Example Programs for IDA v2.3.0

Alan C. Hindmarsh and Radu Serban
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

April 2005

UCRL-SM-208113

This research was supported under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Contents

1 Introduction

This report is intended to serve as a companion document to the User Documentation of IDA [2]. It provides details, with listings, on the example programs supplied with the IDA distribution package.

The IDA distribution contains, in the `sundials/ida/examples_ser` directory, the following four serial examples (using the NVECTOR_SERIAL module):

- `irobx` solves the Robertson chemical kinetics problem [3] which consists of two differential equations and one algebraic constraint.

  The problem is solved with the IDADENSE linear solver using a user-supplied Jacobian.

- `iheatsb` solves a 2-D heat equation, semidiscretized to a DAE on the unit square.

  This program solves the problem with the IDABAND linear solver and the default difference-quotient Jacobian approximation. For purposes of illustration, `IDACalcIC` is called to compute correct values at the boundary, given incorrect values as input initial guesses. The constraint $u > 0.0$ is imposed for all components.

- `iheatsk` solves the same problem as `iheatsb`, with the Krylov linear solver IDASPGMR. The preconditioner uses only the diagonal elements of the Jacobian.

- `iwebsb` solves a system of PDEs modelling a food web problem, with predator-prey interaction and diffusion, on the unit square in 2-D.

  The PDEs are discretized in space to a system of DAEs which are solved using the IDABAND linear solver with the default difference-quotient Jacobian approximation.

In the `sundials/ida/examples_par` directory, the IDA distribution contains the following four parallel examples (using the NVECTOR_PARALLEL module):

- `iheatpk` solves the same problem as `iheatbbd` with a user-supplied preconditioner which uses the diagonal elements of the Jacobian only.

- `iheatbbd` is a parallel implementation of the 2-D heat equation.

  This program solves the problem in parallel, using the Krylov linear solver IDASPGMR and the band-block diagonal preconditioner IDABBDPRE with half-bandwidths equal to 1.

- `iwebpk` solves the same problem as `iwebbbd` with a user-supplied preconditioner.

  The precondtioner supplied to IDASPGMR is the block-diagonal part of the Jacobian with $n_s \times n_s$ blocks arising from the reaction terms only (n_s is the number of species in the model).

- `iwebbbd` is a parallel implementation of the predator-prey problem.

  The problem is solved in parallel using the IDASPGMR linear solver and the IDABBDPRE preconditioner.

In the following sections, we give detailed descriptions of some (but not all) of these examples. The Appendices contain complete listings of those examples described below. We also give our output files for each of these examples, but users should be cautioned that their results may differ slightly from these. Solution values may differ within tolerances, and differences in cumulative counters, such as numbers of steps or Newton iterations, may differ from one machine environment to another by as much as 10% to 20%.

In the descriptions below, we make frequent references to the IDA User Document [2]. All citations to specific sections (e.g. §5.1) are references to parts of that User Document, unless explicitly stated otherwise.

Note. The examples in the IDA distribution are written in such a way as to compile and run for any combination of configuration options during the installation of SUNDIALS (see §2). As a consequence, they contain portions of code that will not be typically present in a user program. For example, all example programs make use of the variable `SUNDIALS_EXTENDED_PRECISION` to test if the solver libraries were built in extended precision and use the appropriate conversion specifiers in `printf` functions.

2 Serial example problems

2.1 A dense example: `irobx`

This example, due to Robertson [3], is a model of a three-species chemical kinetics system written in DAE form. Differential equations are given for species y_1 and y_2 while an algebraic equation determines y_3. The equations for the system concentrations $y_i(t)$ are:

$$\begin{cases} y_1' &= -.04y_1 + 10^4 y_2 y_3 \\ y_2' &= +.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2 \\ 0 &= y_1 + y_2 + y_3 - 1\,. \end{cases} \tag{1}$$

The initial values are taken as $y_1 = 1$, $y_2 = 0$, and $y_3 = 0$ This example computes the three concentration components on the interval from $t = 0$ through $t = 4 \cdot 10^{10}$.

For the source, listed in Appendix A, we give a rather detailed explanation of the parts of the program and their interaction with IDA.

Following the initial comment block, this program has a number of `#include` lines, which allow access to useful items in IDA header files. The `sundialstypes.h` file provides the definition of the type `realtype` (see §5.1 in the user guide [2] for details). For now, it suffices to read `realtype` as `double`. The `ida.h` file provides prototypes for the IDA functions to be called (excluding the linear solver selection function), and also a number of constants that are to be used in setting input arguments and testing the return value of `IDASolve`. The `idadense.h` file provides the prototype for the `IDADense` function. The `nvector_serial.h` file is the header file for the serial implementation of the NVECTOR module and includes definitions of the `N_Vector` type, a macro to access vector components, and prototypes for the serial implementation specific machine environment memory allocation and freeing functions. Finally, note that `idadense.h` also includes the `dense.h` file which provides the definition of the dense matrix type `DenseMat` and a macro for accessing matrix elements.

This program includes the user-defined accessor macro `IJth` that is useful in writing the problem functions in a form closely matching the mathematical description of the DAE system, i.e. with components numbered from 1 instead of from 0. The `IJth` macro is used to access elements of a dense matrix of type `DenseMat`. It is defined using the DENSE accessor macro `DENSE_ELEM` which numbers matrix rows and columns starting with 0. The macro `DENSE_ELEM` is fully described in §5.5.3.

The program prologue ends with prototypes of the two user-supplied functions that are called by IDA and the prototype of the private function `check_flag` which is used to test the return flag from the IDA user-callable functions.

After various declarations, the `main` program begins by allocating memory for the `yy`, `yp`, and `avtol` vectors using `N_VNew_Serial` with a length argument of `NEQ` (= 3). The lines following that load the initial values of the dependent variable vectors into `yy` and `yp` and set the realtive tolerance `rtol` and absolute tolerance vector `avtol`. Serial `N_Vector` values are set by first accessing the pointer to their underlying data using the macro `NV_DATA_S` defined by NVECTOR_SERIAL in `nvector_serial.h`.

The calls to `N_VNew_Serial`, and also later calls to `IDA***` functions, make use of a private function, `check_flag`, which examines the return value and prints a message if there was a failure. This `check_flag` function was written to be used for any serial SUNDIALS application.

The call to `IDACreate` creates the IDA solver memory block. The return value of this function is a pointer to the memory block for this problem. In the case of failure, the return value is `NULL`. This pointer must be passed in the remaining calls to IDA functions.

The call to `IDAMalloc` allocates the solver memory block. Its arguments include the name of the C function `resrob` defining the residual function $F(t, y, y')$, and the initial values of t, y, and y'. The argument `IDA_SV` specifies a vector of absolute tolerances, and this is followed by the address of the relative tolerance `rtol` and the absolute tolerance vector `avtol`. See §5.4.1 for full details of this call.

The calls to `IDADense` (see §5.4.2) and `IDADenseSetJacFn` (see §5.4.5) specify the IDADENSE linear solver with an analytic Jacobian supplied by the user-supplied function `jacrob`.

The actual solution of the DAE initial value problem is accomplished in the loop over values of the output time `tout`. In each pass of the loop, the program calls `IDASolve` in the `IDA_NORMAL` mode, meaning that the integrator is to take steps until it overshoots `tout` and then interpolate to $t =$`tout`, putting the computed value of y(`tout`) and y'(`tout`) into `yy` and `yp`, respectively, with `t` = `tout`. If an error occured during the call to `IDASolve`, the program returns 1 and terminates. On a successful return (indicated by a return value `IDA_SUCCESS`), the program prints the solution, the cumulative number of steps taken so far, and the current method order and step size.

Finally, the main program calls `PrintFinalStats` to extract and print several relevant statistical quantities, such as the total number of steps, the number of residual and Jacobian evaluations, and the number of error test and convergence test failures. It then calls `IDAFree` to free the IDA memory block and `NV_Destroy_Serial` to free the vectors `yy`, `yp`, and `avtol`.

The function `PrintFinalStats` used here is actually suitable for general use in applications of IDA to any problem with a dense Jacobian. It calls various `IDAGet***` and `IDADenseGet***` functions to obtain the relevant counters, and then prints them. Specifically, these are: the cumulative number of steps (`nst`), the number of residual evaluations (`nre`) (excluding those for difference-quotient Jacobian evaluations), the number of residual evaluations for Jacobian evaluations (`nreD`), the number of Jacobian evaluations (`njeD`), the number of nonlinear (Newton) iterations (`nni`), the number of local error test failures (`netf`), and the number of nonlinear convergence failures (`ncfn`). These optional outputs are described in §5.4.7.

The functions `resrob` (of type `IDAResFn`) and `jacrob` (of type `IDADenseJacFn`) are straightforward expressions of the DAE system. The function `jacrob` makes use of the macro `IJth` discussed above. See §5.5.1 for detailed specifications of `IDAResFn`.

Sample output from `irobx` follows.

`irobx sample output`

```

irobx: Robertson kinetics DAE serial example problem for IDA
       Three equation chemical kinetics problem.

Linear solver: IDADENSE, with user-supplied Jacobian.
Tolerance parameters:  rtol = 0.0001   atol = 1e-06 1e-10 1e-06
Initial conditions y0 = (1 0 0)
Constraints and id not used.

-----------------------------------------------------------------------
  t            y1           y2           y3      | nst  k      h
-----------------------------------------------------------------------
4.00e-01   9.8517e-01   3.3864e-05   1.4795e-02 |  77  3   1.1431e-01
```

```
4.00e+00   9.0551e-01   2.2403e-05   9.4470e-02 |  91  4   3.7037e-01
4.00e+01   7.1582e-01   9.1854e-06   2.8417e-01 | 127  4   2.9630e+00
4.00e+02   4.5051e-01   3.2227e-06   5.4949e-01 | 177  3   1.2405e+01
4.00e+03   1.8316e-01   8.9396e-07   8.1684e-01 | 228  3   2.7646e+02
4.00e+04   3.8985e-02   1.6218e-07   9.6101e-01 | 278  5   2.6140e+03
4.00e+05   4.9388e-03   1.9852e-08   9.9506e-01 | 324  5   2.7701e+04
4.00e+06   5.1763e-04   2.0716e-09   9.9948e-01 | 355  4   3.9788e+05
4.00e+07   5.1907e-05   2.0764e-10   9.9995e-01 | 380  3   6.3661e+06
4.00e+08   5.8818e-06   2.3527e-11   9.9999e-01 | 394  1   9.1671e+07
4.00e+09   7.0539e-07   2.8216e-12   1.0000e-00 | 402  1   1.4667e+09
4.00e+10  -7.3001e-07  -2.9200e-12   1.0000e+00 | 407  1   2.3468e+10

Final Run Statistics:

Number of steps                    = 407
Number of residual evaluations     = 557
Number of Jacobian evaluations     = 65
Number of nonlinear iterations     = 557
Number of error test failures      = 6
Number of nonlinear conv. failures = 0
```

2.2 A banded example: `iwebsb`

This example is a model of a multi-species food web [1], in which predator-prey relationships with diffusion in a 2-D spatial domain are simulated. Here we consider a model with $s = 2p$ species: p predators and p prey. Species $1, \ldots, p$ (the prey) satisfy rate equations, while species $p+1, \ldots, s$ (the predators) have infinitely fast reaction rates. The coupled PDEs for the species concentrations $c^i(x, y, t)$ are:

$$\begin{cases} \partial c^i / \partial t = R_i(x, y, c) + d_i (c^i_{xx} + c^i_{yy}) & i = 1, 2, \ldots, p \\ \qquad 0 = R_i(x, y, c) + d_i (c^i_{xx} + c^i_{yy}) & i = p+1, \ldots, s \ , \end{cases} \tag{2}$$

with

$$R_i(x, y, c) = c^i \left(b_i + \sum_{j=1}^{s} a_{ij} c^j \right) .$$

Here c denotes the vector $\{c^i\}$. The interaction and diffusion coefficients (a_{ij}, b_i, d_i) can be functions of (x, y) in general. The choices made for this test problem are as follows:

$$a_{ij} = \begin{cases} -1 & i = j \\ -0.5 \cdot 10^{-6} & i \le p, \ j > p \\ 10^4 & i > p, \ j \le p \\ 0 & \text{all other } (i, j) \, , \end{cases}$$

$$b_i = b_i(x, y) = \begin{cases} (1 + \alpha x y + \beta \sin(4\pi x) \sin(4\pi y)) & i \le p \\ -(1 + \alpha x y + \beta \sin(4\pi x) \sin(4\pi y)) & i > p \, , \end{cases}$$

and

$$d_i = \begin{cases} 1 & i \le p \\ 0.5 & i > p \, . \end{cases}$$

The spatial domain is the unit square $0 \leq x, y \leq 1$, and the time interval is $0 \leq t \leq 1$. The boundary conditions are of homogeneous Neumann type (zero normal derivatives) everywhere. The coefficients are such that a unique stable equilibrium is guaranteed to exist when $\alpha = \beta = 0$ [1]. Empirically, a stable equilibrium appears to exist for (2) when α and β are positive, although it may not be unique. In this problem we take $\alpha = 50$ and $\beta = 1000$. For the initial conditions, we set each prey concentration to a simple polynomial profile satisfying the boundary conditions, while the predator concentrations are all set to a flat value:

$$c^i(x, y, 0) = \begin{cases} 10 + i[16x(1-x)y(1-y)]^2 & i \leq p\,, \\ 10^5 & i > p\,. \end{cases}$$

We discretize this PDE system (2) (plus boundary conditions) with central differencing on an $L \times L$ mesh, so as to obtain a DAE system of size $N = sL^2$. The dependent variable vector C consists of the values $c^i(x_j, y_k, t)$ grouped first by species index i, then by x, and lastly by y. At each spatial mesh point, the system has a block of p ODE's followed by a block of p algebraic equations, all coupled. For this example, we take $p = 1, s = 2$, and $L = 20$. The Jacobian is banded, with half-bandwidths `mu = ml` $= sL = 40$.

The `iwebsb.c` program (listed in Appendix B) includes the file `idaband.h` in order to use the IDABAND linear solver. This file contains the prototype for the `IDABand` routine, the definition for the band matrix type `BandMat`, and the `BAND_COL` and `BAND_COL_ELEM` macros for accessing matrix elements. See §8.2. The main IDA header file `ida.h` is included for the prototypes of the solver user-callable functions and IDA constants, while the file `nvector_serial.h` is included for the definition of the serial `N_Vector` type. The header file `smalldense.h` is included for the `denalloc` function used in allocating memory for the user data structure.

The include lines at the top of the file are followed by definitions of problem constants which include the x and y mesh dimensions, `MX` and `MY`, the number of equations `NEQ`, the scalar relative and absolute tolerances `RTOL` and `ATOL`, and various parameters for the food-web problem.

Spatial discretization of the PDE naturally produces a DAE system in which equations are numbered by mesh coordinates (i, j). The user-defined macro `IJth_Vptr` isolates the translation for the mathematical two-dimensional index to the one-dimensional `N_Vector` index and allows the user to write clean, readable code to access components of the dependent variable. `IJ_Vptr(v,i,j)` returns a pointer to the location in `v` corresponding to the species with index `is` $= 0$, x-index `ix` $= i$, and y-index `jy` $= j$.

The type `UserData` is a pointer to a structure containing problem data used in the `resweb` function. This structure is allocated and initialized at the beginning of `main`. The pointer to it, called `webdata`, is then passed to `IDASetRData` and as a result it will be passed back to the `resweb` function each time it is called.

The `main` program is straightforward and very similar to that for `irobx`. The differences come from the use of the IDABAND linear solver and from the use of the consistent initial conditions algorithm in IDA to correct the initial values. `IDACalcIC` is called with the option `IDA_YA_YDP_INIT`, meaning that IDA is to compute the algebraic components of y and differential components of y', given the differential components of y. This option requires that the `N_Vector id` be set through a call to `IDASetId` specifying the differential and algebraic components. In this example, `id` has components equal to 1 for the prey (indicating differential variables) and 0 for the predators (algebraic variables).

Next, the `IDASolve` function is called in a loop over the output times, and the solution for the species concentrations at the bottom-left and top-right corners is printed, along with the cumulative number of time steps, current method order, and current step size.

Finally, the main program calls `PrintFinalStats` to get and print all of the relevant statistical quantities. It then calls `NV_Destroy_Serial` to free the vectors `cc`, `cp`, and `id`, and `IDAFree` to free the IDA memory block.

The function `PrintFinalStats` used here is actually suitable for general use in applications of IDA to any problem with a banded Jacobian. It calls various `IDAGet***` and `IDABandGet***` functions to obtain the relevant counters, and then prints them. Specifically, these are: the cumulative number of steps (`nst`), the number of residual evaluations (`nre`) (excluding those for difference-quotient Jacobian evaluations), the number of residual evaluations for Jacobian evaluations (`nreB`), the number of Jacobian evaluations (`njeB`), the number of nonlinear (Newton) iterations (`nni`), the number of local error test failures (`netf`), and the number of nonlinear convergence failures (`ncfn`). These optional outputs are described in §5.4.7.

The function `resweb` is a direct translation of the residual of (2). It first calls the private function `Fweb` to initialize the residual vector with the right-hand side of (2) and then it loops over all grid points, setting residual values appropriately for differential or algebraic components. The calculation of the interaction terms R_i is done in the function `WebRates`.

