Pipeline Parallelism and the OpenMP Doacross Construct

COMP515 - guest lecture

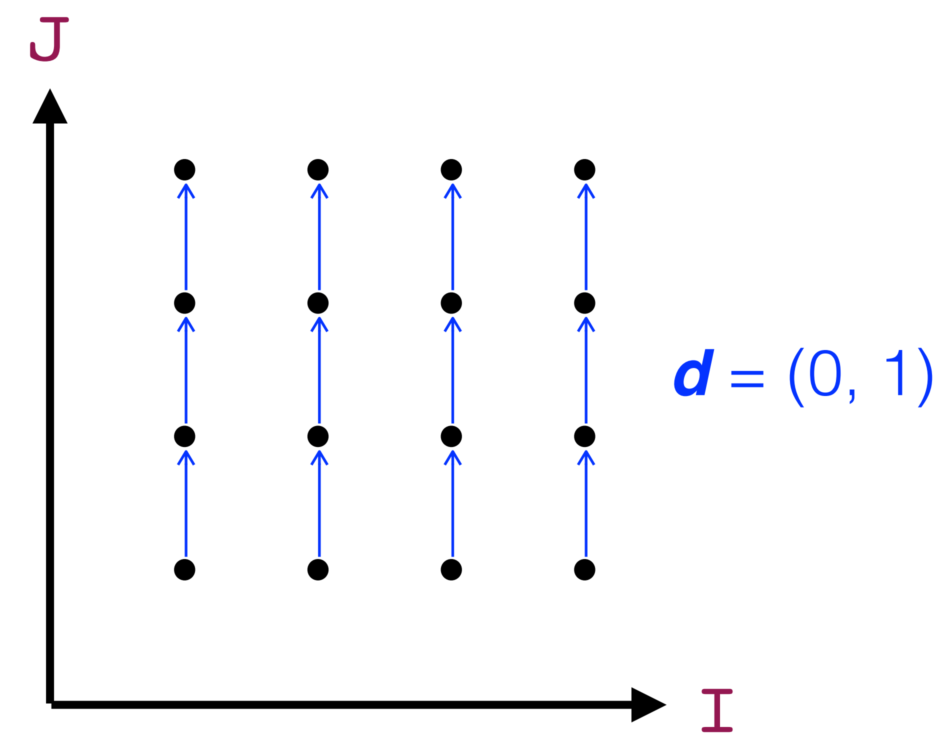
October 27th, 2015

Jun Shirako

Doall Parallelization (Recap)

- No loop-carried dependence among iterations of doall loop
- Parallel execution without synchronization

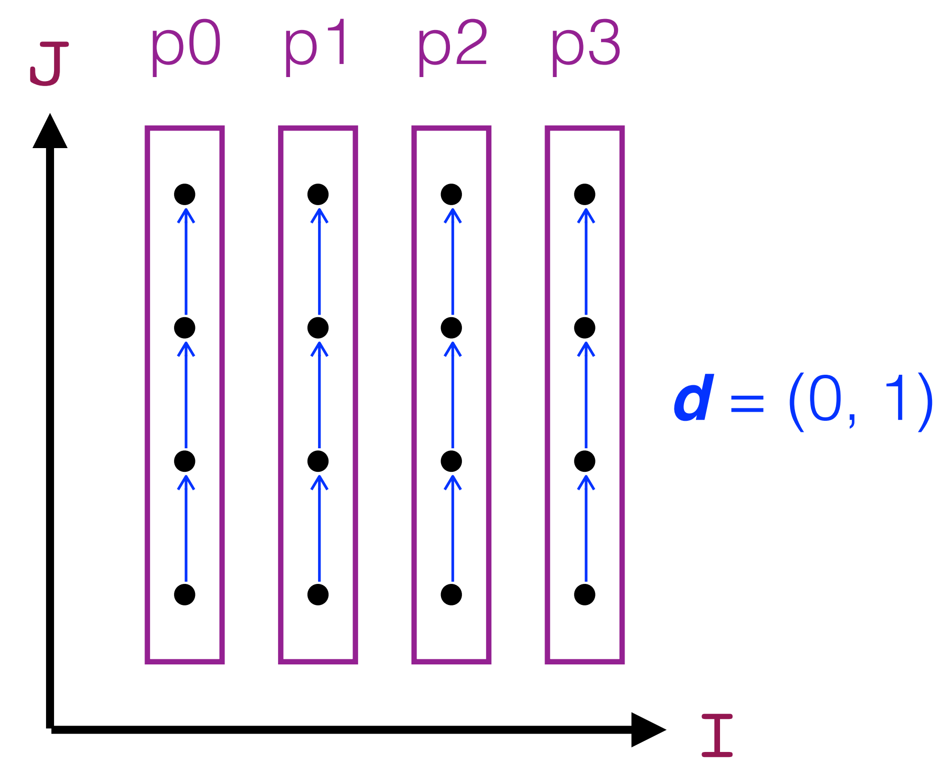
```
! ex.1
DO I = 1, N
  DO J = 1, M
    A(J,I) = A(J-1,I)
  END DO
END DO
```



Doall Parallelization (contd)

- No loop-carried dependence among iterations of doall loop
- Parallel execution without synchronization

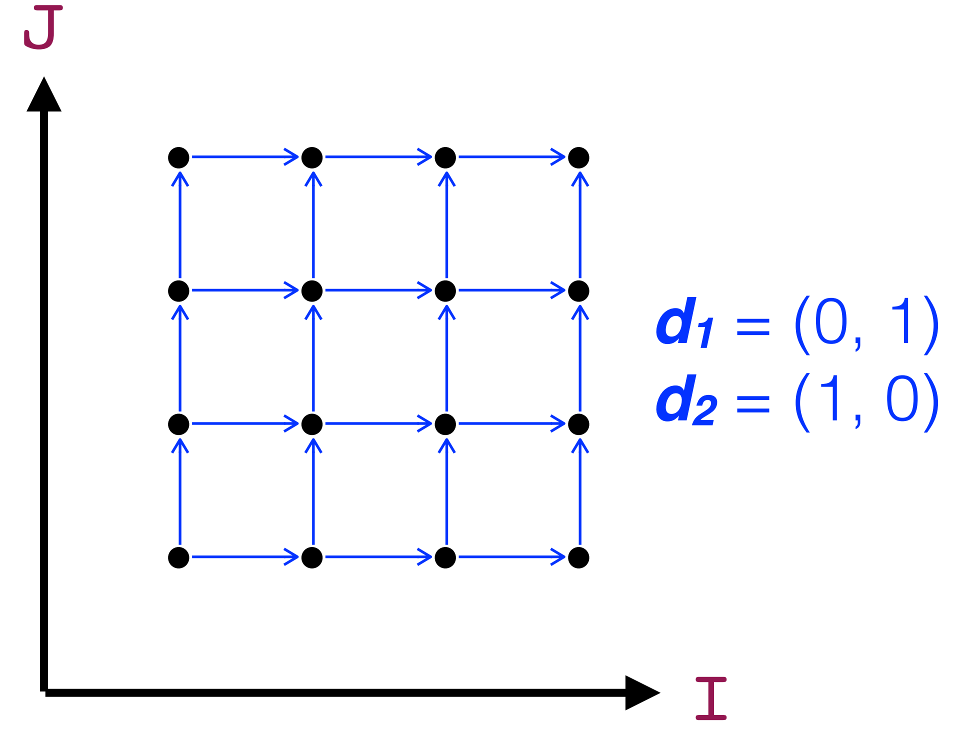
```
! ex.1
PARALLEL DO I = 1, N
  DO J = 1, M
    A(J,I) = A(J-1,I)
  END DO
END DO
```



Wavefront Parallelization (Recap)

- Loop-carried dependences exist among iterations of all loops
  - How to expose doall parallelism?
- Transformations :

