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

Machine-specific Coding

Intrinsics including atomics, remotes, compare-and-swap

Atomic Arithmetic Operations

- Atomic arithmetic operations at the memory
- Four distinct “flavors” of each atomic that specify the value returned and resulting memory value

```
long ATOMIC_<INST>(long *p, long D);
<INST>: ADD/AND/OR/XOR/MAX/MIN
```

Mnemonic	Return Value	New Memory Value
<op>	Result	Unchanged
<op>S	Result	Orig. D Value
<op>M	Result	Result
<op>MS	Orig. Mem. Value	Result

ATOMIC_ADD Examples

- **ATOMIC_ADD(A, D)**
  - Reads value at A
  - Adds D
  - Returns result
  - Mem[A] is unchanged
- **ATOMIC_ADDS(A, D)**
  - Reads value at A
  - Adds D
  - Returns results
  - Mem[A] = D
- **ATOMIC_ADDM(A, D)**
  - Reads value at A
  - Adds D
  - Returns result
  - Mem[A] = result
- **ATOMIC_ADDMS(A, D)**
  - Reads value at A
  - Adds D
  - Returns original Mem[A]
  - Mem[A] = result

Remote Arithmetic Operations

- Significantly reduces migrations by performing atomic updates to memory without migrating the thread
- Sends only the data and operation to be performed
  - Consumes less than half the bandwidth of a typical thread migration
- Does not return a value
- Returns an ACK, may be turned off
- Remotes issued by a thread to the same location guaranteed to complete in order

```
void REMOTE_<INST>(long *p, long D);
<INST>: ADD/AND/OR/XOR/MAX/MIN
```

Memory Fence on Remote Update

- Used to wait for all remote updates to be acknowledged
- Prevents thread from continuing until all outstanding acknowledgements have been received
- Implicit FENCE before migration, thread suspend, etc.

```
void FENCE();
```

Swap Operations

- Atomic swap operation
  - Replace the contents of memory at A with D
  - Return the original contents of memory at A

```
long ATOMIC_SWAP(long *A, long D);
```

- Atomic compare and swap operation
  - Compare the contents of memory at A to “cmp”
  - If they match, swap “new” into A
  - Always return original contents of memory at A

```
long ATOMIC_CAS(long *A, long new, long cmp);
```

Other Operations

- Population count
  - Counts number of 1s in the word referenced by "ptr"
  - Adds this value to sum and returns the result

```
long POPCNT(long sum, long *ptr);
```

- Priority
  - Computes the 6-bit priority encode on "value"
  - i.e., bit position of highest numbered non-zero bit

```
unsigned long PRIORITY(unsigned long value);
```

Thread Management

- Resize the thread to carry only the live registers
  - Used by compiler or programmer before a possible migration to reduce thread size

```
void RESIZE();
```

- Reschedule the thread to the end of the Run Queue
  - Allow a new thread to be scheduled in the core
  - Can be used to minimize the impact of busy wait

```
void RESCHEDULE();
```

System Query

- Return system time on local node (clock ticks)

```
unsigned long CLOCK();
```

- Return the current thread ID

```
unsigned long THREAD_ID();
```

- Return the current Node ID

```
unsigned long NODE_ID();
```

- Return the number of nodes in the current system

```
unsigned long NUM_NODES();
```

- Return the number of bytes per node

```
unsigned long BYTES_PER_NODE();
```

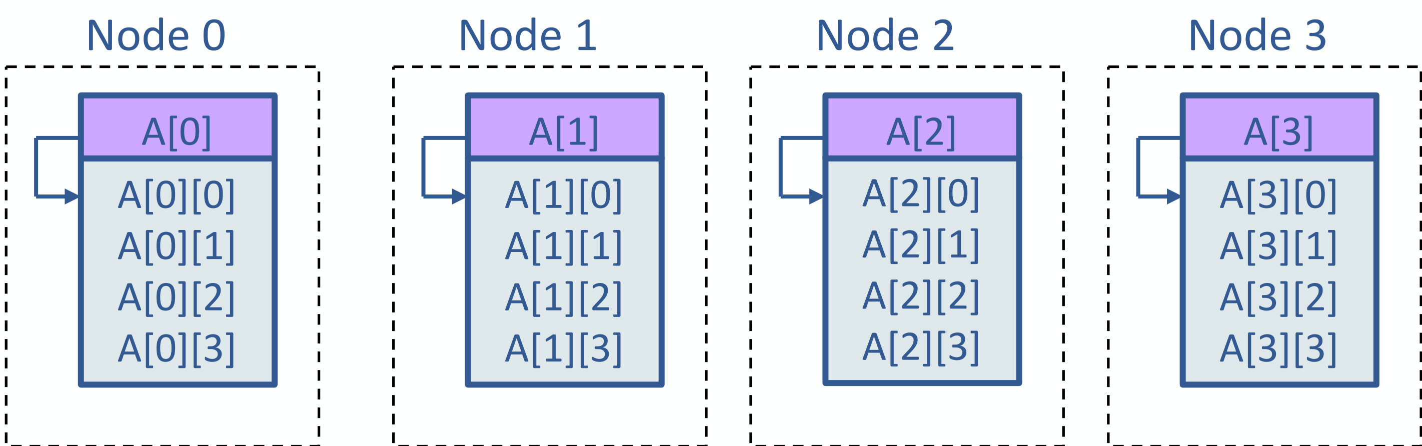
- Return the number of Gossamer cores per node

```
unsigned long GCS_PER_NODE();
```

