Design and Evaluation of a Resource Selection Framework for Grid Applications

Chuang Liu* Lingyun Yang* Ian Foster*# Dave Angulo*[1]

Abstract

While distributed, heterogeneous collections of computers ("Grids") can in principle be used as a computing platform, in practice the problems of first discovering and then configuring resources to meet application requirements are difficult problems. We present a general-purpose resource selection framework that addresses these problems by defining a *resource selection service* for locating Grid resources that match application requirements. At the heart of this framework is a simple but powerful declarative language based on a technique called set matching, which extends the Condor matchmaking framework to support both single resource and multiple resource selection. This framework also provides an open interface for loading application-specific mapping modules to personalize the resource selector. We present results obtained when this framework is applied in the context of a computational astrophysics application, Cactus. These results demonstrate the effectiveness of our technique.

1 Introduction

The development of high-speed networks (10 Gb/s Ethernet, optical networking) makes it feasible, in principle, to execute even communication-intensive applications on distributed computation and storage resources. However, the discovery and configuration of suitable resources for applications in heterogeneous environment remain challenging problems. Like others [1-5], we postulate the existence of a *Resource Selector Service* (RSS) responsible for selecting Grid resources appropriate for a particular problem run based on that run's characteristics; organizing those resources into a virtual machine with an appropriate topology; and potentially also assisting with the mapping of the application workload to virtual machine resources. These three steps—*selection*, *configuration*, and *mapping*—can be interrelated, as it is only after a mapping has been determined that the selector can determine whether one selection is better than another.

Many projects have addressed the resource selection problem. Systems such as NQE [6], PBS [7], LSF [8], I-SOFT [9], and Load Leveler [10] process user-submitted jobs by finding resources that have been identified either explicitly through a job control language or implicitly by submitting the job to a particular queue that is associated with a set of resources. This manually configured queue hinders the dynamic resource discovery. Globus [11] and Legion [12], on the other hand, present resource management architectures that support resource discovery, dynamical resource status monitor, resource allocation, and job control. These architectures make it easy to create a high-level scheduler. Legion also provides a simple, generic default scheduler. But Dail et al. [13] show that this default scheduler can easily be outperformed by a scheduler with special knowledge of the application.

The AppLeS framework [2] guides the implementation of application-specific scheduler logic, which determines and actuates a schedule customized for the individual application and the target computational Grid at execution time. Dongarra et al. developed a more modular resource selector for a ScaLAPACK application [1]. Since they embed the application-specific detail in the resource selection module, however, their tools cannot easily be used for other applications. Systems such as MARS [14], DOME [15], and SEA [16] target particular classes of application (MARS and SEA target applications that can be

[1] * Department of Computer Science, The University of Chicago, Chicago, IL 60637, USA.
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA.
Email: {chliu, lyang, foster, dangulo}@cs.uchicago.edu

represented by dataflow-style program graph, and DOME targets SIMD applications). Furthermore, neither the user nor the owner of resources can control the resource selection process in these systems.

Condor [3] provides a general resource selection mechanism based on the *ClassAds language* [17], which allows users to describe arbitrary resource requests and resource owners to describe their resources. A *matchmaker* [18] is used to match user requests with appropriate resources. When multiple resources satisfy a request, a ranking mechanism sorts available resources based on user-supplied criteria and selects the best match. Because the ClassAds language and the matchmaker were designed for selecting a single machine on which to run a job, however, it has limited applicability in the situation where a job requires multiple resources.

To address these problems, we define a *set-extended ClassAds Language* that allows users to specify aggregate resource properties (e.g., total memory, minimum bandwidth). We also present an extended *set matching* matchmaking algorithm that supports one-to-many matching of set-extended ClassAds with resources. Based on this technique, we present a general-purpose resource selection framework that can be used by different kinds of application. Within this framework, both application resource requirements and application performance models are specified declaratively, in the ClassAds language, while mapping strategies can be determined by user-supplied code. (An open interface is provided which allows users to load the application specific mapping module to customize the resource selector.) The resource selector locates sets of resources that meet user requirements, evaluates them based on specified performance model and mapping strategies, and returns a suitable collection of resources, if any are available. We also present results obtained when this technique was applied in the context of a nontrivial application, Cactus [19, 20].

This paper is organized as follows: In Section 2, we present the set-extended ClassAds language and the set matching mechanism. In Section 3, we describe the resource selector framework. In Section 4, we describe a performance model and mapping strategy of the Cactus application used in our case study. Experimental results are presented in Section 5. Finally, we summarize our work and briefly discuss future activities.

2 Set-Extended ClassAds and Set Matching

We describe here our *set-extended ClassAds language* and *set-matching* algorithm.

