User Documentation for PVODE, An ODE Solver for Parallel Computers

George D. Byrne
Illinois Institute of Technology

Alan C. Hindmarsh
Lawrence Livermore National Laboratory

Center for Applied Scientific Computing

UCRL-ID-130884
May 1998

USER DOCUMENTATION FOR PVODE, AN ODE SOLVER FOR PARALLEL COMPUTERS*

GEORGE D. BYRNE† AND ALAN C. HINDMARSH‡

1. Introduction. PVODE is a general purpose ordinary differential equation (ODE) solver for stiff and nonstiff ODEs. It is based on CVODE [5] [6], which is written in ANSI-standard C. PVODE uses MPI (Message-Passing Interface) [8] and a revised version of the vector module in CVODE to achieve parallelism and portability. PVODE is intended for the SPMD (Single Program Multiple Data) environment with distributed memory, in which all vectors are identically distributed across processors. In particular, the vector module is designed to help the user assign a contiguous segment of a given vector to each of the processors for parallel computation. The idea is for each processor to solve a certain fixed subset of the ODEs.

To better understand PVODE, we first need to understand CVODE and its historical background. The ODE solver CVODE, which was written by Cohen and Hindmarsh, combines features of two earlier Fortran codes, VODE [1] and VODPK [3]. Those two codes were 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 [2] 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). Both CVODE and PVODE are written in such a way that other linear algebraic techniques could be easily incorporated, since the code is written with a layer of linear system solver modules that is isolated, as far as possible, from the rest of the code. Further, the code is structured so that it can readily be converted from double precision to single precision. This precludes the maintenance of two versions of PVODE.

PVODE has been run on an IBM SP2, a Cray-T3D and Cray-T3E, and a cluster of workstations. It is currently being used in a simulation of tokamak edge plasmas at LLNL. (We are grateful to Dr. Michael Minkoff at Argonne National Laboratory for assistance in the use of the IBM SP2 there.) Recently, the PVODE solver was incorporated into the

* 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.

† Computer Science and Applied Mathematics Department, Illinois Institute of Technology, Chicago, IL 60616

‡ Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551

PETSc package (Portable Extensible Toolkit for Scientific computation) [9] developed at Argonne.

The remainder of this paper is organized as follows: Section 2 sets the mathematical notation and summarizes the basic methods. Section 3 summarizes the organization of the PVODE solver, while Section 4 summarizes its usage. Section 5 describes a preconditioner module, and Section 6 describes a set of Fortran/C interfaces. Section 7 describes two example problems, and Section 8 gives some test results.

2. Mathematical Considerations. PVODE solves initial-value problems (IVPs) for systems of ODEs. Such problems can be stated as

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

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 independent variable need not appear explicitly in the N-vector valued function f.

The IVP is solved by one of two numerical methods. These are the backward differentiation formula (BDF) and the Adams-Moulton formula. Both are implemented in a variable-stepsize, variable-order form. The BDF uses a fixed-leading-coefficient form. These formulas 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) [2], which is a Krylov subspace method. In most cases, performance of SPGMR is improved by user-supplied preconditioners. The user may precondition the system on the left, on the right, on both the left and right, or use no preconditioner.

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

$$\|E_n\|_{rms,w} < 1 \ . \tag{6}$$

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

$$\|E_n\|_{rms,w} = \left[\sum_{i=1}^{N} \frac{1}{N} \left(w_i E_{n,i}\right)^2\right]^{1/2} \tag{7}$$

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

$$w_i = \frac{1}{rtol|y_i| + atol_i} \ . \tag{8}$$

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. The value for $rtol$ indicates the number of digits of relative accuracy for a single time step. The specified value for $atol_i$ indicates the values of the corresponding component of the solution vector which may be thought of as being zero, or at the noise level. In particular, if we set $atol_i = rtol \times floor_i$ then $floor_i$ represents the floor value for the ith component of the solution and is that magnitude of the component for which there is a crossover from relative error control to absolute error control. Since these tolerances define the allowed error per step, they should be chosen conservatively. Experience indicates that a conservative choice yields a more economical solution than error tolerances that are too large.

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

3. Code Organization. One way to visualize PVODE is to think of the code as being organized in layers, as shown in Fig. 1. 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. Several supporting modules reside at the fourth level. The LLNLTYPS module defines types `real`, `integer`, and `boole` (boolean), and facilitates changing the precision of the arithmetic in the package from double to single, or the reverse. The LLNLMATH module specifies power functions and provides a function to compute the machine unit roundoff. Finally, the NVECTOR module is discussed below.

The key to being able to move from the sequential computing environment to the parallel computing environment lies in the NVECTOR module. The idea is to distribute the system of ODEs over the several processors so that each processor is solving a contiguous subset of the system. This is achieved through the NVECTOR module, which 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 version of PVODE described so far uses the MPI (Message Passing Interface) system [8] for all inter-processor communication. This achieves a high degree of portability, since MPI is becoming widely accepted as a standard for message passing software. In addition, however, we have prepared a version for the Cray-T3D and -T3E using the Cray Shared Memory (SHMEM) Library. This involves a separate version of the vector module based on SHMEM instead of MPI.

For a different parallel computing environment, some rewriting of the vector module could allow the use of other specific machine-dependent instructions.

4. Using PVODE. This section is concerned with the use of PVODE and consists of three subsections. These treat the header files, the layout of the user's main program, and user-supplied functions or routines. For further details not specific to the parallel extensions, the reader should see the *CVODE User Guide* [5]. The listing of a sample program in the Appendix may also be helpful. That code is intended to serve as a template and is included in the PVODE package.

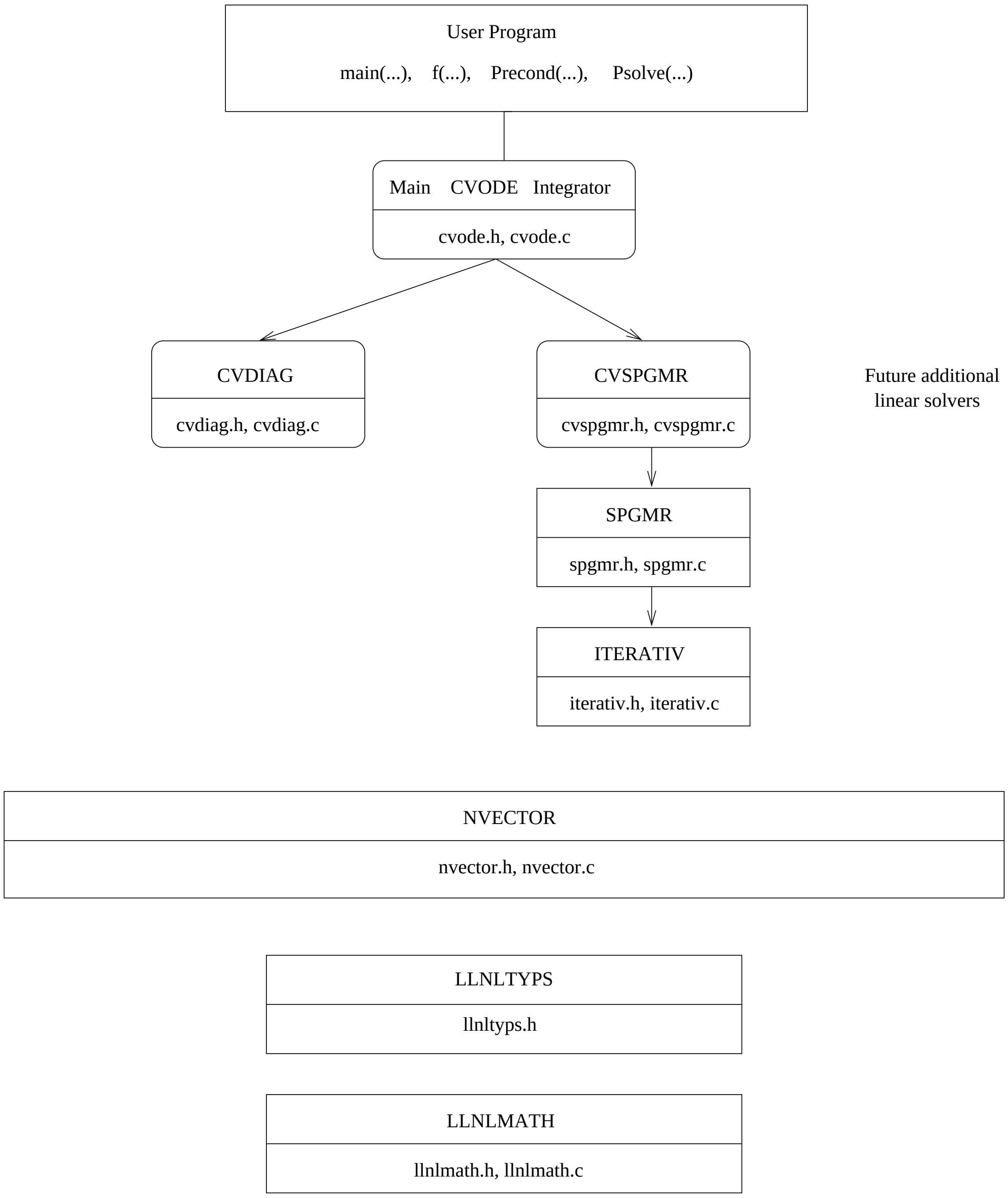

FIG. 1. *Overall structure of the PVODE package. Modules comprising the central solver are distinguished by rounded boxes, while the user program, generic linear solvers, and auxiliary modules are in unrounded boxes.*