Sample output from `iwebsb` follows.

iwebsb sample output

```

iwebsb: Predator-prey DAE serial example problem for IDA

Number of species ns: 2     Mesh dimensions: 20 x 20     System size: 800
Tolerance parameters:  rtol = 1e-05   atol = 1e-05
Linear solver: IDABAND,  Band parameters mu = 40, ml = 40
CalcIC called to correct initial predator concentrations.

-----------------------------------------------------------
  t        bottom-left  top-right    | nst  k      h
-----------------------------------------------------------


0.00e+00   1.0000e+01   9.9949e+04   |   0  0   1.6310e-08
           9.9999e+04   9.9949e+04   |

1.00e-03   1.0318e+01   1.0822e+05   |  32  4   1.0823e-04
           1.0319e+05   1.0822e+05   |

1.00e-02   1.6189e+02   1.9735e+06   | 135  4   1.7964e-04
           1.6189e+06   1.9735e+06   |

1.00e-01   2.4019e+02   2.7072e+06   | 231  1   4.4212e-02
           2.4019e+06   2.7072e+06   |

4.00e-01   2.4019e+02   2.7072e+06   | 233  1   1.7685e-01
           2.4019e+06   2.7072e+06   |

7.00e-01   2.4019e+02   2.7072e+06   | 234  1   3.5370e-01
           2.4019e+06   2.7072e+06   |

1.00e+00   2.4019e+02   2.7072e+06   | 235  1   7.0740e-01
```

```
           2.4019e+06   2.7072e+06   |

-----------------------------------------------------------
Final run statistics:

Number of steps                    = 235
Number of residual evaluations     = 3319
Number of Jacobian evaluations     = 36
Number of nonlinear iterations     = 401
Number of error test failures      = 5
Number of nonlinear conv. failures = 0
```

2.3 A Krylov example: `iheatsk`

This example solves a discretized 2D heat PDE problem. The DAE system arises from the Dirichlet boundary condition $u = 0$, along with the differential equations arising from the discretization of the interior of the region.

The domain is the unit square $\Omega = \{0 \leq x, y \leq 1\}$ and the equations solved are:

$$\begin{cases} \partial u/\partial t = u_{xx} + u_{yy} & (x,y) \in \Omega \\ u = 0 & (x,y) \in \partial\Omega\,. \end{cases} \tag{3}$$

The time interval is $0 \leq t \leq 10.24$, and the initial conditions are $u = 16x(1-x)y(1-y)$.

We discretize the PDE system (3) (plus boundary conditions) with central differencing on a 10×10 mesh, so as to obtain a DAE system of size $N = 100$. The dependent variable vector u consists of the values $u(x_j, y_k, t)$ grouped first by x, and then by y. Each discrete boundary condition becomes an algebraic equation within the DAE system.

The source for this example is listed in appendix C. In this case, `idaspgmr.h` is included for the definitions of constants and function prototypes associated with the SPGMR method.

After various initializations (including a vector of constraints with all components set to 1 imposing all solution components to be non-negative), the main program creates and initializes the IDA memory block and then attaches the IDASPGMR linear solver using the default `MODIFIED_GS` Gram-Scmidt orthogonalization algorithm.

The calls to `IDASpgmrSetPrecSetupFn` and `IDASpgmrSetPsolveFn` specify the use of the user-supplied preconditioner with `data` being the pointer to user data passed to `PsolveHeat` and `PsetupHeat` whenever they are called (specified with the call to `IDASpgmrSetPrecData`). In a loop over the desired output times, `IDASolve` is called in `IDA_NORMAL` mode and the maximum solution norm is printed.

The `main` program then re-initializes the IDA solver and the IDASPGMR linear solver and solves the problem again, this time using the `CLASSICAL_GS` Gramm-Schmidt orthogonalization algorithm. Finally, memory for the IDA solver and for the various vectors used is deallocated.

The user-supplied residual function `resHeat`, of type `IDAResFn`, loads the DAE residual with the value of u on the boundary (representing the algebraic equations expressing the boundary conditions of (3)) and with the spatial discretization of the PDE (using central differences) in the rest of the domain.

The user-supplied functions `PsetupHeat` and `PsolveHeat` together define the left preconditoner matrix P approximating the system Jacobian matrix $J = \partial F/\partial u + \alpha \partial F/\partial u'$

(where the DAE system is $F(t, u, u') = 0$), and solve the linear systems $Pz = r$. Preconditioning is done in this case by keeping only the diagonal elements of the J matrix above, storing them as inverses in a vector `pp`, when computed in `PsetupHeat`, for subsequent use in `PsolveHeat`. In this instance, only `cj` $= \alpha$ and `data` (the user data structure) are used from the `PsetupHeat` argument list.

Sample output from `iheatsk` follows.

`iheatsk sample output`

```

iheatsk: Heat equation, serial example problem for IDA
         Discretized heat equation on 2D unit square.
         Zero boundary conditions, polynomial initial conditions.
         Mesh dimensions: 10 x 10        Total system size: 100

Tolerance parameters:  rtol = 0   atol = 0.001
Constraints set to force all solution components >= 0.
Linear solver: IDASPGMR, preconditioner using diagonal elements.


Case 1: gsytpe = MODIFIED_GS

   Output Summary (umax = max-norm of solution)

  time     umax       k  nst  nni  nje   nre   nreS     h      npe nps
----------------------------------------------------------------------
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
  0.64   5.54028e-21  1   39   51   77    51    77   1.05e-01   12 128
  1.28   0.00000e+00  1   41   53   77    53    77   4.21e-01   14 130
  2.56   0.00000e+00  1   43   55   77    55    77   1.69e+00   16 132
  5.12   0.00000e+00  1   44   56   77    56    77   3.37e+00   17 133
 10.24   0.00000e+00  1   45   57   77    57    77   6.74e+00   18 134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0


Case 2: gstype = CLASSICAL_GS

   Output Summary (umax = max-norm of solution)

  time     umax       k  nst  nni  nje   nre   nreS     h      npe nps
----------------------------------------------------------------------
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
  0.64   5.54028e-21  1   39   51   77    51    77   1.05e-01   12 128
  1.28   0.00000e+00  1   41   53   77    53    77   4.21e-01   14 130
  2.56   0.00000e+00  1   43   55   77    55    77   1.69e+00   16 132
  5.12   0.00000e+00  1   44   56   77    56    77   3.37e+00   17 133
 10.24   0.00000e+00  1   45   57   77    57    77   6.74e+00   18 134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0
```

3 Parallel example problems

3.1 A user preconditioner example: `iheatpk`

As an example of using IDA with the parallel MPI NVECTOR_PARALLEL module and the Krylov linear solver IDASPGMR with user-defined preconditioner, we provide the example `iheatpk` which solves the same 2-D heat PDE problem as `iheatsk`. The source is listed in Appendix D.

In the parallel setting, we can think of the processors as being laid out in a grid of size `NPEX` × `NPEY`, with each processor computing a subset of the solution vector on a submesh of size `MXSUB` × `MYSUB`. As a consequence, the computation of the residual vector requires that each processor exchange boundary information (namely the components at all interior subgrid boundaries) with its neighboring processors. The message-passing (implemented in the function `rescomm`) uses blocking sends, non-blocking receives, and receive-waiting, in routines `BSend`, `BRecvPost`, and `BRecvWait`, respectively. The data received from each neighboring processor is then loaded into a work array, `uext`, which contains this ghost cell data along with the local portion of the solution.

The local portion of the residual vector is then computed in the routine `reslocal`, which assumes that all inter-processor communication of data needed to calculate `rr` has already been done. Components at interior subgrid boundaries are assumed to be in the work array `uext`. The local portion of the solution vector `uu` is first copied into `uext`. The diffusion terms are evaluated in terms of the `uext` array, and the residuals are formed. The zero Dirichlet boundary conditions are handled here by including the boundary components in the residual, giving algebraic equations for the discrete boundary conditions.

The preconditioner (`PsolveHeat` and `PsetupHeat`) uses the diagonal elements of the Jacobian only and therefore involves only local calculations.

The `iheatpk` main program begins with MPI calls to initialize MPI and to set multi-processor environment parameters `npes` (number of processes) and `thispe` (local process index). Then the local and global vector lengths are set, the user-data structure `Userdata` is created and initialized, and `N_Vector` variables are created and initialized for the initial conditions (`uu` and `up`), for the vector `id` specifying the differential and algebraic components of the solution vector, and for the preconditioner (`pp`). As in `iheatsk`, constraints are passed to IDA through the `N_Vector constraints` and the function `IDASetConstraints`. A temporary `N_Vector res` is also created here, for use only in `SetInitialProfiles`. All components of `constraints` are set to 1.0 indicating that non-negativity constraints are to be imposed on each solution component. In addition, for illustration purposes, `iheatsk` also excludes the algebraic components of the solution (specified through the `N_Vector id`) from the error test by calling `IDASetSuppressAlg` with a flag `TRUE`.

Sample output from `iheatpk` follows.

`iheatpk sample output`

```

iheatpk: Heat equation, parallel example problem for IDA
         Discretized heat equation on 2D unit square.
         Zero boundary conditions, polynomial initial conditions.
         Mesh dimensions: 10 x 10        Total system size: 100

Subgrid dimensions: 5 x 5        Processor array: 2 x 2
Tolerance parameters:  rtol = 0   atol = 0.001
Constraints set to force all solution components >= 0.
```

```
SUPPRESSALG = TRUE to suppress local error testing on all boundary components.
Linear solver: IDASPGMR  Preconditioner: diagonal elements only.

   Output Summary (umax = max-norm of solution)

  time     umax       k  nst  nni  nli   nre   nreS     h      npe nps
----------------------------------------------------------------------
  0.00   9.75461e-01  0    0    0    0     0     0   0.00e+00    0   0
  0.01   8.24106e-01  2   12   14    7    14     7   2.56e-03    8  21
  0.02   6.88134e-01  3   15   18   12    18    12   5.12e-03    8  30
  0.04   4.70711e-01  3   18   24   21    24    21   6.58e-03    9  45
  0.08   2.16509e-01  3   22   29   30    29    30   1.32e-02    9  59
  0.16   4.57687e-02  4   28   36   44    36    44   1.32e-02    9  80
  0.32   2.09938e-03  4   35   44   67    44    67   2.63e-02   10 111
  0.64   5.54028e-21  1   39   51   77    51    77   1.05e-01   12 128
  1.28   3.85107e-20  1   41   53   77    53    77   4.21e-01   14 130
  2.56   5.00523e-20  1   43   55   77    55    77   1.69e+00   16 132
  5.12   1.58940e-19  1   44   56   77    56    77   3.37e+00   17 133
 10.24   5.12685e-19  1   45   57   77    57    77   6.74e+00   18 134

Error test failures            = 1
Nonlinear convergence failures = 0
Linear convergence failures    = 0
```

3.2 An IDABBDPRE preconditioner example: `iwebbbd`

In this example, `iwebbbd`, we solve the same food web problem as with `iwebsb`, but in parallel and with the IDASPGMR linear solver and using the IDABBDPRE module, which generates and uses a band-block-diagonal preconditioner. The source is listed in Appendix E.

As with `iheatpk`, we use a `NPEX` × `NPEY` processor grid, with an `MXSUB` × `MYSUB` submesh on each processor. Again, the residual evaluation begins with the communication of ghost data (in `rescomm`), followed by computation using an extended local array, `cext`, in the `reslocal` routine. The exterior Neumann boundary conditions are explicitly handled here by copying data from the first interior mesh line to the ghost cell locations in `cext`. Then the reaction and diffusion terms are evaluated in terms of the `cext` array, and the residuals are formed.

The Jacobian block on each processor is banded, and the half-bandwidths of that block are both equal to `NUM_SPECIES` · `MXSUB`. This is the value supplied as `mudq` and `mldq` in the call to `IDABBDPrecAlloc`. But in order to reduce storage and computation costs for preconditioning, we supply the values `mukeep` = `mlkeep` = 2 (= `NUM_SPECIES`) as the half-bandwidths of the retained band matrix blocks. This means that the Jacobian elements are computed with a difference quotient scheme using the true bandwidth of the block, but only a narrow band matrix (bandwidth 5) is kept as the preconditioner.

The function `reslocal` is also passed to the IDABBDPRE preconditioner as the `Gres` argument, while a `NULL` pointer is passed for the `Gcomm` argument (since all required communication for the evaluation of `Gres` was already done for `resweb`).

In the `iwebbbd` main program, following MPI initializations and creation of user data block `webdata` and `N_Vector` variables, the initial profiles are set, the IDA memory block is created and allocated, the IDABBDPRE preconditioner is initialized, and the IDASPGMR linear solver is attached to the IDA solver. The call to `IDACalcIC` corrects the initial values so that they are consistent with the DAE algebraic constraints.

In a loop over the desired output times, the main solver function `IDASolve` is called, and selected solution components (at the bottom-left and top-right corners of the computational domain) are collected on processor 0 and printed to `stdout`. The main program ends by printing final solver statistics, freeing memory, and finalizing MPI.

Sample output from `iwebbbd` follows.

`iwebbbd sample output`

```
iwebbbd: Predator-prey DAE parallel example problem for IDA

Number of species ns: 2     Mesh dimensions: 20 x 20     Total system size: 800
Subgrid dimensions: 10 x 10     Processor array: 2 x 2
Tolerance parameters:  rtol = 1e-05   atol = 1e-05
Linear solver: IDASPGMR     Max. Krylov dimension maxl: 12
Preconditioner: band-block-diagonal (IDABBDPRE), with parameters
     mudq = 20,  mldq = 20,  mukeep = 2,  mlkeep = 2
CalcIC called to correct initial predator concentrations

-----------------------------------------------------------
  t        bottom-left  top-right    | nst  k      h
-----------------------------------------------------------

0.00e+00   1.0000e+01   1.0000e+01   |   0  0   1.6310e-08
           9.9999e+04   9.9949e+04   |

1.00e-03   1.0318e+01   1.0827e+01   |  33  4   9.7404e-05
           1.0319e+05   1.0822e+05   |

1.00e-02   1.6189e+02   1.9735e+02   | 111  5   1.6153e-04
           1.6189e+06   1.9735e+06   |

1.00e-01   2.4019e+02   2.7072e+02   | 190  1   4.1353e-02
           2.4019e+06   2.7072e+06   |

4.00e-01   2.4019e+02   2.7072e+02   | 193  1   3.3082e-01
           2.4019e+06   2.7072e+06   |

7.00e-01   2.4019e+02   2.7072e+02   | 193  1   3.3082e-01
           2.4019e+06   2.7072e+06   |

1.00e+00   2.4019e+02   2.7072e+02   | 194  1   6.6164e-01
           2.4019e+06   2.7072e+06   |

-----------------------------------------------------------

Final statistics:

Number of steps                    = 194
Number of residual evaluations     = 899
Number of nonlinear iterations     = 237
Number of error test failures      = 0
Number of nonlinear conv. failures = 0
Number of linear iterations        = 660
Number of linear conv. failures    = 0

Number of preconditioner setups    = 26
Number of preconditioner solves    = 899
Number of local residual evals.    = 1092
```

References

[1] Peter N. Brown. Decay to uniform states in food webs. *SIAM J. Appl. Math.*, 46:376–392, 1986.

[2] A. C. Hindmarsh and R. Serban. User Documentation for IDA v2.2.0. Technical Report UCRL-SM-208112, LLNL, 2004.

[3] H. H. Robertson. The solution of a set of reaction rate equations. In J. Walsh, editor, *Numerical analysis: an introduction*, pages 178–182. Academ. Press, 1966.

A Listing of `irobx.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.16.2.4 $
 * $Date: 2005/04/06 23:34:13 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * This simple example problem for IDA/IDAS, due to Robertson,
 * is from chemical kinetics, and consists of the following three
 * equations:
 *
 *      dy1/dt = -.04*y1 + 1.e4*y2*y3
 *      dy2/dt = .04*y1 - 1.e4*y2*y3 - 3.e7*y2**2
 *         0   = y1 + y2 + y3 - 1
 *
 * on the interval from t = 0.0 to t = 4.e10, with initial
 * conditions: y1 = 1, y2 = y3 = 0.
 *
 * The problem is solved with IDA/IDAS using IDADENSE for the linear
 * solver, with a user-supplied Jacobian. Output is printed at
 * t = .4, 4, 40, ..., 4e10.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <math.h>
#include "sundialstypes.h"
#include "sundialsmath.h"
#include "nvector_serial.h"
#include "ida.h"
#include "idadense.h"

/* Problem Constants */

#define NEQ   3
#define NOUT  12

#define ZERO RCONST(0.0);
#define ONE  RCONST(1.0);

/* Macro to define dense matrix elements, indexed from 1. */

#define IJth(A,i,j) DENSE_ELEM(A,i-1,j-1)

/* Prototypes of functions called by IDA */

int resrob(realtype tres, N_Vector yy, N_Vector yp,
           N_Vector resval, void *rdata);

int jacrob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
           N_Vector resvec, realtype cj, void *jdata, DenseMat JJ,
           N_Vector tempv1, N_Vector tempv2, N_Vector tempv3);

/* Prototypes of private functions */
static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y);
static void PrintOutput(void *mem, realtype t, N_Vector y);
static void PrintFinalStats(void *mem);
static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(void)
{
  void *mem;
  N_Vector yy, yp, avtol;
  realtype rtol, *yval, *ypval, *atval;
  realtype t0, t1, tout, tret;
  int iout, retval;

  mem = NULL;
  yy = yp = avtol = NULL;
  yval = ypval = atval = NULL;

  /* Allocate N-vectors. */

  yy = N_VNew_Serial(NEQ);
  if(check_flag((void *)yy, "N_VNew_Serial", 0)) return(1);
  yp = N_VNew_Serial(NEQ);
  if(check_flag((void *)yp, "N_VNew_Serial", 0)) return(1);
  avtol = N_VNew_Serial(NEQ);
  if(check_flag((void *)avtol, "N_VNew_Serial", 0)) return(1);

  /* Create and initialize  y, y', and absolute tolerance vectors. */

  yval  = NV_DATA_S(yy);
  yval[0] = ONE;
  yval[1] = ZERO;
  yval[2] = ZERO;

  ypval = NV_DATA_S(yp);
  ypval[0]  = RCONST(-0.04);
  ypval[1]  = RCONST(0.04);
  ypval[2]  = ZERO;

  rtol = RCONST(1.0e-4);

  atval = NV_DATA_S(avtol);
  atval[0] = RCONST(1.0e-6);
  atval[1] = RCONST(1.0e-10);
  atval[2] = RCONST(1.0e-6);

  /* Integration limits */

  t0 = ZERO;
  t1 = RCONST(0.4);

  PrintHeader(rtol, avtol, yy);