2.1 An Overview of Condor ClassAds and Matchmaking

A ClassAd (Classified Advertisement) [17] is a mapping from *attribute names* to *expressions.* Attribute expressions can be simple constants or a function of other attributes. A protocol is defined for *evaluating* an attribute expression of one ClassAd with respect to another ClassAd. For example, the expression "other.size > 3" in one ClassAd evaluates to **true** if the other ClassAd has an attribute named "size" and the value of that attribute is an integer greater than three. ClassAds can be used to describe arbitrary entities. In the current context, they are used to describe resources and user requests.

Conventional Condor matchmaking [18] takes two ClassAds and evaluates one with respect to the other. Two ClassAds *match* if each ClassAd has an attribute named "requirements" that evaluates to **true** in the context of the other ClassAd. A ClassAd can also include an attribute named "rank" that evaluates to a numeric value representing the quality of the match. When matchmaking is used for resource selection, the matchmaker evaluates a ClassAd request with every available resource ClassAd and then selects a resource that both matches the request and returns the highest rank.

2.2 Set-Extended ClassAds Syntax and Set Request

In set matching, a successful match is defined as occurring between a single *set request* and a *resource set*. The essential idea is as follows. The set request is expressed in set-extended ClassAds syntax, which

is identical to that of a normal ClassAd except that it can indicate both *set expressions*, which place constraints on the collective properties of an entire resource ClassAd set (e.g., total memory size) and *individual expressions*, which must apply individually to each resource in the set (e.g., individual per-resource memory size). The set-matching algorithm attempts to construct a resource set that satisfies both individual and set constraints. This set of resources is returned if the set match is successful.

2.2.1 Set-Extended ClassAds Syntax

The set-extended ClassAd language, as currently defined, extends ClassAds as follows:

- A `Type` specifier is supplied for identifying set-extended ClassAds: the expression `Type="Set"` identifies a set-extended ClassAd.
- Three *aggregation functions*, `Max`, `Min`, and `Sum`, are provided to specify aggregate properties of resource sets.
- A boolean function `Postfix(V, L)` is defined that returns **`True`** if a member of list `L` is the postfix of scalar value `V`.
- A function `SetSize` is defined that can be used to refer to the number of elements within the current resource set.

Three aggregation functions are as follows.

- `Max(`*`expression`*`)` returns the maximum value returned by *`expression`* when applied to each of the ClassAds in a set.
- `Min(`*`expression`*`)` returns the minimum value of *`expression`* in a set of ClassAds.
- `Sum(`*`expression`*`)` returns the sum of the values returned by `expression` when it is applied to each of the ClassAds in a set. For example, `Sum(other.memory)>5G` means the total memory of the set of resources selected should be greater than 5G.

Aggregation functions might be used as follows. If a job consists of several independent subtasks that run in parallel on different machines, its execution time on a resources set is decided by the subtask that ends last. Thus, we might specify the rank of the resource set to be `Rank=1/Max(execution-time)`, which means that the rank of the resource set is decided by the longest subtask execution time.

A user can use the `Postfix` function to constrain the resources considered when performing set matching, to those within particular domains. For example, `Postfix(H, {"ucsd.edu", "utk.edu"})` returns **`True`** for `H="torc1.cs.utk.edu"` because `"utk.edu"` is the postfix of "`torc1.cs.utk.edu`."

2.3 Set-Matching Algorithm

The set-matching algorithm evaluates a set-extended ClassAd request against a set of resource ClassAds and returns a resource set that has highest rank. It comprises two phases.

In the *filtering* phase, individual resources are removed from consideration based on individual expressions in the request. For example, individual expressions `"other.os==redhat6.1 && other.memory>=100M"` would remove any machine with an OS other than Linux Redhat v6.1, and/or with less than 100 Mb of memory. A `Postfix` expression can also be used in this phase, as discussed above. A set-matching implementation can index ClassAds to accelerate such filtering operations.

In the *set construction* phase, the algorithm seeks to identify a resource set that best meets application requirements. As the number of possible resource sets is large (exponential in the number of resources

available), it is not typically feasible to evaluate all possible combinations. Instead, we use the following greedy heuristic algorithm to construct a resource set from the resources remaining after Phase 1 filtering.

```
CandidateSet = NULL;
BestSet=NULL;
LastRank = -1; Rank = 0;
while  (ResourceSet != NULL)
{
  Next = {X : X in ResourceSet && for all Y in ResourceSet,
              rank(X+CandidateSet) > rank(Y+CandidateSet); }
  ResourceSet = ResourceSet - Next;
  CandidateSet = CandidateSet + Next;
  Rank = rank(CandidateSet);
  If (requirements(CandidateSet)==true && Rank > LastRank)
       BestSet=CandidateSet;
       LastRank=Rank;
}
if BestSet ==NULL return failure
else return BestSet
```

In narrative form, the algorithm repeatedly removes the “best” resource remaining in the resource pool (with “best” being determined by the rank of the resulting resource set formed) and adds it to the “candidate set.” If this “candidate set” has higher rank than the “best set” so far, the “candidate set” become the new “best set”. This process stops when the set of resources in the resource pool is exhausted. The algorithm returns the “best set” that satisfies the user’s request, or failure if no such resource set is found.

