Automatic Program Instrumentation with Applications in Performance and Error Analysis

B. Bliss, M.-C. Brunet and E. Gallopoulos
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801 U.S.A.

Abstract
This paper describes the design of tools for the automatic instrumentation of programs for tasks as varied as performance, data flow and error analyses. These tools were built to instrument programs written in Fortran 77 or Cedar Fortran, a language supporting fine, medium and large grain parallelism. The paper also describes uses of the instrumentation tool in reporting floating point activity and utilization of vector hardware, generating computational graphs, and producing information about numerical quality based on deterministic and statistical techniques. These tools make possible the immediate application of previously hard to use techniques for investigating the numerical quality of algorithms.

1 Introduction and Summary
With the proliferation of supercomputers the need has arisen to develop appropriate parallel programming environments which will facilitate their efficient usage. Apart from the well-known need for restructuring compilers, good debuggers, visualization tools, etc. the users are faced with an ever-increasing number of tradeoffs that they have to settle for when tuning their application. For instance, the algorithm selection problem becomes very complicated since the performance of the algorithms used is no longer simply a function of the traditional operation count, but also depends on many other factors such as the algorithm's
utilization of the architecture's vector and parallel resources, the trade-offs between less parallelism overhead and more favorable access patterns to the memory hierarchy, etc. An area which has received somewhat less attention, for reasons which are well explained by Skeel in [29], is the development of tools for providing information to the user regarding the quality of numerical results from his algorithm. In view of the flurry of research in developing new algorithms which run effectively on multiprocessors, we feel that such tools would be very valuable.

In this paper we describe the design and use of a system, developed to fulfill some of the aforementioned needs. In summary, the system is based on a program instrumentation preprocessor. This is described in Section 3. This preprocessor can be used for several seemingly different goals, such as obtaining a count of floating-point operations, or obtaining information about the error behavior of an algorithm. In order to achieve these goals, we developed libraries for linking with the preprocessed code. These are:

- **OP.CT**, designed to provide a report of floating-point operation activity. Its design is discussed in Section 4.

- **LGRAPH**, designed to generate a computational graph of a given algorithm. Its design is discussed in Section 5.1.1.

- **PERTURB**, designed to perform a statistical analysis of algorithm stability. Its design is discussed in Section 5.2.

This system has been integrated (along with other tools) in the parallel computational environment outlined in Section 2.

We provide several examples of the use of the preprocessor and libraries: Since the preprocessor performs a source-level instrumentation of the code after any other source-to-source restructuring, we used it for evaluating the effectiveness of a vectorizing restructurer. Some of these experiments are reported in Section 4. Section 5 discusses tools for performing deterministic and statistical error analyses to provide us with reliable indicators of numerical quality. Finally Section 6 discusses current and future work.

The original objective of our efforts was the parallelization of the automatic error analysis tool in ACM TOMS algorithm 594 [23] and its integration in the programming environment proposed in [15]. The mini-compiler provided in [26] was unsuitable for our work. For this purpose we began the development of a tool for generating the computational
graph. During that effort we found that the design could be modularized into two separate components, namely the instrumentation preprocessor, and the LGRAPH library. This modular approach gave the tool greater versatility, leading us to consider other uses. As the Alliant system provided no means of obtaining FLOP counts, we developed such a library. As we started realizing some of the limitations (discussed in Section 5.3) for the error analysis based on the approach of TOMS 594, we became interested in investigating the advantages and limitations of the statistical approach. PERTURB is the basis of that effort. More details, experiments and examples from this work can be found in [3, 4].

2 Computational Environment

The computational environment consists of an Alliant FX/8 multiprocessor and Cedar, the hierarchical cluster-based vector multiprocessor developed at the University of Illinois Center for Supercomputing Research and Development [13]. With vector, concurrent and multicluster computational capabilities, Cedar allows the exploitation of fine, medium and coarse level parallelism.

The Cedar programming environment is based on the Cedar Restructurer and Cedar Fortran [12], an extension of the Alliant FX/8 Fortran [1]. The latter is Fortran 77 augmented with vector constructs, such as those proposed for the next Fortran standard, as well as constructs to support concurrency across computational elements. To produce Cedar Fortran, the language was further extended with constructs to support cluster level parallelism, global memory, and extra synchronization operations. The Cedar Restructurer is a CSRD-modified version of the KAP restructurer\(^1\) [20]. Another restructurer designed for optimizing only at the levels of vector and concurrent processing is VAST\(^2\), the native Alliant FX/8 optimizer. The components of the CSRD/Cedar environment for Fortran are depicted in Figure 1.

Another component of the CSRD computational environment is the integration of many of the available tools. This X-window based parallel programming environment is described in [15].

---

\(^1\)KAP is a registered trademark of Kuck and Associates, Inc.

\(^2\)VAST is a registered trademark of the Pacific Sierra Research Corporation.
3 Preprocessor Design

The kernel of the instrumentation is a preprocessing phase of the input source code. This consists of steps similar to those of a vector compiler. Specifically, it translates the source code into 3-address statements, creating temporary variables to hold intermediate results. For vector operations, it generates code for the calculation of an operand's length and memory stride. After the code is transformed into 3-address statements, command line options may be specified to replace the operations and intrinsic function calls on a specified data type with user-defined subroutine calls. The program is then linked with a library which performs the basic arithmetic operations and contains the code necessary to carry out the instrumentation.
This preprocessing step is included as a pass of the Cedar Fortran preprocessor, CFTN. CFTN translates Cedar Fortran into Alliant Fortran, as described in Section 2. It is invoked with the -z option, and will be referred to as the -z pass when it becomes necessary to distinguish it from other passes to CFTN, or from CFTN itself. The -z pass is based upon a standard syntax-directed translation scheme, using the LALR(1) parser generator, YACC [19]. CFTN first scans the input stream, breaking it up into tokens, using a modified version of the UNIX Fortran F77 scanner. Implementing a LALR(1) grammar to parse the resulting token stream is relatively straightforward. The translation of the Cedar Fortran extensions requires that the -z pass be imbedded in CFTN, rather than existing as a separate module.

