Multi-stage Learning of Linear Algebra Algorithms*

Victor Eijkhout[†] Erika Fuentes[‡]

Abstract

In evolving applications, there is a need for the dynamic selection of algorithms or algorithm parameters. Such selection is hardly ever governed by exact theory, so intelligent recommender systems have been proposed. In our application area, the iterative solution of linear systems of equations, the recommendation process is especially complicated, since the classes have a multi-dimensional structure. We discuss different strategies of recommending the different components of the algorithms.

1. Introduction

In various areas of numerical analysis, there are several possible algorithms for solving a problem. Examples are the various direct and iterative solvers for sparse linear systems, or routines for eigenvalue computation or numerical optimization. Typically, there is no governing theory for finding the best method, or the theory is in essence uncomputable. For instance, in iterative linear system solving, there are many preconditioners and iterative schemes; the convergence behaviour is determined by the decomposition of the right-hand side in terms of the eigenvectors of the preconditioned operator. However, computing this spectrum is expensive, and the dependency of the iterative scheme on this decomposition is not known in explicit form.

Thus, the choice of the optimal method is in practice determined by experimentation and 'numerical folklore'. However, a more systematic approach is needed, for instance since such choices may need to be made in a dynamic context such as a time-evolving system. Certain recent efforts have tried to tackle this situation by the application of various automatic learning techniques [3, 6, 14, 11, 21]. Such methods perform mining of the problem features, typically relying on a database of prior knowledge, in order to recommend a suitable method.

[†]Texas Advanced Computing Center, The University of Texas at Austin, eijkhout@tacc.utexas.edu

[‡]Currently at Microsoft Research; work performed while at Innovative Computing Laboratory, University of Tennessee, efuente@microsoft.com

In this paper we introduce the problem area, and show methods we have developed for recommending algorithms. Our techniques differ from those presented earlier in that we consider methods to be structured objects, consisting of a composition of algorithms. Correspondingly, we describe strategies for recommending methods that acknowledge this structure, rather than considering methods to be elements out of a flat set of elements.

2. Numerical algorithms and the need for learning

The central problem of this paper, selecting the optimal algorith for a numerical problem, implies that we have a specific problem context in mind. That can be linear system solving, eigenvalue computations, numerical quadrature, et cetera. For each of these areas, there is a definition of a numerical problem: in the case of linear systems, a problem will be a pair $\langle A, b\rangle$ of matrix and righthand side; for eigenvalue problems it denotes a matrix plus optionally an interval for the eigenvalues, et cetera.

We denote the space of numerical problems by $\mathbb{A}$. Corresponding to the problem space, there is a result space $\mathbb{R} = \mathbb{S} \times \mathbb{T}$ containing computed solutions and performance measurements of the computing process. These definitions are again dependent on the context: they can be the solution vectors of linear systems, eigenvalues plus eigenvectors, et cetera; performance measurements can include accuracy measurements as well as runtime.

Having defined problems and results, we define the space of methods $\mathbb{M} = \{\mathbb{A} \mapsto \mathbb{R}\}$ (potentially) solving the class of numerical problems:

$$M(A) = \begin{cases} \bar\sigma & \text{the method `works'} \\ \perp & \text{the method `breaks down'} \end{cases}$$

We introduce a performance measurement function $T(A, M)$, often denoting the time that it takes method M to solve the problem A. In cases where $M(A)$ is undefined because of method breakdown, we can set $T(A, M) = \infty$.

We reduce a numerical problem A to a vector in k-dimensional *feature space* $\mathbb{F}$. The feature extraction function is denoted $\Phi : \mathbb{A} \mapsto \mathbb{F}$.

Feature space can be considerably complicated in structure: elements can be real numbers (for instance matrix elements), positive real (norms), integer (quantities related to matrix size and sparsity pattern), or elements of a finite set of choices. They can even be array-valued in any of these types.

