EMU Chick Overview and Hands-On

4/14/19

Hosted by Dr. Jason Riedy, Dr. Jeffrey Young (GT) with assistance from

Janice McMahon (Emu)

**Emu-related tutorial 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 **(9-10 AM)**
  - Example 1: Hello World
    - Memory replication
    - Basic thread Spawning Strategies
- Programming for the Chick **(10-11:30 AM)**
  - Example 2: STREAM
    - Spawn strategies
    - Granularity
    - Locality awareness

EMU CHICK INTRODUCTION

Emu Innovation Overview

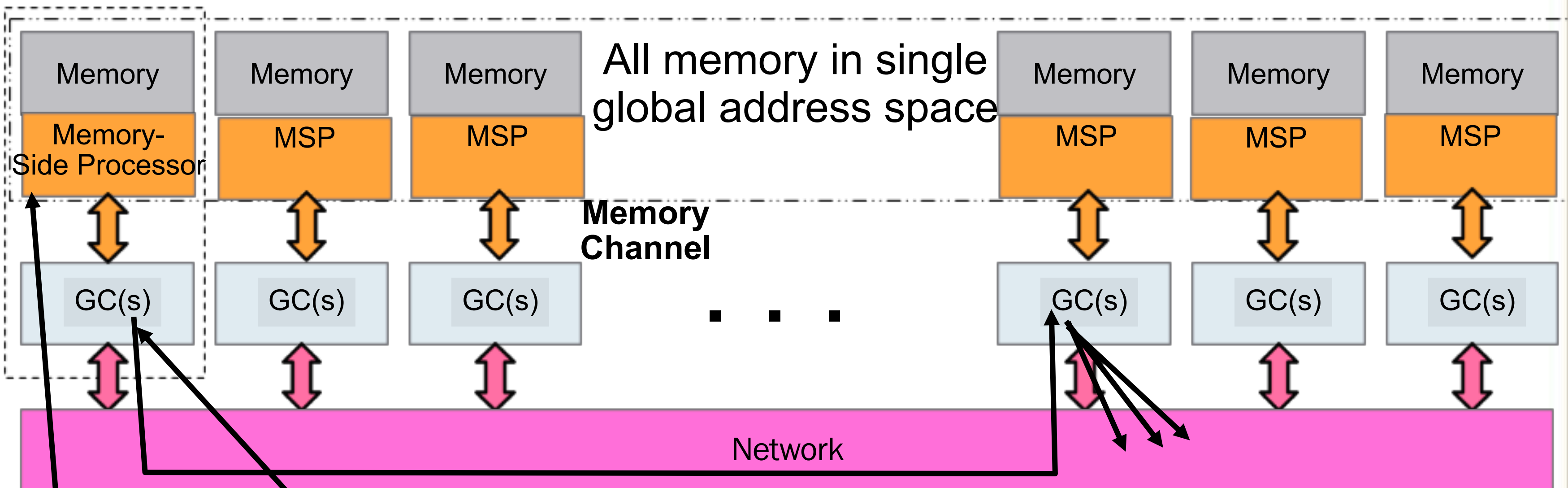
Designed from the ground up to deal with applications that exhibit little locality

