L15: Dynamic Scheduling

Administrative

- STRSM due March 23 (EXTENDED)
- Midterm coming
  - In class March 28, can bring one page of notes
  - Review notes, readings and review lecture
  - Prior exams are posted
- Design Review
  - Intermediate assessment of progress on project, oral and short
  - In class on April 4
- Final projects
  - Poster session, April 23 (dry run April 18)
  - Final report, May 3

Schedule of Remaining Lectures

March 19 (today):	Dynamic Scheduling
March 21:	Sorting
March 26:	Midterm Review
April 2:	Tree Algorithms
April 4:	Design Reviews
April 9:	Fast Fourier Transforms or TBD
April 11:	TBD
April 16:	Open CL
April 18:	Poster Dry Run
April 23:	Public Poster Presentation

Sources for Today's Lecture

- "On Dynamic Load Balancing on Graphics Processors," D. Cederman and P. Tsigas, Graphics Hardware (2008).

http://www.cs.chalmers.se/~tsigas/papers/GPU_Load_Balancing-GH08.pdf

- "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," P. Tsigas and Y. Zhang, SPAA 2001.

(more on lock-free queue)

- Thread Building Blocks

http://www.threadingbuildingblocks.org/

(more on task stealing)

Motivation for Next Few Lectures

- Goal is to discuss prior solutions to topics that might be useful to your projects
  - Dynamic scheduling (TODAY)
  - Tree-based algorithms
  - Sorting
  - Combining CUDA and Open GL to display results of computation
  - Combining CUDA with MPI for cluster execution (6-function MPI)
  - Other topics of interest?
- End of semester
  - CUDA 4 Features
  - Open CL

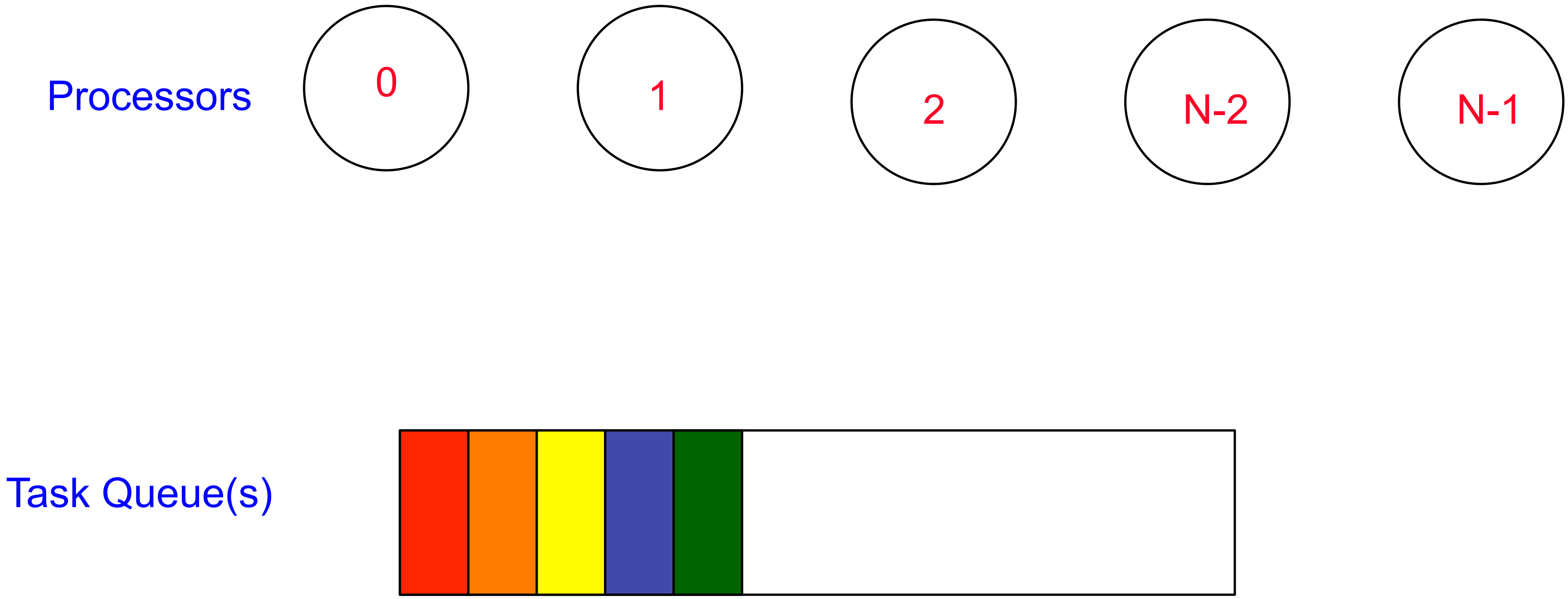
Motivation: Dynamic Task Queue

- Mostly we have talked about how to partition large arrays to perform identical computations on different portions of the arrays
  - Sometimes a little global synchronization is required
- What if the work is very irregular in its structure?
  - May not produce a balanced load
  - Data representation may be sparse
  - Work may be created on GPU in response to prior computation

