Towards an Automatic and Application-Based Eigensolver Selection

Yeliang Zhang
Electrical and Computer Engineering Dept.
The University of Arizona
Tucson, AZ 85721
Email: zhang@ece.arizona.edu

Xiaoye S. Li
Osni Marques
Lawrence Berkeley National Lab
Computational Research Division
1 Cyclotron Road, MS 50F-1650
Berkeley, CA 94720-8139
Email: xsli@lbl.gov, oamarques@lbl.gov

Abstract

The computation of eigenvalues and eigenvectors is an important and often time-consuming phase in computer simulations. Recent efforts in the development of eigensolver libraries have given users good algorithms without the need for users to spend much time in programming. Yet, given the variety of numerical algorithms that are available to domain scientists, choosing the "best" algorithm suited for a particular application is a daunting task. As simulations become increasingly sophisticated and larger, it becomes infeasible for a user to try out every reasonable algorithm configuration in a timely fashion. Therefore, there is a need for an intelligent engine that can guide the user through the maze of various solvers with various configurations.

In this paper, we present a methodology and a software architecture aiming at determining the best solver based on the application type and the matrix properties. We combine a decision tree and an intelligent engine to select a solver and a preconditioner combination for the application submitted by the user. We also discuss how our system interface is implemented with third party numerical libraries. In the case study, we demonstrate the feasibility and usefulness of our system with a simplified linear solving system. Our experiments show that our proposed intelligent engine is quite adept in choosing a suitable algorithm for different applications.

I. Introduction and Motivation

The computation of eigenvalues and eigenvectors is an important and often time-consuming phase in computer simulations. Eigenvalues and eigenvectors are used in the study of nuclear reactor dynamics (stability of neutron fluxes [28]), in finite element dynamic analysis of structural models (e.g., seismic simulations of civil infrastructure [9], [10]), in the design of the next generation of particle accelerators [17], in the definition of a set of eigenfaces in biometric-based identification systems [27], in the solution of Schrödinger's equation in chemistry and physics [23], in the design of microelectromechanical systems (MEMS [11]), in the study of conformational changes of proteins [21], and in many other applications. Because of the need for higher level of simulation details and accuracy, the size and complexity of the computations grow as fast as the advancement of the computer hardware.

In order to cope with the increasing need for solving eigenvalue problems, various useful numerical algorithms that are suitable for solving large-scale eigenvalue problems have been developed. Some of these numerical libraries also have parallel implementations [7], [18], [20], [29]. With the growing availability of solvers, domain scientists are no longer facing the problem of lacking algorithms to use, but facing the problem of too many algorithms to choose from. (In this paper, "algorithm" is interchangeable with "solver"). However, little consideration is given to mechanisms that provide a robust and effective way to sift out a best solver for a particular application. For sequential or parallel algorithms, a suitable choice of solver may have an order of magnitude impact on an application compared to a bad one. We identify the following challenges in the selection of the "best" solver:

- Given the vast number of algorithmic and computer architectures, how to facilitate a naïve user to choose an algorithm "best" suited for a particular application. Different algorithms have different convergence behaviors, memory requirements, and trade-offs between accuracy and performance. It is difficult and tedious to manually track these metrics and select an algorithm based on these metrics. Ideally, this should be done automatically.
- There is no *unified software framework* targeted for high-end computers that facilitates swapping among different algorithm implementations. The *Eigentemplates* book [3] provides an excellent algorithmic guideline.

Yet, in order to fully appreciate that book a potential user must be equipped with sufficient knowledge of numerical analysis and programming skills, in particular on parallel computers. What is needed to bridge the gap between *Eigentemplates* and the end user is a software toolbox that contains efficient and scalable implementations of those algorithms. The libraries currently available are limited in scope, scattered at different places with different interfaces. Table I presents a brief survey of a variety of eigenvalue packages.

- Usually, there are a large number of parameters associated with each algorithm, which can be adjusted in order to make the algorithm perform more efficiently. For example, the electronic structure calculation codes often employ conjugate gradient minimization based eigensolvers that require inner-outer iterations, where the number of iterations is set by the user and usually requires information about the target problem, and therefore may greatly impact the computational performance. Even if there is a collection of libraries with a unified interface, as simulations become increasingly sophisticated and larger, it becomes infeasible for the potential user to try out every algorithm configuration in a timely fashion.

