6.035

Fall 2006

Parallelization

Saman Amarasinghe
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Multicores Are Here!

of cores

512
256
128
64
32
16
8
4
2
1

1970
1975
1980
1985
1990
1995
2000
2005
20??

4004
8008
8080
8086
286
386
486
Pentium
P2
P3
Athlon
P4
Itanium
Itanium 2
Power4
PA-8800
PExtreme
Opteron
Yonah
Xbox360
Power6
Tanglewood
Boardcom 1480
Opteron 4P
Xeon MP
Niagara
Cell
Raw
Raza XLR
Cavium Octeon
Intel Tflops
Cisco CSR-1
Picochip PC102
Ambric AM2045

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).

Types of Parallelism

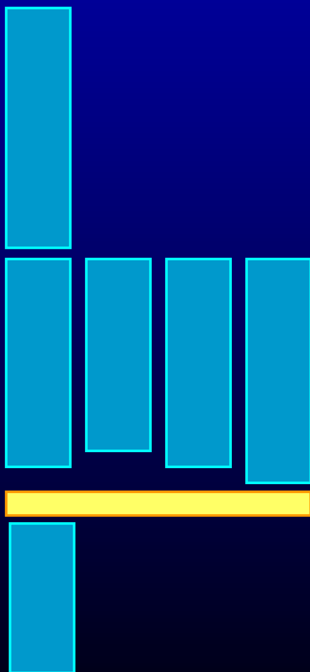
• Instruction Level Parallelism (ILP)	→ Scheduling and Hardware
• Task Level Parallelism (TLP)	→ Mainly by hand
• Loop Level Parallelism (LLP) or Data Parallelism	→ Hand or Compiler Generated
• Pipeline Parallelism	→ Hardware or Streaming
• Divide and Conquer Parallelism	→ Recursive functions

Why Loops?

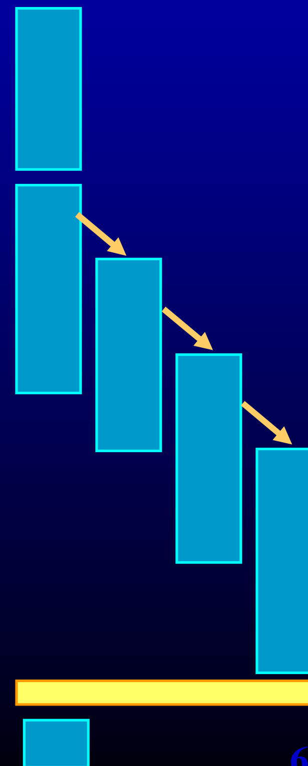
- 90% of the execution time in 10% of the code
  – Mostly in loops
- If parallel, can get good performance
  – Load balancing
- Relatively easy to analyze

Programmer Defined Parallel Loop

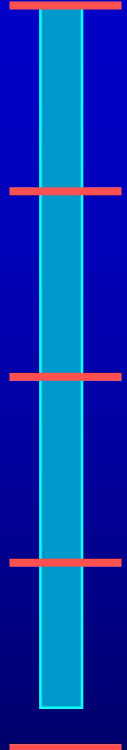
- FORALL
  - No "loop carried dependences"
  - Fully parallel



- 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

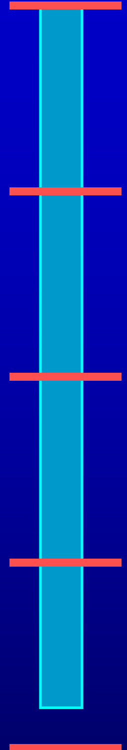
```
FORPAR I = 0 to N
   A[I] = A[I] + 1
```

- 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 that fork a function

```
Iters = ceiling(N/NUMPROC);
ParallelExecute(func1);
…
Func1(inteter myPid)
{
   FOR I = myPid*Iters to MIN((myPid+1)*Iters, N)
     A[I] = A[I] + 1
}
```



Parallel Execution

- SPMD
  - Need to get all the processors go through the control flow
    - Extra synchronization overhead or redundant computation on all processors or both
- Fork
  - Local variables not visible within the function
    - Either make the variables used/defined in the loop body global or pass and return them as arguments
    - Function call overhead

