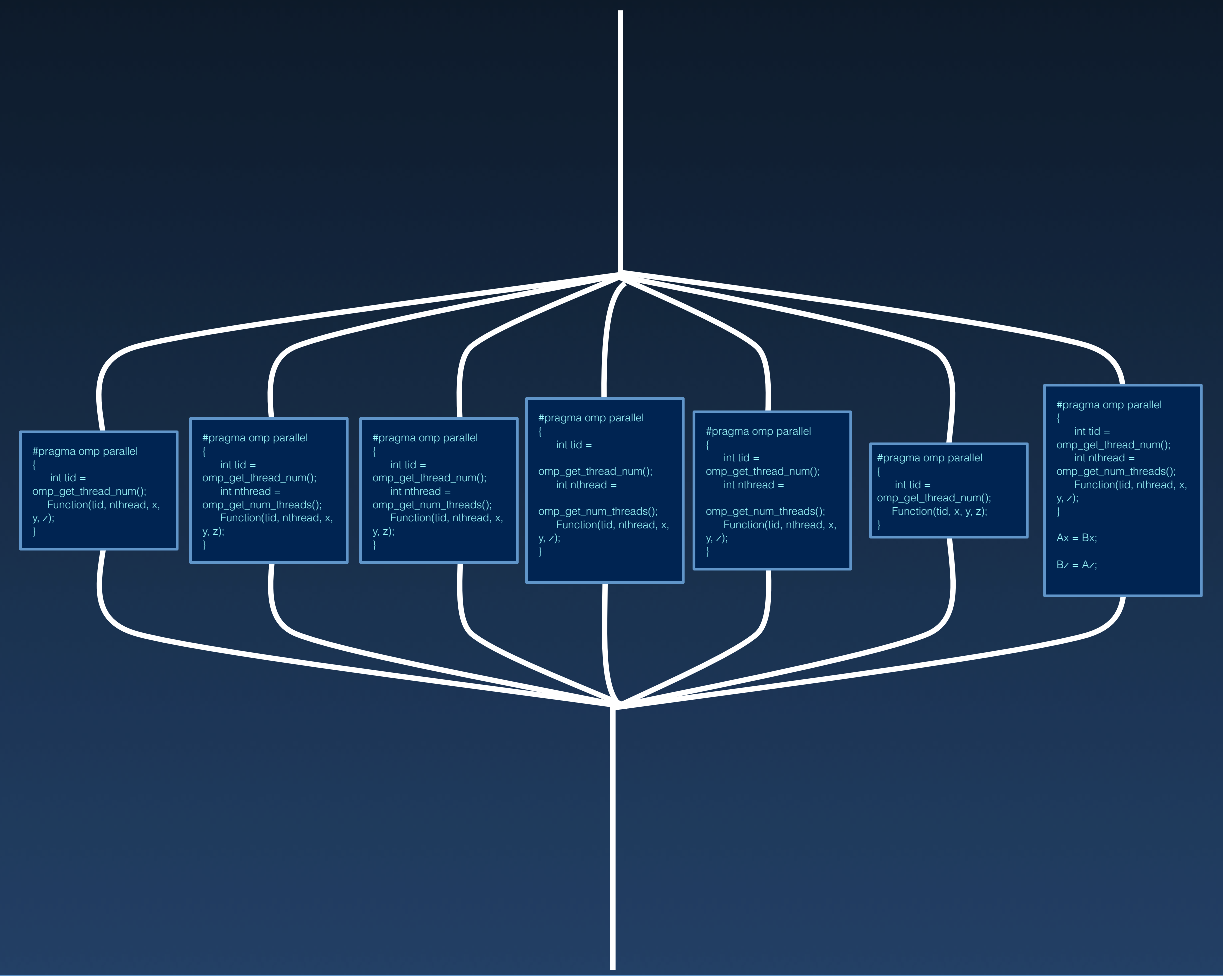
COL380

Introduction to
Parallel & Distributed Programming

Hello OpenMP

- Fork-Join programming model
  - ➡ Teams of threads
- Shared Memory programming model
- Well defined memory model
- High level synchronization



```
#pragma omp parallel
{
	int tid = omp_get_thread_num();
	int nthread = omp_get_num_threads();
	Function(tid, nthread, x, y, z);
}
```

OS Threads

- “Light-weight” process
  - ➡ Execution context: Stack, Registers
  - ➡ Context switch of Registers, SP, PC
    - Not caches, virtual page tables
- Kernel threads vs User threads (Light-weight thread)
  - ➡ Kernel threads are scheduled by OS
  - ➡ User threads when no context switching is required

Parallel Program

Multiple Threads of Execution

OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands
OP operands	OP operands

Shared Variable Store

(May not share actual memory)

- How are they instantiated?
- Execution and memory model
- How do the interact?

