Runtime Algorithm Selection For Grid Environments: A Component Based Framework

Prachi Champalal Bora

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Dr. Calvin J. Ribbens
Dr. Naren Ramakrishnan
Dr. Srinidhi Varadarajan

June 17th, 2003
Blacksburg, Virginia

Keywords: Grid Computing, Algorithm Selection, Linear Equations, Common Component Architecture, Recommender Systems.

Runtime Algorithm Selection For Grid Environments: A Component Based Framework

Prachi Champalal Bora

Abstract

Grid environments are inherently heterogeneous. If the computational power provided by collaborations on the Grid is to be harnessed in the true sense, there is a need for applications that can automatically adapt to changes in the execution environment. The application writer should not be burdened with the job of choosing the right algorithm and implementation every time the resources on which the application runs are changed.

A lot of research has been done in adapting applications to changing conditions. The existing systems do not address the issue of providing a unified interface to permit algorithm selection at runtime. The goal of this research is to design and develop a unified interface to applications in order to permit seamless access to different algorithms providing similar functionalities. Long running, computationally intensive scientific applications can produce huge amounts of performance data. Often, this data is discarded once the application's execution is complete. This data can be utilized in extracting information about algorithms and their performance. This information can be used to choose algorithms intelligently.

The research described in this thesis aims at designing and developing a component based unified interface for runtime algorithm selection in grid environments. This unified interface is necessary so that the application code does not change if a new algorithm is used to solve the problem. The overhead associated with making the algorithm choice transparent to the application is evaluated. We use a data mining approach to algorithm selection and evaluate its potential effectiveness for scientific applications.

Acknowledgements

Thank you: **Dr. Calvin J. Ribbens**, for guidance in numerical algorithms and for the caring and support that was provided throughout my progress. **Dr. Naren Ramakrishnan**, for helping with developing ideas and guidance with recommender systems. **Dr. Srinidhi Varadarajan**, for helping with the infrastructure required to conduct experiments. **Sandeep Prabhakar**, for being a good friend and keeping my morale high at all times.

Table Of Contents

Table of Figures

List of Tables

Chapter 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. Moreover, efficient application of scientific computing techniques requires specialized knowledge of numerical analysis, computer and network architectures, and programming languages that many researchers do not have time, energy and inclination to acquire. The goal of this thesis is to describe an approach and architecture for runtime algorithm selection for the Grid – a technology that can contribute much toward dealing with the heterogeneity and unpredictability of grid computing.

1.1 Algorithm Selection

There often exist a large number of different algorithms to solve the same problem. Also, there are cases when arbitrarily many variations of a single algorithm can be easily generated. This situation arises when the algorithm specification has one or more parameters, which can be set to any value from some continuous set of values. An example is the successive over relaxation (SOR) iterative method for solving linear systems of equations [1]. Different values of algorithm parameters lead to different rates of convergence for iterative methods such as SOR. Algorithm selection refers to choosing the most suitable computational structure for a given set of constraints from a set of functionally equivalent alternatives. Algorithm selection is a difficult problem [55].

The motivating applications for our research (e.g., [2]) 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 essentially 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 structures, 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.

1.2 Aims of Thesis

Heterogeneity of grid environments adds to the difficulty of the algorithm selection problem. The primary motivation of this research is to identify the requirements of a transparent runtime algorithm selection system for grid environments. Based on these requirements, we have designed a component based unified interface by which applications running on the grid can access software providing the same functionality. The idea of transparent algorithm selection is demonstrated using a high-performance scientific application that solves large linear systems of equations. The thesis first identifies the issues in hiding the linear algebra solvers behind a common interface. It then demonstrates how the unified interface helps the applications to switch between commonly used linear algebra solvers, without requiring the application writer to change the code.

The unified interface to algorithms involves some overhead. This thesis evaluates the overhead involved and discusses the attempts made to minimize both the time and memory costs.

A unified interface to allow seamless access to algorithms is beneficial. The choice of the algorithm used should be based on problem characteristics and the execution environment characteristics. We make use of a data mining based approach to algorithm selection. This approach is called Recommender Systems. Recommender systems take as input the performance data of applications from prior runs and suggest algorithms and associated parameters. The thesis also evaluates the effectiveness of this approach to runtime algorithm selection.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 first provides an introduction to numerical algorithms and techniques that are used to demonstrate the runtime algorithm selection concept. It explains prior efforts to application adaptation and algorithm selection. It also gives an introduction to the various component technologies available that can possibly be used to transparently switch between algorithms. Chapter 3 introduces the design of the algorithm selection system and describes the various pieces of the framework. It also explains design choices and implementation aspects of a transparent algorithm selector for large scientific codes. Chapter 4 evaluates the impact of introducing this middleware to perform solver selection and compares the benefit of switching between various solvers. Chapter 5 concludes with a summary of contributions of this thesis and directions for future research in the area.

Chapter 2. Background and Related Work

This chapter reviews the existing work in adapting applications to different conditions and environments. Application adaptation can be done either by means of tuning algorithmic parameters or by choosing a fundamentally different algorithm for solving the problem. Different techniques for selecting and adapting algorithms along with complete system architectures are discussed here. The chapter then reviews popularly used component technologies. Component technologies are important because they offer modularity and language interoperability. This is useful for algorithm selection because algorithms can be written as modular, reusable components. Language interoperability allows applications to be developed in a different language from the one in which the algorithms are implemented. This is particularly important for scientific applications with their large legacy code base. A survey of some linear algebra solvers used in this project is also presented. Section 2.6 discusses recommender systems – a data mining based approach to algorithm selection – and its relevance to this thesis.

2.1 Adaptive Algorithms and Algorithmic Tuning

The idea of adapting algorithms to resource and or problem characteristics has been a very fruitful one in scientific computing for many years. A well-known example is Automatically Tuned Linear Algebra Software (ATLAS) [3], which aims at automatically tuning numerical linear algebra kernels for specific architectures. ATLAS generates optimized Basic Linear Algebra Subprograms (BLAS) [23] 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. However, characteristics of a particular problem instance are not taken into account in making this optimization.

Computational Steering [4] allows a scientist to design and modify simulations interactively via a dataflow-programming model. The SCIRun [5] software system and the CUMULVS [6] library encapsulate such a capability. The SCIRun system enables scientists to design and modify models and automatically change parameters and boundary conditions as well as the mesh discretization level needed for an accurate numerical solution. However, this process is mostly manual and requires the programmer to instrument the application code to allow the parameter values to be extracted. Also, SCIRun targets single workstations and SMP environments; hence it is not suitable for the Grid where an application could be scheduled to run on a multitude of architectures. CUMULVS [6] allows user-defined parameters to be steered in a distributed simulation environment. It requires the user to instrument the code to be able to steer the parameters. It also allows a programmer to easily extract data from a running parallel simulation and send the data to a visualization package.

ILU tuning work at Boeing [7] chooses many different parameters determining ILU decomposition. The aim is to optimize either space or time, depending on the class of matrices. The approach however is specific to ILU decomposition and is not applicable to general algorithms. The choice of parameters is dependent only on the problem instance, the class of matrices in this case. Resources on which the application runs are not considered when making a choice of the parameters.

2.2 Algorithm Selection Techniques

Algorithm selection using reinforcement learning [8] tries to dynamically select an algorithm to solve a particular problem instance with the goal of minimizing the total execution time. Markov decision processes are used to model the algorithm selection problem by allowing multiple state transitions. The system uses a variation of a Q-learning algorithm adapted to account for multiple state transitions. They require that the programmer provide a set of algorithms providing same functionality and also a set of instance features to describe the problem. Variations in performance due to the resource characteristics have not been taken into account. The system does not consider distribution of input data and all the input data has been assumed to come from same uniform random distribution. The learning system employed is static and it needs a mechanism for continuous learning to be able to adapt rapidly to changing conditions.

The Adaptive Algorithm Selection Method (AASM) [9] was designed to dynamically tune software at run time. The system is built into the calling sequence of a library. When the library is called, AASM is activated and it selects and executes the optimum algorithm from the set of algorithms registered with the system, based on data and resource characteristics. Learning is based on neural networks. The performance test data of the registered algorithms is used to extract rules. These rules relate the problem features with the best algorithm to use. The size of the neural network depends on the input units (features used to select the algorithm) and output units (the number of algorithms registered). If the number of algorithms registered is large, the neural network becomes huge. Scalability of the neural networks is an issue that has not been addressed in the paper.

Algorithmic Bombardment [10] attempts to solve a problem by applying multiple algorithms at the same time and choosing the one that first returns the output. Although this polyalgorithmic approach guarantees the quickest response, it is expensive. The cost of applying different algorithms keeps growing as more algorithms become available to solve a particular problem. When a multitude of algorithms and resource types (architectures) are considered, the number of possible algorithm and resource-type combinations becomes prohibitive.

The Algorithm Portfolio [11, 12] approach is similar to the concept of polyalgorithms. It combines several algorithms into a portfolio and runs them in parallel or interleaves them on a single processor. As discussed earlier, the cost of applying several algorithms together is very high. Such an approach is unsuitable for computationally intensive problems where applications are expected to run for days together.

Adaptive constraint satisfaction [13] changes an algorithm during the course of an application's run, if the algorithm is found to be performing inefficiently. The system requires the algorithm developer / implementer to provide a measure of efficient performance. The system also requires that the algorithm writer provide a monitor that checks the algorithm's performance. Since different basic operations could be used to evaluate running time, if performance evaluation is left to the algorithm developer's discretion, there is no uniform evaluation and comparison function for the algorithms under consideration. Also, the paper does not discuss the costs of converting from the data-structures required by one algorithm to another when the application switches from one algorithm to another.

The Linear System Analyzer (LSA) system [14] provides a PSE (problem solving environment) to solve large, sparse, unstructured linear system of equations. It allows the user to interactively choose algorithm components to compose an application. This permits easy experimentation and exploration of the solution space, while eliminating the time and effort spent in writing complex code and compiling it. In this sense, it turns out to be more of a test bed for interactive user experimentation rather than a system with built-in intelligence for selection of algorithms and associated parameters for large production runs.

Grid ScaLAPACK [15] aims at developing parameterizable algorithms and software enabled with performance contracts. The performance contracts specify information about negotiation of resources and runtime adaptive strategies for controlling the path of execution. The system has a dynamic optimizer and run-time system that uses these performance contracts to make a resource-efficient algorithm selection. This is a work in progress and runtime algorithm selection is not yet implemented. The paper discusses algorithm choice based on resource characteristics, but problem parameters have not been taken into consideration.

2.3 System Architectures for Algorithm Selection

Active Harmony [16] is a system that allows runtime adaptation of algorithms, data distribution and load balancing. The system is composed of Library Specification Layer, Monitoring Component and Adaptation Controller. The Library Specification Layer allows different libraries providing similar functionality to be grouped together under a uniform interface. The Adaptation Controller is responsible for selecting a library from the different available implementations. The monitoring component is responsible for monitoring the application behavior. Active Harmony exports a metric interface to the applications, allowing them to access processor, network, and operating system parameters. The applications in turn expose tuning options to the system, which permits automatic optimization of resource allocation. The system requires that the libraries provide function calls in their APIs to support the measurement of performance metrics.

Our approach to algorithm selection adds a layer that takes care of measuring and recording performance data, thereby leaving the underlying library code unchanged. Thus, in our system, any new library can be incorporated without requiring any changes.

Self Adapting Numerical Software (SANS) [17] is another system that aims at selecting algorithms based on resource characteristics and problem features in order to deliver best performance in grid environments. The system collects performance data of an application in a history database. XML is used to define behavioral properties of algorithms and performance data in the database. The XML based vocabulary allows extensible definitions. The system has an intelligent agent that analyzes the history database to choose the best algorithmic strategy for solving the current problem. The agent has a system component that manages access to the computational grid. This is a work in progress and the learning techniques of the intelligent agent have not been discussed.

FRAME [18] is an adaptive software framework developed to adapt applications to specific computing environments. FRAME assumes a component view of the application to be adapted. It postpones the assembly of the application to execution time so that the resources on which the application has to run can be probed and decisions about which components to use can be made depending on the performance constraints defined by the user. FRAME is a Java based framework; hence applications developed are portable to any platform that supports Java. It does not take into consideration other programming languages, so, its applicability is limited. FRAME only considers resource constraints. The possibility of problem parameters affecting the performance is not taken into consideration. This system makes an implicit assumption that the computing environment does not change much. This assumption is not appropriate for grid environments, where resource structure and load is not guaranteed to remain the same.

NetSolve [19] allows transparent selection of algorithms. Users are given access to complex algorithms that solve a variety of problems, like solving linear systems of equations. To allow non-expert users to properly and efficiently use these algorithms without climbing the steep learning curve and needing to know under which conditions one algorithm is better suited as compared to others, the system allows the users to generically call a ‘LinearSolve’ routine which transparently analyzes the input matrix and determines which algorithm to use based on input characteristics. However, NetSolve runs the application on its own servers and returns the results to the users. The system is not usable on local resources.

Selection by Performance Prediction (SPP) [20] automatically selects a branch and bound algorithm for a search problem. Given a set of algorithms, SPP estimates the running time of every candidate algorithm in the set for a particular problem instance. It then selects the algorithm that gives the smallest expected running time. The problem with this approach is that it makes an estimate and does not consider the actual running time when the algorithm is applied. Given the heterogeneity of resources on the Grid, evaluation of an algorithm should be based on actual performance and not estimation.

