Georgia Tech School of Computer Science

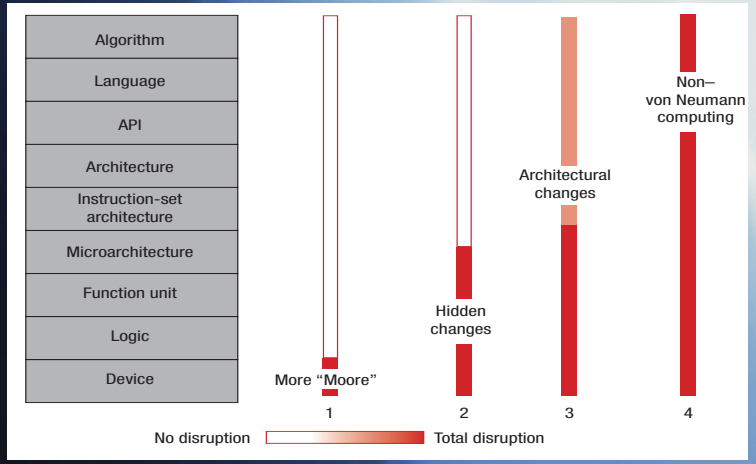
Preparing HPC Education for a Post-Moore Future

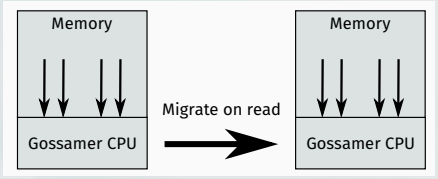
Vivek Sarkar

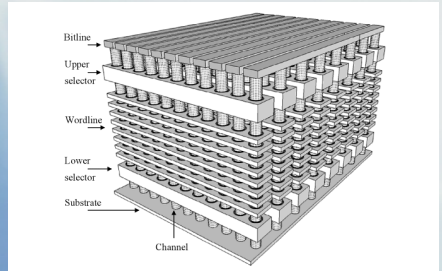
Professor, School of Computer Science

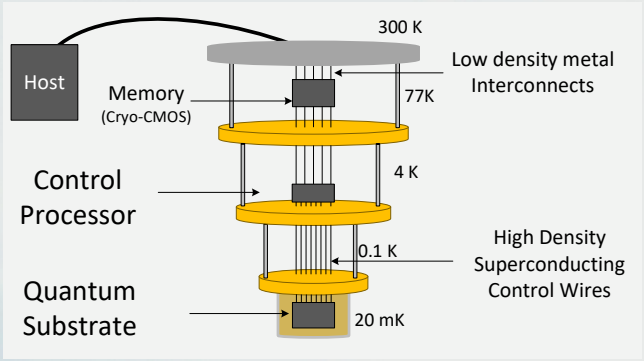
Stephen Fleming Chair for Telecommunications, College of Computing

Georgia Institute of Technology









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

Habanero group photo, Rice University Graduation, May 2015

Georgia Tech PL/SE outing, October 2017

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

Parallel Applications

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*

Habanero Programming Models

Habanero Compilers & Parallel IRs

Habanero Runtime Systems

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

Extreme Scale Platforms

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

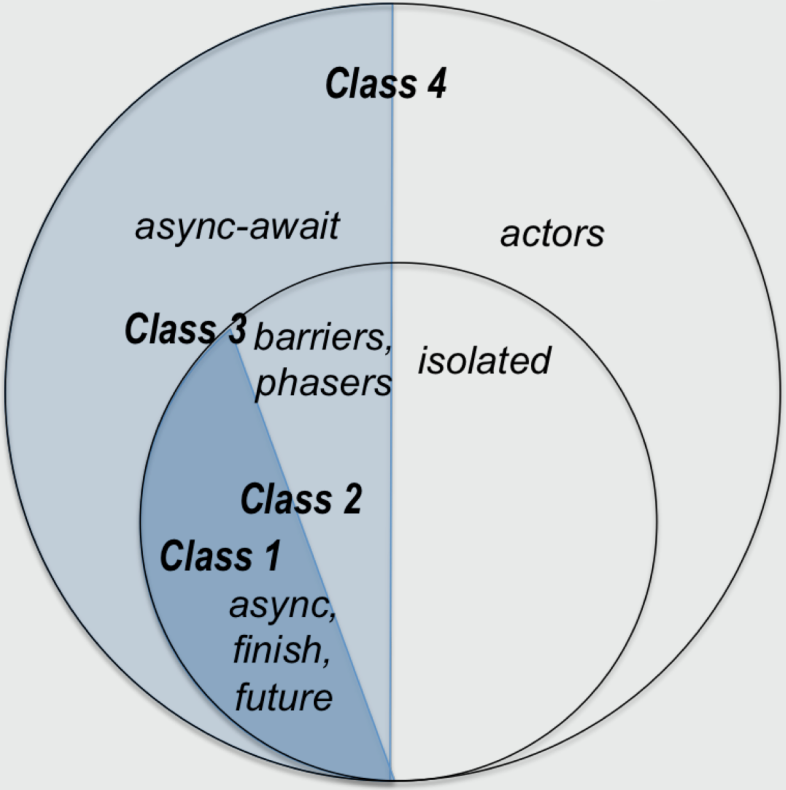
1) Lightweight asynchronous tasks and data transfers
- Creation: *async tasks, future tasks, data-driven tasks*
- Termination: *finish, future get, await*
- Data Transfers: *asyncPut, asyncGet*

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



"Habanero-Java: the New Adventures of Old X10." Vincent Cave, Jisheng Zhao, Jun Shirako, Vivek Sarkar PPPJ 2011.

Properties of interest

- DLF = DeadLock-Freedom
- DRF = Data-Race-Freedom
- DET = Determinacy
  (Functional & Structural determinism)
- DRF➔DET = DRF implies DET
- DRF➔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 ➔ DLF: Deadlock freedom guaranteed with futures if there are no data races on future objects
- DRF ➔ 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 ➔ all executions have same output*
- *Structural Determinism property*
  - *Same input ➔ 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*

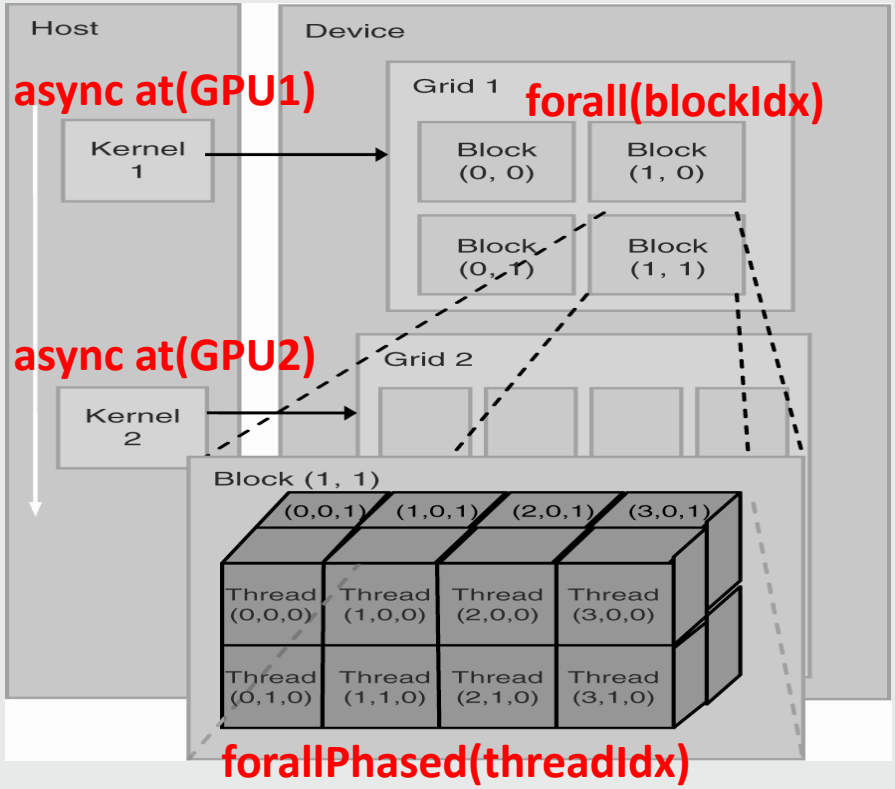
"Automatic Parallelization of Pure Method Calls via Conditional Future Synthesis". R.Surendran, V.Sarkar. OOPSLA 2016.

"Deadlock Avoidance in Parallel Programs with Futures: Why parallel tasks should not wait for strangers". T.Cogumbreiro, R.Surendran, F.Martins, V.Sarkar, V.Vasconcelos, M.Grossman. OOPSLA 2017.