  /* Call IDACreate and IDAMalloc to initialize IDA memory */

  mem = IDACreate();
  if(check_flag((void *)mem, "IDACreate", 0)) return(1);
  retval = IDAMalloc(mem, resrob, t0, yy, yp, IDA_SV, rtol, avtol);
  if(check_flag(&retval, "IDAMalloc", 1)) return(1);

  /* Free avtol */
  N_VDestroy_Serial(avtol);

  /* Call IDADense and set up the linear solver. */

  retval = IDADense(mem, NEQ);
  if(check_flag(&retval, "IDADense", 1)) return(1);
  retval = IDADenseSetJacFn(mem, jacrob, NULL);
  if(check_flag(&retval, "IDADenseSetJacFn", 1)) return(1);

  /* Loop over tout values and call IDASolve. */

  for (tout = t1, iout = 1; iout <= NOUT ; iout++, tout *= RCONST(10.0)) {
    retval=IDASolve(mem, tout, &tret, yy, yp, IDA_NORMAL);
    if(check_flag(&retval, "IDASolve", 1)) return(1);
    PrintOutput(mem,tret,yy);
  }

  PrintFinalStats(mem);

  /* Free memory */

  IDAFree(mem);
  N_VDestroy_Serial(yy);
  N_VDestroy_Serial(yp);

  return(0);

}

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY IDA
 *--------------------------------------------------------------------
 */

/*
 * Define the system residual function.
 */

```

```
int resrob(realtype tres, N_Vector yy, N_Vector yp, N_Vector rr, void *rdata)
{
  realtype *yval, *ypval, *rval;

  yval = NV_DATA_S(yy);
  ypval = NV_DATA_S(yp);
  rval = NV_DATA_S(rr);

  rval[0]  = RCONST(-0.04)*yval[0] + RCONST(1.0e4)*yval[1]*yval[2];
  rval[1]  = -rval[0] - RCONST(3.0e7)*yval[1]*yval[1] - ypval[1];
  rval[0] -=  ypval[0];
  rval[2]  =  yval[0] + yval[1] + yval[2] - ONE;

  return(0);

}

/*
 * Define the Jacobian function.
 */

int jacrob(long int Neq, realtype tt, N_Vector yy, N_Vector yp,
           N_Vector resvec, realtype cj, void *jdata, DenseMat JJ,
           N_Vector tempv1, N_Vector tempv2, N_Vector tempv3)
{

  realtype *yval;

  yval = NV_DATA_S(yy);

  IJth(JJ,1,1) = RCONST(-0.04) - cj;
  IJth(JJ,2,1) = RCONST(0.04);
  IJth(JJ,3,1) = ONE;
  IJth(JJ,1,2) = RCONST(1.0e4)*yval[2];
  IJth(JJ,2,2) = RCONST(-1.0e4)*yval[2] - RCONST(6.0e7)*yval[1] - cj;
  IJth(JJ,3,2) = ONE;
  IJth(JJ,1,3) = RCONST(1.0e4)*yval[1];
  IJth(JJ,2,3) = RCONST(-1.0e4)*yval[1];
  IJth(JJ,3,3) = ONE;

  return(0);

}

/*
 *--------------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *--------------------------------------------------------------------
 */

/*
 * Print first lines of output (problem description)
 */

```

```
static void PrintHeader(realtype rtol, N_Vector avtol, N_Vector y)
{
  realtype *atval, *yval;

  atval  = NV_DATA_S(avtol);
  yval  = NV_DATA_S(y);

  printf("\nirobx: Robertson kinetics DAE serial example problem for IDA \n");
  printf("       Three equation chemical kinetics problem. \n\n");
  printf("Linear solver: IDADENSE, with user-supplied Jacobian.\n");
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg %Lg %Lg \n",
         rtol, atval[0],atval[1],atval[2]);
  printf("Initial conditions y0 = (%Lg %Lg %Lg)\n",
         yval[0], yval[1], yval[2]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg %lg %lg \n",
         rtol, atval[0],atval[1],atval[2]);
  printf("Initial conditions y0 = (%lg %lg %lg)\n",
         yval[0], yval[1], yval[2]);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g %g %g \n",
         rtol, atval[0],atval[1],atval[2]);
  printf("Initial conditions y0 = (%g %g %g)\n",
         yval[0], yval[1], yval[2]);
#endif
  printf("Constraints and id not used.\n\n");
  printf("---------------------------------------------------------------------\n");
  printf("  t           y1           y2           y3");
  printf("      | nst  k      h\n");
  printf("---------------------------------------------------------------------\n");

}

/*
 * Print Output
 */

static void PrintOutput(void *mem, realtype t, N_Vector y)
{
  realtype *yval;
  int retval, kused;
  long int nst;
  realtype hused;

  yval  = NV_DATA_S(y);

  retval = IDAGetLastOrder(mem, &kused);
  check_flag(&retval, "IDAGetLastOrder", 1);
  retval = IDAGetNumSteps(mem, &nst);
  check_flag(&retval, "IDAGetNumSteps", 1);
  retval = IDAGetLastStep(mem, &hused);
  check_flag(&retval, "IDAGetLastStep", 1);
#if defined(SUNDIALS_EXTENDED_PRECISION)
```

```
  printf("%8.2Le %12.4Le %12.4Le %12.4Le | %3ld  %1d %12.4Le\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%8.2le %12.4le %12.4le %12.4le | %3ld  %1d %12.4le\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#else
  printf("%8.2e %12.4e %12.4e %12.4e | %3ld  %1d %12.4e\n",
         t, yval[0], yval[1], yval[2], nst, kused, hused);
#endif
}

/*
 * Print final integrator statistics
 */

static void PrintFinalStats(void *mem)
{
  int retval;
  long int nst, nni, njeD, nre, nreD, netf, ncfn;

  retval = IDAGetNumSteps(mem, &nst);
  check_flag(&retval, "IDAGetNumSteps", 1);
  retval = IDAGetNumResEvals(mem, &nre);
  check_flag(&retval, "IDAGetNumResEvals", 1);
  retval = IDADenseGetNumJacEvals(mem, &njeD);
  check_flag(&retval, "IDADenseGetNumJacEvals", 1);
  retval = IDAGetNumNonlinSolvIters(mem, &nni);
  check_flag(&retval, "IDAGetNumNonlinSolvIters", 1);
  retval = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&retval, "IDAGetNumErrTestFails", 1);
  retval = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&retval, "IDAGetNumNonlinSolvConvFails", 1);
  retval = IDADenseGetNumResEvals(mem, &nreD);
  check_flag(&retval, "IDADenseGetNumResEvals", 1);

  printf("\nFinal Run Statistics: \n\n");
  printf("Number of steps                    = %ld\n", nst);
  printf("Number of residual evaluations     = %ld\n", nre+nreD);
  printf("Number of Jacobian evaluations     = %ld\n", njeD);
  printf("Number of nonlinear iterations     = %ld\n", nni);
  printf("Number of error test failures      = %ld\n", netf);
  printf("Number of nonlinear conv. failures = %ld\n", ncfn);

}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */
```

```

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;
  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr,
            "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1);
  } else if (opt == 1) {
    /* Check if flag < 0 */
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1);
    }
  } else if (opt == 2 && flagvalue == NULL) {
    /* Check if function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1);
  }

