Elementary loop transformations

A number of transformations can be applied to loops for performance improvement.

Unrolling and interchanging have already been mentioned, but there are many others. We will discuss:

- Distribution
- Interchanging
- Fusion
- Stripmining
- Skewing
- Tiling
**Loop distribution**

We can decompose loops into several loops, one for each strongly-connected component (π-block) in the dependence graph.

```plaintext
DO I=1,N
S1:     A(I)-B(I)+C(I)
S2:     D(I)=D(I-1)+A(I)
S3:     IF(X.GT.A(I) THEN
S4         X=A(I)
         ENDF
         ENDIF
END DO

\[\downarrow\]

DO I=1,N
A(I)=B(I)+C(I)
END DO
DO I=1,N
D(I)=D(I-1)+A(I)
END DO
DO I=1,N
IF (X.GT.A(I) THEN
    X=A(I)
END IF
END DO
```
Loop interchanging

- The dependence information determines whether or not the loop headers can be interchanged.
- For example, the following loop headers can be interchanged

\[
\begin{align*}
do\ i=1,n \\
&\quad do\ j=1,n \\
&\quad \quad a(i,j) = a(i,j-1) + a(i-1,j) \\
&\quad end\ do \\
end\ do
\end{align*}
\]

- However, the headers in the following loop cannot be interchanged
do i=1,n
do j=1,n
    a(i,j) = a(i,j-1) + a(i-1,j+1)
end do
end do

The general rule is that it is illegal to interchange two adjacent headers if there is a dependence with the direction vector containing < > in the position corresponding to the two loops.
Loop fusion

\[
\begin{align*}
    &\text{do } i = 1 \text{ to } n \\
    &\quad \text{body-1} \\
    &\text{do } j=1 \text{ to } n \\
    &\quad \text{body-2} \\
    &\text{do } i=1,n \\
    &\quad \text{body-1} \\
    &\quad \text{body-2} \text{ (with all occurences of } j \text{ replaced by } i)
\end{align*}
\]

This transformation increases the loop size (granularity). In practice loop fusion is feasible where the iteration spaces are the same. It is legal to fuse two loops if no new dependence are introduced by the fusion.
Stripmining is useful for exploiting multiple levels of parallelism, cache locality, achieving fixed-size temporary arrays and reducing synchronization/communications costs (by doing send/receive every $m$ iterations instead of every iteration.)
**Loop skewing**

```plaintext
do i=1 to n
    do j=1 to m
        a(i,j) = a(i-1,j-1) ...
    do k=0, n+m-1
        do p = max(0,k-n), min(m,k)
            a(k-p,p) = a(k-p-1,p-1) ...
```

Diagram showing the skewing pattern in a 2D array.
**Loop reversal**

\[
\begin{align*}
\text{do } & i=1 \text{ to } n \\
\text{do } & j=1 \text{ to } m \\
\quad & a(i,j) = a(i-1,j+1) \ldots
\end{align*}
\]

After reversal and interchanging, the loop can be parallelized:

\[
\begin{align*}
\text{do } & k=0,1 \\
\text{do } & m=0,n \\
\quad & a(m,1-k)=a(m-1,1-k+1)
\end{align*}
\]
8. Program parallelization

One of the most important applications of source-to-source transformations is vectorization and parallelization.

In this tutorial we focus on loop parallelization/vectorization. Straight-line code parallelization is most often applied by the back-end compiler to exploit instruction-level parallelism although it can in some cases also be applied profitably for coarse-grain parallelism.
Multitasking

A task (or sequential process) is the execution of a program by a processor (a single control unit).

A multitasked program includes two or more tasks that could execute simultaneously.

There is a processor (real or virtual) associated with each task.

Virtual processors are created by multiplexing physical processors.

Each task executes at a speed greater than zero (unless it is blocked).
**Microtasking**

Work is based on *helper* or *implicit* tasks which usually execute several sequential code segments (called here *microtasks*).

This technique avoids some of the overhead of task creation since the task is created once and reused several times.

The implicit tasks fetch *microtasks* from a queue.

