

Lecture 01—Introduction

ECE 459: Programming for Performance

Patrick Lam

University of Waterloo

January 8, 2013

[Thanks to Jon Eyolfson for slides!]

Course Website

<http://patricklam.ca/p4p/>

Staff

Instructor

Patrick Lam p.lam@ece.uwaterloo.ca DC 2597D/DC2534

Teaching Assistants

Xavier Noumbissi xnouumbis@uwaterloo.ca DC 2553

Morteza Nabavi mnabavi@uwaterloo.ca E5-4131

Schedule

Lectures: January 8—April 4, TTh, 8:30 AM, RCH 103

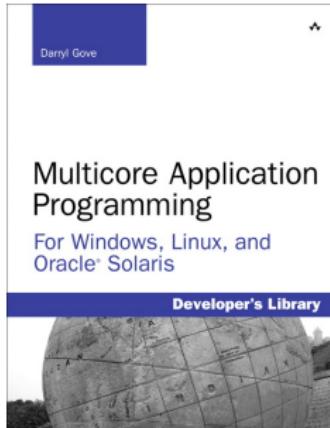
Tutorials: January 11—April 5, F, 4:30 PM, RCH 103

Midterm: February 27, W, 7:00 PM—8:20PM, RCH 301/309

Office Hours

- Suggestions?

Recommended Textbook



Multicore Application Programming For Windows, Linux, and Oracle Solaris. Darryl Gove. Addison-Wesley, 2010.

Goal

Make programs run faster!

Making Programs Faster

Two main ways:

- Increase bandwidth (tasks per unit time); or
- Decrease latency (time per task).

Examples of bandwidth/latency:

Network (connection speed/ping), traffic (lanes/speed)

Our Focus

Primarily on increasing bandwidth (more tasks/unit time).

- Do tasks in parallel

Decreasing time/task usually harder, with fewer gains.

CPUs have been going towards more cores rather than raw speed.

A Bit on Improving Latency

We won't return to these topics, but we'll touch on them now.

- Profile the code;
- Do less work;
- Be smarter; or
- Improve the hardware.

Increasing Bandwidth: Parallelism

Some tasks are easy to run in parallel.

Examples: web server requests, computer graphics, brute-force searches, genetic algorithms

Others are more difficult.

Example: linked list traversal (why?)

Hardware

- Use pipelining (all modern CPU do this):
 - ▶ Implement this in software by splitting a task into subtasks and running the subtasks in parallel
- Increase the number of cores/CPUs.
- Use multiple connected machines.
- Use specialized hardware, such as a GPU which contains hundreds of simple cores.

Barriers to parallelization

- Independent tasks (“embarrassingly parallel problems”) are trivial to parallelize, but dependencies cause problems.
- Unable to start task until previous task finishes.
- May require synchronization and combination of results.
- More difficult to reason about, since execution may happen in any order.

Limitations

- Sequential tasks in the problem will always dominate maximum performance
- Some sequential problems may be parallelizable by reformulating the implementation
- However, no matter how many processors you have, you won't be able to speed up the program as a whole (known as **Amdahl's Law**)

Data Race

- Two processors accessing the same data.
- For example, consider the following code:

```
x = 1  
print x
```

You run it and see it prints 5

- **Why?** Before the print, another thread wrote a new value for x. This is an example of a data race.

Deadlock

Two processors trying to access a shared resource.

- Consider two processors trying to get two resources:

Processor 1

Get Resource 1

Get Resource 2

Release Resource 2

Release Resource 1

Processor 2

Get Resource 2

Get Resource 1

Release Resource 1

Release Resource 2

- Processor 1 gets Resource 1, then Processor 2 gets Resource 2, now they both wait for each other (**deadlock**).

Objectives

- Implement parallel programming involving synchronization
- Describe and use parallel computing frameworks
- Ability to investigate software and improve its performance
- Use and understand specialized GPU programming/programming languages

Assignments

- ① Manual parallelization using Pthreads
- ② Automatic parallelization and OpenMP
- ③ Application profiling and improvement (groups of 2)
- ④ GPU programming

Breakdown

- 40% Assignments (10% each)
- 10% Midterm
- 50% Final

Grace Days

- 4 grace days to use over the semester for late assignments
- **No mark penalty** for using grace days
- Try not to use them just because they're there

Suggestions?

- Just let me know

Lecture 02—Amdahl's Law, Modern Hardware

ECE 459: Programming for Performance

Patrick Lam

University of Waterloo

January 10, 2013

Limitations

Our main focus is parallelization.

- Most programs have a sequential part and a parallel part; and,
- Amdahl's Law answers, "what are the limits to parallelization?"

Formulation (1)

S : fraction of serial runtime in a serial execution.

P : fraction of parallel runtime in a serial execution.

Therefore, $S + P = 1$.

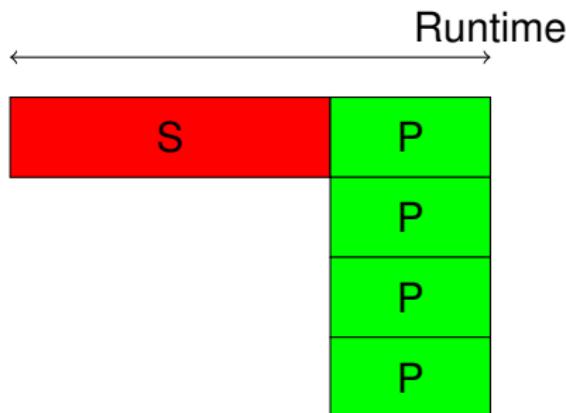
With 4 processors, best case, what can happen to the following runtime?



Formulation (1)



We want to split up the parallel part over 4 processors



Formulation (2)

T_s : time for the program to run in serial

N : number of processors/parallel executions

T_p : time for the program to run in parallel

- Under perfect conditions, get N speedup for P

$$T_p = T_s \cdot \left(S + \frac{P}{N} \right)$$

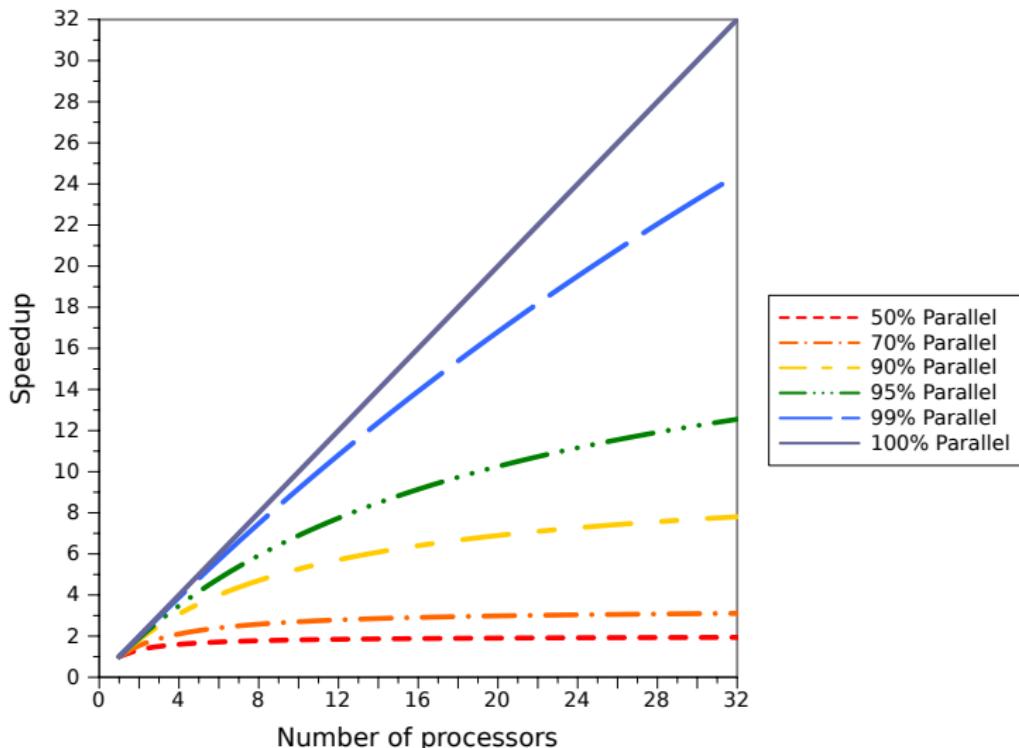
Formulation (3)

How much faster can we make the program?

$$\begin{aligned} \text{speedup} &= \frac{T_s}{T_p} \\ &= \frac{T_s}{T_s \cdot (S + \frac{P}{N})} \\ &= \frac{1}{S + \frac{P}{N}} \end{aligned}$$

(assuming no overhead for parallelizing; or costs near zero)

Scaling with Fraction of Parallel Code



Amdahl's Law

Replace S with $(1 - P)$:

$$\text{speedup} = \frac{1}{(1-P) + \frac{P}{N}}$$

$$\text{maximum speedup} = \frac{1}{(1-P)}, \text{ since } \frac{P}{N} \rightarrow 0$$

As you might imagine, the asymptotes in the previous graph are bounded by the maximum speedup.

Amdahl's Law Generalization

The program may have many parts, each of which we can tune to a different degree.

Let's generalize Amdahl's Law.

f_1, f_2, \dots, f_n : fraction of time in part n

$S_{f_1}, S_{f_2}, \dots, S_{f_n}$: speedup for part n

$$\text{speedup} = \frac{1}{\frac{f_1}{S_{f_1}} + \frac{f_2}{S_{f_2}} + \dots + \frac{f_n}{S_{f_n}}}$$

Application (1)

Consider a program with 4 parts in the following scenario:

Part	Fraction of Runtime	Speedup	
		Option 1	Option 2
1	0.55	1	2
2	0.25	5	1
3	0.15	3	1
4	0.05	10	1

We can implement either Option 1 or Option 2.
Which option is better?

Application (2)

“Plug and chug” the numbers:

Option 1

$$\text{speedup} = \frac{1}{0.55 + \frac{0.25}{5} + \frac{0.15}{3} + \frac{0.05}{5}} = 1.53$$

Option 2

$$\text{speedup} = \frac{1}{\frac{0.55}{2} + 0.45} = 1.38$$

Empirically estimating parallel speedup P

Useful to know, don't have to commit to memory:

$$P_{\text{estimated}} = \frac{\frac{1}{\text{speedup}} - 1}{\frac{1}{N} - 1}$$

- Quick way to guess the fraction of parallel code
- Use $P_{\text{estimated}}$ to predict speedup for a different number of processors

Another Example

We run a program in serial and find it spends 12.5% of its execution on serial code and 87.5% on parallel code. How many processors do we need to get within 10% of the perfect parallel runtime?

