Parallelizing and Optimizing Compilers

S.Orlando

August 22, 1994

Contents

In this chapter some compiling techniques to parallelize and to optimize programs for several parallel architectures are surveyed.

Most work in parallelizing compilers is concerned with detection and extraction of parallelism at the *vector* level. This kind of parallelism can be extracted from number-crunching, sequential programs. In fact, the main interest in this kind of compilers derives from the necessity of parallelizing dusty-deck programs, usually writter in Fortran. Similar techniques are used to parallelize for shared-memory multiprocessors. These parallelizing techniques can be considered as an extension and an adaptation of the vectorization technology. They discover the possible parallel activities, allowing certain nested loops to be converted to tasks. In this kind of architectures, however, once determined the parallel tasks, in order to obtain efficient implementations, the most difficult problems to be solved are the reduction of the possible collisions generated by concurrent accesses to shared memory.

Other important parallelization techniques are those proposed to generate parallel code for VLIW (Very Long Instruction Word) machines. The sequential programs on which VLIW compilers work well are of the same kind as the programs from which a large number of vector operations can be extracted. In principle, however, also less regular parallelism can be exploited. Similar techniques are also adopted by current compilers for superscalar or superpipelined RISC processors.

The automatic techniques to synthesize systolic algorithms from nested loops can be also considered as parallelizing compiling techniques. The architecture exploited in this case is very different from previous ones, because no shared memory is provided for the processing elements of systolic arrays.

Finally, we illustrate some other techniques, used to efficiently compile onto distributed-memory MIMD multiprocessors. However, they are not truly parallelizing techniques, as the programmers is asked to specify where is the parallelism in its program, and which data distribution is the most useful to exploit that kind of paralleism. These innovative compiling technique derive an optimized message-passing program that implement a shared-memory parallel program.

The data-flow analysis constitutes the basis of the techniques we are going to illustrate. Remember that in a sequential program the ordering of instructions is given by the flow of control. The data-flow analysis determines whether such ordering can be modified without changing the semantics of the program. The analysis identifies only the critical dependencies which needs to be obeyed. Since these dependencies produce a partial order, the instructions between which no dependency relationships exist can be executed in any order and also in parallel.

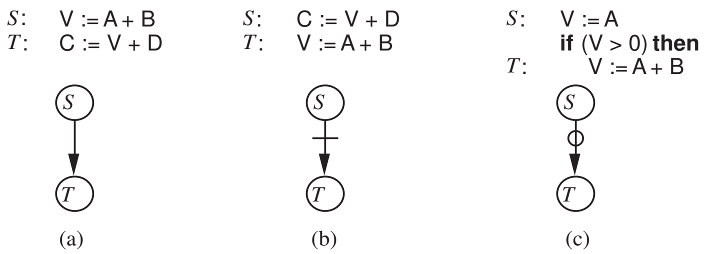


Figure 1: Examples of dependency relationships and their graphical representation. Statement T is (a) *flow-dependent*, (b) *anti-dependent* or (c) *output-dependent* on statement S.

1 Dependency analyses for parallelization

In this section we present a careful dependency analysis of sequential code, supposed to be expressed by means of an intermediate code. This is the analysis to be used when sequential code must be parallelized. The following definition states the dependency relationships that may exist between two statements belonging to a loop-free portion of code [17]:

Definition 1 *Let S and T be two assignments in a sequential portion of code, such that S precedes T in the sequence. The following dependency relationships may exist between S and T:*

a) T is flow-dependent *(also* data *or* true-dependent*) on S iff there exists a variable* `V` *such that S defines* `V`*, T uses* `V` *(i.e. read* `V`*), and S is a reaching definition of T (see Figure 1.(a));*

b) T is anti-dependent *on S iff there exists a variable* `V` *such that S uses* `V`*, and T defines* `V` *(see Figure 1.(b));*

c) T is output-dependent *on S iff there exist a variable* `V` *such that both S and T define* `V` *(see Figure 1.(c)).*

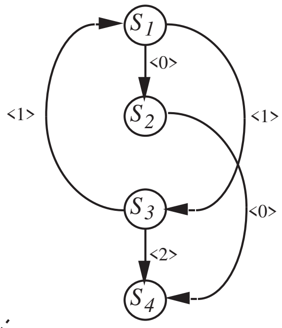
The dependency relationships between the operations of a loop-free portion of code may be represented by a *Direct Acyclic Graphs* (DAG), where nodes represent statements, and arcs dependencies. Figure 1 shows the arcs associated with each kind of dependency.

It is worth pointing out that the antidependencies and output dependencies can mainly be removed by introducing further variables.

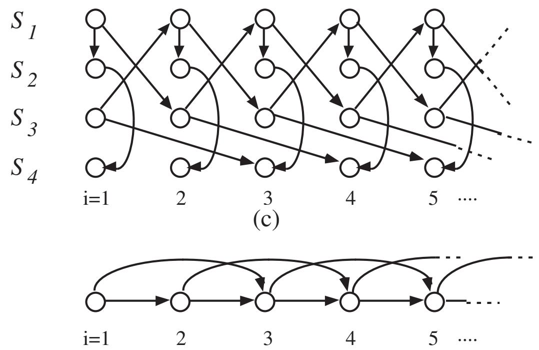
Dependencies in multi-nested loops The same definitions of dependency relationships may be extended to the body of a multi-nested loop. In order to

```
do  i = 1, N
      S1:  A[i] := B[i] + C[i - 1]
      S2:  D[i] := A[i] * 2
      S3:  C[i] := A[i - 1] + C[i]
      S4:  E[i] := D[i] + C[i - 2]
enddo
```

(a)



(b)



(c)

(d)

Figure 2: (a) Example of a loop with dependencies that cross iterations. (b) Dependency graph of the loop whose flow dependency arcs are labelled by distance vectors, corresponding to the number of iterations that are crossed. (c) Iteration-space-dependency graph whose nodes represent the various instances of every statement in the loop body. (d) Iteration-space-dependency graph showing dataflow from one iteration of the loop body to the following ones.

do this, it is necessary to characterize all the possible executions of the same statement with respect to each iteration of the loop body. A multi-nested loop is of the form:

```
do  i1 = 1, n1
    do i2 = 1, n2
       ...
         do ik = 1, nk
            <Loop body>
         enddo
       ...
    enddo
enddo
```

The body of the loop may be executed for several times, and each execution may be associated with a *k-tuple*, which corresponds to the values of indexes: **i1**$, \ldots ,$ **ik**. In the same way, also statements S and T inside the loop body may be characterized by the values of these indexes. Thus, $S_{l1,\ldots,lk}$, called an instance of S, corresponds to the execution of S when **i1** $= l1, \ldots,$ **ik** $= lk$. Consider two statements, and a pair of their instances. Let S and T be the two statements, and $S_{l1,\ldots,lk}$ and $T_{m1,\ldots,mk}$ the corresponding istances. Suppose the two instances do not correspond to the same iteration of the loop body. Hence, between these instances, there exists an order entailed by the execution order of the corresponding iteration of the loop bodies. If $S_{l1,\ldots,lk}$ is executed before $T_{m1,\ldots,mk}$, the same dependency relationships defined by Definition 1 may occur between them.

Dependencies between statement instances that are executed during distinct iterations of a loop are said dependencies *crossing* from one iteration to another, or *loop-carried dependencies*. In the graphical representation of these dependencies a single node may be used for every statement, even though there exist several instances of it. So, a flow, anti, or output dependency arc exists from the node corresponding to statement S to the node corresponding to T iff an instance of T is, respectively, flow dependent, anti-dependent, or output dependent on an instance of S. An example of this graph is shown in Figure 1.(b). The graph corresponds to the loop in Figure 1.(a). The arcs are labelled by *distance vectors*, representing the number of iterations that are crossed by the dependency relationships. As an example, suppose $T_{m1,\ldots,mk}$ depends on $S_{l1,\ldots,lk}$. Thus the *k-tuple* $< v1, \ldots, vk >$, where $< v1, \ldots, vk > = < m1 - l1, \ldots, mk - lk >$, is the distance vector to be associated with the arc from S to T. Note that the distance vectors associated with loop-carried dependency arcs cannot be equal to $< 0, \ldots, 0 >$.

On the other hand, another graph, called *iteration-space-dependency*, is characterized by distinct nodes for every instance of the statements involved, or for every iteration of the loop body. Examples of this type of graph are shown in Figure 1.(c) and Figure 1.(d). Note that in both graphs one dimension corre-

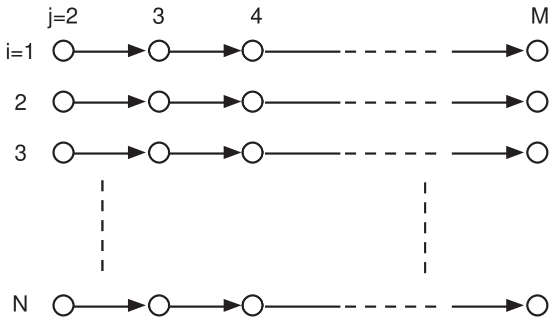


Figure 3: *Iteration-space-dependency* graph for the nested loop (2), where each node represents a new instance of statement S.

sponds to the various iterations of the loop. Such graphs are usually used to represent flow dependency relationships.

Dependencies in loops may be classified as *forward* and *backward*. A dependency relationship is called *forward* if the corresponding arc in the dependency graph goes from a statement S to a statement T, and T occurs later in the *lexical ordering*, otherwise a dependency is called *backward*. Note that all the dependencies represented in the graph in Figure 1.(b) are forward, except for the dependency corresponding to the arc going from $S3$ to $S1$.

2 Vectorizing techniques

Vector machines have pipelined execution units and fast memory that can be exploited to efficiently execute long sequences of independent identical operations.

