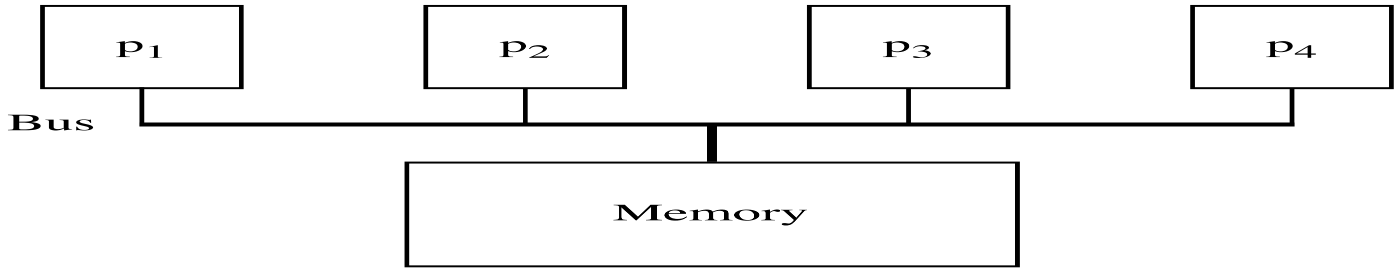
Coarse-Grained Parallelism

Variable Privatization, Loop Alignment, Loop Fusion, Loop interchange and skewing, Loop Strip-mining

Introduction

- Our previous loop transformations target vector and superscalar architectures
  - Now we target symmetric multiprocessor machines
  - The difference lies in the granularity of parallelism
- Symmetric multi-processors accessing a central memory
  - The processors are unrelated, and can run separate processes/threads
  - Starting processes and process synchronization are expensive
  - Bus contention can cause slowdowns
- Program transformations
  - Privatization of variables; loop alignment; shift parallel loops outside; loop fusion



Privatization of Scalar Variables

- Temporaries have separate namespaces
  - Definition: A scalar variable x in a loop L is said to be privatizable if every path from the loop entry to a use of x inside the loop passes through a definition of x
  - Alternatively, a variable x is private if the SSA graph doesn’t contain a phi function for x at the loop entry
  - Compare to the scalar expansion transformation

```
    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

What about privatizing array variables?

```
    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
```

```
   PARALLEL DO I = 1,100
        PRIVATE t
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 Alignment

- Many carried dependencies are due to alignment issues
  - Solution: align loop iterations that access common references
- Profitability: alignment does not work if
  - There is a dependence cycle
  - Dependences between a pair of statements have different distances

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

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

Alignment and Replication

- Replicate computation in the mis-aligned iteration

```
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
     X(I) = A(I+1)+A(1)
   ELSE
     X(I) = A(I+1)+(B(I-1)+C)
   END IF
ENDDO
```

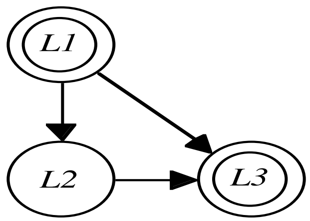
Theorem: Alignment, replication, and statement reordering are sufficient to eliminate all carried dependencies in a single loop containing no recurrence, and in which the distance of each dependence is a constant independent of the loop index

Loop Distribution and Fusion

- Loop distribution eliminates carried dependences by separating them across different loops
  - However, synchronization between loops may be expensive
  - Good only for fine-grained parallelism
- Coarse-grained parallelism requires sufficiently large parallel loop bodies
  - Solution: fuse parallel loops together after distribution
  - Loop strip-mining can also be used to reduce communication
- Loop fusion is often applied after loop distribution
  - Regrouping of the loops by the compiler

Loop Fusion

- Transformation: opposite of loop distribution
  - Combine a sequence of loops into a single loop
  - Iterations of the original loops now intermixed with each other
- Ordering Constraint
  - Cannot bypass statements with dependences both from and to the fused loops
- Safety: cannot have fusion-preventing dependences
  - Loop-independent dependences become backward carried after fusion



Fusing L1 with L3 violates the ordering constraint.

```
    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
```

Loop Fusion Profitability

- Parallel loops should generally not be merged with sequential loops.
  - A dependence is parallelism-inhibiting if it is carried by the fused loop
  - The carried dependence may be realigned via Loop alignment
- What if the loops to be fused have different lower and upper bounds?
  - Loop alignment, peeling, and index-set splitting

```
    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
```

