# Self-adapting numerical software (SANS) effort

J. Dongarra
G. Bosilca
Z. Chen
V. Eijkhout
G. E. Fagg
E. Fuentes
J. Langou
P. Luszczek
J. Pjesivac-Grbovic
K. Seymour
H. You
S. S. Vadhiyar

*The challenge for the development of next-generation software is the successful management of the complex computational environment while delivering to the scientist the full power of flexible compositions of the available algorithmic alternatives. Self-adapting numerical software (SANS) systems are intended to meet this significant challenge. The process of arriving at an efficient numerical solution of problems in computational science involves numerous decisions by a numerical expert. Attempts to automate such decisions distinguish three levels: algorithmic decision, management of the parallel environment, and processor-specific tuning of kernels. Additionally, at any of these levels we can decide to rearrange the user's data. In this paper we look at a number of efforts at the University of Tennessee to investigate these areas.*

## Introduction

The increasing availability of advanced-architecture computers is having a very significant effect on all spheres of scientific computation, including algorithm research and software development. In numerous areas of computational science—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 can last for hours or days. A major portion of this execution time is usually spent inside numerical routines, such as for the solution of large-scale nonlinear and linear systems that derive from discretized systems of nonlinear partial differential equations. Driven by the desire of scientists for ever higher levels of detail and accuracy in their simulations, the size and complexity of required computations is growing at least as fast as improvements in processor technology. Unfortunately, it is getting more difficult to achieve the necessary high performance from available platforms because of the specialized knowledge in numerical analysis, mathematical software, compilers, and computer architecture that is required, and because rapid innovation in hardware and systems software quickly makes performance-tuning efforts obsolete. Additionally, an optimal scientific environment would have to adapt itself dynamically to changes in the computational platform (e.g., network conditions) and the developing characteristics of the problem to be solved (e.g., during a time-evolution simulation).

With good reason, scientists expect their computing tools to serve them, not the other way around. It is not uncommon for applications that involve a large amount of communication or a large number of irregular memory accesses to run at 10% of peak performance or less. Were this gap to remain fixed, we could simply wait for Moore's law to solve our problems; however, it is growing. Our goal in the self-adapting numerical software (SANS) project is to address this widening gap. There are four challenges to closing it:

- *Challenge 1:* The complexity of modern machines and compilers is so great that very few people know enough, or should be expected to know enough, to predict the performance of an algorithm expressed in a high-level language; there are too many layers of translation from the source code to the hardware. Seemingly small changes in the source code can change performance greatly. In particular, *where* a particular piece of data resides in a deep memory hierarchy is both hard to predict and critical to performance.
- *Challenge 2:* The speed of innovation in hardware and compilers is so great that even if one knew enough



**0018-8646/06/$5.00 © 2006 IBM**

to tune an algorithm for high performance on a particular machine and with a particular compiler, that work would soon be obsolete. Also, platform-specific tuning impedes the portability of the code.
- *Challenge 3:* The number of algorithms, or even algorithmic kernels in standard libraries [1], is large and growing too rapidly for the few experts to keep up with tuning—or to even know them all.
- *Challenge 4:* The need for tuning cannot be restricted to problems that can be solved by libraries in which all optimization is done at design time, installation time, or even compile time. In particular, sparse matrix computations require information about matrix structure for tuning, while interprocessor communication routines require information about machine size and the configuration used for a particular program run. It may be critical to use tuning information captured in prior runs to tune future runs.

In this paper we discuss our efforts to meet these challenges with the following approaches:

- *Generic code optimization:* a system for automatically generating optimized kernels.
- *LAPACK for clusters:* software that manages parallelism of dense linear algebra transparently to the user.
- *SALSA (self-adaptive large-scale solver architecture):* a system for picking optimal algorithms based on statistical analysis of the user problem.
- *Fault-tolerant linear algebra:* a linear algebra approach to fault tolerance and error recovery.
- *Optimized communication library:* optimal implementations of Message Passing Interface (MPI) collective primitives that adapt to the network properties and topology, and the characteristics (such as message size) of the user data.

## Structure of a SANS system

A SANS system has the following large-scale building blocks: application, analysis modules, intelligent switch lower-level libraries, numerical components, database, and modeler. We briefly discuss each and the interfaces needed. Note that not all of the adaptive systems in this paper contain all of the above components.

### Application

The problem to be solved by a SANS system typically derives from an application in a field such as physics or chemistry. This application would normally call a library routine, picked and parameterized by an application expert. Absent such an expert, the application calls the SANS routine that solves the problem.

For maximum ease of use, then, the application programming interface (API) of the SANS routine should be largely similar to the library call it replaces. However, this ignores the issue that we may want the application to pass its metadata to the SANS system. Other application questions to be addressed relate to the fact that we may call the SANS system repeatedly on data that varies only a little between instances. In such cases, we want to limit the effort expended by the analysis modules.

### Analysis modules

Certain kinds of SANS systems—for instance, the ones that govern optimized kernels or communication operations—require little dynamic runtime analysis of the user data. On the other hand, a SANS system for numerical algorithms operates largely at runtime. For this dynamic analysis, we introduce analysis modules into the general structure of a SANS system.

Analysis modules have a two-level structure of categories and elements inside the categories. Categories are mostly intended to be conceptual, but they can also be dictated by practical considerations. An analysis element can be computed either exactly or approximately.

### Intelligent switch

The intelligent switch determines which algorithm or library code to apply to the problem. On different levels, different amounts of intelligence are needed. For instance, dense kernels, such as in the automatically tuned linear algebra software (ATLAS) [2], make essentially no runtime decisions; software for optimized communication primitives typically chooses among a small number of schemes based on only a few characteristics of the data. Systems such as SALSA have a complicated decision-making process based on dozens of features, some of which can be relatively expensive to compute.

### Numerical components

To make numerical library routines more manageable, we embed them in a component framework. This introduces a level of abstraction, as there need not be a one-to-one relation between library routines and components. In fact, we define two kinds of components: *library components* are uniquely based on library routines, but they carry a specification in the numerical adaptivity language that describes their applicability; and *numerical components*, which are based on one or more library routines and have an extended interface that accommodates passing numerical metadata. This distinction allows us to create both generic (*preconditioner*) components and those that correspond

to the specific algorithm level (*incomplete LU with drop tolerance*).

### Database

The database of a SANS system contains information that couples problem features to method performance. While problem features can be standardized (this is numerical metadata), method performance is very much dependent on the problem area and the actual algorithm.

As an indication of some of the problems in defining method performance, consider linear system solvers. The performance of a direct solver can be characterized by the amount of memory and the time it takes, and one can aim to optimize for either or both of them. The amount of memory here is strongly variable among methods and should perhaps be normalized by the memory needed to store the problem. For iterative solvers, the amount of memory is usually limited to a small multiple of the problem memory and is therefore of less concern. However, in addition to the time to solution, one could add a measure such as "time to a certain accuracy," which is interesting if the linear solver is used in a nonlinear solver context.

### Modeler

The intelligence in a SANS system resides in two components: the *intelligent switch*, which makes the decisions, and the *modeler*, which draws up the rules that the switch applies. The modeler draws on the database of problem characteristics (as laid down in the metadata) to make rules.

## Empirical code optimization

As processor speeds double every 18 months following Moore's law [3], memory speed lags behind. Because of this increasing gap between the speeds of processors and memory, new techniques, such as longer pipelines, deeper memory hierarchy, and hyperthreading, have been introduced into hardware design to achieve high performance on modern systems. Meanwhile, compiler optimization techniques have been developed to transform programs written in high-level languages to run efficiently on modern architectures [4, 5]. These program transformations include loop blocking [6, 7], loop unrolling [4], loop permutation, fusion, and distribution [8, 9]. To select optimal parameters such as block size, unrolling factor, and loop order, most compilers would compute these values with analytical models, a process referred to as *model-driven optimization*. In contrast, empirical optimization techniques generate a large number of code variants with different parameter values for an algorithm—for example, matrix multiplication. All of these candidates are then run on the target machine, and the one that gives the best performance is chosen. With this empirical optimization approach, ATLAS [2, 10], PHiPAC [11], and Fastest Fourier Transform in the West (FFTW) [12] successfully generate highly optimized libraries for dense, sparse linear algebra kernels and fast Fourier transform (FFT). It has been shown that empirical optimization is more effective than model-driven optimization [13].