As an example, the multiplication of two N-element floating-point vectors requires the execution of a sequences of independent identical operations. The corresponding sequential program is:

```
do  i=1, N
    A[i]  :=  B[i] * C[i]
enddo
```

In a vector machine this inner loop can be executed in parallel, so that it can be replaced by the following instruction:

```
A[1:N] :=   B[1:N] * C[1:N]}
```
(1)

The semantics of the above instruction states that all the multiplications involving corresponding elements of `B` and `C` are to be executed in parallel, even though vector machines execute these multiplications in pipeline. However, according to the terminology introduced in Section 1, to apply the transformation of the loop, no *loop-carried* data dependencies have to exist between distinct instances of the statements in the loop body [1].

It is possible that loop-carried dependencies in some nested-loops do not allow the inner loop to be replaced by a vector instruction such as (1). As an example, consider the following nested loop:

```
do i = 1, N
   do j = 2, M
      S:   A[i,j]  :=  A[i,j-1] * B[i]                    (2)
   enddo
enddo
```

The iteration-space-dependency graph in which every node represents an iteration of the loop body, is shown in Figure 2. Note that, for every $i \in \{1, \ldots, N\}$ and $j \in \{2, \ldots, M-1\}$, $S_{i,j+1}$ flow depends on $S_{i,j}$. Since these instances of S correspond to distinct iterations of the inner loop, there exist flow dependencies that cross the iterations and do not allow the vectorization. This loop may become vectorizable by interchanging the i and the j indexes:

```
do j = 2, M
   do i = 1, N
      S:  A[i,j]  :=  A[i,j-1] * B[i]
   enddo
enddo
```

In this case, for every $i \in \{1, \ldots, N\}$ and $j \in \{2, \ldots, M-1\}$, $S_{j+1,i}$ flow depends on $S_{j,i}$, so that, considering the inner loop, no dependencies exist between distinct iterations. Therefore, the vectorization of this loop can take place:

```
do  j = 2, M
    A[1:N, j]  :=  A[1:N, j-1] * B[1:N]
enddo
```

Loop partitioning In some cases only a portion of the loop body can be vectorized. This case occurs when loop-carried dependencies involve only some statements included in the body, while the other statement are independent of each other. Consider the following loop:

```
do i = 1, N
   S1:  A[i] := B[i] * C[i]
   S2:  ACC := ACC * A[i]
enddo
```

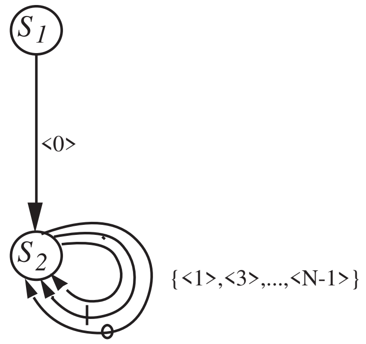


Figure 4: Dependency graph for the loop to be partitioned.

Figure 2 shows the dependency graph of this loop. Note that:

i) statement $S1$ does not depend on any other statements;

ii) statement $S2$ flow depends on $S1$. This dependency does not cross the iterations of the loop because the distance vector is $< 0 >$;

iii) statement $S2$ flow, anti and output depends on itself. All these dependencies are loop-carried because the distance vectors are not equal to $< 0 >$.

The loop may be partitioned into two loops as follows:

```
do i = 1, N
   S1:  A[i] := B[i] * C[i]
enddo
do i = 1, N
   S2:  ACC := ACC * A[i]
enddo
```

Since in the former loop there are no loop-carried dependencies, this loop may be replaced by the following vector instruction:

```
A[1:N] := B[1:N] * C[1:N]
```

3 Parallelizing for multiprocessors

The techniques [17, 15, 19] used to parallelize program loops for shared-memory multiprocessors may be considered as an extension of the vectorization techniques. In fact, the data-flow analyses are the same as those adopted to find out independent identical operations to be replaced with vector instructions.

However, the grain size of parallelism efficiently exploited by multiprocessors is coarser than the size of vector machines. Compilers have to look for tasks that include several elementary operations and that can executed in parallel by distinct processors.

The techniques surveyed try to transform sequential loops into two types of concurrent loops: **doall** and **doacross**. The sequential loops transformed by these techniques are singly-nested loops. In both concurrent constructs, the various iterations of a loops become tasks allocated to distinct processors. In the former construct no dependency exists between the iterations, so that no synchronization between the processors is necessary. The latter construct is usually used when synchronization is needed due to some dependencies crossing iterations, i.e. an iteration of the loop may wait for a signal from a previous iteration to proceed.

The following loop can be transformed into a **doall** loop, because in the dependency graph

- there are no cycles;
- the distance vectors are all zero.

```
do i = 2, N                          doall i = 2, N
   A[i] := B[i] * C[i]                  A[i] := B[i] * C[i]
   if  A[i] < 0                         if  A[i] < 0
       A[i] := A[i] + D        ⇒            A[i] := A[i] + D
   endif                                endif
enddo                                end doall
```

Conversely, in the following loop some dependencies cross iterations, because in the dependency graph the distance vectors are not all zero. Hence the loop can be transformed into a **doacross** loop, in which some synchronization primitives are inserted to obey the loop-carried dependencies.

```
                                   doacross i = 2, N
                                      S1:  A[i] := B[i] * C[i]
do i = 2, N                                send(A_SYNC, i)
   S1:  A[i] := B[i] * C[i]                if (I>2)
   S2:  if (A[i-1] < 0)                       wait(A_SYNC, i-1)
   S3:     A[i] := A[i] + D        ⇒       endif
        endif                         S2:  if (A[i-1] < 0)
enddo                                 S3:     A[i] := A[i] + D
                                           endif
                                   end doacross
```

The **send** and the **wait** instructions are the synchronization primitives that have been inserted into the parallel doacross loop. These synchronization instructions are defined as follows

- the `send` instruction signals that a given event has occurred, where the event is identified by a label (`A_SYNC` in this case) and by the iteration number (`i` in this case);
- the `wait` instruction blocks execution until some other processor executes the corresponding send instruction, that is the instruction with the same label and the same iteration number.

The algorithm to insert synchronization instructions is very simple, and exploits the distance vectors associated with loop-carried dependencies. In fact, given two statement S and T, such that T depends on S and D is a distances associated with this dependency:

i) a `wait(LABEL, i - D)` is inserted just before statement T to ensure that S is executed first;

ii) a `send(LABEL,i)`is placed just after S to signal that S has been executed.

This is the algorithm applied to insert the synchonization instructions into the example above. In fact the statement $S2$ flow depends on statements $S1$, and it is straightforward to verify that this loop-carried dependency is associated with a distance vector $< 1 >$.

The execution of the statements inside the doacross loop is overlapped. In fact processors access data as follows:

	P1	**P2**	**P3**	...	*time*
S1 :	`A[2]` := ..	`A[3]` := ..	`A[4]` := ..		0
S2 :	`ifA[1]` < `0`	`ifA[2]` < `0`	`ifA[3]` < `0`		1
S3 :	`A[1]` := ..	`A[2]` := ..	`A[3]` := ..		2

Since some delays may be introduced by the synchronization primitives, the diagram above must be considered idealized.

Synchronization removing In order to avoid the use of `doacross` constructs along with the insertion of synchronization primitives, we may exploit some techniques to remove loop-carried dependencies. Such techniques are called *loop distribution* and *loop alignment*.

The *loop distribution* technique can be exploited in case the dependency graph associated with the loop body is acyclic. The loop must be distributed to produce new loops without loop-carried dependencies. The following example explains this transformation:

```
do i = 1, N
   A[i] := B[i] * C[i]
   D[i] := A[i-1] + 2
enddo
```

⇒

```
doall i = 1, N
      A[i] := B[i] * C[i]
end doall

doall i = 1, N
      D[i] := A[i-1] + 2
end doall
```

Note that `doall` constructs are thought of as a forking of the various iterations to distinct processors, and a following join that synchronizes all the processors before continuing the execution of the next `doall` loop.

The *loop alignment* technique shifts the access patterns of some statements, i.e. the indexes of the arrays referred to in the statements. This allows loop-carried dependencies to be removed. Consider the following example and the transformation after the alignment:

```
do i = 1, N
   S1: A[i+1] := B[i] * C[i]
   S2: D[i] := A[i] + 2
enddo
```

⇒

```
doall i = 1, N+1
   if (i >= 2)
      S1:  A[i] := B[i-1] * C[i-1]
   endif
   if (i <= N)
      S2:  D[i] := A[i] + 2
   endif
end doall
```

Figure 3 shows graphically the shift of the access patterns that allows loop alignment to be achieved. The nodes of the iteration-space-dependency graph represent distinct instances of the statements inside the loop. However, the example presented above is very simple. Code replication is often necessary to remove synchronization and to obtain the alignment.

Improving the speedup of `doacross` loops In the previous example of `doacross` loop the execution of the various statements is overlapped. In some cases, however, the overlapping may be very small. As an example, consider the loop in Figure 1. The corresponding `doacross` loop is the following:

```
doacross i = 1, N
         wait(C_SYNC, i-1)
     S1: A[i+1] := B[i] * C[i-1]
         send(A_SYNC, i)
     S2: D[i] := A[i] * 2
         wait(A_SYNC, i-1)
     S3: C[i] := A[i-1] * C[i]
         send(C_SYNC, i)
         wait(C_SYNC, i-2)
     S4: E[i] := D[i] * C[i-2]
end doacross
```

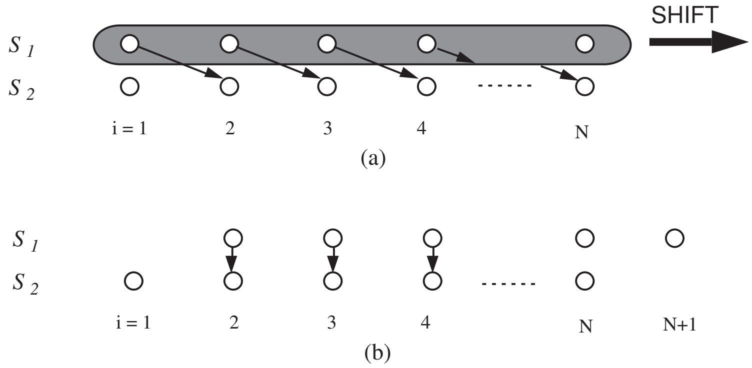


Figure 5: Iteration-space-dependency graphs of a loop to be aligned (a), and of the same loop after the alignment (b). Notice that $S1$ has not to be executed for `i=1`, while $S2$ has not to be executed for `i=N+1`.

