International Journal of High Performance Computing Applications
http://hpc.sagepub.com

Automatic Middleware Deployment Planning On Clusters

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

International Journal of High Performance Computing Applications 2006; 20; 517
DOI: 10.1177/1094342006068404

AUTOMATIC MIDDLEWARE DEPLOYMENT PLANNING ON CLUSTERS

Pushpinder Kaur Chouhan
Holly Dail
Eddy Caron
Frédéric Vivien

ÉCOLE NORMALE SUPÉRIEURE DE LYON, FRANCE,
(EDDY.CARON@ENS-LYON.FR)

Abstract

The use of remotely distributed computing resources as a single system offers great potential for compute-intensive applications. Increasingly, users have access to hundreds or thousands of machines at once and wish to utilize those resources concurrently. To provide a reasonable user experience, such systems must provide an effective, scalable scheduling system. Unfortunately, the great majority of job schedulers are centralized and many do not scale well to thousands or even hundreds of nodes.

In this paper we study how distributed scheduling systems can be designed most effectively; we focus on the problem of selecting an optimal arrangement of schedulers, or a deployment, for hierarchically organized systems. We show that the optimal deployment is a complete spanning d-ary tree; this result conforms with results from the scheduling literature. More importantly, we present an approach for determining the optimal degree d for the tree. To test our approach, we use DIET, a middleware system that uses hierarchical schedulers. We develop detailed performance models for DIET and validate these models in a real-world environment. Finally, we demonstrate that our approach selects deployments that are near-optimal in practice.

Key words: Deployment, grid middleware, hierarchical scheduler, cluster computing

The International Journal of High Performance Computing Applications,
Volume 20, No. 4, Winter 2006, pp. 517–530
DOI: 10.1177/1094342006068404

1 Introduction

In the past five years, it has become feasible to use very large collections of distributed computing resources as a single large computational resource. In fact, this trend has become so predominant that dozens of such grids have been created. Therefore, users increasingly have access to hundreds or thousands of machines at once and are developing applications that can utilize a large amount of computational power concurrently.

However, making effective use of such large collections of machines is not simple. Users typically rely on software designed to automate many aspects of managing a distributed application on distributed resources. Schedulers are a key component of such systems; the scheduling service matches user requests with available resources and may also negotiate amongst competing users and keep track of the status and expected performance of resources.

Schedulers should ideally provide fast response time to users and good system throughput. To be usable on these newly available systems composed of hundreds or sometimes thousands of machines, schedulers must be highly scalable. Unfortunately, the great majority of existing grid schedulers are based on a centralized design and many do not provide sufficient scalability. In fact, many scheduler designers note the need for distributed scheduling approaches and suggest that they plan to work on distributed approaches in their own future work.

We are interested in whether a distributed scheduling system could provide better scalability. As a test case, we look at the Distributed Interactive Engineering Toolbox (DIET, see Caron and Desprez 2006). DIET is a network enabled server (NES) environment that provides clients simplified access to computational servers. DIET is compelling for our purposes because a distributed scheduling approach has already been incorporated into the system and is readily available in the public software toolkit. Figure 1 shows two experiments performed with DIET with the scheduler either in a distributed arrangement or in a centralized arrangement. These experiments were performed using 151 nodes of the Orsay cluster of Grid'5000 (Capello et al. 2005), a set of distributed computational resources in France. In the first test, one node is dedicated to a centralized scheduler that is used to manage scheduling for the remaining 150 nodes, which are dedicated computational nodes servicing requests. In the second test, three nodes are dedicated to scheduling and are used to manage scheduling for the remaining 148 nodes, which are dedicated to servicing computational requests. The three-node scheduler used a hierarchical arrangement with one top-level node and two children.

In this test, the centralized scheduler is able to complete 22867 requests in the allotted time of about 1400

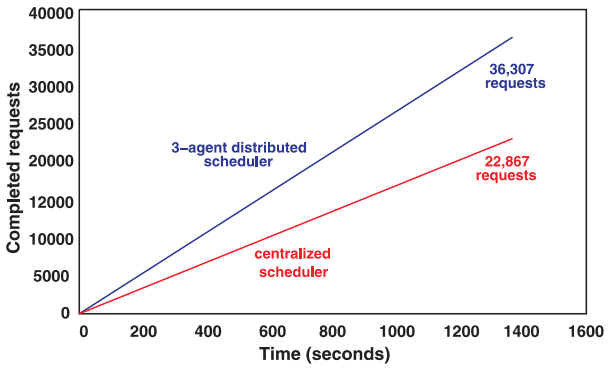

Fig. 1 Comparison of requests completed by a centralized DIET scheduler versus a three agent distributed DIET scheduler.

