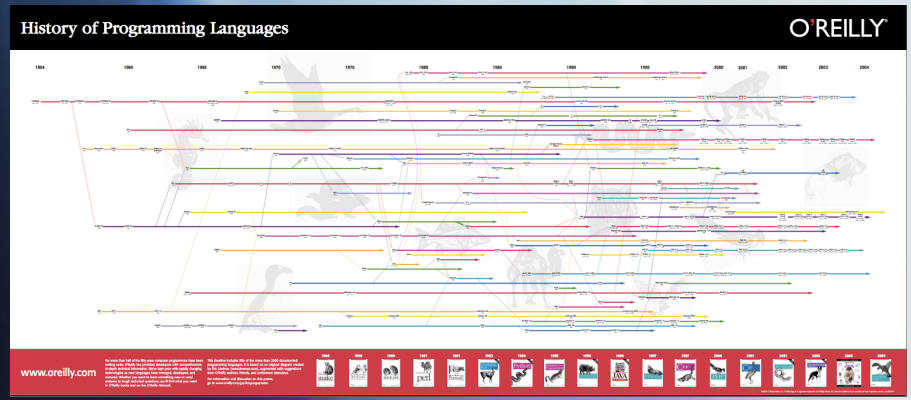
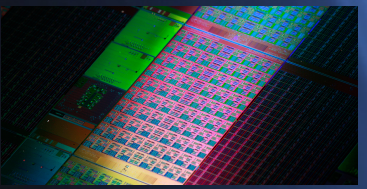
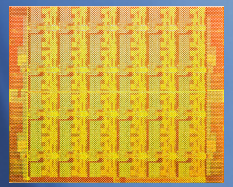
RICE George R. Brown School of Engineering
Computer Science

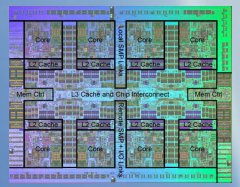
Programming Models for Exascale Systems: What, When and How?

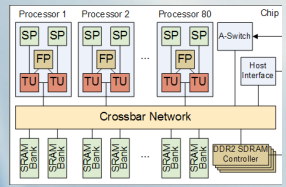


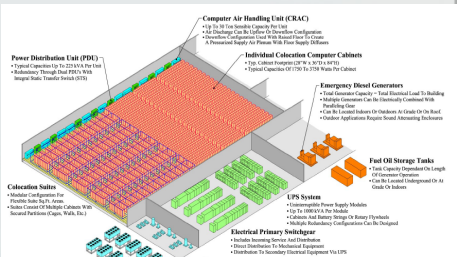
Vivek Sarkar
E.D. Butcher Chair in Engineering
Professor of Computer Science
Rice University
vsarkar@rice.edu











Early Days of HPC (1980s): Statically Predictable Hardware w/ Minimal Runtime Software

- Vector machines
  - Compiler/user code vectorization + runtime library for vectorized intrinsics
- Early SMPs
  - Evolution of SPMD model with self-scheduling/auto-tasking
  - Runtime support focused on synchronization libraries for barriers, work-sharing, atomics, etc.
- Early distributed memory systems
  - Low-level message-passing libraries

Two important trends in 1990s

1. Adoption of clusters as path to scalable parallelism
   - Led to standardization of MPI
   - Question: how would MPI have been defined if processors had very high core counts in the 1990s?
2. Impact of caches and memory hierarchy on performance

➔ Low-hanging fruit for parallel computing was found in “regular” applications where both 1. and 2. could be statically predicted to some degree

➔ Irregular applications could only be supported with a big loss in programmability

Fast Forward to Exascale & Extreme Scale Systems

- Characteristics of Extreme Scale systems in the next decade
  - *Massively multi-core (~ 100's of cores/chip)*
  - *Performance driven by parallelism, constrained by energy & data movement*
  - *Subject to frequent faults and failures*
- Many Classes of Extreme Scale Systems

Mobile, < 10 Watts, O(10^1) concurrency

Terascale Embedded, 100's of Watts, O(10^3) concurrency

Petascale Departmental, 100's of KW, O(10^6) concurrency

Exascale Data Center > 1 MW, O(10^9) concurrency

RICE

Source: DARPA Exascale Study, 2008; DARPA Exascale Software study, 2009.

Opportunities for Order-of-Magnitude Improvements through Hardware-Software Customization (AES example)

AES 128bit key 128bit data	Throughput	Power	Figure of Merit (Gb/s/W)
0.18mm CMOS	3.84 Gbits/sec	350 mW	11 (1/1)
FPGA [1]	1.32 Gbit/sec	490 mW	2.7 (1/4)
ASM StrongARM [2]	31 Mbit/sec	240 mW	0.13 (1/85)
ASM Pentium III [3]	648 Mbits/sec	41.4 W	0.015 (1/800)
C Emb. Sparc [4]	133 Kbits/sec	120 mW	0.0011 (1/10,000)
Java [5] Emb. Sparc	450 bits/sec	120 mW	0.0000037 (1/3,000,000)

Acknowledgment: NSF Expeditions Center for Domain-Specific Computing

RICE

Performance Variability is on the rise in Extreme Scale Systems

- Concurrency --- increased performance variability with increased parallelism
- Energy efficiency --- increased performance variability with increased non-uniformity and heterogeneity in processors
- Locality --- increased performance variability with increased memory hierarchy depths
- Resiliency --- increased performance variability with fault tolerance adaptation (migration, rollback, redundancy, …)

How should exascale applications be programmed?

