Loop Parallelization Techniques

- Data-Dependence Analysis
- Dependence-Removing Techniques
- Parallelizing Transformations
- Performance-enhancing Techniques

Some motivating examples

Do i = 1, n
  a(i) = b(i)
  c(i) = a(i-1)
End do

Is it legal to
- Run the i loop in parallel?
- Put S2 first in the loop?

Do i = 1, n
  a(i) = b(i)
End do
Do i = 1, n
  c(i) = a(i-1)
End do

Is it legal to
- Fuse the two i loops?

In general, it is desirable to determine if two references access the same memory location, and the order they execute, so that we can determine if the references might execute in a different order after some transformation.

Dependence, an example

Do i = 1, n
  a(i) = b(i)
  c(i) = a(i-1)
End do

Indicates dependences, i.e. the statement at the head of the arc is somehow dependent on the statement at the tail.

Can this loop be run in parallel?

Assume 1 iteration per processor, then if for some reason some iterations execute out of lock-step, bad things can happen.

In this case, read of a(2) in i=3 will get an invalid value!

Can we change the order of the statements?

Do i = 1, n
  a(i) = b(i)
  c(i) = a(i-1)
End do

No problem with a serial execution.

Types of dependence

- Flow or true dependence – data for a read comes from a previous write (write/read hazard in hardware terms)
- Anti-dependence – write to a location cannot occur before a previous read is finished
- Output dependence – write a location must wait for a previous write to finish

Dependences always go from earlier in a program execution to later in the execution. Anti and output dependences can be eliminated by using more storage.
Eliminating anti-dependence

Anti-dependence – write to a location cannot occur before a previous read is finished.

Let the program in be:

\[ a(2) = \ldots \]
\[ \ldots = a(2) \]
\[ a(2) = \ldots \]
\[ = \ldots a(2) \]

Create additional storage to eliminate the anti-dependence.

The new program is:

\[ a(2) = \ldots \]
\[ \ldots = a(2) \]
\[ a(2) = \ldots \]
\[ = \ldots a(2) \]

No more anti-dependence!

Getting rid of output dependences

Output dependence – write a location must wait for a previous write to finish.

Let the program in be:

\[ a(2) = \ldots \]
\[ \ldots = a(2) \]
\[ a(2) = \ldots \]
\[ = \ldots a(2) \]

Again, by creating new storage we can eliminate the output dependence.

The new program is:

\[ a(2) = \ldots \]
\[ \ldots = a(2) \]
\[ a(2) = \ldots \]
\[ = \ldots a(2) \]

Eliminating dependences

• In theory, can always get rid of anti- and output dependences.
• Only flow dependences are inherent, i.e. must exist, thus the name “true” dependence.
• In practice, it can be complicated to figure out how to create the new storage.
• Storage is not free – cost of creating new variables may be greater than the benefit of eliminating the dependence.

Can we fuse the loop?

Do i = 1, n
\[ a(i) = b(i) \]
End do

DON'T Do i = 1, n
\[ \text{Before i-1th iteration} \]
\[ c(i) = a(i-1) \]
End do

1. Is ok after fusing, because get a(i-1) from the value assigned in the previous iteration.
2. No “output” dependence on a(i) or c(i), not overwritten.
3. No input flow, or true dependence on a b(i), so value comes from outside the loop.

Data Dependence Tests:

Other Motivating Examples

Statement Reordering

Can these two statements be swapped?

DO i=1,100,2
\[ B(2*i) = \ldots \]
\[ = \ldots B(3*i) \]
ENDDO

Loop Parallelization

Can the iterations of this loop be run concurrently?

DO i=1,100,2
\[ B(2*i) = \ldots \]
\[ = \ldots B(2*i)+B(3*i) \]
ENDDO

An array data dependence exists between two data references iff:
• Both references access the same storage location
• At least one of them is a write access.
Dependence sources and sinks

- The sink of a dependence is the statement at the head of the dependence arrow.
- The source is the statement at the tail of the dependence arrow.

```latex
\begin{verbatim}
for (i=1; i < n; i++) {
    a[i] = ...
    ...
    a[i] = a[i-1]
}
```

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.
- Iteration space graphs: the un-abstracted form of a dependence graph with one node per statement instance.

