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

IWOMP Workshop
June 14th, 2011

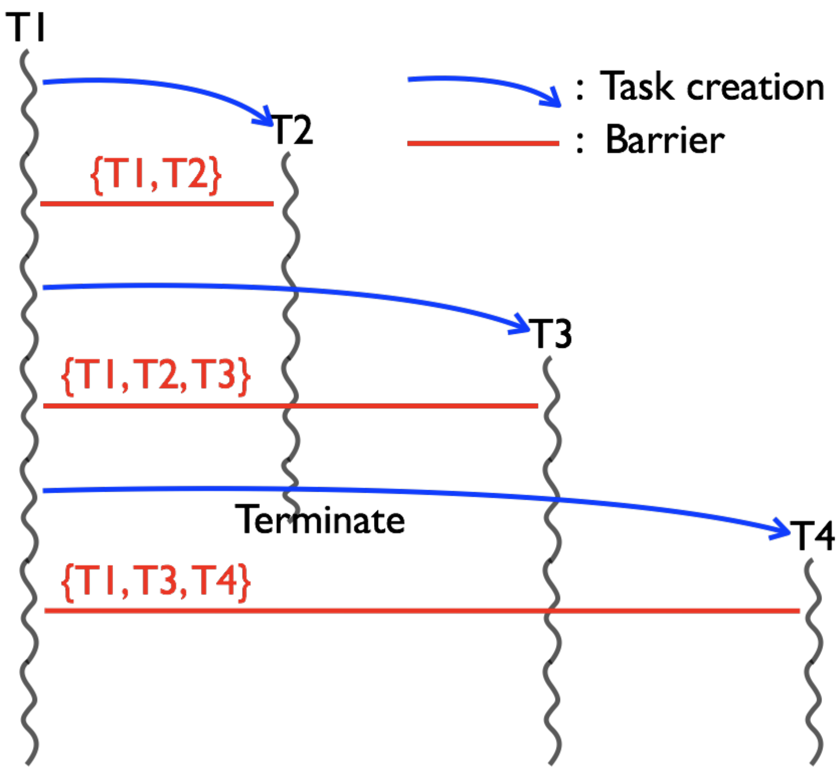
Jun Shirako, Kamal Sharma, and Vivek Sarkar
Rice University

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**

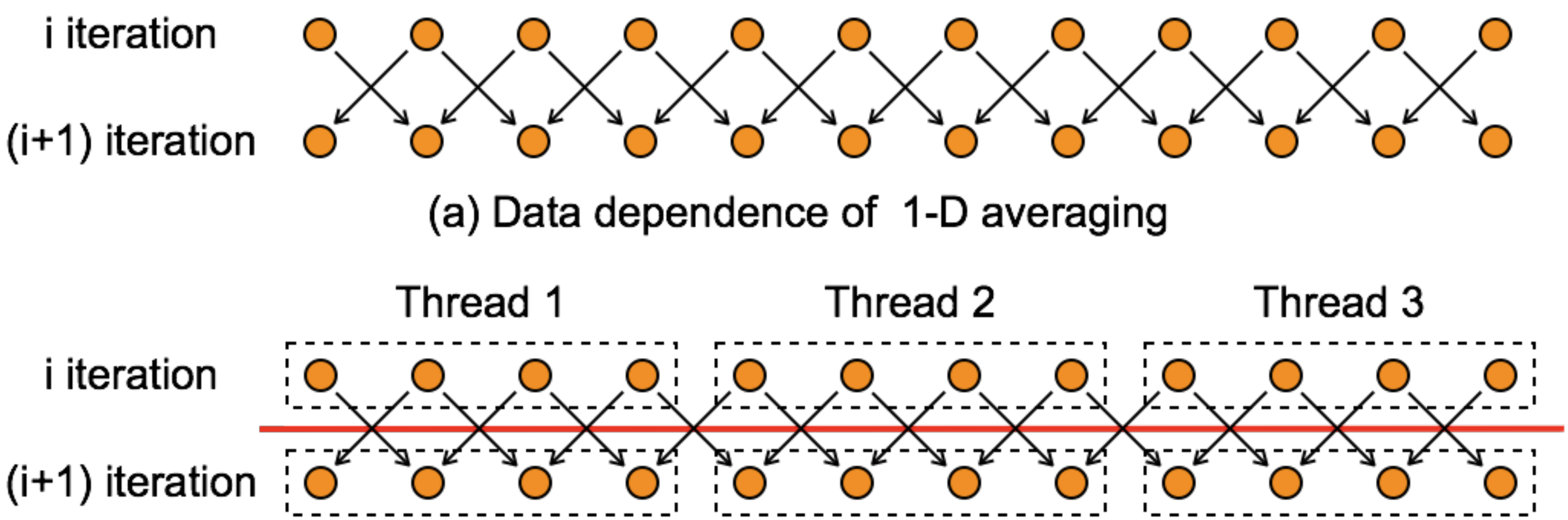
Review of Some OpenMP Constructs

```
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

