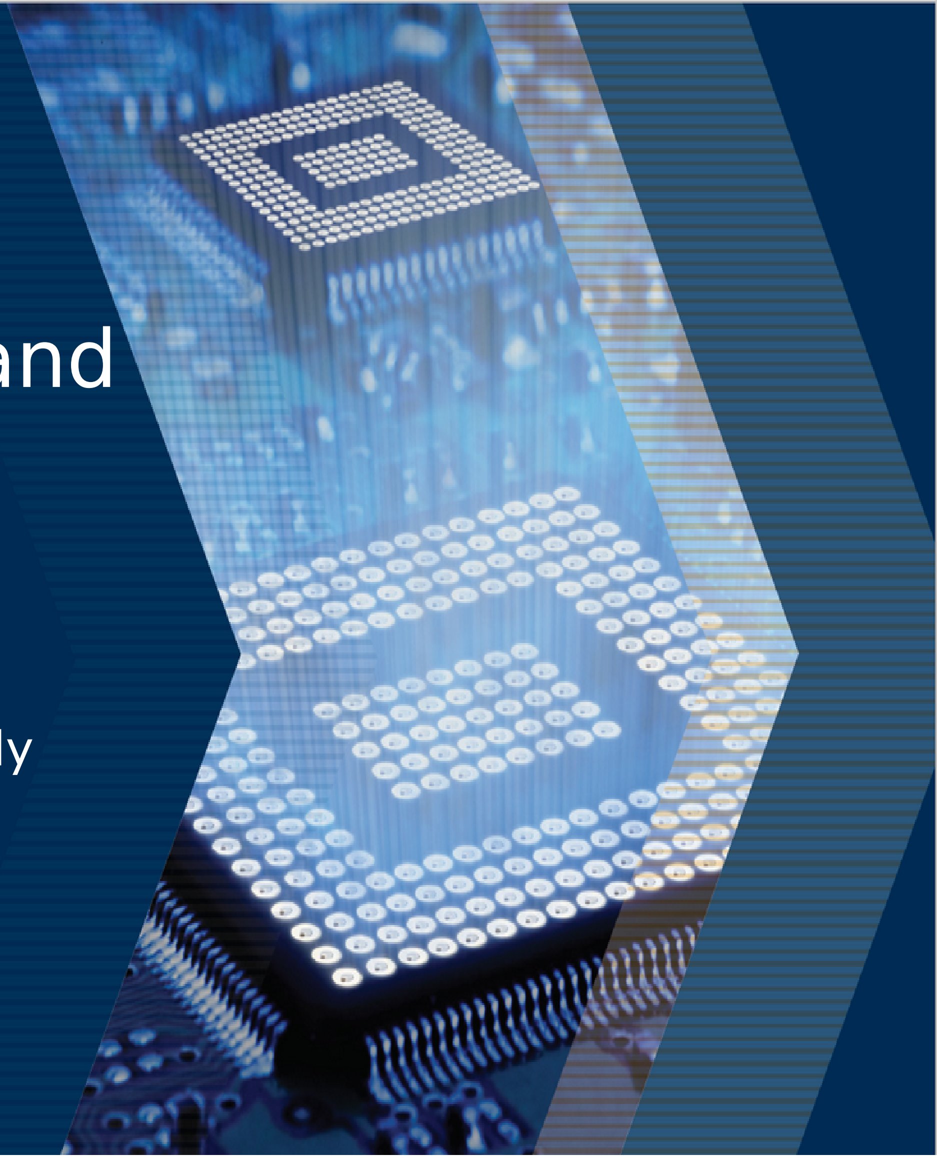
EMU Chick Overview and Hands-On

31 July 2019

Hosted by Will Powell and Jason Riedy

Emu Technology (c) material will be denoted by the use of the Emu logo



CREATING THE NEXT MOORE'S LAW

Georgia Tech Center for Research into Novel Computing Hierarchies

For more information, visit our website at **www.crnch.gatech.edu** or send email to **crnch@gatech.edu**

Outline

- Introduction to the Emu Chick **(1:30-2:00 PM)**
  - Example 1: Hello World
    - Memory replication
    - Basic thread Spawning Strategies
- Programming for the Chick **(2:00-4:30 PM)**
  - Example 2: STREAM
    - Spawn strategies
    - Granularity
    - Locality awareness
  - Example 3: Sparse Matrix Vector Multiply (SpMV)
    - COO for fully interleaved operation
    - CSR decomposition for the Chick

Emu Chick Overview

Emu Innovation Overview

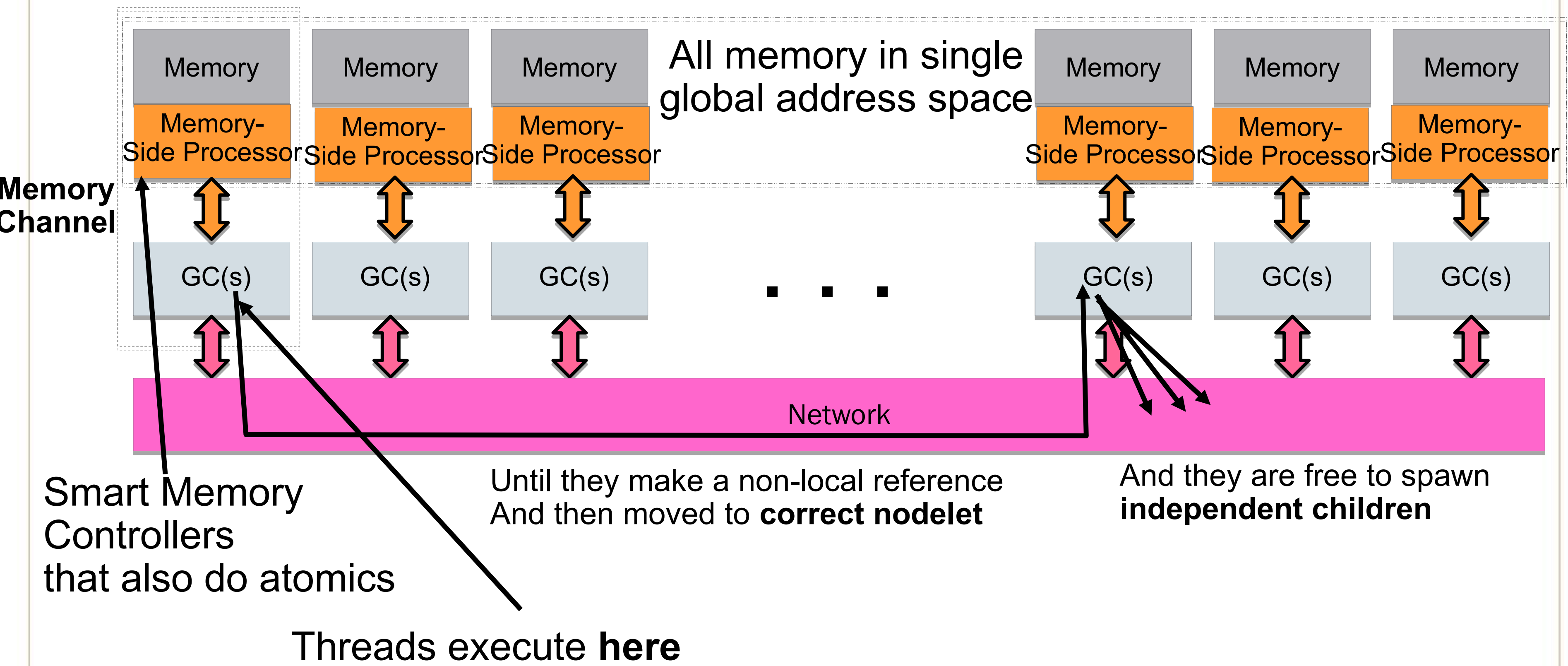
Designed from the ground up to deal with applications that exhibit little locality

