Parallelizing Compilers for Multicores

Course Offered at the Universitat Politècnica de Catalunya

Rudi (Rudolf) Eigenmann

Purdue University

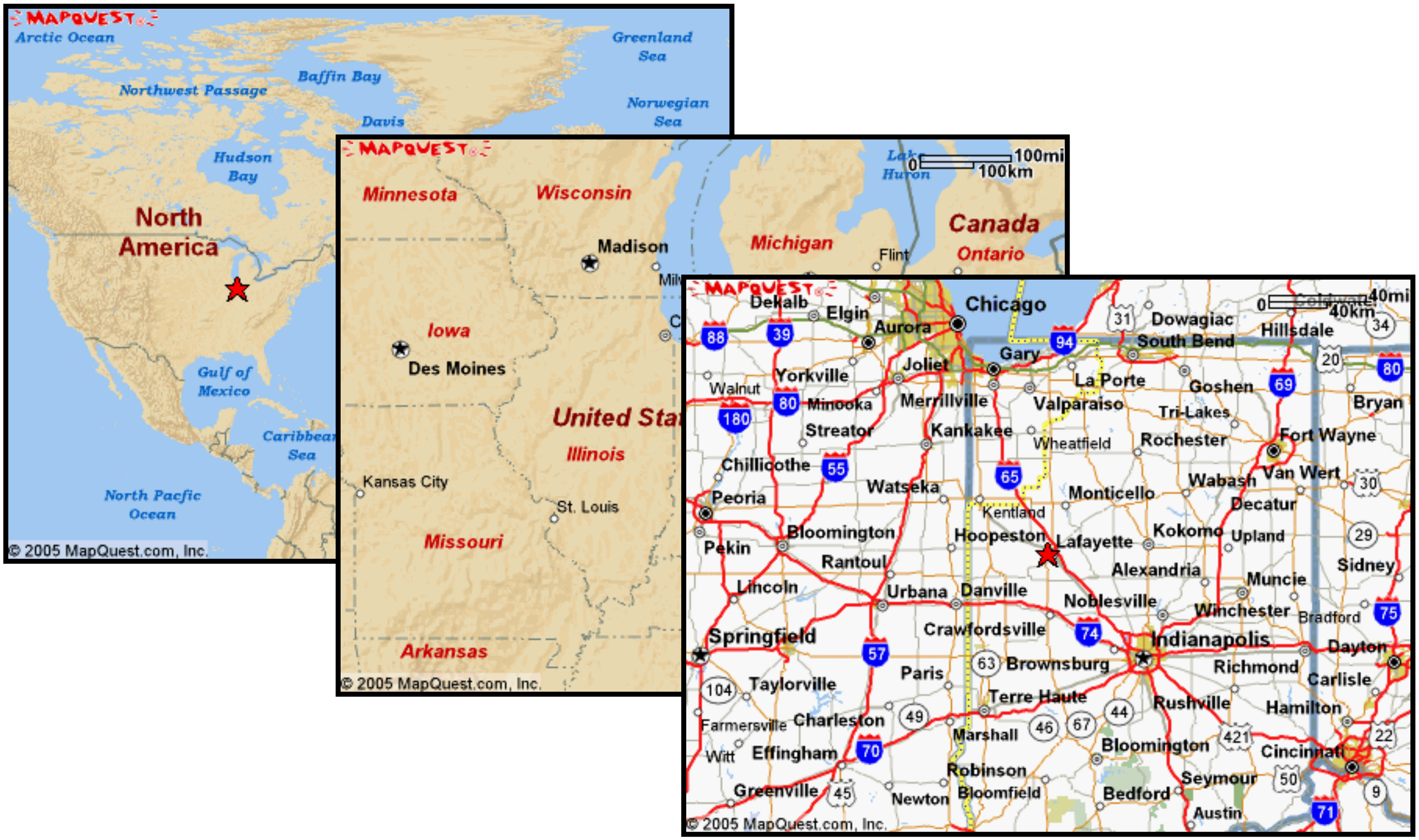
School of Electrical and Computer Engineering

Computing Research Institute

Summer 2010

www.ece.purdue.edu/~eigenman/app/

How to get to Purdue University



Course Schedule

Parallelizing Compilers for Multicore

During the period 7th June– 18th June

June 7th - 11th, 10:00 - 13:00 C6-E101
June 14th - 18th, 10:00 - 13:00 C6-E101

Grading will be based on several in-class exercises and class interaction

Office hours: by appointment eigenman@purdue.edu

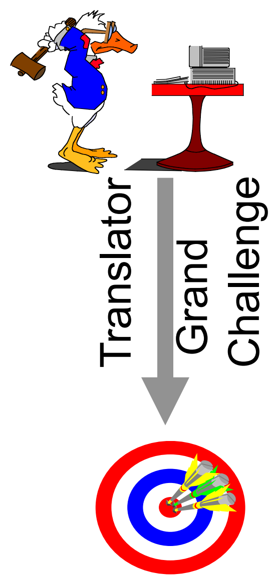
Course Content

- Introduction and motivation
- Detecting parallelism
- Mapping parallelism to the machine

Optimizing Compilers are the Center of the Universe

Translate increasingly advanced human interfaces onto increasingly sophisticated target machines

Today	Tomorrow
Fortran C, Java	Problem Specification Language
↓	↓
Workstation Multicores NOW	Globally Distributed/Parallel Resources



Optimizing compilers are of particular importance where performance matters most. Hence our focus on High-Performance Computing.

Issues in Optimizing / Parallelizing Compilers

The Goal:

- We would like to run standard (C, Java, Fortran) programs on 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 memory machines (clusters or global networks)
  - Heterogeneous architectures
  - Instruction-level parallelism
  - Vector machines
- Locality enhancement

Parallelizing Compiler Books and Survey Papers

Books:

- Ken Kennedy, John Allen: Optimizing Compilers for Modern Architectures: A Dependence-based Approach (2001)
- Michael Wolfe: High-Performance Compilers for Parallel Computing (1996)
- Utpal Banerjee: several books on Data Dependence Analysis and Transformations

Survey Papers:

- *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*

Initial course schedule:

- Blume study - the simple techniques
- The Cedar Fortran Experiments
- Analysis and Transformation techniques in the Cetus compiler
- Additional transformations (for GPGPUs and other architectures)

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 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 data references iff:

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

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 any data dependences, and
- dependence-free code could be executed by today's multicores directly and efficiently.

There are enough hard problems to fill several courses!

Part I: Performance of Basic Automatic Program Parallelization

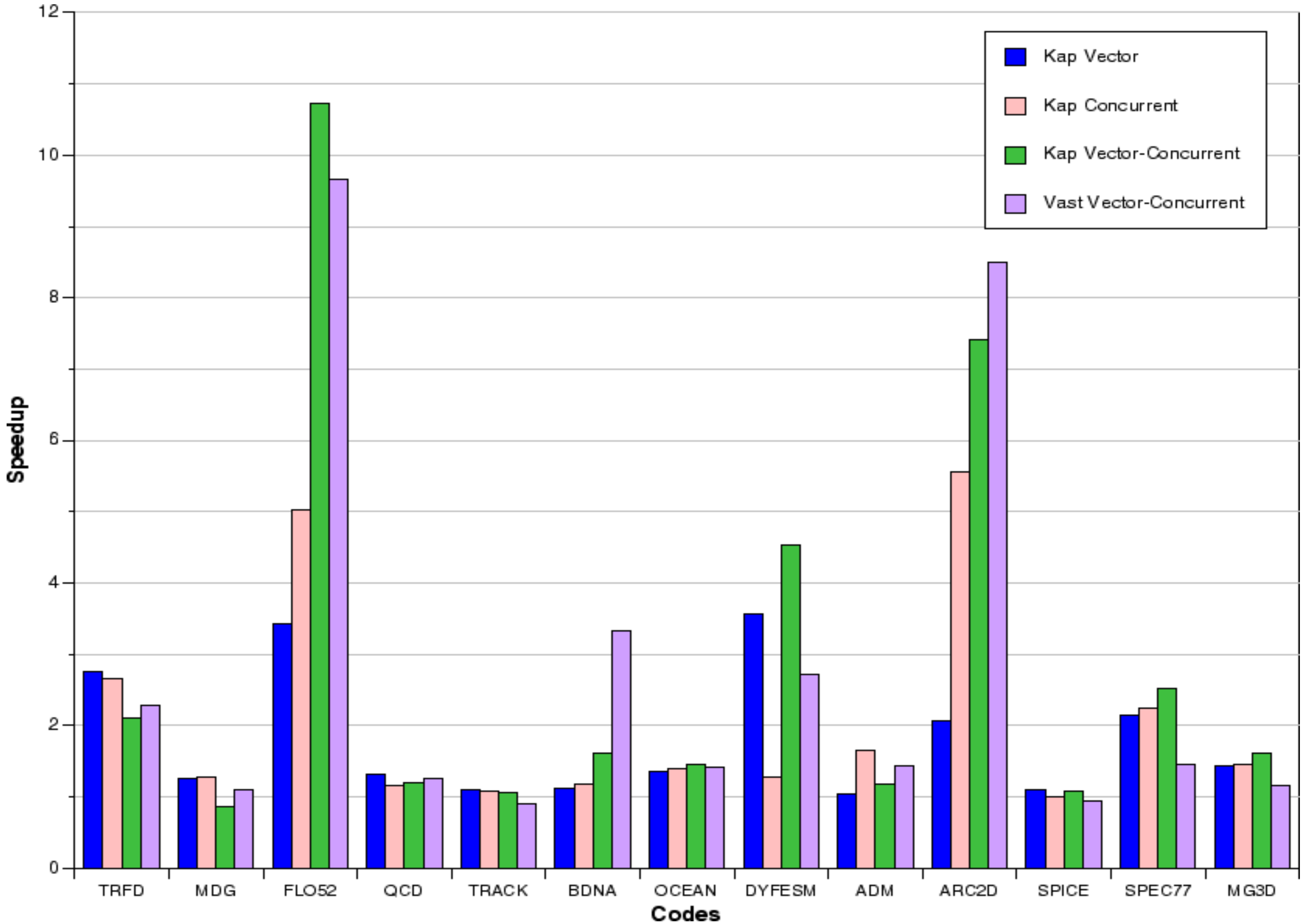
15 Years of Parallelizing Compilers

A Performance study at the beginning of the 90es (Blume study)

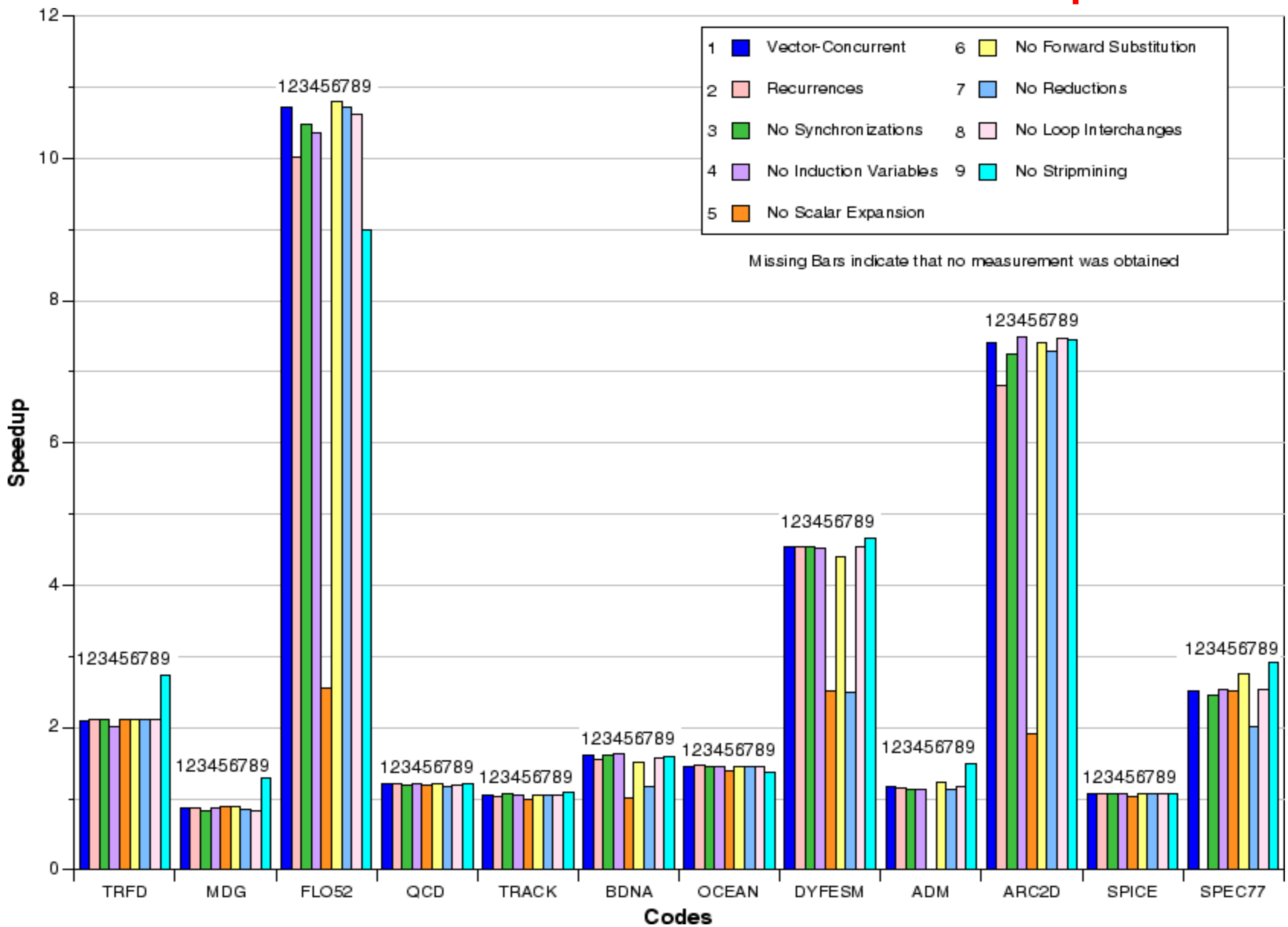
Analyzed the performance of state-of-the-art parallelizers and vectorizers using the Perfect Benchmarks.

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.

Overall Performance



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

DO j=1,n
    t = a(j)+b(j)
    c(j) = t + t^2
ENDDO

output
flow
anti

Privatization

DO PARALLEL j=1,n
PRIVATE t
    t = a(j)+b(j)
    c(j) = t + t^2
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
- processor share the work of parallel loops

Parallel Loop Syntax and Semantics

OpenMP:

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

  <loop body code>

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

executed by all participating processors (threads) exactly once

work (iterations) shared by participating processors (threads)

Reduction Parallelization

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

```
!$OMP PARALLEL, PRIVATE (s)
s = 0
!$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

```
ind = ind0
DO j = 1,n
   a(ind) = b(j)
   ind = ind+k
ENDDO
```

→

```
ind = ind0
DO PARALLEL j = 1,n
    a(ind0+k*(j-1)) = b(j)
ENDDO
```

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

```
real d(20,100)
DO j=1,n
  d(1,j)=0
ENDDO
```

→

```
loop:
 ...
R0   ← &d+20*j
(R0) ← 0
...
jump loop
```

→

```
R0 ← &d
loop:
 ...
(R0) ← 0
...
R0   ← R0+20
jump loop
```

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
```

```
a = x*y
b = a+2
c = b + 4
```