  return(0);
}
```

B Listing of `iwebsb.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.18.2.2 $
 * $Date: 2005/04/04 22:36:47 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example program for IDA: Food web problem.
 *
 * This example program (serial version) uses the IDABAND linear
 * solver, and IDACalcIC for initial condition calculation.
 *
 * The mathematical problem solved in this example is a DAE system
 * that arises from a system of partial differential equations after
 * spatial discretization. The PDE system is a food web population
 * model, with predator-prey interaction and diffusion on the unit
 * square in two dimensions. The dependent variable vector is:
 *
 *         1   2         ns
 *   c = (c , c ,  ..., c  ) , ns = 2 * np
 *
 * and the PDE's are as follows:
 *
 *     i             i      i
 *   dc /dt = d(i)*(c    + c  )  +  R (x,y,c)   (i = 1,...,np)
 *                   xx     yy       i
 *
 *              i      i
 *   0 = d(i)*(c    + c  )  +  R (x,y,c)   (i = np+1,...,ns)
 *              xx     yy       i
 *
 *   where the reaction terms R are:
 *
 *                   i             ns         j
 *   R  (x,y,c)  =  c  * (b(i)  + sum a(i,j)*c )
 *    i                           j=1
 *
 * The number of species is ns = 2 * np, with the first np being
 * prey and the last np being predators. The coefficients a(i,j),
 * b(i), d(i) are:
 *
 *  a(i,i) = -AA   (all i)
 *  a(i,j) = -GG   (i <= np , j >  np)
 *  a(i,j) =  EE   (i >  np, j <= np)
 *  all other a(i,j) = 0
 *  b(i) = BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i <= np)
 *  b(i) =-BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y)) (i  > np)
 *  d(i) = DPREY   (i <= np)
 *  d(i) = DPRED   (i > np)
 *
 * The various scalar parameters required are set using '#define'
 * statements or directly in routine InitUserData. In this program,
 * np = 1, ns = 2. The boundary conditions are homogeneous Neumann:
 * normal derivative = 0.
 *
 * A polynomial in x and y is used to set the initial values of the
 * first np variables (the prey variables) at each x,y location,
 * while initial values for the remaining (predator) variables are
 * set to a flat value, which is corrected by IDACalcIC.
 *
 * The PDEs are discretized by central differencing on a MX by MY
 * mesh.
 *
 * The DAE system is solved by IDA using the IDABAND linear solver.
 * Output is printed at t = 0, .001, .01, .1, .4, .7, 1.
 * -----------------------------------------------------------------
 * References:
 * [1] Peter N. Brown and Alan C. Hindmarsh,
 *     Reduced Storage Matrix Methods in Stiff ODE systems, Journal
 *     of Applied Mathematics and Computation, Vol. 31 (May 1989),
 *     pp. 40-91.
 *
 * [2] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Using Krylov Methods in the Solution of Large-Scale
 *     Differential-Algebraic Systems, SIAM J. Sci. Comput., 15
 *     (1994), pp. 1467-1488.
 *
 * [3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Consistent Initial Condition Calculation for Differential-
 *     Algebraic Systems, SIAM J. Sci. Comput., 19 (1998),
 *     pp. 1495-1512.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sundialstypes.h"   /* Definitions of realtype and booleantype       */
#include "ida.h"             /* Main header file                              */
#include "idaband.h"         /* Use IDABAND linear solver                     */
#include "nvector_serial.h"  /* Definitions of type N_Vector, macro NV_DATA_S */
#include "smalldense.h"      /* Contains definitions for denalloc routine     */

/* Problem Constants. */

#define NPREY       1              /* No. of prey (= no. of predators). */
#define NUM_SPECIES 2*NPREY

#define PI          RCONST(3.1415926535898)
#define FOURPI      (RCONST(4.0)*PI)

#define MX          20             /* MX = number of x mesh points      */
#define MY          20             /* MY = number of y mesh points      */
#define NSMX        (NUM_SPECIES * MX)
#define NEQ         (NUM_SPECIES*MX*MY)
```

```
#define AA          RCONST(1.0)    /* Coefficient in above eqns. for a  */
#define EE          RCONST(10000.) /* Coefficient in above eqns. for a  */
#define GG          RCONST(0.5e-6) /* Coefficient in above eqns. for a  */
#define BB          RCONST(1.0)    /* Coefficient in above eqns. for b  */
#define DPREY       RCONST(1.0)    /* Coefficient in above eqns. for d  */
#define DPRED       RCONST(0.05)   /* Coefficient in above eqns. for d  */
#define ALPHA       RCONST(50.)    /* Coefficient alpha in above eqns.  */
#define BETA        RCONST(1000.)  /* Coefficient beta in above eqns.   */
#define AX          RCONST(1.0)    /* Total range of x variable         */
#define AY          RCONST(1.0)    /* Total range of y variable         */
#define RTOL        RCONST(1.e-5)  /* Relative tolerance                */
#define ATOL        RCONST(1.e-5)  /* Absolute tolerance                */
#define NOUT        6              /* Number of output times            */
#define TMULT       RCONST(10.0)   /* Multiplier for tout values        */
#define TADD        RCONST(0.3)    /* Increment for tout values         */
#define ZERO        RCONST(0.)
#define ONE         RCONST(1.0)

/*
 * User-defined vector and accessor macro: IJ_Vptr.
 * IJ_Vptr is defined in order to express the underlying 3-D structure of
 * the dependent variable vector from its underlying 1-D storage (an N_Vector).
 * IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
 * species index is = 0, x-index ix = i, and y-index jy = j.
 */

#define IJ_Vptr(vv,i,j) (&NV_Ith_S(vv, (i)*NUM_SPECIES + (j)*NSMX))

/* Type: UserData.  Contains problem constants, etc. */

typedef struct {
  long int Neq, ns, np, mx, my;
  realtype dx, dy, **acoef;
  realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef[NUM_SPECIES];
  N_Vector rates;
} *UserData;

/* Prototypes for functions called by the IDA Solver. */

static int resweb(realtype time, N_Vector cc, N_Vector cp, N_Vector resval,
                  void *rdata);

/* Prototypes for private Helper Functions. */

static void InitUserData(UserData webdata);
static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
                               UserData webdata);
static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol);
static void PrintOutput(void *mem, N_Vector c, realtype t);
static void PrintFinalStats(void *mem);
static void Fweb(realtype tcalc, N_Vector cc, N_Vector crate, UserData webdata);
static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                     UserData webdata);
static realtype dotprod(long int size, realtype *x1, realtype *x2);
static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main()
{
  void *mem;
  UserData webdata;
  N_Vector cc, cp, id;
  int iout, retval;
  long int mu, ml;
  realtype rtol, atol, t0, tout, tret;

  mem = NULL;
  webdata = NULL;
  cc = cp = id = NULL;

  /* Allocate and initialize user data block webdata. */

  webdata = (UserData) malloc(sizeof *webdata);
  webdata->rates = N_VNew_Serial(NEQ);
  webdata->acoef = denalloc(NUM_SPECIES);

  InitUserData(webdata);

  /* Allocate N-vectors and initialize cc, cp, and id. */

  cc  = N_VNew_Serial(NEQ);
  if(check_flag((void *)cc, "N_VNew_Serial", 0)) return(1);

  cp  = N_VNew_Serial(NEQ);
  if(check_flag((void *)cp, "N_VNew_Serial", 0)) return(1);

  id  = N_VNew_Serial(NEQ);
  if(check_flag((void *)id, "N_VNew_Serial", 0)) return(1);

  SetInitialProfiles(cc, cp, id, webdata);

  /* Set remaining inputs to IDAMalloc. */

  t0 = ZERO;
  rtol = RTOL;
  atol = ATOL;

  /* Call IDACreate and IDAMalloc to initialize IDA. */

  mem = IDACreate();
  if(check_flag((void *)mem, "IDACreate", 0)) return(1);

  retval = IDASetRdata(mem, webdata);
  if(check_flag(&retval, "IDASetRdata", 1)) return(1);

  retval = IDASetId(mem, id);
  if(check_flag(&retval, "IDASetId", 1)) return(1);

  retval = IDAMalloc(mem, resweb, t0, cc, cp, IDA_SS, rtol, &atol);
  if(check_flag(&retval, "IDAMalloc", 1)) return(1);

  /* Call IDABand to specify the IDA linear solver. */

  mu = ml = NSMX;
  retval = IDABand(mem, NEQ, mu, ml);
  if(check_flag(&retval, "IDABand", 1)) return(1);

  /* Call IDACalcIC (with default options) to correct the initial values. */

  tout = RCONST(0.001);
  retval = IDACalcIC(mem, t0, cc, cp, IDA_YA_YDP_INIT, tout);
  if(check_flag(&retval, "IDACalcIC", 1)) return(1);

  /* Print heading, basic parameters, and initial values. */

  PrintHeader(mu, ml, rtol, atol);
  PrintOutput(mem, cc, ZERO);

  /* Loop over iout, call IDASolve (normal mode), print selected output. */

  for (iout = 1; iout <= NOUT; iout++) {

    retval = IDASolve(mem, tout, &tret, cc, cp, IDA_NORMAL);
    if(check_flag(&retval, "IDASolve", 1)) return(retval);

    PrintOutput(mem, cc, tret);

    if (iout < 3) tout *= TMULT; else tout += TADD;

  }

  /* Print final statistics and free memory. */

  PrintFinalStats(mem);

  /* Free memory */

  IDAFree(mem);

  N_VDestroy_Serial(cc);
  N_VDestroy_Serial(cp);
  N_VDestroy_Serial(id);


  denfree(webdata->acoef);
  N_VDestroy_Serial(webdata->rates);
  free(webdata);

  return(0);
}

/* Define lines for readability in later routines */

#define acoef  (webdata->acoef)
#define bcoef  (webdata->bcoef)
#define cox    (webdata->cox)
#define coy    (webdata->coy)

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY IDA
 *--------------------------------------------------------------------
 */

/*
 * resweb: System residual function for predator-prey system.
 * This routine calls Fweb to get all the right-hand sides of the
 * equations, then loads the residual vector accordingly,
 * using cp in the case of prey species.
 */

static int resweb(realtype tt, N_Vector cc, N_Vector cp,
                  N_Vector res,  void *rdata)
{
  long int jx, jy, is, yloc, loc, np;
  realtype *resv, *cpv;
  UserData webdata;

  webdata = (UserData)rdata;

  cpv = NV_DATA_S(cp);
  resv = NV_DATA_S(res);
  np = webdata->np;

  /* Call Fweb to set res to vector of right-hand sides. */
  Fweb(tt, cc, res, webdata);

  /* Loop over all grid points, setting residual values appropriately
     for differential or algebraic components.                        */

  for (jy = 0; jy < MY; jy++) {
    yloc = NSMX * jy;
    for (jx = 0; jx < MX; jx++) {
      loc = yloc + NUM_SPECIES * jx;
      for (is = 0; is < NUM_SPECIES; is++) {
        if (is < np)
          resv[loc+is] = cpv[loc+is] - resv[loc+is];
        else
          resv[loc+is] = -resv[loc+is];
      }
    }
  }

  return(0);

}

/*
 *-----------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *-----------------------------------------------------------------
 */

/*
 * InitUserData: Load problem constants in webdata (of type UserData).
 */

static void InitUserData(UserData webdata)
{
  int i, j, np;
  realtype *a1,*a2, *a3, *a4, dx2, dy2;

  webdata->mx = MX;
  webdata->my = MY;
  webdata->ns = NUM_SPECIES;
  webdata->np = NPREY;
  webdata->dx = AX/(MX-1);
  webdata->dy = AY/(MY-1);
  webdata->Neq= NEQ;

  /* Set up the coefficients a and b, and others found in the equations. */
  np = webdata->np;
  dx2 = (webdata->dx)*(webdata->dx); dy2 = (webdata->dy)*(webdata->dy);

  for (i = 0; i < np; i++) {
    a1 = &(acoef[i][np]);
    a2 = &(acoef[i+np][0]);
    a3 = &(acoef[i][0]);
    a4 = &(acoef[i+np][np]);
    /*  Fill in the portion of acoef in the four quadrants, row by row. */
    for (j = 0; j < np; j++) {
      *a1++ =  -GG;
      *a2++ =   EE;
      *a3++ = ZERO;
      *a4++ = ZERO;
    }

    /* Reset the diagonal elements of acoef to -AA. */
    acoef[i][i] = -AA; acoef[i+np][i+np] = -AA;

    /* Set coefficients for b and diffusion terms. */
    bcoef[i] = BB; bcoef[i+np] = -BB;
    cox[i] = DPREY/dx2; cox[i+np] = DPRED/dx2;
    coy[i] = DPREY/dy2; coy[i+np] = DPRED/dy2;
  }
```

```

}

/*
 * SetInitialProfiles: Set initial conditions in cc, cp, and id.
 * A polynomial profile is used for the prey cc values, and a constant
 * (1.0e5) is loaded as the initial guess for the predator cc values.
 * The id values are set to 1 for the prey and 0 for the predators.
 * The prey cp values are set according to the given system, and
 * the predator cp values are set to zero.
 */

static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
                               UserData webdata)
{
  long int loc, yloc, is, jx, jy, np;
  realtype xx, yy, xyfactor, fac;
  realtype *ccv, *cpv, *idv;

  ccv = NV_DATA_S(cc);
  cpv = NV_DATA_S(cp);
  idv = NV_DATA_S(id);
  np = webdata->np;

  /* Loop over grid, load cc values and id values. */
  for (jy = 0; jy < MY; jy++) {
    yy = jy * webdata->dy;
    yloc = NSMX * jy;
    for (jx = 0; jx < MX; jx++) {
      xx = jx * webdata->dx;
      xyfactor = RCONST(16.0)*xx*(ONE-xx)*yy*(ONE-yy);
      xyfactor *= xyfactor;
      loc = yloc + NUM_SPECIES*jx;
      fac = ONE + ALPHA * xx * yy + BETA * sin(FOURPI*xx) * sin(FOURPI*yy);

      for (is = 0; is < NUM_SPECIES; is++) {
        if (is < np) {
            ccv[loc+is] = RCONST(10.0) + (realtype)(is+1) * xyfactor;
          idv[loc+is] = ONE;
        }
        else {
          ccv[loc+is] = RCONST(1.0e5);
          idv[loc+is] = ZERO;
        }
      }
    }
  }

  /* Set c' for the prey by calling the function Fweb. */
  Fweb(ZERO, cc, cp, webdata);

  /* Set c' for predators to 0. */
  for (jy = 0; jy < MY; jy++) {
    yloc = NSMX * jy;
    for (jx = 0; jx < MX; jx++) {
      loc = yloc + NUM_SPECIES * jx;
      for (is = np; is < NUM_SPECIES; is++) {
        cpv[loc+is] = ZERO;
      }
    }
  }
}

/*
 * Print first lines of output (problem description)
 */

static void PrintHeader(long int mu, long int ml, realtype rtol, realtype atol)
{
  printf("\niwebsb: Predator-prey DAE serial example problem for IDA \n\n");
  printf("Number of species ns: %d", NUM_SPECIES);
  printf("     Mesh dimensions: %d x %d", MX, MY);
  printf("     System size: %d\n", NEQ);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
#endif
  printf("Linear solver: IDABAND,  Band parameters mu = %ld, ml = %ld\n",mu,ml);
  printf("CalcIC called to correct initial predator concentrations.\n\n");
  printf("-----------------------------------------------------------\n");
  printf("  t        bottom-left  top-right");
  printf("    | nst  k      h\n");
  printf("-----------------------------------------------------------\n\n");

}

/*
 * PrintOutput: Print output values at output time t = tt.
 * Selected run statistics are printed.  Then values of the concentrations
 * are printed for the bottom left and top right grid points only.
 */

static void PrintOutput(void *mem, N_Vector c, realtype t)
{
  int i, kused, flag;
  long int nst;
  realtype *c_bl, *c_tr, hused;

  flag = IDAGetLastOrder(mem, &kused);
  check_flag(&flag, "IDAGetLastOrder", 1);
  flag = IDAGetNumSteps(mem, &nst);
  check_flag(&flag, "IDAGetNumSteps", 1);
  flag = IDAGetLastStep(mem, &hused);
  check_flag(&flag, "IDAGetLastStep", 1);

  c_bl = IJ_Vptr(c,0,0);
  c_tr = IJ_Vptr(c,MX-1,MY-1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("%8.2Le %12.4Le %12.4Le   | %3ld  %1d %12.4Le\n",
         t, c_bl[0], c_tr[1], nst, kused, hused);
  for (i=1;i<NUM_SPECIES;i++)
    printf("         %12.4Le %12.4Le   |\n",c_bl[i],c_tr[i]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("%8.2le %12.4le %12.4le   | %3ld  %1d %12.4le\n",
         t, c_bl[0], c_tr[1], nst, kused, hused);
  for (i=1;i<NUM_SPECIES;i++)
    printf("         %12.4le %12.4le   |\n",c_bl[i],c_tr[i]);
#else
  printf("%8.2e %12.4e %12.4e   | %3ld  %1d %12.4e\n",
         t, c_bl[0], c_tr[1], nst, kused, hused);
  for (i=1;i<NUM_SPECIES;i++)
    printf("         %12.4e %12.4e   |\n",c_bl[i],c_tr[i]);
#endif

  printf("\n");
}

/*
 * PrintFinalStats: Print final run data contained in iopt.
 */

static void PrintFinalStats(void *mem)
{
  long int nst, nre, nreB, nni, nje, netf, ncfn;
  int flag;

  flag = IDAGetNumSteps(mem, &nst);
  check_flag(&flag, "IDAGetNumSteps", 1);
  flag = IDAGetNumNonlinSolvIters(mem, &nni);
  check_flag(&flag, "IDAGetNumNonlinSolvIters", 1);
  flag = IDAGetNumResEvals(mem, &nre);
  check_flag(&flag, "IDAGetNumResEvals", 1);
  flag = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&flag, "IDAGetNumErrTestFails", 1);
  flag = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&flag, "IDAGetNumNonlinSolvConvFails", 1);
  flag = IDABandGetNumJacEvals(mem, &nje);
  check_flag(&flag, "IDABandGetNumJacEvals", 1);
  flag = IDABandGetNumResEvals(mem, &nreB);
  check_flag(&flag, "IDABandGetNumResEvals", 1);

  printf("-----------------------------------------------------------\n");
  printf("Final run statistics: \n\n");
  printf("Number of steps                    = %ld\n", nst);
  printf("Number of residual evaluations     = %ld\n", nre+nreB);
  printf("Number of Jacobian evaluations     = %ld\n", nje);
  printf("Number of nonlinear iterations     = %ld\n", nni);
  printf("Number of error test failures      = %ld\n", netf);
  printf("Number of nonlinear conv. failures = %ld\n", ncfn);

}

/*
 * Fweb: Rate function for the food-web problem.
 * This routine computes the right-hand sides of the system equations,
 * consisting of the diffusion term and interaction term.
 * The interaction term is computed by the function WebRates.
 */

static void Fweb(realtype tcalc, N_Vector cc, N_Vector crate,
                 UserData webdata)
{
  long int jx, jy, is, idyu, idyl, idxu, idxl;
  realtype xx, yy, *cxy, *ratesxy, *cratexy, dcyli, dcyui, dcxli, dcxui;

  /* Loop over grid points, evaluate interaction vector (length ns),
     form diffusion difference terms, and load crate.                    */

  for (jy = 0; jy < MY; jy++) {
    yy = (webdata->dy) * jy ;
    idyu = (jy!=MY-1) ? NSMX : -NSMX;
    idyl = (jy!= 0  ) ? NSMX : -NSMX;

    for (jx = 0; jx < MX; jx++) {
      xx = (webdata->dx) * jx;
      idxu = (jx!= MX-1) ?  NUM_SPECIES : -NUM_SPECIES;
      idxl = (jx!=  0  ) ?  NUM_SPECIES : -NUM_SPECIES;
      cxy = IJ_Vptr(cc,jx,jy);
      ratesxy = IJ_Vptr(webdata->rates,jx,jy);
      cratexy = IJ_Vptr(crate,jx,jy);

      /* Get interaction vector at this grid point. */
      WebRates(xx, yy, cxy, ratesxy, webdata);

      /* Loop over species, do differencing, load crate segment. */
      for (is = 0; is < NUM_SPECIES; is++) {

        /* Differencing in y. */
        dcyli = *(cxy+is) - *(cxy - idyl + is) ;
        dcyui = *(cxy + idyu + is) - *(cxy+is);

        /* Differencing in x. */
        dcxli = *(cxy+is) - *(cxy - idxl + is);
        dcxui = *(cxy + idxu +is) - *(cxy+is);

        /* Compute the crate values at (xx,yy). */
        cratexy[is] = coy[is] * (dcyui - dcyli) +
          cox[is] * (dcxui - dcxli) + ratesxy[is];

      } /* End is loop */
    } /* End of jx loop */
  } /* End of jy loop */
```

```

}

/*
 * WebRates: Evaluate reaction rates at a given spatial point.
 * At a given (x,y), evaluate the array of ns reaction terms R.
 */

static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                     UserData webdata)
{
  int is;
  realtype fac;

  for (is = 0; is < NUM_SPECIES; is++)
    ratesxy[is] = dotprod(NUM_SPECIES, cxy, acoef[is]);

  fac = ONE + ALPHA*xx*yy + BETA*sin(FOURPI*xx)*sin(FOURPI*yy);

  for (is = 0; is < NUM_SPECIES; is++)
    ratesxy[is] = cxy[is]*( bcoef[is]*fac + ratesxy[is] );

}

/*
 * dotprod: dot product routine for realtype arrays, for use by WebRates.
 */

static realtype dotprod(long int size, realtype *x1, realtype *x2)
{
  long int i;
  realtype *xx1, *xx2, temp = ZERO;

  xx1 = x1; xx2 = x2;
  for (i = 0; i < size; i++) temp += (*xx1++) * (*xx2++);
  return(temp);

}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

  if (opt == 0 && flagvalue == NULL) {
    /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1);
  } else if (opt == 1) {
    /* Check if flag < 0 */
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1);
    }
  } else if (opt == 2 && flagvalue == NULL) {
    /* Check if function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1);
  }

  return(0);
}
```

C Listing of `iheatsk.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.16.2.4 $
 * $Date: 2005/04/06 23:34:13 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem for IDA/IDAS: 2D heat equation, serial, GMRES.
 *
 * This example solves a discretized 2D heat equation problem.
 * This version uses the Krylov solver IDASpgmr.
 *
 * The DAE system solved is a spatial discretization of the PDE
 *          du/dt = d^2u/dx^2 + d^2u/dy^2
 * on the unit square. The boundary condition is u = 0 on all edges.
 * Initial conditions are given by u = 16 x (1 - x) y (1 - y). The
 * PDE is treated with central differences on a uniform M x M grid.
 * The values of u at the interior points satisfy ODEs, and
 * equations u = 0 at the boundaries are appended, to form a DAE
 * system of size N = M^2. Here M = 10.
 *
 * The system is solved with IDA/IDAS using the Krylov linear solver
 * IDASPGMR. The preconditioner uses the diagonal elements of the
 * Jacobian only. Routines for preconditioning, required by
 * IDASPGMR, are supplied here. The constraints u >= 0 are posed
 * for all components. Output is taken at t = 0, .01, .02, .04,
 * ..., 10.24. Two cases are run -- with the Gram-Schmidt type
 * being Modified in the first case, and Classical in the second.
 * The second run uses IDAReInit and IDAReInitSpgmr.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sundialstypes.h"
#include "nvector_serial.h"
#include "ida.h"
#include "idaspgmr.h"

/* Problem Constants */

#define NOUT  11
#define MGRID 10
#define NEQ   MGRID*MGRID
#define ZERO  RCONST(0.0)
#define ONE   RCONST(1.0)
#define TWO   RCONST(2.0)
#define FOUR  RCONST(4.0)

/* User data type */
```

```

typedef struct {
  long int mm;  /* number of grid points */
  realtype dx;
  realtype coeff;
  N_Vector pp;  /* vector of prec. diag. elements */
} *UserData;

/* Prototypes for functions called by IDA */

int resHeat(realtype tres, N_Vector uu, N_Vector up,
            N_Vector resval, void *rdata);

int PsetupHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               realtype c_j, void *prec_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

int PsolveHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               N_Vector rvec, N_Vector zvec,
               realtype c_j, realtype delta, void *prec_data,
               N_Vector tmp);

/* Prototypes for private functions */

static int SetInitialProfile(UserData data, N_Vector uu, N_Vector up,
                             N_Vector res);
static void PrintHeader(realtype rtol, realtype atol);
static void PrintOutput(void *mem, realtype t, N_Vector uu);
static int check_flag(void *flagvalue, char *funcname, int opt);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main()
{
  void *mem;
  UserData data;
  N_Vector uu, up, constraints, res;
  int ier, iout;
  realtype rtol, atol, t0, t1, tout, tret;
  long int netf, ncfn, ncfl;

  mem = NULL;
  data = NULL;
  uu = up = constraints = res = NULL;

  /* Allocate N-vectors and the user data structure. */

  uu = N_VNew_Serial(NEQ);
  if(check_flag((void *)uu, "N_VNew_Serial", 0)) return(1);

  up = N_VNew_Serial(NEQ);
  if(check_flag((void *)up, "N_VNew_Serial", 0)) return(1);

  res = N_VNew_Serial(NEQ);
  if(check_flag((void *)res, "N_VNew_Serial", 0)) return(1);

  constraints = N_VNew_Serial(NEQ);
  if(check_flag((void *)constraints, "N_VNew_Serial", 0)) return(1);

  data = (UserData) malloc(sizeof *data);
  data->pp = NULL;
  if(check_flag((void *)data, "malloc", 2)) return(1);

  /* Assign parameters in the user data structure. */

  data->mm  = MGRID;
  data->dx = ONE/(MGRID-ONE);
  data->coeff = ONE/(data->dx * data->dx);
  data->pp = N_VNew_Serial(NEQ);
  if(check_flag((void *)data->pp, "N_VNew_Serial", 0)) return(1);

  /* Initialize uu, up. */

  SetInitialProfile(data, uu, up, res);

  /* Set constraints to all 1's for nonnegative solution values. */

  N_VConst(ONE, constraints);

  /* Assign various parameters. */

  t0   = ZERO;
  t1   = RCONST(0.01);
  rtol = ZERO;
  atol = RCONST(1.0e-3);

  /* Call IDACreate and IDAMalloc to initialize solution */

  mem = IDACreate();
  if(check_flag((void *)mem, "IDACreate", 0)) return(1);

  ier = IDASetRdata(mem, data);
  if(check_flag(&ier, "IDASetRdata", 1)) return(1);

  ier = IDASetConstraints(mem, constraints);
  if(check_flag(&ier, "IDASetConstraints", 1)) return(1);
  N_VDestroy_Serial(constraints);

  ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
  if(check_flag(&ier, "IDAMalloc", 1)) return(1);

  /* Call IDASpgmr to specify the linear solver. */

  ier = IDASpgmr(mem, 0);
  if(check_flag(&ier, "IDASpgmr", 1)) return(1);

  ier = IDASpgmrSetPreconditioner(mem, PsetupHeat, PsolveHeat, data);
  if(check_flag(&ier, "IDASpgmrSetPreconditioner", 1)) return(1);

  /* Print output heading. */
  PrintHeader(rtol, atol);

  /*
   * -------------------------------------------------------------------------
   * CASE I
   * -------------------------------------------------------------------------
   */

  /* Print case number, output table heading, and initial line of table. */

  printf("\n\nCase 1: gsytpe = MODIFIED_GS\n");
  printf("\n   Output Summary (umax = max-norm of solution) \n\n");
  printf("  time     umax       k  nst  nni  nje   nre   nreS     h      npe nps\n" );
  printf("----------------------------------------------------------------------\n");

  /* Loop over output times, call IDASolve, and print results. */

  for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
    ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
    if(check_flag(&ier, "IDASolve", 1)) return(1);
    PrintOutput(mem, tret, uu);
  }

  /* Print remaining counters. */

  ier = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&ier, "IDAGetNumErrTestFails", 1);

  ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);

  ier = IDASpgmrGetNumConvFails(mem, &ncfl);
  check_flag(&ier, "IDASpgmrGetNumConvFails", 1);

  printf("\nError test failures         = %ld\n", netf);
  printf("Nonlinear convergence failures = %ld\n", ncfn);
  printf("Linear convergence failures    = %ld\n", ncfl);

  /*
   * -------------------------------------------------------------------------
   * CASE II
   * -------------------------------------------------------------------------
   */

  /* Re-initialize uu, up. */

  SetInitialProfile(data, uu, up, res);

  /* Re-initialize IDA and IDASPGMR */

  ier = IDAReInit(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
  if(check_flag(&ier, "IDAReInit", 1)) return(1);

  ier = IDASpgmrSetGSType(mem, CLASSICAL_GS);
  if(check_flag(&ier, "IDASpgmrSetGSType",1)) return(1);

