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 2

Linux Metacomputer

> "There's certainly a strong case for people disliking Microsoft because their operating systems... suck."
>
> —*Linus Torvalds*

2.1 Introduction

The adjustable parameters in our computer models of white dwarfs presently include the total mass, the temperature, hydrogen and helium layer masses, core composition, convective efficiency, and internal chemical profiles. Finding a proper set of these to provide a close fit to the observed data is difficult. The traditional procedure is a guess-and-check process guided by intuition and experience, and is far more subjective than we would like. Objective procedures for determining the best-fit model are essential if asteroseismology is to become a widely-accepted and reliable astronomical technique. We must be able to demonstrate that for a given model, within the range of different values the model parameters can assume, we have found the only solution, or the best one if more than one is possible. To address this problem, we have applied a search-and-fit technique employing a genetic algorithm (GA), which can explore the myriad parameter combinations possible and select for us the best one, or ones (cf. Goldberg, 1989; Charbonneau, 1995; Metcalfe, 1999).

2.2 Motivation

Although genetic algorithms are often more efficient than other global techniques, they are still quite demanding computationally. On a reasonably fast computer, it takes about a minute to calculate the pulsation periods of a single white dwarf model. However, finding the best-fit with the GA method requires the evaluation of hundreds of thousands of such models. On a single computer, it would take more than two months to find an answer. To develop this method on a reasonable timescale, we realized that we would need our own parallel computer.

It was January 1998, and the idea of parallel computing using inexpensive personal computer (PC) hardware and the free Linux operating system started getting a lot of attention. The basic idea was to connect a bunch of PCs together on a network, and then to split up the computing workload and use the machines collectively to solve the problem more quickly. Such a machine is known to computer scientists as a *metacomputer*. This differs from a *supercomputer*, which is much more expensive since all of the computing power is integrated into a single unified piece of hardware.

There are several advantages to using a metacomputer rather than a more traditional supercomputer. The primary advantage is price: a metacomputer that is just as fast as a 5-year-old supercomputer can be built for only about 1 percent of the cost—about $10,000 rather than $1 million! Another major advantage is access: the owner and administrator of a parallel computer doesn't need to compete with other researchers for time or resources, and the hardware and software configuration can be optimized for a specific problem. Finally if something breaks, replacement parts are standard off-the-shelf components that are widely available, and while they are on order the computer is still functional at a slightly reduced capacity.

2.3 Hardware

The first Linux metacomputer, known as the Beowulf cluster[5] (Becker et al., 1995), has now become the prototype for many general-purpose Linux clusters.

[5] http://www.beowulf.org/

Our machine is similar to Beowulf in the sense that it consists of many independent PCs, or *nodes*; but our goal was to design a special-purpose computational tool with the best performance possible per dollar, so our machine differs from Beowulf in several important ways.

We wanted to use each node of the metacomputer to run identical tasks (white dwarf pulsation models) with small, independent sets of data (the parameters for each model). The results of the calculations performed by the nodes consisted of just a few numbers (the root-mean-square differences between the observed and calculated pulsation periods) which only needed to be communicated to the master process (the genetic algorithm), never to another node. Essentially, network bandwidth was not an issue because the computation to communication ratio of our application was extremely high, and hard disks were not needed on the nodes because our problem did not require any significant amount of data storage. We settled on a design including one master computer and 64 minimal nodes connected by a simple coaxial network (see Figure 2.1).

We developed the metacomputer in four phases. To demonstrate that we could make the system work, we started with the master computer and only two nodes. When the first phase was operational, we expanded it to a dozen nodes to

Figure 2.1: The 64 minimal nodes of the metacomputer on shelves surrounding the master computer.

demonstrate that the performance would scale. In the third phase, we occupied the entire bottom shelf with a total of 32 nodes. Months later, we were given the opportunity to expand the system by an additional 32 nodes with processors donated by AMD and we filled the top shelf, yielding a total of 64 nodes.

2.3.1 Master Computer

Our master computer, which we call Darwin, is a Pentium-II 333 MHz system with 128 MB RAM and two 8.4 GB hard disks (see Figure 2.2). It has three NE-2000 compatible network cards, each of which drives 1/3 of the nodes on a subnet. No more than 30 devices (e.g. ethernet cards in the nodes) can be included on a single subnet without using a repeater to boost the signal. Additional ethernet cards for the master computer were significantly less expensive than a repeater.

