
COMP 322: Principles of Parallel Programming
**Lecture 9: Successive Over-Relaxation,
Barriers, HJ Phasers (Chapter 6)**

Fall 2009

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu



Summary of Previous Lecture

- **Google's MapReduce Framework**
 - Map, Reduce primitives
 - Word Count Example
 - Execution Overview
 - Fault Tolerance

Acknowledgments for Today's Lecture

- Course text: "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
 - Includes resources available at <http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html>
- "Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization", Jun Shirako, David M. Peixotto, Vivek Sarkar, William N. Scherer III
 - <http://www.cs.rice.edu/~vs3/PDF/SPSS08-phasers.pdf>

Summary of HJ Constructs Studied thus far

- Task creation
 - forall
 - async statement
 - async expression with future<T> return value
- Task termination
 - finish statement
 - force operation on future<T>
- Task isolation
 - isolated statement (undirected synchronization)

Today's lecture: Barrier and point-to-point directed synchronization using HJ *phasers*

Case Study: Successive Over-Relaxation (SOR), pp. 174-186

- SOR is an algorithmic technique for solving systems of differential equations
- Iterative approach for 2D SOR
 - Assign fixed values to boundary elements
 - Compute new array by replacing each interior element in old array with the average of its four neighbors
 - Swap pointers to old and new arrays
 - Continue iterating until difference between old and new values is $<$ threshold for all array elements

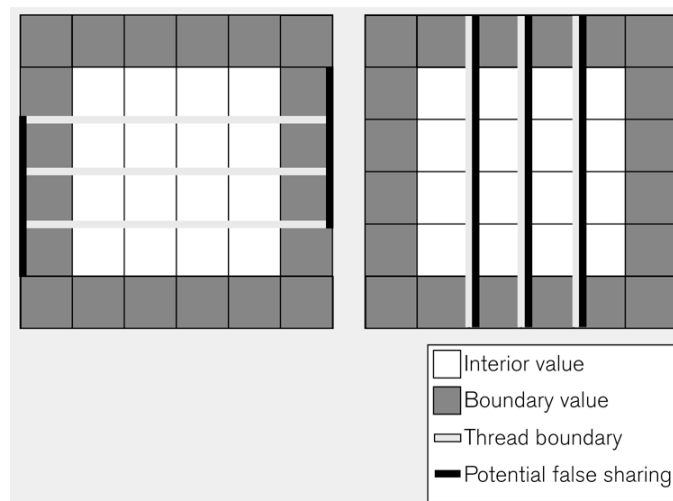
Figure 6.14 A 2D relaxation replaces—on each iteration—all interior values by the average of their four nearest neighbors.

