COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Krishna Palem
Prof. Vivek Sarkar
Department of Computer Science
Rice University
{palem,vsarkar}@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

Coarse-Grain Parallelism (contd)

Chapter 6 of Allen and Kennedy

- Acknowledgment: Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - http://www.cs.rice.edu/~ken/comp515/

Chapter 6 Summary

- Coarse-Grained Parallelism
  - Privatization
  - Loop distribution
  - Loop alignment
  - Loop fusion
  - Loop interchange
  - Loop reversal
  - Loop skewing
  - Pipeline parallelism
  - Scheduling

Scalar Privatization

- The analog of scalar expansion is privatization.
- Temporaries can be given separate namespaces for each iteration.

```
    DO I = 1,N
S1     T = A(I)
S2     A(I) = B(I)
S3     B(I) = T
    ENDDO
```

```
    PARALLEL DO I = 1,N
      PRIVATE t
S1    t = A(I)
S2    A(I) = B(I)
S3    B(I) = t
    ENDDO
```

Array Privatization

We need to privatize array variables.

For iteration J, upwards exposed variables are those exposed due to loop body without variables defined earlier.

```
     DO I = 1,100
S0      T(1)=X
L1      DO J = 2,N
S1        T(J) = T(J-1)+B(I,J)
S2        A(I,J) = T(J)
        ENDDO
     ENDDO
```

$$up(L_1) = \bigcup_{J=2}^{N} (\{T(J-1)\} \setminus \{T(n) : 2 \le n \le j\})$$

So for this fragment, T(1) is the only exposed variable.

Array Privatization

- Using this analysis, we get the following code:

```
      PARALLEL DO I = 1,100
           PRIVATE t(N)
S0         t(1) = X
L1         DO J = 2,N
S1            t(J) = t(J-1)+B(I,J)
S2            A(I,J)=t(J)
           ENDDO
        ENDDO
```

Loop Distribution

- Loop distribution can convert loop-carried dependences to loop-independent dependences.
- Consequently, it often creates opportunity for outer-loop parallelism.
- However, we must add extra barriers to keep distributed loops from executing out of order, so the overhead may override the parallel savings.

Loop Alignment

- Many carried dependencies are due to array alignment issues.
- If we can align all references, then dependencies would go away, and parallelism is possible.
- This is also related to Software Pipelining

```
DO I = 2,N
   A(I) = B(I)+C(I)
   D(I) = A(I-1)*2.0
ENDDO
```

```
DO I = 1,N   ! Aligned loop
   IF (I .GT. 1) A(I) = B(I)+C(I)
   IF (I .LT. N) D(I+1) = A(I)*2.0
ENDDO
```

```
D(2) = A(1)*2.0
DO I = 2,N-1   ! Pipelined loop
   A(I) = B(I)+C(I)
   D(I+1) = A(I)*2.0
ENDDO
A(N) = B(N)+C(N)
```

Alignment

- There are other ways to align the loop:

```
DO I = 2,N
  J = MOD(I+N-4,N-1)+2
  A(J) = B(J)+C
  D(I)=A(I-1)*2.0
ENDDO
```

```
D(2) = A(1)*2.0
DO I = 2,N-1
  A(I) = B(I)+C(I)
  D(I+1) = A(I)*2.0
ENDDO
A(N) = B(N)+C(N)
```

Code Replication

- If an array is involved in a recurrence, then alignment isn't possible.
- If two dependencies between the same statements have different dependency distances, then alignment doesn't work.
- We can fix the second case by replicating code:

```
DO I = 1,N
   A(I+1) = B(I)+C
   X(I) = A(I+1)+A(I)
ENDDO
```

```
DO I = 1,N
   A(I+1) = B(I)+C
   ! Replicated Statement
   IF (I .EQ 1) THEN
      t = A(I)
   ELSE
      t = B(I-1)+C
   END IF
   X(I) = A(I+1)+t
ENDDO
```

Strip Mining

- Converts available parallelism into a form more suitable for the hardware (assume THRESHOLD = minimum iters for parallel loop)

```
DO I = 1, N
   A(I) = A(I) + B(I)
ENDDO
```

==>

```
k = MAX(THRESHOLD, CEIL (N / P))
PARALLEL DO I = 1, N, k
        DO i = I, MIN(I + k-1, N)
                A(i) = A(i) + B(i)
        ENDDO
END PARALLEL DO
```

Loop Fusion

- Loop distribution was a method for separating parallel parts of a loop.
- Our solution attempted to find the maximal loop distribution.
- The maximal distribution often finds parallelizable components too small for efficient parallelism.
- Two obvious solutions:
  - Strip mine large loops to create larger granularity.
  - Perform maximal distribution, and then fuse together parallelizable loops.
  - Both solutions can be combined as well.

