Lighthouse: A Taxonomy-based Solver Selection Tool

Kanika Sood
Computer and Information Science,
University of Oregon, Eugene, OR
kanikas@cs.uoregon.edu

Boyana Norris
Computer and Information Science,
University of Oregon, Eugene, OR
norris@cs.uoregon.edu

Elizabeth Jessup
Department of Computer Science,
University of Colorado, Boulder, CO
jessup@cs.colorado.edu

Abstract

Linear algebra provides the building blocks for a wide variety of scientific and engineering simulation codes. Users face a world of continuously developing new algorithms and high-performance implementations of these fundamental calculations. In this paper, we describe new capabilities of our Lighthouse framework, whose goal is to match specific problems in the area of high-performance numerical computing with the best available solutions developed by experts. Lighthouse provides a searchable taxonomy of popular but difficult to use numerical software for dense and sparse linear algebra. Because multiple algorithms and implementations of the same mathematical operations are available, Lighthouse also classifies algorithms based on their performance. We introduce the design of Lighthouse and show some examples of the taxonomy interfaces and algorithm classification results for the preconditioned iterative linear solvers in the Parallel Extensible Toolkit for Scientific Computation (PETSc).

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques

General Terms linear algebra, mathematical software, machine learning, taxonomy

1. Introduction

Solving large linear systems and computing eigenvalues are fundamental problems in high-performance scientific and engineering computing (HPC). In response to the need for high-performance algorithms, applied mathematics and computer science researchers have created a number of comprehensive numerical software packages that are widely used today. Because of the number and complexity of the algorithms available in these packages, finding the most suitable solution to a particular problem is a nontrivial task, even for experts in numerical methods. For example, for the relatively limited problem of solving a dense system of linear equations, one numerical library LAPACK [1] alone offers over 100 different functions.

Lighthouse is a framework for creating, maintaining, and using a taxonomy of available software for highly optimized matrix algebra computations. The taxonomy serves as a guide to HPC application developers seeking to learn what is available for their programming tasks, how to use it, and how the various parts fit together. In this paper, we briefly introduce the Lighthouse tool (described in more detail in [5]), then focus on new Lighthouse capabilities in the area of parallel iterative solvers for sparse linear systems in the Portable Extensible Toolkit for Scientific Computation (PETSc) [7–9]. PETSc has been used for modeling in many areas, ranging from acoustics [29] to brain surgery [6] to ocean dynamics [28].

2. Design and Implementation

The main components of the Lighthouse framework are illustrated in Figure 1. Lighthouse defines its software taxonomy using Django [3] models, which correspond to MySQL databases that store information about the different packages. Currently Lighthouse contains routines from LAPACK, SLEPc [2] and PETSc.

Lighthouse enables search to the user via three methods: guided search, advanced search and keyword search. In the guided search, users are asked detailed questions in order to describe the problem they wish to solve with Lighthouse. After answering all the questions, the user sees exactly one subroutine that corresponds to all the answers provided. The keyword search interface supports keyword-based search of the taxonomy information. In it, Lighthouse supports auto completion of words and spelling correction. Finally, the advanced search is geared toward users who are familiar with the packages.

Lighthouse generates two types of code. First, given a specific search result, Lighthouse provides users with the option of generating a complete program that uses those function(s) correctly, including declarations and initialization of

SEPS '15, October 25–30, 2015, Pittsburgh, Pennsylvania, United States.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

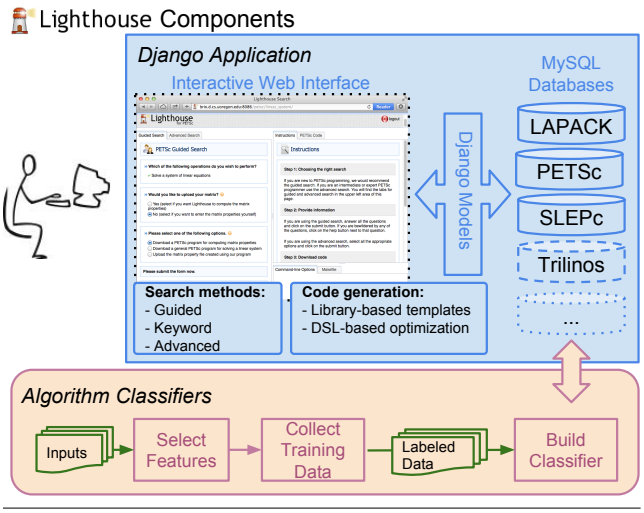

Figure 1. Lighthouse overview

all data structures. Second, Lighthouse provides a separate experimental interface that allows users to define their own high-level linear algebra computations for dense and some sparse linear systems and vectors. Using a MATLAB-like language as input, Lighthouse interfaces with the BTO compiler [10, 27] to generate highly optimized C implementations that can be downloaded and used in larger applications.

