Issues in Runtime Algorithm Selection for Grid Environments

Prachi Bora, Calvin Ribbens, Sandeep Prabhakar, Gautam Swaminathan,
Malarvizhi Chinnusamy, Ashwin Jeyakumar, Beatriz Diaz-Acosta
{pbora,ribbens,sprabhak,gswamina,mchinnus,ajeyakum,beatriz}@vt.edu
Department of Computer Science
Virginia Tech

Abstract

The heterogeneity and unpredictability of grid computing environments is a severe challenge to algorithm selection for large-scale scientific codes. This paper discusses a set of requirements for an algorithm selection web service, which supports runtime, context-aware algorithm recommendations in a multi-language, multi-platform environment. An architecture and prototype implementation of such a system is described, along with a perspective and case study of how such a system will be helpful to other grid middleware such as meta-scheduling systems.

1. Introduction

If grid computing is to realize its much publicized promise, developers of algorithms, applications, and middleware must deal with a formidable list of challenges. Computational grids are dynamic, unpredictable, diverse, and heterogeneous; and each of these adjectives causes great difficulty to developers of algorithms and infrastructure for the grid. The goal of this paper is to describe an approach and architecture for runtime algorithm selection and problem-aware meta-scheduling for the grid – two technologies which alone, and especially in combination, can contribute much toward dealing with the heterogeneity and unpredictability of the grid.

The motivating applications for this research (e.g., [1, 27]) are typical of many grid computing applications: large parallel SPMD codes, requiring a large and perhaps diverse set of computational resources, but with computation time dominated by only a few steps (repeated many times on different data sets), and with myriad algorithmic and implementation choices for those basic steps. Furthermore, we are focusing on applications that run *many* times on similar problem instances. This is typical of scientific computing applications, where parameter sweeps, simulation ensembles, and high-level problem solving strategies such as optimization and design are common. In fact, even a single simulation of a large time-dependent or nonlinear system may require huge numbers of basically the same step. Our approach leverages as much of this problem-solving context as possible in order to deal with the extreme heterogeneity of the grid setting: heterogeneity in the multitude of algorithms to choose from, and heterogeneity in the unpredictable and diverse set of machines on which to run.

The canonical example of a dominant scientific computing calculation is the solution of systems of linear algebraic equations. In the last 50 years there has been an incredible proliferation of linear solver algorithms and implementations, especially when one considers not only algorithmic issues (direct or iterative, which iterative, which preconditioner, what parameter values, etc.) but also implementation issues (data structure, data distribution, blocking, communication strategy, etc.). It is difficult enough to select a good parallel algorithm/implementation when the computational resource is known at compile time. In a grid setting, however, the computational resource is determined at scheduling time, and important characteristics may not even be known until runtime, e.g., network properties. In such a situation it is not possible to choose a good algorithm statically. The choice of algorithm can be done only at runtime, after the resources on which the application will run have been determined. Hence,

one would like to choose the algorithm as late as possible. Note also that delaying the algorithm choice until runtime allows for the possibility of choosing algorithms based on problem characteristics that are known only at runtime, e.g., matrix sparsity, eigenvalue estimates, and problem parameters. What is needed is a runtime algorithm selection framework that meets the following requirements:

- **Transparency to the application**. The application program generally represents a large complex code base. The user should not be expected to change the program based on which algorithms run well on a given set of resources.
- **Selection determined by runtime conditions.** Factors such as problem parameters, matrix sparsity, interconnection network performance, available swap space, and cache size can affect the choice of an algorithm.
- **Complementary to other grid middleware.** The algorithm selection system should provide information that is beneficial to other systems that improve performance or seek to reduce burden on the user. A meta-scheduler is one such system that could use information stored in the algorithm selection system to make a better choice of resources.
- **Transparency to the user.** The algorithm selection system should not expect every library to be present on every system if the choice of library or algorithm is to be made at runtime. If a library is not present on the system on which the application is to run, an attempt should be made to bring that library to the system for dynamic linking.
- **Language independence.** Yet another aspect of heterogeneity present in our context is in languages. The implementation language of the application and the libraries should not be expected to match.

