A Note on Embarrassingly Parallel Computation using OpenMosix and Ox *

Max Bruche
Financial Markets Group
London School of Economics
Houghton Street
London WC1A 2AE, U. K.
m.e.bruche@lse.ac.uk

This version: August 13, 2003

Abstract

OpenMosix is a Linux kernel extension that facilitates the construction of high-powered clusters of computers from simple PCs. Processes that are run on the cluster will automatically be moved by the operating system to nodes that are less busy to optimise resource allocation. Any application that can be split up into several processes can take advantage of this immediately without any need to rewrite code. Since Ox does not do this automatically, a simple workaround for embarrassingly parallel problems like Monte Carlo simulations is to start several instances of Ox performing part of the calculations with e. g. a simple shell script. OpenMosix then automatically distributes the running instances of Ox among the nodes of the cluster. This note describes the author's experience in implementing such a cluster.

1 Introduction

1.1 The need for computing power

In econometrics, economics and finance, there exist many problems that are only tractable with heavy computation. A subset of these problems can be described as "parallel", in the sense that the task can be broken up into smaller sub-tasks to be allocated to several different computers, processors, or "nodes". Problems are called "embarrassingly parallel" whenever this splitting up into tasks is embarrassingly easy to achieve. This will be the case e. g. whenever the nodes dealing with the sub-tasks will not have to communicate much. Typically, in this case adding nodes to completing the task produces a near-linear increase in computational speed.

*I would like to thank Jon Danielsson and Oliver Linton for providing the computers for the OpenMosix cluster. Comments on this note are very welcome.

A typical example of an embarrassingly parallel problem would be a Monte Carlo simulation to examine the small-sample properties of an estimator. Artificial datasets are generated and the estimation is run on each of these artificial datasets. This generates a small sample distribution of the estimator. Of course, the data generation and estimation are steps that can easily be spread across several nodes.

1.2 The evolution of hardware and software

In the early days of computing, hardware was expensive. Research centres would buy large mainframe computers to be used by many different users. Since often people would be doing things on these machines in parallel or executing parallel tasks on these machines, they were built with several processors. A lot of these machines are still sold today, e. g. by IBM, DEC, SGI, Sun Microsystems, and Hewlett-Packard to name but a few. Usually, these large machines will run variations of the Unix operating system, which was specifically designed for such machines.

With the advent of the personal computer, however, a more cost effective solution presented itself in hooking up large numbers of off-the-shelf components to produce a cluster with computing power to rival the large machines. These clusters can either consist of dedicated machines and are then often called Beowulf clusters, or of workstations that are hooked up together at night when they are not in use (a Cluster of Workstations).

A natural operating system to run on a dedicated Beowulf cluster is Linux, the free version of Unix. It presents an environment very similar to the proprietary Unices, with the added advantages that one does not have to pay license fees for its use, and that it is open source, i. e. possible to modify it (if one so wishes) to suit one's purposes.

1.3 System architectures

Essentially, there are two different ways to implement parallelisation: Communication via shared memory and message passing. To illustrate different ways of implementing parallelisation, the Beowulf HOWTO [1] suggests the analogy of waiting in long lines at stores before the checkout: Suppose that a cash register/ cashier represents a processor and each customer is a computer program. There are several possible ways of processing customers.

1.3.1 Single-processor machines

There is only one cash register.

Single-tasking operating system Everyone has to queue at the cash register to pay.

[1] http://www.tldp.org/HOWTO/Beowulf-HOWTO.html

Multi-tasking operating system A little bit of each order is processed in turn, giving customers the satisfaction that their order is being processed. Of course, this is slow and not really any faster than if they had to wait their turn.

1.3.2 Multi-processor machines

There are several cash registers.

Multi-tasking operating systems Several queues are formed, one in front of each cash-register. Your order is processed, but never faster than it would be if you were the only person in front of one cash register. This is also called symmetric multi-processing (SMP).