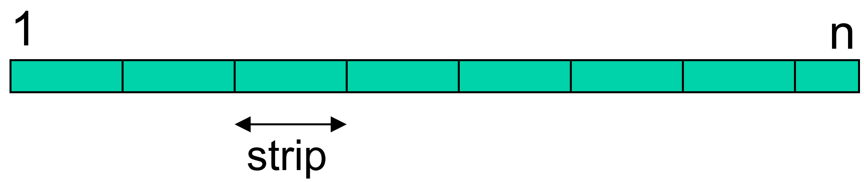
dependences

→

```
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 j=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
0	$\Delta a(10)$	$\Delta a(10)+\Delta a(20)$	$\Delta a(10)+\Delta a(20)+\Delta a(30)$

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

Loop Interchanging

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

→

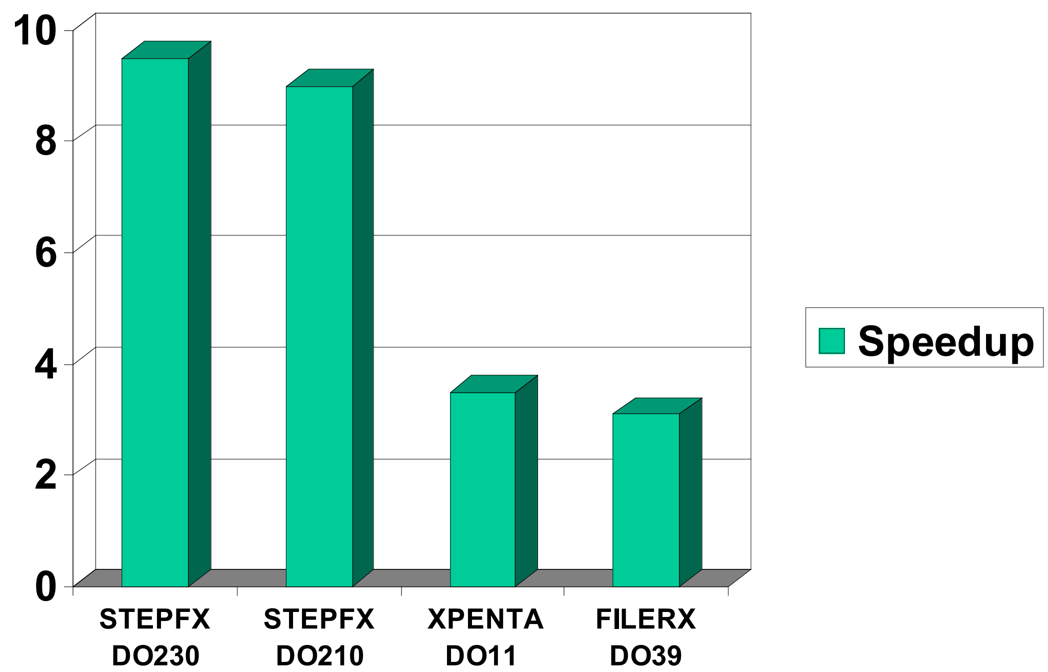
```
DO j= 1,m
  DO i=1,n
    a(i,j) =a(i,j)+a(i,j-1)
  ENDDO
ENDDO
```

- 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
- impact of cache organization ?
- parallelism at outer position reduces loop fork/join overhead

Effect of Loop Interchanging

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

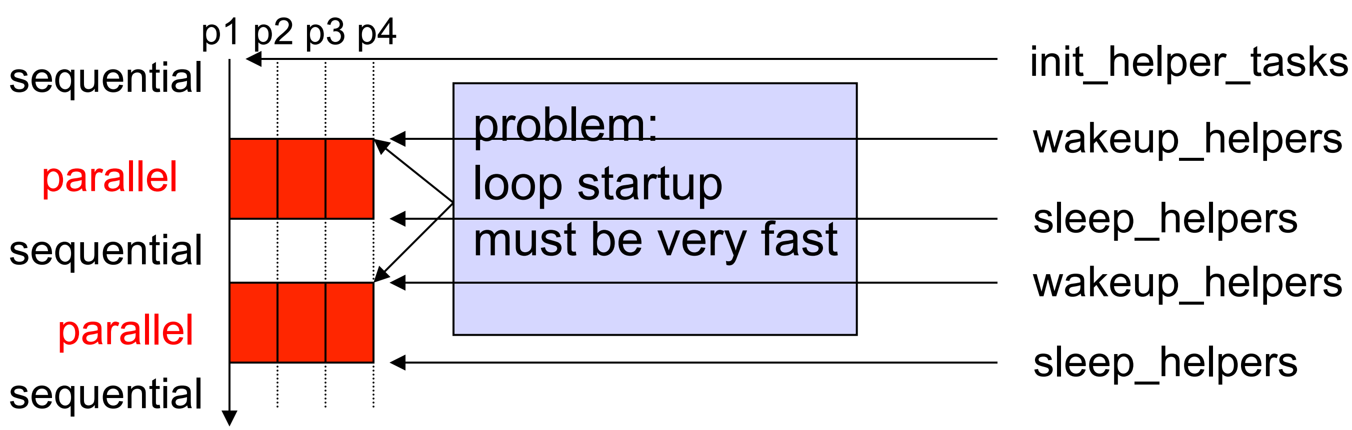
```
a=0
DO i=1,n
   b(i) = 2
ENDDO
b=3
```

```
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

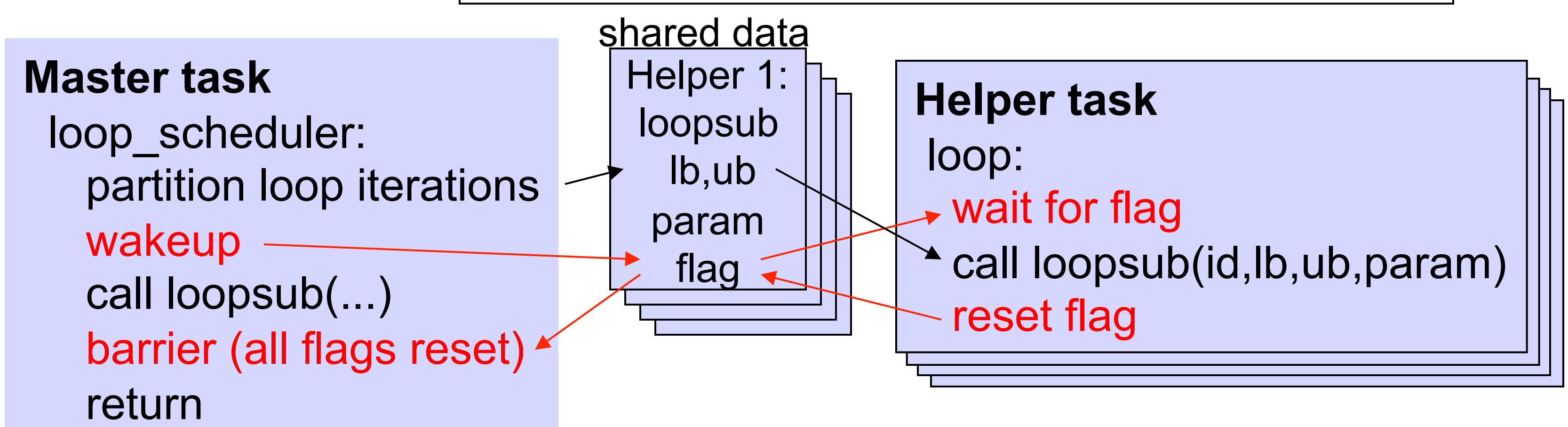
Compiler Transformation for the Microtasking Scheme

```
a=0
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
```



Performance of Parallelization Techniques

Rudolf Eigenmann, Jay Hoeflinger, and David Padua**, On the AutomaticParallelization of the Perfect Benchmarks**. *IEEE Transactions on Parallel and Distributed Systems*, volume 9, number 1, January 1997, pages 5-23.

Compiler Evaluation (1990)

Study	Test Suite			Measures						Machines	Compilers
	K	A	P	V	N	T	S	I	F		
[56]		x					x			simulated	
[149]	x			x	x	x	x			Cyber 203/5	FTN200, KAP, VAST
[158]		x					x			simulated	Parafrase
[155]		x					x	x		simulated	Parafrase
[144, 143]	x			x		x				see Table 1	
[150]	x			x	x	x				Cyber 205	FTN200, KAP, VAST
[145]	x			x						see Table 1	
[151]	x					x				NAS 160	KAP, VAST
[148]	x			x	x		x			see Table 2	
[153]			x		x	x	x	x		Alliant FX/8	KAP, VAST
[152]		x	x		x		x			Cray Y-MP	KAP, fpp
[156]		x	x						x	Alliant FX/8	VAST
[101, 154]		x	x				x		x	FX/8, Cedar	KAP
[157]		x	x						x	simulated	KAP

test suite:	K=Kernels A=Algorithms P=Application programs
measures :	V=shows rate of successfully vectorized loops
	N=compares performance numbers of different compilers
	T=compares transformations of different compilers
	S=shows speedups due to automatic parallelization
	I=evaluates individual compiler techniques
	F=discusses future compiler improvements

Table 5: Summary of compiler effectiveness studies

Compiler Evaluation (1990)

	ADM	*ARC2D*	*BDNA*	*DYFESM*	*FLO52*	*MDG*	*MG3D*	*OCEAN*	*QCD*	*SPEC77*	*SPICE*	*TRACK*	*TRFD*
vectorized (Y-MP, 1 CPU)	1.2	1	1	1	1	1	0.9	1.2	1	1	1	1	1
vector-concurrent (Y-MP, 8 CPUs)	1	3.1	1	1.2	2.5	1	0.9	1.1	1	1.2	1	1	1
vectorized (FX/8, 1 CPU)	1.1	2.0	1.1	3.6	3.4	1.2	2.3	1.3	1.2	2.2	1.1	1.1	2.8
vector-concurrent (FX/8, 8 CPUs)	1.3	8.0	3.3	4.3	10.2	1.1	1.6	1.3	1.2	2.3	1.1	1.0	2.2
manually improved (FX/8, 8 CPUs)	7.5		4.2	7.7	16	5.5	4.4	8.3	7.0	5.5		5.1	14.3

Table 4: Performance improvements of the Perfect Benchmarks. First two lines: Improvement over manually vector-optimized programs on Cray Y-MP [152]. Third and fourth line: Improvements over serial program execution on Alliant FX8 [153]. Fifth line: manual improvements over serial program execution on Alliant FX8 [154]

Improving Compiler-Parallelized Code (1995)

- beyond basic techniques -

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	[a]		3.3	2.1	1.1	21	15.2			3.4		[b]
substitute generalized induction variables								8.3				12.7
parallelize loops with non-affine array subscripts				3				11.5				13

Effect of Array Privatization

PROGRAM-subroutine/loop	total loop execution time in seconds T_{best}	$T_{without}$	$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
ARC2D-filerx/15	7.3	22.0	3.0	1.1
ARC2D-filery/39	3.4	12.0	3.5	1.06
ADM-dudtz/40	3.8	92.5	24	2.1
ADM-dvdtz/40	3.5	76.5	21	1.9
ADM-dtdtz/40	3.8	78.8	21	1.9
ADM-dcdtz/40	2.6	51.7	20	1.6
ADM-dkzmh/30	2.5	37.6	15	1.4
ADM-dkzmh/60	3.8	86.8	23	2.0
ADM-run/20	4.4	72.2	16.5	1.8
ADM-run/30	4.4	72.0	16.4	1.8
ADM-run/40	4.2	72.1	17	1.8
ADM-run/50	3.4	50.8	15	1.6
ADM-run/60	4.4	72.0	16.4	1.8
ADM-run/100	3.2	50.0	15.6	1.5
ADM-wcont/40	2.7	36.2	13.4	1.4
ADM-smooth/10	1.4	18.5	13.2	1.2
BDNA-actfor/240	19.0	62.0	3.3	1.4
DYFESM-mxmult/10	19.0	60.0	3.2	1.7
DYFESM-solvh/20	11.5	26.5	2.3	1.3
MDG-interf/1000	163.0	3792.0	23.2	19
MDG-poteng/2000	13.4	352.0	26.3	2.7
MDG-intraf/1000	1.9	11.4	6.0	1.05
MG3D-migrat/200	264.0	5226.4	19.8	19.7
OCEAN-acac/30	3.3	92.5	28	1.5
OCEAN-ocean/60	0.3	0.05	5.6	0.9
OCEAN-ocean/270	1.3	16.2	12.5	1.0
OCEAN-ocean/340	2.9	30.7	10.6	1.1
OCEAN-ocean/360	2.7	25.6	9.5	1.1
OCEAN-ocean/400	2.3	20.9	9.1	1.1
OCEAN-ocean/420	2.3	25.6	11.1	1.1
OCEAN-ocean/440	2.6	24.5	9.5	1.1
OCEAN-ocean/460	7.7	103.7	13.5	1.5
OCEAN-ocean/480	4.1	91.4	22.3	1.4
OCEAN-ocean/500	2.2	48.0	21.8	1.2
OCEAN-scsc/40	2.5	48.0	19.2	1.2
QCD-measur/3	1.8	2.9	1.6	1.0
QCD-qqqmea/1	3.7	108.6	29.4	6.2
QCD-rotmea/2	3.7	48.4	13.1	3.2
SPEC77-gloop/1000	58.7	743.7	12.7	5.5
SPEC77-gwater/1000	13.5	248.0	18.4	2.6
TRACK-extend/400	4.0	48.9	12.2	2.7
TRACK-fptrack/300	3.0	15.5	5.2	1.4
TRACK-nlfilt/300	3.5	76.7	22	3.8
TRFD-olda/100	8.0	(174)	21.8	9.3
TRFD-olda/300	5.4	(85)	15.7	5

Effect of Advanced Parallel Reductions

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
MDG-interf/1000	163	3792.0	23.2	19
MDG-poteng/2000	13.4	352.0	26.3	2.6
DYFESM-mxmult/10	19.0	60.0	3.2	1.7
DYFESM-formr0/20	7.0	20.0	2.8	1.2
BDNA-actfor/240	19.0	62.0	3.3	1.4
BDNA-actfor/500	21.0	253.0	12.0	3
FLO52-euler/70	0.5	5.7	11.4	1.1
MG3D-migrat/200	264.0	5226	19.8	19.7
SPEC77-gloop/1000	58.7	743.7	12.7	5.5
SPEC77-gwater/1000	13.5	248.0	18.4	2.6

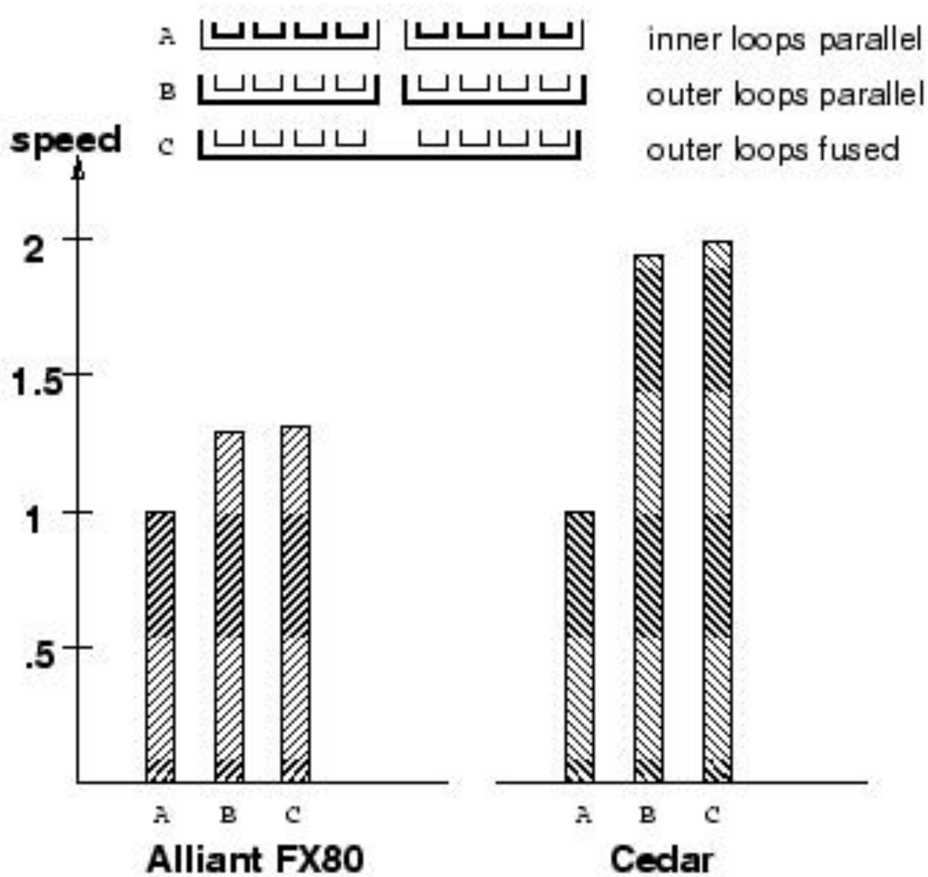
Effect of Generalized Induction Variable Substitution

PROGRAM-subroutine/loop	total loop execution time in seconds T_{best}	$T_{without}$	$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
OCEAN-ftrvmt/109	89	1377	15.5	8.3
TRFD-olda/100	8.0	174.2	21.8	9.3
TRFD-olda/300	5.4	85.3	15.8	5.0

Effect of Balanced Stripmining

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
FLO52-eflux/10	3.9	5.9	1.5 (2D)	1.03
FLO52-eflux/20	1.1	3.0	2.7 (1D)	1.03
FLO52-eflux/30	3.9	5.7	1.5 (1D)	1.04
FLO52-dflux/30	3.2	5.5	1.7 (1D)	1.03
FLO52-dflux/38	0.4	0.5	1.25 (2D)	1.00
FLO52-bcwall/30	1.7	2.8	1.6 (MV)	1.02

Effect of Increasing Parallel Loop Granularity



Effect of Locality Enhancement

PROGRAM-subroutine/loop	access	total loop execution time in seconds T_{best}	$T_{without}$	$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
FLO52-psmoo/40&80	r/w	9.9	19.8	2.0	1.17
FLO52-step/20	r/o	1.7	2.4	1.4	1.01
ARC2D-xpenta/11	r/o	5.8	6.7	1.15	1.01
MDG-interf/1000	r/o	163	187	1.15	1.12
TRFD-olda/100 & 300	r/o	13.4	15.91	1.18	1.13

Effect of Runtime Data-Dependence Testing

PROGRAM-subroutine/loop	total loop execution time in seconds		$T_{without}/T_{best}$ for loop	$T_{without}/T_{best}$ for program
	T_{best}	$T_{without}$		
OCEAN-ftrvmt/109	89.3	1376.9	15.4	7.3
OCEAN-csr/20	10.5	172.0	16.4	1.9
OCEAN-ftrvmt/116	10.8	99.5	9.2	1.5
OCEAN-acac/30	3.3	92.5	28.0	1.5
OCEAN-acac/40	4.0	79.0	19.8	1.4
OCEAN-scsc/30	2.3	59.3	25.8	1.3
OCEAN-rcs/20	3.4	57.7	17.0	1.3

Part II
A Catalog of Advanced Analysis and Transformation Techniques

- 1 Data-dependence testing
- 2 Parallelism enabling transformations
- 3 Techniques for multiprocessors/multicores
- 4 Techniques for heterogeneous multicores
- 5 Techniques for other architectures
  (vector, distributed-memory,…)

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
$lower \leq i_1, i_2 \leq upper$?

