The ASSIST 1.3 Tutorial

November 30, 2007

Abstract

This manual is an early draft of the ASSIST user manual, it might therefore contain some imprecisions. Moreover, the ASSIST Grid-aware programming environment is a product under continuous evolution. This manual purposedly describes a proper subset (more stable features, actually) of the version 1.3. Please regularly check the ASSIST web-site for updates, which are already ongoing (http://www.di.unipi.it/groups/architetture/). The ASSIST programming environment as been developed at Computer Science department of University of Pisa, Italy with the support of several national projects (especially *Grid.it*). It is an open source product under GPL license. Any comment is welcome.

Contents

Chapter 1

Introduction

ASSIST (A Software development System based on Integrated Skeleton Technology) is a parallel programming environment provided with a coordination language, namely ASSIST-CL, for the encoding of distributed and high performance applications on a wide range of target architectures including massively parallel cluster/networks of workstations.

The main goals of the environments are:

- providing performance portability, reusing of software, integration of standard and language interoperability
- flexibility and possibility to exploit all the feasibility of the structured parallel programming paradigm
- allowing the usage of homogeneous and distributed platforms
- defining an programming environment open to new constructs and mechanisms

The structure is modular and allows both to exploit the sequential code written in common languages such as C, C++ and FORTRAN, and to write new applications using one of these host languages. This document describes the environment and provides a preliminary guide for first time programmers.

1.1 ASSIST-CL keywords

ASSIST-CL provides the following fundamental constructs:

- sequential module
- parallel module (`parmod`)
- graph connecting two or more modules (`generic`)
- stream establishing the connections, i.e. the program graph
- `proc`, in which the user encapsulates the code to be executed

ASSIST allows to organize parallel and sequential modules in a graph programming structure in which data flow from the root to the peripheral vertexes. Flowing of data is regulated by a data-flow semantics, i.e. each node of the graph "consumes" a data flowing through it's input edges as soon as they are available as input. The graph is described by expressing (combination of) high level constructs building the application structure. Thus, ASSIST provides construct for expressing a sequential function, a parallel activity, a non deterministic choice between two or more data-flow input on a node level and so on. The basic building blocks of this program construction are pieces of sequential code representing the "pure" computation, that can be written in several programming language: C, C++, FORTRAN.

In the sequel, we will formalize all these intuitive ideas by means of the following concepts:

- the concept of **module**: a module is the entity representing a node of the application graph, and it can be identified by a name; the edges through which data flow are represented by connections between modules; such connections are expressed in terms of input/output parameters defined on the module interface
- an edge is represented by a **stream**, i.e. a (possibly) infinite set of values flowing from a source to a destination. A stream is the programming entity joining two (or more) modules, i.e. two (or more) nodes of the graph.
- the **graphs** one can expressed are *not* only DAG. Inner loops can be written keeping safety and reliability of the program.
- the **parmod** is a special kind of module that will allow us to structure, describe and implement the parallel behavior of the application. is a special kind of parallel

This tutorial is organized as follows: Chapter 2 provides the basic notions about the language and the compilation steps needed to run an ASSIST application; Chapter 3 details how to implement a sequential module; Chapter 4 provides the description of the `parmod` structure and give some sample application; Chapter 5 provides a complete overview of the syntax related to the implementation of a `parmod`.

Chapter 2

ASSIST-CL basics

2.1 An overview of the ASSIST program structure

An ASSIST program is represented by a graph whose vertexes, called *modules*, are connected by arcs, called *streams*. Each module has its own control flow and implements a sequential or a parallel program accepting and processing data from one or more `input_streams` and sending data to one or more `output_streams`. Streams are statically typed.

Figure 2.1 depicts an example of ASSIST program whose graph is composed by four modules M1, M2, M3 and M4, each of which could have a parallel or a sequential behavior. M1 is the "root" vertex that sends tasks to M2, M3, and M4 through the streams s1, s2, and s3, respectively. These streams can be managed either *independently* or *collectively* by M1. In the same way, M4 can receive either non-deterministically or according to a defined policy from its input streams s3, s4, and s5.

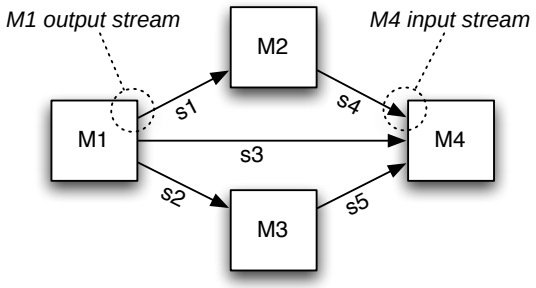

Figure 2.1: A typical ASSIST graph

The definition of an ASSIST program consists of two main steps: the definition of the graph of modules, and the definition of all the modules. Those definitions should be included in a single file, which may include other files via a suitable "include" statement.

The graph of modules is defined in a single block identified by the `generic main` keywords. This block includes the definition of all streams appearing in the application, and the prototypes of all application modules. The wiring among modules is defined by occurences of stream names in modules prototypes.

Modules are defined as blocks identified by a name and by a list of optional formal parameters. These parameters are variables representing input or output streams.

```
generic main(){
  // definition of streams
  stream long s1;
  stream long s2;
  stream float s3;
  stream float s4;
  stream float s5;

  // prototypes if modules and wiring
  M1 (output_stream s1, s2, s3);
  M2 (input_stream s1 output_stream s4);
  M3 (input_stream s2 output_stream s5);
  M4 (input_stream s4, s3, s5);
}

// definition of modules
M1 (output_stream long a, long b, float c) {
// module body
}

M2 (input_stream long a output_stream float b) {
// module body
}

M3 (input_stream long x output_stream float y) {
// module body
}

M4 (input_stream float first, float second, float third) {
// module body
}
```

The module body resemble a function definition in the C++ language. As we shall see in Chapter 3 and 4, an ASSIST module can exhibit either a sequential or parallel control flow. A flow of control is defined by a sequential procedure, called `proc`, which can be programmed in one of the sequential guest languages supported by ASSIST, such as C, C++, and Fortran. The sequential module is the simplest ASSIST module. It just calls a `proc`. Let us exemplify the concept by means of the classic "Hello world!" example.

2.1.1 Hello world!

We implement the classical "Hello World!" example by means of a module called `helloworld`. The (sequential) module just calls a standard C++ function called `print_helloworld`. The `$c++` and `c++$` modifiers are used to denote the definition of a C++ `proc`; the `inc<"filename">` statement is used to include header files in the `proc` definition and the `path<"absolute_path">` statment declares where they are located on the filesystem.

```
generic main(){
  //no streams, just a single module
  helloworld();
}

helloworld(){
   print_helloworld();
}
```

```
proc print_helloworld()
path<"/usr/include/c++/4.1.1">
inc<"iostream">
$c++{
  std::cerr << "Hello World!" << std::endl;
}c++$
```

Environment settings In order to compile and run the application, check that your environment has been correctly defined. Particularly, check that paths and variables defined in your `/.ast_rc.sh` file are coherent with your system configuration and remember to include the loading of the needed settings in your shell configuration profile.

Compilation Let us suppose the code has been saved in the file `HelloWorld.ast`. In order to execute such code, run the ASSIST-CL compiler. In your working directory type

```
user@localhost:~ > astCC -c HelloWorld.ast
```

for compiling the `HelloWorld.ast` application. The compiler will store the file representing the executable code `ast.out.xml` in the ASSIST binary target directory specified in the ASSIST configuration file (∼`/.ast_rc`) and represented by the `compiledir` tag.

Execution In order to run the application, you need to invoke the ASSIST loader, which will load the binary of the application on the target platforms (specified in the `$LOADER_ROOT xml cluster.xml` file). The ASSIST loader works in three successive steps: *i*) the set of target platform is defined, *ii*) the binary code is deployed onto the target platforms, *iii*) the binary code is launched onto the target platforms. The `loadEx.sh` script, in the `$LOADER_ROOT comandi` directory, calls all the steps in sequence and it can be lauched by typing

```
user@localhost:~ > loadEx.sh $LOADER_HOST $LOADER_PORT
/shared/ASSIST/tmp/assist_compiledir/bin/ast.out.xml noLibTransfer

/usr/java/jdk1.5.0_12/bin/java it.unipi.di.gam.LoadExecuteDriver q1 12700
/shared/ASSIST/tmp/assist_compiledir/bin/ast.out.xml noLibTransfer
Class LoadExecute Driver args.length: 4
Identifying the option: noLibTransfer
Connected with server q1/192.168.0.1:12700
Result CaricaCluster :
SUCCESS
Connected with server q1/192.168.0.1:12700
Result Load :
SUCCESS
libHandle -> 5
Connected with server q1/192.168.0.1:12700
Result Exec :
SUCCESS
runHandle -> 5
```

`$LOADER_HOST` is the name of machine where the ASSIST loader is running (on port $LOADER_PORT), and `ast.out.xml` is the XML file generated by the compiler. The `NoLibTransfer` option avoids the deployment of common libraries, which is usually a not required step in a cluster enviroment. The advanced usage of the ASSIST loader is described in the ASSIST loader manual.

> **VirtuaLinux Feature 1** *The VirtuaLinux distribution comes with a pre-configured and pre-installed version of ASSIST v1.3. In VirtuaLinux the standard binary target directory is `/shared/ASSIST/tmp/assist_compiledir/`, and all nodes of the VirtuaLinux cluster*

are target platforms, which are listed in `/usr/local/astCC1_3/Loader/xml/cluster.xml`. *The* `$LOADER_HOST` *and* `$LOADER_PORT` *are pre-defined with the following values:* `$LOADER_HOST` = *"first node of the cluster" and* `$LOADER_PORT` = *12700. The ASSIST loader manual can be found in* `/usr/local/astCC1_3/Loader` *folder.*

2.2 Type system

ASSIST supports C, C++ and Fortran languages even if all the examples given in this tutorial will be coded by using the C++ programming language. In order to provide compatibility between ASSIST-CL syntax and C, C++ and Fortran ones, the ASSIST type system is described as a set of mappings between ASSIST types and C, C++, Fortran types and the list of such mappings is given in Table 2.1. Indeed, in order to cope with high-performance specific needs, the type system is extended with some structured types (i.e. defined by the `typedef` directive in the C, C++ syntax) that are provided with ASSIST-CL without corresponding definition in the native language: `fcomplex` and `dcomplex` representing complex values and `ref_t` representing references to external objects.

