COMP 515: Advanced Compilation for Vector and Parallel Processors

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

Announcement

- Feb 17th class is cancelled --- please email Bob Garcia (rxg@rice.edu) with your availability for a make-up class on Feb 20th

Acknowledgments

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

Pipeline Parallelism

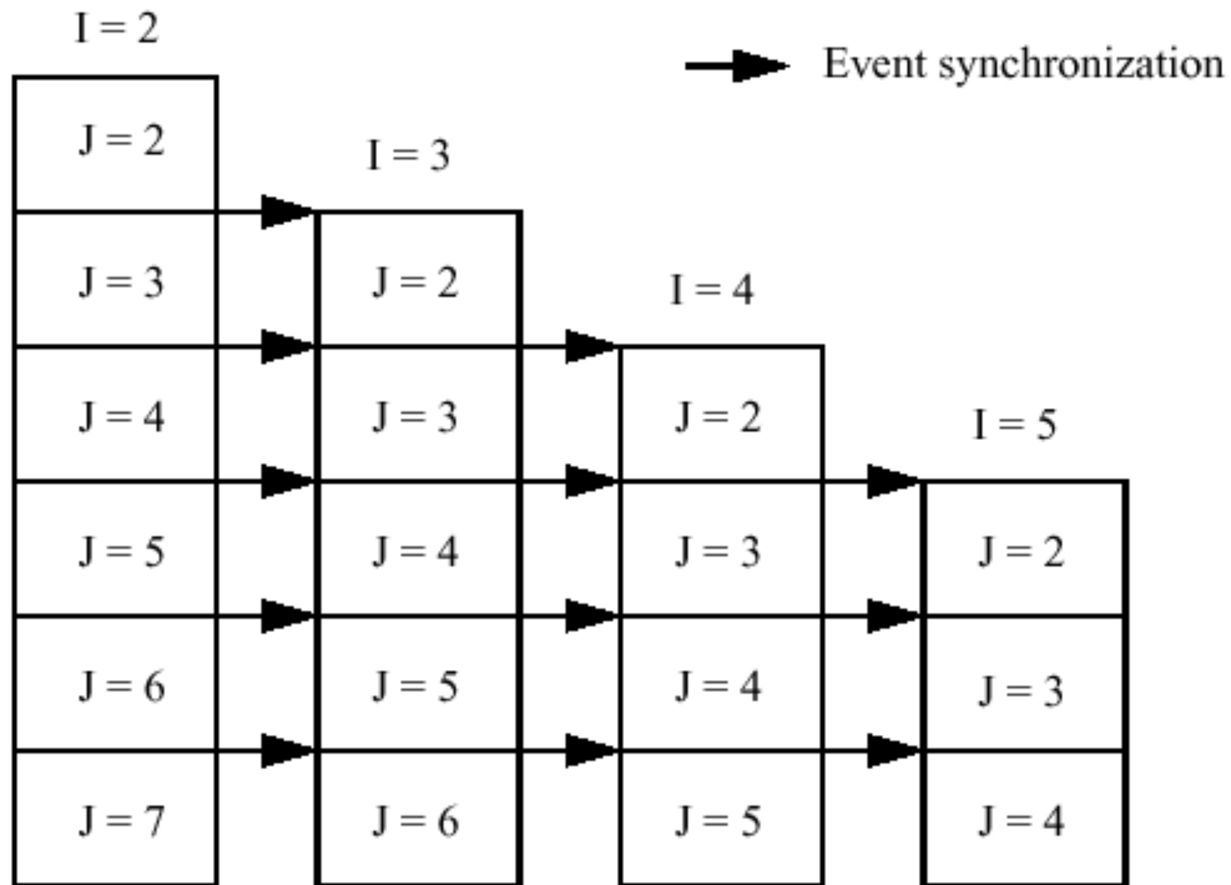
- Fortran command DOACROSS
- Useful where parallelization is not available
- High synchronization costs on old multiprocessors
  - Cheaper on-chip synchronization on multicore

```
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

Event synchronization

I = 2

J = 2

J = 3

J = 4

J = 5

J = 6

J = 7

I = 3

J = 2

J = 3

J = 4

J = 5

J = 6

I = 4

J = 2

J = 3

J = 4

J = 5

I = 5

J = 2

J = 3

J = 4

Phasers: a Unification of Barrier and Point-to-Point Synchronizations

- Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization. Jun Shirako, David Peixotto, Vivek Sarkar, William Scherer. Proceedings of the 2008 ACM International Conference on Supercomputing (ICS), June 2008.
