Self-Adapting Numerical Software and Automatic Tuning of Heuristics*

Jack Dongarra and Victor Eijkhout

Innovative Computing Laboratory, University of Tennessee, Knoxville TN 37996, USA

Abstract. Self-Adapting Numerical Software (SANS) systems aim to bridge the knowledge gap that exists between the expertise of domain scientists, and the know-how that is needed to fulfill efficiently their computational demands. This know-how extends to algorith choice, computational grid utilization, and use of properly optimized kernels. A SANS system is a piece of meta software that mediates between the application program and the computational platform so that application scientists – with disparate levels of knowledge of algorithmic and programmatic complexities of the underlying numerical software – can easily realize numerical solvers and efficiently solve their problem.
The main component of a SANS system is an Intelligent Agent that automates method selection based on data, algorithm and system attributes. The IA uses heuristics to make its decisions. In this paper we explain how the heuristics of the IA can be tuned over time by redundant testing and using the nature of many applications.

1 Introduction

In numerous technologically important areas, such as aerodynamics (vehicle design), electrodynamics (semiconductor device design), magnetohydrodynamics (fusion energy device design), and porous media (petroleum recovery), production runs on expensive, high-end systems last for hours or days, and a major portion of the execution time is usually spent inside of numerical routines, such as for the solution of large-scale nonlinear and linear systems that derive from discretized systems of nonlinear partial differential equations. These numerical parts of the code can contain a large number of tuning parameters, the choice of which greatly influences the efficiency of the total code, or can even make the difference between obtaining a solution and obtaining none.

Such numerical concerns, however, are *artifactual* from the perspective of the scientific and engineering users, who are usually more concerned with modeling and discretization issues. The classic response to numerics was to encode the requisite mathematical, algorithmic and programming expertise into libraries that could be easily reused by a broad spectrum of domain scientists. However, in high-performance computing this solution is no longer sufficient. There is typically more than one algorithm for the stated purpose, and since several levels of

* This work was funded in part by the Los Alamos Computer Science Institute through the subcontract # R71700J-29200099 from Rice University.

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2660, pp. 759–767, 2003.

algorithms are needed in a large-scale application; the different algorithms can have interlocking parameter settings, and the availability of parallel computing platforms influences algorithmic decisions. Since the difference in performance between an optimal choice of algorithm and hardware, and a less than optimal one, can span orders of magnitude, it is unfortunate that selecting the right solution strategy requires specialized knowledge of both numerical analysis and of computing platform characteristics. Our SANS system aims to assist the application user to navigate this maze of computational possibilities.

This discrepancy in expertises can of course be bridged with manpower based solution: a large enough project can afford to hire a post-doc from computational science, or have a post-doc of its own cross-train in computational science. We believe, however, that it is possible have meta-software – software operating on software – to deliver as good a solution, without the wasted manpower. *Self-adapting Numerical Software (*SANS*)* systems have several levels on which they automate the computational choices for the application scientist.

1.1 Components of a SANS System

We will now describe the various aspects of a SANS system; we will go into details in the following sections. A SANS system comprises the following:

- An *Intelligent Agent* that includes an *automated data analyzer* to uncover necessary information about logical and numerical structure of the user's data, *a data model* for expressing this information as structured metadata, and a *self-adapting decision engine* that can combine this problem metadata with other information (e.g. about past performance of the system) in order to choose the best library and algorithmic strategy for solving the current problem at hand;
- A *history database* that not only records all the information that the intelligent component creates or acquires, but also all the data (e.g., algorithm, hardware, or performance related) that each interaction with a numerical routine produces;
- A *system component* that provides the interface to the available computational resources (whether on a desktop, in a cluster or on a Grid), combining the decision of the intelligent component with both historical information and its own knowledge of available resources in order to schedule the given problem for execution;
- A *metadata vocabulary* that expresses properties of the user data and of performance profiles, and that will be used to build the performance history database. By considering this as *behavioural metadata*, we are led to *intelligent software components* as an extension of the CCA [1,9] framework. The metadata associated with user input makes it possible for the user to pass various degrees of information about the problem to be solved. In the cases where the user passes little information, the Intelligent Agent uses heuristics to uncover as much of this information as is possible.

– One or more *prototype libraries*, for instance for sparse matrix computations, that accept information about the structure of the user's data in order to optimize for execution on different hardware platforms.

One of the more interesting aspects of a SANS system is that it will gradually increase its intelligence. To this end, the IA needs to tune its heuristics, and build new ones over time. In this article we will outline various approaches that can be taken to this end.