```
#pragma omp parallel private(iter) firstprivate(newA, oldA)
{
  for (iter = 0; iter < NUM_ITERS; iter++) {
    #pragma omp for schedule(static) nowait
    for (j = 1; j < n-1; j++) {
      newA[j] = (oldA[j-1] + oldA[j+1]) / 2.0;
    }
    double *temp = newA; newA = oldA; oldA = temp;
    #pragma omp barrier
} }
```

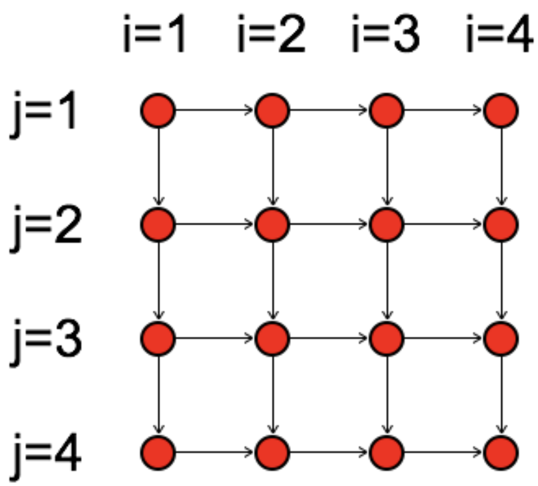


(a) Data dependence of 1-D averaging

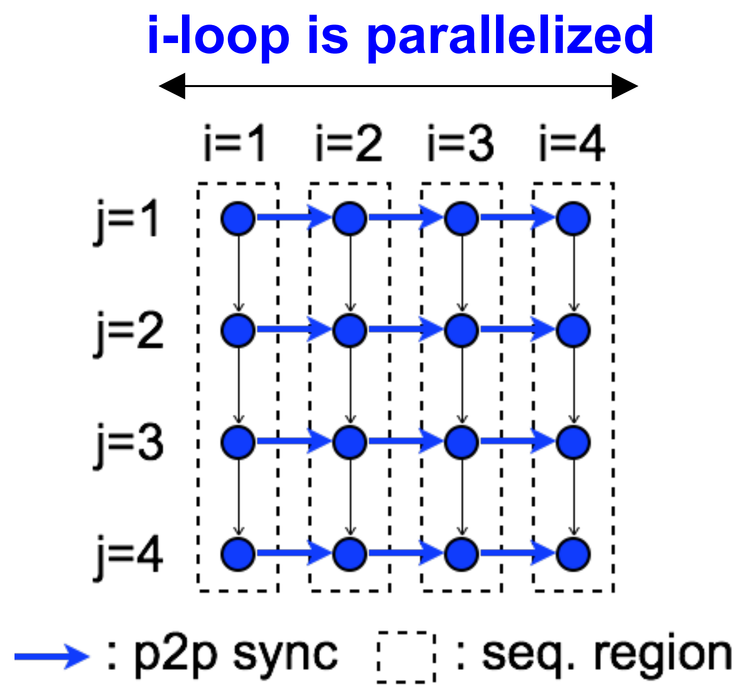
(b) Barrier synchronization

Stencil with Pipeline Parallelism