The program transformation technique described by Burstall and Darlington [51, 52] replaces a function with an equivalent low-cost function (shorter execution time). Futamura [53] discusses a Partial Evaluation System that remakes the program by varying a parameter value and tries to reduce the execution time. The emphasis in Maes and Nardi [54] is on a Computational Reflection System that observes the program behavior from a meta-level and remakes the program when it does not meet user defined performance criteria. Although these techniques support dynamic tuning, they are expensive to execute and do not have a sophisticated decision making process to determine the best algorithm.

2.4 Linear Solvers and Libraries

Although any application can benefit from algorithm selection, we believe that long running, performance critical scientific applications are an especially promising class of applications that can make use of runtime algorithm selection. The advantage is two fold. First, applications will show good performance in varied conditions. Second, this will save the end user from having to write and maintain different versions of the application code employing different algorithms. A wide variety of linear algebra solvers and packages exist. However, this section discusses only a few used in this project.

Linear Algebra Package (LAPACK) [22] is a library that provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, algebraic eigenvalue problems, and singular value problems. These routines take a direct approach, in that they perform a factorization such as Gaussian elimination on a given matrix to get upper and lower triangular matrices. Then the system is solved using forward and backward substitution on the desired vector. LAPACK also provides different factorization techniques required for 'direct solution' of the problems. LAPACK uses block-partitioned algorithms to minimize frequency of data movement between different levels of memory hierarchy. It makes extensive use of Basic Linear Algebra Subroutines (BLAS) [23], which implement basic operations like matrix multiplication, rank-k matrix updates, and the solution of triangular systems. LAPACK is suitable for dense and banded matrices, but not for general sparse matrices.

ScaLAPACK stands for ‘Scalable LAPACK’ [24]. This library is similar to LAPACK in its functionality. LAPACK is to be used on workstations and shared memory computers, while ScaLAPACK is usable on distributed memory machines. The basic aim is to keep performance per node constant as the problem size scales with the number of nodes. ScaLAPACK uses a two-dimensional block cyclic decomposition technique to distribute the matrix over all the nodes participating in the computations. Just like LAPACK, ScaLAPACK uses block-partitioned algorithms and direct techniques to solve linear systems. It uses distributed memory versions of the Basic Linear Algebra Subprograms called the PBLAS (parallel BLAS).

Parallel Linear Algebra Package (PLAPACK) [25] takes an object-oriented approach to provide a high-level abstraction for solving linear systems. It hides the details of how the parallel linear algebra solvers distribute the data. PLAPACK provides Cholesky, LU, and QR factorization based solvers and is suitable for dense matrices only.

Iterative solvers obtain the solution of a linear system of equations by applying successive approximations to obtain more accurate solution at every step of iteration [1]. These solvers are often used for linear systems involving sparse matrices. A few iterative solvers that are used in this research are discussed here.

Generalized Minimal Residual (GMRES) [26] can be used to solve symmetric as well as nonsymmetric linear systems. It computes a sequence of approximate solutions, defined in terms of a growing set of basis vectors. The algorithm stores all the basis vectors computed in earlier iterations. There is also a ‘restart’ version that avoids large storage requirements of previously computed vectors. This means that after a chosen number of iterations, the accumulated data is cleared and the intermediate results are used as initial data for the next set of iterations. The crucial element of successful application of GMRES is the restart parameter that decides when to restart. The best value of restart parameter varies from problem to problem. However, there are no rules governing the choice of the restart parameter and choosing the right value is a matter of experience. Very often, users use a default value provided in a particular implementation of GMRES. However, this default value does not guarantee the best performance.

The Conjugate Gradient (CG) [27] method is used for symmetric positive definite systems. It generates vector sequences of successive approximations to the solution, residuals corresponding to the approximations and search directions used in updating the approximations and their residuals. Although the vector sequences can be large, only a small number need to be kept in memory. The CG method is not suitable for nonsymmetric systems. A variant of CG, the BiConjugate Gradient (BICG) method, can be used with nonsymmetric systems. However, the BiCG method displays some irregular convergence patterns. A variant of BiCG, called the BiConjugate Gradient Stabilized (Bi-CGSTAB) [35] method, overcomes this problem of irregular convergence. Bi-CGSTAB was used in the experiments conducted for this thesis.

The Quasi Minimal Residual (QMR) [28] method also attempts to overcome the irregular convergence patterns of the BiCG method. The convergence behavior of QMR is smoother than BiCG. QMR requires a matrix-by-vector product with not only the coefficient matrix A, but also its transpose A^T. However, data generated using the transpose does not directly contribute to the solution. The ‘transpose-free’ version of this method, called the TFQMR (Transpose Free QMR) eliminates the need to compute the transpose. This is useful for the class of applications for which the coefficient matrix is not available explicitly. In such situations, the transpose matrix also is not available. The convergence rate of TFQMR is comparable to BICGSTAB.

The algorithms described above are most commonly implemented in legacy languages like C and FORTRAN. However, there is a increasing interest in the scientific computing community in object based approaches. For example, The Equation Solver Interface (ESI) Standards [21] project aims at developing a set of standards for equation solver services and components. These are represented as an interoperable set of interface specifications. While object-based libraries such as this make runtime algorithm selection easier to implement, they do not help with the large quantity of legacy scientific applications and libraries.

2.5 Component Technologies

Cactus [29] defines a modularized, component-based approach for developing high performance parallel applications. The Cactus framework has a core called the ‘flesh’ and modules called the ‘thorns’. The flesh, independent of all thorns, provides the main program to activate appropriate thorns and pass control to the thorns when required. The flesh contains the basic utility routines used for communication between the thorns. The thorns are the basic working modules within Cactus. All user-supplied code goes into thorns, which are independent of each other. A thorn communicates with other thorns using the flesh API and the interface of other thorns. Thorn APIs are written in the Cactus Configuration Language (CCL). Multiple thorns providing the same functionality can co-exist, but only one of them can be active while the application runs. The thorns to be used are selected and configured by the user before the application starts running. If this process of thorn selection and configuration is automated, Cactus could be used for composing an application at runtime.

Microsoft’s Component Object Model (COM) [30] is a component software architecture that allows applications and systems to be built from components supplied by different software vendors. COM defines a binary standard for component interoperability, but it is not dependent on any particular programming language. Applications interact with each other and with the system using a collection of interfaces. The operating system acts as a central registry for objects developed using the COM binary standards. The OS is also responsible for managing the lifetime of the objects. This means that the OS creates the objects when required and deletes them when they are no longer in use. A distributed version of the component model called Distributed Component Object Model (DCOM) is used for distributed applications. Both COM and DCOM models are business-oriented and do not have special support for scientific applications.

Common Object Request Broker Architecture (CORBA) [31] aims at providing language and vendor interoperability between objects in a heterogeneous, distributed environment. The CORBA standards are based on the Object Management Group’s (OMG) Object Model. Objects (also termed as servers) are required to expose their functionality through well-defined interfaces written using the Interface Definition Language (IDL). The client requests services from servers through these interfaces. Like COM, CORBA also targets commercial applications and does not have special accommodations for scientific component development.

Babel [32, 33, 34] is a component framework targeted at facilitating code reuse in parallel scientific applications. It provides language interoperability. It requires that the functionality provided by a component be 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 supports FORTRAN style dynamic arrays and complex and double complex data types to facilitate use of Babel in 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 the interfaces defined in the SIDL. Currently, Babel supports FORTRAN, C, C++, Java and Python.

Babel also has a XML based repository called Alexandria. User defined SIDL files are converted to XML representation and stored in the repository. Alexandria provides a hierarchical collection of components, a search capability to locate these components and a web based user interface to the Babel's language interoperability tool. The XML information allows a Babel user to view the interface definitions without having to download the SIDL files.

2.6 Recommender Systems

Recommender systems [36, 37] are an approach to studying algorithm performance based on data mining. Recommender systems help application scientists in making selections from varied choices of algorithms and software. The fundamental idea is to accumulate performance data for a set of test problems and algorithms. This accumulated data is mined (generalized) to obtain a set of rules that can be used to provide recommendations (suggestions) for related problems. These are different from adaptive algorithms in the sense that the adaptive systems are designed for specific applications while the recommender systems approach can be applied to any application or problem provided the performance data is available.

Pythia-II [38] is recommender system based software that provides recommendations about scientific software. A user of Pythia-II specifies a problem to be solved and the computational goals and requirements of the problem. Pythia-II selects the algorithm/software to be used for that particular problem instance. In addition to selecting the algorithm, Pythia-II also suggests values of algorithmic parameters. When the application completes its execution, the system evaluates the recommendations provided in order to learn and improve on future recommendations. Pythia-II assumes that enough data is available for learning about algorithms and providing recommendations. This data could be obtained from prior runs of the same application or from related problems in the same domain.

Pythia-II has four layers: user interface layer, database layer, relational engine layer, and the data generation and mining layer. The user interface layer allows the knowledge engineer (recommender system builder) to use the system to derive rules and query the database. It is also used by an end user as a 'recommender' to obtain domain specific information about the algorithms to be used and associated parameter values. The database layer stores information about problems, their features and performance data. The relational engine layer provides access to the database for the user interface and data mining layers. The data generation and mining layer provides two functionalities. First, it generates performance data for the application. Second, it does statistical analysis of the performance data and mines the data to extract patterns and generate rules. This layer uses two learning methods for extracting patterns: Case Based Reasoning [39] and Inductive Logic Programming [40, 41]. The MyPythia Grid portal [42] provides an interface to the recommender systems – for both the knowledge engineer and the end user.

Recommender systems can work in two modes - offline or at runtime. If the mode of operation is offline, the database of performance information is collected first and mined to obtain recommendation rules. The generated rules are used to obtain recommendations about which software or algorithm to use. The suggested algorithm is then linked with the application and the application is executed. This offline mode of operation still needs manual intervention for linking the appropriate software with the application. Runtime systems are more sophisticated in that the performance data is captured and mined on the fly. Thus, a runtime recommender system interacts with its environment dynamically and learns through these interactions. The runtime mode of operation is important because certain application characteristics cannot be known in advance and are available only when the application is run. For example, sparsity of coefficient matrices in linear systems of equations is not known before the matrix is constructed.

Scientific applications are developed over years and used by scientists and engineers for a long time. These applications are computationally intensive and can benefit from the right choice of algorithms. As applications are used for a long time, huge amount of performance data can be gathered, making them very suitable for using the recommender systems approach for algorithm selection. This thesis is targeted at using recommender systems for selecting algorithms at runtime in heterogeneous grid environments.

Chapter 3. Design and Implementation

This chapter discusses the requirements of a runtime algorithm selection system for Grid environments. Then a high level architecture of the system is described. The API of the system as implemented and its functioning are explained. This chapter also provides information about design choices made in transparent algorithm selection for linear solvers. Existing implementations of the motivating application, `transport`, are also discussed. The chapter concludes with a summary of the changes required in the `transport` code in order to use the API for runtime algorithm selection.

3.1 Requirements of a Runtime Algorithm Selection System

The goal of this research is to facilitate good algorithm recommendation, in the dynamic and heterogeneous context of the Grid, and without putting a significant burden on the application or the user. Toward that end we seek to design 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 and matrix sparsity in the case of linear equations, resource features like interconnection network performance, available swap space, and cache size can affect the performance of an algorithm. These factors are not known when the application is compiled. Information about these can only be gathered at runtime.

- **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.

- **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.

3.2 System Architecture

The runtime algorithm selection system consists of several components (see Figure 3.1). 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 [32] 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 3.2 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 3.1 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 the 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., block size for data distributions or restart parameter for Generalized Minimal Residual (GMRES) algorithm used to solve linear systems.

The RS depends on the availability of a sizeable amount of data on similar types of problems in order that a good recommendation is 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).

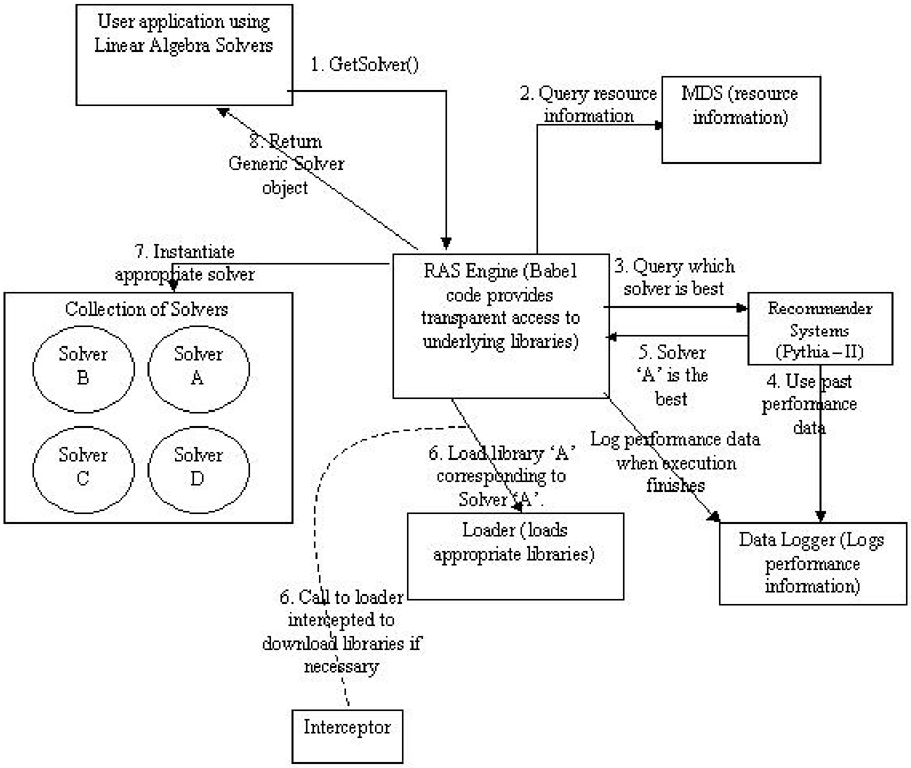
User application using Linear Algebra Solvers
1. GetSolver()
2. Query resource information
MDS (resource information)
8. Return Generic Solver object
7. Instantiate appropriate solver
Collection of Solvers
Solver B
Solver A
Solver C
Solver D
RAS Engine (Babel code provides transparent access to underlying libraries)
3. Query which solver is best
Recommender Systems (Pythia – II)
5. Solver 'A' is the best
4. Use past performance data
Log performance data when execution finishes
6. Load library 'A' corresponding to Solver 'A'.
Loader (loads appropriate libraries)
Data Logger (Logs performance information)
6. Call to loader intercepted to download libraries if necessary
Interceptor

Figure 3.1. Architecture of Runtime Algorithm Selection (RAS) System.