Parallelizing Compilers

- Finding FORALL Loops out of FOR loops

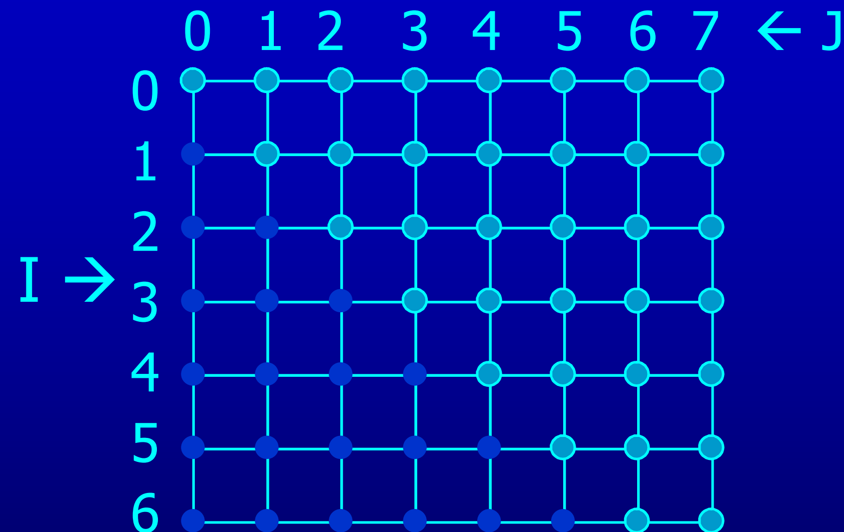
- Examples

```
FOR I = 0 to 5
  A[I+1] = 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 cartesian space
  - Normalized loops: assume step size = 1

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



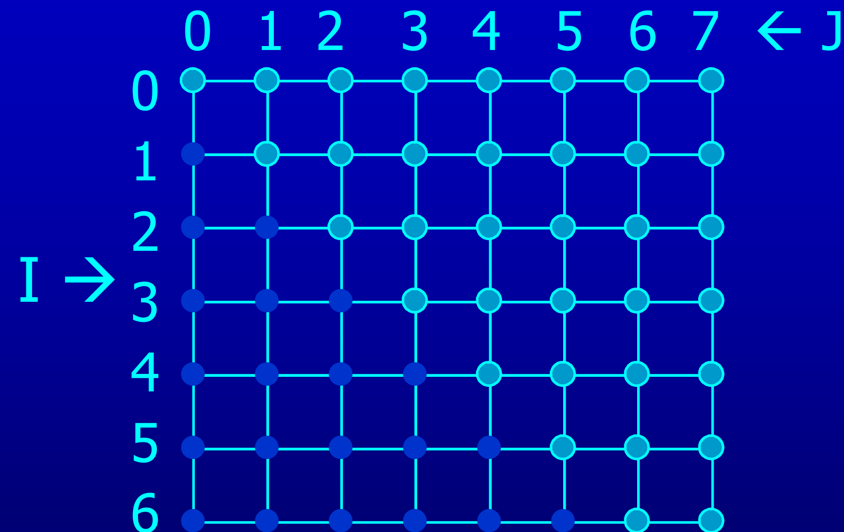
- Iterations are represented as coordinates in iteration space
- Sequential execution order of iterations → Lexicographic order

[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 cartesian space
  - Normalized loops: assume step size = 1

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

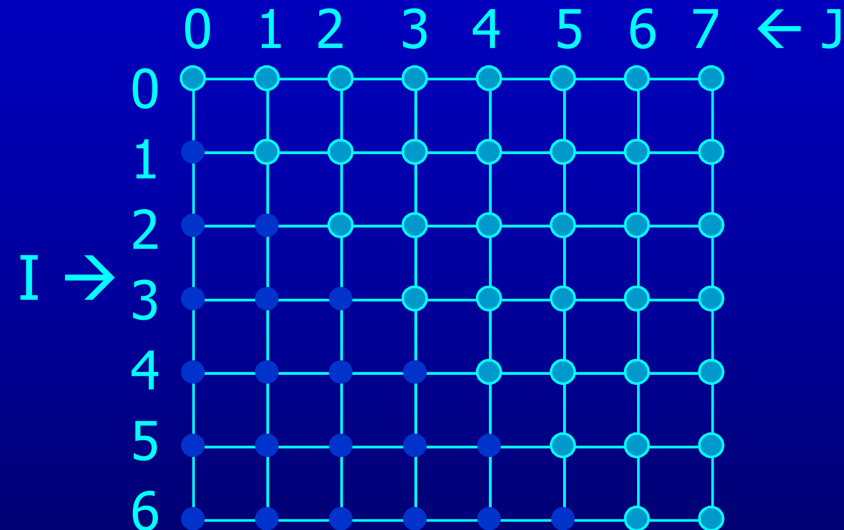


- Iterations are represented as coordinates in iteration space
- Sequential execution order of iterations → Lexicographic order
