Multi-core CPUs, Clusters, and Grid Computing: a Tutorial*

Michael Creel
Department of Economics and Economic History
Edifici B, Universitat Autònoma de Barcelona
08193 Bellaterra (Barcelona) Spain
michael.creel@uab.es

and

William L. Goffe
Department of Economics
SUNY—Oswego
416 Mahar Hall
Oswego, NY 13126
goffe@oswego.edu

June, 2005

Abstract

The nature of computing is changing and it poses both challenges and opportunities for economists. Instead of increasing clock speed, future microprocessors will have "multi-cores" with separate execution units. "Threads" or other multi-processing techniques that are rarely used today are required to take full advantage of them. Beyond one machine, it has become easy to harness multiple computers to work in clusters. Besides dedicated clusters, they can be made up of unused lab computers or even your colleagues' machines. Finally, grids of computers spanning the Internet are now becoming a reality and one is ready for use by economists.

*We would like to thank, without implicating, Ken Judd, Aaron Reece, and Robert J. Tetlow.

1 Introduction

For nearly two decades, economists have used desktop computers with a single processor for most of their computing needs. Each year their power has increased and economists have been able to tackle more difficult problems. This era is coming to an end. Future improvements in microprocessors will come from adding additional processors to each chip; the power of a single processor is unlikely to substantially change. This transition will not be transparent to programmers or sophisticated users. Soon economists with a computational bent will be programming "multi-core" microprocessors with tools they likely have not used before.

At the same time, improvements in software and networking are enabling organizations to join computers together in clusters or grids. Clusters join computers in one location while grids go beyond one location and may span the Internet. They might scavenge otherwise unused cycles, or they might build or buy clusters for computationally demanding problems, or they might put existing computers into a grid. Recent software developments make it relatively easy for economists to create temporary or permanent clusters and grids. With changes in microprocessors and the introduction of clusters and grids, the computing environment for economists is poised for dramatic change.

There have been a fair number of papers in the field on parallel computing in economics; a partial list includes Gilli and Pauletto (1993), Nagurney et al. (1995), Nagurney (1996), Nagurney and Zhang (1998), Kontoghiorghes et al. (2000), Kontoghiorgies, ed (2000), Swann (2002), Doornik et al. (2002), Doornik et al. (2005), and Kontoghiorgies, ed (2005). Yet, it seems safe to say that parallel computing is not terribly widespread in the economics profession. Given recent and future technological changes that makes parallel computing both easier and more desirable, we hope to lead people to use the economic and econometric techniques in these papers more often. In short, this paper largely focuses on techniques that will enable economists to easily use parallel technologies in their research.

This paper is organized as follows. After a review of trends in computing performance and a look at the future of microprocessors, suggestions on how to make the most of a single CPU will be addressed (where performance gains might be most obtainable). Next is general material on programming more than one CPU. Following that is more detailed information on programming mutli-core processors and next is a section on programming clusters. The final section is devoted to grids.

2 Trends and Developments in Microprocessors and Networks

This section describes the reasons for rising interest in computing with more than one processor. But, before these factors are described, it is helpful to briefly describe com-

puter performance to put these changes into perspective. Unfortunately, the numerous benchmarks that attempt to measure computer performance all suffer from a variety of flaws; perhaps the most powerful criticism is that benchmarks are really only useful if they measure the sort of programs you use, and given the very wide variety of actual programs and very few benchmarks, few actually do. In addition, most benchmarks change with time so tracking changes in performance over the years is quite difficult. The sole widely-used and relatively stable one is the Linpack benchmark (Dongarra; TOP500 Supercomputer Sites) which solves a dense system of linear equations. It comes in three varieties: one solves a 100x100 system, another solves a 1000x1000 system, and the third is for large parallel machines. Table 1 shows performance for a small number of platforms in MFLOPS (millions of floating point operations per second). Unless otherwise noted, all are for one processor.[1]

Table 1: Linpack Performance in Megaflops

Computer	100x100	1000x1000	Parallel	Peak
IBM PC w/ 8087	.0069	-	N/A	-
Gateway 66 MHz 486	2.4	-	N/A	-
Cray Y-MP	161	324	N/A	333
Intel Pentium 4 (2.53 Ghz)	1,190	2,355	N/A	5,060
Virginia Tech Apple Cluster	-	-	12,250,000	20,240,000
Blue Gene/L DD2	-	-	70,720,000	91,750,000

The math coprocessor for the Intel 8086/8088, the 8087, was introduced in 1980, the first 486 came in 1989 (at less than at 66 Mhz), and the Cray Y-MP, a leading supercomputer of its era was introduced in 1988 (CPU World; Cray History). The Intel Pentium 4 is the current model (the current maximum clock speed is a bit less than 4 Ghz (Schmid and Roos)). The Virginia Tech Apple Cluster uses 1,100 Apple Xserve dual G5 servers connected with a high speed network. While the seventh fastest, it is remarkable for using commodity hardware and its low price per flop. The IBM Blue Gene/L DD2, with 2^{15} processors is the current Linpack parallel champ.

Today's leading desktop computer thus has an astonishing five orders of magnitude more power than the first widely-used desktop and nearly an order of magnitude more power than a leading supercomputer of less than two decades ago. Not surprisingly, the leading desktop's power is dwarfed by the current Linpack champ.

The increased speed of processors has come from two sources: a greater number of transistors and an increased clock speed[2]. More transistors means that a given operation can be done in fewer clock cycles, and of course a higher clock speed means that more operations can be done per unit time. The original 8088 had a clock speed of 5Mhz (CPU World) while the most recent Pentium 4 has a clock speed of nearly 4Ghz (Schmid and Roos). The 8086 had 29,000 transistors (CPU World) while the latest Pentium 4 has 169 million (Schmid and Roos). Given the nearly four orders of magnitude increase in clock frequency, it seems clear that transistor counts long ago reached diminishing returns. A side effect of higher clock speed is more waste heat generated by processors. The

latest Pentium produces as much as 115 watts from a chip of 135 square millimeters (a bit less than the typical thumb nail). The resulting power density is breathtaking: in a famous talk Grove (2002) (then chairman of Intel) pointed out that in approximately 1998 Pentium power density passed that of a hot plate, and at projected trends, they would pass a nuclear reactor in about 2005 and a rocket nozzle before 2010. Intel's chairman emeritus, Gordon Moore,[3] puts the situation this way: "It got absurd... I'm surprised that people are willing to sit still for 100-watt processors." (Clark)

The rise in waste heat has generated a dramatic turn in microprocessor design (Clark). To continue increasing performance, rather than increasing clock speed or adding more transistors (and their nearly diminished returns), both Intel and Advanced Micro Devices are shifting to producing "multi-core" CPUs. A single core is fully capable of independent operation; to the computer user, they are largely identical to adding another complete CPU to the machine.[4] In the Spring of 2005 both Intel and AMD introduced dual-core CPUs for desktop computers[5]. Currently, they can be fairly expensive, but the future is clear. By the end of 2006, Intel expects that fully 70% of desktop and mobile computers will ship with dual-core chips (Hachman). Intel's current president and CEO pointed out that in the past by increasing the clock speed and by other changes Intel increased the performance of his processors by a factor of three every four years. In the same time frame, multi-core and related technologies will increase performance by a factor of ten. He added, "This is very, very critical to understand where Intel is going." (Hachman)

Further in the future, Intel's former chief technology officer, Pat Gelsinger, predicts:

> As we go into the multicore era, it will be a period of great innovation, and this period will last the next 15 years. We will have tens of cores on a die, with each core capable of lots of threads by itself. So you are looking at a machine that is capable of the simultaneous execution of hundreds of threads on a single chip. (Strom and Gruener)

Or, to put it more starkly, Gelsinger warned Bill Gates (Clark):

> This thing is coming like a freight train, buddy.

Gelsinger's warning illustrates how hardware manufacturers are tossing a problem over the fence to those who write and use software. As described below, programming multi-core CPUs takes special techniques that are uncommon today. Further, they are not fully automated—anyone programming these chips will have to explicitly take account of the various cores. Running current software simply will not take advantage of them.

A taste of the future might be found with the recently introduced "Cell Processor" by IBM, Sony, and Toshiba (Becker). It will be in the upcoming Sony PlayStation 3 so one can assume that it will achieve volume production. Its nine processors have a maximum speed of 256 gigaflops. Unfortunately, there are reports that this is in single precision and its double precision performance is substantially less.

As processors have increased in power, the ability to connect computers together has become faster and easier as well. This plays an important roll for clusters and grids. Locally this is seen by increasing Ethernet and WiFi speeds. The first version of Ethernet became a standard in 1983 at a speed of 10 megabits/s, while in 2002 10 gigabit/s Ethernet was standardized (Ethernet Standards). While often not seen directly by users, the cost of connecting them over the Internet has fallen drastically due to extraordinary overcapacity in fiber optic lines resulting from the telecom crash of the late 1990s. Even more than 5 years later it is estimated that only 5% of fiber optic capacity is currently being used (Young.) While lines are no longer being laid, more and more signals can be sent over a given line at different wavelengths so unused capacity has remained constant since 2001.

