



# Acknowledgments --- Habanero Team

- Faculty
  - Vivek Sarkar
- Senior Research Scientist
  - Michael Burke
- Research Scientists
  - Zoran Budimlić, Philippe Charles, Jun Shirako, Jisheng Zhao
- Research Programmer
  - Vincent Cavé
- Postdoctoral Researchers
  - Akihiro Hayashi
- PhD Students
  - Kumud Bhandari, Shams Imam, Deepak Majeti, Alina Sbîrlea, Dragoş Sbîrlea, Kamal Sharma, Rishi Surendran, Saĝnak Taşırılar, Nick Vrvilo, Yunming Zhang
- Undergraduate Students
  - Kyle Kurihara, Bing Xue
- Supported in part by the National Science Foundation, and by the X-Stack program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR).



# Multicore Processors and 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*
  - *Subject to frequent faults and failures*
- Many Classes of Extreme Scale Systems



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



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



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



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

## Key Challenges

- **Energy Efficiency**
- **Concurrency**
- **Resiliency**

## References:

- DARPA Exascale Software study, V. Sarkar et al, Sep 2009
- “Software Challenges in Extreme Scale Systems”. V. Sarkar, W. Harrod, A.E. Snively. SciDAC Review, January 2010.



# Brief Summary of Exascale Software Study

- **Area 1: Application Development**

- Challenge: need for  $\sim 1000x$  more concurrency in applications with  $\sim 0.01 - 0.1$  bytes/ops, and significant reductions in data movement

→ Focus of US DOE co-design centers **Opportunities for OpenSHMEM**

- **Area 2: Expressing Parallelism and Locality (Programmability)**

- Challenge: forward-scalable and portable expression of intrinsic parallelism and locality

→ Focus of US DOE X-Stack programs

- **Area 3: Managing Parallelism and Locality (Performance)**

- Challenge: integration of compilers, runtime, OS with auto-tuning

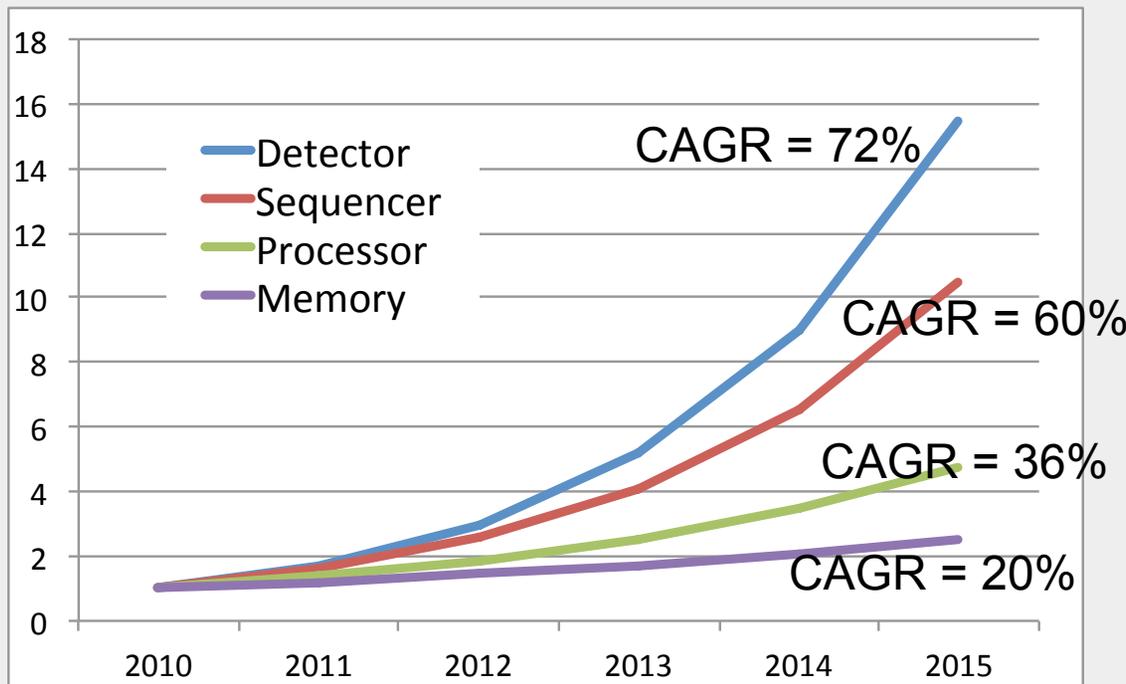
→ Focus of US DOE X-Stack and OS/R programs

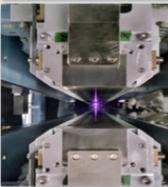
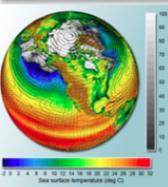
- See report for details!



# Data Challenges in Science

Overall trend: most science domains will become data-intensive in the exascale timeframe (and many well before then)



	<b>Genomics</b> Data Volume increases to 10 PB in FY21
	<b>High Energy Physics (Large Hadron Collider)</b> 15 PB of data/year
	<b>Light Sources</b> Approximately 300 TB/day
	<b>Climate</b> Data expected to be hundreds of 100 EB

Reference: DOE report on Synergistic Challenges in Data-Intensive Science and Exascale Computing, March 2013.



# Brief Summary of DOE ASCAC Data Subcommittee Report

## ■ Findings

- Data analysis needs extreme scale computing and will be subject to extreme scale challenges (moving compute to data, etc.)
- Integration of data analytics with exascale simulations represents a new class of workloads for exascale computing
- Urgent need to simplify the workflow for data-intensive science
- Urgent need to increase pool of scientists trained in both exascale and data-intensive computing

## ■ Recommendations

- Give higher priority to investments that can benefit both data-intensive science and exascale computing
- Give higher priority to investments that simplify the workflow for data-intensive science
- Adjust funding to increase pool of scientists trained in both exascale and data-intensive computing



■ See report for details!



# What is “Hybrid Programming”?



Zonkey



Liger



Jaglion



## Observation: definition of “Hybrid” depends on your starting point

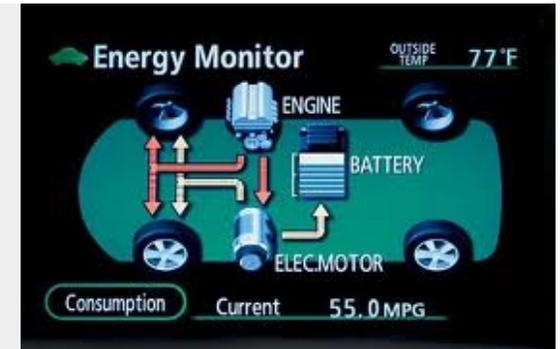
- If your starting point is a bulk-synchronous SPMD program with one thread per rank, then “hybridizations” have to be implemented as special-case extensions, e.g.,
  - Asynchronous data movements across ranks
  - Task parallelism within a rank
  - Accelerator parallelism
  - Task/process cancellation and migration
  - . . .



# Another Metaphor for Hybrid Programming Today ....



## Alternate Approach: Hybrid by Design



- If your starting point is a general unified execution model and runtime system for extreme scale computing, then “hybridizations” are simply combinations of features, e.g.,
  - Asynchronous data movements across ranks
  - Task parallelism within a rank
  - Accelerator parallelism
  - Task/process cancellation and migration
  - . . .



# Rice Habanero Multicore Software Project: Enabling Technologies for Extreme Scale

## Parallel Applications

### Portable 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 synchronization: *phasers*

Habanero  
Programming  
Languages

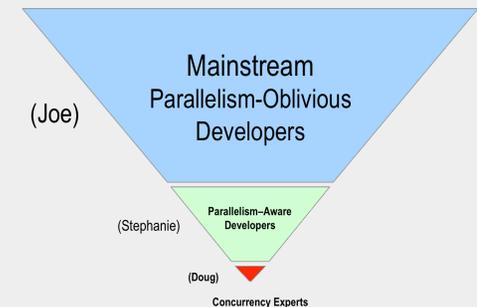
Habanero Static  
Compiler &  
Parallel  
Intermediate  
Representation

Habanero  
Runtime  
System

### Two-level programming model

Declarative Coordination  
Language for Domain Experts:  
CnC-HC, CnC-Java, CnC-Python,  
CnC-Matlab, ... +

Task-Parallel Languages for  
Parallelism-aware Developers:  
Habanero-C, Habanero-Java,  
Habanero-Scala



## Extreme Scale Platforms



## Performance Variability is on the rise ...

- 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, ...)

***Increasing performance variability →***

- ***Need for asynchrony and reduction of ordering constraints***
- ***Runtime becomes an increasingly critical component of software stack***



## Role of Runtime Systems

- Inherent variability in irregular applications and extreme scale platforms calls for a runtime system that is
  - abstract
  - asynchronous
  - adaptive
  - portable
  - a true manifestation of future execution models



# Target Platforms

Habanero programs have been executed on a wide range of production and experimental systems

- Multicore SMPs (IBM, Intel)
- Discrete GPUs (AMD, NVIDIA)
- Integrated GPUs (AMD, Intel)
- FPGA (Convey, w/ GPU added)
- HPC Clusters
- Hadoop Clusters
- Experimental processors: IBM Cyclops, Intel SCC
- . . . .



# Elements of 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*

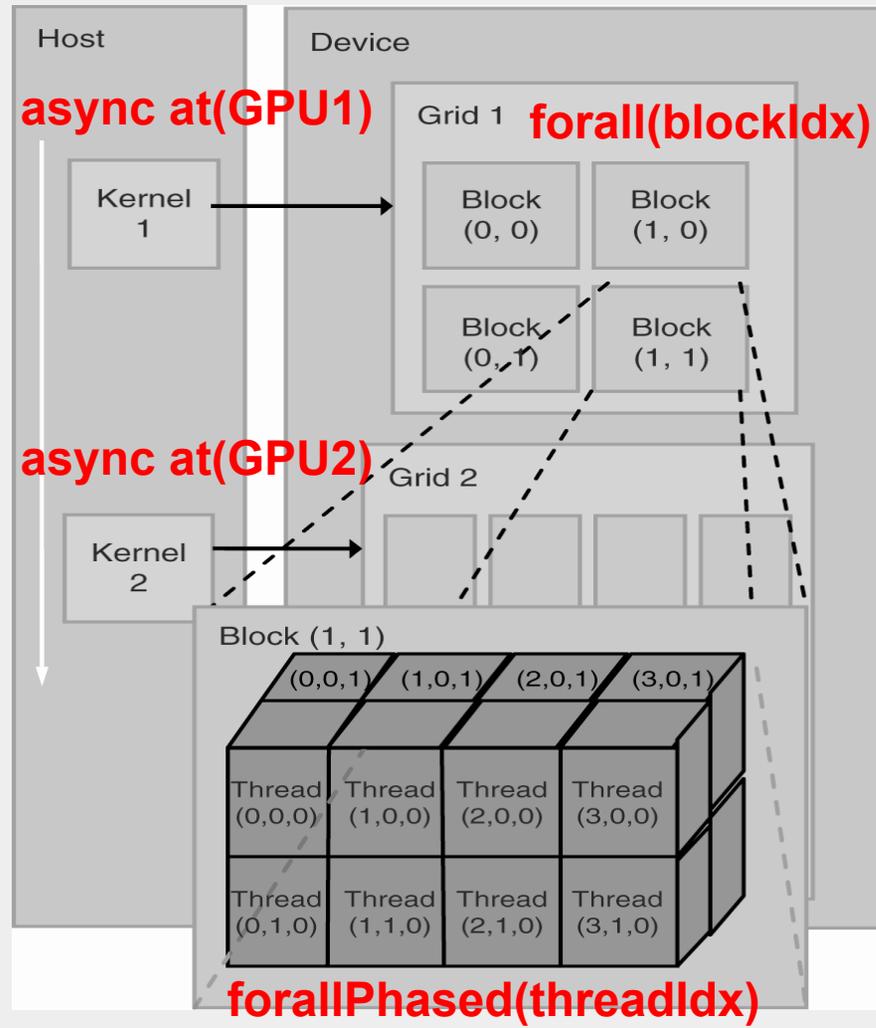
## 3) Inter-task synchronization operations

- Mutual exclusion: *global/object-based isolation, actors*
- Collective and point-to-point synchronization: *phasers*

