Unifying Barrier and Point-to-Point Synchronization in OpenMP with Phasers

Jun Shirako, Kamal Sharma, and Vivek Sarkar

Rice University

Introduction

- **Synchronization in a parallel program**
  - Thread / task termination (worker to master synchronization)
    - Join operation
  - Directed synchronization
    - Collective-barrier, point-to-point synchronization
  - Undirected synchronization (mutual exclusion)
    - Lock, transactional memory
- **Directed synchronization in OpenMP**
  - OpenMP barrier
    - All-to-all synchronization
    - Overkill for a certain class of applications
- **Optimizing directed synchronization in OpenMP**
 - Phasers: Unified synchronization construct to support various synchronization patterns

Introduction

- **Habanero-Java**
  - Task parallel language at Rice University
  - http://habanero.rice.edu/hj

- **Phasers in HJ**
  - Synchronization among dynamically created tasks
  - Various synchronization pattern
    - Barriers, point-to-point sync
  - Reduction
  - Single statement
  - Some functionalities were added to Java 7 library
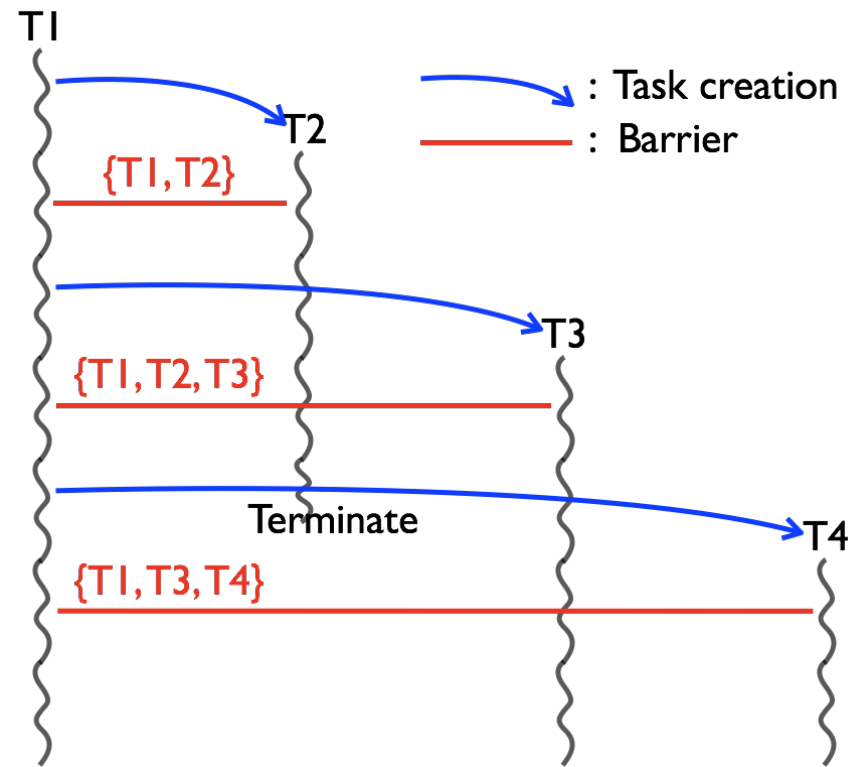
Outline

- **Introduction**
- **Case study for synchronization patterns**
  - Iterative averaging
  - Stencil algorithm
- **Phasers for optimized synchronization in OpenMP**
  - Thread-level phaser
  - Iteration-level phaser
- **Implementation**
  - Spin-lock with shared variable
- **Experimental results**
- **Conclusions**

Review of Some OpenMP Constructs

```
set_omp_num_threads(n);  // Set # threads for parallel regions

#pragma omp parallel      // Start parallel region by n threads
{
  foo();                  // All n threads execute foo

  #pragma omp barrier    // All-to-all synchronization by n threads
  ...
  #pragma omp for        // Parallel loop
  for (i = 0; i < m; i++) {
    ...
    #pragma omp barrier // Illegal usage of barrier
    ...
  }
  ...
}                         // End of parallel region
```
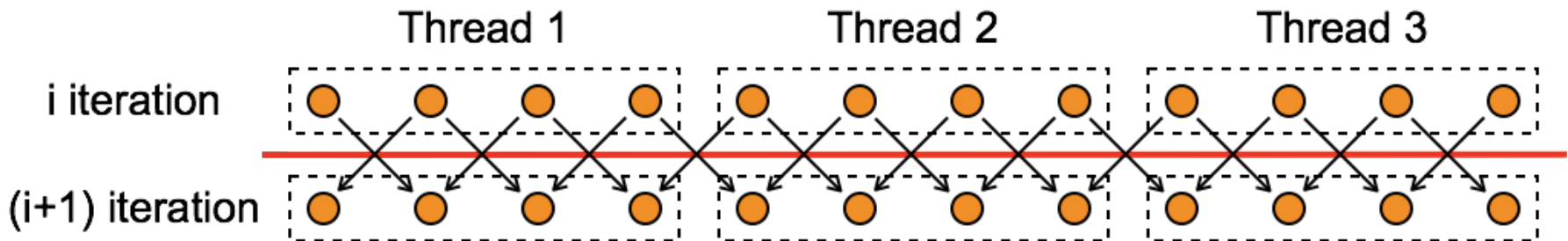
- **OpenMP barrier**
  - All threads synchronize with other threads
  - Not allowed to be in parallel for loops

Iterative Averaging

```
 1: #pragma omp parallel private(iter) firstprivate(newA, oldA)
 2: {
 3:   for (iter = 0; iter < NUM_ITERS; iter++) {
 4:     #pragma omp for schedule(static) nowait
 5:     for (j = 1; j < n-1; j++) {
 6:       newA[j] = (oldA[j-1] + oldA[j+1]) / 2.0;
 7:     }
 8:     double *temp = newA; newA = oldA; oldA = temp;
 9:     #pragma omp barrier
10: } }
```
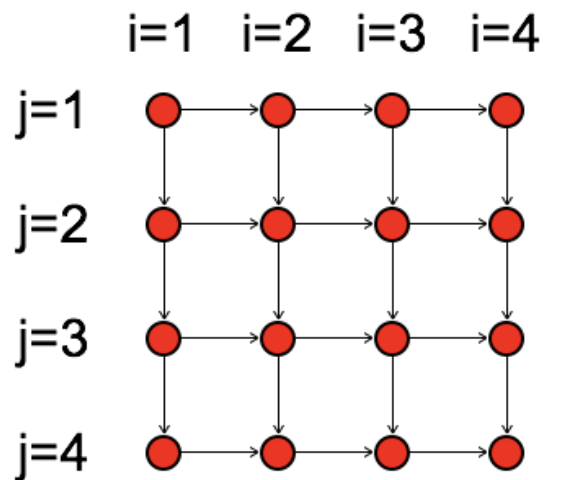