The Typed Fusion Algorithm

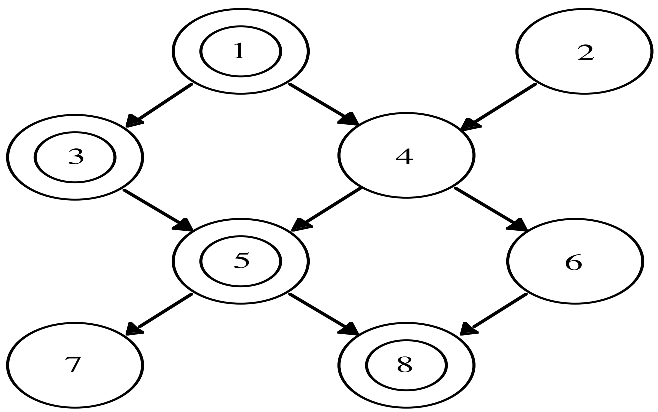
- Input: loop dependence graph (V,E)
- Output: a new graph where loops to be fused are merged into single nodes
- Algorithm
  - Classify loops into two types: parallel and sequential
  - Gather all dependences that inhibit fusion --- call them bad edges
  - Merge nodes of V subject to the following constraints
    - Bad Edge Constraint: nodes joined by a bad edge cannot be fused.
    - Ordering Constraint: nodes joined by path containing non-parallel vertex should not be fused

Typed Fusion Procedure

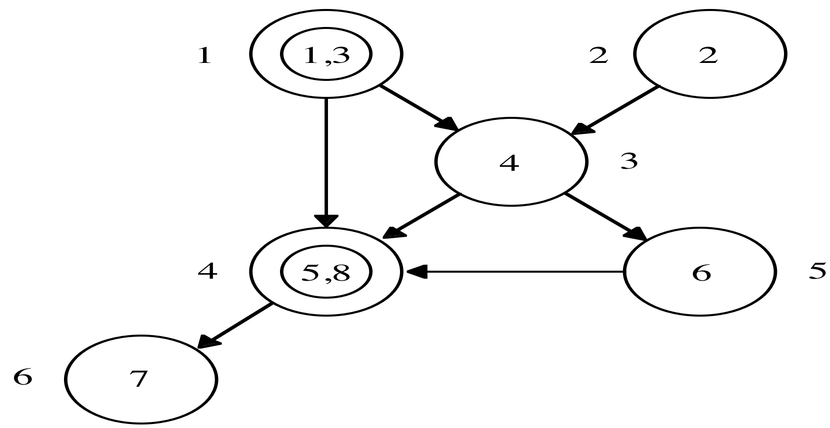
```
procedure TypedFusion(V,E,B,t0)
  for each node n in V
    num[n] = 0  //the group # of n
    maxBadPrev[n]=0 //the last group non-compatible with n
    next[n]=0 //the next group non-compatible with n
  W = {all nodes with in-degree zero};  fused = 0  // last fused node
  while W isn't empty
     remove node n from W;   Mark n as processed;
     if type[n] = t0
          if maxBadPrev[n] = 0 then p ← fused
          else p ← next[maxBadPrev[n]]
          if p != 0 then num[n] = num[p]
          else { if fused != 0 then {next[fused] = n} fused=n; num[n]=fused;}
    else { num[n]=newgroup(); maxBadPrev[n]=fused; }
    for each dependence d : n -> m in E:
        if (d is a bad edge in B) maxBadPrev[m] = max(maxBadPrev[m],num[n]);
        else maxBadPrev[m] = max(maxBadPrev[m],maxBadPrev[n]);
        if  all predecessors of m are processed: add m to W
```

Typed Fusion Example

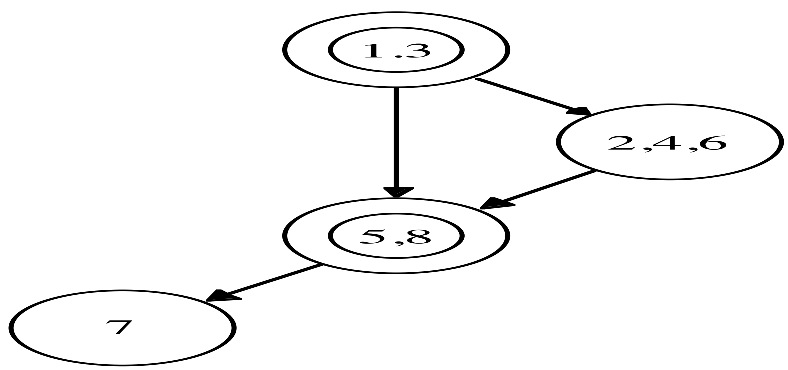
Original loop graph



After fusing parallel loops



After fusing sequential loops



So far…

- Single loop methods
  - Privatization
  - Alignment
  - Loop distribution
  - Loop Fusion
- Next we will cover
  - Loop interchange
  - Loop skewing
  - Loop reversal
  - Loop strip-mining
  - Pipelined parallelism

