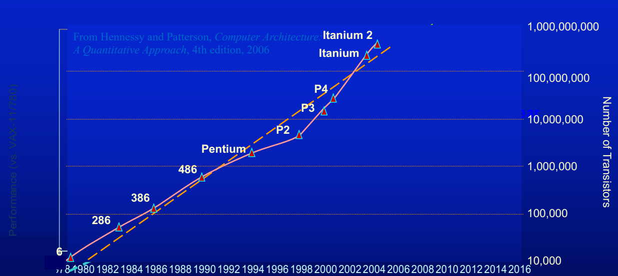
6.1100

Parallelization

Outline

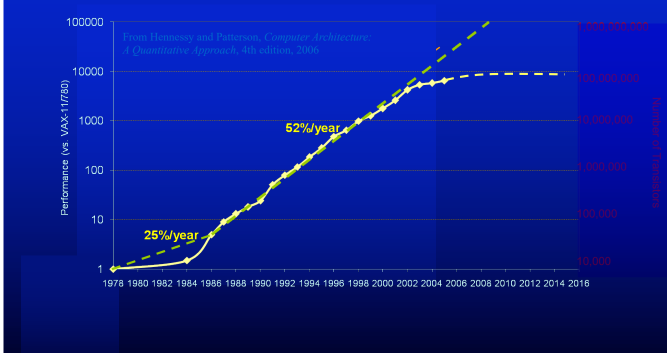
- **Why Parallelism**
- Parallel Execution
- Parallelizing Compilers
- Dependence Analysis
- Increasing Parallelization Opportunities

Moore's Law



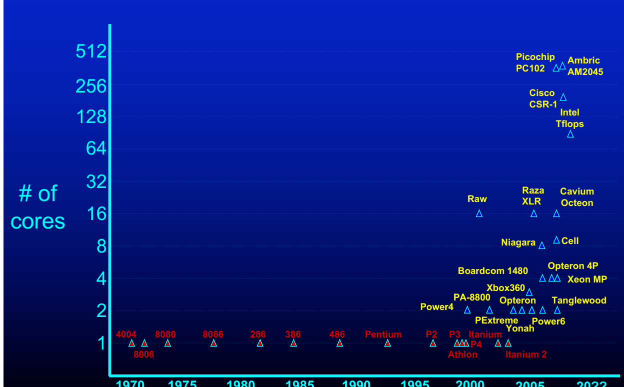
From David Patterson

Uniprocessor Performance (SPECint)



From David Patterson

Multicores Are Here!



Issues with Parallelism

- Amdhal's Law
  - Any computation can be analyzed in terms of a portion that must be executed sequentially, Ts, and a portion that can be executed in parallel, Tp. Then for n processors:
  - T(n) = Ts + Tp/n
  - T(∞) = Ts, thus maximum speedup (Ts + Tp) /Ts
- Load Balancing
  - The work is distributed among processors so that ***all*** processors are kept busy when parallel task is executed.
- Granularity
  - The size of the parallel regions between synchronizations or the ratio of computation (useful work) to communication (overhead).

Outline

- Why Parallelism
- **Parallel Execution**
- Parallelizing Compilers
- Dependence Analysis
- Increasing Parallelization Opportunities

Types of Parallelism

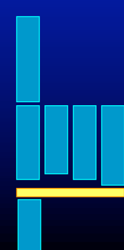
- Instruction Level Parallelism (ILP) → Scheduling and Hardware
- Task Level Parallelism (TLP) → Mainly by hand
- Loop Level Parallelism (LLP) or Data Parallelism → Hand or Compiler Generated
- Pipeline Parallelism → Hardware or Streaming
- Divide and Conquer Parallelism → Recursive functions

Why Loops?