The execution diagram of this doacross loop is the following:

P1	**P2**	**P3**	...	*time*
$S1$				0
$S2$				1
$S3$				2
$S4$	$S1$			3
	$S2$			6
	$S3$			5
	$S4$	$S1$		6
		$S2$		7

The poor speedup obtained by this concurrent implementation depends on the backward arc from $S3$ to $S1$ (see Figure 1). This dependency arc prevents a new loop iteration from being initiated by processor P_i until processor P_{i-1} has completed the execution of $S3$. Since no dependencies exist between statements $S2$ and $S3$, their execution order may be inverted. Such kind of optimization is called instruction reordering. The following loop is so obtained:

```
doacross i = 1, N
         wait(C_SYNC, i-1)
    S1:  A[i+1] := B[i] * C[i-1]
         send(A_SYNC, i)
         wait(A_SYNC, i-1)
    S3:  C[i] := A[i-1] * C[i]
         send(C_SYNC, i)
    S2:  D[i] := A[i] * 2
         wait(C_SYNC, i-2)
```

```
    S4:  E[i] := D[i] * C[i-2]
end doacross
```

The execution diagram of the modified loop points out a better speedup, due to the overlapping of more statements:

P1	**P2**	**P3**	...	*time*
$S1$				0
$S3$				1
$S2$	$S1$			2
$S4$	$S3$			3
$S2$	$S1$			4
$S4$	$S3$			5

.

Other optimizations of `doacross` loops are concerned with the elimination of covered dependencies. In fact some synchronization primitives correspond to redundant dependencies. These dependencies are induced by others, not considered when the synchronization primitives have been inserted into the code. The dependencies not considered are those derived from the serial execution of the statements inside each loop iteration. For example, if we look at the execution diagram of the `doacross` loop above, we can deduce that the only dependency requiring the insertion of synchronization primitives is that between $S3$ and $S1$. In fact, all the processors involved serially execute the statements $S1$, $S3$, $S2$, and $S4$. The only constraint to be obeyed is that a generic processor P_{i+1} must begin the execution of statement $S1$ only when processor P_i has completed the execution of $S3$. Hence the new *doacross* loop is the following:

```
doacross i = 1, N
         wait(C_SYNC, i-1)
     S1: A[i+1] := B[i] * C[i-1]
     S3: C[i] := A[i-1] * C[i]
         send(C_SYNC, i)
     S2: D[i] := A[i] * 2
     S4: E[i] := D[i] * C[i-2]
end doacross
```

whose execution diagram is the same as the previous one.

4 VLIW compiling techniques

Very Long Instruction Word (VLIW) machines are another kind of architectures for which specific parallelizing compilers have been defined. VLIW machines are able to exploit fine-grain parallelism, i.e. parallelism at the level of single instructions or micro-instructions. The VLIW compiling techniques look for such kind of parallelism not only inside inner loops, like vectorizing compilers, but also in less regular code. However, the most part of exploitable parallelism is found out inside the same kind of loops that can be also vectorized. The first VLIW compiling technique has been derived from *Trace Scheduling*, a technique for microcode compaction used to transform vertical (sequential) microcode into efficient horizontal (parallel) microcode [5]. Trace Scheduling has pointed out the existence of a large amount of parallelism in sequential programs, provided that the parallelism is searched not only within single basic blocks, but also in operations belonging to distinct blocks. In order to exploit this parallelism, a prediction analysis of the jumps involved is needed.

Another VLIW compiling technique, *Software Pipelining*, tries to exploit VLIW machine to schedule, in pipeline, the boody of loops. This tecnique avoids the unrolling of the loop, which is needed if Trace Scheduling must be applied.

4.1 Trace Scheduling

Trace Scheduling is a compiling technique first employed to compact microcode, i.e. to convert vertical microcode into efficient horizontal microcode. Other techniques previously proposed compact each basic block separately, and thus are called local compaction techniques. Exploiting local compaction and under the hypothesis of having infinite hardware resources, the speedup obtainable for the most part of sequential programs is only 2 or 3 [16]. Trace Scheduling replaces block-by-block compaction with the compaction of long streams of code. Since these streams, called traces, are composed of instructions belonging to distinct basic blocks, Trace Scheduling is referred to as a *global* technique. The compaction of a trace allows instructions belonging to distinct blocks to be executed in parallel.

The most important phases of Trace Scheduling can be summarized as follows:

i) trace selection;

ii) trace compaction;

iii) bookkeeping.

Phase i) regards an intermediate representation of a sequential program. For this purpose, a *control-flow* graph is built. Important optimizations, besides the sequential optimizations, are those regarding the reduction of data-dependencies between statements. The dependencies removed are anti and output dependencies. A *trace* is picked by selecting those blocks considered as the most likely ones to be executed. In other words, in order to build the trace, the *most probable branches* taken by conditional jumps must be predicted.

The phase ii) of the technique takes the most likely trace, generates sequential machine code for it, and compacts the code so obtained.

The phase iii) recovers the inconsistencies that the phase ii) may introduce into the code. In fact, the compaction of a trace may modify the program semantics, because of some code movements changing the execution order of assignment statements with respect to conditional jumps and/or rejoin points of the trace. In order to recover these inconsistencies and restore the original semantics, extra code has to be placed at the trace exits and entrances, outside of the trace. Because of this code, the performance may be reduced when the control leaves the most likely trace.

The three phases describe above are iterated until no traces to be compacted exist in the control flow graph. When phase i) selects a new trace, this trace has not to contain basic blocks belonging to traces previously compacted.

In the following the three phases of Trace Scheduling are discussed in detail. To simplify the presentation of the technique, and according to [5] we first introduce a more formal terminology:

- let $T = (i_1, \ldots, i_F)$ be a trace, i.e. a loop-free sequence of instructions, where every i_j is a simple statement;
- for every $i_j \in T$, such that $0 < j < F$, instruction i_{j+1} is defined as a follower of i_j;
- if an instruction $i$has more than one follower, then i is a conditional jump.
- the locations, outside of T, to which a conditional jump may go are called *trace exits*;
- if j is a follower of i, then we say that i is a leader of j;
- if an instruction $i$has more than one leader, then i corresponds to a *rejoin point* of the trace;
- jumps, outside T, that may cause the control to enter T (at a rejoin point) are called *trace entrances*.

Picking a trace In order to select the most likely trace, it is necessary to statically predict conditional jumps. The selection is usually based upon some real runs of the program over a set of suitable data samples, so that some profiling information can be collected. The method suggested in [5] is the counting of the number of times each statement is executed over some data samples. This number may be used as an expected average number of executions and be associated with the statement. The expected number may be equivalently associated with each basic block, because all the statements included in every block are executed for the same number of times. An alternative method [4, 2] is to associate an *expected value* with the directions of every conditional jumps. This corresponds to associate this value with every directed arc of the control flow graph.

In some cases it is possible to statically predict conditional jumps. As an example, if we consider the directions that may be followed by conditional jumps going either to the first instruction of a loop or outside the loop, the former is the most likely direction.

Trace Scheduling cannot be exploited successfully on code which is dominated by highly unpredictable jumps [6, 16], e.g. programs parsing loops or tree searches. Thus the programs that can be effectively parallelized remains only the scientific ones, i.e. the same programs that can be also effectively vectorized.

The jump prediction analysis [7] is the most important step of the Trace Scheduling technique. Both the selection and the following compaction of a trace are performed on the basis of this analysis.

The algorithm that picks a trace using expect values associated with every statement is sketched below:

```
procedure Pick_a_trace
          trace     := ∅
          seed      := the uncompacted statement with the
                       largest expect value
          trace     := trace  ∪  seed
          current   := seed
          while     (  in the control flow graph ∃ some
                       uncompacted leaders of current)   do
             current := the uncompacted leader of current with
                       the largest expect value
             trace :=  trace  ∪  current
          endwhile
          current   := seed
          while     (  in the control flow graph ∃ some
                       uncompacted followers of current)   do
             current := the uncompacted followers of current
                       with the largest expect value
             trace :=  trace  ∪  current
          endwhile
```

In the algorithm above, we call *uncompacted statements* those statements that do not belong to other traces previously picked and compacted.

Trace compaction Once picked a trace, the next phase of the technique, called compaction, is applied to produce a parallel schedule. This may require extra space and the lengthening of other traces. The compaction of a trace is performed by moving simple instructions either from one block to another block, or within the same block. These movements produce the packing of several simple instructions into very long instructions (VLI), and convert the sequential code into parallel code.

The technique uses a data-dependency directed acyclic graph (DAG) associated with the trace. The DAG points out the maximum parallelism degree that can be exploited in the trace.

The compacting technique adopted in [5] is based upon *List Scheduling*, a heuristic technique used to compact single basic blocks. In fact, the trace must be compacted as a whole, without considering that it is composed of instructions belonging to several blocks. However, any other methods that work well for basic block compaction may be used to compact the trace.

List Scheduling gives a solution to the discrete processor scheduling problem [8], i.e. the problem of assigning tasks (in our case, simple instructions) to time cycles (in our case, VLIs) belonging to distinct processors. Such an assignment must guarantee that

i) data dependencies are obeyed;

ii) no more than the number of processors available per each cycle is used. An order of the tasks is necessary when a task to be assigned has to be chosen among several ones. This may occur if all these tasks depend on others, all assigned to preceding time cycles.

With respect to a specific VLIW machine, including P functional units, List Scheduling technique tries to insert at most P simple instructions in every VLI. The insertions are performed by scanning the set of data ready instructions, i.e. the instructions whose predecessors in the DAG have been previously inserted. The scanning ordering, needed to apply List Scheduling, is given by the expected values associated with each instruction.

