

Making Programs Faster

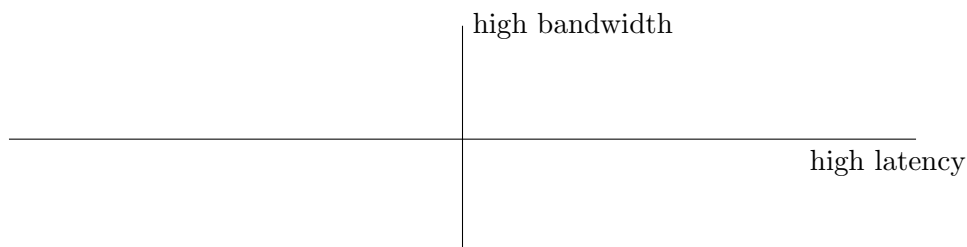
Let's start by defining what it means for programs to be fast. Basically, we have two concepts: items per unit time (bandwidth), and time per item (latency). Increasing either of these will make your program “faster” in some sense.

Items per unit time. This measures how much work can get done simultaneously; we refer to it as bandwidth. Parallelization—or doing many things at once—improves the number of items per unit time. We might measure items per time in terms of transactions per second or jobs per hour. You might still have to wait a long time to get the result of any particular job, even in a high-bandwidth situation; sending a truck full of hard drives across the continent is high-bandwidth but also high-latency.

Time per item. This measures how much time it takes to do any one particular task: we can call this the latency or response time. It doesn't tend to get measured as often as bandwidth, but it's especially important for tasks where people are involved. Google cares a lot about latency, which is why they provide the 8.8.8.8 DNS servers.

Examples. Say you need to make 100 paper airplanes. What's the fastest way of doing this?

Here's another example, containing various communications technologies:



We will focus on doing work, not on transmitting information, but the above example makes the difference between bandwidth and latency clear.

Improving latency: doing each thing faster

Although we'll be mostly focussing on parallelism in this course, a good way of writing faster code is by improving single-threaded performance. Unfortunately, there is often a limit to how much you can improve single-threaded performance; however, any improvements here may also help with the parallelized version. (On the other hand, faster sequential algorithms may not parallelize as well.) Here are some ways you can improve latency.

Profile the code. You can't successfully make your code faster if you don't know why it's slow. Intuition seems to often be wrong here, so run your program with realistic workloads under a profiling tool and figure out where all the time is going. In particular, make sure that you're not losing a lot of time doing non-critical tasks, e.g. those that you didn't really need to do.

This can be remarkably difficult. For instance, I've hacked on the Soot program analysis framework¹, which has a terrible startup time, but I haven't been able to improve it.

Do less work. A surefire way to be faster is to omit unnecessary work. Two (related) ways of omitting work are to avoid calculating intermediate results that you don't actually need; and computing results to only the accuracy that you need in the final output.

A hybrid between "do less work" and "be smarter" is caching, where you store the results of expensive, side-effect-free, operations (potentially I/O and computation) and reuse them as long as you know that they are still valid.

Be smarter. You can also use a better algorithm. (I suspect that this isn't that important in a lot of code, but I have no evidence to support that.) Better algorithms include better asymptotic performance as well as smarter data structures and smaller constant factors. Compiler optimizations (which we'll discuss in this course) help with getting smaller constant factors, as does being aware of the cache and data locality/density issues.

Sometimes you can find this type of improvements in your choice of libraries: you might use a more specialized library which does the task you need more quickly.

The build structure can also help, i.e. which parts of the code are in libraries and which are in the main executable. Gove's book discusses this issue in what I feel is exhaustive detail.

Improve the hardware. While CPUs are not getting much faster these days, your code might be I/O-bound, so you might be able to win by going to solid-state drives; or you might be swapping out to disk, which kills performance (add RAM). Profiling is key here.

On using assembly. This tends to be a bad idea these days. Compilers are going to be better at generating assembly than you are. It's important to understand what the compiler is doing, and why it can't optimize certain things (we'll discuss that), but you don't need to do it yourself.

¹<http://www.sable.mcgill.ca/soot>

Doing more things at a time

Rather than, (or in addition to), doing each thing faster, we can do more things at a time.

Why parallelism?

While it helps to do each thing faster, there are limits to how fast you can do each thing. The (rather flat) trend in recent CPU clock speeds illustrates this point. Often, it is easier to just throw more resources at the problem: use a bunch of CPUs at the same time. We will study how to effectively throw more resources at problems. In general, parallelism improves bandwidth, but not latency. Unfortunately, parallelism does complicate your life, as we'll see.

Different kinds of parallelism. Different problems are amenable to different sorts of parallelization. For instance, in a web server, we can easily parallelize simultaneous requests. On the other hand, it's hard to parallelize a linked list traversal. (Why?)

Pipelining. A key concept is pipelining. All modern CPUs do this, but you can do it in your code too. Think of an assembly line: you can split a task into a set of subtasks and execute these subtasks in parallel.

Hardware. To get parallelism, we need to have multiple instruction streams executing simultaneously. We can do this by increasing the number of CPUs: we can use multicore processors, SMP (symmetric multiprocessor) systems, or a cluster of machines. We get different communication latencies with each of these choices.

We can also use more exotic hardware, like graphics processing units (GPUs).

Difficulties with using parallelism

You may have noticed that it is easier to do a project when it's just you rather than being you and a team. The same applies to code. Here are some of the issues with parallel code.

First, some domains are “embarrassingly parallel” and these problems don't apply to them; for these domains, it's easy to communicate the problem to all of the processors and to get the answer back, and the processors don't need to talk to each other to compute. The canonical example is Monte Carlo integration, where each processor computes the contribution of a subrange of the integral.

I'll divide the remaining discussion into limitations and complications.

Limitations. Parallelization is no panacea, even without the complications that I describe below. Dependencies are the big problem.

First of all, a task can't start processing until it knows what it is supposed to process. Coordination overhead is an issue, and if the problem doesn't have a succinct description, parallelization can be difficult. Also, the task needs to combine its result with the other tasks.

“Inherently sequential” problems are an issue. In a sequential program, it’s OK if one loop iteration depends on the result of the previous iteration. However, such formulations prohibit parallelizing the loop. Sometimes we can find a parallelizable formulation of the loop, but sometimes we haven’t found one yet.

Finally, code often contains a sequential part and a parallelizable part. If the sequential part takes too long to execute, then executing the parallelizable part on even an infinite number of processors isn’t going to speed up the task as a whole. This is known as Amdahl’s Law, and we’ll talk about this soon.

Complications. It’s already quite difficult to make sure that sequential programs work right. Making sure that a parallel program works right is even more difficult.

The key complication is that there is no longer a total ordering between program events. Instead, you have a partial ordering: some events A are guaranteed to happen before other events B , but many events X and Y can occur in either the order XY or YX . This makes your code harder to understand, and complicates testing, because the ordering that you witness might not be the one causing the problem.

Two specific problems are data races and deadlocks.

- A *data race* occurs when two threads or processes both attempt to simultaneously access the same data, and at least one of the accesses is a write. This can lead to nonsensical intermediate states becoming visible to one of the participants. Avoiding data races requires coordination between the participants to ensure that intermediate states never become visible (typically using locks).
- A *deadlock* occurs when none of the threads or processes can make progress on the task because of a cycle in the resource requests. To avoid a deadlock, the programmer needs to enforce an ordering in the locks.

Another complication is stale data. Caches for multicore processors are particularly difficult to implement because they need to account for writes by other cores.

Limits to parallelization

I mentioned briefly in Lecture 1 that programs often have a sequential part and a parallel part. We'll quantify this observation today and discuss its consequences.

Amdahl's Law. One classic model of parallel execution is Amdahl's Law. In 1967, Gene Amdahl argued that improvements in processor design for single processors would be more effective than designing multi-processor systems. Here's the argument. Let's say that you are trying to run a task which has a serial part, taking time S , and a parallelizable part, taking time P , then the total amount of time T_s on a single-processor system is:

$$T_s = S + P.$$

Now, moving to a parallel system with N processors, the parallel time T_p is instead:

$$T_p = S + \frac{P}{N}.$$

As N increases, T_p is dominated by S , limiting the potential speedup.

We can restate this law in terms of speedup, which is generally the original time T_s divided by the sped-up time T_p :

$$S_p = \frac{T_s}{T_p}.$$

If we let f be the parallelizable fraction of the computation, i.e. set f to P/T_s , and we let S_f be the speedup we can achieve on f , then we get¹:

$$S_p(f, S_f) = \frac{1}{(1 - f) + \frac{f}{S_f}}.$$

Plugging in numbers. If $f = 1$, then we can indeed get good scaling, since $S_p = S_f$ in that case; running on an N -processor machine will give you a speedup of N . Unfortunately, usually $f < 1$. Let's see what happens.

f	$S_p(f, 18)$
1	18
0.99	~ 15
0.95	~ 10
0.5	~ 2

¹<http://www.cs.wisc.edu/multifacet/amdahl/>

To get close to a $2\times$ speedup for $f = 0.5$, you'd need to use around 18 cores².

Amdahl's Law tells you how many cores you can hope to leverage in your code, if you can estimate f . I'll leave this as an exercise to the reader: fix some f and set a tolerance, i.e. you don't care about a speedup less than x . Use the equations to figure out how many cores give that speedup.

The graph looks like this:

To get reasonable parallelization with a tolerance of 1.1, f needs to be at least 0.8.

Consequences of Amdahl's Law. For over 30 years, most performance gains did indeed come from increasing single-processor performance. The main reason that we're here today is that, as we saw in the video in Lecture 2, single-processor performance gains have hit the wall.

By the way, note that we didn't talk about the cost of synchronization between threads here. That can drag the performance down even more.

A more optimistic point of view

In 1988, John Gustafson pointed out³ that Amdahl's Law only applies to fixed-size problems, but that the point of computers is to deal with bigger and bigger problems.

In particular, you might vary the input size, or the grid resolution, number of timesteps, etc. When running the software, then, you might need to hold the running time constant, not the problem size: you're willing to wait, say, 10 hours for your task to finish, but not 500 hours. So you can change the question to: how big a problem can you run in 10 hours?

According to Gustafson, scaling up the problem tends to increase the amount of work in the parallel part of the code, while leaving the serial part alone. As long as the algorithm is linear, it is possible to handle linearly larger problems with a linearly larger number of processors.

Of course, Gustafson's Law works when there is some "problem-size" knob you can crank up. As a practical example, observe Google, which deals with huge datasets.

Modern Processors

For most of this lecture, I showed the video by Cliff Click on modern hardware:

<http://www.infoq.com/presentations/click-crash-course-modern-hardware>

²v2 corrected typo: $N = 18$ in Plugging in Numbers

³<http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html>

Concurrency and Parallelism

Concurrency and parallelism both give up the total ordering between instructions in a sequential program, for different purposes.

Concurrency. We'll refer to the use of threads for structuring programs as concurrency. Here, we're not aiming for increased performance. Instead, we're trying to write the program in a natural way. Concurrency makes sense as a model for distributed systems, or systems where multiple components interact, with no ordering between these components, like graphical user interfaces.

Parallelism. We're studying parallelism in this class, where we try to do multiple things at the same time in an attempt to increase throughput. Concurrent programs may be easier to parallelize.

Processor Design Issues

Recall that we listened to Cliff Click describe characteristics of modern processors in Lecture 2. In this lecture we'll continue our quick review of computer architecture and how it relates to programming for performance. Here's another reference about chip multi-threading; we are going to study some of the techniques in the “Writing Scalable Low-Level Code” section.

<http://queue.acm.org/detail.cfm?id=1095419>

Processes and Threads. Let's review the difference between a process and a thread. A *process* is an instance of a computer program that contains program code and its own address space, stack, registers, and resources (file handles, etc). A *thread* usually belongs to a process. The most important point is that it shares an address space with its parent process, hence variables and code as well as resources. Threads have their own stack, registers, and thread-specific data.

Threads and CPUs. In your operating systems class, you've seen implementations of threads (“lightweight processes”). We'll call these threads *software threads*, and we'll program with them throughout the class. Each software thread corresponds to a stream of instructions that the processor executes. On a old-school single-core, single-processor machine, the operating system multiplexes the CPU resources to execute multiple threads concurrently; however, only one thread runs at a time on the single CPU.

On the other hand, a modern chip contains a number of *hardware threads*, which correspond to the virtual CPUs. These are sometimes known as *strands*. The operating system still needs to multiplex the software threads onto the hardware threads, but now has more than one hardware thread to schedule work onto.

What's the term for swapping out the active thread on a CPU?

Implementing (or Simulating) Hardware Threads. There are a number of ways to implement multiple software threads; for instance, the simplest possible implementation, **kernel-level threading** (or 1:1 model) dedicates one core to each thread. The kernel schedules threads on different processors. (Note that kernel involvement will always be required to take advantage of a multicore system). This model is used by Win32, as well as POSIX threads for Windows and Linux. The 1:1 model allows concurrency and parallelism.

Alternately, we could make one core execute multiple threads, in the **user-level threading**, or N:1, model. The single core would keep multiple contexts and could 1) switch every 100 cycles; 2) switch every cycle; 3) fetch one instruction from each thread each cycle; or 4) switch every time the current thread hits a long-latency event (cache miss, etc.) This model allows for quick context switches, but does not leverage multiple processors. (Why would you use these?) The N:1 model is used by GNU Portable Threads.

Finally, it's possible to both use multiple cores and put multiple threads onto one core, in a **hybrid threading**, or M:N, model. Here, we map M application threads to N kernel threads. This is a compromise between the previous two models, which both allows quick context switches and the use of multiple processors. However, it requires increased complexity; the library provides scheduling services, which may not coordinate well with kernel, and increases likelihood of priority inversion (which you've seen in Operating Systems). This method is used by modern Windows threads.

Multicore Processors

As I've alluded to earlier, multicore processors came about because clock speeds just aren't going up anymore. We'll discuss technical details today.

Each processor *core* executes instructions; a processor with more than one core can therefore simultaneously execute multiple (unrelated) instructions.

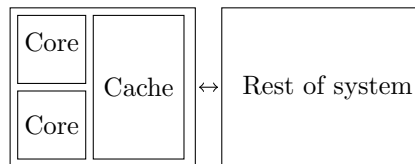
Chips and cores. Multiprocessor (usually SMP, or symmetric multiprocessor) systems have been around for a while. Such systems contain more than one CPU. We can count the number of CPUs by physically looking at the board; each CPU is a discrete physical thing.

Cores, on the other hand, are harder to count. In fact, they look just like distinct CPUs to the operating system:

```
plam@plym:~/courses/p4p/lectures$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Pentium(R) Dual-Core CPU           E6300  @ 2.80GHz
...
processor : 1
vendor_id : GenuineIntel
cpu family : 6
model : 23
model name : Pentium(R) Dual-Core CPU           E6300  @ 2.80GHz
```

If you actually opened my computer, though, you'd only find one chip. The chip is pretending to have two *virtual CPUs*, and the operating system can schedule work on each of these CPUs. In general, you can't look at the chip and figure out how many cores it contains.

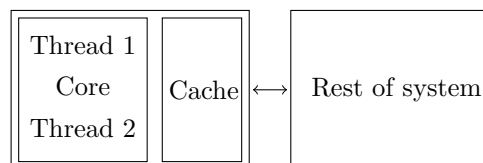
Hardware Designs for Multicores. In terms of the hardware design, cores might share a cache, as in this picture:



(credit: *Multicore Application Programming*, p. 5)

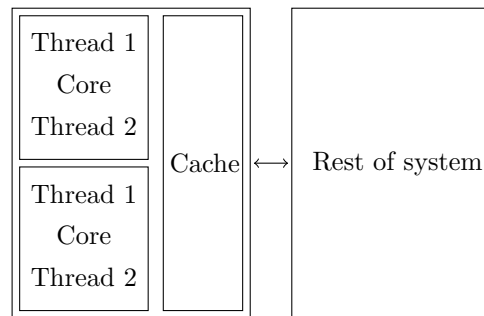
This above Symmetric Multithreading (SMP) design is especially good for the 1:1 threading model. In this case, the design of the cores don't need to change much, but they still need to communicate with each other and the rest of the system.

Or, we can have a design that works well for the N:1 model:



One would expect that executing two threads on one core might mean that each thread would run more slowly. It depends on the instruction mix. If the threads are trying to access the same resource, then each thread would run more slowly. If they're doing different things, there's potential for speedup.