  /* Print case number, output table heading, and initial line of table. */

  printf("\n\nCase 2: gstype = CLASSICAL_GS\n");
  printf("\n   Output Summary (umax = max-norm of solution) \n\n");
  printf("  time     umax       k  nst  nni  nje   nre   nreS     h      npe nps\n" );
  printf("----------------------------------------------------------------------\n");

  /* Loop over output times, call IDASolve, and print results. */

  for (tout = t1,iout = 1; iout <= NOUT ; iout++, tout *= TWO) {
    ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
    if(check_flag(&ier, "IDASolve", 1)) return(1);
    PrintOutput(mem, tret, uu);
  }

  /* Print remaining counters. */

  ier = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&ier, "IDAGetNumErrTestFails", 1);

  ier = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&ier, "IDAGetNumNonlinSolvConvFails", 1);

  ier = IDASpgmrGetNumConvFails(mem, &ncfl);
  check_flag(&ier, "IDASpgmrGetNumConvFails", 1);

  printf("\nError test failures            = %ld\n", netf);
  printf("Nonlinear convergence failures = %ld\n", ncfn);
  printf("Linear convergence failures    = %ld\n", ncfl);

  /* Free Memory */

  IDAFree(mem);

  N_VDestroy_Serial(uu);
  N_VDestroy_Serial(up);
  N_VDestroy_Serial(res);

  N_VDestroy_Serial(data->pp);
  free(data);

  return(0);
}

```

```
/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY IDA
 *--------------------------------------------------------------------
 */

/*
 * resHeat: heat equation system residual function (user-supplied)
 * This uses 5-point central differencing on the interior points, and
 * includes algebraic equations for the boundary values.
 * So for each interior point, the residual component has the form
 *    res_i = u'_i - (central difference)_i
 * while for each boundary point, it is res_i = u_i.
 */

int resHeat(realtype tt,
            N_Vector uu, N_Vector up, N_Vector rr,
            void *res_data)
{
  long int i, j, offset, loc, mm;
  realtype *uu_data, *up_data, *rr_data, coeff, dif1, dif2;
  UserData data;

  uu_data = NV_DATA_S(uu);
  up_data = NV_DATA_S(up);
  rr_data = NV_DATA_S(rr);

  data = (UserData) res_data;

  coeff = data->coeff;
  mm    = data->mm;

  /* Initialize rr to uu, to take care of boundary equations. */
  N_VScale(ONE, uu, rr);

  /* Loop over interior points; set res = up - (central difference). */
  for (j = 1; j < MGRID-1; j++) {
    offset = mm*j;
    for (i = 1; i < mm-1; i++) {
      loc = offset + i;
      dif1 = uu_data[loc-1]  + uu_data[loc+1]  - TWO * uu_data[loc];
      dif2 = uu_data[loc-mm] + uu_data[loc+mm] - TWO * uu_data[loc];
      rr_data[loc]= up_data[loc] - coeff * ( dif1 + dif2 );
    }
  }

  return(0);
}

/*
 * PsetupHeat: setup for diagonal preconditioner for iheatsk.
 *
 * The optional user-supplied functions PsetupHeat and
 * PsolveHeat together must define the left preconditoner
 * matrix P approximating the system Jacobian matrix
 *                  J = dF/du + cj*dF/du'
 * (where the DAE system is F(t,u,u') = 0), and solve the linear
 * systems P z = r.   This is done in this case by keeping only
 * the diagonal elements of the J matrix above, storing them as
 * inverses in a vector pp, when computed in PsetupHeat, for
 * subsequent use in PsolveHeat.
 *
 * In this instance, only cj and data (user data structure, with
 * pp etc.) are used from the PsetupdHeat argument list.
 */

int PsetupHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               realtype c_j, void *prec_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{

  long int i, j, offset, loc, mm;
  realtype *ppv, pelinv;
  UserData data;

  data = (UserData) prec_data;
  ppv = NV_DATA_S(data->pp);
  mm = data->mm;

  /* Initialize the entire vector to 1., then set the interior points to the
     correct value for preconditioning. */
  N_VConst(ONE,data->pp);

  /* Compute the inverse of the preconditioner diagonal elements. */
  pelinv = ONE/(c_j + FOUR*data->coeff);

  for (j = 1; j < mm-1; j++) {
    offset = mm * j;
    for (i = 1; i < mm-1; i++) {
      loc = offset + i;
      ppv[loc] = pelinv;
    }
  }

  return(0);
}

/*
 * PsolveHeat: solve preconditioner linear system.
 * This routine multiplies the input vector rvec by the vector pp
 * containing the inverse diagonal Jacobian elements (previously
 * computed in PrecondHeateq), returning the result in zvec.
 */

int PsolveHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               N_Vector rvec, N_Vector zvec,
               realtype c_j, realtype delta, void *prec_data,
               N_Vector tmp)
{
  UserData data;
  data = (UserData) prec_data;
  N_VProd(data->pp, rvec, zvec);
  return(0);
}

/*
 *--------------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *--------------------------------------------------------------------
 */

/*
 * SetInitialProfile: routine to initialize u and up vectors.
 */

static int SetInitialProfile(UserData data, N_Vector uu, N_Vector up,
                             N_Vector res)
{
  long int mm, mm1, i, j, offset, loc;
  realtype xfact, yfact, *udata, *updata;

  mm = data->mm;

  udata = NV_DATA_S(uu);
  updata = NV_DATA_S(up);

  /* Initialize uu on all grid points. */
  mm1 = mm - 1;
  for (j = 0; j < mm; j++) {
    yfact = data->dx * j;
    offset = mm*j;
    for (i = 0;i < mm; i++) {
      xfact = data->dx * i;
      loc = offset + i;
      udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
    }
  }

  /* Initialize up vector to 0. */
  N_VConst(ZERO, up);

  /* resHeat sets res to negative of ODE RHS values at interior points. */
  resHeat(ZERO, uu, up, res, data);

  /* Copy -res into up to get correct interior initial up values. */
  N_VScale(-ONE, res, up);

  /* Set up at boundary points to zero. */
  for (j = 0; j < mm; j++) {
    offset = mm*j;
    for (i = 0; i < mm; i++) {
      loc = offset + i;
      if (j == 0 || j == mm1 || i == 0 || i == mm1 ) updata[loc] = ZERO;
    }
  }

  return(0);
  }

/*
 * Print first lines of output (problem description)
 */

static void PrintHeader(realtype rtol, realtype atol)
{
  printf("\niheatsk: Heat equation, serial example problem for IDA \n");
  printf("         Discretized heat equation on 2D unit square. \n");
  printf("         Zero boundary conditions,");
  printf(" polynomial initial conditions.\n");
  printf("         Mesh dimensions: %d x %d", MGRID, MGRID);
  printf("        Total system size: %d\n\n", NEQ);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
#endif
  printf("Constraints set to force all solution components >= 0. \n");
  printf("Linear solver: IDASPGMR, preconditioner using diagonal elements. \n");
}

/*
 * PrintOutput: print max norm of solution and current solver statistics
 */

static void PrintOutput(void *mem, realtype t, N_Vector uu)
{
  realtype hused, umax;
  long int nst, nni, nje, nre, nreS, nli, npe, nps;
  int kused, ier;

  umax = N_VMaxNorm(uu);

  ier = IDAGetLastOrder(mem, &kused);
  check_flag(&ier, "IDAGetLastOrder", 1);
  ier = IDAGetNumSteps(mem, &nst);
  check_flag(&ier, "IDAGetNumSteps", 1);
  ier = IDAGetNumNonlinSolvIters(mem, &nni);
  check_flag(&ier, "IDAGetNumNonlinSolvIters", 1);
  ier = IDAGetNumResEvals(mem, &nre);
  check_flag(&ier, "IDAGetNumResEvals", 1);
  ier = IDAGetLastStep(mem, &hused);
  check_flag(&ier, "IDAGetLastStep", 1);
  ier = IDASpgmrGetNumJtimesEvals(mem, &nje);
  check_flag(&ier, "IDASpgmrGetNumJtimesEvals", 1);
  ier = IDASpgmrGetNumLinIters(mem, &nli);
  check_flag(&ier, "IDASpgmrGetNumLinIters", 1);
  ier = IDASpgmrGetNumResEvals(mem, &nreS);
  check_flag(&ier, "IDASpgmrGetNumResEvals", 1);
  ier = IDASpgmrGetNumPrecEvals(mem, &npe);
  check_flag(&ier, "IDASpgmrGetPrecEvals", 1);
  ier = IDASpgmrGetNumPrecSolves(mem, &nps);
  check_flag(&ier, "IDASpgmrGetNumPrecSolves", 1);

#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf(" %5.2Lf %13.5Le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2Le  %3ld %3ld\n",
         t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf(" %5.2f %13.5le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2le  %3ld %3ld\n",
         t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
#else
  printf(" %5.2f %13.5e  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2e  %3ld %3ld\n",
         t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
#endif
}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt)
{
  int *errflag;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && flagvalue == NULL) {
    fprintf(stderr,
            "\nSUNDIALS_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1);
  } else if (opt == 1) {
    /* Check if flag < 0 */
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR: %s() failed with flag = %d\n\n",
              funcname, *errflag);
      return(1);
    }
  } else if (opt == 2 && flagvalue == NULL) {
    /* Check if function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nMEMORY_ERROR: %s() failed - returned NULL pointer\n\n",
            funcname);
    return(1);
  }

  return(0);
}
```

D Listing of `iheatpk.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.15.2.3 $
 * $Date: 2005/04/06 23:34:20 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example problem for IDA: 2D heat equation, parallel, GMRES.
 *
 * This example solves a discretized 2D heat equation problem.
 * This version uses the Krylov solver IDASpgmr.
 *
 * The DAE system solved is a spatial discretization of the PDE
 *          du/dt = d^2u/dx^2 + d^2u/dy^2
 * on the unit square. The boundary condition is u = 0 on all edges.
 * Initial conditions are given by u = 16 x (1 - x) y (1 - y).
 * The PDE is treated with central differences on a uniform MX x MY
 * grid. The values of u at the interior points satisfy ODEs, and
 * equations u = 0 at the boundaries are appended, to form a DAE
 * system of size N = MX * MY. Here MX = MY = 10.
 *
 * The system is actually implemented on submeshes, processor by
 * processor, with an MXSUB by MYSUB mesh on each of NPEX * NPEY
 * processors.
 *
 * The system is solved with IDA using the Krylov linear solver
 * IDASPGMR. The preconditioner uses the diagonal elements of the
 * Jacobian only. Routines for preconditioning, required by
 * IDASPGMR, are supplied here. The constraints u >= 0 are posed
 * for all components. Local error testing on the boundary values
 * is suppressed. Output is taken at t = 0, .01, .02, .04,
 * ..., 10.24.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sundialstypes.h"
#include "sundialsmath.h"
#include "nvector_parallel.h"
#include "ida.h"
#include "idaspgmr.h"
#include "iterative.h"
#include "mpi.h"

#define ZERO  RCONST(0.0)
#define ONE   RCONST(1.0)
#define TWO   RCONST(2.0)

#define NOUT         11             /* Number of output times */
```

```

#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 */

typedef struct {
  long int thispe, mx, my, ixsub, jysub, npex, npey, mxsub, mysub;
  realtype    dx, dy, coeffx, coeffy, coeffxy;
  realtype    uext[(MXSUB+2)*(MYSUB+2)];
  N_Vector    pp;    /* vector of diagonal preconditioner elements */
  MPI_Comm    comm;
} *UserData;

/* User-supplied residual function and supporting routines */

int resHeat(realtype tt,
            N_Vector uu, N_Vector up, N_Vector rr,
            void *res_data);

static int rescomm(N_Vector uu, N_Vector up, void *res_data);

static int reslocal(realtype tt, N_Vector uu, N_Vector up,
                    N_Vector res,  void *res_data);

static int BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
                 long int dsizex, long int dsizey, realtype uarray[]);

static int BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
                     long int ixsub, long int jysub,
                     long int dsizex, long int dsizey,
                     realtype uext[], realtype buffer[]);

static int BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
                     long int dsizex, realtype uext[], realtype buffer[]);

/* User-supplied preconditioner routines */

int PsolveHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               N_Vector rvec, N_Vector zvec,
               realtype c_j, realtype delta, void *prec_data,
               N_Vector tmp);

int PsetupHeat(realtype tt,
               N_Vector yy, N_Vector yp, N_Vector rr,
               realtype c_j, void *prec_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3);