- Iteration $\bar{i}$ is lexicograpically less than $\bar{j}$, $\bar{i} < \bar{j}$ iff there exists c s.t. $i_1 = j_1$, $i_2 = j_2$,… $i_{c-1} = j_{c-1}$ and $i_c < j_c$

Iteration Space

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

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

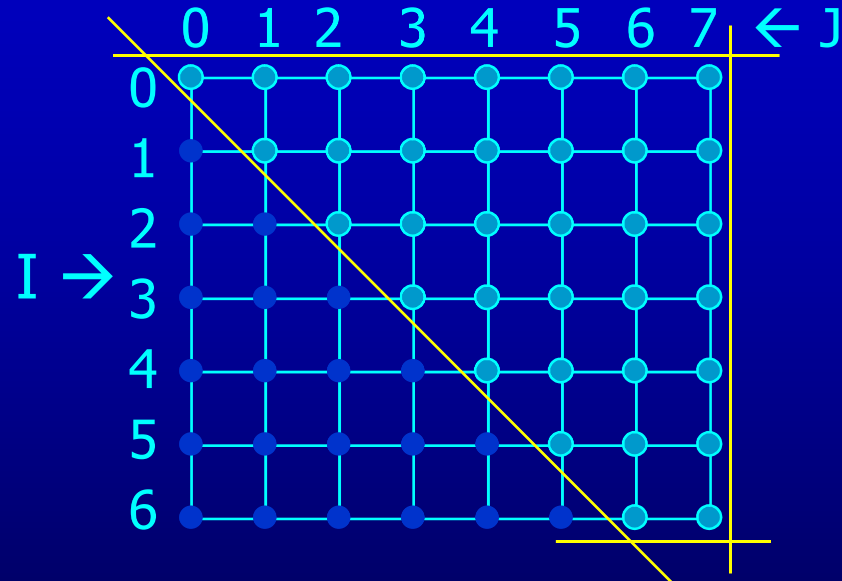


- An affine loop nest
  - Loop bounds are integer linear functions of constants, loop constant variables and outer loop indexes
  - Array accesses are integer linear functions of constants, loop constant variables and loop indexes

Iteration Space

- N deep loops → n-dimensional discrete cartesian 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 liner inequalities

$0 \leq I$

$I \leq 6$

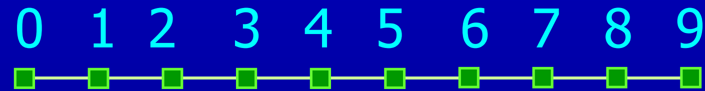
$I \leq J$

$J \leq 7$

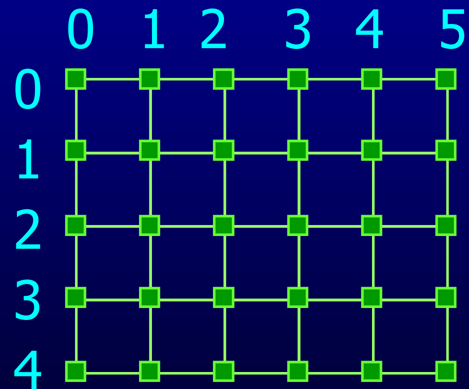
Data Space

- 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  =
```

