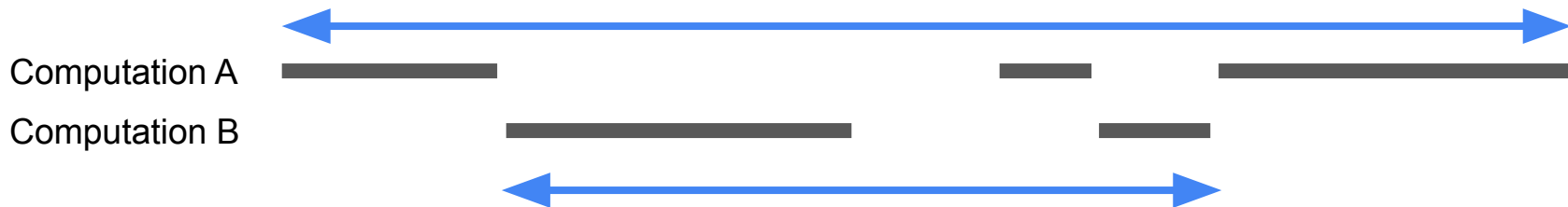


Concurrency, Parallelism & Threads

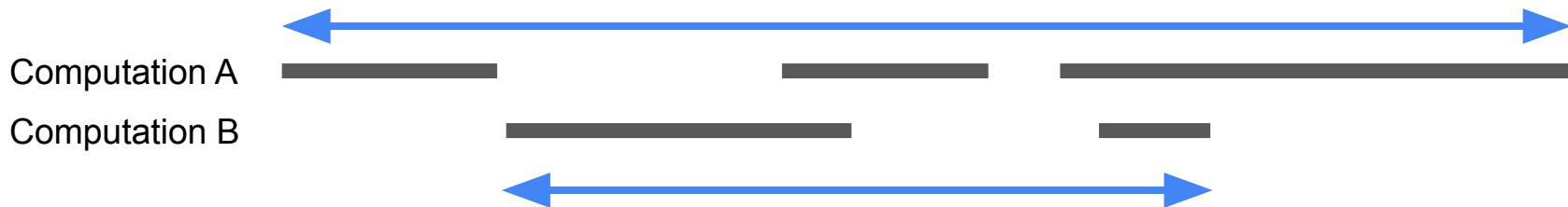
COMP1521 23T3

Concurrency & Parallelism

Concurrency: Multiple computations with *overlapping* time periods. Does not *have* to be simultaneous.



Parallelism: Multiple computations executing *simultaneously*.



Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—Very high-speed computers may be classified as follows:

- 1) Single Instruction Stream—Single Data Stream (SISD)
- 2) Single Instruction Stream—Multiple Data Stream (SIMD)
- 3) Multiple Instruction Stream—Single Data Stream (MISD)
- 4) Multiple Instruction Stream—Multiple Data Stream (MIMD).

“Stream,” as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD

Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

INTRODUCTION

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation

Flynn's Taxonomy for Classifying Parallelism

SISD: Single Instruction, Single Data (“no parallelism”)

- e.g. mipsy

SIMD: Single Instruction, Multiple Data (“vector processing”)

- Multiple cores of a CPU executing (parts of) same instruction
- e.g. GPUs

MISD: Multiple Instruction, Single Data

- e.g., fault tolerance in space shuttles (task replication)

MIMD: Multiple Instruction, Multiple Data (“multiprocessing”)

- Multiple cores of a CPU executing different instructions

Distributing computation across computers: parallel computing

- Parallelism can occur across multiple computers!
- One popular framework is [MapReduce](#)
- Necessary for *very big* computations and *very large* sets of data
- Can be difficult to deal with synchronisation and failure of machines
- Out of scope for COMP1521

```
Hello from flute04! 28 = 7 x 2 x 2
Hello from bongo03! 23 is prime!
Hello from flute22! 95 = 19 x 5
Hello from bongo13! 33 = 11 x 3
Hello from viola09! 44 = 11 x 2 x 2
  Hello from oboe01! 32 = 2 x 2 x 2 x 2 x 2
  Hello from oboe13! 35 = 7 x 5
Hello from tabla22! 27 = 3 x 3 x 3
Hello from cello06! 12 = 3 x 2 x 2
Hello from organ08! 51 = 17 x 3
  Hello from kora17! 53 is prime!
Hello from organ00! 52 = 13 x 2 x 2
  Hello from oboe04! 30 = 5 x 3 x 2
```

Concurrency with processes

- Create multiple processes, and split the job across them
- Each process
 - runs concurrently
 - has its own address space (giving **isolation**)
- Processes can be distributed across cores, giving parallelism
- But this strategy is expensive!
 - Creation/teardown expensive
 - Switching expensive
 - Lots of state per process
 - ⇒ costs memory
 - Communication can be complicated and expensive

```
int posix_spawn_file_actions_destroy(
    posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_init(
    posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(
    posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_adddup2(
    posix_spawn_file_actions_t *file_actions, int fildes, int newfildes);
```

- functions to combine file operations with posix_spawn process creation
- awkward to understand and use — but robust

Introducing threads: concurrency *within* a process

- Threads allows us to create concurrency *within* a process
- Each thread has a separate execution state
 - Separate registers, separate program counter
- Threads within a process *share* the address space:
 - threads share code
 - threads share global variables
 - threads share the heap (malloc)
 - cheap communication!
- Each thread has a separate stack
 - but a thread can still read/write to another thread's stack
- Some other process state is shared
 - environment variables, file descriptors, current working directory, ...

