

# ACM/CS 114

## Parallel algorithms for scientific applications

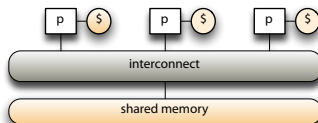
Michael A. G. Aïvázis

California Institute of Technology

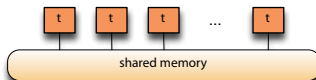
Winter 2010

# Threads and shared memory parallelism

- ▶ recall the shared memory architecture



- ▶ processors are connected to a memory pool with a global address space
  - ▶ processors have their own cache but no private memory
- ▶ model is relevant for *threads*
  - ▶ lightweight processes that can be scheduled independently, but share many OS resources
    - ▶ CPU
    - ▶ memory
    - ▶ but also file descriptors, process environment, etc.
  - ▶ supported by most modern operating systems



# Processes and threads

- ▶ in most operating systems, a process has
  - ▶ process id and group id, user id and group id
  - ▶ environment variables
  - ▶ working directory
  - ▶ scheduling information
  - ▶ registers, stack, heap, instruction stream
  - ▶ file descriptors, signal handlers, other process dependent structures
- ▶ threads
  - ▶ share many of the per-process properties
    - ▶ they are lightweight since they incur low overhead
  - ▶ have their own copy of
    - ▶ registers, stack, instruction stream
    - ▶ scheduling information
- ▶ threads are important programming constructs
  - ▶ every vendor supports a proprietary interface
  - ▶ *pthread*s, the POSIX standard API specification brought portability
  - ▶ standardized creation, management, synchronization

# The pthreads API

- ▶ threads require support from the compiler, the linker, the loader, and the OS kernel
  - ▶ thread safety
- ▶ special command line argument to most compliant compilers
  - ▶ changes the instruction strategy
  - ▶ adds the pthread runtime library to the link line
  - ▶ links against the thread safe runtime
- ▶ naming conventions

Prefix	Functional group
pthread_	access to the threads, and some miscellaneous routines
pthread_attr_	thread attribute objects
pthread_mutex_	mutexes
pthread_mutexattr_	mutex attribute objects
pthread_cond_	condition variables
pthread_condattr_	condition variable attribute objects
pthread_key_	thread-specific data keys
pthread_rwlock_	read/write locks
pthread_barrier_	synchronization barriers

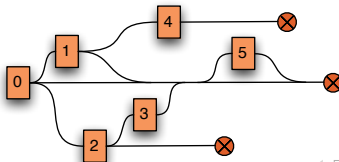
- ▶ the standard specifies the API for C only; FORTRAN support varies
  - ▶ must include `pthread.h`
- ▶ lots of good books; see <http://acm114.caltech.edu/references>

# Creating threads

- ▶ create threads by calling

```
1 int pthread_create(  
2     pthread_t* id, const pthread_attr_t* attr,  
3     void* (*startup)(void*), void* arg);
```

- ▶ initially a process has one thread; every other thread must be explicitly created by calling `pthread_create` and passing
  - ▶ `id`: the location where a unique thread identifier will be stored
  - ▶ `attr`: an opaque attribute object with thread initialization options
  - ▶ `startup`: a pointer to a C function that will be executed by the thread once it gets scheduled
  - ▶ `arg`: user defined data to be passed to `startup`; may be `NULL`
- ▶ once scheduled, threads are first class citizens
- ▶ the maximum number of threads per process depends on the implementation



# Terminating threads

- ▶ several ways to terminate a thread
  - ▶ the thread returns from `main`
  - ▶ the thread explicitly calls `pthread_exit`
  - ▶ the thread is killed when another thread calls `pthread_cancel`
  - ▶ the process terminates due to some system call, e.g. `exit`, `exec`, etc.

