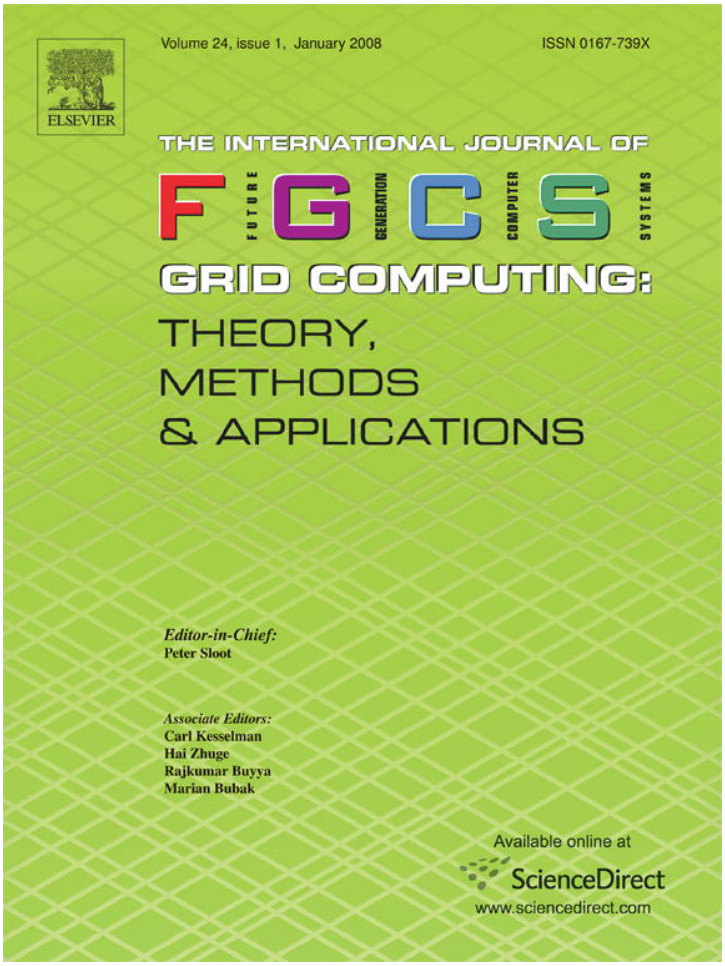


Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 24 (2008) 46–57
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

Design of plug-in schedulers for a GridRPC environment☆

E. Caron[a], A. Chis[a], F. Desprez[a,*], A. Su[b]

[a] LIP/ENS-Lyon/INRIA/CNRS/UCBL, 46 Allée d'Italie, F-69364 Lyon, France
[b] Google Inc., One Broadway, 14th floor, Cambridge, MA 02142, USA

Received 29 January 2007; accepted 8 February 2007
Available online 1 March 2007

Abstract

Grid middleware performance is typically determined by the resources that are chosen. Different classes of applications need different metrics to define a meaningful notion of performance. Such metrics include application workload, execution time estimation, disk or memory space availability, etc. In the past, few environments have allowed schedulers to be tuned for specific application scenarios. Within the DIET (Distributed Interactive Engineering Toolbox) project, we developed an API that allows the resource broker to be tuned for specific application classes. In a seamless way, generic or application-dependent performance measures can be used within the hierarchy of resource brokers.

Keywords: Grid computing; Scheduling; Performance prediction

1. Introduction

The GridRPC approach [12], in which remote servers execute computation requests on behalf of clients connected through the Internet, is one of the most flexible and efficient solutions for porting large-scale applications over the grid. In this model, resource brokers (also called agents) are responsible for identifying the most appropriate servers for a given set of requests. Thanks to the growth of network bandwidth and the reduction of network latency, small computation requests can now be sent to servers available on the grid. To make effective use of today's scalable resource platforms, it is important to ensure scalability in the middleware layers as well.

One of the main problems is, of course, to evaluate the costs associated with the remote execution of these requests. Depending on the target application, several measurements have to be taken into account. Execution time is typically the measure with which users are most concerned, but other factors, such as available memory, disk space, machine load, and batch queue length may also be significant. Moreover, particular applications may exhibit domain-specific execution behavior, which should also be considered. However, these various performance measures can only be taken into account if the middleware allows the tuning of its internal scheduling software by the applications it services.

The Distributed Interactive Engineering Toolbox (DIET) [2] project is focused on the development of scalable middleware by distributing the scheduling problem across multiple agents. DIET consists of a set of elements that can be used together to build applications using the GridRPC paradigm. This middleware is able to find an appropriate server according to the information given in the client's request (problem to be solved, size of the data involved), the performance of the target platform (server load, available memory, communication performance) and the local availability of any data stored during previous computations. The scheduler is distributed using several collaborating hierarchies connected either statically or dynamically (in a peer-to-peer fashion). Data management is provided to allow persistent data to stay within the system for future re-use. This feature avoids unnecessary communication when dependences exist between different requests. Using a hierarchical set of schedulers connecting servers and clients enhances the scalability of the platform, but it also complicates the aggregation of performance information inside the middleware. Moreover, depending on the target applications, different performance measures can be used to

☆ This work has been supported in part by the ANR project LEGO (ANR-05-CIGC-11).

* Corresponding author. Tel.: +33 4 7272 8569; fax: +33 4 7272 8806.
E-mail addresses: Eddy.Caron@ens-lyon.fr (E. Caron), Frederic.Desprez@ens-lyon.fr (F. Desprez), alsu@google.com (A. Su).

0167-739X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2007.02.005

select a set of appropriate servers which are then able to solve a request on behalf of a given client. This paper presents the approach chosen within the DIET (Distributed Interactive Engineering Toolbox) project to allow a resource broker to be tuned for specific application classes. Our design supports the use of generic or application-dependent performance measures in a simple and seamless way.

The remainder of the paper is organized as follows. We begin with a survey of relevant related work in the next section. Section 3 presents the architecture of the DIET middleware framework. Section 4 describes the plug-in scheduler feature, and Section 5 describes the CoRI collector which allows the different performance measures to be managed. Finally, Section 6 presents some early experiments of this new feature of our GridRPC framework before we conclude in Section 7.

2. Related work

Several middleware frameworks follow the GridRPC API from the OGF like Ninf [8] or GridSolve [17], but none of them expose interfaces that allow their scheduling internals to be tuned for specific application classes. The scheduler interface is not part of the GridRPC standard, and thus any optimization can be done at the middleware level.

APST [3] allows some modifications of the internals of the scheduling phase, mainly to be able to choose the scheduling heuristic. Several heuristics can be used like Max–min, Min–min, or X-sufferage.