A simple instruction is scheduled inside a VLI only if it is resource-compatible with the other instructions in the VLI. Two examples of resource incompatibility are the following:

i) if two instructions refer to the same memory bank, then they cannot be placed in the same VLI;

ii) if two instructions (for instance two floating-point multiplications) need the same functional unit to be executed, then they cannot be placed in the same cycle.

Determining whether two instructions refer to the same memory bank, or not, may be difficult in case those instructions use or define two elements of the same array. For this reason, VLIW compilers exploit memory disambiguation techniques, and the speedup obtained heavily depends on the effectiveness of such techniques.

List Scheduling applied to a trace, may produce implicit inter-block code motions, but not all these motions are correct. In order to avoid inconsistent

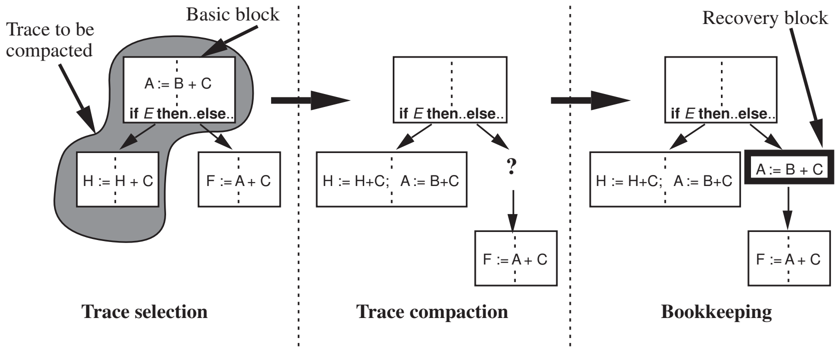


Figure 6: Example of an operation, `A := B+C`, which was originally above a conditional jump, and that has been moved below the jump itself, to be executed in parallel with `H := H+C`. Notice that the same operation has been copied (*recovery code*) outside the trace, at the exit point. The copy is necessary as variable `A` is live at the exit point.

motions, the DAG is augmented with other dependency arcs obtained from information about *live* variables[1]. Given a trace T, suppose that:

- i_j is a conditional jump in T, and
- there exists a variable v,*live* at the exit point (outside T) corresponding to jump i_j.

In this case, v is considered as a variable used by i_j, so that extra arcs are introduced into the DAG, from i_j to later instructions (inside T) re-defining v. These arcs prevent variables, which might be used outside T, from being re-defined by an instruction moved above i_j.

Bookkeeping The last phase of Trace Scheduling is called *bookkeeping*, and re-builds a consistent program. The instructions are duplicated (where necessary) in order to make the code semantically equivalent to the original one. The inconsistencies making the code not semantically equivalent to the original one occur in case the control, at run-time, leaves the trace previously compacted. The instructions that bookkeeping duplicates and inserts outside the trace are called *compensation* or *patch* code. However, we call them recovery code, to make an analogy with fault tolerance systems, in which some code is prepared to be execute in case a fault occurs. Bookkeeping looks at conditional jumps and rejoin points of the compacted trace to discover inconsistencies to be recovered.

[1]A variable is *live* if its value may be used by some instruction that follows in the sequntial control flow of the program.

Figure 7: Example of code motion allowed by the renaming of a variable. If variable `A` is renamed, the augmenting arc in the DAG can be removed. This arc prevents the instruction `A := P + K` from being moved above the conditional jump.

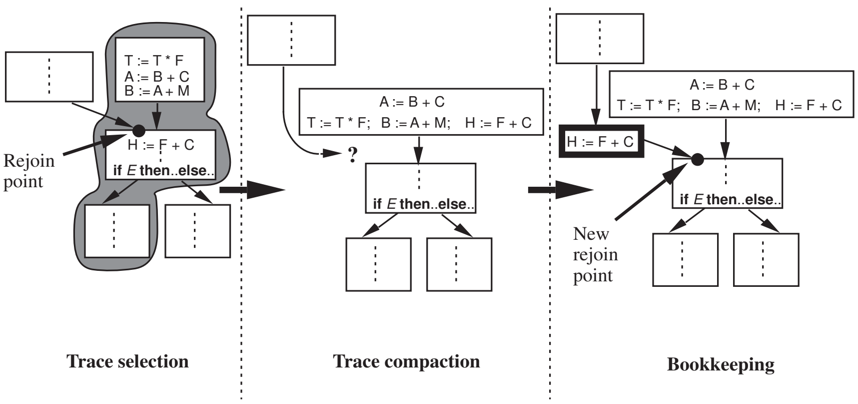


Figure 8: Example of trace compaction that requires the finding of a new rejoin point in the trace, and the insertion of a recovery block to be executed before this new point.

As regards conditional jumps, inconsistencies may arise when an instruction, originally placed above a jump, is moved below the jump itself. This code motion does not cause any inconsistencies if the variable defined by the instruction is *dead* outside of the trace. If the variable is instead *live* at the exit point corresponding to the jump, the instruction has to be also copied to the place jumped to. This generates a new block, that has to be executed before the original block to which the control jumps in case the trace is left. Figure 4.1 shows an example of this kind of recovery technique. Figure 4.1 shows another example of code motion. An instruction originally scheduled below a conditional jump is moved above the jump itself. Code motion may take place after the renaming of a variable. This allows the removing of an augmenting arc that prevents the code motion from being performed.

As regards rejoin points, once the trace has been compacted, the new rejoin points of the trace must be determined. This may be difficult because in some cases it is not possible to find out a point such that the resulting code is semantically equivalent to the original one. Hence recovery code, to be executed before the new rejoin point, may need to be inserted. Specifically, bookkeeping looks for the highest point such that no instructions that are at or below this point had been above the original one. All the instructions that

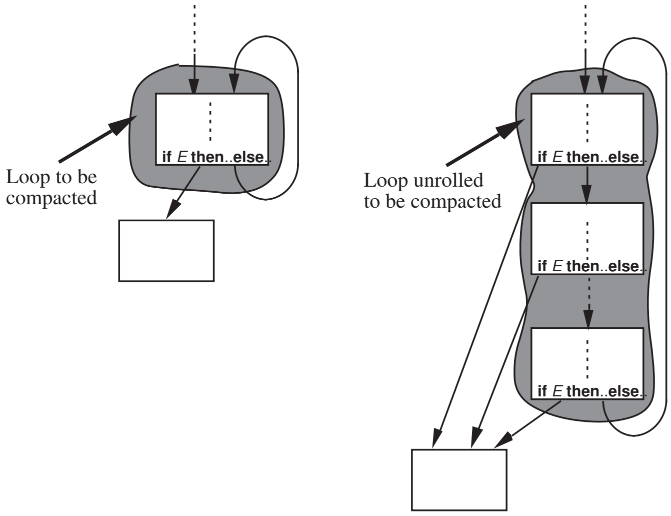


Figure 9: Example of loop unrolling.

are above the new point and had been below the old one, are also copied in a ne/di/discA/homescA/home/discA/homew block, to be executed before the new rejoin point. Figure 4.1 shows an example of this kind of recovery technique.

Code containing loops The loop parallelization, besides for vector machines, is also important for VLIW machines. Though the parallelism extracted from a large number of programs by exploiting VLIW machines and Trace Scheduling is larger than the parallelism obtainable exploiting vector machines and vectorizing compilers, Trace Scheduling obtains the largest speedups from programs containing many vectorizable loops.

Trace Scheduling is separately applied to the loop bodies. After the optimization of a loop, this loop is considered as a simple instruction belonging to a trace to be optimized. When a trace to be compacted contains a loop previously optimized, Trace Scheduling tries to move instructions, which are loop invariant, into the loop. This may occur when such instructions can be executed in parallel together with the instructions belonging to the loop body. Notice that this is the reversal of the transformations usually made by conventional optimizing compilers for sequential machines.

Since Trace Scheduling is separately applied to the body of each loop, the trace of a loop to be optimized includes only instructions belonging to the loop body. To get enough parallelism in the trace, the loop may be unrolled for several times. The unrolling yields a normal trace to be compacted, where all intermediate loop tests are considered simple conditional jumps. Figure 4.1

shows a graphical representation of loop unrolling in terms of the corresponding control flow graphs.

Available VLIW parallelism using Trace Scheduling In [16] some measures of available parallelism for an idealized VLIW machine are reported. The machine is idealized because the measures are based upon the hypothesis of infinite resources availability. Furthermore the existence of an oracle is assumed: the oracle is able to indicate which branch will be chosen by each jump, and is able to resolve all ambiguous memory references (like indirect memory references). For these measures, antidependencies and output dependencies are supposed to be removed by introducing new variables. All programs considered are numerical applications. According to [16], the reasons for this choice can be summarized as follows:

- the CPU intensive peculiarity of the applications, so that a large speedup is felt as a very important need;
- the relatively large life spans of the applications, so that the adoption of expensive compaction techniques are justified;
- the application behaviour, which is fairly independent of the specific input data, so that conditional jumps may be successfully predicted at compile time.
- the presence of array references with linear relation between indexes, so that memory reference disambiguation turns out to be very easy.

The most important measures are the following:

- the speedup ratio, between the number of cycles for sequential execution and the number of cycles for parallel execution (after compaction via Trace Scheduling);
- the average speedup per basic block; in this case the ratio is computed by performing only local compaction, at the level of each single basic block.

It is interesting to compare these measures with the following ones, also reported by [16]:

- the number of added variables, i.e. the variables introduced in order to remove anti and output dependencies;
- the maximal level width, i.e. the number of computational resources used by the most resource intensive VLI of each program.

Another measure that we have introduced is the efficiency, i.e. the ratio between the performance obtained for a specific application and the peak performance offered by the idealized VLIW machine, able to execute the most resource intensive VLI of the program. Thus the efficiency is defined by:

$$\frac{speedup\ ratio}{maximal\ level\ width}$$

From the point of view of the measures reported, the most interesting programs considered are:

GRAM:	Computes the stabilized Gram-Schmidt orthogonalization.
BISECT:	Computes a root of a polynomial using the bisection method.
RFALS:	Computes roots of scalar functions using the Regula Falasi method, with check.
EXTRAP:	Evaluates the derivative of a given function at a given point using extrapolation.