***Goal: unified model of parallelism that spans a wide range of extreme scale platforms***



# Example: Habanero abstraction of a CUDA kernel invocation

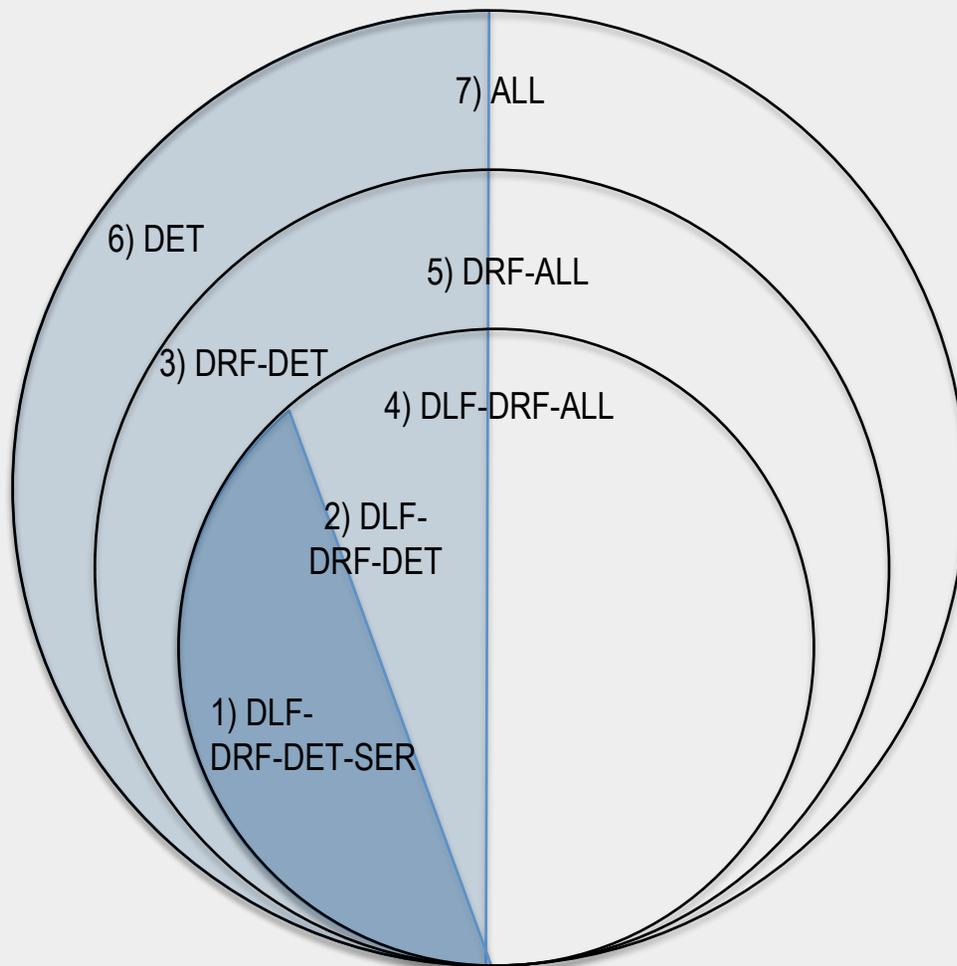


# Properties of Habanero Execution Model

- Deadlock freedom guarantee for large subset of operations
  - All operations except explicit wait in phasers and explicit await clause in async
- Data-race freedom guarantee for subset of data accesses
  - Future values, accumulator values
  - Read-write permission regions
  - Isolated accesses, actors
- Determinacy guarantee for subset of programs
  - Data-race freedom implies determinacy for all programs that do not use mutual exclusion constructs (isolated, actors)
- Amenable to efficient asynchronous and portable implementations
  - Locality-aware work-stealing
  - Hierarchical places with support for heterogeneous processors
  - Integration with cluster-level communication runtime systems
  - Scalable synchronization with phasers and delegated isolation
  - Compiler optimizations for structured parallelism



# Semantic Classification of Habanero Parallel Programs



- Legend
  - DLF = DeadLock-Free
  - DRF = Data-Race-Free
  - DET = Determinate
  - DRF  $\rightarrow$  DET = DRF implies DET
  - SER = Serializable
- If a Habanero program only uses *async*, *finish*, and *future* constructs (no mutual exclusion), then it is guaranteed to belong to the DLF + DRF  $\rightarrow$  DET + SER class
- Adding *phasers* yields programs in the DLF + DRF  $\rightarrow$  DET class
- Adding *async await* yields programs in the DRF  $\rightarrow$  DET class
- Restricting shared data accesses to *futures*, *isolated*, *actors* yields programs in the DRF-ALL class



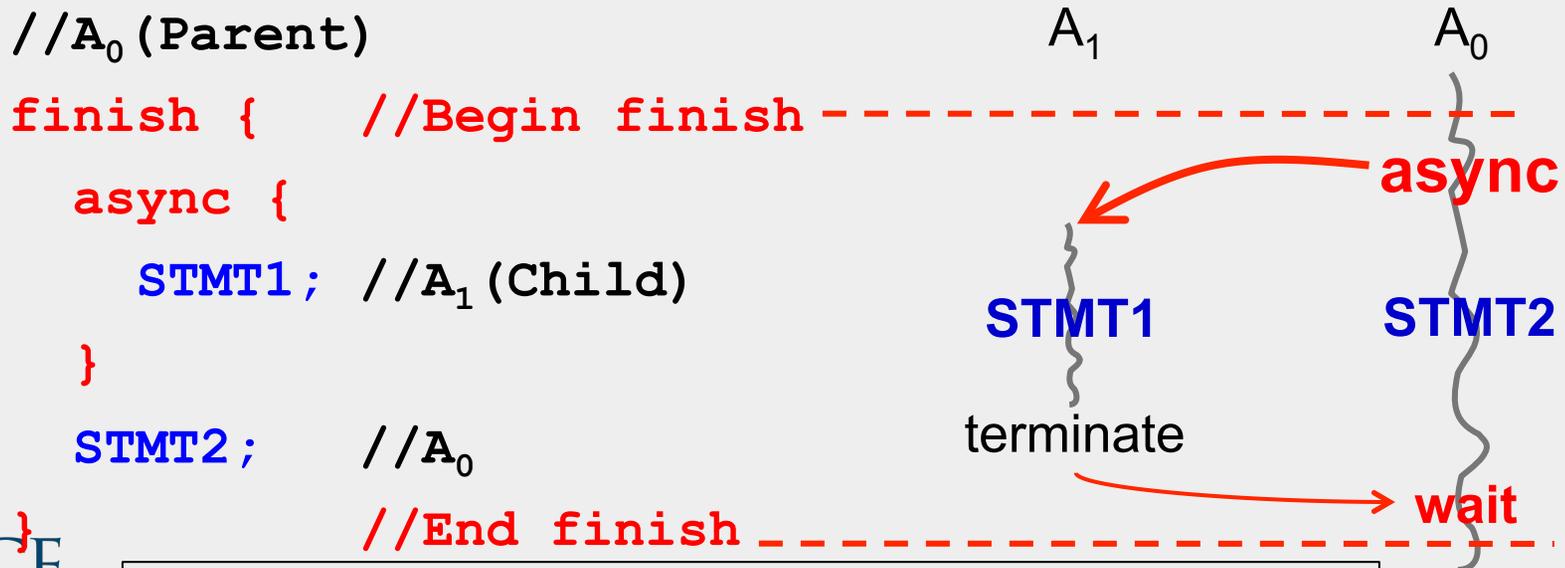
# 1) Primitives for Lightweight Asynchronous Tasks

## async S

- Creates a new child task that executes statement S
- Parent task moves on to statement following the async
- *async can be a computation or a communication task*

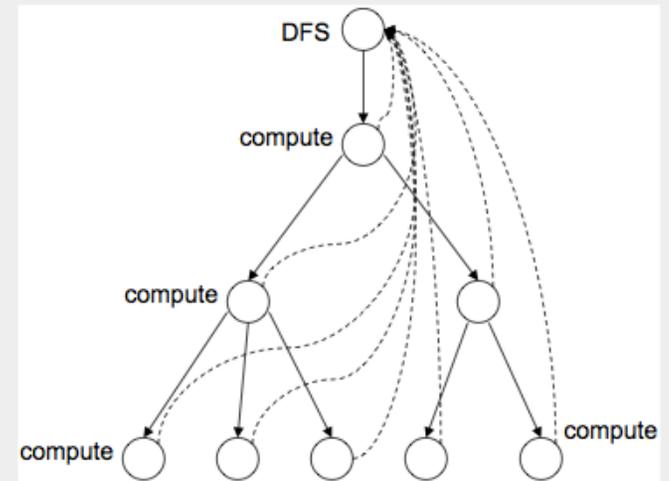
## finish S

- Execute S, but wait until *all* (transitively) spawned asyncs in S's scope have terminated
  - Like OpenMP's taskwait
- Implicit finish between start and end of main program
- Use of finish cannot create a deadlock cycle



# Parallel Spanning Tree Algorithm in Habanero-C

```
1. int try_labeling(Node* node, Node* parent){
2.     /* If node->parent is null, set it to parent */
3.     return CAS(
4.         (void * volatile *) &(node->parent),
5.         (void *) NULL, (void *) parent);
6. }
7. void compute(Node* node){
8.     int i;
9.     for(i = 0; i < node->num_neighbors; i++) {
10.        Node* child = node->neighbors[i];
11.        if(try_labeling(child, node)){
12.            async {compute(child);};
13.        }
14.    }
15.    /* function can return before async's complete */
16. }
17. finish compute(root);
```



—————→  
Async edge

.....→  
Finish edge

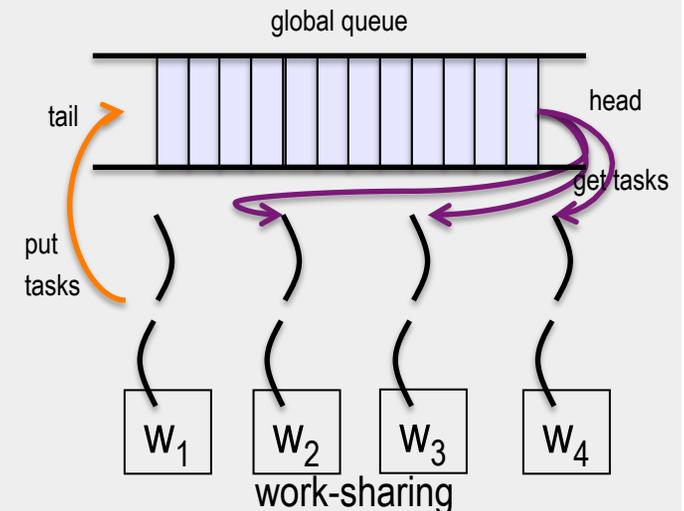
Above synchronization structure  
cannot be expressed in Cilk or  
OpenMP



# Runtime Schedulers for Async-Finish Task Parallelism

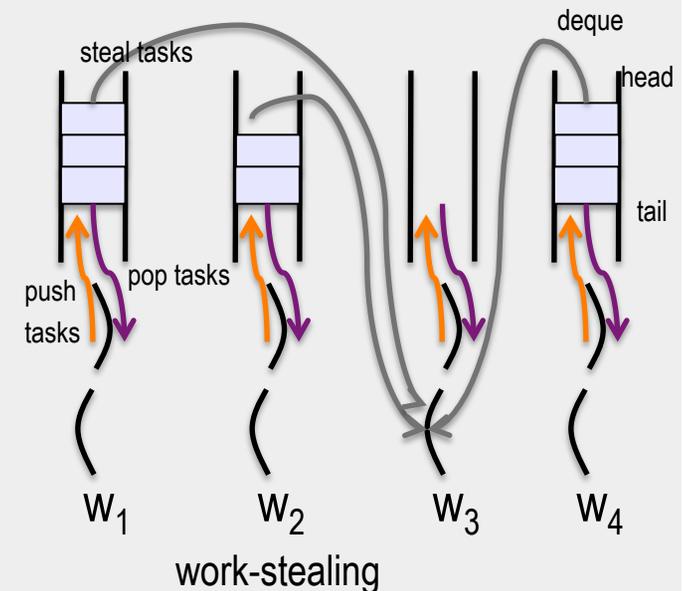
## Work-Sharing (Java ThreadPoolExecutor, OpenMP, ...)

- Busy worker pushes task at one end of global deque
- Access to global deque needs to be synchronized



## Work-Stealing (Cilk, TBB, Java ForkJoin, ...)

- One deque per worker (better scalability)
- Idle worker steals tasks from busy workers
- Two scheduling policies of interest
  - **Work-first policy:** worker executes child task eagerly and leaves continuation to be stolen
  - **Help-first policy:** worker pushes child task to be stolen (asks for help) and executes continuation
- **Hybrid adaptive algorithm** dynamically selects best policy for each async instance

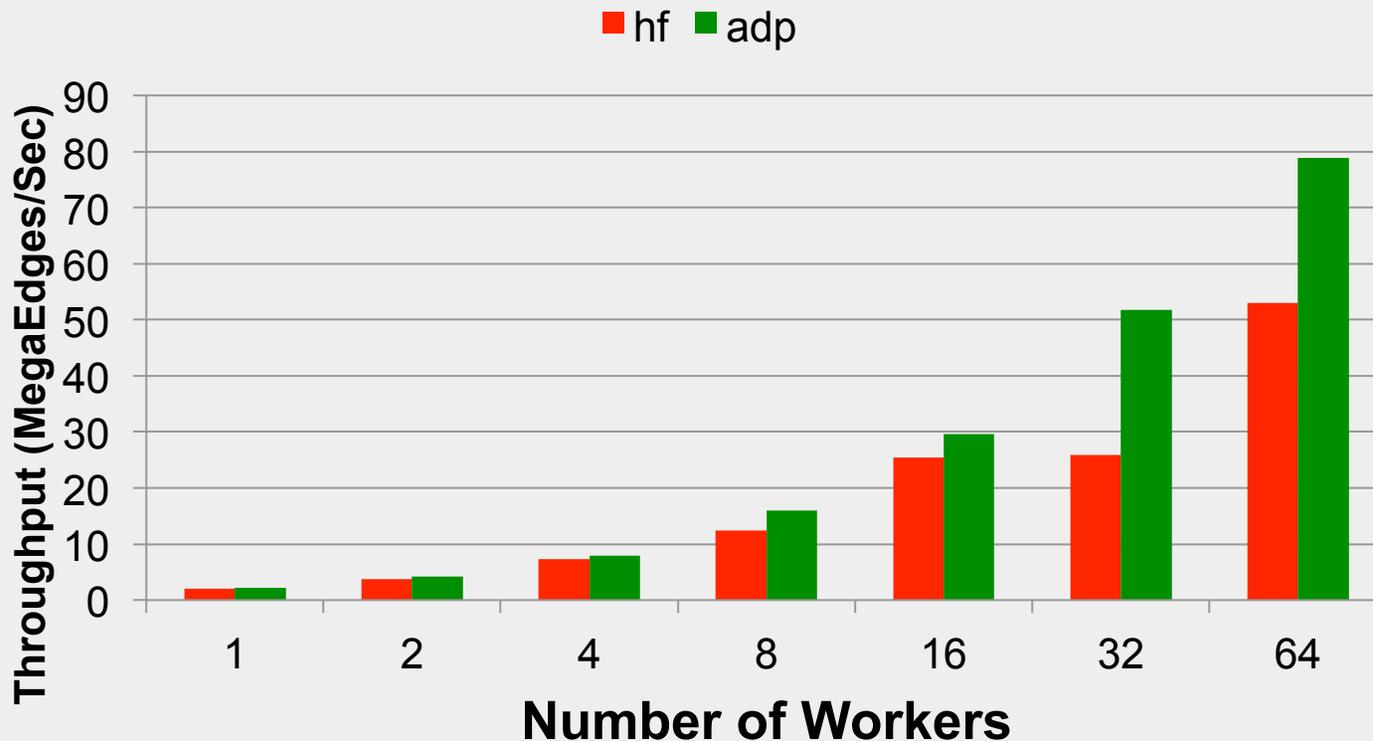


- "Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs", Y.Guo, R.Barik, R.Raman, V.Sarkar, IPDPS 2009.
- "SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Systems", Y.Guo, J.Zhao, V.Cave, V.Sarkar, IPDPS 2010.



