EE663: Optimizing Compilers

Prof. R. Eigenmann

Purdue University

School of Electrical and Computer Engineering

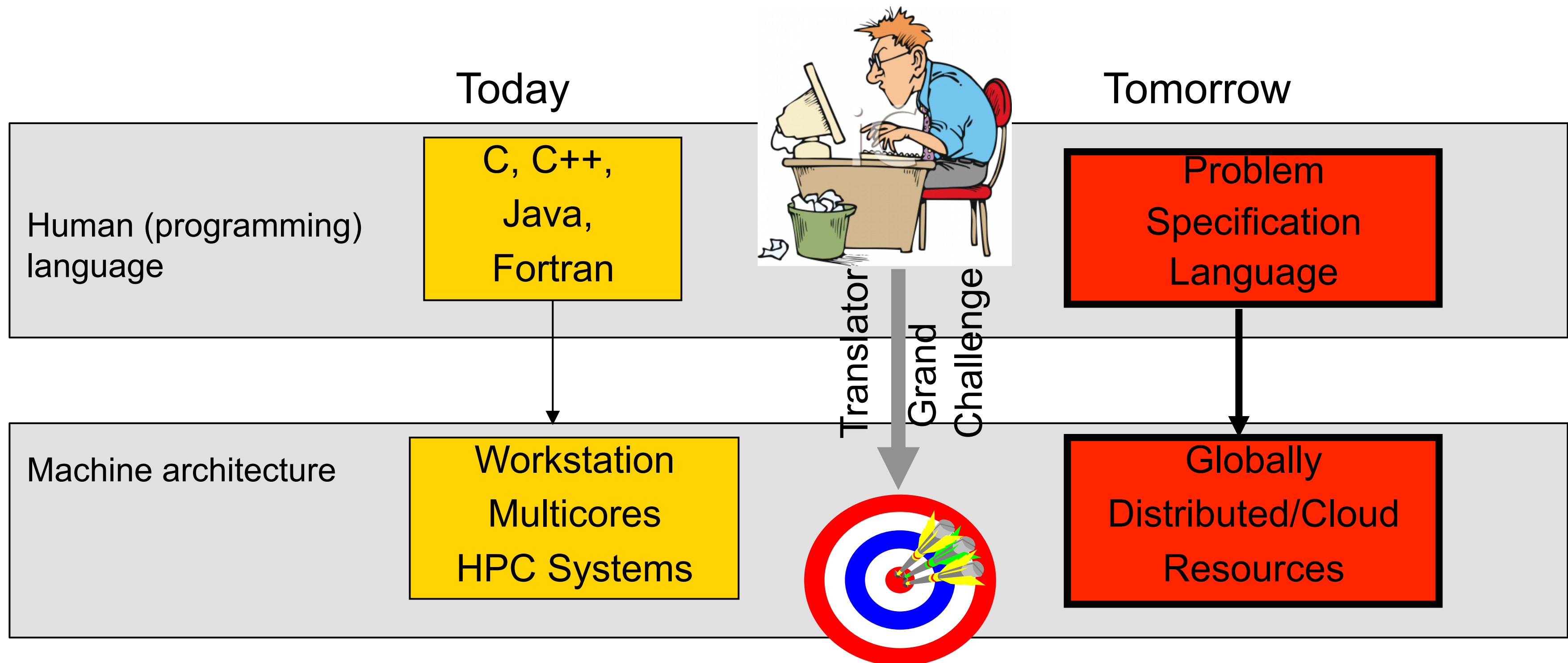
Spring 2012

https://engineering.purdue.edu/~eigenman/ECE663/

I. Motivation and Introduction:

Optimizing Compilers are in the Center of the (Software) Universe

They translate increasingly advanced human interfaces (programming languages) onto increasingly complex target machines



Processors have multiple cores. Parallelization is a key optimization.

Issues in Optimizing / Parallelizing Compilers

The Goal:

- We would like to run standard (C, C++, Java, Fortran) programs on common parallel computers

leads to the following high-level issues:

- How to detect parallelism?
- How to map parallelism onto the machine?
- How to create a good compiler architecture?

Detecting Parallelism

- Program analysis techniques
- Data dependence analysis
- Dependence removing techniques
- Parallelization in the presence of dependences
- Runtime dependence detection

Mapping Parallelism onto the Machine

- Exploiting parallelism at many levels
  - Multiprocessors and multi-cores (our focus)
  - Distributed computers (clusters or global networks)
  - Heterogeneous architectures
  - Instruction-level parallelism
  - Vector machines
- Exploiting memory organizations
  - Data placement
  - Locality enhancement
  - Data communication

Architecting a Compiler

- Compiler generator languages and tools
- Internal representations
- Implementing analysis and transformation techniques
- Orchestrating compiler techniques (when to apply which technique)
- Benchmarking and performance evaluation

Parallelizing Compiler Books and Survey Papers

Books:

- Michael Wolfe: High-Performance Compilers for Parallel Computing (1996)
- Utpal Banerjee: several books on Data Dependence Analysis and Transformations
- Ken Kennedy, John Allen: Optimizing Compilers for Modern Architectures: A Dependence-based Approach (2001)
- Zima, H. and Chapman, B., Supercompilers for parallel and vector computers (1990)
- Scheduling and automatic Parallelization, Darte, A., Robert Y., and Vivien, F., (2000)

Survey Papers:

- Rudolf Eigenmann and Jay Hoeflinger, *Parallelizing and Vectorizing Compilers*, Wiley Encyclopedia of Electrical Engineering, John Wiley &Sons, Inc., 2001
- Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David Padua. *Automatic Program Parallelization. Proceedings of the IEEE, 81(2), February 1993.*
- David F. Bacon, Susan L. Graham, *Compiler transformations for high-performance computing, ACM Computing Surveys (CSUR), Volume 26, Issue 4, December 1994, Pages: 345 - 420,1994*

Course Approach

There are many schools on optimizing compilers. Our approach is *performance-driven.*

We will discuss:

- Performance of parallelization techniques
- Analysis and Transformation techniques in the Cetus compiler (for multiprocessors/cores)
- Additional transformations (for GPGPUs and other architectures)
- Compiler infrastructure considerations

The Heart of Automatic Parallelization

Data Dependence Testing

If a loop does not have data dependences between any two iterations then it can be safely executed in parallel

In science/engineering applications, loop parallelism is most important. In non-numerical programs other control structures are also important

Data Dependence Tests:
Motivating Examples

Loop Parallelization

Can the iterations of this loop be run concurrently?

```
DO i=1,100,2
  B(2*i) = ...
    ...    = B(2*i) +B(3*i)
ENDDO
```

DD testing is needed to detect parallelism

Statement Reordering

can these two statements be swapped?

```
DO i=1,100,2
  B(2*i) = ...
    ...    = B(3*i)
ENDDO
```

DD testing is important not just for detecting parallelism

A data dependence exists between two adjacent data references iff:

- both references access the same storage location and
- at least one of them is a write access

Data Dependence Tests: Concepts

Terms for data dependences between statements of loop iterations.

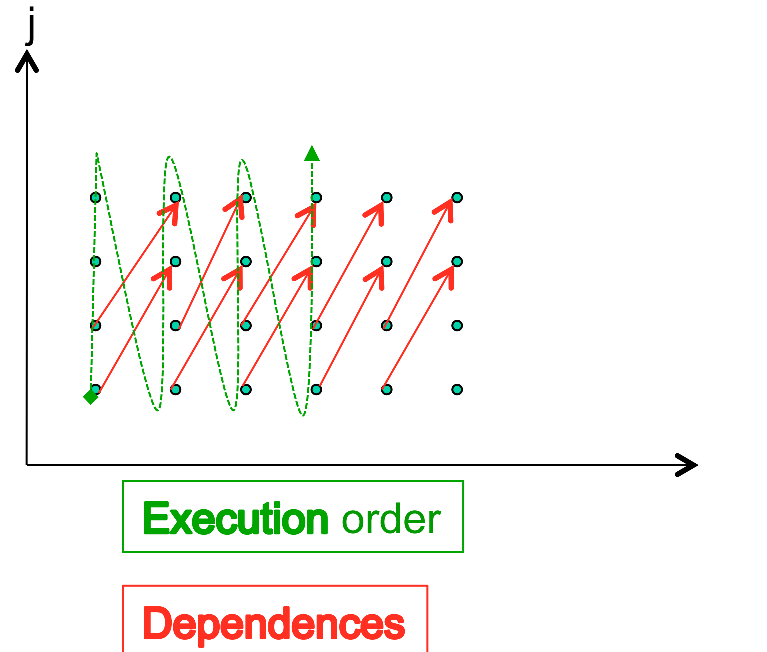
- Distance (vector): indicates how many iterations apart are source and sink of dependence.
- Direction (vector): is basically the sign of the distance. There are different notations: (<,=,>) or (-1,0,+1) meaning dependence (from earlier to later, within the same, from later to earlier) iteration.
- Loop-carried (or cross-iteration) dependence and non-loop-carried (or loop-independent) dependence: indicates whether or not a dependence exists within one iteration or across iterations.
  - For detecting parallel loops, only cross-iteration dependences matter.
  - *equal* dependences are relevant for optimizations such as statement reordering and loop distribution.

Data Dependence Tests: Concepts

- Iteration space graphs: the un-abstracted form of a dependence graph with one node per statement instance.

Example:

```
DO i=1,n
   DO j=1,m
      a(i,j) = a(i-1,j-2)+b(I,j)
   ENDDO
ENDDO
```



Data Dependence Tests:
Formulation of the Data-dependence problem

```
DO i=1,n
   a(4*i) = . . .
   . . .   =  a(2*i+1)
ENDDO
```

the question to answer:
can $4*i_1$ ever be equal to $2*i_2+1$ within $i_1, i_2 \in [1,n]$?

Note that the iterations at which the two expressions are equal may differ. To express this fact, we choose the notation i_1, i_2.

Let us generalize a bit: given

- two subscript functions *f* and *g*, and
- loop bounds *lower*, *upper*,

Does

$f(i_1) = g(i_2)$ have a solution such that
$lower \le i_1, i_2 \le upper$?

This course would now be finished if:

- the mathematical formulation of the data dependence problem had an accurate and fast solution, and
- there were enough loops in programs without data dependences, and
- dependence-free code could be executed by today's parallel machines directly and efficiently.
- engineering these techniques into a production compiler were straightforward.

There are enough hard problems to fill several courses!

II. Performance of Basic Automatic Program Parallelization

Two Decades of Parallelizing Compilers

A Performance study at the beginning of the 90es (Blume study)
Analyzed the performance of state-of-the-art parallelizers and vectorizers using the Perfect Benchmarks.

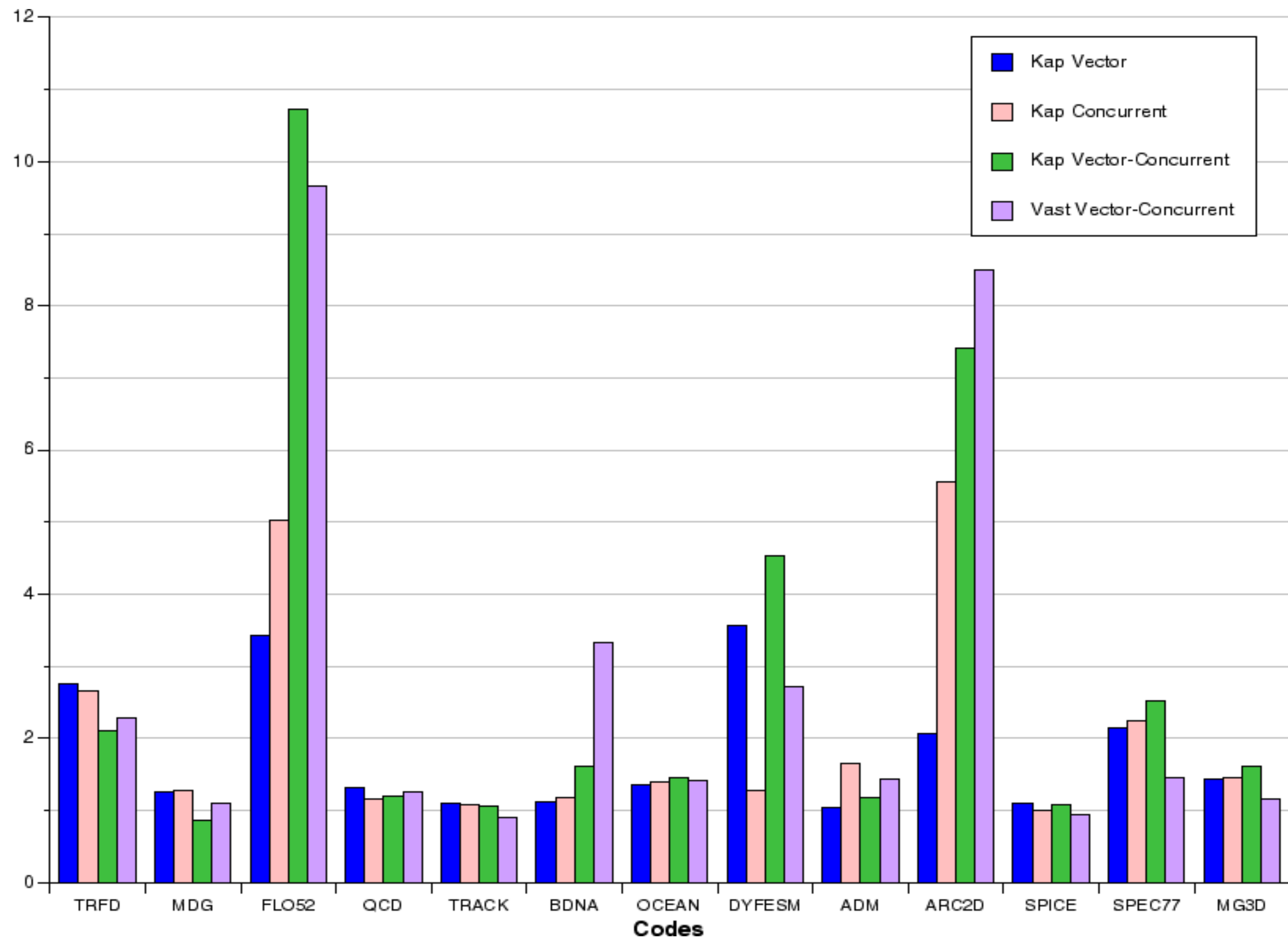
William Blume and Rudolf Eigenmann, **Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs**, *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992, pages 643--656.

Good reasons for starting two decades back:

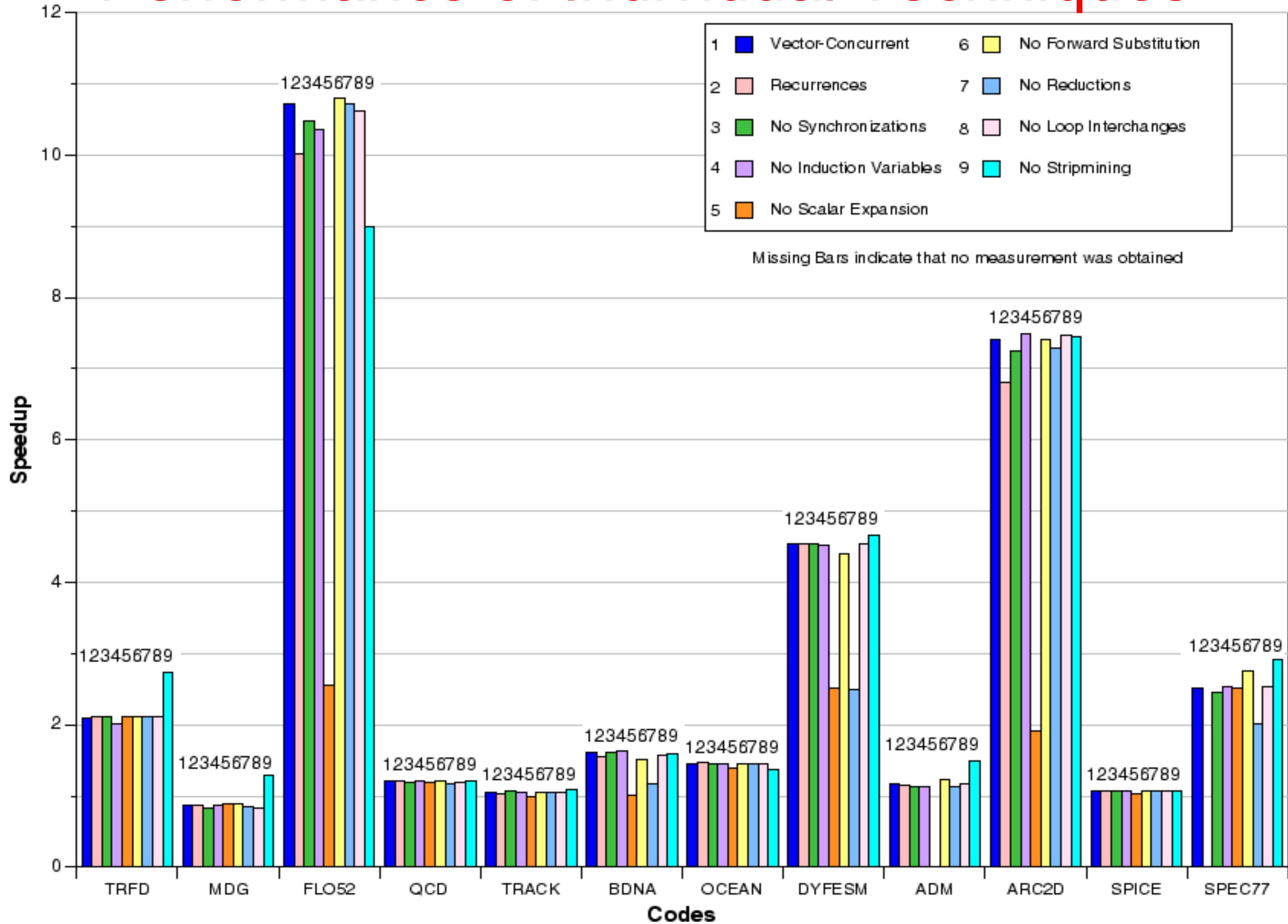
- We will learn simple techniques first.
- We will see how parallelization techniques have evolved
- We will see that extensions of the important techniques back then are still the important techniques today.