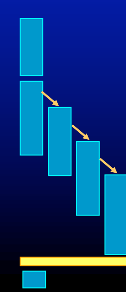
- 90% of the execution time in 10% of the code
  - Mostly in loops
- If parallel, can get good performance
  - Load balancing
- Relatively easy to analyze

Programmer Defined Parallel Loop

- FORALL
  - No "loop carried dependences"
  - Fully parallel



- FORACROSS
  - Some "loop carried dependences"



Parallel Execution

- Example

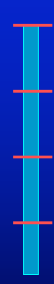
```
FORPAR I = 0 to N
   A[I] = A[I] + 1
```

- Block Distribution: Program gets mapped into

```
Iters = ceiling(N/NUMPROC);
FOR P = 0 to NUMPROC-1
   FOR I = P*Iters to MIN((P+1)*Iters, N)
     A[I] = A[I] + 1
```

- SPMD (Single Program, Multiple Data) Code

```
If(myPid == 0) {
   …
   Iters = ceiling(N/NUMPROC);
}
Barrier();
FOR I = myPid*Iters to MIN((myPid+1)*Iters, N)
   A[I] = A[I] + 1
Barrier();
```



Parallel Execution

- Example

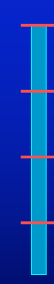
```
FORPAR I = 0 to N
   A[I] = A[I] + 1
```

- Block Distribution: Program gets mapped into

```
Iters = ceiling(N/NUMPROC);
FOR P = 0 to NUMPROC-1
   FOR I = P*Iters to MIN((P+1)*Iters, N)
     A[I] = A[I] + 1
```

- Code fork a function

```
Iters = ceiling(N/NUMPROC);
FOR P = 0 to NUMPROC – 1 { ParallelExecute(func1, P); }
BARRIER(NUMPROC);
void func1(integer myPid)
{
   FOR I = myPid*Iters to MIN((myPid+1)*Iters, N)
     A[I] = A[I] + 1
}
```



Parallel Thread Basics

- Create separate threads
  - Create an OS thread
    - (hopefully) it will be run on a separate core
  - pthread_create(&thr, NULL, &entry_point, NULL)
  - Overhead in thread creation
    - Create a separate stack
    - Get the OS to allocate a thread
- Thread pool
  - Create all the threads (= num cores) at the beginning
  - Keep N-1 idling on a barrier, while sequential execution
  - Get them to run parallel code by each executing a function
  - Back to the barrier when parallel region is done

Outline

- Why Parallelism
- Parallel Execution
- **Parallelizing Compilers**
- Dependence Analysis
- Increasing Parallelization Opportunities

Parallelizing Compilers

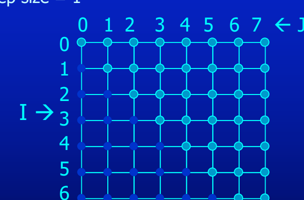
- Finding FORALL Loops out of FOR loops
- Examples

```
FOR I = 0 to 5
  A[I] = A[I] + 1

FOR I = 0 to 5
  A[I] = A[I+6] + 1

For I = 0 to 5
  A[2*I] = A[2*I + 1] + 1
```

Iteration Space

- N deep loops → N-dimensional discrete iteration space
  - Normalized loops: assume step size = 1

```
FOR I = 0 to 6
  FOR J = I to 7
```

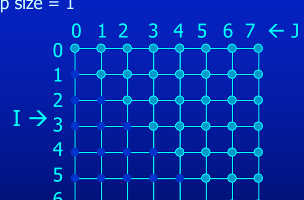


- Iterations are represented as coordinates in iteration space
  - $\bar{I} = [i_1, i_2, i_3, \ldots, i_n]$

Iteration Space

- N deep loops → N-dimensional discrete iteration space
  - Normalized loops: assume step size = 1

```
FOR I = 0 to 6
  FOR J = I to 7
```



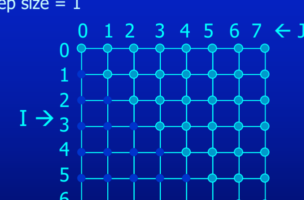
- Iterations are represented as coordinates in iteration space
- Sequential execution order of iterations → Lexicographic order

[0,0], [0,1], [0,2], …, [0,6], [0,7],
[1,1], [1,2], …, [1,6], [1,7],
[2,2], …, [2,6], [2,7],
………
[6,6], [6,7],

Iteration Space

- N deep loops → N-dimensional discrete iteration space
  - Normalized loops: assume step size = 1

```
FOR I = 0 to 6
  FOR J = I to 7
```

