Parallel Performance of a Combustion Chemistry Simulation

Gregg Skinner
Rudolf Eigenmann *

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

September 1, 1994

Abstract

We used a description of a combustion simulation's mathematical and computational methods to develop a version for parallel execution. The result was a reasonable performance improvement on small numbers of processors. We applied several important programming techniques, which we describe, in optimizing the application. This work has implications for programming languages, compiler design, and software engineering.

1 Introduction

Numerical simulations of reactive flow are widely used for problems such as controlling combustion-generated pollutants, reducing knocking in internal combustion engines, studying the environmental impact of compounds emitted from combustion, and disposing of toxic wastes [1]. These simulations require extensive computation. Many can only be served by the advanced capabilities of a parallel supercomputer. In this paper we describe an effort to optimize the parallel performance of a reactive flow simulation written for serial execution. Specifically, we examine PREMIX [2], which simulates combustion, an important subclass of reactive flow.

Reactive flow modeling problems are governed by equations conserving mass, energy, and momentum. They are coupled with a hydrodynamic system driven by the energy released or absorbed from the chemical reactions. Researchers seek to understand the chemical kinetics behavior of large chemical reaction systems and the associated convective and diffusive transport of mass, momentum, and energy.

Complicating the numerical simulation of reactive flow is numerical stiffness. Stiff equations have one or more rapidly decaying solutions and usually require special treatment. In the context of chemical kinetics Curtiss and Hirschfelder [3] first identified the problem of stiffness in ordinary differential equations in 1952. In reactive flow, stiffness often arises as a result of the differing time scales of the chemical kinetics and the hydrodynamics [4]. Chemical reactions occur on the order of picoseconds, while the convective flow occurs on the order of seconds. Stiffness also results where large temperature gradients occur. To overcome these numerical difficulties researchers often employ time-implicit algorithms and adaptive gridding schemes.

A group at Sandia National Laboratories has developed a number of software tools that facilitate simulation of reactive flow. Three basic packages lie at the heart of their effort. The CHEMKIN library [5] is used to analyze gas-phase chemical kinetics. The TRANSPORT [6] library is used for evaluating gas-phase multicomponent transport properties. SURFKIN [7] is a package for analyzing heterogeneous chemical kinetics at a solid-surface – gas-phase interface. These three combustion libraries undergo continual revision as part of an ongoing effort to provide the numerical combustion community with standardized software. This approach is successful because the governing equations for each reactive flow application must share a number of features. A general discussion of this structured approach to simulating reactive flow is found in [1].

*This work is supported by the National Security Agency and by Army contract #DABT63-92-C-0033. This work is not necessarily representative of the positions or policies of the Army or the Government.

Several codes have been built by Sandia to exploit CHEMKIN, TRANSPORT, and SURFKIN. One of them is PREMIX, which is used to predict the steady state temperature and species concentrations in one-dimensional burner-stabilized and freely propagating premixed laminar flames. This combustion is chemically interesting because the large energy release associated with burning gives rise to high temperatures and many exotic chemical species. The high temperatures resulting from the transfer of chemical energy to heat lead to rapid expansion of the gases which in turn affect convective flow.

The goal of this paper is to describe experiences in an effort to improve the performance of the PREMIX application. The machine architectures we considered are shared memory multiprocessors with a modest number of CPU's, such as the Alliant FX series, the Convex C2 series, and the high ends of the Sun SPARCstation, HP Apollo, IBM RS/6000, and Silicon Graphics Iris series. Such machines are becoming less expensive and more widely available.

Only one version of the `FORTRAN 77` source for PREMIX is distributed by Sandia. This code executes without significant modification on all machines from a personal computer to a Cray. To insure the software can still be used by the large established user base, modifications to the code are strictly backward compatible; that is, the subroutine interfaces are fixed. Our main concern, then, was with extracting parallelism from the chemical and thermodynamic computations performed by the CHEMKIN and TRANSPORT libraries.

We approached PREMIX with a simple goal: Reduce the actual time a program requires to produce a solution to a given problem through efficient use of multiprocessing hardware. To accomplish this, independence must be present in the code so that different subproblems can be executed by separate processors concurrently. Often the desired independence, if it exists, is apparent from the mathematical description of the physical problem. This conceptual independence may not, however, be expressed in the actual code. Two factors contribute to the absence of conceptual independence in the final program: (1) the computational method chosen to approximate the mathematical problem may sequentialize formerly independent tasks; (2) the specific implementation of the computational method adds unnecessary synchronizations.

We therefore make a reasonably sharp distinction between the mathematical model of a problem, the computational method for its solution and the particular implementation of the method. We begin in the next section with a brief overview of PREMIX. In Section 3, we observe how well the original version of PREMIX expresses parallelism inherent to the mathematical model and computational method. In Section 4 we describe the program transformation techniques applied to produce an optimized version of PREMIX. In Section 5 we exhibit the resulting performance improvement, and in Section 6 we offer the conclusions drawn from this work.

2 The PREMIX application

PREMIX is a typical example of a library-oriented production `FORTRAN` code. It is a flexible program developed to analyze general problems involving combustion of premixed gases in a flame. PREMIX consists of a driver and four libraries: CHEMKIN [5], used to analyze gas-phase chemical kinetics; TRANSPORT [6], used to evaluate gas-phase multicomponent transport properties; TWOPNT [8], a two–point boundary value problem solver; and LINPACK [9], a popular numerical linear algebra package. Each is a standardized, extensible library intended for use on a wide variety of platforms. The code, approximately thirty thousand lines of standard `FORTRAN 77`, is highly modular, robust, and portable. The program can be stopped at any of several checkpoints and restarted with a new configuration.

Our testing environment was a shared-memory MIMD machine, an Alliant FX/80 [10] with eight processing units. The processors are register-based with chained functional units and memory port. The computational processors are connected by a concurrency bus, which keeps the overhead for concurrency small. A sequential profile for an execution of the nitrogen combustion simulation mentioned earlier appears in Figures 1 and 2. For the test problem the program tracks 34 chemical species and 151 chemical reactions through three simulated burns. The one-dimensional grid begins with 19 grid points and is ultimately refined to 61 grid points.

