Preparing HPC Education for a Post-Moore Future

Vivek Sarkar
Professor, School of Computer Science
Stephen Fleming Chair for Telecommunications, College of Computing
Georgia Institute of Technology

Algorithm
Language
API
Architecture
Instruction-set architecture
Microarchitecture
Function unit
Logic
Device
Algorithm
Language
API
Architecture
Instruction-set architecture
Microarchitecture
Function unit
Logic
Device

No disruption
More “Moore” changes
Hidden changes
Architectural changes
Non-von Neumann computing

Memory
Migrate on read
Gossamer CPU

Memory
Gossamer CPU

Host
Memory (Cryo-CMOS)
Control Processor
Quantum Substrate

Low density metal interconnects
300 K
77 K
4 K
0.1 K
20 mK
High Density Superconducting Control Wires

Prepared by Vivek Sarkar, with content from the presentation by Vivek Sarkar at the Georgia Institute of Technology.
Acknowledgments: Habanero Extreme Scale Software Laboratory

- Faculty
  - Vivek Sarkar
- Research Scientists
  - Max Grossman, Akihiro Hayashi, Jun Shirako, Jisheng Zhao
- Postdoctoral Researcher
  - Sriraj Paul

- Graduate Students
  - Seonmyeong Bak, Prithayan Barua, Prasanth Chatarasi, Sana Damani, Alind Khare, Youngsuk Kim, Ankush Mandal, Caleb Voss, Matthew Whitlock, Fangke Ye, Lechen Yu, Tong Zhou

- Acknowledgments
  - Research supported in part by DARPA, DoD, DOE, NSF
Outline

- Experiences with teaching Parallel, Concurrent, and Distributed Programming at undergraduate & continuing-education levels

- Future Computing Trends and the need for new pedagogies
Focus of our research is on software enablement on a wide range of extreme scale hardware.

**Structured-parallel execution model**

1. Lightweight asynchronous tasks and data transfers
   - Creation: async tasks, future tasks, data-driven tasks
   - Termination: finish, future get, await
   - Data Transfers: asyncPut, asyncGet

2. Locality control for task and data distribution
   - Computation and Data Distributions: hierarchical places, global name space

3. Inter-task synchronization operations
   - Mutual exclusion: isolated, actors
   - Collective and point-to-point operations: phasers, accumulators

**Languages**: X10, DFGL, OpenMP++
**Libraries**: Habanero-C/C++, Habanero-Java, Habanero-Scala

**Compiler Extensions for explicit parallelism**: Polyhedral+AST optimizers for C/C++ programs (src-to-src, LLVM), JVM bytecode optimizers

**Runtime Extensions for asynchronous tasks, data transfers, heterogeneity, resilience**: Habanero-C++ library, Open Community Runtime (OCR), HJlib cooperative runtime, datarace detectors

http://habanero.cc.gatech.edu
Target Platforms

Habanero execution model has been mapped to a wide range of platforms

- **Current/past systems**
  - Multicore SMPs (IBM, Intel)
  - Discrete GPUs (AMD, NVIDIA), Integrated GPUs (AMD, Intel)
  - FPGAs
  - HPC Clusters, Hadoop/Spark Clusters
  - Embedded processors: TI Keystone DSP

- **More forward-looking systems**
  - Members of “Rogues Gallery” in Georgia Tech’s Center for Research into Novel Computing Hierarchies (CRNCH): Emu Chick, 3D Stacked Memories w/ FPGAs, Neuromorphic Hardware, …
Habanero Execution Model underlies all our software research

1) Lightweight asynchronous tasks and data transfers
   - Creation: async tasks, future tasks, data-driven tasks
   - Termination: finish, future get, await
   - Data Transfers: asyncPut, asyncGet

2) Locality control for control and data distribution
   - Computation and Data Distributions: hierarchical places, global name space

3) Inter-task synchronization operations
   - Mutual exclusion: global/object-based isolation, actors
   - Collective and point-to-point operations: phasers, accumulators

Claim: these execution model primitives enable programmability, portability, performance, and verification for extreme scale software and hardware
Semantic Guarantees in Habanero Parallel Programs