Summary of Amdahl's Law

Important to focus on the part of the program with most impact.

Amdahl's Law:

- estimates perfect performance gains from parallelization; but,
- only applies to solving a **fixed problem size** in the shortest possible period of time

Gustafson's Law: Formulation

n : problem size

$S(n)$: fraction of serial runtime for a parallel execution

$P(n)$: fraction of parallel runtime for a parallel execution

$$T_p = S(n) + P(n) = 1$$

$$T_s = S(n) + N \cdot P(n)$$

$$\text{speedup} = \frac{T_s}{T_p}$$

Gustafson's Law

$$\text{speedup} = S(n) + N \cdot P(n)$$

Assuming the fraction of runtime in serial part decreases as n increases, the speedup approaches N .

- Yes! Large problems can be efficiently parallelized.
(Ask Google.)

Driving Metaphor

Amdahl's Law

Suppose you're travelling between 2 cities 90 km apart. If you travel for an hour at a constant speed less than 90 km/h, your average will never equal 90 km/h, even if you energize at your destination.

Gustafson's Law

Suppose you've been travelling at a constant speed less than 90 km/h. Given enough distance, you can bring your average up to 90 km/h.

Lecture 03—Threading I

ECE 459: Programming for Performance

Patrick Lam

University of Waterloo

January 15, 2013

Parallelism versus Concurrency

Parallelism

Two or more tasks are parallel if they are running at the same time.

Main goal: run tasks as fast as possible.

Main concern: dependencies.

Concurrency

Two or more tasks are concurrent if the ordering of the two tasks is not predetermined.

Main concern: synchronization.

Threads



- What are they?
- How do operating systems implement them?
- How can we leverage them?

Processes versus Threads

Process

An instance of a computer program that contains program code and its:

- Own address space / virtual memory;
- Own stack / registers;
- Own resources (file handles, etc.).

Thread

“Lightweight processes”.

In most cases, a thread is contained within a process.

- Same address space as parent process
 - ▶ Shares access to code and variables with parent.
- Own stack / registers
- Own thread-specific data

Software and Hardware Threads

Software Thread:

What you program with.

Corresponds to a stream of instructions executed by the processor.

On a single-core, single-processor machine, someone has to multiplex the CPU to execute multiple threads concurrently; only one thread runs at a time.

Hardware Thread:

Corresponds to virtual (or real) CPUs in a system. Also known as strands.

Operating system must multiplex software threads onto hardware threads, but can execute more than one software thread at once.

Thread Model—1:1 (Kernel-level Threading)

Simplest possible threading implementation.

The kernel schedules threads on different processors;

- NB: Kernel involvement required to take advantage of a multicore system.

Context switching involves system call overhead.

Used by Win32, POSIX threads for Windows and Linux.

Allows concurrency and parallelism.

Thread Model—N:1 (User-level Threading)

All application threads map to a single kernel thread.

Quick context switches, no need for system call.

Cannot use multiple processors, only for concurrency.

- Why would you use user threads?

Used by GNU Portable Threads.

Thread Model—M:N (Hybrid Threading)

Map M application threads to N kernel threads.

A compromise between the previous two models.

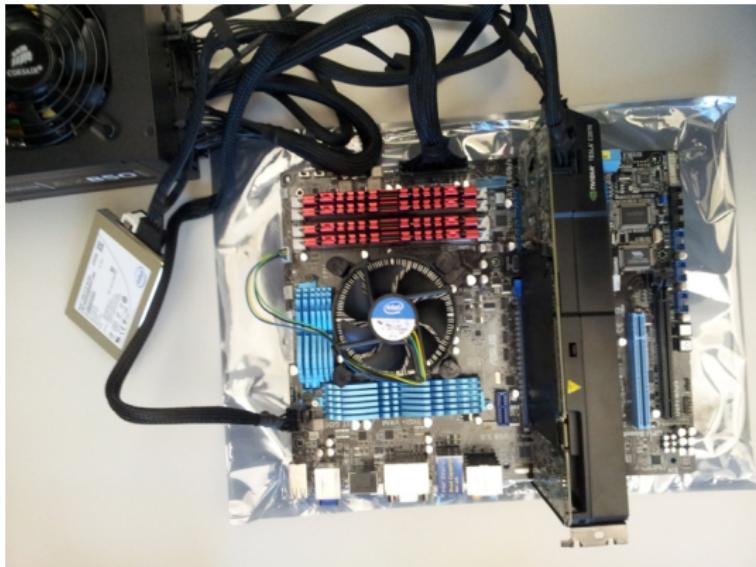
Allows quick context switches and the use of multiple processors.

Requires increased complexity; library provides scheduling.

- May not coordinate well with kernel.
- Increases likelihood of priority inversion (recall from Operating Systems).

Used by modern Windows threads.

Example System—Physical View



- Only one physical CPU

Example System—System View

```
jon@ece459-1 ~ % egrep 'processor|model name' /proc/cpuinfo
processor : 0
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 1
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 2
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 3
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 4
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 5
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 6
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
processor : 7
model name: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
```

- Many processors

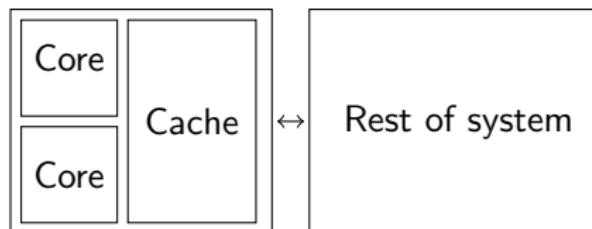
SMP (Symmetric Multiprocessing)

Identical processors or cores, which:

- Are interconnected using buses or another type of communication; and
- Share main memory.

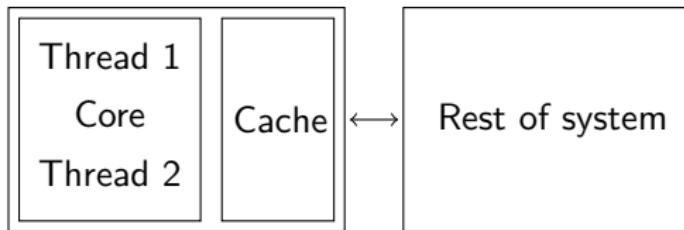
Most common type of multiprocessing system

Example of an SMP System



- Each core can execute a different thread
- Shared memory quickly becomes the bottleneck

Executing 2 Threads on a Single Core



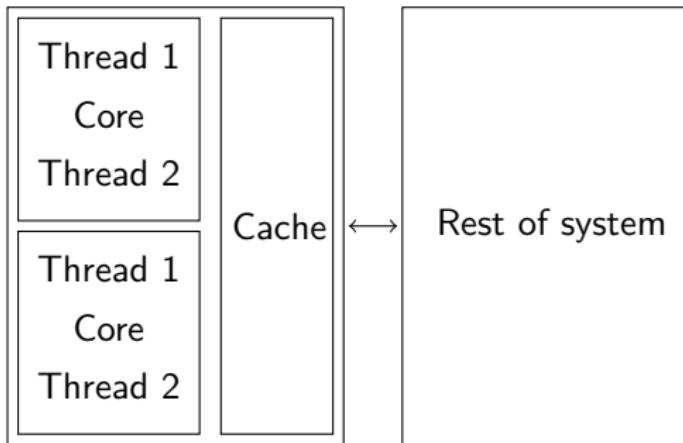
On a single core, must context switch between threads:

- every N cycles; or
- wait until cache miss, or another long event

Resources may be unused during execution.

Why not take advantage of this?

Executing M Threads on a N Cores



Here's a Chip Multithreading example.

UltraSPARC T2 has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

SMT (Simultaneous Multithreading)

Use idle CPU resources (may be calculating or waiting for memory) to execute another task.

Cannot improve performance if shared resources are the bottleneck.

Issue instructions for each thread per cycle.

To the OS, it looks a lot like SMP, but gives only up to 30% performance improvement.

Intel implementation: Hyper-Threading.

Example: Non-SMP system



PlayStation 3 contains a Cell processor:

- PowerPC main core (Power Processing Element, or “PPE”)
- 7 Synergistic Processing Elements (“SPE”s): small vector computers.

NUMA (Non-Uniform Memory Access)

In SMP, all CPUs have uniform (the same) access time for resources.

For NUMA, CPUs can access different resources faster (resources: not just memory).

Schedule tasks on CPUs which access resources faster.

Since memory is commonly the bottleneck, each CPU has its own memory bank.

Processor Affinity

Each task (process/thread) can be associated with a set of processors.

Useful to take advantage of existing caches (either from the last time the task ran or task uses the same data).

Hyper-Threading is an example of complete affinity for both threads on the same core.

Often better to use a different processor if current set is busy.

Lecture 04—Pthreads and Simple Locks (v2)

ECE 459: Programming for Performance

January 17, 2013

Background

Recall the difference between `processes` and `threads`:

- 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)

History

`clone` is not POSIX compliant.

Developers mostly used `fork` in the past, which creates a new process.

- Drawbacks?
- Benefits?

Benefit: 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, uses 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.

Drawback of Processes: Threads are Easier and Faster

- Interprocess communication (IPC) is more complicated and slower than interthread communication.
 - ▶ Need to use operating system utilities (pipes, semaphores, shared memory, etc) instead of thread library (or just memory reads and writes).
- Processes have much higher startup, shutdown and synchronization cost.
- And, [Pthreads](#) fix the issues of clone and provide a uniform interface for most systems (**focus of Assignment 1**).

Appropriate Time to Use Processes

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.
- Want to be safer against bugs and security holes.

Then processes are the way to go.

For performance reasons, along with ease and consistency, we'll use [Pthreads](#).

fork Usage Example (OS refresher)

```
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, which starts execution 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.

Threads Offer a Speedup of 6.5 over fork

Here's a benchmark between `fork` and Pthreads on a laptop, creating and destroying 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`.

Assumptions

First, we'll see 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 can be parallelized ([dependencies](#))
- alternative parallelization patterns
(right now, just use one thread per sub-problem)

POSIX Threads

- Available on most systems
- Windows has Pthreads Win32, but I wouldn't use it; use Linux for this course
- API available by `#include <pthread.h>`
- Compile with pthread flag
`(gcc -pthread prog.c -o prog)`

Creating Threads

```
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

returns 0 on success, error number otherwise
(contents of *thread are undefined)

Creating Threads—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");
}
```

Simply creates a thread and terminates
(usage isn't really right, as we'll see.)

Waiting for Threads

```
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.

returns 0 on success, error number otherwise.

