Hosting Services on the Grid: Challenges and Opportunities

Abhishek Chandra, Rahul Trivedi and Jon Weissman
Department of Computer Science and Engineering
University of Minnesota - Twin Cities
Minneapolis, MN 55455
{chandra, trivedi, jon}@cs.umn.edu

***Abstract*— In this paper, we present the challenges to service hosting on the Grid using a measurement study on a prototype Grid testbed. For this experimental study, we have deployed the bioinformatics service BLAST from NCBI on PlanetLab, a set of widely distributed nodes, managed by the BOINC middleware. Our results indicate that the stateless nature of BOINC presents three major challenges to service hosting on the Grid: the need to deal with substantial computation and communication heterogeneity, the need to handle tight data and computation coupling, and the necessity for handling distinct response-sensitive service requests. We present experimental results that illuminate these challenges and discuss possible remedies to make Grids viable for hosting emerging services.**

I. Introduction

Grid computing is finding application in an increasingly diverse set of domains including Biology [1], Physics [2], and Weather forecasting [3]. Existing Grids have been designed to provide best-effort service and rely on highly decentralized and uncoordinated computation. The most common metric used for Grid computing is throughput which is appropriate for very-long computations consisting of many tasks. Two dominant Grid environments are emerging: compute-oriented Grids in which the amount of computation per data element is relatively large such as in SETI@home, and data-oriented Grids in which the amount of computation per data element is small relative to the size of the data such as in PPDG [2]. In compute-oriented Grids, tasks can be widely dispersed irrespective of the original data-source, but in data-oriented Grids, tasks often must be directed to the data source due to its size to reduce data transfer overhead. Infrastructures have emerged in response to these different environments such as BOINC [4], iVDGL [5], and the virtual data toolkit [6].

Another parallel trend that is emerging in Grid and Internet computing is the use of service-oriented architectures such as Grid and Web services. Services represent an attractive option for out-sourcing elements of application development and execution, and as the resource demands of services continue to grow, we foresee a large number of services being transferred to the Grid. Hosting services on the Grid, however, represents a number of challenges. First, interesting services are often data-rich and would require computations to be data-aware despite the geographic separation of resources. Second, as Grid computing becomes outsourced, services would require some QoS similar to that provided in today's data centers and hosting platforms. Third, while current Grids work best with decentralized and long-running computations, many services would require more tightly-coupled computations with finite and predictable running times. The notion of a service request defines an explicit boundary between separate invocations of a service. The traditional metric of throughput commonly used in Grid computing, is likely inappropriate for the service-oriented model.

In this paper, we identify some of the challenges that Grid developers must tackle if the Grid is to become a viable hosting platform. We illustrate these challenges through a measurement study, and use the results of this study to discuss possible remedies for a service-oriented Grid design. For this study, we have used BOINC [4], an infrastructure which permits the use of geographically separated machines, that we have deployed on PlanetLab [7], a planetary-scale distributed testbed. We have used BLAST [8], an exemplar service in the domain of bioinformatics, as a test case since it represents emerging large-scale data-rich services. BLAST is ideally suited for our study as it not only requires a large amount of data transfer due to the geographic separation of nodes, but its popularity also suggests that multiple concurrent requests for service must be handled efficiently.

Our experimental results indicate that the stateless nature of BOINC presents three major challenges to service hosting on the Grid: (i) the need to deal with substantial computation and communication heterogeneity, (ii) the need to handle tight data and computation coupling, and (iii) the necessity for handling distinct response-sensitive service requests. We use the insights from our results to make recommendations to overcome these challenges.

The rest of this paper is organized as follows. Section II provides a background on general Grid computing models, along with a detailed description of the BOINC infrastructure. Section III outlines the various challenges anticipated in Grid design based on the characteristics of emerging services. We present the results of our measurement study in Section IV, followed by a discussion in Section V of possible remedies to overcome the illustrated challenges. We finally conclude with a summary and future work in Section VI.

II. Background

A. Grids

A number of Grid computing models have emerged over the past few years. These models have different features which both support as well as hinder Grid service hosting. We categorize these models across a number of axes based on how the Grid resources are organized:

1) *Sharing*: The first axis is the degree of Grid resource sharing: Grid resources may be dedicated exclusively to Grid jobs or shared with non-Grid local jobs and users.
2) *Stability*: Stability of a Grid reflects the persistence of Grid resources: Grid resources may be static (established at Grid boot-time and unchanging) or dynamic (resources can both join and leave the Grid).
3) *Interaction*: The third axis is the degree of interaction or communication between Grid resources: a direct interaction allows communication between the Grid nodes while some Grids support only indirect interaction, possibly through a centralized server.
4) *Coupling*: The fourth axis we consider is geographic coupling: a coupled Grid contains only closely connected resources (such as within the same network or campus), while a non-coupled Grid contains wide-area loosely-connected resources (such as over the Internet).
5) *Execution model*: The final axis is the execution model: a fixed Grid is designed only to run a few (perhaps one) computations while a non-fixed execution model allows more general job submission.

