Parallel Computing: Ways to Cooperate

Paul S. Wang, Sofpower.com

May 27, 2023

The history of modern computing is a story of the pursuit of speed. And indeed the increase in processing speeds has been remarkable.

On July 16, 1969, the US launched Apollo 11 that carried astronauts Neil Armstrong, Buzz Aldrin, and Michael Collins to the moon. Armstrong and Aldrin became the first humans to set foot on the lunar surface on July 20, 1969. The onboard Apollo Guidance Computer (AGC) was a compact digital computer responsible for providing guidance, navigation, and control for the spacecraft during various mission phases, including the critical lunar descent and ascent. The AGC had a clock speed of 1.024 MHz (megahertz), which means it executed approximately 1 million instructions per second.

Fast forward to Mid 2023, modern personal computers, laptops, and smartphones typically have processors with clock speeds measured in gigahertz (GHz), which are thousands of times faster than the AGC. Contemporary CPUs can execute billions of instructions per second (GIPS). What a difference.

As you may have guessed, keep making a single CPU smaller and faster can't work forevver. And you are right. We have basically reached the limit of single CPU speeds.

So what to do to get the computer even faster? The idea is actually simple, "use more CPUs". It is like if a job is too much for one person, divide up the job and get more people to help.

For a computer, more people means more *independent processing elements* (PEs) in the computer hardware. A PE can be a CPU or a *core*. Dividing up the job means designing our programs (apps) to have tasks to assign to the available hardware. The approach is known as **parallel computing**.

The idea is simple to understand, but the devil has always been in the details. Because making multiple anything to work together is not easy.

Making hardware and software to support parallel computing can be tricky indeed.

This article will explain how that is done, including the methods, challenges, and solutions. All that knowledge is directly applicable to managing cooperating people who work together in parallel.

Sequential Processing

The ordinary view of a program is this:

1. The program starts to run.
2. It executes a sequence of instructions one by one in the given order.
3. The program stops and finishes.

So far, we have the same view for procedures, algorithms, and flow charts. Figure **??** shows the sequential process of pie baking.

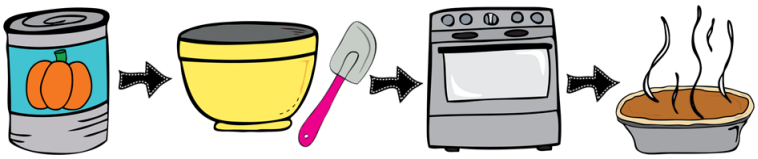

Figure 1: Pie Baking

Classic computer hardware is similarly sequential. It runs one program till it is done than proceed to the next program. It is a simple and straightforward approach. However strict sequential processing is seldom used in practice.

Even a PC with just one CPU can appear to do things in parallel by a technique known as *time slicing*—switching the CPU rapidly among processes (running apps) giving each process a slice of CPU time so they all appear to make progress and be responsive to user input.

Parallel Processing

A CPU in today's PCs usually has multiple *cores*. High end PCs may actually offer multiple CPUs each with multiple cores. They offer multiple PEs to

Figure 2: Traffic Cop in Rome

run different tasks at the same time, or in parallel. Figure **??** shows parallel activities in real life.

A core is part of the CPU and offers independent execution capabilities to the CPU allowing each core to run any *thread* belonging to any parallel app that are being run. Figure **??** shows a quadcore CPU where the cores are organized into two pairs sharing levels of cache memory.

