```
/*
 * Quake benchmark
 * Loukas Kallivokas and David O'Hallaron
 * Carnegie Mellon University, November, 1997
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <iostream>
#include "loop_speculator.h"
using namespace std;

#define AMAX_NAME  128

#ifndef PI
#  define PI 3.141592653589793238
#endif

struct options {
    int quiet;    /* run quietly unless there are errors (-Q) */
    int help;     /* do we want to print a help message (-h -H) */
};

struct excitation {

  double dt;       /* time step */
  double duration; /* total duration */
  double t0;       /* rise time */

};

struct damping {

  double zeta, consta, constb, freq;

};

struct properties {

  double cp;        /* compressional wave velocity */
  double cs;        /* shear wave velocity */
  double den;       /* density */

};

struct source {

  double dip, strike, rake, fault;
  double xyz[3];
  double epixyz[3];
  int sourcenode;
  int epicenternode;

};


/* name of the main program */

char *progname;

struct options options;

/*loop speculator declared globally*/
loop_speculator time_integration_loop_in_batches, time_integration_loop_in_batches_2, time_integration_loop_in_batches_smvp; //[3017];

/* global packfile variables */

FILE *packfile;

/* global Archimedes variables */

int ARCHnodes;
int ARCHpriv;
int ARCHmine;
int ARCHelems;
int ARCHglobalnodes;
int ARCHmesh_dim;
int ARCHglobalelems;
int ARCHcorners;
int ARCHsubdomains;
double ARCHduration;
int ARCHmatrixlen;
int ARCHcholeskylen;

int *ARCHglobalnode;
```

```c
int *ARCHglobalelem;
double **ARCHcoord;
int **ARCHvertex;
int *ARCHmatrixcol;
int *ARCHmatrixindex;

/* functions */

void arch_init(int argc, char **argv, struct options *op);
void mem_init(void);
void arch_readnodevector(double *v, int n);
void slip(double *u, double *v, double *w);
double distance(double p1[], double p2[]);
void centroid(double x[][3], double xc[]);
double point2fault(double x[]);
void abe_matrix(double vertices[][3], int bv[],
                struct properties *prop, double Ce[]);
void element_matrices(double vertices[][3], struct properties *prop,
                      double Ke[][12], double Me[]);
void vv12x12(double v1[], double v2[], double u[]);
void mv12x12(double m[][12], double v[]);
double phi0(double t);
double phi1(double t);
double phi2(double t);

/* new functions to support speculation*/
void* initialize (void* arg);
void* second_part_of_time_integration_loop (void* arg);
void* smvp_for_spec (void* arg);

/* global simulation variables */

int *nodekind;
double *nodekindf;
int *source_elms;
double **M, **C, **M23, **C23, **V23, **vel;
double ***disp, ***K;
int disptplus, dispt, disptminus;
double sim_time;
pthread_mutex_t col_mutex[30169];

struct source Src;
struct excitation Exc;
struct damping Damp;

/*--------------------------------------------------------------------------*/

int main(int argc, char **argv)
{
  int i, j, k, ii, jj, kk, iter, timesteps;
  int verticesonbnd;
  int cor[4], bv[4];
  int Step_stride;

  double Ke[12][12], Me[12], Ce[12], Mexv[12], Cexv[12], v[12];
  double alpha, c0[3], d1, d2, bigdist1, bigdist2, xc[3], uf[3];
  double vertices[4][3];

  struct properties prop;

/* NOTE: There are 5 possible flag values for the node data:

           1 if the node is in the interior
           4 if the node is on a x=const boundary surface
           5 if the node is on a y=const boundary surface
           6 if the node is on the bottom surface (z=z_lower)
           3 if the node is on the surface (z=0) (but not along the edges)
*/

/*--------------------------------------------------------------------------*/

/* Read in data from the pack file */

  arch_init(argc, argv, &options);

/* Dynamic memory allocations and initializations */

  mem_init();

  arch_readnodevector(nodekindf, ARCHnodes);

  fprintf(stderr, "%s: Beginning simulation.\n", argv[0]);

/* Excitation characteristics */

  Exc.dt = 0.0024;
```

```c
  Exc.duration = ARCHduration;
  Exc.t0 = 0.6;
  timesteps = Exc.duration / Exc.dt + 1;

/* Damping characteristics */

  Damp.zeta = 30.0;
  Damp.consta = 0.00533333;
  Damp.constb = 0.06666667;
  Damp.freq = 0.5;

/* Source characteristics */

  Src.strike = 111.0 * PI / 180.0;
  Src.dip = 44.0 * PI / 180.0;
  Src.rake = 70.0 * PI / 180.0;
  Src.fault = 29.640788;
  Src.xyz[0] = 32.264153;
  Src.xyz[1] = 23.814432;
  Src.xyz[2] = - 11.25;
  Src.epixyz[0] = Src.xyz[0];
  Src.epixyz[1] = Src.xyz[1];
  Src.epixyz[2] = 0.0;
  Src.sourcenode = - 1;
  Src.epicenternode = - 1;

/* Prescribe slip motion */

  uf[0] = uf[1] = uf[2] = 0.0;
  slip(&uf[0], &uf[1], &uf[2]);
  uf[0] *= Src.fault;
  uf[1] *= Src.fault;
  uf[2] *= Src.fault;

/* Soil properties (homogeneous material) */

  prop.cp = 6.0;
  prop.cs = 3.2;
  prop.den = 2.0;

/* Output frequency parameter */

  Step_stride = 30;

  disptplus = 0;
  dispt = 1;
  disptminus = 2;

/* Case info */

  fprintf(stderr, "\n");
  fprintf(stderr, "CASE SUMMARY\n");
  fprintf(stderr, "Fault information\n");
  fprintf(stderr, "  Orientation:  strike: %f\n", Src.strike);
  fprintf(stderr, "                   dip: %f\n", Src.dip);
  fprintf(stderr, "                  rake: %f\n", Src.rake);
  fprintf(stderr, "           dislocation: %f cm\n", Src.fault);
  fprintf(stderr, "Hypocenter: (%f, %f, %f) Km\n",
          Src.xyz[0], Src.xyz[1], Src.xyz[2]);
  fprintf(stderr, "Excitation characteristics\n");
  fprintf(stderr, "     Time step: %f sec\n", Exc.dt);
  fprintf(stderr, "      Duration: %f sec\n", Exc.duration);