seconds, while the hierarchical scheduler is able to complete 36307 requests in the same amount of time. The distributed configuration performed significantly better, despite the fact that two of the computational servers are dedicated to scheduling and are not available to service computational requests. Note that the DIET scheduler is designed to allow distributed configurations, and is therefore not perfectly optimized for a centralized configuration. Although it would be interesting to compare a centralized-only scheduler as well, we nevertheless feel that these results are compelling. At least for the DIET toolkit, and most likely for other schedulers as well, distributing the task of scheduling can improve performance in large resource environments. However, the optimal arrangement of schedulers is unknown. Should we have dedicated four machines to scheduling in the above experiment? Perhaps ten? It is clearly impossible to test all possible arrangements for a given environment; we need an automated approach to determine a good deployment.

We predict that scheduler designers will increasingly be focusing on developing and using distributed scheduling solutions. However, little is known about how to find the best distributed arrangement, or *deployment*, of schedulers. While middleware designers often note that the problem of deployment planning is important, only a few algorithms exist (Kichkaylo, Ivan, and Karamcheti 2003; Caron, Chouhan, and Legrand 2004; Kichkaylo and Karamcheti 2004) for efficient and automatic deployment planning. Questions such as "which resources should be used?", "how many resources should be used?", "what arrangement should be used", and "should the fastest and best-connected resource be used for a scheduler or as a computational resource?" remain difficult to answer. Varela, Ciancarini, and Taura (2005) state the need for adaptive middleware technologies for grid environments. Technologies are needed that can select resources from the grid for a better fit to the users' expectations and that can do proper configuration of the selected resources.

The issue of deployment planning for arbitrary arrangements of distributed schedulers is too broad to address in one paper; therefore we focus on hierarchical arrangements. A hierarchy is a simple and effective distribution approach and has been chosen by a variety of middleware environments as their primary distribution approach (Halderen, Overeinder, and Sloot 1998; Dandamudi and Ayachi 1999; Santoso et al. 2001; Caron and Desprez 2006). Before trying to optimize deployment planning on arbitrary, distributed resource sets, we target a smaller sub-problem that has previously remained unsolved: what is the optimal hierarchical deployment on a cluster with hundreds to thousands of nodes? This problem is not as simple as it may sound: one must decide how many resources should be used in total, how many should be dedicated to scheduling or computation, and which hierarchical arrangement of schedulers is more effective (i.e. more schedulers near the servers, near the root agent, or a balanced approach). For instance, we have observed this problem directly in building a DIET deployment on a large Korean cluster at the School of Aerospace and Mechanical Engineering (Seoul National University) which has more than 500 CPUs (Park et al. 2004).

The goal of this paper is to provide an automated deployment planning approach that determines a good deployment for hierarchical scheduling systems in homogeneous cluster environments. We consider that a *good deployment* is one that maximizes the steady-state throughput of the system, i.e. the number of requests that can be scheduled, launched, and completed by the servers in a given time unit. In this context, this paper makes the following contributions. We first show that the optimal arrangement of agents is a complete spanning d-ary tree; this result agrees with existing results in load-balancing and routing from the scheduling and networking literature. More importantly, our approach automatically derives the optimal theoretical degree d for the tree. To test our approach, we use DIET. We develop the first detailed performance models available for scheduling and computation in the DIET system and validate these models in a real-world environment. We then present real-world experiments demonstrating that the deployments automatically derived by our approach are in practice nearly optimal and perform significantly better than other reasonable deployments. We believe that this work can be easily applied to grids composed of clusters of clusters and that this work is an important first step towards effective deployment planning on large collections of distributed, heterogeneous resources.

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 validation of optimal deployment shape 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 middleware platform 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 and can include activities such as configuring, installing, updating, and adapting a software system. Examples of tools that automate software deployment include SmartFrog (Goldsack and Toft 2001), Distributed Ant (Goscinski and Abramson 2004), and Software Dock (Hall, Heimbigner, and Wolf 1999). *System deployment* involves assembling physical hardware as well as organizing and naming whole nodes and assigning activities such as master and slave. Examples of tools that facilitate this process include the Deployment Toolkit (Daughetee and Kazim 2004), Warewulf[1], and Kadeploy (Martin and Richard 2001). 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 problem solving environment (PSE) in such a way as to maximize the number of requests that can be treated in a time unit. In Lacour, Perez, and Priol (2004), 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.

