Input Loop L with dependence graph $D = DG(L)$ and statement set $STAT$.

Output A parallelized program L_{par}, equivalent to L.

Method

procedure parallelize(*R*,*c*,*doall_flag*);

/* *R* is a set of assignment statements in *L*, $c \in \mathbb{N}$ is a level, and *doall_flag* is a boolean output parameter that is used to signal to the calling procedure whether or not this call generates a doall-loop.

This procedure (excluding recursive calls) generates code at level *c* for *R*. If $c > 1$, then the statements in *R* are contained in serial regions at all levels $\leqslant c - 1$, and sequential do-loops have been created at these levels.

We use the following notation:

- $G' = (N', E')$: the acyclic condensation of the level *c* dependence graph, restricted to *R*
- $visited \subseteq N'$. At any time, the value of *visited* is the set of all regions that have already been processed by the call
- *initials* is a shorthand for the expression: $\{n' : (n' \in N' - visited) \wedge (pred(n') \subseteq visited)\}$

- *new_regions*: an auxiliary variable whose values are sets of regions
- $CLUSTER \subseteq N'$: a set of regions that can be combined
- *old_CLUSTER*: an auxiliary variable needed for the stabilization test
- *STAT*(*SEG*): the set of statements contained in a segment *SEG*
- *local_flag*: a local boolean variable
- a call generate_body(*STATS*,*c*) generates serial code for the set of statements in *STATS* at all levels $\geqslant c$. */

```
begin
G' = (N',E') := AC(constrain(D,c)|R);
visited := ∅;
doall_flag := false;

  /* The while loop below performs exactly h iterations, where h ≥ 1 is the
     cardinality of the cluster partition. Iteration j (1 ≤ j ≤ h) first constructs C_j
     (collecting its elements in the variable CLUSTER) and then generates
     code for the cluster */

while initials ≠ ∅ do

  /* Step 1: construct the next cluster C_j */

  CLUSTER := initials;
  visited plus initials;
  repeat
    old_CLUSTER := CLUSTER;
    new_regions := {n': pred(n') ⊆ visited ∧
                        ∀ n'' ∈ pred(n') ∩ CLUSTER:
                        (n'',n') is a barrier-free edge};
    CLUSTER plus new_regions;
    visited plus new_regions
  until old_CLUSTER = CLUSTER;

  /* Step 2: generate code for C_j */

  for every segment SEG ⊆ CLUSTER in topological order do
    if SEG is scalar at level c
    then generate_body(STAT(SEG),c)
    else  /* SEG is serial or parallel: let I_c, T_c and U_c be the do-variable, the
             lower and the upper bound of the loop associated with SEG at
             level c */
      if SEG is serial at level c
      then generate(" DO I_c = T_c,U_c ");
           parallelize(STAT(SEG),c + 1,local_flag);
           if local_flag
           then  /* a doall-loop has been generated at level c + 1: this
                    enforces synchronization for the level c loop (see
                    Example 7.4) */
             generate(" IF I_c < U_c THEN BARRIER FI ")
           fi;
           generate(" END DO I_c ")
```

```
        else   /* SEG is parallel at level c */
          doall_flag := true;
          generate(" DOALL I_c = T_c,U_c ");
          generate_body(STAT(SEG),c + 1);
          generate(" ENDALL I_c ")
        fi
      fi
    end for;
    /* Step 3: if this is not the last cluster, generate a barrier */
    if visited ≠ N' then generate(" BARRIER ") fi
  end while
  end   /* parallelize */
  /* MAIN PROGRAM */
  begin parallelize(STAT,1,doall_flag) end
```