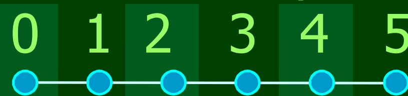
- Output dependence

```
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?

Array Accesses in a loop

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



= A[I]
A[I]

= A[I]
A[I]

= A[I]
A[I]

= A[I]
A[I]

= A[I]
A[I]

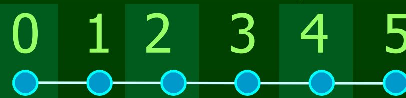
= A[I]
A[I]

Array Accesses in a loop

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

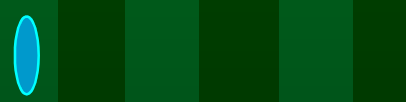
Iteration Space

Data Space



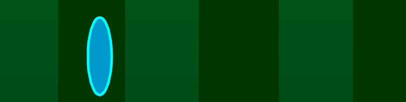
= A[I]

A[I+1]



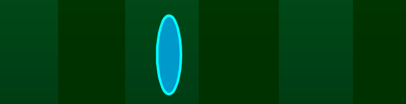
= A[I]

A[I+1]



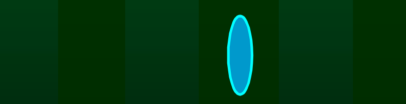
= A[I]

A[I+1]



= A[I]

A[I+1]

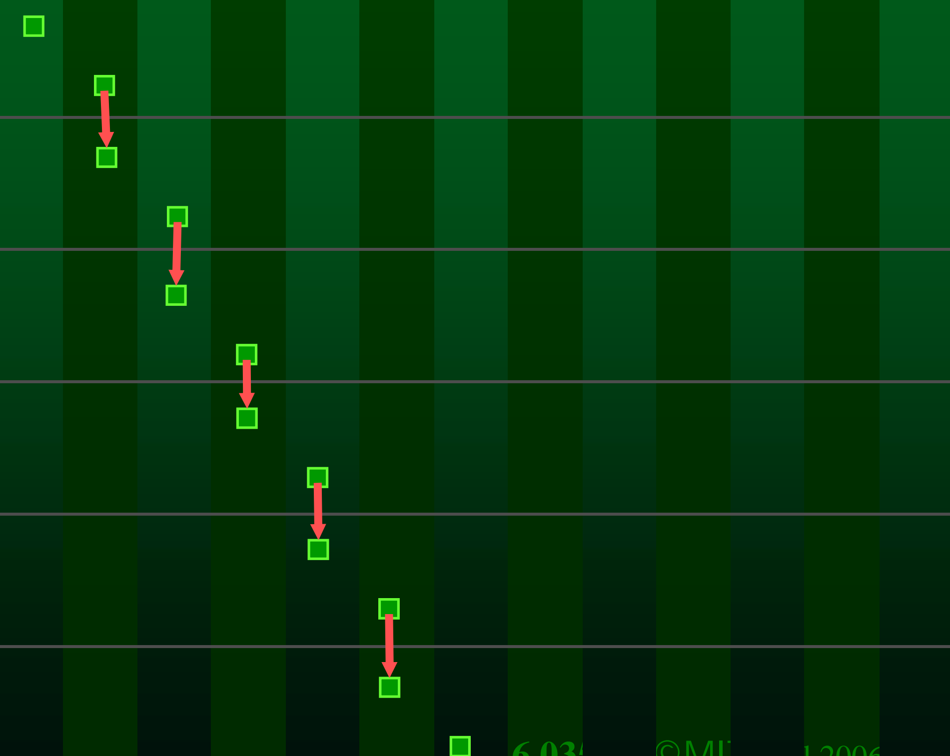


= A[I]

A[I+1]

= A[I]

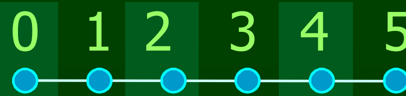
A[I+1]



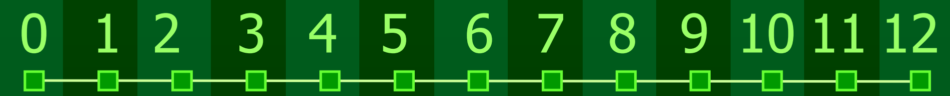
Array Accesses in a loop

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

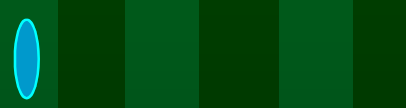
Iteration Space



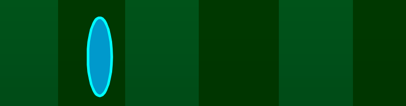
Data Space



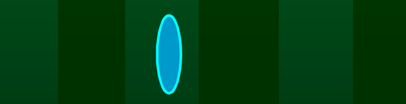
= A[I+2]
A[I]



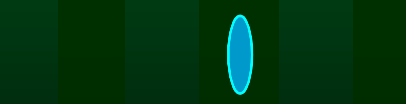
= A[I+2]
A[I]



= A[I+2]
A[I]

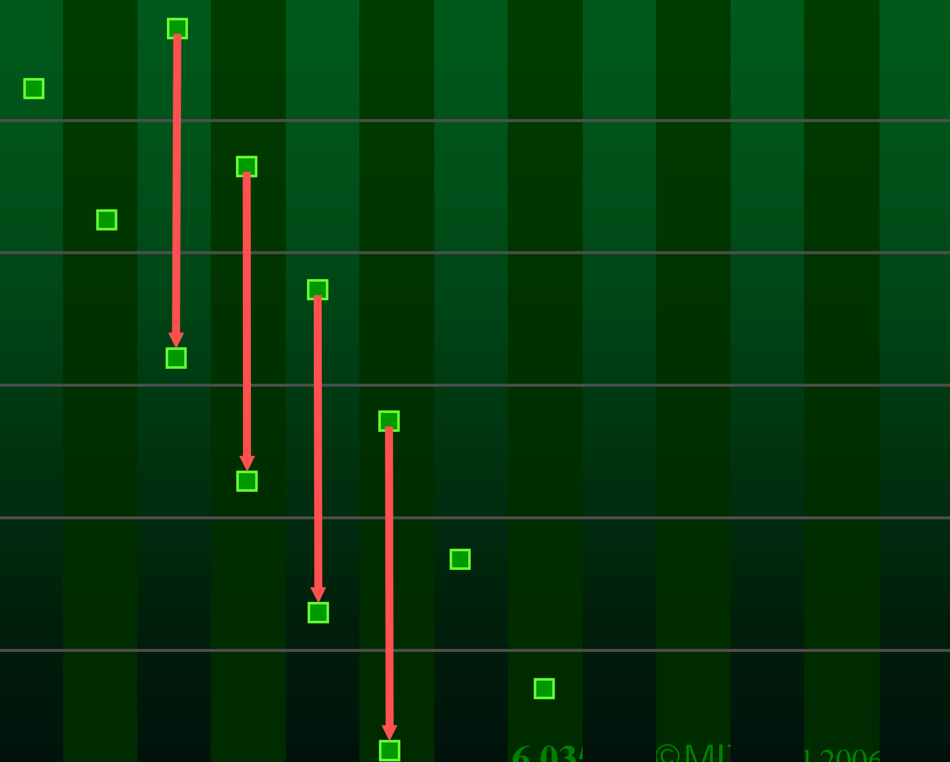


= A[I+2]
A[I]



= A[I+2]
A[I]

= A[I+2]
A[I]



Array Accesses in a loop

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

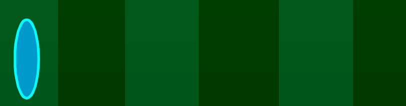
Iteration Space

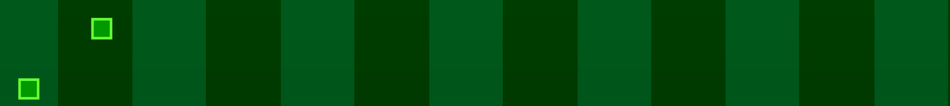
0 1 2 3 4 5

Data Space

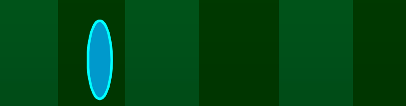
0 1 2 3 4 5 6 7 8 9 10 11 12

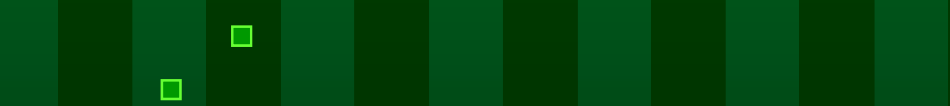
= A[2*+1]
