Habanero-Java Library: a Java 8 Framework for Multicore Programming

PPPJ 2014
September 25, 2014

Shams Imam, Vivek Sarkar
shams@rice.edu, vsarkar@rice.edu
Rice University

https://wiki.rice.edu/confluence/display/PARPROG/HJ+Library
Introduction

• Multicore processors are now ubiquitous
  • server, desktop, and laptop hardware
  • smaller devices: smartphones and tablets
• Parallelism is the future of computing
• Introduce parallelism early into the Computer Science curriculum
Motivation for HJlib

- Writing parallel programs is hard
- Programmers need higher level parallel programming
- Distinguish between parallelism and concurrency
- Language approach requires extensions
  - Special compiler support
- Library-based approach integrates easily with existing code
  - Users can use IDE and tools of choice
  - Java 8 provides an excellent foundation for rich library support for parallelism and concurrency
Contributions of this paper

• Habanero-Java library (HJlib) - a pure Java 8 library implementation of a multi-faceted task-parallel programming model
• EventDrivenControl API which can be used to add new parallel constructs to HJlib
  • All existing HJlib synchronization constructs (e.g., futures, data-driven futures, phasers) are also built using this API
  • Automatic support for AEM and deadlock detection
• Abstract Execution Metrics (AEM) framework for HJlib
• Deadlock detection tool for HJlib
Simple Example: Two-way Parallel Array Sum

- Basic idea:
  - Decompose problem into two tasks for partial sums
  - Combine results to obtain final answer
  - Parallel divide-and-conquer pattern

Task 0: Compute sum of lower half of array
Task 1: Compute sum of upper half of array

Compute total sum
Simple Example: Two-way Parallel Array Sum

1. // Start of Task T0 (main program)
2. sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3. finish(() -> {
4.    async(() -> {
5.        // Child task computes sum of lower half of array
6.        for (int i=0; i < X.length/2; i++) sum1 += X[i];
7.    }); // end async
8.    // Parent task computes sum of upper half of array
9.    for (int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. }); // end finish
11. // Parent task waits for child task to complete (join)
12. return sum1 + sum2;
More complex HJlib example: Parallel Spanning Tree

1. class V {
2.   V [] neighbors; // Input adjacency list
3.   V parent; // Output spanning tree
4.   ...
5.   boolean tryLabeling(final V n) {
6.       isolated(this, () -> {  
7.           if (parent == null) parent = n;
8.       });
9.       return parent == n;
10.   } // end tryLabeling
11.   void compute() {
12.       for (int i=0; i<neighbors.length; i++) {
13.           final V child = neighbors[i];
14.           if (child.tryLabeling(this))
15.               // escaping async
16.               async(() -> { child.compute(); });
17.       }
18.   } // end compute
19. } // end class V
20. root.parent = root; // Use self-cycle to identify root
21. finish(() -> { root.compute(); });

NOTE: this parallel structure cannot easily be expressed using standard Java constructs
HJlib Parallel Constructs

- Structured fork-join style parallelism (*async/finish*)
- Parallel loops (*forall/forasync*)
- Weak Atomicity (*isolated*)
- Task Dependencies (*DataDrivenFuture*)
- Point-2-Point Synchronization (*Phaser*)
- Parallel Reductions (*FinishAccumulator*)
- Asynchronous Message Passing (*Actor*)

**Summary:** HJlib supports an explicit parallel programming model at a higher level of abstraction and a wider range of parallel constructs than standard Java
Implementation

• No external dependencies
• Build on the capabilities offered by the Fork/Join Framework
  • ForkJoinPool work-stealing scheduler in JDK
• Built using Java 8 features
  • Lambda expressions
  • Functional interfaces
  • Older JVMs can be targeted by bytecode transformations
• Publisher/Subscriber model in runtime to add additional features
Runtime Implementation

ForkJoinPool (one task queue per worker thread)

Threads blocked on EDCs

Running threads

Synchronization objects that utilize EDCs
Default Runtime

• Default runtime uses work-first policy with ForkJoin framework
• Blocking conditions block worker threads
  • Scheduler spawns additional worker threads to compensate
  • Runtime executes ‘non-blocking’ tasks before blocking
• All synchronizations implemented using EventDrivenControl data structure
  • Tracks which threads are blocked or can be resumed
• NEW: Alternative cooperative runtime is also available for HJlib
  • Pro: worker threads never block
  • Con: current continuation support adds overhead due to use of exceptions and on-the-fly bytecode transformation in class loader
Event-Driven Control (EDC)