i iteration

(i+1) iteration
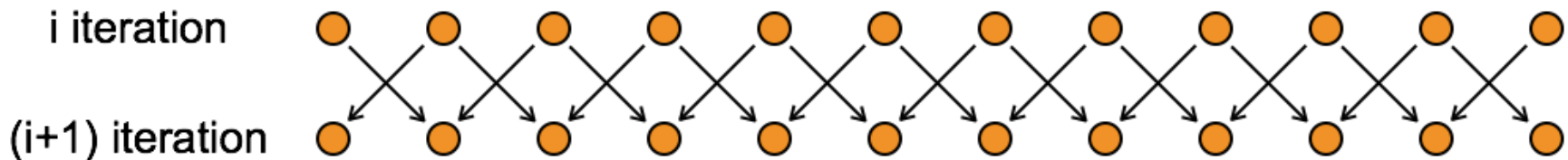
(a) Data dependence of 1-D averaging

Thread 1   Thread 2   Thread 3
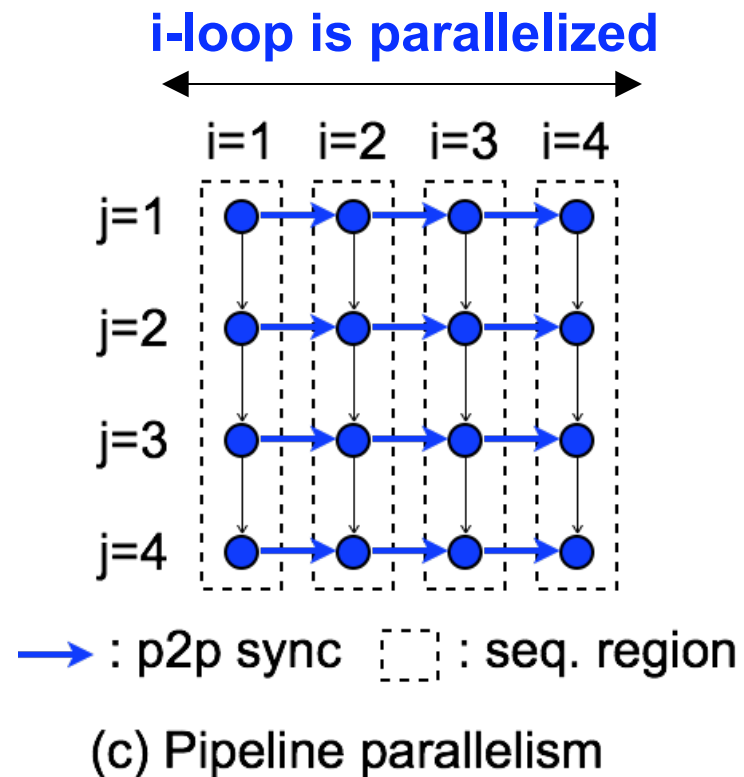
i iteration

(i+1) iteration

(b) Barrier synchronization

Stencil with Pipeline Parallelism

```
    // Sequential version
1: for (i = 1; i < n-1; i++) {
2:    for (j = 1; j < m-1; j++) {
3:      A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
4:                        A[i-1][j], A[i+1][j]);
5:    }
6: }
```
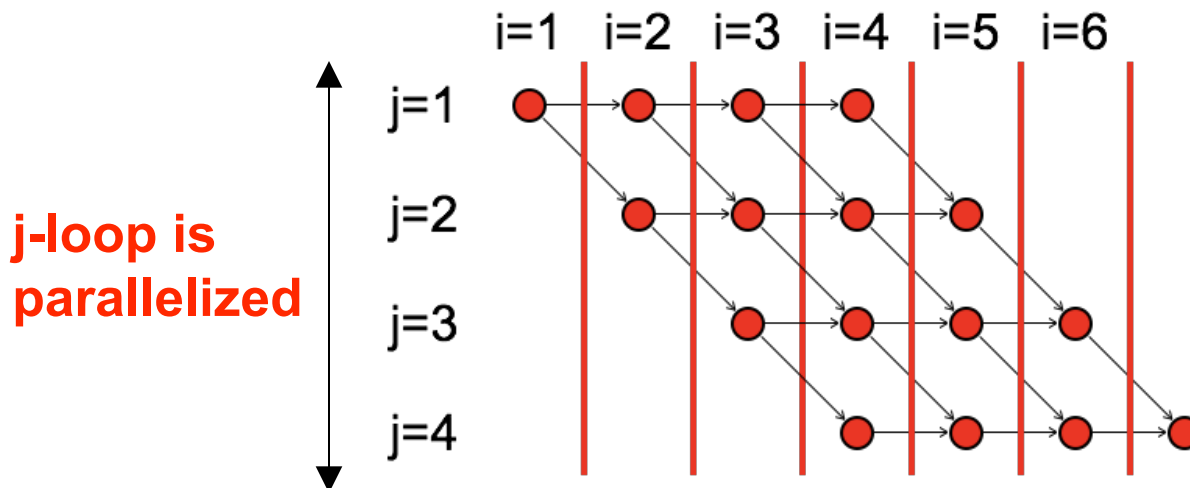
i-loop is parallelized

(a) Data dependence of stencil

: p2p sync   : seq. region

(c) Pipeline parallelism

Stencil with Wavefront Parallelism

```
1:  #pragma omp parallel private(i2)
2:  {
3:    for (i2 = 2; i2 < n+m-3; i2++) {   /* Loop skewing */
4:      #pragma omp for nowait
5:      for (j = max(1,i2-n+2); j < min(m-1,i2); j++) {
6:        int i = i2 - j;
7:        A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
8:                          A[i-1][j], A[i+1][j]);
9:      }
10:     #pragma omp barrier
11: } }
```

j-loop is parallelized

(b) Wavefront parallelism

8

Parallelized Code with Tiling

- **Polyhedral model**
  - Powerful mathematical model for loop transformations
    - Integrate loop fusion, skewing, interchange, etc.
  - Polyhedral parallelization framework
    - Pluto, Ptile

