```
      PROGRAM TEST

!   This is a sample program using openmp showing ways to use version 1.4 of the
!   FM_PARALLEL package for parallel programs.

!   The first 4 examples compare doing a simple summation 1 + 1/2 + 1/3 + ... + 1/n four ways.

!   The last 2 examples solve a more complicated heat flow problem.

  USE OMP_LIB
      USE FMVALS_PARALLEL
      USE FMZM_PARALLEL

      IMPLICIT NONE

!             Set MAX_THREADS to the maximum number of threads available on your machine.

      INTEGER, PARAMETER :: MAX_THREADS = 8

!             This program shows two ways to use the fm_parallel package.

!             1.  Declare variables to be type(multi) and use them by making explicit calls to
!                 the basic FM routines.

!                 The advantage to this is that the program is usually slightly faster than using
!                 the fmzm interface (from about 4% to 20% faster for the sum example, for the
!                 different compilers I tested.)

!                 The disadvantage is that coding a calculation using calls is like writing in
!                 assembly language.  A simple statement like A = A + B/J turns into multiple
!                 calls to the basic FM routines:  convert J to multi, divide, add, copy the
!                 temporary result of the add back to A.

!                 The main reason for showing method 1 here in the sample program is that if
!                 any of the routines from the FM_Sample_Routines.f95 collection are used,
!                 the user must provide the functions passed to those routines, and they must
!                 be coded using method 1.  This program uses routine FM_GENEQ and passes
!                 it function F6, which is defined at the end.  See the TestFM_parallel program
!                 for more examples.

!             2.  Declare variables to be type(fm), type(im), etc., then use the fmzm interface
!                 (just like the non-parallel standard FM) to translate expressions into the
!                 multiple calls that are needed.

!                 The extra overhead involved in using fmzm makes slightly it slower, but the ease
!                 of writing the code in the normal way usually makes this the preferred choice.

!             Examples 1 and 2 use method 1, and the rest use method 2.
!             The more realistic examples 5 and 6, especially the HEAT_TIME subroutine, would
!             be no fun to re-code using explicit calls.


!-----------------------------------  Method 1  ------------------------------------------------

!             Declare the multiple precision variables for the CALL FM version.

!             The MULTI type defines the internal form for all of the multiple precision types
```

```
!             (FM, IM, ZM) that are defined in the FMZM interface.  The user normally doesn't
!             need to know about type(multi), but this is the required type if the user wants
!             to make direct calls to the low-level FM routines instead of using FMZM.

      TYPE (MULTI) :: X1, X2, X3, X4, T1

!             Declare the openmp multiple precision variables for the CALL FM version.

      TYPE (MULTI) :: X1P, X2P, X3P, X4P, T1P, TSP(MAX_THREADS)

!             Declare the local copy of FM's internal variables that are global module variables
!             in the normal non-parallel version of FM.
!             This is needed only when doing direct calls to the basic FM routines, as in
!             examples 1 and 2.  When using the fmzm interface (examples 3 to 6) it is not needed.

!             Another reason why making explicit FM calls in method 1 with openmp is finicky:
!             I haven't been able to make all the compilers I've tested work with the same
!             code.  But I'm no expert with openmp -- please email a better solution if you
!             know one.

!             Pick the right compiler version here and below in example 2 to run on your machine.

!             *****  gfortran, ifort version  *****

      TYPE (FM_SETTINGS), SAVE :: SETTINGS

!$OMP THREADPRIVATE(SETTINGS)

!             *****  nagfor version  *****

!       TYPE (FM_SETTINGS) :: SETTINGS


!-------------------------------------  Method 2  ---------------------------------------------

!             Declare the local multiple precision variables for the TYPE(FM) version.

      TYPE (FM) :: FM_X1, FM_T1, X(20), Y(20)
      TYPE (FM), ALLOCATABLE :: A(:,:), B(:), C(:)

!             Declare the openmp multiple precision variables for the TYPE(FM) version.