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 [39, 40, 41, 43]. 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.

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

```
Procedure GetRecommendation
{
 if(match pattern(resource-
         features, problem-
         features))
    Return matching solver
 else
    return default-solver
}
```

Figure 3.2. Pseudo code of the RAS Engine (left) and RS (right).

Since recommendations are based not only on problem features, but also on resource characteristics, we use the Meta-computing Directory Service (MDS) [44] 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, architecture and memory. This resource information and problem features are given to the RS to obtain suggestions about which algorithm should be used.

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 are 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 machine where a process is running. 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 interesting future work suggested by this thesis. 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.

3.3 Using Babel as the RAS engine

This section illustrates how Babel can be used as the Runtime Algorithm Selection Engine (RAS Engine) by means of a simple 'Hello World' example.

Babel's object model is very similar to that of Java. Three types of objects can be defined in Babel's Scientific Interface Definition Language (SIDL). These are: interfaces, classes and abstract classes. A SIDL interface is similar to a Java interface. It defines methods, but carries no implementation for those methods. Interfaces cannot be instantiated. As compared to interfaces, classes are more concrete. They have implementations for all their methods and can be instantiated. An abstract class lies in between a class and an interface. It has at least one method unimplemented, because of which it cannot be instantiated. However, the abstract class may have several methods implemented, which can be inherited.

SIDL supports multiple inheritance of interfaces and single inheritance of implementation. These two forms of inheritance are distinguished with the keywords '*extends*' and '*implements*'. This means that interfaces can extend multiple interfaces, but they cannot implement anything. Classes can extend at the most one other class, but can implement multiple interfaces.

The SIDL runtime library defines a set of objects that are implicitly inherited by user defined interfaces and classes. All classes that do not explicitly extend another class implicitly extend *SIDL.BaseClass*. All interfaces that do not explicitly extend another interface implicitly extend *SIDL.BaseInterface*. Moreover, *SIDL.BaseClass* implements *SIDL.BaseInterface.* This means that all classes can be cast to *SIDL.BaseClass* and all objects can be cast to *SIDL.BaseInterface*. It is this feature of Babel's object model that is used to build the RAS Engine.

```
version Hello 1.0;
package Hello
{
 interface World
   {
   string getMessage();
   }
 class EnglishWorld implements-all World {};
 class HindiWorld implements-all World {};
}
```

Figure 3.3. SIDL definition for 'Hello World' application.

Figure 3.3 shows the SIDL definition for a simple 'Hello World' application. In this example, all the interfaces and classes are contained in the package *Hello*. The *World* interface has a method called *getMessage* that returns a string value. *EnglishWorld* and *HindiWorld* classes implement all the methods of the *World* interface and thus will provide their own implementations of the *getMessage* method. As mentioned earlier, all classes can be cast to *SIDL.BaseClass* and all objects can be cast to *SIDL.BaseInterface*. Thus, both *EnglishWorld* and *HindiWorld* classes can be cast to *SIDL.BaseInterface*. The subsequent paragraphs describe how this feature can be used for runtime loading of classes.

Figure 3.4 shows how a selection between different classes implementing the same interface can be made. For illustration purposes, this choice is based on an integer parameter passed to the function called getAWorld. In the example, if the input is zero, *EnglishWorld* class should be instantiated; else if the input is non-zero, *HindiWorld* class should be instantiated. For this, we make use of Babel's runtime library called the *SIDL Loader*. This runtime library provides dynamic type identification for Babel objects. As seen in the figure, if the input is zero, *SIDL Loader* is invoked to create a class of type *Hello.EnglishWorld*. The reference obtained to this object is then cast to *SIDL.BaseInterface* (since every Babel object derives from *SIDL.BaseClass*, which in turn derives from the *SIDL.BaseInterface*, every Babel object can be cast to *SIDL.BaseInterface*). This generic reference of type *SIDL.BaseInterface* is returned to the calling application. When the calling application invokes the method *getMessage* on the reference obtained, the SIDL Loader determines the type of the object dynamically to route the method call to the appropriate class, namely *EnglishWorld* or *HindiWorld* in this case. This is very similar to runtime polymorphism offered by C++ [45]. In C++, a derived class can provide its own implementation of a virtual function defined in the base class. The function to be invoked (defined in base class or derived class) is determined at runtime based on the type of object that the method is invoked on.

```
Hello_World getAWorld (int i)
{
  Hello_World chosenWorld;
  SIDL_BaseInterface base;
  if(i = = 0)
      base = (SIDL_BaseInterface) SIDL_Loader_createClass
             (“Hello.EnglishWorld”);
  else
      base = (SIDL_BaseInterface) SIDL_Loader_createClass
             (“Hello.HindiWorld”);
  chosenWorld = (Hello_World) Hello_World__cast(base);
  return chosenWorld;
}
```

Figure 3.4 Code fragment to choose between implementations.

Thus, SIDL Loader's dynamic type identification capability can be used to choose between any number of classes implementing the same interface. We employ this idea in choosing linear algebra solvers transparently. For this to work, linear algebra objects (matrices and vectors) and solvers are represented as Babel objects. The SIDL design for these objects is discussed in detail in the next section.

3.4 SIDL Design and API

If scientific applications have to use linear solvers transparently, the different data distributions employed by different solvers should also remain transparent. Hence, there is not only a need to hide the solvers behind a middleware layer, but the particular distributed data structures defining the linear operators and the process topology should be hidden too. Only generic functions like accessing matrix elements, performing a matrix-matrix multiply or operations to solve a linear system must be exposed to an application.

When an application starts running, the choice of linear solver to be used is made depending on the parameters that define the problem instance and the resources on which the application is running. This information regarding problem instance, resource features, recommended solver and process topology information is encapsulated in an object that is termed *Environment*. This encapsulation is necessary for two reasons. First, every linear operator that is created should adhere to the data representation and data distribution requirements of the solver that will be used. Second, by storing the solver information in the *Environment*, the Recommender System needs to be consulted only once, rather than each time a linear operator or solver object is created. Thus, when an application starts running, it obtains a reference to its *Environment*. It then uses this *Environment* object to create linear operators and solver objects. (Note, however that an application could close an *Environment* and create a new one in order to take advantage of some new problem or resource information.)

Figure 3.5 shows the *Environment* interface. When an application intends to use a parallel linear solver, it first needs to define a process topology. Since the most common topology for linear algebra computations is a two-dimensional grid, we assume from here on that the topology is defined by number of process rows *nprow* and number of process columns *npcol.* Different solvers use different methodologies and communication libraries to form this process grid. This grid formation in an application using the runtime algorithm selection system will take place through the '*initialize*' method. This method takes in as parameters an MPI communicator whose members will solve a linear system in parallel by forming a process grid, and two integers that represent the number of process rows and columns in the process grid. Processes often choose control paths depending on their rank in a communicator or position in the process grid. To obtain the information about a process' location in the grid, the '*environmentInfo*' method is provided; it returns the calling process' row and column position in the grid. The '*createMatrix*' and '*createVector*' methods are used to create matrices and vectors in representations required by a particular solver. The input parameters for these methods are the dimensions of the objects to be created and the block sizes to be used to distribute these over the process grid. Block sizes are irrelevant when sequential solvers like LAPACK are used since only a single process performs all the operations and there is no data distribution. However, application writers do not know in advance whether a parallel solver will be used or a sequential one. Since this is a generic interface, accommodations have been made for block sizes, which are important for parallel solvers and can simply be ignored in the case of sequential ones.

```
interface RAS.Environment
{
  void initialize (in int communicator, in int nprow, in int npcol);
  void environmentInfo (out int myrow, out int mycol);
  Matrix createMatrix (in int m, in int n, in int mb, in int nb);
  Vector createVector (in int m, in int mb);
  Solver createSolver ( );
  bool isOwnSupported ( );
  void close( );
}
```

```
class ScalapackEnvironment implements-all Environment
{
   int getnpRow ( );
   int getnpCol ( );
   int getmyRow ( );
   int getmyCol ( ) ;
}
```

```
class LapackEnvironment implements-all Environment
{

}
```

Figure 3.5. The Environment interface and solver specific classes that implement this interface

Different solvers differ not only in the data distribution but also in the extent to which that distribution is hidden or visible. Data distribution and its visibility to the application make a difference in the way applications are written and parallelized. For example, PLAPACK hides the manner in which data is distributed. Given an index in a matrix or vector, there is no way a process can find out whether it resides locally or on another process. In PLAPACK based applications, different processes work on different blocks of data irrespective of whether the process owns it or not. PLAPACK takes care of data updates on remote processes. As opposed to this, ScaLAPACK supports functions that, given an index in a matrix or a vector, return the home process for that particular element. Applications are typically parallelized based on where data elements reside. Generally, processes operate on the elements that reside locally (the ‘owner-computes’ rule), which may not be contiguous in the global view of the matrix. Considering these different data distribution policies of different solvers, an important question in encapsulating these differences in a uniform way arises. If data distribution were hidden as in PLAPACK, process synchronization methodologies would have to be provided for packages like ScaLAPACK to update data on a remote process. This means that there would be a performance degradation of the solver package due to additional communication for data updates. On the other hand, since packages like PLAPACK do not support knowing the data location, the middleware cannot tell whether the data resides locally or not if solvers like these are chosen for performing the computations. Hence it was decided to support a function called '*isOwnSupported*' that returns false if the underlying solver does not allow knowing the data location, and true otherwise. The burden of parallelizing the application based on the knowledge of ownership is left to the application writer.

Method '*close*' is called when a solver environment is no longer needed. It releases all the state associated with the environment. For example, it will release the process grid. Classes *ScalapackEnvironment* and *LapackEnvironment* (Figure 3.5) are examples of different solver environments. These classes implement all the methods of the *Environment* interface. Additional functions are defined for each class to access the state associated with it (similar to get and set methods for private data members in C++ classes) to adhere to object oriented design. However these class-specific functions are not visible to the application. Library writers encapsulating linear solvers in Babel format will provide these functions and make use of them internally. Each solver package would need to supply a concrete class implementing the *Environment* interface. The concrete classes shown in the figure are for illustration purposes only and are not meant to be a complete list of all available linear solver packages.

Matrices and vectors form the basis of any system of linear equations. The runtime algorithm selection system provides an interface for a matrix and a vector. Figure 3.6 shows the *Matrix* interface and examples of LAPACK and ScaLAPACK matrix classes. As in the *Environment* interface, implementation classes shown in the figure are not a complete list, but just representative examples. A matrix object should first be created using the *createMatrix* method of the *Environment* interface. This returns a reference to a matrix object, which can then be used to invoke the functions shown in Figure 3.6. Any number of matrix objects can be created by an application using this create function.

Method *setToZero* initializes all the matrix elements to zero. Given an index (i,j) in the matrix, *setValue* and *getValue* set and retrieve the value at location (i,j), respectively. The semantics of the retrieval and set functions are dependent on whether the underlying solver library hides the data distribution or not. If the linear algebra package hides the data distribution and, as in PLAPACK, allows any process to manipulate any element at any location in the matrix, these methods set and retrieve the values corresponding to those locations. However if the underlying library does not hide the data distribution and, as in ScaLAPACK, allows manipulation of local data only, these functions will set and retrieve values corresponding to index (i,j) only if they reside locally. In this case, the functions return –1 if the data does not reside locally. An application writer should use these functions in conjunction with *isOwnSupported* of the *Environment* interface and *doIOwn* method of the Matrix interface. The method *doIOwn* returns a true or false for an element location depending on whether it resides locally or not. The method *whoOwns* returns the process rank of the owner of a given matrix element. Again, the operation is valid only if the associated environment returns true for *isOwnSupported*. The value returned is undefined otherwise. Method *addValue* adds a value to the element at location (i,j); semantics of this method with respect to data location and ownership are the same as for *setValue*.

```
interface RAS.Matrix
{
  void setToZero ( );
  int getValue (in int i, in int j, out dcomplex value);
  int setValue (in int i, in int j, in dcomplex value);
  int addValue (in int i, in int j, in dcomplex value);
  void getGlobalSize (out int m, out int n);
  void getLocalSize (out int locr, out locc);
  void getBlockSizes ( out int mb, out int nb);
  void matrixMultiply (in char transa, in char transb, in int m,
                      in int n, in int k, in dcomplex alpha,
                      in dcomplex beta, in Matrix B, inout Matrix C);
  void vectorMultiply (in char transa, in int m, in int n,
                      in dcomplex alpha, in dcomplex beta,
                      in Vector x, inout Vector y);
  bool doIOwn (in int i, in int j);
  int whoOwns (in int i, in int j, out int ownerRow,
               out int ownerCol);
  double getMaxReal ( );
  double getMaxImaginary ( );
  void compareAndSetReal (in double value);
  void compareAndSetImaginary (in double value);
  void sendGlobal (in dcomplex value);
  dcomplex getGlobal (in int i, in int j);
  void close( );
}
```

```
class ScalapackMatrix implements-all Matrix
{
   void setScalapackData (in  array<int,1> descriptor, in int m,
                          in int n, in int mb, in int nb, in int locr,
                          in int locc, in Environment env);
   array<int,1> getDescriptor( );
   array<dcomplex, 1> getElements( );
   void setElements (in array<dcomplex,1> e);
}
```

```
class LapackMatrix implements-all Matrix
{
   void setLapackData (in int m, in int n, in Environment env);
   array<dcomplex, 1> getElements( );
   void setElements (in array<dcomplex,1> e);
}
```

Figure 3.6. The Matrix interface and example classes

The methods *getGlobalSize* and *getLocalSize* return the global and local dimensions of the matrix, respectively. In case of sequential solvers like LAPACK where all the data is stored in a single process, global and local dimensions are the same. Very often, applications need to know how many elements are stored locally, e.g., to allocate memory for proportionally sized data structures. For example, `transport` does disk-cache manipulation with the matrix elements to reduce memory requirements and to save computations for later use. For this, the application needs to know the disk-buffer size to be allocated for the matrix elements that are stored locally.