The program spends most of its execution time in routines from the CHEMKIN and TRANSPORT libraries. Approximately 65% of the sequential execution time is consumed performing chemical kinetics computations in CHEMKIN routines `ckytx`, `ckmmwy`, `ckwyp`, `ckrat`, `ckhml`, `ckcpbs`, `ckrhoy`, and `ckcpms`. (These subprogram names are defined in Table 1.) Another 20% of the execution time is consumed by transport computations in TRANSPORT routines `mtrnpr`, `ckytx`, `mcadif`, `mcedif`, `mceval`, and `mcacon`. Solving systems of linear

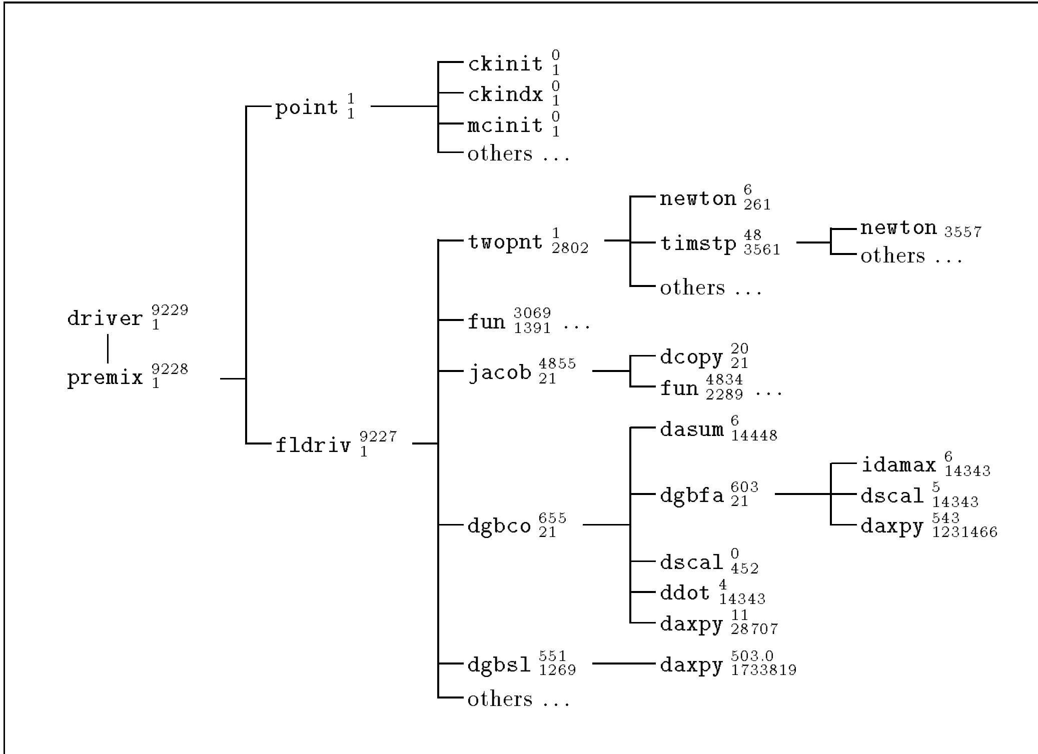

Figure 1: Execution Profile of Sequential Program. Times were obtained on an Alliant FX/80 with serial optimizations (compile command: `fortran -Og -pg`). Elapsed times (in seconds) are superscripted and the number of events is subscripted. Procedure times include time spent in called subprocedures. Total elapsed time is 9305 seconds. Times for the two invocations of "`fun`" are combined in Figure 2.

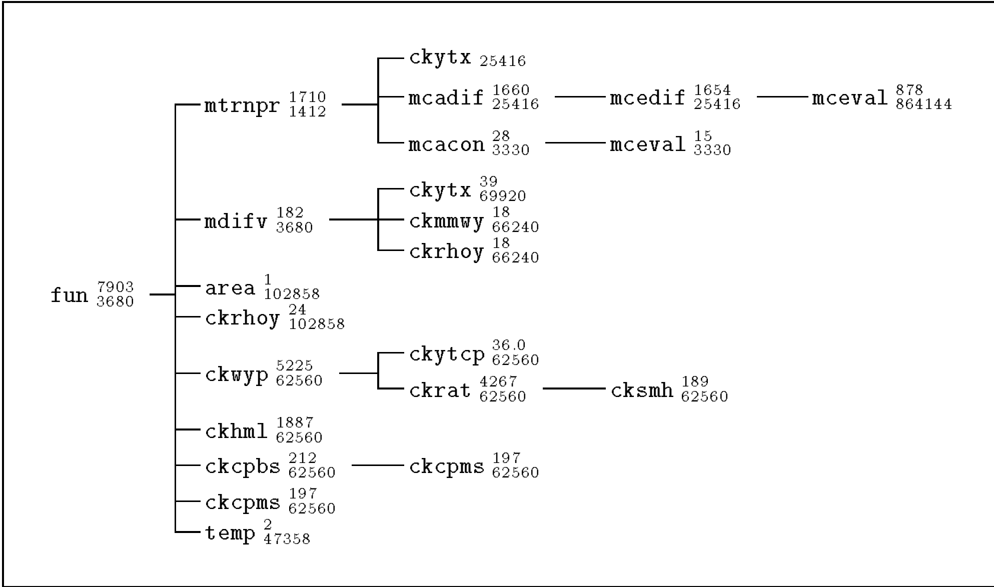

Figure 2: Execution Profile for Procedure `fun`.

equations consumes most of the remaining time. The TWOPNT library simply controls the flow of the computations and thus contributes little to the execution time.

