

EmuTechnology

**Something Really New Is Ready.
Are You?**

Janice McMahon jmcmahon@emutechnology.com
Tutorial at IPDPS 2018, Vancouver, May 22, 2018



Agenda

- Emerging Applications
- Emu System Architecture
- Application Mapping Example
- Programming and Execution Model
- Emu Hardware Roadmap
- Software Development: Current and Future
- Open Cilk/Cilk Hub
- Detailed Programming



Evolution of Challenges Requires New Approaches to Solutions

EMERGING APPLICATIONS



Big Deal About Big Data

- Big Data refers to large, unstructured datasets containing huge amounts of disparate information
 - Often represented as graphs or sparse matrices
 - Many datasets are far too large to fit in a single memory system
- Applications search out relationships between data elements scattered throughout the dataset
 - Requires accessing data across many (100s or thousands) of memory systems
- Conventional computers are designed around an assumption that the vast majority of references are to local memory
 - **This is not the case for Big Data, so processing slows to a crawl**



Data Intensive Characteristics

- Computation dominated by data access & movement – **not flops**
- Large sets of data are often persistent
 - but little reuse during computation
- No predictable regularity
- Scaling to 100s of TBs and more
- Streaming often important



Applications Are Evolving

Benchmark Name	Function Performed	Conventional System Efficiency
LINPACK	Solve $Ax=b$, A is dense	>90% of peak
GUPS	Random updates	~10% of peak
HPCG: Hi Performance Conjugate Gradient	$Ax=b$, A sparse but regular	~2% of peak
SpMV: Sparse Matrix Vector	Ab ; A sparse and irregular	~2% of peak
BFS: Breadth First Search	Find all reachable vertices from root	~2% of peak
Firehose	Find “events” in streams of data	~1% of peak

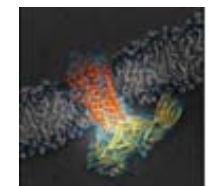
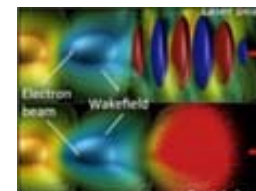
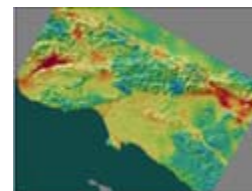
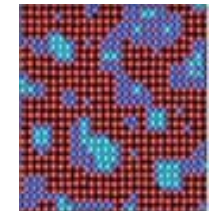
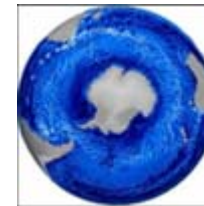
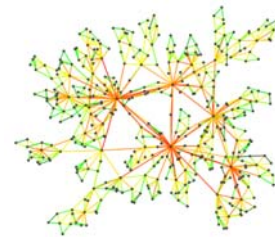
Emu system is efficient for data intensive applications

➤ Expect 20-90% of peak



Markets and Applications

Threat Intelligence
 Graph Analysis
 Big Data Analytics
 Risk and Fraud Analysis
 Signal and Image Processing
 Cybersecurity
 Semi / Unsupervised Learning
 NORA
 Real-time Pattern Matching
 Real-time Trend Analysis



Built Around The Data

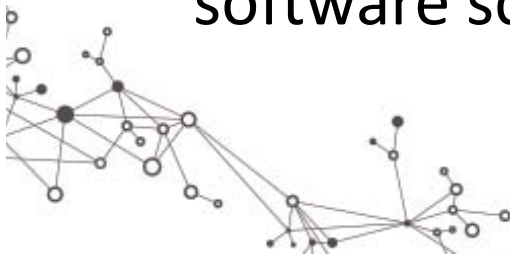
EMU SYSTEM ARCHITECTURE



Emu Innovation Overview

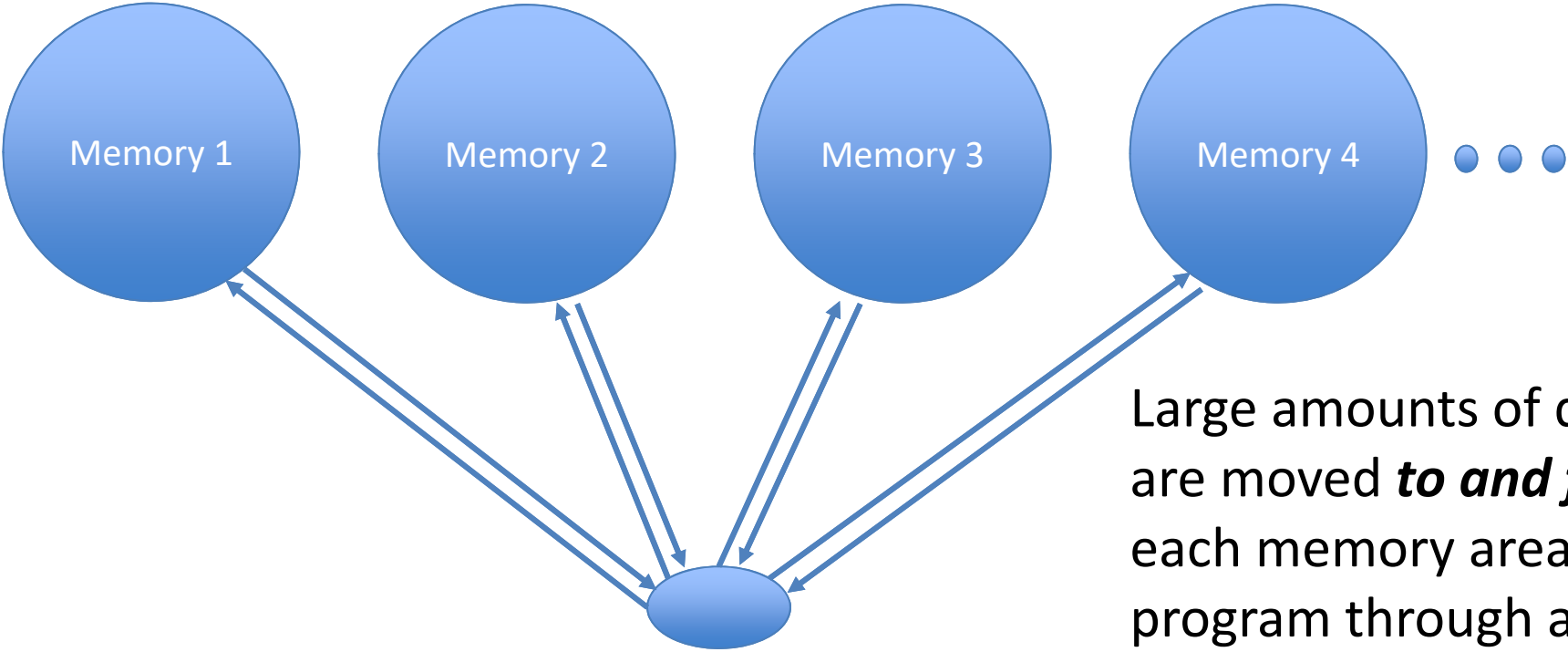
Designed from the ground up to deal with applications that exhibit little locality

- 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



Challenges of Conventional Designs

Massive amounts of data spread over many memories



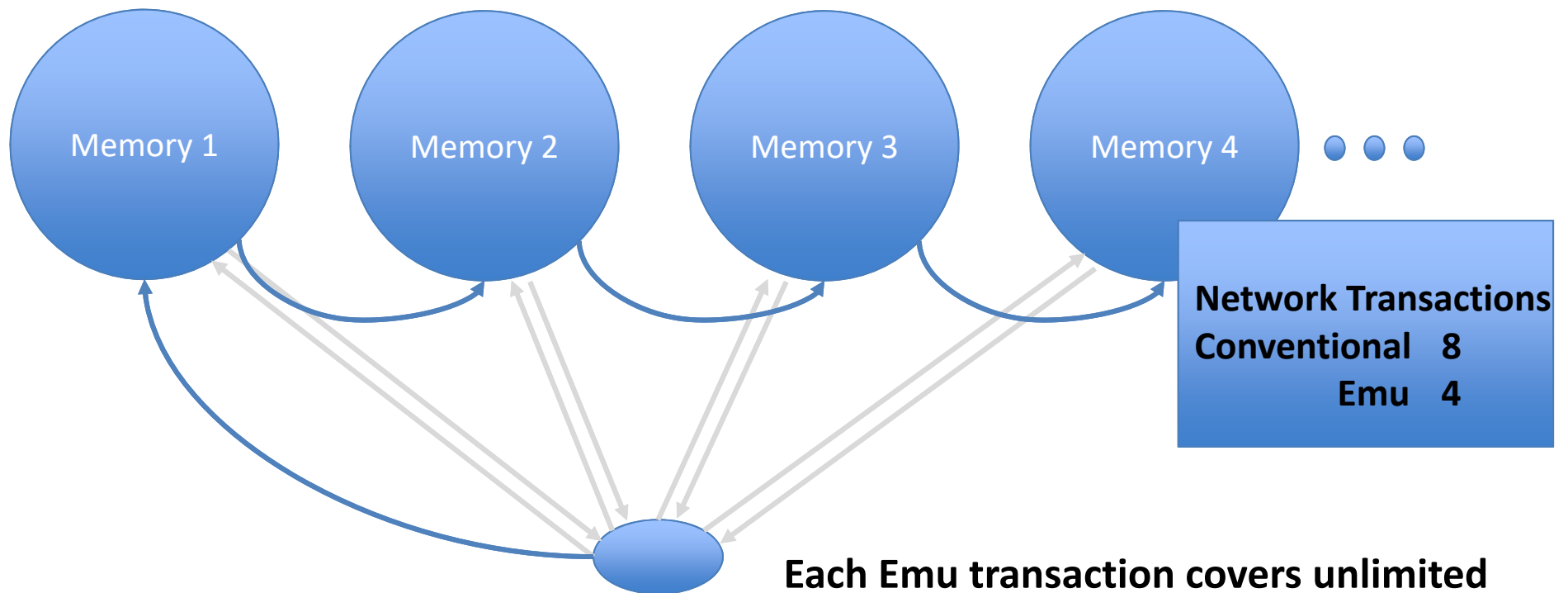
Large amounts of data are moved **to and from** each memory area to the program through a very limited network



(tiny compared to data)

How the Emu Computer Works

The program context moves from memory to memory resulting in 10x+ better usage of the network



Reducing Data Movement

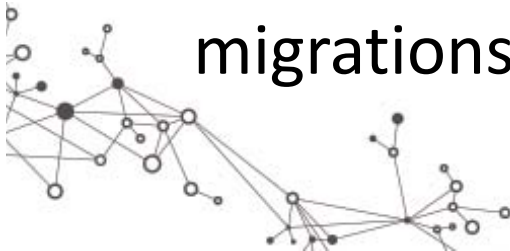
Gossamer cores migrate the program to the data vs. data to the processing element

- Move registers, thread status word, program counter
 - Application code replicated on each nodelet, never moves
- One-way trip
- Reference to non-local address triggers migration
 - Largely invisible to programmer
- Latency is completely hidden if sufficient active threads
- Writes are transmitted on network without migrating



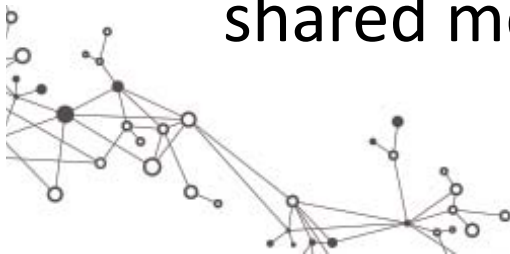
Big Win for Big Data

- Wins big when data access pattern is a series of often brief “visits” to widely dispersed data
- **Improved processor utilization**
 - Processors never stall for long periods waiting for remote reads
- **Simplified network**
 - Doesn't need to support round trip (read / response) messages
- **Atomic operations** always done “locally”
- **Remote Writes** can be performed directly or via migrations, under programmer (compiler) control.



