Self Adapting Numerical Software (SANS) Effort

George Bosilca, Zizhong Chen, Jack Dongarra, Victor Eijkhout, Graham E. Fagg, Erika Fuentes, Julien Langou, Piotr Luszczek, Jelena Pjesivac-Grbovic, Keith Seymour, Haihang You, Sathish S. Vadhiyar
University of Tennessee and Oak Ridge National Laboratory; Indian Institute of Science

June 2005

Abstract

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;
- 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 that are investigating these areas.

1 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 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. 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 the 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 required, and because rapid innovation in hardware and systems software rapidly makes performance tuning efforts obsolete. Additionally, an optimal scientific environment would have to adapt itself dynamically to changes in the computational platform – for instance, network conditions – and the developing characteristics of the problem to be solve – for instance, during a time-evolution simulation.

With good reason scientists expect their computing tools to serve them and 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 or less. If this gap were fixed then we could simply wait for Moore's Law to solve our problems, but the gap is growing. The challenge of closing this gap is exacerbated by four factors.

Scientific applications need to be tuned to extract near peak performance even as hardware platforms change underneath them. Unfortunately, tuning even the simplest real-world operations for high performance usually requires an intense and sustained effort, stretching over a period of weeks or months, from the most technically advanced programmers, who are inevitably in very scarce supply. While access to necessary computing and information technology has improved dra-

matically over the past decade, the efficient application of scientific computing techniques still requires levels of specialized knowledge in numerical analysis, mathematical software, computer architectures, and programming languages that many working researchers do not have the time, the energy, or the inclination to acquire.

Our goal in the SANS projects are to address the widening gap between peak performance of computers and attained performance of real applications in scientific computing.

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 datum 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 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 [41] is large and growing, too fast for the few experts to keep up with tuning or even knowing about them all.

Challenge 4: The need for tuning cannot be restricted to problems that can be solved by libraries where 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 configuration used for a particular program run. It may be critical to use tuning information captured in prior runs to tune future runs.

A SANS system comprises intelligent next generation numerical software that domain scientists – with disparate levels of knowledge of algorithmic and programmatic complexities of the underlying numerical software – can use to easily express and efficiently solve their problem.

The following sections describe the various facets of our effort.

- Generic Code Optimization GCO: a system for automatically generating optimized kernels.
- LFC: software that manages parallelism of dense linear algebra, transparently to the user.
- SALSA: a system for picking optimal algorithms based on statistical analysis of the user problem.
- FTLA: a linear algebra approach to fault tolerance and error recovery.
- Optimized Communication Library: optimal implementations of MPI collective primitives that adapt to the network properties and topology as well as the characteristics (such as message size) of the user data.

2 Structure of a SANS system

A SANS system has the following large scale building blocks:

- Application
- Analysis Modules
- Intelligent switch
- Lower level libraries
- Database
- Modeler

We will give a brief discussion of each of these, and the interfaces needed. Note that not all of the adaptive systems in this paper have all of the above components.

2.1 The Application

The problem to be solved by a SANS system typically derives from a physics, chemistry, et cetera application. This application would normally call a library routine, picked and parametrized 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 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 application 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.

2.2 Analysis Modules

Certain kinds of SANS systems, for instance the ones that govern optimized kernels or communication operations, need 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 either be computed exactly or approximately.

2.3 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 ATLAS [66] make essentially no runtime decisions; software for optimized communication primitives typically chooses between 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.

2.4 Numerical Components

In order to make numerical library routines more managable, we embed them in a component framework. This will also introduce a level of abstraction, as there need not be a one-to-one relation between library routines and components. In fact, we will 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;
- 'numerical components' are based on one or more library routines, and having an extended interface that accomodates passing numerical metadata.

This distinction allows us to make components corresponding to the specific algorithm level ('Incomplete LU with drop tolerance') and generic ('preconditioner').

2.5 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 between 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 therefore of less concern. However, in addition to the time to solution, one could here add a measure such as "time to a certain accuracy", which is interesting if the linear solver is used in a nonlinear solver context.

2.6 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.

3 Empirical Code Optimization

As CPU speeds double every 18 months following Moore's law[47], memory speed lags behind. Because of this increasing gap between the speeds of processors and memory, in order to achieve high performance on modern systems new techniques such as longer pipeline, deeper memory hierarchy, and hyper threading have been introduced into the hardware design. Meanwhile, compiler optimization techniques have been developed to transform programs written in high-level languages to run efficiently on modern architectures[2, 50]. These program transformations include loop blocking[68, 61], loop unrolling[2], loop permutation, fusion and distribution[45, 5]. To select optimal parameters such as block size, unrolling factor, and loop order, most compilers would compute these

values with analytical models 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 mulplication. All these candidates run on the target machine, and the one that gives the best performance is picked. With this empirical optimization approach ATLAS[67, 18], PHiPAC[8], and FFTW[29] successfully generate highly optimized libraries for dense, sparse linear algebra kernels and FFT respectively. It has been shown that empirical optimization is more effective than model-driven optimization[70].

One requirement of empirical optimization methodologies is an appropriate search heuristic, which automates the search for the most optimal available implementation [67, 18]. Theoretically, the search space is infinite, but in practice it can be limited based on 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 micro-benchmark. Usually the bounded search space is still very large and it grows exponentially as the dimension of the search space increases. In order to find optimal cases quickly, certain search heuristics need to 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[49] is a well-known and successful non-derivative direct search method for optimization. We have applied this method to ATLAS, replacing the global search of ATLAS with the simplex method. This section will show 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.

