Comparing the Usability of Library vs. Language Approaches to Task Parallelism

Vincent Cavé, Zoran Budimlic , Vivek Sarkar
habanero.rice.edu
Rice University

Why Task Parallelism ?

- More and more physical cores
- More and more mainstream programmers writing parallel software
- Task parallelism is easy to understand and model
- Two approaches to task parallelism
  - Library
  - Language
- Compare the Java Concurrent Utilities Library and the Habanero-Java language
- How much are they usable for mainstream programmers ?

Outline

- Overview
  - j.u.c
  - Habanero-Java
- Comparison
  - Task Creation and Scheduling
  - Task Synchronization
  - Loop parallelism
- Conclusion

The Java Concurrent Utilities Library

- General purpose Java library for concurrency
- Features
  - Executor framework
    - work-sharing thread pools, fork-join framework
  - Concurrent collections
    - Maps, non blocking queues, etc..
  - Synchronizers
    - Latches, Semaphores, Phasers
  - Locks
    - Reentrant, ReadWrite locks…
- Provides basic constructs to parallelism

The Habanero-Java Language

- Developed at Rice University
  - Used in introductory class to parallel programming
- Derived from X10 version 1.5 Extends Java language with new keywords
- Task oriented programming model
- The HJ language features
  - Task creation: async, futures
  - Task synchronization: finish, phasers
  - Concurrency: isolated
  - Loop parallelism: foreach, forall
- HJ programs are deadlock free

Comparison

- What a mainstream programmer wants ?
- Basic features for Task Parallelism:
  - Task Creation and Scheduling
  - Task Synchronization
  - Loop Parallelism
- How to express these in a library approach (j.u.c) or a language approach (HJ) ?

Task Creation: Library Approach

```
List<Callable<Void>> list = …
for(int i = …) {
  list.add(new Callable<Void>() {
    public Void call() {
     // some computation
    }
   });
}
executor.invokeAll(list);
```

- What do we need ?
  - A task executor
  - A task interface
  - A task implementation
- Drawback
  - Readability
  - Programming chores
    - Manage tasks
    - Schedule tasks
  - Going for troubles !

Task Creation: Language Approach

```
finish {
   for(int i = …) {
      async {
         // some computation
       }
   }
}
```

- What a mainstream programmer wants ?
- Run “this” code in parallel
  - Simple task creation
  - No task management
  - No explicit task scheduling

Task Execution Policies

- Work-sharing / Work-stealing
  - j.u.c has two apis
  - Should be a runtime setting
- Library approach to work-stealing is difficult
  - Not trivial to have a unified api
  - Problem of tasks that blocks
- Language approach is more flexible
  - Can rely on compiler analysis transformation
  - Implements several scheduling policies

Task synchronization: phasers

- Available both in j.u.c and HJ

- A phaser is a synchronization object
  - Tasks can register dynamically to phasers
  - Registered tasks participate in a “phase”
  - Task synchronize on a “next” operation

Phasers in j.u.c

```
final MyPhase p = new MyPhase();
p.register();
for(…) {
  p.register();
  new Thread(new Runnable() {
    public void run() {
      while (cond()) {
        someComputation();
        p.arriveAndAwaitAdvance();
      }
    }
   }).start();
}
p.arriveAndDeregister();
```

- Code poorly readable
- Error prone
  - Phaser registration
  - Barrier code is out of scope

Phasers in HJ

```
finish {
  phaser p = new phaser();
  for(…) {
    async phased {
      while(cond()) {
        someComputation();
        next single {
          someReduction();
        }
      }
    }
  }
}
```

- Task registration
  - Phased keyword ensures registration
  - The parent task deregister automatically when reaching the finish
- Barrier
  - next keyword act as a barrier
  - single allows to specify code to execute at the barrier (optional)

Loop Parallelism

```
forall(point [i] : [0:N-1]) {
  while(cond()) {
    someComputation();
    next single {
      someReduction();
    }
  }
}
```

- Simple way to take advantage of embarrassingly parallel loops
  - "forall" points of an iteration space
  - Run the loop body asynchronously
  - Implicit finish and phaser

Conclusion

- Library task implementations are
  - General purpose and flexible but too low level
  - Need to be conservative
  - Lack of expressiveness
- Language approach to task can
  - Define a programming model semantic
  - Rely on keywords to hide complexity
  - Rely on compiler and possibly runtime
- Programming languages need to evolve to encompass task parallelism for mainstream programmers