While a runtime algorithm selection system that meets these requirements is very useful in itself, it becomes an even more powerful grid resource when combined with a meta-scheduling service. A meta-scheduler provides a set of resources, each possibly under the control of its own local scheduler, on which to run grid applications. It is not obvious how best to combine runtime algorithm selection and grid meta-scheduling. In the simplest case, a meta-scheduler could select a set of resources on which to run a given application; the runtime algorithm selection service could then use this information to recommend algorithms for that set of resources. However, in this scenario the meta-scheduler does not take into account the fact that different algorithm/resource combinations may have potentially very different performance. If the full problem-solving context (e.g., previous results, runtime values of important indicators) were leveraged, could a better choice of both algorithm and machine be made? Our goal is to define an architecture that combines runtime algorithm selection and meta-scheduling to yield high-performance, application-aware and resource-aware grid scheduling. The approach described in this paper is to have the algorithm selection service supply the scheduler with information about machine and algorithm characteristics, which yield the best performance for a given computation. In other words, the recommender system gives the meta-scheduler a ranked list of machine characteristics (or machine "equivalence classes"), each with a good algorithm choice, so that the meta-scheduler can then attempt to acquire the most effective resources for solving a given problem.

The remainder of the paper is organized as follows. Section 2 describes relevant work in runtime algorithm selection and a brief mention of some of the tools our system is built on. In Section 3 we describe our basic approach and architecture, including its distributed design, details of the algorithm selection service, and a description of our approach to meta-scheduling. Section 3.4 shows how these two systems relate to each other and how their efforts can be combined. We conclude in Section 4 with a brief indication of future research directions.

2. Related Work

The idea of adapting algorithms to resource characteristics has been a very fruitful one in scientific computing for many years. A well-known example is ATLAS [20], which aims at automatically tuning numerical linear algebra kernels for specific architectures. ATLAS generates optimized BLAS operations for processors with deep memory hierarchies by obtaining cache information during the compilation of the library. ATLAS fine-tunes parameters of what is basically a fixed algorithm, based on the architecture.

Active Harmony [17] is a system that selects algorithms at runtime and also fine-tunes algorithm parameters. However, the system requires that the libraries provide function calls in their APIs to support the measurement of performance metrics. Our system adds a layer that takes care of measuring and recording performance data, thereby leaving the underlying library code unchanged. Thus, any new library performing similar functions as existing ones can be incorporated without requiring any changes. Active Harmony currently supports C and FORTRAN libraries.

SANS [19] is another system that aims at selecting algorithms based on resource characteristics and problem features. Although similar ideas have been used in this paper, we take a novel approach by using recommender systems to predict best performance conditions in situations unseen before.

A key enabling technology for our framework is Babel [10]. Babel is a tool that provides language interoperability and a component framework targeted at facilitating code reuse in parallel scientific applications. Our system uses Babel to provide a layer of abstraction, which makes algorithm selection transparent to the user. The functionality provided by a component is expressed using Babel's Scientific Interface Definition Language (SIDL). SIDL is similar to interface definition languages in COM and CORBA, but is targeted at scientific applications. SIDL only defines the interfaces and not their implementation. The Babel compiler reads the SIDL definition and generates glue code for each component. It is the responsibility of library writers to fill in library specific code in the implementation prototypes. Application (client) programs can then access components through standard interfaces. Babel supports FORTRAN, C, C++, Java and Python.

Our algorithm selection mechanism is based on the recommender system technology available in Pythia-II [21, 22]. Pythia-II combines Knowledge Discovery in Databases (KDD) methodology with Recommender System technology to provide recommendations about scientific software. Given a database of results from previous runs, Pythia-II can be used to generate rules that recommend algorithms given current resource and problem characteristics.

The Apache Axis toolkit [7] is used to develop a web service framework [9] for both the runtime algorithm selection system and the meta-scheduler. A UDDI registry [8] is used to discover peer web services. Performance API (PAPI) [6] provides an interface for the application to query low-level hardware counters that gives details like number of instructions executed. Such a low level interface to the execution of the code is necessary to obtain noise free information in a grid environment. However making PAPI calls necessitates rewriting of the application. To make PAPI calls transparent to the user, we modified MPICH-G2 to call PAPI functions whenever a MPI call is made. This was found to be sufficient for scheduling purposes.

3. Architecture

3.1 Distributed Design

Unlike most current grid infrastructure [14, 15, 16], we are developing a “zone-based” middleware infrastructure [1] for grid applications. A distributed architecture is necessary for scalability reasons and to allow individual administrative domains to follow policies and

mechanisms suited to its own organizational requirements. The motivation for this distributed approach is analogous to the situation in networking protocols. In the early days of the Internet, all networks were connected to one backbone following one routing protocol. This evolved to a distributed system with each autonomous system running its own policies and algorithms (like RIP and OSPF). Like the nascent days of the Internet, a typical computational grid today exists as a single system, e.g., with a single grid scheduler, which maintains information about all resources. We feel that in the future, grid infrastructure will evolve to be autonomous and distributed, similar to the Internet of today.

