Experiences with hierarchical request flow management for Network-Enabled Server Environments.

Holly Dail, Frédéric Desprez

Laboratoire de l'Informatique du Parallélisme

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

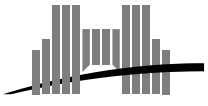
Experiences with hierarchical request flow management for Network-Enabled Server Environments

Holly Dail ,
Frédéric Desprez

February 2005

Research Report N° 2005-07

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



Experiences with hierarchical request flow management for Network-Enabled Server Environments

Holly Dail , Frédéric Desprez

February 2005

Abstract

DIET (Distributed Interactive Engineering Toolbox) is a toolbox for the construction of Network Enabled Server systems. Client requests for computational services are automatically matched with available resources by a distributed hierarchy of scheduling agents. Traditionally NES systems have used a centralized agent for performance prediction and resource information management; in DIET, these services are distributed by providing them at the DIET server level. DIET has traditionally offered an online scheduling model whereby all requests are scheduled immediately or refused. This approach can overload interactive servers in high-load conditions and does not allow adaptation of the schedule to task or data dependencies. In this article we consider an alternative model based on active management of the flow of requests throughout the system. We have added support for (1) limiting the number of concurrent requests on interactive servers, (2) server and agent-level queues, and (3) window-based scheduling algorithms whereby the request release rate to servers can be controlled and some re-arrangement of request to host mappings is possible. We present experiments demonstrating that these approaches can improve performance and that the overheads introduced are not significantly different from those of the standard DIET approach.

Keywords: scheduling, grid, window, distributed, hierarchical

Résumé

DIET (Distributed Interactive Engineering Toolbox) est une boîte à outils pour la création de systèmes type NES (*Network Enabled Server*). L'environnement proposé est capable de déterminer le serveur approprié en tenant compte de la requête de calcul. Ce service est fourni par une hiérarchie d'agents distribués. Typiquement, les systèmes NES prédisent les performances et gérent l'information sur les ressources disponibles à partir d'un agent centralisé. Dans DIET, ces services sont distribués au niveau des serveurs DIET. Actuellement, DIET utilise un modèle d'ordonnancement «on-line» où toutes les requêtes sont soit ordonnanceés soit refuseés immediatement. Avec ce modèle, les serveurs interactifs peuvent devenir surchargés et il est impossible d'échanger le placement des tâches pour prendre en compte les dépendences entre les tâches ou sur les données. Dans cet article, nous présentons un modèle alternatif basé sur la gestion actif du mouvement des tâches au travers du système. Nous avons ajouté la capacité de (1) limiter le nombre de requêtes actives sur un serveur en même temps, (2) garder les requêtes dans une file d'attente au niveau des serveurs et des agents, et (3) ordonnancer les tâches dans une fenêtre de temps, ce qui permet de contrôler le flux des tâches vers les serveurs et rend possible le redistribution du placement des tâches. Nous présentons des expériences qui montrent que ces apports peuvent ameliorér les performances de DIET et que les coûts ajoutés sont proche de celle de l'approche DIET actuel.

Mots-clés: ordonnancement, grille, fenêtres, distribué, hiérarchique

1 Introduction

The use of distributed resources available through high-speed networks has gained broad interest in recent years. Computational grids [2, 16] are now widely available for many applications around the world. The number of resources made available grows every day, and the scalability of middleware platforms is becoming a key issue. Many research projects have produced middleware to cope with the heterogeneity and dynamic nature of the target resource platforms [13, 17, 24, 27] while trying to hide the complexity of the platform as much as possible from the user.

Among middleware approaches, one simple approach consists of using servers available in different administrative domains through the classical client-server or Remote Procedure Call (RPC) paradigm. GridRPC [25] provides a standard API for this approach in grid environments. Several network enabled server (NES) environments have been developed for the grid [5, 7, 8, 21]. In these systems, clients submit computation requests to a scheduling agent whose goal is to find a server available on the grid. Scheduling is frequently applied to balance the work among the servers and a list of available servers is sent back to the client; the client is then able to send the data and the request to one of the suggested servers to solve their problem. Thanks to the growth of network bandwidth and the reduction of network latency, relatively small computation requests can now be sent to servers available on the grid. However, for this paradigm to remain a competitive choice for users, the middleware must offer sufficient performance and scalability.

While many NES environments are based on a single, centralized scheduling agent, we have found that the scalability of this approach is limited. Maintaining up-to-date information on the status of all resources is costly on a large, distributed system and it can become difficult for a single agent to manage scheduling decisions for the entire system. The Distributed Interactive Engineering Toolbox (DIET) [5] is an NES environment that focuses on solving this scalability problem. In DIET, the servers themselves are responsible for collecting and storing their own resource information and for making performance prediction. A distributed hierarchy of agents divides up the work of scheduling tasks.

DIET has traditionally used an on-line scheduling approach whereby client requests are scheduled nearly immediately and client jobs begin computing directly afterwards. This approach provides very fast response time for users, but also has several disadvantages. On interactive machines, new requests are directly executed and no limit is set on the number of requests that can execute concurrently. Thus interactive machines can become overloaded, especially for requests that require significant amounts of memory. Also, under high-load conditions scheduling immediately can lead to scheduling too far in the future; if a server suddenly becomes loaded, the requests that have already been scheduled on that server can not be rescheduled. Finally, in an on-line scenario it is impossible to consider rearranging request placements to account for inter-task data dependencies or particularly strong task-machine affinities.

In this paper, we present an extension of the DIET scheduling approach to support control over the flow of requests through various levels in the agent hierarchy. Our contributions are as follows.

- We present a detailed analysis of the algorithms used in the current DIET scheduling approach. These algorithms have not been analyzed in such detail in previous papers.

- We have added queue-like semantics to the DIET server level. With this support, the number of concurrent jobs allowed on a server can be limited. This control can greatly improve performance for resource-intensive applications where resource sharing can be very harmful to performance. Such control at the server-level is also necessary to support some distributed scheduling approaches of interest.

- We have also added queue-like semantics to the DIET Master Agent (MA) level. Under high-load conditions, incoming requests can be stalled at the Master Agent and then scheduled as a batch at an appropriate time. This window-based scheduling addition can be used to test a variety of scheduling approaches: the MA can re-order tasks to accommodate data dependencies, co-scheduling of multiple tasks on the same resource can be avoided even when the requests arrive nearly simultaneously, and inter-task dependencies can be accounted for in the scheduling process.

Experimentation with scheduling algorithms in DIET is particularly interesting due to the distributed nature of the DIET scheduling architecture. While other window-based scheduling approaches exist, the

algorithm presented in this paper is adapted for the limited information available at the global level in a distributed scheduling system.

This paper is organized as follows. The next section gives an overview of related work in hierarchical scheduling for networks of workstations, SMP machines, clusters, and grids. Section 3 presents the architecture of DIET and Section 4 describes in detail the current scheduling and performance estimation approaches used in DIET. Section 5 presents the extensions we added to support the control of the flow of tasks at various levels of the hierarchy and modification of task placement at the Master Agent level. Section 6 describes experiments comparing the performance of our extensions to the performance of the standard DIET approach. Finally, Section 7 gives a conclusion and a discussion of future work.