ASSIST	**C**	**C++**	**Fortran**
octet	unsigned char	unsigned char	integer*1
char	char	char	character
bool	--	bool	logical
short	short	short	integer*2
long	int	int	integer
long long	long long	long long	integer*8
float	float	float	real*4
double	double	double	real*8
fcomplex	fcomplex	fcomplex	complex*8
dcomplex	dcomplex	dcomplex	complex*16
ref_t	ref_t	ref_t	--
-			

Table 2.1: ASSIST type system in comparison with C, C++ and Fortran types.

2.2.1 Basic and composite types

Table 2.1 lists ASSIST basic types and their corresponding C, C++ and Fortran types. New composite types can be declared by means of the `typedef` specifier, as in the C language. In particular, structures and arrays can be recursively used to define new types. As an example a `myType_t` type can be defined as follows:

```
typedef struct {
  long a;
  char b;
} myType_t;
```

2.2.2 Macros

A macro can be defined by using the `define` keyword. As an example teh following line defines N as the value 10 according to the C language syntax:

```
define N 10
```

All occurrences of the macro `N` will be replaced by the ASSIST pre-processor with the value "`10`". The ASSIST pre-processor is compliant with the C pre-processor.

2.2.3 Array variables

An array variable can be declared following the C syntax:

```
long A[N][N];
```

defines a bi-dimensional array `A` composed of $N \times N$ longs.

2.2.4 Stream variables

A stream represents a one-way channel between ASSIST modules. Informally, a stream is an unbounded ordered list of values of the same type. The declaration of a stream includes its `name` and the `type` of the stream items. Legal types for streams are ASSIST-CL basic types or new types defined by the user, as seen in the previous section. In the following example two streams are declared:

```
stream long streamLong1;
stream myType_t streamMyType1;
```

The stream named `streamLong1` carries values of type `long`; stream `streamMyType1` carries values of type `myType_t`.

Streams of arrays may be directly declared without introducing a new type name. Note that in this case ASSIST requires a slightly different syntax, with respect to the C standard which is normally used to declare ASSIST types. As an example:

```
stream long [N][M] matrixStreamNM;
```

declares a stream which items are bi-dimensional arrays $N \times M$ whose elements are long values.

Chapter 3

The sequential module

A sequential module is a wrap that enables standard sequential functions to inter-operate via streams. A sequential module is activated by the presence of at least one item in all input streams. At the activation time, the module's `proc` is instatiated with the stream items. Once the `proc` has been evaluated, the module may write one or more items in each output stream, and iteratively waits for a new set of items from the input streams. The sequential module terminates when all partners attached to its input streams terminate. In the case the module has no input streams, it is activated just once.

3.1 Sequential module interface

A sequential module is defined by an interface and a body. The interface declaration needs a name for the sequential module, and all input and output streams as formal parameters. The body is specified between brackets.

```
myFirstSeqModule(input_stream <list of parameters>
                 output_stream <list of parameters>)
{
// declarations and body of this module
}
```

In the example above, we can see the declaration of a sequential module interface, in which the name of the module (`myFirstSeqModule`) and input and output streams are specified. Note that the list of input streams follows the keyword `input_stream` and the streams are listed by specifying their *type* and *identifier*. The same rule applies on the list of output streams introduced by the keyword `output_stream`.
As instance, in the following interface declaration

```
myFirstSeqModule(input_stream long n, double d
                 output_stream long A[N][N], float f, long long ll)
{
  // declarations and body of this module
}
```

`myFirstSeqModule` is a module taking as input two input streams of type `long` and `double` respectively, and three output streams of type `long A[N][N]`, `float` and `long long` respectively. Note that the list of output streams follows the list of input streams and *no* comma separates them (while the elements of each list are comma-separated, instead).

3.2 Including Existing Headers and Code

In order to use existing code and declarations already available in different source files (e.g. a C++ class declaration in an header file), their paths and names have to be included in the module definition in the following way:

```
myFirstSecModule(){
  path<"/home/user/src/">
  inc<"iostream","myobject.hpp">

  // body of this module
}
```

The `path` directive indicates the path in the file system where to search for include files; the `inc` directive indicates the names of the files to include.

3.3 The body of a sequential module: the `proc` construct

A sequential module can be implemented as a list of calls to `proc` (see also Sec. A.1 to learn about the implementation of *pure* sequential modules). A `proc` is a kind of procedure wrapping sequential code in any of the host language supported. The host language is declared by using the modifier $ coupled together with the language specification string outside the function body.

A `proc` may have input parameters (whose list is tagged by the `in` keyword), output parameters (whose list is tagged by the `out` keyword) and an output stream identifier tabbed by the `output_stream` keyword. Thus,

`proc (in` *opt_input_param_list*`, out` *opt_out_param_list*`, output_stream` *opt_output_stream_id*)

Input parameters are optional but is extremely important to take into account that *for each* input value belonging to the input list

- *only a single* value or a list of *independent* values are produced at a time (the ones listed in the *opt_input_param_list*), or
- a stream of values are produced, thus instancing the *opt_output_stream_id*.

Fig.3.1 summarizes the behavior of a sequential module: assuming that it provides an input parameter a, it can produce both an output parameter b and a stream of values $c_0, c_1, \ldots, c_{k-1}$. Thus, for each value of a, depending on the `proc` code, *just one* value for b and/or a list of values flowing through the output stream c are produced. With respect to having a single output value as the result of a `proc` execution, the mechanism that allows to produce the stream of values $c_0, c_1, \ldots, c_{k-1}$ while the `proc` keeps running is the `assist_out` routine (see Sec. 3.4). The role of this routine is to instance a single, new value onto the output stream without waiting for the termination of the `proc`.

Let us take into account a producer-consumer example as depicted in Fig.3.2. This graph can be implemented in the following way.

```
// -*- C++ -*-

#define NUM1 10
#define NUM2 15

generic main()
{
  stream long A;
  stream long B;
```

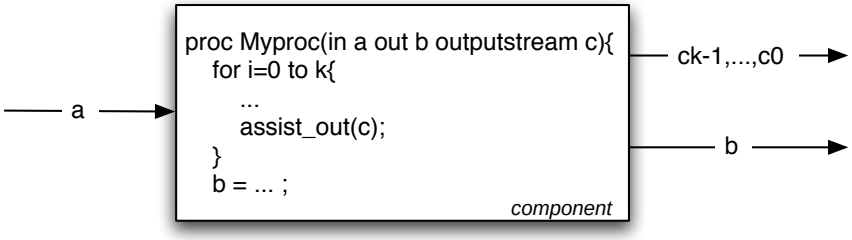

Figure 3.1: A component whose internal behaviour is coded by a `proc`.

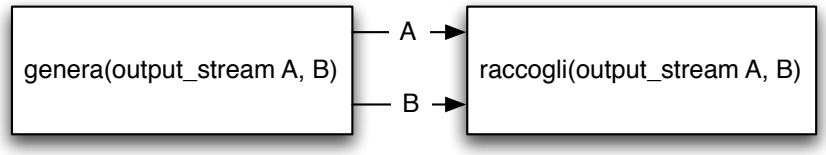

Figure 3.2: A graph implementing a producer-consumer paradigm

```
  genera   (output_stream A, B);
  raccogli (input_stream A, B);
}


genera(output_stream long A, long B) {
  pGenera(out A,B);
}


proc pGenera(out long A, long B)
inc<"iostream">
$c++{
  std::cerr << "Sending value A: " << NUM1 << " B:" << NUM2 <<std::endl;
  A = NUM1;
  B = NUM2;
  std::cerr << "Here you have some other computation" <<std::endl;
}c++$


raccogli(input_stream long A, long B) {
  pRaccogli(in A,B);
}

proc pRaccogli(in long A, long B)
inc<"iostream">
$c++{

  std::cerr << "Receiving A: " << A << "B: " << B << std::endl;

}c++$
```

The producer is implemented by the module `genera` and provides two output streams, A and B, whose single values are produced by the `proc` `pGenera` (actually, here `pGenera` simply implements an assignment to the output value to be produced, A and B and a printing directive). The consumer is implemented by the module `raccogli` and it receives input values through the input stream A and B (the ones produced by module `genera`) and consumes such values by invoking the `proc` `pRaccogli`. Note that if `genera` and `raccogli` are declared and implemented in a different location, they can be linked by specifying the `inc` and `path` directives described in 3.2.
Each receiving module activates its `proc` as soon as a task is available onto the input stream. Thus, the execution of the example will generate the output

```
Sending value A:10 and B: 15
Here you will have some other computation

Received value A:10 and B:15
```

If we slightly change the code of `pgenera` by commenting the assignment of `B` (thus, neglecting the module `genera` to send a value onto the output stream `B`), the output provided by the execution will be

```
Sending value A:10 and B: 15
Here you will have some other computation

Received value A:10 and B:-1077514208
```

e.g. the only availability of a task on the input stream `A` suffices to activate the `proc` `pRaccogli` (we will see in Chapter 4 how such behavior can be pragmatically changed).
In case a module provides no input streams (i.e. no input values), each `proc` is activated at the module launching time and it stay running until the module terminates.
An input stream value can be given as argument to multiple `proc`, while each output stream should appear in *at least one* of the `proc` output parameters. As an example, the following code is ill-formed and during its compilation an error will be raised because of the non-sense of declaring an output stream flowing out two different `proc`:

```
  p1(is_n, out_l1);
  p2(is_n, out_l1);
```

3.4 The `assist_out` command

As mentioned above, the correct use of the sequential module semantics requires that the sending of a result onto the output stream have to be completed after its evaluation has been completed. Nevertheless, this means that *just one result* will be passed on each output stream. However, it is possible to sequentially submit multiple data items on an output stream *during a single activation*. In ASSIST this can be done using the `assist_out` command.

```
// -*- C++ -*-

#define ITERATIONS 10

generic main()
{
  stream long lout;

  seqModuleMultipleSend(output_stream lout);
  raccogli(input_stream lout);
}
```

```
seqModuleMultipleSend(output_stream long lout){
  multipleSend(output_stream lout);
}


proc multipleSend(output_stream long plout)
inc<"iostream">
$c++{
  int i;
  for (i=0;i<ITERATIONS;i++){
    assist_out(plout,i);
    std::cerr << "SENT value i:"<< i << std::endl;
  }
}c++$


raccogli(input_stream long A){
  pRaccogli(in A);
}

proc pRaccogli(in long A)
inc <"iostream">
$c++{
  std::cerr<<" Received value A:"<< A << std::endl;
}c++$
```

The example above represents a variant of the producer-consumer case with just one stream connecting the two modules. The producer, represented by the `genera` module, sends `INTERACTIONS` long values on the output stream named `lout` *in each single activation*. For each value produced, the `assist_out` primitive allows to send the current value `i` on the output stream `plout`.