Loop Interchange

- Move parallel loops to outermost level
  - 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 contain only '=' entries
- Example

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

  - OK for vectorization
  - Problematic for coarse-grained parallelization
    - Need to move the J loop outside

Loop Selection

- Generate most parallelism with adequate granularity
  - Key is to select proper loops to run in parallel
  - Optimality is a NP-complete problem
- Informal parallel code generation strategy
  - Select parallel loops and move them to the outermost position
  - Select a sequential loop to move outside and enable internal parallelism
    - Look at dependences carried by single loops and move such loops outside

```
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
```

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

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
```

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

- Goal: allow a loop to be moved to the outermost
  - Safe only if all dependences have >= at the loop level

```
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
```

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
```

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

- Skewed using k=K+I+J:

```
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
```

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

Loop Skewing + Interchange

```
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
```

- Selection Heuristics
  - Parallelize outermost loop if possible
  - Make at most one outer loop sequential to enable inner parallelism
  - If both fails, try skewing
  - If skewing fails, try minimize the number of outside sequential loops

Loop Strip Mining

- Converts available parallelism into a form more suitable for the hardware

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

k = 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
```

Perfect Loop Nests

- Transformations to perfectly nested loops
  - Safety can be determined using the dependence matrix of the loop nest
  - Transformed dependence matrix can be obtained via a transformation matrix
  - Examples
    - loop interchange, skewing, reversal, strip-mining
    - Loop blocking is combination of loop interchange and strip-mining
- A transformation matrix T is unimodular if
  - T is square
  - All the elements of T are integral and
  - The absolute value of the determinant of T is 1
  - Example unimodular transformations
    - Loop interchange, loop skewing, loop reversal
- Composition of unimodular transformations is unimodular

Profitability-Based Methods

- Many alternatives for parallel code generation
  - Different hardware components require different optimizations
    - Fine-grained vs. coarse-grained parallelism, memory performance
  - Optimality is NP-complete
    - Exponential in the number of loops in a nest
    - Loop upper bounds are unknown at compile time
  - Use static performance estimation functions to select the better performing alternatives
    - May not be accurate
  - Key considerations
    - Cost of memory references
    - Sufficiency of parallelism granularity

Estimating Cost of Memory References

- Goal: assign each loop the cost of memory references when putting the loop innermost
  - At each iteration of the loop nest, compute
    - How many times the memory needs to be accessed?
- Assumptions
  - Data accessed in consecutive iterations are still in cache
  - Data accessed in different outer-loop iterations are not in cache
- Algorithm steps
  - Subdivide memory references in the loop body into reuse groups
    - All references in each group are connected by dependences
    - Input dependences need to be considered as well
  - Determine cost of subsequent accesses to the same reference
    - Loop invariant (carried only by innermost loop): Cost = 1
    - unit stride: Cost=number of iterations / cache line size
    - non-unit stride: Cost = number of iterations

Loop Selection Based on Memory Cost

- Assuming cache line size is L

```
DO I = 1, N
DO J = 1, N
    DO K = 1, N
        C(I, J) = C(I, J) + A(I, K) * B(K, J)
    ENDDO
ENDDO
ENDDO
```

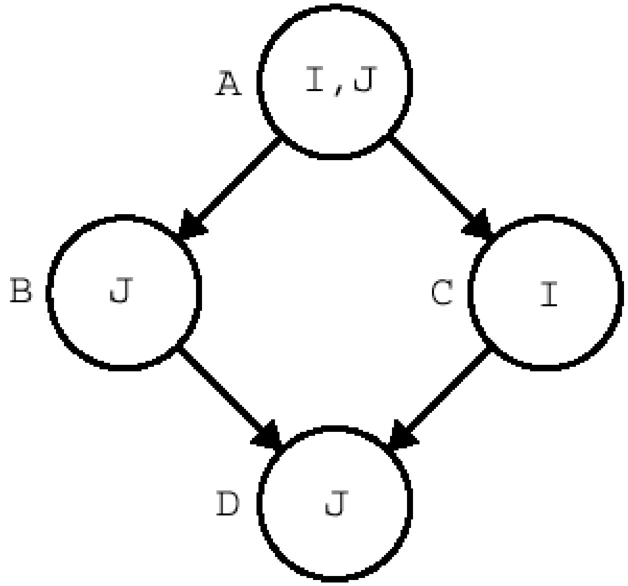
  - Innermost K loop = N*N*N*(1+1/L)+N*N
    - cost(C)=1 cost(A)=N cost(B)=N/L
  - Innermost J loop = 2*N*N*N+N*N
  - Innermost I loop = 2*N*N*N/L+N*N
- Reorder loop from innermost in the order of increasing cost
  - Limited by safety of loop interchange

Parallel Code Generation