Using POSIX Threads (pthreads)

- POSIX Threads is a widely-supported threading model
- Provides an API/model for managing threads (and synchronisation)

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

- Use **-pthread** when compiling
- C11 and later also provides a model/API similar to pthreads
 - Has some small differences with pthreads, and generally less-supported and less used (for now...)

Creating threads with pthread_create

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

- Starts a new thread running `start_routine(arg)`
- An ID for the thread is stored in `thread`
- Thread has attributes specified in `attr` (NULL if you don't want special attributes)
- Returns 0 if OK, otherwise an error number (**does not set `errno`!**)
- Analogous to ***posix_spawn***.

Waiting for threads with pthread_join

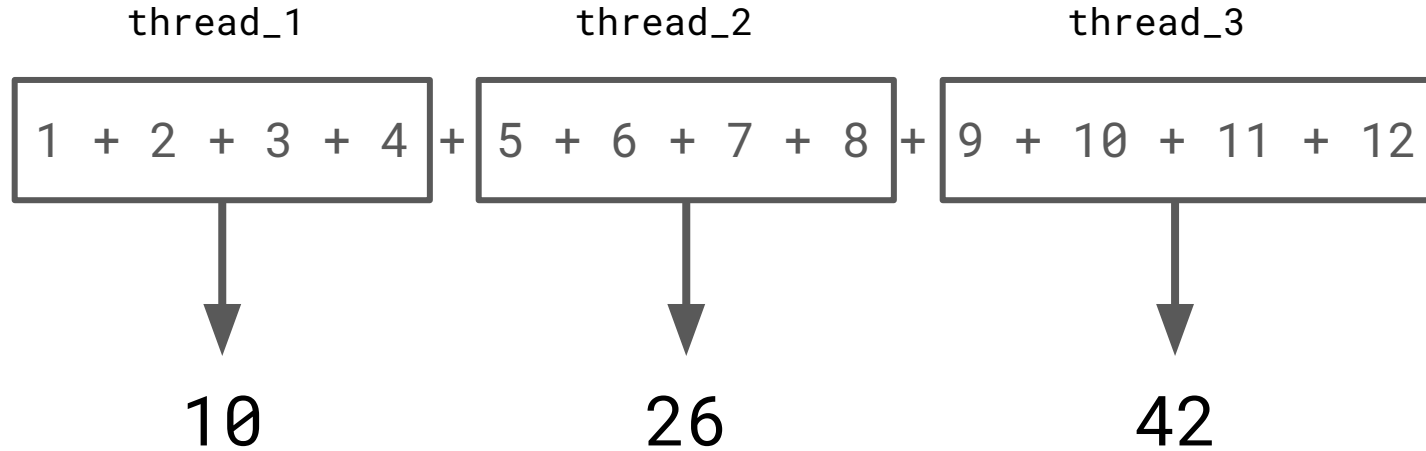
```
int pthread_join(pthread_t thread, void **retval);
```

- Waits for `thread` to terminate, if it hasn't already terminated
- Return/exit value of thread placed in `*retval`
- Analogous to `waitpid`
- When **main** returns, *all* threads terminate