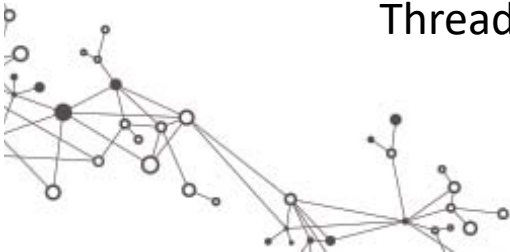
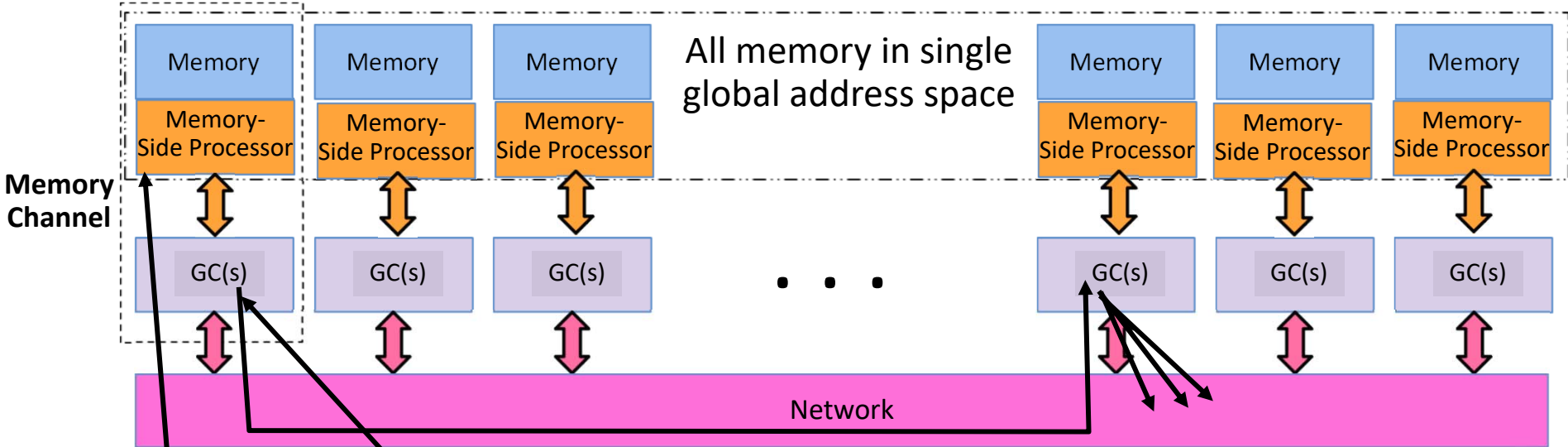
Highly Scalable Modular System

- Fine grain parallelism – scales to millions of cores
 - Single code base
 - Current design scales to over 2 Million cores
- 100X reduction in interprocessor communications
- Partitioned Global Address Space (PGAS) to Petabytes of memory
- Cacheless system
 - Eliminates cache coherency
- High radix RapidIO network provides system-wide shared memory environment



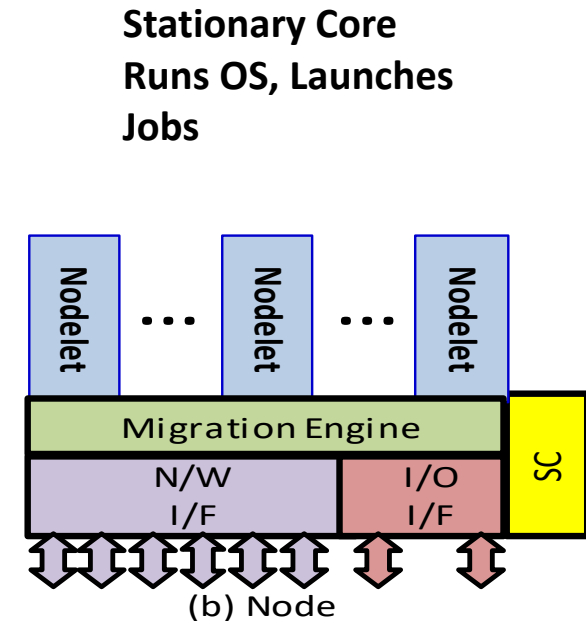
Emu Architecture Functional Diagram

Nodelet: New unit of parallelism



Node Architecture

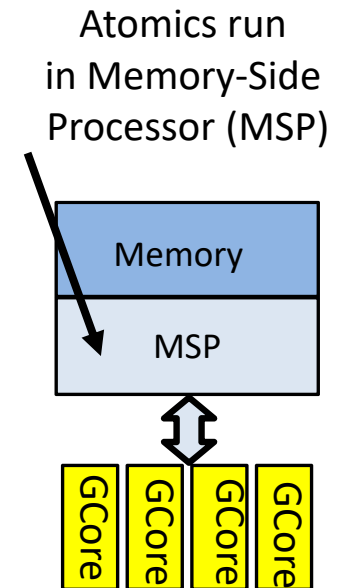
- 8 Nodelets
- Migration Engine
- 6 RapidIO 2.3 4-lane network ports
- Stationary Cores (SCs)
 - DualCore 64-bit Power E5500
 - 2GB DRAM
 - 1 TB SSD
 - PCIe Gen 3
 - Runs Linux



Migrating Threads
are major traffic
on Network

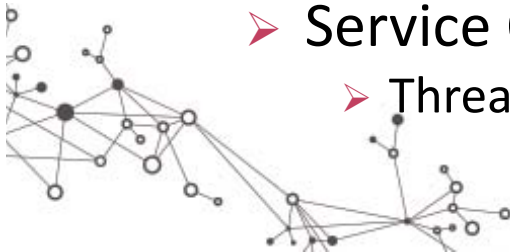
Nodelet Architecture

- 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**



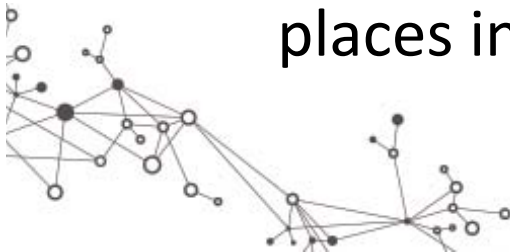
Gossamer Core Architecture

- Deeply pipelined, multithreaded core
 - Custom, accumulator-based ISA
 - Support for 64 active hardware threads
 - Thread Context
 - Program Counter
 - Registers
 - Thread status words
- Multithreading hides instruction latency, including local memory operations



Hardware Thread Management

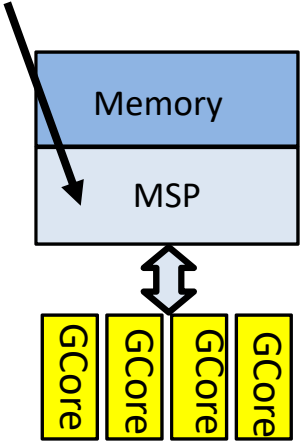
- Thread scheduling in GCs automatically performed by hardware
- SPAWN instruction
 - Creates new thread and places it in Run Queue
- RELEASE instruction
 - Places thread in Service Queue for processing by SC
- Non-local memory reference causes a migration
 - Thread context packaged by hardware and placed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



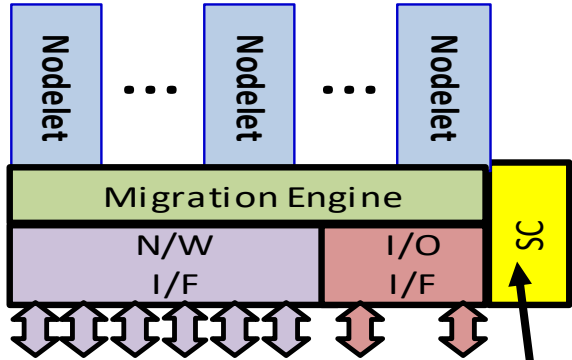
Emu System Hierarchy



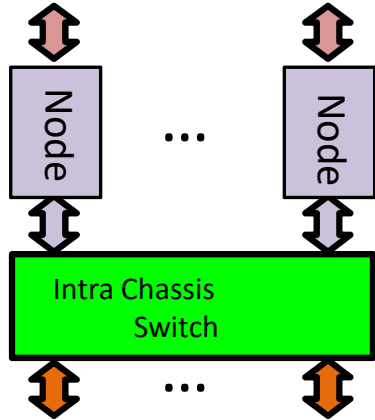
Atomics run in Memory-Side Processor (MSP)



(a) Nodelet



(b) Node



(c) Intra Chassis Switch

Multi-Threaded Cores

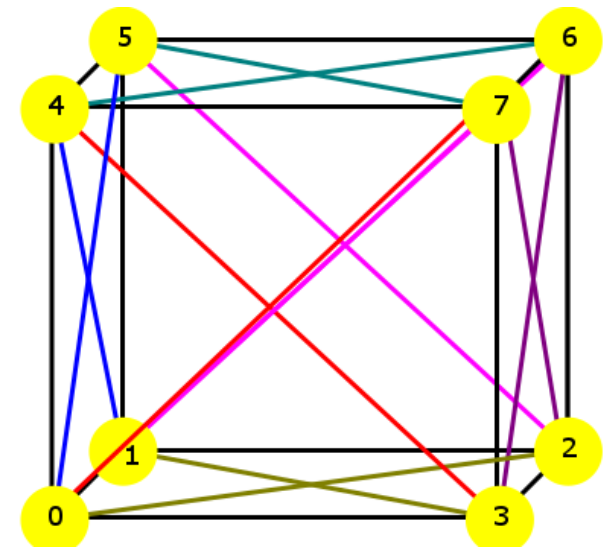


Migrating Threads are major traffic on Network

Stationary Core Runs OS, Launches Jobs

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



System Software

- LINUX runs on the Stationary Cores (SCs).
- OS launches `main()` user program on a Gossamer Core (GC)
 - `main()` then spawns descendants that execute in parallel and migrate throughout system as needed
- Runtime executes primarily on the SCs
 - Handles service requests from threads running on the GCs including: memory allocation, I/O, exception handling, and performance monitoring
- Threads return to `main()` upon completion, which then returns to the OS