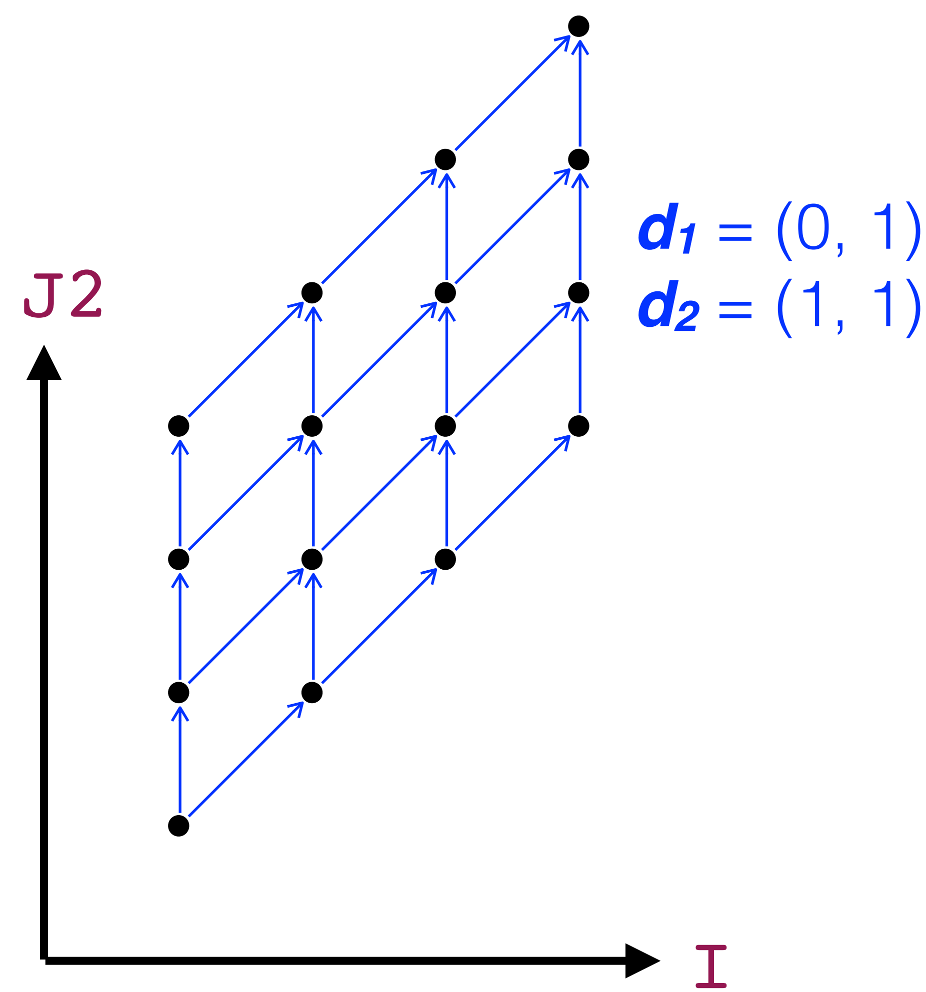
```
! ex.2
DO I = 1, N
  DO J = 1, M
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



Wavefront Parallelization (contd)

- Loop-carried dependences exist among iterations of all loops
  - How to expose doall parallelism?
- Transformations : skewing

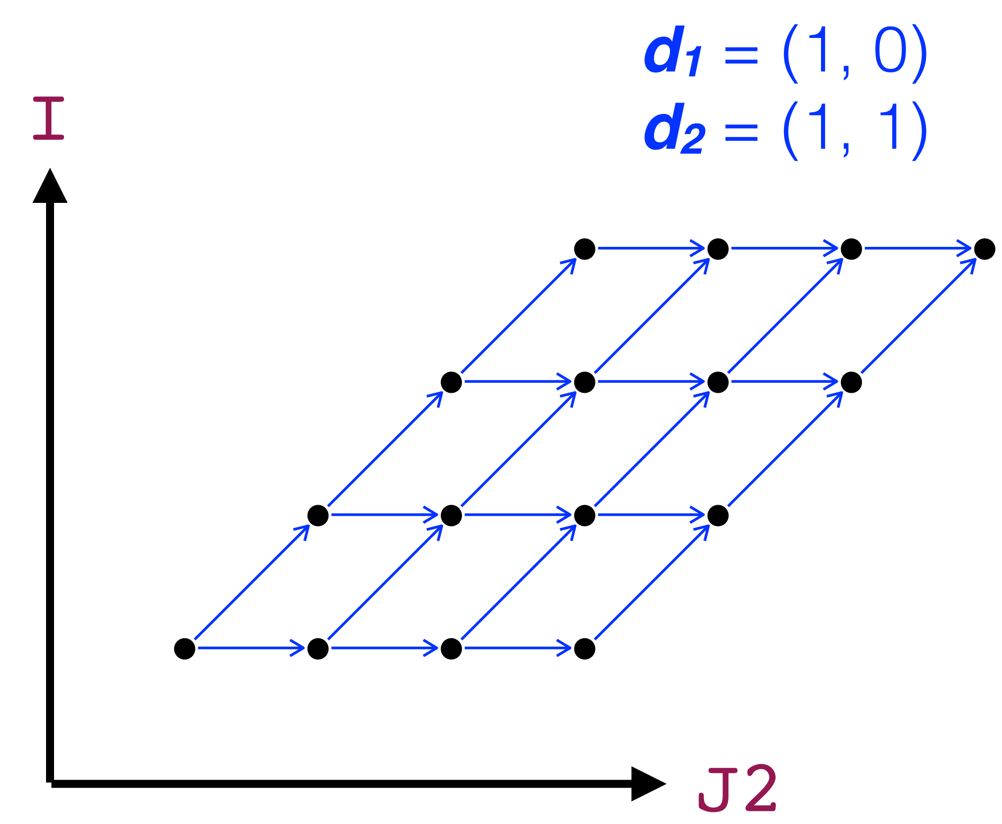
```
! ex.2
DO I = 1, N
  DO J2 = I, M+I-1
    J = J2 - (I-1)
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



Wavefront Parallelization (contd)

- Loop-carried dependences exist among iterations of all loops
  - How to expose doall parallelism?
- Transformations : skewing + interchange

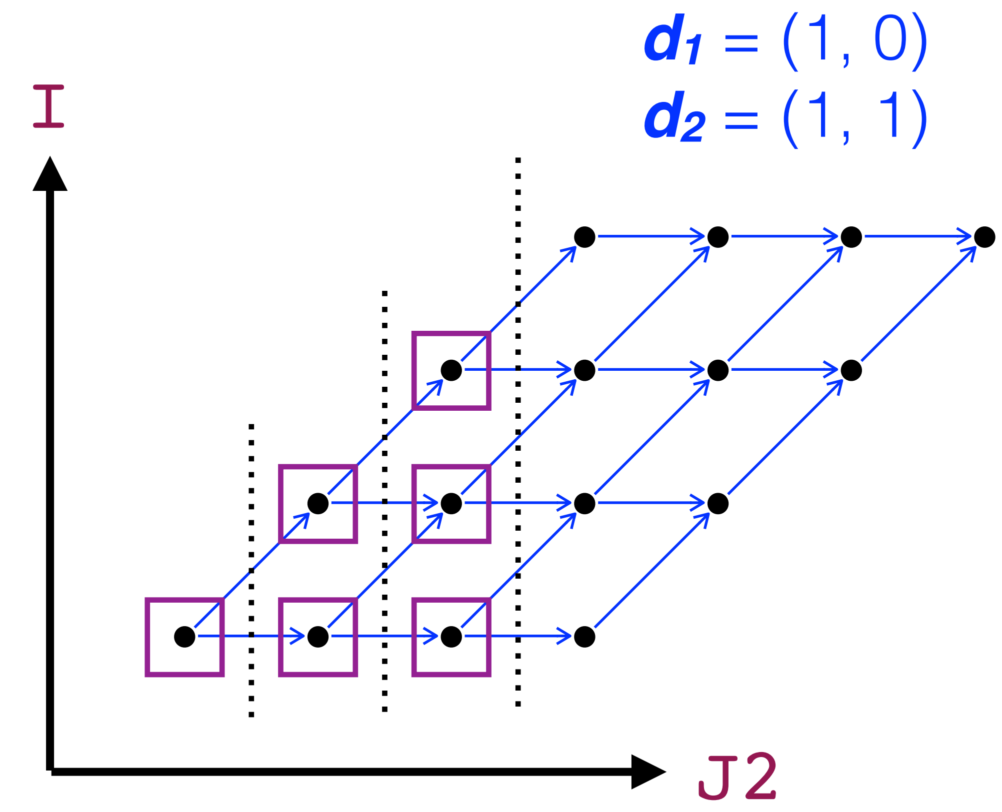
```
! ex.2
DO J2 = 1, N+M-1
  ILW = MAX(1,J2-M+1)
  IUP = MIN(N,J2)
  DO I = ILW, IUP
    J = J2 - (I-1)
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



Wavefront Parallelization (contd)

- Loop-carried dependences exist among iterations of all loops
  - How to expose doall parallelism?
- Transformations : skewing + interchange + inner DOALL

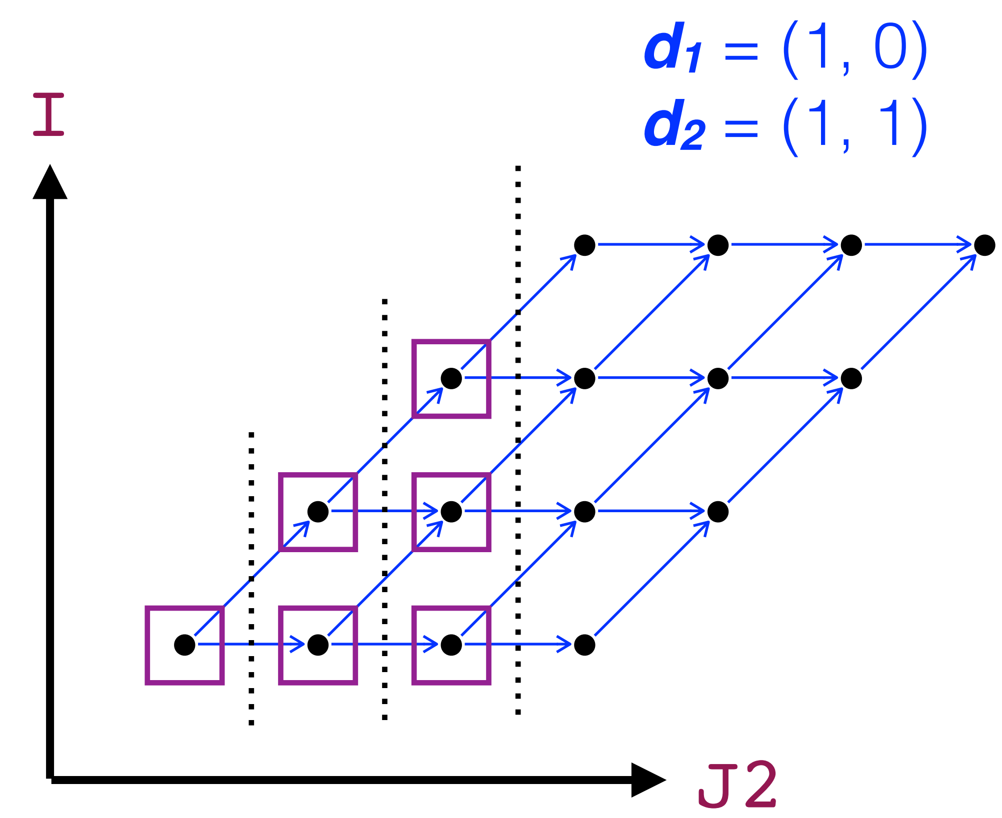
```
! ex.2
DO J2 = 1, N+M-1
  ILW = MAX(1,J2-M+1)
  IUP = MIN(N,J2)
  PARALLEL DO I = ILW, IUP
    J = J2 - (I-1)
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