3. Numerical methods

In a simple-minded view, method space can be considered as a finite, unordered, collection of methods $\{M_1, \ldots, M_k\}$, and in some applications this may even be the most appropriate view. However, in the context of linear system solving a method is a more structured entity: each method consists at least of the choice of a preconditioner and the choice of an iterative scheme (QMR, GMRES, et cetera), both of which are independent of each other. Other possible components of a method are scaling of the system, and permutations for improved load balancing. Thus we arrive at a picture of a number of preprocessing steps that transform the original problem into another one with the same solution – or with a different solution that can easily be transformed into that of the original problem – followed by a solver algorithm that takes the transformed problem and yields its solution.

This section will formalize this further structure of the method space.

3.1. Formal definition

Above, we had defined $\mathbb{M}$ as the set of mappings $\mathbb{A} \mapsto \mathbb{R}$. We now split that as the preprocessors $\mathbb{P} = \{\mathbb{A} \mapsto \mathbb{A}\}$ and the solvers $\mathbb{K} = \{\mathbb{A} \mapsto \mathbb{R}\}$. To model the fact that we have different kinds of preprocessors, we posit the existence of subsets $\mathbb{P}_i \subset \mathbb{P}$, and we will assume that the identity mapping is contained in each. For instance, one $\mathbb{P}_i$ could be the set of scalings of a linear system:

$$\mathbb{P}_4 = \{'\text{none}', '\text{left}', '\text{right}', '\text{symmetric}'\}$$

Other possibilities are permutations of the linear system, or approximations of the coeffient matrix prior to forming the preconditioner.

Applying one preprocessor of each kind then gives us the definition of a method: $m \in \mathbb{M} \colon m = k \circ p_n \circ \cdots \circ p_1$. We leave open the possibility that certain preprocessors can be applied in any sequence (for instance scaling and permuting a system commute), while for others different orderings are allowed but not equivalent. Some preprocessors may need to be executed in a fixed location; for instance, the computation of a preconditioner will usually come last in the sequence of preprocessors.

3.2. Application

In our application, the iterative solution of linear system, we consider one preprocessor: the 'preconditioner', which transforms the system $Ax = b$ into $M^{-1}Ax = M^{-1}b$ with, presumably, favourable properties. The solvers are then the iterative schemes that generate the sequences $\{x_i\}_{i\geq 1}$ that, presumably converge to the solution of the system.

3.3. Feature selection

Feature selection for numerical problems is not trivial; considering a matrix as a mere 'array of numbers' will not give meaningful results. Instead, we use a library called AnaMod that we wrote [8, 19]. It can compute 45 features, in various categories, such as structural features, norm-like features, measures of the spectrum and of the departure from normality. The latter two are obviously approximated rather than computed exactly, since computing them would be more expensive than solving the problem.

4. Method selection

Our method selection problem can be formalized as of constructing a function $\Pi : \mathbb{A} \mapsto \mathbb{M}$ that maps a given problem to the optimal method. We start with a brief discussion of precisely what is meant by 'optimal'. After that, we will refine the definition of Π to reflect the preprocessor/solver structure, and we will address the actual construction of Π.

4.1. Different classification criteria

The simplest (non-constructive) definition of the method selection function Π is:

$$\Pi(A) = M \quad \equiv \quad \forall_{M' \in \mathbb{M}} \colon T(A, M) \leq T(A, M') \quad (1)$$

Several variant definitions are possible. Often, we are already satisfied if we can construct a function that picks a working method. For that criterium, we define Π non-uniquely as

$$\Pi'(A) = M \quad \equiv \quad M \text{ satisfies } T(A, M) < \infty$$

Also, we usually do not insist on the absolutely fastest method: we can relax equation (1) to a method being within a certain tolerance of being the fastest.

Formally, we define two classification types:

classification for reliability This is the problem of finding any method that will solve the problem, that is, that will not break down, stall, or diverge.

classification for performance This is the problem of finding the fastest method for a problem, possibly within a certain margin.

In a logical sense, the performance classification problem also solves the reliability problem. In practice, however, classifiers are not infallible, so there is a danger that the performance classifier will mispredict, not just by recommending a method that is slower than optimal, but also by possibly recommending a diverging method. Therefore, in practice a combination of these classifiers may be preferable.