- Binds a value and a list of `java.lang.Runnable` blocks
  - Runnable blocks are code executed as callbacks
- Dynamic single-assignment of value (event)

The EDC is initially empty
Event-Driven Control (EDC)

- Binds a value and a list of Runnable blocks
- Dynamic single-assignment of value (event)

Runnable blocks attach to the EDC and are not triggered until value is available (i.e. until event is satisfied)
Event-Driven Control (EDC)

- Binds a value and a list of Runnable blocks
- Dynamic single-assignment of value (event)

Eventually, a value becomes available in the EDC (follows from deadlock freedom property of finish, futures, clocks, atomic)
Event-Driven Control (EDC)

• Binds a value and a list of Runnable blocks
• Dynamic single-assignment of value (event)

This enables execution of runnable blocks attached to the EDC
Event-Driven Control (EDC)

- Binds a value and a list of Runnable blocks
- Dynamic single-assignment of value (event)

Once value is available, subsequent runnable block attachment requests...
Event-Driven Control (EDC)

• Binds a value and a list of Runnable blocks
• Dynamic single-assignment of value (event)

Synchronously execute the block
(e.g. schedule a task into the work queue)
Event-Driven Control API

- **currentTaskId()**:
  - returns a unique id of the currently executing task
- **newEDC()**:
  - factory method to create EDC instance
- **suspend( anEdc ):**
  - the current task is suspended if the EDC has not been resolved
  - Implementation attaches runnable block to resume task
- **anEdc.getValue()**
  - retrieves the value associated with the EDC
  - safe to call this method if execution proceeds past a call to `suspend()`
- **anEdc.setValue( aValue )**
  - resolves the EDC
  - triggers the execution of any EBs
Parallel Constructs in Runtime

• Any task-parallel Synchronization Constraint can be supported.
  • Both deterministic and non-deterministic constructs

• All HJlib parallel constructs implemented using EDCs and event listeners (publish/subscriber) attached to runtime

• Key idea is to:
  • Translate the coordination constraints into producer-consumer constraints on EDCs
  • Block/Suspend consumers when waiting on item(s) from producer(s)

• Developers can add their implement own parallel constructs and add to HJlib
  • E.g. EventCount, others noted in future work
Abstract Execution Metrics

- Enable users to reason about the performance of their parallel algorithms
- Compute total work done and critical path length
  - Dynamically generates the computation graph
  - Details for each construct in paper
  - Users can integrate custom parallel constructs
- Performance metrics are reproducible
  - Independent of physical machine used
  - Useful for debugging performance problems and comparing alternate implementations
- Computation graph can also be displayed visually
Abstract Execution Metrics

- Prints the total Work and Critical Path Length (CPL)
- Supported for all HJlib constructs
- Enabled using:
  ```java
  HjSystemProperty.abstractMetrics.set(true)
  ```
- Dump obtained by:
  ```java
  HjMetrics actualMetrics = abstractMetrics();
  AbstractMetricsManager.
      dumpMetrics(actualMetrics);
  ```
- Can also use WORK and CPL metrics to obtain abstract time plots as shown on right
Visual Graphs

• Example: MergeSort
  • Use each comparison operation as work

```java
final int mid = M + (N - M) / 2;
finish(() -> {
  async(() -> mergesort(A, M, mid));
  async(() -> mergesort(A, mid + 1, N));
});
merge(A, M, mid, N);
```
MergeSort Computation Graph
Deadlock detector

- Enable users to debug their programs while using the various synchronization constructs
  - Reports diagnostic error message
- If they venture beyond the deadlock-free subset of Hjlib
  - async/finish/future/phaser programs are deadlock-free
- DDFs/Actors/EventCounts/Other custom constructs can cause deadlocks
- Algorithm relies on tracking the number of
  - ready tasks in the work queue
  - blocked or pending tasks (suspended on EDCs)
  - Deadlock when work queue is empty but there are pending tasks
Deadlock Example