- Phaser Accumulators: a New Reduction Construct for Dynamic Parallelism. Jun Shirako, David Peixotto, Vivek Sarkar, William Scherer. 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2009 (to appear).

Overview of Phasers

- Designed to handle multiple communication patterns
  - Collective Barrier
  - Point-to-point synchronization
- Dynamic parallelism
  - # activities synchronized on phaser can vary dynamically
- Support for "single" statements
  - Phase ordering property
- Deadlock freedom in absence of explicit wait operations
- Amenable to efficient implementation
  - Lightweight local-spin multicore implementation in Habanero project
- Extension of X10 clocks

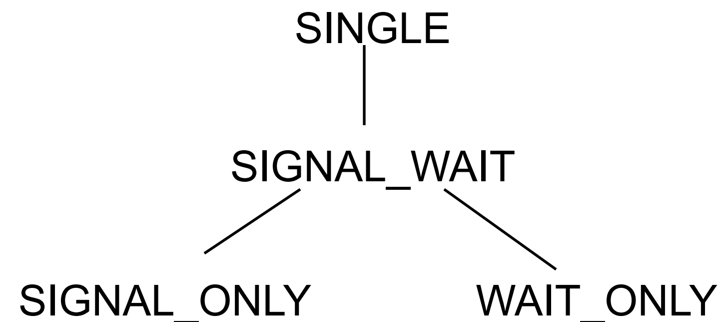
Collective and Point-to-point Synchronization with Phasers

phaser ph = new phaser(MODE);

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

async phased (ph1<MODE1>, ph2<MODE2>, …) 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;

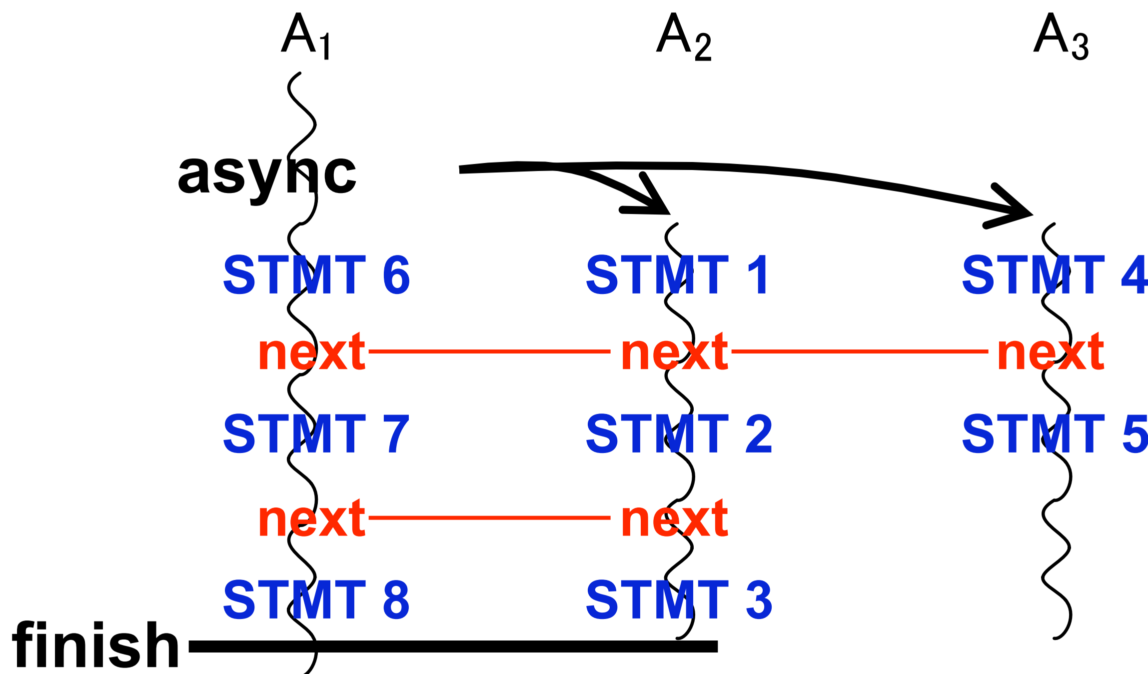
- Perform ph.signal() on each phaser that activity is registered on with a signal capability

Barrier Synchronization with Phasers

