Common Parallel Programming Problems

Too Many Threads

- If more threads than processors, round-robin scheduling is used
  - Scheduling overhead degrades performance
  - Sources of overhead
    - Saving and restoring registers – negligible
    - Saving and restoring cache state – when run out of cache, threads tend to flush other threads cached data
    - Thrashing virtual memory
    - Convoying of threads waiting on a lock, waiting on a thread whose time-slice has expired and which is still holding the lock
- Solution: limit number of threads to
  - Number of hardware threads (cores or hyper-threaded cores) or
  - Number of caches

Which threads cause overhead

- Only runnable threads cause overhead – blocked threads do not
- Helps to separate compute and I/O threads
  - Compute threads are running most of the time and number should correspond to number of cores – they may feed from task queues
  - I/O threads may be blocked most of time and are not a significant factor in having too many threads
- Useful Hints:
  - Let OpenMP choose number of threads
  - Use a thread pool

Useful Practices for Building Efficient Task Queues

- Let OpenMP do it – OpenMP will try to use the optimal number of threads
- Use a thread pool – a set of long lived software threads
  - Eliminates initialization overhead
  - Software thread finish tasks before starting another
  - Windows has routine QueueUserWorkItem, Java has class executor for defining tasks. POSIX has no standard thread pool support
- Write your own task scheduler – only if you are an expert!
  - Preferred method is *work stealing*
  - Each thread has own pool, but steals from another's if it runs out of tasks in order to balance load
  - Bias is to steal large tasks e.g. Cilk scheduler

Data Race

	Thread 1	Thread 2
	t = x	u = x
	x = t + 1	u = x + 2

	Thread 1	Thread 2	x	u
	t = x		0	
		u = x		0
		u = x + 2		2
	x = t + 1		1	

Data Race

	Thread 1	Thread 2	x	u
		u = x	0	0
	t = x		0	
	x = t + 1		1	
		x = u + 2	2	

	Thread 1	Thread 2	x	u
	t = x		0	
	x = t + 1		1	
		u = x		1
		x = u + 2	3	

Data Race is often disguised

Thread 1	Thread 2	
x +=1	x +=2	Expands into separate read and write
a[i] += 1	a[j] += 2	i and j could be equal
*p += 1	*q += 2	p and q might point to same location
Foo (1)	Foo (2)	Foo might add its arg to a shared variable
add [edi],1	add [edi],2	Even at assembler level could be expanded

Care with updates of shared structures etc

- If threads are reading a location that is updated by another thread asynchronously must be careful the write is atomic
  - updates of structures are often done a word or field at a time
  - Types longer than word size might not be written atomically
  - Misaligned loads and stores may not be atomic
  - If access straddles cache line, it becomes 2 separate accesses

Synchronization at too low level

- Suppose we are trying to create a list in which each key appears only once
- List operations such as list.contains and list.insert are atomic
- The instructions to check if key is in list and if not insert

```
if ( !list.contains(key) )
        list.insert(key);
```

- Will not guarantee only one copy of key in list
- Need higher level lock
- Lower level lock becomes redundant

Deadlock – necessary conditions

1. Exclusivity : Access to each resource is exclusive
2. Hold-and-Wait : Thread is allowed to hold one resource while requesting another
3. Non-preemption : No thread is willing to relinquish a resource
4. Cyclic : there is a cycle of threads where each resource is held by one thread and requested by another

Solutions

- Replicate the resource – give each thread its own private copy
  - copies can be merged at end
  - Also improves scalability
- Always acquire resources in same order e.g.
  - Alphabetical
  - For linked list, could be list order; for tree, order of preorder traversal
  - Nested structures – proceed from outside to insider
  - If other options absent, sort locks by address
    - Threads need to know all locks before acquiring any
- Large projects should avoid software components holding locks while calling outside components

More Solutions

- Avoid Hold-and-wait by using "try lock"
- Example of using try lock to acquire 2 locks

```
Void acquireTwoLocksViaBackoff( lock& x, lock& y) {
    for( int t=1; ; t*=2) {
            acquire x
            try to acquire y
            if ( y was acquired) break;
            release x
            wait for random amount of time between 0  and t
     }
}
```

Live Lock Problem – and exponential backoff

- Occurs when threads continue to conflict and then back-off
  - Reason for exponential backoff in example on previous slide
  - Waits for random time chosen from interval that doubles
  - Negative of backoff schemes is that they are not *fair*
    - a particular thread is not guaranteed to make progress

Heavily Contended Locks

- If threads arrive at a lock faster than they can execute the corresponding critical section, they block there
  - often called convoying
- Even worse for fair locks, since, if a thread falls asleep, all other have to wait
-

Priority Inversion

- Some implementations allow thread priorities
  - Higher priority threads get preference
- Can have low priority threads block high priority ones
  - e.g. if low priority acquires a lock and high priority one is waiting, then a medium priority thread could be run in preference to the low priority one
  - Actually happened on Mars Pathfinder mission
  - Can be solved by increasing priority of blocking thread
    - Called *priority inheritance*
    - supported by Windows threads
  - Other option is *priority ceilings*
    - When thread acquires mutex, priority is increased to ceiling (highest possible priority) which holds mutex
  - Both are optional in pthreads
    - If exist can be set by pthread_mutexattr_setprotocol

Solutions for Heavily Contended Locks

- Locks inherently serialize threads
  - Faster locks only improve performance by constant factor, don't improve scalability
  - Solutions
    1. Preferred solution : replicate resource and eliminate lock
    2. Partition resource and use separate locks for each partition
       - Ex: hash table – need to prevent race condition where multiple threads try to do insertion
       - If use one lock for table a lot of contention
       - Can partition the table into subtables, each with own lock - reduce contention
    3. Fine-grained locking –
       - e.g. hash table which is array of buckets could have lock per bucket
       - Easy if fixed number of buckets
       - A lot of overhead if buckets small
       - More complicated if number of buckets can grow

Hash Table with dynamic number of buckets

- To resize the array, may need to exclude all threads
  - Similar to reader/writer lock problem
  - Table has array descriptor with array size and location
    - Protected by reader/writer mutex
  - Each bucket has own plain mutex
  - To access a bucket, a thread
    1. Acquires reader lock on array descriptor
    2. Acquires lock on bucket's mutex
  - To resize the array a thread
    - Acquires writer lock on array descriptor
  - Multiple threads can access different buckets concurrently
  - Disadvantage: must acquire 2 locks – overhead may negate reduction in contention

Problems with Reader-Writer locks

- Can reduce contention, if writers are infrequent
- If rate of incoming readers too high, may suffer memory contention

Non Blocking Algorithms

- Non-blocking algorithms: stopping thread does not prevent progress in the rest of system
- Different Guarantees
  - *Obstruction freedom* : thread makes progress if there is no contention; livelock possible; exponential backoff can solve
  - *Lock Freedom* : system as a whole makes progress
  - *Wait freedom* : every thread makes progress, even when faced with contention. Rare.
- Advantages; immune to lock contention, priority inversion, convoying
- Based on atomic operations