3.1 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 multi-dimensional. The direct search method, namely Nelder-Mead simplex method [49], fits in 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 the function $f(x)$ is continuously differentiable. We modify the search method according to the nature of the empirical optimization technique:

- In a multi-dimensional 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 lower and upper bounds are determined based on hardware information.
- The simplex is initialized along the diagonal of the search space. The size of the simplex is chosen randomly.
- 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.
- Create a searchable record of previous execution timing at each eligible point. Since 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 re-run the code variant for this point.
- As the search can only find the local optimal performance, multiple runs are conducted. In 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, and if a better one is found the local search continues recursively.

3.2 Experiments with ATLAS

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

3.2.1 Structure of ATLAS

By running a set of benchmarks, ATLAS [67, 18] 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 (blocking size) is one of ATLAS's optimization parameters. ATLAS sets N_B's upper bound to be the minimum of 80 and square root of L1 cache size, and 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 [70]. For an optimization problem:

$$min\ f(x_1, x_2, \cdots, x_n)$$

Parameters x_i (where $1 \leq i \leq n$) 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 $x_1, x_2, \cdots, x_{n-1}$.

3.2.2 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 Pentium 4, 900 MHz Itanium 2, 1.3 GHz Power 4, and 900 MHz Sparc Ultra.

Given values for a set of parameters, the ATLAS code generator generates a code variant of matrix multiply. The code gets executed with randomly generated 1000x1000 dense matrices as input. After executing 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 Itanium2 search time (for all search techniques) is much longer than 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 compiler used on the other platforms (gcc). Figure 2 shows the comparision 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.

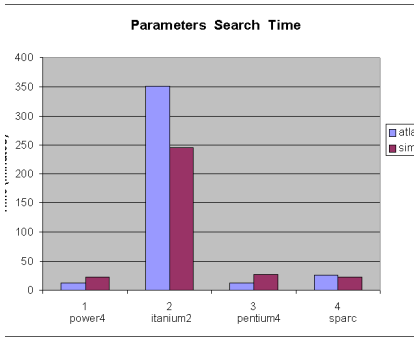

Figure 1: Search time

3.3 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 will require compiler technology to analyze the source code and generate the candidate implementations. The ROSE project[69, 54] from Lawrence Livermore National Laboratory provides, among other things, a source-to-source code transformation tool that can produce blocked and unrolled versions of the input code. Combined with our search heuristic and hardware information, we can use ROSE to perform empirical code optimization. For example, based on an automatic characterization of the hardware, we will 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 evaulations, we have developed a test infrastructure that automatically generates a timing driver for the optimized routine based on 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. Based on these results, the search engine may adjust the parameters used to generate the next

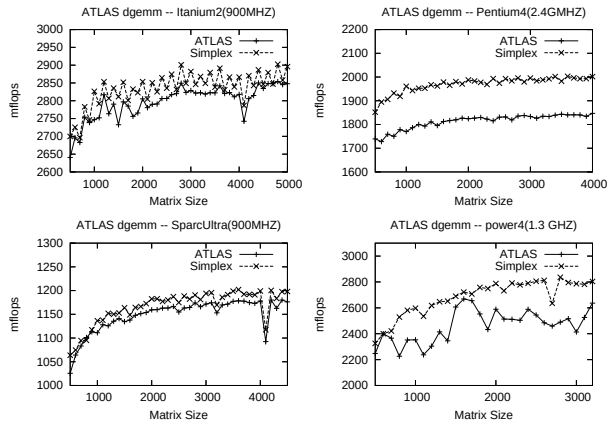

Figure 2: dgemm Performance

code variant. The initial set of parameters can be estimated based on the characteristics of the hardware (e.g. cache size).

To compare the search results against the true optimum performance, we ran an exhaustive search over both dimensions of our search space (block sizes up to 128 and unrolling up to 128). The code being optimized is a naïve implementation of matrix-vector multiply. In general, we see the best results along the diagonal, but there are also peaks along areas where the block size is evenly divisible by the unrolling amount. The triangular area on the right is typically low because when the unrolling amount is larger than the block size, the unrolled portion will not be used. After running for over 30 hours on a 1.7 GHz Pentium M, the best performance was found with block size 11 and unrolling amount 11. This code variant ran at 338 Mflop/s compared to 231 Mflop/s for the default version, using the same compiler.

Of course, due to the large amount of time required, this kind of exhaustive search is not feasible especially as new dimensions are added to the search space. Consequently we are investigating the simplex method as a way to find an optimal set of parameters without performing an exhaustive search. The simplex search technique works the same in this application as it does when applied to ATLAS except in this case we only have two parameters. To evaluate the effectiveness of the simplex search technique, we performed 10,000 runs and compared the results with the true optimum found during the exhaustive search. On average, the simplex technique found code variants that performed at 294 Mflop/s, or 87% of the true optimal performance (338 Mflop/s). At best, the simplex technique can find the true optimum, but that only occurs on 8% of the runs. The worst case was 273 Mflop/s, or 81% of the true optimum. From a statistical point of view, the probability of randomly finding better performance than the simplex average case (294 Mflop/s) is 0.079% and the probability of randomly finding better performance than the simplex worst case (273 Mflop/s) is 2.84%. While the simplex technique generally results in a code variant with less than optimal performance, the total search time is only 10 minutes compared to over 30 hours for the exhaustive search.

4 LFC

The central focus is the LFC (LAPACK For Clusters) software which 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 of) 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 $\mathcal{NP}$-hard problem [3, 15, 32, 33, 34, 37, 42]. Nonetheless, self-adapting software attempts to tune and approximately optimize computational procedures given the pertinent information on the available execution environment.

4.1 Overview

Empirical studies [57] of computing solutions to linear systems of equations demonstrated the viability of the method 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 user's problems that may be stated in terms of numerical linear algebra. The problem may otherwise be dealt with one of the LAPACK [4] and/or ScaLAPACK [9] 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 properties and dimensions of A. All three decompositions are currently supported by LFC.