      TYPE (FM) :: FM_X1P, FM_T1P, FM_TSP(MAX_THREADS)

!-----------------------------------------------------------------------------------------------

!             Declare the user function that will be called from subroutine fm_geneq
!             for example 6.  Because it will be called from a lower-level FM routine,
!             it is type(multi) even though the other arguments to fm_geneq are type(fm).

      TYPE (MULTI), EXTERNAL :: F6

!             Declare the openmp non-multiple precision variables.

      INTEGER :: NP

!             Declare the other variables (not multiple precision).
```

```
      CHARACTER(80)  :: ST1
      CHARACTER(175) :: FMT
      INTEGER :: J, K, L, N
      DOUBLE PRECISION :: TIME1, TIME2



!             The first case is not run in parallel.

!             Example 1.  Sum n terms of a series -- non-parallel using explicit calls.
!                         t1 = 1 + 1/2 + 1/3 + ... + 1/n

      SETTINGS%JFORM1 = 2
      N = 3*10**6

      CALL CLOCK_TIME(TIME1)

      CALL FMI2M(0,T1,SETTINGS)
      CALL FMI2M(1,X1,SETTINGS)
      DO J = 1, N
         CALL FMI2M(J,X2,SETTINGS)
         CALL FMDIV(X1,X2,X3,SETTINGS)
         CALL FMADD(T1,X3,X4,SETTINGS)
         CALL FMEQ(X4,T1,SETTINGS)
      ENDDO

      CALL CLOCK_TIME(TIME2)

      WRITE (*,"(//A,F9.2,A,I9,A/)") ' Example 1.', TIME2-TIME1,  &
                 ' seconds for ', N, ' terms (non-parallel using explicit calls).  Sum ='
      CALL FMPRINT(T1,SETTINGS)




!             Example 2.  Sum n terms of a series -- parallel using explicit calls.

      CALL CLOCK_TIME(TIME1)

!             *****  gfortran, ifort version  *****

!$OMP PARALLEL PRIVATE(J,T1P,X1P,X2P,X3P,X4P) SHARED(NP,TSP)

!             *****  nagfor version  *****

! !$OMP PARALLEL PRIVATE(J,T1P,X1P,X2P,X3P,X4P,SETTINGS) SHARED(NP,TSP)

      NP = 3*10**6
      CALL FMI2M(0,T1P,SETTINGS)
      CALL FMI2M(1,X1P,SETTINGS)
      DO J = OMP_GET_THREAD_NUM()+1, NP, OMP_GET_NUM_THREADS()
         CALL FMI2M(J,X2P,SETTINGS)
         CALL FMDIV(X1P,X2P,X3P,SETTINGS)
         CALL FMADD(T1P,X3P,X4P,SETTINGS)
         CALL FMEQ(X4P,T1P,SETTINGS)
      ENDDO
      CALL FMEQ(T1P,TSP(OMP_GET_THREAD_NUM()+1),SETTINGS)
```

```
!$OMP END PARALLEL

      CALL CLOCK_TIME(TIME2)

      DO J = 2, OMP_GET_MAX_THREADS()
         CALL FMADD_R1(TSP(1),TSP(J),SETTINGS)
      ENDDO

      WRITE (*,"(///A,F9.2,A,I9,A/)") ' Example 2.', TIME2-TIME1,  &
                ' seconds for ', N, ' terms (parallel using explicit calls).  Sum ='
      CALL FMPRINT(TSP(1),SETTINGS)
      WRITE (*,*) ' '


!             Example 3.  Sum n terms of a series -- non-parallel using the fmzm interface.

      N = 3*10**6

      CALL CLOCK_TIME(TIME1)

      FM_T1 = 0
      FM_X1 = 1
      DO J = 1, N
         FM_T1 = FM_T1 + FM_X1/J
      ENDDO

      CALL CLOCK_TIME(TIME2)

      WRITE (*,"(//A,F9.2,A,I9,A,F9.2,A/)") ' Example 3.', TIME2-TIME1,  &
                ' seconds for ', N, ' terms (non-parallel using fmzm).  Sum ='
      WRITE (FMT,"(A,I6,'.',I6)") 'F', FM_SIGNIFICANT_DIGITS+5, FM_SIGNIFICANT_DIGITS
      CALL FM_FORM(TRIM(FMT),FM_T1,ST1)
      WRITE (*   ,"(/5X,A)") TRIM(ST1)


!             Example 4.  Sum n terms of a series -- parallel using the fmzm interface.