The design of our system is distributed in the following sense. Every user of the grid has a "home zone". This zone corresponds to the organization that the user represents. This architecture is required in a grid environment where an organization might refuse to allow an external entity to make scheduling decisions for a resource in its administrative domain. Each zone has one server where the policies and mechanisms of the organization are represented. The server may offer one or more services. The services that our prototype system provides are a scheduler and a runtime algorithm selection service. Since this server is within a single organizational domain, any user in that organization has complete trust in its security mechanisms.

3.2 Algorithm Selection System

The runtime algorithm selection system consists of several components (see Figure 1). The Runtime Algorithm Selection (RAS) Engine provides a layer of abstraction between the end user application and the available algorithms for solving the application's problem. This layer provides transparent access to the underlying algorithms and provides an abstraction for steps such as matrix construction, which may require different data structures for different algorithms. The RAS Engine is a Babel based code; it uses a generic SIDL interface [2] to provide transparent access to the algorithm recommender system. For example, when the application needs a linear solver, it makes a generic call to the GetSolver method of the RAS Engine, and gets back an object instance of the algorithm. Figure 2 shows a high-level version of the GetSolver method (left) and the GetRecommendation method (right) which it calls to interact with the Recommender System (RS). The application then makes a call on this returned object to invoke the desired functionality, e.g., solving a linear system. The pool of solvers shown in Figure 1 is just a collection of various libraries providing similar functionality.

The RS is responsible for making the 'right' choice of algorithm based on input data (e.g., matrix size, matrix symmetry and matrix sparseness in case of linear systems) and resource characteristics (e.g., available memory, CPU load, architecture and operating system). When the RAS Engine is invoked using the GetSolver function, it consults the RS about which solver to instantiate. The RS, based on Pythia-II, not only selects software appropriate for the problem at hand but it can also suggest algorithm parameter values, e.g., blocking factor for data distributions or restart parameter for GMRES.

The RS depends on the availability of a sizeable amount of data on similar types of problems in order that a good recommendation be made. When a recommendation is needed, the past performance data is consulted to determine which algorithm is appropriate for the present scenario and also what tuning parameters should be used for the selected algorithm. Recommendation becomes necessary when the user's objectives cannot be represented as simple database queries. Thus, recommendation can handle situations unseen before. The RS uses a normalized relational database, which is assumed to have tables for problem features (defined/supplied by a knowledge engineer), algorithm features (algorithm name, tunable parameters), resource characteristics (memory, load, interconnection network) and experiments (containing specific instances of previous runs of the problem).