APPLICATION EXAMPLE: SPARSE MATRIX-VECTOR MULTIPLY



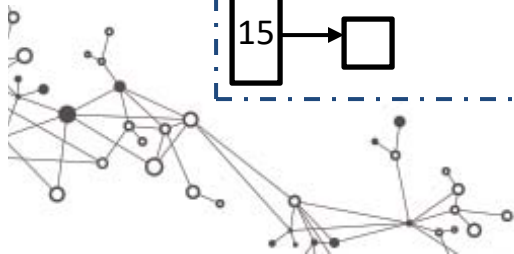
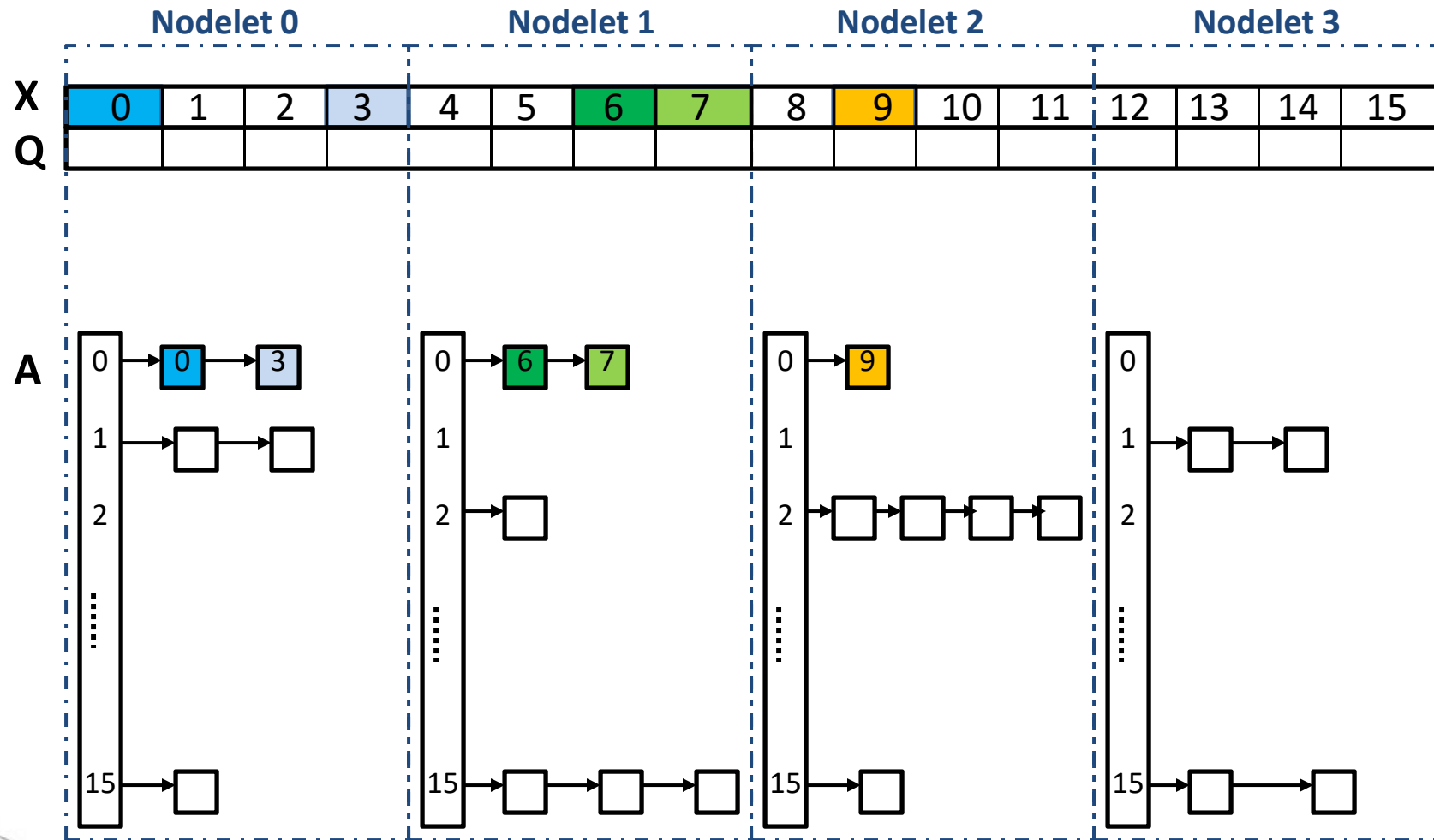
Sparse Matrix Vector Multiply (SpMV)

➤ $Q = AX$

- Distribute vectors X and Q across nodelets in contiguous blocks
- Distribute matrix A as an array of row pointers to the nonzero elements in that row on each nodelet



$$Q=AX$$

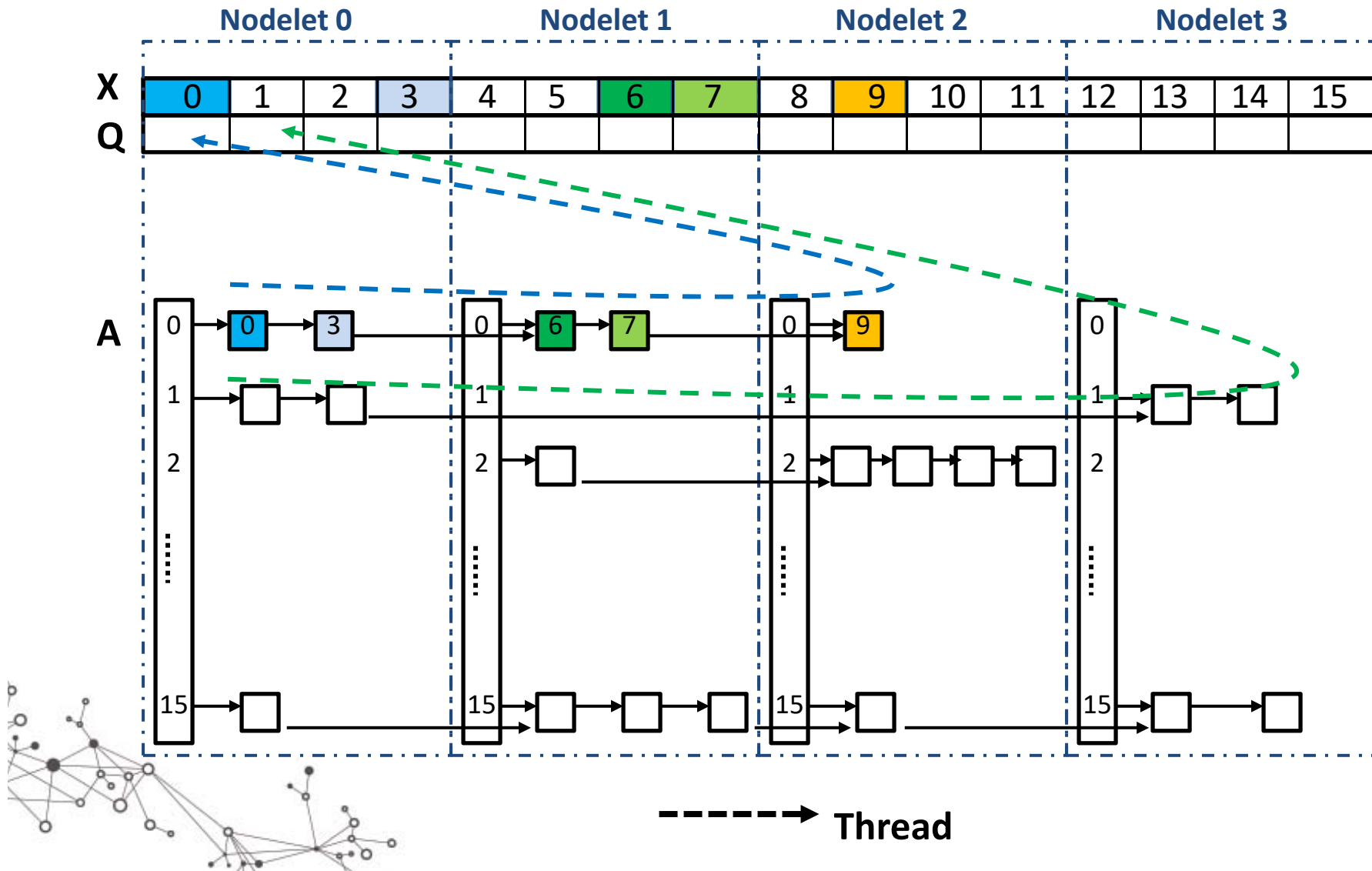


Migrating SpMV

- Thread computes a row, migrating to follow linked list
- For row i
 - Compute $\text{sum} += A[i][j] * X[j]$ for entire row
 - Migrate to $Q[i]$ and add sum
 - Repeat for row $i+1$



Q=AX (Migrating threads)



Debugging and Optimization

- If all threads spawned on nodelet 0
 - Hotspot
 - Limited parallelism – threads proceed in lock-step
 - Solution: start threads at different nodelets
- If matrix is extremely sparse
 - Cost of row headers with no elements can be high
 - Solution: Add row index to represent only non-empty rows
- Migration pattern
 - May primarily use channel btwn nodelet i and $i+1$
 - Limits available bandwidth

