  $\rho = \min(\rho_{sched}, \rho_{service})$
  if $\rho > best\_\rho$ **then**
    $best\_\rho = \rho$
    $best\_d = d$

Algorithm 1: Find optimal degree.

Once an optimal degree $best\_d$ has been calculated using Algorithm 1, we can use Algorithm 2 to construct the optimal CSD tree.

A few examples will help clarify the results of our deployment planning approach. Let us consider that we have 10 available nodes ($|\mathbb{V}| = 10$). Suppose $best\_d = 1$. Algorithm 2 will construct the corresponding best platform - the single root agent with a single server attached. Now suppose $best\_d = 4$. Then Algorithm 2 will construct the corresponding best deployment - the root agent with four children, one of which also has four children; the deployment has two agents, seven servers and one unused node because it can only be attached as a chain.

calculate $best\_d$ using Algorithm 1
Add the root node
if $best\_d == 1$ **then**
  add one server to root node
  Exit
$availNodes = |\mathbb{V}| - 1$
$level = 0$
while $availNodes > 0$ **do**
  if $\exists$ agent at depth $level$ with degree 0 **then**
    $toAdd$ = min($best\_d, availNodes$)
    add $toAdd$ children to the agent
    $availNodes = availNodes - toAdd$
  else
    $level = level + 1$
if $\exists$ agent with degree 1 **then**
  remove the child
convert all leaf nodes to servers

Algorithm 2: Optimal tree construction.

4 Implementation with DIET

To organize the nodes in an efficient manner and use the nodes' power efficiently is out of the scope of end users. That is why end-users have to rely on specialized middleware, like Problem Solving Environments (PSE), to run their applications. Some PSEs, like NetSolve [2], Ninf [19], or DIET [7], already exist and are commonly called Network Enabled Server (NES) environments [18]. We illustrate our deployment approach by applying the results to an existing hierarchical PSE called DIET.

4.1 DIET overview

The Distributive Interactive Engineering Toolbox is built around five main components. The *Client* is an application that uses DIET to solve problems. Agents are used to provide services such as data localization and scheduling. These services are distributed across a hierarchy composed of a single *Master Agent* (MA) and zero or more *Local Agents* (LA). *Server Daemons (SeD)* are the leaves of the hierarchy and may provide a variety of computational services. The MA is the entry point of the DIET environment and thus receives all service requests from clients. The MA forwards service requests onto other agents in the hierarchy, if any exist. Once the requests reach the SeDs, each SeD replies with a prediction of its own performance for the request. Agents sort the response(s) and pass on only the best response(s). Finally, the MA forwards the best server back to the client. The client then submits its service request directly to the selected SeD. The inclusion of LAs in a DIET hierarchy can provide scalability and adaptation to diverse

network environments. Figure 3 shows some possible DIET deployments.

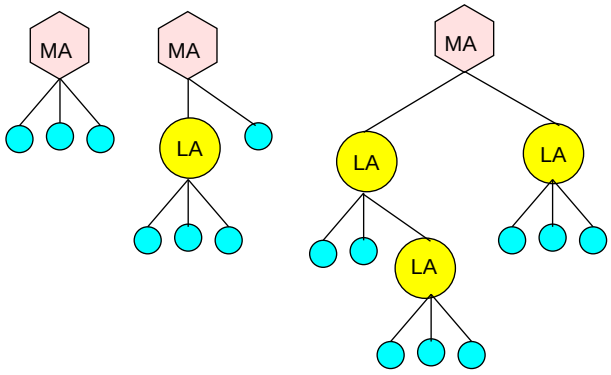


Figure 3: Different possible DIET deployments. Root agent can have either servers/agents or both as children and other agents can have either both or only servers as children.

4.2 Request performance modeling

In order to apply the model defined in Section 3 to DIET, we must have models for the scheduling throughput and the service throughput in DIET. In this section we define performance models to estimate the time required for various phases of request treatment in DIET. These models will be used in the following section to create the needed throughput models.

We make the following assumptions about DIET for performance modeling. The MA and LA are considered as having the same performance because their activities are almost identical and in practice we observe only negligible differences in their performance. The root of the tree is always an MA and thus all clients will submit their request to the DIET platform through one MA. We assume that the work required for an agent to treat responses from SeD-type children and from agent-type children is the same. DIET allows configuration of the number of responses forwarded by agents; here we assume that only the best server is forwarded to the parent.