"Revolutionary" ideas ➔ *Evolutionary adoption*

- asynchronous parallelism everywhere
- portable computation and data mappings
- distributed global address/name space
- hierarchical abstractions of locality
- built-in support for failure recovery
- freedom from deadlocks, data races
- …

- *leverage new standards e.g., C++*
- *Influence and build on future versions of MPI + OpenMP*
- *Leverage library interfaces for ease of adoption & scalable performance*
- *Leverage compiler support for portability and intra-node performance*
- *. . .*

RICE

Rice Habanero Extreme Scale Software Research Project

Parallel Applications

Habanero Programming Languages

Habanero Compiler & PIR (Built on LLVM)

Habanero Runtime System (Built on OCR)

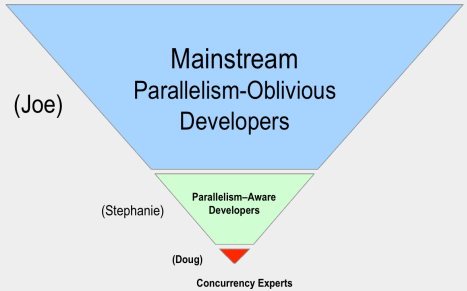
Extreme Scale Platforms

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*

Two-level programming model

1) Declarative Coordination Language for Domain Experts:
   CnC, DFGL
2) Task-Parallel Languages for Parallelism-aware Developers:
   Habanero-C, Habanero-C++, Habanero-Java, Habanero-Scala



RICE

http://habanero.rice.edu

Habanero 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 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*

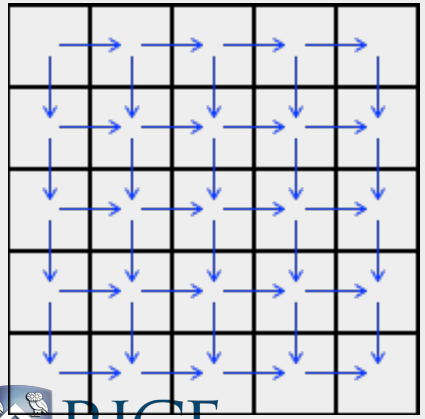
Claim: these execution model primitives enable programmability, portability, and performance for extreme scale software and hardware

Outline: Examples of Programming Models based on the Habanero Execution Model

1. Portable Heterogeneous Intra-Node Parallelism using a Data Flow Graph Language (DFGL)
2. Exploiting Heterogeneous Inter-Node Parallelism with Habanero-C++

Motivation: help Application Developers specify all the parallelism in their code in a portable manner

Current practice: use OpenMP task dependences



```
1. #pragma omp parallel
2. #pragma omp single
3. {
4.   for (int j = iymin; j < iymax; j++){
5.     for (int i = ixmin; i < ixmax; i++){
6. #pragma omp task depend(in:dataptr[i][j-1]) \
7.                  depend(in:dataptr[i-1][j]) \
8.                  depend(out:dataptr[i][j])
9.       process_cell(i,j,nu,ncellx,ncelly,vo,vi, …);
10.    } // for-i
11.  } // for-j
12.} // omp-parallel
```

… but this fine-grained task parallel version won't run efficiently on any platform!

Our Approach: use Data-Flow Graph Language (DFGL) as an embedded DSL amenable to compiler optimizations

```
// C/C++ code declaring vo, process_cell, etc

#pragma dfgl
{
  // Dependences
  [vo:j-1,i],[vo:j,i-1] -> (process_cell:j,i) -> [vo:j,i];

  // Iteration domain
  env :: (process_cell:{iymin..iymax-1},{ixmin..ixmax-1});
}
```