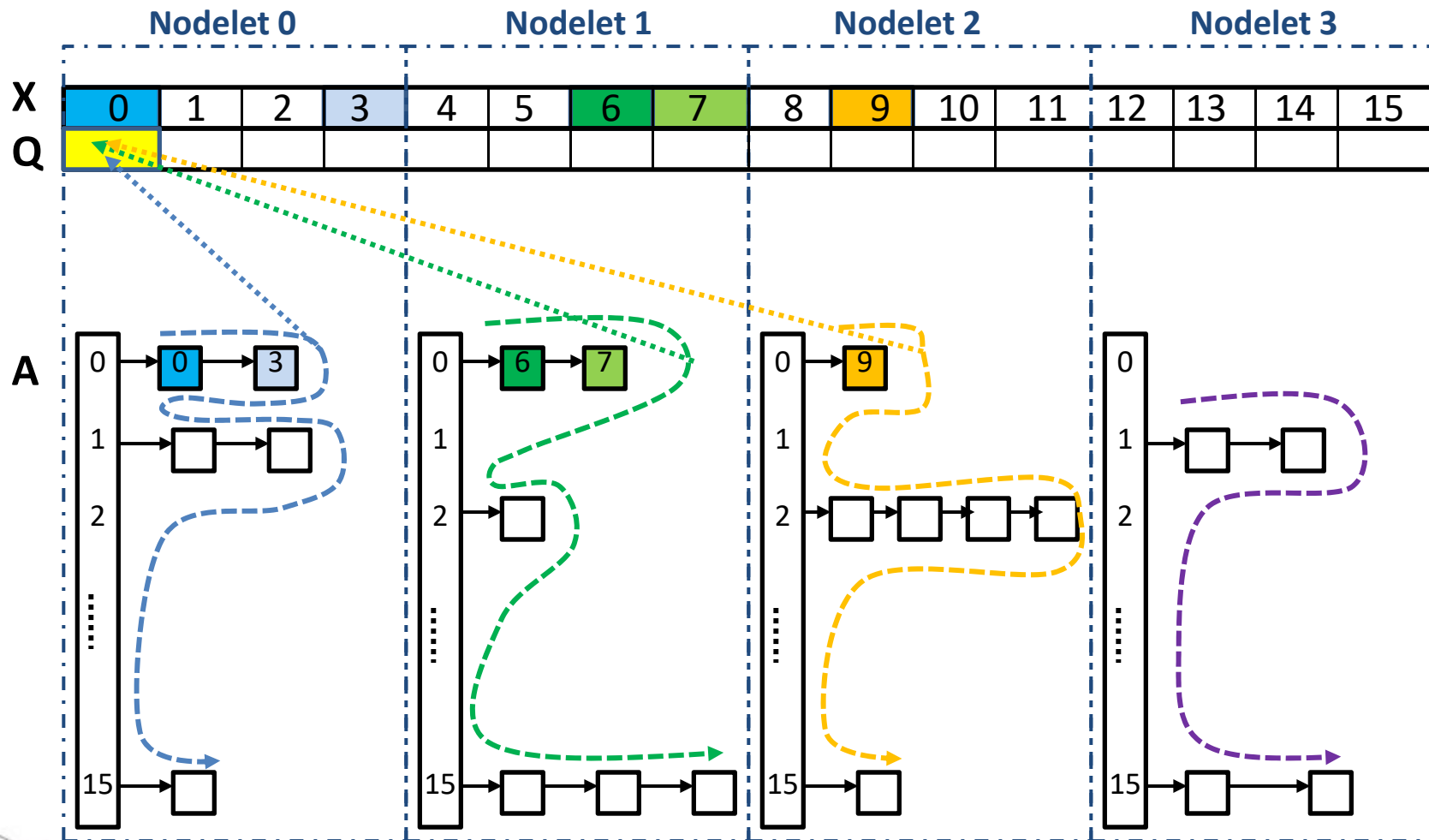


SpMV Using Remote Updates

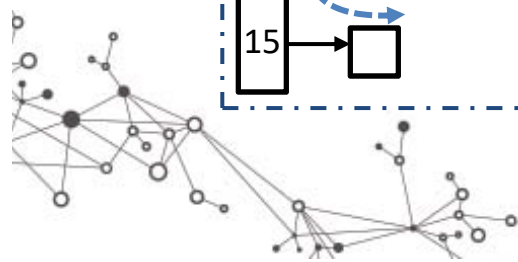
- Thread stays local to a single nodelet
- For row i
 - Compute $\text{sum} += A[i][j] * X[j]$ for the nonzero elements on this nodelet
 - Send a remote add to do $Q[i] += \text{sum}$
 - Repeat for row $i+1$



First Row of Matrix Multiply ($Q=AX$)



- - - - -> Thread
> Remote Add



Characteristics of Remote Updates

- More uniformly uses network channels
- Remote updates are typically smaller than migrations, less network traffic
- Fewer threads needed because no migration delay to mask
- May have multiple updates to Q per row, rather than the 1 update for migrating threads



PROGRAMMING AND EXECUTION MODEL



Programming **Emu**

- The **Emu** architecture is designed to address large data problems that can be expressed as highly multithreaded algorithms
- Graph or Sparse Matrix representations work equally well
- **Emu** Cilk extends C for asynchronous parallel threading



Emu's Migratory Thread Model

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

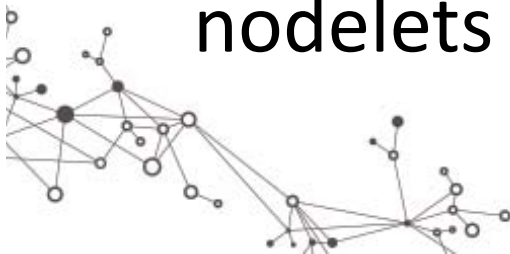
Key Issues:

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



Key Features

- **Cilk**: Extensions to C to support thread management
 - `cilk_spawn`
 - `cilk_sync`
 - `cilk_for`
- **Intrinsics**: Allow access to architecture specific operations such as atomic updates
- **Memory allocation library**: Specialized `malloc/free` for data distributed across nodelets



Emu Cilk

Emu hardware dynamically creates and schedules threads

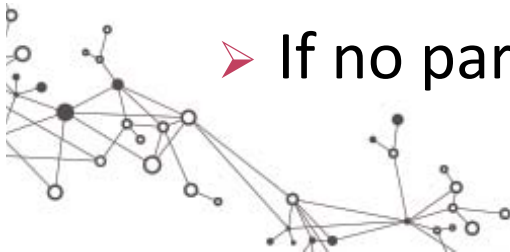
- Normally requires no software intervention
- When a thread completes, it returns values to its parent and dies
- When a thread blocks, it may voluntarily place itself at the back of the run queue (instead of “busy waiting”)
- Number of threads limited only by available memory
- Extremely lightweight – 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
 - First address parameter indicates spawn location
 - If no parameters are pointers, 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

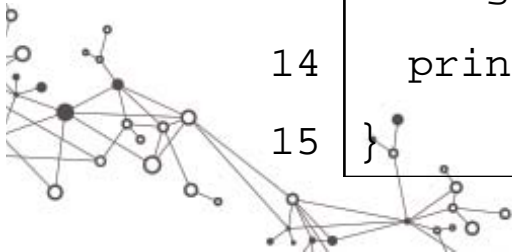


Cilk Fibonacci Example

```
1  #include "memoryweb.h"
2  #include "cilk.h"
3  #define N 10
4  long fib(long n) {
5      if (n < 2)
6          return n;
7      long a = cilk_spawn fib(n-1);
8      long b = cilk_spawn fib(n-2);
9      cilk_sync;
10     return a + b;
11 }
12 int main() {
13     long result = fib(N);
14     printf("fib(%d) = %ld\n", N, result);
15 }
```

Spawn a thread
for each of the
fib() calls

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



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 nodelet

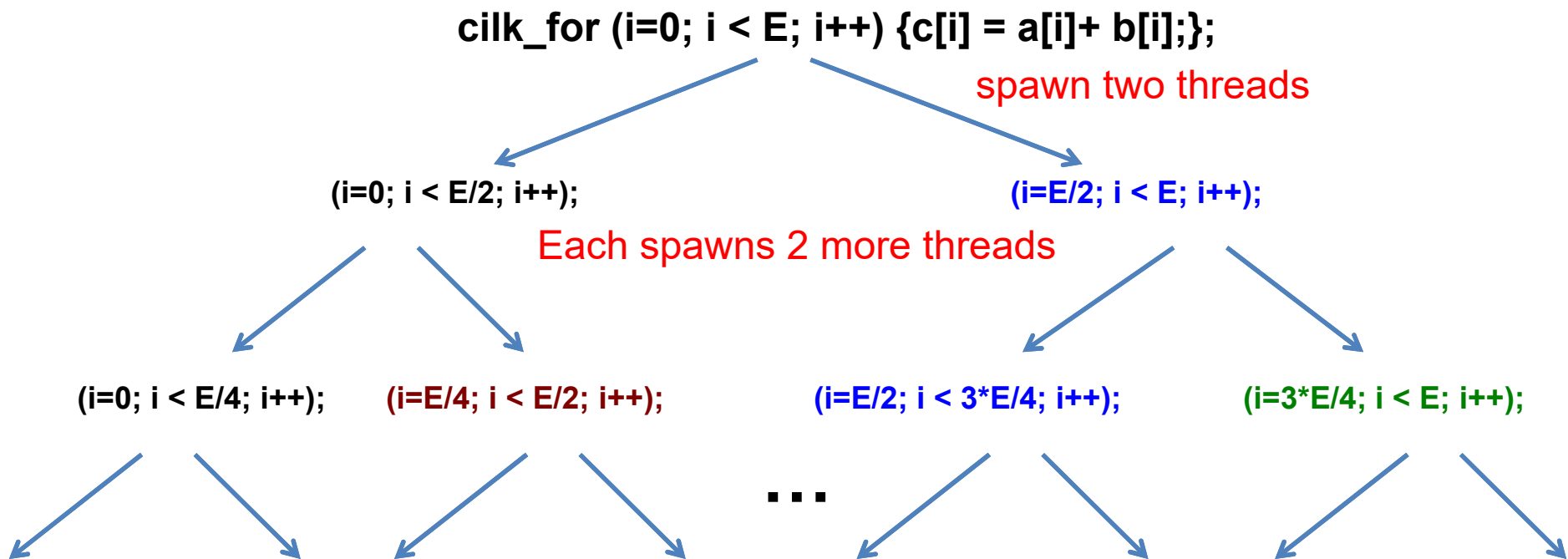


Efficient Spawning

- Distribute spawns across the nodelets 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 nodelet if spawning across multiple nodelets
 - Used in `cilk_for`



Cilk_for Spawn Tree Parallelization



Spawning Static Thread Teams

- Spawn a “team” of threads at each nodelet
- Each thread has a “home” nodelet
 - 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



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



Memory

- Single, shared address space (PGAS)
- Capability to define memory **Views** and place data in those Views
 - Private automatic variables declared normally in Cilk
 - Support for replicated data and allocation of distributed data structures



Memory Allocation

- Data sections defined 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 **different** values
 - Use only when it does **not** matter which instance you access!
 - EX: random number table, pointer to local work queue



Global 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)
```

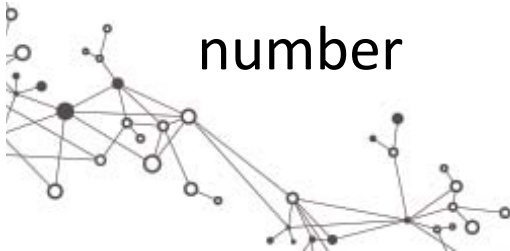
- Initializes each instance of replicated data structure to value

```
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



Local Memory Allocation

- Allocate from the stack section 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)

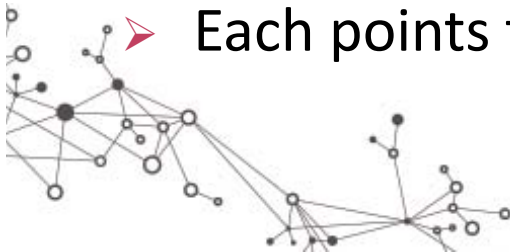
- Block of memory located in same locale as another data structure

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 *allocatedpointer)

- Free data allocated by mw_malloc2d

void mw_localfree(void *allocatedpointer)

- Free data allocated by mw_localmalloc



EMU HARDWARE ROADMAP



The Current Chick Hardware



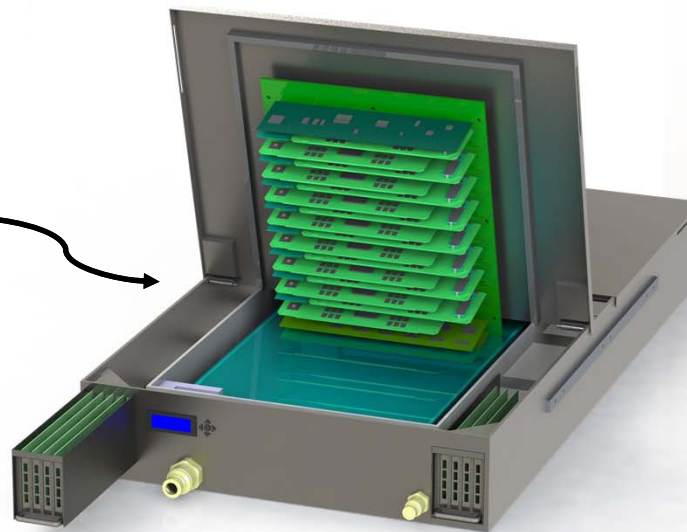
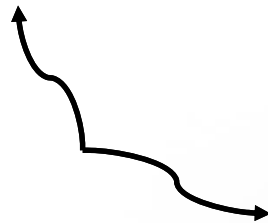
- **8 nodes (64 nodelets)**
- **512 GB Shared Memory**
- **8 TB SSD**