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 Graph: a graph showing statements as nodes and dependences between them as edges. For loops, usually there is only one node per statement instance.
- Iteration Space Graphs: the un-abstracted form of a dependence graph with one node per statement instance. The statements of one loop iteration may be represented as a single node.

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 DD Tests

- The GCD test is simple but not accurate
- Other tests
  - Banerjee test: accurate state-of-the-art 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
```

<u>ranges accesses:</u>

[1:100]

[201:300]

→ independent

Banerjee(-Wolfe) Test continued

Weakness of the test:

Consider this dependence

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

<u>ranges accessed:</u>

[1:100]

[6:105]

→ independent ?

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

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

Banerjee(-Wolfe) Test continued

- Solution idea:

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

Still considering the potential 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. 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.

$$(d_1,d_2,\ldots,d_n) \longrightarrow \begin{pmatrix} *, & *, & \ldots, & * \\ = & = & & = \\ < & < & & < \\ & > & & > \end{pmatrix}$$

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)

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
2. If the upper(lower) bound of this range is less (greater) than the lower(upper) bound of the range accesses in the next iteration, then there is no cross-iteration dependence.

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: $[i_x*m+1:\overbrace{(i_x+1)*m}^{ub_x}]$

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

$\text{DO } i_1=L_1,U_1$
...
$\text{DO } i_n=L_n,U_n$
$A(f(i_0,\ldots i_n)) = \ldots$
$\ldots \quad = A(g(i_0,\ldots i_n))$
ENDDO
...
ENDDO

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

we need range analysis

Determining monotonicity: consider $d = f(\ldots,i_k,\ldots) - f(\ldots,i_k-1,\ldots)$
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

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.

Range Test : 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 $\neq$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.

2 Parallelism Enabling Techniques

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 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 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

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

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

Computing Closed Form, Substituting additive GIVs

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
```

Main:
totalinc = FindIncrement(L_0)
Replace(L_0,iv)
InsertStatement("iv = iv+totalinc")

Insert this statement If iv is live-out

For coupled GIVs: begin with independent iv.

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

FindIncrement(L)
  inc=0
  foreach s_i of type I,L
    if type(s_i)=I   inc += exp
    else /* L */   inc+= FindIncrement(s_i)
    inc_after[s_i]=inc
  inc_into_loop[L]= $\sum_1^{j-1}$(inc) ; inc may depend
  return $\sum_1^{ub}$(inc)   ; on j

Step 2: substitute IV

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

Induction Variables, 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)

- 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

Reduction Parallelization

Scalar Reductions

loop-carried flow dependence

```
DO i=1,n
  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
```

```
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
```

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
```

Reduction Parallelization continued

Reduction recognition and parallelization passes:

induction variable recognition
reduction recognition
privatization
data dependence test
reduction parallelization

recognizes and annotates reduction variables

for parallel loops with reduction variables, performance the reduction transformation

compiler passes

Reduction Parallelization

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

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

```
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
```

```
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

- Pattern Matching:
  - find reduction statements in a loop of the form
    $X = X \otimes expr$,

    where X is either scalar or an array expression (a[sub], where sub must be the same on the LHS and the RHS),

    $\otimes$ is a reduction operation, such as +, *, min, max
  - X must not be used in any non-reduction statement in this loop (however, there may be multiple reduction statements for X)

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.

Recurrence Substitution

loop-carried flow dependence

```
DO j=1,n
   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 continued

Basic idea of the recurrence solver:

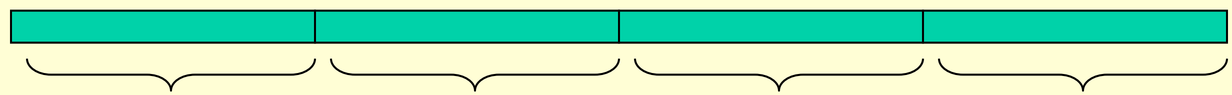
```
DO j=1,40
   a(j) = a(j) + a(j-1)
ENDDO
```

```
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 Δa(10) Δa(10)+Δa(20) Δa(10)+Δa(20)+Δa (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.

Example from FLO52

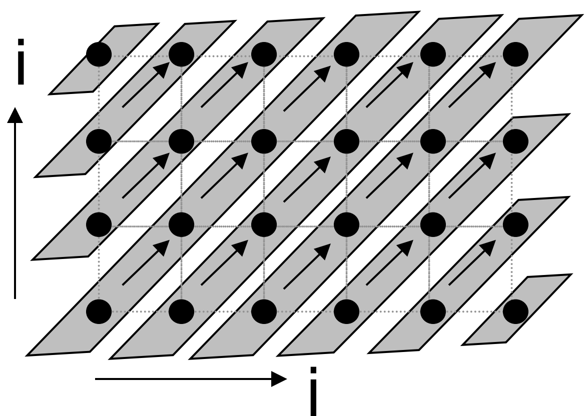
```
   DO 40 II=3,IL
   I         = I  -1
   DO 40 J=2,JL
   DW(I,J,N) = DW(I,J,N)  -R*(DW(I,J,N)  -DW(I+1,J,N))
40 CONTINUE
```

Loop Skewing

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

→

```
!$OMP PARALLEL DO
DO wave=1,?
  i = ?
  j = ?
  wsize = ?
  DO k=0,wsize-1
      A(i+k,j+k)=A(i-1+k,j-1+k)
  ENDDO
ENDDO
```

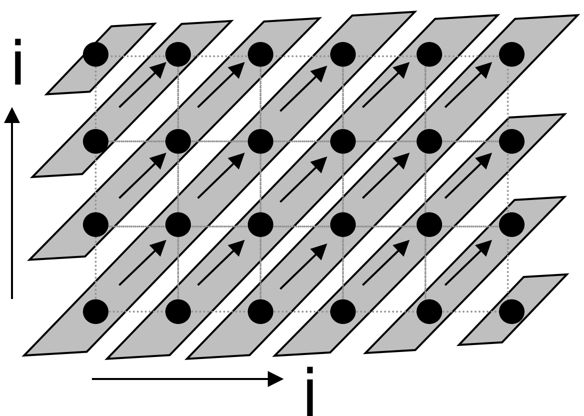


Iteration space graph:
Shared regions show wavefronts of iterations in the transformed code that can be executed in parallel.

Loop Skewing

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

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



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

3 Techniques for Multiprocessors:

Mapping parallelism to shared-memory machines

Loop Fusion

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

PARALLEL DO i=1,n
  C(i) = A(i)+D(i)
ENDDO
```

loop fusion →

```
PARALLEL DO i=1,n
  A(i) = B(i)
  C(i) = A(i)+D(i)
ENDDO
```

- Loop fusion is the reverse of loop distribution.
- reduces the loop fork/join overhead.
- Both transformations reorder computation;
  → data dependences show legality

Enforcing Data Dependence

- Criterion for correct transformation and execution of a computation involving a data dependence with vector v : $(=,\ldots\underset{L_s}{\underset{\downarrow}{<}},\ldots{}^{*})$

Let L_s be the outermost loop with non-"=" DD-direction :
- The direction at L_s must be "<"
- L_s must be executed serially

Note that a data dependence is defined with respect to an ordered (usually serial) execution. A fully parallel loop by definition does not have any cross-iteration dependence.

Loop Coalescing

```
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 Interchange

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

loop interchange →

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

Loop interchange affects:

- granularity of parallel computation (compare the number of parallel loops started)
- locality of reference (compare the cache-line reuse)

these two effects may impact the performance in the same or in opposite directions.

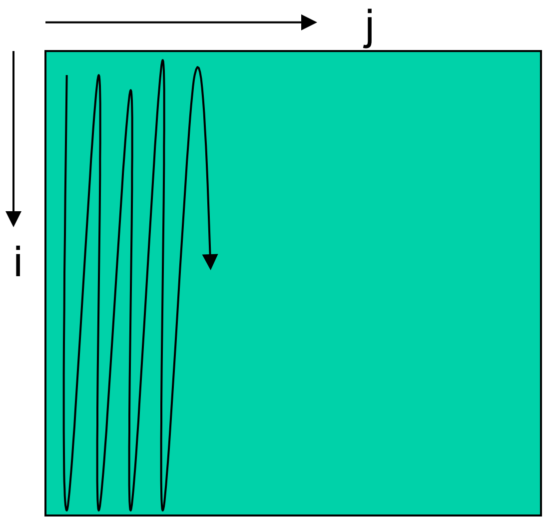
Loop interchange is subject to DD legality constraints.

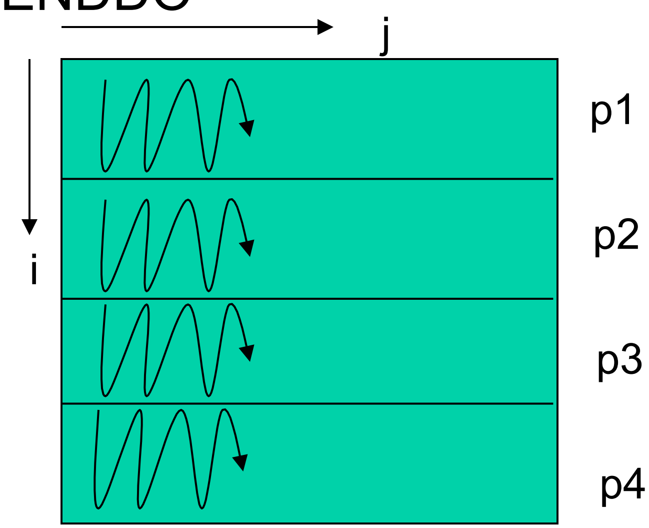
Loop Blocking

```
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
```





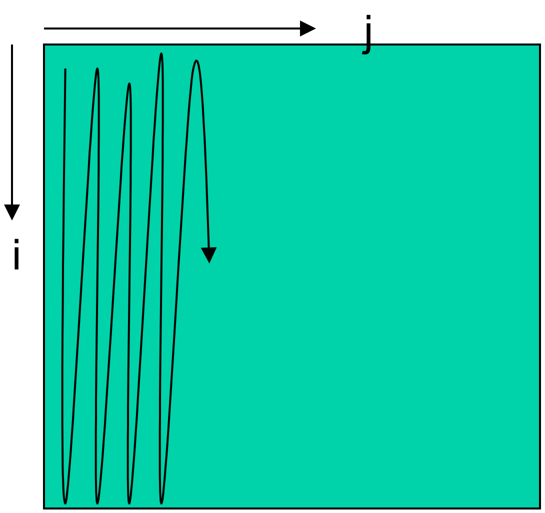
This is basically the same transformation as stripming, but followed by loop interchanging.

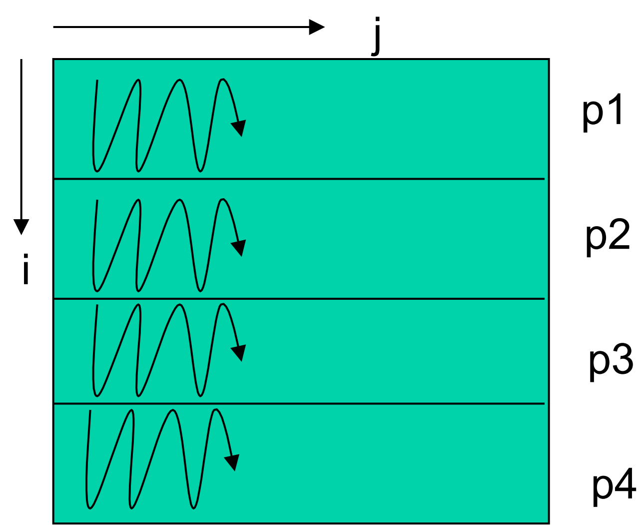
Loop Blocking

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+2)*d**

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

block < cachesize / (r1+r2+2)*d*num_proc

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.

Loop Distribution Enables Other Techniques

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

$\xrightarrow{\text{loop distribution enables interchange}}$

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

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

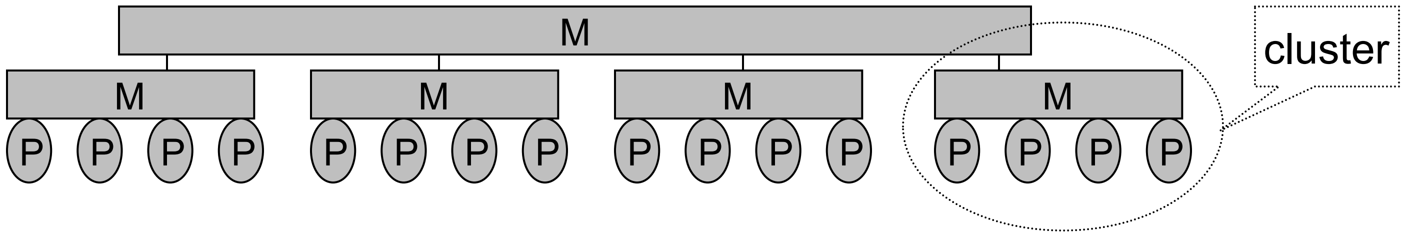
In a program with multiply-nested loops, there can be a large number of possible program variants obtained through distribution and interchanging

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

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.

loop distribution

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

dependence

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

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

vectorization

```
A(1:n)=B(1:n)+C(1:n)
D(1:n)=A(1:n)+A(0:n-1)
```

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

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

loop-carried 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

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

expansion

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

vectorization

```
T(1:n) = A(1:n)+B(1:n)
C(1:n) = T(1:n)+T(1:n)**2
```

Conditional Vectorization

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

$\downarrow$ conditional vectorization

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

Stripmining for Vectorization

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

5 Advanced Program Analysis

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

Step 1: determine *jump functions* for all subroutine arguments

```
Subroutine X(a,b,c)
e = 10
d = b+2
call somesub(c)
f = b*2
call this sub(a,d,c,e,f)
END
```

$\rightarrow$

J1 = a (jump function of first parameter)
J2 = b+2
J3 = $\perp$ (called *bottom, meaning non-constant)*
J4 = 10
J5 = $\perp$

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

Constant Propagation Algorithm

continued

Step 2:

- initialize all formal parameters to the value $\top$ (called *top*, meaning non-yet-known)
- for all jump functions:
  - if it is $\perp$: set formal parameter value to $\perp$
  - if it is constant and the value of the formal parameter is the same constant or $\top$: set it to this constant

Constant Propagation Algorithm

continued

Step 3:

1. put all formal parameters on a work queue
2. take a parameter from the queue:

   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
3. repeat 2 until the queue is empty

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

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

Interproc. DD-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

Interproc. DD-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

Interproc. DD-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

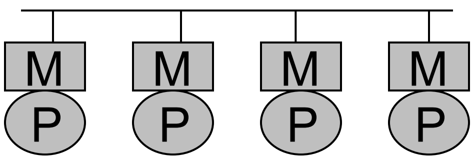
Important: reshaping may lose the implicit assertion that array bounds are not violated!

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

6 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:

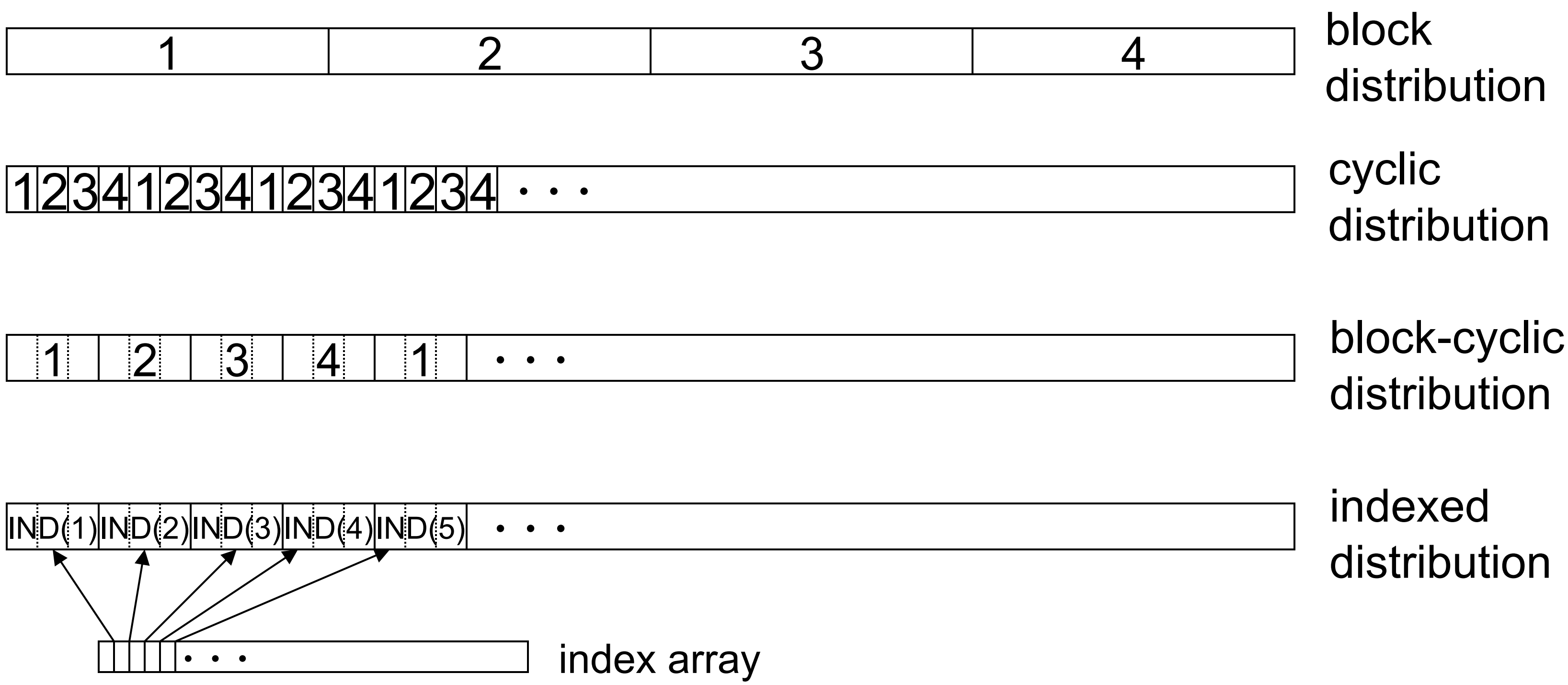
- Data is distributed onto the participating processors' memories