3.1 Preprocessor options

If the preprocessor is invoked without any options, the behavior of the target code is equivalent to the original source code. Transforming arithmetic operations into user-defined subroutine calls is done as follows: With each data type we associate a V_token: a two letter identifier used in the command line arguments, internally in the preprocessor, and in the naming convention of the subroutines. The option -B<V_token> causes the preprocessor to mark data objects and operations of that type. It is common to use the options -BR4, -BR8, -BC8, and -BDC, and thus force the preprocessor to replace operations on 4-byte reals, 8-byte reals, 8-byte complex data, and 16-byte complex data (DOUBLE COMPLEX) with subroutine calls.

3.2 Subroutine Calling Conventions

With each operation or intrinsic function is associated a U_token, a string to be used together with the appropriate V_token to form the naming convention of the subroutines. The U_token SUB, for example, represents a binary subtraction. A subroutine replacing an arithmetic operation is called

\[ \texttt{zz<dim1>}[<dim2>]<V_token><U_token> \]

where \( dim_1 \) and \( dim_2 \) are digits representing the rank of the left and right (if one exists) operands, respectively. The operation, specified by \( U_token \), is performed upon operands with data type \( V_token \). In the
special case of intrinsic functions, the number of rank-specifying digits is equal to the number of function arguments.

The arguments are passed as follows. First, for operations on array operands, the vector length(s) appears first in the parameter list. The next argument is a slot to which the subroutine should assign the result of the operation. If this slot is an array expression, it is followed by the memory stride(s) used in accessing it. The operands (or arguments to intrinsics) follow, together with their respective memory strides, should they be arrays. Some intrinsics (e.g. \texttt{min} and \texttt{max}) adhere to different parameter passing conventions. Note that since in Fortran parameters are passed by reference, the subroutine can access all elements of an arbitrary cross-section of an \(n\)-dimensional array. These points are illustrated in the Appendix of [4].

An important difference of designing the preprocessor to act at the Fortran source level (instead of the assembler) is that it allows a greater amount of information to be used in preparing instrumentation for some applications. Such information is propagated in the form of an arbitrary number of variables associated with data objects of a specified type. These \textit{associated variables} are of the same shape and size as the original variable and follow it in the subroutine argument list. It is thus important that the data types of actual and formal parameters match when using this option. This feature is used in the implementation of the \texttt{LGRAPH} package, described in Section 5.1.1.

We finally note that because of recent work in the area of programming environments (e.g. [14]), future versions of these tools could become more accurate by incorporating information from all the levels of the compilation hierarchy, i.e. source, restructured, and assembler level.

4 Floating Point Count Activity With \texttt{OP\_CT}

To obtain information about a program's floating point activity, the \texttt{OP\_CT} library module was developed. First, the \texttt{-z} pass replaces all floating point operations and intrinsic functions with subroutine calls as described in Section 3. The user then links and compiles with \texttt{OP\_CT}. Each subroutine in the library performs the appropriate operation and also stores information regarding the type of operation, the vector length, and vector stride(s). Implementation details and examples can be found in [4].

The floating-point activity report is generated by a special library
subroutine automatically inserted at the end of the program by the -z pass. This report breaks down each operation, intrinsic function call, and type conversion (explicit or implicit) according to data type, and whether the operation was performed upon scalar or vector operands (and in the case of a binary or intrinsic operation, whether there were one or two vector operands). For vector operations and intrinsics, the average vector lengths and strides of the arrays accessed are also calculated. When calculating the average vector length, it is assumed that vector operations are stripmined into blocks of 32 elements, since this is the length of each vector register in the Cedar cluster\footnote{If the hardware were to change, this value can trivially be set to reflect the change.}.

Such reports can be used to analyze the effectiveness of the vectorizing restructurers. We ran the LINPACK routines DGECO and DGEQL (\cite{10}) for standard Gaussian elimination on a 256 $\times$ 256 system of linear equations through the KAP based Cedar Restructurer, confining its transformations to vectorization only. Figure 2 shows the resulting report after using OP\_CT. In \cite{3}, OP\_CT was used to compare the performance of KAP with the Alliant VAST restructurer.

In order for OP\_CT to correctly reflect the number of operations performed when multiprocessing is in effect (e.g. parallel operation of the computational elements of a Cedar cluster or parallel operation of the clusters), the counters are included within critical sections of code. We note however that the generated report gives no indication that any of the computation was performed concurrently.

The OP\_CT module was incorporated in the X-window based parallel programming environment of \cite{15}. A feature of this implementation is the ability to focus on portions of the code, selected by clicking the mouse: the generated report gives the operation count only for the selected portion. Such a feature is important in the construction of hierarchical instrumentation tools.