```
// Sequential version
for (i = 1; i < n-1; i++) {
  for (j = 1; j < m-1; j++) {
    A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
                               A[i-1][j], A[i+1][j]);
  }
}
```



(a) Data dependence of stencil

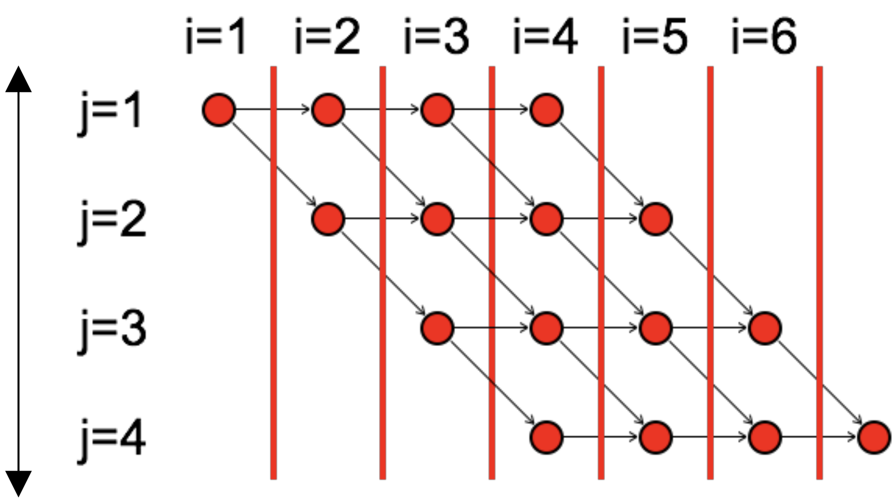


(c) Pipeline parallelism

Stencil with Wavefront Parallelism

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

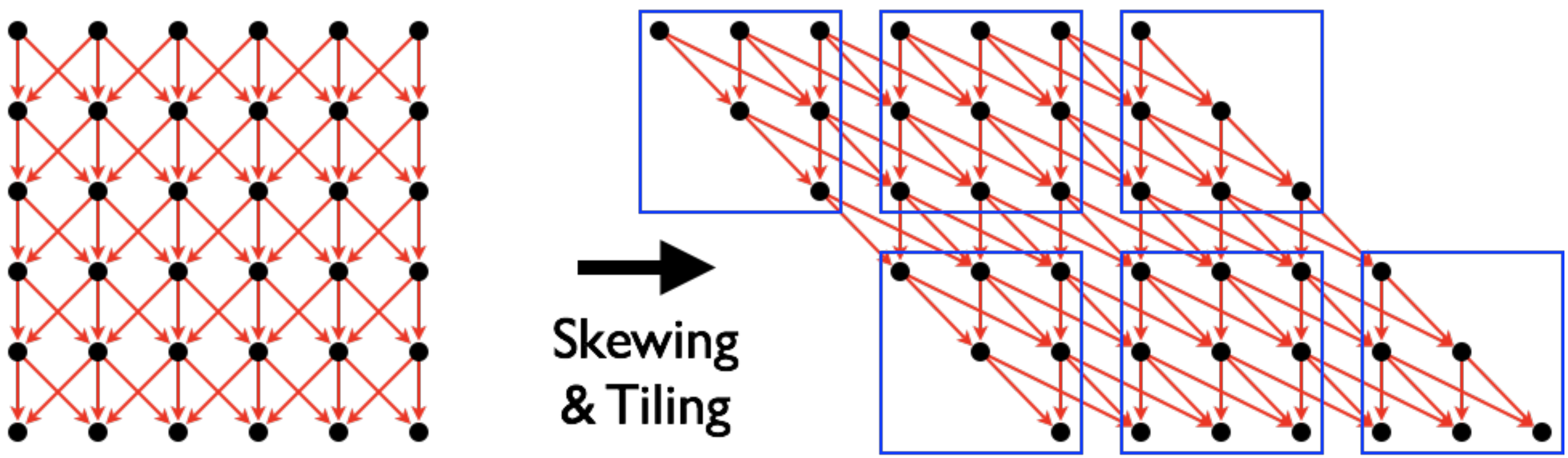
j-loop is parallelized



(b) Wavefront parallelism

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 (≤, ≤, …, ≤) dependence vector
  - Naturally have (at least) 1-level pipeline parallelism



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**

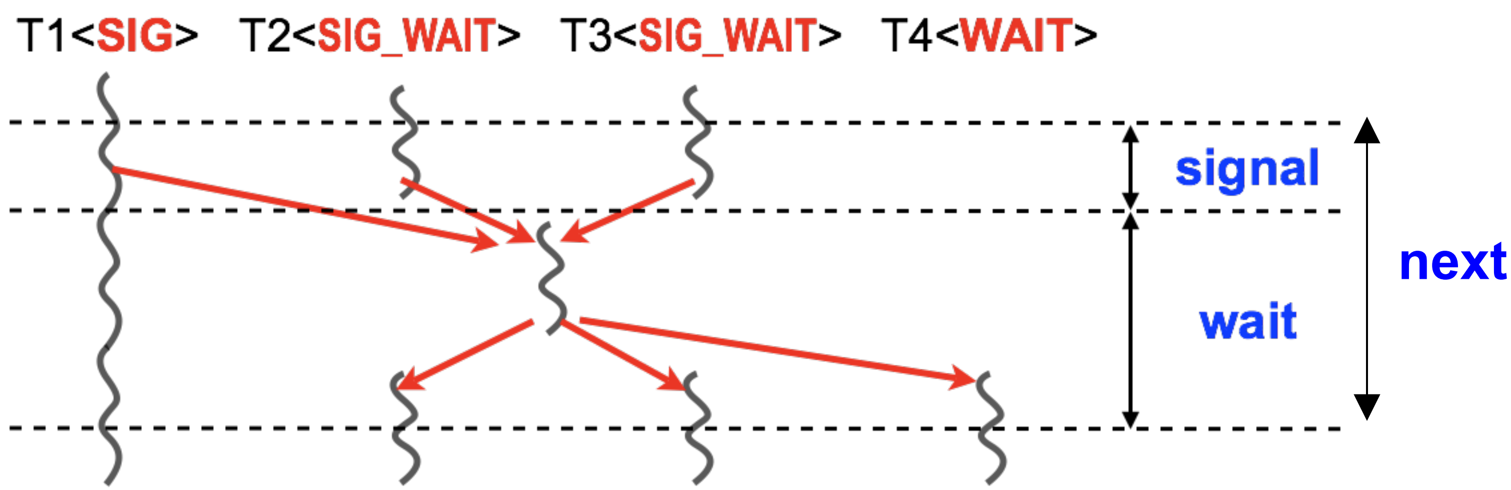
Phasers

- **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)



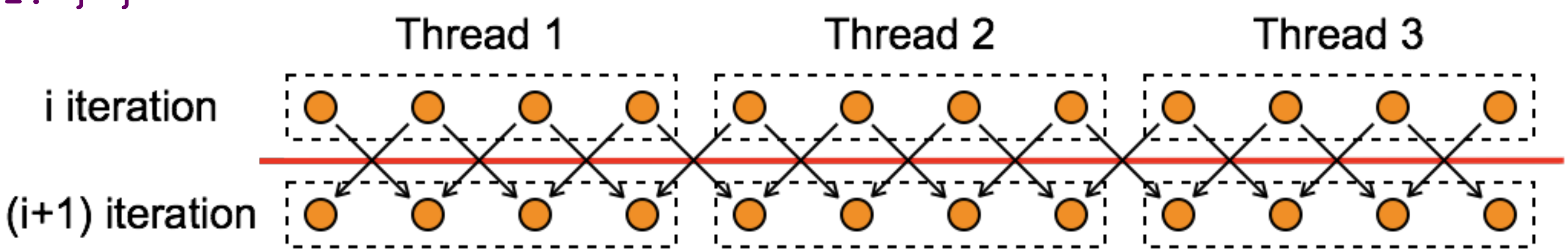
- A master task is selected in tasks w/ wait capability
- It receives all signals and broadcasts a barrier completion notice

Thread-level Phaser API (iterative averaging ex.)

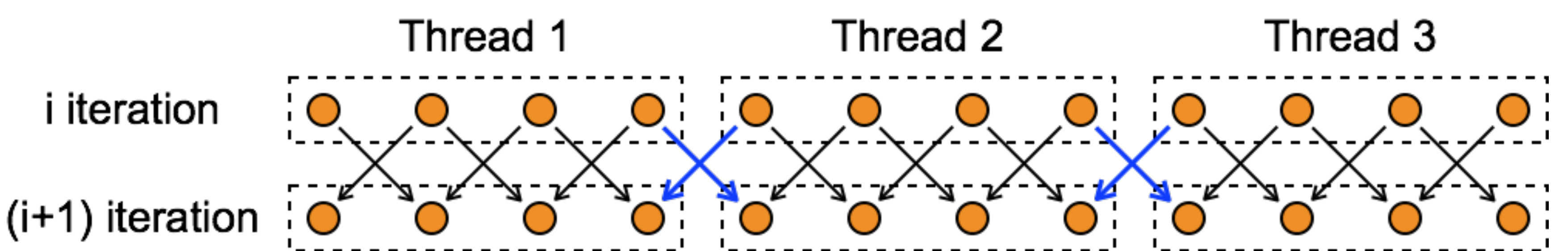
```
/* Phaser allocation in serial region */
phaser **ph = calloc(num_threads+2, sizeof(phaser *));
for (i = 0; i < num_threads+2; i++) ph[i] = phaser_new();