Fusion Safety: Fusion-Preventing Loop-Independent Dependences

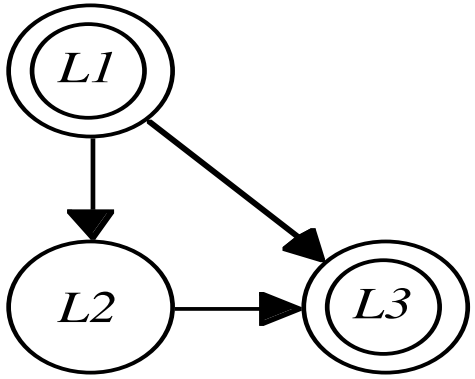
Definition: A loop-independent dependence between statements S1 and S2 in loops L1 and L2 respectively is fusion-preventing if fusing L1 and L2 causes the dependence to be carried by the combined loop in the opposite direction.

```
     DO I = 1,N
S1      A(I) = B(I)+C
     ENDDO
     DO I = 1,N
S2      D(I) = A(I+1)+E
     ENDDO
```

```
     DO I = 1,N
S1      A(I) = B(I)+C
S2      D(I) = A(I+1)+E
     ENDDO
```

Fusion Safety: Ordering Constraint

- We shouldn't fuse loops if the fusing will violate ordering of the dependence graph.
- Ordering Constraint: Two loops can't be validly fused if there exists a path of loop-independent dependencies between them containing a loop or statement not being fused with them i.e., if fusion will result in a cycle in the resulting loop-independent dependences



Fusing L1 with L3 violates the ordering constraint. {L1,L3} must occur both before and after the node L2.

Fusion Profitability

Parallel loops should generally not be merged with sequential loops.

Definition: An edge between two statements in loops L1 and L2 respectively is said to be parallelism-inhibiting if after merging L1 and L2, the dependence is carried by the combined loop.

```
     DO I = 1,N
S1      A(I+1) = B(I) + C
     ENDDO
     DO I = 1,N
S2      D(I) = A(I) + E
     ENDDO
```

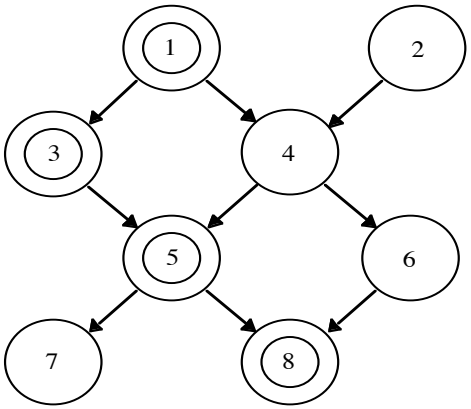
```
     DO I = 1,N
S1      A(I+1) = B(I) + C
S2      D(I) = A(I) + E
     ENDDO
```

Typed Fusion

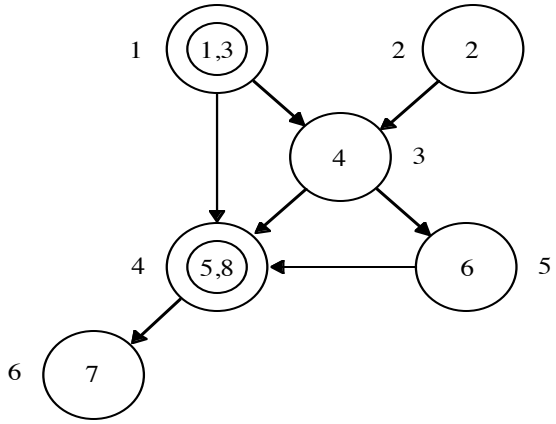
- We start by classifying loops into two types: parallel and sequential.
- We next gather together all edges that inhibit efficient fusion, (i.e., that connect a sequential and a parallel loops) and call them bad edges.
- Given a graph of loop-independent dependences (V,E), we want to obtain a graph (V',E') by merging vertices of V subject to the following constraints:
  - Bad Edge Constraint: vertices joined by a bad edge aren't fused.
  - Ordering Constraint: vertices joined by path containing non-parallel vertex aren't fused