3. Taxonomy Search Interfaces

We briefly illustrate the user interfaces of Lighthouse with two examples: one that leads to functionality implemented by LAPACK and another that involves solving a large sparse linear system using PETSc. For each numerical library supported by Lighthouse, we have created decision trees with the questions used in the guided search interfaces. These trees are stored in databases, from which the actual interactive Web forms are generated dynamically.

Figure 2 shows an example of a guided search interactive session, in which the user wishes to compute the condition number of a matrix. The interfaces for other LAPACK functionality are structured similarly. After answering a few questions, the user is given a single function in LAPACK (which has dozens of functions that relate to computing condition numbers). The user then can have Lighthouse generate a code template in C or Fortran, which is a complete program that can be downloaded, compiled, executed, and modified as needed for integration into larger applications.

Figure 3 shows the guided search dialog in the Lighthouse PETSc interface which has a series of questions for the user. It is an interactive system that enables users to generate as well as download PETSc programs for solving sparse linear systems. To begin, the user has the option to upload a coefficient matrix and make Lighthouse compute the matrix's properties or *features*. If the user chooses this option, Lighthouse computes them and uses them in a machine learning classification process (described in Section 4) to predict a good performing solver for the given system with that coefficient matrix. On the other hand, a user who does not opt to upload the matrix is offered the following options by Lighthouse: a PETSc program to download and use for computing matrix properties or a general PETSc program for solving a linear system.

The code generated by Lighthouse as a result of the guided search is shown in Figure 4. Lighthouse currently generates code templates in C or Fortran90 for the functions returned in the search. The template is a complete program, including all variable declarations and correct invocation of library functions, many of which take very large numbers of parameters. The template program is easy to modify as it is divided into multiple subprograms.

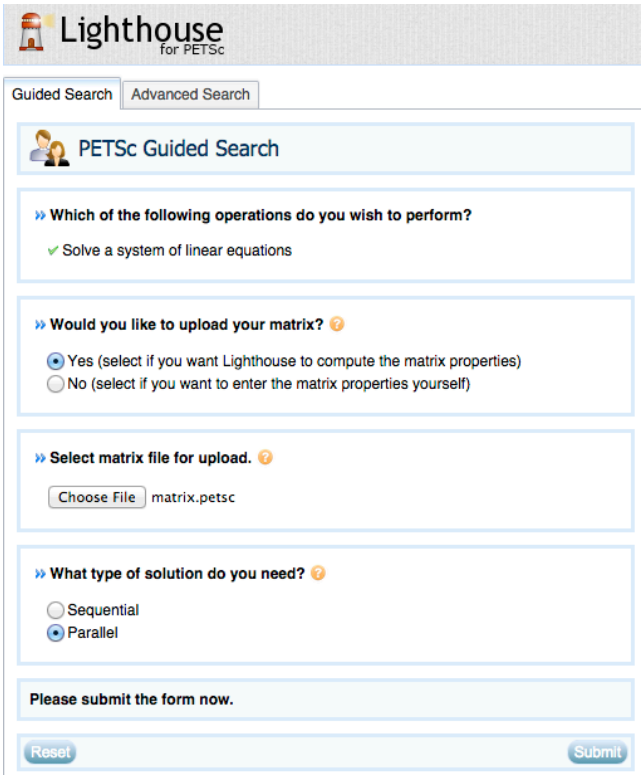

Figure 3. PETSc guided search interface

4. Sparse Linear Solver Classification

Large sparse linear system solution is a fundamental computational step in many scientific and engineering applications. Users generally choose one of many iterative methods for approximating the solution to a system (subsequently, we refer to these methods as *solvers*). Individual solver convergence and performance is highly dependent on the specific input problem characteristics and can be difficult to predict.