Therefore, there is a need for an *intelligent engine* that can guide the user through the maze of different solvers with various configurations and also a database that records all the historical data to provide the *intelligent engine* with information that can be used for a judicious decision .

Name	Method	Version	Date	Language	Parallel
ARPACK	Implicitly Restarted Arnoldi/Lanczos	2	1996	F77	MPI
BLZPACK	Block Lanczos, PO+SO	04/00	2000	F77	MPI
DVDSON	Davidson	-	1995	F77	-
GUPTRI	Generalized Upper Triangular Form	-	1999	F77	-
IETL	Power/RQI, Lanczos-Cullum	2.1	2003	C++	-
JDBSYM	Jacobi-Davidson (symmetric)	0.14	1999	C	-
LANCZOS	Lanczos (Cullum, Willoughby)	-	1992	F77	-
LANZ	Lanczos, PO	1.0	1991	F77	-
LASO	Lanczos	2	1983	F77	-
LOBPCG	Preconditioned Conjugate Gradient	4.10	2004	C	MPI
LOPSI	Subspace Iteration	1	1981	F77	-
MPB	Conjugate Gradient / Davidson	1.4.2	2003	C	-
NAPACK	Power Method	-	-	F77	-
PDACG	Deflation-accel. conjugate gradient	-	2000	F77	MPI
PLANSO	Lanczos, PO	.10	1997	F77	MPI
QMRPACK	Nonsymmetric Lanczos with lookahead	-	1996	F77	-
SLEPc	Power/RQI, Subspace, Arnoldi	2.2.1	2004	C/F77	MPI
SPAM	Subspace Projected Approx. Matrix	-	2001	F90	-
SRRIT	Subspace Iteration	1	1997	F77	-
SVDPACK	SVD via Lanczos, Ritzit & Trace Minim.	-	1992	C/F77	-
TRLAN	Lanczos, dynamic thick-restart	1.0	1999	F90	MPI
Underwood	Block Lanczos	-	1992	F77	-

TABLE I
A SKETCHY SURVEY OF EIGENVALUE LIBRARIES

The objective of this paper is to address some of the aforementioned challenges, and present our design and methodology towards an automatic application-based eigensolver selection toolbox, EIGADEPT, using an intelligent engine. We implemented a small system in our case study to validate our methodology. Furthermore, to provide expert advice to the user, we implemented an intelligent engine using a small training set from Matrix Market [22]. The intelligent engine updates the database adaptively as simulations proceed.

In Section II, we present the system architecture of EIGADEPT, how it is implemented and used, what are its major components and how the component are connected. In Section III, we describe a case study using our framework

