Programming Overview

Janice McMahon

March 2021

Outline

- Emerging Applications
- Emu System Architecture
- Programming and Execution Model
- Software Development Environment

Emerging Applications

Evolution of Challenges Requires New Approaches to Solutions

LUCATA

Lucata Mission

A FUNDAMENTALLY NEW SOLUTION TO IDENTIFY RELATIONSHIPS WITHIN LARGE, UNSTRUCTURED DATASETS WITHOUT SACRIFICING PROGRAMMER PRODUCTIVITY.

- Large graph database problems
  - Distributed over many memories
  - Data movement dominates performance
  - Memory accesses are *irregular*, *remote*, and *unpredictable*
- Traditional system failure
  - Memory caches inefficient
  - Interconnect bandwidth insufficient
  - Power consumption unaffordable

Lucata context-flow architecture designed to meet the needs of today's large graph database 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

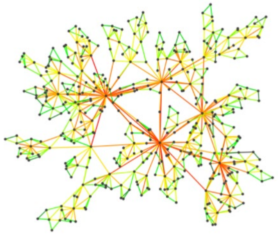
Benchmark Name	Function Performed	Conventional System Efficiency
LINPACK	Solve Ax=b, A is dense	**>90% of peak**
GUPS	Random updates	**~10% of peak**
HPCG: Hi Performance Conjugate Gradient	Ax=b, A sparse but regular	**~2% of peak**
SpMV: Sparse Matrix Vector	Ab; A sparse and irregular	**~2% of peak**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

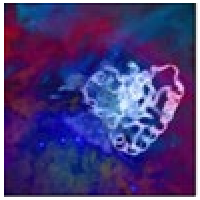
Lucata system is efficient for data intensive applications

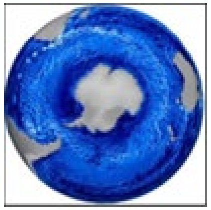
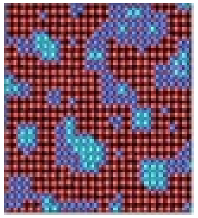
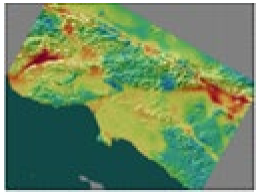
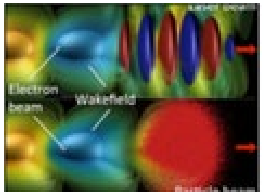
- Expect 20-90% of peak

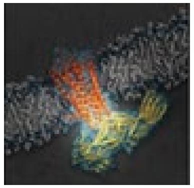
Markets and Applications

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







Lucata System Architecture

Built Around The Data

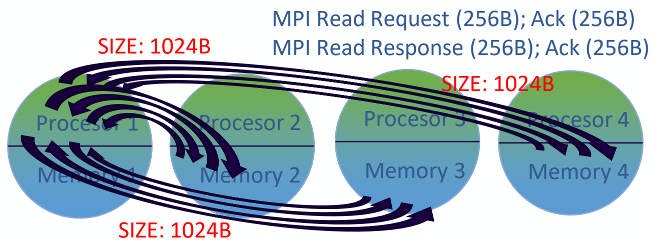
Lucata Innovation Overview

- Designed from the ground up to deal with applications that exhibit little locality
- Massive Shared Memory for in-Memory Computing
  - No I/O bottlenecks
- LUCATA 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

Lucata Architectural Comparison

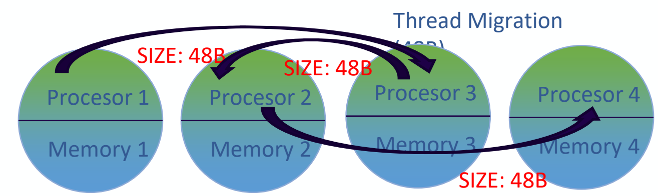
Context flow = program context moves to locale of accessed data via *thread migration*

Conventional computing with message passing



Network Bandwidth Consumed
Random Read Example
1 Local, 3 Remote Accesses
1024 Bytes/Remote Access per cache line
2736 Bytes Wasted

Context flow with thread migration



Network Bandwidth Consumed
Random Read Example
4 Local Memory Accesses + 3 Migrations
48 Bytes/Migration
144 Bytes Total

Order of magnitude reduction in amount of data moved
Wins big when data access pattern is a series of brief visits to widely dispersed data

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 node, 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

LUCATA

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.