- Massive Shared Memory for in-Memory Computing
  - No I/O bottlenecks
- **EMU** moves ("***Migrates***") the program context to the locale of the data accessed
  - Lower energy – less data moved shorter distances
- Finely Grained Parallelism
  - Reduces concurrency limits
- Compute, memory size, memory bandwidth and software scale simultaneously

EmuTechnology

Emu Architecture Functional Diagram

Nodelet: New unit of parallelism



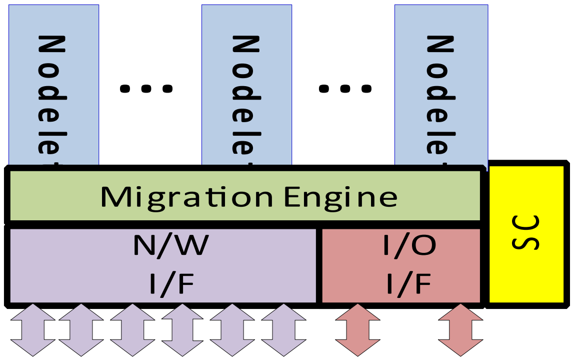
EmuTechnology

Node Architecture

- 8 Nodelets
- Migration Engine
- 6 RapidIO 2.3 4-lane network ports
- Stationary Cores (SCs)

  DualCore 64-bit Power E5500
  - 2GB DRAM
  - 1 TB SSD
  - PCIe Gen 3
  - Runs Linux

Stationary Core Runs OS, Launches Jobs



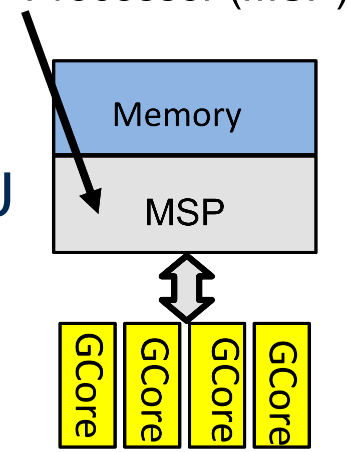
(b) Node

Migrating Threads are major traffic on Network

Nodelet Architecture

- 8 GB DDR4 Narrow Channel Memory
  - Supports 64-bit accesses
- Memory-side Processor (MSP)
  - Handles atomics and remote writes at the memory
- Gossamer Cores (GCs) each with FMA FPU
- Nodelet Queue Manager
  - Run Queue
    - Incoming threads from migrations, spawns, or SC
    - Loaded into vacant execution slots by hardware
  - Migration Queue
    - Threads that need to migrate to non-local data
  - Service Queue
    - Threads that need system services from the SC

Atomics run in Memory-Side Processor (MSP)



(a) Nodelet

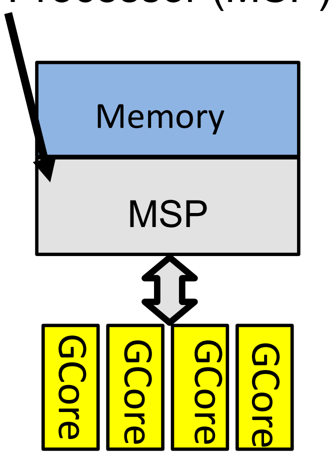
Multi-Threaded Cores

Hardware Thread Management

- Thread scheduling in GCs automatically performed by hardware
- SPAWN instruction
  - Creates new thread and places it in Run Queue
- RELEASE instruction
  - Places thread in Service Queue for processing by SC
- Non-local memory reference causes a migration
  - Thread context packaged by hardware and placed in Migration Queue
  - Migration Engine sends packet to new location and places in Run Queue

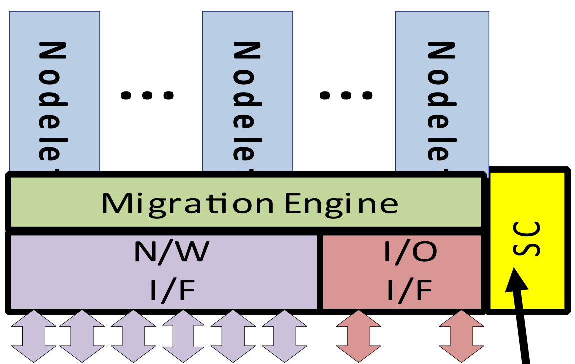
Emu System Hierarchy

Atomics run
in Memory-Side
Processor (MSP)



(a) Nodelet

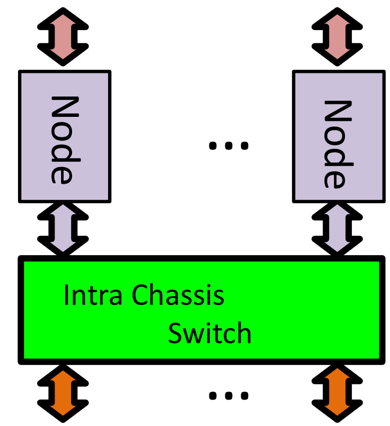
Multi-Threaded Cores



(b) Node

Migrating Threads are major traffic on Network

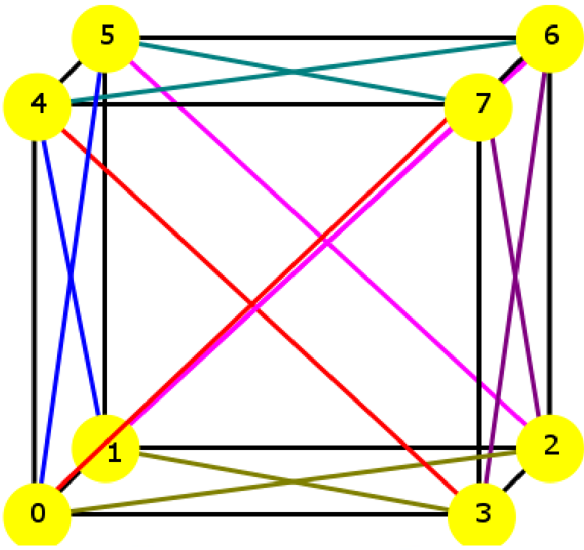
Stationary Core Runs OS, Launches Jobs



Up Links to
(c) Inter-Chassis
Switch

Emu Chick Topology

- System consists of 8 nodes connected in a cube via RapidIO links
  - Each node connects to 6 other nodes
  - Cube edges and face diagonals are connected, but not interior diagonals
- All routes are 2 hops or l
  - 3D diagonals route throug intermediate node
  - All others are 1 hop