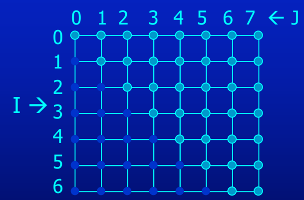


- Iterations are represented as coordinates in iteration space
- Sequential execution order of iterations → Lexicographic order
- Iteration $\bar{I}$ is lexicograpically less than $\bar{J}$, $\bar{I} < \bar{J}$ iff there exists c s.t. $i_1 = j_1, i_2 = j_2, \ldots i_{c-1} = j_{c-1}$ and $i_c < j_c$

Iteration Space

- N deep loops → N-dimensional discrete iteration space
  - Normalized loops: assume step size = 1

```
FOR I = 0 to 6
   FOR J = I to 7
```

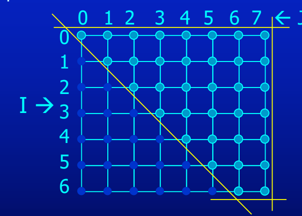


- An affine loop nest
  - Loop bounds are integer linear functions of constants, loop constant variables and outer loop indexes
  - Array accesses are integer linear functions of constants, loop constant variables and loop indexes

Iteration Space

- N deep loops → N-dimensional discrete iteration space
  - Normalized loops: assume step size = 1

```
FOR I = 0 to 6
   FOR J = I to 7
```



- Affine loop nest → Iteration space as a set of linear inequalities

  $0 \le I$

  $I \le 6$

  $I \le J$

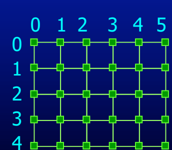
  $J \le 7$

Data Space

- M dimensional arrays → M-dimensional discrete cartesian space
  - a hypercube

```
Integer A(10)
```


```
Float B(5, 6)
```



Dependences

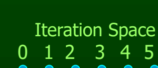
- True dependence
```
a =
  = a
```
- Anti dependence
```
  = a
a =
```
- Output dependence
```
a =
a =
```
- Definition:
  Data dependence exists for a dynamic instance i and j iff
  - either i or j is a write operation
  - i and j refer to the same variable
  - i executes before j
- How about array accesses within loops?

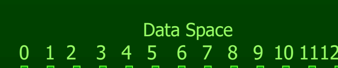
Outline

- Why Parallelism
- Parallel Execution
- Parallelizing Compilers
- **Dependence Analysis**
- Increasing Parallelization Opportunities

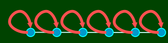
Array Accesses in a loop

```
FOR I = 0 to 5
   A[I] = A[I] + 1
```

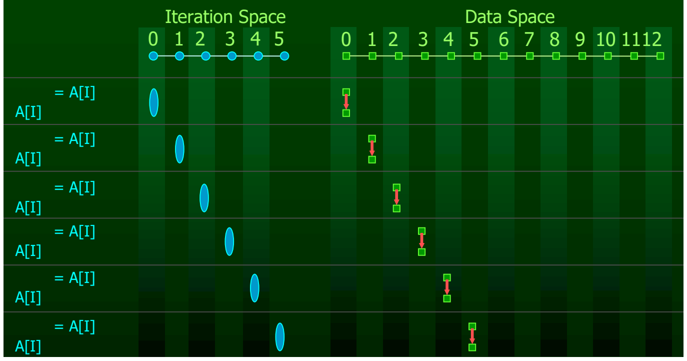




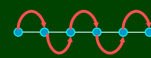
Array Accesses in a loop



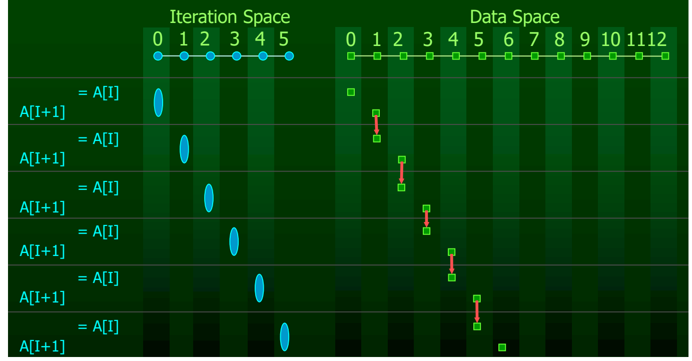
```
FOR I = 0 to 5
  A[I] = A[I] + 1
```