2 Related work

Scheduling systems for single clusters of workstation and supercomputers have been studied extensively in the literature [14, 15, 29].

Hierarchical approaches have also been extensively studied, including work in fields outside of classical job scheduling. For example, hierarchical scheduling systems for distributed WWW servers are discussed in [1]. In this paper, the authors seek to load-balance HTTP requests across a cluster and multiple clusters. The scheduler can choose between moving data to available hosts or placing computational workload on the hosts where the data reside. Example applications can be data-intensive image access applications as well as compute-intensive interactive image zooming and resolution enhancement. However, these approaches are limited to a 2-level cluster hierarchy. A hierarchical disk scheduler for multimedia systems is presented in [6]. This work finds that benefits can be obtained with a 2-level scheduling scheme for disk I/O. The system uses queues for different request types (deterministic, statistical, and best-effort). In [20], a decentralized service discovery system is discussed for global computing grids. This system seeks higher scalability and improved fault-tolerance by using a hierarchy of brokers with node replication. This work presents advanced approaches for service discovery but is not concerned with scheduling tasks; the work is thus complementary to ours.

Shared-memory systems can also benefit from hierarchical scheduling [11]. Additional heuristics for both shared and distributed memory machines are given in the following book [12]. These heuristics are evaluated in simulation, although the idealized parameters used might not reflect real conditions on actual clusters.

Simulation is also used for several hierarchical job scheduling algorithms in [23]. In this paper, the authors compare several heuristics like First Come First Served (FCFS) and Shortest Job First (SJF) at the Global Resource Manager (GRM) level. It is shown that careful resource management at the global level can improve performance of the platform as a whole. In [18], other algorithms are also tested by carefully studying the cost of the parallel jobs sent to the schedulers.

Hierarchical scheduling is applied to the grid in [19]. The authors implemented a distributed approach to scheduling using Global Resource Managers connecting Local Resource Managers (LRM). The biggest difference with our work is that they have no control over the scheduling of jobs at the LRM level because these are classical batch systems. One interesting feature is that the GRMs are replicated for better fault-tolerance. A similar approach is used in [26] but simultaneous requests are sent to every global scheduler. As soon as one request finishes at one site, the others are cancelled. This improves the average job slowdown and turn-around time.

3 DIET Architecture

The DIET architecture is based on a hierarchical approach to provide scalability. The architecture is flexible and can be adapted to diverse environments including heterogeneous network hierarchies. DIET is implemented in CORBA and thus benefits from the many standardized, stable services provided by freely-available CORBA implementations offering good performance.

DIET is based on several components. A **Client** is an application that uses DIET to solve problems using an RPC approach. Users can access DIET via different kinds of client interfaces: web portals, PSEs

such as Matlab or Scilab, or from programs written in C or C++. A **SeD**, or server daemon, provides the interface to computational servers and can offer any number of application specific computational services. A SeD can serve as the interface and execution mechanism for a stand-alone interactive machine, or it can serve as the interface to a parallel supercomputer by providing submission services to a batch scheduler.

Agents provide higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single **Master Agent (MA)** and several **Local Agents (LA)**. Figure 1 shows an example of a DIET hierarchy.

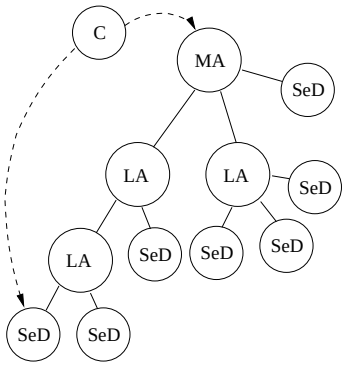


Figure 1: DIET hierarchical organization.

A **Master Agent** is the entry point of our environment. In order to access DIET scheduling services, clients only need a string-based name for the MA (e.g. "MA1") they wish to access; this MA name is matched with a CORBA identifier object via a standard CORBA naming service. Clients submit requests for a specific computational service to the MA. The MA then forwards the request in the DIET hierarchy and the child agents, if any exist, forward the request onwards until the request reaches the SeDs. The SeDs then evaluate their own capacity to perform the requested service; capacity can be measured in a variety of ways including an application-specific performance prediction, general server load, or local availability of data-sets specifically needed by the application. The SeDs forward their responses back up the agent hierarchy. The agents perform a distributed collation and reduction of server responses until finally the MA returns to the client a list of possible server choices sorted in order of desirability. The client program is composed to two pieces: those portions that encapsulate behaviors that are common between all DIET clients and that are hidden from the application developer and those parts that are application-specific that are written by the application developer. The general-purpose DIET client code receives the list of proposed servers and manages the submission of the request to one of the servers. Generally the first server is selected, but the others can be useful in cases of problems with the functionality or performance with the first choice.

There are two broad issues of scheduling that arise for a distributed architecture like that of DIET: deployment and task scheduling. It is important to select an appropriate deployment of the architecture on the grid in terms of the number of agents, the depth and breadth of the agent tree, and the number and placement of servers providing different types of computational services. DIET components can be arranged in anything from a tiny architecture of a single MA with a single SeD offering one service, up to several thousands of servers connected by a deep hierarchy of dozens of agents.

Indeed, depending of the speed of networks connecting the different components and the computing capacity of the machines hosting the agents and servers, bottlenecks can appear that will lower the overall throughput of the platform. In [4], the authors present an approach to discover agent bottlenecks using a linear program based on constraints of communication and computation costs induced by client requests. The bottlenecks are iteratively discovered and removed using a greedy algorithm until the platform throughput is constrained by server performance.

In this paper we address the second issue: given a particular deployment, what are appropriate algorithms for scheduling client requests onto available servers. In the next section we discuss the design of the DIET scheduling system currently in use.

4 DIET Scheduling

The primary interest of the DIET scheduling approach lies in its distribution, both in terms of collaborative decision making and in terms of the information important to the scheduling decision. We return to the general process of servicing a request to provide greater details. When the MA receives a client request, it (1) verifies that the service requested exists in the hierarchy, (2) collects a list of its children that are thought to offer the service, and (3) forwards the request to those subtrees. Local agents use the same approach for forwarding the request to their children, whether the children are other agents or SeDs. Agents obtain information on services available in sub-trees during the deployment process. The DIET deployment process is top-down: higher-levels in the hierarchy must be launched before subtrees can be launched. When a SeD or agent starts up, it joins the DIET hierarchy by contacting its parent agent (located by a string-based name in a naming service). The parent adds the new child to its list of children and records which services are available via that child. The parent need not track whether the service is provided directly by the child (if the child is a server) or by another server in the child's subtree (if the child is an agent); it suffices to know the service is available *via* the child. Thus if an agent has N children and the DIET hierarchy offers a total of M services, the most hierarchy information any agent in the tree will store is N*M service/child mappings.

When an agent forwards a request to its children, it sets a timer restricting the amount of time to wait for child responses. This avoids a deadlock in the hierarchy based on one failed or slow-to-respond server. Eventually, a child will be forgotten if it is unresponsive for long enough.