Scheduling heuristics for different application classes have been designed within the AppLeS [4] and GrADS [1] projects. GrADS is built upon three components [5]. A *Program Preparation System* handles application development, composition, and compilation, and a *Program Execution System* provides on-line resource discovery, binding, application performance monitoring, and rescheduling. Finally, a *binder* performs a resource-specific compilation of the intermediate representation code before it is launched on the available resources. One interesting feature of GrADS is that it provides application-specific performance models. Depending on the target application, these models can be extended, based on analytical evaluations by experts and empirical models obtained by real world experiments. For some applications, simulation may be used to project their behavior on particular architectures. These models are then fed into the schedulers to find the most appropriate resources to run the application.

Recently, some work has been done to be able to cope with dynamic platform performance at the execution time [11,15]. These approaches allow an algorithm to automatically adapt itself at run-time depending on the performance of the target architecture, even if it is heterogeneous.

Within the Condor project [14], the ClassAds language was developed [10]. The syntax of this language is able to express resource query requests with attributes by means of a rich set of language constructs: lists of constants, arbitrary collections of constants, and variables combined with arithmetic and logic operators. The result is a multi-criteria decision problem. The approach presented in our paper allows a scheduling framework to offer useful information to a metascheduler like Condor.

3. DIET aim and design choices

The DIET component architecture is structured hierarchically for improved scalability. Such an architecture is flexible and can be adapted to diverse environments, including arbitrary heterogeneous computing platforms. The DIET toolkit [2] is implemented in CORBA, and thus benefits from the many standardized, stable services provided by freely-available and high performance CORBA implementations. CORBA systems provide a remote method invocation facility with a high level of transparency. This transparency should not substantially affect the performance, as the communication layers in most CORBA implementations are highly optimized [6]. These factors motivate our decision to use CORBA as the communication and remote invocation fabric in DIET.

The DIET framework comprises several components. A *Client* is an application that uses the DIET infrastructure to solve problems using an RPC approach. Clients access DIET via various interfaces: web portals, PSEs such as SCILAB, or programmatically using published C, C++, or Java APIs. A *SeD*, or server daemon, acts as the service provider, exporting functionality via a standardized computational service interface; a single SeD can offer any number of computational services. A SeD can also serve as the interface and execution mechanism for either a stand-alone interactive machine or a parallel supercomputer, by interfacing with its batch scheduling facility. The third component of the DIET architecture, *agents*, facilitate the service location and invocation interactions of clients and SeDs. Collectively, a hierarchy of agents provides higher-level services such as scheduling and data management. These services are made scalable by distributing them across a hierarchy of agents composed of a single *Master Agent* (MA) and several *Local Agents* (LA). Fig. 1 shows an example of a DIET hierarchy.

The *Master Agent* of a DIET hierarchy serves as the distinguished entry point from which the services contained within the hierarchy may be logically accessed. Clients identify the DIET hierarchy using a standard CORBA naming service. Clients submit requests – composed of the name of the specific computational service they require and the necessary arguments for that service – to the MA. The MA then forwards the request to its children, who subsequently forward the request to their children, and so on, such that the request is eventually received by all SeDs in the hierarchy. SeDs then each 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 the local availability of datasets specifically needed by the application. SeDs forward this capacity information back up the agent hierarchy. Based on the capacities of individual SeDs to service the request at hand, agents at each level of the hierarchy reduce the set of server responses to a manageable list of server choices with the greatest potential. This selection is

Table 1
Explanation of the estimation tags

Information tag starts with EST_	Multi-value	Explanation
TCOMP		The predicted time to solve a problem
TIMESINCELASTSOLVE		Time elapsed since last execution started (s)
FREECPU		Amount of free CPU between 0 and 1
LOADAVG		CPU load average
FREEMEM		Amount of free memory (Mb)
NBCPU		Number of available processors
CPUSPEED	x	Frequency of CPUs (MHz)
TOTALMEM		Total memory size (Mb)
BOGOMIPS	x	The BogoMips
CACHECPU	x	Cache size CPUs (Kb)
NETWORKBANDWIDTH		Network bandwidth (Mb/s)
NETWORKLATENCY		Network latency (s)
TOTALDISKSIZE		Size of the partition (Mb)
FREEDISKSIZE		Amount of free place on partition (Mb)
DISKACCESSREAD		Average time to read from disk (Mb/s)
DISKACCESSWRITE		Average time to write to disk (Mb/s)
ALLINFO	x	(empty) Fill all possible fields

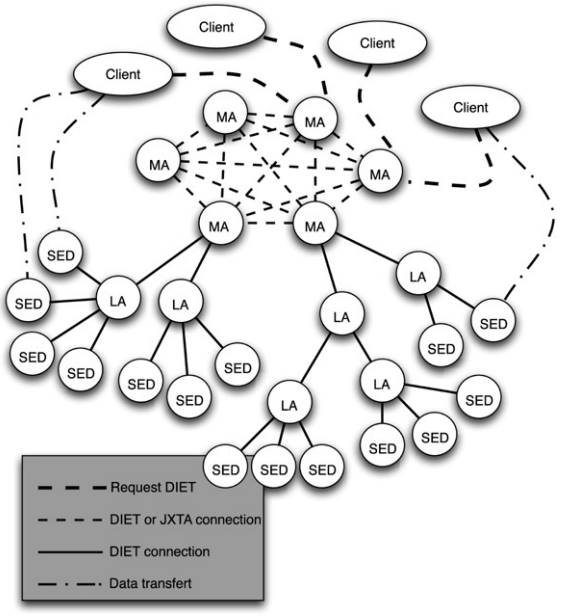


Fig. 1. DIET hierarchical organization. Plug-in schedulers are available in each MA and LA.

performed using an objective function (e.g., computation cost, communication cost, or machine load) that is appropriate for the application. The client program may then submit the request directly to any of the proposed servers, though typically the first server will be preferred as it is predicted to be the most suitable server. The scheduling strategies used in DIET are the subject of this paper.

4. Plug-in scheduler

A first version of scheduling in DIET was based on the FIFO principle — a task submitted by a client was scheduled on the SeD whose response to the service query arrived first at the MA. This approach did not directly take into consideration the dynamic states of the SeDs. A second version allowed a mono-criterion scheduling based on application-specific performance predictions. Later a round-robin scheduling scheme was implemented in DIET, resulting in a good performance for the task distribution over a homogeneous platform; however, this approach was found to be lacking when the workload or the platform was sufficiently heterogeneous.

