PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems

Alan C. Hindmarsh
Allan G. Taylor
Lawrence Livermore National Laboratory

Center for Applied Scientific Computing

UCRL-ID-129739
February 1998

PVODE AND KINSOL: PARALLEL SOFTWARE FOR DIFFERENTIAL AND NONLINEAR SYSTEMS*

ALAN C. HINDMARSH AND ALLAN G. TAYLOR†

Abstract. In this project, parallel general-purpose software for two classes of mathematical problems has been developed. PVODE is a portable solver for ordinary differential equation systems, based on robust mathematical algorithms, and targeted at large systems on parallel machines. It is the parallel extension of the earlier sequential solver CVODE. A related solver called KINSOL has been developed for systems of nonlinear algebraic equations. KINSOL was first developed as a sequential solver, on a design that permitted extending it to a parallel version with fairly minimal additions. Both PVODE and KINSOL are being used within a parallel version of the tokamak edge plasma model UEDGE. KINSOL is also being applied in the ParFlow groundwater flow model to solve a nonlinear pressure equation.

1. Introduction. This is a final report on the PVODE LDRD Project (95-ERP-036), which was funded in Fiscal Years 1995 - 1997. In this section, we summarize the goals and motivations for the project.

A large number of application codes, both within and outside LLNL, make use of modern solvers for ordinary differential equation (ODE) systems, nonlinear algebraic equation systems, and differential-algebraic equation (DAE) systems. The need for higher resolution and speed is forcing many of these applications to move to massively parallel processors (MPPs), where they will need parallel versions of such solvers.

ODE solvers written at LLNL are among the most widely used ODE initial value system solvers anywhere. The initial goal of the project was to produce a code system for parallel machines called PVODE (Parallel Variable-coefficient ODE solver) that combines the capabilities of two older (sequential) solvers. In the case of large stiff systems, implicit methods must be used, and the natural choice for solving the linear systems that arise is that of preconditioned iterative methods. For that case, we have also developed preconditioner modules based on sparse approximations to the system Jacobian matrix.

There are two other widely used sequential software packages, written at LLNL, which solve nonlinear algebraic systems and DAE systems, respectively. Both of these make use of the same preconditioned iterative (Krylov) methods as the ODE solvers. Building on both PVODE and our sequential nonlinear system solver NKSOL, we have developed a parallel nonlinear system solver called KINSOL (Krylov Inexact Newton SOLver). The preconditioner module developed for PVODE has been adapted for use with KINSOL. In a similar spirit, we plan to develop an analogous parallel solver for DAE systems.

To support users with Fortran application programs, we have developed a set of Fortran/C interfaces which allow Fortran users to use PVODE. A similar set of interfaces supports Fortran users of KINSOL.

We have worked on applications of this software in two different areas. The first is to

* Research performed under the auspices of the U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract W-7405-ENG-48. Work supported by LDRD, Project 95-ER-036.

† Center for Applied Scientific Computing, L-561, LLNL, Livermore, CA 94551.

2-D tokamak edge plasma models developed in LLNL's Magnetic Fusion Energy Division. Their sequential code UEDGE has relied on three of our solvers for several years, and a parallel version of it is in development. The other is an application of KINSOL to nonlinear pressure equations in groundwater modeling, within LLNL's Center for Applied Scientific Computing.

2. PVODE – a Parallel ODE Solver.

2.1. Mathematical preliminaries. PVODE solves initial-value problems for systems of ODEs. Such problems can be stated as

$$\dot{y} = f(t, y), \qquad y(t_0) = y_0, \qquad y \in \mathbf{R}^N, \tag{1}$$

where $\dot{y} = dy/dt$ and $\mathbf{R}^N$ is the real N-dimensional vector space. That is, (1) represents a system of N ordinary differential equations and their initial conditions at some t_0. The dependent variable is y and the independent variable is t.

The PVODE solver was developed as an extension to parallel machines of an older software package called CVODE [9, 10]. The ODE solver CVODE, which was written by Cohen and Hindmarsh, combines features of two earlier Fortran codes, VODE [1] and VODPK [5], written by Brown, Byrne, and Hindmarsh. Both use variable-coefficient multistep integration methods, and address both stiff and nonstiff systems. (Stiffness is defined as the presence of one or more very small damping time constants.) VODE uses direct linear algebraic techniques to solve the underlying banded or dense linear systems of equations in conjunction with a modified Newton method in the stiff ODE case. On the other hand, VODPK uses a preconditioned Krylov iterative method [3] to solve the underlying linear system. User-supplied preconditioners directly address the dominant source of stiffness. Consequently, CVODE implements *both* the direct and iterative methods. Currently, with regard to the nonlinear and linear system solution, PVODE has three method options available: functional iteration, Newton iteration with a diagonal approximate Jacobian, and Newton iteration with the iterative method SPGMR (Scaled Preconditioned Generalized Minimal Residual method) [3, 16].

PVODE solves the ODE system by one of two numerical methods — the backward differentiation formula (BDF) and the Adams-Moulton formula. Both are implemented in variable-stepsize, variable-order form. The BDF uses a fixed-leading-coefficient form, as opposed to the fully variable-step form (see [1]). The two formulas used can both be represented by a linear multistep formula

$$\sum_{i=0}^{K_1} \alpha_{n,i} y_{n-i} + h_n \sum_{i=0}^{K_2} \beta_{n,i} \dot{y}_{n-i} = 0, \tag{2}$$

where the N-vector y_n is the computed approximation to $y(t_n)$, the exact solution of (1) at t_n. The stepsize is $h_n = t_n - t_{n-1}$. The coefficients $\alpha_{n,i}$ and $\beta_{n,i}$ are uniquely determined by the particular integration formula, the history of the stepsize, and the normalization $\alpha_{n,0} = -1$. The Adams-Moulton formula is recommended for nonstiff ODEs and is represented by (2) with $K_1 = 1$ and $K_2 = q - 1$. The order of this formula is q and its values range from 1

through 12. For stiff ODEs, BDF should be selected and is represented by (2) with $K_1 = q$ and $K_2 = 0$. For BDF, the order q may take on values from 1 through 5. In the case of either formula, the integration begins with $q = 1$, and after that q varies automatically and dynamically.

For either BDF or the Adams formula, $\dot{y}_n$ denotes $f(t_n, y_n)$. That is, (2) is an implicit formula, and the nonlinear equation

$$\begin{aligned} G(y_n) &\equiv y_n - h_n \beta_{n,0} f(t_n, y_n) - a_n = 0 \\ a_n &= \sum_{i>0} (\alpha_{n,i} y_{n-i} + h_n \beta_{n,i} \dot{y}_{n-i}) \end{aligned} \tag{3}$$

must be solved for y_n at each time step. For nonstiff problems, functional (or fixpoint) iteration is normally used and does not require the solution of a linear system of equations. For stiff problems, a Newton iteration is used and for each iteration an underlying linear system must be solved. This linear system of equations has the form

$$M[y_{n(m+1)} - y_{n(m)}] = -G(y_{n(m)}), \tag{4}$$

where $y_{n(m)}$ is the mth approximation to y_n, and M approximates $\partial G/\partial y$:

$$M \approx I - \gamma J, \qquad J = \frac{\partial f}{\partial y}, \qquad \gamma = h_n \beta_{n,0}. \tag{5}$$

At present, aside from the diagonal Jacobian approximation, the only option implemented in PVODE for solving the linear systems (4) is the iterative method SPGMR (Scaled, Preconditioned GMRES) [3], which is a Krylov subspace method. In most cases, performance of SPGMR is improved by user-supplied preconditioners.