1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0

□ Interior value
■ Boundary value

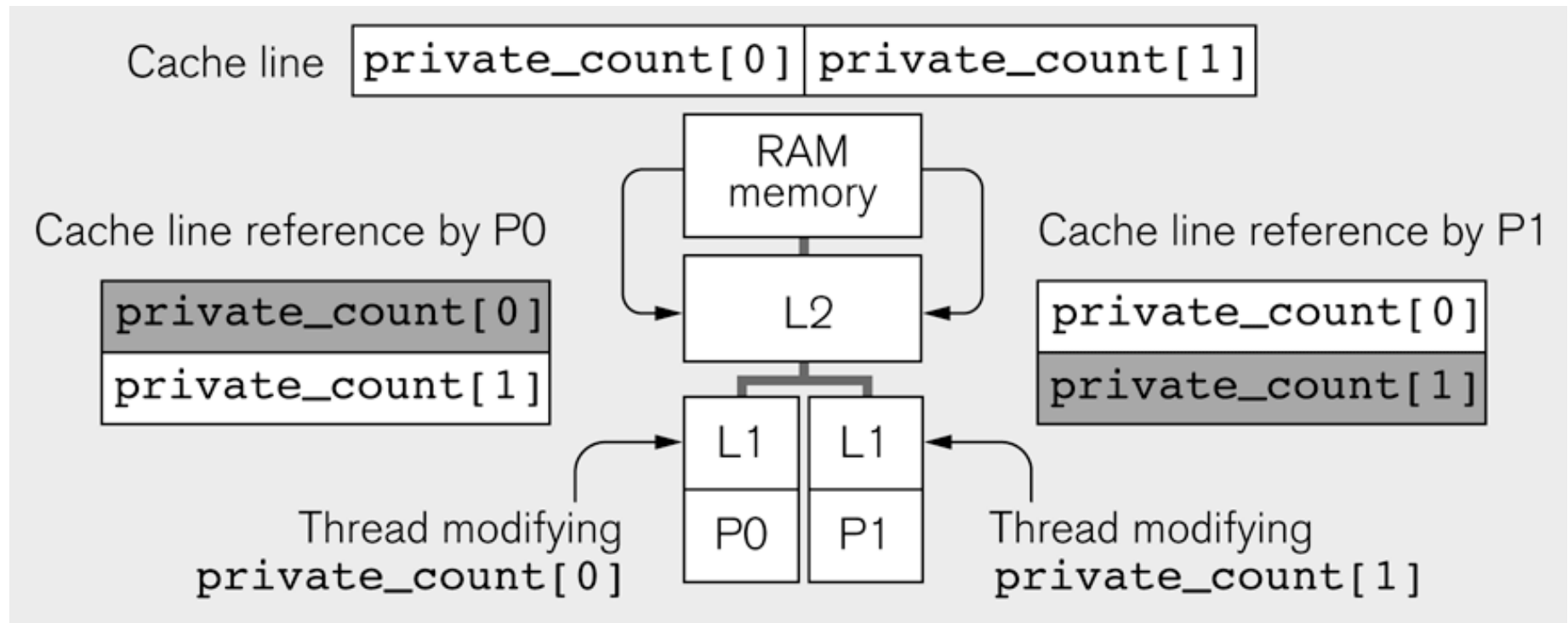
Work Assignment

- Static work assignment is better than dynamic work assignment for 2-D SOR
 - Work per element is regular and static
 - Textbook studies a vertical 1D block decomposition to minimize the number of points where “false sharing” may occur
 - Array padding can be used to completely eliminate false sharing
- Figure 6.15: (Left) Vertical 1D block decomposition, (Right) Horizontal 1D block decomposition



Simple Example of False Sharing

- Figure 1.13 (pg 22). False Sharing. A cache line moves from RAM to L1 cache when Thread 0 modifies `private_count[0]`. When thread 1 modifies `private_count[1]`, the cached copy in Thread 0 is invalidated and communicated to Thread 1.



HJ code for 2D SOR

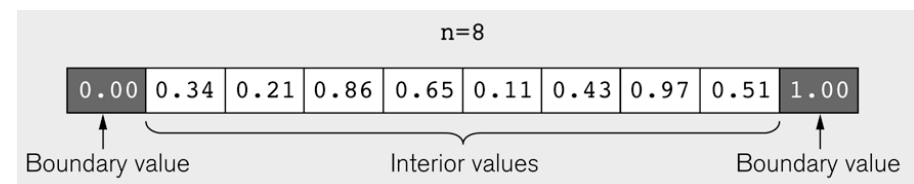
(See pg. 176 for Pthreads version)

```
delta = 0.0; // shared field
forall (point [id] : [0:t-1] ) {
    start = id*rowsPerThread + 1;
    do {
        for (int i=start; i<start+rowsPerThread; i++)
            for (int j=1; j<n+1; j++) {
                average = (myVal[i-1][j] + myVal[i][j+1] +
                           myval[i+1][j] + myVal[i][j-1])/4;
                mydelta = Math.Max(myDelta, Math.Abs(average-myVal[i][j]));
                myNew[i][j] = average;
            } // for i
        temp = myNew; myNew = myVal; myVal = temp; // local variables
        isolated if (myDelta > delta) delta = myDelta;
        next; // barrier
    }
} while(delta < threshold)
} // forall
```

HJ code for 1D SOR w/o Split-phase Barrier (See Figure 6.20, pg. 181 for Pthreads version)

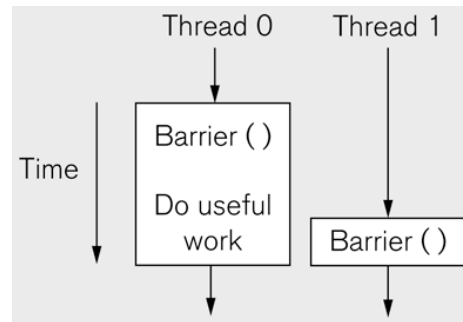
```
forall (point [index] : [0:t-1]) {
    double[] myval = val; double[] myNew = new; double[] temp = null;
    int n_per_thread = n/t;
    int start = index*n_per_thread;
    for (int i=0; i<iterations; i++) {
        for (int j=start; j<start+n_per_thread; j++)
            myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
        temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
        next;
    } // for
} // forall
```

Figure 6.19 A 1D over-relaxation replaces—on each iteration—all interior values by the average of their two nearest neighbors.



Overlapping Synchronization with Computation: Split-phase barrier

- Figure 6.17. It's often profitable to do useful work while waiting for some long-latency operation to complete.