- **Loop tiling to extract locality and parallelism**
  - Fully permutable loop nest with ($\leq$, $\leq$, …, $\leq$) dependence vector
  - Naturally have (at least) 1-level pipeline parallelism
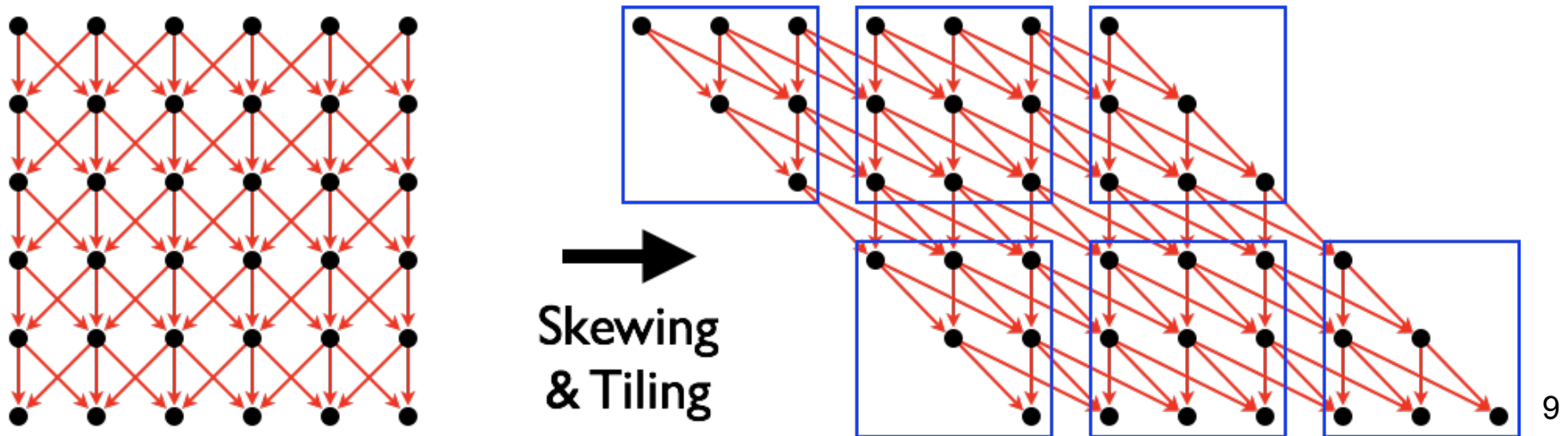
Skewing & Tiling

9

Outline

- **Introduction**
- **Case study for synchronization patterns**
  - Iterative averaging
  - Stencil algorithm
- **Phasers for optimized synchronization in OpenMP**
  - Thread-level phaser
  - Iteration-level phaser
- **Implementation**
  - Spin-lock with shared variable
- **Experimental results**
- **Conclusions**

Phasers

- **Two levels of phasers**
  - Thread-level phaser: Synchronization among OpenMP threads
  - Iteration-level phaser: Sync. among iterations of parallel loop
    - Task: An iteration of parallel loop

- **Registration**
  - **Register thread/task T_i on phaser ph_j with $mode_{i\_j}$**
    - Registration mode: {SIG, WAIT, SIG_WAIT}
    - Define capability that T_i has on ph_j

- **Synchronization**
  - **next:** Equivalent to **signal** followed by **wait**
  - **signal:** Non-blocking operation to notify "I reached the sync point"
 - **wait:** Blocking operation to wait for other tasks/threads' notification

- **Deregistration**
  - **Drop thread/task T_i from phaser ph_j**
    - T_i never attends synchronization on ph_j after deregistration

next / signal / wait

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

- • **Synchronization semantics depends on mode**
  - – SIG_WAIT: **next = signal + wait**
  - – SIG: **next = signal + no-op** (Don't wait for any task)
 - – WAIT: **next = no-op + wait** (Don't signal any task)

T1<**SIG**> T2<**SIG_WAIT**> T3<**SIG_WAIT**> T4<**WAIT**>

signal

next
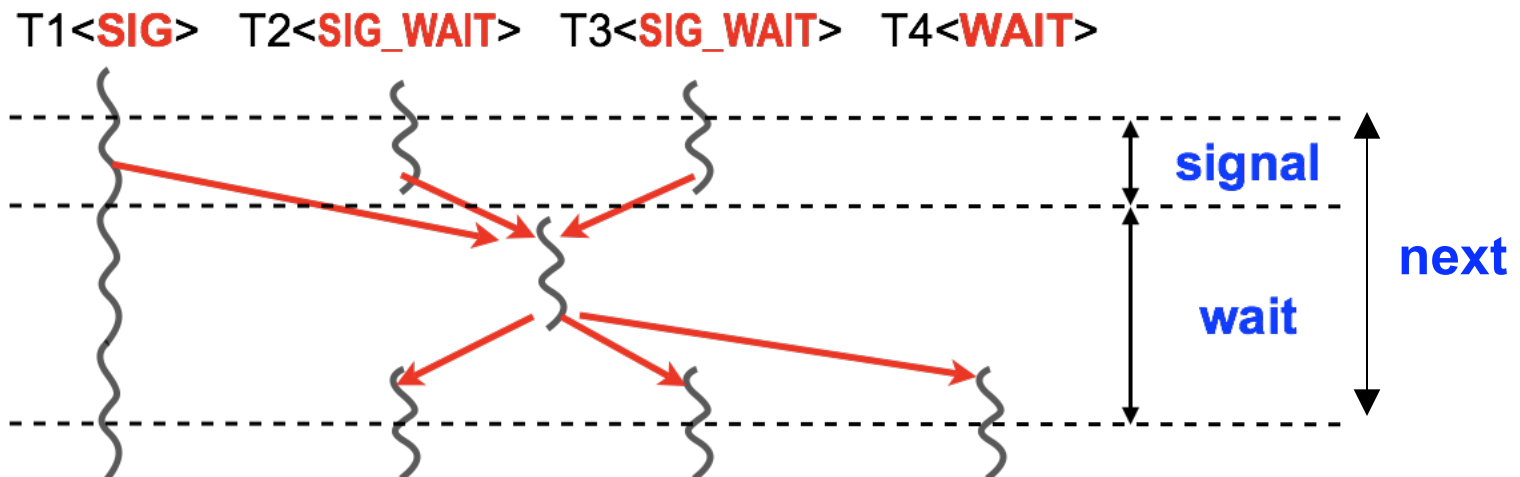
wait

- • **A master task is selected in tasks w/ wait capability**
- • **It receives all signals and broadcasts a barrier completion notice**