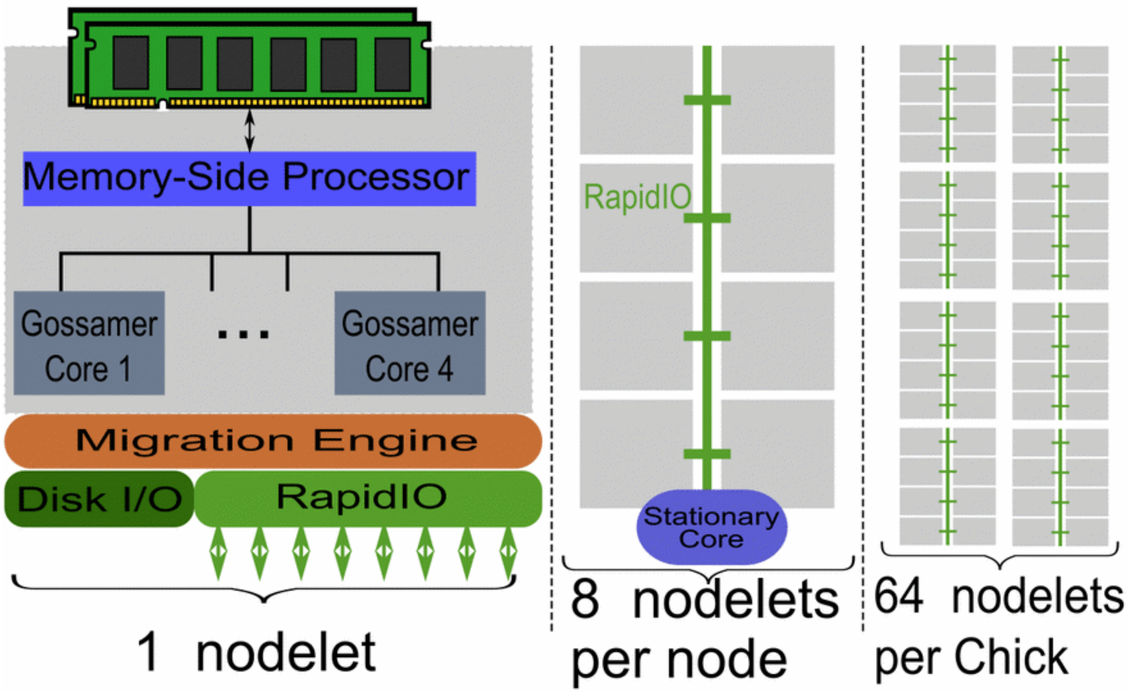


EmuTechnology

The Current Chick Hardware

- 8 nodes (64 nodelets)
- 512 GB Shared Memory
- 8 TB SSD
- 8192 Concurrent Threads
- Copy room environment
- Shipping now

Emu Chick Architecture



Emu's Migratory Thread Model

Massive, fine-grained multithreading where computation migrates to the data so that accesses are always local

Key Issues:

- Thread control: spawning and synchronization
- Data distribution and affinity of execution
  - Load balance
  - Hotspots
  - Migration patterns

Fine-grained Memory Accesses

- Narrow-channel DRAM (NCDRAM)
  - 8-bit bus allows access at 8-byte granularity without waste
  - Many narrow channels instead of few wide channels
- Remote Writes
  - Write to remote nodelet without migrating
  - Proceed directly to the memory front-end, bypassing the GC
- Remote Atomics
  - Performed in Memory Front-End (MFE), near memory

Emu Programming – Key Features

- **Cilk:** Extensions to C to support thread management
  - cilk_spawn
  - cilk_sync
  - cilk_for
- **Emu Cilk:** Extensions to Cilk to support migrating threads
  - cilk_migrate_hint
  - cilk_spawn_at

Emu Programming – Key Features

- **Memory allocation library:** Specialized malloc/free for data distributed across nodelets
- **Intrinsics:** Allow access to architecture specific operations such as atomic updates

Emu Cilk

Emu hardware dynamically creates and schedules threads

- Normally requires no software intervention
- When a thread completes, it returns values to its parent and dies
- When a thread blocks, it may voluntarily place itself at the back of the run queue (instead of “busy waiting”)
- Number of threads limited only by available memory
- Extremely lightweight – Cilk threads can be very small and still be efficient

Cilk Functions

```
long f = cilk_spawn fib(a, b);
```

- Specifies function may run in parallel with caller
  - Child thread spawned to execute function and parent continues in parallel w/child
  - Otherwise parent executes a standard function call
- Spawn location determines location of
  - Synchronization structure
  - Stack frame (if needed)
- Spawn destination
  - Special functions denote spawn location
  - If no direction is given, then spawn is local

Cilk Functions

`cilk_sync;`

- Current function cannot continue past the `cilk_sync` until all children have completed
- Last thread to reach the `cilk_sync` continues execution – **no waiting**
- Implicit sync at termination of a function

Cilk Functions

```
#pragma cilk grainsize = 4
cilk_for(long i=0; i<SIZE; i++)
  {…}
```

- Divides loop among parallel threads, each containing one or more contiguous loop iterations
- Max number of iterations in each chunk is *grainsize*
- Best for situations where
  - Threads are spawned locally
  - Work per element is fairly uniform

Emu CilkPlus Functions

```
cilk_migrate_hint(p);
```

- Specifies nodelet for next cilk_spawn operation
  - Argument p is a pointer into destination nodelet's memory

```
cilk_spawn_at(p) fib(a,b);
```

- Combines cilk_migrate_hint and cilk_spawn into a macro for single-command spawn
  - Implemented as C macro; may require braces for correct operation

Cilk Fibonacci Example

```
#include "memweb.h"
#include <cilk/cilk.h>
#define N 10
long fib(long n) {
  if (n < 2)
    return n;
  long a = cilk_spawn fib(n-1);
  long b = cilk_spawn fib(n-2);
  cilk_sync;
  return a + b;
}
int main() {
  long result = fib(N);
  printf("fib(%d) = %ld\n", N, result);
}
```

Spawn a thread for each of the `fib()` calls

Wait for threads to complete to ensure `a` and `b` are valid

HELLO WORLD AND HANDS-ON

Ex 1: Emu Hello World

This example demonstrates the following:

- Memory replication
- Basic thread Spawning Strategies

Chick Hands-On Information

- System: **notebook.crnch.gatech.edu**
- Username: pearc$(seq 0 49)
- Login: https://notebook.crnch.gatech.edu

  `Jupyterhub → ephemeral JupyterLab`
- Emu directory: /tools/emu
- Already in $PATH:
  - Compiler: emu-cc [-h]
  - Simulator: emusim.x [-h]

Emusim.x can be used to explore differences in..

- Execution Time
- Migrations
- Memory Map
- Remotes Map
- Run Queue
- Migration Queue
- Remote Queue

Configuration and Summary Statistics

- Generated automatically in `<program>.cdc`
- Check number of threads spawned, their distribution, and number of migrations
- View memory map and remotes map to identify hotspots, poor distribution, and migration patterns

EmuTechnology

Examine Simulation Output

- `hello_world.cdc`
  - Execution Time
  - Total Threads
  - Maximum Concurrent Threads
  - Memory Map
  - Remotes Map

Verbose Simulation Statistics

- Generated automatically in `<program>.vsf`
- Identify hotspots/bottlenecks by examining max queue depths
  - Run Queue
  - Migration Queue
  - Remote Queue
- Identify utilization at nodelets
  - IPC (Instructions per cycle: Max 4.0 for 4 cores)
  - Memory Bandwidth (Max 1.0)
  - System IC Bandwidth (Max 1.0)

Examine Simulation Output

- `hello-world.vsf`
  - Much of the same information presented with more detail and different organization
  - Provides queue depths for run queue, migration queue, and remote queue
- Run visualization tool

```
emuvistool hello-world.tqd
```

Ex 1: Exploring Simulation Output

EMU MEMORY ALLOCATION

Memory Allocation

- Replicated, Stack, and Heap sections on each nodelet
- Replicated – global replicated data
- Stack – local memory allocation
  - Thread frames
  - `malloc()/free()`
  - `new()/delete()`
- Heap – distributed memory allocation
  - Specialized `mw_*malloc*()` functions

