TACC Technical Report TR-08-01

Software architecture of an intelligent recommender system

Victor Eijkhout
Texas Advanced Computing Center,
The University of Texas at Austin
eijkhout@tacc.utexas.edu

Erika Fuentes
Innovative Computing Laboratory
University of Tennessee
efuentes@cs.utk.edu

January 2008

This work was funded in part by the Los Alamos Computer Science Institute through the subcontract # R71700J-29200099 from Rice University, and by the National Science Foundation under grants 0203984 and 0406403.

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 this paper we give a formal description of a recommender system for the runtime selection of algorithms. We give a functional description of the system, both in an abstract mathematical framework, and as a proposed software API. Parts of this API have been implemented in released libraries.

Our exposition of recommender systems is more detailed than previous efforts. In particular, we consider algorithms to be composite objects, and 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 [1, **?**, 7, 16, 12, 27]. Such methods perform mining of the problem features, typically relying on a database of prior knowledge, in order to recommend a suitable method. However, we may characterize this earlier research as either being too general and lacking in specific implementations, or being too limited to one area, lacking the conceptual necessary for generalization to other areas.

In this paper we present a theoretical framework for such intelligent systems, as well as libraries that we have developed that embody these concepts. Our framework is more general than what has been studied before, 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.

In addition to the abstract mathematical description, we propose software interfaces implementing the various entities in the framework. Some of this software has been realized in our Salsa (Self-Adapting large-scale Solver Architecture) project; see [19, 20].

We start by giving a formal definition of numerical problems in section 2 and of numerical methods in section 3. Section 4 has a general discussion of intelligent method selection, while section 5 details strategies for recommending composite method objects.

2 Numerical problems and their solution

In this section we formalize the notion of numerical problems and numerical methods for solving problems. The definition of a numerical problem will involve introducing the concept of problem features. Our formalization of numerical methods will consider methods to be composite objects.

2.1 Numerical Problems

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.

It would be pointless to capture this diversity of problems in a rigorous definition, so we will use a leave the concept of 'numerical problem' largely undefined. We denote the space of numerical problems by

$\mathbb{A}$: the set of numerical problems in a class

where the exact definition of this will depend on the problem area.

The implementation of numerical problems, in contrast to that of concepts in the following sections, is similarly domain-specific, so we will give some example definitions here which are based on – but not necessarily sufficient for – PDE problems. Further definitions in this paper will be in terms of the objects defined here.

In the area of linear system solving, the following is a reasonable definition of a numerical problem[1]:

```
struct Problem_ {
    LinearOperator A;
    Vector RHS,KnownSolution,InitialGuess;
    DesiredAccuracy constraint;
};
typedef struct Problem_* Problem;
```

2.2 Results Space

Corresponding to the problem space, there is a result space

$\mathbb{R} = \mathbb{S} \times \mathbb{T}$: results, the space of solutions plus performance measurements (1)

1. After this example we will tacitly omit the `typedef` line and only give the structure definition.

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.

To every problem $A \in \mathbb{A}$ there corresponds a theoretical solution $\bar{\sigma}(A) \in \mathbb{S}$, written $\bar{\sigma}$ for short when there is no ambiguity. A method is deemed to have solved a problem if the computed solution s is close enough to σ.

As with numerical problems, the definition of a solution and performance measurement are context-dependent. The following is an illustration for the case of linear system solving. In the definition of a result we include both the numerical solution and metrics of both performance and numerical accuracy:

```
struct PerformanceMeasurement_ {
    int success;
    double Tsetup,Tsolve;
    double *ConvergenceHistory;
    double BackwardError,ForwardError;
}
struct Result_ {
    Vector ComputedSolution;
    PerformanceMeasurement performance;
}
```

2.3 Method Space

Having defined problems and results, we define the space of methods as the mappings from problems to results.

$\mathbb{M} = \{\mathbb{A} \mapsto \mathbb{R}\}$: the space of methods (potentially) solving the class of numerical problems

Note that methods only purport to solve the problem. In principle we could allow for methods to solve a problem inaccurately or to fail to solve the problem. Reasons for failure can be the divergence of an iterative method, or exceeding a set runtime limit. However, in this paper we only consider two cases: let $M \in \mathbb{M}$ be a method defined on the problem space, then

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

We assume that the performance measurements introduced above always the runtime of the method. We formalize this as a timing function $T(A, M)$ 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 will defer further discussion of the method space $\mathbb{M}$ for section 3.