The specified measures for each program are reported in the following table:

	GRAM	BISECT	RFALS	EXTRAP
SPEEDUP RATIO	988.5	304.03	4.10	66.39
MAXIMAL LEVEL WIDTH	3234	2669	30	198
AVERAGE SPEEDUP PER BASIC BLOCK	1.96	3.23	1.63	1.90
NUMBER OF ADDED VARIABLES	33955	13519	735	1501
EFFICIENCY	0.3	0.11	0.14	0.33

The most important result is the increase in speedup using Trace Scheduling, with respect to the obtained speedup when the compaction only regards basic blocks (local compaction). The speedup range from 2 to 3 in case of local compaction, and from 4 to 988 when Trace Scheduling global compaction is performed. The programs operating on arrays exhibit the largest speedup, even though they need sophisticated memory disambiguation techniques.

All these measures represent, however, an upper bound on the available parallelism. In fact, the considered VLIW machine is unrealistic, since it is supposed to have infinite computational resources. Furthermore the oracle is also idealized, because it is always able to predict the directions taken by conditional jumps, as well as the possible collision of two memory references.

4.2 Pipelined execution of loops

The speedup of the execution of inner loops is an important parameter which heavily influences the overall speedup of a program. If distinct iterations of a loop are independent, then they can be executed in parallel. But if every iteration n depends on preceding iterations, i.e. there exist some loop-carried dependencies, then each iteration may be executed in pipelined fashion to achieve speedup. To obtain a pipelined execution, each instruction of the loop body has to be executed by a distinct computational unit. In such a way, each instruction becomes a step of the pipeline. As an example, if a a loop is composed of three instruction:

```
do i=1,N
   S1
   S2
   S3
enddo
```

and can be pipelined, the following VLIW piece of code is obtained, where each box contains a VLI:

	S1			
	S1	S2		
$L:$	S1	S2	S3	$CJump\ L$
		S2	S3	
			S3	

Note that the operation $CJump\ L$ belonging to the instruction labelled by L continues to branch to L until the last iteration of the original loop is initiated. In this way a new iteration can be initiated every cycle, where a cycle is the time to execute a single step of the pipeline. The technique used to obtain this pipelined execution of loops is called *Software Pipelining* [13]. Not all loops can be pipelined in such a way, but the following condition has to hold:

> all arcs of the dependency graph associated with the loop are to be *forward* with respect to the lexical ordering of instructions, i.e. the ordering of the steps of the pipeline.

If the $n+1-th$ iteration of a loop has some dependencies on the $n-th$ one, Software Pipelining Scheduling is claimed to be better than Trace Scheduling with loop unrolling. Moreover, in case a loop is unrolled k times, it is possible that these k iterations result pipelined after the transformations performed by Trace Scheduling (the same as applying Software Pipeling). Unfortunately, the initial delay (due to the filling of the pipeline) and the final delay (due to the emptying of the pipeline) have to be paied for every block of k iterations.

5 Synthesis of systolic algorithms

The algorithms that can be implemented on systolic arrays [11] are only those belonging to a specific class. They are called recursive and locally dependent algorithms. The reason of such name will become clear later. If a sequential algorithm belongs to this class, it may be easily partitioned into a collection of independent processes to be executed in parallel, or may be decomposed into a set of dependent processes to be executed in pipeline.

Systolic algorithms can be synthesized starting from sequential formulations of the same algorithm [18, 12], such as multi- nested loops or recursive equations. As an example, let us consider the following program that computes a matrix-vector multiplication, `c=Ab`, where `c` and `b` are $4-vectors$ and `A` is a $4 \times 4-matrix$.

```
do i= 1, 4
   do j = 1, 4
      S:  c[i]  :=  c[i] + A[i,j] * b[j]
   enddo
enddo
```

The nested loop above cannot be transformed into a systolic algorithm, because between the various instances of the statement S there exist not only flow dependencies, but also anti and output dependencies. Antidependencies and output dependencies may be removed introducing further variables. A method called scalar expansion [17] may be used in this case. The program is so transformed into single assignment code, in which only a single value is assigned to every variable. The new program is:

```
do i= 1, 4
   do j = 1, 4
      c[i,j+1]  :=  c[i,j] + A[i,j] * b[j]
  enddo
enddo
```

The transformation above derives from the necessity of generating a program for a distributed memory machine. In fact, if only flow dependencies exist between the operations:

- these operations may be assigned to distinct processing nodes (PNs);
- the arcs representing flow dependencies in the corresponding graph may be transformed into communications between the PNs.

Iteration-space-dependency graphs (see Section 1) are used to represent flow dependencies of single assignment programs. Figure 5.(a) shows the graph corresponding to the loop program above. Each computation node in the graph

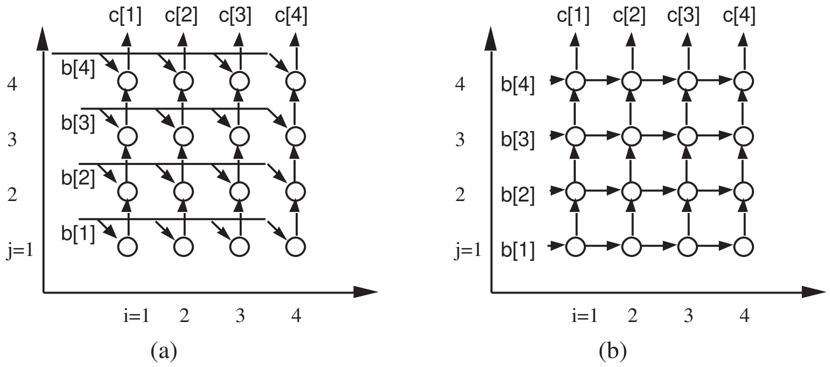


Figure 10: DGs for matrix-vector multiplication, (a) with broadcast data, (b) with transmitted data.

represents an iteration of the loop body. Furthermore, also the variables on which each computation depends and the corresponding flow dependency arcs are shown. Variables and nodes are placed in the corresponding points of the index space (indexes i and j). In this section we refer to this graph as Dependency Graph (DG).

The transformation of the program into a single assignment code does not suffice to obtain a systolic algorithm from the graph. In fact the program does not yet correspond to a locally recursive algorithm. The definition of such kind of algorithms may be given in terms of the DG associated with the corresponding program.

Definition 2 *Locally recursive algorithm, expressed in single assignment code, are associated with DGs that are characterized by:*

i) bounded degree*: each node depends on a bounded number of other nodes or variables, independently of the size of the specific problem;*

ii) local dependency*: the length of each arc is bounded and does not depend on the problem size. In other words, the arcs connect neighbouring nodes, and no data is broadcast to the nodes characterized by certain indexes;*

iii) the same kind of operations*: most nodes of the DG are identical operations.*

The DG in Figure 5.(a) does not belong to the class of graphs corresponding to locally recursive algorithms. In fact the value of `b[i]` should be broadcast to all the nodes having the same index j.

In order to transform the algorithm into a localized one, the broadcast data, `b[i]`, must be transmitted without being modified between the nodes having the same index j. When some data are propagated without any modifications, that are called transmitted data. With respect to the loop program, in order to transform `b[i]` into transmitted data, further variables must be introduced. Let `B` be a new $4 \times 4 - matrix$, such that, for every $j = 1, \ldots, 4$ `B[1,j] = b[j]`. The new program is shown below:

```
do i= 1, 4
   do j = 1, 4
      B[i+1,j] := B[i,j]
      c[i,j+1]  :=  c[i,j] + A[i,j] * B[i,j]
   enddo
enddo
```

The corresponding DG is shown in Figure 5.(b). In this algorithm, the statement that assigns `B[i+1,j]` is flow dependent on the statement that assigns `B[i,j]`. Due to this new dependency, each $(i+1, j) - node$, which represents the loop iteration $(i+1, j)$, depends also on the $(i, j) - node$.

Recursive equations Systolic algorithms are said to belong to the class of locally recursive algorithms because they may be expressed using recursive equations. For instance, the equation corresponding to the matrix-vector multiplication `c = Ab` is defined by:

$$\begin{aligned} c_i^{(1)} &= 0 \\ c_i^{(j+1)} &= c_i^{(j)} + a_i^{(j)} b_i^{(j)} \end{aligned}$$

where the index j is the recursive index, $j = 1, 2, \ldots$, and $a_i^{(j)} = \mathtt{A}[\mathtt{i}, \mathtt{j}]$ and $b_i^{(j)} = \mathtt{b}[\mathtt{j}]$.

The indexes i and j may be used one for time and the other for space. The recursive equations are always given in a single assignment formulation, because a distinct index corresponds to each assignment.

Signal flow graphs The *Signal Flow Graphs* (SFGs) are an abstraction and a graphical representation of locally recursive algorithms. They may be considered as an intermediate expression between the DG of an algorithm and its hardware realization. With respect to the corresponding DG, they consist of nodes that represent computations, arcs denoting dependencies, and delays (D, $2D$, etc.) that may be associated with arcs. Nodes and arcs are abstraction of processors and physical links, respectively. Computations are assumed to occur without producing delays. This assumption simplifies the expression of systolic algorithm using SFGs, because the temporization of the various activities is only described in terms of the delays labelling arcs.