SPGMR is one of a class of preconditioned Krylov methods. Write the linear system (4) simply as

$$Ax = b. \tag{6}$$

A preconditioned Krylov method for (6) involves a preconditioner matrix P that approximates A, but for which linear systems $Px = b$ can be solved easily. For preconditioning on the left, the Krylov method is applied to the equivalent system

$$(P^{-1}A)x = P^{-1}b,$$

while for right preconditioning it is applied to

$$(AP^{-1})(Px) = b.$$

In PVODE, the user may precondition the system on the left, on the right, on both the left and right, or use no preconditioner. (Actually, preconditioning on both sides involves a factorization of P as $P_1 P_2$ into factors used on the two sides of A.) In any case, the Krylov method (in our case GMRES) is applied to the transformed system

$$\bar{A}\bar{x} = \bar{b}.$$

From an initial guess $\bar{x}_0$, an approximate solution $\bar{x}_m = \bar{x}_0 + z$ is obtained for $m = 1, 2, \ldots$ (until convergence), with z chosen from the Krylov subspace $K_m = span\{r_0, \bar{A}r_0, \ldots, \bar{A}^{m-1}r_0\}$ of dimension m, where r_0 is the initial residual $\bar{b} - \bar{A}\bar{x}_0$. Each Krylov iteration requires one matrix-vector multiply operation $\bar{A}v$, which is a combination of multiplies by A and by P^{-1}. Multiplication of a given vector v by A requires the product Jv, and that is approximated by a difference quotient $[f(t, y + \sigma v) - f(t, y)]/\sigma$. Multiplication by P^{-1} is to be provided by the user of the solver, and is generally problem-dependent. In the case of GMRES, the choice in K_m is based on minimizing the L_2 norm of the residual $\bar{b} - \bar{A}\bar{x}_m$ [3, 16].

The integrator computes an estimate E_n of the local error at each time step, and strives to satisfy the local error test

$$\|E_n\|_{rms,w} < 1. \tag{7}$$

Here the weighted root-mean-square norm is defined by

$$\|E_n\|_{rms,w} = \left[\sum_{i=1}^{N} \frac{1}{N} (w_i E_{n,i})^2\right]^{1/2}, \tag{8}$$

where $E_{n,i}$ denotes the ith component of E_n, and the ith component of the weight vector is

$$w_i = \frac{1}{rtol|y_i| + atol_i} \quad . \tag{9}$$

This permits an arbitrary combination of relative and absolute error control. The user-specified relative error tolerance is the scalar $rtol$ and the user-specified absolute error tolerance is $atol$, which may be an N-vector (as indicated above) or a scalar. Since these tolerances define the allowed error per step, they should be chosen conservatively.

In most cases of interest to the PVODE user, the technique of integration will involve BDF, the Newton method, and SPGMR.

2.2. Solver design and development. The CVODE package was designed and developed earlier with the parallel extension in mind. More specifically, PVODE was envisioned as an extension of CVODE which would run on parallel machines in the Single Program, Multiple Data (SPMD) multiprocessor programming paradigm. CVODE has a highly modular design, in which the central algorithm for the ODE integration is separated from those for solving the linear systems. Each of the linear solvers used is incorporated in a generic form, suitable for use in its own right, with a small amount of interface coding connecting it with the ODE solver. Because the applications of our parallel solvers are expected to be large in size, and because direct methods do not parallelize as easily, we chose not to retain in PVODE the direct linear system methods that are in CVODE, but focus mainly on the Krylov iterative method SPGMR.

Another important feature of the CVODE design is that, with the direct linear solvers removed, all operations on N-vectors are carried out in a separate module of vector kernels. It is this vector module that has been rewritten in generating the parallel extension, PVODE. The required modifications include a revised definition of the `N_Vector` type, in

that N-vectors in PVODE are distributed across the multiple processors. Our revised implementations of the vector kernels make use of message passing, and to some extent are specific to the particular parallel machine environment. However, by design, we have isolated the machine-dependent coding, and kept to a minimum the passing of machine-dependent information in the user interface to PVODE.

Our first implementation of PVODE was written for the Cray-T3D machine (256 processors) with its shared memory (SHMEM) programming model. Thus the corresponding revised vector kernels use functions from the SHMEM Library to perform the needed reduction operations, and must set up certain SHMEM work arrays accordingly. We refer to the resulting package as SHMEM_PVODE.

Subsequently, we developed a version of PVODE based on the Message Passing Interface (MPI) system, which is becoming a widely accepted standard interface for message passing software. Our vector kernels in this case are considerably simpler than in the SHMEM case, because MPI operates at a somewhat higher linguistic level. Moreover, since MPI has been widely implemented in most parallel machines, this version, called MPI_PVODE, is highly portable, whereas SHMEM_PVODE is suitable for the Cray-T3 series exclusively.

In both implementations of PVODE, we quickly demonstrated proof of the basic design principle, whereby parallel extensions to CVODE can be isolated to the module of vector kernels. Moreover, the re-entrant design allows two or more instances of PVODE to be run in parallel. The portability of MPI_PVODE is demonstrated in that it has been run on an IBM SP2, a Cray-T3D and Cray-T3E, and a cluster of workstations.

The PVODE package can be thought of as being organized in layers. The user's main program resides at the top level. This program makes various initialization calls, and calls the core integrator `CVode`, which carries out the integration steps. Of course, the user's main program also manages input/output. At the next level down, the core integrator `CVode` manages the time integration, and is independent of the linear system method. `CVode` calls the user supplied function `f` and accesses the linear system solver. At the third level, the linear system solver `CVSpgmr` can be found, along with the approximate diagonal solver `CVDiag`. Actually, `CVSpgmr` calls a generic solver for the SPGMR method, consisting of modules `SPGMR` and `ITERATIV`. `CVSpgmr` also accesses the user-supplied preconditioner solve routine, if specified, and possibly also a user-supplied routine that computes and preprocesses the preconditioner by way of the Jacobian matrix or an approximation to it. Other linear system solvers may be added to the package in the future. Such additions will be independent of the core integrator and `CVSpgmr`. Three supporting modules reside at the fourth level: `LLNLTYPS`, `LLNLMATH`, and `NVECTOR`. The first of these defines types `real` and `integer`. The second specifies power functions, and the third is discussed further below.

As explained earlier, a separate module of vector kernels, `NVECTOR`, handles all calculations on N-vectors in a distributed manner. For any vector operation, each processor performs the operation on its contiguous elements of the input vectors, of length (say) `Nlocal`, followed by a global reduction operation where needed. In this way, vector calculations can be performed simultaneously with each processor working on its block of the vector. Vector kernels are designed to be used in a straightforward way for various vector operations that require the use of the entire distributed N-vector. These kernels include dot products,

weighted root-mean-square norms, linear sums, and so on. The key lies in standardizing the interface to the vector kernels without referring directly to the underlying vector structure. This is accomplished through abstract data types that describe the machine environment data block (type `machEnvType`) and all N-vectors (type `N_Vector`). Functions to define a block of machine-dependent information and to free that block of information are also included in the vector module.