  fprintf(stderr, "     Rise time: %f sec\n", Exc.t0);
  fprintf(stderr, "\n");
  fflush(stderr);

/* Redefine nodekind to be 1 for all surface nodes */

  for (i = 0; i < ARCHnodes; i++) {
    nodekind[i] = (int) nodekindf[i];
    pthread_mutex_init(&col_mutex[i], NULL);
  }

/* Search for the node closest to the point source (hypocenter) and */
/*        for the node closest to the epicenter */

  bigdist1 = 1000000.0;
  bigdist2 = 1000000.0;

  for (i = 0; i < ARCHnodes; i++) {
    c0[0] = ARCHcoord[i][0];
    c0[1] = ARCHcoord[i][1];
    c0[2] = ARCHcoord[i][2];
    d1 = distance(c0, Src.xyz);
    d2 = distance(c0, Src.epixyz);
```

```
    if (d1 < bigdist1) {
      bigdist1 = d1;
      Src.sourcenode = i;
    }

    if (d2 < bigdist2) {
      bigdist2 = d2;
      Src.epicenternode = i;
    }

  }

  if (Src.sourcenode != 0 && Src.sourcenode <= ARCHmine) {
    fprintf(stderr, "The source is node %d at (%f  %f  %f)\n",
            ARCHglobalnode[Src.sourcenode],
            ARCHcoord[Src.sourcenode][0],
            ARCHcoord[Src.sourcenode][1],
            ARCHcoord[Src.sourcenode][2]);
    fflush(stderr);
  }

  if (Src.epicenternode != 0 && Src.epicenternode <= ARCHmine) {
    fprintf(stderr, "The epicenter is node %d at (%f  %f  %f)\n",
            ARCHglobalnode[Src.epicenternode],
            ARCHcoord[Src.epicenternode][0],
            ARCHcoord[Src.epicenternode][1],
            ARCHcoord[Src.epicenternode][2]);
    fflush(stderr);
  }

/* Search for all the elements that contain the source node */

  if (Src.sourcenode != 0) {

    for (i = 0; i < ARCHelems; i++) {
      for (j = 0; j < 4; j++)
        cor[j] = ARCHvertex[i][j];

      if (cor[0] == Src.sourcenode || cor[1] == Src.sourcenode ||
          cor[2] == Src.sourcenode || cor[3] == Src.sourcenode) {

        for (j = 0; j < 4; j++)
          for (k = 0; k < 3; k++)
            vertices[j][k] = ARCHcoord[cor[j]][k];

        centroid(vertices, xc);

        source_elms[i] = 2;
        if (point2fault(xc) >= 0)
          source_elms[i] = 3;

      }
    }
  }

/* Simulation */

  for (i = 0; i < ARCHelems; i++) {
    for (j = 0; j < 12; j++) {
      Me[j] = 0.0;
      Ce[j] = 0.0;
      v[j] = 0.0;
      for (k = 0; k < 12; k++)
        Ke[j][k] = 0.0;
    }

    for (j = 0; j < 4; j++) {
      cor[j] = ARCHvertex[i][j];
    }

    verticesonbnd = 0;
    for (j = 0; j < 4; j++)
      if (nodekind[cor[j]] != 1)
        bv[verticesonbnd++] = j;

    /*
    if (verticesonbnd == 4) {
      fprintf (stderr, "Warning! 4 vertices seem to be on the boundary\n");
      for (j = 0; j < 4; j++)
        fprintf (stderr, "%f %f %f nodekind[cor[%d]]=%d\n",
                 ARCHcoord[cor[bv[j]]][0],
                 ARCHcoord[cor[bv[j]]][1],
                 ARCHcoord[cor[bv[j]]][2],j,nodekind[cor[j]]);
    }
    */

    if (verticesonbnd == 3) {
```

```
    for (j = 0; j < 3; j++)
      for (k = 0; k < 3; k++)
        vertices[j][k] = ARCHcoord[cor[bv[j]]][k];

    abe_matrix(vertices, bv, &prop, Ce);

  }

  for (j = 0; j < 4; j++)
    for (k = 0; k < 3; k++)
      vertices[j][k] = ARCHcoord[cor[j]][k];

  element_matrices(vertices, &prop, Ke, Me);

  /* Damping (proportional) */

  centroid(vertices, xc);

  alpha = 4.0 * PI * Damp.consta * 0.95 / (prop.cs + Damp.constb);

  for (j = 0; j < 12; j++)
    Ce[j] = Ce[j] + alpha * Me[j];

  /* Source mechanism */

  if (source_elms[i] == 2 || source_elms[i] == 3) {

    for (j = 0; j < 4; j++) {

      if (cor[j] == Src.sourcenode) {

        v[3 * j] = uf[0];
        v[3 * j + 1] = uf[1];
        v[3 * j + 2] = uf[2];

      } else {

        v[3 * j] = 0;
        v[3 * j + 1] = 0;
        v[3 * j + 2] = 0;

      }
    }

    vv12x12(Me, v, Mexv);
    vv12x12(Ce, v, Cexv);
    mv12x12(Ke, v);

    if (source_elms[i] == 3)
      for (j = 0; j < 12; j++) {
        v[j] = - v[j];
        Mexv[j] = - Mexv[j];
        Cexv[j] = - Cexv[j];
      }

    /* Assemble vectors3 V23, M23, C23 */

    for (j = 0; j < 4; j++) {
      V23[ARCHvertex[i][j]][0] += v[j * 3];
      V23[ARCHvertex[i][j]][1] += v[j * 3 + 1];
      V23[ARCHvertex[i][j]][2] += v[j * 3 + 2];
      M23[ARCHvertex[i][j]][0] += Mexv[j * 3];
      M23[ARCHvertex[i][j]][1] += Mexv[j * 3 + 1];
      M23[ARCHvertex[i][j]][2] += Mexv[j * 3 + 2];
      C23[ARCHvertex[i][j]][0] += Cexv[j * 3];
      C23[ARCHvertex[i][j]][1] += Cexv[j * 3 + 1];
      C23[ARCHvertex[i][j]][2] += Cexv[j * 3 + 2];
    }

  }

  /* Assemble vectors3 Me, Ce and matrix3 Ke */

  for (j = 0; j < 4; j++) {
    M[ARCHvertex[i][j]][0] += Me[j * 3];
    M[ARCHvertex[i][j]][1] += Me[j * 3 + 1];
    M[ARCHvertex[i][j]][2] += Me[j * 3 + 2];
    C[ARCHvertex[i][j]][0] += Ce[j * 3];
    C[ARCHvertex[i][j]][1] += Ce[j * 3 + 1];
    C[ARCHvertex[i][j]][2] += Ce[j * 3 + 2];
    for (k = 0; k < 4; k++) {
      if (ARCHvertex[i][j] <= ARCHvertex[i][k]) {
        kk = ARCHmatrixindex[ARCHvertex[i][j]];
        while (ARCHmatrixcol[kk] != ARCHvertex[i][k]) {
          kk++;
        }
        for (ii = 0; ii < 3; ii++)
```

```
          for (jj = 0; jj < 3; jj++)
            K[kk][ii][jj] += Ke[j * 3 + ii][k * 3 + jj];
        }
      }
    }
  }

/* Time integration loop */

  fprintf(stderr, "\n");

  script_vector input_functions;

  vector <void*> input_vars;
  input_vars.push_back((void*)0); //Input for first iteration


  for (iter = 1; iter <= timesteps; iter++) {
        input_vars[0]=(void*)0; //Input for first iteration
        input_functions.clear();
        input_functions.push_back(initialize);
