COMP 322: Fundamentals of Parallel Programming

Lecture 37: Comparison of Parallel Programming Models

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
In the following example from slide 23, assume that each UPC array is distributed by default across threads with a cyclic distribution. In the space below, identify an iteration of the upc forall construct for which all array accesses are local, and an iteration for which all array accesses are non-local (remote). Explain your answer in each case.

```c
shared int a[100], b[100], c[100];
int i;
upc forall (i=0; i<100; i++; (i*THREADS)/100)
    a[i] = b[i] * c[i];
```

Solution:
- Iteration 0 has affinity with thread 0, and accesses a[0], b[0], c[0], all of which are located locally at thread 0
- Iteration 1 has affinity with thread 0, and accesses a[1], b[1], c[1], all of which are located remotely at thread 1
Announcements

• Graded midterm exams can be picked up from Sherry Nassar in Duncan Hall 3139

• Homework 6 is officially due on April 19th, but everyone can get an automatic penalty-free extension till April 26th

• Final exam will be given on April 19th to be taken in any two-hour duration returned to Sherry Nassar by April 26th (as was done with midterm exams)
  — Final exam will cover material from Lectures 19 - 37

• Next lecture (April 19th) is the last lecture!
Acknowledgments

- “Introduction to Parallel Computing” by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003, and accompanying slides
  - http://www-users.cs.umn.edu/~karypis/parbook/

- Slides from COMP 422 course at Rice University
  - http://www.clear.rice.edu/comp422/


- Slides from OpenMP tutorial given by Ruud van der Paas at HPCC 2007
  - http://www.tlc2.uh.edu/hpcc07/Schedule/OpenMP

- “Towards OpenMP 3.0”, Larry Meadows, HPCC 2007 presentation
  - http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc07_Larry.ppt

- Pthreads: A Brief Introduction, CSCI 8530 lecture, University of Nebraska Omaha

- “Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder
  - Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html

- Tim Warburton, Rice University, “Introduction to GPGPU Programming”
  - 5-day course taught at Danish Technical University (DTU) in May 2011

Parallel Programming is a Cross-Cutting Concern

Developer Pyramid (not drawn to scale!)

Software Stack
- Application
- Domain-Specific Langs.
- Middleware
- Programming
- Runtime Systems
- Compilers
- System Libraries
- OS and Hypervisors

Application Developers

Parallel Programming

Infrastructure Developers

System Programmers
Different Parallel Programming Models for different Levels of Developer Pyramid and Software Stack

Software Stack

<table>
<thead>
<tr>
<th>Application</th>
<th>Middleware</th>
<th>Domain-Specific Langs.</th>
<th>Compilers</th>
<th>System Libraries</th>
<th>OS and Hypervisors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Application Developers

Infrastructure Developers

System Programmers

LabView
Matlab
Chapel, X10,
UPC, CAF
Habanero-Java
Habanero-C
Java threads
OpenMP
MPI
CUDA
OpenCL
Pthreads

COMP 322, Spring 2013 (V. Sarkar)
Outline

- Pthreads
- OpenMP
- CUDA
POSIX Thread API (Pthreads)

- Standard user threads API supported by most vendors
- Library interface, intended for system programmers
- Concepts behind Pthreads interface are broadly applicable
  - largely independent of the API
  - useful for programming with other thread APIs as well
    - Windows threads
    - Solaris threads
    - Java threads
    - ...
- Threads are peers, unlike Linux/Unix processes
  - no parent/child relationship
Asynchronously invoke `thread_function` in a new thread (like an async)

```c
#include <pthread.h>
int pthread_create(
    pthread_t *thread_handle, /* returns handle here */
    const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);    /* single argument; perhaps a structure */
```

*attribute* created by `pthread_attr_init`

contains details about

- whether scheduling policy is inherited or explicit
- scheduling policy, scheduling priority
- stack size, stack guard region size

Can use NULL for `pthread_attr_init` for default values
Pthread Termination

- A thread terminates by calling the function `pthread_exit()`. A single argument, a pointer to a void* object, is supplied as the argument to `pthread_exit`. This value is returned to any thread that has blocked while waiting for this thread to exit.

