Parallelization of a Large-Scale Computational Earthquake Simulation Program

K.F. Tiampo(1), J.B. Rundle(2), P. Hopper(1), J. Sá Martins(1), S. Gross(1), and S. McGinnis(1)

(CIRES, University of Colorado, Boulder, CO USA (e-mail: kristy@fractal.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 three General Earthquake Model (GEM)-related codes: 1) a relatively simple data mining procedure based on a Genetic Algorithm; 2) a mean-field slider block model, and 3) the *Virtual California* simulation of GEM. These preliminary results, using a simple, heterogeneous system of processors, existing freeware, and an extremely low initial cost in 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.

1.0Introduction

With the increasing availability of computer and network hardware, accompanied by decreasing cost and ever increasing 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 an implementation, and our 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, to 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 1 GHz, 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. A 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 will also 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 compiler, examples of which include those provided by FORTRAN 90, High Performance FORTRAN (HPF), and others, determines the parallelism. Explicit methods are those where the user determines the parallelism. 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 may be 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 a new language in order to be implemented. In addition, MPI is better suited to 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 timesavings 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. Crossover is the mechanism that provides the recombination of parents into new offspring; random mutation of the genes ensures genetic richness. The process is repeated over subsequent generations, exploiting information in past generations to search the parameter space with improved performance. Note that, depending upon the particular circumstances of the geophysical inversion, the number of parameters or genes, parameter ranges, crossover and mutation rates, and population size can vary widely (for details and examples, see references 9 through 12).

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 [13]. 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 pseudocode version of the parallelization procedure can be found in Appendix A, showing first the original serial version, followed by the revised parallel implementation.

3.1.2 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-generation run.

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 somewhat extemporaneous 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.

3.2 Mean Field Slider Block Model

As the ultimate goal for this work is the parallelization of *Virtual California*, a cellular automata type computer model, we also set about parallelizing a simple and popular model of this kind, the mean field slider block model. Recognizing that MPI, currently without shared memory, was the most practical and likely application in the near future, the parallelization was done in MPI, using three workstations, including one dual 800 MHz processor, and two additional 1 GHz processors, connected by a faster, D-Link, eight port, 100Mb switch.

This program simulates a homogeneous but noisy driven and dissipative threshold system, with "infinite" interaction range. In this model, sites in a square lattice or grid of sites represent asperities in a tectonic fault, or blocks in an array of interconnected blocks. Each block is uniformly coupled to all other blocks in the lattice by springs - hence the "infinite" range. Motion of the blocks is avoided by using a spatially homogeneous static friction. Each block is also connected to a loader plate, and the system is loaded homogeneously in space and uniformly in time by increasing the stress on each block at a constant rate. When the stress on one block reaches the friction threshold, this block - the initiator - fails and slips until its stress reaches a uniform residual stress, with some random noise added to it, and dumps part of the stress drop it undergoes on the rest of the lattice. Because of this added stress, some other block(s) may reach the stress threshold, and will fail in its turn; the whole cycle of stress drop on the failed block(s) and its redistribution among the other blocks continues until all the blocks have stresses below the threshold. The collection of all the failures triggered by a single initiator is the model's equivalent of an earthquake. Each avalanche of failures is considered to be one iteration, and simulations are usually run for at least several million iterations, on lattices with typically 10^5 sites.

Models with these characteristics, with either short range or long-range couplings, have been extensively used in the last decade or so to study the effects of the basic physical ingredients on the statistics of the sequence of avalanches they generate [14-16]. It has been shown that, when conveniently tuned, they are able to reproduce either scaling behavior of the Gutenberg-Richter kind or the characteristic event regime observed in some natural faults [17,18]. Investigations on the origins and physical mechanisms underlying the scaling behavior and its possible connections with nucleation processes were made possible by simulations of long range versions of these models [19,20]. The code we work with has been highly optimized and makes extensive use of linked lists to avoid unnecessary lattice sweeps - a detailed account of the optimization algorithms used was published in Ref. [21].

The version of the code tailored for parallelization, **rjbmpi**, is schematically illustrated in Appendix B, for both its serial and parallel versions. Using Koch as the master, and the two faster processors as the slaves, the implemented parallelization yielded results more than three times faster than the original serial code. While significantly larger amounts of data would have to be passed from master to slave, the similarities between the two models lead us to conclude that, if implemented with care, significant runtime savings could be achieved through the parallelization of the *Virtual California* model.