With our recent enhancements to the DIET toolkit, applications are now able to exert a degree of control over the scheduling subsystem via *plug-in schedulers*. As the applications that are to be deployed on the grid vary greatly in terms of performance demands, the DIET plug-in scheduler facility permits the application designer to express application needs and features in order that they be taken into account when application tasks are scheduled. These features are invoked at runtime after a user has submitted a service request to the MA, which broadcasts the request to its agent hierarchy.

When an application service request arrives at a SeD, it creates a *performance estimation vector* — a collection of *performance estimation values* that are pertinent to the scheduling process for that application. The values to be stored in this structure can be either values provided by CoRI (Collectors of Resource Information) described in Section 5, or custom values generated by the SeD itself. The design of the estimation vector subsystem is modular; future performance measurement systems can be integrated with the DIET platform in a fairly straightforward manner.

CoRI generates a basic set of performance estimation values, which are stored in the estimation vector and identified by system-defined tags; Table 1 lists the tags that may be generated by a standard CoRI installation. Application developers may also define performance values to be included in a SeD response to a client request. For example, a DIET SeD that provides a service to query particular databases may need to include information about which databases are currently resident in its disk cache, in order that an appropriate server be identified for each client request. By default, when

a user request arrives at a DIET SeD, an estimation vector is created via a default estimation function; typically, this function populates the vector with standard CoRI values. If the application developer includes a custom *performance estimation function* in the implementation of the SeD, the DIET framework will associate the estimation function with the registered service. Each time a user request is received by a SeD associated with such an estimation function, that function, instead of the default estimation procedure, is called to generate the relevant performance estimation values.

In the performance estimation routine, the SeD developer should store in the provided estimation vector any performance data needed by the agents to evaluate and compare server responses. Such vectors are then the basis on which the suitability of different SeDs for a particular application service request is evaluated. Specifically, a local agent gathers responses generated by the SeDs that are its descendents, sorts those responses based on application-specific comparison metrics, and then transmits the sorted list to its parent. The mechanics of this sorting process comprises an *aggregation method*, which is simply the logical process by which SeD responses are sorted. If application-specific data *are* supplied (i.e., the estimation function has been redefined), an alternative method for aggregation is needed. Currently, a basic *priority scheduler* has been implemented, enabling an application developer to specify a multi-criteria scheduling policy based on a series of performance values that are to be optimized in succession. A developer may implement a priority scheduler using the following interface:

```
diet_aggregator_desc_t* diet_profile_desc_
aggregator(diet_profile_desc_t* profile);
int diet_aggregator_set_type(diet_aggregator_
desc_t* agg,
diet_aggregator_type_t atype);
int diet_aggregator_priority_max
(diet_aggregator_desc_t* agg,
diet_est_tag_t tag);
int diet_aggregator_priority_min
(diet_aggregator_desc_t*
agg,diet_est_tag_t tag);
int diet_aggregator_priority_maxuser
(diet_aggregator_desc_t* agg, int val);
int diet_aggregator_priority_
minuser(diet_aggregator_desc_t* agg, int val);
```

The `diet_profile_desc_aggregator` and `diet_aggregator_set_type` functions fetch and configure the aggregator corresponding to a DIET service profile, respectively. In particular, a priority scheduler is declared by invoking the latter function with `DIET_AGG_PRIORITY` as the `agg` parameter. Recall that from the point of view of an agent, the aggregation phase is essentially a sorting of the server responses from its children. A priority scheduler logically uses a series of user-specified tags to perform the pairwise server comparisons needed to construct the sorted list of server responses.

In order to define the tags and the order in which they should be compared, four functions are introduced. These functions, of the form `diet_aggregator_priority_*`, serve to identify the estimation values to be optimized during the aggregation phase. The `_min` and `_max` forms indicate that a standard performance metric (e.g., time elapsed since last execution, from the `diet_estimate_lastexec` function) is to be either minimized or maximized, respectively. Similarly, the `_minuser` and `_maxuser` forms indicate the analogous operations on user-supplied estimation values. The order of calls to these functions indicate the order of *precedence* of the tags, with a higher priority given to tags specified earlier in the process of defining the scheduler.

Each time two server responses need to be compared, the values associated with the tags specified in the priority aggregator are retrieved. In the specified order, pairs of corresponding values are successively compared, passing to the next tag only if the values for the current tag are identical. If one server response contains a value for the metric currently being compared, and another does not, the response with a valid value will be selected. If at any point during the treatment of tags *both* responses lack the necessary tag, the comparison is declared indeterminate. This process continues until one response is declared superior to the other, or all tags in the priority aggregator are exhausted (in which case, the responses are judged equivalent).

5. CoRI

As we have seen in the previous section, the scheduler requires performance measurement tools to make effective scheduling decisions. Thus, DIET depends on reliable grid resource information services. In this section, we introduce the exact requirements of DIET for a grid information service, the architecture of the new tool CoRI (Collectors of Resource Information), and the different components inside of CoRI.

5.1. CoRI architecture

In this section, we describe the design of CoRI, the new platform performance subsystem that we have implemented to enable future versions of the DIET framework to more easily interface with third-party performance monitoring and prediction tools. Our goal is to facilitate the rapid incorporation of such facilities as they emerge and become widely available. This issue is especially pertinent, considering the fact that DIET is designed to run on heterogeneous platforms, on which many promising but immature tools may not be universally available. Such a scenario is common, considering that many such efforts are essentially research prototypes. To account for such cases, we have designed the performance evaluation subsystem in DIET to be able to function even in the face of varying levels of information in the system.

We designed CoRI to ensure that it (i) provides timely performance information to avoid impeding the scheduling process, and (ii) presents a general-purpose interface capable of encapsulating a wide range of resource information services. Firstly, it must provide basic measurements that are available regardless of the state of the system. The service developer

can rely on such information even if no other resource performance prediction service like FAST [9] (Fast Agent's System Timer) or NWS [16] is installed. Secondly, the tool must manage the simultaneous use of different performance prediction systems within a single heterogeneous platform. To address these two fundamental challenges, we offer two solutions: the CoRI-*Easy* collector to universally provide basic performance information, and the CoRI *Manager* to mediate the interactions among different collectors. In general, we refer collectively to both these solutions as the CoRI tool, which stands for Collectors of Resource Information. Both subsystems are described in the following section.

Broadly, the CoRI-Easy facility is a set of simple requests for basic resource information, and the CoRI Manager is the tool that enables application developers to add access methods to other resource information services. As CoRI-Easy is fundamentally just a resource information service, we implement it as a collector that is managed by the new CoRI Manager. Note that CoRI-Easy is not the only collector available; for example, FAST can be used as well. Moreover, adding new collectors is a simple matter of implementing a thin interface layer that adapts the performance prediction data output to the reporting API that the CoRI Manager expects.