The method *matrixMultiply* when invoked on a matrix object 'A', performs the operation C = αAB + βC. Inputs *transa* and *transb* denote whether or not the transpose of the matrices are to be used for multiplication. Likewise, the method *vectorMultiply* when invoked on a matrix object 'A', performs the operation y = αAx + βy. An attempt has been made to keep the semantics of these multiplication methods as similar as possible to those of the methods provided by the Basic Linear Algebra Subprograms (BLAS).

A few functions of the *Matrix* interface are inspired by the needs of our original motivating application `transport` (this application is described in detail in the Section 3.5). These functions are: *getMaxReal, getMaxImaginary, compareAndSetReal and compareAndSetImaginary*. If the matrix implementation belongs to the class of solvers that do not hide data distribution from the application, (i.e., *isOwnSupported* function of the *Environment* interface returns true) the methods *getMaxReal* and *getMaxImaginary* respectively return the maximum absolute value of real and imaginary components of the locally stored elements of the matrix. If the application wishes to obtain the maximum value in the global context, it explicitly needs to communicate between its processes to find out the global maximum, e.g., using an MPI_AllReduce function call. However, if the data distribution is hidden, the maximum values in the global context are returned. Likewise, methods *compareAndSetReal* and *compareAndSetImaginary* check the matrix elements and replace the real (or imaginary parts) if they are found to be smaller than the value passed as input for these functions. The corresponding imaginary (or real) part remains the same. Again, these functions act on local data if *isOwnSupported* returns true. If *isOwnSupported* returns false, the compare and set operations are global.

The *sendGlobal* and *getGlobal* methods were also inspired by `transport`. The matrices in this application have four-fold symmetry and hence, computations are done only for one quadrant of the matrix. The computed values are duplicated in symmetric locations. While a process may own a particular element of a matrix, it may or may not own the elements stored at corresponding symmetric positions. When a process needs to send an element value to other processes, it does a *sendGlobal*, which does a broadcast of the value to all the processes in the communicator. Likewise, if a process needs a value stored at a location that it does not own, it does a *getGlobal*, which fetches the value from the owning process. Both these methods are global operations that must be called by all processes in the communicator. The method *close* releases all the state associated with a matrix (typically the memory allocated for storing the elements). The method should be called when the matrix is no longer needed.

Classes *ScalapackMatrix* and *LapackMatrix* are examples of solver specific implementations of the matrix interface. The class-specific functions of these classes are hidden from the user. Only a library writer is concerned with these functions to access solver specific information. For example, all the ScaLAPACK matrices have a *descriptor* [24] associated with them that store information about the way the matrix has been distributed. These class-specific functions are provided for accessing the private data of the class, which is solver specific information. Typically, there needs to be a get and set function for each of these private data members. However, as seen in Figure 3.6, the set functions for private data members that are set only once during the creation of the matrix object, and do not change for the lifetime of the matrix object, have been coalesced in a single function to avoid excessive function call overheads. For example, the distribution *descriptor* of a ScaLAPACK matrix, block size used for distribution and the matrix dimensions are determined when the matrix is created. They remain the same throughout the lifetime of the matrix object and need to be set only once. However, the values of the elements may change any number of times. Although an application might retrieve elements one at a time, functions like matrix-matrix-multiply and matrix-vector-multiply need access to the entire set of elements. Access to these elements has been provided via *getElements* and *setElements*. Again, these functions are visible only to the library writer who provides implementations of methods performing the multiplication operations and are not visible to the user.

Another important point to be emphasized here is that although the interfaces shown use only double-complex data, they are easily extensible for other precisions too. The intention here is to illustrate how solver specific information can be encapsulated in classes and have the user see only the generic functions that the application needs. The illustrations shown in this thesis are not meant to be a comprehensive list of all possible operations.

The *Vector* interface is very similar to the *Matrix* interface. The methods exposed by the interface are shown in Figure 3.7.

Figure 3.8 depicts the *Solver* interface. *Solver* in this context refers to the set of all operations that will help solve a linear system. Solvers can be either iterative or direct. For solving a linear system $Ax = b,$ direct solvers take in the matrix 'A' and factorize it first. There are different types of factorizations available – triangular factorization, Cholesky factorization, QR factorization, etc. The solution step follows factorization. Iterative solvers follow a different approach. They transform the problem to a suitable form by using a 'preconditioner' and solve the equation in several iterations. Again, there exist a wide variety of preconditioners – like ILU, block-Jacobi, block-diagonal, etc. When an application uses direct solvers, it first calls a factorize step to factorize the matrix. It then calls the solve step. When the application uses an iterative solver, it first initializes the preconditioner to be used and then calls the solve step using that preconditioner. If the solver choice has to remain transparent to the application, so should the different steps involved in the solution process. For this reason, the solver interface has a generic method called *setup*. This method does factorization in the case of direct solvers; in the case of iterative solvers, it initializes the preconditioner. The setup method returns an object called *SolverContext,* which contains information needed to define a particular solver. For example, for a general LU factorization solver, the pivot vector is stored in the *SolverContext*. When the application wishes to call the *solve* method, it passes the *SolverContext* object obtained earlier.

```
interface RAS.Vector
{
  void setToZero ( );
  int getValue (in int i, out dcomplex value);
  int setValue (in int i, in dcomplex value);
  int addValue (in int i, in dcomplex value);
  int getGlobalSize ( );
  int getLocalSize ( );
  void getBlockSize ( );
  bool doIOwn (in int i);
  int whoOwns (in int i, out int ownerRow, out int ownerCol);
  double getMaxReal ( );
  double getMaxImaginary ( );
  void compareAndSetReal (in double value);
  void compareAndSetImaginary (in double value);
  void sendGlobal (in dcomplex value);
  dcomplex getGlobal (in int i);
  void close( );
}
```

```
class ScalapackVector implements-all Vector
{
   void setScalapackData (in  array<int,1> descriptor, in int m,
                         in int mb, in int locr, in Environment env);
   array<int,1> getDescriptor( );
   array<dcomplex, 1> getElements( );
   void setElements (in array<dcomplex,1> e);
}
```

```
class LapackVector implements-all Vector
{
   void setLapackData (in int m, in Environment env);
   array<dcomplex, 1> getElements( );
   void setElements (in array<dcomplex,1> e);
}
```

Figure 3.7. Vector interface and implementation classes.

The *SolverContext* interface is shown in Figure 3.9. The interface does not have any methods other than close. This is because the application need not be concerned with issues like whether an iterative solver is used or a direct one, what kind of a pre-conditioner is used, etc.

```
interface Solver
{
   SolverContext setup (inout Matrix a);
   int Solve (in char transa, in int n, in int nrhs, in Matrix a,
             in SolverContext ctx, inout  Vector b);
   void close( );
 }
```

```
class LapackSolver implements-all Solver
{
}
```

```
class ScalapackSolver implements-all Solver
{
}
```

Figure 3.8. The Solver interface and derived classes.

```
interface SolverContext
{
   void close( );
}
```

```
class DirectSolverContext implements-all SolverContext
{
   void setPivots (in int size, in array<int, 1> pivotArray);
   array<int,1> getPivots( );
}
```

```
class IterativeSolverContext implements-all SolverContext
{

}
```

Figure 3.9. The SolverContext interface.

3.5 Motivating Application - Transport

Although the idea of runtime algorithm selection can be applied to any problem that can be solved using many different algorithms, we feel that one of the most promising classes for this approach is the scientific applications that solve linear systems using linear algebra solvers. This is because computationally intensive steps that make use of the linear algebra solvers dominate the scientific applications. Also, these applications generally run for a long time (days, sometimes even weeks), so that the efficiency achieved by making use of different solvers is evident and could make a huge difference to the turn-around time of the application. Moreover, such applications are used by researchers and scientists for years together and so can benefit the most from using algorithms suited to different settings and can use a long history of previous solves to obtain recommendations. The next few paragraphs describe such a motivating application called `transport`.

`Transport` [2] is an application developed at IBM and the Department of Physics at Virginia Tech to study the physical properties of nanoscale electronic devices. The application calculates the non-linear transport properties of molecular structures. A single run of `transport` on 1Ghz machines having 1 GB RAM and a 2.56 Gbps Myrinet interconnect can take up to a few weeks. The duration of a single run of `transport` depends on the experimental setup defined by the number of energy levels and the number of plane waves in x, y and z directions called N_x, N_y and N_z. The application needs to be run 100's or 1000's of times for different orientations of molecules to obtain a characteristic current-voltage (I-V) curve.

3.5.1 Overall Structure of Transport

`Transport` has a master-slave structure as shown in Figure 3.10. The master process assigns *energy levels* to the slave processes. An energy level corresponds to calculating the conductance properties of the molecule under consideration given one particular experimental setup. Each energy level step is dominated by the solution of one or two large dense non-symmetric systems of linear algebraic equations. Each of these equations is a matrix equation with multiple right hand sides for each matrix. Typical values are 32 to 128 energy levels and 64 to 256 right hand sides per energy level.

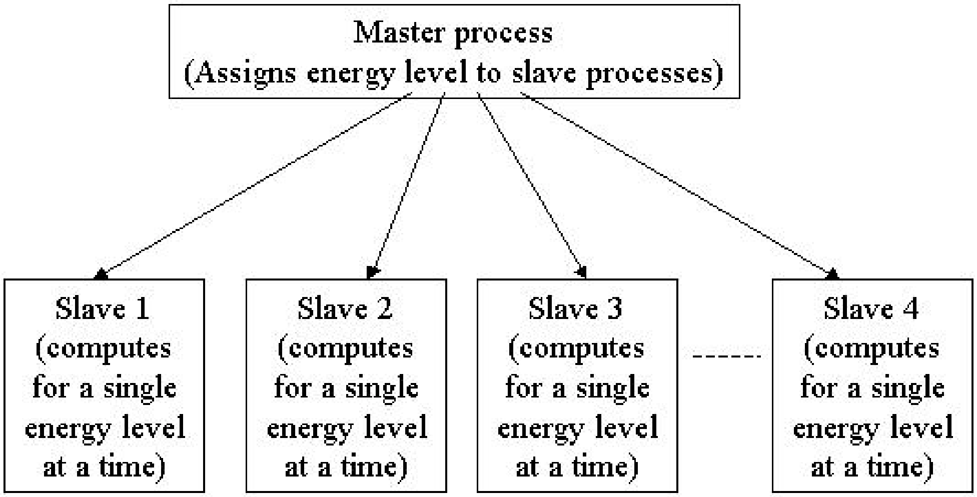

Figure 3.10. The master-slave structure of `transport`.

The master process is a coordinator that assigns energy levels to slaves and does not perform any other work. Not all energy levels involve an equal amount of computation. This could potentially cause load imbalance among the slave processes if the assignment of energy levels to the slaves is done statically. The master-slave structure of `transport` allows for load balancing in that assignment of an energy level to a slave occurs only after the slave has finished computing the results for the previous energy level assigned to it. Load balancing in this manner was found to be very effective for typical problem instances [2].

3.5.2 Transport Using LAPACK

At present, there exist two versions of code of the `transport` application. One version uses the sequential LAPACK solver libraries to solve the system of linear equations and the other uses the parallel solver library ScaLAPACK.

The LAPACK version of `transport` employs a single slave process to compute one energy level at a time. Although the loop over different energy levels is parallelized using different slaves, the computations corresponding to a single energy level are sequential.

The application needs a large amount of memory. Memory requirements are determined by six N_{mat} x N_{mat} double complex arrays, where $N_{mat}=(2N_x + 1) * (2N_y + 1) * (2N_z + 1)$. Also, these arrays are dense in nature and need to be stored completely (unlike sparse matrices where efficient representations reduce memory requirements by storing only non zero elements). The following table provides the memory requirements of each slave process in order to be able to run `transport`, for typical values of N_x, N_y, N_z.