One requirement of empirical optimization methodologies is an appropriate search heuristic to automate the search for the most optimal available implementation [2, 10]. Theoretically, the search space is infinite, but in practice it can be limited on the basis of specific information about the hardware for which the software is being tuned. For example, ATLAS bounds $N_B$ (blocking size) such that $16 \leq N_B \leq \min(\sqrt{L1}, 80)$, where L1 represents the L1 cache size, detected by a microbenchmark. Usually the bounded search space is still very large, and it grows exponentially as the dimension of the search space increases. To find optimal cases quickly, certain search heuristics must be employed. The goal of our research is to provide a general search method that can apply to any empirical optimization system. The Nelder–Mead simplex method [14] is a well-known and successful nonderivative direct search method for optimization. We have applied this method to ATLAS, replacing the global search of ATLAS with the simplex method. This section shows experimental results on four different architectures to compare this search technique with the original ATLAS search, both in terms of the performance of the resulting library and the time required to perform the search.

### Modified simplex search algorithm

Empirical optimization requires a search heuristic for selecting the most highly optimized code from the large number of code variants generated during the search. Because there are a number of different tuning parameters, such as blocking size, unrolling factor, and computational latency, the resulting search space is multidimensional. The direct search method, namely the Nelder–Mead simplex method [14], fits the role perfectly.

The Nelder–Mead simplex method is a direct search method for minimizing a real-valued function $f(x)$ for $x \in \mathbf{R}^n$. It assumes that the function $f(x)$ is continuously differentiable. We modify the search method according to the nature of the empirical optimization technique:

- In a multidimensional discrete space, the value of each vertex coordinate is cast from double precision to integer.
- The search space is bounded by setting $f(x) = \infty$, where $x < l$, $x > u$, and $l$, $u$, and $x \in \mathbf{R}^n$. The determination of the lower and upper bounds is based on hardware information.

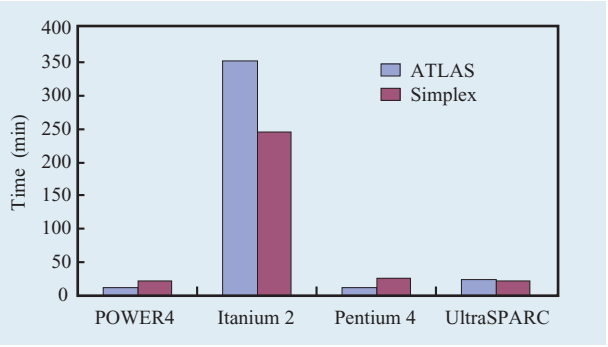


**Figure 1**

Results of search time comparison of ATLAS and Simplex methodologies.

- The simplex is initialized along the diagonal of the search space. The size of the simplex is chosen randomly.
- According to the user-defined restriction conditions, if a point violates the condition, we can simply set $f(x) = \infty$, which saves search time by skipping code generation and execution of this code variant.
- A searchable record of previous execution timing is created at each eligible point. Because execution times would not be identical at the same search point on a real machine, it is very important to be able to retrieve the same function value at the same point. It also saves search time by not having to rerun the code variant for this point.
- Because the search can find only the local optimal performance, multiple runs are conducted. In the search space of $\mathbf{R}^n$, we start $n+1$ searches. The initial simplexes are uniformly distributed along the diagonal of the search space. With the initial simplex of the $n+1$ result vertices of previous searches, we conduct the final search with the simplex method.
- After every search with the simplex method, we apply a local search by comparing performance with neighbor vertices; if a better one is found, the local search continues recursively.

### Experiments with ATLAS

In this section, we briefly describe the structure of ATLAS and then compare the effectiveness of its search technique with that of the simplex search method.

#### *Structure of ATLAS*

By running a set of benchmarks, ATLAS detects hardware information such as L1 cache size, latency for computation scheduling, number of registers, and existence of fused floating-point multiply–add instruction. The search heuristics of ATLAS bound the global search of optimal parameters with detected hardware information. For example, $N_B$ is one of the ATLAS optimization parameters. ATLAS sets the $N_B$ upper bound to be the minimum of 80 and the square root of L1 cache size, and the lower bound as 16, and $N_B$ is a composite of 4. The optimization parameters are generated and fed into the ATLAS code generator, which generates matrix multiply source code. The code is then compiled and executed on the target machine. Performance data is returned to the search manager and compared with previous executions.

ATLAS uses an orthogonal search [13]. For an optimization problem $\min f(x_1, x_2, \cdots, x_n)$, parameters $x_i$ (where $1 \le i \le n$) are initialized with reference values. From $x_1$ to $x_n$, orthogonal search does a linear one-dimensional search for the optimal value of $x_i$, and it uses previously found optimal values for $x_1, x_2, \cdots, x_{n-1}$.

#### *Applying simplex search to ATLAS*

We have replaced the ATLAS global search with the modified Nelder–Mead simplex search and conducted experiments on four different architectures: 2.4-GHz Intel Pentium** 4, 900-MHz Intel Itanium** 2, 1.3-GHz IBM POWER4*, and 900-MHz Sun Ultra**SPARC**.

Given values for a set of parameters, the ATLAS code generator generates a code variant of matrix multiply. The code is executed with randomly generated $1000 \times 1000$ dense matrices as input. After execution of the search heuristic, the output is a set of parameters that gives the best performance for that platform. **Figure 1** compares the total time spent by each of the search methods on the search itself. The Itanium 2 search time (for all search techniques) is much longer than those for the other platforms because we are using the Intel compiler, which, in our experience, takes longer to compile the same piece of code than the GNU compiler collection (GCC) used on the other platforms. **Figure 2** shows the comparison of the performance of matrix multiply on different sizes of matrices using the ATLAS libraries generated by the simplex search and the original ATLAS search.

### Empirical generic code optimization

Current empirical optimization techniques such as ATLAS and FFTW can achieve good performance because the algorithms to be optimized are known ahead of time. We are addressing this limitation by extending the techniques used in ATLAS to the optimization of arbitrary code. Since the algorithm to be optimized is not known in advance, it requires compiler technology to analyze the source code and generate the candidate

implementations. The ROSE project [15, 16] from the Lawrence Livermore National Laboratory provides, among other benefits, a source-to-source code-transformation tool that can produce blocked and unrolled versions of the input code. In combination with our search heuristic and hardware information, we can use ROSE to perform empirical code optimization. For example, on the basis of an automatic characterization of the hardware, we direct their compiler to perform automatic loop blocking at varying sizes, which we can then evaluate to find the best block size for that loop. To perform the evaluations, we have developed a test infrastructure that automatically generates a timing driver for the optimized routine on the basis of a simple description of the arguments.

The generic code optimization system is structured as a feedback loop. The code is fed into the loop processor for optimization and separately fed into the timing driver generator, which generates the code that actually runs the optimized code variant to determine its execution time. The results of the timing are fed back into the search engine. On the basis of these results, the search engine may adjust the parameters used to generate the next code variant. The initial set of parameters can be estimated on the basis of the characteristics of the hardware (e.g., cache size).

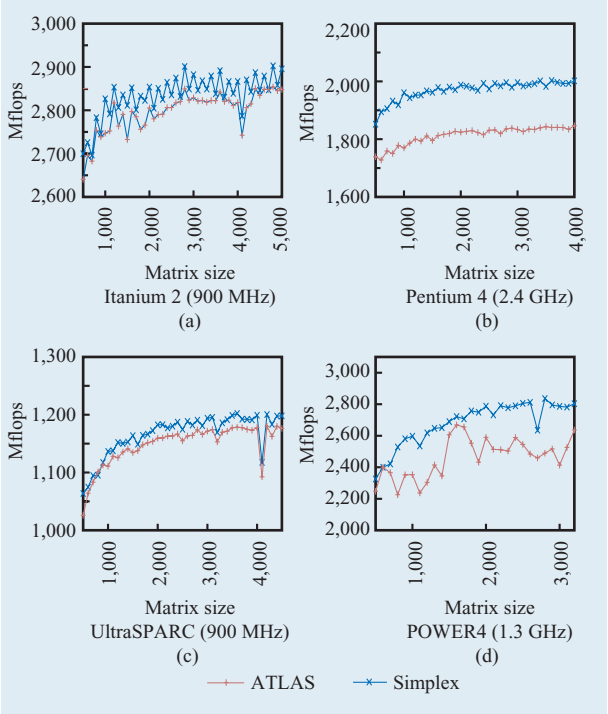


**Figure 2**

Double-precision general matrix multiply (DGEMM) performance.