Properties of interest
- DLF = DeadLock-Freedom
- DRF = Data-Race-Freedom
- DET = Determinacy
  (Functional & Structural determinism)
- DRF $\Rightarrow$ DET = DRF implies DET
- DRF $\Rightarrow$ DLF = DRF implies DLF
- SER = Serial elision

Some classes of parallel programs
- Class 1: \{async, finish, future\}
- Class 2 = Class 1 + \{barriers, phasers\}
- Class 3 = Class 2 + \{async-await\}
- Class 4 = Class 3 + \{isolated, actors\}

Properties of Class 1 = \{async, finish, future\}

- DLF: Deadlock freedom guaranteed for async, finish
- DRF $\Rightarrow$ DLF: Deadlock freedom guaranteed with futures if there are no data races on future objects
- DRF $\Rightarrow$ DET: Structural and Functional Determinism guaranteed for all Class 1 programs that are data-race-free
- SER: Serial elision property guaranteed for all Class 1 programs

- Functional Determinism property
  - Same input $\Rightarrow$ all executions have same output
- Structural Determinism property
  - Same input $\Rightarrow$ all executions have same computation graph
- Serial elision property
  - Removal of parallel constructs results in a sequential program that is one possible correct execution of original parallel program


Pedagogy using Habanero execution model

- Sophomore-level CS Course at Rice ([http://comp322.rice.edu](http://comp322.rice.edu))
  - “Simple things should be simple, complex things should be possible”
  - Introduce students to fundamentals of parallel, concurrent and distributed programming
  - Combine rigorous theoretical concepts with practical programming experiences
  - Use Habanero-Java library (HJlib) as pedagogic software
    - [https://wiki.rice.edu/confluence/display/PARPROG/HJ+Library](https://wiki.rice.edu/confluence/display/PARPROG/HJ+Library)

  - Use pedagogy from Rice COMP 322 course to target Java developers who have never before taken a course on parallel programming
  - Exclude some of the rigorous theoretical concepts due to lack of time in this format
  - Give priority to use of standard Java libraries over HJlib
    - Also created open source subset of HJlib at [https://github.com/habanero-rice/PCDP](https://github.com/habanero-rice/PCDP)
Example: Habanero abstraction of a CUDA kernel invocation in COMP 322

async at(GPU1)  forall(blockIdx)

async at(GPU2)  forallPhased(threadIdx)
1. Fundamentals of Parallelism

- creation and coordination of parallelism (async, finish), abstract performance metrics (work, critical paths), Amdahl's Law, weak vs. strong scaling, data races and determinism, data race avoidance (immutability, futures, accumulators, dataflow), deadlock avoidance, abstract vs. real performance (granularity, scalability), collective & point-to-point synchronization (phasers, barriers), parallel algorithms
COMP 322’s learning outcomes: Module 2

2. Fundamentals of Concurrency:
   - critical sections, atomicity, isolation, high level data races, nondeterminism, linearizability, liveness/progress guarantees, actors, request-response parallelism, Java Concurrency, locks, condition variables, semaphores, memory consistency models
COMP 322’s learning outcomes: Module 3

3. Fundamentals of Distributed-Memory Parallelism:
   - memory hierarchies, locality, cache affinity, data movement, distributed MapReduce, message-passing (MPI), combining multithreading with distributed computing
COMP 322 Learning Activities

- Review module handouts and video lectures before each class
- Complete in-class worksheets collaboratively (“blended” class format includes time for lecture & worksheets)
- Complete weekly lab exercises
- Complete weekly online quizzes
- 5 individual homeworks per semester with written and programming assignments
- 1 written midterm exam
- 1 written final exam
Weekly Lab Exercises (Java based)

Module 1
- Async-Finish task parallelism
- Futures
- Threshold/cutoff strategies for optimizing task granularity
- Java’s ForkJoin Framework
- Data/event-driven tasks
- Loop-level Parallelism with barriers

Module 2
- Isolated statements and atomic variables
- Actors
- Threads and Locks

Module 3
- Message Passing Interface (MPI)
- Apache Spark
Advanced
- Speculative parallelism
Programming Assignments (with auto-grader support)