Note that while individual Grids might lie along a continuum on these axes, we have presented only the “end-points” of each axis for simplicity. For example, the geographical coupling between Grid resources may cover a wide range of options (e.g., wide-area clusters). Our categorization does not exclude such point solutions. Some examples of deployed Grids characterized along these axes are as follows:

1) Sharing: dedicated (TeraGrid [9]), non-dedicated (DOE PPDG [2], Npaci-net [10][1])
2) Stability: static (UWisc-Condor-Grid [11][2]), dynamic (xyz@home [12][3])
3) Interaction: direct (TeraGrid), indirect (xyz@home)
4) Coupling: coupled (UWisc-Condor-Grid), non-coupled (xyz@home)
5) Execution model: fixed (xyz@home), non-fixed (UWisc-Condor-Grid)

Supporting these Grid computing models are differing Grid middleware infrastructures (we list the primary infrastructure as in some cases several are used): Globus [13] supports DOE PPDG [2] and the TeraGrid [9], Condor [14] the UWisc-Condor-Grid [11], BOINC [4] supports many of the xyz@home donation-based Grids [12], and Legion/Avaki [15] supports Npaci-net [10].

In this paper, we study the problem of hosting services on the Grid—its potential and limitations—across the more common yet more challenging Grids: non-dedicated, dynamic, indirect, and non-coupled. The primary question we explore is: how well does this kind of Grid support response-oriented high-performance scientific services, and what are the enablers and bottlenecks? To answer these questions, we use BOINC “out-of-the-box” as it supports this style of Grid. We have installed BOINC on the wide-area PlanetLab [7] infrastructure. Our experimental Grid consists of a slice of this network. We have also deployed the widely used BLAST genome sequence comparison service in BOINC as a test case. Because BLAST operates over large sequence databases which do not exist a priori at the wide-area Grid nodes, it represents an excellent stress test for BOINC. A related project, Lattice [16], is providing infrastructure to support scientific Grid services based on BOINC, but is geared to more throughput-oriented batch services.

[1]Npaci-net is no longer operational, but included here for completeness.
[2]Condor users can take away resources but the Condor network does not grow.
[3]xyz is meant to denote any of the many donation-based Grid applications.

B. Grid Example: BOINC

BOINC (Berkeley Open Infrastructure for Network Computing) [4], [17] is a software platform for distributed public Grid computing. BOINC goes beyond applications such as SETI@home by allowing the resource pool to serve any number of installed projects. A donated node may select the set of projects it wishes to serve. Each project is managed by a separate BOINC server. The server environment of a BOINC project consists of scheduling and data handling servers. The scheduling server handles clients, it delivers work and receives the completed results. Data handling servers transfer input and output files to and from the workers via HTTP.

Tasks are represented in BOINC using ‘workunits’. A workunit represents the inputs to a computation: the application code to run, references to dependent input files, command line arguments and environment variables. Each workunit has parameters specifying the compute, memory and storage requirements and a soft deadline for completion. A ‘result’ represents the final output of a computation consisting of the associated workunit and a list of output files. The workers poll the server for workunits periodically when idle. When there is no work, the workers “back off” until a time-out period. At project deployment time, workunits are created at the server and stored in a MySQL database. A ‘feeder’ daemon picks up these workunits from the database and makes them available to the scheduler for dispatching to workers on demand. The BOINC server environment is stateless with respect to the donated resources, an issue we will return to later.

III. Challenges in Grid Service Hosting

As discussed in the previous section, existing Grid architectures such as BOINC, have been successfully employed for running large-scale applications with embarrassingly parallel computations. In this section, we present some of the important challenges faced by such Grid architectures for hosting services that have vastly different characteristics from traditional Grid applications.

A. Heterogeneity

A non-dedicated and distributed Grid such as in a typical deployment of BOINC is characterized by its use of multiple nodes with widely varying computational capabilities. Different worker nodes in such an infrastructure typically have different CPU speeds, memory and disk capacities. Moreover, nodes have different connection speeds and their bandwidth to the server node is also dependent on their geographical location. Such disparity is not a problem for traditional Grid applications such as SETI@home, as these applications are continuously running and throughput-intensive. Thus, Grids such as BOINC have been designed to ignore the inherent heterogeneity of the infrastructure: for instance, the BOINC server maintains no state information about the individual node capabilities, and workload handed out to different worker nodes is generally independent of their actual computational power or communication quality to the server[4].

