```
# include <cmath>
# include <cstdlib>
# include <ctime>
# include <fstream>
# include <iostream>
# include <mpi.h>

using namespace std;

int main ( int argc, char *argv[] );
double boundary_condition ( double x, double time );
double initial_condition ( double x, double time );
double rhs ( double x, double time );
void timestamp ( );
void update ( int id, int p );

//****************************************************************************80

int main ( int argc, char *argv[] )

//****************************************************************************80
//
//  Purpose:
//
//    MAIN is the main program for HEAT_MPI.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    15 June 2016
//
//  Author:
//
//    John Burkardt
//
//  Reference:
//
//    William Gropp, Ewing Lusk, Anthony Skjellum,
//    Using MPI: Portable Parallel Programming with the
//    Message-Passing Interface,
//    Second Edition,
//    MIT Press, 1999,
//    ISBN: 0262571323,
//    LC: QA76.642.G76.
//
//    Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker,
//    Jack Dongarra,
//    MPI: The Complete Reference,
//    Volume I: The MPI Core,
//    Second Edition,
//    MIT Press, 1998,
//    ISBN: 0-262-69216-3,
//     LC: QA76.642.M65.
//
{
  int id;
  int p;
  double wtime;


  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &id );
  MPI_Comm_size ( MPI_COMM_WORLD, &p );
```

```
  if ( id == 0 )
  {
    timestamp ( );
    cout << "\n";
    cout << "HEAT_MPI:\n";
    cout << "  C++/MPI version\n";
    cout << "  Solve the 1D time-dependent heat equation.\n";
  }
//
//  Record the starting time.
//
  if ( id == 0 )
  {
    wtime = MPI_Wtime ( );
  }

  update ( id, p );
//
//  Record the final time.
//
  if ( id == 0 )
  {
    wtime = MPI_Wtime ( ) - wtime;

    cout << "\n";
    cout << "  Wall clock elapsed seconds = " << wtime << "\n";
  }
//
//  Terminate MPI.
//
  MPI_Finalize ( );
//
//  Terminate.
//
  if ( id == 0 )
  {
    cout << "\n";
    cout << "HEAT_MPI:\n";
    cout << "  Normal end of execution.\n";
    cout << "\n";
    timestamp ( );
  }
  return 0;
}
//****************************************************************************80

void update ( int id, int p )

//****************************************************************************80
//
//  Purpose:
//
//    UPDATE computes the solution of the heat equation.
//
//  Discussion:
//
//    If there is only one processor ( P == 1 ), then the program writes the
//    values of X and H to files.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    14 June 2016
//
```

```
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Input, int ID, the id of this processor.
//
//    Input, int P, the number of processors.
//
{
  double cfl;
  double *h;
  ofstream h_file;
  double *h_new;
  int i;
  int j;
  int j_min = 0;
  int j_max = 400;
  double k = 0.002;
  int n = 11;
  MPI_Status status;
  int tag;
  double time;
  double time_delta;
  double time_max = 10.0;
  double time_min = 0.0;
  double time_new;
  double *x;
  double x_delta;
  ofstream x_file;
  double x_max = 1.0;
  double x_min = 0.0;
//
//  Have process 0 print out some information.
//
  if ( id == 0 )
  {
    cout << "\n";
    cout << "  Compute an approximate solution to the time dependent\n";
    cout << "  one dimensional heat equation:\n";
    cout << "\n";
    cout << "    dH/dt - K * d2H/dx2 = f(x,t)\n";
    cout << "\n";
    cout << "  for " << x_min << " = x_min < x < x_max = " << x_max << "\n";
    cout << "\n";
    cout << "  and " << time_min << " = time_min < t <= t_max = " << time_max << "\n";
    cout << "\n";
    cout << "  Boundary conditions are specified at x_min and x_max.\n";
    cout << "  Initial conditions are specified at time_min.\n";
    cout << "\n";
    cout << "  The finite difference method is used to discretize the\n";
    cout << "  differential equation.\n";
    cout << "\n";
    cout << "  This uses " << p * n << " equally spaced points in X\n";
    cout << "  and " << j_max << " equally spaced points in time.\n";
    cout << "\n";
    cout << "  Parallel execution is done using " << p << " processors.\n";
    cout << "  Domain decomposition is used.\n";
    cout << "  Each processor works on " << n << " nodes, \n";
    cout << "  and shares some information with its immediate neighbors.\n";
  }
//
//  Set the X coordinates of the N nodes.
//  We don't actually need ghost values of X but we'll throw them in
//  as X[0] and X[N+1].
//
```

```
//
  x = new double[n+2];

  for ( i = 0; i <= n + 1; i++ )
  {
    x[i] = ( ( double ) (         id * n + i - 1 ) * x_max
           + ( double ) ( p * n - id * n - i     ) * x_min )
           / ( double ) ( p * n              - 1 );
  }
//
//  In single processor mode, write out the X coordinates for display.
//
  if ( p == 1 )
  {
    x_file.open ( "x_data.txt" );
    for ( i = 1; i <= n; i++ )
    {
      x_file << "  " << x[i];
    }
    x_file << "\n";

    x_file.close ( );
  }
//
//  Set the values of H at the initial time.
//
  time = time_min;
  h = new double[n+2];
  h_new = new double[n+2];
  h[0] = 0.0;
  for ( i = 1; i <= n; i++ )
  {
    h[i] = initial_condition ( x[i], time );
  }
  h[n+1] = 0.0;

  time_delta = ( time_max - time_min ) / ( double ) ( j_max - j_min );
  x_delta = ( x_max - x_min ) / ( double ) ( p * n - 1 );
//
//  Check the CFL condition, have processor 0 print out its value,
//  and quit if it is too large.