12

Thread-level Phaser API (iterative averaging ex.)

```
 1: /* Phaser allocation in serial region */
 2: phaser **ph = calloc(num_threads+2, sizeof(phaser *));
 3: for (i = 0; i < num_threads+2; i++) ph[i] = phaser_new();
 4:
 5: /* Registration */
 6: for (id = 0; id < num_threads; id++) {
 7:   phaserRegisterThread(ph[id], id, WAIT);    // Wait left neighbor
 8:   phaserRegisterThread(ph[id+1], id, SIG);
 9:   phaserRegisterThread(ph[id+2], id, WAIT); // Wait right neighbor
10: }
11: /* Parallel execution with phaser synchronization */
12: #pragma omp parallel private(iter) firstprivate(newA, oldA)
13: {
14:   for (iter = 0; iter < NUM_ITERS; iter++) {
15:     #pragma omp for schedule(static) nowait
16:     for (j = 1; j < n-1; j++) {
17:       newA[j] = (oldA[j-1] + oldA[j+1])/2.0;
18:     }
19:     double *temp = newA; newA = oldA; oldA = temp;
20:     #pragma omp next
21: } }
22: /* Deregistration to change synchronization pattern */
23: dropPhasersAll();
```

Thread-level Phaser API (iterative averaging ex.)

```
12: #pragma omp parallel private(iter) firstprivate(newA, oldA)
13: {
14:    for (iter = 0; iter < NUM_ITERS; iter++) {
15:       #pragma omp for schedule(static) nowait
16:       for (j = 1; j < n-1; j++) {
17:          newA[j] = (oldA[j-1] + oldA[j+1])/2.0;
18:       }
19:       double *temp = newA; newA = oldA; oldA = temp;
20:       #pragma omp next
21: } }
```


(b) Barrier synchronization

(c) Point-to-point synchronization

14
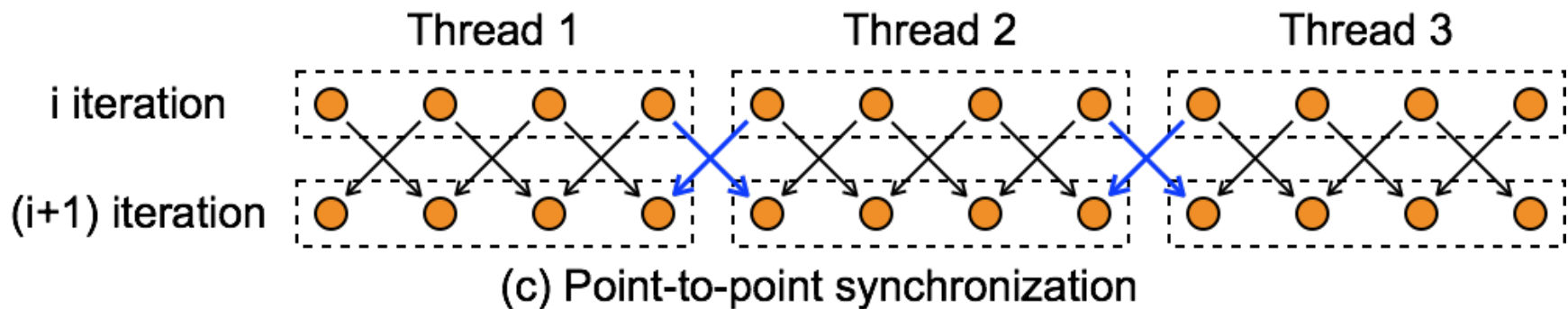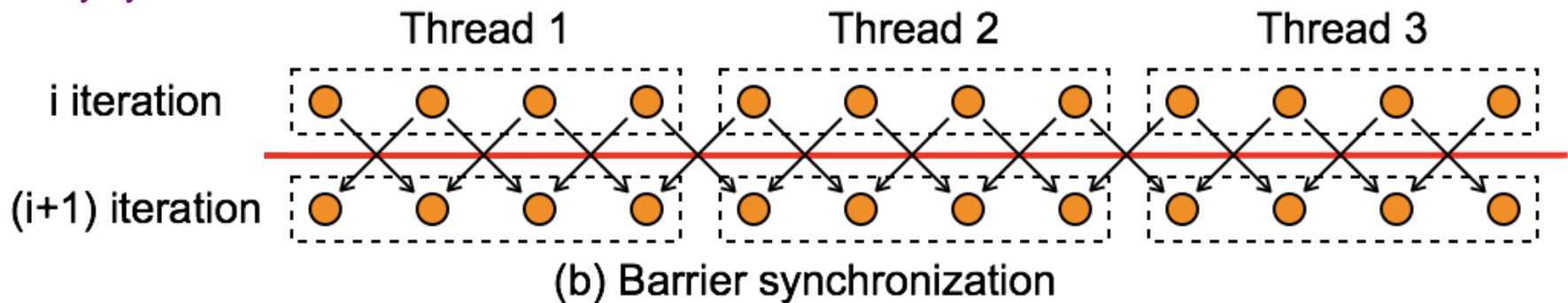
Iteration-level Phaser

- **Synchronization among iterations of parallel loop**
  - Higher level of abstraction
    - Express data dependence among iterations
    - signal / wait / next directives are used within parallel for loops
  - Less flexibility in synchronization pattern than thread-level
    - Direction of synchronization must be one-way (left-to-right)
      - Avoid deadlock
      - Loop chunking can relax this constraint
- **Extension to general OpenMP 3.0 tasks**
  - Synchronization in the presence of dynamic task parallelism
    - Nature of original phaser in Habanero-Java
  - Will be addressed in future work