3 How to Avoid Using More than One CPU

As will be seen below, there can be significant limitations to using more than one processor, be it in a CPU, a cluster, or a grid. In some cases the code might be difficult or impossible to parallelize while in others the code might be too fine-grained for the available hardware. In either case, the only way to get better performance might be to optimize your code. Rather than treating a computer like a black (or beige) box, time spent carefully writing key sections of code could yield very significant benefits. If you consider how long you might be waiting for results, time spent in optimizing code is likely to yield generous returns. Many of these concepts apply to most any computer language, yet they do not seem to be widely described.

The key concept to keep in mind is that processors perform best with a constant stream of instructions on contiguous data. With a constant stream of instructions and data a Pentium 4 can execute two floating point instructions per clock cycle, so at its maximum speed of nearly 4 Ghz this chip's theoretical maximum speed is almost 8 gigaflops (Hinton et al., 2001). AMD's Athlon and the PowerPC G5 used in Macintoshes max out at four flops per cycle (G5 Processor; AMD 3DNow! Technology FAQ). However, their clock speed is lower than the Pentium's and it appears that the AMD's maximum rate is for single precision operations. All of these use the respective processor's SIMD vector units[6].

All processors today have "caches" that store limited amounts of data and programs on the processor as accessing memory is much slower than the chip's execution units[7] Caches are filled from memory when execution units need data or instructions not in a cache. When memory is accessed after a "cache miss" nearby memory locations are also moved to the cache on the high probability that the next instruction or piece of data is close to the previously used one. If the next used memory location or next instruction is further away then the processor might have to wait until memory is accessed. As a result, jumps in the order of instructions may cause a program to execute more slowly. Jumps are typically caused by branches in execution logic, which in turn comes from constructs like loops, case statement, and if statements. Chip designers go to near-

heroic measures to avoid processor "stalls"[8] but their efforts are not always successful and it is wise to be aware of the costs of branches in instructions. A sense of this can be found in Table 1—for the 100x100 Linpack benchmark, the Pentium performs at 24% of its maximum and at 47% for the 1000x1000 benchmark[9]. Even with matrices that large, it is difficult to avoid branches and memory accesses and to keep the floating point units fully loaded.

Thus, the fastest code is likely to have the following characteristics:

- Use of branches, such as loops, if, and case statements, is minimized.
- Calls to subroutines are eliminated or "in-lined" with the appropriate linker options.
- Short loops might need to be "unrolled" (that is, each iteration is explicitly written). Note that some compilers will automatically unroll short loops.
- For nested loops, the long one should be the inner-most to avoid unnecessary branches and loop startups.
- Additions are frequently the quickest operation, then multiplications, and then divisions. Routines should be written with this in mind.
- Consider using professionally written low level libraries like BLAS (Basic Linear Algebra Subprograms) or the Intel Math Kernel Library (Intel Math Kernel Library) for your specific processor and environment.
- In general, use a language's high-level constructs or intrinsic functions rather than writing your own.

These suggestions are particularly important for the "kernel" of a routine (such as a likelihood function) as they are executed repeatedly. One can find program "hot spots" with "performance analyzers" to get a better handle on your code. These include Intel's "VTune" analyzer or MATLAB's M-file Profiler. For additional suggestions it is best to consult the documentation for your specific software. For example, the Intel Fortran compiler offers more than 100 well-organized pages on writing efficient code with this compiler for their processors, and MATLAB offers Improving Performance and Memory Usage . Efficient code generation is doubly important for modern processors' vector units (SSE3 on the latest Pentiums and AMD Athlons, 3DNow! in Athlons, and AltiVec in PowerPC chips used in Macs) as they have fairly exacting requirements. Sometimes they can handle little more than multiplying vectors by scalars, but of course they are very swift. Again, consulting the appropriate documentation is highly recommended. Also, most software packages seems to use these vector units, but it is sensible to check and perhaps base usage decisions upon them.

In addition, besides using the above suggestions for the kernel of a routine, be sure to use professionally written numerical software where appropriate. Below is a list of sites to check.

Netlib http://netlib.org/
likely the large on-line repository of numerical software

Guide to Available Mathematical Software (GAMS) http://gams.nist.gov/
set of menus to find software for a given task[10].

Mathtools.net http://www.mathtools.net/
similar to GAMS

Econometrics Software Laboratory Archive (ELSA) http://elsa.berkeley.edu/
general econometrics repository

GAUSS Resources http://gurukul.ucc.american.edu/econ/gaussres/GAUSSIDX.HTM
GAUSS repository

Statlib http://lib.stat.cmu.edu/
similar to Netlib

Of course, much of this software is built into packages like MATLAB, GAMS, and Gauss. With them, it is generally best to use their intrinsic routines.

4 Parallelizing Code and Related Concepts

First, it helps to define key terms. Related ones are grouped together.

multi-core A multi-core processor is one processor that contains two or more complete functional units. Intel and AMD announced dual-core CPUs for desktops in the Spring of 2005. A multi-core ship is a form of SMP.

SMP Symmetric multiprocessing is where where two or more processors have equal access to the same memory. The processors may or may not be on one chip.

distributed memory A group of computers that are connected via networks. Due to the latency involved, one CPU cannot share memory with another one. Clusters thus typically have distributed memory. Distributed memory machines are appropriate for "loosely coupled" problems (which do not require frequent communication between processors), while "tightly coupled" problems are more appropriate for SMP machines.

cluster A local group of computers networked to work as one. Variations include dedicated clusters and "Networks of Workstations" (NOW) that operate part time as a cluster. A NOW might be computers in labs or in the offices of a building. By their nature, their memory is distributed.

grid A group of computers that might span the world. In one strict definition, it is not centrally controlled, it uses open sources tools, and it delivers nontrivial amounts of services Foster (2002). Such a grid has software approaching an operating system in complexity. Many systems called grids fail on one or more of these points.

process A single sequence of instructions. Most programs written by end users execute as one process. (Thread)

thread Similar to a process, but they tend to be "lighter" in that they are easier to start and contain less information. Some operating systems allow processes with multiple threads. Thus, a single program runs as one process but may have more than one thread. (Thread)

hyper-threading An Intel technology that makes one processor appear as more than one to the user and operating system. Only one thread runs at a time, but the "state" of different threads is saved to rapidly switch between them. (Marr et al., 2002)

coarse grained Also known as "embarrassingly parallel" code. The key parts of such code is independent of each other so it is easy to parallelize. An example would be techniques using Monte Carlo analysis or maximizing a likelihood function.

fine grained Such code has a important interdependencies and is harder to parallelize.

latency In this paper, it refers to the time it takes for one processor to communicate with another one. The lower the latency, the more fine-grained the parallel code can be and still run efficiently. Latency in multi-core chips is measured in nanoseconds (10^{-9}) given clock speeds in gigahertz, in clusters using standard Ethernet latencies are on the order of a 100 microseconds, $100 \cdot 10^{-6}$ (van der Steen and Dongarra, 2004) and across the Internet an order of magnitude greater or more.

Single Instruction Single Data (SSID) A "traditional" computer that has one processor and one set of memory (van der Steen and Dongarra, 2004). They are programmed with long-established tools.

Single Instruction Multiple Data (SIMD) One instruction operates on multiple pieces of data. They are rare today but for "vector processors" which operate on vectors and scalars (van der Steen and Dongarra (2004)). The first generations of Crays were vector machines and modern microprocessors have vector units: SSE3 on Intel and AMD, 3DNow! on AMD, and AltiVec on the PowerPC G4 and G5 processors used in Macintoshes. These are often used for graphic operations, but can be used by numerical software. Vector processor do not literally operate on an entire vector in a clock cycle, but they use one instruction to do vector operations.

4.1 Whether or not to parallelize code

A parallel computer program is one that has portions that run more or less simultaneously on multiple CPUs. Compared to ordinary serial execution on a single CPU, when a program is parallelized, it can run much faster. Before deciding whether or not to write a parallel version of a program, it is important to compare the expected benefits and costs.

The prime benefit to parallelizing code is the potential speedup. A typical program will have portions that can be parallelized with more or less effort and portions of code that are inherently serial. For example, simulation of an i.i.d. random variable is easily parallelizable, since the draws do not depend upon one another. Simulation of a time series is inherently a serial operation, since to create a draw one needs to know the value of the previous draw. When a program is parallelized, some overhead is introduced since the portions of the code that are parallelized have to report back their results, and they have to be combined to give the same result as a serial program. This communications overhead may appear repeatedly as the program moves between blocks of parallelized and serial code.

The main factors that determine the potential speedup from parallelization are the relative proportions of parallelizable to inherently serial code and the size of the communications overhead. These factors are illustrated in Figure 1. If one imagines that the green blocks could be driven to zero execution time by using additional CPUs, the blue and yellow blocks determine the low limit to runtime. If the blue blocks are small relative to the green blocks, and if the yellow communications overhead blocks sum to little time, then parallelization will lead to a good improvement.