- ▶ use `pthread_exit` to kill a thread when it is no longer needed

```
1 int pthread_exit(void * status);
```

- ▶ if `main` finishes and any threads remain
  - ▶ they get killed unless `main` has called `pthread_exit`
  - ▶ otherwise they continue to run
- ▶ thread routines do not have to call `pthread_exit` unless they intend to pass their termination status to their creator
- ▶ `pthread_exit` does not perform any process cleanup: it doesn't flush/close files, release other resources, signal the process parent, etc.

# Hello world

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #define THREADS 10
4
5 void* hello(void* threadID) {
6     long id = (long) threadID;
7     printf("hello from %02ld/%0d\n", id, THREADS);
8     pthread_exit(NULL);
9     return NULL;
10 }
11
12 int main(int argc, char* argv[]) {
13     long id;
14     int status;
15     pthread_t threads[THREADS];
16
17     for (id=0; id<THREADS; id++) {
18         printf("creating thread %02ld\n", id);
19         status = pthread_create(&threads[id], NULL, hello, (void*) id);
20         if (status) {
21             printf("error %d in pthread_create\n", status);
22         }
23     }
24
25     pthread_exit(NULL);
26     return 0;
27 }
```

# Joining and detaching

- ▶ in the example in Slide 7, the main thread exits without knowing whether any of the threads it spawned have finished
  - ▶ saying “hello” is asynchronous
  - ▶ but gathering the results of parallel calculations normally isn’t
- ▶ *thread synchronization* can be achieved using `pthread_join`
  - ▶ the `pthread_create` caller saves the thread id
  - ▶ the thread is scheduled, executes, and calls `pthread_exit`
  - ▶ any other thread can wait for this thread to finish by calling `thread_join` with the saved thread id and also retrieve the termination status
- ▶ for this to work, a thread must be *joinable*
  - ▶ controlled by the thread creation attributes
  - ▶ for portability, you should always mark your joinable threads explicitly
- ▶ a thread that will never be joined may be *detached*
  - ▶ by setting the corresponding attribute during thread creation
  - ▶ or, by calling `pthread_detach` at any point
  - ▶ detaching a thread saves some system resources



# Creating mutexes

- ▶ a *mutex* is a locking mechanism that helps guarantee exclusive access to a section of code, most often to control access to shared variables
- ▶ mutexes are created using

```
1 int pthread_mutex_init(  
2   pthread_mutex_t* mutex, const pthread_mutexattr_t* attr);
```

- ▶ they start out unlocked
  - ▶ the `attr` enables more advanced (but perhaps non-portable) use
- ▶ mutexes are destroyed using

```
1 int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

- ▶ destroy mutexes you are no longer using to prevent resource leakage

# Locking and unlocking mutexes

- ▶ threads manipulate mutexes through

```
1 int pthread_mutex_lock(pthread_mutex_t* mutex);  
2 int pthread_mutex_trylock(pthread_mutex_t* mutex);  
3 int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

- ▶ `pthread_mutex_lock` attempts to gain exclusive access
  - ▶ if the mutex is unlocked, it locks it and returns
  - ▶ otherwise, it blocks until the mutex is unlocked; when the mutex is unlocked, it locks it and returns
- ▶ `pthread_mutex_unlock` attempts to release a mutex
  - ▶ if it was previously locked by this thread, the mutex is unlocked
  - ▶ if it was not previously locked, the call returns with an error code
  - ▶ if it was locked, but not by the calling thread, the call returns an error code
- ▶ `pthread_mutex_trylock` attempts to lock the mutex
  - ▶ if it is unlocked, the call locks it and returns
  - ▶ if it is locked, the call returns immediately with a *busy* error code
- ▶ locking and unlocking mutexes is explicitly orchestrated by the programmer
- ▶ when multiple threads are blocked waiting for a mutex, there is no way to predict which one will succeed when the mutex becomes available

# Condition variables

- ▶ condition variables build upon mutexes to enable threads to signal each other when some condition is met
- ▶ they are created using