4.1. Header Files. The calling program must include several header files so that various macros and data types can be used. The header files that are always required are:

- `llnltyps.h`, which defines the types `real`, `integer`, `boole` (for boolean), and constants `FALSE` and `TRUE`
- `cvode.h`, the header file for CVODE, which defines the several types and various constants, and includes function prototypes
- `nvector.h`, the header file for the NVECTOR module outlined above
- `mpi.h`, the MPI header file

If the user chooses Newton iteration together with the linear system solver SPGMR, then (minimally) the following header file will be required by CVODE:

- `cvspgmr.h`, which is used with the Krylov solver SPGMR in the context of PVODE. This in turn includes a header file (`iterativ.h`) which enumerates the kind of preconditioning and the choices for the Gram-Schmidt process.

Other headers may be needed, according as to the choice of preconditioner, etc. In one of the examples to follow, preconditioning is done with a block-diagonal matrix. For this, the header `smalldense.h` is included.

4.2. A Skeleton of the User's Main Program. The user's program must have the following steps in the order indicated:

- `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`.
- Set `Nlocal`, the local vector length (the sub-vector length for this processor); `neq`, the global vector length (the problem size N, and the sum of all the values of `Nlocal`); and the active set of processors.
- `machEnv = PVecInitMPI(comm, Nlocal, neq, &argc, &argv);` to initialize the NVECTOR module. Here `comm` is the MPI communicator, set in one of two ways: If a proper subset of active processors is to be used, `comm` must be set by suitable MPI calls. Otherwise, to specify that all processors are to be used, `comm` must be either `MPI_COMM_WORLD` or `NULL`.
- Set the vector `y` of initial values. Use the macro `N_VMAKE(y, ydata, machEnv);` if an existing array `ydata` contains the initial values of y. Otherwise, make the call `y = N_VNew(neq, machEnv);` and load initial values into the array defined by `N_VDATA(y)`.
- `cvode_mem = CVodeMalloc(...);` which allocates internal memory for CVODE, initializes CVODE, and returns a pointer to the CVODE memory structure. (See details below.)
- `CVSpgmr(...);` if Newton iteration is chosen. (See details below.)
- `ier = CVode(cvode_mem, tout, y, &t, itask);` for each point `t = tout` at which output is desired. Set `itask` to `NORMAL` to have the integrator overshoot `tout` and interpolate, or `ONE_STEP` to take a single step and return.
- `N_VDISPOSE;` or `N_VFree;` upon completion of the integration, to deallocate the memory for the vector `y`, allocated by `N_VMAKE` or `N_VNew`, respectively.
- `CVodeFree(cvode_mem);` to free the memory allocated for CVODE.
- `PVecFreeMPI(machEnv);` to free machine-dependent data.

The form of the call to CVodeMalloc is

```
cvode_mem = CVodeMalloc(neq, f, t0, y0, lmm, iter, itol, &rtol,
                        atol, f_data, errfp, optIn, iopt, ropt, machEnv)
```

where neq is the number of ODEs in the system, f is the C function to compute f in the ODE, t0 is the initial value of t and y0 is the initial value of y (which can be the same as the vector y described above). The flag lmm is used to select the linear multistep method and may be one of two possible values: ADAMS or BDF. The type of iteration is selected by replacing iter with either NEWTON or FUNCTIONAL. The next three parameters are used to set the error control. The flag itol is replaced by either SS or SV, where SS indicates scalar relative error tolerance and scalar absolute error tolerance, while SV indicates scalar relative error tolerance and vector absolute error tolerance. The latter choice is important when the absolute error tolerance needs to be different for each component of the ODE. The arguments &rtol and atol are pointers to the user's error tolerances, and f_data is a pointer to user-defined space passed directly to the user's f function. The file pointer errfp points to the file where error messages from CVODE are to be written (NULL for stdout). iopt and ropt are integer and real arrays for optional input and output. If optIn is replaced by FALSE, then the user is not going to provide optional input, while if it is TRUE then optional inputs are examined in iopt and ropt. The final argument, machEnv, is a pointer to machine environment-specific information.

The form of the call to CVSpgmr is

```
CVSpgmr(cvode_mem, pretype, gstype, maxl, delt, Precond, PSolve, P_data)
```

Here pretype specifies the preconditioning type, with values NONE, LEFT, RIGHT, or BOTH; and gstype specifies the Gram-Schmidt orthogonalization type, with values MODIFIED_GS or CLASSICAL_GS. The arguments maxl and delt are optional inputs for the maximum Krylov dimension and the SPGMR convergence test constant, respectively. P_data is a pointer to user-defined space which PVODE passes to the user's preconditioning functions for use there.

4.3. User-Supplied Functions. The user-supplied routines consist of one function defining the ODE, and (optionally) one or two functions that define the preconditioner for use in the SPGMR algorithm. The first of these C functions defines f in (1) and must be of type RhsFn. The form of this C function is:

```
void f(integer N, real t, N_Vector y, N_Vector ydot, void *f_data)
```

This function has as input the number of ODEs N, the value of the independent variable t, and dependent variable vector y. The computed value of $f(t, y)$ is stored in the N-vector ydot. The pointer f_data was seen previously in the call to CVodeMalloc in Section 4.2 and points to data required in the computation of $f(t, y)$. There is no return value for a RhsFn.

If preconditioning is used, then the user must provide a C function to solve the linear system $Pz = r$ where P may be either a left or a right preconditioner matrix. This C function must be of type CVSpgmrPSolveFn. The Psolve function has the following form:

```
int PSolve(integer N, real t, N_Vector y, N_Vector fy, N_Vector vtemp,
        real gamma, N_Vector ewt, real delta, long int *nfePtr,
        N_Vector r, int lr, void *P_data, N_Vector z)
```

Its input is N, the number of ODEs and the length of all vectors; t, the current value of the independent variable; y, the current value of the dependent variable vector; fy, the current vector $f(t, y)$; vtemp, a pointer to memory allocated as an N-vector workspace; and gamma, the current value of the scalar γ in the Newton matrix (5). Further input parameters are ewt, the error weight vector; delta, an input tolerance if Psolve is to use an iterative method; nfePtr, a pointer to the PVODE data nfe, the number of calls to the f routine; r, the right hand side vector in the linear system; lr, an input flag set to 1 to indicate a left preconditioner or 2 for a right preconditioner. P_data is the pointer to the user preconditioner data passed to CVSpgmr. The only output argument is z, the vector computed by Psolve. The integer returned value is to be negative if the Psolve function failed with an unrecoverable error, 0 if Psolve was successful, or positive if there was a recoverable error.

If the user's preconditioner requires that any Jacobian related data be evaluated or preprocessed, then this needs to be done in the optional user-supplied C function Precond. Most of the arguments for this function have been seen above. The Precond function has the form:

```
int Precond (integer N, real t, N_Vector y, N_Vector fy, boole jok,
        boole *jcurPtr, real gamma, N_Vector ewt, real h, real uround,
        long int *nfePtr, void *P_data,
        N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
```

The arguments which have not been discussed previously are the following. The input flag jok indicates whether or not Jacobian-related data needs to be recomputed. If jok == FALSE, then it is to be recomputed from scratch. If jok == TRUE, and Jacobian-related data was saved from the previous call to Precond, then the data can be reused with the current value of gamma. The parameter jcurPtr is a pointer to a boolean output flag to be set by Precond. Set *jcurPtr == TRUE if the Jacobian data was recomputed, and set *jcurPtr == FALSE if the Jacobian data was not recomputed and saved data was reused. The last three arguments are temporary N-vectors available for workspace. The current stepsize h and unit roundoff uround are supplied for possible use in difference quotient calculations.