When client requests are sent to the agent hierarchy, DIET is optimized such that large data items like matrices are not included in the problem parameter descriptions (only their sizes are included). These large data items are included only in the final request for computation from client to server. As stated earlier, we assume that we do not have a priori knowledge of client locations and request submission patterns. Thus, we assume that needed data is already in place on the servers and we do not consider data transfer times.

The following variables will be of use in our model definitions.

S_{req} is the size in Mb of the message forwarded down the agent hierarchy for a scheduling request. This message includes only parameters and not large input data sets.

S_{rep} is the size in Mb of the reply to a scheduling request forwarded back up the agent hierarchy. Since we assume that only the best server response is forwarded by agents, the size of the reply does not increase as the response moves up the tree.

W_{req} is the amount of computation in MFlop needed by an agent to process one incoming request.

$W_{rep}(d)$ is the amount of computation in MFlop needed by an agent to merge the replies from its d children.

W_{pre} is the amount of computation in Mflop needed for a server to predict its own performance for a request.

W_{app} is the amount of computation in Mflop needed by a server to complete a service request for *app* service. The provision of this computation is the main goal of the DIET system.

Agent communication model: To treat a request, an agent *receives* the request from its parent, *sends* the request to each of its children, *receives* a reply from each of its children, and *sends* one reply to its parent. By Lemma 2, we are concerned only with the performance of the agent with the highest degree, d. The time in seconds required by an agent for receiving all messages associated with a request from its parent and children is as follows.

$$agent\_receive\_time = \frac{S_{req} + d \cdot S_{rep}}{B} \tag{1}$$

Similarly, the time in seconds required by an agent for sending all messages associated with a request to its children and parent is as follows.

$$agent\_send\_time = \frac{d \cdot S_{req} + S_{rep}}{B} \tag{2}$$

Server communication model: Servers have only one parent and no children, so the time in seconds required by a server for receiving messages associated with a scheduling request is as follows.

$$server\_receive\_time = \frac{S_{req}}{B} \tag{3}$$

The time in seconds required by a server for sending messages associated with a request to its parent is as follows.

$$server\_send\_time = \frac{S_{rep}}{B} \tag{4}$$

Agent computation model: Agents perform two activities involving computation: the processing of incoming requests and the selection of the best server amongst the replies returned from the agent's children.

There are two activities in the treatment of replies: a fixed cost W_{fix} in Mflops and a cost W_{sel} that is the amount of computation in MFlops needed to process the server replies, sort them, and select the best server. Thus the computation associated with the treatment of replies is given

$$W_{rep}(d) = W_{fix} + W_{sel} \cdot d$$

The time in seconds required by the agent for the two activities is given by the following equation.

$$agent\_comp\_time = \frac{W_{req} + W_{rep}(d)}{w} \tag{5}$$

Server computation model: Servers also perform two activities involving computation: performance prediction as part of the scheduling phase and provision of application services as part of the service phase. Let us consider a deployment with a set of servers $\mathbb{S}$ and the activities involved in completing $|\mathbb{S}|$ requests at the server level. All servers complete $|\mathbb{S}|$ prediction requests and each server will complete one service request phase, on average. As a whole, the servers as a group require the following time in seconds to complete the $\mathbb{S}$ requests.

$$\frac{W_{pre} \cdot |\mathbb{S}| + W_{app}}{w}$$

We divide by the number of requests $|\mathbb{S}|$ to obtain the average time required per request by the servers as a group.

$$server\_comp\_time = \frac{W_{pre} + \frac{W_{app}}{|\mathbb{S}|}}{w} \tag{6}$$

4.3 Steady-state throughput modeling

In this section we present models for scheduling and service throughput in DIET. We consider two different theoretical models for the capability of a computing resource to do computation and communication in parallel.

send or receive or compute, single port

In this model, a computing resource has no capability for parallelism: it can either send a message, receive a message, or compute. Only a single port is assumed: messages must be sent serially and received serially. This model is unrealistic for large data transfers as the CPU is not active 100% of the time during a transfer. However, for a service provider system the request messages can be very small. For very small messages, most of the transfer time is taken in setting up and closing down the communication, rather than in the data transfer itself. Since the CPU is often necessary for these activities, this communication model may be reasonable for our case.

