Code Generation for Parallel Computers

Loop Interchange and Parallelization

Single Loop Methods

Perfect Loop Nests

by Ken Kennedy

presented by Cheng Leong

March 21, 1995

Loop Interchange and Parallelization

vectorization prefers parallel *inner* loops

parallelization prefers parallel *outer* loops

A

```
DO I = 1, N                        PARALLEL DO J = 1, M
  DO J = 1, M                        DO I = 1, N
    A(I+1,J) = A(I,J) + C              A(I+1,J) = A(I,J) + C
  ENDDO                              ENDDO
ENDDO                              END PARALLELDO
```

B

```
DO I = 1, N                        PARALLEL DO J = 1, M
  DO J = 1, M                        DO I = 1, N
    A(I+1,J) = A(I,J) + C              A(I+1,J) = A(I,J) + C
  ENDDO                              ENDDO
ENDDO                              END PARALLELDO
```

When can we move a loop to the outermost position?

Theorem 6.1

In a nest of loops, a particular loop can be parallelized at the outermost level if and only if the column of the direction matrix for that level contains only "=" entries

Proof

An all "=" column can be moved out because no first "<" columns will be preceded by anything other than "="

A column containing ">" would be an illegal interchange

A column containing "<" contains a dependence at that level and therefore cannot be parallelized outermost

Q.E.D.

Code Generation Strategy #1

1. If there is an all “=” column, move it outermost, parallelize it, and eliminate the column from the direction matrix. Repeat strategy.

2. Otherwise, pick the column with the most “<” entries and generate a sequential loop for it in the outermost position. Eliminate the column and any dependent rows (those containing “<”) from the direction matrix. Repeat strategy.

Example for Strategy #1

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X
      B(I,J,K+1) = B(I,J,K) + Y
      C(I+1,J+1,K+1) = C(I,J,K) + Z
    ENDDO
  ENDDO
ENDDO
```

```
DO I = 1, N
  PARALLEL DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X
      B(I,J,K+1) = B(I,J,K) + Y
      C(I+1,J+1,K+1) = C(I,J,K) + Z
    ENDDO
  END PARALLEL DO
ENDDO
```

Single-Loop Methods

Situation:

We have a single loop and want to make it parallel.

Problem:

What transformations produce a profitable parallelization?

Answers:

- Tiling
- Privatization
- Alignment
- Code Replication

Tiling

Situation:

Parallel loop with no dependencies, <u>but</u> not much to do

Solution:

Group the iterations into sets and let a set be the minimum amount of work that one processor can do. *(strip mining)*

```
DO I = 1, N

  A(I) = A(I) + B(I)

ENDDO
```

```
k = CEIL(N/P)
  PARALLEL DO I = 1, N, k
    DO i = I, MIN(I+k-1,N)
      A(i) = A(i) + B(i)
    ENDDO
END PARALLEL DO
```

Privatization and Alignment

Privatization

"Scalar and array privatization based on kill analysis"

Situation:

loop contains a carried dependece

Solutions:

1. use loop distribution
2. change the iteration on which the source is calculated

Alignment example

```
DO I = 1, N
  A(I+1) = B(I) + C
  B(I) = A(I) + D
ENDDO
```

distribution

```
DO I = 1, N
  A(I+1) = B(I) + C
ENDDO
DO I = 1, N
  B(I) = A(I) + D
ENDDO
```

naïve alignment

```
DO i = 1, N+1
  IF (i > 1) A(i) = B(i-1) + C
  IF (i ≤ N) B(i) = A(i) + D
ENDDO
```

folded alignment

```
DO i = 1, N
  j = MOD(i-2,N) + 1
  A(j+1) = B(j) + C
  B(j) = A(j) + D
ENDDO
```

When can we use alignment?

For all carried dependences?

No! If there is a recurrence, then we cannot simply align it away.

```
DO I = 1, N
  A(I) = B(I) + C
  B(I+1) = A(I) + D
ENDDO
```

Lemma 6.1.

If loop statements form a recurrence then alignment and reordering alone cannot convert all carried dependences to loop-independent while preserving the meaning of the original loop.

When can we use alignment?

For all carried dependences not in recurrences?

No! There can be an *alignment conflict*—two or more dependences that cannot be simultaneously aligned

```
DO I = 1, N
  A(I+1) = B(I) + C
  X(I) = A(I+1) + A(I)
ENDDO
```

There are two dependences on `A`—one loop-independent and one loop-carried. Aligning the carried dependence will convert the independent dependence to loop-carried.

Code Replication

Situation:

There is an *alignment conflict* in the loop

Solution:

Replace the *use* of the conflicting expression with a *computation* of the expression.