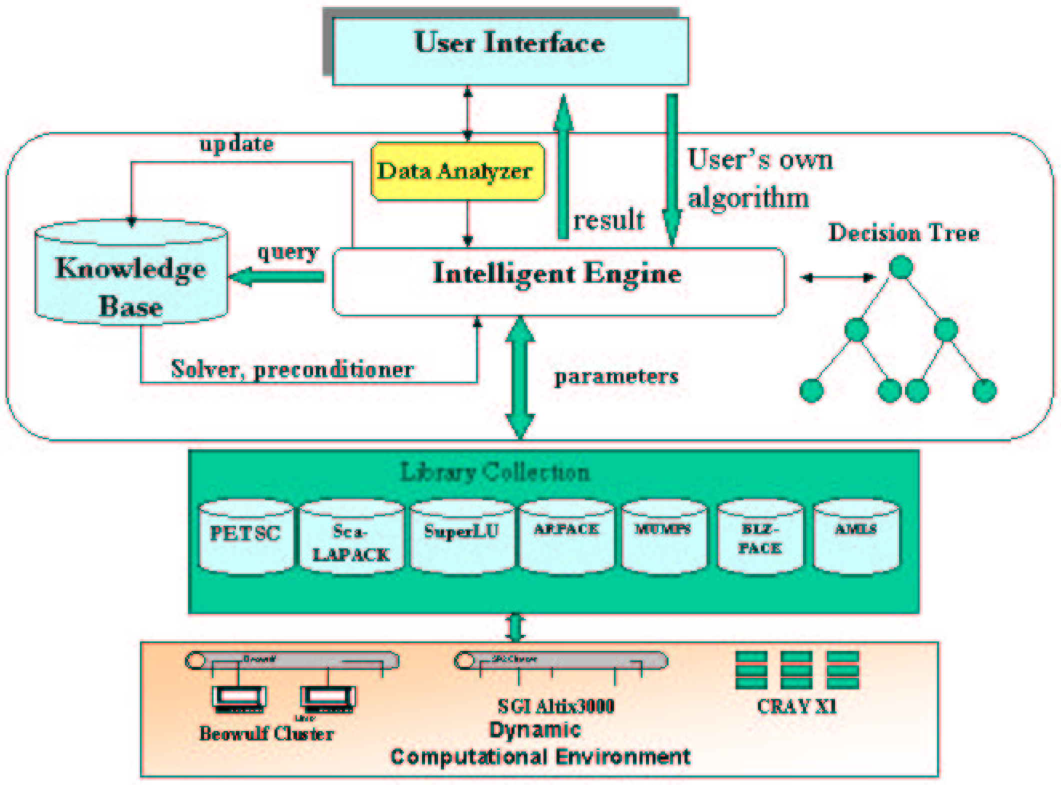

Fig. 1. EIGADEPT Architecture

with a prototype intelligent engine to solve linear systems using various algorithms implemented in PETSc [4]. Section IV discusses the progress and the future plans for the intelligent eigensolver. Section 5 describes related work and states the main differences between our project and others.

II. METHODOLOGY

Choosing an appropriate eigensolver (and its parameters) depends on the mathematical properties of the problem, the desired spectral information, and the available operations and their costs. The Eigentemplates book [3] provides some valuable "decision trees" based on the above information (e.g., Tables 4.1 and 5.1 in [3]). Given the large, high-dimensional algorithm/parameter space, a simple decision-tree mechanism may be insufficient. Furthermore, such information may not be available at runtime or may be costly to obtain. For example, the decision tree needs to know if the matrix is positive definite. The answer to this question may require considerable computation which might not be wanted by the user at runtime. EIGADEPT intends to address the dilemma of "less information but accurate solving" by providing algorithm recommendations based on previous knowledge.

The architecture of EIGADEPT is shown in Figure 1. The main components of the system are:

1) A *Universal User Interface*. For different numerical algorithms with different parameters, it is much easier for the user to call the algorithms with one universal interface. Since most of the legacy scientific codes are programmed in FORTRAN, we provide a universal FORTRAN interface in EIGADEPT. The user call this subroutine in his/her application to invoke EIGADEPT which will then find and execute a suitable algorithm and return the result to the user. The selection process and execution are transparent to the user. The user could also specify his/her preferences for any desired algorithm or execution criteria (such as memory limit, convergence rate, iteration counts, etc.) This information can be useful for choosing the most appropriate solver if there are several candidates available.

2) A *Data Analyzer*. It receives the input data submitted by the user, extracts the necessary information required by the intelligent engine to make algorithm-wise decision, and passes this information to the intelligent engine. For example, the data analyzer decides whether the matrix provided is sparse, symmetric, etc. It also provides the intelligent engine with all the necessary information that can be collected without substantial computation. For example, determining whether a matrix is transposable is costly, so the data analyzer may not be able to provide such information to the intelligent engine. Although this information may be missing, the intelligent engine is still able to find a suitable algorithm provided a similar problem was previously solved.
3) *Decision Trees*. It is a repository of decision trees incorporated from [3].
4) A *Relational Database*. The database initially contains some training sets from which the best solvers and preconditioners for certain applications are known beforehand. The training set is obtained from prior work with various applications. Information about each new application solved by the intelligent engine is stored in the database with all the application properties and solver information. Each database record stores the information about the input problem, including symmetry, sparsity, the best solver and preconditioner, the numerical library that provided this solver and preconditioner, the parameter setting, the execution time using the best solver and preconditioner, and the application type, etc. The database gradually improves its contents as more problems are solved by the intelligent system, thereby making the algorithm prediction increasingly accurate. In particular, the database can be *adapted* at runtime through the repeated solutions of similar eigensystems form a specific application domain. The database is implemented with MySQL, which is a free open source database software with a reliable C API.
5) An *Intelligent Engine*. This is the central part of the EIGADEPT system. After receiving the information from the data analyzer, the intelligent engine first searches the decision tree for an appropriate algorithm. If there is a match, it searches the underlying numerical libraries for a matching subroutine and passes the user's input with the proper parameters to that subroutine. After execution, the intelligent engine returns to the user the results and runtime statistics. If the decision tree does not lead to a choice of a solver, the intelligent engine queries the database for a suitable solver based on the matrix properties and the application type. There could be three outcomes from this query:
   - Exactly one record in the database fits all the specifications provided by the intelligent engine. Then the solver, the preconditioner and the numerical library are selected.
   - More than one record in the database fit the specifications. Then the intelligent engine tries to find an algorithm based on the user's preferred algorithm characteristics such as least amount of memory, convergence rate, etc.
   - No record is found in the database. Then a default solver is used. Note that this decision may not be the best one. However, when computational resources are available, the problem can be solved with different algorithms, in what we call the *off-line execution*. If the off-line execution indicates that a better algorithm could have been used for the application, the database is updated accordingly with better accuracy.