For some problems, parallelization can result in excellent speedups. Another factor to take into account is how often a program will be used. If it will be run many times, perhaps by many people, then time spent parallelizing the code will have a greater benefit.

Turning to costs, the main costs come from the additional logical complexity of parallel computer code. One must take into account the necessary synchronizations during parallel execution. This can result in many more lines of code. If the probability that a program contains a bug is proportional to the lines of code it contains, a parallel version of a program is more likely to contain bugs. Due to the greater logical complexity of the parallel version, it is likely to be more difficult to debug as well.

Comparing costs and benefits, one sees that not all programs are good candidates for parallelization. Many programs run fast enough on a single CPU, and some programs simply cannot be parallelized. But parallelization can make it feasible to tackle problems that are simply too computationally demanding to solve using a single computer. Problems that have too long of a runtime or which have memory requirements beyond what is available on a single computer can often be executed in parallel. This can extend the bound of what research projects are feasible to undertake at a given moment in time.

"Amdahl's Law," quantifies the the potential speedup from converting serial to parallel code (Sloan, 2004, p. 15)[11]. That is, parts of a program may be organized so that

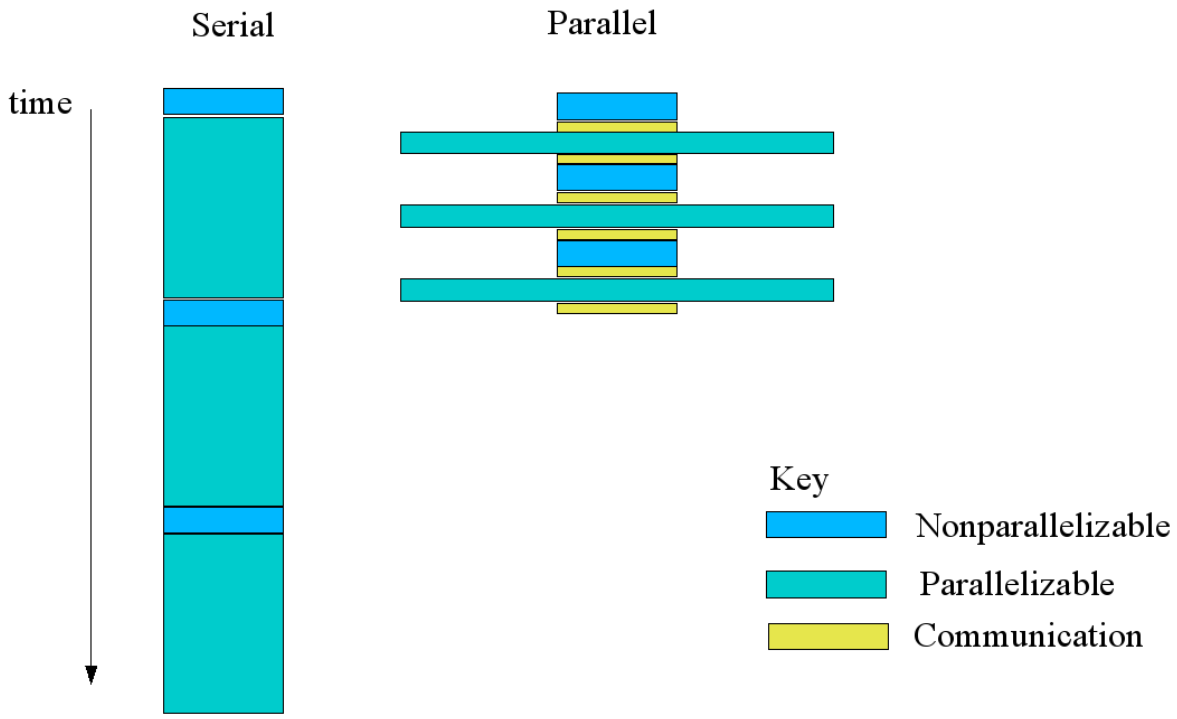

Figure 1: Serial and Parallel Runtimes

they can execute on multi-core CPUs, clusters, or grids. Let s be the fraction of the code that is inherently serial, p the fraction that can be parallelized (so $p + s = 1$), and N the number of processors. The speedup is then

$$\frac{1}{(s + (p/N))}$$

With an infinite number of processors, the maximum speedup one can achieve by parallelizing a program is the inverse of the fraction of the code that cannot be parallelized. Thus, to use more than one processor there is a very considerable premium on having code that is composed of parts that can be run concurrently and thus easily parallelized. While code with dependent segments can sometimes be parallelized, it takes more effort that might not be worth the expense. In addition, one can see that beyond some point adding additional processors yields a small return.

A subtle point to Amdahl's law is that as the problem size rises, p may rise and s fall. Consider a Monte Carlo estimation of an econometric test. Regardless of the number of replications, a portion of the code consists of setting up the data and that is inherently serial. But, as one increases the number of replications, p grows and s declines (Sloan, 2004, p. 16).

Further, Amdahl's Law is the best case. It does not include factors such as coordinating the processes. Clearly this will be smaller for two processors on a chip than computers spread across the world connected by the Internet.

4.2 How to parallelize code

Parallelization often requires communication between the concurrently running parts so that the parts. Different means of parallelizing code can be classified according to the communication scheme that is used.

4.2.1 Data parallelism

Data parallelism executes the same operations on different portions of a large data set. A Monte Carlo study is an example. The sequence of random numbers that are inputs can be thought of as the "data set." A given operation, such as the evaluation of an econometric estimation technique, is applied to all portions of the data set. This operation can be done simultaneously using a number of computers, each of which writes its results to a file. The communication overhead is reflected by reading each of these results and aggregating them. The SETI@home project is another example—in that project, desktop PCs download data from a server and process it during their idle time, then sending the results back to a server, which aggregates them. As seen below, Condor grid software can easily handle such problems.

4.2.2 Shared memory

A shared memory architecture (SMP) is one where two or more CPUs share the same memory. A dual core PC is an example. CPUs communicate using memory—one CPU store a value in a memory location that another uses. In some cases shared memory requires "locks" to ensure that memory is not corrupted. This generates an overhead that can affect the gains from parallelization. There are different ways to implement parallelization on shared memory systems. One method is to do so automatically using specialized compilers that can recognize parallelizable sections, but this is limited by the compiler's ability to recognize parallelism. A second step is to allow users to indicate which sections are to be parallelized, and to give instructions to the compiler how to do so. The OpenMP project is an example and is described below.

Unfortunately, programs written using OpenMP will not execute in parallel on distributed memory systems, which limits the portability of code that uses OpenMP. Another limitation is that shared memory systems that are currently available have a relatively modest number of processors, which limits the speedups that can be obtained.

4.2.3 Message passing

In message passing, different parallel threads coordinated their actions by sending messages passed back and forth. Message passing began with distributed systems where the latency is too great to share memory. It use is clearly mandated in distributed memory machines or clusters, and some types can also be used in SMP machines. Its use is described below.

5 Programming Multi-core CPUs and SMP Machines

As seen above, chip manufactures are rapidly moving to multi-core CPUs. Outside PCs, these machines are roughly synonymous with SMP (symmetric multi-processing) or shared memory machines where two or more CPUs share the same memory on an equal basis. While new to PCs, SMP machines have long been used in servers and data centers so tools for them are mature. In spite of this, their explicit use in economics research seems rather rare. This is likely because today most SMP machines are in a multi-user environments and most research seems to take place on the desktop. Further, in a multi-user environment running multiple threads per program is likely to slow overall system throughput slightly so it may be discouraged.

There are several obvious requirements to using an SMP machine. First, the operating system must support it (if not, only one CPU will be employed). Microsoft Windows XP Professional (but not Home Edition), Apple OS X, and Linux (with a suitably built kernel) all fall into this category. Next, your application must support SMP. Since it has a relatively long history, tools already exist for common languages like C/C++ and Fortran. Unfortunately, it does not appear that MATLAB currently supports threads, which

is required for the type of parallel programming used in this section[12]. However, the GAUSS Engine and some of their applications use threads, while the only GAMS solver that uses threads is a linear programming one. This section illustrates programming with Fortran and C/C++.

The easiest way to parallelize programs may be to just select the correct compiler option. Many current C/C++ and Fortran compilers can automatically generate parallel code for SMP machines.[13]

A common method for programming for explicit parallelization on SMP machines in C/C++ and Fortran is OpenMP ("open specifications for multi-processing") (OpenMP)[14] OpenMP members include IBM, Intel, Sun, and Hewlett-Packard. They describe it as "a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer" (OpenMP). Another benefit of OpenMP is its incremental nature—you can parallelize part of a program and then if need be another part. Fortran and C/C++ compilers supporting OpenMP are available from numerous vendors, including Lahey, HP, IBM, Intel, Portland Group, and Sun on various flavors of Unix and Windows.