LFC assumes that only a C compiler, a Message Passing Interface (MPI) [26, 27, 28] implementation such as MPICH [48] or LAM MPI [40], and some variant of the BLAS routines, be it ATLAS or a vendor supplied implementation, is installed on the target system. Target systems are intended to be "Beowulf-like" [62].

4.2 Algorithm Selection Processes

ScaLAPACK Users' Guide [9] provides the following equation for predicting the total time T spent in one of its linear solvers (LL^T, LU, or QR) on matrix of size n in N_P processes [14]:

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

where:

- N_P number of processes
- N_B blocking factor for both computation and communication
- t_f time per floating-point operation (matrix-matrix multiplication flop rate is a good starting approximation)
- t_m corresponds to latency
- $1/t_v$ corresponds to bandwidth
- C_f corresponds to number of floating-point operations
- C_v and C_m correspond to communication costs

The constands C_f C_v and C_m should be taken from the follwing table:

Driver	C_f	C_v	C_m
LU	$2/3$	$3+1/4\log_2 N_P$	$N_B(6+\log_2 N_P)$
LL^T	$1/3$	$2+1/2\log_2 N_P$	$4+\log_2 N_P$
QR	$4/3$	$3+\log_2 N_P$	$2(N_B \log_2 N_P+1)$

The hard part in using the above equation is obtaining the system-related parameters t_f, t_m, and t_v indepently of each other and without performing costly parameter sweep. At the moment we are not aware of any reliable way of acquiring those parameters and thus we rely on 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 right aspect ratio of the logical process grid: the number of process rows divided by the number of process columns. In hetergeneous environments (for which the equation doesn't account at all), choosing the right subset of processor is crucial as well.

Lastly, the decision making process is influenced by the follwing factors directly related to system policies that define what is the goal of the optimal selection:

- resource utilization (throughput),
- time to solution (response time),
- per-processor performance (parallel efficiency).

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 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 the available processors) a benchmarking engineer would be interested in best floating-point performance.

4.3 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. It is clear that sometimes it might be beneficial not to use all the available processors for computation (the idle processors might be used for example for fault-tolerance purposes). 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 non-expert users will correctly make the right decisions in every such case. It is either a matter of having 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, with respect to the experimental data, it is worth mentioning

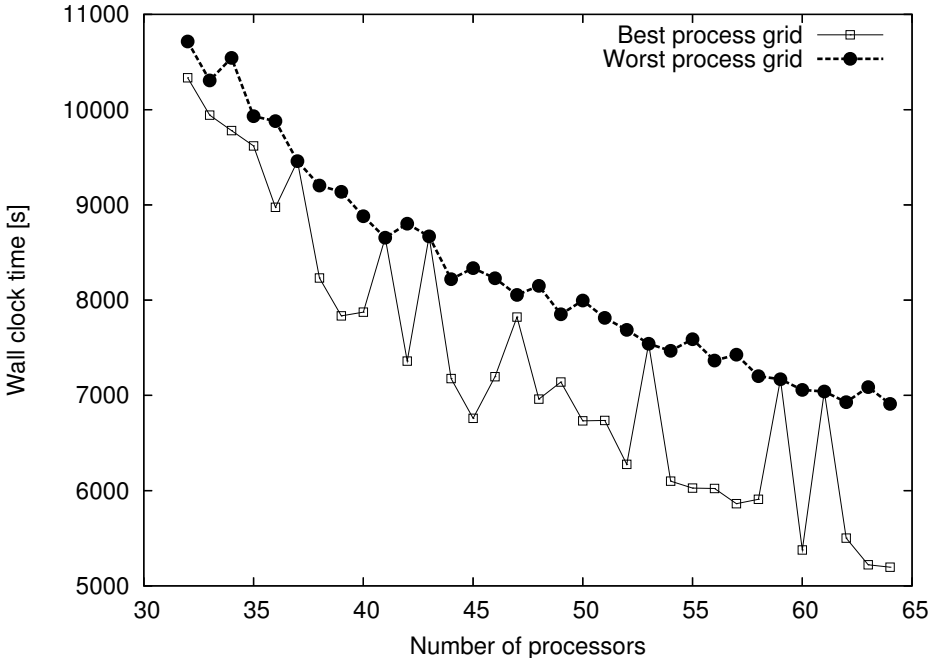

Figure 3: Time to solution of a linear system of order 70000 with the best and worst aspect ratios of the logical process grid. (Each processor was an AMD Athlon 1.4 GHz with 2 GB of memory; the interconnect was Myrinet 2000.)

that 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 three hundred thousand. Matrices of that size are usually suitable for supercomputers (the slowest supercomputer on the Top500 [46] 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 graphs in the figure shows data for various numbers of processors (between 40 and 64) and consequently does not represent a function since, for example, ratio 1 may be obtained with 7 by 7 and 8 by 8 process grids (within the specified range of number of processors). The figure shows performance of both parallel LU decomposition using Gaussian elimination algorithm and parallel Cholesky factorization code. A side note on relevance of data from the figure: they are only relevant for the LFC' parallel linear solvers that are based on the solvers from ScaLAPACK [9], they would not be indicative of performance of a different solver, such as HPL [19] which uses different communication patterns and consequently behaves differently with respect to the process grid aspect ratio.

5 SALSA

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

We will give 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 will be the selection of iterative methods for linear systems in the SALSA (Self-Adaptive Large-scale Solver Architecture) system.