Data Dependence Tests: Iteration space graphs

- Iteration space graphs: the un-abstracted form of a dependence graph with one node per statement instance.

```latex
\begin{verbatim}
do i = 1, n
    a(i, i) = a(i-2, i+3)
end do
\end{verbatim}
```

Data Dependence Tests: Distance Vectors

Distance (vector): indicates how many iterations apart are the 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.

Data Dependence Tests: Direction Vectors

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.

Data Dependence Tests: Loop Carried

- 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.
  - Dependence on the i_{2} loop is loop carried.
  - Dependence on the i_{1} loop is not.
Data Dependence Tests: Loop Carried

• 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.

```
do i = 1, n
dopar i = 1, n
a(i, i+1) = a(i, i+1)
end dopar
end do
```

This is legal since loop splitting enforces the loop carried dependences.

```
i1 = 1 2 3 4 5
i'2 = 2
```

Data Dependence Tests: Loop Carried

```
do i = 1, n
dopar i = 1, n
a(i, i+1) = a(i, i+1)
end dopar
end do
```

This is not legal – turns true into anti-dependence.

```
i1 = 1 2 3 4 5
i'2 = 4
```

A quick aside

A loop
```
do i = 4, n, 3
a(i)
edo
```
Can be always be normalized to the loop
```
do i = 0, (n-1)/3-1, 1
a(3*i+4)
edo
```

This makes discussing the data-dependence problem easier since we only worry about loops from 1, n, 1

More precisely, do
```
do i = lower, upper, stride { a(i)} becomes
```
```
do i' = 0, (upper – lower + increase)/stride – 1, 1 { a(i'*stride + lower)}
```

Diophantine equations

• An equation whose coefficients and solutions are all integers is a Diophantine equation.

• Determining if a Diophantine equation has a solution requires a slight detour into elementary number theory.

• Let \( f(i) = a_1 i_1 + c_1 \) and \( g(i) = b_1 i_1 + c_2 \), then

\[ f(i) = g(i) \Rightarrow a_1 i_1 + b_1 i_1 = c_2 - c_1 \]

fits general form of Diophantine equation of

\[ a_1 i_1 = a_1 i_1 + c \]

Euclid Algorithm: find gcd(a,b)
Repeat
\[ a = a \mod b \]
swap a,b
Until \( b=0 \) →The resulting a is the gcd

Examples:
- \( 15i + 6j - 9k = 12 \) has a solution \( \text{gcd}=3 \)
- \( 2i + 7j = 3 \) has a solution \( \text{gcd}=1 \)
- \( 9i + 3j = 6k = 5 \) has no solution \( \text{gcd}=3 \)

Copyright 2002 Prof. R. Eigenmann, Prof. S. Midkiff
**Finding GCDs**

**Euclid Algorithm: find gcd(a,b):**
- Repeat
  - a = a mod b
  - b = a mod b
- Until b=0
- The resulting a is the gcd

For more than two numbers:
- gcd(a,b,c) = gcd(gcd(a,b),c)

Example: a = 16, b = 6
- a = 16 mod 6 = 4
- b = 4 mod 4 = 0

→ The result is 4.

---

**Determining if a Diophantine equation has a solution**

Let \( g = \text{gcd}(a_1, a_2) \), rewrite the equation as:

\[
g a_1' + \frac{g}{a_2} a_2' = c \Longrightarrow \text{gcd}(a_1', a_2') = c
\]

Because \( a_1' \) and \( a_2' \) are relatively prime, all integers can be expressed as a linear combination of \( a_1' \) and \( a_2' \). Hence, \( a_1' \cdot a_2' = c \) is just such a linear combination and therefore \( a_1' = a_1, a_2' = a_2 \).

Example:
- If \( \text{remainder}(c, g) = 0 \), \( c \) can be expressed as a linear combination and therefore \( a_1' = a_1, a_2' = a_2 \).
- Let \( g = \text{gcd}(1, i) \), \( a_1' = 1, a_2' = i \) generates all integers \( i \), (assuming \( a_1' = a_1, a_2' = a_2 \) can range over the integers)

Other tests and methods are developed by Banerjee, Allen, and others.