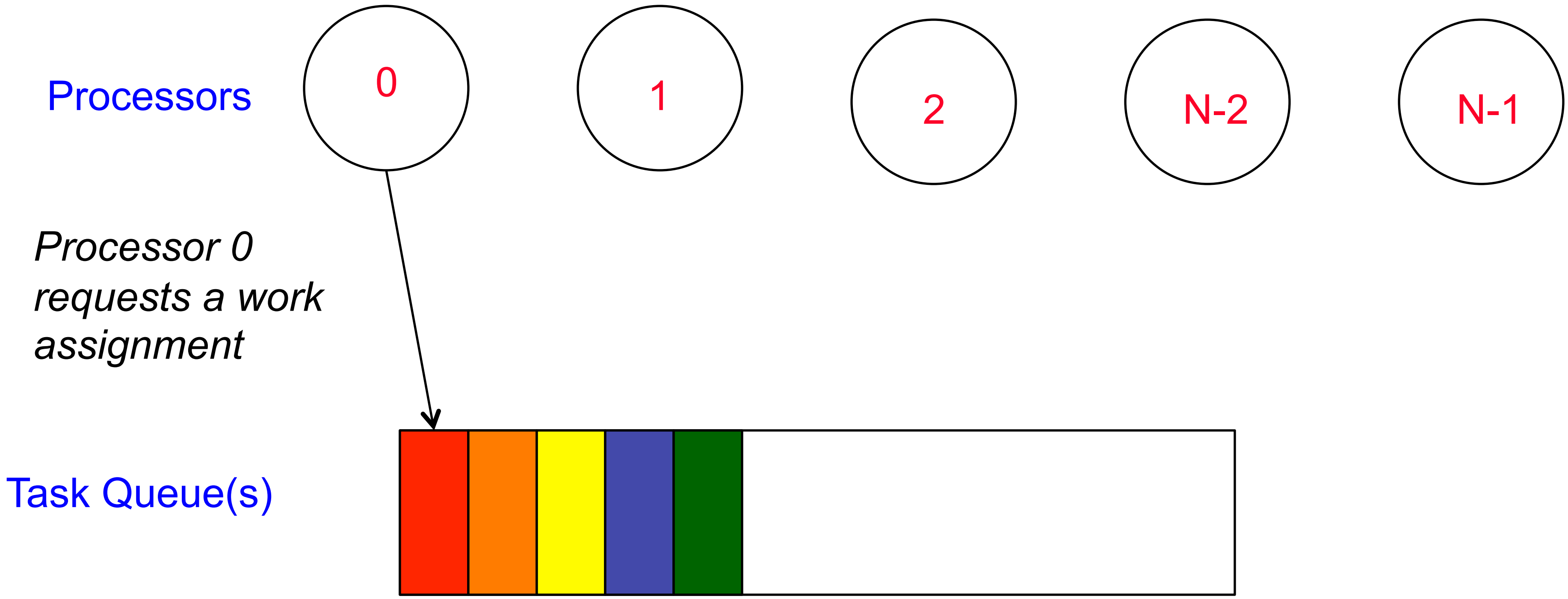
Dynamic Parallel Computations

- These computations do not necessarily map well to a GPU, but they are also hard to do on conventional architectures
  - Overhead associated with making scheduling decisions at run time
  - May create a bottleneck (centralized scheduler? centralized work queue?)
  - Interaction with locality (if computation is performed in arbitrary processor, we may need to move data from one processor to another).
- Typically, there is a tradeoff between how balanced is the load and these other concerns.