- Figure 6.18. A split-phase barrier allows a thread to do useful local work while waiting for the other threads to arrive at the barrier.

```
// Initiate synchronization
barrier.arrived();

// Do useful work

// Complete synchronization
barrier.wait();
```

HJ signal statement

- `signal; // barrier.arrived();`
 - Indicates that this task has completed shared work and is ready to enter the barrier
 - Computation between `signal;` and `next;` statements represents local work executed while waiting for other tasks to reach barrier
 - `signal;` is non-blocking , `next;` is blocking
- `signal` supports split-phase barrier as a primitive statement
 - As opposed to implementing it using lower-level primitives as in Figures 6.22 and 6.24

HJ code for 1D SOR w/ Split-phase Barrier (See Figure 6.21 for Pthreads version)

```
forall (point [index] : [0:t-1]) {
    double[] myval = val; double[] myNew = new; double[] temp = null;
    int n_per_thread = n/t; int start = index*n_per_thread;
    for (int i=0; i<iterations; i++) {
        // Update local boundary values
        int j=start;                myNew[j]=(myVal[j-1] + myVal[j+1])/2.0;
        j=start+n_per_thread-1; myNew[j]=(myVal[j-1] + myVal[j+1])/2.0;
        signal; // Start barrier
        // Update local interior values
        for (int j=start+1; j<start+n_per_thread-1; j++)
            myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
        temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
        next; // Wait for all tasks to enter barrier
    } // for
} // forall
```

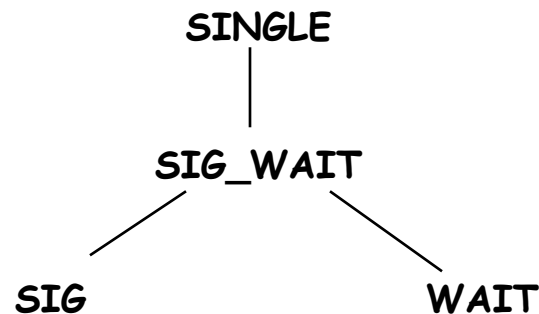
HJ Phasers: Collective and Point-to-point Synchronization with Dynamic Parallelism

phaser ph = new phaser(MODE);

- Allocate a phaser, register current task with it according to MODE. Phase 0 of ph starts.
- MODE can be SIG, WAIT, SIG_WAIT (default) or SINGLE
- *Finish Scope rule:* phaser ph cannot be used outside the scope of its immediately enclosing finish operation

async phased (MODE1(ph1), MODE2(ph2), ...) S

- Spawn S as an asynchronous (parallel) activity that is registered on phasers ph1, ph2, ... according to MODE1, MODE2, ...
- *Capability rule:* parent activity can only transmit phaser capabilities to child activity that are a subset of the parent's capabilities, according to the lattice:



Phaser Primitives (contd)

next;

- Advance *each* phaser that activity is registered on to its next phase; semantics depends on registration mode

next <stmt> // next-with-single statement

- Execute next operation with single instance of <stmt> during phase transition
- All activities executing a next-with-single statement must execute the same static statement

ph.signal();

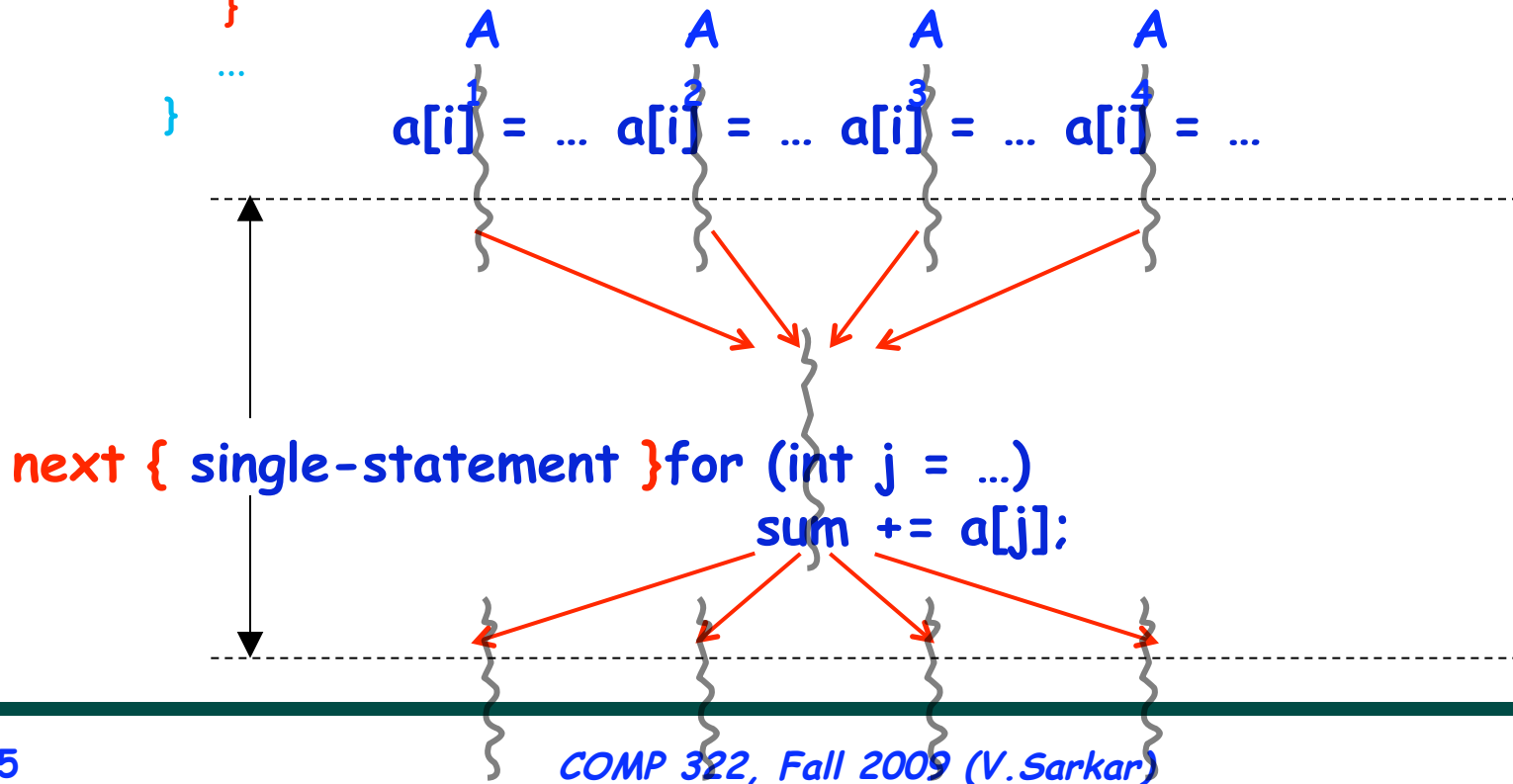
- Nonblocking operation that signals completion of work by current activity for this phase of phaser ph
- Error if activity does not have signal capability