5 Studies in Numerical Quality

As mentioned in the Introduction, an area which deserves greater study in the context of parallel programming environments is the design of tools for the assessment of the numerical quality, a need made even more urgent with the advent of vector and multiprocessor computers. We provide the user with easy access to existing and newly developed software
### REAL * 8 ASSIGNMENTS

<table>
<thead>
<tr>
<th>assignment</th>
<th>scalar count</th>
<th>vec/sca total vector/ scalar ops</th>
<th>av vec length</th>
<th>vec/vec total vector/ vector ops</th>
<th>av vec length</th>
<th>operation total</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>125194</td>
<td>8</td>
<td>256</td>
<td>1.0</td>
<td>32.0</td>
<td>218948</td>
</tr>
<tr>
<td>totals</td>
<td>125194</td>
<td>8</td>
<td>256</td>
<td>1.0</td>
<td>32.0</td>
<td>218948</td>
</tr>
</tbody>
</table>

### REAL * 8 OPERATIONS

<table>
<thead>
<tr>
<th>operation</th>
<th>scalar count</th>
<th>vec/sca total vector/ scalar ops</th>
<th>av vec length</th>
<th>vec/vec total vector/ vector ops</th>
<th>av vec length</th>
<th>operation total</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparisons</td>
<td>36221</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>+</td>
<td>1740</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>binary -</td>
<td>808</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>218400</td>
</tr>
<tr>
<td>unary -</td>
<td>1278</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>38</td>
<td>220092</td>
<td>5799330</td>
<td>2.5</td>
<td>26.3</td>
<td>218400</td>
</tr>
<tr>
<td>/</td>
<td>1320</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
</tr>
<tr>
<td>totals</td>
<td>41405</td>
<td>220092</td>
<td>5799330</td>
<td>2.5</td>
<td>26.3</td>
<td>218400</td>
</tr>
</tbody>
</table>

Unary operations are listed in vector/Scalar columns.
### REAL * 8 INTRINSIC FUNCTION CALLS

<table>
<thead>
<tr>
<th>intrinsic function</th>
<th>scalar count</th>
<th>vec/sca count</th>
<th>scalar ops</th>
<th>av vec stride</th>
<th>av vec length</th>
<th>vec/vec count</th>
<th>total vector/ops</th>
<th>av vec stride</th>
<th>av vec length</th>
<th>operation total</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs ()</td>
<td>2375</td>
<td>5772</td>
<td>164480</td>
<td>1.0</td>
<td>28.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>166855</td>
</tr>
<tr>
<td>dotproduct ()</td>
<td>0</td>
<td>1332</td>
<td>32640</td>
<td>1.0</td>
<td>24.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32640</td>
</tr>
<tr>
<td>firstmaxoffset ()</td>
<td>0</td>
<td>1144</td>
<td>32640</td>
<td>1.0</td>
<td>28.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32640</td>
</tr>
<tr>
<td>max () +</td>
<td>256</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>2.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>256</td>
</tr>
<tr>
<td>sign ()</td>
<td>255</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>sum ()</td>
<td>0</td>
<td>4623</td>
<td>131840</td>
<td>1.0</td>
<td>28.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131840</td>
</tr>
</tbody>
</table>

**totals**
- 2886 scalar ops
- 12876 vector/ops
- 361600 total
- 1.0 av vec stride
- 27.0 av vec length
- 131840 operation total

+ min / max are not vector functions
vec length is # of args
totals, non-elemental and unary intrinsics are listed in vector/scalar columns

**Figure 2:**

OP_CT report for Gaussian elimination using LINPACK routines optimized with the KAP restructurer. The system size is 256 x 256.
for analyzing a code's numerical quality. We are motivated by the fact that much of the currently available software for this work has been quite difficult to use. Our tools will thus allow the collection of valuable data for the better understanding of the advantages and limitations of several automatic methods for estimating numerical quality.

The numerical inaccuracies in the computed results are due to the effect of roundoff in the computation and/or inaccurate representation of input data. The relative magnitude of the effect of each of these errors for a given algorithm and input data will provide information on the numerical stability of the algorithm. One wants to use algorithms for which the error magnification due to the computation is not much worse than the error due to the data. One is also interested in comparing the total effect of roundoff for different algorithms.

We concentrate on two methods ([24, 6]), sufficiently different in theory, to demonstrate the design and use of our tools. Both methods provide information for the neighborhood of the computed result and automate approximate versions of Wilkinson-style backward error analysis. Although both approaches have been adequately described in these references, we provide a short outline to make easier it for the reader to follow the experiments of Section 5.3.