We will now continue with discussing the practical construction of (the various guises of) the selection function Π.

4.2. Method selection on problem features

We define the selection function Π as a function on the feature space $\mathbb{F}$: $\Pi\colon \mathbb{F} \mapsto \mathbb{M}$ is the function that picks the best solution method. The method selection problem then becomes

$$\Pi(\bar{x}) = M \quad \begin{array}{l}\text{if there is a problem } A \text{ with features } x,\\ \text{and } M \text{ is such that}\\ \forall_{M'\in\mathbb{M}}\colon T(A,M) \leq T(A,M').\end{array} \tag{2}$$

Mindful of the above discussion of preprocessors and solvers, we refine the definition of Π and introduce the selector function for preprocessor i $\Pi_i\colon \mathbb{F} \mapsto \mathbb{P}_i$ and $\Pi_k\colon \mathbb{F} \mapsto \mathbb{K}$, the method selector. This also requires us to refine the method selection process. We will discuss this topic further in section 5.

5. Classification of composite methods

In the previous section, we posited a general mechanism of deriving Π, based on a database $\mathcal{D} \subset \{\mathbb{F}\times\mathbb{M} \to \mathbb{T}\}$. This ignores the fact that $\mathbb{M}$ has a structure of preprocessors and solvers, and that we want to recommended each of these individually. In this section we will consider ways of defining databases $\mathcal{D}_\mathbb{K}, \mathcal{D}_\mathbb{P}$ and attendant functions $\Pi_\mathbb{K}, \Pi_\mathbb{P}$, and of combining these into an overall recommendation function.

For the case of Bayesian classification we base the construction of the Π functions by assuming the existence of a mechanism to derive from $\mathcal{A} \subset \mathbb{A}$ a function $\sigma_\mathcal{A} : \mathbb{F} \mapsto [0,1]$. The intended interpretation is that $\mathcal{A}$ is the set of problems for which a certain method M performs best, and $\sigma_\mathcal{A}$ is the function that determines whether a new problem is in the set (presumably a superset of $\mathcal{A}$) for which that method is the best choice. For now, we also restrict our composites to a combination of one preprocessor (in practice, the preconditioner), and one solver (the iterative method); that is $\mathbb{M} = \mathbb{P} \times \mathbb{K}$.

At first we consider the performance problem, where we recommend a method that will minimize solution time (refer to section 4.1 for a definition of the two types of classification). Then, in section 5.4 we consider the reliability problem of recommending a method that will converge, no matter the solution time.

5.1. Combined recommendation

In this strategy, we ignore the fact that a method is a product of constituents, and we simply enumerate the elements of $\mathbb{M}$. Our function Π is then based on the database

$$\mathcal{D} = \{\langle f, \langle p,k\rangle, t\rangle \mid \exists_{a\in\mathcal{A}}\colon t = T(p,k,a)\}$$

and the recommendation function is a straightforward mapping $\Pi^{\text{combined}}(f) = \langle p,k\rangle$.

The main disadvantage to this approach is that, with a large number of methods to choose from, some of the classes can be rather small, leading to insufficient data for an accurate classification.

5.2. Orthogonal recommendation

In this strategy we construct separate functions for recommending elements of $\mathbb{P}$ and $\mathbb{K}$, and we put together their results.

We define two derived databases that associate a solution time with a feature vector and a preprocessor or solver separately, even though strictly speaking both are needed to solve a problem and thereby produce a solution time. For solvers:

$$\mathcal{D}_\mathbb{K} = \left\{\langle f,k,t\rangle \mid k\in\mathbb{K},\, \exists_{a\in\mathcal{A}}\colon f = \phi(a), t = \min_{p\in\mathbb{P}} T(p,k,a)\right\}$$

and for preprocessors:

$$\mathcal{D}_\mathbb{P} = \left\{\langle f,p,t\rangle \mid p\in\mathbb{P},\, \exists_{a\in\mathcal{A}}\colon f = \phi(a), t = \min_{k\in\mathbb{K}} T(p,k,a)\right\}.$$

From these, we derive the functions $\Pi_\mathbb{K}, \Pi_\mathbb{P}$ and we define

$$\Pi^{\text{orthogonal}}(f) = \langle \Pi_\mathbb{P}(f), \Pi_\mathbb{K}(f)\rangle$$

5.3. Sequential recommendation

In the sequential strategy, we first recommend an element of $\mathbb{P}$, use that to transform the system, and recommend an element of $\mathbb{K}$ based on the transformed features.