---

**More information on gcd’s and dependence analysis**

- General books on number theory
  - Randy Allen’s thesis, Rice University
  - Work by Eigenmann & Blume (Purdue) range test
  - Work by Pugh (Omega test) Maryland
  - Work by Hoeflinger, etc. Illinois (LMAD)

---

**What do dependence tests do?**

- Some tests, and Banerjee’s in some situations (affine subscripts, rectangular loops) are precise
  - Definitively proves existence or lack of a dependence
- Most of the time tests are conservative
  - Always indicate a dependence if one may exist
  - May indicate a dependence if it does not exist
- In the case of “may” dependence, run-time test or speculation can prove or disprove the existence of a dependence
- Short answer: tests disprove dependences for some dependences

---

**Banerjee’s Inequalities**

If \( a_1', b_1', c \), does \( a_1 \), has a solution within the loop bounds, and for a given direction vector?

By the mean value theorem, \( c \) can be a solution to the equation \( f(x) = c \) if

\[
\frac{f(ub) - f(lb)}{ub - lb} = f'(c)
\]

\( \text{Assumes f(ub) - f(lb) > 0} \)

The idea behind Banerjee’s Inequalities is to find the maximum and minimum values the dependence equation can take on for a given direction vector, and see if these bounds c. This is done in the real domain since integer solution requires integer programming (in \( NP \)).
Example of where the direction vector makes a difference

\[
\begin{align*}
d & = 1, 100 \\
x(i) & = x(i-1)
\end{align*}
\]

Note: there is a (<) dependence. Let’s test for (=) and (<) dependence.

Dependence equation is \(k' = i - i\). If \(i = i\), then \(k' = 0\), \(\forall i\). If \(i < i\), then \(k' \neq 0\), and when \(P = i + 1\), the equation has a solution.

Banerjee test

If \(a_i - b_i = c\) has a solution, does it have a solution within the loop bounds for a given direction vector (>) or (=) in this case?

For our problem, does \(i - i' = -1\) have a solution

- For \(i = i'\), then it does not (no (=) dependence).
- For \(i < i'\), then it does ((<) dependence).

Program Transformations

- Applying data dependence tests to untransformed loops would determine that most loops are not parallel.

Reason #1: there are many anti and output dependences

\[
\begin{align*}
DO & = 1, n \\
t & = a(i) + b(i) \\
c(i) & = t
\end{align*}
\]

Solution: scalar and array privatization

Scalar Expansion/Privatization

DO = 1, n
Private t
\[
\begin{align*}
t & = a(i) + b(i) \\
c(i) & = t
\end{align*}
\]

PARALLEL

Private creates one copy per parallel loop iteration.

Analysis and Transformation for Scalar Expansion/Privatization

Loop Analysis:
- Find variables that are used in the loop body but dead on entry i.e., the variables are written on all paths through the loop before being used.
- Determine if the variables are live out of the loop (make sure the variable is defined in the last loop iteration).

Transformation (variable t)
- Privatization:
  - put t on private list. Mark as last value if necessary.
- Expansion:
  - declare an array (t(i)), with \(i = \text{loop iterations}.
  - replace all occurrences of t in the loop body with (t(i)), where i is the loop variable.
  - live-out variables: create the assignment \(t = t(n)\) after the loop.

Parallelization of Reduction Operations

DO = 1, n
\[
\begin{align*}
s & = 0 \\
\text{sum} & = \text{sum} + a(i)
\end{align*}
\]

PARALLEL

DIMENSION s(#processors)

DO = 1, n/#processors
Private s
\[
\begin{align*}
s & = 0 \\
\text{sum} & = \text{sum} + s(j)
\end{align*}
\]

Copyright 2002 Prof. R. Eigenmann, Prof. S. Midkiff
Analysis and Transformation for (Sum) Reduction Operations

- Loop Analysis:
  - find reduction statements of the form $s = s + \text{expr}$ where \text{expr} does not use $s$
  - discard $s$ as a reduction variable if it is used in non-reduction statements.

- Transformation:
  - create private or expanded variable and replace all occurrences of reduction variable in loop body.
  - update original variable with sum over all partial sums, using a critical section in a loop postamble or a summation after the loop, respectively.