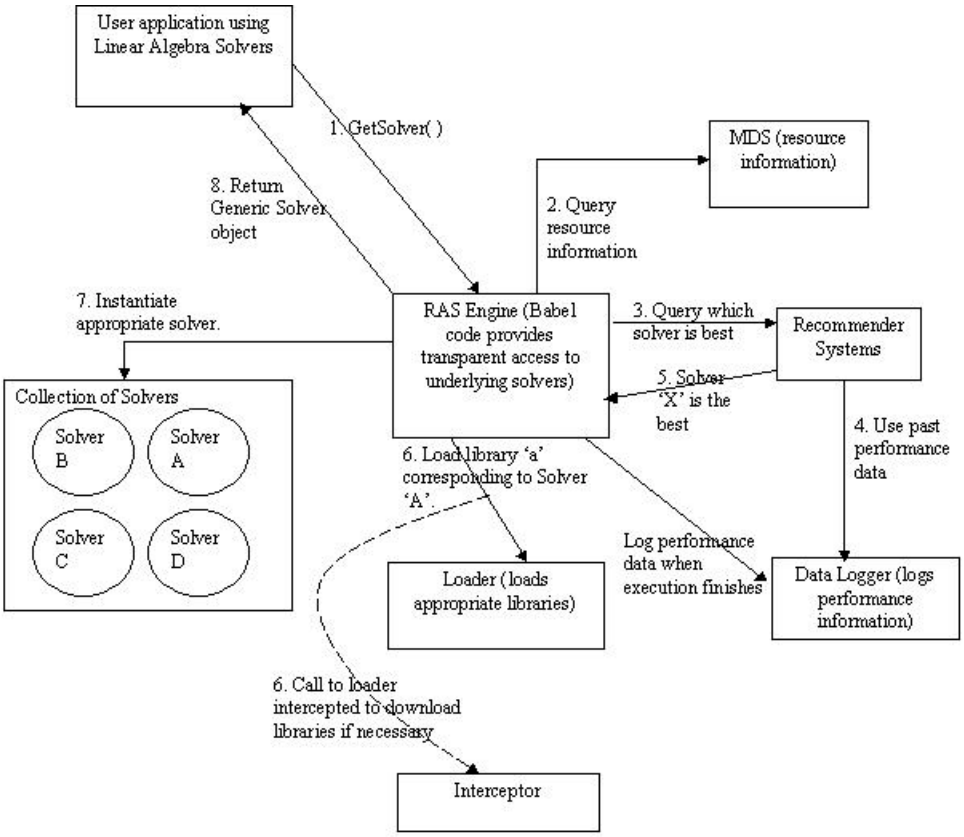

Figure 1: Architecture of Runtime Algorithm Selection (RAS)

```
Procedure GetSolver
{
    Obtain resourceInfo
    solver = GetRecommendation(resourceInfo,
                problem-features)
   BuildDataStructure(solver)
   Instantiate solver
}
```

```
Procedure GetRecommendation
{
  if(match pattern (resource-features, problem-
                features))
        Return recommendation
  else
        Contact peer RS
        Obtain recommendation
        Return recommendation
}
```

Figure 2: Pseudo code of the RAS Engine (left) and RS (right)

Apart from providing suggestions about which software should be used, the RS also generalizes from the data collected in the database by discovering patterns to capture the semantic behavior of the software system. The RS uses 'Case-Based Reasoning', 'Inductive Logic Programming' and 'ID3 (Induction of Decision Trees)' learning techniques [23, 24, 25, 26]. These learning techniques discover patterns and are used for evaluating recommendations provided. The evaluation is based on application performance data that is logged in the experiments table and is helpful in making better decisions in the future. Use of multiple learning methods is beneficial because the strengths of each individual technique can be leveraged.

Since recommendations are based not only on problem features, but also on resource characteristics, we use the Meta-computing Directory Service (MDS) [3, 4] to obtain resource information. When an application calls GetSolver, the RAS Engine contacts the MDS to obtain resource information. The typical information obtained is CPU load and memory. This resource information and problem features are given to the RS to obtain suggestions about which algorithm should be used.

There is one RS per administrative domain of the grid. Each RS is a web-service. When a RS is contacted for a recommendation and the request is previously unseen, the RS could contact a peer RS on the grid to see if it has any recommendations available for a similar problem instance. In addition to consulting peer RS, local decisions can also be made using the learning techniques mentioned above. Also, each RS could have its own learning technique (similar to the concept of local scheduling policies explained in Section 3.3).

In the case of parallel algorithms, it is important that the RS recommend the same algorithm to each member of a "team" of processes that is cooperating on a particular computational step, e.g., solving a single large linear system in parallel. Hence, we assume that a computational team runs on a homogeneous set of machines. (Note that a single large computation may require many teams, either in parallel or in sequence, and that these separate teams may indeed be on different resources and require different algorithms.) This restriction is necessary because if teams are allowed to be heterogeneous, then the RS may recommend different algorithms to different team members, depending on the particular machine that that process is running on. With the homogeneous team assumption in place, there are two possible approaches to select the algorithm. In the first approach, the team leader could ask for an algorithm and communicate the information to the team members. Another approach is that each team member could ask for a recommendation, passing some unique handle identifying the members of the same team. In the latter case, the recommender system should maintain a small cache to remember earlier requests and return the same algorithm to the requesting team member as was returned to its peer. We use the former approach as the network traffic is reduced significantly when the team leader communicates information as opposed to each team member consulting the RS. Also, in this case, the recommender system need not maintain a cache to remember the decisions made.

It is also assumed that processes do not migrate. When a process migrates, it could reside on an altogether different resource type than the previous one and thus, a different algorithm might be needed to continue solving the problem. Conversion from one algorithm's needs to another, during the application's run is beyond the scope of this paper. Also, it is difficult to say if recommending a new algorithm when a process migrates is really worth the cost because the algorithm might be in the final stages of its run.