Formally, we derive $\Pi_\mathbb{P}$ as above from the derived database

$$\mathcal{D}_\mathbb{P} = \left\{\langle f,p,t\rangle \mid p\in\mathbb{P},\, \exists_{a\in\mathcal{A}}\colon f = \phi(a), t = \min_{k\in\mathbb{K}} T(p,k,a)\right\}.$$

but $\Pi_\mathbb{K}$ comes from the database of all preprocessed problems:

$$\begin{array}{l}\mathcal{D}_\mathbb{K} = \cup_{p\in\mathbb{P}}\mathcal{D}_{\mathbb{K},p},\\ \mathcal{D}_{\mathbb{K},p} = \{\langle f,k,t\rangle \mid k\in\mathbb{K},\, \exists_{a\in\mathcal{A}}\colon f = \phi(p(a)), t = T(p,k,a)\}\end{array}$$

which gives us a single function $\Pi_{\mathbb{P}}$ and individual functions $\Pi_{\mathbb{K},p}$. This gives us

$$\Pi^{\text{sequential}}(f) = \langle \text{let } p := \Pi_{\mathbb{P}}(f), k := \Pi_{\mathbb{K}}(p(f)) \text{ or } \Pi_{\mathbb{K},p}(p(f))\rangle$$

This approach to classification is potentially the most accurate, since both the preconditioner and iterator recommendation are made based on the features of the actual problem they apply to. This also means that this approach is the most expensive; both the combined and the orthogonal approach require only the features of the original problem. In practice, with a larger number of preprocessors, one can combine these approaches. For instance, if a preprocessor such as scaling can be classified based on some easy to compute features, it can be tackled sequentially, while the preconditioner and iterator are then recommended with the combined approach based on a full feature computation of the scaled problem.

5.4. The reliability problem

In the reliability problem we classify problems by whether a method converges on them or not. The above approaches can not be used directly in this case, for several reasons. Instead, we take a slightly different approach. For each method m we define a function $\Pi^{(m)} : \mathbb{F} \mapsto \{0, 1\}$ which states whether the method will converge given a feature vector of a problem. We can then define

$$\Pi(f) = \text{a random element of } \{m \colon \Pi^{(m)}(f) = 1\}$$

The above strategies give only a fairly weak recommendation from the point of optimizing solve time. Rather than using reliability classification on its own, we can use it as a preliminary step before the performance classification.

6. Experiments

In this section we will report on the use of the techniques developed above, applied to the problem of recommending a preconditioner and iterative method for solving a linear system. The discussion on the experimental setup and results will be brief; results with much greater detail can be found in [10].

We start by introducing some further concepts that facilitate the numerical tests.

6.1. Experimental setup

We use a combination of released software from the Salsa project [16, 17] and custom scripts. For feature computation we use AnaMod [8, 19]; storage and analysis of features and timings is done with MySQL and custom scripting in Matlab and its statistical toolbox. The Salsa testbed gives us access to the iterative methods and preconditioners of the Petsc package [1, 2], including the preconditioners of externally interfaced packages such as Hypre [9, 12].

We use fairly standard statistical techniques to deal with features that can be invariant or (close to) dependent in the test problem collection; methods that can be very close in their behaviour; evaluation of the accuracy of the classifiers we develop.

Hierarchical classification It is quite conceivable that certain algorithms are very close in behaviour. It then makes sense to to group these methods together and first construct a classifier that can recommend first such a group, and subsequently a member of the group. This has the advantage that the classifiers are build from a larger number of observations, giving a higher reliability.