```
finish {
  delta.f = epsilon+1; iters.i = 0;
  phaser ph = new phaser();
  foreach (point[j]:[1:n]) phased { // will be registered with
    ph
      while ( delta.f > epsilon ) {
        newA[j] = (oldA[j-1]+oldA[j+1])/2.0f ;
        diff[j] = Math.abs(newA[j]-oldA[j]);
        next { // Barrier with "single" statement
          delta.f = diff.sum(); iters.i++;
        }
        temp = newA; newA = oldA; oldA = temp;
      } // while
  } // foreach
} // finish
```

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
}
```



A_1 , A_2 , A_3 are registered on phaser ph

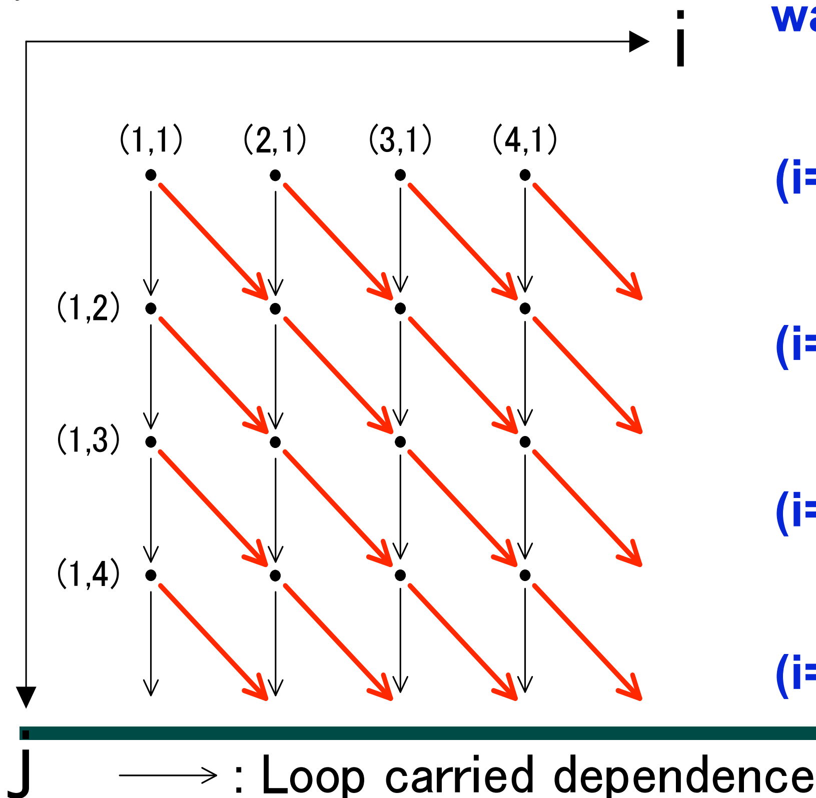
All-to-all barrier

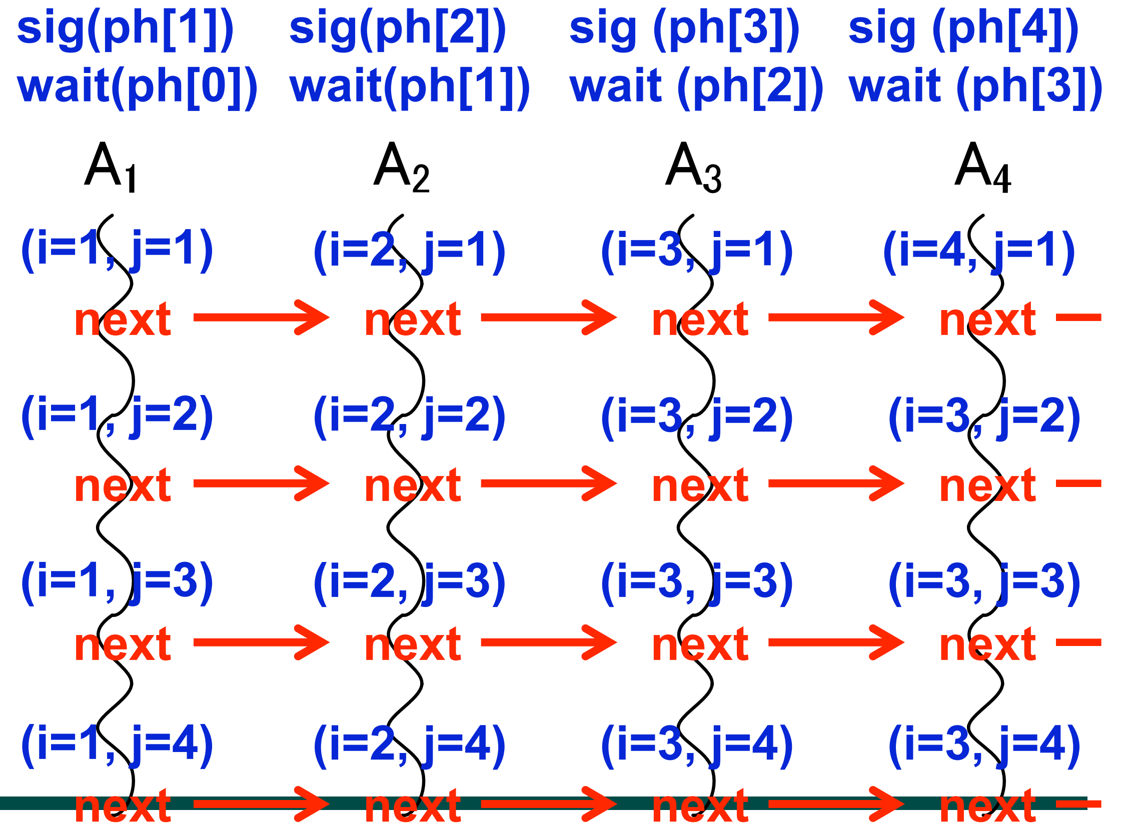
Dynamic parallelism

activities registered on phaser can vary

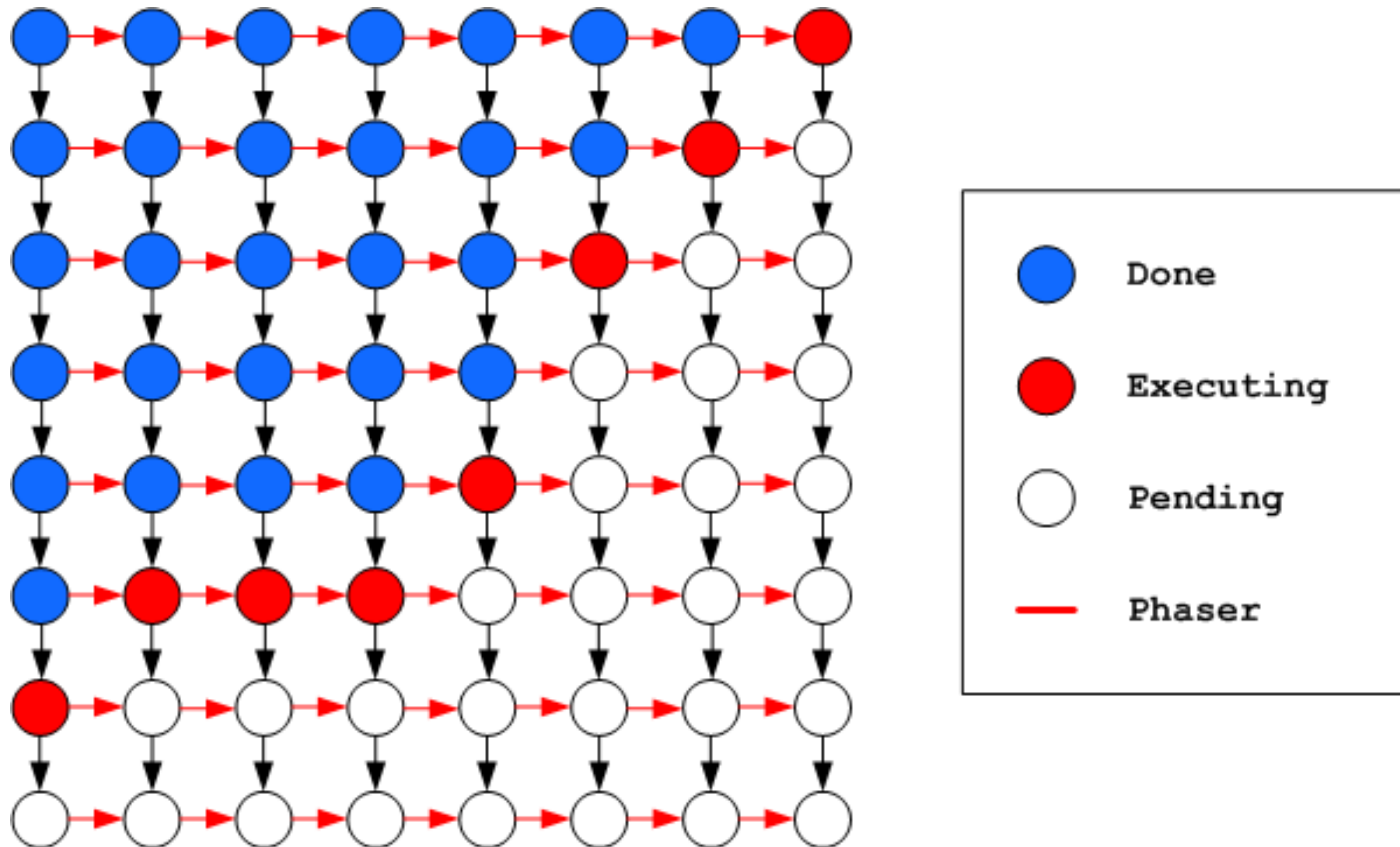
Example of Pipeline Parallelism with Phasers

```
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
```