Lighthouse automates the performance-based solver selection process by employing machine learning techniques to classify solvers based on a training set generated with over 1,000 linear systems arising from different applications. We

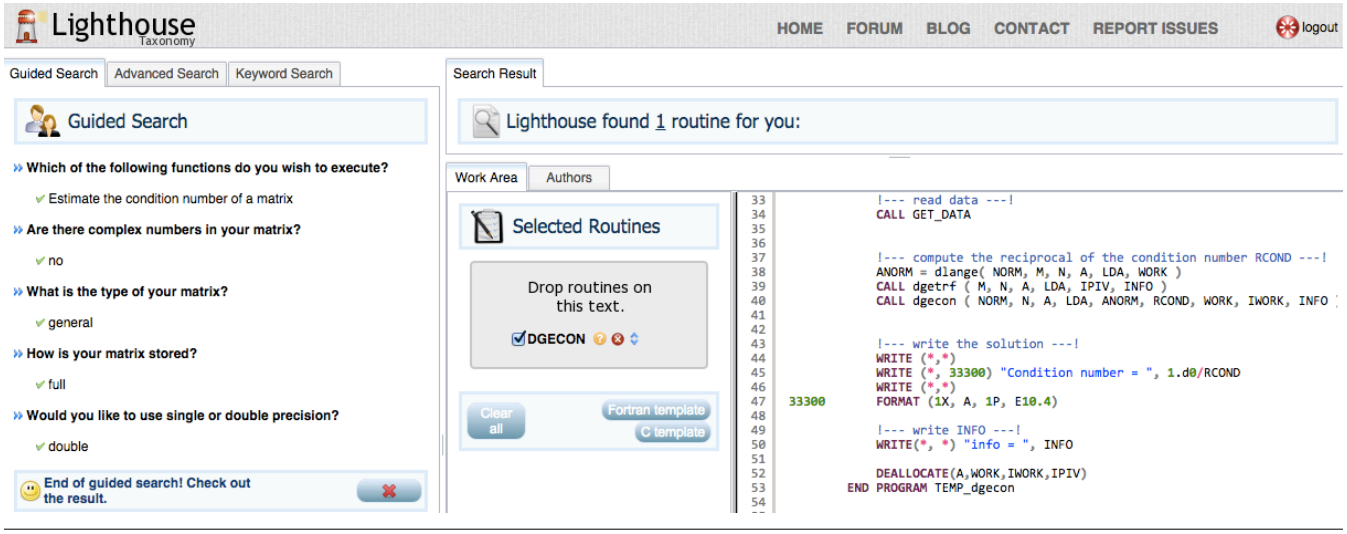

Figure 2. Computing the condition number of a matrix with LAPACK (guided search)

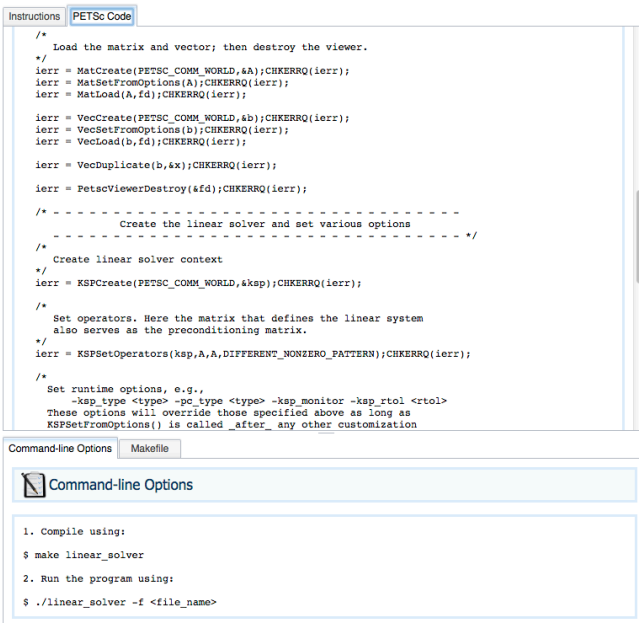

Figure 4. PETSc code template resulting from guided search

applied several supervised machine learning techniques to help make this decision based on relatively few, easily computable properties of the input system. Supervised learning involves determining a classification based on a set of already classified data. These data are split into training and testing sets. The training set is used to build the classifier and the testing set is used to verify the accuracy of the classifier. For our experiments, this process was repeated 10 times (10-fold cross validation), each time with a different subset as the testing set. Our ultimate goal is to provide a scalable and automated approach for choosing an efficient and accurate solver for a given linear system by classifying linear solvers using several machine learning methods. The success of previous research into performance-based multi-solver methods [11, 16, 19] motivates the addition of solver classification to Lighthouse, which we describe next.