The algorithm classes are built by computing the independence of methods. For two algorithms x and y, the 'independence of method x from method y' is defined as

$$I_y(x) = \frac{\#\text{cases where } x \text{ works and } y \text{ not}}{\#\text{cases where } x \text{ works}}$$

The quantity $I_y(x) \in [0, 1]$ describes how much x succeeds on different problems from y. Note that $I_y(x) \neq I_x(y)$ in general; if x works for every problem where y works (but not the other way around), then $I_y(x) = 0$, and $I_x(y) > 0$.

Evaluation In order to evaluate a classifier, we use the concept of accuracy. The accuracy α of a classifier is defined as

$$\alpha = \frac{\#\text{problems correctly classified}}{\text{total } \#\text{problems}}$$

A further level of information can be obtained looking at the details of misclassification: a 'confusion matrix' is defined as $A = (\alpha_{ij})$ where α_{ij} is the ratio of problems belonging in class i, classified in class j, to those belonging in class i. With this, α_{ii} is the accuracy of classifier i, so, for an ideal classifier, A is a diagonal matrix with a diagonal $\equiv 1$; imperfect classifiers have more weight off the diagonal.

6.2. Numerical test

We tested a number of iterative methods and preconditioners on a body of test matrices, collected from Matrix Market and a few test applications. The iterative methods and preconditioners are from the PETSc library.

As described above, we introduced superclasses, as follows. The iterative method superclasses are:

- **B**={ *bcgs, bcgsl, bicg* }, where *bcgs* is BiCGstab [20], and *bcgsl* is BiCGstab(ℓ) [18] with $\ell \geq 2$.

- **G**={ *gmres, fgmres* } where *fgmres* is the 'flexible' variant of GMRES [15].
- **T**={ *tfqmr* }
- **C**={ *cgne* }, conjugate gradients on the noral equations.

and the preconditioner superclasses are:

- **A** = { *asm, rasm, bjacobi* }, where *asm* is the Additive Schwarz method, and *rasm* is its restricted variant [5]; *bjacobi* is block-jacobi with a local ILU solve.
- **BP** = { *boomeramg, parasails, pilut* }; these are three preconditioners from the *hypre* package [9, 12]
- **I** = { *ilu, silu* }, where *silu* is an ILU preconditioner with shift [13].

(a) Iterative Methods

ksp	$\alpha \pm z$
bcgsl	0.59±0.02
bcgs	0.71±0.03
bicg	0.68±0.06
fgmres	0.80±0.02
gmres	0.59±0.04
lgmres	0.81±0.03
tfqmr	0.61±0.05

(b) Preconditioners

pc	$\alpha \pm z$
asm	0.72±0.05
bjacobi	0.11±0.11
boomeramg	0.71±0.06
ilu	0.66±0.02
parasails	0.46±0.12
pilut	0.80±0.06
rasm	0.70±0.04
silu	0.83±0.02

Table 1. Accuracy of classification using one class per available method

In table 1 we report the accuracy (as defined above) for a classification of all individual methods, while table 2 gives the result using superclasses. Clearly, classification using superclasses is superior. All classifiers were based on decision trees [7, 4], constructed with the Matlab statistical toolbox; decision trees in general gave better results than Bayesian classifiers.

We can also report this by drawing up confusion matrices for two different classification strategies for the preconditioner / iterative method combination. The orthogonal approach gives superior results, as evinced by the lesser weight off the diagonal. For this approach, there are fewer classes to build classifiers for, so the modeling is more accurate. As a quantative measure of the confusion matrices, we report in table 3 the average and standard deviation of the fraction of correctly classified matrices.