Example: Distributed Array of Integers

```
#define N 256  // elements per node
long ** A = mw_malloc2d(NUM_NODES(), N * sizeof(long));
for (long j = 0; j < NUM_NODES(); j++)
  for (long i = 0; i < N; i++)
    A[j][i] = i;
```

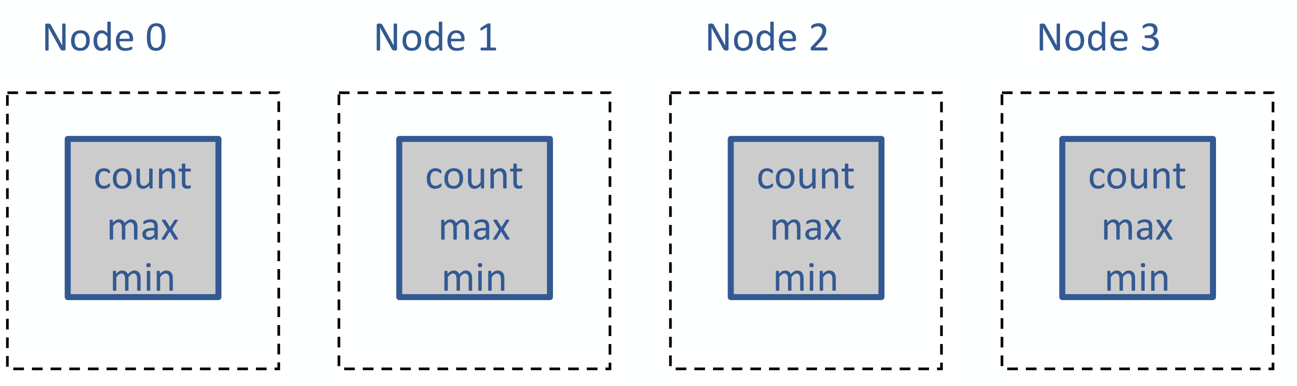
Uses intrinsic for number of nodes in system



Allocate Replicated Data Structure

```
struct stats {
        long count;
        long max;
        long min;
};
replicated struct stats s;
```

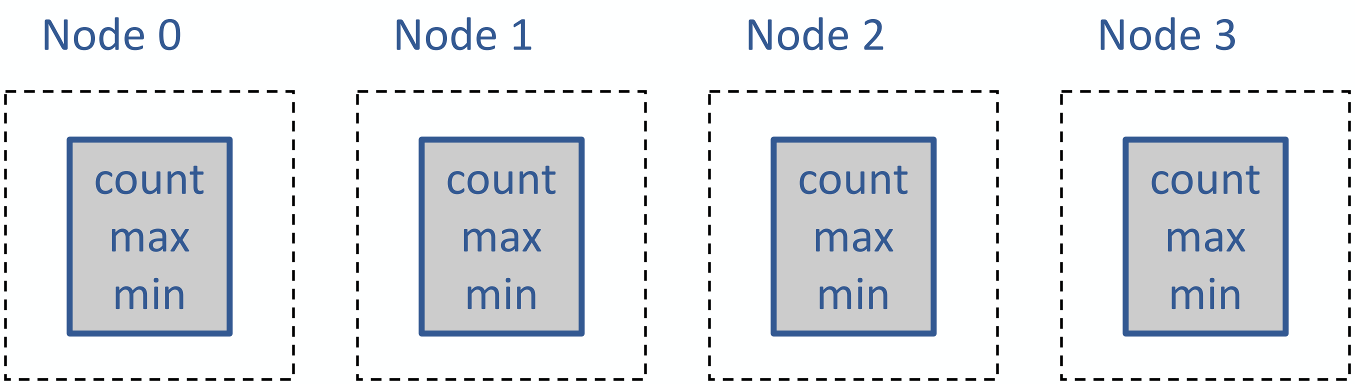
Copy of "stats" structure on each node at the same offset