Nx	Ny	Nz	Nmat	Mest (Gb)
4	4	8	1377	0.17
5	5	10	2541	0.58
6	6	12	4225	1.60
7	7	14	6525	3.81
8	8	16	9537	8.13
9	9	18	13357	15.95
10	10	20	18081	29.23

Table 3.1. Approximate memory requirements of `transport`.

3.5.3 Transport Using ScaLAPACK

`Transport` when using the ScaLAPACK libraries to perform its linear algebra operations has a slightly different structure. Each slave described above now corresponds to a ‘slave team’. Thus there are multiple processes that coordinate in the factorization and solution of a linear system of equations. Since the system of linear equations gets distributed over a team of processes, this structure facilitates solving larger systems of equations and hence larger problems. As Table 1 shows, the estimated memory required to compute a single energy level grows at a rate of $O((N_x\ N_y\ N_z)^2)$. Hence, using a sequential solver allows for experiments only with small problem sizes, e.g., using 1 GB RAM the maximum problem size that can be solved is $(N_x, N_y, N_z) = (5,5,10)$.

3.5.4 Transport with Babel

Two versions of `transport` already exist which employ LAPACK and ScaLAPACK. Pseudo code for `transport` using the PLAPACK library is also written. This gives a deeper insight into the data distribution policies of different solvers and how that affects the design of the algorithm selection system. A third version of `transport` has been developed that makes use of the runtime algorithm selection system. LAPACK and ScaLAPACK libraries are encapsulated in the Babel objects described in Section 3.4. The choice of solvers is made based on the size of the slave team involved in computing an energy level. If the team-size is 1, LAPACK is chosen and if it is greater than 1, ScaLAPACK is chosen. Although this choice of solvers is simplistic, it is sufficient to demonstrate that the system can choose different solvers under different conditions. More sophisticated ways of using recommender systems to choose an algorithm and its parameters are discussed in Chapter 4, by means of results obtained for different runs of various algorithms and problems.

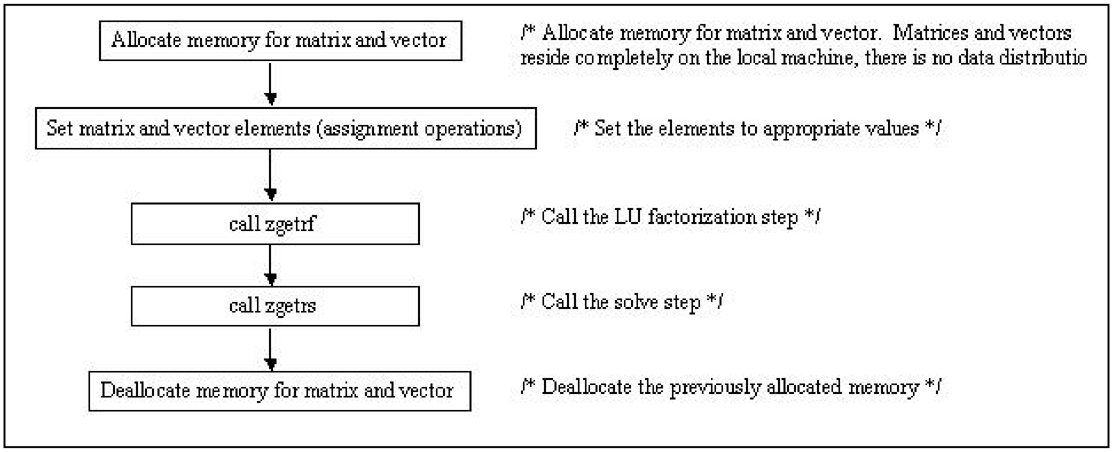
Allocate memory for matrix and vector
/* Allocate memory for matrix and vector. Matrices and vectors reside completely on the local machine, there is no data distributio
Set matrix and vector elements (assignment operations)
/* Set the elements to appropriate values */
call zgetrf
/* Call the LU factorization step */
call zgetrs
/* Call the solve step */
Deallocate memory for matrix and vector
/* Deallocate the previously allocated memory */

Figure 3.11. Pseudo code representing the execution flow of `transport` using LAPACK.

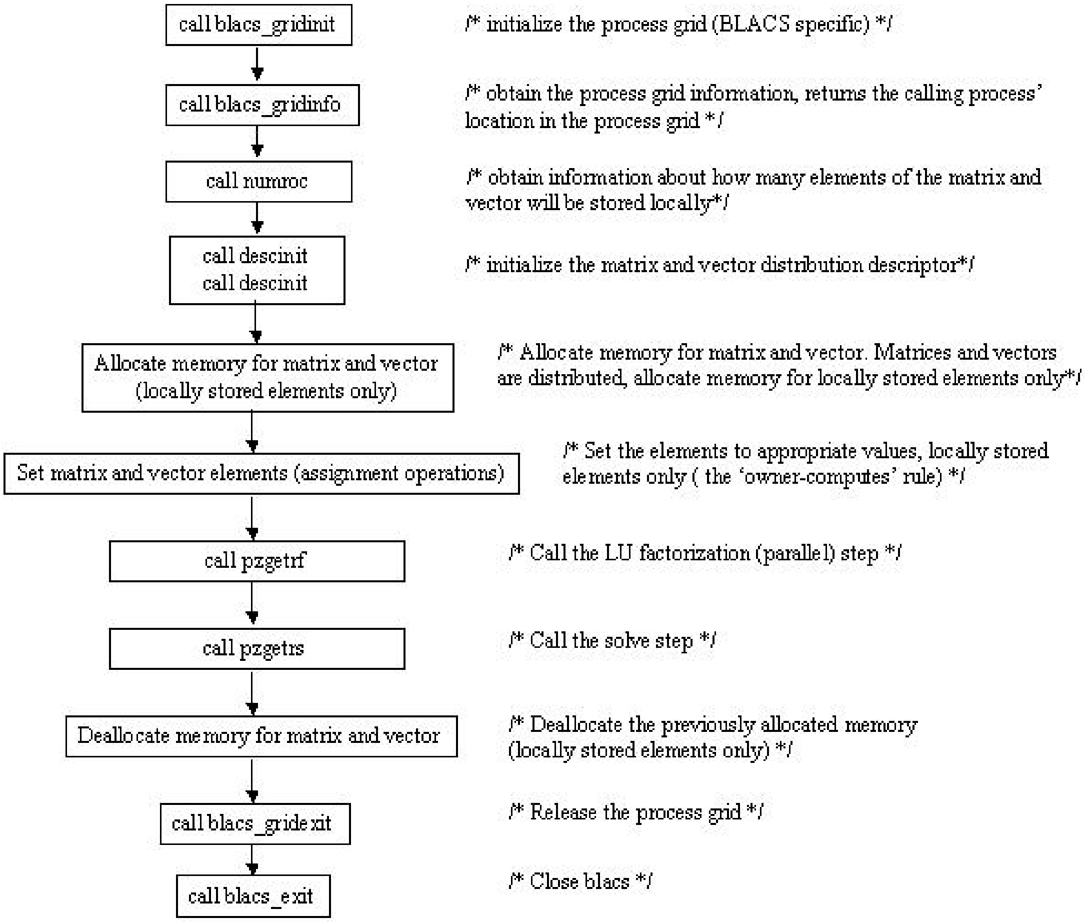
call blacs_gridinit
/* initialize the process grid (BLACS specific) */
call blacs_gridinfo
/* obtain the process grid information, returns the calling process' location in the process grid */
call numroc
/* obtain information about how many elements of the matrix and vector will be stored locally*/
call descinit
call descinit
/* initialize the matrix and vector distribution descriptor*/
Allocate memory for matrix and vector (locally stored elements only)
/* Allocate memory for matrix and vector. Matrices and vectors are distributed, allocate memory for locally stored elements only*/
Set matrix and vector elements (assignment operations)
/* Set the elements to appropriate values, locally stored elements only (the 'owner-computes' rule) */
call pzgetrf
/* Call the LU factorization (parallel) step */
call pzgetrs
/* Call the solve step */
Deallocate memory for matrix and vector
/* Deallocate the previously allocated memory (locally stored elements only) */
call blacs_gridexit
/* Release the process grid */
call blacs_exit
/* Close blacs */

Figure 3.12. Pseudo code representing the execution flow of transport using ScaLAPACK.

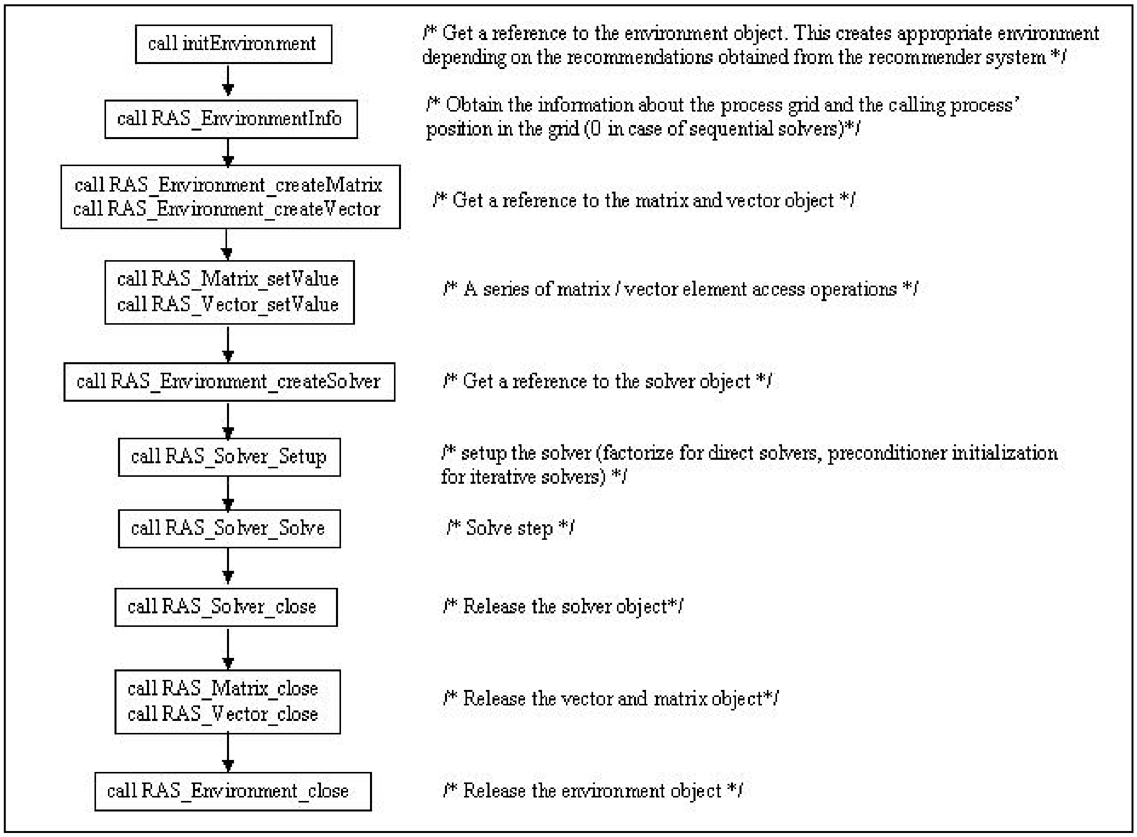
call initEnvironment
/* Get a reference to the environment object. This creates appropriate environment depending on the recommendations obtained from the recommender system */
call RAS_EnvironmentInfo
/* Obtain the information about the process grid and the calling process' position in the grid (0 in case of sequential solvers)*/
call RAS_Environment_createMatrix
call RAS_Environment_createVector
/* Get a reference to the matrix and vector object */
call RAS_Matrix_setValue
call RAS_Vector_setValue
/* A series of matrix / vector element access operations */
call RAS_Environment_createSolver
/* Get a reference to the solver object */
call RAS_Solver_Setup
/* setup the solver (factorize for direct solvers, preconditioner initialization for iterative solvers) */
call RAS_Solver_Solve
/* Solve step */
call RAS_Solver_close
/* Release the solver object*/
call RAS_Matrix_close
call RAS_Vector_close
/* Release the vector and matrix object*/
call RAS_Environment_close
/* Release the environment object */

Figure 3.13. Pseudo code representing the execution flow of `transport` using Babel based runtime algorithm selection system.

Figures 3.11, 3.12 and 3.13 show the pseudo code of the LAPACK, ScaLAPACK and Babel versions of `transport`, respectively. Note that multiple operations depicted in the same step can be executed in any order (for e.g., *setValue* function calls for matrix and vector objects). The Babel version hides the details of how the process grid is formed, which library functions need to be invoked to initialize the process grid, how much memory is allocated for the matrix and vector objects, the location of these elements, what type of solver is used, etc.

When a slave team starts up, it gets a reference to the *Environment* object. The Babel reference for the *Environment* object obtained by each process is local to the resource on which the process runs. The underlying communication libraries of the linear algebra packages take care of process synchronization wherever necessary. Thus, a distributed image of the object references is perceived even though Babel objects themselves are not distributed. For example, if ScaLAPACK is used as the solver package, the *Environment* object uses Basic Linear Algebra Communications Subprograms (BLACS) [46] that synchronize communication between the processes.