signal;

- Perform *ph.signal()* on each phaser that activity is registered on with a signal capability

Single Statements

```
phaser ph = new phaser(SINGLE);  
for (point [i] : [0:n-1])  
  async phased(ph<SINGLE>) {  
    a[i] = foo(...);  
    next {  
      // Only one activity executes here  
      for(int j = 0; j < n; j++) sum += a[j];  
    }  
  }  
  ...  
}
```



Split Phase Barrier with Single Stmt

```
phaser ph = new phaser(SINGLE);  
foreach (point [i] : [0:n-1]) phased(ph<SINGLE>) {  
    a[i] = foo(...);  
    signal;  
    localWork();  
    next {  
        // Only one activity executes here  
        for(int j = 0; j < n; j++) sum += a[j];  
    }  
    ...  
}
```

A A A A

a[i] = ... a[i] = ... a[i] = ... a[i] = ...

signal signal signal signal

local local local local

next { single-statement } for (int j = ...) sum += a[j];

Master activity to process barrier

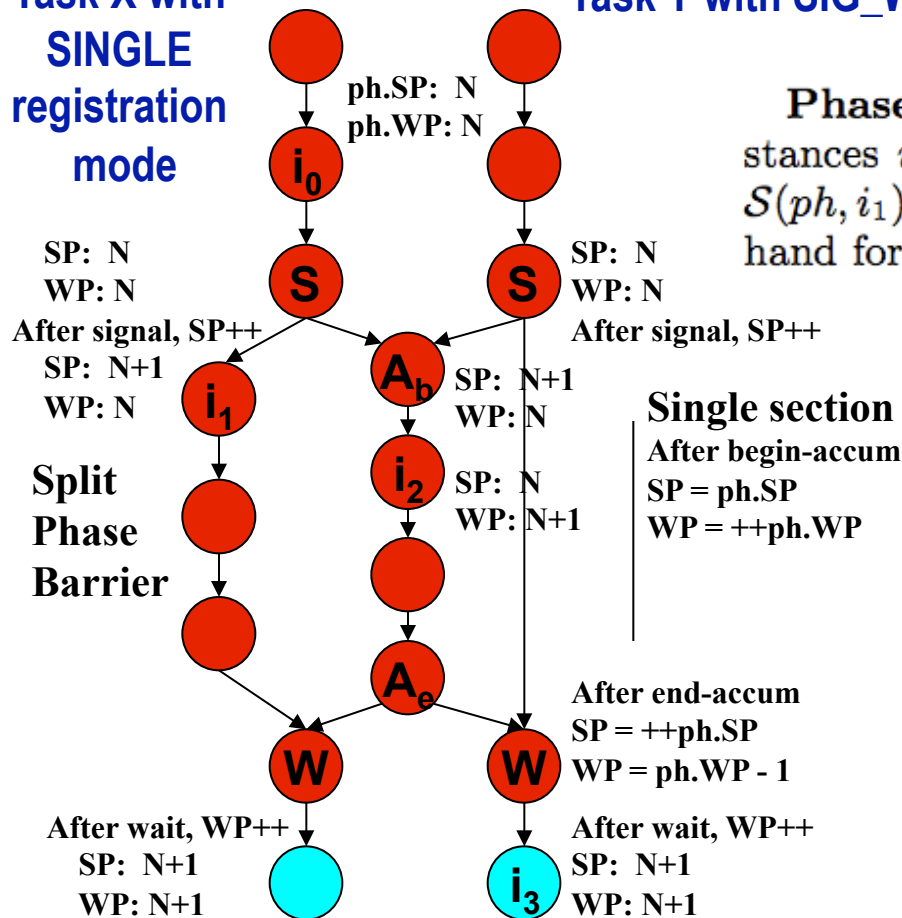
HJ code for 1D SOR w/ Split-phase Barrier and Single Statement

```
forall (point [index] : [0:t-1]) {
    double[] myval = val; double[] myNew = new; double[] temp = null;
    int n_per_thread = n/t; int start = index*n_per_thread;
    for (int i=0; i<iterations; i++) {
        // Update local boundary values
        int j=start;                myNew[j]=(myVal[j-1] + myVal[j+1])/2.0;
        j=start+n_per_thread-1; myNew[j]=(myVal[j-1] + myVal[j+1])/2.0;
        signal; // Start barrier
        // Update local interior values
        for (int j=start+1; j<start+n_per_thread-1; j++)
            myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
        temp = myNew; myNew = myVal; myVal = temp; // swap(myNew, myVal)
        next { System.out.println(myVal[n/2]); } // Single statement
    } // for
} // forall
```

