

# **Parallelism Enabling Techniques**

**Marc González Tallada**

**Dept. d'Arquitectura de Computadors**

**Universitat Politècnica de Catalunya**

# Index

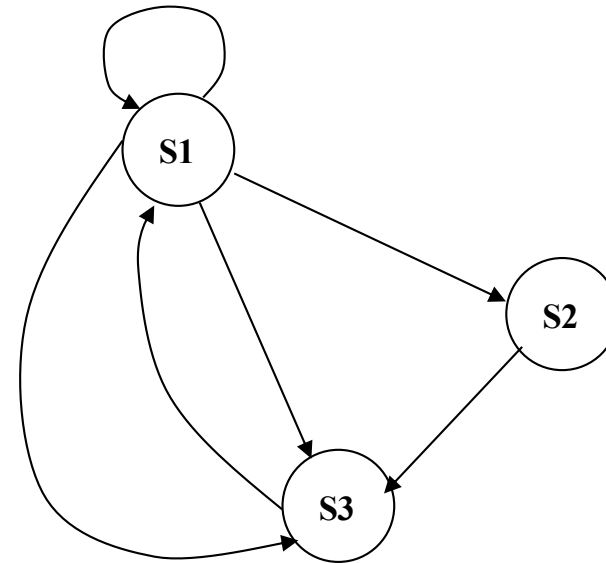
- **Scalar Expansion**
- **Privatization**
- **Induction Variable Substitution**
- **Reduction Parallelization**
- **Recurrence Substitution**
- **Loop Skewing**
- **Forward Substitution**
- **Loop Interchange**
- **Stripmining**
- **Loop Synchronization**

# Scalar Expansion

## ■ Get one scalar instance per iteration

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

```
DO I = 1, N  
S1 VT_(I) = A(I)  
S2 A(I) = B(I)  
S3 B(I) = VT_(I)  
END DO
```

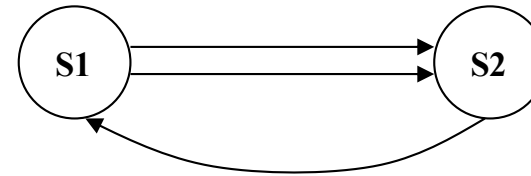


```
S1 VT_(1:N) = A(1:N)  
S2 A(1:N) = B(1:N)  
S3 B(1:N) = VT_(1:N)  
T = VT_(N)
```

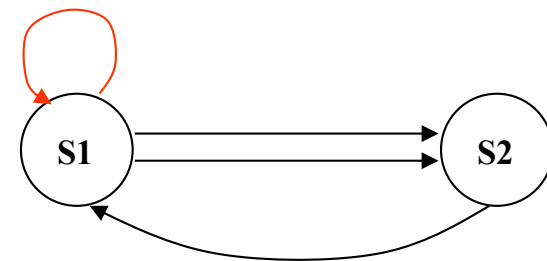
# Scalar Expansion

## ■ But not always is a good solution

```
DO I = 1, N
S1  T = T + A(I) + A(I+1)
S2  A(I) = T
END DO
```



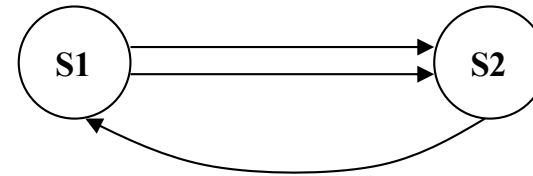
```
VT_(0) = T
DO I = 1, N
S1  VT_(I) = VT_(I-1) + A(I) + A(I+1)
S2  A(I) = VT_(I)
END DO
T = VT_(N)
```