Finally, it's possible to both use multiple cores and put multiple threads onto one core, as in the M:N model:



Here we have four hardware threads; pairs of threads share hardware resources. One example of a processor which supports chip multi-threading (CMT) is the UltraSPARC T2, which has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

Non-SMP systems. The designs we've seen above have been more or less SMP designs; all of the cores are mostly alike. A very non-Smp system is the Cell, which contains a PowerPC main core (the PPE) and 7 Synergistic Processing Elements (SPEs), which are small vector computers.

Non-Uniform Memory Access. In SMP systems, all CPUs have approximately the same access time for resources (subject to cache misses). There are also NUMA, or Non-Uniform Memory Access, systems out there. In that case, CPUs can access different resources at different speeds. (Resources goes beyond just memory).

In this case, the operating system should schedule tasks on CPUs which can access resources faster. Since memory is commonly the bottleneck, each CPU has its own memory bank.

Using CMT effectively. Typically, a CPU will expose its hardware threads using virtual CPUs. In current hardware designs, each of the hardware threads has the same performance.

However, performance varies depending on context. In the above example, two threads running on the same core will most probably run more slowly than two threads running on separate cores, since they'd contend for the same core's resources. Task switches between cores (or CPUs!) are also slow, as they may involve reloading caches.

Solaris "processor sets" enable that operating system to assign processes to specific virtual CPUs, while Linux's "affinity" keeps a process running on the same virtual CPU. Both of these features reduce the number of task switches, and processor sets can help reduce resource contention, along with Solaris's locality groups¹

¹Gove suggests that locality groups help reduce contention for core resources, but they seem to help more with memory.

Processes versus Threads

The first design decision that you need to solve when parallelizing programs is whether you should use threads or processes.

- Threads are basically light-weight processes which piggy-back on processes' address space.

Traditionally (pre-Linux 2.6) you had to use `fork` (for processes) and `clone` (for threads). But `clone` is not POSIX compliant, and its man page says that it's Linux-specific—FreeBSD uses `rfork()`. (POSIX is the standard for Unix-like operating systems).

When processes are better. `fork` is safer and more secure than threads.

1. Each process has its own virtual address space:
 - Memory pages are not copied, they are copy-on-write. Therefore, processes use less memory than you would expect.
2. Buffer overruns or other security holes do not expose other processes.
3. If a process crashes, the others can continue.

Example: In the Chrome browser, each tab is a separate process. Scott McCloud explained this: <http://uncivilsociety.org/2008/09/google-chrome-comic-by-scott-m.html>.

When threads are better. Threads are easier and faster.

1. Interprocess communication (IPC) is more complicated and slower than interthread communication; must use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library (or just memory reads and writes).
2. Processes have much higher startup, shutdown and synchronization costs than threads.
3. Pthreads fix the issues of `clone` and provide a uniform interface for most systems. (You'll work with them in Assignment 1.)

How to choose? If your application is like this:

- mostly independent tasks, with little or no communication;
- task startup and shutdown costs are negligible compared to overall runtime; and
- want to be safer against bugs and security holes,

then processes are the way to go. If it's the opposite of this, then use threads.

For performance reasons, along with ease and consistency across systems, we'll use threads, and Pthreads in particular.

Creating and Using Processes

Let's start with a simple usage example showing how to use the basic `fork()` system call.

```
pid = fork();
if (pid < 0) {
    fork_error_function();
} else if (pid == 0) {
    child_function();
} else {
    parent_function();
}
```

`fork` produces a second copy of the calling process; both run concurrently after the call. The only difference between the copies is the return value: the parent gets the pid of the child, while the child gets 0.

Overhead of Processes. The common wisdom is that processes are expensive, threads are cheap. Let's verify this with a benchmark on a laptop which creates and destroys 50,000 threads:

```
jon@riker examples master % time ./create_fork
0.18s user 4.14s system 34% cpu 12.484 total
jon@riker examples master % time ./create_pthread
0.73s user 1.29s system 107% cpu 1.887 total
```

Clearly Pthreads incur much lower overhead than `fork`. Pthreads offer a speedup of 6.5 over processes in terms of startup and teardown costs.

Using Threads to Program for Performance

We'll start by seeing how to use threads on "embarrassingly parallel problems":

- mostly-independent sub-problems (little synchronization); and
- strong locality (little communication).

Later, we'll see:

- which problems are amenable to parallelization (*dependencies*)
- alternative parallelization patterns
(right now, just use one thread per sub-problem)

About Pthreads. Pthreads stands for POSIX threads. It's available on most systems, including Pthreads Win32 (which I don't recommend). Use Linux, and our provided server, for this course.

Here's a quick `pthread`s refresher. To compile a C or C++ program with pthreads, add the `-pthread` parameter to the compiler commandline.

Starting a new thread. You can start a thread with the call `pthread_create()` as follows:

```
#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    printf("In main\n");
}
```

From the man page, here's how you use `pthread_create`:

```
int pthread_create(pthread_t* thread,
                  const pthread_attr_t* attr,
                  void* (*start_routine)(void*),
                  void* arg);
```

- **thread**: creates a handle to a thread at pointer location
- **attr**: thread attributes (NULL for defaults, more details later)
- **start_routine**: function to start execution
- **arg**: value to pass to start_routine

This function returns 0 on success and an error number otherwise (in which case the contents of `*thread` are undefined).

Waiting for Threads to Finish. If you want to join the threads of execution, use the `pthread_join` call. Let's improve our example.

```
#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
}
```

```

}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    printf("In main\n");
    pthread_join(thread, NULL);
}

```

The main thread now waits for the newly created thread to terminate before it terminates.

Here's the syntax for `pthread_join`:

```

int pthread_join(pthread_t thread,
                 void** retval)

```

- **thread**: wait for this thread to terminate (thread must be joinable).
- **retval**: stores exit status of thread (set by `pthread_exit`) to the location pointed by `*retval`. If cancelled, returns `PTHREAD_CANCELED`. `NULL` is ignored.

This function returns 0 on success, error number otherwise.

Caveat: Only call this one time per thread! Multiple calls to join on the same thread lead to undefined behaviour.

Inter-thread communication. Recall that the `pthread_create` call allows you to pass data to the new thread. Let's see how we might do that...

```

int i;
for (i = 0; i < 10; ++i)
    pthread_create(&thread[i], NULL, &run, (void*)&i);

```

Wrong! This is a *terrible* idea. Why?

1. The value of `i` will probably change before the thread executes
2. The memory for `i` may be out of scope, and therefore invalid by the time the thread executes

This is marginally acceptable:

```

int i;
for (i = 0; i < 10; ++i)
    pthread_create(&thread[i], NULL, &run, (void*)i);
...

void* run(void* arg) {
    int id = (int)arg;

```

It's not ideal, though.

- Beware size mismatches between arguments: you have no guarantee that a pointer is the same size as an int, so your data may overflow. (C only guarantees that the difference between two pointers is an int.)
- Sizes of data types change between systems. For maximum portability, just use pointers you got from `malloc`.

More on inter-thread synchronization. There was a comment on `pthread_join` only working if the target thread was joinable. Joinable threads (which is the default on Linux) wait for someone to call `pthread_join` before they release their resources (e.g. thread stacks). On the other hand, you can also create *detached* threads, which release resources when they terminate, without being joined.

```
int pthread_detach(pthread_t thread);
```

- **thread:** marks the thread as detached

This call returns 0 on success, error number otherwise.

Calling `pthread_detach` on an already detached thread results in undefined behaviour.

Finishing a thread. A thread finishes when its `start_routine` returns. But it's also possible to explicitly end a thread from within:

```
void pthread_exit(void *retval);
```

- **retval:** return value passed to function which called `pthread_join`

Alternately, returning from the thread's `start_routine` is equivalent to calling `pthread_exit`, and `start_routine`'s return value is passed back to the `pthread_join` caller.

Attributes. Beyond being detached/joinable, threads have additional attributes. (Note, also, that even though being joinable rather than detached is the default on Linux, it's not necessarily the default everywhere). Here's a list.

- Detached or joinable state
- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Scheduling contention scope
- Stack size
- Stack address
- Stack guard (overflow) size

Basically, you create and destroy attributes objects with `pthread_attr_init` and `pthread_attr_destroy` respectively. You can pass attributes objects to `pthread_create`. For instance,

```

size_t stacksize;
pthread_attr_t attributes;
pthread_attr_init(&attributes);
pthread_attr_getstacksize(&attributes, &stacksize);
printf("Stack size = %i\n", stacksize);
pthread_attr_destroy(&attributes);

```

Running this on a laptop produces:

```

jon@riker examples master % ./stack_size
Stack size = 8388608

```

Once you have a thread attribute object, you can set the thread state to joinable:

```

pthread_attr_setdetachstate(&attributes,
                           PTHREAD_CREATE_JOINABLE);

```

Warning about detached threads. Consider the following code.

```

#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    pthread_detach(thread);
    printf("In main\n");
}

```

When I run it, it just prints “In main”. Why?

Solution. Use `pthread_exit` to quit if you have any detached threads.

```

#include <pthread.h>
#include <stdio.h>

void* run(void*) {
    printf("In run\n");
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &run, NULL);
    pthread_detach(thread);
    printf("In main\n");
    pthread_exit(NULL); // This waits for all detached
                        // threads to terminate
}

```

Three useful Pthread calls. You may find these helpful.

```

pthread_t pthread_self(void);

int pthread_equal(pthread_t t1, pthread_t t2);

int pthread_once(pthread_once_t* once_control,
                 void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;

```


- `pthread_self` returns the handle of the currently running thread.
- `pthread_equal` compares 2 threads for equality.
- `pthread_once` runs a block of code just once, even across threads, based on a (presumably-global) control variable. This is useful for initialization code.

Threading Challenges.

- Be aware of scheduling (you can also set affinity with pthreads on Linux).
- Make sure the libraries you use are **thread-safe**:
 - Means that the library protects its shared data (we'll see how, below).
- glibc reentrant functions are also safe: a program can have more than one thread calling these functions concurrently. For example, in Assignment 1, we'll use `rand_r`, not `rand`.

Synchronization

You'll need some sort of synchronization to get sane results from multithreaded programs. We'll start by talking about how to use mutual exclusion in Pthreads.

Mutual Exclusion. Mutexes are the most basic type of synchronization. As a reminder:

- Only one thread can access code protected by a mutex at a time.
- All other threads must wait until the mutex is free before they can execute the protected code.

Here's an example of using mutexes:

```
pthread_mutex_t m1_static = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2_dynamic;

pthread_mutex_init(&m2_dynamic, NULL);
...
pthread_mutex_destroy(&m1_static);
pthread_mutex_destroy(&m2_dynamic);
```

You can initialize mutexes statically (as with `m1_static`) or dynamically (`m2_dynamic`). If you want to include attributes, you need to use the dynamic version.

Mutex Attributes. Both threads and mutexes use the notion of attributes. We won't talk about mutex attributes in any detail, but here are the three standard ones.

- **Protocol**: specifies the protocol used to prevent priority inversions for a mutex.
- **Prioceiling**: specifies the priority ceiling of a mutex.

- **Process-shared:** specifies the process sharing of a mutex.

You can specify a mutex as *process shared* so that you can access it between processes. In that case, you need to use shared memory and `mmap`, which we won't get into.

Mutex Example. Let's see how this looks in practice. It is fairly simple:

```
// code
pthread_mutex_lock(&m1);
// protected code
pthread_mutex_unlock(&m1);
// more code
```

- Everything within the `lock` and `unlock` is protected.
- Be careful to avoid deadlocks if you are using multiple mutexes (always acquire locks in the same order across threads).
- Another useful primitive is `pthread_mutex_trylock`. We may come back to this later.

Data Races

Why are we bothering with locks? Data races. A data race occurs when two concurrent actions access the same variable and at least one of them is a **write**. (This shows up on Assignment 1!)

```
...
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        ++counter;
    }
}

int main(int argc, char *argv[])
{
    // Create 8 threads
    // Join 8 threads
    printf("counter = %i\n", counter);
}
```

Is there a datarace in this example? If so, how would we fix it?

```
...
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int counter = 0;

void* run(void* arg) {
    for (int i = 0; i < 100; ++i) {
        pthread_mutex_lock(&mutex);
        ++counter;
        pthread_mutex_unlock(&mutex);
    }
}

int main(int argc, char *argv[])
```

```
{  
    // Create 8 threads  
    // Join 8 threads  
    pthread_mutex_destroy(&mutex);  
    printf("counter = %i\n", counter);  
}
```

The Compiler and You

Making the compiler work for you is critical to programming for performance. We'll therefore see some compiler implementation details in this class. Understanding these details will help you reason about how your code gets translated into machine code and thus executed.

Three Address Code. Compiler analyses are much easier to perform on simple expressions which have two operands and a result—hence three addresses—rather than full expression trees. Any good compiler will therefore convert a program's abstract syntax tree into an intermediate, portable, three-address code before going to a machine-specific backend.

Each statement represents one fundamental operation; we'll consider these operations to be atomic. A typical statement looks like this:

$$\text{result} := \text{operand}_1 \text{ operator } \text{operand}_2$$

Three-address code is useful for reasoning about data races. It is also easier to read than assembly, as it separates out memory reads and writes.

GIMPLE: gcc's three-address code. To see the GIMPLE representation of your code, pass gcc the `-fdump-tree-gimple` flag. You can also see all of the three address code generated by the compiler; use `-fdump-tree-all`. You'll probably just be interested in the optimized version.

I suggest using GIMPLE to reason about your code at a low level without having to read assembly.

The volatile qualifier

We'll continue by discussing C language features and how they affect the compiler. The `volatile` qualifier notifies the compiler that a variable may be changed by “external forces”. It therefore ensures that the compiled code does an actual read from a variable every time a read appears (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Here's an example.

```
int i = 0;
while (i != 255) { ... }
```

`volatile` prevents this from being optimized to:

```
int i = 0;

while (true) { ... }
```

Note that the variable will not actually be `volatile` in the critical section; most of the time, it only prevents useful optimizations. `volatile` is usually wrong unless there is a *very* good reason for it.

The restrict qualifier

The `restrict` qualifier on pointer `p` tells the compiler¹ that it may assume that, in the scope of `p`, the program will not use any other pointer `q` to access the data at `*p`.

The `restrict` qualifier is a feature introduced in C99: “The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables.”

- To request C99 in `gcc`, use the `-std=c99` flag.

`restrict` means: you are promising the compiler that the pointer will never alias (another pointer will not point to the same data) for the lifetime of the pointer. Hence, two pointers declared `restrict` must never point to the same data.

An example from Wikipedia:

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
    *ptrA += *val;
    *ptrB += *val;
}
```

Would declaring all these pointers as `restrict` generate better code?

Well, let’s look at the GIMPLE.

```
1 void updatePtrs(int* ptrA, int* ptrB, int* val) {
2   D.1609 = *ptrA;
3   D.1610 = *val;
4   D.1611 = D.1609 + D.1610;
5   *ptrA = D.1611;
6   D.1612 = *ptrB;
7   D.1610 = *val;
8   D.1613 = D.1612 + D.1610;
9   *ptrB = D.1613;
10 }
```

Now we can answer the question: “Could any operation be left out if all the pointers didn’t overlap?”

- If `ptrA` and `val` are not equal, you don’t have to reload the data on **line 6**.
- Otherwise, you would: there might be a call, somewhere:
`updatePtrs(&x, &y, &x);`

¹<http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html>

Hence, this set of annotations allows optimization:

```
void updatePtrs(int* restrict ptrA,
               int* restrict ptrB,
               int* restrict val)
```

Note: you can get the optimization by just declaring `ptrA` and `val` as `restrict`; `ptrB` isn't needed for this optimization

Summary of restrict. Use `restrict` whenever you know the pointer will not alias another pointer (also declared `restrict`).

It's hard for the compiler to infer pointer aliasing information; it's easier for you to specify it. If the compiler has this information, it can better optimize your code; in the body of a critical loop, that can result in better performance.

A caveat: don't lie to the compiler, or you will get undefined behaviour.

Aside: `restrict` is not the same as `const`. `const` data can still be changed through an alias.

Race Conditions

We'll next use our knowledge of three address code to analyze potential race conditions more rigourously.

Definition. A race occurs when you have two concurrent accesses to the same memory location, at least one of which is a **write**.

When there's a race, the final state may not be the same as running one access to completion and then the other. (But it sometimes is.) Race conditions typically arise between variables which are shared between threads.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void* run1(void* arg)
{
    int* x = (int*) arg;
    *x += 1;
}

void* run2(void* arg)
{
    int* x = (int*) arg;
    *x += 2;
}

int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
```

```

    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}

```

Question: Do we have a data race? Why or why not?