The modules in the PVODE package are listed in Table 1 below. Corresponding to each module name are .h and .c file names (.h only for LLNLTYPS). The routines listed as user-callable are those that a PVODE user would call. The SPGMR module also has user-callable routines, if used as a linear system solver by itself.

Module name	User-callable routines	other contents
CVODE	CVodeMalloc, CVode, CVodeFree, CVodeDky	RHS function type RhsFn
CVDIAG	CVDiag	
CVSPGMR	CVSpgmr	Preconditioner function types CVSpgmrPrecondFn, CVSpgmrPSolveFn
SPGMR		SpgmrMalloc, SpgmrSolve, SpgmrFree
ITERATIV		Routines in support of SPGMR
NVECTOR	PVecInitMPI, PVecFreeMPI, 19 other vector kernels	Type N_Vector; vector macros N_VMAKE, N_VDATA, etc.
LLNLMATH		UnitRoundoff, RPowerI, RPowerR, RSqrt; Macros MIN, MAX, ABS, SQR
LLNLTYPS		Types real, integer, bool

TABLE 1
Modules in the PVODE package

2.3. Usage. We give here a brief summary of the usage of MPI_PVODE by an application written in C. (The usage of SHMEM_PVODE is very similar.) This is not intended as a user manual, and for completed usage information, the reader should see either the PVODE user document [6], or the *CVODE User Guide* [9]. The sample programs should also be helpful in setting up applications for use of PVODE.

The calling program must include several header files so that various constants, macros, and data types can be used. The header files that are always required are: `llnltyps.h`, which defines certain data types and constants; `cvode.h`, which defines several constants related to the integrator and the function type for f; `nvector.h`, which defines the N_vector type and related macros; and `mpi.h`, for MPI-related constants. If the user chooses Newton iteration together with the linear system solver SPGMR, then the calling program must also include `cvspgmr.h`, which defines certain constants and function types related to SPGMR.

The user's program must have the following steps in the order indicated:

1. `MPI_Init(&argc, &argv);` to initialize MPI if used by the user's program. Here `argc` and `argv` are the command line argument counter and array received by `main`.

2. Set `Nlocal`, the local vector length, and `Neq`, the global vector length (the problem size N), and specify the active set of processors.
3. `machEnv = PVecInitMPI(comm, Nlocal, Neq, &argc, &argv);` to initialize the `NVECTOR` module. Here `comm` is the MPI communicator.
4. Set the vector `y` of initial values for the dependent variable in one of two ways. Invoke the macro `N_VMAKE(y, ydata, machEnv);` if an existing array `ydata` contains the values of y. Alternatively, make the call `y = N_VNew(Neq, machEnv);` and load initial values into the array defined by `N_VDATA(y)`.
5. `cvode_mem = CVodeMalloc(...);` which allocates internal memory for CVODE, initializes CVODE, and returns a pointer to the CVODE memory structure. This call specifies the problem size, the name of the routine defining $f(t, y)$, the method options (Adams vs BDF, Newton vs functional iteration), the tolerances, and optional inputs.
6. `CVSpgmr(...);` to invoke Newton iteration with SPGMR. This call specifies the preconditioner type, the names of the user-supplied preconditioner routines, and optional inputs related to SPGMR.
7. `ier = CVode(cvode_mem, tout, y, &t, itask);` for each output point `t = tout` at which the computed solution is desired. The input flag `itask` directs the integrator to either overshoot and interpolate, or take a single step towards `tout` and return.
8. `N_VDISPOSE;` or `N_VFree;` to deallocate the memory for the vector `y`.
9. `CVodeFree(cvode_mem);` to free the memory allocated for CVODE.
10. `PVecFreeMPI(machEnv);` to free machine-dependent data.

Steps 1 – 4 and 10 are specific to the parallel machine environment. Steps 5 – 9, which constitute the bulk of the computation, have exactly the same form here as they do in the use of the sequential solver CVODE.

The user must always provide a C function defining $f(t, y)$. If SPGMR is selected with a preconditioner, then the user must also supply a function that solves linear systems with the preconditioner (or its factors) as the system matrix. If the preconditioner is to use Jacobian-related data that is saved and/or preprocessed, then a second routine must also be supplied for this purpose.

3. KINSOL – a Parallel Nonlinear System Solver. The KINSOL package is a parallel nonlinear system solver callable from either C or Fortran programs. Its most notable feature is that it uses Krylov Inexact Newton techniques in the system's approximate solution, thus sharing significant modules with the PVODE ODE package. It also requires almost no matrix storage for solving the Newton equations as compared to direct methods. The name KINSOL is derived from those techniques: Krylov Inexact Newton SOLver. The package was arranged so that selecting which form of a single module to use will allow the entire package to be compiled in serial (sequential) or parallel form.

3.1. Mathematical preliminaries. The code is a C implementation of a previous code, NKSOL, a nonlinear system solver written in Fortran by Brown and Saad [4]. The module of vector primitives `NVECTOR`, shared with the code PVODE, required several new

primitives for this implementation.

The nonlinear system of equations

$$F(u) = 0, \tag{10}$$

where $F(u)$ is a nonlinear function from $\mathbf{R}^N$ to $\mathbf{R}^N$, is solved by this package. Newton's method is applied to (10) resulting in the following iteration:

Inexact Newton iteration

1. Set $u_0 =$ an initial guess.
2. For $n = 0, 1, 2, ...$ until convergence do:
   (a) Solve $J(u_n)\delta_n = -F(u_n)$,
   (b) Set $u_{n+1} = u_n + \delta_n$,
   (c) Test for convergence,

where $J(u_n) = F'(u_n)$ is the system Jacobian. As this code module is anticipated to be appropriate for large systems, iterative methods are used to solve the system in step 2(a). These solutions are only approximate. Methods of this type used for solution of nonlinear systems are called Inexact Newton methods. At each stage in the iteration process, a multiple of the approximate solution δ_n is added to the previously determined iterated approximate solution to produce a new approximate solution. Convergence is tested before iteration continues.

As only the matrix vector product $J(u)v$ is required in the Krylov method, in this nonlinear equations setting that action is approximated by a difference quotient of the form

$$J(u)v \equiv \frac{F(u + \sigma v) - F(u)}{\sigma}, \tag{11}$$

where u is the current approximation to a root of (10) and σ is a scalar, appropriately chosen to minimize numerical error in the computation of (11). Alternatively, we allow the user to supply, optionally, a routine that computes the product $J(u)v$.

To the above methods are added scaling and preconditioning. Scaling is allowed for both the approximate solution vector and the system function vector. Additionally, right preconditioning is provided for if the preconditioning setup and solve routines are supplied by the user.

While only one linear solver is now implemented for use with this package, the formal structure is in place for alternate solvers. That solver implemented currently is the GMRES solver, where GMRES stands for Generalized Minimal RESidual.

Two methods of applying a computed step δ_n to the previously computed approximate solution vector are implemented. Denoted 'global strategies', they attempt to use the direction implied by δ_n in the most efficient way in furthering convergence of the global (i.e., nonlinear) problem. The first and simplest is the Inexact Newton strategy. A more advanced techniques is implemented in the second strategy, called Linesearch.

In most respects, the KINSOL algorithm is basically that given by Brown and Saad in [4]. But there has been one addition to it — a set of forcing term options developed by Homer Walker [11]. This provides additional choices to the user for stopping the Newton iteration.