SeDs are responsible for collecting and storing all of their own performance and status data. Specifically, the SeD stores a list of problems that can be solved on it, a list of any persistent data that are available locally to the server, and status information such as the number of requests currently running on the SeD and the amount of time elapsed since the last request. When a request arrives at a SeD, the SeD creates a response object containing both status information and performance data. SeDs are capable of collecting dynamic system availability metrics from the Network Weather Service (NWS) [28] or can provide application-specific performance predictions using the performance evaluation module FAST [22]. The availability of dynamic performance data on a particular SeD depends on whether FAST and/or NWS has been deployed on the SeD's machine. When NWS is available, DIET SeDs have access to dynamic resource metrics such as available CPU, free memory, or available bandwidth between two hosts. When FAST is available, DIET SeDs have access to performance predictions for certain problems; these predictions are made based on the interpolation of results from a benchmark database for the same problem. The benchmark database is automatically created at FAST install-time for problems of interest.

For some applications, a suite of automatic macro-benchmarks can not adequately capture application performance. In these cases, DIET also allows the server developer to specify an application-specific performance model to be used by the SeD during scheduling to predict performance.

After the SeDs have formulated a response to the request, they send their response to their parent agent. Each agent is responsible for aggregating the responses of its children and forwarding on a sorted list of responses to the next-higher level in the hierarchy. DIET normally returns to the client program multiple server choices, sorted in order of the predicted desirability of the servers. The number N of servers to return to the client is configurable, but is of course limited by the total number of servers managed by the DIET hierarchy. Since agents have no global knowledge of the DIET hierarchy, to ensure a full list can be returned to the client each agent must return a sorted list of its N best child responses (or less if the agent subtree contains less than N servers).

The agent sorting process uses an efficient binary tree with each child node placed as the leaves. In the case of a server child, the leaf node in the sorting tree consists of just one response. In the case of an agent child, the leaf node consists of an already sorted list of servers available in that child's sub-hierarchy. For small values of N, the sorting overhead incurred by an agent is thus more strongly related to the number of direct children the node has than to the number of SeDs included in the deep sub-hierarchy below the agent. Increasing the number of children an agent has increases the agent's sorting time, while increasing the depth of the agent hierarchy increases the communication latency incurred during the hierarchical decision process. Thus, there is an important tradeoff between sorting time and request latency based on a balance between agent node out-degree and tree depth.

While the agent aggregation routines are designed to select the best servers for a problem, it is in fact

even more important that they ensure a decision is always made. The sorting approach thus relies on a series of comparison options where each comparison level utilizes a different type of SeD-provided data. In this way, the agent hierarchy does not become deadlocked simply because, for example, some of the SeDs do not have the capability to provide an application-specific performance prediction. In fact, for system stability, any agent-level sorting routine should rely on a final random selection option to provide a last-resort option for choosing between servers.

For this paper, we have expanded the ability of the SeDs to collect self-maintained status data (such as monitoring the number of jobs currently running) and have incorporated these data in the agent-level sorting routines. Figure 1 provides a high-level view of the resulting sorting algorithm; for brevity we do not include the binary-tree based sorting algorithm and we include only the most important and frequently used comparison metrics. At each call to the `ExtractBest` routine, up to N of the child $responses$ that can be directly compared on the given metric are added to $best$ servers list. Those servers that can not be compared on the given metric are left in the $responses$ list to be sorted based on one of the lower priority metrics.

Algorithm 1 Agent-level sorting algorithm

procedure SelectBestNServers($responses, N$)
  $numResp \leftarrow Size(responses)$
  $best \leftarrow$ ExtractBest ($responses$, N, execTime) // `Minimize predicted execution time`
  if Size($best$) $== N$ **or** Size($responses$) $== 0$ **then**
    Return $best$
  end if
  $n \leftarrow N$ - Size($best$)
  Append($best$, ExtractBest($responses$, n, numJobs)) // `Minimize number of active jobs`
  if Size($best$) $== N$ **or** Size($responses$) $== 0$ **then**
    Return $best$
  end if
  $n \leftarrow N$ - Size($best$)
  Append($best$, ExtractBest($responses$, n, lastSolve)) // `Maximize time since last solve`
  if Size($best$) $== 0$ **or** Size($responses$) $== 0$ **then**
    Return $best$
  end if
  $n \leftarrow N$ - Size($best$)
  Append($best$, ExtractBest ($responses$, n, random)) // `Guarantee a choice with random`
  Return $best$

The use of on-line scheduling in the current version of DIET has several advantages. Requests issued by clients are scheduled as quickly as possible upon their arrival and scheduling latencies are very low. The gathering of performance information from servers is only done upon request contrary to other NES systems like NetSolve or Ninf [21]. This avoids unnecessary communications when few requests are sent to the agent. Also, performance prediction is normally a process of collecting a variety of performance data and providing a single performance estimate; in DIET the bulkier detailed performance data generally rests at the server level and only the synthesized performance estimate is passed up the tree.

There are, however, conditions in which an on-line strategy can be less desirable. For example, DIET currently continues to launch new executions even if the target server is already fully utilized. For jobs with intensive resource requirements this can lead to reduced throughput; for memory-intensive jobs the result can be dismal performance if disk access is required to manage the multiple competing jobs (e.g. swapping). Also, DIET's on-line strategy is not able to support any re-ordering of requests for satisfaction of priorities, data requirements, or task dependencies.

DIET could also be an intriguing platform for the development of novel distributed scheduling algorithms and as a platform for real-world tests of theoretical scheduling algorithms or heuristics that have been tested only in simulation. However, due to the on-line approach used by the system, the current system is limited to experimentation with new performance metrics and various approaches for aggregating

responses at the agent level.

In the following section we discuss extensions to the DIET scheduling approaches designed to provide greater control over task placement.

5 Scheduling extensions

Our extensions to the DIET scheduling system are designed to limit the flow of requests in the DIET architecture. Our primary goals for extending the DIET scheduling system in this paper are as follows.

- Enable control over the number of requests actively using resources at the server level. This goal is addressed in Section 5.1 with the addition of lightweight SeD-level queues.
- Enable control over how far in advance requests are allocated to specific servers. The goal is to be able to slow down the flow of requests to the servers in high load conditions, while maintaining on-line or almost on-line behavior in low-load conditions. This goal is addressed in Section 5.2 with the addition of an MA-level queue and algorithms controlling the timing of task releases in *windows* by the MA.
- Enable global-level modifications of task assignments. This goal is addressed in Section 5.2 with data sharing between request threads and a special task scheduling thread. The task scheduling thread can re-arrange task assignments during each scheduling window.

To the extent possible, we also want the extensions to be usable independently to provide greater flexibility. A variety of configurations are tested in experimentation in Section 6 demonstrating this feature. Finally, we wish to limit the effects of these changes on the stability and low-overhead characteristics of the standard DIET scheduling approach. These issues are addressed in Section 6 with a number of experiments designed to compare the performance of the standard DIET approach against the performance of the extensions introduced here.