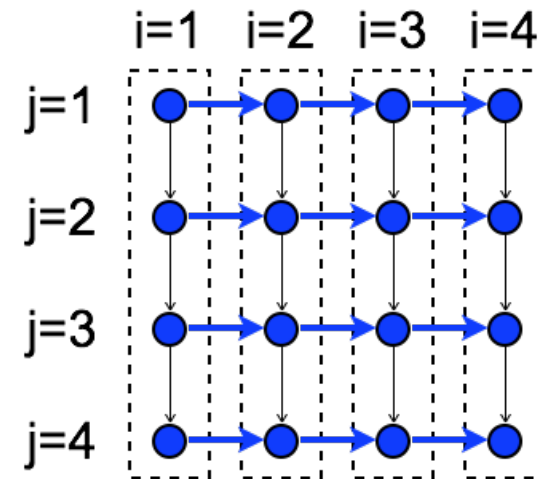# Iteration-level Phaser API (iterative averaging ex.)

```
 1: /* Phaser allocation in serial region */
 2: phaser **ph = calloc(n+1, sizeof(phaser *));
 3: for (i = 0; i < n+1; i++) ph[i] = phaser_new();
 4:
 5: /* Registration */
 6: for (i = 0; i < n; i++) {
 7:    /* Sync direction from left to right */
 8:    phaserRegisterIteration(ph[i], i, WAIT);   // Wait left neighbor
 9:    phaserRegisterIteration(ph[i+1], i, SIG); // Signal right neighbor
10: }
11:
12: /* Parallel execution with phaser synchronization */
13: #pragma omp parallel
14: {
15:    #pragma omp for private(j) schedule(static, 1)
16:    for (i = 1; i < n-1; i++) {
17:      for (j = 1; j < m-1; j++) {
18:        #pragma omp wait
19:        A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
20:                          A[i-1][j], A[i+1][j]);
21:        #pragma omp signal
22:    } }
23: }
24: dropPhasersAll(); /* Deregistration */
```

Iteration-level Phaser API (iterative averaging ex.)

```
13: #pragma omp parallel for private(j) schedule(static, 1)
14: for (i = 1; i < n-1; i++) {
15:    for (j = 1; j < m-1; j++) {
16:       #pragma omp wait
17:       A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
18:                         A[i-1][j], A[i+1][j]);
19:       #pragma omp signal
20: } }
```
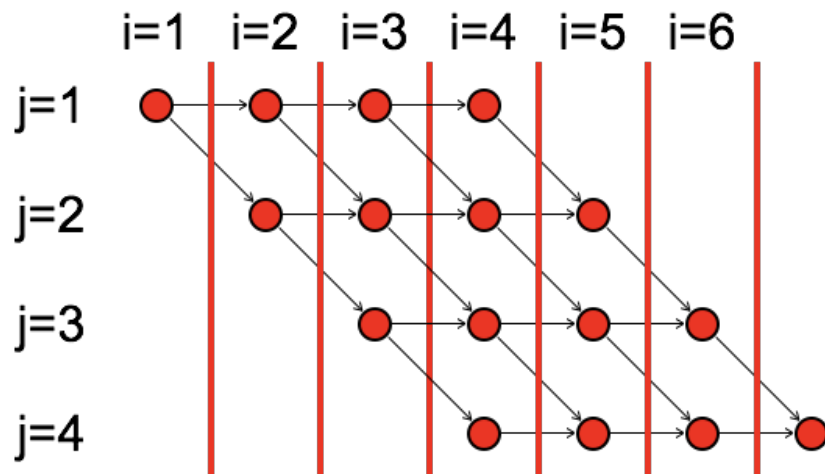

(b) Wavefront parallelism

→ : p2p sync   ⌐⌐⌐⌐ : seq. region

(c) Pipeline parallelism

Pipeline parallelism: Better synchronization efficiency & data locality

Outline

- **Introduction**
- **Case study for synchronization patterns**
  - Iterative averaging
  - Stencil algorithm
- **Phasers for optimized synchronization in OpenMP**
  - Thread-level phaser: SPMD-style
  - Iteration-level phaser: High-level abstraction
- **Implementation**
  - Spin-lock with shared variable
- **Experimental results**
- **Conclusions**

Local-spin Implementation
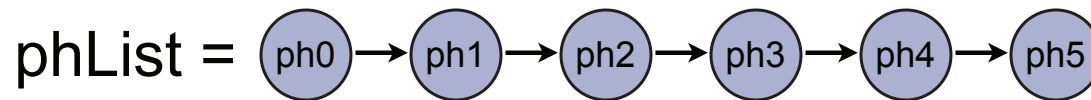
```
1: typedef struct _phaser {
2:     int id;
3:     // Contains Sig/Wait objects
4:     List *sigList, *waitList;
5:
6:     volatile int mSigPhase;
7:     int mWaitPhase;
8:     int masterId;
9:
10:     // Customized for single signaler
11:     int numSig, singleSigId;
12: } phaser;
13:
```

```
14: typedef struct _sig {
15:     int id; // Thread/task
16:     mode md;
17:     volatile int phase;
18:     volatile int isActive;
19: } Sig;
20:
21: typedef struct _wait {
22:     int id; // Thread/task
23:     mode md;
24:     int phase;
25:     int isActive;
26: } Wait;
```

phList = ph0 → ph1 → ph2 → ph3 → ph4 → ph5

sigTbl =

	t0	t1	t2	t3
ph0	■			
ph1		■		
ph2	■		■	
ph3		■		■
ph4			■	
ph5				■

waitTbl =

	t0	t1	t2	t3
ph0				
ph1	■			
ph2		■		
ph3			■	
ph4				■
ph5				

■ : Object
□ : NULL

19

Local-spin Implementation

```
1:    void signalOne(phaser *ph, int id) {
2:      Sig *s = sigTable[ph->id][id+offset];
3:      if (s != NULL) s->phase++;
4:    }

1:    void waitOne(phaser *ph, int id) {
2:      Wait *w = waitTbl[ph->id][id+offset];
3:      if (isMasterTask(ph, id)) {
4:        for (i = 0; i < num_tasks; i++) {
5:          Sig *s = sigTbl[ph->id][i];
6:          if (s != NULL) while (s->phase <= ph->mWaitPhase);
7:        }
8:        ph->mWaitPhase++;
9:        ph->mSigPhase++;
10:     } else { // Process for workers (non-master task)
11:       while (ph->mSigPhase <= w->phase);
12:     }
13:     w->phase++;
14:   }
```

