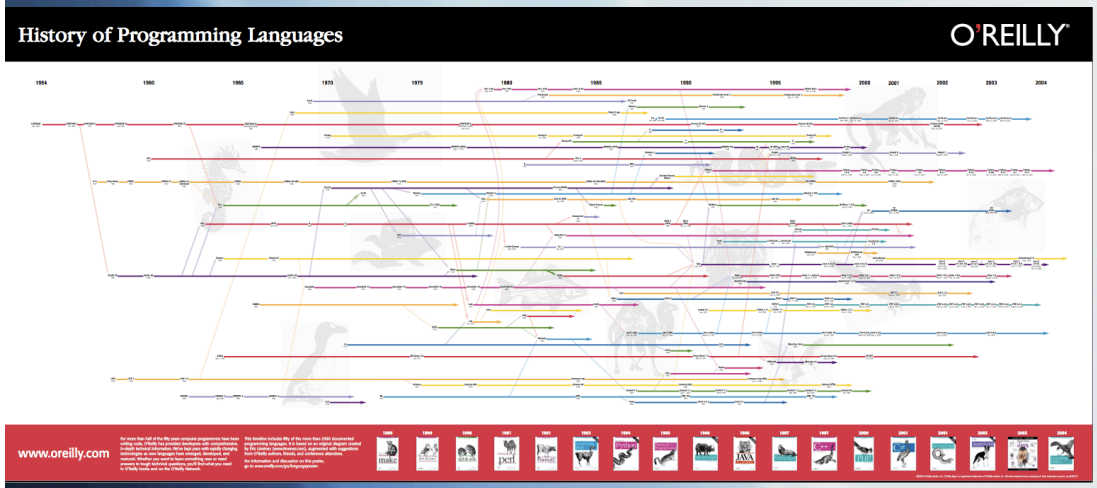
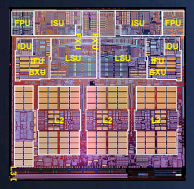
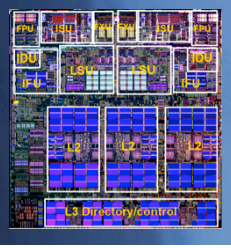
RICE George R. Brown School of Engineering
Computer Science

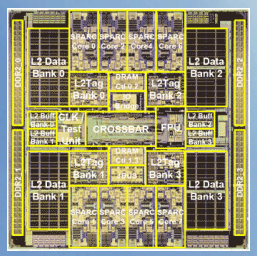
Intermediate Language Extensions for Parallelism

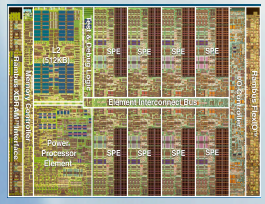


Jisheng Zhao and Vivek Sarkar
Dept. of Computer Science
Rice University









Outline

- Motivation
- Habanero-Java (HJ) language and Parallelism Intermediate Representation design
  - Parallelism features
  - Parallel Intermediate Representation (PIR)
  - Case Study of PIR Optimizations
- Summary

Motivation

- Intermediate Language (IL) for task-parallel languages
  - task-parallel languages: Cilk, X10, Habanero-Java, OpenMP 3.0, Chapel
  - piggy-backed on ILs for sequential languages
  - parallel constructs are translated to runtime calls
- New requirement for analyzing & optimizing parallel program
  - efficient task creation & termination; mutual exclusion; locality control; point-to-point synchronization

Habanero-Java (HJ) Language

- New language that addresses the issues in parallel programming
- Deadlock safety: none of these constructs can cause a deadlock cycle
- Implementation developed in Habanero Multicore Software research project at Rice (http://habanero.rice.edu)
  - Download available at http://habanero.rice.edu/hj-download
  - Derived from X10 v1.5 implementation in 2007
  - HJ is an extension of Java 1.4

Habanero Java (HJ) Execution Model: Portable Parallelism in Four Dimensions

1. Lightweight dynamic task creation & termination
   - *async*, *finish, future, foreach, forall*
2. Collective and point-to-point synchronization
   - *phasers* (extension of X10's clocks)
3. Mutual exclusion and isolation
   - *isolated* (extension of transactions & X10's atomic)
4. Locality control --- task and data distributions
   - *hierarchical place* tree

