# Parallelization of a Large-Scale Computational Earthquake Simulation Program

K.F. Tiampo$^{(1)}$, J.B. Rundle$^{(2)}$, S. Gross$^{(1)}$ and S. McGinnis$^{(1)}$

(CIRES, University of Colorado, Boulder, CO USA (e-mail: kristy@fractal.colorado.edu; sethmc@turcotte.colorado.edu; phone: +01-303-492-4779); (2) Dept. of Physics, Colorado Center for Chaos & Complexity, CIRES, University of Colorado, Boulder, CO, 80309, USA, and Distinguished Visiting Scientist, Jet Propulsion Laboratory, Pasadena, CA, 91125, USA (email: rundle@cires.colorado.edu; phone +01-303-492-4779).

## Abstract

Here we detail both the methods and preliminary results of first efforts to parallelize two General Earthquake Model (GEM)-related codes: 1) a relatively simple data mining procedure based on a Genetic Algorithm; and 2) the *Virtual California* simulation of GEM . These preliminary results, using a simple, heterogeneous processor system, existing freeware, and with an extremely low cost of both manpower and hardware dollars, motivate us to more ambitious work with considerably larger-scale computer earthquake simulations of southern California. The GEM computational problem, which is essentially a Monte Carlo simulation, is well suited to optimization on parallel computers, and we outline how we are proceeding in implementing this new software architecture.

## 1.0 Introduction

With the increasing availability of computer and network hardware, accompanied by decreasing cost and ever better processor speed, the construction of parallel computational systems from off-the-shelf components, in lieu of purchasing CPU time on expensive supercomputers, has become ever more practical and attractive. In this paper we describe a preliminary attempt to adapt existing C code for parallel computing on a simple, heterogeneous processor system, the benefits of such a casual implementation, and future plans to parallelize a large-scale computer earthquake simulation of southern

California, the *Virtual California* simulation of the General Earthquake Model (GEM) project.

## 2.0 Parallelization Methods

### 2.1 Hardware

The term “Beowulf class systems” has come to describe, in general, multi-computer architecture that supports parallel computing [1]. While not true in every case, it frequently consists of a server node, and multiple client nodes connected via Ethernet or other network components. These networks can range from a small set of machines connected through Ethernet cable and a switch, as in our present case, or a large number (on the order of hundreds) of linux processors connected via fast switches, as in our future plans [1].

Our current hardware consists of a heterogeneous mix of eight stand-alone PCs, with CPU speeds ranging from 266 MHz to 800 MHz, connected by an Asante FriendlyNet FS3208, supplying 10 BaseT internally. These machines all run some version of the linux operating system, depending upon their age.

Future plans are to implement our parallelized codes on either a large-scale Beouwulf cluster, or the Maui MHPCC symmetric multi-processor (SMP) supercomputer. The Beouwulf cluster, CoSMIC, is currently under construction by the Physics department at the University of Colorado, and will consist of 352 dual processor 800 MHz Pentium IIIs, each with 512 Mbytes of RAM and an $\approx$ 10 Gbyte disk, configured as an integrated set of subclusters each consisting of 32 dual processors. Eight subclusters will be networked using fast Ethernet, each on their own fast Ethernet switch, effectively isolating them from most network traffic. There also will be three subclusters networked using Myrinet, resulting in a 96-node cluster optimized for multi-node applications. The 10 Gbyte disks will provide both short-term storage, as well as swap space, and will contain any local operating system and software. Long term storage will be provide by 3.6 Tbytes of disk space, consisting of 9 RAID devices installed in three file servers, and backed up to a DLT7000 tape drive. This system is designed to provide both large-scale, load balanced serial or parallel computations in conjunction with low cost localized scientific visualization.

## 2.2 Software

Parallelization can be described as either implicit or explicit. Implicit methods are those where the parallelism is determined by the compiler, examples of which include FORTRAN 90, High Performance FORTRAN (HPF), Bulk Synchronous Parallel (BSP), and others. Explicit methods are those where the parallelism is determined by the user. In this case, the user modifies the computer source code specifically for a parallel computer, adding messages using Parallel Virtual Machine (PVM) or Message Passing Interface (MPI), or POSIX threads [1]. For our initial attempts, we opted for explicit parallelization only.

### 2.2.1 PVM

PVM is a portable, freeware message-passing library, obtainable via http://www.epm.ornl.gov/pvm/pvm_home.html, which supports single-processor and

SMP machines as well as clusters of linked machines. Its primary advantage is that it works across a variety of different types of processors, networks, and configurations. This ability to interface over heterogeneous clusters is offset by the significant overhead associated with the message handling [2,3].

### 2.2.2 MPI

MPI is the new official standard for message passing, available at http://www.mcs.anl.gov:80/mpi. While MPI includes a number of features that go beyond the basic message passing model of PVM, such as remote memory access (RMA) and parallel file I/O, these make it necessary to learn an almost entirely new language in order to implement these features. In addition, MPI assumes that the system is either a massively parallel processor (MPP), or a cluster of nearly identical machines [2,4].

## 3.0 Trial Parallelization

After having studied the options above, we determined to run a trial attempt at parallelism using our existing system, as described above, and by modifying a C program which has been employed for data mining and geophysical inversions for a number of years. The program is a genetic algorithm (GA) inversion code. Genetic algorithms are notoriously parallelizable, and conversion of the code to a parallel implementation provided an opportunity to test the difficulty level of the conversion as well as to benchmark the potential time savings associated with such a heterogeneous network.

Many geophysical optimization problems are nonlinear and result in objective functions with a rough fitness landscape and several local minima. Consequently, local optimization techniques, e.g., linearized matrix inversion, steepest descent, conjugate gradients, etc. can converge prematurely to a local minimum. Genetic algorithms have proven themselves an attractive global search tool suitable for the irregular, multimodal fitness functions typically observed in nonlinear optimization problems in the physical sciences.

In general, geophysical inverse problems involve employing large quantities of measured data, in conjunction with an efficient computational algorithm that explores the model space to find the global minimum associated with the optimal model parameters. In a GA, the parameters to be inverted for are coded as genes, and a large population of potential solutions for these genes is searched for the optimal solution. After starting with an initial range of models, the fitness of each solution is measured by a quantitative, objective function. The fittest members of each population then are combined using probabilistic transition rules to form a new offspring population. This procedure is repeated through a large number of generations until the best solution is obtained, based on the fitness measure [5]. It has been demonstrated that those members of the population with a fitness greater than the average fitness of the population itself will increase in number exponentially, effectively accelerating the convergence of the inversion process [6,7,8]. Our program, shown schematically in Figure 1, employs a random number generator to produce an initial set of 100 potential values for each of the model parameters, which are then coded as genes. One gene for each model parameter is assigned to a particular member of that initial population, creating 100 potential solutions to the inversion problem. These members are ranked, from best to worst, according to an external fitness function. The members with the lowest chi-square value are the fittest and are selected to contribute to the next generation. After completion of both crossover and mutation, the population is reevaluated as above, and the process is repeated over subsequent generations, exploiting information in past generations to search the parameter space with improved performance.

As shown schematically in Figure 1, the GA must evaluate 100 members of the population, using the fitness function, for each generation. In the original program, this operation is performed in serial, but if performed in parallel, there is a significant potential for faster performance.

## 3.1 GA Inversion Code

Parallelizing the GA code requires modifications to the main program in the evaluation module, and in the fitness function itself (see Figure 1). We opted to use PVM for the message passing, due to the heterogeneity of our networked system [9]. The programming modifications and debugging took approximately one day, performed by someone familiar with the GA. The PVM additions are relatively simple, constituting less than 100 lines of code. The following is a pseudocode version of the parallelization procedure, showing first the original serial version, followed by the revised parallel implementation.