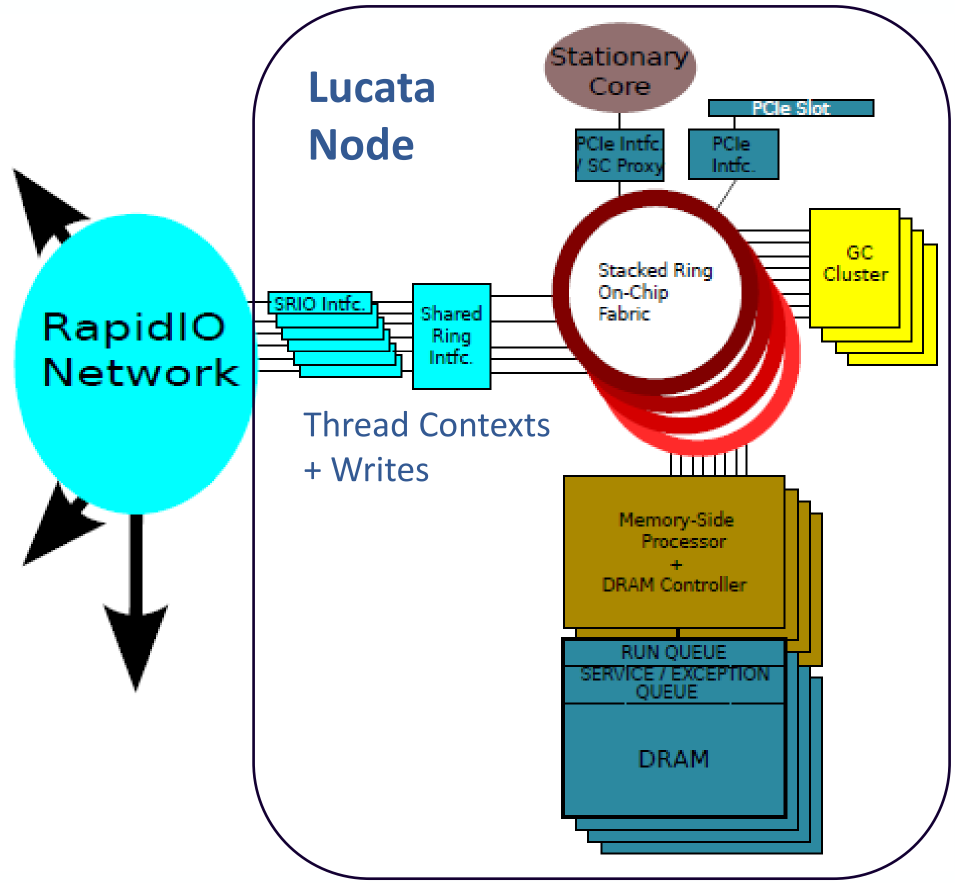
Lower energy — More concurrency — Greater scalability

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 interproccessor 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

Lucata System Architecture

- 8 Nodes per Chassis
- 8 Chassis per Rack
- RapidIO Network with multi-level switch
  - Contexts for migrating threads
  - Write packets for remote memory operations



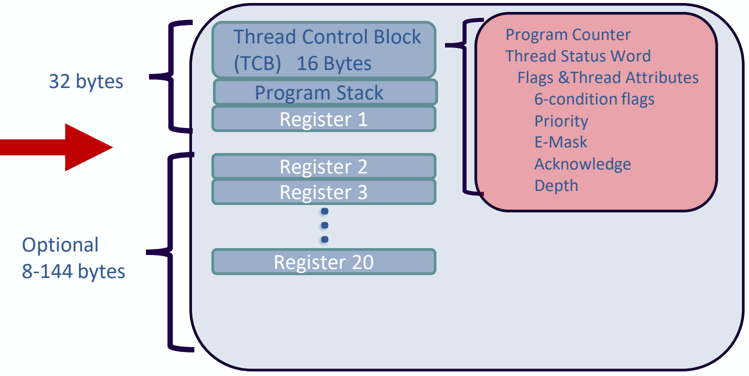
Node Architecture

- 24 Lucata Compute Elements (LCE)
- 4 Memory Side Processors (MSP)
- 64GB DRAM
  - 4 banks of 16GB dual-port DDR4
- Stacked Ring Fabric for on-chip communication
- 6 RapidIO 2.3 4-lane network ports
- Stationary Core (SC)
  - DualCore 64-bit Power E5500
  - 2GB DRAM
  - 1 TB SSD
  - PCIe Gen 3
  - Runs Linux