Typed Fusion Example

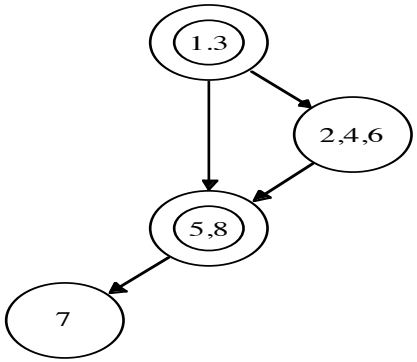
Original loop graph



After fusing parallel loops



After fusing sequential loops



Thus far …

- Single loop methods
  - Privatization
  - Loop distribution
  - Alignment
  - Code replication
  - Loop fusion
- Next, methods for perfect and imperfect loops

Loop Interchange

- Parallelization: move dependence-free loops to outermost level
- Theorem 6.3
  - In a perfect nest of loops, a particular loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only '=' entries

Motivation for Loop Interchange

```
DO I = 1, N
        DO J = 1, N
        A(I+1, J) = A(I, J) + B(I, J)    (<, =)
        ENDDO
ENDDO
```

- Parallelizing the J loop is OK for vectorization
- But inefficient for parallelization (N barriers)

Loop Interchange

```
PARALLEL DO J = 1, N
        DO I = 1, N
        A(I+1, J) = A(I, J) + B(I, J)
        ENDDO
END PARALLEL DO
```

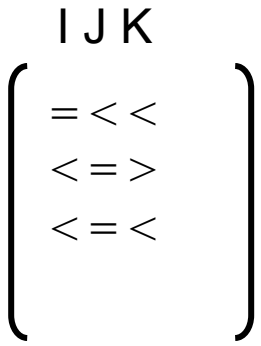
(=, <)

Loop Interchange

```
while L is not empty
   while there exist columns in M with all "="
      success := true;
      l:= loop with all "=" column;
      remove l from L;
      parallelize l at outer level;
      eliminate l's column from M;
   end;
   if L is not empty
      select_loop_and_interchange(L);
      l:= outermost loop; remove l from L; sequentialize l;
      remove column corresponding to l from M;
      remove all rows corresponding to dependences carried by l from M;
```

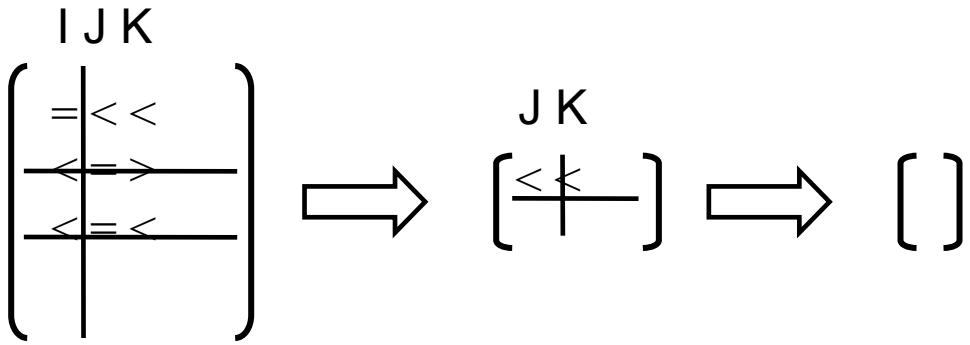
Loop Selection

```
DO I = 2, N+1
    DO J = 2, M+1
        DO K = 1, L
            A(I, J, K+1) = A(I, J-1, K) + A(I-1, J, K+2) + A(I-1, J, K)
        ENDDO
    ENDDO
ENDDO
```