Performance Issues with Wavefront Transformation

- Large synchronization overhead
  - Need barrier for each outer-iteration (J2 loop)
- Performance issues
  - Non-uniform iteration lengths in DOALL loop
  - Non-contiguous data access after skewing (in sequential version or when DOALL loop is chunked)

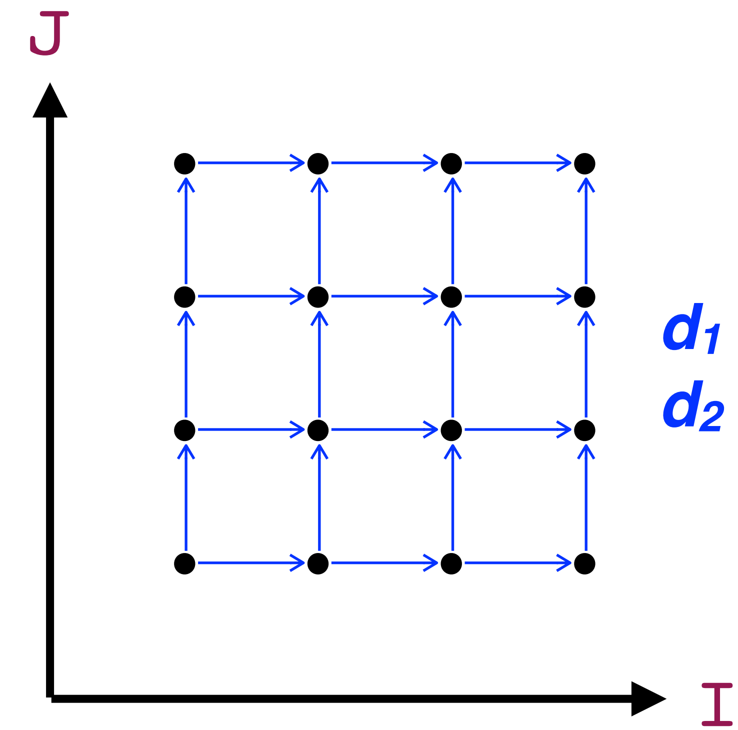
```
! ex.2
DO J2 = 1, N+M-1
  ILW = MAX(1,J2-M+1)
  IUP = MIN(N,J2)
  PARALLEL DO I = ILW, IUP
    J = J2 - I + 1
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



Doacross Parallelization

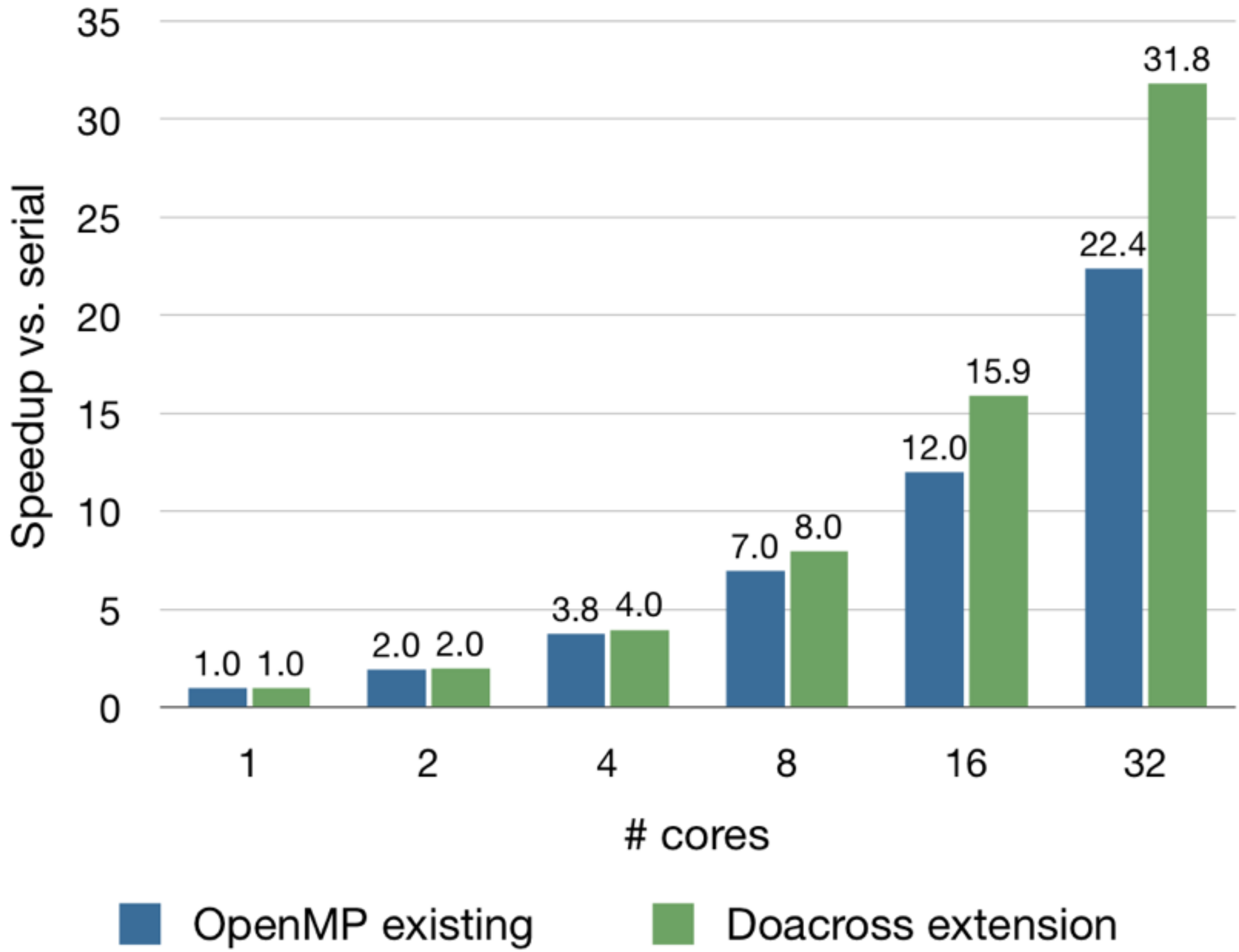
- Outer-level parallelization when dependences exist among iterations
- Parallel execution can be enabled via point-to-point synchronizations among iterations of DOACROSS loop
  - Synchronizations are expressed using POST and WAIT operations
  - Expensive in older SMPs, but cheaper with on-chip synchronization in newer multicore SMPs

```
! ex.2
DO I = 1, N
  DO J = 1, M
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



$d_1 = (0, 1)$
$d_2 = (1, 0)$

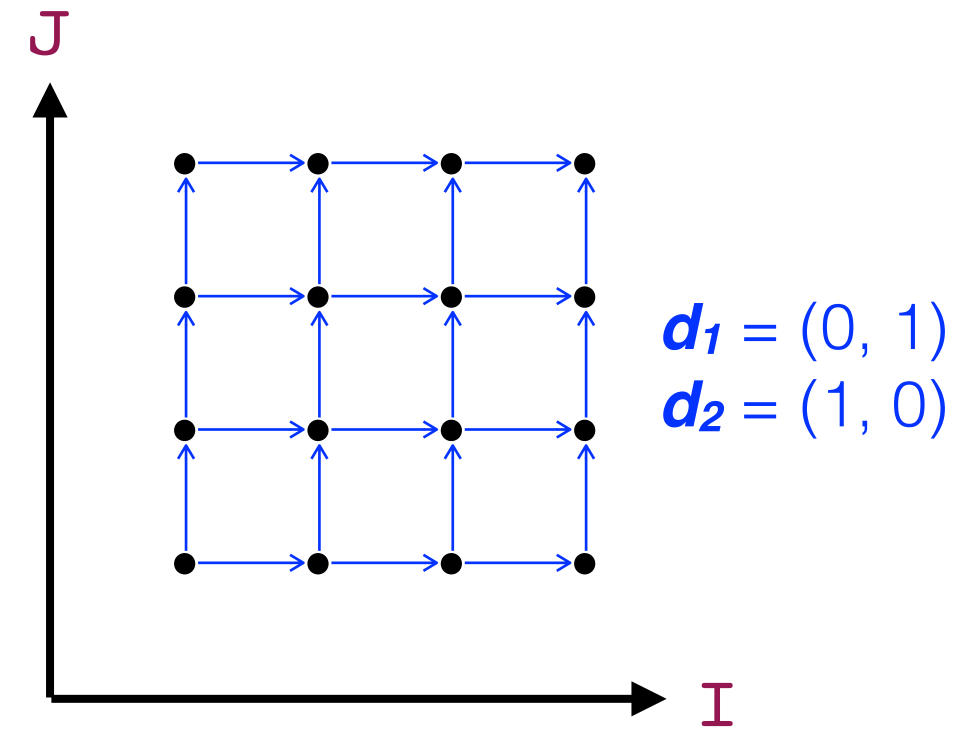
Speedup on 32-core IBM Power7



Doacross Parallelization

- Loop-carried dependences exist among iterations
- Parallel execution can be enabled via point-to-point synchronization among iterations of DOACROSS loop
  - Synchronizations are expressed using POST and WAIT