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

Thread context contained in Thread State Record (TSR)



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 sent over system interconnect to destination node
  - Arriving thread context is placed in Run Queue at destination node

System Level Spawn Control

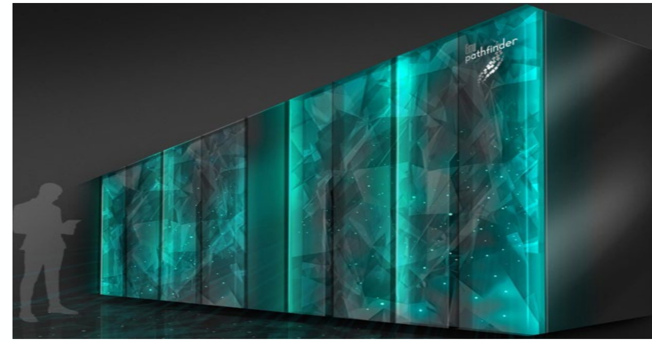
- Threads do not inherently know how many spawns other threads have executed
- Credit-based hardware/software scheme under development to
  - Limit the total number of threads to only what the system can handle
  - Handle hotspots where large numbers of threads converge on a single nodelet
  - Identify and avoid hotspots when possible

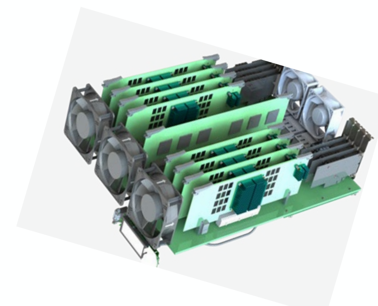
System Software

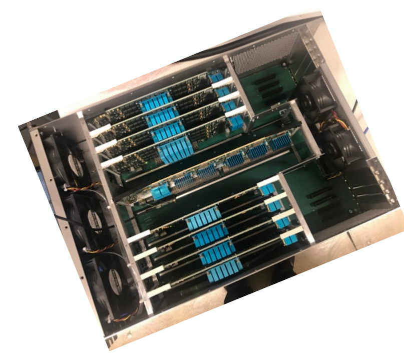
- LINUX runs on the Stationary Cores (SCs)
- OS launches main() user program on a Gossamer Core (GC)
  - main() then spawns descendants that execute in parallel and migrate throughout system as needed
- Runtime executes primarily on the SCs
  - Handles service requests from threads running on the GCs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

Lucata Platform

- ➢ Minimal latency
  - Data stays at node attached to
  - Migrating Threads
- ➢ "Strong-Scale" true linear growth
  - 8,192 nodes
  - no MPI overhead
  - eliminates clustering
- ➢ Large single image memory
  - 1,024TB
  - narrow channel memory access
- ➢ Massive abstracted parallelism
  - Millions of parallel threads
  - 80% Reduction of bandwidth usage
- ➢ Unparallel ingestion rates
  - 1.9 Tb/s/rack
  - greater than 7 TB/s Bisection rate
- ➢ Significantly green platform
  - 10-20x density
  - 60% less power consumption