Caron, Chouhan, and Legrand (2004) 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 by Caron, Chouhan, and Legrand (2004) 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 deployments is an evolving field. Kichkaylo, Ivan, and Karamcheti (2003), proposed an algorithm called Sikitei to address the Component Placement Problem (CPP). This work leverages existing AI planning techniques and the specific characteristics of CPP. 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 (Singh et al. 2005) 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

3.1 Platform Architecture

This section defines our target platform architecture; Figure 2 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 computations, a set of server nodes $\mathbb{S}$ that are providers of computations, 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 2. 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" cannot be reduced.

We do not consider clients to be part of the hierarchy nor part of the deployment; this is because at the time of deployment we do not know where clients will be located. Thus a hierarchy can be described as follows. A

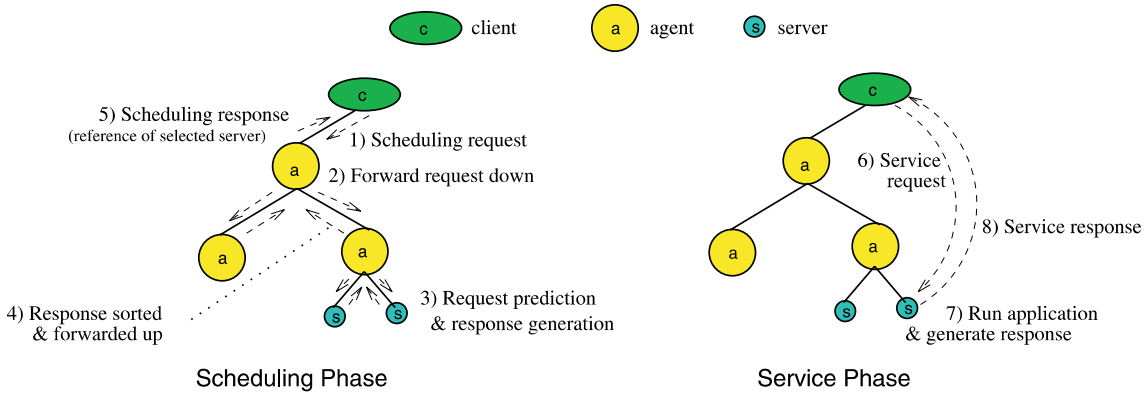

Fig. 2 Platform deployment architecture and execution phases.

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 one or more child agents and/or servers and no parents. 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* that 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 on request type. Therefore servers may or may not make predictions about performance for satisfying the request, depending on the exact system. For example, DIET agents maintain a table describing which services are available via which children, although the agent does not know exactly which SeD has the service, just which child it can use to contact SeDs with a given service. DIET agents use this information to limit the portions of the tree that must participate in scheduling any given request.

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, if any is 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. We do not consider latency in data transfer costs because our model is based on steady-state scheduling techniques (Beaumont et al. 2004). For the sake of simplicity we consider symmetric bandwidths. Usually, the latency is paid once for each of the communications that take place. In steady-state scheduling, however, as a flow of messages takes place between two nodes, the latency is paid only one time (when the flow is initially established) for the whole set of messages. Therefore latency will have an insignificant impact on our model and so we do not take it into account.

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 $|\mathbb{A}| + |\mathbb{S}| \leq |\mathbb{V}|$.

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*, i.e. 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 can produce the services required by the clients. The following lemmas lead to a proof of an optimal deployment shape of the platform.

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.

Case 1: $\rho_{sched} \geq \rho_{service}$. 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; the agent does more work forwarding incoming requests and sorting outgoing requests when it has more children. 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.

Definition 1. A **C**omplete **S**panning **d**-ary (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, with a given homogeneous resource set of size $|\mathbb{V}|$, all such CSD trees are equivalent as they provide exactly the same number

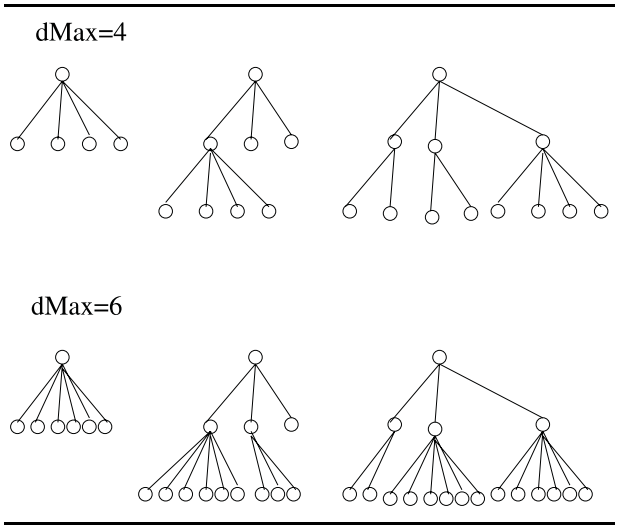

Fig. 3 Deployment trees of *dMax* sets 4 and 6.

of agents and servers, and thus provide exactly the same performance.