Example 2. Here's another example; keep the same thread definitions.

```

int main(int argc, char *argv[])
{
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}

```

Now do we have a data race? Why or why not?

Tracing our Example Data Race. What are the possible outputs? (Assume that initially `*x` is 1.) We'll look at the three-address code to tell.

<pre> 1 run1 2 D.1 = *x; 3 D.2 = D.1 + 1; 4 *x = D.2; </pre>	<pre> run2 D.1 = *x; D.2 = D.1 + 2 *x = D.2; </pre>
--	---

Memory reads and writes are key in data races.

Let's call the read and write from `run1` `R1` and `W1`; `R2` and `W2` from `run2`. Assuming a sane² memory model, R_n must precede W_n .

Here are all possible orderings:

Order				*x
R1	W1	R2	W2	4
R1	R2	W1	W2	3
R1	R2	W2	W1	2
R2	W2	R1	W1	4
R2	R1	W2	W1	2
R2	R1	W1	W2	3

Detecting Data Races Automatically

Dynamic and static tools exist. They can help you find data races in your program. `helgrind` is one such tool. It runs your program and analyzes it (and causes a large slowdown).

Run with `valgrind --tool=helgrind <prog>`.

It will warn you of possible data races along with locations. For useful debugging information, compile your program with debugging information (`-g` flag for `gcc`).

```
==5036== Possible data race during read of size 4 at
           0x53F2040 by thread #3
==5036== Locks held: none
==5036==    at 0x400710: run2 (in datarace.c:14)
...
==5036==
==5036== This conflicts with a previous write of size 4 by
           thread #2
==5036== Locks held: none
==5036==    at 0x400700: run1 (in datarace.c:8)
...
==5036==
==5036== Address 0x53F2040 is 0 bytes inside a block of size
           4 alloc'd
...
==5036==    by 0x4005AE: main (in datarace.c:19)
```

²sequentially consistent

More synchronization primitives

We'll proceed in order of complexity.

Recap: Mutexes. Recall that our goal in this course is to be able to use mutexes correctly. You should have seen how to implement them in an operating systems course. Here's how to use them.

- Call `lock` on mutex ℓ_1 . Upon return from `lock`, your thread has exclusive access to ℓ_1 until it `unlocks` it.
- Other calls to `lock` ℓ_1 will not return until `m1` is available.

For background on implementing mutual exclusion, see Lamport's bakery algorithm. Implementation details are not in scope for this course.

Key idea: locks protect resources; only one thread can hold a lock at a time. A second thread trying to obtain the lock (i.e. *contending* for the lock) has to wait, or *block*, until the first thread releases the lock. So only one thread has access to the protected resource at a time. The code between the lock acquisition and release is known as the *critical region*.

Some mutex implementations also provide a “try-lock” primitive, which grabs the lock if it's available, or returns control to the thread if it's not, thus enabling the thread to do something else.

Excessive use of locks can serialize programs. Consider two resources A and B protected by a single lock ℓ . Then a thread that's just interested in B still has to acquire ℓ , which requires it to wait for any other thread working with A . (The Linux kernel used to rely on a Big Kernel Lock protecting lots of resources in the 2.0 era, and Linux 2.2 improved performance on SMPs by cutting down on the use of the BKL.)

Note: in Windows, the term “mutex” refers to an inter-process communication mechanism. “Critical sections” are the mutexes we're talking about above.

Spinlocks. Spinlocks are a variant of mutexes, where the waiting thread repeatedly tries to acquire the lock instead of sleeping. Use spinlocks when you expect critical sections to finish quickly³. Spinning for a long time consumes lots of CPU resources. Many lock implementations use both sleeping and spinlocks: spin for a bit, then sleep for longer.

Reader/Writer Locks. Recall that data races only happen when one of the concurrent accesses is a write. So, if you have read-only (“immutable”) data, as often occurs in functional programs, you don't need to protect access to that data. For instance, your program might have an initialization phase, where you write some data, and then a query phase, where you use multiple threads to read the data.

Unfortunately, sometimes your data is not read-only. It might, for instance, be rarely updated. Locking the data every time would be inefficient. The answer is to instead use a *reader/writer* lock.

³For more information on spinlocks in the Linux kernel, see <http://lkm1.org/lkm1/2003/6/14/146>.

Multiple threads can hold the lock in read mode, but only one thread can hold the lock in write mode, and it will block until all the readers are done reading.

```
int readData(int c1, int c2) {                               // glib usage example
    g_static_rw_lock_reader_lock (&rwlock);
    int result = data[c1] + data[c2];
    g_static_rw_lock_reader_unlock (&rwlock);
}

void writeData(int c1, int c2, int value) {
    g_static_rw_lock_writer_lock (&rwlock);
    data[c1] += value; data[c2] -= value;
    g_static_rw_lock_writer_unlock (&rwlock);
}
```

Semaphores/condition variables. While semaphores can keep track of a counter and can implement mutexes, you should use them to support signalling between threads or processes.

In pthreads, semaphores can also be used for inter-process communication, while condition variables are like Java's `wait()/notify()`.

Barriers. This synchronization primitive allows you to make sure that a collection of threads all reach the barrier before finishing. In pthreads, each thread should call `pthread_barrier_wait()`, which will proceed when enough threads have reached the barrier. Enough means a number you specify upon barrier creation.

Lock-Free Code. We'll talk more about this in a few weeks. Modern CPUs support atomic operations, such as compare-and-swap, which enable experts to write lock-free code. A recent research result [McK11, AGH⁺11] states the requirements for correct implementations: basically, such implementations must contain certain synchronization constructs.

Semaphores

As you learned in previous courses, semaphores have a **value** and can be used for signalling between threads. When you create a semaphore, you specify an initial value for that semaphore. Here's how they work.

- The **value** can be understood to represent the number of resources available.
- A semaphore has two fundamental operations: **wait** and **post**.
- **wait** reserves one instance of the protected resource, if currently available—that is, if **value** is currently above 0. If **value** is 0, then **wait** suspends the thread until some other thread makes the resource available.
- **post** releases one instance of the protected resource, incrementing **value**.

Semaphore Usage. Here are the relevant API calls.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

This API is a lot like the mutex API:

- must link with `-pthread` (or `-lrt` on Solaris);
- all functions return 0 on success;
- same usage as mutexes in terms of passing pointers.

How could you use a `semaphore` as a `mutex`?

Semaphores for Signalling. Here's an example from the book. How would you make this always print "Thread 1" then "Thread 2" using semaphores?

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#include <stdlib.h>

void* p1 (void* arg) { printf("Thread 1\n"); }

void* p2 (void* arg) { printf("Thread 2\n"); }

int main(int argc, char *argv[])
{
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, p1, NULL);
    pthread_create(&thread[1], NULL, p2, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return EXIT_SUCCESS;
}
```

Proposed Solution. Is it actually correct?

```
sem_t sem;
void* p1 (void* arg) {
    printf("Thread 1\n");
    sem_post(&sem);
}
void* p2 (void* arg) {
    sem_wait(&sem);
    printf("Thread 2\n");
}

int main(int argc, char *argv[])
{
    pthread_t thread[2];
```

```

sem_init(&sem, 0, /* value: */ 1);
pthread_create(&thread[0], NULL, p1, NULL);
pthread_create(&thread[1], NULL, p2, NULL);
pthread_join(thread[0], NULL);
pthread_join(thread[1], NULL);
sem_destroy(&sem);
}

```

Well, let's reason through it.

- `value` is initially 1.
- Say `p2` hits its `sem_wait` first and succeeds.
- `value` is now 0 and `p2` prints “Thread 2” first.
- It would be OK if `p1` happened first. That would just increase `value` to 2.

Fix: set the initial `value` to 0. Then, if `p2` hits its `sem_wait` first, it will not print until `p1` posts, which is after `p1` prints “Thread 1”.

References

- [AGH⁺11] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.
- [McK11] Paul McKenney. Concurrent code and expensive instructions. Linux Weekly News, <http://lwn.net/Articles/423994/>, January 2011.

Dependencies

I've said that some computations appear to be “inherently sequential”. Here's why.

Main Idea. A *dependency* prevents parallelization when the computation XY produces a different result from the computation YX .

Loop- and Memory-Carried Dependencies. We distinguish between *loop-carried* and *memory-carried* dependencies. In a loop-carried dependency, an iteration depends on the result of the previous iteration. For instance, consider this code to compute whether a complex number $x_0 + iy_0$ belongs to the Mandelbrot set.

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
    int iterations = 0;
    double x = x0, y = y0, x2 = x*x, y2 = y*y;
    while ((x2+y2 < 4) && (iterations < 1000)) {
        y = 2*x*y + y0;
        x = x2 - y2 + x0;
        x2 = x*x; y2 = y*y;
        iterations++;
    }
    return iterations;
}
```

In this case, it's impossible to parallelize loop iterations, because each iteration *depends* on the (x, y) values calculated in the previous iteration. For any particular $x_0 + iy_0$, you have to run the loop iterations sequentially.

Note that you can parallelize the Mandelbrot set calculation by computing the result simultaneously over many points at once. Indeed, that is a classic “embarrassingly parallel” problem, because the you can compute the result for all of the points simultaneously, with no need to communicate.

On the other hand, a memory-carried dependency is one where the result of a computation *depends* on the order in which two memory accesses occur. For instance:

```
int val = 0;

void g() { val = 1; }
void h() { val = val + 2; }
```

What are the possible outcomes after executing `g()` and `h()` in parallel threads?

RAW, WAR, WAW and RAR

The most obvious case of a dependency is as follows:

```
int y = f(x);
int z = g(y);
```

This is a read-after-write (RAW), or “true” dependency: the first statement writes `y` and the second statement reads it. Other types of dependencies are:

	Read 2nd	Write 2nd
Read 1st	Read after read (RAR) No dependency	Write after read (WAR) Antidependency
Write 1st	Read after write (RAW) True dependency	Write after write (WAW) Output dependency

The no-dependency case (RAR) is clear. Declaring data immutable in your program is a good way to ensure no dependencies.

Let’s look at an antidependency (WAR) example.

```
void antiDependency(int z) {
    int y = f(x);
    x = z + 1;
}
```

```
void fixedAntiDependency(int z) {
    int x_copy = x;
    int y = f(x_copy);
    x = z + 1;
}
```

Why is there a problem?

Finally, WAWs can also inhibit parallelization:

```
void outputDependency(int x, int z) {
    y = x + 1;
    y = z + 1;
}
```

```
void fixedOutputDependency(int x, int z) {
    y_copy = x + 1;
    y = z + 1;
}
```

In both of these cases, renaming or copying data can eliminate the dependence and enable parallelization. Of course, copying data also takes time and uses cache, so it’s not free. One might change the output locations of both statements and then copy in the correct output.

Breaking Dependencies with Speculation

Recall that computer architects often use speculation to predict branch targets: the direction of the branch depends on the condition codes when executing the branch code. To get around having to wait, the processor speculatively executes one of the branch targets, and cleans up if it has to.

We can also use speculation at a coarser-grained level and speculatively parallelize code. We discuss two ways of doing so: one which we'll call speculative execution, the other value speculation.

Speculative Execution for Threads.

The idea here is to start up a thread to compute a result that you may or may not need. Consider the following code:

```
void doWork(int x, int y) {
    int value = longCalculation(x, y);
    if (value > threshold) {
        return value + secondLongCalculation(x, y);
    }
    else {
        return value;
    }
}
```

Without more information, you don't know whether you'll have to execute `secondLongCalculation` or not; it depends on the return value of `longCalculation`.

Fortunately, the arguments to `secondLongCalculation` do not depend on `longCalculation`, so we can call it at any point. Here's one way to speculatively thread the work:

```
void doWork(int x, int y) {
    thread_t t1, t2;
    point p(x,y);
    int v1, v2;
    thread_create(&t1, NULL, &longCalculation, &p);
    thread_create(&t2, NULL, &secondLongCalculation, &p);
    thread_join(t1, &v1);
    thread_join(t2, &v2);
    if (v1 > threshold) {
        return v1 + v2;
    } else {
```

```

    return v1;
}
}

```

We now execute both of the calculations in parallel and return the same result as before.

Intuitively: when is this code faster? When is it slower? How could you improve the use of threads?

We can model the above code by estimating the probability p that the second calculation needs to run, the time T_1 that it takes to run `longCalculation`, the time T_2 that it takes to run `secondLongCalculation`, and synchronization overhead S . Then the original code takes time

$$T = T_1 + pT_2,$$

while the speculative code takes time

$$T_s = \max(T_1, T_2) + S.$$

Exercise. Symbolically compute when it's profitable to do the speculation as shown above. There are two cases: $T_1 > T_2$ and $T_1 < T_2$. (You can ignore $T_1 = T_2$.)

Value Speculation

The other kind of speculation is value speculation. In this case, there is a (true) dependency between the result of a computation and its successor:

```

void doWork(int x, int y) {
    int value = longCalculation(x, y);
    return secondLongCalculation(value);
}

```

If the result of `value` is predictable, then we can speculatively execute `secondLongCalculation` based on the predicted value. (Most values in programs are indeed predictable).

```

void doWork(int x, int y) {
    thread_t t1, t2;
    point p(x,y);
    int v1, v2, last_value;
    thread_create(&t1, NULL, &longCalculation, &p);
    thread_create(&t2, NULL, &secondLongCalculation,
                  &last_value);
    thread_join(t1, &v1);
    thread_join(t2, &v2);
    if (v1 == last_value) {

```



```

    return v2;
} else {
    last_value = v1;
    return secondLongCalculation(v1);
}
}

```

Note that this is somewhat similar to memoization, except with parallelization thrown in. In this case, the original running time is

$$T = T_1 + T_2,$$

while the speculatively parallelized code takes time

$$T_s = \max(T_1, T_2) + S + pT_2,$$

where S and p are the same as above.

Exercise. Do the same computation as for speculative execution.

When can we speculate?

Speculation isn't always safe. We need the following conditions:

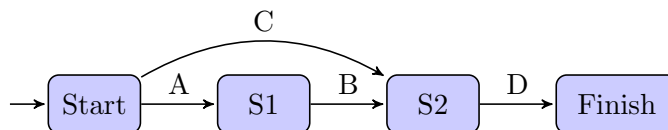
- `longCalculation` and `secondLongCalculation` must not call each other.
- `secondLongCalculation` must not depend on any values set or modified by `longCalculation`.
- The return value of `longCalculation` must be deterministic.

As a general warning: Consider the *side effects* of function calls.

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 parallelism: 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

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];
}
```

If we compile this without SIMD instructions on an x86, we might get 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 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.

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 abstraction. (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” last week. 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 pools¹.

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>

¹Apache 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`², C#'s `System.Threading.ThreadPool`³, and GLib's `GThreadPool`⁴ 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. Consider using GLib's thread pool in your assignment 2, unless you want to implement the work queue yourself. (I see no reason to do that!)

Automatic Parallelization

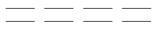
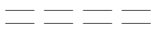
We'll now talk about automatic parallelization. The vision is to take your standard sequential C program and convert it into a parallel C program which leverages multiple cores, CPUs, machines, etc. This was an active area of research in the 1990s, then tapered off in the 2000s (because it's a hard problem!); it is enjoying renewed interest now (but it's still hard!)

What can we parallelize? The easiest kind of program to parallelize is the classic Fortran program which performs a computation over a huge array. C code—if it's the right kind—is a bit worse, but still tractable, given enough hints to the compiler. For us, the right kind of code is going to be array codes. Some production compilers, like the non-free Intel C compiler `icc`, the free-as-in-beer Solaris Studio compiler⁵, and the free GNU C compiler `gcc`, include support for parallelization, with different maturity levels.

Following Gove, we'll parallelize the following code:

```
1 #include <stdlib.h>
2
3 void setup(double *vector, int length) {
4     int i;
5     for (i = 0; i < length; i++)
6     {
7         vector[i] += 1.0;
8     }
9 }
10
11 int main()
12 {
13     double *vector;
14     vector = (double*) malloc (sizeof (double) * 1024 * 1024);
15     for (int i = 0; i < 1000; i++)
16     {
17         setup (vector, 1024*1024);
18     }
19 }
```

Manual Parallelization. Let's first think about how we could manually parallelize this code.

- **Option 1:** horizontal, 
Create 4 threads; each thread does 1000 iterations on its own sub-array.
- **Option 2:** bad horizontal, 
1000 times, create 4 threads which each operate once on the sub-array.

²<http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ThreadPoolExecutor.html>

³<http://msdn.microsoft.com/en-us/library/3dasc8as%28v=vs.80%29.aspx>

⁴<http://library.gnome.org/devel/glib/unstable/glib-Thread-Pools.html#GThreadPool>

⁵<http://www.oracle.com/technetwork/documentation/solaris-studio-12-192994.html>

- **Option 3:** vertical |||| |||| |||| ||||
Create 4 threads; for each element, the owning thread does 1000 iterations on that element.

We can try these and empirically see which works better. As you might expect, bad horizontal does the worst. Horizontal does best.

Automatic Parallelization. The Solaris Studio compiler yields the following output:

```
$ cc -O3 -xloopinfo -xautopar omp_vector.c
"omp_vector.c", line 5: PARALLELIZED, and serial version generated
"omp_vector.c", line 15: not parallelized, call may be unsafe
```

Note: The Solaris compiler generates two versions of the code, and decides, at runtime, if the parallel code would be faster, depending on whether the loop bounds, at runtime, are large enough to justify spawning threads.

Under the hood, most parallelization frameworks use **OpenMP**, which we'll see next time. For now, you can control the number of threads with the `OMP_NUM_THREADS` environment variable.

Autoparallelization in gcc. gcc 4.4 can also parallelize loops, but there are a couple of problems: 1) the loop parallelization doesn't seem very stable yet; 2) I can't figure out how to make gcc tell you what it did; and, perhaps most importantly for performance, 3) gcc doesn't have any heuristics yet for guessing which loops are profitable.

One way to inspect the resulting code is by giving gcc the `-S` option and looking at the resulting assembly code yourself. This is obviously not practical for production software.