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



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

Message Generation for single-owner placement

EXAMPLE

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

$\xrightarrow{\text{message generation}}$

```
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.
- array distributions assumed to match 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!

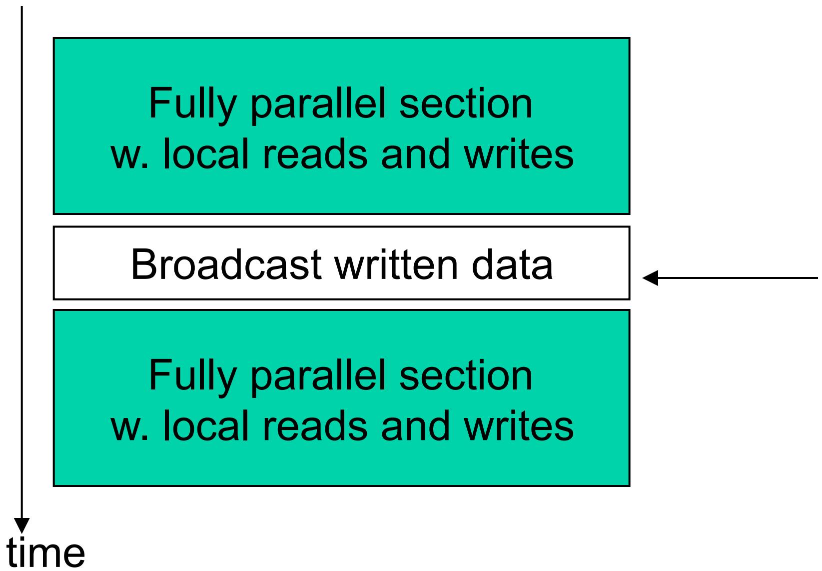
Compiler Optimizations

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 replication



Optimization: reduce broadcast operations to necessary point-to-point communication

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)

References

Data distribution and message generation:

(there is a large number of references on these topics)

- **A Novel Approach Towards Automatic Data Distribution,** Jordi Garcia, Eduard Ayguade and Jesus Labarta, Proc. Of Supercomputing ‘95, 1995.
- **An HPF compiler for the IBM SP2,** M. Gupta and S. Midkiff and E. Schonberg and V. Seshadri and D. Shields and K. Wang and W. Ching and T. Ngo, Proceedings of Supercomputing '95, 1995.

Message Generation under Replication:

- **Towards Automatic Translation of OpenMP to MPI,** Ayon Basumallik and Rudolf Eigenmann, Proc. of the International Conference on Supercomputing, ICS'05, 2005.
- **Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems,** Ayon Basumallik and Rudolf Eigenmann, *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'06),* ACM Press, 2006.

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, however,
- 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

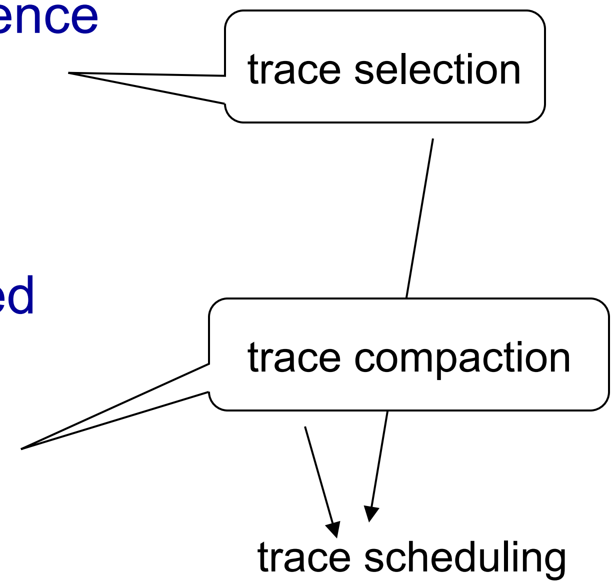
(invented for VLIW processors, still a useful terminology)

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

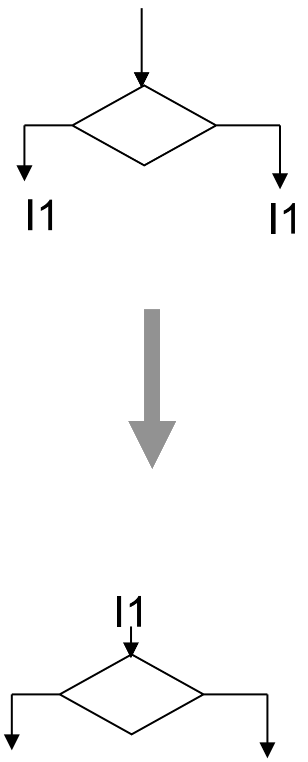
Branch Elimination

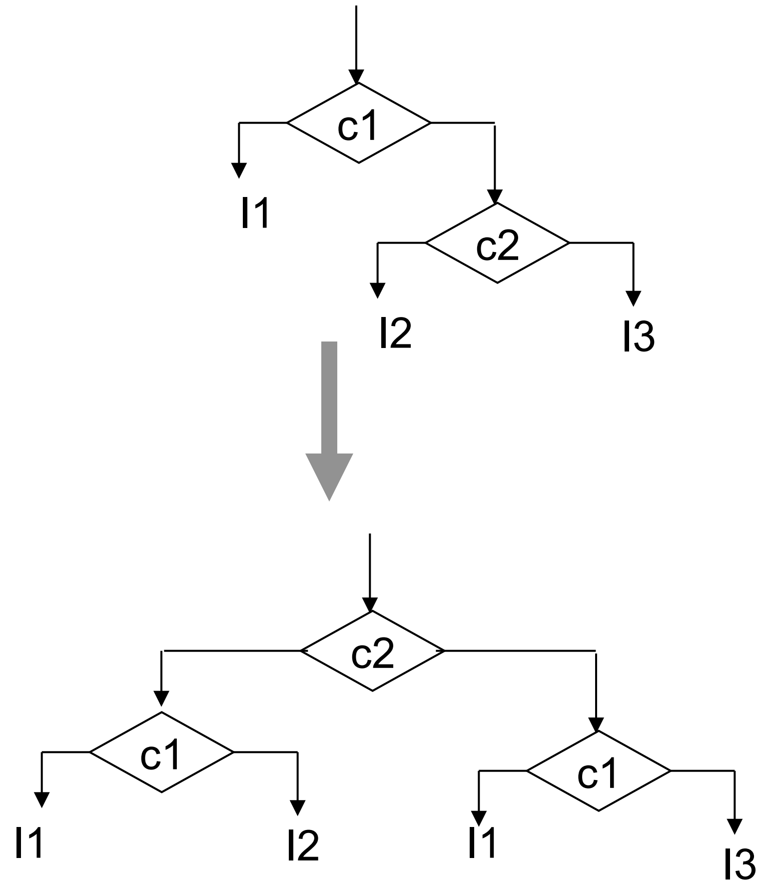
- Example:

```
      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*
$\overline{p}$: 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

```
            ind = i0
dependence  . . .
            ind = ind+1
dependence  . . .
            ind = ind+1
```

→

```
ind = i0
. . .
ind = i0+1
. . .
ind = i0+2
```

```
            sum = sum+expr1
dependence  . . .
            sum = sum+expr2
dependence  . . .
            sum = sum+expr3
dependence  . . .
            sum = sum+expr4
```

→

```
s1=expr1
. . .
s1=s1+expr2
. . .
s2=expr3
. . .
s2=s2+expr4
. . .
sum=sum+s1+s2
```

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.

8 OpenMP for Distributed Parallel Systems

Is OpenMP a Useful Programming Model for Distributed Processors?

- OpenMP is a parallel programming model that assumes a shared address space

  ```
  #pragma OMP parallel for
  for (i=1; 1<n; i++) {a[i] = b[i];}
  ```

- Why is it difficult to implement OpenMP for distributed processors?

  The compiler or runtime system will need to
  - partition and place data onto the distributed memories
  - send/receive messages to orchestrate remote data accesses

  HPF (High Performance Fortran) was a large-scale effort to do so - without success

- So, why should we try again ?

  OpenMP is an easier programming (higher-productivity?) programming model. OpenMP
  - allows programs to be parallelized incrementally, starting from the serial versions,
  - relieves the programmer of the task of managing the movement of logically shared data.

Baseline Translation of OpenMP to MPI Compiler

- Execution Model
  - SPMD model
    - Serial Regions are replicated on all processes
    - Iterations of parallel ***for*** loops are distributed (using static block scheduling)
  - Shared Data is allocated on all nodes
    - There is no concept of “owner” (contrast to HPF)

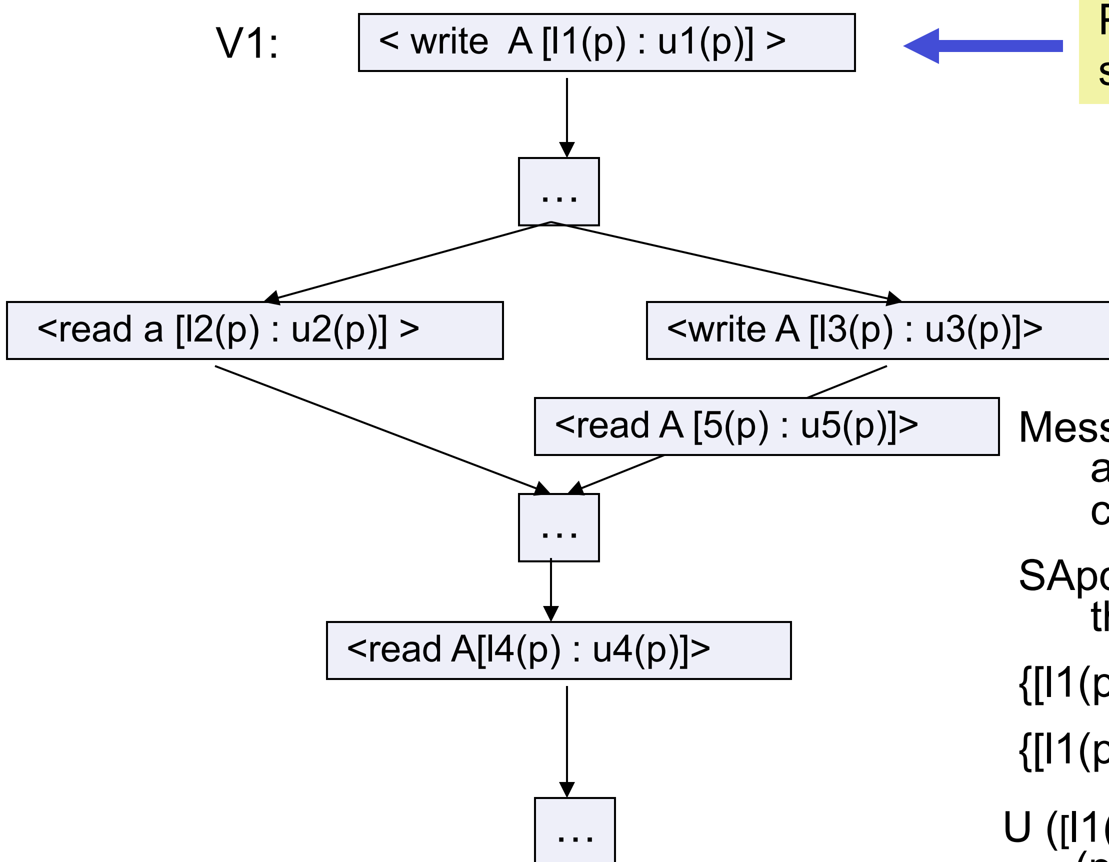
      There are only producers and consumers of shared data
    - At the end of a parallel loop, producers communicate shared data to *potential* future consumers
    - The compiler uses array section analysis for summarizing array accesses

Baseline Translation

Translation Steps:

1. Identify all shared data
2. Create annotations for accesses to shared data (use regular section descriptors to summarize array accesses)
3. Use interprocedural data flow analysis to identify *potential consumers*; incorporate OpenMP relaxed consistency specifications
4. Create message sets to communicate data between producers and consumers

Message Set Generation



Processor p writes array section A from l1(p) to u1(p)

Message Set at RSD vertex V1, for array A from process p to process q computed as

SApq = Elements of A with subscripts in the set

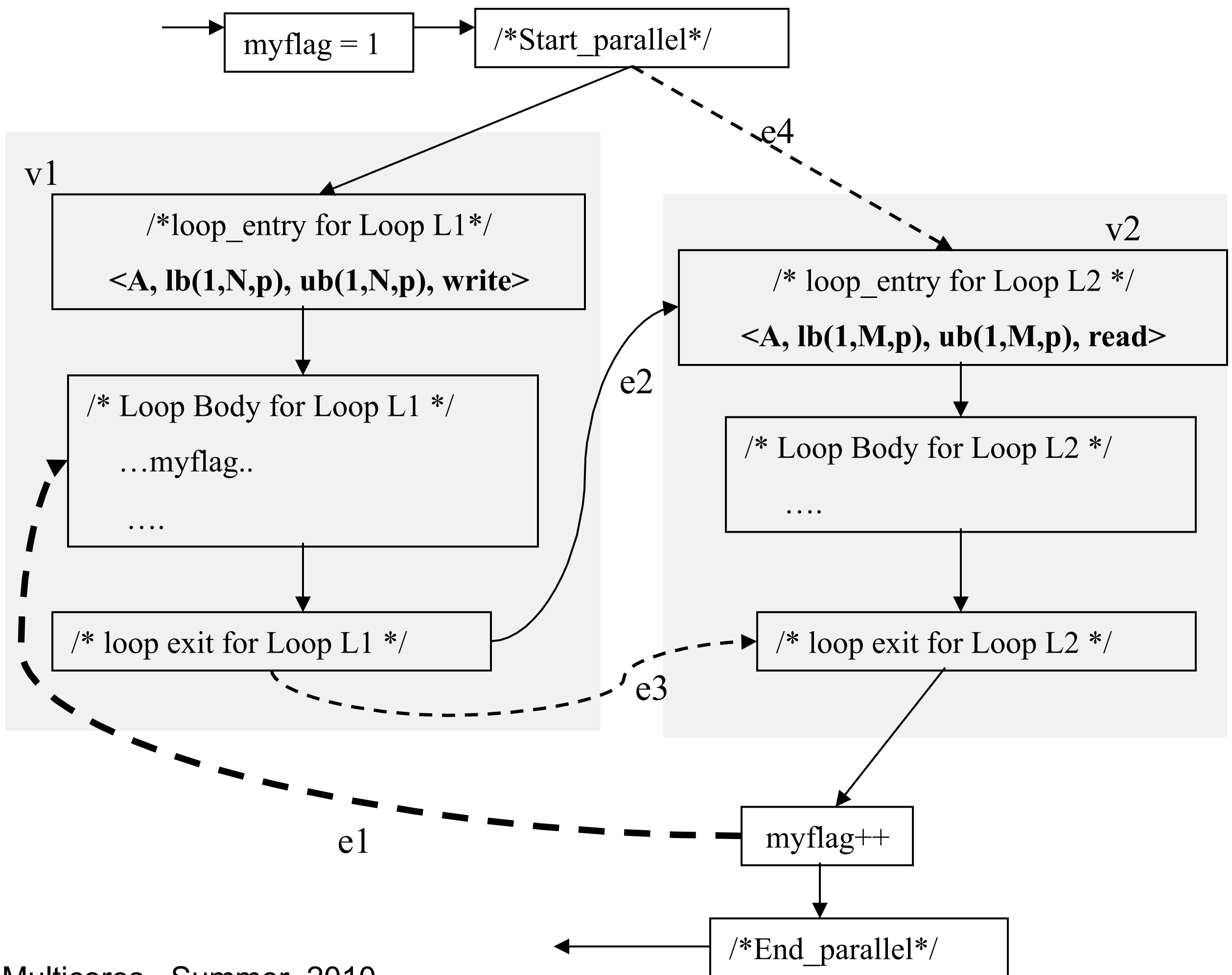
$\{[l1(p),u1(p)]\cap[l2(q),u2(q)]\}$ U

$\{[l1(p),u1(p)]\cap[l4(q),u4(q)]\}$

U $([l1(p),u1(p)]\cap\{[l5(q),u5(q)]-[l3(p),u3(p)]\})$

Incorporating OpenMP Relaxed Memory Consistency Specifications

```
mydo = 1 ;
#pragma omp parallel
{
/* Loop L1 */
#pragma omp for nowait
   for(j=1;j<N;j++) {
#pragma omp flush(myflag)
         …myflag.. ;
    A[j] = …
   }
/* Loop L2 */
#pragma omp for nowait
    for(j=1;j<M;j++) {
         … = A[i]...
    }
  myflag++ ;
}
```



