Background
- Vector processors
- Multi processors
- Vectorizing & parallelizing compilers
- SIMD & MIMD models

Loop Parallelism
- Bräunl 2004

Data Dependence Analysis
- Flow dependence
- Anti dependence
- Output dependence

Vectorization
- Exploiting vector architecture

Vectorization
- A(1:50) = B(1:50) + C(1:50)
- D(1:50) = A(1:50) / 2.0
Vectorization

□ Discovery
- build data dependence graph
- inspect dependence cycles
- inspect each loop statement to see if target machine has vector instruction to execute accordingly

□ Proper course of action?

□ Transformation
- loops with multiple statements must be transformed using the loop distribution
- loops with no loop-carried dependence or has forward flow dependences

DO I = 1, N
  5. A(I+1) = B(I) + C
  6. D(I) = A(I) + E
ENDDO

DO I = 1, N
  5. A(I+1) = B(I) + C
  6. D(I) = A(I) + E
ENDDO

DO I = 1, N
  5. A(I+1) = B(I) + C
  6. D(I) = A(I) + E
ENDDO

DO I = 1, N
  5. A(I+1) = B(I) + C
  6. D(I) = A(I) + E
ENDDO

Dependence Cycles

□ acyclic
  □ solution: re-ordering of statements

DO I = 1, N
  5. D(I) = A(I) + E
  6. A(I+1) = B(I) + C
ENDDO

DO I = 1, N
  5. D(I) = A(I) + E
  6. A(I+1) = B(I) + C
ENDDO

DO I = 1, N
  5. D(I) = A(I) + E
  6. A(I+1) = B(I) + C
ENDDO

DO I = 1, N
  5. D(I) = A(I) + E
  6. A(I+1) = B(I) + C
ENDDO

□ Cyclic
  □ solution: statement substitution
  □ otherwise, distribute loop
- dependence cycle statements in a serial loop
- rest of the loop as vectorized

DO I = 1, N
  5. A(I) = B(I) + C(I)
  6. B(I+1) = A(I) + 2
ENDDO

DO I = 1, N
  5. A(I) = B(I) + C(I)
  6. B(I+1) = A(I) + 2
ENDDO

DO I = 1, N
  5. A(I) = B(I) + C(I)
  6. B(I+1) = A(I) + 2
ENDDO

DO I = 1, N
  5. A(I) = B(I) + C(I)
  6. B(I+1) = A(I) + 2
ENDDO

Nested loops

□ Conditions in loop

DO I = 1, N
  5. IF(A(I) < 0)
  6. A(I) = -A(I)
ENDDO

WHERE(A(1:N) < 0) A(1:N) = -A(1:N)

Parallelization

□ Exploiting multi-processors
- Allocate individual loop iterations to different processors
- Additional synchronization is required depending on data dependences
Parallelization

- Fork/Join parallelism
- Scheduling
  - Static
  - Self-scheduled

Parallelization

The inner loop is to be parallelized:

```plaintext
for i:=1 to n do
  for j:=1 to m do
    S1: A[i,j] := C[i,j];
    S2: B[i,j] := A[i-1,j-1];
  end;
end;
```

- Data dependency: S1 \(\delta\) (\(<,<\)) S2 (due to \(A[i,j]\))
- Synchronization required: NO

Parallelization

After re-ordering and adding sync code

```plaintext
for i:=1 to n do
  S1: A[i] := B[i] + C[i];
  S2: D[i] := A[i] + E[i-1];
  S3: E[i] := E[i] + 2 * B[i];
  S4: F[i] := E[i] + 1;
end;
```

- Data Dependences:
  - S1 \(\delta\) S2 (due to \(A[i]\)) -- no synch. required
  - S3 \(\delta\) S4 (due to \(E[i]\)) -- no synch. required
  - S3 \(\delta\) S2 (due to \(E[i]\)) -- synch. required

Parallelization

```plaintext
for i:=1 to n do
  S1: A[i] := C[i];
  S2: B[i] := A[i];
end;
```

- Data dependency: S1 \(\delta\) (\(\approx\)) S2 (due to \(A[i]\))
- Synchronization required: NO

Parallelization

```plaintext
doacross i:=1 to n do
  S1: A[i] := C[i];
  S2: B[i] := A[i];
endoacross;
```

Review-I

- Data dependence within an instruction

```plaintext
for i:=1 to n do
  S1: A[i] := A[i+1]
end;
```

- Is this loop vectorizable?
Review-II

- Data dependence within an instruction

```
for j:= 1 to n do
  S1: X[j+1] := X[j] + C
end;
```

- Is this loop vectorizable?

References

- *Optimizing Supercompilers for Supercomputers*, Michael Wolfe
- *Parallelizing and Vectorizing Compilers*, Rudolf Eigenmann and Jay Hoeflinger
- *Optimizing Compilers for Modern Architectures*, Randy Allen, Ken Kennedy