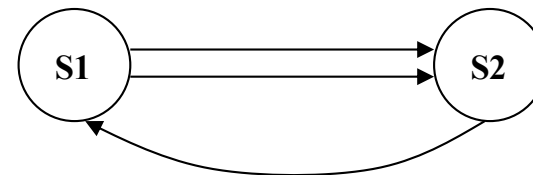
# Privatization

## ■ Another way of doing the same

```
for (i=0; i<N; i++) {  
s1 t = t + A(i) + A(i+1)  
s2 A(i) = t  
}
```



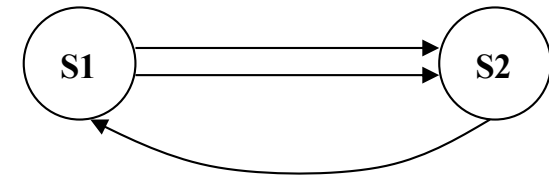
```
#pragma omp parallel do firstprivate(t) shared (A)  
for (i=0; i<N; i++) {  
s1 t = t + A(i) + A(i+1)  
s2 A(i) = t  
}
```



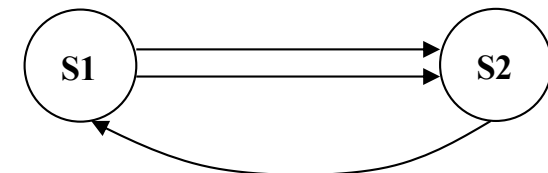
# Array Privatization

## ■ Similar with array

```
DO j=1,n  
s1    t(1:m) = A(j,1:m)+B(j)  
s2    C(j,1:m) = t(1:m) + t(1:m)**2  
ENDDO
```



```
!$OMP PARALLEL DO PRIVATE(t) SHARED(A,B,C)  
DO j=1,n  
s1    t(1:m) = A(j,1:m)+B(j)  
s2    C(j,1:m) = t(1:m) + t(1:m)**2  
ENDDO
```



# Array Privatization

## ■ Capabilities

- array *Def-Use* Analysis
- combining and intersecting subscript ranges
- representing conditionals under which sections are defined/used
- if ranges too complex to represent
  - ✓ overestimate *Uses*, underestimate *Defs*

## ■ Array privatization algorithm:

- For each loop nest:
  - ✓ iterate from innermost to outermost loop:
    - for each statement in the loop
      - find definitions; add them to the existing definitions in this loop.
      - find array uses; if they are covered by a definition, mark this array section as privatizable for this loop, otherwise mark it as upward-exposed in this loop;
    - aggregate defined and upward-exposed, used ranges (expand from range per-iteration to entire iteration space); record them as *Defs* and *Uses* for this loop

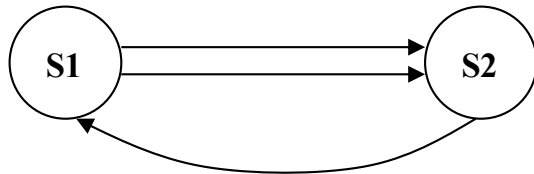
```
k = 5
DO j=1,n
  t(1:10) = A(j,1:10)+B(j)
  C(j,iv) = t(k)
  t(11:m) = A(j,11:m)+B(j)
  C(j,1:m) = t(1:m)
END DO
```

```
DO j=1,n
  IF (cond(j))
    t(1:m) = A(j,1:m)+B(j)
    C(j,1:m) = t(1:m) + t(1:m)**2
  ENDIF
  D(j,1) = t(1)
END DO
```

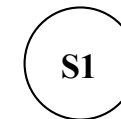
# Induction Variable Substitution

## ■ Simple case

```
ind = k
DO i=1,n
s1 ind = ind + 2
s2 A(ind) = B(i)
ENDDO
```

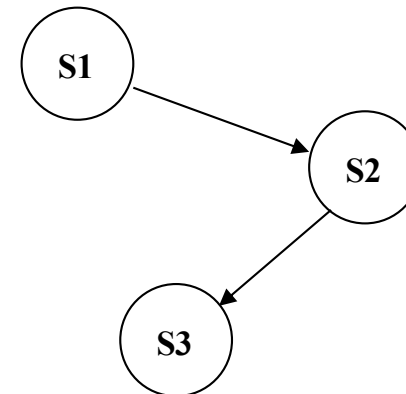


```
DO i=1,n
s1 A(k+2*i) = B(i)
ENDDO
```



## ■ Generalized Induction Variables

```
DO i=1,n
s1 ind1 = ind1 + 1
s2 ind2 = ind2 + ind1
s3 A(ind2) = B(i)
ENDDO
```





# Induction Variable Substitution

## ■ Recognizing Generalized Induction Variables

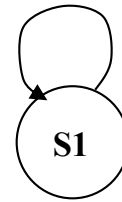
- Pattern Matching:

- ✓ find induction statements in a loop nest of the form  $iv = iv + expr$  or  $iv = iv * expr$ , where  $iv$  is a scalar integer.
- ✓  $expr$  must be loop-invariant or another induction variable (there must not be cyclic relationships among IVs)
- ✓  $iv$  must not be assigned in a non-induction statement
- ✓ Abstract interpretation: find symbolic increments of  $iv$  per loop iteration
- ✓ SSA-based recognition

# Reduction Parallelization

## ■ Simple case

```
DO i=1,n
S1 sum = sum + A(i)
ENDDO
```



```
!$OMP PARALLEL PRIVATE(s)
  s=0
!$OMP DO
  DO i=1,n
    s=s+A(i)
  ENDDO
!$OMP ATOMIC
  sum = sum+s
!$OMP END PARALLEL
```

```
!$OMP PARALLEL DO
!$OMP+REDUCTION(+:sum)
  DO i=1,n
    sum = sum + A(i)
  ENDDO
```

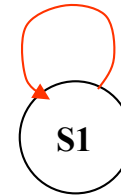
Note, OpenMP has a reduction clause,  
only reduction recognition is needed

# Reduction Parallelization

## ■ Array treatment

- Subscript expression has to point out the same array element

```
DO i=1,n
S1 sum(expr) = sum(expr) + A(i)
ENDDO
```



```
!$OMP PARALLEL PRIVATE(s)
  s(1:m)=0
!$OMP DO
  DO i=1,n
    s(expr)=s(expr)+A(i)
  ENDDO
!$OMP CRITICAL
  sum(1:m) = sum(1:m)+s(1:m)
!$OMP END CRITICAL
!$OMP END PARALLEL
```

# Reduction Parallelization

## ■ Recognition

- Pattern Matching:

- ✓ find reduction statements in a loop of the form  $X = X \# \text{expr}$ , where  $X$  is either scalar or an array expression ( $a[\text{sub}]$ , where  $\text{sub}$  must be the same on the LHS and the RHS),  $\#$  is a reduction operation, such as  $+$ ,  $*$ ,  $\min$ ,  $\max$ , and other
  - associative operation, neuter element
- ✓  $X$  must not be used in any non-reduction statement in this loop (however, there may be multiple reduction statements for  $X$ )

# Reduction Parallelization

## ■ Performance considerations

- Parallelized reductions execute substantially more code than their serial versions
  - ✓ overhead if the reduction is small.
- In many cases (for large reductions) initialization and sum-up are insignificant.
- False sharing can occur, especially in expanded reductions, if multiple processors use adjacent array elements of the temporary reduction array (s).
- Expanded reductions exhibit more parallelism in the sum-up operation.
- Potential overhead in initialization, sum-up, and memory used for large, sparse array reductions
  - ✓ compression schemes can become useful.

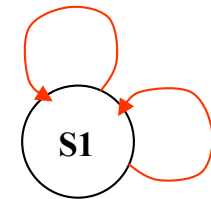
# Recurrence Substitution

- Sometimes, dependences can be summarized under the form of a recurrence equation

- Example

```
DO j=1,n  
s1 a(j) = c0+c1*a(j)+c2*a(j-1)+c3*a(j-2)  
ENDDO
```

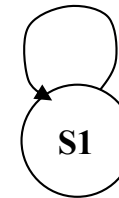
call rec\_solver(a,n,c0,c1,c2,c3)



# Recurrence Substitution

## ■ Basic idea of the recurrence solver

```
DO j=1, 40
  a(j) = a(j) + a(j-1)
ENDDO
```



DO j=1, 10	DO j=11, 20	DO j=21, 30	DO j=31, 40
a(j)=a(j)+a(j-1)	a(j)=a(j)+a(j-1)	a(j)=a(j)+a(j-1)	a(j)=a(j)+a(j-1)
ENDDO	ENDDO	ENDDO	ENDDO
0	$\Delta a(10)$	$\Delta a(10) + \Delta a(20)$	$\Delta a(10) + \Delta a(20) + \Delta a(30)$

### ● Issues:

- ✓ Solver makes several parallel sweeps through the iteration space (n). Overhead can only be amortized if n is large.
- ✓ Many variants of the source code are possible. Transformations may be necessary to fit the library call format (additional overhead).