Only call this one time per thread! Multiple calls on the same thread leads to undefined behaviour.

Waiting for Threads—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);
}
```

This now waits for the newly created thread to terminate.

Passing Data to Threads... Wrongly

Consider this snippet:

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

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

Passing Data to Threads... Wrongly

Consider this snippet:

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

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

- ➊ The value of *i* will probably change before the thread executes
- ➋ The memory for *i* may be out of scope, and therefore invalid by the time the thread executes

Passing Data to Threads

What about:

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

This is suggested in the book, but should carry a warning:

Passing Data to Threads

What about:

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

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

This is suggested in the book, but should carry a warning:

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

Detached Threads

Joinable threads (the default) wait for someone to call `pthread_join` before they release their resources.

Detached threads release their resources when they terminate, without being joined.

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

thread: marks the thread as detached

returns 0 on success, error number otherwise.

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

Thread Termination

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

retval: return value passed to function that calls
pthread_join

start_routine returning is equivalent to calling pthread_exit
with that return value;

pthread_exit is called implicitly when the start_routine of
a thread returns.

Attributes

By default, threads are *joinable* on Linux, but a more portable way to know what you're getting is to set thread attributes.

You can change:

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

Attributes—Example

```
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
```

Setting a thread state to joinable:

```
pthread_attr_setdetachstate(&attributes,
                           PTHREAD_CREATE_JOINABLE);
```

Detached Threads: Warning!

```
#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?

Detached Threads: Solution to Problem

```
#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
}
```

Make the final call `pthread_exit` if you have any detached threads.

Three Useful Routines

```
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.

Use `pthread_equal` if you're comparing 2 threads.

If you want to run a section of code once, you need `pthread_once` (it's well-named). It will run only once per `once_control`.

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.
- glibc reentrant functions are also safe: a program can have more than one thread calling these functions concurrently.
- **Example:** In Assignment 1, we'll use `rand_r`, not `rand`.

Mutual Exclusion

Mutexes are the most basic type of synchronization.

- 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.

Creating Mutexes—Example

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;  
pthread_mutex_t m2;  
  
pthread_mutex_init(&m2, NULL);  
...  
pthread_mutex_destroy(&m1);  
pthread_mutex_destroy(&m2);
```

- Two ways to initialize mutexes: statically and dynamically
- If you want to include attributes, you need to use the dynamic version

Mutex Attributes

- **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.

Using Mutexes: Example

```
// 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.
- Also you can use `pthread_mutex_trylock`, if needed.

Data Race Example

Recall that **dataraces** occur when two concurrent actions access the same variable and at least one of them is a **write**

```
...
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?

Example Problem Solution

```
...
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);
}
```

Lecture 05—C Compiler Features, Race Conditions, More Synchronization (v2)

January 22, 2013

Previously

- How to choose: Multiple processes or threads?
- Creating, joining and exiting POSIX threads.
Remember, they are 1:1 with kernel threads and can run in parallel on multiple CPUs.
- Difference between **joinable** and **detached** threads.
- Using mutual exclusion.

Part I

Making C Compilers Work For You

Three Address Code

- An intermediate code used by compilers for analysis and optimization.
- Statements represent one fundamental operation (we can consider each operation **atomic**).
- Statements have the form:
$$result := operand_1 \ operator \ operand_2$$
- Useful for reasoning about data races and easier to read than assembly (separates out memory reads/writes).

GIMPLE

- GIMPLE is the three address code used by gcc.
- To see the GIMPLE representation of your code use the `-fdump-tree-gimple` flag.
- To 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.
- Use GIMPLE to reason about your code at a low level without having to read assembly.

volatile Keyword

- Used to notify the compiler that the variable may be changed by “external forces”. For instance,

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

volatile prevents this from being optimized to:

```
int i = 0;  
  
while (true) {  
    ...
```

- Variable will not actually be volatile in the critical section and only prevents useful optimizations.
- Usually wrong unless there is a **very** good reason for it.

The restrict Keyword

A new feature of 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.

Example of restrict (1)

Pointers declared with `restrict` must never point to the same data.

From Wikipedia:

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

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

Example of restrict (2)

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 }
```

- Could any operation be left out if all the pointers didn't overlap?

Example of restrict (3)

```
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 }
```

- 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
`updatePtrs(&x, &y, &x);`

Example of restrict (4)

Hence, this markup 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.

⇒ compiler can better optimize your code (more perf!)

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.

Part II

Race Conditions

Race Conditions

- 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.

Race conditions arise between variables which are shared between threads.

Example Data Race (Part 1)

```
#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;
}
```

Example Data Race (Part 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;
}
```

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

Example Data Race (Part 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;
}
```

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

- No, we don't. Only one thread is active at a time.

Example Data Race (Part 2B)

```
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;
}
```

Do we have a data race now? Why or why not?

Example Data Race (Part 2B)

```
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;
}
```

Do we have a data race now? Why or why not?

- Yes, we do. We have 2 threads concurrently accessing the same data.

Tracing our Example Data Race

What are the possible outputs? (initially $*x$ is 1).

1	run1	run2
2	D.1 = *x;	D.1 = *x;
3	D.2 = D.1 + 1;	D.2 = D.1 + 2
4	*x = D.2;	

- Memory reads and writes are key in data races.

Outcome of Example Data Race

- Let's call the read and write from run1 R1 and W1; R2 and W2 from run2.
- Assuming a sane¹ memory model, R_n must precede W_n .

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

¹sequentially consistent

Detecting Data Races Automatically

Dynamic and static tools can help 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 with debugging information (-g flag for gcc).

Helgrind Output for Example

```
==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)
```

Part III

More Synchronization

Mutexes Recap

Our focus is on how to use mutexes correctly:

- Call `lock` on mutex `m1`. Upon return from `lock`, you have exclusive access to `m1` until you `unlock` it.
- Other calls to `lock m1` will not return until `m1` is available.

For background on selection algorithms, look at Lamport's bakery algorithm.

(Not in scope for this course.)

More on Mutexes

Can also “try-lock”: grab lock if available, else return to caller (and do something else).

Excessive use of locks can serialize programs.

- Linux kernel used to rely on a Big Kernel Lock protecting lots of resources in the 2.0 era.
- Linux 2.2 improved performance on SMPs by cutting down on the use of the BKL.

Note: in Windows, “mutex” is an inter-process communication mechanism. Windows “critical sections” are our mutexes.

Spinlocks

Functionally equivalent to mutex.

- `pthread_spinlock_t`, `pthread_spin_lock`,
`pthread_spin_trylock` and friends

Implementation difference: spinlocks will repeatedly try the lock and will not put the thread to sleep.

Good if your protected code is short.

Mutexes may be implemented as a combination between spinning and sleeping (spin for a short time, then sleep).

Read-Write Locks

Two observations:

- If there are only reads, there's no datarace.
- Often, writes are relatively rare.

With mutexes/spinlocks, you have to lock the data, even for a read, since a write could happen.

But, most of the time, reads can happen in parallel, as long as there's no write.

Solution: Multiple threads can hold a read lock (`pthread_rwlock_rdlock`), but only one thread may hold the associated write lock (`pthread_rwlock_wrlock`); it waits until current readers are done.

Semaphores

Semaphores have a value. You specify initial value.

Semaphores allow sharing of a # of instances of a resource.

Two fundamental operations: wait and post.

- wait is like lock; reserves the resource and decrements the value.
 - ▶ If value is 0, sleep until value is greater than 0.
- post is like unlock; releases the resource and increments the value.

Barriers

Allows you to ensure that (some subset of) a collection of threads all reach the barrier before finishing.

Pthreads: A barrier is a `pthread_barrier_t`.

Functions: `_init()` (parameter: how many threads the barrier should wait for) and `_destroy()`.

Also `_wait()`: similar to `pthread_join()`, but waits for the specified number of threads to arrive at the barrier

Lock-Free Algorithms

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.

Lock-free implementations are extremely complicated and must still contain certain synchronization constructs.

Semaphores Usage

```
#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);
```

- Also 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 Usage

```
#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);
```

- Also 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?

- If the initial value is 1 and you use `wait` to lock and `post` to unlock, it's equivalent to 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;
}
```

Semaphores for Signalling

Here's their 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 );
}
```

Semaphores for Signalling

- ① value is initially 1.
- ② Say p2 hits its `sem_wait` first and succeeds.
- ③ value is now 0 and p2 prints “Thread 2” first.
 - If p1 happens first, it would just increase value to 2.

Semaphores for Signalling

- ① value is initially 1.
- ② Say p2 hits its `sem_wait` first and succeeds.
 - ③ value is now 0 and p2 prints “Thread 2” first.
 - If p1 happens first, it 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 (and prints “Thread 1”) first.

Lecture 06—Dependencies

ECE 459: Programming for Performance

January 24, 2013

Last Time

Three-address code, volatile and restrict.

Race conditions:

- detecting them with helgrind.
- avoiding them via synchronization: mutexes, spinlocks, read-write locks, semaphores, barriers, lock-free algorithms.

This time: Dependencies

Dependencies are the main limitation to parallelization.

Example: computation must be evaluated as XY and not YX.

Not synchronization

Assume (for now) no synchronization problems.

Only trying to identify code that is safe to run in parallel.

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42  
x = x + 1
```

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42  
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42  
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?

$x = 43$ or $x = 42$

Next, we'll classify dependencies.

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
```

```
z = x + 5
```

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1  
z = x + 5
```

Yes.

- Variables y and z are independent.
- Variable x is only read.

RAR dependency allows parallelization.

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37  
z = x + 5
```

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37  
z = x + 5
```

No, $z = 42$ or $z = 7$.

RAW inhibits parallelization: can't change ordering.
Also known as a **true dependency**.

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
```

```
x = 37
```

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5  
x = 37
```

No. Again, $z = 42$ or $z = 7$.

- WAR is also known as a [anti-dependency](#).
- But, we can modify this code to enable parallelization.

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x  
z = x_copy + 5  
x = 37
```

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x  
z = x_copy + 5  
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x  
z = x_copy + 5  
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Not always:

```
z = very_long_function(x) + 5  
x = very_long_calculation()
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5  
z = x + 40
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5  
z = x + 40
```

Nope, $z = 42$ or $z = 7$.

- WAW is also known as an [output dependency](#).
- We can remove this dependency (like WAR):

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5  
z = x + 40
```

Nope, $z = 42$ or $z = 7$.