4.4. Use by a C++ Application. PVODE is written in a manner that permits it to be used by applications written in C++ as well as in C. For this purpose, each PVODE header file is wrapped with conditionally compiled lines reading extern "C" { ... }, conditional on the variable __cplusplus being defined. This directive causes the C++ compiler to use C-style names when compiling the function prototypes encountered. Users with C++ applications should also be aware that we have defined, in llnltyps.h, a boolean variable type, boole, since C has no such type. The type boole is equated to type int, and so arguments in user calls, or calls to user-supplied routines, which are of type boole can be typed as either boole or int by the user. The same applies to vector kernels which have a type boole return value, if the user is providing these kernels.

5. A Band-Block-Diagonal Preconditioner Module. A principal reason for using a parallel ODE solver such as PVODE lies in the solution of partial differential equations (PDEs). Moreover, the use of a Krylov iterative method for the solution of many such problems is motivated by the nature of the underlying linear system of equations (4) that must be solved at each time step. The linear algebraic system is large, sparse, and structured. However, if a Krylov iterative method is to be effective in this setting, then a nontrivial preconditioner needs to be used. Otherwise, the rate of convergence of the Krylov iterative method is usually unacceptably slow. Unfortunately, an effective preconditioner tends to be problem-specific.

However, we have developed one type of preconditioner that treats a rather broad class of PDE-based problems. It has been successfully used for several realistic, large-scale problems [11] and is included in a software module within the PVODE package. This module generates a preconditioner that is a block-diagonal matrix with each block being a band matrix. The blocks need not have the same number of super- and sub-diagonals and these numbers may vary from block to block. This Band-Block-Diagonal Preconditioner module is called PVBBDPRE.

One way to envision these preconditioners is to think of the domain of the computational PDE problem as being 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 system. The basic idea is to isolate the preconditioning so that it is local to each processor, and also to use a (possibly cheaper) approximate right-hand side function. This requires the definition of a new function $g(t, y)$ which approximates the function $f(t, y)$ in the definition of the ODE system (1). However, the user may set $g = f$. Corresponding to the domain decomposition, there is a decomposition of the solution vector y into M disjoint blocks y_m, and a decomposition of g into 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 we have

$$g(t, y) = [g_1(t, \bar{y}_1), g_2(t, \bar{y}_2), \ldots, g_M(t, \bar{y}_M)]^T \tag{9}$$

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

The preconditioner associated with this decomposition has the form

$$P = diag[P_1, P_2, \ldots, P_M] \tag{10}$$

where

$$P_m \approx I - \gamma J_m \tag{11}$$

and J_m is a difference quotient approximation to $\partial g_m / \partial y_m$. This matrix is taken to be banded, with 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

$$Px = b \tag{12}$$

reduces to solving each of the equations

$$P_m x_m = b_m \tag{13}$$

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

To use this PVBBDPRE module, the user must supply two functions which the module calls to construct P. These are in addition to the user-supplied right-hand side function `f`.

- A function `gloc(Nlocal, t, ylocal, glocal, f_data)` must be supplied by the user to compute $g(t, y)$. It loads the real array `glocal` as a function of `t` and `ylocal`. Both `glocal` and `ylocal` are of length `Nlocal`, the local vector length.
- A function `cfn(Nlocal, t, y, f_data)` which must be supplied to perform all inter-processor communications necessary for the execution of the `gloc` function, using the input vector `y` of type `N_Vector`.

Both functions take as input the same pointer `f_data` as that passed by the user to `CVodeMalloc` and passed to the user's function `f`, and neither function has a return value. The user is responsible for providing space (presumably within `f_data`) for components of `y` that are communicated by `cfn` from the other processors, and that are then used by `gloc`, which is not expected to do any communication.

The user's calling program should include the following elements:

- `#include ''pvbbdpre.h''` for needed function prototypes and for type `PVBBDData`.
- `PVBBDData p_data;`
- `machEnv = PVecInitMPI(comm, Nlocal, N, argc, argv);`
- `N_VMake(y, ydata, machEnv;`
- `cvode_mem = CVodeMaloc(N, f, ...);`
- `p_data = PVBBDAlloc(Nlocal, mu, ml, gloc, cfn, f_data);` where the upper and lower half-bandwidths are `mu` and `ml`, respectively; `gloc` and `cfn` are names of user-supplied functions; and `f_data` is a pointer to private data.
- `CVSpgmr(cvode_mem, pretype, gstype, maxl, delt, PVBBDPrecon, PVBBDPSol, p_data);` with the memory pointers `cvode_mem` and `p_data` returned by the two previous calls, the four SPGMR parameters (`pretype, gstype, maxl, delt`) and the names of the preconditioner routines (`PVBBDPrecon, PVBBDPSol`) supplied with the PVBBDPRE module.
- `ier = CVode(cvode_mem, tout, y, &t, itask);` to carry out the integration to `t = tout`.
- `PVBBDFree(p_data);` to free the `PVBBDPRE` memory block.
- `CVodeFree(cvode_mem);` to free the CVode memory block.
- `PVecFreeMPI(machEnv);` to free the PVODE memory block.

Three optional outputs associated with this module are available by way of macros. These are:

- `PVBBD_RPWSIZE(p_data)` = size of the real workspace (local to the current processor) used by PVBBDPRE.
- `PVBBD_IPWSIZE(p_data)` = size of the integer workspace (local to the current processor) used by PVBBDPRE.
- `PVBBD_NGE(p_data)` = cumulative number of g evaluations (calls to `gloc`) so far.

The costs associated with PVBBDPRE also include `nsetups` LU factorizations, `nsetups` calls to `cfn`, and `nps` banded backsolve calls, where `nsetups` and `nps` are optional CVODE outputs.

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.

6. The Fortran/C Interface Package. We anticipate that many users of PVODE will work from existing Fortran application programs. To accommodate them, we have provided a set of interface routines that make the required connections to PVODE with a minimum of changes to the application programs. Specifically, a Fortran/C interface package called FPVODE 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. This package entails some compromises in portability, but we have kept these to a minimum by requiring fixed names for user-supplied routines, and by using a name-mapping scheme to set the names of externals in the Fortran/C linkages. The latter depends on two parameters, set in a small header file, which determine whether the Fortran external names are to be in upper case and whether they are to have an underscore character prefix.

The usage of this module is summarized below. Further details can be found in the header file `fpvode.h`. Also, the user should check, and reset if necessary, the paramaters in the file `fcmixpar.h`. The functions which are callable from the user's Fortran program are as follows:

- `FPVINITMPI` 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`.
- `FVFREEMPI` interfaces with `PVecFreeMPI` and is used to free memory allocated for MPI.

The user-supplied Fortran subroutines are as follows. The names of these 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.

The Fortran/C interfaces have been tested on a Cray-T3D and a cluster of Sun workstations.

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 with `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 `PVCOMMF`, which performs communications necessary for `PVLOCFN`.

7. Example Problems. Two test problems are described here. The first is a non-stiff problem which is included to demonstrate the capability of solving such problems and to show that PVODE can be applied with varying numbers of processors. The second problem is a stiff problem and illustrates the capability of solving that class of problems. Both problems involve the method of lines solution of a partial differential equation (PDE).

7.1. Example problem 1 - A nonstiff PDE problem. This problem begins with a prototypical diffusion-advection equation for $u = u(t, x)$

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + 0.5\frac{\partial u}{\partial x} \tag{14}$$

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

$$\begin{aligned} u(t,0) &= 0 \\ u(t,2) &= 0 \\ u(0,x) &= x(2-x)\exp(2x) \end{aligned} \tag{15}$$

A system of MX ODEs is obtained by discretizing the x-axis with $MX + 2$ grid points and replacing the first and second order spatial derivatives with their central difference approximations. Since the value of u is constant at the two endpoints, the semi-discrete equations for those points can be eliminated. The resulting system of ODEs can now be written with u_i the approximation to $u(t, x_i)$, $x_i = i(\Delta x)$, and $\Delta x = 2/(MX + 1)$:

$$\dot{u}_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2} + 0.5\frac{u_{i+1} - u_{i-1}}{2(\Delta x)} \tag{16}$$

The above equation holds for $i = 1, 2, \ldots, MX$ with the understanding that $u_0 = u_{MX+1} = 0$.

In the parallel processing environment, we may think of the several processors as being laid out on a straight line with each processor to compute its contiguous subset of the solution vector. Consequently the computation of the right hand side of (16) requires that each interior processor must pass the first component of its block of the solution vector to its left-hand neighbor, acquire the last component of that neighbor's block, pass the last component of its block of the solution vector to its right-hand neighbor, and acquire the first component of that neighbor's block. If the processor is the first (0th) or last processor, then communication to the left or right (respectively) is not required.