```
procedure Parallelize(L, D)
  success = ParallelizeNest(L);
  if not success then begin
    if L can be distributed then begin
      distribute L into loop nests L1, L2, …, Ln;
      for I = 1,…n, do Parallelize(li, Di);
      TypedFusion({L1, L2, …, Ln});
   else
      for each loop L0 inside L do Parallelize(Lo,D0);
```

Multilevel Loop Fusion

- Commonly used for imperfect loop nests
  - Used after maximal loop distribution



- Decision making needs look-ahead
  - Heuristic: Fuse with a loop that cannot be fused with one of its successors

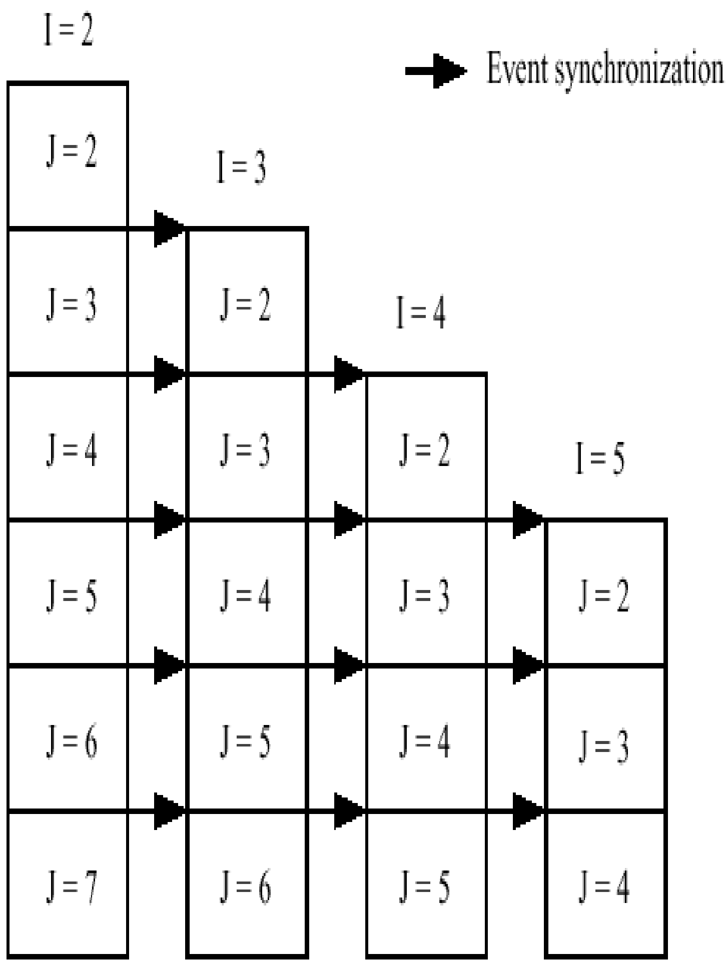
Pipelined Parallelism

- Useful where complete parallelization is not available
  - Higher synchronization costs
  - Fortran command DOACROSS