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

Theorem 1. *In a dMax set, all dMax CSD trees have optimal throughput.*

Proof. We know by Lemma 1 that the throughput ρ of any tree is limited by either its schedule request throughput ρ_{sched} or its service request throughput $\rho_{service}$. As Lemma 2 states that the scheduling request throughput ρ_{sched} is only limited by the throughput of the agent with the highest degree, all trees in a *dMax* set (as shown in Figure 3) have the same scheduling request throughput ρ_{sched}. Thus, to show that any *dMax* CSD tree is an optimal solution among the trees in the *dMax* set, we must prove that the service request throughput $\rho_{service}$ of any CSD tree is at least as large as the $\rho_{service}$ of any other tree in the *dMax* set.

By Lemma 3, we know that the service request 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 (leaf nodes) is largest for deployments with the smallest number of agents (internal nodes). Then, to prove the desired result we just have to show that *dMax* CSD trees have the minimal number of internal nodes among the trees in a *dMax* set.

Let us consider any optimal tree $\mathcal{T}$ in a *dMax* set. We have three cases to consider.

1. $\mathcal{T}$ is a CSD tree. As all CSD trees (in the *dMax* set) have the same number of internal nodes, all CSD trees are optimal. QED.

2. $\mathcal{T}$ is not a CSD tree but all its internal nodes have a degree equal to *dMax,* except possibly one. Then we build a CSD tree having at most as many internal nodes as $\mathcal{T}$. Using case 1, this will prove the result.

Let h be the height of $\mathcal{T}$. As $\mathcal{T}$ is not a CSD tree, from our hypothesis on the degree of its internal nodes, $\mathcal{T}$ must have a node at a height $h' < h - 1$ which has strictly less than *dMax* children. Let h' be the smallest height at which there is such a node, let $\mathcal{N}$ be such a node, and let d be the degree of $\mathcal{N}$. Then remove $dMax - d$ children from any internal node $\mathcal{N}'$ at height $h - 1$ and add them as children of $\mathcal{N}$. Note that, whatever the value of d, among $\mathcal{N}$ and $\mathcal{N}'$ there is the same number of internal nodes whose degree is strictly less than *dMax before* and *after* this transformation. Then, through this transformation, we obtained a new tree $\mathcal{T}'$ whose internal nodes all have a degree equal to *dMax,* except possibly one. Also, there is the same number of leaf nodes in $\mathcal{T}$ and $\mathcal{T}'$. Therefore, $\mathcal{T}'$ is also optimal. Furthermore, if $\mathcal{T}'$ is not a CSD tree, it has one less node at level h' whose degree is strictly less than *dMax* ; if there are zero nodes at level h' whose degree is strictly less than *dMax* after the transformation then the value to "h'" is increased. Repeating this transformation recursively we will eventually end up with a CSD tree.

3. $\mathcal{T}$ has at least two internal nodes whose degree is strictly less than *dMax.* Then, let us take two such nodes $\mathcal{N}$ and $\mathcal{N}'$ of degrees d and d'. If $d + d' \leq dMax$, then we remove all the children of $\mathcal{N}'$ and make them children of $\mathcal{N}$. Otherwise, we remove $dMax - d$ children of $\mathcal{N}'$ and make them children of $\mathcal{N}$. In either case, the number of leaf nodes in the tree before and after the transformation is non-decreasing, and the number of internal nodes whose degree is strictly less than *dMax* is (strictly) decreasing. So, as for the original tree, the new tree is optimal. Repeating this transformation recursively, we will eventually end up with a tree dealt with by case 2. Hence, we can conclude.

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.

Once an optimal degree *best_d* has been calculated using Theorem 2, we can use the algorithm shown in Figure 4 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

```
1: calculate best_d using Theorem 2
2: Add the root node
3: if best_d == 1 then
4:    add one server to root node
5:    Exit
6: availNodes = |𝕍| − 1
7: level = 0
8: while availNodes > 0 do
9:    if ∃ agent at depth level with degree 0 then
10:      toAdd = min(best_d, availNodes)
11:      add toAdd children to the agent
12:      availNodes = availNodes − toAdd
13:   else
14:      level = level + 1
15: if ∃ agent with degree 1 then
16:   remove the child
17: convert all leaf nodes to servers
```

Fig. 4 Algorithm to construct an optimal tree.