As with other greedy algorithms, this set-matching algorithm is not guaranteed to find the best solution if one exists. The set-matching problem can be modeled as an optimization problem under some constraints. It is known that this problem is NP-hard under some situations. Hence it is difficult to find a general algorithm to solve this problem efficiently, especially when the number of resources is large. Our work provides an efficient algorithm with complexity $O(N^2)$ with rank computation as the basic operation.

3 Resource Selection Framework

We have implemented a general-purpose resource selection framework based on the set-matching technique. It accepts user resource requests and finds a set of resources with highest rank based on the resource information provided by Grid Information Service. It also provides an open interface for users to specify the application-specific mapping module to customize the resource selector.

3.1 System Architecture

The architecture of our resource selection system is shown in Figure 1:

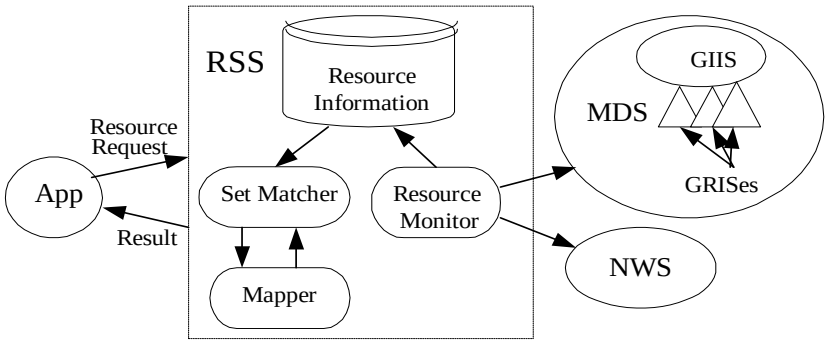


Figure 1: Architecture of Resource Selector

The Grid Information Service is provided by MDS [21] and NWS [22-24]. The *Meta Directory Service* (MDS) is a component of Globus Toolkit [25]. It provides a uniform framework for discovering and accessing system configuration and status information such as compute server configuration and CPU load. The NWS (Network Weather Service) is a distributed system that periodically monitors and dynamically forecasts the performance that various network and computational resources can deliver over a given time interval.

The Resource Selector Service (RSS) comprises three modules. The *resource monitor* acts as a Grid Index Service (GRIS) in the terminology of [21]; it is responsible for querying MDS and NWS to obtain resource information and for caching this information in local memory, refreshing only when associated time-to-live values expire. The *set matcher* uses the set-matching algorithm to match incoming application requests with the best set of available resources. For some applications such as Cactus, their performance is tightly related to the topology of resources and the workload allocation to machines. So it is necessary to map the workload to resources before judging whether the resources are good or bad. The *mapper* is responsible for deciding the topology of the resources and allocating the workload of application to resources.

Because the mapping strategy is tightly related to a particular application, it is difficult to find an efficient general mapping algorithm suitable for all applications. In our system, the mapper is designed and implemented as a user-specified dynamic link library that can be loaded at run time.

3.2 Resource Request

The RSS accepts both synchronous and asynchronous requests described by set-extended ClassAds. It responds to a synchronous request with the best available resource set that satisfies this ClassAd, or "failure" if no such resources are available. An asynchronous request specifies a request lifetime value; the RSS responds if and only if a resource set that satisfies the specified ClassAd becomes available during the specified lifetime.

A resource request may include six types of element:

- *Owner*: The sender of this request.
- *Job description*: The characteristics of the job to be run, for example, the performance model of the job.
- *Type of Service*: Synchronous or asynchronous.
- *Mapper*: The kind of mapper algorithm to be used.
- *Constraint*: User resource requirements, for example, memory capability, type of operating system, software packages installed, etc.
- *Rank*: The criteria to rank the matched resources.

We can use these six elements to describe various resource requests for different kinds of applications. The following example is the request that we used for a Cactus application.

```
1.    [
2.    Service = "InstantService";
3.    MatchType="SET";
4.    iter=100; alpha=100; x=100; y=100; z=100;
5.    cactus=370; cactusC=254; startup=30;
6.    computetime = x*y*alpha/other.cpuspeed*cactus;
7.    comtime= ( other.RLatency+ y*x*cactusC/other.RBandwidth
                 +other.LLatency+y*x*cactusC/other.Lbandwidth);
8.    exectime=(computetime+comtime)*iter+startup;
9.    Mapper = [type ="dll"; libraryname="cactus"; function="mapper"];
10.   requirements = Sum(other.MemorySize) >= (1.757 + 0.0000138*z*x*y)
                     && Postfix(other.machine, domains);
11.   domains={ cs.utk.edu, ucsd.edu};
12.   rank=Min(1/exectime)
13.   ]
```

Line 2 specifies that this is a synchronous request. Lines 4–8 are the job description including the problem size and the Cactus performance model (Section 4). Line 8 models the execution time of every subtask on a machine. Line 9 gives the name and location of the mapping algorithm used for the application. Line 10 is the resource constraints that say the total memory capability of the resource set should be large enough to keep the computation in memory that is described by a formula of the problem size, and resources should be selected from machines in "`cs.utk.edu`" or "`ucsd.edu`" domain that is described in Line 11. Line 12 denotes that the reciprocal of the execution time of the application is used as the criterion to rank candidate resources. Because the execution time of the application is decided by the subtask that finishes last, the rank of a resource set is equal to the minimum value of the reciprocal of the execution time of subtasks as specified in Line 12. If multiple resource sets fulfill the requirements, the resource set on which application gets smallest execution time has the highest rank.