As it can be seen from the following example, the `assist_out` command allows to design more complex application patterns. Let us take into account the producer-consumer schema given in the previous section provided that the producer sends to the consumer two streams of values for each activation of the `pGenera proc`. `pGenera` does not receive input values but, for each activation, produces two streams of long values, named `A` and `B`. Such values are generated inside the body of the for construct (syntactically represented by `tmpA` and `tmpB`), and they are sent to the corresponding output stream through the `assist_out` command each. In fact, the activation of a `proc` stops when the expected output value as been produced: in order to keep in running the `proc` (calling the instructions that follow such value generation), we need to adopt the `assist_out` directive, after which the execution proceeds with the successive instructions.

```
// -*- C++ -*-

#define NUM1 10
#define NUM2 15
#define TIMES 100

generic main()
{
  stream long A;
  stream long B;

  genera   (output_stream A, B);
  raccogli (input_stream A, B);
}
```

```
genera(output_stream long A, long B) {
  pGenera(output_stream A,B);
}


proc pGenera(output_stream long A, long B)
inc<"iostream">
$c++{
  for (unsigned int i=0; i< TIMES ; i++) {
    long tmpA;
    long tmpB;

    tmpA = NUM1;
    tmpB = NUM2;
    std::cerr<<"Task "<<i<<"sent A:"<<tmpA<<" B:"<<tmpB<<std::endl;

    assist_out(A,tmpA);
    assist_out(B,tmpB);

  }
}c++$


raccogli(input_stream long A, long B) {
  pRaccogli(in A,B);
}

proc pRaccogli(in long A, long B)
inc<"iostream">
$c++{

  std::cerr << "Value Received A: " << A << "B: " << B << std::endl;

}c++$
```

Note that this implementation presents a drawback: in the `pRaccogli proc`, it is not possible to count how many times the pair (a,b) has been received because of each activation depends on just one pair. The only way to overcome this lack, is to use a `static` variable inside the C++ code.

As a last example, we will show how to write a producer-consumer application in which each task of the stream is of type `DataType`, a C++ `struct` encapsulating two long values and a counter of the tasks of the stream.

```
// -*- C++ -*-

#define NUM1 10
#define NUM2 15
#define TIMES 100

typedef struct {
  long A;
  long B;
  long msgId;
} DataType;

generic main()
```

```
{
  stream DataType A;

  genera   (output_stream A);
  raccogli (input_stream A);
}

genera(output_stream DataType A) {
  pGenera(output_stream A);
}

proc pGenera(output_stream DataType msg)
inc<"iostream">
$c++{
  for (unsigned int i=0; i< TIMES ; i++) {
    DataType elemento;
    elemento.A = NUM1;
    elemento.B = NUM2;
    elemento.msgId = i;
    std::cerr << "Task " << i << "sent A: " << element.A << " B:" << element.B <<std::endl;
    assist_out(msg,elemento);
  }
}c++$


raccogli(input_stream DataType A) {
  pRaccogli(in A);
}

proc pRaccogli(in DataType data)
inc<"iostream">
$c++{
   if (data.msgId==99) {
    std::cerr << "All tasks received " << std::endl;
  }
}c++$
```

The `procpGenera` produces TIMES tasks to be sent onto the stream; each task is a `DataType` type element and is "tagged" by its own task identifier `msgId`. Each task is sent onto the stream through the `assist_out` command and received by the `pRaccogli proc` that simply counts how many data items has been received (it simply checks if the current message carries 99 as task identifier). Such pattern can be easily applied each time a pipeline computation is involved: `pGenera` produces task for the pipeline, `pRaccogli` eventually represents the last stage of the pipeline and in the middle of these stages, an application could have a number of intermediate stages that send and receive tasks on their streams.

3.5 Generic graphs

Let us suppose we want to implement the graph depicted in Fig.3.3. Trivially, the graph represents the evaluation of the expression

$$f(x) = (x * 10) \times 2 + (x * 10)^2$$

Each operator is represented by a node and the arch of the graph represents the dependencies among arithmetic expressions.

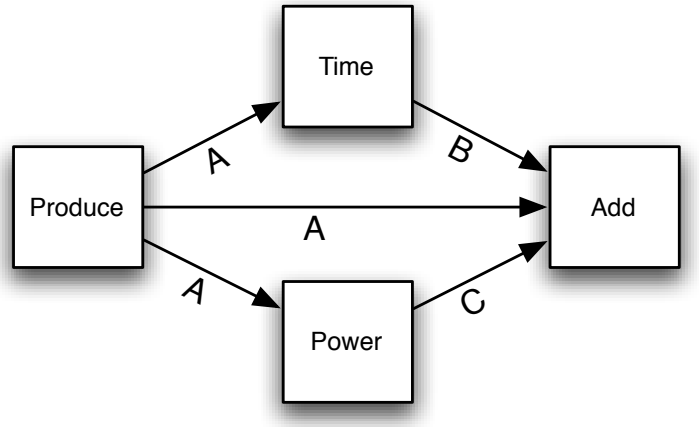

Figure 3.3: A more complex graph of sequential modules

The graph represents some of the most typical programming situations and we will show how to encode them:

- a sequential module (`Produce`) producing a stream of values and connected to (three) different modules (`Power`, `Time`, `Add`) through a common stream (`A`); e.g. the same input stream can be consumed by several receivers;
- two sibling (`Time` and `Power`) working independently, the one with respect to the other, on common inputs;
- a (sub)graph that could be provided with a double root: `Time` and `Power` are both roots of a (sub)graph representing the expression $y \times 2 + y^2$
- a sequential module connected to three different and unrelated input streams

All these situations are interchangeable, they could appear on the same node or in different nodes of the graph.

```
// -*- C++ -*-
#define N 10

generic main(){
  stream long A;
  stream long B;
  stream long C;

  // declaration of the root
  producer(output_stream A);

  // first of two siblings
  multiplier(input_stream A output_stream B);

  // second of two siblings
  power(input_stream A output_stream C);

  // this module receives from two sources
  add(input_stream B, C);
}
```

```
 producer(output_stream long A){
   p_producer(output_stream A);
 }

 proc p_producer(output_stream long A)
 inc<"iostream", "stdlib.h","sys/types.h","sys">
 $c++{
   int i;
   int v;
   for ( i=0 ; i < N ; i++){
     v = (rand()%9)*10;
     std::cerr<<"Sending value:"<<v<<std::endl;
     assist_out(A,v);
  }
}c++$

 multiplier(input_stream long A output_stream long B){
   p_mult(in A out B);
 }

 proc p_mult(in long A out long B)
 $c++{
   B = A*2;
 }c++$

 power(input_stream long A output_stream long B){
   p_power(in A out B);
 }

 proc p_power(in long A out long B)
 $c++{
   B = A*A;
 }c++$

 add(input_stream long B, long C){
   p_add(in B,C);
 }

 proc p_sum(in long B, long C)
 inc <"iostream">
 $c++{
   long val = C+B;
   std::cerr<<"Receiving value:"<< val << std::endl;
 }c++$
```

Note that, in order to use the C function `rand()` in the `proc p_producer`, we have included the standard library `stdlib.h` through the `inc` directive.

3.6 Reading from and writing a file

Reading and writing operations from a file system source could be easily done by using the primitives of the host programming language. In the example below, a version of the `proc p_producer` is given, in which the values to which applying the function $f(x)$ are read from the source file `/tmp/contentfile`.

```
proc p_producer(output_stream long A)
 inc<"iostream", "stdlib.h">
$c++{
  int i;
  int v;
  FILE *fd;

  fd = fopen("/tmp/contentfile","r+");

  if (fd==NULL){
   std::cerr<<"Error in opening file"<<std::endl;
   exit(0);
  }else{

    while (!feof(fd)){
      val = getw(fd);
      std::cerr<<"Reading value:"<<val<<std::endl;
      assist_out(A,val);
    }
    fclose(fd);

  }
}c++$
```

Note that a file name could also be passed to a `proc` as the string value of the `proc` ś input stream.

Chapter 4

The parallel module

An ASSIST program is represented by a *graph* of *generic* structure, whose nodes are sequential or parallel modules and edges are stream of data. A parallel module or `parmod` is the main parallel construct in ASSIST-CL because it allows to explicitly express a parallel behaviour inside a single module or inside a single node via compositions of sequential and/or parallel modules. Compositions are mainly expressed in the *data-flow* and/or in the *not-deterministic* style.

The chapter is organized as follows: in Section 4.1 we will give an introduction of the structure of a `parmod` ; in Section 4.2 the `parmod` interface will be presented together with a first overview of the sections to be implemented for programming it; in Sections 4.3-4.6 we present some classic parallel applications exploiting the main `parmod` features. However, a full, detailed description of such features will be given in Chapter 5.

4.1 Structure of the parmod

A `parmod` external interface is similar to a sequential module interface: it exposes input streams and output streams and they are used to interact with the external environment.
ASSIST provides keywords for programming the development and assembly mechanisms at the language level, for accessing the tasks non deterministically arriving from the stream and to collect the results. Moreover, synchronization mechanisms for managing shared tasks inside parallel modules are also provided.

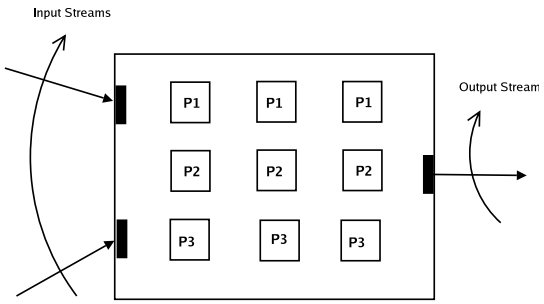

Figure 4.1: `parmod`: virtual processors on row i execute the program P_i.