- Suspend parent thread until child thread terminates (like `Thread.join()` in Java)

```c
#include <pthread.h>
int pthread_join (
    pthread_t thread, /* thread id */
    void **ptr); /* ptr to location for return code a terminating thread passes to pthread_exit */
```
Example: Creation and Termination (main)

```c
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
    ...
    pthread_t p_threads[NUM_THREADS];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    for (i=0; i< NUM_THREADS; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
                       (void*) &hits[i]);
    }
    for (i=0; i< NUM_THREADS; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}
```
Example of Implementing a Reduction Using Mutex Locks

```c
pthread_mutex_t cost_lock;
...
int main() {
    ...
pthread_mutex_init(&cost_lock, NULL);
    ...
}
void *find_best(void *list_ptr) {
    ...
pthread_mutex_lock(&cost_lock); /* lock the mutex */
    if (my_cost < best_cost) /* critical section */
        best_cost = my_cost;
pthread_mutex_unlock(&cost_lock); /* unlock the mutex */
}
```

Composite Synchronization Constructs

• Pthreads provides only basic synchronization constructs

• Build higher-level constructs from basic ones e.g., barriers
  — Pthreads extension includes barriers as synchronization objects (available in Single UNIX Specification)
    – Enable by #define _XOPEN_SOURCE 600 at start of file
  — Initialize a barrier for count threads
    – int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrier_attr_t *attr, int count);
  — Each thread waits on a barrier by calling
    – int pthread_barrier_wait(pthread_barrier_t *barrier);
  — Destroy a barrier
    – int pthread_barrier_destroy(pthread_barrier_t *barrier);

• NOTE: Java threads and HJ worker threads are also implemented as pthreads
<table>
<thead>
<tr>
<th>Pthreads construct</th>
<th>Related HJ/Java constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create()</td>
<td>HJ’s async; Java’s “new Thread” and “Thread.start()”</td>
</tr>
<tr>
<td>pthread_join()</td>
<td>HJ’s finish &amp; future get(); Java’s “Thread.join()”</td>
</tr>
<tr>
<td>pthread_mutex_lock()</td>
<td>HJ’s begin-isolated, actors; Java’s begin-synchronized, and lock() library calls</td>
</tr>
<tr>
<td>pthread_mutex_unlock()</td>
<td>HJ’s end-isolated, actors; Java’s end-synchronized, and unlock() library calls</td>
</tr>
<tr>
<td>pthread_cond_signal()</td>
<td>Deterministic use: HJ’s phasers; Nondeterministic use: j.u.c.locks.condition</td>
</tr>
<tr>
<td>pthread_cond_wait()</td>
<td>Deterministic use: HJ’s phasers; Nondeterministic use: j.u.c.locks.condition</td>
</tr>
</tbody>
</table>
Outline

- Pthreads
- OpenMP
- CUDA
What is OpenMP?

- Well-established standard for writing shared-memory parallel programs in C, C++, Fortran

- Programming model is expressed via
  - Pragmas/directives (not language extensions)
  - Runtime routines
  - Environment variables

- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
  - Latest specification: Version 3.0 (May 2008)
  - Previous specification: Version 2.5 (May 2005)
A first OpenMP example

For-loop with independent iterations

```
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for
    shared(n, a, b, c)
    private(i)
    for (i = 0; i < n; i++)
        c[i] = a[i] + b[i];
```

% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out

OpenMP parallel for loop is like a forall loop in HJ
The OpenMP Execution Model

Fork and Join Model

- Master Thread
- Worker Threads

Parallel region

Synchronization

Parallel region
Terminology

- **OpenMP Team** := Master + Workers

- A **Parallel Region** is a block of code executed by all threads simultaneously
  
  - The master thread always has thread ID 0
  
  - Thread adjustment (if enabled) is only done before entering a parallel region

  - Parallel regions can be nested, but support for this is implementation dependent

  - An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially

- A **work-sharing construct** divides the execution of the enclosed code region among the members of the team; in other words: they split the work
Parallel Region

```c
#pragma omp parallel [clause[,] clause] ...
{
    "this is executed in parallel"
}
(implied barrier)
```

A parallel region is a block of code executed by multiple threads simultaneously in SPMD mode, and supports the following clauses:

- `if` (scalar expression)
- `private` (list)
- `shared` (list)
- `default` (nonel/shared) (C/C++)
- `default` (nonel/shared/private) (Fortran)
- `reduction` (operator: list)
- `copyin` (list)
- `firstprivate` (list)
- `num_threads` (scalar_int_expr)
Work-sharing constructs in a Parallel Region

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads
Legality constraints for work-sharing constructs

- Each worksharing region must be encountered by all threads in a team or by none at all.
- The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.

```c
#pragma omp parallel
{
  do {
    // c1 and c2 may depend on the OpenMP thread-id
    boolean c1 = ... ; boolean c2 = ... ;
    . . .
    if (c2) {
      // Start of work-sharing region with no wait clause
      #pragma omp ...
      . . . // Worksharing statement
    } // if (c2)
  } while (! c1);
}

==> No OpenMP implementation checks for conformance with this rule (unlike HJ’s runtime check for phaser single statements)
```
Example of work-sharing “omp for” loop

```c
#pragma omp parallel default(none)\ 
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

} /*-- End of parallel region --*/
(implied barrier)
```

Implicit finish
Like HJ’s forasync
Like HJ’s forasync


#pragma omp task [clause[[],]clause] ...
structured-block

where clause can be one of:

if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
Example – parallel pointer chasing using tasks

1. #pragma omp parallel
2. {
3.   #pragma omp single private(p)
4.   {
5.     p = listhead ;
6.     while (p) {
7.       #pragma omp task
8.       process (p);
9.       p = p->next ;
10.   }
11. }
12. }

 Spawn call to process(p)

Implicit finish at end of parallel region
### Summary of key features in OpenMP

<table>
<thead>
<tr>
<th>OpenMP construct</th>
<th>Related HJ/Java constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel region #pragma omp parallel</td>
<td>HJ forall (forall iteration = OpenMP thread)</td>
</tr>
<tr>
<td>Work-sharing constructs: parallel loops, parallel sections</td>
<td>No direct analogy in HJ or Java</td>
</tr>
<tr>
<td>Barrier #pragma omp barrier</td>
<td>HJ forall-next on implicit phaser</td>
</tr>
<tr>
<td>Single #pragma omp single</td>
<td>HJ’s forall-next-single on implicit phaser (but HJ does not support single + nowait)</td>
</tr>
<tr>
<td>Reduction clauses</td>
<td>HJ’s finish accumulators (in forall)</td>
</tr>
<tr>
<td>Critical section #pragma omp critical</td>
<td>HJ’s isolated statement</td>
</tr>
<tr>
<td>Task creation #pragma omp task</td>
<td>HJ’s async statement</td>
</tr>
<tr>
<td>Task termination #pragma omp taskwait</td>
<td>HJ’s finish statement</td>
</tr>
</tbody>
</table>
Outline

- Pthreads
- OpenMP
- CUDA
# Flynn’s Taxonomy for Parallel Computers

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single Data</strong></td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td><strong>Multiple Data</strong></td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

**Single Instruction, Single Data stream (SISD)**
A sequential computer which exploits no parallelism in either the instruction or data streams. e.g., old single processor PC.

**Single Instruction, Multiple Data streams (SIMD)**
A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. e.g. graphics processing unit.

**Multiple Instruction, Single Data stream (MISD)**
Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. e.g. the Space Shuttle flight control computer.

**Multiple Instruction, Multiple Data streams (MIMD)**
Multiple autonomous processors simultaneously executing different instructions on different data. e.g. a PC cluster memory space.
SIMD pattern:
Single Instruction Multiple Data

- Definition: A single instruction stream is applied to multiple data elements.
  - One program text
  - One instruction counter
  - Distinct data streams per Processing Element (PE)
Matrix Multiplication
(with multiple sources of parallelism)

```plaintext
double[][] a, b, c;  // three 2D arrays : a,b,c
int n;  // Assume that all arrays are of size n*n

forall(point[i,j] : [0:n-1,0:n-1] {
    c[i][j] = sum(a[i][0:n-1] * b[0:n-1][j]);
}

Dot product is expressed as SIMD parallelism
(This is pseudocode, not real HJ code)
```
Array slice notation