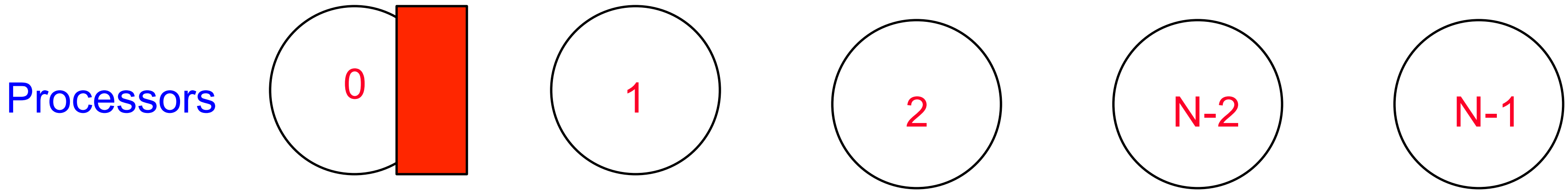
Dynamic Task Queue, Conceptually



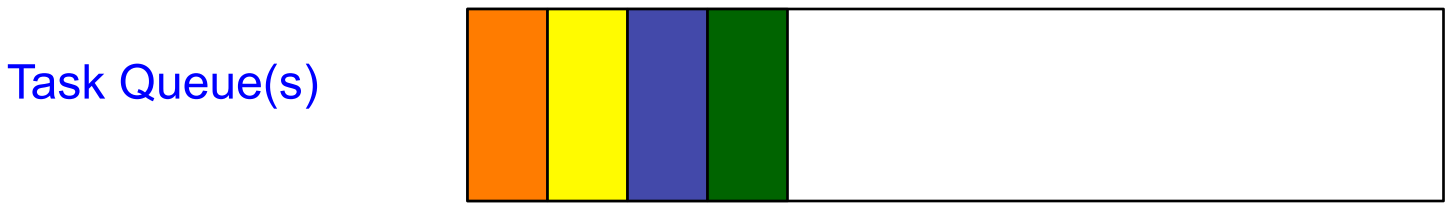
Dynamic Task Queue, Conceptually



Dynamic Task Queue, Conceptually



First task is assigned to processor 0 and task queue is updated



Just to make this work correctly, what has to happen?
Topic of today's lecture!

Constructing a dynamic task queue on GPUs

- Must use some sort of atomic operation for global synchronization to enqueue and dequeue tasks
- Numerous decisions about how to manage task queues
  - One on every SM?
  - A global task queue?
  - The former can be made far more efficient but also more prone to load imbalance
- Many choices of how to do synchronization
  - Optimize for properties of task queue (e.g., very large task queues can use simpler mechanisms)
- Static vs. dynamic scheduling
  - In batches vs. one-by-one
- All proposed approaches have a statically allocated task list that must be as large as the max number of waiting tasks

Suggested Synchronization Mechanism

```
// also unsigned int and long long versions

int atomicCAS(int* address, int compare, int val);
```

reads the 32-bit or 64-bit word old located at the address in global or shared memory, computes (old == compare ? val : old), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old (Compare And Swap). 64-bit words are only supported for global memory.

```
__device__ void getLock(int *lockVarPtr) {
while (atomicCAS(lockVarPtr, 0, 1) == 1);
}
```

Synchronization

- Blocking
  - Uses mutual exclusion to only allow one process at a time to access the object.
- Lockfree
  - Multiple processes can access the object concurrently. At least one operation in a set of concurrent operations finishes in a finite number of its own steps.
- Waitfree
  - Multiple processes can access the object concurrently. Every operation finishes in a finite number of its own steps.

Slide source: Daniel Cederman

Load Balancing Methods

- Static Blocking Task Queue
- Dynamic Blocking Task Queue
- Non-blocking Task Queue
- Task Stealing

Slide source: Daniel Cederman

Blocking Static Task Queue (Simplest)

```
function DEQUEUE(q, id)
  return q.in[id] ;
function ENQUEUE(q, task)
  localtail ← atomicAdd (&q.tail, 1)
  q.out[localtail ] = task
function NEWTASKCNT(q)
  q.in, q.out , oldtail , q.tail ← q.out , q.in, q.tail, 0
  return oldtail
procedure MAIN(taskinit)
  q.in, q.out ← newarray(maxsize), newarray(maxsize)
  q.tail ← 0, Tbid ← 0
  ENQUEUE(q, taskinit )
  blockcnt ← NEWTASKCNT (q)
  while blockcnt != 0 do
    run blockcnt blocks in parallel
      t ← DEQUEUE(q, ++Tbid )
      subtasks ← doWork(t )
      for each nt in subtasks do
        ENQUEUE(q, nt )
   blockcnt ← NEWTASKCNT (q)
```