Overall Performance
of parallelizers in 1990

Speedup on 8 processors with 4-stage vector units



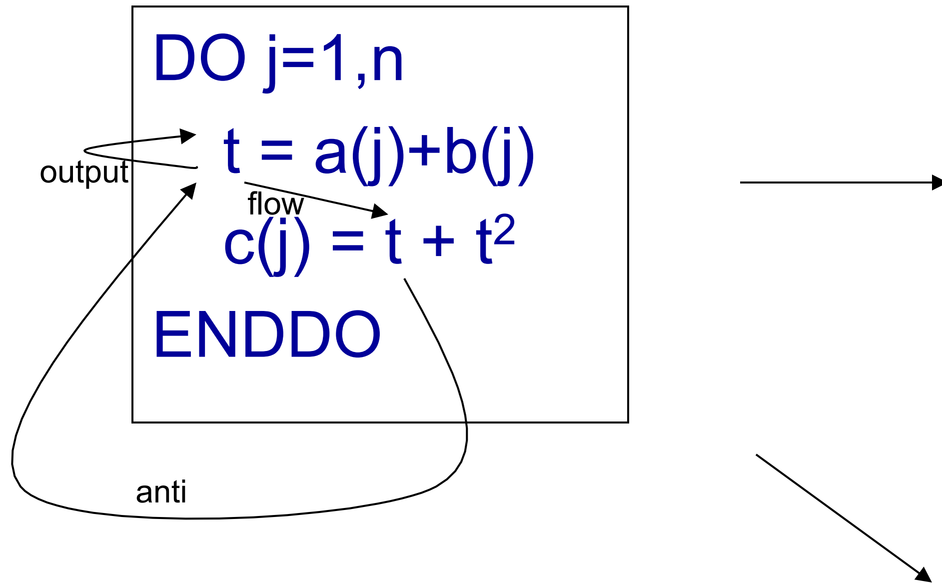
Performance of Individual Techniques



Transformations measured in the “Blume Study”

- Scalar expansion
- Reduction parallelization
- Induction variable substitution
- Loop interchange
- Forward Substitution
- Stripmining
- Loop synchronization
- Recurrence substitution

Scalar Expansion



Privatization

```
DO PARALLEL j=1,n
PRIVATE t
   t = a(j)+b(j)
   c(j) = t + t²
ENDDO
```

Expansion

```
DO PARALLEL j=1,n
   t0(j) = a(j)+b(j)
   c(j) = t0(j) + t0(j)²
ENDDO
```

We assume a shared-memory model:

- by default, data is shared, i.e., all processors can see and modify it
- processors share the work of parallel loops

Parallel Loop Syntax and Semantics in OpenMP

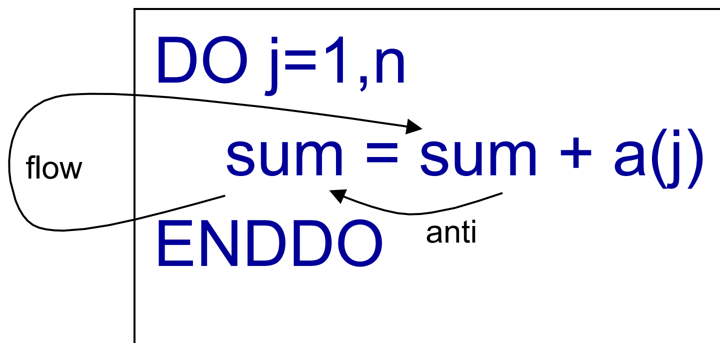
```
#pragma omp parallel for

for (i=lb; i<=ub; i++) {

  <loop body code>

}
```

```
!$OMP PARALLEL PRIVATE(<private data>)
   <preamble code>
!$OMP DO
DO i = lb, ub

  <loop body code>

ENDDO
!$OMP END DO
   <postamble code>
!$OMP END PARALLEL
```

work (iterations) shared by participating processors (threads)

Same code executed by all participating processors (threads)

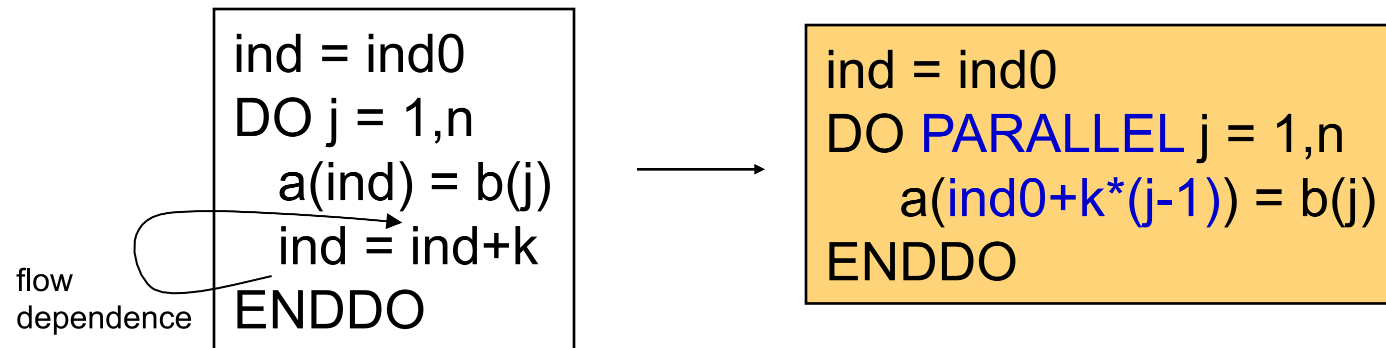
Reduction Parallelization



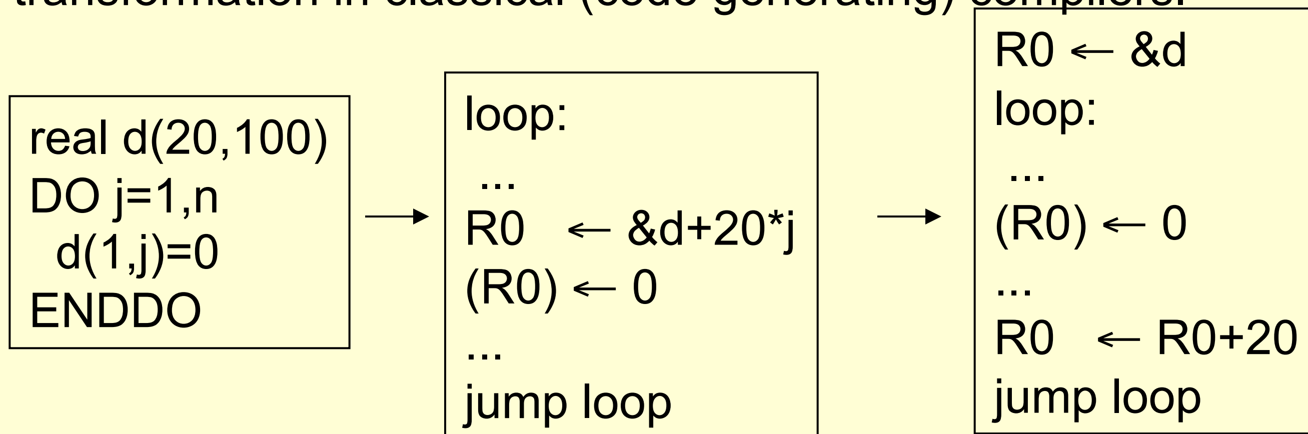

```
!$OMP PARALLEL, PRIVATE (s)
s = 0
!$OMP DO
DO j=1,n
  s = s + a(j)
ENDDO
!$OMP ENDDO
!$OMP ATOMIC
      sum=sum+s
!$OMP END PARALLEL
```

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

Induction Variable Substitution

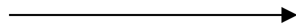


Note, this is the reverse of *strength reduction*, an important transformation in classical (code generating) compilers.

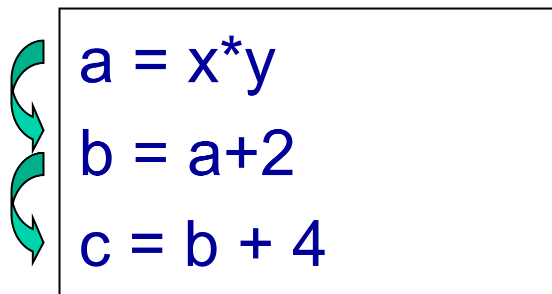


Forward Substitution

```
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
```

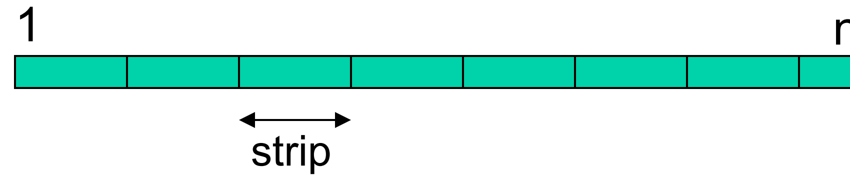


dependences

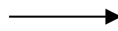
```
a = x*y
b = x*y+2
c = x*y + 6
```

no dependences

Stripmining




```
DO j=1,n
   a(j) = b(j)
ENDDO
```

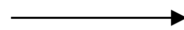


```
DO i=1,n,strip
   DO j=i,min(i+strip-1,n)
      a(j) = b(j)
   ENDDO
ENDDO
```

There are many variants of stripmining
(sometimes called *loop blocking*)

Loop Synchronization

```
DO j=1,n
    a(j) = b(j)
    c(j) = a(j)+a(j-1)
ENDDO
```



```
DOACROSS j=1,n
    a(j) = b(j)
    post(current_iteration)
    wait(current_iteration-1)
    c(j) = a(j)+a(j-1)
ENDDO
```

Recurrence Substitution

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

↓

```
call rec_solver(a(1),n,c0,c1,c2,c3)
```

Basic idea of the recurrence solver:

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

```
DO j=1,10
  a(j) = a(j) + a(j-1)
ENDDO
```

```
DO j=11,20
  a(j) = a(j) + a(j-1)
ENDDO
```

```
DO j=21,30
  a(j) = a(j) + a(j-1)
ENDDO
```

```
DO j=31,40
  a(j) = a(j) + a(j-1)
ENDDO
```

Error: 0 | $\Delta a(10)$ | $\Delta a(10)+\Delta a(20)$ | $\Delta a(10)+\Delta a(20)+\Delta a(30)$

Loop Interchange

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

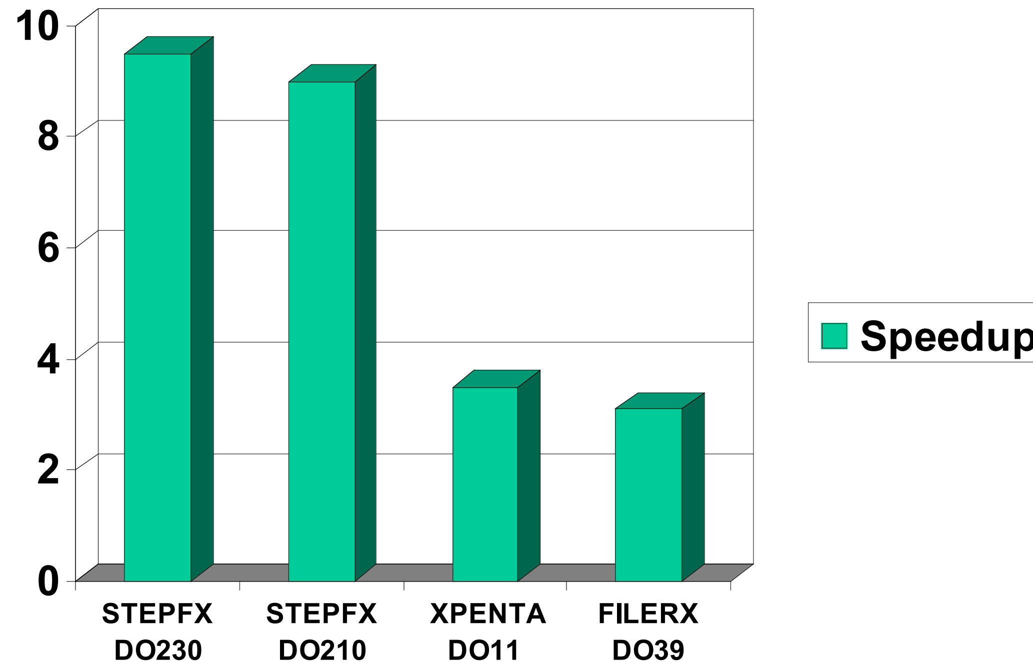
```
DO j= 1,m
  DO i=1,n
    a(i,j) =a(i,j)+a(i,j-1)
  ENDDO
ENDDO
```

- stride-1 references increase cache locality
  - read: increase spatial locality
  - write: avoid false sharing
- scheduling of outer loop is important (consider original loop nest):
  - cyclic: no locality w.r.t. to i loop
  - block schedule: there *may* be some locality
  - dynamic scheduling: chunk scheduling desirable
- cache organization is important
- parallelism at outer position reduces loop fork/join overhead

Effect of Loop Interchange

Example: speedups of the most time-consuming loops in the ARC2D benchmark on 4-core machine

loop interchange applied in the process of parallelization



Execution Scheme for Parallel Loops

1. Architecture supports parallel loops. Example: Alliant FX/8 (1980es)
   - machine instruction for parallel loop
   - HW concurrency bus supports loop scheduling

```
a=0
! DO PARALLEL
DO i=1,n
  b(i) = 2
ENDDO
b=3
```

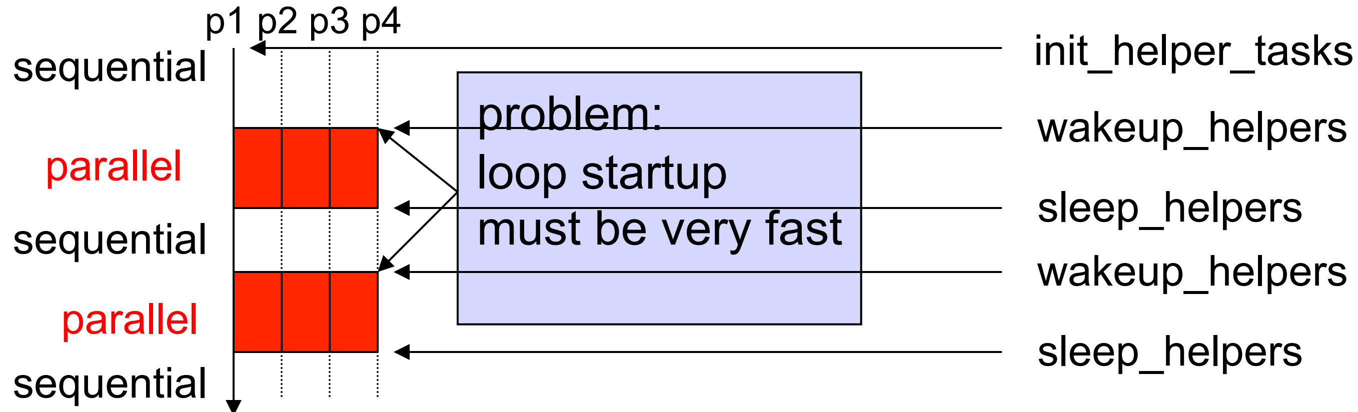
→

```
store #0,<a>
load <n>,D6
sub 1,D6
load &b,A1
cdoall D6
  store #2,A1(D7.r)
endcdoall
store #3,<b>
```

D7 is reserved for the loop variable. Starts at 0.