EmuTechnology

Global Replicated Data Structures

```
replicated long c = 3927883;
```

- Instructs compiler to place an instance on each nodelet
- Uses a “View 0” address that always gives local instance
- Must be a **global** variable
- Example Uses:
  - Constants
    - Copy on each nodelet
    - All initialized to the **same** unchanging value
    - EX: PI, pointer to shared data structure
  - Local data
    - Copy on each nodelet
    - May have **differen**t values
    - Use only when it does **not** matter which instance you access!
    - EX: random number table, pointer to local work queue

Dynamic Replicated Pointers

```
long * mw_mallocrepl(size_t blocksize)
```

- Allocates a block on each nodelet, returns replicated pointer
- Similar to using the replicated keyword
- Used when the size of the data structure is not known at compile time

EmuTechnology

Replicated Data Structures

- Replicating key shared data structures can improve performance
  - Pointers to shared distributed data e.g. array
  - Copy at each nodelet avoids migrations to get address
  - Compiler generates the address rather than having to pass the address to each function call and carry it during migrations
  - Can reduce spills at function calls

Initializing Replicated Data Structures

```
void mw_replicated_init(long *repl_addr, long value)
```

– Initializes each instance of replicated data structure to value

```
void mw_replicated_init_multiple (long *repl_addr,
         long (*init_func)(long) )
```

– Initializes each instance of replicated data structure using the **result** of the user-defined function `init_func(n)` where `n` is the nodelet number

```
void mw_replicated_init_generic(long *repl_addr,
            void (*init_func)(void *, long) )
```

– Initializes each instance of replicated data structure using the user-defined function `init_func(&obj, n)`, where obj is the address of the ted data structure and n is the nodelet

Accessing Replicated Data Structures

```
void * mw_get_localto(void *r_ptr, void *dest_ptr)
```

– Returns a pointer to the instance of a replicated data structure co-located with the destination pointer

```
void * mw_get_nth(void *r_ptr, unsigned n)
```

– Returns a pointer to the nth instance of a replicated data structure

Local Memory Allocation

- Allocate from the stack on the current nodelet using conventional C/C++ functions
  - malloc and free
  - new and delete

Distributed Memory Allocation

```
void * mw_localmalloc(size_t eltsize, void *ptr)
```

- Block of memory located in same locale as another data structure

```
void * mw_malloc1dlong(unsigned numelements)
```

- Array of longs striped across nodelets round robin

```
void ** mw_malloc2d(unsigned nelements,
                    size_t eltsize)
```

- Array of pointers striped across nodelets round robin
- Each points to a block of memory in the same locale

Distributed Free

```
void mw_free(void *allocedpointer)
```

- Free data allocated by `mw_malloc2d`

```
void mw_localfree(void *allocedpointer)
```

- Free data allocated by `mw_localmalloc`

EmuTechnology

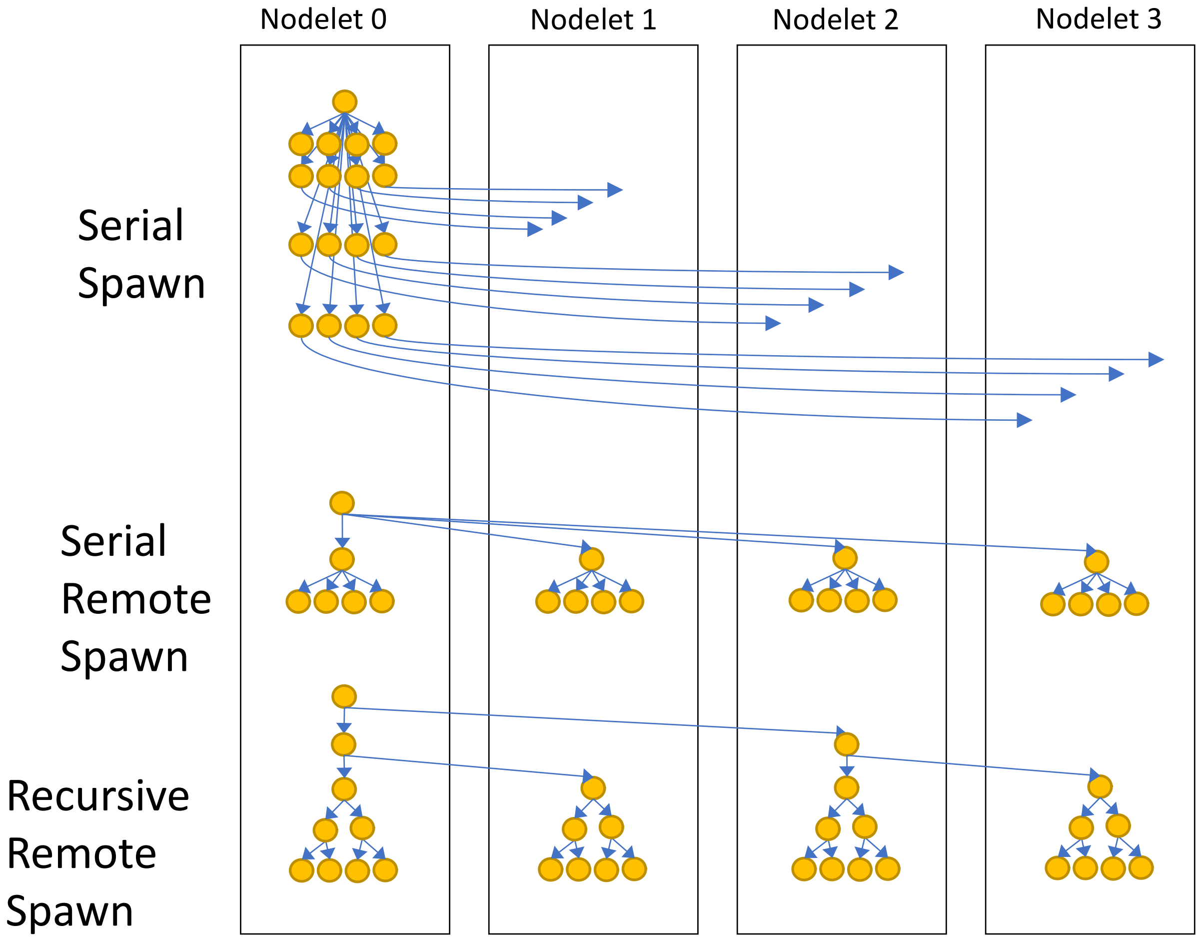
Georgia Tech Center for Research into Novel Computing Hierarchies

Accessing Distributed Data

```
long * mw_arrayindex(void *array2d,
unsigned long i, unsigned long numblocks,
size_t blocksize)
```

- Inputs:
  - Array allocated with mw_malloc2d
  - Index for first dimension
  - Number of blocks used in malloc2d
  - Blocksize used in malloc2d
- Returns address of array2d[i][0]

Added Spawn Strategies



STREAM Implementation

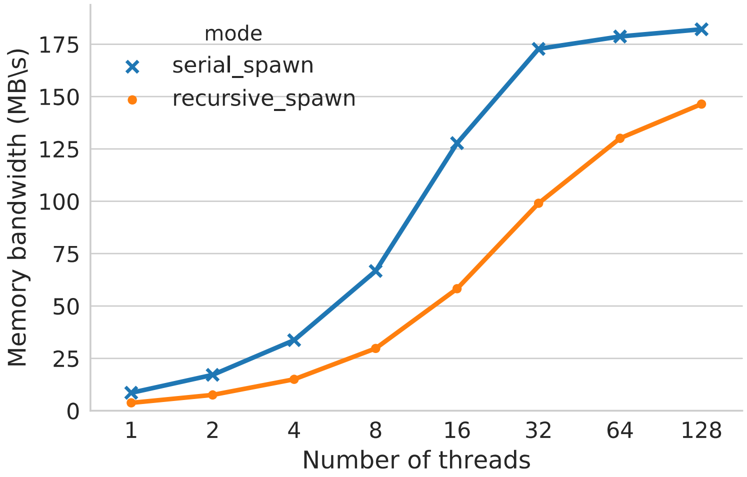
```
// Serial
for (long i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
}

// Parallel
cilk_for (long i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
}
```