For this model, the scheduling throughput in requests per second is then given by the minimum of the throughput provided by the servers for prediction and by the agents for scheduling.

$$\rho_{sched} = min\left(\frac{1}{\frac{W_{pre}}{w} + \frac{S_{req}}{B} + \frac{S_{rep}}{B}}, \frac{1}{\frac{S_{req}+d\cdot S_{rep}}{B} + \frac{d\cdot S_{req}+S_{rep}}{B} + \frac{W_{req}+W_{rep}(d)}{w}}\right) \tag{7}$$

The service throughput in requests per second is given by the following equation.

$$\rho_{service} = \frac{1}{\frac{S_{req}}{C} + \frac{S_{rep}}{C} + \frac{W_{pre}+\frac{W_{app}}{|\mathbb{S}|}}{w}} \tag{8}$$

send || receive || compute, single port

In this model, it is assumed that a computing resource can send messages, receive messages, and do computation in parallel. We still only assume a single port-level: messages must be sent serially and they must be received serially.

Thus, for this model the equations for scheduling and service throughput are as follows.

$$\rho_{sched} = min\left(\frac{1}{max\left(\frac{W_{pre}}{w}, \frac{S_{req}}{B}, \frac{S_{rep}}{B}\right)}, \frac{1}{max\left(\frac{S_{req}+d\cdot S_{rep}}{B}, \frac{d\cdot S_{req}+S_{rep}}{B}, \frac{W_{req}+W_{rep}(d)}{w}\right)}\right) \quad (9)$$

$$\rho_{service} = \frac{1}{max\left(\frac{S_{req}}{C}, \frac{S_{rep}}{C}, \frac{W_{pre}+\frac{W_{app}}{|\mathbb{S}|}}{w}\right)} \quad (10)$$

5 Experimental results

In this section we present experiments designed to test the ability of our deployment model to correctly identify good real-world deployments. Since our performance model and deployment approach focus on maximizing steady-state throughput, our experiments focus on testing the maximum sustained throughput provided by different deployments. The following section describes the experimental design, Section 5.2 describes how we obtained the parameters needed for the model, Section 5.3 presents experiments testing the accuracy of our throughput performance models, and Section 5.4 presents experiments testing whether the deployment selected by our approach provides good throughput as compared to other reasonable deployments. Finally, Section 5.5 provides some forecasts of good deployments for a range of problem sizes and resource sets.

5.1 Experimental design

Software: DIET 2.0 is used for all deployed agents and servers; DIET was compiled with GCC 3.3.5. GoDIET [6] version 2.0.0 is used to perform the actual software deployment.

Job types: In general, at the time of deployment, one can know neither the exact job mix nor the order in which jobs will arrive. Instead, one has to assume a particular job mix, define a deployment, and eventually correct the deployment after launch if it wasn't well-chosen. For these tests, we consider the DGEMM application, a simple matrix multiplication provided as part of the Basic Linear Algebra Subprograms (BLAS) package [8]. For example, when we state that we use DGEMM 100, it signifies that we use square matrices of dimensions 100x100. For each specific throughput test we use a single problem size; since we are testing steady-state conditions, the performance obtained should be equivalent to that one would attain for a mix of jobs with the same average execution time.

Workload: Measuring the maximum throughput of a system is non-trivial: if too little load is introduced the maximum performance may not be achieved, if too much load is introduced the performance may suffer as well. A unit of load is introduced via a script that runs a single request at a time in a continual loop. We then introduce load gradually by launching one client script every second. We introduce new clients until the throughput of the platform stops improving; we then let the platform run with no addition of clients for 10 minutes. Results presented are the average throughput during this 10 minute period. This test is hereafter called a *throughput test*.

Resources: The experiments were performed on two similar clusters. The first is a 55-node cluster at the Ecole Normale Supérieure in Lyon, France. Each node includes dual AMD Opteron 246 processors at 2 GHz, a cache size of 1024 KB, and 2 GB of memory. We used GCC 3.3.5 for all compilations and the linux kernel version was 2.6.8. All nodes are connected by both a Gigabit Ethernet and a 100 Mb/s Ethernet; our experiments used only the Gigabit Ethernet. We measured network bandwidth using the Network Weather Service [21]. Using the default NWS message size of 256 kB we obtain a bandwidth of 909.5 Mb/s; using the message size sent in DIET of 850 bytes we obtain a bandwidth of 20.0 Mb/s.

The second cluster is a 140-node cluster at Sophia in France. The nodes are physically identical to the ones at Lyon but are running the linux kernel version 2.4.21 and all compilations were done with GCC 3.2.3. The machines at Sophia are linked by 6 different Cisco Gigabit Ethernet switches connected with a 32 Gbps bus.