A[2*I]



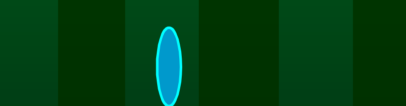


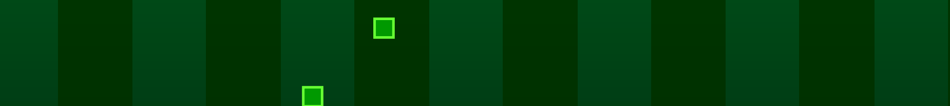
= A[2*I+1]
A[2*I]



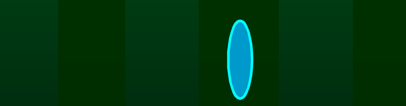


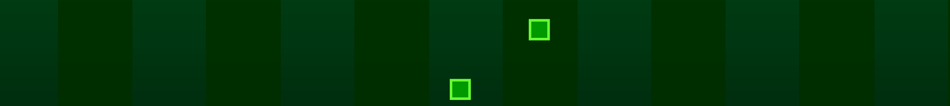
= A[2*I+1]
A[2*I]





= A[2*I+1]
A[2*I]





= A[2*I+1]
A[2*I]

= A[2*I+1]
A[2*I]

Recognizing FORALL Loops

- Find data dependences in loop
  - For every pair of array acceses to the same array
    If they may refer to the same location in at least one instance (one iteration)
    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

- 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]
  - Is 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 $\quad 0 \leq i_w, i_r \leq 5 \quad i_w \neq i_r \quad i_w + 1 = i_r$
- Is there a dependence between A[I+1] and A[I+1]
  - Is 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 $\quad 0 \leq i_1, i_2 \leq 5 \quad i_1 \neq i_2 \quad i_1 + 1 = i_2 + 1$

Integer Programming

- Formulation
  - $\exists$ an integer vector $\bar{i}$ such that $\hat{A}\,\bar{i} \leq \bar{b}$ 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 $0 \leq i_w, i_r \leq 5$ $i_w \neq i_r$ $i_w + 1 = i_r$
  - $i_w \neq i_r$ is not an affine function
    - divide into 2 problems
    - Problem 1 with $i_w < i_r$ and problem 2 with $i_r < i_w$
  - How about $i_w + 1 = i_r$
    - Add two inequalities to single problem
      $i_w + 1 \leq i_r$, and $i_r \leq i_w + 1$

Integer Programming Formulation

- Problem 1

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

- and problem 2 with $i_r < i_w$

Generalization

- An affine loop nest