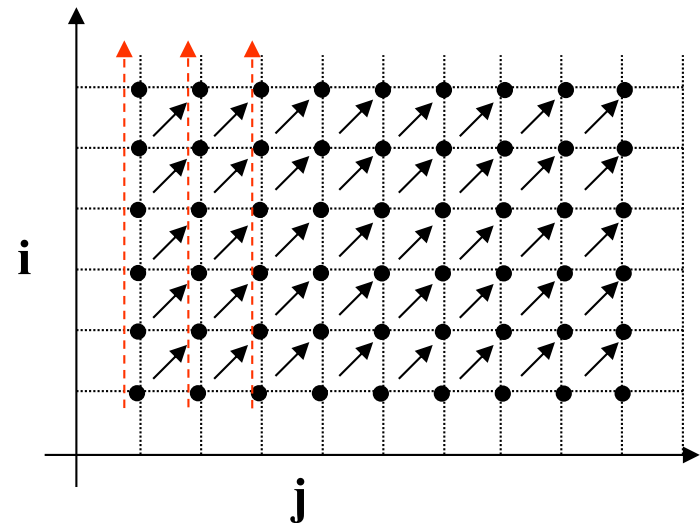
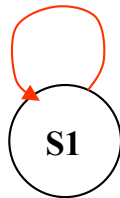


# Loop Skewing

## ■ Dependences in iteration space

- Although, that, some parallelism exists
- Relation between dependences and the way the iteration space is traversed
- Example:

```
DO j=1,NJ
  DO i=1,NI
    A(i,j) = A(i,j) + A(i-1,j-1)
  ENDDO
ENDDO
```



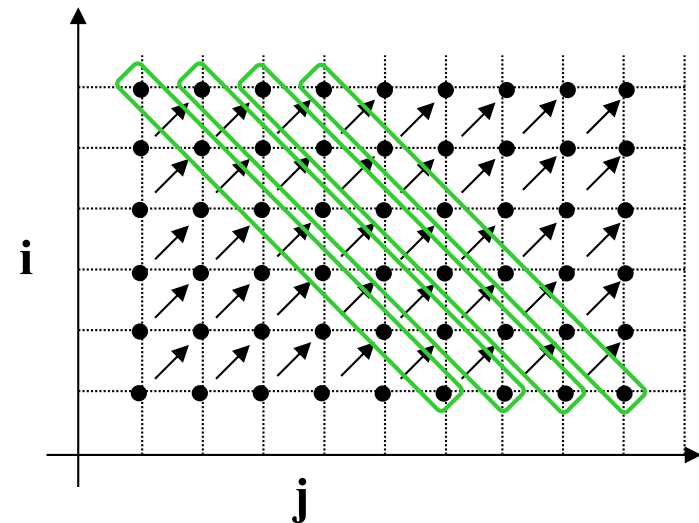
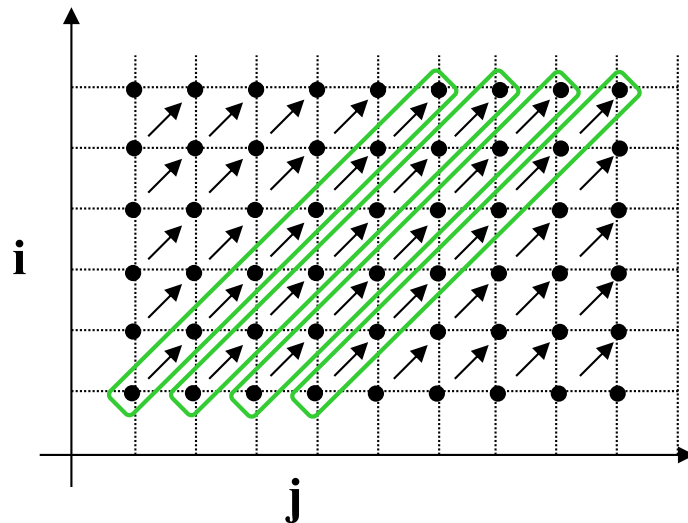
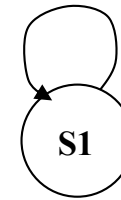


# Loop Skewing

## ■ Iteration space graph:

- Shared regions show wave-fronts of iterations in the iteration space that can be executed in parallel.