T1<SIG> T2<SIG_WAIT> T3<SIG_WAIT> T4<WAIT>

signal

wait
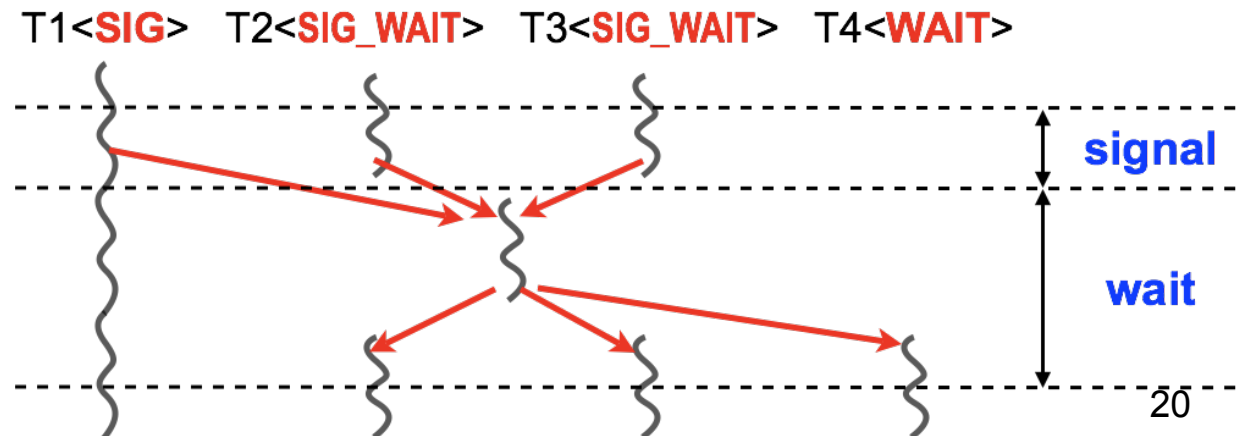
Experimental Setup

- **Platforms**
  - Intel Nehalem
    - 2.4GHz 8-core (2 Core i7)
    - Intel compiler v11.1 with –O3 option
  - Intel Xeon E7330
    - 2.4GHz 16-core (4 Core-2-Quad)
    - Intel compiler v11.0 with –O3 option
  - IBM Power7
    - 3.55GHz 32-core (SMT turned off)
   - IBM XLC v10.1 with –O5 option
- **Benchmarks**
  - EPCC syncbench microbenchmark
    - All-to-all barrier performance