- 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



Smart Memory Controllers
that also do atomics

Until they make a non-local reference
And then moved to **correct nodelet**

And they are free to spawn
independent children

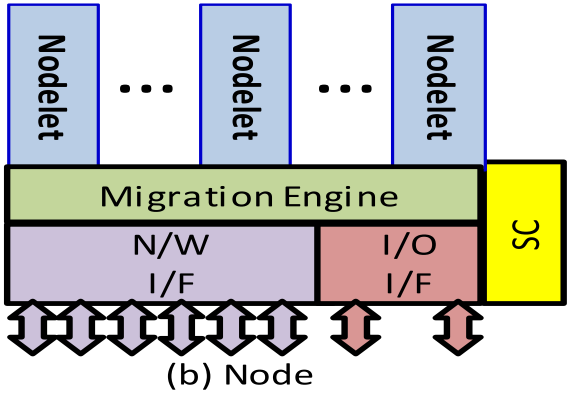
Threads execute **here**

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

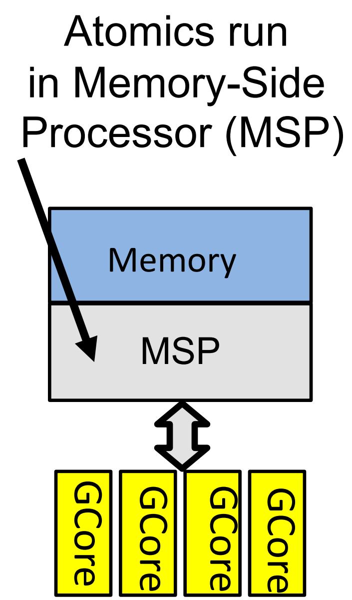
Stationary Core Runs OS, Launches Jobs



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



(a) Nodelet

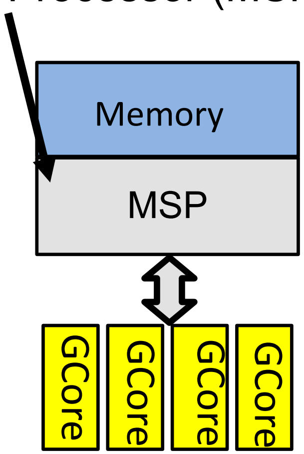
Multi-Threaded Cores

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

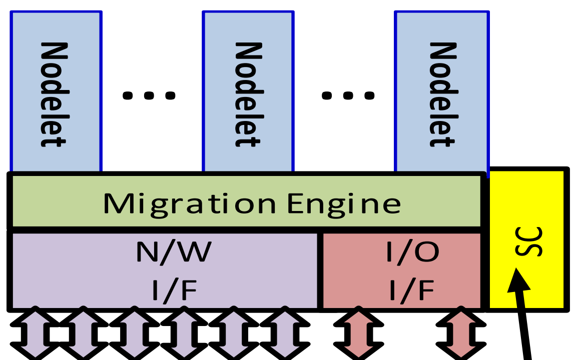
Emu System Hierarchy

Atomics run in Memory-Side Processor (MSP)