Linear solver classification in Lighthouse is performed as a sequence of steps. Note that this process is generalizable to other problems beyond sparse linear system solution. The features used in building the solver classifiers include linear system properties such as simple (norm-like) quantities, variance (heuristics estimating how different matrix elements are), normality (estimates of the departure from normality), structure (nonzero structure properties), and spectral properties. Because computing so many matrix properties (in our case 68) while an application is executing is too costly, we also performed feature set reduction, removing the features that do not contribute significantly to the process of deciding the best solver for a given linear system. Feature set reduction brings down the overall cost of the process for building *and* using the classifiers. The next step involves building classifiers by using several machine learning methods in Weka [18]: BayesNet [12], k-nearest neighbor [15], Alternate Decision Trees [17], Random Forests [13], J48 [25], Support Vector Machines (SVM) [14], Decision Stump [21] and LADtree [20] methods. We compared the performance of these methods to select the one that produces the best (most accurate) solver classification.

4.1 Classification Results

To generate the input dataset used by the machine learning methods to classify solvers, we measured the performance of several solver and preconditioner combinations in PETSc version 3.5.3 on a Blue Gene/Q supercomputer on more than 1,000 linear systems from the University of Florida matrix collection [4]. For each input, we computed the lin-

ear system's features, then solved the system using a specific solver configuration and measured the execution time, resulting in 4,648 data points. Next, these data were used to build solver classifiers by using different machine learning methods. The results were evaluated using 10-fold cross validation. In Lighthouse, we focus only on true positives (i.e., best-performing solvers predicted as best-performing). Among all machine learning methods we used, BayesNet produced the best accuracy of 87.6% with 68 input features. Using only eight computationally inexpensive features, the BayesNet-based classifier predicted good (well-performing) solvers correctly 86.9% of the time. Figure 5 shows the true positive prediction accuracy of several of the machine learning methods we tested for the full and reduced feature sets.

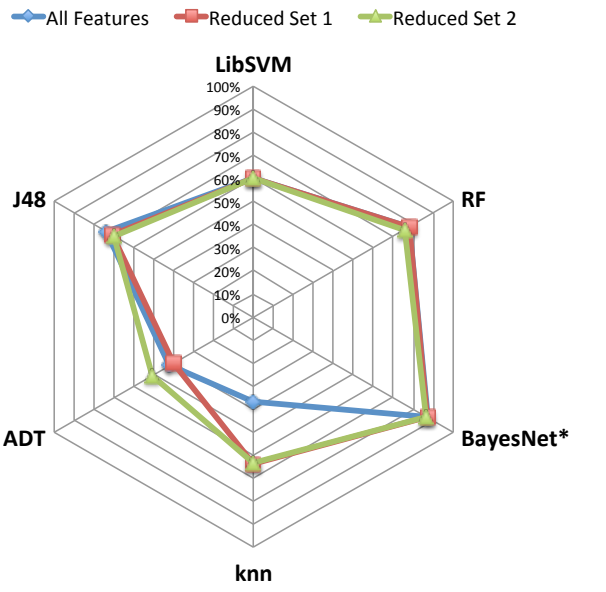

Figure 5. Machine learning method accuracy comparison.

In addition to the accuracy of the classifier, we also considered the time it took to build the classifier, which can vary by orders of magnitude between different methods. The BayesNet method, which has the highest accuracy, took a fraction of a second to build the PETSc solver classifier on an Intel Core i5 MacBook Pro. The slowest method was LibSVM, taking over a minute to build the classifier.

4.2 Using the Classification Results to Select Solvers

When a user is able to provide an example linear system from a specific application, Lighthouse can compute the reduced feature set and use the previously generated classifier to return a list of solver configurations that are likely to perform well. In future work, we also plan to enable users to check whether a user-specified solver configuration is likely to perform well.

5. Evaluation

As Lighthouse continues to grow, we plan to apply formal methods of user testing from human computer interaction (e.g., [22–24, 26]) to evaluate its usability. For example, we will employ techniques for interface developers, such as cognitive walkthrough [26] and heuristic evaluation [23, 24], and we will continue to use those directed at users, such as thinking aloud [22].

To date, we have carried out thinking aloud evaluations of Lighthouse in graduate courses at the University of Colorado Boulder. The clients had education and expertise in fields of engineering (aerospace, electrical, civil), mathematics, computer science, and computational science and varying expertise in linear algebra and programming. These clients discovered a number of usability issues and also made very good suggestions for improvement of Lighthouse. These preliminary evaluations aptly demonstrate the value of user feedback.

6. Conclusions and Future Work

At present, Lighthouse for PETSc has an efficient navigation system that allows users to generate, download and extend PETSc applications for solving large sparse linear systems. Because there are thousands of valid linear solver configurations, we have added to Lighthouse a machine learning-based solver classification. It can substantially improve developer productivity because there is no need to experiment with various solvers in an ad hoc fashion, and it can improve application performance.