Emu Hardware Roadmap



The node boards are the same



Emu Chick

- 8192 concurrent threads
- Copy room environment
- Shipping now



Emu Rack

- >260000 concurrent threads/rack
- Server room environment
- In development

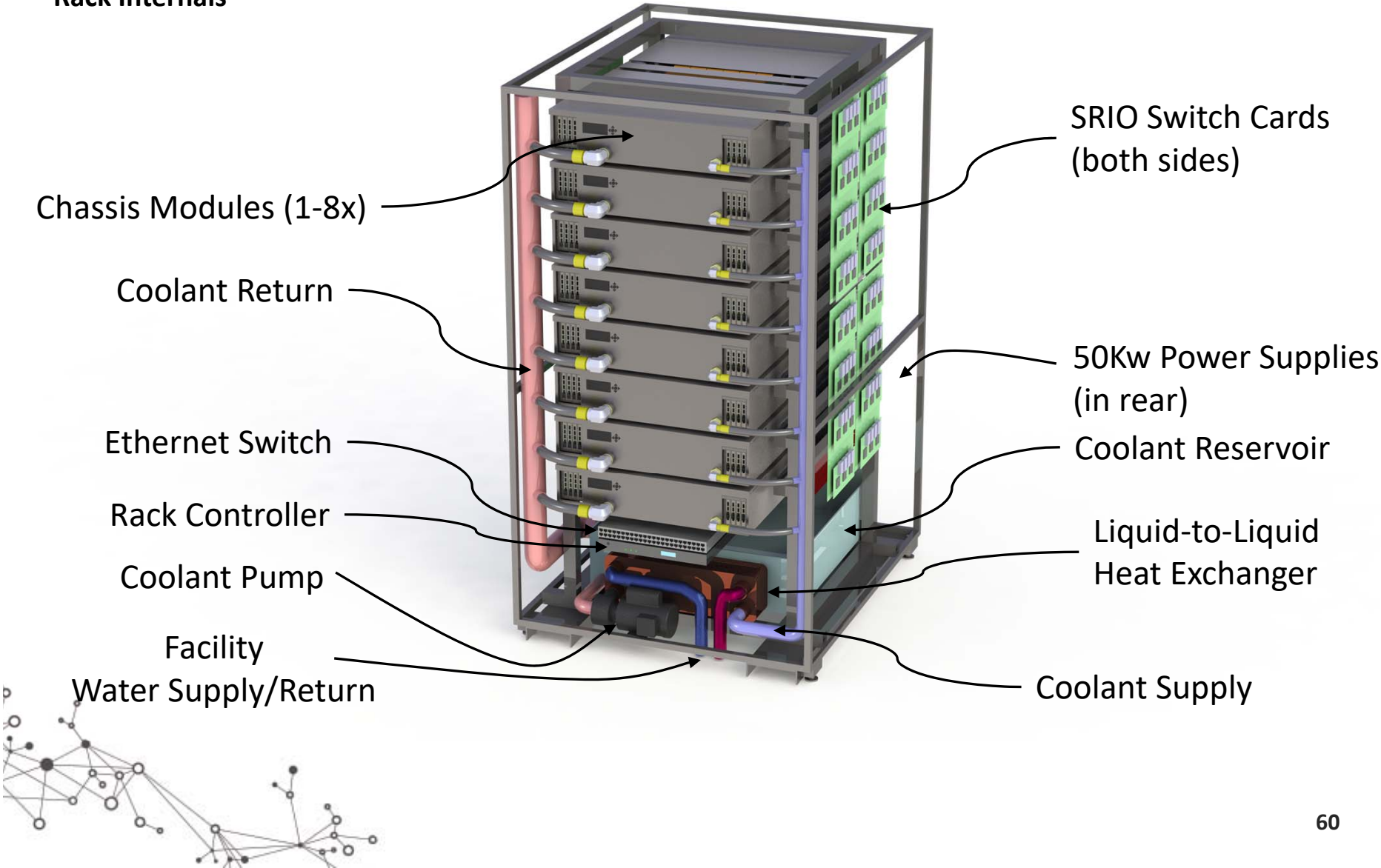
Emu Rack

- 32 to 256 Gossamer-A / S Node Cards per rack
 - 32, 64, 128, or 256 nodes
 - Single system images to 256 racks (64K nodes)
- 64 / 128 GB DDR4 DRAM per node
- SRIO Gen 3 Switched Network Fabric (node links are SRIO Gen 2 for Gossamer-A nodes)
- Up to 16 PCIeexpress slots per Chassis (128/rack)
- Immersion cooled with OptiCool Fluid coolant
 - Option for liquid-liquid or liquid-air heat exchanger