Execution Scheme for Parallel Loops

2. Microtasking scheme (dates back to early IBM mainframes)



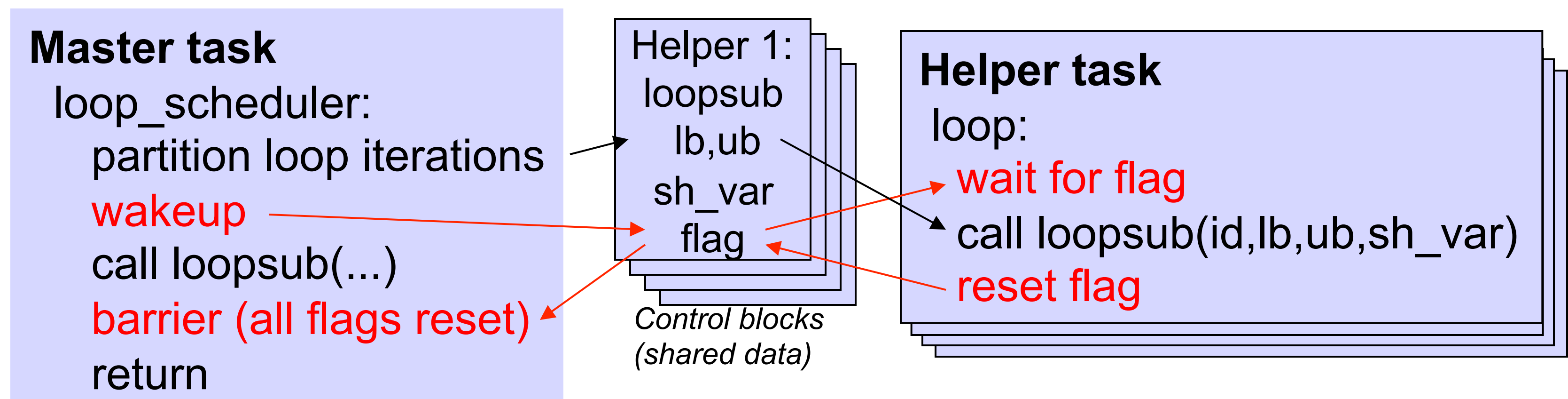
microtask startup: 1 μs
pthreads startup: up to 100 μs

Compiler Transformation and Runtime Function for the Microtasking Scheme

```
a=0
! DO PARALLEL
DO i=1,n
   b(i) = 2
ENDDO
b=3
```

```
call init_microtasking() // once at program start
...
a=0
call loop_scheduler(loopsub,i,1,n,b)
b=3
```

```
subroutine loopsub(mytask,lb,ub,b)
DO i=lb,ub
   b(i) = 2
ENDDO
END
```



III. Performance of Advanced Parallelization

Manual Improvements of the Perfect Benchmarks (1995)

Same information as on Slide 17

program	serial execution time (seconds)	Performance improvement factor			
		Automatically compiled		Manually improved	
		FX/8 (VAST)	Cedar (KAP)	FX/8	Cedar
ARC2D	2943	8.7	13.5	10.6	20.8
FLO52	906	9.0	5.5	14.6	15.3
BDNA	969	1.9	1.8	5.6	8.5
DYFESM	663	3.9	2.2	10.3	11.4
ADM	796	1.2	0.6	7.1	10.1
MDG	4134	1.0	1.0	7.3	20.6
MG3D	12164	1.5	0.9	13.3	48.8 [a]
OCEAN	2947	1.4	0.7	8.9	16.7
TRACK	136	1.0	0.4	4.0	5.2
TRFD	864	2.2	0.8	16.0	43.2
QCD	416	1.1	0.5	7.4	20.8 [b]
SPEC77	2375	2.4	2.4	10.2	15.7

[a] eliminated file I/O
[b] parallelized random number generator

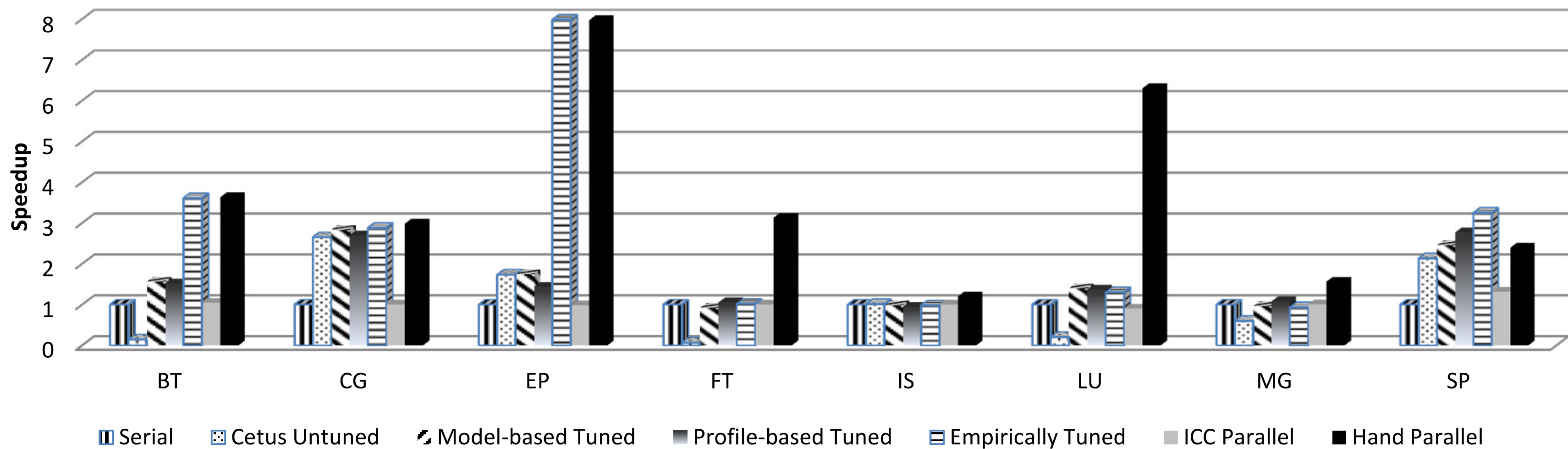
Rudolf Eigenmann, Jay Hoeflinger, and David Padua, **On the Automatic Parallelization of the Perfect Benchmarks**. *IEEE Transactions on Parallel and Distributed Systems*, volume 9, number 1, January 1998, pages 5-23.

Performance of Individual Techniques in Manually Improved Programs (1995)

Technique	ADM	ARC2D	BDNA	DYFESM	FLO52	MDG	MG3D	OCEAN	QCD	SPEC77	TRACK	TRFD
privatize arrays	9.6	1.2	1.4	2.2	1	21	18	3.8	8.2	6.8	6	13.3
parallelize complex reductions			3.3	2.1	1.1	21	15.2			3.4		
substitute generalized induction variables								8.3				12.7
parallelize loops with non-affine array subscripts				3				11.5				13

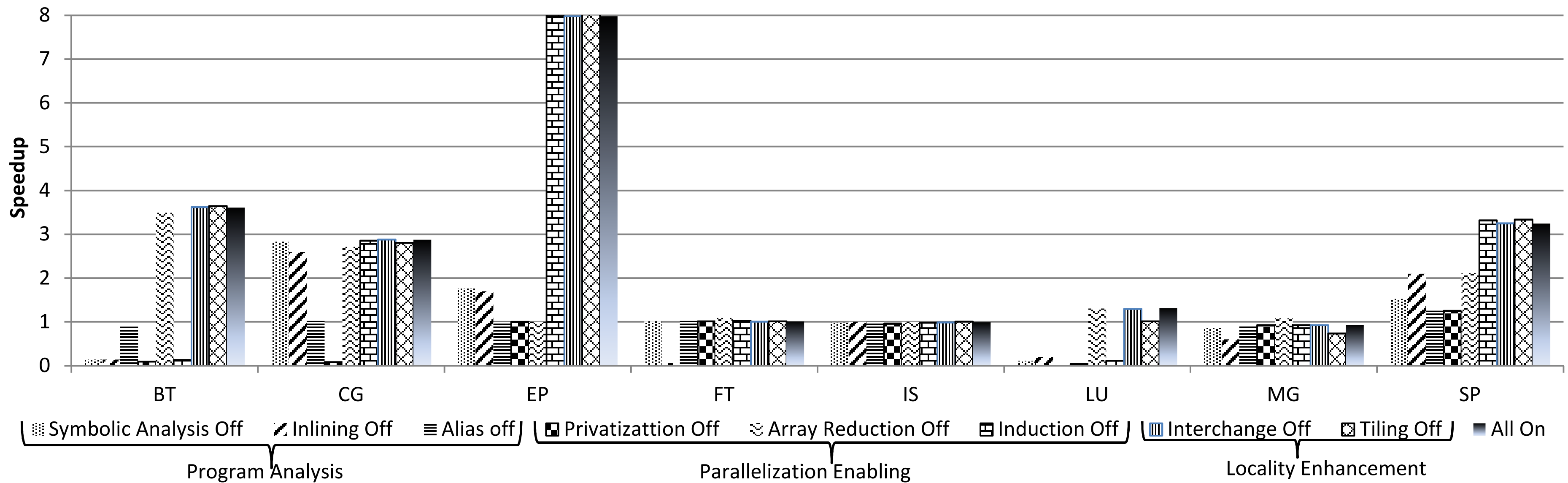
Performance loss when disabling individual techniques (Cedar machine)

Overall Performance of the Cetus and ICC Compilers (2011)



NAS (Class A) Benchmarks on 8-core x86 processor

Performance of Individual Cetus Techniques (2011)



NAS Benchmarks (Class A) on 8-core x86 processor

IV. Analysis and Transformation Techniques

- 1 Data-dependence analysis
- 2 Parallelism enabling transformations
- 3 Techniques for multiprocessors/multicores
- 4 Advanced program analysis
- 5 Dynamic decision making
- 6 Techniques for vector architectures
- 7 Techniques for heterogeneous multicores
- 8 Techniques distributed-memory machines

IV.1 Data Dependence Testing

Earlier, we have considered the simple case of a 1-dimensional array enclosed by a single loop:

```
DO i=1,n
  a(4*i) = . . .
  . . .    =  a(2*i+1)
ENDDO
```

the question to answer:
can 4*i ever be equal to 2*i+1 within $i \in [1,n]$?

In general: given
- two subscript functions *f* and *g* and
- loop bounds lower, upper.

Does

$f(i_1) = g(i_2)$ have a solution such that
$lower \leq i_1, i_2 \leq upper$?

DDTests: doubly-nested loops

- Multiple loop indices:

```
DO i=1,n
  DO j=1,m
    X(a1*i + b1*j + c1) = . . .
    . . .   =  X(a2*i + b2*j + c2)
  ENDDO
ENDDO
```

dependence problem:

$a_1*i_1 - a_2*i_2 + b_1*j_1 - b_2*j_2 = c_2 - c_1$

$1 \leq i_1, i_2 \leq n$

$1 \leq j_1, j_2 \leq m$

Almost all DD tests expect the coefficients a_x to be integer constants. Such subscript expressions are called *affine.*

DDTests: even more complexity

- Multiple loop indices, multi-dimensional array:

```
DO i=1,n
  DO j=1,m
    X(a1*i1 + b1*j1 + c1, d1*i1 + e1*j1 + f1) = . . .
              . . .   =  X(a2*i2 + b2*j2 + c2, d2*i2 +e2*j2 + f2)
  ENDDO
ENDDO
```

dependence problem:

$a_1*i_1 - a_2*i_2 + b_1*j_1 - b_2*j_2 = c_2 - c_1$

$d_1*i_1 - d_2*i_2 + e_1*j_1 - e_2*j_2 = f_2 - f_1$

$1 \leq i_1, i_2 \leq n$

$1 \leq j_1, j_2 \leq m$

Data Dependence Tests:
The Simple Case

Note: variables i_1, i_2 are integers → diophantine equations.

Equation $a * i_1 - b * i_2 = c$ has a solution if and only iff

gcd(a,b) (evenly) divides c

in our example this means: gcd(4,2)=2, which does not divide 1 and thus there is no dependence.

If there **is** a solution, we can test if it lies within the loop bounds. If not, then there is no dependence.

Performing the GCD Test

- The diophantine equation

$$a_1 * i_1 + a_2 * i_2 + \ldots + a_n * i_n = c$$

has a solution iff $\gcd(a_1, a_2, \ldots, a_n)$ evenly divides c

Examples:

15*i +6*j -9*k = 12	has a solution	gcd=3
2*i + 7*j = 3	has a solution	gcd=1
9*i + 3*j + 6*k = 5	has no solution	gcd=3

Euklid Algorithm: find gcd(a,b)

```
Repeat
    a ← a mod b
    swap a,b
Until b=0
```

→*The resulting a is the gcd*

for more than two numbers:
gcd(a,b,c) = (gcd(a,gcd(b,c))

Other Data Dependence Tests

- The GCD test is simple but not accurate
- Other tests
  - Banerjee(-Wolfe) test: widely used test
  - Power Test: improvement over Banerjee test
  - Omega test: “precise” test, most accurate for linear subscripts
  - Range test: handles non-linear and symbolic subscripts
  - many variants of these tests

The Banerjee(-Wolfe) Test

Basic idea:

if the total subscript range accessed by *ref1* does not overlap with the range accessed by *ref2*, then *ref1* and *ref2* are independent.

```
DO j=1,100
   a(j) = …
   …   = a(j+200)
ENDDO
```

ranges accesses:

[1:100]

[201:300]

→ independent

Mathematical Formulation of the Test – Banerjee's Inequalities

$j_1 - j_2 = 200$

Min: 1-100=-99

Max: 100-1=99

The general case of a doubly-nested loop and single subscript, as shown on Slide 40:

$a_1 * i_1 - a_2 * i_2 + b_1 * j_1 - b_2 * j_2 = c_2 - c_1$

Min: $a_1 - a_2 * n$
Max: $a_1 * n - a_2$

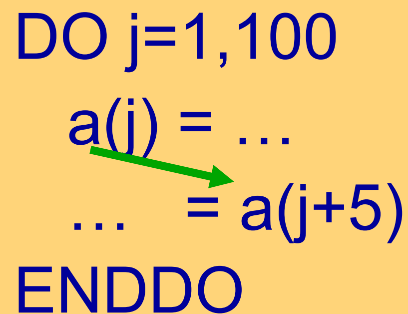
Min: $b_1 - b_2 * m$
Max: $b_1 * n - b_2$

Assuming positive coefficients

Multiple dimensions: apply test separately on each subscript or linearize

Banerjee(-Wolfe) Test continued

Weakness of the test:



ranges accessed:

[1:100]

[6:105]

→ no dependence ?

We did not take into consideration that only *loop-carried* dependences matter for parallelization.

A loop-carried flow dependence only exists, if a read in some iteration, j_1, conflicts with a write in some later iteration, $j_2 > j_1$

Using Dependence Direction Information in the Banerjee(-Wolfe) Test

Idea for overcoming the weakness:

for loop-carried dependences, make use of the fact that j in *ref2* is greater than in *ref1*

Still considering the potential flow dependence from a(j) to a(j+5)

```
DO j=1,100
 a(j) = …
   …   = a(j+5)
ENDDO
```

Ranges accessed by iteration j_1 and any other iteration j_2, where $j_1 < j_2$:

$[j_1]$

$[j_1+6:105]$

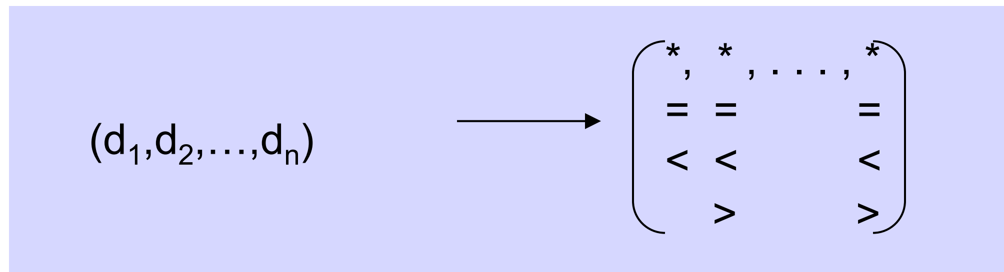
→ Independent for ">" direction

Clearly, this loop **has** a dependence. But, it is an anti-dependence from a(j+5) to a(j)

This is commonly referred to as the *Banerjee test with direction vectors.*

DD Testing with Direction Vectors

Considering direction vectors can increase the complexity of the DD test substantially. For long vectors (corresponding to deeply-nested loops), there are many possible combinations of directions.



A possible algorithm:

1. try (*,*…*) , i.e., do not consider directions
2. (if not independent) try (<,*,*…*), (=,*,*…*)
3. (if still not independent) try (<,<,*…*),(<,>,*…*) ,(<,=,*…*)
   (=,=,*…*), (=,<,*…*)

. . .