3 Description of the algorithm

We first give a description of the mathematical model and the computational method, which assisted us in discovering which level of outer loop parallelism is best to obtain a granularity sufficient to saturate available processors with reasonably sized parcels of independent work [11]. A mathematical description of the general problem appears in several references (e.g., [2]). We review them briefly here. We then consider the computational methods employed to solve the combustion problem and explore the potential for parallelism in these methods. Finally, we describe the particular implementation of these methods and explore the remaining potential for parallelism in the actual program.

3.1 Mathematical model

PREMIX computes the steady state temperature and species concentrations in one-dimensional burner-stabilized and freely propagating premixed laminar flames. The steady state is defined by the following conservation equations [2]:

$$\dot{M} = \rho u A = \text{constant} \qquad \text{(mass)}, \tag{1}$$

$$\dot{M}\frac{dT}{dx} - \frac{1}{c_p}\frac{d}{dx}\left(\lambda A \frac{dT}{dx}\right) + \frac{1}{c_p}\sum_{k=1}^{K}(\rho A Z_k) c_{p_k}\frac{dT}{dx} + \frac{A}{c_p}\sum_{k=1}^{K}\dot{\omega}_k h_k W_k = 0 \qquad \text{(energy)}, \tag{2}$$

$$\dot{M}\frac{dY_k}{dx} + \frac{d}{dx}(\rho A Z_k) - A\dot{\omega}_k W_k = 0 \qquad (k = 1, \ldots, K) \qquad \text{(momentum)}, \tag{3}$$

where K is the number of chemical species. Thus, $K + 2$ conservation equations govern the steady state of the system. The symbols appearing in these equations are defined in Table 1.

The chemical kinetics computations occur in evaluating the molar rates of species production $\dot{\omega}_k$, the specific form of which is determined by the input dataset according to the equation,

$$\dot{\omega}_k = \sum_{i=1}^{K} \nu_{k,i} q_i \tag{4}$$

where the $\nu_{k,i}$ are user-specified integer stoichiometric coefficients and the q_i are the computed reaction rates. Determining the value of q_i is computationally intensive, consisting of numerous exponentials, logarithms, and reductions, both multiplicative and additive.

The heat generated or absorbed by these reactions strongly affects the gas flow. In PREMIX, the chemical kinetics are computed first from the input data; then the hydrodynamic system governed by the conservation equations (1) – (3) is solved in the presence of the chemical reactions.

Equations (2) and (3) are discretized using finite difference approximations. A grid is numbered from 1 at the cold (input) boundary to J at the hot (output) boundary. The convective terms, $\left(\dot{M}\frac{dT}{dx}\right)$ from the energy equation and $\left(\dot{M}\frac{dY_k}{dx}\right)$ from the momentum equation, are modeled by either first order windward or central differences as necessary. The other derivatives are approximated by first and second order central differences. The diffusive term of the species conservation equation, $\frac{d}{dx}(\rho A Z_k)$, is approximated in the same manner. Appropriate boundary conditions are implemented for both the cold and hot boundaries, yielding a two–point boundary value problem. (See equations (10)–(21) in [2] and discussion therein for a detailed description.) The nitrogen combustion problem is solved first using windward differences for the convective terms. Then the initial solution is used as a starting condition for a run using central differences for the convective terms.

The finite difference approximations reduce the stiff two–point boundary value problem to a system of nonlinear algebraic equations. The boundary value problem is modeled first on a coarse mesh. When necessary, new grid points are added (nonuniformly) in regions where the solution or its gradients change

Symbol	Quantity	Where Computed
x	spatial coordinate along flow direction	–
T	temperature	subroutine `fun`
$\dot{M}$	mass flow rate (independent of x)	subroutine `fun`
Y_k	mass fraction of the kth species	subroutine `fun`
λ	thermal conductivity of the mixture	subroutines `mcmcdt`, `mcacon`, `mceval`
$Z_k = Y_k V_k$	mass fraction times diffusion velocity of the kth species	subroutines `mdifv`, `mcatdr`, `mtrnpr`, `mcadif`, `mcedif`, `mceval`, `ckytx`
$\rho = \frac{p\overline{W}}{RT}$	mass density	subroutine `ckrhoy`
h_k	specific enthalpy of the kth species	subroutine `ckhml`
c_{p_k}	constant pressure heat capacity of the kth species	subroutine `ckcpms`
c_p	constant pressure heat capacity of the mixture	subroutines `ckcpbs`, `ckcpms`
$\dot{\omega}_k$	molar rate of production of the kth species per unit volume	subroutines `ckwyp`, `ckrat`
$\overline{W}$	mean molecular weight of the mixture	subroutine `ckmmwy`
V_k	diffusion velocity of the kth species	read from input
W_k	molecular weight of the kth species	read from input
A	cross-sectional area of the stream tube encompassing the flame	subroutine `area`
p	pressure (constant)	read from input
u	velocity of the fluid mixture (constant)	read from input
R	universal gas constant	read from input

Table 1: Symbols Appearing in the Premixed Flame Equations (1) – (4). More detail is available from the CHEMKIN and TRANSPORT documentation [5, 6].

rapidly. Assuming a unique solution exists, this process ends when the solution has been resolved to a specified degree.

The nonlinear system is solved using the modified Newton-Raphson algorithm. We seek a vector ϕ which satisfies

$$\boldsymbol{F}(\psi) = 0. \tag{5}$$

We begin with a (usually poor) approximation $\hat{\phi}$ to ϕ. It is clear that $\boldsymbol{F}(\hat{\phi})$ is not zero. The quantity

$$\boldsymbol{y} = \boldsymbol{F}(\hat{\phi}) \tag{6}$$

is called the *residual*.