1. Parallel Sort with abstract metrics (ideal parallelism with zero task overhead)
2. Parallel Matrix Multiplication with abstract metrics (but with nonzero task overhead)
3. Parallel Smith-Waterman algorithm on 8+ cores (using university cluster)
4. Parallelization of Boruvka’s Minimum Spanning Tree algorithm on 8+ cores (using university cluster)
5. Parallelization of Pi Computation using MPI with 8+ processes (using university cluster)
Coursera Specialization on “Parallel, Concurrent, and Distributed Programming in Java” was influenced by COMP 322

Top 10 countries in enrollment

- United States
- India
- Russian Federation
- Canada
- United Kingdom
- Germany
- China
- Poland
- Ukraine
- Netherlands

https://www.coursera.org/specializations/pcdp
Overview of 3-course specialization

- Split into three courses on parallelism, concurrency, and distribution
- Each course divided into four weeks (units). Each week includes:
  - 5 lecture videos (~5 minutes each) with a short reading to accompany each video
  - A graded multiple choice quiz for the week
  - A demonstration video, showing application of concepts from the week in implementing a parallel program
  - A graded mini-project exercising concepts from the week, similar to what was shown in demonstration video
    - Graded using correctness and performance tests on Coursera’s auto-grading cloud
- Instructor-learner interaction occurs primarily through per-course forums
- For details see EduHPC’18 paper on “A One Year Retrospective on a MOOC in Parallel, Concurrent, and Distributed Programming in Java”
Key challenge in online courses: how to motivate students to get started on content!

- Introductory Module: No content, test mini-project.
- Week 1 Module
- Week 2 Module
- Interview Video
- Week 3 Module
- Week 4 Module
- Wrap-Up
Key challenge in online courses: how to motivate students to get started on content!

- Massive drop in enrollment after content-free introductory module
- Requires more thought on exciting/engaging learners immediately
  - Offer concrete examples of impactful applications?
  - Point out value of skills in job market?
Outline

- Experiences with teaching Parallel, Concurrent, and Distributed Programming at undergraduate & continuing-education levels

- Future Computing Trends and the need for new pedagogies
Technology disruptions – from “killer micros” to the ”killer of micros”

- 1965: Moore’s Law published
- 1970s: Moore’s Law updated; Dennard’s geometric scaling rule
- 1980s: “Killer micros” overtake special-purpose HPC processors
- 1990s: Slowdown in CMOS wires: superscalar era begins
- 2005: Power Wall: Single thread exponential scaling ends (Intel Prescott, “the killer of micros”)
- 2010s: Homogeneity Wall: Heterogeneous accelerated systems with NVIDIA GPUs enter Top 500 list
- 2020s: Accelerator Wall: diminishing returns from accelerators
- 2030s: Post-Moore computing era begins …
Dennard scaling ended in 2005
Lack of code portability is already a significant issue, and it is becoming worse.
After years of effort, less than half of codes running on DOE supercomputers use GPUs and still fewer use them effectively.
Extreme Heterogeneity in computer architectures is relatively new, and not well understood by the HPC community.

Source: http://www.slideshare.net/top500/top500-list-november-2014?related=1
Mainstream Example: GPU-Accelerated Computing Instances in Amazon EC2 (https://aws.amazon.com/ec2/instance-types/)

P2 instances are intended for general-purpose GPU compute applications.

**Features:**

- High Frequency Intel Xeon E5-2686v4 (Broadwell) Processors
- High-performance NVIDIA K80 GPUs, each with 2,496 parallel processing cores and 12GiB of GPU memory
- Supports GPUDirect™ (peer-to-peer GPU communication)
- Provides Enhanced Networking using the Amazon EC2 Elastic Network Adaptor with up to 20Gbps of aggregate network bandwidth within a Placement Group

<table>
<thead>
<tr>
<th>Model</th>
<th>GPUs</th>
<th>vCPU</th>
<th>Mem (GiB)</th>
<th>GPU Memory (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2.xlarge</td>
<td>1</td>
<td>4</td>
<td>61</td>
<td>12</td>
</tr>
<tr>
<td>p2.8xlarge</td>
<td>8</td>
<td>32</td>
<td>488</td>
<td>96</td>
</tr>
<tr>
<td>p2.16xlarge</td>
<td>16</td>
<td>64</td>
<td>732</td>
<td>192</td>
</tr>
</tbody>
</table>

**Use Cases**

Machine learning, high performance databases, computational fluid dynamics, computational finance, seismic analysis, molecular modeling,
Mainstream Example: FPGA Computing Instances in Amazon EC2 (https://aws.amazon.com/ec2/instance-types/)

DEVELOP
Develop custom Amazon FPGA Images (AFI) using the Hardware Development Kit (HDK) and full set of design tools and simulators.