Access functions

Computation step instance

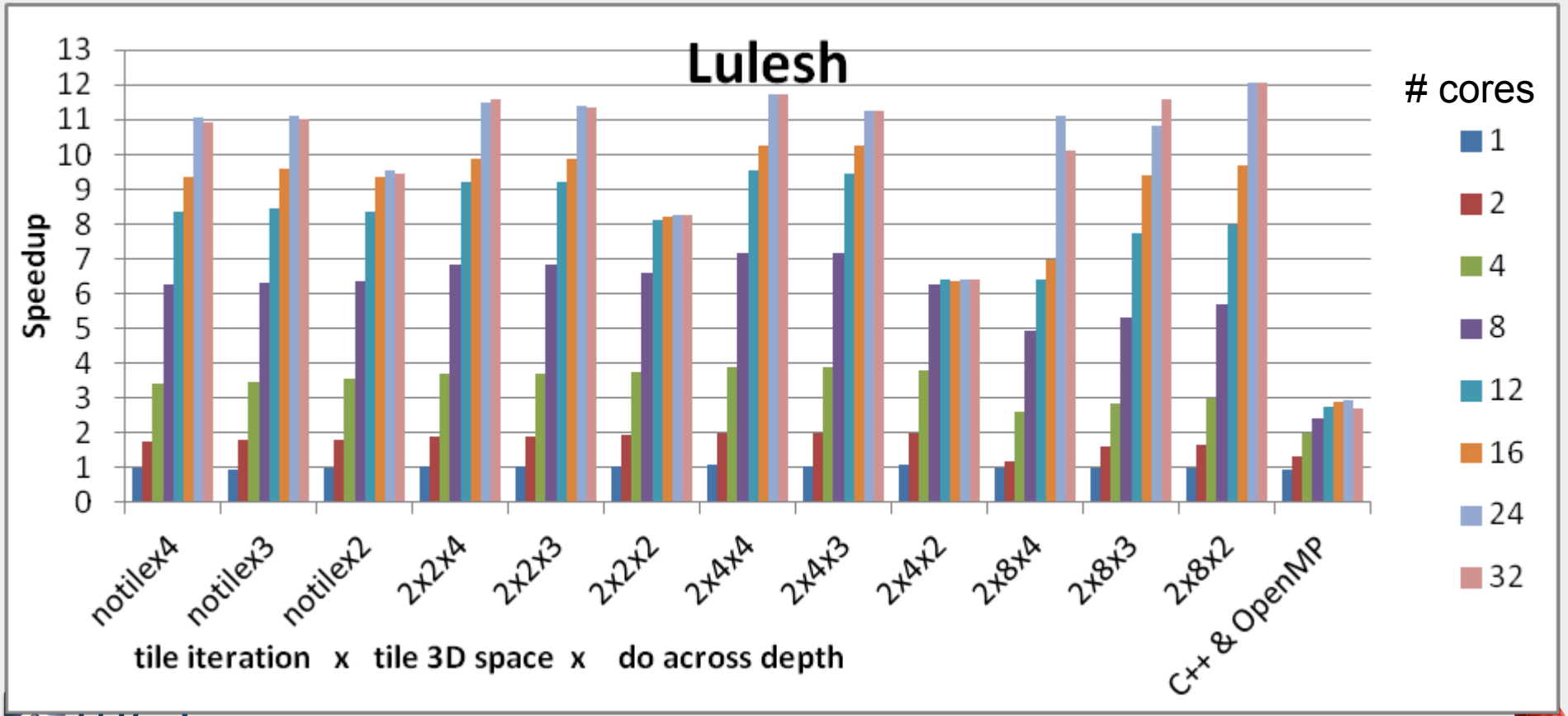
Iteration domains specified as ranges

The environment starts the initial steps in the graph

Prototype Implementation of DFGL in LLNL D-TEC project

- Automatic DFGL → SCoP → OpenMP transformations
  - OpenMP used as a portable target for higher level programming models
- SCoP transformations generate tiled OpenMP parallel code with new OpenMP 4.1 doacross construct for pipeline parallelism
  - Addition of doacross construct to OpenMP 4.1 standard was the result of a joint IBM+Rice proposal to the OpenMP standards committee
- Experimental results obtained for DFGL version of LULESH
  - Single POWER7 node: 32 cores, 3.86GHz (BlueBiou system @ Rice)
  - LULESH problem size: 50 iterations, 100x100x100 space

LULESH speedup on POWER7



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:
  - order-of-magnitude performance and energy efficiency improvements
  - real-time clinical applications and simulations using patient imaging data