5.2. CoRI Manager

The CoRI Manager provides access to different *collectors*, which are software components that can provide performance information about the system. This modular design decouples the choice of measurement facilities and the utilization of such information in the scheduling process. Even if the manager should aggregate across performance data originating from different resource information services, the raw trace of data remains, and so its origin can be determined. For example, it could be important to distinguish the data coming from the CoRI-Easy collector and the FAST collector, because the performance prediction approach that FAST uses is more highly tuned to the features of the targeted application. Furthermore, the modular design of the CoRI Manager also permits a great deal of extensibility, in that additional collectors based on systems, such as Ganglia [7] or NWS [16], can be rapidly implemented via relatively thin software interface layers. This capability enables DIET users to more easily evaluate prototype measurement systems even before they reach full production quality.

5.3. Technical overview of CoRI

In this section, we describe in greater detail the structure of the *estimation vector*, the data structure used within DIET to transmit resource performance data. We then enumerate the standard metrics of performance used in DIET and present the various CoRI Manager functions. The estimation vector is divided into two parts. The first part represents "native" performance measures that are available through CoRI (e.g., the number of CPUs, the memory usage, etc.) and the scheduling subsystem (e.g., the time elapsed since a server's last service invocation). The second part is reserved for developer-defined measurements that are meaningful solely in the context of the application being developed. The vector supports the storage of both singleton data values (which we call scalars) and data value series, because some performance prediction measurements are inherently a list of values (e.g., the load on each processor of a multi-processor node). An estimation vector is essentially a container for a complete performance snapshot of a server node in the DIET system, composed of multiple scalars and lists of performance data.

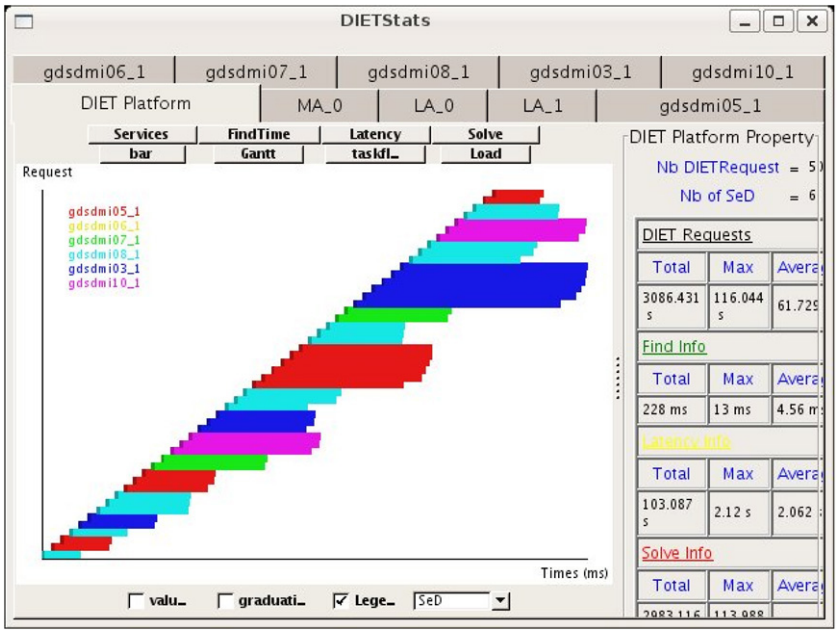
To differentiate among the various classes of information that may be stored in an estimation vector, a *data tag* is associated with each scalar value or list of related values. This tag enables DIET's performance evaluation subsystems to identify and extract performance data. There are two types of tags: system tags and user-defined tags. System tags correspond to application-independent data that are stored and managed by the CoRI Manager; Table 1 enumerates the set of tags that are supported in the current DIET architecture. User-defined tags represent application-specific data that are generated by specialized performance estimation routines that are defined at compile-time for the SeD. Note also that the `EST_ALLINFOS` tag is in fact a pseudo-tag that is simply used to express a request for all performance information that is available from a particular collector. At the moment of service registration, a SeD also declares a priority list of comparison operations based on these data that logically expresses the desired performance optimization semantics. The details of the mechanisms for declaring such an optimization routine fall outside the scope of this paper; for a fully detailed description of this API, please consult the DIET website [13].

The basic public interface of the CoRI Manager that is available to DIET service developers consists of three functions. The first function allows the initialization of a given collector and adds the collector to the set of collectors that are under the control of the CoRI Manager. The second function provides access to measurements. The last function tests the availability of the CoRI-Easy collector.

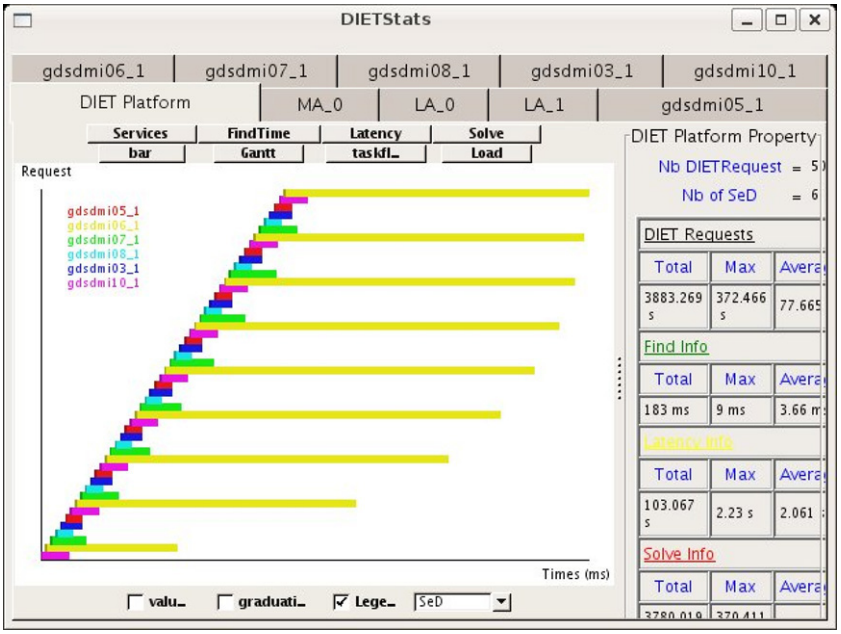
5.4. CoRI-easy collector

The CoRI-Easy collector is a resource collector that provides the basic performance measurements of the SeD. Via the interface with the CoRI Manager, the service developer and the DIET developer are able to access CoRI-Easy metrics. We first introduce the basic design of CoRI-Easy, and then we will discuss some specific problems. CoRI-Easy should be extensible like CoRI Manager, i.e. the measurement functions must be easily replaceable or extended by a new functionality as needed.