Example of Pipeline Parallelism with Phasers (contd)



Scheduling Parallel Work

- Let σ_0 = serial scheduling overhead per processor
  - ➔ Contributes p * σ_0 overhead to parallel execution time for p processors
- N = number of iterations in parallel loop
- B = sequential execution time for one iteration
- Then, parallel execution is slower than serial execution if

$$\sigma_0 \geq (NB)/p$$

- Bakery-counter scheduling
  - —High synchronization overhead

Guided Self-Scheduling

- Reduces synchronization overhead while still adapting to load imbalances
  - Schedules groups of iterations unlike the bakery counter method
  - Going from large to small chunks of work
- Keep all processors busy at all times
- Iterations dispensed at step t when N_t iterations remain:

$$x = \left\lceil \frac{N_t}{p} \right\rceil$$

- Alternatively, we can have GSS(k) that guarantees that all blocks handed out are of size k or greater

Guided Self-Scheduling

- *GSS(1)*

Step(t)	Processor	No. Remaining	No. Allocated	Iterations Done
1	P1	20	5	1-5
2	P2	15	4	6-9
3	P3	11	3	10-12
4	P4	8	2	13,14
5	P4	6	2	15,16
6	P3	4	1	17
7	P2	3	1	18
8	P4	2	1	19
9	P1	1	1	20

Chapter 6 Summary

- Coarse-Grained Parallelism
  - Privatization
  - Loop distribution
  - Loop alignment
  - Loop fusion
  - Loop interchange
  - Loop reversal
  - Loop skewing
  - Profitability-based methods
  - Pipeline parallelism
  - Scheduling
- Next: Handling Control Flow

Control Dependences

Chapter 7

Control Dependences

- **Constraints posed by control flow**

```
         DO 100 I = 1, N
S1               IF (A(I-1).GT. 0.0) GO TO 100
S2                  A(I) = A(I) + B(I)*C
100    CONTINUE
```

$S_2 \; \delta_1 \; S_1$

If we vectorize by...

```
S2       A(1:N) = A(1:N) + B(1:N)*C
      DO 100 I = 1, N
S1             IF (A(I-1).GT. 0.0) GO TO 100
100      CONTINUE
```

…we get the wrong answer

- **We are missing dependences**
- **There is a dependence from S_1 to S_2 - a control dependence**

Control Dependences

- Two strategies to deal with control dependences:
  - If-conversion: expose by converting to data dependences. Used for vectorization
    - Also supported in SIMT hardware (e.g., GPGPUs) by masking out statements with control conditions = false
  - Explicitly expose as control dependences. Used for automatic parallelization