      CALL CLOCK_TIME(TIME1)

!$OMP PARALLEL PRIVATE(J,FM_T1P,FM_X1P) SHARED(NP,FM_TSP)

      NP = 3*10**6
      FM_T1P = 0
      FM_X1P = 1
      DO J = OMP_GET_THREAD_NUM()+1, NP, OMP_GET_NUM_THREADS()
         FM_T1P = FM_T1P + FM_X1P/J
      ENDDO

      FM_TSP(OMP_GET_THREAD_NUM()+1) = FM_T1P

!$OMP END PARALLEL

      CALL CLOCK_TIME(TIME2)

      DO J = 2, OMP_GET_MAX_THREADS()
         FM_TSP(1) = FM_TSP(1) + FM_TSP(J)
      ENDDO
```

```
      WRITE (*,"(///A,F9.2,A,I9,A,F9.2,A/)") ' Example 4.', TIME2-TIME1,  &
                ' seconds for ', N, ' terms (parallel using fmzm)  Sum ='
      WRITE (FMT,"(A,I6,'.',I6)") 'F', FM_SIGNIFICANT_DIGITS+5, FM_SIGNIFICANT_DIGITS
      CALL FM_FORM(TRIM(FMT),FM_TSP(1),ST1)
      WRITE (*   ,"(/5X,A)") TRIM(ST1)




!             Example 5.  Approximate the solution to a heat-flow partial differential equation.

!                         Find the time that the center of a 2x2 square plate reaches
!                         temperature u=1.
!                         u(x,y,t) is the temperature at location (x,y) at time t, where
!                         -1 <= x <= 1, -1 <= y <= 1 and t >= 0.
!                         Initial conditions: u(x,y,0)=0 at time t=0, and at t>0 the x=1 side is
!                                             u(1,y,t)=5, and the other three sides are held at u=0.

!                         So we need to solve for t:  u(0,0,t) = 1.

!                         Usually we want to control the parallel execution from the highest
!                         level of the program, since the overhead from starting and stopping
!                         multiple threads many times at a lower level will reduce the speedup.

!                         For this example we show how the parallel control can be done at a
!                         lower level, from within the HEAT_TIME subroutine.

      CALL CLOCK_TIME(TIME1)

      N = 30
      CALL HEAT_TIME(N,FM_T1)

      CALL CLOCK_TIME(TIME2)
      WRITE (*,"(///A,F9.2,A,I2,A,I2,A,I2,A/)")  &
               ' Example 5.', TIME2-TIME1, ' seconds for the heat equation using N = ', N,  &
               ' and a ', 2*N+1,'x',2*N+1,' grid.'
      WRITE (*,"(A,F17.13///)") ' Critical time  t1 = ', TO_DP(FM_T1)


!             Example 6.  Approximate the solution to a heat-flow partial differential equation.

!                         The result from just a single call to HEAT_TIME as in example 5 is
!                         not very accurate.  Using N=30 and a 61x61 grid gives a result that
!                         is accurate to only 3 digits (0.424).

!                         We can do better by making several calls and then using some theory
!                         about the errors that come from the finite difference method used
!                         in HEAT_TIME.

      WRITE (*,"(A/)") ' Example 6.  Get several approximations to t1 for different grid sizes.'

      DO J = 2, 40, 2
         N = J
         X(J/2) = N
         CALL HEAT_TIME(N,Y(J/2))
         WRITE (*,"(A,I2,A,F17.13)") ' N = ', J, '  t1 = ', TO_DP(Y(J/2))
```

```
      ENDDO

      WRITE (*,"(//A/)") '             Fit increasingly accurate error formulas' //  &
                         ' for better accuracy.'

!             To fit an error formula of degree K to the data (X(1),Y(1)), ..., (X(L),Y(L)),
!             subroutine FM_GENEQ will return matrix A and vector B such that the coefficients C
!             in the error formula can be found by solving the linear system  A * C = B.