- WAW is also known as an [output dependency](#).
- We can remove this dependency (like WAR):

```
z_copy = x + 5  
z = x + 40
```

Summary of Memory-carried Dependencies

		Second Access	
		Read	Write
First Access	Read	No Dependency Read After Read (RAR)	Anti-dependency Write After Read (WAR)
	Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)

Part II

Loop-carried Dependencies

Loop-carried Dependencies (1)

Can we run these lines in parallel?
(initially $a[0]$ and $a[1]$ are 1)

```
a[4] = a[0] + 1  
a[5] = a[1] + 2
```

Loop-carried Dependencies (1)

Can we run these lines in parallel?
(initially $a[0]$ and $a[1]$ are 1)

```
a[4] = a[0] + 1  
a[5] = a[1] + 2
```

Yes.

- There are no dependencies between these lines.
- However, this is not how we normally use arrays...

Loop-carried Dependencies (2)

What about this? (all elements initially 1)

```
for (int i = 1; i < 12; ++i)
    a[i] = a[i - 1] + 1
```

Loop-carried Dependencies (2)

What about this? (all elements initially 1)

```
for (int i = 1; i < 12; ++i)
    a[i] = a[i - 1] + 1
```

No, $a[2] = 3$ or $a[2] = 2$.

- Statements depend on previous loop iterations.
- An example of a **loop-carried dependency**.

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)  
    a[i] = a[i-4] + 1
```

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)  
    a[i] = a[i-4] + 1
```

Yes, to a degree.

- We can execute 4 statements in parallel:
 - ▶ $a[4] = a[0] + 1, a[8] = a[4] + 1$
 - ▶ $a[5] = a[1] + 1, a[9] = a[5] + 1$
 - ▶ $a[6] = a[2] + 1, a[10] = a[6] + 1$
 - ▶ $a[7] = a[3] + 1, a[11] = a[7] + 1$

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)  
    a[i] = a[i-4] + 1
```

Yes, to a degree.

- We can execute 4 statements in parallel:

- ▶ $a[4] = a[0] + 1, a[8] = a[4] + 1$
- ▶ $a[5] = a[1] + 1, a[9] = a[5] + 1$
- ▶ $a[6] = a[2] + 1, a[10] = a[6] + 1$
- ▶ $a[7] = a[3] + 1, a[11] = a[7] + 1$

Always consider dependencies between iterations.

Larger example: Loop-carried Dependencies

```
// 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;
}
```

How can we parallelize this?

Larger example: Loop-carried Dependencies

```
// 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;
}
```

How can we parallelize this?

- Run `inMandelbrot` sequentially for each point, but parallelize different point computations.

Summary

Identified memory-carried dependencies:

- 3 types of dependencies (RAW, WAR, WAW).

Removed output and anti-dependencies (at a price).

Identified loop-carried dependencies.

Lecture 07—Speculation and Parallelization Patterns

ECE 459: Programming for Performance

January 29, 2013

Last Time

Dependencies:

		Second Access			
		Read		Write	
First Access	Read	No Dependency		Anti-dependency	
	Read	Read After Read (RAR)		Write After Read (WAR)	
	Write	True Dependency		Output Dependency	
	Write	Read After Write (RAW)		Write After Write (WAW)	

We also saw how to break WAR and WAW dependencies.

Part I

Breaking Dependencies with Speculation

Breaking Dependencies

Speculation: architects use it to predict branch targets.

- Need not wait for the branch to be evaluated.

We'll use speculation at a coarser-grained level:
speculatively parallelize source code.

Two ways: **speculative execution** and **value speculation**.

Speculative Execution: Example

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;  
    }  
}
```

Will we need to run `secondLongCalculation`?

Speculative Execution: Example

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;  
    }  
}
```

Will we need to run `secondLongCalculation`?

- OK, so: could we execute `longCalculation` and `secondLongCalculation` in parallel if we didn't have the conditional?

Speculative Execution: Assume No Conditional

Yes, we could parallelize them. Consider this code:

```
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 do both the calculations in parallel and return the same result as before.

- What are we assuming about longCalculation and secondLongCalculation?

Estimating Impact of Speculative Execution

T_1 : time to run longCalculation.

T_2 : time to run secondLongCalculation.

p : probability that secondLongCalculation executes.

In the normal case we have:

$$T_{\text{normal}} = T_1 + pT_2.$$

S : synchronization overhead.

Our speculative code takes:

$$T_{\text{speculative}} = \max(T_1, T_2) + S.$$

Exercise. When is speculative code faster? Slower?
How could you improve it?

Shortcomings of Speculative Execution

Consider the following code:

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

Now we have a true dependency; can't use speculative execution.

But: if the value is predictable, we can execute `secondLongCalculation` using the predicted value.

This is **value speculation**.

Value Speculation Implementation

This Pthread code does value speculation:

```
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: this is like memoization (plus parallelization).

Estimating Impact of Value Speculation

T_1 : time to run `longCalculation`.

T_2 : time to run `secondLongCalculation`.

p : probability that `secondLongCalculation` executes.

S : synchronization overhead.

In the normal case, we again have:

$$T = T_1 + pT_2.$$

This speculative code takes:

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

Exercise. Again, when is speculative code faster?
Slower? How could you improve it?

When Can We Speculate?

Required conditions for safety:

- `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.

General warning: Consider `side effects` of function calls.

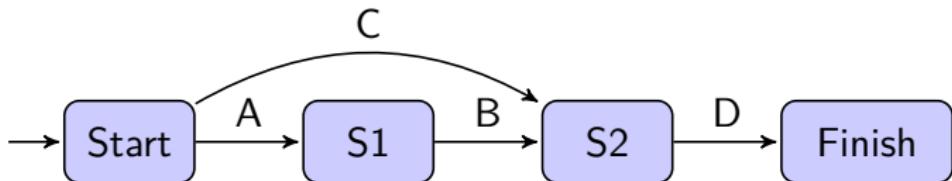
Part II

Parallelization Patterns

Critical Paths

Should be familiar with critical paths from other courses (Gantt charts).

Consider the following diagram:



- B depends on A, C has no dependencies, and D depends on B and C.
- Can execute A-then-B in parallel with C.
- Keep dependencies in mind when calculating speedups for more complex programs.

Data and Task Parallelism

Data parallelism is performing *the same* operations on different input.

Example: doubling all elements of an array.

Task parallelism is performing *different* operations on different input.

Example: playing a video file: one thread decompresses frames, another renders.

Data Parallelism: Single Instruction, Multiple Data

We'll discuss SIMD in more detail later. An overview:

- You can load a bunch of data and perform arithmetic.
- Instructions process multiple data items simultaneously.
(Exact number is hardware-dependent).

For x86-class CPUs, MMX and SSE extensions provide SIMD instructions.

SIMD Example

Consider the following code:

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

In this scenario, we have a regular operation over block data.

We could use threads, but we'll use SIMD.

SIMD Example—Assembly without SIMD

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
```

Just loads, adds, writes and increments.

SIMD Example—Assembly with SIMD

Instead, compiling to SIMD instructions gives:

```
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
```

- Now processing two elements at a time on the same core.
- Also no need for stack-based x87 code.

SIMD Overview

- Operations *packed*: operate on multiple data elements at the same time.
- On modern 64-bit CPUs, SSE has 16 128-bit registers.
- Very good if your data can be *vectorized* and performs math.
- Usual application: image/video processing.
- We'll see more about SIMD as we get into GPU programming (they're very good at these types of applications).

Task-Based Patterns: Overview

- We'll now look at thread and process-based parallelization.
- Although threads and processes differ, we don't care for now.

Pattern 1: Multiple Independent Tasks

Only useful to maximize system utilization.

- Run multiple tasks on the same system
(e.g. database and web server).

If one is memory-bound and the other is I/O-bound, for example, you'll get maximum utilization out of your resources.

Example: cloud computing, each task is independent and can tasks spread themselves over different nodes.

- Performance can increase linearly with the number of threads.

Pattern 2: Multiple Loosely-Coupled Tasks

Tasks aren't quite independent, so there needs to be some inter-task communication (but not much).

- Communication might be from the tasks to a controller or status monitor.

Refactoring an application can help with latency.

For instance: split off the CPU-intensive computations into a separate thread—your application may respond more quickly.

Example: A program (1) receives/forwards packets and (2) logs them. You can split these two tasks into two threads, so you can still receive/forward while waiting for disk. This will increase the **throughput** of the system.

Pattern 3: Multiple Copies of the Same Task

Variant of multiple independent tasks: run multiple copies of the same task (probably on different data).

- No communication between different copies.

Again, performance should increase linearly with number of tasks.

Example: In a rendering application, each thread can be responsible for a frame (gain throughput; same latency).

Pattern 4: Single Task, Multiple Threads

Classic vision of “parallelization”.

Example: Distribute array processing over multiple threads—each thread computes results for a subset of the array.

- Can decrease [latency](#) (and increase [throughput](#)), as we saw with [Amdahl's Law](#).
- Communication can be a problem, if the data is not nicely partitioned.
- Most common implementation is just creating threads and joining them, combining all results at the join.

Pattern 5: Pipeline of Tasks

Seen briefly in computer architecture.

- Use multiple stages; each thread handles a stage.

Example: a program that handles network packets:

(1) accepts packets, (2) processes them, and (3) re-transmits them. Could set up the threads such that each packet goes through the threads.

- Improves **throughput**; may increase **latency** as there's communication between threads.
- In the best case, you'll have a linear speedup.

Rare, since the runtime of the stages will vary, and the slow one will be the bottleneck (but you could have 2 instances of the slow stage).

Pattern 6: Client-Server

To execute a large computation, the server supplies work to many clients—as many as request it.

Client computes results and returns them to the server.

Examples: botnets, SETI@Home, GUI application
(backend acts as the server).

Server can arbitrate access to shared resources (such as network access) by storing the requests and sending them out.

- Parallelism is somewhere between single task, multiple threads and multiple loosely-coupled tasks

Pattern 7: Producer-Consumer

Variant on the pipeline and client-server models.

Producer generates work, and consumer performs work.

Example: producer which generates rendered frames; consumer which orders these frames and writes them to disk.

Any number of producers and consumers.

- This approach can improve throughput and also reduces design complexity

Combining Strategies

Most problems don't fit into one category, so it's often best 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.

Tip: estimate to see what divisions of strategies would work best (might have to do more iterations of Amdahl's law depending on the amount of strategies you can use).

Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate.**

- build system, e.g. parallel make
- optical character recognition system

Midterm Questions from 2011 (1)

For each of the following situations, **name an appropriate parallelization pattern and the granularity at which you would apply it, explain the necessary communication, and explain why your pattern is appropriate.**

- build system, e.g. parallel make
 - ▶ Multiple independent tasks, at a per-file granularity
- optical character recognition system
 - ▶ Pipeline of tasks
 - ▶ 2 tasks - finding characters and analyzing them

Midterm Questions from 2011 (2)

Give a concrete example where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:

- producer-consumer (no rendering frames, please):

Midterm Questions from 2011 (2)

Give a concrete example where you would use the following parallelization patterns. **Explain** the granularity at which you'd apply the pattern.