   After choosing the algorithm, the intelligent engine passes the function call with the proper parameters to the corresponding numerical library and returns the execution results to the user when finished.

   Complementary to the system's intelligent engine, the user can provide his/her own choice of algorithm. Then the system will directly use this algorithm from the available numerical libraries without consulting the decision tree and the database. If the user-specified algorithm is not available in the current library repository, the intelligent engine will ask the user to provide his/her own implementation or returns an error message. Each execution will be fed back to the database by the intelligent engine for future reference. Since the algorithm chosen from the decision tree or the database may or may not be optimal, the off-line execution and feedback mechanisms provide adaptation for the algorithm selection.
   The intelligent engine is programmed in C because of its portability, low overhead, and ease of interfacing with other languages. Another reason we use C is because many modern, popular numerical libraries such as PETSc is programmed in C. Therefore, our C program will be easily incorporated into the applications that are using these libraries. We have implemented a Fortran wrapper so that the intelligent engine can also be

called from a Fortran application.

6) *Numerical Libraries*. This is a collection of the currently available libraries for the eigenvalue problems. They are implemented in different programming languages and with different calling sequences. Once an algorithm is chosen by the intelligent engine, it is the intelligent engine's responsibility to pass the parameters required by the underlying library.
7) *Computational Environment*. This is where the application is executed. It can be a stand alone desktop, a Linux cluster, or a hybrid of MPP and shared memory machines.

III. Case Study

In this section, we present a case study using our proposed system to select an iterative solver and a preconditioner for solving a linear system $Ax = b$. Here, we used PETSc [4] as the underlying numerical library since PETSc is widely used and provides a uniform interface to many different linear solvers. The intelligent engine and relational database are implemented in C and MySQL, respectively. The decision tree used is shown in Figure 2, which is taken from [5]. PETSc contains all the algorithms in this decision tree. Among all the solvers and preconditioners provided by PETSc, we have chosen eight solvers: Conjugate Gradient (cg), BiConjugate Gradient (bicg), Generalized Minimal Residual (gmres), BiCGSTAB (bcgs), Conjugate Gradient Squared (cgs), Transpose-Free Quasi-Minimal Residual (tfqmr), Conjugate Residual (cr) and Least Squares Method (lsqr), and four preconditioners: No Preconditioner (none), Jacobi (jacobi), Block Jacobi (bjacobi) and Additive Schwarz (asm). Altogether, there are 32 solver/preconditioner combinations to choose from.

The user invokes the solution process through our universal interface, which has the following prototype:

solve (mp_1, mp_2, mp_3, ..., A, b)

where $mp_i(i = 1...n)$ contains the matrix properties such as $mp_1 : symmetric$, $mp_2 : sparse$, etc. Those matrix properties that the user does not know can be left as NULL. The *Data Analyzer* receives the function call from the user interface, and checks if it can provide further information for the NULL fields. It then passes all the matrix properties to the intelligent engine after analyzing the input information.

After receiving all the information it needs, the intelligent engine first searches the decision tree (see Figure 2) to find out if there is a suitable solver for the problem. In some cases, the decision-tree-search does not result in a leaf if there is missing information. From that point, the intelligent engine will query the database to find a matrix with similar properties and its previous solver. The database will return the solver and preconditioner from PETSc and their required parameters. The intelligent engine will use this information to solve the problem. Often, the decision tree is sufficient to determine a solver but the choice of preconditioner remains open. If so, the database can still be consulted to select the right preconditioner.