```
$ gcc -std=c99 omp_vector.c -O2 -ftree-parallelize-loops=2 -S
```

The resulting `.s` file contains the following code:

```
call    GOMP_parallel_start
movl    %edi, (%esp)
call    setup_loopfn.0
call    GOMP_parallel_end
```

gcc code appears to ignore `OMP_NUM_THREADS`. Here's some potential output from a parallelized program:

```
$ export OMP_NUM_THREADS=2
$ time ./a.out
real 0m5.167s
user 0m7.872s
sys 0m0.016s
```

(When you use multiple (virtual) CPUs, CPU usage can increase beyond 100% in `top`, and real time can be less than user time in the `time` output, since user time counts the time used by all CPUs.)

Let's look at some gcc examples from: <http://gcc.gnu.org/wiki/AutoparRelated>.

Loops That gcc's Automatic Parallelization Can Handle.

Single loop:

```
for (i = 0; i < 1000; i++)
    x[i] = i + 3;
```

Nested loops with simple dependency:

```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        X[i][j] = X[i][j] + Y[i-1][j];
```

Single loop with not-very-simple dependency:

```
for (i = 0; i < 10; i++)
    X[2*i+1] = X[2*i];
```

Loops That gcc's Automatic Parallelization Can't Handle.

Single loop with if statement:

```
for (j = 0; j <= 10; j++)
    if (j > 5) X[i] = i + 3;
```

Triangle loop:

```
for (i = 0; i < 100; i++)
    for (j = i; j < 100; j++)
        X[i][j] = 5;
```

Case study: Multiplying a Matrix by a Vector.

Next, we'll see how automatic parallelization does on a more complicated program. We will progressively remove barriers to parallelization for this program:

```
1 void matVec (double **mat, double *vec, double *out,
2             int *row, int *col)
3 {
4     int i, j;
5     for (i = 0; i < *row; i++)
6     {
7         out[i] = 0;
8         for (j = 0; j < *col; j++)
9         {
10            out[i] += mat[i][j] * vec[j];
11        }
12    }
13 }
```

The Solaris C compiler refuses to parallelize this code:

```
$ cc -O3 -xloopinfo -xautopar fploop.c
"fploop.c", line 5: not parallelized, not a recognized for loop
"fploop.c", line 8: not parallelized, not a recognized for loop
```

For definitive documentation about Sun's automatic parallelization, see Chapter 10 of their *Fortran Programming Guide* and do the analogy to C:

<http://download.oracle.com/docs/cd/E19205-01/819-5262/index.html>

In this case, the loop bounds are not constant, and the write to `out` might overwrite either `row` or `col`. So, let's modify the code and make the loop bounds `ints` rather than `int *s`.

```
1 void matVec (double **mat, double *vec, double *out,
2             int row, int col)
3 {
4     int i, j;
5     for (i = 0; i < row; i++)
6     {
7         out[i] = 0;
8         for (j = 0; j < col; j++)
9         {
10            out[i] += mat[i][j] * vec[j];
11        }
12    }
13 }
```

This changes the error message:

```
$ cc -O3 -xloopinfo -xautopar fploop1.c
"fploop1.c", line 5: not parallelized, unsafe dependence
"fploop1.c", line 8: not parallelized, unsafe dependence
```

Now the problem is that `out` might alias `mat` or `vec`; as I've mentioned previously, parallelizing in the presence of aliases could change the run-time behaviour.

restrict qualifier. Recall that the `restrict` qualifier on pointer `p` tells the compiler⁶ that it may assume that, in the scope of `p`, the program will not use any other pointer `q` to access the data at `*p`.

```
1 void matVec (double **mat, double *vec, double * restrict out,
2             int row, int col)
3 {
4     int i, j;
5     for (i = 0; i < row; i++)
6     {
7         out[i] = 0;
8         for (j = 0; j < col; j++)
9         {
10            out[i] += mat[i][j] * vec[j];
11        }
12    }
13 }
```

Now Solaris `cc` is happy to parallelize the outer loop:

```
$ cc -O3 -xloopinfo -xautopar fploop2.c
"fploop2.c", line 5: PARALLELIZED, and serial version generated
"fploop2.c", line 8: not parallelized, unsafe dependence
```

⁶<http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html>

There's still a dependence in the inner loop. This dependence is because all inner loop iterations write to the same location, `out[i]`. We'll discuss that problem below.

In any case, the outer loop is the one that can actually improve performance, since parallelizing it imposes much less barrier synchronization cost waiting for all threads to finish. So, even if we tell the compiler to ignore the reduction issue, it will generally refuse to parallelize inner loops:

```
$ cc -g -O3 -xloopinfo -xautopar -xreduction fploop2.c
"fploop2.c", line 5: PARALLELIZED, and serial version generated
"fploop2.c", line 8: not parallelized, not profitable
```

Summary of conditions for automatic parallelization. Here's what I can figure out; you may also refer to Chapter 3 of the Solaris Studio *C User's Guide*, but it doesn't spell out the exact conditions either. To parallelize a loop, it must:

- have a recognized loop style, e.g. `for` loops with bounds that don't vary per iteration;
- have no dependencies between data accessed in loop bodies for each iteration;
- not conditionally change scalar variables read after the loop terminates, or change any scalar variable across iterations;
- have enough work in the loop body to make parallelization profitable.

Reductions. The concept behind a reduction (as made "famous" in MapReduce, which we'll talk about later) is reducing a set of data to a smaller set which somehow summarizes the data. For us, reductions are going to reduce arrays to a single value. Consider, for instance, this function, which calculates the sum of an array of numbers:

```
1 double sum (double *array, int length)
2 {
3     double total = 0;
4
5     for (int i = 0; i < length; i++)
6         total += array[i];
7     return total;
8 }
```

There are two barriers: 1) the value of `total` depends on what gets computed in previous iterations; and 2) addition is actually non-associative for floating-point values. (Why? When is it appropriate to parallelize non-associative operations?)

Nevertheless, the Solaris C compiler will explicitly recognize some reductions and can parallelize them for you:

```
$ cc -O3 -xautopar -xreduction -xloopinfo sum.c
"sum.c", line 5: PARALLELIZED, reduction, and serial version generated
```

Note: If we try to do the reduction on the `restricted` version of the case study, we'll get the following:

```
$ cc -O3 -xautopar -xloopinfo -xreduction -c fploop.c
"fploop.c", line 5: PARALLELIZED, and serial version generated
"fploop.c", line 8: not parallelized, not profitable
```

Dealing with function calls. Generally, function calls can have arbitrary side effects. Production compilers will usually avoid parallelizing loops with function calls; research compilers try to ensure that functions are pure and then parallelize them. (This is why functional languages are nice for parallel programming: impurity is visible in type signatures.)

For builtin functions, like `sin()`, you can promise to the compiler that you didn't replace them with your own implementations (`-xbuiltin`), and then the compiler will parallelize the loop.

Another option is to crank up the optimization level (`-xO4`), or to explicitly tell the compiler to inline certain functions (`-xinline=`), thereby enabling parallelization.

Helping the compiler parallelize. Let's summarize what we've seen. To help the compiler, we can use the `restrict` qualifier on pointers (possibly copying a pointer to a `restrict`-qualified pointer: `int * restrict p = s->p;`); and, we can make sure that loop bounds don't change in the loop (e.g. by using temporary variables). Some compilers can automatically create different versions for the alias-free case and the (parallelized) aliased case; at runtime, the program runs the aliased case if the inputs permit.

Lecture 9 — February 5, 2013

Patrick Lam

version 1

Now that we've seen automatic parallelization, let's talk about manual parallelization using OpenMP.

About OpenMP. OpenMP (Open Multiprocessing) is an API specification which allows you to tell the compiler how you'd like your program to be parallelized. Implementations of OpenMP include compiler support (present in Intel's compiler, Solaris's compiler, `gcc` as of 4.2, and Microsoft Visual C++) as well as a runtime library.

You use OpenMP¹ by specifying directives in the source code. In C and C++, these directives are pragmas of the form `#pragma omp . . .`. There is also OpenMP syntax for Fortran.

Here are some benefits of the OpenMP approach:

- Because OpenMP uses compiler directives, you can easily tell the compiler to build a parallel version or a serial version (by ignoring the directives). This can simplify debugging, since you have some chance of observing differences in behaviour between the versions.
- OpenMP's approach also separates the parallelization implementation (inserted by the compiler) from the algorithm implementation (which you provide), making the algorithm easier to read. Plus, you're not responsible for dealing with thread libraries.
- The directives apply to limited parts of the code, thus supporting incremental parallelization of the program, starting with the hotspots.

Let's look at a simple example:

```
void calc (double *array1, double *array2, int length) {  
    #pragma omp parallel for  
    for (int i = 0; i < length; i++) {  
        array1[i] += array2[i];  
    }  
}
```

This `#pragma` instructs the C compiler to parallelize the loop. It is the responsibility of the developer to make sure that the parallelization is safe; for instance, `array1` and `array2` had better not overlap. You no longer need to supply `restrict` qualifiers, although it's still not a bad idea.

OpenMP will always start parallel threads if you tell it to, dividing the iterations contiguously among the threads.

Let's look at the parts of this `#pragma`.

- `#pragma omp` indicates an OpenMP directive;

¹More information: <https://computing.llnl.gov/tutorials/openMP/>

- `parallel` indicates the start of a parallel region; and
- `for` tells OpenMP to run the following `for` loop in parallel.

When you run the parallelized program, the runtime library starts up a number of threads and assigns a subrange of the loop range to each of the threads.

Restrictions. OpenMP places some restrictions on loops that it's going to parallelize:

- the loop must be of the form

```
for (init expression; test expression; increment expression);
```

- the loop variable must be integer (signed or unsigned), pointer, or a C++ random access iterator;
- the loop variable must be initialized to one end of the range;
- the loop increment amount must be loop-invariant (constant with respect to the loop body);
- the test expression must be one of `>`, `>=`, `<`, or `<=`, and the comparison value (bound) must be loop-invariant.

(These restrictions therefore also apply to automatically parallelized loops.) If you want to parallelize a loop that doesn't meet the restriction, restructure it so that it does, as we saw last time.

Runtime effect. When you compile a program with OpenMP directives, the compiler generates code to spawn a *team* of threads and automatically splits off the worker-thread code into a separate procedure. The code uses fork-join parallelism, so when the master thread hits a parallel region, it gives work to the worker threads, which execute and report back. Then the master thread continues running, while the worker threads wait for more work.

You can specify the number of threads by setting the `OMP_NUM_THREADS` environment variable (adjustable by calling `omp_set_num_threads()`), and you can get the Solaris compiler to tell you what it did by giving it the options `-xopenmp -xloopinfo`.

Variable scoping

When using multiple threads, some variables, like loop counters, should be thread-local, or *private*, while other variables should be *shared* between threads. Changes to shared variables are visible to all threads, while changes to private variables are visible only to the changing thread. Let's look at the defaults that OpenMP uses to parallelize the above code.

```
$ er_src parallel-for.o
1. void calc (double *array1, double *array2, int length) {
    <Function: calc>
```

```

Source OpenMP region below has tag R1
Private variables in R1: i
Shared variables in R1: array2, length, array1
2.    #pragma omp parallel for

Source loop below has tag L1
L1 autoparallelized
L1 parallelized by explicit user directive
L1 parallel loop-body code placed in function _$d1A2.calc along with 0 inner loops
L1 multi-versioned for loop-improvement:dynamic-alias-disambiguation.
    Specialized version is L2
3.    for (int i = 0; i < length; i++) {
4.        array1[i] += array2[i];
5.    }
6. }

```

We can see that the loop variable `i` is private, while the `array1`, `array2` and `length` variables are shared. Actually, it would be fine for the `length` variable to be either shared or private, but if it was private, then you would have to copy in the appropriate initial value. The `array` variables, though, need to be shared.

Summary of default rules. Loop variables are private; variables defined in parallel code are private; and variables defined outside the parallel region are shared.

You can disable the default rules by specifying `default(none)` on the `parallel` pragma, or you can give explicit scoping:

```
#pragma omp parallel for private(i) shared(length, array1, array2)
```

Reductions

Recall that we introduced the concept of a reduction, e.g.

```
for (int i = 0; i < length; i++) total += array[i];
```

What is the appropriate scope for `total`? We want each thread to be able to write to it, but we don't want race conditions. Fortunately, OpenMP can deal with reductions as a special case:

```
#pragma omp parallel for reduction (+:total)
```

specifies that the `total` variable is the accumulator for a reduction over `+`. OpenMP will create local copies of `total` and combine them at the end of the parallel region.

Accessing Private Data outside a Parallel Region

A related problem with private variables is that sometimes you need access to them outside their parallel region. Here's some contrived code.

```

int data=1;
#pragma omp parallel for private(data)
for (int i = 0; i < 100; i++)
    printf ("data=%d\n", data);

```

Since `data` is private, OpenMP is not going to copy in the initial value 1 for it, so you'll get an undefined value. To make OpenMP initialize private variables with the master thread's values for those variables, use `firstprivate(data)`. Conversely, to get a value out from the (sequentially) last iteration of the loop, use `lastprivate(data)`.

Thread-private Data. A variant on `private` is `threadprivate`. Thread-private data is also local to each thread. The main difference is that `private` is for transient data, typically local variables, declared at the start of a region, while `threadprivate` is for persistent data, e.g. declared at file scope, which lives beyond a single parallel region. Instead of `firstprivate(local)` for `private(local)`, use `copyin(global)` for `threadprivate(global)`.