5.2 Model parametrization

Table 1 presents the parameter values we use for DIET in the models for ρ_{sched} and $\rho_{service}$. Our goal is to parametrize the model using only easy-to-collect micro-benchmarks. In particular, we seek to use only values that can be measured using a few clients executions. The alternative is to base the model on actual measurements of the maximum throughput of various system elements; while we have these measurements for DIET, we feel that the experiments required to obtain such measurements are difficult to design and run and their use would prove an obstruction to the application of our model for other systems.

To measure message sizes S_{req} and S_{rep} we deployed a Master Agent (MA) and a single DGEMM server (SeD) on the Lyon cluster and then launched 100 clients serially. We collected all network traffic between the MA and the SeD machines using `tcpdump` and analyzed the traffic to measure message sizes using the Ethereal Network Protocol analyzer[3]. This approach provides a measurement of the entire message size including headers. Using the same MA-SeD deployment, 100 client repetitions, and the statistics collection functionality in DIET [7], we then collected detailed measurements of the time required to process each message at the MA and SeD level. The parameter W_{rep} depends on the number of children attached to an agent. We measured the time required to process responses for a variety of star deployments including an MA and different numbers of SeDs. A linear data fit provided a very accurate model for the time required to process responses versus the degree of the agent with a correlation coefficient of 0.997. We thus use this linear model for the parameter W_{rep}. Finally, we measured the capacity

[3] http://www.ethereal.com

of our test machines in MFlops using a mini-benchmark extracted from Linpack and used this value to convert all measured times to estimates of the MFlops required.

Components	W_{req} (Mflop)	W_{rep} (Mflop)	W_{pre} (Mflop)	S_{rep} (Mb)	S_{req} (Mb)
Agent	1.4×10^6	$5.0 \times 10^7 + 5.3 \times 10^7 \cdot d$	-	6.4×10^{-5}	5.3×10^{-5}
SeD	-	-	7.1×10^6	6.4×10^{-5}	5.3×10^{-5}

Table 1: Parameter values

5.3 Throughput model validation

This section presents experiments testing the accuracy of the DIET agent and server throughput models presented in Section 4.3.

First, we examine the ability of the models to predict *agent throughput* and, in particular, to predict the effect of an agent's degree on its performance. To test agent performance, the test scenario must be clearly agent-limited. Thus we selected a very small problem size of DGEMM 10. To test a given agent degree d, we deployed an MA and attached d SeDs to that MA; we then ran a throughput test as described in Section 5.1. The results are presented in Figure 4. We verify that these deployments are all agent-limited by noting that the throughput is lower for a degree of two than for a degree of 1 despite the fact that the degree two deployment has twice as many SeDs.

Figures 4 (a) and (b) present model predictions for the serial and parallel models, respectively. In each case three predictions are shown using different values for the network bandwidth. The values of 20 Mb/s and 909.5 Mb/s are the values obtained with NWS with DIET's message size and the default message size, respectively. Comparison of predicted and measured leads us to believe that these measurements of the network bandwidth are not representative of what DIET actually obtains. This is not surprising given that DIET uses very small messages and network performance for this message size is highly sensitive to the communication layers used. The third bandwidth in each graph is chosen to provide a good fit of the measured and predicted values. For the purposes of this rest of this paper we will use the serial model with a bandwidth of 190 Mb/s because it provides a better fit than the parallel model. In the future we plan to investigate other measurement techniques for bandwidth that may better represent the bandwidth achieved when sending many very small messages as is done by DIET.

Next, we test the accuracy of throughput prediction for the *servers*. To test server performance, the test scenario must be clearly SeD-limited. Thus we selected a relatively large problem size of DGEMM 1000. To test whether performance scales as the number of servers increases, we deployed an MA and attached different numbers of SeDs to the MA. The results are presented in Figure 5. Only the serial model with a bandwidth of 190 Mb/s is shown; in fact, the results with the parallel model and with different bandwidths are all within 1% of this model since the communication is overwhelmed by the solve phase itself.