3.2. Solver design and development. The package was heavily based on the coding style and structure preexisting in CVODE/PVODE. This was predicated upon the requirement that the same vector kernel implementation and GMRES solvers be used in both codes. At the same time, those features somewhat unique to the Fortran language (e.g., those constructs used in the original code NKSOL), were placed appropriately in a C language setting. Considerable simplification of the calling sequences resulted from this process. Of course, the resulting C language structure maintains relative privacy for definitions for each portion of the code. The resulting code has proven to be readily adaptable to either sequential or parallel execution by means of two versions of the module `NVECTOR`.

As the algorithms of NKSOL had several unique features, notably the way that constraints were handled, several new vector kernels were written and added to the module `NVECTOR`. The changes, completely transparent to CVODE/PVODE, have now been incorporated in the 'common' version of `NVECTOR`.

The code is organized as shown in Table 2. For each module there are two corresponding files. For example, KINSOL requires both the files `kinsol.c` and `kinsol.h`.

Module name	User-callable routines	other contents
KINSOL	KINMalloc, KINSol, KINFree	system function type SysFn; linear solver function pointers linit, lsetup, lsolve, lfree
KINSPGMR	KINSpgmr	KINSpgmrPrecondFn type KINSpgmrPrecondSolveFn type KINSpgmrAtimesFn type
SPGMR		SpgmrMalloc, SpgmrSolve, SpgmrFree
ITERATIV		Routines in support of SPGMR
NVECTOR	PVecInitMPI, PVecFreeMPI, 19 other vector kernels	Type N_Vector; vector macros N_VMAKE, N_VDATA, etc.
LLNLMATH		UnitRoundoff, RPowerI, RPowerR, RSqrt; Macros MIN, MAX, ABS, SQR
LLNLTYPS		Types real, integer, bool

TABLE 2
Modules in the KINSOL package

3.3. Usage. We give here a brief summary of the usage of KINSOL by an application program written in C. This is not intended as a user manual, and for complete usage information, the reader should see the appropriate documentation and header files supplied in the package. The sample programs should also be helpful in setting up applications for use of KINSOL.

As with PVODE, the program calling KINSOL must include several header files so that various constants, macros, and data types can be used. The header files which are always required are: `llnltyps.h`, which defines certain data types and constants; `kinsol.h`, which defines several constants related to the integrator and the function type for F; `nvector.h`,

which defines the N_vector type and related macros; `mpi.h`, for MPI-related constants; and `kinspgmr.h`, which defines certain constants and function types related to SPGMR.

The user's program must have the following steps in the order indicated:

1. `MPI_Init(&argc, &argv);` to initialize MPI if used by the user's program. Here `argc` and `argv` are the command line argument counter and array received by `main`.
2. Set `Nlocal`, the local vector length, and `Neq`, the global vector length (the problem size N), and specify the active set of processors.
3. `machEnv = PVecInitMPI(comm, Nlocal, Neq, &argc, &argv);` to initialize the `NVECTOR` module. Here `comm` is the MPI communicator.
4. Set the vector `uu` of initial values for the dependent variable u in one of two ways. Invoke the macro `N_VMAKE(uu, udata, machEnv);` if an existing array `udata` contains the values of u. Alternatively, make the call `uu = N_VNew(Neq, machEnv);` and load initial values into the array defined by `N_VDATA(uu)`.
5. `kmem = KINMalloc(Neq, msgfp, machEnv);` which allocates internal memory for KINSOL and returns a pointer to the KINSOL memory structure. This call uses the problem size, `Neq`, in allocating memory appropriately.
6. `KINSpgmr(...);` to invoke the SPGMR as the linear solver. This call specifies the names of the user-supplied preconditioner routines, and optional inputs related to SPGMR.
7. `ier = KINSol(kmem, uu, func, ... );` This call invokes the KINSOL solver for the initial guess preloaded into `uu` and for the system defined by the function `func` representing $F(u)$. Other user options are specified here as well.
8. `N_VDISPOSE;` or `N_VFree;` to deallocate the memory for the vector `uu`.
9. `KINFree(kmem);` to free the memory allocated for `KINSol`.
10. `PVecFreeMPI(machEnv);` to free machine-dependent data.

Steps 1 – 4 and 10 are specific to the parallel machine environment. Steps 5 – 9, which constitute the bulk of the computation, are exactly the usage steps in using KINSOL for serial execution.

The user must always provide a C function defining $F(u)$. If a preconditioner is used with SPGMR, then the user must also supply a function that solves linear systems with the preconditioner (or its factors) as the system matrix. If the preconditioner is to use Jacobian-related data that is saved and/or preprocessed, then a second routine must also be supplied for this purpose i.e., a preconditioner setup routine).

4. Preconditioners Based on Domain Decomposition. A critical feature in the use of Krylov iterative methods is the choice of preconditioner. For realistic problems, some non-trivial preconditioner is usually essential to achieve an acceptable rate of convergence. Yet the details of suitable preconditioners tend to be specific to the problem at hand.

We have investigated a type of preconditioner that is fairly effective and yet rather general in scope of applicability. To do this, we restrict our attention to problems that are based on discretized partial differential equations (PDEs) or PDE systems, leading to either large ODE systems or large nonlinear algebraic systems. In the solution of such a system on a multiprocessor, a natural approach is to subdivide the spatial domain into subdomains, and assign all the unknowns associated to one subdomain to one processor.

Given such a domain decomposition, we can build preconditioners for the complete system by building preconditioners on each subdomain separately and combining them as a block-diagonal matrix.

Specifically, suppose that the domain of the computational problem has been subdivided into M non-overlapping subdomains. Each of these subdomains is then assigned to one of the M processors to be used to solve the ODE or algebraic system. Denote the system function by $f(y)$ (or $f(t, y)$ in the ODE case), where for the sake of uniformity, we will use y to denote the vector of unknowns, and f to denote the system function, in both the ODE and algebraic system cases. We wish to approximate the Jacobian $J = \partial f/\partial y$, at least in a local sense on each processor. In practice, however, it may be more cost-effective to work with some approximation to $f(y)$, say $g(y)$, whose partial derivatives are less costly but still sufficiently close to those of f numerically. Corresponding to the domain decomposition, the solution vector y can be partitioned into M non-overlapping blocks y_m, and likewise g has M blocks g_m. The block g_m depends on y_m and also on components of blocks $y_{m'}$ associated with neighboring subdomains (so-called ghost-cell data). Let $\bar{y}_m$ denote y_m augmented with those other components on which g_m depends. Then the function $g(y)$ can be written

$$g(y) = [g_1(\bar{y}_1), g_2(\bar{y}_2), \ldots, g_M(\bar{y}_M)]^T, \tag{12}$$

and each of the functions $g_m(\bar{y}_m)$ is uncoupled from the others.