        time_integration_loop_in_batches.run(input_functions, input_vars);
        time_integration_loop_in_batches.append(initialize, (void*)1);
        time_integration_loop_in_batches.append(initialize, (void*)2);
        time_integration_loop_in_batches.append(initialize, (void*)3);
        time_integration_loop_in_batches.commit();
        time_integration_loop_in_batches.get_results();


        input_functions.clear();
        input_functions.push_back(smvp_for_spec);
//      void * ptr_to_disp=&disp[0][0][0];
/*      for (int input_var=0; input_var<ARCHnodes+10000; input_var+=10000){
                if (input_var<ARCHnodes){
                        input_vars[0]=(void*)input_var;
                        time_integration_loop_in_batches_smvp.run((void*&)ptr_to_disp, input_functions, input_vars); //[input_var/10]
                        for (int aux_i=(input_var)+1000; aux_i<(input_var+10000); aux_i+=1000){
                                if (aux_i<ARCHnodes){
                                        while (time_integration_loop_in_batches_smvp.append(smvp_for_spec, (void*)aux_i)!=0){ //[input_var
                                                cout<<"APPEND ERROR"<<input_var<<endl;
                                        }
                                }
                        }
                        time_integration_loop_in_batches_smvp.commit(); //[input_var/10]
                        while (time_integration_loop_in_batches_smvp.get_results()!=0){ //[input_var/10]
                                cout<<"GET RESULTS ERROR"<<input_var<<endl;
                        }
                }
                else{
                        input_var=ARCHnodes+5000;
                }
        }*/

        time_integration_loop_in_batches_smvp.run(input_functions, input_vars); //Iterations 0-7541
//      for (int aux_k = 10000; aux_k < ARCHNodes; aux_k+=10000){
                time_integration_loop_in_batches_smvp.append(smvp_for_spec, (void*)7542);//Iterations 7542-15083
                time_integration_loop_in_batches_smvp.append(smvp_for_spec, (void*)15084);//Iterations 15084-22625
                time_integration_loop_in_batches_smvp.append(smvp_for_spec, (void*)22626);//Iterations 22626-30618
//      }
//      smvp_for_spec((void*)22626);
        time_integration_loop_in_batches_smvp.commit();
        time_integration_loop_in_batches_smvp.get_results();



        sim_time = iter * Exc.dt;

        input_functions.clear();
        input_functions.push_back(second_part_of_time_integration_loop);
        input_vars.clear();
        input_vars.push_back((void*)(0));

        time_integration_loop_in_batches_2.run(input_functions, input_vars);

        time_integration_loop_in_batches_2.append(second_part_of_time_integration_loop, (void*)1);
        time_integration_loop_in_batches_2.append(second_part_of_time_integration_loop, (void*)2);
        time_integration_loop_in_batches_2.append(second_part_of_time_integration_loop, (void*)3);
        time_integration_loop_in_batches_2.commit();
        time_integration_loop_in_batches_2.get_results();

    /* Print out the response at the source and epicenter nodes */

    if (iter % Step_stride == 0) {

      fprintf(stderr, "Time step %d\n", iter);
```

```c
      if (Src.sourcenode <= ARCHmine)
        printf("%d: %.2e %.2e %.2e\n", ARCHglobalnode[Src.sourcenode],
               disp[disptplus][Src.sourcenode][0],
               disp[disptplus][Src.sourcenode][1],
               disp[disptplus][Src.sourcenode][2]);

      if (Src.epicenternode <= ARCHmine)
        printf("%d: %.2e %.2e %.2e\n", ARCHglobalnode[Src.epicenternode],
               disp[disptplus][Src.epicenternode][0],
               disp[disptplus][Src.epicenternode][1],
               disp[disptplus][Src.epicenternode][2]);

      fflush(stdout);
    }

    i = disptminus;
    disptminus = dispt;
    dispt = disptplus;
    disptplus = i;

  }

  fprintf(stderr, "%s: %d nodes %d elems %d timesteps\n",
          progname, ARCHglobalnodes, ARCHglobalelems, timesteps);
  fprintf(stderr, "\n");
  fflush(stderr);

  if (!options.quiet) {
    fprintf(stderr, "%s: Done. Terminating the simulation.\n", progname);
  }

  return 0;
}
/* --------------------------------------------------------------------------*/


/* --------------------------------------------------------------------------*/
/* Compute shape function derivatives                                        */
/* N_1 = 1 - r - s - t                                                       */
/* N_2 = r                                                                   */
/* N_3 = s                                                                   */
/* N_4 = t                                                                   */

void shape_ders(double ds[][4])
{
  ds[0][0] = - 1;
  ds[1][0] = - 1;
  ds[2][0] = - 1;
  ds[0][1] = 1;
  ds[1][1] = 0;
  ds[2][1] = 0;
  ds[0][2] = 0;
  ds[1][2] = 1;
  ds[2][2] = 0;
  ds[0][3] = 0;
  ds[1][3] = 0;
  ds[2][3] = 1;
}

/* --------------------------------------------------------------------------*/
/* Calculate Young's modulus E and Poisson's ratio nu,                       */
/* given a pair of compressional (cp) and shear (cs) wave velocities         */

void get_Enu(struct properties *prop, double *E, double *nu)
{
  double ratio;

  ratio = prop->cp / prop->cs;
  ratio = ratio * ratio;
  *nu = 0.5 * (ratio - 2.0) / (ratio - 1.0);
  *E = 2.0 * prop->den * prop->cs * prop->cs * (1.0 + *nu);
}
/* --------------------------------------------------------------------------*/