2.4 Features and features extraction

The problem space $\mathbb{A}$ defined above is, even in a specific context, too amorphous to allow formal algorithm selection, so we introduce the concept of problem features, which will be the basis for any recommendation strategy. We reduce a numerical problem A to a vector in k-dimensional *feature space*:

$\mathbb{F}$: the space of feature vectors of the numerical problems

We posit the existence of feature extraction function:

$\Phi : \mathbb{A} \mapsto \mathbb{F}$: a function that extracts features of numerical problems

where $\Phi(A) = \bar{x} \in \mathbb{F}$ is the feature vector describing the numerical problem.

Feature space can be considerably more complicated in structure than $\mathbf{R}^k$: elements of $\bar{x}$ 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.

Because feature space harbours such unhomogeneous elements, it makes sense to subdivide the feature space in separate dimensions

$$\mathbb{F} = \mathbb{F}_1 \times \cdots \mathbb{F}_k\text{: the categories in feature space} \tag{2}$$

Accordingly, we also split up the feature extraction function into component feature functions:

$\Phi = \langle \Phi_, \ldots \Phi_k \rangle$, $\Phi_i \colon \mathbb{A} \mapsto \mathbb{F}_i$: individual feature calculations

From the above observed heterogeneity of feature space it follows that no general implementation can easily be given either. Instead, we describe the implementation functionally through the API that sets and queries feature vector elements.

The following API serves as a basis for manipulating feature vectors:

```
NewFeaturesObject(Features *metadata);
AddFeature(Features metadata,
    char *name,MetaDataValue value,MetaDatatype type);
GetFeature(Features metadata,
    char *name,MetaDataValue *value);
```

The NMD (Numerical MetaData) library [9] implements this API, including various utility functions.

We adopt the modular approach to feature vectors for our implementation of the feature extraction functions Φ_i. We posit the existence of a parametrized function that can compute one feature at a time: spaces $\mathbb{F}_i$.

```
ComputeQuantity(Problem problem,
  char *feature,ReturnValue *result,TruthValue *success);
```

Here, the `ReturnValue` type is a union of all returnable types, and the `success` parameter indicates whether the quantity was actually computed. Computation can fail for any number of reasons: if sequential software is called in parallel, if properties such as matrix bandwidth are asked of an implicitly given operator, et cetera. Failure is not considered catastrophic, and the calling code should allow for this eventuality.

For a general and modular setup, we do not hardwire the existence of any module. Instead, we use a function

```
DeclareModule(char *feature,ReturnType type,
    void(*module)(Problem,char*,ReturnValue*,TruthValue*));
```

to add individual feature computations to a runtime system.

The AnaMod (Analysis Modules) library [9] implements the above ideas, though in a slightly different form. Rather than considering feature space as a simple product of one-dimensional spaces, we make the working assumption that feature space has a two-level structure of 'categories' and 'elements' inside each category. This structure is both appropriate from a logical, and from a software point of view. Logically, a linear system (or any other linear algebra problem) can have structural properties, relating to the sparsity of the system, norm-like properties, spectral properties, et cetera. In software, this division is apparent in that each category can often be computed as a whole, by one piece of software. Indeed, various elements in a category can often be computed together at the same cost as each individually. Consider for instance the norms of the symmetric and anti-symmetric part of the matrix. The AnaMod library has provisions for taking this into account.

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

$$\boxed{\mathbb{P} = \{\mathbb{A} \mapsto \mathbb{A}\}\text{: the set of all mappings from problems into problems}}$$

and the solvers[2]

$$\boxed{\mathbb{K} = \{\mathbb{A} \mapsto \mathbb{R}\}\text{: the set of all solvers}}$$

To illustrate the fact that the preprocessing stages are really mappings $\mathbb{A} \mapsto \mathbb{A}$, consider a right scaling D of a linear system, which maps the problem/solution tuple $\langle A, b, \bar{\sigma}\rangle$ to $\langle AD, b, D^{-1}\sigma\rangle$.

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:

$$m \in \mathbb{M}\colon m = k \circ p_n \circ \cdots \circ p_1, \quad k \in \mathbb{K}, p_i \in \mathbb{P}_i$$

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

2. In the context of linear system solving, these will be Krylov methods, hence the choice of the letter 'K'.

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.