In most current implementations, once an implicit task gets a microtask $m$, it remains associated to $m$ until this microtask completes execution (*Non-preemptive microtasking*). This saves context switching costs.

A microtasked program should always work regardless of the number of processors.

Microtasks are not tasks and, for this reason, only a certain class of synchronization patterns is valid between them.

Microtasking is usually implemented in the run-time library. There are portable libraries (e.g. SCHEDULE and The Force) and machine-specific libraries (e.g. for Sequent and Cray)
The **DOALL** statement

- Each iteration is a microtask
- Iterations are queued in any order

**e.g.**

```plaintext
doall i=1,n
    X(i) = X(i) + 1
    Y(i) = X(i) + Z(i)
end doall
```

```plaintext
doall i=1,n
    B(i) = A(i)
    do while (B(i) ** 2 - A(i) .gt. epsilon)
        B(i) = (B(i) + A(i) / B(i)) / 2.0
    end do
end doall
```
Critical regions in DOALL loops

There could be interaction between iterations but no producer consumer type of synchronization.

e.g

do i=1,n
    A(K(i)) = A(K(i)) + C(i)
end do

↓

doall i=1,n
    critical region K(i)
        A(K(i)) = A(K(i)) + C(i)
    end critical region K(i)
end doall
**Private variables in DOALL**

In the loop

```plaintext
doall i=1,n
   real x
   x = a(i)*2 + b(i)*2
   c(i) = x + 1
   d(i) = 2*x
end doall
```

there is a copy of x per implicit task.

In the absence of a construct such as `dofirst` (shown on the next page) having a copy per implicit task is equivalent to having a copy per iteration.
Three examples of DOALL

1. IBM Parallel Fortran

sum = 0
ssq = 0
parallel loop 1 i = 1, n
private (psum, pssq)
do first
  psum = 0
  pssq = 0
do every
  do 20 j = 1 , m
    psum = psum + c(j,i)
    pssq = pssq + c(j,i) ** 2
  do final lock
    sum = sum + psum
    ssq = ssq + pssq
 1 continue

Executed once by each implicit task

Executed once per iteration
2. Cray microtasking

cmic$ do global
    do 10 i=1,n
        do 20 j=1,m
            localsum = localsum + a(i,j)
        20 continue
    cmic$ guard
        globalsum = globalsum + localsum
    cmic$ end guard
    10 continue
3. PCF Fortran

```fortran
subroutine ex13 (a,b,sum)
real b(0:100)
gate a
guards a(sum)
unlock (a)
sum = 0.0
parallel do i=1,10
  private t
  t = b(i) * b(i-1)
critical section (a)
    sum = sum + t
critical section
end parallel do
end
```
The Burrough’s FMP DOALL statement

Iterations were completely independent (this restriction was relaxed in later versions)

USING and GIVING clauses specified values to be read (before the DOALL started) and values to be written (after the DOALL completed)

Since iterations did not interact, programs whose only parallel construct was this type of DOALL were determinate

Loop index were tuples of the form (i,j,k). The iteration space was formed by operations between tuples such as the cartesian product (e.g. [1..10].c.[1..20])

Seems to originate form the IVTRAN vector statement.
The **DOACROSS** statement

Is a parallel loop similar to DOALL, but it allows producer-consumer type of synchronization.

Synchronization is allowed from lower to higher iterations since it is assumed that lower iterations are selected first by the implicit tasks.

If synchronization were not from lower to higher iterations, deadlock could occur. Assume for example that the first iteration waits at point $w$ for an event from the second iteration. If there were only one implicit task it would wait forever at $w$ since there is no context switching.
Examples of DOACROSS

1. Cedar Fortran

```fortran
C     *
C  example 1. no delay
C     *
     post (ev(0))
doacross i=1,n
     post(ev(i))
a(i) = b(i) + d(i)
wait(ev(i-1))
x(i) = a(i-1) + 2
end doacross
```

<table>
<thead>
<tr>
<th>P1</th>
<th>a(1)</th>
<th>x(1)</th>
<th>a(4)</th>
<th>x(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>a(2)</td>
<td>x(2)</td>
<td>a(5)</td>
<td>x(5)</td>
</tr>
<tr>
<td>P3</td>
<td>a(3)</td>
<td>x(3)</td>
<td>a(6)</td>
<td>x(6)</td>
</tr>
</tbody>
</table>

