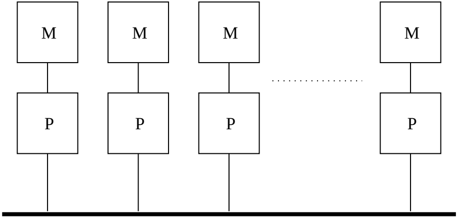
Threads intro

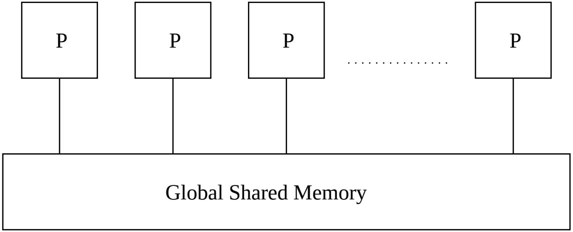
- We have mainly looked at writing parallel programs where each task has its own private memory space
- This model works well on distributed memory systems (clusters)
- You need to explicitly send data from one process to another



Threads intro

- In a machine with multiple processors it can be beneficial to have several processors all working on the same data set
- We use lightweight processes or threads to achieve this aim
- All programs we have written so far have only a single thread of execution
- Processes can have multiple threads executing at the same time
- These threads share the same global memory (heap) but each has its own stack

Threads intro



- Each processor in the system can access any physical memory location
- Some memory may be further away (NUMA)
- Sample machines
  - SGI Ultraviolet
  - Any multi-socket Intel/AMD server

Benefits of threads

- Portability — easy to move code from a serial to a parallel machine
- Latency — we can hide I/O and memory latencies by having one thread do work while another is waiting
- Load balancing — by allocating each task to a separate thread the system can load balance. No need for the programmer to explicity schedule interaction
- Ease of use — threads are generally easier to write code for than MPI
- Future proof — modern machines are shipping with greater numbers of CPU cores. Threads will play a major role
- Major downside — need a machine with multiple cores to see any benefit

Creating threads

- Need to include `pthread.h`
- Need to link with `-lpthread`
- Use `pthread_create(...)` to create a new thread of execution
- Four arguments
  - pthread_t *handle — an identifier for the thread
  - const pthread_attr_t *att — attributes of the thread. Can use NULL
  - void * (*function)(void *) — the “main” function of the thread
  - void * arg — a single argument to the function. Again NULL acceptable

Ending threads

- All threads are terminated when `main()` exits
- An individual thread terminates when you call `pthread_exit` within the thread
- A thread also ends if you reach the end of the thread's main function
- Threads can be terminated by other threads calling `pthread_cancel`

Waiting for threads

- Need a way to wait for the threads to finish properly
- Use `pthread_join(...)`
- Two arguments
  - pthread_t *handle — the identifier of the thread you are waiting for
  - void **ptr — the value passed to `pthread_exit`
- Note multiple threads can't join a single thread
- Threads can become `detached` and cannot be joined. This should generally be avoided

Other management calls

- `pthread_self` gets your own thread id
- `pthread_equal` compares two threads. Cannot just use normal == comparison as thread objects are not basic data types
- `pthread_once(ctrl, func)` executes `func` exactly one time per process. Every other call to the function has no effect.
- Often used for an initialization function

Shared Variables

- The best thing about threads is that the communications just happens
- The worst thing is that you need to synchronise the threads so they don't mess up the shared data
- Consider the following code

```
if( localval > globalval)
        globalval = localval;
```

- If globalval is 100 and you have two threads with localval set to 150 and 200
- If both execute this section of code simulataneously then you might end up with globalval set to either 150 or 200

Mutexes

- The main method of avoiding these *race conditions* is using mutexes (mutual exclusions)
- A mutex has two states — locked and unlocked
- Only one thread can have a lock at a time
- If you try to lock a mutex that is already locked by another thread your execution is blocked
- When the other thread unlocks the lock then you can get the lock and use the shared data

```
Get_Lock();
if( localval > globalval)
        globalval = localval;
Release_Lock();
```

Mutexes

- The pthread library defines a new data type called `pthread_mutex_t` to handle mutex locks
- Use the function `pthread_mutex_lock( *lock)` to acquire the lock
- If the mutex is already locked you will wait until it becomes unlocked
- Need to be careful about circular deadlocks!
- Use the function `pthread_mutex_unlock( *lock)` to release the lock
- Before using a mutex it needs to be initialised
- Use `pthread_mutex_init( *lock, *attrib)` to initialise. Just use NULL for the attributes for the moment.

Conditional Locks

- Using two mutexes we can avoid most deadlocks
- We need to request them in different orders and use `pthread_mutex_trylock` to try to grab the locks

Thread 1	Thread 2

<pre>pthread_mutex_lock(lock1);
pthread_mutex_lock(lock2);
 	
/* Do calculations */
 	
pthread_mutex_unlock(lock2);
pthread_mutex_unlock(lock1);</pre>
	<pre>while(1) {
	  pthread_mutex_lock(lock2);
	  if(pthread_mutex_trylock(lock1) == 0)
	    break;
	  pthread_mutex_unlock(lock2);
	}
	
	/* Do calculations */
	
	pthread_mutex_unlock(lock1);
	pthread_mutex_unlock(lock2);</pre>

Thread attributes

- Specific attributes of the thread can be set at create time using the `pthread_attr_t` object
  - Detached or joinable
  - Scheduling information
  - Stack details (size, address, overflow size)
- Need to use the `pthread_attr_init` and `pthread_attr_destroy` functions to managed the attribute object

Thread Safety

- Library calls often use a global data structure (seed of RNG)
- With multiple threads calling routines the data structure may not remain consistent
- The library needs to put in some synchronisation to prevent corruption
- This makes the library *thread safe*
- If using libraries and threads, make sure they are thread safe
- Otherwise you will have to serialize the function calls

Debugging Threads

- As always start debugging using printf
- Can do thread debugging in gdb
- `break function thread 2`
- This will stop the execution once thread 2 calls `function`
- Usually all threads will stop execution. However, on some platforms there is a mode where all other threads continue executing in the background.
- Be careful stepping through code — other threads may execute more than one instruction in the time it takes the thread of interest to step.