1 $\text{finish}()$ -> {
2   $\text{HjDataDrivenFuture}<$Long$>$ A = $\text{newDDF}()$;
3   $\text{HjDataDrivenFuture}<$Long$>$ B = $\text{newDDF}()$;
4   $\text{asyncAwait}(B, () \rightarrow A.put(B.get() + 3))$;
5   $\text{asyncAwait}(A, () \rightarrow B.put(A.get() + 5))$;
6 });
Pedagogic programming model

- Attractive tool for educators
- Educational resources available from COMP 322 website
  - Lecture notes and videos
- Extensive documentation and examples available
- Institutions can introduce parallel programming earlier in curricula
  - Based purely in Java
  - Garnered positive feedback from COMP 322 students
- Plan to use in MOOC version of COMP 322

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
DEMO

• Setting up a simple HJlib project
• Some examples
  • ArraySum
  • QuickSort (Abstract Metrics)
  • Deadlock Detection
    • Data-Driven Futures
    • EventCount
Future work, Ongoing Research

- Performant Cooperative runtime
- Visual Computation Graphs for AEMs
- Visual display of Deadlocks
- Parallel constructs
  - Selectors – an extension to actors
  - Eureka-style computations
- Use of HJlib for multicore parallelism in Android applications
- ...

- See http://habanero.rice.edu for related research & publications
Summary

• Pure library implementation on Java 8
• Introduces orthogonal parallel constructs with important safety properties
  • Simplifies parallel programming
• Feedback capabilities help the programmer debug applications
• Pedagogic: Attractive tool for both educators and researchers
• Educational material already available (COMP 322 lectures, videos, etc.)
Questions

• Pure library implementation on Java 8
• Introduces orthogonal parallel constructs with important safety properties
• Simplifies parallel programming
• Feedback capabilities help the programmer debug applications
• Pedagogic: Attractive tool for both educators and researchers
• Educational material already available (COMP 322 lectures, videos, etc.)

```java
import pppj.audience.Questions;
import pppj.audience.Comments;
import pppj.audience.Feedback;
```
Backup-Slides
Acknowledgments

• Rest of the Habanero Group
  • Vincent Cave
  • Jun Shirako
  • Sagnak Tasirlar
  • Jisheng Zhao
Habanero-Java library

- Inspired from pedagogic Habanero-Java (HJ) language
- Emphasis on the usability and safety of parallel constructs
- Used in second-year undergraduate course at Rice

COMP 322: Fundamentals of Parallel Programming (Spring 2014)

<table>
<thead>
<tr>
<th>Instructor:</th>
<th>Prof. Vivek Sarkar, DH 3080</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assistant:</td>
<td>Benny Anderson, <a href="mailto:andersen@rice.edu">andersen@rice.edu</a>, DH 3080</td>
</tr>
<tr>
<td>Graduate TA:</td>
<td></td>
</tr>
</tbody>
</table>

- Actively used in multiple research projects at Rice
Habanero-Java library

• Supports an explicit parallel programming model
  • A high level of abstraction
  • A wider range of parallel programming constructs
• A powerful and portable task parallel programming model
  • For the Java ecosystem
• Parallelize both regular and irregular applications
Example: EventCount

• Keeps a count of the number of events in a particular class
• advance: signal the occurrence of an event
• await(v): suspends task until the value of the eventcount is at least v
• read: return count of the advance operations so far
public final class EventCount {

    Map<Long, EventDrivenControl> eventMap = new ConcurrentHashMap<>();
    AtomicLong eventCounter = new AtomicLong(0);

    public EventCount() {
        EventDrivenControl edc = newEDC();
        eventMap.put(0L, edc);
        edc.setValue(Boolean.TRUE);
    }

    public void advance() {
        long v = eventCounter.incrementAndGet();
        eventMap.putIfAbsent(v, newEDC());
        EventDrivenControl edc = eventMap.get(v);
        edc.setValue(Boolean.TRUE);
    }

    public void wait(final long v) {
        eventMap.putIfAbsent(v, newEDC());
        EventDrivenControl edc = eventMap.get(v);
        suspend(edc);
    }

    public long read() {
        return eventCounter.get();
    }
}