/* Registration */
for (id = 0; id < num_threads; id++) {
  phaserRegisterThread(ph[id], id, WAIT);   // Wait left neighbor
  phaserRegisterThread(ph[id+1], id, SIG);
  phaserRegisterThread(ph[id+2], id, WAIT); // Wait right neighbor
}
/* Parallel execution with phaser synchronization */
#pragma omp parallel private(iter) firstprivate(newA, oldA)
{
  for (iter = 0; iter < NUM_ITERS; iter++) {
    #pragma omp for schedule(static) nowait
    for (j = 1; j < n-1; j++) {
      newA[j] = (oldA[j-1] + oldA[j+1])/2.0;
    }
    double *temp = newA; newA = oldA; oldA = temp;
    #pragma omp next
} }
/* Deregistration to change synchronization pattern */
dropPhasersAll();
```

Thread-level Phaser API (iterative averaging ex.)

```
#pragma omp parallel private(iter) firstprivate(newA, oldA)
{
  for (iter = 0; iter < NUM_ITERS; iter++) {
    #pragma omp for schedule(static) nowait
    for (j = 1; j < n-1; j++) {
      newA[j] = (oldA[j-1] + oldA[j+1])/2.0;
    }
    double *temp = newA; newA = oldA; oldA = temp;
    #pragma omp next
} }
```



(b) Barrier synchronization



(c) Point-to-point synchronization

Iteration-level Phaser

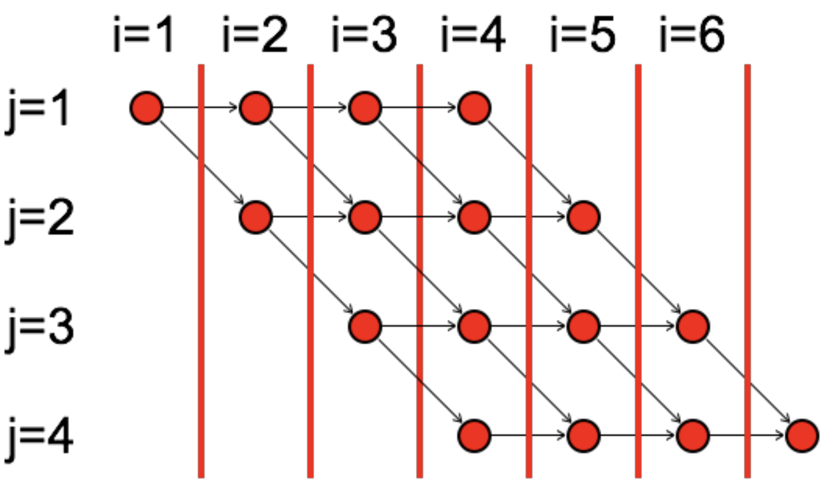
- **Synchronization among iterations of parallel loop**
  - Higher level of abstraction
    - Express data dependence among iterations
    - signal / wait / next directives are used within parallel for loops
  - Less flexibility in synchronization pattern than thread-level
    - Direction of synchronization 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.)

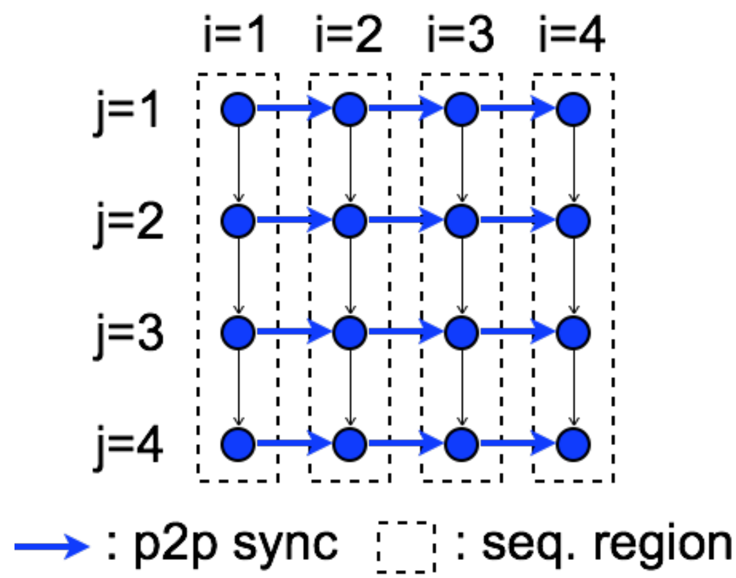
```
/* Phaser allocation in serial region */
phaser **ph = calloc(n+1, sizeof(phaser *));
for (i = 0; i < n+1; i++) ph[i] = phaser_new();

/* Registration */
for (i = 0; i < n; i++) {
  /* Sync direction from left to right */
  phaserRegisterIteration(ph[i], i, WAIT);  // Wait left neighbor
  phaserRegisterIteration(ph[i+1], i, SIG); // Signal right neighbor
}

/* Parallel execution with phaser synchronization */
#pragma omp parallel
{
  #pragma omp for private(j) schedule(static, 1)
  for (i = 1; i < n-1; i++) {
    for (j = 1; j < m-1; j++) {
      #pragma omp wait
      A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
                        A[i-1][j], A[i+1][j]);
      #pragma omp signal
  } }
}
dropPhasersAll(); /* Deregistration */
```

Iteration-level Phaser API (iterative averaging ex.)