A preprocessed problem $p(A)$ will in general have a different solution from the original problem. As remarked above, a right scaled system $\langle AD, b\rangle$ has a solution $D^{-1}\sigma$ where $\sigma = \sigma(\langle A, b\rangle)$. Thus, the solution needs to be backtransformed. We extend the definition of $\mathbb{P}$ to allow the notation

$$\sigma(A) = p^{-1}(\sigma(p(A))), \qquad A \in \mathbb{A}, p \in \mathbb{P}$$

or

$$\begin{array}{ccc} A & \stackrel{\text{solution}}{\longrightarrow} & x = \sigma(A) \\ p\downarrow & & \uparrow p^{-1} \\ p(A) & \stackrel{\text{solution}}{\longrightarrow} & p(x) = \sigma(p(A)) \end{array}$$

3.2 Implementation

The set of system preprocessors, like that of the analysis modules above, has a two level structure. First, there is the preprocessor type; for instance 'scaling'. Then there is the specific choice within the type; for instance ''left scaling'. Additionally, but not discussed here, there can be parameters associated with either the type or the specific choice; for instance, we can scale by a block diagonal, with the parameter indicating the size of the diagonal blocks.

We implement the sequence of preprocessors by a recursive routine:

```
PreprocessedSolving
    (char *method,Problem problem,Result *solution)
{
  ApplyPreprocessor(problem,&preprocessed_problem);
  if ( /* more preprocessors */ )
    PreprocessedSolving(next_method,
      preprocessed_problem,&preprocessed_solution);
  else
    Solve(final_method,
      preprocessed_problem,&preprocessed_solution);
  UnApplyPreprocessor(preprocessed_solution,solution);
}
```

The actual implementation is more complicated, but this pseudo-code conveys the essence.

We again adopt a modular approach where preprocessors are dynamically declared:

```
DeclarePreprocessor(char *type,char *choice,
    void(*preprocessor)(Problem,Problem*));
```

The SysPro (System Preprocessor) library provides a number of preprocessors, including the forward and backward transformation of the systems. It also includes a framework for looping over the various choices of a preprocessor type, for instance for an exhaustive test.

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:

$$\Pi(A) = M \qquad \equiv \qquad \forall_{M' \in \mathbb{M}} \colon T(A, M) \leq T(A, M') \tag{3}$$

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

$$\Pi'(A) = M \qquad \text{where } M \text{ is any method such that} \qquad T(A, M) < \infty$$

Also, we usually do not insist on the absolutely fastest method: we can relax equation (3) to

$$\Pi(A) = M \qquad \equiv \qquad \forall_{M' \in \mathbb{M}} \colon T(A, M) \leq (1 - \epsilon) T(A, M') \tag{4}$$

which, for sufficient values of ϵ, also makes the definition non-unique. In both of the previous cases we do not bother to define Π as a multi-valued function, but implicitly interpret $\Pi(A) = M$ to mean 'M is one possible method satisfying the selection criterion'.

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

The problem space $\mathbb{A}$ is too amorphous to deal with, so we define Π as a function on the feature space $\mathbb{F}$ instead:

$$\boxed{\Pi\colon \mathbb{F} \mapsto \mathbb{M}\text{: the function that picks the best solution method}} \tag{5}$$

The method selection problem then becomes

$$\Pi(\bar{x}) = M \quad \text{if there is a problem } A \text{ such that } \Phi(A) = \bar{x} \text{ and } M \text{ is such that } \forall_{M' \in \mathbb{M}}\colon T(A,M) \leq T(A,M'). \tag{6}$$

Finding a method, given a problem, now becomes the function

$$\Pi \circ \Phi\colon \mathbb{A} \mapsto \mathbb{M}$$

Mindful of the above discussion of preprocessors and solvers, we refine the definition of Π and introduce

$$\boxed{\Pi_i\colon \mathbb{F} \mapsto \mathbb{P}_i\text{: the selector function for preprocessor } i} \tag{7}$$

and

$$\boxed{\Pi_k\colon \mathbb{F} \mapsto \mathbb{K}\text{: the method selector}}$$

This also requires us to refine the method selection process. For the moment we merely note two approaches; we will discuss this topic further in section 5. Technically the most correct way of choosing preprocessors and a solver for a problem A is:

$$m = k \circ p_n \circ \cdots \circ p_1,$$

where

$$A_1 = A, \quad \forall_i\colon p_i = \Pi_i(A_i),\ A_{i+1} = p_i(A_i), \quad k = \Pi_k(A_{n+1})$$

A simpler procedure, though conceptually not optimal, is described by:

$$\forall_i\colon p_i = \Pi_i(A), \quad k = \Pi_k(A).$$

We will elaborate this point below.

