# COMPUTATIONAL ASTEROSEISMOLOGY

by

**TRAVIS SCOTT METCALFE, B.S., M.A.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2001

# Chapter 3

# Parallel Genetic Algorithm

> "Evolution is cleverer than you are."
>
> —*Francis Crick*

## 3.1 Background

The problem of extracting useful information from a set of observational data often reduces to finding the set of parameters for some theoretical model which results in the closest match to the observations. If the physical basis of the model is both accurate and complete, then the values of the parameters for the best-fit model can yield important insights into the nature of the object under investigation.

When searching for the best-fit set of parameters, the most fundamental consideration is: where to begin? Models of all but the simplest physical systems are typically non-linear, so finding the least-squares fit to the data requires an initial guess for each parameter. Generally, some iterative procedure is used to improve upon this first guess in order to find the model with the absolute minimum residuals in the multi-dimensional parameter-space.

There are at least two potential problems with this standard approach to model fitting. The initial set of parameters is typically determined by drawing upon the past experience of the person who is fitting the model. This *subjective* method is particularly disturbing when combined with a *local* approach to iterative improvement. Many optimization schemes, such as differential corrections (Proctor & Linnell, 1972) or the simplex method (Kallrath & Linnell, 1987), yield final results which depend to some extent on the initial guesses. The consequences of this sort of behavior are not serious if the parameter-space is well behaved—that is, if it contains a single, well defined minimum. If the parameter-space contains many local minima, then it can be more difficult for the traditional approach to find the global minimum.

## 3.2 Genetic Algorithms

An optimization scheme based on a genetic algorithm (GA) can avoid the problems inherent in more traditional approaches. Restrictions on the range of the parameter-space are imposed only by observations and by the physics of the model. Although the parameter-space so-defined is often quite large, the GA provides a relatively efficient means of searching globally for the best-fit model. While it is difficult for GAs to find precise values for the set of best-fit parameters, they are well suited to search for the *region* of parameter-space that contains the global minimum. In this sense, the GA is an objective means of obtaining a good first guess for a more traditional method which can narrow in on the precise values and uncertainties of the best-fit.

The underlying ideas for genetic algorithms were inspired by Charles Darwin's (1859) notion of biological evolution through natural selection. The basic idea is to solve an optimization problem by *evolving* the best solution from an initial set of completely random guesses. The theoretical model provides the framework within which the evolution takes place, and the individual parameters controlling it serve as the genetic building blocks. Observations provide the selection pressure. A comprehensive description of how to incorporate these ideas in a computational setting was written by Goldberg (1989).

Initially, the parameter-space is filled uniformly with trials consisting of randomly chosen values for each parameter, within a range based on the physics that the parameter is supposed to describe. The model is evaluated for each trial, and the result is compared to the observed data and assigned a *fitness* based on the relative quality of the match. A new generation of trials is then created by selecting from this population at random, weighted by the fitness.

To apply genetic operations to the new generation of trials, their characteristics must be encoded in some manner. The most straightforward way of encoding them is to convert the numerical values of the parameters into a long string of numbers. This string is analogous to a chromosome, and each number represents a gene. For example, a two parameter trial with numerical values $x_1 = 1.234$ and $y_1 = 5.678$ would be encoded into a single string of numbers '12345678'.

Next, the encoded trials are paired up and modified in order to explore new regions of parameter-space. Without this step, the final solution could ultimately be no better than the single best trial contained in the initial population. The two basic operations are *crossover* which emulates sexual reproduction, and *mutation* which emulates happenstance and cosmic rays.

As an example, suppose that the encoded trial above is paired up with another trial having $x_2 = 2.468$ and $y_2 = 3.579$, which encodes to the string '24683579'. The crossover procedure chooses a random position between two numbers along the string, and swaps the two strings from that position to the end. So if the third position is chosen, the strings become

$$123|45678 \rightarrow 123|83579$$
$$246|83579 \rightarrow 246|45678$$

Although there is a high probability of crossover, this operation is not applied to all of the pairs. This helps to keep favorable characteristics from being eliminated or corrupted too hastily. To this same end, the rate of mutation is assigned a relatively low probability. This operation allows for the spontaneous transformation of any particular position on the string into a new randomly chosen value. So if the mutation operation were applied to the sixth position of the second trial, the result might be

$$24645|6|78 \rightarrow 24645|0|78$$

After these operations have been applied, the strings are decoded back into sets of numerical values for the parameters. In this example, the new first string '12383579' becomes $x_1 = 1.238$ and $y_1 = 3.579$ and the new second string '24645078' becomes $x_2 = 2.464$ and $y_2 = 5.078$. This new generation replaces the old one, and the process begins again. The evolution continues until one region of parameter-space remains populated while other regions become essentially empty. The robustness of the solution can be established by running the GA several times with different random initialization.