To begin our experiment, we need to establish the initial content for the database. To this end, we chose 20 small sized matrices from Matrix Market [22] as our training set. These matrices come from such diverse application domains as acoustic scattering, astrophysics, biochemistry, fluid flow, and quantum physics, etc. We include matrices from different domains for the sake of generality. Although the training set contains only limited types of applications, it is a good starting point for the solver and preconditioner selection for the user application. For each of the 20 matrices in the training set, we ran all 32 solver/preconditioner combinations provided by PETSc, and recorded the best solver/preconditioner in the database, together with the matrix properties and the application domain. In this case study, the "best" means the one with the fastest execution time. For the sake of simplicity, we used only one processor in these experiments. Note that information regarding the new problems and their performance using the solver/preconditioner chosen by the intelligent engine will be added into the database.

We then tested our intelligent system with three matrices from real applications. Table II tabulates the main properties of the three matrices, including application domain, data type, size and number of nonzeros. These three matrices were drawn from the same application domains as the matrices in the training set, but they are of much larger size and more representative of the production runs. For each of the three matrices, we input the following properties through the universal input interface: symmetry, real or complex, size, and application domain. For all three matrices, these properties are not sufficient to arrive at a solver based solely on the decision tree. Therefore, the intelligent engine actually consulted the database for the knowledge of the previous runs of the similar problems in the training set.

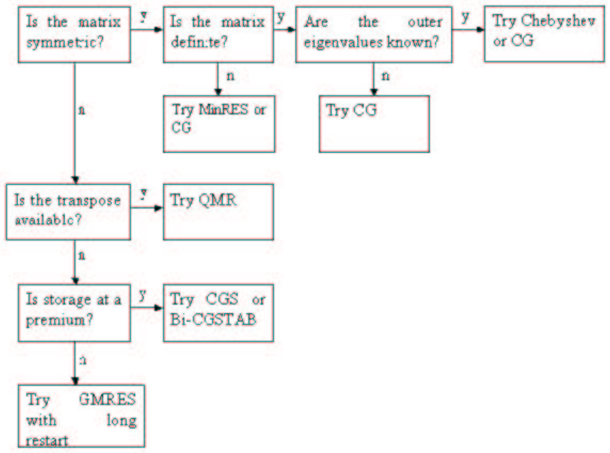

Fig. 2. Decision Tree For Iterative Methods (Taken from [5])

Apps	Matrix	Type	Size	nnz
Reaction-Diffusion	rdb3200l	real, unsymmetric	3200×3200	18880
BCS Structural Engineering	bcsstk26	real, symmetric	1922×1922	30336
Bounded Finline Dielectric Waveguide	bfw782b	real, symmetric	782×782	5982

TABLE II

PROPERTIES OF THE APPLICATION MATRICES

Table III summarizes the results of the experiments. The second column of the table shows the solver/preconditioner selected by the intelligent system. As a comparison, we ran all 32 solver/preconditioner algorithm combinations for the three matrices. Columns 3 and 4 show the best and worst solution times and the associated algorithm choices are given in the parenthesis. For these three linear systems, the algorithm selected by our intelligent engine is always the best. The last column in the table shows the percentage of failures (non-converged) among all the solver/preconditioner combinations. As can be seen, if a user with little numerical experience decided to use any of the 32 combinations, the algorithm could fail to converge or take long time to converge. For example, for rdb3200l, 56.2% solver/preconditioner combinations do not converge. Thus, based only on random selection without any guidance, the user may have to try many times before finding one algorithm that converges. Even if the user chosen solver/preconditioner algorithm converges, the convergence rate varies greatly. The worst/best performance ratios are 8.73, 7.29 and 1.46 for rdb3200l, bcsstk26 and bfw782b, respectively.

We recorded the runtime overhead of our intelligent engine. For all the three applications, the overhead from

Matrix	Automatic Selection	Best time	Worst time	% of Failures
rdb3200l	bicg + none	2.54 (bicg + none)	22.16 (bcgs + asm)	56.2%
bcsstk26	cr + bjacobi	1.62 (cr + bjacobi)	11.78 (bicg + none)	43.8%
bfw782b	cr + jacobi	0.0645 (cr + jacobi)	0.0942 (lsqr + asm)	9.3%

TABLE III

PERFORMANCE OF THE ITERATIVE METHODS AND OUR INTELLIGENT ENGINE. TIMES ARE IN SECONDS.

intelligent engine is less than 1% of the total execution time.