Translation of Irregular Accesses

- Irregular Access – A[B[i]], A[f(i)] where B[i] is a subscript array or f(i) is a non-affine function of the loop index.
  - Reads: assumed the whole array accessed
  - Writes: inspect at runtime, communicate at the end of parallel loop
- A key property is Monotonicity : $i > j \rightarrow B[i] > B[j]$

  Monotonicity is useful because it provides bounds on the irregular subscript in terms of the loop bounds

  - For lb<i<ub, B[lb] < B[i] < B[ub]

  Monotonicity allows the compiler to

  - tighten array sections
  - avoid runtime inspection of writtten array sections that do not overlap

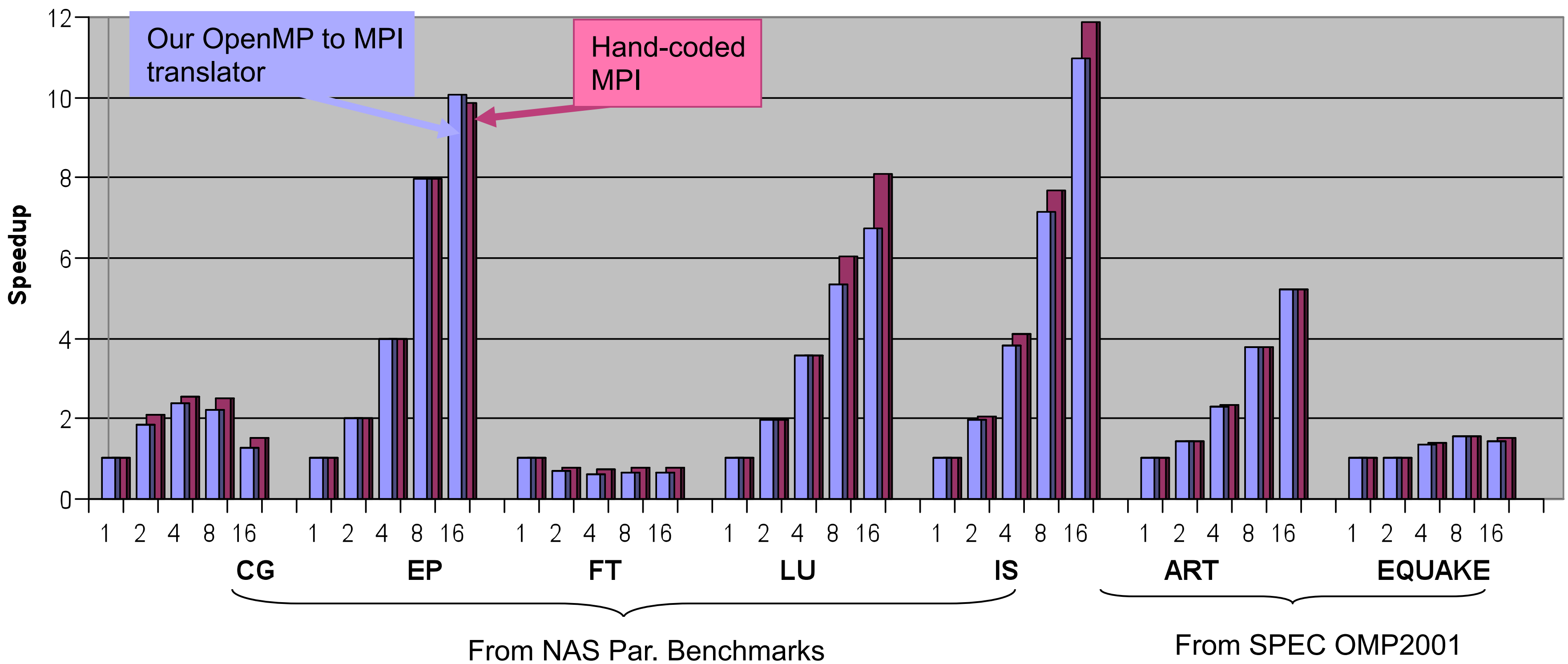
Optimizations based on Collective Communication

- Recognition of Reduction Idioms
  - Recognize program patterns that implement array reductions – usually: combination of parallel loop and critical section.
  - Translate them to MPI_Reduce / MPI_Allreduce functions.
- Casting sends/receives in terms of *alltoall* calls
  - In general, communication between producers and consumers are done using non-blocking send/recv and MPI_Wait
  - There may be insufficient distance between the production and consumption points in the program to allow overlap of computation and communication
  - When the producer-consumer relationship is many-to-many and there is insufficient distance between producers and consumers, cast the sends/recvs into a single MPI_Alltoallv call

Performance Evaluation

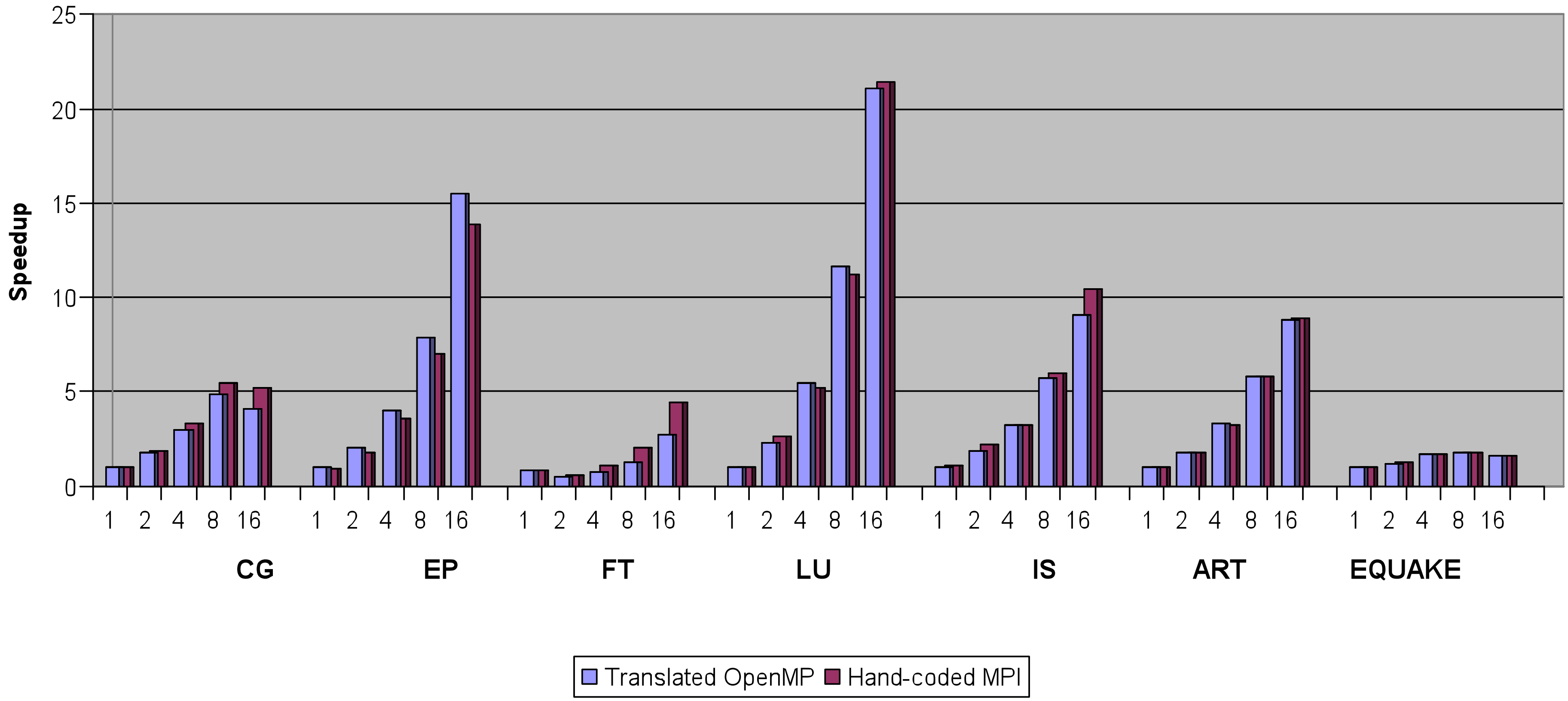
of Baseline Translation

Platform I – Cluster of sixteen PIII 800 MHz Linux nodes, with 256 MB memory per node, connected by a commodity 100 Mbps Ethernet network.

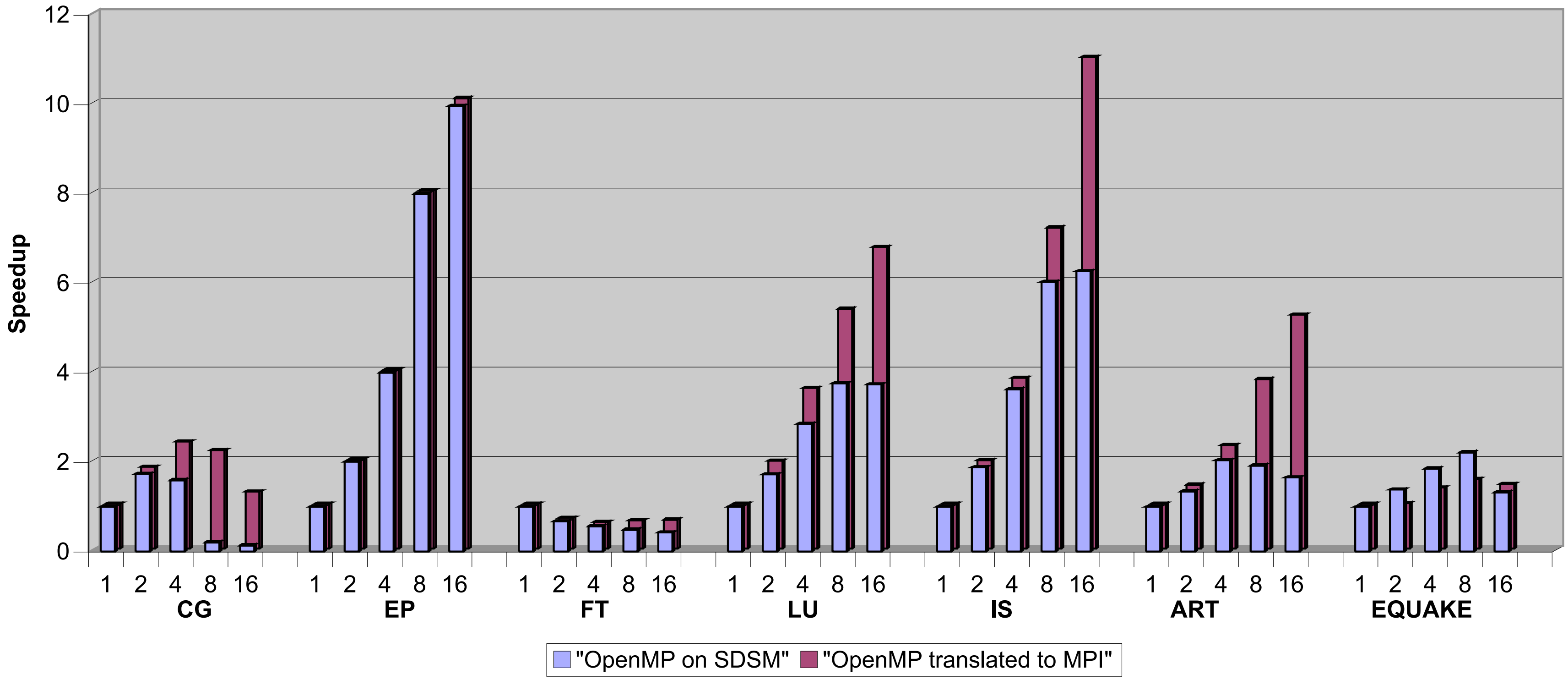


Performance on IBM-SP2

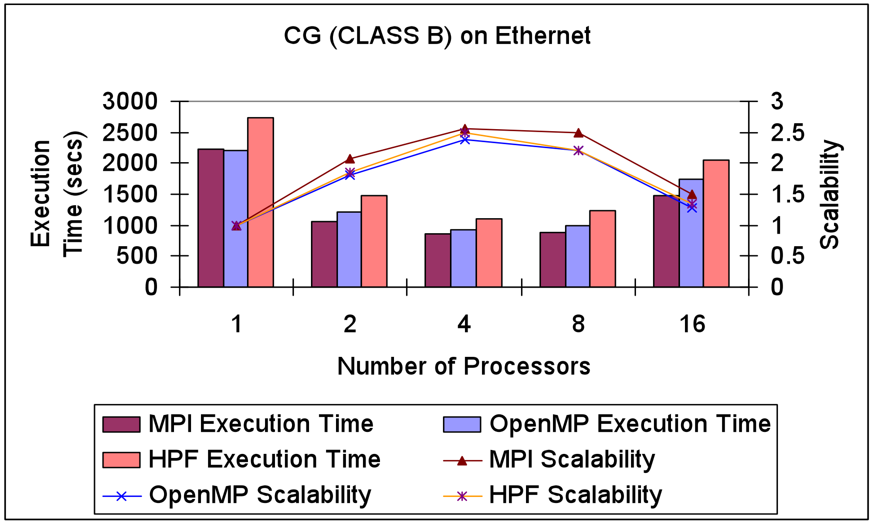
Platform II – Sixteen IBM SP-2 WinterHawk-II nodes connected by a high-performance switch.

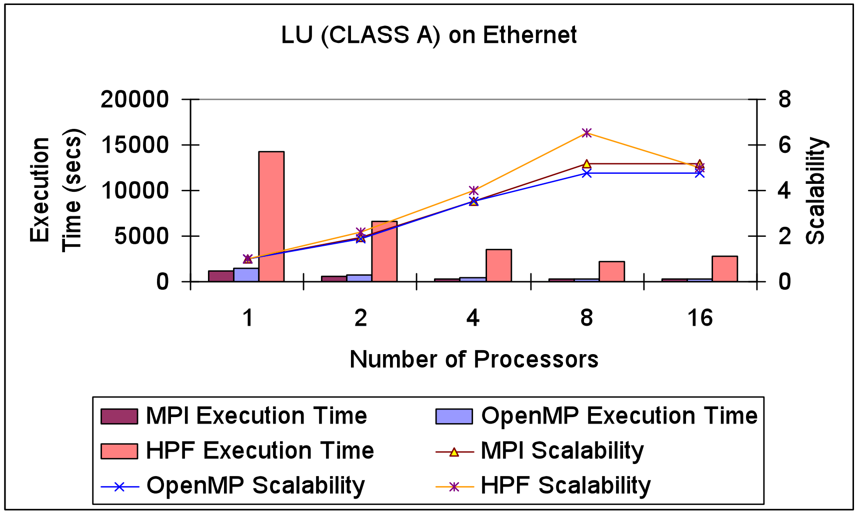


Comparison with SDSM on Linux Cluster



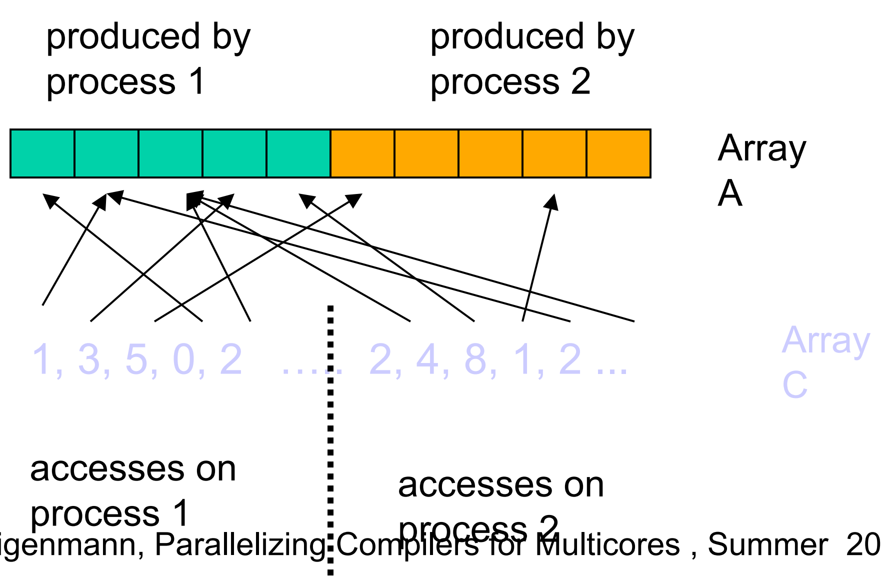
Performance Comparison with HPF on Linux Cluster





We can do more for Irregular Applications

```
L1 : #pragma omp parallel for
        for(i=0;i<10;i++)
              A[i] = ...

L2 : #pragma omp parallel for
        for(j=0;j<20;j++)
              B[j] = A[C[j]] + ...
```