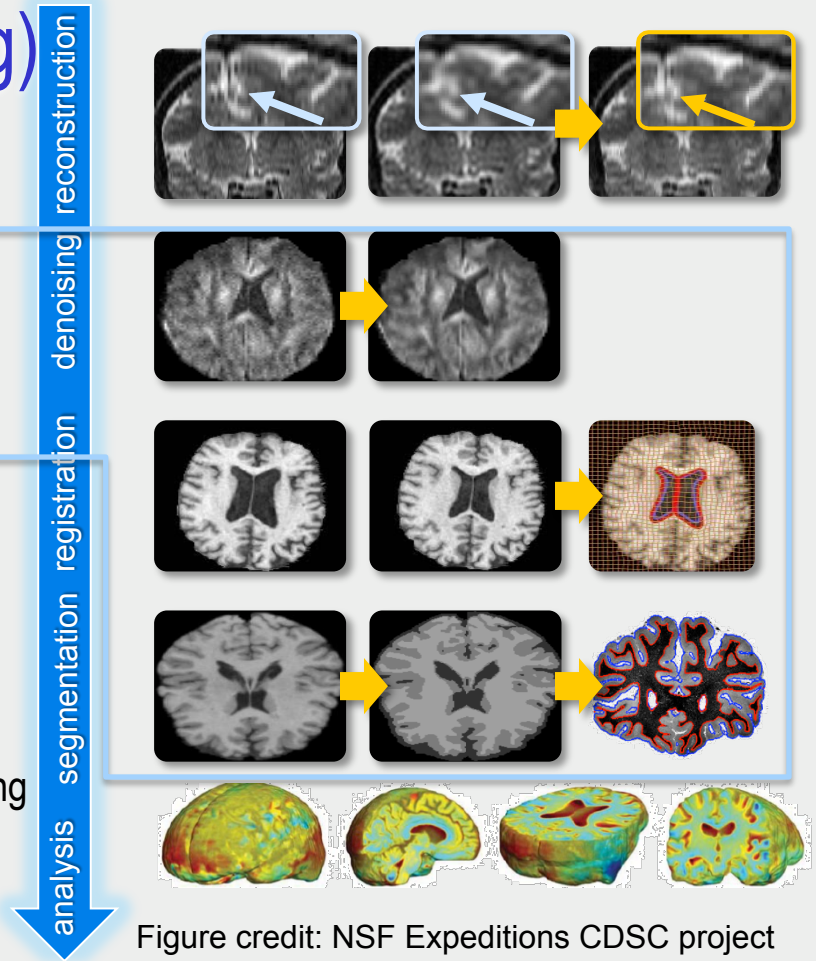
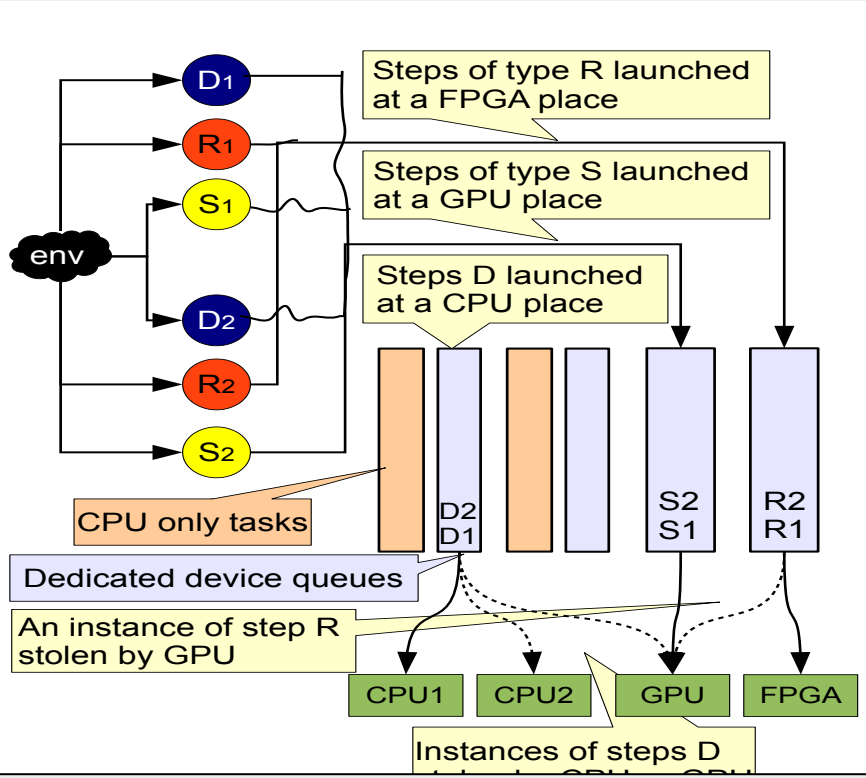


Figure credit: NSF Expeditions CDSC project

Adding Affinity Annotations for Heterogeneous Computing to Dataflow Model



DFGL graph representation extended with affinity annotations:

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

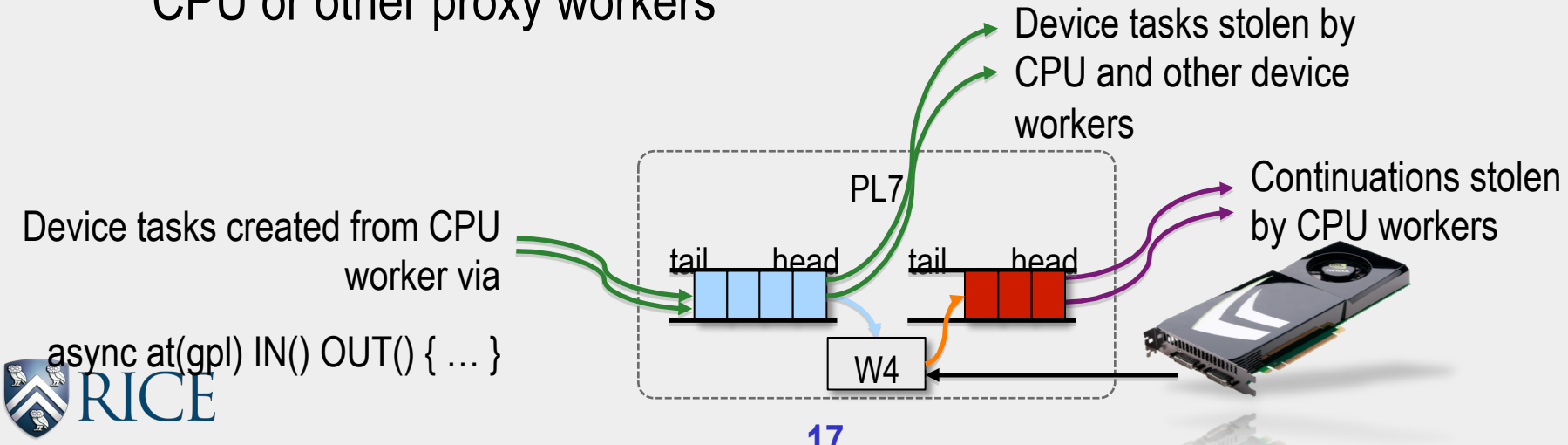
- [IN : k-1] → (D : k) → [IN2 : k+1];
- [IN2 : 2*k] → (R : k) → [IN3 : k/2];
- [IN3 : k] → (S : k) → [OUT : IN3[k]];