The file `pvnx.c` is included in the PVODE package and is the code for this problem. It uses the Adams (non-stiff) integration formula and functional iteration. The intent of this problem is to illustrate the basic user-supplied code and to show that for a fixed problem size the number of processors can be varied. As it stands, it is an unrealistically small, simple problem. Using more than one processor simply demonstrates that this can be done. The output shown below is for 10 grid points and four processors. Varying the number of processors will alter the output, only because of roundoff-level differences in various vector operations.

```
 1-D advection-diffusion equation, mesh size = 10

 Number of PEs =   4

At t = 0.00    max.norm(u) =  1.569909e+01
At t = 0.50    max.norm(u) =  3.052881e+00   nst = 113
At t = 1.00    max.norm(u) =  8.753188e-01   nst = 191
At t = 1.50    max.norm(u) =  2.494926e-01   nst = 265
At t = 2.00    max.norm(u) =  7.109674e-02   nst = 333
At t = 2.50    max.norm(u) =  2.026039e-02   nst = 404
At t = 3.00    max.norm(u) =  5.772786e-03   nst = 490
At t = 3.50    max.norm(u) =  1.644895e-03   nst = 608
At t = 4.00    max.norm(u) =  4.690811e-04   nst = 727
At t = 4.50    max.norm(u) =  1.343719e-04   nst = 801
At t = 5.00    max.norm(u) =  3.852882e-05   nst = 878

Final Statistics..

nst = 878     nfe  = 1358    nni = 0       ncfn = 90      netf = 5
```

7.2. Example problem 2 - A stiff PDE system. This test problem is based on a two-dimensional system of two PDEs involving diurnal kinetics, advection, and diffusion. The PDEs can be written as

$$\frac{\partial c^i}{\partial t} = K_h \frac{\partial^2 c^i}{\partial x^2} + V \frac{\partial c^i}{\partial x} + \frac{\partial}{\partial y} K_v(y) \frac{\partial c^i}{\partial y} + R^i(c^1, c^2, t) \quad (i = 1, 2), \tag{17}$$

where the superscripts i are used to distinguish the chemical species, and where the reaction terms are given by

$$\begin{aligned} R^1(c^1, c^2, t) &= -q_1 c^1 c^3 - q_2 c^1 c^2 + 2q_3(t) c^3 + q_4(t) c^2 \\ R^2(c^1, c^2, t) &= q_1 c^1 c^3 - q_2 c^1 c^2 - q_4(t) c^2 \end{aligned} \tag{18}$$

The spatial domain is $0 \leq x \leq 20,\ 30 \leq y \leq 50$. The constants and parameters for this problem are as follows: $K_h = 4.0 \times 10^{-6}$, $V = 10^{-3}$, $K_v = 10^{-8} \exp(y/5)$, $q_1 =$

1.63×10^{-16}, $q_2 = 4.66 \times 10^{-16}$, $c^3 = 3.7 \times 10^{16}$, and the diurnal rate constants are defined as follows:

$$\begin{aligned} q_i(t) &= \exp[-a_i/\sin\omega t], \quad \text{for } \sin\omega t > 0 \\ q_i(t) &= 0, \quad \text{for } \sin\omega t \le 0 \end{aligned}$$

where $i = 3, 4$, $\omega = \pi/43200$, $a_3 = 22.62$, $a_4 = 7.601$. The time interval of integration is $[0, 86400]$, representing 24 hours measured in seconds.

Homogeneous Neumann boundary conditions are imposed on each boundary and the initial conditions are

$$\begin{aligned} c^1(x,z,0) &= 10^6\alpha(x)\beta(y), \; c^2(x,z,0) = 10^{12}\alpha(x)\beta(y) \\ \alpha(x) &= 1 - (0.1x-1)^2 + (0.1x-1)^4/2 \\ \beta(y) &= 1 - (0.1y-4)^2 + (0.1y-4)^4/2 \end{aligned} \tag{19}$$

These equations represent a simplified model for the transport, production, and loss of the oxygen singlet and ozone in the upper atmosphere.

As before, we discretize the PDE system with central differencing, to obtain an ODE system $\dot{u} = f(t,u)$ representing (17). For this example, we may think of the processors as being laid out in a rectangle, and each processor being assigned a subgrid of size $MXSUB \times MYSUB$ of the $x-y$ grid. If there are $NPEX$ processors in the x direction and $NPEY$ processors in the y direction then the overall grid size is $MX \times MY$ with $MX = NPEX \times MXSUB$ and $MY = NPEY \times MYSUB$. There are $2 \times MX \times MY$ equations in this system of ODEs. To compute f in this setting, the processors pass and receive information as follows. The solution components for the bottom row of grid points in the current processor are passed to the processor below it and the solution for the top row of grid points is received from the processor below the current processor. The solution for the top row of grid points for the current processor is sent to the processor above the current processor, while the solution for the bottom row of grid points is received from that processor by the current processor. Similarly the solution for the first column of grid points is sent from the current processor to the processor to its left and the last column of grid points is received from that processor by the current processor. The communication for the solution at the right edge of the processor is similar. If this is the last processor in a particular direction, then message passing and receiving are bypassed for that direction.

The code listing for this example is given in the Appendix, while the code itself is in the file `pvkx.c` in the PVODE package. The purpose of this code is to provide a more complicated example than Example 1, 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. The left preconditioner is the block-diagonal part of the Newton matrix, with 2×2 blocks, and 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.

Sample output from `pvkx.c` follows. Again, the output will vary if the number of processors is changed. The output is for four processors (in a 2×2 array) with a 5×5 subgrid on each processor.

```
2-species diurnal advection-diffusion problem

t = 7.20e+03   no. steps = 219   order = 5   stepsize = 1.59e+02
At bottom left:  c1, c2 =    1.047e+04    2.527e+11
At top right:    c1, c2 =    1.119e+04    2.700e+11

t = 1.44e+04   no. steps = 251   order = 5   stepsize = 3.77e+02
At bottom left:  c1, c2 =    6.659e+06    2.582e+11
At top right:    c1, c2 =    7.301e+06    2.833e+11

t = 2.16e+04   no. steps = 277   order = 5   stepsize = 2.75e+02
At bottom left:  c1, c2 =    2.665e+07    2.993e+11
At top right:    c1, c2 =    2.931e+07    3.313e+11

t = 2.88e+04   no. steps = 301   order = 5   stepsize = 2.23e+02
At bottom left:  c1, c2 =    8.702e+06    3.380e+11
At top right:    c1, c2 =    9.650e+06    3.751e+11

t = 3.60e+04   no. steps = 347   order = 4   stepsize = 4.37e+01
At bottom left:  c1, c2 =    1.404e+04    3.387e+11
At top right:    c1, c2 =    1.561e+04    3.765e+11

t = 4.32e+04   no. steps = 411   order = 4   stepsize = 4.64e+02
At bottom left:  c1, c2 =    1.001e-08    3.382e+11
At top right:    c1, c2 =    8.489e-08    3.804e+11

t = 5.04e+04   no. steps = 430   order = 4   stepsize = 2.82e+02
At bottom left:  c1, c2 =    1.592e-08    3.358e+11
At top right:    c1, c2 =    2.259e-08    3.864e+11

t = 5.76e+04   no. steps = 444   order = 5   stepsize = 4.60e+02
At bottom left:  c1, c2 =    1.257e-10    3.320e+11
At top right:    c1, c2 =    1.766e-10    3.909e+11
```

```
t = 6.48e+04   no. steps = 456   order = 5   stepsize = 6.97e+02
At bottom left:  c1, c2 =    6.114e-12    3.313e+11
At top right:    c1, c2 =    9.739e-12    3.963e+11

t = 7.20e+04   no. steps = 467   order = 5   stepsize = 6.97e+02
At bottom left:  c1, c2 =    7.140e-12    3.330e+11
At top right:    c1, c2 =    1.010e-11    4.039e+11

t = 7.92e+04   no. steps = 477   order = 5   stepsize = 6.97e+02
At bottom left:  c1, c2 =   -2.748e-13    3.334e+11
At top right:    c1, c2 =   -3.909e-13    4.120e+11

t = 8.64e+04   no. steps = 487   order = 5   stepsize = 6.97e+02
At bottom left:  c1, c2 =   -2.804e-15    3.352e+11
At top right:    c1, c2 =   -3.875e-15    4.163e+11


Final Statistics..

lenrw   =  2000    leniw =     0
llrw    =  2046    lliw  =     0
nst     =   487    nfe   =  1278
nni     =   636    nli   =   639
nsetups =    84    netf  =    32
npe     =     8    nps   =  1213
ncfn    =     0    ncfl  =     0
```

A third example is provided with the PVODE package, in the file `pvkxb.c`. It uses the same ODE system as in the above stiff example, but a slightly different solution method. It uses the `PVBBDPRE` preconditioner module to generate a band-block-diagonal preconditioner, using half-bandwidths equal to 2.