Figure 2.2: Darwin, the master computer controlling all 64 nodes.

2.3.2 Slave Nodes

We assembled the nodes from components obtained at a local discount computer outlet. Each node includes only an ATX tower case and power supply with a motherboard, a processor and fan, a single 32 MB RAM chip, and an NE-2000 compatible network card (see Figure 2.3). Half of the nodes contain Pentium-II 300 MHz processors, while the other half are AMD K6-II 450 MHz chips. We added inexpensive Am27C256 32 kb EPROMs (erasable programmable read-only memory) to the bootrom sockets of each network card. The nodes are connected in series with 3-ft ethernet coaxial cables, and the subnets have 50 Ω terminators on each end. The total cost of the system was around $25,000 but it could be built for considerably less today, and less still tomorrow.

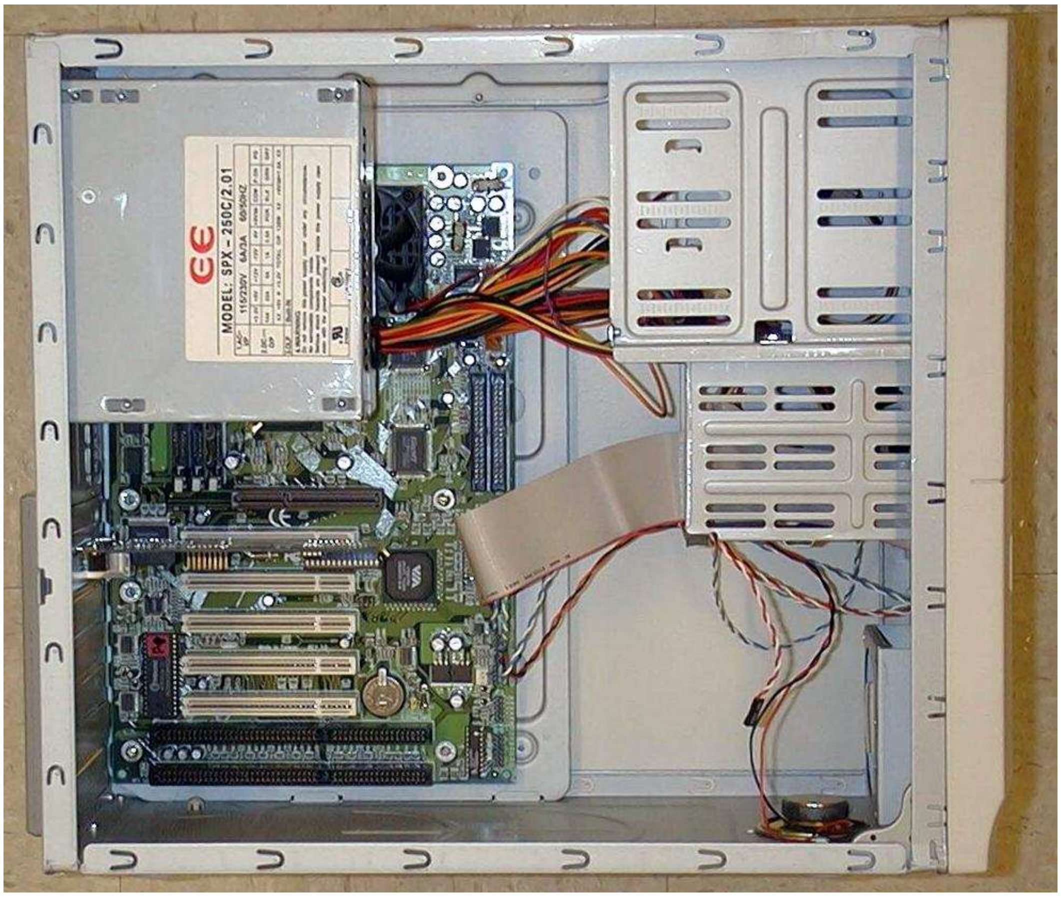

Figure 2.3: A view inside one of the metacomputer nodes.

