Critical Paths

You should be familiar with the concept of a critical path from previous courses; it is the minimum amount of time to complete the task, taking dependencies into account, and without speculating.

Consider the following diagram, which illustrates dependencies between tasks (shown on the arrows). Note that B depends on A, and D depends on B and C, but C does not depend on anything, so it could be done in parallel with everything else. You can also compute expected execution times for different strategies.
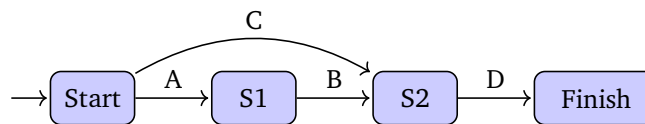
Data and Task Parallelism

There are two broad categories of paralellism: data parallelism and task parallelism. An analogy to data parallelism is hiring a call center to (incompetently) handle large volumes of support calls, *all in the same way*. Assembly lines are an analogy to task parallelism: each worker does a *different* thing.

More precisely, in data parallelism, multiple threads perform the *same* operation on separate data items. For instance, you have a big array and want to double all of the elements. Assign part of the array to each thread. Each thread does the same thing: double array elements.

In task parallelism, multiple threads perform *different* operations on separate data items. So you might have a thread that renders frames and a thread that compresses frames and combines them into a single movie file.

We'll continue by looking at a number of parallelization patterns, examples of how to apply them, and situations where they might apply.

Data Parallelism with SIMD

The "typical" boring standard uniprocessor is Single Instruction Single Data (SISD) but since the mid-1980s we've had more options than that. We'll talk about single-instruction multiple-data (SIMD) later on in this course, but here's a quick look. Each SIMD instruction operates on an entire vector of data. These instructions originated with supercomputers in the 70s. More recently, GPUs; the x86 SSE instructions; the SPARC VIS instructions; and the Power/PowerPC AltiVec instructions all implement SIMD.

Code.   Let's look at an application of SIMD instructions.

```
void vadd(double * restrict a, double * restrict b, int count) {
  for (int i = 0; i < count; i++)
    a[i] += b[i];
}
```

Compiling this without SIMD on a 32-bit x86 (`gcc -m32 -march=i386 -S`) might give this:

```
loop:
  fldl  (%edx)
  faddl (%ecx)
  fstpl (%edx)
  addl  8, %edx
  addl  8, %ecx
  addl  1, %esi
  cmp   %eax, %esi
  jle   loop
```

We can instead compile to SIMD instructions (`gcc -m32 -march=prescott -mfpmath=sse`) and get something like this:

```
loop:
  movupd (%edx),%xmm0
  movupd (%ecx),%xmm1
  addpd  %xmm1,%xmm0
  movpd  %xmm0,(%edx)
  addl   16,%edx
  addl   16,%ecx
  addl   2,%esi
  cmp    %eax,%esi
  jle    loop
```

The *packed* operations (p) operate on multiple data elements at a time (what kind of parallelism is this?) The implication is that the loop only needs to loop half as many times. Also, the instructions themselves are more efficient, because they're not stack-based x87 instructions.

SIMD is different from the other types of parallelization we're looking at, since there aren't multiple threads working at once. It is complementary to using threads, and good for cases where loops operate over vectors of data. These loops could also be parallelized; multicore chips can do both, achieving high throughput. SIMD instructions also work well on small data sets, where thread startup cost is too high, while registers are just there to use.

We'll see a case study that uses SIMD (also known as vector instructions) in Lecture 19. In [Lem18], Daniel Lemire argues that vector instructions are, in general, a more efficient way to parallelize code than threads. That is, when applicable, they use less overall CPU resources (cores and power) and can even run faster.

Parallelization using Threads or Processes

We'll be looking at thread-based or process-based parallelization for the next bit. We don't care about the distinction between threads and processes for the moment. In fact, we could even distribute work over multiple systems.