8. Testing. The stiff example problem described in Section 7.2 has been modified and expanded to form a test problem for PVODE. This work was largely carried out by M. Wittman and reported in [10].

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, the function $\beta(y)$ in Eq. (19) has been replaced by:

$$\beta(y) = .75 + .25 \tanh(10y - 400) \tag{20}$$

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 differenci ng 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 $\partial c^i/\partial x$, namely:

$$\partial c/\partial x|_{x_j} \approx \left[\frac{3}{2}c_{j+1} - c_j - \frac{1}{2}c_{j-1}\right]/(2\Delta x) \tag{21}$$

With this modified form of the problem, we performed tests similar to those 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 `NPEX` = 8 (for a vertical mesh size of `MY` = 400), and set `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 [7]
- EPCC: an implemenation of MPI by the Edinburgh Parallel Computer Centre [4]
- 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 1
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.

9. Availability. At present, the PVODE package has not been released for general distribution. However, plans are in progress for a release that is limited to non-commercial use of the package. Interested potential users should contact Alan Hindmarsh, alanh@llnl.gov. The CVODE package, however, on which PVODE is based, is freely available from the Netlib collection. See for example the listing `cvode.tar.qz` at the web site

`http://www.netlib.org/ode/index.html`

The Netlib version of the CVODE package includes the CVODE User Guide [5] in the form of a PostScript file.

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), 1038-1051.

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

[3] 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.

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

[5] Scott D. Cohen and Alan C. Hindmarsh, *CVODE User Guide*, Lawrence Livermore National Laboratory report UCRL-MA-118618, Sept. 1994.

[6] 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.

[7] 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.

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

[9] S. Balay, W. Gropp, L. McInnes, and B. Smith, *PETSc 2.0 Users Manual*, Argonne National Laboratory, 1996.

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

[11] Alan C. Hindmarsh and Allan G. Taylor, *PVODE and KINSOL: Parallel Software for Differential and Nonlinear Systems*, Lawrence Livermore National Laboratory report UCRL-ID-129739, February 1998.