## LAPACK for clusters

The LAPACK for clusters (LFC) software has a serial, single-process user interface, but delivers the computing power achievable by an expert user working on the same problem who optimally utilizes the parallel resources of a cluster. The basic premise is to design numerical library software that addresses both computational time and space complexity issues on the user's behalf and in a manner transparent to the user. The details for parallelizing the user's problem—such as resource discovery, selection, and allocation, mapping the data onto (and off) the working cluster of processors, executing the user's application in parallel, freeing the allocated resources, and returning control to the user's process in the serial environment from which the procedure began—are all handled by the software. Achieving optimized software in the context described here is an NP-hard problem (i.e., a nondeterministic polynomial-time hard problem) [17–23]. Nonetheless, self-adapting software attempts to tune and approximately optimize computational procedures given the pertinent information on the available execution environment.

### *Overview*

Empirical studies [24] of computing solutions to linear systems of equations demonstrated the viability of the LFC method by finding that (on the clusters tested) there is a problem size that serves as a threshold. For problems greater in size than this threshold, the time saved by the self-adaptive method scales with the parallel application, justifying the approach.

LFC addresses users' problems that may be stated in terms of numerical linear algebra. The problems may otherwise be dealt with by the LAPACK [25], ScaLAPACK [26], or both routines supported in LFC. In particular, suppose that the user has a system of $m$ linear equations with $n$ unknowns, $Ax = b$. Three common factorizations (LU, Cholesky-$LL^T$, and QR) apply for such a system, depending on the properties and dimensions of $A$. All three decompositions are currently supported by LFC.

LFC assumes that only a C compiler, an MPI [27–29] implementation such as MPICH [30] or LAM MPI [31], and some variant of the basic linear algebra subprograms (BLAS), be it ATLAS or a vendor-supplied implementation, are installed on the target system. Target systems are intended to be "Beowulf-like" [32].

### *Algorithm selection processes*

The ScaLAPACK Users' Guide [26] provides the following equation for predicting the total time $T$ spent in

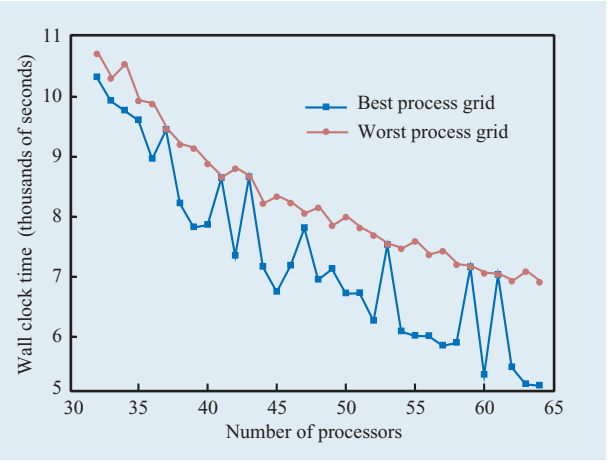


**Figure 3**

Time to solution of a linear system of order 70k with the best and worst aspect ratios of the logical process grid.

one of its linear solvers ($LL^T$, LU, or QR) on a matrix of size $n$ in $N_P$ processes [33]:

$$T(n, N_P) = \frac{C_f n^3}{N_P} t_f + \frac{C_v n^2}{\sqrt{N_P}} t_v + \frac{C_m n}{N_B} t_m , \quad (1)$$

where $N_P$ is the number of processes, $N_B$ is the blocking factor for both computation and communication, $t_f$ is the time per floating-point operation (matrix–matrix multiplication flops rate is a good starting approximation), $t_m$ is the latency, $1/t_v$ is the bandwidth, $C_f$ is the number of floating-point operations, and $C_v$ and $C_m$ are the communication costs; the constants, $C_f$, $C_v$, and $C_m$ should be taken from **Table 1**.

The hard part in using Equation (1) is obtaining the system-related parameters $t_f$, $t_m$, and $t_v$ independently of one another and without performing a costly parameter sweep. At the moment, since we are not aware of any reliable way of acquiring those parameters, we rely on a parameter-fitting approach that uses timing information from previous runs. Furthermore, with respect to runtime software parameters, the equation includes only the process count $N_P$ and blocking factor $N_B$. However, the key to good performance is the correct aspect ratio of the logical process grid: the number of process rows divided by the number of process columns. In heterogeneous environments (which are not taken into account by the equation at all), choosing the right subset of processors is crucial as well. Also, the decision-making process is influenced by the following factors that are directly related to the system policies that define the goal of the optimal selection: resource utilization (*throughput*), time to solution (*response time*), and per-processor performance (*parallel efficiency*).

**Table 1** Constants for use in Equation (1).

| *Driver* | $C_f$ | $C_v$ | $C_m$ |
|---|---|---|---|
| LU | 2/3 | $3 + \frac{1}{4} \log_2 N_P$ | $N_B(6 + \log_2 N_P)$ |
| $LL^T$ | 1/3 | $2 + \frac{1}{2} \log_2 N_P$ | $4 + \log_2 N_P$ |
| QR | 4/3 | $3 + \log_2 N_P$ | $2(N_B \log_2 N_P + 1)$ |

Trivially, the owner (or manager) of the system is interested in optimal resource utilization, while the user expects the shortest time to obtain the solution. Instead of aiming at the optimization of either the former (by maximizing memory utilization and sacrificing the total solution time by minimizing the number of processes involved) or the latter (by using all of the available processors), a benchmarking engineer would be interested in best floating-point performance.

### Experimental results

**Figure 3** illustrates how the time to solution is influenced by the aspect ratio of the logical process grid for a range of process counts. (Each processor was a 1.4-GHz AMD Athlon** with 2 GB of memory; the interconnect was Myricom Myrinet** 2000.) It is clear that sometimes it might be beneficial not to use all of the available processors for computation (the idle processors might be used, for example, for fault-tolerance reasons). This is especially true if the number of processors is a prime number, which leads to a one-dimensional process grid and thus very poor performance on many systems. It is unrealistic to expect that nonexpert users will make the correct decision in every case. It is a matter of having either expertise or relevant experimental data to guide the choice, and our experiences suggest that perhaps a combination of both is required to make good decisions consistently. As a side note, the collection of data for Figure 3 required a number of floating-point operations that would compute the LU factorization of a square dense matrix of order almost 300k. Matrices of that size are usually suitable for supercomputers (the slowest supercomputer on the TOP500** [34] list that factored such a matrix was on position 16 in November 2002).

**Figure 4** shows the large extent to which the aspect ratio of the logical process grid influences another facet of numerical computation: per-processor performance of the LFC parallel solver. The plots in the figure show data for various numbers of processors (between 40 and 64) and consequently do not represent a function, because, for example, ratio 1 may be obtained with $7 \times 7$ and $8 \times 8$ process grids (within the specified range of the number of processors). The figure shows the performance of both parallel LU decomposition using the Gaussian

elimination algorithm and parallel Cholesky factorization code. A side note: The data is relevant for only the LFC parallel linear solvers that are based on the solvers from ScaLAPACK [26]; it would not be indicative of the performance of a different solver, such as HPL [35], which uses different communication patterns and consequently behaves differently with respect to the process grid aspect ratio.

## SALSA

Algorithm choice, the topic of this section, is an inherently dynamic activity in which the numerical content of the user data is of prime importance. Speaking abstractly, we could say that the need for dynamic strategies arises here from the fact that any description of the input space is of a very high dimension. As a corollary, we cannot hope to search this input space exhaustively, and we have to resort to some form of modeling of the parameter space.

We provide a general discussion of the issues in dynamic algorithm selection and tuning, present our approach, which uses statistical data modeling, and give some preliminary results obtained with this approach. Our context here is the selection of iterative methods for linear systems in the SALSA system.

### *Dynamic algorithm determination*

In finding the appropriate numerical algorithm for a problem, we are faced with two issues: First, there are often several algorithms that, potentially, solve the problem; second, algorithms often have one or more parameters of some sort. Thus, given user data, we have to choose an algorithm and choose a proper parameter setting for it. Our strategy in determining numerical algorithms by statistical techniques is globally as follows:

- We solve a large collection of test problems by every available method, that is, every choice of algorithm, and a suitable binning (or categorizing) of algorithm parameters.
- Each problem is assigned to a class corresponding to the method that gives the fastest solution.
- We draw up a list of characteristics of each problem.
- We then compute a probability density function for each class.