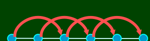
Array Accesses in a loop



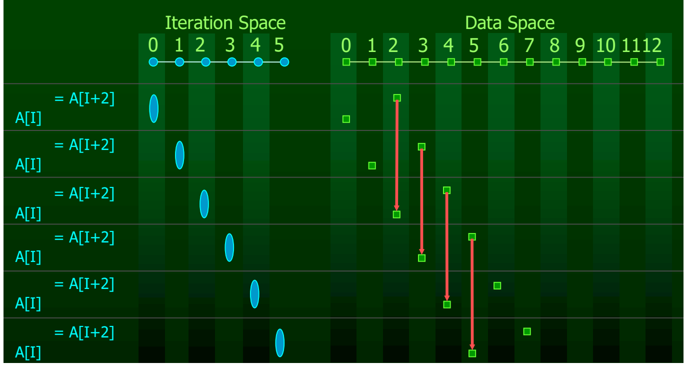
```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```



Array Accesses in a loop

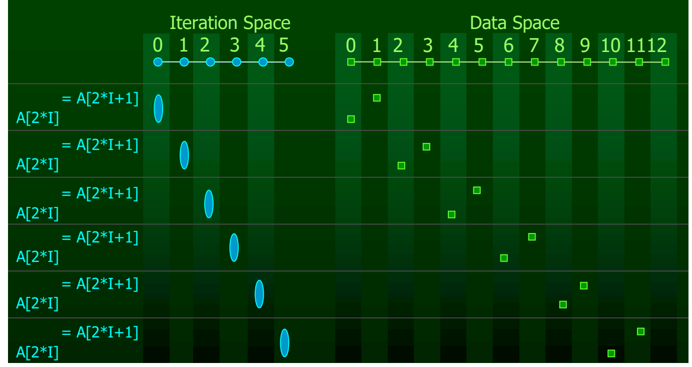


```
FOR I = 0 to 5
  A[I] = A[I+2] + 1
```



Array Accesses in a loop

```
FOR I = 0 to 5
  A[2*I] = A[2*I+1] + 1
```



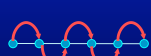
Distance Vectors

- A loop has a distance d if there exist a data dependence from iteration i to j and d = j-i

$dv = [0]$

```
FOR I = 0 to 5
  A[I] = A[I] + 1
```

$dv = [1]$



```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

$dv = [2]$

```
FOR I = 0 to 5
  A[I] = A[I+2] + 1
```

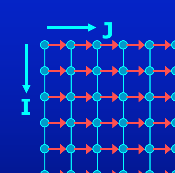
$dv = [1], [2] \ldots = [*]$

```
FOR I = 0 to 5
  A[I] = A[0] + 1
```

Multi-Dimensional Dependence

```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I, J-1] + 1
```

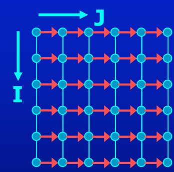
$dv = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$



Multi-Dimensional Dependence

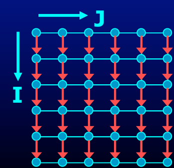
```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I, J-1] + 1
```

$$dv = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$




```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I+1, J] + 1
```

$$dv = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

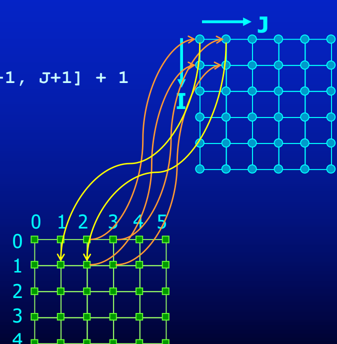


Outline

- Dependence Analysis
- Increasing Parallelization Opportunities

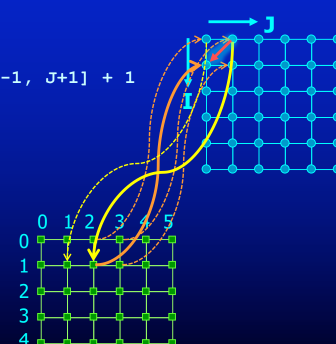
What is the Dependence?

```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I-1, J+1] + 1
```



What is the Dependence?