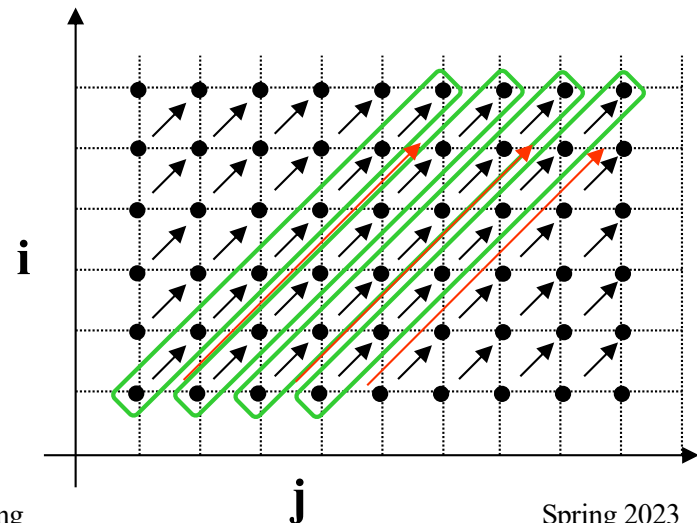
```
DO j=1,NJ
  DO i=1,NI
    A(i,j) = A(i,j) + A(i-1,j-1)
  ENDDO
ENDDO
```



# Loop Skewing

## ■ Change the way iteration space is traversed

```
DO wave=1,NI+NJ-1
  i = max(NI-wave,1)
  j = max(-NJ+wave,1)
  wsize = min(NI,NJ-1-abs(wave-NJ-1))
  DO k=0,wsize-1
    A(i+k,j+k) = A(i+k,j+k) + A(i+k-1,j+k-1)
  ENDDO
ENDDO
```



# Forward Substitution

## ■ Value propagation

- Requires def-use edges, and Data Flow analysis

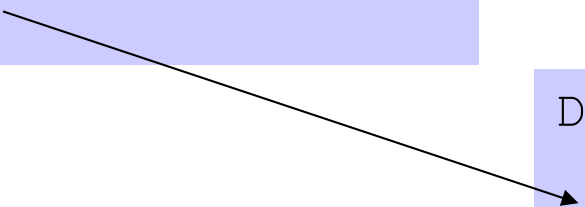
```
m = n+1
...
DO j=1,n
  a(j) = a(j+m)
ENDDO
```

```
m = n+1
...
DO j=1,n
  a(j) = a(j+n+1)
ENDDO
```

# Loop Interchange

- **Loop interchanging alters the data reference order significantly**
  - Affects locality of references
  - Data dependences determine the legality of the transformation
- **Loop interchanging may also impact the granularity of the parallel computation (inner loop may become parallel instead of outer)**

```
DO i=1, n
  DO j=1, m
    a(i,j) =a(i,j)+a(i-1,j)
  ENDDO
ENDDO
```



```
DO j= 1,m
  DO i=1,n
    a(i,j) =a(i,j)+a(i-1,j)
  ENDDO
ENDDO
```