```
#include <omp.h>
#include <stdio.h>

int tid, a, b;

#pragma omp threadprivate(a)

int main(int argc, char *argv[])
{
    printf("Parallel #1 Start\n");
    #pragma omp parallel private(b, tid)
    {
        tid = omp_get_thread_num();
        a = tid;
        b = tid;
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    printf("Sequential code\n");
    printf("Parallel #2 Start\n");
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("T%d: a=%d, b=%d\n", tid, a, b);
    }

    return 0;
}
```

This yields something like the following output:

```
% ./a.out
Parallel #1 Start
T6: a=6, b=6
T1: a=1, b=1
T0: a=0, b=0
T4: a=4, b=4
```



```

T2: a=2, b=2
T3: a=3, b=3
T5: a=5, b=5
T7: a=7, b=7
Sequential code
Parallel #2 Start
T0: a=0, b=0
T6: a=6, b=0
T1: a=1, b=0
T2: a=2, b=0
T5: a=5, b=0
T7: a=7, b=0
T3: a=3, b=0
T4: a=4, b=0

```

Collapsing Loops. Usually, when you have nested loops, it's best to parallelize the outermost loop. Why?

Sometimes, however, that just doesn't work out. The main issue is that the outermost loop may have too low a trip count to be worth parallelizing. You could instead parallelize the inner loop by putting the pragma just before the inner loop, but then you pay more overhead than you need to. OpenMP therefore supports *collapsing* loops, which creates a single loop performing all the iterations of the collapsed loops. Consider:

```

#include <math.h>
int main() {
    double array[2][10000];
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 10000; j++)
            array[i][j] = sin(i+j);
    return 0;
}

```

Parallelizing the outer loop only enables the use of 2 threads. Parallelizing both loops together enables the use of up to 20,000 threads, although the loop body is too small for that to be worthwhile.

Where have you seen something like a manually collapsed loop?

Better Performance Through Scheduling

OpenMP tries to guess how many iterations to distribute to each thread in a team. The default mode is called *static scheduling*; in this mode, OpenMP looks at the number of iterations it needs

to run, assumes they all take the same amount of time, and distributes them evenly. So for 100 iterations and 2 threads, the first thread gets 50 iterations and the second thread gets 50 iterations.

This assumption doesn't always hold; consider, for instance, the following (contrived) code:

```
double calc(int count) {
    double d = 1.0;
    for (int i = 0; i < count*count; i++) d += d;
    return d;
}

int main() {
    double data[200][100];
    int i, j;
    #pragma omp parallel for private(i, j) shared(data)
    for (int i = 0; i < 200; i++) {
        for (int j = 0; j < 100; j++) {
            data[i][j] = calc(i+j);
        }
    }
    return 0;
}
```

This code gives sublinear scaling, because the earlier iterations finish faster than the later iterations, and the program needs to wait for all iterations to complete.

Telling OpenMP to use a *dynamic schedule* can enable better parallelization: the runtime distributes work to each thread in chunks, which results in less waiting. Just add `schedule(dynamic)` to the pragma. Of course, this has more overhead, since the threads need to solicit the work, and there is a potential serialization bottleneck in soliciting work from the single work queue.

The default chunk size is 1, but you can specify it yourself, either using a constant or a value computed at runtime, e.g. `schedule(dynamic, n/50)`. Static scheduling also accepts a chunk size.

OpenMP has an even smarter work distribution mode, **guided**, where it changes the chunk size according to the amount of work remaining. You can specify a minimum chunk size, which defaults to 1. There are also two meta-modes, **auto**, which leaves it up to OpenMP, and **runtime**, which leaves it up to the `OMP_SCHEDULE` environment variable.

Beyond for Loops: OpenMP parallel sections and tasks

The part of OpenMP we've seen so far has been strictly less powerful than pthreads (but harder to misuse): we have only parallelized specific forms of **for** loops. This reflects OpenMP's scientific-computation heritage, where you have huge FORTRAN matrix calculations to parallelize. However, these days we also care about parallelism in more general settings, so OpenMP now provides more ways to parallelize.

Parallel Sections. The first mechanism, *parallel sections*, is a purely-static mechanism for specifying independent units of work which ought to be run in parallel. For instance, you can set up

two (and exactly two, in this example) linked lists simultaneously as follows:

```
#include <stdlib.h>

typedef struct s { struct s* next; } S;

void setuplist (S* current) {
    for (int i = 0; i < 10000; i++) {
        current->next = (S*) malloc (sizeof(S));
        current = current->next;
    }
    current->next = NULL;
}

int main() {
    S var1, var2;
    #pragma omp parallel sections
    {
        #pragma omp section
        { setuplist (&var1); }
        #pragma omp section
        { setuplist (&var2); }
    }
    return 0;
}
```

Note that the structure of the parallelism is explicitly visible in the structure of the code, and that you don't get to start an unbounded number of threads with the parallel sections mechanism.

What's another potential barrier to parallelism in the above code?

Nested Parallelism. Instead of collapsing loops, we can specify nested parallelism; we might have, for instance, two parallel sections, each of which contain parallel for loops. To enable such nested parallelism, you have to call `omp_set_nested` with a non-zero value. The runtime might refuse; call `omp_get_nested` to find out if the runtime complied or not. You can also set the `OMP_NESTED` environment variable to enable nesting.

Here's an example of nested parallelism.

```
#include <stdlib.h>
#include <omp.h>

int main() {
    double *array1, *array2;
    omp_set_nested(1);
    #pragma omp parallel sections shared(array1, array2)
    {
        #pragma omp section
        {
            array1 = (double*) malloc(sizeof (double)*1024*1024);
            #pragma omp parallel for shared(array1)
            for (int i = 0; i < 1024*1024; i++)

```

```

        array1[i] = i;
    }
#pragma omp section
    {
        array2 = (double*) malloc(sizeof (double)*1024*512);
#pragma omp parallel for shared(array2)
        for (int i = 0; i < 1024*512; i++)
            array2[i] = i;
    }
}
}

```

Tasks: OpenMP’s thread-like mechanism. The main new feature in OpenMP 3.0 is the notion of *tasks*. When the program executes a `#pragma omp task` statement, the code inside the task is split off as a task and scheduled to run sometime in the future. Tasks are more flexible than parallel sections, because parallel sections constrain exactly how many threads are supposed to run, and there is also always a join at the end of the parallel section. On the other hand, the OpenMP runtime can assign any task to any thread that’s running. Tasks therefore have lower overhead.

Two examples which show off tasks, from [ACD⁺09], include a web server (with unstructured requests) and a user interface which allows users to start tasks that are to run in parallel.

Here’s pseudocode for the Boa webserver main loop from [ACD⁺09].

```

#pragma omp parallel
/* a single thread manages the connections */
#pragma omp single nowait
while (!end) {
    process any signals
    foreach request from the blocked queue {
        if (request dependencies are met) {
            extract from the blocked queue
            /* create a task for the request */
#pragma omp task untied
            serve_request(request);
        }
    }
    if (new connection) {
        accept_connection();
        /* create a task for the request */
#pragma omp task untied
        serve_request(new connection);
    }
    select();
}

```

The **untied** qualifier lifts restrictions on the task-to-thread mapping; we won’t talk about that. The **single** directive indicates that the runtime is only to use one thread to execute the next statement; otherwise, it could execute N copies of the statement, which does belong to a OpenMP **parallel** construct.

More OpenMP features. Random fact: you can use the `flush` directive to make sure that all values in registers or cache are written to memory. Usually, this isn't a problem for OpenMP programs, because of the way they're written. On a related note, the `barrier` directive explicitly instructs the runtime to wait for all threads to complete; OpenMP also has implicit barriers at the end of parallel sections.

OpenMP also supports critical sections (one thread at a time), atomic sections, and typical mutex locks (`omp_set_lock`, `omp_unset_lock`).

References

- [ACD⁺09] Eduard Ayguadé, Nawal Coptý, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009.

We are going to go and present OpenMP in more detail. This will help you use it in Assignment 2.

OpenMP is a portable, easy-to-use parallel programming API. It combines compiler directives with runtime library routines and uses environment variables for setup. Note that you can detect its presence in your code with `#ifdef _OPENMP`.

For exhaustive documentation, see <http://www.openmp.org/mp-documents/OpenMP3.1.pdf>.

Directive Format. All OpenMP directives follow this pattern:

```
#pragma omp directive-name [clause [[,] clause]*]
```

There are **16** OpenMP directives. Each directive applies to the immediately-following statement, which is either a single statement or a compound statement `{ ... }`.

Most clauses have a *list* as an argument. A list is a comma-separated list of list items. For C and C++, a list item is simply a variable name.

Data Terminology

OpenMP includes three keywords for variable scope and storage:

- `private`;
- `shared`; and
- `threadprivate`.

Private Variables. You can declare private variables with the `private` clause. This creates new storage, on a per-thread basis, for the variable—it does not copy variables. The scope of the variable extends from the start of the region where the variable exists to the end of that region; the variable is destroyed afterwards.

Some Pthread pseudocode for private variables:

```
void* run(void* arg) {  
    int x;  
    // use x  
}
```

Shared Variables. The opposite of a private variable is a **shared** variable. All threads have access to the same block of data when accessing such a variable.

The relevant Pthread pseudocode is:

```
int x;

void* run(void* arg) {
    // use x
}
```

Thread-Private Variables. Finally, OpenMP supports **threadprivate** variables. This is like a **private** variable in that each thread makes a copy of the variable. However, the scope is different. Such variables are accessible to the thread in any parallel region.

This example will make things clearer. The OpenMP code:

```
int x;
#pragma omp threadprivate(x)
```

maps to this Pthread pseudocode:

```
int x;
int x[NUM_THREADS];

void* run(void* arg) {
    // use x[thread_self()]
}
```

A variable may not appear in **more than one clause** on the same directive. (There's an exception for **firstprivate** and **lastprivate**, which we'll see later.) By default, variables declared in regions are private; those outside regions are shared. (An exception: anything with dynamic storage is shared).

Directives

We'll talk about the different OpenMP directives next. These are the key language features you'll use to tell OpenMP what to parallelize.

Parallel

```
#pragma omp parallel [clause [,] clause]*
```

This is the most basic directive in OpenMP. It tells OpenMP to form a team of threads and start parallel execution. The thread that enters the region becomes the **master** (thread 0).

Allowed Clauses: **if**, **num_threads**, **default**, **private**, **firstprivate**, **shared**, **copyin**, **reduction**.

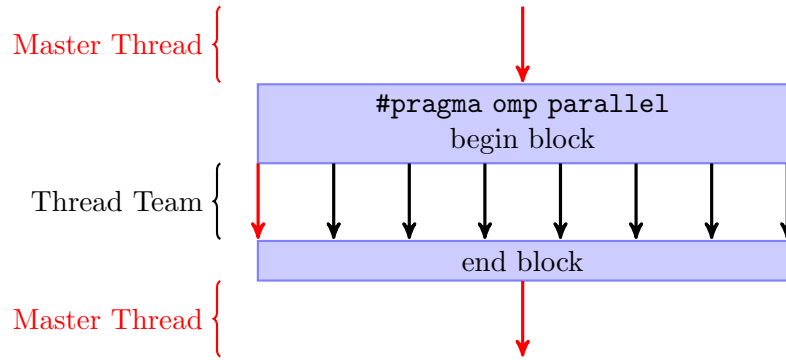


Figure 1: Visual view of `parallel` directive.

Figure 1 illustrates what `parallel` does. By default, the number of threads used is set globally, either automatically or manually. After the parallel block, the thread team sleeps until it's needed again.

```
#pragma omp parallel
{
    printf("Hello!");
}
```

If the number of threads is 4, this produces:

```
Hello!
Hello!
Hello!
Hello!
```

if and num_threads Clauses. Directives take clauses. The first one we'll see (besides the variable scope clauses) are `if` and `num_threads`.

`if(primitive-expression)`

The `if` clause allows you to control at runtime whether or not to run multiple threads or just one thread in its associated parallel section. If *primitive-expression* evaluates to 0, i.e. `false`, then only one thread will execute in the parallel section. (It's what you would expect.)

If the parallel section is going to run multiple threads (e.g. `if` expression is true; or if there is no `if` expression), we can then specify how many threads.

num_threads(*integer-expression*)

This spawns at most **num_threads**, depending on the number of threads available. It can only guarantee the number of threads requested if **dynamic adjustment** for number of threads is off and enough threads aren't busy.

reduction Clause. We saw reductions last time. Here's another look.

reduction(*operator:list*)

Operators (and their associated initial value)

+	(0)		-	(0)			(0)		&&	(1)		max	MAX
	(1)		&	(~0)		^	(0)			(0)		min	MIN

Each thread gets a *private* copy of the variable. The variable is initialized by OpenMP (so you don't need to do anything else). At the end of the region, OpenMP updates your result using the operator.

```
void* run(void* arg) {
    variable = initial value;
    // code inside block—modifies variable
    return variable;
}

// ... later in master thread (sequentially):
variable = initial value
for t in threads {
    thread_variable
    pthread_join(t, &thread_variable);
    variable = variable (operator) thread_variable;
}
```

(For) Loop Directive

Inside a parallel section, we can specify a for loop clause, as follows.

```
#pragma omp for [clause [[,] clause]*]
```

Iterations of the loop will be distributed among the current team of threads. This clause only supports simple “for” loops with invariant bounds (bounds do not change during the loop). Loop variable is implicitly private; OpenMP sets the correct values.

Allowed Clauses: **private**, **firstprivate**, **lastprivate**, **reduction**, **schedule**, **collapse**, **ordered**, **nowait**.

schedule Clause.

```
schedule(kind[, chunk_size])
```

The **chunk_size** is the number of iterations a thread should handle at a time. **kind** is one of:

- **static**: divides the number of iterations into chunks and assigns each thread a chunk in round-robin fashion (before the loop executes).
- **dynamic**: divides the number of iterations into chunks and assigns each available thread a chunk, until there are no chunks left.
- **guided**: same as dynamic, except **chunk_size** represents the minimum size. This starts by dividing the loop into large chunks, and decreases the chunk size as fewer iterations remain.
- **auto**: obvious (OpenMP decides what’s best for you).
- **runtime**: also obvious; we’ll see how to adjust this later.

collapse and ordered Clauses.

```
collapse(n)
```

This collapses n levels of loops. Obviously, this only has an effect if $n \geq 2$; otherwise, nothing happens. Collapsed loop variables are also made private.

```
ordered
```

This enables the use of **ordered** directives inside loop, which we’ll see below.

Ordered directive

#pragma omp ordered

To use this directive, the containing loop must have an **ordered** clause. OpenMP will ensure that the ordered directives are executed the same way the sequential loop would (one at a time). Each iteration of the loop may execute **at most one** ordered directive.

Let's see what that means by way of two examples.

Invalid Use of Ordered. This doesn't work.

```
void work(int i) {
    printf("i = %d\n", i);
}
...
int i;
#pragma omp for ordered
for (i = 0; i < 20; ++i) {

    #pragma omp ordered
    work(i);

    // Each iteration of the loop has 2 "ordered" clauses!
    #pragma omp ordered
    work(i + 100);
}
```

Valid Use of Ordered. This does.

```
void work(int i) {
    printf("i = %d\n", i);
}
...
int i;
#pragma omp for ordered
for (i = 0; i < 20; ++i) {
    if (i <= 10) {
        #pragma omp ordered
        work(i);
    }
    if (i > 10) {
        // two ordered clauses are mutually-exclusive
        #pragma omp ordered
        work(i+100);
    }
}
```

Note: if we change $i > 10$ to $i > 9$, the use becomes invalid because the iteration $i = 9$ contains two **ordered** directives.

Tying It All Together. Here's a larger example.

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int j, k, a;
    #pragma omp parallel num_threads(2)
    {
        #pragma omp for collapse(2) ordered private(j,k) \
            schedule(static,3)
        for (k = 1; k <= 3; ++k)
            for (j = 1; j <= 2; ++j) {
                #pragma omp ordered
                printf("t[%d] k=%d j=%d\n",
                    omp_get_thread_num(),
                    k, j);
            }
    }
    return 0;
}
```

Output of Previous Example. And here's what it does.

```
t[0] k=1 j=1
t[0] k=1 j=2
t[0] k=2 j=1
t[1] k=2 j=2
t[1] k=3 j=1
t[1] k=3 j=2
```

Note: the output will be deterministic; still, the program will run two threads as long as the thread limit is at least 2.

Parallel Loop Directive

This directive is shorthand:

```
#pragma omp parallel for [clause [,] clause]*]
```

We could equally well write:

```
#pragma omp parallel
{
    #pragma omp for
    {
```

```

    }
}

```

but the single directive is shorter; this idiom happens a lot.

Allowed Clauses: everything allowed by **parallel** and **for**, except **nowait**.

Sections

Another OpenMP parallelism idiom is sections.

```
#pragma omp sections [clause [[,] clause]*]
```

Allowed Clauses: **private**, **firstprivate**, **lastprivate**, **reduction**, **nowait**.

Each **sections** directive must contain one or more **section** directive:

```
#pragma omp section
```

Sections distributed among current team of threads. They statically limit parallelism to the number of sections which are lexically in the code.

Parallel Sections. Another common idiom.

```
#pragma omp parallel sections [clause [[,] clause]*]
```

As with parallel for, this is basically shorthand for:

```
#pragma omp parallel
{
    #pragma omp sections
    {

    }
}

```

Allowed Clauses: everything allowed by **parallel** and **sections**, except **nowait**.

Single

When we'd be otherwise running many threads, we can state that some particular code block should be run by only one method.

```
#pragma omp single
```

Only a single thread executes the region following the directive. The thread is not guaranteed to be the master thread.

Allowed Clauses: **private**, **firstprivate**, **copyprivate**, **nowait**. Must not use **copyprivate** with **nowait**

Barrier

```
#pragma omp barrier
```

Waits for all the threads in the team to reach the barrier before continuing. In other words, this constitutes a synchronization point. Loops, sections, and single all have an implicit barrier at the end of their region (unless you use **nowait**). Note that barrier *cannot* be used in any conditional blocks.

This mechanism is analogous to `pthread_barrier` in Pthreads.