/* --------------------------------------------------------------------------*/
/* Calculate the inverse and the determinant of the Jacobian,                */
/* given the Jacobian                                                        */
/* (a on input holds the Jacobian and on output its inverse                  */

void inv_J(double a[][3], double *det)
{
  double d1;
  double c[3][3];
  int i, j;
```

```c
  c[0][0] = a[1][1] * a[2][2] - a[2][1] * a[1][2];
  c[0][1] = a[0][2] * a[2][1] - a[0][1] * a[2][2];
  c[0][2] = a[0][1] * a[1][2] - a[0][2] * a[1][1];
  c[1][0] = a[1][2] * a[2][0] - a[1][0] * a[2][2];
  c[1][1] = a[0][0] * a[2][2] - a[0][2] * a[2][0];
  c[1][2] = a[0][2] * a[1][0] - a[0][0] * a[1][2];
  c[2][0] = a[1][0] * a[2][1] - a[1][1] * a[2][0];
  c[2][1] = a[0][1] * a[2][0] - a[0][0] * a[2][1];
  c[2][2] = a[0][0] * a[1][1] - a[0][1] * a[1][0];
  *det = a[0][0] * c[0][0] + a[0][1] * c[1][0] + a[0][2] * c[2][0];
  d1 = 1.0 / *det;
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      a[i][j] = c[i][j] * d1;
}
/* ------------------------------------------------------------------------*/


/* ------------------------------------------------------------------------*/
/* Calculate the element stiffness (Ke[12][12]) and                        */
/*           the element mass matrices (Me[12]),                           */
/* given the four vertices of a tetrahedron                                */

void element_matrices(double vertices[][3], struct properties *prop, double Ke[][12], double Me[])
{
  double ds[3][4];
  double sum[3];
  double jacobian[3][3];
  double det;
  double volume;
  double E, nu;
  double c1, c2, c3;
  double tt, ts;
  int i, j, m, n, row, column;

  shape_ders(ds);

  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++) {
      sum[0] = 0.0;
      for (m = 0; m < 4; m++)
        sum[0] = sum[0] + ds[i][m] * vertices[m][j];
      jacobian[j][i] = sum[0];         /* compute Jacobian */
    }

  inv_J(jacobian, &det);               /* compute J^-1 & its determinant */

  for (m = 0; m < 4; m++) {

    for (i = 0; i < 3; i++) {
      sum[i] = 0.0;
      for (j = 0; j < 3; j++)
        sum[i] = sum[i] + jacobian[j][i] * ds[j][m];
    }

    for (i = 0; i < 3; i++)
      ds[i][m] = sum[i];

  }

  volume = det / 6.0;

  if (volume <= 0) {
    fprintf(stderr, "Warning: Element volume = %f !\n", volume);
  }

  get_Enu(prop, &E, &nu);

  c1 = E / (2.0 * (nu + 1.0) * (1.0 - nu * 2.0)) * volume;
  c2 = E * nu / ((nu + 1.0) * (1.0 - nu * 2.0)) * volume;
  c3 = E / ((nu + 1.0) * 2.0) * volume;

  row = - 1;

  for (m = 0; m < 4; m++) {            /* lower triangular stiffness matrix */
    for (i = 0; i < 3; ++i) {
      ++row;
      column = - 1;
      for (n = 0; n <= m; n++) {
        for (j = 0; j < 3; j++) {
          ++column;
          ts = ds[i][m] * ds[j][n];
          if (i == j) {
            ts = ts * c1;
            tt = (ds[0][m] * ds[0][n] +
                  ds[1][m] * ds[1][n] +
```

```c
                  ds[2][m] * ds[2][n]) * c3;
          }
          else {
            if (m == n) {
              ts = ts * c1;
              tt = 0;
            }
            else {
              ts = ts * c2;
              tt = ds[j][m] * ds[i][n] * c3;
            }
          }
          Ke[row][column] = Ke[row][column] + ts + tt;
        }
      }
    }
  }
  tt = prop->den * volume / 4.0;
  for (i = 0; i < 12; i++)
    Me[i] = tt;

  for (i = 0; i < 12; i++)
    for (j = 0; j <= i; j++)
      Ke[j][i] = Ke[i][j];
}
/* --------------------------------------------------------------------------*/


/* --------------------------------------------------------------------------*/
/* Calculate the area of a triangle given the coordinates of                 */
/* its three vertices                                                        */

double area_triangle(double vertices[][3])
{
  double a, b, c;
  double x2, y2, z2;
  double p;
  double area;

  x2 = (vertices[0][0] - vertices[1][0]) * (vertices[0][0] - vertices[1][0]);
  y2 = (vertices[0][1] - vertices[1][1]) * (vertices[0][1] - vertices[1][1]);
  z2 = (vertices[0][2] - vertices[1][2]) * (vertices[0][2] - vertices[1][2]);
  a = sqrt(x2 + y2 + z2);

  x2 = (vertices[2][0] - vertices[1][0]) * (vertices[2][0] - vertices[1][0]);
  y2 = (vertices[2][1] - vertices[1][1]) * (vertices[2][1] - vertices[1][1]);
  z2 = (vertices[2][2] - vertices[1][2]) * (vertices[2][2] - vertices[1][2]);
  b = sqrt(x2 + y2 + z2);

  x2 = (vertices[0][0] - vertices[2][0]) * (vertices[0][0] - vertices[2][0]);
  y2 = (vertices[0][1] - vertices[2][1]) * (vertices[0][1] - vertices[2][1]);
  z2 = (vertices[0][2] - vertices[2][2]) * (vertices[0][2] - vertices[2][2]);
  c = sqrt(x2 + y2 + z2);

  p = (a + b + c) / 2.0;

  area = sqrt(p * (p - a) * (p - b) * (p - c));

  return area;

}
/* --------------------------------------------------------------------------*/


/* --------------------------------------------------------------------------*/
/* Generate the element damping matrix for the absorbing boundary            */
/* (triangular plane element)                                                */

void abe_matrix(double vertices[][3], int bv[], struct properties *prop, double Ce[])
{
  int i, j;
  double area;

  area = area_triangle(vertices);

  for (i = 0; i < 3; i++) {
    j = 3 * bv[i];
    Ce[j] = Ce[j] + prop->cs * prop->den * area / 3.0;
    Ce[j + 1] = Ce[j + 1] + prop->cs * prop->den * area / 3.0;
    Ce[j + 2] = Ce[j + 2] + prop->cp * prop->den * area / 3.0;
