Outline

This presentation covers the following topics

- Session 1 – Cilk and the Lucata API
  - Basic programming
  - Data distribution
- Session 2 – Lucata Workflow
  - X86 Debugging
  - Simulation
  - Hardware
- Session 3 – Measuring Performance
  - Timing Hooks
  - Profiling
- Session 4 – Coding Optimizations
  - Machine-specific coding
  - Parallel computation
- **Section 5**
  - ***Advanced topics***

Slides originally developed by Janice McMahon, Lucata Corporation

LUCATA

Advanced Topics

Patterns for spawning threads

Balancing Parallelism and Overhead

- Number of threads vs work per thread
  - Enough parallelism to keep the cores busy and mask migrations
  - Enough work per thread to offset thread overhead
- Target ~64 threads per core
  - Larger systems and/or more migrations may require more threads to offset those in transit
  - Maximum 512 threads per node

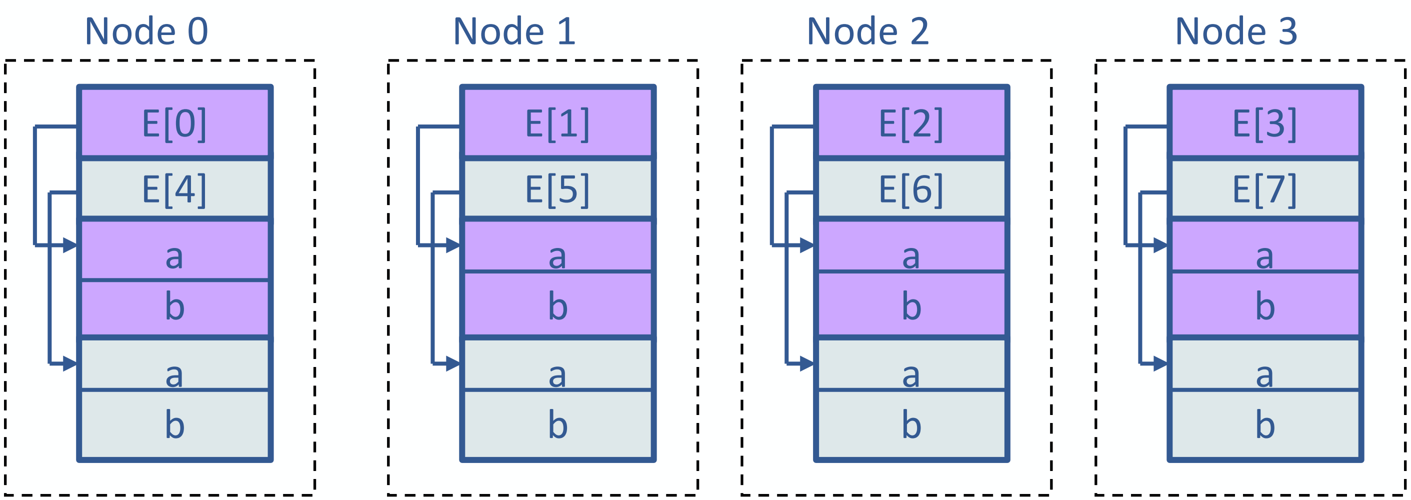
Common Issues

- Hotspots/Extra Migrations
  - Shared data found on single node
  - Spawns not distributed
- Wave parallelism
  - All threads move in lock step, not distributed
- Unbalanced/limited network usage
  - Sequential : only uses channel to neighbor e.g. BFS
  - Other non-uniform communication e.g. GUPS

"Wave" parallelism

```
#define N 8
#pragma grainsize = 4
cilk_for (long i=0; i<N; i++) {
        E[i]->a = i;
        E[i]->b = 0;
}
```

- Accessing a striped array in order will cause a migration on every visit
- Starting all threads on node 0 utilizes only 1 node at a time



LUCATA

Spawning Static Thread Teams

- Spawn a “team” of threads at each node
- Each thread has a “home” node
  - Threads may stay entirely local
  - Threads may migrate away then return for more work
- If work varies greatly, a work queue often performs better than assigning N elements to each thread
  - Automatically load balances
  - Reduces cost of spawn and sync by creating a single set of threads
  - Atomic increment used to grab next unit of work