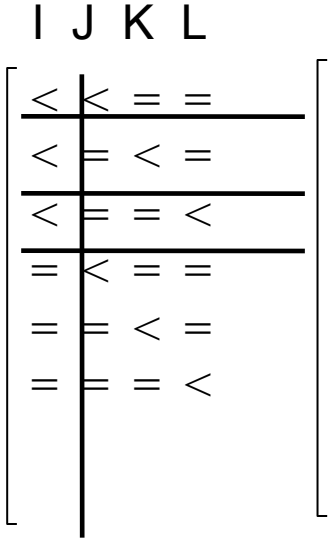
Loop Selection

```
DO I = 2, N+1
   DO J = 2, M+1
      PARALLEL DO K = 1, L
         A(I, J, K+1) = A(I, J-1, K) + A(I-1, J, K+2) + A(I-1, J, K)
      ENDDO
   ENDDO
ENDDO
```



Loop Selection

- Is it possible to derive a selection heuristic that provides optimal code?
  - NP-complete problem
- Assume simple approach of selecting the loop with the most '<' directions to eliminate the max number of rows from the direction matrix
  - Applying to this matrix will fail



Loop Selection

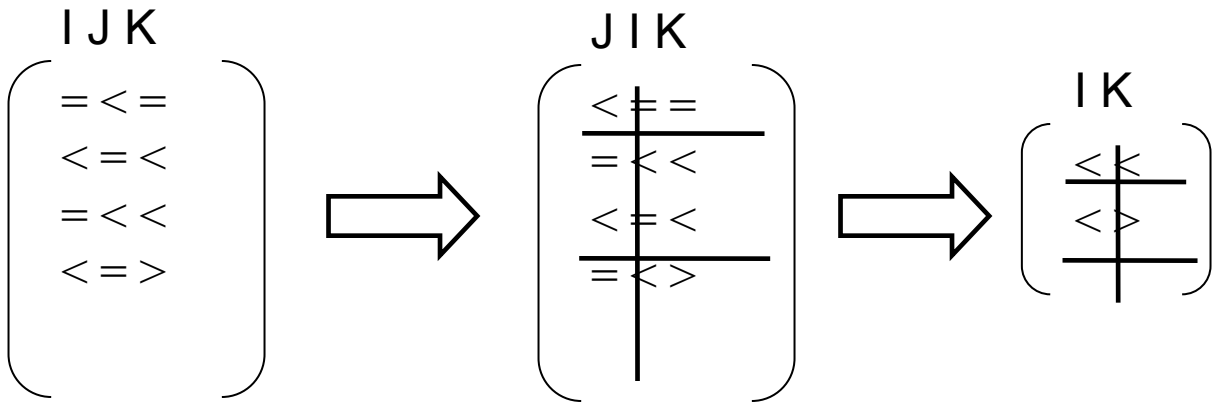
- Favor the selection of loops that must be sequentialized before parallelism can be uncovered
- If there exists a loop that can legally be moved to the outermost position and there is a dependence for which that loop has the only '<' direction, sequentialize that loop
- All such loops will need to be sequentialized at some point in the process

$$\begin{array}{c} J\ K\ L\ I \\ \left[\begin{array}{cccc} < & = & = & < \\ = & < & = & < \\ = & = & < & < \\ < & = & = & = \\ = & < & = & = \\ = & = & < & = \end{array}\right] \end{array}$$

Loop Selection

- Example of principles involved in heuristic loop selection

```
DO J = 2, M
    DO I = 2, N
            PARALLEL DO K = 2, L
            A(I, J, K) = A(I, J-1, K) + A(I-1, J, K-1) +
  A(I, J+1, K+1) + A(I-1, J, K+1)
            ENDDO
ENDDO  ENDDO
```