  }

}
/* --------------------------------------------------------------------------*/

/* --------------------------------------------------------------------------*/
```

```c
/* Excitation (ramp function)                                                */

double phi0(double t)
{
  double value;

  if (t <= Exc.t0) {

    value = 0.5 / PI * (2.0 * PI * t / Exc.t0 - sin(2.0 * PI * t / Exc.t0));
    return value;

  }
  else
    return 1.0;

}
/* ------------------------------------------------------------------------*/


/* ------------------------------------------------------------------------*/
/* First derivative of the excitation (velocity of ramp function)           */

double phi1(double t)
{
  double value;

  if (t <= Exc.t0) {

    value = (1.0 - cos(2.0 * PI * t / Exc.t0)) / Exc.t0;
    return value;

  }
  else
    return 0.0;
}
/* ------------------------------------------------------------------------*/


/* ------------------------------------------------------------------------*/
/* Second derivative of the excitation (acceleration of ramp function)      */

double phi2(double t)
{
  double value;

  if (t <= Exc.t0) {

    value = 2.0 * PI / Exc.t0 / Exc.t0 * sin(2.0 * PI * t / Exc.t0);
    return value;

  }
  else
    return 0.0;
}
/* ------------------------------------------------------------------------*/


/* ------------------------------------------------------------------------*/
/* Calculate the slip motion at the source node                             */

void slip(double *u, double *v, double *w)
{
  *u = *v = *w = 0.0;
  *u = (cos(Src.rake) * sin(Src.strike) -
        sin(Src.rake) * cos(Src.strike) * cos(Src.dip));
  *v = (cos(Src.rake) * cos(Src.strike) +
        sin(Src.rake) * sin(Src.strike) * cos(Src.dip));
  *w = sin(Src.rake) * sin(Src.dip);
}
/* ------------------------------------------------------------------------*/


/* ------------------------------------------------------------------------*/
/* Calculate the distance between two points p1 and p2                      */

double distance(double p1[], double p2[])
{
  return ((p1[0] - p2[0]) * (p1[0] - p2[0]) +
          (p1[1] - p2[1]) * (p1[1] - p2[1]) +
          (p1[2] - p2[2]) * (p1[2] - p2[2]));
}
/* ------------------------------------------------------------------------*/


/* ------------------------------------------------------------------------*/
/* Calculate the centroid of a tetrahedron                                  */
```

```
void centroid(double x[][3], double xc[])
{

  int i;

  for (i = 0; i < 3; i++)
    xc[i] = (x[0][i] + x[1][i] + x[2][i] + x[3][i]) / 4.0;

}
/* --------------------------------------------------------------------------*/


/* --------------------------------------------------------------------------*/
/* Calculate the distance to the fault from a given point x                 */

double point2fault(double x[])
{

  double nx, ny, nz;
  double d0;

  nx = cos(Src.strike) * sin(Src.dip);
  ny = - sin(Src.strike) * sin(Src.dip);
  nz = cos(Src.dip);

  d0 = - (nx * Src.xyz[0] + ny * Src.xyz[1] + nz * Src.xyz[2]);

  return (double) nx * x[0] + ny * x[1] + nz * x[2] + d0;
}
/* --------------------------------------------------------------------------*/


/* --------------------------------------------------------------------------*/
/* Matrix (12x12) times vector (12x1) product                                */

void mv12x12(double m[][12], double v[])
{
  int i, j;
  double u[12];

  for (i = 0; i < 12; i++) {

    u[i] = 0;
    for (j = 0; j < 12; j++)
      u[i] += m[i][j] * v[j];
  }

  for (i = 0; i < 12; i++)
    v[i] = u[i];

}
/* --------------------------------------------------------------------------*/


/* --------------------------------------------------------------------------*/
/* Vector (12x1) times vector (12x1) product                                 */

void vv12x12(double v1[], double v2[], double u[])
{
  int i;

  for (i = 0; i < 12; i++)
    u[i] = v1[i] * v2[i];

}

/*--------------------------------------------------------------------------*/
/* Graceful exit                                                            */

void arch_bail(void) {
    exit(0);
}
/*--------------------------------------------------------------------------*/


/*--------------------------------------------------------------------------*/
void arch_info(void)
{
    printf("\n");
    printf("You are running an Archimedes finite element simulation called %s.\n\n", progname);
    printf("The command syntax is:\n\n");
    printf("%s [-Qh] < packfile\n\n", progname);
    printf("Command line options:\n\n");
    printf("    -Q  Quietly suppress all explanation of what this program is doing\n");
    printf("        unless an error occurs.\n");
```

```c
    printf("    -h  Print this message and exit.\n");
}
/*--------------------------------------------------------------------------*/


/*--------------------------------------------------------------------------*/
/*
 * arch_parsecommandline - parse the command line
 */
void arch_parsecommandline(int argc, char **argv, struct options *op)
{
    int i, j;

    /* first set up the defaults */
    op->quiet = 0;
    op->help = 0;

    /* now see if the user wants to change any of these */
    for (i=1; i<argc; i++) {
        if (argv[i][0] == '-') {
            for (j = 1; argv[i][j] != '\0'; j++) {
                if (argv[i][j] == 'Q') {
                    op->quiet = 1;
                }
                if ((argv[i][j] == 'h' ||argv[i][j] == 'H')) {