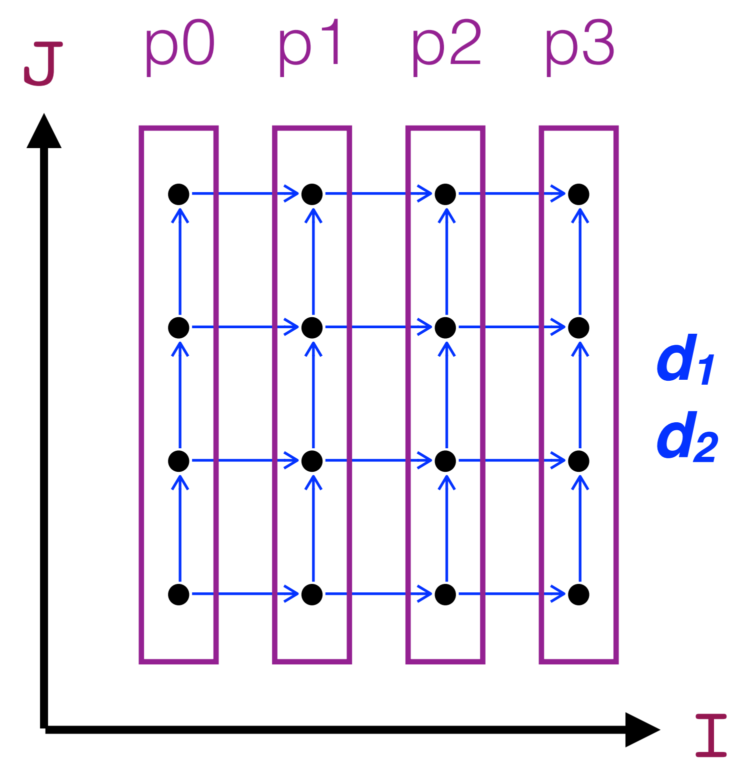
```
! ex.2
DO I = 1, N
  DO J = 1, M
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```



Doacross Parallelization

- Loop-carried dependences exist among iterations
- Parallel execution can be enabled via point-to-point synchronization among iterations of DOACROSS loop
  - Synchronizations are expressed using POST and WAIT

```
! ex.2
DOACROSS I = 1, N
  DO J = 1, M
    A(J,I) = A(J-1,I) + A(J,I-1)
  END DO
END DO
```

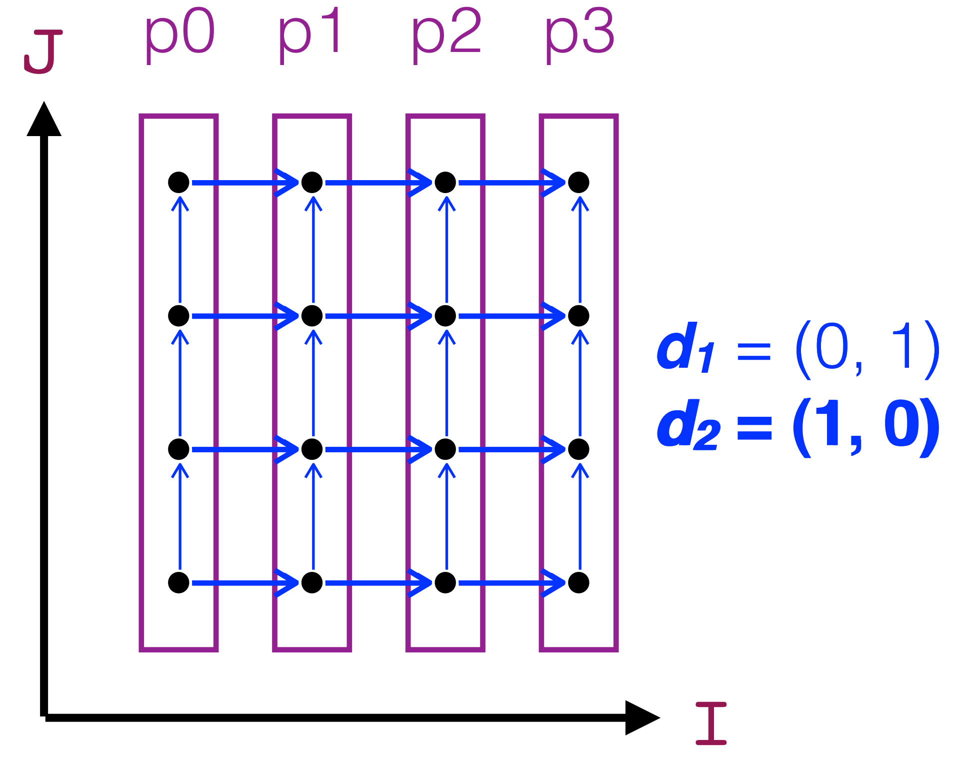


$d_1 = (0, 1)$
$d_2 = (1, 0)$

Doacross Parallelization

- Loop-carried dependences exist among iterations
- Parallel execution can be enabled via point-to-point synchronization among iterations of DOACROSS loop
  - Synchronizations are expressed using POST and WAIT

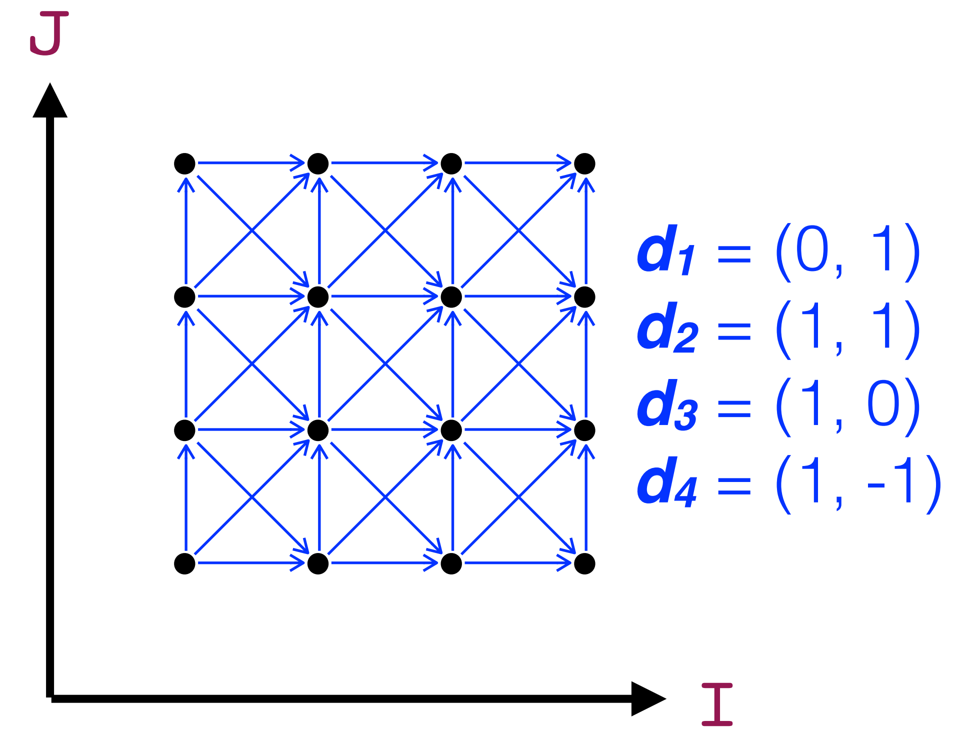
```
! ex.2
DOACROSS I = 1, N
  DO J = 1, M
    IF (I.GE.2) WAIT(I-1,J)
    A(J,I) = A(J-1,I) + A(J,I-1)
    POST(I,J)
  END DO
END DO
```



Doacross Parallelization

- Loop-carried dependences exist among iterations
- Parallel execution can be enabled via point-to-point synchronization among iterations of DOACROSS loop
  - Synchronizations are expressed using POST and WAIT

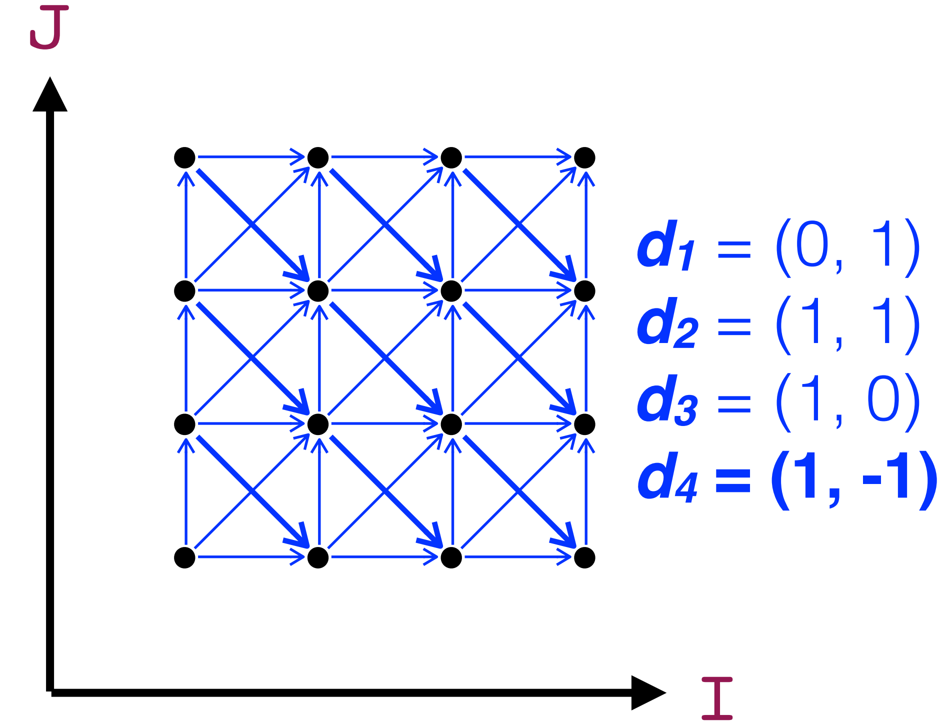
```
! ex.3
DOACROSS I = 1, N
  DO J = 1, M
    IF (I.GE.2) WAIT(…)
    A(J,I) = A(J-1,I) + A(J-1,I-1)
           + A(J,I-1) + A(J+1,I-1)
    POST(I,J)
  END DO
END DO
```



Doacross Parallelization

- Synchronizations are expressed using POST and WAIT
    - Dependence folding: Multiple dependence vectors can be covered by a single conservative pair of post-wait synchronization operations (due to transitivity)