A final aspect of our algorithm selection system is designed to remove the assumption that every algorithm/library is available on every resource. In a grid environment, it is unreasonable to expect that all the necessary software can be found on the resources that are selected to run the program. The idea is to make use of dynamic loading of libraries. When the RAS Engine attempts to instantiate a particular algorithm, it dynamically loads the library corresponding to the algorithm. This call to the loader is intercepted. It is checked whether the necessary library is present on the local resource. If so, the library can be loaded and the call completed. However, if

the library is not present, a server should be contacted to download the necessary library to the local resource. However, this feature (shown as the interceptor module in Figure 1) is not yet available in our system. We plan to investigate the NetBuild [11] system as a way to meet this need.

3.3 Distributed Meta-Scheduler

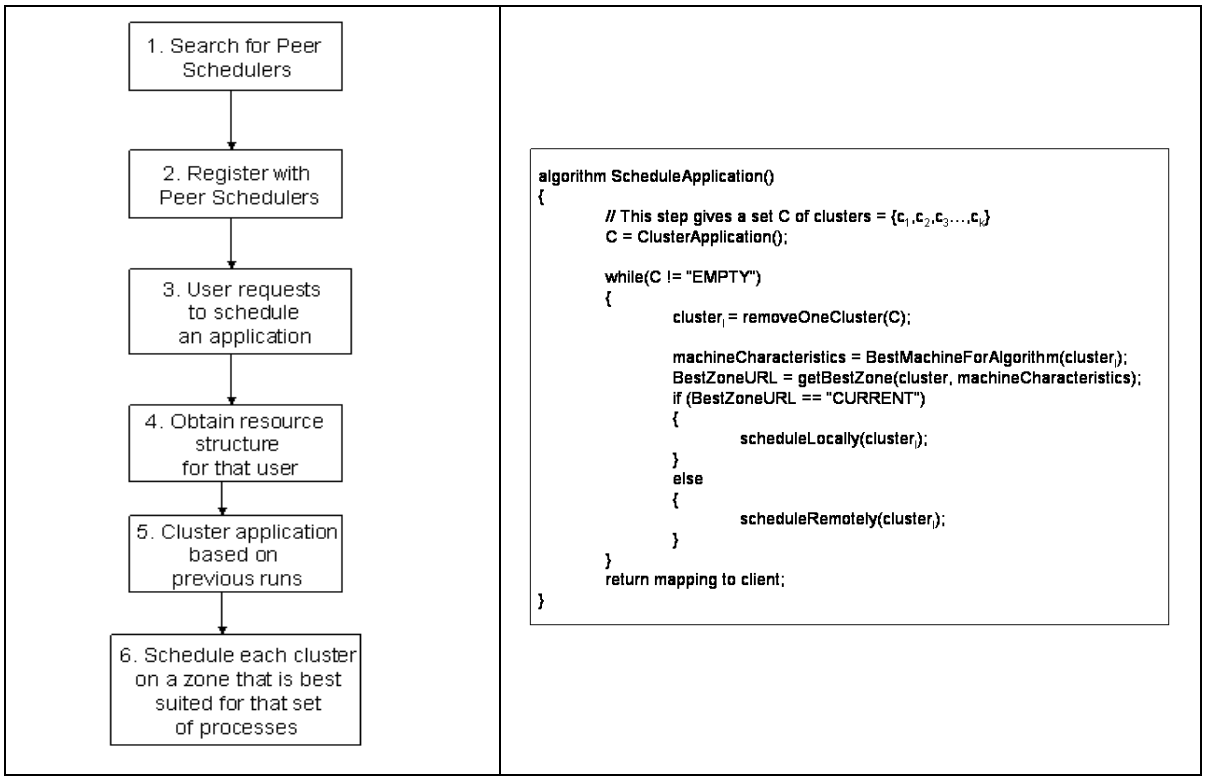

Figure 3: Phases involved in meta-scheduling (left) and algorithm used (right).

Design of the Scheduler

To schedule a parallel application, we need to perform a mapping from the processes of the application to the resources. Essentially this problem involves mapping the task graph (representing the application processes) to the resource graph (representing the resources). Both the graphs are weighted with weights on both the nodes as well as the edges.

The mapping is done as follows. First we identify subsets (or "clusters") of processes that communicate extensively among themselves. The clustering operation is based on the application's communication patterns from previous runs, and results in a certain number of clusters that is not fixed ahead of time (unlike many clustering algorithms that first fix the number of clusters 'k' and then perform clustering to obtain 'k' clusters). The clustering algorithm stops when the ratio of the decrease in inter-cluster communication to overall communication does not change significantly. Thus the application is split into parts, with each part interacting heavily within itself and relatively infrequently with other parts.