Loop Reversal

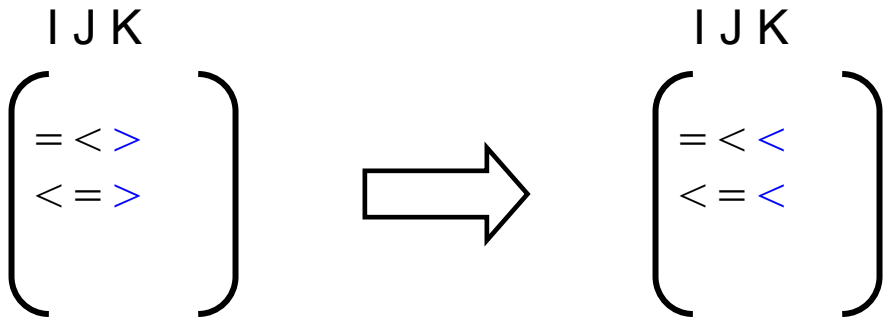
```
DO I = 2, N+1
    DO J = 2, M+1
        DO K = 1, L
            A(I, J, K) = A(I, J-1, K+1) + A(I-1, J, K+1)
        ENDDO
    ENDDO
ENDDO
```

I J K

$$\begin{pmatrix} = < > \\ < = > \end{pmatrix}$$

Loop Reversal

```
DO I = 2, N+1
    DO J = 2, M+1
        DO K = L, 1, -1
            A(I, J, K) = A(I, J-1, K+1) + A(I-1, J, K+1)
        ENDDO
    ENDDO
ENDDO
```



After Loop Reversal & Interchange

```
DO K = L, 1, -1
        PARALLEL DO I = 2, N+1
        PARALLEL DO J = 2, M+1
                A(I, J, K) = A(I, J-1, K+1) + A(I-1, J, K+1)
        END PARALLEL DO
        END PARALLEL DO
ENDDO
```

K I J

$$\begin{pmatrix} < & = & < \\ < & < & = \end{pmatrix}$$

- Increase the range of options available for loop selection heuristics

Loop Skewing

```
DO I = 2, N+1
    DO J = 2, M+1
        DO K = 1, L
            A(I, J, K) = A(I, J-1, K) + A(I-1, J, K)
            B(I, J, K+1) = B(I, J, K) + A(I, J, K)
        ENDDO
    ENDDO
ENDDO
```

I J K

$$\begin{pmatrix} = < = \\ < = = \\ = = < \\ = = = \end{pmatrix}$$

Loop Skewing

- Skewed using k = K + I + J yield:

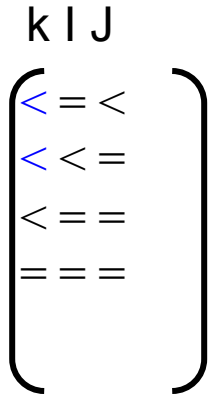
```
DO I = 2, N+1
        DO J = 2, M+1
        DO k = I+J+1, I+J+L
                A(I, J, k-I-J) = A(I, J-1, k-I-J) + A(I-1,
J, k-I-J)
                B(I, J, k-I-J+1) = B(I, J, k-I-J) + A(I, J,
k-I-J)
        ENDDO
        ENDDO
ENDDO
```

I J k

$$\begin{pmatrix} = & < & < \\ < & = & < \\ = & = & < \\ = & = & = \end{pmatrix}$$

Loop Skewing

```
DO k = 5, N+M+1
    PARALLEL DO I = MAX(2, k-M-L-1), MIN(N+1, k-L-2)
        PARALLEL DO J = MAX(2, k-I-L), MIN(M+1, k-I-1)
            A(I, J, k-I-J) = A(I, J-1, k-I-J) + A(I-1, J, k-I-J)
            B(I, J, k-I-J+1) = B(I, J, k-I-J) + A(I, J, k-I-J)
        ENDDO
    ENDDO
ENDDO
```



Loop Skewing

- Transforms skewed loop into one that can be interchanged to the outermost position without changing the meaning of the program
- Can be used to transform the skewed loop in such a way that, after outward interchange, it will carry all dependences formerly carried by the loop with respect to which it is skewed

$$\begin{matrix} & k & l & J \\ \begin{pmatrix} \end{pmatrix} & < & = & < \\ & < & < & = \\ & < & = & = \\ & = & = & = \end{matrix}$$

3

Loop Skewing

- Selection Heuristics
  1. Parallelize as many loops as possible
  2. Sequentialize at most one loop to find parallelism in the current outermost loop
  3. If 1 and 2 fails, try skewing
  4. If 3 fails, sequentialize the loop that can be moved to the outermost position and cover the most other loops