(a) Iterative methods

Super	Class	α	Compound
B		0.95	
	bcgs	0.93	0.87
	bcgsl	0.92	0.87
	bicg	0.89	0.84
G		0.98	
	fgmres	0.96	0.94
	gmres	0.91	0.89
	lgmres	0.94	0.93
T		0.91	
	tfqmr	–	0.91

(b) Preconditioners

Super	Class	α	Compound
A		0.95	
	asm	0.98	0.93
	bjacobi	0.67	0.64
	rasm	0.82	0.78
BP		0.99	
	boomeramg	0.80	0.80
	parasails	0.78	0.77
	pilut	0.97	0.96
I		0.94	
	ilu	0.82	0.75
	silu	0.97	0.91

Table 2. Hierarchical classification results

strategy	average	std.dev.
combined	.3	.15
orthogonal	.78	.04

Table 3. Average and standard deviation of the correct classification rate

7. Conclusion

We have defined the relevant concepts for the use of machine learning for algorithm selection. Apart from the formalization itself, an innovative aspect of our discussion is the multi-leveled approach to the set of objects (the algorithms) to be classified. An important example of levels is the distinction between the iterative process and the preconditioner in iterative linear system solvers. We have defined various strategies for classifying subsequent levels. A numerical test testifies to the feasibility of using machine learning to begin with, as well as the necessity for our multi-leveled approach.

References

[1] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry 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. Birkhauser Press, 1997.

[2] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1999.

[3] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of machine learning to the selection of sparse linear solvers. *Int. J. High Perf. Comput. Appl.*, 2006. submitted.

[4] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *CART: Classification and Regression Trees*. 1983.

[5] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 21:792–797, 1999.

[6] Jack Dongarra, George Bosilca, Zizhong Chen, Victor Eijkhout, Graham E. Fagg, Erika Fuentes, Julien Langou, Piotr Luszczek, Jelena Pjesivac-Grbovic, Keith Seymour, Haihang You, and Satish S. Vadiyar. Self adapting numerical software (SANS) effort. *IBM J. of R.& D.*, 50:223–238, June 2006. `http://www.research.ibm.com/journal/rd50-23.html`, also UT-CS-05-554 University of Tennessee, Computer Science Department.

[7] M.H. Dunham. *Data Mining: Introductory and Advanced Topics*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2002.

[8] Victor Eijkhout and Erika Fuentes. A standard and software for numerical metadata. Technical Report TR-07-01, Texas Advanced Computing Center, The University of Texas at Austin, 2007. to appear in ACM TOMS.

[9] R.D. Falgout, J.E. Jones, and U.M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers, A.M. Bruaset and A. Tveito, eds.*, volume 51, pages 267–294. Springer-Verlag, 2006. UCRL-JRNL-205459.

[10] Erika Fuentes. *Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems*. PhD thesis, University of Tennessee, Knoxville TN, USA, 2007.

[11] E.N. Houstis, V.S. Verykios, A.C. Catlin, N. Ramakrishnan, and J.R. Rice. PYTHIA II: A knowledge/database system for testing and recommending scientific software, 2000.

[12] Lawrence Livermore Lab, CASC group. Scalable Linear Solvers. `http://www.llnl.gov/CASC/linear_solvers/`.

[13] T.A. Manteuffel. An incomplete factorization technique for positive definite linear systems. *Math. Comp.*, 34:473–497, 1980.

[14] Naren Ramakrishnan and Calvin J. Ribbens. Mining and visualizing recommendation spaces for elliptic PDEs with continuous attributes. *ACM Trans. Math. Software*, 26:254–273, 2000.

[15] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Stat. Comput.*, 14:461–469, 1993.

[16] Salsa Project. SALSA: Self-Adapting Large-scale Solver Architecture. `http://icl.cs.utk.edu/salsa/`.

[17] Salsa Project. SourceForge: Self-Adapting Large-scale Solver Architecture. `http://sourceforge.net/projects/salsa/`.

[18] G.L.G. Sleijpen, H.A. Van der Vorst, and D.R. Fokkema. Bicgstab(ℓ) and other hybrid Bi-cg methods. submitted.

[19] The Salsa Project. AnaMod software page. `http://icl.cs.utk.edu/salsa/software`.

[20] Henk van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[21] David C. Wilson, David B. Leake, and Randall Bramley. Case-based recommender components for scientific problem-solving environments. In *Proceedings of the Sixteenth International Association for Mathematics and Computers in Simulation World Congress*, 2000.