Ex: 2 Emu STREAM

This example demonstrates the following:

- More complex spawning strategies
- Grain size settings
- Locality awareness (i.e., limiting some migrations)



HARDWARE EXECUTION OVERVIEW

Emu Chick Configuration

- Single-node Execution
  - Program runs on a single node
  - Can access all 8 nodelets but no other nodes
  - Users can work independently on different nodes
- Multi-node Execution
  - Program runs on full system
  - Can access all 8 nodes (64 nodelets)
  - Single user

Emu Chick Hardware Execution

- Compile programs on notebook.crnch.gatech.edu then `scp` to Chick node
- Single-node Execution
  - Launched on node using `emu_handler_and_loader`
- Multi-node Execution
  - Launched on node 0 using `emu_multinode_exec`

Program Execution Utilities

- **Load** program and data to all nodelets
- **Launch** initial thread into the system
- **Monitor** the system exception queue and handle system services until a thread quits or an exception occurs
- **Terminate** by issuing a checkpoint to clear the system and dump any remaining threads
- **Print** information to log files for each thread that quits, exits, generates an exception, or is checkpointed
- **Return** the program’s return value

System Services

- Thread suspends itself and writes thread state registers (TSR) to the system exception queue (SEQ)
- Handler polls system SEQ for threads that need services
- Handler reads the thread from the SEQ and performs the requested service
- Handler then relaunches the thread to nodelet 0

Running Code on the Chick

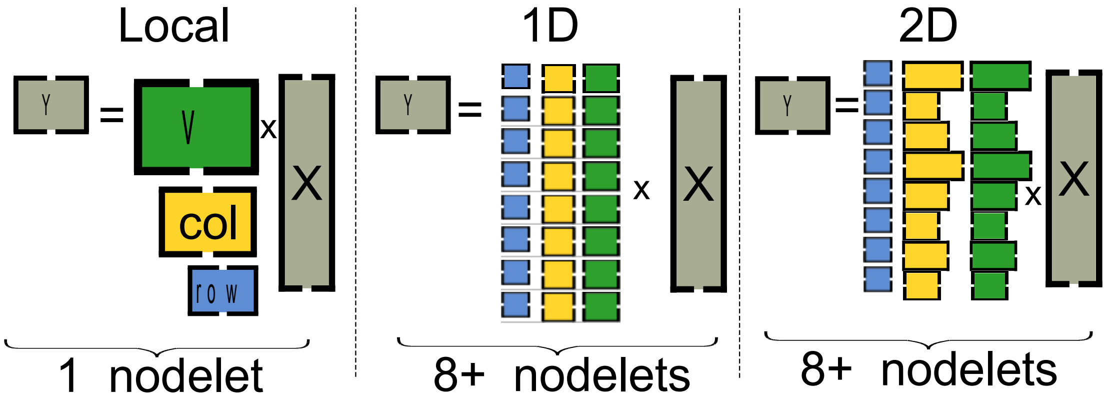
- System: **karrawingi-login.crnch**
  - SSH keys should be set up for you to use
  - ~/.ssh/config includes proxy setup
- Username: pearc$(seq 0 49)
  - *ssh karrawingi-login*
- Emu Chick nodes
  - *ssh n0 – n7*

Ex 3: Emu SpMV

This example demonstrates the following:

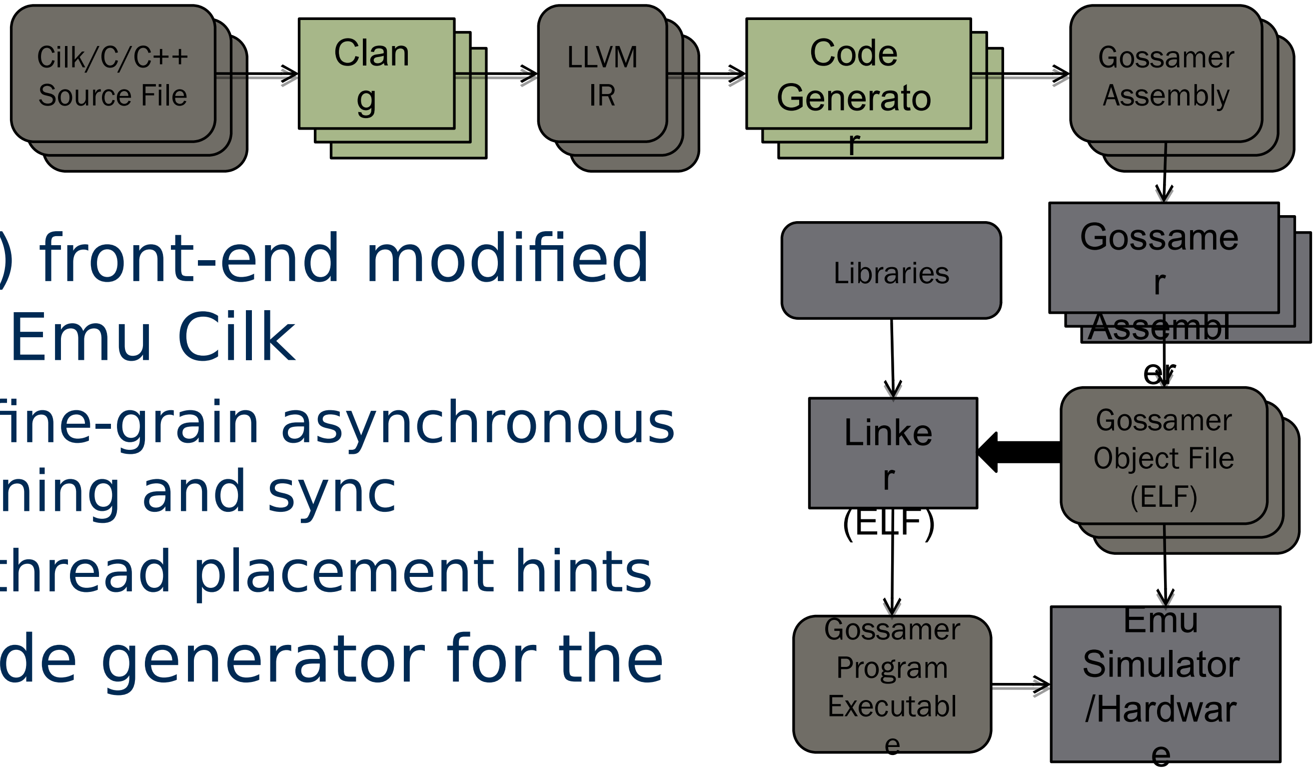
– COO vs CSR layouts

– More advanced data distributions



EMU TOOLCHAINS AND DEVELOPMENT ENVIRONMENT

Emu Cilk Toolchain



- Cilk (clang) front-end modified to support Emu Cilk
  - Supports fine-grain asynchronous task spawning and sync
  - Supports thread placement hints
- Custom code generator for the Emu GCs
- Custom calling convention and run-time support
- Custom assembler and linker