This paper does not have sufficient room to describe every detail of OpenMP but its basics are illustrated. It should enable readers to write simple code and should make reading complete guides easier. It also makes the point that writing code for multiprocessors can be relatively straightforward. Two good tutorials are Barney (2005) and Hermanns (2002) while OpenMP Architecture Review Board (2005) is the official standard.

You access OpenMP in three ways: through compiler directives (they appear as comments to compilers not using OpenMP or when OpenMP is not invoked by OpenMP capable compilers with the appropriate compiler option), library routines (i.e. call to functions and subroutines), and environment variables. The interface, but not the syntax, is basically the same in C/C++ or Fortran. As the library routines are often not needed (and very easily found and removed given that all contain `omp_`), code parallelized with OpenMP can generally easily be run in serial mode if desired.

OpenMP uses a "fork and join" model. A program starts off serially with a "master" thread for all computations. Compiler directives then branch execution into a parallel section where code is executed concurrently with different threads as part of a "team" (of which the master is a member). Another directive then collapses the team back to the master thread. This process can be repeated. Memory can be shared between the threads, or it can be private to them, or there can be various combinations of private and shared memory. In the next sections on clusters and grids, memory cannot be shared between threads due to the much greater latency of connecting over networks than in a CPU or motherboard. Instead, "messages" are passed (typically with MPI, the "message passing interface") between threads to coordinate actions.

Listing 1 contains a very simple program that illustrates many of these ideas and Result 1 shows its output[15] First, note that it is standard Fortran. The USE command calls the OpenMP module. After the write command what appears to be a comment is

actually picked up by OpenMP compilers as a directive to parallelize a region until the paired closing directive. This is the most important directive—without it, code will not execute in parallel. The `!$comp` is a "sentinel" that informs OpenMP Fortran compilers that a directive for OpenMP follows. A non-OpenMP compiler will simply see this as a comment, as will OpenMP compilers when OpenMP is not invoked. If using fixed source form Fortran, the sentinel syntax is slightly different; they must start in column one and can be one of `!$omp`, `c$omp`, or `*$omp`. The sentinel for C/C++ is `#pragma omp`. The final notable part of the example is the the library routine `omp_get_thread_num()` which determines the current thread number.

```
1   PROGRAM p1
2   USE omp_lib
3
4   WRITE(*,'("Serial Region")')
5
6   !$omp parallel
7   WRITE(*,'("Thread = ", i1)'), omp_get_thread_num()
8   !$omp end parallel
9
10  WRITE(*,'("Serial Region")')
11
12  STOP
13  END
```

Listing 1: p1.f90

```
Serial Region
 Thread:  0
 Thread:  1
Serial Region
```

Result 1

The program results show that there were two threads in the parallel region. For this implementation of OpenMP, Intel's Fortran Compiler version 8.1, the default number of threads is the number of processors (it can be set to other values by an environment variable or a library routine). Thread 0 is the master and thread 1 is another member of the team. When the output is first examined it almost looks like a loop executed but remember it is each thread reporting its own output. Finally, thread 1 died when the parallel region ended.

To further describe OpenMP, it is useful to define various terms:

structured block A block of code with a single entry and single exit. Only structured blocks can be parallelized.

race condition This occurs if threads that are scheduled differently by the operating system or run on different speed processors generate different results. This can be avoided by synchronizing the threads, but this can be expensive. Race conditions can thus hinder parallelization.

barrier A place in the code where all threads must terminate before further code executes. It is often implicit at the end of a directive (this occurred in Listing 1 at the end of the parallel section).

thread safe A program that functions as intended even when it executes as separate threads.

Listing 2 shows the use of "private" and "shared" variables. The former occur in each thread individually while the latter are shared between threads (thus, "shared" memory). When threads are created, each private variable begins with undefined instances of its private variables. When threads finish they are again undefined (but with an option they can be saved for the following serial execution). With a shared variable the same memory location is used by all threads. As one might imagine, reading shared variables from memory involves fewer issues than writing them to memory. The latter is allowed, but the standard permits a delay between changing the value of a shared variable and other threads becoming aware of this change. If one wishes to make all threads aware of a change in a shared variable, the compiler directive `flush` must be used. If you wish to put its value in memory by different threads, the `reduction` or `atomic` directives must be used so that more than one thread is not updating it at a time.

The uses for private and shared variables are obvious—shared variables can be used for common inputs across threads while private ones can be used for unique results in a thread. There are additional options for setting all but a set of variables as shared or private as well as for more subtle use of shared and private variables.

```
1   PROGRAM p2
2   USE omp_lib
3
4   a = 2.0
5   !$omp parallel private(thd_num) shared(a)
6   num_thd = omp_get_thread_num()
7   WRITE(*,'("Thread = ", i1, " while a = ", f3.1)'), num_thd, a
8   !$omp end parallel
9
10  STOP
11  END
```

Listing 2: p2.f90

```
Thread = 1 while a = 2.0
Thread = 0 while a = 2.0
```

Result 2

The compiler directive `parallel` only allows identical code to be run in the different threads. OpenMP offers greater flexibility with "work-sharing" constructs that allow different operations in different threads. They must be inside parallel directives, which actually launch the threads. At the end of a work sharing region a barrier is assumed (that is, by default, all threads must finish before the next statement). There are four types of work sharing directives: for `do/for` loops, sections (of arbitrary code), for single threads, and workshare (for some Fortran array operations).

The `do` directive allows Fortran `do` loops and C/C++ `for` loops to be executed in parallel. Thus, the execution time of long loops can be reduced by spreading them across threads. Listing 3 contains an example.

```
1   PROGRAM p3
2   USE omp_lib
3
4   INTEGER, PARAMETER :: n = 10000 ! Set value of arrays
5   REAL, DIMENSION(n) :: a = 1, b = 2, c ! Initial values for 2 arrays
6
7   !$omp parallel shared(a, b, c) private (i)
8
9   !$omp do schedule (static)
10  do i = 1, n
11    c(i) = a(i) + b(i)
12  end do
13  !$omp end do
14
15  !$omp end parallel
16
17  STOP
18  END
```

Listing 3: p3.f90

After using Fortran's array syntax to set values to arrays `a` and `b`, parallel processing is then established. The arrays are shared so their values can be accessed by different threads if need be. The index for the loop is private so each thread will have its own value. The `do` directive before the `do` loop tells an OpenMP compiler to break up this loop into threads. The static option says to break up the loop into approximately equal sizes and assign one to each thread[16].

While seemingly straightforward, there are subtle points to parallelizing loops. Consider Listing 4 where the elements of a 10-element array with all values of 1 is first set up. Next, to illustrate how the number of threads can be set, a library routine is called (it must be before a parallel region). The loop that sums the elements of `a` is parallelized as before. Another work sharing construct, `single` is used (only the master thread is allowed to operate in it). It is used here as I/O operations are not thread-safe (one is mixing a serial operation, writing, with a parallel operation, and the result is unpre-

dictable). Finally, the sum of `a` is printed.

```
1   PROGRAM p4
2   USE omp_lib
3
4   INTEGER, PARAMETER :: n = 10 ! Set size of array
5   REAL, DIMENSION(n) :: a = 1 ! Set value of array a
6
7   CALL omp_set_num_threads(5) ! Set number of threads
8
9   !$omp parallel shared(a) private (i)
10
11  !$omp do schedule (static) ! Parallelize loop
12  do i = 2, n
13    a(i) = a(i) + a(i-1)
14  end do
15  !$omp end do
16
17  !$omp single ! Don't parallelize the write of a
18  WRITE(*,'(F6.2)'), a
19  !$omp end single
20
21  !$omp end parallel
22
23  WRITE(*,'("Sum of a: ", F8.2)'), a(n)
24
25  STOP
26  END
```

Listing 4: p4.f90

```
1.00 2.00 3.00 4.00 5.00
6.00 7.00 8.00 9.00 10.00
Sum of a:  10.00
```

Result 4a

```
1.00 2.00 3.00 4.00 5.00
6.00 7.00 2.00 3.00 4.00
Sum of a:  4.00
```

Result 4b

Results 4a and 4b (slightly reformatted) show the results of two consecutive runs of the program. Different runs produced different results! This is due to a dependency in the do loop: `a(i) = a(i) + a(i-1)`. As written, `a` is a shared variable, but there are no restrictions on how the threads might run. Thus, the previous value of `a(i-1)` could well be 1 and not the summed value. This race condition is a programming error

that the compiler will not catch and is an illustration of code that is not thread safe. However, it can be made thread safe with the do loop option `ordered` along with the directives `ordered` and `end ordered` before and after the problem line 13.

There are several other major directives not described above. `sections` denotes a region with two or more uses of `section`. Each section can operate concurrently. There are also various synchronization constructs that can be used to avoid problems like the above besides `ordered`: `master` (only the master thread may run), `critical` (only one thread at a time may run), `barrier` (all threads must finish before further processing), `atomic` (a load or store of one variable is done one at a time), and `flush` (all threads must write to memory).