3.3 Resource Selection Result

The result returned by Resource Selector (expressed in XML) indicates the selected resources and mapping scheme. The following example is the result that we obtained for the Cactus application.

```
<virtualMachine>
    <result statusCode="200" statusMessage="OK"/>
    <machineList>
    <machine dns="torc2.cs.utk.edu"   processor= 2    x= 20>
    <machine dns="torc3.cs.utk.edu"   processor= 2    x= 15>
    <machine dns="torc6.cs.utk.edu"   processor= 2    x= 15>
    </machineList>
</virtualMachine>
```

This returned resource set includes three machines, each of which has two processors. These three machines have one-dimensional topology, and the workload is allocated to the machines according to ratio 20:15:15.

4 Cactus Application

We applied our prototype in the context of a Cactus application. The Cactus application we used is the simulation of the 3D scalar field produced by two orbiting sources. The solution is found by finite differencing a hyperbolic partial differential equation for the scalar field. This application decomposes the 3D scalar field over processors and places an overlap region on each processor. For each time step, each processor updates its local grid point and then synchronizes the boundary value.

4.1 Performance Model

In this Cactus experiment, we use expected execution time as the criterion to rank all the sets of candidate resources. For a 3D space of X*Y*Z grid points, the performance model is specified by the following formulas, which describe the required memory and estimated execution time.

Requested Memory(MB) > = (1.757 + 0.0000138*X*Y*Z)

Execution time =(computation (0) + communication (0)) * slowdown(CPU load) +start-up-time

Function *slowdown(CPU load)* presents the contention effect on the execution time of the application. CPU load is defined as the number of processes running on the machine. Silvia Figueira modeled the effect of contention on a single-processor machine [26, 27]. Assuming that the CPU load is caused by CPU-bounded processes and the machine uses round-robin scheduling method, we extended her work by modeling the effect of contention on the dual-processor machine. We found that the execution time is smaller if we divide a job into two small subtasks than if we run this job as one task on dual-processor machines. We applied this allocation strategy to dual-processor machines and obtained the following contention model, which we validate in Section 5.1:

slowdown (CPU load)= (2* CPU Count – 1 +CPU load)/(2*CPU Count –1)

This formula is applicable when the CPU count is equal to one or two. We validate this formula in.

Computation (0) and *communication (0)*, the computation time and communication time of the Cactus application in the absence of contention, can be calculated by formulas described in [28]. We incur a startup time when initiating computation on multiple processors in a Grid environment. In these experiments, this time was measured to be around 40 seconds when machines are from different clusters (sites) and 25 seconds when machines are in the same cluster.

4.2 Mapping Algorithm

We decompose the workload in the Z direction and decide the resource topology as follows:

1. Pick the machine with the highest CPU speed as the first machine of the line.
2. Find the machine that has the highest communication speed with the last machine in the line, and add it to the end of the line.
3. Continue step 2 to extend the line until all machines are in the line.

We thus minimize WAN communications by putting machines from the same cluster or domain in adjacent locations.

The mapper then allocates the workload to these resources. Our strategy is to allocate the workload to each processor inversely proportional to the predicted execution time on that processor.

5 Experimental Results

To verify the validity of our RSS and the mapping algorithm of the Cactus application, we conducted experiments in the context of the Cactus application on the GrADS [29] test bed, which comprises

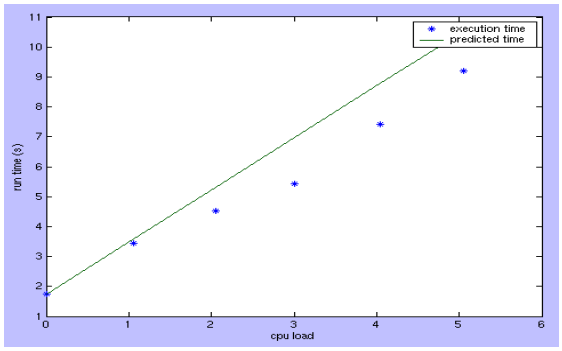
workstation clusters at universities across the United States, including the University of Chicago, UIUC, UTK, UCSD, Rice University, and USC/ISI. We tested the execution time prediction function, the Cactus mapping strategy, and the set-matching algorithm, respectively.