```
DO I = 1, N
  A(I+1) = B(I) + C
  X(I) = A(I+1) + A(I)
ENDDO
```

```
DO I = 1, N
  IF (I == 1) THEN t = A(I)
  ELSE t = B(I-1) + C
  ENDIF
  A(I+1) = B(I) + C
  X(I) = A(I+1) + t
ENDDO
```

`t` is private so there are no carried dependences

Single Loop Algorithm

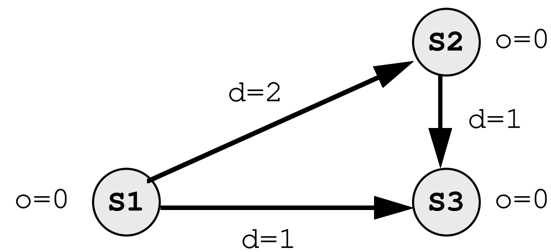
Is alignment + replication + reordering enough to eliminate non-recurrence carried dependences?

YES!

Theorem 6.2
In a single loop that contains no recurrence and in which the distance of each dependence is a constant independent of the loop index, alignment, replication and statement reordering are sufficient to eliminate all carried dependences.

Single Loop Example

```
   DO I = 1, N
S1  A(I+2) = B(I) + C
S2  X(I+1) = A(I) + D
S3  Y(I) = A(I+1) + X(I)
   ENDDO
```



o(v) is the offset of vertex v

d(e) is the distance for dependence edge e

G=(V,E) is *carry-free* if for each edge e=(v1,v2)
o(v1) + d(e) = o(v2)

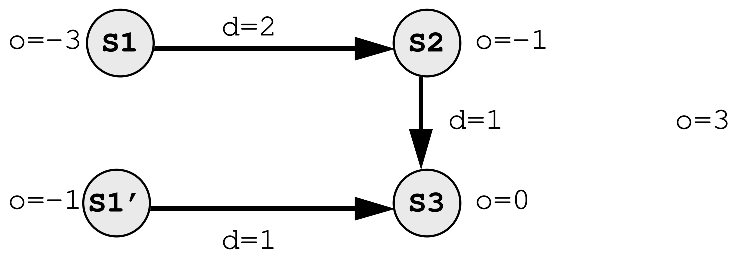
The goal of the algorithm is to generate a *carry-free* graph to generate code from.

The algorithm picks a node and then aligns all the other nodes with it, using code replication to resolve alignment conflicts.

Alignment and Replication Algorithm

```
/*  V is vertex set.          E is edge set.          d is dependence distance table.
    o is the offset table.   W is a work list for the algortihm. */
forall v∈V do o(v) = undefined od;
while V ≠ Ø do
    select and remove an arbitrary element v0 from V;  W = {v0};  o(v0) = 0;
  while W ≠ Ø do
    select and remove an arbitrary element v from W;
    foreach edge e incident on v do
      if w∈V  then
        W = W ∪ {w};  o(w) = o(v) - d(e);
      else if o(w) ≠ o(v) - d(e) then /* resolve conflict by replicating the upstream vertex */
        /* let e=(up, down) */
        create a new vertex up';
        replace edge e=(up,down) with e'=(up',down);
        foreach edge into up do
          replicate the edge, replacing up with up';
        od;
        W = W ∪ {up'}; o(up') = o(down) - d(e);
      endif
    od
  od
od
```

Single Loop Example




```
  DO I = 1, N
S1  IF (I ≥ 4) THEN A(I−1) = B(I−3) + C
S1’ IF ((I ≥ 2) && (I ≤ N+1)) THEN
            t = B(I−1) + C
    ELSE
            t = A(I+1)
    ENDIF
S2  IF ((I ≥ 2) && (I ≤ N+1)) THEN X(I) = A(I−1) + D
S3  IF (I ≤ N) THEN Y(I) = t + X(I)
  ENDDO
```

Single Loop Example

```
  DO I = 1, N
    i1 = MOD(I-4,N) + 1
S1  A(i1+2) = B(i1) + C
    i2 = MOD(I-2,N) + 1
S1' IF (i2 ≥ 2) THEN
      t = A(i2) + D
    ELSE
      t = X(I)
    ENDIF
S2  X(i2+1) = A(i2) + D
S3  Y(I) = t + X(I)
  ENDDO
```

Perfect Loop Nests

Situation:

We have a perfect loop nest.

Problem:

We want to maximize the parallelism while retaining adequate granularity.

Solutions:

- Loop Interchange
- Loop Reversal
- Loop Skewing
- Fixed Distance Vector Transformations

Loop Interchange

choose an outer loop that leaves max parallelism

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J+1,K+1) = A(I,J,K) + A(I+1,J+2,K)
    ENDDO
  ENDDO
ENDDO
```

$$\begin{bmatrix} = & < & < \\ < & > & < \end{bmatrix} \quad \begin{bmatrix} < & < & < \\ = & > & < \end{bmatrix}$$

```
DO K = 1, L
  PARALLEL DO J = 1, M
    PARALLEL DO I = 1, N
      A(I+1,J+1,K+1) = A(I,J,K) + A(I+1,J+2,K)
    END PARALLEL DO
  END PARALLEL DO
ENDDO
```

$$\begin{bmatrix} < & < & = \\ > & < & < \end{bmatrix} \quad \begin{bmatrix} < & < & < \\ > & < & = \end{bmatrix}$$

Loop Reversal

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I+1,J+1,K+1) = A(I,J-1,K+1) + A(I-1,J,K+1)
    ENDDO
  ENDDO
ENDDO
```

$$\begin{bmatrix} = & < & > \\ < & = & > \end{bmatrix}$$