```

```
/* Private function to check function return values */

static int InitUserData(int thispe, MPI_Comm comm, UserData data);

static int SetInitialProfile(N_Vector uu, N_Vector up, N_Vector id,
                             N_Vector res, UserData data);

static void PrintHeader(long int Neq, realtype rtol, realtype atol);

static void PrintOutput(int id, void *mem, realtype t, N_Vector uu);

static void PrintFinalStats(void *mem);

static int check_flag(void *flagvalue, char *funcname, int opt, int id);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  MPI_Comm comm;
  void *mem;
  UserData data;
  int iout, thispe, ier, npes;
  long int Neq, local_N;
  realtype rtol, atol, t0, t1, tout, tret;
  N_Vector uu, up, constraints, id, res;

  mem = NULL;
  data = NULL;
  uu = up = constraints = id = res = NULL;

  /* 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, &thispe);

  if (npes != NPEX*NPEY) {
    if (thispe == 0)
      fprintf(stderr,
              "\nMPI_ERROR(0): npes = %d is not equal to NPEX*NPEY = %d\n",
              npes,NPEX*NPEY);
    MPI_Finalize();
    return(1);
  }

  /* Set local length local_N and global length Neq. */

  local_N = MXSUB*MYSUB;
  Neq     = MX * MY;

  /* Allocate and initialize the data structure and N-vectors. */

  data = (UserData) malloc(sizeof *data);
  data->pp = NULL;
  if(check_flag((void *)data, "malloc", 2, thispe))
    MPI_Abort(comm, 1);

  uu = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)uu, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  up = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)up, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  res = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)res, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  constraints = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)constraints, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  id = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)id, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  /* An N-vector to hold preconditioner. */
  data->pp = N_VNew_Parallel(comm, local_N, Neq);
  if(check_flag((void *)data->pp, "N_VNew_Parallel", 0, thispe))
    MPI_Abort(comm, 1);

  InitUserData(thispe, comm, data);

  /* Initialize the uu, up, id, and res profiles. */

  SetInitialProfile(uu, up, id, res, data);

  /* Set constraints to all 1's for nonnegative solution values. */

  N_VConst(ONE, constraints);

  t0 = ZERO; t1 = RCONST(0.01);

  /* Scalar relative and absolute tolerance. */

  rtol = ZERO;
  atol = RCONST(1.0e-3);

  /* Call IDACreate and IDAMalloc to initialize solution. */

  mem = IDACreate();
  if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);

  ier = IDASetRdata(mem, data);
  if(check_flag(&ier, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);

  ier = IDASetSuppressAlg(mem, TRUE);
  if(check_flag(&ier, "IDASetSuppressAlg", 1, thispe)) MPI_Abort(comm, 1);

  ier = IDASetId(mem, id);
  if(check_flag(&ier, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);

  ier = IDASetConstraints(mem, constraints);
  if(check_flag(&ier, "IDASetConstraints", 1, thispe)) MPI_Abort(comm, 1);
  N_VDestroy_Parallel(constraints);

  ier = IDAMalloc(mem, resHeat, t0, uu, up, IDA_SS, rtol, &atol);
  if(check_flag(&ier, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);

  /* Call IDASpgmr to specify the linear solver. */

  ier = IDASpgmr(mem, 0);
  if(check_flag(&ier, "IDASpgmr", 1, thispe)) MPI_Abort(comm, 1);

  ier = IDASpgmrSetPreconditioner(mem, PsetupHeat, PsolveHeat, data);
  if(check_flag(&ier, "IDASpgmrSetPreconditioner", 1, thispe)) MPI_Abort(comm, 1);

  /* Print output heading (on processor 0 only) and intial solution  */

  if (thispe == 0) PrintHeader(Neq, rtol, atol);
  PrintOutput(thispe, mem, t0, uu);

  /* Loop over tout, call IDASolve, print output. */

  for (tout = t1, iout = 1; iout <= NOUT; iout++, tout *= TWO) {

    ier = IDASolve(mem, tout, &tret, uu, up, IDA_NORMAL);
    if(check_flag(&ier, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);

    PrintOutput(thispe, mem, tret, uu);

  }

  /* Print remaining counters. */

  if (thispe == 0) PrintFinalStats(mem);

  /* Free memory */

  IDAFree(mem);

  N_VDestroy_Parallel(id);
  N_VDestroy_Parallel(res);
  N_VDestroy_Parallel(up);
  N_VDestroy_Parallel(uu);

  N_VDestroy_Parallel(data->pp);
  free(data);

  MPI_Finalize();

  return(0);

}

/*
 *--------------------------------------------------------------------
 * FUNCTIONS CALLED BY IDA
 *--------------------------------------------------------------------
 */

/*
 * resHeat: heat equation system residual function
 * This uses 5-point central differencing on the interior points, and
 * includes algebraic equations for the boundary values.
 * So for each interior point, the residual component has the form
 *    res_i = u'_i - (central difference)_i
 * while for each boundary point, it is res_i = u_i.
 *
 * This parallel implementation uses several supporting routines.
 * First a call is made to rescomm to do communication of subgrid boundary
 * data into array uext.  Then reslocal is called to compute the residual
 * on individual processors and their corresponding domains.  The routines
 * BSend, BRecvPost, and BREcvWait handle interprocessor communication
 * of uu required to calculate the residual.
 */

int resHeat(realtype tt,
            N_Vector uu, N_Vector up, N_Vector rr,
            void *res_data)
{
  int retval;

  /* Call rescomm to do inter-processor communication. */
  retval = rescomm(uu, up, res_data);

  /* Call reslocal to calculate res. */
  retval = reslocal(tt, uu, up, rr, res_data);

  return(0);

}

/*
 * PsetupHeat: setup for diagonal preconditioner for heatsk.
 *
 * The optional user-supplied functions PsetupHeat and
 * PsolveHeat together must define the left preconditoner
 * matrix P approximating the system Jacobian matrix
 *                    J = dF/du + cj*dF/du'
 * (where the DAE system is F(t,u,u') = 0), and solve the linear
 * systems P z = r.   This is done in this case by keeping only
 * the diagonal elements of the J matrix above, storing them as
 * inverses in a vector pp, when computed in PsetupHeat, for
 * subsequent use in PsolveHeat.
 *
 * In this instance, only cj and data (user data structure, with
 * pp etc.) are used from the PsetupHeat argument list.
 *
 */

int PsetupHeat(realtype tt,
               N_Vector yy, N_Vector yp, N_Vector rr,
               realtype c_j, void *prec_data,
               N_Vector tmp1, N_Vector tmp2, N_Vector tmp3)
{
  realtype *ppv, pelinv;
  long int lx, ly, ixbegin, ixend, jybegin, jyend, locu, mxsub, mysub;
  long int ixsub, jysub, npex, npey;
  UserData data;

  data = (UserData) prec_data;

  ppv = NV_DATA_P(data->pp);
  ixsub = data->ixsub;
  jysub = data->jysub;
  mxsub = data->mxsub;
  mysub = data->mysub;
  npex  = data->npex;
  npey  = data->npey;

  /* Initially set all pp elements to one. */
  N_VConst(ONE, data->pp);

  /* Prepare to loop over subgrid. */
  ixbegin = 0;
  ixend   = mxsub-1;
  jybegin = 0;
  jyend   = mysub-1;
  if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
  if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;
  pelinv = ONE/(c_j + data->coeffxy);

  /* Load the inverse of the preconditioner diagonal elements
     in loop over all the local subgrid. */

  for (ly = jybegin; ly <=jyend; ly++) {
    for (lx = ixbegin; lx <= ixend; lx++) {
      locu  = lx + ly*mxsub;
      ppv[locu] = pelinv;
    }
  }

  return(0);

}

/*
 * PsolveHeat: solve preconditioner linear system.
 * This routine multiplies the input vector rvec by the vector pp
 * containing the inverse diagonal Jacobian elements (previously
 * computed in PsetupHeat), returning the result in zvec.
 */

int PsolveHeat(realtype tt,
               N_Vector uu, N_Vector up, N_Vector rr,
               N_Vector rvec, N_Vector zvec,
               realtype c_j, realtype delta, void *prec_data,
               N_Vector tmp)
{
  UserData data;

  data = (UserData) prec_data;

  N_VProd(data->pp, rvec, zvec);

  return(0);

}

/*
 *--------------------------------------------------------------------
 * SUPPORTING FUNCTIONS
 *--------------------------------------------------------------------
 */


/*
 * rescomm routine.  This routine performs all inter-processor
 * communication of data in u needed to calculate G.
 */

static int rescomm(N_Vector uu, N_Vector up, void *res_data)
{
  UserData data;
  realtype *uarray, *uext, buffer[2*MYSUB];
  MPI_Comm comm;
  long int thispe, ixsub, jysub, mxsub, mysub;
  MPI_Request request[4];

  data = (UserData) res_data;
  uarray = NV_DATA_P(uu);

  /* Get comm, thispe, subgrid indices, data sizes, extended array uext. */
  comm = data->comm;  thispe = data->thispe;
  ixsub = data->ixsub;   jysub = data->jysub;
  mxsub = data->mxsub;   mysub = data->mysub;
  uext = data->uext;

  /* Start receiving boundary data from neighboring PEs. */
  BRecvPost(comm, request, thispe, ixsub, jysub, mxsub, mysub, uext, buffer);

  /* Send data from boundary of local grid to neighboring PEs. */
  BSend(comm, thispe, ixsub, jysub, mxsub, mysub, uarray);

  /* Finish receiving boundary data from neighboring PEs. */
  BRecvWait(request, ixsub, jysub, mxsub, uext, buffer);

  return(0);

}

/*
 * reslocal routine.  Compute res = F(t, uu, up).  This routine assumes
 * that all inter-processor communication of data needed to calculate F
 * has already been done, and that this data is in the work array uext.
 */

static int reslocal(realtype tt,
                    N_Vector uu, N_Vector up, N_Vector rr,
                    void *res_data)
{
  realtype *uext, *uuv, *upv, *resv;
  realtype termx, termy, termctr;
  long int lx, ly, offsetu, offsetue, locu, locue;
  long int ixsub, jysub, mxsub, mxsub2, mysub, npex, npey;
  long int ixbegin, ixend, jybegin, jyend;
  UserData data;

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

  data = (UserData) res_data;
  uext = data->uext;
  uuv = NV_DATA_P(uu);
  upv = NV_DATA_P(up);
  resv = NV_DATA_P(rr);
  ixsub = data->ixsub; jysub = data->jysub;
  mxsub = data->mxsub; mxsub2 = data->mxsub + 2;
  mysub = data->mysub; npex = data->npex; npey = data->npey;

  /* Initialize all elements of rr to uu. This sets the boundary
     elements simply without indexing hassles. */

  N_VScale(ONE, uu, rr);

  /* Copy local segment of u vector into the working extended array uext.
     This completes uext prior to the computation of the rr vector.     */

  offsetu = 0;
  offsetue = mxsub2 + 1;
  for (ly = 0; ly < mysub; ly++) {
    for (lx = 0; lx < mxsub; lx++) uext[offsetue+lx] = uuv[offsetu+lx];
    offsetu = offsetu + mxsub;
    offsetue = offsetue + mxsub2;
  }

  /* Set loop limits for the interior of the local subgrid. */

  ixbegin = 0;
  ixend   = mxsub-1;
  jybegin = 0;
  jyend   = mysub-1;
  if (ixsub == 0) ixbegin++; if (ixsub == npex-1) ixend--;
  if (jysub == 0) jybegin++; if (jysub == npey-1) jyend--;

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

  for (ly = jybegin; ly <=jyend; ly++) {
    for (lx = ixbegin; lx <= ixend; lx++) {
      locu  = lx + ly*mxsub;
      locue = (lx+1) + (ly+1)*mxsub2;
      termx = data->coeffx *(uext[locue-1]      + uext[locue+1]);
      termy = data->coeffy *(uext[locue-mxsub2] + uext[locue+mxsub2]);
      termctr = data->coeffxy*uext[locue];
      resv[locu] = upv[locu] - (termx + termy - termctr);
   }
  }
  return(0);

}

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

static int BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
                 long int dsizex, long int dsizey, realtype uarray[])
{
  long int ly, offsetu;
  realtype bufleft[MYSUB], bufright[MYSUB];

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

  if (jysub != 0)
    MPI_Send(&uarray[0], dsizex, PVEC_REAL_MPI_TYPE, thispe-NPEX, 0, comm);

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

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

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

  if (ixsub != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = ly*dsizex;
      bufleft[ly] = uarray[offsetu];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, thispe-1, 0, comm);
  }

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

  if (ixsub != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetu = ly*MXSUB + (MXSUB-1);
      bufright[ly] = uarray[offsetu];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, thispe+1, 0, comm);
  }

  return(0);

}

/*
 * Routine to start receiving boundary data from neighboring PEs.
 * Notes:
 *   1) buffer should be able to hold 2*MYSUB realtype 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 int BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
                     long int ixsub, long int jysub,
                     long int dsizex, long int dsizey,
                     realtype uext[], realtype buffer[])
{
  long int offsetue;
  /* Have bufleft and bufright use the same buffer. */
  realtype *bufleft = buffer, *bufright = buffer+MYSUB;

  /* If jysub > 0, receive data for bottom x-line of uext. */
  if (jysub != 0)
    MPI_Irecv(&uext[1], dsizex, PVEC_REAL_MPI_TYPE,
              thispe-NPEX, 0, comm, &request[0]);

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

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

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

  return(0);

}

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

static int BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
                     long int dsizex, realtype uext[], realtype buffer[])
{
  long int ly, dsizex2, offsetue;
  realtype *bufleft = buffer, *bufright = buffer+MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2;

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

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

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

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

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

    /* Copy the buffer to uext */
    for (ly = 0; ly < MYSUB; ly++) {
      offsetue = (ly+2)*dsizex2 - 1;
      uext[offsetue] = bufright[ly];
    }
  }

  return(0);

}

/*
 *--------------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *--------------------------------------------------------------------
 */

/*
 * InitUserData initializes the user's data block data.
 */

static int InitUserData(int thispe, MPI_Comm comm, UserData data)
{
  data->thispe = thispe;
  data->dx = ONE/(MX-ONE);       /* Assumes a [0,1] interval in x. */
  data->dy = ONE/(MY-ONE);       /* Assumes a [0,1] interval in y. */
  data->coeffx  = ONE/(data->dx * data->dx);
  data->coeffy  = ONE/(data->dy * data->dy);
  data->coeffxy = TWO/(data->dx * data->dx) + TWO/(data->dy * data->dy) ;
  data->jysub   = thispe/NPEX;
  data->ixsub   = thispe - data->jysub * NPEX;
  data->npex    = NPEX;
  data->npey    = NPEY;
  data->mx      = MX;
  data->my      = MY;
  data->mxsub = MXSUB;
  data->mysub = MYSUB;
  data->comm    = comm;
  return(0);

}

/*
 * SetInitialProfile sets the initial values for the problem.
 */

static int SetInitialProfile(N_Vector uu, N_Vector up,  N_Vector id,
                             N_Vector res, UserData data)
{
  long int i, iloc, j, jloc, offset, loc, ixsub, jysub;
  long int ixbegin, ixend, jybegin, jyend;
  realtype xfact, yfact, *udata, *iddata, dx, dy;

  /* Initialize uu. */

  udata = NV_DATA_P(uu);
  iddata = NV_DATA_P(id);

  /* Set mesh spacings and subgrid indices for this PE. */
  dx = data->dx;
  dy = data->dy;
  ixsub = data->ixsub;
  jysub = data->jysub;

  /* Set beginning and ending locations in the global array corresponding
     to the portion of that array assigned to this processor. */
  ixbegin = MXSUB*ixsub;
  ixend   = MXSUB*(ixsub+1) - 1;
  jybegin = MYSUB*jysub;
  jyend   = MYSUB*(jysub+1) - 1;

  /* Loop over the local array, computing the initial profile value.
     The global indices are (i,j) and the local indices are (iloc,jloc).
     Also set the id vector to zero for boundary points, one otherwise. */

  N_VConst(ONE,id);
  for (j = jybegin, jloc = 0; j <= jyend; j++, jloc++) {
    yfact = data->dy*j;
    offset= jloc*MXSUB;
    for (i = ixbegin, iloc = 0; i <= ixend; i++, iloc++) {
      xfact = data->dx * i;
      loc = offset + iloc;
      udata[loc] = RCONST(16.0) * xfact * (ONE - xfact) * yfact * (ONE - yfact);
      if (i == 0 || i == MX-1 || j == 0 || j == MY-1) iddata[loc] = ZERO;
    }
  }

  /* Initialize up. */

  N_VConst(ZERO, up);    /* Initially set up = 0. */

  /* resHeat sets res to negative of ODE RHS values at interior points. */
  resHeat(ZERO, uu, up, res, data);

  /* Copy -res into up to get correct initial up values. */
  N_VScale(-ONE, res, up);

  return(0);
}

/*
 * Print first lines of output and table heading
 */

```

```
static void PrintHeader(long int Neq, realtype rtol, realtype atol)
{
  printf("\niheatpk: Heat equation, parallel example problem for IDA \n");
  printf("         Discretized heat equation on 2D unit square. \n");
  printf("         Zero boundary conditions,");
  printf(" polynomial initial conditions.\n");
  printf("         Mesh dimensions: %d x %d", MX, MY);
  printf("        Total system size: %ld\n\n", Neq);
  printf("Subgrid dimensions: %d x %d", MXSUB, MYSUB);
  printf("        Processor array: %d x %d\n", NPEX, NPEY);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
#endif
  printf("Constraints set to force all solution components >= 0. \n");
  printf("SUPPRESSALG = TRUE to suppress local error testing on ");
  printf("all boundary components. \n");
  printf("Linear solver: IDASPGMR  ");
  printf("Preconditioner: diagonal elements only.\n");

  /* Print output table heading and initial line of table. */
  printf("\n   Output Summary (umax = max-norm of solution) \n\n");
  printf("  time     umax       k  nst  nni  nli   nre   nreS     h      npe nps\n");
  printf("----------------------------------------------------------------------\n");
}

/*
 * PrintOutput: print max norm of solution and current solver statistics
 */

static void PrintOutput(int id, void *mem, realtype t, N_Vector uu)
{
  realtype hused, umax;
  long int nst, nni, nje, nre, nreS, nli, npe, nps;
  int kused, ier;

  umax = N_VMaxNorm(uu);

  if (id == 0) {

    ier = IDAGetLastOrder(mem, &kused);
    check_flag(&ier, "IDAGetLastOrder", 1, id);
    ier = IDAGetNumSteps(mem, &nst);
    check_flag(&ier, "IDAGetNumSteps", 1, id);
    ier = IDAGetNumNonlinSolvIters(mem, &nni);
    check_flag(&ier, "IDAGetNumNonlinSolvIters", 1, id);
    ier = IDAGetNumResEvals(mem, &nre);
    check_flag(&ier, "IDAGetNumResEvals", 1, id);
    ier = IDAGetLastStep(mem, &hused);
    check_flag(&ier, "IDAGetLastStep", 1, id);
    ier = IDASpgmrGetNumJtimesEvals(mem, &nje);
    check_flag(&ier, "IDASpgmrGetNumJtimesEvals", 1, id);
    ier = IDASpgmrGetNumLinIters(mem, &nli);
    check_flag(&ier, "IDASpgmrGetNumLinIters", 1, id);
    ier = IDASpgmrGetNumResEvals(mem, &nreS);
    check_flag(&ier, "IDASpgmrGetNumResEvals", 1, id);
    ier = IDASpgmrGetNumPrecEvals(mem, &npe);
    check_flag(&ier, "IDASpgmrGetPrecEvals", 1, id);
    ier = IDASpgmrGetNumPrecSolves(mem, &nps);
    check_flag(&ier, "IDASpgmrGetNumPrecSolves", 1, id);

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf(" %5.2Lf %13.5Le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2Le  %3ld %3ld\n",
           t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf(" %5.2f %13.5le  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2le  %3ld %3ld\n",
           t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
#else
    printf(" %5.2f %13.5e  %d  %3ld  %3ld  %3ld  %4ld  %4ld  %9.2e  %3ld %3ld\n",
           t, umax, kused, nst, nni, nje, nre, nreS, hused, npe, nps);
#endif

  }
}

/*
 * Print some final integrator statistics
 */

static void PrintFinalStats(void *mem)
{
  long int netf, ncfn, ncfl;

  IDAGetNumErrTestFails(mem, &netf);
  IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  IDASpgmrGetNumConvFails(mem, &ncfl);

  printf("\nError test failures            = %ld\n", netf);
  printf("Nonlinear convergence failures = %ld\n", ncfn);
  printf("Linear convergence failures    = %ld\n", ncfl);
}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

  if (opt == 0 && flagvalue == NULL) {
    /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1);
  } else if (opt == 1) {
    /* Check if flag < 0 */
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
              id, funcname, *errflag);
      return(1);
    }
  } else if (opt == 2 && flagvalue == NULL) {
    /* Check if function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1);
  }

  return(0);
}
```