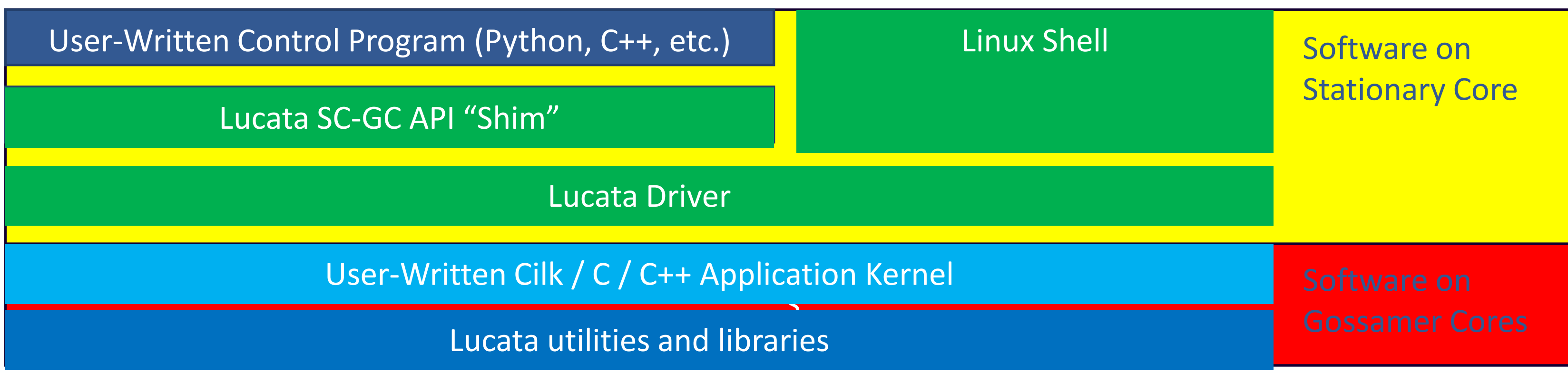


Programming and Execution Model

Flexible, easy-to-learn and use

Native Programming Model

- Program Lucata Engine in Cilk / C / C++
- Employ Lucata C/C++ utilities and standard libraries
- Control program from Python or other language running on SC
- Maximizes performance and flexibility



Software Constructs

- Dynamic parallelism using Cilk language
  - cilk_spawn: spawn a thread (locally or remotely)
  - cilk_for: distribute iterations of a loop among parallel threads
  - cilk_sync: synchronize all the threads spawned in a function
- Helper libraries / tools
  - memoryWeb: data allocation & distribution across Lucata Nodes
  - C/C++ utilities: optimized stripped arrays, chunked arrays (w/ spatial locality)
  - Timing hooks for profiling of code
- Intrinsic functions to access architecture-specific operations
  - Atomic memory operations for lock-free or fine grain locking algorithms
  - Remote memory operations without thread migrations
  - Special register access operations
- Replicated variables to avoid unnecessary migrations
  - Each node holds its own copy (e.g.: constants)

Programming Lucata

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

Lucata'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

Lucata Cilk

- Lucata 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

7/18/2021

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

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

Intrinsics

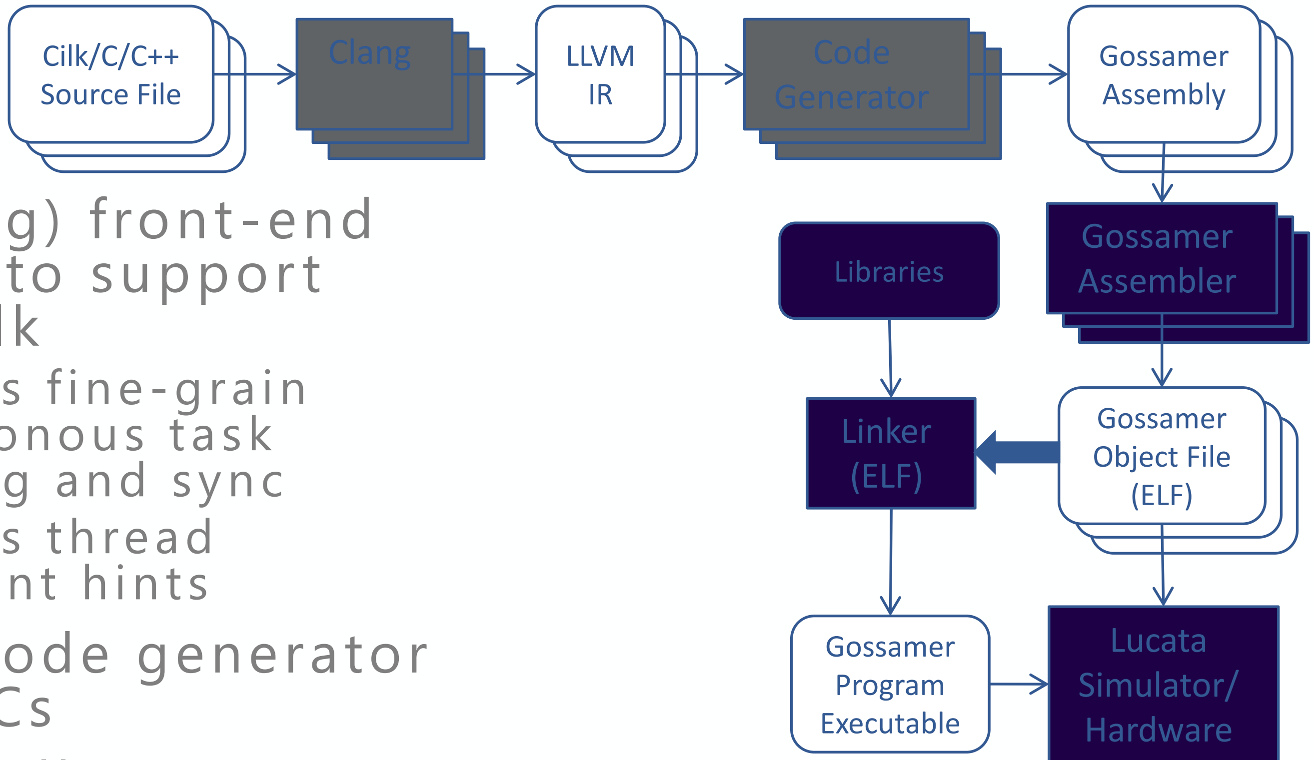
- Set of compiler recognized functions to access architecture specific operations
  - Atomic Arithmetic Operations
  - Remote Arithmetic Operations
  - Other Architecture Specific Operations
  - Thread Management Functions
  - System Queries