Support for C, C++, and CilkPlus provides **familiar development environment**

CilkPlus Beta Release

- Latest Clang front-end to support
  - C/C++ 2011
  - CilkPlus
- Key CilkPlus features
  - **Reducers:** list, min/max, addition, bitwise AND/OR/XOR, multiplication, ostream, string, vector
  - **Pedigrees:** unique naming convention for threads
- No support for
  - CilkPlus vector operations
- Currently delivered in parallel with standard release

Emu C Utilities

- Set of common patterns for thread-parallel code implemented efficiently as library calls
- Working with local arrays
  - Alternative to cilk_for, no compiler support
- Working with distributed striped arrays
  - 2-level spawn tree, split array for worker functions
- Working with distributed chunked arrays
  - Calculates indices, applies functions to blocked arrays
- Timing hooks
  - Timer subsystem for performance analysis

User Libraries

- GNU Multiple Precision Arithmetic (GMP) Library
  - Library for arbitrary precision arithmetic
  - Currently support integer GMP for Emu
  - Included in current release
- Under development
  - GraphBLAS (UMBC / SEI)
  - OpenMP (Stony Brook University)
- Other efforts
  - STINGER Graph Library (Georgia Tech)
  - Kokkos C++ Ecosystem (Georgia Tech / Sandia)

What have we not covered today?

- Atomics and intrinsics
- Multi-node execution
- Thread management

Backup

Intrinsics

- Set of compiler recognized functions to access architecture specific operations
  - Atomic Arithmetic Operations
  - Remote Arithmetic Operations
  - Other Architecture Specific Operations
  - Thread Management Functions
  - System Queries

Atomic Arithmetic Operations

- Atomic arithmetic operations at the memory

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

- Four distinct “flavors” of each atomic that specify the value returned and resulting mem

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

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

- 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

WRD – Remote Write

- Automatically generated by the compiler in place of the store instruction
- Writes the value to remote memory and sends an acknowledgement (ACK)
- If the access is local, treated as a store

FENCE on Remote Update

```
void FENCE();
```

- 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.

Swap Operations

```
long ATOMIC_SWAP(volatile void *A, long D)
```

- Replace the contents of memory at `A` with `D`
- Return the original contents of memory at `A`

```
long ATOMIC_CAS(volatile void *A,
  long newVal, long cmpVal)
```

- Compare the contents of memory at `A` to `cmpVal`.
- If they match, swap `newVal` into `A`
- Always return original contents of memory at `A`

Other Operations

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

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

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

- Computes the 6 bit priority encode on `value`
- i.e. bit position of highest numbered non-

Thread Management

`void MIGRATE(volatile void *A)`

- Force a migration to an address

`void RESIZE()`

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

`void RESCHEDULE()`

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

Thread Management (cont)

- void ENABLE_ACKS() – require acks on remotes
- void DISABLE_ACKS() – turn off acks
- void DISABLE_MIGRATIONS() – cause exception on migration (for debugging)
- void ENABLE_MIGRATIONS() – resume migrations without exceptions
- void DISABLE_INTERRUPTS() – prevent thread from timeslice interupt
- void ENABLE_INTERRUPTS() – allow timeslice interrupts from thread
- void ENTER_CRITICAL_SECTION() – combine DISABLE_MIGRATIONS and DISABLE_INTERRUPTS
- void EXIT_CRITICAL_SECTION() – combine ENABLE_MIGRATIONS and ENABLE_INTERRUPTS

System Query

- CLOCK() - Returns system time on local nodelet (clock ticks)
- THREAD_ID() – Returns the current thread ID
- NODE_ID() – Returns the current Nodelet ID
- NODELETS() - Returns the number of nodelets in the current system
- BYTES_PER_NODELET() – Returns the number of bytes per nodelet
- MAXDEPTH() – maximum spawn depth*

*planned, not yet implemented

Initializing Replicated Data Structures

```
void mw_replicated_init(long *repl_addr, long value)
```

– Initializes each instance of replicated data structure to the same value

```
replicated long N;
int main()
{   …
    long n_elements = compute_n_elements();
    mw_replicated_init(&N, n_elements);
    …
}
```

Initializing Replicated Data Structures

```
void mw_replicated_init_multiple
      (long *repl_addr, long (*init_func)(long))
```

– Initializes each instance of the replicated data structure using the **result** of the user-defined function `init_func(n)` where `n` is the nodelet number

```
replicated long B;
long init_func(long nid) {
   return nid * 5;
}
int main()
{  …
   mw_replicated_init_multiple(&B, init_func);
   …
}
```

Initializing Replicated Data Structures

```
void mw_replicated_init_generic(long *repl_addr,
            void (*init_func)(void *, long) )
```

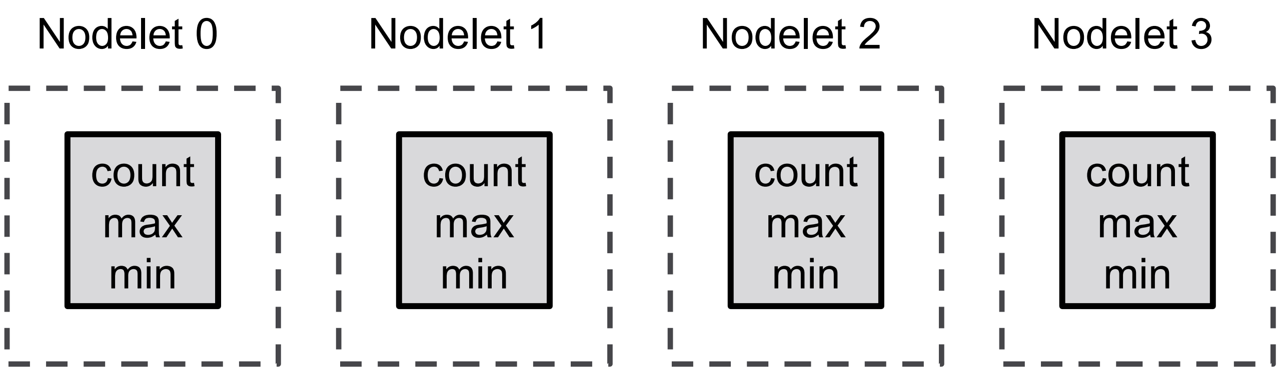
– Initializes each instance of replicated data structure using the user-defined function `init_func(&obj, n)` where `obj` is the address of the replicated data structure and `n` is the nodelet number

```
replicated struct info { long x; long y; } info;
void init_info(void *obj; long nid) {
    struct info *i = (struct info*) obj;
    i->x = node;
    i->y = 5*node + 4;
}
int main()
{   …
    mw_replicated_init_multiple(&info, init_info);
    …
}
```