4.2.1 Consistency

In equation (6) the definition was dependent on finding a problem A with the desired features. This raises the question: what if there are two problems with the same features. Such a situation indeed occurs in practice, and it is the reason that we can not directly hope to predict runtimes, and base our function Π on such runtime prediction.

For instance, a sequence of Poisson problems derived from increasingly refined discretizations of the same domain will have a number of identical, very similar, features relating to norms and specture, but the solution time goes up with the matrix size. Conversely, multiplying a linear problem by a scalar will not influence the solution time, provided the solver uses a relative stopping test, at all.

For this reason we regularly normalize features to make problems scaling-invariant.

We also see that our analysis function Φ has to be sufficiently detailed. Suppose problems A_1 and A_2 are such that $\Phi(A_1) = \Phi(A_2) = \bar{x}$, that is, they have approximately the same features. Let $M = \Pi(\bar{x})$. We require that for consistency both

$$T(A_1, M) = \min_{M'} T(A_1, M') \qquad \text{and} \qquad T(A_2, M) = \min_{M'} T(A_2, M')$$

that is, for two problems with the same features, the same method should be optimal. Note that we are not requiring that the runtimes are actually the same ($T(A_1, M) = T(A_2, M)$), for reasons outlined above.

4.2.2 Examples

Let us consider some adaptive systems, and the shape that $\mathbb{F}$, $\mathbb{M}$, and Π take in them.

Atlas Atlas [26] is a system that determines the optimal implementation of Blas kernels such as matrix-matrix multiplication. One could say that the implementation chosen by Atlas is independent of the inputs[3] and only depends on the platform, which we will consider a constant in this discussion. Essentially, this means that $\mathbb{F}$ is an empty space. The number of dimensions of $\mathbb{M}$ is fairly low, consisting of algorithm parameters such unrolling, blocking, and software pipelining parameters.

In this case, Π is a constant function defined by

$$\Pi(f) \equiv \min_{M \in \mathbb{M}} T(A, M)$$

where A is a representative problem. This minimum value can be found by a sequence of line searches, as done in Atlas, or using other minimization techniques such a modified simplex method [28].

3. There are some very minor caveats for special cases, such as small or 'skinny' matrices.

Scalapack/LFC The distributed dense linear algebra library Scalapack [6] gives in its manual a formula for execution time as a function of problem size N, the number of processors N_p, and the block size N_b. This is an example of a two-dimensional feature space (N, N_p), and a one-dimensional method space: the choice of N_b. All dimensions range through positive integer values. The function involves architectural parameters (speed, latency, bandwidth) that can be fitted to observations.

Unfortunately, this story is too simple. The LFC software [17] takes into account the fact that certain values of N_p are disadvantageous, since they can only give grids with bad aspect ratios. A prime number value of N_p is a clear example, as this gives a degenerate grid. In such a case it is often better to ignore one of the available processors and use the remaining ones in a better shaped grid. This means that our method space becomes two-dimensional with the addition of the actually used number of processors. This has a complicated dependence on the number of available processors, and this dependence can very well only be determined by exhaustive trials of all possibilities.

Collective communication Collective operations in MPI [22] can be optimized by various means. In work by Vadhyar *et al.* [24], the problem is characterized by the message size and the number of processors, which makes $\mathbb{F}$ have dimension 2. The degrees of freedom in $\mathbb{M}$ are the segment size in which messages will be subdivided, and the 'virtual topology' algorithm to be used.

Assume for now that the method components can be set independently. The segment size is then computed as $s = \Pi_s(m, p)$. If we have an *a priori* form for this function, for instance $\Pi_s(m, p) = \alpha + \beta m + \gamma p$, we can determine the parameters by a least squares fit to some observations.

Suppose the virtual topology depends only on the message size m. Since for the virtual topology there is only a finite number of choices, we only need to find the crossover points, which can be done by bisection. If the topology depends on both m and p, we need to find the areas in (m, p) space, which can again be done by some form of bisection.