Threads on a multi-tasking operating system Suppose there is one huge shopping trolley of 1000 items. Suppose you split that up into 10 shopping trolleys (you could call these 'threads') of a 100 items each. If you have ten cash registers, than each cashier could work on one trolley, and write the subtotal on a blackboard. Once finished, the cashiers could add up the subtotals to get the total. Of course, there is a limit to how many cash registers you can have in one place (close to a blackboard).

Messages on multi-tasking operating systems Suppose in your department store, we have a group of 5 cash registers on the first floor, and 5 cash registers on the second floor. If we have a very large shopping trolley with a 1000 items, we could break it up into 10 shopping trolley of 100 items each as before, and send half of the trolleys to the first floor, and the other half to the second floor. On each floor, the cashiers would each do one trolley and write their subtotals on a blackboard. They would add up their subtotals. When they are finished, they can communicate over telephone to add it all up to the grand total.

The difference between the last two versions is the method of communication. Threads communicate by reading and writing from shared memory (the blackboard), whereas message passing is based on ... passing messages between different computers (floors). In general, it is easy to see that which of these methods works best will depend on the type of parallel task to be executed, and the speed of communication via memory (the blackboard) versus the speed of communication via messages (the phone). A lot of the large computers running proprietary flavours of Unix that are sold by specialised vendors such as SGI or IBM are SMP machines where processors communicate via shared memory, whereas Beowulf clusters are usually built out of cheap computers with local memory that communicate via messages (typically over Ethernet).

1.4 Levels of parallelisation

Given a particular architecture, it is clear that whatever the method of communication is, there are several levels at which parallelisation can be implemented.

1. Compilers
   Compilers translate low level languages like C into assembly code and

then into machine code. If the compiler is aware of the architecture of the machine, including how communication between different processors is possible, it can automatically parallelise code to take account of this.

2. Code in low level languages like C
   If the parallelisation is to be effected via passing messages, it can also be coded directly in a low level language like C. There are APIs that are used for this purpose, the most prevalent being the Message Passing Interface (MPI[2]) and the Parallel Virtual Machine (PVM[3]).

3. Operating system
   Alternatively, an operating system can be designed which automatically takes care of (at least some of the) parallelisation. An example is OpenMosix (to be discussed below).

4. A high level language like Ox
   Standard routines like numerical maximisation functions could be implemented in Ox to automatically take advantage of a parallel architecture if present.

5. Explicit code in a high level language like Ox
   The parallelisation (message passing) could be done explicitly in code written in a high-level language like Ox.

Of course, combinations of the approaches are also possible.

Doornik, Hendry, and Shephard (2002) use C code based on MPI from within an Ox program to achieve parallelisation. While writing special code for a parallel architecture (be it an SMP machine, a Beowulf cluster or a Cluster of Workstations) allows more control over how parallelisation is achieved and the resulting program is likely to be more efficient, it is often tiresome. The easiest solution from point of view of a user often is to have parallelisation already embedded in a compiler, an operating system or a high-level language.

1.5 OpenMosix

A version of an operating system that is able to parallelise computations to some extent is readily available in the form of OpenMosix. Originally, MOSIX was a project started by Amnon Barak of Hebrew University of Jerusalem[4]. Some disagreements about whether the code should be freely available and redistributable eventually led to a split in the development between MOSIX and what is now called OpenMosix[5], headed by Barak's former student Moshe Bar. All OpenMosix code is released under the GPL license[6], which implies that it is freely available, redistributable and open source, as well as ensuring that it will remain freely available, redistributable and open source.

[2] http://www-unix.mcs.anl.gov/mpi/
[3] http://www.csm.ornl.gov/pvm/pvm_home.html
[4] http://www.mosix.org
[5] http://sourceforge.openmosix.net
[6] http://www.gnu.org/copyleft/gpl.html

OpenMosix is only a patched version of the Linux kernel, any code that runs under Linux will also run under OpenMosix. In particular, the Ox version for Linux will run under OpenMosix (the author has tested version Ox version 3.3).