  - JGF multithread v1.0 SOR
    - Ported from Java to C
    - Thread-level phaser
  - Polybench 2d-fdtd and 2d-seidel
    - Parallelized with loop tiling by PTile (polyhedral framework)
    - Iteration-level phaser

All-to-all Barrier Performance on Intel Nehalem, Xeon and IBM Power7
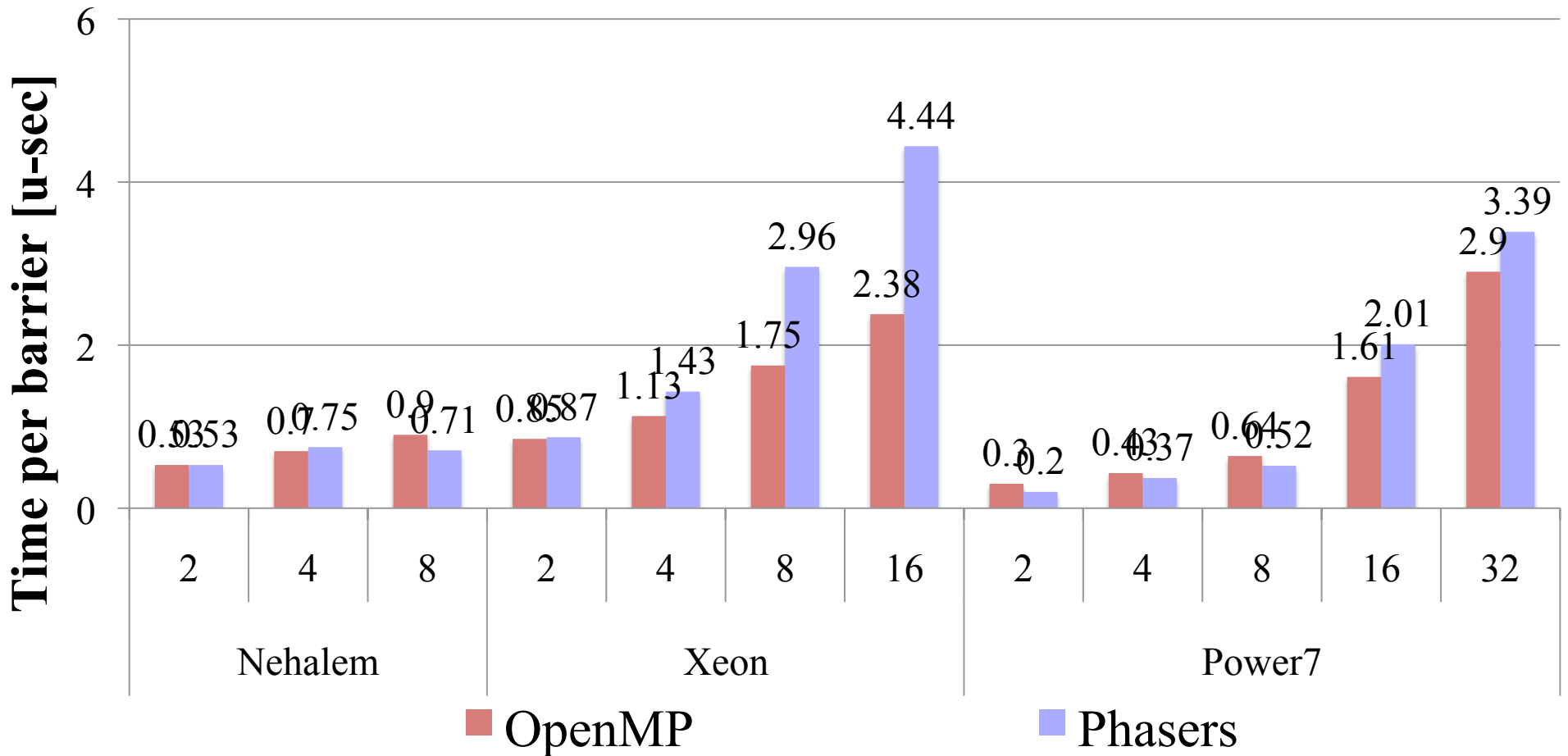
- All-to-all barrier performance by OpenMP and Phasers
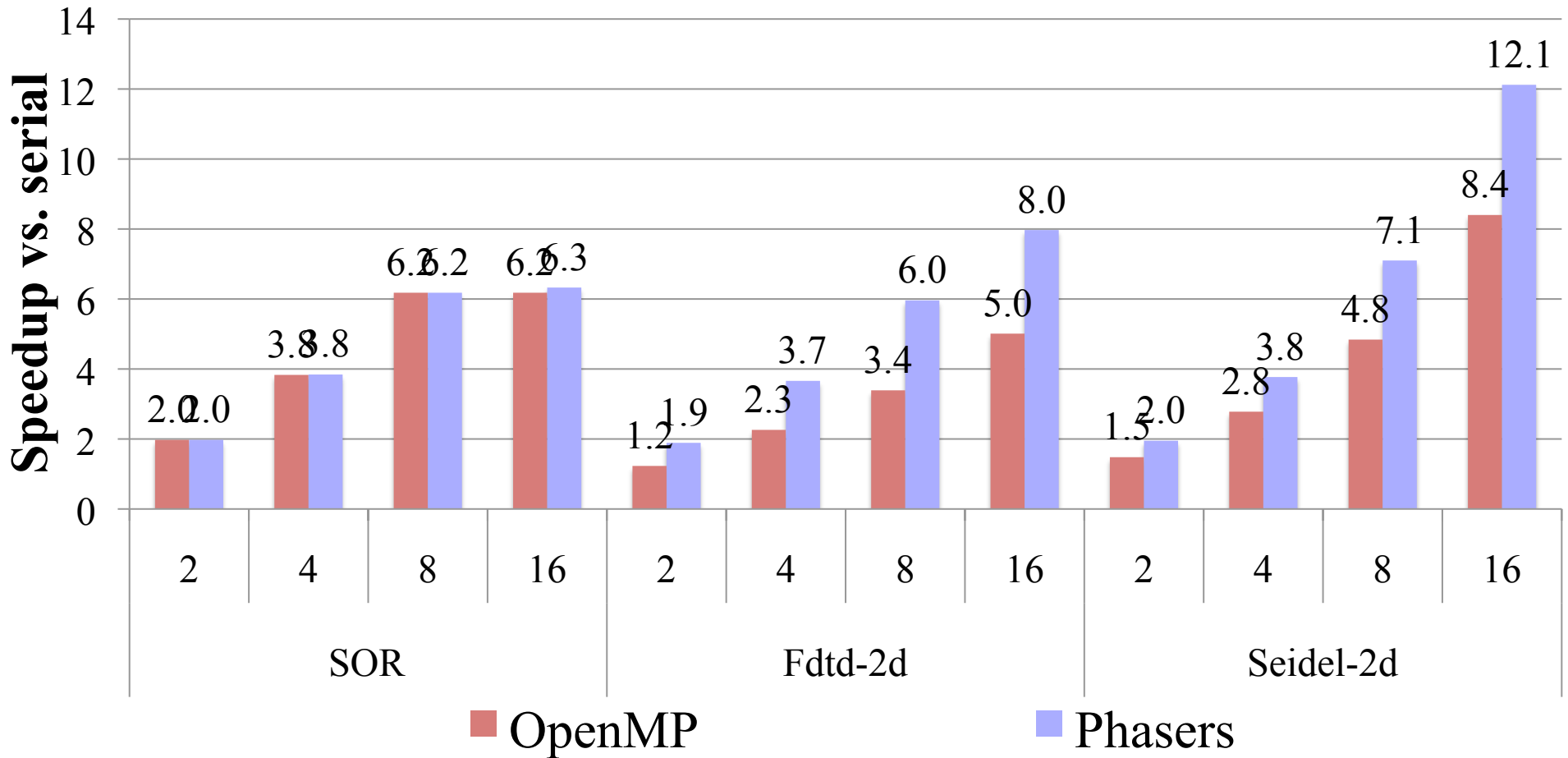- Vender implementation of OpenMP barrier is very efficient