Second, the clusters are scheduled such that each cluster runs in a single zone. The selection of the zone to schedule each cluster depends on the structure of that cluster as well as the structure of the resources in the zones. We run an algorithm that selects the best zone for that cluster based on both these characteristics. Then the scheduler on the zone that was selected to

run the cluster is contacted and requested to make a mapping. The scheduler makes a mapping and returns it to the scheduler in the user's zone. This process is repeated for every cluster of the application. In the prototype implementation we use circuit clustering [12] to obtain clusters and a mapping algorithm [13] to map processes to processors within a single zone. Each organization is free to choose its own clustering and mapping algorithms that suit the organization's policies.

Scheduler Implementation

The scheduler seeks to use performance information from previous runs of the application. To prevent environmental noise in the performance data from affecting scheduling decisions, only those features of the application program that are independent of the resources on which it executes (e.g., number of instructions executed and number of bytes communicated) have been considered.

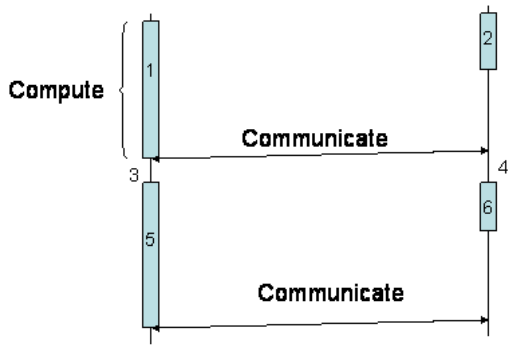

Figure 4: Structure of parallel programs

Typical parallel programs have a structure as shown in Figure 4, i.e., they have a sequence of computations and then they communicate their results. In order to minimize job completion time, the computation steps (steps 1 and 2 in Figure 4) at all nodes should take approximately the same time. This means that a computationally intensive process should be scheduled on a faster resource. Similarly the time for which processes block (3 and 4) should be minimized by placing processes that need to interact close to each other on the network.

Two components of information about the application are of use to the scheduler – the amount of computation between communication steps (granularity) required by each process, and the importance of communication between every pair of processes. Let us assume that there are n processes in the SPMD program. To represent the computation granularity, we construct a vector of size n, the i^{th} element of which has the average number of instructions executed at the i^{th} process between two communication operations. In our current implementation, a process fills in its computation requirement in this vector by making PAPI [6] calls every time an MPI function call is made. The averaged difference in the number of instructions executed between the last and the current MPI call gives the computation requirement of the process. We also construct a matrix of size n x n to represent the amount of communication between any two processes. Each element (i,j) of this matrix represents the communication between processes i and j. The communication volume between process pair (i,j) is obtained by means of the profiling interface of MPI. This allows us to obtain the number of bytes of communication between two processes. The larger this number, the more tightly coupled those processes are and the more likely those two processes will be scheduled on nearby resources. In addition, we also measure the frequency of communication. For numerical libraries, the frequency of communication also matters for cases like nearest neighbor topologies where the amount of communication is small but processes need to talk at every cycle. Each (i,j) element in the matrix of communication represents the importance of

scheduling processes i and j on nearby resources. Both the computation and communication parameters that we obtain are independent of the machine on which the application ran and hence can be used to schedule it on its next run, as the data is known not to have noise.

On the next run, the local scheduler identifies clusters in the communication matrix of the application and sends out each cluster of processes to be scheduled on their respective chosen zones. The local scheduler at that zone uses an algorithm like [13] to map processes to resources. The mapping algorithm in [13] maps a weighted task graph to a weighted resource graph heuristically. For the task graph, the weights on the nodes correspond to the number of instructions between two communication operations and the weights on the edges correspond to the communication requirement. For the resource graph, the weights on the nodes correspond to the anticipated load on that node and the weights on the edges correspond to network properties (e.g., latency, available bandwidth). The resource information is for the duration of the program's execution and this information is obtained using Network Weather Service (NWS) [5].

While the resource independent model of the application is useful in itself to make scheduling decisions, we show in Section 3.4 how a resource dependent application model is more useful to a scheduler. This resource dependent application model is obtained by interacting with the runtime algorithm selection system. The runtime algorithm selection system gives resource specific information about the structure of the application in a controlled fashion, isolating only the factors of the application that are directly related to the application and removing any noise.

