Parallel Programming POSIX Threads (Pthreads) for Shared Address Space Programming

http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
Pthreads

• Posix threads package
  • Available on all machines (portable standard)
  • Sort of like doing “parallel” (not “parallel for”) in OpenMP explicitly

• Basic calls:
  • pthread_create: creates a thread, to execute a given function
  • Pthread_join
  • barrier, lock, mutex
  • Thread private variables

• Many online resources:
  • E.g., https://computing.llnl.gov/tutorials/pthreads/
• **Spawn an attached thread**

```
#include <pthread.h>

pthread_create(&thread1, NULL, func, &arg)

pthread_join(thread1, status)
```

• **Thread execution**

```
void func(&arg)
{
    // Thread code
    return(*status);
}
```

• **Detached threads**

  • Join is not needed
  • The OS destroys thread resources when they terminate
  • A parameter in the create call indicates a detached thread
Executing a Thread

Main Program

. .

pthread_create(&thread1, NULL, func, &arg);

. .

pthread_join(thread1, *status);

. .

thread1

func (&arg)

{

. .

return (*status);

}
Locks

• Declare a lock: pthread_mutex_t mutex;
• Declare a mutex attribute: pthread_mutexattr_t mta;
• Initializing an attribute: (spin_only, limited_spin, no_spin, recursive, metered)
  pthread_mutexattr_init(&mta);
  pthread_mutexattr_settype(&mta, PTHREAD_MUTEX_RECURSIVE);
  pthread_mutexattr_setname_np(&mta, "My Mutex");
• Initialize a mutex
  pthread_mutex_init(&mutex, NULL); // Use defaults
  pthread_mutex_init(&mutex, &mta); // or use designated attributes
• Enter and release
  Pthread_mutex_lock(&mutex); and pthread_mutex_unlock(&mutex);
• Try lock without blocking: pthread_mutex_trylock(&mutex);
• Release resources
  pthread_mutex_destroy(mutex); and pthread_mutexattr_destroy(&mta);

The attributes are used to deal with more complex scenarios, such as across process locks or thread priorities. We will use the basic behavior by passing NULL for the attribute parameter.
Hello World: Pthreads

```c
void* Hello ( void* myRank )
{
    printf ( "Hello from thread %ld\n" , (long)(*myRank) ) ;
    return NULL ;
}

void main ( int argc , char argv[] )
{
    long t;
    pthread_t[] threadHandles ;
    int threads = strtol ( argv[ 1 ] , NULL , 10 ) ;
    thread_handles = malloc ( threads * sizeof( pthread_t ) ) ;
    for ( t = 0 ; t< threads; t ++ )
        pthread_create(&threadHandles[ t ] , NULL , Hello , ( void * ) &t );
    printf ( "Hello from the main thread\n" ) ;
    for ( t = 0 ; t < threads; t ++) pthread_join ( threadHandles[ t ] , NULL ) ;
    free( threadHandles ) ;
} 
```
Calculate $\pi$

- **Sequential version**
  ```c
  int inTheCircle=0, numtrials=1000000;
  double x, y;
  for (i=0; i<numTrials; i++)
    { x = rand();
      y = rand();
      if((x*x + y*y) < 1.0)
        inTheCircle++;
    }
  pi = 4*inTheCircle/numTrials;
  ```
Pthread Solution

Main routine

```c
void* myPi ( void *rank, int trials ) {
    int myRank = ( int ) (*rank);
    double x, y;
    for (i=0; i< trials; i++)
    {
        x = rand();
        y = rand();
        if((x*x + y*y) < 1.0)
            inTheCircle++;
    }
}

for ( t = 0 ; t< threads; t ++ ){
    int localTrials = numTrials/threads;
    pthread_create(&threadHandles[t],NULL ,myPi ,(void*) &t,(void*) &localTrials);
}

Pi = 4*inTheCirlce/numTrials;
```

Failed pthread version

```c
void* myPi( void *rank, int trials ) {  
    int myRank = ( int ) (*rank);
    double x, y;
    for (i=0; i< trials; i++)
    {
        x = rand();
        y = rand();
        if((x*x + y*y) < 1.0)
            inTheCircle++;
    }
}
```
void* myPi ( void *rank, int trials )
{
    int myRank = ( int ) (*rank);
    double x, y;
    for (i=0; i< trials; i++)
    {
        x = rand();
        y = rand();
        if((x*x + y*y) < 1.0) {
            pthread_mutex_lock(&mutex)
            inTheCircle++;
            pthread_mutex_unlock(&mutex);
        }
    }
    pthread_mutex_lock(&mutex)
    inTheCircle += myinTheCircle;
    pthread_mutex_unlock(&mutex);
}
OpenMP vs. Pthreads

• OpenMP is great for parallel loops
  • And for many simple situations with just “#pragma omp parallel” as well
• But when there is complicated synchronization, and performance is important, pthreads are (currently) better
• However, pthreads are not available on all machines/OS’s
  • Especially Windows
Performance Oriented Programming in Pthreads

• Pthreads as used in OS programming don’t need to be as performance oriented as what we need in HPC
  • E.g., “synchronizing” every few microseconds
    • I.e., exchanging data or waiting for signals

• Improving performance:
  • Always use affinity
  • Decide the number of pthreads to avoid any over-subscription and use SMT only if memory bandwidth (and floating point intensity) permit
  • Minimize barriers, using point-to-point synchronization as much as possible (say, between producer and a consumer, as in Gauss-Seidel)
  • Reduce cross-core communication (it’s much better to use the data produced on one core on the same core if/when possible)
  • Locks cause serialization of computation across threads
Threads and Resources

• Suppose you are running on a machine with K cores
  • Each core may have 2 “hardware threads”
  • This is often called hyperthreading on SMT (symmetric multi-threading)

• How many pthreads can you create?
  • Unlimited (well ... the system may run out of resources like memory)
  • Can be smaller or larger than K

• Which cores does each thread run on?
  • By default: any (i.e., OS suspends each running thread every few ms, and runs another thread)
Affinity

• Which cores does each thread run on?
  • By default: any (i.e., OS suspends each running thread every few ms, and runs another thread)
  • Even if you have fewer threads than the hardware threads
  • But that’s bad for cache locality
    • Caches will be polluted by the work by other threads ... you will do a “cold” start almost always when you get scheduled every few ms
  • Pthreads provide a way for “binding” threads to hardware resources for this purpose
Pthread Affinity

• Affinity (or pinning) assigns a thread to a location, e.g., core
• Performance impacts (OS processes, cache, interference, migration)
• Set of one (or more) legal core(s)
• Can use topo info to pin to processors, sockets, NUMA domains, etc.

• Example:

```c
...  
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(PEnum, &cpuset); // can be called multiple times
pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset)
...```
• Suppose all threads are doing the following
  • The work in `dowork()` is $t_w$
  • The time in `critical` is $t_c$
  • The serialization cost becomes a problem as the number of threads increases, but can be small up to $\frac{t_w}{t_c}$.

```
for i = 0, N
    dowork();
    lock(x);
    critical ...;
    unlock(x)
```
Non-serialization

• With a large number of threads, every thread ends up waiting at every iteration