Another method of communicating with OpenMP is with library routines; several were used above and the most useful ones are defined below. Following the description is the Fortran syntax (you may need `USE omp_lib`) and next is the C/C++ syntax (you may need to declare the file `omp.h`).

omp_set_num_threads Set the number of threads for the next parallel section (must be called from a serial section). It can be more or less than the number of processors.

```
subroutine omp_set_num_threads(num_threads) num_threads: integer
void omp_set_num_threads(int num_threads);
```

omp_get_num_threads Reports the number of threads in the parallel region in which it is called.

```
integer function omp_get_num_threads()
int omp_get_num_threads(void);
```

omp_get_max_threads Reports the maximum value possible from omp_get_num_threads.

```
integer function omp_get_max_threads()
int omp_get_max_threads(void);
```

omp_get_thread_num Reports the current thread number. The master is 0.

```
integer function omp_get_thread_num()
int omp_get_thread_num(void);
```

omp_get_num_procs Reports the number of available processors.

```
integer function omp_get_num_procs()
int omp_get_num_procs(void);
```

omp_in_parallel Reports if the code is currently using more than one thread.

```
logical function omp_in_parallel()
int omp_in_parallel(void);
```

Finally, OpenMP uses environment variables (the syntax for setting them various with the shell and operating system).

OMP_SCHEDULE Set the scheduling for loops. Only `static` was used above.

OMP_NUM_THREADS Sets the number of threads to use. The default is not defined by OpenMP and is implementation specific. An integer variable.

OMP_DYNAMIC Permits or denies changing the number of threads. Either TRUE or FALSE.

OMP_NESTED Permits or denies nesting parallel sections. Either TRUE or FALSE.

6 MPI and Clusters

6.1 MPI

Message passing has become a very common way to write portable code for parallel computing. In fact, one standard, MPI, is available for SMP machines as well as clusters, so economists might want to consider it for all their parallel processing needs[17]. The most widely used parallel libraries are the Parallel Virtual Machine (PVM) and various implementations of the MPI (Message Passing Interface) standard (*MPI-2: Extensions to the Message-passing Interface*, 1997) and (*LAM/MPI Parallel Computing*, 2005). MPI is now more widely used than PVM, and we focus on it. MPI is a mechanism for passing instructions and data between different computational processes. Code written with MPI will run on many platforms and is often optimized.

MPI has been implemented in a number of packages, including LAM/MPI (*LAM/MPI Parallel Computing*, 2005), and MPICH (Gropp et al., 1996). These packages provide libraries of C functions and Fortran subroutines, along with support programs to use them. To make direct use of the libraries, one must program in C/C++, or Fortan. Many higher level languages have bindings to the low level C or Fortan functions. For example, MATLAB, Octave, Ox, Python, and R all have extensions that allow use of MPI. Potential users may wish to check Doornik et al. (2005), `http://www.mathworks.com/products/distribtb/` and `http://www.ll.mit.edu/MatlabMPI/`.

The processes that communicate with one another are referred to as a "communicator." The default communicator that is created at initialization contains all processes, and has the name MPI_COMM_WORLD. Other communicators can be created if needed. Each process in a communicator is assigned a "rank," which identifies it and allows other processes to send to and receive from it. Some of the fundamental functions specified by MPI (we present the C interface) are

- `MPI_Init (&argc,&argv)` Initializes the MPI environment. Creates `MPI_COMM_WORLD`, for example.
- `MPI_Comm_size (comm,&size)` Used to determine how many processes are in a communicator
- `MPI_Comm_rank (comm,&rank)` Used by a process to determine its own rank. Functions can be written to do different things depending upon the value of rank.

- `MPI_Send (&buf,count,datatype,dest,tag,comm)` Used to send a message to a given rank
- `MPI_Recv (&buf,count,datatype,source,tag,comm,&status)` Used to receive a message from a given rank
- `MPI_Bcast (&buffer,count,datatype,root,comm)` Used to send a message from the root process (rank=0) to all the other processes in the communicator. It might send out data, for example.

To see how MPI can be used, Listing 5 gives an example of a code snippet for performing a Monte Carlo study using the GNU Octave language. In line 3, the Monte Carlo study is done serially by default, otherwise in line 5 the parallel implementation begins. `LAM_Init` is a high level Octave function that takes care of initialization details, such as calling `MPI_Init`. In line 10, we see another high level function that sends data and instructions to the various CPUs that the program run on. This function embeds lower level MPI functions. In line 19 we see the direct use of an MPI function to receive the results of the slaves' calculations.

```
1  <snip>
2   if !PARALLEL # ordinary serial version
3     for i = 1:reps output(i,:) = feval(f, f_args); endfor
4    else # parallel version
5     LAM_Init(nslaves);
6     # The command that the slave nodes will execute
7     cmd=['contrib = montecarlo_nodes(f, f_args, n_returns, nn); ',\
8         'MPI_Send(contrib,0,TAG,NEWORLD);'];
9     nn = floor(reps/(NSLAVES + 1)); # How many reps per slave? Rest is for master
10    NumCmds_Send({'f', 'f_args', 'n_returns','nn','cmd'}, \
11      {f, f_args, n_returns, nn, cmd}); # Send data to all nodes
12    # run command locally for last block (slaves are still busy)
13    n_master = reps - NSLAVES*nn; # how many to do?
14    contrib = montecarlo_nodes(f, f_args, n_returns, n_master);
15    output(reps - n_master + 1:reps,:) = contrib;
16    # collect slaves' results
17    contrib = zeros(nn,n_returns);
18    for i = 1:NSLAVES
19      MPI_Recv(contrib,i,TAG,NEWORLD);
20      startblock = i*nn - nn + 1;
21      endblock = i*nn;
22      output(startblock:endblock,:) = contrib;
23    endfor
24 <snip>
```

Listing 5: montecarlo.m

In the Listing 5, lines 7-8 contain a command that the slave nodes execute. The first part of this command calls the function `montecarlo_nodes`, which appears in Listing 6. The slaves evaluate this to define the variable `contrib`, which is sent back to the master node in line 8 of Listing 5.

This example is intended to show how data and instructions may be sent back and forth between nodes. It also shows how high level languages can create new functions that can simplify the use of MPI. The whole of MPI is much more rich than this simple example indicates. There are many excellent tutorials available that go into the details of the MPI specification. Useful links include `http://www.llnl.gov/computing/tutorials/mpi/`, `http://www.mpi-hd.mpg.de/personalhomes/stiff/MPI/` and `http://www.lam-mpi.org/tutorials/`.

```
1 # this is the block of the montecarlo loop that is executed on each slave
2 function contrib = montecarlo_nodes(f, f_args, n_returns, nn)
3   contrib = zeros(nn, n_returns);
4   for i = 1:nn
5     contrib(i,:) = feval(f, f_args);
6   endfor
7 endfunction
```

Listing 6: montecarlo_nodes.m

6.2 Building Clusters

A cluster is a group of computers connected by a network that work together to accomplish tasks. They are generally close together. Clusters are not necessarily used for parallel computing—load balancing clusters shift processes between nodes to keep an even load on them. This solution works well for database and web servers. Packages like openMosix (`http://openmosix.sourceforge.net/`) take care of distributing the load transparently to users.

Here, we deal with clusters for MPI-based parallel computing. Clusters for parallel computing require a high-speed, low-latency network in order to achieve high performance. An ordinary 10 MB/sec Ethernet is sufficient for many purposes, but demanding applications require specialized networking technologies, such as Myrinet (`http://www.myricom.com/myrinet/overview/`) or Infiniband (`http://infiniband.sourceforge.net/`). The CPUs used in a cluster also affect its performance. However, even the most advanced clusters such as the IBM Blue Gene/L, which is currently the fastest supercomputer on Earth, use CPUs that are not radically more powerful than those found in commodity desktop computers. The main differences are in the bus speeds that connect the CPU to memory, power consumption per CPU, and in the networking technology that connects the CPUs to one another. The Blue Gene/L cluster has a peak Linpack score of 9.2×10^7, which is $1.8 \cdot 10^4$ times that of a Pentium 4 2.53 Ghz processor. However, the Blue Gene/L cluster has $3.2 \cdot 10^4$ processors. So performance per processor is more than comparable.

Clusters of the sort used by most economists will consist of desktop or workstation class machines with various operating systems connected by Ethernets with speeds of 10-1000 MB/sec. The physical creation of a cluster is not very demanding at all, but the installation of software can be tedious, since versions usually need to be the same across all nodes in the cluster. Manual installation on each node is both time consuming and prone to mistakes. A different option is to create a system image on a single computer and install it on the other nodes. This is easiest to do if the nodes are homogeneous.