# Parallel Spanning Tree on a Torus Graph with 4M vertices (Habanero-Java implementation)

## PDFS – Niagara 2



Work-first policy is unable to complete due to stack overflow  
Adaptive (adp) policy performs better than help-first policy



# Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs) in Habanero-C language

```
DDF_t* ddfA = DDF_CREATE();
```

- Allocate an instance of a data-driven-future object (container)

```
async AWAIT(ddfA, ddfB, ...) <Stmt>
```

- Create a new data-driven-task to start executing **Stmt** after all of **ddfA**, **ddfB**, ... become available (i.e., after task becomes “enabled”)

```
DDF_PUT(ddfA, V);
```

- Store object **V** in **ddfA**, thereby making **ddfA** available
- Single-assignment rule: at most one put is permitted on a given DDF

```
DDF_GET (ddfA)
```

- Return value stored in **ddfA**
- No blocking needed --- should only be performed by tasks that contain **ddfA** in their **AWAIT** clause, or when some other synchronization (e.g., **finish**) guarantees that **DDF\_PUT** must have been performed.

DDFs and DDTs can be more efficient than OpenMP regions and barriers

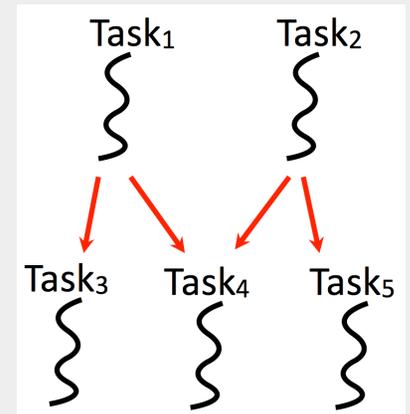


# Example Habanero-C code fragment with Data-Driven Futures (Dag Parallelism)

```
1. DDF_t* left = DDF_CREATE();
2. DDF_t* right = DDF_CREATE();
3. finish {
4.   async AWAIT(left) leftReader(DDF_GET(left)); // Task3
5.   async AWAIT(right) rightReader(DDF_GET(right)); // Task5
6.   async AWAIT(left,right) // Task4
7.     bothReader(DDF_GET(left), DDF_GET(right));
8.   async DDF_PUT(left,leftWriter()); //Task1
9.   async DDF_PUT(right,rightWriter()); //Task2
10. }
```

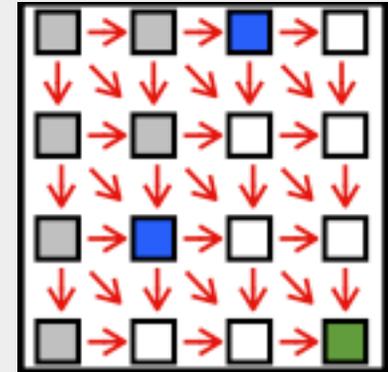
**AWAIT** clauses capture data flow relationships

This example cannot be expressed in OpenMP either



# Smith Waterman example with DDFs (Habanero-C)

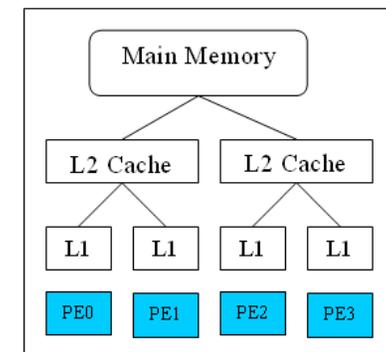
```
finish { // matrix is a 2-D array of DDFs
  for (i=0,i<H;++i) {
    for (j=0,j<W;++j) {
      DDF_t* curr = matrix[i][j];
      DDF_t* above = matrix[i-1][j];
      DDF_t* left = matrix[i][j-1];
      DDF_t* uLeft = matrix[i-1][j-1];
      async AWAIT (above, left, uLeft){
        Elem* currElem =
          init(DDF_GET(above),DDF_GET(left), DDF_GET(uLeft));
        compute(currElem);
        DDF_PUT(curr, currElem);
      }/*async*/
    }/*for-j*/
  }/*for-i*/
}/*finish*/
```



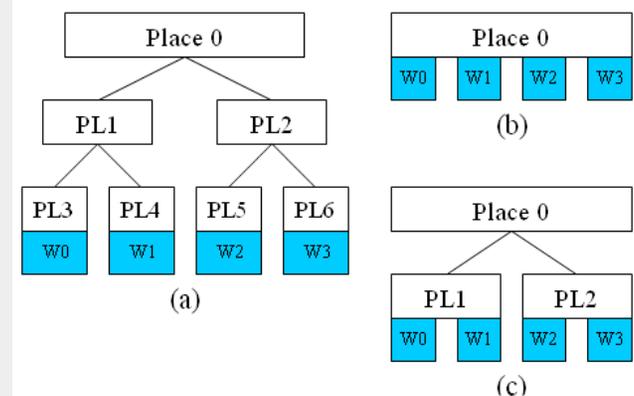
## 2) Locality control for task and data distribution: Hierarchical Place Trees (HPT) abstraction

- HPT approach
  - Hierarchical memory + Dynamic parallelism
- Place denotes affinity group at memory hierarchy level
  - L1 cache, L2 cache, CPU memory, GPU memory
- Leaf places include worker threads
  - e.g., W0, W1, W2, W3
- Explore multiple HPT configurations
  - For same hardware and application
  - Trade-off between locality and load-balance

“Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement”, Y.Yan et al, LCPC 2009

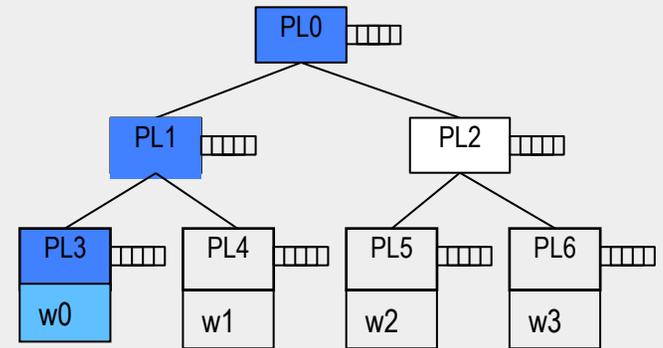


A Quad-core workstation

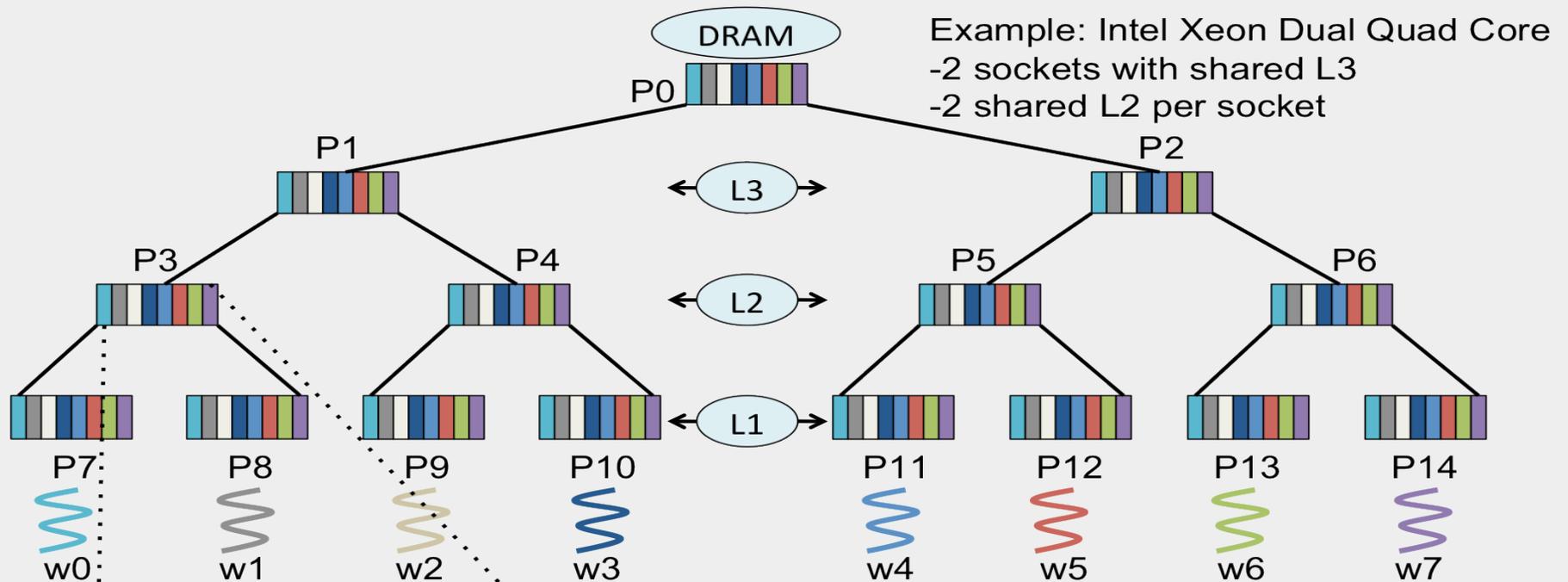


# Locality-aware Scheduling using the HPT

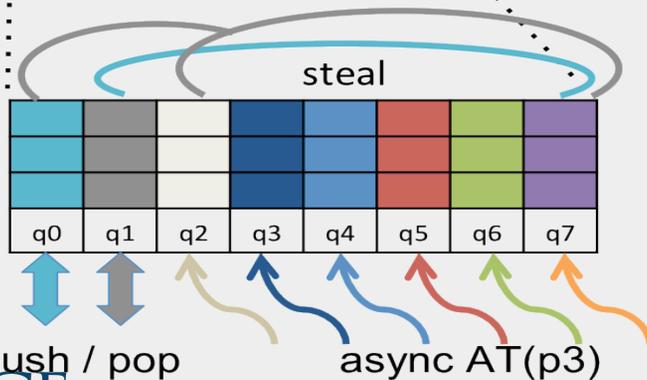
- Workers attached to leaf places
  - Bind to hardware core
- Each place has a queue
  - async at(<pl> <stmt>**: push task onto place *pl*'s queue
- A worker executes tasks from ancestor places from bottom-up
  - W0 executes tasks from PL3, PL1, PL0
- Tasks in a place queue can be executed by all workers in the place's subtree
  - Task in PL2 can be executed by workers W2 or W3