5.1 Execution Time Prediction Test

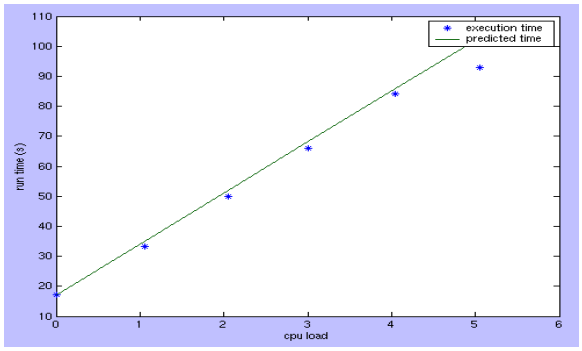
From the above description, we can see that both the mapping strategy and the set-matching algorithm are based on the predicted execution time of the Cactus application. The correction of the execution time prediction function is the base of the validity of the mapping strategy and set-matching algorithm. We tested the execution time prediction function both without communication time and with it.

5.1.1 Computation Time Prediction Test

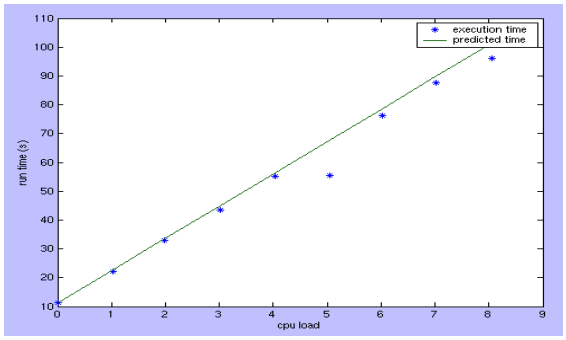
When the Cactus application runs on only one machine, there is no communication cost. To validate the computation time prediction function, we ran the Cactus application on one machine and compared the predicted computation time with the measured computation time. We did the experiments with diverse configurations, including (1) different problem sizes (20*20*20, 50*50*50, 100*100*100), (2) different clusters (UTK cluster, UIUC cluster, and UCSD cluster), (3) different CPU speed (cmajor.cs.uiuc.edu 266MHz, mystere.ucsd.edu 400 MHz, torc.cs.utk.edu 547 MHz), (4) on machines with different number of processor (UIUC and UCSD machines have 1 processor, UTK machines have 2 processors), and (5) different CPU load.



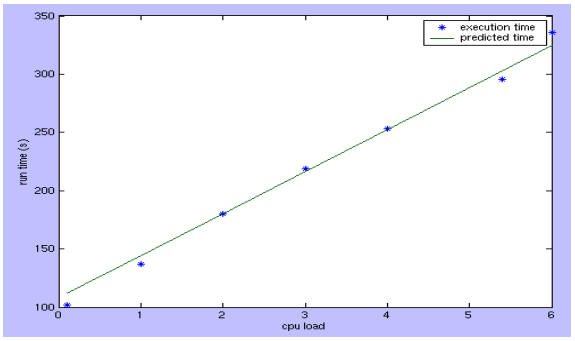
(1) Machine name: cmajor.cs.uiuc.edu
CPU count =1
CPU speed =266MHz
Problem size 20*20*20



(2) Machine name: cmajor.cs.uiuc.edu
CPU count =1
CPU speed=266MHz
Problem size=50*50*50



(3) Machine name: mystere.ucsd.edu
CPU count =1
CPU speed=400MHz
Problem size=50*50*50



(4) Machine name: torc1.cs.utk.edu
CPU count=2
CPU speed=547 MHz
Problem size=100*100*100

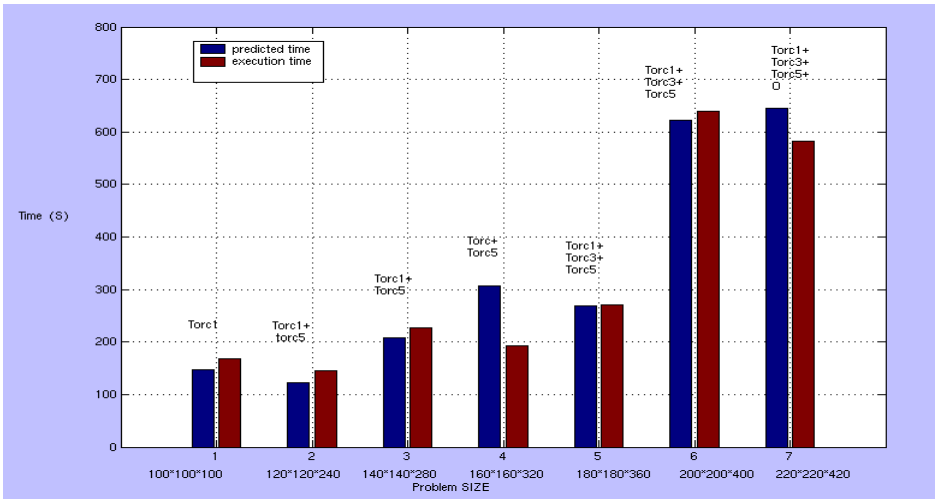
Figure 2: Predicted computation time and measured computation time of the Cactus application

Figure 2 illustrates the predicted computation time and the measured computation time of the Cactus application with different CPU loads and various machine configurations. The figure shows that the computation time prediction function gives acceptable prediction in all cases. The error in this experiment was within 6.2% on average.