OpenMosix is a so called Single System Image (SSI) cluster operating system, which means that to the user, the cluster appears as a single machine. Users log into the cluster (and not into particular machines that are part of the cluster), and end up in one of the nodes of the cluster. They can start a process, and do not need to think about which node the process is started on, as the operating system will take care of distributing running processes across nodes in an optimal fashion. In terms of our earlier analogy, customers are automatically sent to the floor where the cash registers are the least busy. This in effect allows the cluster to operate like a multi-processor multi-tasking operating system.

1.5.1 Communication under OpenMosix

OpenMosix unfortunately does not work with multi-threading (at least at the moment): Multi-threaded applications are not migrated across nodes. Communication between processes under OpenMosix can be achieved, however, e. g. via message-passing. In this case, the advantage of using OpenMosix rather than a different operating system comes from the fact that the operating system will automatically achieve an optimal balance of processes across the nodes, so that the programmer will not have to worry about this aspect.

Of course, this means that code will still have to be rewritten. For many applications such as Monte Carlo studies, however, the communication needs are virtually nil (these are the "embarrassingly parallel" problems). In these cases, it is often possible to allow communication via the filesystem. For example, several processes could be started up each performing estimation on some different randomly generated data. The output (estimates and tests) are saved. When all processes have stopped running, the saved output can be aggregated. As it turns out, this works very well in the practical example discussed below.

1.6 An application: State space models via Simulated Maximum Likelihood Estimation

Durbin and Koopman (1997) propose a methodology to evaluate the likelihood of a non-linear and/ or non-Gaussian latent variable model via importance sampling: Essentially, they propose calculating the likelihood of an approximating model and then correcting for the approximation by multiplying with a factor that is related to the ratio of the true density to the approximate density. This factor needs to be calculated by simulation (for a more detailed description of the estimation methodology, please see the appendix). For complicated problems, maximising the likelihood by evaluating it numerically can take quite long. A Monte Carlo experiment to examine the properties of the ML estimator based on this method can potentially take up a lot of computing time, depending on the complexity of the model being estimated.

The author was faced with having to produce a Monte Carlo study of an estimator based on this method, with code written in Ox using SsfPack (Koopman,

Shephard, and Doornik 1999). On a MS Windows 2000 machine (Intel Pentium 4, 2.6 Ghz), 1000 simulation runs took 32 hours for the simplest model. On the same machine, running an optimized Gentoo Linux (2.4.20) distribution compiled from source for this specific machine, 1000 simulation runs took 24 hours for the simplest model. Since some preliminary investigation indicated that a more complex model would increase computational time easily by a factor of about 10, a decision was made to investigate running Ox on a cluster of OpenMosix machines.

An embarrassingly parallel calculation like the example discussed can be performed on a cluster of machines running OpenMosix simply by starting up several processes that are then automatically spread over the available nodes. For an Ox program, this can be achieved with some simple shell commands, e. g.:

```
oxl myMonteCarlo.ox &
oxl myMonteCarlo.ox &
...
oxl myMonteCarlo.ox &
```

The processes save their output (the estimates and the tests). Once the calculations are completed, we simply need to load this output and aggregate to get e. g. the desired moments of the small sample distribution of our estimator. Since in this application, there is no need for communication between the different threads, it is very simple to run under OpenMosix.

From the user's perspective, all that is needed is to write code that saves output, and possible some code that aggregates results from the saved output. Then, several instances of the code have to be started, and the processes migrate to different nodes on the cluster automatically, where the operating system ensures the balancing of the nodes. Note that this would also work well for very heterogeneous clusters made up of computers of different speeds: OpenMosix would simply allocate more processes to the faster machines.