4.0 Computational Structure of the Earthquake Simulation Problem

The GEM computational model for the numerical simulation of earthquakes, *Virtual California*, involves a layered series of codes whose structure we now describe. Although at present the simulation and data mining analysis are written as a set of C and FORTRAN 77 codes, the problem is essentially a Monte Carlo simulation and therefore well suited to optimization on a parallel computer system.

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. [22,23,24]. 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 [22]. 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 [23]. 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 [25], slip-dependent or velocity-dependent friction [26], and rate-and-state [27]. Here we use a parameterization of recent laboratory friction experiments [28,29], 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 α, 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 [25]. 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 i^{th} segment, as well as values of α for each segment.

These two data files are used, together with standard methods [23] from elasticity theory to compute the stress Green's functions (stress transfer coefficients) by 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, where N is the number of fault segments. These stress transfer coefficients, together with the fault slip rate data in **Fault_Data.d** and friction data in **Fault_Friction.d**, are then used as the basic inputs to the earthquake simulation code **EQ_Simulator.c,** which computes the stress evolution on each fault patch for a given time. 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 slip on the i^{th} fault segment are:

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

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

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 $F((Vt - s_i) - \varphi_i^*)$ is an odd nonlinear function of s with amplitude ε_i and parameter φ_i^* [23]. The parameter t_F is any time t at which $\sigma_i(t) \geq \sigma_i^F$, δ is the Dirac delta function, and η is a random noise ("overshoot" or "undershoot"). The nonlinear function $F((Vt - s_i) - \varphi_i^*)$ 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 [30]. Physically, the functions $F((Vt - s_i) - \varphi_i^*)$ and parameters φ_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 [31,32]. The exact form of $F((Vt - s_i) - \varphi_i^*)$ is unimportant, since small fluctuations about $\{\varphi_i^*\}$ are controlled by the first nonlinear term in $F((Vt - s_i) - \varphi_i^*)$, 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 be computed; this deformation 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. [33].

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 C. 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. Again, N is the number of fault segments.

5.0 Conclusions

The promising results from the parallelization of both a genetic algorithm program and a slider block model, using a simple, heterogeneous processor system, existing freeware, and at an extremely low cost of both manpower and hardware dollars, 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. Although we anticipate that the individual calculations will remain small enough for the processor memory, one issue, associated with the amount and variety of information that might have to be passed from master to slave appears to be the greatest problem yet to be solved. However, there are potential solutions such as data consolidation and stored or shared memory. As a result, we believe that conversion of the GEM computational problem, which is essentially a Monte Carlo simulation, is well suited to optimization on parallel computers such as the Maui MHPCC SMP machine or a Beowulf Linux cluster.

Appendix A - Genetic Algorithm Program Code

A.1 Serial

A.1.1 Main Program (Inversion.c)

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

```
    END FOREACH
END WHILE
```

A.1.2 Fitness Function (Fit.c)

```
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 )
```

A.2 Parallel

A.2.1 Master (Inversion Program)

```
main():
initialize_random_genes()
WHILE best_fitness() < target
    select_top_hundred(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
```

A.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
```

Appendix B - Mean Field Slider Block Model

B.1 Serial

```
BEGIN PROGRAM
  Initializes lattice with random stress values
  WHILE current_iteration < total_iterations
        Finds max stress in array
        Resets critical and residual stresses
        Initializes failing_sites list with initiator
        Avalanche()
        Computes averages
        Writes data to file
  END WHILE

  Avalanche():
  WHILE failing_sites list not empty
        Initializes next_failing list
        Computes stress drop for every site in the list AND
              Updates failing sites (subtract stress drop)
        Redistributes stress to all other sites:
              if stress on site larger than threshold, include in next_failing list
        Switches failing_sites and next_failing lists
  END WHILE
END PROGRAM
```

B.2 Parallel

B.2.1 Main, rjbmpi

```
BEGIN PROGRAM
  Assigns to each processor 1 / (number of processors) of lattice
  WHILE current_iteration < total_iterations
          Receives max stress and site index from each slave
          Finds maximum of these max stresses
          Broadcasts max stress and process containing it to all slaves
          Avalanche()
          Receives event totals from each slave
          Computes averages
          Writes data to file
  END WHILE

  Avalanche():
    WHILE ( total number of failed sites > 0 )
          Receives stress drop from each slave
          Sums all stress drops into total stress drop
          Broadcasts total stress drop to all slaves
          Receives number of failed sites from each slave
          Sums number of failed sites into total failed sites
          Broadcasts total number of failed sites
    END WHILE
END PROGRAM
```

B.2.2 Slave, rjbmpi

```
BEGIN PROGRAM
  Assigns to self 1 / (number of processors) of lattice
  Initializes local lattice with random stress values
  WHILE current_iteration < total_iterations
          Finds max stress in local lattice
          Sends local max stress and site index to master
          Receives global max stress and processor its on
          Avalanche()
  END WHILE