                    op->help = 1;
                }
            }
        }
    }
    if (op->help) {
        arch_info();
        exit(0);
    }
}
/*--------------------------------------------------------------------------*/


/*--------------------------------------------------------------------------*/
/*
 * arch_readnodevector - read a vector of nodal data from the pack file
 *                        called by READNODEVECTOR.stub
 */
void arch_readnodevector(double *v, int n) {
    int i;
    int type, attributes;

    fscanf(packfile, "%d %d\n", &type, &attributes);

    if (type != 2) {
        fprintf(stderr,
                "READNODEVECTOR: unexpected data type\n");
        arch_bail();
    }
    if (attributes != 1) {
        fprintf(stderr,
                "READNODEVECTOR: unexpected number of attributes\n");
        arch_bail();
    }
    for (i=0; i<n; i++) {
        fscanf(packfile, "%lf", &v[i]);
    }
}
/*--------------------------------------------------------------------------*/


/*--------------------------------------------------------------------------*/
/*
 * arch_readelemvector - read a vector of element data from the pack file
 *                        called by READELEMVECTOR.stub
 */
void arch_readelemvector(double *v, int n) {
    int i;
    int type, attributes;

    fscanf(packfile, "%d %d\n", &type, &attributes);
    if (type != 1) {
        fprintf(stderr,
                "READELEMVECTOR: unexpected data type\n");
        arch_bail();
    }
    if (attributes != 1) {
        fprintf(stderr,
                "READELEMVECTOR: unexpected number of attributes\n");
        arch_bail();
    }
```

```
    for (i=0; i<n; i++) {
        fscanf(packfile, "%lf", &v[i]);
    }
}
/*--------------------------------------------------------------------------*/


/*--------------------------------------------------------------------------*/
/*
 * arch_readdouble - read a floating point number from the pack file
 */
void arch_readdouble(double *v) {
    int type, attributes;

    fscanf(packfile, "%d %d\n", &type, &attributes);
    if (type != 3) {
        fprintf(stderr,
                "READDOUBLE: unexpected data type\n");
        arch_bail();
    }
    if (attributes != 1) {
        fprintf(stderr,
                "READDOUBLE: unexpected number of attributes\n");
        arch_bail();
    }
    fscanf(packfile, "%lf", &v[0]);
}
/*--------------------------------------------------------------------------*/


/*--------------------------------------------------------------------------*/
void readpackfile(FILE *packfile, struct options *op) {
  int oldrow, newrow;
  int i, j;
  int temp1, temp2;

  fscanf(packfile, "%d", &ARCHglobalnodes);
  fscanf(packfile, "%d", &ARCHmesh_dim);
  fscanf(packfile, "%d", &ARCHglobalelems);
  fscanf(packfile, "%d", &ARCHcorners);
  fscanf(packfile, "%d", &ARCHsubdomains);
  fscanf(packfile, "%lf", &ARCHduration);

  /* only one subdomain allowed */
  if (ARCHsubdomains != 1) {
    fprintf(stderr, "%s: too many subdomains(%d), rerun slice using -s1\n",
            progname, ARCHsubdomains);
    arch_bail();
  }

  /* read nodes */
  if (!op->quiet) {
    fprintf(stderr, "%s: Reading nodes.\n", progname);
  }

  fscanf(packfile, "%d %d %d", &ARCHnodes, &ARCHmine, &ARCHpriv);

  ARCHglobalnode = (int *) malloc(ARCHnodes * sizeof(int));
  if (ARCHglobalnode == (int *) NULL) {
    fprintf(stderr, "malloc failed for ARCHglobalnode\n");
    fflush(stderr);
    exit(0);
  }

  ARCHcoord = (double **) malloc(ARCHnodes * sizeof(double *));
  for (i = 0; i < ARCHnodes; i++)
    ARCHcoord[i] = (double *) malloc(3 * sizeof(double));

  for (i=0; i<ARCHnodes; i++) {
    fscanf(packfile, "%d", &ARCHglobalnode[i]);
    for (j=0; j<ARCHmesh_dim; j++) {
      fscanf(packfile, "%lf", &ARCHcoord[i][j]);
    }
  }

  /* read elements */
  if (!op->quiet)
    fprintf(stderr, "%s: Reading elements.\n", progname);

  fscanf(packfile, "%d", &ARCHelems);

  ARCHglobalelem = (int *) malloc(ARCHelems * sizeof(int));
  if (ARCHglobalelem == (int *) NULL) {
    fprintf(stderr, "malloc failed for ARCHglobalelem\n");
    fflush(stderr);
    exit(0);
```

```
  }

  ARCHvertex = (int **) malloc(ARCHelems * sizeof(int *));
  for (i = 0; i < ARCHelems; i++)
    ARCHvertex[i] = (int *) malloc(4 * sizeof(int));

  for (i=0; i<ARCHelems; i++) {
    fscanf(packfile, "%d", &ARCHglobalelem[i]);
    for (j=0; j<ARCHcorners; j++) {
      fscanf(packfile, "%d", &ARCHvertex[i][j]);
    }
  }

  /* read sparse matrix structure and convert from tuples to CSR */
  if (!op->quiet)
    fprintf(stderr, "%s: Reading sparse matrix structure.\n", progname);

  fscanf(packfile, "%d %d", &ARCHmatrixlen, &ARCHcholeskylen);

  ARCHmatrixcol = (int *) malloc((ARCHmatrixlen + 1) * sizeof(int));
  if (ARCHmatrixcol == (int *) NULL) {
    fprintf(stderr, "malloc failed for ARCHmatrixcol\n");
    fflush(stderr);
    exit(0);
  }

  ARCHmatrixindex = (int *) malloc((ARCHnodes + 1) * sizeof(int));
  if (ARCHmatrixindex == (int *) NULL) {
    fprintf(stderr, "malloc failed for ARCHmatrixindex\n");
    fflush(stderr);
    exit(0);
  }

  oldrow = -1;
  for (i = 0; i < ARCHmatrixlen; i++) {
    fscanf(packfile, "%d", &newrow);
    fscanf(packfile, "%d", &ARCHmatrixcol[i]);
    while (oldrow < newrow) {
      if (oldrow+1 >= ARCHnodes+1) {
        printf("%s: error: (1)idx buffer too small (%d >= %d)\n",
               progname, oldrow+1, ARCHnodes+1);
        arch_bail();
      }
      ARCHmatrixindex[++oldrow] = i;
    }
  }