1.2 Further Outline of This Paper

We start off with a brief inventory of earlier research into adaptive systems in section 2. Section 3 further describes the metadata that is used internally and externally to store information about problems and algorithms. In section 4 we will go into further details on the Intelligent Agent, in particular describing the heuristic building process. Finally, in section 5 we briefly describe the system components of a SANS system.

2 Related Work

We note the following examples of earlier research into numerical poly-algorithms [20] and adaptive numerical software, focusing mainly on how our proposed research differs from, or often goes beyond, earlier approaches.

– Brewer [6] and Sussman [22] find an *a priori* model, accurately predicting runtime of their algorithms, for which they tune the parameters by measuring specific runs. By contrast, in our application no actual prediction of runtime is possible; we can only weight options against each other as more or less likely to solve the problem faster. Also, they have only a very finite number of algorithms to choose from, whereas our search space is for all practical purposes infinite.
– ATLAS [24] and Sparsity [16] optimize the dense and sparse BLAS [17], respectively. ATLAS relies on a one-time expensive installation, after which its use at runtime engenders no overhead. Sparsity has a runtime component that needs to be amortized, for instance over multiple systems with identical structure. These packages, and in particular ATLAS, are to a large extent independent of the nature of the user data, so no runtime decisions are needed.
– The LINSOL package of Weiss et al. [23] includes a poly-algorithm that picks one among a small number of iterative solvers through backtracking. No analysis of the linear system is performed prior to the iteration process; all decision making is done during the iteration and based on tracking the error norm. Also, any backtracking is only through the space of iterative methods; no ordering of preconditioners is attempted.

- LSA [12,18] is a project for componentizing linear solver software into a problem solving environment. Its stated goal is a visually programmed testbed, rather than an adaptive, intelligent, solver, although there is mention of integrating Case-Based Reasoning for some intelligent assistance.
- Houstis et al. [13] consider the problem of building a 'recommender system' out of a database of performance results. As a case study of their PYTHIA II system they consider solving PDEs with PELLPACK. However, their choices in linear system solvers are extremely limited.
- Some compilers already make use of trace data about previous executions in making decisions about compile-time optimization for code generation. Our approach extends this to both algorithmic choices and to runtime optimizations.

3 Metadata and User Interaction

The operations typically performed by traditional libraries are on data that has been abstracted from the original problem. For instance, one may solve a linear system in order to solve a system of ODEs. However, the fact that the linear system comes from ODEs is lost once the library receives the matrix and right hand side. By introducing metadata, we gain the facility of annotating problem data with information that is typically lost, but which may inform a decision making process. A SANS system will have the facility for the user to pass such metadata explicitly in order to guide the intelligent library. However, one needs to design heuristics for uncovering such lost data, taking the burden completely off the user.

3.1 Types of Metadata

The metadata passed by the user can not only be of varying levels of detail and sophistication, it can also lie on various points of a scale between purely numerical specification on the one extreme, and user application terms on the other. The former corresponds to the traditional parameter-passing approach of numerical libraries: users who are well-versed in numerics can express guidelines regarding the method to be used. However, most users are not knowledgeable about numerics; they can at most be expected to have expert knowledge of their application area. By building in a – heuristic – translation from application domain concepts to numerical concepts we allow the user to annotate the data in problem-native terms, while still assisting the SANS system in decision making.

3.2 Usage Modes of a SANS System

SANS systems can be employed in various ways, depending for instance on the level of expertise of the application user, and on the way the system is called from the application code.

- For a non-expert user, a SANS system acts like an expert system, fully taking the burden of finding the best solver off the user's hands. In this scenario, the user knows little or nothing about the problem – or is perhaps unable to formulate and pass on such information – and leaves it up to the intelligent software to analyze structural and numerical properties of the problem data.
- Users willing and able to supply information about their problem data can benefit from a SANS system in two ways. Firstly, decisions that are made heuristically by the system in expert mode can now be put on firmer ground by the system interrogating the user or the user passing on the information in the call. Secondly, users themselves can search for appropriate solution methods by using the system in 'testbed' mode.
- Finally, expert users, who know by what method they want to solve their problem, can benefit from a SANS system in that it offers a simplified and unified interface to the underlying libraries. Even then, the system offer advantages over the straightforward use of existing libraries in that it can supply primitives that are optimized for the available hardware, and indeed, choose the best available hardware.