### 3.1.1 Serial

#### 3.1.1.1 Inversion program

```
main():
initialize_random_genes()
WHILE best_fitness() < target
    select_top_ten(genes)
    breed_new_population(genes)
    FOREACH gene
        evaluate_fitness(gene)
    END FOREACH
END WHILE
```

#### 3.1.1.2 Fitness Function

```
evaluate_fitness():
locs[] = read_observation_locations()
real_deform[] = read_observed_data()
source_parameters = F(gene)
model_deform[] = calculate_displacements(locs[], source_parameters)
FOREACH loc
    chisq += ( model_deform[loc] - real_deform[loc]) )**2
END FOREACH
return fitness = exp( - chisq )
```

### 3.1.2 Parallel (Master/Slave)

#### 3.1.2.1 Master Inversion Program

```
main():

initialize_random_genes()

WHILE best_fitness() < target

    select_top_ten(genes)

    breed_new_population(genes)

    DO
        IF receive_ready_message()
            pack_gene_into_message()
            send_message_to_slave_process()
            ++outstanding
        END IF
        IF outstanding && receive_finished_message()
            receive_fitness_message()
            unpack_fitness_from_message()
            --outstanding
        END IF

    UNTIL outstanding == 0 && num_evaluated == num_genes

    END DO

END WHILE
```

#### 3.1.2.2 Slave fitness program

```
main():

locs[] = read_observation_locations()

real_deform[] = read_observed_data()
```

```
LOOP FOREVER
    send_ready_message()
    receive_gene_message()
    unpack_source_parameters_from_message()
    model_deform[] = calculate_displacements(locs[], source_parameters)
    FOREACH loc
        chisq += ( model_deform[loc] - real_deform[loc]) )**2
    END FOREACH
    fitness = exp( - chisq )
    pack_fitness_into_message()
    send_result_message()
END LOOP
```

## 3.2 Results

Table 1 shows the results for various configurations of a two-processor system. Koch is an 800 MHz machine, while Richter has a 266 MHz processor. We benchmarked both the fitness function for a spherical point source, as well as a fitness function for an ellipsoidal point source, which takes a substantially longer processor time to run. We ran each GA inversion on each machine singly, and then on both machines using PVM. One variation on the two-processor configuration is that we compared the results using one machine as the master, with the other as a slave, and then reversed them. The average CPU time, in seconds, is shown for a thousand generations.

The results in Table 1 show a significant increase in runtime savings for two processors over one, with the greatest increase coming if Koch, the faster machine, is used as the master. A greater percentage in timesavings is accomplished for the elliptical source, the fitness function which takes a much greater time to run. These results for what is a relatively casual attempt at parallelization, with an investment of only a few hours time, and using a heterogeneous mixture of machines inside a relatively slow switch which can be purchased today for less than $100, leads us to optimistically view our future attempts to parallelize the GEM computer simulation for California.

## 4.0 Computational Structure of the Earthquake Simulation Problem

The GEM computational model for the numerical simulation of earthquakes involves a layered series of codes whose structure we now describe. Although at present, the simulation and data mining analysis codes are written in a series of C and Fortran 77, the problem is essentially a Monte Carlo simulation and is therefore well suited to optimization on a parallel computers such as the Maui MHPCC SMP machine, or on a Beowulf Linux cluster.

### 4.1 Model Physics

We first begin with a description of the physics that we simulate. General methods for carrying out the network simulations have been discussed in refs. [10,11,12]. Briefly, one defines a fault geometry in an elastic medium, computes the stress Greens functions (i.e., stress transfer coefficients), assigns frictional properties to each fault, then drives the system via the slip deficit (defined below). The elastic interactions produce mean field dynamics in the simulations [10]. We focus here on the major horizontally slipping strike-slip (horizontal motion) faults in southern California that produce the most frequent and largest magnitude events. We used the tabulation of strike slip faults and fault properties as published in ref [13]. All major faults in southern California, together with the major historic earthquakes, are shown in Figure 2. Figure 3 shows our model fault network. Each fault was assigned a uniform depth of 20 km, the maximum depth of earthquakes in California, and was subdivided into segments having a horizontal scale size of approximately 10 km each.

Several friction laws are described in the literature, including Coulomb failure [14], slip-dependent or velocity-dependent friction [14], and rate-and-state [15]. Here we use a parameterization of recent laboratory friction experiments [16,17], in which the stiffness of the loading machine is low enough to allow for unstable stick-slip when a failure threshold $\sigma^F(V)$ is reached, where $\sigma^F(V)$ is a weak (logarithmic) function of the load point velocity V. Sudden slip then occurs in which the stress decreases to the level of a residual stress $\sigma^R(V)$, again a weak function of V. Stable precursory slip, characterized by a leakage parameter $\alpha$, is observed to occur whose velocity increases with stress level, reaching a magnitude of a few percent of the driving load point velocity just prior to failure at $\sigma = \sigma^F(V)$. For the simplest model that describes this frictional physics displayed in the experiments, we find that

$$\alpha = \frac{2\, s_{ss}}{VT^2}, \tag{1}$$

where V is the plate velocity as before, T is the average interval between sudden unstable slip events, and $s_{ss}$ is the total stable slip that occurs during T. The fraction of stable to total slip that occurs in a laboratory experiment or on a fault is in principle observable, and has in fact been tabulated for a variety of faults in California [13]. The observable quantity $\sigma^F(V) - \sigma^R(V)$ determines the magnitude of the unstable slip. Thus the important parameters of the model that describes laboratory friction can be readily set by either laboratory or field observations.

## 4.2 Computational Structure

The *Virtual California* simulation of the GEM project is a Monte Carlo, Cellular Automaton version of a Langevin-type dynamics. The actual geometric structure of the fault system in California can be implemented as a coarse-grained mesh of fault segments embedded in a layered elastic half space. The various pieces of the fault segments interact by means of elastic interactions. Parameters for the friction law must be specified on each of the fault segments, together with the long-term rate of slip, $V(\mathbf{x}_i)$, on the segment centered at $\mathbf{x}_i$.

The implementation of the model is carried out in three layers of codes, beginning with two data files. The first data file, **Fault_Data.d**, contains the basic geometry of the N fault segments, specifically the coordinates of each of the four corners of the fault segments. This data file also contains the long-term rate of slip $V(\mathbf{x}_i)$ for each of the segments. A second data file, **Fault_Friction.d**, contains the average recurrence time intervals $T_i$ between unstable slip events on the ith segment, as well as values of $\alpha$ for each segment.

These two data files are used, together with standard methods [11] from elasticity theory to compute the stress Greens functions (stress transfer coefficients) by a code **SG_Compute.c**. Since the form of the elastic stress transfer coefficients is known analytically, the computations performed by **SG_Compute.c** are simply function evaluations. The output from this code is contained within a data file **SG_Coefficients.d**, and is a set of $N^2$ stress transfer coefficients (including the self-stress term) for all of the fault segments. These stress transfer coefficients, together with the fault slip rate data in **Fault_Data.d** and friction data in **Fault_Friction.d**, then are used as the basic inputs to the earthquake simulation code **EQ_Simulator.c**. The latter is essentially a Monte Carlo algorithm that encodes the CA Langevin dynamics, assuming a random component during each unstable fault slip event. The equations for the solved on the ith fault segment are:

$$\frac{ds_i}{dt} = \frac{\Delta\sigma_i}{K_i}\left\{\alpha_i + (1+\eta_i)\delta(t - t_F)\right\} - \varepsilon_i\, F\left((Vt - s_i) - \phi_i^*\right) \tag{2}$$

$$\sigma_i = \sum_j T_{ij}\left(V_j t - s_j\right) \tag{3}$$

where $\Delta\sigma_i = \sigma_i - \sigma^R_i$, $T_{ij}$ is the matrix of stress transfer coefficients, $K_i = \sum_j T_{ij}$, and

F() is an odd (nonlinear function) of s with amplitude $\varepsilon_i$ and parameter $\varphi_i^*$ [11]. The parameter $t_F$ is any time t at which $\sigma_i(t) \geq \sigma_i^F$, $\delta$ is the Dirac delta function, and $\eta$ is random noise ("overshoot" or "undershoot"). The nonlinear function F() is present because all of the eigenvalues of the linear part of (2) are negative, and therefore the physics has an instability similar to a Peierls instability [18]. Physically, the functions F() and parameters $\varphi_i^*$ correspond to a potential well on a high-dimensional rough energy landscape upon which the system evolves. The set of N parameters $\{\varphi_i^*\}$ represent a fixed point about which the system fluctuates. This physical picture has been established through the use of simulations that demonstrate that the mean field dynamics of the model, a result of the long-range elastic interactions, induces local ergodicity [19,20]. The exact form of F() is unimportant, since for small fluctuations about $\{\varphi_I^*\}$ are controlled by the first nonlinear term in F(), which is always cubic.