(This forms a tree)

Data-dependence Test Driver

```
procedure DataDependenceAnalysis( PROG )
input : Program representing all source files: PROG
output : Data dependence graph containing dependence arcs DDG
// Collect all FOR loops meeting eligibility
// Checks: Canonical, FunctionCall, ControlFlowModifier
ELIGIBLE LOOPS = getOutermostEligibleLoops( PROG )
foreach LOOP in ELIGIBLE LOOPS
        // Obtain lower bounds, upper bounds and loop steps
        //      for this loop and all enclosed loops i.e. the loop-nest
        // Substitute symbolic information if available,
        LOOP_INFO = collectLoopInformation( LOOP and enclosed nest )
        // Collect all array access expressions appearing within the
        //    body of this loop, this includes enclosed loops and non-perfectly
        //    nested statements
        ACCESSES = collectArrayAccesses( LOOP and enclosed nest )
        // Traverse all array accesses, test relevant pairs and
        //    create a set of dependence arcs for the loop-nest
        LOOP_DDG = runDependenceTest( LOOP_INFO, ACCESSES )
        // Add loop dependence graph to the program-wide DDG
        // The program-wide DDG is initially empty
        DDG += LOOP_DDG
// return the program-wide data dependence graph once all loops are done
return DDG
```

Data-dependence Test Driver (continued)

```
procedure runDependenceTest( LOOP_INFO, ACCESSES )
input : Loop information for the current loop nest LOOP_INFO
        List of array access expressions, ACCESSES
output : Loop data dependence graph LOOP_DDG
foreach ARRAY_1 in ACCESSES of type write
    // Obtain alias information i.e. aliases to this array name
    // Alias information in Cetus is generated through points-to analysis
    ALIAS_SET = getAliases( ARRAY_1 )
    // Collect all expressions/references to the same array from the entire list of accesses
    TEST_LIST = getOtherReferences( ALIAS_SET, ACCESSES )
    foreach ARRAY_2 in TEST_LIST
        // Obtain the common loops enclosing the pair
        COMMON NEST = getCommonNest( ARRAY_1, ARRAY_2 )
        // Possibly empty set of direction vectors under which
        //    dependence exists is returned by the test
        DV_SET = testAccessPair( ARRAY_1, ARRAY_2, COMMON_NEST, LOOP_INFO )
        foreach DV in DV_SET
            // Create arc from source to sink
            DEP_ARC = buildDependenceArc( ARRAY_1, ARRAY_2, DV )
            // Build the loop dependence graph by accumulating all arcs
            LOOP_DDG += DEP ARC
// All expressions have been tested, return the loop dependence graph
return LOOP_DDG
```

Data-dependence Test Driver (continued)

```
procedure testAccessPair( A1, A2, COMMON_NEST, LOOP_INFO)
input :     Pair of array accesses to be tested A1 and A2
            Nest of common enclosing loops COMMON NEST
            Information for these loops LOOP INFO
output :    Possibly empty set of direction vectors under
            which dependence exists DV SET
// Partition the subscripts of the array accesses into dimension pairs
// Coupled subscripts may be handled
PARTITIONS = partitionSubscripts( A1, A2, COMMON_NEST )
foreach PARTITION in PARTITIONS
    // Depending on the number of loop index variables in the partition,
    // use the corresponding test.
    if( ZIV )    // zero index variables ZIV
        DVs = simpleZIVTest( PARTITION )
    else         // single or multi-loop index variables: SIV, MIV
        // traverse and prune over tree of direction vectors, collect DVs where
        // dependence exists (traversal not shown here)
        foreach DV in DV_TREE using prune
            // In Cetus, the MIV test is performed using Banerjee or Range test
            DVs += MIVTest( PARTITION, DV, COMMON_NEST, LOOP_INFO )
// Merge DVs for all partitions
DV_SET = merge( DVs )
return DV_SET
```

Non-linear and Symbolic DD Testing

Weakness of most data dependence tests:
subscripts and loop bounds must be *affine*, i.e., linear with integer-constant coefficients

Approach of the Range Test:

- capture subscript ranges symbolically
- compare ranges: find their upper and lower bounds by determining *monotonicity*. Monotonically increasing/decreasing ranges can be compared by comparing their upper and lower bounds.

The Range Test

Basic idea :

1. Find the range of array accesses made in a given loop iteration *j => r(j).*
2. If *r(j)* does not overlap with *r(j+1) then* there is no cross-iteration dependence

Symbolic comparison of ranges *r1* and *r2:*
max(r1)<min(r2) OR min(r1)>max(r2) => no overlap

Example: testing independence of the outer loop:

```
DO i=1,n
  DO j=1,m
    A(i*m+j) = 0
  ENDDO
ENDDO
```

range of A accessed in iteration i_x: $[i_x*m+1:\underbrace{(i_x+1)*m}_{ub_x}]$

range of A accessed in iteration i_x+1: $[\underbrace{(i_x+1)*m+1}_{lb_{x+1}}:(i_x+2)*m]$

$ub_x < lb_{x+1} \Rightarrow$ no cross-iteration dependence

Range Test continued

we need powerful expression manipulation and comparison utilities

```
DO i1=L1,U1
  ...
  DO in=Ln,Un
     A(f(i1,...in)) = ...
          ...      = A(g(i1,...in))
  ENDDO
 ...
ENDDO
```

Assume *f,g* are monotonically increasing w.r.t. all i_x:
find upper bound of access range at loop k, 1<k<n:
successively substitute i_x with U_x, x={n,n-1,...,k-1}
lowerbound is computed analogously

If *f,g* are monotonically decreasing w.r.t. some i_y, then substitute L_y when computing the upper bound.

Determining monotonicity: consider d = f(...,i_k,...) - f(...,i_k-1,...)
If d>0 (for all values of i_k) then f is monotonically increasing w.r.t. k
If d<0 (for all values of i_k) then f is monotonically decreasing w.r.t. k

we need range analysis

What about symbolic coefficients?
- in many cases they cancel out
- if not, find their range (i.e., all possible values they can assume at this point in the program), and replace them by the upper or lower bound of the range.

Handling Non-contiguous Ranges

```
DO i1=1,u1
  DO i2=1,u2
     A(n*i1+m*i2)) = …
  ENDDO
ENDDO
```

The basic Range Test finds independence
of the outer loop
    if n >= u2 and m=1
But not
    if n=1 and m>=u1

Idea:
- temporarily (during program analysis) interchange the loops,
- test independence,
- interchange back

Issues:
- legality of loop interchanging,
- change of parallelism as a result of loop interchanging

Some Engineering Tasks and Questions for DD Test Pass Writers

- Start with the simple case: linear (affine) subscripts, single nests with 1-dim arrays. Subscript and loop bounds are integer constants. Stride 1 loop, lower bound =1
- Deal with multiple array dims and loop nests
- Add capabilities for non-stride-1 loops and lower bounds ≠1
- How to deal with symbolic subscript coefficients and bounds
- Ignore dependences in private variables and reductions
- Generate DD vectors
- Mark parallel loops
- Things to think about:
  - -- how to handle loop-variant coefficients
  - -- how to deal with private, reduction, induction variables
  - -- how to represent DD information
  - -- how to display the DD info
  - -- how to deal with non-parallelizable loops (IO op, function calls, other?)
  - -- how to find eligible DO loops?
  - -- how to find eligible loop bounds, array subscripts?
  - -- what is the result of the pass? Generate DD info or set parallel loop flags?
  - -- what symbolic analysis capabilities are needed?

Data-Dependence Test, References

- Banerjee/Wolfe test
  - M.Wolfe, U.Banerjee, "Data Dependence and its Application to Parallel Processing", Int. J. of Parallel Programming, Vol.16, No.2, pp.137-178, 1987
- Power Test
  - M. Wolfe and C.W. Tseng, The Power Test for Data Dependence, IEEE Transactionson Parallel and Distributed Systems, IEEE Computer Society, 3(5), 591-601,1992.
- Range test
  - William Blume and Rudolf Eigenmann. Non-Linear and Symbolic Data Dependence Testing, IEEE Transactions of Parallel and Distributed Systems, Volume 9, Number 12, pages 1180-1194, December 1998.
- Omega test
  - William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence. Proceedings of the 1991 ACM/IEEE Conference on Supercomputing,1991
- I Test
  - Xiangyun Kong, David Klappholz, and Kleanthis Psarris, "The I Test: A New Test for Subscript Data Dependence," *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. II, pages 204-211, August 1990.

IV.2 Parallelism Enabling Techniques

Advanced Privatization

loop-carried anti dependence

```
DO i=1,n
  t      =  A(i)+B(i)
  C(i) =  t + t**2
ENDDO
```

scalar privatization

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(t)
DO i=1,n
  t      =  A(i)+B(i)
  C(i) =  t + t**2
ENDDO
```

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

array privatization

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

Array Privatization

```
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)
ENDDO
```

```
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)
ENDDO
```

Capabilities needed for Array Privatization

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

Array Privatization continued

Array privatization algorithm:

- For each loop nest:
  - iterate from innermost to outermost loop:
    - for each statement in the loop
      - Find array 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 uses (expand from range per-iteration to entire iteration space); record them as Defs and Uses for this loop

Some Engineering Tasks and Questions for Privatization Pass Writers

- Start with scalar privatization
- Next step: array privatization with simple ranges (contiguous; no range merge) and singly-nested loops
- Deal with multiply-nested loops (-> range aggregation)
- Add capabilities for merging ranges
- Implement advanced range representation (symbolic bounds, non-contiguous ranges)
- Deal with conditional definitions and uses (too advanced for this course)
- Things to think about
  - what symbolic analysis capabilities are needed?
  - how to represent advanced ranges?
  - how to deal with loop-variant subscript terms?
  - how to represent private variables?

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

Reduction Parallelization

Scalar Reduction

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

loop-carried flow dependence

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

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

Privatized reduction implementation

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

Expanded reduction implementation

```
DO i=1,num_proc
   s(i)=0
ENDDO
!$OMP PARALLEL DO
DO i=1,n
  s(my_proc)=s(my_proc)+A(i)
ENDDO
DO i=1,num_proc
  sum=sum+s(i)
ENDDO
```

Parallelizing Array Reductions

Array Reductions (a.k.a. irregular or histogram reductions)

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

Privatized reduction implementation

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

Expanded reduction implementation

```
DIMENSION sum(m),s(m,#proc)
!$OMP PARALLEL DO
DO i=1,m
DO j=1,#proc
   s(i,j)=0
ENDDO
ENDDO
!$OMP PARALLEL DO
DO i=1,n
  s(expr,my_proc)=s(expr,my_proc)+A(i)
ENDDO
!$OMP PARALLEL DO
DO i=1,m
DO j=1,#proc
   sum(i)=sum(i)+s(i,j)
ENDDO
ENDDO
```

Note, OpenMP 1.0 does not support such array reductions

Recognizing Reductions

Recognition Criteria:

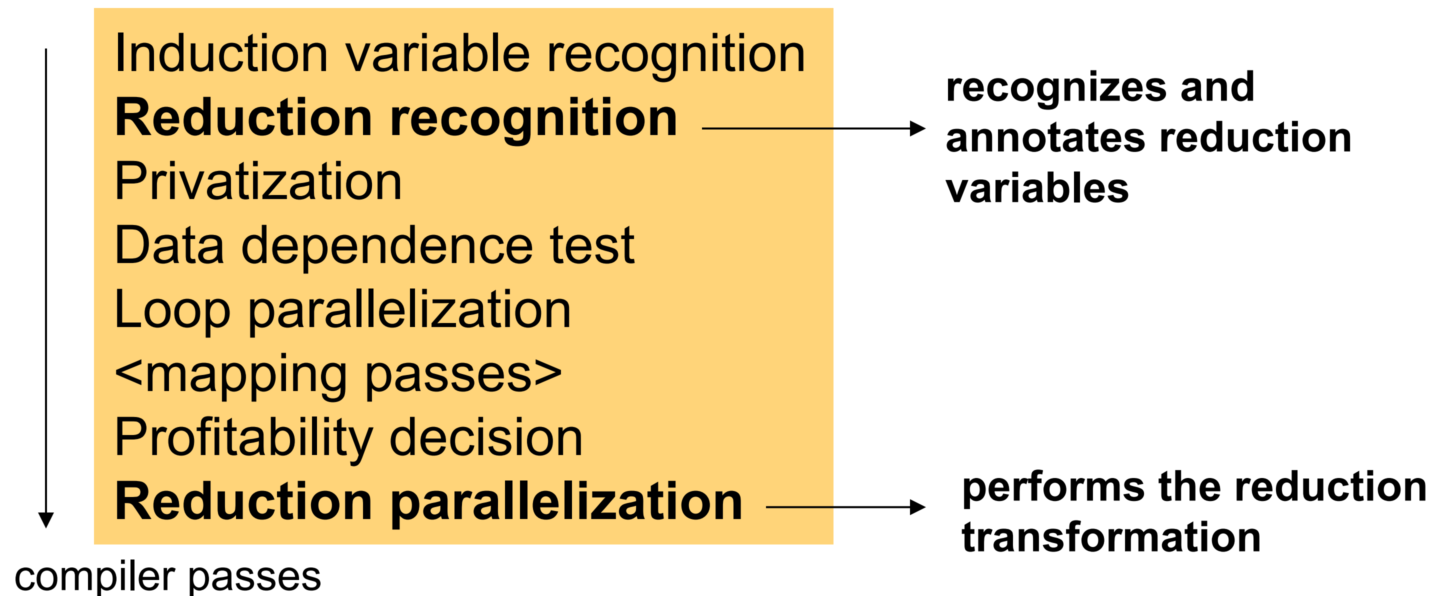
1. the loop may contain one or more reduction statements of the form $X=X \otimes expr$, where
   - X is either scalar or an array expression, *a[sub]*
     (*sub* must be the same on LHS and RHS)
   - $\otimes$ is a reduction operation, such as +, *, min, max
2. X must not be used in any non-reduction statement of the loop, nor in expr

Reduction Recognition Algorithm

```
procedure RecognizeSumReductions (L)
Input : Loop L
Output: reduction annotations for loop L, inserted in the IR
    REDUCTION = {} // set of candidate reduction expressions
    REF = {} // set of non-reduction variables referenced in L
    foreach stmt in L
        localREFs = findREF(stmt) // gather all variables referenced in stmt
        if (stmt is AssignmentStatement)
            candidate = lhs_expr(stmt)
            increment = rhs_expr(stmt) – candidate // symbolic subtraction
            if ( !(baseSymbol(candidate) in  findREF(increment)) ) // criterion1 is satisfied
                REDUCTION = REDUCTION ∪ candidate
                localREFs = findREF(increment) // all variables referenced in inc. expr.
        REF = REF ∪ localREFs // collect non-reduction variables for criterion 2
    foreach expr in REDUCTION
        if ( ! (baseSymbol(expr) in REF) ) // criterion 2 is satisfied
            if (expr is ArrayAccess AND expr.subscript is loop-variant)
                CreateAnnotation(sum-reduction, ARRAY, expr)
            else
                CreateAnnotation(sum-reduction, SCALAR, expr)
end procedure
```

Reduction Compiler Passes

Reduction recognition and parallelization passes:



Performance Considerations for Reduction Parallelization

- Parallelized reductions execute substantially more code than their serial versions $\Rightarrow$ overhead if the reduction (*n*) 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 $\Rightarrow$ compression schemes can become useful.

Induction Variable Substitution

loop-carried flow dependence

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

→

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

This is the simple case of an induction variable

Generalized Induction Variables

```
ind=k
DO j=1,n
   ind  = ind + j
   A(ind) = B(j)
ENDDO
```

→

```
Parallel DO j=1,n
   A(k+(j**2+j)/2) = B(j)
ENDDO
```

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

```
DO i=1,n
   DO j=1,i
      ind  = ind + 1
      A(ind) = B(i)
   ENDDO
ENDDO
```

Recognizing GIVs

- Pattern Matching:
  - find induction statements in a loop nest of the form iv=iv+expr or iv=iv*expr, where iv is an 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

GIV Closed-form Computation and Substitution Algorithm

Loop structure L_0: stmt type

```
For j: 1..ub
…
S1:  iv=iv+exp          I
…
S2:  loop using iv      L
…
S3:  stmt using iv      U
…
Rof
```

Step1: find the increment rel. to start of loop L

```
FindIncrement(L)
  inc=0
  foreach si of type I,L
    if type(si)=I       inc += exp
    else  /* L */        inc+= FindIncrement(si)
    inc_after[si]=inc
  inc_into_loop[L]= ∑1^(j-1)(inc) ; inc may depend
  return ∑1^ub(inc)               ; on j
```

Main:

```
totalinc = FindIncrement(L0)
Replace(L0,iv)
InsertStatement(“iv =  iv+totalinc”)
```

Insert this statement If iv is live-out

Step 2: substitute IV

```
Replace (L,initval)
  val = initval+inc_into_loop[L]
  foreach si of type I,L,U
    if type(si)=L      Replace(si,val)
    if type(si)=L,I    val=initialval
                         +inc_into_loop[L]
                         +inc_after[si]
    if type(si)=U   Substitute(si,iv,val)
```

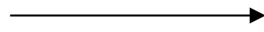
For coupled GIVs: begin with independent iv.

Induction Variables, References

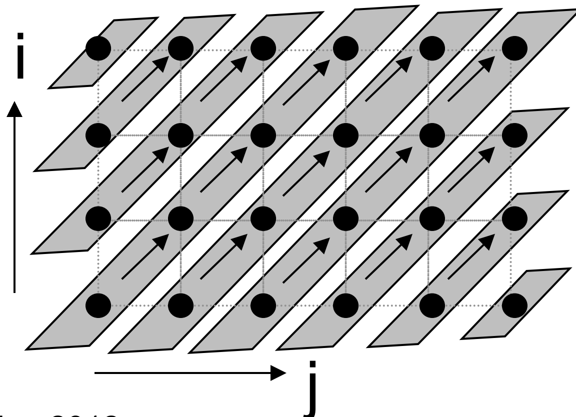
- B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. ACM Int. Conf. on Supercomputing (ICS'95), June 1995.
- Mohammad R. Haghighat , Constantine D. Polychronopoulos, Symbolic analysis for parallelizing compilers, ACM Transactions on Programming Languages and Systems (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

Loop Skewing

```
DO i=1,4
  DO j=1,6
    A(i,j)= A(i-1,j-1)
  ENDDO
ENDDO
```



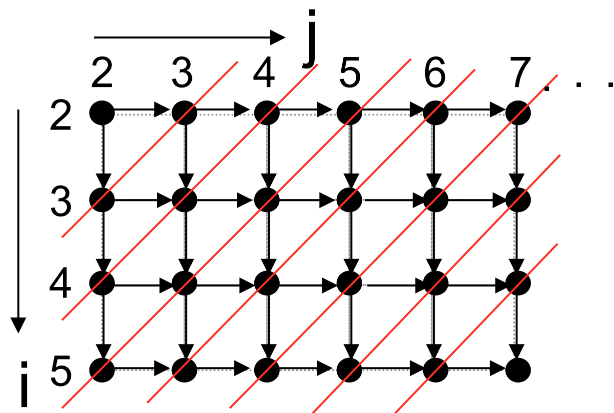
```
!$OMP PARALLEL DO
DO set=1,9
  i = max(5-set,1)
  j = max(-3+set,1)
  setsize = min(4,5-abs(set-5))
  DO k=0,setsize-1
      A(i+k,j+k)=A(i-1+k,j-1+k)
  ENDDO
ENDDO
```



Iteration space graph:
Shared regions show sets of iterations in the transformed code that can be executed in parallel.

Loop Skewing for the Wavefront Method