5.1 SeD-level limit on concurrent jobs

The first extension to the DIET scheduling model allows one to control the number of concurrent requests that can be executing at once at the SeD level. As a simple first approach we do not attempt to keep extra jobs from reaching the SeD. Instead, once solve requests reach the SeD we place their threads in what we will call a **SeD-level queue**. In fact, to keep overheads low we implement a very lightweight approach that offers some, but not all, of the semantics of a full queue. We add a counting semaphore to the SeD and initialize the semaphore with a user-configurable value defining the desired limit on the number of requests that can concurrently use resources such as the CPU. When each request finishes its computational work, it calls a post on the counting semaphore to allow another request to begin computing. The order in which processes will be woken up while waiting on a semaphore is not guaranteed on many systems; therefore we augmented the semaphore to ensure that threads are released in the appropriate order.

To support consideration of queue effects in the scheduling process, we use a number of approaches for tracking queue statistics. It is not possible to have complete information on all the jobs in the “queue” without adding significant overhead for coordinating the storage of queue data between all requests. Thus we approximate queue statistics by storing the number of jobs waiting in the queue and the sum of the predicted execution times for all waiting jobs. Once jobs begin executing we individually store and track the jobs’ predicted completion times. By combining these data metrics and taking into account the number of physical processors and the user-defined limit on concurrent solves, we can provide a rough estimate of when a new job would begin execution. This estimate is included by the SeD with the other performance estimates passed up the hierarchy during a schedule request.

There are some disadvantages to this method of controlling request flow. Most importantly, requests are in fact resident on the server while they wait for permission to begin their solve phase. Thus, if the parameters of the problem sent in the solve phase include large data sets, memory-usage conflicts could arise between the running jobs and the waiting requests. Some DIET applications with very large data sets use a different approach for transferring their data where only the file location is sent in the problem

parameters and the data is retrieved at the beginning of the solve. The impact of this problem will therefore depend on the data approach used by the application.

A second problem with this approach arises from the fact that once requests are allocated to a particular server, DIET does not currently support movement of the request to a different server. When system conditions change, although the jobs have not begun executing, DIET can not adjust the placement to adapt to the new situation. Thus performance will suffer in cases of unexpected competing load or poorly predicted job execution time. Also, in the case of improvements in the system, such as the dynamic addition of server resources, DIET can not take advantage of the resources for those tasks already allocated to servers.

5.2 MA-level task flow control

In this extension to DIET, we implement several additions at the Master Agent level to support request flow control and task assignment re-ordering. In the standard DIET system, requests are each assigned an independent thread in the Master Agent process and that thread persists until the request has been forwarded in the DIET hierarchy, the response received, and the final response forwarded on to the user. In this approach, the only data object shared among threads is a counter that is used to assign a unique request ID to every request.

Algorithm 2 Global task flow management algorithm.

procedure $GlobalTaskManager(minWinTime, maxWinSize, winAdaptFactor, taskLatency)$

$windowTime \leftarrow minWinTime$
while $true$ **do**
  $numWaiting \leftarrow$ GetWaitingTaskCount
  if $numWaiting$ == 0 **then**
    $windowTime \leftarrow \frac{windowTime}{2}$
    if $windowTime < minWinTime$ **then**
      $windowTime \leftarrow minWinTime$
    end if
  else
    $windowSize \leftarrow$ Minimum ($numWaiting$, $maxWinSize$),
    $requests \leftarrow$ GetFirstNRequests ($windowSize$)
    ReleaseRequestsToHierarchy ($requests$)
    $responses \leftarrow$ WaitNResponsesFromHierarchy ($windowSize$)
    ($finalResponses$,$minQueueWait$) $\leftarrow$ RefinePlacement ($responses$)
    ReleaseResponsesToClient ($finalRespones$)
    if $minQueueWait < taskLatency$ **then**
      $windowTime \leftarrow taskLatency$
    else
      $windowTime \leftarrow winAdaptFactor \times minQueueWait$
    end if
  end if
  Sleep ($windowTime$)
end while

In the modified Master Agent, each request is still assigned a thread that persists until the response has been sent back to the client. However, we introduce one additional thread that provides higher-level management of request flow; the algorithm for this GlobalTaskManager is given in Algorithm 2. Scheduling proceeds in distinct phases called windows and both the number of requests scheduled in a window and the time interval spent between windows are configurable. An interesting aspect of this algorithm is that the Master Agent can only discern characteristics of the DIET hierarchy, such as server availabil-

ity, by forwarding a request in the hierarchy. We avoid sending any task twice in the hierarchy, thus the GlobalTaskManager must schedule some jobs in order to have information about server loads and queue lengths.

The maximum size of the scheduling window ($maxWinSize$) derives from the distributed nature of DIET: it should be equal to the number of responses $maxResponses$ forwarded by the aggregation routines in the DIET agents (this is itself a configurable property of DIET). The $minWinTime$ is the minimum amount of time to sleep between windows and is introduced to avoid busy wait behavior when request volume is low and the GlobalTaskManager runs in an on-line mode (in our experiences a value of 5 msec is effective). The window time is re-calculated after each scheduling window with the following goals: when request volume is low, the window should be very short to provide fast response time; when the servers become loaded and jobs are already waiting in the SeD-level queues, the window should be longer to avoid scheduling too far into the future; and when jobs have just been scheduled, the window should be long enough to allow the clients of the just scheduled requests to submit the requests to the servers. This last point is a problem for the standard DIET approach as well: the time required to schedule many DIET tasks is much shorter than the time required for the scheduling decision to take effect (that is, the time required for the client to receive the response and start the solve on the selected server). Thus, multiple jobs can be assigned the same server before the server reports that a new job has been launched. For the GlobalTaskManager algorithm, we introduce the parameter $taskLatency$ as the minimum window time to use after some tasks are scheduled; in our experiences a value of one second was reasonable, however this value is certainly platform dependent and additional experiences will be needed to gain intuition as to the correct value.

Algorithm 3 Global schedule refinement algorithm

procedure $RefinePlacement(responses)$
  $numTasks \leftarrow responses \rightarrow GetSize$
  $resp \leftarrow responses[1]$
  $finalPlacements[1] \leftarrow resp.servers[1]$
  for $i \leftarrow 2, 3, \ldots, numTasks$ **do**
    $resp \leftarrow responses[i]$
    for all $server \in resp.servers$ **do**
      if $server \notin finalPlacements$ **then**
        $finalPlacements[i] \leftarrow server$
      end if
    end for
  end for
  Return ($finalPlacements$, $minQueueWait$)

The RefinePlacement call in Algorithm 2 provides the opportunity to experiment with refining the placement of tasks to servers. DIET is a multi-user system and thus we are concerned with providing fairness to users; we consider that job placements can only be re-arranged within the scheduling window and the window can be made smaller to improve fairness or larger to improve opportunities for performance gains. Algorithm 3 provides a simple RefinePlacement approach; this example avoids placing tasks in the same window on the same host, if possible. While in this paper we focus on a detailed description of the current DIET scheduling approach and on the practical aspects of our extensions, in future work we plan to investigate more sophisticated RefinePlacement algorithms incorporating, for example, data dependencies and task inter-dependencies.

6 Experiments

6.1 General configuration

6.1.1 Testbed

To test our algorithms, we performed experiments on the Grid'5000 testbed [30]. Specifically, we used a 56-node cluster located at the École normale superiéure de Lyon (ENS Lyon). Each node in this cluster has 2 GB of memory, dual 2 GHz AMD Opteron processors, and 1 MB of cache. The cluster has two networks and each node has two Ethernet connections: a 100 Mbit connection for administration and a 1 Gbit connection for computational applications. Since the faster network connection is the second, non-default network interface, we were careful to explicitly configure all software to use the appropriate interface.