From a high abstraction level, a `parmod` is represented by a set of *virtual processors* (in the following, VPs), i.e. the units of parallelism of a `parmod`. The logic at the basis of such set is that VPs execute in parallel, possibly different, programs and they are programmed similarly to a sequential module, i.e. as a list of calls to `procs`. Moreover, some specific ASSIST-CL constructs are provided in order to specialize the module behavior.

An example of `parmod` with nine virtual processors is shown in Figure 4.1, where if two of them are on the same row, then they execute the same program (for example in the first row they execute the program P1). Virtual processors can communicate together by means of state sharing, as we will see in the subsequent paragraphs. Thus, a `parmod` can have a state that can be replicated or partitioned among the virtual processors and receives/sends data through the input/output streams provided by the external interface.

Operationally, the `parmod` behaves differently from a sequential module since

- it can activate himself even if not all its input streams received a token;
- it can receive/send data items on input/output streams even if internally there is still an active computation.

These characteristics will depend upon the "configuration" the programmer gives to the `parmod`.

4.2 The `parmod` interface

A `parmod` is declared by specifying its name and its input and output streams exactly as if it was a sequential module. Look at the following example

```
generic main(){

 parmod myFirstParmod (input_stream <list of input streams>
                       output_stream <list of output streams>) {
   /* Definitions section */
   /* Input section */
   /* Virtual processor section */
   /* Output section */
 }
}
```

The `parmod`'s body include four main programming *sections*:

- in the *definition* section a *topology* must be specified, and attributes and control variables needed to the program can be declared;
- in the *input* section the `parmod`'s behaviour regarding the input streams and the elements distribution is specified;
- in the *virtual processors* section the virtual processors can be programmed;
- in the *output* section the programmer specifies the `parmod`'s behaviour with respect to both the reception of results from VPs and the sending of them to the output streams.

In the following we examine in detail each of the above sections in three specific classes of problems:

- an embarrassingly stream-parallel application (a farm skeleton)
- an embarrassingly data-parallel application (a map skeleton)
- a data-parallel application with stencil (i.e. with functional dependencies between tasks and/or data overlapping between virtual processors).
- the fourth example will highlight the not-deterministic reception of input tasks

The examples proposed in the next sections will give a first perspective of how to program a `parmod`, whereas Chapter 5 will focus on all the syntactic options and details about these four main sections.

4.3 Stream parallel application

A classic stream parallel application is the one evaluating the Mandelbrot set of a squared space of N points. The computation of the color to be assigned to a point of the space is independent for each point with respect to the others, thus the problem can be described through a farm skeleton, i.e. by the following graph application:

```
generic main()
{
  stream dcomplex[N] A; // the set of pixel coordinates
  stream T_col[N] C;    // the range of colors

  generate     (output_stream A);              // generates a stream of pixels
  mandelbrot (input_stream A output_stream C); // evaluates mandelbrot
  collect   (input_stream C);                  // collects the image colors
}
```

where `T_col` represents the association between a pixel a color and it's a structured user defined type defined as follows:

```
typedef struct {
  dcomplex pos; // the pixel coordinates
  long col;     // the color of the pixel
} T_col;
```

The module `mandelbrot` encapsulates the parallel behavior of the application, while `generate` and `collect` are sequential modules producing and consuming the stream values, respectively, as already discussed in Chapter 3. Since these last two modules do not offer particular new challenges (see Section 3.3 for details), we will focus here only on the `parmod` code.

```
// -*- C++ -*-
#define N 10000 #define MAXITER 4096

// other code....

parmod mandelbrot (input_stream dcomplex A[N] output_stream T_col C[N])
{
  topology none Pv;

  do input_section {
    guard1: on , , A {
      distribution A on_demand to Pv;
    }
  } while (true)

  virtual_processors {
    mandel (in guard1 out C) {
      VP {
        Fmandel (in A out C);
      }
    }
  }
  output_section {
    collects C from ANY Pv;
  }
}
```

```
proc Fmandel (in dcomplex pos[N] out T_col sol[N])
inc<"iostream">
$c++{
  for (int n=0; n<N; n++) {
    dcomplex z=0, c=pos[n];
    long i=0;
    while (i<MAXITER) {
      z=z*z-c;
      if (abs(z)>2) break;
      i++;
    }
    sol[n].pos = c;
    sol[n].col = i;
  }
}c++$
```

The `parmod` takes a stream `A` of `dcomplex` arrays as input and produces a stream of `T_col` array as output `B`, so that for each element $x_i \in$ `A`, $c_i \in$ `T_col` represents the color associated to x_i.

The topology

```
topology none Pv;
```

The topology section defines the set of virtual processors onto which the elaboration will be eventually distributed. A topology of type `none` defines a set of anonymous virtual processors, all executing the same parallel elaboration.

Input section In the input section the behavior of the parmod with respect to the reception of the input values will be explicited. If the `parmod` has multiple input streams from which items can be received, a set of *guards* will allow the non deterministic reception of input items from all the streams.

```
do input_section {
   guard1: on , , A {
     distribution A on_demand to Pv;
   }
 } while (true)
```

The guards are listed in the `input_section{...}` block and the expression

`guard1: on, , A{}`

states that the guard `guard1` is activated (thus his body is executed) as soon as a new item is available onto the input stream `A`; the keyword `on` assigns the same priority of all the guards of the list. Eventually, a boolean condition on the guard can follow the priority keyword, otherwise a blank parameter can be given as the boolean value *true*.

The distribution strategy The body of a guard, is responsible for the distribution of the incoming items to the virtual processors. The keyword `distribution` introduces a strategy as, for example, the assignment *on-demand* of each incoming task to a virtual processor. The assignment anonymous and is defined by the run-time system aiming at keeping a good load-balancing.
The input section can be enclosed in a (possibly infinite) loop in order to keep running its activation after each item arrival.

Virtual processors section In this section the actual elaboration of the virtual processors is explicited.

```
virtual_processors {
   mandel (in guard1 out C) {
      VP {
        Fmandel (in A out C);
      }
   }
 }
```

In this context, `mandel(in guard1 out C){}` represents the *parallel elaboration* of the application. The execution depends on the activation of the guard in the input section (i.e. on the availability of an input item).

The body of the parallel elaboration specify the function applied by each anonymous virtual processor. In our example, all the virtual processors apply the function `Fmandel` on their assigned tasks. In other words, the construct `VP{...}` specifies that *all* the VP of this `parmod` will execute the function (the `proc`) included in the body of the construct that takes a value `A` as input and produces a value `C` as output.

Output section The output section is represented by the construct `output_section{...}` and it explicits how to collect the results provided by the virtual processor.

```
  output_section {
    collects C from ANY Pv;
  }
```

Here, the collection strategy is introduced by the keyword `collects` followed by the result identifier appearing in the parallel elaboration signature and the keyword `from ANY`. Such keyword specifies that the output section waits non-deterministically for the first result sent from *any* virtual processor belonging to Pv.

4.4 Data parallel application

We will take an application multiplying two squared matrices as an example of data parallel computation. The application can be coded by the following graph application:

```
generic main()
{
  stream long[N][M] Matrix1;
  stream long[M][L] Matrix2;
  stream long[N][L] Matrix_ris;

  generate (output_stream Matrix1);
  generate (output_stream Matrix2);
  matrixmult (input_stream Matrix1, Matrix2 output_stream Matrix_ris);
  end (input_stream Matrix_ris);
}
```

where `generate` represents the sequential module that generates the two matrices by assigning to each position randomly generated values, as follows:

```
generate(output_stream long Matrix1[N][M]) {
  fgen1(output_stream Matrix1);
}
proc fgen1(output_stream long Matrix1[N][M])
inc<"iostream">
$c++{
  long a[N][M];
  for (int i=0;i<N;i++)
    for (int j=0;j<M;j++)
      a[i][j] = ...              // generation of the value
  assist_out(Matrix1, a);
}c++$
```

while `end` trivially prints the resulting matrix as follows:

```
fine(input_stream long Matrix_ris[N][L])
 inc<"iostream">
 $c++{
  int ok = 0;
  for (int i=0;i<N;i++)
    for (int j=0;j<M;j++)
      std::cerr << Matrix_ris[i][j] << std::endl;
}c++$
```

Let's analyze in detail the `matrixmult` module representing the `parmod` showed below:

```
parmod prodotto_matrici (input_stream long Matrix1[N][M], long Matrix2[M][L]
                         output_stream long Matrix_ris[N][L]) {

  topology array [i:N][j:L] Pv;

  attribute long A[N][M] scatter A[*ia][*ja] onto Pv[ia][ja];
  attribute long B[M][L] scatter B[*ib][*jb] onto Pv[ib][jb];
  attribute long C[N][L] scatter C[*ic][*jc] onto Pv[ic][jc];

  stream long ris;
```

```
  do input_section {
    guard1: on , , Matrix1 && Matrix2 {
      distribution Matrix1[*i0][*j0] scatter to Pv[i0][j0];
      distribution Matrix2[*i1][*j1] scatter to Pv[i1][j1];
    }
  } while (true)

  virtual_processors {
    elab1 (in guard1 out ris) {
      VP i, j {
        UserSet(in Matrix1[i][j], Matrix2[i][j] out A[i][j], B[i][j]);
        sync;
        f_mul (in A[i][], B[][j] out C[i][j]);
        sync;
        assist_out(ris, C[i][j]);
      }
    }
  }

  output_section {
    collects ris from ALL Pv[i][j] {
      int elem;
      int Matrix_ris_[N][L];
      AST_FOR_EACH(elem) {
        Matrix_ris_[i][j]=elem;
      }
      assist_out(Matrix_ris, Matrix_ris_);
    }<>;
  }
}
```

The `parmod` takes two streams as input, each providing one of the two matrices, respectively and produces a third stream flushing the tasks belonging to the result matrix, as output value.
In this implementation, the evaluation of the multiplication proceeds as soon as both `Matrix1` and `Matrix2` are completely available as input to the `parmod` Thus, each element of the output matrix `Matrix_ris` can be computed *in parallel* as the Cartesian product between the $i-$th row of `Matrix1` and the $j-$th row of `Matrix2`.