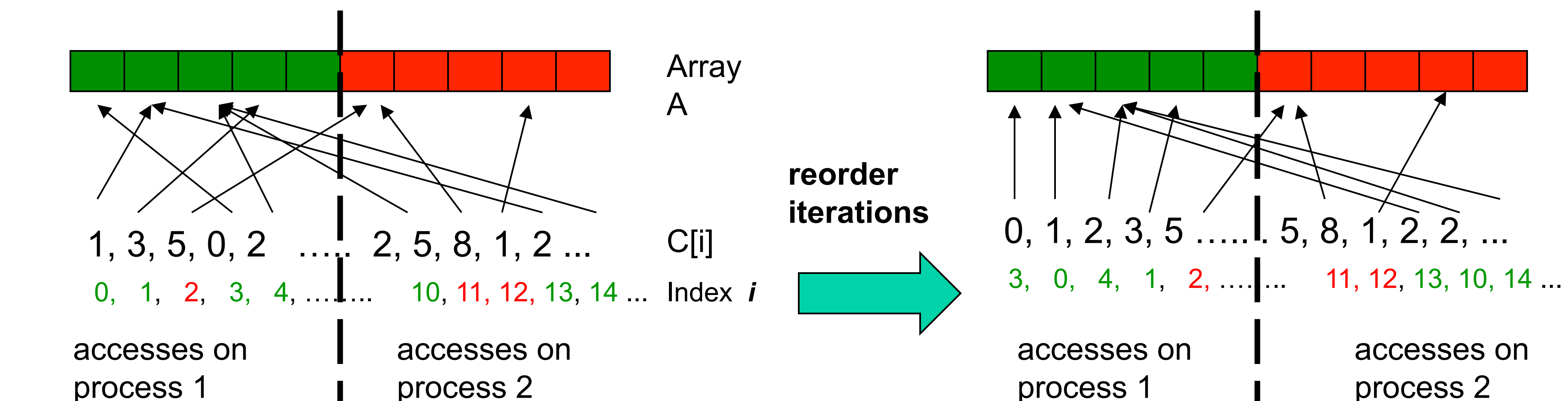


- Subscripts of accesses to shared arrays not always analyzable at compile-time
- Baseline OpenMP to MPI translation:
  - Conservatively estimate that each process accesses the entire array
  - Try to deduce properties such as monotonicity for the irregular subscript to refine the estimate
- Still, there may be redundant communication
  - Runtime tests (inspection) are needed to resolve accesses

Inspection

- Inspection allows accesses to be differentiated (at runtime) as local and non-local accesses.
- Inspection can also map iterations to accesses. This mapping can then be used to re-order iterations:
  - first, iterations that access local data
  - then, iterations that access remote data

  => Communication of remote data can be overlapped with the computation of iterations that access local data



Loop Restructuring

- Simple iteration reordering may not be sufficient to expose the full set of possibilities for computation-communication overlap.

```
L1 : #pragma omp parallel for
       for(i=0;i<N;i++)
              p[i] = x[i] + alpha

L2 : #pragma omp parallel fo
       for(j=0;j<N;j++) {
              w[j] = 0 ;
              for(k=rowstr[j];k<rowstr[j+1];k++)
S2:                          w[j] = w[j] + a[k]*p[col[k]] ;
       }
```

Distribute loop L2 to form loops L2-1 and L2-2

```
L1 : #pragma omp parallel for
       for(i=0;i<N;i++)
           p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
         for(j=0;j<N;j++) {
                w[j] = 0 ;
         }

L2-2: #pragma omp parallel for
         for(j=0;j<N;j++) {
         for(k=rowstr[j];k<rowstr[j+1];k++)
             S2: w[j] = w[j] + a[k]*p[col[k]] ;
         }
```

Reordering loop ***L2*** may still not club together accesses from different sources

Loop Restructuring continued

```
L1 : #pragma omp parallel for
     for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
       for(j=0;j<N;j++) {
             w[j] = 0 ;
        }

L2-2: #pragma omp parallel for
       for(j=0;j<N;j++) {
       for(k=rowstr[j];k<rowstr[j+1];k++)
           S2: w[j] = w[j] + a[k]*p[col[k]] ;
       }
```

Coalesce nested loop L2-2 to form loop L3

```
L1 : #pragma omp parallel for
     for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
       for(j=0;j<N;j++) {
             w[j] = 0 ;
        }

L3: for(i=0;i<num_iter;i++)
            w[T[i].j] = w[T[i].j] +
              a[T[i].k]*p[T[i].col] ;
```

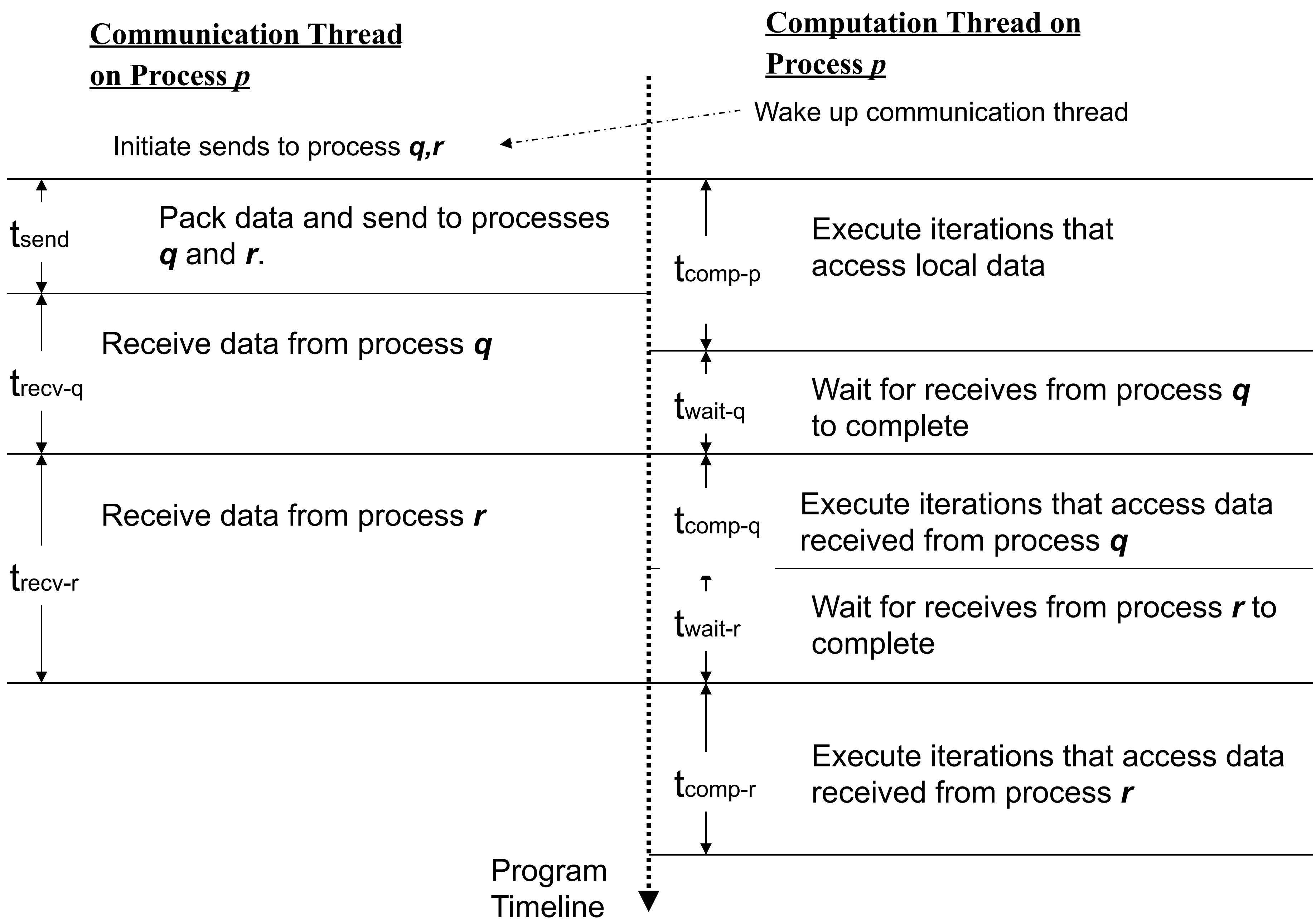
Reorder iterations of loop L3 to achieve computation-communication overlap

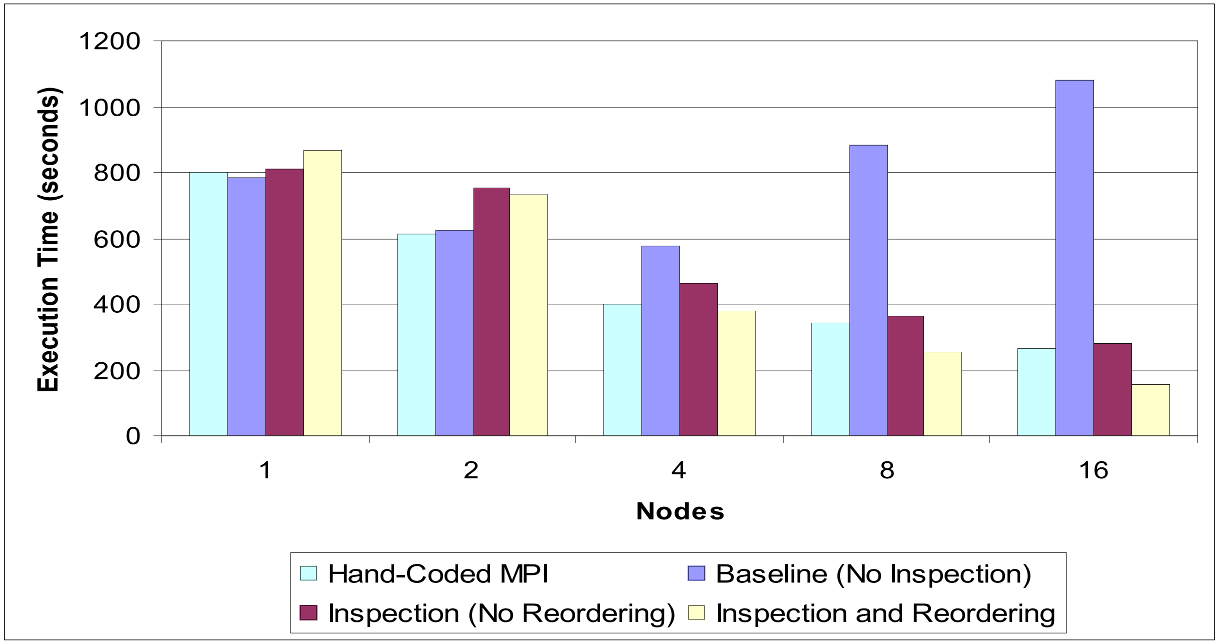
Final restructured and reordered loop

The T[i] data structure is created and filled in by the inspector

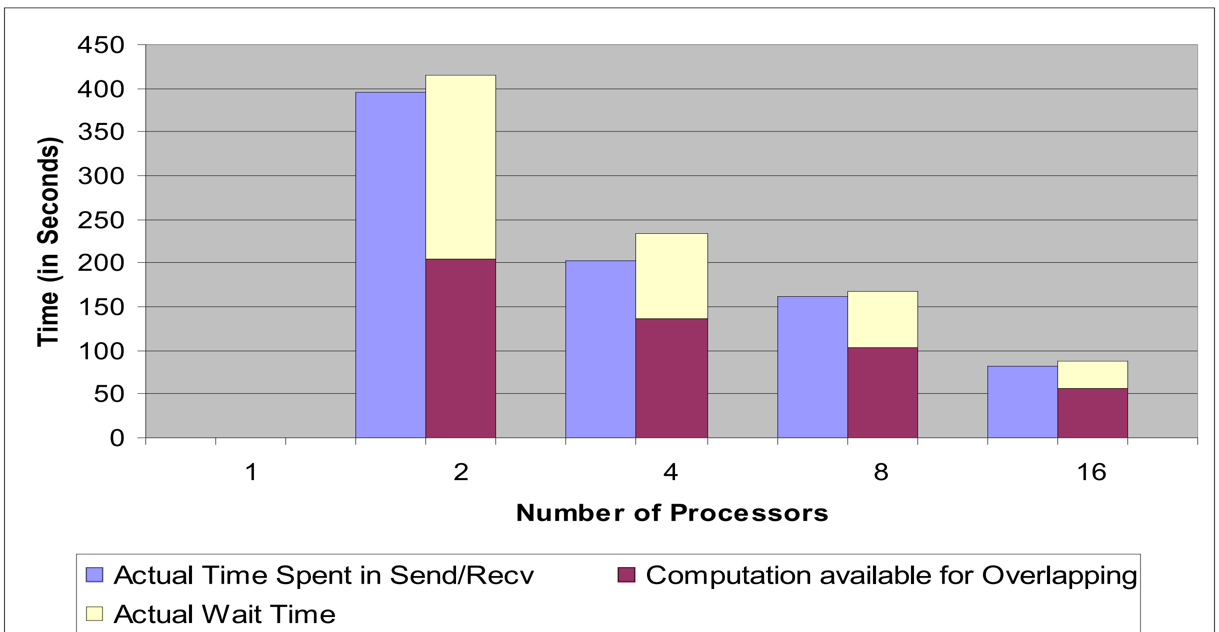
Achieving Actual Computation-Communication Overlap

- Non-blocking send/recv calls may not actually progress concurrently with computation.
  - Use a multi-threaded runtime system with separate computation and communication threads – on dual CPU machines these threads can progress concurrently.
- The compiler extracts the send/recvs along with the packing/unpacking of message buffers into a communication thread.

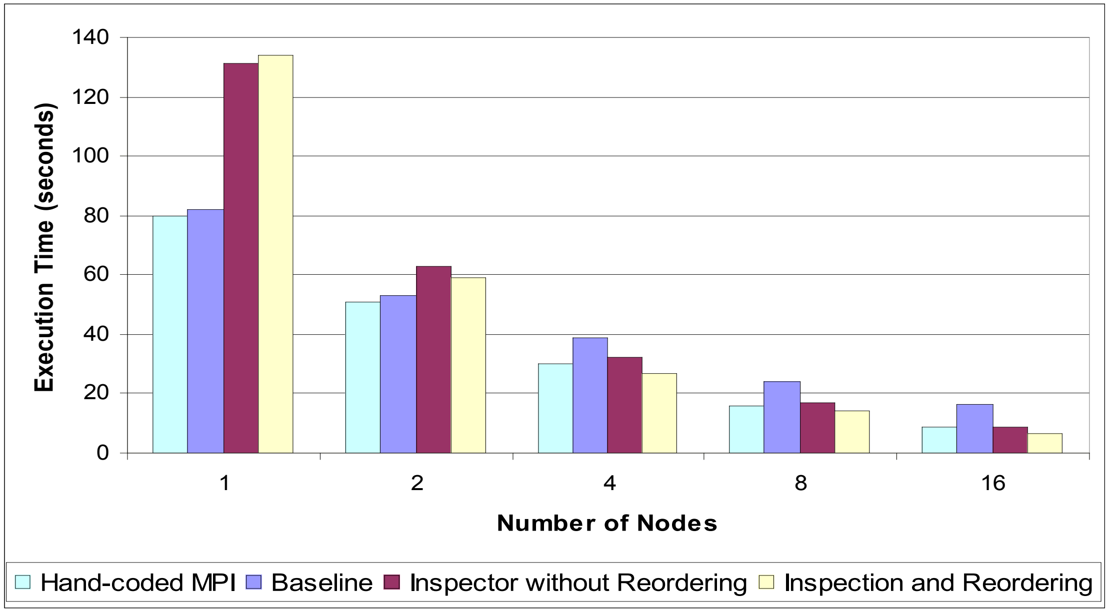

Communication Thread on Process p
Computation Thread on Process p
Wake up communication thread
Initiate sends to process q,r
t_send
Pack data and send to processes q and r.
t_comp-p
Execute iterations that access local data
t_recv-q
Receive data from process q
t_wait-q
Wait for receives from process q to complete
t_comp-q
Execute iterations that access data received from process q
Receive data from process r
t_recv-r
t_wait-r
Wait for receives from process r to complete
t_comp-r
Execute iterations that access data received from process r
Program Timeline