  while (oldrow < ARCHnodes) {
    ARCHmatrixindex[++oldrow] = ARCHmatrixlen;
  }

  /* read comm info (which nodes are shared between subdomains) */
  fscanf(packfile, "%d %d", &temp1, &temp2);

}
/*--------------------------------------------------------------------------*/


/*--------------------------------------------------------------------------*/
/*
 * arch_init - initialize the Archimedes simulation
 *             called by ARCHIMEDES_INIT.stub
 */
void arch_init(int argc, char **argv, struct options *op)
{

  /* parse the command line options */
  progname = argv[0];
  arch_parsecommandline(argc, argv, op);

  /* read the pack file */
  packfile = stdin;
  readpackfile(packfile, op);

}
/*--------------------------------------------------------------------------*/

/*--------------------------------------------------------------------------*/
/* Dynamic memory allocations and initializations                          */

void mem_init(void) {

int i, j, k;

/* Node vector */

  nodekindf = (double *) malloc(ARCHnodes * sizeof(double));
```

```
  if (nodekindf == (double *) NULL) {
    fprintf(stderr, "malloc failed for nodekindf\n");
    fflush(stderr);
    exit(0);
  }

/* Node vector */

  nodekind = (int *) malloc(ARCHnodes * sizeof(int));
  if (nodekind == (int *) NULL) {
    fprintf(stderr, "malloc failed for nodekind\n");
    fflush(stderr);
    exit(0);
  }

/* Element vector */

  source_elms = (int *) malloc(ARCHelems * sizeof(int));
  if (source_elms == (int *) NULL) {
    fprintf(stderr, "malloc failed for source_elms\n");
    fflush(stderr);
    exit(0);
  }

/* Velocity array */

  vel = (double **) malloc(ARCHnodes * sizeof(double *));
  if (vel == (double **) NULL) {
    fprintf(stderr, "malloc failed for vel\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHnodes; i++) {
    vel[i] = (double *) malloc(3 * sizeof(double));
    if (vel[i] == (double *) NULL) {
      fprintf(stderr, "malloc failed for vel[%d]\n",i);
      fflush(stderr);
      exit(0);
    }
  }

/* Mass matrix */

  M = (double **) malloc(ARCHnodes * sizeof(double *));
  if (M == (double **) NULL) {
    fprintf(stderr, "malloc failed for M\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHnodes; i++) {
    M[i] = (double *) malloc(3 * sizeof(double));
    if (M[i] == (double *) NULL) {
      fprintf(stderr, "malloc failed for M[%d]\n",i);
      fflush(stderr);
      exit(0);
    }
  }

/* Damping matrix */

  C = (double **) malloc(ARCHnodes * sizeof(double *));
  if (C == (double **) NULL) {
    fprintf(stderr, "malloc failed for C\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHnodes; i++) {
    C[i] = (double *) malloc(3 * sizeof(double));
    if (C[i] == (double *) NULL) {
      fprintf(stderr, "malloc failed for C[%d]\n",i);
      fflush(stderr);
      exit(0);
    }
  }

/* Auxiliary mass matrix */

  M23 = (double **) malloc(ARCHnodes * sizeof(double *));
  if (M23 == (double **) NULL) {
    fprintf(stderr, "malloc failed for M23\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHnodes; i++) {
    M23[i] = (double *) malloc(3 * sizeof(double));
    if (M23[i] == (double *) NULL) {
      fprintf(stderr, "malloc failed for M23[%d]\n",i);
```

```
      fflush(stderr);
      exit(0);
    }
  }

/* Auxiliary damping matrix */

  C23 = (double **) malloc(ARCHnodes * sizeof(double *));
  if (C23 == (double **) NULL) {
    fprintf(stderr, "malloc failed for C23\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHnodes; i++) {
    C23[i] = (double *) malloc(3 * sizeof(double));
    if (C23[i] == (double *) NULL) {
      fprintf(stderr, "malloc failed for C23[%d]\n",i);
      fflush(stderr);
      exit(0);
    }
  }

/* Auxiliary vector */

  V23 = (double **) malloc(ARCHnodes * sizeof(double *));
  if (V23 == (double **) NULL) {
    fprintf(stderr, "malloc failed for V23\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHnodes; i++) {
    V23[i] = (double *) malloc(3 * sizeof(double));
    if (V23[i] == (double *) NULL) {
      fprintf(stderr, "malloc failed for V23[%d]\n",i);
      fflush(stderr);
      exit(0);
    }
  }

  /* Displacement array disp[3][ARCHnodes][3] */

  disp = (double ***) malloc(3 * sizeof(double **));
  if (disp == (double ***) NULL) {
    fprintf(stderr, "malloc failed for disp\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < 3; i++) {
    disp[i] = (double **) malloc(ARCHnodes * sizeof(double *));
    if (disp[i] == (double **) NULL) {
      fprintf(stderr, "malloc failed for disp[%d]\n",i);
      fflush(stderr);
      exit(0);
    }
    for (j = 0; j < ARCHnodes; j++) {
      disp[i][j] = (double *) malloc(3 * sizeof(double));
      if (disp[i][j] == (double *) NULL) {
        fprintf(stderr, "malloc failed for disp[%d][%d]\n",i,j);
        fflush(stderr);
        exit(0);
      }
    }
  }

  /* Stiffness matrix K[ARCHmatrixlen][3][3] */

  K = (double ***) malloc(ARCHmatrixlen * sizeof(double **));
  if (K == (double ***) NULL) {
    fprintf(stderr, "malloc failed for K\n");
    fflush(stderr);
    exit(0);
  }
  for (i = 0; i < ARCHmatrixlen; i++) {
    K[i] = (double **) malloc(3 * sizeof(double *));
    if (K[i] == (double **) NULL) {
      fprintf(stderr, "malloc failed for K[%d]\n",i);
      fflush(stderr);
      exit(0);
    }
    for (j = 0; j < 3; j++) {
      K[i][j] = (double *) malloc(3 * sizeof(double));
      if (K[i][j] == (double *) NULL) {
        fprintf(stderr, "malloc failed for K[%d][%d]\n",i,j);
        fflush(stderr);
        exit(0);
      }
    }
```

```
  }

  /* Initializations */