Dynamic Spawning

- Traverse a data structure and spawn based on characteristics of the structure
- Example: BFS
  - Spawn a team of threads to process vertices
  - Dynamically spawn additional threads to process edges in parallel based on size of the edge list
    - Number of edges at each vertex is unknown and may vary greatly

Efficient Spawning

- Distribute spawns across the nodes then spawn additional local threads
- Recursively spawn threads in a tree-like fashion (for large # threads)
  - Allows parallel spawning rather than sequential
  - Reduces hotspot at a single node if spawning across multiple nodes
  - Used in cilk_for

Cilk_for Spawn Tree Parallelization

cilk_for (i=0; i < E; i++) {c[i] = a[i]+ b[i];};

spawn two threads

(i=0; i < E/2; i++);

(i=E/2; i < E; i++);

Each spawns 2 more threads

(i=0; i < E/4; i++);

(i=E/4; i < E/2; i++);

(i=E/2; i < 3*E/4;

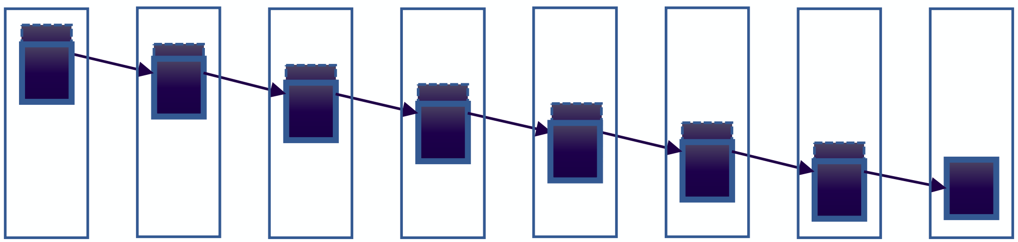
i++);

(i=3*E/4; i < E; i++);

...

LUCATA

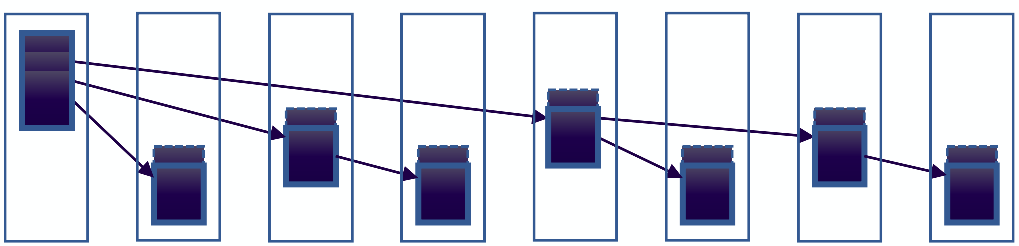
Recursive vs. Linear Spawn



Linear migration pattern with sequential spawns

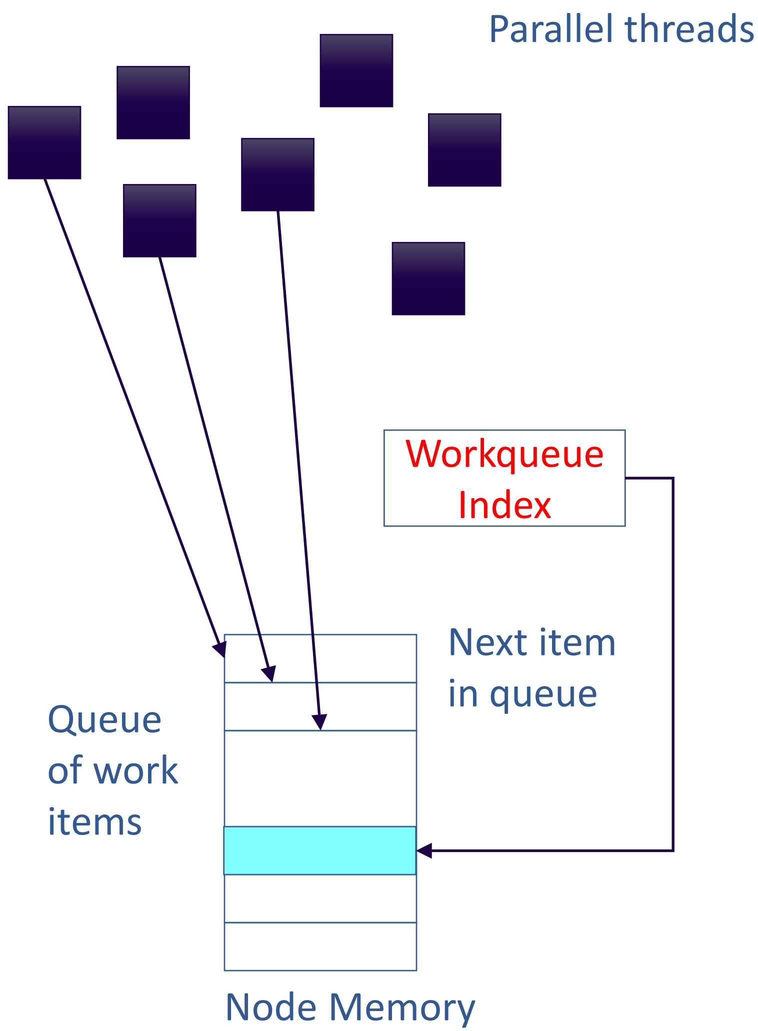
```
void linear() {
  for (long i = 0; i < NUM_NODES(); i++)
  { // rvar is a replicated variable
    long *p = mw_get_nth(&rvar, i);
    cilk_spawn_at (p) node_function();
  }
}
```