(a) Nodelet

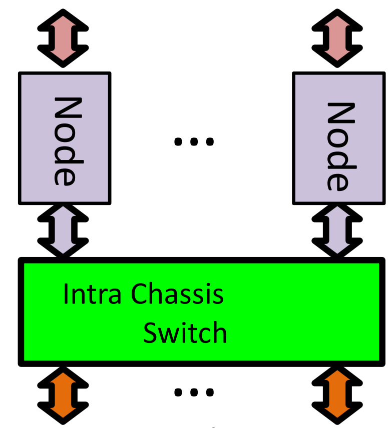
Multi-Threaded Cores



(b) Node

Migrating Threads are major traffic on Network

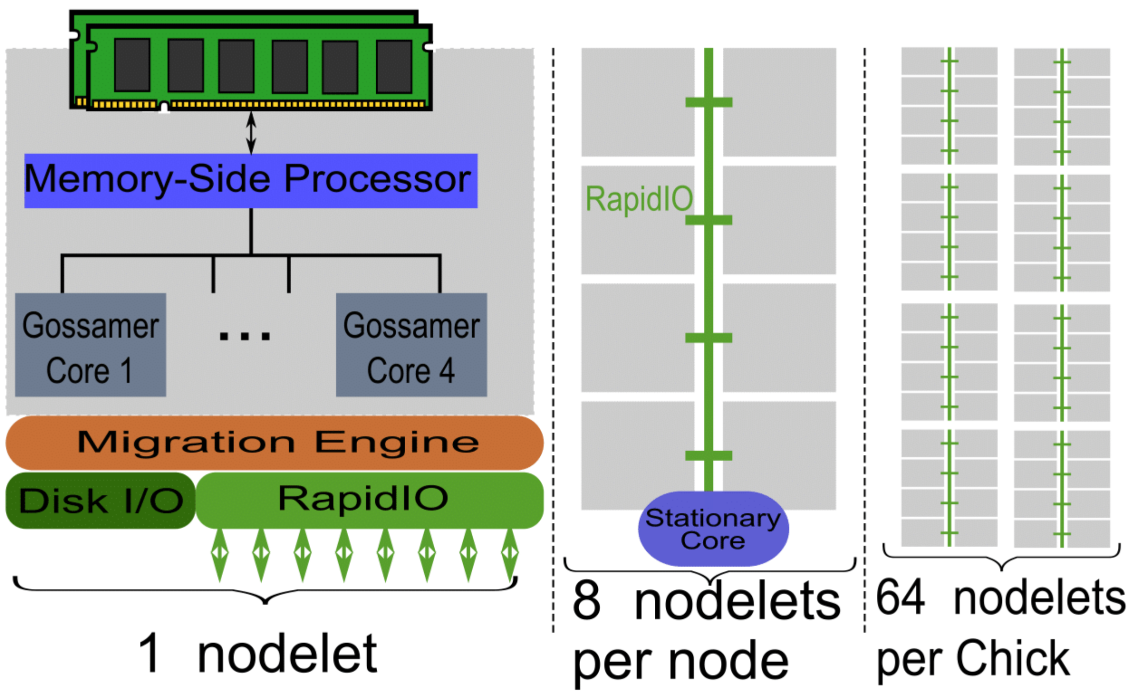
Stationary Core Runs OS, Launches Jobs



(c) Up Links to Inter-Chassis Switch

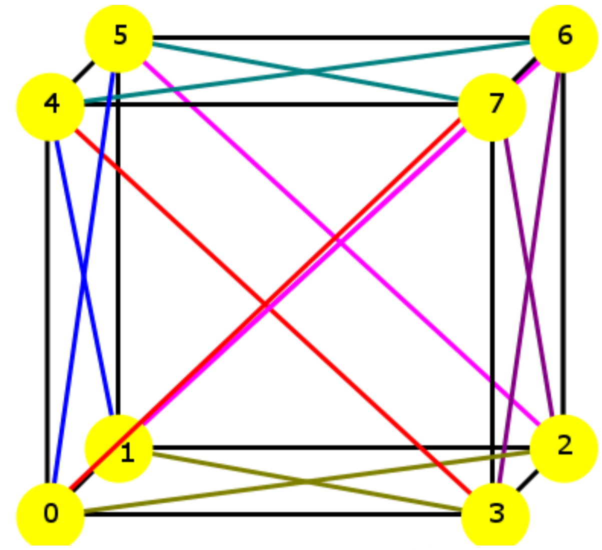
EmuTechnology

Emu Chick Architecture



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 less
  - 3D diagonals route through 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
- Currently available – v2 (Condor) is in development

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

See https://gitlab.com/crnch-rg/asplos-tutorial-2019 to work through this example via a Python notebook.

Chick Hands-On Information

Please refer to the related tutorial site at https://gitlab.com/crnch-rg/asplos-tutorial-2019 for hands-on information and notebook tutorials.

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

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

See https://gitlab.com/crnch-rg/asplos-tutorial-2019 to work through this example via a Python notebook.

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

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

EmuTechnology

Georgia Tech | Center for Research into Novel Computing Hierarchies

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

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 replicated data structure and n is the nodelet number