5.1 The Deterministic Approach: ACM TOMS Algorithm 594

The following summary is taken largely from [21]. The program is represented as a computational graph, that is a directed acyclic graph (DAG), with arcs from results to operands, where each node is either an input, or an output or a intermediate result [2, 24]. Each node is subject to local relative error due to representation or roundoff. These errors accumulate as they propagate forward and generate total relative error. The total relative error consists of two components, one due to those errors entirely due to computational roundoff, the other due to errors entirely due to inaccurate input data. We can thus write $t = \tilde{A}d + \tilde{B}g$ where $d$ and $g$ are the vectors of local relative errors in the data and computational nodes respectively, and $t$ is the vector of total relative errors in the output nodes. $A$ and $B$ are matrices of partial relative derivatives specifying the effect of the local relative errors. Since both $\|d\|$ and $\|g\|$ are at most 1, it follows that $\|t\| \leq \|\tilde{A}\| + \|\tilde{B}\|$. Componentwise this inequality means that $|t_j| \leq \tau_j$ where $\tau_j \equiv \|\tilde{A}_{j,:}\| + \|\tilde{B}_{j,:}\|$. If $\eta_j$ denotes an exact output, then the computed output $\tilde{\eta}_j$ satisfies $\tilde{\eta}_j = (1 + \tau_j \epsilon)\eta_j$. Moreover $\|\tilde{A}_{j,:}\|$
and $\| \overline{B}_j \|$ are the condition of the problem and the condition of the algorithm for output $j$. Then $t_A \equiv (\| \overline{A}_j \|)_{j=1,n}$ and $t_B \equiv (\| \overline{B}_j \|)_{j=1,n}$ are the condition vectors of the problem and of the algorithm respectively. To investigate stability TOMS 594 uses a modified form of backward error analysis, called $B$-analysis. As hinted in Section 5, it tries to find the smallest $\theta$ such that for all solution vectors, $\theta' t_A$ is near (but not necessarily equal to) $t_B$, where $\theta' \leq \theta$. The above steps were consolidated into ACM TOMS Algorithm 594 [23]. TOMS 594 expects as input an encoded form of the computational graph and produces a report containing componentwise information ($\{ \tau_j \}, t_A, t_B$) from the forward error analysis step, and the stability measure $\theta$ from the $B$-analysis. In the next section we discuss the design of LGRAPH, a library module for the generation of the encoded form of the computational graph required for TOMS 594.

5.1.1 Generation of the computational graph: LGRAPH

To generate the computational graph, the program is first run through the preprocessor with each floating point operation replaced with a library subroutine call, as is done with OP CT (and PERTURB). Unlike the other modules however, generating such a graph also requires that the result of each operation (as well as constants and variables read during program execution) be assigned a unique identifier distinguishing it as a unique node in the DAG.

Variable addresses could serve as identifiers. This is possible since arguments are passed by reference in Fortran. Complications arise when an assignment statement or function call copies the result of an operation to two or more addresses, or with the run-time stack space being reused, so that a single address can reference the result of two or more distinct operations. Moreover, Fortran doesn't allow access to the address of an argument. The preprocessor provides a solution to this problem: with each variable of a specified data type, it creates an associated variable (cf. Section 3.2). This is passed along with the original variable to the subroutine calls which replace the arithmetic operations. Note that this method effectively converts data entities into record structures, which can be operated upon by the subroutines of the library module.

This associated variable uniquely identifies the result of each operation; it indexes a global data structure representing the DAG of the computation performed so far.

A call to a library routine is automatically inserted by the prepro-
cessor at the end of the program, as was done for OP_CT (cf. 4). This produces the computational graph, in the format required by TOMS 594 and TOMS 532. These points are illustrated in the examples and Appendix of [4].

We next note that W. Miller and collaborators had built a “minicomputer” [26], to produce the computational graph as input to TOMS 532. Unfortunately, the source was expected to be in a severely restricted Fortran-like language. This was in part due to the fact that the graph had to be obtained at compile-time. This was also restricting TOMS 594, which was based on the minicompiler for generating its DAG.

5.2 The Statistical Approach and PERTURB

In Section 5.1 we described a “deterministic” method for error analysis. In this section, we describe the design of an approach based on statistical techniques implemented as the module PERTURB. The basic method is described in detail in [6]. To speak in terms of a physical analogy, it is based on introducing “random noise” in the computation and then measuring its “echo”. The result of every floating-point operation is randomly perturbed in a relative way, within the machine epsilon (\(\epsilon\)) of the computer.

By repeating this process \(N\) times a corresponding sample is produced for each output of the program. Numerical quality information is then obtained from sample statistics. Consider for example the general problem: Given input data vector \(y\), solve

\[ F(x) = y \]

for \(x\). Let \(x\) be a vector of length \(n\). If the result of the algorithm is \((X_i)_{i=1,n}\), then the method generates one sample \(\{X_i^k\}_{k=1,N}\) for each component \(X_i\). We can therefore compute the empirical mean \(\bar{X}_i\) and empirical standard deviation \(\delta_i\) for each result:

\[ \bar{X}_i = \frac{1}{N} \sum_{k=1}^{N} X_i^k, \quad \delta_i^2 = \frac{1}{N-1} \sum_{k=1}^{N} (X_i^k - \bar{X}_i)^2. \]

We let \(m \equiv (\bar{X}_1, \ldots, \bar{X}_n)\) and \(\sigma \equiv (\delta_1, \ldots, \delta_n)\) be the corresponding empirical mean and standard deviation vectors for the data. We first note that intuitively, a large \(\delta_i\) (resp. \(\delta_i/\bar{X}_i\)) is an indication of a significant propagation and accumulation of absolute (resp. relative) error in the \(i^{th}\) result. In order to obtain further information about numerical
stability we need information about the residual \( y - F(X) \). This requires access to the map \( F(.) \), which is the multivariable vector function representing the problem. If this is available, we can also produce the sample \( \{ F(X^k_i) \}_{k=1}^N = \{ F(X^k_1, X^k_2, ..., X^k_i) \}_{k=1}^N \). We can then compute the corresponding empirical mean and standard deviation vectors \( \rho \) and \( u \). Then \( \|u\| \) represents a distance between the original problem and the problem we have actually solved, and can be used as a measure of the numerical stability of the algorithm. If (and only if) this quantity \( \|u\| \) is small (stable algorithm) then \( \|\sigma\|/\|u\| \) will be a measure of the condition of the problem. This approach has been studied in [6] and utilized in the solution of linear systems and eigenvalue problems.