```
#pragma omp parallel for private(j) schedule(static, 1)
for (i = 1; i < n-1; i++) {
  for (j = 1; j < m-1; j++) {
    #pragma omp wait
    A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
                       A[i-1][j], A[i+1][j]);
    #pragma omp signal
} }
```



(b) Wavefront parallelism



(c) Pipeline parallelism

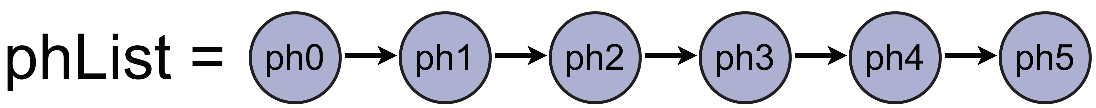
Pipeline parallelism: Better synchronization efficiency & data locality

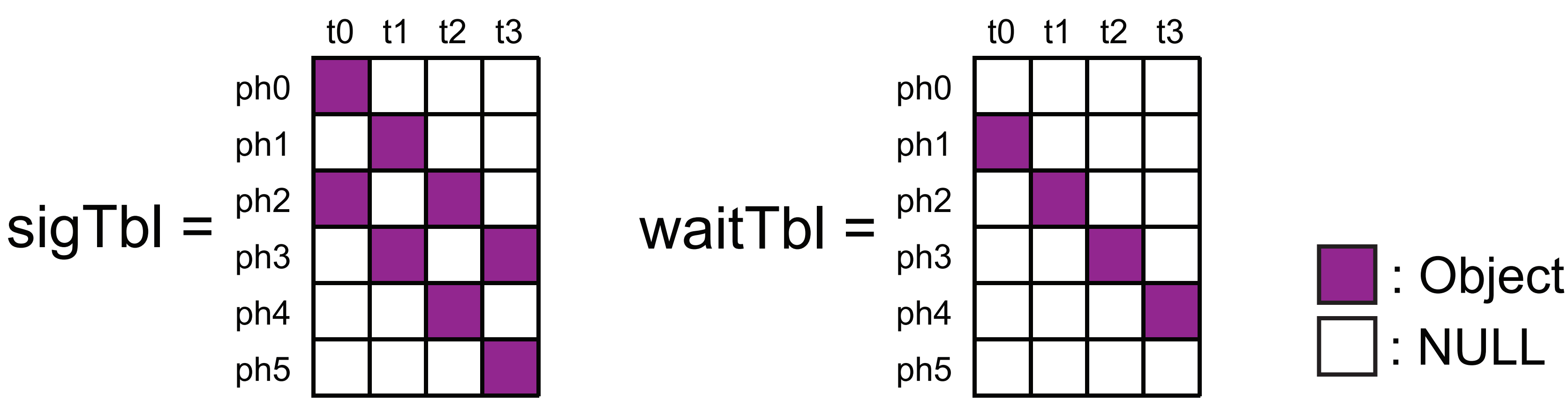
Outline

- 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

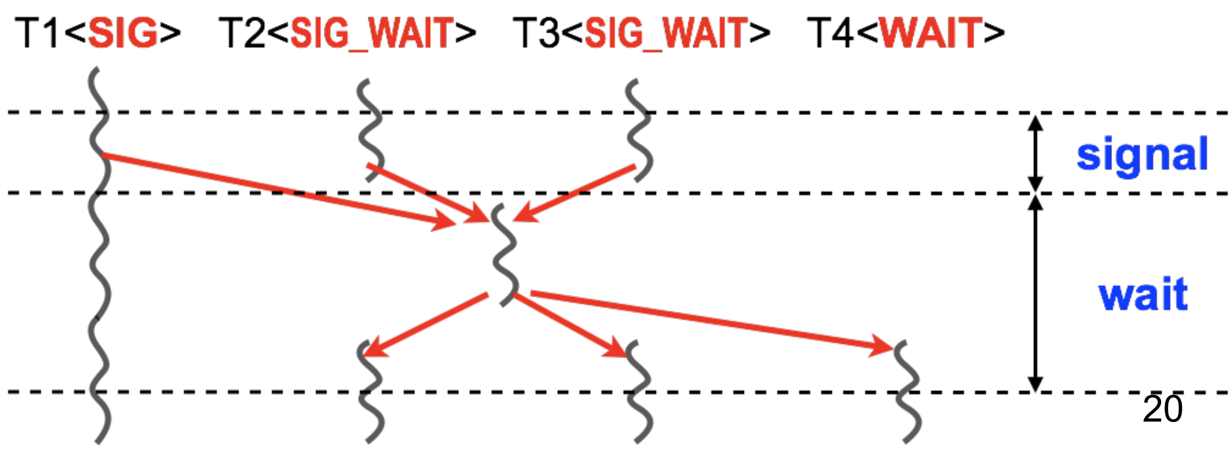
```
typedef struct _phaser {
  int id;
  // Contains Sig/Wait objects
  List *sigList, *waitList;

  volatile int mSigPhase;
  int mWaitPhase;
  int masterId;