Such a heterogeneity-agnostic design could have serious implications for hosting services on Grids. As a service typically has end-clients submitting requests, there is a need to bound the request completion times. Such requirements can be met more efficiently by smart exploitation of the inherent infrastructure heterogeneity. For instance, if the request completion time of a request is dependent on the slowest worker in a parallel computation, the computation should be divided among the workers according to their individual capabilities (e.g., by assigning more work to faster workers with faster communication paths to the server).

In the current BOINC architecture, the granularity of work allocation is divorced from the individual node capabilities, so that similar amount of work could be allocated to nodes with very different capacities. However, *a service-oriented architecture would require incorporating knowledge of the infrastructure heterogeneity in the Grid's division of labor.*

B. Data Dependency

Due to their distributed nature, Grids have traditionally been employed to perform large-scale compute-oriented parallel computations. These applications are typically characterized by embarrassingly parallel computations that rely on relatively small input data sets, thus obviating the need of any large data transfers or communication between the server and worker nodes. Therefore, the computational power of worker nodes has been the primary factor influencing the performance of Grid applications.

However, many emerging services are characterized by their dependence on large quantities of data. For instance, many bioinformatics services require access to large genomic databanks to perform meaningful gene comparisons. Similarly, Physics and weather prediction services rely on terabytes of data being collected continuously by instruments and sensors all around the world. Hosting such services on large Grids raises several novel challenges. First, parallelizing the work into separate decoupled tasks becomes more difficult due to the inherent data dependency of these services. Second, if the server has to transfer large amounts of data to the worker nodes, the server-worker network bandwidth becomes an important factor in addition to the worker's computational resources. Moreover, to avoid large overheads in server-worker interactions, computation may have to be colocated with the data, which could require careful selection of worker nodes in close proximity to the data store.

Thus, *the data-rich nature of emerging services would create new bottlenecks in the execution path,* and would require careful design of the Grid architecture and workload distribution.

C. Request-Oriented Service Architecture

Traditional Grid applications such as SETI@home are essentially never-ending computations that have no completion boundaries. Thus, different units of the computation just contribute to incrementally building up partial results, and there is no (or only a weak) notion of a final result. Such an application architecture implies that the Grid does not have to distinguish between different workunits in terms of their allocation or result processing. Thus, Grid architectures like BOINC maintain little state about individual workunits and treat them equally from the application perspective.

However, a hosted service would be characterized by separate requests being submitted by end-users, and these requests would have to be executed concurrently on the Grid. The Grid would then require mechanisms to differentiate between workunits of different requests and collate their results separately. For instance, the server would have to maintain some state regarding the execution status of each individual request (e.g., its component workunits, completed partial results, workers allocated to the request, etc.). At the same time, the interaction between the server and the worker nodes would have to be modified by building in more awareness of request-oriented nature of the hosted service. For instance, the server must be able to push out work for new requests as soon as possible to reduce the request execution time.

In other words, *a Grid would have to be aware of the request-oriented nature of its hosted services to ensure correctness and efficiency.*

Having outlined some of the main challenges in Grid service hosting, we next present a measurement study that further explores and illuminates these challenges.

IV. Experimental Results

In this section, we quantitatively explore the impact of the various factors described above on a real service in a prototype Grid implementation. We first describe our experimental testbed and prototype details, followed by performance results obtained from the testbed.

A. Prototype Testbed

We run our Grid prototype on PlanetLab [7]—a shared, distributed infrastructure consisting of donated machines. Our testbed consists of 9 PlanetLab nodes, one of which is set up as the Grid server and the remaining 8 as the Grid worker nodes. Table I shows the location, and CPU and memory specifications of these PlanetLab nodes. Each node runs the Fedora Core 2 Linux 2.6.8 kernel, and has 5 GB of disk space. As can be seen from the table, the nodes have varying

[4]In typical BOINC deployments, the only differentiation between workers is based upon statically configured user preferences.