```
DO i=2,n-1
  DO j=2,n-1
    A(i,j)= (A(i+1,j) +A(i-1,j)
             +A(i,j+1) +A(i,j-1))/4
  ENDDO
ENDDO
```



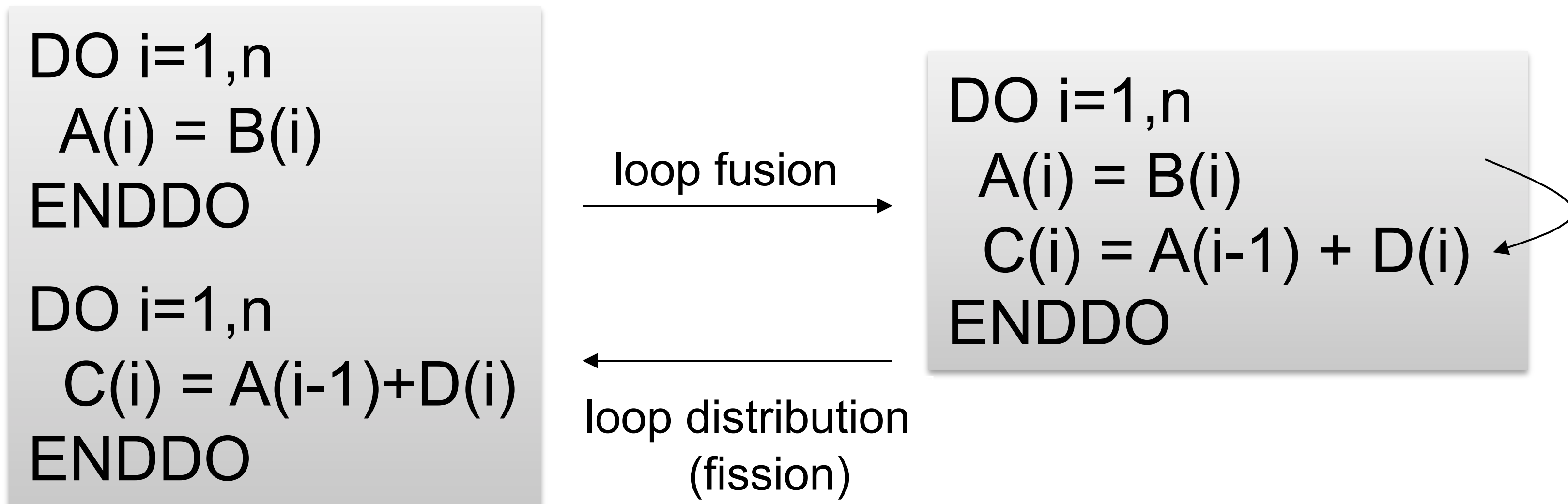
Outer loop is serial
Inner loop is parallel

```
DO j=4, n+n-2
       DOALL i= max(2, n- j+ 1), min(n- 1, j- 2)
              A(i, j- i) = (A(i+ 1, j- i) + A(i- 1, j- i)
                            +A(i, j+ 1- i) + A(i, j- 1 +i)/4
       ENDDO
ENDDO
```

IV.3 Techniques for Multiprocessors:

Mapping Parallelism to Shared-memory Machines

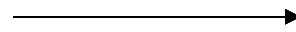
Loop Fusion and Distribution



- Loop fusion is the reverse of loop distribution
- Fusion reduces the loop fork/join overhead and enhances data affinity
- Distribution inserts a barrier synchronization between parallel loops
- Both transformations reorder computation
- Legality: dependences in fused loop must be lexically forward

Loop Distribution Enables Other Techniques

```
DO i=1,n
   A(i) = B(i)+A(i-1)
   DO j=1,m
      D(i,j)=E(i,j)
   ENDDO
ENDDO
```



- enables interchange
- separates out partial paralleism

```
DO i=1,n
   A(i) = B(i)+A(i-1)
ENDDO

DOALL j=1,m
   DO i=1,n
      D(i,j)=E(i,j)
   ENDDO
ENDDO
```

In a program with multiply-nested loops, there can be a large number of possible program variants obtained through distribution and interchanging

Enforcing Data Dependence

Criterion for correct transformation and execution of a computation involving a data dependence with vector v : (=,…<,…*)

Let L_s be the outermost loop with non-“=” DD-direction :

L_s

- L_s must be executed serially
- The direction at L_s must be “<”

Same rule applies to all dependences

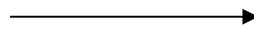
Note that a data dependence is defined with respect to an ordered execution. For autoparallelization, this is the serial program order.

User-defined, fully parallel loops by definition do not have cross-iteration dependences. Legality rules for transforming already parallel programs are different.

Loop Interchange

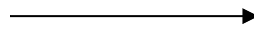
Legality of Loop interchange and resulting parallelism can be tested with the above rules:
After loop interchange, the two conditions must still hold.

```
DO i=1,n
  DOALL j=1,m
      A(i,j) = A(i-1,j)
  ENDDO
ENDDO
```



```
DOALL j=1,m
  DO i=1,n
      A(i,j) = A(i-1,j)
  ENDDO
ENDDO
```

```
DOALL i=1,n
  DO j=1,m
      A(i,j) = A(i-1,j-1)
  ENDDO
ENDDO
```



```
DOALL j=1,m
  DO i=1,n
      A(i,j) = A(i-1,j-1)
  ENDDO
ENDDO
```

Loop Coalescing

a.k.a. loop collapsing

```
PARALLEL DO i=1,n
  DO j=1,m
     A(i,j) = B(i,j)
  ENDDO
ENDDO
```

loop coalescing →

```
PARALLEL DO ij=1,n*m
   i = 1 + (ij-1) DIV m
   j = 1 + (ij-1) MOD m
  A(i,j) = B(i,j)
ENDDO
```

Loop coalescing

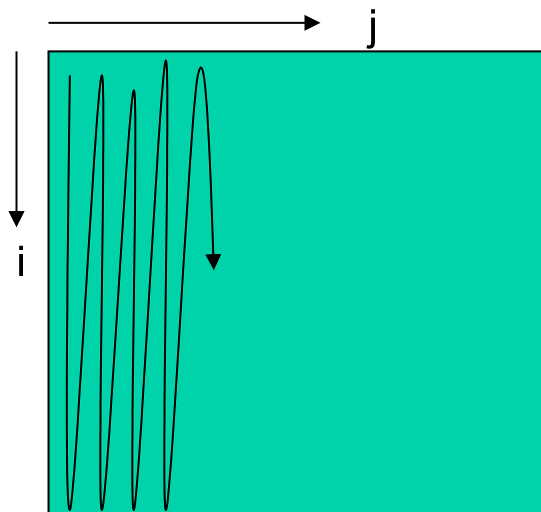
- can increase the number of iterations of a parallel loop
  → load balancing
- adds additional computation
  → overhead

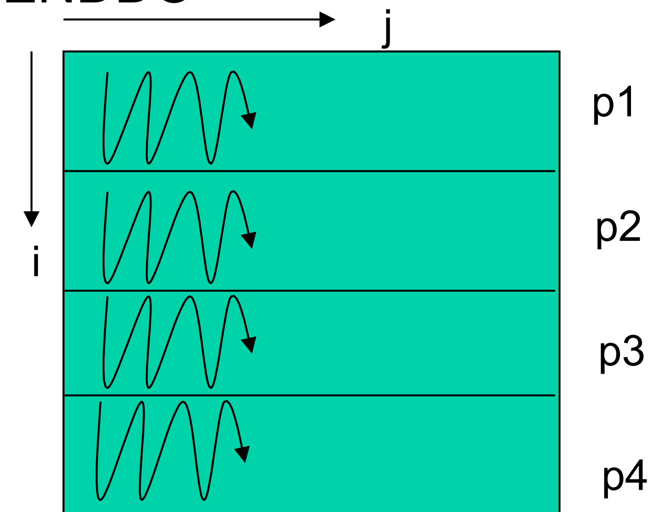
Loop Blocking/Tiling

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

loop blocking

```
DO PARALLEL i1=1,n,block
  DO j=1,m
    DO i=i1,min(i1+block-1,n)
      B(i,j)=A(i,j)+A(i,j-1)
    ENDDO
  ENDDO
ENDDO
```





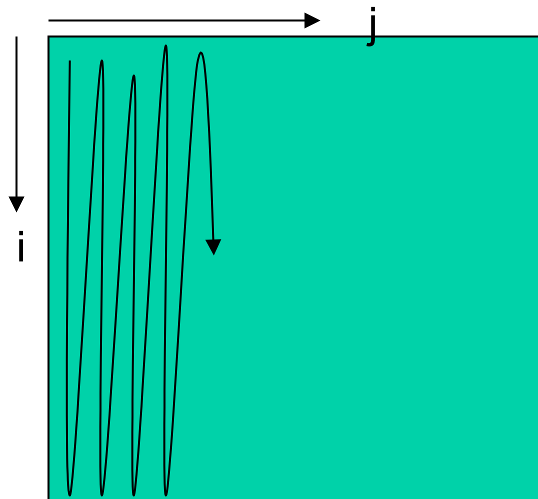
This is basically the same transformation as stripmining, but followed by loop interchanging.

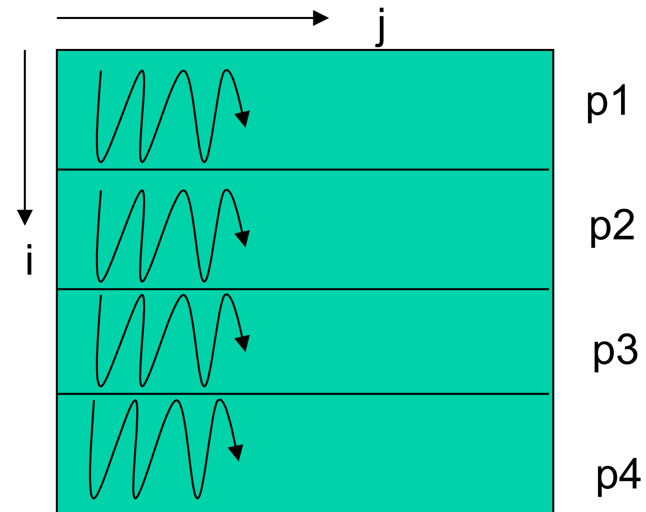
Loop Blocking/ Tiling

continued

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

```
!$OMP PARALLEL
DO j=1,m
!$OMP DO
   DO i=1,n
      B(i,j)=A(i,j)+A(i,j-1)
   ENDDO
!$OMP ENDDO NOWAIT
ENDDO
!$OMP END PARALLEL
```





Choosing the Block Size

The block size must be small enough so that all data references between the use and the reuse fit in cache.

```
DO j=1,m
  DO k=1,block
    … (r1 data references)
    … = A(k,j) + A(k,j-d)
    … (r2 data references)
  ENDDO
ENDDO
```

Number of references made between the access A(k,j) and the access A(k,j-d) when referencing the same memory location:
(r1+r2+3)*d*block
→ **block < cachesize / (r1+r2+3)*d**

If the cache is shared, all cores use it simultaneously. Hence the effective cache size appears smaller:

block < cachesize / (r1+r2+3)*d*num_cores

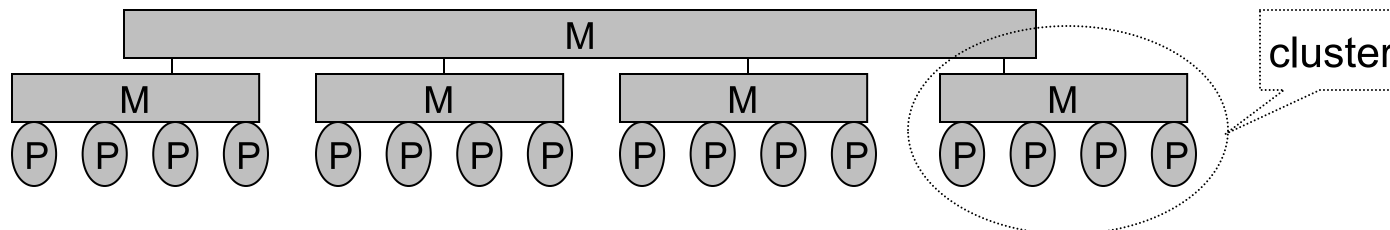
Reference: Zhelong Pan, Brian Armstrong, Hansang Bae and Rudolf Eigenmann, **On the Interaction of Tiling and Automatic Parallelization,** *First International Workshop on OpenMP (Wompat)*, 2005.

Multi-level Parallelism from Single Loops

```
DO i=1,n
  A(i) = B(i)
ENDDO
```

strip mining for multi-level parallelism

```
PARALLEL DO (inter-cluster) i1=1,n,strip
   PARALLEL DO (intra-cluster) i=i1,min(i1+strip-1,n)
      A(i) = B(i)
   ENDDO
ENDDO
```



References

- **High Performance Compilers for Parallel Computing**, Michale Wolfe, Addison-Wesley, ISBN 0-8053-2730-4.
- **Optimizing Compilers for Modern Architectures**: A Dependence-based Approach, Ken Kennedy and John R. Allen, Morgan Kaufmann Publishers, ISBN 1558602860

IV.4 Advanced Program Analysis

Interprocedural Analysis

- Most compiler techniques work intra-procedurally
- Ideally, inter-procedural analyses *and* transformations available
- In practice: inter-procedural operation of basic analysis works well
- Inline expansion helps but no silver bullet

Interprocedural Constant Propagation

Making constant values of variables known across subroutine calls

```
Subroutine A
   j = 150
   call B(j)
END
```

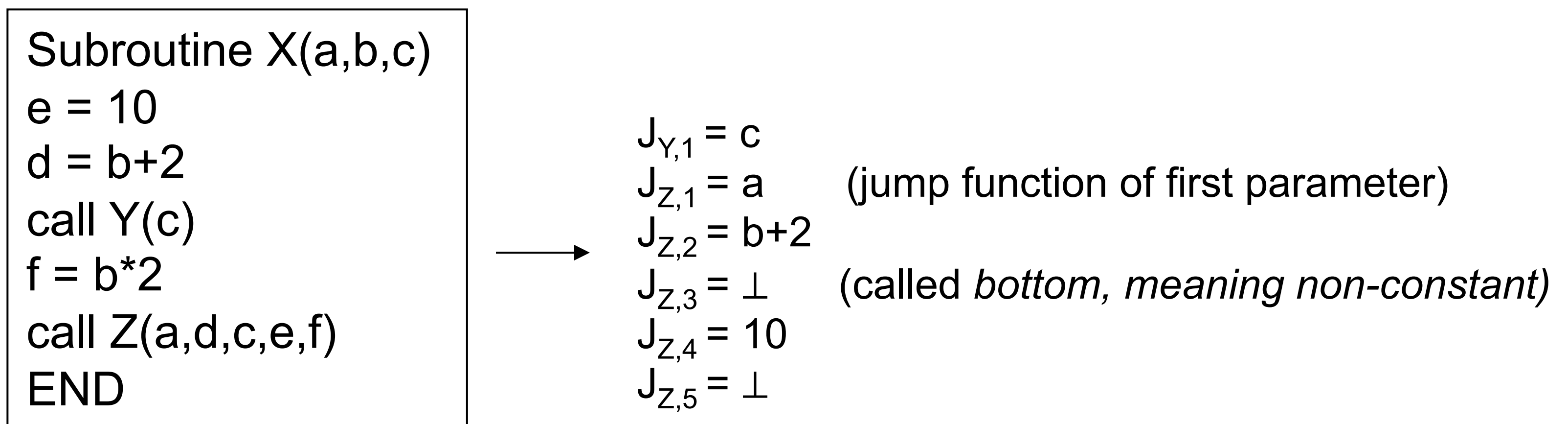
```
Subroutine B(m)

DO k=1,100
   X(i)=X(i+m)
ENDDO

END
```

knowing that m>100 allows this loop to be parallelized

An Algorithm for Interprocedural Constant Propagation

Intra-procedural part:

determine *jump functions* for all subroutines



- Mechanism for finding jump functions: (local) forward substitution and interprocedural MAYMOD information.
- Here we assume the compiler supports jump functions of the form *P+const* (P is a subroutine parameter of the callee).

Constant Propagation Algorithm:

Interprocedural Part

1. initialize all formal parameters to the value T (called *top* = non yet known)
2. for all jump functions:
   – if it is ⊥: set formal parameter value to ⊥ (called bottom = unknown)
   – if it is constant and the value of the formal parameter is the same constant or T : set the value to this constant
3. put all formal parameters on a work queue
4. repeat: take a parameter from the queue until queue is empty

   for all jump functions that contain this parameter:
   - determine the value of the target parameter of this jump function. Set it to this value, or to ⊥ if it is different from a previously set value.
   - if the value of the target parameter changes, put this parameter on the queue

Examples of Constant Propagation

```
x = 3
Call SubY(x)

Subroutine SubY(a)

… = ….a…
```

```
x = 3
Call SubY(x)

Subroutine SubY(a)
b = a+2
Call SubZ(b)

Subroutine SubZ(e)

 …   = … e….
```

```
x = 3
Call SubY(x)

t = 6
Call SubU(t)

Subroutine SubY(a)
b = a+2
Call SubZ(b)

Subroutine SubU(c)
d = c-1
Call SubZ(d)

Subroutine SubZ(e)

 …   = … e….
```

Consider what happens if t = 7

Interprocedural Data-Dependence Analysis

- Motivational examples:

```
DO i=1,n
  call clear(a,i)
ENDDO
```

```
Subroutine clear(x,j)
   x(j) = 0
END
```

```
DO i=1,n
  a(i) = b(i)
  call dupl(a,i)
ENDDO
```

```
Subroutine dupl(x,j)
   x(j) = 2*x(j)
END
```

```
DO k=1,m
   DO i=1,n
      a(i,k) = math(i,k)
      call smooth(a(i,k))
ENDDO
```

```
Subroutine smooth(x,j)
   x(j) = (x(j-1)+x(j)+x(j+1))/3
END
```

Interprocedural Data-Dependence Analysis

- Overall strategy:
  - subroutine inlining
  - move loop into called subroutine
  - collect array access information in callee and use in the analysis of the caller
    - → will be discussed in more detail

Interprocedural Data-Dependence Analysis

- Representing array access information
  - summary information
    - [low:high] or [low:high:stride]
    - sets of the above
  - exact representation
    - essentially all loop bound and subscript information is captured
  - representation of multiple subscripts
    - separate representation
    - linearized

Interprocedural Data-Dependence Analysis

- Reshaping arrays
  - simple conversion
    - matching subarray or 2-D→1-D
  - exact reshaping with div and mod
  - linearizing both arrays
  - equivalencing the two shapes
    - can be used in subroutine inlining

Important: reshaping may lose the implicit assertion that array bounds are not violated!

Symbolic Analysis

- Expression manipulation techniques
  - Expression simplification/normalization
  - Expression comparison
  - Symbolic arithmetic
- Range analysis
  - Find lower/upper bounds of variable values at a given statement
    - For each statement and variable, or
    - Demand-driven, for a given statement and variable

Symbolic Range Analysis Example

```
int foo(int k) {}
                                        []
        int i, j;
                                        []
        double a;
                                        []
        for ( i=0; i<10; ++i ) {
                                        [0<=i<=9]
                a=(0.5*i);
        }
                                        [i=10]
        j=(i+k);
                                        [i=10, j=(i+k)]
        return j;
}
```

Alias Analysis

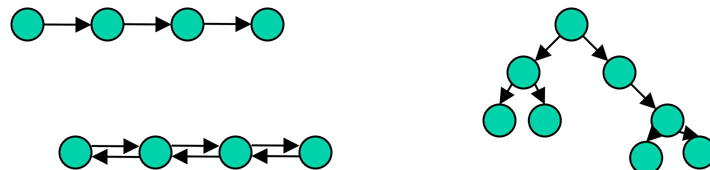
Find references to the same storage by different names
$\Rightarrow$ Program analyses and transformations must consider *all* these names

Simple case: different named variables allocated in same storage location

- Fortran Equivalence statement
- Same variable passed to subroutine by-reference as two different parameters (can happen in Fortran and C++, but not in C)
- Global variable also passed as subroutine parameter

Pointer Alias Analysis

- More complex: variables pointed to by named pointers
  - p=&a; q=&a => *p, *q are aliases
  - Same variable passed to C subroutines via pointer
- Most complex: pointers between dynamic data structure objects
  - This is commonly referred to as *shape analysis*



Is Alias Analysis in Parallelizing Compilers Important?

- Fortran77: alias analysis is simple/absent
  - By Fortran rule, aliased subroutine parameters must not be written to
  - there are no pointers
- C programs: alias analysis is a must
  - Pointers, pointer arithmetic
  - No Fortran-like rule about subroutine parameters
  - Without alias information, compilers would have to be very conservative => big loss of parallelism
  - Classical science/engineering applications do not have dynamic data structures => no shape analysis needed

IV.5 Dynamic Decision Support

Achilles' Heel of Compilers

Big compiler limitations:

- Insufficient compile-time knowledge
  - Input data
  - Architecture parameters (e.g., cache size)
  - Memory layout
- Even if this information is known: Performance models too complex

Effect:

- Unknown profitability of optimizations
- Inconsistent performance behavior
- Conservative behavior of compilers
- Many compiler options
- Users need to experiment with options

Multi-version Code

```
IF (d>n)
   PARALLEL DO i=1,n
      a(i) = a(i+d)
   ENDDO
ELSE
   DO i=1,n
      a(i) = a(i+d)
   ENDDO
```

Limitations

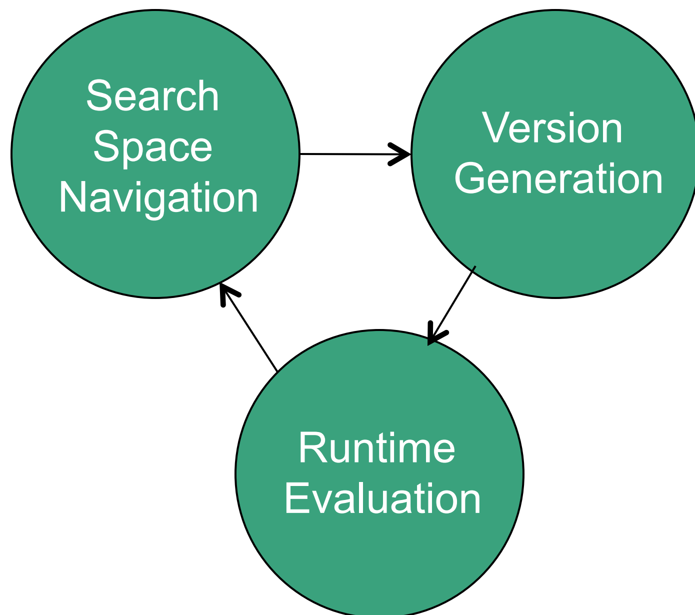
- Less readable
- Additional code
- Not feasible for all optimizations
- Combinatorial explosion when trying to apply to many optimization decisions

Profiling

- Gather missing information in a profile run
  - Compiler instruments code that gathers at runtime information needed for optimization decisions
- Use the gathered profile information for improved decision making in a second compiler invocation
- Training vs. production data
- Initially used for branch prediction. Now increasingly used to guide additional transformations.
- Requires a compiler performance model

Autotuning – Empirical Tuning

Try many optimization variants; pick the best at runtime.



- No compiler performance model needed
- Optimization decisions based on true execution time
- Dependence on training data (same as profiling)
- Potentially huge search space
- Whole-program vs. section-level tuning

Many active research projects

IV.4 Techniques for Vector Machines

Vector Instructions

A vector instruction operates on a number of data elements at once.

Example: vadd va,vb,vc,32

vector operation of length 32 on vector registers va,vb, and vc

- va,vb,vc can be
  - Special cpu registers or memory → classical supercomputers
  - Regular registers, subdivided into shorter partitions (e.g., 64bit register split 8-way) → multi-media extensions
- The operations on the different vector elements can overlap → vector pipelining

Applications of Vector Operations

- Science/engineering applications are typically regular with large loop iteration counts.

  This was ideal for classical supercomputers, which had long vectors (up to 256; vector pipeline startup was costly).

- Graphics applications can exploit “multi-media” register features and instruction sets.

Basic Vector Transformation

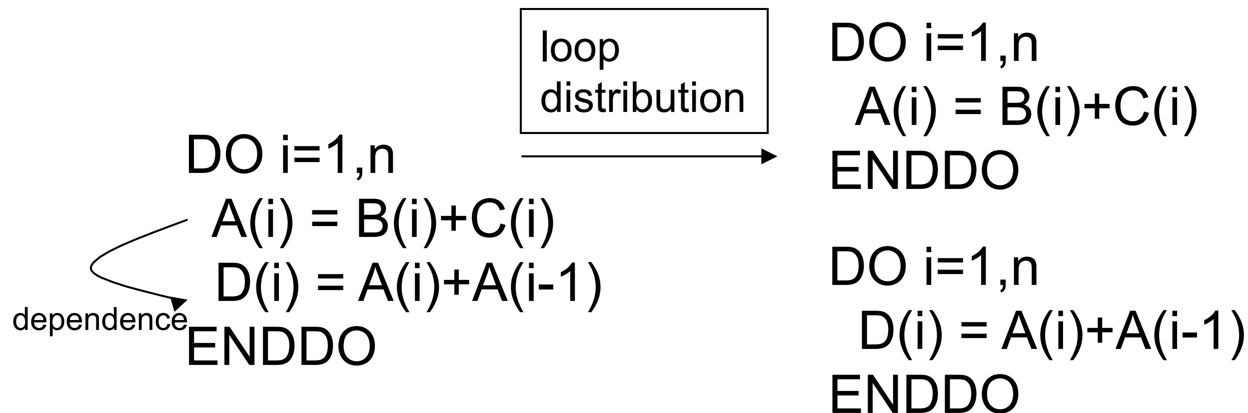
```
DO i=1,n
  A(i) = B(i)+C(i)        →        A(1:n)=B(1:n)+C(1:n)
ENDDO
```

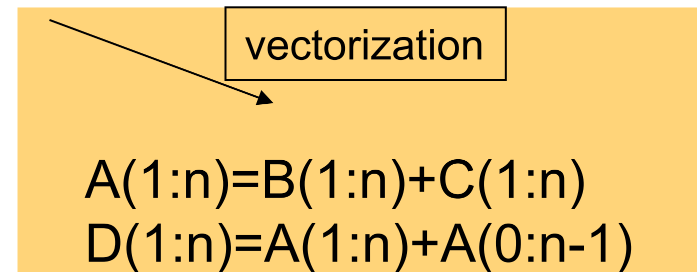
```
DO i=1,n
  A(i) = B(i)+C(i)        →        A(1:n)=B(1:n)+C(1:n)
  C(i-1) = D(i)**2                 C(0:n-1)=D(1:n)**2
ENDDO
```

The triplet notation is interpreted to mean “vector operation”. Notice that this is not (necessarily) the same meaning as in Fortran 90,

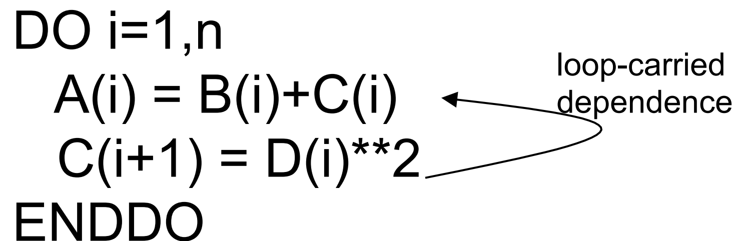
Distribution and Vectorization

The transformation done on the previous slide involves loop distribution. Loop distribution reorders computation and is thus subject to data dependence constraints.





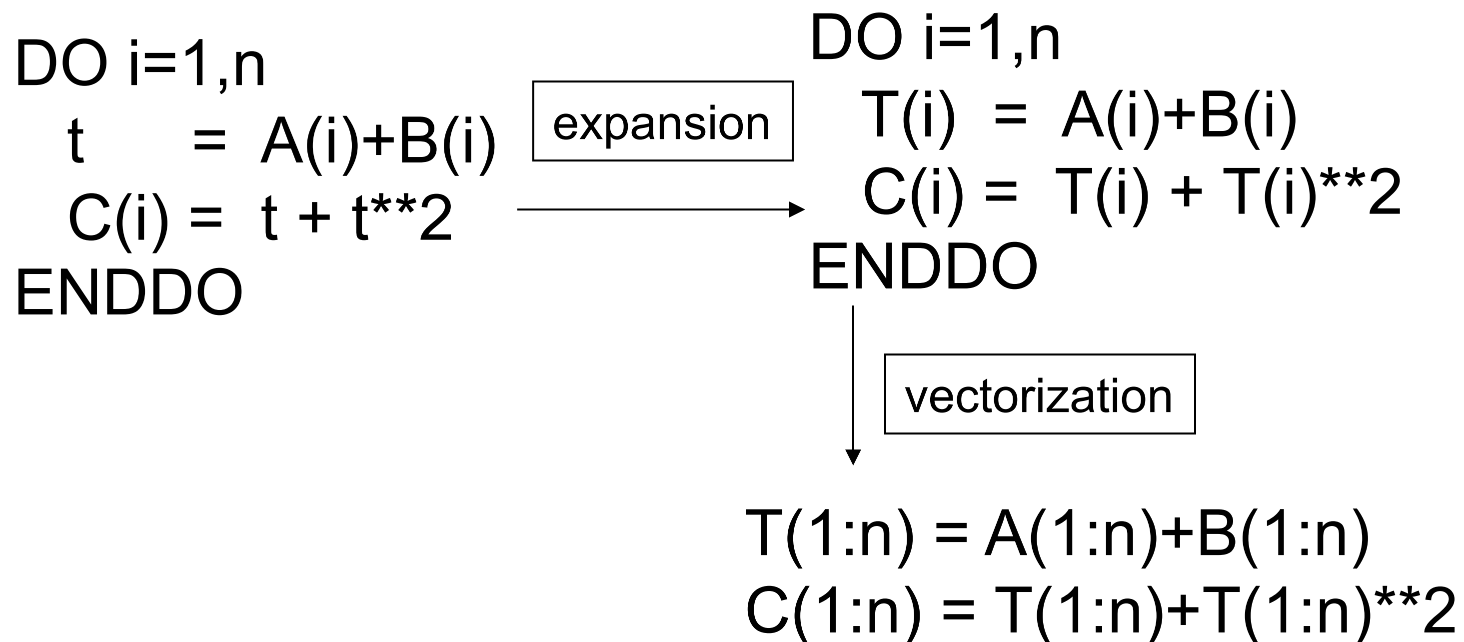
The transformation is not legal if there is a lexical-backward dependence:



Statement reordering may help resolve the problem. However, this is not possible if there is a dependence cycle.

Vectorization Needs Expansion

... as opposed to privatization



Conditional Vectorization

```
DO i=1,n
  IF (A(i) < 0) A(i)=-A(i)
ENDDO
```

conditional vectorization

```
WHERE (A(1:n) < 0) A(1:n)=-A(1:n)
```

Stripmining for Vectorization

```
DO i=1,n
  A(i) = B(i)
ENDDO
```

stripmining →

```
DO i1=1,n,32
  DO i=i1,min(i1+31,n)
     A(i) = B(i)
  ENDDO
ENDDO
```

Stripmining turns a single loop into a doubly-nested loop for two-level parallelism. It also needs to be done by the code-generating compiler to split an operation into chunks of the available vector length.

IV.7 Compiling for Heterogeneous Architectures

Why Heterogeneous Architectures?

- Performance
  - Fast uniprocessor best for serial code
  - Many simple cores best for highly parallel code
  - Special-purpose architectures for accelerating certain code patterns
    - E.g., math co-processor
- Energy
  - Same arguments hold for power savings

Examples of Accelerators

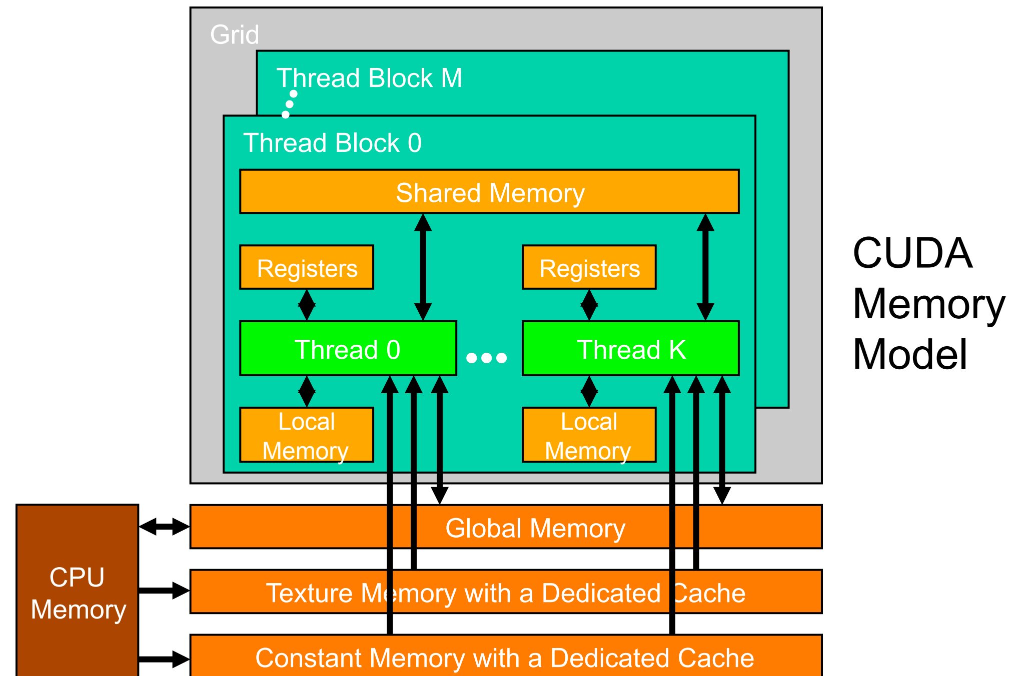
- GPU
- nvidia GPGPU
- IBM Cell
- Intel MIC
- FPGAs
- Crypto processor
- Network processor
- Video Encoder/decoder

Accelerators are typically used as co-processors.
- CPU+accelerator = heterogeneous
- Shared or distributed address space

Accelerator Architecture

Example GPGPU:

- Address space is separate from CPU
- Complex Memory hierarchy
- Large number of cores
- Multithreaded SIMD execution
- Optimized for coalesced (stride-1) accesses



CUDA Memory Model

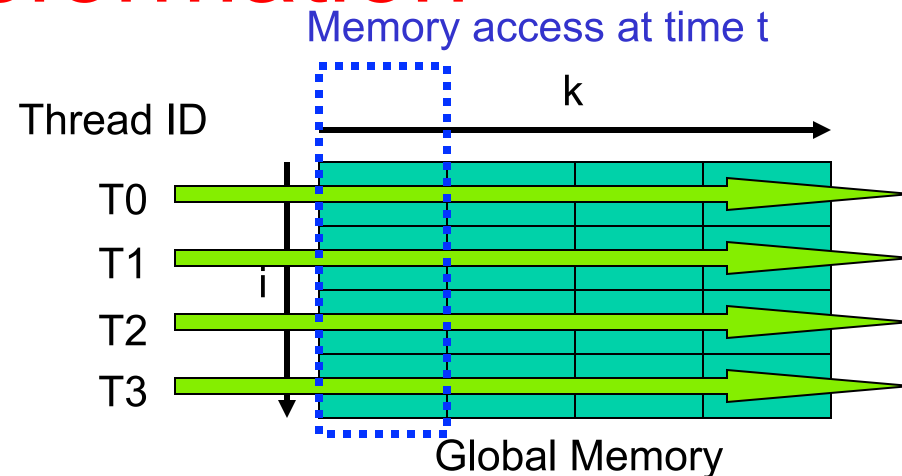
Compiler Optimizations for GPGPUs

- Optimizing GPU Global Memory Accesses
  - Parallel Loop Swap
  - Loop Collapsing
  - Matrix Transpose
- Exploiting GPU On-chip Memories
- Optimizing CPU-GPU Data Movement
  - Resident GPU Variable Analysis
  - Live CPU Variable Analysis
  - Memory Transfer Promotion Optimization

Parallel Loop-Swap Transformation

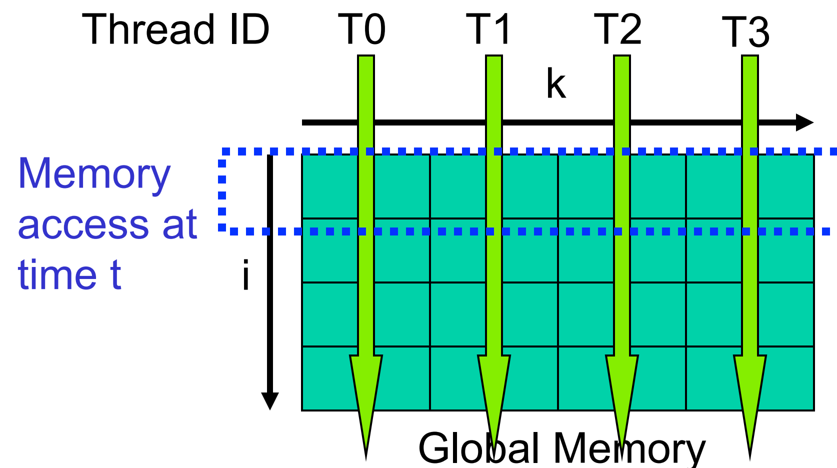
```
#pragma omp parallel for
for(i=0; i< N; i++)
    for(k=0; k<N; k++)
        A[i][k] = B[i][k];
```

Input OpenMP code