($|\mathbb{V}| = 10$). Suppose *best_d* = 1. Algorithm 4 will construct the corresponding best platform – a root agent with a single server attached. Now suppose *best_d* = 4. Then Algorithm 4 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. Note that we are not throwing away a useful resource by this operation; our calculations predict that there is no way to obtain better performance from the system by including this resource because adding it cannot simultaneously improve the scheduling and service performance.

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, such as Problem Solving Environments (PSE), to run their applications. Some PSEs, for example NetSolve (Arnold et al. 2001), Ninf (Nakada, Sato, and Sekiguchi 1999), or DIET (Caron and Desprez 2006), already exist and are commonly called Network Enabled Server (NES) environments. 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, as will be demonstrated by the experiments presented in Section 5.

This description of the agent/server architecture focuses on the common-case usage of DIET. An extension of this architecture with several hierarchies and several MA is also available (Caron et al. 2004). This multiMA architecture uses JXTA peer-to-peer technology to allow the forwarding of client requests between MAs, and thus the sharing of work between otherwise independent DIET hierarchies.

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. 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. Given the bandwidth B in Mb/second, the time in seconds required by an agent for receiving all messages associated with a request from its parent and children is shown in equation 1. Similarly, the time in seconds required by an agent for sending all messages associated with a request to its children and parent is shown in equation 2.

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

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

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 and the time in seconds required by a server for sending messages associated with a request to its parent is shown in equation 3 and equation 4 respectively.

$$server\_receive\_time = \frac{S_{rep}}{B} \quad (3)$$

$$server\_send\_time = \frac{S_{rep}}{B} \quad (4)$$

Agent computation model Agents perform two activities involving computation: the processing of incoming requests and treatment of replies. 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. Given a resource computing power w in MFlop/second, the time in seconds required by the agent for request processing is given by the following equation.

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

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

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 $(W_{pre} \cdot |\mathbb{S}| + W_{app})/w$ time in seconds to complete the $\mathbb{S}$ requests. We divide by the number of requests $|\mathbb{S}|$ to obtain the average time required per request by the servers as a group.

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

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 may be reasonable for systems with small messages as these messages are often quite CPU intensive. As shown in the following equation, 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.

$$\rho = \min\left(\frac{1}{\frac{W_{pre}}{w} + \frac{S_{req}}{B} + \frac{S_{rep}}{B}}, \frac{1}{\frac{S_{req} + d \cdot S_{rep}}{B} + \frac{d \cdot S_{req} + S_{rep}}{B} + \frac{W_{req} + W_{rep}(d)}{w}}, \frac{1}{\frac{S_{req}}{B} + \frac{S_{rep}}{B} + \frac{W_{pre} + \frac{W_{app}}{|\mathbb{S}|}}{w}} \right)$$

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 throughput can be calculated as follows.

$$\rho = \min\left(\frac{1}{\max\left(\frac{W_{pre}}{w}, \frac{S_{req}}{B}, \frac{S_{rep}}{B}\right)}, \frac{1}{\max\left(\frac{S_{req} + d \cdot S_{rep}}{B}, \frac{d \cdot S_{req} + S_{rep}}{B}, \frac{W_{req} + W_{rep}(d)}{w}\right)}, \frac{1}{\max\left(\frac{S_{req}}{B}, \frac{S_{rep}}{B}, \frac{W_{pre} + \frac{W_{app}}{|\mathbb{S}|}}{w}\right)}\right)$$

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.

5.1 Experimental Design

Software: DIET 2.0 is used for all deployed agents and servers; GoDIET (Caron, Chouhan, and Dail 2006) 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. For example, when we state that we use Dgemm 100, it signifies that we use matrix multiplication with square matrices of dimensions 100 × 100. 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 École 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 a Gigabit Ethernet. We measured network bandwidth using the Network Weather Service (Wolski, Spring, and Hayes 1999). 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 Parameterization

Table 1 presents the parameter values we used for DIET in the models for ρ_{sched} and $\rho_{service}$. Our goal was to parameterize the model using only easy-to-collect micro-benchmarks. In particular, we sought to use only values

Table 1
Parameter values

Components	W_{req} (KFlop)	W_{fix} (KFlop)	W_{sel} (KFlop)	W_{pre} (KFlop)	S_{rep} (Kb)	S_{req} (Kb)
Agent	32	10	9.6	–	6.4	5.3
SeD	–	–	–	6.4	6.4	5.3

that can be measured using a few clients executions. The alternative was to base the model on actual measurements of the maximum throughput of various system elements; while we have these measurements for DIET, we felt that the experiments required to obtain such measurements were 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[2]. 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 (Caron and Desprez 2006), 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 of our test machines in KFlops using a mini-benchmark extracted from Linpack and used this value to convert all measured times to estimates of the KFlops required.