Genetic algorithms have been used a great deal for optimization problems in other fields, but until recently they have not attracted much attention in astronomy. The application of GAs to problems of astronomical interest was promoted by Charbonneau (1995), who demonstrated the technique by fitting the rotation curves of galaxies, a multiply-periodic signal, and a magneto-hydrodynamic wind model. Many other applications of GAs to astronomical problems have appeared in the recent literature. Hakala (1995) optimized the accretion stream map of an eclipsing polar. Lang (1995) developed an optimum set of image selection criteria for detecting high-energy gamma rays. Kennelly et al. (1995) used radial velocity observations to identify the oscillation modes of a $\delta$ Scuti star. Lazio (1997) searched pulsar timing signals for the signatures of planetary companions. Charbonneau et al. (1998) performed a helioseismic inversion to constrain solar core rotation. Wahde (1998) determined the orbital parameters of interacting galaxies. Metcalfe (1999) used a GA to fit the light curves of an eclipsing binary star. The applicability of GAs to such a wide range of astronomical problems is a testament to their versatility.

## 3.3 Parallelizing PIKAIA

There are only two ways to make a computer program run faster—either make the code more efficient, or run it on a faster machine. We made a few design improvements to the original white dwarf code, but they decreased the runtime by only $\sim$10%. We decided that we really needed access to a faster machine. We looked into the supercomputing facilities available through the university, but the idea of using a supercomputer didn't appeal to us very much; the process seemed to involve a great deal of red tape, and we weren't certain that we could justify time on a supercomputer in any case. To be practical, the GA-based fitting technique required a dedicated instrument to perform the calculations. We designed and configured such an instrument—an isolated network of 64 minimal

PCs running Linux (Metcalfe & Nather, 1999, 2000). To allow the white dwarf code to be run on this metacomputer, we incorporated the message passing routines of the Parallel Virtual Machine (PVM) software into the public-domain genetic algorithm PIKAIA.

### 3.3.1 Parallel Virtual Machine

The PVM software (Geist et al., 1994) allows a collection of networked computers to cooperate on a problem as if they were a single multi-processor parallel machine. All of the software and documentation was free. We had no trouble installing it, and the sample programs that came with the distribution made it easy to learn how to use. The trickiest part of the whole procedure was figuring out how to split up the workload among the various computers.

The GA-based fitting procedure for the white dwarf code quite naturally divided into two basic functions: evolving and pulsating white dwarf models, and manipulating the results from each generation of trials. When we profiled the distribution of execution time for each part of the code, this division became even more obvious. The majority of the computing time was spent evolving the starter model to a specific temperature. The GA is concerned only with collecting and organizing the results of *many* of these models, so it seemed reasonable to allocate many *slave* computers to carry out the model calculations while a *master* computer took care of the GA-related tasks.

In addition to decomposing the function of the code, a further division based on the data was also possible. Since there were many trials in each generation, the data required by the GA could easily be split into small, computationally manageable units. One model could be sent to each available slave computer, so the number of machines available would control the number of models which could be calculated at the same time.

One minor caveat to the decomposition of the data into separate models to be calculated by different computers is the fact that half of the machines are slightly faster than the other half. Much of the potential increase in efficiency from this parallelizing scheme could be lost if fast machines are not sent more models to compute than slow ones. This may seem trivial, but there is no mechanism built in to the current version of the PVM software to handle this procedure automatically.

It is also potentially problematic to send out new jobs only after receiving the results of previous jobs because the computers sometimes hang or crash. Again, this may seem obvious—but unless specifically asked to check, PVM cannot tell the difference between a crashed computer and one that simply takes a long time to compute a model. At the end of a generation of trials, if the master process has not received the results from one of the slave jobs, it would normally just continue to wait for the response indefinitely.

### 3.3.2 The PIKAIA Subroutine

PIKAIA is a self-contained, genetic-algorithm-based optimization subroutine developed by Paul Charbonneau and Barry Knapp at the High Altitude Observatory in Boulder, Colorado. Most optimization techniques work to *minimize* a quantity—like the root-mean-square (r.m.s.) residuals; but it is more natural for a genetic algorithm to *maximize* a quantity—natural selection works through survival of the fittest. So PIKAIA maximizes a specified FORTRAN function through a call in the body of the main program.

Unlike many GA packages available commercially or in the public domain, PIKAIA uses decimal (rather than binary) encoding. Binary operations are usually carried out through platform-dependent functions in FORTRAN, which makes it more difficult to port the code between the Intel and Sun platforms.