Once a cluster has been configured with communication between nodes, installation of an MPI package is next. A number of solutions are available, depending upon the operating system. Without attempting to be exhaustive, for Microsoft Windows there is WMPI-II (`http://www.criticalsoftware.com/hpc/`) and MP-MPICH (`http://www.lfbs.rwth-aachen.de/content/mp-mpich`), and for Apple OS X[18] and Linux, both MPICH and LAM/MPI can be used.

Setting up a dedicated cluster is relatively straightforward, but it requires special knowledge that will likely require hiring information technology personnel. Configuration and security issues are not trivial. If many users are to access the cluster, installation and maintenance of the packages they need is another source of work for IT personnel. If you do wish to tackle it yourself, one good guide is Sloan (2004).

Since many economists do not have the knowledge needed to build a dedicated cluster, nor the budget to hire support personnel, other solutions are needed. One possibility is described in Creel (2004). His ParallelKnoppix is a bootable CD-ROM that allows creation of a working Linux cluster on a network of IA-32 computers (Intel Pentium/Xeon or AMD Athlon/Duron) in minutes. Knoppix (`http://www.knoppix.org/`) is a Linux distribution that comes on one CD-ROM. Rather than installing itself on a hard disk, it only uses your RAM, yet is still quite usable. Thus, when you reboot, your machine returns to its original state with whatever operating system you had installed. Creel modified Knoppix to first load itself on your server and then onto the nodes of your temporary cluster via their network. The cluster can be made up of homogeneous or heterogeneous computers, and they need not have Linux installed. It is quite easy to use, and it allows software packages and personal files to be added, so that individual users can tailor the it to their needs.

ParallelKnoppix added packages for MPI and contains scripts that configure the nodes of the cluster almost automatically with Knoppix. The CD is distributed preconfigured to support from 2 to 201 nodes. It has been used to build clusters up to 50 nodes. Once the cluster is shut down, the computers return to their original state, with their hard disks and operating systems unaltered. Unfortunately, there are some limitations to ParallelKnoppix: it is very insecure, so it might best be used on a network not connected to the Internet. Further, while the hard disks on nodes are not used, an unscrupulous ParallelKnoppix user could read or manipulate them. Thus, it is likely best deployed in a closed computer lab, where the disks presumably do not contain sensitive information, and the machines are not otherwise in use. Condor, an alternative that uses currently working machines, can be set up to use MPI. It is described below in the Grid section. As one might guess, its greater flexibility comes at the cost of a more difficult

installation.

ParallelKnoppix is not suitable for creation of a permanent, multi-user cluster. For that task, other solutions such as the Rocks cluster distribution (`http://www.rocksclusters.org`) or OpenSSI (`http://openssi.org`) are appropriate. These solutions attempt to make installation and maintenance of a cluster as easy as possible, but nevertheless are considerable more complicated to use than is ParallelKnoppix. For quick and simple creation of a cluster, there is probably no other tool that is easier to use than ParallelKnoppix. The Bootable Cluster CD (`http://bccd.cs.uni.edu/`) is probably the next most easy to use solution, but it is both more complicated to set up and is more oriented to learning about clusters, whereas ParallelKnoppix is intended to make a cluster available for use with minimal time and effort.

7 Grids

7.1 Introduction

"Grid computing" has become quite a buzzword. Many vendors use the term indiscriminately and its meaning has become quite fluid in a move to push products. It is easy to see why—Foster (2003), one of its leading architects, makes the point that many things we consume today are "virtualized"—for example, the details of water and power production are hidden from us. But, the details of computer use are anything but hidden. As he puts it, "We should not accept a situation in which every home and business had to operate its own power plant, library, printing press and water reservoir. Why should we do so for computers?"[19] Grid computing aims to make computer use as virtual as many other products we use today.

Already, some grid applications have achieved some notoriety among the computing public. They include

SETI@home Users download a screensaver that processes signals from radio telescopes in the "Search for Extraterrestrial Intelligence (SETI)." By early June 2005, more than 2.3 million CPU years and $7 \cdot 10^{21}$ floating point operations had been performed.

Smallpox Research Grid This project was organized by the grid firm United Devices, IBM, and others. There is a renewed concern about smallpox as a bioterror weapon and this project was designed to test millions of possible drugs against several proteins on the virus. Like SETI@home, it uses volunteers spare cycles. Now complete, it took nearly 70,000 CPU years.

Great Internet Mersenne Prime Search This project looks for a specific type of prime number empirically. By May of 2004 some 482,000 CPU years had been devoted to the project, and in early June of 2005 it was operating at approximately 17 teraflops. The grid firm Entropia is involved in this project.

More details on these projects can be found at SETI@home (2005), Smallpox Research Grid (2005), and GIMPS (2005).

While interesting, these projects and the many like them do not hold many lessons for economists. They are limited grids as each is custom designed for one purpose. The problems are very loosely coupled—it may be days or weeks between downloads of new "work packets" by participants. Further, it is hard to imagine many economics problems that would capture the interest of so many member of the public or a large number of economists. Nonetheless, they clearly give a sense of what grids can do.

With these projects in mind it is helpful to better define a grid. Foster (2002) provides what might be the most used definition. A grid

1. coordinates resources that are not subject to centralized control
2. using standard, open, general-purpose protocols and interfaces
3. to deliver nontrivial qualities of service.

He elaborates on these points as follows: a grid coordinates resources (from desktops to high performance clusters) of different organizations. As part of a grid, they are willing to share, but only on their terms, so no central dictates are possible. Each party must be able to decide all details of their contribution. Thus, cluster management software, like Sun's Grid Engine, is not a grid as management is centralized. Flexibility also dictates open standards. A very close analogy are the open standards of much of the Internet—anyone can use the protocols for e-mail or the web and interoperate with others[20]. Finally, a grid must generate useful amounts of computer power for its users. Unlike the web, where information and files are exchanged, in a grid compute cycles are shared. Foster goes on to list some first generation grid software; the only non-commercial one he mentions is Condor (`http://www.cs.wisc.edu/condor/`).

A somewhat different set of important concepts for the grid is described in The Five Big Ideas Behind the Grid (2005). They are

Resource Sharing Currently, spare cycles generally go to waste while others have a need for cycles. In short, there is no market.

Secure Access Trading resources requires security. This consists of access policies, authentication, and authorization.

Resource Use Uses are found for excess resources.

The Death of Distance High speed networks and low latency are bringing networked resources closer together.

Open Standards If diverse resources are shared in the grid, they must by their nature use the same protocols; this is much easier if they are open to all.

These ideas come from a larger document, Understanding Grids (2005), that is more detailed than Foster (2003) or Foster (2002). Foster and Kesselman, eds (2003) goes beyond these and describes grid architecture in considerable depth. Grids consist of layers (much like networking protocols that range from physical connections like Ethernet to TCP/IP to HTTP). The bottom "fabric" layer consists of computers, networks, and storage, next are protocols for connecting them, then there are "collective services" that allocate and monitor resources, and finally there are user applications. Many grid protocols borrow from other technologies like "web services," so to some degree constructing grid technologies is not entirely novel. Nonetheless, the level of complexity of a grid seems to rival an operating system.

Grid standards are set by the Global Grid Forum (`http://www.ggf.org`); the actual standards are called the "Open Grid Grid Services Architecture" (OGSA) and many of them are coded in the "Globus Toolkit" (`http://www.globus.org`). Unfortunately, the Globus Toolkit is not designed for end users, but rather "It provides standard building blocks and tools for use by application developers and system integrators." (The Role of the Globus Toolkit in the Grid Ecosystem) A number of projects have used the toolkit for custom applications, but all appear to have relied on programmers to mold the toolkit into a set of applications for users. Firms such as Univa, United Devices, Platform, Entropia, and DataSynapse either have products or will soon introduce them, but to date none offers general purpose grid computing software that would be suitable for economists. Perhaps more tellingly, a search of the Microsoft web site yields precious little when one searches for "OGSA" or "Globus."

Despite these limitation, there is an impressive range of grids in development and early production use. Some notable ones include

DOE Science Grid This grid aims to underpin the U.S. Department of Energy's computational resources. It currently connects resources at Lawrence Berkeley National Laboratory, Argonne National Laboratory, National Energy Research Scientific Computing Center, Oak Ridge National Laboratory, and Pacific Northwest National Laboratory.

GriPhyN The Grid Physics Network "is developing Grid technologies for scientific and engineering projects that must collect and analyze distributed, petabyte-scale datasets."[21] Among the physics addressed is analyzing the massive amount of output of the Large Hardon Collider at CERN.

NVO The National Virtual Observatory aims to make it easier for astronomers to coordinate the terabytes of data at different wavelengths from across the sky.

BIRN This NIH lead consortium of some 40 entities is focused on "brain imaging of human neurological disorders and associated animal models."

GEON This grid project is for geoscientists and concentrates on "a more quantitative understanding of the 4-D evolution of the North American lithosphere."

Access Grid Rather than computing, this grid is focuses on "large-scale distributed meetings, collaborative work sessions, seminars, lectures, tutorials, and training."