Phaser operations and states on Dynamic Computation DAG

Task X with SINGLE registration mode

Task Y with SIG_WAIT registration mode



Phase-ordering property: Given two instruction instances i_1 and i_2 in the DCD, if \exists a phaser ph such that $S(ph, i_1) < W(ph, i_2)$ then $i_1 \prec i_2$, where $i_1 \prec i_2$ is shorthand for “ i_1 precedes i_2 ”.

Operation	Registration Mode		
	<i>signal-wait-next</i> or <i>signal-wait</i>	<i>signal-only</i>	<i>wait-only</i>
create	$S = 0$ $W = 0$	$S = 0$ $W = 0$	$S = \infty$ $W = 0$
async	$S = p.S$ $W = p.W$	$S = p.S$ $W = p.W$	$S = \infty$ $W = p.W$
signal	$S = S + 1$ $W = W$	$S = S + 1$ $W = W$	ERROR ERROR
wait	$S = S$ $W = W + 1$	ERROR ERROR	$S = \infty$ $W = W + 1$
begin-accum	$S = ph.S$ $W = ++ph.W$	$S = S$ $W = W$	$S = S$ $W = W$
end-accum	$S = ++ph.S$ $W = ph.W - 1$	$S = S$ $W = W$	$S = S$ $W = W$

Figure 6: Example of a Dynamic Computation DAG and its associated S (SP) and W (WP) values for the phaser ph . The figure shows two signal nodes (S), two wait nodes (W), a begin-accum (A_b), and end-accum (A_e) node.

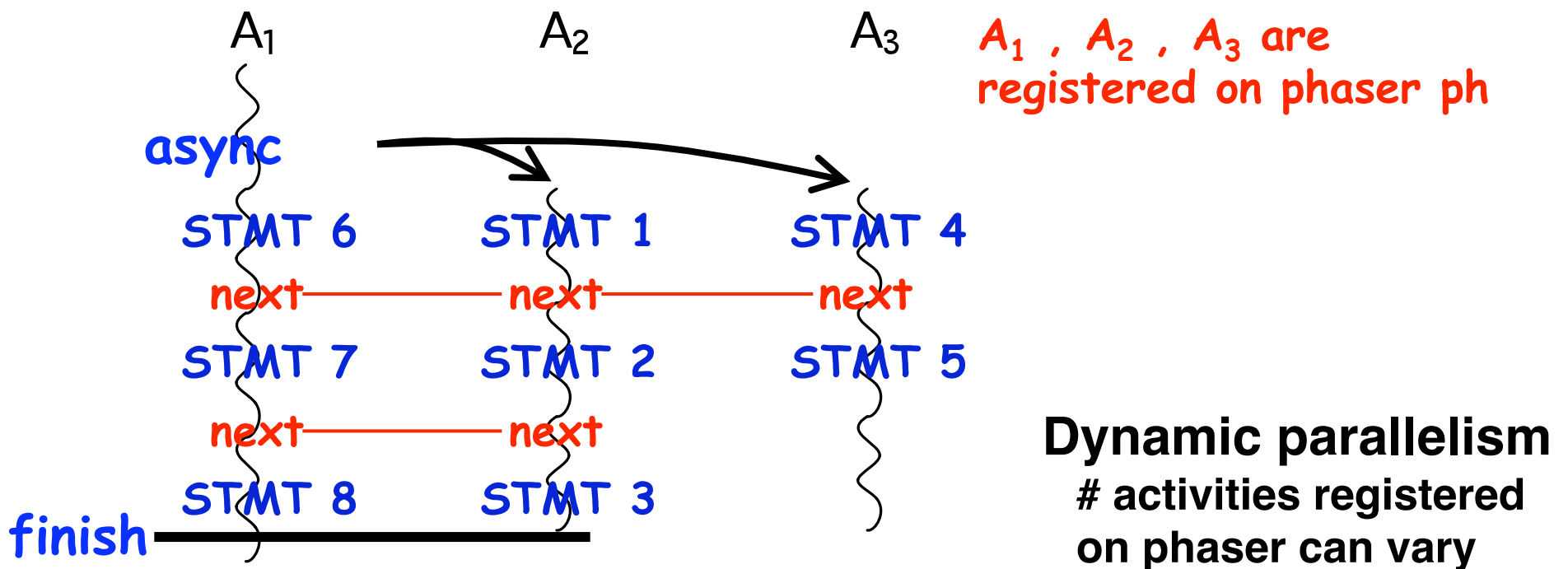
Table 2: Definition of the functions S and W . Here, $p.S$ is the value of S for the parent activity that executed the async and $ph.S$ is the private value of the signal phase for the phaser ph .