Consequently, we use a functional approach: functions are categorized by the information they provide. Each logical type of performance information is identified by an *information class*, which enables users to simply test for the existence of a function providing a desired class of information. Our goal was not to create another sensor system or monitor service;



(a) CPU scheduler task distribution.



(b) RR scheduler task distribution.

Fig. 2. CPU vs RR scheduler — taskflow for 50 requests spaced at 5 s.

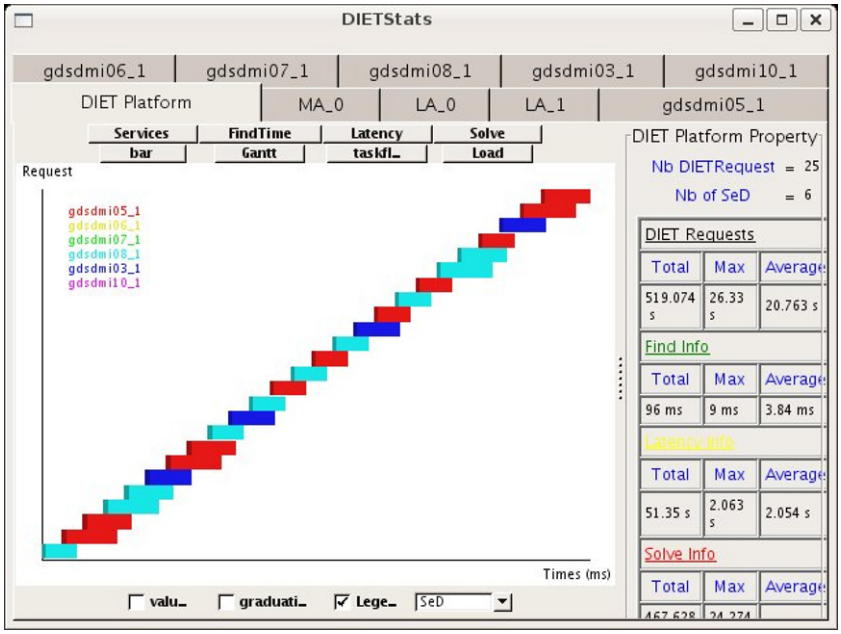
CoRI-Easy is simply a set of routines that produce basic performance metrics. Note that the data measured by CoRI-Easy are available via the interface of the CoRI Manger.

CPU evaluation. CoRI-Easy provides CPU information about the node that it monitors: the *number of CPUs*, the *CPU frequency* and the *CPU cache size*. These static measurements do not provide a reliable indication of the actual load of CPUs, so we also measure the node's *BogoMips* (a normalized indicator of the CPU power), the *load average*, and *CPU utilization* for indicating the CPU load.

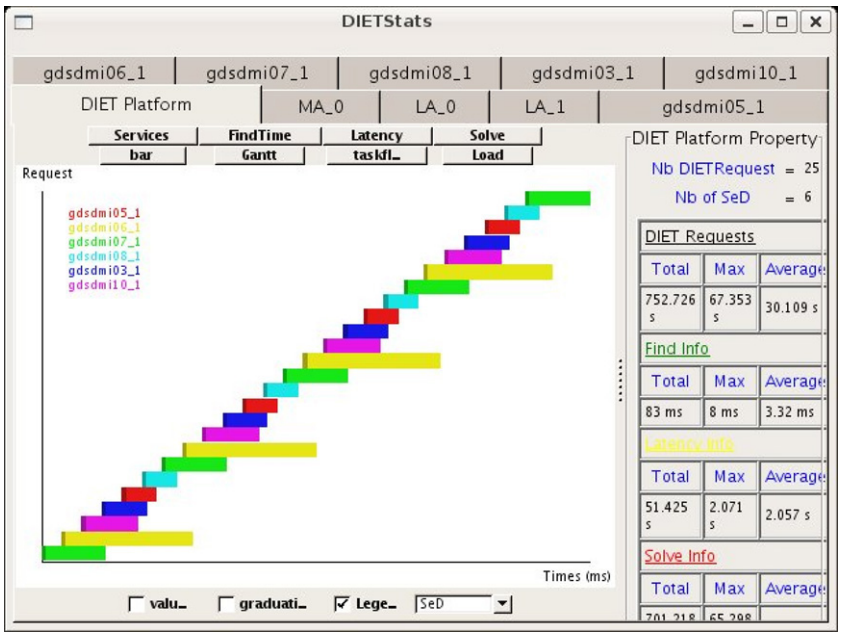
Memory capacity. The memory is the second important factor that influences performance. CoRI monitors the *total memory size* and the *available memory size*.

Disk performance and capacity. CoRI-Easy also measures the *read and write performance* of any storage device available to the node, as well the *maximal capacity* and the *free capacity* of any such device.

Network performance. CoRI-Easy should monitor the performance of interconnection networks that are used to



(a) CPU scheduler task distribution.



(b) RR scheduler task distribution.

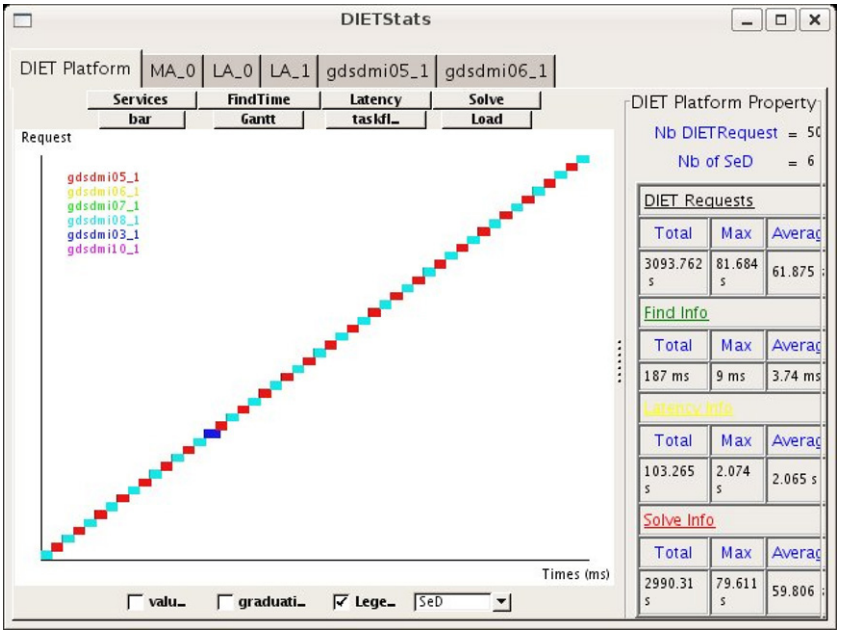
Fig. 3. CPU vs RR scheduler — taskflow for 25 requests spaced at 10 s.

reach other nodes, especially those in the DIET hierarchy to which that node belongs; this functionality is planned for a future release.