Consider solving \( Ax = y \). In this case the residual is trivially computable. We define \( m, \sigma, \rho, u \) as before. Furthermore define \( \hat{u} \) by \( \|\hat{u}\|^2 \equiv \|u\|^2 + \|\rho\|^2 \).

The indicators of stability, written in relative form, are the following:

- \( StabA = \|\hat{u}\|/\|u\| \)
- \( StabP = \|\sigma\|/\|\hat{u}\| \)

Note that the derivation of the formula \( StabP \) is based on certain approximation steps explained in [6].

5.2.1 The design of perturb

The \(-z\) pass of the preprocessor instruments the program by replacing floating-point arithmetic with subroutine calls. This is done in nearly the same manner as for \texttt{OP\_CT}, except that it is not necessary to call subroutines to perform floating-point computation when the result only affects the control flow of the program (e.g. comparison operations). The program is then linked with the \texttt{PERTURB} library module.

The perturbation method requires that the program be executed a number of times. The results of these runs are subsequently collected together to produce the sample statistics and any further manipulations.

The subroutines of the \texttt{PERTURB} library module first calculate the result of the appropriate operation and then perturb the result by the machine \( \epsilon \), under the following rule: The value is increased or decreased relatively by \( \epsilon \) with equal, 0.25 probabilities, and remains unperturbed with probability 0.5.

We note that these rules can be changed by modifying the library. The generation of random numbers and the perturbation of the result
are performed entirely in vector mode. Additionally, any routine in the library may be called concurrently, so the package does not inhibit parallelism whatsoever.

5.3 Use and Comments

We can now use the relevant instrumentation tools to conduct experiments leading to the better understanding of the deterministic and statistical methods of Sections 5.1 and 5.2. We thus applied LGRAPH, TOMS 594 and PERTURB to obtain indicators of numerical quality for the solution of linear systems with Gaussian elimination and compare these indicators with those obtained with other approaches reported in the literature. This would test the strictness of the "upper bound" indicators of TOMS 594 and the accuracy of the "statistical" indicators of PERTURB.

We note that without the tools, the preparation of these experiments would have been very tedious.

We compared with the usual exact and estimated condition numbers:

- \( K_\infty = \|A\|_\infty \|A^{-1}\|_\infty \),
- \( K_2 = \|A\|_2 \|A^{-1}\|_2 = \sigma_1 / \sigma_n \) where \( \sigma_i, i = 1, n \) are the \( n \) singular values of \( A \), and \( \sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0 \),
- The estimate of \( K_1 = \|A\|_1 \|A^{-1}\|_1 \) proposed by N. Higham in [17].

We also compared with various effective condition numbers:

- \( Ecn = \min_k \left\{ \frac{\sigma_{n-k+1}}{\sigma_n} \left( \frac{\|P_k b\|_2}{\|b\|_2} \right)^{-1} \right\} \), where \( P_k = U_k U_k^T \) for \( 1 \leq k \leq n \) is a projection vector, \( U_k = (u_{n-k+1}, ..., u_n) \in \mathbb{R}^{n \times k} \) whose columns are the \( k \) last left singular vectors \( (A = U \Sigma V^T \) and \( U = (u_1, ..., u_n)^T \) as proposed by Chan and Foulser [8];
- \( Skeel(x) = \|A^{-1}x\|_\infty / \|x\|_\infty \) was proposed by Skeel [30].

Remarks

1. Since \( P_n b = b \), it follows:

\[
Ecnn \leq \frac{\sigma_1}{\sigma_n} (= K_2)
\]

2. The maximum value of \( Skeel(x) \) is attainable for \( x \equiv e = (1, ..., 1)^T \):

\[
Skeel(e) = \|A^{-1} \cdot |A| \|_\infty
\]
The indicators from TOMS 594 and PERTURB were obtained on a single cluster of Cedar (i.e. an Alliant FX/8 - cf. Section 2), using double precision arithmetic satisfying the IEEE standard (the machine epsilon \( \epsilon \) is of the order of \( 10^{-16} \)). All other indicators were obtained on a Sun 3/50 using MATLAB [27].

5.3.1 Experimental Data

We will use the following notations for vectors (of size \( n \)):

- \( e = (1, 1, ..., 1)^T \)
- \( e_1 = (1, 0, ..., 0)^T \)
- \( t = (1, 2, ..., n)^T \)

The first two matrices were provided by the software of N. Illyahm [18].

Example 1: \( A_1 \) is a Vandermonde matrix.

\[
A_1 = \begin{pmatrix}
1 & 1 & \ldots & 1 \\
1 & 2 & \ldots & 2^n \\
\vdots & \vdots & \ddots & \vdots \\
1 & 2^{n-1} & \ldots & 2^{n-1}
\end{pmatrix}
\]

with \( a_i = 1/2^i \) (and \( x = t \)).

Example 2: \( A_2 = kahan(n, 3) \) is a matrix constructed by W. Kahan (and \( x = t \)). It was taken from [18].

Example 3: The following matrix was taken from [16].

\[
A_3 = \begin{pmatrix}
1 & -100 & -100 & -100 \\
0 & 1 & -100 & -100 \\
0 & 0 & 1 & -100 \\
0 & 0 & 1 & 1
\end{pmatrix}
\]

with \( b = (-299, -199, -99, 1)^T \) (and \( x = e \)), or with \( b = x = e_1 \).