- env → [IN : { 0 .. 9 }], < C : { 0 .. 9 } >;
- [OUT : 1] → env;

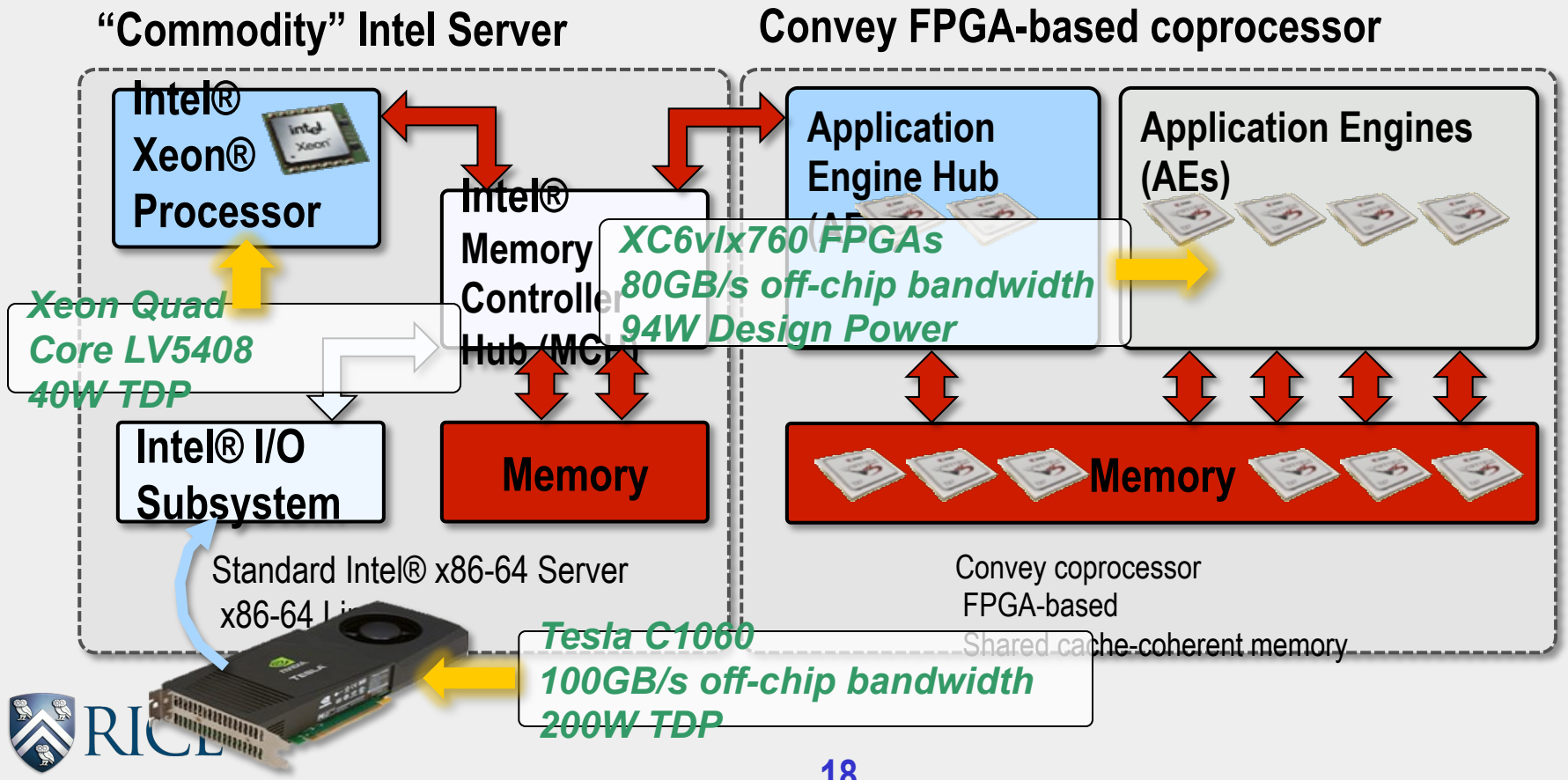
"Mapping a Data-Flow Programming Model onto Heterogeneous Platforms." Alina Sbirlea, Yi Zou, Zoran Budimlic, Jason Cong, Vivek Sarkar. LCTES 2012