Server		
Location	CPU	Memory
UCSD	Pentium-III 1.26GHz	1GB

Workers		
Location	CPU	Memory
UCSB	Pentium-4 1.80GHz	2GB
Princeton	Pentium-III 1.26GHz	1GB
UWashington	Pentium-III 1.26GHz	1GB
Intel Berkeley	Pentium-III 1.26GHz	1GB
UNC-Chapel Hill	Pentium-4 2.80GHz	1GB
UMich-Ann Arbor	Pentium-4 1.80GHz	2GB
UC-Berkeley	Pentium-4 3.06GHz	1.1GB
Columbia Univ.	Pentium-III 1.26GHz	1GB

TABLE I
PLANETLAB TESTBED.

hardware capabilities and are geographically distributed. We used the BOINC stable (boinc_public) version 4.19 to set up our Grid prototype on the PlanetLab testbed.

We modified a standalone bioinformatics application called BLAST (Basic Local Alignment Search Tool) [8] to run as a service on top of our Grid prototype. BLAST is an algorithm for rapid searching of DNA and protein databases. The BLAST algorithm compares novel DNA sequences with previously characterized genes, and is used to identify the function of newly discovered proteins. BLAST takes an input sequence and compares it to a formatted database file and generates an output file containing a similarity score and similarity matches with the database.

In our setup, BLAST is modified to run as a BOINC project: it is hosted on the BOINC server which hands out the application workunits to the worker nodes for computation. In this setup, the BLAST executable is kept unmodified and a BOINC-specific wrapper is written around it. A workunit consists of an input sequence and a portion of the BLAST database provided as input files. The result of each workunit execution is an output file containing a similarity score generated by the BLAST code. The BLAST computation at each worker node is performed in two steps. The first step consists of formatting the database using a BLAST command 'formatdb', after which the actual sequence comparison is performed to yield a result file. In our BOINC setup, the results are sent back to the server which merges them together into a single output file. We used a 77 MB formatted file of sequences (drosoph.nt) available on the NCBI site [8] as the BLAST database of gene sequences. The input sequence used for comparison was a randomly-selected sequence from the database; the input sequence length was 569 bytes. The BOINC workunits for the BLAST service were generated by splitting the database into equal-sized chunks (e.g., 8 chunks for 8 workers).

Our BLAST project setup differs from standard BOINC projects in one crucial aspect. Unlike existing BOINC projects such as SETI@home, where all workunits contribute towards a single overall computation, our BLAST project is request-oriented. We define a BLAST service request to be a complete BLAST computation for *a specific input sequence* using the whole BLAST database. Thus, once the results for a single input sequence are collated from multiple worker nodes, we assume a BLAST request to have completed. Subsequent computations are considered to be new requests and handled separately. We could also have multiple requests (corresponding to multiple input sequences) executing concurrently in our setup, however, here we present results only from non-concurrent request execution to illustrate other challenges more clearly.

We conducted our experiments by executing multiple (non-concurrent) BLAST requests on our testbed, and measuring the total request execution times along with their component costs such as computation and communication times at each worker node. We now present our experimental results and discuss their implications on Grid design.

B. Impact of Heterogeneity

Figures 1(a) and (b) show the computation and communication time taken by different worker nodes over multiple request executions[5]. These graphs show that the execution times vary across nodes as well as across runs on the same nodes. Figures 2(a) and (b) depict the same results by plotting the average computation and communication time for multiple runs. The error bars in these figures represent the standard deviation of the time across nodes for each run. We make several interesting observations from these figures. First, we see that *the inter-node variation in computation time is much larger than that in the communication time.* For instance, while the standard deviation of the computation time varies between 14.0-23.1 seconds for the different runs, the standard deviation for the communication time is much smaller (about 7.4-18.0 seconds). Second, we observe that while the average computation time is relatively stable across runs (25.5 - 32.2 seconds), the average communication time shows large variations (ranging from 37-95.5 seconds). Moreover, as can be seen from Figures 1(b) and 2(b), the communication times are highly correlated across nodes.

Figures 3(a) and (b) plot the average per-node computation time and communication time over multiple runs along with the 95% confidence intervals. Figure 3(a) clearly shows the wide diversity in the computational capability of different nodes. For instance, while node 1 takes only about 9.4 seconds on average for its computation, node 8 takes about 53.2 seconds. However, Figure 3(b) shows that the average communication times taken by different worker nodes is much closer.

Another interesting observation we make from Figures 2(a) and 3(a) is the difference between the inter-node vs. the intra-node variation in computation time. The large values of standard deviation from Figure 2(a) indicate a large inter-node variation in computation time even over the same run, while the tight confidence intervals in Figure 3(a) imply small intra-node variation even across multiple runs. This difference can

[5]We use the terms "request" and "run" interchangeably in the discussion of our results.