5.1.2 Computation Time and Communication Time prediction Test

Then we tested the execution time prediction function that includes both computation time and communication time. In this experiment, we ran the Cactus application on various machine combinations and compared the measured execution time with the predicted execution time. We conducted experiments with various configurations, including (1) different problem sizes (100*100*100, 120*120*240, 140*140*280, 160*160*320, 200*200*400 and 220*220*420), (2) different clusters (UCSD cluster and UTK cluster), (3) different CPU speed (o.ucsd.edu 400 MHz, torx.cs.utk.edu 547 MHz), (4) different numbers of processors (UCSD machines have 1 processor, UTK machines have 2 processors), and (5) different machine combinations.

The predicted execution time and the measured execution time of the Cactus application running with various configurations are shown in Figure 3. The error rate is 13.13% on average. We can see that in most of the cases, the time prediction formula works well. But for the problem size 160*160*320, the predicted time is much greater than the execution time (the error rate is as high as 59%). We monitored the CPU load of the machines on which the application had run during experiments. We found that a competing application had been running on torc1 and torc5 when the resource selector collected system information to predict the execution time and this application terminated before our application run. We therefore believe that the reason for this large error rate is that the CPU load information we used to predict the performance of application does not reflect the real CPU load when the application ran.



Machines: torc1.cs.utk.edu, torc3.cs.utk.edu, torc5.cs.utk.edu
o.ucsd.edu

Figure 3: Predicted and real execution time when considering communication time

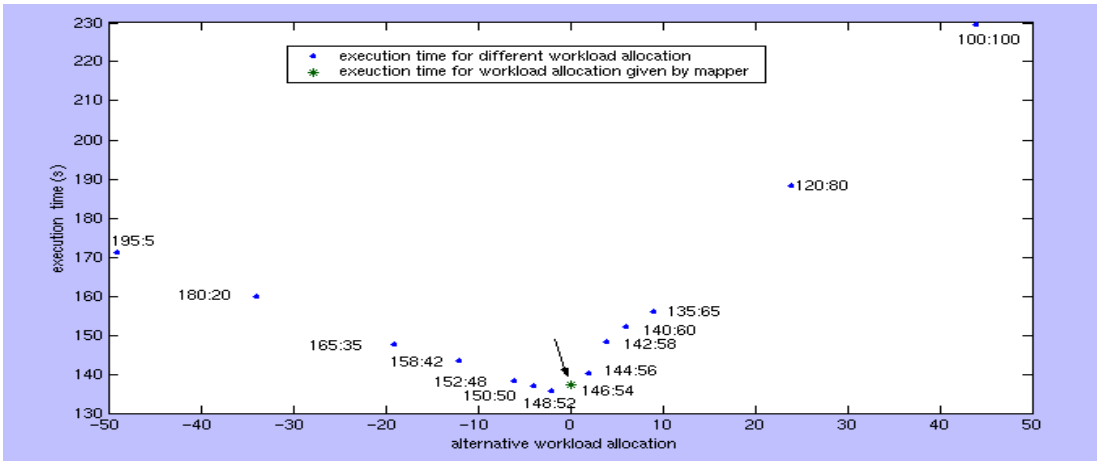
5.2 Mapping Strategy Test

In our mapping strategy experiment, we tested what benefit was gained from the mapping strategy. As mentioned in Section 4.2, the mapping strategy put machines with high bandwidth connection into adjacent positions in the topological arrangement. Clearly, this one-dimensional

arrangement minimizes the communication via WAN and thus reduces the total communication cost. In this section, we focused on testing how well the workload allocation strategy works. In particular, we tested whether the execution time of the Cactus application with allocation given by the mapper is shorter than its execution time with any other allocation strategy.

We tested the workload allocation strategy on two machines (dralion.ucsd.edu and cirque.ucsd.edu). One machine (dralion) has a CPU speed of 450 MHz and no CPU load during the experiment. The other machine (cirque) has a CPU speed of 500 MHz and a CPU load of 2 during the experiment. We set up the Cactus application with a 3D space of 100*100*200 grid points and one-dimensional decomposition. According to our workload allocation strategy, the best performance was obtained when the workload was allocated on the two machines in the proportion of 146:54 (dralion:cirque) in the Z direction. We ran the Cactus application with this workload allocation and its variations (obtained by moving the division point to the right and left), and compared the execution time of the application with other workload allocation strategies.

