Data Race Detection in HJ

- Dynamic Program Structure Tree (DPST) [3]
- Contains a node for each step that has no parallel constructs
- A node for each async and finish construct
- The root of the tree is always the finish node that encloses the main method
- Each time an async is encountered a new branch splits off
- Determining Dynamic May Happen in Parallel (DMHP) [3]
- Find the least common ancestor (LCA) of two steps
- Find the left child of the LCA that is also an ancestor of one of the steps
- If that child is an async, the steps may happen in parallel

```
public static void main (String[] args)
{
    AnyObject = new AnyObject(20);
    async {AnyObject.field = 21;}
    async phased (phaser<WAIT>);
    finish
}
```

The below DPST shows the lack of a data race in the accompanying code. While steps 2 and 3 still write to the same memory location, they cannot occur concurrently.

```
public static void main (String[] args)
{
    AnyObject = new AnyObject(20);
    finish
    {  async{AnyObject.field = 21;}
    }
    async phased (phaser<SIG>);
    finish
    {  async{AnyObject.field = 22;}
    }
}
```

Background

- Many programmers struggle with parallel programming due to its often nondeterministic nature
- Data races (such as the one shown to the right) are bugs that can easily go unseen as they may pass many traditional tests
- Data Race: Two threads access the same location in memory at the same time
- At least one of the threads performs a write on the location
- In general, data race detection is an NP-hard problem [2]
- Languages like HJ (e.g., Cilk and X10) are structured to provide support for parallel programming—making race detection significantly simpler [3]

```
async phased (phaser<WAIT>);
finish
```

Focus: Phasers

- Provide point-to-point synchronization [1]
- Registered on one or more async tasks
- Can be registered in 3 different modes
  - Signal/Wait
  - Wait Only
  - Signal Only
- Can additionally include a statement to be executed once by one of the tasks in the barrier formed by next

```
async phased (phaser<SIG>);
finish
```

Contributions

- Modified DPST to account for phaser constructs
- Created algorithm to work with modified DPST to detect Data Races
  - Follows the original HJ algorithm (with some additional record keeping)
  - Then adds additional support for phasers (by changing the DMHP method)

```
finish
{  phased = new phaser<SYNC_Wait>(){
  async phased (phaser<SIG>);
  next;
  /step2
  /step3
  async phased (phaser<WAIT>);
  /step4
  /step5
}
```

Algorithm

Determine if two steps may happen in parallel:

```
Traverse up the DPST to the two steps’ LCA {
    During the traversal keep a count of each phaser’s signals and waits (barriers);
}
if (steps can happen in parallel according to the Cave et al. algorithm)
{
    if(tasks have no phasers registered or different phasers registered)
    { Steps may happen in parallel; }
    for (each phaser that both tasks are registered on)
    { if ((number of signals between step1 and LCA >= the number of waits between step2 and LCA) or (number of signals between step2 and LCA >= the number of waits between step1 and LCA))
        { Steps may happen in parallel; }
    }
    Steps cannot happen in parallel given the same input;
}
else { Steps cannot happen in parallel given the same input; }
```

Further Work

- Implement the modified DPST and algorithm by adding it to HJ’s source (i.e. modify ESPbags.java and RaceDetInstrumentor.java in the HJ source files)
- Test the modified algorithms against benchmarks as done by Cave et al.

References