# HPT Implementation in Habanero-C



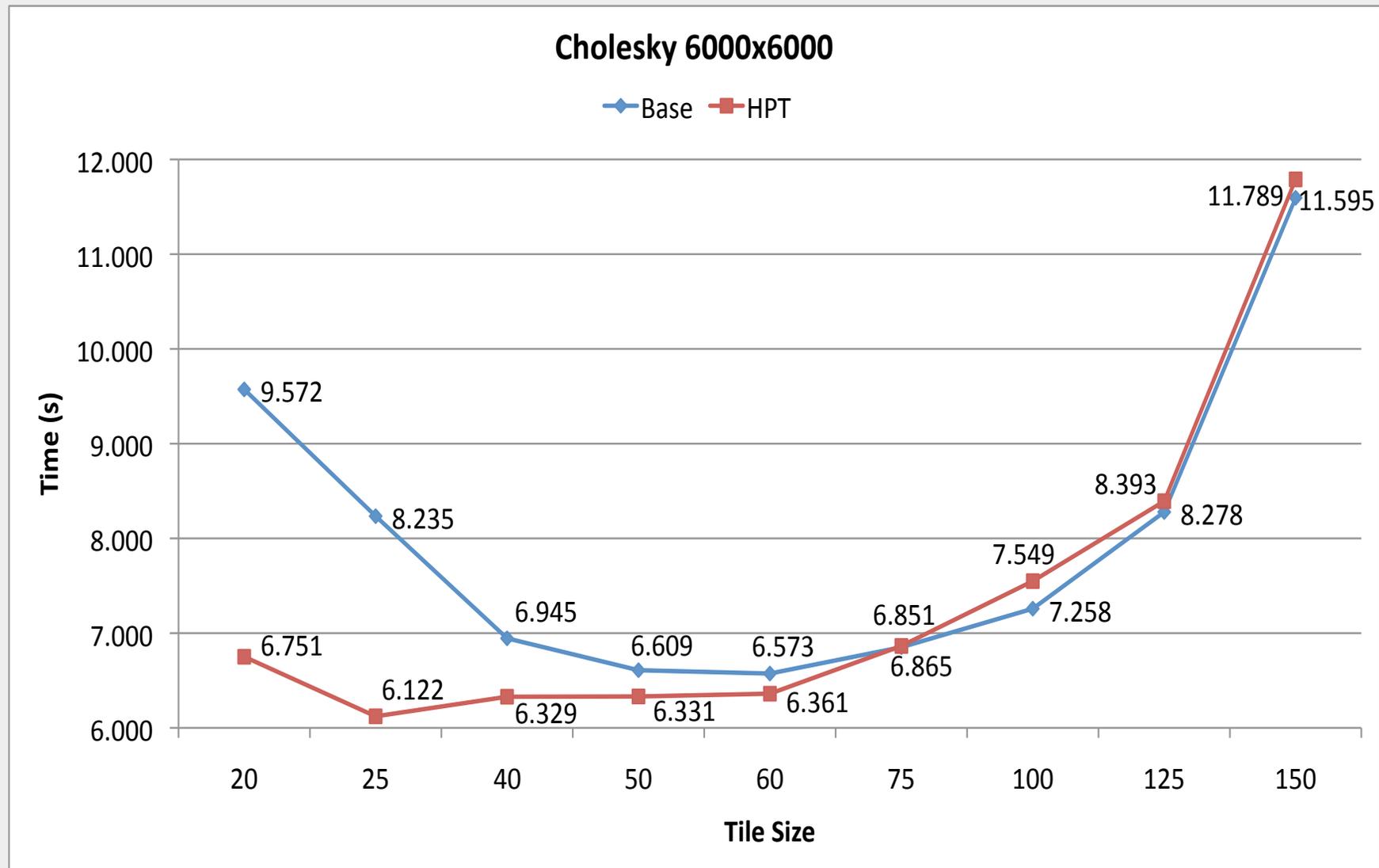
Example: Intel Xeon Dual Quad Core  
 -2 sockets with shared L3  
 -2 shared L2 per socket



- Each place has one queue per worker
- Ensures non-synchronized push and pop
- Workers attach to (own) leaf places
- Any worker can push a task at any place
- Pop / steal access permitted to subtree workers
- Workers traverse path from leaf to root
- Tries to pop, then steal, at every place
- After successful pop / steal worker returns to leaf
- Worker threads are bound to cores



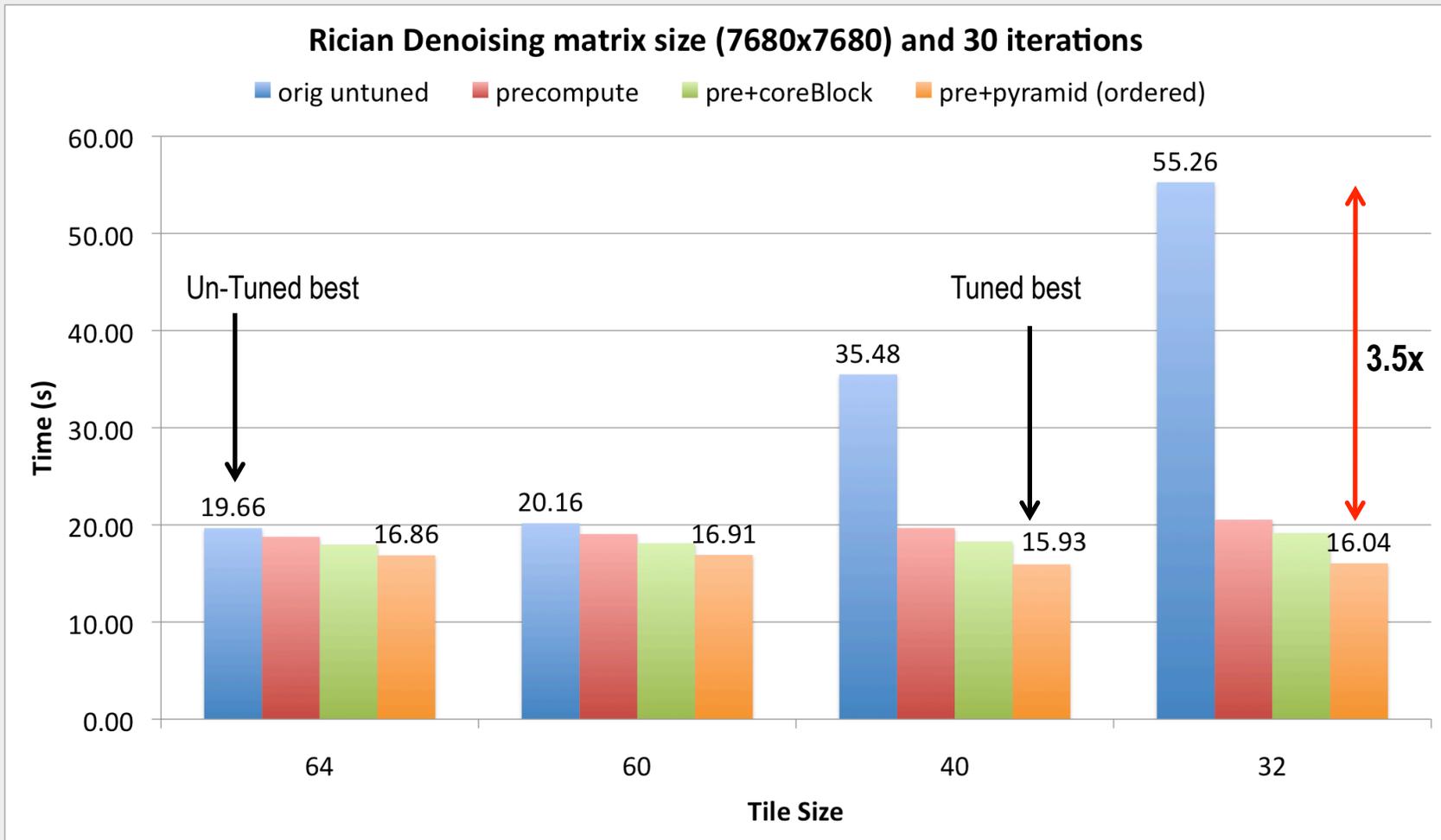
# Example: Cholesky Performance with HPT (12-core SMP)



Reference: *Runtime Systems for Extreme Scale Platforms.*  
Sanjay Chatterjee. Ph.D Thesis, December 2013



# Preliminary Results with Affinity-based Scheduling using Tuning Annotations --- Rician Denoising

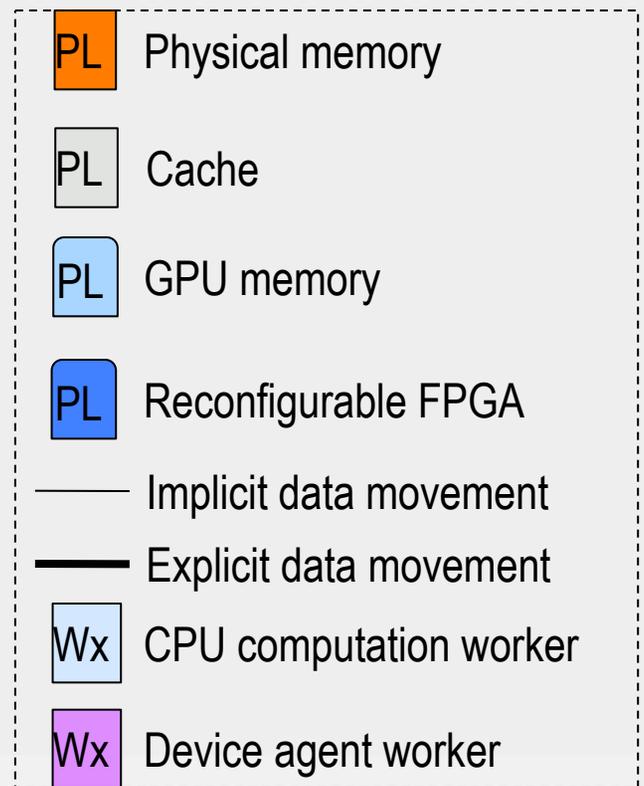
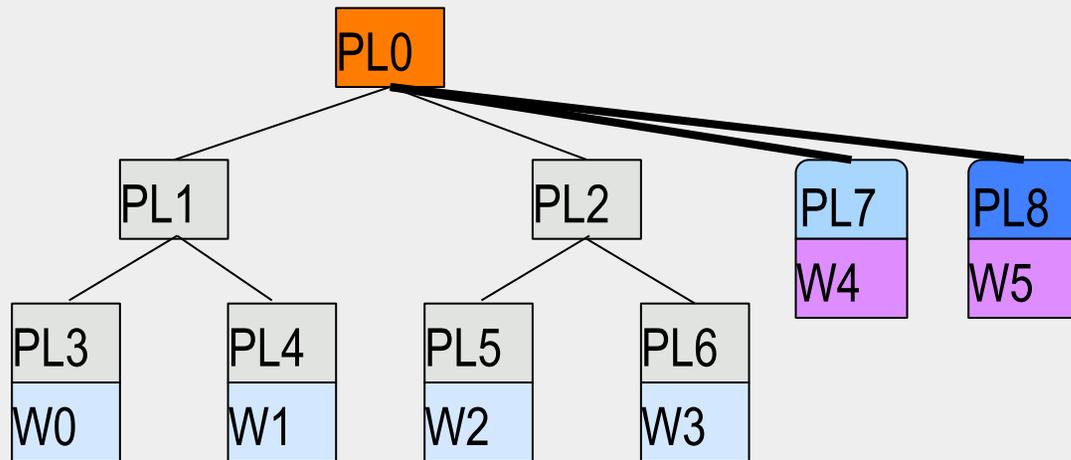


Up to 3.5x improvement on specific tile size

Overall improvement of 19% over best optimal tiled version



# Habanero Hierarchical Place Trees for heterogeneous architectures and accelerators



- **Devices (GPU or FPGA) are represented as memory module places and agent workers**
  - GPU memory configuration are fixed, while FPGA memory are reconfigurable at runtime
- **async at(P) S**
  - Creates new activity to execute statement S at place P
- **Explicit data transfer between main memory and device memory when needed**
  - Use of copyin/copyout clauses to improve programmability of data transfers
- **Device agent workers**
  - Perform asynchronous data copy and task launching for device



# Habanero-C constructs for Heterogeneous Computing

**finish**{ body }

- Ensures all tasks spawned inside the *body* are completed.

**async copyin**(args1) **copyout**(args2) **at**(dev-list) { Body };

- Asynchronously copy data between the device and the host.

**forasync point** (args) **range** (<low:high>) **seq** (args) **scratchpad**(args)  
**reduce** (args) **at**(dev-list) **partition**(ratio){ Body }

- Data/Task parallel loop.

**phased next**;

- Point to point synchronization support.