The topology

```
  topology array [i:N][j:L] Pv;
```

The topology section defines the number of virtual processors onto which the elaboration will be eventually distributed. This implementation uses $N \times L$ virtual processors, one for each element $c_{i,j}$ we expect in the result matrix.
The topology type `array` allows to define $N \times L$ virtual processors as if they were elements of an array Pv. Thus, in the scope of the `parmod` each virtual processor will be indexed by `Pv[i,j]`, provided that $i \in [1, N]$ and $j \in [1, L]$.

Mapping of attributes Once we have given a topology to logically organize the set of virtual processors, we could need to partition one or more global variables that will collectively used by the virtual processors.

```
 attribute long A[N][M] scatter A[*ia][*ja] onto Pv[ia][ja];
 attribute long B[M][L] scatter B[*ib][*jb] onto Pv[ib][jb];
```

```
attribute long C[N][L] scatter C[*ic][*jc] onto Pv[ic][jc];
stream long ris;
```

The attributes represented by `A`, `B`, and `C` are bi-dimensional arrays of specific sizes, shared between all the virtual processors. The expression `scatter A[*ia][*ja] onto Pv[ia][ja]` means that ∀`i0,j0` A[i0][j0] is mapped onto the virtual processor `Pv[i0][j0]`. In other words, `Pv[i0][j0]` can access the whole matrix that becomes part of the `parmod` state.

Input section In the input section the behavior of the parmod with respect to the reception of the input values will be explicited.

```
do input_section {
  guard1: on , , Matrix1 && Matrix2 {
    distribution Matrix1[*i0][*j0] scatter to Pv[i0][j0];
    distribution Matrix2[*i1][*j1] scatter to Pv[i1][j1];
  }
} while (true)
```

The guards are listed in the `input_section{...}` block and the expression

`guard1: on, , Matrix1 && Matrix2{...}`

states that the guard `guard1` (the only one needed) is activated (thus his body executed) as soon as a new item is available onto both the input stream `Matrix1` *and* the input stream `Matrix2`. The keyword `on` assigns a default priority to this guard and the blank parameter is equivalent to expressing a condition on the guard that always evaluates *true*.

The input section can be enclosed in a (possibly infinite) loop in order to keep running its activation after each item arrival.

The distribution strategy The body of a guard is responsible for the distribution of the incoming items to the virtual processors.

```
distribution Matrix1[*i0][*j0] scatter to Pv[i0][j0];
distribution Matrix2[*i1][*j1] scatter to Pv[i1][j1];
```

The keyword `distribution` introduces a strategy corresponding, to the mapping of all the elements of the matrices to all the virtual processors, as specified by the free and fresh variables `i0` and `j0`

Virtual processors section In this section the actual elaboration of the virtual processors is explicited.

```
virtual_processors {
  elab1 (in guard1 out ris) {
    VP i, j {
      UserSet(in Matrix1[i][j], Matrix2[i][j] out A[i][j], B[i][j]);
      sync;
      f_mul (in A[i][], B[][j] out C[i][j]);
      sync;
      assist_out(ris, C[i][j]);
    }
  }
}
```

where `f_mul` and `UserSet` are trivially defined as follows:

```
proc f_mul(in long A[M], long B[M] out long C)
inc<"iostream">
```

```
$c++{
  // computes the actual cartesian product
  register long r=0;
  for (register int k=0; k<M; ++k)
    r += A[k]*B[k];
  C = r;
}c++$

proc UserSet(in long A, long B out long C, long D)
$c++{
  C = A;
  D = B;
}c++$
```

The `proc f_mul` evaluates the Cartesian product between a row of `Matrix1` and a column of `Matrix2`, while `UserSet` simply copies two values. Indeed, the virtual processors section defines a parallel evaluation named `elab1` that will be activated as soon as `guard1` is ready. The expression `VP i, j` stands for $\forall i, j$, while the keyword `sync` imposes that each virtual processor will execute `f_mul` on the `parmod` state *updated* by the execution of `UserSet`[1]. Thus, the body of the parallel evaluation specify that all the virtual processors, compute the same block of operations, i.e.:

- they create a local copy of the values in position (i, j)
- they wait for the updates
- they execute the `proc f_mul` taking as input a row and a column and producing a long value as result
- they wait for the updates
- they produce the final result, i.e. the value in position (i, j)

Output section The output section collects all the results computed by the virtual processors activated by the guard, and builds the $N \times L$ resulting matrix.

```
output_section {
  collects ris from ALL Pv[i][j] {
    int elem;
    int Matrix_ris_[N][L];

    AST_FOR_EACH(elem) {
       Matrix_ris_[i][j]=elem;
    }
    assist_out(Matrix_ris, Matrix_ris_); }<>; } }
```

The collecting strategy is introduced by the `collects` keyword and is specified by the `from ALL` construct. This construct states that the output section waits for a new result `ris` from *all* virtual processors and the body is executed as soon as an incoming value is available on `Pv[i][j]`. In particular, the `AST_FOR_EACH` construct states that such incoming element (renamed `elem` in the scope of the current block),will be assigned to the resulting matrix in position (i, j).
Finally, the `assist_out` command allows to produce the output stream for the `parmod` Àt the end, the body of the `collects` statement could be followed by a sequence of control variable updates to enclosed between the angled brackets `<>`.

[1]The construct `sync` is not strictly a barrier among the virtual processor, rather a synchronization on the partitioned state

4.5 Data parallel with stencil

In this section we will detail how to implement the well known Floyd-Warshall algorithm for the evaluation of the shortest path for traversing a graph. The algorithm can be represented by the following piece of code:

```
for h to N do
  for i to N do
    for j to N do
      A_{i,j} = max(A_{i,h},A_{h,j})
```

As it can be seen, *for each* cycle the element in position (i, j) depends on the elements in position (i, h) and (h, j), creating strict dependencies between positions and among iterations as well. The problem can be easily represented by the following graph application:

```
// -*- C++ -*-
#define N 5
#define MAX_ITER 10
generic main() {
  stream long[N][N] A;
  stream long[N][N] B;
  generate (output_stream A);
  elab     (input_stream A output_stream B);
  print    (input_stream B);
}
```

The module `elab` encapsulates the parallel behavior of the application, while `generate` and `print` are sequential modules producing and consuming the stream values, respectively, as already discussed in Chapter 3. Since these last two modules do not offer particular new challenges, we will focus here only on the `parmod` code.

```
parmod elab (input_stream long A[N][N] output_stream long B[N][N]) {
  topology array [i:N][j:N] Pv;
  attribute long S[N][N] scatter S[*u0][*v0] onto Pv[u0][v0];
  stream long ris;

  do input_section {
    guard1: on , , A {
      distribution A[*u][*w] scatter to S[u][w];
    }
  } while (true)

  virtual_processors {
    elab (in guard1 out ris) {
      VP i,j {
        for (h=0; h<N; h++) {
          Felab (in S[i][h], S[h][j], S[i][j] out S[i][j]);
        };
        assist_out (ris, S[i][j]);
      }
    }
  }

  output_section {
    collects ris from ALL Pv[i][j] {
      long el;
      long B_[N][N];
```

```
      AST_FOR_EACH(el) {
        B_[i][j] = el;
      }
      assist_out (B, B_);
    }<>;
  }
}

proc Felab (in long a, long b, long s  out long S)
inc<"iostream">
$c++{
  long t = a+b;

  if (t < s)
    S = t;
  else
    S = s;
}c++$
```

The `parmod` takes an stream as input providing $N \times N$ matrices and produces a stream of resulting matrices.

The topology

```
  topology array [i:N][j:N] Pv;
```

The topology section defines the number of virtual processors onto which the elaboration will be eventually distributed. This implementation uses $N \times N$ virtual processors, one for each element $c_{i,j}$ we expect in the result matrix.
The topology type `array` allows to define $N \times N$ virtual processors as if they were elements of an array Pv. Thus, in the scope of the `parmod` each virtual processor will be indexed by `Pv[i,j]`, provided that $i \in [1, N]$ and $j \in [1, L]$.

The mapping of attributes Once we have given a topology to logically organize the set of virtual processors, we could need to partition one or more global variables that will collectively used by the virtual processors.

```
  attribute long S[N][N] scatter S[*u0][*v0] onto Pv[u0][v0];
  stream long ris;
```

The attribute represented by `S` is a bi-dimensional array, shared between all the virtual processors. The expression `scatter S[*u0][*v0] onto Pv[u0][v0]` means that ∀`u0,v0` S[u0][v0] is mapped onto the virtual processor `Pv[u0][v0]`. In other words, `Pv[u0][v0]` can access the whole matrix that becomes part of the `parmod` state.

Input section In the input section the behavior of the parmod with respect to the reception of the input values will be explicited.

```
  do input_section {
    guard1: on , , A {
      distribution A[*u][*w] scatter to S[u][w];
    }
  } while (true)
```

As seen in the previous example, the input section hosts only one guard, namely `guard1`, that will activated on the basis of a given default priority as soon as an element is available on the input stream `A`.

The distribution strategy The body of a guard is responsible for the distribution of the incoming items to the virtual processors and, as we have seen in Section 4.4, the following `distribution` construct

```
distribution A[*u][*w] scatter to S[u][w];
```

states that the matrix is fully distributed onto all the virtual processors.

Virtual processors section In this section the actual elaboration of the virtual processors is explicited.

```
virtual_processors {
  elab (in guard1 out ris) {
    VP i,j {
      for (h=0; h<N; h++) {
        Felab (in S[i][h], S[h][j], S[i][j] out S[i][j]);
      };
      assist_out (ris, S[i][j]);
    }
  }
 }
```

where `Felab` is a `proc` defined as follows:

```
proc Felab (in long a, long b, long s  out long S)
inc<"iostream">
$c++{
  long t = a+b;
  if (t < s) S = t;
    else S = s;
}c++$
```

The virtual processors section defines a parallel evaluation named `elab` that will be activated as soon as `guard1` is ready. The body of the parallel evaluation specify that all the virtual processors (`VP i, j` stands for $\forall i, j$), compute the same block of operations.
Such block includes the ASSIST-CL `for` construct through which the ASSIST run-time keeps the shared state (i.e. the attribute S) *consistent* at each iteration of the loop. Alternatively, ASSIST-CL offers a `while` construct for writing loops: in this case, the body of the `VP` block appears as follows:

```
do{
  Felab (in S[i][h], S[h][j], S[i][j] out S[i][j]);
  h = h +1 ;
}while (h<N)
```

Output section In this case, the output section is quite similar to the one already seen in Section 4.4: it collects the results waiting for the computation of *all* the virtual processors activated by the guard, and builds the $N \times N$ resulting matrix.

```
output_section {
  collects ris from ALL Pv[i][j] {
    long el;
      long B_[N][N];
      AST_FOR_EACH(el) {
        B_[i][j] = el;
      }
      assist_out (B, B_);
```

```
    }<>;
  }
}
```

4.6 Parmod with multiple guards

In this section we will explain how ASSIST-CL allows to program a not-deterministic activation of a set of guard inside a `parmod` module.