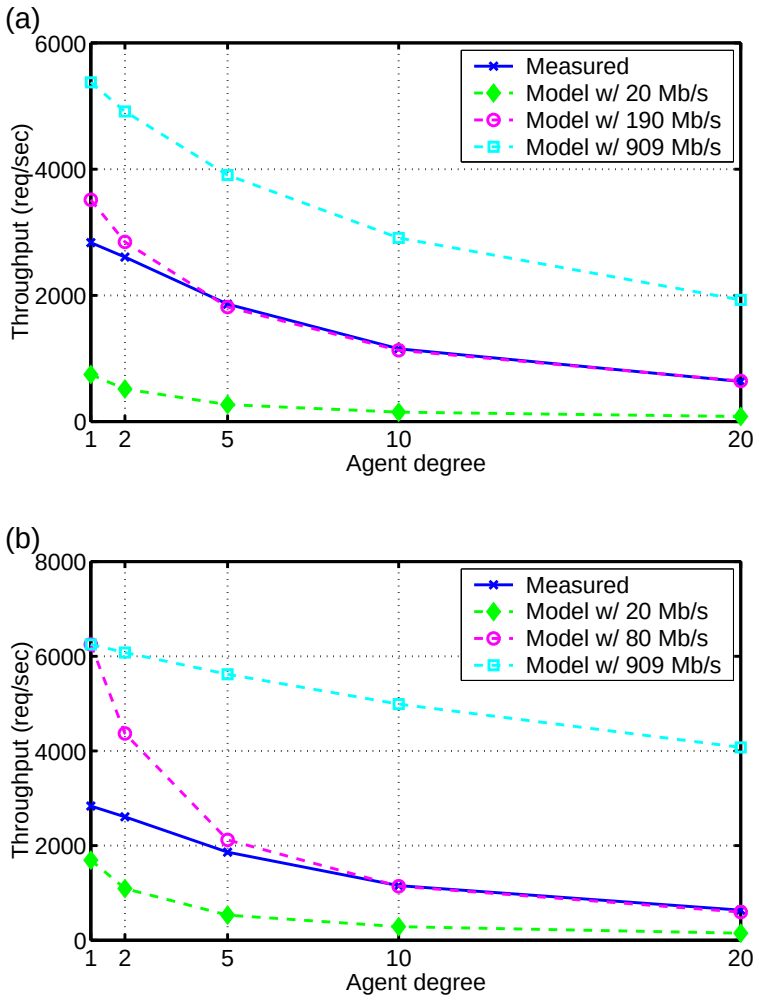


Figure 4: Measured and predicted platform throughput for DGEMM size 10; predictions are shown for several bandwidths and for (a) the serial model and (b) the parallel model.

5.4 Deployment selection validation

In this section we present experiments that test the effectiveness of our deployment approach in selecting a good deployment. For each experiment, we select a cluster, define the total number of resources available, and define a DGEMM problem size. We then apply our deployment algorithms to predict which CSD tree will provide the best throughput and we measure the throughput of this CSD tree in a real-world deployment. We then identify and test a suitable range of other CSD trees including the star, the most popular middleware deployment arrangement.

Figure 6 shows the predicted and actual throughput for a DGEMM size of 200 where 25 nodes in the Lyon cluster are available for the deployment. Our model predicts that the best throughput is provided by CSD trees with degrees of 12, 13 and 14. These trees have the same predicted throughput because they have the same number of SeDs and the throughput is limited

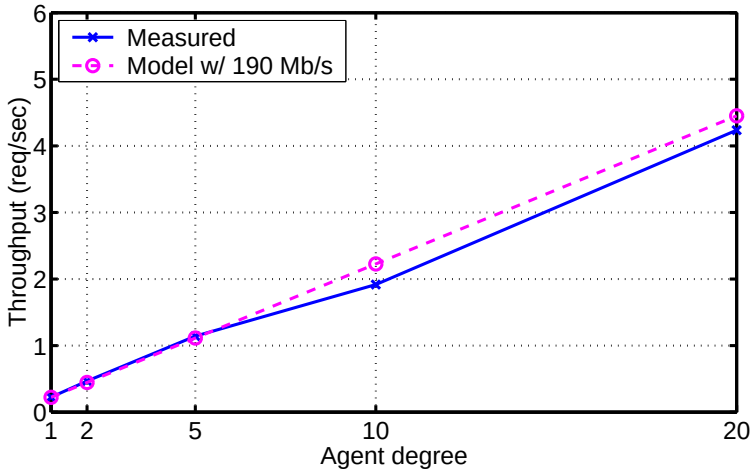


Figure 5: Measured and predicted platform throughput for DGEMM size 1000; predictions are shown for the serial model with bandwidth 190 Mb/s.

by the SeDs. Experiments show that the CSD tree with degree 12 does indeed provide the best throughput; the selected tree is shown in Figure 7. The model prediction overestimates the throughput; we believe that there is some cost associated with having multiple levels in a hierarchy that is not accounted for in our model. However, it is more important that the model correctly predicts the shape of the graph and identifies the best degree than that it correctly predicts absolute throughput.

