

ACM/CS 114

Parallel algorithms for scientific applications

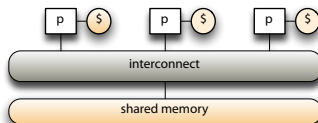
Michael A. G. Aïvázis

California Institute of Technology

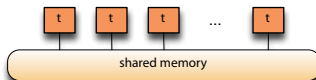
Winter 2010

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

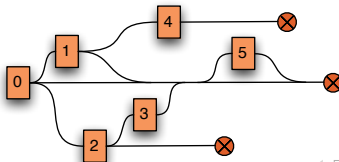
- ▶ the standard specifies the API for C only; FORTRAN support varies
 - ▶ must include `pthread.h`
- ▶ lots of good books; see <http://acm114.caltech.edu/references>

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     /* there is a problem here... */
25     pthread_exit(NULL);
26     return 0;
27 }
```