4.3 Intelligent method selection

We can now introduce our *Intelligent Solver*, which we describe using a lambda-notation, in terms of previously defined operators:

$$Q = \lambda(A)\Pi(\Phi(A))(A)$$

(this describes extracting problem features, choosing a method accordingly, and solving the problem using this method.)

From the definitions of Π and Φ we find that $Q \in \mathbb{M}$. In terms of software, this implies that the Intelligent Solver can immediately be substituted for any solver that is currently used in a code, since they both have the same input/output specification.

Similarly we use the selection functions for the preprocessors to define intelligent preprocessors

$$Q_i = \lambda(A)\Pi_i(\Phi(A))(A)$$

which are indeed themselves in the set of preprocessors: $Q_i \in \mathbb{P}$.

4.4 Constructing the Π function

Above we only stated that intelligent functions $\Pi\colon \mathbb{F} \mapsto \mathbb{M}$ (equation (5)) and $\Pi_i\colon \mathbb{F} \mapsto \mathbb{P}_i$ (equation (7)) exist. In this section we will examime the process of constructing Π, Π_i in detail[4]. In some cases, for instance with decision tree classifiers, these functions can be constructed directly. In other cases, such as with Bayesian classification, a characterization of the behaviour of individual methods is used.

4.4.1 General construction

The function Π is constructed from a database of performance results that results from solving a set of problems $\mathbf{A} \subset \mathbb{A}$ by each of a collection of methods $\mathbf{M} \subset \mathbb{M}$, each combination yielding a result $r \in \mathbb{R}$ (equation (1)). Thus we store features of the problem, an identifier of the method used, and the resulting performance measurement:

$\mathcal{D} \subset \mathbb{D} = \{\mathbb{F} \times \mathbb{M} \rightarrow \mathbb{T}\}$: the database of features and performance results of solved problems

If we had access to the whole set $\mathbb{D}$, we could implement the function Π as[5]

$$\Pi(F) = \arg\min_M \mathbb{D}(F, M)$$

that is,

$$\Pi(F) = M \quad \equiv \quad \forall_{M' \neq M}\colon \mathbb{D}(F, M) < \mathbb{D}(F, M').$$

However, in practice $\mathbb{D}$ may not be filled in for most values of f: $\neg\exists_{m,t}\colon \langle f, m, t\rangle \in \mathcal{D}$.

Instead, we simply posit a mechanism (which differs per classifcation strategy) that constructs Π from a database $\mathcal{D}$. Where needed we will express the dependency of Π on the database explicitly as $\Pi_{\mathcal{D}}$.

4. The reader note that we actually use two definitions of Π: at first this described the ideal function that picks the optimal method. When we talk of 'constructing' Π, the function being constructed is an approximation to this ideal function, and various accuracy measurements apply to gauge the result of this process.

5. We take '$\arg\min_m g(m)$' to mean 'the value of m that minimizes $g(m)$'.

4.4.2 Construction through method suitability

In methods like Bayesian classification, we take an approach to constructing Π where we characterize each method individually, and let Π be the function that picks the most suitable one.

Starting with the database $\mathcal{D}$ as defined above, We note for each problem – and thus for each feature vector – which method was the most successful:

$$\mathcal{D}'\colon \mathbb{F}\times\mathbb{M}\to\{0,1\} \quad \text{defined by} \quad \mathcal{D}'(f,m)=1 \;\equiv\; m=\arg\min_m \mathcal{D}(f,m)$$

This allows us to draw up indicator functions for each method[6]:

$$\mathbb{B}'\colon \mathbb{M}\to \mathrm{pow}(\mathbb{F}) \quad \text{defined by} \quad f\in\mathbb{B}'(m)\Leftrightarrow \mathbb{D}'(f,m)=1 \tag{8}$$

These functions are generalized (multi-dimensional) histograms: for each method they plot the feature (vector) values of sample problems for which this method is was found to be optimal. However, since these functions are constructed from a set of experiments we have that, most likely, $\cup_{m\in\mathbb{M}}\mathbb{B}'(m)\subsetneqq\mathbb{F}$. Therefore, it is not possible to define

$$\Pi(f)=m \quad\equiv\quad f\in\mathbb{B}'(m),$$

since for many values of f, the feature vector may not be in *any* $\mathbb{B}'(m)$. Conversely, it could also be in $\mathbb{B}'(m)$ for several values of m, so the definition is not well posed.