The execution time for different workload allocations is shown in Figure 4. We can see that the execution time with the allocation given by the mapper is very close to optimal (only 1.2% higher than optimal). Moreover, the execution time increases when the deviation from our workload allocation scheme increases. Thus we can say that the workload allocation strategy works well.



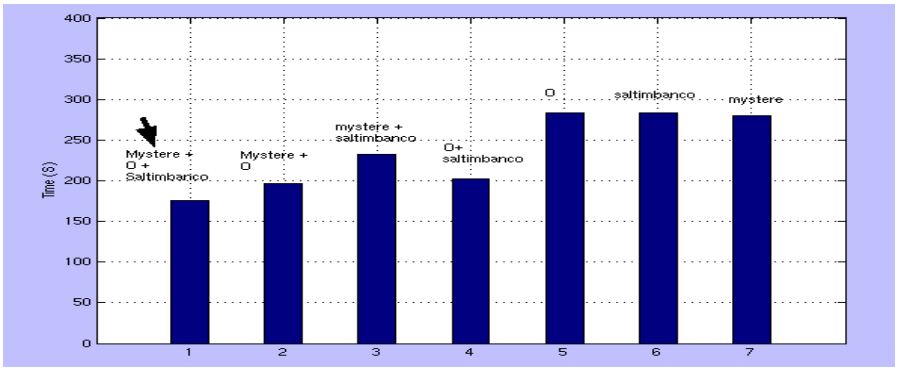
Machines: dralion.ucsd.edu (450MHZ), cirque.ucsd.edu(500 MHZ)
Problem size: 100*100*200

Figure 4: The execution time for different workload allocations.

5.3 Resource Selection Algorithm Test

To validate the resource selection algorithm, we asked the resource selector to select a set of machines from a pool of candidates. We ran the Cactus application on all possible machine combinations and then compared its execution time on these different combinations.

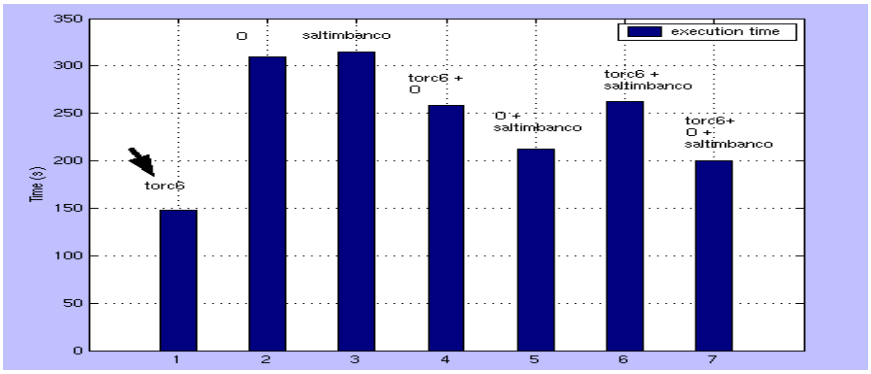
In our experiment, we limited the number of candidate machines to three. Hence, there were seven possible machine combinations. We carried out the experiment both on machines from a single cluster and on machines from different clusters.



Machines candidates: mystere.ucsd.edu, o.ucsd.edu, saltimbanco.ucsd.edu
Selected machine: mystere.ucsd.edu, o.ucsd.edu, saltimbanco.ucsd.edu

Figure 5: Execution time on all combinations and selected machines
(Candidate machines from one cluster)

When the three candidate machines were in a single cluster (UCSD cluster in our experiment), the machines were connected via a high-bandwidth network (100 Mbps Ethernet for the UCSD cluster). The communication cost between machines is relatively small, so more machines mean shorter execution time. As we expected, the resource selector selected all three candidates into the selected machine set. The execution time of the application on all of the machine combinations is shown in Figure 5. We can see that the execution time of the application on the three selected machines is shorter than on all other six combinations.



Machine candidates: o.ucsd.edu, saltimbanco.ucsd.edu, torc6.cs.utk.edu
Selected machine: torc6.cs.utk.edu

Figure 6: Execution time on all combinations and selected machines
(Candidate machines from different clusters)

When the three candidates were selected from two different clusters connected by WAN, the resource selector selected one machine in the UTK cluster on which the application was expected to run faster than on any other combination. In this configuration, the use of more machines does not result in higher performance (as demonstrated in the previous experiment), because the high inter-machine communication cost outweighs the benefits of greater processing power. The measured execution time of application on all machine combinations is shown in Figure 6. We can see that the selected machine has a shorter execution time than do all the other six combinations.

6 Conclusion and Future Work

Grids enable the aggregation of computational resources to achieve higher performance, and/or lower cost, than can be achieved on a single system. The heterogeneous and dynamic nature of Grids, however, leads to numerous technical problems, of which resource selection is one of the most challenging.

We have presented a general-purpose resource selection framework that provides a common resource selection service (RSS) for different kinds of application. This framework combines application characteristics and real-time status information to identify a suitable resource set. A language called set-extended ClassAds is used to express resource requests, and a new technique called set matching is used to identify suitable resources. We have used an application, Cactus, to validate the design and implementation of the resource selection framework, with promising results.