As a result of this process, we find a function $p_i(\overline{x})$ where $i$ ranges over all classes, that is, all methods, and $\overline{x}$ is in the space of the vectors of features of the input problems. Given a new problem and its feature vector $\overline{x}$, we then decide to solve the problem with the method $i$ for which $p_i(\overline{x})$ is maximized.

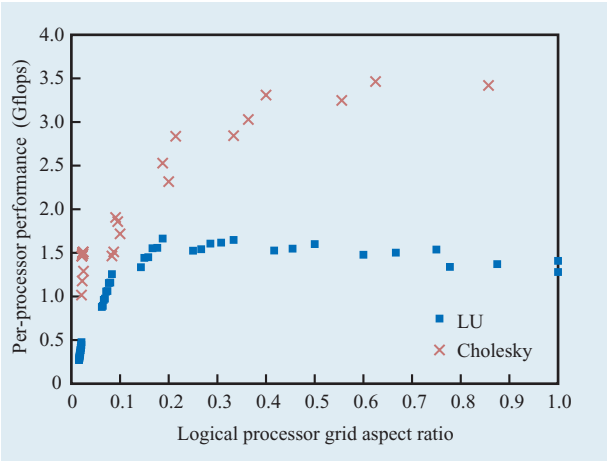


**Figure 4**

Per-processor performance of the LFC parallel linear LU and parallel Cholesky solvers on the Pentium 4 cluster as a function of the aspect ratio of the logical process grid. (Matrix size was 72k and the number of central processing units (CPUs) varied between 40 and 64.)

We now discuss the details of this statistical analysis and present some proof-of-concept numerical results evincing the potential usefulness of this approach.

### *Statistical analysis*

In this section we give a short overview of the way in which a multivariate Bayesian decision rule can be used in numerical decision making. We stress that statistical techniques are merely one of the possible ways of using non-numerical techniques for algorithm selection and parameterization, and the technique we describe here is, in fact, only one among many possible statistical techniques. We describe here the use of parametric models, a technique with obvious implicit assumptions that very likely will not hold overall, but, as we show in the next section, have surprising applicability.

#### *Feature extraction*

The basis of the statistical decision process is the extraction of features from the application problem and the characterization of algorithms in terms of these features. In [36] we have given examples of relevant features of application problems. In the context of linear and nonlinear system solving, we can identify at least the following categories of features: structural features pertaining to the nonzero structure, simple quantities that can be computed exactly and cheaply, spectral properties that have to be approximated, and measures of the variance of the matrix.

**Table 2** Results for sets of features tested.

| *Class* | *Parametric* (%) | *Nonparametric* (%) |
|---|---|---|
| *Features:* nonzeros, bandwidth left and right, splits number, ellipse axes | | |
| Induced | 70 | 60 |
| ParMETIS | 98 | 73 |
| Icmk | 93 | 82 |
| *Features:* diagonal, nonzeros, bandwidth left and right, splits number, ellipse axes and center | | |
| Induced | 70 | 80 |
| ParMETIS | 95 | 76 |
| Icmk | 90 | 90 |

### *Training stage*

On the basis of the feature set described above, we now engage in an expensive and time-consuming training phase in which a large number of problems is solved by every available method. For linear system solving, methods can be described as an orthogonal combination of several choices: the iterative method, the preconditioner, and preprocessing steps, such as scaling or permuting the system. Several of these choices involve numerical parameters, such as the generalized minimal residual (GMRES) restart length or the number of incomplete LU factorization fill levels.

In spite of this multidimensional characterization of iterative solvers, for the purpose of this exposition, we consider methods as a singly indexed set. The essential step in the training process is that each numerical problem is assigned to a class, where the classes correspond to the solvers, and the problem is assigned to the class of the method that gives the fastest solution. As a result of this, we find for each method (class) a multivariate density function:

$$p_j(\overline{x}) = \frac{1}{2\pi|\sum|^{1/2}} e^{-(1/2)(\overline{x}-\overline{\mu})\sum^{-1}(\overline{x}-\overline{\mu})},$$

where $\overline{x}$ are the features, $\overline{\mu}$ the means, and $\Sigma$ the covariance matrix.

Having computed these density functions, we can compute the *a posteriori* probability of a class ("given a sample $\overline{x}$, what is the probability that it belongs in class $j$") as

$$P(w_i|\overline{x}) = \frac{p(\overline{x}|w_j)P(w_j)}{p(\overline{x})}.$$

We then select the numerical method for which this quantity is maximized.

### *Numerical test*

To study the behavior and performance of the statistical techniques described in the previous section, we performed several tests on a number of matrices from different Matrix Market [37] sets. To collect the data, we generated matrix statistics by running an exhaustive test of the different coded methods and preconditioners; for each of the matrices, we collected statistics for each possible existing combination of permutation, scaling, preconditioner, and iterative method.

From this data, we selected those combinations that converged and had the minimum solving time (those that did not converge were discarded). Each possible combination corresponds to a class, but because the number of these combinations is too large, we reduced the scope of the analysis and concentrated only on the behavior of the possible permutation choices. Thus, we have three classes corresponding to the partitioning types: *Induced*—the default PETSc distribution induced by the matrix ordering and the number of processors; *ParMETIS*—using the ParMETIS [38, 39] package; and *Icmk*—a heuristic [40] that, without permuting the system, tries to find meaningful split points based on the sparsity structure. Our test is to see whether the predicted class (method) is indeed the one that yields minimum solving time for each case.

### *Results*

Our results were obtained using the following approach for both training and testing the classifier: Considering that Induced and Icmk are the same except for the Block–Jacobi case, if a given sample is not using Block–Jacobi, it is classified as both Induced and Icmk; if it is using Block–Jacobi, the distinction is made, and it is classified as Induced or Icmk. For the testing set, the verification is performed with the same criteria. For instance, if a sample from the class Induced is not using Block–Jacobi and is classified as Icmk by the classifier, it is still counted as a correct classification (the same as for the inverse case of Icmk classified as Induced). However, if a sample from the Induced class is classified as Induced and is using Block–Jacobi, it is counted as a misclassification. The results for the different sets of features tested are shown in **Table 2**.

It is important to note that although we have many possible features from which we can choose, there may not be enough degrees of freedom (i.e., some of these features may be correlated), so it is important to continue experimenting with other sets of features. The results of these tests may give some lead for further experimentation and understanding of the application of statistical methods on numerical data. These tests and results are a first glance at the behavior of the statistical methods presented, and there is a large amount of

information that can be extracted and explored in other experiments and perhaps using other methods.

## Fault-tolerant linear algebra

As the number of processors in today's high-performance computers continues to grow, the mean time to failure of these computers is becoming significantly shorter than the execution time of many current high-performance applications. Consequently, failures of a node or a process become events to which numerical software must adapt, preferably in an automatic way.

Fault-tolerant linear algebra (FT-LA) stands for and aims at exploring parallel distributed numerical linear algebra algorithms in the context of volatile resources. The parent of FT-LA is FT-MPI [41, 42], which enables an implementer to write fault-tolerant code while providing maximum freedom to the user. With the use of this library, it becomes possible to create more and more fault-tolerant algorithms and software without the need for specialized hardware, thus providing us with the ability to explore new areas for implementation and development.

In this section, we focus only on the solution of sparse linear systems of equations using iterative methods. (For the dense case, we refer to [43] and for eigen value computation, [44].) We assume that a failure of one of the processors (nodes) results in the loss of all of the data stored in its memory (local data). After the failure, the remaining processors are still active, another processor is added to the communicator to replace the failed one, and the MPI environment is intact. These assumptions are true in the FT-MPI context. In the context of iterative methods, once the MPI environment and the initial data (matrix, right-hand sides) are recovered, the next issue, which is our main concern, is to recover the data created by the iterative method.

### Diskless checkpoint-restart technique

To recover the data from any of the processors while maintaining a low storage overhead, we are using a checksum approach at checkpoints. Our checkpoints are diskless, in the sense that the checkpoint is stored in the memory of a processor and not on a disk. To achieve this, an additional processor, referred to as the *checkpoint processor*, is added to the environment; its role is to store the checksum. The checkpoint processor can be viewed as a disk, but with low latency and high bandwidth, because it is located in the network of processors. For more information, we refer the reader to [43, 45], which discuss special interests with respect to simultaneous failures and the scalability of diskless checkpointing (the first reference uses PVM—a message passing library that predates MPI, the second one, FT-MPI).