A simple choice of preconditioner is the block-diagonal matrix

$$P = diag[P_1, P_2, \ldots, P_M] \tag{13}$$

in which each block P_m uses an approximation to the Jacobian of the block function g_m. In the nonlinear system case, where P is intended to approximate J directly, this means we take

$$P_m = J_m \approx \partial g_m/\partial y_m. \tag{14}$$

In the ODE case, P is intended to approximate $I - \gamma J$ for a scalar γ (see Eqn. (5)), so we take

$$P_m = I - \gamma J_m. \tag{15}$$

In either case, we use a difference quotient scheme to generate J_m as a band matrix by way of evaluations of g_m. It has upper and lower half-bandwidths `mu` and `ml`, defined as the number of non-zero diagonals above and below the main diagonal, respectively. The difference quotient approximation is computed using `mu + ml + 2` evaluations of g_m. The parameters `ml` and `mu` need not be the true half-bandwidths of the Jacobian of the local block of g, if smaller values provide a more efficient preconditioner. Also, they need not be the same on every processor. The solution of the complete linear system

$$Px = b \tag{16}$$

reduces to solving each of the equations

$$P_m x_m = b_m \tag{17}$$

and this is done by banded LU factorization of P_m followed by a banded backsolve.

A software module within the PVODE package called PVBBDPRE, and an analogous module within KINSOL called KINBBDPRE, implement this band-block-diagonal preconditioner. To use it, the user must supply, in addition to the right-hand side function f, two functions which the module calls to construct P. One is a function that performs just the inter-processor communications needed for the evaluation of g_m, i.e. of components of $\bar{y}_m$ not included in y_m. The other user function is one that computes $g(y)$ in a distributed manner, i.e. it computes each $g_m(\bar{y}_m)$ on processor m, assuming that necessary communications have been done. The data communicated in the first function must be stored in user-defined space available to the second function.

In using this preconditioner module in conjunction with either PVODE or KINSOL, the user's calling program differs from that outlined in Sections 2.3 and 3.3 in three spots. First, memory allocation and initialization associated with the preconditioner is done with a call to `PVBBDAlloc` or `KBBDAlloc` following the call to `CVodeMalloc` or `KINMalloc`, respectively (Step 5). This call includes the half-bandwidths to be used and the names of the two associated user-supplied functions. Next, the call to `CVSpgmr` or `KINSpgmr` (Step 6) must include the specific names, `PVBBDPrecon` and `PVBBDPSol`, or `KBBDPrecon` and `KBBDPSol`, as the preconditioner routines to be called. Finally, a routine `PVBBDFree` or `KBBDFree` must be called to free the PVBBDPRE/KINBBDPRE memory block. There are also a few optional outputs associated with this module, made available by way of macros, giving workspace sizes and function evaluation counters.

Similar block-diagonal preconditioners could be considered with different treatment of the blocks P_m. For example, incomplete LU factorization or an iterative method could be used instead of banded LU factorization.

Some work has been done by Chow [8] on a somewhat more powerful preconditioner, also based on domain decomposition for PDE problems. In an algorithm called ABLU, for Approximate Block LU factorization, the user supplies an approximate system function g and an associated communication function, as before, but an approximation is implicitly constructed to the elements of the Jacobian matrix which are neglected in the block-diagonal preconditioner. The preconditioner linear system is posed in terms of separate blocks of internal and interface variables, and the solution formulated in terms of the so-called Schur Complement method. The Schur matrix is then approximated as a band matrix, again by way of difference quotients. This algorithm has been implemented for use with PVODE or KINSOL, but only in a preliminary form, with very little testing so far.

5. Support for Fortran Applications. Many of the users, and anticipated users, of PVODE and KINSOL are working with existing Fortran application programs. Our packages are written in C, but such users are reluctant to rewrite their programs in C (a major effort). So in order to apply our software packages to these applications, we have provided a set of interface routines that make the connections between the Fortran programs and the C solvers with a minimum of changes to the application programs.

Mixing Fortran and C requires some compromises in portability, because compilers do not hold to any one standard for the linkages across this language boundary. Moreover, our interfaces must cross that boundary in both directions, since the Fortran user program calls

the C solver, and the C solver calls Fortran user routines. We have kept the difficulties to a minimum by passing as arguments only scalars and arrays. This forces the Fortran user to use fixed names for user-supplied routines, rather than arbitrary names which are passed to the solver.

The remaining issue in the Fortran/C interfaces is the naming of externals in the linkages. While there is no universal standard for this, there are only a few different conventions in use by current compilers. In most cases, the Fortran compiler uses lower case names for all externals, and appends an underscore to the names of C external routines and to the names of compiled Fortran routines. On the other hand, Cray compilers use upper case names, with no appended underscore. We have developed a scheme that accommodates these naming conventions, with just a few machine-dependent lines of code. These are isolated in a single header file, which defines a pair of parameters. The values of these parameters are then used in other header files to set a sequence of dummy names to the appropriate actual names for the various routines that are called across the Fortran/C language boundary.

5.1. Fortran interfaces for PVODE. We have written a package of interface routines called FPVODE to support Fortran users of PVODE. This is a collection of C language functions and header files which enables the user to write a main program and all user-supplied subroutines in Fortran and to use the C language PVODE package. A small additional set of interfaces called FPVBBD consists of interfaces needed by a Fortran application to use the PVBBDPRE preconditioner package in combination with PVODE. The organization of these modules is summarized here.

The following is a list of the functions which are callable from the user's Fortran program, along with the routine in PVODE that is called by each.

- `FPVECINITMPI` interfaces with `PVecInitMPI` and is used to initialize the `NVECTOR` module.
- `FPVMALLOC` interfaces with `CVodeMalloc` and is used to initialize `CVode`.
- `FCVDIAG` interfaces with `CVDiag` and is used when the diagonal approximate Jacobian has been selected.
- `FCVSPGMR0, FCVSPGMR1, FCVSPGMR2` interface with `CVSpgmr` when SPGMR has been chosen as the linear system solver. These three interface routines correspond to the cases of no preconditioning, preconditioning with no saved matrix data, and preconditioning with saved matrix data, respectively.
- `FCVODE` interfaces with `CVode`.
- `FCVDKY` interfaces with `CVodeDky` and is used to compute a derivative of order $k, 0 \leq k \leq$ `qu`, where `qu` is the order used for the most recent time step. The derivative is calculated at the current output time.
- `FCVFREE` interfaces with `CVodeFree` and is used to free memory allocated for `CVode`.
- `FPVECFREEMPI` interfaces with `PVecFreeMPI` and is used to free memory allocated for MPI.

The user-supplied Fortran subroutines are listed below. As explained above, the names of these Fortran routines are fixed and are case-sensitive.

- `PVFUN` which defines the function f, the right-hand side function of the system of ODEs.

- PVPSOL which solves the preconditioner equation, and is required if preconditioning is used.
- PVPRECO which computes the preconditioner, and is required if preconditioning involves pre-computed matrix data.

A similar interface package, called FPVBBD, has been written for the PVBBDPRE preconditioner module. It works in conjunction with the FPVODE interface package. The three additional user-callable functions here are: FPVBBDIN, which interfaces to PVBBDAlloc and CVSpgmr; FPVBBDOPT, which accesses optional outputs; and FPVBBDF, which interfaces to PVBBDFree. The two user-supplied Fortran subroutines required, in addition to PVFUN to define f, are: PVLOCFN, which computes $g(t, y)$; and PVCOMMFN, which performs communications necessary for PVLOCFN.

5.2. Fortran interfaces for KINSOL. We have written a package of interface routines called FKINSOL to support Fortran users of KINSOL. This is a collection of C language functions and header files which enables the user to write a main program and all user-supplied subroutines in Fortran and to use the C language KINSOL package. A small additional set of interfaces called FKINBBD consists of interfaces needed by a Fortran application to use the KINBBDPRE preconditioner package in combination with KINSOL. The organization of these modules is summarized here.