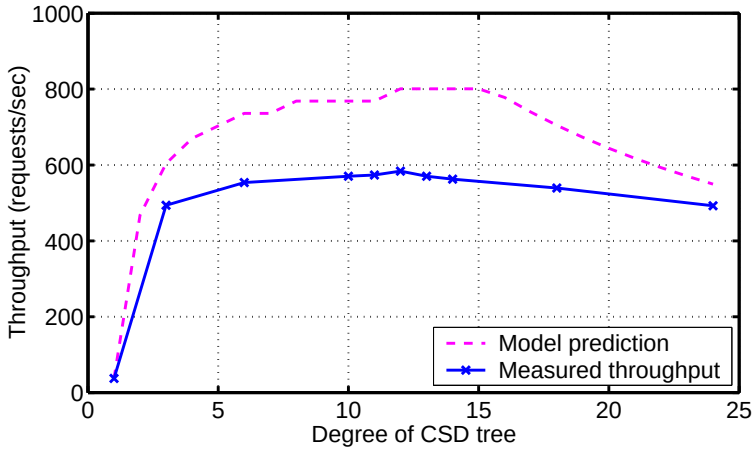


Figure 6: Predicted and measured throughput for different CSD trees for DGEMM 200 with 25 available nodes in the Lyon cluster.

For the next experiment, we use the same problem size of 200 but change the number of available nodes to 45 and the cluster to Sophia. We use the same problem size to demonstrate

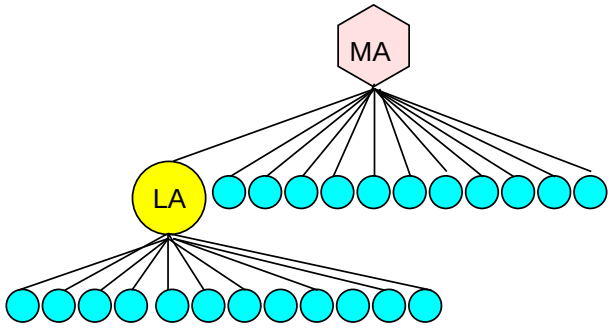


Figure 7: Degree 12 CSD-tree with 25 nodes.

that the best deployment is dependent on the number of resources available, rather than just the type of problem. The results are shown in Figure 8. The model predicts that the best deployment will be a degree eight CSD tree while experiments reveal that the best degree is three. The model does however correctly predict the shape of the curve and selects a deployment that achieves a throughput that is 87.1% of the optimal. By comparison, the popular star deployment (degree 44) obtains only 40.0% of the optimal performance.

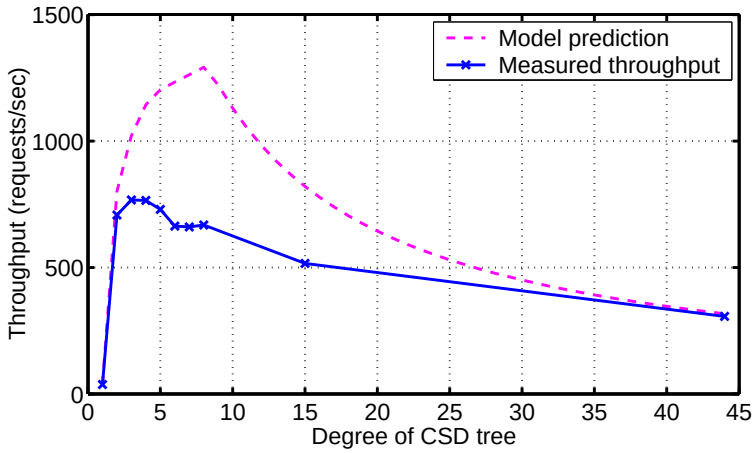


Figure 8: Predicted and measured throughput for different CSD trees for DGEMM 200 with 45 available nodes in the Sophia cluster.

For the last experiment, we again use a total of 45 nodes from the Sophia cluster but we increase the problem size to 310; we use the same resource set size to show that the best deployment is also dependent on the type of workload expected. The results are shown in Figure 9. In this test case, the model predictions are generally much more accurate than in the previous two cases; this is because $\rho_{service}$ is the limiting factor over a greater range of degrees due to the larger problem size used here. Our model predicts that the best deployment is a 22 degree CSD tree while in experimentation the best degree is 15. However, the deployment chosen by

our model achieves a throughput that is 98.5% of that achieved by the optimal 15 degree tree. By comparison, the star and tri-ary tree deployments achieve only 73.8% and 24.0% of the optimal throughput.