Lightweight Task Creation and Termination

async [seq(cond)] S

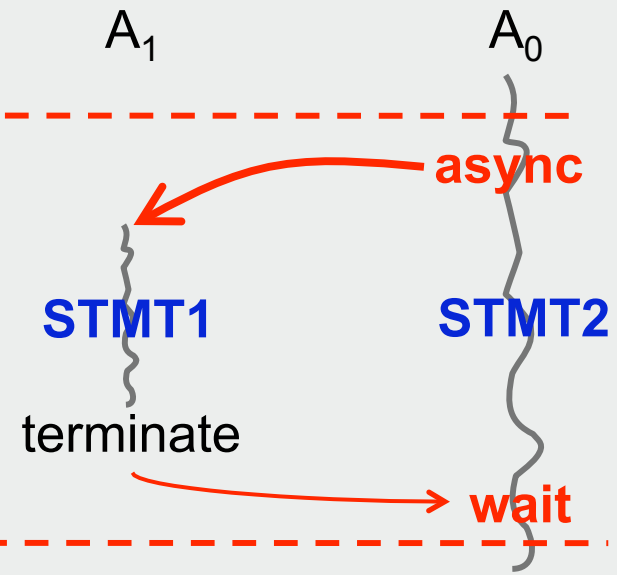
- Creates a new child task that executes statement S ; parent task can proceed immediately to operation following the async
- Optional "seq" clause
  - async seq(cond) <stmt> ≡

    if (cond) <stmt> else async <stmt>

finish S

- Execute S, but wait until *all* (transitively) spawned asyncs in S's scope have terminated.
- Implicit finish between start and end of main program
- *Use of finish synchronization cannot create a deadlock cycle*

```
//A0 (Parent)
finish {    //Begin finish
  async {
    STMT1;  //A1 (Child)
  }
  STMT2;    //A0
}           //End finish
```



Mutual Exclusion and Isolation: HJ isolated statement

isolated <body>

- Only one task can execute an isolated statement at a time
  - ➔ <body> is executed in isolation of other instances of <body>
  - ➔ Guarantees mutual exclusion between any pair of isolated statement instances
  - ➔ *Isolation consistency memory model* (weak atomicity)
    - ➔ no guarantee on interactions with non-isolated statements i.e., to (isolated, non-isolated) and (non-isolated, non-isolated) pairs of statement instances
- Isolated statements may be nested
- Isolated statements must not contain any other parallel statement e.g., forall, async, finish, force, next
- Delegated Isolation

