Module 18: Loop Optimizations
Lecture 35: Amdahl's Law

The Lecture Contains:

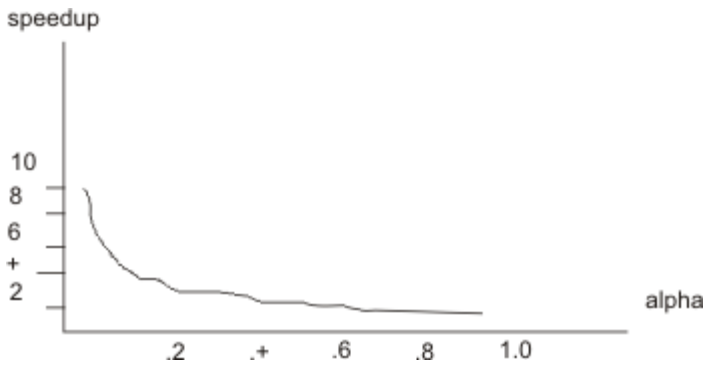
Previous Next

Loop Optimizations

Amdahl's Law

- Determines speed up
- a: fraction of code is scalar
- 1-a: fraction of code which is parallelizable
- 1 operation per unit time is scalar unit
- ζoperations per unit time in parallel units
- $$speedup = \frac{1}{\alpha + \frac{1-\alpha}{\zeta}}$$
- where 0 = a= 1, and ζ=1

a	ζ=10	ζ=20
0	10	20
0.1	5.26	6.9
0.25	3.08	3.48
0.5	1.82	1.90
1	1	1



- To achieve any significant speedup, parallel code must be greater then 90%
- Most of the execution time is spent in small sections of code. So concentrate on critical sections (loops)
- Loop parallelization is beneficial
- Inner most loop parallelization is the most beneficial
- Sscalar component of the code is the limiting factor

Previous Next

Module 18: Loop Optimizations

Lecture 35: Amdahl's Law

- A variable whose values form an arithmetic progression
- Variables are usually expressed as function of loop index
- Elimination reduces number of operations inside the loop
- Increases parallelization by reducing dependence cycles

```
for i =1,n
j = 2 * i + 1
A[i] = (A[i] + B[j])/2
endfor
```

```
for I = 1,n
A[i] = (A[i] + B[2*i+1])/2
endfor
```

- Induction variable form a series except first or the last term
- Parial loop unrolling may be required

```
j = n
for I = 1,n
A[i] = (B[i]+B[j]))/2
j = I
endfor
A[1] = (B[1]+B[n])/2
for I = 2,n
A[i] = (B[i] + B[i-1])/2
endfor
```

Index Recurrence

- Loop defined index variables are used to index array elements
- Their values do not form a progression

```
for I = 1,n
j = j +I
A[i] = A[i] + B[j]
endfor
```

```
for I = 1,n
A[i]=A[i]+B[i*(i+1)/2]
endfor
```

Previous Next

Loop Unrolling

- Change the loop stride (most common)
- Peel of one or more iterations at the beginning or at the end

```
for j = 1, n, k
for I = j, min(j+k, n)
A[i] = B[i] + C[i]
endfor
endfor
```

```
N1 = trunc(N/K)
N2 = N1 * K
N3 = N –N2
For j = 1, N2, K
for I = j, j+k
A[i] = B[i] + C[i]
endfor
Endfor
For I = N3+1, N
A[i] = B[i] + C[i]
endfor
```

Constant Propagation And Expression Evaluation

- Most common optimization

```
for I = 1,n          pi = 3.14
pi = 3.14            pd = 6.28
pd = 2*pi            for I = 1, n
D[i] = pd*R[i]       D[i] = 6.28*R[i]
endfor               endfor
```