Induction Variable Substitution

\[
\text{ind} = \text{ind0} \\
\text{DO } j = 1, n \\
a(\text{ind0} + k(j-1)) = b(j) \\
\text{ENDDO}
\]

This is of the form $a*j + c$, which is good for dependence analysis. $j$ is the loop canonical induction variable.

Induction Variable Analysis and Transformation

- Loop Analyis:
  - find induction statements of the form $s = s + \text{expr}$ where \text{expr} is a loop-invariant term or another induction variable.
  - discard variables that are modified in non-induction statements.

- Transformation:
  - find the following increments:
    - for each use of IV:
      - compute the increment inc with respect to the start of the outermost loop in which it is an induction sequence in
      - Replace IV by inc+\text{ind0}.

Loop Fusion and Distribution

- necessary form for vectorization
- can provide synchronization necessary for "looped" dependence
- can create perfectly nested loops

Both transformations change the statement execution order. Data dependencies need to be considered!
Loop Fusion and Distribution

\begin{align*}
\text{DO } & j = 1, n \\
\text{S1: } & a(j-1) = b(j) \\
\text{ENDDO} \\
\text{DO } & k = 1, n \\
\text{S2: } & c(k) = a(k) \\
\text{ENDDO}
\end{align*}

Fusion

Distribution

Flow dependence from S1 to S2

Anti-dependence from S2 to S2

Loop Interchange

\begin{align*}
\text{PDO } & i = 1, n \\
\text{PDO } & j = 1, m \\
\text{DO } & j = 1, n \\
\text{DO } & i = 1, n \\
\text{a(i,j)} & = b(i,j) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}

• loop interchanging alters the data reference order
  • significantly affects locality of reference
  • data dependences determine the legality of the transformation:
    - dependence structure should stay the same
  • loop interchanging may also impact the granularity of the parallel computation (inner loop may become parallel instead of outer)

General Iteration Space Partitioning

Given an iteration dependence graph, the graph is partitioned such that:

• no data dependences cross partitions
• locality is maximized

Dependence graphs

\begin{align*}
\text{DO } & j = 1, n \\
\text{S1: } & a(j) = b(j) \\
\text{ENDDO} \\
\text{DO } & k = 1, n \\
\text{S2: } & c(k) = a(k) \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } & j = 1, n \\
\text{S1: } & a(j) = b(j) \\
\text{ENDDO} \\
\text{DO } & k = 1, n \\
\text{S2: } & c(k) = a(k) \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } & j = 1, n \\
\text{S1: } & a(j) = b(j) \\
\text{ENDDO} \\
\text{DO } & k = 1, n \\
\text{S2: } & c(k) = a(k) \\
\text{ENDDO}
\end{align*}

\begin{align*}
\text{DO } & j = 1, n \\
\text{S1: } & a(j) = b(j) \\
\text{ENDDO} \\
\text{DO } & k = 1, n \\
\text{S2: } & c(k) = a(k) \\
\text{ENDDO}
\end{align*}

Loop Blocking

(a.k.a. Stripmining)

\begin{align*}
\text{PDO } & j = 1, n \\
\text{PDO } & j = 1, m \\
\text{DO } & j = 1, n, \text{block} \\
\text{DO } & k = 0, \text{block} - 1 \\
\text{a(j+k)} & = b(j+k) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}

Many variants:

• adjustment if \( n \) is not a multiple of \( \text{block} \)
• number of blocks = number of processors or vector register length
• cyclic or block-cyclic split

Criteria for Parallelization

• Vectorization:
  – no “backward dependence”.
  – If we allow statement reordering: no dependence cycles
• Parallelization:
  – no loop-carried dependence

Note, loops inside a dependence-carrying loop are dependence free (w.r.t. a given reference pair)
Parallel Execution Scheme

- Most widely used: Microtasking scheme

Main task

Helper tasks

- Main task creates helpers
- Wake up helpers
- Barrier, helpers go back to sleep
- Wake up helpers
- Barrier, helpers go back to sleep

Program Translation for Microtasking Scheme

Subroutine x

C$OMP PARALLEL DO
DO j=1,n
a(j)=b(j)
ENDDO

END

Subroutine x

call scheduler(1,n,a,b,loopsub)

END

Subroutine loopsub(lb,ub,a,b)

integer lb,ub
DO
jj = lb,ub
a(jj) = b(jj)
ENDDO
END