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

EmuTechnology

Georgia Tech Center for Research into Novel Computing Hierarchies

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

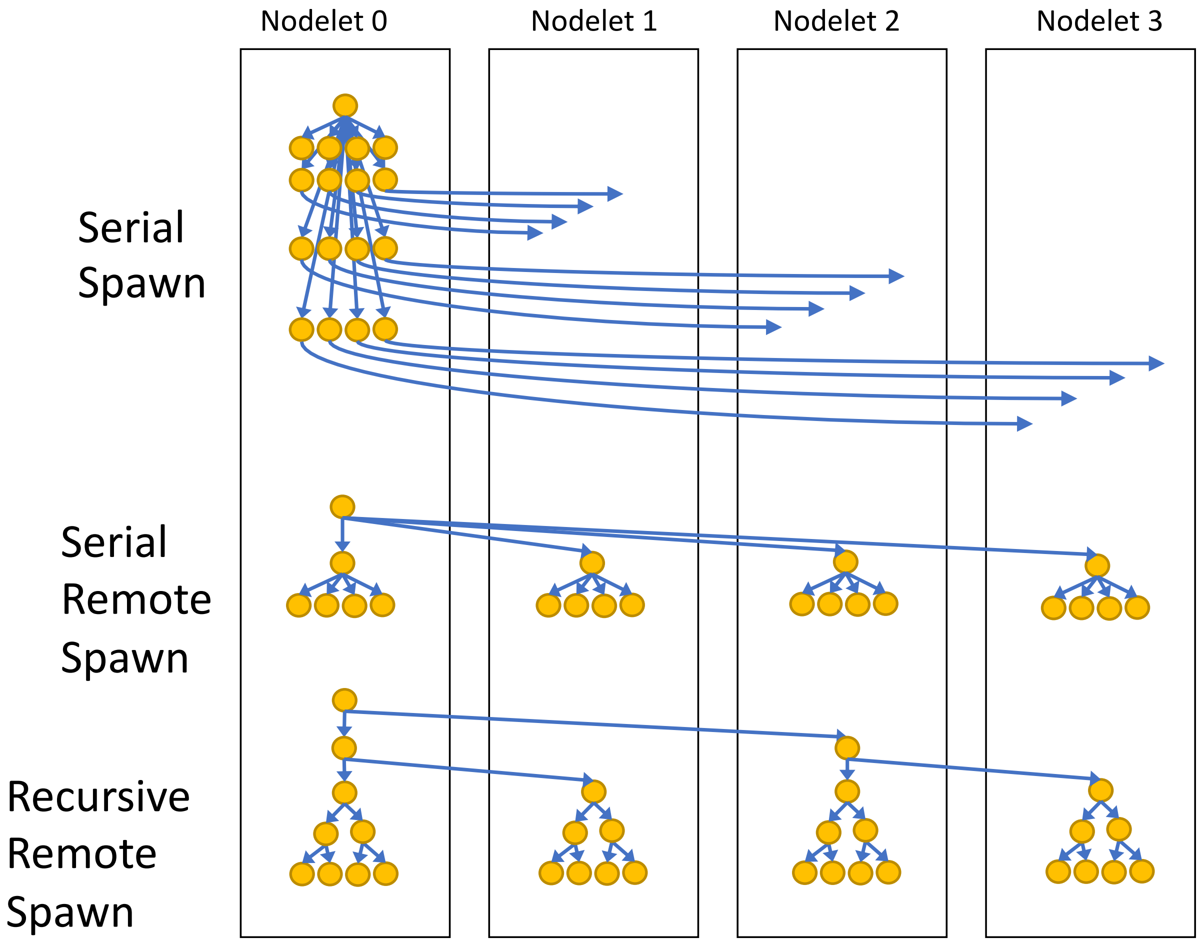
Georgia Tech Center for Research into Novel Computing Hierarchies