Pedagogy using Habanero execution model

- **Sophomore-level CS Course at Rice (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
- **3-course introductory Coursera specialization on "Parallel, Concurrent and Distributed Programming in Java" launched in July 2017 (http://bit.ly/pcdpjava)**
  - 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

Example: Habanero abstraction of a CUDA kernel invocation in COMP 322



COMP 322's learning outcomes: Module 1

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

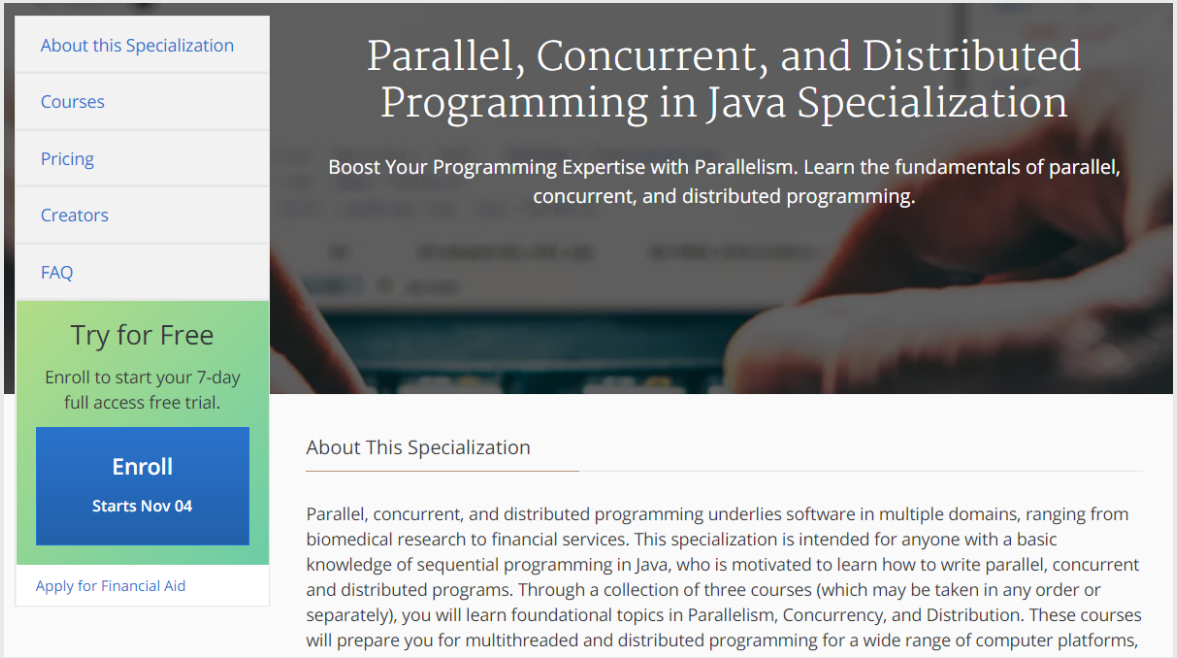
Advanced

- 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

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

↓

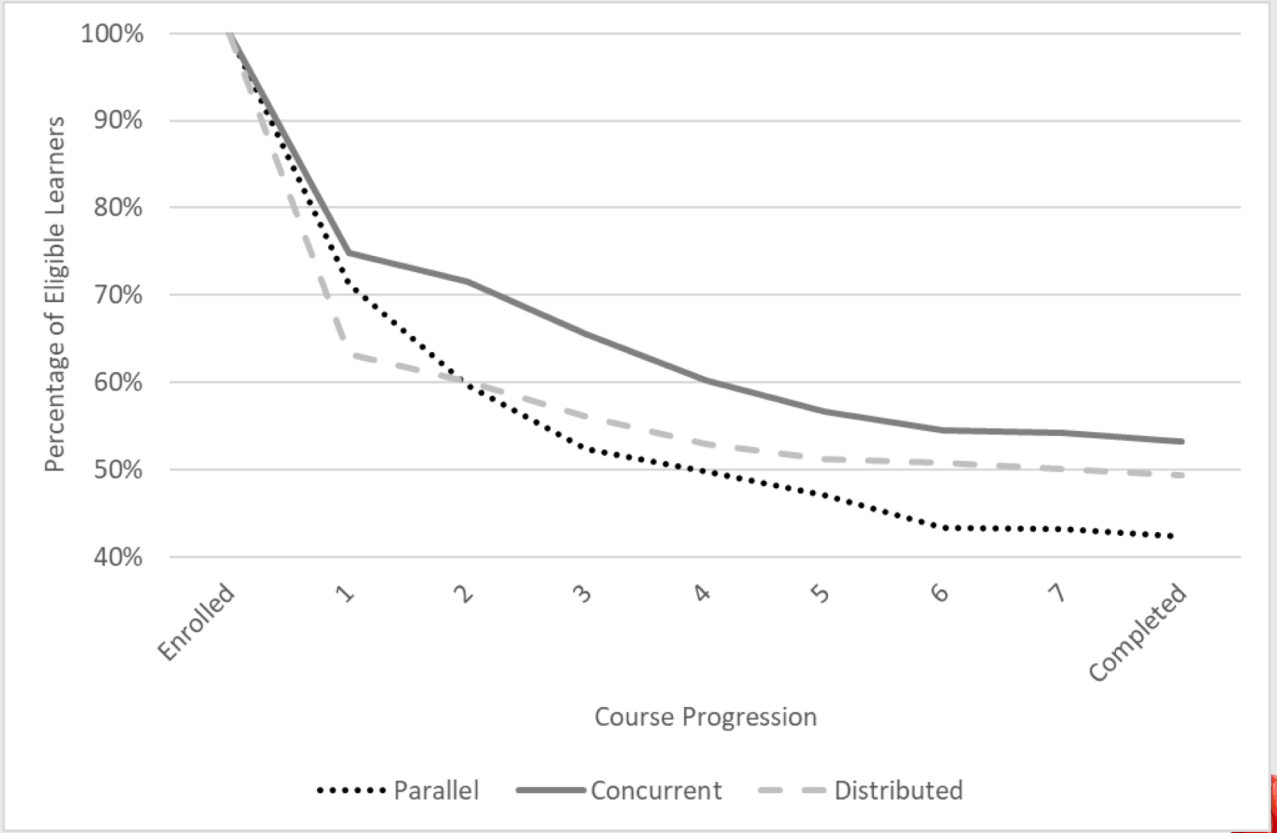
Week 3 Module

↓

Week 4 Module

↓

Wrap-Up



Key challenge in online courses: how to motivate students to get started on content!

Introductory Module
No content, test mini-project.