...
c  *
c example 2. delay between consecutive iterations

c  *

    post(ev(0))
    doacross i = 1, n
        wait(ev(i-1))
        a(i) = b(i) + a(i-1)
        post(ev(i))
        x(i) = a(i) + 2
    end doacross
c *
c example 3. delay between
c non-consecutive iterations.
c *
post (ev(0))
post (ev(1))
doacross i = 2, n
    wait (ev(i-2))
    a(i) = b(i) + a(i-2)
    post (ev(i))
    x(i) = a(i) + 2
end doacross
* example 4. doubly nested loop

* 

doacross i = 1, n 
integer j 
do j = 1 , n 
wait (ev(i-1,j)) 
a(i, j) = a(i-1, j) + a(i, j-1) 
post (ev(i, j)) 
end do 
end doacross
2. Sequent Fortran (Osterhaug 1987)

\[
\begin{align*}
\text{c$doacross order(gx,gy),}
\text{c$&share(x,xa,xb,y,ya,yb,rho),}
\text{c$&local(dx,dy)}
\text{do 10 i=1,n}
\text{c$order gx}
\text{x(i) = xa(i) + xb(i)}
\text{dx = x(i)-x(i-1)}
\text{c$endorder gx}
\text{c$order gy}
\text{y(i) = ya(i) + yb(i)}
\text{dy = y(i)-y(i-1)}
\text{c$endorder gy}
\text{rho(i)=sqrt(dx*dx+dy*dy+dz*dz)}
\text{10 continue}
\end{align*}
\]

- The \$order construct implies singly-nested parallelism, and causes delay between consecutive iterations.
3. PCF Fortran

```fortran
subroutine ex22 (b,e)
  real b(100), c
  event e(100)
  parallel do i=1,97
    if (i .lt. 4) then
      post e(i)
    else
      clear e(i)
    end if
  end parallel do
  parallel do (ordered) i=4,100
    private c
    c = sin(b(i))
    wait e(i-3)
    b(i) = b(i) + b(i-3) * c
    post e(i)
  end parallel do
end
```
Execution time of DOACROSS when ordered critical sections have constant execution time.

- Consider the loop

```plaintext
doacross i=1,n
  c$order a
    a
  c$endorder a
  c$order b
  ... 
  c$order c
  ... 
  c$order d
  ... 
  c$order e
  ... 
  end doacross
```

Assume its execution time lines have the following form:
which in terms of performance is equivalent to the following time lines:

where a constant delay $D$ between the start of consecutive iterations is evident. This delay is equal to the time of the longest ordered critical section (i.e $D=T(c)$ in this case).
• The execution time of the previous loop using \( n \) processors is:

\[
T(a)+T(b)+nT(c)+T(d)+T(e)
\]

as can be seen next:

\[
T(a)+T(b) \quad nT(c)=nD \quad T(d)+T(e)
\]

In general the execution time when there are as many processors as iterations is

\[
nD+(B-D)=(n-1)D+B
\]

where \( B \) is the execution time of the whole loop body.

\[
S_p = nB/[(n-1)D+B] \approx B/D
\]
• When there are $p < n$ processors the execution time of the loop depends on whether $B \geq pD$ or not.

Case 1: $B \geq pD$

If $p = 3$, for the previous loop we have:

$$T(\text{loop}) = \lceil n/3 \rceil B + T(c)((n-1) \mod 3)$$

In general the formula is:

$$\lceil n/p \rceil B + D((n-1) \mod p)$$
Case 2: B < pD

For the previous loop, and in general we have

\[ T(\text{loop}) = nD + B - D \]
From the previous two statements we have that

\[ T(\text{loop})= \]

if \( B \geq pD \) then \( \left\lfloor \frac{n}{p} \right\rfloor - 1 \) \( B + D \) ((n-1) mod p) + B
else \( (n-1)D + B \)

but

\[ n-1 = p\left(\left\lfloor \frac{n}{p} \right\rfloor - 1\right) + (n-1) \mod p \]

therefore

\[ T(\text{loop})= \]

if \( B \geq pD \) then \( \left\lfloor \frac{n}{p} \right\rfloor - 1 \) \( B + D \) ((n-1) mod p) + B
else \( (p\left(\left\lfloor \frac{n}{p} \right\rfloor - 1\right) + (n-1) \mod p)D + B \)

and

\[ T(\text{loop})= \]