Some examples

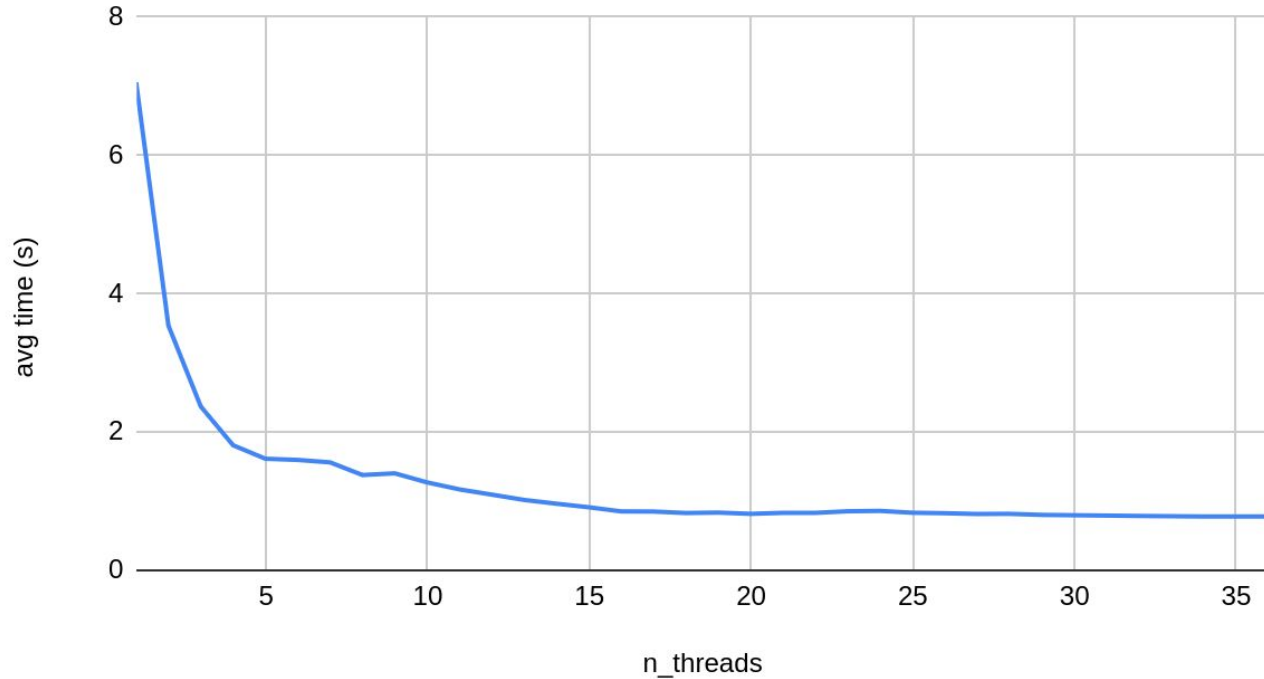
- `two_infinite_loops.c`
- `two_infinite_loops_my_puts.c`
- `two_threads.c`

Example thread_sum



A graph of the performance of thread_sum.c

avg time (s) vs. n_threads (summing to 10,000,000,000)



Some other concurrency benefits

- One thread can wait for I/O (block) while others make progress or wait for separate I/O
- Useful for user interface programming

Bank accounts, and data races

- Concurrency access to shared data, when at least one of the accesses is a *write*, leads to a data race!
- <https://cgi.cse.unsw.edu.au/~xavc/data-race/>
- We need to ensure *mutual exclusion* for these shared resources

A solution: mutices

- We need a way of guaranteeing *mutual exclusion* for certain shared resources (such as ***bank_account***)
- We associate each of those resources with a ***mutex***
- Only one thread can hold a mutex, any other threads which attempt to lock the mutex must wait until the mutex is unlocked
- So only one thread will be executing the section between the mutex lock and the mutex unlock

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```


Mutex the world!


- Mutexes solve all our data race problems!
 - So... just put a mutex around everything!
 - This works... but then we lose the advantages of parallelism
 - Mutexes also have overhead
-
- Python 🦆 does this (although [they're trying to stop...](#))
 - Linux used to do this (they removed the 'Big Kernel Lock' in 2011)

Deadlocks

- No thread can make progress!
- The system is deadlocked


THREAD 1


- 1. acquire lock_A
- 2. acquire lock_B **✗ BLOCKED!**
- 3. do_somthing(A, B)
- 4. release lock_B
- 5. release lock_A


lock_A 

THREAD 2

- 1. acquire lock_B
- 2. acquire lock_A **✗ BLOCKED!**
- 3. do_somthing(A, B)
- 4. release lock_A
- 5. release lock_B

lock_A 

lock_B 

lock_B 

This slide has animations, use the 'slideshow' button to view it.

Solving deadlocks

- A simple rule to avoid deadlocks:
 - All thread *must* acquire locks in the same order
 - (also good if locks are released in reverse order, if possible)
- e.g., always acquire lock_A before lock_B

THREAD 1

1. acquire lock_A
2. acquire lock_B
3. do_something(A, B)
4. release lock_B
5. release lock_A

THREAD 2

1. acquire lock_A
2. acquire lock_B
3. do_something(A, B)
4. release lock_B
5. release lock_A

Atomics

- With hardware support, we can avoid data races without needing to use locks!
 - In C, we can use ‘atomic types’, which guarantee that certain operations using them will be performed *atomically* (indivisibly) \Rightarrow no data race!
 - Also avoids overhead of mutexes
 - And since no locks are involved, we can’t introduce deadlock
-
- A subset of functions in **stdatomic.h**:
 - `atomic_fetch_add`
 - `atomic_fetch_sub`
 - `atomic_fetch_or`
 - `atomic_fetch_xor`
 - `atomic_fetch_and`

More on atomics

- `x += 1;` is different from `x = x + 1;` if `x` is atomic!
- Atomics don't solve all concurrency problems
- There are still some subtle problems (which we don't cover in COMP1521)

Concurrency is really complex!

- This is just a taste of concurrency!
- Other fun concurrency problems/concepts: livelock, starvation, thundering herd, memory ordering, semaphores, software transactional memory, user threads, fibers, etc.
- A number of courses at UNSW offer more:
 - COMP3231/COMP3891: [Extended] operating systems
 - COMP3151: Foundations of Concurrency
 - COMP6991: Solving Modern Programming Problems with Rust
 - ... and more!