Let us suppose we want to encode a parmod depicted in Fig. 4.2 The virtual processors

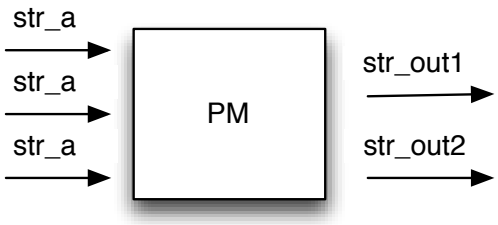

Figure 4.2: A parmod with multiple guards

belonging to the `parmod PM` execute a data-flow elaboration on the input data provided by the streams `str_a` and `str_b`. Not-deterministically, the `parmod` can also receive inputs from the stream `str_c`.

A possibly implementation of the section in charge to receive the input values is proposed below:

```
parmod PM(input_stream long str_a[N][N], long str_b, long str_c[N]
          output_stream long str_out1[N][N], long str_out2) {

  topology array [i:N][j:N] Pv;
  attribute long S[N][N] scatter S[*i0][*j0] onto Pv[i0][j0];
  attribute bool lg_str;
  attribute long p1, p2;

  init{
    lg_str = false;
    p1 =1 ;
    p2 = 0;
  }

  input_section {
    guard1: p1 , , str_a && str_b {
      distribution str_a[*j] scatter to Pv;
      distribution b broadcast to Pv;
      operation{
        lg_str = true;
      }
    }
    guard2: p2 , lg_str , str_c {
      long max;
      distribution A2[*k] scatter to S[k];
    }
  }

  // other section of the parmod
```

The `parmod` takes three parameters as input (one providing bi-dimensional matrices, one provided long values and one providing a uni-dimensional array) and produces two streams as output `str_out1` and `str_out2`.

The topology

```
topology array [i:N][j:N] Pv;
```

This type of topology allows to define $N \times N$ processors.

The mapping of attributes

```
attribute long S[N][N] scatter S[*i0][*j0] onto Pv[i0][j0];
attribute bool lg_str replicated;  // control variable
attribute long p1, p2 replicated;  // values of priorities
```

The attribute represented by `S` is a bi-dimensional array scattered onto the virtual processors as already seen in the previous example. The attributes `lg_str`, `p1` and `p2` are *replicated* on each virtual processor, as declared by the keyword `replicated`.

Initialization of control variables The construct `init{...}` allows to execute some initialization operations as, for instance, the one related to the control variables. In our case, the replicated attributes are initialized.

Input section The guards are listed in the `input_section{...}` block.

```
input_section {
   guard1: p1 , , str_a && str_b {
     distribution str_a[*i][*j] scatter to Pv;
     distribution str_b broadcast to Pv;
     operation{
       lg_str = true;
     }
   }
   guard2: p2 , lg_str , str_c {
     {
       proc_f(in str_c out max)
     }
     distribution max broadcast to Pv
   }
 }
```

As already seen in the previous example, the activation of a guard depends on:

1. the availability of *at least* one task on the input stream (or on the conjunction/disjunction of two or more of them)
2. a value of priority, given by a variable or a constant positive value.
3. a boolean expression, typically concerning the evaluation of one or more control variables

If two guards are ready (1 and 3 are true), the guard with highest priority is executed; otherwise one of the ready guards is selected by the run-time system on the basis of a policy unknown to the programmer.

The distribution strategy The body of a guard, is responsible for the distribution of the incoming items to the virtual processors. The keyword `distribution` introduces a strategy as, for example, the *broadcast* of the value `str_b` to all the virtual processor of `Pv`. Moreover, the construct

```
operation{
  lg_str = true;
}
```

can be used to update the control variable `lg_str` before leaving the guard block.

An alternative behavior is given by the body of `guard2` where a `proc` is simply invoked and the result of such elaboration is distributed onto the virtual processors.

Chapter 5

Assist-CL details

In this chapter we will detail the ASSIST-CL syntax, extending and completing the overview given in the previous chapter with respect to the coding of the `parmod`.

We recall that the structure of an ASSIST-CL program can be sketched as followed:

```
generic main(){

   // declaration of global variables
   // declaration of sequential modules
   // declaration of proc

   parmod parmod_name(input_stream <list of input stream>
                      output_stream <list of output stream>){
       // Definition section
       // Input section
       // Virtual processors section
       // Output section
   }
}
```

5.1 Definition section

In this section we have to define the *topology* of the virtual processors and the *guards*. We can also define *attributes* and other items. We see each one of them in details.

5.1.1 Topologies of the virtual processors

We can have three different *types* of topologies of virtual processors:

1. **one**: defines a `parmod` with just one virtual processor and is used to implement a sequential module with nondeterministic behaviour upon receptions on the input streams.
2. **none**: defines a `parmod` with anonymous virtual processors.
3. **array**: generates a naming for the virtual processors that can addressed using a numeric index.

Clearly, we will choose a topology depending on what kind of parallel algorithm we want to implement. In a subsequent section we will show how to implement classical skeletons and the associated topology needed. The topology influences the type of state we can define in a `parmod` and Tab. 5.1 summarizes such mapping. Moreover, section 5.1.2 gives more insights on the definition and kind of the state of a `parmod`.

Topology	Characteristics of the parmod's state allowed
one	state of the VP (trivial)
none	replicated state
array	replicated and partitioned state

Table 5.1: Features of the `parmod`'s state allowed for the three topologies of the virtual processors

The topology of the virtual processors is specified by the keyword `topology` followed by its *type*, as shown in the following example:

```
generic main(){

 parmod myFirstParmod (input_stream <list of input streams>
                       output_stream <list of output streams>) {

   // Definition section
      topology array [i:N][j:N] myVPs;
   // Input section
   // Virtual processors section
   // Output section
 }
}
```

In this case we are using two indexes, i and j, to name a virtual processor; so, taking the `parmod` in Fig. 4.1 as an example, we can write something like: `myVPs[0][1]` to name the VP in "position" (i, j). We can also have:

```
topology none otherVPs;
```

to define an anonymous virtual processor and

```
topology one Pv;
```

to define a singleton virtual processor, i.e. a nondeterministic sequential module.
In the following, we will see that the name we give to the set of the virtual processors (`Pv` in the latter example) is also used to address the virtual processors in the *distribution* and *collection* operations performed by the input and output section, respectively.

5.1.2 Attributes

Attributes are variables in the scope of the input section and to the virtual processors. They are used to define a state to the `parmod`, surviving its activations. There are three *types* of attributes:

- *replicated*: each virtual processor has its own copy of the attribute and it is the only one that can read and write it. In fact, there isn't a shared state. An example of definition of a replicated attribute is the following one, in which the boolean variable `diff` representes a *replicated* attribute:

  ```
  generic main(){
    parmod parmodWithReplicatedAtt (...) {
       // Definition section
       topology none myVP;
       attribute bool diff replicated;

       // other sections
    }
  }
  ```

- *partitioned*: attributes of this type are meant to be spreaded over the virtual processors according to their naming and are introduced by the keyword `scatter`. The access to partitioned attributes is ruled by the *Owner-Compute* rule stating that, since an attribute's element has an owner, that owner is the only one that can read *and* write it; other non-owner virtual processors can just read that element. As we saw in the topology description, we can define partitioned attributes only if the `parmod` has the array topology. An example of the definition of a partitioned attribute is the following one:

```
parmod parmodWithScatteredAtt (...) {
   // Definition section
   topology array [i:N][j:N] Pv; attribute long S[N][N] scatter
   S[*i0][*j0] onto Pv[i0][j0];

   // other sections
}
```

  In this example the attribute `S` is a bidimensional $N \times N$ array that is scattered onto the virtual processors according to an isomorphism: the processor named `Pv[i][j]` is the owner of `S[i][j]`. The variables `*i0` and `*j0` that appear in the code are free and must be fresh. Clearly there is a mapping between the free variables and the variables used in the indexing of the `Pv` (see Sec. A.5 for details about a correct usage of indexes)

  Finally, notice that the name we give to the set of virtual processors (i.e. `Pv`) is used to address them in the definition of the attributes.

- attributes in topology `one`: this is a special case relative to the `one` topology. Clearly, there is not need of a replicated or partitioned attribute because we have just one virtual processor but attributes of can be declared and used as well resulting in a variable of common use.

```
parmod parmodOneWithAtt (...) {
   // Definition section
     topology one Pv;
     attribute long S onto Pv;

    // other sections
 }
```

5.1.3 Control variables

Control variables represent values are shared between the input and ouput sections in order to implement some kind of control on the reception and sending of data items from and to the external world. Another use of these variables consists in expressing an implicit *loop* on the streams. They're declared exactly as an attribute (without the `replicated` or `partitioned` properties) and we will see how to implement these behaviours in details once we've introduced the input section.

5.1.4 Internal streams

Internal streams are used to allow an explicit communication between the virtual processors and the output section. They are introduced by the keyword `stream` and are defined as we can see in this example:

```
parmod myFirstInternalStream (...) {
  topology none Pv;
  stream long result;
```

```
  // other code...
}
```

Internal streams are introduced in order to solve some implementation problems: suppose that we are dealing with the implementation of a function but the type of the returned value isn't the one you'd like to get. For example, the function returns a *long* but you want your `parmod` to send a collection of the returned values as a bidimensional array (namely `long S[N][N]`). We can avoid this problem by defining an internal stream of type `stream long result` that carries the value returned by the function (invoked multiple times) to the output section that collects a bunch of them and sends the collected results on an output stream. We will see an example of this implementation in a following section.