**single** { Body };

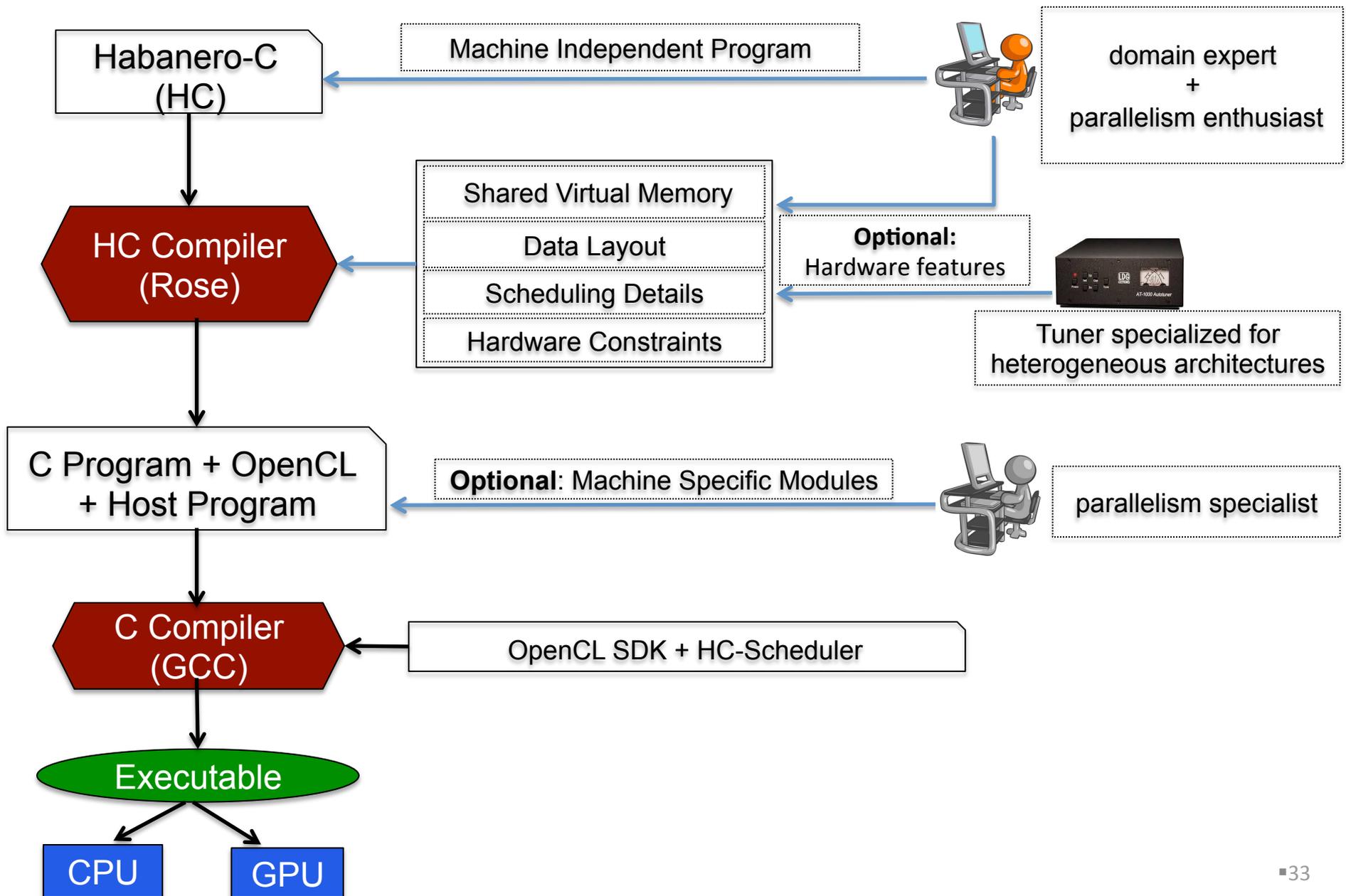
- *Body* is executed only once.

**await** (events);

- Wait until specified events are completed.



# Habanero-C Compilation Framework



# HC to OpenCL Code Generation

```
int main(){
    finish async copyin(b, c) at(gpu);
    finish forasync point(i, j) range(0:M, 0:N) at(gpu)
    {
        a[i * M + j] = b[i * M + j] + c[ i * M + j];
    }
    finish async copyout(a) at(gpu);
}
```

HC Code  
2D Matrix Addition

Host Code

```
void offload(float *a, float *b, float *c,
            char *kernel_name, char* ocl_kernel, int dev)
{
    //Build and execute the kernel
    //..... More Code.....
    ind0 = get_buffer((unsigned long)C);
    ind1 = get_buffer((unsigned long)A);
    ind2 = get_buffer((unsigned long)B);
    err = clSetKernelArg(kernel[1][dev], 0, sizeof(cl_mem), &buffer[ind0].mem_buffer[dev_id]);
    err = clSetKernelArg(kernel[1][dev], 1, sizeof(cl_mem), &buffer[ind1].mem_buffer[dev_id]);
    err = clSetKernelArg(kernel[1][dev], 2, sizeof(cl_mem), &buffer[ind2].mem_buffer[dev_id]);
    err = clSetKernelArg(kernel[1][dev], 3, sizeof(cl_int), &NK);
    err = clSetKernelArg(kernel[1][dev], 4, sizeof(cl_int), &NJ);
    check_error("Failed to set kernel arguments");
    global[1]=high1; global[0]=high2;
    local[1]= seq1 local[0]= seq2;
    offset[1]=low1; offset[0]=low2;
    err = clEnqueueNDRangeKernel(commands[,kernel, 2, offset, global,local, 0, NULL,
    &Event);
    //..... More Code.....
    err= clGetPlatformInfo (platforms[ct], CL_PLATFORM_NAME, 1024, device_name, NULL);
    err = clGetDeviceIDs(platforms[ct], CL_DEVICE_TYPE_CPU, 4, device_id, &num_devices);
    //..... More Code.....
}
```

C Program

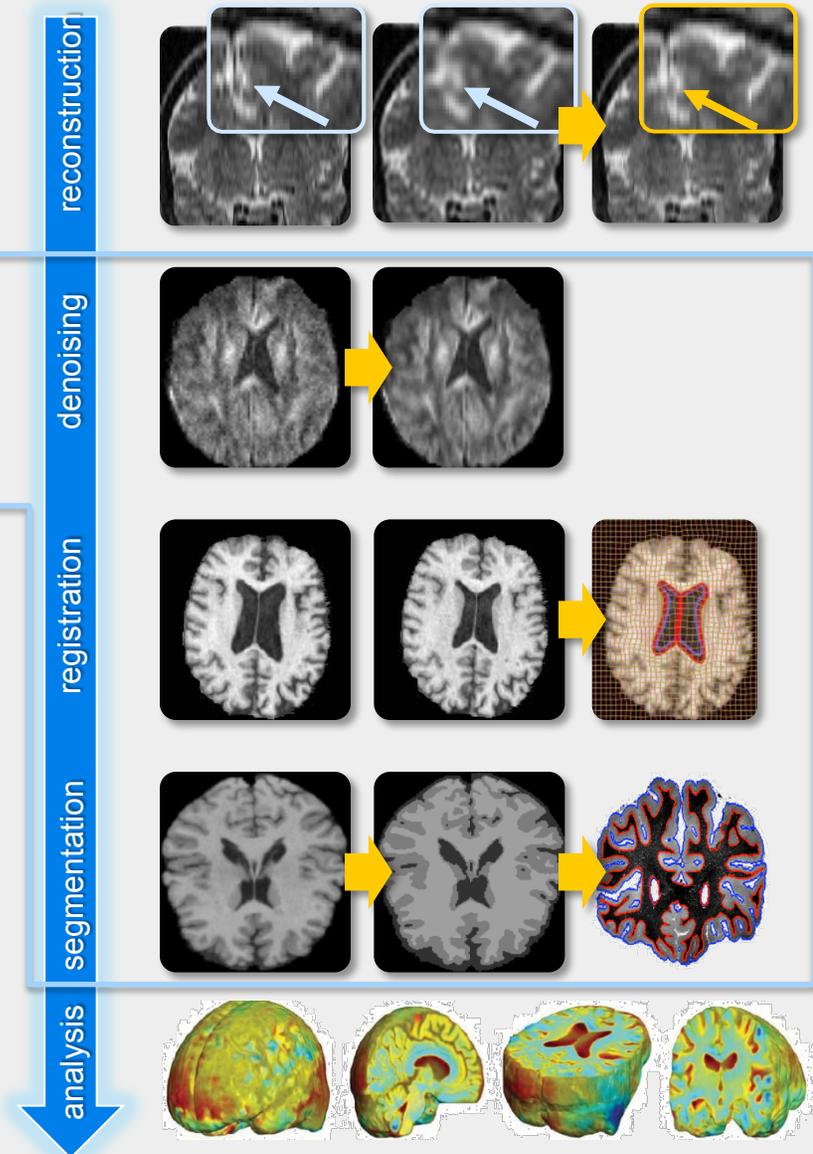
```
int main(){
    copy_to_device(b,gpu);
    copy_to_device(c,gpu);
    cl_finish(gpu);
    offload(a, b, c, "kernel_1", Kernel_string, gpu);
    cl_finish(gpu);
    copy_from_device(a,gpu);
    cl_finish(gpu);
}
```

OpenCL Kernel

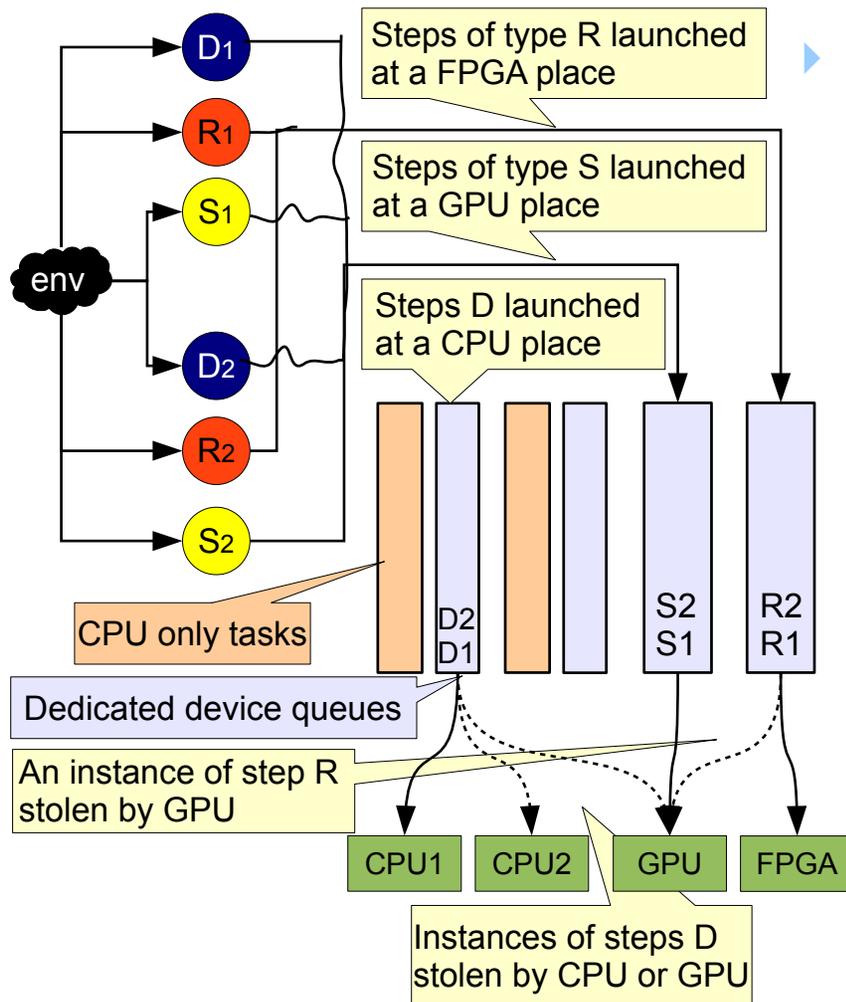
```
Kernel_string=
"void kernel_1(__global float *a, __global float *b, __global float *c, int M, int N) {
    i = get_global_id(1);
    j = get_global_id(0);
    a[i * M + j] = b[i * M + j] + c[ i * M + j];
}";
```

# Medical imaging application (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 improvement
  - power efficiency improvements
  - real-time clinical applications and simulations using patient imaging data



# Adding Affinity Annotations for Heterogeneous Computing

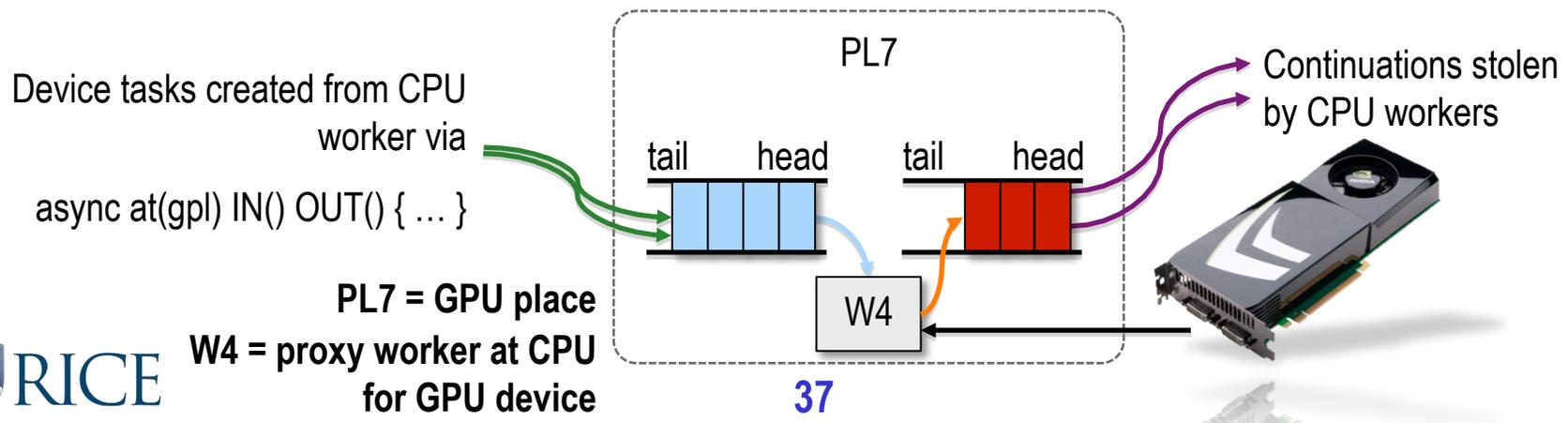


- ▶ CnC graph representation extended with **tag functions** and **affinity annotations**:
  - ▶  $\langle C \rangle :: (D @CPU=20, GPU=10);$
  - ▶  $\langle C \rangle :: (R @GPU=5, FPGA=10);$
  - ▶  $\langle C \rangle :: (S @GPU=12);$
- ▶  $[IN : k-1] \rightarrow (D : k) \rightarrow [IN2 : k+1];$
- ▶  $[IN2 : 2*k] \rightarrow (R : k) \rightarrow [IN3 : k/2];$
- ▶  $[IN3 : k] \rightarrow (S : k) \rightarrow [OUT : IN3[k)];$
- ▶  $env \rightarrow [IN : \{0 .. 9\}], \langle C : \{0 .. 9\} \rangle;$
- ▶  $[OUT : 1] \rightarrow env;$