The ENS-Lyon Grid'5000 cluster used for these experiments had just been installed at the time of these experiments and was not yet in common use by other users. Thus although we did not have explicit dedicated access, all observations lead to the conclusion that, in practice, we did in fact have dedicated use of the machine. Dedicated access provides us the opportunity to study the performance of DIET and our scheduling approaches in a detailed, controlled, and reproducible manner on a real system. These types of experiments are not often done by middleware designers and thus detailed, comparable results on middleware performance are rare. However, these experiments are also limited in scope and do not adequately test issues of robustness to heterogeneity, load variability, or failures. In future work we plan to complement these controlled, detailed experiments with case studies in distributed, heterogeneous environments to study these important issues.

6.1.2 User model

To test system performance in a controlled environment, we define a usage scenario for each experiment and then implement the scenario by scripts that submit jobs following the pattern defined by the given usage scenario. We define two general usage styles: sequential users and batch users.

In the `sequential user` model, each user has a number of tasks to run sequentially. This model emulates users who use the results of previous runs to select parameters for their next run; steering an interactive instrument such as a large telescope would create a workload of this type. Since DIET is a multi-user system, we test varying system loads by varying the number of sequential users that are performing this style of interactive usage at once. In the `batch user` model, we consider users that have a number of tasks to submit all at once and who are more interested in the completion of the group rather than each individual task. Parameter sweep applications typically create this type of workload.

6.1.3 Scheduling Approaches

We compare three scheduling approaches in order to study performance differences between the standard DIET approach and our extensions. The **Standard** approach is the regular DIET on-line scheduling approach. The **SeD Queue** approach uses the SeD Queue extension at the SeD-level, but does not alter MA-level scheduling from the `Standard` approach. The **Task Scheduler** approach uses both the SeD-level queue extensions and the MA-level window-based task scheduling approach. For these experiments, the number of requests allowed to actively use resources at once at the SeD was limited to two for both the `SeD Queue` and the `Task Scheduler` approaches.

6.1.4 Application model

As an initial study, we use a simple matrix application with varying problem sizes. Specifically, we use a BLAS DIET server that internally uses the `dgemm` library function to solve problems of type $C = \alpha C + \beta AB$. A matrix operation of this sort is so computationally simple relative to the communication required that there are few conditions in practice under which it would be run remotely (i.e. on the grid). A more reasonable scenario for grid execution is one in which many different matrix operations are performed remotely for each communication of the matrices; this is, in fact, the behavior of a broad variety of scientific

applications. We thus modified the DIET BLAS server to perform 10 iterations of the matrix operations; this provides a more realistic user scenario while allowing us to maintain a simple application as an initial case study.

6.2 Performance for bursty request arrival

In this experiment we study the performance of each scheduling strategy in an interactive scenario: a user that quickly submits a group of tasks and who may be interested in the turnaround time for each task and/or the total makespan of time required to complete all of the tasks. A challenging aspect of this type of scenario is that in real-world situations there is no control over client behavior; clients may send any sized burst of tasks with any sort of send pattern. Thus we focus in particular on the ability of each strategy to perform well under a variety of user submission scenarios.

6.2.1 Experimental design

We use a DIET hierarchy with one Master Agent, two Local Agents, and eight servers (four attached to each Local Agent). We model a user scenario in which the user has a group of eight tasks to run and submits them all as a group to DIET; all requests were for dgemm with square matrices of size 1500x1500. In real-world situations, the user or user script may require some amount of time between submissions to prepare data or calculate input to the request; thus we test task inter-arrival times of 0, 0.25, 0.5, 0.75 and 1 second. We chose one second as the upper limit after some exploratory testing indicated that, for our particular test system, scheduler behavior for inter-arrival times larger than one second match those of one second.

Given a particular inter-arrival period and a particular scheduling approach, we performed the following test: on a DIET system with no other activity, use a `user`-emulation script to launch eight tasks in succession with a sleep of the appropriate period between each submission. For each such group of tasks, we measured the mean request turnaround time and the makespan for completion of all eight tasks. This test was repeated five times for each configuration of period and scheduling approach.

6.2.2 Results

Figure 2 shows the mean and standard deviation of the makespan (Figure 2a) and the task turnaround time (Figure 2b). The most important result shown in these figures is that the `Task Scheduler` approach provides consistently good performance regardless of the rate at which the user sends tasks. The `Standard` and `SeD Queue` approaches reach similar performance levels for an inter-arrival time of one second, but are unable to provide good performance for other inter-arrival times. While we have control over the "user" behavior and can configure the send rate for these experiments, normally users will send at a variety of different rates. Furthermore, even if users were willing to send requests at a particular rate, the appropriate rate will vary greatly with system characteristics and with the DIET deployment. Thus in this scenario the `Task Scheduler` approach is preferable as it provides good, stable performance regardless of user behavior.

In comparing the performance of the `Standard` and `SeD Queue` approaches it is interesting to note that the `Standard` approach typically provides a better makespan but the `SeD Queue` approach typically provides a better task turnaround time. This result is due to the fact that we have dual-processor machines and a small number of tasks. Figure 3 provides a small example explaining this effect.

6.3 Performance under changing resource conditions

In the previous section we tested the robustness of each scheduling approach to task inter-arrival times. We found that in our test environment, a task inter-arrival time of at least one second was sufficiently long for all three strategies to receive up-to-date information on previously scheduled jobs and therefore to make reasonable scheduling decisions. However, there are many situations in which the performance of the three strategies will vary despite a perfectly-chosen inter-arrival time. In this section, we examine a situation

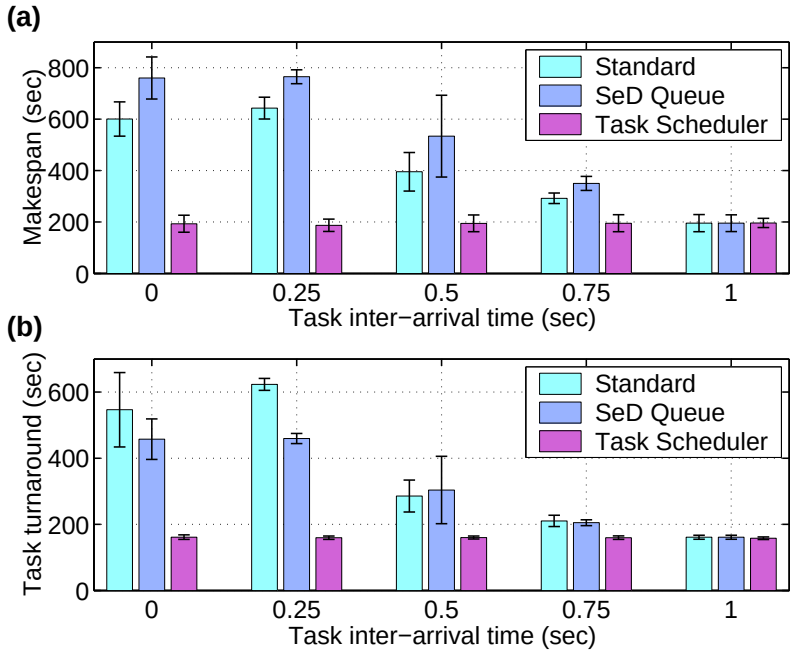


Figure 2: The effect of client send behavior on scheduler performance as shown by the (a) makespan for completion of all eight tasks and (b) the mean task turnaround time over the eight tasks. The x-axis in the graphs gives the time in seconds introduced in the user script between each request submission.

where resource conditions change mid-execution to test whether the three strategies are able to react to the change.