  Lublinerman et. al. "Delegated Isolation" SPLASH'11

Point-to-Point synchronization: HJ Phasers

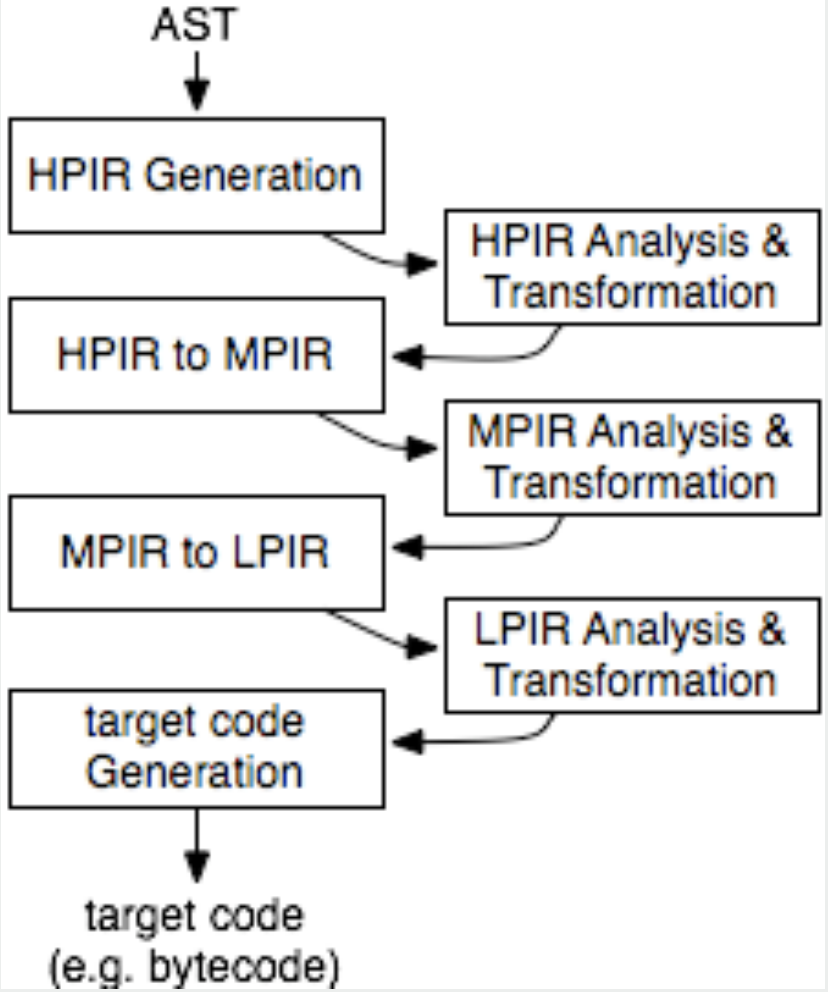
- The phaser construct addresses past limitations by supporting
  - General m→n synchronization patterns including barriers and point-to-point
  - Multiple modes for a task to register on a phaser (SIG, WAIT, SIG_WAIT)
  - Dynamic parallelism (set of registered tasks can vary dynamically)
  - A integrated “next” statement by a task to advance each phaser that it is registered on according to its registration mode
  - Single-instance statements that are executed once during phase transitions
  - A signal statement for “split-phase barriers”
  - Guaranteed deadlock avoidance

Example of HJ program

```
finish {
    final int nproc = nthreads;
    final phaser ph = new phaser();
    foreach (point [proc]:[0:nproc-1])
            phased (ph<hj.lang.phaserMode.SIG_WAIT>) {
        for (int o = 0; o <= 1; o ++) {
            int lim = (M-o) / 2;
            SORrunIter(G, o, lim, proc, nproc);
            next; // barrier operation
        }
    }
}
```


Parallel Intermediate Representation (PIR) Design

- A parallelism-aware intermediate representation
- Multiple level intermediate representation
  - address different requirements for program analysis and transformation
  - 3-level PIR implementation in our HJ compiler
- Portable design
  - Easy to extend available compiler infrastructures, e.g. Soot, LLVM.
  - HJ PIR extends from Soot JIMPLE IR