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



From Eric Hein's thesis: Near-Data Processing for Dynamic Graph Analytics, 2018

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];
}
```

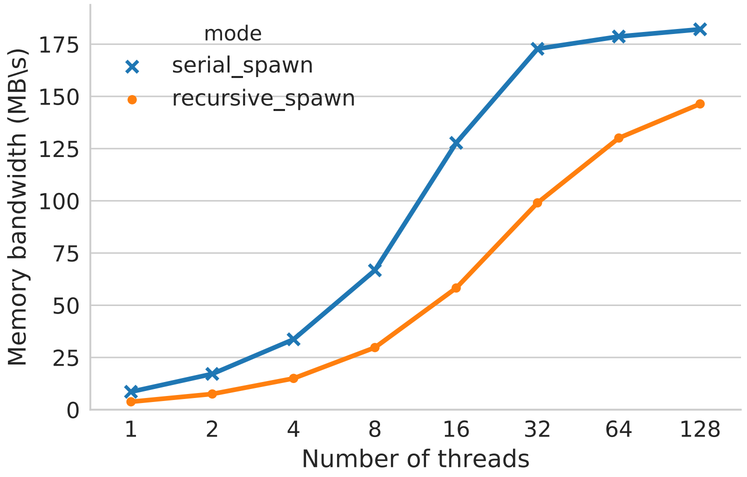
From Eric Hein's thesis: Near-Data Processing for Dynamic Graph Analytics, 2018

Ex: 2 Emu STREAM

This example demonstrates the following:

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

See https://gitlab.com/crnch-rg/asplos-tutorial-2019 to work through this example via a Python notebook.



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`

EmuTechnology

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

EmuTechnology

Georgia Tech | Center for Research into Novel Computing Hierarchies

System Services

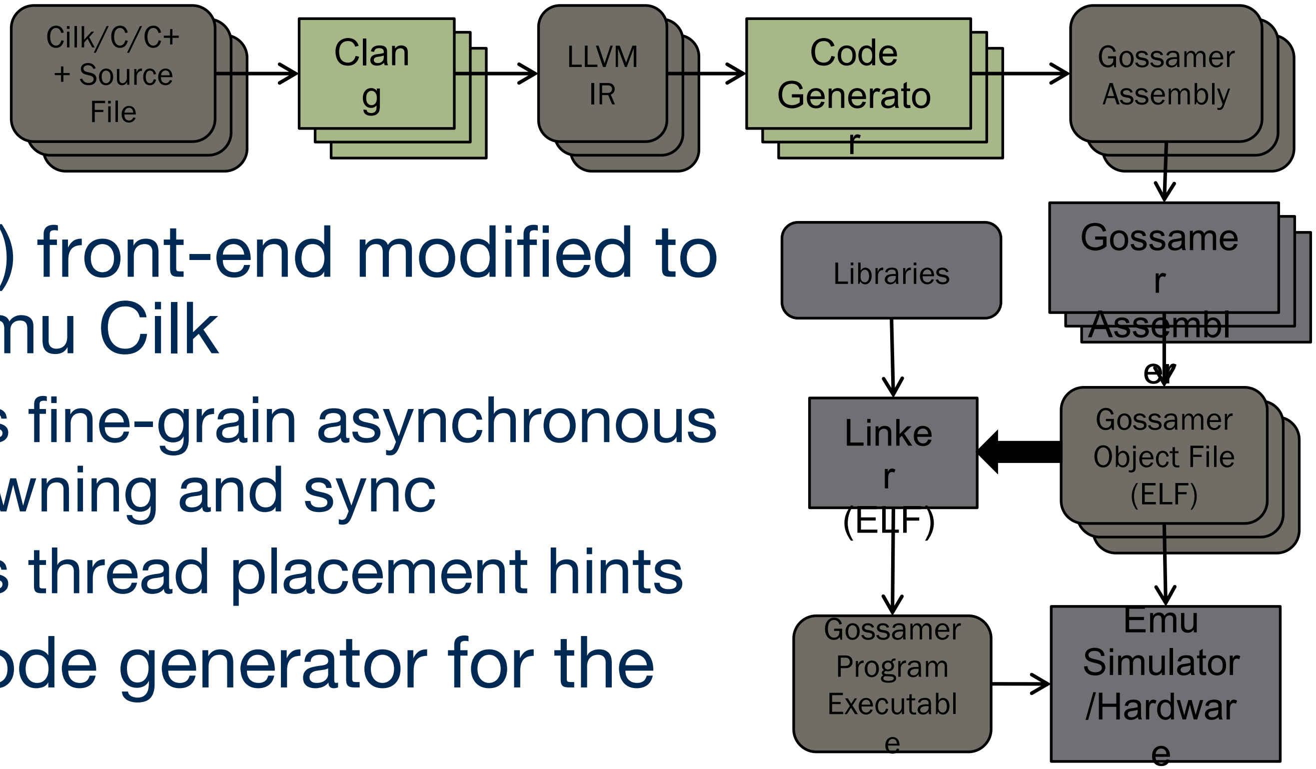
- Thread suspends itself and writes thread state registers (TSR) to the system exception queue (SEQ)
- Handler polls system SEQ for threads that need services
- Handler reads the thread from the SEQ and performs the requested service
- Handler then relaunches the thread to nodelet 0

Running Code on the Chick

- System: **karrawingi-login.crnch.gatech.edu**
  - SSH keys should be set up for you to use
- Username: asplos-$(seq 1 30)
  - *ssh asplos$@karrawingi-login.crnch.gatech.edu*
- Emu Chick nodes
  - *ssh n0 – n7*

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