- Designating different slices of an array.

```
A[:][:]
A[3][:]
A[:][3]
A[2][3]
A[1:3][:]
A[1:3][2:4]
```
SPMD Pattern

- **SPMD**: Single Program Multiple Data
- Run the same program on P processing elements (PEs)
- Use the “rank” … an ID ranging from 0 to (P-1) … to determine what computation is performed on what data by a given PE
- Different PEs can follow different paths through the same code (unlike the SIMD pattern)
- Convenient pattern for hardware platforms that are not amenable to efficient forms of dynamic task parallelism
  - General-Purpose Graphics Processing Units (GPGPUs)
  - Distributed-memory parallel machines
- Key design decisions --- what data and computation should be replicated or partitioned across PEs?
SPMD Example: Iterative Averaging

1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
2. gVal[n+1] = 1; // Boundary condition
3. int Cj = Runtime.getNumOfWorkers();
4. forall (point [jj]:[0:Cj-1]) { // SPMD computation
5.   double[] myVal = gVal; double[] myNew = gNew; // Local copy
6.   for (point [iter] : [0:numIters-1]) {
7.     // Compute MyNew as function of input array MyVal
8.     for (point [j]:getChunk([1:n],[Cj],[jj]))
9.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.    next; // Barrier before executing next iteration of iter loop
11.   } // for
12. } // forall

CPUs and GPUs have fundamentally different design philosophies

**GPU = Graphics Processing Unit**

**Single CPU core**
- Control
- ALU
- ALU
- Cache
- DRAM

**Multiple GPU processors**
- Control
- ALU
- ALU
- Streaming Multiprocessor
- DRAM

GPUs are provided to accelerate graphics, but they can also be used for non-graphics applications that exhibit large amounts of data parallelism and require large amounts of “streaming” throughput ⇒ SIMD parallelism within an SM, and SPMD parallelism across SMs
Process Flow of a CUDA Kernel Call (Compute Unified Device Architecture)

- Data parallel programming architecture from NVIDIA
  - Execute programmer-defined kernels on extremely parallel GPUs
  - CUDA program flow:
    1. Push data on device
    2. Launch kernel
    3. Execute kernel and memory accesses in parallel
    4. Pull data off device
- Device threads are launched in batches
  - Blocks of Threads, Grid of Blocks
- Explicit device memory management
  - cudaMalloc, cudaMemcpy, cudaFree, etc.
- NOTE: OpenCL is a newer standard for GPU programming that is more portable than CUDA

Figure source: Y. Yan et. al “JCUDA: a Programmer Friendly Interface for Accelerating Java Programs with CUDA.” Euro-Par 2009.
Execution of a CUDA program

- Integrated host+device application
  - Serial or modestly parallel parts on CPU host
  - Highly parallel kernels on GPU device

Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)

Device Kernel
(large number of threads)

Host Code
(small number of threads)
Matrix multiplication kernel code in CUDA (SPMD model with index = threadIdx)

```c
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue stores the Pd element that is computed by the thread
    float Pvalue = 0;

    for (int k = 0; k < Width; ++k)
    {
        float Mdelement = Md[ty * Width + k];
        float Ndelement = Nd[k * Width + tx];
        Pvalue += Mdelement * Ndelement;
    }

    // Write the matrix to device memory each thread writes one element
    Pd[ty * Width + tx] = Pvalue;
}
```
Host Code in C for Matrix Multiplication

1. void MatrixMultiplication(float* M, float* N, float* P, int Width) {
2.    int size = Width*Width*sizeof(float); // matrix size
3.    float* Md, Nd, Pd; // pointers to device arrays
4.    cudaMalloc((void**)&Md, size); // allocate Md on device
5.    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice); // copy M to Md
6.    cudaMalloc((void**)&Nd, size); // allocate Nd on device
7.    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice); // copy N to Nd
8.    cudaMalloc((void**)&Pd, size); // allocate Pd on device
9.    dim3 dimBlock(Width,Width); dim3 dimGrid(1,1);
10.   // launch kernel (equivalent to “async at(GPU), forall, forall”
11.   MatrixMulKernel<<<dimGrid,dimBlock>>>(Md, Nd, Pd, Width);
12.   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost); // copy Pd to P
13.   // Free device matrices
14.   cudaFree(Md); cudaFree(Nd); cudaFree(Pd);
15. }