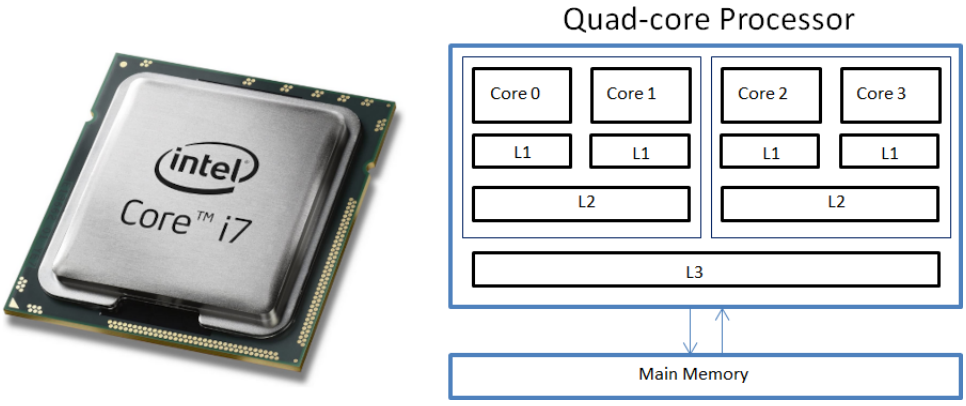

Figure 3: A Quadcore CPU

To To take advantage of multiple available PEs the operating system and application programs all need to written for that purpose. Thus, parallel software is also important for parallel computing. A parallel program breaks up tasks into parts to be run by the available PEs. The part of a program

that may run separately can be a processe (the whole app) or a thread (an independent part of an app).

The arrangement is like managing a busy restaurant—equip it with several kitchens (CPUs and cores) so that a number of chefs, cooks, and assistants (the various independent tasks) can work simultaneously to get the job done.

> **CT concept–*Parallel programming is advantageous***: *Multiple threads can make programs not only easier to write but also run faster and more responsively.*

Challenges for Parallel Programs

To support parallel processing, the hardware, operating system, and software programs (apps) all need to do their part. We'll now see how apps can utilize parallelism and the challenges thereof.

Parallel programming involves *threads*. By employing multiple threads to do different required tasks in parallel, a program can be faster and more efficient. We can think of each thread as a worker in the kitchen.

What Is a Thread?

A program under execution is called a *process*. A process consists of routines, data, stack, and operating system code and structures.

Within a process, control usually follows a single *execution thread*, starting with the first step, through a sequence of steps, and ending with the last step. This is the *single thread*

To perform tasks in parallel, multiple concurrent *threads of execution* (*multithreading*) can be used. As an independently running entity, a thread is much easier to create than a new process. Hence, a thread is sometimes known as a *lightweight process*.

Advantages of Multithreading

Single-threaded programs are good for simple calculations. Dynamic, interactive, or *event-driven* programs usually consist of multiple active parts that naturally perform independently and interact or cooperate in some way to achieve the intended goals. An event-driven program basically performs tasks

in reaction to external events such as user input or signals from other processes or threads. For example, consider satellite based navigation systems

Figure 4: Satellite Navigation in Car

(Figure **??**). Separate threads can take care of user control, map rendering/update, satellite signal tracking and location determination, and so on.

As another example, a video game program has independent parts for user controls, graphical rendering, motion generation, score keeping, etc. A single-threaded video game would be enormously complicated, if not impossible. A multithreaded program can model each of these parts with a different thread. Also, a Web browser is a natural candidate for multithreading. Figure **??** shows parallel processing in a chocolate assembly line.

Figure 5: A Chocolate Assembly Line

Furthermore, graphical rendering and user control processing take place simultaneously. Responsive handling of such concurrency is difficult in a single-threaded program.

Challenges of Multithreading

Basically, a multithreaded program has to coordinate several independent activities and avoid the possibility of them *tripping over one another*. Multithreaded programs involve four important new aspects not present in ordinary single-threaded programs: *mutual exclusion, synchronization, scheduling,* and *deadlock*. A good understanding of these concepts will help you better handle real-world activities and better manage cooperation with others.

Mutual Exclusion

Threads running in parallel usually need to cooperate to achieve intended tasks. Cooperation typically involves different threads accessing the same program constructs. When multiple threads share a common resource, a

Figure 6: Mutex

piece of data or a file, for example, simultaneous access by more than one thread can take place. Such simultaneous accesses is called a *race condition* and there is no telling which thread may win the race and the outcome of the entire program can be unpredictable or erroneous. For example, if two threads increase the same `counter` by one at the same time, the resulting

count can be wrong. To avoid simultaneous access, it is necessary to arrange *mutually exclusive access* to shared quantities. When programmed correctly, only one thread at a time can access the same quantity protected by *mutual exclusion* (mutex).