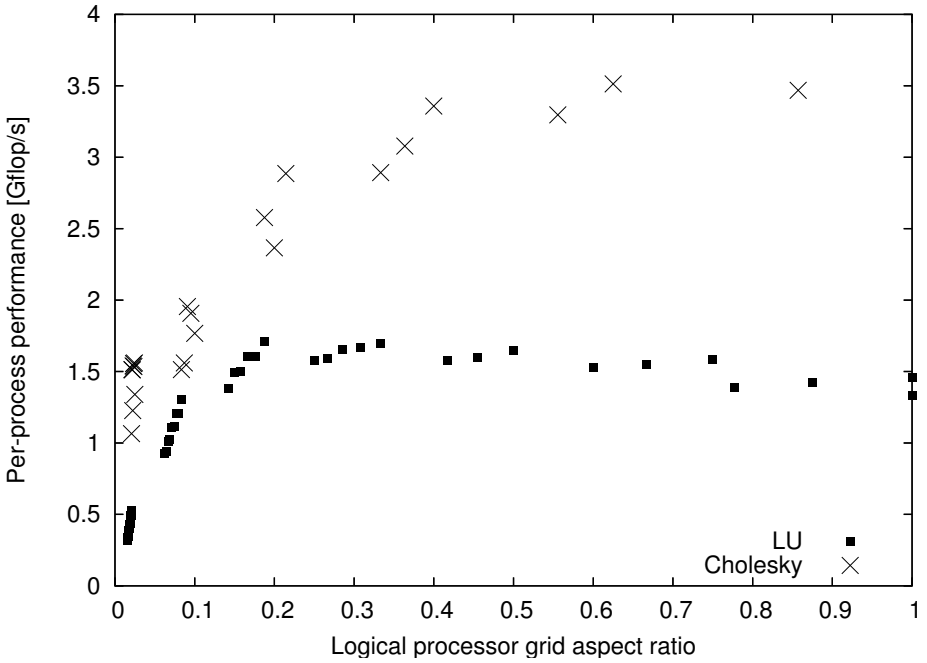

Figure 4: Per-processor performance of the LFC parallel linear LU and parallel Cholesky solvers on the Pentium 4 cluster as a function of the aspect ratio of the logical process grid (matrix size was 72000 and the number of CPUs varied between 40 and 64).

5.1 Dynamic Algorithm Determination

In finding the appropriate numerical algorithm for a problem we are faced with two issues:

1. There are often several algorithms that, potentially, solve the problem, and
2. 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' of algorithm parameters.
- Each problem is assigned to a class corresponding to the method that gave the fastest solution.
- We also 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 $p_i(\bar{x})$ where i ranges over all classes, that is, all methods, and $\bar{x}$ is in the space of the vectors of features of the input problems. Given a new problem and its feature vector $\bar{x}$, we then decide to solve the problem with the method i for which $p_i(\bar{x})$ is maximised.

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

5.2 Statistical analysis

In this section we give a short overview of the way a multi-variate 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 in fact, the technique we describe here is only one among many possible statistical techniques. We will describe here the use of parametric models, a technique with obvious implicit assumptions that very likely will not hold overall, but, as we shall

show in the next section, they have surprising applicability.

5.2.1 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 [21] we have given examples of relevant features of application problems. In the context of linear/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, which have to be approximated; measures of the variance of the matrix.

5.2.2 Training stage

Based on the feature set described above, we now engage in an – expensive and time-consuming – training phase, where a large number of problems is solved with 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 GMRES restart length, or the number of ILU fill levels.

In spite of this multi-dimensional characterization of iterative solvers, we will for the purpose of this exposition 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 gave the fastest solution. As a result of this, we find for each method (class) a multivariate density function:

$$p_j(\bar{x}) = \frac{1}{2\pi|\Sigma|^{1/2}} e^{-(1/2)(\bar{x}-\bar{\mu})\Sigma^{-1}(\bar{x}-\bar{\mu})}$$

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

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

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

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

5.3 Numerical test

To study the behavior and performance of the statistical techniques described in the previous section, we perform several tests on a number of matrices from different sets from the Matrix Market [44]. To collect the data, we generate matrix statistics by running an exhaustive test of the different coded methods and preconditioners; for each of the matrices we collect statistics for each possible existing combination of: permutation, scaling, preconditioner and iterative method.

From this data we select those combinations that converged and had the minimum solving time (those combinations that didn't converge are discarded). Each possible combination corresponds to a class, but since the number of these combination is too large, we reduce the scope of the analysis and concentrate 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 [60, 51] package,

Icmk a heuristic [20] that, without permuting the system, tries to find meaningful split points based on the sparsity structure.

Our test is to see if the predicted class (method) is indeed the one that yields minimum solving time for each case.

5.4 Results

These 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, and if it is using Block-Jacobi then the distinction is made and it is classified as Induced or Icmk. For the testing set, the verification is performed with same criteria, for instance, if a sample from the class Induced is not using Block-Jacobi and is classified as Icmk by the classifier, is still counted as a correct classification (same for the inverse case of Icmk classified as Induced), however, if a sample from the Induced class is classified as Induced and was using block-jacobi, it is counted as a misclassification.

The results for the different sets of features tested, are as follows:

Features: non-zeros, bandwidth left and right, splits number, ellipse axes

	Parametric	Non-Parametric
Induced Class:	70%	60%
Parmetis Class:	98%	73%
Icmk Class:	93%	82%

Features: diagonal, non-zeros, bandwidth left and right, splits number, ellipse axes and center

	Parametric	Non-Parametric
Induced Class:	70%	80%
Parmetis Class:	95%	76%
Icmk Class:	90%	90%

It is important to note that although we have many possible features to choose from, there might not be enough degrees of freedom (i.e. some of these features might be corelated), so it is important to continue experimentation with other sets of features. The results of these tests might 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 plenty of information that can be extracted and explored in other experiments and perhaps using other methods.

6 FTLA

As the number of processors in today's high performance computers continue 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 becomes events to which numerical software needs to adapt, preferably in an automatic way.