High-Level PIR

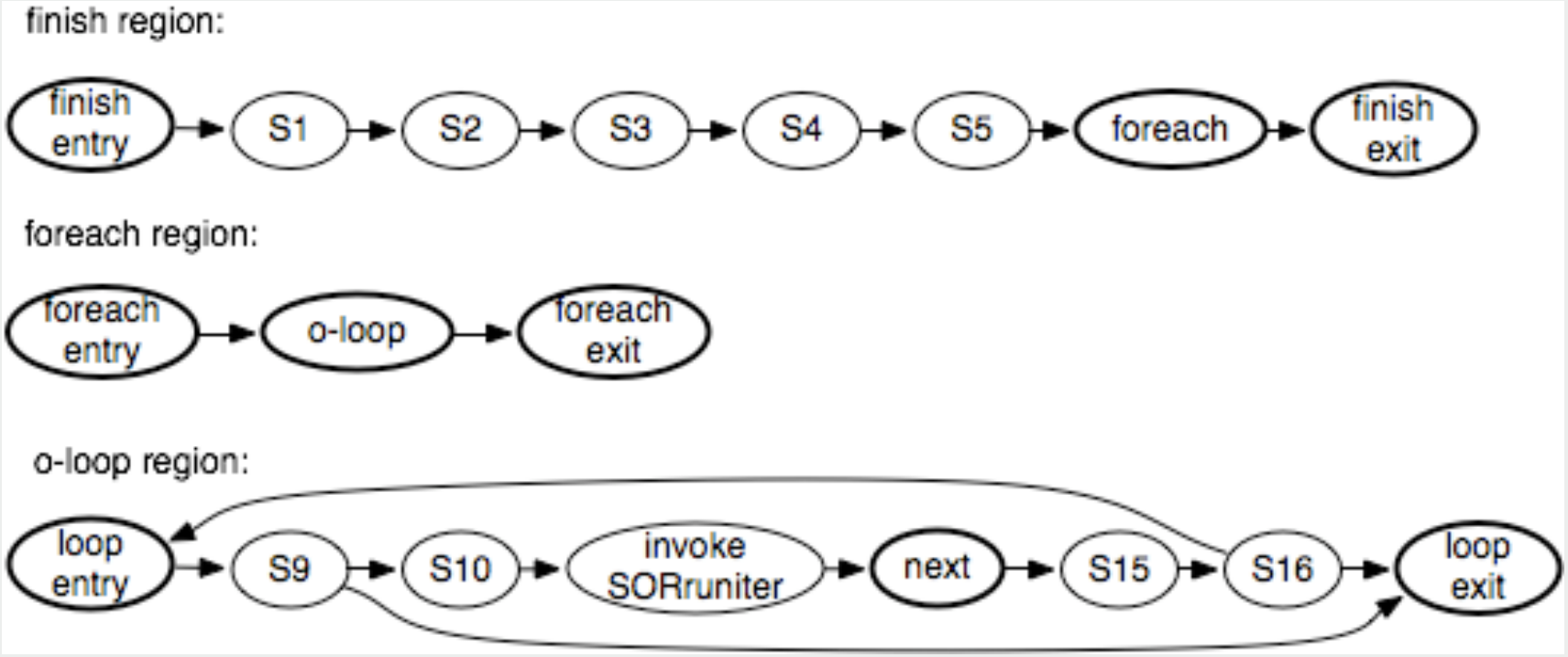
- High abstraction of parallel program
  - rich information for parallelism
  - all parallel constructs and objects have special IR annotations
  - usability: help compilation passes understand and transfer parallel program easily
- Hierarchical Representation
  - Region Control-Flow Graph (RCFG)
  - Region Structure Tree (RST)
- Application
  - program analysis and transformations that need the information about the structure of parallel program, e.g. the organization of parallel constructs, iteration space of parallel loops.
  - examples: may-happen-in-parallel analysis, chunking parallel loops.

Example of HPIR (Pseudo code)