- single task, multiple threads:
 - ▶ Computation of a mathematical function with independent sub-formulas.
- producer-consumer (no rendering frames, please):
 - ▶ Processing of stock-market data: a server might generate raw financial data (quotes) for a particular security. The server would be the producer. Several clients (or consumers) may take the raw data and use them in different ways, e.g. by computing means, generating charts, etc.

Lecture 08—Automatic Parallelization

ECE 459: Programming for Performance

January 29, 2013

Last Time

Breaking dependencies with speculation:

- speculative execution
- value speculation

Parallelization patterns (7 examples).

How To Parallelize Code: Strategy

Four-step outline:

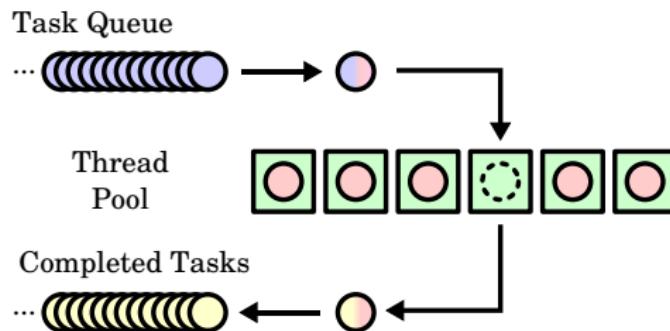
- ① Profile the code.
- ② Look at hotspots; find and optimize dependencies; parallelize dependency chains; change the algorithm if you can.
- ③ Estimate benefits.
- ④ If not good enough, step back and try higher level of abstraction.

Always try to minimize synchronization.

Low-level Implementation Tactic: Thread Pools

Instead of creating threads, destroying them and recreating them, you can use a [thread pool](#).

- It creates n threads; you just push work onto them.



- Only question is: How many threads should you create? (You should have a pretty good feel after Assignment 1).
- Implementation from GLib: `GThreadPool`.

Introduction to Automatic Parallelization

Vision: take a sequential C program and automatically convert it into a parallel version.

Lots of research in the early 1990s, then tapered off.
(it's hard!)

Renewed interest now since multicores are so common.
(it's still hard!)

What Can We Parallelize?

- Some languages are easier than others to reason about (and therefore to automatically parallelize).
- C can be easy to parallelize, given the right code, plus compiler hints.
- “The right code” = arrays with no loop-carried dependencies.

Automatic Parallelization in Practice

Some production compilers support automatic parallelization:

- `icc` (Intel's non-free compiler);
- `solarisstudio` (Oracle's free-as-in-beer compiler¹);
- `gcc` (GNU's free-as-in-speech compiler).

¹<http://www.oracle.com/technetwork/documentation/solaris-studio-12-192994.html>

Example Code from the Textbook

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 }
```

Parallelizing the Example Code

What can we do to parallelize this code?

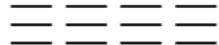
Option 1:

Option 2:

Option 3:

Parallelizing the Example Code

What can we do to parallelize this code?

Option 1: horizontal 

- Create 4 threads; each thread does 1000 iterations on its own sub-array.

Option 2:

Option 3:

Parallelizing the Example Code

What can we do to 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.

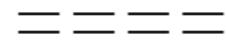
Option 3:

Parallelizing the Example Code

What can we do to 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.

Option 3: vertical 

- Create 4 threads; for each element, the owning thread does 1000 iterations on that element.

Manual Parallelization Demo

I'll show a demo of three example PThread parallelizations.

Methodology: compiling with `solarisstudio`, flags `-O3 -lpthread`.

Which manual option performs better?

Automatic Parallelization of Example Code

Let's also try with automatic parallelization.

Compiling with `solarisstudio` and automatic parallelization yields the following:

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

How will this code compare to our manual efforts?

Note: `solarisstudio` generates two versions of the code, and decides, at runtime, if the parallel code would be faster.

Comparing Parallelization Methods

Under the hood, most parallelization frameworks use OpenMP, which we'll see next lecture.

For now: you can control the number of threads with the OMP_NUM_THREADS environment variable.

How does autoparallelization compare?

Automatic Parallelization in gcc

gcc (since 4.3) can also auto-parallelize loops. However, there are a few problems:

- ① It will not tell you which loops it parallelizes (nicely).
- ② It only operates with a fixed number of threads.
- ③ The profitability metrics are quite simple.
- ④ Only operates in simple cases.

Use the flag `-ftree-parallelize-loops=N` where N is the number of threads.

Note: gcc also uses OpenMP but ignores the `OMP_NUM_THREADS` environment variable.

Understanding Automatic Parallelization in gcc

Flag `-fdump-tree-parloops-details` shows what the automatic parallelizations were, but it's quite unreadable.

Instead, you can look at the assembly code to see the parallelizations (obviously, impractical for a large project).

```
% gcc -std=c99 -O3 -ftree-parallelize-loops=4  
omp_vector_gcc.c -S -o omp_vector_gcc_auto.s
```

The resulting .s file contains the following code:

```
call    GOMP_parallel_start  
leaq    80(%rsp), %rdi  
call    setup._loopfn.0  
call    GOMP_parallel_end
```

Note: gcc also parallelizes `main._loopfn.2` and `main._loopfn.3`, although it looks like it serves little purpose.

Case Study: Multiplying a Matrix by a Vector

Let's see how automatic parallelization does on a more complicated program (could we parallelize this?):

```
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 }
```

Reminder:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 14 \\ 32 \end{bmatrix}$$

Case Study: Automatic Parallelization, Attempt 1

Well, based on our knowledge, we could parallelize the outer loop.

Let's see what `solarisstudio` will do for us...

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

... it refuses to do anything, guesses?

Case Study: Automatic Parallelization, Attempt 2

- The loop bounds are not constant, since one of the variables may alias `row` or `col`, even though `int` \neq `double`.

So, let's add `restrict` to `row` and `col` and see what happens...

```
% solarisstudio-cc -O3 -xautopar -xloopinfo -c fploop.c  
"fploop.c", line 5: not parallelized, unsafe dependence  
"fploop.c", line 8: not parallelized, unsafe dependence
```

Now it recognizes the loop, but still won't parallelize it. Why?

Case Study: Automatic Parallelization, Attempt 3

- `out` might alias `mat` or `vec`, which would make this unsafe

Let's add another `restrict` to `out`:

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

Now, we can get the outer loop to parallelize.

- Parallelizing the outer loop is almost always better than inner loops, and usually it's a waste to do both, so we're done.

Note: We can parallelize the inner loop as well (it's similar to Assignment 1). We'll see that `solarisstudio` can do it automatically.

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;
```

Examples from:

<http://gcc.gnu.org/wiki/AutoparRelated>

Summary of Conditions for Automatic Parallelization

From Chapter 10 of Oracle's *Fortran Programming Guide*² translated to C, a loop 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; and
- have enough work in the loop body to make parallelization profitable.

²<http://download.oracle.com/docs/cd/E19205-01/819-5262/index.html>

Reductions

- Reductions combine input data into a smaller (summary) set.
- We'll see a more complete definition when we touch on functional programming.
- Simplest instance: computing the sum of an array.

Consider the following code:

```
double sum (double *array, int length)
{
    double total = 0;

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

Can we parallelize this?

Reduction Problems

Barriers to parallelization:

- ① value of total depends on previous iterations;
- ② addition is actually non-associative for floating-point values
(is this a problem?)

Recall that “associative” means:

$$a + (b + c) = (a + b) + c.$$

In this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative

Reduction Problems

Barriers to parallelization:

- ① value of total depends on previous iterations;
- ② addition is actually non-associative for floating-point values
(is this a problem?)

Recall that “associative” means:

$$a + (b + c) = (a + b) + c.$$

In this case, the program probably isn't sensitive to rounding, but you should always consider if an operation is associative

Automatic Parallelization via Reduction

If we compile the program with solarisstudio and add the flag `-xreduction`, it will parallelize the code

```
% solarisstudio -cc -xautopar -xloopinfo -xreduction -O3 -c 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:

```
% solarisstudio -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

- A general function could have arbitrary side effects.
- Production compilers tend to avoid parallelizing any loops with function calls.

Some built-in functions, like `sin()`, are “pure”, have no side effects, and are safe to parallelize.

Note: this is why functional languages are nice for parallel programming: impurity is visible in type signatures.

Dealing with Function Calls in solarisstudio

- For solarisstudio you can use the `-xbuiltin` flag to make the compiler use its whitelist of “pure” functions.
- The compiler can then parallelize a loop which uses `sin()` (you shouldn’t replace built-in functions with your own if you use this option).

Other options which may work:

- ① Crank up the optimization level (`-xO4`).
- ② Explicitly tell the compiler to inline certain functions (`-xinline=`, or use the `inline` keyword).

Summary of Automatic Parallelization

To help the compiler, we can:

- use `restrict` (make a restricted copy); and,
- make sure that loop bounds are constant (temporary variables).

Some compilers automatically create different versions for the alias-free case and the (parallelized) aliased case.

At runtime, the program runs the aliased case if correct.

Lecture 09—OpenMP Basics

ECE 459: Programming for Performance

February 5, 2013

Assignment 1 Summary

- Use Pthreads, get practice with benchmarking.
- Results: (TBA)

Assignment 2 Preview

Three parts:

- Restructuring code for Automatic Parallelization.
- Manual parallelization of a loop with OpenMP.
- Using OpenMP tasks effectively.

Should be harder than Assignment 1;
still, didn't take me very long to make solution.

Last Time

Automatic parallelization:

- Compiler splits effort for executing loop code among threads.
- Under the hood, compiler inserts OpenMP directives.

Today: how to write OpenMP directives manually.

Part I

OpenMP

Introduction

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

- OpenMP (Open Multiprocessing) is an API specification which allows you to tell the compiler how you'd like your program to be parallelized.
- All major compilers have OpenMP (GNU, Solaris, Intel, Microsoft).

You use OpenMP¹ by specifying directives in the source code. In C/C++, these directives are pragmas of the form `#pragma omp ...`

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

Benefits of OpenMP

- uses compiler directives—
 - ▶ easily compile same codebase for serial or parallel.
- separates the parallelization implementation from the algorithm implementation.

Directives apply to limited parts of the code,
supporting incremental parallelization of the program.
(Start with the hotspots.)

Simple OpenMP Example

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

Could we parallelize this automatically?

How The Example Works

#pragma will make the compiler parallelize the loop.

- It does not look at loop contents, only loop bounds.
- **It is your responsibility to make sure the code is safe.**

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

You don't need to declare restrict, although it's a good idea. Need restrict for automatic parallelization.

Basic pragma syntax

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

- `#pragma omp` indicates an OpenMP directive;
- `parallel` indicates the start of a parallel region.
- `for` tells OpenMP: run the next `for` loop in parallel.

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

What OpenMP can Parallelize

