Hierarchical Phasers for Scalable Synchronization and Reductions in Dynamic Parallelism

IPDPS 2010
April 22nd, 2010
Jun Shirako and Vivek Sarkar
Rice University
Introduction

Major crossroads in computer industry
Processor clock speeds are no longer increasing
⇒ Chips with increasing # cores instead
Challenge for software enablement on future systems
~ 100 and more cores on a chip
Productivity and efficiency of parallel programming
Need for new programming model

Dynamic Task Parallelism
New programming model to overcome limitations of Bulk Synchronous Parallelism (BSP) model
Chapel, Cilk, Fortress, Habanero-Java/C, Intel Threading Building Blocks, Java Concurrency Utilities, Microsoft Task Parallel Library, OpenMP 3.0 and X10

Set of lightweight tasks can grow and shrink dynamically
Ideal parallelism expressed by programmers
Habanero-Java/C


Task parallel language and execution model built on four orthogonal constructs

- Lightweight dynamic task creation & termination
  - Async-finish with Scalable Locality-Aware Work-stealing scheduler (SLAW)
- Locality control with task and data distributions
  - Hierarchical Place Tree
- Mutual exclusion and isolation
  - Isolated
- Collective and point-to-point synchronization & accumulation
  - Phasers
Outline

Introduction

Habanero-Java parallel constructs
Async, finish
Phasers

Hierarchical phasers
Programming interface
Runtime implementation

Experimental results

Conclusions
Async and Finish

Based on IBM X10 v1.5

Async = Lightweight task creation

Finish = Task-set termination

Join operation

```
async {
// T1
  STMT1; STMT4; STMT7;
} //T2
async {
// T3
  STMT2; STMT5;
  STMT3; STMT6; STMT8;
} //T1
```

Dynamic parallelism

wait← End finish
Phasers

Designed to handle multiple communication patterns

Collective Barriers
Point-to-point synchronizations

Supporting dynamic parallelism
# tasks can be varied dynamically

Deadlock freedom
Absence of explicit wait operations

Accumulation
Reductions (sum, prod, min, …)
combined with synchronizations

Streaming parallelism
As extensions of accumulation to support buffered streams

References
Phasers

Phaser allocation

```java
phaser ph = new phaser(mode)
```

- Phaser `ph` is allocated with **registration mode**
- Mode:
  - **SINGLE**
  - Registration mode defines capability
  - **SIG_WAIT**(default)
    - There is a lattice ordering of capabilities

Task registration

```java
async phased (ph1<mode1>, ph2<mode2>, ... ) {STMT}
```

Created task is registered with `ph1` in `mode1`, `ph2` in `mode2`, ...

child activity’s capabilities must be subset of parent’s

Synchronization

```java
next:
```

Advance each phaser that activity is registered on to its next phase

Semantics depend on registration mode

Deadlock-free execution semantics
Using Phasers as Barriers with Dynamic Parallelism

```java
finish {
    phaser ph = new phaser(SIG_WAIT); //T1
    async phased(ph<SIG_WAIT>){ STMT1; next; STMT4; next; STMT7; } //T2
    async phased(ph<SIG_WAIT>){ STMT2; next; STMT5; } //T3
    STMT3; next; STMT6; next; STMT8; //T1
}
```

T1, T2, T3 are registered on phaser ph in SIG_WAIT

Dynamic parallelism set of tasks registered on phaser can vary
Phaser Accumulators for Reduction

```java
class PhaserAccumulators {
    phaser ph = new phaser(SIG_WAIT);
    accumulator a = new accumulator(ph, accumulator.SUM, int.class);
    accumulator b = new accumulator(ph, accumulator.MIN, double.class);

    // foreach creates one task per iteration
    foreach (point [i] : [0:n-1]) phased (ph<SIG_WAIT>) {
        int iv = 2*i + j;
        double dv = -1.5*i + j;
        a.send(iv);
        b.send(dv);
        // Do other work before next
        next;
        int sum = a.result().intValue();
        double min = b.result().doubleValue();
        ...
    }
}
```