6.3.1 Experimental design

As for the last section, we use a base DIET hierarchy with one MA, two LAs, and eight servers (four attached to each LA). We also use the same user model as for the last section, except we now model a user who has a group of 54 tasks to run in total where all requests are for dgemm with square matrices of size 1200x1200. To remove the effect of task inter-arrival times on performance, we use the results of the last section and introduce a fixed sleep of one second between each task submission.

The tests were run as followed: first all 54 clients were launched at a rate of one per second, then a pause of ten seconds was taken, and then four additional SeDs were attached to the DIET hierarchy (two on each LA). After all 54 clients requests finished, we calculated the makespan (based on the time of launch for the first client to the completion of the last request in the system) and the mean turnaround time.

6.3.2 Results

Table 1 provides the results of these experiments. The `Task Scheduler` approach achieves the best performance in this scenario because it is able to take advantage of the newly added SeDs. In this approach requests are not scheduled as far in advance and thus scheduling choices for those “stalled” tasks can be adapted to changing resource conditions. This approach scheduled 5-6 requests on each of the 8 original SeDs and 3 requests on each of the 4 newly added SeDs.

Neither the `SeD Queue` nor the `Standard` strategies were able to place requests on the newly added SeDs in these conditions as the schedule for all 54 requests had already been defined at the time the new SeDs were added. The difference in performance between the two strategies is due to a difference in the load-balance achieved between the 8 original SeDs. Specifically, the `Standard` approach scheduled 9

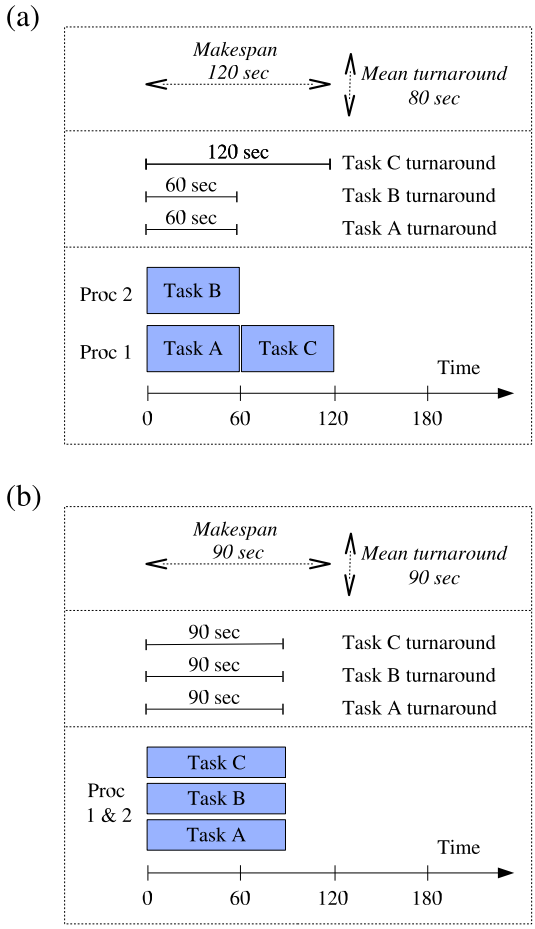


Figure 3: Comparison of makespan and request turnaround time calculations when (a) the number of concurrent jobs on a machine are limited and (b) when all processes are run concurrently and standard unix timesliced scheduling is used to share the two processors amongst all jobs.

	Makespan (seconds)	Task Turnaround (seconds)
Standard	460.1	254.7 ± 61.7
SeD Queue	369.1	179.2 ± 76.7
Task Scheduler	312.7	147.6 ± 52.7

Table 1: Summary of experiments testing ability of each strategy to react to the addition of SeDs mid-execution.

requests on one SeD, 7 requests on 3 SeDs, and 6 requests on 4 SeDs while the `SeD Queue` approach scheduled 7 requests on 6 SeDs and 6 requests on 2 SeDs.

6.4 Steady-state performance studies

The previous section described execution environments where the `SeD Queue` and `Task Scheduler` strategies are effective. It is difficult to differentiate the overhead introduced by a strategy as compared to the performance difference due to the scheduling strategy itself. In this section, we present experiments designed to reveal any overheads or other performance problems introduced by our extensions. The experiments use a steady, long-running, heavy request volume of uniform tasks, homogeneous resources, and no competing load. Assuming that all three strategies can keep all servers continuously busy, there can be no performance benefit due to the scheduling strategies themselves. Instead, any differences in performance can be attributed to overheads or inability to keep all servers busy.

6.4.1 Experimental design

For these experiments we use a DIET architecture with 1 MA, 4 LAs, and 24 servers; 6 servers are attached to each LA. Each server and agent was provided a dedicated machine.

We use a `sequential user` model with 96 concurrent users of DIET, thus on average there are four `users` per server or two per processor. Each `user` is emulated with a script that submits one request at a time in a continuous loop; all requests were for dgemm with square matrices of size 1000x1000. Since we had a limited number of resources available for these experiments, we placed 6 `users` per machine. All requests were for `dgemm` with square matrices of size 1000 in each dimension. Since we had a limited number of resources available for these experiments, we placed 6 `users` per machine.

The `user` scripts were launched at a rate of one every five seconds until all 96 were in place (about eight minutes). Three hours later the `user` scripts were stopped. For the calculation of statistics, tasks are considered only if they entered the system more than 60 minutes after the start of the first `user` script and more than 15 minutes before the scripts were stopped.

6.4.2 Request handling

Figure 4 shows a five-minute snapshot of task execution for each strategy during the three-hour experiment. These graphs decompose the turnaround time for a request in DIET into (1) the time required for the DIET hierarchy to select the best server(s) for the request, (2) the time that the request spends at the SeD-level waiting to begin execution, and (3) the time required to perform the computation specified by the request; all times are wall-clock time. The arrival of tasks (i.e. the start of the `Time to choose server` phase) is relatively steady because task arrival is controlled by the 96 `user` scripts which act independently of each other. Task completion (i.e. the end of the `Time spent computing`) is highly irregular because 24 different servers are in use for task completion and the time required to complete each request depends on many factors such as the number of other requests allocated to the same server.