5.2 Input section

In this section we will show how to program the different behavior of items reception from the input streams, the distribution of the items to the virtual processors and the refreshing of the control variables at each activation. The input section is declared as:

```
generic main(){

  parmod myFirstParmod (...) {
    topology none Pv;
    // Input section
    init{
      // initializations
    }

    input_section {
       // actions....
    }
    // other sections
  }
}
```

In the next subparagraphs we describe all the items that can be specified in the input section.

5.2.1 Initialization

When the `parmod` starts we need to initialize control and state variables. This is done using the **init** statement, as we saw above for the initialization of the control variable *count*. The expression braced by the **init** is evaluated when the `parmod` starts its execution. Below we show an example of the initialization of a replicated attribute and a control variable:

```
  parmod parmodInitializationExample (...) {
    topology array [i:N] my_vp;
    attribute bool diff replicated;
    init {
      VP i {
        init_false(out diff);
      }
    }
    // other sections
  }

  proc init_false(out bool b)
  $c++{
```

```
  b = false;
}c++$
```

In this example the replicated variable `diff` is initialized to *false* in all the virtual processors. Notice that the **init** statement isn't inserted into the **input_section** statement but, conceptually, it belongs to the initialization phase. Notice also that in the **init** statement the virtual processors are addressed by means of the keyword **VP** and not by the name we've introduced (i.e. `my_vp`) because the latter name is used just for distribution and collection of data items in the input and output section.

5.2.2 Guards

Suppose that a `parmod` has multiple input streams and that it has received a new data item from a subset of them; suppose also that we want to receive from just one of them in a nondeterministic fashion. The nondeterminism can be programmed by a proper definition of the *guards* in the input section. A guard is defined by a *unique identifier* (in the scope of each `parmod`) and from the following optional parameters:

- *priority*: if two guards are activated we choose the one with the highest value of this field. If we don't need an explicit priority we'll use the **on** keyword.
- *condition*: a guard is activated only if this expression is evaluated to *true*.
- *expression on the streams*: a guard can be related to an input stream; in this case the guard is activated if the condition on the input stream (given by this expression) is evaluated to *true*. The expression A, where A is the name of an input stream, in this context is semantically equivalent to: "if a new value has been received from A".

The following example shows how options work together:

```
1  parmod myFirstGuards(input_stream long A[N][N],
2  long long B[M][M]) {
3    topology none Pv;
4    attribute bool matrix_ready;
5    input_section {
6      guard1: on, !matrix_ready, A {
7        // body of the guard
8      }
9      guard2: on, matrix_ready, B {
10       // body of the guard
11     }
12   }
13   // other sections
14 }
```

The guard named `guard1` (line 6) is activated when the variable `matrix_ready` is evaluated to *false*[1] and a new data has been received from the input stream named `A`. Otherwise we receive from the input stream named `B`, evaluating the guard named `guard2` (line 9). In this example the priority option isn't used.
In the case we don't have or want a control variable we can write:

```
parmod mySecondGuards(input_stream long A[N][N],
                      long long B[M][M]) {
  topology none Pv;
  input_section {
    guard1: on, , A {
```

[1] the ! negates the subsequent expression, as in the *C++* style.

```
      // body of the guard...
    }
    guard2: on, , B {
      // body of the guard...
    }
  }
  // other sections
}
```

and, as obvious, we choose the first incoming data on `A` or `B` in a nondeterministic fashion.

On the other hand, two control variables can be evaluated as a conjunction of values: as an example, if in line 6 appears

```
guard1:  on, !matrix_ready, A & B
```

then `guard1` will be evaluated when `matrix_ready` is evaluated to *false* and a new data value has been received from the input stream `A` *and* from the input stream `B`.

5.2.3 Mapping strategies

As we saw in the previous subsections when a new data item is received on an input stream we have to map it to the virtual processors in order to make some computation. In the input section we define the distribution policy, that can be:

- *scatter*: the data item is scattered among the virtual processors according to the scatter rule specified, i.e. it is partitioned and sent to the virtual processors.
- *broadcast*: the data item is sent to all the virtual processors.
- *on_demand*: the data item is sent to the first free virtual processor, where a virtual processor is free if it isn't performing any computation.
- *scheduled*: a virtual processor is directly addressed according to the rule specified. If the virtual processor isn't free, the distribution (and all the input section) is blocked[2].

Below, we show some examples of distribution policies:

- **scatter** isomorph on one attribute:

```
1   parmod isomorphAttScatter(input_stream long A[N][N]) {
2     topology array [i:N][j:N] myVP;
3     attribute long S[N][N];
4     input_section {
5        guard1: on, , A {
6          distribution A[*u][*w] scatter to S[u][w];
7        }
8     }
9     // other sections
10  }
```

  In this example the new data received on the input stream named `A` is scattered to the attribute named `S` (line 6). After the distribution we'll have: $\forall i, j.$`S[i][j]` $=$ `A[i][j]`.

- **scatter** isomorph on the virtual processors:

```
1   parmod isomorphVPScatter(input_stream long A[N][N]) {
2     topology array [i:N][j:N] myVP;
3     input_section {
```

[2]For this kind of distribution there're some constraints: see the topology examples section to for a list of them.

```
4         guard1: on, , A {
5           distribution A[*u][*w] scatter to myVP[u][w];
6         }
7       }
8       // other sections
9     }
```

In this example the new data received on the input stream named A is scattered to the virtual processors (line 5); Clearly the virtual processors section will have to deal with the reception of the corresponding element of the array.

- **scatter** by rows:

```
1   parmod scatterByRow(input_stream long A[N][N]) {
2     topology array [i:N] myVPs;
3       input_section {
4         guard1: on, , A {
5           distribution A[*k0][] scatter to myVP[k0];
6         }
7       }
8       // other sections
9   }
```

In this case the bidimensional array received is scattered by rows on the virtual processors, i.e. the i-th row is all given to the i-th virtual processor. Notice that the dimension of the bidimensional array (i.e. `N`) and the number of virtual processors must be equal (lines 1 and 2): otherwise we'd have a run time error.

- **broadcast**:

```
1     parmod broadcastExample(input_stream long value) {
2       topology none myVPs;
3       input_section {
4         guard1: on, , value {
5           distribution value broadcast to myVPs;
6         }
7       }
8       // other sections
9     }
```

Here the `value` received is sent to all the virtual processors (line 5).

- **on_demand**: if the computations performed by the virtual processors are independent we can choose to implement a reservation protocol; so a virtual processor requests the input section for a new input data, processes it (eventually sends results to the output section) and then returns to request again for a new data. This is done automatically in ASSIST as we can see in the next example:

```
parmod onDemandExample(input_stream double value) {
   topology none myVPs;
   input_section {
      guard1: on, , value {
        distribution value on_demand to myVPs;
      }
    }
    // other sections
  }
```

- constant **scheduled**: let's see an example:

```
parmod constantScheduled(input_stream double A, ...) {
  topology array VPs;
  input_section {
    guard1: on, , A {
      distribution A scheduled to VPs[3];
    }
  }
   // other sections
}
```

Whenever the guard named `guard1` is activated (by the reception on the input stream named `A`) the new value is sent to the virtual processor indexed with `3`. If it is not ready, the input section is blocked.

5.2.4 Updating the control variables

Control variables can be refreshed inside the **operation** statement that will be executed after the activation of a guard. For example:

```
1  parmod operationUsageExample(input_stream double A[N], ...) {
2    topology array VPs;
3    attribute bool matrixReady;
4    input_section {
5      guard1: on, , A {
6         // body of the guard
7      }
8      operation {
9        matrixReady = true;
10     }<use matrixReady>;
11      // other guards
12   }
13   // other sections
14 }
```

As we can see from this example the **operation** statement must be tagged by the **use** directive (line 10) in order to inform the ASSIST-CL compiler which control variables are used in the statement. Moreover the **operation** statement is coded in *C++* and must terminate in order to allow a correct behaviour.
Now we're ready to see some more examples regarding the *scheduled* distribution without constant addressing:

- variable **scheduled**:

```
1    typedef struct {
2      ....
3      int idx;
4    } myStruct;
5
6    parmod variableScheduled(input_stream myStruct A, ...) {
7      topology array [i:N] VPs;
8      attribute long round;
9      input_section {
10       guard1: on, , A {
11         distribution A scheduled to VPs[*destination];
12       }
```

```
13          operation {
14            destination = (A.idx+round)%7;
15          }<use round>;
16        }
17        // other sections
18      }
```

In this example the virtual processor addressed at each distribution varies according to the rule at line 14: in fact the sender of the variable A can control (by means of the `idx` field) the addressing of the virtual processors.

- **scheduled** in the `none` topology: also if the virtual processors in the `none` topology haven't an explicit naming we can, for example, address one of them in the subsequent way:

```
parmod constantScheduled(input_stream myStruct A, ...) {
   topology none VPs;
   attribute long round;
   input_section {
      guard1: on, , A {
        distribution A scheduled to index;
      }
      operation {
        index = (A.idx+round)%7;
      }<use round>;
    }
    // other sections
    }
```

Finally recall that, when we described the control variables, we told that there is a way of expressing an implicit cycle of the reception upon the input streams. Now we can see an example:

```
1   parmod cycledInputSection(input_stream double A[N], ...) {
2     topology none Pv;
3     attribute long count;
4     init {
5       count = 0;
6     }
7
8     do input_section {
9       guard1: on, , A {
10        distribution A on_demand to Pv;
11      }
12
13      operation {
14        count++;
15      }<use count>;
16    } while (count < 10)
17    // other sections
18  }
```

The reception and distribution of the data items received upon the stream named `A` is performed until the variable `count` reaches the value of `10` or the input stream is closed by the sender. This behaviour introduce us to the next subparagraph.

5.2.5 Termination condition

Finally let's see how a `parmod` can terminate without errors. The implicit way is determined by the closing of all the input streams; for example:

```
1  parmod implicitTermination(input_stream double A[N], ...) {
2    topology none Pv;
3    attribute long count;
4    init {
5      count = 0;
6    }
7
8    do input_section {
9      guard1: on, , A {
10       distribution A on_demand to Pv;
11     }
12   }while (true)
13   // other sections
14 }
```

The `parmod` will terminate when the input stream `A` is closed by the sender. If more than one guard control the activation, the `parmod` terminates when all the associated streams has been closed.
A `parmod` can be terminated also explicitly by using conditions on the control variables. In the example in Section 5.2.4 the `parmod` terminates when the input stream `A` is closed or when the `count` variable reach the value `10`. In the second case, if the input stream `A` still contains data items, we've a run time error; so the programmer has to deal with this kinds of events by coordinating conditions on control variables and communications over streams.