HJ abstraction of a CUDA kernel invocation: async at + forall + forall
• `cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)`

• copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind is one of
  —`cudaMemcpyHostToHost`
  —`cudaMemcpyHostToDevice`
  —`cudaMemcpyDeviceToHost`
  —`cudaMemcpyDeviceToDevice`

• The memory areas may not overlap

• Calling `cudaMemcpy()` with dst and src pointers that do not match the direction of the copy results in an undefined behavior.
CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong> int LocalVar;</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- _**device**_ is optional when used with _**local**_, _**shared**_, or _**constant**_.
- **Automatic variables** without any qualifier reside in a *register*
  - Except arrays that reside in local memory
- **Pointers** can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    _**global**_ void KernelFunc(float* ptr)
  - Obtained as the address of a global variable: float* ptr = &GlobalVar;
CUDA Storage Classes

- **Local Memory**: per-thread
  - Private per thread
  - Auto variables, register spill
- **Shared Memory**: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- **Global Memory**: per-application
  - Shared by all threads
  - Inter-Grid communication
# Summary of key features in CUDA

<table>
<thead>
<tr>
<th>CUDA construct</th>
<th>Related HJ/Java constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel invocation, <code>&lt;&lt;&lt;. . .&gt;&gt;&gt;</code></td>
<td><code>async at(gpu-place)</code></td>
</tr>
<tr>
<td>1D/2D grid with 1D/2D/3D blocks of threads</td>
<td>Outer 1D/2D forall with inner 1D/2D/3D forall</td>
</tr>
<tr>
<td>Intra-block barrier, <code>__syncthreads()</code></td>
<td>HJ forall-next on implicit phaser for inner forall</td>
</tr>
<tr>
<td><code>cudaMemcpy()</code></td>
<td>No direct equivalent in HJ/Java (can use <code>System.arraycopy()</code> if needed)</td>
</tr>
<tr>
<td>Storage classes: local, shared, global</td>
<td>No direct equivalent in HJ/Java (method-local variables are scalars)</td>
</tr>
</tbody>
</table>
## Comparison of Multicore Programming Models along Selected Dimensions

<table>
<thead>
<tr>
<th>Model</th>
<th>Dynamic Parallelism</th>
<th>Locality Control</th>
<th>Mutual Exclusion</th>
<th>Collective &amp; Point-to-point Synchronization</th>
<th>Data Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk</td>
<td>Spawn, sync</td>
<td>None</td>
<td>Locks</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Java Concurrency</td>
<td>Executors, Task Queues</td>
<td>None</td>
<td>Locks, monitors, atomic classes</td>
<td>Synchronizers</td>
<td>Concurrent collections</td>
</tr>
<tr>
<td>Intel C++ Threading Building Blocks</td>
<td>Generic algorithms, tasks</td>
<td>None</td>
<td>Locks, atomic classes</td>
<td>None</td>
<td>Concurrent containers</td>
</tr>
<tr>
<td>.Net Parallel Extensions</td>
<td>Generic algorithms, tasks</td>
<td>None</td>
<td>Locks, monitors</td>
<td>Futures</td>
<td>PLINQ</td>
</tr>
<tr>
<td>OpenMP</td>
<td>SPMD (v2.5), Tasks (v3.0)</td>
<td>None</td>
<td>Locks, critical, atomic</td>
<td>Barriers</td>
<td>None</td>
</tr>
<tr>
<td>CUDA</td>
<td>None until recently (v5)</td>
<td>Device, grid, block, threads</td>
<td>None</td>
<td>Barriers</td>
<td>SPMD</td>
</tr>
<tr>
<td>Habanero-Java (builds on Java Concurrency)</td>
<td>Async, finish</td>
<td>Places</td>
<td>Isolated blocks, Java atomic classes</td>
<td>Phasers, futures, data-driven tasks</td>
<td>Parallel array operations, Java concurrent collections</td>
</tr>
</tbody>
</table>