2 The cluster setup

The OpenMosix cluster was first built out of 2 and later out of 4 identical machines with Pentium 4 processors (2.6 Ghz), and 512MB RAM each, small hard-disks, cheap CD-ROMs, Fast Ethernet (100Mbps) cards and a small Shuttle chassis (to economise on space). It might be possible to build a faster cluster by either using Gigabit Ethernet (1000Mbps) or Firewire (IE1394)[7], although latency is more likely to be an issue than bandwidth for most applications.

The machines are connected with a combined Fast Ethernet switch / router. This serves to isolate the machines from the Local Area Network in two ways: Firstly, the OpenMosix traffic does not go over the general LAN, and so cannot interfere with other traffic, and other traffic cannot interfere with the OpenMosix

[7] http://howto.ipng.be/FireWireClustering/

inter-node communication. Secondly, the machines are hidden behind a router with most ports closed, which means that they are harder to find for potential attackers (see figure 2).

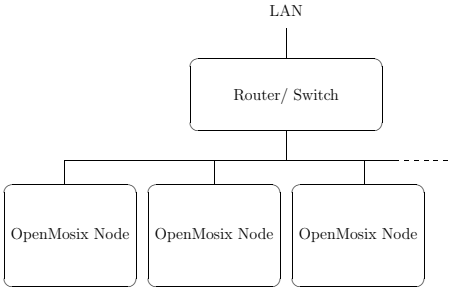

Figure 1: Network layout

All machines run an OpenMosix-patched Gentoo Linux (2.4.20) kernel[8], and have Ox version 3.3 for Linux installed.

It is possible (but more involved) to setup a cluster such that nodes boot over the network. This can be useful if one is interested in having nodes that consist only of a processor, a motherboard and some memory (i.e. no hard-disks, CD-ROMs etc.), or to set up a cluster of workstations to boot into OpenMosix at night. More information on this can be found e. g. at the website of the Linux Terminal Server Project[9] or at linuxhpc.org[10].

3 Performance

To compare the performance increase on the cluster, the same Ox code was run on a single machine, running first MS Windows 2000, then Gentoo Linux (2.4.20), then on a two-machine OpenMosix Cluster, and then on a four machine OpenMosix cluster. On the single-processor machines, the code was run in a single thread. On the cluster, the code was run in 10 separate processes, to allow the kernel enough scope for effective load-balancing. The table below reports the results:

Number of machines	Operating System	time
1	MS Windows 2000	32 hours
1	Gentoo Linux (2.4.20)	23 hours
2	Gentoo Linux/OpenMosix	10.5 hours
4	Gentoo Linux/OpenMosix	6.15 hours

We can see that the time it takes to run the simulation is approximately halved when using two computers. Overall the computation seems to scale up very well.

[8] http://www.gentoo.org
[9] http://www.ltsp.org
[10] http://www.linuxhpc.org

4 Conclusion

This note has described how a simple OpenMosix cluster setup can be used for running Ox programs in parallel. It is likely that for most 'embarrassingly parallel' problems like the one described here, processing power scales up approximately linearly with an increased number of nodes. OpenMosix is a very simple and straightforward way to run embarrassingly parallel computations on a cluster with no (or minimal) changes to existing code, and is a viable alternative to message-passing-based parallelisation in the code as proposed by Doornik, Hendry, and Shephard (2002), albeit a less sophisticated one, and one that is likely to be inappropriate for less parallel problems.

References

Doornik, J., D. Hendry, and N. Shephard, 2002, "Computationally-intensive Econometrics using a Distributed Matrix-programming Language," *Philosophical Transactions of the Royal Society, Series A*, 360, 1245–1266.

Doornik, J. A., 2002, *Objected-Oriented Matrix Programming Using Ox*, Timberlake Consultants Press, London, UK, 3 edn.

Durbin, J., and S. J. Koopman, 1997, "Monte Carlo Maximum Likelihood Estimation for Non-Gaussian State Space Models," *Biometrika*, 84, 669–684.