2.4 Software

Configuring the software was not much more complicated than setting up a diskless Linux system. The main difference was that we wanted to minimize network activity by giving each node an identical, independent filesystem rather than mounting a shared network filesystem. Since the nodes had no hard disks, we needed to create a self-contained filesystem that could be downloaded and mounted in a modest fraction of the 32 MB RAM.

To make the system work, we relied heavily on the open-source Linux operating system and other software packages that were available for free on the Internet. A piece of software called YARD allowed us to create the minimal Linux filesystems that we needed for each node to run independently. We used a package called NETBOOT to program the EPROM chips; this allowed each node to automatically download and boot the filesystem, which was mounted in part of the RAM. Finally, we used the PVM software to exploit the available resources and take care of the communication required to manage the parallel processing operations.

2.4.1 Linux

In 1991, a young student named Linus Torvalds at the University of Helsinki in Finland created a free Unix-like operating system as a hobby. He posted his work on the Internet and, together with a small group of friends, he continued to develop it. In 1994, version 1.0 of "Linux" was released. Today, millions of people worldwide use Linux as an alternative to the operating systems sold by Microsoft (Windows) and Sun Microsystems (Solaris). Unlike these more common operating systems, the source code for Linux is freely available to everyone.

The computer code used to create the Linux operating system is known as the *kernel*. To ensure that the hardware components of our nodes would be recognized by the operating system, we custom compiled the Linux 2.0.34 kernel. We included support for the NE-2000 ethernet card, and specified that the filesystem was on the network and should be retrieved using the `bootp` protocol (see below) and mounted in RAM.

Getting the master computer to recognize its three ethernet cards required extra options to be passed to the kernel at boot time. We specified the addresses of these devices and passed them to the Linux kernel through LOADLIN, a DOS-based program that boots up Linux.

Each card on the network needed to be assigned a unique IP (Internet Protocol) address, which is a sequence of four numbers between 0 and 255 separated by periods. The IP addresses that are reserved for subnets (which do not operate on the Internet) are:

10.0.0.0	(Class A network)
172.16.0.0 → 172.31.0.0	(Class B network)
192.168.0.0 → 192.168.255.0	(Class C network)

Since we were dealing with a relatively small number of machines, we used the first three numbers to specify the domain (pvm.net), and the last number to specify the hostname (e.g. node001). Our first ethernet card (`eth1`) was assigned control of the 192.168.1.0 subnet while 192.168.2.0 and 192.168.3.0 were handled by `eth2` and `eth3` respectively.

We used the Bootstrap Protocol (`bootp`) and the Trivial File Transfer Protocol (`tftp`) to allow the nodes to retrieve and boot their kernel, and to download a compressed version of their root filesystem. We relied heavily on Robert Nemkin's Diskless HOWTO[6] to make it work.

The main configuration file for `bootp` is `/etc/bootptab`, which contains a list of the hostnames and IP addresses that correspond to each ethernet card on the subnet. Each card is identified by a unique hardware address—a series of 12 hexadecimal numbers (0-9,a-f) assigned by the manufacturer. In addition to various network configuration parameters, this file also describes the location of the bootimage to retrieve with `tftp`. Since each node is running an identical copy of the bootimage, setting up `tftp` was considerably easier than it would have been in general. We simply created a `/tftpboot` directory on the server and placed a copy of the bootimage there.

[6]http://www.linuxdoc.org/HOWTO/Diskless-HOWTO.html

2.4.2 YARD

To create the self-contained root filesystem, we used Tom Fawcett's YARD (Yet Another Rescue Disk) package[7]. This piece of software was designed to make rescue disks—self-contained minimal filesystems that can fit on a single floppy disk and be used in emergencies to boot and fix problems on a Linux system. Since the white dwarf pulsation code does not require a great deal of system memory to run, we were free to use half of the 32 MB RAM for our filesystem, which allowed us to include much more than would fit on a floppy disk.

There are two files that control the properties and content of the YARD filesystem: `Config.pl` and `Bootdisk_Contents`. The `Config.pl` file controls the size of the filesystem, the location of the kernel image, and other logistical matters. The `Bootdisk_Contents` file contains a list of the daemons, devices, directories, files, executable programs, libraries, and utilities that we explicitly wanted to include in the filesystem. The scripts that come with YARD automatically determine the external dependences of anything included, and add those to the filesystem before compressing the whole thing.