Pipeline Parallelism

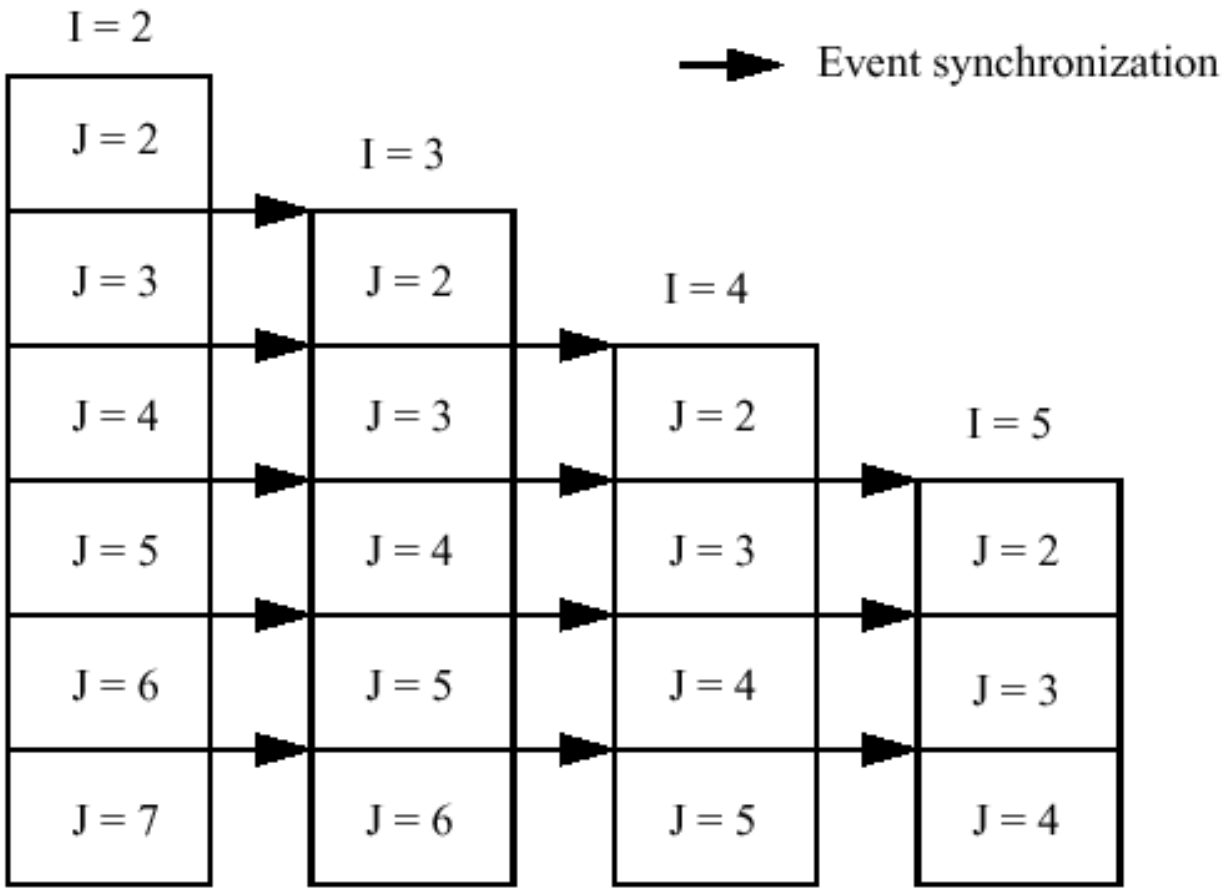
- Fortran command DOACROSS
- Useful where parallelization is not available
- High synchronization costs on old multiprocessors
  - Cheaper on-chip synchronization on multicore

```
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
```

Pipeline Parallelism

```
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
```

Pipeline Parallelism



Pipeline Parallelism with Strip Mining

```
POST (EV(1, 1))
DOACROSS I = 2, N-1
    K = 0
    DO J = 2, N-1, 2   ! CHUNK 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
```

Pipeline Parallelism