Initialize Replicated Data Structure

```
for (long i = 0; i < NUM_NODES(); i++) {
  struct stats * si = mw_get_nth(&s, i);
  si->count = 0;
  si->max = 0;
  si->min = LONG_MAX;
}
```

Initialize using mw_get_nth

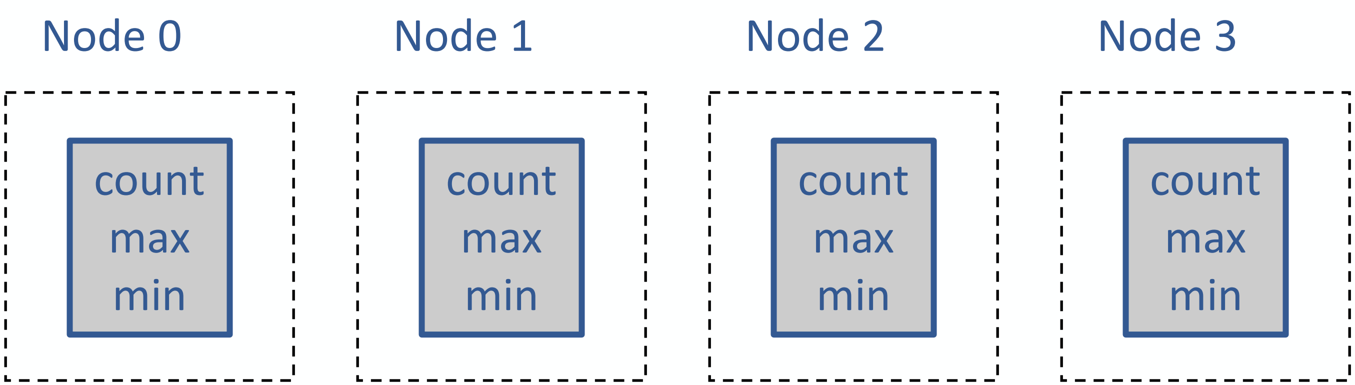


Update Replicated Data Structure

```
cilk_for (long i = 0; i < N; i++) {
  long score = scoreval(A[i]);
  if (score != 0) {
    ATOMIC_ADDM(&(s.count), 1);
    ATOMIC_MAXM(&(s.max), score);
    ATOMIC_MINM(&(s.min), score);
  }
}
```

Loops over array; updates instance on node co-located with thread

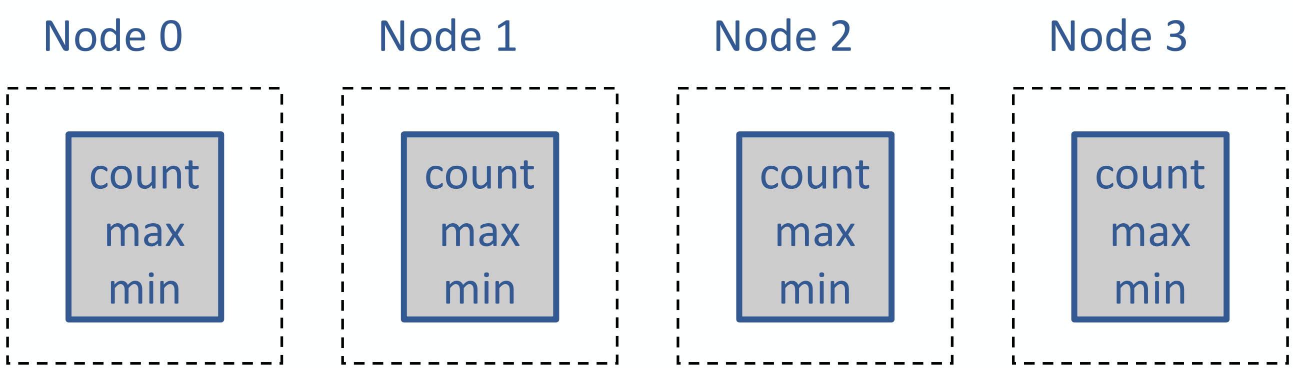
Uses intrinsics for atomic operations on shared data



Reduce Replicated Data Structure

```
unsigned long count = 0;
unsigned long max = 0;
unsigned long min = LONG_MAX;
for (long i = 0; i < NUM_NODES(); i++) {
  struct stats * si = mw_get_nth(&s, i);
  count += si->count;
  if (si->max > max) max = si->max;
  if (si->min < min) min = si->min;
}
```