  for (i = 0; i < ARCHnodes; i++) {
    nodekind[i] = 0;
    for (j = 0; j < 3; j++) {
      M[i][j] = 0.0;
      C[i][j] = 0.0;
      M23[i][j] = 0.0;
      C23[i][j] = 0.0;
      V23[i][j] = 0.0;
      disp[0][i][j] = 0.0;
      disp[1][i][j] = 0.0;
      disp[2][i][j] = 0.0;
    }
  }

  for (i = 0; i < ARCHelems; i++) {
    source_elms[i] = 1;
  }

  for (i = 0; i < ARCHmatrixlen; i++) {
    for (j = 0; j < 3; j++) {
      for (k = 0; k < 3; k++) {
        K[i][j][k] = 0.0;
      }
    }
  }
}
/*---------------------------------------------------------------------------*/
/*Functions optimized to be used with speculation*/

void* initialize (void* arg){
        int j=(int)arg;
        for (int i=j*10000; i<((j+1)*10000); i++){
                if (i<ARCHnodes){
                        disp[disptplus][i][0] = 0.0;
                        disp[disptplus][i][1] = 0.0;
                        disp[disptplus][i][2] = 0.0;
                }
                else {
                        i=((j+1)*10000);
                }
        }
        time_integration_loop_in_batches.commit();
}

void* second_part_of_time_integration_loop (void* arg){
     int j=(int)arg;
     for (int i=j*10000; i<((j+1)*10000); i++){
        if (i<ARCHnodes){
     disp[disptplus][i][0] *= - Exc.dt * Exc.dt;
     disp[disptplus][i][0] += 2.0 * M[i][0] * disp[dispt][i][0] -
          (M[i][0] - Exc.dt / 2.0 * C[i][0]) * disp[disptminus][i][0] -
            Exc.dt * Exc.dt * (M23[i][0] * phi2(sim_time) / 2.0 +
                               C23[i][0] * phi1(sim_time) / 2.0 +
                               V23[i][0] * phi0(sim_time) / 2.0);
        disp[disptplus][i][0] = disp[disptplus][i][0] /
                                (M[i][0] + Exc.dt / 2.0 * C[i][0]);
        vel[i][0] = 0.5 / Exc.dt * (disp[disptplus][i][0] -
                                    disp[disptminus][i][0]);


      disp[disptplus][i][1] *= - Exc.dt * Exc.dt;
      disp[disptplus][i][1] += 2.0 * M[i][1] * disp[dispt][i][1] -
          (M[i][1] - Exc.dt / 2.0 * C[i][1]) * disp[disptminus][i][1] -
            Exc.dt * Exc.dt * (M23[i][1] * phi2(sim_time) / 2.0 +
                               C23[i][1] * phi1(sim_time) / 2.0 +
                               V23[i][1] * phi0(sim_time) / 2.0);
        disp[disptplus][i][1] = disp[disptplus][i][1] /
                                (M[i][1] + Exc.dt / 2.0 * C[i][1]);
        vel[i][1] = 0.5 / Exc.dt * (disp[disptplus][i][1] -
                                    disp[disptminus][i][1]);


      disp[disptplus][i][2] *= - Exc.dt * Exc.dt;
      disp[disptplus][i][2] += 2.0 * M[i][2] * disp[dispt][i][2] -
          (M[i][2] - Exc.dt / 2.0 * C[i][2]) * disp[disptminus][i][2] -
            Exc.dt * Exc.dt * (M23[i][2] * phi2(sim_time) / 2.0 +
                               C23[i][2] * phi1(sim_time) / 2.0 +
                               V23[i][2] * phi0(sim_time) / 2.0);
        disp[disptplus][i][2] = disp[disptplus][i][2] /
                                (M[i][2] + Exc.dt / 2.0 * C[i][2]);
        vel[i][2] = 0.5 / Exc.dt * (disp[disptplus][i][2] -
                                    disp[disptminus][i][2]);
        }
```

```
        else {
                i=(j+1)*10000;
        }
        }
        time_integration_loop_in_batches_2.commit();

}

void* smvp_for_spec (void* arg){
   int base= (int)arg;
    double vi0, vi1, vi2, sum0, sum1, sum2, value, value1, value2;
    double vcol0, vcol1, vcol2, wcol0, wcol1, wcol2;
    int Anext, Alast, col;
   int top=base+7542;
   if (top>30000){
       top=ARCHnodes;
   }
   for (int i=base; i<top; i++){
    Anext = ARCHmatrixindex[i];
    Alast = ARCHmatrixindex[i + 1];
    vi0 = disp[dispt][i][0];
    vi1 = disp[dispt][i][1];
    vi2 = disp[dispt][i][2];
    sum0 = K[Anext][0][0] * vi0 + K[Anext][0][1] * vi1 + K[Anext][0][2] * vi2;
    sum1 = K[Anext][1][0] * vi0 + K[Anext][1][1] * vi1 + K[Anext][1][2] * vi2;
    sum2 = K[Anext][2][0] * vi0 + K[Anext][2][1] * vi1 + K[Anext][2][2] * vi2;
    Anext++;
    while (Anext < Alast) {
      col = ARCHmatrixcol[Anext];
      vcol0 = disp[dispt][col][0];
      vcol1 = disp[dispt][col][1];
      vcol2 = disp[dispt][col][2];
      value = K[Anext][0][0];
      sum0 += value * vcol0;
      wcol0 = value * vi0;
      value = K[Anext][0][1];
      sum0 += value * vcol1;
      wcol1 = value * vi0;
      value = K[Anext][0][2];
      sum0 += value * vcol2;
      wcol2 = value * vi0;
      value = K[Anext][1][0];
      sum1 += value * vcol0;
      wcol0 += value * vi1;
      value = K[Anext][1][1];
      sum1 += value * vcol1;
      wcol1 += value * vi1;
      value = K[Anext][1][2];
      sum1 += value * vcol2;
      wcol2 += value * vi1;
      value = K[Anext][2][0];
      sum2 += value * vcol0;
      value1=  K[Anext][2][1];
      value2 = K[Anext][2][2];
      sum2 += value1 * vcol1;
      sum2 += value2 * vcol2;
      pthread_mutex_lock(&col_mutex[col]);
      disp[disptplus][col][0] += wcol0 + value * vi2; //Exposed read & write
      disp[disptplus][col][1] += wcol1 + value1 * vi2; //Exposed read & write
      disp[disptplus][col][2] += wcol2 + value2 * vi2; //Exposed read & write
      pthread_mutex_unlock(&col_mutex[col]);

      Anext++;
    }
    pthread_mutex_lock(&col_mutex[i]);
    disp[disptplus][i][0] += sum0; //Exposed read & write
    disp[disptplus][i][1] += sum1; //Exposed read & write
    disp[disptplus][i][2] += sum2; //Exposed read & write
    pthread_mutex_unlock(&col_mutex[i]);
  }

    time_integration_loop_in_batches_smvp.commit();
}
```