DEPLOY
Deploy your AFI directly on F1 instances and take advantage of all the scalability, agility, and security benefits of EC2.

OFFER
Offer AFIs you design on the AWS Marketplace for other customers.

PURCHASE
Purchase AFIs built and listed on AWS Marketplace to quickly implement common hardware accelerations.

Source: https://aws.amazon.com/ec2/instance-types/f1
Growth in Node-level programming models for future HPC

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>HIP</th>
<th>Kokkos</th>
<th>OpenACC</th>
<th>OpenMP</th>
<th>RAJA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Languages</strong></td>
<td>C/C++</td>
<td>C/C++</td>
<td>C/C++</td>
<td>C/C++/Fortran</td>
<td>C/C++/Fortran</td>
<td>C/C++</td>
</tr>
<tr>
<td><strong>Parallelism</strong></td>
<td>SIMT</td>
<td>SIMT</td>
<td>OpenMP, Pthreads, CUDA</td>
<td>SIMD, CUDA, Fork-Join</td>
<td>SPMD, SIMD, Tasks, CUDA, Fork-Join</td>
<td>OpenMP, CUDA</td>
</tr>
<tr>
<td><strong>Licensing/Accessibility</strong></td>
<td>Proprietary</td>
<td>Open-source</td>
<td>Open-source</td>
<td>Open-source</td>
<td>Open-source</td>
<td>Open-source</td>
</tr>
<tr>
<td><strong>Abstraction Level</strong></td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>High/Medium</td>
<td>High/Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

- **C++ Library / Template**: Kokkos, RAJA
- **Perf. Abstract Layer**: OpenMP, OpenACC
- **Directive-Based Approaches**: C++ Library / Template, Perf. Abstract Layer
- **Granularity**: Parallelism, Memory
- **Programming Model**: C++ Library / Template, Perf. Abstract Layer
- **Service Layers**: C++ Library / Template, Perf. Abstract Layer

29
End of Moore’s Law

A slow tapering off --- feature sizes will continue to diminish until 1nm in 2033, with monolithic 3D transistors expected from 2024 onwards

<table>
<thead>
<tr>
<th>YEAR OF PRODUCTION</th>
<th>2017</th>
<th>2019</th>
<th>2021</th>
<th>2024</th>
<th>2027</th>
<th>2030</th>
<th>2033</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic industry &quot;Node Range&quot; Labeling (nm)</td>
<td>10</td>
<td>7.5</td>
<td>5</td>
<td>3</td>
<td>2.1</td>
<td>1.5</td>
<td>1.0</td>
</tr>
<tr>
<td>BM-Foundry node labeling</td>
<td>10-7</td>
<td>7-5</td>
<td>5-3</td>
<td>3-2.1</td>
<td>1.5-1</td>
<td>1</td>
<td>0.7</td>
</tr>
<tr>
<td>Logic device structure options</td>
<td>finFET</td>
<td>finFET</td>
<td>LGAA</td>
<td>LGAA</td>
<td>VGAA</td>
<td>VGAA</td>
<td>VGAA</td>
</tr>
<tr>
<td>Logic device mainstream device</td>
<td>finFET</td>
<td>finFET</td>
<td>LGAA</td>
<td>LGAA</td>
<td>VGAA</td>
<td>VGAA</td>
<td>VGAA</td>
</tr>
</tbody>
</table>