```
FOR i_1 = f_11(c_1…c_k) to I_u1(c_1…c_k)
  FOR i_2 = f_12(i_1,c_1…c_k) to I_u2(i_1,c_1…c_k)
    ……
      FOR i_n = f_1n(i_1…i_n-1,c_1…c_k) to I_un(i_1…i_n-1,c_1…c_k)
            A[f_a1(i_1…i_n,c_1…c_k), f_a2(i_1…i_n,c_1…c_k),…,f_am(i_1…i_n,c_1…c_k)]
```

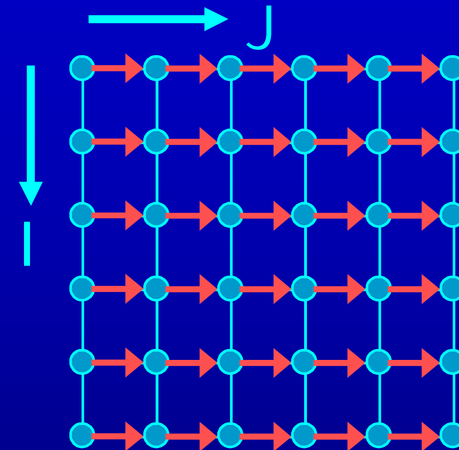
- Solve 2*n problems of the form
  - $i_1 = j_1, i_2 = j_2, \ldots\ldots i_{n-1} = j_{n-1}, i_n < j_n$
  - $i_1 = j_1, i_2 = j_2, \ldots\ldots i_{n-1} = j_{n-1}, j_n < i_n$
  - $i_1 = j_1, i_2 = j_2, \ldots\ldots i_{n-1} < j_{n-1}$
  - $i_1 = j_1, i_2 = j_2, \ldots\ldots j_{n-1} < i_{n-1}$

  ………………

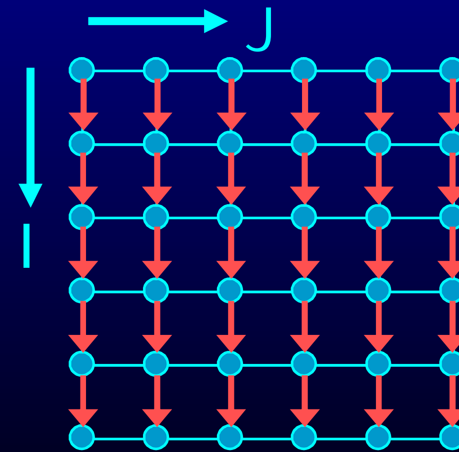
  - $i_1 = j_1, i_2 < j_2$
  - $i_1 = j_1, j_2 < i_2$
  - $i_1 < j_1$
  - $j_1 < i_1$

Multi-Dimensional Dependence

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

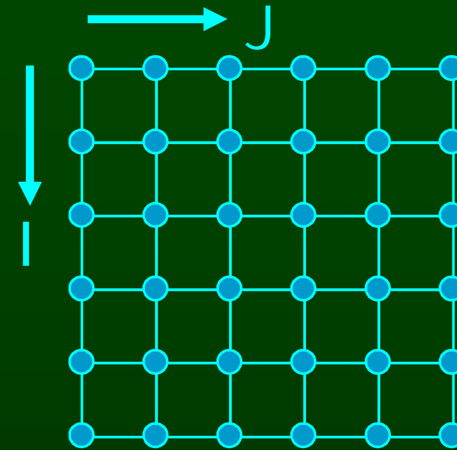