Using Phasers as Barriers with Dynamic Parallelism

```

finish {
  phaser ph = new phaser(); //A1
  async phased(ph) { STMT1; next; STMT2; next; STMT3; } //A2
  async phased(ph) { STMT4; next; STMT5; } //A3
                    STMT6; next; STMT7; next; STMT8; //A1
}

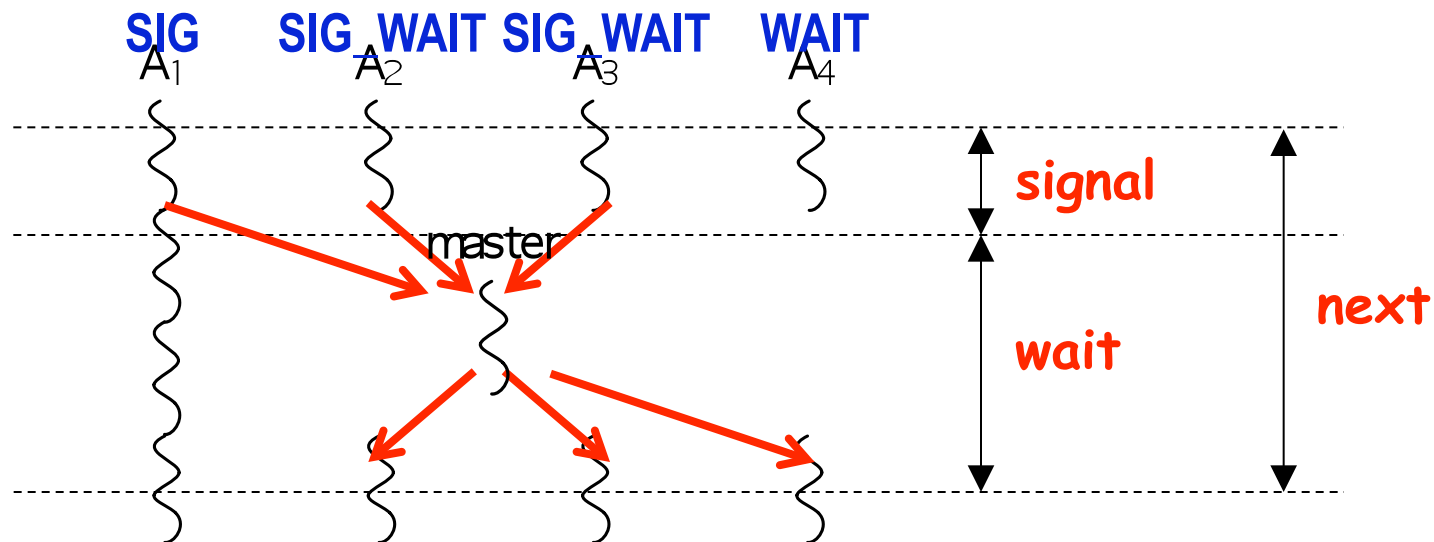
```



next / signal / wait

next = {
• Notify "I reached next" = **signal** (or **ph.signal()**)
• Wait for others to notify = **wait**

- Semantics of **next** depends on registration mode
 - **SIG_WAIT**: **next** = **signal** + **wait**
 - **SIG**: **next** = **signal** (Don't wait for any activity)
 - **WAIT**: **next** = **wait** (Don't disturb any activity)