```
! ex.3
DOACROSS I = 1, N
  DO J = 1, M
    IF (I.GE.2) WAIT(I-1,J+1)
    A(J,I) = A(J-1,I) + A(J-1,I-1)
           + A(J,I-1) + A(J+1,I-1)
    POST(I,J)
  END DO
END DO
```

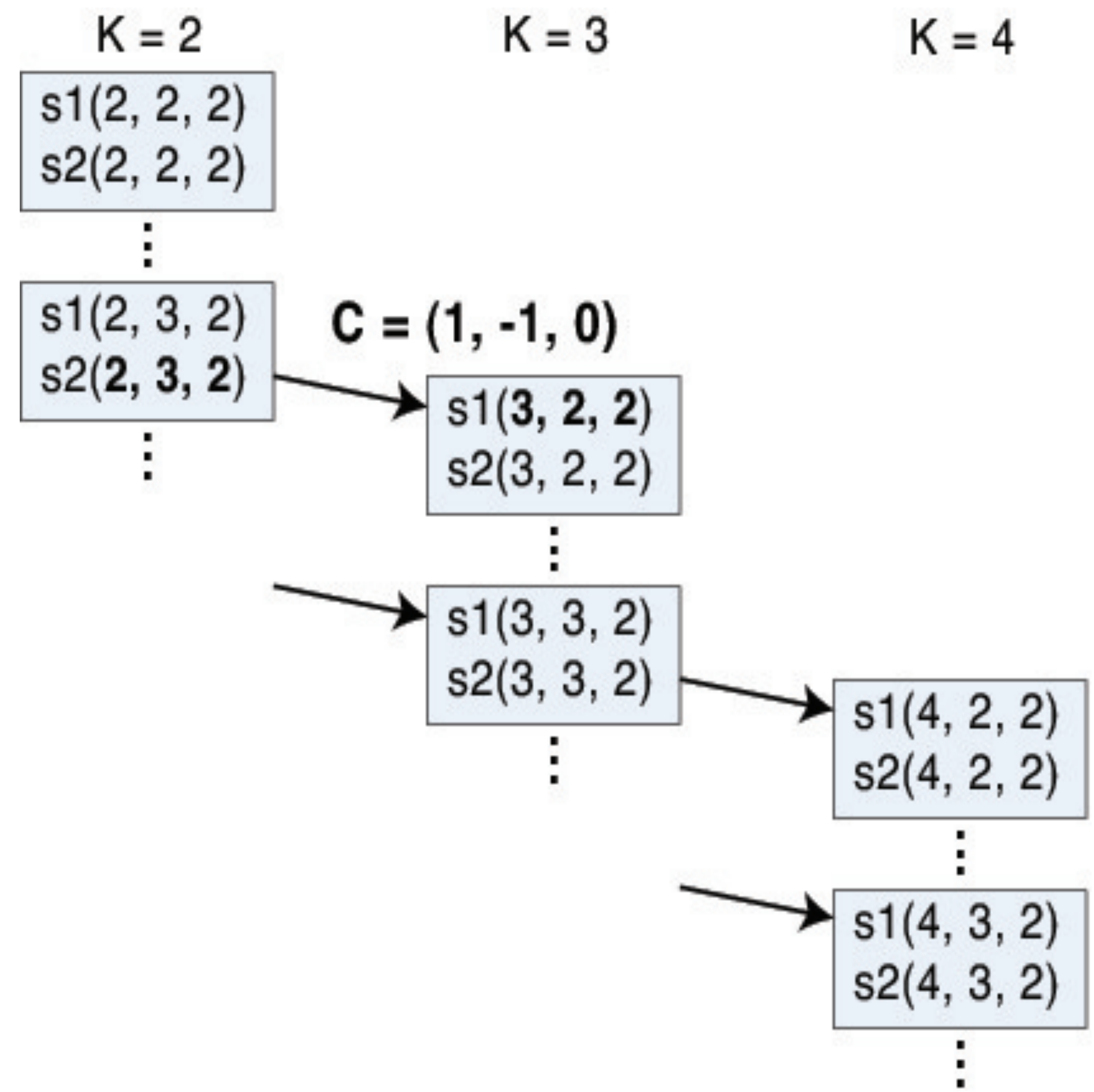


Dependence Folding

- Goal: Identify a single dependence vector that conservatively subsumes all loop-carried dependences in a doacross loop nest
  - Pros: reduce post-wait synchronization overhead
  - Cons: may give up some parallelism as a result
- Source statement for conservative dependence = Lexically Latest Source (LLS) statement in loop nest
- Sink statement for conservative dependence = Lexically Earliest Sink (LES) statement in loop nest
- Distances in conservative dependence vector can be computed using GCD and related operations
- Reference: “A Practical Approach to DOACROSS Parallelization”. Priya Unnikrishnan, Jun Shirako, Kit Barton, Sanjay Chatterjee, Raul Silvera, and Vivek Sarkar. International European Conference on Parallel and Distributed Computing (Euro-Par), August 2012.

Example of Dependence Folding: Poisson Benchmark