2.4.3 NETBOOT

We used Gero Kuhlmann's NETBOOT package[8] to create the bootimage that each node downloads from the master computer. The bootimage is really just a concatenated copy of the Linux kernel (`zImage.node`) and the compressed root filesystem (`root.gz`). The NETBOOT software also includes a utility for creating a ROM image that is used to program the EPROMs in the ethernet card for each node. Although our ROM image was only 16 kb, we used Am27C256 (32 kb) EPROMs because they were actually cheaper than the smaller chips.

2.4.4 PVM

The Parallel Virtual Machine (PVM) software[9] allows a collection of computers connected by a network to cooperate on a problem as if they were a single multi-

[7] http://www.croftj.net/~fawcett/yard/

[8] http://www.han.de/~gero/netboot/

[9] http://www.epm.ornl.gov/pvm/

processor parallel machine. It was developed in the 1990's at Oak Ridge National Laboratory (Geist et al., 1994). The software consists of a daemon running on each host in the virtual machine, and a library of routines that need to be incorporated into a computer program so that it can utilize all of the available computing power.

2.5 How it works

With the master computer up and running, we turn on one node at a time (to prevent the server from being overwhelmed by many simultaneous `bootp` requests). By default, the node tries to boot from the network first. It finds the bootrom on the ethernet card, and executes the ROM program. This program initializes the ethernet card and broadcasts a `bootp` request over the network.

When the server receives the request, it identifies the unique hardware address, assigns the corresponding IP address from the `/etc/bootptab` file, and allows the requesting node to download the bootimage. The node loads the Linux kernel image into memory, creates a 16 MB initial ramdisk, mounts the root filesystem, and starts all essential services and daemons.

Once all of the nodes are up, we start the PVM daemons on each node from the master computer. Any computer program that incorporates the PVM library routines and has been included in the root filesystem can then be run in parallel.

2.6 Benchmarks

Measuring the absolute performance of the metacomputer is difficult because the result strongly depends on the fraction of Floating-point Division operations (FDIVs) used in the benchmark code. Table 2.1 lists four different measures of the absolute speed in Millions of FLoating-point Operations Per Second (MFLOPS).

The code for MFLOPS(1) is essentially scalar, which means that it cannot exploit any advantages that are intrinsic to processor instruction sets; the percentage of FDIVs (9.6%) is considered somewhat high. The code for MFLOPS(2) is fully vectorizable, which means that it can exploit advantages intrinsic to each processor, but the percentage of FDIVs (9.2%) is still on the high side. The code

Table 2.1: The Absolute Speed of the Metacomputer

Benchmark	P-II 300 MHz	K6-II 450 MHz	Total Speed
MFLOPS(1)	80.6	65.1	4662.4
MFLOPS(2)	47.9	67.7	3699.2
MFLOPS(3)	56.8	106.9	7056.0
MFLOPS(4)	65.5	158.9	7180.8

for MFLOPS(3) is also fully vectorizable and the percentage of FDIVs (3.4%) is considered moderate. The code for MFLOPS(4) is fully vectorizable, but the percentage of FDIVs is zero. We feel that MFLOPS(3) provides the best measure of the expected performance for the white dwarf code because of the moderate percentage of FDIVs. Adopting this value, we have achieved a price to performance ratio near $3.50/MFLOPS.

The relative speed of the metacomputer is easy to measure. We simply compare the amount of time required to compute a specified number of white dwarf models using all 64 nodes to the amount of time required to calculate the same number of models using only one of the nodes. We find that the metacomputer is about 60 times faster than a single node by itself.

2.7 Stumbling Blocks

After more than 3 months without incident, one of the nodes abruptly died. One of the graduate students working in our lab reported, “One of your babies is crying!” As it turned out, the power supply had gone bad, frying the motherboard and the CPU fan. The processor overheated, shut itself off, and triggered an alarm. We now keep a few spare CPU fans and power supplies on hand. This is the only real problem we have had with the system, and it was easily fixed.

Since the first incident, this scenario has repeated itself five times over a three year period. This implies that such events can be expected at the rate of 2 per year for this many nodes. In addition to the more serious failures, there have been ten other power supply failures which did not result in peripheral hardware damage. The rate for these failures is 3-4 per year.