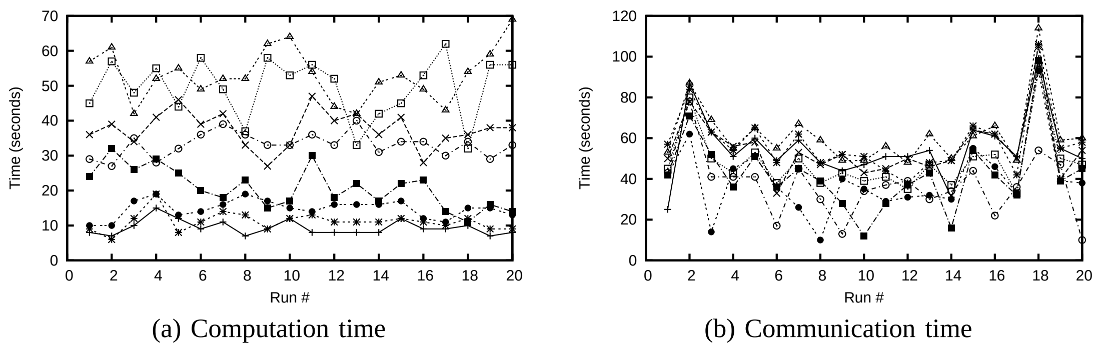

Fig. 1. The computation and communication time taken by different worker nodes over multiple runs. Each curve corresponds to a single worker node.

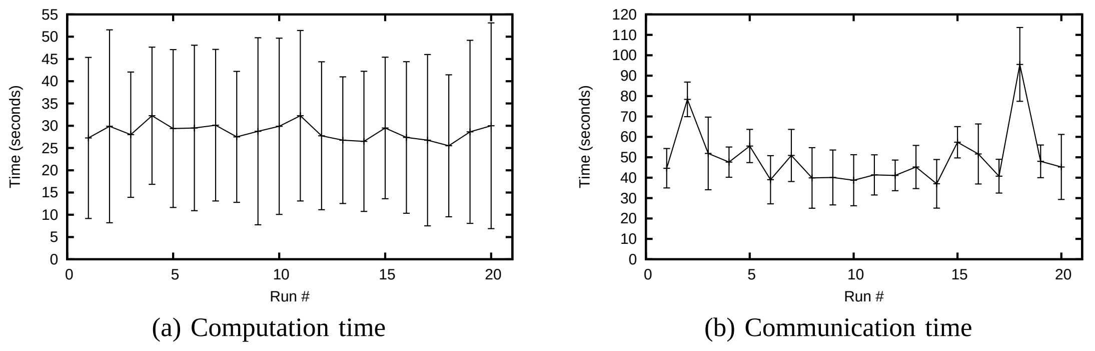

Fig. 2. The variability in the computation and communication time over multiple runs. The error-bars represent the standard deviation of the times measured across nodes for each run.

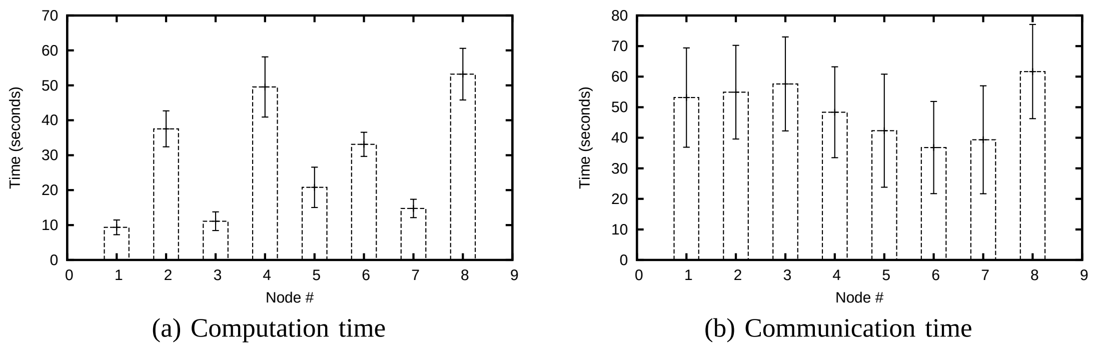

Fig. 3. The average computation and communication time taken by different worker nodes. The error-bars represent 95% confidence intervals.

also be observed visually from Figure 1(a), where the variations in individual curves (corresponding to individual nodes) is less pronounced than the differences between different curves. These results imply that *the service performance within a node is relatively stable over time, compared to the inter-node variations within each run.* This observation suggests that it might be easier to distinguish between different node capabilities, thus exploiting Grid heterogeneity without having to worry about dynamic intra-node performance fluctuations.