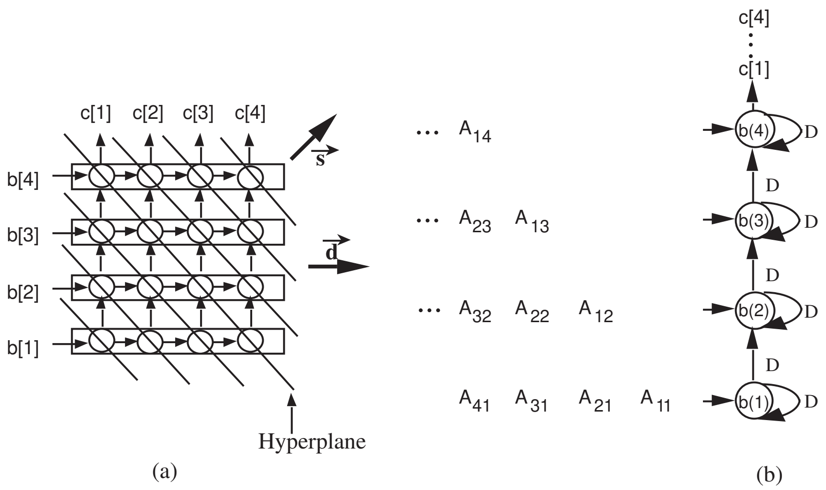


Figure 11: Vectors determining projection and scheduling for matrix-vector multiplication (a), and its corresponding systolic SFG (b).

Mapping DG to SFG Given a DG corresponding to a locally recursive algorithm, in order to design the systolic array that implements the algorithm, a mapping of the DG to an SFG has to be produced. Notice that a DG such as that in Figure 5.(a) can be mapped in a straightforward way by assigning a processing element to each node in the DG. This is possible provided that there exists a physical link for every dependency arc in the DG. However, this mapping is not efficient, because each processing element remains active for a small fraction of the overall computation time.

Since the mapping of DG to SFG has to improve the efficiency on the implementation, the number of the nodes in an SFG must be less than the number of the nodes in the corresponding DG.

The phases of the mapping may be summarized as follows:

i) *processor assignment*: the nodes of the SFG (processors) to which the nodes of the DG are assigned are determined;

ii) *scheduling*: the execution order of the computations (nodes of the DG) assigned to every processor is specified.

The mapping we describe is termed linear. Its main characteristic is the regularity.

Figure 5.(a) shows the DG of the matrix-vector multiplication, while Figure 5.(b) shows the corresponding SFG generated by the mapping.

As regards the processor assignment, all the DG nodes along a straight line, called projection direction, are assigned to the same SFG node. This kind of assignment is called linear because the projection direction can be formally specified by a vector $\vec{d}$. Figure 5.(a) shows several boxes that contain all the DG nodes projected onto the same SFG node.

As regards the scheduling of computations, it may be thought of as a function that assigns the DG nodes, characterized by a two-dimension space (indexes i and j), to a one-dimensional space, i.e. the temporal space.To perform this, it is necessary to determine some parallel *hyperplanes*, such that all the nodes on the same hyperplane can be executed at the same time. Such hyperplanes may be formally represented by a vector $\vec{s}$ perpendicular to the direction of the hyperplanes (see Figure 5.(a)).

The SFG shown in Figure 5.(b) is called systolic, and is characterized by at least a delay D associated with every arc. The data needed for computation are shown on the left side of the SFG and inside each node of the SFG. Note the temporization of the arrivals of the elements of matrix **A**, i.e a delay D elapses between the arrival of A_{11} and the arrival of both A_{21} and A_{12}.

Algebraic verification of mapping The processor assignment and the scheduling of the computation may be verified in terms of algebraic transformations. Given a projection direction characterized by a vector $\vec{d}$, and a possible linear schedule characterized by a vector $\vec{s}$, the following conditions must be satisfied:

$\vec{s}T\vec{e} > 0$	for every dependency arc in the DG. The direction of the arc is given by vector $\vec{e}$.	(3)
$\vec{s}T\vec{d} > 0$		(4)

Condition (3) guarantees that:

- the temporal ordering specified by $\vec{s}$ for two DG nodes is not opposed to the ordering specified by the dependency arc connecting them;
- DG nodes that are connected by a dependency arc are not placed on the same hyperplane.

Condition (4) prevents DG nodes, placed on the same hyperplane, from being projected on the same SFG node.

As regards the DG in Figure 5, we have first found the linear schedule characterized by:

$$\vec{s} = [1\ 1]^T$$

This vector represents a feasible schedule, because condition (3) is verified. In fact, $\vec{s}$ forms an acute angle together with all vectors associated with the directed dependency arcs in the DG.

Next, we have found the projection vector:

$$\vec{d} = [1\ 0]^T$$

This vector represents a feasible projection, because condition (4) is verified. In fact, this vector forms an acute angle together with vector $\vec{s}$.

6 Parallelizing for multicomputers

In this sections we describe some techniques used to compile loops on distributed-memory (DM) multiprocessors (*multicomputer*) [20]. Much research activity is currently concentrated on this topic, since only these machines are able to exploit large scale parallelism. One focus of the research is concerned with the *language constructs*, that should allow programmers to express parallelism within normal sequential programs. All the proposals concern different flavors of the `doall`, whose syntax and semantics have been presented above.

Thus, the user can write programs using global data references, but, in many cases, he is requested to specify a distribution of these data on an abstract MIMD architecture. The data distribution policy is intended to be used by a compiler, which allocates the iterations of the `doall` according to the data distribution, transforming global references in local and non-local ones. In particular, the hardest task is to transform the non-local references in a set of communications with the owners of the data to be referenced.

6.1 Languages constructs

There exists several proposals of parallel loop constructs to be inserted into a sequential language [3]. Usually, for many reasons, the sequential language considered is Fortran. In the following, we present two of the main extension to Fortran, Fortran D and HPF.

Fortran D In Fortran D [10], the parallelizing techniques are to be applied to sequential loops, in which programmers introduce directives to impose a given data distribution. However, a parallelizing compiler is needed to determine if the sequential loop can be transformed into a `doall` one. Hence, either programmers or clever compilers have to remove loop-carried dependencies.

HPF HPF is a proposal of a language [9] that includes several features of FortranD. The main difference in concerned with the introduction of an explicit `FORALL` construct. In this loop all the assignments are performed in parallel as a two step process. In the former step, all the right sides of the assignments are computed. In the latter, all the assignments take place. On the other hand, the `doall` maintains the usual sequential semantics. To write a program with the same semantics as that of `FORALL`, we need a pair of `doall` loops. The former should assign the left side of each assignment to some *new* arrays. The latter should simply assign the *new* arrays to the original ones.

Data distribution In all the proposals of compiling techniques for distributed memory parallel machine, one of the main problem is to distribute data in an "optimal" way, and, at the same time, to map the parallel computations in order to minimize the need of communications. This job is very difficult, and, in general, it is not possible to devise an optimal solution for each possible program. Many factors may influence the choice of the data distribution. In many cases, for example, the minimization of the communication overhead may involve the reduction of the parallelism exploited. Thus a compromise is needed. Moreover, in case the references that a loop makes to a distributed data structure are not predictable statically, an optimal solution to the data distribution problem may become very hard.

Taking into account all these problems, along with the high cost of communications, the current compiling techniques are based on *static* data distribution, for which the programmer can supply some directives. In particular, given a virtual grid of processors, the programmer may specify whether a given array is to be distributed *blocking* or *cyclic* with respect to its various dimensions. Moreover, he may give directives to align arrays with each other. All these directives should enable a programmer to specify data allocation policies that entail, as much as possible, *locality* of references.

Mapping of computation The computation mapping, i.e the allocation of the various iterations of the parallel loops, is usually guided by the data distribution. Since the data are usually distributed statically by the compiler on the basis of the user directives, also computations are mapped statically. A rule that is frequently adopted by compilers to map computations is called *owner-computes*. The processor to which a given data partition has been assigned is considered the "owner" of the partition. Each assignment statement occurring within a `doall` is computed by the processor that owns the data item appearing in the left part of the statement.

6.2 An example

In this section a simple example is introduced, written before in sequential Fortran, then in Fortran D and in HPF.

This is the sequential version:

```
REAL A(1000,1000), B(1000,1000), C(1000, 1000)
DO k = 1,1000
   DO i = 1,999
      DO j = 1,1000
```

```
            C(i,j) = (A(i,j) + A(i+1,j)) * B(i+1,j))
         END DO
      END DO

      DO i = 1,999
         DO j = 1,1000
            A(i,j) = C(i,j)
         END DO
      END DO
   END DO
```

The following is the Fortran D version, where we have introduced three new directives: **DECOMPOSITION**, **ALIGN**, and **DISTRIBUTE**. The first one is related to a virtual topology on which data must be aligned. The second one specifies the alignment (mapping) of the data structures to the virtual grid. This task is also called *problem mapping*. Finally, the last directive determines the policy to adopt to distribute the virtual topology (on which the data structures have been aligned) onto the actual processor topology. This task is also called *machine mapping*. For each dimension of the virtual topology, a programmer can specify whether that dimension must be distributed or not, and how must be distributed.

```
   REAL A(1000,1000), B(1000,1000), C(1000, 1000)

   DECOMPOSITION D(1000,1000)
   ALIGN A(i,j), B(i+1,j), C(i,j) with D
   DISTRIBUTE(BLOCK,BLOCK)

   DO k = 1,1000
      DO i = 1,999
         DO j = 1,1000
            C(i,j) = (A(i,j) + A(i+1,j)) * B(i+1,j))
         END DO
      END DO

      DO i = 1,999
         DO j = 1,1000
            A(i,j) = C(i,j)
         END DO
      END DO
   END DO
```

In particular, in the previous Fortran D example, we have introduced a virtual bidimensional topology `D`. Then we have specified a problem mapping that exactly aligns both the arrays `A` and `C` with the topology `D`, while `B` is shifted of one position on its first dimension before being aligned with `D`. This kind of problem mapping should avoid non-local references within the body of the first loop, since `A(i,j)` and `B(i+1,j)` are both allocated to the same virtual processor. Finally, we have suggested a machine mapping where `D`, and thus all the arrays mapped onto it, should be mapped `BLOCKED` on both the dimensions. Thus, if `D` has to be mapped to four processors $P_1, \ldots, P_4$, then we should obtain the following mapping: `D(1:500,1:500)` on P_1, `D(1:500,501:1000)` on P_2, `D(501:1000,1:500)` on P_3, and `D(501:1000,501:1000)` on P_4.