- Replace constant in array index expressions with their values

Module 18: Loop Optimizations
Lecture 35: Amdahl's Law

Loop Vectorization

- Generate vector instructions out of the loop
- Check for all dependencies in the loop

```
for I = 1,n
A[i] = B[i] + C[i]
D[i] = B[i] * k
endfor
A[1..n] = B[1..n] + C[1..n]
D[1..n] = k * B[1..n]
```

Partial Loop Vectorization

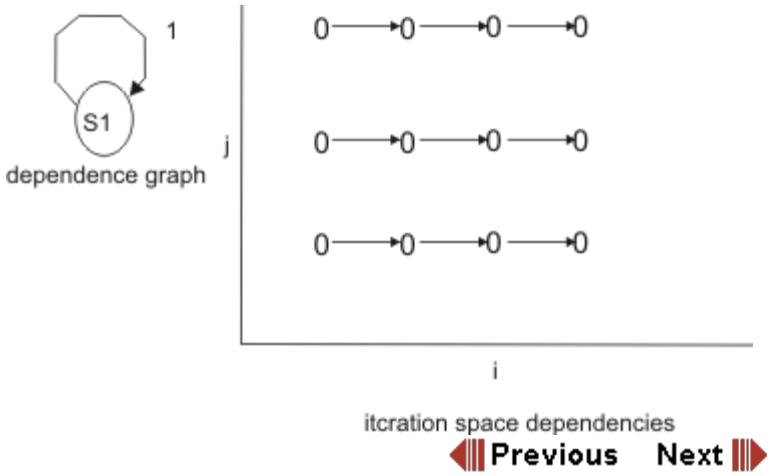
```
For I = 1,n
A[i+1] = B[I-1] + C[i]
B[i] = A[i] * k
C[i] = B[i] -1
endfor
```

```
for I = 1,n
A[i+1] = B[i-1] + C[i]
B[i] = A[i] * k
endfor
C[1..n] = B[1..n] –1
```

Nested Loops

- Useful to identify which loop 'carries the dependence'.
- Assume loops are numbered: outermost is 1, next one is 2 and so on
- Example

```
for I=1,N
for J=1,M
X[I,J]=X[I-1,J]
endfor
endfor
```



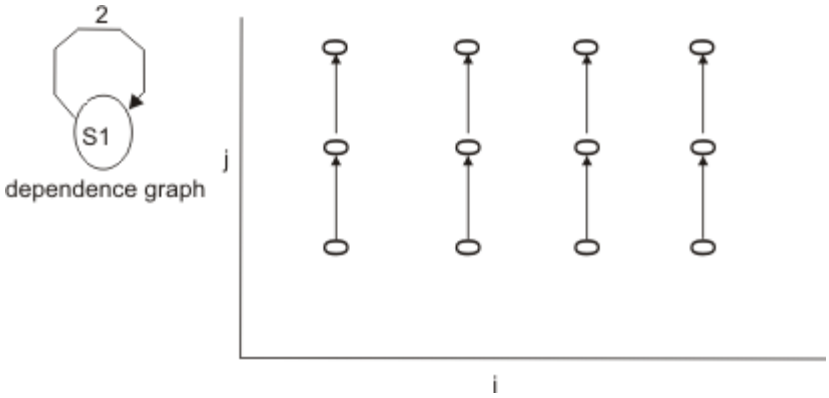
Previous Next

Module 18: Loop Optimizations

Lecture 35: Amdahl's Law

- Values written during one iteration of the outer loop is read during some iteration of outer loop
- Dependence disappears if outer loop is kept fixed and inner loop runs free
- Therefore, the dependence is carried by the outer loop and label the dependence with 1
- Parallel code:
  for I=1,N
  X[I,1..M]=X[I-1,1..M]
  endfor

- Example
  for I=1,100
  for J=1,100
  X[I,J]=X[I,J-1]
  endfor
  endfor



- dependence is carried by the inner loop
- Parallelization requires loop interchange