↓

Week 1 Module

↓

Week 2 Module

↓

Interview Video

↓

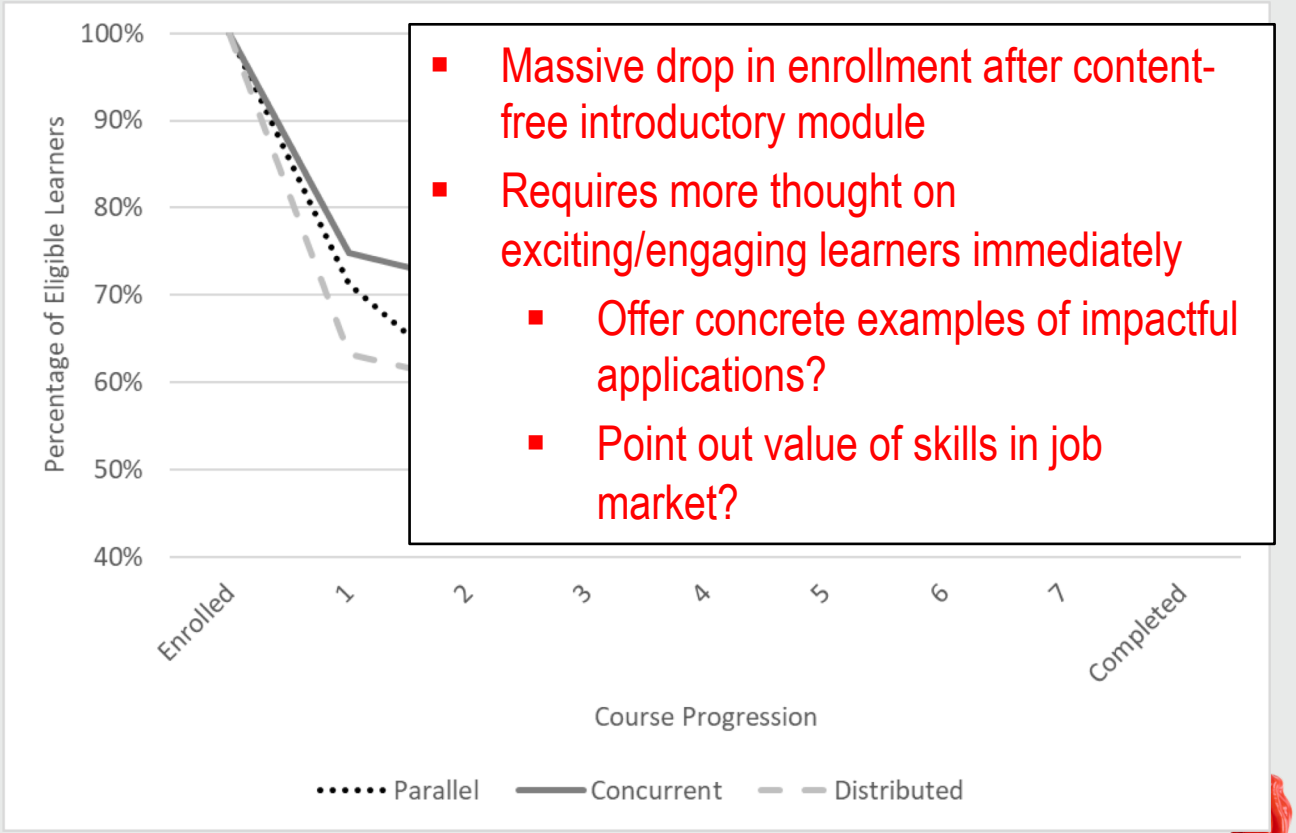
Week 3 Module

↓

Week 4 Module

↓

Wrap-Up



- 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

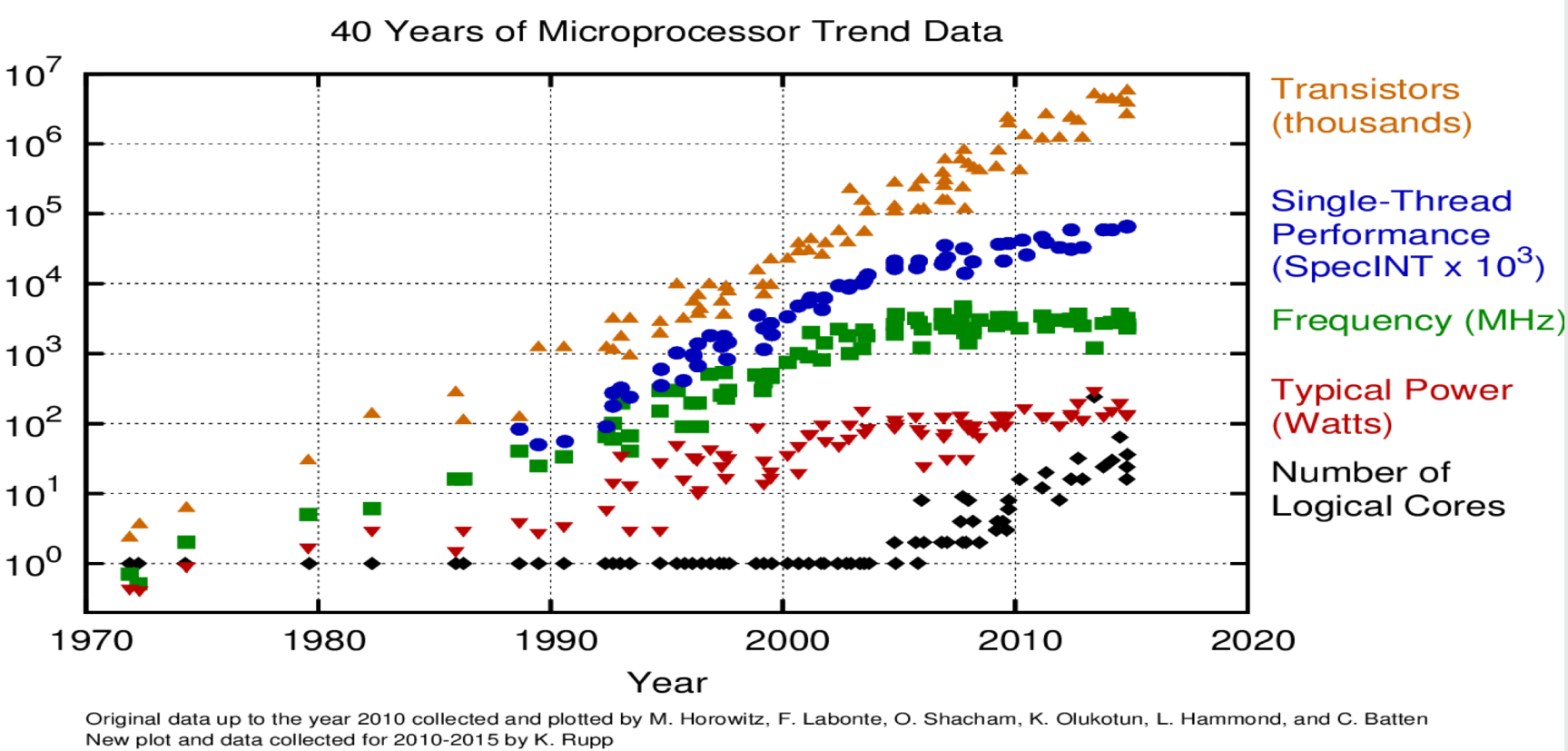
- Experiences with teaching Parallel, Concurrent, and Distributed Programming at undergraduate & continuing-education levels

- Future Computing Trends and the need for new pedagogies

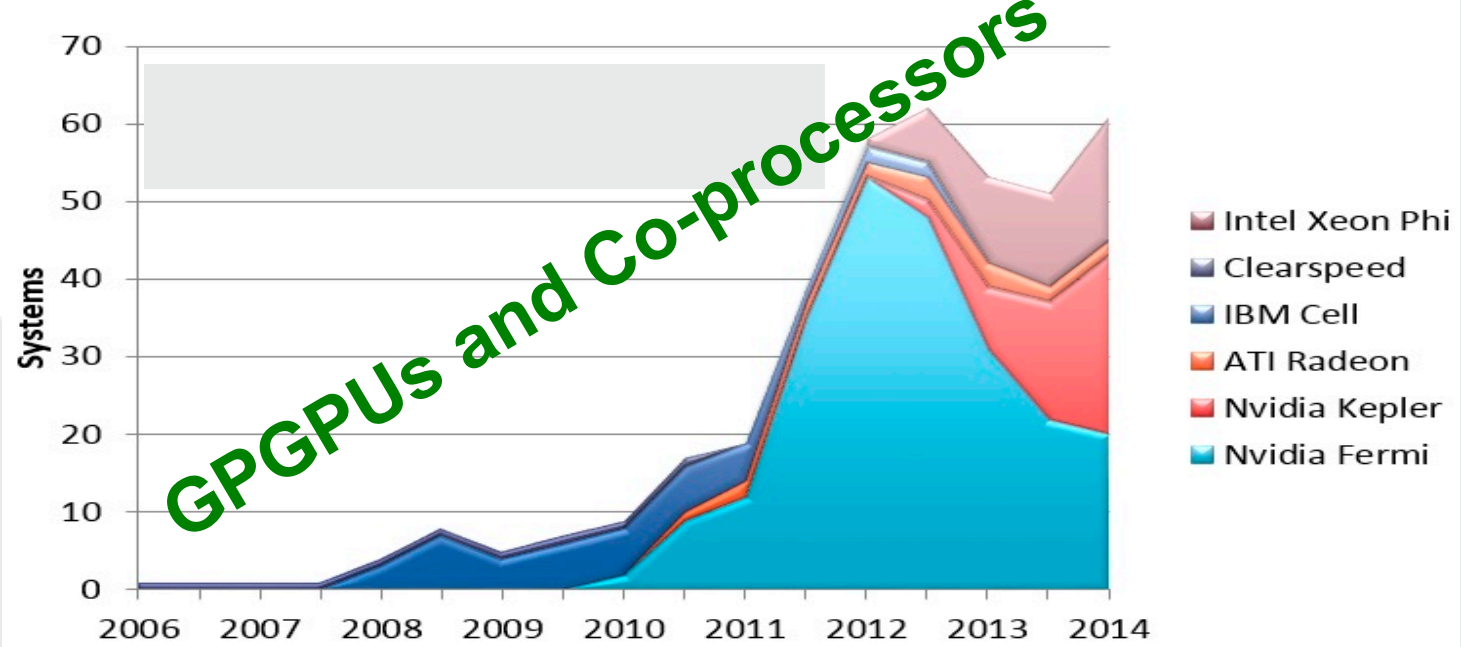
Technology disruptions – from “killer micros” to the ”killer of micros”

- 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



Accelerators in Top500



- 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:

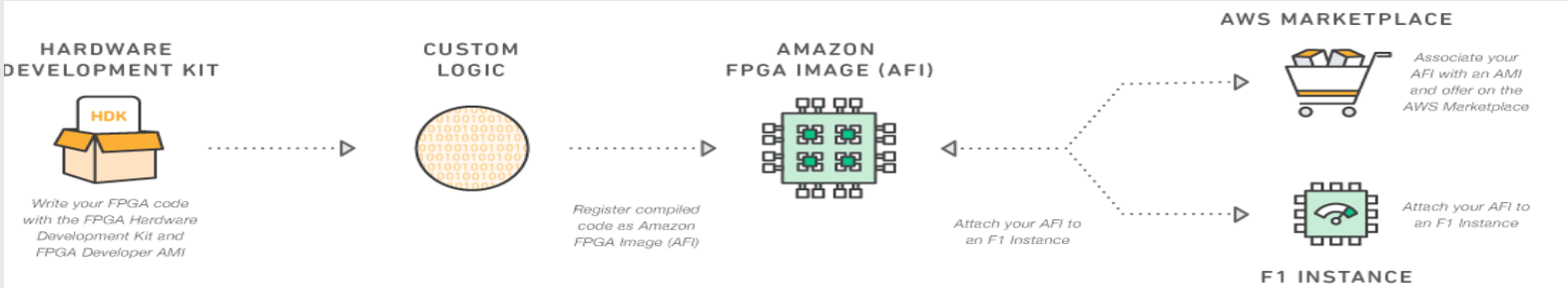
- High Frequency Intel Xeon E5-2686v4 (Broadwell) Processors
- High-performance NVIDIA K80 GPUs, each with 2,496 parallel processing cores and 12GiB of GPU memory
- Supports GPUDirect™ (peer-to-peer GPU communication)
- Provides Enhanced Networking using the Amazon EC2 Elastic Network Adaptor with up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

Model	GPUs	vCPU	Mem (GiB)	GPU Memory (GiB)
p2.xlarge	1	4	61	12
p2.8xlarge	8	32	488	96
p2.16xlarge	16	64	732	192

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.

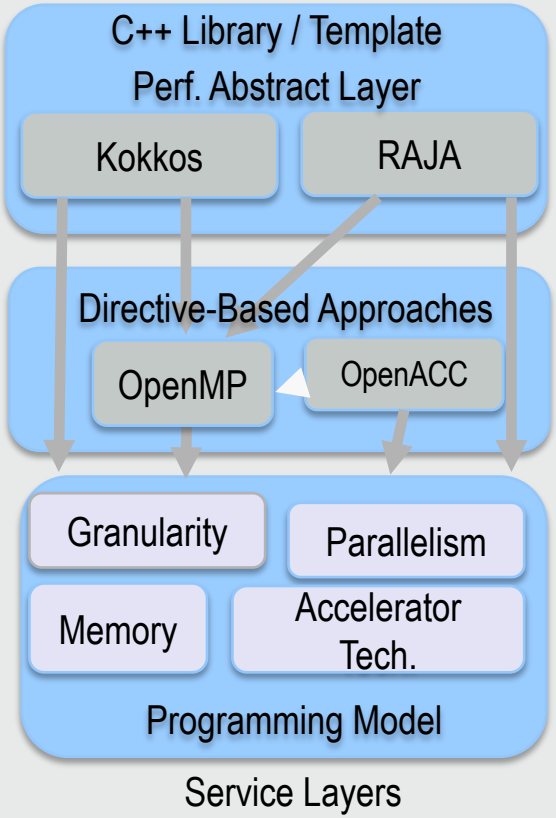
PURCHASE

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

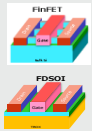
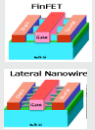
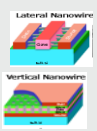
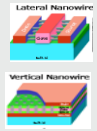
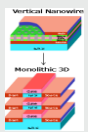
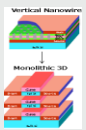
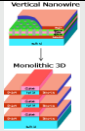
	CUDA	HIP	Kokkos	OpenACC	OpenMP	RAJA
Languages	C/C++	C/C++	C/C++	C/C++/ Fortran	C/C++/ Fortran	C/C++
Prog. Style	Extensions	Extensions	Template Meta-programming, C++11 lambdas	Directive-based	Directive-based	C++11 lambdas
Parallelism	SIMT	SIMT	OpenMP, Pthreads, CUDA	SIMD, CUDA, Fork-Join	SPMD, SIMD, Tasks, CUDA, Fork-Join	OpenMP, CUDA
Licensing/ Accessibility	Proprietary	Open-Source	Open-source	Open-Source	Open-source	Open-source
Abstraction Level	Low	Low	High	High/Medium	High/Medium	High



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*

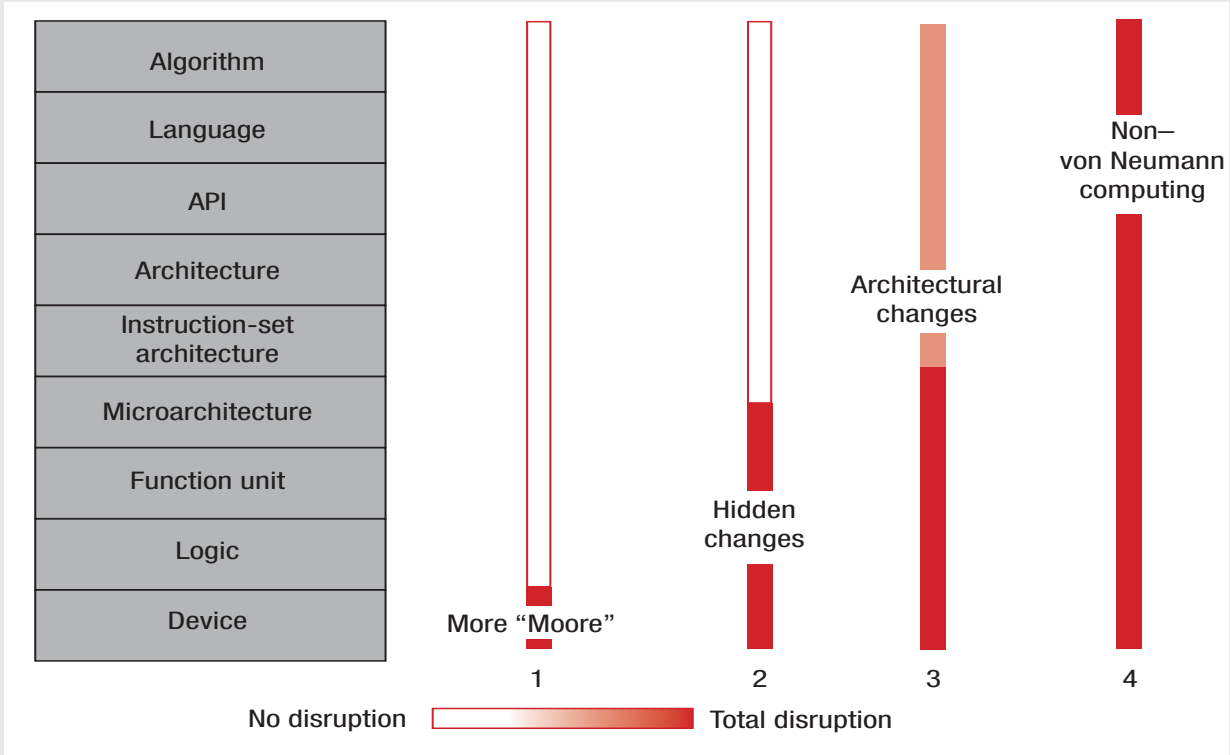
Table MM01 - More Moore - Logic Core Device Technology Roadmap

YEAR OF PRODUCTION	2017	2019	2021	2024	2027	2030	2033
	P54M36	P48M28	P42M24	P36M21	P28M14G1	P26M14G2	P24M14G3
Logic industry "Node Range" Labeling (nm)	"10"	"7"	"5"	"3"	"2.1"	"1.5"	"1.0"
IDM-Foundry node labeling	i10-f7	i7-f5	i5-f3	i3-f2.1	i2.1-f1.5	i1.5-f1.0	i1.0-f0.7
Logic device structure options	finFET FDSOI	finFET LGAA	LGAA VGAA	LGAA VGAA	VGAA M3D	VGAA M3D	VGAA M3D
Logic device mainstream device	finFET	finFET	LGAA	LGAA	VGAA	VGAA	VGAA
Logic device technology naming							
Patterning technology inflection for Mx interconnect	193i	193i, EUV	193i, EUV	193i, EUV	193i, EUV	193i, EUV	193i, EUV
Channel material technology inflection	Si	SiGe25%	SiGe50%	Ge, IIIV (TFET)	Ge, IIIV (TFET)	Ge, IIIV (TFET)	Ge, IIIV (TFET)
Process technogy inflection	Conformal deposition	Conformal Doping, Contact	Channel, RMG	CFET	Seq. 3D	Seq. 3D	Seq. 3D
Stacking generation	2D	2D	2D 3D: W2W or D2W	3D: P-over-N	3D: SRAM-on-Logic	3D: Logic-on-Logic, Hetero	3D: Logic-on-Logic, Hetero
Design-technology scaling factor for standard cell	-	1.11	2.00	1.13	0.53	1.00	1.00
Design-technology scaling factor for SRAM (111) bitcell	1.00	1.00	1.00	1.00	1.25	1.00	1.00
Number of stacked devices in one tier	1	1	3	4	1	1	1
Tier stacking scaling factor for SoC	1.00	1.00	1.00	1.00	1.80	1.80	1.80
Vdd (V)	0.75	0.70	0.65	0.60	0.50	0.45	0.40
Physical gate length for HP Logic (nm)	20.00	18.00	14.00	12.00	10.00	10.00	10.00
SoC footprint scaling node-to-node - 50% digital, 35% SRAM, 15% analog+IO	-	64.9%	51.3%	64.3%	64.2%	50.9%	50.7%

Source: IEEE IRDS 2017 Edition

30

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***.