10. Appendix: Listing of Stiff Example Program.

```
/************************************************************************
 *                                                                      *
 * File: pvkx.c                                                         *
 * Programmers: S. D. Cohen, A. C. Hindmarsh, M. R. Wittman @ LLNL      *
 * Version of 14 May 1998                                               *
 *----------------------------------------------------------------------*
 * Example problem.                                                     *
 * An ODE system is generated from the following 2-species diurnal      *
 * kinetics advection-diffusion PDE system in 2 space dimensions:       *
 *                                                                      *
 * dc(i)/dt = Kh*(d/dx)^2 c(i) + V*dc(i)/dx + (d/dy)(Kv(y)*dc(i)/dy)    *
 *                 + Ri(c1,c2,t)      for i = 1,2,   where              *
 *   R1(c1,c2,t) = -q1*c1*c3 - q2*c1*c2 + 2*q3(t)*c3 + q4(t)*c2 ,       *
 *   R2(c1,c2,t) =  q1*c1*c3 - q2*c1*c2 - q4(t)*c2 ,                    *
 *   Kv(y) = Kv0*exp(y/5) ,                                             *
 * Kh, V, Kv0, q1, q2, and c3 are constants, and q3(t) and q4(t)        *
 * vary diurnally.   The problem is posed on the square                 *
 *   0 <= x <= 20,    30 <= y <= 50   (all in km),                      *
 * with homogeneous Neumann boundary conditions, and for time t in      *
 *   0 <= t <= 86400 sec (1 day).                                       *
 * The PDE system is treated by central differences on a uniform        *
 * mesh, with simple polynomial initial profiles.                       *
 *                                                                      *
 * The problem is solved by PVODE on NPE processors, treated as a       *
 * rectangular process grid of size NPEX by NPEY, with NPE = NPEX*NPEY. *
 * Each processor contains a subgrid of size MXSUB by MYSUB of the      *
 * (x,y) mesh.  Thus the actual mesh sizes are MX = MXSUB*NPEX and      *
 * MY = MYSUB*NPEY, and the ODE system size is neq = 2*MX*MY.           *
 *                                                                      *
 * The solution with PVODE is done with the BDF/GMRES method (i.e.      *
 * using the CVSPGMR linear solver) and the block-diagonal part of the  *
 * Newton matrix as a left preconditioner. A copy of the block-diagonal *
 * part of the Jacobian is saved and conditionally reused within the    *
 * Precond routine.                                                     *
 *                                                                      *
 * Performance data and sampled solution values are printed at selected *
 * output times, and all performance counters are printed on completion.*
 *                                                                      *
 * This version uses MPI for user routines, and the MPI_PVODE solver.   *
 * Execution: pvkx -npes N  with N = NPEX*NPEY (see constants below).   *
 ************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "llnltyps.h"  /* definitions of real, integer, boole, TRUE,FALSE */
```

```
#include "cvode.h"     /* main CVODE header file                              */
#include "iterativ.h"  /* contains the enum for types of preconditioning  */
#include "cvspgmr.h"   /* use CVSPGMR linear solver each internal step    */
#include "smalldense.h" /* use generic DENSE solver in preconditioning    */
#include "nvector.h"   /* definitions of type N_Vector, macro N_VDATA     */
#include "llnlmath.h"  /* contains SQR macro                              */
#include "mpi.h"


/* Problem Constants */

#define NVARS        2             /* number of species         */
#define KH           4.0e-6        /* horizontal diffusivity Kh */
#define VEL          0.001         /* advection velocity V      */
#define KV0          1.0e-8        /* coefficient in Kv(y)      */
#define Q1           1.63e-16      /* coefficients q1, q2, c3   */
#define Q2           4.66e-16
#define C3           3.7e16
#define A3           22.62         /* coefficient in expression for q3(t) */
#define A4           7.601         /* coefficient in expression for q4(t) */
#define C1_SCALE     1.0e6         /* coefficients in initial profiles    */
#define C2_SCALE     1.0e12

#define T0           0.0           /* initial time */
#define NOUT         12            /* number of output times */
#define TWOHR        7200.0        /* number of seconds in two hours  */
#define HALFDAY      4.32e4        /* number of seconds in a half day */
#define PI       3.1415926535898   /* pi */

#define XMIN          0.0          /* grid boundaries in x  */
#define XMAX         20.0
#define YMIN         30.0          /* grid boundaries in y  */
#define YMAX         50.0

#define NPEX         2              /* no. PEs in x direction of PE array */
#define NPEY         2              /* no. PEs in y direction of PE array */
                                    /* Total no. PEs = NPEX*NPEY */
#define MXSUB        5              /* no. x points per subgrid */
#define MYSUB        5              /* no. y points per subgrid */

#define MX           (NPEX*MXSUB)   /* MX = number of x mesh points */
#define MY           (NPEY*MYSUB)   /* MY = number of y mesh points */
                                    /* Spatial mesh is MX by MY */

/* CVodeMalloc Constants */

#define RTOL    1.0e-5            /* scalar relative tolerance */
#define FLOOR   100.0             /* value of C1 or C2 at which tolerances */
```

```
                                  /* change from relative to absolute      */
#define ATOL    (RTOL*FLOOR)      /* scalar absolute tolerance */


/* User-defined matrix accessor macro: IJth */

/* IJth is defined in order to write code which indexes into small dense
   matrices with a (row,column) pair, where 1 <= row,column <= NVARS.

   IJth(a,i,j) references the (i,j)th entry of the small matrix real **a,
   where 1 <= i,j <= NVARS. The small matrix routines in dense.h
   work with matrices stored by column in a 2-dimensional array. In C,
   arrays are indexed starting at 0, not 1. */

#define IJth(a,i,j)        (a[j-1][i-1])


/* Type : UserData
   contains problem constants, preconditioner blocks, pivot arrays,
   grid constants, and processor indices */

typedef struct {
  real q4, om, dx, dy, hdco, haco, vdco;
  real uext[NVARS*(MXSUB+2)*(MYSUB+2)];
  integer my_pe, isubx, isuby, nvmxsub, nvmxsub2;
  MPI_Comm comm;
} *UserData;

typedef struct {
  void *f_data;
  real **P[MXSUB][MYSUB], **Jbd[MXSUB][MYSUB];
  integer *pivot[MXSUB][MYSUB];
} *PreconData;


/* Private Helper Functions */

static PreconData AllocPreconData(UserData data);
static void InitUserData(integer my_pe, MPI_Comm comm, UserData data);
static void FreePreconData(PreconData pdata);
static void SetInitialProfiles(N_Vector u, UserData data);
static void PrintOutput(integer my_pe, MPI_Comm comm, long int iopt[],
                        real ropt[], N_Vector u, real t);
static void PrintFinalStats(long int iopt[]);
static void BSend(MPI_Comm comm, integer my_pe, integer isubx, integer isuby,
                  integer dsizex, integer dsizey, real udata[]);
static void BRecvPost(MPI_Comm comm, MPI_Request request[], integer my_pe,
      integer isubx, integer isuby,
```

```
      integer dsizex, integer dsizey,
      real uext[], real buffer[]);
static void BRecvWait(MPI_Request request[], integer isubx, integer isuby,
      integer dsizex, real uext[], real buffer[]);
static void ucomm(integer N, real t, N_Vector u, UserData data);
static void fcalc(integer N, real t, real udata[], real dudata[], UserData data);

/* Functions Called by the CVODE Solver */

static void f(integer N, real t, N_Vector u, N_Vector udot, void *f_data);

static int Precond(integer N, real tn, N_Vector u, N_Vector fu, boole jok,
                   boole *jcurPtr, real gamma, N_Vector ewt, real h,
                   real uround, long int *nfePtr, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);

static int PSolve(integer N, real tn, N_Vector u, N_Vector fu, N_Vector vtemp,
                  real gamma, N_Vector ewt, real delta, long int *nfePtr,
                  N_Vector r, int lr, void *P_data, N_Vector z);


/***************************** Main Program ******************************/

main(int argc, char *argv[])
{
  real abstol, reltol, t, tout, ropt[OPT_SIZE];
  long int iopt[OPT_SIZE];
  N_Vector u;
  UserData data;
  PreconData predata;
  void *cvode_mem;
  int iout, flag;
  integer neq, local_N, my_pe, npes;
  machEnvType machEnv;
  MPI_Comm comm;

  /* Set problem size neq */

  neq = NVARS*MX*MY;

  /* Get processor number and total number of pe's */

  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &npes);
  MPI_Comm_rank(comm, &my_pe);

  if (npes != NPEX*NPEY) {
```

```
    if (my_pe == 0)
      printf("\n npes=%d is not equal to NPEX*NPEY=%d\n", npes,NPEX*NPEY);
    return(1);
  }

  /* Set local length */

  local_N = NVARS*MXSUB*MYSUB;

  /* Allocate and load user data block; allocate preconditioner block */

  data = (UserData) malloc(sizeof *data);
  InitUserData(my_pe, comm, data);
  predata = AllocPreconData (data);

  /* Set machEnv block */

  machEnv = PVecInitMPI(comm, local_N, neq, &argc, &argv);
  if (machEnv == NULL) return(1);

  /* Allocate u, and set initial values and tolerances */

  u = N_VNew(neq, machEnv);
  SetInitialProfiles(u, data);
  abstol = ATOL; reltol = RTOL;

/* Call CVodeMalloc to initialize CVODE:

     neq     is the problem size = number of equations
     f       is the user's right hand side function in u'=f(t,u)
     T0      is the initial time
     u       is the initial dependent variable vector