  Avalanche():
    WHILE ( total number of failing sites > 0 )
          Computes stress drop for every failing site in local lattice
          Sums stress drops for all failing sites into local stress drop
          Sends local stress drop to master
          Receives total stress drop from master
          Redistributes total stress drop to all sites in lattice AND
                    Tests for new failing sites
```

```
        Sends number of failing sites to master

        Receives total number of failing sites

    END WHILE

END PROGRAM
```

Appendix C - Virtual California Program Code

C.1 Serial

C.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
        }
```

C.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)

        time = time + time_index * time_step;

        WHILE(N<number_of_segments)

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

                Procedure Stress_State_Compute(i,time);

        /*  Check to see which segments have $?_i(t)\ ?\ ?_i^F$  */

                WHILE (any $?_i(t)\ ?\ ?_i^F$ )

                        WHILE (i<N)

                /*      Update slip s(i)   */
                                s(i)= s(i) +  $?\ ?_i / K_i$  +  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
```

C.2 Parallel Code (Master/Slave)

C.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
```

C.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
```

C.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
```

C.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
```

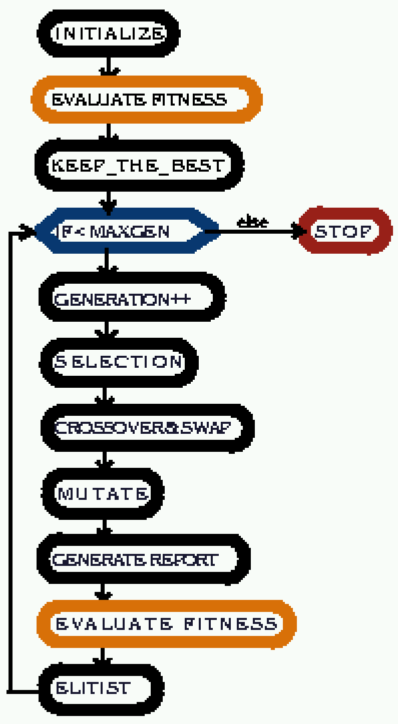


Figure 1: Flow chart, genetic algorithm inversion program.

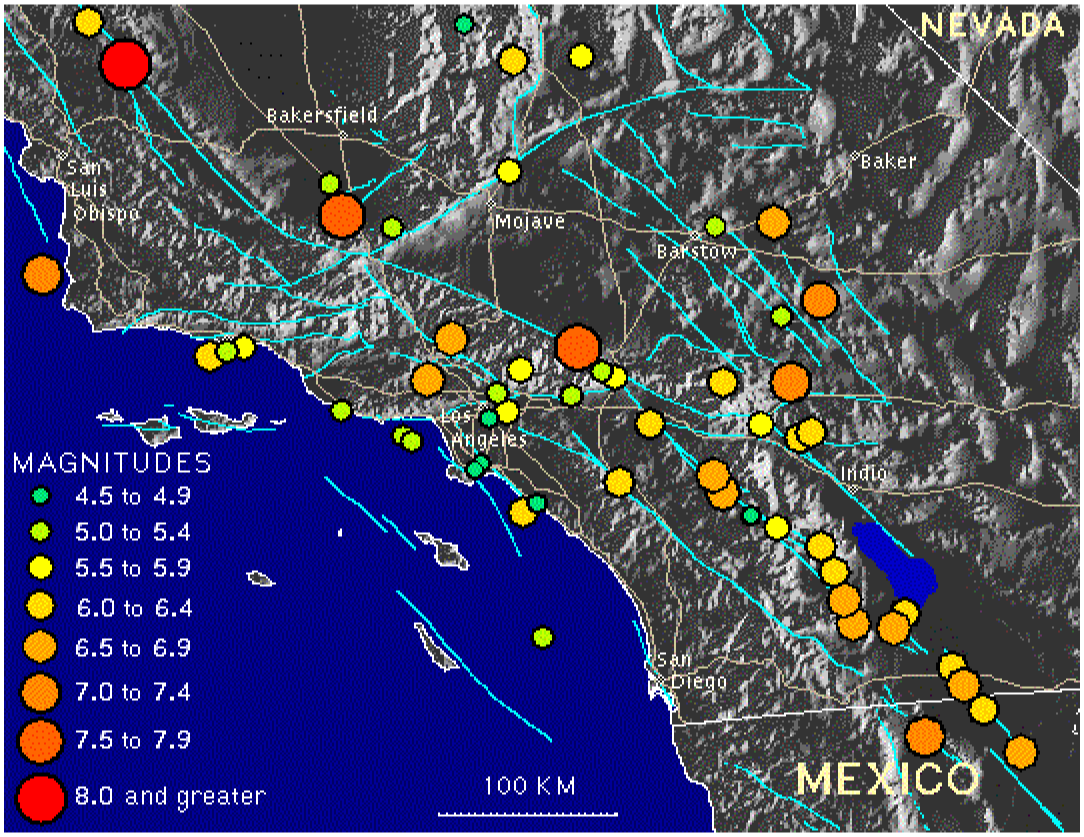