For the `Standard` DIET approach, nearly all time is spent computing the request and the DIET hierarchy performs server selection very quickly. For the `SeD Queue` approach, the DIET hierarchy also responds very quickly, requests spend a significant amount of time waiting in the queue at the SeD, and the computation time is very low relative to the `Standard` approach. For the `Task Scheduler` approach, requests are stalled for a significant amount of time during the server selection phase and the window-based scheduling approach is evident from the "stair-step" appearance of the end of the `Time to choose server` phase. After server selection, the requests generally wait a relatively short time in the SeD queue before entering the computation phase. It is notable that task completion times are more stable for the `Task Scheduling` approach than for the other approaches.

6.4.3 Request throughput

Figure 5 shows the mean and standard deviation of the request completion rate of DIET with each scheduling strategy. The standard DIET approach provides the highest request throughput; this result is reasonable

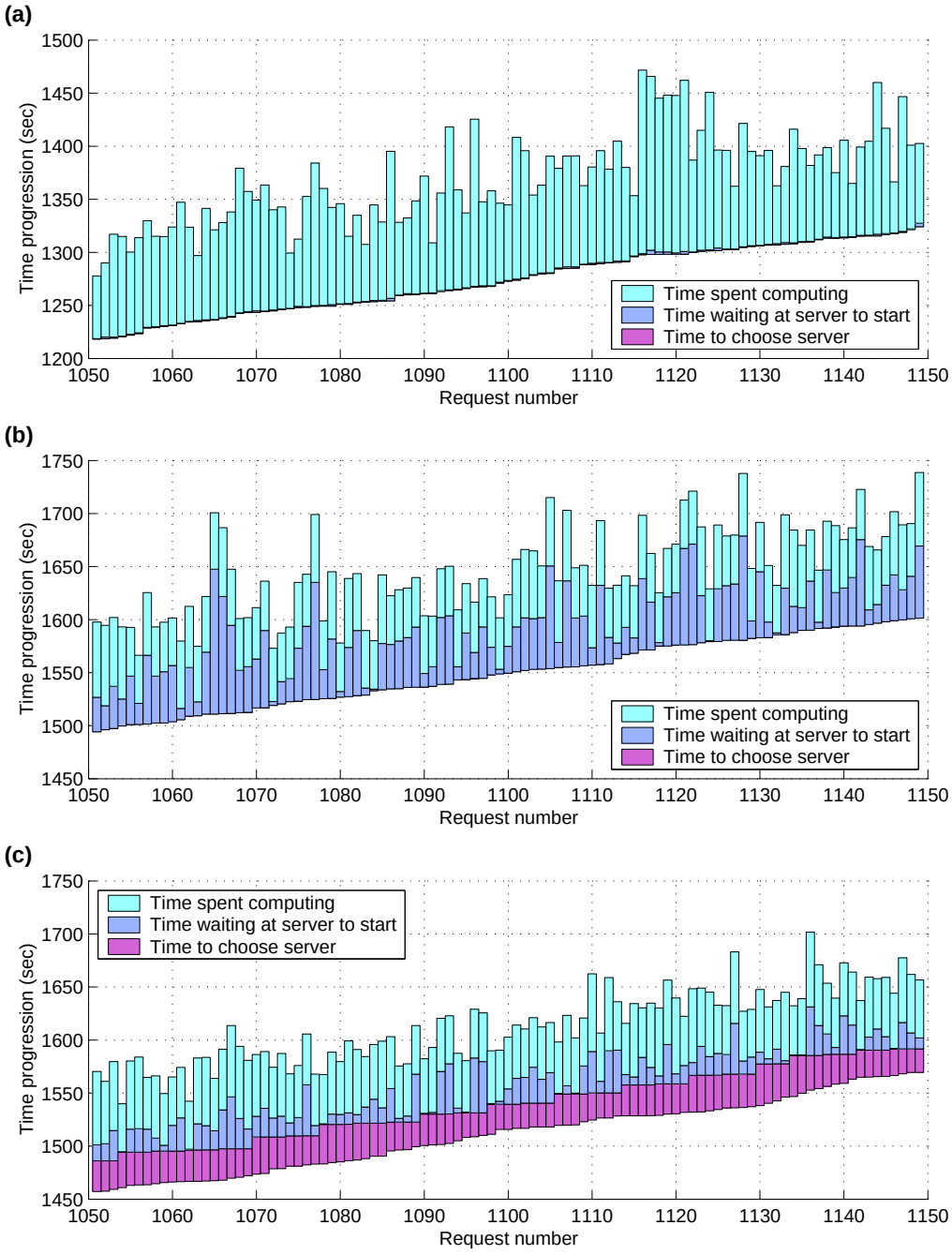


Figure 4: A snapshot of a small portion of the steady-state experiment executions for (a) the `Standard` DIET approach, (b) the `SeD Queue` approach, and (c) the `Task Scheduler` approach.

since the on-line strategy used by the standard DIET strategy has very low overhead. For these experiments, the standard DIET strategy typically scheduled many jobs at once on a server; in many cases, this approach will cause performance problems due to the overheads of switching between the jobs and/or the interference between the jobs for resources such as memory and cache. However, the implementation of `dgemm` used for these experiments is not well optimized for cache or memory usage and is not very susceptible to interference from competing requests; in detailed experiments we verified, for example, that two competing `dgemm` calculations run in nearly the same time as one (the tests were run on a dual-processor machine) and that four competing `dgemm` calculations run in roughly twice the time as two. For tasks that experience a greater level of interference from multiple tasks running simultaneously, the standard DIET approach will likely show related performance problems.

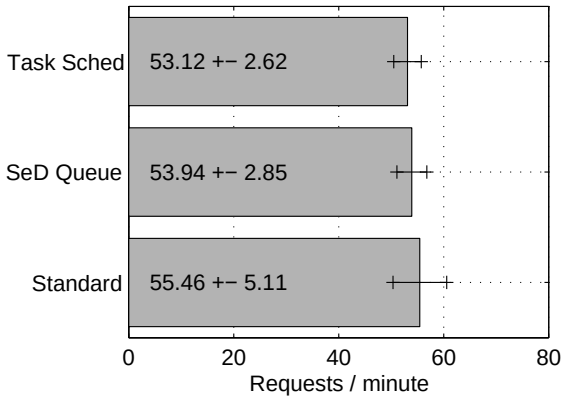


Figure 5: Comparison of request completion rate sustained by each scheduling strategy in steady-state, high-load conditions.

The `SeD Queue` approach provided throughput that was 2.8% lower than the `Standard` approach. This loss of performance can be attributed to two aspects of the `SeD Queue` approach. First, some overhead is associated with managing the synchronization of requests at the server to provide queue-like behavior. Second, the number of requests that can be calculating at once was restricted to two for this strategy in these experiments. There is thus no opportunity to make progress on requests during idle phases in the computation. For a problem such as `dgemm` that does not exhibit much interference between competing requests and given the high-load, steady-state arrangement of these experiments, the `SeD Queue` strategy would likely have performed slightly better with a limit of three or four concurrent requests.

The `Task Scheduler` approach provided throughput that was 4.5% lower than the `Standard` approach and 1.5% lower than the `SeD Queue` approach. The primary source of lost performance for this approach is that the MA-level task scheduler may sometimes define a window that is too long; in these cases, servers are sometimes left for short periods of time with no requests to run.

In general, we feel that the overheads associated with the `SeD Queue` and `Task Scheduler` approaches are acceptable given the possible advantages associated with the extensions.