Instead, we use a more general mechanism. First we define suitability functions:[7]

$$\boxed{\mathbb{S}=\{\mathbb{F}\to[0,1]\}\text{: the space of suitability measurements of feature vectors}} \tag{9}$$

This is to be interpreted as follows. For each numerical method there will be one function $\sigma\in\mathbb{S}$, and $\sigma(f)=0$ means that the method is entirely unsuitable for problems with feature vector f, while $\sigma(f)=1$ means that the method is eminently suitable.

We formally associate suitability functions with numerical methods:

$$\boxed{\mathbb{B}\colon \mathbb{M}\to\mathbb{S}\text{: the method suitability function}} \tag{10}$$

and use the function $\mathbb{B}$ to define the selection function:

$$\Pi(f)=\arg\max_m \mathbb{B}(m)(f). \tag{11}$$

6. There is actually no objection to having $\mathcal{D}(f,m)$ return 1 for more than one method m; this allows us to equivocate methods that are within a few percent of each other's performance. Formally we do this by extending and redefining the $\arg\min$ function.
7. Please ignore the fact that the symbol $\mathbb{S}$ already had a meaning, higher up in this story.

Since elements of $\mathbb{S}$ are defined on the whole space $\mathbb{F}$, this is a well-posed definition.

Remark. Equations (10) and (11) are of course heavily inspired by the Bayesian classification algorithm.

The remaining question is how to construct the suitability functions. For this we need constructor functions

$$\boxed{\mathbb{C}\colon P(\mathbb{F}) \to \mathbb{S}\text{: classifier constructor functions}} \tag{12}$$

The mechanism of these can be any of a number of standard statistical techniques, such as fitting a Gaussian distribution through the points in the subset $\cup_{f\in F} f$ where $F \in P(\mathbb{F})$.

Clearly, now $\mathbb{B} = \mathbb{C}\circ\mathbb{B}'$, and with the function $\mathbb{B}$ defined we can construct the selection function Π as in equation (11).

4.5 Applying the intelligence

Strictly speaking, the above is a sufficient description of the construction and use of the Π functions. However, as defined above, they use the entire feature vector of a problem, and in practice a limited set of features may suffice for any given decision. This is clear in such cases as when we want to impose "This method only works for symmetric problems", where clearly only a single feature is needed. Computing a full feature vector in this case would be wasteful. Therefore, we introduce the notation $\mathbb{F}_I$ where $I \subset \{1,\ldots,k\}$. This corresponds to the subspace of those vectors in $\mathbb{F}$ that are null in the dimensions not in I.

We will also assume that for each method $m \in \mathbb{M}$ there is a feature set I_m that suffices to evaluate the suitability function $\mathbb{B}(m)$. Often, such a feature set will be common for all methods in a set $\mathbb{P}_i$ of preprocessors. For instance, graph algorithms for fill-in reduction only need structural information on the matrix.

Here is an example of the API (as used in the Salsa system) that defines and uses feature sets. First we define a subset of features:

```
FeatureSet symmetry;
NewFeatureSet(&symmetry);
AddToFeatureSet(symmetry,
    "simple","norm-of-symm-part",&sidx);
AddToFeatureSet(symmetry,
    "simple","norm-of-asymm-part",&aidx);
```

After problem features have been computed, a suitability function for a specific method can then obtain the feature values and use them:

```
FeatureSet symmetry; // created above
FeatureValues values;
NewFeatureValues(&values);
InstantiateFeatureSet(problem,symmetry,values);
GetFeatureValue(values,sidx,&sn,&f1);
GetFeatureValue(values,aidx,&an,&f2);
if (f1 && f2 && an.r>1.e-12*sn.r)
   printf("problem too unsymmetric\n");
```

5 Classification of composite methods

In the previous section, we posited a general mechanism (which we described in detail for methods like Bayesian classification) of deriving $\Pi_{\mathcal{D}}$ from a database $\mathcal{D} \subset \{\mathbb{F} \times \mathbb{M} \rightarrow \mathbb{T}\}$. 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 on the suitability functions of equation (9): we then assume 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. In terms of the functions introduced in section 4.4.2 this would be $\sigma_{\mathcal{A}} = \mathbb{C} \circ \Phi(\mathcal{A})$, where for the specific case of Bayesian classification

$$\sigma(\bar{x}) = C \exp((\bar{x}-\mu)^t \Sigma^{-1} (\bar{x}-\mu)).$$

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 $\Pi^{\text{combined}}(f) = \langle p, k\rangle$.