```
   DO K=2,N3-1
     DO J=2,N2-1
       DO I=2,N1-1
         WAIT (K-1,J+1,I) Use{A} Def{A}
s1 :     Z = B(1)*(A(I+1,J  ,K  )+A(I-1,J  ,K  )
  &                +A(I  ,J+1,K  )+A(I  ,J-1,K  )
  &                +A(I  ,J  ,K+1)+A(I  ,J  ,K-1))
  &           + B(2)*(A(I+1,J+1,K  )+A(I-1,J+1,K  )
  &                +A(I+1,J  ,K+1)+A(I-1,J  ,K+1)
  &                +A(I+1,J-1,K  )+A(I-1,J-1,K  )
  &                +A(I+1,J  ,K-1)+A(I-1,J  ,K-1)
  &                +A(I  ,J+1,K+1)+A(I  ,J-1,K+1)
  &                +A(I  ,J+1,K-1)+A(I  ,J-1,K-1))
s2 :      A(I,J,K) = (A(I,J,K) + Z)*0.5D0
          POST (K, J, I) Use{A} Def{A}
       END DO
     END DO
   END DO
```



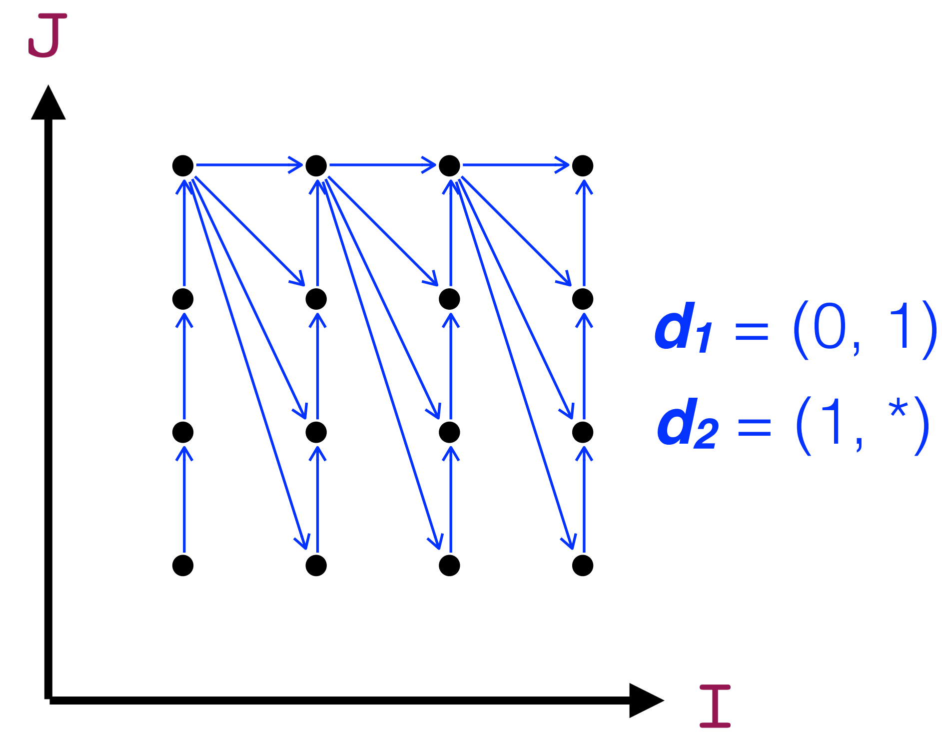
Lexically Latest Source = S2, Lexically Earliest Sink = S1, Conservative dependence vector from S2 to S1 = (1,-1, 0)

NOTE: & is just a line continuation character, and has no other semantics

Compilation Issues for Doacross

- Detection of doacross parallelism
  - Legality : need to insert synchronization operations that cover all dependences
  - Profitability : synchronization operations enable useful parallelism (considering overlap and overhead)
- Dependence folding
  - Input : all dependences in target nest
  - Output : a single conservative dependences that covers all original dependences

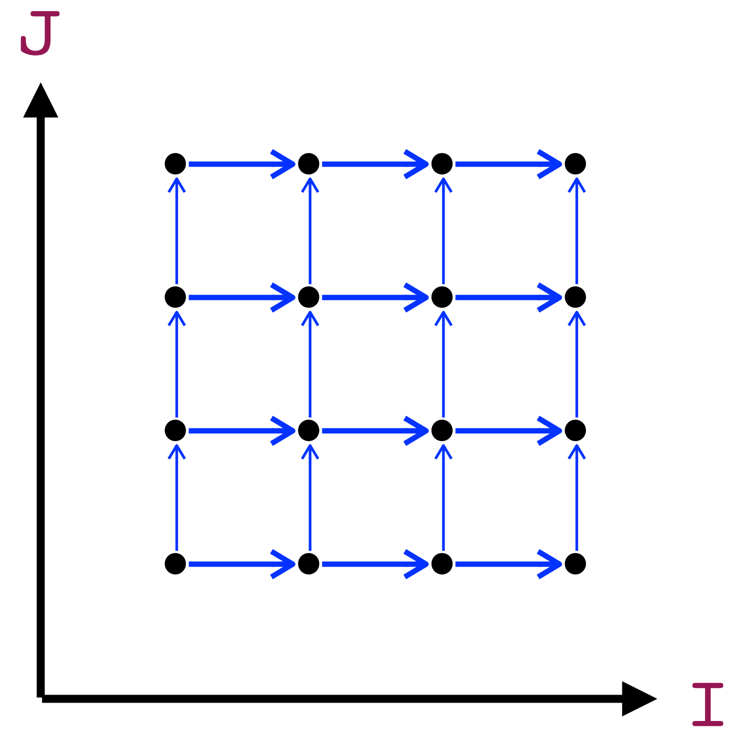
```
! ex.4
DO I = 1, N
  DO J = 1, M
    A(J,I) = A(J-1,I) + A(M,I-1)
  END DO
END DO
```



Compilation Issues for Doacross

- Compile-time granularity control
  - Loop unrolling / loop tiling

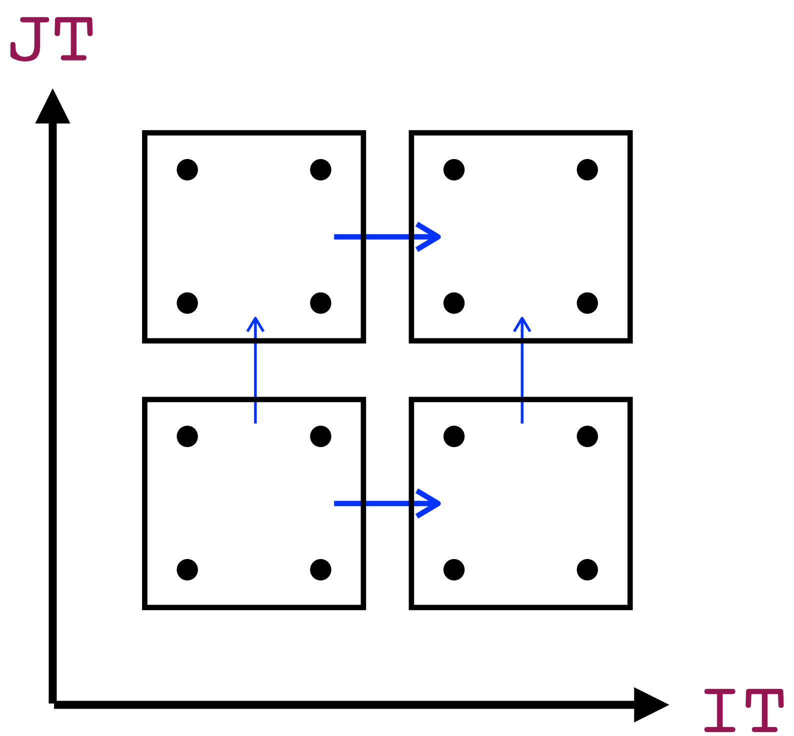
```
! ex.2
DOACROSS I = 1, N
  DO J = 1, M
    IF (I.GE.2) WAIT(I-1,J)
    A(J,I) = A(J-1,I) + A(J,I-1)
    POST(I,J)
  END DO
END DO
```



Compilation Issues for Doacross

- Compile-time granularity control
  - Loop unrolling / loop tiling

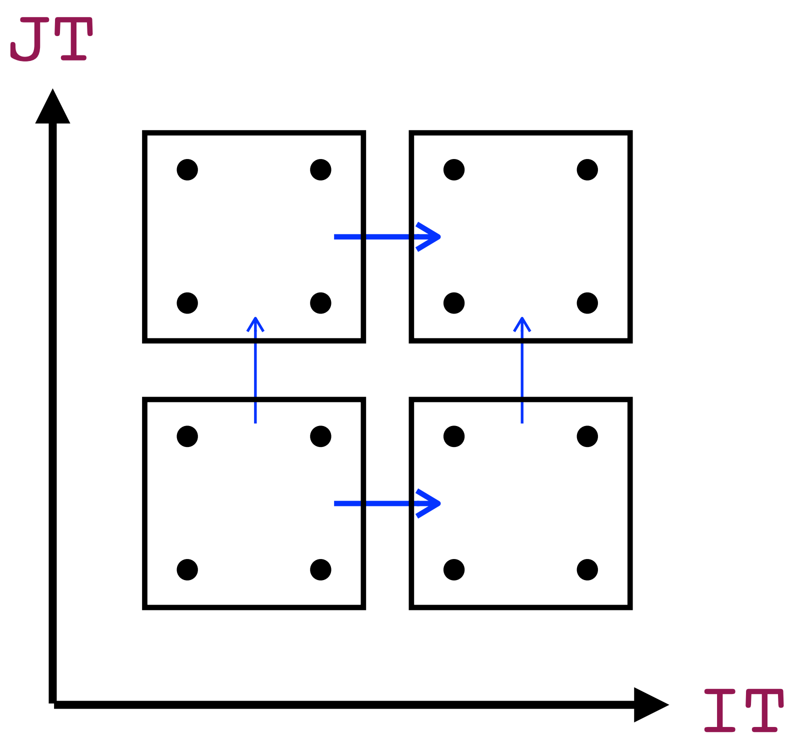
```
! ex.2
DOACROSS IT = 1, N, T1
  DO JT = 1, M, T2
    IF (IT.GE.2) WAIT(IT-1,JT)
    DO I = IT, MIN(IT+T1-1,N)
      DO J = JT, MIN(JT+T2-1,M)
        A(J,I) = A(J-1,I) + A(J,I-1)
      END DO
    END DO
    POST(IT,JT)
  END DO
END DO
```



Compilation Issues for Doacross

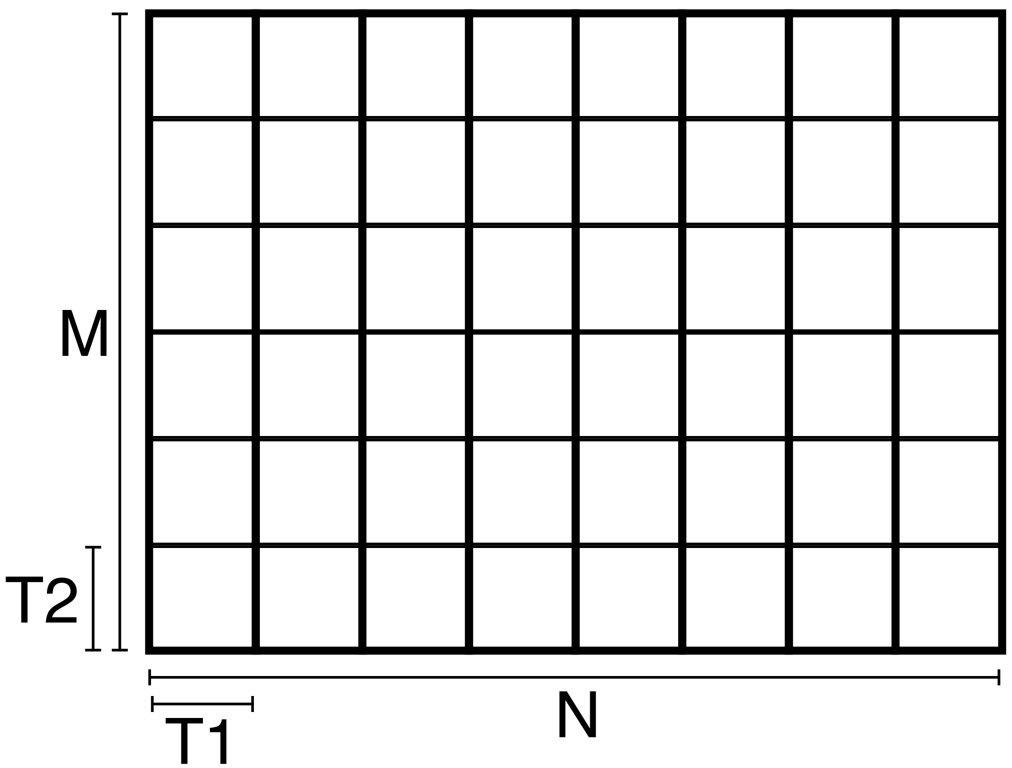
- Compile-time granularity control by loop tiling
  - Pros : increased computation granularity per synchronization
  - Cons : reduced parallelism
  - Challenge : selecting best tile sizes to balance pros and cons

```
! ex.2
DOACROSS IT = 1, N, T1
  DO JT = 1, M, T2
    IF (IT.GE.2) WAIT(IT-1,JT)
    DO I = IT, MIN(IT+T1-1,N)
      DO J = JT, MIN(JT+T2-1,M)
        A(J,I) = A(J-1,I) + A(J,I-1)
      END DO
    END DO
    POST(IT,JT)
  END DO
END DO
```



Granularity Control by Tiling