Future work includes extending the present Lighthouse interface to automate more fully the solver selection process. More automation will enable the regular regeneration of the classifiers as more data are collected from different applications, both user-provided and our own. Our current analysis considers only sequential runs; we aim towards adding runs in parallel in the future as well. Other future goals involve expanding the number of iterative solvers and preconditioners covered by the framework and improving the performance accuracy.

Acknowledgments

This work is supported by the U.S. Department of Energy Office of Science (Contract No. DE-SC0013869) and by the National Science Foundation (NSF) awards CCF-1219089, CCF-155063 and CCF-1550202.

References

[1] LAPACK - Linear Algebra PACKage. `http://www.netlib.org/lapack/`, 2014.

[2] Scalable Library for Eigenvalue Problem Computations (SLEPc). `http://www.grycap.upv.es/slepc/`, 2014.

[3] Django. `https://www.djangoproject.com/`, 2015.

[4] The University of Florida Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/, 2015.

[5] The Lighthouse Project. http://lighthousehpc.github.io/lighthouse/, 2015.

[6] Y. Ataseven, Z. Akalın-Acar, C. Acar, and N. G. Gençer. Parallel implementation of the accelerated bem approach for emsi of the human brain. *Medical & Biological Engineering & Computing*, 46(7):671–679, 2008.

[7] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[8] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015. URL http://www.mcs.anl.gov/petsc.

[9] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc Web page, 2015. URL http://www.mcs.anl.gov/petsc.

[10] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, 2009. ACM. ISBN 978-1-60558-744-8. .

[11] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of alternating decision trees in selecting sparse linear solvers. In *Software Automatic Tuning*, pages 153–173. Springer, 2010.

[12] C. Bielza and P. Larrañaga. Discrete Bayesian network classifiers: A survey. *ACM Comput. Surv.*, 47(1):5:1–5:43, July 2014. ISSN 0360-0300. . URL http://doi.acm.org/10.1145/2576868.

[13] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001. ISSN 0885-6125. . URL http://dx.doi.org/10.1023/A:1010933404324.

[14] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[15] P. Cunningham and S. J. Delany. k-nearest neighbour classifiers. *Multiple Classifier Systems*, pages 1–17, 2007.

[16] P. R. Eller, J.-R. C. Cheng, and R. S. Maier. Dynamic linear solver selection for transient simulations using machine learning on distributed systems. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1915–1924. IEEE, 2012.

[17] Y. Freund and L. Mason. The alternating decision tree learning algorithm. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 124–133, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-612-2. URL http://dl.acm.org/citation.cfm?id=645528.657623.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[19] A. Holloway and T.-Y. Chen. Neural networks for predicting the behavior of preconditioned iterative solvers. In *Computational Science–ICCS 2007*, pages 302–309. Springer, 2007.

[20] G. Holmes, B. Pfahringer, R. Kirkby, E. Frank, and M. Hall. Multiclass alternating decision trees. In *Machine learning: ECML 2002*, pages 161–172. Springer, 2002.

[21] W. Iba and P. Langley. Induction of one-level decision trees. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 233–240, 1992.

[22] C. Lewis. Using the thinking-aloud method in cognitive interface design. Technical Report IBM Research Report RC 9265, IBM, Yorktown Heights, NY, 1982.

[23] J. Nielsen. How to conduct a heuristic evaluation. http://www.useit.com/papers/heuristic/heuristic_evaluation.html, 2007.

[24] J. Nielsen. Ten usability heuristics. http://www.useit.com/papers/heuristic/heuristic_list.html, 2007.

[25] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[26] J. Rieman, M. Franzke, and D. Redmiles. Usability evaluation with the cognitive walkthrough. http://www.acm.org/sigs/sigchi/chi95/Electronic/documnts/tutors/jr_bdy.htm, 2004.

[27] J. G. Siek, I. Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *Workshop on Performance Optimization for High-Level Languages and Libraries (PO HLL 2008)*, pages 1–8, April 2008.

[28] A. D. T. van Scheltinga, P. G. Myers, and J. D. Pietrzak. A finite element sea ice model of the Canadian Arctic Archipelago. *Ocean dynamics*, 60(6):1539–1558, 2010.

[29] J. Wang, Y. Wang, W.-k. Hu, and J.-k. Du. The high performance of finite element analysis and applications of surface acoustic waves in finite elastic solids. In *Piezoelectricity, Acoustic Waves, and Device Applications, 2008. SPAWDA 2008. Symposium on*, pages 66–71. IEEE, 2008.