3.4 Implication of a Runtime Algorithm Selection System for Meta-Scheduling

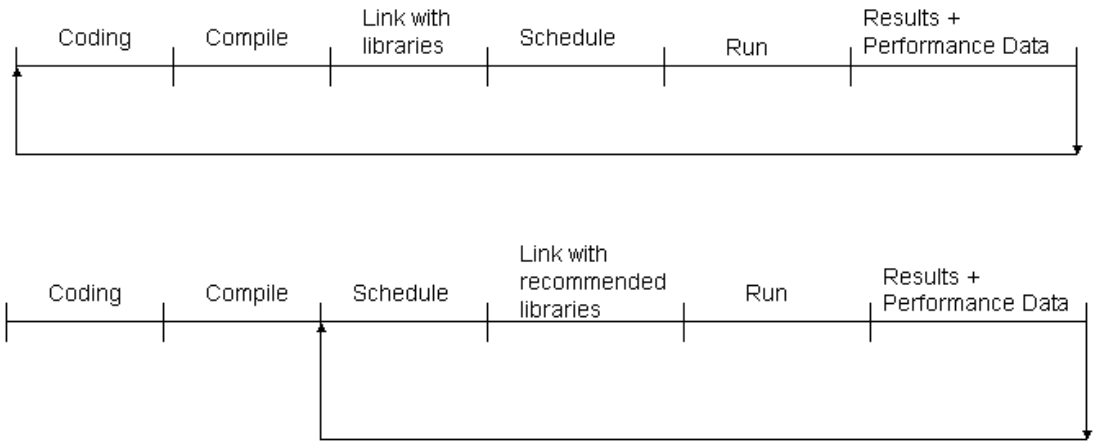

Figure 5: Performance improvement cycle for an application: without algorithm selection (top), with algorithm recommendation (bottom)

In Figure 5 (top), we see the typical development cycle of high-end computational programs. In order to improve performance of a particular code, the code is run, performance data is obtained and analyzed, and the code rewritten to improve performance, either by using a different algorithm, by changing parameter values or by choosing a different set of machines on which to run the application. In an effort to take the burden of trying different combinations of algorithms, parameters and machines off the application scientist, there is a need for middleware that will automate the process. We believe this middleware should include a runtime algorithm selection system and a meta-scheduler. Thus in Figure 5 (bottom), choosing a solver from a set of libraries and tuning the associated parameters, is done by the runtime algorithm selection system. However, this changes the sequence of operations that are done to improve the performance of the application. The scheduler can no more make intelligent decisions about which resources to

select for the application as this information is encapsulated in the runtime algorithm selection system. Thus there is a need for these two components to interact with each other.

A promising interaction scenario is as follows. The scheduler queries the runtime algorithm selection system to obtain information about what type of resources are best for the particular application. Using this information the search space of the scheduler will become more focused. A typical interaction scenario is depicted below. From prior runs of an application, the runtime algorithm selection system finds out performance information that is stored in a table such as this:

	Machine Characteristics	Algorithm Selected	Performance Data
Class 1	{Pentium II, cache size 1, memory size 1}	A1	Time 1
Class 2	{Alpha, cache size 2, memory size 2}	A2	Time 2

The two rows shown above represent the performance data of an application. This table is sorted by performance. Thus the application when run on Pentium II machines with cache size 1, memory size 1 and algorithm A1, performs better as compared to the second combination of machine type and algorithm. (For now, we assume "best performance" means lowest time.) Although the scheduler is not interested in the algorithm selected, the characteristics of the machines on which the application performed well is of importance to it. In the above example, the application will perform better if it is scheduled to run on "class 1" machines. Thus the scheduler will attempt to choose machines with similar characteristics when assigning processes to processors.

4. Conclusions

Grid computing has many systems interacting with each other. To obtain good performance, no system can be designed and implemented without taking into account its interaction with other grid middleware. We show how such interaction between a runtime algorithm selection system and a meta-scheduler is beneficial to the overall goal of obtaining best performance. A web service approach simplifies the interaction between the systems and at the same time allows each system to be developed independently.

References

[1] Prabhakar, S., Ribbens, C., Bora, P., "Multifaceted Web Services: An Approach to Secure and Scalable Grid Scheduling", Proceedings of Euroweb 2002, December 2002, Oxford, UK.

[2] Ribbens, C. J., Bora, P., Di Ventra, M., Hauck, J., Prabhakar, S., Taylor, C., "From Clusters to the Grid: A Case Study in Scaling-Up a Molecular Electronics Simulation Code", High Performance Computing Symposium, March 2003, Orlando (to appear).