Pattern 1: Multiple Independent Tasks. If you're just trying to maximize system utilization, you can use one system to run a number of independent tasks; for instance, you can put both a web server and database on one machine. If the web server happens to be memory-bound while the database is I/O-bound, then both can use system resources. If the web server isn't talking to the database (rare these days!), then the tasks would not get in each others' way.

Most services probably ought to be run under virtualization these days, unless they're trivial or not mission-critical.

A more relevant example of multiple independent tasks occurs in cluster/grid/cloud computing: the cloud might run a number of independent tasks, and each node would run some of the tasks. The cloud can retry a task (on a different node, perhaps) if it fails on some node. Note that the performance ought to increase linearly with the number of threads, since there shouldn't be communication between the tasks.

Pattern 2: Multiple Loosely-Coupled Tasks. Some applications contain tasks which aren't quite independent (so there is some inter-task communication), but not much. In this case, the tasks may be different from each other. The communication might be from the tasks to a controller or status monitor; it would usually be asynchronous or be limited to exceptional situations.

Refactoring an application this way can help with latency: if you split off the CPU-intensive computations into a sub-thread, then the main thread can respond to user input more quickly.

Here's an example. Assume that an application needs to receive and forward network packets, and also must log packet activity to disk. Then the two tasks are clear: receive/forward, and log. Since logging to disk is a high-latency event, a single-threaded application might incur latency while writing to disk. Splitting into subtasks allows the receive/forward to run without waiting for previous packets to be logged, thus increasing the throughput of the system.

Pattern 3: Multiple Copies of the Same Task. A common variant of multiple independent tasks is multiple copies of the same task (presumably on different data). In this case, we'd require there to be no communication between the different copies, which would enable linear speedup. An example is a rendering application running on multiple distinct animations. We gain throughput, but need to wait just as long for each task to complete.

Pattern 4: Single Task, Multiple Threads. This is the classic vision of "parallelization": for instance, distribute array processing over multiple threads, and let each thread compute the results for a subset of the array.

This pattern, unlike many of the others before it, can actually decrease the time needed to complete a unit of work, since it gets multiple threads involved in doing the single unit simultaneously. The result is improved latency and therefore increased throughput. Communication can be a problem, if the data is not nicely array-structured, or has dependencies between different array parts.

Other names and variations for this pattern include "fork-join", where the main process forks its execution and gives work to all of the threads, with the join synchronizing threads and combining the results; and "divide-and-conquer", where a thread spawns subthreads to compute smaller and smaller parts of the solution.

Pattern 5: Pipeline of Tasks. We've seen pipelining in the context of computer architecture. It can also work for software. For instance, you can use pipelining for packet-handling software, where multiple threads, as above, might confound the order. If you use a three-stage pipeline, then you can have three packets in-flight at the same time, and you can improve throughput by a factor of 3 (given appropriate hardware). Latency would tend to remain the same or be worse (due to communication overhead).

Some notes and variations on the pipeline: 1) if a stage is particularly slow, then it can limit the performance of the entire pipeline, if all of the work has to go through that stage; and 2) you can duplicate pipeline stages, if you know that a particular stage is going to be the bottleneck.

Pattern 6: Client-Server. Botnets work this way (as does SETI@Home, etc). To execute some large computation, a server is ready to tell clients what to do. Clients ask the server for some work, and the server gives work to the clients, who report back the results. Note that the server doesn't need to know the identity of the clients for this to work.

A single-machine example is a GUI application where the server part does the backend, while the client part contains the user interface. One could imagine symbolic algebra software being designed that way. Window redraws are an obvious candidate for tasks to run on clients.

Note that the single server can arbitrate access to shared resources. For instance, the clients might all need to perform network access. The server can store all of the requests and send them out in an orderly fashion.

The client-server pattern enables different threads to share work which can somehow be parcelled up, potentially improving throughput. Typically, the parallelism is somewhere between single task, multiple threads and multiple loosely-coupled tasks. It's also a design pattern that's easy to reason about.