The output from **EQ_Simulator.c** is a record of the slip events and stress history of the dynamics for a fixed time period, and we can call it **EQ_History_01.d**. The "01" denotes the fact that **EQ_History_01.d** can be input back into **EQ_Simulator.c** as an initial condition to produce a second earthquake history file **EQ_History_02.d** which continues the dynamical evolution of the fault system to later times. Once the output data files **EQ_History_xx.d** have been computed, their data can be displayed in various ways, for example by a general visualization code **EQ_Visualize.pro** written in IDL or other script.

In addition to slip histories, deformation that would be expected on the surface of the half space can computed, deformation that could in principal be observed via GPS or satellite radar interferometry. To enable this calculation, kinematic Green's functions must be computed via a code **KG_Compute.c** that produces an output file **KG_Coefficients.d**. This data file is then used in a code **EQ_Deformation.c** to compute the surface deformation file **Surface_Deformation.d** , which is then used as input to the general visualization code **EQ_Visualize.pro**.

The flow diagram for this set of computation, simulation, and visualization codes is shown in the Figure 4. Examples of the visualized output from these codes can be found in ref. [21].

### 4.3 Parallelization Procedures

We begin by schematically illustrating the structure of the two codes, 1) **SG_Compute.c** and 2) **EQ_Simulator.c**, as they exist for serial computation, in Appendix B. We then show (again schematically) how these codes are adapted to parallel computation. For the parallel implementation, we will assume that the multiprocessor is an SMP system. The codes **KG_Compute.c** and **EQ_Deformation.c** are similarly structured and modified. Here N is the number of fault segments.

## 5.0 Conclusions

The promising results of a trial attempt at parallelization of a GA program encourage us to more ambitious work with the large-scale computer earthquake simulation of southern California, the *Virtual California* simulation of the General Earthquake Model (GEM) project. These preliminary results, using a simple, heterogeneous processor system, existing freeware, and at an extremely low cost of both manpower and hardware dollars, lead us to believe that conversion of the GEM computational problem, which is essentially a Monte Carlo simulation, is well suited to optimization on a parallel computers such as the Maui MHPCC SMP machine, or on a Beowulf Linux cluster.

## Appendix A – Genetic Algorithm Program Code

### A.1 Serial

#### A.1.1 Main Program – Inversion.c (Evaluation function)

```
/**************************************************************/
/* Evaluation function: This takes a user defined function.  */
/* Each time this is changed, the code has to be recompiled. */
/* The current function is a data calculation and chisquare fit. */
/**************************************************************/

void evaluate(void)
{
int mem;
int k;
double x[NVARS+1];



for (mem = 0; mem < POPSIZE; mem++)
      {
      for (k = 0; k < NVARS; k++)
            x[k+1] = population[mem].gene[k];


      population[mem].fitness = fit(x[1], x[2], x[3], x[4]);
      }

}
```