     BDF     specifies the Backward Differentiation Formula
     NEWTON  specifies a Newton iteration
     SS      specifies scalar relative and absolute tolerances
     &reltol and &abstol are pointers to the scalar tolerances
     data    is the pointer to the user-defined block of coefficients
     FALSE   indicates there are no optional inputs in iopt and ropt
     iopt    and ropt arrays communicate optional integer and real input/output

     A pointer to CVODE problem memory is returned and stored in cvode_mem.  */

  cvode_mem = CVodeMalloc(neq, f, T0, u, BDF, NEWTON, SS, &reltol,
                          &abstol, data, NULL, FALSE, iopt, ropt, machEnv);
  if (cvode_mem == NULL) { printf("CVodeMalloc failed."); return(1); }

  /* Call CVSpgmr to specify the CVODE linear solver CVSPGMR with
     left preconditioning, modified Gram-Schmidt orthogonalization,
```

```
     default values for the maximum Krylov dimension maxl and the tolerance
     parameter delt, preconditioner setup and solve routines Precond and
     PSolve, and the pointer to the preconditioner data block.             */

  CVSpgmr(cvode_mem, LEFT, MODIFIED_GS, 0, 0.0, Precond, PSolve, predata);

  if (my_pe == 0)
    printf("\n2-species diurnal advection-diffusion problem\n\n");

  /* In loop over output points, call CVode, print results, test for error */

  for (iout=1, tout = TWOHR; iout <= NOUT; iout++, tout += TWOHR) {
    flag = CVode(cvode_mem, tout, u, &t, NORMAL);
    PrintOutput(my_pe, comm, iopt, ropt, u, t);
    if (flag != SUCCESS) {
      if (my_pe == 0) printf("CVode failed, flag=%d.\n", flag);
      break;
    }
  }

  /* Free memory and print final statistics */

  N_VFree(u);
  FreePreconData(predata);
  CVodeFree(cvode_mem);
  if (my_pe == 0) PrintFinalStats(iopt);
  PVecFreeMPI(machEnv);
  MPI_Finalize();

  return(0);
}


/*********************** Private Helper Functions ************************/

/* Allocate memory for data structure of type UserData */

static PreconData AllocPreconData(UserData fdata)
{
  int lx, ly;
  PreconData pdata;

  pdata = (PreconData) malloc(sizeof *pdata);

  pdata->f_data = fdata;

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
```

```
      (pdata->P)[lx][ly] = denalloc(NVARS);
      (pdata->Jbd)[lx][ly] = denalloc(NVARS);
      (pdata->pivot)[lx][ly] = denallocpiv(NVARS);
    }
  }

  return(pdata);
}

/* Load constants in data */

static void InitUserData(integer my_pe, MPI_Comm comm, UserData data)
{
  integer isubx, isuby;

  /* Set problem constants */
  data->om = PI/HALFDAY;
  data->dx = (XMAX-XMIN)/((real)(MX-1));
  data->dy = (YMAX-YMIN)/((real)(MY-1));
  data->hdco = KH/SQR(data->dx);
  data->haco = VEL/(2.0*data->dx);
  data->vdco = (1.0/SQR(data->dy))*KV0;

  /* Set machine-related constants */
  data->comm = comm;
  data->my_pe = my_pe;
  /* isubx and isuby are the PE grid indices corresponding to my_pe */
  isuby = my_pe/NPEX;
  isubx = my_pe - isuby*NPEX;
  data->isubx = isubx;
  data->isuby = isuby;
  /* Set the sizes of a boundary x-line in u and uext */
  data->nvmxsub = NVARS*MXSUB;
  data->nvmxsub2 = NVARS*(MXSUB+2);

}

/* Free data memory */

static void FreePreconData(PreconData pdata)
{
  int lx, ly;

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      denfree((pdata->P)[lx][ly]);
      denfree((pdata->Jbd)[lx][ly]);
      denfreepiv((pdata->pivot)[lx][ly]);
```

```
    }
  }

  free(pdata);
}

/* Set initial conditions in u */

static void SetInitialProfiles(N_Vector u, UserData data)
{
  integer isubx, isuby, lx, ly, jx, jy, offset;
  real dx, dy, x, y, cx, cy, xmid, ymid;
  real *udata;

  /* Set pointer to data array in vector u */

  udata = N_VDATA(u);

  /* Get mesh spacings, and subgrid indices for this PE */

  dx = data->dx;         dy = data->dy;
  isubx = data->isubx;   isuby = data->isuby;

  /* Load initial profiles of c1 and c2 into local u vector.
  Here lx and ly are local mesh point indices on the local subgrid,
  and jx and jy are the global mesh point indices. */

  offset = 0;
  xmid = .5*(XMIN + XMAX);
  ymid = .5*(YMIN + YMAX);
  for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    y = YMIN + jy*dy;
    cy = SQR(0.1*(y - ymid));
    cy = 1.0 - cy + 0.5*SQR(cy);
    for (lx = 0; lx < MXSUB; lx++) {
      jx = lx + isubx*MXSUB;
      x = XMIN + jx*dx;
      cx = SQR(0.1*(x - xmid));
      cx = 1.0 - cx + 0.5*SQR(cx);
      udata[offset  ] = C1_SCALE*cx*cy;
      udata[offset+1] = C2_SCALE*cx*cy;
      offset = offset + 2;
    }
  }
}

/* Print current t, step count, order, stepsize, and sampled c1,c2 values */
```

```
static void PrintOutput(integer my_pe, MPI_Comm comm, long int iopt[],
                        real ropt[], N_Vector u, real t)
{
  real *udata, tempu[2];
  integer npelast, i0, i1;
  MPI_Status status;

  npelast = NPEX*NPEY - 1;
  udata = N_VDATA(u);

  /* Send c1,c2 at top right mesh point to PE 0 */
  if (my_pe == npelast) {
    i0 = NVARS*MXSUB*MYSUB - 2;
    i1 = i0 + 1;
    if (npelast != 0)
      MPI_Send(&udata[i0], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else {
      tempu[0] = udata[i0];
      tempu[1] = udata[i1];
    }
  }

  /* On PE 0, receive c1,c2 at top right, then print performance data
     and sampled solution values */
  if (my_pe == 0) {
    if (npelast != 0)
      MPI_Recv(&tempu[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);
    printf("t = %.2e   no. steps = %d   order = %d   stepsize = %.2e\n",
           t, iopt[NST], iopt[QU], ropt[HU]);
    printf("At bottom left:  c1, c2 = %12.3e %12.3e \n", udata[0], udata[1]);
    printf("At top right:    c1, c2 = %12.3e %12.3e \n\n", tempu[0], tempu[1]);
  }
}

/* Print final statistics contained in iopt */

static void PrintFinalStats(long int iopt[])
{
  printf("\nFinal Statistics.. \n\n");
  printf("lenrw   = %5ld    leniw = %5ld\n", iopt[LENRW], iopt[LENIW]);
  printf("llrw    = %5ld    lliw  = %5ld\n", iopt[SPGMR_LRW], iopt[SPGMR_LIW]);
  printf("nst     = %5ld    nfe   = %5ld\n", iopt[NST], iopt[NFE]);
  printf("nni     = %5ld    nli   = %5ld\n", iopt[NNI], iopt[SPGMR_NLI]);
  printf("nsetups = %5ld    netf  = %5ld\n", iopt[NSETUPS], iopt[NETF]);
  printf("npe     = %5ld    nps   = %5ld\n", iopt[SPGMR_NPE], iopt[SPGMR_NPS]);
  printf("ncfn    = %5ld    ncfl  = %5ld\n \n", iopt[NCFN], iopt[SPGMR_NCFL]);
}
```

```
/* Routine to send boundary data to neighboring PEs */

static void BSend(MPI_Comm comm, integer my_pe, integer isubx, integer isuby,
                  integer dsizex, integer dsizey, real udata[])
{
  int i, ly;
  integer offsetu, offsetbuf;
  real bufleft[NVARS*MYSUB], bufright[NVARS*MYSUB];

  /* If isuby > 0, send data from bottom x-line of u */

  if (isuby != 0)
    MPI_Send(&udata[0], dsizex, PVEC_REAL_MPI_TYPE, my_pe-NPEX, 0, comm);

  /* If isuby < NPEY-1, send data from top x-line of u */

  if (isuby != NPEY-1) {
    offsetu = (MYSUB-1)*dsizex;
    MPI_Send(&udata[offsetu], dsizex, PVEC_REAL_MPI_TYPE, my_pe+NPEX, 0, comm);
  }

  /* If isubx > 0, send data from left y-line of u (via bufleft) */

  if (isubx != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = ly*dsizex;
      for (i = 0; i < NVARS; i++)
        bufleft[offsetbuf+i] = udata[offsetu+i];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
  }

  /* If isubx < NPEX-1, send data from right y-line of u (via bufright) */

  if (isubx != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetu = offsetbuf*MXSUB + (MXSUB-1)*NVARS;
      for (i = 0; i < NVARS; i++)
        bufright[offsetbuf+i] = udata[offsetu+i];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
  }

}
```

```
/* Routine to start receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB real entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */

static void BRecvPost(MPI_Comm comm, MPI_Request request[], integer my_pe,
      integer isubx, integer isuby,
      integer dsizex, integer dsizey,
      real uext[], real buffer[])
{
  integer offsetue;
  /* Have bufleft and bufright use the same buffer */
  real *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;

  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Irecv(&uext[NVARS], dsizex, PVEC_REAL_MPI_TYPE,
      my_pe-NPEX, 0, comm, &request[0]);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1) {
    offsetue = NVARS*(1 + (MYSUB+1)*(MXSUB+2));
    MPI_Irecv(&uext[offsetue], dsizex, PVEC_REAL_MPI_TYPE,
                                         my_pe+NPEX, 0, comm, &request[1]);
  }

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Irecv(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE,
                                         my_pe-1, 0, comm, &request[2]);
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Irecv(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE,
                                         my_pe+1, 0, comm, &request[3]);
  }

}

/* Routine to finish receiving boundary data from neighboring PEs.
   Notes:
   1) buffer should be able to hold 2*NVARS*MYSUB real entries, should be
   passed to both the BRecvPost and BRecvWait functions, and should not
   be manipulated between the two calls.
   2) request should have 4 entries, and should be passed in both calls also. */
```

```
static void BRecvWait(MPI_Request request[], integer isubx, integer isuby,
      integer dsizex, real uext[], real buffer[])
{
  int i, ly;
  integer dsizex2, offsetue, offsetbuf;
  real *bufleft = buffer, *bufright = buffer+NVARS*MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2*NVARS;

  /* If isuby > 0, receive data for bottom x-line of uext */
  if (isuby != 0)
    MPI_Wait(&request[0],&status);

  /* If isuby < NPEY-1, receive data for top x-line of uext */
  if (isuby != NPEY-1)
    MPI_Wait(&request[1],&status);