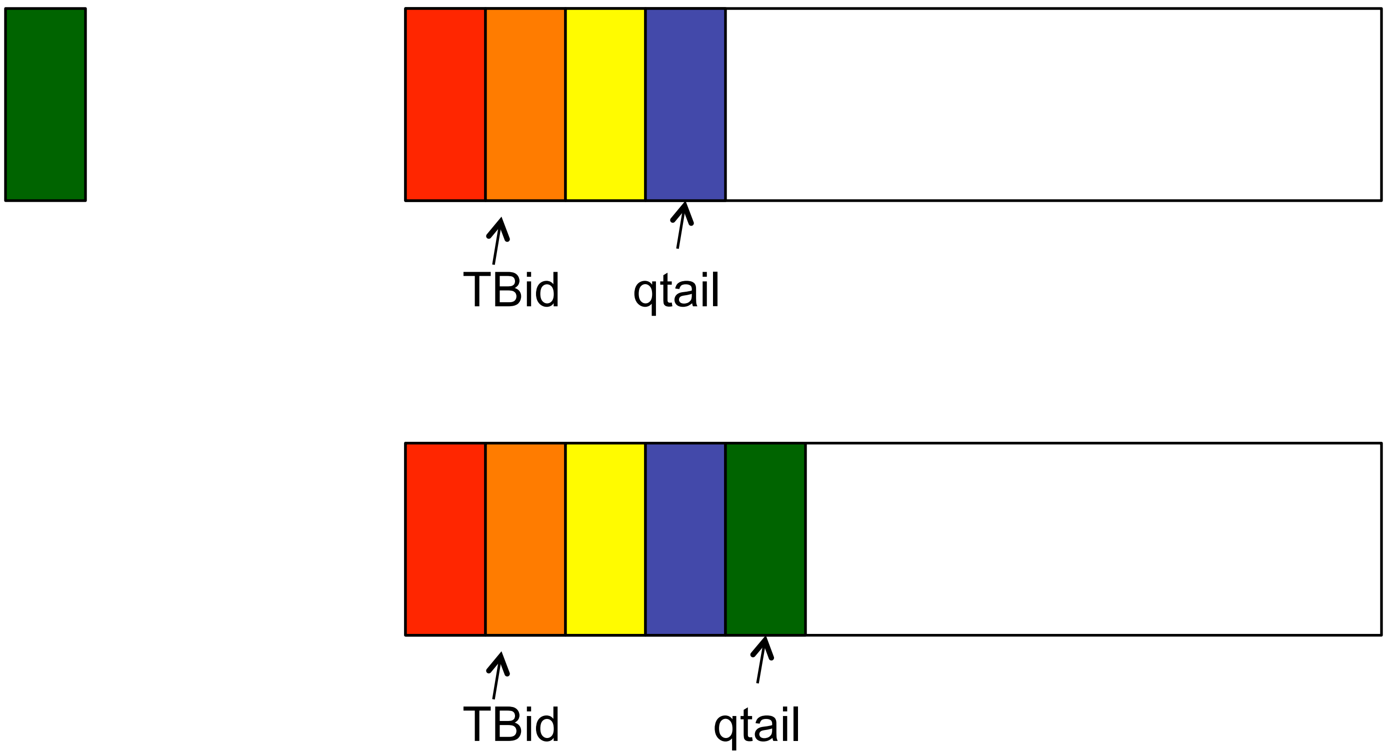
Two lists:
  q_in is read only and not synchronized
  q_out is write only and is updated atomically

When NEWTASKCNT is called at the end of major task scheduling phase, q_in and q_out are swapped

Synchronization required to insert tasks, but at least one gets through (wait free)