\( \left\lfloor \frac{n}{p} \right\rfloor - 1 \) max(B,pD) + D ((n-1) mod p) + B
The **SPMD model of parallel execution**

- In this model several implicit tasks may execute redundantly some part of the program (Cytron et al 1990)
- For example, assume the following routine is executed using the SPMD model

```plaintext
subroutine sub ( ... a...)
S1: z = a**2
    doall i=1,n
        d(i) = d(i) + z
    end doall
S1: x = sin (a) + cos(a)
S2: y = a ** 3
    doall i =1,n
        c(i) = b(i) + x - y
    end doall
return
```

In this model each implicit task starts at $S_1$ and compute its own (private) copy of $z$. Then the implicit tasks execute the first `doall` by distributing the iterations between themselves. After all implicit tasks complete its share of the first doall, they continue and redundantly execute $S_2$ and $S_3$ (each compute their own copy of $x$ and $y$). Finally, the implicit tasks execute the second `doall`. 
Advantages of the SPMD model

- If in a routine there are several parallel loops or parallel slc constructs, the SPMD model could save some of the overhead for releasing and reallocating processes. This overhead includes swapping registers, allocating control structures and storage, and copying context.

- Main disadvantage is that programming using redundant execution is conceptually more difficult than the conventional form of parallel programming.

- The main difficulty is that redundant execution is not always valid as shown in the next program.

```plaintext
doall i=1,n
    a(i) = b(i) + c(i)
    if (i.eq.n) x = 0  
    \textit{x is a shared variable}
end doall

   x = x+25  \textit{only one implicit task should execute this statement}

doall i=1,n
    a(i) = a(i) + x
end doall
```
Examples of SPMD constructs

1. Cray Fortran

    CMIC$ MICRO
    subroutine nonbon (... enb, ehb, eel, ..
    dimension ... lpack(maxatoms)
    ...
    CMIC$ PROCESS    non-redundant execution
    enb = 0.0
    eel = 0.0
    ehb = 0.0
    CMIC$ END PROCESS
    ...
    lpack (1) = 1
    do 50 i=1,natom-1
      lpack (i+1) = ...
    50 continue
    ...
    CMIC$ DO GLOBAL
    do 1000 i =1, natom-1
    ...

2. PCF Fortran

subroutine ex57 (a, amax, n)
real a(0:n)
amax = 0.0
parallel
    private almax
    pdo (extend) i=1,n
        if (abs(a(i) .gt. abs(almax))
almax = a(i)
    end pdo
    critical section
        if
            (abs(almax).gt.abs(amax)) then
                amax=almax
        end if
    end critical section
end extend
single process
    almax = a(1) + a(n)
    if(amax.lt.almax)amax=1.0+amax
end single process
pdo i=1,n
    a(i) = abs(a(i)/amax)
end pdo
end parallel
end
Loop parallelization and vectorization

A loop whose dependence graph is cycle-free can be parallelized or vectorized.

e.g.

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

```
X(1:N)=B(1:N)+1 PARALLEL DO I=1,N
A(1:N)=X(1:N)+1 X(I)=B(I)+1
   A(I)=X(I)+1
END PARALLEL DO
```
**Pattern matching**

- Some program patterns occur frequently in programs. A parallelizer can use pattern-matching to identify and replace them with a parallel algorithm.

  e.g.

  
  ```
  DO I=1,N
  A(I)=A(I-1)+B(I)
  END DO

  A(1:N)=REC1N(B(1:N),A(0),N)
  ```

  ```
  X=A(1)
  DO I=2,N
  IF(X.GT.A(I))X=A(I)
  END DO
  ```

  ```
  X=MIN(A(1:N))
  ```
**DO to DOALL transformation**

- When the dependence graph inside a DO loop has no cross-iteration dependences, it can be transformed into a DOALL.