**Allocation:** Specify operator and type

**send:** Send a value to accumulator

**next:** Barrier operation; advance the phase

**result:** Get the result from previous phase (no race condition)
Scalability Limitations of Single-level Barrier + Reduction (EPCC Syncbench) on Sun 128-thread Niagara T2

**Single-master / multiple-worker implementation**

Bottleneck of scalability

Need support for tree-based barriers and reductions, in the presence of dynamic task parallelism.
Outline

Introduction

Habanero-Java parallel constructs

Async, finish

Phaser

Hierarchical phasers

Programming interface

Runtime implementation

Experimental results

Conclusions
Flat Barrier vs. Tree-Based Barriers

Barrier = \textit{gather} + \textit{broadcast}

Gather: single-master implementation is a scalability bottleneck

Tree-based implementation

Parallelization in gather operation

Well-suited to processor hierarchy
Flat Barrier Implementation

Gather by single master

class phaser {
    List<Sig> sigList;
    int mWaitPhase;
    ...
}

class Sig {
    volatile int sigPhase;
    ...
}

Phaser

// Signal by each task
Sig mySig = getMySig();
mySig.sigPhase++;

// Master waits for all signals
//  -> Major scalability bottleneck
for (/*iterates over sigList*/) {
    Sig sig = getAtomically(sigList);
    while (sig.sigPhase <= mWaitPhase);
}
mWaitPhase++;

-----→ Hash table access by each task
→→→→→ List access by master task
API for Tree-based Phasers

Allocation

phaser ph = new phaser(mode, nTiers, nDegree);

- nTiers: # tiers of tree
  - “nTiers = 1” is equivalent to flat phasers

- nDegree: # children on a sub-master (node of tree)

  (nTiers = 3, nDegree = 2)

Registration

Same as flat phaser

Synchronization

Same as flat phaser
Tree-based Barrier Implementation

Gather by hierarchical sub-masters

class phaser {
    ...
    // 2-D array [nTiers][nDegree]
    SubPhaser [][] subPh;
    ...
}

class SubPhaser {
    List <Sig> sigList;
    int mWaitPhase;
    volatile int sigPhase;
    ...
}

nDegree = 2
Flat Accumulation Implementation

Single atomic object in phaser

class phaser {
    List <Sig>sigList;
    int mWaitPhase;
    List <accumulator>accums;
    ...
}

class accumulator {
    AtomicInteger ai;
    Operation op;
    Class dataType;
    ...

    void send(int val) {
        // Eager implementation
        if (op == Operation.SUM) {
            ...
        } else if(op == Operation.PROD){
            while (true) {
                int c = ai.get();
                int n = c * val;
                if (ai.compareAndSet(c,n))
                    break;
                else
                    delay();
            }
        } else if ...
    }
}
Hierarchical structure of atomic objects

class phaser {
    int mWaitPhase;
    List <Sig>sigList;
    List <accumulator>accums;
    ...
}

class accumulator {
    AtomicInteger ai;
    SubAccumulator subAccums [][];
    ...
}

class SubAccumulator {
    AtomicInteger ai;
    ...
}

nDegree = 2, lighter contention
Outline

Introduction
Habanero-Java parallel constructs
Async, finish
Phaser
Hierarchical phasers
Programming interface
Runtime implementation
Experimental results
Conclusions
Experimental Setup

Platforms

Sun UltraSPARC T2 (Niagara 2)

- 1.2 GHz
- Dual-chip 128 threads (16-core x 8-threads/core)
- 32 GB main memory

IBM Power7

- 3.55 GHz
- Quad-chip 128 threads (32-core x 4-threads/core)
- 256 GB main memory

Benchmarks

EPCC syncbench microbenchmark

- Barrier and reduction performance
Experimental Setup

Experimental variants

JUC CyclicBarrier
- Java concurrent utility

OpenMP for
- Parallel loop with barrier
- Supports reduction

OpenMP barrier
- Barrier by fixed # threads
- No reduction support

Phasers normal
- Flat-level phasers

Phasers tree
Barrier Performance with EPCC Syncbench on Sun 128-thread Niagara T2

Time per barrier [micro secs]