FTLA stands for Fault Tolerant Linear Algebra and aims at exploring parallel distributed numerical linear algebra algorithms in the context of volatile resources. The parent of FTLA is FT-MPI [24, 31]. FT-MPI enables an implementer to write fault tolerant code while providing the maximum of freedom to the user. Based on this library, it becomes possible to create more and more fault-tolerant algorithms and software without the need for specialized hardware; thus providing us the ability to explore new areas for implementation and development.

In this section, we only focus on the solution of sparse linear systems of equations using iterative methods. (For the dense case, we refer to [52], for eigenvalue computation [11].) We assume that a failure of one of the processors (nodes) results in the lost of all 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 environenment is sane. 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

In order to recover the data from any of the processors while maintaining a low storage overhead, we are using a checksum approach to checkpointing.

Our checkpoints are diskless, in the sense that the checkpoint is stored in memory of a processor and not on a disk. To achieve this, an additional processor is added to the environment. It will be referred as the checkpoint processor and its role is to store the checksum. The checkpoint processor can be viewed as a disk but with low latency and high bandwidth since it is located in the network of processors. For more information, we refer the reader to [13, 52] where special interests are given on simultaneous failures and the scalability of diskless checkpointing (the first reference uses PVM, the second one FT-MPI).

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

When the size of the information is pretty large (our case), a pipelined broadcasting algorithm enables the computation of checksum to have almost perfect scaling with respect to constant charge of the processors (see [58] for a theoretical bound). For example, in Table 1, we report experimental data on two different platforms at the University of Tennessee: the boba Linux

	4	8	16	32
boba	0.21	0.22	0.22	0.23
frodo	0.71	0.71	0.72	0.73

Table 1: Time (in seconds) for computing an 8-MByte checksum on two different platforms for different number of processors, this time is (almost) independent of the number of processors in this case since the size of the messages is large enough

cluster composed of 32 dual processors Intel Xeon at 2.40 GHz with Intel e1000 interconnect, and the frodo Linux cluster composed of 65 dual processors AMD Opteron at 1.40 GHz with Myrinet 2000 interconnect. The results in Table 1 attest to the good scalabiliy of the checksum algorithm.

Fault Tolerance in Iterative Methods, a simple example

In this Section, we present different choices and implementations that we have identified for fault tolerance in iterative methods (see as well [11]).

c_F strategy: Full checkpointing

In iterative methods like GMRES, k vectors are needed to perform the kth iteration. Those vectors are unchanged after their first initialization. The c_F strategy checkpoints the vectors when they are created. Since 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 kth iteration can be computed from the knowledge of only a constant number of vectors. This is the case in three-term methods such as CG, 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 happens then we restart the computation from the last checkpointed version of those vectors.

l: lossy strategy

An alternative strategy to checkpoint useful 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 $x^{(old)}$ 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 sub-matrix which rows are stored on processor i and with column indexes corresponding to the rows stored on the processor j, x_j is the local part of the vector x stored on the processor j, and if processor f fails then we propose to construct a new approximate solution $x^{(new)}$ via

$$\begin{aligned} x_j^{(new)} &= x_j^{(old)} \quad \text{for } j \neq f \\ x_f^{(new)} &= A_{f,f}^{-1}(b_f - \sum_{j \neq f} A_{f,j} x_j^{(old)}). \end{aligned} \tag{2}$$

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

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 [22] 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, 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.

Experimental Results

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 2. The different lines of Table 2 correspond to: the number of iterations to converge (# iters), the time to solution ($T_{\texttt{Wall}}$), the time of the checkpoint ($T_{\texttt{chkpt}}$), the time lost in the rollback ($T_{\texttt{Rollback}}$), the time for the recovery ($T_{\texttt{Recov}}$), the time for recovering the MPI environment ($T_{\texttt{I}}$), the time for recovering the static data ($T_{\texttt{II}}$) and finally the time for recovering the dynamic data ($T_{\texttt{III}}$).

	Fault-tolerant Strategy					
	without failure			failure at step 10		
	los.	c_F	c_R	los.	c_F	c_R
# iters	18	18	18	18	18	28
$T_{\texttt{Wall}}$	7.98	8.43	8.15	14.11	13.65	16.00
$T_{\texttt{Chkpt}}$		0.52	0.00		0.52	0.00
$T_{\texttt{Rollback}}$						2.29
$T_{\texttt{Recov}}$				5.50	5.19	5.15
$T_{\texttt{I}}$				1.05	1.10	1.11
$T_{\texttt{II}}$				3.94	3.94	3.94
$T_{\texttt{III}}$				0.35	0.13	0.01

Table 2: Comparison of the checkpoint fault-tolerant algorithm and the lossy fault-tolerant algorithm, times are given in seconds, GMRES(30) with Block Jacobi preconditioner with matrix `stomach` from Matrix Market [10] of size $n = 213,360$ on 16 processors.

For this experiment, we shall have the following identities:

$$\begin{aligned} T_{\texttt{Wall}} &= T_{\texttt{Wall}}(\text{lossy}) + T_{\texttt{chkpt}} + T_{\texttt{Rollback}} + T_{\texttt{Recov}}, \\ T_{\texttt{Recov}} &= T_{\texttt{I}} + T_{\texttt{II}} + T_{\texttt{III}}, \end{aligned}$$

$T_{\texttt{II}}$ and $T_{\texttt{III}}$ 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 diagnal dominant and the size of the restart is large then the lossy strategy is in general the best; for small restart size, then c_R is in general better. More experiments including eigenvalue computation can be found in [11].

FTLA provides several implementation of fault-tolerant algorithm for numerical linear algebra algorithm, this enables the program to survive failures by adapting itself to the fault and restart from a coherent state.