Replicated Data Structures

- ➢Replicating key shared data structures on each node can improve performance
  - Pointers to shared distributed data e.g. array
  - Copy at each node 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

Software Development Environment

Lucata Cilk Toolchain

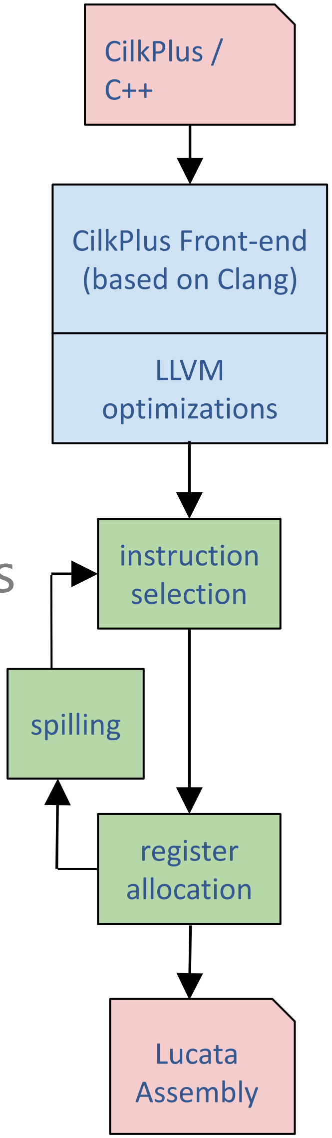


- ➢Cilk (clang) front-end modified to support Lucata Cilk
  - Supports fine-grain asynchronous task spawning and sync
  - Supports thread placement hints
- ➢Custom code generator for the GCs
- ➢Custom calling convention and run-time support
- ➢Custom assembler and linker

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

Custom Code Generator for Lucata

- 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



Lucata 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, typeinfo
    - 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.

Current Lucata Cilk Toolchain

- Updated Clang front-end
  - clang 6.0
  - Support for C++ 2017
- Integrates MIT's LLVM-Tapir framework
  - cilk.mit.edu
  - Tapir parallel optimization pass to enable optimization across parallel regions (threads)
  - Framework to support OpenCilk scalability analysis and profiling tools*
- Set of integer, commutative reducers using atomic updates: MIN, MAX, AND, ADD, OR, XOR

*Not yet integrated

Lucata 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
- Native x86 build capability
  - Useful for debugging

User Libraries

- GNU Multiple Precision Arithmetic (GMP) Library
  - Library for arbitrary precision arithmetic
  - Currently support integer GMP for Lucata
  - Included in current release
- Under development
  - GraphBLAS
  - RedisGraph
- Other research efforts
  - Streaming graph analysis
  - Kokkos C++ Ecosystem (Georgia Tech / Sandia)