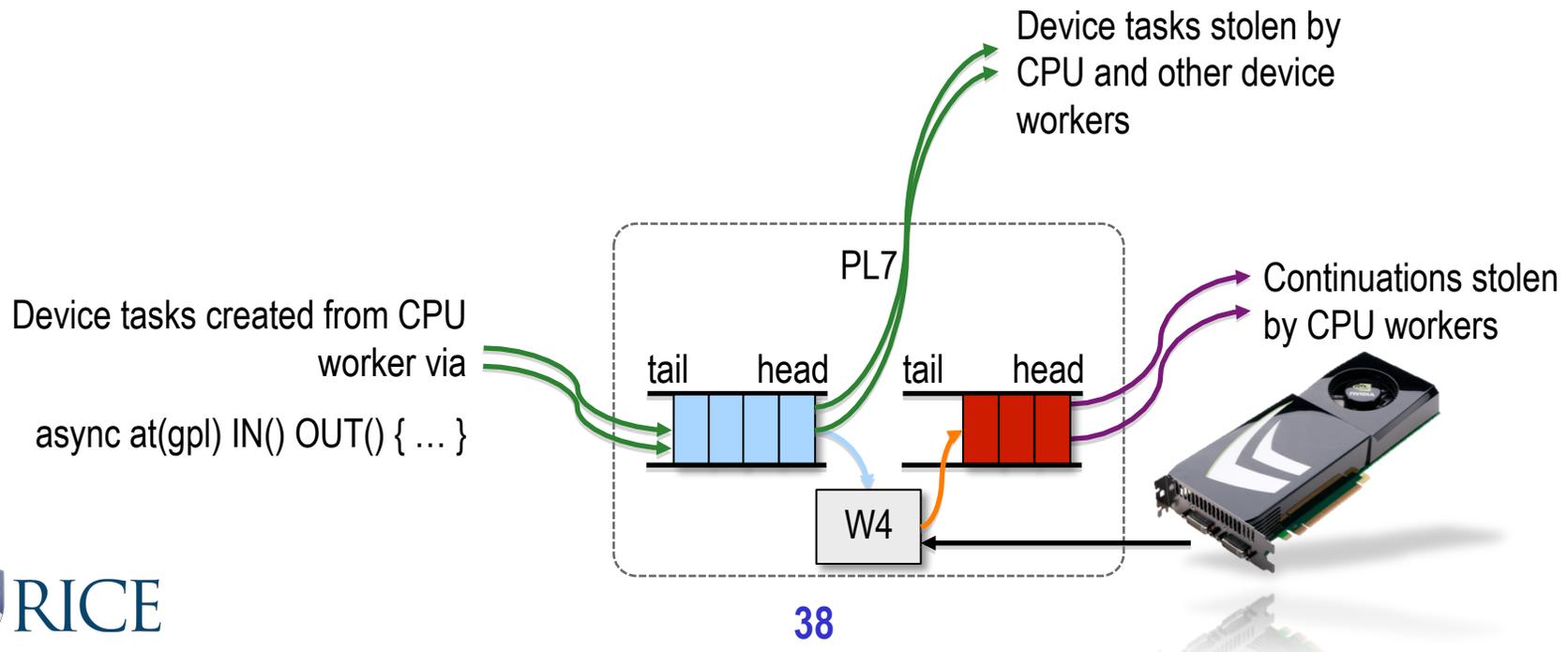
# Hybrid Scheduling for Heterogeneous Nodes

- ◆ Device place has two HC (half-concurrent) mailboxes: inbox (green) and outbox (red)
  - No locks – highly efficient
- ◆ Inbox maintains asynchronous device tasks (with IN/OUT)
  - Concurrent enqueueing device tasks by CPU workers from tail
  - Sequential dequeuing tasks by device “proxy” worker
- ◆ Outbox maintains continuation of the finish scope of tasks
  - Sequential enqueueing continuation by “proxy” worker
  - Concurrent dequeuing (steal) by CPU workers

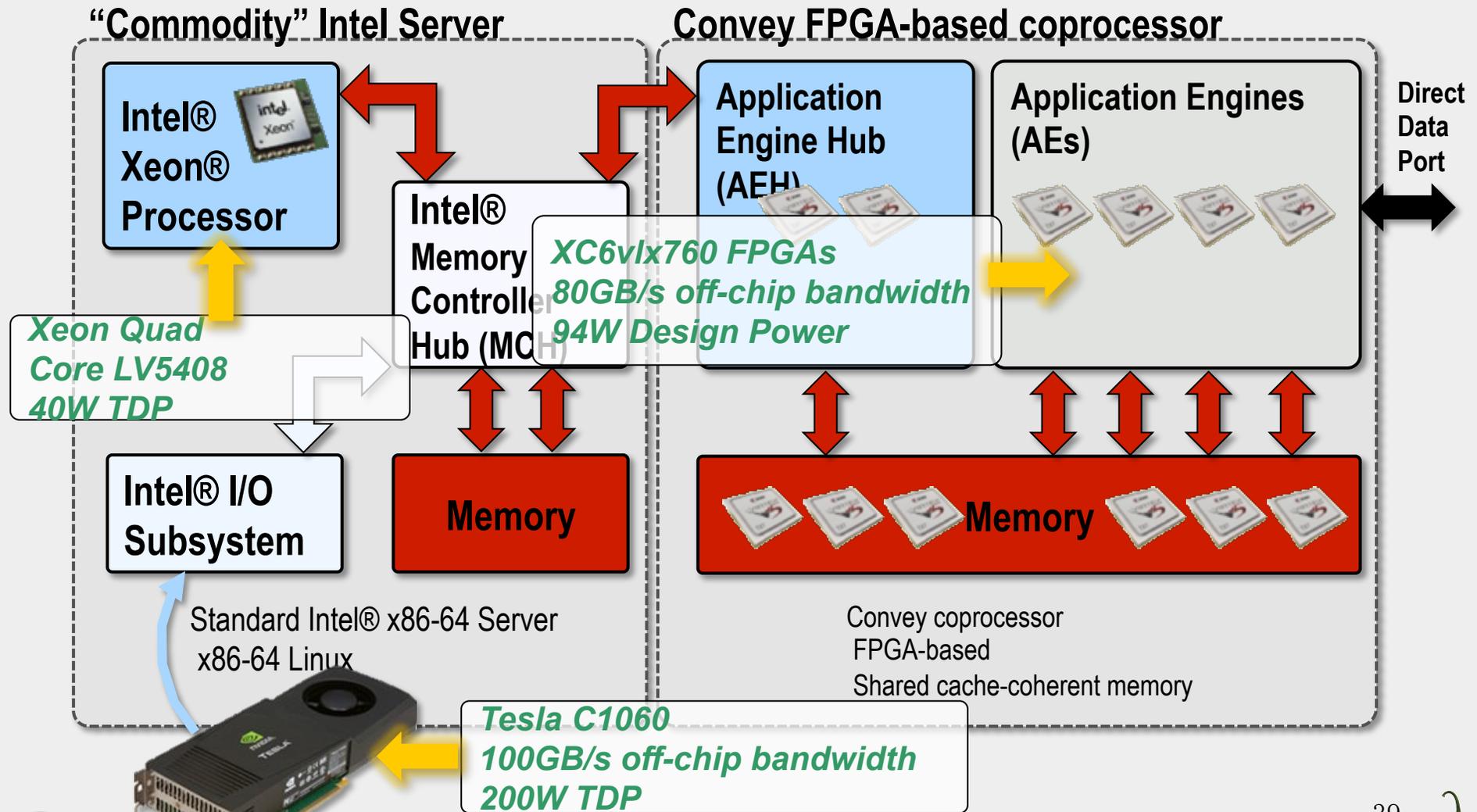


# Hybrid Scheduling with Cross-Platform Work Stealing

- ◆ Steps are compiled for execution on CPU, GPU or FPGA
  - Same-source multiple-target compilation in future
- ◆ Device inbox is now a concurrent queue and tasks can be stolen by CPU or other device workers
  - Multitasks, range stealing and range merging in future



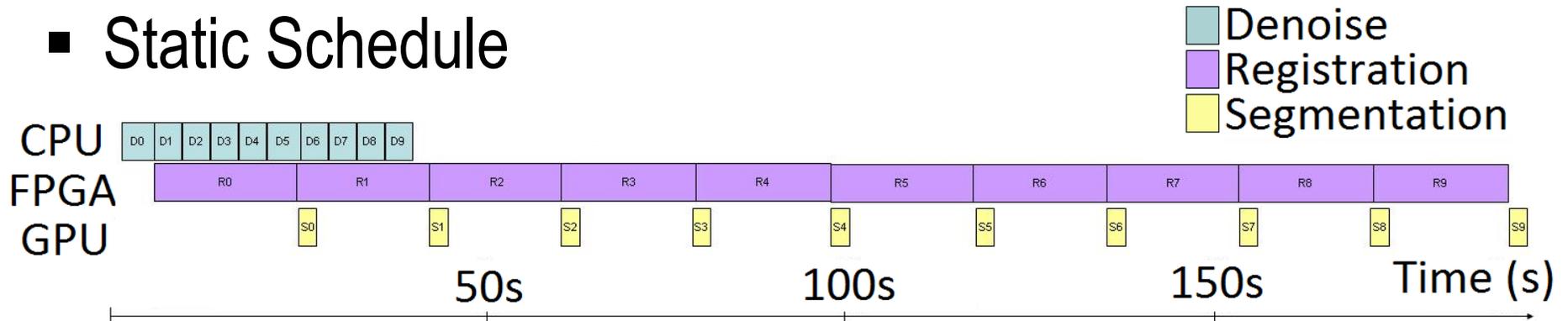
# Convey HC-1ex Testbed



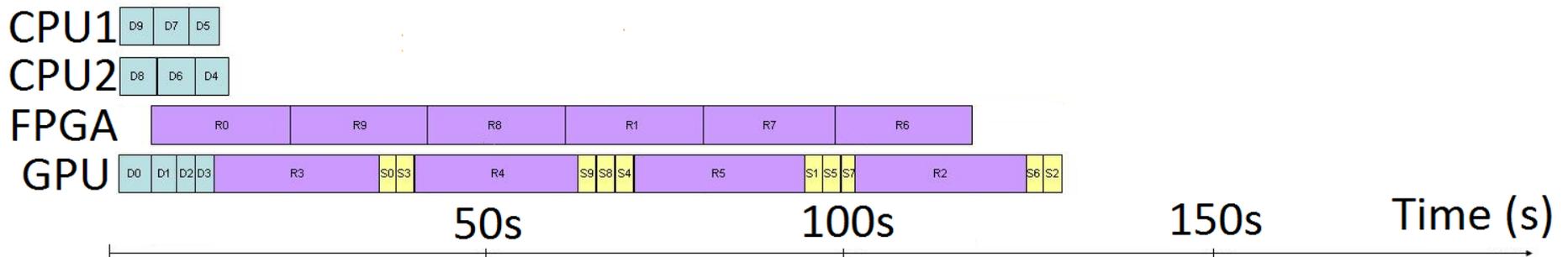
# Static vs Dynamic Scheduling

- ▶  $\langle C \rangle :: (D @CPU=20, GPU=10);$
- ▶  $\langle C \rangle :: (R @GPU=5, FPGA=10);$
- ▶  $\langle C \rangle :: (S @GPU=12);$

## Static Schedule



## Dynamic Schedule



## Experimental results

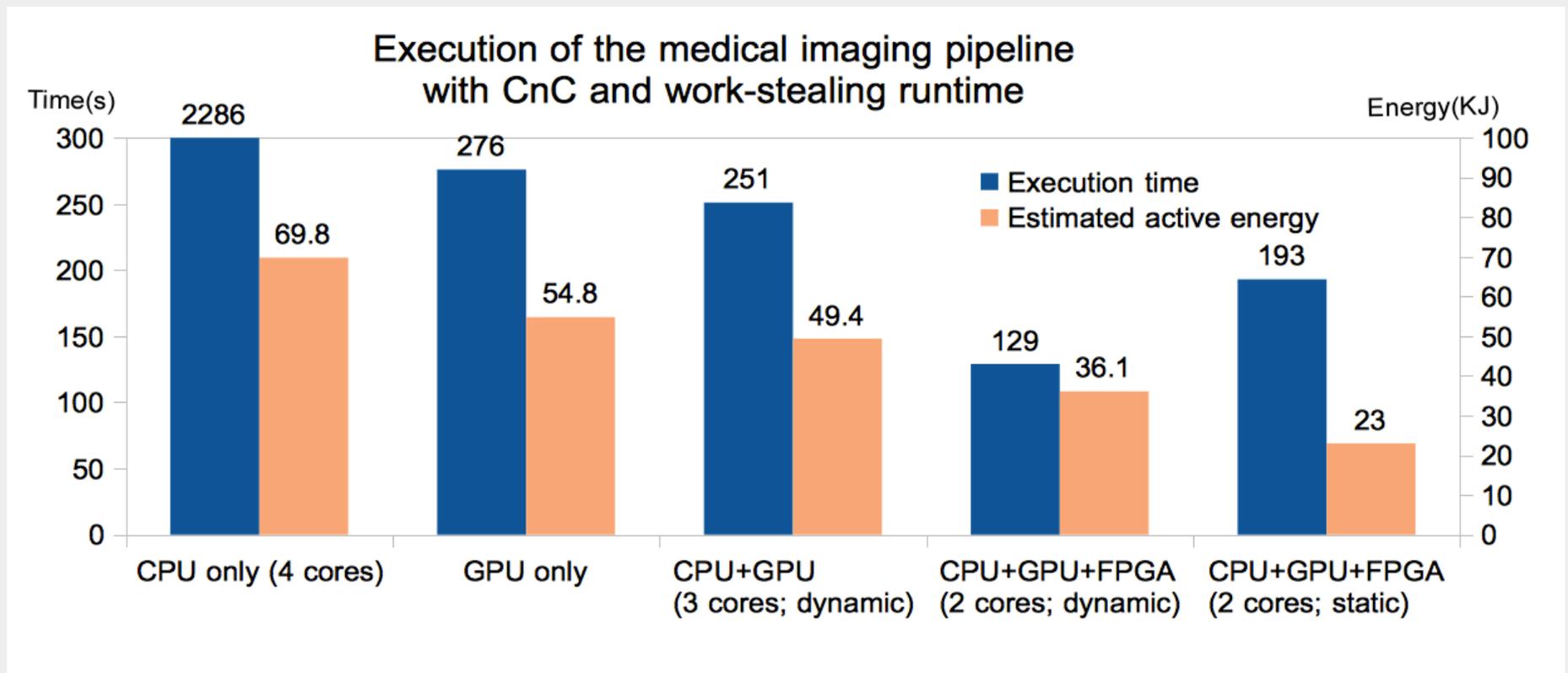
- Performance for medical imaging kernels