**Table 3** Time in seconds to compute an 8-MB checksum on two different platforms for a different number of processors.

| | *No. of processors* | | | |
|---|---|---|---|---|
| | *4* | *8* | *16* | *32* |
| boba | 0.21 | 0.22 | 0.22 | 0.23 |
| frodo | 0.71 | 0.71 | 0.72 | 0.73 |

The information is encoded in the following trivial way: If there are $n$ processors for each of which we wish to save the vector $x_k$ (for simplicity, we assume that the size of $x_k$ is the same on all of the processors), the checkpoint unit stores the checksum $x_{n+1}$ such that $x_{n+1} = \sum_{i=1,\cdots,n} x_i$. If processor $f$ fails, we can restore $x_f$ via $x_f = x_{n+1} \sum_{i=1,\cdots,n;i\neq f} x_i$. The arithmetic used for the operations + and − can be either binary or floating-point arithmetic.

When the size of the information is large (as in our case), a pipelined broadcasting algorithm enables the computation of checksum to have almost perfect scaling with respect to the number of processors (see [46] for a theoretical bound). For example, in **Table 3** we report experimental data on two different platforms at the University of Tennessee: the boba Linux** cluster composed of 32 dual Intel Xeon** processors at 2.40 GHz with Intel e1000 interconnect, and the frodo Linux cluster composed of 65 AMD Opteron** dual processors at 1.40 GHz with Myrinet 2000 interconnect. The time is (almost) independent of the number of processors in this case, because the size of the messages is large enough. These results attest to the good scalability of the checksum algorithm.

### Fault tolerance in iterative methods

In this section, we present different choices and implementations that we have identified for fault tolerance in iterative methods (see as well [44]).

#### c_F strategy: Full checkpointing

In iterative methods such as GMRES, $k$ vectors are needed to perform the $k$th iteration. Those vectors are unchanged after their first initialization. The c_F strategy checkpoints the vectors when they are created. Because this is not scalable in storage terms, we apply this idea only to restarted methods, storing all vectors from a restart point.

#### c_R strategy: Checkpointing with rollback

Sometimes the $k$th iteration can be computed from the knowledge of only a constant number of vectors. This is the case in three-term methods, such as a conjugate

**Table 4** Comparison of the checkpoint fault-tolerant algorithm and the lossy fault-tolerant algorithm. GMRES with Block–Jacobi preconditioner with matrix `stomach` from Matrix Market of size $n = 213{,}360$ on 16 processors.

| *Fault-tolerant strategy* | *Without failure* | | | *Failure at step 10* | | |
|---|---|---|---|---|---|---|
| | *lossy* (s) | *c_F* (s) | *c_R* (s) | *lossy* (s) | *c_F* (s) | *c_R* (s) |
| Number of iterations | 18 | 18 | 18 | 18 | 18 | 28 |
| Time to solution ($T_{\text{Wall}}$) | 7.98 | 8.43 | 8.15 | 14.11 | 13.65 | 16.00 |
| Time of checkpoint ($T_{\text{Chkpt}}$) | | 0.52 | 0.00 | | 0.52 | 0.00 |
| Time lost in rollback ($T_{\text{Rollback}}$) | | | | | | 2.29 |
| Time for recovery ($T_{\text{Recov}}$) | | | | 5.50 | 5.19 | 5.15 |
| Time for recovering MPI environment ($T_{\text{I}}$) | | | | 1.05 | 1.10 | 1.11 |
| Time for recovering static data ($T_{\text{II}}$) | | | | 3.94 | 3.94 | 3.94 |
| Time for recovering dynamic data ($T_{\text{III}}$) | | | | 0.35 | 0.13 | 0.01 |

gradient algorithm or in restarted GMRES, if we store the vectors at a restart point. The c_R strategy checkpoints the vectors of a given iteration only from times to times. If a failure occurs, we restart the computation from the last checkpointed version of those vectors.

***l: Lossy strategy***

An alternative strategy to checkpoint in the context of iterative methods is the lossy strategy. Assuming that no checkpoint of the dynamic variables has been performed and a failure occurs, the local data of the approximate solution before failure $x^{(\text{old})}$ is lost on a processor; however, it is still known on all other processors. Thus, one could create a new approximate solution from the data on the other processors. In our example, this is done by solving the local equation associated with the failed processor. If $A_{i,j}$ represents the submatrix with rows stored on processor $i$ and with column indexes corresponding to the rows stored on processor $j$, $x_j$ is the local part of the vector $x$ stored on the processor $j$, and if processor $f$ fails, we propose to construct a new approximate solution $x^{(\text{new})}$ with

$$x_j^{(\text{new})} = x_j^{(\text{old})} \text{ for } j \neq f$$

and

$$x_f^{(\text{new})} = A_{f,f}^{-1}\left(b_f - \sum_{j \neq f} A_{f,f} x_j^{(\text{old})}\right). \tag{2}$$

If $x^{(\text{old})}$ was the exact solution of the system, Equation (2) constructs $x_f^{(\text{new})} = x_f^{(\text{old})}$; the recovery of $x$ is exact. In general the failure happens when $x_f^{(\text{old})}$ is an approximate solution, in which case $x_f^{(\text{new})}$ is not exactly $x_f^{(\text{old})}$. After this recovery step, the iterative method is restarted from $x_f^{(\text{new})}$.

The lossy approach is strongly connected to the Block–Jacobi algorithm. Indeed, a failure step with the lossy approach is a step of the Block–Jacobi algorithm on the failed processor. Related work by Engelmann and Geist [47] proposes to use the Block–Jacobi itself as an algorithm that is robust under processor failure. However, the Block–Jacobi step can be performed for data recovery and embedded in any solver. Thus, the user is free to use any iterative solver, in particular, Krylov methods that are known to be more robust than the Block–Jacobi method.

In this section, we provide a simple example of the use of these three strategies (c_R, c_F, and lossy) in the context of GMRES. The results are reported in **Table 4**.

For this experiment, we have the following identities:

$$T_{\text{Wall}} = T_{\text{Wall}}(\text{lossy}) + T_{\text{Chkpt}} + T_{\text{Rollback}} + T_{\text{Recov}}$$

and

$$T_{\text{Recov}} = T_{\text{I}} + T_{\text{II}} + T_{\text{III}}\,.$$

$T_{\text{II}}$ and $T_{\text{III}}$ are independent of the recovery strategy.

The best strategy here is c_F. In a general case, the best strategy depends on the methods used. For example, if the matrix is block diagonal dominant and the size of the restart is larger, the lossy strategy is in general the best; for small restart size, c_R is, in general, better. More experiments, including eigen value computation, can be found in [44].

FT-LA provides several implementations of fault-tolerant algorithms for numerical linear algebra algorithms. This enables the program to survive failures by adapting itself to the fault and restart from a coherent state.

## Optimized communication

Previous studies of application usage show that the performance of collective communications is critical to high-performance computing (HPC). Rabenseifner's profiling study [48] showed that some applications spend more than 80% of the time taken for a transfer in collective operations. Given this fact, it is essential for MPI implementations to provide high-performance collective operations. However, collective operation performance is often overlooked in comparison to point-to-point performance. A general algorithm for a given collective communication operation may not give good performance on all systems because of the differences in architectures, network parameters, and the buffering of the underlying MPI implementation. Hence, collective communications have to be tuned for the system on which they will be executed. To determine the optimum parameters for collective communications on a given system, they must be modeled effectively at some level, since exhaustive testing may not produce meaningful results in a reasonable time as system size increases.

### *Collective operations, implementations, and tunable parameters*

Collective operations can be classified as either *one-to-many*/*many-to-one* (single producer or consumer) or *many-to-many* (every participant is both a producer and a consumer). These operations can be generalized in terms of communication via virtual topologies. Our experiments currently support a number of these virtual topologies, such as flat-tree/linear, pipeline (single chain), binomial tree, binary tree, and $k$-chain tree ($k$ fanout followed by $k$ chains). Our tests show that given a collective operation, message size, and number of processes, each of the topologies can be beneficial for some combination of input parameters. An additional parameter that we use is segment size—the size of a block of contiguous data into which the individual communications can be broken down. By breaking a large single message communication into smaller communications and scheduling many of them in parallel, it is possible to increase the efficiency of any underlying communication infrastructure. Thus, for many operations we need to specify both parameters; the virtual topology and the segment size. **Figure 5** shows the number of crossover points between different implementations that can exist for a single collective operation on a small number of nodes when finding the optimal (faster) implementation. Note the logarithmic scale. The number of crossovers demonstrates quite clearly why limiting the number of methods available per MPI operation at runtime can, in many instances, lead to poor performance across the possible usage (parameter) range. The MPI operations currently supported within our various frameworks include `barrier`, `broadcast`, `reduce`, `allreduce`, `gather`, `alltoall`, and `scatter` operations.