Table MM01 - More Moore - Logic Core Device Technology Roadmap

<table>
<thead>
<tr>
<th>Logic device technology naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern technology inflexion for Mx interconnect</td>
</tr>
<tr>
<td>Channel material technology inflexion</td>
</tr>
<tr>
<td>Process technology inflexion</td>
</tr>
<tr>
<td>Stack generation</td>
</tr>
<tr>
<td>Design-technology scaling factor for standard cell</td>
</tr>
<tr>
<td>Design-technology scaling factor for SRAM (1T1) bitcell</td>
</tr>
<tr>
<td>Number of stacked devices in one tier</td>
</tr>
<tr>
<td>Tier stacking scaling factor for SoC</td>
</tr>
<tr>
<td>Vdd (V)</td>
</tr>
<tr>
<td>Physical gate length for HP Logic (nm)</td>
</tr>
<tr>
<td>SoC footprint scaling node-to-node - 50% digital, 35% SRAM, 15% analog+IO</td>
</tr>
</tbody>
</table>

Source: IEEE IRDS 2017 Edition
Levels of Disruption in Moore’s Law End-Game and Post-Moore eras

At the far right (level 4) are non–von Neumann architectures, which completely disrupt all stack levels, from device to algorithm.

At the least disruptive end (level 1) are more “Moore” approaches, such as new transistor technology and 3D circuits, which affect only the device and logic levels.

All future hardware directions are characterized by extreme heterogeneity.

Disruption in Mainstream Computing Stack has begun at both ends …

Examples:
- Deep learning algorithms
- Go, Julia, Halide, Python
- Caffe, Spark, TensorFlow

… and guess where they meet in the middle!

Accelerators, Reconfigurable Logic
Memory-centric Computing
3D devices
Disruptive Example: Emu Chick

- “Migratory Memory Side Processing” to exploit locality.
- Data for graph edge attributes, documents, etc. reside nearby even if accessed irregularly.
- Moving threads to data on reads means all accesses are local, common case needs to tolerate less latency.

// Habanero abstraction of // remote read of location A at (HOME(A)) { 
  continuation after reading A, 
  
  }
Kuhn’s History of Science


Revolution: A new paradigm emerges

Old Theory: well established, many followers, many anomalies, attempts to address anomalies by “puzzle solving activities” governed by the rules of the paradigm

New Theory: few followers, untested, new concepts/techniques, accounts for anomalies, asks new questions

Figure source: www.philosophy.ed.ac.uk/ug_study/ug_phil_sci1h/phil_sci_files/L10_Kuhn1.ppt
Classical parallel/concurrent primitives were “good enough” for enabling applications on past systems …

- Threads
- Locks
- Barriers
- Message-passing

... but are inadequate for the extreme levels of parallelism, heterogeneity and distribution needed in future systems

- Performance limitations
  - Future systems need execution models with pervasive asynchrony and heterogeneity across multiple levels of distributed compute and memory modules, *with no blocking/waiting operations*

- Correctness limitations
  - Future applications need structured-parallel constructs with compositional invariants

→ Explore combination of dataflow execution and Habanero asynchronous task models as foundation for new pedagogies!
The first opinion on dataflow: Jack Dennis

Proposed building general-purpose parallel machines based on a dataflow graph model of computation

Inspired all the major players in dataflow during the early years (1970s – 1980s)

Source: “Dataflow: Passing the Token”, talk given by Prof. Arvind at ISCA 2005
Dataflow Graphs

\[
\begin{align*}
\{ & x = a + b; \\
& y = b \times 7 \\
\text{in} & (x-y) \times (x+y) \}
\end{align*}
\]

Values in dataflow graphs are represented as tokens

\[
\text{token} \; \langle \text{ip, p, v} \rangle
\]

An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators

Source: “Dataflow: Passing the Token”, talk given by Prof. Arvind at ISCA 2005
A small set of dataflow operators can be used to define a general programming language.

Source: “Dataflow: Passing the Token”, talk given by Prof. Arvind at ISCA 2005
A Second Opinion and the start of a “Dataflow Winter”

- Published in IEEE Computer 1982
- Identified limitations of the data flow approach, especially with respect to the overheads of distributed control and lack of locality management.
A Third Opinion -- Dataflow Execution Models offer a promising foundation for software and hardware in future "extreme heterogeneity" platforms.