In order to obtain a block-tridiagonal structure in the Jacobian, the mass flow rate, $\dot{M}$, is treated as an independent variable $\dot{M}_j$ at each grid point, and the additional equation stating that they are all equal,

$$\frac{d\dot{M}_j}{dx_j} = 0 \qquad (j = 1, \ldots, J) \tag{7}$$

is added with a suitable boundary condition. This mass conservation equation, coupled with the energy conservation equation (2) and the K equations of momentum conservation (3) yield a total of $K+2$ equations. The approximate solution vector $\hat{\phi}$ has the form,

$$\hat{\phi} = (\hat{\phi}_1, \hat{\phi}_2, \ldots, \hat{\phi}_J) \tag{8}$$

where

$$\hat{\phi}_j = (T_j, Y_{j,1}, Y_{j,2}, \ldots, Y_{j,K}, \dot{M}_j). \tag{9}$$

Equation 9 corresponds to the independent variables for temperature, species concentration, and mass flow rate for each grid point, j.

The modified Newton-Raphson algorithm produces a sequence $\{\phi^{(n)}\}$,

$$\phi^{(n+1)} = \phi^{(n)} - \mu^{(n)}(\boldsymbol{J}^{(n)})^{-1}\boldsymbol{F}(\phi^{(n)}). \tag{10}$$

In the equation, μ is a damping parameter and $\boldsymbol{J}$ is a finite difference approximation to the Jacobian matrix. The sequence converges to the solution of the nonlinear equations $\boldsymbol{F}(\phi)$ given a sufficiently good starting estimate $\phi^{(0)}$. It is rejected if it does not converge.

Should the Newton algorithm fail to converge, a user-specified number of artificial time integrations are performed to improve the conditioning of the nonlinear system. The discretized time integration is again a system of nonlinear equations. The modified Newton-Raphson method is employed to solve the nonlinear system, but in this case it is much more likely to converge. See the discussion in [2] for more details.

Independence inherent to the computational method

Each Newton or time-stepping iteration depends directly on the result of the previous iteration, so we will not discover independence necessary for parallelization outside the computations within a single iteration. We will show, however, that Jacobian evaluation contains considerable independence, in that all residual differences can be computed simultaneously. Additionally, many of the properties evaluated for each species and reaction within a single residual evaluation are independent in principle. Others are not independent, but many have the form of a *reduction*, a computation amenable to partial parallel optimization.

Let $\phi^{(n)}$ represent the vector of independent variables after Newton iteration n. It has been shown in [12] that $\boldsymbol{y} = \boldsymbol{F}(\phi^{(n)})$ depends only on the partial vectors,

$$\phi_{j-1}^{(n)}, \phi_{j}^{(n)}, \phi_{j+1}^{(n)}, \phi_{j-1}^{(n-n_0)}, \phi_{j}^{(n-n_0)}, \phi_{j+1}^{(n-n_0)}. \tag{11}$$

(The dependence on some previous evaluation $n - n_0$ arises from the fact that the transport coefficients are not recomputed for each iteration.) It follows that $\boldsymbol{y}$ depends only on solution vectors $\phi^{(n)}$ and $\phi^{(n-n_0)}$, both of which are available at the beginning of Newton iteration $n+1$. That is, $\boldsymbol{y} = \boldsymbol{F}(\phi^{(n)})$ is a completely explicit computation. Thus, the computations for each grid point sectioning of $\boldsymbol{y}$ can be performed simultaneously. It follows that all the residuals needed to approximate the Jacobian can be computed concurrently.

We see that there exists the potential for several levels of significant parallelism in PREMIX. Note, however, the hierarchy is not strict. For efficiency, the Jacobians are often reused. Thus, a significant number of residual evaluations occur which are not part of Jacobian evaluation. In the nitrogen combustion simulation we used for testing, one third of the residual evaluations occur independent of Jacobian evaluation. This suggests that if a single level of parallelism is to be exploited, it should be done at the level of residual evaluation.

3.2 Specific implementation

The control flow of PREMIX can be viewed as in Figure 3. The CHEMKIN INTERPRETER [5] and TRANSPORT PROPERTY FITTING CODE [6] are each external modules which access databases to create "linking" files to be read during execution. The CHEMKIN and TRANSPORT libraries require access to many problem-specific constants, such as the molecular weights of the species. In addition, each library requires some scratch space, or memory locations used to store values needed only temporarily. Tracking the use of these scratch arrays is significant when analyzing for parallelism.

Because the libraries are general-purpose and used in a wide variety of applications, these work arrays must be of arbitrary size. Thus, a "dynamic" memory allocation scheme is used. Both CHEMKIN and TRANSPORT implement dynamic memory allocation in a way common to scientific programs written in `FORTRAN`. For each data type employed by one of the program libraries (character, integer, double precision floating point), a single, large array is carved into sections by a sequence of integer offsets computed at runtime. The indices are computed during initialization and stored in `COMMON` blocks for future use. They are never modified after initialization. The work arrays for each of the libraries are passed as arguments down the calling tree. A `COMMON` block for each of the libraries encapsulates the pointers into their respective integer and floating point work arrays. It is important to note that the `COMMON` blocks for a particular library are declared only in procedures within that library.

Returning to Figure 3, we see that each time the outer control loop iterates, either the Newton solver or time stepping is invoked. The Newton solver is always invoked first; time stepping is only performed when the Newton solution phase fails to converge. A single Newton iteration consists of the following steps [2]:

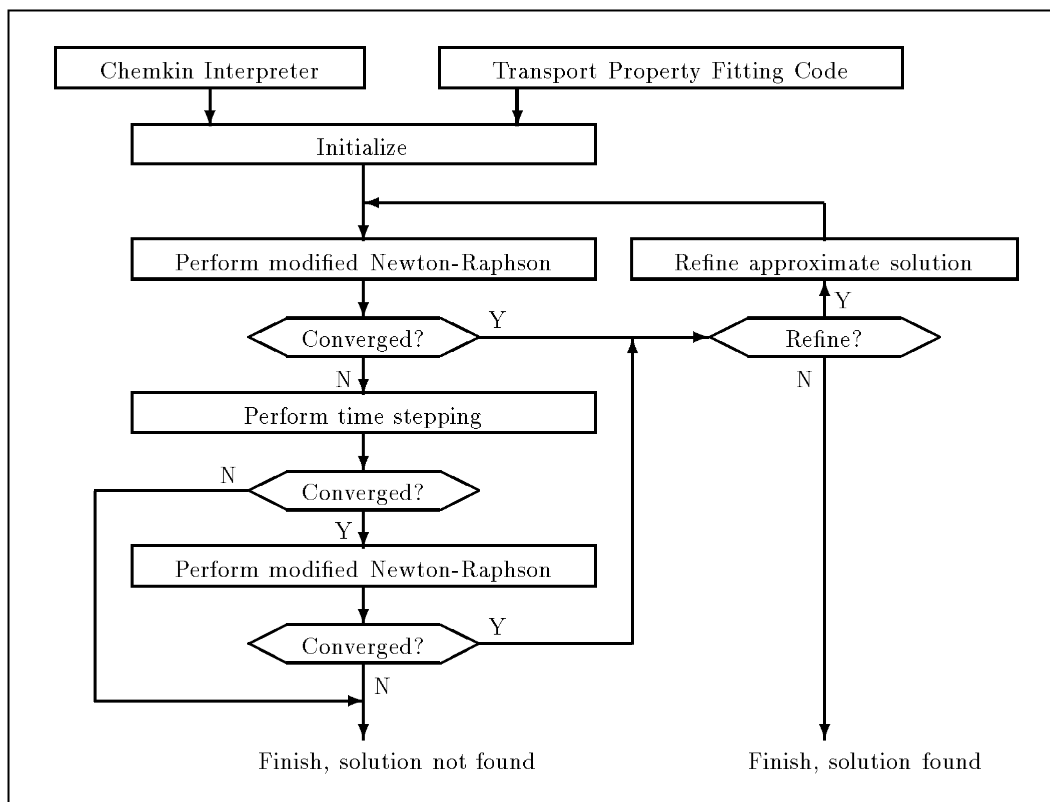

Figure 3: Flow Diagram for Premix. The nonlinear discretized system is solved using the modified Newton-Raphson algorithm. Should the Newton algorithm fail to converge, a user-specified number of artificial time integrations are performed to improve the conditioning of the nonlinear system. The time stepping algorithm also uses the Newton method.

- calculate the residual (`fun`),
- if necessary, evaluate (`jacob`) and factor (`dgbco`) the Jacobian matrix, and
- backsolve (`dgbsl`).

Because chemical computations involve only a grid block and its immediate neighbors (equation (11)), the chemistry is local. As the residual evaluations are independent of one another, no conceptual reason exists that the residuals cannot be computed efficiently in parallel.

Computing the residual requires numerous chemical and thermodynamic property evaluations at each grid point. The computation has three distinct steps. First, the transport coefficients are evaluated, if necessary. Then the diffusion velocities are computed. Finally, the chemical kinetics terms are evaluated and the residuals of the governing equations (2), (3), and (7) are determined.

However, the specific implementation of the computational methods hides some of the potential for parallelism. Concurrent evaluation of the residuals is hampered by the presence of shared local variables and work arrays. The chemical and thermodynamic computations for each grid point, which we also identified as independent in principle, cannot be executed concurrently either. In addition to shared local variables and work arrays, the nearest-neighbor communication of density and area data forces a sequentializing synchronization. The next section describes techniques to overcome some of these problems.

4 Programming techniques and optimization

In this section we describe the program transformation techniques we applied to the specific implementation of Premix and the program analysis that was necessary to do this. We compare these techniques

```
real temp(kk), c(jj)

do j = 1, jj
   do k = 1, kk
      temp(k) = k * b(k)
   end do

   do k = 1, kk
      c(j) = c(j) + temp(k)
   end do
end do
```

```
real temp(kk,jj), c(jj)

doall j = 1, jj
   do k = 1, kk
      temp(k,j) = k * b(k)
   end do

   do k = 1, kk
      c(j) = c(j) + temp(k,j)
   end do
end do
```

Figure 4: Privatization of Arrays. In the second code fragment, each iteration of the outer loop is provided a separate copy of work array "`temp`".

to those applied in other application programs and discuss some implications on programming languages, compiler design, and software engineering issues.

4.1 Transformation and analysis techniques

The basic program modification that enabled multiple processors to participate in the parallel execution of the program was to declare a number of time-consuming loops to be executable concurrently. Simply speaking, in order to do this we first had to recognize that the iterations of these loops were potentially independent, then perform some transformations to make them truly independent, and finally insert a directive informing the compiler that the loops shall be executed in parallel.

By far the most important transformation in this process was the *privatization of arrays* (Figure 4) that are used as temporary work spaces within loop iterations. In the original program all such loop iterations use the same array(s) for storing temporary results. In a parallel execution of the unmodified program, every iteration would have to wait before using this array until the previous iteration is done using it, which effectively would serialize the loop. However, by giving each iteration a separate copy of the array, we can avoid these dependences. The difficulty of this transformation is in making sure that it is a truly temporary array where no array element passes information from one loop iteration to the next. This is usually done by an array *definition/use analysis* of the program.

An additional technique – the parallelization of reduction operations – we have found to be applicable in our program. However, we have not done this because we exploited an outer level of parallelism. The transformation will become important on machine architectures that support the exploitation of multiple levels of parallelism, for example machines that have cluster structure so that the outer parallel loops can be spread across clusters while the inner loops exploit the parallel resources within the cluster.

For both the definition/use analysis and the detection of independence of the loops we had to analyze the program interprocedurally. Often, array sections were defined (i.e., written) in one subroutine and used (i.e., read) in another subroutine. Even more difficult was the analysis of accessed array sections that involved program input data. Sometimes it was only knowledge of the application that could ensure that, in all reasonable executions of the program, input variables would relate so that the defined array ranges would always cover the used ranges or that the ranges accessed in different loop iterations would never overlap.