IV. Status and Future Work for Eigensolver

Automatically choosing an optimal eigensolver is in many ways more difficult than doing so with linear solvers. In eigenvalue analyses, some of the important data to be taken into account include: the dimension of the problem, number of eigenvalues (and/or eigenvectors) required, location of the required solutions in the eigenvalue spectrum (i.e. smallest, largest, close to a reference value, etc.), accuracy of the required solutions, and availability of approximate solutions (e.g., obtained from previous simulations with similar problems) [3]. In contrast to iterative linear solvers, the available memory is also important because it can influence the effectiveness of restarting strategies implemented in several algorithms. For many large-scale applications an iterative scheme is the method of choice. However, some simulations may require increasing number of eigenvalues to be computed from one run to another. In this case, we may reach a breaking point where switching to a direct method becomes a viable alternative [7].

We focus on a class of eigensolvers based on projection methods, which transform the original eigenvalue problem into a problem associated with an appropriate subspace of much reduced dimension, and find the best approximations from this reduced subspace. These methods are amenable to scalable implementations. Algorithms of this type have already been implemented in various parallel libraries, including ARPACK (implicitly restarted Arnoldi method) [18], BLZPACK (block Lanczos method) [20], JaDa (block Jacobi-Davidson method) [26], and TRLan (thick-restart Lanczos method) [29]. Table I lists the other candidate implementations. We will enhance some of these eigensolvers with shift-and-invert capabilities using existing scalable sparse direct linear solvers, such as SuperLU [12] and MUMPS [2]. Parallel implementations for some other newly emerged algorithms are not yet available, which is the case with the automatic multilevel substructuring (AMLS) method [6], [15].

We have already identified a number of applications that can be used to further study and validate the ideas discussed in this paper (see the beginning of Section I). The fundamental obstacle was that the aforementioned solvers have different interfaces and parameters. Therefore, our first task was to effectively and efficiently package them so as to deal with their distinct requirements in inputs and outputs. This is the "Library Collection" layer depicted in Figure 1. We implemented this layer in C++. The object programming paradigm in C++ gives us convenient way to add new linear solver or eigensolver into EIGADEPT. Polymorphism provided by C++ enables EIGADEPT to solve the problem using the same library but different input parameters. Another reason of using C++ to implement the Library-Collection layer is due to C++'s good interoperability with C and Fortran. This feature makes C++ a better choice than another popular object-oriented language Java.

By now, we have implemented three major classes: `LINSolver` (Linear Solver) class, `EIGSolver` (Eigenvalue solver) class and `Options` (Options for EIGSolver) class. For sparse matrices, we support both Compress Row Storage (CRS) and Compressed Column Storage (CCS) schemes. We have a `Matrix` class to define the two storage formats and have implemented various methods of the needed operations with these sparse formats. The `Options` class contains all the eigensolver libraries' common input options, such as how many eigen values needed, tolerance etc. If the user has his own eigensolver, he can add it into EIGADEPT by extending `EIGSolver` and `Options` classes. The same scenario holds if the user wants to add a new linear solver into EIGADEPT. In user's application, after defining all the LINSolver, EIGSolver and Options objects, the user shall be able to use a unified interface to solve his problem. If the user wants EIGADEPT to select an eigensolver, any undefined option value is assigned a default value and is passed to the eigensolver chosen by EIGADEPT. The user may choose to use a particular solver by specifying the solver name in the user interface. In this scenario, the user needs to define all the options for this sovler. We have incorporated PARPACK in the EIGADEPT framework using these classes. We are currently importing several other numerical libraries into the framework, including BLZPACK and AMLS.

So far, we have not considered the memory limit, the available computing resources and the user's time requirement. The user will be able to specify this kind of information through the input interface and this information will affect the algorithm selection. The information regarding the application execution environment and the underlying architecture will be used to choose the right implementation. In order to ensure scalable and portable performance of the eigensolvers, we will identify and isolate the performance critical kernels, capture the important hardware characteristics, including memory/communication latency and bandwidth, cache size and incorporate these information into our knowledge base.