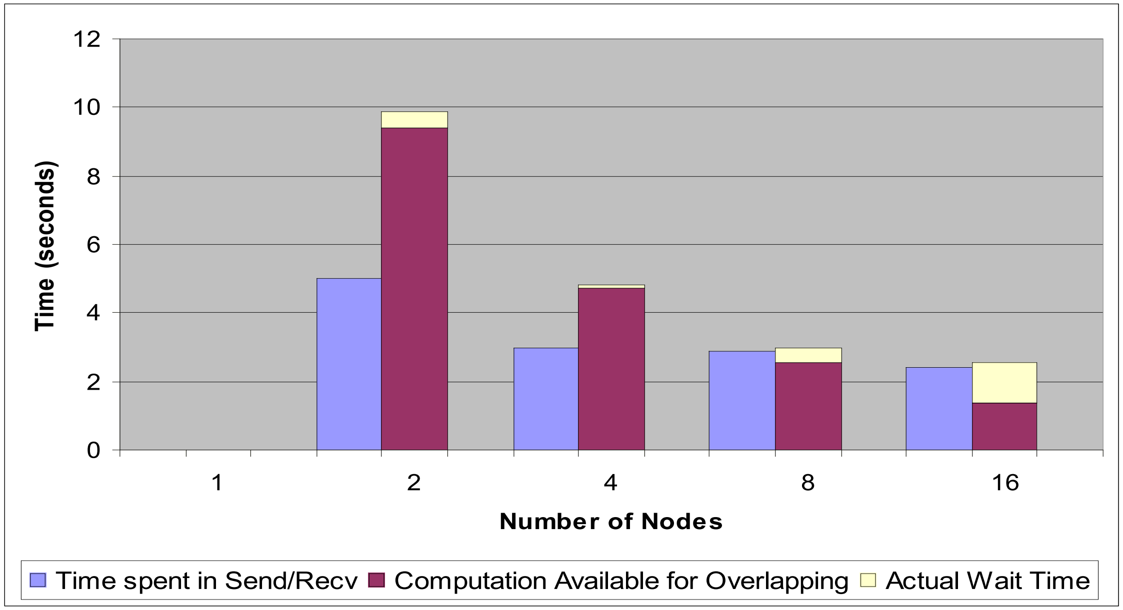
Performance of Equake



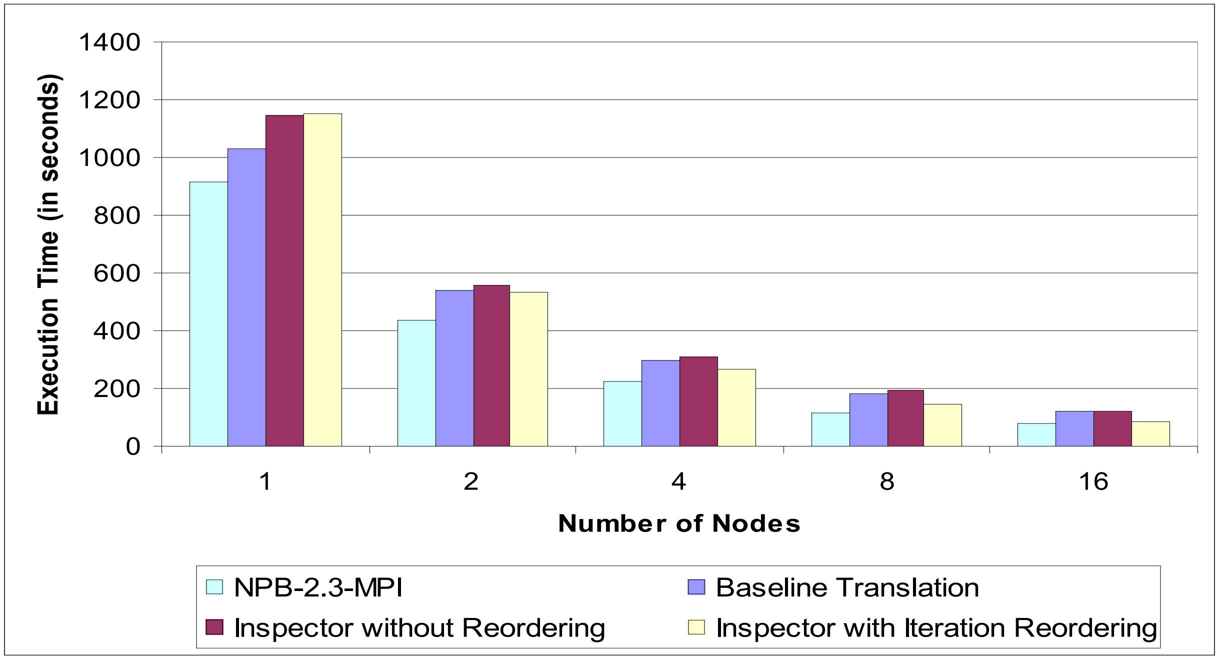
Computation-communication overlap in Equake



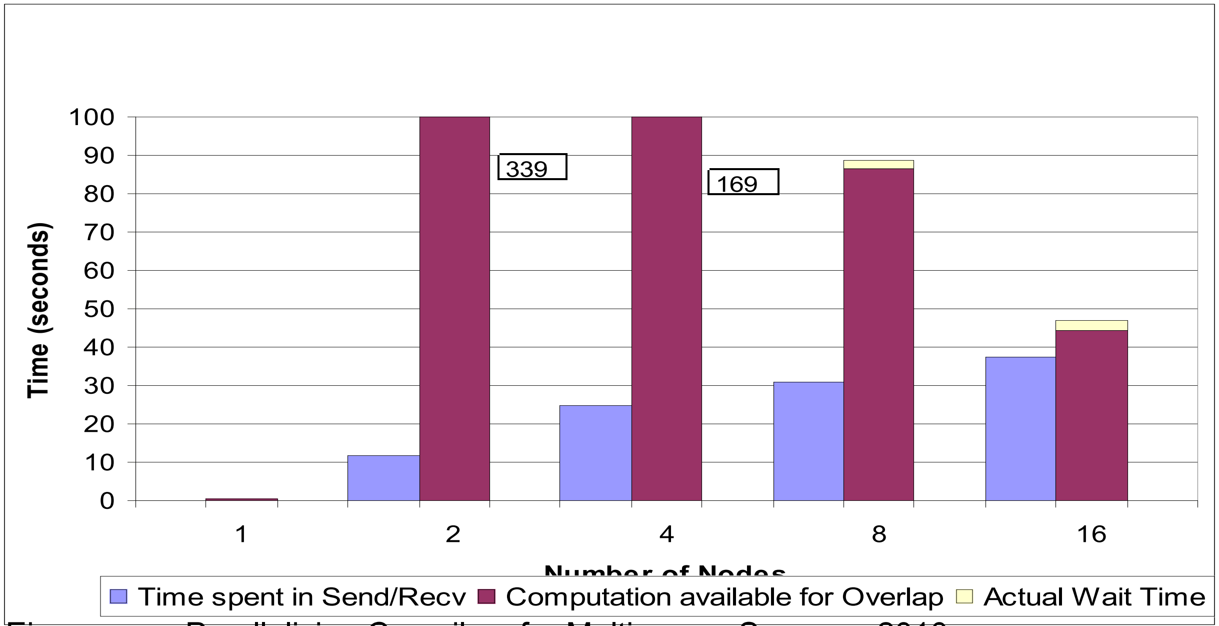
Performance of Moldyn



Computation-communication overlap in Moldyn



Performance of CG



Computation-communication overlap in CG

Summary

OpenMP for Distributed Systems

- There is hope for easier programming models on distributed processing systems (DPS)
- OpenMP can be translated effectively onto DPS; we have used benchmarks from
  - SPEC OMP
  - NAS
  - additional irregular codes
- Caveats:
  - black-belt programmers will always be able to do better
  - advanced compiler technology is involved. There will be performance surprises
  - Larger set of and full compiler implementation are needed
    => this is ongoing work

References

- **Towards Automatic Translation of OpenMP to MPI**, Ayon Basumallik and Rudolf Eigenmann, *Proc. of the International Conference on Supercomputing, ICS'05*, pages 189--198, 2005.

- **Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems**, Ayon Basumallik and Rudolf Eigenmann, *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'06),* ACM Press, 2006.

- **Incorporation of OpenMP Memory Consistency into Conventional Dataflow Analysis**, Ayon Basumallik and Rudolf Eigenmann, *Proc. of IWOMP'08 Int'l Workshop on OpenMP*, 2008

9. Autotuning: Moving Compile-time Decisions Into Runtime

Why Autotuning ?
my bias

Ultimate goal: Dynamic Optimization Support For Compilers and More

- Runtime decisions for compilers are necessary because compile-time decisions are too conservative
  - Insufficient information about program input, architecture
  - When to apply what transformation in which flavor?
  - Polaris compiler has some 200 switches

    Example of an important switch: parallelism threshold
  - Early runtime decisions:
    - Multi-version loops, runtime data-dependence test, 1980s
- My goals:
  - Looking for tuning parameters and evidence of performance difference
  - Go beyond the "usual": unrolling, blocking, reordering
  - Show performance on real programs

Is there Potential ?

You bet!

- Imagine you (the compiler) had full knowledge of input data and execution platform of the program

"Amdahl's law of dynamic optimization"

Early Results on Fully-Dynamic Adaptation

- ADAPT system (Michael Voss - 2000)
- Features:
  - Triage
    - tune the most deserving program sections first
  - Used remote compilation
    - Allowed standard compilers and all options to be used
  - AL - adapt language
- Issues:
  - Scalability to large number of optimizations
  - Shelter and re-tune

More Recent Work

Offline Tuning - “Profile-time” tuning

Zhelong Pan

Challenges:

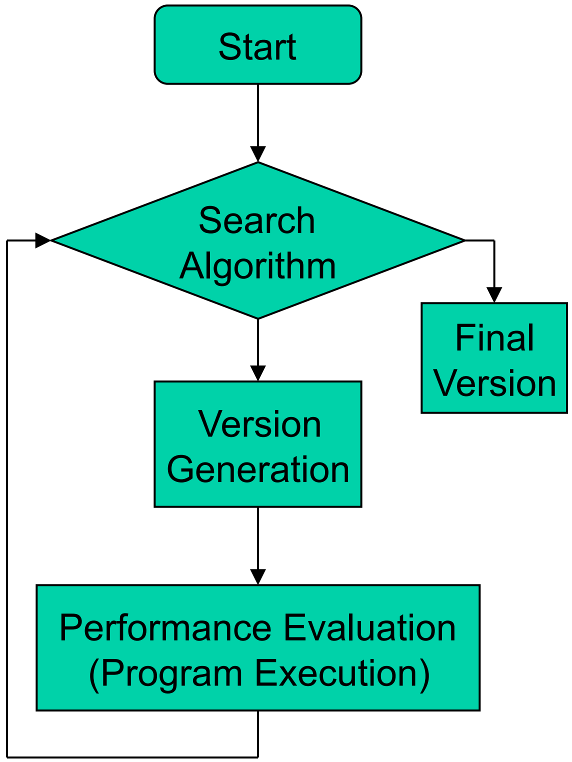
1. Explore the optimization space
   Empirical optimization algorithm - CGO 2006
2. Comparing performance
   Fair Rating methods - SC 2004
   - Comparing two (differently optimized) subroutine invocations
3. Choosing procedures as tuning candidates
   Tuning section selection - PACT 2006
   - Program partitioning into tuning sections

Two goals : increase program performance and reduce tuning time

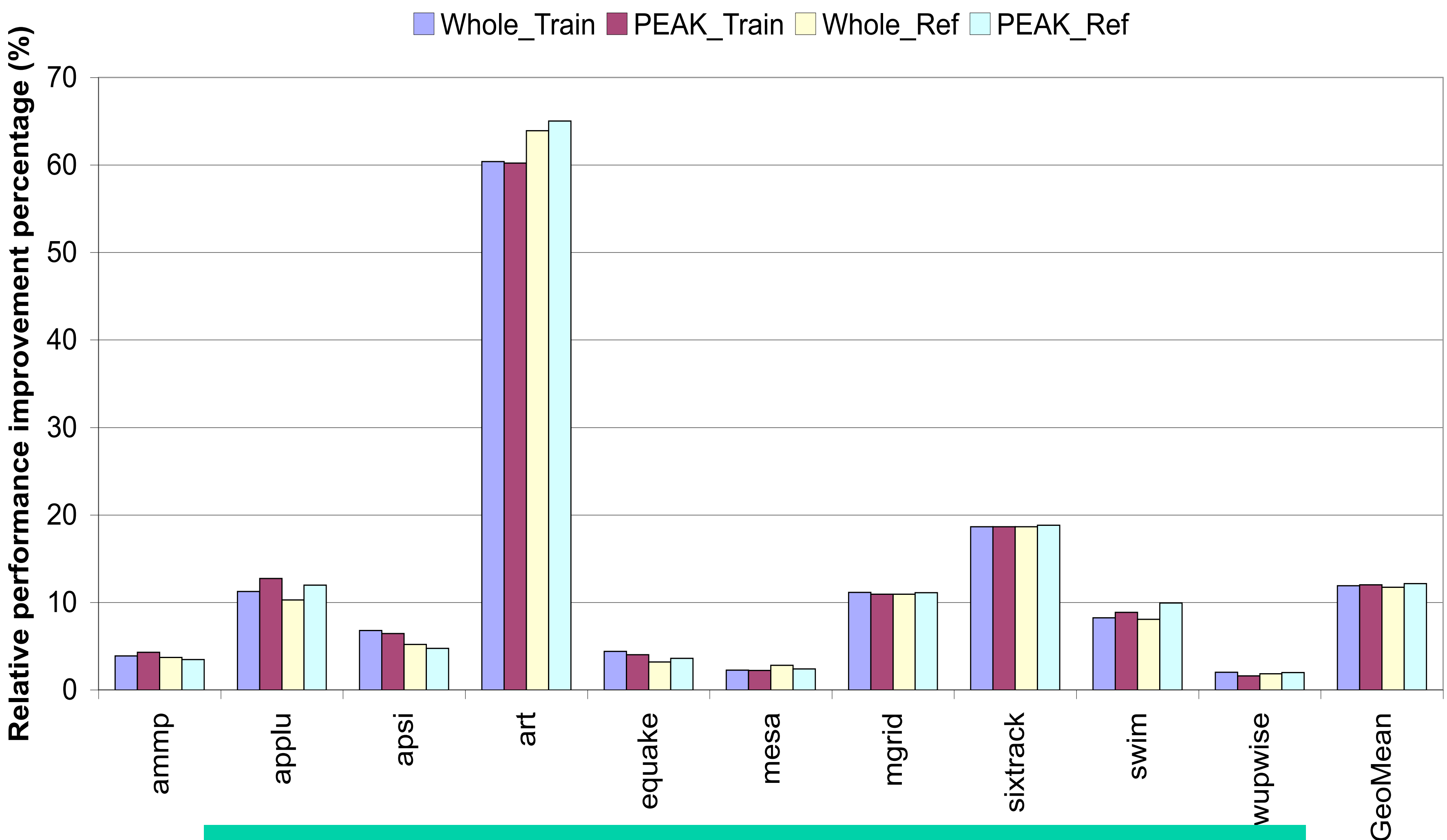
Whole-Program Tuning

Search Algorithms

- BE: batch elimination
  - Eliminates “bad” optimizations in a batch => fast
  - Does not consider interaction => not effective
- IE: iterative elimination
  - Eliminates one “bad” optimization at a time => slow
  - Considers interaction => effective
- **CE: combined elimination (final algorithm)**
  - **Eliminates a few “bad” optimizations at a time**
- Other algorithms
  - optimization space exploration, statistical selection, genetic algorithm, random search



Performance Improvement



Tuning Goal: determine the best combination of GCC options

Tuning at the Procedure Level

(1) Tuning Section Selection (TSS)

(2) Rating Method Analysis (RMA)

(3) Code Instrumentation (CI)

(4) Driver Generation (DG)

Pre-Tuning

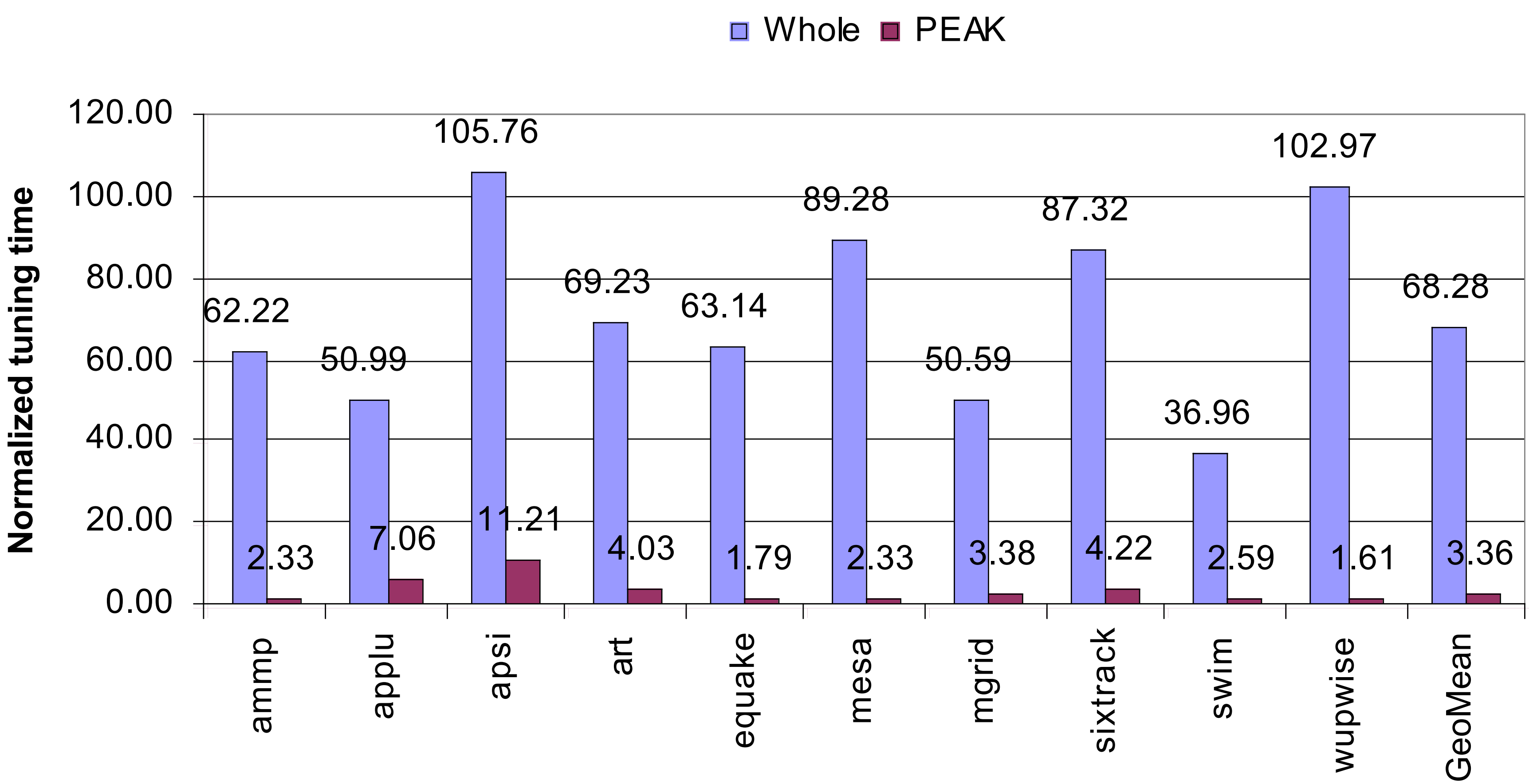
(5) Performance Tuning (PT)