  // Customized for single signaler
  int numSig, singleSigId;
} phaser;

typedef struct _sig {
  int id; // Thread/task
  mode md;
  volatile int phase;
  volatile int isActive;
} Sig;

typedef struct _wait {
  int id; // Thread/task
  mode md;
  int phase;
  int isActive;
} Wait;
```





Local-spin Implementation

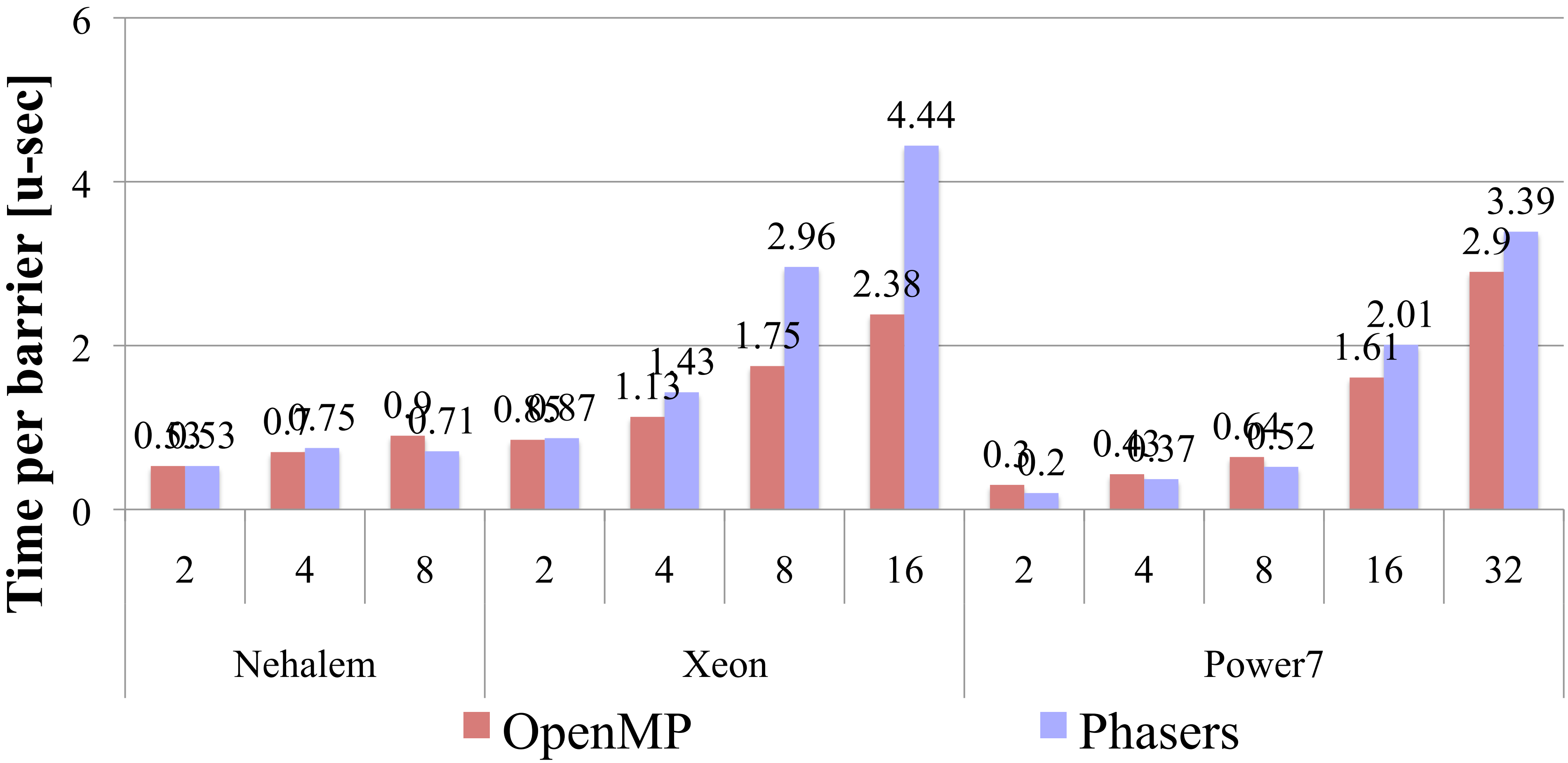
```
void signalOne(phaser *ph, int id) {
  Sig *s = sigTable[ph->id][id+offset];
  if (s != NULL) s->phase++;
}