Blocking Static Task Queue

ENQUEUE



Blocking Dynamic Task Queue

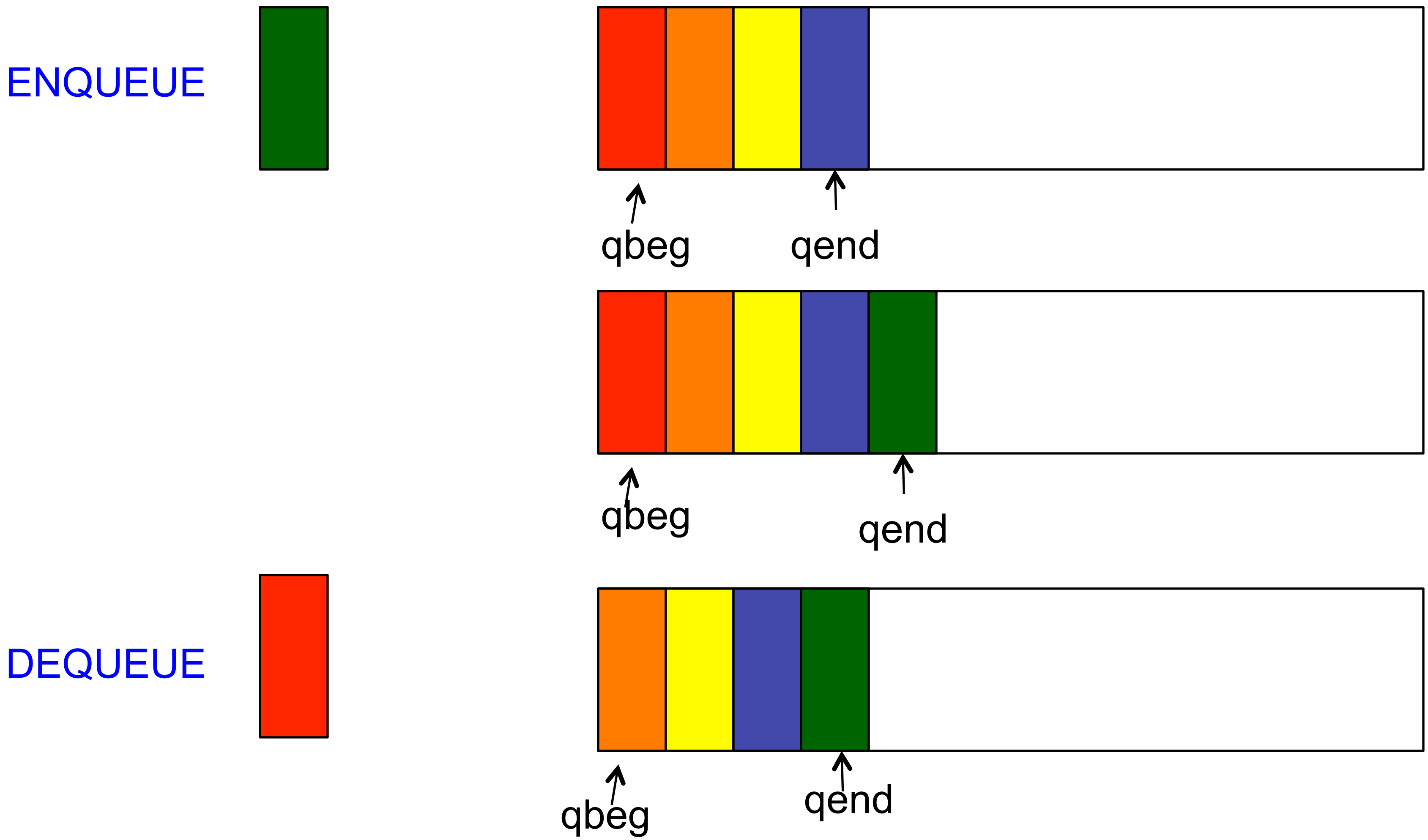
```
function DEQUEUE(q)
  while atomicCAS(&q.lock, 0, 1) == 1 do;
  if q.beg != q.end then
    q.beg ++
    result ← q.data[q.beg]
  else
    result ← NIL
  q.lock ← 0
  return result

function ENQUEUE(q, task)
  while atomicCAS(&q.lock, 0, 1) == 1 do;

  q.end++
  q.data[q.end ] ← task
  q.lock ← 0
```

Use lock for both adding and deleting tasks from the queue.

All other threads block waiting for lock.

Potentially very inefficient, particularly for fine-grained tasks

Blocking Dynamic Task Queue



Lock-free Dynamic Task Queue

```
function DEQUEUE(q)
  oldbeg ← q.beg
  lbeg ← oldbeg
  while task = q.data[lbeg] == NIL do
    lbeg ++
  if atomicCAS(&q.data[lbeg], task, NIL) != task then
    restart
  if lbeg mod x == 0 then
    atomicCAS(&q.beg, oldbeg, lbeg)
  return task
function ENQUEUE(q, task)
  oldend ← q.end
  lend ← oldend
  while q.data[lend] != NIL do
    lend ++
  if atomicCAS(&q.data[lend], NIL, task) != NIL then
    restart
  if lend mod x == 0 then
    atomicCAS(&q.end , oldend, lend )
```

Idea:
At least one thread will succeed to add or remove task from queue

Optimization:
Only update beginning and end with atomicCAS every x elements.

Task Stealing

- No code provided in paper
- Idea:
  - A set of independent task queues.
  - When a task queue becomes empty, it goes out to other task queues to find available work
  - Lots and lots of engineering needed to get this right
  - Best implementions of this in Intel Thread Building Blocks and Cilk

General Issues

- One or multiple task queues?
- Where does task queue reside?
  - If possible, in shared memory
  - Actual tasks can be stored elsewhere, perhaps in global memory

Remainder of Paper

- Octtree partitioning of particle system used as example application
- A comparison of 4 implementations
  - Figures 2 and 3 examine two different GPUs
  - Figures 4 and 5 look at two different particle distributions