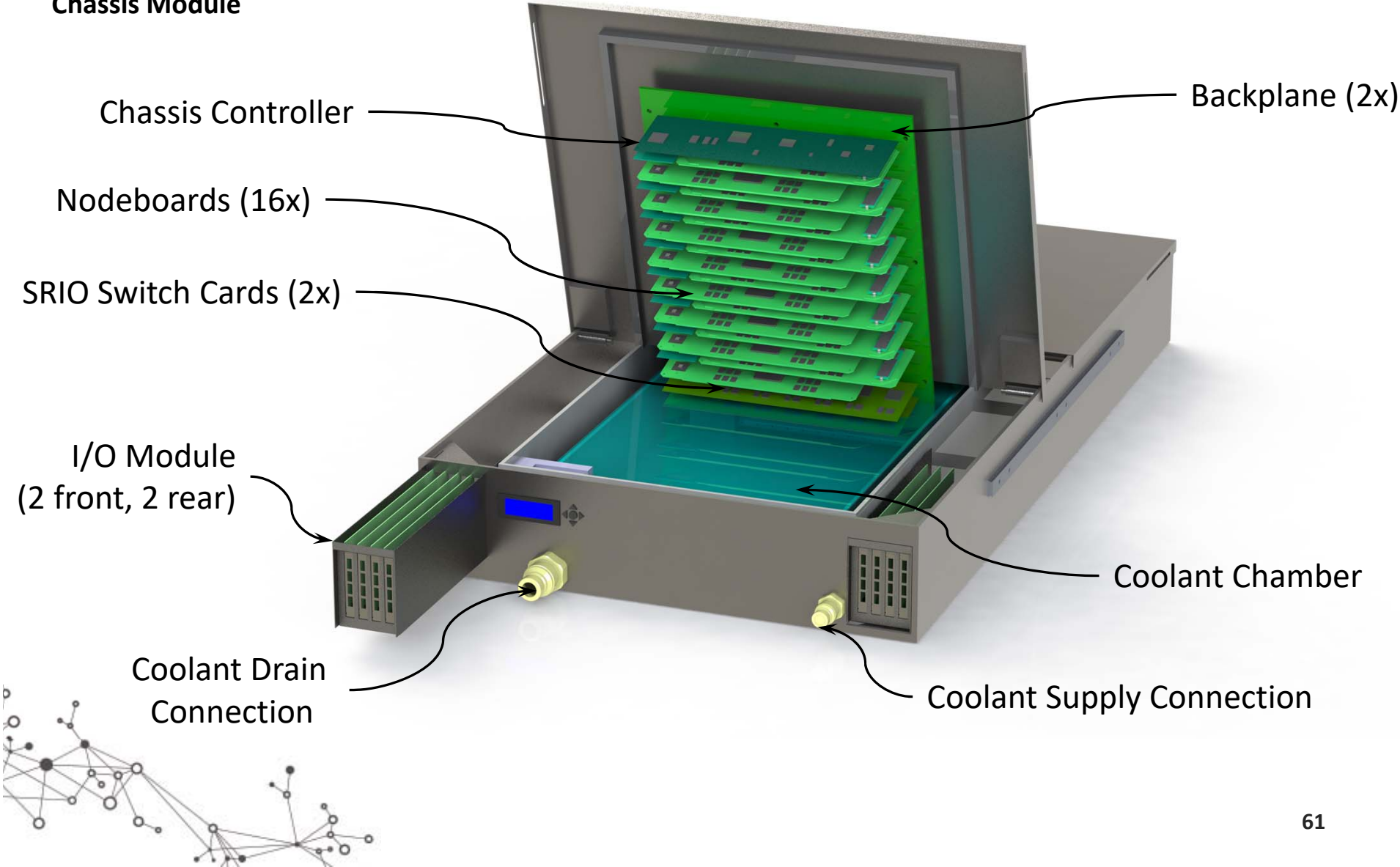
Rack Internals

Rack Internals



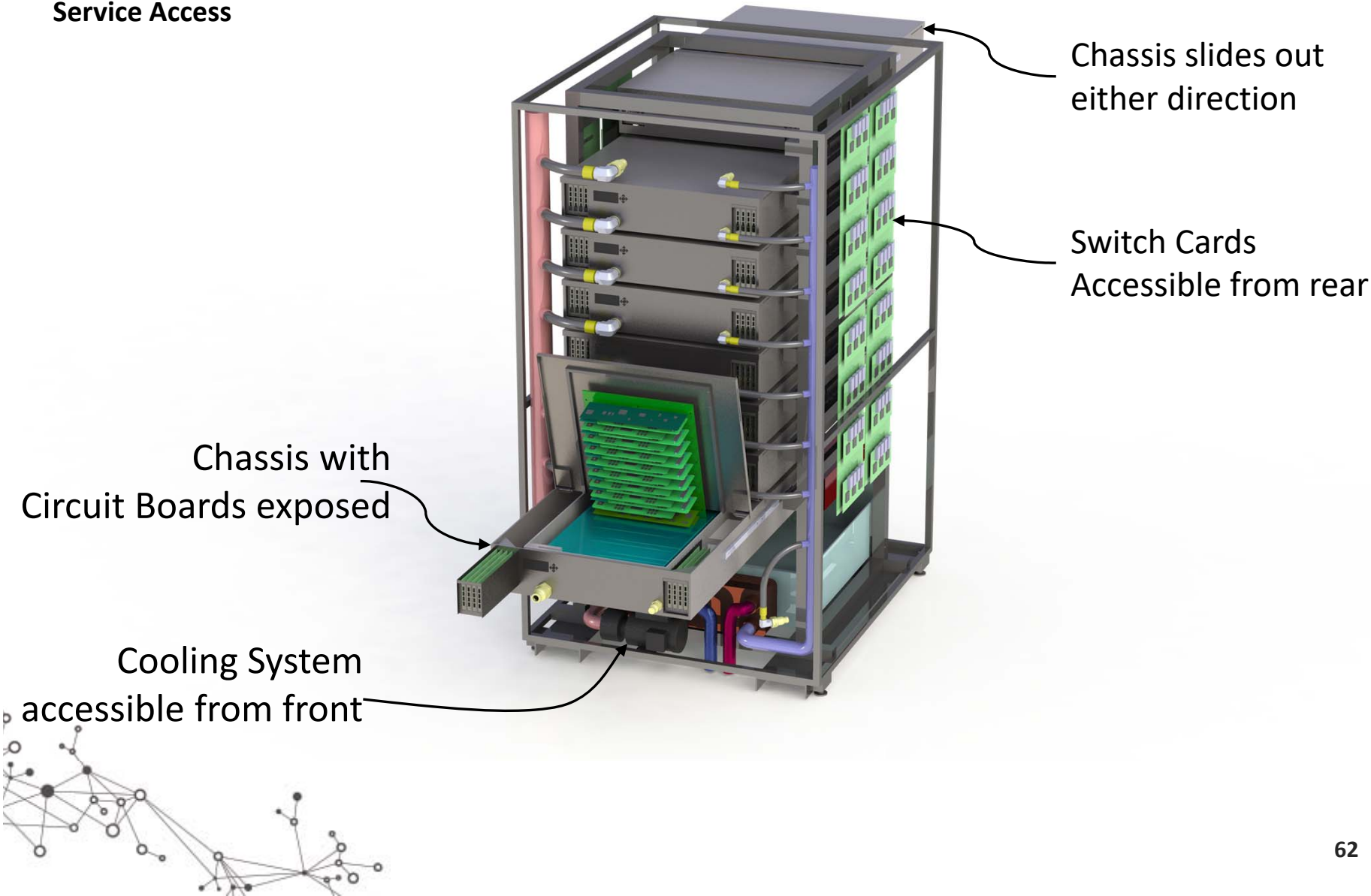
Chassis Module

Chassis Module



Chassis Module

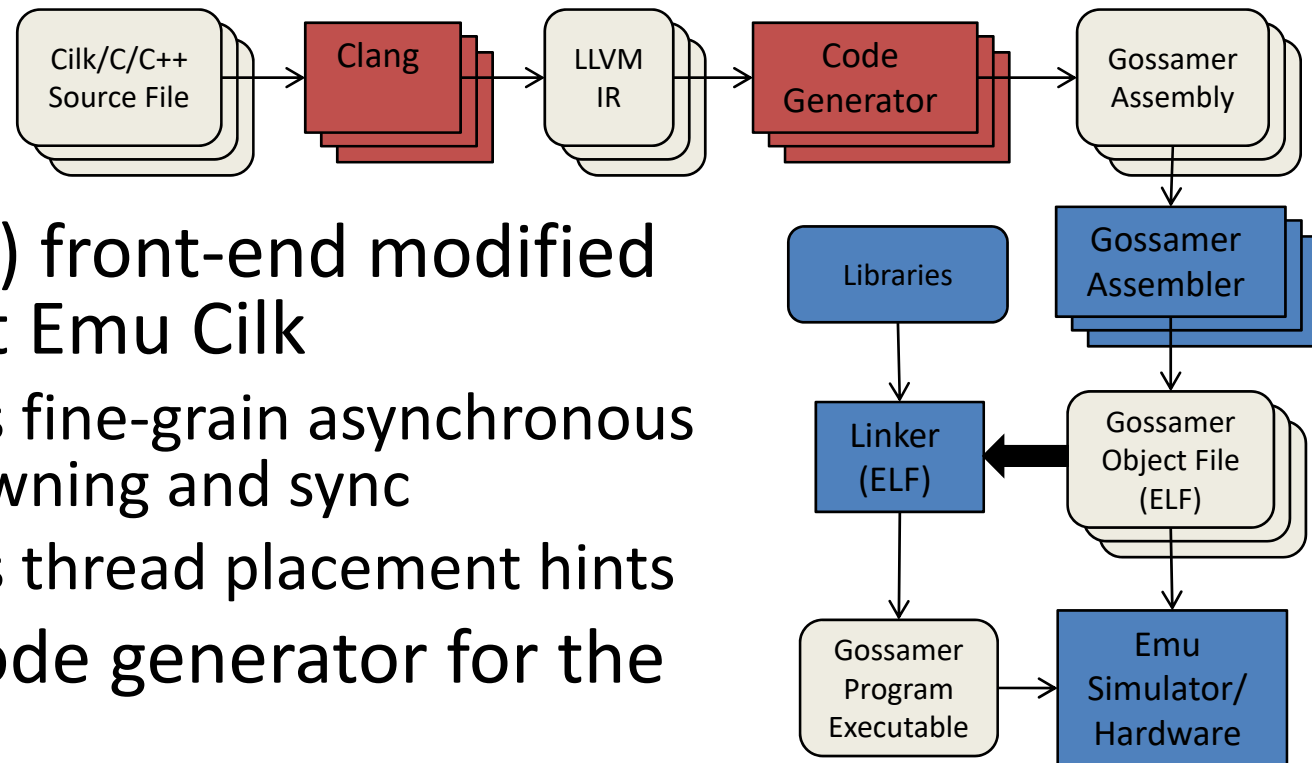
Service Access



SOFTWARE DEVELOPMENT: CURRENT AND FUTURE



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**



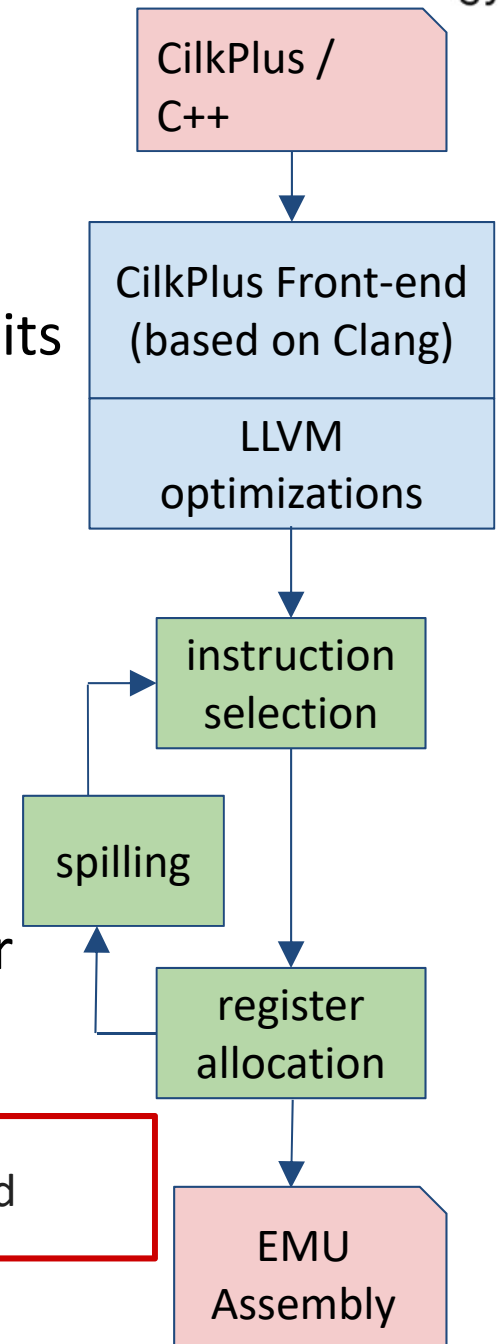
Custom Code Generator for EMU

- Accumulator-based architecture, emphasizing small code (smallest instruction is a nibble)
- Two accumulators and 16 GP registers, all 64-bits
- LLVM's code generator is ok for traditional architectures, but lacks flexibility to effectively accommodate out-of-the-box ideas

Custom approach for code generation

- Instruction selection using BURG techniques
- Register allocation via graph coloring
- Trivial scheduling (no VLIW/SIMD/superscalar)
- Integrated, overcoming traditional phase-order problems

Good algorithms, combined in new ways, carefully implemented



Emu Compiler Features

- Thread spawn and migration via Cilk
- Manages cactus stack
- Manages limited register set
- Limits register spilling due to migration
- Use of remote write instructions vs. migrating store
- Thread re-sizing



Standard C Library

- Port of musl-libc
 - <http://www.musl-libc.org/>
 - Prioritize most frequently used functionality
 - No support for pthreads



Standard C++ Library

- Port of libcxx: <https://libcxx.llvm.org/>
 - Supports most frequently used functionality:
 - Containers – array, deque, forward_list, unordered_set, vector
 - General – algorithm, chrono, iterator, tuple
 - Language support – limits, new, typeid
 - Numerics – valarray, numeri, ratio
 - Strings
 - Streams
 - No support for
 - Exception handling (e.g. throw/catch)
 - Atomic operations for data types less than 64 bits
 - Distributed containers

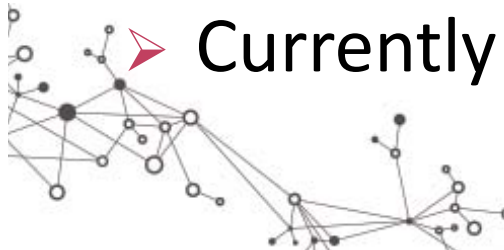
➤ Testing and debugging is ongoing.



CilkPlus Under Development

- Latest Clang front-end to support
 - C/C++ 11, 14
 - CilkPlus Runtime
- 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 being debugged



Toolchain Next Steps

- Complete CilkPlus debugging
- C++ exception handling
- Extend toolchain testing coverage
- Optimize frequently used features such as `cilk_for` and reducer implementations



User Libraries

- GNU Multiple Precision Arithmetic (GMP) Library
 - Library for arbitrary precision arithmetic
 - Currently support integer GMP for Emu
- Exploring
 - Ligra Graph Library for Shared Memory (Cilk)
 - <http://jshun.github.io/ligra/docs/introduction.html>



Collaborations

- STINGER Graph Library
 - <http://www.stingergraph.com/>
 - Georgia Tech
- Open MP
 - <http://www.openmp.org/>
 - Stony Brook
- GraphBLAS
 - <http://www.graphblas.org>
 - SEI/UMBC
- Kokkos C++ Programming EcoSystem
 - <https://github.com/kokkos>
 - Georgia Tech/Sandia



Software Roadmap

2018

2H '18

2019+



Cilk
C++
GMP Lib

Centos 7.3
Cilk Plus
Stinger Graph Lib

Open MP
GraphBLAS
Python Front End
Cilk/Tapir Parallel
Optimization

Machine Learning Lib
Lustre/GPFS
Python Libs on GCs
Cilk Race Detector
Cilk Profiler



Open Cilk

I-Ting Angelina Lee
Washington University in St. Louis

*Emu Tutorial @ IPDPS
May 22, 2018*



Open Cilk

I-Ting Angelina Lee
Washington University in St. Louis

*Emu Tutorial @ IPDPS
May 22, 2018*



What Is Cilk?

- **Cilk** extends C/C++ with a small set of linguistic control constructs to support **fork-join parallelism**.
- **Cilk** focuses on:
 - Shared-memory multiprocessing
 - Client-side multiprogrammed environments
 - Regular and irregular computations
 - New applications and legacy codebases
 - Response-time-sensitive application programming
 - Predictable and composable performance



Features of Cilk

- A **processor-oblivious** programming model with simple, effective, and **composable** language constructs for expressing parallelism
- A **provably** and **practically efficient work-stealing** scheduler
- A rich suite of **productivity tools**:
 - **Cilksan**: Determinacy race detector
 - **Cilkscale**: Scalability analyzer
 - **Cilkprof**: Scalability profiler



Features of Cilk

- A **processor-oblivious** programming model with simple, effective, and **composable** language constructs for expressing parallelism
- A provably and practically efficient **work-stealing** scheduler
- A rich suite of **productivity tools**:
 - **Cilksan**: Determinacy race detector
 - **Cilkscale**: Scalability analyzer
 - **Cilkprof**: Scalability profiler



Nested Parallelism in Cilk

```
uint64_t fib(uint64_t n) {  
    if (n < 2) {  
        return n;  
    } else {  
        uint64_t x, y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return (x + y);  
    }  
}
```

The named **child** function may execute in parallel with the **parent** caller.

Control cannot pass this point until all spawned children have returned.

Cilk keywords **grant permission** for parallel execution. They do not **command** parallel execution (**processor oblivious**).