```
for (int i = 0; i < length; i++) { ... }
```

Can only parallelize loops which satisfy these conditions:

- must be of the form:

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

Note: these restrictions therefore also apply to automatically
parallelized loops.

What OpenMP Does

- Compiler generates code to spawn a `team` of threads; automatically splits off worker-thread code into a separate procedure.
- Generated code uses `fork-join` parallelism; when the master thread hits a parallel region, it gives work to the worker threads, which execute and report back.
- Afterwards, the master thread continues running, while the worker threads wait for more work .

As we saw, you can specify the number of threads by setting the `OMP_NUM_THREADS` environment variable. (You can also adjust by calling `omp_set_num_threads()`).

- Solaris compiler tells you what it did if you use the flags `-xopenmp -xloopinfo`.

Variable Scoping

Concept: thread-local variables ([private](#)) vs shared variables.

- Changes to private variables are visible only to the changing thread.
- Changes to shared variables are visible to all threads.

Variable Scoping for Example

```
for (int i = 0; i < length; i++) { ... }
```

- `length` could be either shared or private.
 - ▶ if it was private, then you would have to copy in the appropriate initial value.
- array variables must be shared.

Default Variable Scoping

Let's look at the defaults that OpenMP uses to parallelize the parallel-for code:

```
% er_src parallel-for.o
1. <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
```

Parallelization information (via OpenMP)

```
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. }
```

Default Variable Scoping Rules: A Summary

- Loop variables are private.
- Variables defined in parallel code are private.
- 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?

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`?

Well, it should be **shared**.

- We want each thread to be able to write to it.

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?

Well, it should be **shared**.

- We want each thread to be able to write to it.
- But, is there a race condition? (of course)

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 the + operator.

Accessing Private Data outside a Parallel Region

Sometimes you want **private** variables, but want them initialized before the loop.

Consider this (silly) code:

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

- data is private, so OpenMP will not copy it initial 1.
- To make OpenMP copy the data before the threads start, use `firstprivate(data)`.
- To publish a variable after the (sequentially) last iteration of the loop, use `lastprivate(data)`.

Thread-Private Data

You might have a global variable, for which each thread should have a persistent local copy—lives across parallel regions.

- Use the `threadprivate` directive.
- Add `copyin` if you want something like `firstprivate`.
- There is no `lastprivate` since the data is accessible after the loop.

Thread-Private Data Example (1)

```
#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");
}
```

Thread-Private Data Example (2)

```
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;
}
```

```
% ./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

- Normally, it's best to parallelize the outermost loop.

Consider this code:

```
#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;
}
```

- Would parallelizing this outer loop benefit us?
The inner loop?

OpenMP supports *collapsing* loops:

- Creates a single loop for all the iterations of the two loops.
- Outer loop only enables the use of 2 threads.
- Collapsed loop lets us use up to 20,000 threads.

Better Performance Through Scheduling: An Example

Default mode: *Static scheduling*.

- Assumes each iteration takes the same running time.

Does that assumption hold for this 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 < 200; j++) {  
            data[i][j] = calc(i+j);  
        }  
    }  
    return 0;  
}
```

Better Performance Through Scheduling

- In example, earlier iterations are faster than later iterations.
Result: sublinear scaling—wait for all iterations to finish.
- Turn on *dynamic schedule* mode by adding `schedule(dynamic)` to the pragma:
 - ▶ Breaks the work into chunks;
 - ▶ Distributes the work to each thread in chunks;
 - ▶ Higher overhead;
 - ▶ Default chunk size of 1
(can modify, e.g. `schedule(dynamic, n/50)`).

More Scheduling

Other schedule modes exist, e.g. `guided`, `auto` and `runtime`.

- `guided` changes the chunk size based on work remaining.
 - ▶ Default minimum chunk size = 1 (can modify)
- `auto` lets OpenMP decide what's best.
- `runtime` doesn't pick a mode until runtime.
 - ▶ Tune with `OMP_SCHEDULE` environment variable

Part II

Beyond for loops: OpenMP Parallel Sections
and Tasks

Why more than for?

- So far, we can parallelize (some) for loops with OpenMP.
- Less powerful than Pthreads. (Also harder to get wrong.)
- Reflects OpenMP's scientific-computation heritage.
- Today, we need more general parallelism, not just matrices.

Parallel Sections Example: Linked Lists (1)

Purely-static mechanism for specifying independent work units which should run in parallel.

Linked list example:

```
#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;
}
```

Parallel Sections Example: Linked Lists (2)

(Exactly) 2 linked lists:

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

Parallelism structure explicitly visible.

Finite number of threads.

(What's another barrier to parallelism here?)

Nested Parallelism

Sometimes you don't want to collapse loops.

Example: (better example in PDF notes!)

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        #pragma omp parallel for
        for (int i = 0; i < 1000; i++) { ... }
    }
    #pragma omp section
    {
        #pragma omp parallel for
        for (int i = 0; i < 1000; i++) { ... }
    }
}
```

To enable nested parallelism, call `omp_set_nested(1)` or set `OMP_NESTED`. (Runtime might refuse.)

OpenMP Tasks

Main new feature in OpenMP 3.0.

`#pragma omp task:` code splits off and scheduled to run later.

More flexible than parallel sections:

- can run as many threads as needed;
- tasks do not need to join (like detached threads).

OpenMP does the task-to-thread mapping—lower overhead.

Examples of tasks

Two examples:

- web server—unstructured requests
- user interface—allows users to start tasks which run in parallel.

Boa webserver main loop example

```
#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();
    }
```

Other OpenMP qualifiers

`untied`: lifts restrictions on task-to-thread mapping.

`single`: only one thread runs the next statement (not N copies).

`flush` directive: write all values in registers or cache to memory.

`barrier`: wait for all threads to complete. (OpenMP also has implicit barriers at ends 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`).

Lecture 10—A Principled View of OpenMP

ECE 459: Programming for Performance

February 5, 2013

What is OpenMP?

A portable, easy to use parallel programming API.

Combines:

- Compiler directives;
- Runtime library routines; and
- Environment variables.

Compiling with OpenMP also defines _OPENMP for ifdefs.

Documentation:

<http://www.openmp.org/mp-documents/OpenMP3.1.pdf>

Directive Format

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

There are **16** directives.

Either a single statement or a compound statement { } goes after the directive.

Most clauses have a **list** as an argument.

- A **list** is a comma-separated list of **list items**.
A **list item** is simply a variable name (for C/C++)

Part I

Data Terminology

Three Keywords for Variable Scope and Storage

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

Private Variables

Declared with `private` clause in OpenMP.

Creates new storage (does not copy values) for the variable.

Scope extends from the start of the region to the end.

Destroyed afterwards.

Pthread pseudocode for private variables:

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

Shared Variables

Declared with `shared` clause in OpenMP.

All threads have access to the same block of data.

Pthread pseudocode:

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

Thread-Private Variables

Declared with `threadprivate` directive in OpenMP.

Each thread makes a copy of the variable.

Variable accessible to the thread in any parallel region.

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()]  
}
```

Contents of Clauses

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` and outside are `shared` (exception: anything with dynamic storage is shared).

Part II

Directives

Parallel

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

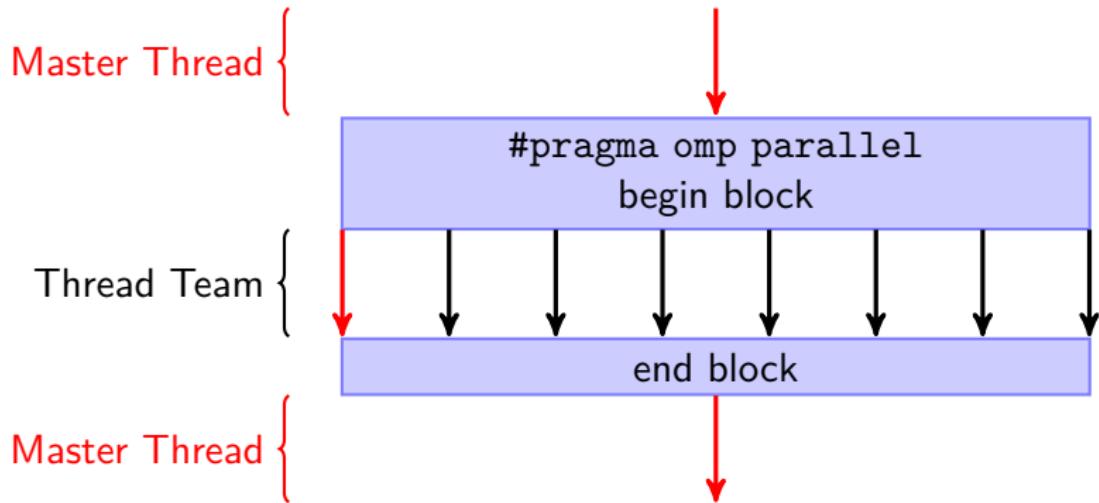
This is the most basic directive in OpenMP.

Forms a team of threads and starts parallel execution.

The thread that enters the region becomes the **master** (thread 0).

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

Visual Explanation of Parallel



- By default, the number of threads used is set globally automatically or manually.
- After the parallel block, the thread team sleeps until it's needed.

Parallel Example

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

If the number of threads is 4, this produces:

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

if and num_threads Clauses

if(*primitive-expression*)

- If primitive-expression false, then only one thread will execute.

If the parallel section is going to run multiple threads (e.g. **if** expression is true), we can specify how many:

num_threads(*integer-expression*)

- Spawns at most **num_threads**, depending on the number of threads available.
- 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

reduction(*operator:list*)

Operators (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.

reduction Clause Pthreads Pseudocode

```
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 Clause

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

Iterations of the loop will be distributed among the current team of threads.

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 single thread should handle at a time.

kind is one of:

- **static**
- **dynamic**
- **guided**
- **auto**
- **runtime**

auto is obvious (OpenMP decides what's best for you).

runtime is also obvious; we'll see how to adjust this later.

schedule Clause kinds

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.
- Starts by dividing the loop into large chunks, and decreases the chunk size as fewer iterations remain.

collapse and ordered Clauses

collapse(n)

This collapses n levels of loops.

n should be at least 2, otherwise nothing happens.