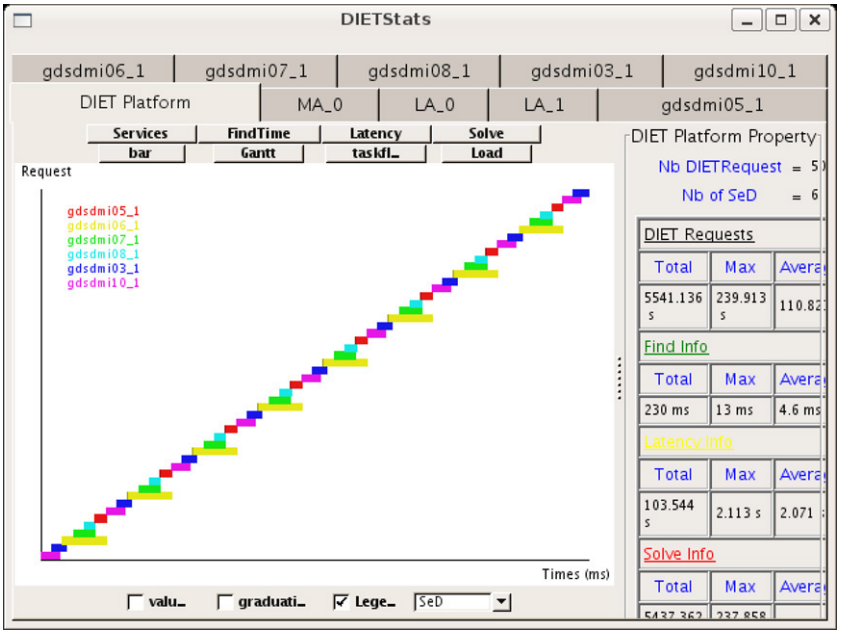
6. Experimental evaluation

Two experiments have been performed, aimed at proving the utility of the new functionality added to DIET. They both compare a basic application-independent round robin tasks distribution with a distribution obtained by a scheduler that accounts for processor capacities (for computationally intensive homogeneous tasks), and with a distribution based on the criterion of disk access speed respectively (for I/O intensive tasks), hypothesizing that the schedulers using application-relevant information will perform better than the round robin distribution policy on a heterogeneous platform.

The test platform for the first experiment, comprising 1 MA, 2 LAs, and 6 SeDs (each LA having 3 associated SeDs), was deployed on a homogeneous cluster. The computationally intensive task consisted of several runs of the DGEMM (i.e. matrix–matrix multiplication) function from BLAS (Basic Linear Algebra Subprograms) library.



(a) CPU scheduler task distribution.



(b) RR scheduler task distribution

Fig. 4. CPU vs RR scheduler — taskflow for 50 requests spaced 1 min apart.

The Round Robin scheduler (RR scheduler) is a priority scheduler that aggregates SeD responses based on the time elapsed since the last execution started (the `EST_TIMESINCELASTSOLVE` CoRI tag), whereas the CPU scheduler is a priority scheduler that maximizes the ratio $\frac{\text{BOGOMIPS}}{1+\text{load\_average}}$ (the `EST_BOGOMIPS` and `EST_LOADAVG` CoRI tags respectively). The BOGOMIPS metric characterizes the raw processor speed, and the load average provides an indication of CPU contention among runnable processes — in this case an estimation over the last minute. The different CPU speeds of the resources were simulated by running the DGEMM function a different number of times (on each resource) and adapting the BOGOMIPS value obtained through CoRI accordingly. The behavior of the two schedulers was studied for requests spaced at different time intervals.

Fig. 2 shows the task distribution provided by the two schedulers for a request inter-arrival time of 5 s. For such a small interval between requests, the load average estimation over the last minute is not sufficiently sensitive. Hence, tasks scheduled by the CPU scheduler accumulate successively at the fastest nodes, overloading them and resulting in increased computation times for all tasks (Fig. 2(a)). Tasks scheduled by

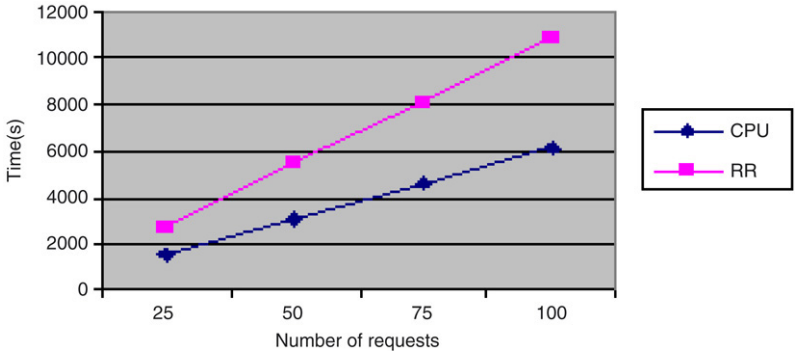


Fig. 5. CPU vs RR scheduler — total computation time for requests spaced at 1 min.

the RR scheduler are evenly distributed across the platform, resulting in a generally reduced computation time compared to the other case. However, tasks assigned to the slowest processor are heavily penalized, and thus suffer from greatly increased computation time (Fig. 2(b)). The improvement ratio for the overall computation time is 20%.

As shown in Fig. 3, with requests spaced at 10 s, the load average estimation is already more accurate, resulting in a better task distribution for the CPU scheduler with fairly equal computation times for all tasks. As expected, the RR scheduler policy still assigned a substantial number of tasks to slow processors. The improvement ratio in this case is 30%.

As shown in Fig. 4, with requests spaced at 1 min, the load average estimation is even more accurate: the tasks distributed by the CPU scheduler are scheduled alternately on the fastest processors. The improvement ratio is 44%.

The results show that the CPU-based scheduling policy provides an equitable allocation of platform resources, as overall, the run times for the tasks are almost equal. Conversely, the RR scheduler was unable to balance the aggregate load of the tasks: some completed quickly, whereas those assigned to heavily loaded processors were delayed substantially. As seen in Fig. 5, the total runtime of the system under varying loads is consistently better when using the CPU scheduler, relative to the performance observed with the RR scheduler (for 1 min task inter-arrival time).