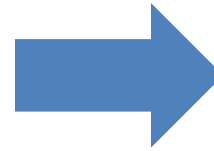
Loop Parallelism in Cilk

Example:

In-place
matrix
transpose

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

A

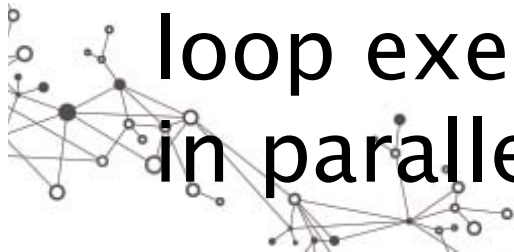


$$\begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix}$$

A^T

The
iterations of
a `cilk_for`
loop execute
in parallel.

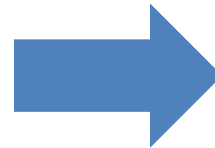
```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        int temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```



Serial Semantics

Cilk source

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```



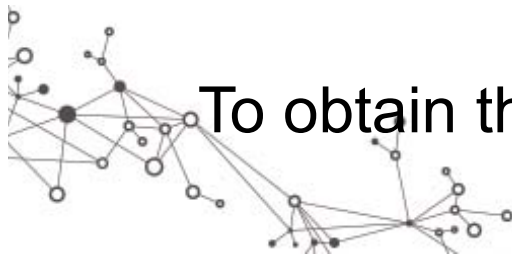
serialization

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = fib(n-1);
    y = fib(n-2);

    return (x + y);
  }
}
```

The **serialization** of a Cilk program is always a legal interpretation of the program's semantics.

Remember, Cilk keywords **grant permission** for parallel execution. They do not **command** parallel execution.



To obtain the serialization:

```
#define cilk_for for
#define cilk_spawn
#define cilk_syncz
```

Features of Cilk

- A **processor-oblivious** programming model with simple, effective, and composable language constructs for expressing parallelism
- A **provably** and **practically efficient work-stealing** scheduler
- A rich suite of **productivity tools**:
 - **Cilksan**: Determinacy race detector
 - **Cilkscale**: Scalability analyzer
 - **Cilkprof**: Scalability profiler

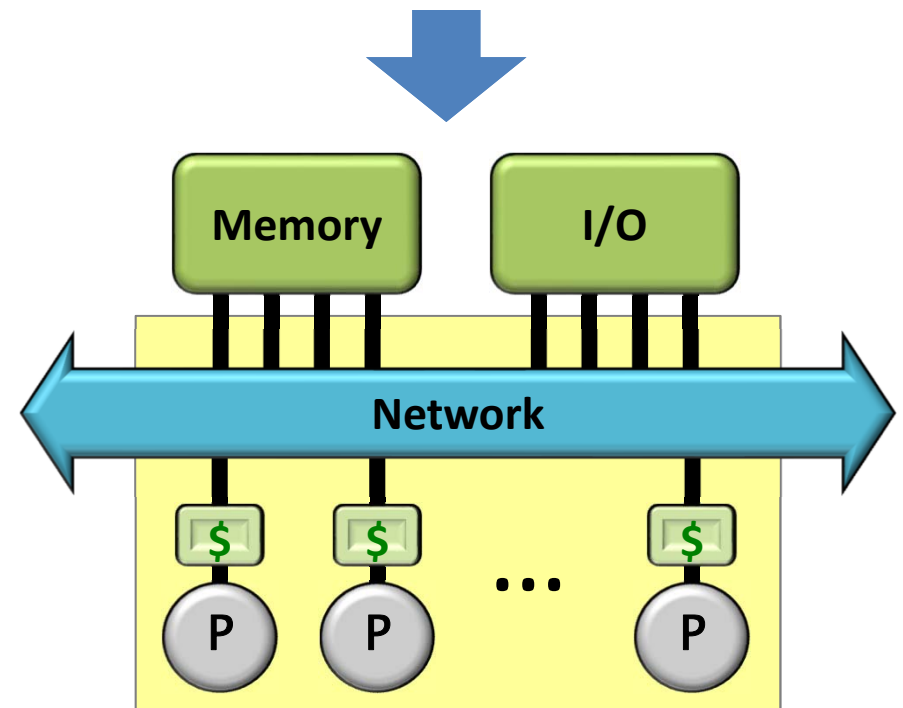


Scheduling in Cilk

- The Cilk concurrency platform allows the programmer to express **logical parallelism** in an application.
- The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.
- Cilk's **work-stealing scheduler** is **provably efficient**.



```
uint64_t fib(uint64_t n) {  
    if (n < 2) {  
        return n;  
    } else {  
        uint64_t x, y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return (x + y);  
    }  
}
```



Cilk Performance Bound

Definition. T_P — execution time on P processors

T_1 — **work** T_∞ — **span**

T_1 / T_∞ — **parallelism**

Theorem [BL94]. A work-stealing scheduler can achieve expected running time

$$T_P = T_1 / P + O(T_\infty)$$

on P processors.

In Practice. Cilk's scheduler achieves execution time

$$T_P \approx T_1 / P + T_\infty$$

on P processors.



Linear Speedup

Corollary. Cilk scheduler achieves near-perfect linear speedup whenever $T_1/T_\infty \gg P$.

Proof. Since $T_1/T_\infty \gg P$ is equivalent to $T_\infty \ll T_1/P$, Cilk's performance bound gives us

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\approx T_1/P. \quad (\text{first term dominates}) \end{aligned}$$

Thus, the speedup is $T_1/T_P \approx P$. ■



Features of Cilk

- A **processor-oblivious** programming model with simple, effective, and composable language constructs for expressing parallelism
- A provably and practically efficient **work-stealing** scheduler
- A rich suite of **productivity tools**:
 - **Cilksan**: Determinacy race detector
 - **Cilkscale**: Scalability analyzer
 - **Cilkprof**: Scalability profiler



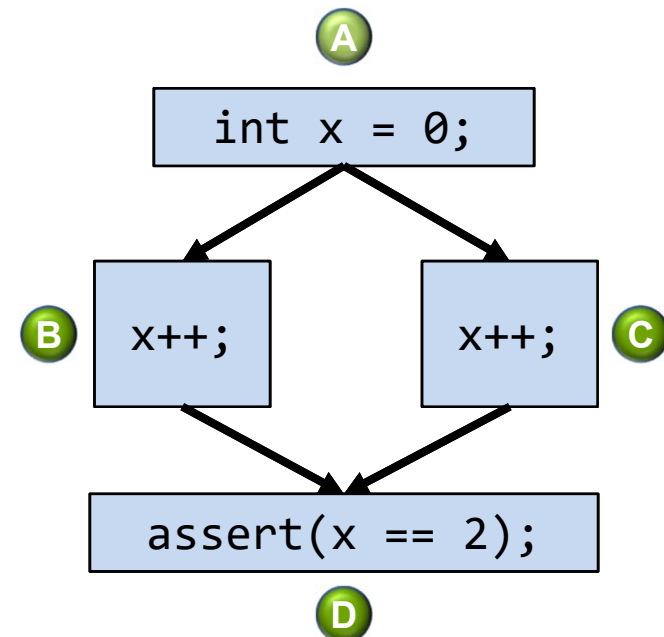
Determinacy Race

A **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

```

A int x = 0;
  cilk_for(int i=0, i<2, ++i) {
B   C     x++;
  }
D assert(x == 2);
  
```



computation DAG



Determinacy Race

A **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

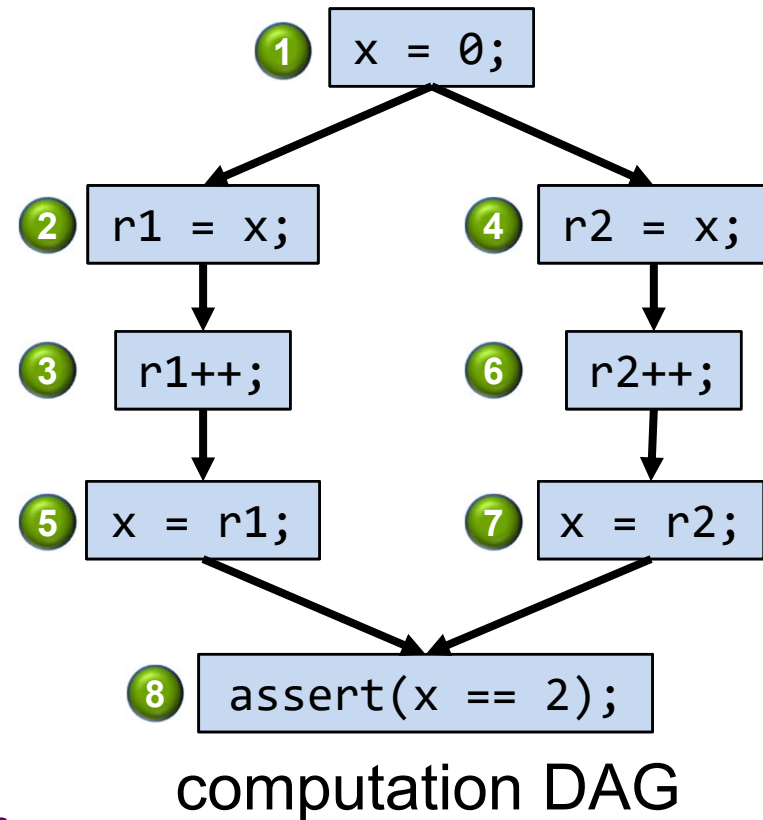
```

A int x = 0;
  cilk_for(int i=0, i<2, ++i) {
B   C   x++;
  }
D assert(x == 2);

```

x can be either 1 or 2.

This race can be fixed by declaring x to be a **cilk reducer**.



Cilk Platform

```
uint64_t fib(uint64_t n) {  
  if (n < 2) { return n; }  
  else {  
    uint64_t x, y;  
    x = cilk_spawn fib(n-1);  
    y = fib(n-2);  
    cilk_sync;  
    return (x + y);  
  }  
}
```

cilk code

Cilk Compiler

Runtime System

Linker

Binary

Program input



Parallel Performance

The compiler and runtime library together implement the scheduler.



