Algorithm 1, Part 2: Code generation.

Input:

- Sets Sources(i), 1≤i≤q, represented with affine forms, as produced by Algorithm 1, Part 1; relation R as the result of step 2 of Algorithm 1, Part 1.

Output:

- Code scanning synchronization-free slices and the operations of each slice in lexicographical order for the SCC.

Method:

```
begin
foreach i, 1≤i≤q do
    genLoops (in: Sources(i); out: OuterLoops, L_I);
    foreach I in L_I do
        S_Slice := R* (R_UCS(i)* (I))
        /* note that if R_UCS(i) = ∅ ,  then  R_UCS(i)*(I) ==I */
        genLoops (in: S_Slice; out: InnerLoops, L_J);
        foreach J in L_J  do
            genLoopBody (in: OuterLoops, InnerLoops, J; out: LoopBody);
end
```

where

- *genLoops (in: OperSet; out: Loops, VectorList)* generates set *Loops* of loop nests to scan operations comprised in set *OperSet*, and returns a list of corresponding parameterized iteration vectors *VectorList* comprising the loops index values. To implement function *GenLoops* any well-known technique, such as [,] can be applied.
- *genLoopBody(in: OuterLoops, InnerLoops, Iter; out: LoopBody)* generates body *LoopBody* of *InnerLoops* containing the statements of the source loop to be executed at iteration *Iter* and inserts loops *InnerLoops* in the corresponding nest of loops *OuterLoops*. To generate *LoopBody*, the last elements of tuples representing set *InnerLoop* are taken into consideration (they represent the statement identifiers and allow for choosing appropriate statements to be inserted in the loop body).

An Illustrative Example

To illustrate Algorithm 1, let us consider the following example, which is a slight modification of an example proposed in [].

Example 1.

```
for (i=1; i<=n;i++)

   for (j=1;j<=n;j++) {

s1:        a(i,j)=a(i,j)+ b(i-2,j);

s2:        b(i,j)=a(i,j-2)* b(i,j); }
```

Petit extracts the following dependence relations for this example (see Figure 2).

R_{12}:={[i,j] -> [i,j+2] : 1 <= i <= n && 1 <= j <= n-2};
R_{21}:={[i,j] -> [i+2,j] : 1 <= i <= n-2 && 1 <= j <= n};

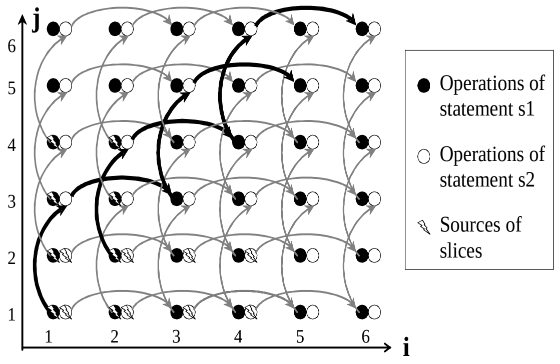

Figure 2. Dependences for Example 1 when n=6

To extract slices, to the source loop we can apply the following affine transformations Φ1 and Φ2 for statements s1 and s2, respectively: Φ1=i-j, Φ2=i-j+2. The transformed loop, for n=6, represents 10 independent threads (there is no other affine transformation extracting more than 10 slices). While analyzing Figure 2, we can see that there actually exist 16 independent slices in the source loop. The loop transformed on the basis of the above affine transformations combines some synchronization-free slices together. For example, slices with sources in s1 (1,1) and s1 (2,2) (shown in black boldface in Figure 2) are mapped to the same partition. Our approach can be applied to this example as follows, thus extracting all available slices.

Algorithm 1. Part 1.

1. R_{12}:={[i,j,1]→[i,j+2,2] : 1≤i≤ n & 1≤j≤ n-2};
   R_{21}:={[i,j,2]→[i+2,j,1] : 1≤i≤ n-2 & 1≤j≤ n}.
2. R:={[i,j,2]→[i+2,j,1]: 1≤j≤n & 1≤i≤ n-2}∪{[i,j,1]→[i,j+2,2]:1≤j≤n-2 & 1≤i≤n}.
3. UDS(1):= {[i,j,1]: 1≤j≤n-2 & 1≤i≤2};
   UDS(2):= {[i,j,2]: 1≤j≤2 & 1≤i≤ n-2}.
4. UDS:= {[i,j,1]: 1≤j≤n-2 & 1≤i≤2} ∪ {[i,j,2]: 1≤j≤2 & 1≤i≤n-2}.
5. *For statement 1:*
   R_UCS(1)=∅; Sources(1)=UDS(1)={[i,j,1]: 1≤j≤n-2 & 1≤i≤2};
   For statement 2:
   R_UCS(2)=∅; Sources(2)=UDS(2)={[i,j,2]: 1≤j≤2 & 1≤i≤ n-2}.

Algorithm 1. Part 2.