These results together illustrate the level of heterogeneity in the Grid. These results show that while the computation time on a node is inherently dependent on its computational capability, the communication time to different nodes may be dependent on a bottleneck resource close to the server (e.g., a common link or server disk or CPU).

The heterogeneity in computational power of different nodes suggests that the workload should be divided among them according to their capacity. To do this, we break the BLAST database into larger number of chunks of correspondingly smaller sizes. The goal is to have faster workers pick up more workunits, thereby decreasing the overall request execution time. Figure 4 shows the variation in the total request execution time as each request is broken up into 8, 16, 24 and 32 workunits respectively. The figure shows that while the total request time decreases initially from about 128.5 to 115.4 seconds as the number of workunits is increased from 8 to 16 (corresponding to a decrease in the workunit granularity), the total execution time flattens out beyond that. This result

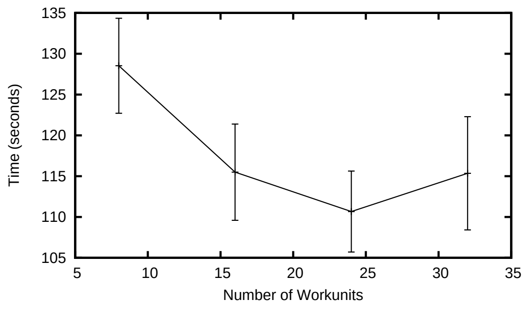

Fig. 4. Service performance with varying workunit sizes.

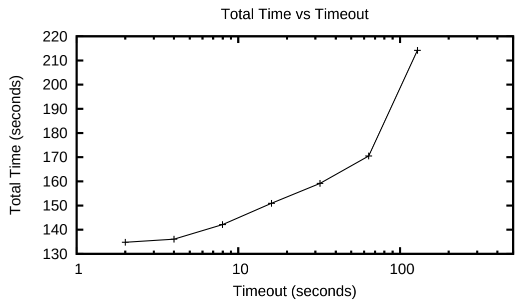

Fig. 6. Effect of the worker timeout value on the request service time.

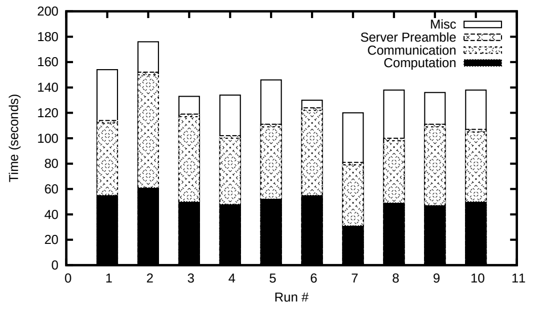

Fig. 5. The breakdown of request execution into its component costs.

shows that while decreasing the workunit granularity leads to better load balancing, there is an inherent limit beyond which the overheads of server-worker interactions start dominating. We note that Figure 4 corresponds to a specific database size, and different services and request would have different cross-over points.

Overall, we see from our results that *heterogeneity is inherent in Grid nodes, and this heterogeneity has significant impact on the service performance*.

C. Impact of Data Dependency

Figure 5 shows the breakdown of the request execution time into its components for multiple runs. The main components shown in the figure are the request setup time at the server (the server preamble), the data transfer or communication time between the server and the workers (the bottleneck node here), and the computation time at the nodes (the bottleneck node). The remaining (miscellaneous) time corresponds to other execution costs such as the time taken by the workers to pick up the workunits, the time to collate results, etc.

As shown in Figure 5, we see that the communication time forms a significant proportion (about 42.3% on an average) of the total execution time. This result shows the high cost of data transfer for each workunit, and shows the impact of high data dependency for our example service. This result also indicates an inherent limitation of the BOINC architecture which has been primarily designed to support compute-intensive applications.

Overall, this result shows the impact of data dependency on the request service time. We see that *communication has a substantial overhead because of transfer of large amount of data along with the workunit*, and hence, it is important to factor in communication cost along with the computational capability of a node while allocating work.

D. Impact of Request-Oriented Service Architecture

BOINC is a purely pull-based Grid architecture, where workers request work from the server whenever they are idle. However, if the server has no new work to allocate, the workers set a timeout value (default=60 minutes), and wait for the timeout period before requesting new work again. Traditional applications using BOINC are throughput-intensive and assumed to have a forever-running model. In other words, the application is always assumed to have useful work available at the server, and the above scenario of a worker having to sit idle waiting for server to generate useful work, is a rare occurrence.