Dev Flow: Serial Testing First

```
uint64_t fib(uint64_t n) {
  if (n < 2) { return n; }
  else {
    uint64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

cilk code

```
uint64_t fib(uint64_t n) {
  if (n < 2) { return n; }
  else {
    uint64_t x, y;
    x = fib(n-1);
    y = fib(n-2);
    return (x + y);
  }
}
```

serial elision

C/C++ Compiler

Binary

Serial regression tests

P

Reliable single-threaded code



Alternative Serial Testing

```
uint64_t fib(uint64_t n) {  
    if (n < 2) { return n; }  
    else {  
        uint64_t x, y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return (x + y);  
    }  
}
```

cilk code

Cilk Compiler

Binary

Serial
regression
tests

P

Reliable single-
threaded code

The parallel program executing on one core should behave **exactly the same** as the execution of the **serial elision**.

Cilk's serial semantics enable simple serial testing.

Parallel Testing

```
uint64_t fib(uint64_t n) {
    if (n < 2) { return n; }
    else {
        uint64_t x, y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x + y);
    }
}
```

cilk code

Cilk Compiler
with Cilksan

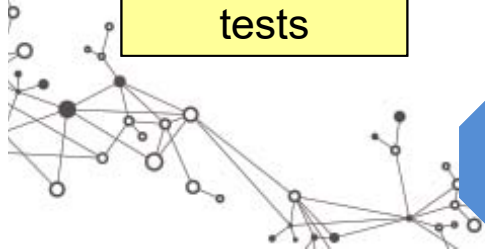
Binary

Parallel
regression
tests

P

Reliable single-
threaded code

Cilksan finds and
localizes race bugs.



Scalability Analysis

```
uint64_t fib(uint64_t n) {  
    if (n < 2) { return n; }  
    else {  
        uint64_t x, y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return (x + y);  
    }  
}
```

cilk code

Cilk Compiler
with Cilksan

Binary

Parallel
regression
tests

P

Reliable single-
threaded code

Cilkscale analyzes
how well your
program will scale
to larger machines.

Open Cilk

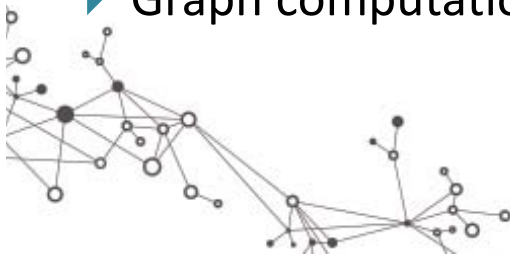
I-Ting Angelina Lee
Washington University in St. Louis

*Emu Tutorial @ IPDPS
May 22, 2018*



Sample Cilk Applications

- ▶ Artificial evolution
- ▶ Computer chess
- ▶ Connectomics
- ▶ Cryptographic hashing
- ▶ Data compression
- ▶ Dense and sparse linear algebra
- ▶ Discrete hedging for quantitative finance
- ▶ DNA sequence alignment
- ▶ Electromagnetism simulation
- ▶ Fast multipole method
- ▶ Friction-stir welding simulation
- ▶ Graph computations
- ▶ Graphics rendering — ray-tracing & radiosity
- ▶ Image analysis
- ▶ Lattice-Boltzmann methods
- ▶ Machine learning
- ▶ Model checking (Murphi)
- ▶ Sorting
- ▶ Symbolic computer algebra
- ▶ 3D solid modeling
- ▶ Video encoding/decoding
- ▶ Virus-shell assembly
- ▶ Wave and heat equations
- ▶ ...



Cilk Awards

Year	Org.	Award
1998	ICFP	Programming Contest — First Prize
2006	SC	HPC Challenge Class 2 (Productivity) — First Prize
2008	PLDI	Most Influential 1998 PLDI Paper Award
2009	SPAA	Best Paper Award
2012	SPAA	Best Paper Award
2013	ACM	Paris Kanellakis Theory and Practice Award
2017	PPoPP	Best Paper Award

The **Ligra** graph-processing library and other parallel software described in **Julian Shun**'s Ph.D. thesis, which won the 2015 ACM Doctoral Dissertation Award for best Ph.D. thesis in computer science, were all programmed in Cilk.



Impact of Cilk on Research

Google Scholar (2017-01-05)

- Cilk runtime system: 2094 citations
- Cilk language: 1326 citations
- Cilk scheduler: 1579 citations

Research papers that **meaningfully** rely on **Cilk** have appeared in the following professional venues:

3PGCIC, ACM-SE, ACTAE, AIMS, ALENEX, ASPLOS, BIG DATA, CC, CF, CGO, COMPSAC, CSS, DAC, DCC, DFM, DS-RT, ESA, ESEM, HPCC, HPCS, HPCSA, HiPC, I-SPAN, ICACT, ICCSE, ICDE, ICESS, ICPADS, ICPP, ICS, ICTAI, ICWC, IPDPS, ISCA, ISSAC, JACM, LLVM-HPC, MIPRO, OOPSLA/SPLASH, PACT, PASCO, PDP, PLDI, POPL, PPOPP, RTSS, SC, SIGCSE, SIGMETRICS, SIGOPS, SODA, SPAA, SoftCOM, TOCS, TOPC, TOPLAS, VLDB, VL/HCC, VPA, and WOSC.

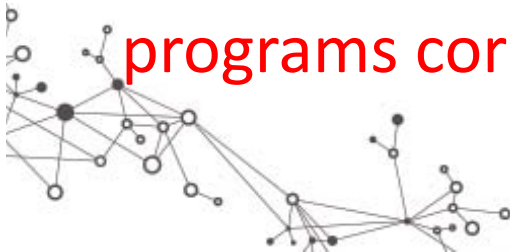


Impact of Cilk on Education

Cilk has been used in **numerous educational courses** across the world, including at the following universities:

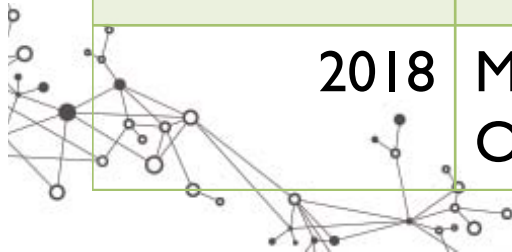
Alabama, ANU, Binghamton, CMU, Cornell, Duke, Fudan, George Washington, Georgetown, Georgia Tech, Harvard, Indiana, Johannes Kepler, Knox College, Lehigh, Maryland, Michigan, MIT, NTU, NUS, Oregon, Otago, Oxford, Princeton, Purdue, Rice, Rochester, Rutgers, Stanford, Stony Brook, TU Wien, Tel Aviv, Texas, UC Berkeley, UCSB, UNC, Washington, WUSTL, and Yale, and more.

A CMU study [CSMSA15] of teaching **Cilk** versus OpenMP documented that students found **Cilk easier to get programs correct** and their **Cilk** programs **ran faster**.



A Brief History of Cilk

1994–2006	Cilk project formed at MIT LCS. Cilk offers simple C-based multithreaded programming combined with execution efficiency.
2006–2009	Cilk Arts, Inc., spun out of MIT CSAIL. Cilk++ provides support for C++, parallel loops, and reducer hyperobjects.
2009–2014	Intel Corporation acquires Cilk Arts. Cilk Plus offers Cilk++ and vector ops in ICC and GCC.
2014–2017	Due to attrition in Intel's Cilk team, the development of Cilk Plus at Intel stagnates.
2017	Intel announces it is dropping support for Cilk Plus, and GCC follows suit.
2018	MIT forms Cilk Hub to support and develop a new Open Cilk platform.



Intel Cilk Plus vs Open Cilk

Component	Cilk Plus	Open Cilk
Language	Cilk++ & vector ops	Cilk++ → linguistic enhancements
Compiler	ICC*, GCC, (LLVM)	Based on Tapir/LLVM
Runtime	Cilk Plus	Cilk Plus → new
Instrumentation	Custom compiler & generic binary instrumentation	Generic compiler instrumentation based on CSI
Productivity tools	Cilkscreen race detector*, Cilkview scalability analyzer*	Cilksan race detector, Cilkscale scalability analyzer, Cilkprof scalability profiler

*Closed source software

Open Cilk R&D

- **Compiler front ends and back ends:** Generic parallelism support, e.g., Emu, Julia, OpenMP.
- **Reducers:** Simplified syntax, **compiler optimized**.
- **Random-number generation:** Faster deterministic parallel RNG based on pedigrees, **n-way independence**.
- **Pipeline parallelism:** Non-fork-join linguistic constructs, enhancements to race detector, **automatic throttling**.
- **Splitter hyperobjects:** **Cactus-stack-like semantics of shared memory**, e.g., for Boolean satisfiability.
- **Attached processors:** **linguistic integration, runtime**.
- **Processor affinity:** **Execution of loop iterations wherever data is likely to be from prior iterations**.
- **Parallel I/O:** Append semantics, latency hiding.
- **Tools:** Faster and smarter, e.g., parallelism profiling, **compressed shadow spaces, better debugging info**, etc.



Open Cilk Architecture

- **Compatibility:** **Open Cilk** will provide backward compatibility with Cilk Plus minus vector ops (i.e., Cilk++).
- **Open source:** The entire **Open Cilk** platform will be distributed under liberal open-source licenses.
- **Componentization:** The **Open Cilk** system will be divided into distinct software components with well-defined interfaces.
- **Integration:** As individual **Open Cilk** components are enhanced, they will continue to interoperate with the entire platform.
- **Reliability:** **Open Cilk** will feature an extensive suite of unit tests, regression tests, and benchmarks to ensure that releases are stable, perform well, and are free of serious bugs.



Cilk Hub

Cilk Hub is a new community-driven organization devoted to maintaining and enhancing **Open Cilk**.

Executive Director

- ▶ Prof. Charles E. Leiserson, MIT

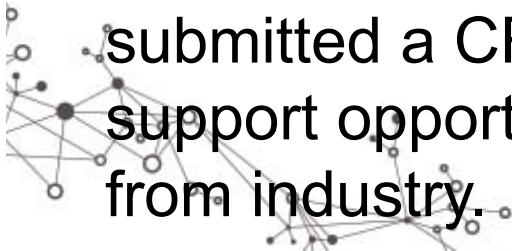
Director, Chief Architect, and Chair of Compiler Infrastructure

- ▶ Dr. Tao B. Schardl, MIT

Director and Chair of Runtime Support

- ▶ Prof. I-Ting Angelina Lee, WUSTL

Cilk Hub is operating under the auspices of MIT. We have submitted a CRI proposal to NSF. We wish to explore support opportunities from other government agencies and from industry.



Cilk Hub Advisory Board

Umut Acar, CMU
Vikram Adve, UIUC
David Bader, Georgia Tech
Pavan Balaji, ANL
Guy E. Blelloch, CMU
Aydın Buluç, LBNL
David Bunde, Knox College
Andrew Chien, Chicago
Rezaul Chowdhury, Stony Brook
Martin Deneroff, Emu Technologies
Chen Ding, Rochester
Alan Edelman, MIT
Jeremy Fineman, Georgetown
John Gilbert, UCSB
Phillip Gibbons, CMU

Shahin Kamali, Manitoba
Marc Moreno Maza, Western Ontario
John Mellor-Crummey, Rice
David Padua, UIUC
Keshav Pingali, UT, Austin
Nikos Pitsianis, Duke
Jan Prins, UNC
Lawrence Rauchwerger, Texas A&M
Vivek Sarkar, Georgia Tech
Nir Shavit, MIT
Julian Shun, MIT
Guy L. Steele Jr., Oracle Labs
Xiaobai Sun, Duke
Michael Bedford Taylor, Washington
Charles Tolle, SDSMT



Cilk Hub Mission

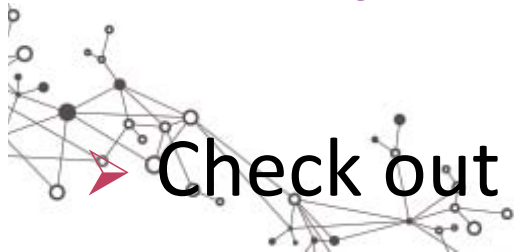
Cilk Hub provides quality open-source parallel-programming software and responsive support services for the benefit of application developers, parallel-language researchers, and teachers of parallel computing.

website: cilkhub.org



Features of Cilk

- A **processor-oblivious** programming model with simple, effective, and **composable** language constructs for expressing parallelism
- A **provably** and **practically efficient work-stealing** scheduler
- A rich suite of **productivity tools**:
 - **Cilksan**: Determinacy race detector
 - **Cilkscale**: Scalability analyzer
 - **Cilkprof**: Scalability profiler



➤ Check out Open Cilk @ Cilk Hub: cilkhub.org