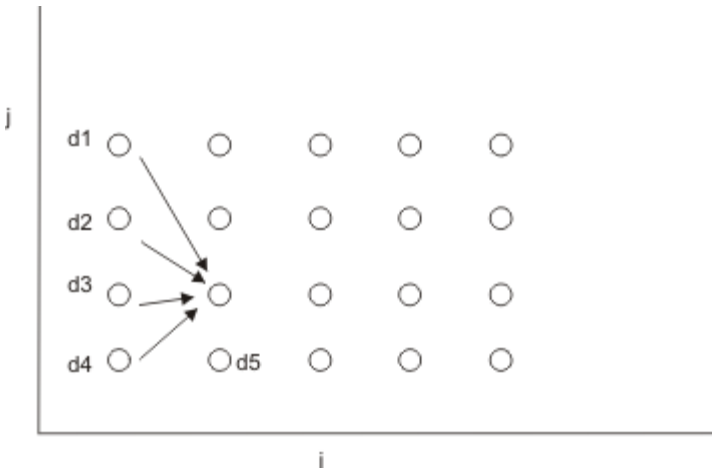
Loop Interchange

- The most important loop restructuring transformation
- It was developed for automatic parallelization of loops
- If a loop carried all the dependence then it would be brought to the outermost position
- Rest of the loops which did not carry dependence will be executed in parallel
- It can be done if there are no dependence cycles
- If outer loop iterates many times and inner loop only a few times then loop startup overhead is high
- Interchanging loops will improve performance
- Interchange can lead to spatial locality (remember matrix multiplication?)

Previous Next

Loop Interchange

```
for I=1,N
for J=1,M
.....
endfor
endfor
```



- d1, d2: Loop interchange is illegal
  for I=1,100
  for J=1,100
  X[I,J]=....X[I-1,J+1]…

- d3: Loop interchange is legal but may not be profitable
  for I=1,100
  for J=1,100
  X[I,J]=X[I-1,J]

- d4, d5: Interchanging loops can enhance parallel content
  for I=1,100
  for J=1,100
  X[I,J[=X[I,J-1]....X[I-1,J-1]

- Makes the inner most loop parallelizable
  for I = I,n
  for j = 1,m
  A[I,j] = A[I,j-1] + 1
  endfor
  endfor
  for j = 1,m
  A[1..n, j] = A[1..n, j-1] + 1
  endfor

Previous Next

Loop interchange

- Loop interchange is not always legal
- A forall loop can be changed with any loop nested inside it
- A serial loop cannot always be interchanged with a loop surrounding it
- Following loops can not be interchanged
  for I = 2,n
  for j = 1, m
  A[I,j] = A[i-1, j+1] + 1
  endfor
  endfor

Loop Limits in Loop Interchange

- When the loop limits of the inner loop are invariant in outer loop the loops can be changed without changing limits
- When the limits vary in the outer loop the limits can change
- Interchanging two loops is equivalent to transposing the iteration space
- Fourier Motzkin projections are used for finding new limits

For I = 1, 10
for j = I, 12
A[I,j] = A[I, j+1]
endfor
Endfor

- The lower limit of j varies in I
- The limits before interchanging are:
  -I = -1 I = 10 i-j = 0 j = 12
- The iteration space is shown in the figure on next foil