PIKAIA incorporates only the two basic genetic operators: uniform one-point crossover, and uniform one-point mutation. The mutation rate can be dynamically adjusted during the evolution, using either the linear distance in parameter-space or the difference in fitness between the best and median solutions in the population. The practice of keeping the best solution from each generation is called elitism, and is a default option in PIKAIA. Selection is based on ranking rather than absolute fitness, and makes use of the Roulette Wheel algorithm. There are three different reproduction plans available in PIKAIA: Steady-State-Delete-Random, Steady-State-Delete-Worst, and Full Generational Replacement. Only the last of these is easily parallelizable.

## 3.4 Master Program

Starting with an improved unreleased version of PIKAIA, we incorporated the message passing routines of PVM into a parallel fitness evaluation subroutine. The original code evaluated the fitnesses of the population of trials one at a time in a DO loop. We replaced this procedure with a single call to a new subroutine that evaluates the fitnesses in parallel on all available processors.

```
c     initialize (random) phenotypes
      do ip=1,np
         do k=1,n
            oldph(k,ip)=urand()
         enddo
c     calculate fitesses
c        fitns(ip) = ff(n,oldph(1,ip))
      enddo
c     calculate fitnesses in parallel
      call pvm_fitness('ff_slave', np, n, oldph, fitns)
```

The parallel version of PIKAIA constitutes the master program which runs on Darwin, the central computer in the network. A full listing of the parallel fitness evaluation subroutine (PVM_FITNESS.F) is included in Appendix C. A flow chart for this code is shown in Figure 3.1.

After starting the slave program on every available processor (64 for our metacomputer), PVM_FITNESS.F sends an array containing the values of the parameters to each slave job over the network. In the first generation of the GA, these values are completely random; in subsequent generations, they are the result of the selection and mutation of the previous generation, performed by the non-parallel portions of PIKAIA.

Next, the subroutine listens for responses from the network and sends a new set of parameters to each slave job as it finishes the previous calculation. When all sets of parameters have been sent out, the subroutine begins looking for jobs that seem to have crashed and re-submits them to slaves that have finished and would otherwise sit idle. If a few jobs do not return a fitness after about five times the average runtime required to compute a model, the subroutine assigns them a fitness of zero. When every set of parameters in the generation have been assigned a fitness value, the subroutine returns to the main program to perform the genetic operations resulting in a new generation of models to calculate. The