We will conduct more research on our data analyzer because the more information it provides, the closer is the selected algorithm to the optimal. Some information such as “Is the matrix transpose available?” or “Is the matrix positive definite?” are useful, but the answer to these questions may require intensive computation. Therefore, we need to find an effective approach to obtain an accurate guess with low overhead.

We will wrap our intelligent system as a CCA [8] component. A great deal of effort has been dedicated to building numerical algorithms with CCA. With the CCA providing/using the “port” mechanism, our intelligent engine can be directly coupled with the other newly developed eigensolver components. Along these lines, the intelligent engine can even be used outside our eigensolver context through well-defined component interface to accommodate different scientific applications.

V. Related Work

There exists a number of research projects seeking goals similar to ours but using different approaches. The Self-Adaptive Numerical Software (SANS) system [13], [14] contains an intelligent engine, a history database, a network scheduler and the underlying adaptable libraries. SANS employ a user provided metadata to automatically analyze the problem and then a scripting language to compose a “ploy-algorithm”. For problems in which that information cannot be quickly obtained, SANS searches a heuristic database storing previous performance data for self-tuning rules. Netsolve [25], [1] is a client-agent-server system which provides remote access to hardware and software resources from a variety of scientific problem solving environments. After the client submits a function request, from a list of all available servers, the agent finds an available server in which the demand of the software request can be met. Then the server executes the function for the client and returns the results. Unlike SANS and our system, Netsolve does not provide suggestions to the user about which solver will be the best and it loyally selects the server hosting the service that the client requests to fulfill the task.

Grid-TLSE [16] provides user a web interface for sparse matrix computation on a computational grid. The main part is an expert site consist of: writing the procedures for the expertise, inclusion of the sparse matrix software, building a database including a bibliography on sparse matrix software and collections of sparse matrices (Rutherford-Boeing and PARASOL data sets). The Scalable Multimethod Sparse Solvers project (SuperSolvers) [24] aims at developing new sparse solution schemes, their analysis, and applications to computational modeling and simulation. SuperSolvers incorporate hybrid solvers (using flexible incomplete sparse factorization preconditioners combined with a range of pure iterative and pure direct method) and composite solvers (using a sequence of basic solution schemes on a single linear system and an adaptive solver dynamically selecting a sparse solution scheme to match changing numerical attributes across iterations of a long running simulation). SuperSolvers focus on selecting a robust and scalable solver tailored to meet application demands. In [19], the authors developed a Linear System Analyzer (LSA) based on component programming paradigms. Users can choose available linear system solver components defined by IDL. As a prototype to exploit the usefulness of component architecture on distributing scientific problem solving, LSA does not give much advice to the users on how to select the solvers and preconditioners to reduce the runtime and improve performance. Thus for a naive user, LSA will not be able to deliver a best optimized result.

EigAdept is focusing on large-scale eigenvalue applications. Unlike SANS, EigAdept is not using metadata defined by the user to describe the input data. Instead, information comes from the heuristic database and the data analyzer that examines the user input data. The intelligent engine acts as a middleware to connect the user application and the numerical libraries. The user does not need to change her application code if she wants to use another library. The combination of on-line and off-line mechanisms adaptively increase the “wisdom” of the intelligent engine algorithm selection procedure.

Acknowledgment

This work was supported by the Director, Office of Advanced Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

We would like to thank Parry Husbands and Tony Drummond of Lawrence Berkeley National Laboratory for their valuable suggestions on our system design. We thank Konrad Malkowski of Penn State University for providing the initial C++ implementation of the eigensolver interface.

References

[1] S. Agrawal, J. Dongarra, K. Seymour, and S. Vadhiyar. Netsolve: Past, present, and future - a look at a grid enabled server. In A. Hey eds. F. Berman, G. Fox, editor, *Making the Global Infrastructure a Reality*, 2003.

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. MUltifrontal Massively Parallel Solver (MUMPS version 4.3) Users' Guide. Technical report, ENSEEIHT-IRIT, Toulouse, France, July 2003.

[3] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, 2000.

[4] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. Mclnnes, B. F. Smith, and H. Zhang. PETSc Users Manual. `http://www.mcs.anl.gov/petsc`.