The dynamic memory allocation scheme, mentioned in Section 3.2, further complicated the situation. We had to track array subscripts which were themselves subscripted array elements in order to determine which sections of the original, large array are read or written. Since the subscript arrays are read-only after their initialization, it is possible to determine temporary arrays and parallel loops from the analysis of the program code. However, this process is tedious and it makes the interesting question of whether such techniques could be automated in a compiler quite challenging.

4.2 Tools, languages, and programming methodology

A profile facility that identified the most time-consuming loops in the program was the basic instrument for our program analysis. In addition, the most helpful tool was an array section analysis facility that determined the array sections read and written in each subroutine and loop. This information was then propagated up the calling tree so that the summary of all accessed arrays could be seen at each loop.

The actual transformations were done in a conventional text editor. Compared to the time consumed by the program analysis this task was not overly expensive, although the mechanics of array privatization could be somewhat tedious as described below.

We restructured our program by explicitly specifying parallel activities, rather than changing the program so that the compiler could recognize the parallelism automatically. The language we used is `FORTRAN 77` plus directives. The only directive we used is `CNCALL`, which specifies that the loop shall be executed in parallel. Private arrays were specified in two forms, both using available `FORTRAN 77` constructs. One form is to declare the array local to a subroutine that is called inside the parallel loop and the other is to expand the array by one dimension and index this dimension with the loop variable. The second form is usually called *array expansion*. Sometimes, subroutine parameter lists had to be modified in order to pass expanded arrays from calling to the called routine.

Common extensions to `FORTRAN 77` are constructs for dynamic array declaration. Arrays of arbitrary size and dimension can be declared locally, within a subprogram. Had we used this extension, we would not have had to modify any of the subprogram parameter lists, leaving the Chemkin and Transport libraries backward compatible.

The availability of a directive that declares variables private to a loop would have been very useful for our purposes because it would have allowed us to leave the existing program text unchanged. Such a directive would also have to support the privatization of a partial array. We encountered situations where part of an array was read-only and another part was used for temporary storage. To handle this situation we split the arrays into different parts and privatized the temporarily used sections. The need for a `PRIVATE` directive is an important conclusion of our work, and it corresponds to findings of related work.

The method of program optimization we have applied consists of identifying the time-consuming loops in the program, analyzing array sections that are read and written in these loops, and deriving privatizable and independent array sections. The parallel loops in our program could then be determined from this information. The actual transformations necessary to express the parallelism were straightforward. This programming scheme seems generally applicable and may be used as a programming methodology that can be applied in a systematic way. Although we have found this to be useful for optimizing other programs as well, we should note that there are time-consuming optimization steps for which we don't know generally applicable methods. Such steps are the gathering of knowledge about the application that goes beyond the analysis of the program text. We have found this to be important in some cases for our program optimization.

4.3 Comparison to findings of related projects

In a related project of optimizing application programs for parallel computers similar results were found. Such projects include the Cedar Fortran project [13, 14] which was completed at our center in 1992, and the follow-on Polaris project [15]. Both projects studied transformation techniques that are needed to speed up

real programs. This was done by hand-parallelizing a suite of codes, including the Perfect Benchmarks and some applications of relevance to the users at the NCSA[1].

The most important transformations identified were the same as in our project. Array privatization was most effective, followed by the parallelization of reduction operations. Interprocedural definition/use analysis was a crucial technique to determine the applicability of the transformations. The transformations yielded fully parallel loops whose iterations could be executed independently on multiple processors.

Our application is relevant for these other projects in that it confirms the results and thus shows that they carry over from the sample benchmark suite to new programs. One difference seems worth noting. The ultimate goal of the above related projects was to find techniques that can be automated in a parallelizing compiler, and in fact most of the transformations identified were reported to be automatable. In our program we have found that some crucial information for determining the applicability of the parallelization techniques is known only from the input files and thus is not available at compile time. Although there are compilation techniques that are able to parallelize such situations at runtime [16], our findings indicate that it will be at least difficult to detect the parallelism automatically. A full discussion of this point is beyond the scope of this paper and is the object of future projects.

A related approach to methodologies for parallel programming is described in [17]. Our findings largely agree with this approach. One difference is that [17] envisions a "program-level" optimization, in which all necessary information for transforming the program can be gathered from the program text. As we have mentioned, for optimizing PREMIX there was sometimes a need to use knowledge about the mathematical and physical properties of the problem that could not easily be gathered from the specific implementation of the program.

Our findings can also be compared with the parallel programming methodology that envisions the design of application programs from optimized libraries. The parallelism would be hidden in these libraries and the programming method for the user of these libraries would be no different from sequential programming. A further advantage of this approach is that the libraries could be optimized specifically for each machine and the application program would be portable. Because PREMIX uses standard libraries, it would be a natural candidate for such an approach. However, we have found that exploiting parallelism within the libraries does not lead to significant speedup. The parallelism we exploited is at a higher loop level and the libraries themselves execute on one processor each.

5 Results

We gathered performance data on the Alliant FX/80 for four versions of PREMIX:

- Original Sequential – the original PREMIX code compiled with sequential optimizations (`fortran -Og`);
- Original Parallel – the original code optimized for parallel execution by the FX/FORTRAN automatic compiler (`fortran -Ogc`);
- Optimized Parallel – Original Parallel with explicit parallel constructs added, as described in Section 4; and
- Optimized Sequential – Optimized Parallel compiled for sequential execution (`fortran -Og`).

The profiling option (`-pg`) was disabled for these experiments. We also excluded vectorization optimizations from our performance tests because the vectors were too short to be useful with the FX/80 architecture. Enabling vectorization consistently resulted in greater execution times.

The performance improvement can be seen in Figure 5. The third group of bars shows total execution times for the four versions of PREMIX. We see that the Optimized Parallel version of the code executes approximately 4.4 times faster than Original Sequential. The added overhead of the manual parallelization, seen by comparing the execution time of Optimized Sequential to Original Sequential, is minimal (less than 0.3%). Automatic compiler optimizations, isolated in the Original Parallel version of the code, are responsible for about half the performance improvement. This result can also be seen in Figure 6, which exhibits the inverse execution times of the parallel versions of the code for varying numbers of processors.