Master

Sometimes we do want to guarantee that only the master thread runs some code.

```
#pragma omp master
```

This is similar to the **single** directive, except that the master thread (and only the master thread) is guaranteed to enter this region. No implied barriers.

Also, no clauses.

Critical

We turn our attention to synchronization constructs. First, let's examine critical.

```
#pragma omp critical [(name)]
```

The enclosed region is guaranteed to only run one thread at a time (on a per-name basis). This is the same as a block of code in Pthreads surrounded by a **mutex** lock and unlock.

Atomic

We can also request atomic operations.

```
#pragma omp atomic [read | write | update | capture]
                expression-stmt
```

This ensures that a specific storage location is updated atomically. Atomics should be more efficient than using critical sections (or else why would they include it?)

- **read expression:** `v = x;`
- **write expression:** `x = expr;`
- **update expression:** `x++; x--; ++x; --x; x binop= expr; x = x binop expr;`
`expr` must not access the same location as `v` or `x`. `v` and `x` must not access the same location; must be primitives. All operations to `x` are atomic.
- **capture expression:** `v = x++; v = x--; v = ++x; v = --x; v = x binop= expr;`
Capture expressions perform the indicated update. At the same time, they also store the original or final value computed into location `v`.

Atomic Capture

There's also a directive for atomic capture where you specify a more complicated block to be run atomically.

```
#pragma omp atomic capture
                structured-block
```

Other Directives

We'll get into these next lecture.

- `task`
- `taskyield`
- `taskwait`
- `flush`

More Scoping Clauses: `firstprivate`, `lastprivate`, `copyin`, `copyprivate` and `default`

Besides the `shared`, `private` and `threadprivate`, OpenMP also supports `firstprivate` and `lastprivate`, which work like this.

Pthreads pseudocode for **firstprivate** clause:

```
int x;

void* run(void* arg) {
    int thread_x = x;
    // use thread_x
}
```

Pthread pseudocode for the **lastprivate** clause:

```
int x;

void* run(void* arg) {
    int thread_x;
    // use thread_x
    if (last_iteration) {
        x = thread_x;
    }
}
```

In other words, **lastprivate** makes sure that the variable **x** has the same value as if the loop executed sequentially.

copyin is like **firstprivate**, but for threadprivate variables.

Pthreads pseudocode for **copyin**:

```
int x;
int x[NUM_THREADS];

void* run(void* arg) {
    x[thread_num] = x;
    // use x[thread_num]
}
```

The **copyprivate** clause is only used with **single**. It copies the specified private variables from the thread to all other threads. It cannot be used with **nowait**.

Defaults. **default(shared)** makes all variables shared; **default(none)** prevents sharing by default (creating compiler errors if you treat a variable as shared.)

Runtime Library Routines

To use the runtime library and call OpenMP functions (rather than using pragmas), you need to `#include <omp.h>`.

- `int omp_get_num_procs();` return the number of processors in the system.

- `int omp_get_thread_num();` return the thread number of the currently executing thread (the master thread will return 0).
- `int omp_in_parallel();` returns true if currently in a parallel region.
- `int omp_get_num_threads();` return the number of threads in the current team.

Locks in OpenMP. OpenMP provides two types of locks:

- Simple: cannot be acquired if it is already held by the task trying to acquire it.
- Nested: can be acquired multiple times by the same task before being released (like Java).

Lock usage is similar to Pthreads:

<code>omp_init_lock</code>	<code>omp_init_nest_lock</code>
<code>omp_destroy_lock</code>	<code>omp_destroy_nest_lock</code>
<code>omp_set_lock</code>	<code>omp_set_nest_lock</code>
<code>omp_unset_lock</code>	<code>omp_unset_nest_lock</code>
<code>omp_test_lock</code>	<code>omp_test_nest_lock</code>

Timing. You can measure how long things take, which is useful for domain-specific profiling.

- `double omp_get_wtime();` elapsed wall clock time in seconds (since some time in the past).
- `double omp_get_wtick();` precision of the timer.

Other Routines. Wight see these in later lectures. Included for completeness:

- `int omp_get_level();`
- `int omp_get_active_level();`
- `int omp_get_ancestor_thread_num(int level);`
- `int omp_get_team_size(int level);`
- `int omp_in_final();`

Internal Control Variables

OpenMP uses internal variables to control how it handles threads. These can be set with clauses, runtime routines, environment variables, or just from defaults. Routines will be represented as all-lower-case, environment variables as all-upper-case.

Clause > Routine > Environment Variable > Default Value

All values (except 1) are implementation defined.

Operation of Parallel Regions. We can control how parallel regions work with the following variables.

dyn-var

- is dynamic adjustment of the number of threads enabled?
- **Set by:** `OMP_DYNAMIC omp_set_dynamic`
- **Get by:** `omp_get_dynamic`

nest-var

- is nested parallelism enabled?
- **Set by:** `OMP_NESTED omp_set_nested`
- **Get by:** `omp_get_nested`
- Default value: `false`

thread-limit-var

- maximum number of threads in the program
- **Set by:** `OMP_NUM_THREADS omp_set_num_threads`
- **Get by:** `omp_get_max_threads`

max-active-levels-var

- Maximum number of nested active parallel regions
- **Set by:** `OMP_MAX_ACTIVE_LEVELS
omp_set_max_active_levels`
- **Get by:** `omp_get_max_active_levels`

Operation of Parallel Regions/Loops. These apply to both parallel regions and loops.

nthreads-var

- number of threads requested for parallel regions.
- **Set by:** `OMP_NUM_THREADS omp_set_num_threads`
- **Get by:** `omp_get_max_threads`

run-sched-var

- **schedule** that the runtime schedule clause uses for loops.
- **Set by:** `OMP_SCHEDULE omp_set_schedule`
- **Get by:** `omp_get_schedule`

Program Execution. We can also control program execution.

bind-var

- Controls binding of threads to processors.
- **Set by:** OMP_PROC_BIND

stacksize-var

- Controls stack size for threads.
- **Set by:** OMP_STACK_SIZE

wait-policy-var

- Controls desired behaviour of waiting threads.
- **Set by:** OMP_WAIT_POLICY

Summary

- Main concepts:
 - **parallel**
 - **for (ordered)**
 - **sections**
 - **single**
 - **master**
- Synchronization:
 - **barrier**
 - **critical**
 - **atomic**
- Data sharing: **private, shared, threadprivate**
- You now should be able to use OpenMP effectively with a reference.

Reference Card. <http://openmp.org/mp-documents/OpenMP3.1-CCard.pdf>

A Warning About Using OpenMP Directives. Write your code so that simply eliding OpenMP directives gives a valid program. For instance, this is wrong:

```
if (a != 0)
    #pragma omp barrier // wrong!
if (a != 0)
    #pragma omp taskyield // wrong!
```

Use this instead:

```
if (a != 0) {
    #pragma omp barrier
}
if (a != 0) {
    #pragma omp taskyield
}
```

OpenMP Memory Model, Its Pitfalls, and How to Mitigate Them

OpenMP uses a **relaxed-consistency, shared-memory** model. This doesn't do what you want. Here are its properties:

- All threads share a single store called *memory*—this store may not actually represent RAM.
- Each thread can have its own *temporary* view of memory.
- A thread's *temporary* view of memory is not required to be consistent with memory.

We'll talk more about memory models later. Now we're going to talk about the OpenMP model and why it's a problem.

Memory Model Pitfall. Consider this code.

```

                                a = b = 0
/* thread 1 */                  /* thread 2 */

atomic(b = 1) // [1]            atomic(a = 1) // [3]
atomic(tmp = a) // [2]          atomic(tmp = b) // [4]
if (tmp == 0) then              if (tmp == 0) then
    // protected section        // protected section
end if                          end if
```

Does this code actually prevent simultaneous execution? Let's reason about possible states.

Order				t1 tmp	t2 tmp
1	2	3	4	0	1
1	3	2	4	1	1
1	3	4	2	1	1
3	4	1	2	1	0
3	1	2	4	1	1
3	1	4	2	1	1

Looks like it (at least intuitively).

Sorry! With OpenMP's memory model, no guarantees: the update from one thread may not be seen by the other.

Restoring Sanity with Flush. We do rely on shared memory working "properly", but that's expensive. So OpenMP provides the **flush** directive.

```
#pragma omp flush [(list)]
```

This directive makes the thread's temporary view of memory consistent with main memory; it:

- enforces an order on the memory operations of the variables.

The variables in the list are called the *flush-set*. If you give no variables, the compiler will determine them for you.

Enforcing an order on the memory operations means:

- All read/write operations on the *flush-set* which happen before the **flush** complete before the flush executes.
- All read/write operations on the *flush-set* which happen after the **flush** complete after the flush executes.
- Flushes with overlapping *flush-sets* can not be reordered.

To show a consistent value for a variable between two threads, OpenMP must run statements in this order:

1. t_1 writes the value to v ;
2. t_1 flushes v ;
3. t_2 flushes v also;
4. t_2 reads the consistent value from v .

Let's revise the example again.

```

                                a = b = 0
/* thread 1 */                  /* thread 2 */

atomic(b = 1)                    atomic(a = 1)
flush(b)                         flush(a)
flush(a)                         flush(b)
atomic(tmp = a)                  atomic(tmp = b)
if (tmp == 0) then               if (tmp == 0) then
    // protected section        // protected section
end if                           end if

```

OK. Will this now prevent simultaneous access?

Well, no.

The compiler can reorder the `flush(b)` in thread 1 or `flush(a)` in thread 2. If `flush(b)` gets reordered to after the protected section, we will not get our intended operation.

Correct Example. We have to provide a list of variables to `flush` to prevent re-ordering:

```

                                a = b = 0
/* thread 1 */                  /* thread 2 */

atomic(b = 1)                    atomic(a = 1)
flush(a, b)                      flush(a, b)
atomic(tmp = a)                  atomic(tmp = b)
if (tmp == 0) then               if (tmp == 0) then
    // protected section        // protected section
end if                           end if

```

Where There's No Implicit Flush:

- at entry to **for**;
- at entry to, or exit from, **master**;
- at entry to **sections**;
- at entry to **single**;
- at exit from **for**, **single** or **sections** with a **nowait**
 - **nowait** removes implicit flush along with the implicit barrier

This is not true for OpenMP versions before 2.5, so be careful.

Final thoughts on flush. We've seen that it's very difficult to use flush properly. Really, you should be using mutexes or other synchronization instead of flush¹, because you'll probably just get it wrong. But now you know what flush means.

OpenMP Task Directive

`#pragma omp task [clause [[,] clause]*]`

Generates a task for a thread in the team to run. When a thread enters the region it may:

- immediately execute the task; or
- defer its execution. (any other thread may be assigned the task)

Allowed Clauses: **if**, **final**, **untied**, **default**, **mergeable**, **private**, **firstprivate**, **shared**

if and final Clauses.

if (*scalar-logical-expression*)

When expression is **false**, generates an undeferred task—
the generating task region is suspended until execution of the undeferred task finishes.

final (*scalar-logical-expression*)

When expression is **true**, generates a final task.
All tasks within a final task are *included*.
Included tasks are undeferred and also execute immediately in the same thread.

Let's look at some examples of these clauses.

```
void foo () {
    int i;
    #pragma omp task if(0) // This task is undeferred
    {
        #pragma omp task
        // This task is a regular task
        for (i = 0; i < 3; i++) {
            #pragma omp task
            // This task is a regular task
        }
    }
}
```

¹<http://www.thinkingparallel.com/2007/02/19/please-dont-rely-on-memory-barriers-for-synchronization/>

```

        bar();
    }
}
#pragma omp task final(1) // This task is a regular task
{
    #pragma omp task // This task is included
    for (i = 0; i < 3; i++) {
        #pragma omp task
        // This task is also included
        bar();
    }
}
}

```

untied and mergeable Clauses.

untied

- A suspended task can be resumed by any thread.
- “untied” is ignored if used with **final**.
- Interacts poorly with thread-private variables and `gettid()`.

mergeable

- For an undeferred or included task, allows the implementation to generate a merged task instead.
- In a merged task, the implementation may re-use the environment from its generating task (as if there was no task directive).

For more: docs.oracle.com/cd/E24457_01/html/E21996/gljyr.html

```

#include <stdio.h>
void foo () {
    int x = 2;
    #pragma omp task mergeable
    {
        x++; // x is by default firstprivate
    }
    #pragma omp taskwait
    printf("%d\n",x); // prints 2 or 3
}

```


This is an incorrect usage of **mergeable**: the output depends on whether or not the task got merged. Merging tasks (when safe) produces more efficient code.

Taskyield.

```
#pragma omp taskyield
```

This directive specifies that the current task can be suspended in favour of another task.

Here's a good use of **taskyield**.

```
void foo (omp_lock_t * lock, int n) {
    int i;
    for ( i = 0; i < n; i++ )
        #pragma omp task
        {
            something_useful();
            while (!omp_test_lock(lock)) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

Taskwait.

```
#pragma omp taskwait
```

Waits for the completion of the current task's child tasks.

OpenMP Examples

We are next going to look at a sequence of examples showing how to use OpenMP.

```
struct node {
    struct node *left;
    struct node *right;
};
```

```

extern void process(struct node *);

void traverse(struct node *p) {
    if (p->left)
        #pragma omp task
        // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task
        // p is firstprivate by default
        traverse(p->right);
    process(p);
}

```

If we want to guarantee a post-order traversal, we simply need to insert an explicit `#pragma omp taskwait` after the two calls to `traverse` and before the call to `process`.

Parallel Linked List Processing. We can spawn tasks to process linked list entries. It's hard to use two threads to traverse the list, though.

```

// node struct with data and pointer to next
extern void process(node* p);

void increment_list_items(node* head) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            node * p = head;
            while (p) {
                #pragma omp task
                {
                    process(p);
                }
                p = p->next;
            }
        }
    }
}

```

Using Lots of Tasks. Let's see what happens if we spawn lots of tasks in a `single` directive.

```

#define LARGENUMBER 10000000
double item[LARGENUMBER];
extern void process(double);

```

```

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            int i;
            for (i=0; i<LARGENUMBER; i++)
                #pragma omp task
                // i is firstprivate, item is shared
                process(item[i]);
        }
    }
}

```

In this case, the main loop generates tasks, which are all assigned to the executing thread as it becomes available (because of **single**). When too many tasks get generated, OpenMP suspends the main thread, runs some tasks, then resumes the loop in the main thread.

Improved code. It would be better to **untied** the spawned tasks, enabling them to run on multiple threads. Surround the for loop with **#pragma omp task untied**.

About Nesting: Restrictions. Let's consider nesting of parallel constructs.

- You cannot nest **for** regions.
- You cannot nest **single** inside a **for**.
- You cannot nest **barrier** inside a **critical/single/master/for**.

Here's something that OpenMP does allow:

```

void good_nesting(int n)
{
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp parallel shared(i, n)
            {
                #pragma omp for
                for (j=0; j < n; j++)
                    work(i, j);
            }
        }
    }
}

```

```
    }
}
```

Why Your Code is Slow

Code too slow? Want it to run faster? Avoid these pitfalls:

1. Unnecessary flush directives.
2. Using critical sections or locks instead of atomic.
3. Unnecessary concurrent-memory-writing protection:
 - No need to protect local thread variables.
 - No need to protect if only accessed in **single** or **master**.
4. Too much work in a critical section.
5. Too many entries into critical sections.