The computational cost of the requests can be calculated either by using modeling techniques like the ones we have used above, or by using an automatic performance prediction tool such as FAST (Desprez, Quinson, and Suter 2001).

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 5. 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 5 (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. Comparison of predicted and measured throughput 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 the

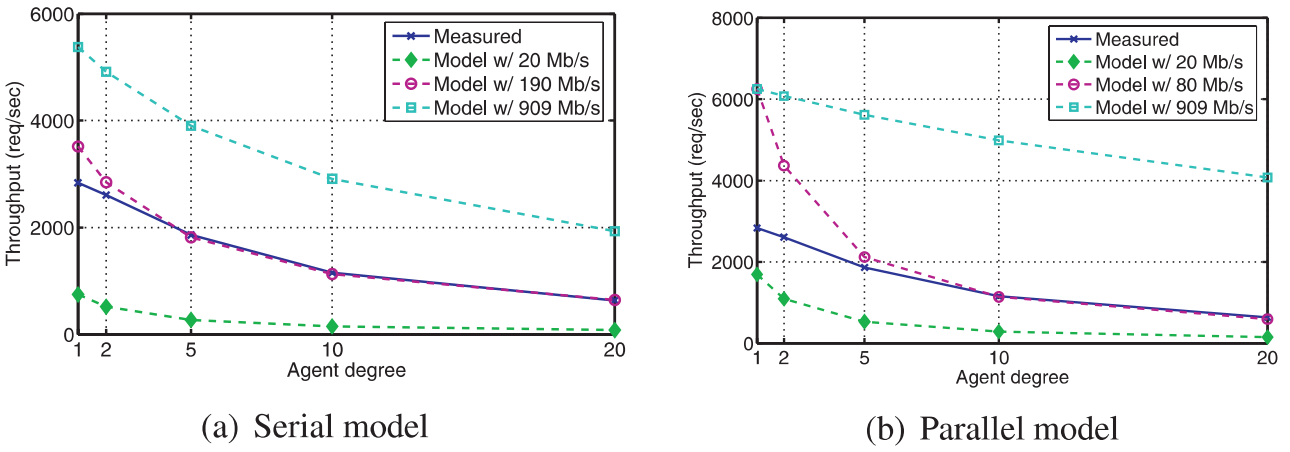

Fig. 5 Measured and predicted platform throughput for Dgemm size 10.

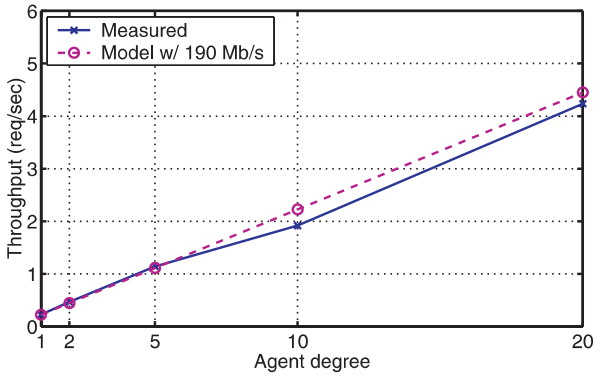

Fig. 6 Measured and predicted platform throughput for Dgemm size 1000 (serial model with a bandwidth of 190 Mb/s).

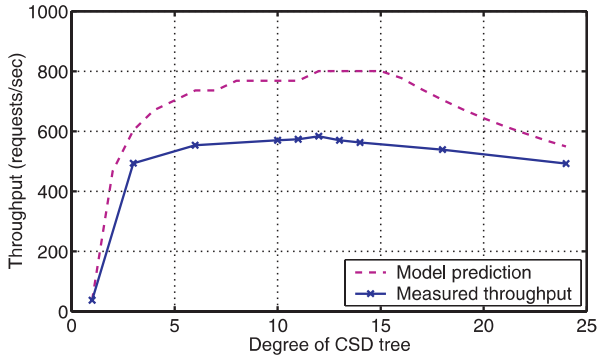

Fig. 7 Predicted and measured throughput for different CSD trees for Dgemm 200 with 25 available nodes in the Lyon cluster.