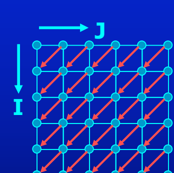
```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I-1, J+1] + 1
```



What is the Dependence?

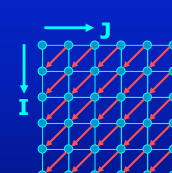
```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I-1, J+1] + 1
```

$$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

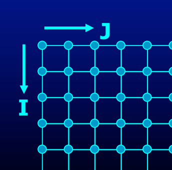


What is the Dependence?

```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I-1, J+1] + 1
```



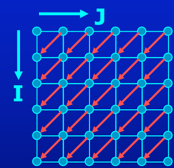

```
FOR I = 1 to n
  FOR J = 1 to n
    B[I] = B[I-1] + 1
```



What is the Dependence?

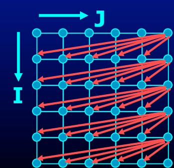
```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I-1, J+1] + 1
```

$$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$



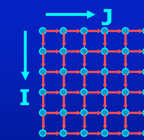

```
FOR I = 1 to n
  FOR J = 1 to n
    B[I] = B[I-1] + 1
```

$$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \begin{bmatrix} 1 \\ -3 \end{bmatrix}, \ldots = \begin{bmatrix} 1 \\ * \end{bmatrix}$$



What is the Dependence?

```
FOR i = 1 to N-1
  FOR j = 1 to N-1
    A[i,j] = A[i,j-1] + A[i-1,j];
```



$$dv = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Recognizing FORALL Loops

- Find data dependences in loop
  - For every pair of array acceses to the same array
    If the first access has at least one dynamic instance (an iteration) in which it refers to a location in the array that the second access also refers to in at least one of the later dynamic instances (iterations).
    Then there is a data dependence between the statements
  - (Note that same array can refer to itself – output dependences)
- Definition
  - Loop-carried dependence:
    dependence that crosses a loop boundary
- If there are no loop carried dependences → parallelizable

Data Dependence Analysis

- I: Distance Vector method
- II: Integer Programming

Distance Vector Method

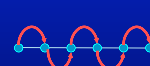
- The i^{th} loop is parallelizable for all dependence $d = [d_1,\ldots,d_i,..d_n]$ either
  one of $d_1,\ldots,d_{i-1}$ is > 0
  or
  all $d_1,\ldots,d_i = 0$

Is the Loop Parallelizable?

$dv = [0]$ **Yes**

```
FOR I = 0 to 5
  A[I] = A[I] + 1
```

$dv = [1]$ **No**



```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

$dv = [2]$ **No**

```
FOR I = 0 to 5
  A[I] = A[I+2] + 1
```

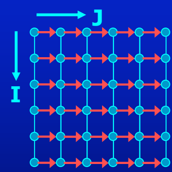
$dv = [*]$ **No**

```
FOR I = 0 to 5
  A[I] = A[0] + 1
```

Are the Loops Parallelizable?

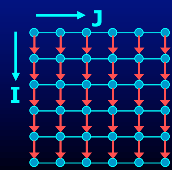
```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I, J-1] + 1
```

$dv = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ **Yes** **No**




```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I+1, J] + 1
```

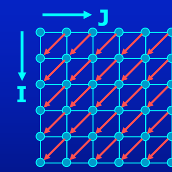
$dv = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ **No** **Yes**



Are the Loops Parallelizable?

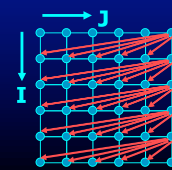
```
FOR I = 1 to n
  FOR J = 1 to n
    A[I, J] = A[I-1, J+1] + 1
```

$dv = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ **No** **Yes**




```
FOR I = 1 to n
  FOR J = 1 to n
    B[I] = B[I-1] + 1
```

$dv = \begin{bmatrix} 1 \\ * \end{bmatrix}$ **No** **Yes**



Integer Programming Method

- Example