# threads

CyclicBarrier > OMP-for ≈ OMP-barrier > phaser

Tree-based phaser is faster than flat phaser when # threads ≥ 16
Barrier + Reduction with EPCC Syncbench on Sun 128-thread Niagara T2

Time per barrier [micro secs]

<table>
<thead>
<tr>
<th># threads</th>
<th>0</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
<th>350</th>
<th>400</th>
<th>450</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>513</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1479</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OMP for-reduction(+) > phaser-flat > phaser-tree

CyclicBarrier and OMP barrier don’t support reduction
## Barrier Performance with EPCC Syncbench on IBM 128-thread Power7 (Preliminary Results)

<table>
<thead>
<tr>
<th># threads</th>
<th>88.2</th>
<th>186.0</th>
<th>379.4</th>
<th>831.8</th>
</tr>
</thead>
</table>

**Time per barrier [micro secs]**

CyclicBarrier > phaser-flat > OMP-for > phaser-tree > OMP-barrier

Tree-based phaser is faster than flat phaser when # threads ≥ 16
Barrier + Reduction with EPCC Syncbench on IBM 128-thread Power7

Time per barrier [micro secs]

# threads

phaser-flat > OMP for + reduction > phaser-tree

187.2
(2 tiers, 16 degree) shows best performance for both barriers and reductions
Impact of (# Tiers, Degree) Phaser Configuration on IBM 128-thread Power7

(2 tiers, 32 degree) shows best performance for barrier
(2 tiers, 16 degree) shows best performance for reduction
Application Benchmark Performance on Sun 128-thread Niagara T2

Speedup vs. serial

- 62.8
- 62.4
- 31.8
- 31.5
Preliminary Application Benchmark Performance on IBM Power7 (SMT=1, 32-thread)

Speedup vs. serial
Preliminary Application Benchmark Performance on IBM Power7 (SMT=2, 64-thread)
For CG.A and MG.A, the Java runtime terminates with an internal error for 128 threads (under investigation)
Related Work

Our work was influenced by past work on hierarchical barriers, but none of these past efforts considered hierarchical synchronization with dynamic parallelism as in phasers

Tournament barrier


Adaptive combining tree


Extensions to combining tree


Analysis of MPI Collective and reducing operations

Conclusion

Hierarchical Phaser implementations

Tree-based barrier and reduction for scalability

Dynamic task parallelism

Experimental results on two platforms

Sun UltraSPARC T2 128-thread SMP

- Barrier
  - 94.9x faster than OpenMP for, 89.2x faster than OpenMP barrier,
  - 3.9x faster than flat level phaser

- Reduction
  - 77.2x faster than OpenMP for + reduction, 16.3x faster than flat phaser

IBM Power7 128-thread SMP

- Barrier
The flexible barrier functionality that was previously restricted to ForkJoinTasks (in class forkjoin.TaskBarrier) is being redone as class Phaser (targeted for j.u.c, not j.u.c.forkjoin), that can be applied in all kinds of tasks. For a snapshot of API, see http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/Phaser.html

Comments and suggestions are very welcome as always. The API is likely to change a bit as we scope out further uses, and also, hopefully, stumble upon some better method names.

Among its capabilities is allowing the number of parties in a barrier to vary dynamically, which CyclicBarrier doesn’t and can’t support, but people regularly ask for.

The nice new class name is due to Vivek Sarkar. For a preview of some likely follow-ons (mainly, new kinds of FJ tasks that can register in various modes for Phasers, partially in support of analogous X10 functionality), see the paper by Vivek and others:
http://www.cs.rice.edu/~vsarker/PDF/SPSS08-phasers.pdf

-Doug
A task allocates phaser tree by "new phaser(...)"