The following is a list of the functions which are callable from the user's Fortran program, along with the routine in KINSOL that is called by each.

- FPVECINITMPI interfaces with PVecInitMPI and is used to initialize the NVECTOR module.
- FPKINMALLOC interfaces with KINMalloc and is used to allocate memory for KINSol.
- FKINSPGMR00, FKINSPGMR01, FKINSPGMR10, FKINSPGMR11, FKINSPGMR20, FKINSPGMR21 interface with KINSpgmr, the linear solver. The three ending in 0 correspond to the cases of no preconditioning, preconditioning with no setup, and preconditioning with setup, respectively. The three ending in 1 are the corresponding routines in the case that a user-supplied KATIMES routine is supplied.
- FKINSOL interfaces with KINSol.
- FKINFREE interfaces with KINFree and is used to free memory allocated for KINSol.
- FPVECFREEMPI interfaces with PVecFreeMPI and is used to free memory allocated for MPI.

The user-supplied Fortran subroutines are listed below. As explained above, the names of these Fortran routines are fixed and are case-sensitive.

- KFUN which defines the function F, the nonlinear system function.
- KPSOL which solves the preconditioner equation, and is required if preconditioning is used.
- KPRECO which computes the preconditioner, and is required if preconditioning involves pre-computed matrix data.
- KATIMES which is the user-supplied Jacobian-vector multiply routine required if that option is exercised.

A similar interface package, called FKINBBD, has been written for the KINBBDPRE preconditioner module. It works in conjunction with the FKINSOL interface package. The

additional user-callable functions here are: FKINBBDINIT0 and FKINBBDINIT1 which interface to KBBDAlloc and KINSpgmr; FKINBBDOPT, which accesses optional outputs; and FKINBBDFREE, which interfaces to KBBDFree. The two user-supplied Fortran subroutines required, in addition to KFUN to define F, are: KLOCFN, which computes $g(t, y)$; and KCOMMFN, which performs communications necessary for KLOCFN.

The following is a summary of parallel usage of KINSOL, using the Fortran interface:

1. call mpi_init(...): Initialize MPI.
2. call fpvecinitmpi(nlocal, neq, ier): Initialize the NVECTOR interface to MPI. Here, nlocal and neq are the local and global sizes of the dependent variable arrays to be used.
3. call mpi_comm_size(...) or call mpi_comm_rank(...): Optional calls to determine logical processor number and count.
4. call fpkinmalloc(...): Allocate space for KINSOL.
5. call fkinspgmr20(...): Set up the linear solver. The choice of linear solver interface routine taken here, one of six possible, is for both a setup and solve preconditioner routine to be supplied by the user in Fortran, but with default (internal) Jacobian-vector product.
6. call fkinsol(...): Call KINSOL, through the Fortran interface.
7. call fkinfree: Free memory usage by KINSOL and its Fortran interface
8. call fpvecfreempi: Free MPI interface

6. Examples and Tests.

6.1. PVODE examples. We have used three example problems to demonstrate the PVODE package. All are based on the treatment of a PDE by the Method of Lines.

The first example problem is a nonstiff ODE system derived from a single PDE in one space dimension. The PDE is a prototypical diffusion-advection equation for $u = u(t, x)$,

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 0.5\frac{\partial u}{\partial x} \tag{18}$$

for $0 \le x \le 2$, $0 \le t \le 5$, and subject to homogeneous Dirichlet boundary conditions and initial values, given by:

$$\begin{aligned} u(t,0) &= 0, \quad u(t,2) = 0, \\ u(0,x) &= x(2-x)\exp(2x). \end{aligned} \tag{19}$$

A system of M ODEs is obtained by discretizing the x-axis with $M + 2$ grid points and replacing the first and second order spatial derivatives in (18) with their standard central difference approximations at all interior grid points. The values at the two boundaries are eliminated by way of (19).

In the parallel implementation of this example, the function $f(t, y)$ is evaluated in a distributed manner, with each processor owning a block of components of y and f. Prior to the evaluation of f, message-passing calls are made to communicate the first and last component of each block of y to the neighboring processor (except for boundary blocks, where

only one component is passed). Then each processor evaluates the three-point difference expression corresponding to the right-hand side of (18) for its block of components of f.

The file `pvnx.c` is included in the PVODE package as the example program for this problem. It uses the Adams (non-stiff) integration formula and functional iteration. The problem is unrealistically small, but serves as a simple example for both the Method of Lines and the use of PVODE.

The second example is a stiff system derived from a system of two coupled PDEs in two space dimensions, involving diurnal kinetics, advection, and diffusion. The PDEs can be written as

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V\frac{\partial c^i}{\partial x} + \frac{\partial}{\partial y}K_v(y)\frac{\partial c^i}{\partial y} + R^i(c^1, c^2, t) \quad (i = 1, 2), \tag{20}$$

The spatial domain is $0 \le x \le 20,\ 30 \le y \le 50$. The time interval of integration is $[0, 86400]$, representing 24 hours measured in seconds. These equations represent a simplified model for the transport, production, and loss of the oxygen singlet and ozone in the upper atmosphere. Homogeneous Neumann boundary conditions are imposed on each boundary, along with simple polynomial initial conditions. We omit the details, which can be found elsewhere (see [3] and references cited there).

As before, we discretize the PDE system with central differencing, to obtain an ODE system $\dot{u} = f(t, u)$ representing (20). A five-point stencil is involved in each finite difference expression. For this example, we think of the processors as being laid out in a rectangular array, and each processor being assigned a subgrid of size $MXSUB \times MYSUB$ of the $x - y$ grid. If the array of processors is $NPEX \times NPEY$, then the overall grid size is $MX \times MY$ with $MX = NPEX \cdot MXSUB$ and $MY = NPEY \cdot MYSUB$. There are $2 \cdot MX \cdot MY$ equations in this system of ODEs. To compute f in this setting, the processors pass and receive information as follows. In each processor, the solution components on the top and bottom rows of subgrid points, and those on the leftmost and rightmost columns of subgrid points, are passed to the corresponding neighboring processor which borders the given processor. Of course if the processor is on an edge of the processor array, then only a subset of these communications is performed. Some care has been taken to implement these steps with MPI calls which maximize efficiency and minimize the possibility of a deadlock. In the terminology of MPI, the sequence of calls is: non-blocking receives, then blocking sends, then waiting on non-blocking receives. Once all of the communications have been done, the evaluation of all components of f can proceed.

The program for this example is provided in the file `pvkx.c` in the PVODE package. The purpose of this code is to provide a more complicated example than the first one and to provide a template for a stiff ODE system arising from a PDE system. The solution method is BDF with Newton iteration and SPGMR. Preconditioning is on the left, and the preconditioner matrix is the block-diagonal part of the Newton matrix, with 2×2 blocks. The corresponding diagonal blocks of the Jacobian are saved each time the preconditioner is generated, for re-use later under certain conditions.

The organization of the `pvkx.c` program deserves some comments. The right-hand side routine `f` calls two other routines: `ucomm`, which carries out inter-processor communication; and `fcalc`, which operates on local data only and contains the actual calculation of $f(t, u)$.