Example 1:

```plaintext
do i=1,n
S_1: a(i) = b(i) + c(i)
S_2: d(i) = x(i) + 1
end do
```

Example 2:

```plaintext
do i=1,n
S_1: a(i) = b(i) + c(i)
S_2: d(i) = a(i) + 1
end do
```
Example 3:

\[\text{do } i=1, n\]
\[S_1: \quad b(i) = a(i)\]
\[S_2: \quad \text{do while } b(i) + a(i)/b(i) / 2.0 \gt \text{epsilon}\]
\[S_3: \quad b(i) = (b(i) + a(i)/b(i)) / 2.0\]
\[\text{end do while}\]
\[\text{end do}\]
• When there are cross iteration dependences, but no cycles, do loops can be *aligned* to be transformed into DOALLs

Example 1:

```plaintext
do i=1,n
  S₁:  a(i) = b(i) + 1
  S₂:  c(i) = a(i-1)**2
end do

↓

do i=0,n
  S₁:  if i>0 then a(i) = b(i) + 1
  S₂:  if i<n then c(i+1) = a(i)**2
end do
```
Sometimes we have to replicate to achieve alignment

Example 2:

do i=1,n
    a(i) = b(i) + c(i)
    d(i) = a(i) + a(i-1)
end do

↓

do i=1,n
    a(i) = b(i) + c(i)
    a1(i) = b(i) + c(i)
    d(i) = a1(i) + a(i-1)
end do

↓

do i=0,n
    if i>0 then a(i) =b(i) + c(i)
    if i<n then a1(i+1)=b(i+1)+c(i+1)
        d(i+1)=a1(i+1)+a(i)
end do
Need for replication could propagate.

Example 3:

```plaintext
do i=1,n
    c(i) = 2 * f(i)
    a(i) = c(i) + c(i-1)
    d(i) = a(i) + a(i-1)
end do

↓

do i=1,n
    c(i) = 2 * f(i)
    c1(i) = 2 * f(i)
    c2(i) = 2 * f(i)
    a(i) = c2(i) + c(i-1)
    a1(i) = c1(i) + c(i-1)
    d(i) = a1(i) + a(i-1)
end do
```

The problem of finding the minimum amount of code replication sufficient to align a loop is NP-hard in the size of the input loop (Allen et al 1987)
To do alignment, we may need to do topological sort of the statements according to the partial order given by the dependence graph.

Example 4:

```
    do i=1,n
        S_1: a(i) = b(i) + c(i-1)
        S_2: c(i) = d(i)
    end do
```

Performing alignment without sorting first will clearly be incorrect in this case.
Another method for eliminating cross-iteration dependences is to perform loop distribution.

Example:

\[
\begin{align*}
\text{do } & i=1,n \\
& a(i) = b(i) + 1 \\
& c(i) = a(i-1) + 2 \\
\text{end do}
\end{align*}
\]

\[
\downarrow
\]

\[
\begin{align*}
\text{do } & i=1,n \\
& a(i) = b(i) + 1 \\
\text{end do} \\
\text{do } & i=1,n \\
& c(i) = a(i-1) + 2 \\
\text{end do}
\end{align*}
\]
Loop Coalescing for DOALL loops

- A perfectly nested DOALL loop such as

\[
\begin{align*}
\text{doall } i &= 1, n1 \\
\text{doall } j &= 1, n2 \\
\text{doall } k &= 1, n3 \\
&\vdots \\
\text{end doall} \\
\text{end doall} \\
\text{end doall}
\end{align*}
\]

could be trivially transformed into a singly-nested loop with a tuple of variables as index:

\[
\begin{align*}
\text{doall } (i, j, k) &= (1..n1).c.(1..n2).c.(1..n3) \\
&\vdots \\
\text{end doall}
\end{align*}
\]

This coalescing transformation is convenient for scheduling and could reduce the overhead involved in starting DOALL loops.
Cyclic dependences -- DOACROSS

A loop with cyclic dependences can be transformed into DOACROSS as shown next:

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

↓

c$doacross order(aa,bb),share(a,b,c)$

do i=1,n

c$order aa$
\[
\begin{align*}
a(i) &= b(i) + a(i-1) \\
end order aa \\
\end{align*}
\]

c$order cc$

c$endorder cc$
\[
\begin{align*}
c(i) &= a(i) + c(i-1) \\
end do \\
\end{align*}
\]

DOACROSS has the advantage that all implicit tasks execute the same code. This facilitates code assignment.