For Bayesian classification we get for each $\langle p, k\rangle \in \mathbb{M}$ the class

$$C_{p,k} = \{A \in \mathcal{A}\colon T(p,k,A) \text{ is minimal}\}$$

and corresponding function $\sigma_{p,k}$. We can then define

$$\Pi^{\text{combined}}(f) = \arg\max_{p,k} \sigma_{p,k}(f)$$

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.

In an alternative derivation of this approach, we consider the $C_{p,k}$ to be classes of preprocessors, but conditional upon the choice of a solver. We then recommend p and k, not as a pair but sequential: we first find the k for which the best p can be found:

$$\Pi^{\text{conditional}} = \begin{cases} \text{let } k := \arg\max_k \max_p \sigma_{p,k}(f) \\ \text{return } \langle \arg\max_p \sigma_{p,k}(f), k\rangle \end{cases}$$

However, this is equivalent to the above combined approach.

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

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

In Bayesian classification, the classes here are

$$C_k = \{A \colon \min_p T(p,k,A) \text{ is minimal over all } k\}$$

and

$$C_p = \{A \colon \min_k T(p,k,A) \text{ is minimal over all } p\},$$

giving functions σ_p, σ_k. (Instead of classifying by minimum over the other method component, we could also use the average value.) The recommendation function is then

$$\Pi^{\text{orthogonal}}(f) = \langle \arg\max_p \sigma_p(f), \arg\max_k \sigma_k(f) \rangle$$

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

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

For Bayesian classification, we define the classes C_p as above:

$$C_p = \{A \colon \min_k T(p,k,A) \text{ is minimal over all } p\},$$

but we have to express that C_k contains preconditioned features:

$$C_k = \{A \in \bigcup_p p(\mathcal{A}) \colon T(p,k,A) \text{ is minimal, where } p \text{ is such that } A \in p(\mathcal{A})\}$$

Now we can define

$$\Pi^{\text{sequential}}(f) = \langle \text{let } p := \arg\max_p \sigma_p(f), k := \arg\max_k \sigma_k(p(f)) \rangle$$

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.

- The above approaches are based on assigning each problem to a single classes based on minimum solution time. In the reliability problem each problem would be assigned to multiple classes, since typically more than one method would converge on the problem. The resulting overlapping classes would lead to a low quality of recommendation.
- The sequential and orthogonal approaches would run into the additional problem that, given a preconditioner, there is usually at least one iterative method that gives a converging combination. Separate recommendation of the preconditioner is therefore impossible.

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

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

For Bayesian classification, we can adopt the following strategy. For each $M \in \mathbb{M}$, define the set of problems on which it converges:

$$C_M = \{A \colon T(M, A) < \infty\}$$

and let $\bar{C}_M$ be its complement:

$$\bar{C}_M = \{A \colon T(M,A) = \infty\}.$$

Now we construct functions $\sigma_M, \bar{\sigma}_M$ based on both these sets. This gives a recommendation function:

$$\Pi(f) = \{M \colon \sigma_M(f) > \bar{\sigma}_M(f)\}$$

This function is multi-valued, so we can either pick an arbitrary element from $\Pi(f)$, or the element for which the excess $\sigma_M(f) - \bar{\sigma}_M(f)$ is maximized.

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 [11].

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 [19, 20] and custom scripts. For feature computation we use AnaMod [9, 23]; storage and analysis of features and timings is done with MySQL and custom scripting in Matlab and its statistical toolbox.

The AnaMod package 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.

The Salsa testbed gives us access to the iterative methods and preconditioners of the Petsc package [2, 3], including the preconditioners of externally interfaced packages such as Hypre [10, 14].

6.2 Practical issues

The ideas developed in the previous sections are sufficient in principle for setting up a practical application of machine learning to numerical method selection. However, in practice we need some auxiliary mechanisms to deal with various ramifications of the fact that our set of test problems is not of infinite size. Thus, we need

- A way of dealing with features that can be invariant or (close to) dependent in the test problem collection.
- A way of dealing with methods that can be very close in their behaviour.
- An evaluation of the accuracy of the classifiers we develop.

6.2.1 Feature analysis

There are various transformations we apply to problem features before using them in various learning methods.