```
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
```

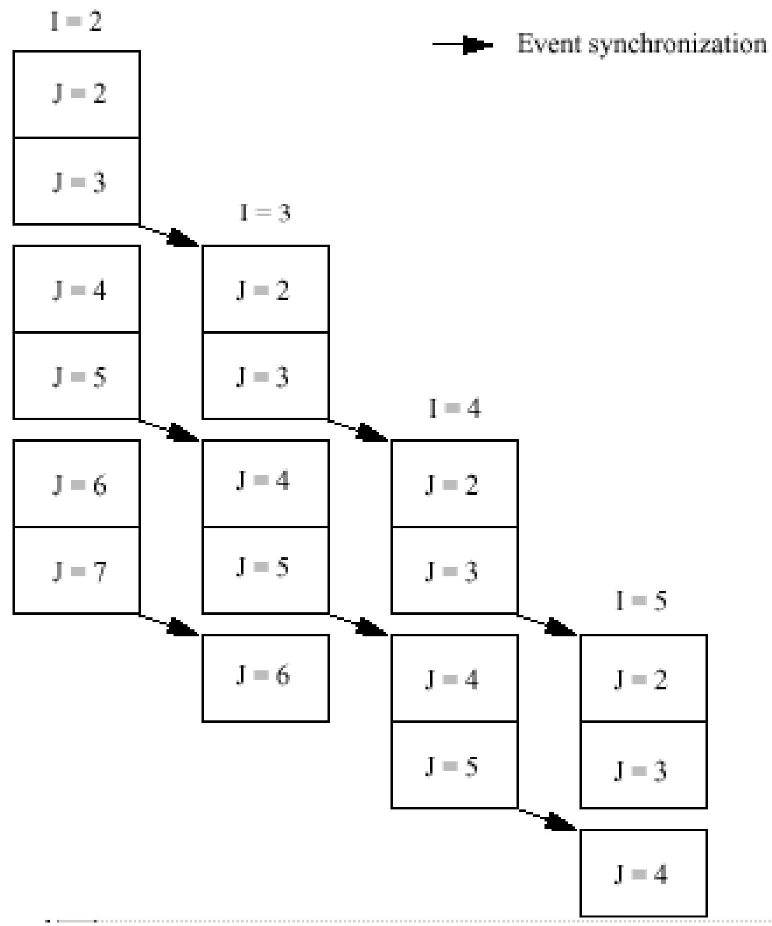
  - Pipelined Parallelism

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



Reducing Synchronization Cost

```
DOACROSS I = 2, N-1
    POST (E(1))
    K = 0
    DO J = 2, N-1, 2
        K = K+1
        WAIT(EV(K))
        DO j = J, MAX(J+1, N-1)
            A(I, J) = .25*(A(I-1,J) +
    A(I,J-1) + A(I+1,J) + A(I,J+1)
        ENDDO
        POST (EV(K+1))
    ENDDO
ENDDO
```



Scheduling Parallel Work

- Parallel execution is not beneficial if $\sigma_0 \geq (NB)/p$
  - Bakery-counter scheduling has high synchronization cost
- Guided Self-Scheduling
  - Minimize synchronization overhead
  - Schedules groups of iterations together
    - Go from large to small chunks of work
  - Keep all processors busy at all times
  - Iterations dispensed at time t follows: $x = \left\lceil \frac{N_t}{p} \right\rceil$
    - Alternatively we can have GSS(k) that guarantees that all blocks handed out are of size k or greater

Erlebacher

```
DO J = 1, JMAXD
  DO I = 1, IMAXD
    F(I, J, 1) = F(I, J, 1) * B(1)
```

```
DO K = 2, N-1
  DO J = 1, JMAXD
    DO I = 1, IMAXD
      F(I,J,K)=(F(I,J,K)–A(K)*F(I,J,K-1))*B(K)
```

```
DO J = 1, JMAXD
  DO I = 1, IMAXD
    TOT(I, J) = 0.0
```

```
DO J = 1, JMAXD
  DO I = 1, IMAXD
    TOT(I, J) = TOT(I, J) + D(1) * F(I, J, 1)
```

```
DO K = 2, N-1
  DO J = 1, JMAXD
    DO I = 1, IMAXD
      TOT(I, J) = TOT(I, J) + D(K) * F(I, J, K)
```

Loop Fusion+Parallelization

```
PARALLEL DO J= 1, JMAXD
    DO I = 1, IMAXD
        F(I, J, 1) = F(I, J, 1) * B(1)
    DO K = 2, N – 1
        DO I = 1, IMAXD
            F(I, J, K) = (F(I, J, K) – A(K) * F(I, J, K-1)) * B(K)
    DO I = 1, IMAXD
        TOT(I, J) = 0.0
    DO I = 1, IMAXD
        TOT(I, J) = TOT(I, J) + D(1) * F(I, J, 1)
    DO K = 2, N-1
        DO I = 1, IMAXD
            TOT(I, J) = TOT(I, J) + D(K) * F(I, J, K)
```

Multi-level Fusion

```
PARALLEL DO J = 1, JMAXD
   DO I = 1, IMAXD
      F(I, J, 1) = F(I, J, 1) * B(1)
      TOT(I, J) = 0.0
      TOT(I, J) = TOT(I, J) + D(1) * F(I, J, 1)
   ENDDO

   DO K = 2, N-1
      DO I = 1, IMAXD
         F(I, J, K) = ( F(I, J, K) – A(K) * F(I, J, K-1)) * B(K)
         TOT(I, J) = TOT(I, J) + D(K) * F(I, J, K)
      ENDDO
   ENDDO
ENDDO
```