We separated the linear algebra and chemistry computations in Figure 5 to demonstrate how the nature of the Optimized Parallel version of the program has changed from the original. While performance of

[1] The National Center for Supercomputer Applications at the University of Illinois

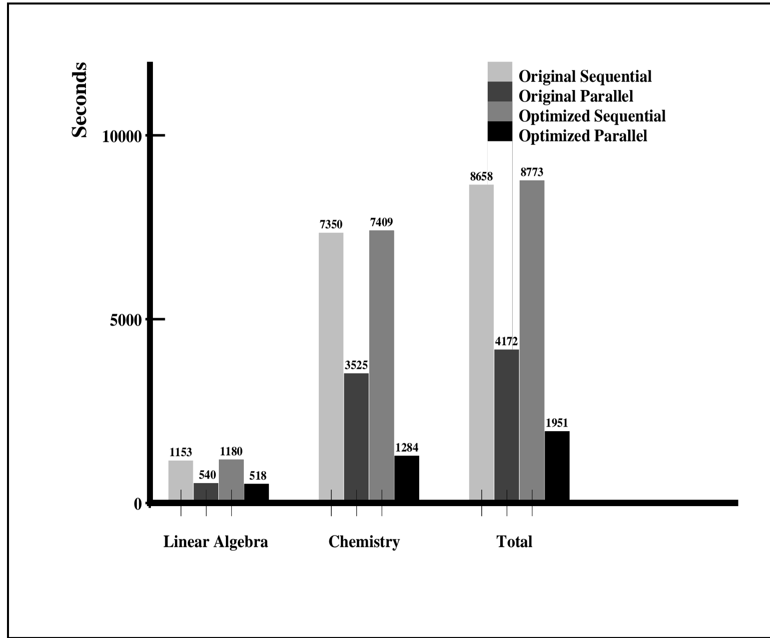

Figure 5: Comparative Performance of Four Versions of PREMIX. Times were obtained using an eight-processor Alliant FX/80 with the FX/FORTRAN parallelizing compiler.

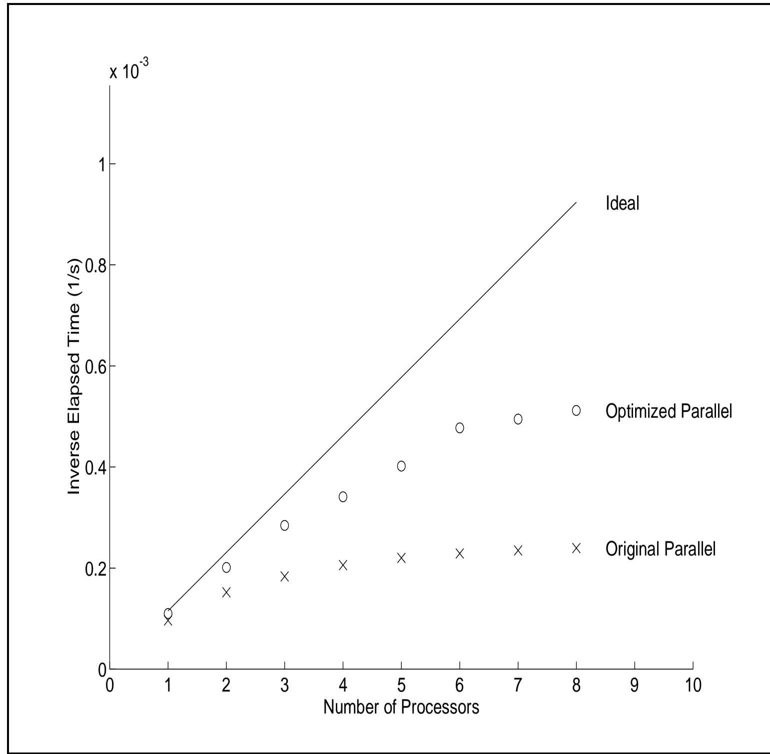

Figure 6: Inverse Execution Times Versus Number of Processors. Times for original and optimized parallel versions of PREMIX were obtained on an Alliant FX/80. An ideal performance improvement line is included for comparison.

	Original Sequential		Original Parallel		Optimized Sequential		Optimized Parallel		Performance Improvement
	Sec. (a)	%	Sec. (b)	%	Sec. (c)	%	Sec. (d)	%	(c)/(d)
Residual Evaluation Loop	5472	63.2	2283	54.7	5430	61.9	920	47.1	5.9
Transport Loop	1651	19.1	1118	26.8	1735	20.0	296	15.1	5.9
Diffusion Loop	170	2.0	82	2.0	167	1.9	28	1.4	6.0
Other Chemistry	57	0.0	42	0.0	77	0.0	40	0.0	1.9
Total Chemistry	7350	84.9	3525	84.5	7409	84.4	1284	65.8	5.8
Linear Algebra	1153	13.3	540	12.9	1180	13.5	518	26.5	2.3
Two–Point BVP Solver	68	0.8	37	0.9	86	1.0	60	3.1	1.4
I/O and OS	1.0	70	1.7	98	1.1	89	4.6	1.1	0.2
Total	8658	100.0	4172	100.0	8773	100.0	1951	100.0	4.5

Table 2: Elapsed Execution Times of Four Versions of PREMIX. The parallel versions were executed using eight processors.

the chemical computations improved significantly, by a factor of almost six, the gain for the linear algebra routines, which we were only able to parallelize partially, was a more modest 2.3. Linear algebra commanded only about 13% of the Original Sequential execution time. In the Optimized Parallel version linear algebra is responsible for about 27%. As the number of processors grows, linear algebra computations will increasingly dominate execution time.