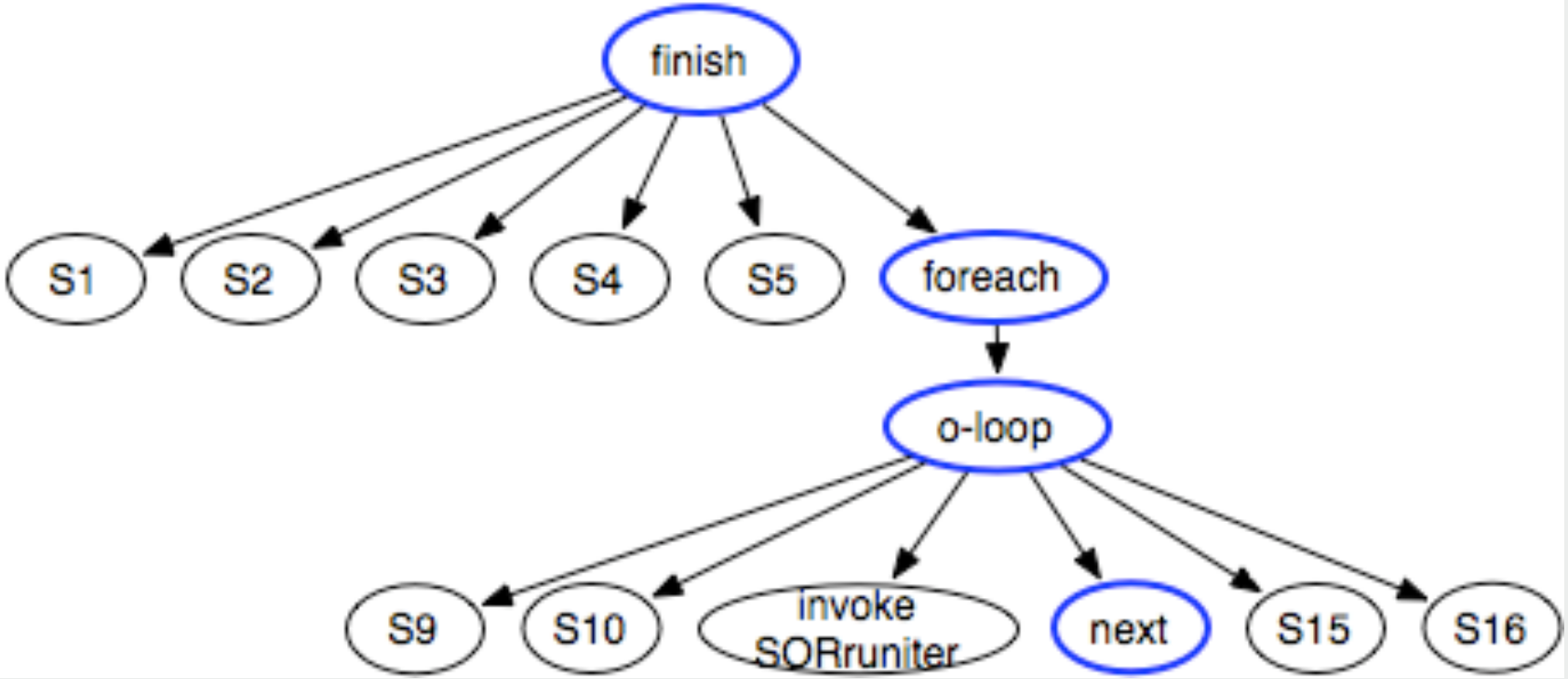
```
FinishRegionEntry;
  nproc = nthreads;
  ph = new phaser();
  specialInvoke ph.<init>();
  i0 = nproc - 1;
  ForeachRegionEntry iter(proc) region(0:i0)
                phasers(ph, SIG_WAIT>)
    LoopRegionEntry iter(o) region([0:1])
      if (o > 1) goto LoopRegionExit;
      lim = (M-o) / 2;
      staticInvoke SORrunIter(G, o, lim, proc, nproc);

      NextOperation;
      o = o + 1;
      goto LoopRegionEntry;
    LoopRegionExit;
  ForeachRegionExit;
FinishRegionExit;
```


Example of RCFG



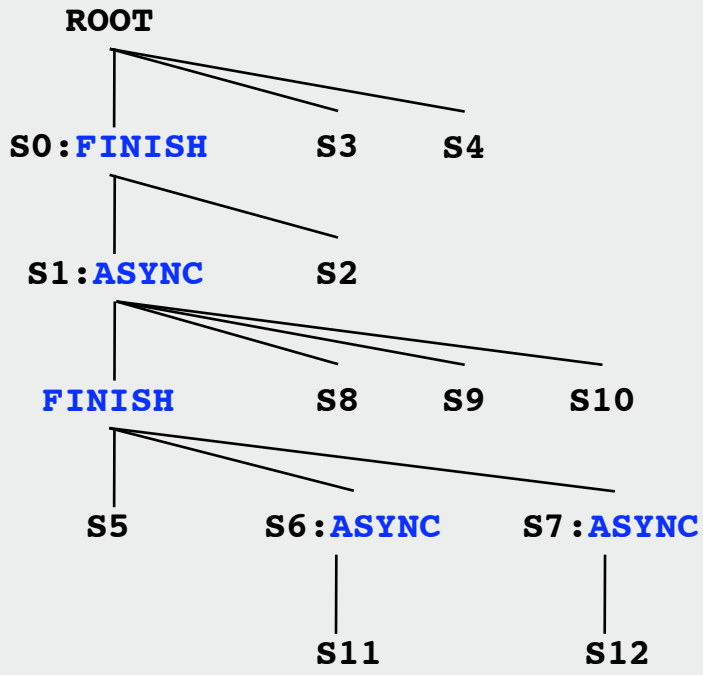
Example of RST



HPIR Case Study: May-Happen-In-Parallel Analysis

```
S0: finish {
    S1: async {
        finish {
                S5: ...
                S6: async S11
                S7: async S12
                ...
        }
        S8: ...
        S9: ...
        S10: ...
    }
    S2: ...
}
S3: ...
S4: ...
```

Program Structure Tree




```
MHP(S4, S11) = false
MHP(S2, S12) = true
```

"May-happen-in-parallel analysis of X10 programs", S.Agarwal et al. PPoPP 2007.