#### A.1.2 Fitness Function – Fit.c

```
/***** Inversion definitions and variables *******/
```

```
#define MAXGENS 5000             /* max. number of generations */
#define NSPH 1   /* no. of spheres */
#define NNORM 0 /* no. of normal faults */
#define NUMELL 0 /* no. of ellipses  */
#define NUMLIN 100   /* no. of potential lines */
#define NGPS 1000   /* no. of potential gps points */
#define NUP 4000   /* no. of potential uplift points */
#define NFIXUP 40  /* no. of potential fixed pts for leveling */
#define NUPEACH 100 /* no. of potential uplift pts for each fixed
levelling pt */
#define NLASER
#define NDATA 8100   /* no. of potential data points */
#define S 0.23873241 /*  (3/4)/pi   */
#define SS 0.026525824 /*  (1/12)/pi   */
#define PI 3.1415926535

int numlin, numsph, numell, num_normal, nup, nup1, nup2, nup3, nup4,
nup5, nup6, nup7, ndata, ngps, nfixup, nupeach, nlaser; /* see
definitions in function below */
int generation;                 /* current generation no. */
int upeach[7];

double datav[NDATA], data[NDATA], error[NDATA];
double xwline[NUMLIN], xeline[NUMLIN];
double ywline[NUMLIN], yeline[NUMLIN];
double xgps[NGPS], ygps[NGPS], zgps[NGPS];
double xup[NUP], yup[NUP];
double xlaser[NLASER], ylaser[NLASER];
double xfixup[NFIXUP], yfixup[NFIXUP];
double xfix, yfix, uxcalr, uycalr;

FILE *in_file;
FILE *out;

double uxsph(double, double, double, double);  /* subroutines for
spherical point source displacements  */
double uysph(double, double, double, double);  /* called by fit function
*/
double uzsph(double, double, double, double);

/*******************************************************************/
/*  This is the fitness function subroutine for the genetic algorithm
for inverting geodetic data for two volcanic sources - one spherical,
one elliptical, and a normal fault.

    Definitions:

      gene(NMOD)       --   Model vector
      data(NDATA)      --   Data vector
      vardat(NDATA)    --   Data variances
      xeline(NUMLIN)   --   x coord. of east end of trilateration line
      yeline(NUMLIN)   --   y coord. of east end of trilateration line
      xwline(NUMLIN)   --   x coord. of west end of trilateration line
      ywline(NUMLIN)   --   y coord. of west end of trilateration line
      uex, uey         --   x and y displacements @ east end of line
      uwx, uzy         --   x and y displacements @ west end of line
      xe, ye, xw, yw   --   Distance from endpoints to sphere(s)
      length           --   Length of trilateration line
      unitx, unity     --   Unit vectors along line, east to west
      xsph, ysph       --   Sphere x and y coordinates
      sphdep           --   Sphere depth
      volexp           --   Sphere point volume expansion
      xfix, yfix       --   Benchmark location x and y coordinates
      uxf, uyf, uzf    --   x, y, x displacement of benchmark due to
sphere
      xgps, ygps, zgps --   GPS point locations
      x(NGPS or NUP)   --   Distance in x direction from gps point to
sphere
      y(NGPS or NUP)   --   Distance in y direction from gps point to
sphere
      uxcal(NGPS)      --   x displacement of GPS point due to sphere
      uycal(NGPS)      --   y displacement of GPS point due to sphere
      uzcal(NGPS)      --   z displacement of GPS point due to sphere
      uzup(NUP)        --   z displacement of uplift point due to sphere
      n                --   Number of data points = NUMLIN + NUP +
3*NGPS
      datav(n)         --   Calculated change in data points;
                          for lines, equals line expansion;
                          for points, equals difference between motion
                          at GPS or uplift point and benchmark.
      chisqr           --   Value of reduced chi square. */


double fit(double A, double B, double C, double D)
{
int n, m, i, j, k, nmod, upsum;
double phiback, squares;

      double uex, uey, uwx, uwy, uexx, ueyy, uwxx, uwyy, xe, xw, ye, yw,
uxf, uyf, uzf;
      double length, powx, powy, unitx, unity;
      double chisq, chisqr;
      double x, y, r, s;
      double diff;
      double xr, yr, xer, yer, xwr, ywr, uxfr, uyfr;
      double uxcal[ngps], uycal[ngps], uzcal[ngps];
      double tuxgps[ngps], tuygps[ngps], tuzgps[ngps];
      double uexp, ueyp, uwxp, uwyp, xep, yep, xwp, ywp, theta1;
      double uzup[nup], tuzup[nup];
      double uzlaser[nlaser];
      double xsph[NSPH], ysph[NSPH], sphdep[NSPH], volexp[NSPH];

FILE *output_file;
FILE *test_file;

/********  Initialize data vector indices  *************/

n = 0;
numsph = NSPH;
numell = NUMELL;
num_normal = NNORM;

in_file = fopen("fit1.in", "r");
```

```
/*  Unpack the model vector, get point source locations   */

fscanf(in_file, "%i %i %i %i %i %i %i %i %i %i %i", &numlin, &ngps,
&nfixup, &nup1, &nup2, &nup3, &nup4, &nup5, &nup6, &nup7, &nlaser);
fclose(in_file);

nup = nup1 + nup2 + nup3 + nup4 + nup5 + nup6 + nup7;

      xsph[0] = A;
      ysph[0] = B;
      sphdep[0] = C;
      volexp[0] = D;

/*  Trilateration lines   */

/*  Compute the line vectors      */

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

      powx = pow((xeline[i] - xwline[i]),2);
      powy = pow((yeline[i] - ywline[i]),2);
      length = sqrt(powx + powy);
      unitx = (xeline[i] - xwline[i])/length;
      unity = (yeline[i] - ywline[i])/length;

/*  Initialize displacements  */

      uex = uey = uwx = uwy = 0.;
      uexx = ueyy = uwxx = uwyy = 0.;
      xer= yer = xwr = ywr = 0.0;

/*  Calculate isotropic point expansions  */

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

            xe = xeline[i] - xsph[j];
            ye = yeline[i] - ysph[j];
            xw = xwline[i] - xsph[j];
            yw = ywline[i] - ysph[j];

            uex = uxsph(sphdep[j],xe,ye,volexp[j]) + uex;
            uey = uysph(sphdep[j],xe,ye,volexp[j]) + uey;
            uwx = uxsph(sphdep[j],xw,yw,volexp[j]) + uwx;
            uwy = uysph(sphdep[j],xw,yw,volexp[j]) + uwy;

}

/*  Calculate line expansion, data point  */
/*  Count each line data point  */

      n++;
      datav[n] = ((uex - uwx) * unitx) + ((uey - uwy) * unity);
/*fprintf(test_file,"%lf\n",datav[n]);*/
}

/*  GPS point displacements  */
/*  First calculate expansion @ benchmark */

uxf = uyf = uzf = 0.;
x = y = 0.0;
xr = yr = 0.0;

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

      x = xfix - xsph[i];
      y = yfix - ysph[i];

      uxf = uxsph(sphdep[i],x,y,volexp[i]) + uxf;
      uyf = uysph(sphdep[i],x,y,volexp[i]) + uyf;
      uzf = uzsph(sphdep[i],x,y,volexp[i]) + uzf;
}

/*   Initialize and calculate the GPS displacement  */

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

      uxcal[i] = 0.;
      uycal[i] = 0.;
      uzcal[i] = 0.;
      xr = yr = 0.0;

/*  Isotropic point expansion  */

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

            x = xgps[i] - xsph[j];
            y = ygps[i] - ysph[j];
            uxcal[i] = uxsph(sphdep[j], x, y, volexp[j]) + uxcal[i];
            uycal[i] = uysph(sphdep[j], x, y, volexp[j]) + uycal[i];
            uzcal[i] = uzsph(sphdep[j], x, y, volexp[j]) + uzcal[i];
}

/*  Add data point to point count, n  */
/*  Calculate difference between motion of benchmark and GPS */

      n++;
      datav[n] = uxcal[i] - uxf;
      n++;
      datav[n] = uycal[i] - uyf;
      n++;
      datav[n] = uzcal[i] - uzf;
}

/*  End of GPS loop  */

/*  Leveling lines  */

/*  Initialize z-displacement  */

upsum=0;

for (k=0; k < 7; k++){
```

```
/*  First calculate expansion @ leveling loop benchmark */

uxf = uyf = uzf = 0.;
x = y = 0.0;
xr = yr = 0.0;

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

      x = xfixup[k] - xsph[i];
      y = yfixup[k] - ysph[i];
      uxf = uxsph(sphdep[i],x,y,volexp[i]) + uxf;
      uyf = uysph(sphdep[i],x,y,volexp[i]) + uyf;
      uzf = uzsph(sphdep[i],x,y,volexp[i]) + uzf;
}

/*  Now calculate displacements on leveling loops    */

for (i = upsum; i < (upsum + upeach[k]); i++){

      uzup[i] = 0.;
      xr = yr = 0.0;

/*  Calculate isotropic point expansions */

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

            x = xup[i] - xsph[j];
            y = yup[i] - ysph[j];
            uzup[i] = uzsph(sphdep[j],x,y,volexp[j]) + uzup[i];
}

/* Calculate difference between motion and benchmark, increment n */

      n++;
      datav[n] = uzup[i] - uzf;
}

upsum=upsum + upeach[k];

}

/*  End of uplift loop  */
/*  Uplift points (laser altimeter)  */

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

x = y = 0.0;
xr = yr = 0.0;
      uzlaser[i] = 0.;

/*  Calculate isotropic point expansions */

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

            x = xlaser[i] - xsph[j];
            y = ylaser[i] - ysph[j];
            uzlaser[i] = uzsph(sphdep[j],x,y,volexp[j]) + uzlaser[i];
}
            n++;
      datav[n] = uzlaser[i];
}

/* End of laser calculations  */

/* Calculate chi square; number of data points equals n */

ndata = n;

/*  Check that ndata = numlin + nup + 3*ngps + nlaser */

m = numlin + nup + 3*ngps + nlaser;

if (n != m)
{
fprintf(output_file, "Caution: ndata does not equal numlin + nup +
3*ngps + nlaser! n=");
fprintf(output_file, "%i %i \n", n, upsum);
}

diff = 0.0;
chisq = 0.0;
squares = 0.0;
chisqr = 0.0;

for (i = 1; i <= ndata; i++){

      diff = (data[i] - datav[i])/error[i];
      chisq = chisq + pow(diff,2.0);
}

squares = exp(-chisq);

return squares;
}

/*****************************************************************/
/*  Functions  */

/*  Expansions - compute displacements due to dilating sphere  */

      /*  X-displacements due to a dilating sphere */

double uxsph(double h, double x, double y, double volexp)
{
      double r, ux, r3;
      r = sqrt((h*h) + (y*y) + (x*x));
      r3 = pow(r,3.);
      ux = ((S*x)*volexp)/(r3);

      return ux;
}

      /*  Y-displacements due to a dilating sphere */
```

```
double uysph(double h, double x, double y, double volexp)
{
      double r, uy, r3;
      r = sqrt((h*h) + (y*y) + (x*x));
      r3 = pow(r,3.);
      uy = ((S*y)*volexp)/(r3);

      return uy;
}

      /*  Z-displacements due to a dilating sphere */

double uzsph(double h, double x, double y, double volexp)
{
      double r, uz, r3;
      r = sqrt((h*h) + (y*y) + (x*x));
      r3 = pow(r,3.);
      uz = ((S*h)*volexp)/(r3);

      return uz;
}
```

## A.2 Parallel – Master/Slave

### A.2.1 Master Program – Inversion.c (Evaluation Function)

```
/**************************************************************/
/* Evaluation function: This takes a user defined function.  */
/* Each time this is changed, the code has to be recompiled. */
/* The current function is a data calculation and chisquare fit. */
/**************************************************************/

void eval_pvm(void)
{

int i, j, k, l;
int next, outstanding;
int tid, msgtag, bytes;
int bufid, status, done;

double xx[NVARS];
double fit;

outstanding=0;
next = 0;
done = 0;

/*  So we know - a message tag of 6, from the slave(s), means they
    are ready for data.  A message tag of 3, is sent from here, to send
    the packed genome data.  A message tag of 8 means that there is a
    slave ready to send data, and a message tag of 7 is the data to be
    received from the slave. */
```

```
while (!done)
      {

      /* Check to see if there is a slave ready  */

      if ((bufid=pvm_nrecv(-1, 6)) != 0)
        {
          if(bufid   0)
            {

            /*  Get the id of the ready slave  */

            pvm_bufinfo(bufid, &bytes, &msgtag, &tid);

            /* Create the data set, and send to that slave  */

            for (k = 0; k < NVARS; k++){
              xx[k] = population[next].gene[k];
            }

            pvm_initsend(PvmDataDefault);
            pvm_pkdouble(xx, NVARS, 1);
            pvm_pkdouble(datatest,NTEST*NTEST,1);
            pvm_send(tid,3);

            /*  Increment outstanding to keep track of how
                many slaves are out there computing away */

            outstanding++;
            }
          else
            {
            printf("Error from pvm_nrecv, to fit: %i\n", bufid);
            }
          next++;
        }

      /*  If there are computations out to the slaves, check to see if
          one is ready to send data back.  */

          if (outstanding   0 && (bufid=pvm_nrecv(-1, 8)) != 0)
            {
            if (bufid   0)
              {
                /*  Get data from willing slave.  Place in
                 temporary file so as not to overwrite best fitness
                 (stored in population[POPSIZE].fitness.*/

                pvm_bufinfo(bufid, &bytes, &msgtag, &tid);
                pvm_recv(tid, 7);
                pvm_upkdouble(&fit, 1, 1);
                temp_fit[next] = fit;
              }
            else
              {
                  printf("Error from pvm_nrecv, from fit: %i\n",bufid);
              }
            /*  Decrease number of outstanding computations at slaves.
*/

            outstanding--;
            }

          /*  Check to see if done.  */

          if (next  = POPSIZE && outstanding == 0) done = 1;
      }

/*  Copy over temporary fitnesses to correct array for GA manipulations.
Dump extras created by faster processors.  */

for (i=0; i < POPSIZE; i++){
  population[i].fitness = temp_fit[i];
}

}
```