During Tuning

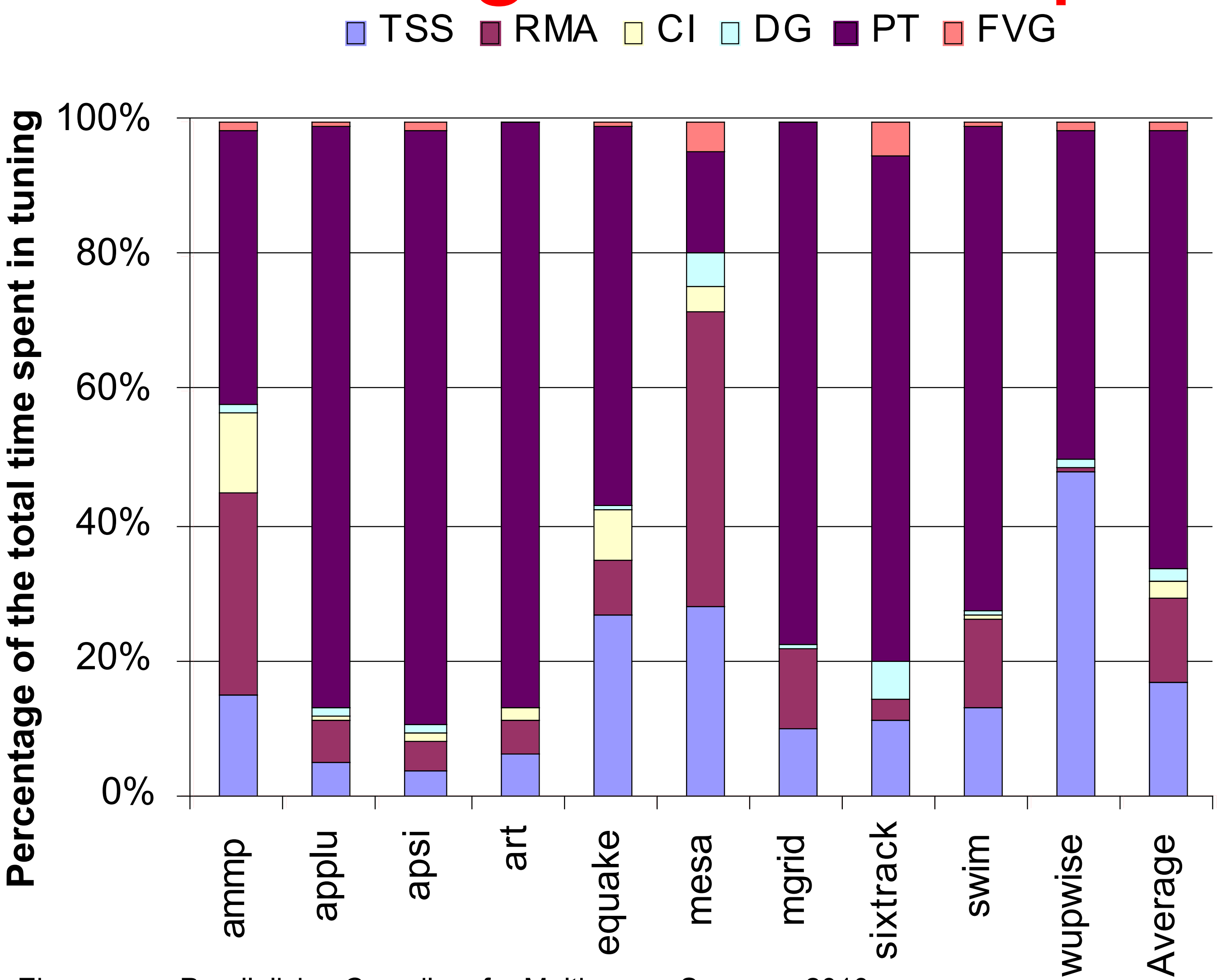
(6) Final Version Generation (FVG)

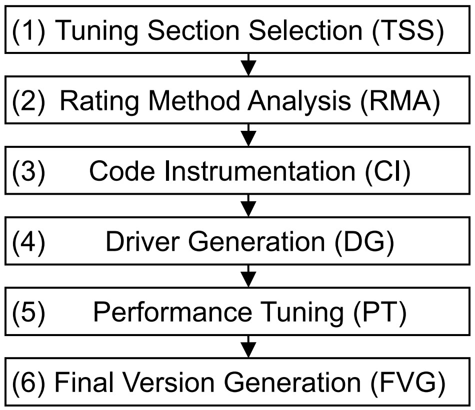
Post-Tuning

Reduction of Tuning Time through Procedure-level Tuning



Tuning Time Components



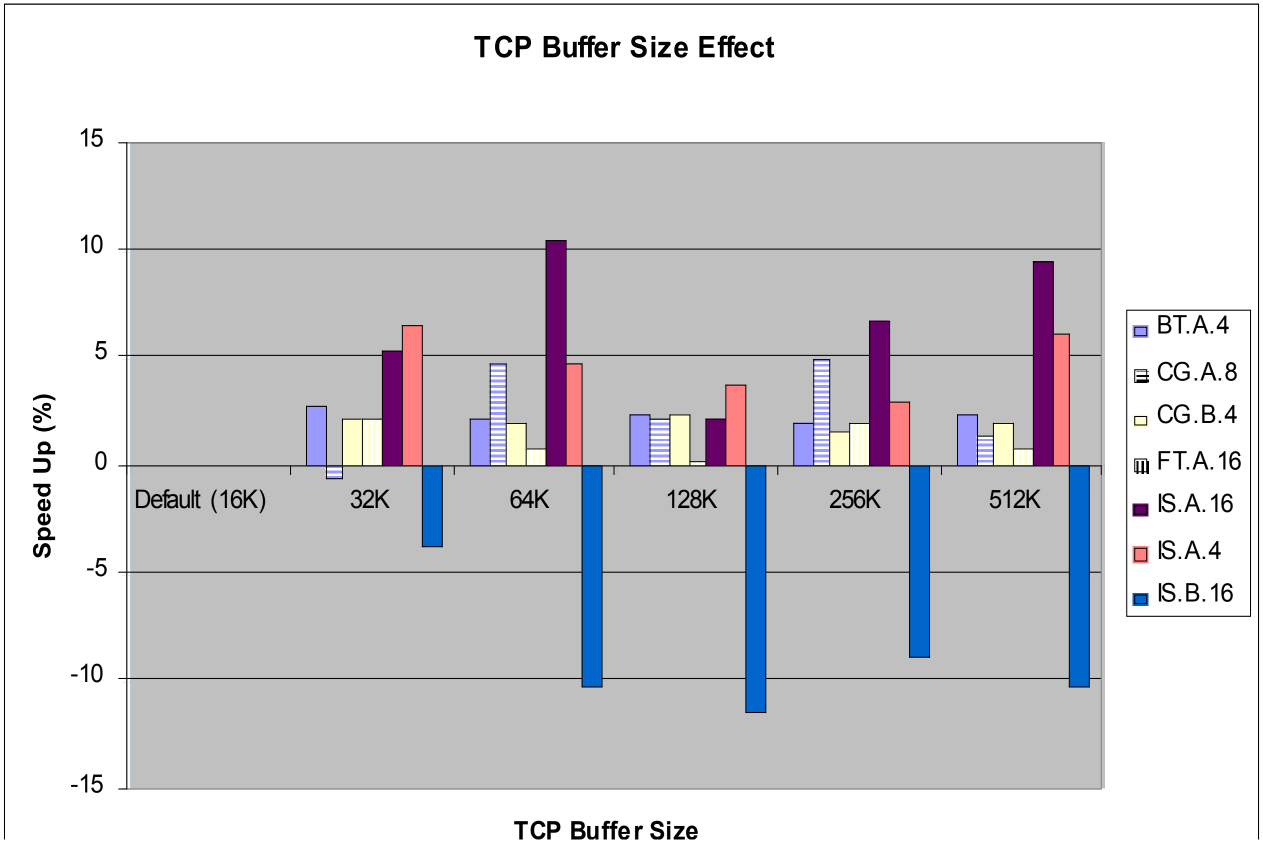


Ongoing Work

Beyond autotuning of compiler options

- New applications of the tuning system
  - MPI parameter tuning
  - Tuning library selection - (ScalaPack, ...)
  - OpenMP to MPI translator

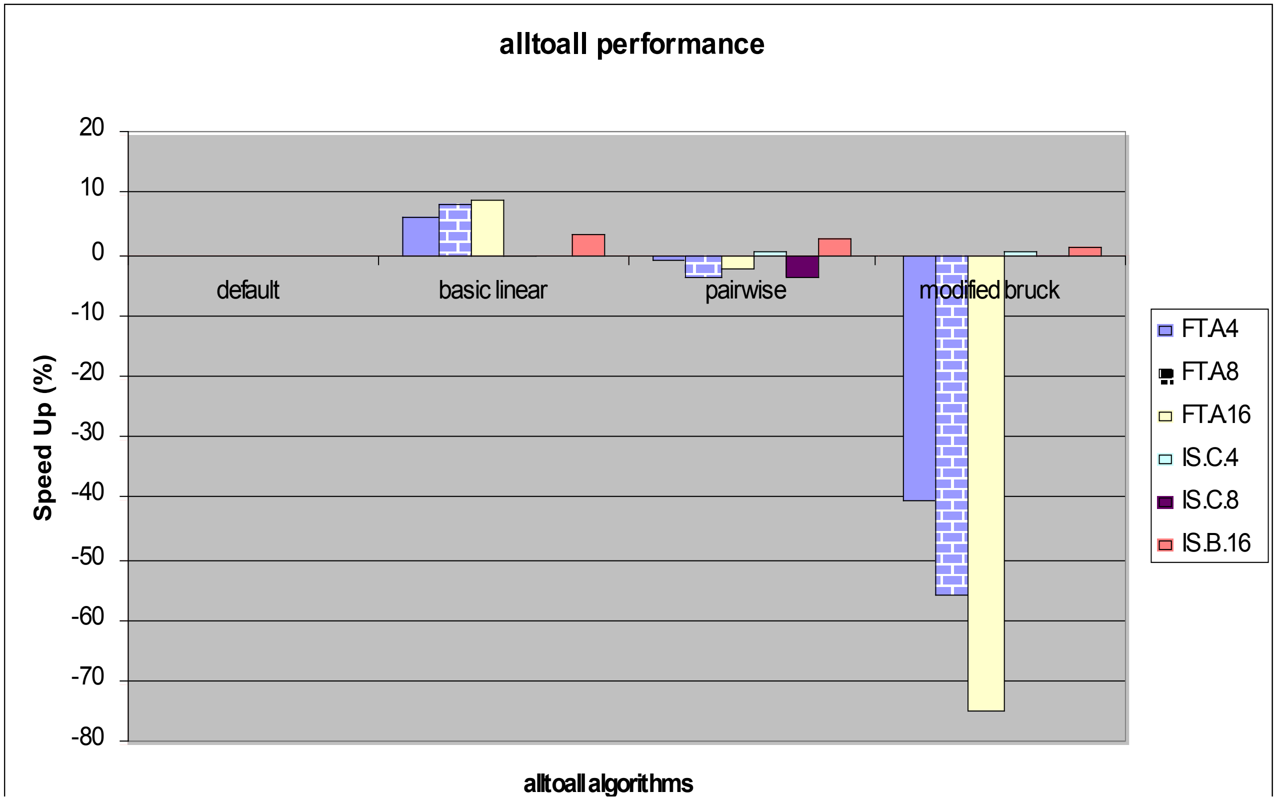
TCP Buffer Size Effect on NPB



Target system: Hamlet (Dell IA-32 P4 nodes) clusters in Purdue RAC

Used MPI: MPICH1

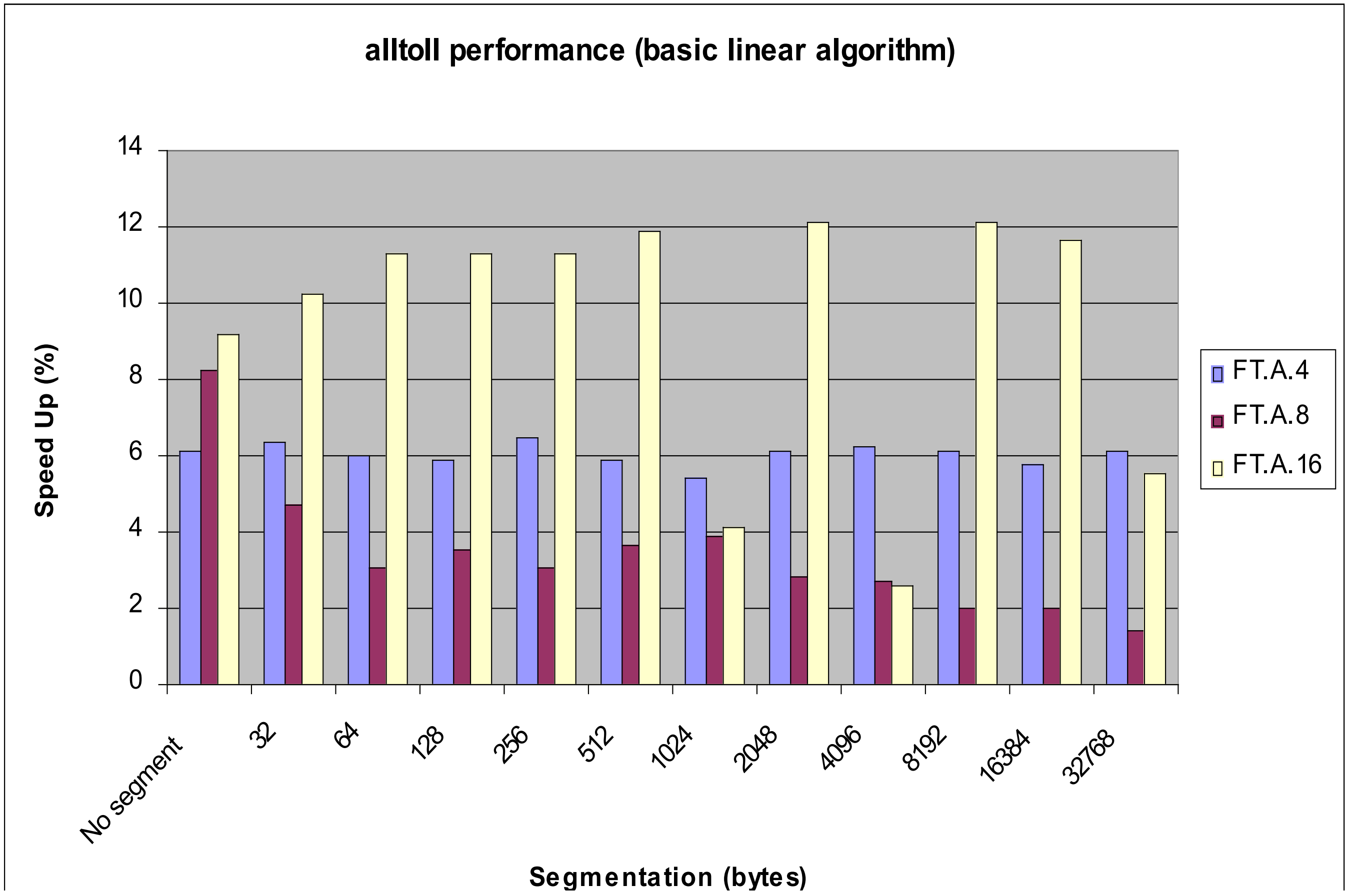
Alltoall collective call performance



Target system: Hamlet (Dell IA-32 P4 nodes) clusters in Purdue RAC

Used MPI: Open MPI 1.2.2

Segmentation Effect on Basic Linear Alltoall Algorithm



Target system: Hamlet (Dell IA-32 P4 nodes) clusters in Purdue RAC

Used MPI: Open MPI 1.2.2

Automatic Tuning for Multicore

- Starting point was the Polaris compiler
  - 200 switches
- Early results on dynamic serialization
- Goal: parallelizing compiler that never lowers the performance of a program
- OpenMP to MPI translation
- Tuning NICA architectures
  - Multicore + <u>ni</u>che <u>ca</u>pabilities (accelerators and more)

Conclusions and Discussion

Dynamic Adaptation is one of the most exciting research topics

There are still issues to Sink your Teeth in

- Runtime overhead: when to shelter/re-tune
- Fine-grain tuning
- Model-guided pruning of search space
- Architecture of an autotuner
  - If we could agree, we could plug-in our modules
- AutoAuto - autotuning autoparallelizer
- How to get order(s) of magnitude improvement
  - Wanted: tuning parameters and their performance effects

Tuning Speculative Secction Selection

Benchmark	Single Thread	[Vijay Micro 98]	[Johnson PLDI 04]	[Johnson PPoPP 07]	
	IPC		Min-Cut	Greedy	Hierarchical
bzip2	0.70	1.01	1.09	1.07	1.17
gzip	0.72	1.27	1.35	1.11	1.17
mcf	0.07	1.01	1.63	1.07	1.09
parser	0.51	0.87	1.24	1.20	1.18
vpr	0.63	1.38	1.09	1.38	1.38
geometric mean		1.09	1.27	1.16	1.19

Speedup factors