7 Discussion

7.1 Summary of findings

This paper began with a detailed discussion and analysis of the DIET architecture and the hierarchical scheduling approach used by DIET agents and servers. Previous papers have not given such a detailed analysis of DIET scheduling, thus this provides an important general contribution and lays the groundwork for our extensions. We then described two extensions made to the standard DIET scheduling approach. The first extension provides the ability to limit the number of concurrent jobs that can run on a server at a time. This feature can be important under high-load conditions or for very resource-intensive jobs to

avoid overloading the server. The feature can also be used as a more friendly configuration of DIET that provides more equal sharing with non-DIET processes. The second extension involved adding a level of global task scheduling control at the Master Agent. This extension provides two advantages: the ability to control the flow of requests into the DIET system, thus avoiding the problem of scheduling too far in advance under heavy load conditions; and the ability to re-arrange task placements within a scheduling window. We presented a very simple algorithm for task placement refinement, but it is our goal to explore other algorithms in future work.

We then presented experiments comparing the performance of the `Standard` DIET scheduling approach against that of the `SeD Queue` and `Task Scheduler` approaches. We demonstrated that under bursty load conditions, the `Task Scheduler` provides the best and most reliable performance regardless of user behavior. Next, we presented the results of long-running experiments with a steady, heavy request load. We found that the `SeD Queue` and `Task Scheduler` approaches introduced some additional overhead as compared to the `Standard` approach, but that the overhead was reasonable.

In combination, these results indicate that we succeeded in introducing useful new functionality to DIET scheduling approach without losing the performance benefits of the standard approach.

7.2 Future work

Our future work will consist in experimenting with distributed scheduling algorithms at different levels of the hierarchy. First we plan to experiment with different task / server affinity models. When data dependences occur between requests on the same server (or even between different servers), we need to move requests in the queues to optimize data management [3].

Other heuristics could be implemented within our schedulers. A paper from the APST team presents different algorithms for scheduling jobs in grid environments (Min-min, Max-Min, Sufferage, and X-Sufferage) first simulated [9] and then tested in real-world deployment [10]. It will be interesting to adapt these strategies to our distributed, window-based approach.

We plan to extend our tests on the Grid'5000 [30] experimental platform to incorporate multiple clusters distributed throughout France. This platform provides a unique opportunity to perform experiments under a relatively controlled environment that is also distributed and heterogeneous. We expect to show better gain of hierarchical scheduling at a larger scale and on a heterogeneous set of machines.

Finally, we are working on plug-in schedulers. This will allow the user to play with the internals of agents and tune DIET's scheduling by changing the heuristics, adding queues, changing the performance metrics, and changing the aggregation functions. We believe that this feature will be useful both for computer scientists to test their algorithms on a real platform and expert application scientists to tune DIET for specific application behavior.

Acknowledgements

The authors would like to thank the reviewers for useful insights on ways to improve the paper. We are also grateful to Alan Su, Raphaël Bolze, and Eddy Caron for many enlightening discussions about distributed scheduling.

References

[1] D. Andresen and T. McCune. H-SWEB: A Hierarchical Scheduling System for Distributed WWW Server Clusters. *Concurrency: Practice and Experience*, 12:189–210, 2000.

[2] F. Berman, G.C. Fox, and A.J.H. Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003.

[3] G. Bosilca, G. Fedak, and F. Cappello. OVM: Out-of-Order Execution Parallel Virtual Machine. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID'01)*, 2001.

[4] E. Caron, P.K. Chouhan, and A. Legrand. Automatic Deployment for Hierarchical Network Enabled Server. In *The 13th Heterogeneous Computing Workshop (HCW 2004)*, Santa Fe. New Mexico, USA, April 2004.

[5] E. Caron, F. Desprez, F. Lombard, J.-M. Nicod, M. Quinson, and F. Suter. A Scalable Approach to Network Enabled Servers. In *Proc. of EuroPar 2002*, Paderborn, Germany, 2002.

[6] J. Carreto, J. Fernández, F. García, and A. Chouhary. A Hierarchical Disk Scheduler for Multimedia Systems. *Future Generation Computer Systems*, 19:23–35, 2003.

[7] H. Casanova, S. Matsuoka, and J. Dongarra. Network-Enabled Server Systems: Deploying Scientific Simulations on the Grid. In *High Performance Computing Symposium (HPC'01)*, Seattle, Washington (USA), April 2001.

[8] Henri Casanova and Jack Dongarra. NetSolve: A network server for solving computational science problems. In *Proceedings of Supercomputing Conference (SC'96)*, November 1996.

[9] Henri Casanova, Arnaud Legrand, Dmitrii Zagorodnov, and Francine Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *9th Heterogeneous Computing Workshop (HCW'00)*, 2000.

[10] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The apples parameter sweep template: User-level middleware for the grid. In *Proceedings of Supercomputing Conference (SC'2000)*, November 2000.

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

[12] Sivarama Dandamudi. *Hierarchical Scheduling in Parallel and Cluster Systems*. Kluwer Academic/Plenum Publishers, New York, USA, 2003.

[13] Gilles Fedak, Cécile Germain, Vincent Néri, and Franck Cappello. XtremWeb: A generic global computing system. In *Workshop on Global Computing on Personal Devices, held in conjunction with CCGrid'2001*, Brisbane, Australia, May 2001. IEEE Press.

[14] D.G. Feitelson, L. Rudolf, U. Schwiegelshohn, K.C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D. Feitelson and L. Rudolf, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer Verlag, 1997.

[15] G. Feitelson. A Survey of Scheduling in Multiprogrammed Parallel Systems. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, 1997.

[16] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[17] Ian Foster and Carl Kesselman. The Globus Project: A status report. In *7th Heterogeneous Computing Workshop*, pages 4–18. IEEE Press, 1998.

[18] Jorn Gehring and Thomas Preiss. Scheduling a Metacomputer with Un-cooperative Subschedulers. In *Proc. IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *LNCS*, Puerto Rico, April 1999. Springer.

[19] 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.

[20] Z. Juhasz, A. Andics, and S. Pota. Towards A Robust And Fault-Tolerant Multicast Discovery Architecture For Global Computing Grids. In *4th Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2002)*, Linz, Austria, September 2002.

[21] 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.

[22] M. Quinson. Dynamic Performance Forecasting for Network-Enabled Servers in a Metacomputing Environment. In *International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'02)*, April 15-19 2002.

[23] 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.

[24] Edward Seidel, Gabrielle Allen, André Merzky, and Jarek Nabrzyski. GridLab–a grid application toolkit and testbed. *Future Generation Comp. Syst.*, 18(8):1143 – 1153, 2002.

[25] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. An Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *3rd International Workshop on Grid Computing*, November 2002.

[26] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. Distributed Job Scheduling on Computational Grids Using Multiple Simultaneous Requests. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*. IEEE Computer Society, 2002.

[27] Douglas Thain, Todd Tannenbaum, , and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 2005.

[28] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6):757–768, Oct. 1999.

[29] Workshops on job scheduling strategies for parallel processing. `http://www.cs.huji.ac.il/~feit/parsched/`, 1995-2003.

[30] Grid 5000 project. `http://www.grid5000.org`.