Pattern 7: Producer-Consumer.   The producer-consumer is a variant on the pipeline and client-server models. In this case, the producer generates work, and the consumer performs work. An example is a producer which generates rendered frames, and a consumer which orders these frames and writes them to disk. There can be any number of producers and consumers. This approach can improve throughput and also reduces design complexity.

Combining Strategies.   If one of the patterns suffices, then you're done. Otherwise, you may need to combine strategies. For instance, you might often start with a pipeline, and then use multiple threads in a particular pipeline stage to handle one piece of data. Or, as I alluded to earlier, you can replicate pipeline stages to handle different data items simultaneously.

Note also that you can get synergies between different patterns. For instance, consider a task which takes 100 seconds. First, you take 80 seconds and parallelize it 4 ways (so, 20 seconds). This reduces the runtime to 40 seconds. Then, you can take the serial 20 seconds and split it into two threads. This further reduces runtime to 30 seconds. You get a $2.5\times$ speedup from the first transformation and $1.3\times$ from the second, if you do it after the first. But, if you only did the second parallelization, you'd only get a $1.1\times$ speedup.

How to Parallelize Code

Here's a four-step outline of what you need to do.

1. Profile the code.

2. Find dependencies in hotspots. For each dependency chain in a hotspot, figure out if you can execute the chain as multiple parallel tasks or a loop over multiple parallel iterations. Think about changing the algorithm, if that would help.

3. Estimate benefits.

4. If they're not good enough (e.g. far from linear speedup), step back and see if you can parallelize something else up the call chain, or at a higher level of absraction. (Think Mandelbrot sets and computing different points in parallel).

Try to reduce the amount of synchronization that you have to do (waiting for parallel tasks to finish), because that always slows you down.

Thread Pools

We talked about "single task, multiple threads" above. The idea behind a *thread pool* is that it's relatively expensive to start a thread; it costs resources to keep the threads running at the operating system level; and the threads won't run optimally anyway, because they'll spend too much time swapping state in and out of the cache. Instead, you start an appropriate number of threads, which each grab work from a work queue, do the work, and report the results back. Web servers are a good application of thread pools[1].

A key question is: how many threads should you create? This depends on which resources your threads use; if you are writing computationally-intensive threads, then you probably want to have fewer threads than the number of virtual CPUs. You can also use Amdahl's Law to estimate the maximum useful number of threads, as discussed previously.

Here's a longer discussion of thread pools:

http://www.ibm.com/developerworks/library/j-jtp0730.html

---

[1]Apache does this: http://httpd.apache.org/docs/2.0/mod/worker.html. Also see an assignment where the students write thread-pooled web servers: http://www.cse.nd.edu/~dthain/courses/cse30341/spring2009/project4/project4.html.

Modern languages provide thread pools; Java's `java.util.concurrent.ThreadPoolExecutor` [Ora10], C#'s `System.Threading.ThreadPool` [Cor05], and GLib's `GThreadPool` [The15] all implement thread pools.

GLib. GLib is a C library developed by the GTK team. It provides many useful features that you might otherwise have to implement yourself in C.

References

[Cor05] Microsoft Corporation. How to Use a Thread Pool (C# Programming Guide), 2005. Online; accessed 15-November-2015. URL: `http://msdn.microsoft.com/en-us/library/3dasc8as%28v=vs.80%29.aspx`.

[Lem18] Daniel Lemire. Multicore versus SIMD instructions: the "fasta" case study, 2018. Online; accessed 03-January-2018. URL: `https://lemire.me/blog/2018/01/02/multicore-versus-simd-instructions-the-fasta-case-study/`.

[Ora10] Oracle. Class ThreadPoolExecutor, 2010. Online; accessed 15-November-2015. URL: `http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html`.

[The15] The GNOME Project. Thread pools, 2015. Online; accessed 15-November-2015. URL: `http://library.gnome.org/devel/glib/unstable/glib-Thread-Pools.html`.