`Transport` is written in Fortran 90. The Babel objects for matrices, vectors, solvers, etc. are implemented in C. This was done so as to be able to evaluate the worst-case overhead when the application is written in a language different than that used to write the library. Another reason for choosing C over Fortran as the implementation language is to reduce memory overheads. When matrices and vectors are encapsulated in objects, the matrix or vector elements can be accessed only through the get and set functions of the respective class. Babel requires arrays to be passed in SIDL arrays format. When a function call like matrix-multiply is made, it internally calls functions like ‘zgemm’ (in case of LAPACK) or ‘pzgemm’ (in case of ScaLAPACK). If arrays are to be fed in the required format to these functions, there will be a need to copy the array elements from SIDL arrays structure to a general array format. This means that for every matrix required in an operation, we would need twice as much memory - one for the elements stored in SIDL array format and one required in the form needed by the package being used. However if these objects are implemented in C, elements stored in SIDL array structure can be accessed directly through memory pointers as opposed to accessing elements through the SIDL array API functions.

Chapter 4. Experiments and Evaluation

Chapter 3 discusses the runtime selection of linear algebra solvers in scientific applications. Illustrations of how the middleware for runtime selection of algorithms can be used are made with the help of Physics application `transport`. In this chapter, we first evaluate the overhead associated with the use of the runtime algorithm selection middleware. We also discuss how the overhead can be minimized. Then we compare the performance of `transport` using LAPACK and ScaLAPACK solvers. This illustrates and quantifies the importance of being able to choose between sequential solvers and parallel ones at runtime depending to factors like problem size and memory available. We then illustrate how different algorithms affect the performance of applications. Examples of choosing the right values of algorithmic parameters are also presented. The use of recommender systems to choose the right algorithm and the associated parameters is discussed and evaluated.

4.1 Evaluation of Runtime Algorithm Selection in Transport

This section seeks to answer the following questions:

- What is the overhead in introducing the middleware that permits runtime selection of algorithms?
- What is the performance obtained for varying problem sizes? Can sequential solvers prove to be useful even when a multitude of parallel solvers can be used for large problem sizes?
- What is the effect of varying algorithmic parameters on the performance of an application?

4.1.1 Experimental Setup

The experiments described in this section were conducted on a 200-node Linux cluster. Each of these nodes is equipped with a 1GHz AMD Athlon processor and 1GB RAM. The nodes have a 2.56 Gbps Myrinet interconnect. MPICH-GM is used as the message-passing library. All experiments were conducted on unloaded machines.

4.1.2 Overhead of Introducing Middleware

The middleware API discussed in Chapter 3 when used for runtime algorithm selection, introduces overheads due to extra function calls. In order that the runtime algorithm selection system be beneficial, the overhead involved should not offset the performance gains obtained by choosing an algorithm that is well suited for the conditions.

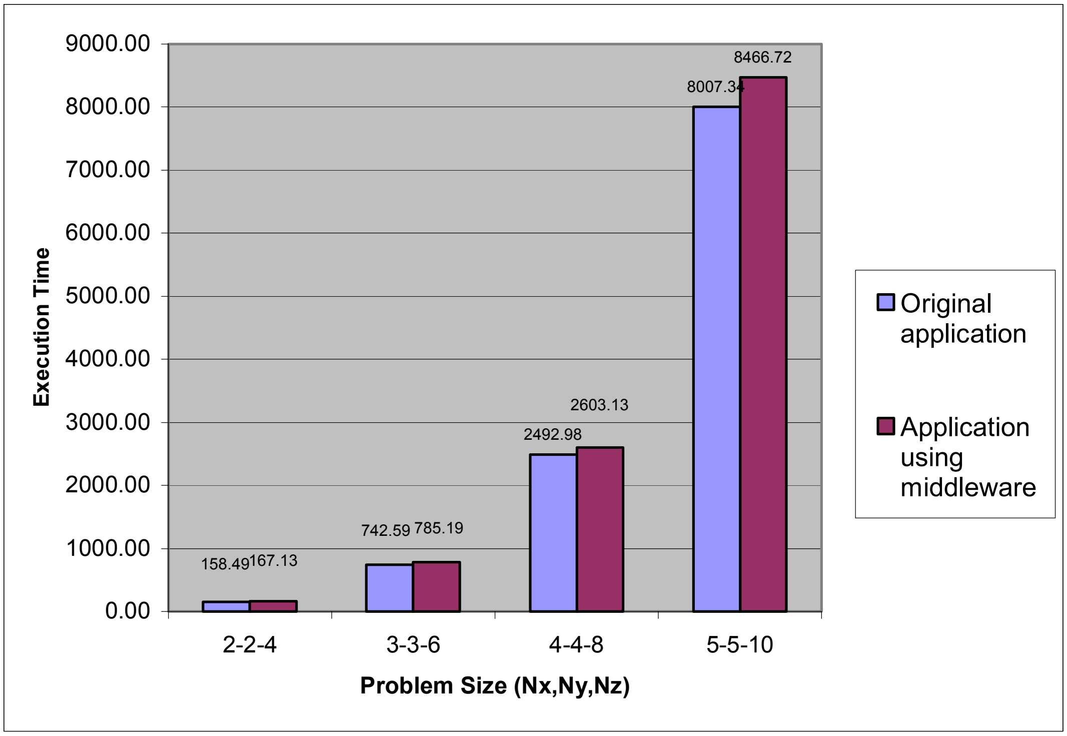

Figure 4.1 Evaluation of runtime algorithm selection overhead in the application `Transport`.

Figure 4.1 shows the overhead incurred in using runtime algorithm selection middleware `transport`. Four different problem sizes defined by (N_x, N_y, N_z) are considered. The slave team consists of two processes for each run. The original application uses ScaLAPACK to solve for 32 energy levels. In order to measure the overhead, it is necessary to keep constant all the other factors affecting the application performance. Hence the same number of processes should be used in a slave team, and the same algorithm has to be chosen by the algorithm selection system. For our empirical evaluation the choice of algorithm was based on team size. If the team size was greater than one, the algorithm selection system chose ScaLAPACK. We used a team size of two (as in the case of the original application). Hence the difference in execution times of the two runs measured the overhead of the dynamic selection of algorithm. The execution time observed with the code using the Babel framework is comparable to that of the original application. The overhead is within 6% of the time required by the original code.

As seen in Figure 4.1, the overhead scales with problem size. Recall from Section that matrix and vector elements are accessed using the *getValue* and *setValue* functions. As problem size increases, the dimensions of matrix and vector objects increase. As more number of elements are accessed using the *getValue* and *setValue* function calls, function call overhead increases. However, the overhead does not increase with number of processors or slave team-size. So, if the problem size per processor is constant (i.e., team size grows with problem size) then the relative overhead will remain roughly constant.

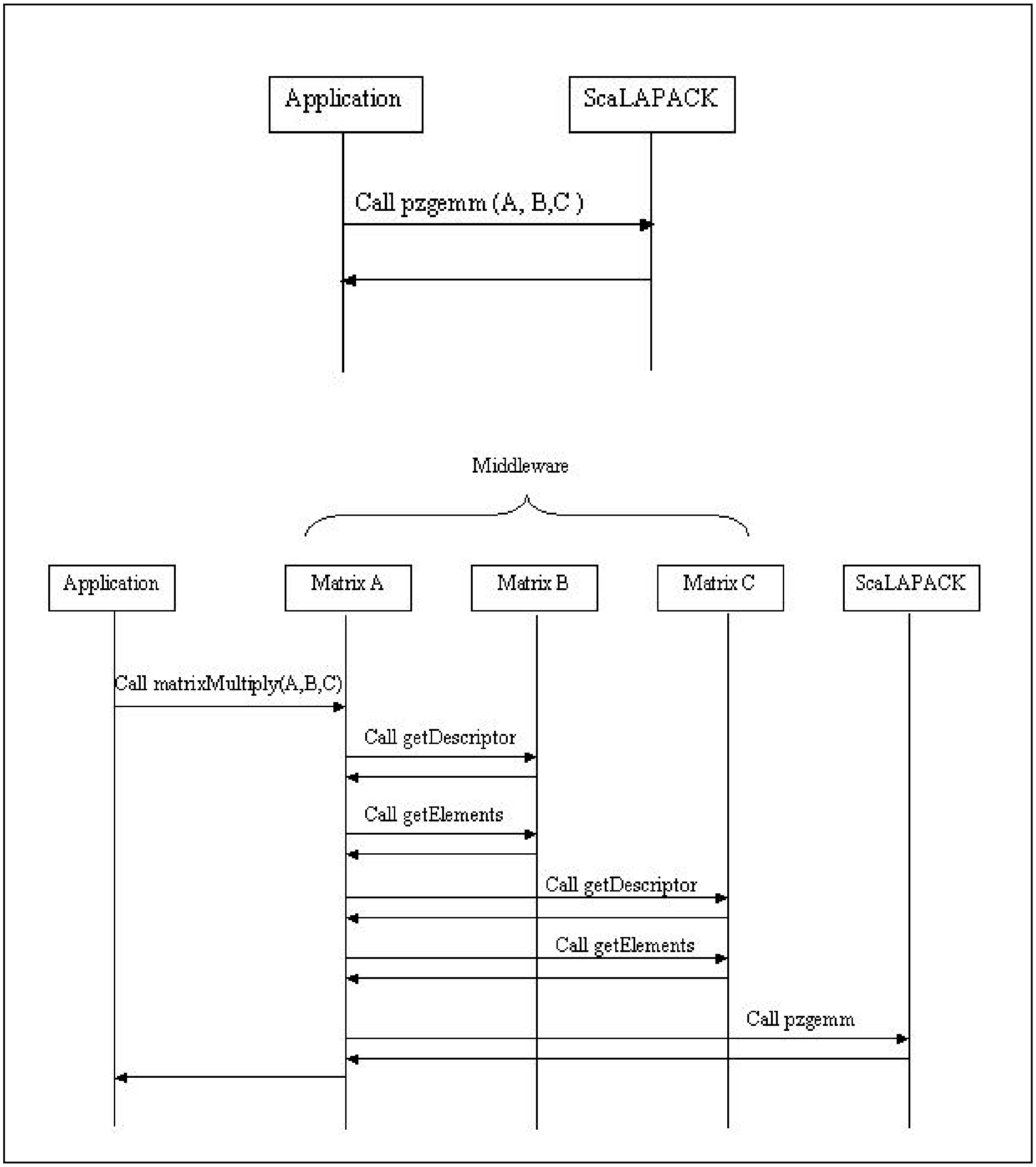

Figure 4.2 Function calling sequence for application using ScaLAPACK in a hardwired manner (top) and for application using runtime algorithm selection middleware (bottom).

There are two sources of overhead in using the middleware. First source is the indirection in using a solver. This is because the middleware encapsulates the solver functions in the interfaces exposed using the SIDL file. When a solver function is to be used, the application invokes the function of the middleware. The middleware then calls the appropriate solver specific function and returns the result. This indirection causes function call overheads. Second source of overhead is the encapsulation of matrices and vectors as objects. The data elements of these matrix and vector objects are not directly visible to the calling function. For example, when the matrix-multiply method is called on matrix A passing as arguments matrices B and C (to perform the operation C = αAB + βC), data elements of matrices B and C are not visible to matrix A. The data can only be accessed via get and set methods of the matrix (or vector) object. The get and set methods again cause function call overheads. Figure 4.2 depicts the function call sequence for a matrix-matrix-multiply operation in both the original application code and in the new version that uses the algorithm selection system. Note that in the new version, the application only calls the function exposed by the matrix interface (matrix multiply in the above example). The solver specific function (e.g., pzgemm) that a particular implementation (e.g., ScaLAPACK in Figure 4.2) calls is not visible to the application. Also note that function calls like *getDescriptor* shown in the figure are specific to a particular solver's implementation (ScaLAPACK in this case). There may not be such function calls in every solver implementation (e.g., LAPACK does not have any matrix distribution descriptor). For such solvers, the overhead incurred is reduced.

4.1.3 Performance comparison of LAPACK and ScaLAPACK

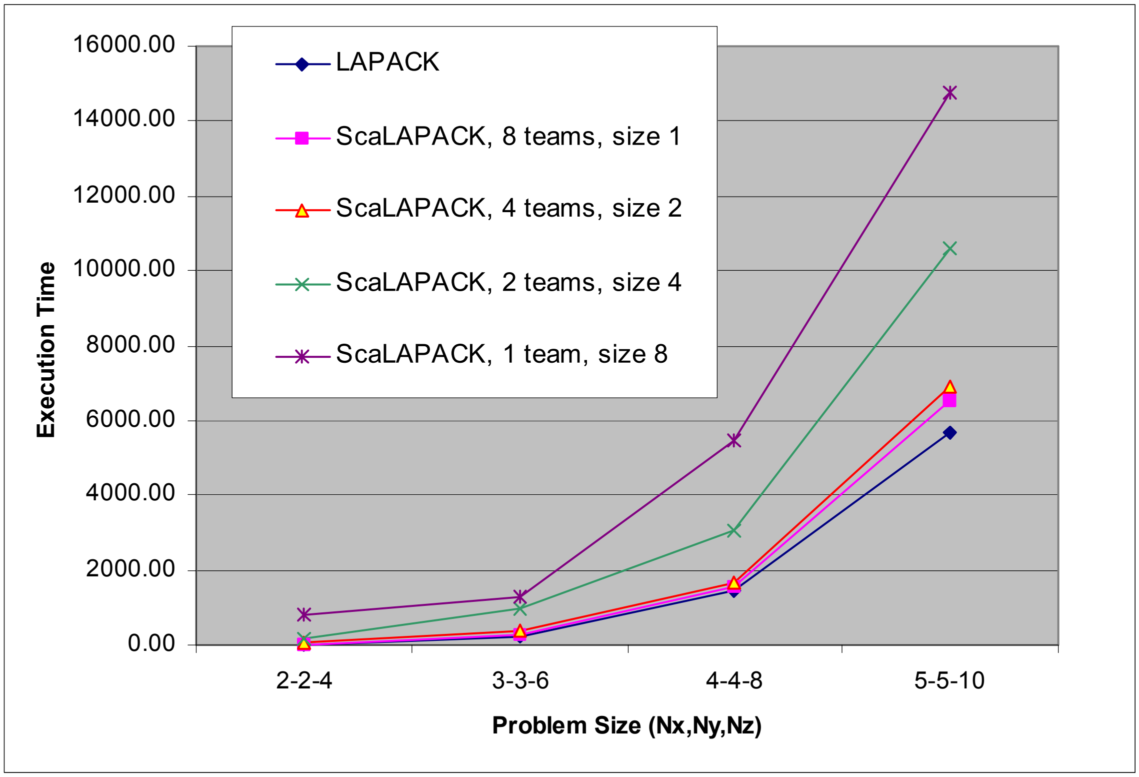

Figure 4.3 Performance comparisons of LAPACK and ScaLAPACK for varying problem sizes in `Transport`.