	Denoise	Registration	Segmentation
Num iterations	3	100	50
CPU (1 core)	3.3s	457.8s	36.76s
GPU	0.085s (38.3 ×)	20.26s (22.6 ×)	1.263s (29.1 ×)
FPGA	0.190s (17.2 ×)	17.52s (26.1 ×)	4.173s (8.8 ×)

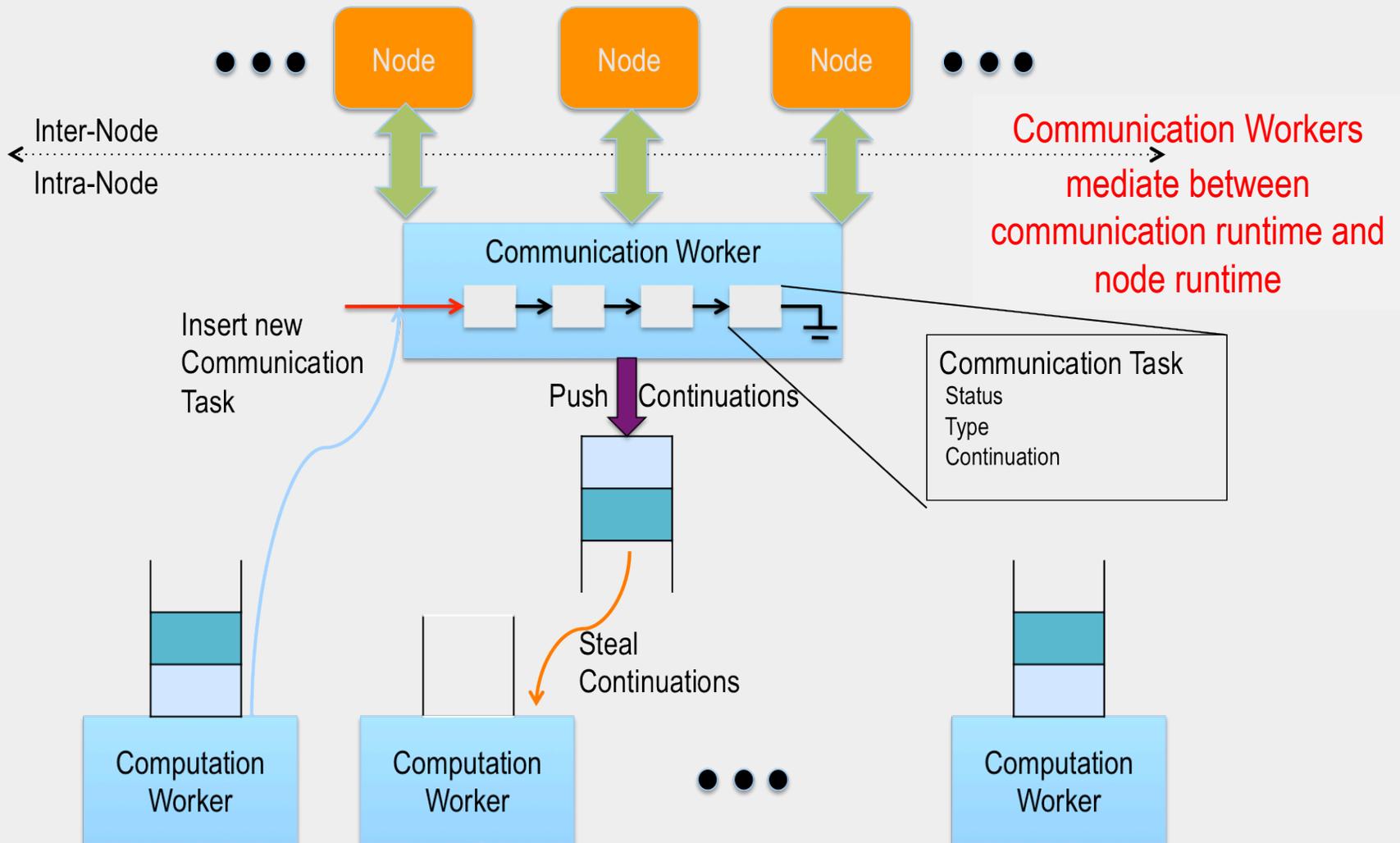


# Experimental results

- Execution times and active energy with dynamic work stealing



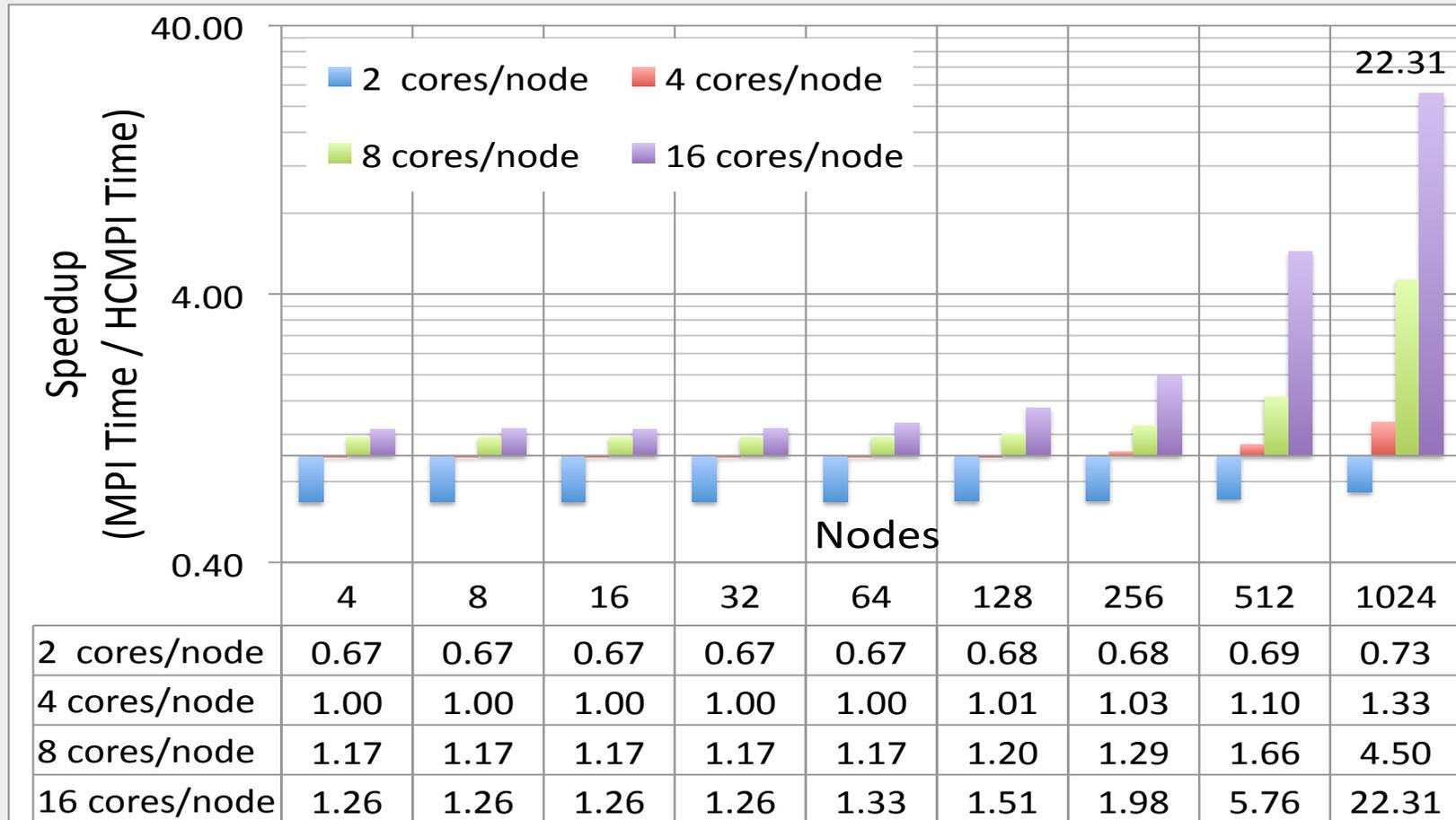
# From Locality to Communication --- Integrating Inter-node Communication with Intra-node Task Scheduling



"Integrating Asynchronous Task Parallelism with MPI." Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlic, Vincent Cave, Milind Chabbi, Max Grossman, Yonghong Yan, Vivek Sarkar. IPDPS 2013.



# UTS Performance on T1XXL

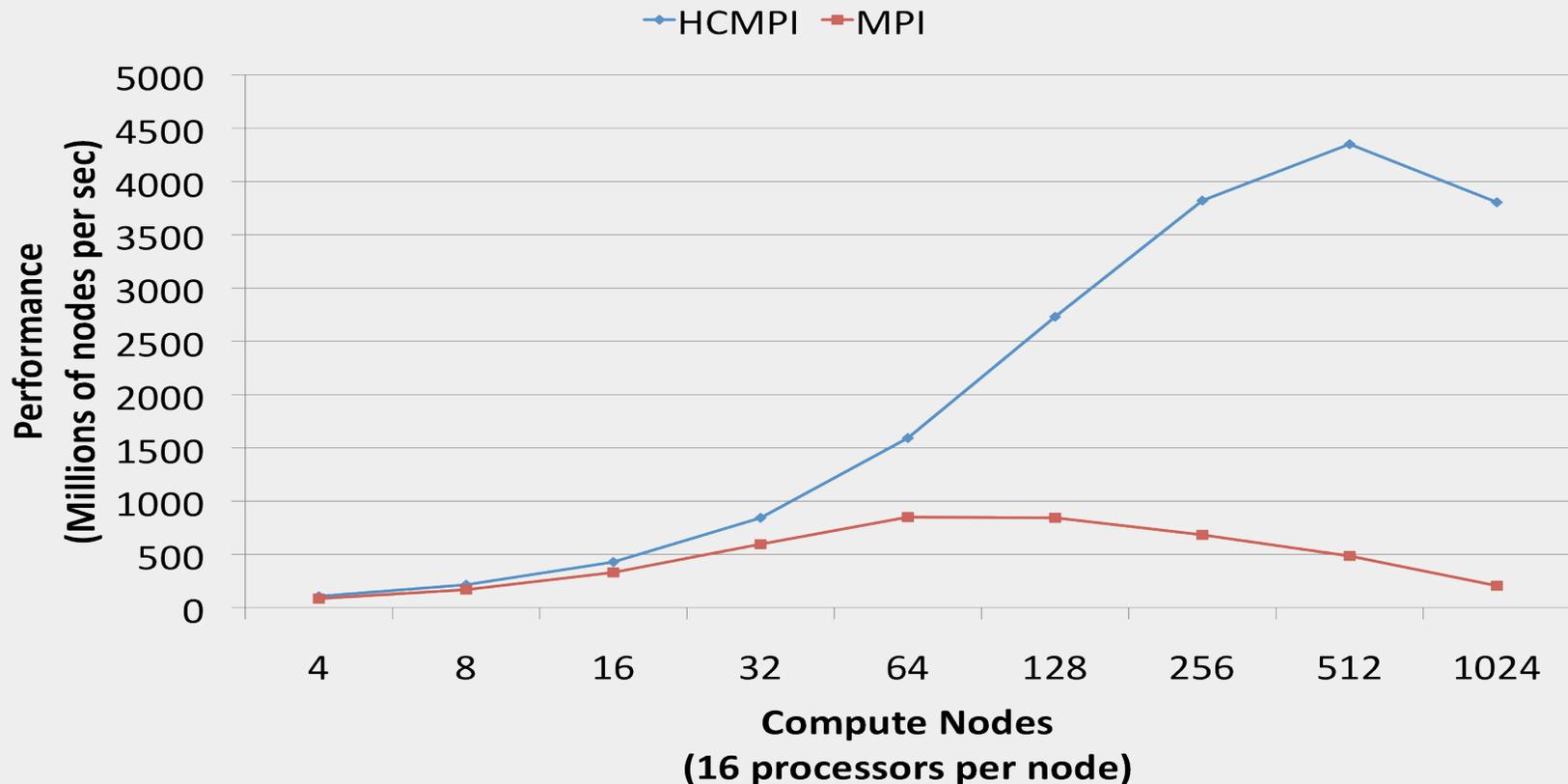


- Jaguar Supercomputer at ORNL
- 18688 nodes with Gemini Interconnect
- 16 core AMD Opteron nodes with 32 GB memory



# UTS Scaling on T1XXL

## Unbalanced Tree Search Performance Scaling



Failed steals lead to scalability bottleneck in MPI

- At 256 nodes: MPI suffers 2.35M failed steals while HCMPI suffers 0.82M
- At 1024 nodes: MPI suffers 94.75M failed steals while HCMPI suffers 8.83M



# PGNS Programming Model

- **Philosophy :**
  - In the Habanero Partitioned Global Name Space (PGNS) programming model, distributed tasks communicate via distributed data-driven futures, each of which has a globally unique id/name (guid).
  - PGNS can be implemented on a wide range of communication runtimes including GASNet and MPI (and OpenSHMEM), regardless of whether or not a global address space is supported.