E Listing of `iwebbbd.c`

```
/*
 * -----------------------------------------------------------------
 * $Revision: 1.21.2.2 $
 * $Date: 2005/04/04 22:36:42 $
 * -----------------------------------------------------------------
 * Programmer(s): Allan Taylor, Alan Hindmarsh and
 *                Radu Serban @ LLNL
 * -----------------------------------------------------------------
 * Example program for IDA: Food web, parallel, GMRES, IDABBD
 * preconditioner.
 *
 * This example program for IDA uses IDASPGMR as the linear solver.
 * It is written for a parallel computer system and uses the
 * IDABBDPRE band-block-diagonal preconditioner module for the
 * IDASPGMR package. It was originally run on a Sun SPARC cluster
 * and used MPICH.
 *
 * The mathematical problem solved in this example is a DAE system
 * that arises from a system of partial differential equations after
 * spatial discretization. The PDE system is a food web population
 * model, with predator-prey interaction and diffusion on the unit
 * square in two dimensions. The dependent variable vector is:
 *
 *         1   2         ns
 *   c = (c , c ,  ..., c  ) , ns = 2 * np
 *
 * and the PDE's are as follows:
 *
 *     i             i      i
 *   dc /dt = d(i)*(c    + c  )  +  R (x,y,c)   (i = 1,...,np)
 *                   xx     yy       i
 *
 *              i      i
 *   0 = d(i)*(c    + c  )  +  R  (x,y,c)   (i = np+1,...,ns)
 *              xx     yy       i
 *
 *   where the reaction terms R are:
 *
 *                   i             ns         j
 *   R  (x,y,c)  =  c  * (b(i)  + sum a(i,j)*c )
 *    i                           j=1
 *
 * The number of species is ns = 2 * np, with the first np being
 * prey and the last np being predators. The coefficients a(i,j),
 * b(i), d(i) are:
 *
 *   a(i,i) = -AA  (all i)
 *   a(i,j) = -GG  (i <= np , j >  np)
 *   a(i,j) =  EE  (i >  np,  j <= np)
 *   all other a(i,j) = 0
 *   b(i) = BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y))  (i <= np)
 *   b(i) =-BB*(1+ alpha * x*y + beta*sin(4 pi x)*sin(4 pi y))  (i  > np)
 *   d(i) = DPREY  (i <= np)
 *   d(i) = DPRED  (i > np)
 *
 * Note: The above equations are written in 1-based indices,
 * whereas the code has 0-based indices, being written in C.
 *
 * The various scalar parameters required are set using '#define'
 * statements or directly in routine InitUserData. In this program,
 * np = 1, ns = 2. The boundary conditions are homogeneous Neumann:
 * normal derivative  =  0.
 *
 * A polynomial in x and y is used to set the initial values of the
 * first np variables (the prey variables) at each x,y location,
 * while initial values for the remaining (predator) variables are
 * set to a flat value, which is corrected by IDACalcIC.
 *
 * The PDEs are discretized by central differencing on a MX by MY
 * mesh, and so the system size Neq is the product
 * MX * MY * NUM_SPECIES. The system is actually implemented on
 * submeshes, processor by processor, with an MXSUB by MYSUB mesh
 * on each of NPEX * NPEY processors.
 *
 * The DAE system is solved by IDA using the IDASPGMR linear solver,
 * in conjunction with the preconditioner module IDABBDPRE. The
 * preconditioner uses a 5-diagonal band-block-diagonal
 * approximation (half-bandwidths = 2). Output is printed at
 * t = 0, .001, .01, .1, .4, .7, 1.
 * -----------------------------------------------------------------
 * References:
 * [1] Peter N. Brown and Alan C. Hindmarsh,
 *     Reduced Storage Matrix Methods in Stiff ODE systems,
 *     Journal of Applied Mathematics and Computation, Vol. 31
 *     (May 1989), pp. 40-91.
 *
 * [2] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Using Krylov Methods in the Solution of Large-Scale
 *     Differential-Algebraic Systems, SIAM J. Sci. Comput., 15
 *     (1994), pp. 1467-1488.
 *
 * [3] Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
 *     Consistent Initial Condition Calculation for Differential-
 *     Algebraic Systems, SIAM J. Sci. Comput., 19 (1998),
 *     pp. 1495-1512.
 * -----------------------------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sundialstypes.h"    /* Definitions of realtype and booleantype       */
#include "iterative.h"        /* Contains the types of preconditioning         */
#include "ida.h"              /* Main header file                              */
#include "idaspgmr.h"         /* Use IDASPGMR linear solver                    */
#include "nvector_parallel.h" /* Definitions of type N_Vector, macro NV_DATA_P */
```

```
#include "sundialsmath.h"     /* Contains RSqrt routine                        */
#include "smalldense.h"       /* Contains definitions for denalloc routine     */
#include "mpi.h"              /* MPI library routines                          */
#include "idabbdpre.h"        /* Definitions for the IDABBDPRE preconditioner  */

/* Problem Constants. */

#define NPREY       1        /* Number of prey (= number of predators). */
#define NUM_SPECIES 2*NPREY

#define PI          RCONST(3.1415926535898) /* pi */
#define FOURPI      (RCONST(4.0)*PI)        /* 4 pi */

#define MXSUB       10    /* Number of x mesh points per processor subgrid */
#define MYSUB       10    /* Number of y mesh points per processor subgrid */
#define NPEX        2     /* Number of subgrids in the x direction */
#define NPEY        2     /* Number of subgrids in the y direction */
#define MX          (MXSUB*NPEX)      /* MX = number of x mesh points */
#define MY          (MYSUB*NPEY)      /* MY = number of y mesh points */
#define NSMXSUB     (NUM_SPECIES * MXSUB)
#define NEQ         (NUM_SPECIES*MX*MY) /* Number of equations in system */
#define AA          RCONST(1.0)    /* Coefficient in above eqns. for a */
#define EE          RCONST(10000.) /* Coefficient in above eqns. for a */
#define GG          RCONST(0.5e-6) /* Coefficient in above eqns. for a */
#define BB          RCONST(1.0)    /* Coefficient in above eqns. for b */
#define DPREY       RCONST(1.0)    /* Coefficient in above eqns. for d */
#define DPRED       RCONST(0.05)   /* Coefficient in above eqns. for d */
#define ALPHA       RCONST(50.)    /* Coefficient alpha in above eqns. */
#define BETA        RCONST(1000.)  /* Coefficient beta in above eqns. */
#define AX          RCONST(1.0)    /* Total range of x variable */
#define AY          RCONST(1.0)    /* Total range of y variable */
#define RTOL        RCONST(1.e-5)  /*  rtol tolerance */
#define ATOL        RCONST(1.e-5)  /*  atol tolerance */
#define ZERO        RCONST(0.)     /* 0. */
#define ONE         RCONST(1.0)    /* 1. */
#define NOUT        6
#define TMULT       RCONST(10.0)   /* Multiplier for tout values */
#define TADD        RCONST(0.3)    /* Increment for tout values */

/* User-defined vector accessor macro IJ_Vptr. */

/*
 * IJ_Vptr is defined in order to express the underlying 3-d structure of the
 * dependent variable vector from its underlying 1-d storage (an N_Vector).
 * IJ_Vptr(vv,i,j) returns a pointer to the location in vv corresponding to
 * species index is = 0, x-index ix = i, and y-index jy = j.
 */

#define IJ_Vptr(vv,i,j) (&NV_Ith_P(vv, (i)*NUM_SPECIES + (j)*NSMXSUB ))

/* Type: UserData.  Contains problem constants, preconditioner data, etc. */

typedef struct {
  long int ns, np, thispe, npes, ixsub, jysub, npex, npey;
  long int mxsub, mysub, nsmxsub, nsmxsub2;
  realtype dx, dy, **acoef;
  realtype cox[NUM_SPECIES], coy[NUM_SPECIES], bcoef[NUM_SPECIES],
    rhs[NUM_SPECIES], cext[(MXSUB+2)*(MYSUB+2)*NUM_SPECIES];
  MPI_Comm comm;
  N_Vector rates;
  long int n_local;
} *UserData;

/* Prototypes for functions called by the IDA Solver. */

static int resweb(realtype tt,
                  N_Vector cc, N_Vector cp, N_Vector rr,
                  void *res_data);

static int reslocal(long int Nlocal, realtype tt,
                    N_Vector cc, N_Vector cp, N_Vector res,
                    void *res_data);

static int rescomm(long int Nlocal, realtype tt,
                   N_Vector cc, N_Vector cp,
                   void *res_data);

/* Prototypes for supporting functions */

static void BSend(MPI_Comm comm, long int thispe, long int ixsub, long int jysub,
                  long int dsizex, long int dsizey, realtype carray[]);

static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int thispe,
                      long int ixsub, long int jysub,
                      long int dsizex, long int dsizey,
                      realtype cext[], realtype buffer[]);

static void BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
                      long int dsizex, realtype cext[], realtype buffer[]);

static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                     UserData webdata);

static realtype dotprod(long int size, realtype *x1, realtype *x2);

/* Prototypes for private functions */

static void InitUserData(UserData webdata, int thispe, int npes,
                         MPI_Comm comm);

static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
                               N_Vector scrtch, UserData webdata);

static void PrintHeader(long int SystemSize, int maxl,
                        long int mudq, long int mldq,
                        long int mukeep, long int mlkeep,
                        realtype rtol, realtype atol);