```
FOR I = 1 to n
  FOR J = 1 to n
   A[I][J] = A[I+1, J] + 1
```

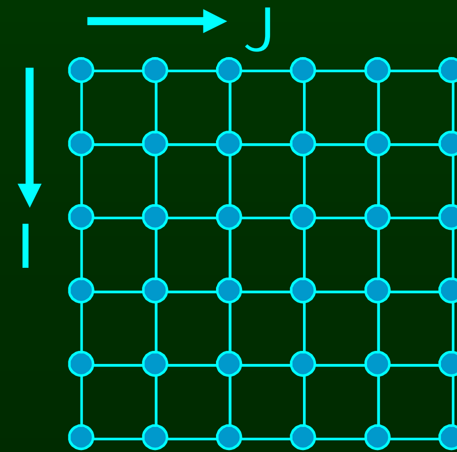


What is the Dependence?

```
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
   A[I][1] = A[I-1, 1] + 1
```



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

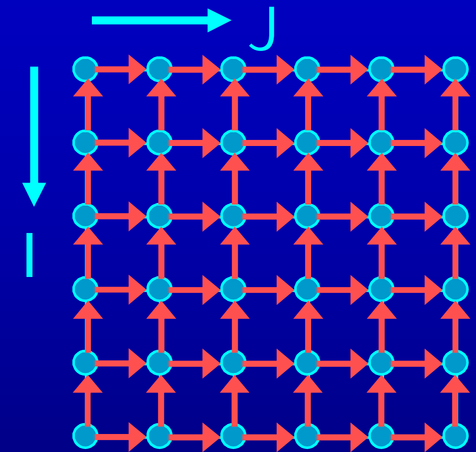
Array Privatization

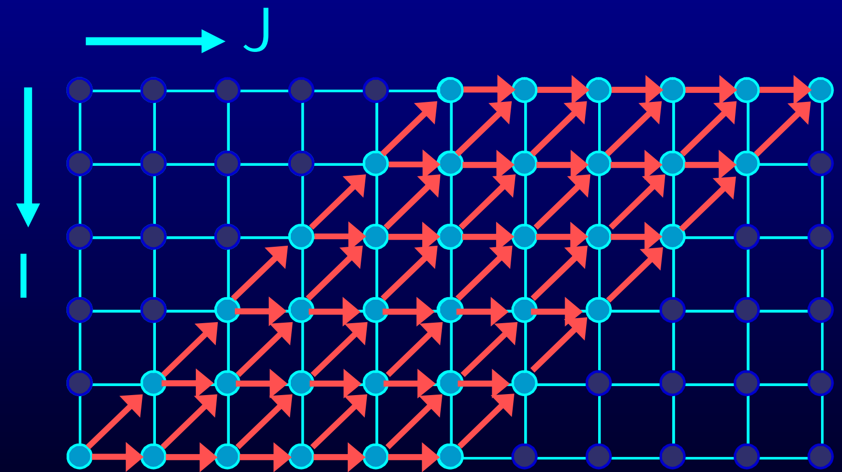
- Similar to scalar privatization
- However, analysis is more complex
  - Array Data Dependence Analysis: Checks if two iterations access the same location
  - Array Data Flow Analysis: Checks if two iterations access the same value
- Transformations
  - Similar to scalar privatization
  - Private copy 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];
```