Source: "Rebooting Computing: The Road Ahead", T.M.Conte, E.P.DeBenedictis, P.A.Gargini, E.Track, IEEE Computer, 2017.

Disruption in Mainstream Computing Stack has begun at both ends …

Examples:

Deep learning algorithms

Go, Julia, Halide, Python

Caffe, Spark, TensorFlow

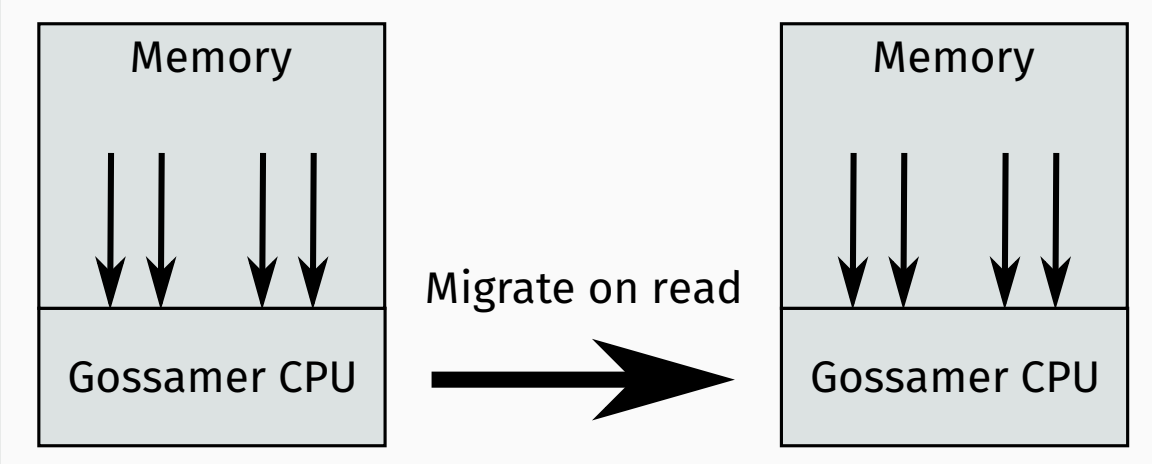
… and guess where they meet in the middle!

Accelerators, Reconfigurable Logic

Memory-centric Computing

3D devices

Disruptive Example: Emu Chick

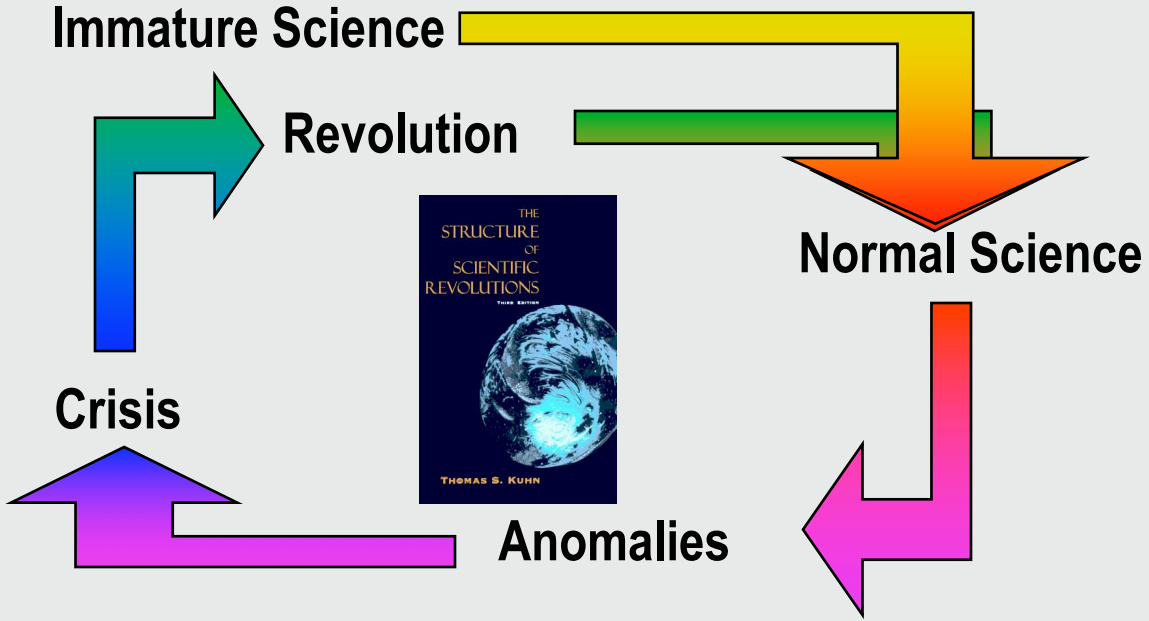



```
// Habanero abstraction of
// remote read of location A
at (HOME(A)) {
      continuation after reading A,
          . . .
}
```

- "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.

Kuhn's History of Science

"The Structure of Scientific Revolutions", Thomas S. Kuhn (1970)



Revolution: A new paradigm emerges

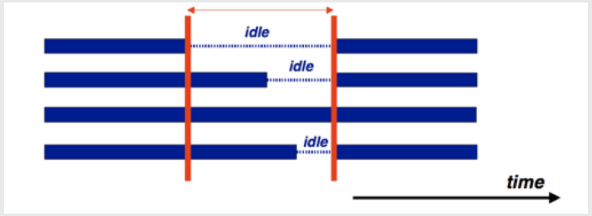
Old Theory: well established, many followers, many anomalies, attempts to address anomalies by "puzzle solving activities" governed by the rules of the paradigm

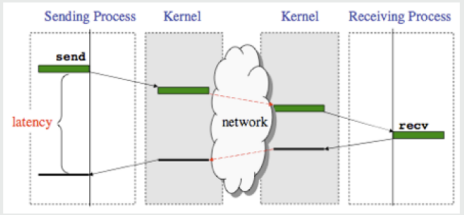
New Theory: few followers, untested, new concepts/techniques, accounts for anomalies, asks new questions

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

```
{x = a + b;
 y = b * 7
in
    (x-y) * (x+y)}
```

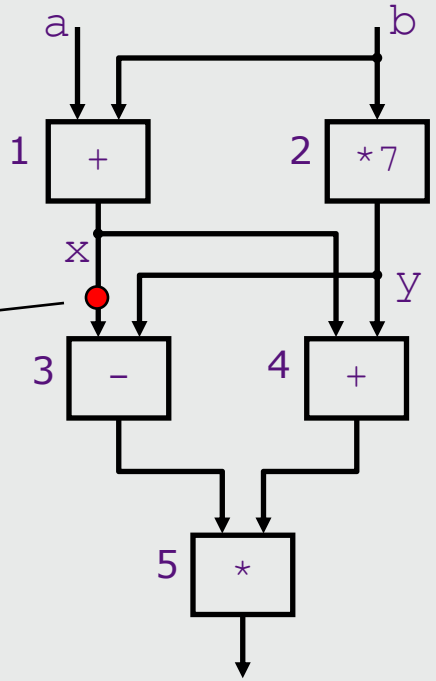
Values in dataflow graphs are represented as tokens

token < ip , p , v >

instruction ptr, port, data

ip = 3

p = L



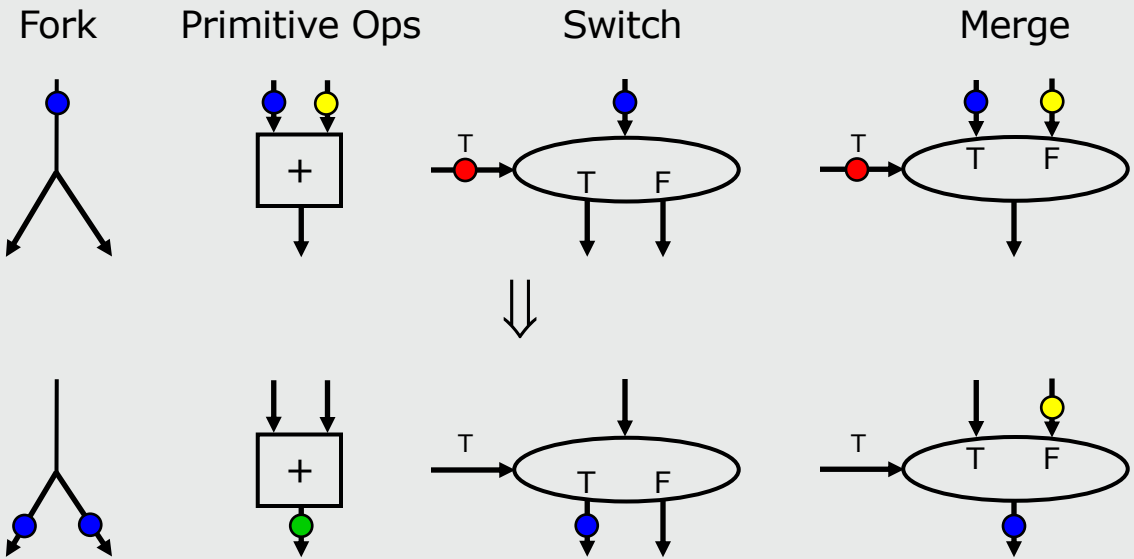
An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators

no separate control flow

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

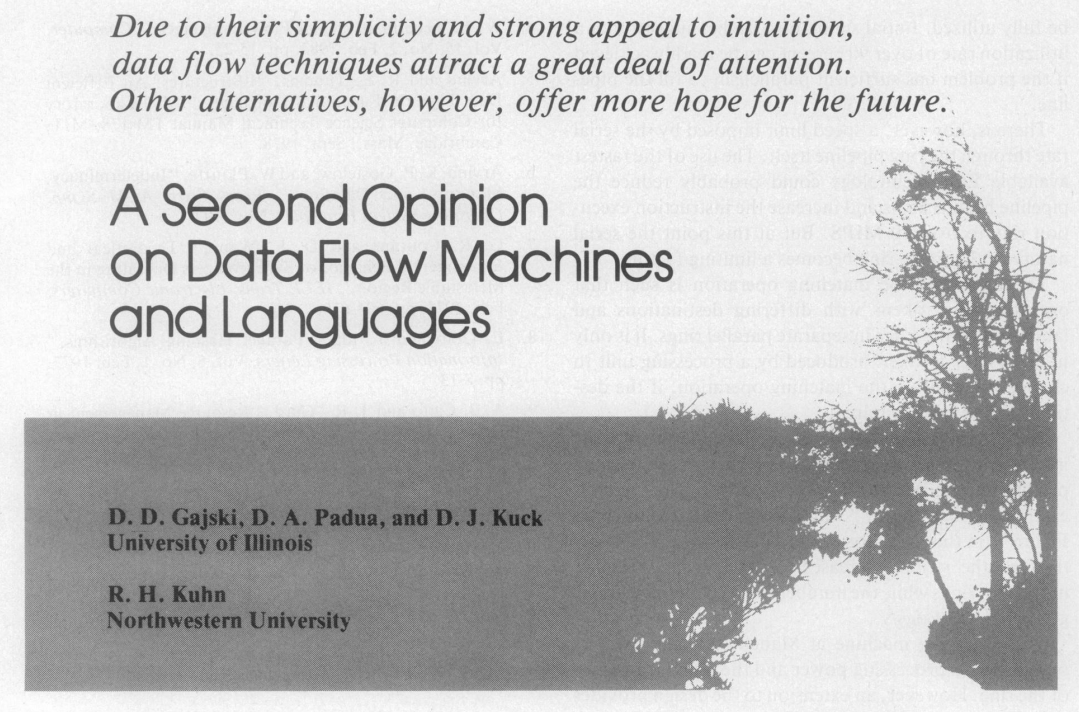
Dataflow Operators

A small set of dataflow operators can be used to define a general programming language



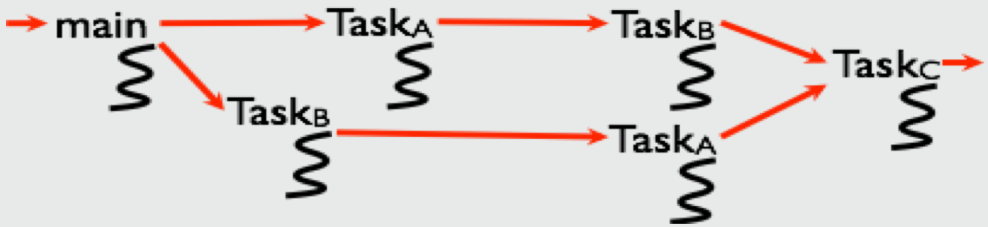
Source: "Dataflow: Passing the Token", talk given by Prof. Arvind at ISCA 2005

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.

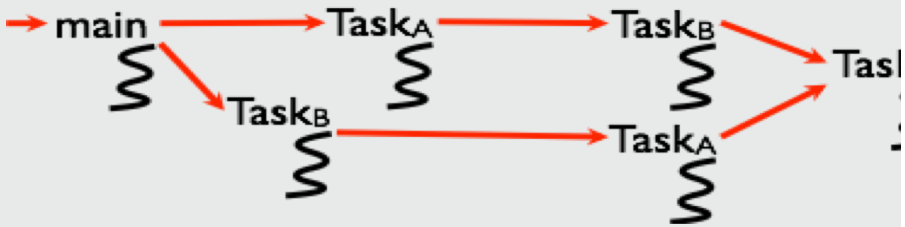
A Third Opinion -- Dataflow Execution Models offer a promising foundation for software and hardware in future "extreme heterogeneity" platforms.



Event synchronization /
Promise communication

- 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 Mo
software and hardware in future "extreme



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

RICE

Partitioning Parallel Programs for Macro-Dataflow

Vivek Sarkar and John Hennessy
Computer Systems Laboratory
Stanford University

Abstract

Partitioning techniques are necessary to execute functional programs at a coarse granularity. Fine granularity execution is inefficient on general purpose multiprocessors. There is a trade-off between parallelism and the overhead of exploiting parallelism. We present a compile-time partitioning approach to achieve this trade-off.

1. Introduction

Functional programs offer implicit parallelism at all levels. Data dependencies are their only sequencing constraints. Several parallel evaluation models exist for functional languages, e.g. dataflow [8], graph reduction [17], Concurrent Prolog [14]. These models define a granularity of parallelism at the finest level possible, e.g. instructions in dataflow, combinators in graph reduction, goals in Concurrent Prolog. The enormous scheduling and communication overhead incurred by fine grain parallelism has prompted several implementers to attempt a coarser granularity. In some implementations, the level of granularity is determined by language constructs, such as compound expressions or user-defined functions, causing the programming style to dramatically affect multiprocessor performance. We believe that the optimal granularity should be dictated by performance characteristics - specifically execution time, communication overhead and scheduling overhead. It should represent a trade-off between parallelism and the overhead of exploiting parallelism.

In this paper, we present a compile-time partitioning algorithm to partition program graphs into subgraphs that can execute in parallel. This partition provides a coarser 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(Π), that defines the cost of partition Π. The partitioning algorithm attempts to build a partition with the smallest value of F(Π).

The dataflow model is traditionally defined at the granularity of instructions or dataflow operators. With our compile-time partition, 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 be able to run to completion once all its inputs are available. This allows for non-preemptive run-time scheduling with no task-switching overhead. Cyclic dependencies are thus forbidden among macro-actors. This restriction is called the *convexity constraint* and is discussed in detail in Section 4.

A compile-time partitioner has been implemented to process program graphs in the intermediate language, IF1 [16]. IF1 represents computation as dataflow graphs, as described in Section 5. A list of target parameters (e.g. number of processors, communication and scheduling overhead) drives the partitioning for a given multiprocessor architecture. Using a front-end from SISAL [11] to IF1, we apply this system to programs written in the single-assignment language SISAL. However, our approach is applicable to any

This work has been supported by the National Science Foundation under grant # DCR8351269 and by the Defense Research Projects Agency under contract # MDA 903-83-C-0335.

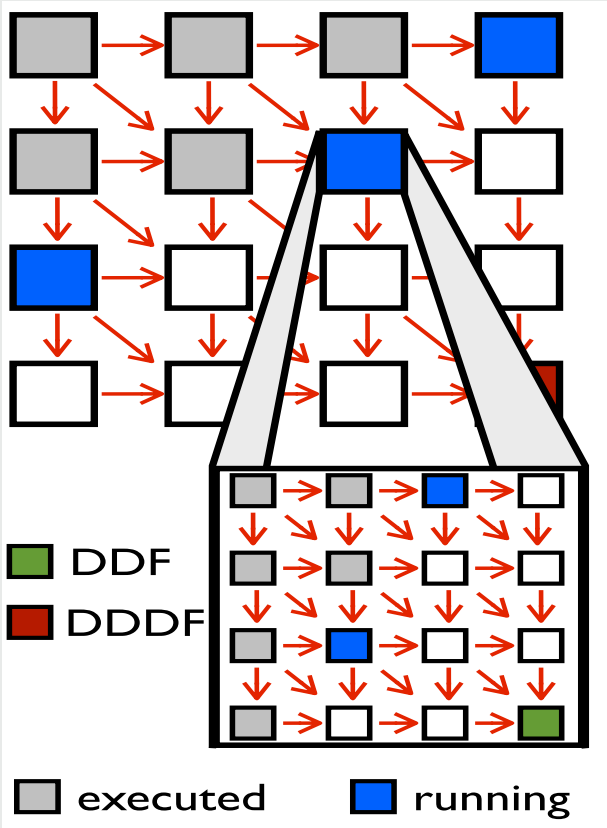
Why now for Dataflow Execution Models?