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 6. 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.

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 7 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 by the SeDs. Experiments show that the CSD tree with degree 12 does indeed provide the best throughput. 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.

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 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.

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

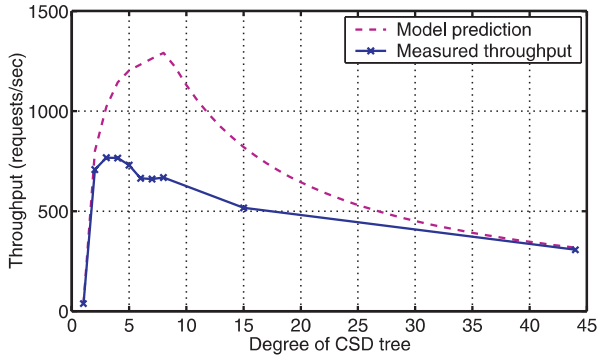

Fig. 8 Predicted and measured throughput for different CSD trees for Dgemm 200 with 45 available nodes in the Sophia cluster.

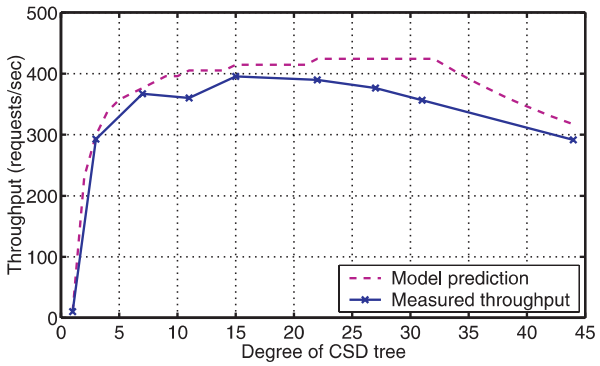

Fig. 9 Predicted and measured throughput for different CSD trees for Dgemm 310 with 45 available nodes in the Sophia cluster.

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.

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.

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.

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

Dgemm size	Nodes $\lvert\mathbb{V}\rvert$	Optimal degree	Model selected degree	Our model performance	Star performance	Tri-ary performance
10	21	1	1	100.0%	22.4%	50.5%
100	25	2	2	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 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

Dgemm size	10			100			500			1000		
$\|\mathbb{V}\|$	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

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. We presented experiments validating the DIET throughput performance models and demonstrating that our approach can effectively build a tree for deployment which is nearly optimal and which performs significantly better than other reasonable deployments.

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 grids composed of clusters of clusters and test our model on multiMA hierarchies of DIET. An obvious extension of this work would be to build, for each of the clusters, a CSD tree as explained in this paper, and then to connect these CSD trees in a hierarchical way. Our first test case will be the Grid'5000 testbed which consists of about ten large clusters composed of between 53 and 218 dual-processor nodes each. Our final goal is to develop deployment planning and re-deployment algorithms for middleware on heterogeneous clusters and Grids.

Acknowledgment

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 Grid'5000. This work has been supported by INRIA, CNRS, ENS Lyon, UCBL, Grid'5000 from the French Department of Research, and the INRIA associated team I-Arthur.

Author Biographies

Pushpinder Kaur Chouhan is a Ph.D. student at École Normale Supérieure de Lyon. She received her Masters in C.S. from ENS-Lyon, France in 2003. She did B.Tech. from SLIET, India. Her research interests include grid middleware, grid computing and problem solving environments.

Holly Dail is a member of the GRAAL research team at the ENS Lyon, France. She received her M.Sc. in Computer Science in 2002 from the University of California at San Diego and her B.Sc. in Physics and Oceanography in 1996 from the University of Washington. Her research interests focus on new approaches and algorithms for distributed computing and the application of these technologies to the domain sciences.

Eddy Caron is an assistant professor at École Normale Supérieure de Lyon and holds a position with the LIP laboratory (ENS Lyon, France). He is a member of GRAAL project and technical manager for the DIET software package. He received his Ph.D. in C.S. from University de Picardie Jules Verne in 2000. His research interests include parallel libraries for scientific computing on parallel distributed memory machines, problem solving environments, and grid computing. See http://graal.ens-lyon.fr/~ecaron for further information.

Frederic Vivien was born in 1971 in Saint-Brieuc, France. He received the Ph.D. degree from École Normale Supérieure de Lyon in 1997. From 1998 to 2002, he had been an associate professor at Louis Pasteur University of Strasbourg. He spent the year 2000 working in the

Computer Architecture Group of the MIT Laboratory for Computer Science. He is currently a full researcher from INRIA. His main research interests are scheduling techniques, parallel algorithms for clusters and grids, and automatic compilation/parallelization techniques.