Separate from HW
(Who does the execution?)

Programming Models

- Shared Memory model
- Distributed Memory model
- Task based model
- Work-queue model
- Stream processing model
- Map-reduce model
- Client-server model

Example Models

```
void *threadFn
{
    // Do somet
}
...
{
    int arg;
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, threa
    // Continue doing other things
    pthread_join(thread_id, NULL);
}
```

```
int threadFn(int arg)
{
    // Do something with ‘arg’
}
...
{
    int x;

    x = spawn threadFn(1);
    // Continue doing other th
    sync;
}
```

```
func threadFn(id int, arg string, wg *sync.WaitGroup,
                                              ch chan int)
{
    defer wg.Done()
    // Do thing ‘id’ with ‘arg’
                      x <- ch
}
...
{
    var wg sync.WaitGroup
                      chA := make(chan int)
    wg.Add(1)
    go threadFn(1, “a string”, &wg, chA)
    // Continue doing other things
    wg.Wait()
                      chA <- result
}
```

Subodh Kumar

Loop Parallelization

- High level language

  for i=0 to N
      a[i] = *f*(b[i], c[i], d[i])

  ➡ Derive parallelism

  ➡ Generate threads and map to processors

  ➡ Addresses for a, b, c, d accessible to all

    - also the code for f

  ➡ Map i to thread ID

  ➡ Impact on cache coherence?

- Generate new threads of control
  - ➡ function per thread, work sharing construct
- Synchronize
- Specify variable scopes
  - ➡ Maybe, for an arbitrary group of threads
- Ways to map threads to processor?
  - ➡ May have more threads than processors
  - ➡ Scheduling hints

OpenMP Execution Model

- Encountering thread creates a team:
  - ➡ Itself (master) + zero or more additional threads.
- Applies to structured block immediately following
  - ➡ Each thread executes separately the code in {}
    - But, also see: Work-sharing constructs
- There's an implicit barrier at the end of block
- Only master continues beyond the barrier
- May be nested
  - ➡ disabled by default

Subodh Kumar

- Notion of temporary view of memory
  - ➡ Allows local caching
  - ➡ Need to relax consistency model
- Supports threadprivate memory
  - ➡ global scope
- Variables declared before parallel construct:
  - ➡ Shared by default
  - ➡ May be designated as private
    - n-1 copies of the original variable is created Initialized if requested

- Shared:
  - ➡ Heap allocated storage
  - ➡ Static data members, const variable (no mutable member)
- Private:
  - ➡ auto Variables declared in a scope inside the construct
  - ➡ Loop variable in for construct
- Others are shared by default

Can declare private or change default

- Arguments passed by reference inherit from original

Relaxed Consistency

- Unsynchronized access:
  - ➡ If two threads write to the same shared variable the result is undefined
  - ➡ If a thread reads and another writes, the read value is undefined
- Memory atom size is implementation dependent
- Flush x,y,z .. enforces consistency

Thread A	Thread B
X = 5	Flush(X)
…	…
Flush(X)	v = X

Compiler Re-ordering

a = b = 0

Thread A	Thread B
b = 1	a = 1
flush(b) flush(a)	flush(a) flush(b)
flush(a,b)	flush(a,b)
if (a == 0) {	if (b == 0) {
critical section	critical section
}	}

#pragma omp flush(a,b)

Subodh Kumar

1-Producer
1-Consumer

int data, flag = 0;

Thread 0	Thread 1

// Produce data
data = 42; produce
// Flush
F #pragma omp flush(flag, data)
// Set flag to signal Thread 1
A flag = 1;
// Flush
#pragma omp flush(flag)
	// Busy-wait until flag is signalled
	#pragma omp flush(flag)
	while (flag != 1) {
	    #pragma omp flush(flag) data
	}
	#pragma omp flush(flag, data)
	// Consume data
	printf(data=%d\n", data); consume

"1" ⇒ A has started and hence F has happened

Subodh Kumar

Acquire & Release Flush

a = b = 0

Thread A	Thread B
b = 1	a = 1
#pragma omp flush release	#pragma tmp flush acquire
if (a == 0) {	if (b == 0) {
critical section	critical section
}	}