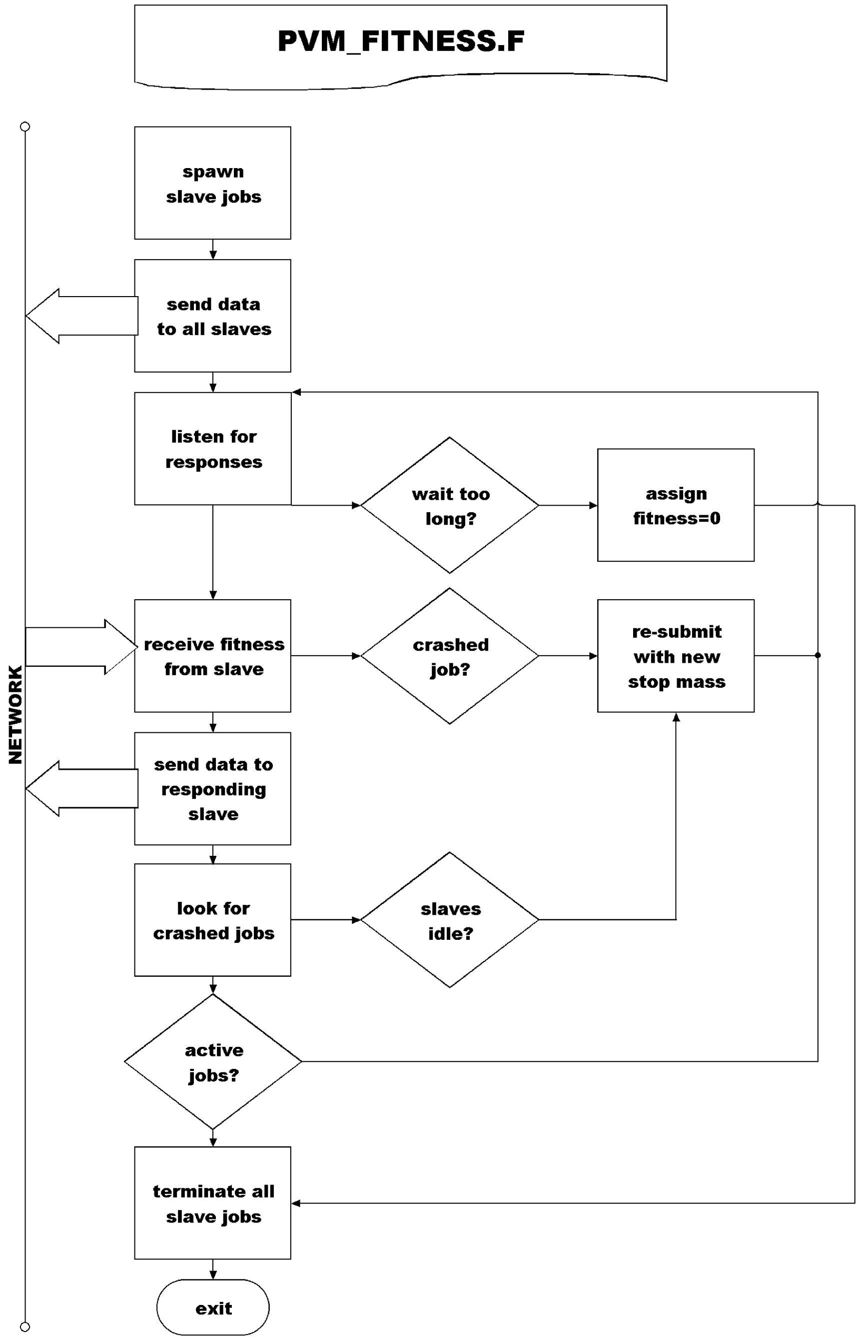


Figure 3.1: Flow chart for the parallel fitness evaluation subroutine, which runs on the master computer.

process continues for a fixed number of generations, chosen to maximize the efficiency of the search. The optimal number of generations is determined by applying the method to a test problem with a known solution.

## 3.5 Slave Program

The original white dwarf code came in three pieces: (1) the evolution code, which evolves a starter model to a specific temperature, (2) the prep code, which converts the output of the evolution code into a different format, and (3) the pulsation code, which uses the output of the prep code to determine the pulsation periods of the model.

To get the white dwarf code to run in an automated way, we merged the three components of the original code into a single program, and added a front end that communicated with the master program through PVM routines. This

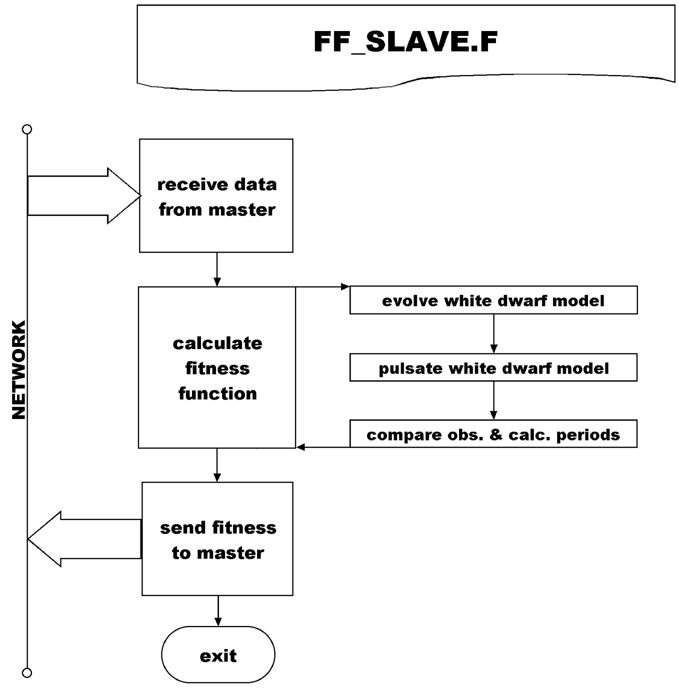


Figure 3.2: Flow chart for the slave program of the parallel code, which runs on each of the 64 nodes of the metacomputer.

code (FF_SLAVE.F) constitutes the slave program, and is run on each node of the metacomputer. A full listing of this code is included in Appendix C, and a flow chart is shown in Figure 3.2.

The operation of the slave program is relatively simple. Once it is started by the master program, it receives a set of parameters from the network. It then calls the fitness function (the white dwarf code) with these parameters as arguments. The fitness function evolves a white dwarf model with characteristics specified by the parameters, determines the pulsation periods of this model, and then compares the calculated periods to the observed periods of a real white dwarf. A fitness based on how well the two sets of periods match is returned to the main program, which sends it to the master program over the network. The node is then ready to run the slave program again and receive a new set of parameters from the master program.