Other advantage of the DOACROSS construct over the DOPIPE construct are illustrated in the following examples.
Example 1:

The same translation works for two or three processors:

Two processors

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

Three processors

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
Example 2:

- Increasing the number of processors improve performance
Example 3

When the following loop is executed as a doacross on two processors

\[
\text{do } i=1,n \\
\text{S: } a(i) = b(i-2) - 1 \\
\text{T: } b(i) = a(i-3) * k \\
\text{end do}
\]

we get the following time lines (\(S^i\) stands for statement S in iteration i)

<table>
<thead>
<tr>
<th>Proc.</th>
<th>1</th>
<th>(S^1)</th>
<th>(T^1)</th>
<th>(S^3)</th>
<th>(T^3)</th>
<th>(\ldots)</th>
<th>2</th>
<th>(S^2)</th>
<th>(T^2)</th>
<th>(S^4)</th>
<th>(T^4)</th>
</tr>
</thead>
</table>

Cycle shrinking takes place automatically.

This is also true in the case of multiply-nested loops where all what is needed is to use a tuple as the loop index as in

\[
\text{doacross } (i,j,k)=[1..n_1].c.[1..n_2].c.[1..n_3]
\]
Example 4:

The following loop

\[
\begin{array}{c}
do \ i=1,n \\
\quad \ do \ j=1,n \\
\quad \quad \ S: \quad a(i,j) = a(i-1,j) + a(i,j-1) \\
\quad \quad \quad \ end \ do \\
\end{array}
\]

\[
\begin{array}{c}
end \ do \\
\end{array}
\]

can be translated into the following doacross loop:

\[
\begin{array}{c}
doacross \ i=1,n \\
\quad \ do \ j=1,n \\
\quad \quad \ wait \ (ev(i-1,j)) \\
\quad \quad \ S: \quad a(i,j) = a(i-1,j) + a(i,j-1) \\
\quad \quad \quad \ post \ (ev(i,j)) \\
\quad \quad \quad \ end \ do \\
\end{array}
\]

\[
\begin{array}{c}
end \ doacross \\
\end{array}
\]
The iteration space of the previous loop is:

\[
\begin{align*}
S_{1,1} & \rightarrow S_{1,2} \rightarrow S_{1,3} \rightarrow S_{1,4} \\
S_{2,1} & \rightarrow S_{2,2} \rightarrow S_{2,3} \rightarrow S_{2,4} \\
S_{3,1} & \rightarrow S_{3,2} \rightarrow S_{3,3} \rightarrow S_{3,4} \\
S_{4,1} & \rightarrow S_{4,2} \rightarrow S_{4,3} \rightarrow S_{4,4}
\end{align*}
\]

and its time lines when executed on \( n \) processors are:

\[
\begin{array}{ccc}
S_{1,1} & S_{1,2} & S_{1,3} \\
S_{2,1} & S_{2,2} & S_{2,3} \\
S_{3,1} & S_{3,2} & S_{3,3}
\end{array}
\]
Statement ordering and DOACROSS execution time.

- Consider the following dependence graph for the body of a singly-nested do loop.

- When the DOACROSS body has the original statement order, there is no speedup ($S_1$ of iteration $i+1$ cannot start executing until $S_5$ of iteration $i$ completes execution).

- When the body is permuted into the order $S_1 S_2 S_4 S_5 S_3$, then there will be speedup as shown in the following time lines.
• Selecting an optimum statement ordering to minimize the delay is NP-Hard (Cytron 1989).
References


P. Wegener. The Vienna Definition language. Computing Surveys 4(1). 1971


M. Wolfe and U. Banerjee. Data dependence and its applications to parallel processing. IJPP 16(2). 1987