7 Communication

Previous studies of application usage show that the performance of collective communications are critical to high-performance computing (*HPC*). Profiling study [55] showed that some applications spend more than eighty percent of a transfer time 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 when compared to the point-to-point performance. A general algorithm for a given collective communication operation may not give good performance on all systems due to 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. In order to determine the optimum parameters of collective communications on a given system the collective communications need to be modeled effectively at some level as exhaustive testing might not produce meaningfull results in a reasonable time as system sizes increase.

7.1 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) operations. 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 fan out 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 utilize is segment size. This is the size of a block of contiguous data that the individual communications can be broken down into. By breaking large single communications into smaller communications and scheduling many communications in parallel it is possible to increase the effiecency of any underlying communication infastructure. Thus for many operations we need to specify both parameters; the virtual topology and the segment size. Figure 5 shows how many crossover points between different implementation can exist for a single collective operation on a small number of nodes when finding the optimal (faster implementation).[1] The number of crossovers demonstrates quite clearly why limiting the number of methods available per MPI operation at runtime can lead to poor performance in many instances 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.

1. Note the logrithmic scale.

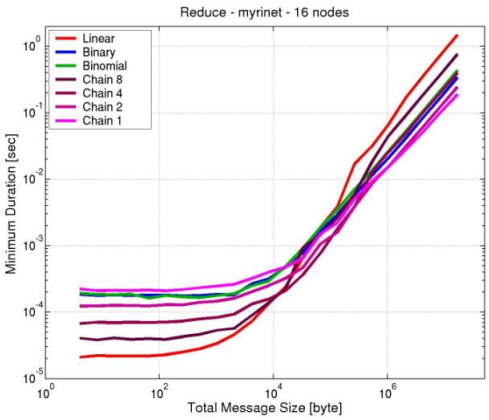

Figure 5: Multiple Implementations of the MPI Reduce Operation on 16 nodes

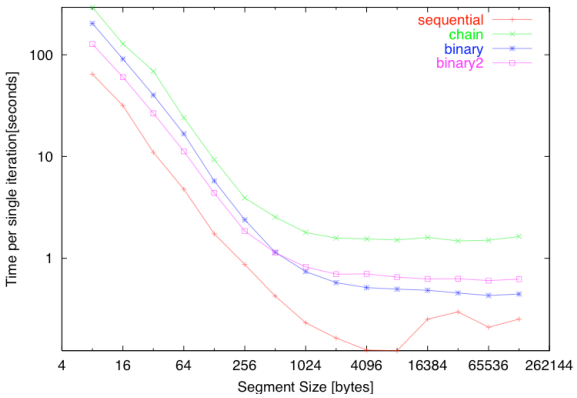

Figure 6: Multiple Implementations of the MPI Scatter operation on 8 nodes for various segmentation sizes

7.2 Exhaustive and directed searching

Simple, yet time consuming method to find 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 [65]. Tuning exhaustively for eight MPI collectives on a small (40 node) IBM SP-2 upto message sizes of one MegaByte involved approximately 13000 individual experiments, and took 50 hours to complete. Even though this only needed to occur once, tuning all of the MPI collectives in a similar manner would take days for a modaratly 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. 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. Reducing the time needed for running the actual experiments can be achieved at many different levels, such as not testing at every point and interpolating results. i.e. testing 8, 32, 128 processes rather than 8, 16, 32, 64, 128 etc. Additionally using instrumented application runs to build a table of only those collective operations that are required, i.e. not tuning operations that will never be called, or are called infrequently. We are currently testing this instrumentation method via a newly developed profiling interface known as the Scalable Application Instrumention System (SAIS).

Another method used to reduce the search space in an intelligent manner is to use traditional optimization techniques such as gradient decent with domain knowledge. Figure 6 shows the performance of four different methods for implementing an 8 processor MPI Scatter for 128 KBytes of data on the 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 decent methods to control runtime tests we can reduce the time to find the optimal segmentation size from 12613 seconds and 600 tests to 40 seconds and just 90 tests [59]. Thus simple methods can still allow semi-exhaustive testing in a reasonable time.

7.3 Communication modelling

There are many parallel communicational models that predict performance of any given collective operation based on standardizing network and system parameters. Hockney [35], LogP [17], LogGP [1], and PLogP [38] models are frequently used to analyze parallel algorithm performance. Assessing the parameters for these models within local area network is relatively straightforward and the methods to approximate them have already been established and are well understood [16][38]. Thakur et al. [63] and Rabenseifner et al. [56] use Hockney model to analyze the performance of different collective operation algorithms. Kielmann

et al. [39] use PLogP model to find optimal algorithm and parameters for topology-aware collective operations incorporated in the MagPIe library. Bell et al. [6] use extensions of LogP and LogGP models to evaluate high performance networks. Bernaschi et al. [7] analyze the efficiency of reduce-scatter collective using LogGP model. Vadhiyar et al. [65][59] used a modified LogP model which took into account the number of pending requests that had been queued. The most important factor concerning communication modelling 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 then can build a mathematical model of a particular collective operation as shown in [36].

Experiments testing these models on a 32 node cluster for the MPI barrier operation using MPICH2 are shown in figure 7. As can be clearly seen, none of the models produce perfect results but do allow a close approximation to the gross performance of the actual implementations. Figure 8 shows the normalised 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. Tests were conducted under FT-MPI 1.2 [23]. 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 measages.