- Characteristics of extreme heterogeneity platforms
  - Increasing number of heterogenous 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

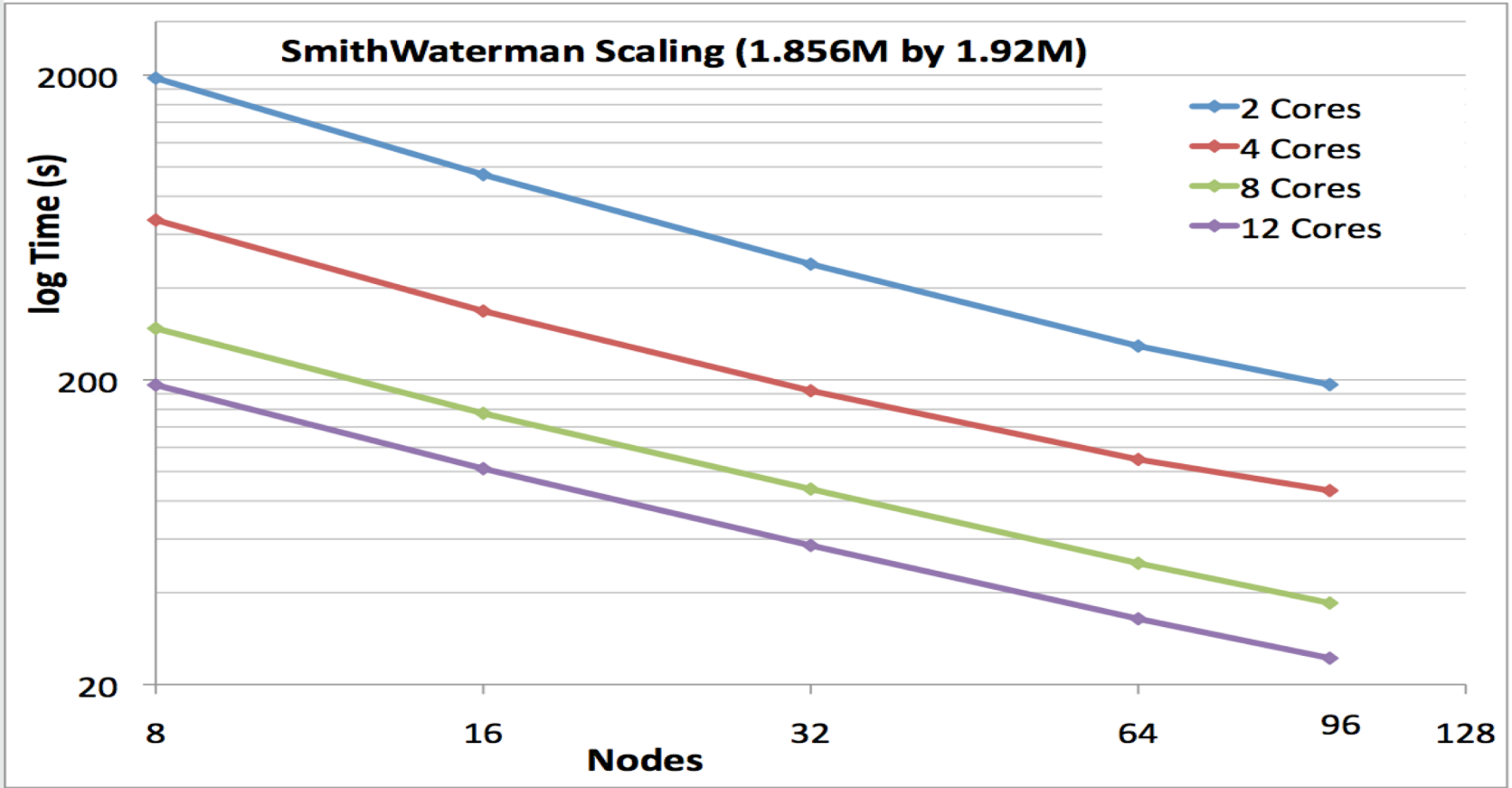
- Machine Learning frameworks, e.g., TensorFlow
- OpenMP “depends” clause (similar idea in OmpSS, Legion, …)
- TBB’s flow graph interface
- Dag parallelism in numerical libraries (PYRROS, PLASMA, SLATE, …)
- National Instruments’ LabView product
- Event-driven programming in JavaScript, reactive programming systems
- Futures and promises in modern languages (C++, Java, JavaScript, …)
- HPVM for heterogenous 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