In this experiment, we compare the performance of a sequential solver and a parallel solver. The application transport was run for varying problem sizes defined by (N_x, N_y, N_z). Eight processes were employed as slaves. In case of LAPACK, these eight slaves form eight slave teams of one process each. With ScaLAPACK, the eight slave processes form different number of teams with different sizes. The number of teams and team sizes combinations considered here are (1, 8), (2, 4), (4, 2), (8,1). As seen in Figure 4.3 LAPACK shows significant performance gains as compared to ScaLAPACK for all problem sizes.

Two factors contribute to the better performance displayed by LAPACK. First is the communication costs incurred in parallel solvers. This means that the performance gain achieved using LAPACK will increase if the available network bandwidth between the machines on which the application executes is less. Second is the unequal computations involved in different energy levels and the manner in which energy levels are assigned to slave teams in transport. Recall from Section 3.5.1 that the assignment of an energy level to a slave takes place only after it has finished computing the previously assigned energy level. This means that while a single team of eight slave processes is computing an energy level in ScaLAPACK, multiple teams each consisting of a single process in LAPACK may compute more number of energy levels independently. To observe the effect of the number of teams and the team size on the application's performance, different numbers of teams with varying team-size were considered. It is observed that the more the number of teams, the better the performance. This is because of the load balancing in transport explained earlier. Also, smaller teams perform better than larger teams. This is because of the communication costs of parallel solvers. The communication cost is less with fewer number of processes participating in a computational problem.

The problem with using LAPACK for large problem sizes is that the memory requirements are high and may exceed the amount of RAM available on the machine. For example, LAPACK cannot be used with problem sizes greater than (N_x, N_y, N_z) = (5, 5, 10) on machines with 1GB RAM.

The better performance of LAPACK as compared to ScaLAPACK highlights the importance of using a sequential solver rather than a parallel solver if a particular problem instance can be solved with the amount of RAM available on the resource. This is especially true in the case of heterogeneous environments where uneven network bandwidth and delays degrade the performance of a parallel solver even more.

4.1.4 Effect of Number of Teams on Transport Performance

As described in Section 3.5, a slave team can consist of a number of processes. The number of slave teams and the size of a slave team have an effect on the running time of transport. In addition to these factors, unequal work involved in different energy level computations also affects the execution time. Figure 4.4 depicts the running time of transport for problem size (N_x,N_y,N_z) = (5,5,10)solving 32 energy levels, when 32 slave processes are used. Since LAPACK is a sequential solver, all 32 processes form different teams of size one each. In the case of ScaLAPACK, 16 teams each of size two were used. One would expect that the sequential solver (LAPACK in this case) would win over the parallel one (ScaLAPACK) because of the communication costs involved in parallel solvers. However, the load imbalance caused by different amount of work involved in different energy levels causes some of the slave processes in the sequential solver case to remain idle while other slave processes complete their computations. This is balanced out in the case of 16 teams used with ScaLAPACK. The behavior varies for different problem sizes and different team sizes. This suggests that the number of teams and the team size cannot be determined in advance.

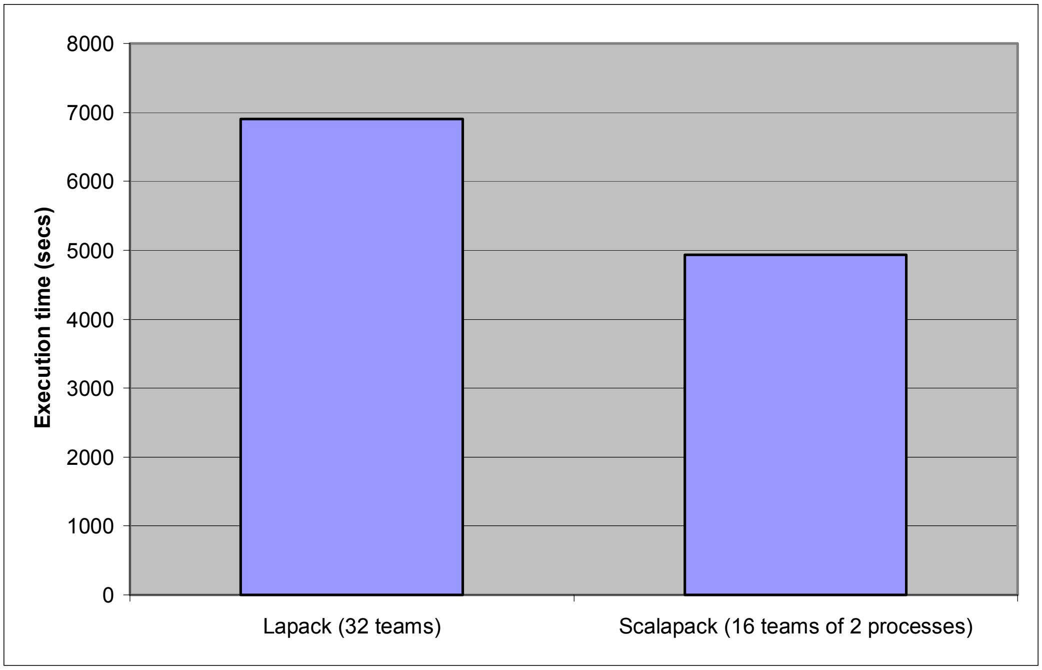

Figure 4.4 Performance comparison of transport with different team-sizes

4.2 Effect of algorithmic parameters on performance

The experiments in this section attempt to investigate the effect of varying algorithmic parameters on the performance of an algorithm.

4.2.1 Experimental Setup

A Linux cluster of 200 nodes, each equipped with a 1GHz AMD Athlon processor and 1GB RAM was used for these experiments. The nodes have a 2.56 Gbps Myrinet interconnect. MPICH-GM was used as the message-passing library. All experiments were conducted on unloaded machines. A parallel version of a subset of the ELLPACK system [47], a system for numerically solving elliptic partial differential equations (PDEs) was used to solve PDE problems.

4.2.2 Performance of GMRES with different values of restart parameter

Figure 4.4 depicts the relative performance penalty of using a default GMRES restart value (the size of the Krylov subspace) of 20 as opposed to the optimal value determined by the recommender system. A test PDE problem 15 from [47] was run for different problem sizes and number of processors. The number of grid lines used for discretization determines the problem size (this is the X-axis of the graph in Figure 4.4).

The value of the restart parameter 'k' can have a significant effect on the performance of the algorithm; and more importantly, the optimal choice for k depends in subtle ways on the problem, the amount of memory available, the CPU speed, and the interconnection network. In practice, most users simply set k to a constant value and proceed. However, Figure 4.5 shows that even for a modestly large problem, a better choice for k can improve running time by 40% or more. In cases like this, the recommender system can derive simple rules that recommend good algorithm (parameter) choices, as a function of machine and problem parameters such as the number of processors and the size of the linear system. However, the RS-derived rules (not shown here) also suggest that when per-processor memory is limited, but a large number of processors is to be used, an alternative algorithm should be selected (since the penalty for using a small value of k will be so high).

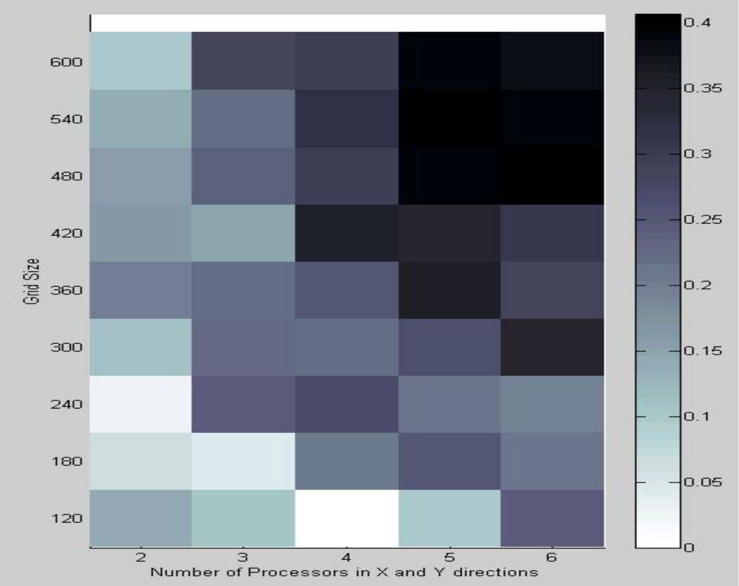

Figure 4.5. Relative performance penalty of using GMRES restart value 20 instead of the optimal value (as determined by the recommender systems), as a function of problem size and number of processors. Matrix generated from finite difference discretization of test PDE problem 15 from [47].

Chapter 5. Conclusions and Future Work

5.1 Conclusions and Contributions

In order for a runtime algorithm selection system to be useful, it is necessary for a unified interface to be provided to the user. The user should not be expected to have knowledge of each and every algorithm that could be used. The primary contribution of this thesis is in identifying and resolving the conflicts arising while offering a unified interface to diverse linear algebra solvers.

The Babel based framework for runtime algorithm selection (discussed in Chapter 3) has been implemented. The linear algebra solvers offered by LAPACK and ScaLAPACK packages have been incorporated. The application `transport` was modified to make use of this framework. In a simplistic case, the framework permits choosing a linear algebra solver based on the number of processes participating in the computations. LAPACK is chosen if the number of processes is one; ScaLAPACK is chosen if the number of processes is greater than one. In this way, `transport` developers can maintain only one version of this code, and have it run well anywhere on the Grid.

The advantages of transparency and optimal choice of algorithm based on runtime conditions comes at the cost of overhead involved in introducing the layer that performs this abstraction. The overhead of introducing this middleware has been evaluated. The performance achieved using the unified interface is comparable to that of the original application that does not use this interface.

One of the important challenges identified while designing a common interface for linear algebra solvers, is the varying degree to which solvers permit the knowledge of data distribution. If all solvers that can be accessed using a common interface hide their data distribution completely, then a mechanism has to be developed for processes to synchronize their data updates. However, the degree to which the performance of the underlying solver would degrade because of such a synchronization mechanism is not known. On the other hand, the common interface cannot permit the application to know the data distribution in all cases. This is due to the inherent lack of visibility of data distribution in some solvers. Hence, an application using the runtime algorithm selection system should first verify if the data distribution in the solver selected is visible or not. If the data distribution is visible, the processes of the SPMD application can choose different control paths based on data ownership. If the data distribution were not visible, processes would work on different blocks of data irrespective of ownership. Thus, the burden of parallelizing the application appropriately was left to the user.

Another point that becomes clear when developing a common interface for linear solvers is that if an application wants to benefit from the right choice of solvers (including parallel solvers) in the true sense, the application writer has to 'think parallel' in the first place. This means that computational steps have to be written considering that a set of processes will perform them instead of a single process. In other words, while details of linear algebra algorithms and data structures can be abstracted away, the user must still be aware that parallel solver may be used.

Finally, the recommender systems approach to algorithm selection was evaluated. This approach is good for applications that produce a lot of performance data, and whose computational and communication requirements are relatively predictable. The recommendations are only as good as the database used for making recommendations. However, the performance data gathered from one problem can be used to select an algorithm for similar problems.

5.3 Future Work

Incorporating other Linear Algebra Solvers: The runtime selection of linear algebra solvers is successful in the case of `transport`. At present, ScaLAPACK and LAPACK have been incorporated in the framework. Other solvers need to be incorporated to take advantage of the differences in performance shown by different solvers depending on problem and machine characteristics.

Automated Downloading of Libraries: In a grid environment, it is unreasonable to expect that all the necessary software and libraries required to run an application can be found on the resources that are selected to run the program. A system to dynamically load libraries to the machines on which the program executes is required. When the runtime algorithm selection engine attempts to instantiate a particular algorithm, it dynamically loads the library corresponding to the algorithm. This call to the loader could be intercepted. The Runtime Algorithm Selection Engine should check whether the necessary library is present on the local resource. If so, the library can be loaded and the call can be completed. However, if the library is not present, a server should be contacted to download the necessary library. NetBuild [50] is one such system that allows for dynamic downloading of libraries and linking them to the application. We are investigating how such systems can work in tandem with the runtime algorithm selection system. This would take the burden of ensuring the availability of libraries off the user.

Support for process migration: At present, the runtime algorithm selection assumes that an application process will not migrate. This restriction should be relaxed. If processes migrate from one resource to another, the algorithm selected initially for the application's execution may not perform well in the new computing environment. This means that a different algorithm suitable for the new resource type should be chosen. This necessitates the support for check pointing and conversion from one algorithm's needs to another (for e.g., data structures). Cooperation between the runtime algorithm selection and process migration system is an interesting area of research.