For statement s1:

Generate <u>outer loops</u> to scan the lexicographically minimal sources of synchronization-free slices and being represented with operations of statement s1 (let us note that for n<3 no slices are presented, thus no code is generated).	`if(n>=3) {`
	`  parfor(t1=1;t1<=2;t1++)`
	`    for(t2=1;t2<=n-2;t2++)`
	`      s1(t1,t2,1); }`
	
	List L_I contains vector I equal to (t1,t2,1)'.

Generate <u>inner loops</u> to enumerate operations belonging to the slice with a source represented by vector I=(t1,t2,1)' and being contained in set S_Slice := R* (R_UCS* (I)) = {[i,t2,1]: i=t1}∪{[t1,t2+2,2]: 1≤t1≤n & 1≤t2≤n -2}∪{[i,t2-t1+i,1]:∃ (alpha: 0 = t1+i+ 2alpha & 1≤t1≤i-2 & i≤n & 1≤t2 & t2+i≤t1+n)} ∪ {[i,t2-t1+i+2,2] :∃ (alpha : 0=t1+i+2alpha & 1≤t1≤i-2 & i≤n & 2+t2+i≤t1+n & 1≤t2)}.	`s1(t1,t2,1);`
	`s1(t1,t2+2,2);`
	`for(t3=t1+2;t3<=min(n,t1-t2+n-2);t3+=2) {`
	`  s1(t3,t2-t1+t3,1);`
	`  s1(t3,t2-t1+t3+2,2);`
	`}`
	`for(t3=max(2*intDiv(2t1-t2+n,2)-t1,t1+2);`
	`    t3<min(-t2+t1+n,n); t3+=2)`
	`  s1(t3,t2-t1+t3,1);`

Generate the <u>body of inner loops</u> containing statements of the source loop body to be executed at iteration J, and <u>insert it</u> as the body of outer loops.	`if(n>=3) {`
	`parfor(t1=1;t1<=2;t1++)`
	` for(t2=1;t2<=n-2;t2++) {`
	`  a(t1,t2)=a(t1,t2)+b(t1-2,t2);`
	`  b(t1,t2)=a(t1,t2)*b(t1,t2);`
	`  for(t3=t1+2;t3<=min(n,t1-t2+n-2);t3+=2){`
	`    a(t3,t2-t1+t3)=a(t3,t2-t1+t3)+`
	`                   b(t3-2,t2-t1+t3);`
	`    b(t3,t2-t1+t3+2)=a(t3,t2-t1+t3)*`
	`                     b(t3,t2-t1+t3+2);}`
	`  for(t3=max(2*intDiv(2t1-t2+n,2)-t1,t1+2);`
	`      t3<min(-t2+t1+n,n);t3+=2)`
	`    a(t3,t2-t1+t3)=a(t3,t2-t1+t3)+`
	`                   b(t3-2,t2-t1+t3);}}`

For statement s2:

Generate <u>outer loops</u> to scan sources of synchronization-free slices and being operations of statement s2.	`parfor(t1=1; t1<=n-2; t1++)`
	`  for(t2=1; t2<=2; t2++)`
	`    s1(t1,t2,2);`
	List L_I contains vector I equal to (t1,t2,2)'.

Generate <u>inner loops</u> to enumerate iterations belonging to the slice with a source represented by vector I=(t1,t2,2)' and being contained in set S_Slice := R* (R_UCS*(I)) ={[i,t2,2]: i=t1}∪{[t1+2, t2,1]: 1≤t1 ≤n-2 & 1≤t2≤n}∪{[i,t2-t1+i,2]:∃ (alpha: 0=t1 +i+2alpha &	`s1(t1,t2,2);`
	`s1(t1+2,t2,1);`
	`if (t2 <= n-2) s1(t1+2,t2+2,2);`
	`for(t3=t1+4;t3<= min(t1-t2+n+2,n);t3+= 2){`
	`  s1(t3,t2-t1+t3-2,1);`
	`  s1(t3,t2-t1+t3,2);`

1≤t1≤i-2 & i≤n & 1≤t2 & t2+i≤t1+n)} ∪ {[i,t2-t1+i-1]: ∃ (alpha:2alpha=t1+i&1≤t1 ≤ i-4 & i≤n & t2+i≤2+t1+n & 1≤t2)}.	**}**

Generate the body of inner loops containing statements of the source loop body to be executed at iteration J, and insert it as the body of outer loops.	**parfor**(t1=1; t1<=n-2; t1++)
	 for(t2=1; t2<=2; t2++) **{**
	  b(t1,t2)=a(t1,t2-2)*b(t1,t2);
	  a(t1+2,t2)=a(t1+2,t2)+b(t1,t2);
	  if (t2<=n-2) s1(t1+2,t2+2,2);
	  for(t3=t1+4;t3<=min(t1-t2+n+2,n);t3+=2)**{**
	    a(t3,t2-t1+t3-2)=a(t3,t2-t1+t32)+
	                 b(t3-2,t2-t1+t3);
	    b(t3,t2-t1+t3)=a(t3,t2-t1+t3-2)*
	               b(t3,t2-t1+t3); **}}**

Note that the codes enumerating sources of slices can be executed in parallel, this is denoted with keyword “parfor”