Previous Next

Module 18: Loop Optimizations
Lecture 35: Amdahl's Law

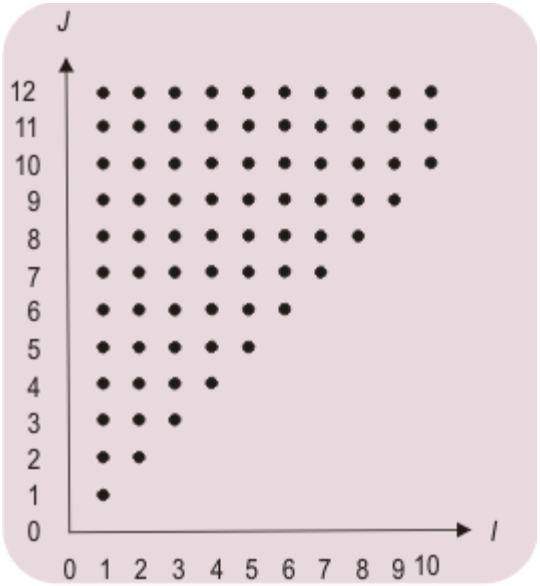


Figure Courtesy: High Performance Compilers For Parallel Computing by Wolfe

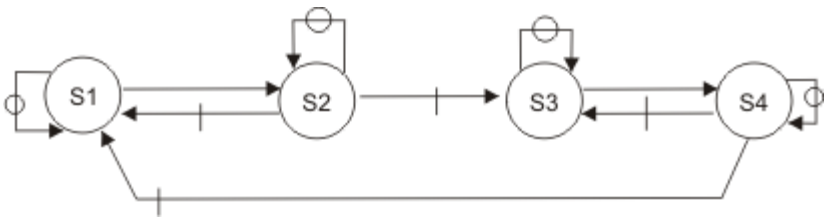
Transformed Loop
For j = 1, 12
for I = 1, min(10,j)
A[I,j] = A[I, j+1]
endfor
endfor

Removal of Pseudo Dependencies

- Anti dependencies and output dependencies arise from reuse of storage
- These dependencies can be eliminated by avoiding the reuse of storage
- Most common transformations
  - Renaming: Give the occurrences of a variable with disjoint lifetimes different names
  - Expansion: Make a scalar into an array
  - Node splitting: Make a copy of an array

Example

for I=1,100
A=..... S1
B=.....A... S2
A=..... S3
C=.....A... S4



Previous Next

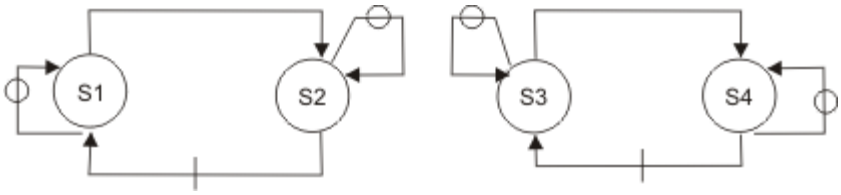
Life Time Range Splitting

- Rename second occurrence of A

for I=1,100

A=.....	S1
B=.....A....	S2
A'=.....	S3
C=.....A'....	S4

endfor



Expansion of All Scalar Variables

For I=1,100

A[I]=.....	S1
B[I]=....A[I]...	S2
A'[I]=.....	S3
C[I]=.....A'[I]....	S4

endfor

Previous Next

Node Splitting

- Loop parallelization is impossible when the statements are involved in a dependence cycle
- Sometimes dependence cycles can be broken resulting in total/partial parallelization of loop
- Flow dependence cycles are very hard to break
- Anti-and output-dependence cycles can be broken by renaming variables
- Following loop can not be parallelized

```
for I = 1,n
A[i] = B[i] + C[i]
D[i] = A[i-1] + A[i+1]
endfor
```

Node Splitting ...

```
for I = 1,n
A[i] = B[i] + C[i]
temp[i] = A[i+1]
D[i] = A[i-1] + temp[i]
endfor
```

```
for I = 1,n
temp[i] = A[i+1]
A[i] = B[i] + C[i]
D[i] = A[i-1] + temp[i]
endfor
```

```
temp[1..n] = A[2..n+1]
A[1..n] = B[1..n] + C[1..n]
D[1..n] = A[0..n-1] + temp[1..n]
```

Cycle Shrinking

- In many loops dependence cycles are impossible to break
- Such dependence cycles usually involve only few dependences
- Cycle shrinking is used to extract any parallelism that may be present in the loop

Previous Next