Example 4: The matrix \( A_4 \) is diagonally dominant and taken from [22].

\[
A_4 = \begin{pmatrix}
4 & 1 & 1 & 1 \\
2 & -5 & 1 & 1 \\
1 & 2 & 6 & 1 \\
3 & 1 & 2 & -7
\end{pmatrix}
\]

with \( b = (13, -1, 27, -17)^T \) (and \( x = t \)).
5.3.2 Results, comments and some comparisons

We first used \texttt{PERTURB} for the matrix $A_1$ (for matrix sizes $n = 4, \ldots, 15$) and $A_2$ (for matrix sizes $n = 5, \ldots, 8$). The systems were solved by means of the \texttt{LINPACK} subroutines for Gaussian elimination with partial pivoting. All the aforementioned usual and effective condition numbers together with the two indicators $StabA$ and $StabP$ produced by \texttt{PERTURB} (for a sample size $N = 20$) are reported in Table 1, Figure 3 and Table 2, Figure 4, respectively.

**Table 1:** Condition numbers for the Vandermonde matrix $A_1$ and $x = t$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$K_\infty$</th>
<th>$Higham$</th>
<th>$Skeel(e)$</th>
<th>$Skeel(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.720E+03</td>
<td>1.371E+03</td>
<td>2.065E+02</td>
<td>7.475E+01</td>
</tr>
<tr>
<td>5</td>
<td>6.789E+04</td>
<td>2.721E+04</td>
<td>1.887E+03</td>
<td>4.684E+02</td>
</tr>
<tr>
<td>6</td>
<td>2.910E+06</td>
<td>9.704E+05</td>
<td>3.188E+04</td>
<td>5.895E+03</td>
</tr>
<tr>
<td>7</td>
<td>2.295E+08</td>
<td>6.558E+07</td>
<td>1.050E+06</td>
<td>1.575E+05</td>
</tr>
<tr>
<td>8</td>
<td>3.450E+10</td>
<td>8.626E+09</td>
<td>6.820E+07</td>
<td>8.731E+06</td>
</tr>
<tr>
<td>9</td>
<td>1.007E+13</td>
<td>2.239E+12</td>
<td>8.796E+09</td>
<td>9.890E+08</td>
</tr>
<tr>
<td>10</td>
<td>5.770E+15</td>
<td>1.154E+15</td>
<td>2.261E+12</td>
<td>2.274E+11</td>
</tr>
<tr>
<td>11</td>
<td>6.502E+18</td>
<td>1.182E+18</td>
<td>1.156E+15</td>
<td>1.054E+14</td>
</tr>
<tr>
<td>12</td>
<td>1.966E+22</td>
<td>3.277E+21</td>
<td>1.601E+18</td>
<td>1.336E+17</td>
</tr>
<tr>
<td>13</td>
<td>3.019E+22</td>
<td>5.735E+21</td>
<td>1.859E+18</td>
<td>1.432E+17</td>
</tr>
<tr>
<td>14</td>
<td>4.088E+22</td>
<td>9.233E+21</td>
<td>8.269E+17</td>
<td>5.911E+16</td>
</tr>
<tr>
<td>15</td>
<td>7.169E+22</td>
<td>1.644E+22</td>
<td>8.652E+17</td>
<td>5.773E+16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>$K_2 = \sigma_1/\sigma_n$</th>
<th>$Ecn$</th>
<th>$StabP$</th>
<th>$StabA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.148E+03</td>
<td>1.148E+03</td>
<td>3.230E+02</td>
<td>1.961E+00</td>
</tr>
<tr>
<td>5</td>
<td>2.494E+04</td>
<td>2.494E+04</td>
<td>2.848E+03</td>
<td>2.123E+00</td>
</tr>
<tr>
<td>6</td>
<td>9.668E+05</td>
<td>9.668E+05</td>
<td>5.327E+04</td>
<td>2.597E+00</td>
</tr>
<tr>
<td>7</td>
<td>7.032E+07</td>
<td>7.032E+07</td>
<td>2.555E+06</td>
<td>2.417E+00</td>
</tr>
<tr>
<td>8</td>
<td>9.868E+09</td>
<td>9.868E+09</td>
<td>1.247E+08</td>
<td>2.935E+00</td>
</tr>
<tr>
<td>9</td>
<td>2.712E+12</td>
<td>2.712E+12</td>
<td>1.969E+10</td>
<td>3.143E+00</td>
</tr>
<tr>
<td>10</td>
<td>1.472E+15</td>
<td>1.472E+15</td>
<td>7.205E+12</td>
<td>3.081E+00</td>
</tr>
<tr>
<td>11</td>
<td>1.584E+18</td>
<td>1.584E+18</td>
<td>3.624E+15</td>
<td>3.517E+00</td>
</tr>
<tr>
<td>12</td>
<td>3.206E+20</td>
<td>6.292E+18</td>
<td>1.589E+15</td>
<td>1.311E+01</td>
</tr>
<tr>
<td>13</td>
<td>1.951E+21</td>
<td>2.596E+19</td>
<td>2.037E+15</td>
<td>1.385E+01</td>
</tr>
<tr>
<td>14</td>
<td>5.618E+21</td>
<td>3.958E+19</td>
<td>8.950E+14</td>
<td>3.726E+01</td>
</tr>
<tr>
<td>15</td>
<td>3.025E+21</td>
<td>9.969E+19</td>
<td>3.369E+14</td>
<td>1.236E+02</td>
</tr>
</tbody>
</table>