void waitOne(phaser *ph, int id) {
  Wait *w = waitTbl[ph->id][id+offset];
  if (isMasterTask(ph, id)) {
    for (i = 0; i < num_tasks; i++) {
      Sig *s = sigTbl[ph->id][i];
      if (s != NULL) while (s->phase <= ph->mWaitPhase);
    }
    ph->mWaitPhase++;
    ph->mSigPhase++;
  } else { // Process for workers (non-master task)
    while (ph->mSigPhase <= w->phase);
  }
  w->phase++;
}
```



Experimental Setup

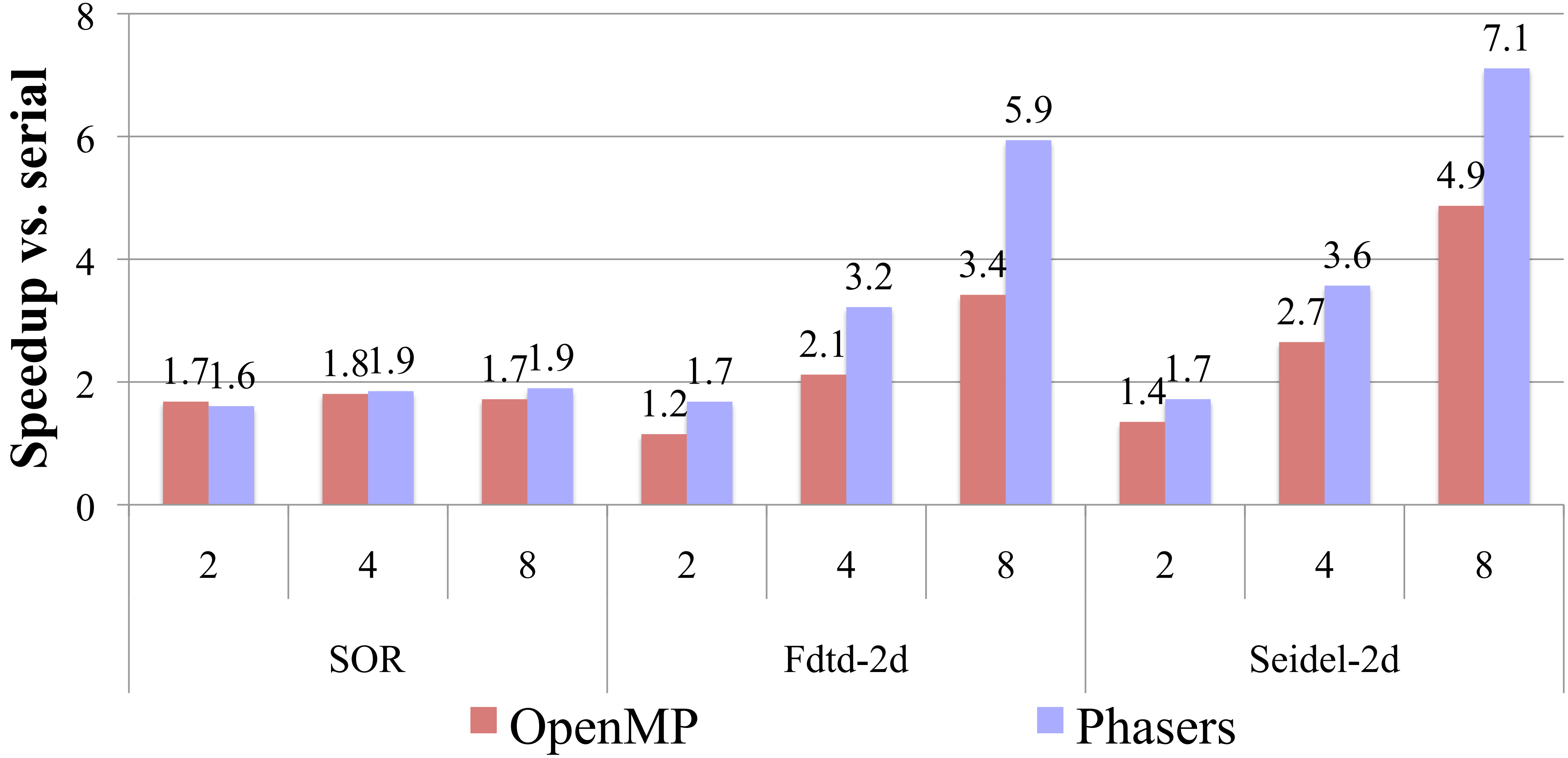
- **Platforms**
  - Intel Nehalem
    - 2.4GHz 8-core (2 Core i7)
    - Intel compiler v11.1 with –O3 option
  - Intel Xeon E7330
    - 2.4GHz 16-core (4 Core-2-Quad)