Hybrid Scheduling with Heterogeneous Work Stealing

- Steps are compiled for execution on CPU, GPU or FPGA
  - Aim for single-source multi-target compilation!
- Designate a CPU core as a proxy worker for heterogeneous device
- Device inbox is now a concurrent queue and tasks can be stolen by CPU or other proxy workers



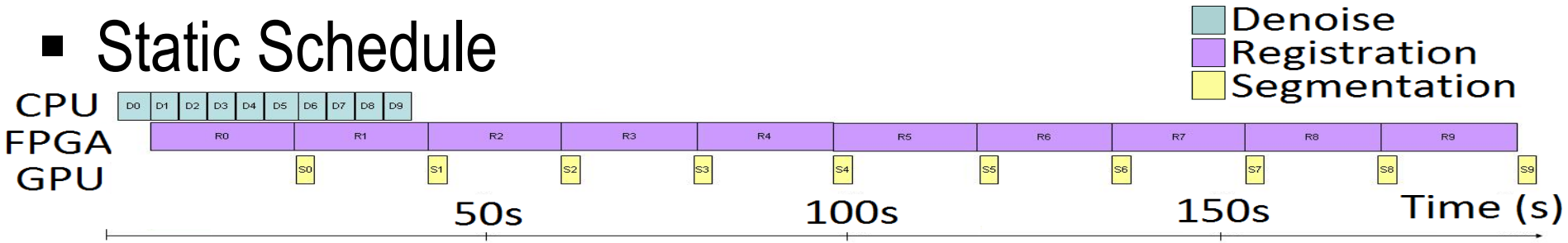
Convey HC-1ex Testbed



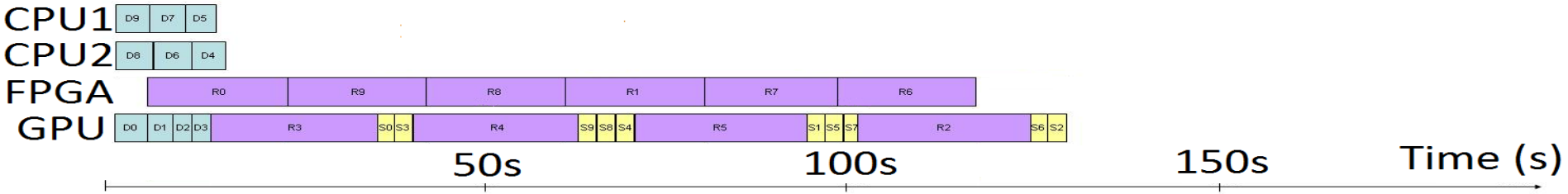
Static vs Dynamic Scheduling

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

- Static Schedule

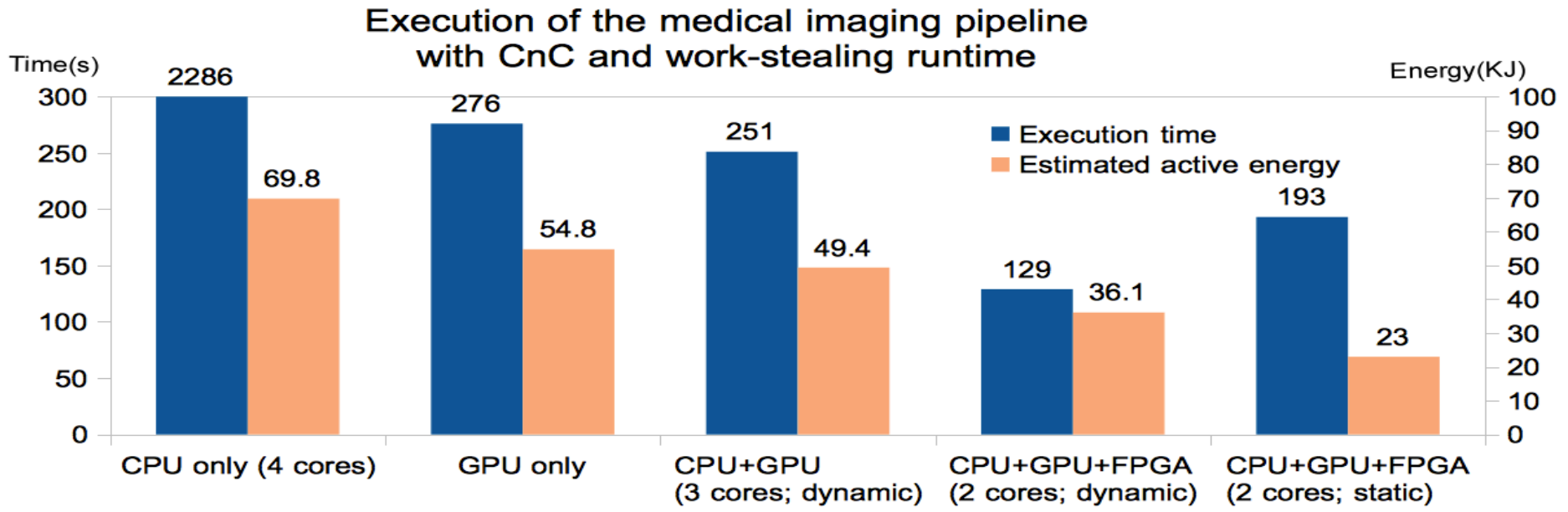


- Dynamic Schedule



Experimental Results for Medical Imaging Workload

- Execution times and active energy for dynamic vs. static scheduling on heterogeneous processors



Outline: Examples of Programming Models based on the Habanero Execution Model

1. Portable Heterogeneous Intra-Node Parallelism using a Data Flow Graph Language (DFGL)
2. Exploiting Heterogeneous Inter-Node Parallelism with Habanero-C++

RICE

Motivation: enable PGAS developers to leverage productivity benefits of new C++ features