!             The constant term in the fitted formula, C(1), is the estimate for t1.

!             F6 is the user-supplied function defining the form of the error series that FM_GENEQ
!             will use to generate the linear system.

!             Routines FM_GENEQ and FM_LIN_SOLVE are in the "FM_Sample_Routines.f95" file in the
!             standard version of FM, and thread-safe versions of those sample routines are
!             included in the all-in-one parallel version in file "FM_parallel.f95" that is used
!             to run this sample program.

!             Since all these matrices are no larger than 20x20, the operations in this loop
!             take less than a second for all 18 fits together, so there is no reason to
!             make them parallel.

      L = 20
      DO K = 3, L
         ALLOCATE(A(K,K),B(K),C(K))
         CALL FM_GENEQ(F6,A,B,K,X,Y,L)
         CALL FM_LIN_SOLVE(A,C,B,K,FM_X1)

         FMT = 'F35.30'
         CALL FM_FORM(TRIM(FMT),C(1),ST1)
         WRITE (*,"(A,I2,A,A)") ' Degree of error polynomial = ', K, '       t1 =', TRIM(ST1(3:80))
         FM_T1 = C(1)
         DEALLOCATE(A,B,C)
      ENDDO

!             Check to see how accurate the last estimate is.
!             References for the correct value:
!             https://oeis.org/A117238
!             https://en.wikipedia.org/wiki/Hundred-dollar,_Hundred-digit_Challenge_problems

      FM_X1 = ABS( TO_FM('0.4240113870336883637974336685932564512477620906642747621971l2') - FM_T1 )
      WRITE (*,"(/A,ES20.10//)") ' As a check, subtracting this last result from the correct' //  &
                                 ' answer gives ', TO_DP(FM_X1)

      END PROGRAM TEST


      SUBROUTINE CLOCK_TIME(CLOCK_SECONDS)

!  Convert intrinsic function DATE_AND_TIME character output to a real second count.
!  It seems optimistic to worry about the century changing during the time interval
!  being measured, so that field is ignored.

!  Since not all months have the same number of days, using the average number of seconds/month
!  below may cause a wrong elapsed time to be computed if the start time and stop time calls
!  occur in different months.

!  DATE = "ccyymmdd"  for century, year, month, date
```

```
!  TIME = "hhmmss.sss"  for hour, minute, seconds

      IMPLICIT NONE
      DOUBLE PRECISION :: CLOCK_SECONDS
      CHARACTER( 8) :: DATE
      CHARACTER(10) :: TIME
      INTEGER :: FIELD(8)

      CALL DATE_AND_TIME(DATE,TIME)

      READ( DATE , "(4I2)" ) FIELD(1:4)
      READ( TIME , "(3I2,1X,I3)" ) FIELD(5:8)

      CLOCK_SECONDS = FIELD(2) * 3.15576D7 + FIELD(3) * 2.6298D6 + FIELD(4) * 8.64D4 +  &
                      FIELD(5) * 3.6D+3 + FIELD(6) * 60.0D0 + FIELD(7) + FIELD(8) / 1000.0D0

      END SUBROUTINE CLOCK_TIME


      SUBROUTINE HEAT_TIME(N,FM_T1)
  USE OMP_LIB
      USE FMZM_PARALLEL

!  Approximate the time that the center of a 2x2 square plate reaches temperature u=1.
!  u(x,y,t) is the temperature at location (x,y) at time t, where -1 <= x <= 1 and -1 <= y <= 1.
!  So we want to solve  u(0,0,t) = 1.
!  Initial conditions: u(x,y,0)=0 at time t=0, and at t>0 the x=1 side is u(1,y,t)=5,
!                      while the other three sides are held at u=0.

!  N determines the number of grid points on the plate at which to approximate the solution.
!    These points are 1/N distance apart in the x and y directions, with the grid subscripts
!    running from -N to N, so there is a total of (2*N+1)*(2*N+1) points in thr grid.

!  FM_T1 is returned as an estimate for the time at which u(0,0,t) = 1.

      IMPLICIT NONE
      INTEGER, PARAMETER :: NMAX = 40
      INTEGER :: J,K,N,NSTEP
      TYPE (FM) :: FM_T1,C,H,T,GCENTER(12)

!             GRID( x , y ) is the approximate solution, and is stepped forward in time.