For the remaining matrices we also used \texttt{TOMS 594}. We define the summary indicators $CA \equiv \|t_B\|$ to be the condition of the algorithm and
Table 2: Condition numbers for the Kahan matrix $A_2$ and $x = t$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$K_\infty$</th>
<th>Higham</th>
<th>Skeel($e$)</th>
<th>Skeel($x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.415E+04</td>
<td>5.812E+03</td>
<td>8.920E+00</td>
<td>5.744E+00</td>
</tr>
<tr>
<td>6</td>
<td>1.203E+05</td>
<td>4.119E+04</td>
<td>1.090E+01</td>
<td>6.767E+00</td>
</tr>
<tr>
<td>7</td>
<td>9.939E+05</td>
<td>2.919E+05</td>
<td>1.288E+01</td>
<td>7.780E+00</td>
</tr>
<tr>
<td>8</td>
<td>8.047E+06</td>
<td>2.068E+06</td>
<td>1.486E+01</td>
<td>8.787E+00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>$K_2 = \sigma_1/\sigma_n$</th>
<th>$Ecn$</th>
<th>StabP</th>
<th>StabA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>7.941E+03</td>
<td>7.941E+03</td>
<td>6.985E+00</td>
<td>1.644E+00</td>
</tr>
<tr>
<td>6</td>
<td>6.165E+04</td>
<td>6.165E+04</td>
<td>8.787E+00</td>
<td>1.926E+00</td>
</tr>
<tr>
<td>7</td>
<td>4.720E+05</td>
<td>4.720E+05</td>
<td>1.131E+01</td>
<td>1.596E+00</td>
</tr>
<tr>
<td>8</td>
<td>3.576E+06</td>
<td>3.576E+06</td>
<td>1.104E+01</td>
<td>2.061E+00</td>
</tr>
</tbody>
</table>

$CP \equiv \|t_A\|$ to be the condition of the problem. Results are reported in Tables 3 to 5. For Perturb, we also present the indicators for varying sample size ($N = 20, 50, 75, 100$). For the matrix $A_3$ we report experiments with two different right-hand sides in Table 3 and 4, and for $A_4$ in Table 5.

We compare $CP$, StabP and all other condition numbers for the stability of the problem, $\theta$ and StabA for the numerical stability of the algorithm: there is no analogue of CA (measure of algorithm condition) in Perturb.

From the experiments we infer the following:

- Figures 3 and 4 show that StabP keeps close to the effective condition number Skeel($x$).

- Figure 3 shows that StabP is a reliable indicator of the stability of the problem as long as StabA is small.

- Tables 3 and 4 show how the change in the right-hand side affects the condition numbers. The usual condition numbers do not reflect this fact since they only depend on the matrix $A$. For the second right-hand side, both methods report an impressively smaller condition number, as predicted also in Skeel($x$). The algorithm remains stable in both cases and this is accurately reported by both methods through the $\theta$ and StabA indicators.

- For a well-conditioned matrix ($A_4$), all the condition numbers are close (see Table 5).
Figure 3: Condition numbers for the Vandermonde matrix $A_1$.

- A sample size larger than $N = 20$ does not cause any significant differences to $StabA$ or $StabP$ (cf. Tables 3, 4 and 5).

Extensive experiments and comparisons will be provided in a forthcoming paper [7].

For the benefit of the user, we provide some comparative observations for the two methods. The approach of TOMS 532 can apply directly to any code, provided the input data is available. It also provides useful componentwise information. We mentioned in Section 5 that the results of the forward error analysis can be used to help select an algorithm. Larson in [22] shows that additional comparative information can be obtained from $\theta$. For an algorithm with $r$ computations, $n$ inputs and $m$ outputs, the complexity of computing $\tilde{A}, \tilde{B}$ is $O(mn + mr)$ operations.
Figure 4: Condition numbers for the Kahan matrix $A_2$.

and storage. To this must be added the cost for the more expensive B-analysis step. Hence the complexity can quickly become unmanageable. It thus seems that this approach is more suitable for studying numerical quality of localized portions of a program. Since the analysis is based on linearization of the error effects and dropping terms of the form $\epsilon^2$, it is not clear how a program combining both single and double precision arithmetic could be analyzed.

PERTURB can also be used on any algorithm. Moreover, its cost is

<table>
<thead>
<tr>
<th>TOMS 594</th>
<th>CA</th>
<th>$\theta$</th>
<th>CP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.229E+07</td>
<td>2.998E+00</td>
<td>4.101E+06</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PERTURB</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N</td>
<td>StabA</td>
<td>StabP</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>2.795E+00</td>
<td>8.857E+05</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>2.488E+00</td>
<td>9.312E+05</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>2.449E+00</td>
<td>9.523E+05</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>2.449E+00</td>
<td>9.143E+05</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>$K_\infty$</th>
<th>$Skeel(\epsilon)$</th>
<th>$Skeel(x)$</th>
<th>$K_2 = \sigma_1/\sigma_n$</th>
<th>$Ecn$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.101E+08</td>
<td>2.061E+06</td>
<td>2.061E+06</td>
<td>2.288E+08</td>
<td>1.020E+06</td>
</tr>
</tbody>
</table>
Table 4: Results obtained for the matrix $A_3$ and $x = e_1$.

<table>
<thead>
<tr>
<th>TOMS 594</th>
<th>CA</th>
<th>$\theta$</th>
<th>CP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.532E+01</td>
<td>3.500E+00</td>
<td>6.204E+00</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>StabA</td>
<td>StabP</td>
</tr>
<tr>
<td>PERTURB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1.571E+00</td>
<td>1.00130E+00</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1.616E+00</td>
<td>1.00010E+00</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>1.537E+00</td>
<td>1.00027E+00</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1.557E+00</td>
<td>1.00019E+00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>$K_\infty$</th>
<th>Skeel(e)</th>
<th>Skeel(x)</th>
<th>$K_2 = \sigma_1/\sigma_n$</th>
<th>Ecn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.101E+08</td>
<td>2.061E+06</td>
<td>1</td>
<td>2.288E+08</td>
<td>2.288E+08</td>
</tr>
</tbody>
</table>

Table 5: Results obtained for the matrix $A_4$ and $x = t$.