Debug in x86 mode

- For GC: link with pre-installed library
  - –lemu_c_utils or for cmake: link_libraries(emu_c_utils)
- For x86: provides cross-compilation for Emu codes on x86
  - Use for rapid building and testing of codes before deployment to Emu architecture
  - Treats system as single nodelet with multiple Cilk threads
  - Requires Cilk support in x86 compiler
  - Use x86 library and include paths when building

Development Tools

- Cycle-accurate simulator, running on x86 servers
  - Timed or untimed to help initial debugging
  - Detailed information about memory usage, migrations, etc.
  - Helps develop the code
- Profiler
  - Parses simulator output files
  - Helps identifying hotspots and poor data distribution

Simulator

- Platform for
  - Toolchain verification
  - Application development and debugging
  - Architectural exploration
- Estimate application performance on Lucata system via cycle-approximate model
- Provide feedback to application writers to improve application performance
- Timed and non-timed modes of execution
  - Timed mode produces extra statistics and details about program execution
  - Enter timed mode via function call in program (can be overridden by simulator option)

Configuration and Summary Statistics

- Generated automatically in <program>.cdc
- Overview information about program execution
  - System configuration details (clock rates, bandwidths)
  - Wall clock time
- Extra information in timed mode
  - Total run time and cycle count (core, interconnect, and memory)
  - Total system thread counts (active, created, died)

Verbose Simulation Statistics

- Generated automatically in <program>.vsf
- Detailed thread statistics for each node
  - Migrations, threads created, threads died, spawn failures
  - For each MSP: number of reads, write, remote operations
- Extra information in timed mode
  - For each GC: packet, thread, and memory statistics, stall counts
  - For each MSP: read, write, remote, queue, and datacache statistics
  - For each SRIO port: requests, responses, performance counters
  - For each node: ring station counters

Memory Map

- Generated automatically in <program>.mps
- Memory map statistics in JSON format
  - Enqueue map shows when a thread is moved from GC to run or service exception queue
  - Remaining maps identify key memory operations (read, write, atomic, remote)
- Table of source-dest memory operations
  - Source and dest are nodes for the enqueue maps
  - Source and dest are MSPs for the remaining maps
- Threads moving into run queue of different node are migrating
- Threads moving into local run queue are rescheduled or spawned