Scaling Certain transformations on a test problem can affect the problem features, without affecting the behaviour of methods, or being of relevance for the method choice. For instance, scaling a linear system by a scalar factor does not influence the convergence behaviour of iterative solvers. For this reason, we normalize features, for instance scaling them by the largest diagonal element. We also mean-center features for classification methods that require this.

Elimination Depending on the collection of test problems, a feature may be invariant, or dependent on other features. We apply Principal Component Analysis [13] to the set of features, and use that to weed out irrelevant features.

6.2.2 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 $I_x(y) > 0$.

6.2.3 Evaluation

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

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

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.

6.3 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:

- **B**={ *bcgs, bcgsl, bicg* }, where *bcgs* is BiCGstab [25], and *bcgsl* is BiCGstab(ℓ) [21] with $\ell \geq 2$.
- **G**={ *gmres, fgmres* } where *fgmres* is the 'flexible' variant of GMRES [18].
- **T**={ *tfqmr* }
- **C**={ *cgne* }, conjugate gradients on the noral equations.

for iterative methods and

- **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 [10, 14]
- **I** = { *ilu, silu* }, where *silu* is an ILU preconditioner with shift [15].

for preconditioners.

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 [8, 4].

Finally, in figures 1, 2 we give 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.

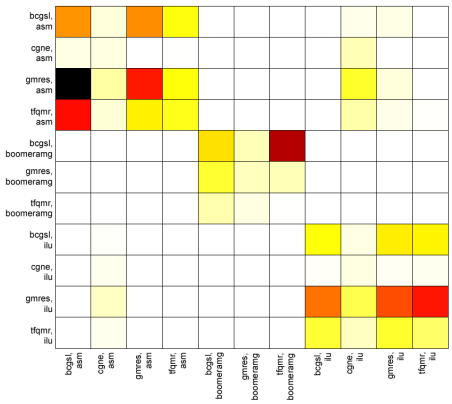

Figure 1: Confusion matrix for combined approach for classifying (pc, ksp).

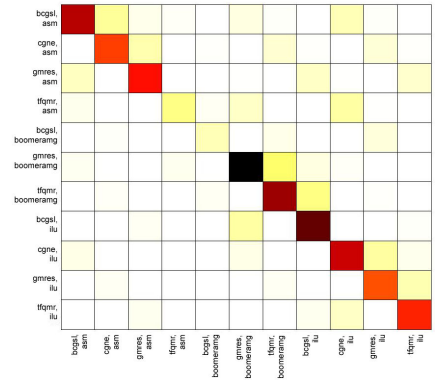

Figure 2: Confusion matrix for orthogonal approach for classifying (pc, ksp).

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

(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 2: Hierarchical classification results

(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

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.

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.

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

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

References

[1] D.C. Arnold, S. Blackford, J. Dongarra, V. Eijkhout, and T. Xu. Seamless access to adaptive solver algorithms. In M. Bubak, J. Moscinski, and M. Noga, editors, *SGI Users' Conference*, pages 23–30. Academic Computer Center CYFRONET, October 2000.

[2] 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.

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

[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] Yaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the fourth symposium on the frontiers of massively parallel computation (Frontiers '92), McLean, Virginia, Oct 19–21, 1992*, pages 120–127, 1992.

[7] 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.

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

[9] 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. submitted to ACM TOMS.

[10] 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 Solu-*

tion 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.

[11] Erika Fuentes. *Statistical and Machine Learning Techniques Applied to Algorithm Selection for Solving Sparse Linear Systems.* PhD thesis, 2007.

[12] 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.

[13] J.E. Jackson. *A User's Guide to Principal Components.* Wiley-IEEE, 2003.

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

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

[16] 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.

[17] Kenneth J. Roche and Jack J. Dongarra. Deploying parallel numerical library routines to cluster computing in a self adapting fashion, 2002. Submitted.

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

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

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

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

[22] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, Volume 1, The MPI-1 Core.* MIT Press, second edition edition, 1998.

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

[24] Sathish S. Vadhiyar, Graham E. Fagg, and Jack J. Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3. IEEE Computer Society, 2000.

[25] 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.

[26] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University

of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (www.netlib.org/lapack/lawns/lawn147.ps).

[27] 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.

[28] Qing Yi, Ken Kennedy, Haihang You, Keith Seymour, and Jack Dongarra. Automatic Blocking of QR and LU Factorizations for Locality. In *2nd ACM SIGPLAN Workshop on Memory System Performance (MSP 2004)*, 2004.