<table>
<thead>
<tr>
<th>TOMS 594</th>
<th>CA</th>
<th>$\theta$</th>
<th>CP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.912E+01</td>
<td>3.033E+00</td>
<td>1.266E+00</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>StabA</td>
<td>StabP</td>
</tr>
<tr>
<td>PERTURB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>4.227E+00</td>
<td>0.953E+00</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>4.182E+00</td>
<td>0.858E+00</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>3.938E+00</td>
<td>0.856E+00</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>3.766E+00</td>
<td>0.859E+00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MATLAB</th>
<th>$K_\infty$</th>
<th>Skeel(e)</th>
<th>Skeel(x)</th>
<th>$K_2 = \sigma_1/\sigma_n$</th>
<th>Ecn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.392E+00</td>
<td>2.961E+00</td>
<td>1.825E+00</td>
<td>2.475E+00</td>
<td>1.906E+00</td>
</tr>
</tbody>
</table>

only $O(Nr)$, where $N$ is the number of runs (typically less than 20). Mixed precision arithmetic can be used at will. Comparisons between algorithms can be made readily. However, unlike for TOMS 594, in order to fully investigate any algorithm, an accompanying theory needs to be developed. A final point is that as it is based on statistical hypotheses it cannot provide exact information. The condition comparisons of Section 5.3 however give evidence that at least in the case of linear systems, we can obtain very reliable information (cf. [7]).

The error analyses described above are certainly not the only ones available. For example, since ACM TOMS algorithm 532 [26] uses exactly the same input as ACM TOMS algorithm 594, it is also incorporated in the numerical quality package, although we have not used it in our experiments. We also cite the methods of [31] and [25]. We believe that all these algorithms, their application made easier with some of the provided software, will play a useful role in the study of numerical quality of parallel algorithms. We cite for example [3], where LGRAPH
and TOMS 594 were used to compare standard and block schemes from
matrix multiplication, and the work in [5, 11, 28].

6 Conclusions and Comments on Future Work

Our decision to structure the software into preprocessor and library mod-
ules was very beneficial. It increased the number of potential applica-
tions, making the generation of new ones only a matter of coding new
library modules. Moreover, by having each tool share the preprocessing
stage, the decision to modularize resulted in significant software reuse
and code economization.

We note that some Fortran compilers have an option to automatic-
ically generate calls to user-defined subroutines at the assembler level,
whenever a floating point computation is encountered. The user can then
build a library module, to perform some of the tasks we specified above.
Our tools produce similar instrumentation, not previously available in
the Fortran environment of Section 2. Although this instrumentation is
not at the assembler level (resulting in some imprecisions) it allows extra
information to be carried (using associated variables) which is necessary
to accomplish tasks such as the generation of LGRAPH.

Figure 5 depicts the components of the instrumentation environment.
The shaded areas represent current or future work. These we discuss
next.

Performance. Work has already started in generating a library for an-
alyzing the utilization of the data caches of the Cedar clusters. The
library routines will separate hit/miss ratios statistics for different
cache management schemes.

Numerical quality Interval arithmetic can be implemented by asso-
ciating with each floating point data object a pair of “interval
bounding” associated variables passed as arguments to the subrou-
tine library as described in Section 5.1.1. To implement extended
precision arithmetic, an arbitrary number of associated variables
can be used to hold extra precision for the mantissa, or to index an
area of memory managed as a heap, where the actual floating-point
number are stored. A similar approach can be taken to implement
infinite precision arithmetic. We finally note that the implementa-
tion of the recent work reported in [9] can make use of our tools.
Automatic library generation: We note that the creation of any additional library module involves tedious coding. Moreover, there exists much repetitiveness to the task. It would not be difficult to automate this creation process, using as inputs the user's specifications of the library module.

Templates consisting of the Fortran code used in the body of the subroutine would be duplicated for all valid combinations of each argument’s rank and data type, for each operation and intrinsic. Macros are replaced with the appropriate U_token or V_token to use in the naming conventions of identifiers. The algorithm for generating the library follows:
For all operations and intrinsics:
  For all valid combinations of operand data types:
  For all valid combinations of operand shapes:
    Generate function headers, declare arguments.
    Include appropriate template (more than one would be needed for a complete library) in every generated subroutine with macros replaced by U_tokens and V_tokens.

Our conclusion is that the current and planned tools can provide the user valuable information for his "algorithm selection" decisions, especially as new programming environments (e.g. [14]) allow more accurate instrumentation (cf. Section 3).

Acknowledgments

We would like to thank Dave Jablonowski for his work in integrating our tools in the computational environment described in [15] and Constantine Polychronopoulos for his comments on the extended abstract version of the paper. We also thank Nick Higham for providing us with an early version of [18].

This research was supported by the National Science Foundation under Grants NSF MIP-8410110, NSF CCR-8717942 and the Department of Energy under Grant DOE DE-FG02-85ER25001. Additional support provided by the U.S. Air Force Office for Scientific Research under Grants AFOSR-85-0211, and by AT&T under Grant AT&T AFFL67Sameh.

References