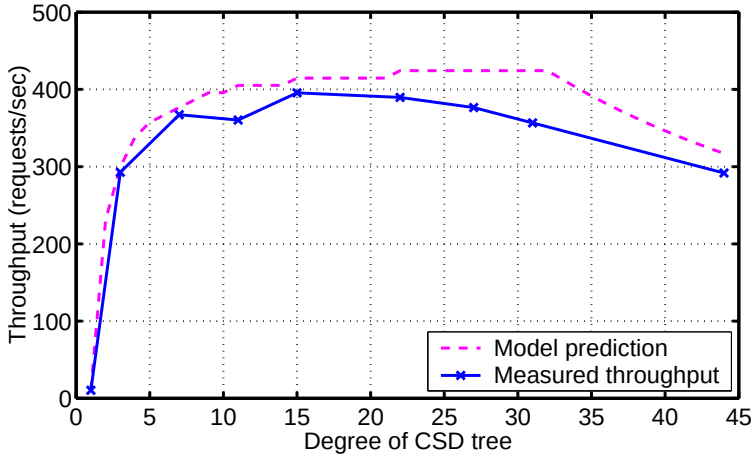


Figure 9: Predicted and measured throughput for different CSD trees for DGEMM 310 with 45 available nodes in the Sophia cluster.

Table 2 summarizes the results of these three experiments by reporting the percentage of optimal achieved for the tree selected by our model, the star, and the tri-ary tree. The table also includes data for problem size 10, for which an MA with one SeD is correctly predicted to be optimal, and problem size 1000, for which a star deployment is correctly predicted to be optimal. These last two cases represent the usage of the model in clearly SeD-limited or clearly agent-limited conditions.

DGEMM Size	Nodes $\|\mathbb{V}\|$	Optimal Degree	Selected Degree	Model Performance	Star	Tri-ary
10	21	1	1	100.0%	22.4%	50.5%
200	25	12	12	100.0%	84.4%	84.6%
200	45	3	8	87.1%	40.0%	100.0%
310	45	15	22	98.5%	73.8%	74.0%
1000	21	20	20	100.0%	100.0%	65.3%

Table 2: A summary of the percentage of optimal achieved by the deployment selected by our model, a star deployment, and a tri-ary tree deployment.

5.5 Model forecasts

In the previous section we presented experiments demonstrating that our model is able to automatically identify a deployment that is close to optimal. In this section we use our model to

forecast optimal deployments for a variety of scenarios. These forecasts can then be used to guide future deployments at a larger scale than we were able to test in these experiments. Table 3 summarizes model results for a variety of problem sizes and a variety of platform sizes for a larger cluster with the characteristics of the Lyon cluster.

DGEMM Size / $\|\mathbb{V}\|$	10			100			500			1000		
	d	$\|\mathbb{A}\|$	$\|\mathbb{S}\|$	d	$\|\mathbb{A}\|$	$\|\mathbb{S}\|$	d	$\|\mathbb{A}\|$	$\|\mathbb{S}\|$	d	$\|\mathbb{A}\|$	$\|\mathbb{S}\|$
25	1	1	1	2	11	12	24	1	24	24	1	24
50	1	1	1	2	11	12	49	1	49	49	1	49
100	1	1	1	2	11	12	50	2	98	99	1	99
200	1	1	1	2	11	12	40	5	195	199	1	199
500	1	1	1	2	11	12	15	34	466	125	4	496

Table 3: Predictions for the best degree d, number of agents used $|\mathbb{A}|$, and number of servers used $|\mathbb{S}|$ for different DGEMM problem sizes and platform sizes $|\mathbb{V}|$. The platforms are assumed to be larger clusters with the same machine and network characteristics as the Lyon cluster.

6 Conclusion and future Work

This paper has presented an approach for determining an optimal hierarchical middleware deployment for a homogeneous resource platform of a given size. The approach determines how many nodes should be used and in what hierarchical organization with the goal of maximizing steady-state throughput.

In Section 3, we presented a proof that an optimal deployment for hierarchical middleware systems on clusters is provided by a dMax CSD tree. In Section 4 we instantiated the model for the hierarchical scheduling system used in the DIET Network Enabled Server environment. We presented request performance models followed by throughput models for agents and servers. In Section 5 we presented experiments validating the DIET throughput performance models and demonstrating that our approach can effectively select an appropriate degree CSD tree for deployment.