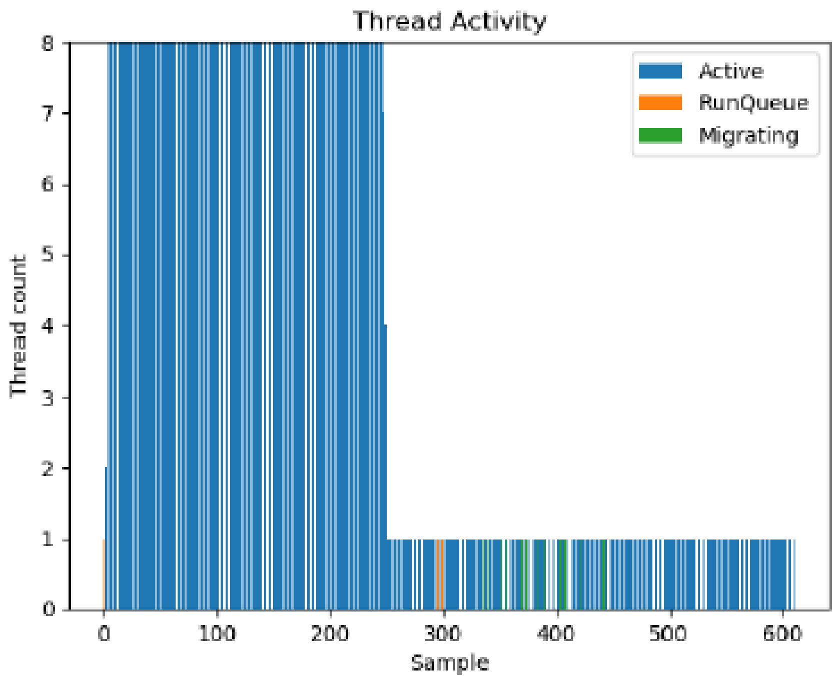
Optional Simulator Outputs

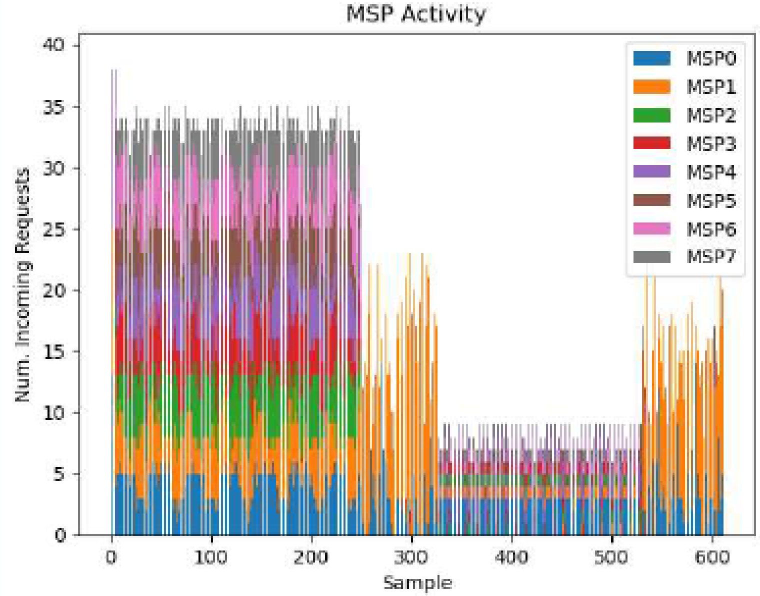
- Instruction counts for each function
  - Simulator option produces <program>.uis file
- Queue and resource statistics
  - Simulator option produces <program>.tqd file
- Verbose ISA trace for program execution
  - Simulator option to standard output
  - Options for whole program or specific thread
- Short trace of spawn, migrate, and quit thread operations
  - Simulator option to standard output
  - Timed and untimed modes

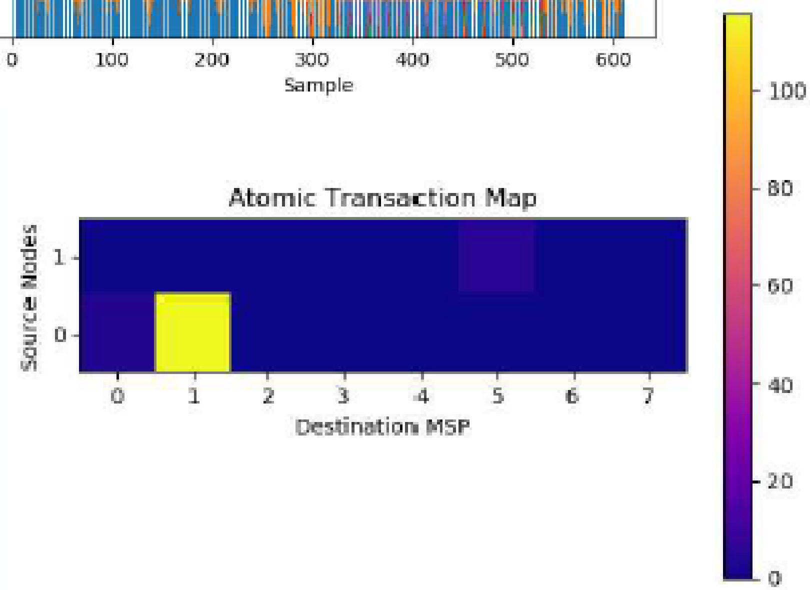
Profiler

- Runs a program in the Lucata simulator and automatically plot simulator statistics
  - From command line: emusim_profile
- Generates HTML files in a destination directory with images for different categories of results
  - Active threads
  - Total instructions by function
  - Memory map for local accesses
  - Remote map for remote atomic instructions
  - Percent of migrations by function
- Used to identify performance issues