Reduce over replicated instances on each node



Example: Intrinsics

```
int main(int argc, char **argv)
{
  long **A = distr_array(); // create 2d array
  initialize(); // initialize replicated data
  unsigned long start = CLOCK(); // clock start
  // update replicated data
  for (long I = 0; I < NUM_NODES(); i++) {
    cilk_spawn_at (A[i]) update(A[i]);
  }
  cilk_sync;
  // reduce replicated data
  long count = 0;
  long max = 0;
  long min = LONG_MAX;
  for (long i = 0; i < NUM_NODES(); i++) {
    struct stats *si = mw_get_nth(&s, i);
    count += si->count;
    if (si->max > max) max = si->max;
    if (si->min < min) min = si->min;
  }
  long total = CLOCK() - start; // total cycles
  printf("cycles %lu count %ld max %ld min %ld\n",
         total, count, max, min);
}
```

Use intrinsic to measure clock cycles

Sample Program Execution: intrs.c

```
>>>>>>>> /usr/local/emu/bin/emu-cc  intrs.c -o intrs.mwx
>>>>>>>> /usr/local/emu/bin/emusim.x --total_nodes 4 -- intrs.mwx

        SystemC 2.3.3-Accellera --- Mar 24 2021 16:05:40
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
Start untimed simulation with local date and time= Sun Mar 28 23:33:05 2021

cycles = 4599
End untimed simulation with local date and time= Sun Mar 28 23:33:05 2021

>>>>>>>> more intrs.vsf
Node ID: Outbound Migrations, Threads Created, Threads Died, Spawn Fails
0: 6, 64, 68, 0
1: 3, 64, 63, 0
2: 3, 64, 63, 0
3: 3, 64, 63, 0

NodeID.MspID: num_reads, num_writes, num_rmws
0.0: 411, 594, 184
1.0: 958, 558, 1362
2.0: 402, 600, 146
3.0: 447, 529, 296
```

- Thread creations distributed over nodes
- Migrations only for initial thread spawn
- All accesses are local to replicated variables

Unit Summary: Machine-Specific Coding

- Use of intrinsics for allocating data and distributing data precisely
- Use of intrinsics to measure clock cycles
- Use of atomics to update node-local statistics shared among multiple threads
- Use of intrinsics to perform reductions over replicated variables

<u>Exercises:</u>
Measure other codes using intrinsics
Use atomics to implement more complex structures such as distributed queues
Use remote intrinsics to avoid migrations altogether

Parallel Computation

Local, distributed, and chunked arrays in C/C++ utilities

Parallel Computation with C Utilities

- Functions for local, distributed striped, and distributed chunked
- Programmer supplies function pointer, library spawns threads across system for parallel slides
- Grain size (work per threadlet) can be computed automatically via system settings

Functions for Local Arrays

- Apply function to range in parallel

```
void emu_local_for(…);
```

- Initialize array in parallel

```
void emu_local_for_set_long(…);
```

- Copy array in parallel

```
void emu_local_for_copy_long(…);
```

- Copy a range of bytes

```
void emu_memcpy(…);
```

- Choose grain size based on array size

```
static inline long LOCAL_GRAIN_MIN(…);
```

Local Arrays: example

```
void worker(long begin, long end, va_list args)
{
  long *x = va_arg(args, long*);
  long b = va_arg(args, long);
  for (long i = begin; i < end; ++i) x[i] += b;
}
```

Worker function called on each array slice over iteration space defined by <begin, end>

Spawn parallel threads within a node:

```
long *x = malloc(1024 * sizeof(long));
long begin = 0;
for (long i = 0; i < 1024; i += grain)
{
  long end = MIN(i + grain, 1024);
  cilk_spawn worker(i, end, x, 5);
}
```

Grain size is number of elements to assign to each thread; computed from total size and number of threads

Requires number of threads and grain size to be program variables within application code

Local Arrays: C utilities

```
void worker(long begin, long end, va_list args)
{
  long *x = va_arg(args, long*);
  long b = va_arg(args, long);
  for (long i = begin; i < end; ++i) x[i] += b;
}
```

Same worker function

Spawn parallel threads within a node:

Grain size parameter is minimum number of elements to assign to each thread

```
long *x = malloc(1024 * sizeof(long));
emu_local_for(0, 1024, LOCAL_GRAIN_MIN(1024, 64),
              worker, x, 5);
```

Built-in function computes grain size suitable for processing all items on a single node given a lower bound