```
#pragma omp parallel for
    schedule(static, 1)
for(k=0; k<N; k++)
    for(i=0; i<N; i++)
        A[i][k] = B[i][k];
```

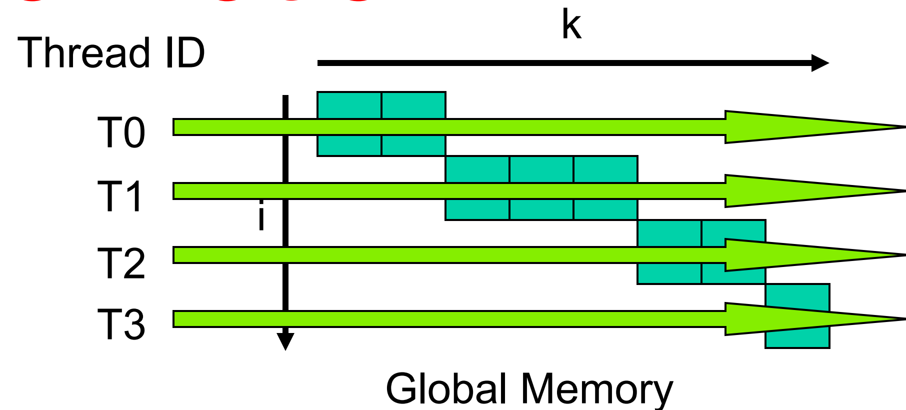
Optimized OpenMP code



Loop Collapsing Transformation

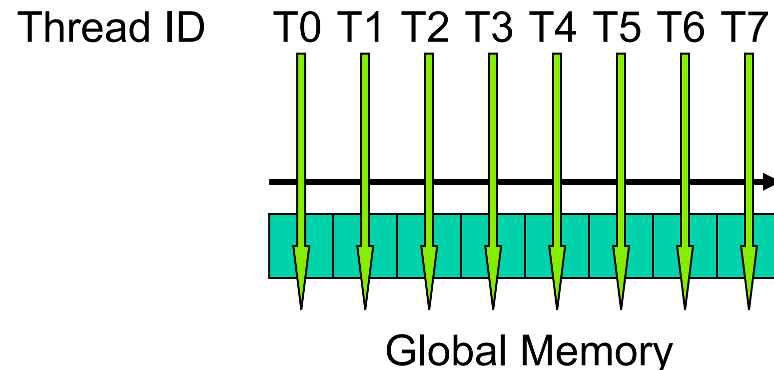
```
#pragma omp parallel for
for(i=0; i<n_rows; i++)
    for(k=rptr[i]; k<rptr[i+1]; k++)
        w[i] += A[k]*p[col[k]];
```

Input OpenMP code



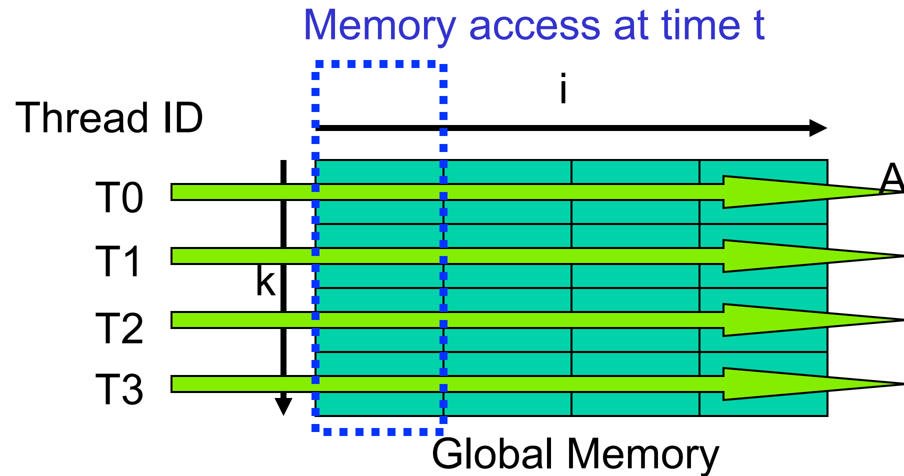

```
#pragma omp parallel
#pragma omp for collapse(2)
    schedule(static, 1)
for(i=0; i<n_rows; i++)
    for(k=rptr[i]; k<rptr[i+1]; k++)
        w[i] += A[k]*p[col[k]];
```

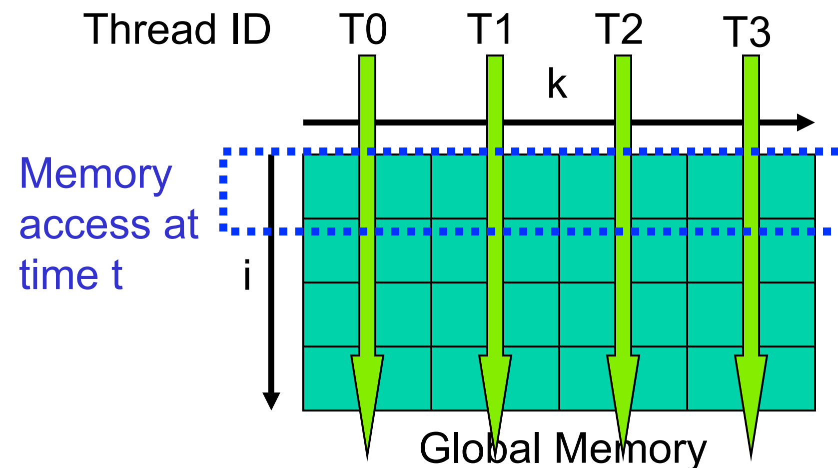
Optimized OpenMP code



Matrix-Transpose Transformation

```
float d[N][M]
...
<transpose d on transfer to GPU>
#kernel function:
float d[M][N]
# pragma omp parallel
for(k=0; i< N; i++)
    for(i=0; k<M; k++)
        …d[i,k] …];
```





Techniques to Exploit GPU On-chip Memories

Caching Strategies

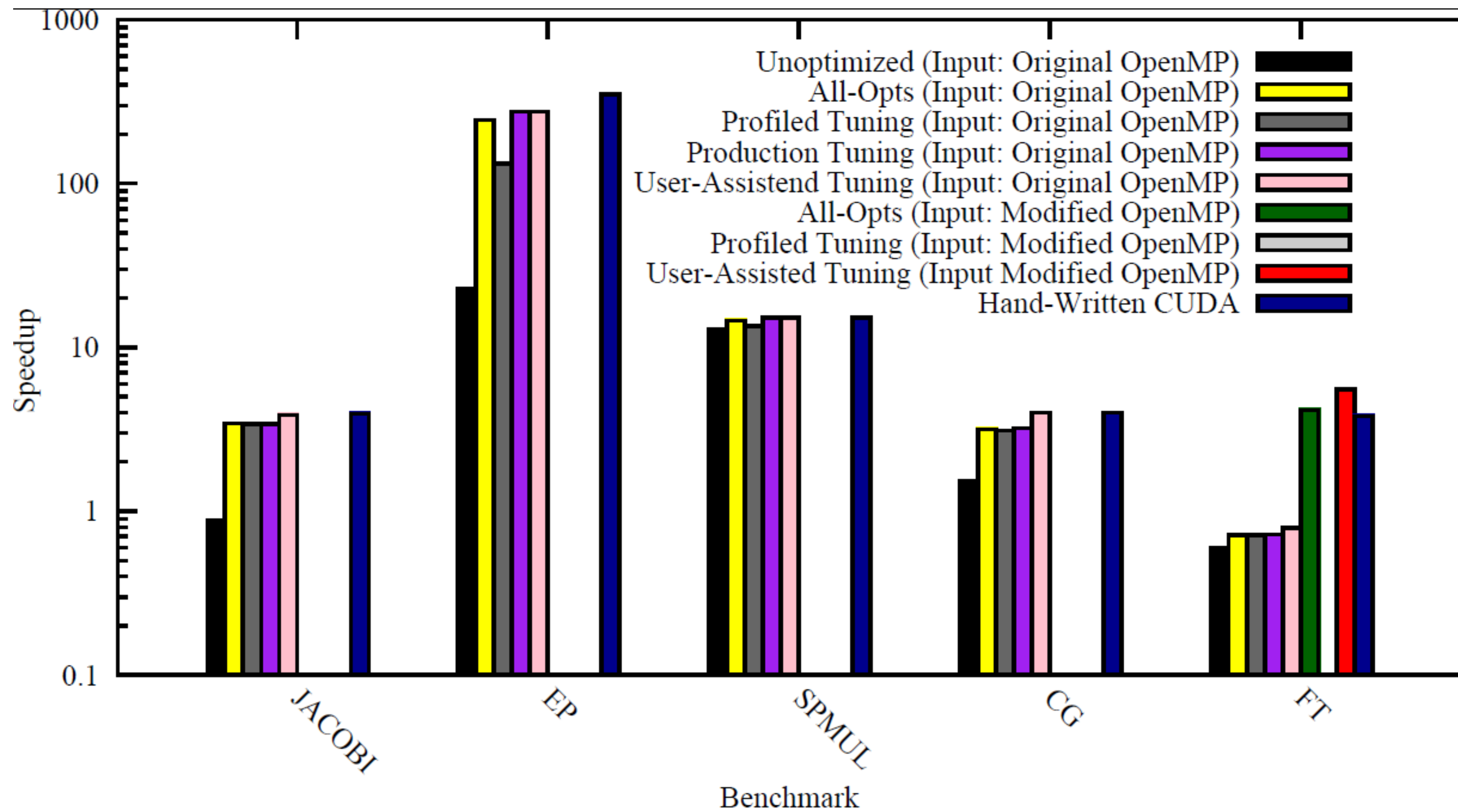
Variable Type	Caching Strategy
R/O shared scalar w/o locality	SM
R/O shared scalar w/ locality	SM, CM, Reg
R/W shared scalar w/ locality	Reg, SM
R/W shared array element w/ locality	Reg
R/O 1-dimensional shared array w/ locality	TM
R/W private array w/ locality	SM

Reg: Registers
CM: Constant Memory
SM: Shared Memory
TM: Texture Memory

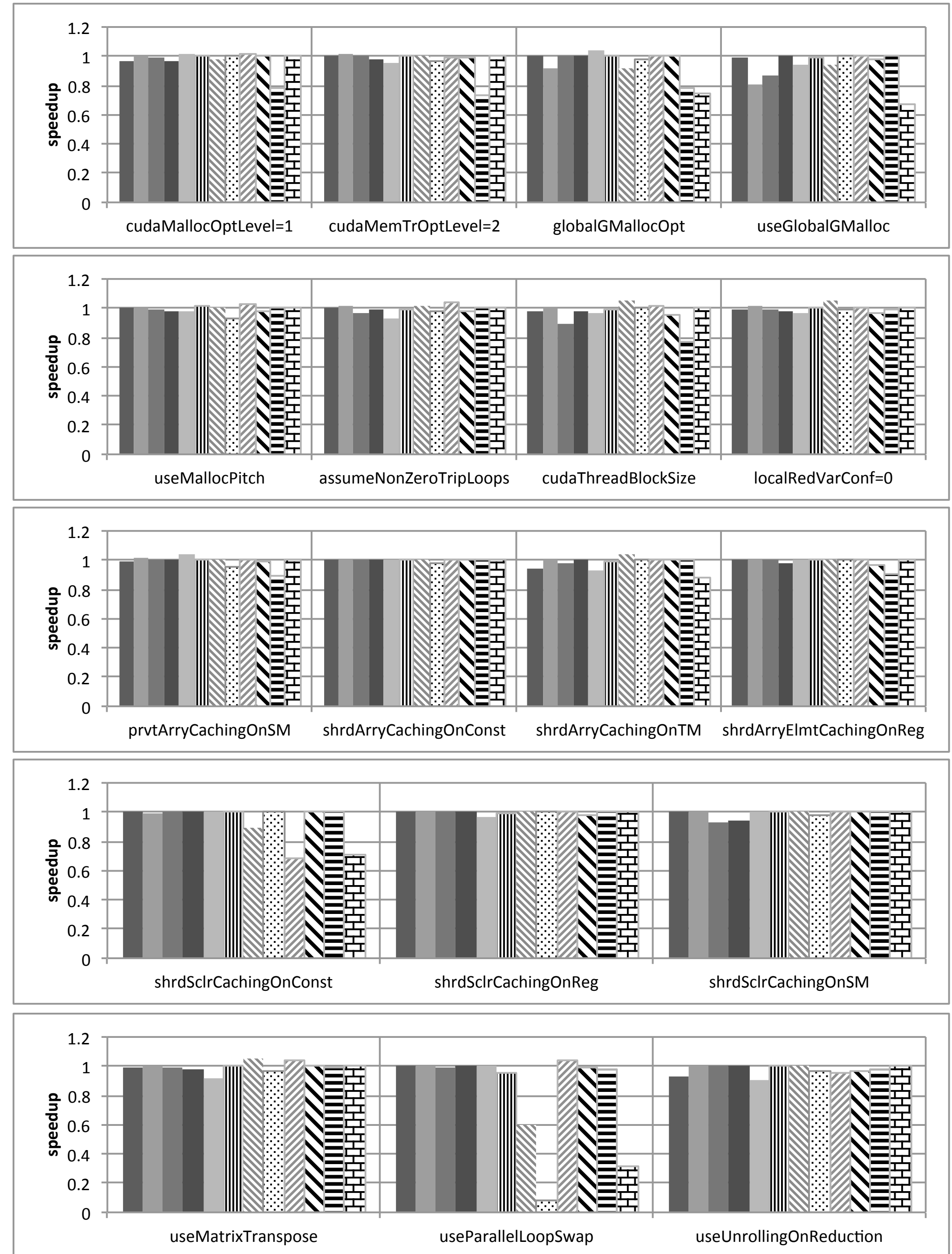
Techniques to Optimize Data Movement between CPU and GPU

- Resident GPU Variable Analysis
  - Up-to-date data in GPU global memory: do not copy again from CPU.
- Live CPU Variable Analysis
  - After a kernel finishes: only transfer live CPU variables from GPU to CPU.
- Memory Transfer Promotion Optimization
  - Find optimal points to insert necessary memory transfers

GPGPU Performance Relative to CPU

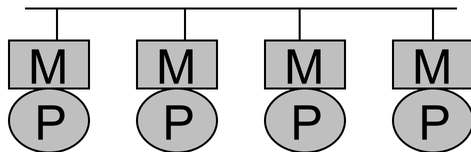


Importance of Individual Optimizations



IV.8 Techniques Specific to Distributed-memory Machines

Execution Scheme on a Distributed-Memory Machine



Typical execution scheme:

- All nodes execute the same program
- Program uses *node_id* to select the subcomputation to execute on each participating processor and the data to access.

For example,