```

```
static void PrintOutput(void *mem, N_Vector cc, realtype time,
                        UserData webdata, MPI_Comm comm);

static void PrintFinalStats(void *mem, void *P_data);

static int check_flag(void *flagvalue, char *funcname, int opt, int id);

/*
 *--------------------------------------------------------------------
 * MAIN PROGRAM
 *--------------------------------------------------------------------
 */

int main(int argc, char *argv[])
{
  MPI_Comm comm;
  void *mem, *P_data;
  UserData webdata;
  long int SystemSize, local_N, mudq, mldq, mukeep, mlkeep;
  realtype rtol, atol, t0, tout, tret;
  N_Vector cc, cp, res, id;
  int thispe, npes, maxl, iout, retval;

  cc = cp = res = id = NULL;
  webdata = NULL;
  mem = P_data = NULL;

  /* Set communicator, and get processor number and total number of PE’s. */

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

  if (npes != NPEX*NPEY) {
    if (thispe == 0)
      fprintf(stderr,
              "\nMPI_ERROR(0): npes = %d not equal to NPEX*NPEY = %d\n",
              npes, NPEX*NPEY);
    MPI_Finalize();
    return(1);
  }

  /* Set local length (local_N) and global length (SystemSize). */

  local_N = MXSUB*MYSUB*NUM_SPECIES;
  SystemSize = NEQ;

  /* Set up user data block webdata. */

  webdata = (UserData) malloc(sizeof *webdata);
  webdata->rates = N_VNew_Parallel(comm, local_N, SystemSize);
  webdata->acoef = denalloc(NUM_SPECIES);

  InitUserData(webdata, thispe, npes, comm);

  /* Create needed vectors, and load initial values.
     The vector res is used temporarily only.        */

  cc  = N_VNew_Parallel(comm, local_N, SystemSize);
  if(check_flag((void *)cc, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);

  cp  = N_VNew_Parallel(comm, local_N, SystemSize);
  if(check_flag((void *)cp, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);

  res = N_VNew_Parallel(comm, local_N, SystemSize);
  if(check_flag((void *)res, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);

  id  = N_VNew_Parallel(comm, local_N, SystemSize);
  if(check_flag((void *)id, "N_VNew_Parallel", 0, thispe)) MPI_Abort(comm, 1);

  SetInitialProfiles(cc, cp, id, res, webdata);

  N_VDestroy_Parallel(res);

  /* Set remaining inputs to IDAMalloc. */

  t0 = ZERO;
  rtol = RTOL;
  atol = ATOL;

  /* Call IDACreate and IDAMalloc to initialize solution */

  mem = IDACreate();
  if(check_flag((void *)mem, "IDACreate", 0, thispe)) MPI_Abort(comm, 1);

  retval = IDASetRdata(mem, webdata);
  if(check_flag(&retval, "IDASetRdata", 1, thispe)) MPI_Abort(comm, 1);

  retval = IDASetId(mem, id);
  if(check_flag(&retval, "IDASetId", 1, thispe)) MPI_Abort(comm, 1);

  retval = IDAMalloc(mem, resweb, t0, cc, cp, IDA_SS, rtol, &atol);
  if(check_flag(&retval, "IDAMalloc", 1, thispe)) MPI_Abort(comm, 1);

  /* Call IDABBDPrecAlloc to initialize the band-block-diagonal preconditioner.
     The half-bandwidths for the difference quotient evaluation are exact
     for the system Jacobian, but only a 5-diagonal band matrix is retained. */

  mudq = mldq = NSMXSUB;
  mukeep = mlkeep = 2;
  P_data = IDABBDPrecAlloc(mem, local_N, mudq, mldq, mukeep, mlkeep,
                           ZERO, reslocal, NULL);
  if(check_flag((void *)P_data, "IDABBDPrecAlloc", 0, thispe)) MPI_Abort(comm, 1);

  /* Call IDABBDSpgmr to specify the IDA linear solver IDASPGMR and specify
     the preconditioner routines supplied
     maxl (max. Krylov subspace dim.) is set to 12.   */

  maxl = 12;
  retval = IDABBDSpgmr(mem, maxl, P_data);
  if(check_flag(&retval, "IDABBDSpgmr", 1, thispe)) MPI_Abort(comm, 1);

  /* Call IDACalcIC (with default options) to correct the initial values. */

  tout = RCONST(0.001);
  retval = IDACalcIC(mem, t0, cc, cp, IDA_YA_YDP_INIT, tout);
  if(check_flag(&retval, "IDACalcIC", 1, thispe)) MPI_Abort(comm, 1);

  /* On PE 0, print heading, basic parameters, initial values. */

  if (thispe == 0) PrintHeader(SystemSize, maxl,
                               mudq, mldq, mukeep, mlkeep,
                               rtol, atol);
  PrintOutput(mem, cc, t0, webdata, comm);

  /* Call IDA in tout loop, normal mode, and print selected output. */

  for (iout = 1; iout <= NOUT; iout++) {

    retval = IDASolve(mem, tout, &tret, cc, cp, IDA_NORMAL);
    if(check_flag(&retval, "IDASolve", 1, thispe)) MPI_Abort(comm, 1);

    PrintOutput(mem, cc, tret, webdata, comm);

    if (iout < 3) tout *= TMULT;
    else          tout += TADD;

  }

  /* On PE 0, print final set of statistics. */

  if (thispe == 0)  PrintFinalStats(mem, P_data);

  /* Free memory. */

  N_VDestroy_Parallel(cc);
  N_VDestroy_Parallel(cp);
  N_VDestroy_Parallel(id);

  IDABBDPrecFree(P_data);

  IDAFree(mem);

  denfree(webdata->acoef);
  N_VDestroy_Parallel(webdata->rates);
  free(webdata);

  MPI_Finalize();

  return(0);
}
```

```

/*
 *--------------------------------------------------------------------
 * PRIVATE FUNCTIONS
 *--------------------------------------------------------------------
 */

/*
 * InitUserData: Load problem constants in webdata (of type UserData).
 */

static void InitUserData(UserData webdata, int thispe, int npes,
                         MPI_Comm comm)
{
  int i, j, np;
  realtype *a1,*a2, *a3, *a4, dx2, dy2, **acoef, *bcoef, *cox, *coy;

  webdata->jysub = thispe / NPEX;
  webdata->ixsub = thispe - (webdata->jysub)*NPEX;
  webdata->mxsub = MXSUB;
  webdata->mysub = MYSUB;
  webdata->npex = NPEX;
  webdata->npey = NPEY;
  webdata->ns = NUM_SPECIES;
  webdata->np = NPREY;
  webdata->dx = AX/(MX-1);
  webdata->dy = AY/(MY-1);
  webdata->thispe = thispe;
  webdata->npes   = npes;
  webdata->nsmxsub = MXSUB * NUM_SPECIES;
  webdata->nsmxsub2 = (MXSUB+2)*NUM_SPECIES;
  webdata->comm = comm;
  webdata->n_local = MXSUB*MYSUB*NUM_SPECIES;

  /* Set up the coefficients a and b plus others found in the equations. */

  np = webdata->np;
  dx2 = (webdata->dx)*(webdata->dx);
  dy2 = (webdata->dy)*(webdata->dy);

  acoef = webdata->acoef;
  bcoef = webdata->bcoef;
  cox = webdata->cox;
  coy = webdata->coy;

  for (i = 0; i < np; i++) {
    a1 = &(acoef[i][np]);
    a2 = &(acoef[i+np][0]);
    a3 = &(acoef[i][0]);
    a4 = &(acoef[i+np][np]);
    /*  Fill in the portion of acoef in the four quadrants, row by row. */
    for (j = 0; j < np; j++) {
      *a1++ =  -GG;
      *a2++ =   EE;
      *a3++ = ZERO;
      *a4++ = ZERO;
    }

    /* Reset the diagonal elements of acoef to -AA. */
    acoef[i][i] = -AA; acoef[i+np][i+np] = -AA;

    /* Set coefficients for b and diffusion terms. */
    bcoef[i] = BB; bcoef[i+np] = -BB;
    cox[i] = DPREY/dx2; cox[i+np] = DPRED/dx2;
    coy[i] = DPREY/dy2; coy[i+np] = DPRED/dy2;
  }

}

/*
 * SetInitialProfiles: Set initial conditions in cc, cp, and id.
 * A polynomial profile is used for the prey cc values, and a constant
 * (1.0e5) is loaded as the initial guess for the predator cc values.
 * The id values are set to 1 for the prey and 0 for the predators.
 * The prey cp values are set according to the given system, and
 * the predator cp values are set to zero.
 */

static void SetInitialProfiles(N_Vector cc, N_Vector cp, N_Vector id,
                               N_Vector res, UserData webdata)
{
  long int ixsub, jysub, mxsub, mysub, nsmxsub, np, ix, jy, is;
  realtype *cxy, *idxy, *cpxy, dx, dy, xx, yy, xyfactor;

  ixsub = webdata->ixsub;
  jysub = webdata->jysub;
  mxsub = webdata->mxsub;
  mysub = webdata->mxsub;
  nsmxsub = webdata->nsmxsub;
  dx = webdata->dx;
  dy = webdata->dy;
  np = webdata->np;

  /* Loop over grid, load cc values and id values. */
  for (jy = 0; jy < mysub; jy++) {
    yy = (jy + jysub*mysub) * dy;
    for (ix = 0; ix < mxsub; ix++) {
      xx = (ix + ixsub*mxsub) * dx;
      xyfactor = 16.*xx*(1. - xx)*yy*(1. - yy);
      xyfactor *= xyfactor;

      cxy = IJ_Vptr(cc,ix,jy);
      idxy = IJ_Vptr(id,ix,jy);
      for (is = 0; is < NUM_SPECIES; is++) {
        if (is < np) { cxy[is] = RCONST(10.0) + (realtype)(is+1)*xyfactor; idxy[is] = ONE; }
        else { cxy[is] = 1.0e5; idxy[is] = ZERO; }
      }
    }
  }

  /* Set c' for the prey by calling the residual function with cp = 0. */

  N_VConst(ZERO, cp);
  resweb(ZERO, cc, cp, res, webdata);
  N_VScale(-ONE, res, cp);

  /* Set c' for predators to 0. */

  for (jy = 0; jy < mysub; jy++) {
    for (ix = 0; ix < mxsub; ix++) {
      cpxy = IJ_Vptr(cp,ix,jy);
      for (is = np; is < NUM_SPECIES; is++) cpxy[is] = ZERO;
    }
  }
}

/*
 * Print first lines of output (problem description)
 * and table headerr
 */

static void PrintHeader(long int SystemSize, int maxl,
                        long int mudq, long int mldq,
                        long int mukeep, long int mlkeep,
                        realtype rtol, realtype atol)
{
  printf("\niwebbbd: Predator-prey DAE parallel example problem for IDA \n\n");
  printf("Number of species ns: %d", NUM_SPECIES);
  printf("     Mesh dimensions: %d x %d", MX, MY);
  printf("     Total system size: %ld\n",SystemSize);
  printf("Subgrid dimensions: %d x %d", MXSUB, MYSUB);
  printf("     Processor array: %d x %d\n", NPEX, NPEY);
#if defined(SUNDIALS_EXTENDED_PRECISION)
  printf("Tolerance parameters:  rtol = %Lg   atol = %Lg\n", rtol, atol);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
  printf("Tolerance parameters:  rtol = %lg   atol = %lg\n", rtol, atol);
#else
  printf("Tolerance parameters:  rtol = %g   atol = %g\n", rtol, atol);
#endif
  printf("Linear solver: IDASPGMR     Max. Krylov dimension maxl: %d\n", maxl);
  printf("Preconditioner: band-block-diagonal (IDABBDPRE), with parameters\n");
  printf("     mudq = %ld,  mldq = %ld,  mukeep = %ld,  mlkeep = %ld\n",
         mudq, mldq, mukeep, mlkeep);
  printf("CalcIC called to correct initial predator concentrations \n\n");
  printf("-----------------------------------------------------------\n");
  printf("  t        bottom-left  top-right");
  printf("    | nst  k      h\n");
  printf("-----------------------------------------------------------\n\n");
}


/*
```

```
 * PrintOutput: Print output values at output time t = tt.
 * Selected run statistics are printed.  Then values of c1 and c2
 * are printed for the bottom left and top right grid points only.
 */

static void PrintOutput(void *mem, N_Vector cc, realtype tt,
                        UserData webdata, MPI_Comm comm)
{
  MPI_Status status;
  realtype *cdata, clast[2], hused;
  long int nst;
  int i, kused, flag, thispe, npelast, ilast;;

  thispe = webdata->thispe;
  npelast = webdata->npes - 1;
  cdata = NV_DATA_P(cc);

  /* Send conc. at top right mesh point from PE npes-1 to PE 0. */
  if (thispe == npelast) {
    ilast = NUM_SPECIES*MXSUB*MYSUB - 2;
    if (npelast != 0)
      MPI_Send(&cdata[ilast], 2, PVEC_REAL_MPI_TYPE, 0, 0, comm);
    else { clast[0] = cdata[ilast]; clast[1] = cdata[ilast+1]; }
  }

  /* On PE 0, receive conc. at top right from PE npes - 1.
     Then print performance data and sampled solution values. */

  if (thispe == 0) {

    if (npelast != 0)
      MPI_Recv(&clast[0], 2, PVEC_REAL_MPI_TYPE, npelast, 0, comm, &status);

    flag = IDAGetLastOrder(mem, &kused);
    check_flag(&flag, "IDAGetLastOrder", 1, thispe);
    flag = IDAGetNumSteps(mem, &nst);
    check_flag(&flag, "IDAGetNumSteps", 1, thispe);
    flag = IDAGetLastStep(mem, &hused);
    check_flag(&flag, "IDAGetLastStep", 1, thispe);

#if defined(SUNDIALS_EXTENDED_PRECISION)
    printf("%8.2Le %12.4Le %12.4Le   | %3ld  %1d %12.4Le\n",
         tt, cdata[0], clast[0], nst, kused, hused);
    for (i=1;i<NUM_SPECIES;i++)
      printf("         %12.4Le %12.4Le   |\n",cdata[i],clast[i]);
#elif defined(SUNDIALS_DOUBLE_PRECISION)
    printf("%8.2le %12.4le %12.4le   | %3ld  %1d %12.4le\n",
         tt, cdata[0], clast[0], nst, kused, hused);
    for (i=1;i<NUM_SPECIES;i++)
      printf("         %12.4le %12.4le   |\n",cdata[i],clast[i]);
#else
    printf("%8.2e %12.4e %12.4e   | %3ld  %1d %12.4e\n",
         tt, cdata[0], clast[0], nst, kused, hused);
    for (i=1;i<NUM_SPECIES;i++)
```

```
      printf("         %12.4e %12.4e   |\n",cdata[i],clast[i]);
#endif
    printf("\n");

  }

}

/*
 * PrintFinalStats: Print final run data contained in iopt.
 */

static void PrintFinalStats(void *mem, void *P_data)
{
  long int nst, nre, nreS, netf, ncfn, nni, ncfl, nli, npe, nps, nge;
  int flag;

  flag = IDAGetNumSteps(mem, &nst);
  check_flag(&flag, "IDAGetNumSteps", 1, 0);
  flag = IDAGetNumResEvals(mem, &nre);
  check_flag(&flag, "IDAGetNumResEvals", 1, 0);
  flag = IDAGetNumErrTestFails(mem, &netf);
  check_flag(&flag, "IDAGetNumErrTestFails", 1, 0);
  flag = IDAGetNumNonlinSolvConvFails(mem, &ncfn);
  check_flag(&flag, "IDAGetNumNonlinSolvConvFails", 1, 0);
  flag = IDAGetNumNonlinSolvIters(mem, &nni);
  check_flag(&flag, "IDAGetNumNonlinSolvIters", 1, 0);

  flag = IDASpgmrGetNumConvFails(mem, &ncfl);
  check_flag(&flag, "IDASpgmrGetNumConvFails", 1, 0);
  flag = IDASpgmrGetNumLinIters(mem, &nli);
  check_flag(&flag, "IDASpgmrGetNumLinIters", 1, 0);
  flag = IDASpgmrGetNumPrecEvals(mem, &npe);
  check_flag(&flag, "IDASpgmrGetNumPrecEvals", 1, 0);
  flag = IDASpgmrGetNumPrecSolves(mem, &nps);
  check_flag(&flag, "IDASpgmrGetNumPrecSolves", 1, 0);
  flag = IDASpgmrGetNumResEvals(mem, &nreS);
  check_flag(&flag, "IDASpgmrGetNumResEvals", 1, 0);

  flag = IDABBDPrecGetNumGfnEvals(P_data, &nge);
  check_flag(&flag, "IDABBDPrecGetNumGfnEvals", 1, 0);

  printf("------------------------------------------------------------\n");
  printf("\nFinal statistics: \n\n");

  printf("Number of steps                    = %ld\n", nst);
  printf("Number of residual evaluations     = %ld\n", nre+nreS);
  printf("Number of nonlinear iterations     = %ld\n", nni);
  printf("Number of error test failures      = %ld\n", netf);
  printf("Number of nonlinear conv. failures = %ld\n\n", ncfn);

  printf("Number of linear iterations        = %ld\n", nli);
  printf("Number of linear conv. failures    = %ld\n\n", ncfl);

  printf("Number of preconditioner setups    = %ld\n", npe);
  printf("Number of preconditioner solves    = %ld\n", nps);
  printf("Number of local residual evals.    = %ld\n", nge);

}

/*
 * Check function return value...
 *   opt == 0 means SUNDIALS function allocates memory so check if
 *            returned NULL pointer
 *   opt == 1 means SUNDIALS function returns a flag so check if
 *            flag >= 0
 *   opt == 2 means function allocates memory so check if returned
 *            NULL pointer
 */

static int check_flag(void *flagvalue, char *funcname, int opt, int id)
{
  int *errflag;

  if (opt == 0 && flagvalue == NULL) {
    /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1);
  } else if (opt == 1) {
    /* Check if flag < 0 */
    errflag = (int *) flagvalue;
    if (*errflag < 0) {
      fprintf(stderr,
              "\nSUNDIALS_ERROR(%d): %s() failed with flag = %d\n\n",
              id, funcname, *errflag);
      return(1);
    }
  } else if (opt == 2 && flagvalue == NULL) {
    /* Check if function returned NULL pointer - no memory allocated */
    fprintf(stderr,
            "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n",
            id, funcname);
    return(1);
  }

  return(0);
}

/*
 *-----------------------------------------------------------------
 * FUNCTIONS CALLED BY IDA & SUPPORTING FUNCTIONS
 *-----------------------------------------------------------------
 */

/*
 * resweb: System residual function for predator-prey system.
 * To compute the residual function F, this routine calls:
 * rescomm, for needed communication, and then
 * reslocal, for computation of the residuals on this processor.
 */

static int resweb(realtype tt,
                  N_Vector cc, N_Vector cp, N_Vector rr,
                  void *res_data)
{
  int retval;
  UserData webdata;
  long int Nlocal;

  webdata = (UserData) res_data;

  Nlocal = webdata->n_local;

  /* Call rescomm to do inter-processor communication. */
  retval = rescomm(Nlocal, tt, cc, cp, res_data);

  /* Call reslocal to calculate the local portion of residual vector. */
  retval = reslocal(Nlocal, tt, cc, cp, rr, res_data);

  return(0);
}

/*
 * rescomm: Communication routine in support of resweb.
 * This routine performs all inter-processor communication of components
 * of the cc vector needed to calculate F, namely the components at all
 * interior subgrid boundaries (ghost cell data).  It loads this data
 * into a work array cext (the local portion of c, extended).
 * The message-passing uses blocking sends, non-blocking receives,
 * and receive-waiting, in routines BRecvPost, BSend, BRecvWait.
 */

static int rescomm(long int Nlocal, realtype tt,
                   N_Vector cc, N_Vector cp,
                   void *res_data)
{

  UserData webdata;
  realtype *cdata, *cext, buffer[2*NUM_SPECIES*MYSUB];
  long int thispe, ixsub, jysub, nsmxsub, nsmysub;
  MPI_Comm comm;
  MPI_Request request[4];

  webdata = (UserData) res_data;
  cdata = NV_DATA_P(cc);

  /* Get comm, thispe, subgrid indices, data sizes, extended array cext. */

  comm = webdata->comm;
  thispe = webdata->thispe;

  ixsub = webdata->ixsub;
  jysub = webdata->jysub;
  cext = webdata->cext;
  nsmxsub = webdata->nsmxsub;
  nsmysub = (webdata->ns)*(webdata->mysub);

  /* Start receiving boundary data from neighboring PEs. */

  BRecvPost(comm, request, thispe, ixsub, jysub, nsmxsub, nsmysub,
            cext, buffer);

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

  BSend(comm, thispe, ixsub, jysub, nsmxsub, nsmysub, cdata);

  /* Finish receiving boundary data from neighboring PEs. */

  BRecvWait(request, ixsub, jysub, nsmxsub, cext, buffer);

  return(0);
}

/*
 * BRecvPost: Start receiving boundary data from neighboring PEs.
 * (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype 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 is also passed in both calls.
 */

static void BRecvPost(MPI_Comm comm, MPI_Request request[], long int my_pe,
                      long int ixsub, long int jysub,
                      long int dsizex, long int dsizey,
                      realtype cext[], realtype buffer[])
{
  long int offsetce;
  /* Have bufleft and bufright use the same buffer. */
  realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;

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

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

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

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

}

/*
 * BRecvWait: Finish receiving boundary data from neighboring PEs.
 * (1) buffer should be able to hold 2*NUM_SPECIES*MYSUB realtype 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 is also passed in both calls.
 */

static void BRecvWait(MPI_Request request[], long int ixsub, long int jysub,
                      long int dsizex, realtype cext[], realtype buffer[])
{
  int i;
  long int ly, dsizex2, offsetce, offsetbuf;
  realtype *bufleft = buffer, *bufright = buffer+NUM_SPECIES*MYSUB;
  MPI_Status status;

  dsizex2 = dsizex + 2*NUM_SPECIES;

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

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

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

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

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

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

/*
 * BSend: Send boundary data to neighboring PEs.
 * This routine sends components of cc from internal subgrid boundaries
 * to the appropriate neighbor PEs.
 */

static void BSend(MPI_Comm comm, long int my_pe, long int ixsub, long int jysub,
                  long int dsizex, long int dsizey, realtype cdata[])
{
  int i;
  long int ly, offsetc, offsetbuf;
  realtype bufleft[NUM_SPECIES*MYSUB], bufright[NUM_SPECIES*MYSUB];

  /* If jysub > 0, send data from bottom x-line of cc. */

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

  /* If jysub < NPEY-1, send data from top x-line of cc. */

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

  /* If ixsub > 0, send data from left y-line of cc (via bufleft). */

  if (ixsub != 0) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NUM_SPECIES;
      offsetc = ly*dsizex;
      for (i = 0; i < NUM_SPECIES; i++)
        bufleft[offsetbuf+i] = cdata[offsetc+i];
    }
    MPI_Send(&bufleft[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe-1, 0, comm);
  }

  /* If ixsub < NPEX-1, send data from right y-line of cc (via bufright). */

  if (ixsub != NPEX-1) {
    for (ly = 0; ly < MYSUB; ly++) {
      offsetbuf = ly*NUM_SPECIES;
      offsetc = offsetbuf*MXSUB + (MXSUB-1)*NUM_SPECIES;
      for (i = 0; i < NUM_SPECIES; i++)
        bufright[offsetbuf+i] = cdata[offsetc+i];
    }
    MPI_Send(&bufright[0], dsizey, PVEC_REAL_MPI_TYPE, my_pe+1, 0, comm);
  }
}

/* Define lines are for ease of readability in the following functions. */

#define mxsub      (webdata->mxsub)
#define mysub      (webdata->mysub)
#define npex       (webdata->npex)
#define npey       (webdata->npey)
#define ixsub      (webdata->ixsub)
#define jysub      (webdata->jysub)
#define nsmxsub    (webdata->nsmxsub)
#define nsmxsub2   (webdata->nsmxsub2)
#define np         (webdata->np)
#define dx         (webdata->dx)
#define dy         (webdata->dy)
#define cox        (webdata->cox)
#define coy        (webdata->coy)
#define rhs        (webdata->rhs)
#define cext       (webdata->cext)
#define rates      (webdata->rates)
#define ns         (webdata->ns)
#define acoef      (webdata->acoef)
#define bcoef      (webdata->bcoef)

/*
 * reslocal: Compute res = F(t,cc,cp).
 * This routine assumes that all inter-processor communication of data
 * needed to calculate F has already been done.  Components at interior
 * subgrid boundaries are assumed to be in the work array cext.
 * The local portion of the cc vector is first copied into cext.
 * The exterior Neumann boundary conditions are explicitly handled here
 * by copying data from the first interior mesh line to the ghost cell
 * locations in cext.  Then the reaction and diffusion terms are
 * evaluated in terms of the cext array, and the residuals are formed.
 * The reaction terms are saved separately in the vector webdata->rates
 * for use by the preconditioner setup routine.
 */

static int reslocal(long int Nlocal, realtype tt,
                    N_Vector cc, N_Vector cp, N_Vector rr,
                    void *res_data)
{
  realtype *cdata, *ratesxy, *cpxy, *resxy,
    xx, yy, dcyli, dcyui, dcxli, dcxui;
  long int ix, jy, is, i, locc, ylocce, locce;
  UserData webdata;

  webdata = (UserData) res_data;

  /* Get data pointers, subgrid data, array sizes, work array cext. */

  cdata = NV_DATA_P(cc);

  /* Copy local segment of cc vector into the working extended array cext. */

  locc = 0;
  locce = nsmxsub2 + NUM_SPECIES;
  for (jy = 0; jy < mysub; jy++) {
    for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
    locc = locc + nsmxsub;
    locce = locce + nsmxsub2;
  }

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

  /* If jysub = 0, copy x-line 2 of cc to cext. */
  if (jysub == 0)
    { for (i = 0; i < nsmxsub; i++) cext[NUM_SPECIES+i] = cdata[nsmxsub+i]; }

  /* If jysub = npey-1, copy x-line mysub-1 of cc to cext. */
  if (jysub == npey-1) {
    locc = (mysub-2)*nsmxsub;
    locce = (mysub+1)*nsmxsub2 + NUM_SPECIES;
    for (i = 0; i < nsmxsub; i++) cext[locce+i] = cdata[locc+i];
  }

  /* If ixsub = 0, copy y-line 2 of cc to cext. */
  if (ixsub == 0) {
    for (jy = 0; jy < mysub; jy++) {
      locc = jy*nsmxsub + NUM_SPECIES;
      locce = (jy+1)*nsmxsub2;
      for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
    }
  }

  /* If ixsub = npex-1, copy y-line mxsub-1 of cc to cext. */
  if (ixsub == npex-1) {
    for (jy = 0; jy < mysub; jy++) {
      locc  = (jy+1)*nsmxsub - 2*NUM_SPECIES;
      locce = (jy+2)*nsmxsub2 - NUM_SPECIES;
      for (i = 0; i < NUM_SPECIES; i++) cext[locce+i] = cdata[locc+i];
    }
  }

  /* Loop over all grid points, setting local array rates to right-hand sides.
     Then set rr values appropriately for prey/predator components of F. */

  for (jy = 0; jy < mysub; jy++) {
    ylocce = (jy+1)*nsmxsub2;
    yy     = (jy+jysub*mysub)*dy;

    for (ix = 0; ix < mxsub; ix++) {
      locce = ylocce + (ix+1)*NUM_SPECIES;
      xx = (ix + ixsub*mxsub)*dx;

      ratesxy = IJ_Vptr(rates,ix,jy);
      WebRates(xx, yy, &(cext[locce]), ratesxy, webdata);

      resxy = IJ_Vptr(rr,ix,jy);
      cpxy = IJ_Vptr(cp,ix,jy);

      for (is = 0; is < NUM_SPECIES; is++) {
        dcyli = cext[locce+is]          - cext[locce+is-nsmxsub2];
        dcyui = cext[locce+is+nsmxsub2] - cext[locce+is];

        dcxli = cext[locce+is]             - cext[locce+is-NUM_SPECIES];
        dcxui = cext[locce+is+NUM_SPECIES] - cext[locce+is];

        rhs[is] = cox[is]*(dcxui-dcxli) + coy[is]*(dcyui-dcyli) + ratesxy[is];

        if (is < np) resxy[is] = cpxy[is] - rhs[is];
        else         resxy[is] =          - rhs[is];

      }
    }
  }

  return(0);
}

/*
 * WebRates: Evaluate reaction rates at a given spatial point.
 * At a given (x,y), evaluate the array of ns reaction terms R.
 */

static void WebRates(realtype xx, realtype yy, realtype *cxy, realtype *ratesxy,
                     UserData webdata)
{
  int is;
  realtype fac;

  for (is = 0; is < NUM_SPECIES; is++)
    ratesxy[is] = dotprod(NUM_SPECIES, cxy, acoef[is]);

  fac = ONE + ALPHA*xx*yy + BETA*sin(FOURPI*xx)*sin(FOURPI*yy);

  for (is = 0; is < NUM_SPECIES; is++)
    ratesxy[is] = cxy[is]*( bcoef[is]*fac + ratesxy[is] );

}

/*
 * dotprod: dot product routine for realtype arrays, for use by WebRates.
 */

static realtype dotprod(long int size, realtype *x1, realtype *x2)
```

```
{
  long int i;
  realtype *xx1, *xx2, temp = ZERO;

  xx1 = x1;
  xx2 = x2;
  for (i = 0; i < size; i++)
    temp += (*xx1++) * (*xx2++);

  return(temp);
}

```