• Macro-dataflow = extension of dataflow model to task-level operations

• General idea: computation unfolds as a task graph
  • Node = non-preemptive task that can execute on some subset of “places”
  • Edge = event signal or communication of a single-assignment future/promise value

• Semantic guarantees: race-freedom, determinism
A Third Opinion -- Dataflow Execution Models for software and hardware in future "extreme" parallelism

- Macro-dataflow = extension of dataflow
- General idea: computation unfolds as a task graph
  - Node = non-preemptive task that can execute
  - Edge = event signal or communication of a single node
  - Semantic guarantees: race-freedom, determinism

In this paper, we present a compile-time partitioning algorithm to partition program graphs into subgraphs that can execute in parallel. This partitioning provides a more general granularity to efficiently implement parallel evaluation models on multiprocessors. For convenience, we define a macro-actor to be a dynamic invocation of a (static) subgraph. A macro-actor's inputs and outputs are determined by the corresponding inter-subgraph input and output edges. Our compile-time partitioning algorithm is driven by costs for execution times and communication sizes. We introduce a simple analytical model and derive an objective function, \( F(T) \), that defines the cost of partition \( T \). The partitioning algorithm attempts to build a partition with the smallest value of \( F(T) \).

The dataflow model is traditionally defined as the granularity of instructions or dataflow operators. With our compile-time partitioning, a macro-dataflow model can be defined at the granularity of macro-actors: each macro-actor executes sequentially, but there is parallelism among macro-actors.

A fundamental design decision in our approach is that a macro-actor is able to run to completion once all its inputs are available. This allows for non-preemptive on-chip scheduling with no task-switching overhead. Cyclic dependencies are thus forbidden among macro-actors. This restriction is called the continuity constraint and is discussed in detail in Section 6.

A compile-time partitioner has been implemented to process program graphs in the intermediate language, \( IFI \) [1]. IFI represents computation as dataflow graphs, as described in Section 5. A list of target parameters (e.g. number of processors, communication and scheduling overhead) drive the partitioning for a given multiprocessor architecture. Using a front-end (from SISAL [11] to IFI), we apply this system to programs written in the single-assignment language SISAL. However, our approach is applicable to any

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
Why now for Dataflow Execution Models?

- Characteristics of extreme heterogeneity platforms
  - Increasing number of heterogeneous processors and custom accelerator
  - Increasing heterogeneity in memory systems
  - Near-memory/in-memory computation structures
  - Inclusion of non von Neumann computing elements in some cases

- Programmability of extreme heterogeneity platforms
  - Significant challenges relative to multicore parallelization, which itself was viewed as a hard problem
  - Mapping of data across discrete memories in heterogeneous devices adds a new dimension of complexity
  - As does synchronization among kernels executing on heterogeneous devices
Increasing recent adoption of Dataflow Execution Models

- Machine Learning frameworks, e.g., TensorFlow
- OpenMP “depends” clause (similar idea in OmpSS, Legion, …)
- TBB’s flow graph interface
- Dag parallelism in numerical libraries (PYRROS, PLASMA, SLATE, …)
- National Instruments’ LabView product
- Event-driven programming in JavaScript, reactive programming systems
- Futures and promises in modern languages (C++, Java, JavaScript, …)
- HPVM for heterogeneous systems
- Work done in Habanero project
  - Async-await primitive for homogeneous/heterogeneous/distributed parallelism
  - Open Community Runtime (OCR)
  - Concurrent Collections (CnC), and Data Flow Graph Language (DFGL)
- …
Multi-Node Smith-Waterman using Data-Driven Tasks

1. `#define DDF_HOME(guid) . . .`
2. . . .
3. for (i=0;i<H;++i)
4.   for (j=0;j<W;++j)
5.     matrix[i][j] = DDF_HANDLE(i*H+j);
6. . . .
7. `finish` { // matrix is a 2-D array of DDFs
8.   for (i=0,i<H;++i) {
9.     for (j=0,j<W;++j) {
10.    DDF_t* curr = matrix[i][j];
11.    DDF_t* above = matrix[i-1][j];
12.    DDF_t* left = matrix[i][j-1];
13.    DDF_t* uLeft = matrix[i-1][j-1];
14.    async AWAIT (above, left, uLeft) {
15.       Elem* currElem = . . .
16.       DDF_PUT(curr, currElem);
17.    }/*async*/ }/*for-j*/ }/*for-i*/
18.}/*finish*/
Results for APGNS version of Smith-Waterman (communication runtime uses MPI under the covers)
Locality-aware Scheduling for CPUs using the Hierarchical Place Tree (HPT) abstraction