Middle-Level PIR

- Flat Representation
  - lowering high level parallel constructs, e.g. foreach, forall
  - flatten control flow that only uses region labels to annotate program closures, e.g. async, finish, isolated
  - parallelism: happen-before, mutual-exclusion
- Application
  - program transformations that work on sequential code but still need to be aware of parallelism and synchronization (e.g. memory model issues)
  - example: load elimination, delegated isolation

Example of MPIR (Pseudo code)

```
FinishRegionEntry;
  nproc = nthreads;
  ph = new phaser();
  specialInvoke ph.<init>();
  i0 = nproc - 1;
  proc = 0;
entry_0:
  if (proc > i0) goto exit_0;
  AsyncRegionEntry phasers(ph, SIG_WAIT>)
    o = 0;
entry_1:
    if (o > 1) goto exit_1;
    lim = (M-o) / 2;
    staticInvoke SORrunIter(G, o, lim,
   proc, nproc);

    NextOperation;
    o = o + 1;
    goto entry_1;
exit_1:
  AsyncRegionExit;
  proc = proc + 1;
  goto entry_0;
exit_0:
FinishRegionExit;
```


MPIR Case Study: Inter-procedural load elimination for parallel program

```
1: void main() {
2:   p.x = …
3:   s.w = …
4:   finish { //f
5:     if (…) {
6:       async { //async_1
7:         p.x = …
8:         isolated { q.y = …; … = q.y }
9:         … = p.x
10:      }
11:  }
12:    … = p.x
13:    foo()
14:  }
```

Can be replaced by a scalar

Cannot be replaced by a scalar

Effectiveness of scalar replacement algorithm is critically dependent on PIR primitives for parallelism

"Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs". Rajkishore Barik, Vivek Sarkar. PACT 2009.

Low-Level PIR

- Low level representation
  - lowering all parallel constructs into the normal compiler IR and runtime APIs
  - normal 3-address IR that can be directly mapped to Java bytecode
- Low level optimizations
  - optimization of runtime book-keeping, e.g. work-stealing schedulers
  - sequential code optimizations, e.g. LICM, dead code elimination

Example of LPIR (Pseudo code)

```
act = staticInvoke hj.runtime.getCurrentActivity();
virtualInvoke act.startFinish();
  nproc = nthreads;
  ph = new phaser();
  specialInvoke ph.<init>();
  i0 = nproc - 1;
  proc = 0;
entry_0:
  if (proc > i0) goto exit_0;
  a0 = new Activity0;
  specialInvoke a0.<init>(ph, this, proc, nproc);
  act = staticInvoke
   hj.runtime.getCurrentActivity();
  place = virtualInvoke act.getPlace();
  virtualInvoke place.runAsync(a0);
  proc = proc + 1;
  goto entry_0;
exit_0:
act = staticInvoke hj.runtime.getCurrentActivity();
virtualInvoke act.stopFinish();
```


Example of LPIR (contd.)

```
public class Activity0 {// Activity class
  public phaser ph;
  public Sor thisobj;
  public int proc;
  public int nproc;

  public void runHjTask() {// async
   closure
    o = 0;
entry_1:
    if (o > 1) goto exit_1;
    lim = (thisobj.M-o) / 2;
    staticInvoke SORrunIter(thisobj.G, o,
                        lim, proc, nproc);

    virtualInvoke ph.doNext();
    o = o + 1;
    goto entry_1;
exit_1:
  }
```

```
  public void <init>(phaser,
    Sor, int, int) {
     this.ph = @param0;
     this.thisobj = @param1;
     this.proc = @param2;
     this.nproc = @param3;
   }
}
```


Summary

- A multi-level IR system for representing the parallel program
- Address the different issues related to the parallelism:
  - lightweight task creation & synchronization, point-to-point synchronization, mutual exclusion and locality control
- Ongoing research
  - dataflow analysis
  - locality optimization
  - C compiler
- More detail: HJ Poster (5pm ~ 6:30pm) & HJ Demonstration in OOPSLA (Tues, Wed and Thu)