5.3 Virtual processors section

In this section we've to define the behaviour of the virtual processors. This behaviour can be implemented by a call (or a list of calls) to proc and can differ depending on the naming of the virtual processors. The virtual processors section is declared as:

```
parmod myFirstVPSection(input_stream double A[N], ...) {
  topology none Pv;
  input_section {
     // body of the input section
  }

  virtual_processors {
    // VPs behaviour
  }
  // other sections
}
```

From an implementation point of view, in the virtual processors section we define the mapping of the input streams to the proc used to perform the computation. For example:

```
1  parmod myFirstVPImplementation(input_stream long A[N][N], ...) {
2    topology array [i:N][j:N] Pv;
3    attribute long S[N][N];
4    do input_section {
5      guard1: on, , A {
6        distribution A[*u][*w] scatter to S[u][w];
```

```
7       }
8     } while(true)
9
10    virtual_processors {
11      compute1(in guard1 out result) {
12        VP i,j {
13          // VP behaviour
14        }
15      }
16    }
17    // other sections
18  }
```

The name `compute1` (line 11) is a variable chosen by the programmer; important are the declarations that follow that name: **in** `guard1` means that this computation is activated by the activation of the guard named `guard1`. So when the `parmod` receives a new value on the input stream named `A` the computation named `compute1` is activated. Also, in the `compute1` statement the variable `A`, i.e. the value received on the input stream, is visible. The keyword **VP** (line 12) is used to address virtual processors: in the previous example it had to be followed by two indexes (i.e. `i` and `j`) because we've chosen an array topology. The following example shows how different virtual processors can be addresses depending on their indexes:

```
1   proc proc_border(in long i, long j out long result) {
2     // compute on the border elements
3   }
4
5   proc proc_internal(in long i, long j out long result) {
6    // compute on the inner elements
7   }
8
9   parmod mySecondVPImplementation(input_stream long A[N][N], ...) {
10    topology array [i:N][j:N] Pv;
11    attribute long S[N][N];
12    do input_section {
13      guard1: on, , A {
14        distribution A[*u][*w] scatter to S[u][w];
15      }
16    } while(true)
17
18    virtual_processors {
19      compute2(in guard1 out result) {
20        VP i = 0..0, j = 0..N-1 {
21          proc\_border(in i,j out S[i][j]);
22        }
23        VP i = N-1..N-1, j = 0..N-1 {
24          proc_border(in i,j out S[i][j]);
25        }
26        VP i = 1..N-2, j = 0..0 {
27          proc_border(in i,j out S[i][j]);
28        }
29        VP i = 1..N-2, j = N-1..N-1 {
30          proc_border(in i,j out S[i][j]);
31        }
32        VP i = 1..N-2, j = 1..N-2  {
33          proc_internal(in i,j out S[i][j]);
34        }
```

```
35        }
36      }
37      // other sections
38    }
```

In this example virtual processors are disposed in a bidimensional array topology: those on the four borders of the array will execute the proc named `proc_border` (lines 20-31); the others will execute the `proc_internal` (lines 32-34). Still in the array topology, if we want to address all the virtual processors, it is sufficient to write:

```
parmod myFirstVPSection(input_stream double A[N], ...) {
  topology array [i:N][j:N] Pv;
  input_section {
    // boby of the input section
  }

  virtual_processors {
    VP i,j {
      matrixCompute(in S[i][j] output_stream result);
    }
  }
  // other sections
}
```

Now, clearly, `matrixCompute` is the name of a proc defined somewhere in the code.
The addressing is simpler w.r.t the previous one, when we've a `none` or a `one` topology:

```
parmod VPAddressing(input_stream double A[N], ...) {
  topology none Pv;
  input_section {
    // boby of the input section
  }
  virtual_processors {
    VP {
       // VPs behaviours
    }
  }
  // other sections
}
```

Finally we introduce, as for the input section, the iteration control; ASSIST-CL provides two construct: **for** and **while**. An example of usage of the **for** command is:

```
1   parmod myFirstForConstructor(input_stream double A[N], ...) {
2     topology array [i:N][j:N] Pvs;
3     attribute long S[N][N];
4     stream long result;
5     input_section {
6       // boby of the input section
7     }
8
9     virtual_processors {
10      VP i,j {
11        for (h = 0; h < N; h++) {
12          Felab(in S[i][h], S[h][i], S[i][j] out S[i][j]);
13        };
```

```
14          assist_out(result, S[i][j]);
15        }
16      }
17      // other sections
18    }
```

The ASSIST run-time keeps the shared state (i.e. the attribute `S`) *consistent* at each iteration of the loop (lines 11-13). This is done in order to allow all the virtual processors to read a consistent value for their neighbours elements.
We can write the same code by using the `while` command:

```
1   parmod myFirstForConstructor(input_stream double A[N], ...) {
2     topology array [i:N][j:N] Pvs;
3     attribute long S[N][N];
4     stream long result;
5     input_section {
6       // boby of the input section
7     }
8
9     virtual_processors {
10      VP i,j {
11        do {
12          Felab(in S[i][h], S[h][i], S[i][j] out S[i][j]);
13          h = h +1;
14        }while(h<N);
15        assist_out(result, S[i][j]);
16      }
17    }
18    // other sections
19  }
```

Here we use the classical **reduce** function (line 13) to test a convergence condition on the `diff` value computed by all the results of the evaluation of the `compute` proc in each virtual processor. The associative operator used in the reducing operation is, here, the *or* operator (as in *C* defined by the symbol ||).

5.4 Output section

In this section we implement the collection of the results from the virtual processors. The section is declared as:

```
parmod myFirstOutputSection(input_stream double A[N], ...) {
  topology array [i:N][j:N] Pvs;
  input_section {...}

  virtual_processors {...}

  output_section {
    // boby of the output section
  }
}
```

There are two kinds of policies for the collection of the result values:

- **ANY**: the output section waits for the first result sent from any virtual processor (in a nondeterministic fashion).

- **ALL**: the output section waits for a new result from all the virtual processors.

Also, in the output section we can insert some code that represents a kind of final computation on the results collected. As example:

```
1  parmod outputSectionImplemented(input_stream double A[N], ...) {
2    topology array [i:N][j:N] Pvs;
3    stream long result;
4    input_section {
5      // boby of the input section
6    }
7
8    virtual_processors {
9      // boby of the section
10   }
11
12   output_section {
13     collects result from ALL Pvs[i][j] {
14       int el;
15       int B_[N][N];
16       AST_FOR_EACH(el) {
17         B_[i][j] = el;
18       }
19       assist_out(B, B_);
20     }<>
21   }
22 }
```

In this example we wait for all results returned by the virtual processors and, by means of the construct **AST_FOR_EACH** (line 16), we iterate a command (the assignment at line 17) on each result returned. Clearly the value received upon the internal stream, named *result*, is bound to the parameter *el* of the **AST_FOR_EACH** construct. Notice that the **collects** statement is coded in *C++*, so we'd have to pay attention to stack overflows, as usual in *C++* (eventually, by using dynamic memory allocation). Moreover, if we don't use control variables inside the **collects** statement we will leave empty the <> directive. We can also avoid specific computations in the output section as shown in the following example:

```
1  parmod outputSectionANY(... output_stream long C) {
2    topology none Pvs;
3    input_section {
4      // boby of the input section
5    }
6
7    virtual_processors {
8      elab(... output_stream long C)
9      // boby of the section
10   }
11
12   output_section {
13     collects C from ANY Pvs;
14   }
15 }
```

In this example each value sent by any virtual processor is collected and sent directly to the output stream (line 13).

Appendix

Appendix A

Appendix

A.1 Pure sequential module

There are two ways to code the body of a module expressing a sequential behavior: the first modality is described in Sec. 3.3 and we will described in this section how to implement a so called *pure* sequential module. Such a method consists in directly inserting the host language code inside the module definition.
The body is specified between brackets and the brackets are labeled by the modifier $ coupled with the host language specification string. The body may be written in any of the host language supported (currently, C, C++ and FORTRAN).

```
1 generic main(){
2
3   directlyImplSeqMod(input_stream long lin, char cin
4               output_stream long lout, char cout)
5       $c{
6          lout = lin + 1;
7          cout = cin;
8       }c$
9 }
```

As instance, the `directlyImplSeqMod` is a pure sequential module implementing a C function. At the end of the evaluation of the expression between the brackets, the results are implicitly sent onto all the output streams. So, the example above operationally corresponds to: at each activation, wait for a new data on `lin` and `cin` streams; evaluate the two expressions at the lines 4 and 5; send the new values of `lout` and `cout` on the output streams.

A.2 Assist Preprocessor Substitution

Be careful with the usage of macros within sequential code and strings. For instance, if we declare a string like

```
char s[50] = ‘‘this is just a N test’’;
```

and we print it to the screen, the result will be:

```
this is just a 10 test;
```

A.3 ASSIST Preprocessor and Language Markers

When looking for language end markers like `}c++$`, `}c$` the ASSIST preprocessor is not aware of the sequential sintax, and also looks within strings (see Sec. A.2). Thus

1. unbalanced open/closed parenthesis will not be checked and will produce errors in the sequential compilation step (this is coherent with ASSIST assumption);
2. a language marker within a string will be recognized (this is a bug), and ASSIST compilation will fail, trying to parse the remaining sequential code as ASSIST-CL code.

A.4 Input parameters to C code

For implementation reasons, parameter passed to c code `$c{ ...}c$` have to be used as pointer within the code; e.g. a stream `B` of long will be assigned to a long variable `Y` this way : `Y = *B;`

The same happens for more complex types like arrays.

A.5 Isomorphism between indexes

The ASSIST run-time only allows to distribute `Pv` in a prefix isomorphism (line 4) and that's why we could write

```
scatter S[*i0][*j0] onto Pv[i0][j0];
```

or

```
scatter S[*i0][*j0] onto Pv[i0][];
```

but not

`scatter S[*i0][*j0] onto Pv[j0][i0]` (line 4)