Interaction with Other Grid Middleware: The runtime algorithm selection system selects algorithms based on resource and problem characteristics. It ensures that given a set of resources on which the application runs, the solver that would perform the best on those resources will be chosen. However, the choice of the solver and hence the performance would vary if the application runs on a different set of resources. This means that the performance data that is used for selecting an algorithm can also be used to make a choice of the resources on which to run the application. If schedulers for the Grid can use this information about an application's performance on various resources, it would lead to performance improvement for the application [48, 49].

A promising interaction scenario between the runtime algorithm selection system and a scheduler is as follows. The scheduler queries the runtime algorithm selection system to obtain information about what type of resources are best suited for the particular application. Using this information the search space of the scheduler can be focused on those resources on which the application to be scheduled performs well. From prior runs of an application, the runtime algorithm selection system obtains performance information. Table 5.1 shows an example of how such performance data would look like.

	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

Table 5.1. Records exchanged between Runtime Algorithm Selection System and a Grid Scheduler.

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. (Here, we assume 'best performance' means lowest execution 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 characteristics similar to class 1 when assigning processes to processors.

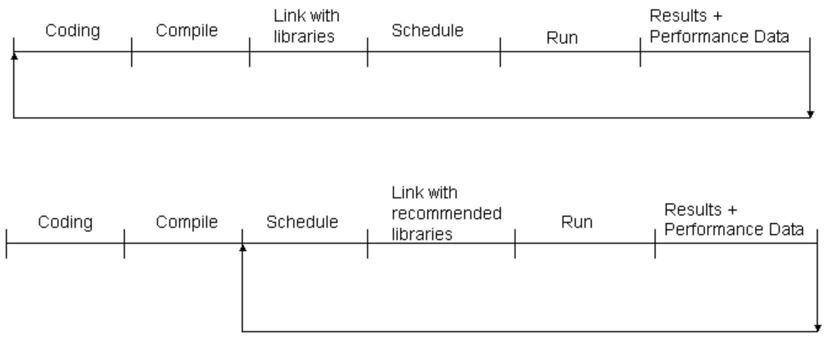
Coding
Compile
Link with libraries
Schedule
Run
Results + Performance Data
Coding
Compile
Schedule
Link with recommended libraries
Run
Results + Performance Data

Figure 5.1. Performance improvement cycle for an application: without algorithm selection (top), with algorithm recommendation (bottom).

This interaction between the scheduler and the algorithm selection system changes the development cycle for applications. In Figure 5.1 (top), we see the typical existing development cycle for high-end computational programs. In order to improve performance of a particular code, the code is run, performance data is obtained. This performance data is then analyzed, and the code is rewritten to improve performance, by using a different algorithm, by changing parameter values or by choosing a different set of machines on which to run the application. When the burden of trying different combinations of algorithms, parameters and machines is taken off the application scientist, linking with libraries happens at runtime. Thus, Figure 5.1 (bottom) depicts the new sequence of operations involved in a performance improvement cycle of the application.

Bibliography

[1] Y. Saad. **Iterative Methods for Sparse Linear Systems**, Society for Industrial and Applied Mathematics, 2nd edition, April 30, 2003.

[2] C. J. Ribbens, P. Bora, J. Hauck, S. Prabhakar, C. Taylor, M. Di Ventra. **From Cluster to Grid: A Case Study in Scaling-Up a Molecular Electronics Simulation Code**, Proceedings of the High Performance Computing Symposium, I. Banicescu (ed.), Society for Modeling and Simulation International, San Diego, pp. 54-62, 2003.

[3] R. Whaley, J. Dongarra. **Automatically Tuned Linear Algebra Software (ATLAS)**, SC '89 Proceedings (Electronic Publication), IEEE Publication.

[4] W. Gu, J. Vetter, K. Schwann. **An annotated Bibliography of Interactive Program Steering**, SIGPLAN Notices 29 (1994) 140-8 and Technical Report GIT-CC-94-15, file://ftp.cc.gatech.edu/pub/tech_reports/1994/GIT-CC-94-15.ps.Z

[5] S. G. Parker, D. M. Weinstein, C. R. Johnson. **The SCIRun Computational Steering Software System**, In Modern Software Tools in Scientific Computing, Edited by E. Arge and A.M. Bruaset and H.P. Langtangen, Birkhauser Press, Boston, pp. 1-40, 1997

[6] G. A. Geist, J. A. Kohl, P. M. Papadopoulos. **CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications**, International Journal of High Performance Computing Applications, Vol. 11, pp. 224–236, 1997.

[7] J. Lewis. **Cruising (approximately) at 41,000 feet - Iterative methods at Boeing**, Talk presented at the Seventh SIAM Conference on Applied Linear Algebra, 2000. www.siam.org/meetings/la00.

[8] M. G. Lagoudakis, M. L. Littman, **Algorithm Selection using Reinforcement Learning,** Proceedings of the Sixteenth International Conference on Machine Learning, AAAI Press, 2000.

[9] K. Suzaki, T. Kurita, H. Tanuma, S. Hirano**. Adaptive Algorithm Selection Method (AASM) for Dynamic Software Tuning,** Proceedings of 17th Int. Comp. Software and Applications.

[10] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, C. Romine**, 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.

[11] B. A. Huberman, R. M. Lukose, T. Hogg. **An Economics Approach to Hard Computational Problems**, Journal of Science, Vol. 275, pp. 51-54, January 3, 1997.

[12] C. P. Gomes, B. Selman. **Practical Aspects of Algorithm Portfolio Design**, Proceedings of Third ILOG International Users Meeting, 1997.

[13] J. Borrett, E. P. K. Tsang, N. R. Walsh**, Adaptive Constraint Satisfaction: The Quickest First Principle**, European Conference on Artificial Intelligence, 1996.

[14] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, E. Akman, J. Balasubramanian, F. Breg, S. Diwan, M. Govindaraju, **The Linear System Analyzer**, Technical Report TR-511, Computer Science Dept, Indiana University, 1998.

[15] A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, S. Vadhiyar, **Numerical Libraries and The Grid**, International Journal of High performance Applications and Supercomputing, Vol. 15, no 4, pp. 359-374, Winter, 2001.

[16] C. Þãpuº, I. Chung, J. K. Hollingsworth**, Active Harmony: Towards Automated Performance Tuning**, SC'02, Nov 2002.

[17] J. Dongarra, V. Eijkhout, **Self-adapting Numerical Software for Next Generation Applications**, Technical report, Innovative Computing Laboratory, University of Tennessee, August 2002. http://icl.cs.utk.edu/iclprojects/pages/sans.html

[18] R. Ko, M. W. Mutka, **FRAME for Achieving Performance Portability within Heterogeneous Environments**, Proceedings of the 9th IEEE Conference on Engineering of Computer-Based Systems (ECBS 2002), April 2002.

[19] D. C. Arnold, J. Dongarra, **The NetSolve Environment: Progressing Towards the Seamless Grid**, 2000 International Conference on Parallel Processing (ICPP-2000), Toronto Canada, August 21-24, 2000.

[20] L. Lobjois, M. Lematre. **Branch and Bound Algorithm Selection by Performance Prediction**, Proceedings of the Fifteenth National Conference on Artificial Intelligence, Menlo Park: AAAI Press, pp. 353—358, 1998.

[21] **Equation Solver Interface (ESI) Standards**, http://z.ca.sandia.gov/esi/

[22] **Lapack Working Notes**, www.netlib.org/lapack/

[23] **Basic Linear Algebra Subprograms (BLAS)**, www.netlib.org/blas

[24] **ScaLAPACK user's guide**, www.netlib.org/scalapack

[25] P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, Y. J. Wu. **PLAPACK: Parallel Linear Algebra Package**, Proceedings of the SIAM Parallel Processing Conference, 1997.

[26] Y. Saad, M. H. Schultz**. A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems**, SIAM Journal of Scientific Statistical Computing, Vol. 7, pp. 856–869, 1986.

[27] M. R. Hestenes, E. Stiefel. **Methods of Conjugate Gradients for Solving Linear Systems**, J. Res. National Bureau of Standards, Vol. 49, pp. 409–435, 1952,.

[28] R. W. Freund, N. M. Nachtigal. **QMR: a Quasi-Minimal Residual Method for non-Hermitian Linear Systems**, Journal of Numer. Math., Vol. 60, pp. 315—339, 1991.

[29] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, J. Shalf. **The Cactus Framework and Toolkit: Design and Applications**, Vector and Parallel Processing - VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science.

[30] **Component Object Model specifications**, http://www.microsoft.com/com/

[31] **Common Object Request Broker Architecture (CORBA) Specifications**, http://www.omg.org/technology/documents/spec_catalog.htm

[32] T. Epperly, S. Kohn, G. Kumfert**. Component Technology For High-Performance Scientific Simulation Software**, Working Conference on "Software Architectures for Scientific Computing Applications", International Federation for Information Processing, Ottowa, Ontario, Canada, October 2-4, 2000

[33] S. Kohn, G. Kumfert, J. Painter, C. Ribbens**. Divorcing Language Dependencies From A Scientific Software Library**, 10th SIAM Conference on Parallel Processing, Portsmouth, VA, March 12-14, 2001

[34] N. Elliott, S. Kohn, B. Smolinski**. Language Interoperability for High Performance Parallel Scientific Components**, International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE '99), San Francisco, CA, September 29-October 2, 1999

[35] H. van der Vorst. **Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems**, SIAM J. Sci. Statist. Comput., Vol. 13, pp. 631–644, 1992.

[36] N. Ramakrishnan, C. J. Ribbens. **Mining and Visualizing Recommendation Spaces for Elliptic PDEs with Continuous Attributes**, ACM Trans. Math. Software, Vol. 26, pp. 254-273, 2000.

[37] N. Ramakrishnan, S. Varadarajan**. Novel Runtime Systems Support for Adaptive Compositional Modeling on the Grid**, Technical Report CS.CE/0301018, Computing Research Repository (CoRR), Virginia Tech, Dec 2002.

[38] E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, C. E. Houstis, **PYTHIA-II: A Knowledge/Database System for Managing Performance Data and Recommending Scientific Software**, ACM Transactions on Mathematical Software, Vol. 26, No. 2, pp. 227-253, June 2000.

[39] A. Joshi, S. Weerawarana, N. Ramakrishnan, E. Houstis, J. Rice, **Neurofuzzy Support for PSEs: A Step Toward the Automated Solution of PDEs**, Special Joint Issue of IEEE Computer & IEEE Computational Science and Engineering, Vol. 3, 1, pp. 44–56, 1996.

[40] S. Muggleton, L. D. Raedt, **Inductive Logic Programming: Theory and Methods**, Journal of Logic Programming, Vol. 19, No. 20, pp. 629–679, 1994.

[41] I. Bratko, S. Muggleton, **Applications of Inductive Logic Programming**, Communications of the ACM, Vol. 38, No. 11, pp. 65–70, 1995.

[42] E. N. Houstis, A. C. Catlin, N. Dhanjani, J. R. Rice, N. Ramakrishnan, V. S. Verykios**, MyPYTHIA: A Recommendation Portal for Scientific Software and Services**, Concurrency and Computation: Practice and Experience, Vol. 14, Nos. 13-14, pp. 1481-1505, Nov-Dec 2002.

[43] J. R. Quinlan, **Induction of Decision Trees**, Machine Learning, Vol. 1, No. 1, pp. 81–106, 1986.

[44] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. **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.

[45] B. Stroustrup, **The C++ Programming Language**, Addison-Wesley Pub. Co., 3rd edition, February 15, 2000.

[46] **Basic Linear Algebra Communications Subprograms**, www.netlib.org/blacs

[47] J. R. Rice, et al. **Solving Elliptic Problems Using ELLPACK**, Springer Verlag, 1984.

[48] P. Bora, C. Ribbens, S. Prabhakar, G. Swaminathan, M. Chinnusamy, A. Jeyakumar, B. Diaz-Acosta. **Issues in Runtime Scientific Algorithm Selection for Grid Environments**, Challenges of Large Applications in Distributed Environments, HPDC-12, June 2003. (To appear)

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

[50] K. Moore, J. Dongarra. **NetBuild: Transparent Cross-Platform Access to Computational Software Libraries**, submitted to Concurrency: Practice and Experience, July 2001.

[51] F. Burstall, J. Darlington, **A Transformation System for developing recursive programs**, JACM, Vol. 24, 1981

[52] J. Darlington, **An Experimental Program Transformation Synthesis System**, Artificial Intelligence, Vol. 16, pp. 1-46, 1981.

[53] Y. Futamura, **Partial Computation of Programs**, LNCS147, Springer-Verlag, 1983.

[54] P. Maes, D. Nardi, **Meta-level architectures and Reflection**, North-Holland, 1988.

[55] J.R. Rice, **The Algorithm Selection Problem**, Advances in Computers, Vol. 15, pp. 65 -118, 1976.

Vita

Prachi Champalal Bora

Prachi Bora was born in Ahmednagar, India on October 24th 1978. She did her schooling there and then moved to Pune for her Bachelors degree.

After securing a B.E (Bachelor of Engineering) degree in Computer Engineering from Pune University, she worked as a Software Engineer for a period of one year in Mahindra British Telecom Ltd., Pune, an SEI CMM level V company.

After this she joined Virginia Polytechnic Institute & State University in Fall 2001 for a Master's degree in Computer Science & Applications and graduated in June 2003. Her interests lie in Networking and Operating Systems.