The `ucomm` function in turn calls three routines which do, respectively, non-blocking receive operations, blocking send operations, and receive-waiting. All three use MPI, and transmit data from the local `u` vector into a local working array `uext`, an extended copy of `u`. The `fcalc` function copies `u` into `uext`, so that the calculation of $f(t, u)$ can be done conveniently by operations on `uext` only.

The third PVODE example uses the same ODE system as the second, but a slightly different solution method. It uses the PVBBDPRE preconditioner module to generate a band-block-diagonal left preconditioner. For the half-bandwidths `ml` and `mu` supplied to PVBBDPRE, we experimented with values ranging from 1 up to the true value of half-bandwidths of the diagonal Jacobian blocks, namely $2 \cdot MXSUB$. The most cost-effective choice was a value of `ml = mu = 2`, giving a band matrix that is only slightly wider than the block-diagonal matrix of the second example. The program for this example is provided as `pvkxb.c` in the PVODE package.

We have also used a simple diagonal ODE system in order to validate the Fortran/C interfaces. The system is given by $\dot{y}_i = -\alpha i y_i$ for $i = 1, \ldots, N$. We supply example Fortran programs for three cases:

- a nonstiff case, with $\alpha = 10/N$;
- a stiff case, with $\alpha = 10$, and a diagonal preconditioner that includes the correct Jacobian element for positions 4 through N only; and
- the same stiff case, solved with the PVBBDPRE module (with half-bandwidths set to zero).

6.2. KINSOL examples. Four sample application programs have been written solving the same simple, nonlinear diagonal problem. They solve the system with the function

$$F(u) = (F_i(u)) \quad , \quad F_i(u) = (u_i)^2 - i^2 \qquad (i = 1, \ldots, N). \tag{21}$$

This problem was solved using a serial implementation with file `diags2.c`, using a parallel implementation with file `diagp2.c`, using a parallel implementation via the Fortran interface package with file `diagp2f.f`, and using the parallel/Fortran interface with the Block-Banded-Diagonal preconditioner package in file `diagbbdf.f`.

A more demanding example is that of the so-called predator-prey PDE system. This example problem is a model of a multi-species food web [2], in which mutual competition and/or predator-prey relationships in a spatial domain are simulated. For this problem the dependent variable c replaces the generic dependent variable u used above. Here we consider a model with $s = 2p$ species, where both species $1, \cdots, p$ (the prey) and $p+1, \cdots, s$ (the predators) have infinitely fast reaction rates:

$$\begin{cases} 0 = f_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) & (i = 1, 2, \cdots, p), \\ 0 = f_i(x, y, c) + d_i(c^i_{xx} + c^i_{yy}) & (i = p+1, \cdots, s), \end{cases} \tag{22}$$

with

$$f_i(x, y, c) = c^i(b_i + \sum_{j=1}^{s} a_{ij} c^j). \tag{23}$$

The interaction and diffusion coefficients (a_{ij}, b_i, d_i) could be functions of (x, y) in general. The choices made for this test problem are for a simple model of p prey and p predator species, arranged in that order in the vector c. We take the various coefficients to be as follows:

$$\begin{cases} a_{ii} = -1 \quad (\text{all } i) \\ a_{ij} = -0.5 \cdot 10^{-6} \quad (i \le p, j > p) \\ a_{ij} = 10^4 \quad (i > p, j \le p) \end{cases} \tag{24}$$

(all other $a_{ij} = 0$),

$$\begin{cases} b_i = b_i(x, y) = (1 + \alpha xy) \quad (i \le p) \\ b_i = b_i(x, y) = -(1 + \alpha xy \quad (i > p) \end{cases} \tag{25}$$

and

$$\begin{cases} d_i = 1 \quad (i \le p) \\ d_i = 0.5 \quad (i > p). \end{cases} \tag{26}$$

The domain is the unit square $0 \le x, y \le 1$. The boundary conditions are of Neumann type (zero normal derivatives) everywhere. The coefficients are such that a unique stable equilibrium is guaranteed to exist when α is zero [2]. Empirically, for (22) a stable equilibrium appears to exist when α is positive, although it may not be unique. In this problem we take $\alpha = 1$. The initial conditions used for this problem are taken to be constant functions by species type. These satisfy the boundary conditions and very nearly satisfy the constraints, given by

$$\begin{aligned} c^i &= 1.16347 \quad (i = 1, \cdots, p) \\ c^i &= 34903.1 \quad (i = p + 1, \cdots, s). \end{aligned}$$

The PDE system (22) (plus boundary conditions) was discretized with central differencing on an $L \times L$ mesh, with the resulting nonlinear system has size $N = sL^2$.

6.3. PVODE testing. The stiff example problem described in Section 6.1 has been modified and expanded to form a test problem for PVODE. This work was largely carried out by M. Wittman and reported in [17].

To start with, in order to add realistic complexity to the solution, the initial profile for this problem was altered to include a rather steep front in the vertical direction. Specifically, in the initial profile, the polynomial dependence on y was replaced by the function

$$\beta(y) = .75 + .25 \tanh(10y - 400). \tag{27}$$

This function rises from about .5 to about 1.0 over a y interval of about .2 (i.e. 1/100 of the total span in y). This vertical variation, together with the horizonatal advection and diffusion in the problem, demands a fairly fine spatial mesh to achieve acceptable resolution.

In addition, an alternate choice of differencing is used in order to control spurious oscillations resulting from the horizontal advection. In place of central differencing for that term, a biased upwind approximation is applied to each of the terms $\partial c^i / \partial x$.

With this modified form of the problem, we performed tests similar to that described above for the example. Here we fix the subgrid dimensions at `MXSUB = MYSUB =` 50, so that the local (per-processor) problem size is 5000, while the processor array dimensions, `NPEX` and `NPEY`, are varied. In one (typical) sequence of tests, we fix `NPEY` = 8 (for a vertical mesh size of `MY` = 400), and take three cases: `NPEX` = 8 (`MX` = 400), `NPEX` = 16 (`MX` = 800), and `NPEX` = 32 (`MX` = 1600). Thus the largest problem size N is $2 \cdot 400 \cdot 1600 = 1,280,000$. For these tests, we also raise the maximum Krylov dimension, `maxl`, to 10 (from its default value of 5).

For each of the three test cases, the test program was run on a Cray-T3D (256 processors) with each of three different message-passing libraries:

- MPICH: an implemenation of MPI on top of the Chameleon library [12]
- EPCC: an implemenation of MPI by the Edinburgh Parallel Computer Centre [7]
- SHMEM: Cray's Shared Memory Library

The following table gives the run time and selected performance counters for these 9 runs. In all cases, the solutions agreed well with each other, showing expected small variations with grid size. In the table, M-P denotes the message-passing library, RT is the reported run time in CPU seconds, `nst` is the number of time steps, `nfe` is the number of f evaluations, `nni` is the number of nonlinear (Newton) iterations, `nli` is the number of linear (Krylov) iterations, and `npe` is the number of evaluations of the preconditioner.