While each of these projects is for production use, each also contains significant research in just how to construct a grid. Each uses the Globus Toolkit, but each appears to customize it for their own circumstances. Thus, none of these offers near term solutions to economist's computing needs.

7.2 Condor

Unlike software based on the Globus Toolkit, there is at least one piece of software that economists can use to form grids: Condor (`http://www.cs.wisc.edu/condor/`). While it does not have the full functionality that many in the gird field are aiming for, it does offer sufficient features for production use. Indeed, today there are more than 1,600 known Condor "pools" with nearly 60,000 computers. Condor is well tested; the first version went into production in 1986.

Condor's basic operation might best be described as follows:

> A user submits the job to Condor. Condor finds an available machine on the network and begins running the job on that machine. Condor has the capability to detect that a machine running a Condor job is no longer available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard). It can checkpoint the job and move (migrate) the jobs to a different machine which would otherwise be idle. Condor continues job on the new machine from precisely where it left off. (Condor Team, 2005, p. 2)

Condor's emphasis is on high throughput computing (HTC) and not high performance computing (HPC). The latter is centered around familiar supercomputers and high performance clusters. HTC is based on the observation that many researchers are more concerned with getting a large number of compute cycles than getting a large number in a short time span. A related idea is that not many years ago compute cycles were centered in a few large computers, while today most organizations have many more compute cycles, but they are locked up in numerous PCs and workstations. Condor unlocks those cycles, so it is partly a "cycle scavenging" system that finds unused cycles in an organization and puts them to use. Computer owners set up a "ClassAd" file that describes what type of jobs they will accept and under what conditions, while those submitting jobs set their own requirements and wishes (there is great flexibility in the ClassAd system). Condor then matches both sides and dynamically schedules the jobs.

Jobs run in various run time environments (the Condor term is "universe"). The most likely ones are standard, vanilla, and MPI. In the standard environment, users must relink their jobs (with the output of pretty much any compiler). The Condor scheduler can stop these jobs (checkpointing) when needed (such as when a user returns to

his computer) and may migrate them to another pool member. Regardless of where the job runs, all file access is handled on the machine that submitted the job via remote procedure calls. A vanilla job cannot be checkpointed, migrated, or use remote procedure calls. Such jobs are suited for programs that cannot be relinked (such as programs from commercial binaries). In Condor one can also set up dedicated clusters using MPI or PVM jobs (but, jobs cannot be migrated off of clusters). "Job clusters," distinct from clusters, can be set up for data parallel jobs. With them a user submits one job with different inputs (ideal for Monte Carlo jobs). Condor's DAGMan extends job clusters by setting up jobs where one's output in input for another. Finally, Condor pools can be grouped into "flocks" that might span the Internet.

There are some limitations to Condor. Many system calls are not allowed in the standard environment (as a practical matter, this effects few economists). In a Unix environment, jobs run as the user `nobody` so unscrupulous programs can fill up the `tmp` directory or read world-readable files. While it runs on just about all conceivable hardware and software, jobs must run on the hardware and software the code was written for. Further, given its Unix roots, the Windows version is "clipped"—it does not yet support checkpointing or remote procedure calls. Condor nodes also requires a fair number of open ports to communicate so it can be hard to set up flocks between organizations behind firewalls with uncooperative network administers. While a basic setup is fairly straightforward in Unix[22], the documentation is voluminous and there are many options for ClassAd. The Appendix illustrates the installation of Condor on a Linux system.

At its most basic, Condor is remarkably easy to use. Listing 7 shows a Fortran input file—a simple "hello world" program. Note that it is standard Fortran. Condor requires no changes in the source code.

```
1       PROGRAM hello
2
3       PRINT*, "Hello World!"
4
5       STOP
6       END
```

Listing 7: hello.f for Condor

The program is then compiled for Condor use with `condor_compile g77 -o hello.remote hello.f`. The program `condor_compile` takes as its argument a a regular compiler command (most any compiler can work with it). The executable output file is `hello.remote`.

Listing 8 shows the companion Condor command file.

```
1 # Condor command file for hello.f
2 executable    = hello.remote
3 output        = hello.out
4 error         = hello.err
5 log           = hello.log
6 queue
```

Listing 8: hello.cmd Condor

Comments begin with a single pound sign. The executable was created above and the other commands should be obvious. Note that the program cannot be run interactively, so you must configure input and output files. In practice, the `hello.cmd` would likely have ClassAd values to tailor where and how the program would run.

It is then submitted into Condor `condor_submit hello.cmd`. As one could expect, the file `hello.out` contains `Hello World!`. In addition, the log file contains considerable information on the run: nodes it ran on, resources used, and the like. The file `hello.err` was thankfully empty.

These are some of the key Condor commands.

condor_compile Precede this command with your regular compile command to generate a Condor executable. This command does not work for programs that do not make their object code available (such as commercial software that only executes as a binary).

condor_submit Submit a job to the Condor system.

condor_status Show the current status of the Condor pool.

condor_q Display information about jobs in the Condor queue.

condor_rm Remove a job from the queue.

condor_history Show a list of your jobs that have completed.

7.3 How to Employ Condor

It seems clear that many computational economists could profitably use Condor. It is relatively straightforward to set up and it is proven in its use at more than 1,600 sites (surely a good argument for decision makers who might be reticent to allow its use). One thought would be to use Condor to connect computationally inclined economists. Yet, as a group, we are likely to have have few spare cycles. Plus, it can be difficult to run it through campus firewalls. Perhaps a better option would be to encourage departments, colleges, campuses, and other organizations to install Condor if they have not already[23]. Computational economists could then use otherwise wasted local cycles. In a spot check of the University of Wisconsin Condor site (`http://pumori.cs.wisc.edu/condor-view-applet/` roughly half the months in the last year showed significant spare cycles in their system. This suggests that by using Condor many computational economists might find many more cycles. One would suspect that Condor would be an easy sell to administrators based on the increased research productivity at a relatively small cost. Finally, given the "clipped" version of Condor for Windows and the large use of Windows by computational economists, perhaps the SCE could fund a full Condor version for Windows.

8 Conclusion

This paper describes how the computing environment will soon be changing due to changes in microprocessor design and networks. For some years economists have been easily able to undertake more challenging research with rising levels of microprocessor performance. To get increased performance in the future they will have to employ hardware like multi-core processors, clusters, and grids and programming tools like OpenMP and MPI. Here we showed the basics of OpenMP and MPI and introduced setting up temporary clusters using ParallelKnoppix or setting up Condor for cycle scavenging or clusters. Condor can even be extended into rudimentary, but usable grids.

Appendix: Setting Up Condor

While the Condor Manual is exhaustive, it might be helpful to illustrate the key parts of installing Condor on a Linux system (of course, you will still want to read the manual). The steps are surprisingly easy, but most certainly it helps to have some system administration experience.

Condor install files can be found at `http://www.cs.wisc.edu/condor/downloads/`. In this example `cook.rfe.org` is the manager and Condor is installed in `/usr/local/condor`.

1. Create a `condor` group and user with no login rights (set no password).

2. Add the following to `/etc/profile`:
   ```
   export CONDOR_CONFIG=/usr/local/condor/etc/condor_config
   export PATH=$PATH:/usr/local/condor/bin/
   export PATH=$PATH:/usr/local/condor/sbin/
   ```

3. Configure `/etc/manpath.config` for the Condor man pages in `/usr/local/condor`.

4. Untar the Condor installation file and from the resulting directory execute
   ```
   ./condor\_configure --install --install-dir=/usr/local/condor \
   --local-dir=/home/condor --type=submit,execute,manager        \
   --owner=condor --central-manager=cook.rfe.org}
   ```
   The options for `install-dir` and `local-dir` were described above. The `type` option says that this Condor node has three functions: you can submit Condor jobs on it, it can run Condor jobs, and it can manage the Condor system. If you wish to just set up an execute node (likely the most common), only `execute` would be used. The final two options are consistent with what is described above for the owner and manager.

5. `/usr/local/condor/etc/condor_config` and `/home/condor/condor_config.local` contain the configuration files. You will spend more time on these than the above.

Notes

[1]For brevity and since the goal here is to show broad outlines in performance, compiler details are not given.

[2]In each "clock cycle" a processor can accomplish a simple task. As the clock speed increases (more cycles per second), the processor can do more in that second.

[3]As one might guess, he is the Moore behind Moore's Law which correctly predicted in 1965 that processor density would double every 18 to 24 months.

[4]Depending upon the implementation, a dual core CPU might access memory differently than a true two-CPU computer.

[5]Apple introduced its first dual-processor machine in June 2004

[6]Note that there are great architectural differences between these chips so their theoretical performance is a poor metric for purchase decisions.

[7]In fact, most have two levels of caches on-chip—a very fast one right next to the execution units ("level one cache") and a larger one that takes a few clock cycles to access ("level two cache").

[8]Intel's hyper-threading is one example—if a thread is stalled it executes another one.

[9]Part of the difference likely comes from the rules of the tests—for the 100x100 only compiler optimizations are permitted but for the 1000x1000 test one must simply correctly invert the matrix.