7.4 Collective communication status and future work

Current experiments comparing both the exhaustively tested collective operations and the modelled operations have shown that the choice of the wrong implementation of a collective communication can greatly reduce application performance, while the correct choice can be orders of magnitude better. Currently exhaustive testing is the only sure way to find the absolute optimal combination of parameters that make the optimal implementation. Finding these values exhaustively for anything other than a very small system or a very constrained (and repetative application) is not practical in terms of time to examine the complete search space. We have presented several methods used to reduce or eliminate this search space. Overall we have found that both

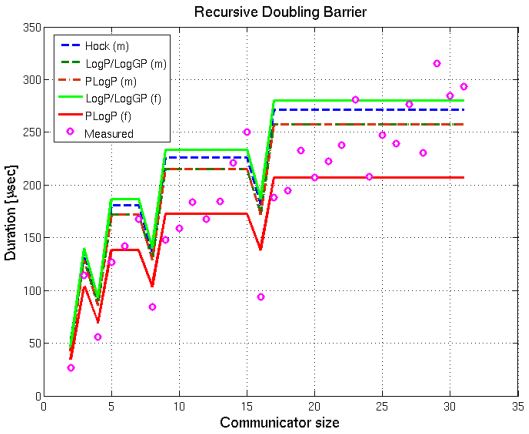

Figure 7: Measured verses modelled MPI Barrier Operation based on recursive doubling

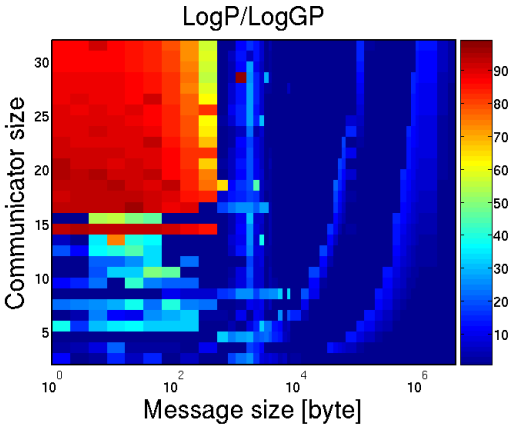

Figure 8: Error (colour) between the exhaustively measured optimal implementation verses implementation choosen using values from the LogP/LogGP modelled broadcast operations

targeted exhaustive tuning and modelling each have their place depending on the target applications and systems. Although modelling provides the fastest solution to deciding which implementation of a collective operation to use, it can still choose a very incorrect result for large node counts on small message sizes, while on larger message sizes it appears very accurate. This is encouraging as 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 mixure of methods will still be used until such times as the collective communication models become more accurate for a wider range of parameters such as node count and data size.

8 Related Work

We list here, briefly, a number of existing projects and their relations to our SANS systems.

The University of Indiana's Linear System Analyzer [43] (LSA) is building a problem solving environment (PSE) for solving large, sparse, unstructured linear systems of equations. It differs from our proposed systems in that it mostly provides a testbed for user experimentation, instead of a system with intelligence built in. A proposed LSA intelligent component (www.extreme.indiana.edu/pseware/lsa/LSAfuture.html) is more built on Artificial Intelligence techniques than numerical analysis.

The TUNE project [64] seeks to develop a toolkit that will aid a programmer in making programs more memory-friendly.

The Berkeley projects PHIPAC, Sparsity, BeBop target optimization of the sparse kernels. Mostly, they use tiling of the sparse matrix to get more effcient operations by the use of small dense blocks. This approach is also used in AcCELS [12]. These operations partly static, partly dynamic, since they depend on the sparsity pattern of the runtime data.

FFTW [30, 25] is a very successful package that tunes the Fast Fourier operations to the user architecture.

An interesting project that combines ideas of dynamic optimization and low-level kernel optimization is Spiral [53] which generates optimal implementations of DSP algorithms.

9 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 fast as the improvements in the 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 main 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 based on the problem characteristics, data attributes, and the state of the grid.

Our efforts together for others in the community for obtaining tuned high-performance kernels, and for automatically choosing suitable algorithms holds great promise for today high-performance systems as well as future systems.

References

[1] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM Press, 1995.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2002.

[3] E. Amaldi and V. Kann. On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems. *Theoretical Computer Science*, 209:237–260, 1998.

[4] E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.

[5] Utpal Banerjee. A Theory of Loop Permutations. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 54–74. Pitman Publishing, 1990.

[6] Christian Bell, Dan Bonachea, Yannick Cote, Jason Duell, Paul Hargrove, Parry Husbands, Costin Iancu, Michael Welcome, and Katherine Yelick. An evaluation of current high-performance networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 28.1. IEEE Computer Society, 2003.

[7] Massimo Bernaschi, Giulio Iannello, and Mario Lauria. Efficient implementation of reduce-scatter in MPI. *J. Syst. Archit.*, 49(3):89–108, 2003.

[8] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.

[9] L. Suzan Blackford, J. Choi, Andy Cleary, Eduardo F. D'Azevedo, James W. Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[10] Ronald Boisvert, Roldan Pozo, Karin Remington, Richard Barrett, and Jack Dongarra. Matrix Market : a web resource for test matrix collections. In R. Boisvert, editor, *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137, Chapman and Hall, London, 1997.

[11] George Bosilca, Zizhong Chen, Jack Dongarra, and Julien Langou. Recovery patterns for iterative methods in a parallel unstable environment. Technical Report UT-CS-04-538, University of Tennessee Computer Science Department, 2004.

[12] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. Technical Report ICL-UT-04-05, ICL, Department of Computer Science, University of Tennessee, 2004.

[13] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Building fault survivable mpi programs with ft-mpi using diskless-checkpointing. In *Proceedings for ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[14] J. Choi, Jack J. Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clint Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[15] Pierluigi Crescenzi and Viggo Kann (editors). A compendium of NP optimization problems. `http://www.nada.kth.se/theory/problemlist.html`.

[16] D. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, 16:35–43, 1996.

[17] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12. ACM Press, 1993.