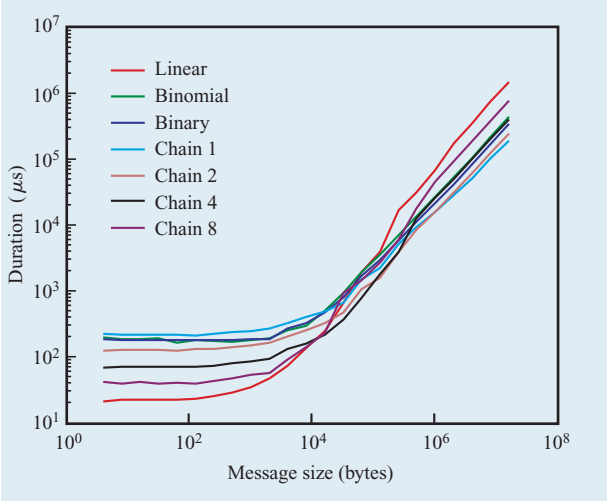


**Figure 5**

Multiple implementations of the MPI reduce operation on 16 nodes.

### *Exhaustive and directed searching*

A simple yet time-consuming method for finding an optimal implementation of an individual collective operation is to run an extensive set of tests over a parameter space for the collective on a dedicated system. However, running such detailed tests, even on relatively small clusters, can take a substantial amount of time [49]. Tuning exhaustively for eight MPI collectives on a small (40-node) IBM SP2* up to message sizes of 1 MB involved approximately 13,000 individual experiments and took 50 hours to complete. Even though this had to occur only once, tuning all of the MPI collectives in a similar manner would take days for a moderately sized system or weeks for a larger system.

Finding the optimal implementation of any given collective can be broken down into a number of stages, with the first stage being dependent on message size, number of processors, and MPI collective operation. The secondary stage is an optimization at these parameters for the correct method (topology–algorithm pair) and segmentation size. The time required for running the actual experiments can be reduced at many levels, for example by not testing at every point and interpolating results (e.g., testing 8, 32, and 128 processes rather than 8, 16, 32, 64, 128, etc.). Additionally, instrumented application runs can be used to build a table of only those

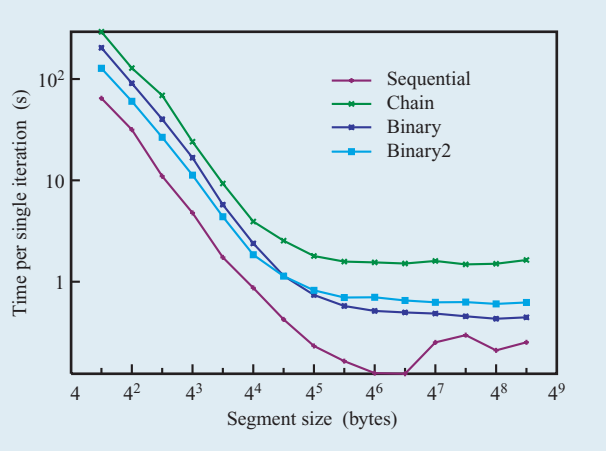


**Figure 6**

Multiple implementations of the MPI `scatter` operation on eight nodes for various segmentation sizes.

collective operations that are required (i.e., not tuning operations that are never called or are called infrequently). We are currently testing this instrumentation method via a newly developed profiling interface known as the scalable application instrumentation system (SAIS).

Another method used to reduce the search space in an intelligent manner is the use of traditional optimization techniques, such as gradient descent with domain knowledge. **Figure 6** shows the performance of four different methods for implementing an eight-processor MPI `scatter` for 128 KB of data on the Sun UltraSPARC cluster when varying the segmentation size. From the resulting shape of the performance data, we can see that the optimal segmentation size occurs for larger sizes and that tests of very small segmentation sizes are very expensive. By using various gradient descent methods to control runtime tests, we can reduce the time to find the optimal segmentation size from 12,613 seconds and 600 tests to 40 seconds and just 90 tests [50]. Thus, simple methods can still allow semi-exhaustive testing in a reasonable time.

### *Communication modeling*

There are many parallel communication models that predict the performance of any given collective operation on the basis of standardizing network and system parameters. Hockney [51], LogP [52], LogGP [53], and PLogP [54] models are frequently used to analyze parallel algorithm performance. Assessing the parameters for these models within a local area network is relatively straightforward, and the methods of approximating them have already been established and are well understood [54, 55]. Thakur and Gropp [56] and Rabenseifner and Träff [57] use the Hockney model to analyze the performance of different collective operation algorithms. Kielmann et al. [58] use the PLogP model to find the optimal algorithm and parameters for topology-aware collective operations incorporated in the MagPIe library. Bell et al. [59] use extensions of LogP and LogGP models to evaluate high-performance networks. Bernaschi et al. [60] analyze the efficiency of the reduce-scatter collective using the LogGP model. Vadhiyar et al. [49, 50] use a modified LogP model that takes into account the number of pending requests that have been queued. Barnett et al. [61] use the LogP model to evaluate and improve the performances of a restricted set of collective communications on a two-dimensional physical mesh (Intel Paragon**). The most important factor concerning communication modeling is that the initial collection of (point-to-point communication) parameters is usually performed by executing a microbenchmark that takes seconds to run followed by some computation to calculate the time per collective operation, and thus is much faster than exhaustive testing. Once a system has been parameterized in terms of a communication model, we can build a mathematical model of a particular collective operation, as shown in [62].

The complexity of identifying the most suitable algorithm for a specific collective operation increases drastically with the number of possible algorithms to be taken into account. **Figure 7** shows the modeled approximation for two different collective communications, one with few available algorithms (`barrier`) and one with a large number of available algorithms. Both tests were done using the MPI library FT-MPI 1.2 [63]. Experiments testing these models on a 32-node cluster for the MPI `barrier` operation are shown in Figure 7(a). As can be clearly seen, none of the models produce perfect results, but they do allow a close approximation to the gross performance of the actual implementations. Figure 7(b) shows the normalized error between the exhaustively found optimal implementation for a broadcast operation and the time for the optimal operation as predicted using the LogP/LogGP parameter models. As can be seen, the models accurately predicted the time to completion of the broadcast for most message sizes and node counts, except for larger node counts when sending smaller messages.

### *Collective communication status and future work*

Current experiments comparing both exhaustively tested collective operations and modeled operations have shown that choosing the wrong implementation of a collective communication can greatly reduce application

performance, while the correct choice can be orders of magnitude better. Exhaustive testing is currently the only sure way to determine the absolute best combination of parameters to create an optimal implementation. Finding these values exhaustively for anything other than a very small system or a very constrained (and repetitive) application is not practical in terms of time to examine the complete search space. We have presented several methods used to reduce or eliminate the search space. Overall, we have found that targeted exhaustive tuning and modeling each have their place, depending on the target applications and systems. Although modeling provides the fastest solution to deciding which implementation of a collective operation to use, it can still produce a very incorrect result for large node counts on small message sizes, while on larger message sizes it appears very accurate. This is encouraging, since the large message sizes are the ones that are impractical to test exhaustively, whereas the small message sizes can be intelligently tested in a reasonable time. Thus, a mixture of methods will still be used until such time as the collective communication models become more accurate for a wider range of parameters, such as node count and data size.

## Conclusion

The emergence of scientific simulation as a pillar of advanced research in many fields is adding new pressure to the demand for a method of rapidly tuning software for high performance on a relentlessly changing hardware base. Driven by the desire of scientists for ever higher levels of detail and accuracy, the size and complexity of computations is growing at least as rapidly as improvements in processor technology, so that applications must continue to extract near-peak performance even as the hardware platforms change underneath them. The problem is that programming these applications is hard, and optimizing them for high performance is even harder.

Speed and portability are conflicting objectives in the design of scientific software. One of the primary obstacles to the efficient solution of scientific problems is the problem of tuning software, both to the available architecture and to the user problem at hand.

A SANS system can dramatically improve the ability of computational scientists to model complex, interdisciplinary phenomena with maximum efficiency and a minimum of extra-domain expertise. SANS innovations (and their generalizations) will provide to the scientific and engineering community a dynamic computational environment in which the most effective library components are automatically selected on the basis of problem characteristics, data attributes, and the state of the grid.