Durbin, J., and S. J. Koopman, 2001, *Time Series Analysis by State Space Methods*, Oxford University Press, Oxford, UK.

Koopman, S. J., N. Shephard, and J. A. Doornik, 1999, "Statistical Algorithms for Models in State Space Using SsfPack 2.2," *Econometrics Journal*, 2, 113–166.

Ripley, B. D., 1987, *Stochastic Simulation*, Wiley, New York, USA.

Shephard, N., and M. K. Pitt, 1997, "Likelihood Analysis of Non-Gaussian Measurement Time Series," *Biometrika*, 84, 653–667.

5 Appendix

5.1 Simulated Maximum Likelihood

Suppose we have a system

$$\alpha_t = d + \alpha_{t-1} + \eta_t \tag{1}$$
$$y_t = Z_t\left(\alpha_t, \psi_z\right) + \varepsilon_t, \tag{2}$$

where α is the latent (unobserved) variable and y is the observed variable, then typically, it is relatively easy to write down the joint density $p(y, \alpha)$. However, depending on the non-linear function $Z(\cdot)$ and the distribution of the errors, it can be very difficult to write down $p(y)$, which is the likelihood function we are interested in.

Using the original notation as much as possible, this likelihood can be evaluated numerically using the following procedure: Defining the likelihood as

$$L(\psi) = p(y \mid \psi) = \int p(\alpha,\ y \mid \psi)\, d\alpha. \tag{3}$$

we use importance sampling (c.f. e.g. Ripley 1987) from a density $g(\alpha \mid y,\ \psi)$ to evaluate this density. An obvious choice for the importance density is the density of a linear Gaussian model, since it is straightforward to handle. We can now write the likelihood as

$$L(\psi) = g(y) \int \frac{p(a,\ y \mid \psi)}{g(a,\ y \mid \psi)} g(\alpha \mid y,\ \psi)\, d\alpha \tag{4}$$

$$= L_g(\psi)\, E_g[w(a,\ y)]. \tag{5}$$

Where L_g is the likelihood of the approximating linear Gaussian model and

$$w(a,\ y) = \frac{p(a,\ y \mid \psi)}{g(a,\ y \mid \psi)}. \tag{6}$$

We can interpret this likelihood function as consisting of the likelihood of the approximating model, multiplied by a factor to correct for the approximation.

The likelihood function of the approximating model can be calculated using the Kalman filter, and the correction factor $E_g[w(a,\ y)]$ can easily be evaluated using Monte-Carlo techniques. A procedure for estimating the likelihood would therefore be

1. Calculate the linear Gaussian approximation.
2. Calculate the likelihood of the linear Gaussian approximation.
3. To obtain a correction factor, simulate α from the importance density, and numerically calculate the expectation term.

We can write

$$\hat{L}(\psi) = L_g(\psi)\, \bar{w} \tag{7}$$

where

$$\bar{w} = \frac{1}{M} \sum_{i=1}^{M} w_i, \quad w_i = \frac{p(\alpha^i,\ y \mid \psi)}{g(a^i,\ y \mid \psi)}, \tag{8}$$

and α^i is drawn from the importance density. The accuracy of this numerically evaluated likelihood only depends on M, the size of the Monte Carlo simulation. In practice, the log transformation of the likelihood is used. This introduces a bias for which a modification has been suggested that corrects

for terms up to order $O\left(M^{-3/2}\right)$ (cf. Shephard and Pitt 1997, Durbin and Koopman 1997):

$$\log \hat{L}(\psi) = \log L_g(\psi) + \log \bar{w} + \frac{s_w^2}{2M\bar{w}^2},$$

with $s_w^2 = (M-1)^{-1} \sum_{i=1}^{M} (w_i - \bar{w})^2$.

The technique described here allows the numerical evaluation of the likelihood function and hence its numerical maximisation. For examples in which this technique has been applied, c.f. e.g. the book by Durbin and Koopman (2001).