# Distributed Data-Driven Futures (DDDFs)

```
int DDF_HOME (int guid) {...};
```

- a globally unique DDDF id → *home* rank

```
int DDF_SIZE (int guid) {...};
```

- a globally unique DDDF id → size of DDDF in bytes

```
DDF_t* ddfA = DDF_HANDLE(guid); (contrast with DDF_CREATE of shared  
memory)
```

- Allocate an instance of a distributed data-driven-future object (container)
- Multiple nodes can acquire handle to DDDF via its guid

```
async AWAIT(ddfA, ddfB, ...) <Stmt>
```

- Create a new data-driven-task to start executing *Stmt* after all of *ddfA*, *ddfB*, ... become available (i.e., after task becomes “enabled”)
- Seamless usage of distributed and shared memory DDFs
- Await registration handles the communication implicitly
  - Batches all communication at start of task (instead of individual get's)



# Distributed Data-Driven Futures (DDDFs)

**DDF\_PUT(ddfA, V);**

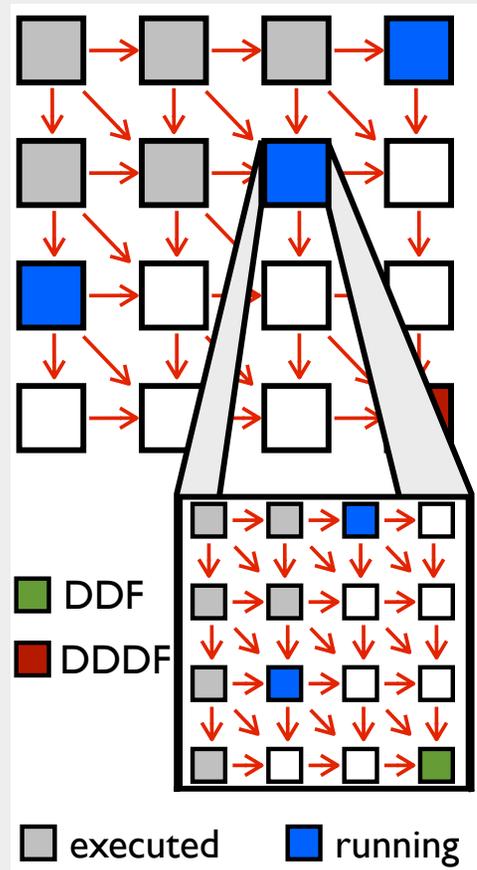
- Store object V in **ddfA**, thereby making **ddfA** available
- Single-assignment rule: at most one put is permitted on a given DDF
- Restricted only to *home* rank
- Handles communication to registrants implicitly

**DDF\_GET (ddfA) (macro for \_\_ddfGet(DDF\_t\* ddfObj))**

- Return value stored in **ddfA**
- Ensured to be safely performed by async's that contain **ddfA** in their await clause
- needs to be preceded by await clause on **ddfA** if the producer is remote
  - await can be in a different task provided local synchronization ensures the await precedes get



# Multi-Node SmithWaterman



```

#define DDF_HOME(guid) (guid%NPROC)
#define DDF_SIZE(guid) (sizeof(Elem))

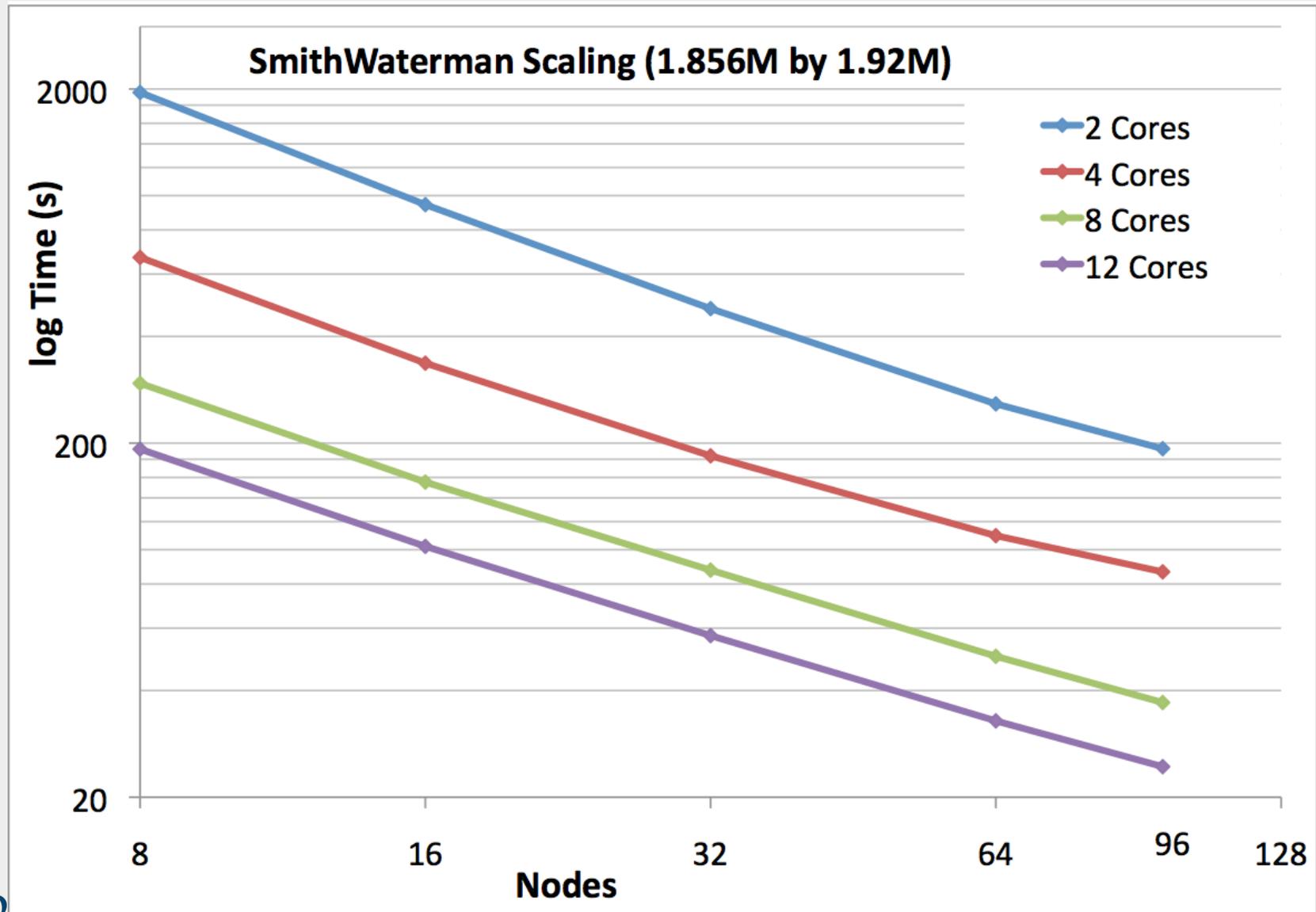
for (i=0;i<H;++i)
  for (j=0;j<W;++j)
    matrix[i][j] = DDF_HANDLE(i*H+j);

doInitialPuts(matrix);
finish {
  for (i=0,i<H;++i) {
    for (j=0,j<W;++j) {
      DDF_t* curr = matrix[i][j];
      DDF_t* above = matrix[i-1][j];
      DDF_t* left = matrix[i][j-1];
      DDF_t* uLeft = matrix[i-1][j-1];
      if ( isHome(i,j) ) {
        async AWAIT (above, left, uLeft){
          Elem* currElem =
            init(DDF_GET(above),
                DDF_GET(left),
                DDF_GET(uLeft));
          compute(currElem);
          DDF_PUT(curr, currElem);
        }/*async*/
      }/*if*/
    }/*for*/
  }/*for*/
}/*finish*/

```



# Results for APGNS version of SmithWaterman (communication runtime uses MPI under the covers)



# Open Community Runtime (OCR) Open Source Project

- Hosted on O1.org since 2012:
- Goals
  - Modularity
  - Support for multiple programming systems e.g., programming languages, libraries, compilers, DSLs, ...
  - Support for hardware platforms e.g., homogeneous manycore, heterogeneous accelerators, clusters, ...
- Development process
  - Continuous integration
  - Development plans driven by community milestones
- Organization
  - Steering Committee (SC) --- sets overall strategic directions and technical plans
  - Core Team (CT) --- executes technical plan and decides actions to take for source code contributions
  - Users and Contributors --- members of OCR community i.e., you!!

# OCR Acknowledgments

- Design strongly influenced by
  - Intel Runnemedede project (via DARPA UHPC program)
    - power efficiency, programmability, reliability, performance
  - Codelet philosophy – Prof. Gao’s group at U. Delaware
    - implicit notions of dataflow
  - Habanero project – Prof. Sarkar’s group at Rice U.
    - data-driven tasks, data-driven futures, hierarchical places
  - Concurrent Collections model – Intel Software/Solutions Group
    - decomposition of algorithm into steps/items/tags, tuning
  - Observation-based Scheduling – Intel Labs
    - monitoring and dynamic adaptation to load and environment
- *Partial support for the OCR development was provided through the X-Stack program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR)*

# Evolutionary vs. Revolutionary Approaches to Extreme Scale Runtime Systems

- Wide agreement that execution models for extreme scale systems will differ significantly from past execution models
  - Shoehorning a new execution model into an old runtime system is counter-productive
  - Instead, make a fresh start but carry forward reusable components from current runtime systems as appropriate
- Motivation for Open Community Runtime framework that ...
- is representative of future execution models
  - can be targeted by multiple high-level programming systems
  - can be mapped on to multiple extreme scale platforms
  - is available as an open-source testbed
  - reduces duplication of new infrastructure efforts
  - enables us to address revolutionary challenges collaboratively

# OCR Assumptions

- A *fine-grained, asynchronous event-driven runtime framework with movable data blocks and sophisticated observation* enables the next wave of high-performance computing
- *Fine-grained parallelism* helps achieve concurrency levels required for extreme scale
- *Asynchronous events and movable data blocks* help cope with performance variability, data movement, non-uniformity, heterogeneity, and resilience in extreme scale systems
- *Sophisticated observation* enables introspection into system behavior, feedback to OCR client, and adaptation based on algorithmic and performance tuning

# Example API: Creating an Event-Driven Task (EDT)

- `u8 ocrEdtCreate(ocrGuid_t * guid, ocrGuid_t templateGuid, u32 paramc, u64* paramv, u32 depc, ocrGuid_t *depv, u16 properties, ocrGuid_t affinity, ocrGuid_t *outputEvent);`
  - `guid [out]`: the assigned guid
  - `templateGuid`: the template the EDT is an instance of
  - `paramc`: nb of u64 parameters
  - `paramv`: pointer to u64 parameters
  - `depc`: nb of guid parameters
  - `depv`: array of guid dependences (if known at creation or NULL)
  - `properties`: can specify if finish-edt here.
  - `affinity`: affinity guid
  - `outputEvent [out]`: edt completion notification

# OCR Vision

C, C++, Fortran

R-Stream, ROSE, LLVM

CnC, Chapel

HC-lib, Habanero-UPC,  
PyGAS, OpenSHMEM, ...

Hero  
Programmer

Smart  
Compiler

Higher-level  
language

Higher-level  
library

OCR fills  
a major  
gap in  
intra-  
node  
runtime  
systems

Open Community Runtime Framework

External Runtime Components

MPI, GASNet,  
Portals, PAMI,  
UCCS, ARMCI, ...

Extreme Scale Platforms

Open  
Community  
Runtime

# Conclusions

- Holistic redesign of software stack is needed to address concurrency, energy, and resiliency challenges of Extreme Scale systems
- Urgent need for execution models that span multiple scales of parallelism and heterogeneity – multicore, accelerators, multi-node, HPC cluster, data center cluster
- Well-designed runtime primitives can provide foundation for new execution models, with synergistic innovation in languages and compilers
- OpenSHMEM is well suited to help with these challenges because of its lightweight support for a partitioned global address space
  - Integration of OpenSHMEM and OCR is an interesting opportunity
- Key question: how to simultaneously advance innovation in OpenSHMEM along two fronts?
  - Exploration of new ideas, as in Habanero and other research projects
  - Standardization of ideas to support broader adoption