Our efforts, together with those of others in the community, to obtain tuned high-performance kernels to make it possible for suitable algorithms to be automatically chosen hold great promise for evolving high-performance systems.

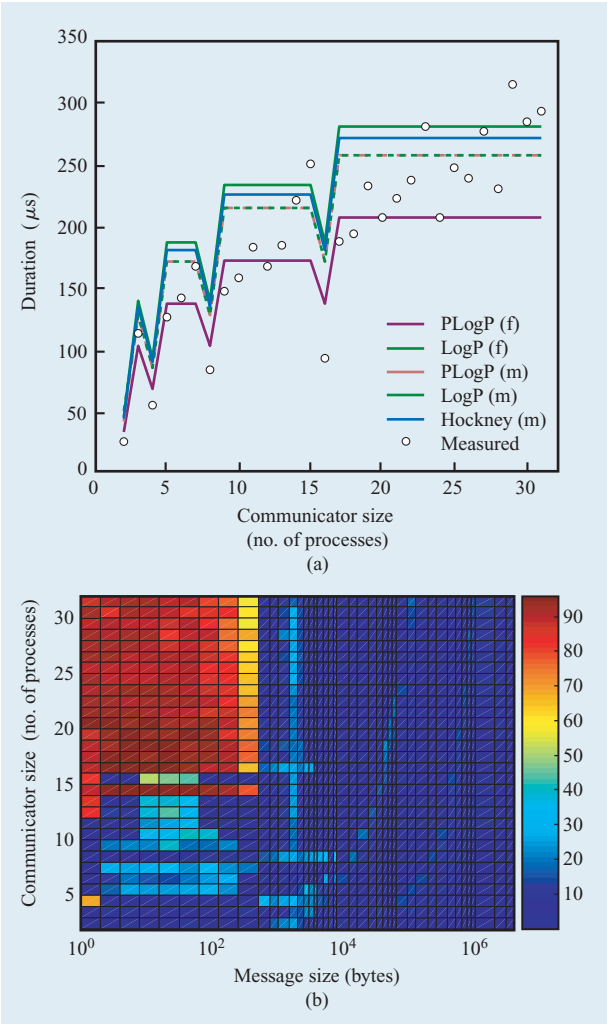


**Figure 7**

(a) Results of measured compared with modeled MPI `barrier` operation based on recursive doubling. (b) Error (color) between the exhaustively measured optimal implementation compared with the implementation chosen using values from the LogP/LogGP-modeled broadcast operations.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Intel Corporation, Sun Microsystems, Inc., SPARC International, Inc.,

Advanced Micro Devices, Inc., Myricom, Inc., TOP500.org, or Linus Torvalds in the United States, other countries, or both.

## References


1. C. L. Lawson, R. J. Hanson, F. T. Krogh, and D. R. Kincaid, "Algorithm 539: Basic Linear Algebra Subprograms for FORTRAN Usage [F1]," *ACM Trans. Math. Software* **5**, No. 3, 324–325 (1979).
2. R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing* **27**, No. 1/2, 3–35 (2001).
3. G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics* **38**, No. 8, 114–117 (1965).
4. R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan-Kaufmann Publishing Co., San Francisco, 2002.
5. D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Source Commun. ACM* **29**, No. 12, 1184–1201 (1986).
6. Q. Yi, K. Kennedy, H. You, K. Seymour, and J. Dongarra, "Automatic Blocking of QR and LU Factorizations for Locality," *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*, 2004, pp. 12–22.
7. R. Schreiber and J. Dongarra, "Automatic Blocking of Nested Loops," *Technical Report CS-90-108*, Department of Computer Science, University of Tennessee, Knoxville, TN 37996, 1990.
8. K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Program. Lang. & Syst.* **18**, No. 4, 424–453 (1996).
9. U. Banerjee, "A Theory of Loop Permutations," *Selected Papers of the 2nd Workshop on Languages and Compilers for Parallel Computing*, Pitman Publishing Ltd., London, 1990, pp. 54–74.
10. J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self-Adapting Linear Algebra Algorithms and Software," *Proc. IEEE* **93**, No. 2, 293–312 (2005).
11. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology," *Proceedings of the International Conference on Supercomputing*, 1997, pp. 340–347.
12. M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 1998, pp. 1381–1384.
13. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A Comparison of Empirical and Model-Driven Optimization," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003, pp. 63–76.
14. J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer J.* **7**, No. 4, 308–313 (1965).
15. Q. Yi and D. Quinlan, "Applying Loop Optimizations to Object-Oriented Abstractions Through General Classification of Array Semantics," *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004; see *http://www.cs.utsa.edu/~qingyi/papers/LCPC04.pdf*.
16. D. Quinlan, M. Schordan, Q. Yi, and A. Saebjornsen, "Classification and Utilization of Abstractions for Optimization," *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods*, 2004, pp. 2–9.
17. E. Amaldi and V. Kann, "On the Approximability of Minimizing Nonzero Variables or Unsatisfied Relations in Linear Systems," *Theoret. Computer Sci.* **209**, 237–260 (1998).
18. P. Crescenzi and V. Kann, Eds., *A Compendium of NP Optimization Problems*, 2005; see *http://www.nada.kth.se/theory/problemlist.html*.
19. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, 1979.
20. J. Gergov, "Approximation Algorithms for Dynamic Storage Allocation," *Proceedings of the 4th Annual European Symposium on Algorithms*, 1996, pp. 52–61.
21. D. S. Hochbaum and D. B. Shmoys, "A Polynomial Approximation Scheme for Machine Scheduling on Uniform Processors: Using the Dual Approach," *SIAM J. Computing* **17**, No. 3, 539–551 (1988).
22. V. Kann, "Strong Lower Bounds on the Approximability of Some NPO PB-Complete Maximization Problems," *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science*, 1995, pp. 227–236.
23. J. Lenstra, D. Shmoys, and E. Tardos, "Approximation Algorithms for Scheduling Unrelated Parallel Machines," *Math. Program.* **46**, No. 3, 259–271 (1990).
24. K. J. Roche and J. J. Dongarra, "Deploying Parallel Numerical Library Routines to Cluster Computing in a Self-Adapting Fashion," *Parallel Computing: Advances and Current Issues*, Imperial College Press, London, 2002.
25. E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK User's Guide*, Third Edition, Society for Industrial and Applied Mathematics, Philadelphia, 1999.
26. L. S. Blackford, J. Choi, A. Cleary, E. F. D'Azevedo, J. W. Demmel, I. S. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1997.
27. Message Passing Interface Forum,"MPI: A Message-Passing Interface Standard," *Intl. J. Supercomputer Appl. & High Perform. Computing* **8**, No. 3/4, 159–416 (1994).
28. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 1.1, 1995; see *http://www.mpi-forum.org/docs/docs.html*.
29. Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, 1997; see *http://www.mpi-forum.org/docs/mpi2-report.pdf*.
30. MPICH; see *http://www.mcs.anl.gov/mpi/mpich/*.
31. LAM/MPI Parallel Computing; see *http://www.lam-mpi.org/*.
32. T. Sterling, *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA, October 2001.
33. J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," *Sci. Program.* **5**, No. 3, 173–184 (1996).
34. TOP500 Supercomputer Sites; see *http://www.top500.org* and *http://www.netlib.org/benchmark/top500.html*.
35. J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present, and Future," *Concurrency & Computation: Pract. & Exper.* **15**, No. 9, 803–820 (2003).
36. V. Eijkhout and E. Fuentes, "A Proposed Standard for Numerical Metadata," *Technical Report ICL-UT-03-02*, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, 2003.
37. Matrix Market; see *http://math.nist.gov/MatrixMarket*.
38. K. Schloegel, G. Karypis, and V. Kumar, "Parallel Multilevel Algorithms for Multi-Constraint Graph Partitioning," *Proceedings of the 6th International Euro-Par Conference*, 2000, pp. 296–310.
39. The ParMETIS/METIS package; see *http://glaros.dtc.umn.edu/gkhome/views/metis/*.
40. V. Eijkhout, "Automatic Determination of Matrix Blocks," *Technical Report UT-CS-01-458*, Department of Computer Science, University of Tennessee, Knoxville, TN 37996, 2001.
41. G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra, "Extending the MPI Specification for Process Fault Tolerance on High