However, in the case of a service executing multiple distinct requests, each request has a finite request boundary and the server may run out of work if it has no pending requests. In such a scenario, the timeout value of the workers would critically determine the request service latency. Figure 6 shows the impact of the worker timeout value on the request service time[6]. As can be seen from the figure, as the timeout value is increased towards the default (rarely-used) value of BOINC, it starts dominating the turn-around time of the requests irrespective of the actual computation and communication costs.

Note that this effect happens purely because of the request-oriented nature of our BLAST service, and would have little impact on a single, continuously-running application. This result implies that *the conventional design of BOINC (with default parameter settings), which does not distinguish between different request boundaries, is not suited for hosting a request-oriented service.*

Overall, the results presented in this section reinforce our contention that the existing design of Grid architectures is inadequate for hosting new services that diverge from conventional Grid applications.

[6]In the previous results, we have adjusted the time-outs to their smallest possible value of 1 second

V. Recommendations for Grid Service Hosting

Based on the insights gained from our experimental results in the previous section, we now discuss some remedies and present some recommendations to make Grids viable for hosting services.

A. Recommendation 1: Exploit Heterogeneity

As we saw in the previous section, different nodes in a large-scale Grid have vastly different computational capabilities. In order to host services successfully, the Grid must be designed not only to handle but to *exploit* this heterogeneity. The main way to exploit heterogeneity is by matching the work allocation to individual node capabilities. For instance, for our BLAST service, larger workunits could be allocated to faster machines. Such work allocation would ensure the maximum parallelism from the Grid, resulting in reduced request execution latency.

There are several strategies that could be adopted to achieve such intelligent work allocation. One strategy could be to divide the work into small units, thus enabling faster workers to pick up more workunits and thus performing more work (as illustrated in Figure 4 in the previous section). For instance, with our BLAST example, we could divide the BLAST database into small chunks, each consisting of a single gene sequence, and let faster workers perform more comparisons by picking up more workunits. This strategy raises several challenges. For instance, making the workunits too small may result in increased communication overhead. Further, the workunit size may be limited by the service's internal structure and data dependencies, and it may be difficult to determine the suitable workunit size a priori.

Another strategy could be for the server to determine individual node capabilities through either active probing or passive monitoring (e.g., based on their history of earlier responses). The server could then allocate different size workunits to different workers based on their observed capabilities. This strategy would require maintaining more state at the server, and also possible re-sizing of workunits according to individual node capabilities.

We expect the first strategy of fine-grained work allocation to be more easily deployable in existing Grid architectures, as it requires less server state. We are investigating techniques to dynamically determine suitable workunit sizes which provide a good trade-off between communication overheads and granularity of work allocation.

B. Recommendation 2: Couple Data and Computation

As seen from the previous section, data transfer and communication times form a significant proportion of the total request execution time for data-rich services such as BLAST. These services do not satisfy the assumption of large computation units with small input data sets. Rather, these services have a strong data dependency and data access and transfer constitute a substantial part of the execution path. Therefore, smarter strategies need to be designed to couple the service data with request computation.

One data-aware strategy could be to cache service data at the workers, so that subsequent computations on cached data could avoid data retransmission. For instance, for our BLAST service, each worker node could cache the portion of the genome database that it receives with its initial workunits, and use the cached data to execute subsequent workunits. Such a strategy could save substantial communication overhead between the server and workers. BOINC supports the notion of "sticky" files that are not deleted by the workers after a workunit execution. However, the use of such sticky files is currently limited to those applications that have the *whole input data* replicated at *all* the workers, which is infeasible for highly data-rich applications such as BLAST. Thus, efficient exploitation of data caching would require the server to be aware of the location of the data caches, so that it could direct specific workunits to the workers with the relevant cached data. Such a caching strategy would also require cache invalidations and updates if data changes dynamically. However, since data caching is only a performance-enhancing heuristic, such invalidations and updates do not have to be performed synchronously, thus avoiding high synchronization overheads and any impact on correctness.

Another data-aware strategy could be to allocate work to computational nodes in close proximity to the service data. Such a strategy could reduce the data transfer and caching overhead, but would require the server to be aware of the geographical locations of the data and the computational nodes.

We expect both of these strategies to be useful for data-intensive services. The first strategy of data caching could be implemented in existing Grid architectures by adding small amounts of caching state at the server and the workers. The second strategy of data-aware routing of workunits would require the knowledge of the Grid topology. We are currently investigating the benefits and shortcomings of each strategy.

C. Recommendation 3: Provide Request Differentiation

As seen from the previous section, ignorance of request boundaries could result in degraded performance of a hosted service. Thus, the Grid must be made aware of multiple concurrent requests executing in a hosted service environment. Each service request must be identified separately, and its execution must be differentiated from that of other requests. Moreover, the Grid server as well as the workers must be aware of request boundaries so that they can easily transition their state to handle new requests.