Figure 2: Historic seismicity, southern California (Southern California Earthquake Center, http://www.scecdc.scec.org/clickmap.html).

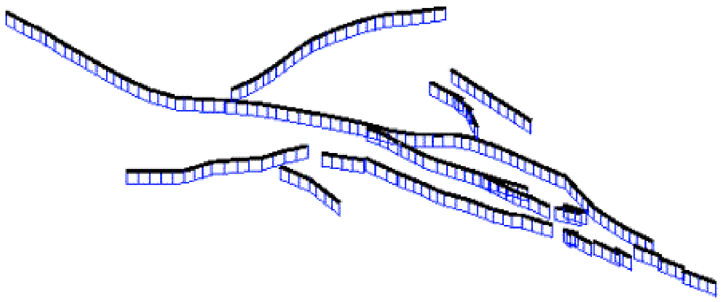
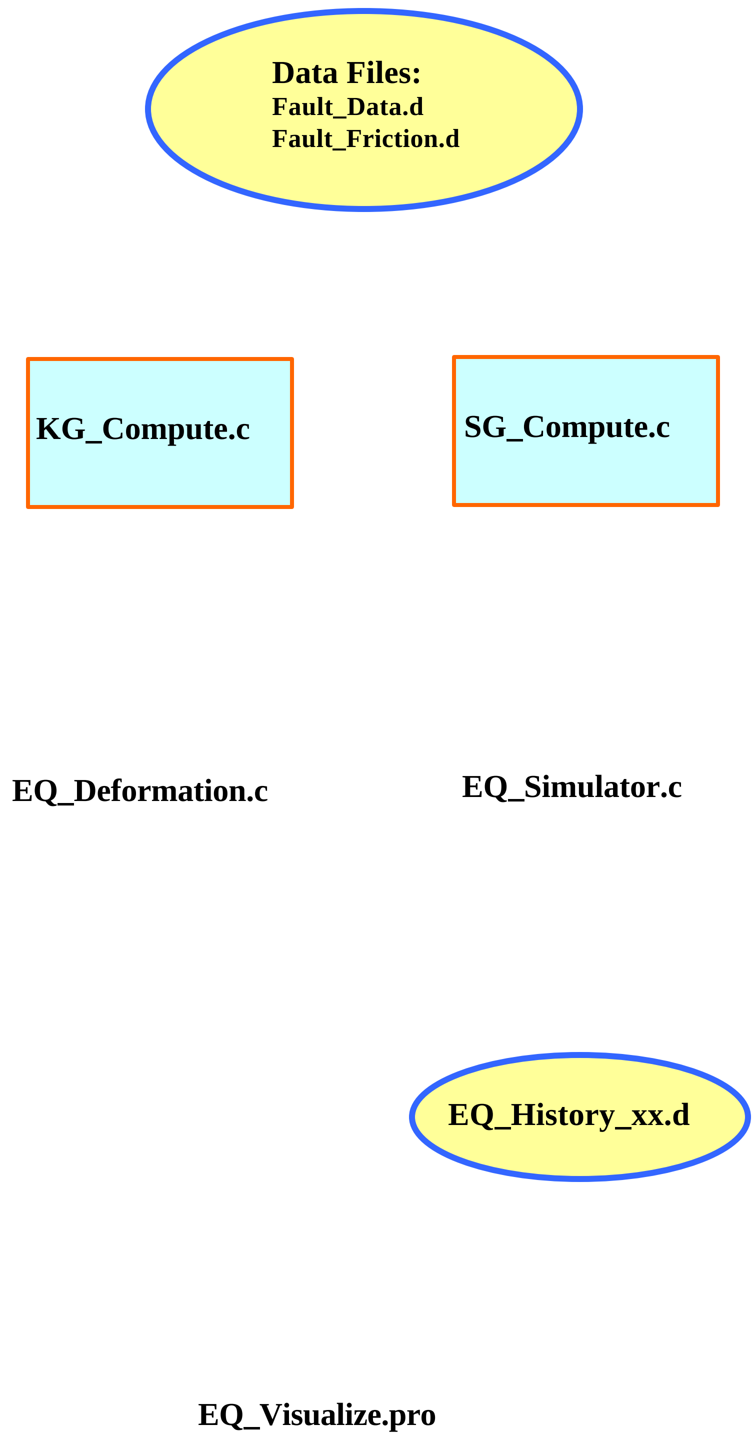


Figure 3: Map of the 215 fault segments used in the implementation of the Virtual California simulation.
Only the strike slip faults are represented in this simplified model.


Data Files:
Fault_Data.d
Fault_Friction.d
KG_Compute.c
SG_Compute.c
EQ_Deformation.c
EQ_Simulator.c
EQ_History_xx.d
EQ_Visualize.pro

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.

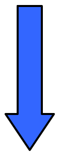


Figure 4: Flow diagram for parallelization of *Virtual California* earthquake simulation program. Green's functions in KG_Compute.c and SG_Compute.c are used to calculate the input to EQ_Deformation.c and EQ_Simulator.c, where the deformation and earthquake slip histories are computed, respectively.

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] T.T. Yu, J. Fernandez, and J.B. Rundle, *Comp. and Geo.*, **24**, 173 (1998).
[10] J. Bhattacharyya, A.F. Sheehan, K.F. Tiampo, and J.B. Rundle, *BSSA*, **89**, 202 (1998).
[11] K.F. Tiampo, J.B. Rundle, J. Fernandez, and J.Langbein, *J. Vol. Geoth. Res.*, **102**, 199 (2000).
[12] J. Fernandez, K.F. Tiampo, G. Jentzsch, M. Charco, and J.B. Rundle, *GRL*, **28**, 2349 (2001).
[13] 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).
[14] S.R. Brown, C.H. Scholz, and J.B. Rundle, *Geophys. Res. Lett.*, **18**, 215 (1991); J.B. Rundle and D.D. Jackson, *Bull. Seismol. Soc. Am.*, **67**, 1363 (1977).
[15] Z. Olami, H.J.S. Feder, and K. Christensen, *Phys. Rev. Lett.*, **68**, 1244 (1992).