22

Speedup for Application Benchmarks 2.4GHz 8-core Intel Nehalem
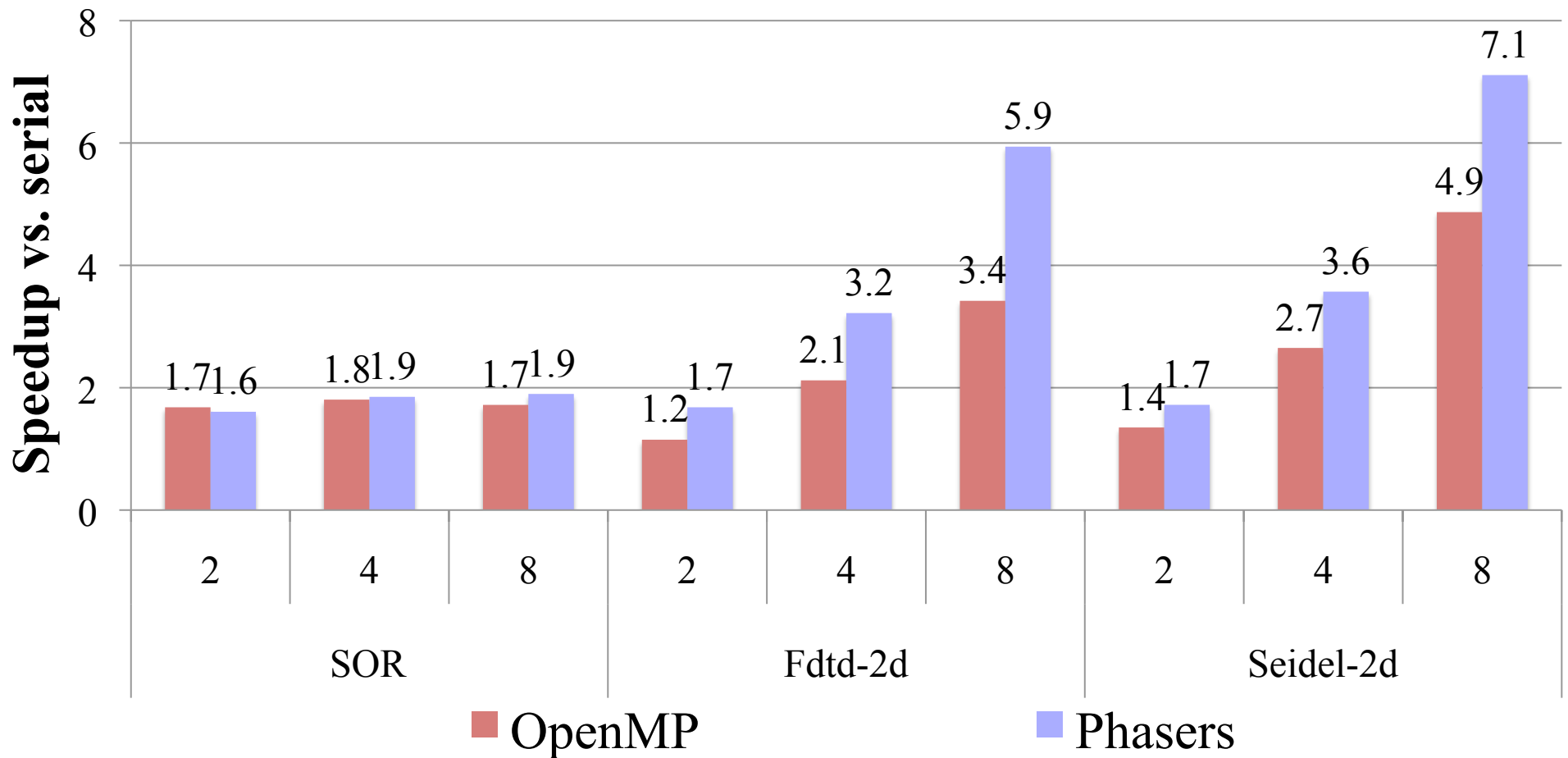
- SOR: 1.1x speedup with 8-core (thread-level)
- Fdtd-2d / Seidel-2d: 1.7x / 1.5x speedup with 8-core (iteration-level)
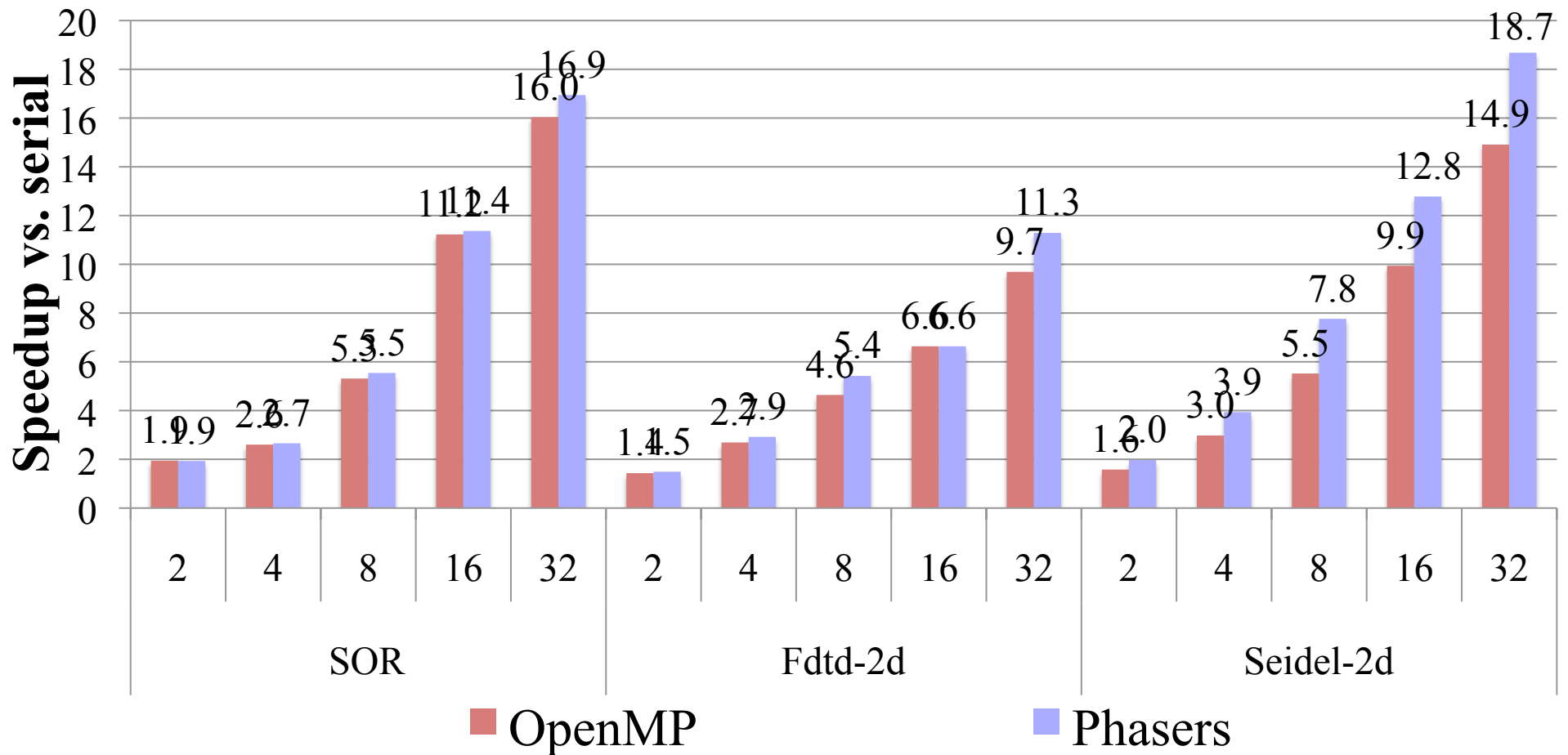
Speedup for Application Benchmarks
2.4GHz 16-core Intel Xeon

- SOR: 1.02x speedup with 16-core (thread-level)
- Fdtd-2d / Seidel-2d: 1.6x / 1.4x speedup with 16-core (iteration-level)

Speedup for Application Benchmarks 3.55GHz 32-core IBM Power7

- SOR: 1.06x speedup with 32-core (thread-level)
- Fdtd-2d / Seidel-2d: 1.2x / 1.3x speedup with 32-core (iteration-level)

Conclusion

- **Phasers for unified synchronization in OpenMP**
  - Collective barrier
  - Point-to-point synchronizations
- **Experimental results on three platforms**
  - 8-core Intel Core i7
    - 1.1x faster for SOR, 1.7x for Fdtd-2d and 1.5x on Seidel-2d
  - 16-core Intel Xeon
    - 1.02x faster for SOR, 1.6x for Fdtd-2d, and 1.4x for Seidel-2d
  - 32-core IBM Power7
    - 1.06x faster for SOR, 1.2x for Fdtd-2d, and 1.3x for Power7
- **Future work**
  - Synchronization support for dynamic task parallelism
  - Support of reduction and single statement
  - Compiler support of loop chunking with barrier operations