In practice, running a throughput test on all deployments to find the optimal deployment is unmanageable; even testing a few samples is time consuming. Our model provides an efficient and effective approach for identifying a deployment that will perform well.

This article provides only the initial step for automatic middleware deployment planning. We plan to test our approach with experiments on larger clusters using a variety of problem sizes as well as a mix of applications. While our current approach depends on a predicted workload, it will be interesting to develop re-deployment approaches that can dynamically adapt the deployment to workload levels after the initial deployment. We also plan to extend our work to consider heterogeneous computation abilities for agents and test our approach on heterogeneous clusters. Our final goal is to develop deployment planning and re-deployment algorithms for middleware on heterogeneous clusters and Grids.

Acknowledgement

The authors would like to thank Frédéric Desprez and Yves Robert for their insightful ideas and Stéphane D'Alu for assistance with the Lyon cluster in Grid5000. This work has been supported by INRIA, CNRS, ENS Lyon, UCBL, Grid5000 from the French Department of Research, and the INRIA associated team I-Arthur.

References

[1] Simplifying system deployment using the Dell OpenManage Deployment Toolkit, October 2004. Dell Power Solutions.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4. UTK Computer Science Dept. Technical Report CS-01-467, 2001.

[3] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters: why and how? In *6th Workshop on Advances in Parallel and Distributed Computational Models*, 2004.

[4] J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *The International Conference on Automated Planning & Scheduling*, June 2003.

[5] E. Caron, P.K. Chouhan, and A. Legrand. Automatic deployment for hierarchical network enabled server. In *The 13th Heterogeneous Computing Workshop*, Apr. 2004.

[6] E. Caron and H. Dail. GoDIET: a tool for managing distributed hierarchies of DIET agents and servers. INRIA Research Report RR-5520, 2005.

[7] E. Caron and F. Desprez. DIET: A scalable toolbox to build network enabled servers on the Grid. *International Journal of High Performance Computing Applications*, 2005. To appear.

[8] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. Van de Geijn. Parallel implementation of BLAS: General techniques for level 3 BLAS. Technical Report CS-TR-95-40, University of Texas, Austin, Oct. 1995.

[9] S. Dandamudi and S. Ayachi. Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems. *IEEE Trans. on Computers*, 48(11):1202–1213, 1999.

[10] P. Goldsack and P. Toft. Smartfrog: a framework for configuration. In *Large Scale System Configuration Workshop*. National e-Science Centre UK, 2001. http://www.hpl.hp.com/research/smartfrog/.

[11] W. Goscinski and D. Abramson. Distributed Ant: A system to support application deployment in the Grid. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, Nov. 2004.

[12] A.W. Halderen, B.J. Overeinder, and P.M.A. Sloot. Hierarchical Resource Management in the Polder Metacomputing Initiative. *Parallel Computing*, 24:1807–1825, 1998.

[13] R.S. Hall, D. Heimbigner, and A.L. Wolf. A cooperative approach to support software deployment using the Software Dock. In *Proceedings of the International Conference on Software Engineering*, May 1999.

[14] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide area networks using AI planning techniques. In *International Parallel and Distributed Processing Symposium*, Apr. 2003.

[15] T. Kichkaylo and V. Karamcheti. Optimal resource aware deployment planning for component based distributed applications. In *The 13th High Performance Distributed Computing*, June 2004.

[16] S. Lacour, C. Pérez, and T. Priol. Deploying CORBA components on a Computational Grid: General principles and early experiments using the Globus Toolkit. In *2nd International Working Conference on Component Deployment*, May 2004.

[17] C. Martin and O. Richard. Parallel launcher for cluster of PC. In *Parallel Computing, Proceedings of the International Conference*, Sep. 2001.

[18] S. Matsuoka, H. Nakada, M. Sato, and S. Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid, 2000. Advanced Programming Models Working Group Whitepaper, Global Grid Forum.

[19] H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems*, 15(5-6):649–658, 1999.

[20] J. Santoso, G.D. van Albada, B.A.A. Nazief, and P.M.A. Sloot. Simulation of Hierarchical Job Management for Meta-Computing Systems. *International Journal of Foundations of Computer Science*, 12(5):629–643, 2001.

[21] R. Wolski, N.T. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *The Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.