Collapsed loop variables are also made **private**.

ordered

Enables the use of ordered directives inside loop.

Ordered

```
#pragma omp ordered
```

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.

Invalid Use of Ordered

```
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

```
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);
    }
}
```

Valid Use of Ordered

```
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 $i > 9$, use becomes invalid

Tying It All Together

```
#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

```
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: output is deterministic; program will run two threads as long as thread limit is at least 2.

Parallel Loop

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

Basically shorthand for:

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

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

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.
- **Sections** statically limit parallelism to the number of sections lexically in the code.

Parallel Sections

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

Again, basically shorthand for:

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

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

Single

```
#pragma omp single
```

Only a single thread executes the region.

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—a synchronization point.

Loops, Sections, Single have an implicit barrier at the end of their region (unless you use **nowait**).

Cannot be used in any conditional blocks.

Also available in pthreads as `pthread_barrier`.

Master

```
#pragma omp master
```

Similar to the **single** directive.

Master thread (and only the master thread) is guaranteed to enter this region.

No implied barriers, no clauses.

Critical

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

The enclosed region is guaranteed to only run one thread at a time (on a per-name basis).

Same as a block of code in Pthreads surrounded by a mutex lock and unlock.

Atomic

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

Ensures a specific storage location is updated atomically.

More efficient than using critical sections (or else why would they include it?)

Atomic Capture

read expression: $v = x;$

write expression: $x = \text{expr};$

update expression: $x++; x--; ++x; --x;$

$x \text{ binop}=\text{expr}; x = x \text{ binop expr};$

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: $v = x++; v = x--; v = ++x; v = --x;$
 $v = x \text{ binop}=\text{expr};$

Performs the indicated update. Also stores the original or final value computed.

Atomic Capture

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

Structured blocks are equivalent to the expanded expressions.

Other Directives

- **task**
- **taskyield**
- **taskwait**
- **flush**

We'll get into these next lecture.

firstprivate and lastprivate Clauses

Pthreads pseudocode for **firstprivate** clause:

```
int x;

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

Pthread pseudocode for **lastprivate** clause:

```
int x;

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

- Same value as if the loop executed sequentially.

copyin, copyprivate and default Clauses

- **copyin** 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]
}
```

copyprivate is only used with **single**.

- Copies the specified private variables from the thread to all other threads.
- Cannot be used with **nowait**.

default(shared) makes all variables shared;
default(none) prevents sharing by default.

Part III

Runtime Library Routines

Execution Environment

To use the runtime library you need to `#include <omp.h>`.

- `int omp_get_num_procs();`
number of processors in the system.
- `int omp_get_thread_num();`
thread number of the currently executing thread
(master thread will return 0).
- `int omp_in_parallel();`
whether or not currently in a parallel region.
- `int omp_get_num_threads();`
number of threads in current team.

Locks

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).

Usage 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

- `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

Might 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();`

Part IV

Internal Control Variables

Internal Control Variables

Control how OpenMP handles threads.

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 (1)

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

Operation of Parallel Regions (2)

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

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

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

Part V

Summary

Summary

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

Reference Card

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

Lecture 11—Advanced OpenMP

ECE 459: Programming for Performance

February 12, 2013

Last Lecture

Main concepts:

- **parallel**
- **for (ordered)**
- **sections**
- **single**
- **master**

Synchronization:

- **barrier**
- **critical**
- **atomic**

Data sharing: **private**, **shared**, **threadprivate**

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
}
```

Memory Model

OpenMP uses a **relaxed-consistency, shared-memory** model:

- All threads share a single store called **memory**.
(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.

Preventing Simultaneous Execution?

```
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?

Possible States for Example

```
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
```

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).

Memory Model Gotcha

```
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
```

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

Flush: Ensuring Consistent Views of Memory

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

Makes the thread's temporary view of memory consistent with main memory; hence:

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

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

Explaining Flush

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.

Flush Correctness

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 .

Take 2: Same Example, now improved with Flush

```
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
```

- OK. Will this now prevent simultaneous access?

No Luck Yet: Why Flush Fails

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.

Should you use flush?

Probably not, but now you know what it does.

Same Example, Finally Correct

```
a = b = 0
/* thread 1 */          /* thread 2 */
atomic(b = 1)
flush(a, b)
atomic(tmp = a)
if (tmp == 0) then
    // protected section
end if
atomic(a = 1)
flush(a, b)
atomic(tmp = b)
if (tmp == 0) then
    // protected section
end if
```

OpenMP Directives Where Flush Isn't Implied

- 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.

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**

Tasks: if and final Clauses

if (*scalar-logical-expression*)

When expression is `false`, generates an `undefined` task—
the generating task region is suspended until execution of the
`undefined` 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 `undefined` and also execute immediately in
the same thread.

if and final Clauses: Examples

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

Bad mergeable Example

```
#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
```

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 Example: Tree Traversal

```
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);
}
```

OpenMP Example 2: Post-order Tree Traversal

```
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);
    #pragma omp taskwait
    process(p);
}
```

Note: Used an explicit **taskwait** before processing.

OpenMP Example: Parallel Linked List Processing

```
// 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;
            }
        }
    }
}
```

OpenMP Example: Lots of Tasks

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

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

The main loop generates tasks, which are all assigned to the executing thread as it becomes available.

When too many tasks generated: suspends main thread, runs some tasks, then resumes the loop in main thread.

OpenMP Example: Improved version of Lots of Tasks

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

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            int i;
            #pragma omp task untied
            {
                for (i=0; i<LARGE_NUMBER; i++)
                    #pragma omp task
                    process(item[i]);
            }
        }
    }
}
```

- **untied** lets another thread take on tasks.

About Nesting: Restrictions

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

```
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

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.

Example: 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];
        }
    }
}
```

Summary

Finished exploring OpenMP. Key points:

- How to use **flush** correctly.
- How to use OpenMP **tasks** to parallelize unstructured problems.

Lecture 12—Memory Models, Ordering, and Other Atomic Operations

ECE 459: Programming for Performance

February 14, 2013 ♡

Part I

Memory Models

Memory Models

Sequential program: statements execute in order.

Your expectation for concurrency: sequential consistency.

“... 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

In brief:

- ① for each thread: in-order execution;
- ② interleave the threads' executions.

No one has it: too expensive.

Recall the worked example for **flush** last time.

Memory Models: Sequential Consistency

Another view of sequential consistency:

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

Reordering

Compilers and processors may reorder non-interfering memory operations.

$$T1 : x = 1; r1 = y;$$

If two statements are independent:

- OK to execute them in either order.
- (equivalently: publish their results to other threads).

Reordering is a major compiler tactic to produce speedup.

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.

Still, **reorderings** only allowed if they look safe in current context (i.e. independent; different memory addresses).

2011 Final Exam Question

```
x = y = 0  
  
/* thread 1 */           /* thread 2 */  
x = 1;                  y = x;  
r1 = y;                 r2 = x;
```

Assume architecture not sequentially consistent
(weak consistency).

Show me all possible (intermediate and final) memory values
and how they arise.

2011 Final Exam Question: Solution

must include every permutation of lines (since they can be in any order);
then iterate over all the values.

Probably actually too long, but shows how memory reorderings complicate things.

The Compiler Reorders Memory Accesses

When it can prove safety, the **compiler** may reorder instructions (not just the hardware).

Example: want thread 1 to print value set in thread 2.

```
f = 0

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

- If thread 2 reorders its instructions, will we get our intended result?

No.

The Compiler Reorders Memory Accesses

When it can prove safety, the **compiler** may reorder instructions (not just the hardware).

Example: want thread 1 to print value set in thread 2.

```
f = 0

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

- If thread 2 reorders its instructions, will we get our intended result?

No.

Preventing Memory Reordering

A **memory fence** prevents memory operations from crossing the fence (also known as a **memory barrier**).

```
f = 0

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

/* thread 2 */
x = 42;
// memory fence
f = 1;
```

- Now prevents reordering; get expected result.

Preventing Memory Reordering in Programs

Step 1: Don't use volatile on variables¹.

Syntax depends on the compiler.

- 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 */
);
```

Last slide used **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

Memory Fences: Preventing Hardware Memory Reordering

Memory barrier: no access after the barrier becomes visible to the system (i.e. takes effect) until after all accesses before the barrier become visible.

Note: these are all x86 asm instructions.

`mfence`:

- All loads and stores before the fence finish before any more loads or stores execute.

`sfence`:

- All stores before the fence finish before any more stores execute.

`lfence`:

- All loads before the fence finish before any more loads execute.

Preventing Hardware Memory Reordering (Option 2)

Some compilers also support preventing hardware reordering:

- Microsoft Visual Studio C++ Compiler:

```
MemoryBarrier();
```

- Solaris Studio (Oracle) Compiler:

```
--machine_r_barrier();
--machine_w_barrier();
--machine_rw_barrier();
```

- GNU Compiler:

```
--sync_synchronize();
```

Memory Barriers and OpenMP

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();`

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).

Part II

Atomic Operations

Atomic Operations

We saw the **atomic** directive in OpenMP.

Most OpenMP atomic expressions map to atomic hardware instructions.

Other atomic instructions exist.

Compare and Swap

Also called **compare and exchange** (cmpxchg instruction).

```
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 it returned oldval.
- If it did, you know you changed it.

Implementing a Spinlock

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* int) {
    __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, nothing has changed, right?

ABA Problem

Sometimes you'll read a location twice.

If the value is the same, nothing has changed, right?

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

You can combat this by “tagging”: modify value with nonce upon each write.

Can keep value separately from nonce; double compare and swap atomically swaps both value and nonce.