# Stripmining (also called blocking)

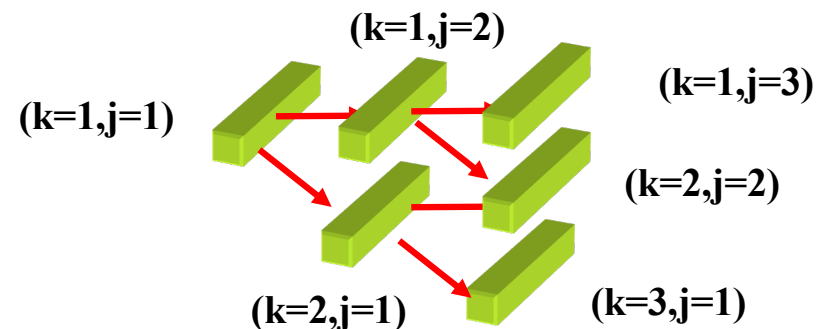
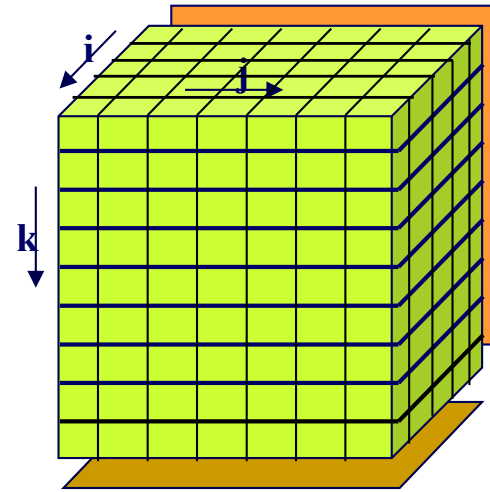
## ■ Data dependences

```
do k = 1, nz
do j = 1, ny
do i = 1, nx
    A(i,j,k)=A(i-1,j,k)+A(i,j-1,k)+A(i,j,k-1)
end do
end do
end do
```

```
!$omp parallel do
```

```
do k = 1, nz
```

```
...
```



# Loop Synchronization

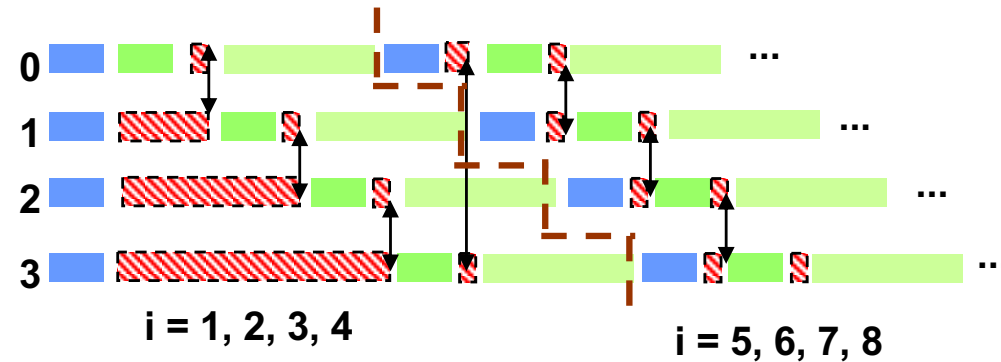
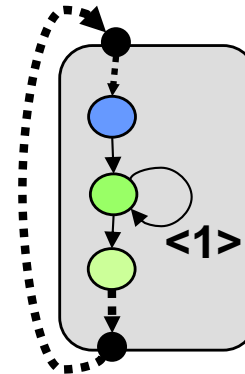
## ■ Single-level loop

```
!$omp parallel do  
!$omp& schedule (static,1)  
do i=1,10
```

```
  [blue bar]  
  CALL wait(i-1)
```

```
  [green bar]  
  CALL sync (i+1)
```

```
  [light green bar]  
enddo
```



# END



# Scalar and Array Privatization

## ■ References

- Peng Tu and D. Padua. Automatic Array Privatization. Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science 768, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.), Springer-Verlag, 1994.
- Zhiyuan Li, Array Privatization for Parallel Execution of Loops, Proceedings of the 1992 ACM International Conference on Supercomputing



# Induction Variable Substitution

## ■ References

- B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. ACM Int. Conf. on Supercomputing (ICS'95), June 1995. (Extended version: Parallelization in the presence of generalized induction and reduction variables. [www.ece.ecn.purdue.edu/~eigenman/reports/1396.pdf](http://www.ece.ecn.purdue.edu/~eigenman/reports/1396.pdf))
- Mohammad R. Haghighat , Constantine D. Polychronopoulos, Symbolic analysis for parallelizing compilers, ACM Transactions on Programming Languages and Systems (TOPLAS), v.18 n.4, p.477- 518, July 1996
- Michael P. Gerlek , Eric Stoltz , Michael Wolfe, Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form, ACM Transactions on Programming Languages and Systems (TOPLAS), v.17 n.1, p.85-122, Jan. 1995