```
! ex.2
DOACROSS IT = 1, N, T1
  DO JT = 1, M, T2
    IF (IT.GE.2) WAIT(IT-1,JT)
    DO I = IT, MIN(IT+T1-1,N)
      DO J = JT, MIN(JT+T2-1,M)
        A(J,I) = A(J-1,I) + A(J,I-1)
      END DO
    END DO
    POST(IT,JT)
  END DO
END DO
```



N : outer loop count
M : inner loop count
T1 : outer tile size
T2 : inner tile size
P : number of processors

Parallelism constraint:

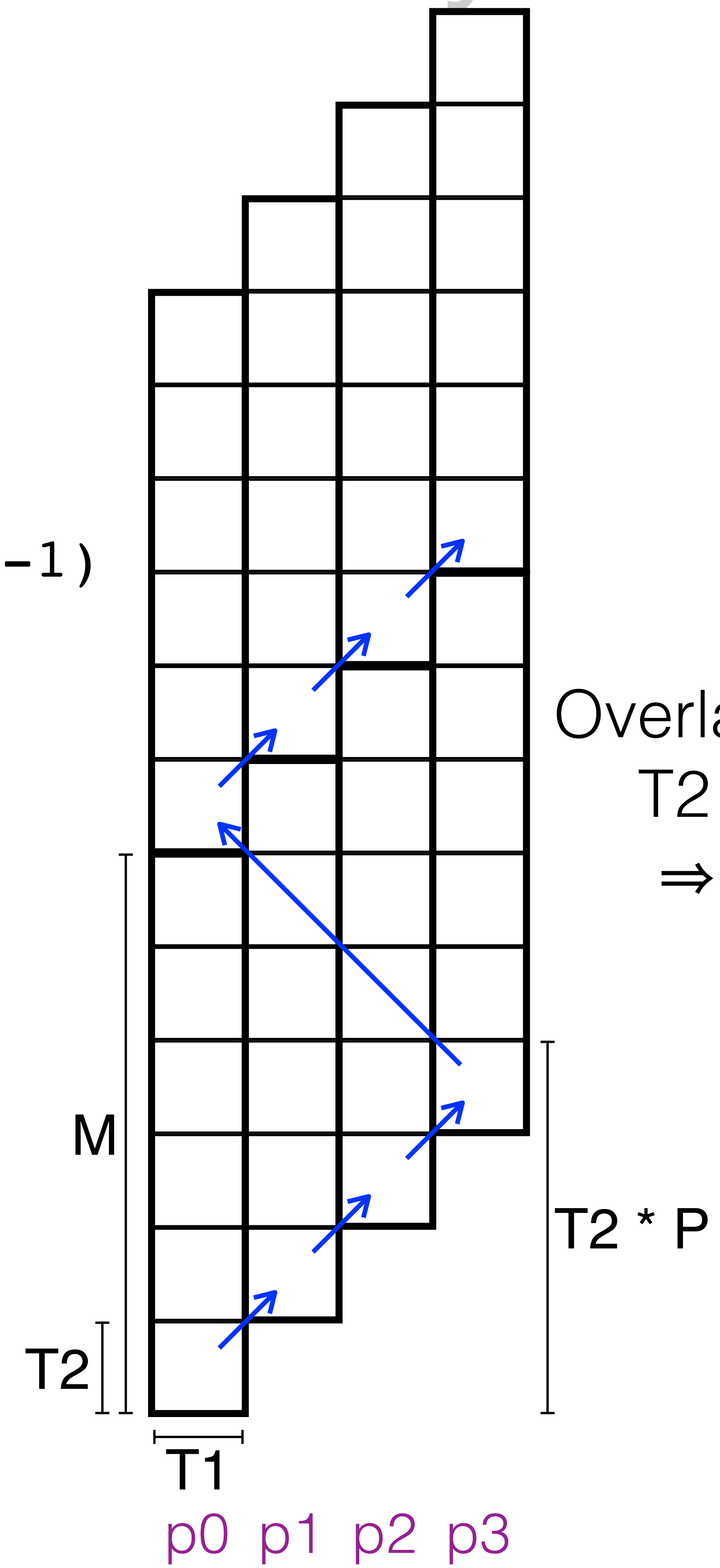
$N / T1 \geq P$

$\Rightarrow T1 \leq N / P$

Granularity Control by Tiling

```
! ex.2
DOACROSS IT = 1, N, T1
  DO JT = 1, M, T2
    IF (IT.GE.2) WAIT(IT-1,JT)
    DO I = IT, MIN(IT+T1-1,N)
      DO J = JT, MIN(JT+T2-1,M)
        A(J,I) = A(J-1,I) + A(J,I-1)
      END DO
    END DO
    POST(IT,JT)
  END DO
END DO
```

N : outer loop count
M : inner loop count
T1 : outer tile size
T2 : inner tile size
P : number of processors



Overlap constraint:
$T2 * P \le M$
$\Rightarrow T2 \le M / P$

Granularity Control by Tiling

- Parameters
  - N / M : outer / inner loop count
  - T1 / T2 : outer / inner tile size
  - P : number of processors
- Constraints on tile sizes
  - Parallelism : T1 $\leq$ N / P
  - Overlap : T2 $\leq$ M / P
- Ideal cost and overhead (ignoring synchronizations)
  - Total computation cost : N * M * cost_per_body
  - Cost per processor : N * M / P * cost_per_body
  - Delay to start last processor : T1 * T2 * (P - 1) * cost_per_body

Implementing POST and WAIT operations

Two approaches:

1. **Use event variables (Section 6.6.2 of textbook)**
    - Allocate an array of event variables, one per iteration
    - Perform POST and WAIT operations on event variables, e.g., POST (EV(I, J)) and WAIT (EV(I-1, J))
    - Pros: straightforward implementation approach
    - Cons: inefficient in space, not adaptable to available hardware parallelism
2. **Special runtime support for post/wait (OpenMP 4.1)**
    - Each processor maintains only n integer synchronization variables, where n is the number of loops in a doacross loop nest
    - Dependent iteration examines source iteration's sync variables to check ready condition
    - Pros: space-efficient (only n*P sync variables for P processors)
    - Cons: need runtime support in addition to compiler transformation

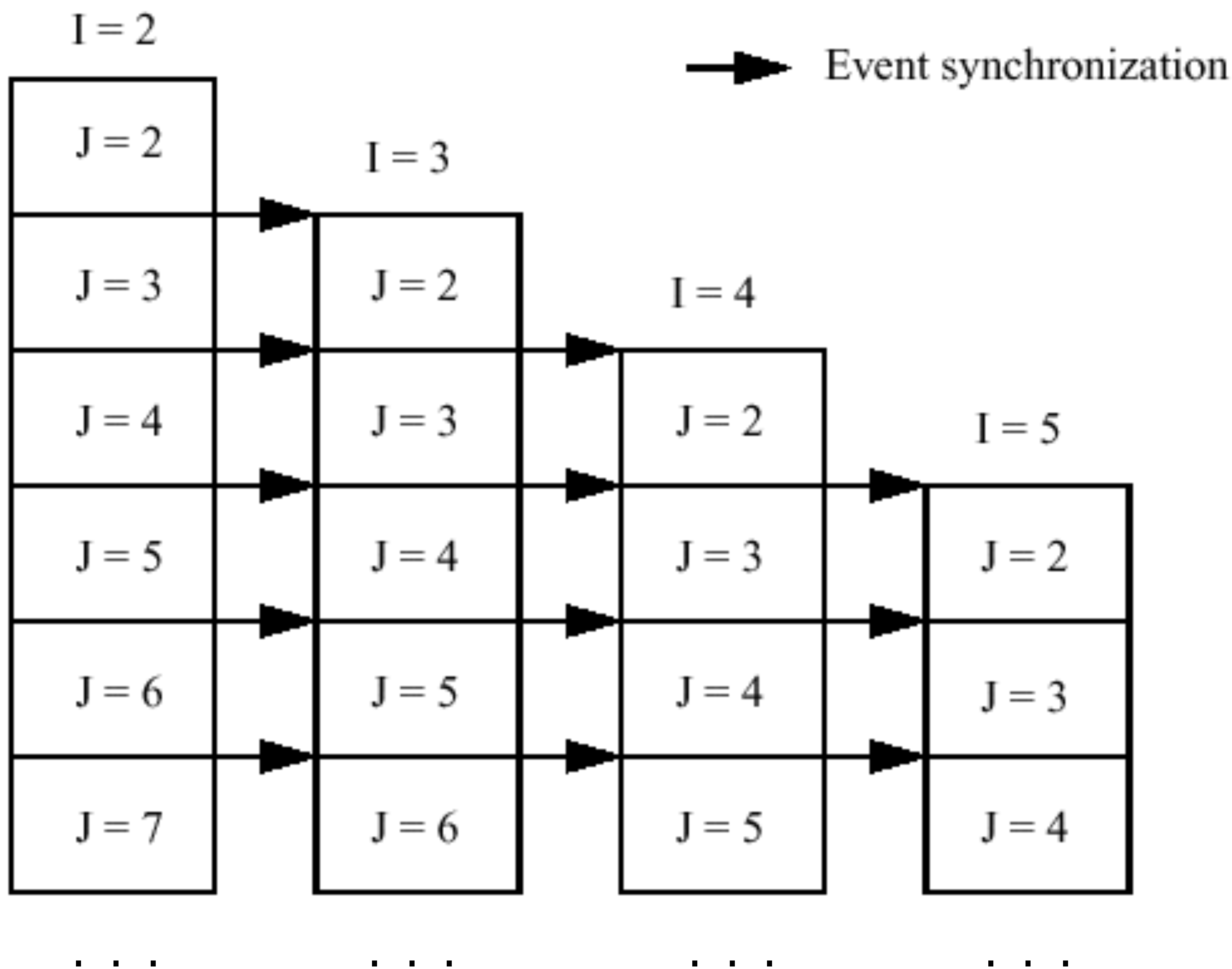
Example using event variables (Section 6.6.2)

```
DO I = 2, N-1
    DO J = 2, N-1
        A(I, J) = .25 * (A(I-1, J) + A(I, J-1) +
                         A(I+1, J) + A(I, J+1))
    ENDDO
ENDDO