- A master activity is selected from activities w/ wait capability
- It receives all signals and broadcasts a barrier completion notice
- Master can be fixed or unfixed

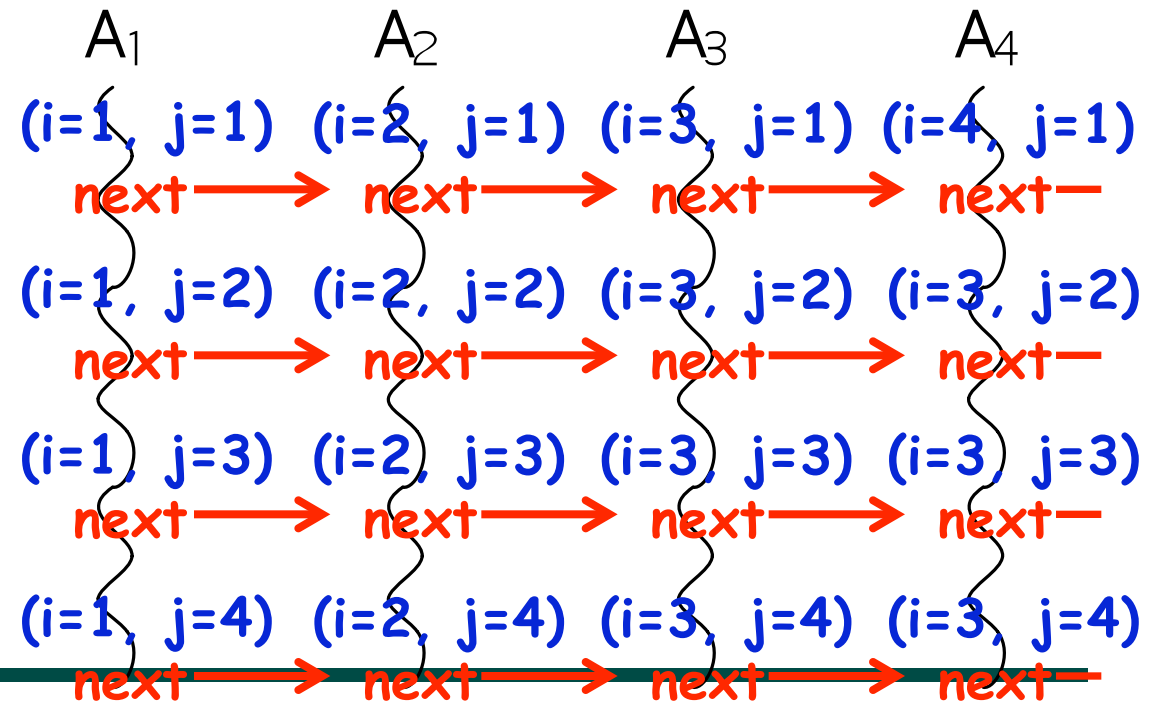
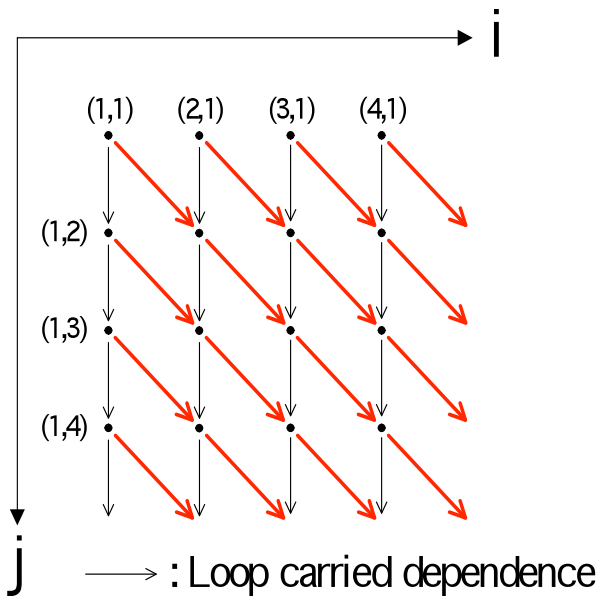
Example of Pipeline Parallelism with Phasers

```

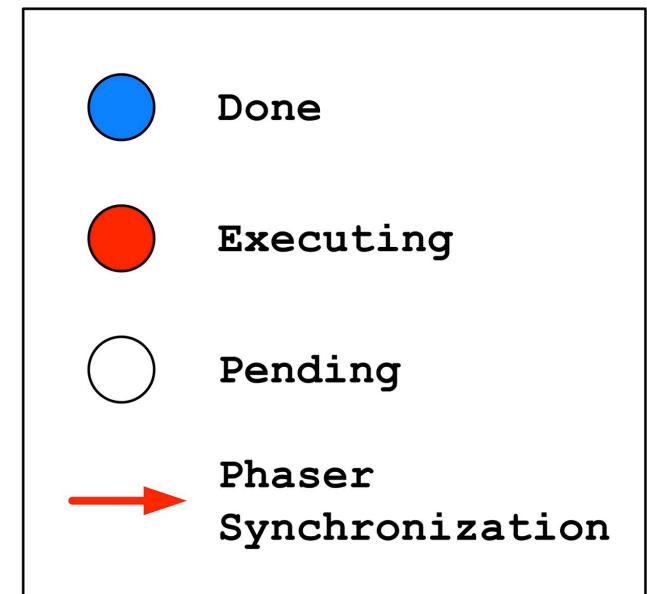
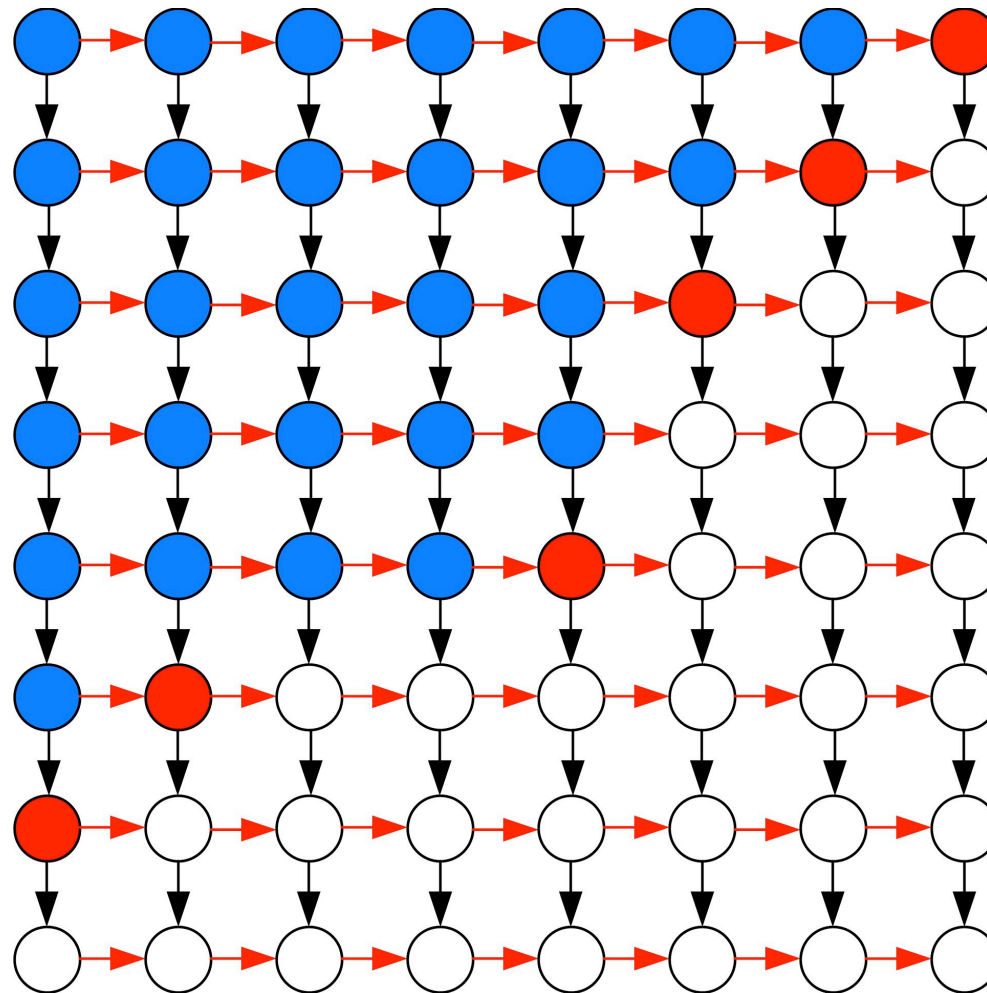
finish {
  phaser [] ph = new phaser[m+1];
  for (int i = 1; i < m; i++)
    async phased (ph[i]<SIG>, ph[i-1]<WAIT>){
      for (int j = 1; j < n; j++) {
        a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
        next;
      } // for
    } // finish
}

```

sig(ph[1]) sig(ph[2]) sig(ph[3]) sig(ph[4])
 wait(ph[0]) wait(ph[1]) wait(ph[2]) wait(ph[3])



Example of Pipeline Parallelism with Phasers (contd)



© Daniel Orozco, CAPSL 2008

Summary of Today's Lecture

- Case study: Successive Over-Relaxation
- HJ Phasers
 - SIG, WAIT, SIG_WAIT registration modes
 - next statement (barrier)
 - signal statement (split-phase barrier)
 - single statement (computation during phase transition)