- Model target system as a tree/hierarchy
- Workers attached to leaf places
  - Bind to hardware core
- Each place has a queue
  - `async at(<p>) <stmt>`: push task onto place `p`'s queue
  - Destination place can be determined by programmer, compiler or runtime
- Current policy
  - A worker executes tasks from ancestor places from bottom-up
    - W0 executes tasks from PL3, PL1, PL0
  - Tasks in a place queue can be executed by all workers in its subtree
    - Task in PL2 can be executed by workers W2 or W3
Extending the HPT abstraction for heterogeneous architectures & accelerators

- Devices (GPU or FPGA) are represented as memory module places and agent workers
  - GPU memory configuration is fixed, while FPGA memory can be reconfigured at runtime
  - async at(P) S
    - Creates new task to execute statement S at place P

- Physical memory
- Cache
- GPU memory
- Reconfigurable FPGA
- Implicit data movement
- Explicit data movement
- CPU computation worker
- Device agent worker
Medical imaging applications (NSF Expeditions Center for Domain-Specific Computing)

- New reconstruction methods
  - decrease radiation exposure (CT)
  - number of samples (MR)
- 3D/4D image analysis pipeline
  - Denoising
  - Registration
  - Segmentation
- Analysis
  - Real-time quantitative cancer assessment applications
- Potential impact:
  - order-of-magnitude performance and energy efficiency improvements
  - real-time clinical applications and simulations using patient imaging data

Figure credit: NSF Expeditions CDSC project
<table>
<thead>
<tr>
<th>Domain-specific modeling</th>
<th>UCLA</th>
<th>Rice</th>
<th>UCSB</th>
<th>Ohio State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application drivers</td>
<td>Bui, Reinman, Potkonjak</td>
<td>Sarkar, Baraniuk</td>
<td>Cheng</td>
<td>Sadayappan</td>
</tr>
<tr>
<td>CHP creation</td>
<td>Chang, Cong, Reinman</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHP mapping</td>
<td>Cong, Palsberg, Potkonjak</td>
<td>Sarkar</td>
<td>Cheng</td>
<td>Sadayappan</td>
</tr>
<tr>
<td>Experimental systems</td>
<td>Aberle, Bui, Chien, Hsu, Vese</td>
<td>Baraniuk</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>All (led by Cong &amp; Bui)</td>
<td>All</td>
<td>All</td>
<td>All</td>
</tr>
</tbody>
</table>
Use of Dataflow Model for Heterogeneous Computing in CDSC for CPU+GPU+FPGA platform

- DFGL graph representation extended with affinity annotations:
  - $<C> :: (D_{CPU=20,GPU=10});$
  - $<C> :: (R_{GPU=5, FPGA=10});$
  - $<C> :: (S_{GPU=12});$
  - $[IN : k-1] \rightarrow (D : k) \rightarrow [IN2 : k+1];$
  - $[IN2 : 2^k] \rightarrow (R : k) \rightarrow [IN3 : k/2];$
  - $[IN3 : k] \rightarrow (S : k) \rightarrow [OUT : IN3[k]];$
  - $env \rightarrow [IN : \{0..9\}], <C : \{0..9\}>;$
  - $[OUT : 1] \rightarrow env;$

Mapping a Data-Flow Programming Model onto Heterogeneous Platforms.” Alina Sbirlea, Yi Zou, Zoran Budimlic, Jason Cong, Vivek Sarkar. LCTES 2012
Example Application-Driven Milestones in CDSC

Baseline (2010)
- 1.5 min (Regular CT)
- 18 hours (Low-dose CT)

CDSC implementation (2015)
- 6 min (Low-dose CT)

- 5 minutes
- 3 seconds
- 10 minutes
- 30 seconds
- 20 minutes
- 1 minute
- 45 minutes
- 5 minutes

~19 hours total
~12 minutes total
Summary

- Habanero dataflow-based execution model for enhanced performance and enhanced verifiability
- Presented experiences with use of Habanero Execution Model in teaching
  - COMP 322 undergraduate (sophomore-level) class at Rice University
  - 3-course specialization on Coursera
- Extreme heterogeneity anticipated in future computing platforms
- Conclusion: combination of dataflow execution and Habanero asynchronous task models can contribute to productivity, performance, and pedagogy for post-Moore computing!
BACKUP SLIDES START HERE
Sample in-class “acting-ivity”: What is the critical path length of this parallel computation?