We conducted a second experiment that illustrates the difference between the task distribution on a round robin basis versus a distribution based on the criterion of disk access speed. We hypothesize that the former will result in degraded performance due to disk device contention for I/O-intensive tasks.

We used the same platform as we described in our previous experiment (1 MA, 2 LAs, and 6 SeDs), while the load is composed of tasks that executed multiple disk writes (according to the disk write speed to be simulated) and the same number of DGEMM calls. To truly test the hypothesis, we designed the tasks such that the disk access time was significantly greater than the DGEMM computing time.

To model a heterogeneous platform, the disk speed estimations retrieved by CoRI were scaled by a factor, and the number of disk writes performed by the associated service function was scaled accordingly.

The RR scheduler is identical to the one in the first experiment — a priority scheduler that maximizes the time elapsed since the last execution started (`EST_TIMESINCELASTSOLVE` CoRI tag) whereas the I/O scheduler is a priority scheduler that maximizes the disk write speed (`EST_DISKACCESSWRITE` CoRI tag). The behavior of the two schedulers was studied for requests spaced at different time intervals — here we focus on requests spaced at 35 s.

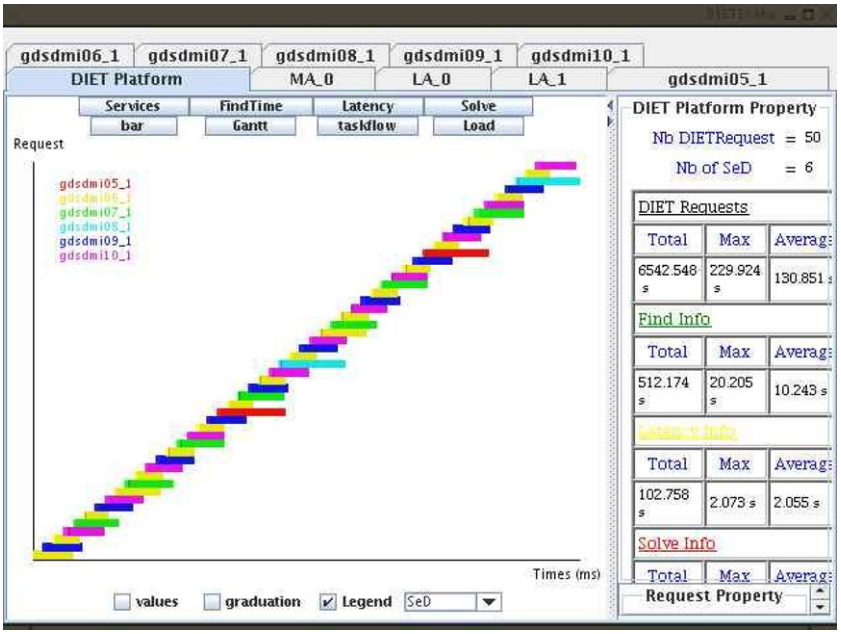
Fig. 6 presents the task distribution for 50 consecutive requests when using the two schedulers. The I/O scheduler allocates tasks mainly to the fastest nodes in terms of write disk speed with only 4 tasks out of 50 allocated to the slowest nodes (red and light blue in the figure), thus achieving an improvement ratio of 8%.

As seen in Fig. 7, the runtime on the platform is improved by using the I/O scheduler.

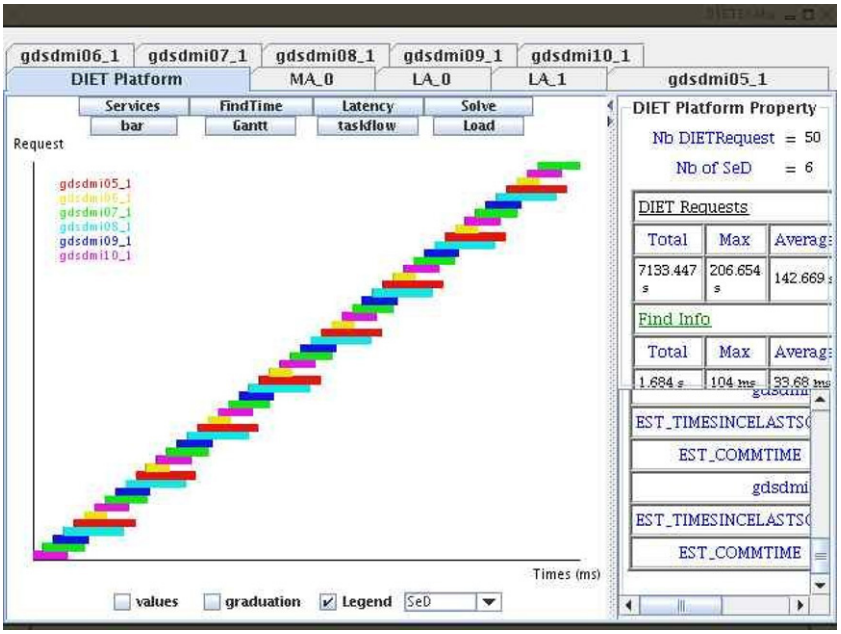
7. Conclusion

In this paper, we described the design of a plug-in scheduler in a grid environment, and we reported on an implementation of this design in the DIET toolkit. To provide the underlying resource performance data needed for plug-in schedulers, we designed a tool that facilitates the management of different performance measures and different kinds of resource collector tools. As a proof of this concept, we developed a basic resource information collector that enabled very basic performance data to be incorporated into DIET's scheduling logic.

We then conducted two experiments that illustrated the potential benefits of utilizing such information in realistic distributed computing scenarios. Though these performance results are not unexpected, the ease with which the experiments were conducted (i.e. based on disk access capacity, a fairly non-standard performance metric) is an indication of the utility of the resource performance gathering and plug-in scheduling facilities that we have implemented. Prior to their introduction, similar results were attainable on comparable (and indeed more complex) applications through the FAST performance prediction system. However, the framework described in this paper has a distinct advantage over the capabilities previously available to DIET application developers: the "barriers to entry" to realizing practical application-tuned scheduling have been dramatically lowered. Previously, for maximal



(a) I/O scheduler task distribution.



(b) RR scheduler task distribution.