Profiler Results



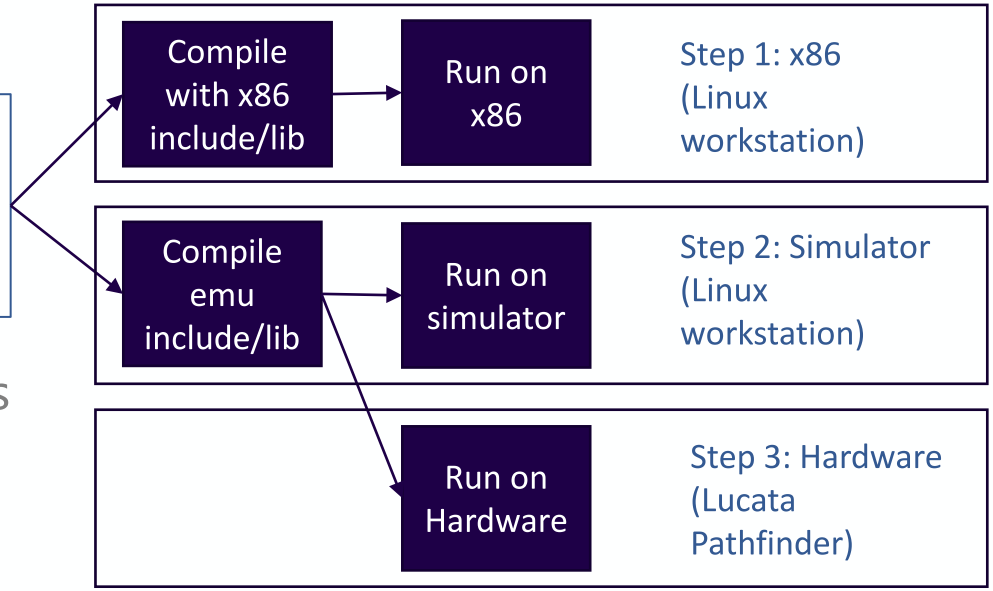




Software Development Workflow

Single program

```
#ifdef X86
#include "memoryweb_x86.h"
#else
#include "memoryweb.h"
#endif
// rest of C/C++ Cilk program
```



- Only difference is include file
- Program uses intrinsics, mw_malloc functions for distributed data
- X86 version mimics single node with single core

Development steps should be done in sequence
Lucata hardware platform used only to run final code
All tools run on Linux workstation

Workflow Step 1

Single program

```
#ifdef X86
#include "memoryweb_x86.h"
#else
#include "memoryweb.h"
#endif
// rest of C/C++ Cilk program
```

Compile with x86 include/lib → "exe" file → Run on x86

Step 1: x86

- Uses standard Linux compiler requiring Cilk support (GCC v5-v7)
- Compile with special paths (flags to compiler)
- Could use standard Linux tools (editor, debugger, profiler)
- Program is built and run the same manner as any C program

Example Linux commands:

```
> gcc -DX86 -I /usr/local/emu/x86/include main.c -o main
     -L /usr/local/emu/x86/lib -lemu_c_utils
> main
```

Produces executable file

Runs program on x86

Verify correct operation of parallel Lucata program

Workflow Step 2

Single program

```
#ifdef X86
#include "memoryweb_x86.h"
#else
#include "memoryweb.h"
#endif
// rest of C/C++ Cilk program
```

Compile emu include/lib

"mwx" file

Run on simulator

Step 2: Simulator

Example Linux commands:

```
> emu-cc -o main.mwx main.c -lemu_c_utils
> emusim.x -- main.mwx
> emusim_profile dir – main.mwx
```

Produces "mwx" executable file

Simulator mimics execution of mwx and produces output files with detailed statistics

Runs simulator and produces image files from statistics that can be viewed in a browser

- Use Lucata compiler, linker and simulator on Linux platform
- Simulator produces thread migration and memory usage statistics
- Used with profiler for visualization of those statistics

Verify correctness of Emu Cilk program
Architecture modeling and study
Understand parallel performance characteristics for *small data sets*

Workflow Step 3

Single program

```
#ifdef X86
#include "memoryweb_x86.h"
#else
#include "memoryweb.h"
#endif
// rest of C/C++ Cilk program
```

- Use Lucata compiler and linker on Linux platform
- Executable must be copied to Lucata machine over LAN and run on that machine (i.e., cross-compiled)
- Execution time can be measured but no statistics gathered

Example Linux commands:

```
> emu-cc -o main.mwx main.c -lemu_c_utils
> scp main.mwx LUCATA:
> ssh LUCATA
LUCATA> emu_multimode_exec main.mwx
```

Produces "mwx" executable file

Copy mwx to Lucata machine over network

Run command executed on LUCATA machine

Run and measure program on Lucata machine

Summary

- Lucata platform is a revolutionary new architecture to address the growing needs of large-scale database applications
- Lucata Pathfinder scales to thousands of cores and petabytes of memory with greater power efficiency and programmer productivity than today's architectures
- Lucata offers a comprehensive and growing set of software libraries and development tools that enable programming with the migratory thread model