With these directives a programmer has suggested that the instructions which involve arrays `A`, `B`, and `C` should be executed in parallel, but the compiler has to check the independence of the various iterations in the following `do` loops before parallelizing.

Finally, we show the same example written in HPF. One of the main difference regards the use of the directive `TEMPLATE` in place of `DECOMPOSITION`, while all the directives are preceded by the keyword `!HPF$`. Moreover, since we can use the HPF `FORALL` construct, we can avoid using the array `C`, that in the Fortran D example has been used only as an intermediate place to put the results of the first loop.

```
REAL A(1000,1000), B(1000,1000)

!HPF$ TEMPLATE D(1000,1000)
!HPF$ ALIGN A(i,j), B(i+1,j) WITH D
!HPF$ DISTRIBUTE(BLOCK,BLOCK)

DO k = 1,1000
   FORALL (i=1,999, j=1,1000)
         A(i,j) = (A(i,j) + A(i+1,j)) * B(i+1,j))
   END FORALL
END DO
```

6.3 Basic compiling techniques

In this section we illustrate how the example above can be parallelized and loaded to a distributed memory multiprocessor. For this purpose we consider an abstract DM-MIMD machine that supplies a sequential language for programming each node with some statements for explicit message passing.

Suppose that the starting program is the Fortran D version, and that the actual process topology is a bidimensional mesh of processors. Thus, according to the `DISTRIBUTE` directive, the virtual grid `D` will be distributed by blocking. For example, supposing that the mesh comprises the processors $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, and $P_{2,2}$, we should have

D(1 : 500, 1 : 500)	*mapped to*	$P_{1,1}$
D(1 : 500, 501 : 1000)	*mapped to*	$P_{1,2}$
D(501 : 1000, 1 : 500)	*mapped to*	$P_{2,1}$
D(501 : 1000, 501 : 1000)	*mapped to*	$P_{2,2}$

Now the main task of the compiler is map the loop to be parallelized so that each processor only executes disjoint subsets of all the loop iterations, while each partition must be allocated to the local memory of each processor. Moreover, other local memory structures must be allocated to host data items accessed through *non-local* references, i.e. data items that do not belong to the local data partition assigned to each processor. Finally, all the global data references must be transformed into local ones, where non-local references must be preceded by communications to load local data structures.

The optimized code of a generic processor $P_{s,t}$, whose partition corresponds to the data structures aligned with `D(x : x+LEN-1, y : y+LEN-1)`, should be the following

```
REAL A(1:LEN+1,1:LEN), B(1:LEN,1:LEN), C(1:LEN,1:LEN),

SEND(P(s-1,t), A(1:1,1:LEN))
RECEIVE(P(s+1,t), A(LEN+1:LEN+1,1:LEN))

DO k = 1,1000
   DO i = 1,LEN
      DO j = 1,LEN
         C(i,j) = (A(i,j) + A(i+1,j)) * B(i,j))
      END DO
   END DO

   DO i = 1,LEN
      DO j = 1,LEN
         A(i,j) = C(i,j)
      END DO
   END DO
END DO
```

Note that all the array references are made to local copies of the various data structures. These structures are supposed to be loaded with the partitions assigned to the corresponding processor $P_{s,t}$. The array `A` is aligned with `B` shifted of one position on its first dimension. Hence, even if the code in the loop body now refers to `B(i,j)` instead of `B(i+1,j)`, the semantics of the original loop is preserved. Moreover, a pair of `SEND/RECEIVE` statements have been added. This brings the column `A(LEN+1:LEN+1,1:LEN)` into the local memory of each processor $P_{s,t}$. This makes the references to `A(i+1,j)` local. Note that the size of the local array `A` permits the column received through the pair of

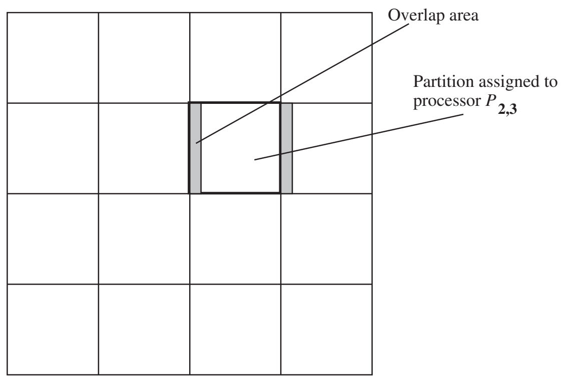


Figure 12: Partition associated with a generic processor of a 4×4 mesh and *overlap area*.

send/receive statements to be stored locally. This extra space surrounding the original local data partition `A((1:LEN,1:LEN)` is known as *overlap area*, and is illustrated in Figure 6.3.

The program illustrated above can be associated with most of the processors of the actual mesh. However, in order to have a single program that can be associated with all the processors, we need to specialize that code. For example, in the program associated with those processors to which the borders of the virtual mesh `D` have been mapped, some communications do not have to occur. The following is the program that takes into account the indexes of the associated generic processor $P_{s,t}$, belonging to a $N \times N$ processor mesh:

```
REAL A(1:LEN+1,1:LEN), B(1:LEN,1:LEN), C(1:LEN,1:LEN),

IF (s .GT. 1) THEN SEND(P(s-1,t), A(1:1,1:LEN))
IF (s .LT. N) THEN RECEIVE(P(s+1,t), A(LEN+1:LEN+1,1:LEN))

IF (s = N) THEN LEN1=LEN-1 ELSE LEN1=LEN

DO k = 1,1000
   DO i = 1,LEN1
      DO j = 1,LEN
         C(i,j) = (A(i,j) + A(i+1,j)) * B(i,j))
      END DO
   END DO

   DO i = 1,LEN
      DO j = 1,LEN
         A(i,j) = C(i,j)
```

```
        END DO
      END DO
    END DO
```

Note that, if $s = 1$, then the *send* statement is not executed, since processor $P_{s-1,t}$ does not exist. For the same reason, the *receive* statement does not occur when $s = N$. Finally, since the original loop runs for i that goes from `1` to `999`, the code associated with the processors corresponding to the borders of `D`, namely $P_{N,t}$, has to run for i that goes from `1` to `LEN-1`. The third `IF` statement just guarantees this behaviour.

The style of this program is the so-called *Single Program Multiple Data (SPMD)*. A single program is, in fact, allocated to all the processors, and each copy of the program refers to distinct arrays, stored in multiple private address spaces. This is the main difference with respect to a *partitioned shared-memory program* to be executed on a shared-memory multiprocessor. In the latter case, in fact, while the various loop iterations are distributed according to a given layout strategy, the data are kept shared. Hence, the various processors continue referring to the same shared address space, using the same indexes as in the original sequential program.

Synchronization issues The semantics of the original program requires that the various iterations of the outermost loop are executed sequentially. Since, in the parallel version, the body of the outermost, sequential, loop is a parallel one, then it is needed that all the parallel activations are terminated before beginning the next iteration of the outermost loop. In the parallel SPMD version, however, we have not introduced special synchronization primitives. In fact, we rely on the blocking features of the asynchronous send/receive primitives, and on the correct delivering order that is guaranteed by the communication subsystem. If the same parallel program is to be executed on a shared memory multiprocessor, i.e. as a *partitioned shared-memory program*, then a special *barrier synchronization* primitive should be executed by all the processors before beginning the next iteration of the outermost loop.

6.4 Advanced compiling techniques

In the previous section we have introduced the basic techniques currently used to compile on a DM multiprocessors by using message passing mechanisms. The techniques to adopt has thus to "solve" data-dependencies among computations statically mapped on a given target machine. In fact, at the end of each parallel iteration, a *synchronization/communication phase* has to take place. During this phase some communication statements exchange data to be read by each processor before beginning the next parallel iteration.

Unfortunately, since we are parallelizing shared-memory loops, the references to the distributed data structures may be, in principle, whatever. In contrast, the regularity of the references, if predictable at compile time, might be used to optimize the corresponding communication/synchronization phase. The recognition of reference patterns characterized by a specific *structure* is the main goal of the work of M. Chen and J. Li [14].

In this work they defines a set of *basic communication routines* to be provided by an abstract machine. These routines are exactly those to be used to realize the synchronization/communication phases in the target parallel program implementing a given loop. The motivation of this choice derives from the ability of tuning the performance of each routine for every specific target machine. In order to use these routines a compiler has to recognize that some references between processors "match" a given communication routine. Hence, the authors introduce unique symbolic representation of both *reference patterns* and *communication routines*, and define an algorithm that select communication routines by *pattern matching*.

We show the application of the technique to the example above, and in particular, we refer to a HPF-like version:

```
REAL A(1000,1000), B(1000,1000)
dom D = [1...999][1...1000] /* spatial domain */
dom T = [1...1000] /* temporal domain */
DO (k : T)
   FORALL ((i,j) : D)
      A(i,j) = (A(i,j) + A(i+1,j)) * B(i+1,j))
   END FORALL
END DO
```

The domain over which the iterations of the bidimensional FORALL loop range is denoted by $D = [1 \ldots 999] \times [1 \ldots 1000]$. The domain space over which the outermost sequential DO loop is denoted by T.

Reference patterns The reference patterns are concerned with which data items are needed to perform the assignments of some other data items within an n-dimensional FORALL loop. In other words, they represent a collections of *data dependencies*. For each assignment inside the loop body, e.g.

$$\mathtt{a}(i_1, i_2, \ldots, i_n) = \mathtt{b}(\tau_1, \tau_2, \ldots, \tau_n) \tag{5}$$

a quoted symbolic form is introduced:

$$\ulcorner a(i_1, i_2, \ldots, i_n) \leftarrow b(\tau_1, \tau_2, \ldots, \tau_n) : D \mid \gamma \urcorner$$

where γ if the *guard* of the reference pattern. γ specifies the condition under which the reference pattern is valid. This is necessary, for example, when the assignemnt (5) is guarded by an `IF (`γ`) THEN ...` construct.