Consider writers cooperating on an magazine article. If two writers work on the same article from different workstations concurrently, disaster strikes. Mutual exclusion in this case can be arranged by insisting that every writer *lock* the article before working on it. No one else can obtain access to the article until it is *unlocked*.

Mutual exclusion is actually commonplace. Think about passengers on an airliner. Every passenger must lock the restroom door after entering it and unlock it to exit. Other passengers must wait until the door is unlocked before entering. That is simply politeness. In parallel computing, we call it *mutual exclusion* (Figure **??**).

Synchronization

Mutual exclusion avoids threads tripping over one another. But you still need a way for threads to communicate and coordinate their actions in order to cooperate. Threads make progress at independent and unpredictable rates. Thus, it is necessary to coordinate the order in which some tasks are performed.

If a task **must not be started** before some other tasks are finished, it is important to make sure that is the case. For example, imagine each thread is a worker in an assembly line . Then a thread must wait until another thread has finished a part it needs. Such time-related coordination of concurrent activities is called *synchronization*. For example, we all obviously know that synchronization is critical for symphony orchestras (Figure **??**).

Consider workers in a kitchen making bread. One is working on preparing the dough, another will take the dough that is ready to start the baking it. The baking worker must wait until the dough preparation has been done.

Thread synchronization usually involves delaying a thread until certain conditions are met or certain computations by other threads are done.

A thread is said to be *blocked* if its continued execution is delayed until a later time.

Figure 7: Synchronization

Thread Scheduling

When a process involves multiple threads, the available CPU and/or cores execute all threads in rapid succession. Exactly which currently executing thread is stopped and which waiting thread is run next Figure **??**) depend on the *scheduling policy* of the thread system and the priority settings of the threads involved.

Figure 8: Ready And Waiting

Deadlock

In a situation where multiple threads are interdependent in many ways, with resources shared under mutual exclusion and subtasks under synchroniza-

tion, there is the possibility of *deadlock*. Figure **??** shows a traffic deadlock situation. Deadlock happens when threads are waiting for events that will

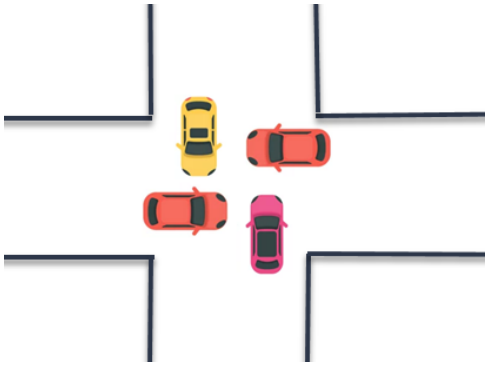

Figure 9: Deadlock

never happen. For example, thread `A` is waiting for data from thread `B` before producing output for `B`. `B` is waiting to receive some output from `A` before it can produce data for `A`. In real life, two friends wanted to talk but neither are willing to call first, a deadlock situation indeed. Of course, in parallel computing as well as in real life, we must avoid such problems.

> **CT concept–*Cooperating parallel programs need coordination***: *Otherwise they can run into problems and won't work well or at all.*

Finally

In parallel processing, we make use of multiple CPUs/cores and write multithreading programs for operating systems as well as important apps such as Web browsers. This way we can increase execution speed and perform tasks more quickly and effectively.

Parallel computing is an important area and we have only scratched the surface of that topic. Even then, we can see the links between parallel processing and everyday activities and understand, as computational thinkers should, that cooperation is a very good thing for doing big and complicated jobs by getting help from many. Yet, at the same time organizing a large cooperation we must deal with challenges such as mutual exclusion, synchronization, scheduling and deadlock.

Such understanding have lots of application in daily living and working.