If-conversion

- **Underlying Idea: Convert statements affected by branches to conditionally executed statements**

```
      DO 100 I = 1, N
S1              IF (A(I-1).GT. 0.0) GO TO 100
S2                     A(I) = A(I) + B(I)*C
100   CONTINUE
```

can be converted to:

```
     DO I = 1, N
        IF (A(I-1).LE. 0.0) A(I) = A(I) + B(I)*C
     ENDDO
```

If-conversion

```
     DO 100 I = 1, N
S1            IF (A(I-1).GT. 0.0) GO TO 100
S2                  A(I) = A(I) + B(I) * C
S3                  B(I) = B(I) + A(I)
100  CONTINUE
```

- **can be converted to:**

```
      DO 100 I = 1, N
S2            IF (A(I-1).LE. 0.0) A(I) = A(I) + B(I) * C
S3            IF (A(I-1).LE. 0.0) B(I) = B(I) + A(I)
100   CONTINUE
```

- **vectorize using the Fortran WHERE statement:**

```
       DO 100 I = 1, N
S2        IF (A(I-1).LE. 0.0) A(I) = A(I) + B(I) * C
100    CONTINUE
S3       WHERE (A(0:N-1).LE. 0.0) B(1:N) = B(1:N) + A(1:N)
```

If-conversion

- If-conversion assumes a target notation of guarded execution in which each statement implicitly contains a logical expression controlling its execution

```
S1      IF (A(I-1).GT. 0.0) GO TO 100
S2                A(I) = A(I) + B(I)*C
100   CONTINUE
```

- with guarded execution instead:

```
S1      M = A(I-1).GT. 0.0
S2      IF (.NOT. M) A(I) = A(I) + B(I)*C
100  CONTINUE
```

Branch Classification

- **Forward Branch:** transfers control to a target that occurs lexically after the branch but at the same level of nesting
- **Backward Branch:** transfers control to a statement occurring lexically before the branch but at the same level of nesting
- **Exit Branch:** terminates one or more loops by transferring control to a target outside a loop nest

If-conversion

- **If-conversion is a composition of two different transformations:**
  1. **Branch relocation**
  2. **Branch removal**

Branch removal

- **Basic idea:**
  - Make a pass through the program.
  - Maintain a Boolean expression cc that represents the condition that must be true for the current expression to be executed
  - On encountering a branch, conjoin the *controlling expression* into cc
  - On encountering a target of a branch, its *controlling expression* is disjoined into cc

Branch Removal: Forward Branches

- Remove forward branches by inserting appropriate guards

```
      DO 100 I = 1,N
C1          IF (A(I).GT.10) GO TO 60
20          A(I) = A(I) + 10
C2              IF (B(I).GT.10) GO TO 80
40              B(I) = B(I) + 10
60              A(I) = B(I) + A(I)
80       B(I) = A(I) - 5
     ENDDO
```

```
      DO 100 I = 1,N
        m1 = A(I).GT.10
20      IF(.NOT.m1) A(I) = A(I) + 10
        IF(.NOT.m1) m2 = B(I).GT.10
40      IF(.NOT.m1.AND..NOT.m2) B(I) = B(I) + 10
60      IF(.NOT.m1.AND..NOT.m2.OR.m1)A(I) = B(I) + A(I)
80      IF(.NOT.m1.AND..NOT.m2.OR.m1.OR..NOT.m1
        .AND.m2) B(I) = A(I) - 5
     ENDDO
```

Branch Removal: Forward Branches

- **We can simplify to:**

```
    DO 100 I = 1,N
        m1 = A(I).GT.10
20              IF(.NOT.m1) A(I) = A(I) + 10
       IF(.NOT.m1) m2 = B(I).GT.10
40              IF(.NOT.m1.AND..NOT.m2)
       B(I) = B(I) + 10
60              IF(m1.OR..NOT.m2)
       A(I) = B(I) + A(I)
80              B(I) = A(I) - 5
    ENDDO
```

- **vectorize to:**

```
    m1(1:N) = A(1:N).GT.10
20  WHERE(.NOT.m1(1:N)) A(1:N) = A(1:N) + 10
    WHERE(.NOT.m1(1:N)) m2(1:N) = B(1:N).GT.10
40    WHERE(.NOT.m1(1:N).AND..NOT.m2(1:N))
     B(1:N) = B(1:N) + 10
60    WHERE(m1(1:N).OR..NOT.m2(1:N))
     A(1:N) = B(1:N) + A(1:N)
80    B(1:N) = A(1:N) - 5
```