Allocate Replicated Data Structure

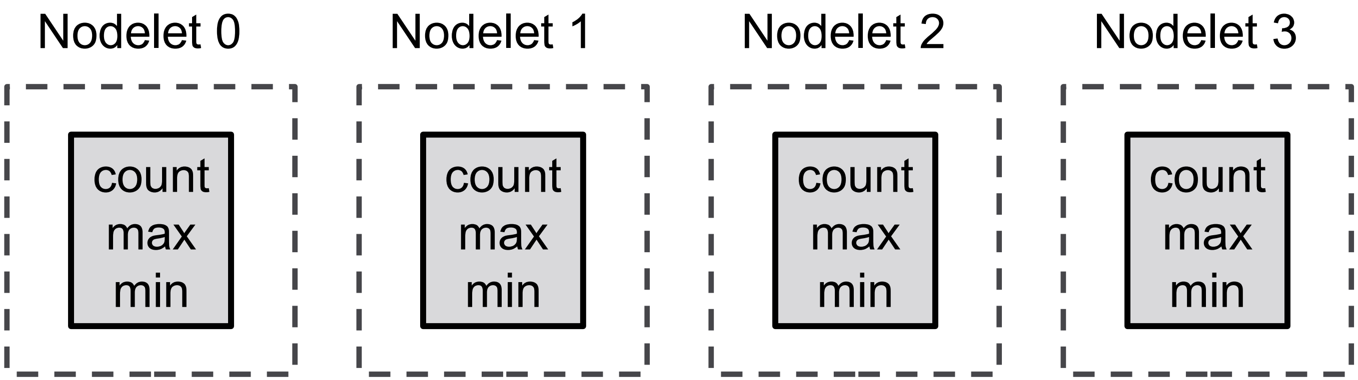
Copy of stats on each nodelet at the same offset

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



Initialize Replicated Data Structure

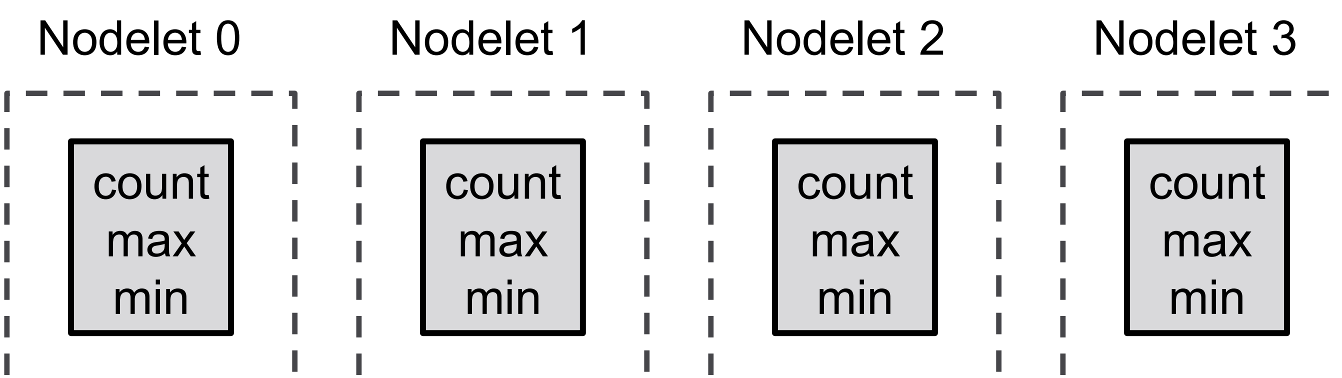
```
// Initialize using mw_get_nth
for (long i=0; i<NODELETS(); i++) {
    struct stats * si = mw_get_nth(&s, i);
    si->count = 0;
    si->max = 0;
    si->min = LONG_MAX;
}
```



Update Replicated Data Structure

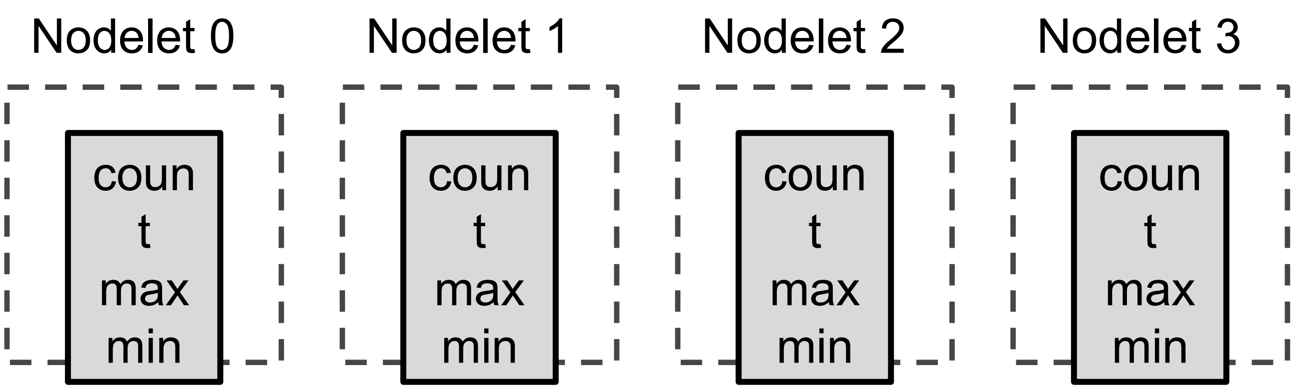
```
// Update stats
cilk_for (long i=0; i<N; i++) {
    unsigned long score = score(A[i]);
    if (score != 0) {
        ATOMIC_ADDM(&(s.count), 1);
        ATOMIC_MAXM(&(s.max), score);
        ATOMIC_MINM(&(s.min), score);
    }
    }
```

Remember to use atomics for shared data



Reduce Replicated Data Structure

```c
// Reduce using mw_get_nth
unsigned long count = 0;
unsigned long max = 0;
unsigned long min = LONG_MAX;
for (long i=0; i<N; 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;
}
```

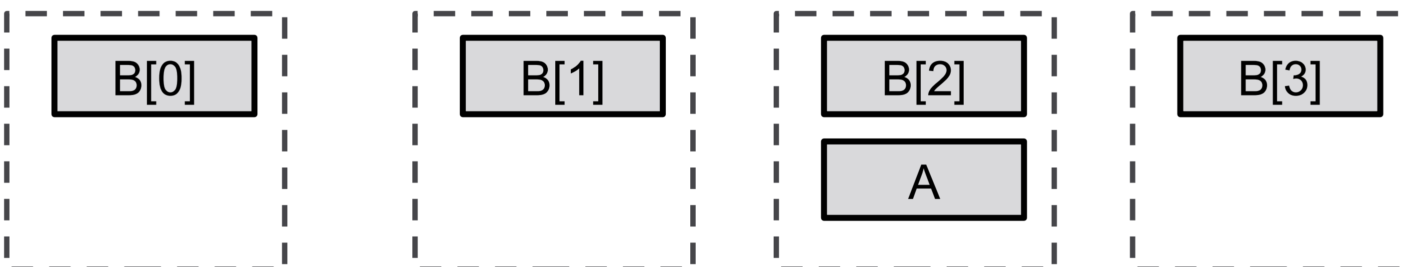


Allocate Co-located Data

```
void * mw_localmalloc(size_t eltsize, void *ptr)
```

- Block of memory of size `eltsize` co-located with address `ptr`
- Accessed using traditional C semantics
- Leaves you at the new location
- Example:

```
long * A = (long *)
        mw_localmalloc(sizeof(long), &B[2]);
```