1. `finish {  // F1`
2. `async A;  // Boil water & pasta (20)`
3. `finish {  // F2`
4. `async B1;  // Chop veggies (5)`
5. `async B2;  // Brown meat (10)`
6. `}  // F2`
7. `B3;  // Make pasta sauce (5)`
8. `}  // F1`

Another popular activity – acting out the Dining Philosophers Problem!
Some discussion topics for future HPC pedagogic foundations

- What should the pedagogic modules be?
  - Extend Parallelism, Concurrency, Distribution with
    - Heterogeneous computations?
    - Heterogeneous memories?
- What base language to use for programming projects?
  - Most popular mainstream languages today are JavaScript, Java, and Python
- Which level(s) should these concepts be introduced at?
  - Lower-level undergraduate, upper-level undergraduate, lower-level graduate, upper-level graduate, continuing education?
Publications related to semantic guarantees in Habanero execution model

1) Verification of Classes of Programs
   - Deadlock Avoidance with Futures [OOPSLA ’17, PPoPP ’19]
   - Formalization of Habanero Phasers using Coq [PLACES’16, JLAMP ‘17]
   - Integrating Actors with Task Parallelism [OOPSLA ’12, AGERE ’14]
   - Delegated Isolation for Nested Task Parallelism [OOPSLA ’11, OOPSLA ’13]
   - Deterministic reductions [WoDet ‘11, WoDet ’13]
   - Determinacy and Repeatability of Parallel Program Schemata [DFM ’12]

2) Verification of a given program (for all inputs)
   - Static Race Detection for SPMD Programs [LCPC ‘16]
   - Object-based Isolation [EuroPar ‘15]
   - Model Checking Task Parallel Programs using Gradual Permissions [ASE ’15]
   - Permissions for Race-Free Parallelism [RV ‘11, ECOOP ‘12]
   - Type inference for locality [PPoPP ’08]

3) Verification of a given program and input
   - Graph Traversal Based Data Race Detection [EuroPar ’18]
   - Dynamic Determinacy Race Detection for Futures [RV ‘16]
   - Test-Driven Repair of Data Races [PLDI ‘14]
   - Determinism checking [SAS ‘10, WoDet ’14]
   - Dynamic Data Race Detection for Structured Task Parallelism [PLDI ‘12]
Publications related to performance benefits of Habanero execution model

1) Parallelism
- Automatic parallelization of Futures [OOPSLA’16]
- Automatic GPU parallelization of Hadoop & Spark [HPDC’16]
- Automatic GPU parallelization of Java [PACT’15]
- Optimization of Structured Parallelism [TOPLAS’13, PACT’15]
- Polyhedral optimization of Dataflow Programs [LCPC’15]
- Heterogeneous work-stealing for CPU+DSP [HPEC’15]
- Bounded-Memory Scheduling [PACT’14]
- Cooperative Scheduling of General Parallel Constructs [ECOOP’14]
- Optimized Doacross for OpenMP [EuroPar’12, IWOMP’13]
- Heterogeneous work-stealing for CPU+GPU+FPGA [LCTES’12]
- GPU work-stealing [LCPC’11]

2) Locality
- Data layout optimizations for CPUs & GPUs [HeteroPar’13, CC’16]
- Data-driven Tasks [ICPP’11]
- SLAW: Scalable Locality-aware Work-stealing [IPDPS’10]
- Hierarchical Phasers [IPDPS’10]
- Hierarchical Place Tree [LCPC’09]

3) Distribution
- Polyhedral optimizations for distribution [SC’16]
- Integrating async tasks with OpenSHMEM [OpenSHMEM’16]
- Distributed work-stealing [IA^3 ’16]
- Distributed actors [PPPJ’16, ManLang’17]
- Integrating async tasks with UPC++ [PGAS’14]
- Integrating async tasks with MPI [IPDPS’13]
- Compiler optimizations for Distributed Memory [IPDPS’14]