```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

- Is there a loop-carried dependence between A[I+1] and A[I]
  - Are there two distinct iterations i_w and i_r such that A[i_w+1] is the same location as A[i_r]
  - $\exists$ integers i_w, i_r $0 \le i_w, i_r \le 5$ $i_w \ne i_r$ $i_w + 1 = i_r$
- Is there a dependence between A[I+1] and A[I+1]
  - Are there two distinct iterations i_1 and i_2 such that A[i_1+1] is the same location as A[i_2+1]
  - $\exists$ integers i_1, i_2 $0 \le i_1, i_2 \le 5$ $i_1 \ne i_2$ $i_1 + 1 = i_2 + 1$

Integer Programming Method

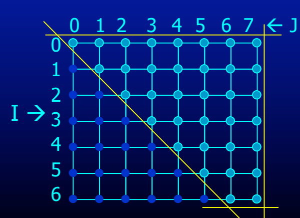
```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

- Formulation
  - $\exists$ an integer vector $\bar{\imath}$ such that $\hat{A}\bar{\imath} \le \bar{b}$ where $\hat{A}$ is an integer matrix and $\bar{b}$ is an integer vector

Iteration Space

```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

- N deep loops → n-dimensional discrete cartesian space
- Affine loop nest → Iteration space as a set of linear inequalities
  - $0 \le I$
  - $I \le 6$
  - $I \le J$
  - $J \le 7$



Integer Programming Method

```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

- Formulation
  - $\exists$ an integer vector $\bar{\imath}$ such that $\hat{A}\bar{\imath} \le \bar{b}$ where $\hat{A}$ is an integer matrix and $\bar{b}$ is an integer vector
- Our problem formulation for A[i] and A[i+1]
  - $\exists$ integers i_w, i_r $0 \le i_w, i_r \le 5$ $i_w \ne i_r$ $i_w + 1 = i_r$
  - $i_w \ne i_r$ is not an affine function
    - divide into 2 problems
    - Problem 1 with $i_w < i_r$ and problem 2 with $i_r < i_w$
    - If either problem has a solution → there exists a dependence
  - How about $i_w + 1 = i_r$
    - Add two inequalities to single problem $i_w + 1 \le i_r$, and $i_r \le i_w + 1$

Integer Programming Formulation

```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

- Problem 1
  - $0 \le i_w$
  - $i_w \le 5$
  - $0 \le i_r$
  - $i_r \le 5$
  - $i_w < i_r$
  - $i_w + 1 \le i_r$
  - $i_r \le i_w + 1$

Integer Programming Formulation

```
FOR I = 0 to 5
  A[I+1] = A[I] + 1
```

- Problem 1
  - $0 \le i_w \rightarrow -i_w \le 0$
  - $i_w \le 5 \rightarrow i_w \le 5$
  - $0 \le i_r \rightarrow -i_r \le 0$
  - $i_r \le 5 \rightarrow i_r \le 5$
  - $i_w < i_r \rightarrow i_w - i_r \le -1$
  - $i_w + 1 \le i_r \rightarrow i_w - i_r \le -1$
  - $i_r \le i_w + 1 \rightarrow -i_w + i_r \le 1$

Integer Programming Formulation

- Problem 1

			$\hat{A}$		$\bar{b}$
$0 \le i_w$	→	$-i_w \le 0$	-1	0	0
$i_w \le 5$	→	$i_w \le 5$	1	0	5
$0 \le i_r$	→	$-i_r \le 0$	0	-1	0
$i_r \le 5$	→	$i_r \le 5$	0	1	5
$i_w < i_r$	→	$i_w - i_r \le -1$	1	-1	-1
$i_w + 1 \le i_r$	→	$i_w - i_r \le -1$	1	-1	-1
$i_r \le i_w + 1$	→	$-i_w + i_r \le 1$	-1	1	1

- and problem 2 with $i_r < i_w$

Generalization

- An affine loop nest

```
FOR i1 = f11(c1…ck) to Iu1(c1…ck)
  FOR i2 = f12(i1,c1…ck) to Iu2(i1,c1…ck)
    ……
      FOR in = f1n(i1…in-1,c1…ck) to Iun(i1…in-1,c1…ck)
         A[fa1(i1…in,c1…ck), fa2(i1…in,c1…ck),…,fam(i1…in,c1…ck)]
```