### A.2.2 Slave Fitness Function – Fitsphere.c

```
#include "pvm3.h"
#include <stdio.h
#include <stdlib.h
#include <math.h

/***** Inversion definitions and variables *******/

#define NSPH 1   /* no. of spheres */
#define NNORM 0 /* no. of normal faults */
#define NUMELL 0 /* no. of ellipses  */
#define NVARS 4*NSPH + 7*NUMELL + 1*NNORM  /* Number of inversion
variables */
#define NUMLIN 100   /* no. of potential lines */
#define NGPS 200   /* no. of potential gps points */
#define NFIXUP 40  /* no. of potential fixed pts for leveling */
#define NUPEACH 100 /* no. of potential uplift pts for each fixed
levelling pt */
#define NLASER 0
#define NUP  200
#define NDATA 200   /* no. of potential data points */
#define S 0.23873241 /*  (3/4)/pi   */
#define SS 0.026525824 /*  (1/12)/pi   */

int numlin, numsph, numell, num_normal, nup, nup1, nup2, nup3, nup4,
nup5, nup6, nup7, ndata, ngps, nfixup, nupeach, nlaser; /* see
definitions in function below */
int generation;                 /* current generation no. */
int upeach[7];

int n, m, i, j, k, nmod, upsum;
int mytid, master;
int done, status, bufid, bytes, msgtag, tid;
```

```
double xx[NVARS];
double squares;

double uex, uey, uwx, uwy, uexx, ueyy, uwxx, uwyy, xe, xw, ye, yw, uxf,
uyf, uzf;
double length, powx, powy, unitx, unity;
double chisq, chisqr;
double x, y, r, s;
double diff;
double xr, yr, xer, yer, xwr, ywr, uxfr, uyfr;
double xsph[NSPH], ysph[NSPH], sphdep[NSPH], volexp[NSPH];

double datav[NDATA], data[NDATA], error[NDATA];
double xwline[NUMLIN], xeline[NUMLIN];
double ywline[NUMLIN], yeline[NUMLIN];
double xgps[NGPS], ygps[NGPS], zgps[NGPS];
double xup[NUP], yup[NUP];
double xlaser[NLASER], ylaser[NLASER];
double xfixup[NFIXUP], yfixup[NFIXUP];
double xfix, yfix, uxcalr, uycalr;
double xnfe[NNORM], ynfe[NNORM], xnfw[NNORM], ynfw[NNORM];
double dip[NNORM], ntop[NNORM], nbottom[NNORM];
double xnmid[NNORM], ynmid[NNORM], semi_lngth[NNORM], thetaf[NNORM];
double uxcal[NGPS], uycal[NGPS], uzcal[NGPS];
double tuxgps[NGPS], tuygps[NGPS], tuzgps[NGPS];
double uexp, ueyp, uwxp, uwyp, xep, yep, xwp, ywp, theta1;
double uzup[NUP], tuzup[NUP];
double uzlaser[NLASER];

FILE *in_file;
FILE *out;
FILE *output_file;

double uxsph(double, double, double, double);  /* subroutines for
spherical point source displacements  */
double uysph(double, double, double, double);  /* called by fit function
*/
double uzsph(double, double, double, double);
void read_field_data(void);

/*******************************************************************/
/*  This is the fitness function subroutine for the genetic algorithm
for inverting geodetic data for a spherical source.

    Definitions:

      gene(NMOD)       --   Model vector
      data(NDATA)      --   Data vector
      vardat(NDATA)    --   Data variances
      xeline(NUMLIN)   --   x coord. of east end of trilateration line
      yeline(NUMLIN)   --   y coord. of east end of trilateration li
      xwline(NUMLIN)   --   x coord. of west end of trilateration line
      ywline(NUMLIN)   --   y coord. of west end of trilateration line
      uex, uey         --   x and y displacements @ east end of line
      uwx, uzy          --   x and y displacements @ west end of line
      xe, ye, xw, yw    --   Distance from endpoints to sphere(s)
      length            --   Length of trilateration line
      unitx, unity      --   Unit vectors along line, east to west
      xsph, ysph        --   Sphere x and y coordinates
      sphdep            --   Sphere depth
      volexp            --   Sphere point volume expansion
      xfix, yfix        --   Benchmark location x and y coordinates
      uxf, uyf, uzf     --   x, y, x displacement of benchmark due to
sphere
      xgps, ygps, zgps --   GPS point locations
      x(NGPS or NUP)    --   Distance in x direction from gps point to
sphere
      y(NGPS or NUP)    --   Distance in y direction from gps point to
sphere
      uxcal(NGPS)       --   x displacement of GPS point due to sphere
      uycal(NGPS)       --   y displacement of GPS point due to sphere
      uzcal(NGPS)       --   z displacement of GPS point due to sphere
      uzup(NUP)         --   z displacement of uplift point due to sphere
      n                 --   Number of data points = NUMLIN + NUP +
3*NGPS
      datav(n)          --  Calculated change in data points;
                         for lines, equals line expansion;
                         for points, equals difference between motion
                         at GPS or uplift point and benchmark.
      chisqr            --   Value of reduced chi square.  */

main()
{

done = 0;
while(!done){

/*  Unpack the model vector, get point source locations   */

mytid = pvm_mytid();
master = pvm_parent();

pvm_initsend(PvmDataDefault);
pvm_send(master, 6);

pvm_recv(master, 3);
pvm_upkdouble(xx, NVARS, 1);

pvm_upkdouble(datatest,NTEST*NTEST,1);

      xsph[0] = xx[0];
      ysph[0] = xx[1];
      sphdep[0] = xx[2];
      volexp[0] = xx[3];


/*******************************/
/* Initialize and read in data */
/*******************************/

   read_field_data();
   n = 0;
   numsph = NSPH;

   nup = nup1 + nup2 + nup3 + nup4 + nup5 + nup6 + nup7;

/*  Trilateration lines   */
/*  Compute the line vectors      */

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

      powx = pow((xeline[i] - xwline[i]),2);
      powy = pow((yeline[i] - ywline[i]),2);
      length = sqrt(powx + powy);
      unitx = (xeline[i] - xwline[i])/length;
      unity = (yeline[i] - ywline[i])/length;

/*  Initialize displacements  */

      uex = uey = uwx = uwy = 0.;
      uexx = ueyy = uwxx = uwyy = 0.;
      xer= yer = xwr = ywr = 0.0;

/*  Calculate isotropic point expansions  */

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

            xe = xeline[i] - xsph[j];
            ye = yeline[i] - ysph[j];
            xw = xwline[i] - xsph[j];
            yw = ywline[i] - ysph[j];

            uex = uxsph(sphdep[j],xe,ye,volexp[j]) + uex;
            uey = uysph(sphdep[j],xe,ye,volexp[j]) + uey;
            uwx = uxsph(sphdep[j],xw,yw,volexp[j]) + uwx;
            uwy = uysph(sphdep[j],xw,yw,volexp[j]) + uwy;

}

/*  Calculate line expansion, data point  */
/*  Count each line data point  */

      n++;
      datav[n] = ((uex - uwx) * unitx) + ((uey - uwy) * unity);
}

/*  GPS point displacements  */