4 The SaNS Agent

4.1 The Intelligent Component

The Intelligent Component of a SANS system is the software that accepts the user data and performs a numerical and structural analysis on it to determine what feasible algorithms and data structures for the user problem are. We allow the users to annotate their problem data with 'metadata' (section 3), but in the most general case the Intelligent Component will do this by means of automated analysis (section 4.2). Moreover, any rules used in analyzing the user data and determining solution strategies are subject to tuning (section 4.3) based on performance data (section 4.4) gained from solving the problems. Below we present each of these aspects of the SANS agent in turn.

4.2 Automated Analysis of Problem Data

Users making a request of a SANS system pass to it both data and an operation to be performed on the data. The data can be stored in any of a number of formats, and the intended operation can be expressed in a very global sense ('solve this linear system') or with more detail ('solve this system by an iterative method, using an additive Schwarz preconditioner'). The fewer such details the user specifies, the more the SANS will have to determine the appropriate algorithm, computational kernels, and computing platform. This determination can be done with user guidance, or fully automated. Thus, a major component of a SANS system is an intelligence component that performs various tests to determine the nature of the input data, and makes choices accordingly.

Some of these tests are simple and give an unambiguous answer ('is this matrix symmetric'), others are simple but have an answer that involves a tuning

parameter ('is this matrix sparse'); still others are not simple at all but may involve considerable computation ('is this matrix positive definite'). For the tests with an answer on a continuous scale, the appropriateness of certain algorithms as a function of the tested value can only be preprogrammed to a limited extent. Here the self-adaptivity of the system comes into play: the intelligence component will consult the history database of previous runs in judging the match between algorithms and test values, and after the problem has been solved, data reflecting this run will be added to the database.

4.3 Self-Tuning Rules for Software Adaptation

The Intelligent Component can be characterized as self-tuning in the following sense: The automated analysis of problem data concerns both questions that can be settled quickly and decisively, and ones that can not be settled decisively, or only at prohibitive cost. For the latter category we will use heuristic algorithms. Such algorithms typically involve a weighing of options, that is, parameters that need to be tuned over time by the experience gained from problem runs. Since we record performance data in the history database (section 4.4) of the SANS Agent, we have a mechanism to provide feedback for the adaptation of the analysis rules used by the Intelligent Component, thus leading to a gradual increase in its intelligence.

4.4 History Database

We can gather statistics during the actual problem solving process on a number of levels. There are tools for gathering hardware statistics, such as PAPI [7]. The vocabulary of statistics is also relatively well established in this case.

For the algorithmic level there is no standardization of this vocabulary. In the case of systems solving the obvious measure is the time to solution. However, this time is obviously influenced by hardware considerations (did the problem fit in memory or was page swapping a substantial part of the solution time) so these may need to be recorded too. In iterative methods there is the number of iterations to record, but more detail is obtained if we store the full convergence history. In fact, even more informative is storing the convergence history of the individual Ritz values. This of course requires a solver package (such as Petsc [2]) that allows such sophisticated iteration monitors to be installed.

4.5 Heuristic Building

We may view heuristics as a mapping from the space of problem properties to the space of possible algorithms and algorithm parameters. For each input, this mapping can yield either a specific output, or several outputs with a plausibility ranking; if one method fails, one can fall back one the next ranked.

Heuristics can be based on a single problem parameter, where reaching a certain value triggers a switch over from method to another, or a related variant of

it. Such a heuristic can be discovered by performing statistical analysis, correlating a range of the parameter against a range of performance results. For this to be possible, preferably we need performance results on a sequence of problems where only the parameter under investigation varies. There are several ways in which uch an analysis is possible.

- We can, given enough time, for instance during a setup phase of the system, run exhaustive tests, where a collection of test matrices is subjected to every available method.
- In a production system with only modest time criticality we can occasionally let a system be solved by two methods that only differ in one variable. Note that, given the right software setup, such redundant solving need not come at a 100% overhead cost. For instance, it was shown in [3] how many iterative methods have the same structure, so in running two methods side-by-side their communication stages can be combined, largely eliminating the communication cost of the second method. Even on a single processor, combining two methods can be efficient due to cache effects when a matrix that is operating on two vectors need not be reloaded for the second vector.
- In time-dependent problems and nonlinear solvers linear system occur that are of a gradually evolving nature. This also gives data from which statistical correlations can be harvested.

Heuristics based on multiple problem parameters are not so intuitively straightforward to construct, but the statistical techniques sketched above will still hold. However, since the method parameters usually vary discretely rather than continuously, the performance data stored will likewise be coarse grained. Finding correlations in a multi-dimensional space then requires more confirmation before we trust a thusly tuned heuristic.