- Solve 2*n problems of the form
  - $i_1 = j_1,\ i_2 = j_2, \ldots\ i_{n-1} = j_{n-1},\ i_n < j_n$
  - $i_1 = j_1,\ i_2 = j_2, \ldots\ i_{n-1} = j_{n-1},\ j_n < i_n$
  - $i_1 = j_1,\ i_2 = j_2, \ldots\ i_{n-1} < j_{n-1}$
  - $i_1 = j_1,\ i_2 = j_2, \ldots\ j_{n-1} < i_{n-1}$
  - …………
  - $i_1 = j_1,\ i_2 < j_2$
  - $i_1 = j_1,\ j_2 < i_2$
  - $i_1 < j_1$
  - $j_1 < i_1$

Outline

- Why Parallelism
- Parallel Execution
- Parallelizing Compilers
- Dependence Analysis
- **Increasing Parallelization Opportunities**

Increasing Parallelization Opportunities

- Scalar Privatization
- Reduction Recognition
- Induction Variable Identification
- Array Privatization
- Loop Transformations
- Granularity of Parallelism
- Interprocedural Parallelization

Scalar Privatization

- Example

```
FOR i = 1 to n
  X = A[i] * 3;
  B[i] = X;
```

- Is there a loop carried dependence?
- What is the type of dependence?

Privatization

- Analysis:
  - Any anti- and output- loop-carried dependences
- Eliminate by assigning in local context

```
FOR i = 1 to n
  integer Xtmp;
  Xtmp = A[i] * 3;
  B[i] = Xtmp;
```

- Eliminate by expanding into an array

```
FOR i = 1 to n
  Xtmp[i] = A[i] * 3;
  B[i] = Xtmp[i];
```

Privatization

- Need a final assignment to maintain the correct value after the loop nest
- Eliminate by assigning in local context

```
FOR i = 1 to n
  integer Xtmp;
  Xtmp = A[i] * 3;
  B[i] = Xtmp;
  if(i == n) X = Xtmp
```

- Eliminate by expanding into an array

```
FOR i = 1 to n
  Xtmp[i] = A[i] * 3;
  B[i] = Xtmp[i];
X = Xtmp[n];
```

Another Example

- How about loop-carried true dependences?
- Example

```
FOR i = 1 to n
  X = X + A[i];
```

- Is this loop parallelizable?

Reduction Recognition

- Reduction Analysis:
  - Only associative operations
  - The result is never used within the loop
- Transformation

```
Integer Xtmp[NUMPROC];
Barrier();
FOR i = myPid*Iters to MIN((myPid+1)*Iters, n)
    Xtmp[myPid] = Xtmp[myPid] + A[i];
Barrier();
If(myPid == 0) {
  FOR p = 0 to NUMPROC-1
    X = X + Xtmp[p];
  …
```

Induction Variables

- Example

```
FOR i = 0 to N
  A[i] = 2^i;
```

- After strength reduction

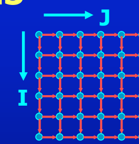
```
t = 1
FOR i = 0 to N
  A[i] = t;
  t = t*2;
```

- What happened to loop carried dependences?
- Need to do opposite of this!
  - Perform induction variable analysis
  - Rewrite IVs as a function of the loop variable

Array Privatization

- Similar to scalar privatization
- However, analysis is more complex
  - Array Data Dependence Analysis: Checks if two iterations access the same location
  - Array Data Flow Analysis: Checks if two iterations access the same value
- Transformations
  - Similar to scalar privatization
  - Private copy for each processor or expand with an additional dimension

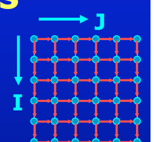
Loop Transformations



- A loop may not be parallel as is
- Example

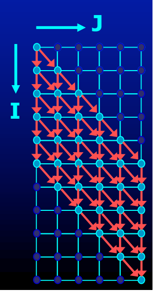
```
FOR i = 1 to N-1
  FOR j = 1 to N-1
    A[i,j] = A[i,j-1] + A[i-1,j];
```