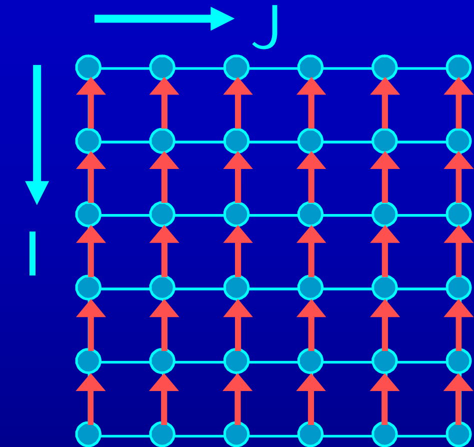
- After loop Skewing

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

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

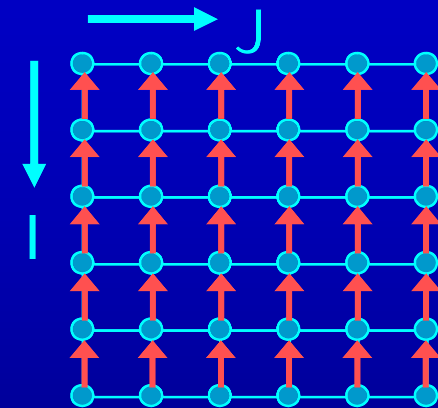
Granularity of Parallelism

- Inner loop parallelism can be expensive
  - May lead to slowdowns over sequential execution
- Solutions
  - Don't parallelize if the amount of work within the loop is too small
  - Transform into outer-loop parallelism

Outer Loop Parallelism

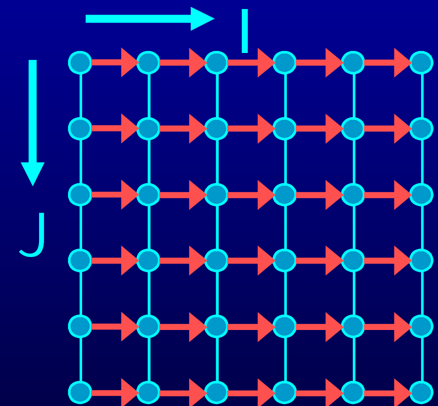
- Example

```
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();
```

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

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