```
#pragma omp parallel for
for (i = 0; i < N; ++i) {
    #pragma omp critical
    {
        if (arr[i] > max) max = arr[i];
    }
}
```

would be better as:

```
#pragma omp parallel for
for (i = 0 ; i < N; ++i) {
    #pragma omp flush(max)
    if (arr[i] > max) {
        #pragma omp critical
        {
            if (arr[i] > max) max = arr[i];
        }
    }
}
```

Memory Consistency, Memory Barriers, and Reordering

Today we'll talk a bit more about memory consistency, memory barriers and reordering in general. We'll start with instruction reordering by the CPU and move on to reordering initiated by the compiler. I'll also touch on some CPU instructions for atomic operations.

Memory Consistency. In a sequential program, you expect things to happen in the order that you wrote them. So, consider this code, where variables are initialized to 0:

```
T1: x = 1; r1 = y;  
T2: y = 1; r2 = x;
```

We would expect that we would always query the memory and get a state where some subset of these partially-ordered statements would have executed. This is the *sequentially consistent* memory model.

“... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” — Leslie Lamport

What are the possible values for the variables?

Another view of sequential consistency:

- each thread induces an *execution trace*.
- always, the program has executed some prefix of each thread's trace.

It turns out that sequential consistency is too expensive to implement. (Why?) So most systems actually implement weaker memory models, such that both **r1** and **r2** might end up unchanged. Recall the **flush** example from last time.

Reordering. Compilers and processors may reorder non-interfering memory operations within a thread. For instance, the two statements in T1 appear to be independent, so it's OK to execute

them—or, equivalently, to publish their results to other threads—in either order. Reordering is one of the major tools that compilers use to speed up code.

When is reordering a problem?

Memory Consistency Models

Here are some flavours of memory consistency models:

- Sequential consistency: no reordering of loads/stores.
- Relaxed consistency (only some types of reorderings):
 - Loads can be reordered after loads/stores; and
 - Stores can be reordered after loads/stores.
- Weak consistency: any reordering is possible.

In any case, **reorderings** only allowed if they look safe in current context (i.e. they reorder independent memory addresses). That can still be problematic, though.

Compilers and reordering. When it can prove that a reordering is safe with respect to the programming language semantics, the **compiler** may reorder instructions (so it's not just the hardware).

For example, say we want thread 1 to print value set in thread 2.

`f = 0`

<pre>/* thread 1 */ while (f == 0) /* spin */; printf("%d", x);</pre>	<pre>/* thread 2 */ x = 42; f = 1;</pre>
---	--

If thread 2 reorders its instructions, will we get our intended result? *No!*

Memory Barriers

We previously talked about OpenMP barriers: at a `#pragma omp barrier`, all threads pause, until all of the threads reach the barrier. Lots of OpenMP directives come with implicit barriers unless you add `nowait`.

A rather different type of barrier is a *memory barrier* or *fence*. This type of barrier prevents reordering, or, equivalently, ensures that memory operations become visible in the right order. A memory barrier ensures that no access occurring after the barrier becomes visible to the system, or takes effect, until after all accesses before the barrier become visible.

The x86 architecture defines the following types of memory barriers:

- **mfence**. All loads and stores before the barrier become visible before any loads and stores after the barrier become visible.
- **sfence**. All stores before the barrier become visible before all stores after the barrier become visible.
- **lfence**. All loads before the barrier become visible before all loads after the barrier become visible.

Note, however, that while an **sfence** makes the stores visible, another CPU will have to execute an **lfence** or **mfence** to read the stores in the right order.

Consider the example again:

`f = 0`

```
/* thread 1 */           /* thread 2 */
while (f == 0) /* spin */; x = 42;
// memory fence          // memory fence
printf("%d", x);          f = 1;
```

This now prevents reordering, and we get the expected result.

You can use the **mfence** instruction to implement *acquire barriers* and *release barriers*. An acquire barrier ensures that memory operations after a thread obtains the mutex doesn't become visible until after the thread actually obtains the mutex. The release barrier similarly ensures that accesses before the mutex release don't get reordered to after the mutex release. Note that it is safe to reorder accesses after the mutex release and put them before the release.

Preventing Memory Reordering in Programs: Compiler Barriers. First: Don't use volatile on variables ¹. However, you can prevent reordering using compiler-specific calls.

- Microsoft Visual Studio C++ Compiler:

```
_ReadWriteBarrier()
```

- Intel Compiler:

```
__memory_barrier()
```

- GNU Compiler:

```
__asm__ __volatile__ ("" ::: "memory");
```

The compiler also shouldn't reorder across e.g. Pthreads mutex calls.

¹<http://stackoverflow.com/questions/78172/using-c-pthreads-do-shared-variables-need-to-be-volatile>.

Aside: gcc Inline Assembly. Just as an aside, here's gcc's inline assembly format

```
--asm__ ( assembler template
        : output operands                /* optional */
        : input operands                 /* optional */
        : list of clobbered registers    /* optional */
        );
```

Note that we've just seen `--volatile--` with `--asm--`. This isn't the same as the normal C volatile. It means:

- The compiler may not reorder this assembly code and put it somewhere else in the program

Back to Memory Reordering in Programs. Fortunately, an OpenMP **flush** (or, better yet, **mutexes**) also preserve the order of variable accesses. Stops reordering from both the compiler and hardware. For GNU, flush is implemented as `--sync_synchronize()`;

volatile. This qualifier ensures that the code does an actual read from a variable every time it asks for one (i.e. the compiler can't optimize away the read). It does not prevent re-ordering nor does it protect against races.

Note: proper use of memory fences makes **volatile** not very useful (again, **volatile** is not meant to help with threading, and will have a different behaviour for threading on different compilers/hardware).

Atomic Operations

We saw the **atomic** directive in OpenMP. Most OpenMP atomic expressions map to atomic hardware instructions. However, other atomic instructions exist.

Compare and Swap. This operation is also called **compare and exchange** (implemented by the `cmpxchg` instruction on x86). Here's some pseudocode for it.

```
int compare_and_swap (int* reg, int oldval, int newval)
{
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    return old_reg_val;
}
```

Afterwards, you can check if the CAS returned `oldval`. If it did, you know you changed it.

Implementing a Spinlock. You can use compare-and-swap to implement spinlock:

```
void spinlock_init(int* l) { *l = 0; }

void spinlock_lock(int* l) {
    while(compare_and_swap(l, 0, 1) != 0) {}
    __asm__ ("mfence");
}

void spinlock_unlock(int* l) {
    __asm__ ("mfence");
    *l = 0;
}
```

You'll see **cmpxchg** quite frequently in the Linux kernel code.

ABA Problem

Sometimes you'll read a location twice. If the value is the same both times, nothing has changed, right?

No. This is an **ABA problem**.

You can combat this by “tagging”: modify value with nonce upon each write. You can also keep the value separately from the nonce; double compare and swap atomically swaps both value and nonce.

Just something to be aware of. “Not on exam”.

Good C++ Practice

Lots of people use postfix (**i++**) out of habit, but prefix (**++i**) is better.

In C, this isn't a problem. In some languages (like C++), it can be.

Why? Overloading. In C++, you can overload the **++** and **--** operators.

```
class X {
public:
    X& operator++();
    const X operator++(int);
    ...
};
```

```
X x;  
++x; // x.operator++();  
x++; // x.operator++(0);
```

Prefix is also known as **increment and fetch**, and might be implemented like this:

```
X& X::operator++()  
{  
    *this += 1;  
    return *this;  
}
```

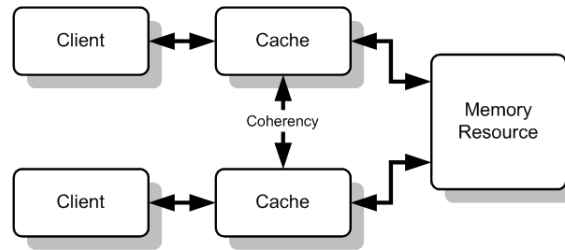
Postfix is also known as **fetch and increment**. Note that you have to make a copy of the old value:

```
const X X::operator++(int)  
{  
    const X old = *this;  
    ++(*this);  
    return old;  
}
```

So, if you're the least concerned about efficiency (and why else would you be taking programming for performance?), always use *prefix* increments/decrements instead of defaulting to postfix.

Only use **postfix** when you really mean it, to be on the safe side.

Cache Coherency



—Wikipedia

We talked about memory ordering and fences last time. Today we'll look at what support the architecture provides for memory ordering, in particular in the form of cache coherence. Since this isn't an architecture course, we'll look at this material more from the point of view of a user, not an implementer.

Cache coherence means that:

- the values in all caches are consistent; and
- to some extent, the system behaves as if all CPUs are using shared memory.

Cache Coherence Example. We will use this example to illustrate different cache coherence algorithms and how they handle the same situation.

Initially in main memory: $x = 7$.

1. CPU1 reads x , puts the value in its cache.
2. CPU3 reads x , puts the value in its cache.
3. CPU3 modifies $x := 42$
4. CPU1 reads $x \dots$ from its cache?
5. CPU2 reads x . Which value does it get?

Unless we do something, CPU1 is going to read invalid data.

High-Level Explanation of Snoopy Caches. The simplest way to “do something” is to use snoopy caches. The setup is as follows:

- Each CPU is connected to a simple bus.
- Each CPU “snoops” to observe if a memory location is read or written by another CPU.
- We need a cache controller for every CPU.

Then:

- Each CPU reads the bus to see if any memory operation is relevant. If it is, the controller takes appropriate action.

Write-Through Caches

Let’s put that into practice using write-through caches, the simplest type of cache coherence.

- All cache writes are done to main memory.
- All cache writes also appear on the bus.
- If another CPU snoops and sees it has the same location in its cache, it will either *invalidate* or *update* the data.
(We’ll be looking at invalidating.)

Normally, when you write to an invalidated location, you bypass the cache and go directly to memory (aka **write no-allocate**).

Write-Through Protocol. The protocol for implementing such caches looks like this. There are two possible states, **valid** and **invalid**, for each memory location. Events are either from a processor (**Pr**) or the **Bus**. We then implement the following state machine.

State	Observed	Generated	Next State
Valid	PrRd		Valid
Valid	PrWr	BusWr	Valid
Valid	BusWr		Invalid
Invalid	PrWr	BusWr	Invalid
Invalid	PrRd	BusRd	Valid

Example. For simplicity (this isn’t an architecture course), assume all cache reads/writes are atomic. Using the same example as before:

Initially in main memory: **x** = 7.

1. CPU1 reads **x**, puts the value in its cache. (valid)

2. CPU3 reads x, puts the value in its cache. (valid)
3. CPU3 modifies $x := 42$. (write to memory)
 - CPU1 snoops and marks data as invalid.
4. CPU1 reads x, from main memory. (valid)
5. CPU2 reads x, from main memory. (valid)

Write-Back Caches

Let's try to improve performance. What if, in our example, CPU3 writes to x 3 times?

Main goal. Delay the write to memory as long as possible. At minimum, we have to add a “dirty” bit, which indicates the our data has not yet been written to memory. Let's do that.

Write-Back Implementation. The simplest type of write-back protocol (MSI) uses 3 states instead of 2:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Shared**—location is unmodified, up-to-date with main memory; may be present in other caches (also up-to-date).
- **Invalid**—same as before.

The initial state for a memory location, upon its first read, is “shared”. The implementation will only write the data to memory if another processor requests it. During write-back, a processor may read the data from the bus.

MSI Protocol. Here, bus write-back (or flush) is **BusWB**. Exclusive read on the bus is **BusRdX**.

State	Observed	Generated	Next State
Modified	PrRd		Modified
Modified	PrWr		Modified
Modified	BusRd	BusWB	Shared
Modified	BusRdX	BusWB	Invalid
Shared	PrRd		Shared
Shared	BusRd		Shared
Shared	BusRdX		Invalid
Shared	PrWr	BusRdX	Modified
Invalid	PrRd	BusRd	Shared
Invalid	PrWr	BusRdX	Modified

MSI Example. Using the same example as before:

Initially in main memory: $x = 7$.

1. CPU1 reads x from memory. (BusRd, shared)
2. CPU3 reads x from memory. (BusRd, shared)
3. CPU3 modifies $x = 42$:
 - Generates a BusRdX.
 - CPU1 snoops and invalidates x .
4. CPU1 reads x :
 - Generates a BusRd.
 - CPU3 writes back the data and sets x to shared.
 - CPU1 reads the new value from the bus as shared.
5. CPU2 reads x from memory. (BusRd, shared)

An Extension to MSI: MESI

The most common protocol for cache coherence is MESI. This protocol adds yet another state:

- **Modified**—only this cache has a valid copy; main memory is **out-of-date**.
- **Exclusive**—only this cache has a valid copy; main memory is **up-to-date**.
- **Shared**—same as before.
- **Invalid**—same as before.

MESI allows a processor to modify data exclusive to it, without having to communicate with the bus. MESI is safe. The key is that if memory is in the E state, no other processor has the data.

MSEIF: Even More States!

MESIF (used in latest i7 processors):

- **Forward**—basically a shared state; but, current cache is the only one that will respond to a request to transfer the data.

Hence: a processor requesting data that is already shared or exclusive will only get one response transferring the data. This permits more efficient usage of the bus.

Cache coherence vs flush

Cache coherency seems to make sure my data is consistent. Why do I have to have something like flush or fence?

Sadly, no. Cache coherence isn't enough. Writes may be to registers rather than memory, and those won't be coherent. Use fences or flushes.

Well, I read that `volatile` variables aren't stored in registers, so then am I okay?

Again, sadly, no. Recall that `volatile` in C was only designed to:

- Allow access to memory mapped devices.
- Allow uses of variables between `setjmp` and `longjmp`.
- Allow uses of `sig_atomic_t` variables in signal handlers.

Remember, things can also be reordered by the compiler, and `volatile` doesn't prevent reordering. Also, it's likely your variables could be in registers the majority of the time, except in critical areas.

Coherence summary. We saw four cache coherence protocols, from MSI through MESIF. There are many other protocols for cache coherence, each with their own trade-offs.

Recall: OpenMP flush acts as a **memory barrier/fence** so the compiler and hardware don't reorder reads and writes.

- Neither cache coherence nor `volatile` will save you.

Building Servers: Concurrent Socket I/O

Switching gears, we'll talk about building software that handles tons of connections. From a Quora question:

What is the ideal design for server process in Linux that handles concurrent socket I/O?

So far in this class, we've seen:

- processes;
- threads; and
- thread pools.

We'll analyze the answer by Robert Love, Linux kernel hacker¹.

¹<https://plus.google.com/105706754763991756749/posts/VPMT8ucAcFH>

The Real Question.

How do you want to do I/O?

The question is not really “how many threads should I use?”.

Your Choices. The first two both use blocking I/O, while the second two use non-blocking I/O.

- Blocking I/O; 1 process per request.
- Blocking I/O; 1 thread per request.
- Asynchronous I/O, pool of threads, callbacks, each thread handles multiple connections.
- Nonblocking I/O, pool of threads, multiplexed with select/poll, event-driven, each thread handles multiple connections.

Blocking I/O; 1 process per request. This is the old Apache model.

- The main thread waits for connections.
- Upon connect, the main thread forks off a new process, which completely handles the connection.
- Each I/O request is blocking, e.g., reads wait until more data arrives.

Advantage:

- “Simple to understand and easy to program.”

Disadvantage:

- High overhead from starting 1000s of processes. (We can somewhat mitigate this using process pools).

This method can handle ~10 000 processes, but doesn’t generally scale beyond that, and uses many more resources than the alternatives.

Blocking I/O; 1 thread per request. We know that threads are more lightweight than processes. So let’s use threads instead of processes. Otherwise, this is the same as 1 process per request, but with less overhead. I/O is the same—it is still blocking.

Advantage:

- Still simple to understand and easy to program.

Disadvantages:

- Overhead still piles up, although less than processes.
- New complication: race conditions on shared data.

Asynchronous I/O. The other two choices don't assign one thread or process per connection, but instead multiplex the threads to connections. We'll first talk about using asynchronous I/O with select or poll.

Here are (from 2006) some performance benefits of using asynchronous I/O on lighttpd²:

version		fetches/sec	bytes/sec	CPU idle
1.4.13	sendfile	36.45	3.73e+06	16.43%
1.5.0	sendfile	40.51	4.14e+06	12.77%
1.5.0	linux-aio-sendfile	72.70	7.44e+06	46.11%

(Workload: 2×7200 RPM in RAID1, 1GB RAM, transferring 10GBytes on a 100MBit network).

The basic workflow is as follows:

1. enqueue a request;
2. ... do something else;
3. (if needed) periodically check whether request is done; and
4. read the return value.

Some code which uses the Linux asynchronous I/O model is:

```
#include <aio.h>

int main() {
    // so far, just like normal
    int file = open("blah.txt", ORDONLY, 0);

    // create buffer and control block
    char* buffer = new char[SIZE_TO_READ];
    aiocb cb;

    memset(&cb, 0, sizeof(aiocb));
    cb.aio_nbytes = SIZE_TO_READ;
    cb.aio_fildes = file;
    cb.aio_offset = 0;
    cb.aio_buf = buffer;

    // enqueue the read
    if (aio_read(&cb) == -1) { /* error handling */ }
```

²<http://blog.lighttpd.net/articles/2006/11/12/lighty-1-5-0-and-linux-aio/>

```

do {
    // ... do something else ...
    while ( aio_error(&cb) == EINPROGRESS); // poll

    // inspect the return value
    int numBytes = aio_return(&cb);
    if (numBytes == -1) { /* error handling */ }

    // clean up
    delete [] buffer;
    close( file );
}

```

Using Select/Poll. The idea is to improve performance by letting each thread handle multiple connections. When a thread is ready, it uses select/poll to find work:

- perhaps it needs to read from disk into a mmap'd tempfile;
- perhaps it needs to copy the tempfile to the network.

In either case, the thread does work and updates the request state.

Callback-Based Asynchronous I/O Model

Finally, we'll talk about a not-very-popular programming model for non-blocking I/O (at least for C programs; it's the only game in town for JavaScript and a contender for Go). Instead of select/poll, you pass a callback to the I/O routine, which is to be executed upon success or failure.