Tree migration pattern with parallel spawns



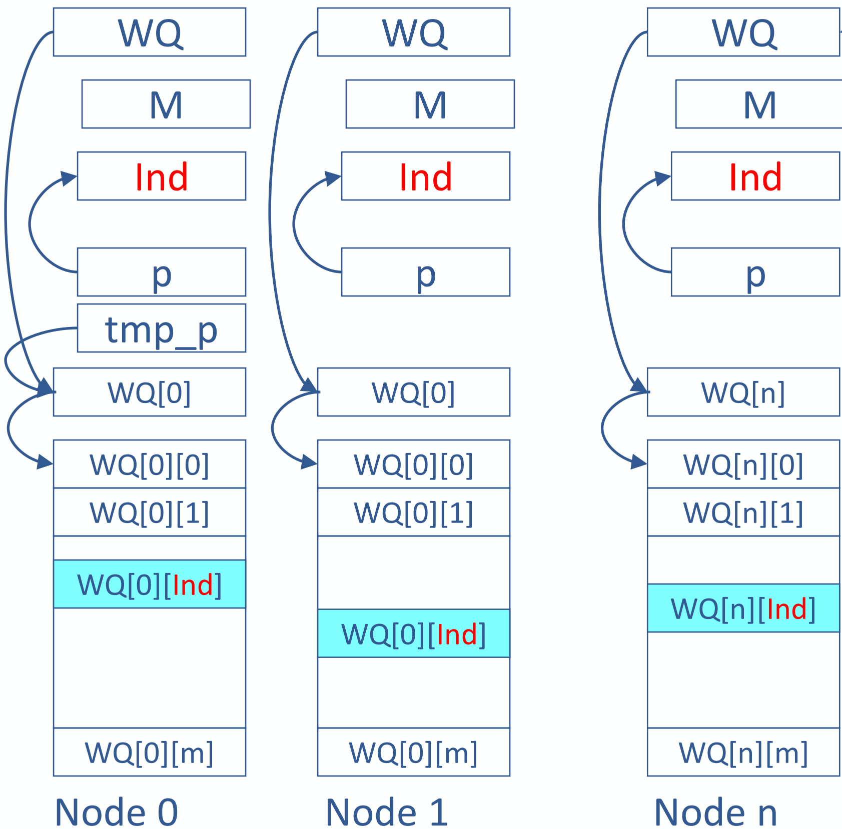
```
void recurse(long s, long e) {
  while (1) {
    long count = e - s;
    if (count <= 1) break;
    long m = s + count / 2;
    long *p = mw_get_nth(&rvar, m);
    cilk_spawn_at (p) recurse(m, e);
    e = m;
  } // assert (s ==  NODE_ID());
  node_function(s);
}
```

Work Queue in each Node



- Multiple threads are spawned on each node
- Each thread grabs the next work item from the queue for processing
- Next work item is designated by an index in node memory
- Threads grab items via an atomic increment of the index in that node
  - Uses the ATOMIC_ADDMS intrinsic
  - Increments memory and returns previous value