[10]There is no relation to General Algebraic Modeling System (GAMS)

[11]Amdahl's Law is often applied to the more general case of speeding up one part of a computation and its impact on overall computation time

[12]However, the next section describes MPI and there are extensions to MATLAB for MPI.

[13]While C/C++ in GCC supports automatic parallelization, their old g77 Fortran compiler does not and it does not appear that their new Fortran 95 compiler does either. Academics looking for a free and capable Fortran might wish to consider Intel's Fortran 95 which is free for non-commercial use.

[14]It turns out that MPI, discussed below, can also be used to program SMP machines.

[15]All programs in this section were compiled on a Dell PowerEdge 2650 (a dual Xeon machine) with hyper-threading turned off. Compilation was with Intel Fortran 8.1 and it was invoked with `fort -fpp -openmp -threads`.

[16]There are other options for dealing with loops where work is not evenly distributed.

[17]This decision might include their environment and what they would like to learn.

[18]The Xgrid software for Apple OS X is designed for data parallel problems.

[19]As an economist, it is interesting to read others talk about the desirability of splitting production and consumption.

[20]The opposite case is instant messaging—there is no single standard, but only competing, proprietary ones, and arguably instant messaging suffers for it

[21]A petabyte is 1,000 terabytes

[22]Goffe set up a 2-node system on Linux in 2 days of reading documentation and installing software; additional nodes would be very quick to set up.

[23]Indeed, many economists may find it already available at their institution.

References

AMD 3DNow! Technology FAQ, <http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_861_1028,00.html> 2005.

Barney, Blaise, "OpenMP," 2005. <http://www.llnl.gov/computing/tutorials/openMP/>.

Becker, David, "PlayStation 3 Chip Has Split Personality," *ZDNet*, February 7 2005. <http://news.zdnet.com/PlayStation+3+chip+has+split+personality/2100-1040_22-5566340.html?part=rss&tag=feed&subj=zdnn>.

Clark, Don, "New Chips Pose a Challenge To Software Makers," *Wall Street Journal*, April 14 2005.

Condor Team, "Condor Version 6.6.9 Manual," <http://www.cs.wisc.edu/condor/manual/v6.6.9/condor-V6_6_9-Manual.pdf> May 25 2005.

CPU World, <http://www.cpu-world.com/> 2005.

Cray History, <http://www.cray.com/about_cray/history.html> 2005.

Creel, Michael, "ParallelKnoppix - Rapid Deployment of a Linux Cluster for MPI Parallel Processing Using Non-Dedicated Computers," 2004. <http://econpapers.repec.org/paper/aubautbar/625.04.htm>.

Dongarra, Jack, "Linpack Benchmark," 2005. <http://performance.netlib.org/performance/html/linpack.data.col0.html>.

Doornik, Jurgen A., David F. Hendry, and Neil Shephard, "Parallel Computation in Econometrics: a Simplifed Approach," in Erricos J. Kontoghiorgies, ed., *Handbook on Parallel Computing and Statistics*, Marcel Dekker, 2005. <http://www.doornik.com/research.html>.

_ **, Neil Shephard, and David F. Hendry**, "Computationally-intensive Econometrics Using a Distributed Matrix-programming Language," *Philosophical Transactions of the Royal Society of London, Series A*, 2002, *360*, 1245–1266.

Ethernet Standards, <http://www.processor.com/articles/P2718/23p18/23p18chart.pdf?guid=> 2005.

Foster, Ian, "What is the Grid," <http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf> July 20 2002.

_ , "The Grid: Computing without Bounds," *Scientific American*, April 2003. <http://www.eweek.com/article2/0,1759,1737050,00.asp>.

_ **and Carl Kesselman, eds**, *The Grid 2: Blueprint for a New Computer Infrastructure*, Kluwer, <http://www.univa.com/grid/pdfs/Chapter04_ID.pdf>.

G5 Processor, <http://www.apple.com/g5processor/executioncore.html> 2005.

Gilli, Manfred and Giorgio Pauletto, "Econometric Model Simulation on Parallel Computers," *International Journal of Supercomputer Applications*, 1993, *7*, 254–264.

GIMPS, <http://mersenne.org/> 2005.

Gropp, William, Ewing Lusk, Nathan Doss, and Anthony Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, 1996, *22*, 789–828. <http://www-unix.mcs.anl.gov/mpi/mpich/>.

Grove, Andy, "Changing Vectors of Moore's Law," December 10 2002. <http://www.intel.com/pressroom/archive/speeches/grove_20021210.htm>.

Hachman, Mark, "Intel Firms Up Dual-Core Plans," *eWeek.com*, December 7 2004. <http://www.eweek.com/article2/0,1759,1737050,00.asp>.

_ , "Intel Execs Run Down Dual-Core Roadmap," *ExtremeTech*, March 1 2005. <http://www.extremetech.com/article2/0,1558,1771366,00.asp>.

Hermanns, Miguel, "Parallel Programming in Fortran 95 using OpenMP," 2002. <http://www.openmp.org/presentations/miguel/F95_OpenMPv1_v2.pdf>.

Hinton, Glenn, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, February 12 2001, *5* (1). <http://developer.intel.com/technology/itj/archive/2001.htm>.

Improving Performance and Memory Usage (MATLAB), <http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/matlab_prog.html> 2005.

Intel Math Kernel Library, <http://www.intel.com/software/products/mkl/features/vml.htm> 2005.

Kontoghiorghes, Erricos J., Anna Nagurney, and Berc Rustem, "Parallel Computing in Economics, Finance and Decision-making," *Parallel Computing*, 2000, *26*, 507–509.

Kontoghiorgies, Erricos. J., ed., *Parallel Algorithms for Linear Models: Numerical Methods and Estimation Problems*, Kluwer Academic Publishers, 2000.

_ **, ed.**, *Handbook of Parallel Computing and Statistics*, Marcel Dekker, 2005.
LAM/MPI Parallel Computing

LAM/MPI Parallel Computing, *<http://www.lam-mpi.org> 2005.*

Marr, Deborah T., Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton, *"Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, February 14 2002, 3 (1).* <http://developer.intel.com/technology/itj/archive/2002.htm>.
MPI-2: Extensions to the Message-passing Interface

MPI-2: Extensions to the Message-passing Interface, 1997. Message Passing Interface Forum.

Nagurney, Anna, "Parallel Computation," in "Handbook of Computational Economics / Handbooks in Economics, volume 13," Elsevier Science, North-Holland, 1996, pp. 335–404.

_ **and D. Zhang**, "A Massively Parallel Implementation of a Discrete-time Algorithm for the Computation of Dynamic Elastic Demand and Traffic Problems Modeled as Projected Dynamical Systems," *Journal of Economic Dynamics and Control*, 1998, *22*, 1467–1485.

_ **, T. Takayama, and D. Zhang**, "Massively Parallel Computation of Spatial Price Equilibrium Problems as Dynamical Systems," *Journal of Economic Dynamics and Control*, 1995, *19*.

OpenMP, <http://www.openmp.org/> 2005.

OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 2.5," 2005. <http://www.openmp.org/drupal/mp-documents/spec25.pdf>.

Schmid, Patrick and Achim Roos, "Prescott Reworked: The P4 600 Series and Extreme Edition 3.73 GHz," *Tom's Hardware Guide*, February 21 2005. <http://www.tomshardware.com/cpu/20050221/index.html>.

SETI@home, <http://seticlassic.ssl.berkeley.edu./totals.html> 2005.

Sloan, Joseph D., *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI*, first ed., O'Reilly Media, 2004.

Smallpox Research Grid, <http://www.grid.org/projects/smallpox/index.htm> 2005.

Strom, David and Wolfgang Gruener, "Pat Gelsinger: A Shot at Running Intel," *Tom's Hardware Guide*, May 6 2005. <http://www.tomshardware.com/business/20050506/index.html>.

Swann, Christopher A., "Maximum Likelihood Estimation Using Parallel Computing: An Introduction to MPI," *Computational Economics*, 2002, *19*, 145–178.

The Five Big Ideas Behind the Grid, <http://gridcafe.web.cern.ch/gridcafe/challenges/challanges.html> 2005.

The Role of the Globus Toolkit in the Grid Ecosystem, <http://www.globus.org/grid_software/role-of-gt.php> 2005.

Thread (computer science), <http://en.wikipedia.org/wiki/Thread_%28computer_science%29> 2005.

TOP500 Supercomputer Sites, <http://www.top500.org/> 2005.

Understanding Grids, <http://www.gridforum.org/ggf_grid_understand.htm> 2005.

van der Steen, Aad J. and Jack Dongarra, "Overview of Recent Supercomputers," 2004. <http://www.top500.org/ORSC/2004/overview.html>.

Young, Shawn, "Why the Glut In Fiber Lines Remains Huge," *Wall Street Journal*, May 12 2005. <http://online.wsj.com/article/0,,SB111584986236831034,00.html>.