[16] C.G. Sammis and S.W. Smith, *Pure Appl. Geophys.*, **155**, 307 (1999).
[17] K. Dahmen, D. Ertas, and Y. Ben-Zion, *Phys. Rev. E*, **58,** 1494 (1998).
[18] J.S. Sá Martins, J.B. Rundle, M. Anghel, and W. Klein, e-print cond-mat/0101343, submitted (2001).
[19] W. Klein, M. Anghel, C.D. Ferguson, J.B. Rundle, and J.S. Sá Martins, *in Geocomplexity and the Physics of Earthquakes*, ed. JB Rundle, DL Turcotte and W. Klein (Amer. Geophys. Un., Washington DC, 2000).
[20] M. Anghel, W. Klein, J.B. Rundle, and J.S. Sá Martins, e-print cond-mat/0002459, submitted (2001).
[21] E.F. Preston, J.S. Sá Martins, J.B. Rundle, M. Anghel, and W. Klein, *Comp. Sci. Eng.*, **2**, 34 (2000).
[22] J.B. Rundle, W. Klein, K.F. Tiampo and S.J. Gross, *Phys. Rev. E*, **61**, 2418 (2000).
[23] J.B. Rundle, *J. Geophys. Res.*, **93**, 6255 (1988).
[24] S.N. Ward, *Bull. Seism. Soc. Am.*, **90**, 370 (2000).

[25] J. Deng and L.R. Sykes, *J. Geophys. Res.*, **102**, 9859 (1997).
[26] B.N.J. Persson, *Sliding Friction, Physical Principles and Applications* (Springer-Verlag, Berlin, 1998).
[27] J. Dieterich, *J. Geophys. Res.*, **84**, 2161 (1979).
[28] T.E. Tullis, *Proc. Nat. Acad. Sci*., **93**, 3803 (1996).
[29] 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)
[30] See for example, R. Peierls, *Phys. Rev*., **54**, 918 (1934).
[31] Rundle, J.B., W. Klein, S. Gross, and D.L. Turcotte, *Phys. Rev. Lett.,* **75**, 1658-1661 (1995).
[32] W. Klein, JB Rundle and CD Ferguson, *Phys. Rev. Lett*., **78**, 3793 (1997).
[33] J.B. Rundle, P.B. Rundle, W. Klein, J.S. Sá Martins, K. F. Tiampo, A. Donnellan, and L. H. Kellogg, *PAGEOPH*, submitted.