```
1 int pthread_cond_init(  
2     pthread_cond_t* condition, const pthread_condattr_t* attr);
```

- ▶ and destroyed using

```
1 int pthread_cond_destroy(pthread_cond_t* condition);
```

# Using condition variables

- ▶ the following three routines implement the condition variable semantics

```
1 int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);  
2 int pthread_cond_signal(pthread_cond_t* cond);  
3 int pthread_cond_broadcast(pthread_cond_t* cond);
```

- ▶ `pthread_cond_wait` blocks the calling thread until the specified condition is *signaled*
  - ▶ it must be called with the mutex locked by the calling thread
  - ▶ `pthread_cond_wait` releases the lock while the thread is blocked
  - ▶ after the matching signal is received, the thread is awakened and the mutex locked
  - ▶ the thread is responsible for releasing the mutex when it is done
- ▶ `pthread_cond_signal` wakes up a thread that is waiting for the given condition variable
  - ▶ mutex must be locked before calling it
  - ▶ mutex must be unlocked after signaling, so blocking threads can be awakened
- ▶ `pthread_cond_broadcast` can be used instead if multiple threads are waiting for a signal

# Condition variable caveats

- ▶ be careful with condition variables; make sure that
  - ▶ a thread has called `pthread_cond_wait` before any thread calls `pthread_cond_signal`
  - ▶ the mutex associated with the condition is locked before calling `pthread_cond_wait`, otherwise it might *not block*
  - ▶ the thread that calls `pthread_cond_signal` unlocks the associated mutex, otherwise the threads waiting for the signal will continue to block

# Attributes of threads, mutexes and condition variables

- ▶ threads, mutexes and condition variables have associated attribute structures that can be used to tune the default creation parameters
- ▶ they are created and destroyed using

```
1 int pthread_attr_init(pthread_attr_t* attr);  
2 int pthread_attr_destroy(pthread_attr_t* attr);  
3  
4 int pthread_mutexattr_init(pthread_mutexattr_t* attr);  
5 int pthread_mutexattr_destroy(pthread_mutexattr_t* attr);  
6  
7 int pthread_condattr_init(pthread_condattr_t* attr);  
8 int pthread_condattr_destroy(pthread_condattr_t* attr);
```

- ▶ typically, the defaults are adequate and tuned to the details of the operating system
- ▶ if you make excessive use of the stack, e.g. large arrays as local variable or deep recursion, you might want to know about

```
1 int pthread_attr_getstacksize(pthread_attr_t* attr, size_t* size);  
2 int pthread_attr_setstacksize(pthread_attr_t* attr, size_t size);
```

# Other useful routines

- ▶ a thread can access its unique id assigned by the system by calling

```
1 pthread_t pthread_self(void);
```

- ▶ since system thread ids are opaque types, you cannot use == to compare them; instead, use

```
1 int pthread_equal(pthread_t id1, pthread_t id2);
```

- ▶ you can place all thread initialization code in a startup routine and call

```
1 int pthread_once(pthread_once_t* control_structure, void (*startup_routine)
```

# Advanced topics

- ▶ there is quite a bit more in the standard
- ▶ *keys*: creating and accessing per-thread data
  - ▶ as the code get more complicated, it becomes increasingly difficult to pass complete thread-specific information from function to function
  - ▶ possible solutions:
    - ▶ the FORTRAN syndrome, where subroutines end up having dozens of arguments
    - ▶ global variables
    - ▶ an associative container that allows each thread to store and retrieve arbitrary data
- ▶ finer control over thread scheduling
  - ▶ scheduling algorithms and priorities are implementation dependent
  - ▶ there are routines in the standard that enable explicit tuning
  - ▶ the standard guarantees that the routines will be *available*, but they don't have to be *implemented*
- ▶ condition variable sharing across processes
- ▶ explicitly canceling threads
- ▶ the somewhat complicated interactions between threads and signals
- ▶ other synchronization constructs: barriers and read/write locks