/*  First calculate expansion @ benchmark */

uxf = uyf = uzf = 0.;
x = y = 0.0;
xr = yr = 0.0;

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

      x = xfix - xsph[i];
      y = yfix - ysph[i];
      uxf = uxsph(sphdep[i],x,y,volexp[i]) + uxf;
      uyf = uysph(sphdep[i],x,y,volexp[i]) + uyf;
      uzf = uzsph(sphdep[i],x,y,volexp[i]) + uzf;
}

/*   Initialize and calculate the GPS displacement  */

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

      uxcal[i] = 0.;
      uycal[i] = 0.;
      uzcal[i] = 0.;
      xr = yr = 0.0;

/*  Isotropic point expansion  */

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

            x = xgps[i] - xsph[j];
            y = ygps[i] - ysph[j];
            uxcal[i] = uxsph(sphdep[j], x, y, volexp[j]) + uxcal[i];
            uycal[i] = uysph(sphdep[j], x, y, volexp[j]) + uycal[i];
            uzcal[i] = uzsph(sphdep[j], x, y, volexp[j]) + uzcal[i];
}
/*  Add data point to point count, n  */
/*  Calculate difference between motion of benchmark and GPS */

      n++;
      datav[n] = uxcal[i] - uxf;
      n++;
      datav[n] = uycal[i] - uyf;
      n++;
      datav[n] = uzcal[i] - uzf;
/*fprintf(test_file,"%lf\n",datav[n]);*/
}

/*  End of GPS loop  */

/*  Leveling lines  */

/*  Initialize z-displacement  */

upsum=0;

for (k=0; k < 7; k++){

/*  First calculate expansion @ leveling loop benchmark */

uxf = uyf = uzf = 0.;
x = y = 0.0;
xr = yr = 0.0;

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

      x = xfixup[k] - xsph[i];
      y = yfixup[k] - ysph[i];
      uxf = uxsph(sphdep[i],x,y,volexp[i]) + uxf;
      uyf = uysph(sphdep[i],x,y,volexp[i]) + uyf;
      uzf = uzsph(sphdep[i],x,y,volexp[i]) + uzf;
}

/*  Now calculate displacements on leveling loops    */

for (i = upsum; i < (upsum + upeach[k]); i++){

      uzup[i] = 0.;
      xr = yr = 0.0;

/*  Calculate isotropic point expansions */

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

            x = xup[i] - xsph[j];
            y = yup[i] - ysph[j];
            uzup[i] = uzsph(sphdep[j],x,y,volexp[j]) + uzup[i];
}

/*  Calculate difference between point motion and benchmark, increment n
*/

      n++;
      datav[n] = uzup[i] - uzf;
}

upsum=upsum + upeach[k];
}

/*  End of uplift loop  */

/*  Uplift points (laser altimeter)  */

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


x = y = 0.0;
xr = yr = 0.0;
      uzlaser[i] = 0.;

/*  Calculate isotropic point expansions */

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

            x = xlaser[i] - xsph[j];
            y = ylaser[i] - ysph[j];
            uzlaser[i] = uzsph(sphdep[j],x,y,volexp[j]) + uzlaser[i];
}

      n++;
      datav[n] = uzlaser[i];
}

/* End of laser calculations  */
/*  Calculate chi square; number of data points equals n */

ndata = n;

/*  Check that ndata = numlin + nup + 3*ngps + nlaser */

m = numlin + nup + 3*ngps + nlaser;

if (n != m)
{
fprintf(output_file, "Caution: ndata does not equal numlin + nup +
3*ngps + nlaser! n=");
fprintf(output_file, "%i %i \n", n, upsum);
}

diff = 0.0;
chisq = 0.0;
squares = 0.0;
chisqr = 0.0;


for (i = 1; i <= ndata; i++){

      diff = (data[i] - datav[i])/error[i];
      chisq = chisq + pow(diff,2.0);
}

squares = exp(-chisq);

pvm_initsend(PvmDataDefault);
pvm_send(master, 8);

pvm_initsend(PvmDataDefault);
pvm_pkdouble(&squares, 1, 1);
pvm_send(master, 7);

}

pvm_exit();

}

/****************************************************************/
/*  Functions  */
/****************************************************************/

/*  Expansions - compute displacements due to dilating sphere  */

      /*  X-displacements due to a dilating sphere */

double uxsph(double h, double x, double y, double volexp)
{
      double r, ux, r3;
      r = sqrt((h*h) + (y*y) + (x*x));
      r3 = pow(r,3.);
      ux = ((S*x)*volexp)/(r3);
      return ux;
}

      /*  Y-displacements due to a dilating sphere */

double uysph(double h, double x, double y, double volexp)
{
      double r, uy, r3;
      r = sqrt((h*h) + (y*y) + (x*x));
      r3 = pow(r,3.);
      uy = ((S*y)*volexp)/(r3);

      return uy;
}

      /*  Z-displacements due to a dilating sphere */

double uzsph(double h, double x, double y, double volexp)
{
      double r, uz, r3;
      r = sqrt((h*h) + (y*y) + (x*x));
      r3 = pow(r,3.);
      uz = ((S*h)*volexp)/(r3);

      return uz;
}

/*******************************************************/
/* A procedure for reading in field data.             */
/* The field data should be stored in "fit1.in"       */
/*******************************************************/

void read_field_data()
{
int i, k;
double semix_normal[NNORM],semiy_normal[NNORM];

FILE *in_motion;
FILE *in_error;

in_file = fopen("/pvm_temp/fit/fit1.in","r");
in_motion = fopen("/pvm_temp/fit/motion.in", "r");
in_error = fopen("/pvm_temp/fit/error.in","r");

num_normal=NNORM;

fscanf(in_file, "%i %i %i %i %i %i %i %i %i %i %i", &numlin, &ngps,
&nfixup, &nup1, &nup2, &nup3, &nup4, &nup5, &nup6, &nup7, &nlaser);


nup = nup1 + nup2 + nup3 + nup4 + nup5 + nup6 + nup7;

ndata = numlin + 3*ngps + nup + nlaser;

upeach[0]=nup1;
upeach[1]=nup2;
upeach[2]=nup3;
```

```
upeach[3]=nup4;
upeach[4]=nup5;
upeach[5]=nup6;
upeach[6]=nup7;

/*  Get trilateration line locations   */

for (i=0; i < numlin; i++){
      fscanf(in_file, "%lf %lf %lf %lf", &xeline[i], &yeline[i],
&xwline[i], &ywline[i]);
      fprintf(out, "%lf %lf %lf %lf \n", xeline[i], yeline[i],
xwline[i], ywline[i]);
}
/*  GPS point displacements  */
/*  First get location of benchmark, initialize displacements */

fscanf(in_file, "%lf %lf", &xfix, &yfix);

/*  Then get GPS locations  */

for (i=0; i < ngps; i++)
{
      fscanf(in_file, "%lf %lf %lf", &xgps[i], &ygps[i], &zgps[i]);
      }
/*  Finally, get uplift point locations  */
/*  First, get fixed point locations  */

for (k=0; k < 7; k++)
{
      fscanf(in_file, "%lf %lf", &xfixup[k], &yfixup[k]);

}

for (i=0; i < nup; i++)
{
      fscanf(in_file, "%lf %lf", &xup[i], &yup[i]);
}

/*  Oh, yeah, get laser locations   */

for (i=0; i < nlaser; i++)
{
      fscanf(in_file, "%lf %lf", &xlaser[i], &ylaser[i]);
}

fclose(in_file);

for (i = 1; i <= ndata; i++){

      fscanf(in_motion, "%lf ", &data[i]);
}

fclose(in_motion);

for (i = 1; i <= ndata; i++){

      fscanf(in_error, "%lf ", &error[i]);
}
fclose(in_error);

}
```