Loop Transformations



- A loop may not be parallel as is
- Example

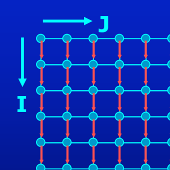
```
FOR i = 1 to N-1
  FOR j = 1 to N-1
    A[i,j] = A[i,j-1] + A[i-1,j];
```



- After loop Skewing $\begin{bmatrix} i_{new} \\ j_{new} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_{old} \\ j_{old} \end{bmatrix}$

```
FOR i = 1 to 2*N-3
  FORPAR j = max(1,i-N+2) to min(i, N-1)
    A[i-j+1,j] = A[i-j+1,j-1] + A[i-j,j];
```

Granularity of Parallelism



- Example

```
FOR i = 1 to N-1
  FOR j = 1 to N-1
    A[i,j] = A[i,j] + A[i-1,j];
```

- Gets transformed into

```
FOR i = 1 to N-1
  Barrier();
  FOR j = 1+ myPid*Iters to MIN((myPid+1)*Iters, n-1)
    A[i,j] = A[i,j] + A[i-1,j];
  Barrier();
```

- Inner loop parallelism can be expensive
  - Startup and teardown overhead of parallel regions
  - Lot of synchronization
  - Can even lead to slowdowns

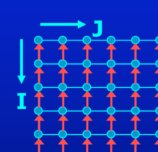
Granularity of Parallelism

- Inner loop parallelism can be expensive
- Solutions
  - Don't parallelize if the amount of work within the loop is too small

  or

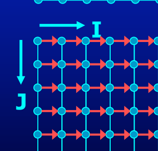
  - Transform into outer-loop parallelism

Outer Loop Parallelism



- Example

```
FOR i = 1 to N-1
  FOR j = 1 to N-1
    A[i,j] = A[i,j] + A[i-1,j];
```



- After Loop Transpose

```
FOR j = 1 to N-1
  FOR i = 1 to N-1
    A[i,j] = A[i,j] + A[i-1,j];
```

- Get mapped into

```
Barrier();
FOR j = 1+ myPid*Iters to MIN((myPid+1)*Iters, n-1)
  FOR i = 1 to N-1
    A[i,j] = A[i,j] + A[i-1,j];
Barrier();
```

Unimodular Transformations

- Interchange, reverse and skew
- Use a matrix transformation

  $I_{new} = A\ I_{old}$

- Interchange $\begin{bmatrix} i_{new} \\ j_{new} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i_{old} \\ j_{old} \end{bmatrix}$
- Reverse $\begin{bmatrix} i_{new} \\ j_{new} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_{old} \\ j_{old} \end{bmatrix}$
- Skew $\begin{bmatrix} i_{new} \\ j_{new} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i_{old} \\ j_{old} \end{bmatrix}$

Legality of Transformations

- Unimodular transformation with matrix A is valid iff. For all dependence vectors v

  the first non-zero in Av is positive
- Example

```
FOR i = 1 to N-1
  FOR j = 1 to N-1
    A[i,j] = A[i,j] + A[i-1,j];
```

$dv = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

- Interchange $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ ✓
- Reverse $A = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$ $\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$ ✗
- Skew $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ ✓✓

Interprocedural Parallelization

- Function calls will make a loop unparallelizatble
  - Reduction of available parallelism
  - A lot of inner-loop parallelism
- Solutions
  - Interprocedural Analysis
  - Inlining

Interprocedural Parallelization

- Issues
  - Same function reused many times
  - Analyze a function on each trace → Possibly exponential
  - Analyze a function once → unrealizable path problem
- Interprocedural Analysis
  - Need to update all the analysis
  - Complex analysis
  - Can be expensive
- Inlining
  - Works with existing analysis
  - Large code bloat → can be very expensive

```
HashSet h;
for i = 1 to n
    int v = compute(i);
    h.insert(i);
```

Are iterations independent?

Can you still execute the loop in parallel?

Do all parallel executions give same result?

Summary

- Multicores are here
  - Need parallelism to keep the performance gains
  - Programmer defined or compiler extracted parallelism
- Automatic parallelization of loops with arrays
  - Requires Data Dependence Analysis
  - Iteration space & data space abstraction
  - An integer programming problem
- Many optimizations that'll increase parallelism