The task is registered on a leaf sub-phaser

Only sub-phasers which the task accesses are **active** at the beginning

Inactive sub-phasers don't attend barrier

---

**Diagram:**

- **Tier 0** (global)
  - **Phaser**
    - **sub** (active)
    - **sub** (inactive)
- **Tier 1**
  - **sub** (active)
  - **sub** (inactive)
  - **sub** (inactive)
- **Tier 2** (leaf)
  - **sub** (active)
  - **sub** (inactive)
  - **sub** (inactive)
  - **sub** (inactive)
Task Registration (Local)

Tasks creation & registration on tree

Newly spawned task is also registered to leaf sub-phasers

Registration to local leaf when # tasks on the leaf < nDegrees
Task Registration (Remote)

Task creation & registration on tree

Registration to remote leaf when \# tasks on the leaf \geq n\text{Degree}

The remote sub-phaser is \textit{activated} if necessary
Task creation & registration on tree

Registration to remote leaf when \# tasks on the leaf \geq nDegree

The remote sub-phaser is activated if necessary.
Pipeline Parallelism with Phasers

```java
finish {
    phaser [] ph = new phaser[m+1];
    // foreach creates one async per iteration
    foreach (point [i] : [1:m-1]) phased (ph[i]<SIGNAL>, ph[i-1]<WAIT>)
        for (int j = 1; j < n; j++) {
            a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
            next;
        }
    } // foreach
} // finish
```

![Diagram of pipeline parallelism with phasers](image)

- Loop carried dependence:
  - (i=1, j=1)
  - (i=2, j=1)
  - (i=3, j=1)
  - (i=4, j=1)

- Next:
  - (i=1, j=2)
  - (i=2, j=2)
  - (i=3, j=2)
  - (i=4, j=2)

- Next:
  - (i=1, j=3)
  - (i=2, j=3)
  - (i=3, j=3)
  - (i=4, j=3)

- Next:
  - (i=1, j=4)
  - (i=2, j=4)
  - (i=3, j=4)
  - (i=4, j=4)
Thread Suspensions for Workers

1. Wait for master in busy-wait loop
2. Call Object.wait() to suspend (release CPU)

doWait() {
    WaitSync myWait =
    getCurrentActivity().waitTbl.get(this);
    if (isMaster(...)) { ... } else { // Code for workers
        boolean done = false;
        while (!done) {
            for (int i = 0; < WAIT_COUNT; i++) {
                if (masterSigPhase >
                    myWait.waitPhase) {
                    done = true; break;
                }
            }
        }

        if (!done) {
            int currVal = myWait.waitPhase;
            int newVal = myWait.waitPhase + 1;
            castID.compareAndSet(currVal,
                newVal);  // Java library
            synchronized (myWait) {
                if (masterSigPhase <=
                    myWait.waitPhase)
                    myWait.wait();
                // Java library
            }
        }
    }
}
Wake suspended Workers

- Call Object.notify() to wake workers up if necessary

```java
void doWait() {
    WaitSync myWait = getCurrentActivity().waitTbl.get(this);
    if (isMaster(...)) { // Code for master
        waitForWorkerSignals(); masterWaitPhase++;
        masterSigPhase++;
        int currVal = masterSigPhase - 1;
        int newVal = masterSigPhase;
        if (!castID.compareAndSet(currVal, newVal)) {
            for (int i = 0; i < waitList.size(); i++) {
                final WaitSync w = waitList.get(i);
                synchronized (w) {
                    // Java library notify operation
                }
            }
        }
    } else {
        // Code for worker
    }
}
```
Accumulator API

Allocation (constructor)

accumulator(Phaser ph, accumulator.Operation op, Class type);

- ph: Host phaser upon which the accumulator will rest
- op: Reduction operation
  - sum, product, min, max, bitwise-or, bitwise-and and bitwise-exor
- type: Data type
  - byte, short, int, long, float, double

Send a data to accumulator in current phase

void Accumulator.send(Number data);

Retrieve the reduction result from previous phase

Number Accumulator.result();

Result is from previous phase, so no race with send
Different implementations for the accumulator API

- **Eager**
  - send: Update an atomic var in the accumulator
  - next: Store result from atomic var to read-only storage

- **Dynamic-lazy**
  - send: Put a value in accumCell
  - next: Perform reduction over accumCells

- **Fixed-lazy**
  - Same as dynamic-lazy (accumArray instead of accumCells)
  - Lightweight implementations due to primitive array access
  - For restricted case of bounded parallelism (up to array size)

Activities: \(a_1, a_2, a_3\)

Accumulators:
- **Eager**: Atomic var
- **Dynamic Lazy**: accumCells
- **Fixed Lazy**: accumArray

* fixed size array