Functions for Distributed (striped) Arrays

- Distributed parallel for loop

```
void emu_1d_array_apply(…);
```

- Distributed parallel reduction

```
void emu_1d_array_reduce_sum(…);
```

- Choose grain size based on array size

```
static inline long GLOBAL_GRAIN_MIN(…);
```

Striped Arrays: example

```
void worker(long *array, long begin, long end, va_list args)
{
  long *x = array;
  long b = va_arg(args, long);
  for (long i = begin; i < end; i += NUM_NODES()) x[i] += b;
}
```

Worker function called on each array slice over iteration space defined by <begin, end> with stride NUM_NODES()

Spawn parallel threads within a node:

```
long * x = malloc1dlong(1024 * sizeof(long));
long begin = 0;
for (long j = 0; j < NUM_NODES(); j++) {
  for (long i = 0; i < 1024; i += grain) {
    long end = MIN(i + grain, 1024);
    cilk_spawn worker(&x[j], i, end, 5);
  }
}
```

Grain size is number of elements to assign to each thread; computed from total size and number of threads

Requires number of nodes, number of threads and grain size to be program variables within application code; requires distributed spawn tree to be efficient

Striped Arrays: C utilities

```
void worker(long *array, long begin, long end, va_list args)
{
  long *x = array;
  long b = va_arg(args, long);
  for (long i = begin; i < end; i += NUM_NODES()) x[i] += b;
}
```

Same worker function

Spawn parallel threads within a node:

Grain size parameter is minimum number of elements to assign to each thread

```
long *x = malloc1dlong(1024);
emu_1d_array_apply(x, 1024,
     GLOBAL_GRAIN_MIN(1024, 64), worker, 5);
```

Built-in function computes grain size suitable for processing all items on the entire system; handles distributed spawn inside function hidden from programmer

Functions for Distributed (chunked) Arrays

- Allocate and return replicated pointer

```
emu_chunked_array * emu_chunked_array_replicated_new(…);
```

- Free replicated pointer

```
void emu_chunked_array_replicated_free(…);
```

- Initialize chunked array

```
void emu_chunked_array_replicated_init(…);
```

- Deallocate chunked array

```
void emu_chunked_array_replicated_deinit(…);
```

- Pointer to i-th element

```
static inline void * emu_chunked_array_index(…);
```

- Number of elements

```
long emu_chunked_array_size(…);
```

Functions for Distributed (chunked) Arrays (cont.)

- Distributed parallel for loop

```
void emu_chunked_array_apply(…);
```

- Initialize array in parallel

```
void emu_chunked_array_set_long(…);
```

- Distributed parallel reduction

```
long emu_chunked_array_reduce_sum(…);
```

- Scatter from local array to distributed array

```
void emu_chunked_array_from_local(…);
```

- Gather from distributed array to local

```
void emu_chunked_array_to_local(…);
```

Chunked Arrays: example

```
void worker(long *x, long begin, long end, va_list args)
{
  long b = va_arg(args, long);
  for (long i = begin; i < end - begin; ++i) x[i] += b;
}
```

Worker function called on each array slice of chunked array over iteration space defined by <begin, end>

Spawn parallel threads within a node:

```
long **x = mw_malloc2d(1024, sizeof(long));
long begin = 0;
for (long j = 0; j < NUM_NODES(); j++) {
  for (long i = 0; i < 1024; i += grain) {
    long end = MIN(i + grain, 1024);
    cilk_spawn worker(&x[j][0], i, end, 5);
  }
}
```

Grain size is number of elements to assign to each thread; computed from total size and number of threads

Requires number of nodes, number of threads and grain size to be program variables within application code; requires replicated variables and distributed spawn to be efficient

Chunked Arrays: C utilities

```
void worker(emu_chunked_array *array, long begin, long end, va_list args)
{
  long *x = emu_chunked_array_index(array, begin);
  long b = va_arg(args, long);
  for (long i = begin; i < end - begin; ++i) x[i] += b;
}
```

Worker function uses chunked array

Spawn parallel threads within a node:

Grain size parameter is minimum number of elements to assign to each thread

```
emu_chunked_array *x =
  emu_chunked_array_replicated_new(1024, sizeof(long));
emu_chunked_array_apply(x, GLOBAL_GRAIN_MIN(1024, 64),
  worker, 5);
```

Built-in function computes grain size suitable for processing all items on the entire system; hides distributed spawn from programmer

Unit Summary: Parallel Computation

- C Utilities for parallel programming
- Local, distributed, and chunked arrays

<u>Exercises:</u>
Re-implement saxpy using distributed, chunked arrays