Allocate 1D Array of Longs

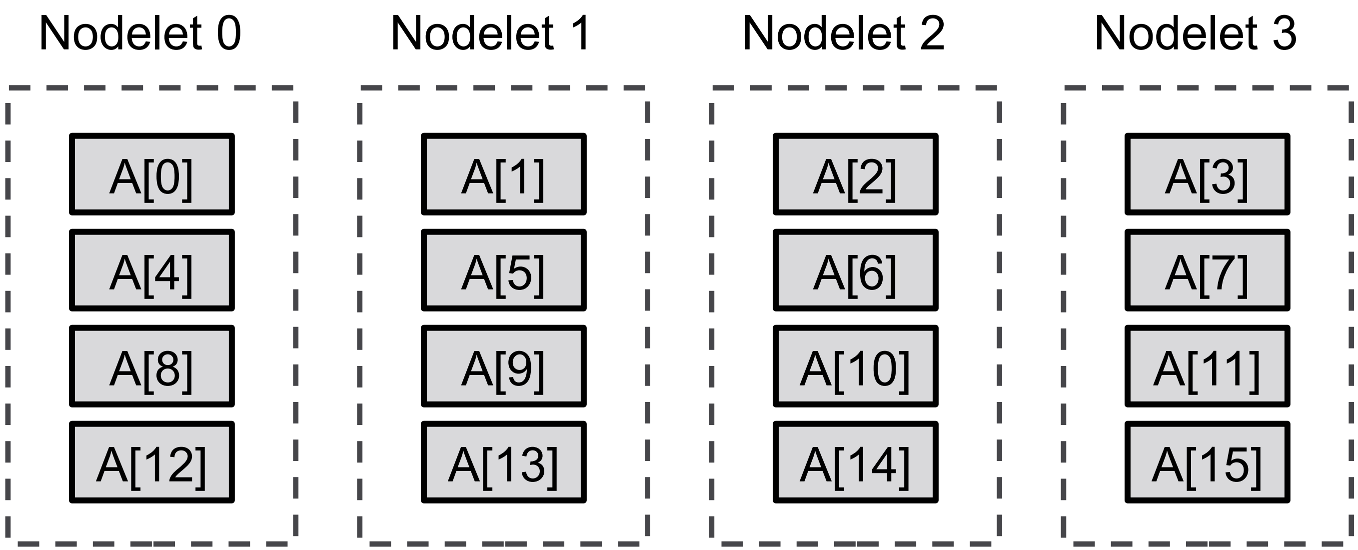
```
long * mw_malloc1dlong(unsigned nelements)
```

- Array of `nelements` longs striped across nodelets round robin
- Accessed using 1D array notation (e.g. A[i])
- ONLY works for 64-bit types (e.g. long)

Example: 1D Array

Array of N elements, striped across nodelets round-robin

```
#define N 16
long * A = (long *) mw_malloc1dlong(N);
for (long i=0; i<N; i++)
    A[i] = i;
```



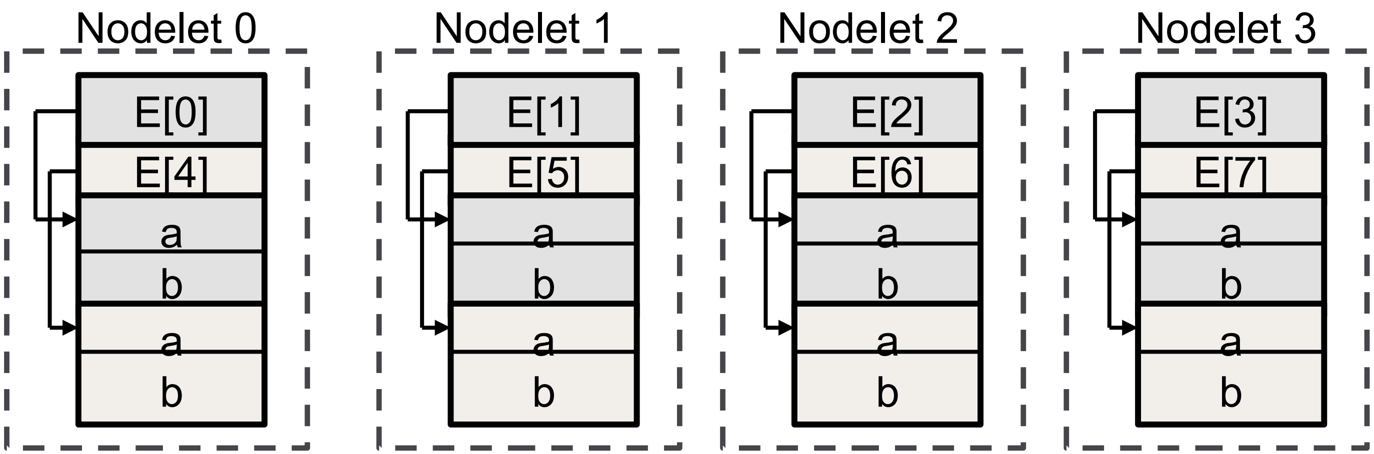
Allocate 2D Array of Elements

```
void ** mw_malloc2d(unsigned nelements,
                    size_t eltsize)
```

- Array of `nelements` pointers striped across nodelets round robin
- Each points to co-located memory block of size `eltsize`
- May be an array of pointers to any type

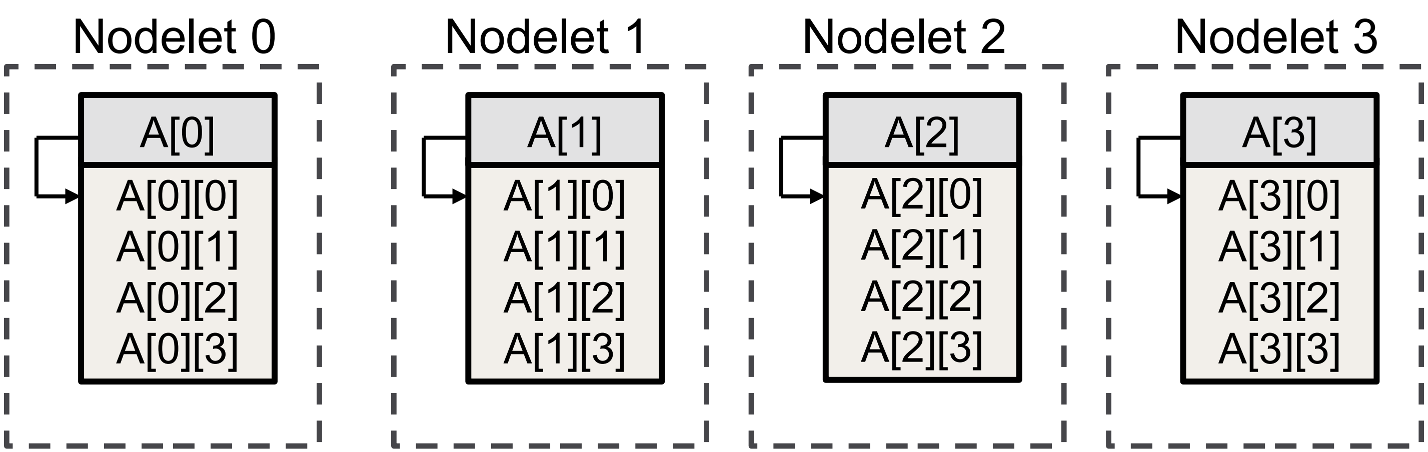
Malloc2d Ex: Array of structs

```
#define N 8
struct element { long a; long b; };
struct element ** E = (struct element **)
    mw_malloc2d(N, sizeof(struct element));
for (long i=0; i<N; i++) {
    E[i]->a = i;
    E[i]->b = 0;
}
```



Malloc2d Ex: Blocked Distribution

```
#define N 16  // # elements in array (power of 2)
long epn = N/NODELETS();  // elements per nodelet
long ** A = (long **)
    mw_malloc2d(NODELETS(), epn * sizeof(long));
for (long i=0; i<N; i++)
    A[i/epn][i%epn] = i;
```



Malloc2d Ex: Wrapped Block Dist.

```
#define N 16
long nBlocks = 8;
long blockSize = N/nBlocks;
long ** A = (long **)
    mw_malloc2d(nBlocks, blockSize * sizeof(long));
for (long i=0; i<N; i++)
    A[i/blockSize][i%blockSize] = i;
```