C+11 lambda expressions

```
// create lambda
auto func =
   [ capture_list ]
   (formal_params) { … };
. . .
// execute lambda
func (argument_list);
```

C++11 futures

```
// create async task w/ result
auto f = std::async(
                  <lambda-expr>);
. . .
// Retrieve result
// (wait if needed)
int result = f .get ();
```

Our Approach: HabaneroUPC++ Library (Example constructs)

Remote task creation

```
asyncAt ( destPlace,
    [capture_list] ( ) {
        Statements1;
    });
```

Asynchronous one-sided data movement

```
asyncCopy (src, dest, count, ddf);
```

Message-driven task activation

```
asyncAwait(ddf, capture_list] ( ) {
        Statements2;
 });
```

"HabaneroUPC++: A Compiler-free PGAS Library." V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, V. Sarkar, PGAS 2015.

LSMS example (MPI and Habanero-UPC++ versions)

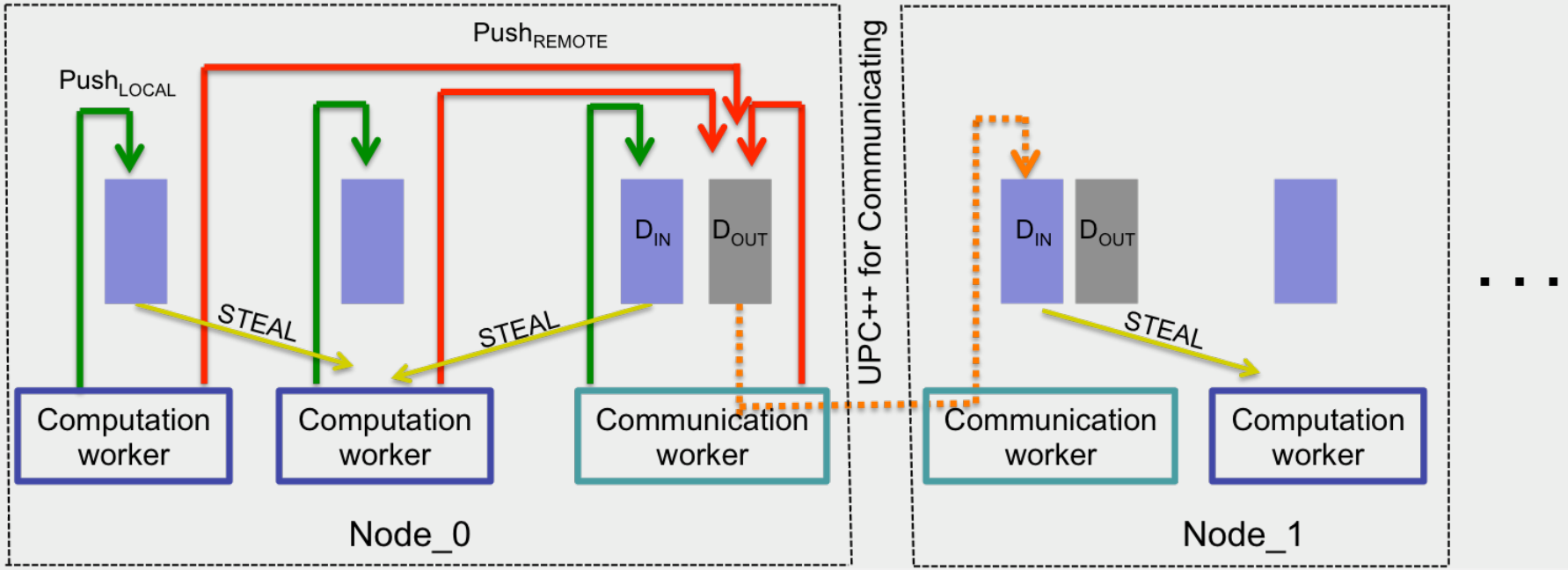
MPI version:

```
// Post MPI_IRecv() calls
. . .
// Post MPI_ISend() calls
. . .
// Perform all MPI_Wait()
// calls
. . .
// Perform tasks
. . .
```