  /* If isubx > 0, receive data for left y-line of uext (via bufleft) */
  if (isubx != 0) {
    MPI_Wait(&request[2],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+1)*dsizex2;
      for (i = 0; i < NVARS; i++)
        uext[offsetue+i] = bufleft[offsetbuf+i];
    }
  }

  /* If isubx < NPEX-1, receive data for right y-line of uext (via bufright) */
  if (isubx != NPEX-1) {
    MPI_Wait(&request[3],&status);

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NVARS;
      offsetue = (ly+2)*dsizex2 - NVARS;
      for (i = 0; i < NVARS; i++)
uext[offsetue+i] = bufright[offsetbuf+i];
    }
  }

}

/* ucomm routine.  This routine performs all communication
```

```
   between processors of data needed to calculate f. */

static void ucomm(integer N, real t, N_Vector u, UserData data)
{

  real *udata, *uext, buffer[2*NVARS*MYSUB];
  MPI_Comm comm;
  integer my_pe, isubx, isuby, nvmxsub, nvmysub;
  MPI_Request request[4];

  udata = N_VDATA(u);


  /* Get comm, my_pe, subgrid indices, data sizes, extended array uext */

  comm = data->comm;  my_pe = data->my_pe;
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub;
  nvmysub = NVARS*MYSUB;
  uext = data->uext;

  /* Start receiving boundary data from neighboring PEs */

  BRecvPost(comm, request, my_pe, isubx, isuby, nvmxsub, nvmysub, uext, buffer);

  /* Send data from boundary of local grid to neighboring PEs */

  BSend(comm, my_pe, isubx, isuby, nvmxsub, nvmysub, udata);

  /* Finish receiving boundary data from neighboring PEs */

  BRecvWait(request, isubx, isuby, nvmxsub, uext, buffer);



}

/* fcalc routine. Compute f(t,y).  This routine assumes that communication
   between processors of data needed to calculate f has already been done,
   and this data is in the work array uext. */

static void fcalc(integer N, real t, real udata[], real dudata[], UserData data)
{
  real *uext;
  real q3, c1, c2, c1dn, c2dn, c1up, c2up, c1lt, c2lt;
  real c1rt, c2rt, cydn, cyup, hord1, hord2, horad1, horad2;
  real qq1, qq2, qq3, qq4, rkin1, rkin2, s, vertd1, vertd2, ydn, yup;
  real q4coef, dely, verdco, hordco, horaco;
```

```
int i, lx, ly, jx, jy;
integer isubx, isuby, nvmxsub, nvmxsub2, offsetu, offsetue;

/* Get subgrid indices, data sizes, extended work array uext */

isubx = data->isubx;   isuby = data->isuby;
nvmxsub = data->nvmxsub; nvmxsub2 = data->nvmxsub2;
uext = data->uext;

/* Copy local segment of u vector into the working extended array uext */

offsetu = 0;
offsetue = nvmxsub2 + NVARS;
for (ly = 0; ly < MYSUB; ly++) {
  for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
  offsetu = offsetu + nvmxsub;
  offsetue = offsetue + nvmxsub2;
}

/* To facilitate homogeneous Neumann boundary conditions, when this is
a boundary PE, copy data from the first interior mesh line of u to uext */

/* If isuby = 0, copy x-line 2 of u to uext */
if (isuby == 0) {
  for (i = 0; i < nvmxsub; i++) uext[NVARS+i] = udata[nvmxsub+i];
}

/* If isuby = NPEY-1, copy x-line MYSUB-1 of u to uext */
if (isuby == NPEY-1) {
  offsetu = (MYSUB-2)*nvmxsub;
  offsetue = (MYSUB+1)*nvmxsub2 + NVARS;
  for (i = 0; i < nvmxsub; i++) uext[offsetue+i] = udata[offsetu+i];
}

/* If isubx = 0, copy y-line 2 of u to uext */
if (isubx == 0) {
  for (ly = 0; ly < MYSUB; ly++) {
    offsetu = ly*nvmxsub + NVARS;
    offsetue = (ly+1)*nvmxsub2;
    for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
  }
}

/* If isubx = NPEX-1, copy y-line MXSUB-1 of u to uext */
if (isubx == NPEX-1) {
  for (ly = 0; ly < MYSUB; ly++) {
    offsetu = (ly+1)*nvmxsub - 2*NVARS;
    offsetue = (ly+2)*nvmxsub2 - NVARS;
```

```
    for (i = 0; i < NVARS; i++) uext[offsetue+i] = udata[offsetu+i];
  }
}

/* Make local copies of problem variables, for efficiency */

dely = data->dy;
verdco = data->vdco;
hordco  = data->hdco;
horaco  = data->haco;

/* Set diurnal rate coefficients as functions of t, and save q4 in
data block for use by preconditioner evaluation routine */

s = sin((data->om)*t);
if (s > 0.0) {
  q3 = exp(-A3/s);
  q4coef = exp(-A4/s);
} else {
  q3 = 0.0;
  q4coef = 0.0;
}
data->q4 = q4coef;


/* Loop over all grid points in local subgrid */

for (ly = 0; ly < MYSUB; ly++) {

  jy = ly + isuby*MYSUB;

  /* Set vertical diffusion coefficients at jy +- 1/2 */

  ydn = YMIN + (jy - .5)*dely;
  yup = ydn + dely;
  cydn = verdco*exp(0.2*ydn);
  cyup = verdco*exp(0.2*yup);
  for (lx = 0; lx < MXSUB; lx++) {

    jx = lx + isubx*MXSUB;

    /* Extract c1 and c2, and set kinetic rate terms */

    offsetue = (lx+1)*NVARS + (ly+1)*nvmxsub2;
    c1 = uext[offsetue];
    c2 = uext[offsetue+1];
    qq1 = Q1*c1*C3;
    qq2 = Q2*c1*c2;
```

```
      qq3 = q3*C3;
      qq4 = q4coef*c2;
      rkin1 = -qq1 - qq2 + 2.0*qq3 + qq4;
      rkin2 = qq1 - qq2 - qq4;

      /* Set vertical diffusion terms */

      c1dn = uext[offsetue-nvmxsub2];
      c2dn = uext[offsetue-nvmxsub2+1];
      c1up = uext[offsetue+nvmxsub2];
      c2up = uext[offsetue+nvmxsub2+1];
      vertd1 = cyup*(c1up - c1) - cydn*(c1 - c1dn);
      vertd2 = cyup*(c2up - c2) - cydn*(c2 - c2dn);

      /* Set horizontal diffusion and advection terms */

      c1lt = uext[offsetue-2];
      c2lt = uext[offsetue-1];
      c1rt = uext[offsetue+2];
      c2rt = uext[offsetue+3];
      hord1 = hordco*(c1rt - 2.0*c1 + c1lt);
      hord2 = hordco*(c2rt - 2.0*c2 + c2lt);
      horad1 = horaco*(c1rt - c1lt);
      horad2 = horaco*(c2rt - c2lt);

      /* Load all terms into dudata */

      offsetu = lx*NVARS + ly*nvmxsub;
      dudata[offsetu]   = vertd1 + hord1 + horad1 + rkin1;
      dudata[offsetu+1] = vertd2 + hord2 + horad2 + rkin2;
    }
  }

}


/***************** Functions Called by the CVODE Solver ******************/

/* f routine.  Evaluate f(t,y).  First call ucomm to do communication of
   subgrid boundary data into uext.  Then calculate f by a call to fcalc. */

static void f(integer N, real t, N_Vector u, N_Vector udot, void *f_data)
{
  real *udata, *dudata;
  UserData data;

  udata = N_VDATA(u);
  dudata = N_VDATA(udot);
```

```
  data = (UserData) f_data;


  /* Call ucomm to do inter-processor communicaiton */

  ucomm (N, t, u, data);

  /* Call fcalc to calculate all right-hand sides */

  fcalc (N, t, udata, dudata, data);

}


/* Preconditioner setup routine. Generate and preprocess P. */

static int Precond(integer N, real tn, N_Vector u, N_Vector fu, boole jok,
                   boole *jcurPtr, real gamma, N_Vector ewt, real h,
                   real uround, long int *nfePtr, void *P_data,
                   N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3)
{
  real c1, c2, cydn, cyup, diag, ydn, yup, q4coef, dely, verdco, hordco;
  real **(*P)[MYSUB], **(*Jbd)[MYSUB];
  integer nvmxsub, *(*pivot)[MYSUB], ier, offset;
  int lx, ly, jx, jy, isubx, isuby;
  real *udata, **a, **j;
  PreconData predata;
  UserData data;

  /* Make local copies of pointers in P_data, pointer to u's data,
     and PE index pair */

  predata = (PreconData) P_data;
  data = (UserData) (predata->f_data);
  P = predata->P;
  Jbd = predata->Jbd;
  pivot = predata->pivot;
  udata = N_VDATA(u);
  isubx = data->isubx;   isuby = data->isuby;
  nvmxsub = data->nvmxsub;

  if (jok) {

  /* jok = TRUE: Copy Jbd to P */

    for (ly = 0; ly < MYSUB; ly++)
      for (lx = 0; lx < MXSUB; lx++)
        dencopy(Jbd[lx][ly], P[lx][ly], NVARS);
```

```
*jcurPtr = FALSE;

}

else {

/* jok = FALSE: Generate Jbd from scratch and copy to P */

/* Make local copies of problem variables, for efficiency */

q4coef = data->q4;
dely = data->dy;
verdco = data->vdco;
hordco  = data->hdco;

/* Compute 2x2 diagonal Jacobian blocks (using q4 values
   computed on the last f call).  Load into P. */

  for (ly = 0; ly < MYSUB; ly++) {
    jy = ly + isuby*MYSUB;
    ydn = YMIN + (jy - .5)*dely;
    yup = ydn + dely;
    cydn = verdco*exp(0.2*ydn);
    cyup = verdco*exp(0.2*yup);
    diag = -(cydn + cyup + 2.0*hordco);
    for (lx = 0; lx < MXSUB; lx++) {
      jx = lx + isubx*MXSUB;
      offset = lx*NVARS + ly*nvmxsub;
      c1 = udata[offset];
      c2 = udata[offset+1];
      j = Jbd[lx][ly];
      a = P[lx][ly];
      IJth(j,1,1) = (-Q1*C3 - Q2*c2) + diag;
      IJth(j,1,2) = -Q2*c1 + q4coef;
      IJth(j,2,1) = Q1*C3 - Q2*c2;
      IJth(j,2,2) = (-Q2*c1 - q4coef) + diag;
      dencopy(j, a, NVARS);
    }
  }

*jcurPtr = TRUE;

}

/* Scale by -gamma */

  for (ly = 0; ly < MYSUB; ly++)
```

```
      for (lx = 0; lx < MXSUB; lx++)
        denscale(-gamma, P[lx][ly], NVARS);

  /* Add identity matrix and do LU decompositions on blocks in place */

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
      denaddI(P[lx][ly], NVARS);
      ier = gefa(P[lx][ly], NVARS, pivot[lx][ly]);
      if (ier != 0) return(1);
    }
  }

  return(0);
}


/* Preconditioner solve routine */

static int PSolve(integer N, real tn, N_Vector u, N_Vector fu, N_Vector vtemp,
                  real gamma, N_Vector ewt, real delta, long int *nfePtr,
                  N_Vector r, int lr, void *P_data, N_Vector z)
{
  real **(*P)[MYSUB];
  integer nvmxsub, *(*pivot)[MYSUB];
  int lx, ly;
  real *zdata, *v;
  PreconData predata;
  UserData data;

  /* Extract the P and pivot arrays from P_data */

  predata = (PreconData) P_data;
  data = (UserData) (predata->f_data);
  P = predata->P;
  pivot = predata->pivot;

  /* Solve the block-diagonal system Px = r using LU factors stored
     in P and pivot data in pivot, and return the solution in z.
     First copy vector r to z. */

  N_VScale(1.0, r, z);

  nvmxsub = data->nvmxsub;
  zdata = N_VDATA(z);

  for (lx = 0; lx < MXSUB; lx++) {
    for (ly = 0; ly < MYSUB; ly++) {
```

```
      v = &(zdata[lx*NVARS + ly*nvmxsub]);
      gesl(P[lx][ly], NVARS, pivot[lx][ly], v);
    }
  }

  return(0);
}
```