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

Part III

Good C++ Practice

Prefix and Postfix

Lots of people use postfix out of habit, but prefix 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);
```

Common Increment Implementations

Prefix is also known as **increment and fetch**.

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

Postfix is also known as **fetch and increment**.

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

Efficiency

If you're the least concerned about efficiency, always use **prefix** increments/decrements instead of defaulting to postfix.

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

Summary

Memory ordering:

- Sequential consistency;
- Relaxed consistency;
- Weak consistency.

How to prevent memory reordering with fences.

Other atomic operations.

Good increment practice.

Lecture 13—Cache Coherency; Building Servers

ECE 459: Programming for Performance

February 26, 2013

Part I

Cache Coherency

Last Time

⇒ Memory ordering:

- Sequential consistency;
- Relaxed consistency;
- Weak consistency.

⇒ How to prevent memory reordering with fences.

Also:

- Other atomic operations.
- Good increment practice.

Introduction

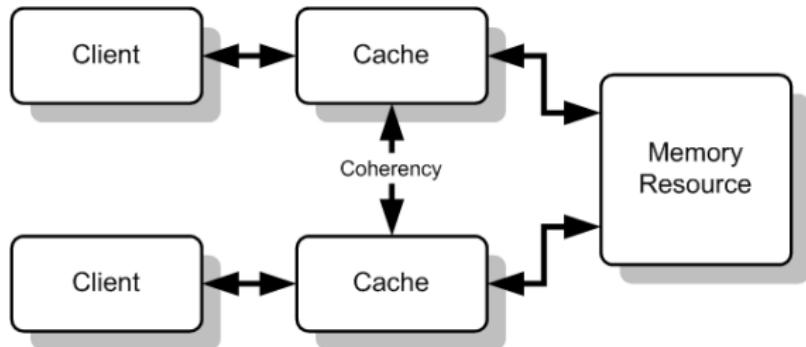


Image courtesy of Wikipedia.

Coherency:

- Values in all caches are consistent;
- System behaves as if all CPUs are using shared memory.

Cache Coherence Example

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

- 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.

What happens?

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

Write-Through Cache

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.)

For write-through caches: normally, when you write to an invalidated location, you bypass the cache and go directly to memory (aka **write no-allocate**).

Write-Through Protocol

- Two states, **valid** and **invalid**, for each memory location.
- Events are either from a processor (**Pr**) or the **Bus**.

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

Write-Through Protocol 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 Cache

- 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:
Indicates the our data has not yet been written to memory.

Write-Back Implementation

The simplest type of write-back protocol (MSI), with 3 states:

- **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.

Initial state, upon first read, is “shared”.

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

- 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 .
 - ▶ CPU3 changes x 's state to modified.
- 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.

Adds 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**: in E state, no other processor has the data.

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.

Permits more efficient usage of the bus.

Good Questions (1)

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

- You might be ok, if all of the writes on processors are to the cache. But they're not!
- Cache coherency won't update any values modified in registers.

Good Questions (2)

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

Good Questions (2)

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



Good Questions (2)

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

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,
volatile doesn't prevent this.

Also, it's likely your variables could be in registers the majority of
the time, except in critical areas.

Cache Coherency Summary

We saw the basics of cache coherence (good to know, but more of an architecture thing).

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.

Part II

Building Servers

Concurrent Socket I/O

Complete change of topic. 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 see the answer by Robert Love, Linux kernel hacker¹.

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

The Real Question

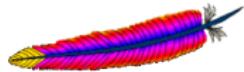
How do you want to do I/O?

Not really “how many threads?”.

Four Choices

- 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



Old Apache model:

- Main thread waits for connections.
- Upon connect, 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.
(can somewhat mitigate with process pool).

Can handle \sim 10 000 processes, but doesn't generally scale.

Blocking I/O; 1 thread per request

We know that threads are more lightweight than processes.

Same as 1 process per request, but less overhead.

I/O is the same—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 Benefits

In 2006, perf benefits of 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).

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

Using Asynchronous I/O in Linux (select/poll)

Basic workflow:

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

Asynchronous I/O Code Example I: Setup

```
#include <aio.h>

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

Asynchronous I/O Code Example II: Read

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

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);
```

Nonblocking I/O in Servers using Select/Poll

Each thread handles 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

Weird programming model; not popular.

Instead of select/poll, pass along a callback,
to be executed upon success or failure.

JavaScript does this extensively, but more unwieldy in C.

We'll see the Go programming model, which makes this easy.

Callback-Based Example

```
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 );

    c->killing_task = delaycb(10, 0, wrap(&clean_up, c));

    /* next step: read information on the new connection */
    fdcb ( cfd , selread , wrap (&read_http_cb , cfd , c , true ,
                                wrap(&read_req_complete_cb )) );
}
```

node.js: A Superficial View

node.js is another event-based nonblocking I/O model.

(Since JavaScript is singlethreaded, nonblocking I/O mandatory.)

Canonical example from 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.

Building on node.js

Usually we want a higher-level view, e.g. expressjs³.

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/tglinex/nodrr/blob/master/controllers/nod.js>

Summary: Building Servers

- Blocking I/O; 1 process per request (old Apache).
- Blocking I/O; 1 thread per request (Java).
- Asynchronous I/O, pool of threads, callbacks,
each thread handles multiple connections. (no one does this)
- Nonblocking I/O, pool of threads,
multiplexed with select/poll, event-driven,
each thread handles multiple connections. (JavaScript)

Lecture 14—Lock granularity;
Reentrant vs. Thread-safe; Inlining; Benchmarking A2;
High-Level Language Perf Tweaks
ECE 459: Programming for Performance

February 28, 2013

Previous Lecture

- Cache coherency;
- Implementing servers to handle many connections.

Oh, and a midterm.

Part I

Locking Granularity

Locking

Locks prevent data races.

- Locks' extents constitute their **granularity**—do you lock large sections of your program with a big lock, or do you divide the locks and protect smaller sections?

Concerns when using locks:

- overhead;
- contention; and
- deadlocks.

Locking: 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.

Locking: Contention

Most locking time is wasted waiting for the lock to become available.

How can we fix this?

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

Locking: Deadlocks

The more locks you have, the more you have to worry about deadlocks.

Key condition:

waiting for a lock held by process X
while holding a lock held by process X' . ($X = X'$ allowed).

Flashback: From Lecture 1

Consider two processors trying to get two *locks*:

Thread 1

Get Lock 1

Get Lock 2

Release Lock 2

Release Lock 1

Thread 2

Get Lock 2

Get 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**).

Key to Preventing Deadlock

Always be careful if
your code **acquires a lock while holding one.**

Here's how to prevent a deadlock:

- Ensure consistent ordering in acquiring locks; or
- Use `trylock`.

Preventing Deadlocks—Ensuring Consistent Ordering

```
void f1() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&l1);
}

void f2() {
    lock(&l1);
    lock(&l2);
    // protected code
    unlock(&l2);
    unlock(&l1);
}
```

This code will not deadlock: you can only get **l2** if you have **l1**.

Preventing Deadlocks—Using `trylock`

Recall: Pthreads' `trylock` returns 0 if it gets the lock.

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

This code also won't deadlock: it will give up **l1** if it can't get **l2**.

Coarse-Grained Locking (1)



Coarse-Grained Locking (2)

Advantages:

- Easier to implement;
- No chance of deadlocking;
- Lowest memory usage / setup time.

Disadvantages:

- Your parallel program can quickly become sequential.

Coarse-Grained Locking Example—Python GIL

This is the main reason (most) scripting languages have poor parallel performance; Python's just an example.

- Python puts a lock around the whole interpreter (global interpreter lock).
- 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, it'll slow down your system).

Fine-Grained Locking (1)



Fine-Grained Locking (2)

Advantages:

- Maximizes parallelization in your program.

Disadvantages

- May be mostly wasted memory / setup time.
- Prone to deadlocks.
- Generally more error-prone (be sure you grab the right lock!)

Fine-Grained Locking Examples

The Linux kernel used to have **one big lock** that essentially made the kernel sequential.

- (worked fine for single-processor systems!)

Now uses finer-grained locks for performance.

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

Can lock individual objects.

Part II

Reentrancy

Reentrancy

⇒ A function can be suspended in the middle and **re-entered** (called again) before the previous execution returns.

Does not always mean **thread-safe** (although it usually is).

- Recall: **thread-safe** is essentially “no data races”.

Moot point if the function only modifies local data, e.g. `sin()`.

Reentrancy Example

Courtesy of Wikipedia (with modifications):

```
int t;

void swap(int *x, int *y) {
    t = *x;
    *x = *y;
    // hardware 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);
```

Reentrancy Example—Explained (a trace)

```
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

Reentrancy Example, Fixed

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t; // save global variable
    t = *x;
    *x = *y;
    // hardware 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);
```

Reentrancy Example, Fixed—Explained (a trace)

```
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

Previous Example: thread-safety

Is the previous reentrant code also thread-safe?
(This is more what we're concerned about in this course.)

Let's see:

```
int t;

void swap(int *x, int *y) {
    int s;

    s = t; // save global variable
    t = *x;
    *x = *y;
    // hardware interrupt might invoke isr() here!
    *y = t;
    t = s; // restore global variable
}
```

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

Previous Example: thread-safety trace

```
global: t

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

Reentrancy vs Thread-Safety (1)

- Re-entrant does not always mean thread-safe (as we saw)
 - ▶ But, for most sane implementations, it is thread-safe

Ok, but are **thread-safe** functions reentrant?

Reentrancy vs Thread-Safety (2)

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.

Summary of Reentrancy vs Thread-Safety

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 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.

Another Definition of Thread-Safe Functions

“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

```
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.

Part III

Good 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.

No overhead... sounds like better performance...
let's inline everything!

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:

```
inline max(const int& x, const int& y) {
    return x < y ? y : x;
}
```

The Other Side of Inlining

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.

Compilers on Inlining

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.

From a Usability Point-of-View

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.

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. (non-binary-compatible change!)

Not a problem for non-inlined functions—programs execute the new function dynamically at runtime.

Notes on Benchmarking A2—Sequential and Parallel Physical Cores

- 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.

A2—Parallel Virtual CPUs vs Virtual CPUs + 1

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.

Part IV

High-Level Language Performance Tweaks

Introduction

So far, we've only seen C—we haven't seen anything complex.

C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard.

Common C sights:

- **#define macros**
- **void***

C++11 has made major strides towards readability and efficiency
(it provides light-weight abstractions).

Goal

Sort a bunch of integers.

In **C**, usually use qsort from stdlib.h.

```
void qsort (void* base, size_t num, size_t size,  
           int (*comparator) (const void*, const void*));
```

- A fairly ugly definition (as usual, for generic C functions)

How ugly? qsort usage

```
#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.

C++ sort

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
);
```

¹... nicer to use, after you get over templates (they're useful, I swear).

C++ sort Usage

```
#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.
By default, `sort` uses `operator<` on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

Timing Various Sorts

[Shown: actual runtimes of `qsort` vs `sort`]

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 Sort

[Shown: actual runtimes of custom sort vs sort]

- The C++ version is still faster (although it's close).
- However, 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.

Lecture Fun

Let's throw Java-style programming (or at least collections) into the mix and see what happens.

Vectors vs. Lists: Problem

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

- **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

[Shown: actual runtimes of vectors and lists]

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.

Performance Tips: Bullets

- 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.

Programming for Performance with the Compiler

- 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

- 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.