//
  cfl = k * time_delta / x_delta / x_delta;

  if ( id == 0 )
  {
    cout << "\n";
    cout << "UPDATE\n";
    cout << "  CFL stability criterion value = " << cfl << "\n";;
  }

  if ( 0.5 <= cfl )
  {
    if ( id == 0 )
    {
      cout << "\n";
      cout << "UPDATE - Warning!\n";
      cout << "  Computation cancelled!\n";
      cout << "  CFL condition failed.\n";
      cout << "  0.5 <= K * dT / dX / dX = " << cfl << "\n";
    }
    return;
  }
//
//  In single processor mode, write out the values of H.
//
  if ( p == 1 )
  {
```

```
  {
    h_file.open ( "h_data.txt" );

    for ( i = 1; i <= n; i++ )
    {
      h_file << "  " << h[i];
    }
    h_file << "\n";
  }
//
//  Compute the values of H at the next time, based on current data.
//
  for ( j = 1; j <= j_max; j++ )
  {

    time_new = ( ( double ) (         j - j_min ) * time_max
               + ( double ) ( j_max - j         ) * time_min )
               / ( double ) ( j_max     - j_min );
//
//  Send H[1] to ID-1.
//
    if ( 0 < id )
    {
      tag = 1;
      MPI_Send ( &h[1], 1, MPI_DOUBLE, id-1, tag, MPI_COMM_WORLD );
    }
//
//  Receive H[N+1] from ID+1.
//
    if ( id < p-1 )
    {
      tag = 1;
      MPI_Recv ( &h[n+1], 1,  MPI_DOUBLE, id+1, tag, MPI_COMM_WORLD, &status );
    }
//
//  Send H[N] to ID+1.
//
    if ( id < p-1 )
    {
      tag = 2;
      MPI_Send ( &h[n], 1, MPI_DOUBLE, id+1, tag, MPI_COMM_WORLD );
    }
//
//  Receive H[0] from ID-1.
//
    if ( 0 < id )
    {
      tag = 2;
      MPI_Recv ( &h[0], 1, MPI_DOUBLE, id-1, tag, MPI_COMM_WORLD, &status );
    }
//
//  Update the temperature based on the four point stencil.
//
    for ( i = 1; i <= n; i++ )
    {
      h_new[i] = h[i]
      + ( time_delta * k / x_delta / x_delta ) * ( h[i-1] - 2.0 * h[i] + h[i+1] )
      + time_delta * rhs ( x[i], time );
    }
//
//  H at the extreme left and right boundaries was incorrectly computed
//  using the differential equation.  Replace that calculation by
//  the boundary conditions.
//
    if ( 0 == id )
    {
      h_new[1] = boundary_condition ( x[1], time_new );
```

```
    }
    if ( id == p - 1 )
    {
      h_new[n] = boundary_condition ( x[n], time_new );
    }
//
//  Update time and temperature.
//
    time = time_new;

    for ( i = 1; i <= n; i++ )
    {
      h[i] = h_new[i];
    }
//
//  In single processor mode, add current solution data to output file.
//
    if ( p == 1 )
    {
      for ( i = 1; i <= n; i++ )
      {
        h_file << "  " << h[i];
      }
      h_file << "\n";
    }
  }

  if ( p == 1 )
  {
    h_file.close ( );
  }

  delete [] h;
  delete [] h_new;
  delete [] x;

  return;
}
//****************************************************************************80

double boundary_condition ( double x, double time )

//****************************************************************************80
//
//  Purpose:
//
//    BOUNDARY_CONDITION evaluates the boundary condition of the differential equation.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    23 April 2008
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Input, double X, TIME, the position and time.
//
//    Output, double BOUNDARY_CONDITION, the value of the boundary condition.
//
{
```

```
  double value;
//
//  Left condition:
//
  if ( x < 0.5 )
  {
    value = 100.0 + 10.0 * sin ( time );
  }
  else
  {
    value = 75.0;
  }
  return value;
}
//****************************************************************************80

double initial_condition ( double x, double time )

//****************************************************************************80
//
//  Purpose:
//
//    INITIAL_CONDITION evaluates the initial condition of the differential equation.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    23 April 2008
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Input, double X, TIME, the position and time.
//
//    Output, double INITIAL_CONDITION, the value of the initial condition.
//
{
  double value;

  value = 95.0;

  return value;
}
//****************************************************************************80

double rhs ( double x, double time )

//****************************************************************************80
//
//  Purpose:
//
//    RHS evaluates the right hand side of the differential equation.
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    23 April 2008
//
```

```
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    Input, double X, TIME, the position and time.
//
//    Output, double RHS, the value of the right hand side function.
//
{
  double value;

  value = 0.0;

  return value;
}
//****************************************************************************80

void timestamp ( )

//****************************************************************************80
//
//  Purpose:
//
//    TIMESTAMP prints the current YMDHMS date as a time stamp.
//
//  Example:
//
//    31 May 2001 09:45:54 AM
//
//  Licensing:
//
//    This code is distributed under the GNU LGPL license.
//
//  Modified:
//
//    08 July 2009
//
//  Author:
//
//    John Burkardt
//
//  Parameters:
//
//    None
//
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct std::tm *tm_ptr;
  size_t len;
  std::time_t now;

  now = std::time ( NULL );
  tm_ptr = std::localtime ( &now );

  len = std::strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm_ptr );

  std::cout << time_buffer << "\n";

  return;
# undef TIME_SIZE
}
```