```
DO i=1,n
. . .
ENDDO
```

→

```
mystrip=⌈n/max_nodes⌉
lb = node_id*mystrip +1
ub = min(lb+mystrip-1,n)
DO i=lb,ub
   . . .
ENDDO
```

how to place and access data ?

how/when to synchronize ?

This is called Single-Program-Multiple-Data (SPMD) execution scheme

Data Placement

Single owner:

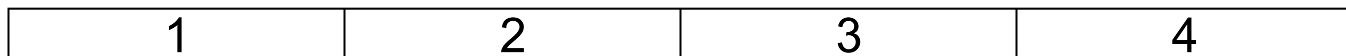
- Data is distributed onto the participating processors' memories

Replication:

- Multiple versions of the data are placed on some or all nodes.

Data Distribution Schemes

numbers indicate the node of a 4-processor distributed-memory machine on which the array section is placed

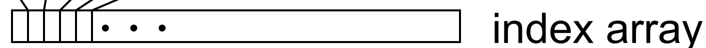


block distribution

cyclic distribution

block-cyclic distribution

indexed distribution



index array

Automatic data distribution is difficult because it is a global optimization.

Message Generation for single-owner placement

EXAMPLE

message generation

```
DO i=1,n
  B(i) = A(i)+A(i-1)
ENDDO
```

→

```
send (A(ub),my_proc+1)
receive (A(lb-1),my_proc-1)
DO i=lb,ub
  B(i) = A(i)+A(i-1)
ENDDO
```

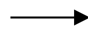
- lb,ub determine the iterations assigned to each processor.
- data uses block distribution and matches the iteration distribution
- my_proc is the current processor number

Compilers for languages such as HPF (High-Performance Fortran) have explored these ideas extensively

Owner-computes Scheme

In general, the elements accessed by a processor are different from the elements owned by this processor as defined by the data distribution

```
DO i=1,n
 A(i)=B(i)+B(i-m)
 C(ind(i))=D(ind2(i))
ENDDO
```



```
DO i=1,n
   send/receive what's necessary
   IF I_own(A(i)) THEN
      A(i) = B(i)+B(i-m)
   ENDIF
   send/receive what's necessary
   IF I_own(C(ind(i)) THEN
      C(ind(i))=D(ind2(i))
   ENDIF
ENDDO
```

- nodes execute those iterations and statements whose LHS they own
- first they receive needed RHS elements from remote nodes
- nodes need to send all elements needed by other nodes

Example shows basic idea only. Compiler optimizations needed!

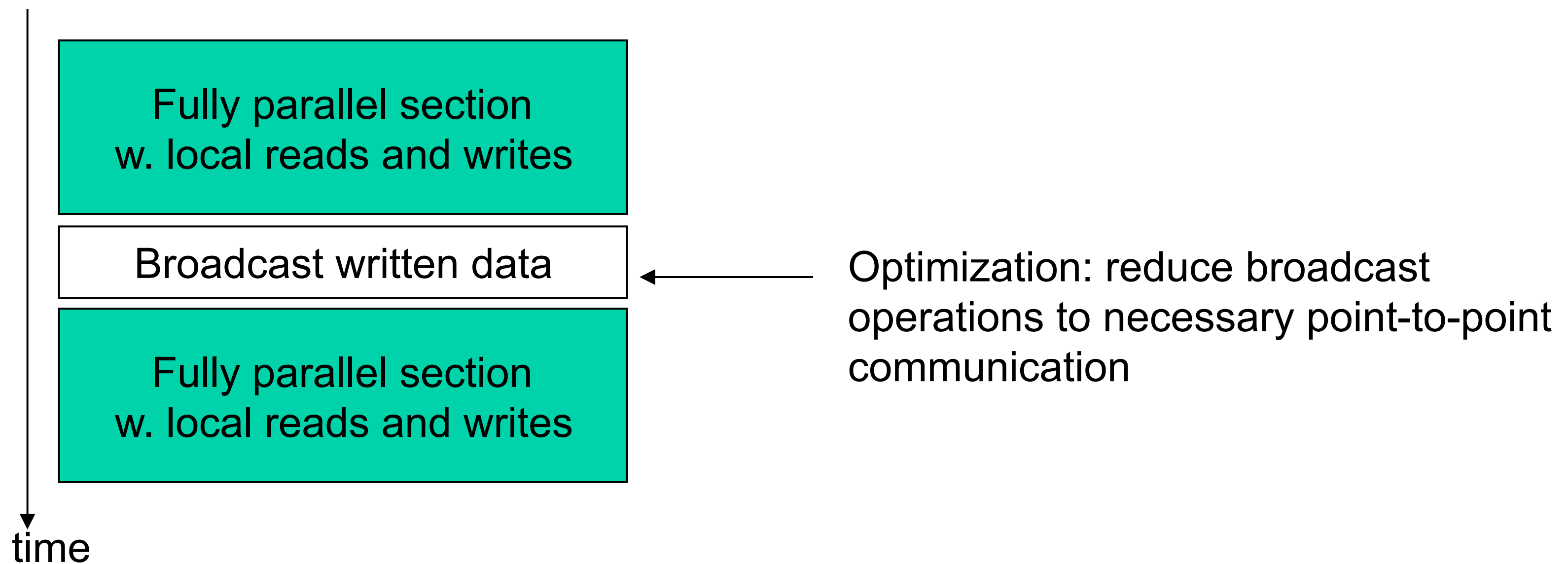
Compiler Optimizations

for the raw owner computes scheme

- Eliminate conditional execution
  - combine if statements with same condition
  - reduce iteration space if possible
- Aggregate communication
  - combine small messages into larger ones
  - tradeoff: delaying a message enables message aggregation but increases the message latency.
- Message Prefetch
  - moving *send* operations earlier in order to reduce message latencies.

there is a large number of research papers describing such techniques

Message Generation for Virtual Data Replication



Advantages:
- Fully parallel sections with local reads and writes
- Easier message set computation (no partitioning per processor needed)

Disadvantages:
- Not data-scalable
- More write operations necessary (but, collective communication can be used)

7 Techniques for Instruction-Level Parallelization

Implicit vs. Explicit ILP

Implicit ILP: ISA is the same as for sequential programs.

- most processors today employ a certain degree of implicit ILP
- parallelism detection is entirely done by the hardware
- compiler can assist ILP by arranging the code so that the detection gets easier.

Implicit vs. Explicit ILP

Explicit ILP: ISA expresses parallelism.

- parallelism is detected by the compiler
- parallelism is expressed in the form of
  - VLIW (very long instruction words): packing several instructions into one long word
  - EPIC (Explicitly Parallel Instruction Computing): bundles of (up to three) instructions are issued. Dependence bits can be specified.

    Used in Intel/HP IA-64 architecture. The processor also supports predication, early (speculative) loads, prepare-to-branch, rotating registers.

Trace Scheduling

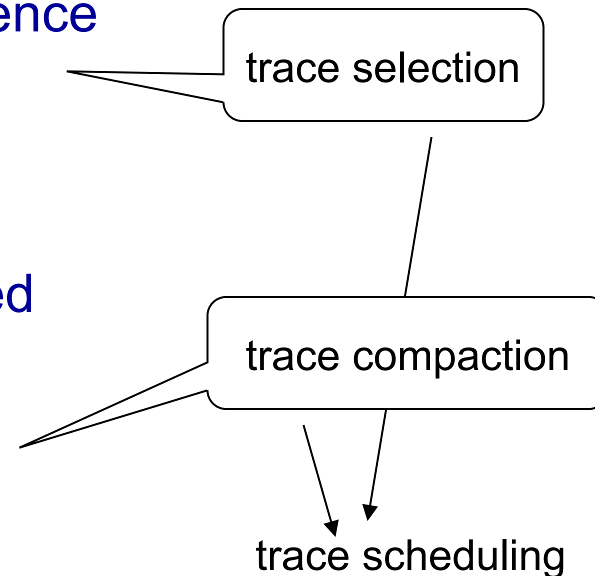
(invented for VLIW processors, still a useful terminology)

Two big issues must be solved by all approaches:

1. Identifying the instruction sequence that will be inspected for ILP.

   Main obstacle: branches

2. reordering instructions so that machine resources are exploited efficiently.



Trace Selection

- It is important to have a large instruction window (block) within which the compiler can find parallelism.
- Branches are the problem. Instruction pipelines have to be flushed/squashed at branches
- Possible remedies:
  - eliminate branches
  - code motion can increase block size
  - block can contain out-branches with low probability
  - predicated execution

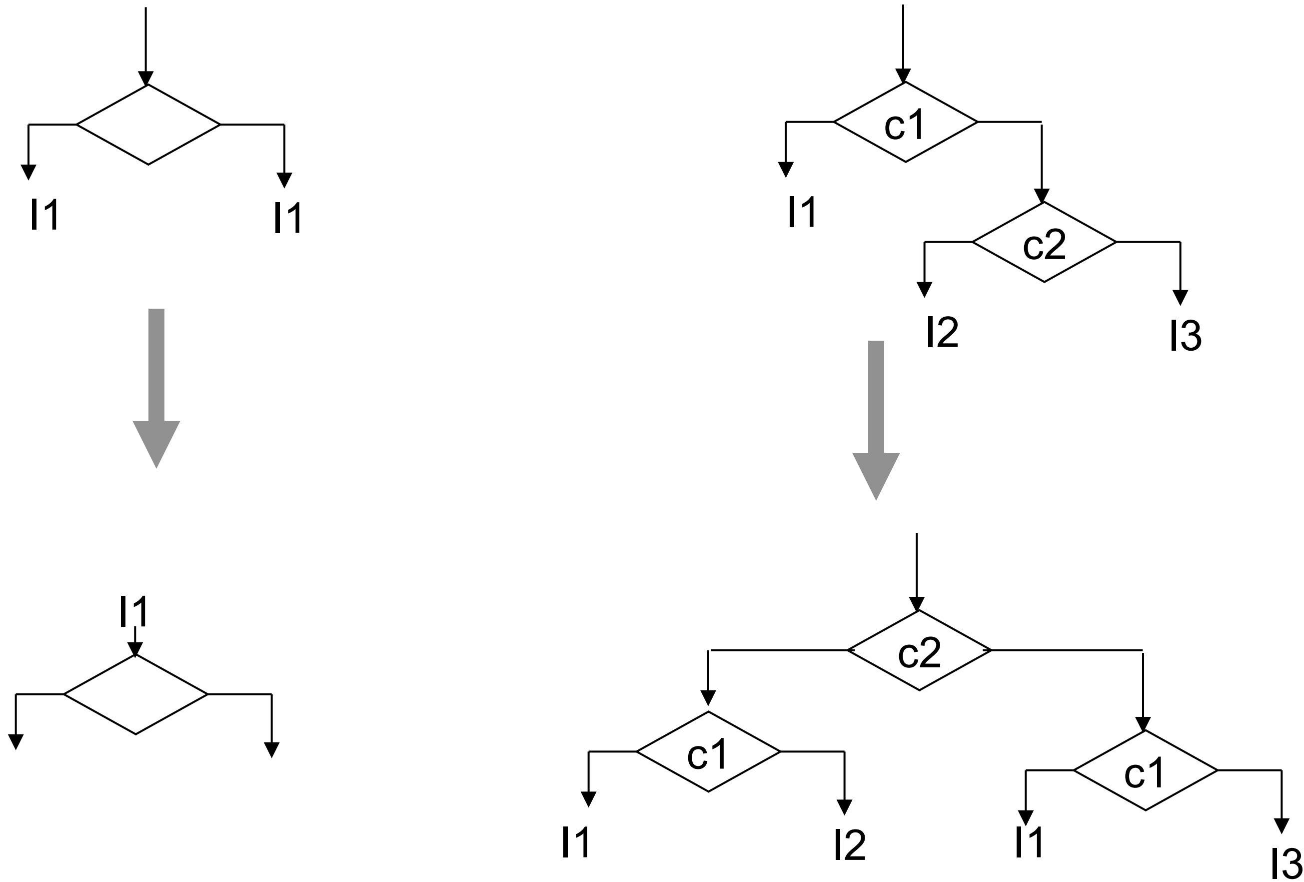
Branch Elimination

- Example:

```
     comp R0 R1
     bne  L1:
```
```
     bra  L2:
```
```
L1:  . . .
     . . .

L2:  . . .
```

```
     comp R0 R1
     beq  L2:
```
```
L1:  . . .
     . . .

L2:  . . .
```

Code Motion



Code motion can increase window sizes and eliminate subtrees

Predicated Execution

```
IF (a>0) THEN
  b=a
ELSE
  b=-a
ENDIF
```

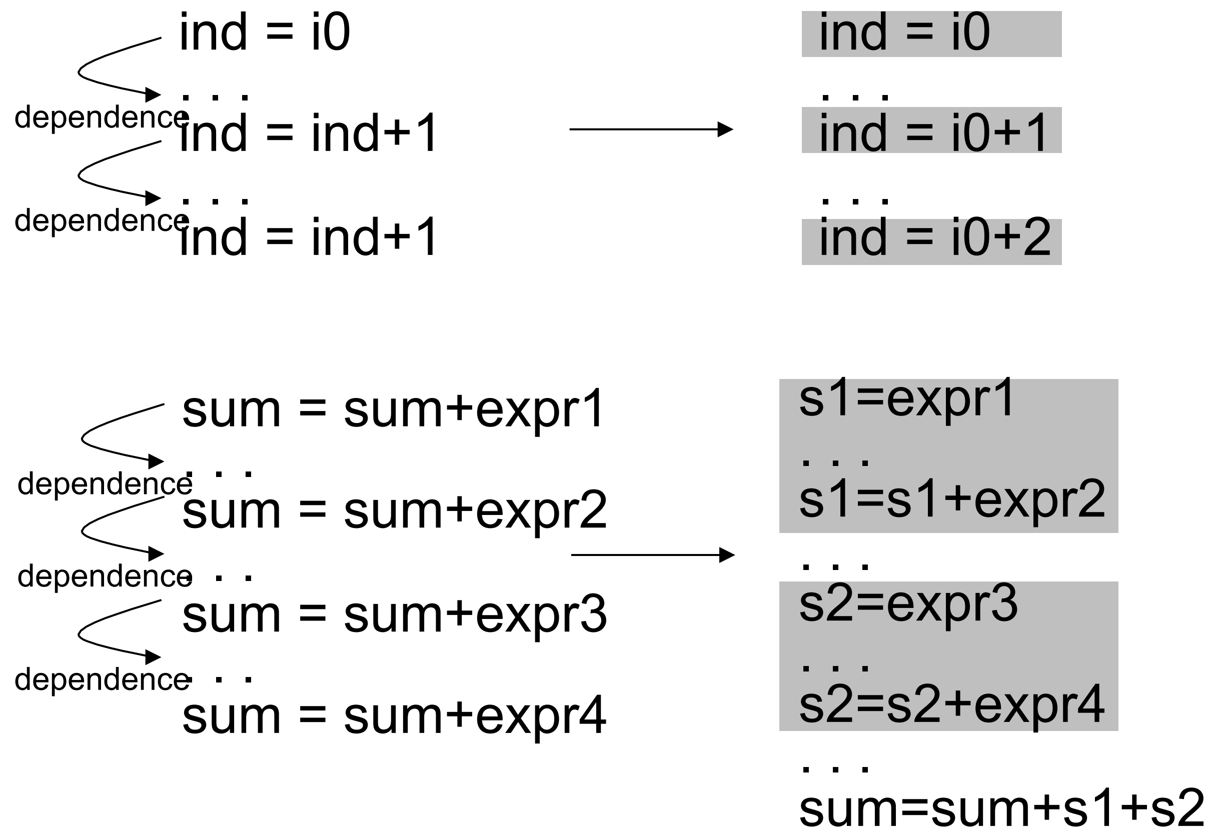
p = a>0 *; assignment of predicate*
p: b=a *; executed if predicate true*
$\bar{p}$: b=-a *; executed if predicate false*

Predication
- increases the window size for analyzing and exploiting parallelism
- increases the number of instructions “executed”

These are opposite demands!

Compare this technique to conditional vectorization

Dependence-removing ILP Techniques



shaded blocks of statements are independent of each other and can be executed as parallel instructions

Speculative ILP

Speculation is performed by the architecture in various forms

- Superscalar processors: compiler only has to deal with the performance model. ISA is the same as for non-speculative processors
- Multiscalar processors: (research only) compiler defines tasks that the hardware can try execute speculatively in parallel. Other than task boundaries, the ISA is the same.

References:

- **Task Selection for a Multiscalar Processor**, T. N. Vijaykumar and Gurindar S. Sohi, The 31st International Symposium on Microarchitecture (MICRO-31), pp. 81-92, December 1998.
- **Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution,** Seon-Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T.N. Vijaykumar,, In Proc. of PPOPP'01, Symposium on Principles and Practice of Parallel Programming, 2001.

Compiler Model of Explicit Specluative Parallel Execution
(Multicalar Processor)

- **Overall Execution:** *speculative threads* choose and start the execution of any predicted next thread.
- **Data Dependence and Control Flow Violations** lead to roll-backs.
- **Final Execution**: satisfies all cross-segment flow and control dependences.
- **Data Access**: Writes go to thread-private speculative storage. Reads read from ancestor thread or memory.
- **Dependence Tracking**: Data Flow and Control Flow dependences are detected directly. Lead to roll-back. Anti and Output dependences are satisfied via speculative storage.
- **Segment Commit**: Correctly executed threads (I.e., their final execution) commit their speculative storage to the memory, in sequential order.