Habanero-UPC++ version:

```
// Issue one-sided
// asyncCopy() calls
. . .
// Issue data-driven tasks
// in any order without any
// wait/barrier operations
hcpp::asyncAwait(
      result1, result2,
      [=]() { task body });
```

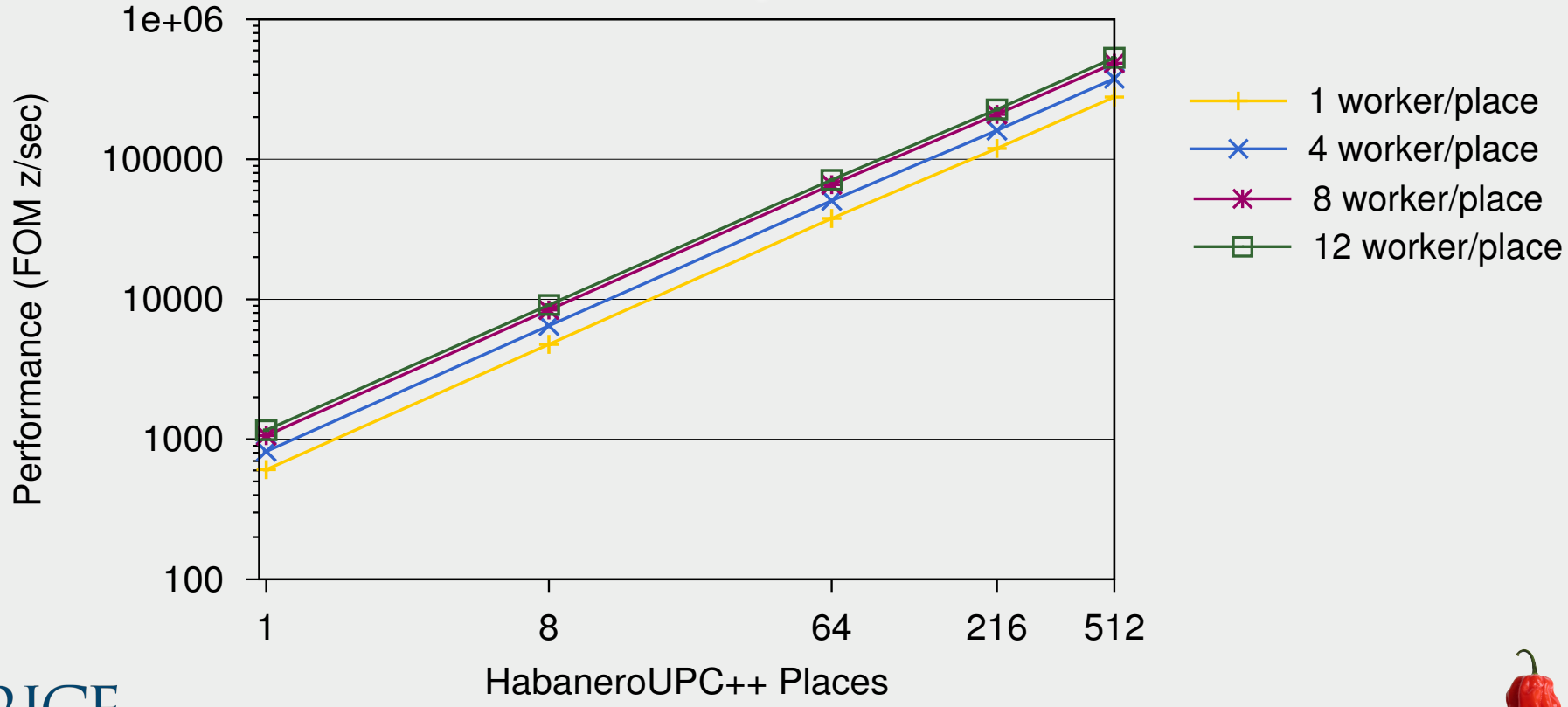

Habanero-UPC++ is enabled by tight integration of task and communication runtimes



This integration has been demonstrated separately for GASNet and MPI --- motivation for use of Open Community Runtime (OCR) as a common interface for different communication libraries

RICE

Weak Scaling Result for Habanero-UPC++ version of LULESH on NERSC Edison system

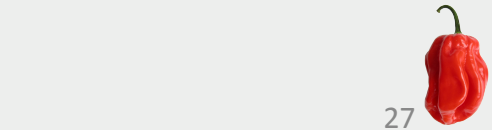


RICE

Summary: Extreme Scale Challenges for Applications

- Goal: forward-scalable and portable expression of parallelism, locality, data movement, and resilience
- Our premise: fundamental advances in programming models, compilers, and runtimes are necessary to achieve this goal
  - dynamic asynchronous parallelism
  - asynchronous data movement
  - distributed global address/name space
  - hierarchical abstractions of locality
  - support for failure recovery
  - . . .

RICE



27