Hence, the reference patterns to be associated with the assignment of our loop are

$$\begin{array}{lll} P_1 & : & \ulcorner A(k,i,j) \leftarrow A(k,h,j) : D \mid i \leq h \leq (i+1) \urcorner \\ P_2 & : & \ulcorner A(k,i,j) \leftarrow B(k,i+1,j) : D \mid True \urcorner \end{array}$$

where h is a specific coordinate of the temporal index domain T.

If we only consider the spatial index domain (D) to describe the reference patterns, the following new denotation is introduced, called *spatial reference pattern*:

$$\begin{array}{lll} P'_1 & : & \ulcorner A@(h,j) \Rightarrow (i,j) : D \mid i \leq h \leq (i+1) \urcorner \\ P'_2 & : & \ulcorner B@(i+1,j) \Rightarrow (i,j) : D \mid True \urcorner \end{array}$$

Note that while in the reference pattern notation we use $b(x) \leftarrow a(y)$ to denote that $a(y)$ is needed to compute $b(x)$, in the spatial reference pattern notation we use $a@y' \Rightarrow x'$, where x' and y' contain only the spatial coordinates, to denote that the element of the array a in the spatial coordinates x' must be sent to spatial coordinates y'. The motivation of this new notation is related to next pass of the technique, which distributes data and iterations of the the **FORALL** loop, which just ranges over the spatial index domain D.

Communication patterns Interprocessor communications, and relative communication routines, can be represented by a notation, similar to the *spatial reference patterns*, called *communication patterns*. In particular, given an abstract machine denoted by $M = [1 \ldots N_1] \times \ldots \times [1 \ldots N_n]$, and a tuple $(i_1, \ldots, i_n)$ ranging over D, the following is the denotation of a *communication pattern*

$$\ulcorner a@(\sigma_1, \ldots, \sigma_n) \Rightarrow (\delta_1, \ldots, \delta_n) : M \mid \gamma \urcorner \tag{6}$$

where both $(\sigma_1, \ldots, \sigma_n)$ and $(\delta_1, \ldots, \delta_n)$ are expression of $(i_1, \ldots, i_n)$, γ is a Boolean predicate over $(i_1, \ldots, i_n)$, and a denotes an array data structure . The meaning of (6) is similar to the *spatial reference pattern* presented above. In this case, however, we are referring to an abstract machine, denoted by M, on which a generic array a has been distributed by a given strategy (e.g. **BLOCKING**).

The various communication routines can be denoted by using this formalism. Some important routines are `One-All-Broadcast`, `Single-Send-Receive`, and `Uniform-Shift`. For example, `Uniform-Shift`$(M, \mathbf{c}, a, a_1, B)$ defines the communication of all the data pointed by a in every virtual processor $\mathbf{i}$ in M, to the virtual processor $\mathbf{i}+\mathbf{c}$. a and a_1 denotes the input and the output data structures, respectively. B denotes the size of the data to be transmitted. $\mathbf{i}$ and $\mathbf{c}$ are shorthand for index tuples $(i_1, \ldots, i_n)$ and $(c_1, \ldots, c_n)$.

The authors distinguish between these communication routines, called *general communication routines*, and some *simple communication routines*, which involve only a single dimension of the index domain M. Some of these simple routines are `Spread`, `Reduce`, and `Shift`. For example, `Shift`$(M, p, c, a, a1, B)$ involves the pth dimension of M. It defines the communication of all the data pointed by a in every virtual processor $(i_1, \ldots, i_{p-1}, i, i_{p+1}, \ldots, i_n)$ to the virtual processor $(i_1, \ldots, i_{p-1}, i + c, i_{p+1}, \ldots, i_n)$.

By using the *communication pattern* notation, we have that the `Uniform-Shift` routine corresponds to the pattern

$$\ulcorner a@\mathbf{i} \Rightarrow \mathbf{i} + \mathbf{c} : M \urcorner$$

while the `Shift` routine corresponds to

$$\ulcorner a@(i_1, \ldots, i_{p-1}, i, i_{p+1}, \ldots, i_n) \Rightarrow (i_1, \ldots, i_{p-1}, i + c, i_{p+1}, \ldots, i_n) : M \urcorner$$

Matching algorithm Once we define spatial reference and communication patterns, and a given data distribution is applied, it is possible to devise an algorithm that selects the best communication routines (or combination of routines) that matches a given reference pattern. The matching algorithm works for a minimal granularity implementation, where the shape of the spatial index domain D is the same as the abstract machine index domain M, i.e $M = D$. Since each routine can be associated with a cost, in case there does not exist a unique perfect matching between a reference pattern and a communication pattern corresponding to a given routines, the algorithm selects the matching communication pattern with *the lowest cost*.

In our example, we have a perfect matching between a subset of the the following spatial reference pattern

$$P_1' \; : \; \ulcorner A@(h, j) \Rightarrow (i, j) : D \mid i \leq h \leq (i + 1) \urcorner$$

and the routine

`Shift`$(D, 1, 1, A, A1, 1)$, which corresponds to the following communication pattern

$$R \; : \; \ulcorner A@(i + 1, j) \Rightarrow (i, j) : D \mid True \urcorner$$

In fact, the communication pattern R corresponds to the following set of pairs, where each element of a pair $((i, j), (h, k))$, namely (i, j) and (h, k), identifies the source and the target, respectively, of a communication:

$$\begin{array}{ll} \{ & ((2,1),(1,1)),\ ((3,1),(2,1)), \ldots, ((1000,2),(999,2)), \\ & ((2,2),(1,2)),\ ((3,2),(2,2)), \ldots, ((1000,2),(999,2)), \\ & \vdots \\ & ((2,1000),(1,1000)),\ ((3,1000),(2,1000)), \ldots, ((1000,1000),(999,1000)) \\ \} & \end{array}$$

On the other hand, the reference pattern P'_1 identifies a superset of the set above. In fact, P'_1 identifies even more pairs, namely $\{((i,j),(i,j)) \mid \forall\ (i,j) \in D\}$, which, however, do not generate any communication, and are solved by the algorithm by simply generating local memory access.

Finally, the other spatial reference pattern P'_2

$$P'_2 \ : \ \ulcorner B@(i+1,j) \Rightarrow (i,j) : D \mid True \urcorner$$

becomes $P''_2 \ : \ \ulcorner B@(i,j) \Rightarrow (i,j) : D \mid True \urcorner$, under the hypothesis that `B(i+1,j)` is aligned with `A(i,j)` on the index domain D. Thus, even in this case, no communication is generated.

6.5 Open issues

All the compiling techniques we have surveyed translate a shared-memory program into a parallel program, where choices such as the number of processors to use, or the mapping of computation and data are performed *statically*. This approach guarantees the optimization of the communication routines to be inserted into the code to satisfy the data dependencies. Each routine, at run time and on the basis of this static knowledge, can arrange all the needed transmissions. This approach suffers from *load balancing* problems. In fact, the static approach determines an a-priori scheduling, which, depending on the specific data to be elaborated, can produce a very large load unbalance on the processors involved. There are several proposals to overcome this problem One of the most interesting is concerned with special language constructs to allow programmers to define a redistribution, to be performed at fixed time instants during the computation.

Another problem is concerned with the *distribution* strategies. The ability of programmers to specify effective distributions can be very critical. Unfortunately, when the references are statically predictable, there exists some proposals of automatic heuristics to generate suitable distributions. When this is no longer true, neither an automatic tool nor a programmer can determine the best strategy to use. Hence, programmers have to perform a process of *trial and error*, during which several distributions are forced into the program, and, at the end, the solution characterized by the best performance is chosen.

References

[1] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] P. P. Chang and W. W. Hwu. Trace Selection for Compiling Large C Application Programs to Microcode. In *Proc. of 21th Annual Workshop on Microprogramming and Microarchitecture*, pages 21–29, December 1988.

[3] C. Cugia and F. Pasqualetti. *Paradigmi di Programmazione Data-Parallel.* Dipartimento di Informatica – Universitá di Pisa, 1993. Master thesis (in italian).

[4] J.R. Ellis. *Bulldog: a compiler for VLIW architectures.* MIT Press, 1986.

[5] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[6] J.A. Fisher. The VLIW Machine: A Multiprocessor for Compiling Scientific Code. *IEEE Computer*, 17(7):45–53, July 1984.

[7] J.A. Fisher and S.M. Freudenberg. Predicting Conditional Branch Direction from Previous Runs of a Programs. In *ASPLOS-V Proceedings*, pages 85–95, Boston, Massachusetts, October 1992.

[8] M.J. Gonzales. Jr. Deterministic Processor Scheduling. *ACM Computing Surveys*, 9(3):173–204, September 1977.

[9] High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993. Version 1.0.

[10] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):67–80, August 1992.

[11] H.T. Kung. Why Systolic Architecture ? *IEEE Computer*, 15(1):37–46, January 1982.

[12] S.Y. Kung. *VLSI Array Processors.* Prentice-Hall International, 1988.

[13] M Lam. Software Pipelining: An Effective Scheduling Technique for VLIW. In *Proc. of ACM SIGPLAN '88 Conf. on Prog. Lang. Design and Impl.*, pages 318–328, Atlanta, Georgia, June 1988.

[14] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–383, July 1991.

[15] S.P. Midkiff and D.A. Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[16] A. Nicolau and J.A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers*, C-33(11):968–976, November 1984.

[17] D.A. Padua, D.J. Kuck, and D.H. Lawrie. High-Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[18] P. Quinton. An Introduction to Systolic Architectures. In *Future Parallel Computers: an Advanced Course*, pages 387–400, Pisa, Italy, June 1986. LNCS 272 Spinger-Verlag.

[19] M. Wolfe. Multiprocessor Synchronization for Concurrent Loops. *IEEE Software*, pages 34–42, January 1988.

[20] H.S. Zima and B.M. Chapman. Compiling for Distributed-Memory Systsssems. *Proceeding of the IEEE*, pages 264–287, February 1993.