# Appendix B – Virtual California Program Code

## B.1 Serial

### B.1.1 Main, SG_Compute.c:

```
Procedure Main ‘SG_Compute’

Procedure Stress_Greens_Function(N,N)

Main()

Read_fault_data()

/*  Begin stress Green’s function computation loop */

WHILE (j,i< number_of_patches)

/*  Compute each stress transfer coefficient  */

                    Procedure Stress_Greens_Function(j,i)

END WHILE

Output_computed_Stress_Green’s_Function()

end
      }
```

### B.1.2 Main, EQ_Simulator.c:

```
Procedure Main ‘EQ_Simulator’;

/*Computes the state of stress on segment i at time t. */

Procedure Stress_State_Compute(N,s,Number_time_steps);
```

```
Main()

Read_fault_data()
Read_friction_data()
Read_Stress_Greens_Funtions()


WHILE (t < number_of_time_steps)

    WHILE(N<number_of_segments)

        time = time + time_index * time_step;

    /*  Compute new value of stress on segment i */

        Procedure Stress_State_Compute(i,time);

    /*  Check to see which segments have $\sigma_i(t) \geq \sigma_i^F$  */
        WHILE (any $\sigma_i(t) \geq \sigma_i^F$ )

            WHILE (i<N)

        /*      Update slip s(i)    */
                s(i)= s(i) +  $\Delta\sigma_i / K_i$  +  random number;

            END WHILE

        END WHILE

    /*   Record time and state of slip in EQ_History.d  */

        Output_computed_Stress_State()

    END WHILE
END WHILE

end
```

## B.2 Parallel Code (Master/Slave)

### B.2.1 Master - SG_Compute.c

/*Computes the stress transferred from fault i to fault j when fault i slips by a unit amount. */

```
Main():

Read_fault_data()

WHILE (N < number_of_patches)

    DO
        IF receive_ready_message()
            Pack_location_id_of_segment()
            Send_message_to_slave()
            ++outstanding
        ENDIF
        IF outstanding && receive_finish_message()
            Receive_Stress_Green's_Function()
            Unpack_Stress_Green's_Function()
            --outstanding
        ENDIF

    UNTIL outstanding == 0 && num_evaluated ==N

    END DO

END WHILE
```

### B.2.2 Slave - SG_Compute.c

```
Main():

Locs[]=read_in_faultgeometry_data()
Parameters[]=read_in_sourceparameters()

LOOP FOREVER

    Send_ready_message()
    Receive_fault_message()
    Unpack_location_index
        Stress_Green's_Functions =
        Calculate_Stress_Green's_Functions(Locs[],Parameters[])

    Pack_Stress_Green's_Function_into_message()
    Send_result_message()

END LOOP
```

## B.2.3 Master, EQ_Simulator.c

```
/*Computes the state of stress on segment i at time t. */

Main():

Read_fault_data()
Read_friction_data()
Read_Stress_Greens_Funtions()

WHILE (t < number_of_time_steps)

        WHILE(N<number_of_segments)

        IF Stress(N) > Failure_stress

                DO
                        IF receive_ready_message()
                                Pack_slip_rates()
                                Pack_Friction_Parameters()
                                Pack_Stress_Green's_Functions()
                                Send_message_to_slave()
                                ++outstanding
                        ENDIF
                        IF outstanding && receive_finish_message()
                                Receive_Stress_State()
                                Unpack_Stress_State()
                                --outstanding
                        ENDIF

                END DO
                Check_stress_state_on_each _segment()

        END IF
        END WHILE

        UNTIL outstanding == 0 && time_steps ==T

END WHILE
```

## B.2.3 Slave, EQ_Simulator.c

```
Main():
```

```
Locs[]=read_in_faultgeometry_data()
Parameters[]=read_in_sourceparameters()

LOOP FOREVER

        Send_ready_message()
        Receive_stress_message()
        Unpack_ slip_rates()
        Unpack_Friction_Parameters()
        Unpack_Stress_Green's_Functions()
                Stress_State =
                Calculate_Stress_State(Slip_rate[],Friction[],Green's_Function)
        Pack_Stress_State_into_message()
        Send_result_message()

END LOOP
```

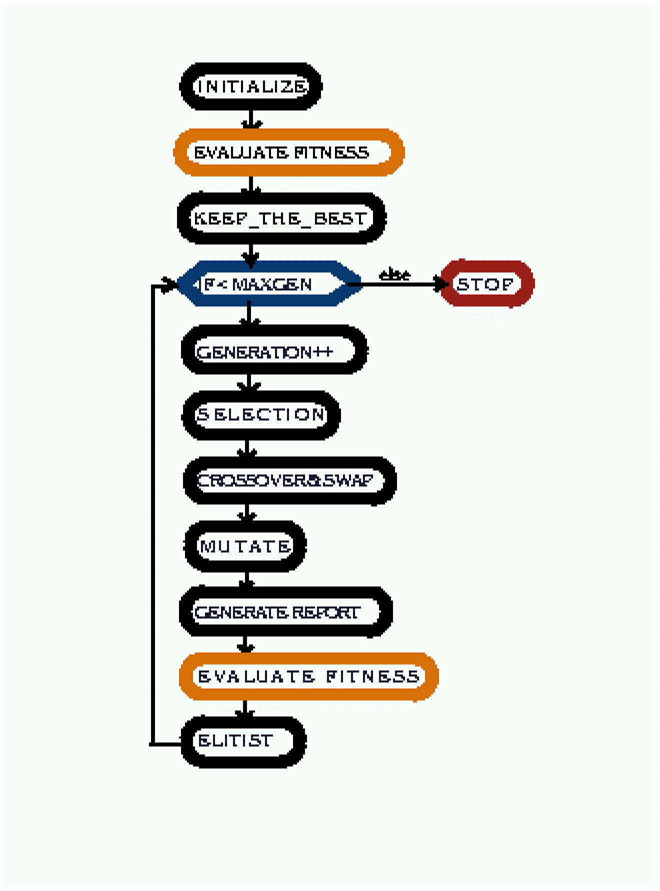

Figure 1: Flow chart, genetic algorithm inversion program.

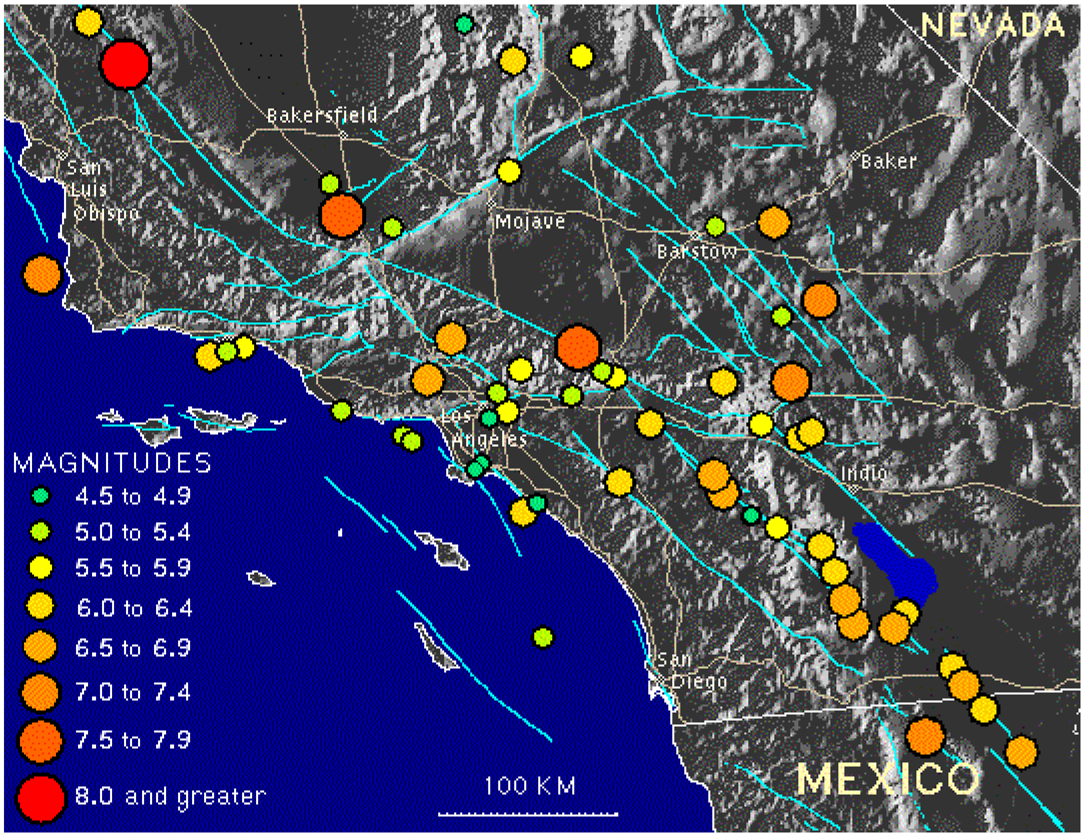

Figure 2: Historic seismicity, southern California (Southern California Earthquake Center, http://www.scecdc.scec.org/clickmap.html).