Such request awareness would require maintaining state at the server about request boundaries as well as individual request data and workunits. The server must be able to differentiate between different concurrent requests, and keep track of their partial results and pending workunits. The workers also have to be request-oriented, in that the absence of work at the server should be treated as a normal occurrence and not an exception. The server must have a mechanism to signal the arrival of new requests or completion of existing requests. Moreover, the server must have the capability to provide differentiated service to different requests, e.g., by allocating more resources to important requests.

We are designing basic mechanisms that would allow concurrent requests to be executed in a Grid. We envision request differentiation to be implemented in a Grid by explicitly tagging workunits and maintaining their state at the server. Moreover, the service could be allowed to define the policies for providing request differentiation in the Grid.

Overall, *we recommend building more state into the Grid to overcome the multiple challenges described in Section III.* Some of this state could be easily introduced into existing Grid designs with small modifications to built-in mechanisms and parameters. Adding other state would require more substantial overhauls to the components and mechanisms currently existing in the Grid architecture. However, *we recommend introducing additional state mainly as an optimization and not as a critical functional component.* The state should be "soft" and easily removable or discardable to avoid high overheads and consistency requirements. Moreover, the state should be mainly used to provide hints for better decisions, and the Grid should be able to function smoothly even in the absence or corruption of this additional state. We believe that careful introduction of meaningful state into Grid design will enhance the performance of services substantially, and will make the Grid not only viable but also desirable for hosting services.

VI. Conclusions and Future Work

We have identified the key obstacles for Grid infrastructures to become viable as hosting platforms for data-rich services. We have performed our analysis using a measurement study with the BOINC Grid middleware running on top of the shared PlanetLab infrastructure in the context of the BLAST application. Our results indicated that substantial resource heterogeneity exists and impacts the number, size, and allocation of workunits. The data-rich nature of BLAST posed additional challenges as over 40% of the service time was due to communication of target library database chunks to workers. We also found that BOINC was inappropriately configured for response-oriented concurrent service requests due to excessively high time-out parameters and the absence of per-request workunit state. Despite these issues, the Grid-based BLAST service can still out-perform a smaller cluster-based counterpart as shown in earlier work [18], but it can do much better once these obstacles are removed. These obstacles arise, in part, due to the stateless nature of BOINC. We believe all of these challenges can be overcome with careful addition of state layered on top of the existing BOINC infrastructure and we are in the process of defining such an architecture. We envision state at both the server (worker node state, workunit execution profiling) and worker (data caches, node hierarchies) levels. Once completed, we believe that the range of applications and services open to the Grid will be much greater and not limited to the "low-hanging fruit" of embarrassingly parallel compute-intensive and throughput-oriented applications.

References

[1] "Genome@home," http://www.stanford.edu/group/pandegroup/genome.

[2] "PPDG: Particle Physics Data Grid," http://www.ppdg.net.

[3] "Climateprediction," http://www.climateprediction.net.

[4] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, Nov. 2004, pp. 4–10.

[5] "iVDGL: International Virtual Data Grid Laboratory," http://www.ivdgl.org.

[6] "The Virtual Data Toolkit," http://www.cs.wisc.edu/vdt.

[7] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating System Support for Planetary-Scale Network Services," in *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI'04)*, Mar. 2004.

[8] "BLAST," http://www.ncbi.nlm.nih.gov/blast.

[9] "TeraGrid," http://www.teragrid.org.

[10] "NPACI-NET," http://legion.virginia.edu/npacinet.html.

[11] "Condor: The Compute Cluster Home Page," http://www.cs.wisc.edu/condor/cluster.

[12] "BOINCstats," http://www.boincstats.com.

[13] "Globus," http://www.globus.org.

[14] M. Litzkow, M. Livny, and M. Mutka, "Condor-A hunter of idle workstations," in *Proceedings of the 8th International Conference On Distributed Computing Systems*, June 1988, pp. 104–111.

[15] A. Grimshaw, W. Wulf, and the Legion team, "The legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, pp. 39–45, Jan. 1997.

[16] "The Lattice Project," http://lattice.umiacs.umd.edu.

[17] "BOINC: Berkeley Open Infrastructure for Network Computing," http://boinc.berkeley.edu.

[18] R. Trivedi, A. Chandra, and J. Weissman, "Platform-of-Platforms: A Modular, Integrated Resource Framework for Large-Scale Services," in *Poster Session of the Second USENIX Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.