5 System Components

5.1 Scheduler

The System Component of the SANS agent manages the different available computation resources (hardware and software), which in today's environment can range from a single workstation, to a cluster, to a Computational Grid. This means that after the intelligent component has analyzed the user's data regarding its structural and numerical properties the system component will take the user data, the metadata generated by the intelligent component, and the recommendations regarding algorithms it has made, and based on its knowledge of available resources farm the problem out to a chosen computational server and a software library implemented on that server. Eventually the results are returned to the user. Empirical data is also extracted from the run and inserted into the database; see section 4.4.

However, this process is not a one-way street. The intelligent component and system component can actually engage in a dialogue as they weigh preferred algorithms against, for instance, network conditions that would make the available implementation of the preferred algorithm less computationally feasible.

Part of the System Component is scheduling operations and querying network resources. Software for this part of a SANS system already exists, in the Netsolve [8], GrADS [4,19], and LFC [21] packages.

5.2 Libraries

Automation of the process of architecture-dependent tuning of numerical kernels can replace the current hand-tuning process with a semiautomated search procedure. Current limited prototypes for dense matrix-multiplication (ATLAS [24] and PHIPAC [5]) sparse matrix-vector-multiplication (Sparsity [15, 14], and FFTs (FFTW [11,10]) show that we can frequently do as well as or even better than hand-tuned vendor code on the kernels attempted.

Current projects use a hand-written *search-directed code generator* (SDCG) to produce many different C implementations of, say, matrix-multiplication, which are all run on each architecture, and the fastest one selected. Simple performance models are used to limit the search space of implementations to generate and time. Since C is generated very machine specific optimizations like instruction selection can be left to the compiler. This approach can be extended to a much wider range of computational kernels by using compiler technology to automate the production of these SDCGs.

References

1. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. C. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of High Performance Distributed Computing*, pages 115–124, 1999.
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. Birkhäuser Press, 1997.
3. Richard Barrett, Michael Berry, Jack Dongarra, Victor Eijkhout, and Charles Romine. Algorithmic bombardment for the iterative solution of linear systems: a poly-iterative approach. *J. Comp. Appl. Math.*, 74:91–109, 1996.
4. F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Applications and Supercomputing*, 15(4):327–344, Winter 2001.
5. J. Bilmes, K. Asanović, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austria, July 1997. ACM SIGARC. see `http://www.icsi.berkeley.edu/~bilmes/phipac`.
6. E.A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of Principles and Practice of Parallel Programming*, pages 80–91, 1995.
7. S. Browne, J Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14:189–204, Fall 2000.

8. H Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 1997.
9. Common Component Architecture Forum. see `www.cca-forum.org`.
10. Matteo Frigo. A fast fourier transform compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia*, May 1999.
11. Matteo Frigo and Stephen Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Seattle, Washington*, May 1998.
12. D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Component architectures for distributed scientific problem solving. *IEEE CS&E Magazine on Languages for Computational Science and Engineering*. to appear.
13. 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. to appear.
14. Eun-Jin Im. *Automatic Optimization of Sparse Matrix - Vector Multiplication*. PhD thesis, University of California at Berkeley, May 2000. To Appear.
15. Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix vector multiplication on SMPs. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.
16. Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science – ICCS 2001. International Conference, San Francisco, CA, USA, May 2001, Proceedings, Part I.*, pages 127–136. Springer Verlag, 2001.
17. C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *Transactions on Mathematical Software*, 5:308–323, 1979.
18. http://www.extreme.indiana.edu/pseware/LSA/LSAhome.html.
19. A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar. Numerical libraries and the grid: The GrADS experiments with scalapack. Technical Report ut-cs-01-460, University of Tennessee, Computer Science Department, 2001.
20. J.R. Rice. On the construction of poly-algorithms for automatic numerical analysis. In *Interactive Systems for Experimental Applied Mathematics. M. Klerer and J.Reinfelds*, pages 301–313. Academic Press, 1968.
21. Kenneth J. Roche and Jack J. Dongarra. Deploying parallel numerical library routines to cluster computing in a self adapting fashion, 2002. Submitted.
22. A. Sussman. Model-driven mapping onto distributed memory parallel computers. In *Supercomputing '92, Minneapolis, MN*, pages 818–829. IEEE Computer Society Press, 1992.
23. Rüdiger Weiss, Hartmut Höfner, and Willi Schönauer. LINSOL (LINear SOLver) – description and user's guide for the parallelized version. Technical Report 61-95, University of Karlsruhe, 1995.
24. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.