Data Structures for Work Queue

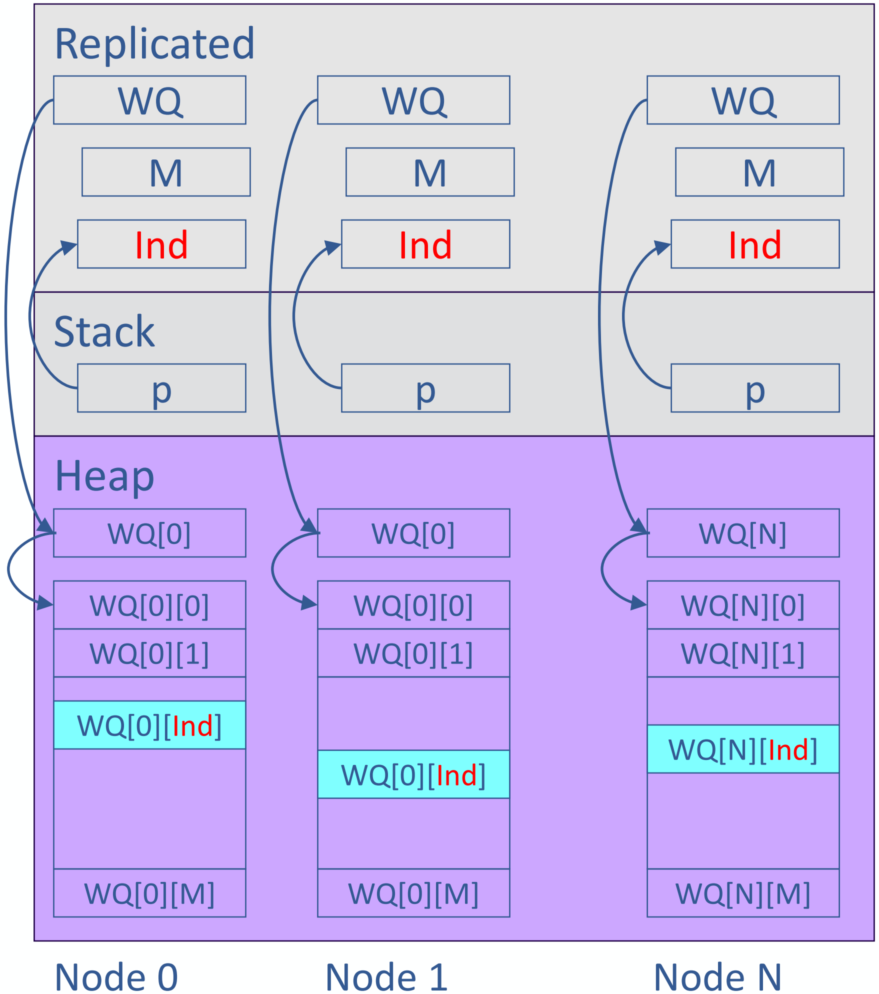


Pointer is constant and can be replicated to save migrations

```
replicated dtype **WQ;
replicated long M;
replicated long Ind;
main() {
  long N = NUM_NODES(); // 10 items/node
  mw_replicated_init(&M, 10);
  dtype **tmp_p = // work item type
    mw_malloc2d(N, M * sizeof(dtype));
  mw_replicated_init(&WQ, tmp_p);
  recurse(0, N);
}

void node_function(long nnum) {
 // assert (nnum ==  NODE_ID());
  long *p = mw_get_nth(&Ind, nnum);
  *p = 16; // 16 threads per node
  for (long i = 0; i < 16; i++)
    cilk_spawn worker(p, i, nnum);
}
```

Worker Thread Function



Each thread grabs the next work item using ATOMIC_ADDMS; done when all items processed

```
void worker(long *p, long off, long n) {
  // assert n == NODE_ID();
  while (off < M) {
    …
    // process item WQ[n][off]
    …
    off = ATOMIC_ADDMS(*p, 1);
  }
}
```

Replicated variable complexity

- Using replicated pointers reduces number of migrations over both argument values and global variable with function calls
- With thread spawns, arguments are packaged into thread state, no migration for access
- Register spills can cause extra migrations, eliminating the performance gain from using replicated variables
- Use of noinline keyword essential for preventing unwanted migrations due to register spills