Note

1 http://www.warewulf-cluster.org/cgi-bin/trac.cgi

2 http://www.ethereal.com

References

Arnold, D., Agrawal, S., Blackford, S., Dongarra, J., Miller, M., Sagi, K., Shi, Z., and Vadhiyar, S. 2001. Users' Guide to NetSolve V1.4. UTK Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN. http://www.cs.utk.edu/netsolve/.

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

Cappello, F., Caron, E., Dayde, M., Desprez, F., Jeannot, E., Jegou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., and Richard, O. 2005. Grid'5000: A large scale, reconfigurable, controlable and monitorable Grid platform. In *SC'05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing Grid'2005*, Seattle, USA, pp. 99–106.

Caron, E., Chouhan, P., and Legrand, A. 2004. Automatic deployment for hierarchical network enabled server. *The 13th Heterogeneous Computing Workshop*, Santa Fe, New Mexico, April, p. 109b (10 pages).

Caron, E., Chouhan, P. K., and Dail, H. 2006. Godiet: A deployment tool for distributed middleware on grid 5000. *EXPEGRID Workshop at HPDC2006*, Paris, June, pp. 1–8.

Caron, E. and Desprez, F. 2006. DIET: A scalable toolbox to build network enabled servers on the Grid. *International Journal of High Performance Computing Applications* 20(3): 335–352.

Caron, E., Desprez, F., Petit, F., and Tedeschi, C. 2004. Resource Localization Using Peer-To-Peer Technology for Network Enabled Servers. Research report 2004-55, Laboratoire de l'Informatique du Parallélisme (LIP), December.

Dandamudi, S. and Ayachi, S. 1999. Performance of hierarchical processor scheduling in shared-memory multiprocessor systems. *IEEE Transactions on Computers* 48(11): 1202–1213.

Daughetee, A. and Kazim, Z. 2004. Simplifying system deployment using the Dell OpenManage Deployment Toolkit. *Dell Power Solutions*, October, pp. 108–110.

Desprez, F., Quinson, M., and Suter, F. 2001. Dynamic performance forecasting for network enabled servers in an heterogeneous environment. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, June 25–28. CSREA Press.

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

Goscinski, W. and Abramson, D. 2004. Distributed Ant: A system to support application deployment in the Grid. *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, November, Pittsburgh, USA.

Halderen, A., Overeinder, B., and Sloot, P. 1998. Hierarchical resource management in the polder metacomputing initiative. *Parallel Computing* 24:1807–1825.

Hall, R. S., Heimbigner, D., and Wolf, A. L. 1999. A cooperative approach to support software deployment using the software dock. *Proceedings of the 21st International Conference on Software Engineering*, May, Los Angeles, USA, pp. 174–183. ACM Press.

Kichkaylo, T., Ivan, A., and Karamcheti, V. 2003. Constrained component deployment in wide area networks using AI planning techniques. *International Parallel and Distributed Processing Symposium*, April, Nice, France.

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

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

Martin, C. and Richard, O. 2001. Parallel launcher for cluster of PC. *Parallel Computing, Proceedings of the International Conference*, September, Naples, Italy.

Nakada, H., Sato, M., and Sekiguchi, S. 1999. Design and implementations of Ninf: towards a global computing infrastructure. *Future Generation Computing Systems* 15(5-6): 649–658.

Park, J. W., Park, S. H., Hwang, I. S., Moon, J. J., Yoon, Y., and Kim, S. J. 2004. Optimal blade system design of a new concept vtol vehicle using the departmental computing grid system. *ACM/IEEE Super Computing 2004 Conference (SC'04)*, Pittsburgh, USA, p. 36.

Santoso, J., van Albada, G., Nazief, B., and Sloot, P. 2001. Simulation of hierarchical job management for meta-Computing systems. *International Journal of Foundations of Computer Science* 12(5):629–643.

Singh, G., Deelman, E., Mehta, G., Vahi, K., Su, M.-H., Berriman, G. B., Good, J., Jacob, J. C., Katz, D. S., Lazzarini, A., Blackburn, K., and Koranda, S. 2005. The pegasus portal: web based grid computing. *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pp. 680–686, New York, NY. ACM Press.

Varela, C. A., Ciancarini, P., and Taura, K. 2005. Worldwide computing: Adaptive middleware and programming technology for dynamic Grid environments. *Scientific Programming Journal* 13(4):255–263. Guest Editorial.

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