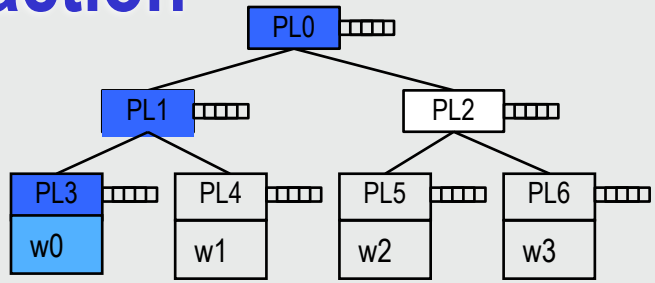

```
1. #define DDF_HOME(guid) . . .
2. . . .
3. for (i=0;i<H;++i)
4.   for (j=0;j<W;++j)
5.     matrix[i][j] = DDF_HANDLE(i*H+j);
6. . . .
7. finish { // matrix is a 2-D array of DDFs
8.   for (i=0,i<H;++i) {
9.     for (j=0,j<W;++j) {
10.      DDF_t* curr = matrix[i][j];
11.      DDF_t* above = matrix[i-1][j];
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)



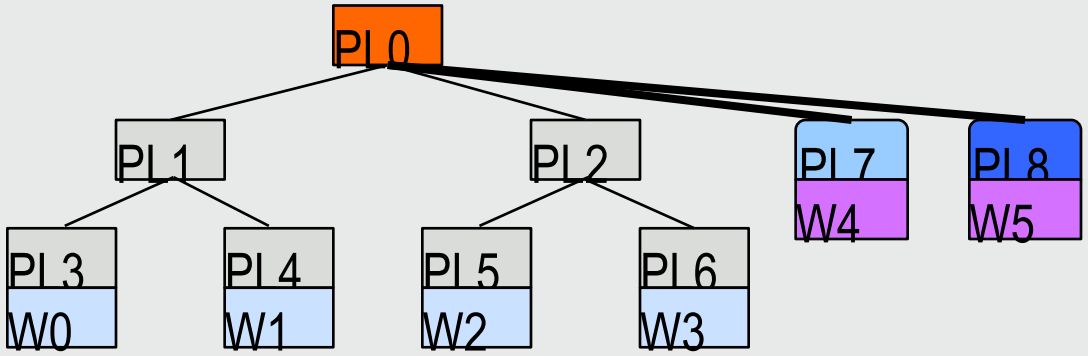
Locality-aware Scheduling for CPUs using the Hierarchical Place Tree (HPT) abstraction

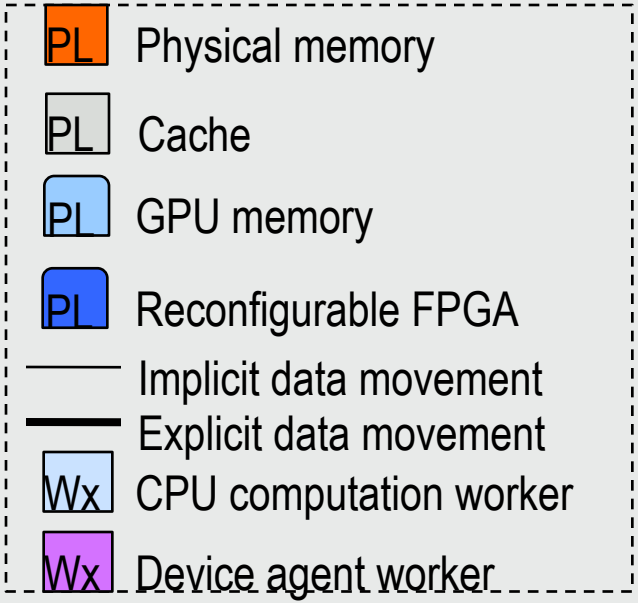
- Model target system as a tree/hierarchy
- Workers attached to leaf places
  - Bind to hardware core
- Each place has a queue
  - **async at**(<*pl*>) <*stmt*>: push task onto place *pl*'s queue
  - Destination place can be determined by programmer, compiler or runtime
- Current policy
  - A worker 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



Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Date Movement. Yonghong Yan, Jisheng Zhao, Yi Guo, Vivek Sarkar. LCPC 2009.

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**

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

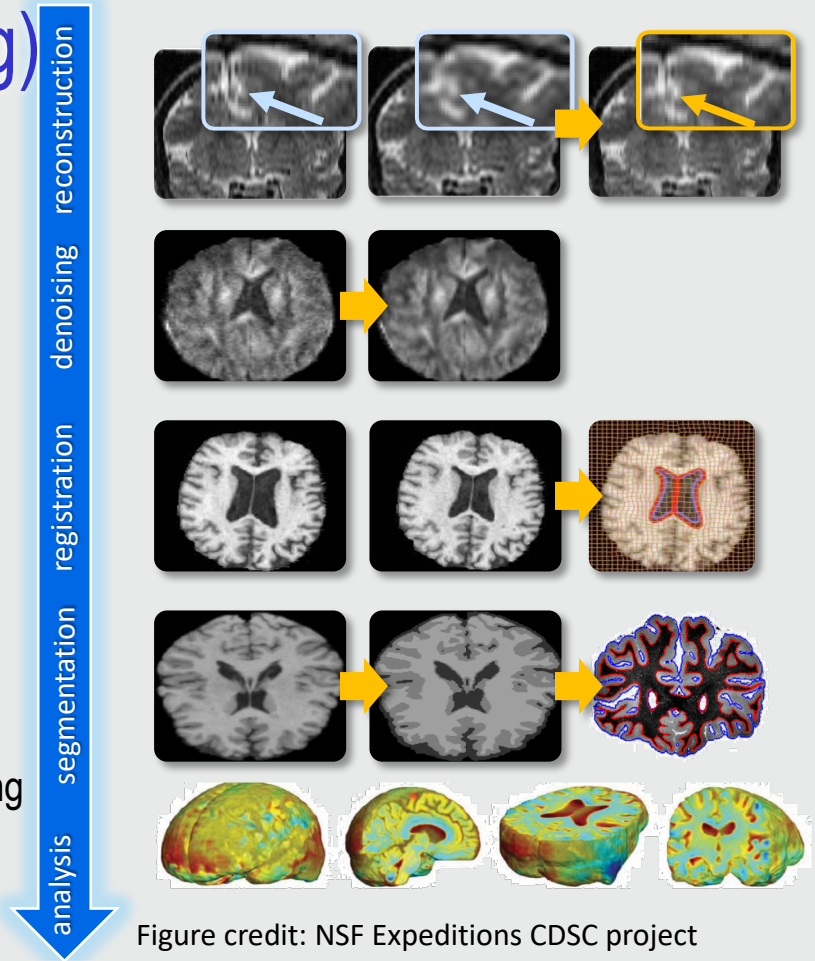


Figure credit: NSF Expeditions CDSC project


Aberle


Baraniuk


Bui

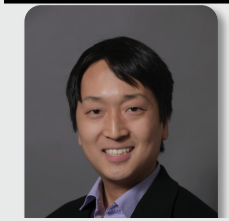

Chang


Chien


Cheng


Cong (Director)

	UCLA	Rice	UCSB	Ohio State
Domain-specific modeling	Bui, Reinman, Potkonjak	**Sarkar,** Baraniuk		Sadayappan
CHP creation	Chang, Cong, **Reinman**		Cheng	
CHP mapping	Cong, Palsberg, Potkonjak	**Sarkar**	Cheng	Sadayappan
Application drivers	Aberle, **Bui,** Chien, Hsu, Vese	Baraniuk		
Experimental systems	All (led by **Cong** & **Bui**)	All	All	All


Hsu


Palsberg


Potkonjak

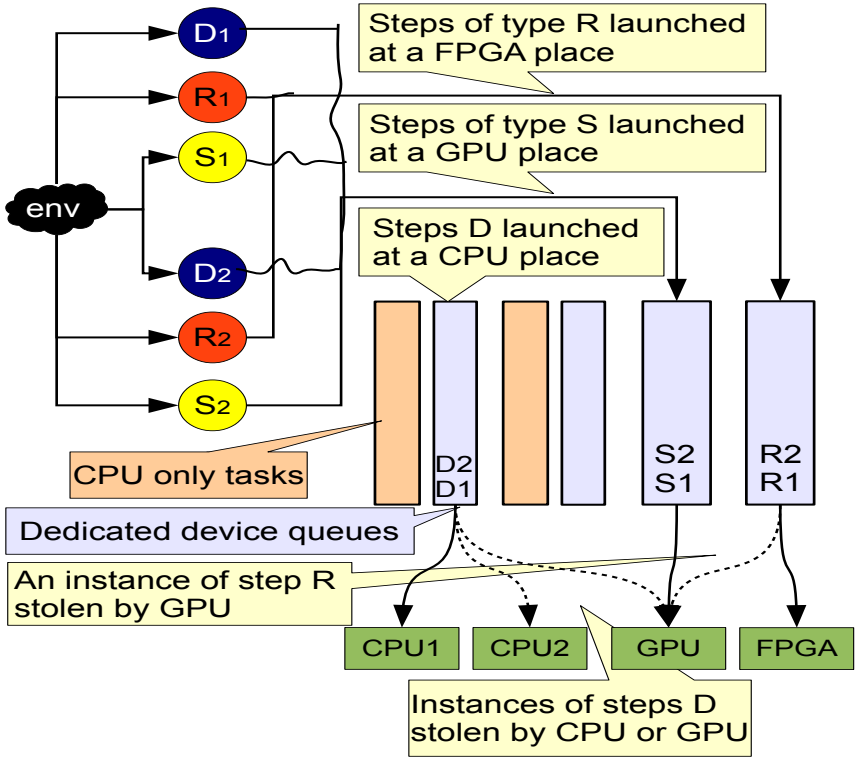

Reinman


Sadayappan


Sarkar (Associate Director)


Vese

Use of Dataflow Model for Heterogeneous Computing in CDSC for CPU+GPU+FPGA platform



DFGL graph representation extended with affinity annotations:

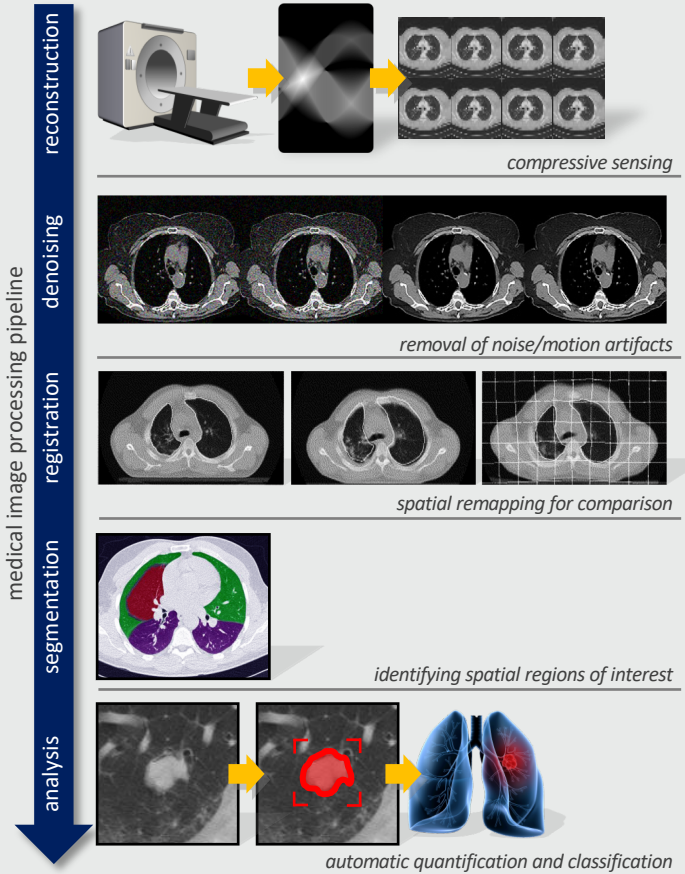
- < C > :: (D @CPU=20,GPU=10);
- < C > :: (R @GPU=5, FPGA=10);
- < C > :: (S @GPU=12);

- [IN : k-1] → (D : k) → [IN2 : k+1];
- [IN2 : 2*k] → (R : k) → [IN3 : k/2];
- [IN3 : k] → (S : k) → [OUT : IN3[k]];

- env → [IN : { 0 .. 9 }], < C : { 0 .. 9 } >;
- [OUT : 1] → 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)	CDSC implementation (2015)
1.5 min (Regular CT)	6 min (Low-dose CT)
18 hours (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
```

Step A

Step B1

Step B2

Step B3

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]