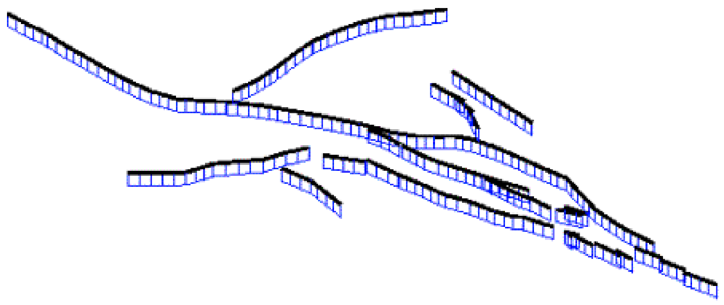

Figure 3:  Map of the 215 fault segments used in the implementation of the Virtual California simulation. Only the strike slip faults are represented in this simplified model.

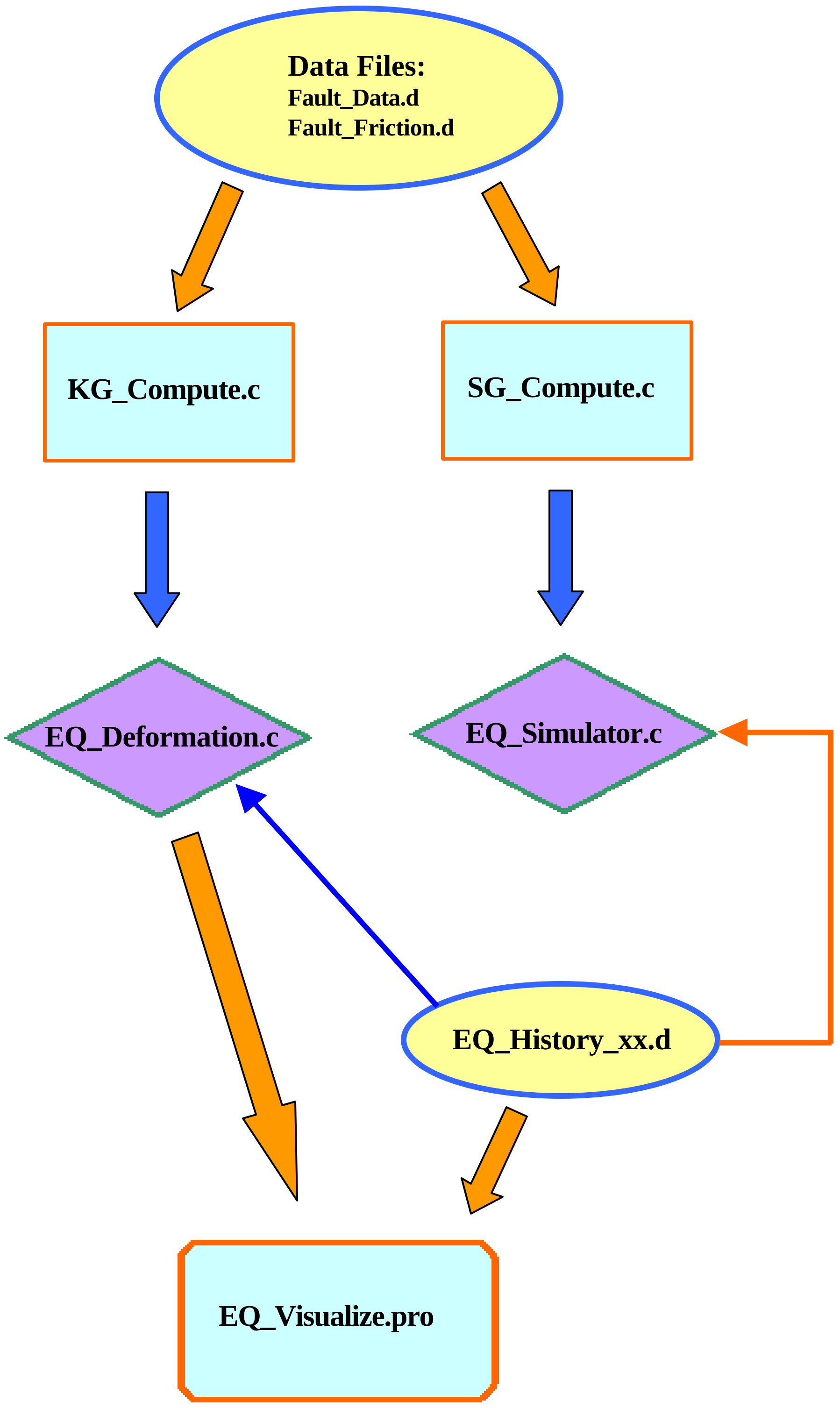

Figure 4: Flow diagram for parallelization of *Virtual California* earthquake simulation program.

| FUNCTION | RICHTER | RICHTER TO KOCH | KOCH | KOCH TO RICHTER |
|---|---|---|---|---|
| SPHERE | 3940 | 1025 | 545 | 420 |
| ELLIPSE | 6300 | 1220 | 830 | 625 |

TABLE 1: Time, in seconds, to process 1000 generations with the processors listed. "Richter to Koch" means that the master process was resident on Richter and there were two slave processes, one on Richter and one on Koch.

## References

[1] J. Radajewski and D. Eadline, *Linux Beowulf How-To*, www.ibiblio.org/mdw/HOWTO/Beowulf-HOWTO.html (1998).
[2] H. Dietz, *Parallel Processing How-To*, www.ibiblio.org/mdw/HOWTO/Parallel-Processing-HOWTO.html (1998).
[3] PVM website, www.epm.ornl.gov/pvm/pvm_home.html.
[4] MPI website, www.unix.mcs.anl.gov/mpi/.
[5] Z. Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, New York, NY (1992).
[6] J.H. Holland, *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA (1975).
[7] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley, Reading, MA (1989).
[8] M. Mitchell, *Proc. First Int. Conf. Artificial Life.* Paris, France, 245 (1992).
[9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA (1994).
[10] J.B. Rundle, W. Klein, K.F. Tiampo and S.J. Gross, *Phys. Rev. E*, **61**, 2418 (2000).
[11] J.B. Rundle, *J. Geophys. Res.*, **93**, 6255 (1988).
[12] S.N. Ward, *Bull. Seism. Soc. Am.*, **90**, 370 (2000).
[13] J. Deng and L.R. Sykes, *J. Geophys. Res.*, **102**, 9859 (1997).
[14] B.N.J. Persson, *Sliding Friction, Physical Principles and Applications* (Springer-Verlag, Berlin, 1998).
[15] J. Dieterich, *J. Geophys. Res.*, **84**, 2161 (1979).
[16] T.E. Tullis, *Proc. Nat. Acad. Sci.*, **93**, 3803 (1996).
[17] S. L. Karner and C. Marone, in *GeoComplexity and the Physics of Earthquakes*, ed. JB Rundle, DL Turcotte and W. Klein (Amer. Geophys. Un., Washington DC, 2000)
[18] See for example, R. Peierls, **Phys. Re**v., *54*, 918 (1934).
[19] Rundle, J.B., W. Klein, S. Gross, and D.L. Turcotte, *Phys. Rev. Lett.*, **75**, 1658-1661 (1995).
[20] W. Klein, JB Rundle and CD Ferguson, *Phys. Rev. Lett.*, **78**, 3793 (1997).
[21] J.B. Rundle, P.B. Rundle, W. Klein, J. de sa Martins, K. F. Tiampo, A. Donnellan, and L. H. Kellogg, *PAGEOPH*, submitted.