[5] Richard Barrett, Michael W. Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, , and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1994.

[6] J. K. Bennighof and R. B. Lehoucq. An automated multileve substructuring method for eigenspace computation in linear elastodynamics. *SIAM Journal on Scientific Computing*, 25(6):2084–2106, 2004.

[7] L. S. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.

[8] The common component architecture forum. `http://www.cca-forum.org`.

[9] F. Y. Cheng. *Matrix Analysis of Structural Dynamics: Applications and Earthquake Engineering*. Marcel Dekker, New York, N.Y., 2000.

[10] A. K. Chopra. *Dynamics of Structures: Theory and Applications to Earthquake Engineering*. Prentice Hall, Upper Saddle River, N.J., 2nd edition, 2000.

[11] J. V. Clark, D. Bindel, N. Zhou, S. Bhave, Z. Bai, J. Demmel, and K. S. J. Pister. SUGAR: Advancements in a 3D multi-domain simulation package for MEMS. In *Proceedings of the Microscale Systems: Mechanics and Measurements Symposium*, Portland, OR, June 4 2001.

[12] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. Software is available at `http://crd.lbl.gov/~xiaoye/SuperLU/`.

[13] J. Dongarra and V. Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *Proceedings of the International Conference on Computational Science*, June 2 2003.

[14] J. Dongarra and V. Eijkhout. Self-adapting numerical software for next generation applications. *The International Journal of High Performance Computing Applications*, 17(2):125–131, 2003.

[15] Weiguo Gao, Xiaoye S. Li, Chao Yang, and Zhaojun Bai. Performance evaluation of a multilevel sub-structuring method for sparse eigenvalue problems. In *Proceedings of the 16th International Conference on Domain Decomposition Methods*, January 2005.

[16] GRID-TLSE. `http://www.enseeiht.fr/lima/tlse`.

[17] K. Ko, N. Folwell, L. Ge, A. Guetz, V. Ivanov, L. Lee, Z. Li, I. Malik, W. Mi, C. Ng, and M. Wolf. Electromagnetic systems simulation - from simulation to frabrication. SciDAC Report, 2003. Menlo Park, CA.

[18] Rich Lehoucq, Kristi Maschhoff, Denny Sorensen, and Chao Yang. Parallel ARPACK.
`http://www.caam.rice.edu/~kristyn/parpack\_home.html`.

[19] The linear system analyzer. `http://www.extreme.indiana.edu/pseware/LSA/LSAhome.html`.

[20] O. A. Marques. BLZPACK: Description and User's Guide. Technical Report TR/PA/95/30, CERFACS, Toulouse, France, 1995.

[21] O. A. Marques and Y.-H. Sanejouand. Hinge-Bending Motion in Citrate Syntase Arising from Normal Modes Calculations. *Proteins: Structure, Function and Genetics*, 23:557–560, 1995.

[22] Matrix Market. `http://math.nist.gov/MatrixMarket`.

[23] M. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J.D. Joannopoulos. Iterative minimization techniques for ab initio total energy calculations: Molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, 1045, 1992.

[24] B. Norris S. Bhowmick, L. McInnes and P. Raghavan. Robust algorithms and software for parallel pde-based simulations. In *Proceedings of HPC 2004, The Twelfth Special Symposium on High Performance Computing at the 2004 Advanced Simulation Technologies Conference*, pages 37–42, Arlington, VA, April 2004.

[25] K. Seymour, A. Yarkhan, S. Agrawal, and J. Dongarra. Netsolve: Grid enabling scientific computing environments. In L. eds. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*, 2005.

[26] A. Stathopoulos and J.R. McCombs. A Parallel, Block, Jacobi-Davidson Implementation for Solving Large Eigenproblems on Coarse Grain Environments. In *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications*, pages 2920–2926. CSREA Press, 1999.

[27] M. Turk and A. Pentland. Eigenfaces for Recognition. Technical report, Vision and Modeling Group, The Media Laboratory, MIT, 1990.

[28] G. Verdu, D. Ginestar, V. Vidal, and J. L. Muñoz Cobo. 3D λ-Modes of the Neutron-Diffusion Equation. *Ann. Nucl. Energy*, 21:405–421, 1994.

[29] Kesheng Wu and Horst Simon. Thick-restart lanczos method for large symmetric eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 22(2):602–616, 2001.