Formerly the chemistry was so expensive that the time spent in linear algebra could be easily ignored. Now that the chemistry can be made relatively cheap, the algorithmic trade-offs have changed. Alternatives to the overall solution strategy should be reviewed. Some discussion of parallel methods for solving two–point boundary value problems can be found in [18]. The LAPACK effort [19] offers parallel versions of banded system solvers, exploiting parallelism in multiple right-hand sides and blocking algorithms. We did not, however, obtain any performance improvement when we replaced the LINPACK linear algebra routines with their LAPACK counterparts. PREMIX has no multiple right-hand sides to exploit, but blocking should have yielded some improvement. The reason it did not do so is still under investigation.

Table 2 shows the execution times of the four versions of PREMIX. The three loops we manually parallelized constitute nearly all the significant chemical computations in the code. As these loops are explicitly parallel in the Optimized Parallel version of the code, we have successfully modified the implementation to express the parallelism inherent to the computational method. However, the execution times of these loops exhibit only a six-fold improvement on eight processors. We believe this is largely due to an imbalance in the work load. The program spends much of its time working with a 19-point grid. If we assume each iteration of the loop over the grid points executes in the same amount of time, 19 iterations are completed in the time that the eight processors could execute 24. This roughly 79% efficiency would reduce the performance improvement factor to 6.3. A further source of inefficiency is the limited memory bandwidth of the FX/80 machine, which provides only a four-way path between the eight processors and the shared memory, and penalizes applications with poor cache hit ratios. Factoring in these inefficiences models the measured performance with good accuracy, so that we can characterize the scalability of our application as follows.

The PREMIX application runs with reasonable efficiency on machines with small numbers of processors. It is potentially scalable to larger numbers of processors for the solution of larger problems with significantly more than 65 grid points. As the chemistry component of the application speeds up, the linear algebra part becomes speed limiting. We have not investigated possible improvements to this component of the application; however, it seems possible to resolve this limitation through appropriate changes in the used algorithms.

6 Conclusion

We performed a detailed analysis of the mathematical model used in PREMIX coupled with a study of the computational methods to gain a picture of a hierarchy of parallelism inherent to the problem being solved. A manual analysis of the code followed, from which we determined to what extent the parallelism inherent to the implementation was expressed in the original version. We then chose an outer loop level appropriate to our target machine and applied a handful of manual parallelizations. In all, we modified less than 100 lines of code. The result was a greater than four-fold improvement in the simulation's execution time on an Alliant FX/80 with eight processors.

In this work we have found that the PREMIX combustion chemistry application runs with reasonable speed on small numbers of processors and potentially scales up to more highly parallel systems. The most important program transformation to achieve our performance improvement was the privatization of arrays. To determine the applicability of this transformation we had to do a careful, interprocedural analysis of defined and used array sections. The available language constructs were not always adequate for expressing dynamically-sized loop-private arrays, and we suggest that such constructs be included in future language designs. The method used for optimizing our program seems generally applicable and, with the provision of supporting tools, we believe they represent a step toward the understanding and improvement of the process of optimizing large application codes for high-performance computers.

References

[1] R. Kee and J. Miller. A structured approach to the computational modeling of chemical kinetics and molecular transport in flowing systems. Technical Report SAND86–8841, Sandia National Laboratories, 1986.

[2] R. Kee, J. Grcar, M. Smooke, and J. Miller. A FORTRAN program for modeling steady laminar one–dimensional premixed flames. Technical Report SAND85–8240, Sandia National Laboratories, 1985.

[3] C. Curtiss and J. Hirschfelder. Integration of stiff equations. *Proceedings of the National Academy of Sciences of the United States of America*, 38:235–243, 1952.

[4] E. Oran and J. Boris. *Numerical Simulation of Reactive Flow*. Elsevier, 1987.

[5] R. Kee, F. Rupley, and J. Miller. CHEMKIN–II: A FORTRAN chemical kinetics package for the analysis of gas–phase chemical kinetics. Technical Report SAND89–8009, Sandia National Laboratories, 1989.

[6] R. Kee, G. Dixon-Lewis, J. Warnatz, M. Coltrin, and J. Miller. A FORTRAN computer code package for the evaluation of gas–phase, multicomponent transport properties. Technical Report SAND86–8426, Sandia National Laboratories, 1986.

[7] M. Coltrin, R. Kee, and F. Rupley. Surface CHEMKIN: A FORTRAN package for analyzing heterogeneous chemical kinetics at a solid-surface–gas-phase interface. Technical Report SAND90-8003, Sandia National Laboratories, 1990.

[8] J. Grcar. The Twopnt program for boundary value problems. Technical Report SAND91-8230, Sandia National Laboratories, April 1992.

[9] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK Users' Guide*. Society of Industrial and Applied Mathematics, Philadelphia, 1979.

[10] Alliant Computer Systems Corporation, Acton, MA. *FX/FORTRAN Programmer's Handbook*, 1985.

[11] J. Tyler, A. Bourgoyne, D. Logan, J. Baron, T. Li, and D. Schneider. A vector-parallel version of BOAST II for the IBM 3090. Internal Report, IBM Kingston, 1990.

[12] G. Skinner. Finding and Exploiting Parallelism in a Production Combustion Simulation Program. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1993.

[13] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmarks Programs. In *Lecture Notes in Computer Science 589*, pages 65–83. Springer Verlag, NY, 1992.

[14] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, and G. Jaxon. The Cedar Fortran Project. Technical Report 1262, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., April 1992.

[15] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing. Technical Report 1348, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., April 1994.

[16] L. Rauchwerger and D. Padua. Speculative Run-Time Parallelization of Loops. Technical Report 1339, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., March 1994.

[17] R. Eigenmann. Toward a methodology of optimizing programs for high-performance computers. In *Proceedings of 1993 International Conference on Supercomputing, Tokyo, Japan*, pages 27–36, Tokyo, Japan, July 19-23 1993. ACM Press.

[18] S. Wright. Stable parallel algorithms for two–point boundary value problems. *SIAM Journal on Scientific and Statistical Computing*, 1992.

[19] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.