```

void new_connection_cb (int cfd)
{
    if (cfd < 0) {
        fprintf (stderr, "error_in_accepting_connection!\n");
        exit (1);
    }

    ref<connection_state> c =
        new refcounted<connection_state>(cfd);

    // what to do in case of failure: clean up.
    c->killing_task = delaycb(10, 0, wrap(&clean_up, c));

    // link to the next task: got the input from the connection
    fdcb (cfd, selread, wrap (&read_http_cb, cfd, c, true,
                             wrap(&read_req_complete_cb)));
}

```

node.js: A Superficial View. We'll wrap up today by talking about the callback-based `node.js` model. `node.js` is another event-based nonblocking I/O model. Given that JavaScript doesn't have threads, the only way to write servers is using non-blocking I/O.

The canonical example from the `node.js` homepage:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello␣World␣\n');
}).listen(1337, '127.0.0.1');
console.log('Server␣running␣at␣http://127.0.0.1:1337/');
```

Note the use of the callback—it's called upon each connection.

However, usually we don't want to handle the fields in the request manually. We'd prefer a higher-level view. One option is `expressjs`³, and here's an example from the Internet⁴:

```
app.post('/nod', function(req, res) {
  loadAccount(req, function(account) {
    if(account && account.username) {
      var n = new Nod();
      n.username = account.username;
      n.text = req.body.nod;
      n.date = new Date();
      n.save(function(err){
        res.redirect('/');
      });
    }
  });
});
```

³<http://expressjs.com>

⁴<https://github.com/tglines/nodrr/blob/master/controllers/nod.js>

1 Locking Granularity

Locks prevent data races.

However, using locks involves a trade-off between coarse-grained locking, which can significantly reduce opportunities for parallelism, and fine-grained locking, which requires more careful design, increases locking overhead and is more prone to bugs. Locks' extents constitute their **granularity**. In coarse-grained locking, you lock large sections of your program with a big lock; in fine-grained locking, you divide the locks and protect smaller sections.

We'll discuss three major concerns when using locks:

- overhead;
- contention; and
- deadlocks.

We aren't even talking about under-locking (i.e. remaining race conditions).

Lock Overhead. Using a lock isn't free. You pay:

- allocated memory for the locks;
- initialization and destruction time; and
- acquisition and release time.

These costs scale with the number of locks that you have.

Lock Contention. Most locking time is wasted waiting for the lock to become available. We can fix this by:

- making the locking regions smaller (more granular); or
- making more locks for independent sections.

Deadlocks. Finally, the more locks you have, the more you have to worry about deadlocks.

As you know, the key condition for a deadlock is waiting for a lock held by process X while holding a lock held by process X' . ($X = X'$ is allowed).

Consider, for instance, two processors trying to get two locks.

Thread 1	Thread 2
Get Lock 1	Get Lock 2
Get Lock 2	Get Lock 1
Release Lock 2	Release Lock 1
Release Lock 1	Release Lock 2

Processor 1 gets Lock 1, then Processor 2 gets Lock 2. Oops! They both wait for each other. (Deadlock!).

To avoid deadlocks, always be careful if your code **acquires a lock while holding one**. You have two choices: (1) ensure consistent ordering in acquiring locks; or (2) use trylock.

As an example of consistent ordering:

<pre>void f1() { lock(&l1); lock(&l2); // protected code unlock(&l2); unlock(&l1); }</pre>	<pre>void f2() { lock(&l1); lock(&l2); // protected code unlock(&l2); unlock(&l1); }</pre>
--	--

This code will not deadlock: you can only get **l2** if you have **l1**. Of course, it's harder to ensure a consistent deadlock when lock identity is not statically visible.

Alternately, you can use trylock. Recall that Pthreads' **trylock** returns 0 if it gets the lock. So, this code also won't deadlock: it will give up **l1** if it can't get **l2**.

```
void f1() {
    lock(&l1);
    while (trylock(&l2) != 0) {
        unlock(&l1);
        // wait
        lock(&l1);
    }
    // protected code
    unlock(&l2);
    unlock(&l1);
}
```

Coarse-Grained Locking

One way of avoiding problems due to locking is to use few locks (perhaps just one!). This is *coarse-grained locking*. It does have a couple of advantages:

- it is easier to implement;

- with one lock, there is no chance of deadlocking; and
- it has the lowest memory usage and setup time possible.

It also, however, has one big disadvantage in terms of programming for performance:

- your parallel program will quickly become sequential.

Example: Python (and other interpreters). Python puts a lock around the whole interpreter (known as the *global interpreter lock*). This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

Two major implications:

- The only performance benefit you'll see from threading is if a thread is waiting for IO.
- Any non-I/O-bound threaded program will be **slower** than the sequential version (plus, the interpreter will slow down your system).

Fine-Grained Locking

On the other end of the spectrum is *fine-grained locking*. The big advantage:

- it maximizes parallelization in your program.

However, it also comes with a number of disadvantages:

- if your program isn't very parallel, it'll be mostly wasted memory and setup time;
- plus, you're now prone to deadlocks; and
- fine-grained locking is generally more error-prone (be sure you grab the right lock!)

Examples. The Linux kernel used to have **one big lock** that essentially made the kernel sequential. This worked fine for single-processor systems! The introduction of symmetric multiprocessor systems (SMPs) required a more aggressive locking strategy, though, and the kernel now uses finer-grained locks for performance.

Databases may lock fields / records / tables. (fine-grained → coarse-grained).

You can also lock individual objects (but beware: sometimes you need transactional guarantees.)

Reentrancy versus Thread-Safety

On a different note, we're going to discuss the distinction between reentrant functions and thread-safe functions. There is overlap, but these terms are actually different.

A function is *reentrant* if it can be suspended in the middle and re-entered, or called again, before the previous execution returns.

Reentrant does not always mean **thread-safe** (although it usually is). Recall: **thread-safe** is essentially “no data races”.

The distinction is moot if the function only modifies local data, e.g. `sin()`. Those functions are both reentrant and thread-safe.

Example. Courtesy of Wikipedia (with modifications), here’s a program (to the left) and its trace (to right):

<pre> int t; void swap(int *x, int *y) { t = *x; *x = *y; // interrupt might invoke isr() here! *y = t; } void isr() { int x = 1, y = 2; swap(&x, &y); } ... int a = 3, b = 4; ... swap(&a, &b); </pre>	<pre> call swap(&a, &b); t = *x; // t = 3 (a) *x = *y; // a = 4 (b) call isr(); x = 1; y = 2; call swap(&x, &y) t = *x; // t = 1 (x) *x = *y; // x = 2 (y) *y = t; // y = 1 *y = t; // b = 1 Final values: a = 4, b = 1 Expected values: a = 4, b = 3 </pre>
---	--

We can fix the example by storing the global variable in stack variable `s`, as follows.

<pre> int t; void swap(int *x, int *y) { int s; s = t; // save global variable t = *x; *x = *y; // interrupt might invoke isr() here! *y = t; t = s; // restore global variable } void isr() { int x = 1, y = 2; swap(&x, &y); } ... int a = 3, b = 4; ... swap(&a, &b); </pre>	<pre> call swap(&a, &b); s = t; // s = UNDEFINED t = *x; // t = 3 (a) *x = *y; // a = 4 (b) call isr(); x = 1; y = 2; call swap(&x, &y) s = t; // s = 3 t = *x; // t = 1 (x) *x = *y; // x = 2 (y) *y = t; // y = 1 t = s; // t = 3 *y = t; // b = 3 t = s; // t = UNDEFINED Final values: a = 4, b = 3 Expected values: a = 4, b = 3 </pre>
--	--

The obvious question: is the previous reentrant code also thread-safe? (This is more what we’re concerned about in this course.)

Let's see. Consider two calls to the reentrant swap:

`swap(a, b), swap(c, d)` with `a = 1, b = 2, c = 3, d = 4`.

```
global: t

/* thread 1 */
a = 1, b = 2;
s = t;    // s = UNDEFINED
t = a;    // t = 1

/* thread 2 */
c = 3, d = 4;
s = t;    // s = 1
t = c;    // t = 3
c = d;    // c = 4
d = t;    // d = 3

a = b;    // a = 2
b = t;    // b = 3
t = s;    // t = UNDEFINED

t = s;    // t = 1

Final values:
a = 2, b = 3, c = 4, d = 3, t = 1

Expected values:
a = 2, b = 1, c = 4, d = 3, t = UNDEFINED
```

To recap what we know so far: re-entrant does not always mean thread-safe. (But, for most sane implementations, reentrant is also thread-safe.)

But, are **thread-safe** functions reentrant? Nope! Consider:

```
int f() {
    lock();
    // protected code
    unlock();
}
```

Recall: Reentrant functions can be suspended in the middle of execution and called again before the previous execution completes.

`f()` obviously isn't reentrant. Plus, it will deadlock.

Interrupt handling is more for systems programming, so the topic of reentrancy may or may not come up again.

To sum up, here's the difference between reentrant and thread-safe functions:

Reentrancy.

- Has nothing to do with threads—assumes a **single thread**.
- Reentrant means the execution can context switch at any point in in a function, call the **same function**, and **complete** before returning to the original function call.
- Function's result does not depend on where the context switch happens.

Thread-safety.

- Result does not depend on any interleaving of threads from concurrency or parallelism.
- No unexpected results from multiple concurrent executions of the function.

If it helps, here's another definition of thread-safety.

“A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another, in an undefined order, even if the actual execution is interleaved.”

Good Example of an Exam Question. Consider the following function.

```
void swap(int *x, int *y) {  
    int t;  
    t = *x;  
    *x = *y;  
    *y = t;  
}
```

- Is the above code thread-safe?
- Write some expected results for running two calls in parallel.
- Argue these expected results always hold, or show an example where they do not.

Good Programming Practices: Inlining

We have seen the notion of inlining:

- Instructs the compiler to just insert the function code in-place, instead of calling the function.
- Hence, no function call overhead!
- Compilers can also do better—context-sensitive—operations they couldn't have done before.

OK, so inlining removes overhead. Sounds like better performance! Let's inline everything! There are two ways of inlining in C++.

Implicit inlining. (defining a function inside a class definition):

```
class P {  
public:  
    int get_x() const { return x; }  
    ...  
private:  
    int x;  
};
```

Explicit inlining. Or, we can be explicit:

```
inline max(const int& x, const int& y) {  
    return x < y ? y : x;  
}
```

The Other Side of Inlining. Inlining has one big downside:

- Your program size is going to increase.

This is worse than you think:

- Fewer cache hits.
- More trips to memory.

Some inlines can grow very rapidly (C++ extended constructors). Just from this your performance may go down easily.

Note also that inlining is merely a suggestion to compilers. They may ignore you. For example:

- taking the address of an “inline” function and using it; or
- virtual functions (in C++),

will get you ignored quite fast.

Implications of inlining. Inlining can make your life worse in two ways. First, debugging is more difficult (e.g. you can’t set a breakpoint in a function that doesn’t actually exist). Most compilers simply won’t inline code with debugging symbols on. Some do, but typically it’s more of a pain.

Second, it can be a problem for library design:

- If you change any inline function in your library, any users of that library have to **recompile** their program if the library updates. (Congratulations, you made a non-binary-compatible change!)

This would not be a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Some Notes on Benchmarking

- Make sure your results are consistent (nothing else is running).
- Follow the 10 second guideline (60 second runs are no fun).

- Since we are assuming 100% parallel, the runtime should decrease by a factor of `physicalcores`.
- Results should be close to predicted, therefore our assumption holds (could estimate P in Amdahl's law and find it's 0.99).
- Overhead of threading (create, joining, mutex?) is insignificant for this program.
- Hyperthreading results were weird, slower the majority of the time.
- It's better to have a number of threads that match the number of virtual CPUs than an unbalanced number.
- If it's unbalanced, one thread will constantly be context switching between virtual CPUs.
- Worst case: 9 threads on 8 virtual CPUs. 8 threads complete, each doing a ninth of the work in parallel, last ninth of the work runs only on one CPU.

High-Level Language Performance Tweaks

So far, we've only seen C—we haven't seen anything complex, and C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard, especially when `#define` macros and `void *` beckon.

C++11 has made major strides towards readability and efficiency—it provides light-weight abstractions. We'll look at a couple of examples.

Sorting. Our goal is simple: we'd like to sort a bunch of integers. In C, you would usually just use `qsort` from `stdlib.h`.

```
void qsort (void* base, size_t num, size_t size,
           int (*comparator) (const void*, const void*));
```

This is a fairly ugly definition (as usual, for generic C functions). How ugly is it? Let's look at a usage example.

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*((int*)a) - (*((int*)b)));
}

int main(int argc, char* argv[])
{
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

This looks like a nightmare, and is more likely to have bugs than what we'll see next.

C++ has a sort with a much nicer interface¹:

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first ,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first ,
    RandomAccessIterator last ,
    Compare comp
);
```

It is, in fact, easier to use:

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

Note: Your compare function can be a function or a functor. (Don't know what functors are? In C++, they're functions with state.) By default, `sort` uses `operator<` on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

The second question is empirical. Let's see. We generate an array of 2 million ints and sort it (10 times, taking the average).

- `qsort`: 0.49 seconds
- C++ `sort`: 0.21 seconds

The C++ version is **twice** as fast. Why?

- The C version just operates on memory—it has no clue about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

- Custom C sort: 0.29 seconds

¹... well, nicer to use, after you get over templates.

Now the C++ version is still faster (but it's close). But, this is quickly going to become a maintainability nightmare.

- Would you rather read a custom sort or 1 line?
- What (who) do you trust more?

Lesson

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

Vectors vs Lists

Consider two problems.

1. Generate **N** random integers and insert them into (sorted) sequence.

Example: 3 4 2 1

- 3
- 3 4
- 2 3 4
- 1 2 3 4

2. Remove **N** elements one-at-a-time by going to a random position and removing the element.

Example: 2 0 1 0

- 1 2 4
- 2 4
- 2
-

For which **N** is it better to use a list than a vector (or array)?

Complexity analysis. As good computer scientists, let's analyze the complexity.

Vector:

- Inserting
 - $O(\log n)$ for binary search
 - $O(n)$ for insertion (on average, move half the elements)
- Removing
 - $O(1)$ for accessing
 - $O(n)$ for deletion (on average, move half the elements)

List:

- Inserting
 - $O(n)$ for linear search
 - $O(1)$ for insertion
- Removing
 - $O(n)$ for accessing
 - $O(1)$ for deletion

Therefore, based on their complexity, lists should be better.

Reality. OK, here's what happens.

```
$ ./vector_vs_list 50000
Test 1
=====
vector: insert 0.1s   remove 0.1s   total 0.2s
list:   insert 19.44s remove 5.93s   total 25.37s
Test 2
=====
vector: insert 0.11s  remove 0.11s  total 0.22s
list:   insert 19.7s  remove 5.93s  total 25.63s
Test 3
=====
vector: insert 0.11s  remove 0.1s   total 0.21s
list:   insert 19.59s remove 5.9s    total 25.49s
```

Vectors dominate lists, performance wise. Why?

- Binary search vs. linear search complexity dominates.
- Lists use far more memory. **On 64 bit machines:**
 - Vector: 4 bytes per element.
 - List: At least 20 bytes per element.

- Memory access is slow, and results arrive in blocks:
 - Lists' elements are all over memory, hence many cache misses.
 - A cache miss for a vector will bring a lot more usable data.

So, here are some tips for getting better performance.

- Don't store unnecessary data in your program.
- Keep your data as compact as possible.
- Access memory in a predictable manner.
- Use vectors instead of lists by default.
- Programming abstractly can save a lot of time.
- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient.**
- Think at a low level if you need to optimize anything.
- Readable code is good code—different hardware needs different optimizations.

Summary

In this lecture, we saw:

- Fine vs. Coarse-Grained locking tradeoffs.
- Ways to prevent deadlocks.
- Difference between reentrant and thread-safe functions.
- Limit your inlining to trivial functions:
 - makes debugging easier and improves usability;
 - won't slow down your program before you even start optimizing it.
- Tell the compiler high-level information but think low-level.