      TYPE (FM), SAVE :: GRID(-NMAX:NMAX , -NMAX:NMAX), GNEW(-NMAX:NMAX , -NMAX:NMAX)

!             h is the stepsize for x and y.

      H = TO_FM(1)/N

      DO J = -N, N
         DO K = -N, N
            GRID(J,K) = 0
         ENDDO
      ENDDO

      DO J = 1, 12
         GCENTER(J) = 0
      ENDDO
```

```
!             Initialize for x = 1.

      DO K = -N, N
         GRID(N,K) = 5
      ENDDO

      NSTEP = 0

!             Iterate a finite difference approximation with time step h^2/6.

  110 NSTEP = NSTEP + 1

!$OMP PARALLEL PRIVATE(J,K) SHARED(GRID,GNEW,N)

      DO J = -N+1+OMP_GET_THREAD_NUM(), N-1, OMP_GET_NUM_THREADS()

!             From symmetry, only half the grid must be computed.

         DO K = 0, N-1
            GNEW(J,K) = (  GRID(J-1,K-1) +  4*GRID(J-1,K) +   GRID(J-1,K+1) +  &
                         4*GRID(J  ,K-1) + 16*GRID(J  ,K) + 4*GRID(J  ,K+1) +  &
                           GRID(J+1,K-1) +  4*GRID(J+1,K) +   GRID(J+1,K+1)  ) / 36
         ENDDO
      ENDDO

!$OMP END PARALLEL

!$OMP PARALLEL PRIVATE(J,K) SHARED(GRID,GNEW,N)

      DO J = -N+1+OMP_GET_THREAD_NUM(), N-1, OMP_GET_NUM_THREADS()
         DO K = 0, N-1
            GRID(J,K) = GNEW(J,K)
         ENDDO
      ENDDO

!$OMP END PARALLEL

      DO J = -N+1, N-1
         GRID(J,-1) = GNEW(J,1)
      ENDDO

      DO J = 2, 12
         GCENTER(J-1) = GCENTER(J)
      ENDDO
      GCENTER(12) = GRID(0,0)

      IF (GCENTER(7) < 1) GO TO 110

!             The last 12 time steps for grid(0,0) are saved in GCENTER.
!             The first 6 points are before the temperature reached 1.0, and the last
!             6 points are above 1.0.

!             Interpolate these 12 points to get an estimate of the time when u(0,0,t) = 1

      FM_T1 = 0
      C = 1
      DO J = 1, 12
```

```
         T = NSTEP*H*H/6-(12-J)*H*H/6
         DO K = 1, 12
            IF (K == J) CYCLE
            T = T * (C-GCENTER(K)) / (GCENTER(J)-GCENTER(K))
         ENDDO
         FM_T1 = FM_T1 + T
      ENDDO

      END SUBROUTINE HEAT_TIME

      FUNCTION F6(J,X,SETTINGS)     RESULT (RETURN_VALUE)

!  Many of the routines from FM_Sample_Routines.f95, like fm_geneq in example 6, need to call
!  a function defined in the user's program.  That function defines the model to be fitted,
!  the function to be minimized, integrated, etc.

!  Because these user functions are called from a lower-level FM routine, they use FMVALS_PARALLEL
!  and not FMZM_PARALLEL.  Like examples 1 and 2 they need a SETTINGS variable added to their
!  argument list and they need to do their calculations using calls to the low-level FM routines.

      USE FMVALS_PARALLEL
      IMPLICIT NONE

!  This defines the model function being fitted to the data points.
!  For the heat equation example, the model function is:

!  F6(J,X) = 1/X**(2*J-2)

!  This will fit the data using the error model function  c1 + c2/n**2 + c3/n**4 + ....

      INTEGER :: J
      TYPE (MULTI) :: RETURN_VALUE, X
      TYPE (FM_SETTINGS) :: SETTINGS

      CALL FMIPWR(X,2-2*J,RETURN_VALUE,SETTINGS)

      END FUNCTION F6
```