NPEX	M-P	RT	nst	nfe	nni	nli	npe
8	MPICH	436.	1391	9907	1512	8392	24
8	EPCC	355.	1391	9907	1512	8392	24
8	SHMEM	349.	1999	10,326	2096	8227	34
16	MPICH	676.	2513	14,159	2583	11,573	42
16	EPCC	494.	2513	14,159	2583	11,573	42
16	SHMEM	471.	2513	14,160	2581	11,576	42
32	MPICH	1367.	2536	20,153	2696	17,454	43
32	EPCC	737.	2536	20,153	2696	17,454	43
32	SHMEM	695.	2536	20,121	2694	17,424	43

TABLE 3
PVODE test results vs problem size and message-passing library

Some of the results were as expected, and some were surprising. For a given mesh size, variations in performance counts were small or absent, except for moderate (but still acceptable) variations for SHMEM in the smallest case. The increase in costs with mesh size can be attributed to a decline in the quality of the preconditioner, which neglects most of the spatial coupling. The preconditioner quality can be inferred from the ratio `nli/nni`, which is the average number of Krylov iterations per Newton iteration. The most interesting (and unexpected) result is the variation of run time with library: SHMEM is the most efficient, but EPCC is a very close second, and MPICH loses considerable efficiency by comparison, as the problem size grows. This means that the highly portable MPI version of PVODE, with an appropriate choice of MPI implementation, is fully competitive with the Cray-specific

version using the SHMEM library. While the overall costs do not prepresent a well-scaled parallel algorithm (because of the preconditioner choice), the cost per function evaluation is quite flat for EPCC and SHMEM, at .033 to .037 (for MPICH it ranges from .044 to .068).

For tests that demonstrate speedup from parallelism, we consider runs with fixed problem size: `MX` = 800, `MY` = 400. Here we also fix the vertical subgrid dimension at `MYSUB` = 50 and the vertical processor array dimension at `NPEY` = 8, but vary the corresponding horizontal sizes. We take `NPEX` = 8, 16, and 32, with `MXSUB` = 100, 50, and 25, respectively. The runs for the three cases and three message-passing libraries all show very good agreement in solution values and performance counts. The run times for EPCC are 947, 494, and 278, showing speedups of 1.92 and 1.78 as the number of processors is doubled (twice). For the SHMEM runs, the times were slightly lower and the ratios were 1.98 and 1.91. For MPICH, consistent with the earlier runs, the run times were considerably higher, and in fact show speedup ratios of only 1.54 and 1.03.

7. Applications.

7.1. Application to Tokamak Edge Plasma Models. We are working with LLNL's Magnetic Fusion Energy Division on parallel software for 2-D tokamak plasma simulation. The MFE community's primary tokamak edge model, UEDGE, now uses three of our solvers in its sequential version: the nonlinear algebraic system solver NKSOL, the ODE solver VODPK, and the DAE solver DASPK. A preliminary parallel version of UEDGE using both PVODE and KINSOL has been completed [14, 15]. It makes use of the band-block-diagonal preconditioner modules in combination with both PVODE and KINSOL, and also uses our Fortran interface packages.

Development and testing have now been done for two cases: (a) single-region problems and (b) multiply-connected domains that arise in the tokamak geometry. The tests show that the algorithm is behaving correctly. Work is now being done to document the scaling of the problem on larger meshes and improving the user interface with the Cray-T3E, since the previously-used BASIS system is not available there. When fully developed, this software should enable problem sizes sufficient to resolve boundary features and impurity effects that are not adequately resolved now. In addition, this MFE group is working to develop a parallel version of their 3-D plasma fluid turbulence code, BOUT, by utilizing PVODE.

7.2. Application to a Variably Saturated Flow Model. We are working with Carol Woodward, CASC, in applying KINSOL to problems in modeling groundwater flow [18]. Specifically, a nonlinear Richards' equation representing the pressure field in a variably saturated three-dimensional medium is treated by implicit differencing in time and finite differences in space. The resulting nonlinear algebraic system is solved with KINSOL. Preconditioning is done with a multigrid algorithm applied to the symmetric part of the Jacobian.

7.3. Other Applications. The parallel preconditioned GMRES solver SPGMR used within PVODE and KINSOL is suitable as a general-purpose linear system solver. As such, it has been used in a simulation of a nonlinear, steady-state, plasma fluid problem. It has also been used in the three-dimensional Boltzmann transport solver Ardra.

REFERENCES

[1] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh, *VODE, a Variable-Coefficient ODE Solver*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1038–1051.

[2] P. N. Brown, *Decay to Uniform States in Food Webs*, SIAM J. Appl. Math., 46 (1986), 376-392.

[3] P. N. Brown and A. C. Hindmarsh, *Reduced Storage Matrix Methods in Stiff ODE Systems*, J. Appl. Math. & Comp. 31 (1989), pp. 40–91.

[4] P. N. Brown and Y. Saad. *Hybrid Krylov methods for nonlinear systems of equations*, SIAM J. Sci Stat. Comput., 11 (1990), pp. 450-481.

[5] George D. Byrne, *Pragmatic Experiments with Krylov Methods in the Stiff ODE Setting*, in *Computational Ordinary Differential Equations*, J. R. Cash and I. Gladwell (Eds.), Oxford University Press, Oxford, 1992, pp. 323–356.

[6] George D. Byrne and Alan C. Hindmarsh, *User Documentation for PVODE, an ODE Solver for Parallel Computers*, LLNL informal document, in progress, 1998.

[7] K. Cameron, L. J. Clarke, and A. G. Smith, *Using MPI on the Cray T3D*, Edinburgh Parallel Computing Centre informal document, November 1995.

[8] E. T. Chow, private communication.

[9] S. D. Cohen and A. C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory report UCRL-MA-118618, September 1994.

[10] Scott D. Cohen and Alan C. Hindmarsh, *CVODE, a Stiff/Nonstiff ODE Solver in C*, Computers in Physics, 10, No. 2 (1996), pp. 138-143.

[11] S. Eisenstat and H. Walker, *Choosing the Forcing Terms in an Inexact Newton Method*, SIAM J. Sci. Comp., 17 (1996), pp. 16-32.

[12] W. D. Gropp and E. Lusk, *A Test Implementation of the MPI Draft Message-Passing Standard*, Technical Report ANL-92/47, Argonne National Laboratory, December 1992.

[13] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI Portable Parallel Programming with the Message-Passing Interface,* The MIT Press, Cambridge, MA, 1994.

[14] T.D. Rognlien, X. Xu, P. N. Brown, A. C. Hindmarsh, and A. G. Taylor, *Parallelization of an Edge Plasma Transport Code via Domain Decomposition*, Bull. Am. Phys. Soc., Vol. 42, 1584 (1997) (Abst. for Int. Conf. on Comp. Phys., Aug. 25-28, 1997, Santa Cruz, CA); LLNL Report UCRL-JC-127375abs.

[15] T. D. Rognlien, X. Xu, A. C. Hindmarsh, P. N. Brown, and A. G. Taylor, *Algorithms and Results for a Parallelized Fully-Implicit Edge-Plasma Transport Code*, Abst. for Int. Conf. Num. Sim. Plasmas, Feb. 10-12, 1998, Santa Barbara, CA; LLNL Report UCRL-JC-129223abs.

[16] Y. Saad and M. H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comp. 7 (1986), pp. 856–869.

[17] Michael R. Wittman, *Testing of PVODE, a Parallel ODE Solver,* Lawrence Livermore National Laboratory report UCRL-ID-125562, August 1996.

[18] Carol S. Woodward, *A Newton-Krylov-Multigrid Solver for Variably Saturated Flow Problems*, Lawrence Livermore National Laboratory report UCRL-JC-129371, January 1998.