```
DO K = L, 1, -1
  PARALLEL DO J = 2, M+1
    PARALLEL DO I = 2, N+1
      A(I+1,J+1,K+1) = A(I,J-1,K+1) + A(I-1,J,K+1)
    END PARALLEL DO
  END PARALLEL DO
ENDDO
```

$$\begin{bmatrix} < & = & < \\ < & < & = \end{bmatrix}$$

Parallel Loop Selection Algorithm

```
procedure parallelize (N)
    compute direction matrix M;
    let L be the list of loops in the N;
    while L ≠ Ø do begin
      while there exist columns in M with all "=" directions do begin
        l = loop corresponding to outermost all "=" column;
        generate a parallel loop for l at the outermost position;
        remove l from L; remove the column for l from M;
       end;

      if L ≠ Ø then begin
        l = heuristically selected loop;
        generate a sequential loop for l at the outermost position;
        remove l from L; remove the column for l from M;
        remove all rows corresponding to dependences carried by loop l from M;
      endif
    end
end
```

use heuristic to select loop—do not solve an NP-complete problem

Code Generation with Loop Skewing

use loop skewing to change “=” directions to “<”

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I,J,K) = A(I,J−1,K) + A(I−1,J,K)
      B(I,J,K+1) = B(I,J,K) + A(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

$$\begin{bmatrix} = & < & = \\ < & = & = \\ = & = & \leq \end{bmatrix}$$

substitute `k = K + I + J`

```
DO I = 2, N+1
  DO J = 2, M+1
    DO k = 1, I+J+1, I+J+L
      A(I,J,k−I−J) = A(I,J−1,k−I−J) + A(I−1,J,k−I−J)
      B(I,J,k−I−J+1) = B(I,J,k−I−J) + A(I,J,k−I−J)
    ENDDO
  ENDDO
ENDDO
```

$$\begin{bmatrix} = & < & < \\ < & = & < \\ = & = & \leq \end{bmatrix}$$

Code Generation with Loop Skewing

move the inner loop out

```
DO k = 1, 5, N+M+1
  PARALLEL DO I = MAX(2,k-M-L-1), MIN(N+1,k-L-2)
    PARALLEL DO J = MAX(2,k-I-L), MIN(M+1,k-I-1)
      A(I,J,k-I-J) = A(I,J-1,k-I-J) + A(I-1,J,k-I-J)
      B(I,J,k-I-J+1) = B(I,J,k-I-J) + A(I,J,k-I-J)
    END PARALLEL DO
  END PARALLEL DO
ENDDO
```

$$\begin{bmatrix} < & = & < \\ < & < & = \\ \le & = & = \end{bmatrix}$$

Modification to Parallel Loop Selection Algorithm

if $L \neq \emptyset$ **then begin**

let FP be a collection of loops that can be moved to the outermost remaining loop positions and which are **fully permutable**;

l = heuristically selected loop from FP;

skew l with respect to the loops in FP that it does not cover;

A loop nest is *fully permutable* iff any permutation of the loops leaves all of the dependence distance vectors lexicographically positive.

Theorem 6.3

Loops l_i through l_j of a nest consisting of loops $l_1, l_2, \ldots, l_n$ are fully permutable iff each dependence in the nest is either carried at a level lower than i or D_i through D_j are all “<” or “=”

Theorem 6.4

Any loop nest for which each dependence has a distance vector of constants known at compile time can be made fully permutable by skewing

Nests with Fixed Distance Vectors

Situation:

all dependences can be represented by distance vectors of constants known at compile time

Solution:

use *unimodular* transformations of the loop indices

A transformation T is *unimodular* if

1. T is square,
2. all elements of T are integral, and
3. the absolute value of the determinant of T is 1.

Note: if T_1 and T_2 are unimodular, then so is $T_1 \bullet T_2$

Unimodular Transformations

Permutation:

identity
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j \\ k \end{bmatrix}$$

loop interchange
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} j \\ i \\ k \end{bmatrix}$$

Reversal:
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ -j \\ k \end{bmatrix}$$

Skewing by factor f:
$$\begin{bmatrix} 1 & 0 & 0 \\ f & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j+fi \\ k \end{bmatrix}$$

Theorem 6.5

Unimodular transformation T is legal iff for each distance vector *d* in the loop nest T*d*>0

Skewing Modification to Parallel Loop Selection Algorithm

```
procedure parallelize (N) /* with an emphasis on skewing */
    compute direction matrix M;
    let L be the list of loops in the N;
    while there exist columns in M with all “=” directions do begin
      l = loop corresponding to outermost all “=” column;
      generate a parallel loop for l at the outermost position;
      remove l from L; remove the column for l from M;
    end;

    if L ≠ Ø then begin
      skew the remaining loops to be fully permutable if necessary;
      l = heuristically selected loop from remaining loops;
      skew l with respect to the other loops remaining in L that it does not cover;
      generate a sequential loop for l at the outermost position;
      remove l from L;
      generate parallel loops for the loops remaining in L;
    endif
end
```