Performance Computing Systems," *Proceedings of the International Supercomputer Conference*, 2004.
42. E. Gabriel, G. E. Fagg, A. Bukovsky, T. Angskun, and J. J. Dongarra, "A Fault-Tolerant Communication Library for Grid Environments," *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS'03), International Workshop on Grid Computing*, 2003; see *http://icl.cs.utk.edu/news_pub/submissions/FTMPI-SF-gabriel.pdf*.
43. J. S. Plank, Y. Kim, and J. J. Dongarra, "Fault-Tolerant Matrix Operations for Networks of Workstations Using Diskless Checkpointing," *J. Parallel & Distr. Computing* **43**, No. 2, 125–138 (1997).
44. G. Bosilca, Z. Chen, J. Dongarra, and J. Langou, "Recovery Patterns for Iterative Methods in a Parallel Unstable Environment," *Technical Report UT-CS-04-538*, Computer Science Department, University of Tennessee, Knoxville, TN 37996, 2004.
45. Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Building Fault Survivable MPI Programs with FT-MPI Using Diskless Checkpointing," *Proceedings of the ACM SIG-PLAN Symposium on Principles and Practice of Parallel Programming*, 2005, pp. 213–223.
46. P. Sanders and J. F. Sibeyn, "A Bandwidth Latency Tradeoff for Broadcast and Reduction," *Info. Process. Lett.* **86**, No. 1, 33–38 (2003).
47. C. Engelmann and G. A. Geist, "Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors," see *http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf*.
48. R. Rabenseifner, "Automatic MPI Counter Profiling of All Users: First Results on a CRAY T3E 900-512," *Proceedings of the Message Passing Interface Developer's and User's Conference*, 1999, pp. 77–85.
49. S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications," *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2000, p. 3.
50. S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra, "Towards an Accurate Model for Collective Communications," *Intl. J. High Perform. Computing Appl.* **18**, No. 1, 159–167 (2004).
51. R. W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Computing* **20**, No. 3, 389–398 (March 1994).
52. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 1–12.
53. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. J. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation," *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1995, pp. 95–105.
54. T. Kielmann, H. E. Bal, and K. Verstoep, "Fast Measurement of LogP Parameters for Message Passing Platforms," *Proceedings of the 15th IPDPS Workshops on Parallel and Distributed Processing*, 2000, pp. 1176–1183.
55. D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa, "Assessing Fast Network Interfaces," *IEEE Micro* **16**, No. 1, 35–43 (1996).
56. R. Thakur and W. Gropp, "Improving the Performance of Collective Operations in MPICH," *Proceedings of the 10th European PVM/MPI User's Group Meeting*, 2003, pp. 257–267.
57. R. Rabenseifner and J. L. Träff, "More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems," *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, 2004, pp. 36–46.
58. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems," *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1999, pp. 131–140.
59. C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick, "An Evaluation of Current High-Performance Networks," *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003, p. 28.
60. M. Bernaschi, G. Iannello, and M. Lauria, "Efficient Implementation of Reduce-Scatter in MPI," *J. Syst. Arch.* **49**, No. 3, 89–108 (2003).
61. M. Barnett, L. Shuler, S. Gupta, D. G. Payne, R. van de Geijn, and J. Watts, "Building a High-Performance Collective Communication Library," *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1994, pp. 107–116.
62. J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance Analysis of MPI Collective Operations," *Proceedings of the 4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems*, 2005, p. 272a.
63. G. E. Fagg, A. Bukovsky, and J. J. Dongarra, "HARNESS and Fault Tolerant MPI," *J. Parallel Computing* **27**, No. 11, 1479–1495 (2001).

*Received June 21, 2005; accepted for publication August 22, 2005; Internet publication February 28, 2006*

**Jack Dongarra** *Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (dongarra@cs.utk.edu)*. Dr. Dongarra is a University Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee. He holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), and is an Adjunct Professor in the Computer Science Department at Rice University. Dr. Dongarra specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, TOP500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports, and technical memoranda, and he is a coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high-performance computers using innovative approaches. Dr. Dongarra is a Fellow of the AAAS, ACM, and the IEEE, and is a member of the National Academy of Engineering.

**George Bosilca** *Innovative Computing Laboratory, Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (bosilca@cs.utk.edu)*. Dr. Bosilca is a Senior Research Associate at the Innovative Computing Laboratory (ICL). He received a Ph.D. degree in parallel architectures from the Université de Paris XI. He was the main developer of the channel memory subsystem for MPICH-V. Dr. Bosilca is currently working on the Open MPI project.

**Zizhong Chen** *Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (zchen@cs.utk.edu)*. Mr. Chen is a Ph.D. candidate in computer science at the University of Tennessee. He received a B.S. degree in mathematics from Beijing Normal University, China, and an M.S. degree in computer science from the University of Tennessee. His research interests are primarily in the areas of high-performance computing, parallel and distributed systems, fault tolerance and checkpoint, scientific computing, and numerical software for high-performance computers. Mr. Chen is currently involved in several high-performance computing projects, including SANS, fault-tolerant MPI, and LFC.

**Victor Eijkhout** *Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (eijkhout@cs.utk.edu)*. Dr. Eijkhout holds a Ph.D. degree in numerical mathematics from the University of Nijmegen, The Netherlands. His research includes numerical linear algebra, distributed computing, and performance modeling and optimization. Dr. Eijkhout recently began investigating statistical and other nonnumerical techniques to solve the problem of choosing the best numerical algorithm.

**Graham E. Fagg** *Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (fagg@cs.utk.edu)*. Dr. Fagg is a Research Associate Professor at the University of Tennessee involved in the development of a number of metacomputing and GRID middleware systems. He received a B.S. degree in computer science and cybernetics from the University of Reading, and a Ph.D. degree in computer science. Dr. Fagg is a member of the IEEE.

**Erika Fuentes** *Innovative Computing Laboratory, Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (efuentes@cs.utk.edu)*. Miss Fuentes is a Graduate Research Assistant at the ICL. She is also a Ph.D. student in the Computer Science Department at the University of Tennessee, where she received an M.S. degree. Miss Fuentes is currently working on the SANS-SALSA project.

**Julien Langou** *Innovative Computing Laboratory, Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (langou@cs.utk.edu)*. Dr. Langou is a Research Scientist. He received a B.Sc. degree in propulsion engineering from École National Supérieure de l'Aéronautique et de l'Espace (SUPAERO), France, and a Ph.D. degree in applied mathematics from the National Institute of Applied Sciences (INSA), France. His research interest is in numerical linear algebra with application in high-performance computing.

**Piotr Luszczek** *Innovative Computing Laboratory, Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (luszczek@cs.utk.edu)*. Dr. Luszczek is a Research Scientist at the ICL. He received a Ph.D. degree in computer science from the University of Tennessee. His current research is focused on numerical linear algebra, parallel processing, benchmarking, and self-adapting software. Dr. Luszczek is one of the core developers of the HPC Challenge Benchmark and the LFC project. He is also an active contributor to LAPACK and ScaLAPACK packages.

**Jelena Pjesivac-Grbovic** *Innovative Computing Laboratory, Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (jpjesiva@utk.edu)*. Ms. Pjesivac-Grbovic is a Graduate Research Assistant at the ICL, working toward a Ph.D. degree in computer science. She received a B.S. degree in computer science and physics from Ramapo College of New Jersey. Her research interests are parallel communication libraries and computer architectures, scientific and grid computing, and modeling of biophysical systems.

**Keith Seymour** *Innovative Computing Laboratory, Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (seymour@cs.utk.edu)*. Mr. Seymour is a Senior Research Associate at the ICL. He received an M.S. degree in computer science from the University of Tennessee. His research interests include grid computing and empirical code optimization.

**Haihang You** *Innovative Computing Laboratory, Computer Science Department, University of Tennessee, 1122 Volunteer Boulevard, Knoxville, Tennessee 37996 (you@cs.utk.edu)*. Mr. You is a Research Associate at the ICL. He received an M.S. degree in computer science from the University of Tennessee. His research interests include performance analysis and empirical code optimization.

**Sathish S. Vadhiyar** *Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, India (vss@serc.iisc.ernet.in)*. Dr. Vadhiyar is an Assistant Professor at the Supercomputer Education and Research Centre, Bangalore. He received a B.E. degree in computer science and engineering from Thiagarajar College of Engineering, India, an M.S. degree in computer science from Clemson University, and a Ph.D. degree in computer science from the University of Tennessee. Dr. Vadhiyar's research interests include parallel and grid computing.