    - Intel compiler v11.0 with –O3 option
  - IBM Power7
    - 3.55GHz 32-core (SMT turned off)
    - IBM XLC v10.1 with –O5 option
- **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



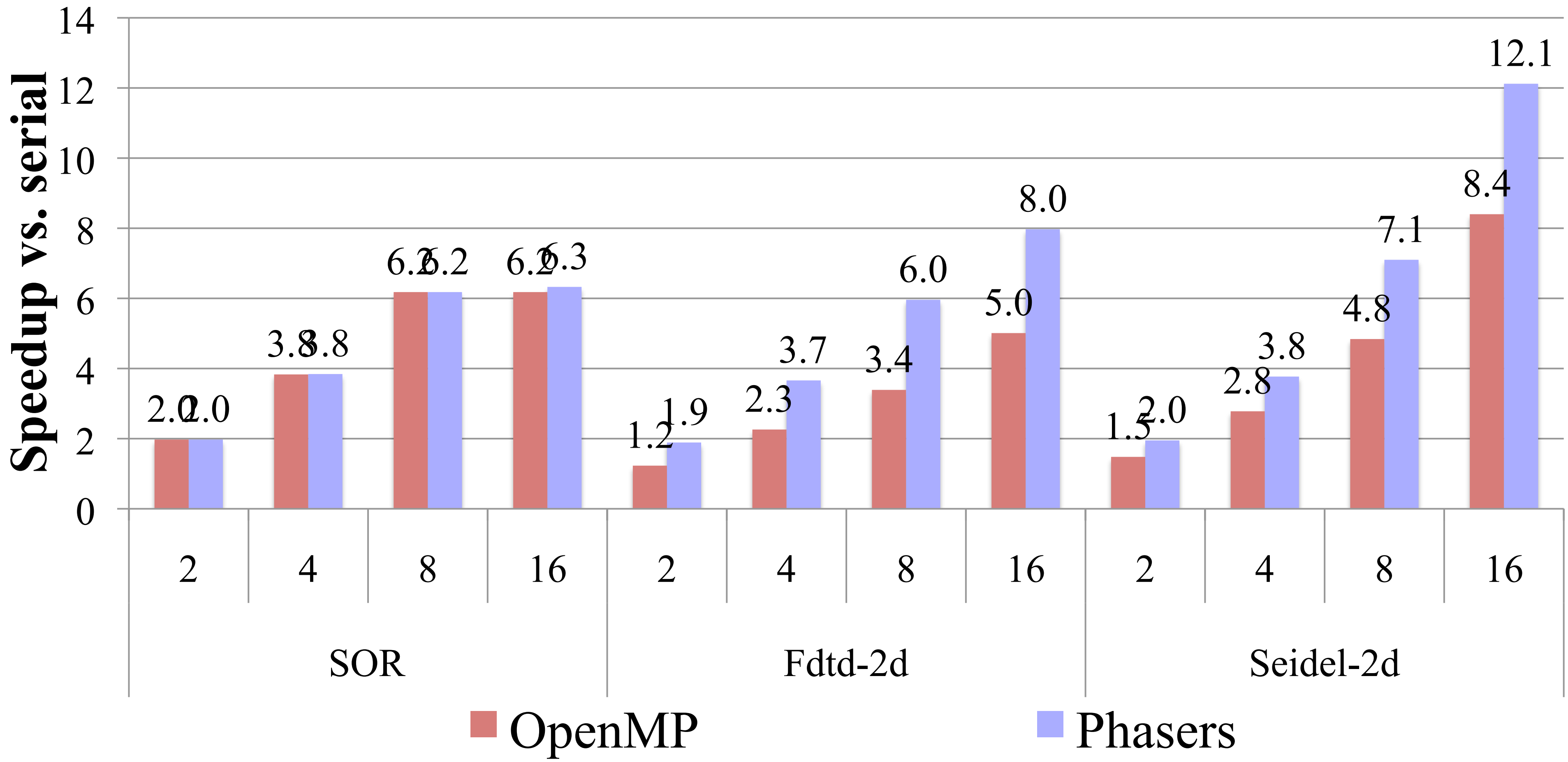
- All-to-all barrier performance by OpenMP and Phasers
- Vender implementation of OpenMP barrier is very efficient

Speedup for Application Benchmarks 2.4GHz 8-core Intel Nehalem



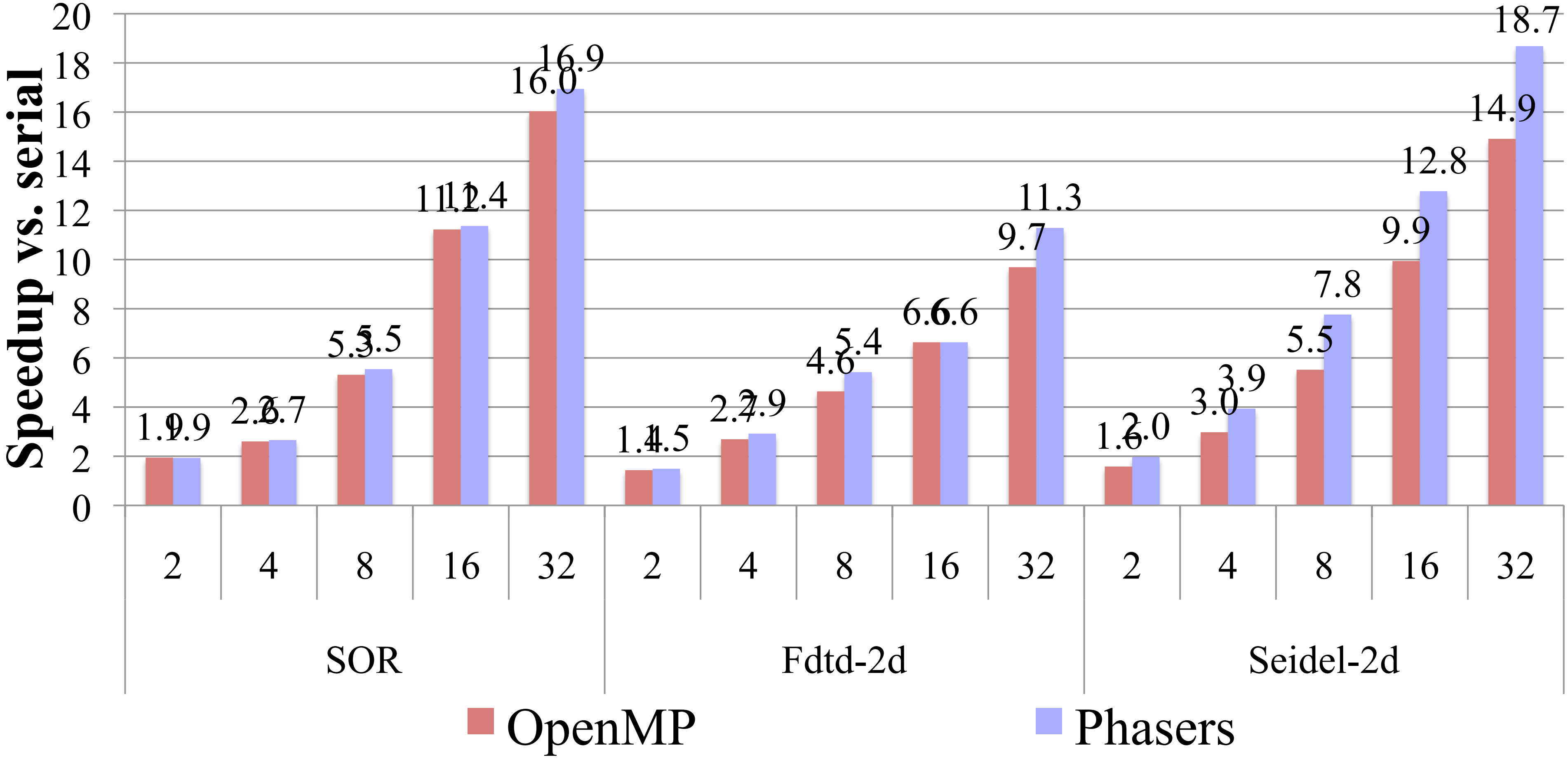
- SOR: 1.1x speedup with 8-core (thread-level)
- Fdtd-2d / Seidel-2d: 1.7x / 1.5x speedup with 8-core (iteration-level)

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