Fig. 6. I/O vs RR scheduler — taskflow for 50 requests. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

efficacy FAST required that the application's performance be benchmarked for a range of parameters, even if the application execution behavior was well-understood. The plug-in facilities enable this information to be readily encoded with the application, eliminating the need for potentially expensive microbenchmarks. Moreover, the CoRI resource performance collection infrastructure enables middleware maintainers to closely assess and control the intrusiveness of the monitoring subsystem. By configuring CoRI to gather only that information that is needed for the applications that a specific DIET hierarchy is meant to support, excessive monitoring costs are effectively avoided. Collectively, these advantages represent a substantial improvement in DIET's application monitoring and scheduling capabilities.

The performance estimations provided for scheduling are suitable for dedicated resource platforms with batch scheduler facilities. In a shared resource environment, there may be differences between the estimations obtained at the SeD selection moment and the values existing when the client-SeD communication begins. However, even though in some cases

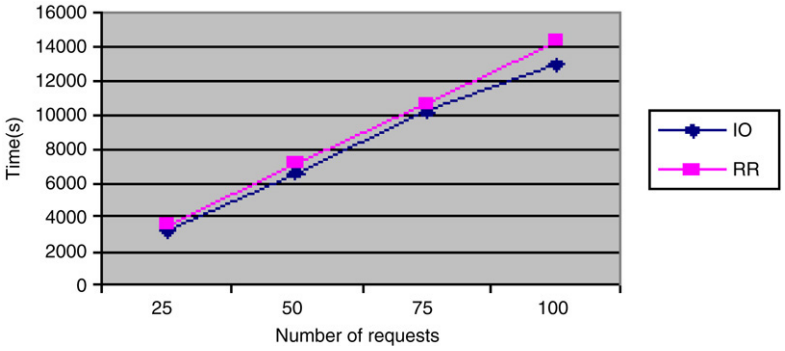


Fig. 7. I/O vs RR scheduler — total computation time.

the information could be marginally incorrect, an informed decision is preferable.

As future work, we plan to use the plug-in scheduler to design highly-tuned application-specific schedulers for several real applications that we have begun to study. We also intend to incorporate new collectors that seem promising in terms of providing resource performance data that may be useful in the DIET scheduling process.

Acknowledgments

We would like to give a very special thanks to Peter Frauenkron for his help on the design and implementation of CoRI. DIET was developed with financial support from the French Ministry of Research (RNTL GASP and ACI ASP) and the ANR (Agence Nationale de la Recherche) through the LEGO project referenced ANR-05-CIGC-11.

References

[1] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, R. Wolski, The GrADS project: Software support for high-level Grid application development, The International Journal of High Performance Computing Applications 15 (4) (2001) 327–344.

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

[3] H. Casanova, F. Berman, Parameter sweeps on the grid with APST, in: Grid Computing, Wiley Series in Communications Networking & Distributed Systems, Wiley, 2003.

[4] H. Casanova, G. Obertelli, F. Berman, R. Wolski, The appleS parameter sweep template: User-level middleware for the grid, Scientific Programming 8 (3) (2000) 111–126.

[5] H. Dail, O. Sievert, F. Berman, H. Casanova, S. YarKahn, J. Dongarra, C. Liu, L. Yang, D. Angulo, I. Foster, Scheduling in the grid application development software project, in: Grid Resource Management, Kluwer Academic Publisher, September 2003, pp. 73–98.

[6] A. Denis, C. Perez, T. Priol, Towards high performance CORBA and MPI middlewares for grid computing, in: C.A. Lee (Ed.), Proc. of the 2nd International Workshop on Grid Computing, in: LNCS, vol. 2242, Springer-Verlag, Denver, Colorado, USA, November 2001, pp. 14–25.

[7] M. Massie, B. Chun, D. Culler, The ganglia distributed monitoring system: design, implementation, and experience, Parallel Computing 30 (7) (2004) 817–840.

[8] H. Nakada, M. Sato, S. Sekiguchi, Design and implementations of ninf: towards a global computing infrastructure, Future Generation Computing Systems, Metacomputing Issue 15 (5–6) (1999) 649–658. http://ninf.apgrid.org/papers/papers.shtml.

[9] 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, in Conjunction with IPDPS'02, April 2002.

[10] R. Raman, M. Solomon, M. Livny, A. Roy, Scheduling in the grid application development software project, in: The ClassAds Language, Kluwer Academic Publisher, September 2003, pp. 255–270.

[11] D.A. Reed, C.L. Mendes, Intelligent monitoring for adaptation in grid applications, Proceedings of the IEEE 93 (2) (February 2005) 426–435.

[12] K. Seymour, C. Lee, F. Desprez, H. Nakada, Y. Tanaka, The end-user and middleware APIs for GridRPC, in: Workshop on Grid Application Programming Interfaces, in Conjunction with GGF12, Brussels, Belgium, September 2004.

[13] GRAAL Team. DIET User's Manual. http://graal.ens-lyon.fr/DIET.

[14] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: The condor experience, Concurrency — Practice and Experience 17 (2–4) (2005) 323–356.

[15] S. Vadhiyar, J. Dongarra, Self adaptability in grid computing, Concurrency and Computation: Practice and Experience 17 (2–4) (2005).

[16] R. Wolski, N.T. Spring, J. Hayes, The network weather service: a distributed resource performance forecasting service for metacomputing, Future Generation Computing Systems, Metacomputing Issue 15 (5–6) (1999) 757–768.

[17] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, J. Dongarra, Recent developments in gridsolve, International Journal of High Performance Computing Applications 20 (1) (2006) 131–141.

E. Caron is an assistant professor at Ecole Normale rieure de Lyon and holds a position with the LIP laboratory (ENS Lyon, France). He is a member of the GRAAL project and technical manager for the DIET software package. He received his Ph.D. in C.S. from the 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.

A. Chis is a master student at Ecole Normale Superieure de Lyon. She received her engineer diploma in Computer Science from the Technical University of Cluj-Napoca, Romania in 2006. Her main research interest is grid computing and scheduling for distributed platforms.

F. Desprez is a director of research at INRIA and holds a position at LIP laboratory (ENS Lyon, France). He received his Ph.D. in C.S. from the Institut National Polytechnique de Grenoble in 1994 and his M.S. in C.S. from the ENS Lyon in 1990. 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/~desprez for further information.

A. Su is currently a software engineer at Google Inc., with ongoing research interests in the fields of distributed computing and high-performance network infrastructure. Prior to joining Google, he was a post-doctoral researcher in the Laboratoire de l'Informatique du Parallélisme (LIP) at l'École Normale Supérieure de Lyon (ENS-Lyon) in Lyon, France, where he pursued research in the theoretical and practical aspects of grid computing. Alan obtained a Ph.D. in Computer Science and Engineering from the University of California, San Diego.