[18] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[19] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

[20] Victor Eijkhout. Automatic determination of matrix blocks. Technical Report ut-cs-01-458, Department of Computer Science, University of Tennessee, 2001. Lapack Working Note 151.

[21] Victor Eijkhout and Erika Fuentes. A proposed standard for numerical metadata. Technical Report ICL-UT-03-02, Innovative Computing Laboratory, University of Tennessee, 2003. Poster presentation at Supercomputing 2003.

[22] Christian Engelmann and G. Al Geist. Development of naturally fault tolerant algorithms for computing on 100,000 processors. http://www.csm.ornl.gov/~geist.

[23] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and Fault Tolerant MPI. *Journal of Parallel Computing*, 27(11), 2001.

[24] Graham E. Fagg, Edgar Gabriel, George Bosilca, Thara Angskun, Zizhong Chen, Jelena Pjesivac-Grbovic, Kevin London, and Jack J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference*, June 2004.

[25] `http://www.fftw.org`.

[26] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[27] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1), 1995. Available at: `http://www.mpi-forum.org/`.

[28] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 18 July 1997. Available at `http://www.mpi-forum.org/docs/mpi-20.ps`.

[29] Matteo Frigo and Steven G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[30] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.

[31] Edgar Gabriel, Graham E. Fagg, Antonin Bukovsky, Thara Angskun, and Jack J. Dongarra. A fault-tolerant communication library for grid environments. In *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS'03), International Workshop on Grid Computing*, 2003.

[32] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[33] J. Gergov. Approximation algorithms for dynamic storage allocation. In *Proceedings of the 4th Annual European Symposium on Algorithms*, pages 52–56. Springer-Verlag, 1996. Lecture Notes in Computer Science 1136.

[34] D. Hochbaum and D. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approach. *SIAM Journal of Computing*, 17:539–551, 1988.

[35] R.W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994.

[36] George Bosilca Graham E. Fagg Edgar Gabriel Jelena Pjesivac-Grbovic, Thara Angskun and Jack J. Dongarra. Performance analysis of mpi collective operations. In *Proceedings 4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems PMEO-PDS'05*, 2005.

[37] V. Kann. Strong lower bounds on the approximability of some NPO PB-complete maximization problems. In *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science*, pages 227–236. Springer-Verlag, 1995. Lecture Notes in Computer Science 969.

[38] T. Kielmann, H.E. Bal, and K. Verstoep. Fast measurement of LogP parameters for message passing platforms. In José D. P. Rolim, editor, *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 1176–1183, Cancun, Mexico, May 2000. Springer-Verlag.

[39] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140. ACM Press, 1999.

[40] LAM/MPI parallel computing. `http://www.mpi.nd.edu/lam/`.

[41] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *Transactions on Mathematical Software*, 5:308–323, 1979.

[42] J. Lenstra, D. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[43] `http://www.extreme.indiana.edu/pseware/lsa/index.html`.

[44] `http://math.nist.gov/MatrixMarket`.

[45] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. Program. Lang. Syst.*, 18(4):424–453, 1996.

[46] Hans W. Meuer, Erich Strohmaier, Jack J. Dongarra, and Horst D. Simon. *Top500 Supercomputer Sites*, 20th edition edition, November 2002. (The report can be downloaded from `http://www.netlib.org/benchmark/top500.html`).

[47] Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[48] MPICH. `http://www.mcs.anl.gov/mpi/mpich/`.

[49] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 8:308–313, 1965.

[50] David A. Padua and Michael Wolfe. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[51] `http://www-users.cs.umn.edu/~karypis/metis/parmetis/index.html`.

[52] James S. Plank, Youngbae Kim, and Jack Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43:125–138, 1997.

[53] M. Püschel, B. Singer, J. Xiong, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Int'l Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[54] Dan Quinlan, Markus Schordan, Qing Yi, and Andreas Saebjornsen. Classification and untilization of abstractions for optimization. In *Ths First International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, Oct 2004.

[55] Rolf Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the Message Passing Interface Developer's and User's Conference*, pages 77–85, 1999.

[56] Rolf Rabenseifner and Jesper Larsson Träff. More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In *Proceedings of EuroPVM/MPI*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[57] Kenneth J. Roche and Jack J. Dongarra. Deploying parallel numerical library routines to cluster computing in a self adapting fashion. In *Parallel Computing: Advances and Current Issues*. Imperial College Press, London, 2002.

[58] Peter Sanders and Jop F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Information Processing Letters*, 86(1):33–38, 2003.

[59] Graham E. Fagg Sathish S. Vadhiyar and Jack J. Dongarra. Towards an accurate model for collective communications, 2005.

[60] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of EuroPar-2000*, 2000.

[61] Robert Schreiber and Jack Dongarra. Automatic Blocking of Nested Loops. Technical Report CS-90-108, Knoxville, TN 37996, USA, 1990.

[62] Thomas Sterling. *Beowulf Cluster Computing with Linux (Scientific and Engineering Computation)*. MIT Press, October 2001.

[63] Rajeev Thakur and William Gropp. Improving the performance of collective operations in MPICH. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 2840 in LNCS, pages 257–267. Springer Verlag, 2003. 10th European PVM/MPI User's Group Meeting, Venice, Italy.

[64] `http://www.cs.unc.edu/Research/TUNE/`.

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

[66] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[67] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, January 2001.

[68] Qing Yi, Ken Kennedy, Haihang You, Keith Seymour, and Jack Dongarra. Automatic Blocking of QR and LU Factorizations for Locality. In *2nd*

ACM SIGPLAN Workshop on Memory System Performance (MSP 2004), 2004.

[69] Qing Yi and Dan Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.

[70] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A Comparison of Empirical and Model-driven Optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 63–76. ACM Press, 2003.