Our framework should adapt to different applications and computational environments. Further experiments on other kinds of application are needed to validate and improve our work. We also plan to provide more mapping algorithms for different kinds of application.

Acknowledgments

This work was supported by the Grid Application Development Software (GrADS) project of the NSF Next Generation Software program, under Grant No. 9975020.

References

1. Petitet, A., S. Blackford, and J. Dngarra, *Numerical Libraries And The Grid: The GrADS Experiments With ScaLAPACK*. 2001, University of Tennesse.
2. Berman, F. and R. Wolski. *The AppLeS project: A Status Report*. in *Proceedings of the 8th NEC Research Symposium*. 1997. Berlin, Germany.
3. Litzkow, M., M. Livny, and M. Mutka. *Condor - A Hunter of Idle Workstations*. in *Proceedings of the 8th International Conference of Distributed Computing Systems*. 1998.
4. Abramson, D., J. Giddy, and L. Kotler. *High Performance Modeling with Nimrod/G: Killer Application for the Global Grid*. in *International Parallel and Distributed Processing Symposium*. 2000.
5. Shao, G. and F. Berman. *Master-slave computing on the Grid*. in *Proceedings of the 9th Heterogeneous Computing Workshop*. 2000.
6. Cray, R., *Document number in-2153 2/97*. 1997, Cray Research.
7. Henderson, R. and D. Tweten, *Portable Batch System: External reference specification*. 1996, Ames Research Center.
8. Zhou, S. *LSF: Load sharing in large-scale heterogenous distributed system*. in *Proc. Workshop on Cluster Computing*. 1992.
9. Foster, I., et al., *Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment*. 1998.
10. I.B.M., C., *IBM Load Leveler: User's Guide*. 1993.
11. Czajkowski, K., et al. *A Resource Management Architecture for Metacomputing Systems*. in *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*. 1998.
12. Chapin, S.J., et al. *Resource Management in Legion*. in *Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99)*. 1999. San Juan, Puerto Rico.

13. Dail, H., G. Obertelli, and F. Berman. *Application-Aware Scheduling of a Magnetohydrodynamics Applications in the Legion Metasystem*. in *Proceedings of the 9th Heterogeneous Computing Workshop*. 2000.

14. Gehring, J. and A. Reinefeld, *MARS-A Framework for Minimizing the Job Execution Time in a Metacomputing Environment.* Future Generation Computer Systems, 1996. **12**(1): p. 87-99.

15. Arabe, J.N.C., et al., *DOME: Parallel programming in a heterogeneous multi-user environment*. 1995, CMU.

16. Sirbu, M.G. and D.C. Marinescu. *A scheduling expert advisor for heterogeneous environments*. in *Proceedings of the 6th Heterogeneous Computing Workshop (HCW' 97)*. 1997.

17. Raman, R., *ClassAds Programming Tutorial (C++)*. 2000.

18. Raman, R., M. Livny, and M. Solomon. *Matchmaking Distributed Resource Management for High Throughput Computing*. in *Proceedings of The Seventh IEEE International Symposium on High Performance Distributed Computing*. 1998. Chicago, IL.

19. Allen, G., et al., *Cactus Tools for Grid Applications.* Cluster Computing, 2001(4): p. 179-188.

20. Allen, G., et al., *Solving Einstein's Equation on Supercomputers.* IEEE Computer, 1999(32): p. 52-59.

21. Globus, *MDS document*: www.globus.org/mds.

22. Wolski, R., *Dynamically Forecasting Network Performance Using the Network Weather Service.* Journal of Cluster Computing, 1998.

23. Wolski, R., N. Spring, and J. Hayes, *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing.* Journal of Future Generation Computing Systems, 1999. **15**(5-6): p. 757-768.

24. Wolski, R., N. Spring, and J. Hayes. *Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid*. in *Proceedings of the 8th High-Performance Distributed Computing Conference, August, 1999*. 1999.

25. Foster, I. and C. Kesselman, *Globus: A Tookkit-Based Grid Architecture.*, in *The Grid: Blueprint for a New computing Infrastructure*. 1999, Morgan Kaufmann. p. 259-278.

26. Figueira, S.M. and F. Berman. *Modeling the Effects of contention on the Performance of Heterogeneous Application*. in *HPDC*. 1996.

27. Figueria, S.M. and F. Berman. *Predicting slowdown for Networked Workstations*. in *HPDC*. 1997.

28. Ripeanu, M., A. Iamnitchi, and I. Foster, *Performance Predictions for a Numerical Relativity Package in Grid Environments.* International Journal of High Performance Computing Applications, 2001. **15**.

29. Berman, F., A. Chien, and K. Cooper, *The GrADS Project: Software support for High level Grid application development.* International Journal of Supercomputer Applications, 2001. **14**(4): p. 327-344.