==>

POST (EV(1, 2))
DOACROSS I = 2, N-1
   DO J = 2, N-1
        WAIT (EV(I-1, J))
        A(I, J) = .25 * (A(I-1, J) + A(I, J-1) +
                         A(I+1, J) + A(I, J+1))
        POST (EV(I, J))
    ENDDO
ENDDO
```

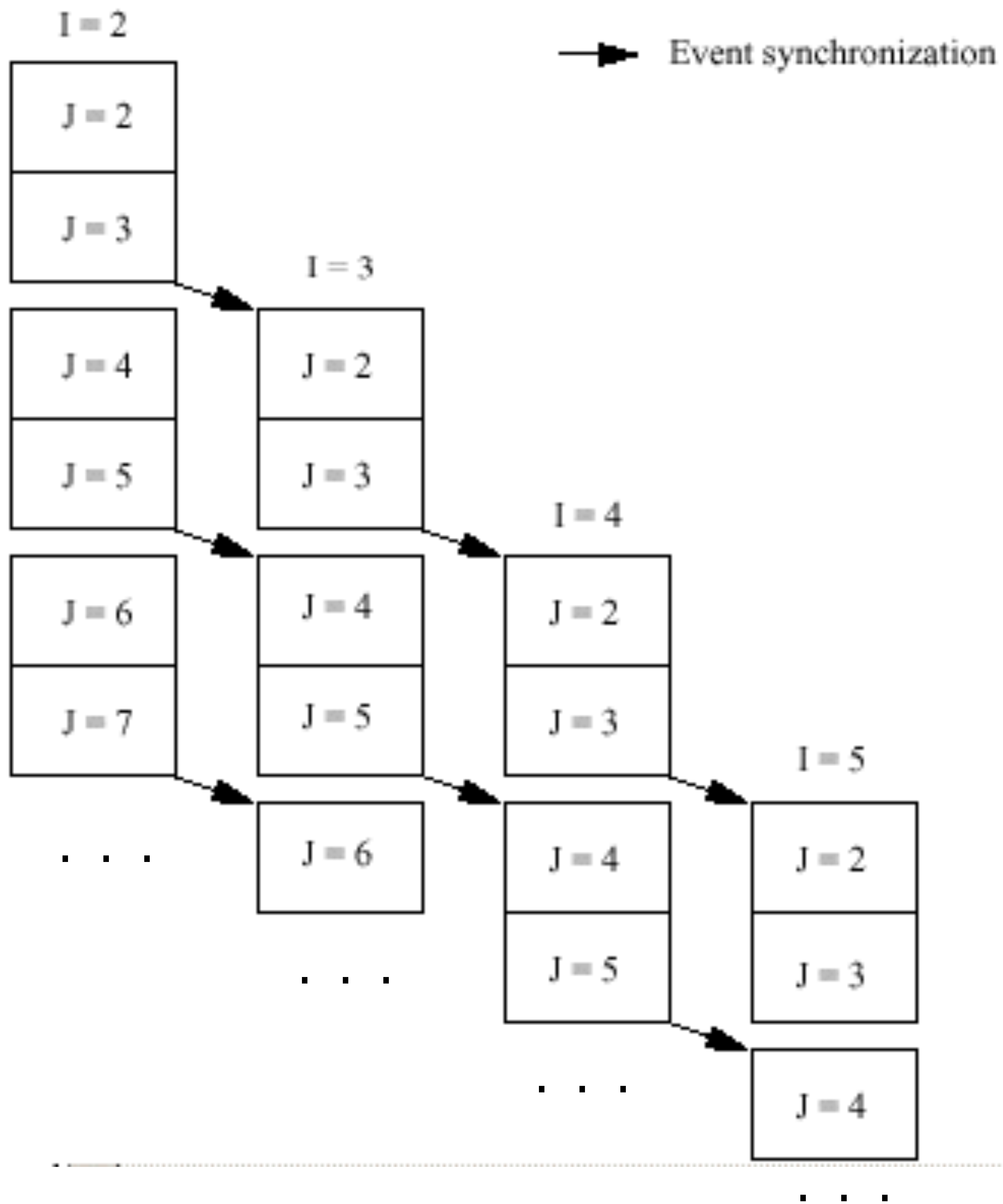
Example using event variables (contd)



Extension with 2x unroll/tiling

```
DO I = 2, N-1
    DO J = 2, N-1
        A(I, J) = .25 * (A(I-1, J) + A(I, J-1) +
                         A(I+1, J) + A(I, J+1))
    ENDDO
ENDDO
==>
POST (EV(1, 1))
DOACROSS I = 2, N-1
    K = 0
    DO J = 2, N-1, 2   ! TILE SIZE = 2
        K = K+1
        WAIT (EV(I-1,K))
        DO m = J, MIN(J+1, N-1)
            A(I, m) = .25 * (A(I-1, m) + A(I, m-1) +
                             A(I+1, m) + A(I, m+1))
        ENDDO
        POST (EV(I, K+1))
    ENDDO
ENDDO
```

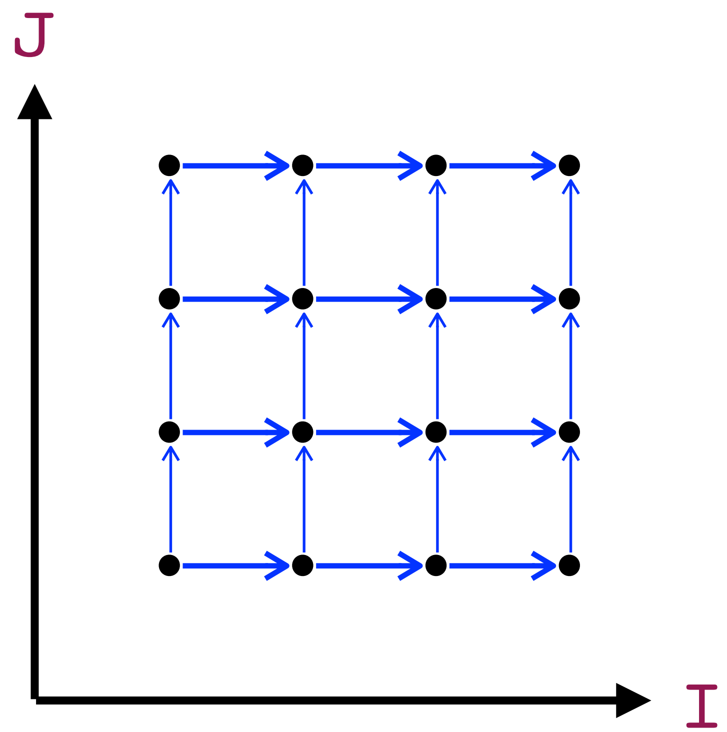
Extension with 2x unroll/tiling (contd)



Doacross Support in Future OpenMP

- `ordered(`*n*`)` : *n* specifies nest-level of doacross
- `depend(sink:` *vect*`)` : wait for iteration *vect* to reach `source`
- `depend(source)` : notify that current iteration reached
- Fortran code example

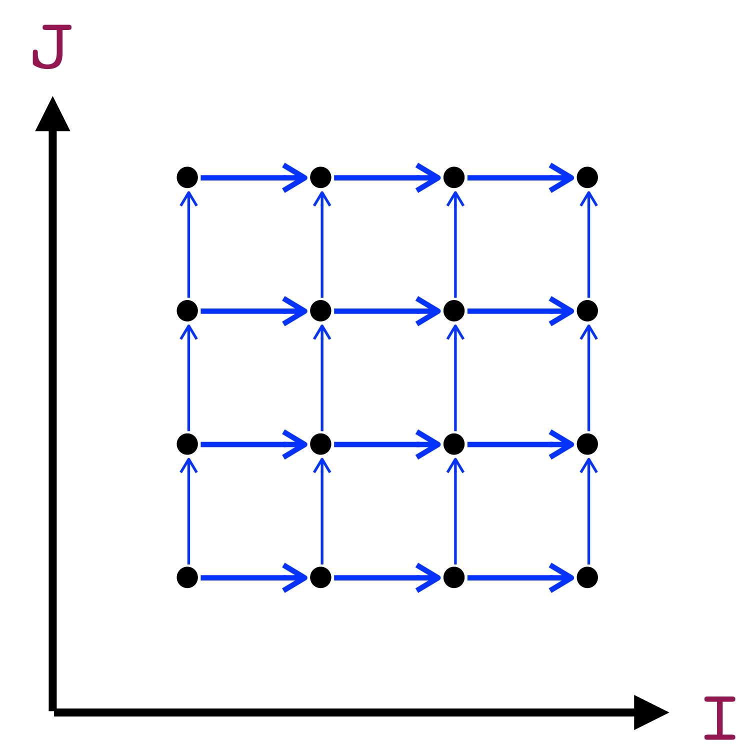
```
! ex.5a
!$omp for ordered(2)
      DO I = 1, N
        DO J = 1, M
          A(J,I) = FOO(…)             ! S1
!$omp ordered depend(sink: i-1,j)
!$omp&        depend(sink: j,i-1)
          B(J,I) = BAR(A(J,I), B(J,I-1),
                       B(J-1,I))   ! S2
!$omp ordered depend(source)
          C(J,I) = BAZ(B(J,I))     ! S3
        END DO
      END DO
```



Doacross Support in Future OpenMP

- **`ordered(`*n*`)`** : *n* specifies nest-level of doacross
- **`depend(sink:`** *vect***`)`** : wait for iteration *vect* to reach **`source`**
- **`depend(source)`** : notify that current iteration reached
- C code example

```
! ex.5b
#pragma omp for ordered(2)
for (i = 1; i < n; i++) {
  for (j = 1; j < m; j++) {
    A[i][j] = foo(i, j);          // S1
    #pragma omp ordered depend(sink: i-1,j) \\
                        depend(sink: j,i-1)
    B[i][j] = bar(A[i][j],
                  B[i-1][j],
                  B[i][j-1]);   // S2
    #pragma omp ordered depend(source)
    C[i][j] = baz(B[i][j]);     // S3
}
```



References

- A Practical Approach to DOACROSS Parallelization
  - Priya Unnikrishnan, Jun Shirako, Kit Barton, Sanjay Chatterjee, Raul Silvera, and Vivek Sarkar. International European Conference on Parallel and Distributed Computing (Euro-Par), August 2012
- Expressing DOACROSS Loop Dependences in OpenMP
  - Jun Shirako, Priya Unnikrishnan, Sanjay Chatterjee, Kelvin Li, and Vivek Sarkar. International Workshop on OpenMP (IWOMP), September 2013
- OpenMP Specification 4.1 (draft)
  - http://openmp.org/wp/openmp-specifications/