Editor: **Diomidis Spinellis**
Athens University of Economics and Business, dds@aueb.gr

Faking It

Diomidis Spinellis

THIS COLUMN IS about a tool we no longer have: the continuous rise of CPU clock frequencies. We were enjoying this trend for decades, but in the past few years, progress stalled. CPUs are no longer getting faster because their makers can't handle the heat of faster-switching transistors. Furthermore, increasing the CPU's sophistication to execute our instructions more cleverly has hit the law of diminishing returns. Consequently, CPU manufacturers now package the constantly increasing number of transistors they can fit onto a chip into multiple cores—processing elements—and then ask us developers to put the cores to good use.

If you haven't tried it, let me assure you that writing parallel code is a fiendishly difficult task. First, you have to battle against your conscious thought processes, which are sequential. Whether we write a cookbook or an aircraft checklist, we think in steps that execute in sequence. (Sometimes forever, as instructed on my shampoo bottle: "lather, rinse, repeat.") Second, even if you succeed in devising a scheme to split a problem among many cores, you'll find that the total speedup is typically much less than you expected. This happens because, as Gene Amdahl stated, the total speedup is limited by the sequential fraction of your program, which typically includes many synchronization tasks. Finally, it's very easy to mess up the synchronization between the multiple elements. Race conditions, deadlocks, and livelocks are only some of the bugs that show up when you try to put multiple cores to work.

Post your comments online by visiting the column's blog:
www.spinellis.gr/tools

The Road Ahead

You'll get a chance to appreciate all these problems if you program for multiple cores the hard way, by explicitly using multiple threads or even a higher-level API, such as OpenMP. An alternative approach involves using a programming language that can easily exploit multiple cores. Functional programming languages have an edge here because the building blocks of the programs you write in don't step on each other's toes. If you're familiar with Java, you can try switching to Scala, which adds support for functional constructs to Java's framework. Or you might decide to go all the way and use a pure functional language, such as Haskell. This might make sense if you're starting from scratch and your system performs heavy data processing with little interaction. If your work is tied to existing APIs, try a functional language tied to a popular framework, like Clojure (JVM) or F# (.NET). And since you're switching to a new programming language, you might as well try Erlang, which was explicitly designed to support concurrency.

Yet, the choice between using a treacherous API and learning a new programming paradigm or language is hardly appetizing. Fortunately, there's a third way. It involves faking your application's multicore-handling dexterity by handing over this responsibility to other software. As with all conjuring tricks, it isn't always possible to pull this off, but when you can, the results are spectacular. With little effort (and some cash), you can achieve remarkable speedups.

Multiple Processes

At the highest level, it's easy to put multiple cores to work if your application serves Web requests or dishes out SQL statements. Web application servers divide their work into threads or processes and thereby exploit all processing cores at their disposal. The only thing you need to do is to make your application run within the application server's framework, such as Java EE. The other easy case involves having a commercial relational database management system handle your SQL queries. The query optimization engine of such systems often finds ways to split a query among all available cores—for instance, each `WHERE` clause filtering a table can be assigned to a separate core. Under this scenario, your main responsibility is to let the database do as much of your work as possible. Never write explicit procedural code for what you can express in SQL statements.

...continued on p. 95

0740-7459/11/$26.00 © 2011 IEEE

...continued from p. 96

Another high-level way to utilize multiple cores is to let the operating system do it for you by splitting your processing among independent processes. Your application might match the pipes-and-filters architecture, which has one process generating the output that the next one will consume (see "Working with Unix Tools," *IEEE Software*, vol. 22, no. 6, 2005, pp. 9–11). The pipeline syntax, popularized by the Unix shell, lets you painlessly join multiple processes together in a series, without having to think about allocating your work among the cores. Any modern operating system worth its salt will do this automatically for you. Let's say you want to change a file's compression format from bzip2 into gzip. By running the pipeline

```
bzip2 -dc file.bz2 | gzip -c >file.gz
```

the decompression program bzip2 will run concurrently with the compression program gzip. This puts my laptop's two-core CPU to work with a utilization of around 65 percent. In one instance, the 34 percent speedup over a sequential execution shaved off 41 seconds from the conversion of a 0.5-Gbyte file.

If a process will work sequentially through some discrete data chunks (say, files, lines, or other records) then you can easily split the work among multiple cores using GNU **parallel**. All you do is feed your data chunks into it, and it runs as many jobs as needed to keep your CPU 100 percent busy. For instance, if you want to create thumbnails from your camera pictures, you'd invoke the JPEG decompression and compression programs through the **parallel** command as follows:

```
ls *.jpg | parallel 'djpeg -scale 1/16 {} | cjpeg
>thumb/{}'
```

(You can also achieve this by using the more common Unix **xargs -P** command.) On my laptop, this shrank the time needed to process 266 photos from 1 minute to 36 seconds. If your input is in one large file, **parallel** can split it among cores as needed. As an example, I ran the following command, which parsed a 1.1-Gbyte Web server log to print the client's software, on an eight-core machine:

```
parallel --pipe print_user_agent <access_log
```

With parallel, the command executed in 22 seconds, versus the 1 minute 15 seconds of the serial version, or 3.4 times faster.

Unfortunately, we can't model all processing tasks as linear processes that can be run as pipelines or as independent processes in parallel. In many cases, the various steps have dependencies. For us developers, the most familiar case is the building of software: to link object files, you need to compile them first. Other examples include the production of digital media, such as books and films, and also many data analysis tasks. In such cases, you can profit by specifying the dependencies between the various files and the actions required to build one from the other as a **Makefile** that the Unix **make** tool can process. Modern versions of **make** accept a **-j** argument, which instructs the tool to run many jobs in parallel as long as their dependencies allow it.

And Multiple Threads

The **parallel** command works by applying a process on chunks of data. You can do the same trick within your application by employing the *map-reduce* and *filter-reduce* techniques. These allow you to apply a function on the elements of a container (such as a vector) to either change each one of them (*map*) or select some of them (*filter*). You can then also use another function to *reduce* the results into a single element. If the function is at least moderately expensive, this operation can be profitably split through application-level threads among multiple cores. (Google uses the same idea at a massively larger scale to index the Web.) As an example, if you were reimplementing PowerPoint, you could code the slide-sorter view by mapping each image into a thumbnail and then reducing all thumbnails into a single image. The QtConcurrent C++ library has functions that perform all the tasks I've described. The only thing you need to do is to use a container compatible with QtConcurrent and write the *map*, *filter*, or *reduce* functions.

Many common, parallelizable, heavy-processing tasks are available in libraries hand-tuned to exploit the capabilities of multicore CPUs. Processor vendors, including AMD and Intel, offer libraries with functions covering audio and video encoding and decoding; image, speech, and general signal processing; cryptography; compression; and rendering. More specialized libraries are also available—for example, the NAG Numerical Components Library for SMP and multicore provides parallelized implementations of numerical computing and statistical algorithms. If you can express your problem in terms of, say, matrix operations (that's the case for a surprising variety of problems), then your application's efficiency will benefit simply by linking it to the library.

As you can see, although writing genuine parallel code is difficult, faking it effectively can be quite easy.

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.