[3] Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C., "Grid Information Services for Distributed Resource Sharing", Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[4] Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W., Tuecke, S., "Directory Service for Configuring High-Performance Distributed Computations", Proceedings of 6th IEEE Symposium on High-Performance Distributed Computing, pp. 365-375, 1997.

[5] Wolski, R., Spring, N., Hayes, J., "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", Journal of Future Generation Computing Systems, Vol. 15 No. 5-6, pp. 757-768, October 1999.

[6] Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D., "Using PAPI for hardware performance monitoring on Linux systems", Presented at Linux Clusters: The HPC Revolution, July 2001.
[7] Apache Axis homepage - http://xml.apache.org/axis/
[8] UDDI specifications - http://www.uddi.org/
[9] Stal, M., "Web Services: Beyond Component Based Computing", Communications of the ACM, Vol. 45 No. 10, pp. 71-76, October 2002.
[10] Kohn, S., Kumfert, G., Painter, J., Ribbens, C., "Divorcing Language Dependencies from a Scientific Software Library", 2000.
[11] Moore, K., Dongarra, J., "NetBuild: Transparent Cross-Platform Access to Computational Software Libraries", submitted to Concurrency: Practice and Experience, July 2001.
[12] Yeh, C. W., Cheng, C. K., Lin, T. T. Y., "Circuit Clustering Using a Stochastic Flow Injection Method", IEEE Transactions on Computer-Aided Design, Vol. 14 No. 2, pp. 154-62, February 1995.
[13] Taura, K., Chien, A., "A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources", 9th Heterogeneous Computing Workshop, May 2000.
[14] Vadhiyar, S. S., Dongarra, J., "A Metascheduler For The Grid", To appear in the proceedings of the HPDC 2002, July 2002.
[15] Dail, H., Casanova, H., Berman, F., "A Decoupled Scheduling Approach for Grid Application Development Environments", Proceedings of Supercomputing, November 2002.
[16] Dail, H., Casanova, H., Berman, F., "A Modular Scheduling Approach for Grid Application Development Environments", Submitted to Journal of Parallel and Distributed Computing [In review].
[17] Tapus, C., Chung, I., Hollingsworth, J. K., "Active Harmony: Towards Automated Performance Tuning", Proceedings of Supercomputing 2002, November 2002, Baltimore.
[18] Barrett, R., Berry, M., Dongarra, J., Eijkhout, V., Romine, C., "Algorithmic Bombardment for the Iterative Solution of Linear Systems: A Poly-Iterative Approach", Journal of Computational and Applied Mathematics, Vol. 74, No. 1-2, pp. 91-110, 1996,
[19] Dongarra, J., Eijkhout, V., "Self-adapting Numerical Software for Next Generation Applications", 2002.
[20] Whaley, R. C., Dongarra, J., "Automatically Tuned Linear Algebra Software", 1998.
[21] Ramakrishnan, N., Ribbens, C. J., "Mining and Visualizing Recommendation Spaces for Elliptic PDEs with Continuous Attributes", ACM Transactions on Mathematical Software, Vol. 26, No. 2, pp. 254-273, June 2000.
[22] Houstis, E. N., Catlin, A. C., Rice, J. R., Verykios, V. S., Ramakrishnan, N., and Houstis, C., "PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientfic Software", ACM Transactions on Mathematical Software, Vol. 26., No. 2, June 2000, pp. 277-253.
[23] Quinlan, J. R., "Induction of decision trees" Machine Learning, Vol. 1 No. 1, pp. 81–106, 1986.
[24] Muggleton, S., Raedt, L. D., "Inductive logic programming: theory and methods", Journal of Logic Programming, Vol. 19 No. 20, pp. 629–679, 1994.
[25] Dzeroski, S., "Inductive Logic Programming And Knowledge Discovery In Databases", In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, And R. Uthurusamy Eds., Advances In Knowledge Discovery And Data Mining, Pp. 117–152, AAAI Press/MIT Press, 1996.
[26] Bratko, I., Muggleton, S., "Applications Of Inductive Logic Programming", Communications of the ACM, Vol. 38 No. 11, pp. 65–70, 1995.
[27] Metz, B., Wienckowski, J., Ribbens, C., Di Ventra, M., "Performance of a parallel transport code for molecular electronics simulations," Department of Computer Science Report 02-05, Virginia Tech, Blacksburg, 2002.